Spis treści

1 Wstęp

Celem tych notatek jest zwięzłe przedstawienie kompletu zagadnień związanych z szeroko pojętym uczeniem głębokim jako podejściem do Sztucznej Inteligencji (SI). Zaczynamy od minimalnego zbioru wymaganych tematów z zakresu rachunku prawdopodobieństwa i statystyki matematycznej. Następnie opisujemy podstawowe metody uczenia maszynowego z probabilistycznego punktu widzenia. W końcu przechodzimy do zasadniczej części związanej z uczeniem głębokim i sieciami neuronowymi. W każdej części staramy się przedstawiać opisywane tematy w sposób minimalistyczny, skupiając się głównie na matematycznej i ideowej, a nie implementacyjnej stronie zagadnień. Liczymy, iż takie podejście zapewni odpowiednio głębokie zrozumienie tematu, dzięki któremu dalsze studiowanie całej gamy specyficznych technicznych tematów nie sprawi żadnego problemu.

1.1 Notacja

W dalszej części tekstu będziemy stosować przedstawioną tutaj pokrótce notację. Wektory, które traktujemy jako elementy przestrzeni $\mathbb{R}^d$ ze standardowo zdefiniowanymi operacjami dodawania i mnożenia przez skalar będziemy oznaczali wytłuszczonymi małymi lub wielkimi literami np. $\boldsymbol{x}, \boldsymbol{X}$. Wielkość $\boldsymbol{x}^i$ będzie oznaczać dany element wektora (w tym przypadku i–ty element $\boldsymbol{x}$). Wielkość $\boldsymbol{x}_\mu$ będzie oznaczać pewien (w tym przypadku μ–ty) element pewnego zbioru wektorów. Macierze oraz wielowymiarowe tablice (zwane również niefortunnie tensorami) będziemy oznaczać (jedynie) wytłuszczonymi wielkimi literami np. $\boldsymbol{X}, \boldsymbol{\Phi}$. Analogicznie jak w przypadku wektorów przez $\boldsymbol{X}^{i_1 i_2 \cdots i_k}$ będziemy oznaczać $(i_1, i_2, \ldots, i_k)$ element k–wymiarowej tablicy $\boldsymbol{X}$, natomiast $\boldsymbol{X}_\mu$ będzie oznaczać μ–ty element pewnego zbioru tablic.

1.2 Uczenie nadzorowane

Uczenie nadzorowane jest jednym z dwóch podstawowych (pomijając tzw. uczenie ze wzmocnieniem) paradygmatów w uczeniu maszynowym, którego ogólną ideą jest zdefiniowanie pewnego modelu odwzorowującego dane wejściowe na wyjściowe predykcje. Zakładamy w nim, iż mamy dostępny zbiór obserwacji $\mathcal{X} = \{y_i(\boldsymbol{x}_i)\}_{i=1}^n$, gdzie $\boldsymbol{x} \in \mathbb{R}^m$ nazywamy wektorem cech a $y(\boldsymbol{x})$ jest prawidłową wartością odpowiedzi dla tych cech. Dwa najbardziej podstawowe przypadki zagadnień tego rodzaju to regresja oraz klasyfikacja. W przypadku regresji zmienna y przyjmuje wartości z pewnego podzbioru

liczb rzeczywistych. W przypadku klasyfikacji zmienna y przyjmuje wartości ze skończonego zbioru kategorii, przy czym wartości z tego zbioru nie powinny posiadać naturalnej tj. wynikającej z natury problemu, relacji porządku.

W jaki sposób tworzymy model odwzorowujący $\boldsymbol{x}$ na y? W dalszych paragrafach poznamy różne metody, ale najczęściej modelem jest pewna rodzina funkcji postaci $\phi(\boldsymbol{x}; \boldsymbol{w})$ parametryzowana skończoną liczbą parametrów, które możemy łącznie zapisać jako pewien wektor $\boldsymbol{w}$. Aby znaleźć parametry $\boldsymbol{w}$, dzięki którym dla konkretnego zagadnienia model będzie zadowalająco odwzorowywał cechy na predykcje (innymi słowy aby nauczyć model) wprowadzamy dodatkowo funkcjonał kosztu (z ang. *loss function*) $L(\mathcal{X}; \boldsymbol{w})$, który kwantyfikuje odpowiedzi modelu ϕ w stosunku do znanych prawidłowych odpowiedzi y dla danych ze zbioru $\mathcal{X}$. Najczęściej ma on postać

$$L(\mathcal{X}; \boldsymbol{w}) = -\frac{1}{n} \sum_{i=1}^{n} \log p\left(y_i(\boldsymbol{x}_i) \mid \boldsymbol{w}\right) ,$$

gdzie $p(y(\boldsymbol{x}) \mid \boldsymbol{w})$ jest warunkową gęstością prawdopodobieństwa danej obserwacji $y(\boldsymbol{x})$ warunkowaną przez wartość parametrów $\boldsymbol{w}$ (oczywiście gęstość ta zależy od estymowanych parametrów $\boldsymbol{w}$ przez funkcję $\phi(\boldsymbol{x}; \boldsymbol{w})$. Do powyższego funkcjonału możemy również dodawać tzw. człony regularyzujące (z ang. *regularizers*). Trening modelu polega wówczas na znalezieniu parametrów $\boldsymbol{w}^*$, które minimalizują funkcjonał kosztu na zbiorze treningowym $\mathcal{X}$

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} L(\mathcal{X}; \boldsymbol{w}) .$$

Zauważmy, że takie podejście ma jedną zasadniczą wadę – istotnie nie interesuje nas tak naprawdę, jak model radzi sobie na zbiorze treningowym (tzn. zagadnienie uczenia jest czymś więcej niż numeryczną minimalizacją funkcji), tylko jak będzie radził sobie na nowych, niewidzianych wcześniej danych (zależy nam przede wszystkim na generalizacji). Sytuację, w której model bardzo dobrze modeluje dane w zbiorze treningowym, ale słabo radzi sobie na nowych danych nazywamy przeuczeniem lub nadmiernym dopasowaniem (z ang. *overfitting*). Sytuację, w której model słabo radzi sobie zarówno na zbiorze treningowym, jak i na nowych danych nazywamy niedouczeniem lub niedopasowaniem (z ang. *underfitting*). Występowanie overfittingu i underfittingu jest powiązane z pojemnością (z ang. *capacity*) modelu. Złożony model o dużej pojemności potrafi dopasować się do bardzo skomplikowanych obserwacji (jest elastyczny), ale istnieje ryzyko jego przeuczenia (mówimy wówczas o *high variance*). Dla prostego modelu o małej

pojemności istnieje z kolei ryzyko, iż nie ma on wystarczająco ekspresywności (mówimy wówczas o *high bias*).

1.3 Uczenie nienadzorowane

W przypadku uczenia nienadzorowanego naszym celem nie jest znalezienie modelu odwzorowującego cechy na predykcje. Chcemy raczej zrozumieć wewnętrzną strukturę danych oraz odkryć zależności między zmiennymi lub grupami zmiennych. Modele tego rodzaju znajdują zastosowanie w analizie biznesowej, gdzie pozwalają, chociażby na analizę ważności poszczególnych wskaźników, czy wizualizację wysoko-wymiarowych danych.

1.4 Praktyka uczenia maszynowego

W dalszej części skupiamy się przede wszystkim na matematycznej stronie prezentowanych zagadnień, ale należy pamiętać, iż dowolną próbę wdrożenia modelu uczenia maszynowego należy zacząć od dokładnej inspekcji danych, dla których przygotowujemy ów model („become one with the data", A. Karpathy), a zakończyć dogłębną analizą metryk pozwalających na ewaluację wytrenowanego modelu. Warunkiem koniecznym udanego wdrożenia modelu jest więc odpowiednie zebranie, analiza i przygotowanie danych, które trafiają następnie jako wejście do modelu ML, a następnie odpowiedni dobór i dogłębna analiza wyników ewaluacji modelu. W tym paragrafie pokrótce opisujemy elementarne praktyki, o których należy pamiętać przy wdrażaniu modeli ML.

1.4.1 Przygotowanie danych

Kluczem do uzyskania dobrych wyników przy korzystaniu z algorytmów uczenia maszynowego jest odpowiednie przygotowanie danych (z ang. *preprocessing*). Typowo preprocessing składa się z:

- eksploracji danych oraz wstępnego czyszczenia, w szczególności usunięcia jawnych wartości odstających (z ang. *outliers*) oraz cech posiadających zbyt dużo wartości brakujących;
- analizy rozkładu zmiennej docelowej oraz ewentualnej transformacji logarytmicznej, która poprawia stabilność numeryczną, gdy przewidywane wartości są dużymi dodatnimi liczbami rzeczywistymi, zmienia dziedzinę zmiennej objaśnianej z $\mathbb{R}_+$ na $\mathbb{R}$ oraz dodatkowo jest przykładem transformacji stabilizującej wariancję;

- *podziału zbioru na część treningową oraz testową*;
- dokonania skalowania i imputacji brakujących wartości cech (metody `.fit()` wywołujemy jedynie dla zbioru treningowego);
- usunięcia silnie skorelowanych cech;
- zakodowania wartości kategorycznych za pomocą tzw. *one–hot encoding* pamiętając o *dummy variable trap* – jedną z k kategorii kodujemy za pomocą wektora *one–hot* długości $n-1$, aby uniknąć zależności liniowej między cechami (opcja `drop="first"` w `OneHotEncoder` w scikit-learn);
- wykonania feature engineering – dodania wielomianów cech do naszych danych lub skonstruowania innych cech (np. cech określających miesiąc, dzień itp.).

Podział zbioru na część treningową i testową jest najważniejszym etapem preprocessingu. Zbiór testowy wydzielamy, aby po wytrenowaniu modelu sprawdzić, jak poradzi on sobie na nowych, niewidzianych wcześniej danych. Powinniśmy go traktować jako dane, które będziemy w przyszłości dostawać po wdrożeniu modelu do realnego systemu. Takie dane również będziemy musieli przeskalować, zakodować itp., ale parametry potrzebne do wykonania tych transformacji możemy wziąć jedynie z dostępnego wcześniej zbioru treningowego. Wykorzystanie danych testowych w procesie treningu to *błąd wycieku danych* (z ang. *data leakage*). Skutkuje on niepoprawnym, nadmiernie optymistycznym oszacowaniem jakości modelu.

1.4.2 Metryki do oceny regresji i klasyfikacji

Zasadniczo, aby ocenić predykcje modelu używamy odpowiednich metryk, których wartości określają jak dobry jest model.

W przypadku regresji najczęściej używanymi metrykami są RMSE (z ang. *Root Mean Squared Error*) oraz MAE (z ang. *Mean Absolute Error*) zdefiniowane odpowiednio jako

$$\mathrm{RMSE} := \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}\,, \quad \mathrm{MAE} := \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|\,.$$

Metryki te mają jednakową jednostkę jak predykcje. Jeśli chcielibyśmy mieć liczbę względną określającą jakość modelu to mamy do dyspozycji metryki

MAPE (z ang. *Mean Absolute Percentage Error*) oraz SMAPE (z ang. *Symmetric Mean Absolute Percentage Error*) zdefiniowane odpowiednio jako

$$\text{MAPE} := \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right|, \quad \text{SMAPE} := \frac{1}{n}\sum_{i=1}^{n}\frac{|y_i - \hat{y}_i|}{|y_i| + |\hat{y}_i|}.$$

Obie te metryki mają zakres od 0 do 1, przy czym niższa wartość oznacza lepszy model. Metryki te mają jednak szereg problemów, z których najpoważniejsze to: problemy, gdy wartości są bliskie 0, asymetryczne traktowanie predykcji za dużych oraz za małych. Z tych powodów znacznie lepszą względną metryką jest MASE (z ang. *Mean Absolute Scaled Error*)

$$\text{MASE} := \frac{\sum_{i=1}^{n}|y_i - \hat{y}_i|}{\sum_{i=1}^{n}|y_i - \overline{y}|},$$

gdzie $\overline{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$. Metryka MASE jest zatem względnym błędem MAE jaki popełnia nasz model w stosunku do modelu naiwnego, który przewiduje zawsze wartość średnią.

W przypadku zadania klasyfikacji binarnej naszym celem dla danego wektora cech jest zwrócenie jednej z dwóch klas, które będziemy nazywać klasą pozytywną i negatywną. O ile w przypadku regresji pomiar jakości modelu był całkiem prosty, o tyle w przypadku klasyfikacji sytuacja jest nieco bardziej skomplikowana. Zauważmy bowiem, iż mamy 4 możliwości odpowiedzi klasyfikatora

- *True Positive (TP)* – poprawnie zaklasyfikowaliśmy klasę pozytywną jako pozytywną
- *True Negative (TN)* – poprawnie zaklasyfikowaliśmy klasę negatywną jako negatywną
- *False Positive (FP)* – niepoprawnie zaklasyfikowaliśmy klasę negatywną jako pozytywną
- *False Negative (FN)* – niepoprawnie zaklasyfikowaliśmy klasę pozytywną jako negatywną

Na podstawie ilości TP, TN, FP i FN w zbiorze testowym możemy wykreślić tzw. *macierz pomyłek* (z ang. *confusion matrix*) pokazującą ilość każdej z możliwości. Następnie możemy obliczyć różne stosunki tych wartości, aby uzyskać różne metryki. Najbardziej standardowymi są *accuracy*, *precision* oraz *recall* (lub inaczej sensitivity) zdefiniowane jako

$$\text{Accuracy} := \frac{\text{TP} + \text{TN}}{n}, \quad \text{Precision} := \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad \text{Recall} := \frac{\text{TP}}{\text{TP} + \text{FN}}.$$

Wartość accuracy mówi po prostu jaki stosunek przykładów został poprawnie zaklasyfikowany (zauważmy tutaj, że $\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN} = n$). Nie jest to jednak dobra miara jakości, gdy nasz zbiór jest niezbalansowany, tj. zawiera więcej przykładów określonej klasy.

		Predicted condition			
	Total population $= \mathrm{P} + \mathrm{N}$	**Predicted Positive (PP)**	**Predicted Negative (PN)**	Informedness, bookmaker informedness (BM) $= \mathrm{TPR} + \mathrm{TNR} - 1$	Prevalence threshold (PT) $= \frac{\sqrt{\mathrm{TPR} \times \mathrm{FPR}} - \mathrm{FPR}}{\mathrm{TPR} - \mathrm{FPR}}$
Actual condition	**Positive (P)** [a]	**True positive** (TP), hit[b]	**False negative** (FN), miss, underestimation	True positive rate (TPR), recall, sensitivity (SEN), probability of detection, hit rate, power $= \frac{\mathrm{TP}}{\mathrm{P}} = 1 - \mathrm{FNR}$	False negative rate (FNR), miss rate type II error [c] $= \frac{\mathrm{FN}}{\mathrm{P}} = 1 - \mathrm{TPR}$
	Negative (N)[d]	**False positive** (FP), false alarm, overestimation	**True negative** (TN), correct rejection[e]	False positive rate (FPR), probability of false alarm, fall-out type I error [f] $= \frac{\mathrm{FP}}{\mathrm{N}} = 1 - \mathrm{TNR}$	True negative rate (TNR), specificity (SPC), selectivity $= \frac{\mathrm{TN}}{\mathrm{N}} = 1 - \mathrm{FPR}$
	Prevalence $= \frac{\mathrm{P}}{\mathrm{P} + \mathrm{N}}$	Positive predictive value (PPV), precision $= \frac{\mathrm{TP}}{\mathrm{PP}} = 1 - \mathrm{FDR}$	False omission rate (FOR) $= \frac{\mathrm{FN}}{\mathrm{PN}} = 1 - \mathrm{NPV}$	Positive likelihood ratio (LR+) $= \frac{\mathrm{TPR}}{\mathrm{FPR}}$	Negative likelihood ratio (LR−) $= \frac{\mathrm{FNR}}{\mathrm{TNR}}$
	Accuracy (ACC) $= \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{P} + \mathrm{N}}$	False discovery rate (FDR) $= \frac{\mathrm{FP}}{\mathrm{PP}} = 1 - \mathrm{PPV}$	Negative predictive value (NPV) $= \frac{\mathrm{TN}}{\mathrm{PN}} = 1 - \mathrm{FOR}$	Markedness (MK), deltaP (Δp) $= \mathrm{PPV} + \mathrm{NPV} - 1$	Diagnostic odds ratio (DOR) $= \frac{\mathrm{LR+}}{\mathrm{LR-}}$
	Balanced accuracy (BA) $= \frac{\mathrm{TPR} + \mathrm{TNR}}{2}$	$\mathrm{F_1}$ score $= \frac{2\,\mathrm{PPV} \times \mathrm{TPR}}{\mathrm{PPV} + \mathrm{TPR}} = \frac{2\,\mathrm{TP}}{2\,\mathrm{TP} + \mathrm{FP} + \mathrm{FN}}$	Fowlkes–Mallows index (FM) $= \sqrt{\mathrm{PPV} \times \mathrm{TPR}}$	Matthews correlation coefficient (MCC) $= \sqrt{\mathrm{TPR} \times \mathrm{TNR} \times \mathrm{PPV} \times \mathrm{NPV}} - \sqrt{\mathrm{FNR} \times \mathrm{FPR} \times \mathrm{FOR} \times \mathrm{FDR}}$	Threat score (TS), critical success index (CSI), Jaccard index $= \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN} + \mathrm{FP}}$

Rysunek 1: Macierz pomyłek oraz możliwe metryki oceny jakości klasyfikatora. Źródło: en.wikipedia.org/wiki/Confusion_matrix

Wartość precision określa jak pewny jest klasyfikator przy wykrywaniu klasy pozytywnej, natomiast recall mówi o tym jak dobrze klasyfikator „wyławia" przykłady pozytywne. Zauważmy jednak, iż nie możemy stosować żadnej z tych metryk w odosobnieniu. Istotnie klasyfikator, który zwraca zawsze klasę pozytywną ma maksymalny recall, a klasyfikator, który zwraca zawsze klasę negatywną ma nieokreślony precision (i jest oczywiście beznadziejnym klasyfikatorem). Musimy więc zawsze ewaluować model na obu tych metrykach i jedynie dobry wynik obu z nich mówi o jakości klasyfikatora. Oczywiście czasami chcielibyśmy określić jakość modelu za pomocą jednej liczby, a niekoniecznie sprawdzać zawsze macierz pomyłek (choć jest to bardzo użyteczne) lub podawać wartości dwóch metryk. Metryką, która

łączy precision i recall jest F_β–*score* zdefiniowany jako

$$F_\beta := (1+\beta^2)\frac{\text{Precision}\cdot\text{Recall}}{\beta^2\cdot\text{Precision}+\text{Recall}}\,,$$

gdzie β określa ile razy bardziej ważny jest recall od precision. Typowo używa się F_1–score

$$F_1 = 2\frac{\text{Precision}\cdot\text{Recall}}{\text{Precision}+\text{Recall}}\,.$$

Wiele klasyfikatorów oprócz twardych predykcji zwraca również rozkład prawdopodobieństwa nad klasami. W przypadku klasyfikacji binarnej jest to oczywiście rozkład zero-jedynkowy z parametrem p określającym prawdopodobieństwo klasy pozytywnej dla danego wektora cech. Standardowo oczywiście twardą predykcją jest ta z klas, która ma większe prawdopodobieństwo, czyli (co równoważne) predykcją jest klasa pozytywna jeśli $p > 0.5$. W niektórych problemach chcemy jednak zmienić ten próg i dokonać tzw. *threshold tuning*. Wykresem, który pozwala na dokonanie tuningu progu jest *krzywa ROC* (z ang. *Receiver Operatic Characteristic curve*), która jest krzywą parametryczną wyznaczoną przez punkty $(\text{FPR}(\text{threshold}), \text{TPR}(\text{threshold}))$ dla progów z zakresu $[0;1]$, gdzie

$$\text{TPR} := \frac{\text{TP}}{\text{TP}+\text{FN}}\,,\quad \text{FPR} := \frac{\text{FP}}{\text{FP}+\text{TN}}\,.$$

Metryką niezależną od wybranego progu jest tzw. *AUROC* (z ang. *Area under ROC curve*) zdefiniowany jako pole powierzchni pod krzywą ROC dla danego klasyfikatora. Zauważmy, że klasyfikator losowy, który zwraca zawsze klasę pozytywną z prawdopodobieństwem równym wartości progu ma wartość AUROC równą 0.5, natomiast idealny klasyfikator, który niezależnie od wartości progu klasyfikuje wszystkie przykłady poprawnie ma AUROC równy 1.

Inną analogiczną metryką jest *AUPRC*, gdzie zamiast krzywej ROC stosujemy krzywą *PRC* (z ang. *Precision–Recall Curve*), w której zamiast TPR i FPR używamy odpowiednio Precision i Recall. Metryka AUPRC jest często wykorzystywana w przypadku klasyfikacji ekstremalnie niezbalansowanej, w której mamy bardzo mało ($< 1\%$) klasy pozytywnej.

W przypadku klasyfikacji wieloklasowej używamy zasadniczo takich samych metryk jak w klasyfikacji binarnej, ale wprowadzamy mikro i makro uśrednianie (z ang. *micro/macro-averaging*). Przez TP_k będziemy rozumieć liczbę prawidłowo zaklasyfikowanych przykładów z klasy k, FP_k to liczba przykładów z innych klas, które zaklasyfikowaliśmy nieprawidłowo jako k–tą klasę, FN_k to liczba przykładów z klasy k, które zaklasyfikowaliśmy jako

inną klasę. Wówczas odpowiednie metryki mają postać

$$\text{MicroPrecision} := \frac{\sum_k \text{TP}_k}{\sum_k \text{TP}_k + \sum_k \text{FP}_k},$$
$$\text{MacroPrecision} := \frac{1}{K} \sum_{k=1}^{K} \frac{\text{TP}_k}{\text{TP}_k + \text{FP}_k}$$

oraz

$$\text{MicroRecall} := \frac{\sum_k \text{TP}_k}{\sum_k \text{TP}_k + \sum_k \text{FN}_k},$$
$$\text{MacroRecall} := \frac{1}{K} \sum_{k=1}^{K} \frac{\text{TP}_k}{\text{TP}_k + \text{FN}_k}.$$

W przypadku klasyfikacji wieloklasowej macierz pomyłek jest macierzą wymiaru $K \times K$, gdzie K jest liczbą klas.

