

THE SCHOLAR SQUAD

Uczenie Maszynowe

Teoria

The Scholar Squad

June 29, 2023

Abstract

Przygotujcie się na niezapomnianą przygodę z kursem z uczenia maszynowego. Kurs ten zabierze Was w podróż po klasycznych algorytmach uczenia maszynowego, które potrafią zrobić niesamowite rzeczy, takie jak regresja, klasyfikacja, klastrowanie, redukcja wymiarów i wyszukiwanie wzorców. Poznacie również tajemnice sieci neuronowych i ich różnorodne typy. Kurs ten uświadomi Wam również podstawowe problemy, które mogą Wam się nawarzyć podczas uczenia maszynowego i oświeci ich matematyczne źródła. Będziecie w stanie wybrać odpowiedni algorytm uczenia maszynowego w zależności od konkretnego zadania i dostępnych danych. Ponadto, kurs ten obejmuje tematy związane z walidacją modelu i rozwiązywaniem typowych kłopotów, które mogą Wam się przyplać podczas trenowania modelu. Zdobędziecie umiejętność przeprowadzania walidacji modelu oraz rozwiązywania występujących bolączek.

Contents

1 Rodzaje uczenia maszynowego	6
2 Metryki	7
2.1 <i>Klasyfikacja</i>	8
2.1.1 Accuracy	8
2.1.2 Precision	8
2.1.3 Recall (True Positive Rate, Sensitivity, Probability of Detection)	9
2.1.4 F1-score	9
2.1.5 Kompromis Precision/Recall	9
2.2 <i>Regresja</i>	9
2.2.1 Mean square error (błąd średnio kwadratowy):	9
2.2.2 Mean absolute error (błąd średnio bezwzględny):	9
2.2.3 Entropia:	10
2.2.4 Gini:	10
2.2.5 Entropia krzyżowa:	10
3 Regresja	10
3.1 Regresja	11
3.1.1 Regresja Liniowa	11
3.1.2 Regresja wielomianowa	11
3.1.3 Regresja Logistyczna	11
3.2 Gradient Descent	11
3.2.1 SGD Stochastic Gradient Descent:	11
3.3 Learning Curves	12
3.3.1 Bias	12
3.3.2 Variance	12
3.3.3 Irreducible Error	12
3.3.4 Kompromis między <i>Bias</i> a <i>Variance</i>	12
3.4 Regularyzowane modele liniowe	12
3.4.1 Ridge Regression	12
3.4.2 Lasso Regression	12
3.4.3 Early Stopping	12
4 SVM (Support Vector Machines)	12
4.1 Hard Margin Classification	13
4.2 Soft Margin Classification	13
4.3 Nieliniowa klasyfikacja SVM	13
4.3.1 Polynomial Kernel	13
4.4 Regresor SVM	13
4.5 Drzewa Decyzyjne	13

4.5.1	Hiperparametry	14
4.5.2	Regresja	14
5	Ensemble Learning i Random Forests	14
5.1	Klasyfikacja	15
5.1.1	<i>Hard Voting Classifier</i>	15
5.1.2	<i>Soft Voting Classifier</i>	15
5.2	Bagging i Pasting	15
5.3	Random Forests	15
5.3.1	Extremely Randomized Trees Ensemble	16
5.4	Boosting	16
5.4.1	AdaBoost	16
5.4.2	Gradient Boosting	16
5.5	Stacking	16
6	Redukcja Wymiarów	16
6.1	Curse of Dimensionality	17
6.2	PCA - Principal Component Analysis	17
6.2.1	SVD - Singular Value Decomposition	18
6.2.2	Incremental PCA	18
6.3	Rozmaitości	18
6.3.1	LLE - Locally Linear Embedding	18
7	Uczenie nienadzorowane	18
7.1	Soft Clustering	20
7.2	Hard Clustering	20
7.3	DBSCAN	20
7.4	KNN - K-nearest neighbors	21
7.5	Algorytm centroidów (k-średnich) <i>K-Means</i>	22
7.5.1	Wyznaczanie liczby klastrów	23
8	Sieci neuronowe - wprowadzenie	23
8.1	Perceptron	24
8.1.1	Uczenie perceptronu	24
8.2	Funkcje aktywacji	25
8.2.1	Dlaczego potrzebujemy funkcji aktywacji?	25
8.2.2	Funkcje aktywacji - przegląd	25
8.3	Warstwy	28
8.3.1	Warstwa gęsta	28
9	Głębokie sieci neuronowe	28
9.1	Budowa modelu	29
9.1.1	Keras Sequential API	29

9.1.2	Keras Functional API	29
9.2	Kompilacja i uczenie modelu	30
9.3	Callbacks	31
9.4	Analiza procesu uczenia	32
9.5	Przeszukiwanie przestrzeni hiperparametrów	33
9.5.1	SciKit-Learn	33
9.5.2	Keras Tuner	33
10	Konwolucyjne sieci neuronowe	33
10.1	Konwolucja	34
10.2	Typowe błędy podczas projektowania CNN	34
10.3	Pooling	35
10.4	Dropout	35
10.5	Uczenie rezydualne (Residual Learning)	35
10.6	Klasyfikacja i Lokalizacja obiektów	36
10.6.1	Bounding Boxes	36
10.6.2	Fully Convolutional Networks	36
10.6.3	YOLO You Only Look Once	36
10.6.4	<i>Transponowana warstwa konwolucyjna (Transposed Convolutional Layer)</i>	37
10.6.5	Segmentacja semantyczna	37
10.6.6	Metryki:	37
11	Rekurencyjne sieci neuronowe	37
11.1	Rodzaje RNN ze względu na rodzaj danych wejściowych/wyjściowych	38
11.1.1	Sequence to sequence network	38
11.1.2	Vector to sequence network (Dekoder)	38
11.1.3	Sequence to vector network (Enkoder)	39
11.2	Działanie RNN w kilku krokach:	39
11.3	Przewidywanie kilku kroków czasowych do przodu	39
11.4	Unrolling (rozwijanie)	39
11.5	Osadzenia	39
11.6	Rozwiązywanie problemu niestabilnych gradientów	39
11.7	Wady RNN	40
11.8	Przygotowanie danych do RNN	40
11.8.1	Dzielenie sekwencyjnego zestawu danych na wiele okien	40
11.8.2	Sezonowość	40
12	Sieci Enkoder-Dekoder	40
12.1	Rola enkodera	41
12.2	Rola dekodera	42
12.3	Dlaczego ją stosujemy do tłumaczenia języków?	42
12.4	Sieci Enkoder-Dekoder z różnymi rodzajami sieci	42

12.4.1 CNN jako Enkoder, RNN/LSTM jako Dekoder	42
12.4.2 RNN/LSTM jako Enkoder, RNN/LSTM jako Dekoder	42
13 Attention Mechanisms (Mechanizm Uwagi)	42
13.1 Zasada działania	43
13.1.1 Miara podobieństwa	44
13.2 Logika stojąca za mechanizmem uwagi	44
13.3 Self-Attention	44
13.4 Soft Attention i Hard Attention	45
14 Autoenkoder	45
14.1 Stacked (Deep) Autoencoders	46
14.2 Konwolucyjny Autoenkoder	46
14.3 Rekurencyjny Autoenkoder	46
15 GAN (Generative Adversarial Network)	47
15.1 Generator	48
15.1.1 Implementacja	48
15.2 Dyskryminator	49
15.3 Trenowanie GAN	49
15.3.1 Implementacja	49
15.4 Rodzaje GAN	51
15.4.1 Conditional GAN	51
15.4.2 StyleGAN	51
15.5 Zalecenia w tworzeniu modeli GAN	51
15.6 Techniki regularyzacyjne	51
16 Reinforcement Learning (Uczenie przez wzmacnianie)	51
16.1 Terminologia	52
16.2 Credit Assignment Problem	52
16.2.1 Maksymalizacja nagrody	54
16.3 Policy Search (Wyszukiwanie Polityki)	55
16.3.1 Neural Network Policies czyli użycie sieci neuronowej do wyszukiwania polityki	55
16.3.2 Kompromis między Eksploracją a Eksplotacją	56
16.3.3 Explore Policy Space (Eksploracja przestrzeni polityki)	56
16.4 Catastrophic Forgetting	56
16.5 Multi-Agent Reinforcement Learning	57
16.5.1 Rodzaje Multi-Agent systemów:	57
16.6 Markov Decision Processes	58
16.7 Q-Learning	58
16.7.1 Q-Value (Quality Value)	59
16.7.2 Q-Table	59

16.7.3 Uczenie Monte Carlo	59
16.7.4 Temporal Difference (TD) Learning	60
17 Porównania	60
17.1 Modele	61

1 Rodzaje uczenia maszynowego

Podziały ze względu na:

- nadzór:
 - **uczenie nadzorowane** - trzeba etykietować zbiór uczący
 - **uczenie nienadzorowane** - nie trzeba etykietować zbioru uczącego (uczenie bez nauczyciela)
 - **częściowo nadzorowane** - część danych jest etykietowana, część nie
 - **uczenie przez wzmacnianie** - polega na dawaniu kar i nagród za konkretne wybory, które algorytm uwzględnia przy kolejnych próbach
- sposób uogólnienia:
 - **instancje (nieparametryczny)** (np. KNN, drzewa) - uczenie na pamięć, szuka najbliższego podobnego do pytanego obiektu i podpisuje go tak samo według poznanych zasad, przykład: K-nearest neighbors; przede wszystkim wielkość modelu jest nieznana przed rozpoczęciem uczenia
 - **model (parametryczny)** (np. sieci neuronowe, regresja logistyczna, SVM) - szuka zależności między wartościami i na ich podstawie dobiera parametry funkcji, na podstawie której potem przewiduje wartość; wielkość modelu jest znana przed rozpoczęciem uczenia
- dostęp do danych:
 - **batch** - posiadamy dostęp do kompletnego zbioru danych, jeśli możemy trzymać cały zbiór danych i dane nie zmieniają się zbyt często
 - **online (mini-batches)** - karmimy model ciągle małymi porcjami danych, jeśli nie mamy miejsca na utrzymanie całego zbioru lub dane ciągle się zmieniają
- prostota (interpretowalność)
 - **White Box** - Prosty do zinterpretowania- wiemy dlaczego podjął taką a nie inną decyzję.
 - **Black Box** - Trudny do zinterpretowania- nie wiemy dlaczego podjął taką a nie inną decyzję.

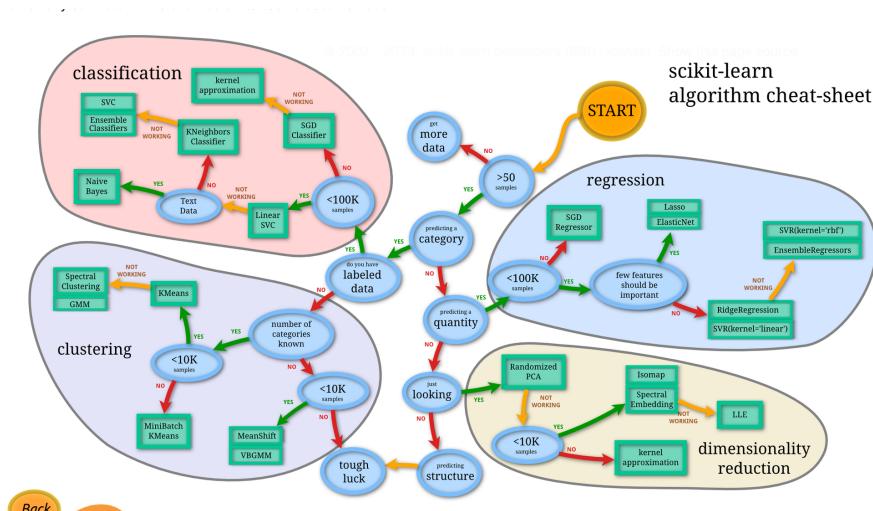


Figure 1: mapka wyboru algorytmu

2 Metryki

2.1 Klasyfikacja

- Confusion Matrix

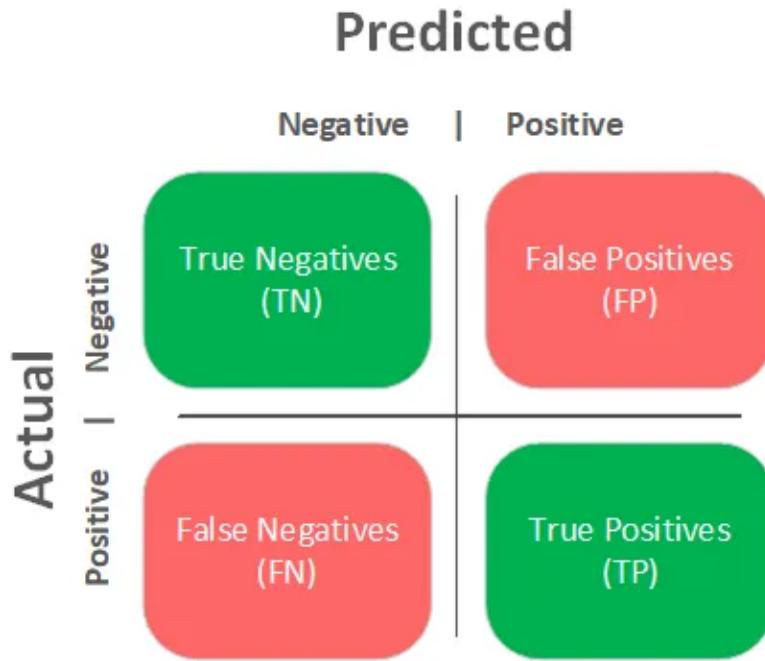


Figure 2: Confusion Matrix

2.1.1 Accuracy

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

- Classification Error
 - Wyraża jaka część instancji została dobrze sklasyfikowana

2.1.2 Precision

$$\text{Precision} = \frac{TP}{TP + FP}$$

- Stosowana gdy wymagamy od modelu wysoką wartość *True Positives* i chcemy zminimalizować liczbę *False Positives*
- Proporcja *True Positives* do sumy *True Positives* i *False Positives*
- W przypadku diagnozowania poważnych chorób, takich jak rak. W tym przypadku chcemy minimalizować błędne diagnozy, aby uniknąć niepotrzebnych badań i leczenia dla osób, które nie potrzebują takiej interwencji.
- Np. Jak wiele wiadomości zaklasyfikowanych jako spam faktycznie jest spamem?

2.1.3 Recall (True Positive Rate, Sensitivity, Probability of Detection)

$$Recall = \frac{TP}{TP + FN}$$

- Stosowana gdy wymagamy od modelu wysoką wartość *True Positives* i chcemy zminimalizować liczbę *False Negatives*
- Proporcja *True Positives* do sumy *True Positives* i *False Negatives*
- W przypadku wykrywania rzadkich chorób. Tutaj celem jest maksymalizacja liczby poprawnych diagnoz, aby zapewnić pacjentom odpowiednie leczenie w czasie.
- Np. Jak wiele wiadomości spamu zostało zaklasyfikowanych jako spam?

2.1.4 F1-score

$$F1\text{-score} = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}}$$

- Metryka stosowana do porównywania modeli.
- Korzystne dla modeli z podobną wartością *Precision* i *Recall*.
- Średnia harmoniczna obu wartości.

2.1.5 Kompromis Precision/Recall

- *Precision* zmniejsza *Recall* i vice versa.

2.2 Regresja

2.2.1 Mean square error (błąd średnio kwadratowy):

$$MSE(x, y) = \frac{1}{n} \sum_{i=1}^n (f(x_i) - y_i)^2$$

- Błąd średnio-kwadratowy, najczęściej stosowany w przypadku regresji liniowej
- Stosowana ogólnie w regresjach
- Gdy funkcja f jest różniczkowalna, to MSE jest różniczkowalny ze względu na parametry funkcji f
- Równoważna z normą l_2 (Norma Euklidesowa)

2.2.2 Mean absolute error (błąd średnio bezwzględny):

$$MAE(x, y) = \frac{1}{n} \sum_{i=1}^n |f(x_i) - y_i|$$

- Błąd średniego odchylenia wartości bezwzględnej
- Stosowana ogólnie w regres
- Stosowane gdy jest dużo *outlier'ów* w zbiorze
- Równoważna z normą l_1 (Norma Manhattan)

2.2.3 Entropia:

$$H(X) = - \sum_{i=1}^n p(x_i) \log p(x_i)$$

- p - proporcja wystąpień wartości docelowych w danych regresyjnych
- Wyraża ilość informacji, którą możemy uzyskać po otrzymaniu instancji ze zbioru
- Tworzy zbilansowane drzewa
- Tak dzielimy zbiór tworząc drzewa, aby zysk entropii był jak największy (dowiadujemy się najwięcej dzieląc w ten sposób)

2.2.4 Gini:

$$Gini(X) = 1 - \sum_{i=1}^n p(x_i)^2$$

- Wyraża czystość zbioru
- Szybsza do obliczenia (względem entropii, nie trzeba liczyć logarytmu)
- Ma tendencję do izolowania najczęściej występującej klasy w osobnej gałęzi drzewa.
- Jest zerowa gdy wszystkie instancje w zbiorze są tej samej klasy
- Jest maksymalna gdy instancje są równomiernie rozłożone po klasach
- Wykorzystywana w algorytmie *CART* (Classification and Regression Tree).

