
DeepDive™

Przygotowanie Danych

errors

inconsistent data

NaN

outliers

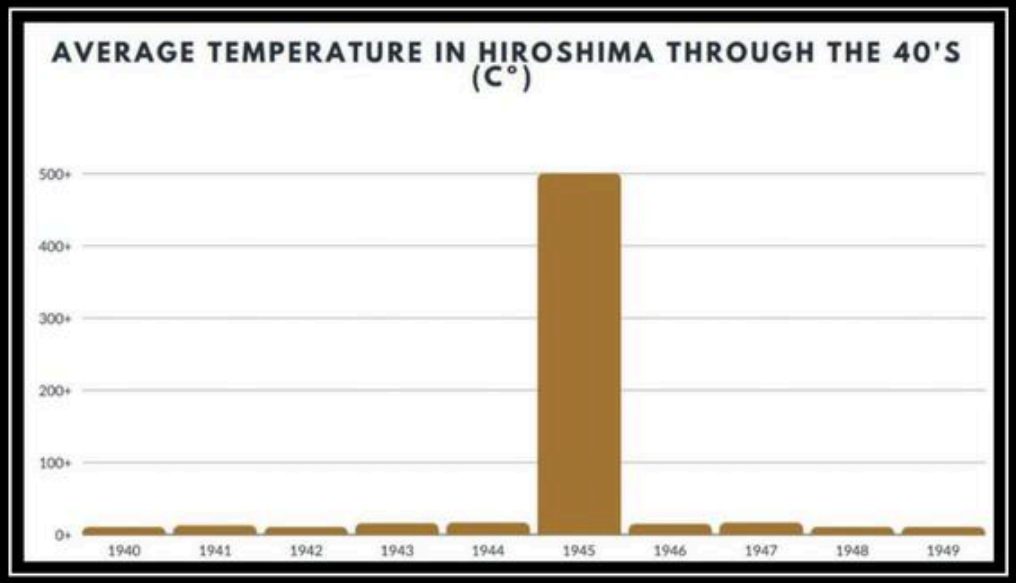
DeepDive™

Przygotowanie Danych

Kiedy pozyskujemy dane do wykorzystanie w tworzeniu modeli, mogą przybierać różne formy i wartości.

Aby mieć pewność, że nasze dane będą odpowiednie dla trenowania modeli, musimy je przeanalizować w celu wykluczenia odstających parametrów z zestawu lub też elementów bezsensownych.

Przygotowanie Danych



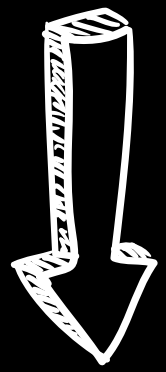
GLOBAL WARMING

https://knowyourmeme.com/photos/2104681-okbuddyretard

DeepDive™

Przygotowanie Danych

STANDARYZACJA



średnia = 0
odchylenie = 1

DeepDive™

Przygotowanie Danych

Dane, na których chcemy pracować mogą przybierać różną formę i znajdować się w niejednorodnym przedziale wartości. Dlatego też najlepiej jest sprowadzić je do jednorodnego standardu.

Proces ten nazywa się zwykle standaryzacją, czyli sprowadzeniem danych do rozkładu normalnego (o średniej 0 i odchyleniu standardowym 1). Pozwala on ujednolicić zbiór danych, przez co modele trenujące się na nim będą stabilniejsze a ich zdolności predykcyjne będą znacznie lepsze.

Standaryzacja Danych

Aby sprowadzić dane do rozkładu normalnego:

- Wczytujemy dane,
- Z całego przedziału danych odczytujemy średnią i odchylenie standardowe,
- Wykonujemy standaryzację: S = (dane - średnia) / odchylenie.

Aby przywrócić dane do stanu przed standaryzacją:

- Wczytujemy parametry użyte w standaryzacji
- dane = S * odchylenie + średnia

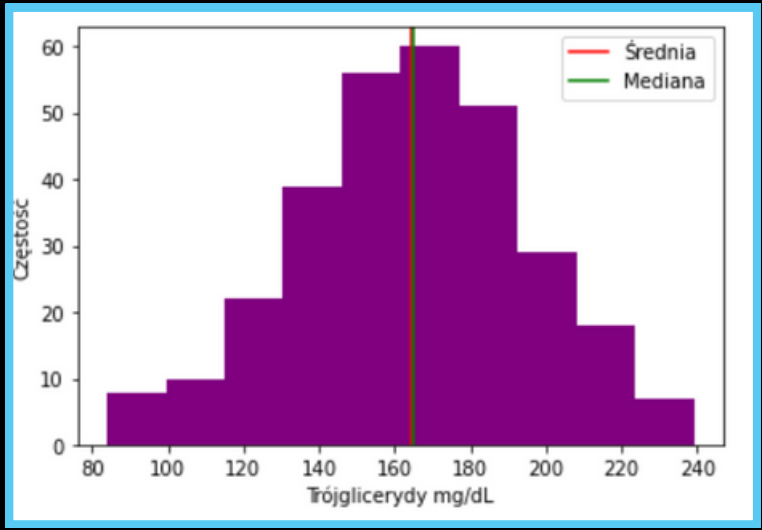
Standaryzacja Danych

```python
import numpy as np
import numpy.typing as npt

# dane = []
# for _ in range(10):
#     dane.append(np.random.randint(-4,10))

# Generujemy losowe liczby całe z zakresu [-4; 10)
dane: list[int] = [np.random.randint(-4, high: 10) for _ in range(10)]  # ← Komprehensja list. Analog w komentarzu wyże
dane_array: npt.NDArray = np.array(dane)  # Konwersja na np.NDArray


def standaryzacja(dane: npt.NDArray) → tuple[npt.NDArray, tuple[np.float64, np.float64]]:  1 usage
    srednia: np.float64 = dane.mean()
    odchylenie: np.float64 = dane.std()
    return (dane - srednia) / odchylenie, (srednia, odchylenie)


def przywrocenie_standaryzacji(standaryzowane: npt.NDArray, srednia: np.float64, odchylenie: np.float64) → npt.NDArray:
    return standaryzowane * odchylenie + srednia

print(f"Dane wejściowe: {dane_array}")
standaryzowane, parametry = standaryzacja(dane_array)
print(f"Dane standaryzowane: {standaryzowane}\nŚrednia: {parametry[0]}\nSTD: {parametry[1]}")
przywrocone_dane = przywrocenie_standaryzacji(standaryzowane, *parametry)
print(f"Przywrócone dane: {przywrocone_dane}")
```