1.4.3 Tuning hiperparametrów i walidacja skrośna

Praktycznie wszystkie modele uczenia maszynowego mają hiperparametry, często liczne, które w zauważalny sposób wpływają na wyniki, a szczególnie na underfitting i overfitting. Ich wartości trzeba dobrać zatem dość dokładnie. Proces doboru hiperparametrów nazywa się tuningiem hiperparametrów (z ang. *hyperparameter tuning*).

Istnieje na to wiele sposobów. Większość z nich polega na tym, że trenuje się za każdym razem model z nowym zestawem hiperparametrów i wybiera się ten zestaw, który pozwala uzyskać najlepsze wyniki. Metody głównie różnią się między sobą sposobem doboru kandydujących zestawów hiperparametrów. Najprostsze i najpopularniejsze to:

- pełne przeszukiwanie (z ang. *grid search*) – definiujemy możliwe wartości dla różnych hiperparametrów, a metoda sprawdza ich wszystkie możliwe kombinacje (czyli siatkę),
- losowe przeszukiwanie (z ang. *randomized search*) – definiujemy możliwe wartości jak w pełnym przeszukiwaniu, ale sprawdzamy tylko ograniczoną liczbę losowo wybranych kombinacji.

Jak ocenić, jak dobry jest jakiś zestaw hiperparametrów? Nie możemy sprawdzić tego na zbiorze treningowym – wyniki byłyby zbyt optymistyczne. Nie możemy wykorzystać zbioru testowego – mielibyśmy wyciek danych, bo

wybieralibyśmy model explicite pod nasz zbiór testowy. Trzeba zatem osobnego zbioru, na którym będziemy na bieżąco sprawdzać jakość modeli dla różnych hiperparametrów. Jest to zbiór walidacyjny (z ang. *validation set*). Zbiór taki wycina się ze zbioru treningowego.

Jednorazowy podział zbioru na części nazywa się *split validation* lub *holdout*. Używamy go, gdy mamy sporo danych, i 10-20% zbioru jako dane walidacyjne czy testowe to dość dużo, żeby mieć przyzwoite oszacowanie. Zbyt mały zbiór walidacyjny czy testowy da nam mało wiarygodne wyniki – nie da się nawet powiedzieć, czy zbyt pesymistyczne, czy optymistyczne. W praktyce niestety często mamy mało danych. Trzeba zatem jakiejś magicznej metody, która stworzy nam więcej zbiorów walidacyjnych z tej samej ilości danych. Taką metodą jest walidacja skrośna (z ang. *cross validation*, CV). Polega na tym, że dzielimy zbiór treningowy na K równych podzbiorów, tzw. foldów. Każdy podzbiór po kolei staje się zbiorem walidacyjnym, a pozostałe łączymy w zbiór treningowy. Trenujemy zatem K modeli dla tego samego zestawu hiperparametrów i każdy testujemy na zbiorze walidacyjnym. Mamy K wyników dla zbiorów walidacyjnych, które możemy uśrednić (i ewentualnie obliczyć odchylenie standardowe). Takie wyniki są znacznie bardziej wiarygodne.

2 Probabilistyka

2.1 Zmienne losowe

Zmienna losowa to formalnie odwzorowanie ze zbioru zdarzeń elementarnych Ω tj. zbioru atomowych wyników doświadczenia losowego w zbiór $\mathbb{R}^n$

$$\boldsymbol{X} : \Omega \mapsto \mathbb{R}^n \,.$$

Jest to zatem funkcja, która przyporządkowuje zdarzeniom losowym wartość liczbową. Każda zmienna losowa opisuje więc zmienną w klasycznym sensie, której wartości pochodzi z pewnego rozkładu. Rozkład ten jest zadany jednoznacznie przez funkcję $F : \mathbb{R}^n \mapsto [0; 1]$ taką, że

$$F(\boldsymbol{x}) := \Pr\big(\boldsymbol{X}^1 \leq \boldsymbol{x}^1, \ldots, \boldsymbol{X}^n \leq \boldsymbol{x}^n\big) \,,$$

którą nazywa się dystrybuantą (z ang. *cumulative distribution function, cdf*). Każdy rozkład zmiennej losowej można opisać za pomocą dystrybuanty jednak jest to niewygodne. W dwóch przypadkach: rozkładów dyskretnych i rozkładów ciągłych rozkład zmiennej losowej można opisać prościej za pomocą odpowiednio funkcji prawdopodobieństwa (z ang. *probability*

mass function, pmf) oraz gęstości prawdopodobieństwa (z ang. *probability density function, pdf*).

Definicja 2.1. Zmienna losowa $\boldsymbol{X}$ ma dyskretny rozkład prawdopodobieństwa, jeśli istnieje skończony lub przeliczalny zbiór $\mathcal{S} \subset \mathbb{R}^n$ taki, że $\Pr(\boldsymbol{X} \in \mathcal{S}) = 1$. Wówczas rozkład ten jest zadany przez podanie funkcji prawdopodobieństwa $p(\boldsymbol{x}) = \Pr(\boldsymbol{X} = \boldsymbol{x})$ dla $\boldsymbol{x} \in \mathcal{S}$.

Definicja 2.2. Zmienna losowa $\boldsymbol{X}$ ma z kolei ciągły rozkład prawdopodobieństwa, jeśli istnieje funkcja $p : \mathbb{R}^n \mapsto \mathbb{R}_+$ taka, że

$$\Pr\big(\boldsymbol{X}^1 \in (a_1; b_1), \ldots, \boldsymbol{X}^n \in (a_n; b_n)\big) = \int_{a_1}^{b_1} \cdots \int_{a_n}^{b_n} p(\boldsymbol{x}) \, \mathrm{d}^n \boldsymbol{x}$$

dla dowolnej kostki $(a_1; b_1) \times \ldots \times (a_n; b_n)$.

Definicja 2.3 (Wartości oczekiwanej)**.** Wartością oczekiwaną funkcji zmiennej losowej $\boldsymbol{f}(\boldsymbol{X})$ nazywamy wektor $\mathbb{E}[\boldsymbol{f}(\boldsymbol{x})]$ określoną wzorem

$$\sum_{\boldsymbol{x} \in \mathcal{S}} \boldsymbol{f}(\boldsymbol{x}) p(\boldsymbol{x}) \,, \quad \int_{\mathbb{R}^n} \boldsymbol{f}(\boldsymbol{x}) p(\boldsymbol{x}) \, \mathrm{d}^n \boldsymbol{x}$$

odpowiednio dla rozkładu dyskretnego i ciągłego.

Definicja 2.4 (Macierzy kowariancji)**.** Macierzą kowariancji funkcji zmiennej losowej $\boldsymbol{f}(\boldsymbol{X})$ nazywamy macierz

$$\mathbb{E}\left[(\boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{m}_{\boldsymbol{f}})(\boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{m}_{\boldsymbol{f}})^T\right] \,,$$

gdzie

$$\boldsymbol{m}_{\boldsymbol{f}} = \mathbb{E}\left[\boldsymbol{f}(\boldsymbol{x})\right] \,.$$

Elementy diagonalne macierzy kowariancji nazywamy wariancjami, a elementy pozadiagonalne kowariancjami.

Definicja 2.5 (Kwantyla i mody)**.** Kwantylem q_p rzędu $p \in (0;1)$ zmiennej losowej jednowymiarowej o rozkładzie ciągłym z dystrybuantą F nazywamy dowolne rozwiązanie równania

$$F(x) = p\,.$$

Modą tej zmiennej nazywamy dowolne maksimum lokalne gęstości tego rozkładu.

Twierdzenie 2.1. Niech zmienna n–wymiarowa $\boldsymbol{X}$ ma rozkład ciągły o gęstości $p_{\boldsymbol{X}}$ i niech $\boldsymbol{Y}^i = \boldsymbol{\varphi}^i(\boldsymbol{X})$ dla $i = 1, \ldots, n$. Jeśli odwzorowanie $\boldsymbol{\varphi}$ jest różniczkowalne i odwracalne, przy czym odwzorowanie odwrotne $\boldsymbol{\psi} = \boldsymbol{\varphi}^{-1}$ jest różniczkowalne, to n–wymiarowa zmienna $\boldsymbol{Y}$ ma rozkład o gęstości

$$p_{\boldsymbol{Y}}(\boldsymbol{y}) = |J| p_{\boldsymbol{X}}(\boldsymbol{\psi}(\boldsymbol{y}))\,,$$

gdzie $J := \det\left[\frac{\partial \boldsymbol{\psi}^j}{\partial \boldsymbol{y}^i}\right]$ jest jakobianem odwzorowania $\boldsymbol{\psi}$.

Twierdzenie 2.2. Dla dowolnych zmiennych losowych $\boldsymbol{X}$, $\boldsymbol{Y}$ zachodzi

$$p_{\boldsymbol{X}}(\boldsymbol{x}) = \int\limits_{\mathbb{R}^k} p(\boldsymbol{x}, \boldsymbol{y})\, \mathrm{d}^k \boldsymbol{y} \quad \text{(sum rule)}$$

Twierdzenie 2.3. Dla dowolnych zmiennych losowych $\boldsymbol{X}$, $\boldsymbol{Y}$ zachodzi

$$p(\boldsymbol{x}, \boldsymbol{y}) = p(\boldsymbol{x} \mid \boldsymbol{y}) p_{\boldsymbol{Y}}(\boldsymbol{y}) \quad \text{(product rule)}$$

Twierdzenie 2.4. Zmienne losowe $\boldsymbol{X}$ i $\boldsymbol{Y}$ są niezależne wtedy i tylko wtedy, gdy

$$p(\boldsymbol{x}, \boldsymbol{y}) = p_{\boldsymbol{X}}(\boldsymbol{x}) p_{\boldsymbol{Y}}(\boldsymbol{y}) \quad \text{(independence)}$$

Twierdzenie 2.5. Dla dowolnych zmiennych losowych $\boldsymbol{X}$, $\boldsymbol{Y}$ zachodzi

$$\underbrace{p(\boldsymbol{x} \mid \boldsymbol{y})}_{\text{posterior}} = \frac{\overbrace{p(\boldsymbol{y} \mid \boldsymbol{x})}^{\text{likelihood}} \overbrace{p_{\boldsymbol{X}}(\boldsymbol{x})}^{\text{prior}}}{\underbrace{\int_{\mathbb{R}^k} p(\boldsymbol{y} \mid \boldsymbol{x}) p_{\boldsymbol{X}}(\boldsymbol{x}) \,\mathrm{d}^k \boldsymbol{x}}_{\text{evidence}}} \quad \text{(Bayes theorem)}$$

Definicja 2.6 (Warunkowej wartości oczekiwanej)**.** Warunkową wartość oczekiwaną $\boldsymbol{f}(\boldsymbol{X})$ pod warunkiem $\boldsymbol{Y} = \boldsymbol{y}$ nazywamy wielkość

$$\mathbb{E}[\boldsymbol{f}(\boldsymbol{x}) \mid \boldsymbol{y}] = \int_{\mathbb{R}^n} \boldsymbol{f}(\boldsymbol{x}) p(\boldsymbol{x} \mid \boldsymbol{y}) \,\mathrm{d}^n \boldsymbol{x}$$

2.2 Ważne rozkłady jednowymiarowe

Definicja 2.7 (Rozkładu dwupunktowego)**.** Jeśli X jest zmienną losową rzeczywistą o rozkładzie dyskretnym i $\mathcal{S} = \{x_1, x_2\}$ oraz $p(x_1) = p$, to mówimy, że X ma rozkład dwupunktowy z parametrem p. Jeśli $x_1 = 1$ i $x_2 = 0$ to taki rozkład dwupunktowy nazywamy *rozkładem zero–jedynkowym* (lub rozkładem Bernoulliego) i oznaczamy jako $X \sim \mathrm{Ber}(p)$.

Definicja 2.8 (Schematu dwumianowego)**.** Rozważmy doświadczenie losowe o dwu możliwych wynikach: sukces osiągamy z prawdopodobieństwem p, porażkę z prawdopodobieństwem $1-p$. Doświadczenie tego rodzaju nazywamy *próbą Bernoulliego*. Doświadczenie takie jest modelowane zmienną losową o rozkładzie dwupunktowym z parametrem p. Schematem dwumianowym (lub schematem Bernoulliego) nazywamy doświadczenie polegające na n–krotnym powtórzeniu próby Bernoulliego, przy założeniu, iż poszczególne próby są od siebie niezależne.

Definicja 2.9 (Rozkładu dwumianowego)**.** Niech X będzie zmienną losową taką, że X jest liczbą sukcesów w schemacie dwumianowym długości n z prawdopodobieństwem sukcesu w każdej próbie równym p. Wówczas

$$\Pr(X = k) = \binom{n}{k} p^k (1-p)^{n-k} .$$

Rozkład prawdopodobieństwa określony powyższym wzorem nazywam się rozkładem dwumianowym o parametrach n, p. Jeśli zmienna X ma rozkład dwumianowy to stosujemy notację $X \sim \mathrm{Bin}(n, p)$.

Definicja 2.10 (Rozkładu geometrycznego)**.** Mówimy, że zmienna losowa X ma rozkład geometryczny z parametrem $p \in (0; 1)$, tj. $X \sim \mathrm{Geo}(p)$, jeśli $\mathcal{S} = \mathbb{N} \setminus \{0\}$, a funkcja prawdopodobieństwa ma postać

$$p(x) = (1-p)^{x-1} p .$$

Zmienna X opisuje czas oczekiwania na pierwszy sukces w schemacie dwumianowym o nieskończonej długości.

Definicja 2.11 (Rozkładu Poissona)**.** Jeśli zmienna X o wartościach w $\mathbb{N}$ opisuje liczbę wystąpień pewnego powtarzalnego zdarzenia w przedziale czasowym $[0; t]$, przy czym spełnione są następujące założenia:

- powtórzenia zdarzenia występują niezależnie od siebie;
- „intensywność" wystąpień r jest stała;
- w danej chwili (rozumianej jako odpowiednio mały przedział) może zajść co najwyżej jedno zdarzenie

to zmienna ta ma rozkład Poissona z parametrem $\lambda = rt$, tj. $X \sim \mathrm{Pos}(\lambda)$. Jeśli $X \sim \mathrm{Pos}(\lambda)$, to

$$\Pr(X = k) = \frac{\mathrm{e}^{-\lambda} \lambda^k}{k!} .$$

Twierdzenie 2.6 (Poissona)**.** Niech (X_n) będzie ciągiem zmiennych losowych takich, że $X_n \sim \mathrm{Bin}(n, p_n)$, gdzie (p_n) jest ciągiem takim, że

$$\lim_{n\to\infty} np_n = \lambda$$

dla pewnej liczby $\lambda > 0$. Wówczas

$$\lim_{n\to\infty} \Pr(X_n = k) = \frac{\mathrm{e}^{-\lambda}\lambda^k}{k!}\,.$$

Definicja 2.12 (Rozkładu jednostajnego)**.** Mówimy, że zmienna X o rozkładzie ciągłym ma rozkład jednostajny na przedziale $[a; b]$ tzn. $X \sim \mathcal{U}(a, b)$ jeśli jej gęstość wyraża się wzorem

$$p(x) = \begin{cases} \frac{1}{b-a}\,, & x \in [a; b] \\ 0\,, & x \notin [a; b] \end{cases}\,.$$

Definicja 2.13 (Rozkładu wykładniczego)**.** Niech T będzie zmienną modelującą czas oczekiwania na pierwsze zdarzenie w ciągu zdarzeń takim, że czas wystąpienia każdego z nich w przedziale $[0; t]$ jest opisany przez zmienną $X \sim \mathrm{Pos}(\lambda t)$. wtedy

$$\Pr(T > t) = \Pr(X = 0) = \mathrm{e}^{-\lambda t}$$

oraz

$$\Pr(T > 0) = 1\,.$$

Mówimy wtedy, że T ma rozkład wykładniczy z parametrem λ, tzn. $T \sim \mathrm{Exp}(\lambda)$. Gęstość rozkładu wykładniczego ma postać

$$p(t) = \begin{cases} 0\,, & t \le 0 \\ \lambda\mathrm{e}^{-\lambda t}\,, & t > 0 \end{cases}\,.$$

Definicja 2.14 (Rozkładu normalnego)**.** Mówimy, że zmienna losowa X o gęstości $p(x)$ ma rozkład normalny z parametrami $\mu \in$

$\mathbb{R}, \sigma^2 \in [0; +\infty)$, tzn. $X \sim \mathcal{N}(\mu, \sigma^2)$, jeśli

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

Jeśli $\mu = 0$ i $\sigma = 1$, to taki rozkład nazywamy standardowym rozkładem normalnym.

2.3 Wielowymiarowy rozkład normalny

Definicja 2.15 (Standardowego wielowymiarowego rozkładu normalnego)**.** Zmienna losowa $\boldsymbol{X}$ ma standardowy n–wymiarowy rozkład normalny jeśli jej składowe są niezależne i dla każdego $i = 1, \ldots, n$ $\boldsymbol{X}^i \sim \mathcal{N}(0, 1)$. Jest to rozkład ciągły o gęstości

$$p(\boldsymbol{x}) = \frac{1}{\sqrt{(2\pi)^n}} \exp\left(-\frac{1}{2}\boldsymbol{x}^T\boldsymbol{x}\right).$$

Definicja 2.16 (Wielowymiarowego rozkładu normalnego)**.** Zmienna losowa $\boldsymbol{X}$ ma n–wymiarowy rozkład normalny (z ang. *Multivariate Normal Distribution, MVN*), tzn. $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ jeśli istnieje k–wymiarowa zmienna losowa $\boldsymbol{Z}$ o standardowym rozkładzie normalnym dla pewnego $k \leq n$ oraz istnieje $\boldsymbol{\mu} \in \mathbb{R}^n$ i macierz $\boldsymbol{A} \in \mathbb{R}^{n\times k}$ takie, że $\boldsymbol{\Sigma} = \boldsymbol{A}\boldsymbol{A}^T$ oraz

$$\boldsymbol{X} = \boldsymbol{A}\boldsymbol{Z} + \boldsymbol{\mu}.$$

Jeśli macierz $\boldsymbol{\Sigma}$ jest dodatnio określona, to rozkład $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ jest ciągły, a jego gęstość jest dana przez

$$p(\boldsymbol{x}) = \frac{1}{\sqrt{(2\pi)^n \det \boldsymbol{\Sigma}}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right)$$

Macierz $\boldsymbol{\Sigma}^{-1}$ nazywa się *macierzą precyzji*.

Poziomice gęstości niezdegenerowanego wielowymiarowego rozkładu normalnego są elipsoidami, których półosie są skierowane wzdłuż wektorów własnych macierzy $\boldsymbol{\Sigma}$ i mają długości proporcjonalne do pierwiastka z war-

tości własnych.

Twierdzenie 2.7 (Własności niezdegenerowanego rozkładu normalnego)**.** Niech $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ dla dodatnio określonej macierzy $\boldsymbol{\Sigma}$, wówczas

1. Wszystkie rozkłady brzegowe i warunkowe $\boldsymbol{X}$ są rozkładami normalnymi. Istotnie jeśli

$$\begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \boldsymbol{\mu_x} \\ \boldsymbol{\mu_y} \end{bmatrix}, \begin{bmatrix} \boldsymbol{\Sigma_{xx}} & \boldsymbol{\Sigma_{xy}} \\ \boldsymbol{\Sigma_{yx}} & \boldsymbol{\Sigma_{yy}} \end{bmatrix} \right)$$

   to można pokazać, iż

$$\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{\mu_x}, \boldsymbol{\Sigma_{xx}}), \quad \boldsymbol{y} \sim \mathcal{N}(\boldsymbol{\mu_y}, \boldsymbol{\Sigma_{yy}}).$$

   oraz $\boldsymbol{x} \mid \boldsymbol{y} \sim \mathcal{N}(\boldsymbol{\mu_{x|y}}, \boldsymbol{\Sigma_{x|y}})$, gdzie

$$\boldsymbol{\mu_{x|y}} = \boldsymbol{\mu_x} + \boldsymbol{\Sigma_{xy}}\boldsymbol{\Sigma_{yy}^{-1}}(\boldsymbol{y} - \boldsymbol{\mu_y}), \quad \boldsymbol{\Sigma_{x|y}} = \boldsymbol{\Sigma_{xx}} - \boldsymbol{\Sigma_{xy}}\boldsymbol{\Sigma_{yy}^{-1}}\boldsymbol{\Sigma_{yx}}.$$

2. Zmienne składowe $\boldsymbol{X}^1, \ldots, \boldsymbol{X}^n$ są niezależne wtedy i tylko wtedy, gdy $\boldsymbol{\Sigma}$ jest macierzą diagonalną.

2.4 Wnioskowanie statystyczne

Niech zmienna losowa $\boldsymbol{X}$ określa model rozkładu pewnej cechy (cech) w ustalonym zbiorze instancji (tzw. *populacji generalnej*). Innymi słowy, przyjmujemy, że wartości cech zachowują się jakby zostały wybrane losowo zgodnie z rozkładem zmiennej $\boldsymbol{X}$. Do podstawowych zagadnień wnioskowania statystycznego należą:

- oszacowanie wielkości charakteryzujących rozkład $\boldsymbol{X}$ (np. wartości średniej albo wariancji);
- weryfikacja hipotez dotyczących rozkładu $\boldsymbol{X}$ (tym nie będziemy się zajmować).

Definicja 2.17 (Modelu statystycznego)**.** Modelem statystycznym nazywamy parę $(\mathcal{P}, \mathcal{X})$, gdzie $\mathcal{P}$ jest rodziną rozkładów prawdopo-

dobieństwa na zbiorze $\mathcal{X}$. Zazwyczaj przyjmuje się

$$\mathcal{P} = \{p(\cdot \mid \boldsymbol{\theta}) : \boldsymbol{\theta} \in \Theta\}$$

dla pewnego zbioru parametrów Θ. Model statystyczny nazywamy *parametrycznym* jeśli $\Theta \subset \mathbb{R}^k$.

Definicja 2.18 (Prostej próby losowej)**.** Prostą próbą losową o liczności n nazywamy ciąg niezależnych zmiennych losowych $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n$ o tym samym rozkładzie $p(\cdot \mid \boldsymbol{\theta}) \in \mathcal{P}$ (z ang. *independent and identically distributed, i.i.d*).