2.2.5 Entropia krzyżowa:

$$H(p, q) = - \sum_{i=1}^n p(x_i) \log q(x_i)$$

- Stosowana w klasifikacji
- Wyraża oczekiwana ilość informacji o instancji, jeżeli zakodujemy ją przy użyciu modelu q zamiast p
- $p(x_i)$ - prawdziwy rozkład prawdopodobieństwa
- $q(x_i)$ - rozkład prawdopodobieństwa przewidywany przez model
- Podczas uczenia modelu q staramy się minimalizować entropię krzyżową, ponieważ to oznacza, że potrzebujemy mniejszej liczby bitów, żeby przewidzieć klasę instancji z rozkładu p (dla rozkładu p podczas uczenia zazwyczaj dokładnie znamy klasy każdej z instancji, więc entropia rozkładu p jest równa 0).

3 Regresja

3.1 Regresja

3.1.1 Regresja Liniowa

- Opiera się na założeniu, że istnieje liniowa zależność między zmiennymi wejściowymi a zmienną wyjściową.
- Dopasowuje hiperpłaszczyznę (określoną funkcją g), dla której średnia odległość instancji od wartości funkcji g jest najmniejsza.
- Mamy zbiór wektorów $A \subseteq \mathbb{R}^{n+1}$ i funkcję $f : A \rightarrow \mathbb{R}$, która przyporządkowuje każdemu wektorowi $x \in A$ wartość $f(x)$
- Każdy wektor ze zbioru A ma postać $x = [x_0, x_1, \dots, x_n]$, gdzie $x_0 = 1$
- Chcemy znaleźć funkcję $g : \mathbb{R}^{n+1} \rightarrow \mathbb{R}$ taką, że $g(x) = \Theta^T x$ dla pewnego wektora $\Theta \in \mathbb{R}^n$ i wektor Θ minimalizuje $MSE(x, \Theta) = \frac{1}{n} \sum_{i=1}^n (\Theta^T x - f(x))^2$
- Można pokazać, że jeżeli mamy wektor z wszystkich wartości $f(x)$ dla wszystkich wektorów ze zbioru A oraz X jest macierzą złożoną ze wszystkich wektorów z A , to $\Theta = (X^T X)^{-1} X^T$

3.1.2 Regresja wielomianowa

- Regresja liniowa, ale zamiast liniowej funkcji g używamy wielomianu g stopnia n
- Do każdej instancji x dodajemy nowe cechy $x_2 = x^2, x_3 = x^3, \dots, x_n = x^n$, następnie stosujemy regresję liniową na nowym zbiorze cech.

3.1.3 Regresja Logistyczna

- Wykorzystuj funkcję aktywacji $f(x) = \frac{1}{1+e^{-x}}$ (Sigmoid)
- Szacuje prawdopodobieństwo przynależności instancji do pewnej klasy.
- Stosuje funkcję *sigmoid* do zwrócenia prawdopodobieństwa (Sigmoid zwraca wartości między 0 a 1).

3.2 Gradient Descent

- Stosowany jeżeli nie można znaleźć rozwiązania analitycznego (np. w przypadku regresji logistycznej), a rozważana funkcja jest ciągła i różniczkowalna w rozważanej dziedzinie
- Zaczynamy ze startowym wektorem x z dziedziny analizowanej funkcji
- Obliczamy gradient funkcji w punkcie x
- Przesuwamy się w kierunku przeciwnym do wektora gradientu, ponieważ gwarantuje to najszybsze możliwe zmniejszanie się wartości funkcji
- Znajduje minimum lokalne.

3.2.1 SGD Stochastic Gradient Descent:

- Stosowany w przypadku, gdy zbiór danych jest bardzo duży
- Do obliczania gradientu wybieramy losowo podzbiór danych
- Znajduje minimum lokalne, szybciej niż *Gradient Descent*, ale nie jest tak dokładny.

3.3 Learning Curves

3.3.1 Bias

- Błąd generalizacji wynikający ze złych założeń. Prowadzi do *underfittingu*
- Model jest najprawdopodobniej zbyt prosty.

3.3.2 Variance

- Nadmierna wrażliwość na małą wariancję w zbiorze danych. Prowadzi do *overfittingu*
- Model jest najprawdopodobniej zbyt skomplikowany.

3.3.3 Irreducible Error

- Wynika z zaszumionego zbioru danych.

3.3.4 Kompromis między *Bias* a *Variance*

- Zwiększenie złożoności modelu prowadzi do zwiększenia *Variance* i zmniejszenia *Bias'u* i vice versa.

3.4 Regularyzowane modele liniowe

3.4.1 Ridge Regression

- Regularyzowana wersja *Regresji Liniowej*
- Zmusza model do utrzymywania małych wag
- Używa normy l_2

3.4.2 Lasso Regression

- Regularyzowana wersja *Regresji Liniowej*
- Używa normy l_1
- Ma tendencje do usuwania wag dla najmniej ważnych cech
- Zwraca *Rzadki model* (Dużo zer w polach wag)

3.4.3 Early Stopping

- Zatrzymuje proces uczenia w momencie gdy *błąd walidacji* osiąga minimum.

4 SVM (Support Vector Machines)

- Algorytm klasyfikacji oparty o zasadę największego marginesu.

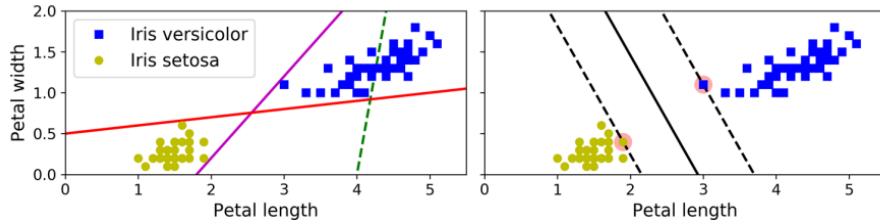


Figure 3: Porównanie regresji liniowej (lewy wykres) z SVM (prawy wykres)

- Wrażliwy na skalowanie danych (Zawsze skalować przed użyciem)

4.1 Hard Margin Classification

- Wszystkie instancje muszą się znaleźć poza marginesem.
- Działa tylko wtedy, gdy dane da się liniowo rozdzielić.
- Wrażliwy na *outliers*'y

4.2 Soft Margin Classification

- Elastyczny model
- Szyka balansu między posiadanym jak największym marginesem, a limitowaniem liczby jego naruszeń.

4.3 Nieliniowa klasyfikacja SVM

- Używaj kiedy dane nie da się rozdzielić liniowo.

4.3.1 Polynomial Kernel

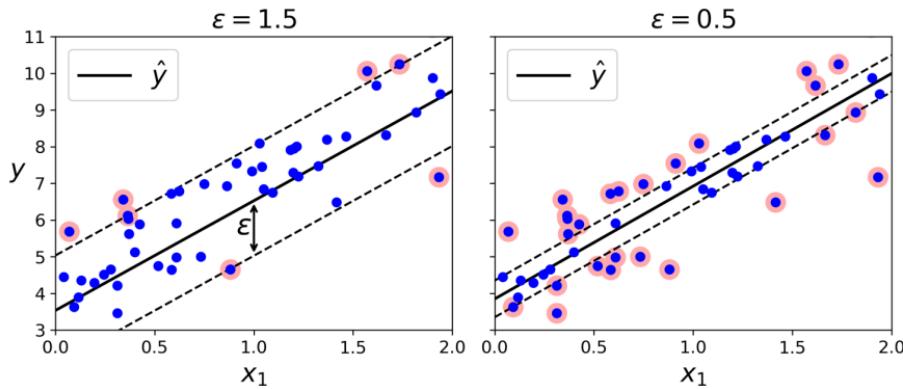
- Sztuczka dzięki której możemy dostać wyniki, jakbyśmy korzystali z wielomianowego modelu bez użycia go.

4.4 Regresor SVM

- By działał musimy odwrócić jego zadanie - zmieścić jak najwięcej instancji w jak najmniejszym marginesie.
- Model jest ϵ niewrażliwy, czyli dodawanie więcej instancji znajdujących się w marginesie nie wpływa na zdolność przewidywania modelu.
- Do rozwiązywania nieliniowych modeli użyj **kernelized SVM model**

4.5 Drzewa Decyzyjne

- Stosowany do klasyfikacji i regresji

Figure 4: Wpływ ϵ na wydajność regresji

- Nie wymaga przygotowania danych, nie trzeba skalować ani centrować
- Scikit używa algorytmu **CART** (próbuje zachłannie minimalizować współczynnik Gini) do trenowania drzew decyzyjnych
- Algorytm **CART** w celu ustalenia miejsca podziału oblicza wartość $J(k, t_k) = \frac{m_{lewa}}{m} * G_{lewa} + \frac{m_{prawa}}{m} * G_{prawa}$, gdzie G_{lewa} i G_{prawa} wyrażają nieczystości lewej i prawej części po podziale, a m_{lewa} i m_{prawa} to liczba instancji w lewej i prawej części, m to liczba wszystkich instancji
- Obrót przestrzeni instancji może całkowicie zmieniać wygenerowane drzewo i jego złożoność.

4.5.1 Hiperparametry

- Bez żadnych ograniczeń model bardzo szybko przeucza się (Wtedy go nazywamy nieparametrycznym, opisany wyżej)
- **Regularizacja** jest procesem mającym przeciwdziałać przeuczeniu, przez dobranie odpowiednich hiperparametrów
 - Najważniejszą wartość jaką możemy dostrajać jest ograniczenie maksymalnej głębokości drzewa (Domyślnie jest ∞)

4.5.2 Regresja

- Struktura drzewa przypomina tą z problemu klasyfikacji.
- Możemy uznać problem regresji jako problem klasyfikacji z nieograniczoną liczbą klas, którą możemy regulować przez maksymalną głębokość drzewa.

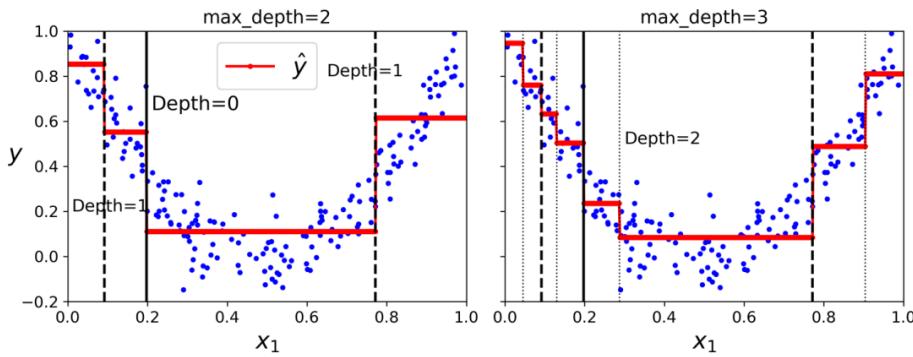


Figure 5: Predykcje 2 regresorów o różnych maksymalnych głębokościach

5 Ensemble Learning i Random Forests

- Wykorzystywana jest moc przyjaźni (ang. *Power of friendship*).
- Stosujemy zasadę *mądrości tłumu* - jeżeli mamy wiele klasyfikatorów, to możemy je zagregować w grupę klasyfikatorów znacznie zwiększaając wydajność modelu.
- Wszystkie klasyfikatory powinny być od siebie niezależne
- Redukuje *Bias* i *Variance*

5.1 Klasifikacja

5.1.1 Hard Voting Classifier

- Wybiera klasę, która jest dominantą zbioru propozycji klas zwroconych przez klasyfikatory.

5.1.2 Soft Voting Classifier

- Wykorzystuje prawdopodobieństwa zwracane przez model, następnie uśrednia je i wybiera klasę z najwyższym średnim prawdopodobieństwem.

5.2 Bagging i Pasting

- Wykorzystują wiele instancji klasyfikatora tego samego typu, ale trenowanych na różnych podzbiorach danych.
- **Bagging** (Bootstrap Aggregating) polega na losowaniu instancji ze zwarcaniem (zastępowaniem) i trenowaniu na nich różnych klasyfikatorów, a następnie wykorzystaniu metody *hard voting* do wyboru klasy.
- **Pasting** jest podobny do *Bagging'u*, ale zamiast losować instancje ze zwarcaniem, losuje je bez zwarcania, co oznacza, że każdy klasyfikator może być trenowany tylko na części danych, a liczba klasyfikatorów jest ograniczona przez liczbę instancji w zbiorze treningowym.

5.3 Random Forests

- Zbiór drzew decyzyjnych

- Dodaje extra losowość
- Umożliwia łatwe sprawdzenie istotności pewnej cechy
- Jeżeli zastosujemy *Bagging* na drzewach decyzyjnych, to otrzymamy *Random Forest*
- Agreguje predykcje ze wszystkich drzew i wybiera klasę o największej ilości głosów (hardvoting)
 - Grupa drzew decyzyjnych
 - Każdy uczy się na innym podzbiorze zbioru danych

5.3.1 Extremely Randomized Trees Ensemble

- Szybciej się uczy
- Stosuje losowe progi dla każdej cechy

5.4 Boosting

- Łączy wiele *weak learners* w *strong learner*
- Trenuje predyktory sekwencyjnie
 - Każdy kolejny próbuje poprawić błędy poprzedniego

5.4.1 AdaBoost

- Adaptive Boosting
- Zwraca uwagę na instancje słabo dopasowane przez poprzednie predyktory.
- Nie skaluje się dobrze.

5.4.2 Gradient Boosting

- Możemy go użyć z różnymi funkcjami straty
- Dopasowuje nowy predyktor do pozostałego błędu przez poprzedni model
- **XGBoost**
 - Aktualnie najlepszy klasyfikator (razem z CatBoostem).

5.5 Stacking

- Metoda podobna do *Voting Classifier'a*, ale zamiast używać prostych funkcji do agregacji predykcji, trenuje model, aby nauczył się jak łączyć predykcje innych modeli
- Możliwe jest stosowanie bardziej zagnieźdzonych architektur, w których występują kolejne warstwy modeli.

6 Redukcja Wymiarów

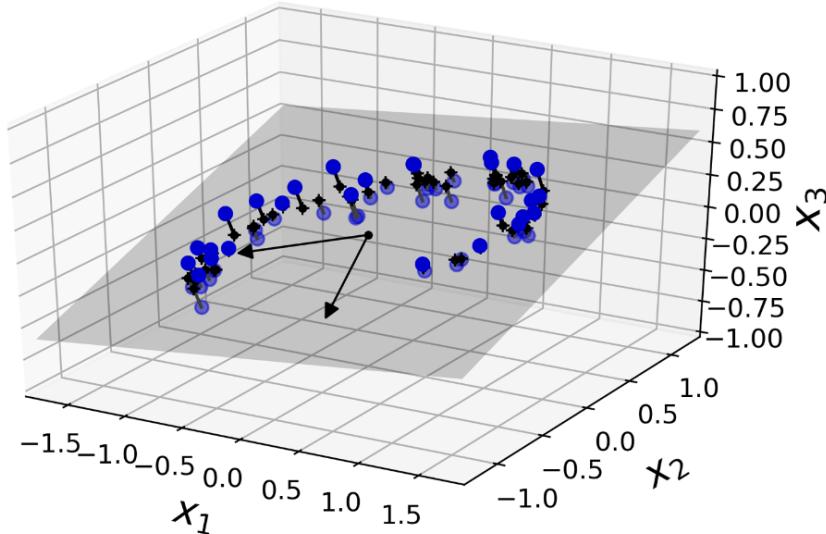


Figure 6: Rzutowanie danych na inną przestrzeń

- Stosujemy do uproszczenia zbioru danych w celu przyspieszenia procesu uczenia modelu
- Prowadzi do utraty części informacji, umożliwiając jednocześnie lepszą wydajność modelu
- Może być również wykorzystywana do wizualizacji danych.

6.1 Curse of Dimensionality

- Odnosi się do zjawiska, w którym dodanie kolejnych wymiarów do zbioru danych powoduje znaczny (eksponencjalny) wzrost wymaganej ilości danych do zachowania odpowiedniej gęstości danych.

6.2 PCA - Principal Component Analysis

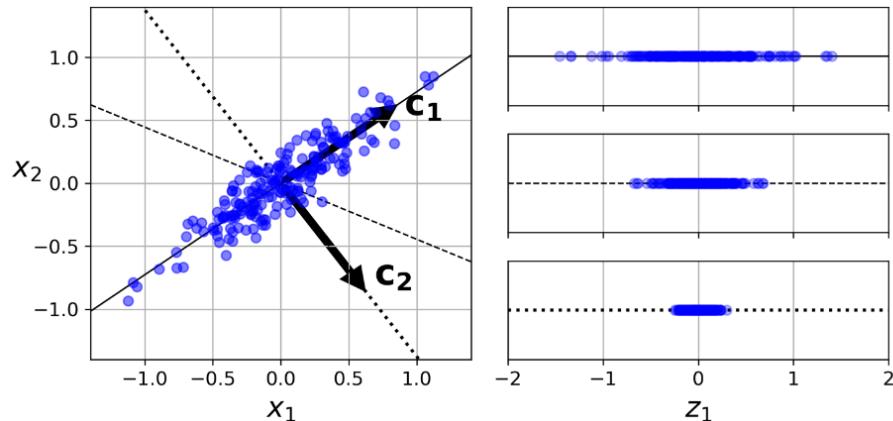


Figure 7: PCA - Principal Component Analysis

- Jest to metoda redukcji wymiarów, w której wybieramy kierunki, które zachowują najwięcej informacji
- Kierunki te są nazywane **principal components**
- PCA znajduje kierunki, które minimalizują *średnią kwadratową odległość* między punktami danych a ich rzutami na kierunki
- Staramy się znaleźć takie kierunki, dla których występuje największa wariancja danych
- Na początku standaryzujemy dane, aby średnie wartości były równe 0
- Znajdujemy bazę przestrzeni, która jest najbardziej zbliżona do danych pod względem *średniej kwadratowej odległości* dla punktów danych i ich rzutów na bazę
- Istnieje szybszy algorytm randomizowany, który znajduje przybliżone rozwiązanie.