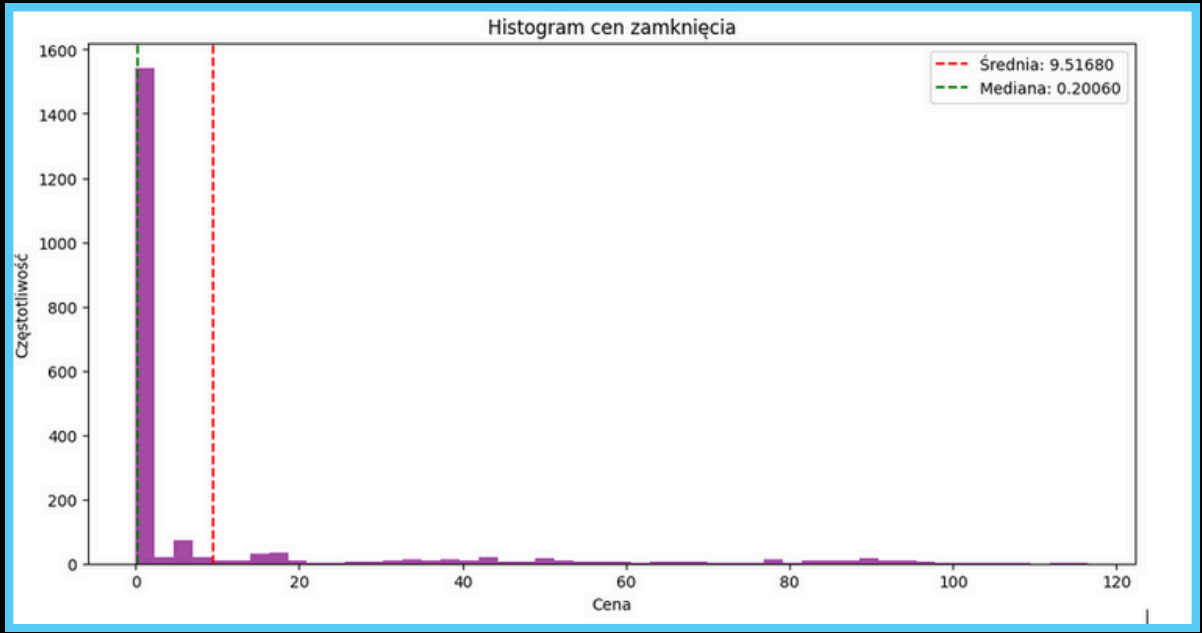
Przykład standaryzacji, Jakub Susoł

DeepDive™

Rozkład



Czy jest normalny?



DeepDive™

Co zrobić jeśli dane mają inny rozkład niż normalny?

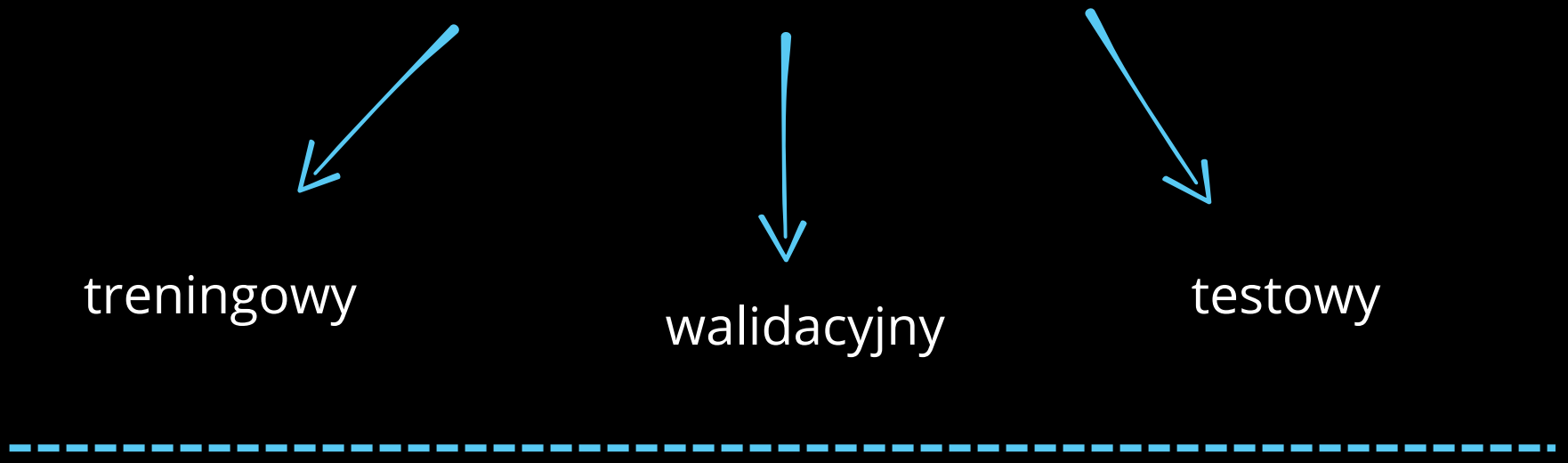
Jeśli zwizualizujemy dane z naszego zbioru w postaci histogramu, i okaże się, że któraś ze zmiennych nie ma cech rozkładu normalnego, tylko jej histogram jest prawo- lub lewoskośny, musimy przeprowadzić odpowiednią transformację danych.