Definicja 2.19 (Estymatora)**.** Estymatorem nazywa się statystykę $\hat{\theta}(\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n)$ służącą do oszacowania wartości parametru θ. Liczbę $\hat{\theta}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ dla konkretnej realizacji prostej próby losowej nazywa się wartością estymatora albo estymatą.

Definicja 2.20 (Funkcji wiarygodności)**.** Funkcją wiarygodności (z ang. *likelihood function*) dla modelu $\mathcal{P} = \{p(\cdot \mid \boldsymbol{\theta}) : \boldsymbol{\theta} \in \Theta\}$ nazywamy funkcję

$$\mathcal{L} : \mathbb{R}^n \times \Theta \ni (\boldsymbol{x}, \boldsymbol{\theta}) \mapsto \mathcal{L}(\boldsymbol{x}; \boldsymbol{\theta}) \in [0; +\infty)$$

wyznaczającą rozkład łączny obserwowanych danych jako funkcję parametru $\boldsymbol{\theta}$.

Niech $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n$ będzie prostą próbą losową. Jeśli $p(\cdot \mid \boldsymbol{\theta})$ opisuje rozkład warunkowy, z którego pochodzą obserwacje, to

$$\mathcal{L}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n; \boldsymbol{\theta}) = \prod_{i=1}^{n} p(\boldsymbol{x}_i \mid \boldsymbol{\theta})\,.$$

Dla wygody obliczeń często rozważa się tzw. zanegowaną logarytmiczną funkcję wiarygodności (z ang. *Negated Log-Likelihood function, NLL*), tzn.

$$L(\boldsymbol{x}; \boldsymbol{\theta}) = -\log \mathcal{L}(\boldsymbol{x}; \boldsymbol{\theta})\,.$$

Wówczas dla realizacji prostej próby losowej mamy

$$L(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n; \boldsymbol{\theta}) = -\sum_{i=1}^{n} \log p(\boldsymbol{x}_i \mid \boldsymbol{\theta}) \,.$$

Definicja 2.21 (Estymatora największej wiarygodności)**.** Estymatorem największej wiarygodności (z ang. *Maximum Likelihood Estimator, MLE*) nazywamy funkcję $\hat{\boldsymbol{\theta}}$, która przy ustalonych wartościach obserwacji (realizacji prostej próby losowej) $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ maksymalizuje wartość funkcji wiarygodności lub, co równoważne, minimalizuje wartość zanegowanej logarytmicznej funkcji wiarygodności tj.

$$\hat{\boldsymbol{\theta}}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) = \arg\min_{\boldsymbol{\theta} \in \Theta} \left[-\sum_{i=1}^{n} \log p(\boldsymbol{x}_i \mid \boldsymbol{\theta}) \right] .$$

Jeśli funkcja wiarygodności jest różniczkowalna względem $\boldsymbol{\theta}$ dla dowolnych $\boldsymbol{x}^i$, to MLE można czasem wyznaczyć analitycznie korzystając z warunku koniecznego optymalności, tzn. rozwiązując układ równań

$$\frac{\partial L(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n; \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = 0 \,.$$

Jeśli MLE nie da się wyliczyć analitycznie, wyznacza się je przy użyciu algorytmów optymalizacji numerycznej. Estymatory MLE są asymptotycznie nieobciążone.

Definicja 2.22 (Estymatora MAP)**.** Estymatorem MAP (z ang. *Maximum a Posteriori Estimator, MAP*) nazywamy funkcję $\hat{\boldsymbol{\theta}}$, która przy ustalonych wartościach obserwacji (realizacji prostej próby losowej) $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ maksymalizuje wartość iloczynu funkcji wiarygodności i priora nad wartościami parametrów lub, co równoważne, minimalizuje zregularyzowaną logarytmiczną funkcję wiarygodności tj.

$$\hat{\boldsymbol{\theta}}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) = \arg\min_{\boldsymbol{\theta} \in \Theta} \left[-\sum_{i=1}^{n} \log p(\boldsymbol{x}_i \mid \boldsymbol{\theta}) - n \log p(\boldsymbol{\theta}) \right] .$$

2.5 Liczby losowe w komputerze

Opiszemy jeszcze pokrótce metody generowania liczb pseudolosowych z dowolnych rozkładów prawdopodobieństwa w sposób algorytmiczny. Podstawowym narzędziem, którego będziemy potrzebować do generowania próbek z bardziej skomplikowanych rozkładów będzie prosty generator liczb z rozkładu jednostajnego $\mathcal{U}(0,1)$. Moglibyśmy oczywiście wykorzystać jakieś fizyczne urządzenie lub proces, który generuje liczby prawdziwie losowe (np. detektor Geigera-Mullera, szum lamp elektronowych, ruletka), ale błędem byłaby rezygnacja z odtwarzalności. Poszukujemy zatem deterministycznej metody, która generuje sekwencje liczb, które są w przybliżeniu losowe. Podstawową metodą do algorytmicznego generowania liczb pseudolosowych jest tzw. *liniowy generator kongruentny* (z ang. *Linear Congruential Generator, LCG*), który jest opisany zależnością rekurencyjną

$$I_{j+1} = (aI_j + c) \mod m\,,$$

gdzie a, c, m to pewne ustalone dodatnie liczby całkowite, a I_0 to tzw. ziarno (z ang. *seed*). LCG generuje liczby całkowite, więc w dużym uproszczeniu liczby zmiennoprzecinkowe z rozkładu $\mathcal{U}(0,1)$ otrzymujemy jako I_j/m (trzeba tutaj jednak uwzględnić problemy wynikające z arytmetyki zmiennoprzecinkowej).

Mając już generator liczb z rozkładu jednostajnego $\mathcal{U}(0,1)$ i znając jawny wzór na dystrybuantę $F(x)$ innego rozkładu jednowymiarowego $\mathcal{D}$ możemy generować liczby z tego rozkładu korzystając z tzw. *metody odwrotnej dystrybuanty*. Istotnie, jeśli $U \sim \mathcal{U}(0,1)$, to $F^{-1}(U) \sim \mathcal{D}$. Istotnie

$$\Pr\big(F^{-1}(U) \le x\big) = \Pr(U \le F(x)) = F(x)\,.$$

Metoda ta ma jedną zasadniczą wadę – musimy znać jawny wzór na dystrybuantę $F(x)$. W przypadku np. tak ważnych rozkładów, jak rozkład normalny dystrybuanta nie jest funkcją elementarną i metoda ta nie jest najlepsza. W przypadku rozkładu normalnego znacznie lepszą metodą jest tzw. *metoda Boxa–Mullera*. Weźmy rozkład łączny dwóch niezależnych zmiennych losowych X, Y pochodzących ze standardowego rozkładu normalnego

$$p(x,y) = \frac{1}{2\pi} \exp\left(\frac{1}{2}(x^2+y^2)\right).$$

Skorzystamy ze wzoru na transformację zmiennych losowych. Istotnie niech

$$X = \sqrt{Z}\cos\Phi\,, \quad Y = \sqrt{Z}\sin\Phi\,,$$

dla $0 < Z$ oraz $0 \leq \Phi < 2\pi$, wówczas

$$q(z,\phi) = p(x(z,\phi), y(z,\phi)) \left| \begin{matrix} \frac{1}{2\sqrt{z}} \cos\phi & \frac{1}{2\sqrt{z}} \sin\phi \\ -\sqrt{z}\sin\phi & \sqrt{z}\cos\phi \end{matrix} \right| = \frac{1}{4\pi} \mathrm{e}^{\frac{z}{2}} .$$

Zauważmy, że otrzymaliśmy sferycznie symetryczny rozkład wykładniczy. Możemy zatem wylosować z rozkładu jednostajnego kąt ϕ oraz z rozkładu wykładniczego wartość z korzystając z metody odwrotnej dystrybuanty. Wówczas wartości $x = \sqrt{z}\cos\phi$, $y = \sqrt{z}\sin\phi$ będą pochodzić ze standardowego rozkładu normalnego. Aby wygenerować próbki z ogólnego wielowymiarowego rozkładu normalnego, korzystamy z definicji, tj. najpierw generujemy n próbek ze standardowego rozkładu normalnego, a następnie korzystamy z przekształcenia afinicznego $\boldsymbol{x} = \boldsymbol{A}\boldsymbol{z} + \boldsymbol{\mu}$, gdzie $\boldsymbol{\Sigma} = \boldsymbol{A}\boldsymbol{A}^T$.

2.6 Monte Carlo

Umiemy już generować próbki z wielowymiarowego rozkładu normalnego. Chcemy teraz poznać metodę, która umożliwi generowanie próbek ze skomplikowanych, wielowymiarowych rozkładów prawdopodobieństwa, których gęstość znamy jedynie z dokładnością do stałej normalizującej, tj. znamy jedynie $\tilde{p}(\boldsymbol{x}) = Zp(\boldsymbol{x})$. Ograniczenie to wynika z chęci próbkowania z posteriora $p(\boldsymbol{x} \mid \boldsymbol{y})$ w sytuacji, gdy znamy jedynie rozkład łączny $p(\boldsymbol{x}, \boldsymbol{y}) = p(\boldsymbol{y} \mid \boldsymbol{x})p_{\boldsymbol{X}}(\boldsymbol{x})$. Okazuje się, iż znajomość rozkładu jedynie z dokładnością do stałej normalizującej jest wystarczająca do generowania próbek z tego rozkładu. Generowanie próbek z kolei wystarcza natomiast, na mocy silnego prawa wielkich liczb, do szacowania wartości średnich dowolnych funkcji zmiennej $\boldsymbol{x}$. Przypomnijmy, iż na mocy silnego prawa wielkich liczb ciąg średnich częściowych $(\overline{\boldsymbol{X}}_n)$ ciągu zmiennych losowych $(\boldsymbol{X}_n)$ i.i.d. z rozkładu $\boldsymbol{X} \sim \mathcal{D}$ jest zbieżny z prawdopodobieństwem 1 do wartości oczekiwanej $\mathbb{E}[\boldsymbol{X}]$ tj.

$$\Pr\left(\lim_{n\to\infty} \overline{\boldsymbol{X}}_n = \mathbb{E}[\boldsymbol{X}] \right) = 1 .$$

Wartość oczekiwaną $\mathbb{E}[\boldsymbol{X}]$ możemy zatem przybliżyć średnią $\overline{\boldsymbol{X}}_n$ z dużej ilości próbek.

Pozostaje pytanie w jaki sposób generować próbki ze skomplikowanych rozkładów prawdopodobieństwa, których gęstości znamy jedynie z dokładnością do stałej normalizującej. Poniżej przedstawimy dwa algorytmy próbkowania: algorytm IS oraz Metropolisa–Hastingsa będący szczególną realizacją całej rodziny algorytmów próbkowania zwanych Markov Chain Monte Carlo (MCMC).

2.6.1 Algorytm Importance Sampling

Załóżmy, iż chcemy obliczyć wartość oczekiwaną pewnej funkcji zmiennej losowej $\boldsymbol{x}$ względem skomplikowanego rozkładu prawdopodobieństwa $p(\boldsymbol{x})$, który znamy jedynie z dokładnością do stałej normalizującej

$$p(\boldsymbol{x}) = \frac{1}{Z_p}\tilde{p}(\boldsymbol{x})$$

tj. szukamy

$$\mathbb{E}_p[f(\boldsymbol{x})] = \int f(\boldsymbol{x})p(\boldsymbol{x})\mathrm{d}^n\boldsymbol{x}\,.$$

Jeśli umiemy generować próbki $\boldsymbol{x}$ z innego (prostszego) rozkładu $q(\boldsymbol{x})$ (np. wielowymiarowego rozkładu normalnego), który nazywamy rozkładem proponującym kandydatów (z ang. *proposal distribution*) to możemy zapisać

$$\begin{aligned}\mathbb{E}_p[f(\boldsymbol{x})] &= \int\limits_{\mathbb{R}^n} f(\boldsymbol{x})p(\boldsymbol{x})\mathrm{d}^n\boldsymbol{x} = \int\limits_{\mathbb{R}^n} f(\boldsymbol{x})\frac{p(\boldsymbol{x})}{q(\boldsymbol{x})}q(\boldsymbol{x})\mathrm{d}^n\boldsymbol{x} \\ &= \mathbb{E}_q\left[f(\boldsymbol{x})\frac{p(\boldsymbol{x})}{q(\boldsymbol{x})}\right] = \frac{Z_q}{Z_p}\mathbb{E}_q\left[f(\boldsymbol{x})\frac{\tilde{p}(\boldsymbol{x})}{\tilde{q}(\boldsymbol{x})}\right]\,.\end{aligned}$$

Zakładamy tutaj, iż nośnik rozkładu p zawiera się w nośniku q tj. $\operatorname{supp} p \subseteq \operatorname{supp} q$. Stosunek stałych Z_p/Z_q również możemy oszacować z próbek z q, gdyż mamy

$$Z_p = \int\limits_{\mathbb{R}^n} \tilde{p}(\boldsymbol{x})\,\mathrm{d}^n\boldsymbol{x} = Z_q \int\limits_{\mathbb{R}^n} \frac{\tilde{p}(\boldsymbol{x})}{\tilde{q}(\boldsymbol{x})}q(\boldsymbol{x})\,\mathrm{d}^n\boldsymbol{x} = Z_q\mathbb{E}_q\left[\frac{\tilde{p}(\boldsymbol{x})}{\tilde{q}(\boldsymbol{x})}\right]\,,$$

skąd ostatecznie

$$\mathbb{E}_p[f(\boldsymbol{x})] = \frac{\mathbb{E}_q\left[f(\boldsymbol{x})\frac{\tilde{p}(\boldsymbol{x})}{\tilde{q}(\boldsymbol{x})}\right]}{\mathbb{E}_q\left[\frac{\tilde{p}(\boldsymbol{x})}{\tilde{q}(\boldsymbol{x})}\right]}\,.$$

Jeśli z rozkładu q wygenerowaliśmy próbki $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$ to na mocy silnego prawa wielkich liczb mamy

$$\boxed{\mathbb{E}_p[f(\boldsymbol{x})] \approx \frac{\sum_{i=1}^m f(\boldsymbol{x}_i)\frac{\tilde{p}(\boldsymbol{x}_i)}{\tilde{q}(\boldsymbol{x}_i)}}{\sum_{j=1}^m \frac{\tilde{p}(\boldsymbol{x}_j)}{\tilde{q}(\boldsymbol{x}_j)}} = \sum_{i=1}^m \lambda_i f(\boldsymbol{x}_i)\,,}$$

gdzie

$$\boxed{\lambda_i = \frac{\tilde{p}(\boldsymbol{x}_i)/\tilde{q}(\boldsymbol{x}_i)}{\sum_{j=1}^m \tilde{p}(\boldsymbol{x}_j)/\tilde{q}(\boldsymbol{x}_j)}\,.}$$

Algorytm Importance Sampling jest prostym algorytmem Monte Carlo, który ma jeden zasadniczy problem. W jaki sposób mamy wybrać rozkład proponujący kandydatów q? Pewną odpowiedź na to pytanie sugeruje analiza wariancji statystyki

$$\overline{f}_m(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m) = \frac{1}{m} \sum_{i=1}^{m} \frac{f(\boldsymbol{x}_i)p(\boldsymbol{x}_i)}{q(\boldsymbol{x}_i)}$$

dla $\boldsymbol{x}_i \sim q$ mamy

$$\mathbb{V}[\overline{f}_m] = \frac{1}{m}\mathbb{V}_q\left[f(\boldsymbol{x})\frac{p(\boldsymbol{x})}{q(\boldsymbol{x})}\right] = \frac{1}{m}\int\limits_{\mathbb{R}^n} \frac{(f(\boldsymbol{x})p(\boldsymbol{x}) - \mu_f q(\boldsymbol{x}))^2}{q(\boldsymbol{x})}\,\mathrm{d}^n\boldsymbol{x}\ .$$

Chcemy oczywiście, aby wariancja była jak najmniejsza, gdyż wówczas mała liczba próbek da dobre przybliżenie wartości oczekiwanej. Rozkład proponujący kandydatów powinien być zatem proporcjonalny do $f(\boldsymbol{x})p(\boldsymbol{x})$, co może być trudne do praktycznego zrealizowania.

2.6.2 Algorytm Metropolisa–Hastingsa

Cała klasa algorytmów próbkowania MCMC opiera się na idei wyrażenia generowania próbek jako ewolucji pewnego łańcucha Markowa.

Definicja 2.23 (Łańcucha Markowa)**.** Łańcuchem Markowa nazwiemy ciąg zmiennych losowych $(\boldsymbol{X}_t)$ o wartościach w $\mathbb{R}^n$ taki, że spełnione jest *kryterium Markowa*

$$\begin{aligned}\forall A \subset \mathbb{R}^n : \Pr(\boldsymbol{X}_t \in A \mid \boldsymbol{X}_{t-1} = \boldsymbol{x}_{t-1}, \ldots, \boldsymbol{X}_0 = \boldsymbol{x}_0)\\ = \Pr(\boldsymbol{X}_t \in A \mid \boldsymbol{X}_{t-1} = \boldsymbol{x}_{t-1})\,.\end{aligned}$$

Elementy ciągu nazywamy stanami łańcucha.

Dany łańcuch jest zadany jednoznacznie przez podanie gęstości prawdopodobieństwa przejścia łańcucha ze stanu $\boldsymbol{x} \to \boldsymbol{y}$, którą będziemy oznaczać przez $\pi(\boldsymbol{y} \mid \boldsymbol{x})$ (zakładamy, iż prawdopodobieństwo przejścia jest niezależne od chwili t – łańcuch taki nazywamy jednorodnym). Funkcja π spełnia oczywiście warunek unormowania

$$\int\limits_{\mathbb{R}^n} \pi(\boldsymbol{y} \mid \boldsymbol{x})\,\mathrm{d}^n\boldsymbol{y} = 1\,,$$

istotnie prawdopodobieństwo przejścia gdziekolwiek ze stanu $\boldsymbol{x}$ jest równe 1. Będziemy zakładać dodatkowo, iż $\forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n : \pi(\boldsymbol{y} \mid \boldsymbol{x}) > 0$. Rozkład $p(\boldsymbol{x})$ łańcucha Markowa (tj. rozkład prawdopodobieństwa z którego losujemy stan łańcucha w danej chwili t) z daną funkcją przejścia π nazwiemy rozkładem stacjonarnym tego łańcucha jeśli

$$p(\boldsymbol{y}) = \int_{\mathbb{R}^n} \pi(\boldsymbol{y} \mid \boldsymbol{x}) p(\boldsymbol{x}) \, \mathrm{d}^n \boldsymbol{x} \ .$$

Rozkład stacjonarny danego łańcucha oznaczymy przez $p^*(\boldsymbol{x})$. Zauważmy, iż jeśli stan początkowy łańcucha $\boldsymbol{X}_0$ pochodzi z rozkładu stacjonarnego p^* to każdy kolejny stan $\boldsymbol{X}_t$ również pochodzi z rozkładu stacjonarnego. Jeśli z kolei stan początkowy pochodzi z jakiegoś innego rozkładu p_0 to rozkład łańcucha w chwili t jest dany przez relację rekurencyjną

$$p_t(\boldsymbol{y}) = \int_{\mathbb{R}^n} \pi(\boldsymbol{y} \mid \boldsymbol{x}) p_{t-1}(\boldsymbol{x}) \, \mathrm{d}^n \boldsymbol{x} \ , \quad \text{dla } t > 1.$$

Rozkładem granicznym łańcucha Markowa nazwiemy granicę w sensie zbieżności punktowej

$$\lim_{t \to \infty} p_t(\boldsymbol{x}) \, .$$

Przy podanych wyżej założeniach istnieje twierdzenie, które mówi iż taki łańcuch Markowa posiada jednoznaczny rozkład stacjonarny tożsamy z rozkładem granicznym. Ponadto warunkiem wystarczającym, aby dany rozkład $p(\boldsymbol{x})$ był rozkładem stacjonarnym łańcucha Markowa jest

$$\forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n : \pi(\boldsymbol{y} \mid \boldsymbol{x}) p(\boldsymbol{x}) = \pi(\boldsymbol{x} \mid \boldsymbol{y}) p(\boldsymbol{y}) \, ,$$

co wynika z scałkowania powyższego równania

$$\int_{\mathbb{R}^n} \pi(\boldsymbol{y} \mid \boldsymbol{x}) p(\boldsymbol{x}) \, \mathrm{d}^n \boldsymbol{x} = \int_{\mathbb{R}^n} \pi(\boldsymbol{x} \mid \boldsymbol{y}) p(\boldsymbol{y}) \, \mathrm{d}^n \boldsymbol{x} = p(\boldsymbol{y}) \int_{\mathbb{R}^n} \pi(\boldsymbol{x} \mid \boldsymbol{y}) \, \mathrm{d}^n \boldsymbol{x} = p(\boldsymbol{y}) \, .$$

Kryterium to nazywamy *kryterium lokalnego balansu* (z ang. *detailed balance condition*).