6.2.1 SVD - Singular Value Decomposition

- Jest to metoda rozkładu macierzy na iloczyn 3 macierzy
- Umożliwia wyznaczenie kierunków, które zachowują najwięcej informacji
- Stosowana w PCA
- Uogólnienie wartości własnych i wektorów własnych na macierze niekwadratowe
- Największe wartości singularne odpowiadają kierunkom, które zachowują najwięcej informacji.

6.2.2 Incremental PCA

- minibatch, out-of-core, praca na strumieniach, trzeba podać liczbę wymiarów
- Czyli w sumie po prostu PCA na online(minibatches), gdzie nie ładujemy całego zestawu danych na raz do modelu

6.3 Rozmaitości

- Są to zbiory danych, które mogą być zredukowane do mniejszej liczby wymiarów, ale nie muszą być przestrzeniami liniowymi
- W małej skali wyglądają jak przestrzenie liniowe, ale w większej skali mogą mieć kształty przeróżne
- Zastosowanie dla nich algorytmu PCA może prowadzić do zbyt intensywnej utraty informacji
- Istnieją algorytmy, które pozwalają na redukcję wymiarów dla takich zbiorów danych.

6.3.1 LLE - Locally Linear Embedding

- Algorytm ten znajduje lokalne zależności między punktami danych, a następnie próbuje zachować te zależności w niższej wymiarowości
- Jest to algorytm nienadzorowany
- Może prowadzić do zniekształcenia danych w dużej skali
- W pierwszym kroku znajduje najbliższych sąsiadów dla każdego punktu danych
- Następnie znajduje wagi, które pozwalają na rekonstrukcję każdego punktu danych jako kombinacji liniowej jego najbliższych sąsiadów
- W ostatnim kroku rzutuje dane na przestrzeń o niższej wymiarowości, zachowując lokalne zależności.

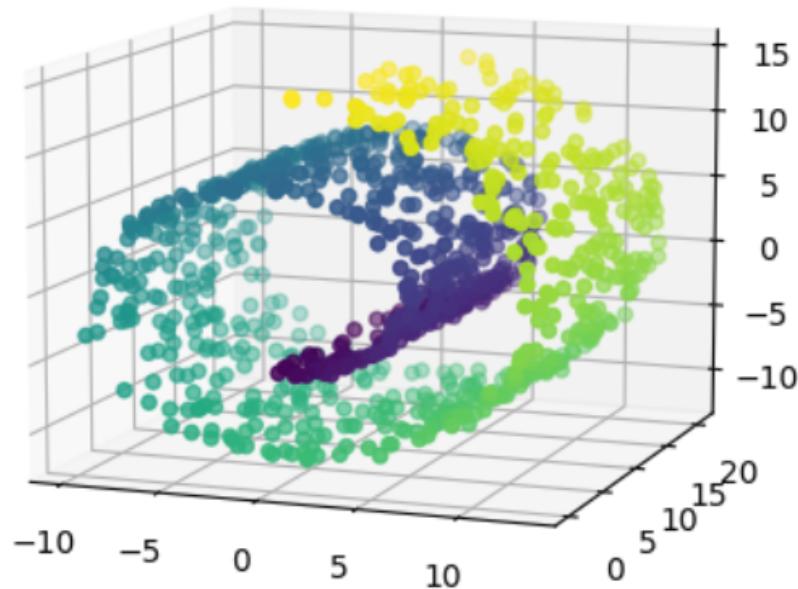


Figure 8: Rozmaitość - przykład Swiss Roll

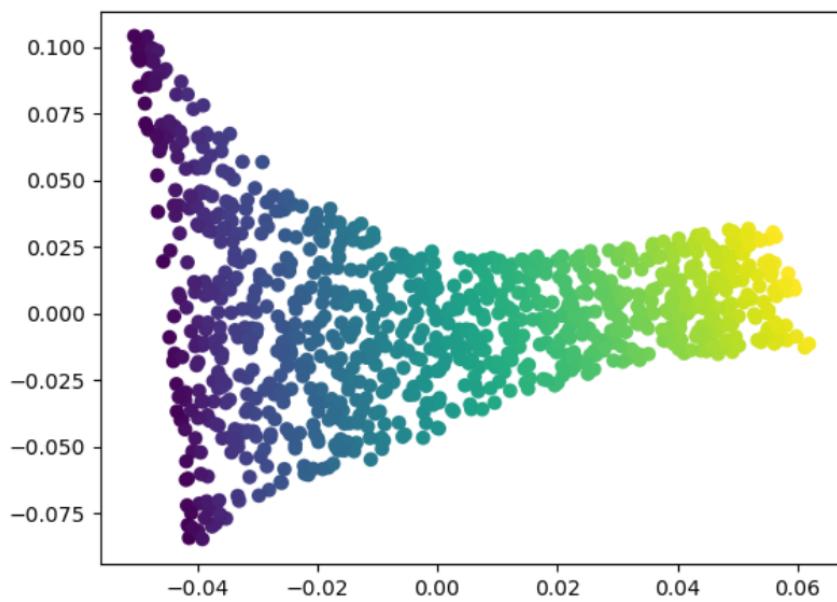


Figure 9: Swiss Roll po zastosowaniu LLE

7 Uczenie nienadzorowane

Kategorie uczenia nienadzorowanego:

- Klasteryzacja *clustering*
 - identyfikacja klas
 - redukcja wymiarów
 - analiza danych (po klasteryzacji, dla każdej klasy osobno)
 - uczenie częściowo nadzorowane
 - segmentacja obrazu, detekcja, kompresja
- Detekcja anomalii
 - detekcja wartości odstających, *outlierów*
- Estymacja gęstości *density estimation*

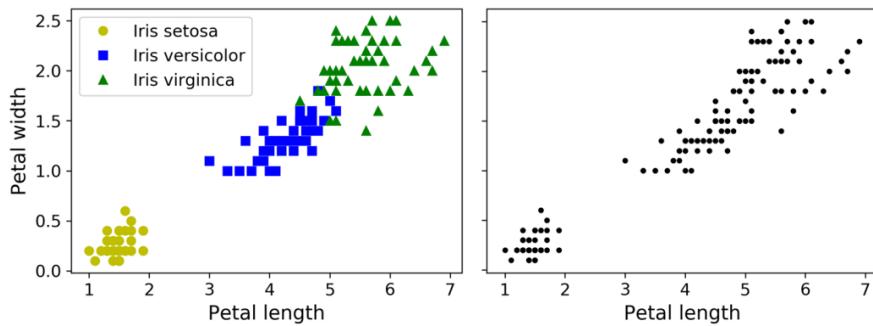


Figure 10: Różnica między danymi wejściowymi w uczeniu nadzorowanym i nienadzorowanym

7.1 Soft Clustering

- Do każdej instancji przypisywany jest wektor wyników przypisania do każdego z klastrów.
 - Wynikiem może być np. dystans pomiędzy instancją a centroidą.

7.2 Hard Clustering

- Każda instancja jest przypisana do 1 klastra.

7.3 DBSCAN

- Algorytm DBSCAN (*Density-Based Spatial Clustering of Applications with Noise*) jest algorytmem klasteryzacji, który znajduje skupiska o wysokiej gęstości
- Algorytm ten znajduje skupiska o wysokiej gęstości, a także punkty odstające
- Algorytm ten nie wymaga określenia liczby klastrów
- Wymaga określenia dwóch parametrów: *eps* i *min_samples*
 - *eps* - maksymalna odległość między dwoma punktami, aby zostały one uznane za sąsiadów
 - *min_samples* - minimalna liczba punktów, aby uznać je za rdzeń (wliczając w to punkt, dla którego szukamy sąsiadów)

- Wszystkie instancje, które nie są rdzeniami, ale mają sąsiadów, są uznawane za brzegi, wchodzą w skład tego samego klastra, co ich rdzeń
- Instancje, które nie są ani rdzeniami, ani brzegami, są uznawane za anomalią (nie należą do żadnego klastra)

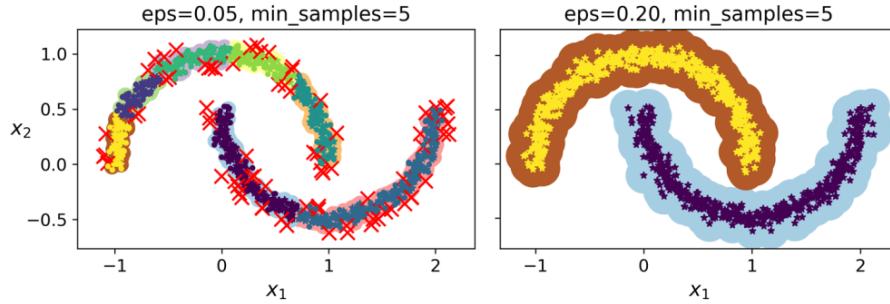


Figure 11: Przedstawienie działania alg. DBSCAN

7.4 KNN - K-nearest neighbors

- Algorytm KNN (*K-nearest neighbors*) jest algorytmem klasyfikacji, który przypisuje nową instancję do klasy, która jest najbardziej popularna wśród k najbliższych sąsiadów
- W przypadku regresji algorytm ten zwraca średnią wartość k najbliższych sąsiadów
- Jeżeli k jest zbyt małe, to algorytm ten jest podatny na szумy
- W przypadku remisu:
 - wybór pierwszej napotkanej instancji (implementacja scikit-learn)
 - wybór losowy
 - wybór liczniejszej klasy
 - wybór wartości najbliższej instancji (tylko dla regresji)
 - brana pod uwagę jest odległość od instancji do sąsiada (średnia ważona) (tylko dla regresji)
 - średnia wartość wszystkich instancji o tej samej odległości (tylko dla regresji)

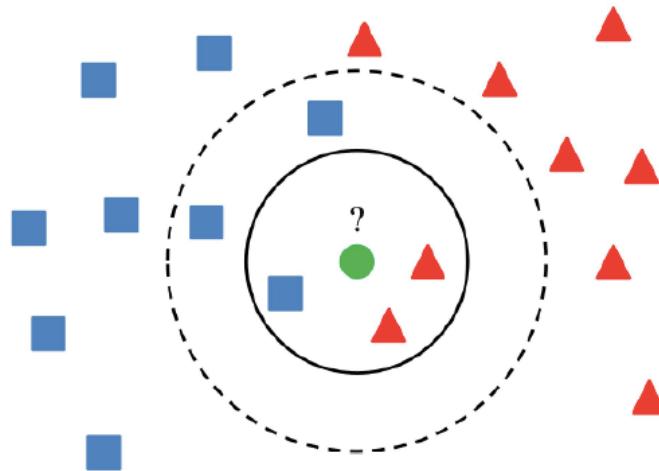


Figure 12: Przedstawienie działania alg. KNN

7.5 Algorytm centroidów (k-średnich) *K-Means*

- Algorytm centroidów (k-średnich) *K-Means* jest jednym z najpopularniejszych algorytmów klasteryzacji.
- Algorytm stara się znaleźć środek każdego z k skupisk
- Algorytm ten przypisuje każdy punkt danych do najbliższego centroidu, a następnie przesuwa centroidy tak, aby minimalizować średnią kwadratową odległość między punktami danych a ich centroidami
- k jest parametrem algorytmu, który musi zostać określony przez użytkownika
- Jest zbieżny
- Nie gwarantuje znalezienia optimum (zależy od kroku 1)
 - Domyślnie algorytm uruchamiany jest 10 razy
 - Wybierany jest model z najmniejszą **inercją**: średnio-kwadratowa odległość między instancjami i ich centroidami
 - * zmierz odległość między instancjami a ich centroidami
 - * zsumuj kwadraty w/w odległości w ramach klastra
 - * zsumuj wartości inercji dla wszystkich klastrów
- Przedstawieniem wyniku działania algorytmu jest Diagram Woronoja *Voronoi*
- *K-Means++*
 - Nowsza wersja
 - W bardziej optymalny sposób dobiera początkowe centroidy
- *Mini batch K-Means*
 - Używa *batch* zamiast całego zbioru danych
- W przypadku równej odległości do więcej niż jednego centroida instancja jest przypisywana do **losowego centroidu**, pierwszego napotkanego centroidu lub wybierany jest centroid grupujący większą liczbę instancji

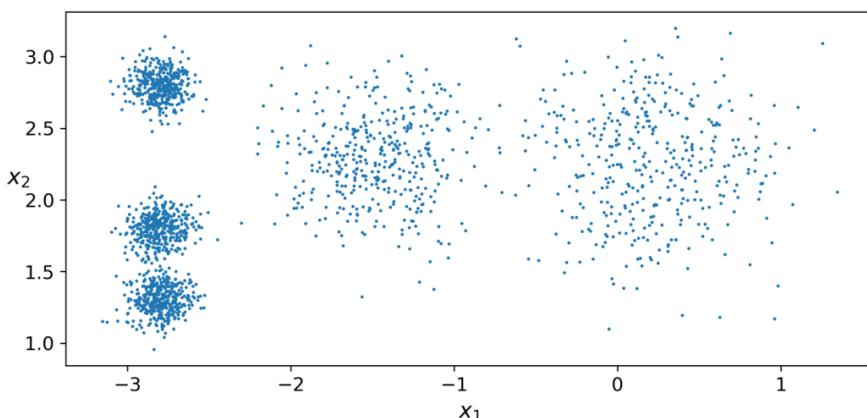


Figure 13: Przykładowy rozkład danych

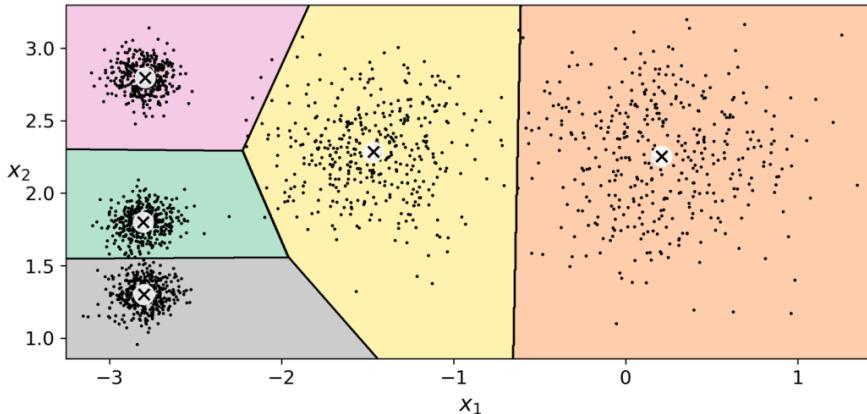


Figure 14: Diagram Woronoja *Voronoi* wyznaczony przez alg. K-Means

7.5.1 Wyznaczanie liczby klastrów

Do wyznaczenia liczby klastrów nie wystarcza sama inercja, ponieważ maleje ona wraz ze zwiększeniem się liczby klastrów.

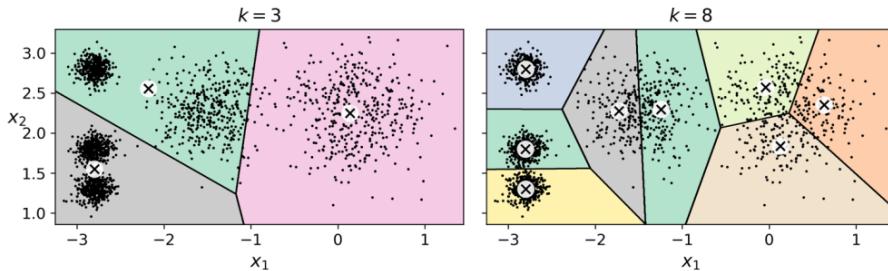


Figure 15: Przykład podziału na niepoprawną liczbę klastrów

Inercja nie wystarcza, ale można ją wykorzystać. Wystarczy wyznaczyć inercję dla różnych wartości k i wybrać tę, która jest na ‘zgięciu’ wykresu.