Takich metod jest zbyt wiele żeby je omówić w przeciągu tego kursu. O wgląd na nie, wytłumaczenie ich działania i zastosowania zapytaliśmy ChatGPT, do którego odpowiedzi odsyłamy poniżej.

https://chatgpt.com/share/67461cb1-c284-8008-9b66-311fdd14f940

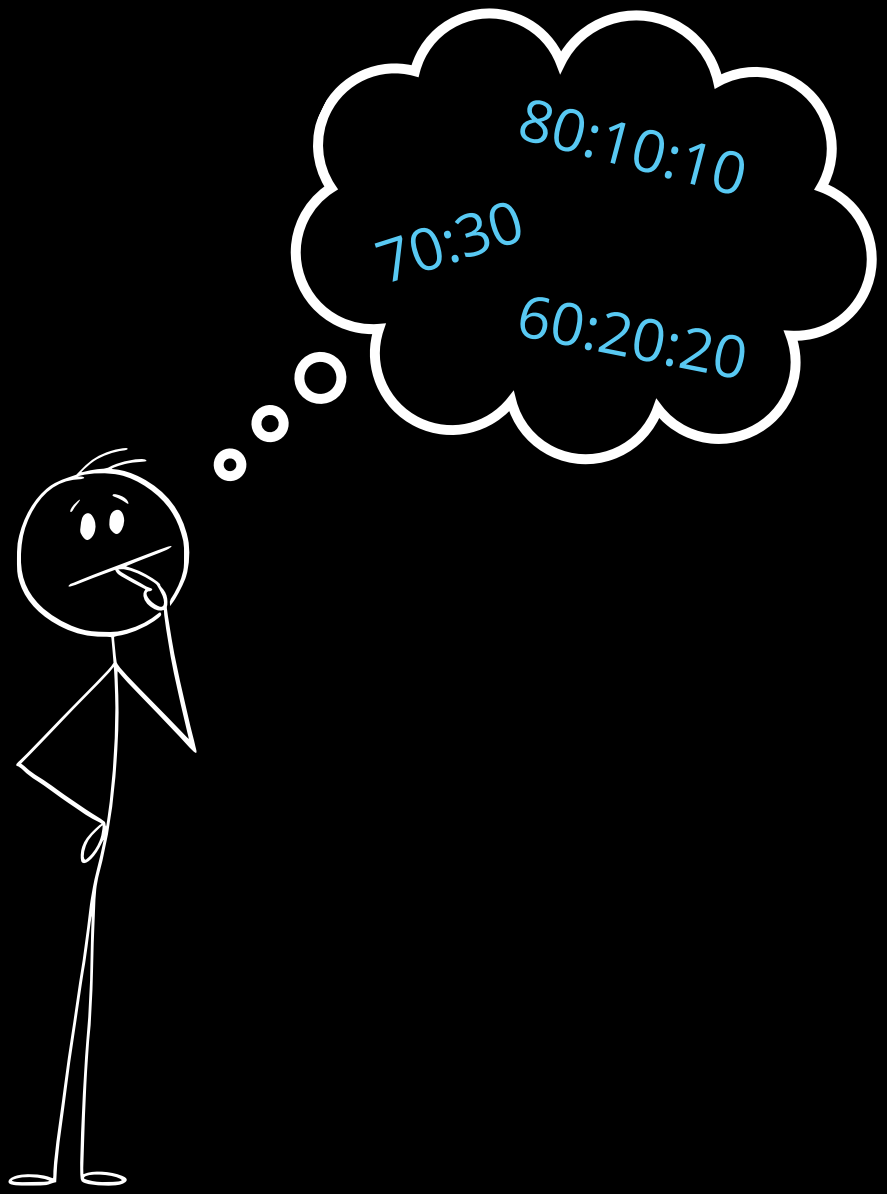
DeepDive™

Podział zbiorów



DeepDive™

Podział zbiorów



DeepDive™

Podział zbiorów

Zbiór treningowy jest zwykle znacznie większy (standardowo 70% całości zbioru danych), aby dać możliwość modelowi na poprawne nauczenie się schematów występujących w zestawie.

Pozostałe dane (standardowo 30%) znajdują się w zbiorze testowym (oraz walidacyjnym), na którym sprawdza się poprawność predykcji modelu.

Konkretne proporcje zależą od wielkości zestawu danych, ilości zmiennych etc.

Podział zbiorów

Aby mieć pewność że nasze modele dobrze nauczą się schematów z danych, zbiór musi być podzielony na zbiór treningowy i testowy. Podział taki można wykonać ręcznie, jeśli pracujemy na adekwatnie małych zbiorach, które to umożliwiają.

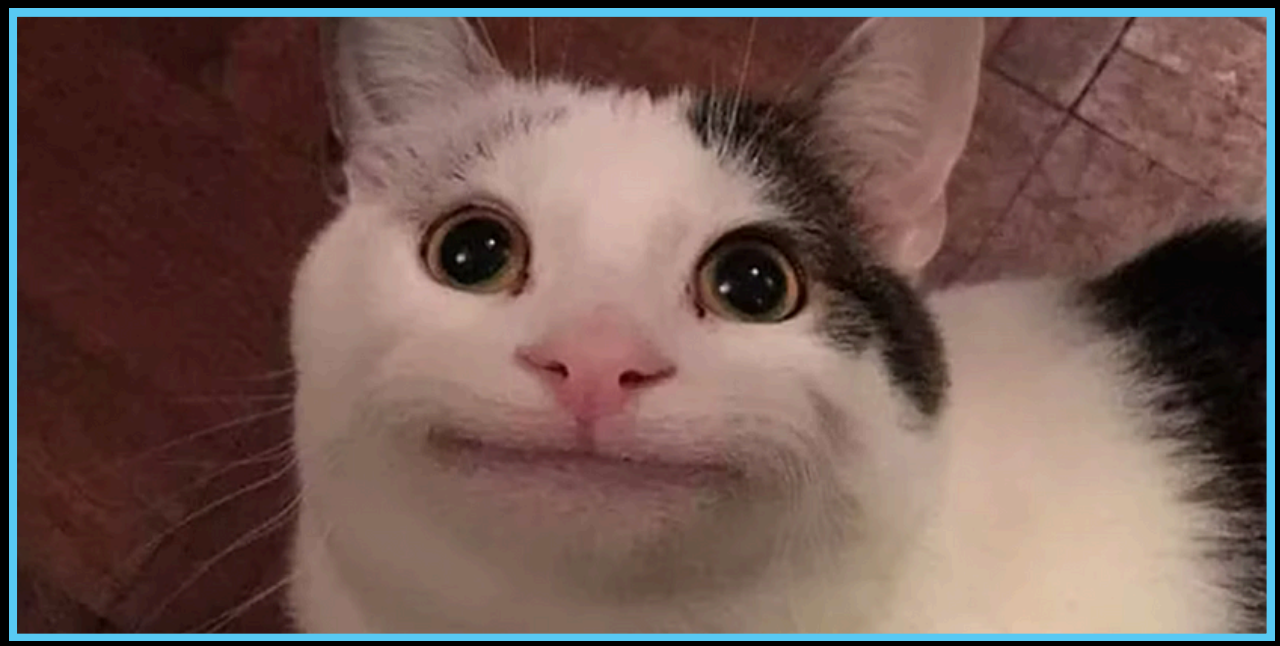
Alternatywnie, istnieją moduły, np. scikit-learn (sklearn), które umożliwiają automatyczny, losowy podział zbioru danych na treningowy i testowy w zadanej proporcji