Podstawowa idea wykorzystania łańcuchów Markowa do generowania próbek ze skomplikowanego rozkładu p jest więc następująca: tworzymy łańcuch Markowa, dla którego p jest rozkładem stacjonarnym, wówczas rozpoczynając w dowolnym dopuszczalnym stanie początkowym $\boldsymbol{X}_0$ po wykonaniu dużej liczby kroków (etap ten nazywamy okresem przejściowym z

ang. *burn-in period*) stan $\boldsymbol{X}_t$ (dla $t \gg 1$) tego łańcucha będzie w przybliżeniu pochodził z rozkładu granicznego p (nie jest jednak prosto stwierdzić po jak długim okresie przejściowym przybliżenie to jest wystarczająco dobre). Aby otrzymać z takiej procedury próbki prawdziwie i.i.d. każda z próbek musiałaby pochodzić z ponownego uruchomienia takiego łańcucha. Oczywiście jest to nieefektywne, więc w praktyce generujemy próbki z jednego łańcucha po prostu odrzucając pewne z nich tak aby uniknąć znaczących korelacji. Pozostaje pytanie jak skonstruować funkcję przejścia $\pi(\boldsymbol{y} \mid \boldsymbol{x})$ dla danego rozkładu granicznego $p(\boldsymbol{x})$. Podstawową konstrukcję podaje algorytm Metropolisa–Hastingsa.

```
# Metropolis-Hastings
# --------------------
# Initialize state x using any admissable value.
for i in range(max_iter):
    # Step 1. Sample candidate y from the conditional
    # proposal distribution q(y | x).

    # Step 2. Accept new candidate with probability
    r(y | x) = min {1, p(y)q(x|y) / p(x)q(y|x)}
    # and change state to y, otherwise stay in state x.
```

Funkcja przejścia ma zatem postać

$$\pi_{\mathrm{MH}}(\boldsymbol{y} \mid \boldsymbol{x}) = q(\boldsymbol{y} \mid \boldsymbol{x}) r(\boldsymbol{y} \mid \boldsymbol{x}) \,.$$

Pozostaje tylko wykazać, iż spełnione jest kryterium lokalnego balansu. Istotnie mamy

$$\begin{aligned} \pi_{\mathrm{MH}}(\boldsymbol{y} \mid \boldsymbol{x}) p(\boldsymbol{x}) &= \min\{q(\boldsymbol{y} \mid \boldsymbol{x}) p(\boldsymbol{x}), q(\boldsymbol{x} \mid \boldsymbol{y}) p(\boldsymbol{y})\} \\ \pi_{\mathrm{MH}}(\boldsymbol{x} \mid \boldsymbol{y}) p(\boldsymbol{y}) &= \min\{q(\boldsymbol{x} \mid \boldsymbol{y}) p(\boldsymbol{y}), q(\boldsymbol{y} \mid \boldsymbol{x}) p(\boldsymbol{x})\} \end{aligned} \,,$$

skąd $\pi_{\mathrm{MH}}(\boldsymbol{y} \mid \boldsymbol{x}) p(\boldsymbol{x}) = \pi_{\mathrm{MH}}(\boldsymbol{x} \mid \boldsymbol{y}) p(\boldsymbol{y})$. Zauważmy, iż nie musimy znać $p(\boldsymbol{x})$ z dokładnością do stałej normalizującej, gdyż

$$\frac{p(\boldsymbol{y})}{p(\boldsymbol{x})} = \frac{\tilde{p}(\boldsymbol{y})/Z_p}{\tilde{p}(\boldsymbol{x})/Z_p} = \frac{\tilde{p}(\boldsymbol{y})}{\tilde{p}(\boldsymbol{x})} \,.$$

Poza algorytmem Metropolisa–Hastingsa jest wiele innych algorytmów z rodziny MCMC. Większość z nich implementuje konkretny sposób generowania (zostawiając resztę struktury) tak, aby zmniejszyć korelację po okresie przejściowym i przyspieszyć zbieżność. Standardowo wykorzystywanymi algorytmami z tej klasy są algorytmy HMC (*Hamiltonian Monte Carlo*) oraz NUTS (*No U-Turn Sampler*).

2.7 Estymator jądrowy gęstości

W poprzednim paragrafie opisaliśmy w jaki sposób mając (nieznormalizowaną) funkcję gęstości prawdopodobieństwa $\tilde{p}(\boldsymbol{x})$ generować algorytmicznie próbki z opisanego przez nią rozkładu. Teraz zajmiemy się problemem odwrotnym tj. mając realizację prostej próby losowej z pewnego rozkładu chcemy znaleźć funkcję $\hat{p}(\boldsymbol{x})$, która estymuje gęstość rozkładu prawdopodobieństwa, z którego pochodzą próbki. Opiszemy tutaj jedną z najprostszych metod zwaną estymatorem jądrowym (z ang. *Kernel Density Estimator, KDE*). Estymatorem jądrowym gęstości funkcji p nazywamy funkcję

$$\hat{p}(\boldsymbol{x}) := \frac{1}{n}\sum_{i=1}^{n} K_{\boldsymbol{H}}(\boldsymbol{x}-\boldsymbol{x}_i)\,,$$

gdzie $\boldsymbol{H}$ jest symetryczną i dodatnio określoną macierzą zwaną *bandwidth matrix*, funkcja $K_{\boldsymbol{H}}$ ma postać

$$K_{\boldsymbol{H}}(\boldsymbol{x}) = (\det \boldsymbol{H})^{-1/2} K(\boldsymbol{H}^{-1/2}\boldsymbol{x})\,,$$

gdzie funkcja K zwana jądrem (z ang. *kernel*) jest gęstością prawdopodobieństwa pewnego sferycznie symetrycznego rozkładu wielowymiarowego. Wybór funkcji K nie jest kluczowy ze statystycznego punktu widzenia, więc możemy bez problemu założyć, iż jest to gęstość wielowymiarowego rozkładu normalnego, tj.

$$K_{\boldsymbol{H}}(\boldsymbol{x}) = \frac{1}{\sqrt{(2\pi)^m \det \boldsymbol{H}}} \exp\left(-\frac{1}{2}\boldsymbol{x}^T \boldsymbol{H}^{-1}\boldsymbol{x}\right)\,,$$

gdzie zakładamy $\boldsymbol{x} \in \mathbb{R}^m$. Większym problemem w przypadku KDE jest wybór odpowiedniego parametru $\boldsymbol{H}$. Jednym z prostszych wyborów w przypadku estymatora

$$\boxed{\hat{p}(\boldsymbol{x}) = \frac{1}{n\sqrt{(2\pi)^m \det \boldsymbol{H}}} \sum_{i=1}^{n} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{x}_i)^T \boldsymbol{H}^{-1}(\boldsymbol{x}-\boldsymbol{x}_i)\right)}$$

jest tzw. *reguła Silvermana*, która podaje następujący przepis na macierz $\boldsymbol{H}$

$$\boldsymbol{H}^{ij} = 0\,, \quad \sqrt{\boldsymbol{H}^{ii}} = \left(\frac{4}{4+m}\right)^{\frac{1}{m+4}} n^{\frac{-1}{m+4}} \sigma_i\,,$$

gdzie σ_i jest estymatorem wariancji i-tej współrzędnej zmiennej $\boldsymbol{X}$. Inną możliwością jest tzw. *reguła Scotta*

$$\boldsymbol{H}^{ij} = 0\,, \quad \sqrt{\boldsymbol{H}^{ii}} = n^{\frac{-1}{m+4}} \sigma_i\,.$$

KDE w praktycznych zastosowaniach często przyspiesza się za pomocą odpowiednich struktur danych do wyszukiwania najbliższych sąsiadów w przestrzeni $\mathbb{R}^m$, tj. zamiast sumować przyczynki od wszystkich punktów $\boldsymbol{x}_i$ dla danego $\boldsymbol{x}$, znajdujemy jego k najbliższych sąsiadów $\boldsymbol{x}$ ze zbioru $\{\boldsymbol{x}_i\}_{i=1}^n$ stosując np. ANN i obliczamy przyczynki do $\hat{p}(\boldsymbol{x})$ tylko od nich.

3 Podstawy statystycznego uczenia maszynowego

Przechodzimy teraz do zagadnień uczenia maszynowego, w których wykorzystamy przedstawioną wcześniej teorię rachunku prawdopodobieństwa (w szczególności teorię zmiennych losowych) oraz wnioskowania statystycznego.

3.1 Regresja liniowa

Rozpatrujemy teraz zagadnienie regresji tj. predykcji ciągłej wartości $y \in \mathbb{R}$ w zależności od wektora cech $\boldsymbol{x}$. Zakładamy ponadto prosty model liniowy, tj. nasze predykcje będą miały postać

$$\phi(\boldsymbol{x}; \boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{x}$$

dla pewnych estymowanych parametrów $\boldsymbol{w}$. Zauważmy, iż taka postać modelu ϕ uwzględnia człon stały poprzez transformację $\boldsymbol{x} \leftarrow \begin{bmatrix}\boldsymbol{x}^T & 1\end{bmatrix}^T$. W przypadku regresji liniowej parametry estymujemy za pomocą metody MLE dla następującego modelu statystycznego

$$y(\boldsymbol{x}) \mid \boldsymbol{w} \sim \mathcal{N}(\phi(\boldsymbol{x}; \boldsymbol{w}), \sigma^2)\,.$$

Niech $\mathcal{X} = \{y_i(\boldsymbol{x}_i)\}_{i=1}^n$ będzie naszym zbiorem obserwacji i.i.d. Wiarygodność ma zatem postać

$$\mathcal{L}(\mathcal{X}; \boldsymbol{w}) = \frac{1}{(2\pi\sigma^2)^{n/2}} \prod_{i=1}^{n} \exp\left(-\frac{1}{2\sigma^2}\left[y_i - \phi(\boldsymbol{x}_i; \boldsymbol{w})\right]^2\right).$$

Zanegowana logarytmiczna funkcja wiarygodności (funkcjonał kosztu) ma zatem postać

$$\boxed{L(\mathcal{X}; \boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{n}\left[y_i - \phi(\boldsymbol{x}_i; \boldsymbol{w})\right]^2\,,}$$

gdzie pominęliśmy multiplikatywne i addytywne człony stałe, gdyż nie wpływają one na zagadnienie optymalizacji. Minimalizując funkcję L względem $\boldsymbol{w}$ otrzymamy estymatę MLE tych parametrów. Otrzymana funkcja L ma postać formy kwadratowej, a otrzymany problem optymalizacyjny nazywamy metodą najmniejszych kwadratów (z ang. *Ordinary Least Squares, OLS*). Wprowadźmy macierz $\boldsymbol{X}^{ij} := \boldsymbol{x}_i^j$ oraz wektory $\boldsymbol{y}^i = y_i$, $\boldsymbol{\phi}^i = \phi(\boldsymbol{x}_i; \boldsymbol{w}) = \sum_j \boldsymbol{X}^{ij}\boldsymbol{w}^j$. Wówczas możemy zapisać funkcję kosztu jako

$$L(\mathcal{X}; \boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{n}\left(\boldsymbol{y}^i - \boldsymbol{\phi}^i(\boldsymbol{X}; \boldsymbol{w})\right)^2 ,$$

skąd

$$\frac{\partial L}{\partial \boldsymbol{w}^a} = \sum_b \frac{\partial L}{\partial \boldsymbol{\phi}^b}\frac{\partial \boldsymbol{\phi}^b}{\partial \boldsymbol{w}^a} .$$

Jednocześnie

$$\frac{\partial L}{\partial \boldsymbol{\phi}^b} = \boldsymbol{y}^b - \boldsymbol{\phi}^b , \quad \frac{\partial \boldsymbol{\phi}^b}{\partial \boldsymbol{w}^a} = \boldsymbol{X}^{ba} ,$$

skąd

$$\frac{\partial L}{\partial \boldsymbol{w}^a} = \sum_b \boldsymbol{X}^{ba}(\boldsymbol{y}^b - \boldsymbol{\phi}^b) ,$$

co możemy zapisać w kompaktowej formie macierzowej

$$\frac{\partial L}{\partial \boldsymbol{w}} = \boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{\phi}) = \boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}) .$$

Przyrównując powyższe równanie do 0 otrzymujemy następujący wzór na estymatę MLE parametrów $\boldsymbol{w}$

$$\boxed{\boldsymbol{w}_{\mathrm{MLE}} = \left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y} = \boldsymbol{X}^+\boldsymbol{y},}$$

gdzie $\boldsymbol{X}^+$ oznacza *pseudoodwrotność Moore'a–Penrose'a*, którą można efektywnie obliczyć korzystając z rozkładu SVD macierzy $\boldsymbol{X}$.

3.2 Regularyzacja

Regularyzacją nazywamy proces polegający na wprowadzeniu ad hoc do zagadnienia optymalizacji dodatkowych członów tak, aby rozwiązanie było regularne (prostsze, nieosobliwe, jednoznaczne). W przypadku funkcji kosztu L najczęściej dodajemy człon penalizujący rozwiązania o dużej normie estymowanego parametru tj. człon postaci $\lambda\|\boldsymbol{w}\|$ dla pewnej normy $\|\cdot\|$ i

hiperparametru λ zwanego *siłą regularyzacji*. W kontekście bayesowskim regularyzację można również rozumieć jako użycie estymacji MAP zamiast MLE dla odpowiednio dobranego priora nad estymowanymi parametrami.

Jeśli dla modelu regresji liniowej jako człon regularyzujący przyjmiemy $\lambda\|\boldsymbol{w}\|_2^2$, tj. kwadrat normy L2 wektora estymowanych parametrów, to otrzymana funkcja kosztu ma postać

$$\boxed{L(\mathcal{X}; \boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{n}\left[y_i - \phi(\boldsymbol{x}_i; \boldsymbol{w})\right]^2 + \lambda\|\boldsymbol{w}\|_2^2 .}$$

Analogiczną postać można otrzymać rozważając estymatę MAP dla identycznego modelu statystycznego z priorem na wartości parametrów $\boldsymbol{w}$ będącym rozkładem normalnym $\mathcal{N}(0, \tau^2\mathbf{1})$ dla odpowiednio dobranego τ. Dla powyższej funkcji kosztu możemy bez problemu wyznaczyć wartość $\boldsymbol{w}$, która ją minimalizuje. Istotnie korzystając z wyników dla modelu regresji liniowej mamy

$$\boldsymbol{w}_{\text{MAP}} = \left(\lambda\mathbf{1} + \boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y}\,,$$

Zagadnienie minimalizacji funkcji kosztu będącej formą kwadratową z dodanym członem regularyzującym w postaci normy L2 wektora nazywamy *regresją grzbietową* (z ang. *ridge regression*).

Innym przykładem regularyzacji jest tzw. regularyzacja L1, która polega na dodaniu do funkcji kosztu członu postaci $\lambda\sum_{j=1}^{d}|\boldsymbol{w}^j|$ tj. normy L1 wektora wag. Zagadnie optymalizacji formy kwadratowej z członem regularyzującym L1 nazywamy *regresją LASSO*. W takim przypadku nie da się prosto analitycznie znaleźć estymaty punktowej MAP i trzeba używać algorytmów optymalizacji numerycznej. W ogólności można połączyć regularyzacje L1 i L2 tj. rozważać zregularyzowaną funkcję kosztu postaci

$$\boxed{L(\mathcal{X}; \boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{n}\left[y_i - \phi(\boldsymbol{x}_i; \boldsymbol{w})\right]^2 + \lambda_1\|\boldsymbol{w}\|_1 + \lambda_2\|\boldsymbol{w}\|_2^2 .}$$

Zagadnienie minimalizacji takiej funkcji kosztu nazywamy ElasticNet i tak jak w przypadku LASSO musimy korzystać z algorytmów optymalizacji numerycznej. Często wykorzystuje się tutaj algorytmy bezgradientowe np. coordinate descent.

3.3 Regresja kwantylowa

Zastanówmy się wpierw na czym tak naprawdę polega modelowanie rozkładu warunkowego $y(\boldsymbol{x}) \mid \boldsymbol{w}$ za pomocą określonego rozkładu prawdopo-

dobieństwa. Na pierwszy rzut oka może się wydawać, iż takie podejście wprowadza bardzo silne założenia, a co za tym idzie ograniczenia w stosowaniu naszego modelu. Zauważmy jednak, iż w przypadku podejścia typu *likelihood* rozkład jest niejako wybierany w taki sposób, aby jego parametry były użyteczne. Istotnie w przypadku regresji zwykle nie ma jak zweryfikować rzeczywistego rozkładu $y(\boldsymbol{x})$, gdyż mamy tylko po jednej wartości y dla danego $\boldsymbol{x}$. Modelując $y(\boldsymbol{x})$ rozkładem normalnym chodzi nam zatem raczej o to, że w takim modelu chcemy znaleźć parametr (prostą) taką, że masa prawdopodobieństwa punktów po obu stronach prostej jest jednakowa i większość masy jest zgromadzona w bliskiej odległości od prostej (wynika to z kształtu rozkładu normalnego, który nie posiada tzw. ciężkich ogonów)

W takim ujęciu możemy zakładać różne inne rozkłady na $y(\boldsymbol{x})$ jeśli interesują nas proste, które inaczej mają rozdzielać masę prawdopodobieństwa między punkty lub też dopuszczać obserwacje leżące daleko od prostej regresji. Problemem w przypadku rozkładu normalnego jest jego czułość na wartości odstające, gdyż w rozkładzie normalnym ogony tego rozkładu mają stosunkowo niewielką masę prawdopodobieństwa. Chcielibyśmy zatem rozkład z ciężkimi ogonami (z ang. *heavy tails*). Dodatkowo chcielibyśmy mieć rozkład, który pozwala znaleźć prostą, która nie rozkłada masy po równo, ale np. tak, że 90% masy prawdopodobieństwa jest pod nią. Oba te problemy możemy rozwiązać modelując rozkład $y(\boldsymbol{x})$ przez tzw. *asymetryczny rozkład Laplace'a* (z ang. *Asymmetric Laplace Distribution, ALD*).

Definicja 3.1 (Asymetrycznego rozkładu Laplace'a)**.** Mówimy, iż zmienna losowa rzeczywista X ma asymetryczny rozkład Laplace'a, tzn. $X \sim \mathrm{ALD}(m, \lambda, q)$ jeśli jej gęstość wyraża się wzorem

$$p(x; m, \lambda, q) = \frac{q(1-q)}{\lambda} \begin{cases} \mathrm{e}^{-\frac{q-1}{\lambda}(x-m)}, & x \leq m \\ \mathrm{e}^{-\frac{q}{\lambda}(x-m)}, & x \geq m \end{cases} .$$

Zauważmy, że dystrybuanta rozkładu ALD ma postać

$$F(x; m, \lambda, q) = \begin{cases} q\mathrm{e}^{\frac{1-q}{\lambda}(x-m)}, & x \leq m \\ 1-(1-q)\mathrm{e}^{-\frac{q}{\lambda}(x-m)}, & x \geq m \end{cases}$$

zatem parametr q określa rząd kwantyla m. W przypadku regresji możemy zatem modelować wartość $y(\boldsymbol{x})$ przez rozkład ALD dla ustalonego q postaci

$$y(\boldsymbol{x}) \mid \boldsymbol{w}, \lambda \sim \mathrm{ALD}(\phi(\boldsymbol{x}; \boldsymbol{w}), \lambda, q)\,,$$

gdzie λ pełni podobną rolę jak σ w przypadku rozkładu normalnego. Widzimy wówczas, iż estymacja MLE $\boldsymbol{w}$ daje prostą regresji taką, że ułamek $1-q$ masy prawdopodobieństwa znajduje się pod prostą (estymujemy zatem warunkowy kwantyl rzędu q). Zanegowana logarytmiczna funkcja wiarygodności (inaczej funkcja kosztu) dla modelu ALD ma postać

$$\boxed{L(\mathcal{X};\boldsymbol{w},\lambda)=\sum_{i=1}^{n}\left[(q-1)r_i\theta(-r_i)+qr_i\theta(r_i)\right]\,,}$$

gdzie

$$r_i := y_i-\phi(\boldsymbol{x}_i;\boldsymbol{w})$$

są tzw. *rezyduami*, a θ oznacza funkcję skokową Heaviside'a. Taką funkcję kosztu nazywamy *pinball loss*, a otrzymane zagadnienie optymalizacji – regresją kwantylową. Modelowanie $y(\boldsymbol{x})$ za pomocą rozkładu ALD pozwala nam w prosty i „robust" sposób znaleźć niepewność naszych predykcji punktowych, tj. dla danego zagadnienia dopasowujemy trzy modele oparte na ALD dla $q=0.5$ (estymacja punktowa, mediana, odporna na outliery) oraz np. $q=0.1$ i $q=0.9$ będące oszacowaniem niepewności punktowej estymaty. Minimalizację powyższej funkcji kosztu można przedstawić jako zagadnienie programowania liniowego i efektywnie rozwiązać korzystając z odpowiednich solverów. Można również dodać człon regularyzujący w postaci normy L1 wektora estymowanych parametrów.

3.4 Procesy gaussowskie

Jak już wspomnieliśmy macierz kowariancji n–wymiarowej zmiennej losowej $\boldsymbol{x}$ o wartości oczekiwanej $\boldsymbol{\mu}$ jest zdefiniowana jako

$$\boldsymbol{\Sigma}=\mathbb{E}\left[(\boldsymbol{x}-\boldsymbol{\mu})(\boldsymbol{x}-\boldsymbol{\mu})^T\right]\,.$$

Wiemy również, iż macierz ta jest nieujemnie określona. Pokażemy teraz, iż dla każdej nieujemnie określonej macierzy symetrycznej $\boldsymbol{K}$ wymiaru $n\times n$ istnieje n–wymiarowa zmienna losowa o wielowymiarowym rozkładzie normalnym, dla której $\boldsymbol{K}$ jest macierzą kowariancji. Istotnie dla każdej nieujemnie określonej macierzy symetrycznej istnieje macierz $\boldsymbol{L}$ taka, że

$$\boldsymbol{K}=\boldsymbol{L}\boldsymbol{L}^T\,,$$

jest to tzw. *dekompozycja Choleskiego*. Niech $\boldsymbol{z}\sim\mathcal{N}(\boldsymbol{0},\boldsymbol{1})$, wówczas zmienna losowa $\boldsymbol{L}\boldsymbol{z}$ ma rozkład o zerowej wartości oczekiwanej i macierzy kowariancji

$$\mathbb{E}\left[(\boldsymbol{L}\boldsymbol{z})(\boldsymbol{L}\boldsymbol{z})^T\right]=\mathbb{E}\left[\boldsymbol{L}\boldsymbol{z}\boldsymbol{z}^T\boldsymbol{L}^T\right]=\boldsymbol{L}\mathbb{E}[\boldsymbol{z}\boldsymbol{z}^T]\boldsymbol{L}^T=\boldsymbol{L}\boldsymbol{1}\boldsymbol{L}^T=\boldsymbol{K}\,.$$

Powyższe własności wskazują, iż macierze kowariancji można w pewnym sensie utożsamiać z nieujemnie określonymi macierzami symetrycznymi.