Do wyznaczenia liczby klastrów możemy również wykorzystać **Wskaźnik sylwetkowy, silhouette score**. Wskaźnik bierze pod uwagę średnią odległość pomiędzy obserwacjami wewnętrz grupy (a_i) i średnią odległość pomiędzy obserwacjami do najbliższej “obcej” grupy (b_i) i dany jest wzorem:

$$s = \frac{1}{k} \sum_{i=1}^k \frac{b_i - a_i}{\max(a_i, b_i)}$$

- Najlepsza wartość: 1
- Najgorsza wartość: -1
- Nakładające się wartości: w pobliżu 0

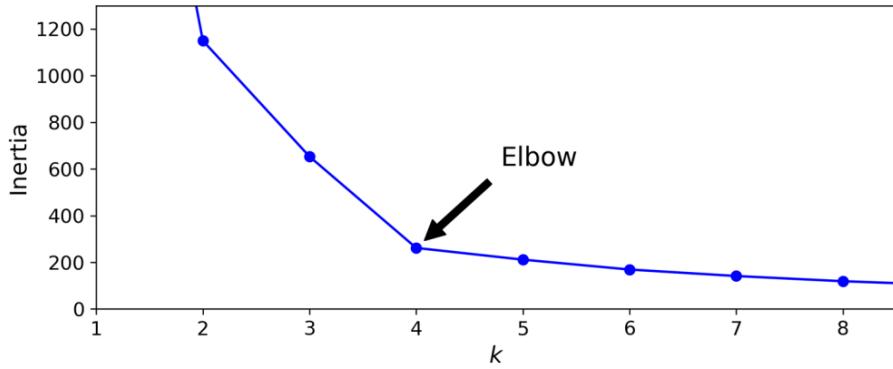


Figure 16: Wykorzystanie inercji do wyznaczenia liczby k

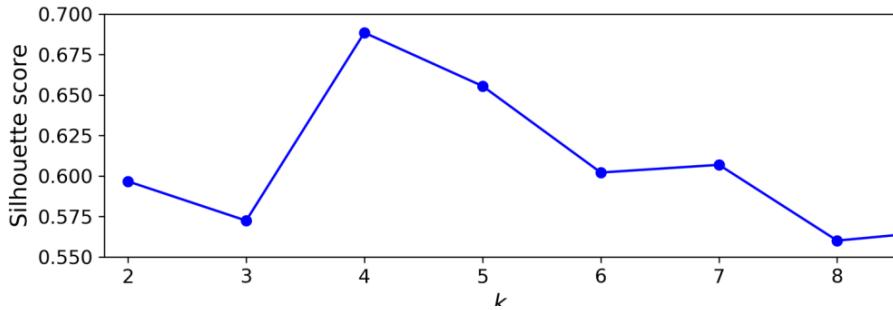


Figure 17: Wykorzystanie wskaźnika sylwetkowego do wyznaczenia liczby k

8 Sieci neuronowe - wprowadzenie

8.1 Perceptron

- Każdy neuron jest jednostką liniową, po której następuje funkcja aktywacji
- Sposób działania:
 - oblicz sumę wejść $z = w_1x_1 + w_2x_2 + \dots + w_nx_n = x^T w$
 - zastosuj funkcję schodkową: $h_w(x) = \text{step}(z)$
- Ograniczenia:
 - Nie potrafią rozwiązać pewnych trywialnych problemów, np. XOR. W takich przypadkach stosuje się sieci wielowarstwowe (MLP)

8.1.1 Uczenie perceptronu

- Uczenie perceptronu polega na znalezieniu wektora wag w , który pozwoli na poprawne sklasyfikowanie jak największej liczby instancji
- Wagi są aktualizowane na podstawie błędu predykcji według wzoru $w_{i,j}^{(\text{nastpna iteracja})} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$
 - $w_{i,j}$ - waga połączenia między neuronem i a neuronem j
 - η - współczynnik uczenia
 - y_j - wartość oczekiwana

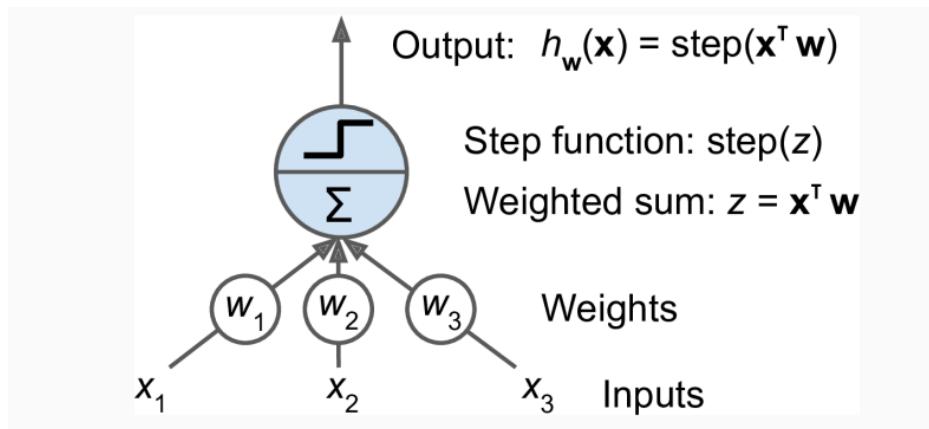


Figure 18: Schemat poglądowy perceptronu

- \hat{y}_j - wartość przewidziana
- x_i - wartość wejścia

8.2 Funkcje aktywacji

8.2.1 Dlaczego potrzebujemy funkcji aktywacji?

- Konieczność nieliniowości
 - Jeżeli używamy liniowych funkcji aktywacji, to kilka nałożonych na siebie warstw jest równoważna z jedną warstwą.
 - Sieć neuronowa będzie zachowywać się jak jedna warstwa neuronów (dla macierzy W_1 i W_2 będzie można znaleźć macierz W , która będzie równoważna działaniu sieci neuronowej, $W = W_2W_1$)
- Potrzebujemy dobrze zdefiniowanej niezerowej pochodnej
 - *Gradient Descent* robi progres w każdym kroku.

8.2.2 Funkcje aktywacji - przegląd

Poniższa lista jest ułożona od najlepszych funkcji aktywacji (oprócz softmax).

1. SeLU (Scaled Exponential Linear Unit)

- Najlepsze dla *Głębokiej Sieci Neuronowej*
- Potrafi się samodzielnie znormalizować
 - Rozwiązuje problem znikających i eksplodujących gradientów.
- Warunki zbioru danych:
 - Wszystkie warstwy muszą być gęste
 - Dane muszą być standaryzowane (średnia = 0, odchylenie standardowe = 1).

$$\text{SeLU}(z) = \begin{cases} \lambda\alpha(e^z - 1) & \text{if } z < 0 \\ \lambda z & \text{if } z \geq 0 \end{cases}$$

2. ELU (Exponential Linear Unit):

- ELU jest podobne do SeLU, ale nie jest zależne od normalizacji danych.
- Funkcja ELU ma mniejszą podatność na problem znikających i wybuchających gradientów.

$$ELU(z) = \begin{cases} \alpha(e^z - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

3. Leaky ReLU

- Leaky ReLU jest modyfikacją funkcji ReLU, która rozwiązuje problem “martwych neuronów” (neury, które zawsze mają wartość 0 dla niektórych danych wejściowych).

$$LeakyReLU(z) = \max(\alpha z, z)$$

4. ReLU (Rectified Linear Unit):

- ReLU jest jedną z najpopularniejszych funkcji aktywacji. Ma dobrą zdolność do modelowania nielinowych relacji.
- Jeżeli wszystkie wartości danych treningowych są ujemne, to neuron z ReLU się nie uczy

$$ReLU(z) = \max(0, z)$$

5. Tanh (tangens hiperboliczny):

- Funkcja tanh jest splotem funkcji sigmoidalnej i może generować wartości z przedziału (-1, 1).
- Funkcja ta ma symetryczny kształt wokół zera i może być przydatna w przypadkach, gdy oczekuje się zarówno wartości dodatnich, jak i ujemnych.

$$\tanh(z) = 2\sigma(2z) - 1$$

6. logistic (funkcja sigmoidalna):

- Funkcja sigmoid, znana również jako funkcja logistyczna, generuje wartości z przedziału (0, 1).
- Często jest używana w warstwie wyjściowej modeli binarnych do przewidywania prawdopodobieństwa przynależności do jednej z dwóch klas.

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

7. Softmax

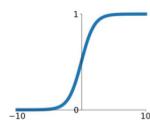
- Funkcja aktywacji wykorzystywana w warstwie wyjściowej klasyfikatorów wieloklasowych, generuje rozkład prawdopodobieństwa.
- Opisuje pewność dopasowania do każdej klasy.

$$Softmax(z)_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$$

Activation Functions

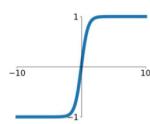
Sigmoid

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



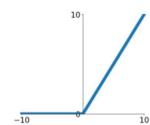
tanh

$$\tanh(x)$$



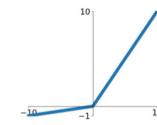
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$



Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

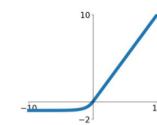


Figure 19: Wykresy wybranych funkcji aktywacji

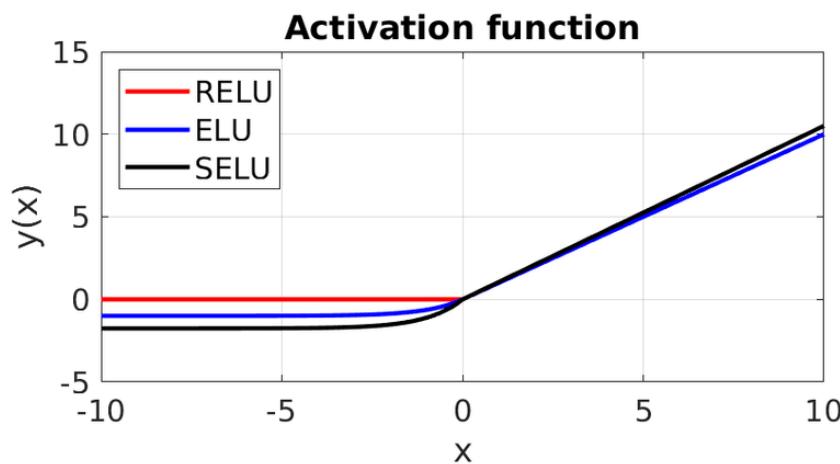


Figure 20: Wykresy wybranych funkcji aktywacji - porównanie ReLU, ELU, SeLU

8.3 Warstwy

8.3.1 Warstwa gęsta

- Każdy neuron jest połączony z każdym neuronem z poprzedniej warstwy
- Wagi połączeń są zapisane w macierzy wag W^*
- Każdy neuron ma dodatkowy parametr b , który jest nazywany *biasem* - w innym przypadku dla wektora zerowego na wejściu, na wyjściu otrzymalibyśmy wektor zerowy (jeżeli funkcja aktywacji ma punkt stały w 0)

9 Głębokie sieci neuronowe

Głębokie sieci neuronowe (DNN - Deep Neural Networks) to sieci neuronowe z wieloma warstwami ukrytymi

9.1 Budowa modelu

9.1.1 Keras Sequential API

- Najprostszy sposób tworzenia sieci neuronowej
- Zakłada, że sieć jest sekwencją warstw
- Warstwy dodajemy jako instancje odpowiednich klas z pakietu keras.layers
- parametr można przekazywać jako ciągi znaków. Jest to zapis uproszczony: zamiast "relu" można przekazać keras.activations.relu
- Normalizację danych można wykonać za pomocą warstwy keras.layers.Normalization, lub keras.layers.Flatten, albo zrobić samemu wcześniej

```
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])
```

9.1.2 Keras Functional API

- Pozwala na tworzenie bardziej skomplikowanych architektur sieci neuronowych
- Pozwala na tworzenie grafów obliczeniowych, w których nie wszystkie warstwy są połączone ze sobą w sekwencji
- Pozwala na tworzenie wielu modeli, które mają współdzielone warstwy
- Do tworzenia modelu wykorzystujemy klasę tf.keras.Model, podaje się w niej warstwy wejściowe i wyjściowe
- Do tworzenia warstw wykorzystujemy klasę tf.keras.layers, podobnie jak w przypadku Sequential API
- Łączenie warstw odbywa się za pomocą operatora (`warstwa`)(`wejście`), podobnie jak w przypadku wywoływanego funkcji, co oznacza, że warstwa jest wywoływana na wejściu otrzymanym z poprzedniej warstwy będącej argumentem wywołania

```
import tensorflow as tf

input_ = tf.keras.layers.Input(shape=[28, 28])
flatten = tf.keras.layers.Flatten(input_shape=[28, 28])(input_)
```

```
hidden1 = tf.keras.layers.Dense(300, activation="relu")(flatten)
hidden2 = tf.keras.layers.Dense(100, activation="relu")(hidden1)
concat = tf.keras.layers.concatenate([input_, hidden2])
output = tf.keras.layers.Dense(10, activation="softmax")(concat)
model = tf.keras.Model(inputs=[input_], outputs=[output])
```

9.2 Kompilacja i uczenie modelu

Po utworzeniu modelu należy go skompilować za pomocą metody `compile()`. Metoda ta przyjmuje następujące parametry:

- **optimizer**: Określa **optymalizator** używany do aktualizacji wag modelu podczas procesu uczenia. Optymalizator reguluje sposób, w jaki model aktualizuje wagi na podstawie straty i algorytmu optymalizacji. Ich argumentem jest m.in. `learning_rate`. Przykładowe optymalizatory:
 - ***SGD*** - Stochastic Gradient Descent
 - ***Momentum*** - SGD z pędem (wykorzystaniem historii aktualizacji)
 - ***Nesterov Accelerated Gradient*** - SGD z pędem Nesterova
 - * Szybka zbieżność
 - * Minimalnie szybsza od *Momentum*
 - ***AdaGrad*** - Adaptive Gradient, nie wykorzystuje pędu, ale dostosowuje współczynnik uczenia dla każdego parametru na podstawie jego historii aktualizacji
 - * Działa dobrze dla prostych problemów kwadratowych
 - * Ryzyko nie osiągnięcia minimum
 - ***Adam*** - Adaptive Moment Estimation, wykorzystuje pęd i historię aktualizacji
 - * Wariancje *Adam*:
 - ***Nadam*** (*Adam* + Nesterov) - Generalnie jest lepsza od *Adam*
 - * Wiele lepszy niż *AdaGrad*
 - * **Problemy Adaptive estimation methods**
 - M. in. Adam, Nadam, RMSProp, Adagrad
 - Mogą źle generalizować zbiory danych
 - Jak są jakieś problemy użyj *Nesterov Accelerated Gradient*
- **loss**: Określa **funkcję straty**, która jest używana przez optymalizator do oceny odchylenia między przewidywaniami modelu a rzeczywistymi wartościami. Przykładowe funkcje straty to ‘mean_squared_error’, ‘categorical_crossentropy’, ‘binary_crossentropy’. Wybór odpowiedniej funkcji straty zależy od rodzaju problemu i rodzaju wyjścia modelu.
- **metrics**: Określa **metryki**, które będą używane do oceny wydajności modelu. Przykładowe metryki to ‘accuracy’, ‘precision’, ‘recall’, ‘mean_absolute_error’ itp. Metryki służą do monitorowania wydajności modelu podczas uczenia i ewaluacji.
- Inne opcjonalne argumenty, takie jak `loss_weights`, `sample_weight_mode`, `weighted_metrics`, które pozwalają na bardziej zaawansowane konfigurowanie procesu komplikacji modelu.

```
model.compile(loss="adam",
              optimizer="sgd",
              metrics=["accuracy"])
```

A następnie wytrenować model za pomocą metody `fit()`. Metoda ta przyjmuje następujące parametry:

- **x: Dane wejściowe** do modelu.
- **y: Dane wyjściowe** (etykiety) odpowiadające danym wejściowym x.
- **batch_size**: Określa liczbę próbek, które są przetwarzane jednocześnie przez model w trakcie jednej iteracji. Mniejsza - zapewnia generalizację, większa - szybsze uczenie, ale ma tendencję do overfittingu.
- **epochs**: Określa liczbę **epok uczenia** - pełnych przebiegów przez zbiór treningowy. Każda epoka oznacza jedno przejście przez cały zbiór treningowy. Mniejsza - ma tendencję do underfittingu, większa - ma tendencję do overfittingu.
- **validation_data**: **Dane walidacyjne** używane do oceny wydajności modelu na każdej epoce. Może to być krotka (`x_val`, `y_val`) zawierająca dane wejściowe i oczekiwane wyjście dla danych walidacyjnych.
- **callbacks**: Lista obiektów zwrotnych (callbacks), które są wywoływanie podczas treningu w różnych momentach. Przykłady to ModelCheckpoint, EarlyStopping, TensorBoard itp. Callbacks pozwalają na dostosowywanie zachowania treningu w zależności od określonych warunków.
- **verbose**: Określa tryb wyświetlania informacji podczas treningu. Może przyjąć wartość 0 (bez wyświetlania), 1 (wyświetlanie paska postępu) lub 2 (wyświetlanie jednej linii na epokę).
- Inne opcjonalne argumenty, takie jak `validation_split`, `shuffle`, `class_weight` itp., które pozwalają na bardziej zaawansowane konfigurowanie procesu treningu modelu.

```
history = model.fit(
    X_train,
    y_train,
    batch_size=32,
    epochs=10,
    validation_data=(X_valid, y_valid),
    callbacks=[early_stopping_cb],
    verbose=1
)
```

9.3 Callbacks

Callbacki pozwalają na wykonywanie dodatkowych operacji w trakcie uczenia modelu. Użyteczne jak mamy długi czas uczenia. Przykładowe callbacki ich zastosowania:

- **ModelCheckpoint** - Zapisywanie punktów kontrolnych

- **EarlyStopping** - zatrzymanie uczenia, jeżeli nie nastąpi poprawa wyniku przez 10 epok (bardzo częste zastosowanie)
- **TensorBoard** - zapisywanie logów do wykorzystania w TensorBoard

```

checkpoint_cb = keras.callbacks.ModelCheckpoint(
    "my_keras_model.h5",
    save_best_only=True
)

early_stopping_cb = keras.callbacks.EarlyStopping(
    patience=10,
    restore_best_weights=True
)

tensorboard_cb = keras.callbacks.TensorBoard(
    log_dir=".my_logs",
    histogram_freq=1,
    profile_batch=100
)

```

Callbacki dodajemy w parametrze `callbacks` metody `fit`.