Podział zbiorów

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)



from torch.utils.data import random_split

train_dataset, val_dataset, test_dataset = random_split(dataset, [train_size, val_size, test_size])
```

DeepDive™

Jak wielki powinien być zbiór danych?



Krótka odpowiedź brzmi: Tak.

DeepDive™

Jak wielki powinien być zbiór danych?

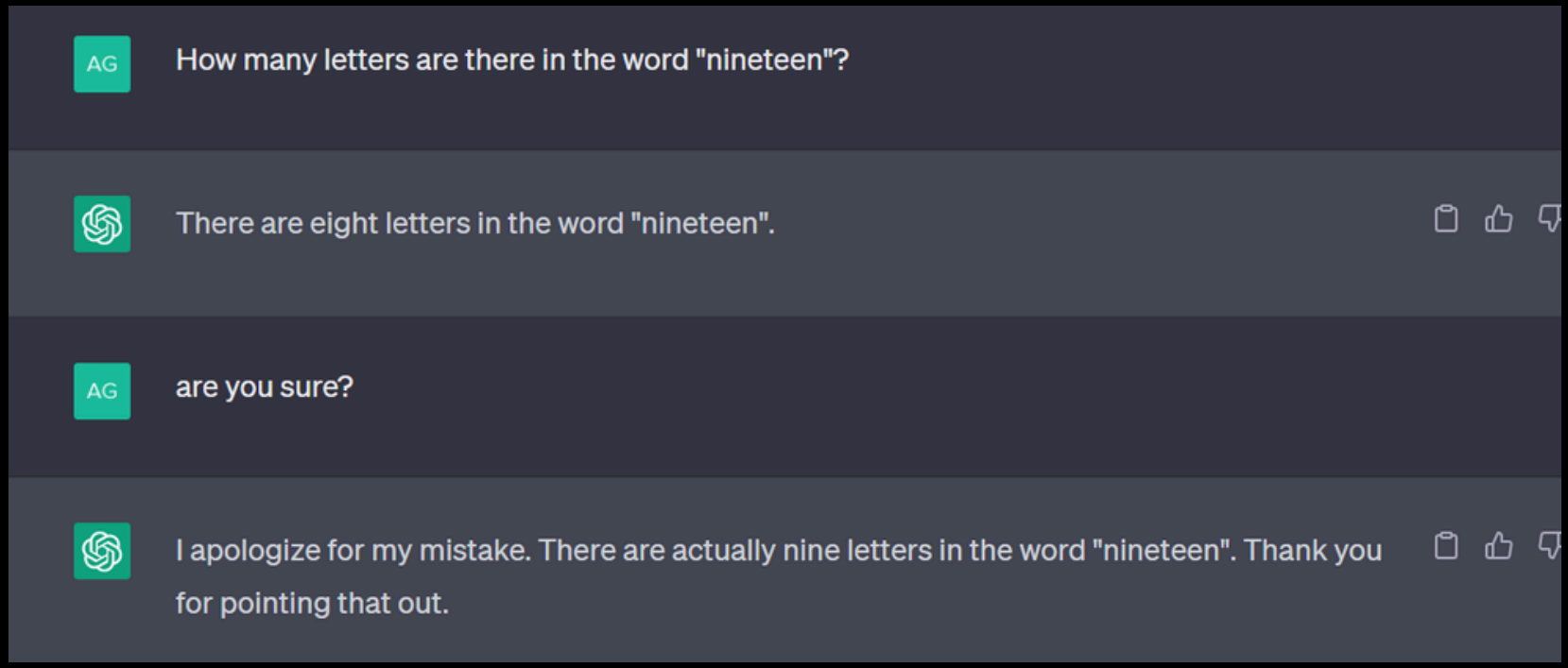
Krótka odpowiedź brzmi: Tak.

Wielkość zbioru danych powinna być tak duża, jak tylko jest możliwe dla danego zastosowania. Im więcej danych model ma do dyspozycji w celu trenowania się, tym lepiej. Należy jednak zwrócić uwagę na to, aby dane były w jakiś sposób rozbieżne oraz zawierały odpowiednio dużo kategorii, aby uniknąć zjawiska “overfittingu” modelu.

Jak wielki powinien być zbiór danych?

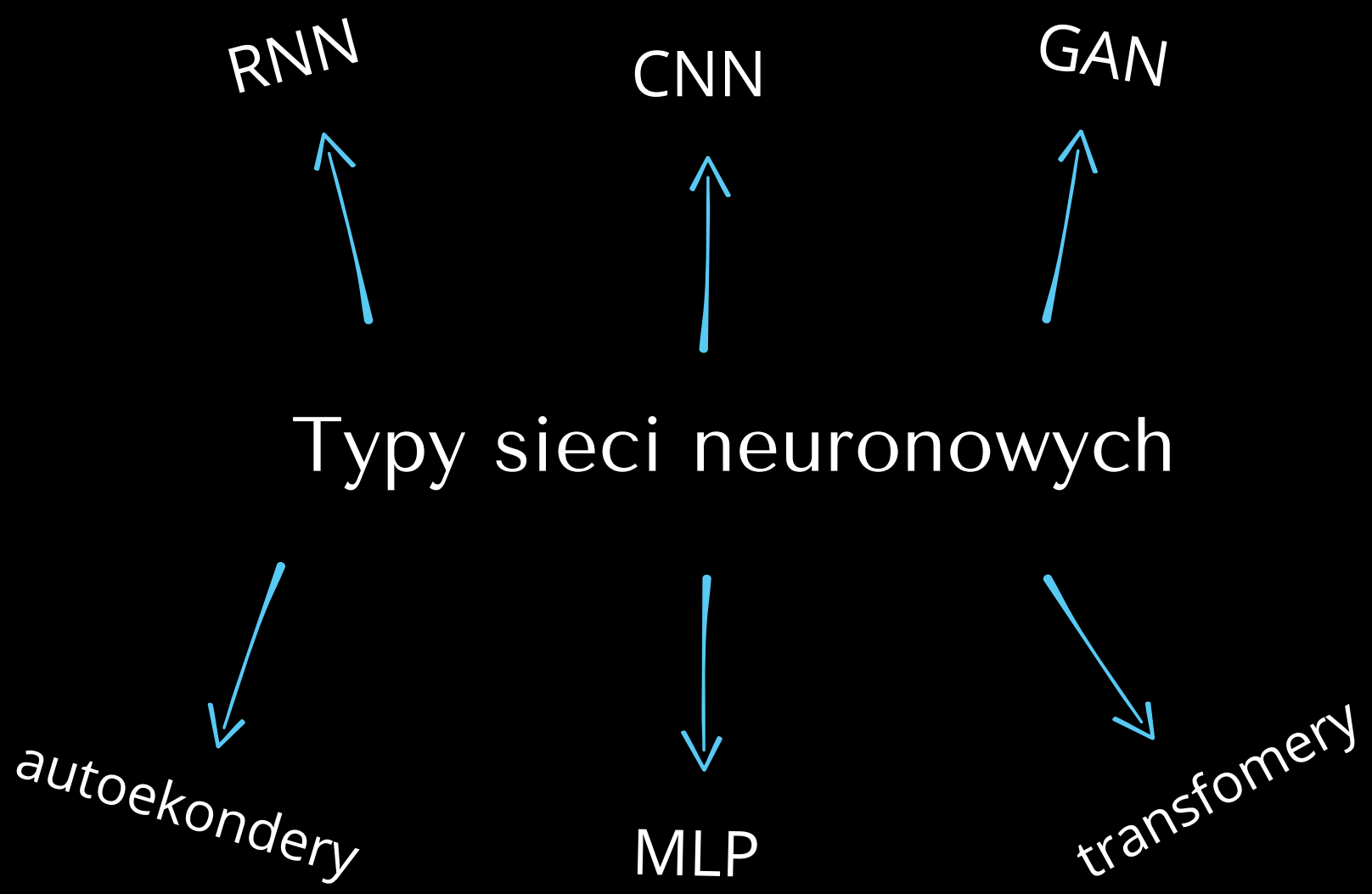
Overfitting jest zjawiskiem, kiedy model nauczony jest na zbyt wąskim przedziale parametrów. Może bardzo poprawnie i dokładnie przewidywać wyniki dal nauczonych parametrów, ale jeśli zada się mu dane spoza zakresu, na którym się uczył, jego możliwości predykcyjne spadają drastycznie, uniemożliwiając poprawne wywnioskowanie sensu z zadanych danych.

Overfitting w akcji, a.k.a halucynacje GPT



https://www.decipherzone.com/blog-detail/chat-gpt-memes

DeepDive™


RNN
CNN
GAN
Typy sieci neuronowych