Definicja 3.2 (Funkcji kowariancji)**.** Funkcję $k : \mathbb{R}^n \times \mathbb{R}^n \mapsto \mathbb{R}$ taką, że $\forall m \in \mathbb{N} : \forall X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\} \subset \mathbb{R}^n$ macierz

$$k(X, X) = \begin{bmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & k(\boldsymbol{x}_1, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_1, \boldsymbol{x}_m) \\ k(\boldsymbol{x}_2, \boldsymbol{x}_1) & k(\boldsymbol{x}_2, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_2, \boldsymbol{x}_m) \\ \vdots & \vdots & \ddots & \vdots \\ k(\boldsymbol{x}_m, \boldsymbol{x}_1) & k(\boldsymbol{x}_m, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_m, \boldsymbol{x}_m) \end{bmatrix}$$

jest dodatnio określoną macierzą symetryczną nazywamy funkcją kowariancji, jądrem dodatnio określonym (z ang. *positive definite kernel*) lub *jądrem Mercera.*

Dla dwóch zbiorów punktów $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\} \subset \mathbb{R}^n$ i $Y = \{\boldsymbol{y}_1, \ldots, \boldsymbol{y}_s\} \subset \mathbb{R}^n$ i funkcji kowariancji k wprowadzimy oznaczenie

$$k(X, Y) := \begin{bmatrix} k(\boldsymbol{x}_1, \boldsymbol{y}_1) & k(\boldsymbol{x}_1, \boldsymbol{y}_2) & \cdots & k(\boldsymbol{x}_1, \boldsymbol{y}_s) \\ k(\boldsymbol{x}_2, \boldsymbol{y}_1) & k(\boldsymbol{x}_2, \boldsymbol{y}_2) & \cdots & k(\boldsymbol{x}_2, \boldsymbol{y}_s) \\ \vdots & \vdots & \ddots & \vdots \\ k(\boldsymbol{x}_m, \boldsymbol{y}_1) & k(\boldsymbol{x}_m, \boldsymbol{y}_2) & \cdots & k(\boldsymbol{x}_m, \boldsymbol{y}_s) \end{bmatrix}.$$

Poniżej podajemy kilka przykładów funkcji kowariancji

- *Gaussian kernel* dla normy $\|\cdot\|$ i hiper-parametrów a, l (amplituda i skala długości)

$$k(\boldsymbol{x}, \boldsymbol{y}) = a^2 \exp\left\{-\frac{1}{2l^2}\|\boldsymbol{x} - \boldsymbol{y}\|^2\right\}$$

- *Periodic kernel* dla normy $\|\cdot\|$ i hiper-parametrów a, l, p (amplituda, skala długości, okres zmienności)

$$k(\boldsymbol{x}, \boldsymbol{y}) = a^2 \exp\left\{-\frac{2}{l^2}\sin^2\left(\frac{\pi}{p}\|\boldsymbol{x} - \boldsymbol{y}\|\right)\right\}$$

- *White noise kernel* dla hiper-parametru σ

$$k(\boldsymbol{x}, \boldsymbol{y}) = \sigma^2 \delta_{\boldsymbol{x},\boldsymbol{y}}$$

- *Matérn kernel* dla normy $\|\cdot\|$ i hiper-parametrów a, l, ν (amplituda, skala długości, regularność)

$$k(\boldsymbol{x}, \boldsymbol{y}) = a^2 \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu}}{l} \|\boldsymbol{x} - \boldsymbol{y}\| \right)^{\nu} K_\nu \left(\frac{\sqrt{2\nu}}{l} \|\boldsymbol{x} - \boldsymbol{y}\| \right),$$

gdzie $\Gamma(x)$ to funkcja gamma Eulera, a $K_\nu(x)$ to zmodyfikowana funkcja Bessela 2-go rodzaju rzędu ν.

Twierdzenie 3.1 (Własności funkcji kowariancji)**.** Suma lub iloczyn dwóch funkcji kowariancji oraz złożenie funkcji kowariancji z wielomianem o nieujemnych współczynnikach jest również funkcją kowariancji.

Definicja 3.3 (Procesu gaussowskiego)**.** Procesem Gaussowskim (z ang. *Gaussian Process*) nazywamy rodzinę skalarnych zmiennych losowych indeksowanych przez punkty $\boldsymbol{x} \in \mathbb{R}^n$

$$\mathcal{GP} = \{ f_{\boldsymbol{x}} \mid \boldsymbol{x} \in \mathbb{R}^n \}$$

taką że każdy skończony podzbiór $\mathcal{GP}$ ma łącznie wielowymiarowy rozkład normalny tj. dla dowolnego zbioru $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\} \subset \mathbb{R}^n$ zachodzi

$$\begin{bmatrix} f_{\boldsymbol{x}_1} \\ \vdots \\ f_{\boldsymbol{x}_m} \end{bmatrix} \sim \mathcal{N}(\boldsymbol{\mu}_X, \boldsymbol{\Sigma}_X).$$

Zauważmy, iż proces Gaussowski możemy jednoznacznie zdefiniować podając przepisy na parametry $\boldsymbol{\mu}_X$ i $\boldsymbol{\Sigma}_X$ dla dowolnego zbioru X. W praktyce często przyjmujemy $\boldsymbol{\mu}_X = \mathbf{0}$, natomiast przepisem na macierz kowariancji może być zdefiniowana wyżej funkcja kowariancji $k(X, X)$ tj.

$$\begin{bmatrix} f_{\boldsymbol{x}_1} \\ \vdots \\ f_{\boldsymbol{x}_m} \end{bmatrix} \sim \mathcal{N}(\mathbf{0}, k(X, X)).$$

Process Gaussowski daje nam w praktyce rozkład prawdopodobieństwa nad funkcjami $f : \mathbb{R}^n \mapsto \mathbb{R}$, których charakter jest określony przez jądro k (np.

funkcja gładka dla jądra Gaussowskiego, okresowa dla jądra periodycznego, itp.). Zauważmy, że nie wnioskujemy tu o parametrach konkretnej rodziny funkcji (jak w przypadku regresji liniowej); interesuje nas jedynie rozkład predykcyjny. Załóżmy, iż w dokładnie znanych przez nas punktach $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m\}$ zaobserwowaliśmy wartości pewnej funkcji, o których zakładamy, iż pochodzą z procesu Gaussowskiego zadanego jądrem k, które wyraża nasze założenia a priori co do charakteru badanej funkcji

$$\boldsymbol{f}_X = \begin{bmatrix} f_{\boldsymbol{x}_1} \\ \vdots \\ f_{\boldsymbol{x}_m} \end{bmatrix} \sim \mathcal{N}(\mathbf{0}, k(X,X)) \,.$$

Powiedzmy, iż chcemy znać wartości $\boldsymbol{f}_Y$ tej funkcji w zadanych punktach $Y = \{\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_s\}$. Ponieważ założyliśmy, iż wartości funkcji pochodzą z procesu Gaussowskiego, więc rozkład łączny $\boldsymbol{f}_X$ i $\boldsymbol{f}_Y$ jest rozkładem normalnym

$$\begin{bmatrix} \boldsymbol{f}_X \\ \boldsymbol{f}_Y \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} k(X,X) & k(X,Y) \\ k(Y,X) & k(Y,Y) \end{bmatrix}\right) .$$

Zauważmy, iż z twierdzenia o własnościach niezdegenerowanego rozkładu normalnego wnioskujemy, iż warunkowy $\boldsymbol{f}_Y \mid \boldsymbol{f}_X$ jest również rozkładem normalnym o parametrach

$$\begin{aligned} \boldsymbol{\mu} &= k(Y,X)k^{-1}(X,X)\boldsymbol{f}_X \\ \boldsymbol{\Sigma} &= k(Y,Y) - k(Y,X)k^{-1}(X,X)k(X,Y) \end{aligned} \,.$$

Dodatkową niepewność związaną z pomiarem wartości $\boldsymbol{f}_X$ możemy uchwycić zmieniając postać jądra

$$k(\boldsymbol{x}, \boldsymbol{y}) \leftarrow k(\boldsymbol{x}, \boldsymbol{y}) + \mathcal{I}_X(\boldsymbol{x})\sigma^2\delta_{\boldsymbol{x},\boldsymbol{y}} \,,$$

gdzie σ jest hiper-parametrem określającym precyzję pomiaru. Oczywiście k jest dalej funkcją kowariancji, gdyż takie podstawienie powoduje jedynie dodanie dodatnich członów do pewnych elementów diagonalnych macierzy kowariancji, więc macierz ta jest nadal symetryczna i dodatnio określona. Wówczas rozkład predykcyjny ma parametry

$$\boxed{\begin{aligned} \boldsymbol{\mu} &= k(Y,X)\left[k(X,X) + \sigma^2\mathbf{1}\right]^{-1}\boldsymbol{f}_X \\ \boldsymbol{\Sigma} &= k(Y,Y) - k(Y,X)\left[k(X,X) + \sigma^2\mathbf{1}\right]^{-1}k(X,Y) \end{aligned}} \,.$$

3.5 Naiwny klasyfikator bayesowski

Przechodzimy teraz do zagadnienia klasyfikacji, tj. predykcji zmiennej t pochodzącej ze skończonego zbioru kategorii (inaczej klas) dla danego wektora cech $\boldsymbol{x}$. Będziemy rozważać jedynie modele probabilistyczne, których wyjściem jest zbiór prawdopodobieństw $p_i(\boldsymbol{x})$ dla $i = 1, \ldots, k$, gdzie k jest liczbą klas, a $p_i(\boldsymbol{x})$ określa prawdopodobieństwo, iż prawidłową klasą dla wektora $\boldsymbol{x}$ jest i-ta klasa. Oczywiście musi zachodzić

$$\sum_j p_j(\boldsymbol{x}) = 1$$

dla dowolnego $\boldsymbol{x}$. Przewidywaną klasą zostaje następnie ta z największym prawdopodobieństwem. W przypadku klasyfikacji binarnej są natomiast możliwe inne reguły, np. przewidywaną klasą jest 0 jeśli $p_0 \geq p_{\text{threshold}}$ dla pewnego ustalonego progu niekoniecznie równego 0.5.

Pierwszym modelem, który rozważymy będzie bardzo prosty model bayesowski. Korzystając z twierdzenia Bayesa możemy zapisać

$$p_i(\boldsymbol{x}) = p(c_i \mid \boldsymbol{x}) = \frac{p(\boldsymbol{x} \mid c_i)p(c_i)}{\sum_j p(\boldsymbol{x} \mid c_j)p(c_j)}\,.$$

Jeśli $\mathcal{X} = \{t_i(\boldsymbol{x}_i)\}_{i=1}^n$ oznacza nasz zbiór obserwacji i.i.d. to gęstości prawdopodobieństwa $p(\boldsymbol{x} \mid c_i)$ i $p(c_i)$ możemy oszacować z danych $\mathcal{X}$. W przypadku drugiego członu możemy go prosto oszacować jako

$$p(c_i) = \frac{|\{t \mid t = c_i, t \in \mathcal{X}\}|}{|\mathcal{X}|}\,.$$

W przypadku pierwszego członu sytuacja jest bardziej skomplikowana. Możemy założyć pewien rozkład parametryczny dla rozkładu warunkowego $p(\boldsymbol{x} \mid c_i)$ dla każdej klasy i następnie oszacować parametry tego rozkładu z danych. Możemy również wykorzystać KDE i estymować rozkłady w sposób nieparametryczny. Oba te podejścia mają jednak problem związany z tzw. *klątwą wymiaru* (z ang. *curse of dimensionality*). Istotnie przestrzenie o dużej liczbie wymiarów są bardzo „puste", tj. aby wyczerpująco wypróbkować taką przestrzeń potrzebujemy wykładniczo wielu punktów. Jedną z możliwości poradzenia sobie z klątwą jest wprowadzenie naiwnego założenia o warunkowej niezależności cech w wektorze $\boldsymbol{x} = \left[x_1, \ldots, x_d\right]^T$. Wówczas gęstości prawdopodobieństw $p(\boldsymbol{x} \mid c_i)$ można zapisać jako

$$p(\boldsymbol{x} \mid c_i) = p(x_1, \ldots, x_d \mid c_i) = \prod_{j=1}^{d} p(x_j \mid c_i)\,.$$

Teraz estymacja rozkładów warunkowych $p(x_j \mid c_i)$ dla każdej z klasy jest o wiele prostsza i możemy wykorzystać jedno ze wspomnianych podejść, tj. albo estymację parametrów pewnego rozkładu, albo KDE. Przewidywaną klasą dla wektora cech zostaje więc

$$t(\boldsymbol{x}) = \arg \max_{c_i \in \{c_1,\ldots,c_k\}} p_i(\boldsymbol{x}) = \arg \max_{c_i \in \{c_1,\ldots,c_k\}} p(c_i) \prod_{j=1}^{d} p(x_j \mid c_i)\,.$$

3.6 Klasyfikator najbliższych sąsiadów

Klasyfikator najbliższych sąsiadów (z ang. *k Nearest Neighbors*) jest jednym z najprostszych klasyfikatorów, w którym prawdopodobieństwo $p_i(\boldsymbol{x})$ jest dane wzorem

$$p_i(\boldsymbol{x}) = \frac{1}{\kappa} \sum_{\boldsymbol{x}' \in N(\boldsymbol{x};\kappa,d)} [t(\boldsymbol{x}') = c_i]\,,$$

gdzie $N(\boldsymbol{x};\kappa,d)$ jest zbiorem κ najbliższych sąsiadów punktu $\boldsymbol{x}$ ze zbioru obserwacji $\mathcal{X}$ względem ustalonej metryki, półmetryki lub podobieństwa d, a $[\cdot]$ oznacza nawias Iversona. Metodę najbliższych sąsiadów można również wykorzystać do regresji, gdzie zamiast sumowania wartości nawiasów Iversona, sumujemy wartości zmiennej objaśnianej dla sąsiadów punktu $\boldsymbol{x}$ ze zbioru $\mathcal{X}$ (powoduje to, że model taki potrafi tylko interpolować wartości, więc nie jest dobrym modelem dla regresji).

Klasyfikator kNN jest bardzo elastycznym modelem z nieliniową granicą decyzyjną. Jakość klasyfikacji silnie zależy od lokalnej gęstości punktów w przestrzeni oraz wybranej wartości κ, będącej hiperparametrem tego modelu. Ogólnie małe κ powoduje, że kNN ma duży variance i dość „poszarpaną" granicę decyzyjną, natomiast duże κ powoduje, że kNN ma duży bias i „gładką" granicę decyzyjną. Typowo wykorzystywane funkcje d to metryka euklidesowa, półmetryka euklidesowa, metryka Manhattan i podobieństwo cosinusowe.

Jedną z modyfikacji klasyfikatora kNN, który może polepszyć wyniki w przypadku, gdy w naszej przestrzeni istnieją obszary, w których mamy małą gęstość punktów ze zbioru $\mathcal{X}$ jest *ważenie sąsiadów*, które polega na tym, iż prawdopodobieństwa danej klasy wśród κ sąsiadów obliczamy teraz jako średnią ważoną, gdzie wagą jest odwrotność odległości danego sąsiada od nowego wektora cech $w(\boldsymbol{x},\boldsymbol{x}') = 1/d(\boldsymbol{x},\boldsymbol{x}')$, tj.

$$p_i(\boldsymbol{x}) = \frac{\sum_{\boldsymbol{x}' \in N(\boldsymbol{x};\kappa,d)} w(\boldsymbol{x},\boldsymbol{x}')[t(\boldsymbol{x}') = c_i]}{\sum_{\boldsymbol{x}' \in N(\boldsymbol{x};\kappa,d)} w(\boldsymbol{x},\boldsymbol{x}')}\,.$$

W przypadku naiwnego kNN podczas treningu zapamiętujemy jedynie zbiór $\mathcal{X}$ natomiast naiwna implementacja predykcji ma złożoność czasową $O(\kappa nm)$, gdzie m jest wymiarem wektora cech, przy założeniu, że złożoność obliczenia funkcji d dla pary punktów ma złożoność $O(m)$. Jest to nieakceptowalna złożoność, gdyż zwykle chcemy używać klasyfikatora kNN do setek milionów punktów. Dwa podejścia, które stosuje się zwykle do rozwiązania tego problemu to:

- zbudowanie odpowiedniej struktury danych w fazie treningu, aby w czasie predykcji można było szybciej znajdywać najbliższych sąsiadów (np. k-d tree, ball tree);
- wykorzystanie algorytmów aproksymacyjnych (z ang. *Approximate Nearest Neighbors, ANN*) do znajdowania sąsiadów, którzy niekoniecznie naprawdę są najbliżsi, ale aproksymacja jest wystarczająco dobra do praktycznych zastosowań.

Obecnie to drugie podejście jest dominujące i wykorzystywane na przykład w przypadku dużych modeli językowych do wyszukiwania kontekstów dla danych zapytań (z ang. *Retrieval Augmented Generation*).

3.7 Regresja logistyczna

Rozpatrzymy teraz problem klasyfikacji binarnej z perspektywy estymacji MLE parametrów. Niech $\mathcal{X} = \{t_i(\boldsymbol{x}_i)\}_{i=1}^n$ będzie zbiorem obserwacji i.i.d., gdzie zakładamy, iż $t \in \{0, 1\}$. Jako model statystyczny przyjmiemy, iż klasa $t(\boldsymbol{x})$ pochodzi z rozkładu Bernoulliego z parametrem $\pi(\boldsymbol{x}; \boldsymbol{w})$ (prawdopodobieństwem klasy pozytywnej, zwanym również funkcją wiążącą, z ang. *link function*) zależnym od estymowanych parametrów $\boldsymbol{w}$

$$t(\boldsymbol{x}) \mid \boldsymbol{w} \sim \mathrm{Ber}(\pi(\boldsymbol{x}; \boldsymbol{w}))\,.$$

W przypadku regresji logistycznej jako funkcję $\pi(\boldsymbol{x}; \boldsymbol{w})$ przyjmiemy $\sigma(\phi(\boldsymbol{x}; \boldsymbol{w}))$, gdzie

$$\sigma(z) = \frac{1}{1 + \mathrm{e}^{-z}}$$

to tzw. *funkcja logistyczna* (będąca szczególnym przypadkiem funkcji sigmoidalnej), natomiast $\phi(\boldsymbol{x}; \boldsymbol{w})$ może być dowolną funkcją wykorzystywaną do zagadnienia regresji. W przypadku regresji logistycznej przyjmujemy prosty model liniowy

$$\phi(\boldsymbol{x}; \boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{x}\,.$$

Ponownie zauważmy, iż taka postać uwzględnia człon stały poprzez podstawienie $\boldsymbol{x} \leftarrow \begin{bmatrix}\boldsymbol{x}^T & 1\end{bmatrix}^T$. Funkcja wiarygodności dla powyższego modelu statystycznego ma postać

$$\mathcal{L}(\mathcal{X}; \boldsymbol{w}) = \prod_{i=1}^{n} \pi(\boldsymbol{x}_i; \boldsymbol{w})^{t_i} (1 - \pi(\boldsymbol{x}_i; \boldsymbol{w}))^{1-t_i} \,,$$

skąd funkcja kosztu

$$\begin{aligned} L(\mathcal{X}; \boldsymbol{w}) &= -\sum_{i=1}^{n} \left[t_i \log \pi(\boldsymbol{x}_i; \boldsymbol{w}) + (1 - t_i) \log(1 - \pi(\boldsymbol{x}_i; \boldsymbol{w})) \right] \\ &= -\sum_{i=1}^{n} \left[t_i \log \sigma(\phi(\boldsymbol{x}_i; \boldsymbol{w})) + (1 - t_i) \log(1 - \sigma(\phi(\boldsymbol{x}_i; \boldsymbol{w}))) \right] \end{aligned} \,.$$

Taką funkcję kosztu nazywamy *entropią krzyżową* (z ang. *cross-entropy function*). Niestety dla takiej postaci funkcji kosztu nie można znaleźć minimum w postaci analitycznej, dlatego musimy wykorzystywać algorytmy optymalizacji numerycznej, które najczęściej wykorzystują pierwsze pochodne funkcji kosztu po parametrach. Wyznaczymy więc jeszcze pochodną funkcji L po parametrach $\boldsymbol{w}$. Wprowadzając macierz $\boldsymbol{X}^{ij} = \boldsymbol{x}_i^j$ oraz wektory $\boldsymbol{t}^i = t_i$, $\boldsymbol{\phi}^i = \phi(\boldsymbol{x}_i; \boldsymbol{w}) = \sum_j \boldsymbol{X}^{ij} \boldsymbol{w}^j$, $\boldsymbol{\sigma}^i = \sigma(\boldsymbol{\phi}^i)$ możemy zapisać

$$L(\mathcal{X}; \boldsymbol{w}) = -\sum_{i=1}^{n} \left[\boldsymbol{t}^i \log \boldsymbol{\sigma}^i + (1 - \boldsymbol{t}^i) \log\left(1 - \boldsymbol{\sigma}^i\right) \right] \,.$$

W takim razie mamy

$$\frac{\partial L}{\partial \boldsymbol{\phi}^a} = \sum_b \frac{\partial L}{\partial \boldsymbol{\sigma}^b} \frac{\partial \boldsymbol{\sigma}^b}{\partial \boldsymbol{\phi}^a} \,,$$

gdzie

$$\frac{\partial L}{\partial \boldsymbol{\sigma}^b} = -\sum_i \left[\frac{\boldsymbol{t}^i}{\boldsymbol{\sigma}^i} \delta_{ib} - \frac{1 - \boldsymbol{t}^i}{1 - \boldsymbol{\sigma}^i} \delta_{ib} \right] = \frac{1 - \boldsymbol{t}^b}{1 - \boldsymbol{\sigma}^b} - \frac{\boldsymbol{t}^b}{\boldsymbol{\sigma}^b}$$

oraz

$$\frac{\partial \boldsymbol{\sigma}^b}{\partial \boldsymbol{\phi}^a} = \sigma'(\boldsymbol{\phi}^b) \delta_{ab} \,, \quad \sigma'(z) = \frac{\mathrm{e}^{-z}}{(1 + \mathrm{e}^{-z})^2} = \sigma(z)(1 - \sigma(z))$$

zatem

$$\frac{\partial L}{\partial \boldsymbol{\phi}^a} = \left(\frac{1 - \boldsymbol{t}^a}{1 - \boldsymbol{\sigma}^a} - \frac{\boldsymbol{t}^a}{\boldsymbol{\sigma}^a} \right) \boldsymbol{\sigma}^a (1 - \boldsymbol{\sigma}^a) = \boldsymbol{\sigma}^a - \boldsymbol{t}^a \,.$$

W takim razie, z powyższego mamy

$$\frac{\partial L}{\partial \boldsymbol{w}^a} = \sum_b \frac{\partial L}{\partial \boldsymbol{\phi}^b} \frac{\partial \boldsymbol{\phi}^b}{\partial \boldsymbol{w}^a} = \sum_b \boldsymbol{X}^{ba}(\boldsymbol{\sigma}^b - \boldsymbol{t}^b)\,,$$

co możemy zapisać w zwartej postaci macierzowej

$$\frac{\partial L}{\partial \boldsymbol{w}} = \boldsymbol{X}^T(\boldsymbol{\sigma} - \boldsymbol{t})\,.$$