9.4 Analiza procesu uczenia

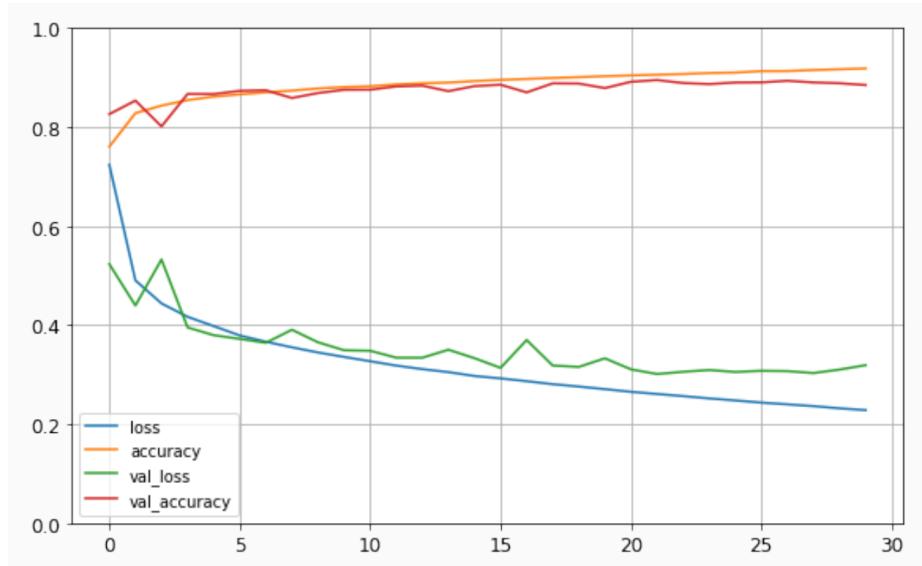


Figure 21: Wykres wartości miar w trakcie procesu uczenia

- **Loss** - miara, która określa, jak bardzo wyniki modelu różnią się od oczekiwanych wartości.
- **Accuracy** - miara, która określa, jak dokładnie model przewiduje klasy lub etykiety dla danych.

- **Recall** - miara, która określa, jak wiele pozytywnych przypadków zostało wykrytych przez model.
- **Precision** - miara, która określa, jak wiele pozytywnych przypadków zostało poprawnie określonych przez model.
- **Val_loss** - strata obliczana na danych walidacyjnych, służy do monitorowania uczenia modelu i unikania przeuczenia.
- **Val_accuracy** - dokładność obliczana na danych walidacyjnych, pomaga ocenić, jak dobrze model generalizuje na nowych danych.

Przykłady funkcji strat zostały przedstawione na początku dokumentu.

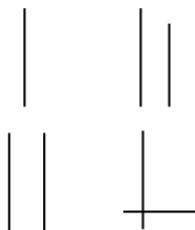


Figure 22: Loss

9.5 Przeszukiwanie przestrzeni hiperparametrów

9.5.1 SciKit-Learn

- `RandomizedSearchCV` - losowe przeszukiwanie przestrzeni hiperparametrów
 - Lepsze od `GridSearch`
- `GridSearchCV` - przeszukiwanie przestrzeni hiperparametrów siatką wartości parametrów
 - Wydajny gdy funkcja jest szybka w obliczeniu. (Model mało skomplikowany)
- Jak mamy bardziej złożony model to polecam bibliotekę `Optuna`

9.5.2 Keras Tuner

- `RandomSearch` - losowe przeszukiwanie przestrzeni hiperparametrów

10 Konwolucyjne sieci neuronowe

- Konwolucyjne sieci neuronowe (CNN - Convolutional Neural Networks)
- CNN są stosowane do przetwarzania wielowymiarowych danych, takich jak obrazy, wideo itp.
- Wykorzystują specjalny rodzaj warstwy zwanej warstwą konwolucyjną (Convolutional Layer), która wykonuje operację konwolucji na danych wejściowych.
- Wymagają mniejszej liczby parametrów (względem *DNN*).
- Rozbijamy większy problem (np. rozpoznawanie obrazów) na mniejsze prostsze problemy (np. wykrywanie krawędzi).

10.1 Konwolucja

- Struktura Hierarchiczna.
- Zamiast 1 wielkiej warstwy używamy wielu tych samych, małych liniowych warstw w każdej pozycji.
- koncentruje się na niskopoziomowych cechach w początkowych ukrytych warstwach, w kolejnej warstwie agreguje je do większej wysokopoziomowej cechy.
- Konwolucja to operacja matematyczna, która łączy dwa zestawy danych za pomocą funkcji matematycznej, aby wygenerować trzeci zestaw danych.
- W przypadku konwolucyjnych sieci neuronowych operacją konwolucji jest iloczyn skalarny (mnożenie element-wise) dwóch zestawów danych.
- Konwolucja jest operacją liniową, która może być używana do wielu celów, takich jak wykrywanie krawędzi i innych wzorców w obrazach, wykrywanie cech w danych itp.
- Polega na wykonywaniu sum ważonych dla fragmentów funkcji wejściowej ważonej przez jądro (kernel, który jest macierzą wag).
- W przypadku sieci neuronowych dane wejściowe są zwykle macierzą wielowymiarową (np. obrazem) i są one łączone z macierzą wag (kernel), aby wygenerować macierz wyjściową
- Wagi są parametrami, które są uczone podczas treningu modelu
- W przypadku obrazów macierz wejściowa zawiera piksele obrazu, a macierz wag zawiera filtry, które są aplikowane na obrazie
- Konwolucja może być obliczana na całym obrazie, ale zwykle stosuje się ją tylko do fragmentu obrazu, aby uzyskać macierz wyjściową o takich samych wymiarach jak macierz wejściowa
- W przypadku obrazów wagi są zwykle małymi macierzami o wymiarach 3x3 lub 5x5. W przypadku obrazów kolorowych, które mają 3 kanały kolorów (RGB), macierz wag ma wymiary 3x3x3 lub 5x5x3.
- Każda warstwa konwolucyjna składa się z wielu filtrów, które są stosowane do danych wejściowych, aby wygenerować różne macierze wyjściowe w celu wykrycia różnych cech w danych wejściowych

10.2 Typowe błędy podczas projektowania CNN

- Stosowanie za dużych jądr konwolucji (Wyjątek: Pierwsza warstwa konwolucyjna)
 - Zamiast tego nałożać więcej mniejszych warstw
 - * Prowadzi to do mniejszej liczby parametrów i mniejszej liczby obliczeń.

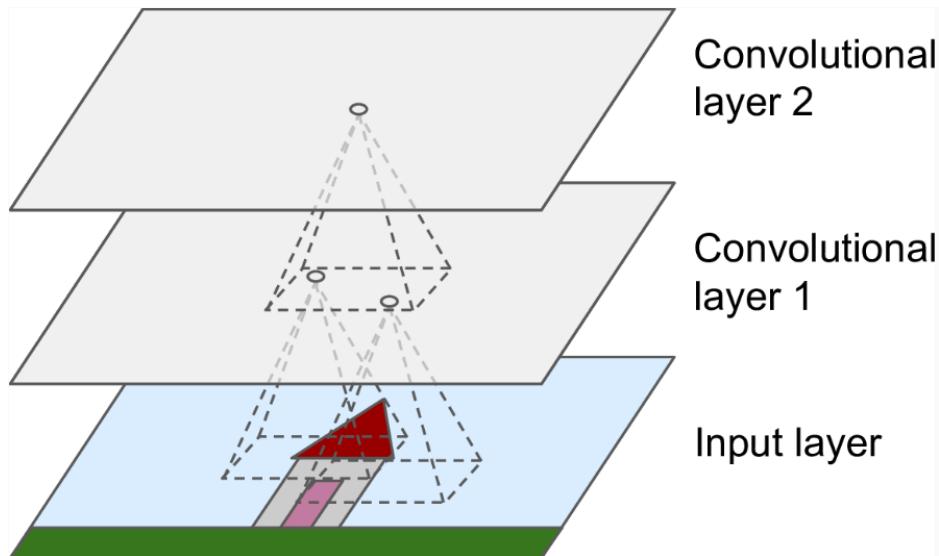


Figure 23: Przykład prostej sieci konwolucyjnej

10.3 Pooling

- Pooling neuron nie posiada wagi
 - Jej celem jest agregacja wejścia korzystając z funkcji *max* lub *mean*.
- Pooling jest operacją, która zmniejsza wymiary danych wejściowych poprzez zastąpienie fragmentu danych wejściowych pojedynczą wartością reprezentującą ten fragment zwracaną przez sprecyzowaną wcześniej funkcję
- Najczęściej stosowaną funkcją agregującą jest funkcja *max*, która zwraca maksymalną wartość w fragmencie danych wejściowych
- Pozwala kolejnym warstwom sieci na wykrywanie cech bardziej ogólnych, poprzez zwielokrotnienie obszaru, na którym bezpośrednio działają
- Często stosowany po warstwie konwolucyjnej, aby zmniejszyć wymiary danych wejściowych
- Najczęściej zmniejsza każdy wymiar danych wejściowych o połowę.

10.4 Dropout

- Sprawia, że wielka sieć działa jak mniejsza losowo trenując podsekcje sieci.
 - *Mniejsze sieci neuronowe nie mają skłonności do przeuczenia*
- Dropout jest techniką regularizacji, która losowo wyłącza neurony podczas uczenia
- Pomaga w zapobieganiu przeuczeniu modelu.

10.5 Uczenie rezydualne (Residual Learning)

- Residual Learning jest techniką uczenia głębokich sieci neuronowych, która skupia się na uczeniu różnic (residuum) pomiędzy wartością rzeczywistą a przewidywaną
- Residual Learning pomaga w zapobieganiu zanikaniu gradientu (vanishing gradient) i przyspiesza proces uczenia modelu

- Wykorzystujemy obejście (skip connection), aby dodać dane wejściowe do danych wyjściowych warstwy, aby uzyskać dane wyjściowe warstwy rezydualnej.
 - Sieć zaczyna robić progres nawet kiedy niektóre warstwy sieci nie zaczęły procesu uczenia.

10.6 Klasyfikacja i Lokalizacja obiektów

- Lokalizacja obiektów jest techniką uczenia głębokich sieci neuronowych, która służy do wykrywania obiektów w konkretnej lokalizacji na obrazie
- Można wykorzystać sieci w pełni konwolucyjne (Fully Convolutional Networks) do lokalizacji obiektów, wtedy każdy element wyjściowej macierzy reprezentuje prawdopodobieństwo wystąpienia obiektu w określonym obszarze obrazu
- Inną metodą jest wykorzystanie przesuwanej okna (sliding window), która polega na przesuwaniu okna po obrazie i sprawdzaniu, czy w oknie znajduje się obiekt, wymaga to wielokrotnego przetwarzania obrazu, co jest bardzo kosztowne obliczeniowo oraz różnych rozmiarów okna, aby wykryć obiekty o różnych rozmiarach

10.6.1 Bounding Boxes

- Sieci takie nazywamy *Region Proposal Network*.
- Gdy zaklasyfikujemy pewien obiekt i chcemy go zlokalizować na obrazie stosujemy *Bounding Boxes* – czyli określamy prostokątem fragment obrazu w którym najprawdopodobniej znajduje się obiekt.
- *non-max suppression*
 - Usuwamy nadmierną detekcję tego samego obiektu.

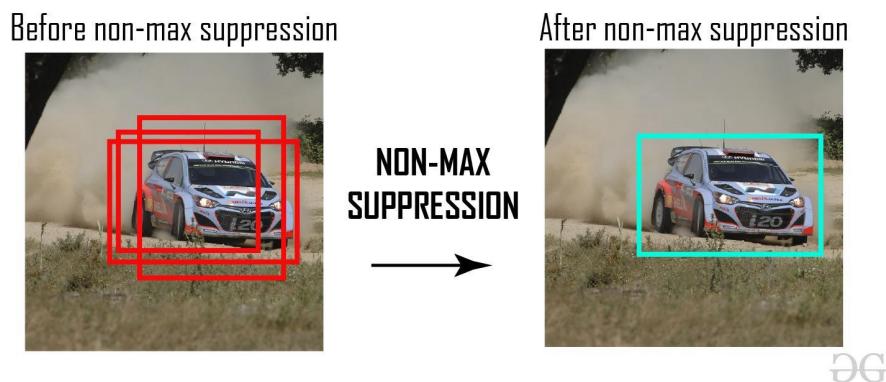


Figure 24: Wizualizacja wyniku działania non-max suppression

10.6.2 Fully Convolutional Networks

- Może być przećwiczona i użyta dla obrazów dowolnej wielkości

10.6.3 YOLO You Only Look Once

- Szybkie i dokładne
- Działa w czasie rzeczywistym

10.6.4 Transponowana warstwa konwolucyjna (*Transposed Convolutional Layer*)

- Może wykonywać interpolację liniową
- Warstwa którą możemy trenować
- Rozciąga zdjęcia przez dodawanie pustych wierszy i kolumn

10.6.5 Segmentacja semantyczna

- Segmentacja semantyczna jest problemem, który polega na przypisaniu każdemu pikselowi obrazu etykiety, która reprezentuje klasę, do której należy dany piksel
- Można w tym celu stosować architekturę U-Net, która składa się z warstw konwolucyjnych, warstw poolingowych i warstw dekonwolucyjnych tworzącą symetryczną strukturę w kształcie litery U.
- Różne obiekty tej samej klasy nie są rozróżnialne.

10.6.6 Metryki:

- *Mean Average Precision*
- *Intersection over Union*
 - Sprawdza jak dobrze model przewiduje *pola ograniczające* (bounding boxes).

11 Rekurencyjne sieci neuronowe

- Rekurencyjne sieci neuronowe (RNN - Recurrent Neural Networks)
- RNN są stosowane do przetwarzania sekwencyjnych danych, takich jak tekst, dźwięk, czasowe serie danych itp.
- Wykonują przewidywania dla sekwencji o dowolnej długości.
- Często wykorzystywane do predykcji na podstawie sekwencji danych wejściowych (o dowolnej długości), najczęściej do przewidywania przyszłości.
- Wykorzystują specjalny rodzaj warstwy zwanej warstwą rekurencyjną (Recurrent Layer), która przechowuje stan wewnętrzny, który jest aktualizowany za każdym razem, gdy warstwa otrzymuje dane wejściowe.
- Sieć wykonuje tą samą operację na każdym elemencie sekwencji, po czym agreguje informacje poprzednich wyrażeń w celu przewidzenia następnego.
- Zastosowania: finanse (giełda), pojazdy autonomiczne, sterowanie, wykrywanie usterek
- **Dużą wagą są znikające i eksplodujące gradienty**
 - gradient ≈ 0 lub zmieża do ∞ .
- Gdy sekwencja danych jest bardzo dłuża, sieć zapomina początkowe wartości

Podstawowym elementem RNN jest komórka rekurencyjna, która ma stan wewnętrzny przechowujący informacje z poprzednich **kroków czasowych (ramek)**. W każdym kroku czasowym komórka otrzymuje dane wejściowe oraz stan wewnętrzny (z poprzedniego kroku) i generuje nowy stan wewnętrzny oraz dane wyjściowe. Ten proces jest powtarzany dla każdego kroku czasowego.

Istnieje kilka różnych typów komórek RNN, takich jak **SimpleRNN**, **LSTM** (Long Short-Term Memory), **GRU** (Gated Recurrent Unit) i **Bidirectional RNN**, które różnią się w sposobie zarządzania i aktualizacji stanu wewnętrznego. Na przykład, LSTM wprowadza bramki, które kontrolują przepływ informacji, pozwalając na efektywne uczenie się zależności na różnych skalach czasowych i unikanie problemu zanikającego gradientu.

11.1 Rodzaje RNN ze względu na rodzaj danych wejściowych/wyjściowych

11.1.1 Sequence to sequence network

Stosowana, gdy input i output mają różne (nieokreślone) długości. Pobiera sekwencje danych wejściowych i generuje sekwencję przewidywanych danych.

Np. rozpoznawanie mowy, podsumowanie tekstu.

11.1.2 Vector to sequence network (Dekoder)

Podaje ten sam wektor danych wejściowych w każdym kroku czasowym i generuje sekwencję przewidywanych danych.

Np. Podpisywanie obrazów.

11.1.3 Sequence to vector network (Enkoder)

Podaj sekwencję danych wejściowych i zignoruj wygenerowaną sekwencję przewidywanych danych poza ostatnią wartością.

Np. Rozpoznawanie emocji w tekście, generowanie obrazów na podstawie opisu.

11.2 Działanie RNN w kilku krokach:

- Dane wejściowe sekwencyjne są podzielone na kroki czasowe.
- Na każdym kroku czasowym, dane wejściowe są przetwarzane przez komórkę rekurencyjną, która aktualizuje swój stan wewnętrzny.
- Dane wyjściowe są generowane na podstawie aktualnego stanu wewnętrznego.
- Proces jest powtarzany dla kolejnych kroków czasowych, przekazując informacje z poprzednich kroków.

11.3 Przewidywanie kilku kroków czasowych do przodu

Rozróżniamy 3 najpopularniejsze sposoby:

- Model przewiduje 1 krok czasowy na raz: Wyjście modelu prowadzimy do wejścia modelu. Jest to najgorsza opcja, błąd jest akumulowany za każdym cyklem.
- Model przewiduje n kroków na raz
- Model przewiduje wszystkie kroki na raz: Najlepsza opcja

11.4 Unrolling (rozwijanie)

Proces rozwinięcia lub dekompresji sieci rekurencyjnej na wielu krokach czasowych. W standardowej definicji RNN, model jest reprezentowany jako powtarzające się jednostki, które operują na danych wejściowych w każdym kroku czasowym. Jednak w celu lepszego zrozumienia i wizualizacji działania sieci, często stosuje się unrolling.