autoekondery
MLP
transfomery

DeepDive™

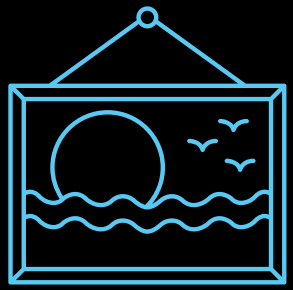
Pre-trenowane modele

Pre-trenowane modele to nic innego jak zrzut gotowego do użycia modelu. Ładując takie modele możemy bezpośrednio zacząć ładować im dane do przetworzenia, przeanalizowania czy innej operacji na nich w zależności od typu modelu.

Czasem łatwiej i lepiej jest skorzystać z takich modeli zamiast pisać swoje własne ze względu na zaoszczędzenie czasu (i w niektórych wypadkach nerwów). Ponadto niektóre modele mogą mieć ogromne zestawy danych na których się uczyły i być bardzo precyzyjne w zadaniach, które mają wykonywać.

Pre-trenowane modele

Analiza obrazów



ResNet

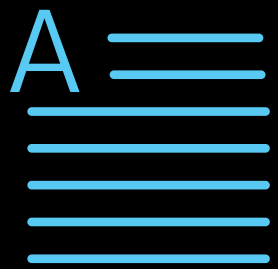
MobileNetV2

Detekcja obiektów

YOLO

Mask R-CNN

Przetwarzanie tekstu



BERT

GPT

DeepDive™

Pre-trenowane modele

TorchVision

https://pytorch.org/vision/stable/models.html

Jak zapisać model?

Stworzone i przetrenowane modele można zapisać w celu załadowania ich w późniejszym czasie w celu użycia w skryptach innych niż natywny skrypt modelu lub w celu dotrenowania ich w późniejszym czasie, jeśli zaopatrzymy się w więcej danych.

Są na to dwa różne sposoby. Modele możemy zapisać natywnie za pomocą PyTorch lub wyeksportować cały konstrukt modelu i danych za pomocą modułu Joblib.

Jak zapisać model?