3.8 Regresja softmax

Regresję logistyczną możemy w prosty sposób uogólnić na przypadek klasyfikacji jednej z k klas. Zakładamy teraz, iż zbiór obserwacji i.i.d. ma postać $\mathcal{X} = \{t_i(\boldsymbol{x}_i)\}_{i=1}^n$ dla $t \in \{1, \ldots, k\}$. Nasz model statystyczny uogólnimy do postaci rozkładu kategorialnego (z ang. *categorical distribution* lub *multinomial distribution*) z parametrem $\boldsymbol{\pi}(\boldsymbol{x}; \boldsymbol{W})$ (wektorem prawdopodobieństw każdej klasy) zależnym od estymowanych parametrów $\boldsymbol{W}$

$$t(\boldsymbol{x}) \mid \boldsymbol{W} \sim \mathrm{Cat}(\boldsymbol{\pi}(\boldsymbol{x}; \boldsymbol{W}))\,.$$

W przypadku regresji softmax jako funkcję $\boldsymbol{\pi} : \mathbb{R}^m \mapsto [0;1]^k$ przyjmiemy funkcję $\boldsymbol{\sigma}(\boldsymbol{\phi}(\boldsymbol{x}; \boldsymbol{W}))$, gdzie $\boldsymbol{\sigma} : \mathbb{R}^k \mapsto [0;1]^k$

$$\boldsymbol{\sigma}^i(\boldsymbol{z}) = \frac{\exp(\boldsymbol{z}^i)}{\sum_{j=1}^k \exp(\boldsymbol{z}^j)}$$

to tzw. *funkcja softmax*, natomiast $\boldsymbol{\phi} : \mathbb{R}^m \mapsto \mathbb{R}^k$, to dowolna funkcja. W przypadku regresji softmax przyjmiemy prosty model liniowy

$$\boldsymbol{\phi}(\boldsymbol{x}; \boldsymbol{W}) = \boldsymbol{W}\boldsymbol{x}\,.$$

Funkcja wiarygodności dla powyższego modelu ma postać

$$\mathcal{L}(\mathcal{X}; \boldsymbol{W}) = \prod_{i=1}^n \prod_{j=1}^k \boldsymbol{\pi}^j(\boldsymbol{x}_i; \boldsymbol{W})^{[t_i = j]}\,,$$

skąd

$$\boxed{\begin{aligned} L(\mathcal{X}; \boldsymbol{W}) &= -\sum_{i=1}^n \sum_{j=1}^k [t_i = j] \log \boldsymbol{\pi}^j(\boldsymbol{x}_i; \boldsymbol{W}) \\ &= -\sum_{i=1}^n \sum_{j=1}^k [t_i = j] \log \boldsymbol{\sigma}^j(\boldsymbol{\phi}(\boldsymbol{x}_i; \boldsymbol{W})) \end{aligned}}\,.$$

Niestety dla takiej postaci funkcji kosztu nie można znaleźć minimum w postaci analitycznej, dlatego musimy wykorzystywać algorytmy optymalizacji numerycznej. Wyznaczymy więc jeszcze pierwsze pochodne funkcji L po parametrach $\boldsymbol{W}$. Wprowadźmy macierze

$$\begin{aligned}
\boldsymbol{T}^{ij} &= [t_i = j] \\
\boldsymbol{X}^{ij} &= \boldsymbol{x}_i^j \\
\boldsymbol{\Phi}^{ij} &= \phi^j(\boldsymbol{x}_i; \boldsymbol{W}) = \sum_l \boldsymbol{W}^{jl} \boldsymbol{X}^{il} \\
\boldsymbol{S}^{ij} &= \boldsymbol{\sigma}^j(\boldsymbol{\phi}(\boldsymbol{x}_i; \boldsymbol{W})) = \frac{\exp\left(\boldsymbol{\Phi}^{ij}\right)}{\sum_{l=1}^k \exp(\boldsymbol{\Phi}^{il})}
\end{aligned} ,$$

gdzie każdy wiersz macierzy $\boldsymbol{X}$ jest wektorem cech danego przykładu; każdy wiersz $\boldsymbol{T}$ jest tzw. *wektorem one–hot* dla danego przykładu tj. wektorem binarnym, w którym dokładnie na jednej pozycji jest wartość 1 i pozycja ta odpowiada prawidłowej klasie dla danego przykładu; każdy wiersz macierzy $\boldsymbol{\Phi}$ jest wektorem *mlogitów* tj. liczb rzeczywistych, które po zastosowaniu funkcji softmax dają wartości prawdopodobieństwa każdej klasy. Wówczas

$$L(\mathcal{X}; \boldsymbol{W}) = -\sum_{i=1}^n \sum_{j=1}^k \boldsymbol{T}^{ij} \log \boldsymbol{S}^{ij} ,$$

skąd

$$\frac{\partial L}{\partial \boldsymbol{\Phi}^{ab}} = \sum_{c,d} \frac{\partial L}{\partial \boldsymbol{S}^{cd}} \frac{\partial \boldsymbol{S}^{cd}}{\partial \boldsymbol{\Phi}^{ab}} ,$$

gdzie

$$\frac{\partial L}{\partial \boldsymbol{S}^{cd}} = -\sum_{i,j} \frac{\boldsymbol{T}^{ij}}{\boldsymbol{S}^{ij}} \delta_{ic} \delta_{jd} = -\frac{\boldsymbol{T}^{cd}}{\boldsymbol{S}^{cd}}$$

oraz

$$\begin{aligned}
\frac{\partial \boldsymbol{S}^{cd}}{\partial \boldsymbol{\Phi}^{ab}} &= \exp\left(\boldsymbol{\Phi}^{cd}\right) \frac{\delta_{ac}\delta_{bd} \left[\sum_l \exp\left(\boldsymbol{\Phi}^{cl}\right)\right] - \delta_{ac} \exp\left(\boldsymbol{\Phi}^{cb}\right)}{\left[\sum_l \exp(\boldsymbol{\Phi}^{cl})\right]^2} \\
&= \boldsymbol{S}^{cd} \delta_{ac} \delta_{bd} - \delta_{ac} \boldsymbol{S}^{cd} \boldsymbol{S}^{cb}
\end{aligned} .$$

Z powyższego

$$\frac{\partial L}{\partial \boldsymbol{\Phi}^{ab}} = \sum_{c,d} \frac{\boldsymbol{T}^{cd}}{\boldsymbol{S}^{cd}} \left(\delta_{ac} \boldsymbol{S}^{cd} \boldsymbol{S}^{cb} - \boldsymbol{S}^{cd} \delta_{ac} \delta_{bd}\right) = \boldsymbol{S}^{ab} - \boldsymbol{T}^{ab} ,$$

gdzie skorzystaliśmy z faktu, iż z konstrukcji dla dowolnego wiersza a macierzy $\boldsymbol{T}$ zachodzi $\sum_d \boldsymbol{T}^{ad} = 1$. W takim razie

$$\frac{\partial L}{\partial \boldsymbol{W}^{ab}} = \sum_{c,d} \frac{\partial L}{\partial \boldsymbol{\Phi}^{cd}} \frac{\partial \boldsymbol{\Phi}^{cd}}{\partial \boldsymbol{W}^{ab}} = \sum_c \left(\boldsymbol{S}^{ca} - \boldsymbol{T}^{ca}\right) \boldsymbol{X}^{cb} ,$$

co możemy zapisać w zwartej postaci macierzowej

$$\frac{\partial L}{\partial \boldsymbol{W}} = (\boldsymbol{S} - \boldsymbol{T})^T \boldsymbol{X} .$$

4 Uczenie głębokie i sieci neuronowe

> [*Introduction, Ch. 1* [1]] Artificial intelligence, or AI, is concerned with building systems that simulate intelligent behavior. It encompasses a wide range of approaches, including those based on logic, search, and probabilistic reasoning. Machine learning is a subset of AI that learns to make decisions by fitting mathematical models to observed data. This area has seen explosive growth and is now (incorrectly) almost synonymous with the term AI.
>
> A deep neural network is a type of machine learning model, and when it is fitted to data, this is referred to as deep learning.

4.1 Sieci w pełni połączone

Neuron modelujemy jako obiekt realizujący przekształcenie $f : \mathbb{R}^n \mapsto \mathbb{R}$

$$f(\boldsymbol{x}; \boldsymbol{w}, b) = a\left(\boldsymbol{w}^T \boldsymbol{x} + b\right) ,$$

gdzie $a : \mathbb{R} \mapsto \mathbb{R}$ jest pewnym nieliniowym przekształceniem. Model ten ma raczej niewiele wspólnego ze współczesnymi modelami matematycznymi biologicznych neuronów, ale jest o nie w luźny sposób oparty. W szczególności wyjście neuronu zależy od kombinacji liniowej wejść $\boldsymbol{x}$ (tzw. *preaktywacji*) oraz odpowiedniej nieliniowej funkcji aktywacji (z ang. *activation function*). Jako funkcje a najczęściej przyjmujemy funkcje ReLU (z ang. *Rectified Linear Unit*) lub podobne – SiLU (z ang. *Sigmoid Linear Unit*), GELU (z ang. *Gaussian Error Linear Unit*)

$$\mathrm{ReLU}(x) = \max(0, x) = x\theta(x) , \quad \mathrm{SiLU}(x) = x\sigma(x) , \quad \mathrm{GELU}(x) = x\Phi(x) ,$$

gdzie $\theta(x)$ jest funkcją skokową Heaviside'a, $\sigma(x)$ jest logistyczną funkcją sigmoidalną, a $\Phi(x)$ jest dystrybuantą standardowego rozkładu normalnego.

Neurony łączymy następnie w sieci, w których wyjście danego neuronu jest wejściem innego. Sieci takie realizują pewne przekształcenia $\boldsymbol{\phi} : \mathbb{R}^{n_{\text{in}}} \mapsto \mathbb{R}^{n_{\text{out}}}$, które są parametryzowane przez *wagi* w na krawędziach łączących neurony oraz obciążenia b (z ang. *bias*). Sieć taka reprezentuje zatem pewien graf obliczeń.

Najprostszą siecią, którą najpierw opiszemy będzie sieć w pełni połączona (z ang. *fully connected network, FCN*) zwana również ze względów historycznych perceptronem wielowarstwowym (z ang. *Multilayer Perceptron, MLP*). W sieci takiej neurony ułożone są w warstwy, przy czym połączenia występują tylko między neuronami w sąsiednich warstwach i nie ma połączeń między neuronami w tej samej warstwie (warstwy takie nazywamy w pełni połączonym – *fully connected layer, FC*). Przykładową sieć w pełni połączoną przedstawiono na rysunku 2. Na rysunku krawędzie grafu reprezentują wagi sieci, a wierzchołki reprezentują wartości funkcji aktywacji. Wierzchołki w pierwszej i ostatniej warstwie zawierają wartości odpowiednio danych wejściowych i wyjściowych i zakładamy, iż dla nich funkcja aktywacji jest funkcją identycznościową. Warstwy te nazywamy *warstwami widocznymi*, natomiast pozostałe – *warstwami ukrytymi* (z ang. *hidden layers*).

Niech $\boldsymbol{L} : \mathbb{R}^n \mapsto \mathbb{R}^m$ oznacza przekształcenie afiniczne postaci

$$\boldsymbol{L}^\mu(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{b}) = \boldsymbol{b}^\mu + \sum_{\nu=1}^{n} \boldsymbol{W}^{\mu\nu} \boldsymbol{x}^\nu \,,$$

gdzie $\boldsymbol{W} \in \mathbb{R}^{m \times n}, \boldsymbol{b} \in \mathbb{R}^m$. Dodatkowo przez złożenie funkcji skalarnej $a : \mathbb{R} \mapsto \mathbb{R}$ z funkcją $\boldsymbol{L}$ będziemy rozumieć funkcję $a \circ \boldsymbol{L} : \mathbb{R}^n \mapsto \mathbb{R}^m$

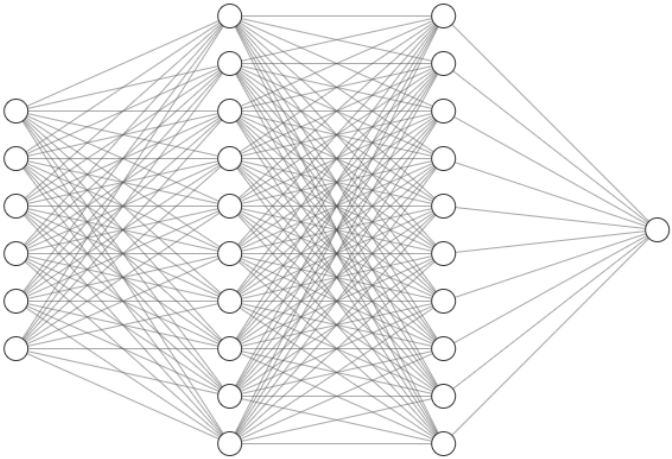
$$(a \circ \boldsymbol{L})^\mu(\boldsymbol{x}) = a\left(\boldsymbol{L}^\mu(\boldsymbol{x})\right) = a\left(\boldsymbol{b}^\mu + \sum_{\nu=1}^{n} \boldsymbol{W}^{\mu\nu} \boldsymbol{x}^\nu\right) .$$

Sieć w pełni połączona złożona z $k{+}2$ warstw o rozmiarach $n_{\text{in}}, n_1, \ldots, n_k, n_{\text{out}}$ realizuje zatem przekształcenie $\boldsymbol{\phi} : \mathbb{R}^{n_{\text{in}}} \mapsto \mathbb{R}^{n_{\text{out}}}$

$$\boxed{\boldsymbol{\phi}\left(\boldsymbol{x}; \{(\boldsymbol{W}_i, \boldsymbol{b}_i)\}_{i=1}^{k+1}\right) = \left(\boldsymbol{L}_{k+1} \circ a \circ \boldsymbol{L}_k \ldots a \circ \boldsymbol{L}_2 \circ a \circ \boldsymbol{L}_1\right)(\boldsymbol{x}) \,,}$$

gdzie $\boldsymbol{L}_i : \mathbb{R}^{n_{i-1}} \mapsto \mathbb{R}^{n_i}$ (przy czym $n_0 = n_{\text{in}}$ i $n_{k+1} = n_{\text{out}}$)

$$\boxed{\boldsymbol{L}_i^\mu\left(\boldsymbol{x}; \boldsymbol{W}_i, \boldsymbol{b}_i\right) = \boldsymbol{b}_i^\mu + \sum_{\nu=1}^{n_{i-1}} \boldsymbol{W}_i^{\mu\nu} \boldsymbol{x}^\nu \,,}$$



Rysunek 2: Przykładowa sieć w pełni połączona

$\boldsymbol{W}_i \in \mathbb{R}^{n_i \times n_{i-1}}, \boldsymbol{b}_i \in \mathbb{R}^{n_i}$ (dla uproszczenia zapisu w miejscach, gdzie nie prowadzi to do niejasności będziemy oznaczać zbiór parametrów sieci $\{(\boldsymbol{W}_i, \boldsymbol{b}_i)\}_{i=1}^{k+1}$ po prostu jako $\boldsymbol{W}$). Okazuje się, iż taka postać rodziny funkcji parametryzowanych przez wagi i obciążenia $\boldsymbol{W}_i, \boldsymbol{b}_i$ potrafi aproksymować niemal dowolną funkcję (*Universal Approximation Theorem*), zatem jest to niemal uniwersalny aproksymator, który możemy zastosować do wszelkich omawianych wcześniej zagadnień regresji i klasyfikacji. Zauważmy, iż już wcześniej rozwinęliśmy teorię w sposób dość ogólny wyprowadzając większość wyników dla pewnej dowolnej funkcji $\boldsymbol{\phi}(\boldsymbol{x}; \boldsymbol{W})$ i jedynie na końcu znajdując dokładne rozwiązania dla prostego odwzorowania liniowego $\boldsymbol{W}\boldsymbol{x}$ (na marginesie takie proste odwzorowanie odpowiada płytkiej sieci neuronowej o $k = 0$ warstwach ukrytych).

W powyższej postaci sieć MLP przetwarza pojedynczo każdy wektor $\boldsymbol{x}$ należący do zbioru danych $\mathcal{X} = \{y_i(\boldsymbol{x}_i)\}_{i=1}^{n}$, a najbardziej kosztowną operacją jest mnożenie wektora przez macierz. Jest to jednak dość nieefektywne, gdyż na współczesnym sprzęcie mamy bardzo dobrze zoptymalizowane implementacje mnożenia macierzy przez siebie. Możemy jednak rozszerzyć odwzorowania $\boldsymbol{L}$ i $\boldsymbol{\phi}$ w taki sposób, aby sieć przetwarzała od razu cały

batch (tj. pewien podzbiór $\mathcal{X} \supseteq \mathcal{B} = \{y_\beta(\boldsymbol{x}_\beta)\}$) przykładów uczących reprezentowany przez macierz $\boldsymbol{X}^{\beta\mu} = \boldsymbol{x}_\beta^\mu$ (stosujemy tutaj tzw. konwencję *batch first* zgodnie z którą pierwszy indeks odpowiada kolejnym przykładom z batcha). Możemy zatem uogólnić postać przekształcenia $\boldsymbol{\phi}$ jakie realizuje sieć do postaci $\boldsymbol{\Phi} : \mathbb{R}^{n_\mathrm{b} \times n_\mathrm{in}} \mapsto \mathbb{R}^{n_\mathrm{b} \times n_\mathrm{out}}$, gdzie $n_\mathrm{b} = |\mathcal{B}|$

$$\boldsymbol{\Phi}\left(\boldsymbol{X}; \{(\boldsymbol{W}_i, \boldsymbol{b}_i)\}_{i=1}^{k+1}\right) = (\boldsymbol{L}_{k+1} \circ a \circ \boldsymbol{L}_k \ldots a \circ \boldsymbol{L}_2 \circ a \circ \boldsymbol{L}_1)(\boldsymbol{X}),$$

gdzie $\boldsymbol{L}_i : \mathbb{R}^{n_\mathrm{b} \times n_{i-1}} \mapsto \mathbb{R}^{n_\mathrm{b} \times n_i}$

$$\boldsymbol{L}_i^{\beta\mu}(\boldsymbol{X}; \boldsymbol{W}_i, \boldsymbol{b}_i) = \boldsymbol{b}_i^\mu + \sum_{\nu=1}^{n_{i-1}} \boldsymbol{W}_i^{\mu\nu} \boldsymbol{X}^{\beta\nu},$$

$\boldsymbol{W}_i \in \mathbb{R}^{n_i \times n_{i-1}}, \boldsymbol{b}_i \in \mathbb{R}^{n_i}$. Przez złożenie funkcji skalarnej $a : \mathbb{R} \mapsto \mathbb{R}$ z funkcją $\boldsymbol{L}$ nadal rozumiemy

$$(a \circ \boldsymbol{L})^{\beta\mu}(\boldsymbol{X}) = a(\boldsymbol{L}^{\beta\mu}(\boldsymbol{X})).$$

Widzimy zatem, iż tak naprawdę w takim sformułowaniu sieć przetwarza n_b przykładów równolegle zamiast przetwarzać je sekwencyjnie pojedynczo, co nie zmienia nic koncepcyjnie, a jedynie sprawia, iż sieć przetwarza dane bardziej efektywnie.

[*Notes, Ch. 4* [1]]

- **Deep learning**: It has long been understood that it is possible to build more complex functions by composing shallow neural networks or developing networks with more than one hidden layer. Indeed, the term "deep learning" was first used by Dechter (1986). However, interest was limited due to practical concerns; it was not possible to train such networks well. The modern era of deep learning was kick-started by startling improvements in image classification reported by [Alex] Krizhevsky et al. (2012) [AlexNet moment]. This sudden progress was arguably due to the confluence of four factors: larger training datasets, improved processing power for training, the use of the ReLU activation function, and the use of stochastic gradient descent.
- **Depth efficiency**: Several results show that there are functions that can be realized by deep networks but not

by any shallow network whose capacity is bounded above exponentially. In other words, it would take an exponentially larger number of units in a shallow network to describe these functions accurately. This is known as the depth efficiency of neural networks.

- **Width efficiency**: Lu et al. (2017) investigate whether there are wide shallow networks (i.e., shallow networks with lots of hidden units) that cannot be realized by narrow networks whose depth is not substantially larger. They show that there exist classes of wide, shallow networks that can only be expressed by narrow networks with polynomial depth. This is known as the width efficiency of neural networks. This polynomial lower bound on width is less restrictive than the exponential lower bound on depth, suggesting that depth is more important. Vardi et al. (2022) subsequently showed that the price for making the width small is only a linear increase in the network depth for networks with ReLU activations.

4.2 Funkcje kosztu

Tak jak zauważyliśmy wyżej wyprowadzoną wcześniej teorię można bezpośrednio zaaplikować dla przypadku sieci neuronowych. Wartości parametrów $\boldsymbol{W}$ znajdujemy zatem w sposób analogiczny jak dla modeli klasycznych (tj. nie głębokich) minimalizując odpowiednią funkcję kosztu $L(\mathcal{X}; \boldsymbol{W})$ zależną od zbioru treningowego $\mathcal{X}$ i parametrów sieci $\boldsymbol{W}$, przy czym w czasie treningu $\mathcal{X}$ jest ustalone, a dostosowujemy parametry $\boldsymbol{W}$, natomiast w czasie inferencji wagi $\boldsymbol{W}$ są ustalone (wytrenowane), a do sieci podajemy nowe przykłady. Wymienione wcześniej funkcje kosztu dla zagadnień regresji, klasyfikacji binarnej oraz wieloklasowej możemy zatem bezpośrednio wykorzystać uwzględniając jednak, iż odwzorowanie $\boldsymbol{\phi}(\boldsymbol{x}; \boldsymbol{W})$ definiujące parametry określonego rozkładu prawdopodobieństwa jest teraz odpowiednią siecią neuronową. Nie ma tutaj więc żadnej zmiany jeśli chodzi o tworzenie modeli matematycznych rozważanych zagadnień w postaci odpowiednich parametrycznych modeli statystycznych, to co się zmienia to większa ekspresywność takich modeli dzięki wykorzystaniu sieci neuronowej jako uniwersalnego aproksymatora dla parametrów modelowanych rozkładów.