Podczas unrollingu, sieć rekurencyjna jest rozwinięta wzduż osi czasu, tworząc sekwencję powiązanych ze sobą jednostek. Każda jednostka reprezentuje stan wewnętrzny (np. LSTM lub GRU) oraz warstwę wyjściową, która otrzymuje dane wejściowe z danego kroku czasowego i generuje dane wyjściowe dla tego kroku. Te powiązane jednostki są połączone ze sobą, przechodząc informacje z jednego kroku czasowego do drugiego.

11.5 Osadzenia

Dokładnie reprezentują ciągi o zmiennej długości przez wektory o stałej długości.

11.6 Rozwiązywanie problemu niestabilnych gradientów

- Użyj tych samych rozwiązań co w przypadku *DNN*
- Nie stosuj nienasyconych funkcji aktywacji
 - np. ReLU

- *Batch Normalization* nie jest przydatne
 - Jak już musisz to stosuj pomiędzy warstwami rekurencyjnymi
- *Layer Normalization*

11.7 Wady RNN

- Wolno się uczy
- Długie sekwencje powodują zanikanie gradientu lub zapomnienie długoterminowych zależności. Pamięć zwykłej sieci rekurencyjnej nie jest tak dobra jeśli chodzi o zapamiętywanie zależności.
 - Przykład: Dla zdania “chmury są na _____” RNN spokojnie sobie poradzi łącząc zależność między niebem a chmurami. Dla zdania “Wychowałem się w Niemczech wraz z moim rodzeństwem. Spędziłem wiele lat tam i nauczyłem się bardzo wiele na temat ich kultury i obyczajów. Dlatego też mówię płynnie po _____” sieć będzie miała duży problem z przewidzeniem, ponieważ dystans między Niemcami a przewidywanym słowem jest o wiele większy.

11.8 Przygotowanie danych do RNN

- Musimy pamiętać, że kolejność danych ma znaczenie, dlatego dzieląc zbiór musimy zapewnić zachowanie kolejności danych.
 - `X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.3, shuffle=False, random_state=42)`
- Najbezpieczniej jest dzielić dane ze względu na czas (Jeśli operujemy na danych czasowych na przestrzeni np. 10 lat, to możemy ostatnie 2 lata zarezerwować na zbiór testowy)
 - Zakładamy, że dane są stacjonarne- zależność danych jest niezmienna

11.8.1 Dzielenie sekwencyjnego zestawu danych na wiele okien

- Konwersja długiej sekwencji danych na wiele krótszych sekwencji.

11.8.2 Sezonowość

- Sezonowością nazywamy regularną, okresową zmianę w śreniej badanej wartości.
- Sezonowość powiązana jest z czasem. Możemy zaobserwować sezonowość w przeciągu dnia, tygodnia, roku itd.
- Sezonowość jest napędzana cyklami świata przyrody (pory roku, cykl dnia/nocy) lub konwencjami zachowań społecznych dotyczących dat i godzin (Święta, czwartki studenckie).

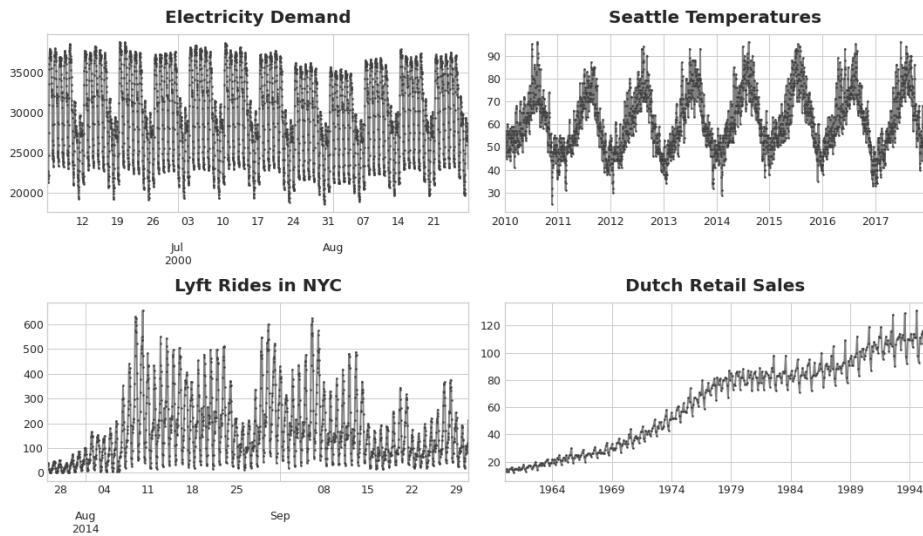


Figure 25: Przykłady zbiorów danych posiadających sezonowość

12 Sieci Enkoder-Dekoder

- Stosowana w problemach *Sequence to Sequence* (np. Przetłumaczenie zdania na inny język, Rozpoznanie mowy, Opis zdjęcia), gdzie długość wektora wejściowego jest różna od długości wektora wyjściowego.
- Sieć składająca się z 2 podsieci: **Enkodera i Dekodera**.
 - Zwykle obie te sieci posiadają taką samą architekturę, przy czym *Dekoder* działa odwrotnie względem Enkodera.
- W sieciach tych możemy stosować dowolnych rodzajów sieci (CNN, RNN, LSTM itp.)

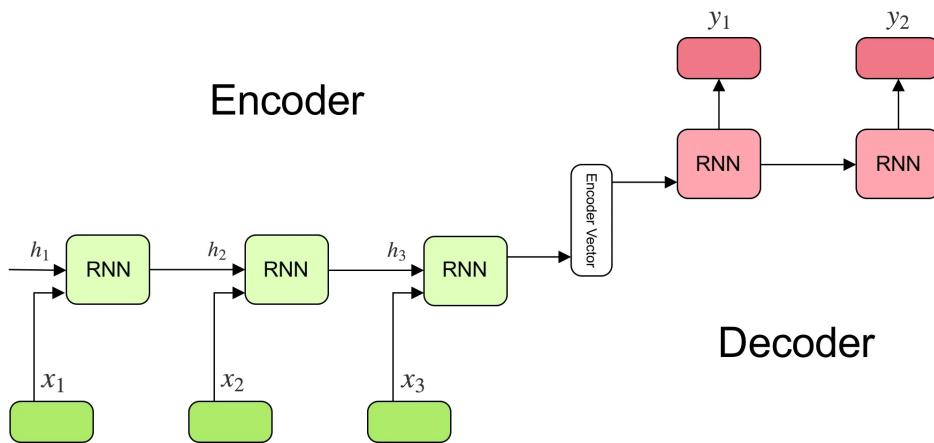


Figure 26: Architektura sieci Enkoder-Dekoder

12.1 Rola enkodera

- Na wejście pobiera sekwencję o zmiennej długości.
- W przypadku przetwarzania języka pobiera zdanie jako input i zwraca sekwencję liczb.
- Zbudowana z kilku warstw rekurencyjnych.

- Sieć rozumie kontekst i zależności między słowami w zdaniu.

12.2 Rola dekodera

- Działa jak model języka warunkowego.
- Jako wejście pobiera sekwencję zwracaną przez enkoder.
- Zwraca tokeny odpowiadające danym słowom w języku docelowym.

12.3 Dlaczego ją stosujemy do tłumaczenia języków?

- W przypadku zwykłej głębokiej sieci neuronowej tłumaczone będą słowa w naiwny sposób- sieć nie będzie dbała o kontekst, a tylko o poprawne przetłumaczenie słowa.
- *Enkoder-Dekoder* rozwiązuje ten problem poprzez użycie 2 sieci neuronowych.
 - *Enkoder* wyciąga znaczenie słowa wejściowego.
 - *Dekoder* przetwarza sekwencję zwróconą przez enkoder i tworzy własną wersję zdania.

12.4 Sieci Enkoder-Dekoder z różnymi rodzajami sieci

- Przy projektowaniu poniższych architektur pamiętajmy o limitacjach związanych z doborem odpowiednich sieci.
 - CNN potrzebują duży zbiór danych treningowych i są kosztowne obliczeniowo.
 - RNN problem zanikających/wybuchających gradientów.
 - Połączenie różnych sieci zwiększa złożoność modelu i czas trenowania.

12.4.1 CNN jako Enkoder, RNN/LSTM jako Dekoder

- Stosowany do generowania podpisów do obrazów.

12.4.2 RNN/LSTM jako Enkoder, RNN/LSTM jako Dekoder

- Tłumaczenie zdań na inny język.

13 Attention Mechanisms (Mechanizm Uwagi)

- Lepsza wydajność względem sieci *Enkoder-Dekoder*.
- Pozwala dekoderowi na wykrycie i wykorzystanie tylko najważniejszych danych i pominięcie tych mniej znaczących.
 - Sprawdza podobieństwo między wyjściem Enkodera a poprzednim ukrytym stanem dekodera.
- Jest odpowiedzią na problem bottleneck'u będący powodem używania wektorów o stałej wielkości jako wejście/wyjście.
 - Przez to dekoder ma ograniczony dostęp do danych wejściowych.
 - Szczególnie problematyczne dla bardzo długich sekwencji.
- Zbiór danych nie musi być sekwencyjny- może być dowolnego rodzaju.
- Mechanizm ten różni się od *LSTM* tym, że Mechanizm Uwagi zwraca uwagę na pewne specyficzne elementy lub obiekty zamiast traktować cały obraz w ten sam sposób.

13.1 Zasada działania

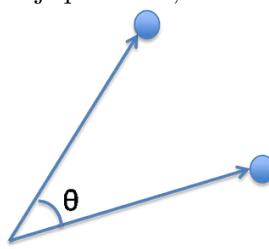
- Mechanizm uwagi składa się z 3 głównych komponentów
 - **Zapytań Q**
 - **Kluczy K**
 - **Wartości V**
- Każdy wektor zapytań (Query Vector) jest porównywany ze zbiorem kluczy aby obliczyć wartości podobieństwa.
 - Wykorzystuje do tego *Attention Mask*
 - * Opisuje jak bardzo podobny jest dany klucz do zapytania.
- Obliczone wartości są podawane do funkcji *softmax* by obliczyć wagi.
- Następnie **Ogólna uwaga (Generalized attention)** jest obliczona przez sumę ważoną wartości wektora, gdzie każda wartość jest sparowana z odpowiednim kluczem.
- W przypadku tłumaczenia zdań na inny język, każde słowo w zdaniu wejściowym ma przypisane własne zapytania, klucze oraz wartości.
- Dla lepszego wyjaśnienia tych komponentów wyobraźmy sobie wyszukiwarkę Google. Wpisując pewien tekst który później wyszukujemy możemy nazwać *Zapytaniem*. Wynikami wyszukiwania nazwiemy *Kluczami*, a treść tych wyszukiwań *Wartościami*. Dlatego szukając najlepszych dopasowań, musimy dla *Zapytania* znaleźć jak najbardziej podobny *Klucz*.
- Zasadniczo gdy podajemy mechanizmowi uwagi sekwencję słów, pobiera on wektor zapytań powiązany z pewnym słowem w sekwencji wejściowej i ocenia go w odniesieniu do każdego innego klucza w zdaniu. Poprzez wykonanie tego działania możemy się dowiedzieć jak bardzo rozpatrywane słowo jest powiązane z innymi w zdaniu. Następnie skalowane są te wartości aby model skupił się na słowach najbardziej istotnych dla danego zapytania. Wynikiem tego skalowania jest *Attention Output* rozpatrywanego słowa.

13.1.1 Miara podobieństwa

- **Cosine Similarity** (Podobieństwo Cosinusowe)

– Wartości pomiędzy -1 a 1, gdzie 1 to najbardziej podobne, a -1 totally niepodobne.

$$\text{sim}(A, B) = \cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|}$$



– Powyższy wzór możemy przepisać:

$$\text{Similarity}(A, B) = \frac{A \cdot B^T}{\text{scaling}}$$

– Ponieważ liczymy podobieństwo między *Query* a *Key* możemy powyższy wzór finalnie przepisać:

$$\text{Similarity}(Q, K) = \frac{Q \cdot K^T}{\text{scaling}}$$

13.2 Logika stojąca za mechanizmem uwagi

- Wyobraź sobie, że przychodzisz w czwartkowy wieczór na miasteczko i kolega wysyła ci zdjęcie gdzie jest. Patrzysz na to zdjęcie i co widzisz?
- Kiedy mówimy, że coś się widzi, mamy na myśli ciąg akcji, czyli poruszanie wzrokiem i zbieranie informacji na temat tego co jest w naszym polu widzenia. Nie widzisz wszystkich pikseli na raz, tylko zwracasz uwagę na poszczególne elementy obrazu jeden po drugim w celu zebrania pełnej informacji. Nawet w takim zatłoczonym obrazku możesz rozpoznać wujaszka Billiego czy pana starostę. Wynika to z tego, że zajmujesz się pewnymi istotnymi aspektami podanego obrazu zamiast sprawdzać piksel po pikselu.
- To jest właśnie ta mechanika, którą chcemy dać naszemu modelowi przez mechanizm uwagi.
- Możemy o tym pomyśleć jako pewna regularyzacja. Model nie będzie marnował czasu na bezmyślnym przeszukiwaniu obrazu, tylko skupi się na tym co jest ważne.

13.3 Self-Attention

- Zapytanie, Klucz i Wartość często pochodzą z innych źródeł zależnie od zadania oraz od rodzaju sieci (Enkoder czy Dekoder)
 - W przypadku tłumaczenia języka język źródłowy jest w Enkoderze, a język docelowy jest w Dekoderze.
- Nie posiada wiedzy na temat kolejności danych.
 - Wiedzę na ten temat możemy uzupełnić poprzez dodanie wartości do słowa lub poprzez osadzenie time step'ów (Time Step Embedding)
- Dla ciekawskich polecam poczytać o BERT.

13.4 Soft Attention i Hard Attention

- W “*Logika stojąca za mechanizmem uwagi*” mamy przedstawiony przykład *Hard Attention* z tego powodu, iż model nie używa wszystkich danych wejściowych do obliczenia *Uwagi*.
 - To które dane pomijamy w obliczaniu uwagi jest zadaniem sieci neuronowej.
- W przypadku *Soft Attention* wszystkie dane wejściowe są uwzględniane w obliczaniu *Uwagi*.
- *Soft Attention* jest bardziej popularną opcją przez większą efektywność procesu propagacji wstecznej.

14 Autoenkoder

- Zadaniem modelu jest odwzorowanie informacji wejściowej na wyjściu.
 - Jest w stanie nauczyć się bardzo skomplikowanych zależności pomiędzy danymi wejściowymi.
 - Model nie wymaga nadzwor, należy do *Self-Supervised Learning*.
 - Zbiór danych wejściowych jest nieetykietowany.
- Podobnie jak *Enkoder-Dekoder* stosowana do problemów *Sequence to Sequence*.
 - *Enkoder* służy do kompresji danych wejściowych do niskopoziomej reprezentacji.
 - *Dekoder* służy do rekonstrukcji oryginalnego obrazu przy podanej niskopoziomowej reprezentacji z Enkodera.
- Stosowany do kompresji obrazów.
- Architektura podobna do *MLP Multi Layer Perceptron*.
 - Tylko trzeba pamiętać, że liczba wejść jest równa liczbie wyjść.

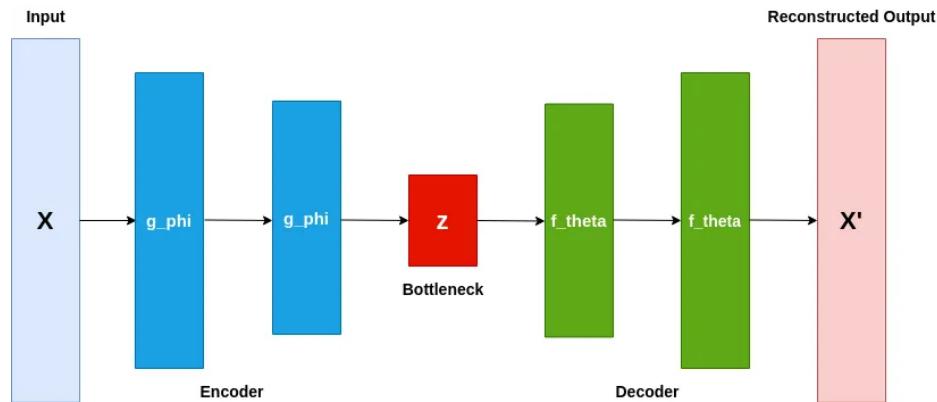


Figure 27: Architektura Autoenkodera

14.1 Stacked (Deep) Autoencoders

- Autoenkoder z wieloma ukrytymi warstwami
- Proces rekonstrukcji w Dekoderze traktowany jako problem klasyfikacji binarnej.
 - Model ma tendencję do szybszej zbieżności.

14.2 Konwolucyjny Autoenkoder

- Enkoder jako sieć *CNN* z warstwami *pooling*
- Dekoder działa jako **Sieć Dekonwolucyjna**
 - Sieć bliźniaczo podobna do sieci konwolucyjnej z tą różnicą, że warstwa konwolucyjna jest zamieniona na **Transponowane Konwolucyjne** warstwy (Transpose Convolutional Layers).

14.3 Rekurencyjny Autoenkoder

- Enkoder jako sieć *Sequence to Vector RNN*
 - Stosujemy *Sequence to Vector* aby skompresować sekwencję wejściową do pojedynczego wektora.