```python
from joblib import dump
```

```python
dump(scaler, 'scaler.joblib')
dump(svd, 'svd.joblib')
dump(vectorizer, 'vectorizer.joblib')
dump(classifier, 'sign_language_classifier.joblib')
```

torch.save()

```python
from joblib import load
```

```python
model = load('sign_language_classifier.joblib')
vectorizer = load('vectorizer.joblib')
scaler = load('scaler.joblib')
svd = load('svd.joblib')
```

torch.load()

DeepDive™

Joblib vs PyTorch

Joblib

- Potrafi zapisać wszystkie obiekty używane w skryptach, łącznie z modelem i różnymi danymi,
- Prosty w użytku,
- Nie obsługuje wewnętrznych struktur modeli (np. state-dict),
- Możliwe problemy z kompatybilnością pomiędzy środowiskami Pythona

PyTorch

- Zapisuje model i jego stan,
- Potrafi zapisać konkretne części modelu (wagi, biasy, itp.),
- Może zapisać TYLKO model PyTorcha, pomija wszystkie inne dane ze skryptu,
- Zapisane modele mogą być kompatybilne tylko z konkretnymi wersjami PyTorch

DeepDive™

Struktura sieci neuronowej

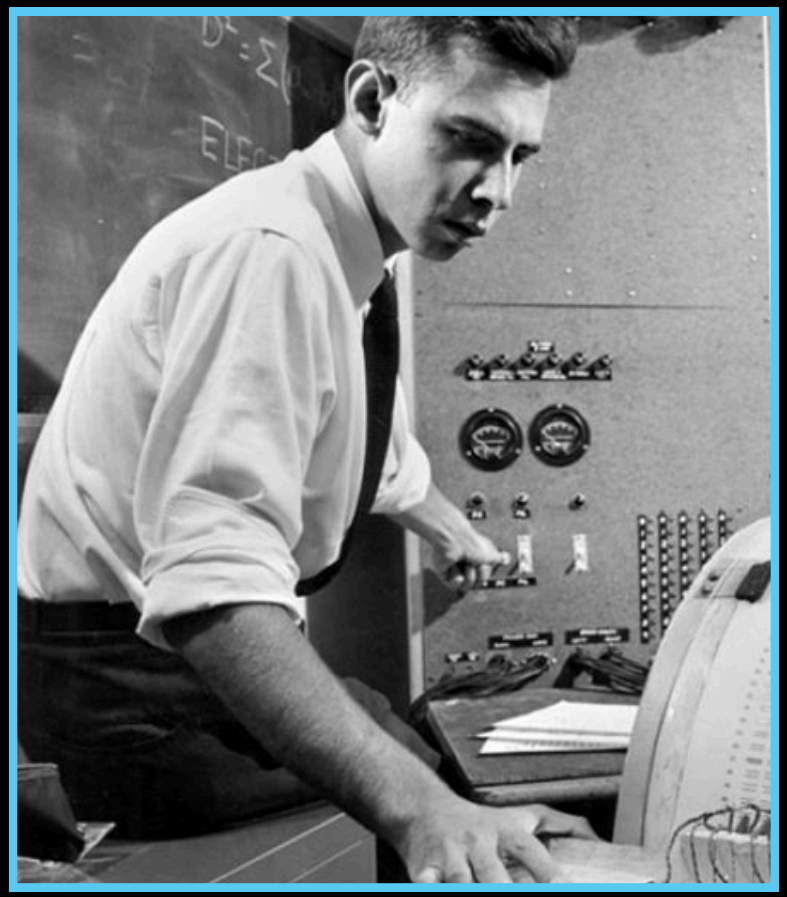
Najprostsza sieć neuronowa jest siecią typu **“Feed Forward”.** Sieci te cechują się linearnym, nierekurencyjnym (bez pętli) przepływem danych.

Każda sieć musi mieć warstwę gdzie wprowadzamy dane. Ta warstwa łączy się z warstwami ukrytymi które przetwarzają dane wejściowe za pomocą określonych funkcji, i ostatecznie łączą się z warstwą wyjściową.

Zależnie od analizowanej ilości parametrów i spodziewanych danych wyjściowych, wymiary tych warstw mogą mieć różny rozmiar.

DeepDive™

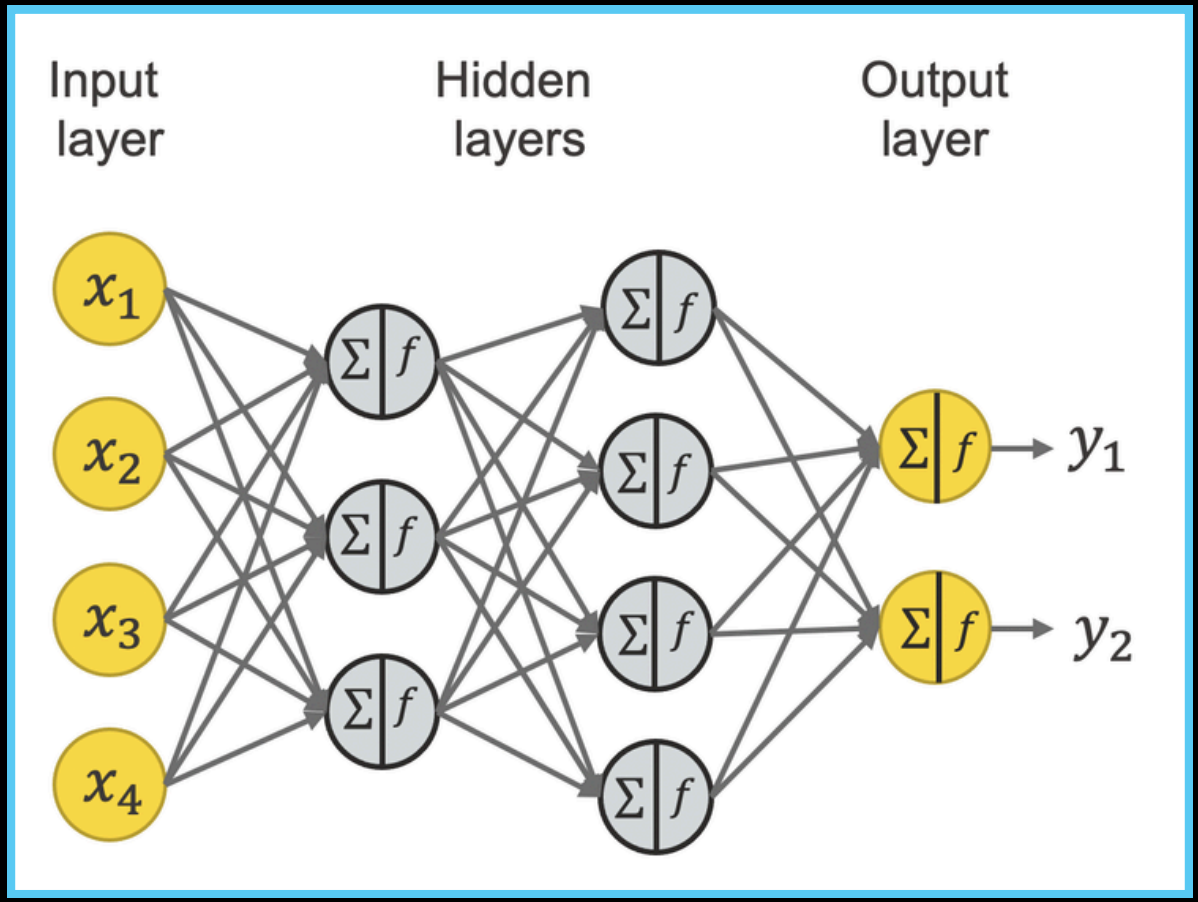
Perceptron



Frank Rosenblatt

DeepDive™

Struktura sieci neuronowej



https://www.knime.com/blog/a-friendly-introduction-to-deep-neural-networks

DeepDive™