Wcześniej wyprowadziliśmy funkcje kosztu (jako zanegowaną logarytmiczną funkcję wiarygodności) dla typowych zagadnień regresji i klasyfikacji – odpowiednio błąd średniokwadratowy oraz entropię krzyżową i

wieloklasową entropię krzyżową. Poniżej zamieszczamy tabelę zawierającą możliwe rozkłady prawdopodobieństwa, które można wykorzystać do skonstruowania funkcji kosztu dla mniej typowych zagadnień uczenia maszynowego.

Domain	*Distribution*	*Use*
$y \in \mathbb{R}$	univariate normal	regression
$y \in \mathbb{R}$	Laplace or t-distribution	robust regression
$y \in \mathbb{R}$	mixture of gaussians	multimodal regression
$y \in \mathbb{R}_+$	exponential or gamma	predicting magnitude
$y \in [0; 1]$	beta	predicting proportion
$\boldsymbol{y} \in \mathbb{R}^d$	multivariate normal	multivariate regression
$y \in (-\pi; +\pi]$	von Mises	predicting direction
$y \in \{0, 1\}$	Bernoulli	binary classification
$y \in \{1, 2, \ldots, K\}$	categorical	multiclass classification
$y \in \{0, 1, 2, 3, \ldots\}$	Poisson	predicting event counts
$\boldsymbol{y} \in \text{Perm}(n)$	Plackett–Luce	ranking

Tabela 1: *Figure 5.11, Ch. 5, [1]*

[*Multiple outputs, Ch. 5,* [1]] To make two or more prediction types simultaneously, we similarly assume the errors in each are independent. For example, to predict wind direction and strength, we might choose the von Mises distribution (defined on circular domains) for the direction and the exponential distribution (defined on positive real numbers) for the strength. The independence assumption implies that the joint likelihood of the two predictions is the product of individual likelihoods. These terms will become additive when we compute the negative log-likelihood.

[*Notes, Ch. 5,* [1]]

- **Class imbalance and focal loss**: Lin et al. (2017c) address data imbalance in classification problems. If the number of examples for some classes is much greater than for others, then the standard maximum likelihood loss does not work well; the model may concentrate on becoming more confident about well-classified examples from the dominant classes and classify less well- represented classes

poorly. Lin et al. (2017c) introduce focal loss, which adds a single extra parameter that down-weights the effect of well-classified examples to improve performance.

- **Learning to rank**: Cao et al. (2007), Xia et al. (2008), and Chen et al. (2009) all used the Plackett–Luce model in loss functions for learning to rank data. This is the listwise approach to learning to rank as the model ingests an entire list of objects to be ranked at once. Alternative approaches are the pointwise approach, in which the model ingests a single object, and the pairwise approach, where the model ingests pairs of objects.
- **Non-probabilistic approaches**: It is not strictly necessary to adopt the probabilistic approach discussed in this chapter, but this has become the default in recent years; any loss function that aims to reduce the distance between the model output and the training outputs will suffice, and distance can be defined in any way that seems sensible. There are several well-known non-probabilistic machine learning models for classification, including support vector machines, which use hinge loss and AdaBoost, which uses exponential loss.

4.3 Aspekty optymalizacyjne

Przedstawimy teraz algorytmy optymalizacji numerycznej najczęściej używane przy treningu głębokich sieci neuronowych. Jest oczywiste, iż nie jest możliwe znalezienie wartości wag sieci neuronowej minimalizujących funkcję kosztu $L(\mathcal{X}; \boldsymbol{W})$ w sposób analityczny. Dodatkowo ze względu na ogromną liczbę parametrów współczesnych sieci (rzędu 10^9) jakiekolwiek wyrafinowane algorytmy optymalizacji numerycznej (np. wykorzystujące drugie pochodne) również nie wchodzą w grę ze względu na zbyt duże wymagania pamięciowe. Biorąc pod uwagę wymienione ograniczenia skupimy się na prostych algorytmach optymalizacji pierwszego rzędu (z ang. *first order methods*).

Podstawowym algorytmem jest metoda spadku wzdłuż gradientu (z ang. *gradient descent*). Polega ona na iteracyjnym aktualizowaniu wartości parametrów $\boldsymbol{W}$ zgodnie z kierunkiem najszybszego spadku wyznaczonego przez zanegowany gradient funkcji kosztu $-\frac{\partial L(\mathcal{X}; \boldsymbol{W})}{\partial \boldsymbol{W}}$. Wielkość kroków jakie wykonujemy w każdej iteracji jest zdeterminowana przez stałą $\lambda > 0$ nazywaną *stałą uczącą* (z ang. *learning rate*). Tutaj należy dodać iż w implementacji

wartość funkcji kosztu jest mnożona przez liczbę przykładów, w taki sposób, iż obliczamy tak naprawdę średni koszt na przykład, czyli $L \leftarrow \frac{1}{|\mathcal{X}|} L$. W przeciwnym wypadku wartość stałej uczącej należałoby dostosowywać do liczby przykładów w zbiorze.

```
# Gradient descent
# ----------------
# Let 𝒳 be the whole training set.
# Initialize heuristically parameters 𝑾 of the neural net.
for epoch in range(max_epochs):
    # Step 1. Compute derivatives of the loss function with
    # respect to the parameters of neural net ∂L(𝒳;𝑾)/∂𝑾.

    # Step 2. Update parameters of the neural net according to
    𝑾 ← 𝑾 − λ ∂L(𝒳;𝑾)/∂𝑾
    # where λ > 0 is a positive constant called 'learning rate'.
```

> [*Gradient descent variants*, [2]] There are three variants of gradient descent, which differ in how much data we use to compute the gradient of the objective function. Depending on the amount of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update.

Podstawowe modyfikacje powyższego algorytmu (zwanego również *full-batch gradient descent*) są związane z ograniczeniem zbioru treningowego dla którego obliczamy gradient. Podejście takie wynika z faktu, iż dla bardzo dużych zbiorów treningowych $\mathcal{X}$ obliczane tensory mogą nie mieścić się w pamięci. Dodatkowo modyfikacja wprowadza losowość przy aktualizacji parametrów sieci, co pozwala na ucieczkę z lokalnych minimów funkcji kosztu. Aby rozwiązać ten problem dzielimy zbiór $\mathcal{X}$ w każdej epoce na rozłączne *batche* $\mathcal{B}$ o tej samej wielkości n_b składające się z losowych przykładów ze zbioru $\mathcal{X}$ (losowane bez zwracania i każdorazowo przy rozpoczęciu nowej epoki). Algorytm taki nazywamy *batch gradient descent* lub *mini-batch gradient descent*.

```
# Mini-batch gradient descent
# ---------------------------
# Let 𝒳 be the whole training set.
# Initialize heuristically parameters 𝑾 of the neural net.
for epoch in range(max_epochs):
    random_shuffle(𝒳)
    for ℬ in get_batches(𝒳, batch_size):
        # Step 1. Compute derivatives of the loss function with
        # respect to the parameters of neural net ∂L(ℬ;𝑾)/∂𝑾.

```

```
        # Step 2. Update parameters of the neural net according to
        W ← W − λ ∂L(B;W)/∂W
        # where λ > 0 is a positive constant called 'learning rate'.
```

W szczególności gdy $n_{\mathrm{b}} = 1$, tj. każdy batch składa się z pojedynczego, losowo wybranego przykładu uczącego, to taki algorytm nazywamy *stochastycznym spadkiem wzdłuż gradientu* (z ang. *stochastic gradient descent, SGD*).

> [*Stochastic gradient descent*, [2]] Gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online. SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily.
>
> While gradient descent converges to the minimum of the basin the parameters are placed in, SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minima. On the other hand, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting. However, it has been shown that when we slowly decrease the learning rate, SGD shows the same convergence behavior as batch gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively.

> [*Mini-batch gradient descent*, [2]] Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of training examples. This way, it a) reduces the variance of the parameter updates, which can lead to more stable convergence; b) can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient.
>
> Common mini-batch sizes range between 50 and 256, but can vary for different applications. Mini-batch gradient descent is typically the algorithm of choice when training a neural network and the term SGD usually is employed also when mini-batches are used.

4.3.1 Metoda pędu

W standardowym sformułowaniu SGD ma pewne problemy. Przede wszystkim dobór odpowiedniej wartości stałej uczącej do uzyskania dobrej zbieżności może być trudny. Dodatkowo istnieje poważny problem, który pojawia się przy minimalizacji silnie niewypukłych funkcji kosztu polegający na tym, iż algorytm utyka w lokalny minimum lub w punkcie siodłowym funkcji. Standardowy algorytm SGD nie posiada żadnych mechanizmów pozwalających na ucieczkę z tych punktów. Podstawowym ulepszeniem algorytmu SGD jest wprowadzenie tzw. pędu (z ang. *momentum*) tj. członu postaci

$$\boldsymbol{V}_t = \gamma \boldsymbol{V}_{t-1} - \lambda \frac{\partial L(\mathcal{B}; \boldsymbol{W})}{\partial \boldsymbol{W}}\Big|_{\boldsymbol{W}}, \quad \boldsymbol{V}_0 = \boldsymbol{0}$$

i aktualizację parametrów sieci zgodnie z

$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \boldsymbol{V}_t\,,$$

gdzie t jest numerem iteracji, a γ to tzw. *momentum term*, którego wartość typowo przyjmuje się równą 0.9.

> [*Momentum*, [2]] Essentially, when using momentum, we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity, if there is air resistance, i.e. $\gamma < 1$). The same thing happens to our parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

4.3.2 Metoda Nesterova

> [*Nesterov accelerated gradient*, [2]] However, a ball that rolls down a hill, blindly following the slope, is highly unsatisfactory. We would like to have a smarter ball, a ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.
>
> Nesterov accelerated gradient is a way to give our momentum term this kind of prescience. We know that we will use our momentum term $\gamma \boldsymbol{V}_{t-1}$ to move the parameters $\boldsymbol{W}$. Computing $\boldsymbol{W} + \gamma \boldsymbol{V}_{t-1}$ thus gives us an approximation of the next position

of the parameters (the gradient is missing for the full update), a rough idea where our parameters are going to be. We can now effectively look ahead by calculating the gradient not at our current parameters $\boldsymbol{W}$ but at the approximate future position of our parameters:

$$\begin{aligned}\boldsymbol{V}_t &= \gamma \boldsymbol{V}_{t-1} - \lambda \frac{\partial L(\mathcal{B}; \boldsymbol{W})}{\partial \boldsymbol{W}}\Big|_{\boldsymbol{W}+\gamma \boldsymbol{V}_{t-1}}, \quad \boldsymbol{V}_0 = \boldsymbol{0}\,, \\ \boldsymbol{W} &\leftarrow \boldsymbol{W} + \boldsymbol{V}_t\,.\end{aligned}$$

4.3.3 Adam

[*Adam, Ch.6*, [1]] Gradient descent with a fixed step size has the following undesirable property: it makes large adjustments to parameters associated with large gradients (where perhaps we should be more cautious) and small adjustments to parameters associated with small gradients (where perhaps we should explore further). When the gradient of the loss surface is much steeper in one direction than another, it is difficult to choose a learning rate that (i) makes good progress in both directions and (ii) is stable.

A straightforward approach is to normalize the gradients so that we move a fixed distance (governed by the learning rate) in each direction.

$$\begin{aligned}\boldsymbol{G} &= \frac{\partial L(\mathcal{B}; \boldsymbol{W})}{\partial \boldsymbol{W}}\Big|_{\boldsymbol{W}}\,, \\ \boldsymbol{W} &\leftarrow \boldsymbol{W} - \lambda \frac{\boldsymbol{G}}{\sqrt{\boldsymbol{G}^2} + \epsilon}\,.\end{aligned}$$

where the square root and division are both elementwise, λ is the learning rate, and ϵ is a small constant that prevents division by zero when the gradient magnitude is zero.

The result is that the algorithm moves a fixed distance λ along each coordinate, where the direction is determined by whichever way is downhill. This simple algorithm makes good progress in both directions but will not converge unless it happens to land exactly at the minimum. Instead, it will bounce back and forth around the minimum.

Adaptive moment estimation, or *Adam*, takes this idea and adds momentum to both the estimate of the gradient and the squared

gradient:

$$\boldsymbol{G} = \frac{\partial L(\mathcal{B}; \boldsymbol{W})}{\partial \boldsymbol{W}}\Big|_{\boldsymbol{W}},$$
$$\boldsymbol{M}_t = \beta_1 \boldsymbol{M}_{t-1} + (1-\beta_1)\boldsymbol{G}, \quad \tilde{\boldsymbol{M}}_t = \frac{\boldsymbol{M}_t}{1-\beta_1^t},$$
$$\boldsymbol{V}_t = \beta_2 \boldsymbol{V}_{t-1} + (1-\beta_2)\boldsymbol{G}^2, \quad \tilde{\boldsymbol{V}}_t = \frac{\boldsymbol{V}_t}{1-\beta_2^t},$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \lambda \frac{\tilde{\boldsymbol{M}}_t}{\sqrt{\tilde{\boldsymbol{V}}_t} + \epsilon}.$$

This algorithm can converge to the overall minimum and makes good progress in every direction in the parameter space.

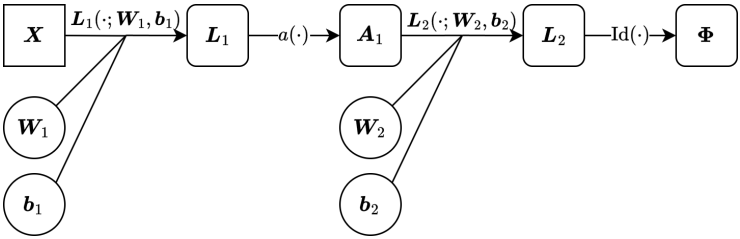
The choices of learning algorithm, batch size, learning rate schedule, and momentum coefficients are all considered hyperparameters of the training algorithm; these directly affect the final model performance but are distinct from the model parameters. Choosing these can be more art than science, and it is common to train many models with different hyperparameters and choose the best one. This is known as *hyperparameter search*.

[*Adam*, [2]] The authors propose default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ϵ. They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms.

4.4 Algorytm wstecznej propagacji błędu

W poprzednim paragrafie opisaliśmy podstawowe algorytmy wykorzystywane do minimalizacji funkcji kosztu, a co za tym idzie trenowania sieci neuronowych. Nie opisaliśmy jednak dwóch dość istotnych szczegółów: 1) jak właściwie obliczać gradient funkcji kosztu po parametrach sieci $\frac{\partial L(\mathcal{B};\boldsymbol{W})}{\partial \boldsymbol{W}}$; 2) w jaki sposób zainicjalizować wartości parametrów sieci. W tym paragrafie zajmujemy się pierwszym z wymienionych problemów, natomiast w następnym poruszamy kwestie regularyzacji oraz inicjalizacji głębokich sieci neuronowych.

Sieć neuronowa (przetwarzająca od razu cały batch przykładów reprezentowany przez tablicę $\boldsymbol{X}$) jest tak naprawdę pewnym skierowanym, acyklicznym grafem (z ang. *Directed Acyclic Graph, DAG*), zwanym *grafem obliczeń*. Przykładowy graf obliczeń reprezentujący prostą sieć MLP z jedną warstwą ukrytą przedstawiono na rysunku 3.



Rysunek 3: Graf obliczeń reprezentujący prostą sieć MLP z jedną warstwą ukrytą

Chcemy w sposób automatyczny z dokładnością do epsilon maszynowego obliczyć wartości pochodnych $\frac{\partial L(\mathcal{X};\boldsymbol{W})}{\partial \boldsymbol{W}_i}$, $\frac{\partial L(\mathcal{X};\boldsymbol{W})}{\partial \boldsymbol{b}_i}$. Zauważmy, iż zgodnie z regułą łańcuchową oznaczając

$$\boldsymbol{\Delta}_i^{\mu_1\mu_2} = \frac{\partial L}{\partial \boldsymbol{L}_i^{\mu_1\mu_2}}$$

mamy

$$\frac{\partial L}{\partial \boldsymbol{W}_i^{\mu_1\mu_2}} = \sum_{\nu_1,\nu_2} \boldsymbol{\Delta}_i^{\nu_1\nu_2} \frac{\partial \boldsymbol{L}_i^{\nu_1\nu_2}}{\partial \boldsymbol{W}_i^{\mu_1\mu_2}} = \sum_{\nu_1} \boldsymbol{\Delta}_i^{\nu_1\mu_1} \boldsymbol{A}_{i-1}^{\nu_1\mu_2}$$

oraz

$$\frac{\partial L}{\partial \boldsymbol{b}_i^{\mu}} = \sum_{\nu} \boldsymbol{\Delta}_i^{\nu\mu} ,$$

gdzie oznaczamy $\boldsymbol{A}_0 = \boldsymbol{X}$. Jednocześnie zauważmy, że

$$\boldsymbol{\Delta}_i^{\mu_1\mu_2} = \sum_{\nu_1,\nu_2} \frac{\partial L}{\partial \boldsymbol{A}_i^{\nu_1\nu_2}} \frac{\partial \boldsymbol{A}_i^{\nu_1\nu_2}}{\partial \boldsymbol{L}_i^{\mu_1\mu_2}}$$

oraz

$$\frac{\partial L}{\partial \boldsymbol{A}_i^{\nu_1\nu_2}} = \sum_{\gamma_1,\gamma_2} \frac{\partial L}{\partial \boldsymbol{L}_{i+1}^{\gamma_1\gamma_2}} \frac{\partial \boldsymbol{L}_{i+1}^{\gamma_1\gamma_2}}{\partial \boldsymbol{A}_i^{\nu_1\nu_2}} = \sum_{\gamma} \boldsymbol{\Delta}_{i+1}^{\nu_1\gamma} \boldsymbol{W}_{i+1}^{\gamma\nu_2}$$

i

$$\frac{\partial \boldsymbol{A}_i^{\nu_1\nu_2}}{\partial \boldsymbol{L}_i^{\mu_1\mu_2}} = a'(\boldsymbol{L}_i^{\nu_1\nu_2}) \delta_{\mu_1\nu_1} \delta_{\mu_2\nu_2} ,$$

skąd

$$\boldsymbol{\Delta}_i^{\mu_1\mu_2} = a'(\boldsymbol{L}_i^{\mu_1\mu_2}) \sum_{\gamma} \boldsymbol{\Delta}_{i+1}^{\mu_1\gamma} \boldsymbol{W}_{i+1}^{\gamma\mu_2} .$$

Powyższe wyniki możemy zapisać w zwartych postaciach macierzowych

$$\boxed{\frac{\partial L}{\partial \boldsymbol{W}_i} = \boldsymbol{\Delta}_i^T \boldsymbol{A}_{i-1}\,, \quad \frac{\partial L}{\partial \boldsymbol{b}_i} = \boldsymbol{\Delta}_i^T \mathbf{1}\,, \quad \boldsymbol{\Delta}_i = a'(\boldsymbol{L}_i) \odot (\boldsymbol{\Delta}_{i+1}\boldsymbol{W}_{i+1})\,,}$$

gdzie $\boldsymbol{A}_0 = \boldsymbol{X}$ oraz $\boldsymbol{\Delta}_{k+1} = \frac{\partial L}{\partial \boldsymbol{\Phi}}$ (zauważmy tutaj, iż człon ten wyznaczyliśmy już powyżej dla typowych funkcji kosztu). Powyższy przykład to wyprowadzenie algorytmu wstecznej propagacji błędu dla typowej sieci w pełni połączonej (reprezentowanej przez graf obliczeń 3 z powielonymi warstwami ukrytymi – na rysunku przedstawiono sieć z jedną warstwą ukrytą). Algorytm obliczania pochodnych możemy zatem zapisać następująco.

```
# Backpropagation through MLP
# ---------------------------
# Let k be the number of hidden layers of the MLP network.

# Step 1. For given batch X perform a forward pass
# by computing and memorizing arrays L_i, A_i.

A_0 = X

for i in range(1, k+2):
    # Compute and memorize
    L_i = A_{i-1} W_i^T + b_i
    A_i = a(L_i)

# Compute loss L(B;W) for the output Φ = L_{k+1}.

# Step 2. Perform backpropagation

# Initialize seed
Δ_{k+1} = ∂L(B;W)/∂Φ

for i in range(k+1, 0, -1):
    # Compute gradients
    ∂L/∂W_i = Δ_i^T A_{i-1},   ∂L/∂b_i = Δ_i^T 1

    # Update seed
    if i > 1:
        Δ_{i-1} = a'(L_{i-1}) ⊙ (Δ_i W_i)
```

Powyższy algorytm jest specjalnie dostosowany do sieci w pełni połączonej. Możemy go jednak prosto uogólnić do dowolnego acyklicznego i skierowanego grafu obliczeń. Załóżmy, iż rozpatrujemy warstwę (wierzchołek w grafie zaznaczony zaokrąglonym kwadratem) $\boldsymbol{F}$ zależną od pewnych parametrów $\boldsymbol{W}$ (w szczególności warstwa może nie mieć żadnych parametrów). Dla dowolnej warstwy $\boldsymbol{F}$ przez $\dot{\boldsymbol{F}}$ oznaczymy pochodną $\dot{\boldsymbol{F}} = \frac{\partial L}{\partial \boldsymbol{F}}$.