- Dekoder jako sieć *Vector to Sequence* RNN.

15 GAN (Generative Adversarial Network)

- Posiada w sobie 2 sieci
 - **Dyskryminator**
 - * Rozróżnia sztuczne dane stworzone przez generator z danymi należącymi do zbioru testowego.
 - **Generator**
 - * Próbuje odwzorować dane ze zbioru testowego.
- Model wykorzystujący **Adversarial Training** do trenowania.
 - Trening następuje przez rywalizujące ze sobą sieci neuronowe.
- *GAN* jest bardzo wrażliwy na hiperparametry
 - Warto spędzić dużo czasu na ich dostrojenie

15.1 Generator

- Tworzy dane, które mają oszukać Dyskryminator
- *Sieć Dekonwolucyjna*
- Po każdej warstwie *Conv2D* zaleca się stosować *BatchNormalization* by zapewnić stabilność.
 - Warstwa ta normalizuje wyjście poprzedniej warstwy przed podaniem go do następnej.
- Nie chcemy żeby *GAN* odwzorowywał dane 1:1, dlatego podajemy na wejście zaszumione dane (możemy zastosować w tym celu `tf.random.normal()`)
- Powszechną praktyką jest zmniejszanie ilości neuronów w kolejnych warstwach wraz z postępowaniem upsamplingu.

15.1.1 Implementacja

- W tym przykładzie będziemy pracować nad zbiorem **CIFAR100**. Jest to zbiór kolorowych obrazków 32x32x3, który możemy pobrać przez `keras.datasets.cifar100.load_data()`
- Jako pierwszą warstwę dajemy *Dense Layer*, która pobiera dane wejściowe. Trzeba pamiętać by ta warstwa miała wystarczającą liczbę neuronów do przechowania zredukowanej wersji obrazu.

```
keras.layers.Dense(4 * 4 * 128, input_shape=[noise_input],
                   activation=keras.layers.LeakyReLU(alpha=0.2)),
keras.layers.Reshape([4, 4, 128]),
```

- Aktualnie mamy obraz o wymiarach 4x4x128 a chcemy żeby miał takie same wymiary wejściowe, czyli 32x32x3. Dlatego też będziemy wykonywać *Dekonwolucję*, upsampling wykorzystując parametr *strides* w warstwie *Conv2DTranspose*

```
keras.layers.Conv2DTranspose(128, kernel_size=4, strides=2, padding="SAME",
                            activation=keras.layers.LeakyReLU(alpha=0.2)),
keras.layers.BatchNormalization(),
```

- Jako, że *strides* wynosi 2, wynikiem tej warstwy jest obraz 8x8x3 (Strides (2,2) zwraca takie same rozmiary obrazu co przy 2).

- Stosujemy *BatchNormalization* do znormalizowania wyników poprzedniej warstwy i do zwiększenia stabilności modelu.
- Jako finalną warstwę powinniśmy dać *Conv2D* z funkcją aktywacji *tanh*, ponieważ chcemy mieć wartości pomiędzy -1 a 1.

```
keras.layers.Conv2D(3, kernel_size=5, activation='tanh', padding='same')
```

15.2 Dyskryminator

- Sieć prostsza od Generatora.
- Redukuje dane korzystając warstw *Conv2D*, a na końcu *Flatten* by model określił czy dane są prawdziwe czy też sztuczne.
- Konieczność użycia *Dropout* o relatywnie dużej wartości (około 0.5, warto sprawdzić jaka wartość najlepiej działa).
 - Nie chcemy żeby Dyskryminator się przećwiczył. Dla niskiej wartości Dropout Dyskryminator działa za dobrze, nie dając możliwości Generatorowi się nauczyć.

15.3 Trenowanie GAN

- Na początku trenujemy Dyskryminator, dopiero potem Generator.
 - Nie chcemy, żeby dyskryminator się uczył na fałszywych danych
 - Możemy specjalnie nakazać sieci, by nie aktualizowała swoich wag poprzez ustawienie `discriminator.trainable = False`
- Nie możemy dopuścić do sytuacji kiedy Dyskryminator lub Generator jest skuteczniejszy od drugiego.
- **Mode collapse**
 - Dane wygenerowane przez generator są coraz mniej zróżnicowane
 - Z każdym cyklem parametry oscylują i model staje się niestabilny
 - Na przemian Dyskryminator rozpoznaje większość danych wygenerowanych i nie rozpoznaje żadnego.
- **Nash equilibrium**
 - GAN możemy przedstawić jako *zero-sum game* pomiędzy Generatorem i Dyskryminatorem
 - Sytuacja w której jakakolwiek zmiana w Dyskryminatorze i Generatorze nie polepszy wydajności modelu.
 - GAN może osiągnąć tylko jedno equilibrium Nash'a.

15.3.1 Implementacja

- Przykład GAN generujące obrazy 32x32x3.
- Na początek ćwiczmy *Dyskryminator*.
- Musimy dodać szum do danych wejściowych z powodu wyżej opisanego.

```
noise = tf.random.normal(shape=[batch_size, random_normal_dimensions])
```

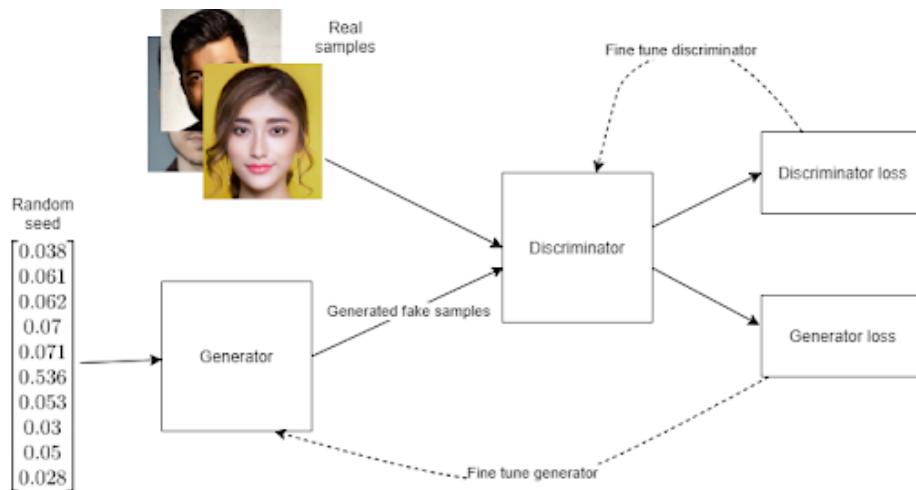


Figure 28: Trening GAN

Przekazujemy teraz szum do generatora, by stworzył sztuczne obrazki.

```
fake_images = generator(noise)
```

Tworzymy etykiety: 0 dla sztucznych, 1 dla prawdziwych zdjęć.

```
mixed_images = tf.concat([fake_images, real_images], axis=0)
```

```
discriminator_labels = tf.constant([[0.]] * batch_size + [[1.]] * batch_size)
```

Wagi dyskryminatora muszą być trenowalne.

```
discriminator.trainable = True
```

Ćwiczenie Dyskryminatora.

```
discriminator.train_on_batch(mixed_images, discriminator_labels)
```

- Ćwiczenie Generatora

Ponownie tworzymy szum.

```
noise = tf.random.normal(shape=[batch_size, random_normal_dimensions])
```

WSZYSTKIE obrazy oznaczamy jako prawdziwe.

```
generator_labels = tf.constant([[1.]] * batch_size)
```

Wyłączamy możliwość trenowania wag dyskryminatora.

```
discriminator.trainable = False
```

Ćwiczymy całą sieć GAN korzystając z szumu stworzonego.

```
gan.train_on_batch(noise, generator_labels)
```

- W początkowych epokach *Dyskryminator* będzie w stanie rozpoznać wszystkie sztuczne obrazy. W wyniku czego generator będzie modyfikować swoje wagi.

- Generator będzie się uczył zależnie od tego, jak dobrze będzie oszukiwać *Dyskryminatora*. Dlatego na początku ćwiczymy dyskryminatora, a dopiero potem generator.

15.4 Rodzaje GAN

15.4.1 Conditional GAN

- GAN z dodatkowym wejściem w generatorze określającym przynależność zdjęcia do odpowiedniej klasy.

15.4.2 StyleGAN

- Ulepszona wersja GAN pozwalająca na foto realistyczne zdjęcia twarzy oraz na kontrolę wyjściowego zdjęcia.

15.5 Zalecenia w tworzeniu modeli GAN

- Jako funkcję straty używaj *LeakyReLU*
 - Ale na ostatniej warstwie Generatora *tanh*
- Stosuj warstwy *BatchNormalization*
- *kernel_size* musi być podzielny przez *strides*
- Kiedy ładujemy zdjęcia musimy wartości między -1 a 1.
- Jako optymizator użyj *Adam*
- Korzystaj z *label softener*
 - Dyskryminator zamiast wskazywać przez wartość binarną czy np. obraz jest prawdziwy lub sztuczny, ustawiamy jako wartość w pewnym zakresie.
 - * Na przykład możemy ustawić, że fałszywe etykiety są przypisywane wartością między 0 a 0.3, a prawdziwe dane przypisywane są wartością mniej niż 0.8 a 1.2
 - Warto jest sprawdzić czy lepiej użyć *label softener* tylko dla prawdziwych etykiet czy też dla obu.

15.6 Techniki regularyzacyjne

- Mini-batch Discrimination
 - Wymusza na generatorze tworzenie bardziej zróżnicowanych danych
 - Miara jak bardzo podobne są dane na przestrzeni batch'a.
 - * Przekazuje tą miarę dyskryminatorowi.

16 Reinforcement Learning (Uczenie przez wzmacnianie)

- Uczenie modelu opiera się na dobraniu optymalnego zachowania w środowisku, aby uzyskać maksymalną nagrodę. Np. Przejście robota z punktu A do punktu B najszybszą i najmniej wymagającą trasą.
- Podstawą uczenia modelu jest mechanika akcji i nagrody/kary. Agent dokonuje pewnej interakcji z otoczeniem dostając informację zwrotną czy akcja miała jakiś wpływ na zamierzony efekt końcowy.
- Uczenie modelu jest dosyć ciężkie przez dużą niestabilność oraz przez dużą wrażliwość na hiperparametry.

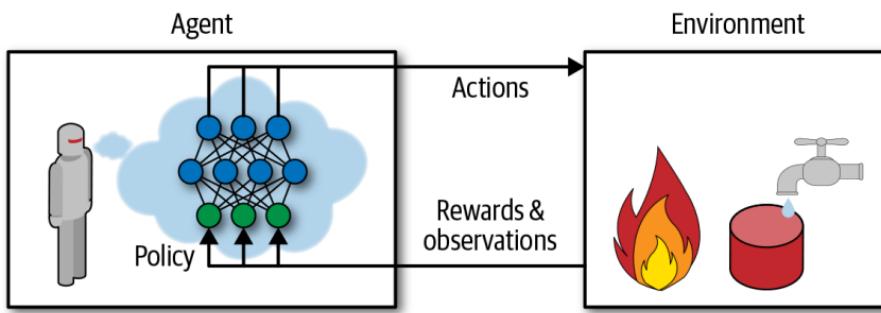


Figure 29: Uczenie przez wzmacnianie z użyciem sieci neuronowej do wyszukiwania polityki

16.1 Terminologia

- **Agent i Środowisko (Environment)**
 - Określenia bardzo ogólne mające wiele znaczeń zależnie od problemu w którym używamy *Reinforcement Learning*.
 - Kilka przykładów:
 - * Program sterujący robotem. W tym przypadku środowiskiem jest (Symulowany albo prawdziwy).
 - * Program grający w Pac-Man'a. Środowiskiem jest symulacja gry.
 - * Program grający w Go. Środowiskiem jest plansza.
 - * Agentem może być nawet termostat kontrolujący temperaturę środowiska.
 - Agent wpływa na środowisko swoimi akcjami.

16.2 Credit Assignment Problem

- Występuje gdy nagrody są bardzo rzadkie i opóźnione
- Model nie wie po których akcjach został nagrodzony
- Jak ten rozwiązać ten problem?
 - *Action Advantage*
 - * Jak wykonana akcja ma się średnio z innymi akcjami.
 - Ocena akcji na podstawie sumy wszystkich nagród, zastosowując ***discount factor*** γ na każdym kroku
 - * $\gamma \in [0, 1]$

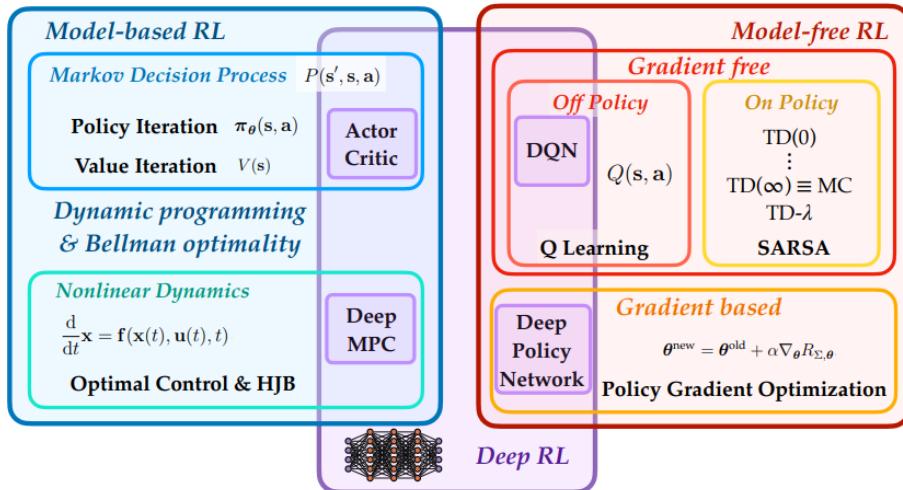
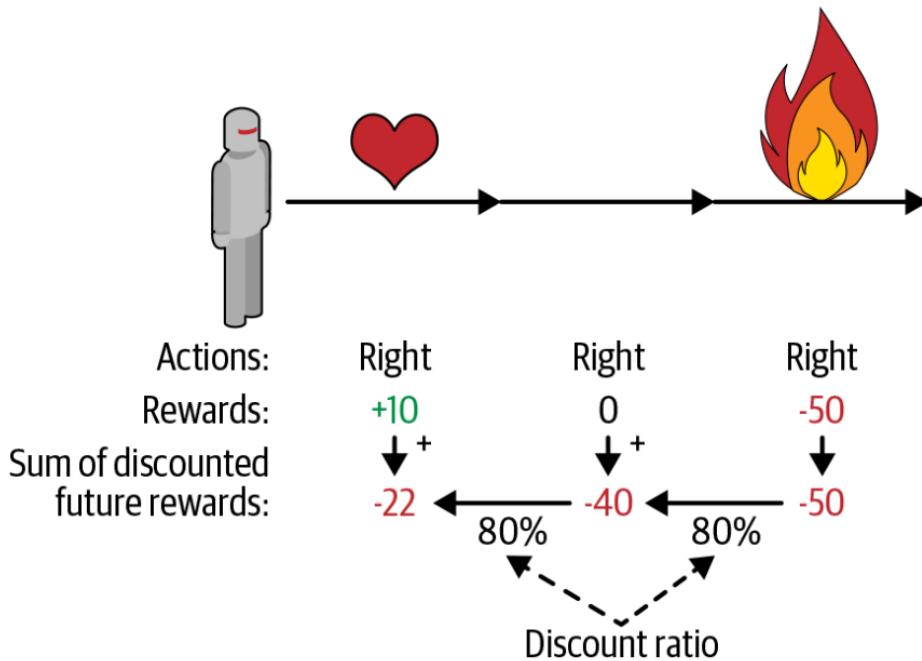


Figure 30: Podział Reinforcement Learning źródło: <https://doi.org/10.52843/cassyni.ss11hp>

- Przyspieszamy zbieżność niektórych algorytmów dając γ mniejszą od 1.
- * Opisuje jak bardzo agenta obchodzą nagrody w dalekiej przyszłości relatywnie do tych w bliskiej przyszłości.
 - Dla $\gamma = 0$ agenta będzie obchodziła tylko największa nagroda w tym momencie.
 - Dla $\gamma \approx 1$ agent ocenia każdą akcję bazując na sumie wszystkich przyszłych nagród.
- Dla przykładu weźmy poniższą sytuację. Jeżeli agent zdecyduje się na 3 ruchy w prawo, dostanie on nagrodę w następujących krokach 10, 0, -50. Natomiast jeśli użyjemy *discount factor $\gamma = 0.8$, pierwsza akcja zwróci nam wartość $10 + \gamma \times 0 + \gamma^2 \times (-50) = -22$.



- Dla lepszego zrozumienia natury *Credit Assignment Problem* będziemy rozważać przykład algorytmu uczącego się grania w grę Pong.