Funkcje aktywacji

skokowa

$$f(z) = \begin{cases} 1 & \text{jeśli } z \geq 0 \\ 0 & \text{jeśli } z < 0 \end{cases}$$

sigmoidalna

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

ReLU (Rectified Linear Unit)

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

tangens hiperboliczny

$$f(z) = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

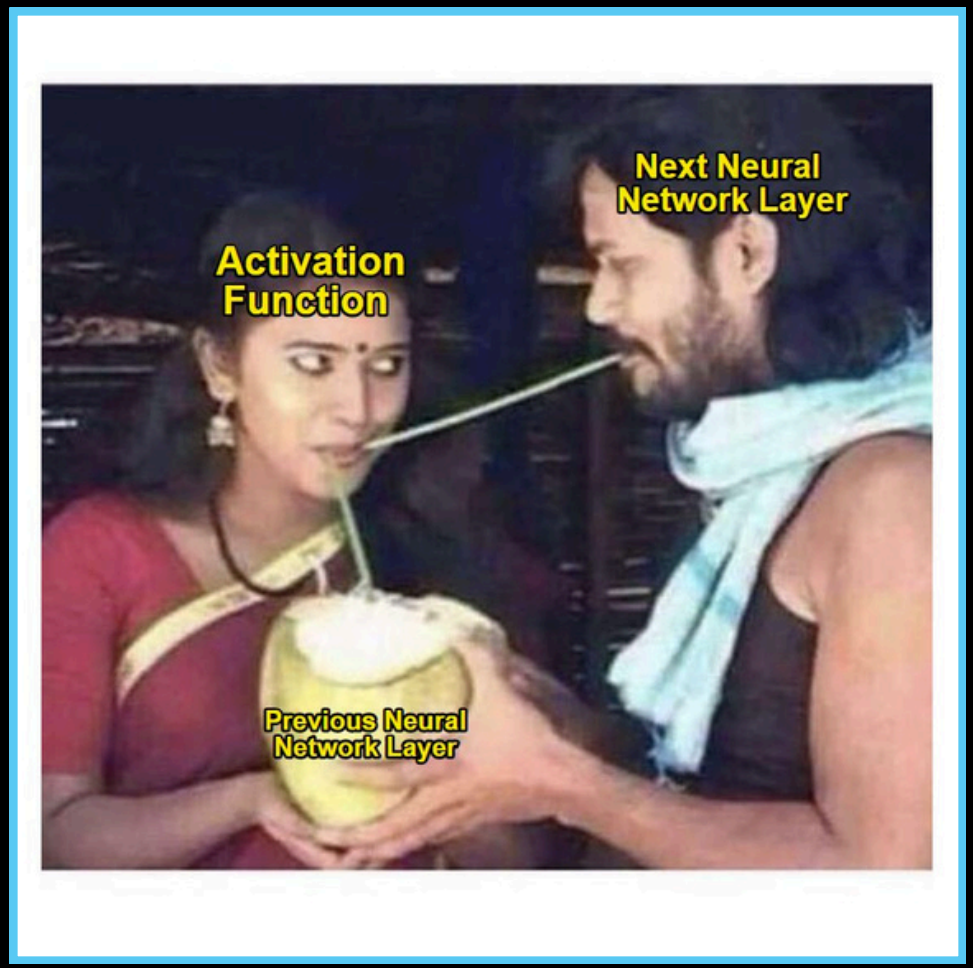
softmax

$$f(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

DeepDive™

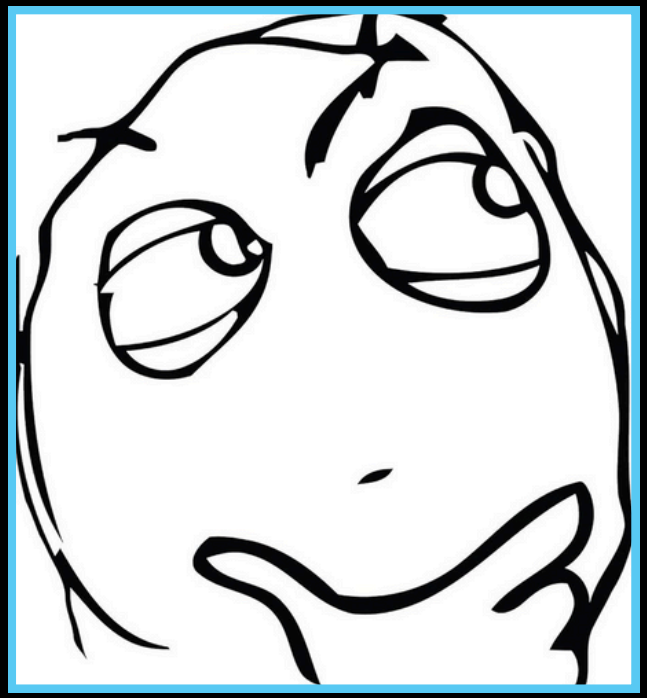
Funkcje aktywacji

Funkcja	Zastosowanie	Zakres	Wady
Skokowa	Prosta klasyfikacja (np. perceptron)	{0, 1}	Nieliniowa, nieróżniczkowalna
Sigmoid	Klasyfikacja binarna, probabilistyczne	(0, 1)	Zanikanie gradientu, nasycenie
tanh	Warstwy ukryte w sieciach neuronowych	(-1, 1)	Zanikanie gradientu
ReLU	Warstwy ukryte w złożonych sieciach	[0, ∞)	Martwe neurony
Softmax	Klasyfikacja wieloklasowa	(0, 1)	Szybko rośnie złożoność obliczeniowa


Next Neural
Network Layer
Activation
Function
Previous Neural
Network Layer

DeepDive™

Funkcje aktywacji



11 grudnia!

Zajęcia z prof. Markiem Krośnickim!

DeepDive™

Zadanie

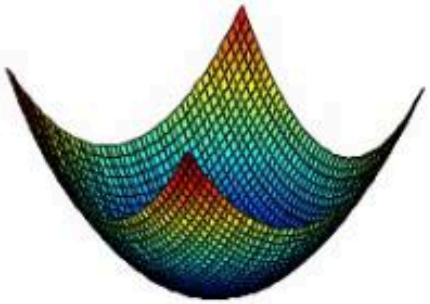
Próbka	Cecha 1 (x1)	Cecha 2 (x2)	Oczekiwany wynik (y)
1	0.5	1.2	0
2	0.8	0.4	1
3	1.0	1.0	1

$$z = w_1 \times x_1 + w_2 \times x_2 + b$$

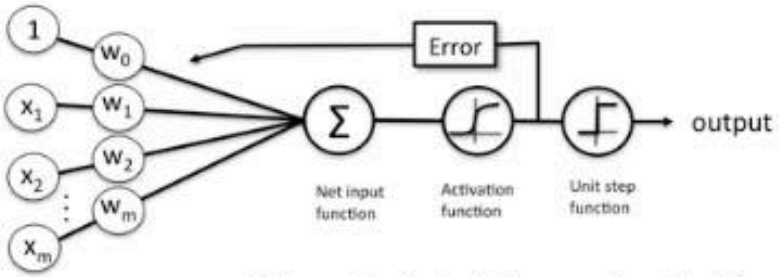
$$a = \sigma(z) = \frac{1}{1 + e^{-z}}$$

DeepDive™

You

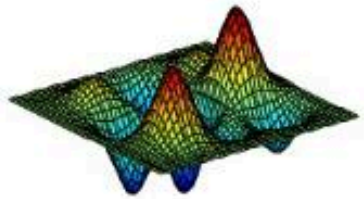


- Unique optimum: global/local.



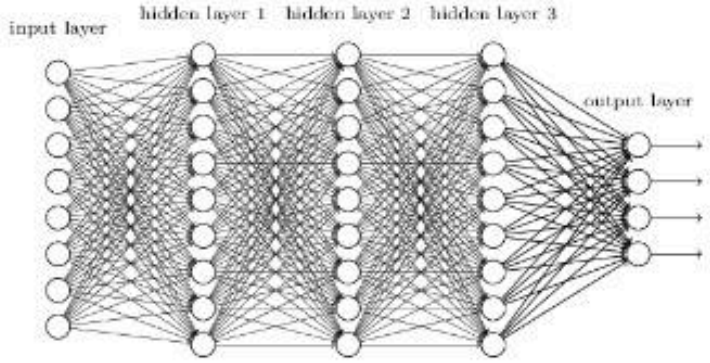
Schematic of a logistic regression classifier.

The guy she tells you not to worry about



- Multiple local optima
- In high dimensions possibly

Deep neural network



Struktura sieci neuronowej