Wówczas z reguły łańcuchowej

$$\frac{\partial L}{\partial \boldsymbol{W}^{\mu_1..\mu_k}} = \sum_{\nu_1..\nu_l} \dot{\boldsymbol{F}}^{\nu_1..\nu_l} \frac{\partial \boldsymbol{F}^{\nu_1..\nu_l}}{\partial \boldsymbol{W}^{\mu_1..\mu_k}} .$$

Niech $\mathrm{succ}(\boldsymbol{F})$ oznacza zbiór następników wierzchołka $\boldsymbol{F}$, tj. warstw (wierzchołków w grafie zaznaczonych zaokrąglonym kwadratem) do których istnieje skierowana krawędź od warstwy $\boldsymbol{F}$. Wówczas

$$\dot{\boldsymbol{F}}^{\mu_1..\mu_k} = \frac{\partial L}{\partial \boldsymbol{F}^{\mu_1..\mu_k}} = \sum_{\boldsymbol{H} \in \mathrm{succ}(\boldsymbol{F})} \sum_{\nu_1..\nu_l} \dot{\boldsymbol{H}}^{\nu_1..\nu_l} \frac{\partial \boldsymbol{H}^{\nu_1..\nu_l}}{\partial \boldsymbol{F}^{\mu_1..\mu_k}} .$$

Z powyższego zatem widzimy, iż możemy obliczać wielkości $\dot{\boldsymbol{F}}$ w sposób iteracyjny przechodząc po warstwach w kolejności odwrotnej do ich porządku topologicznego (z ang. *reversed topological order*) i zapamiętywać wartości $\dot{\boldsymbol{F}}$. Wówczas dla każdej warstwy pochodne $\dot{\boldsymbol{H}}$ jej następników będą już obliczone i będziemy mogli obliczyć $\dot{\boldsymbol{F}}$. Naiwna implementacja powyższego schematu będzie wymagać bardzo dużo pamięci, gdyż dla każdej warstwy musimy przechowywać oprócz jej wartości $\boldsymbol{F}$ również wartość pochodnej $\dot{\boldsymbol{F}}$. Zauważmy, iż w przypadku algorytmu wstecznej propagacji błędu dla sieci MLP nie musimy przechowywać pochodnych funkcji kosztu po wartościach warstw, a jedynie dynamicznie obliczamy i przekazujemy wstecz tzw. *ziarno* (z ang. *seed*) $\boldsymbol{\Delta}$. Wynika to oczywiście z faktu, iż w przypadku sieci MLP warstwy tworzą skierowaną ścieżkę, czyli najprostszy DAG. W przypadku innych architektur sieci neuronowych tak nie musi być i wówczas każdą grupę warstw w obrębie której warstwy nie tworzą ścieżki należy oddzielnie przeanalizować pod kątem wykorzystania pamięci.

[*Notes, Ch.7*, [1]]

- **Reducing memory requirements**: Training neural networks is memory intensive. We must store both the model parameters and the pre-activations at the hidden units for every member of the batch during the forward pass. Two methods that decrease memory requirements are gradient checkpointing (Chen et al., 2016a) and micro-batching (Huang et al., 2019). In gradient checkpointing, the activations are only stored every N layers during the forward pass. During the backward pass, the intermediate missing activations are recalculated from the nearest check- point. In this manner, we can drastically reduce the memory requirements at the computational cost of performing the

forward pass twice. In micro-batching, the batch is subdivided into smaller parts, and the gradient updates are aggregated from each sub-batch before being applied to the network.

4.5 Inicjalizacja

Jak zaznaczyliśmy wyżej ten paragraf poświęcimy zagadnieniu inicjalizacji parametrów sieci neuronowej, który jest wymaganym elementem opisanych algorytmów minimalizacji numerycznej.

[*Parameter initialization, Ch.7*, [1]] The backpropagation algorithm computes the derivatives that are used by stochastic gradient descent and Adam to train the model. We now address how to initialize the parameters before we start training. To see why this is crucial, consider that during the forward pass, each set of pre-activations $\boldsymbol{L}_i$ is computed as:

$$\boldsymbol{L}_i = \boldsymbol{b}_i + a(\boldsymbol{L}_{i-1})\boldsymbol{W}_i^T$$

Imagine that we initialize all the biases to zero and the elements of $\boldsymbol{W}$ according to a normal distribution with mean zero and variance σ^2. Consider two scenarios:

- If the variance is very small (e.g., 10^{-5}), then each element of $\boldsymbol{L}_i$ will be a weighted sum of $\boldsymbol{A}_{i-1}$ where the weights are very small; the result will likely have a smaller magnitude than the input. In addition, the ReLU function clips values less than zero, so the range of $\boldsymbol{A}_i$ will be half that of $\boldsymbol{L}_i$. Consequently, the magnitudes of the pre-activations at the hidden layers will get smaller and smaller as we progress through the network.
- If the variance is very large (e.g., 10^5), then each element of $\boldsymbol{L}_i$ will be a weighted sum of $\boldsymbol{A}_{i-1}$ where the weights are very large; the result is likely to have a much larger magnitude than the input. The ReLU function halves the range of the inputs, but if variance is large enough, the magnitudes of the pre-activations will still get larger as we progress through the network.

In these two situations, the values at the pre-activations can become so small or so large that they cannot be represented with finite precision floating point arithmetic.

> Even if the forward pass is tractable, the same logic applies to the backward pass. Each gradient update consists of multiplying by $\boldsymbol{W}_i$. If the values of $\boldsymbol{W}_i$ are not initialized sensibly, then the gradient magnitudes may decrease or increase un- controllably during the backward pass. These cases are known as the *vanishing gradient problem* and the *exploding gradient problem*, respectively. In the former case, updates to the model become vanishingly small. In the latter case, they become unstable.

Najprostszym pomysłem inicjalizacji parametrów sieci neuronowych jest przyjęcie wartości 0 dla obciążeń $\boldsymbol{b}_i$ i wylosowanie elementów macierzy wag $\boldsymbol{W}_i$ niezależnie z rozkładu normalnego $\mathcal{N}(0, \sigma^2)$. Powstaje pytanie jaka powinna być wartość wariancji σ^2. Pewną odpowiedź na to pytanie sugeruje analiza wariancji elementów aktywacji w kolejnych warstwach

$$\boldsymbol{L}_{i+1}^{\beta\mu} = \boldsymbol{b}_i^{\mu} + \sum_{\nu=1}^{n_{i-1}} \boldsymbol{A}_i^{\beta\nu} \boldsymbol{W}_i^{\mu\nu} \,.$$

W przypadku inicjalizacji mamy $\boldsymbol{b}_i^{\mu} = 0$ oraz $\boldsymbol{W}_i^{\mu\nu} \sim \mathcal{N}(0, \sigma_{W_i}^2)$, zatem

$$\mathbb{E}\left[\boldsymbol{L}_{i+1}^{\beta\mu}\right] = \sum_{\nu=1}^{n_{i-1}} \mathbb{E}\left[\boldsymbol{A}_i^{\beta\nu}\right] \cdot \mathbb{E}\left[\boldsymbol{W}_i^{\mu\nu}\right] = 0$$

oraz

$$\mathbb{V}\left[\boldsymbol{L}_{i+1}^{\beta\mu}\right] = \mathbb{E}\left[(\boldsymbol{L}_{i+1}^{\beta\mu})^2\right] - \mathbb{E}\left[\boldsymbol{L}_{i+1}^{\beta\mu}\right]^2 = \sum_{\nu=1}^{n_{i-1}} \mathbb{V}\left[\boldsymbol{A}_i^{\beta\nu}\right] \cdot \sigma_{W_i}^2 \,.$$

Zakładając, iż wariancje składowych aktywacji są równe $\sigma_{A_i}^2$ oraz pochodzą z rozkładu symetrycznego wokół 0 mamy (dla funkcji aktywacji ReLU)

$$\sigma_{A_{i+1}}^2 = \sum_{\nu=1}^{n_{i-1}} \frac{1}{2} \sigma_{W_i}^2 \sigma_{A_i}^2 = \frac{n_{i-1}}{2} \sigma_{W_i}^2 \sigma_{A_i}^2 \,.$$

Widzimy zatem, iż jeśli chcemy aby rozrzut wartości kolejnych aktywacji był podobny do poprzednich musi zachodzić

$$\sigma_{W_i}^2 = \frac{2}{n_{i-1}} \,.$$

Otrzymujemy zatem następującą heurystykę inicjalizacji

$$\boxed{\boldsymbol{b}_i = \boldsymbol{0}\,, \quad \boldsymbol{W}_i^{\mu\nu} \sim \mathcal{N}\left(0, \frac{2}{n_{i-1}}\right)}$$

zwaną *inicjalizacją He*. Analogiczna analiza dla propagacji wstecznej, gdzie aktualizujemy $\boldsymbol{\Delta}$ sugeruje z kolei przyjęcie

$$\sigma^2_{W_i} = \frac{2}{n_i}\,.$$

Jeśli $n_i \neq n_{i-1}$, tj. macierz $\boldsymbol{W}_i$ nie jest kwadratowa, to jedną z możliwości jest przyjęcie średniej arytmetycznej $\overline{n} = \frac{n_{i-1}+n_i}{2}$

$$\sigma^2_{W_i} = \frac{2}{\overline{n}} = \frac{4}{n_{i-1}+n_i}\,.$$

4.6 Regularyzacja

Jak wspomnieliśmy we wstępie ważnym pojęciem w uczeniu maszynowym jest pojemność modeli. Model o dużej pojemności (czyli na przykład właśnie głęboka sieć neuronowa) może nastręczać poważnych problemów w trakcie uczenia przez zjawisko overfittingu. Model o dużej pojemności może perfekcyjne dopasować się do zbioru trenującego. Równocześnie może on zignorować strukturę, której wykrycie było w istocie celem uczenia. Przyczyną przeuczenia jest fakt, że każdy zbiór trenujący jest skończony: 1) jest skończoną próbą obserwacji z zagadnienia, które jest celem uczenia; 2) każdy zbiór trenujący opisuje więc problem uczenia w sposób niepełny; 3) model przeucza, gdy jego pojemność pozwala mu dopasować się do konkretnego zbioru trenującego w sposób uniemożliwiający uogólnianie na nowe przypadki. Problem ten jest tym istotniejszy, im mniejszy jest zbiór trenujący. Algorytmy i techniki służące do kontroli pojemności modelu nazywamy metodami regularyzacji (z ang. *regularization methods*).

- **Weight decay**. Analogicznie jak w przypadku prostych modeli liniowych jedną z możliwości regularyzacji jest dodanie do funkcji kosztu L czynnika regularyzującego zawierającego odpowiednią sumę norm poszczególnych parametrów sieci

$$L^*(\mathcal{X};\boldsymbol{W}) = L(\mathcal{X};\boldsymbol{W}) + \frac{\gamma_2}{2}\sum_{i=1}^{k+1}\sum_{\mu,\nu}(\boldsymbol{W}_i^{\mu\nu})^2 + \gamma_1\sum_{i=1}^{k+1}\sum_{\mu,\nu}|\boldsymbol{W}_i^{\mu\nu}|\,.$$

  Różnica między regularyzacją L^2 i L^1 jest taka, że norma L^2 „zachęca" sieć do wybierania małych wag, ale nie dokładnie będących 0, natomiast L^1 zmniejsza wagi, ale w szczególności prowadzi do wag wynoszących dokładnie 0. W związku z tym regularyzacja L^1 przeprowadza selekcję cech - "wyłączy" te, które są mało ważne. Zwykle

człony regularyzujące wprowadza się jedynie dla wag sieci, a nie dla obciążeń, skąd nazwa tej metody regularyzacji – *weight decay*.

- **Metoda wczesnego stopu**. Najprostszym, a zarazem powszechnie stosowanym sposobem regularyzacji sieci neuronowych jest metoda wczesnego stopu (z ang. *early stopping*). Polega ona na podziale zbioru *treningowego* na mniejszy zbiór treningowy i tzw. *zbiór walidacyjny*. W trakcie treningu monitorujemy błąd popełniany przez model na zbiorze walidacyjnym (ale do treningu używamy tylko zbioru treningowego). Przerywamy uczenie, gdy błąd na zbiorze walidacyjnym zaczyna rosnąć lub nie maleje przez wybraną liczbę epok. Dlaczego postęp uczenia monitorowany jest na zbiorze walidacyjnym (wydzielonym ze zbioru trenującego) a nie na zbiorze testowym? Monitorując przebieg uczenia na zbiorze testowym dobieralibyśmy parametry uczenia (np. ilość epok) tak, by dawały dobry wynik na zbiorze testowym. Wówczas wynik na zbiorze testowym mógłby być wyraźnie lepszy niż faktyczna skuteczność sieci w docelowym zastosowaniu. Nie możemy dobierać parametrów modelu wprost pod zbiór, na którym będziemy oceniać jego docelową skuteczność. Celem zbioru testowego jest ocena skuteczności modelu dla nowych obserwacji (niedostępnych w czasie uczenia).

  Takie podejście wymaga dość dużego zbioru danych, z których musimy wydzielić trzy zbiory: zbiór treningowy (na którym uczymy model), zbiór walidacyjny (na którym ewaluujemy model przy procedurze early stopingu) oraz zbiór testowy (na którym ewaluujemy model po treningu). Okres, przez który czekamy na uzyskanie lepszego wyniku, to cierpliwość (z ang. *patience*). Im mniejsze, tym mocniejszy jest ten rodzaj regularyzacji, ale trzeba z tym uważać, bo łatwo jest przesadzić i zbyt szybko przerywać trening. Niektóre implementacje uwzględniają tzw. *grace period*, czyli gwarantowaną minimalną liczbę epok, przez którą będziemy trenować sieć, niezależnie od wybranej cierpliwości.

- **Dropout**. Dobrym sposobem regularyzacji modeli uczenia maszynowego jest uśrednianie odpowiedzi wielu modeli popełniających różne błędy. Chcemy więc uczyć wiele modeli (rozwiązujących ten sam problem) w taki sposób, by modele te różniły się między sobą. Jest to tzw. uczenie zespołowe (z ang. *ensemble learning*). Jak zapewnić, by wyuczone modele się między sobą różniły? Jednym ze sposobów budowy modelu zespołowego jest *bagging* (od bootstrap aggregating) polegający na trenowaniu modeli na losowo wybranych (ze zwraca-

niem) podzbiorach zbioru treningowego, odpowiedzią modelu będzie średnia z odpowiedzi wszystkich wyuczonych w taki sposób modeli. Oczywiście w przypadku obliczeniowo kosztownych modeli jak sieci neuronowe zaimplementowanie takiego schematu wprost jest nieefektywne. Metodą, która efektywnie realizuje właśnie uczenie zespołowe na pojedynczej sieci neuronowej jest dropout. Polega on na tym, iż w procesie treningu ze z góry zadanym prawdopodobieństwem „wyłączamy" wyjście neuronu (tj. dla każdej aktualizacji wartości parametrów sieci, w algorytmie wstecznej propagacji błędu mnożymy elementwise wyjście $\boldsymbol{A}_i$ przez tensor $\boldsymbol{D}_i$, którego każdy element może być równy 0 z prawdopodobieństwem p_i lub 1 z prawdopodobieństwem $1 - p_i$). Mechanizmu dropout używamy tylko w procesie treningu. W przypadku inferencji wyjście warstwy $\boldsymbol{A}_i$ jest z kolei mnożone przez $1 - p_i$, czyli prawdopodobieństwo pozostania neuronu w sieci (*weight scaling inference rule*). Typowo usuwa się dość sporo neuronów w warstwach ukrytych, np. 50%. Dzięki dropoutowi można używać dość dużych stałych uczących oraz współczynników pędu, a także trenować znacznie większe i głębsze sieci. Okazuje się, iż w praktyce takie duże, mocno trenowane sieci z regularyzacją uczą się lepiej niż mniejsze, które by tej regularyzacji nie potrzebowały. Jeśli sieć nie przeucza na zbiorze trenującym to najpewniej lepsze wyniki uzyskamy ucząc większą sieć (która normalnie mogłaby przeuczyć) i stosując dropout. Obecnie dropout jest powszechnie stosowanym sposobem regularyzacji warstw w pełni połączonych (np. ostatnich warstw w niektórych sieciach konwolucyjnych).

- **Transfer learning**.

  > [*Transfer learning and multi-task learning, Ch.9*, [1]] When training data are limited, other datasets can be exploited to improve performance. In transfer learning, the network is pre-trained to perform a related secondary task for which data are more plentiful. The resulting model is then adapted to the original task. This is typically done by removing the last layer and adding one or more layers that produce a suitable output. The main model may be fixed, and the new layers trained for the original task, or we may fine-tune the entire model. The principle is that the network will build a good internal representation of the data from the secondary task, which can subsequently be exploited for the original task. Equivalently, transfer learning can be

viewed as initializing most of the parameters of the final network in a sensible part of the space that is likely to produce a good solution.

- **Augmentation**. Metoda data augmentation polega na dodaniu do zbioru treningowego replik przykładów z tego zbioru poddanych pewnym transformacjom względem których oczekujemy, iż odpowiedzi sieci będą niezmienione.

  [*Notes, Ch. 9*, [1]]

  - **Finding wider minima**: It is thought that wider minima generalize better. Here, the exact values of the weights are less important, so performance should be robust to errors in their estimates. One of the reasons that applying noise to parts of the network during training is effective is that it encourages the network to be indifferent to their exact values. Keskar et al. (2017) showed that SGD finds wider minima as the batch size is reduced. This may be because of the batch variance term that results from implicit regularization by SGD. Ishida et al. (2020) use a technique named flooding, in which they intentionally prevent the training loss from becoming zero. This encourages the solution to perform a random walk over the loss landscape and drift into a flatter area with better generalization.

4.7 Hiperparametry

4.7.1 Dobór hiperparametrów

[*Choosing hyperparameters, Ch.8*, [1]] For a deep network, the model capacity depends on the numbers of hidden layers and hidden units per layer as well as other aspects of architecture that we have yet to introduce. Furthermore, the choice of learning algorithm and any associated parameters (learning rate, etc.) also affects the test performance. These elements are collectively termed hyperparameters. The process of finding the best hyperparameters is termed hyperparameter search or (when focused on network structure) neural architecture search.

Hyperparameters are typically chosen empirically; we train many models with different hyperparameters on the same training set, measure their performance, and retain the best model. However,

we do not measure their performance on the test set; this would admit the possibility that these hyperparameters just happen to work well for the test set but don't generalize to further data. Instead, we introduce a third dataset known as a validation set. For every choice of hyperparameters, we train the associated model using the training set and evaluate performance on the validation set. Finally, we select the model that worked best on the validation set and measure its performance on the test set. In principle, this should give a reasonable estimate of the true performance.

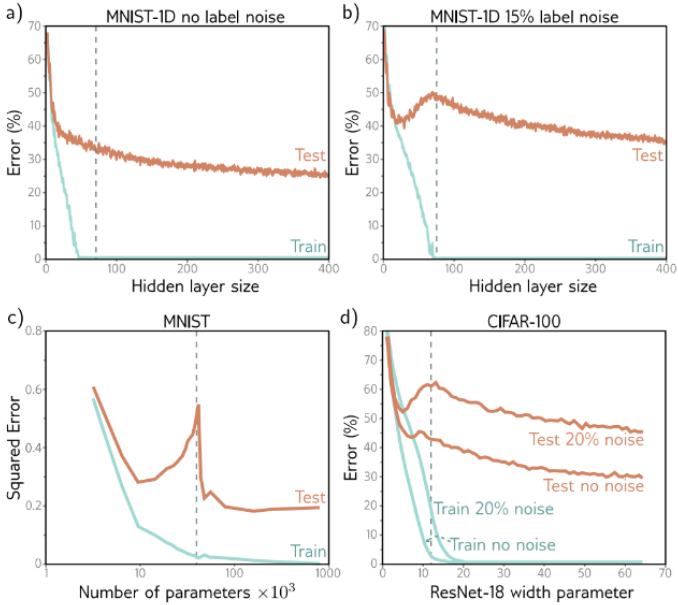
The hyperparameter space is generally smaller than the parameter space but still too large to try every combination exhaustively. Unfortunately, many hyperparameters are discrete (e.g., the number of hidden layers), and others may be conditional on one another (e.g., we only need to specify the number of hidden units in the tenth hidden layer if there are ten or more layers). Hence, we cannot rely on gradient descent methods as we did for learning the model parameters. Hyperparameter optimization algorithms intelligently sample the space of hyperparameters, contingent on previous results. This procedure is computationally expensive since we must train an entire model and measure the validation performance for each combination of hyperparameters.

4.7.2 Zjawisko double descent

Zjawisko double descent polega na nieoczekiwanym zachowaniu wartości błędów popełnianych przez model uczenia maszynowego przy zwiększaniu liczby jego parametrów. Zgodnie z bias-variance tradeoff oczekiwalibyśmy bowiem iż w miarę zwiększania liczby parametrów będziemy coraz bardziej przeuczać model (dopasowywać go wprost do danych treningowych), a przez to zmniejszać jego zdolność generalizacji, co przekładałoby się na gorszy wynik na zbiorze testowym. Okazuje się jednak, iż krzywa błędu na zbiorze testowym w zależności od liczby parametrów początkowo spada, następnie (zgodnie z oczekiwaniami) gdy liczba parametrów jest rzędu liczności zbioru treningowego błąd rośnie, ale (co niespodziewane) gdy liczba parametrów jest znacznie większa od liczności zbioru treningowego, błąd testowy znowu spada.

[*Double descent, Ch.8*, [1]] This phenomenon is known as double descent. For some datasets like MNIST, it is present with the

original data. For others, like MNIST-1D and CIFAR-100, it emerges or becomes more prominent when we add noise to the labels. The first part of the curve is referred to as the *classical* or *under-parameterized regime*, and the second part as the *modern* or *over-parameterized regime*. The central part where the error increases is termed the critical regime.



Rysunek 4: Przykładowa krzywa wartości błędu na zbiorze testowym w zależności od liczby parametrów modelu wykazująca efekt double descent

[*Double descent, Ch. 8*, [1]] To understand why performance continues to improve as we add more parameters, note that once the model has enough capacity to drive the training loss to near zero, the model fits the training data almost perfectly. This implies that further capacity cannot help the model fit the training data any better; any change must occur between the training points. The tendency of a model to prioritize one solution over another as it extrapolates between data points is known as its inductive bias.

The putative explanation for double descent is that as we add capacity to the model, it interpolates between the nearest data points increasingly smoothly. In the absence of information about what happens between the training points, assuming smoothness is sensible and will probably generalize reasonably to new data.

However, this does not explain why over-parameterized models should produce smooth functions. The answer to this question is uncertain, but there are two likely possibilities. First, the network initialization may encourage smoothness, and the model never departs from the sub-domain of smooth function during the training process. Second, the training algorithm may somehow "prefer" to converge to smooth functions. Any factor that biases a solution toward a subset of equivalent solutions is known as a regularizer, so one possibility is that the training algorithm acts as an implicit regularizer.

4.8 Sieci konwolucyjne

4.9 Sieci rezydualne

4.10 Sieci transformer

Literatura

[1] Simon J.D. Prince. *Understanding Deep Learning*. The MIT Press, 2023. Available at http://udlbook.com.

[2] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017. arXiv:1609.04747, available at https://arxiv.org/abs/1609.04747.