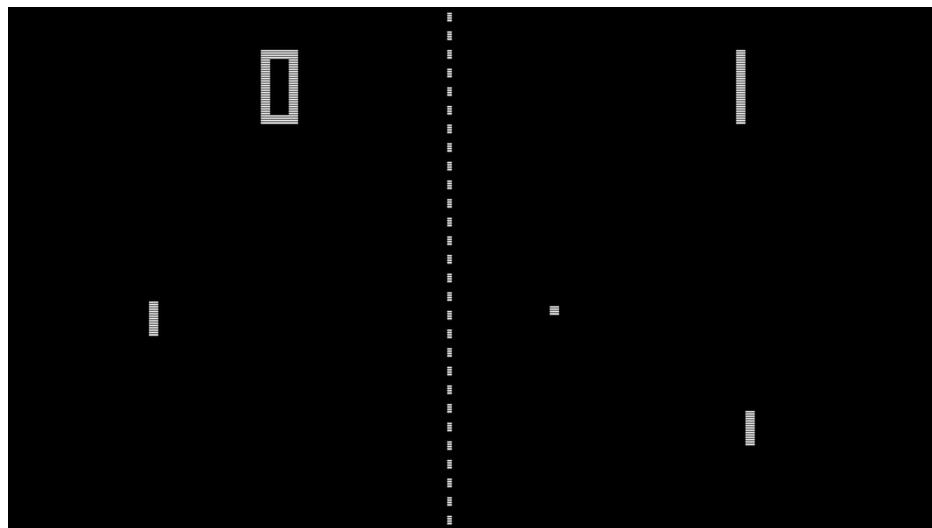
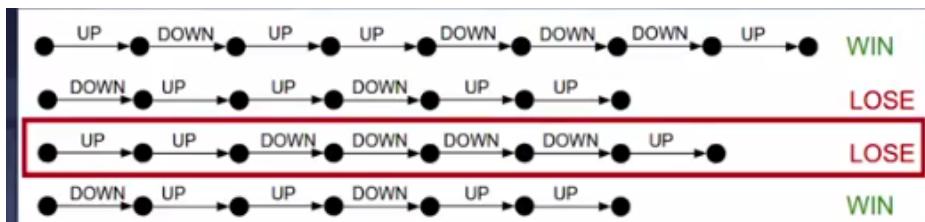
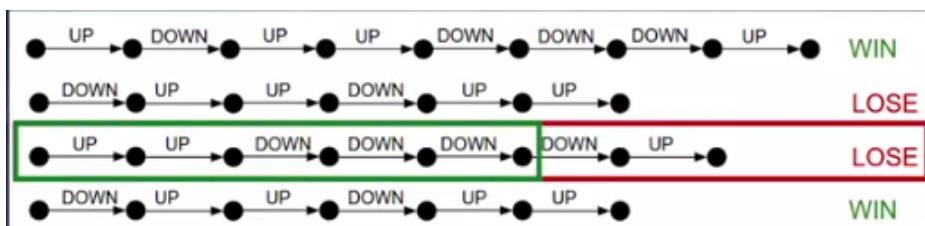


Figure 31: Pong

- Podczas ćwiczenia agenta wystąpią 2 różne scenariusze- wygrana lub przegrana. Kiedy wystąpi scenariusz, w którym agent przegrał, algorytm odrzuca lub obniży prawdopodobieństwo serii akcji, które wystąpiły w tym scenariuszu.



- Problem występuje w momencie gdy początkowe kroki były dobre, a tylko ostatnie 2 kroki spowodowały przegrana. Nie ma sensu odrzucać wszystkich akcji, lecz tylko te, które prowadziły do porażki.



- Właśnie tym problemem jest *Credit Assignment Problem*.

16.2.1 Maksymalizacja nagrody

- Agent kieruje się maksymalizacją nagrody, dlatego też model ten powinien zwrócić najbardziej optymalne akcje dające największą nagrodę.
- Skumulowaną nagrodę w każdym kroku czasowym w zależności od odpowiedniego kroku możemy

zapisać jako:

$$G_t = \sum_{k=0}^T R_{T+k+1}$$

- Wzór jednak okazuje się niekompletny. Jak spojrzymy na poniższy przykład to szybko zobaczymy, że nasz agent (mysz) maksymalizując nagrodę (ser) nie uwzględnia niebezpieczeństw jakim są koty lub porażenie prądem.



- W celu rozwiązania tego problemu dodajemy wyżej opisany *Discount Factor* γ

$$G_t = \sum_{k=0}^T \gamma^k R_{T+k+1}$$

16.3 Policy Search (Wyszukiwanie Polityki)

- Algorytm używany do określenia możliwych akcji.
- Może być siecią neuronową.
- Może być algorytmem stochastycznym.

16.3.1 Neural Network Policies czyli użycie sieci neuronowej do wyszukiwania polityki

- Zasada działania:
 - Wybierz losową informację bazującą na prawdopodobieństwu podanym przez sieć.
 - Daj Agentowi znaleźć balans między eksploracją a eksploatacją akcji.

16.3.2 Kompromis między Eksploracją a Eksplotacją

- Eksploracja, jak nazwa mówi, ma za zadanie znalezienie informacji na temat środowiska w którym się znajduje.
- Eksplotacja jest wykorzystywaniem już znanej wiedzy w celu maksymalizacji nagrody.
- Eksplotacja jest dobra na krótką metę, ale nie wiemy czy eksploracja nie da nam o wiele lepszego wyniku na dłuższą metę.



16.3.3 Explore Policy Space (Eksploracja przestrzeni polityki)

- Szuka najlepszych wartości dla danej polityki
- Używa technik optymalizacji *Policy Gradients*
- Stosuje **algorytmu genetycznego**.
 - Algorytm zainspirowany teorią ewolucji, a dokładniej selekcją naturalną.
 - Początkowo tworzymy 1 generację z losowymi cechami i wypuszczamy go do środowiska. Mierzymy ich skuteczność w osiągnięciu zamierzonego celu i usuwamy np. 80% najgorzej działających aktorów. Następnie pozwalamy reszcie “stworzyć potomstwo” by liczebność wszystkich aktorów w kolejnej generacji była taka sama. Potomstwo jest kopią rodzica z losowymi różnicami (ilość i rozmiar różnicy możemy modyfikować). Cykl się zapętla do momentu aż uznamy, że wynik jest satysfakcyjny.

16.4 Catastrophic Forgetting

- Nowo nauczona wiedza nadpisuje tą starszą.
 - Występuje gdy doświadczenia są wzorzeczależne.



Figure 32: Przykład algorytmu genetycznego uczącego się prowadzić pojazdem w labiryncie

16.5 Multi-Agent Reinforcement Learning

- Skupione na nauce zachowania wielu agentów, którzy koegzystują w jednym środowisku. Każdy agent korzysta w pewnym stopniu z algorytmów uczenia ze wzmacnianiem.

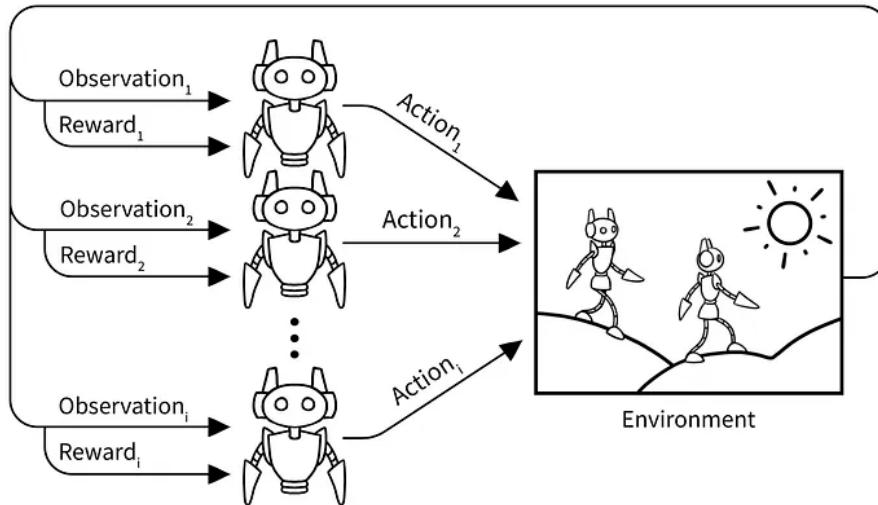


Figure 33: Copyright Justin Terry 2021

16.5.1 Rodzaje Multi-Agent system:

- Kooperacyjny**
 - Agenci o podobnych celach komunikują się między sobą i współpracują do wspólnego celu.
- Rywalizujący**
 - Agenci rywalizują między sobą.
 - Zadaniem agenta jest zmaksymalizowanie swojego wyniku, a co za tym idzie, zminimizować wynik innych agentów.

- **Mieszany**
 - Połączenie modelu *Kooperacyjnego z Rywalizującym*
 - Na przykład mecz koszykówki rozgrywany między 2 zespołami agentów.

16.6 Markov Decision Processes

- Graf skierowany o określonej ilości krawędzi i wierzchołków. Wierzchołek jest zmieniany z prawdopodobieństwem określonym przez wagę krawędzi.
- Stan s_0 ma prawdopodobieństwo przejścia do wierzchołka s_1 wynoszące 0.2, s_3 0.1 oraz s_0 0.7.

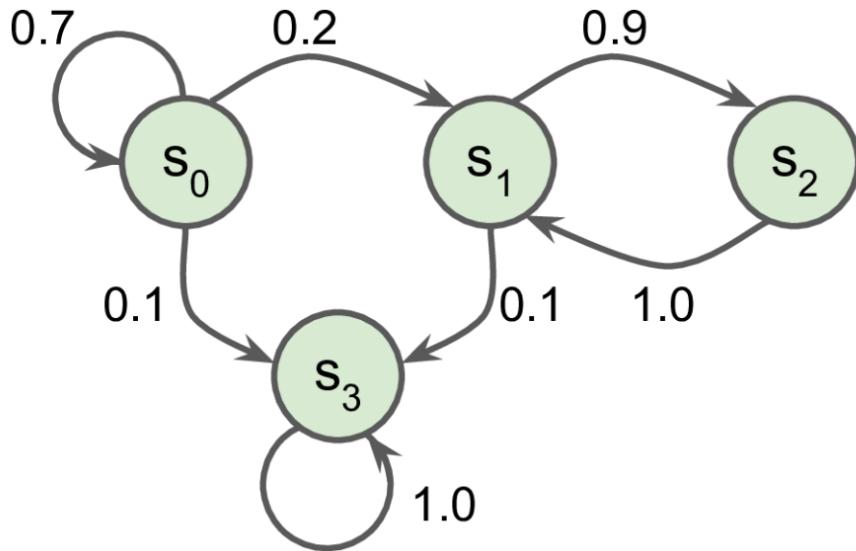


Figure 34: Łąćuch Markowa

- Za pomocą Łąćucha Markowa możemy przedstawić jako akcje które może agent wykonać w danym środowisku.
- Istnieje jednak pewna różnica, agent może wybrać jedną z kilku możliwych akcji i prawdopodobieństwo przejścia do kolejnego stanu zależy od akcji agenta. Dodatkowo niektóre krawędzie zwracają pewną wartość nagrody/kary.
- Zadaniem agenta jest znalezienie takiej polityki, która pozwoli zmaksymalizować wartość nagrody w czasie.
- Reprezentacją tą nazywamy **Proces Decyzyjny Markowa** (Markov Decision Process).

16.7 Q-Learning

- Polityka uczenia ze wzmacnianiem, która znajduje najlepszą następną akcję, przy podanym aktualnym stanie agenta.
- Jest polityką bez modelu
 - Uczy się metodą prób i błędów.
 - Nie korzysta z systemu nagród.
 - Agent dobiera akcje w zależności od własnych predykcji reakcji otoczenia na jego zachowanie.

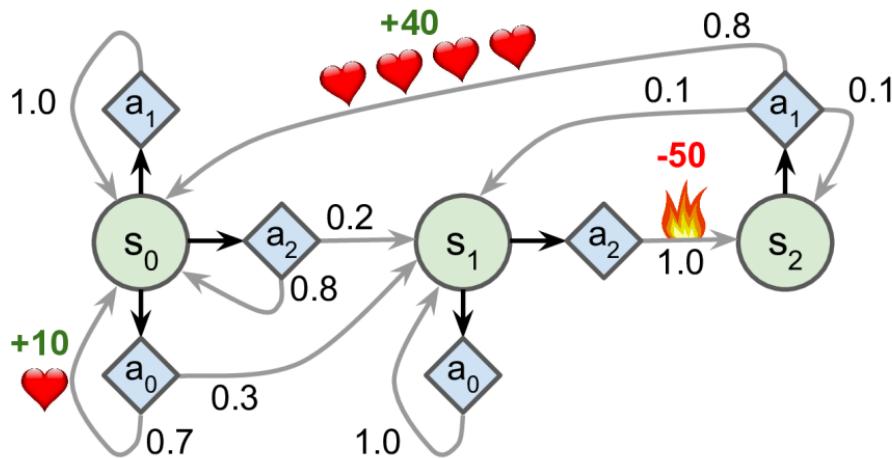


Figure 35: Przykład Procesu Decyzyjnego Markowa

- W dążeniu do tego celu, agent może wymyślić swoje własne zasady albo nie będzie się słuchać tych podanych. Dlatego mówimy, że funkcjonuje bez polityki.
- Przykładem mogą być reklamy na stronie internetowej.

16.7.1 Q-Value (Quality Value)

- Określa jak dobrze pewna akcja a jest dla pewnego stanu s
 - Zapisujemy ją jako funkcję $Q(s, a)$
 - Optymalną wartość Q oznaczamy $Q^*(s, a)$
- Q -Learning jest procesem uczenia się funkcji Q bazując tylko na doświadczeniu.

16.7.2 Q-Table

- Podczas działania algorytmu Q -Learning agent będzie znajdować się w sytuacji, kiedy będzie miał wiele akcji do wyboru. Stosujemy wtedy Q -Table do znalezienia najlepszej akcji.

16.7.3 Uczenie Monte Carlo

- Najprostsze podejście do uczenia przez doświadczenie.
- Następuje przez losowe próbkowanie przestrzeni akcja-stan.
- Wymagamy od RL by był **Epizodyczny**
 - Mamy określony start i koniec po pewnej skończonej liczbie akcji, co prowadzi do skumulowanej nagrody na końcu każdego *epizodu*.
 - Dobrym przykładem są gry.
- W przypadku tego uczenia, skumulowana nagroda pod koniec *epizodu* jest używana do określenia *Quality function Q* przez podzielenie finalnej nagrody równo przez wszystkie pary akcja-stan. Dlatego też jest to jeden z najprostszzych podejść, który rozwiązuje *Credit Assignment Problem*; Kredyt jest równo rozdzielany przez wszystkie przejściowe kroki. Również z tego powodu *Uczenie Monte Carlo* jest wyjątkowo narażone na niedostateczne próbkowanie (Szczególnie jest to widoczne, gdy nagrody

są rzadkie).

16.7.4 Temporal Difference (TD) Learning

- Kolejny sposób uczenia bazujący na próbkowaniu.
- Ma takie samo zadanie jak *Uczenie Monte Carlo*.
- W przeciwieństwie do Monte Carlo, nie jest ograniczony przez *epizodyczność*.
- Szacuje aktualny stan bazując na poprzednio nauczonych szacunkach stanów. Podejścia znane jako *Bootstraping*.

17 Porównania

17.1 Modele

Poniżej znajduje się porównanie modeli ze względu na sposób uogólnienia, rodzaj nadzoru, prostotę (interpretowalność), sposób użycia, zastosowanie, złożoność czasową i złożoność pamięciową.

Za model nieparametryczny (oparty o instancje) uznamy model, który nie ma ustalonej liczby parametrów, które muszą zostać wyznaczone w procesie uczenia. W przypadku modeli parametrycznych, liczba parametrów jest stała i niezależna od ilości danych treningowych.

Model	Nadzór	Sposób uogól-nienia	Prostota	Zastosowanie(uczenie)		Złożoność pamię-ciowa
					czasowa	
Regresja liniowa	nad.	param.	White box	Przewidywanie wartości ciągłych	O(nd)	O(d)
Regresja wielomianowa	nad.	param.	White box	Modelowanie zależności nieliniowych	O(nd)	O(d)
Regresja logistyczna	nad.	param.	White box	Klas. binarna	O(nd)	O(d)
SVM - Support Vector Machines	nad.	param.	Black box	Reg., klas. binarna i wieloklasowa	O(n^2d)	O(n^2)
Drzewa decyzyjne	nad.	inst.	White box	Klas., reg.	O(nd log n)	O(nd)
Las losowy (Random Forest)	nad.	param.	Black box	Klas., reg.	O(ndm log n)	O(ndk)
Gradient Boosting	nad.	param.	Black box	Klas., reg.	O(ndm log n)	O(ndk)
K-Nearest Neighbors	nad.	inst.	White box	Klas., reg.	O(nd)	O(nd)
DBSCAN	NIEnad.	inst.	White box	Grupowanie, wykrywanie anomalii	O(n^2) lub O(n log n)	O(n)
K-Means	NIEnad.	param.	White box	Grupowanie, wykrywanie anomalii	O(nkd)	O(nd)

Model	Nadzór	Sposób uogólnienia	Prostota	Zastosowanie(uczenie)	Złożoność czasowa	Złożoność pamięciowa
Głębokie sieci neuronowe	nad.	param.	Black box	Klas., reg., rozpoznawanie wzroczów, zaawansowana analiza dancyh	Zależy od architektury	Zależy od architektury
Konw. sieci neuronowe	nad.	param.	Black box	Przetwarzanie obrazów	Zależy od architektury	Zależy od architektury
Rekur. sieci neuronowe	nad.	param.	Black box	Przetwarzanie sekwencji, generowanie tekstu	Zależy od architektury	Zależy od architektury

Użyto oznaczeń:

- nad. - nadzorowane
- NIEnad. - nienadzorowane
- param. - parametryczny
- inst. - nieparametryczny (oparty o instancje)
- reg. - regresja
- klas. - klasyfikacja
- n - liczba próbek
- d - liczba cech
- m - liczba modeli
- k - liczba klastrów