```python
class ClassifierCNNNet(nn.Module):  1 usage
    def __init__(self, debug=False):
        super().__init__()
        self.debug = debug
        self.conv1 = nn.Conv2d( in_channels: 3,  out_channels: 20,  kernel_size: 5)
        self.pool = nn.MaxPool2d( kernel_size: 2,  stride: 2)
        self.conv2 = nn.Conv2d( in_channels: 20,  out_channels: 40,  kernel_size: 5)
        self.fc1 = nn.Linear(40 * 61 * 61,  out_features: 1000)
        self.fc2 = nn.Linear( in_features: 1000,  out_features: 500)
        self.fc3 = nn.Linear( in_features: 500,  out_features: 8)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        if self.debug: print(f"After conv1: {x.shape}")
        x = self.pool(x)
        if self.debug: print(f"After pool 1: {x.shape}")
        x = F.relu(self.conv2(x))
        if self.debug: print(f"After conv2: {x.shape}")
        x = self.pool(x)
        if self.debug: print(f"After pool 2: {x.shape}")
        x = torch.flatten(x, 1)  # flatten all dimensions except batch
        if self.debug: print(f"After flaten: {x.shape}")
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Przykład sieci konwolucyjnej do klasyfikacji, Jakub Susoł

DeepDive™


DeepDive™