





















Dane pochodzą z publicznie dostępnych zbiorów danych zawierających obrazy rentgenowskie klatki piersiowej wraz z etykietami chorób.

Opis Datasetu

Zestaw danych składa się z tysięcy obrazów, każdy reprezentujący unikalne warunki zdrowotne pacjenta. Dane pochodzą z publicznie dostępnych zbiorów danych zawierających obrazy rentgenowskie klatki piersiowej wraz z etykietami chorób.

Zestaw danych składa się z tysięcy obrazów, każdy reprezentujący unikalne warunki zdrowotne pacjenta.

Obrazy są klasyfikowane według różnych chorób, takich jak zapalenie płuc, torbiel, guz, itp.

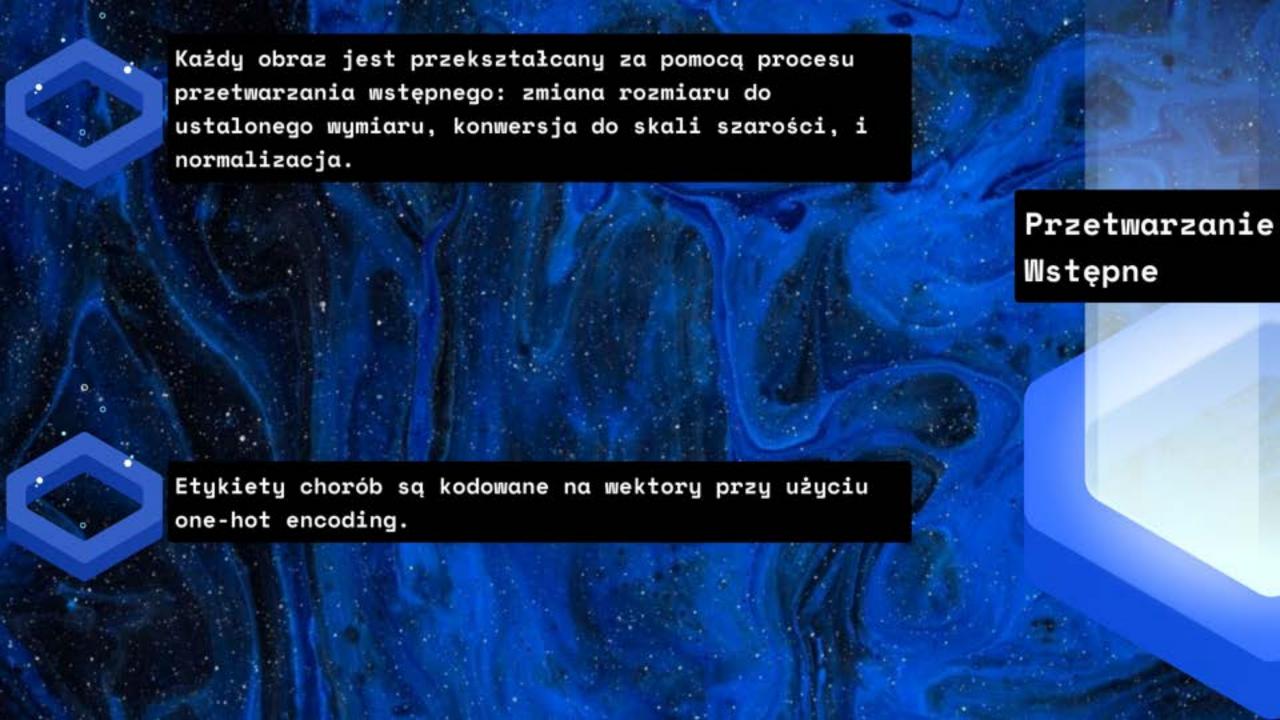
Opis Datasetu





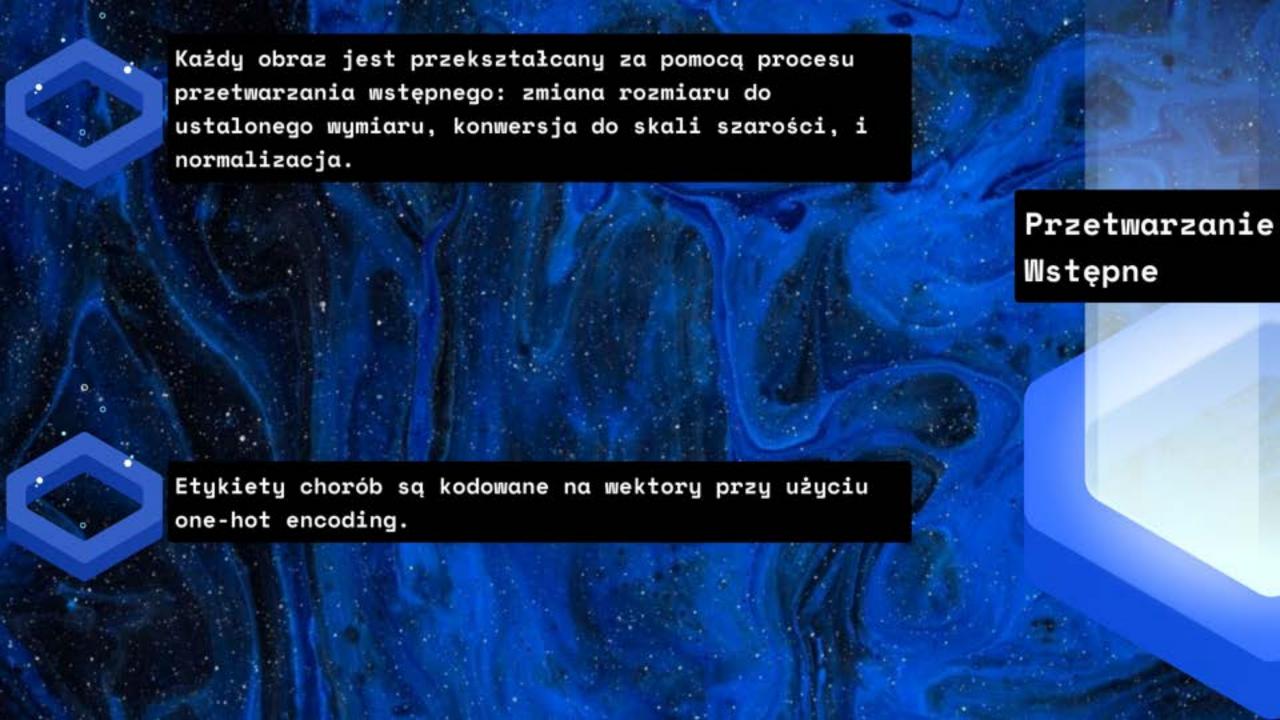






Kod odpowiedzialny za wstępne przetworzenie

```
train_transform = transforms.Compose(
    transforms.Resize((1024, 1024)),
   transforms.Grayscale(num_output_channels=3),
    transforms.ColorJitter(brightness=0.4, contrast=0.4, saturation=0.4, hye=0.1)
   transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.486], std=[0.229, 0.224, 0.225]),
val_transform = transforms.Compose([
   transforms.Grayscale(num_output_channels=3),
   transforms. ToTensor().
   transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
test_transform = transforms.Compose([
    transforms.Grayscale(num_output_channels=3),
   transforms. ToTensor().
   transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
```

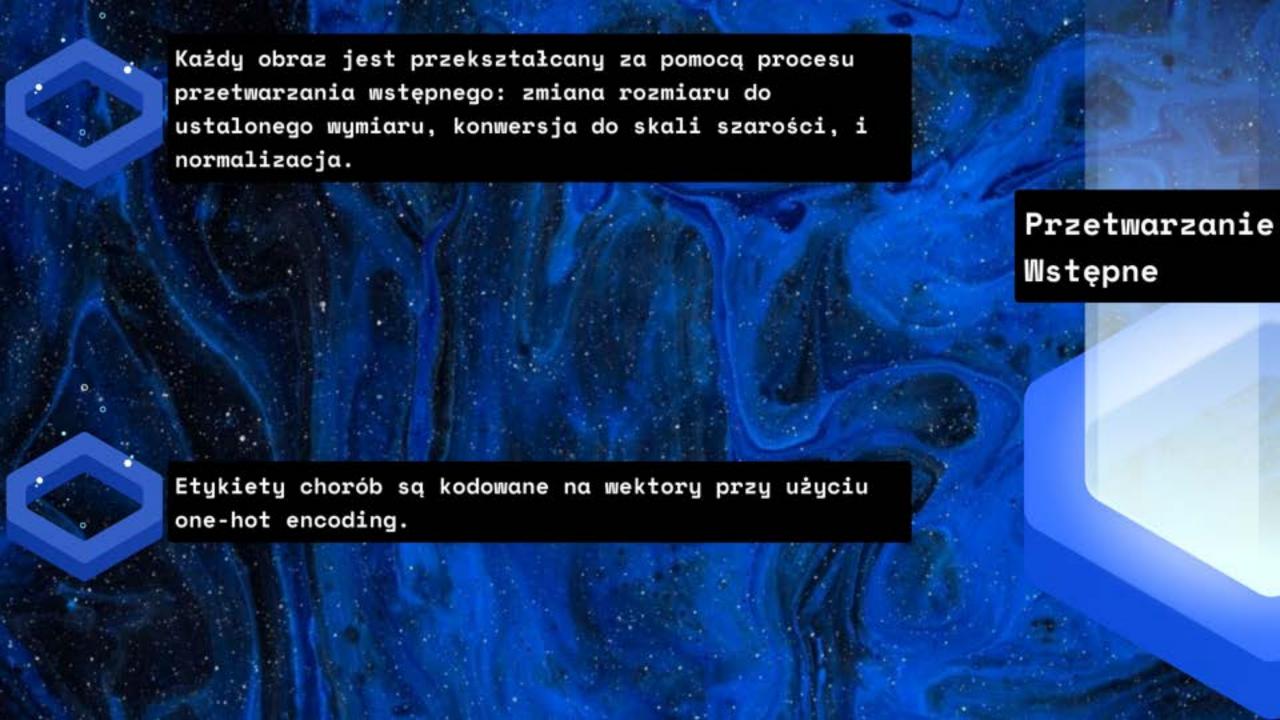


Klasa XrayDataset jest odpowiedzialna za tworzenie niestandardowego zbioru danych dla zadania klasyfikacji rentgenowskich obrazów. Działa jako interfejs między danymi (obrazami rentgenowskimi i etykietami) a modelem.

```
class XrayDataset(Dataset):
   def __init__(self, df, root_dir, transform=None):
        self.labels_frame = df['Finding Labels'].str.get_dummies('|')
       self.image_frame = df['Iwage Index']
       self.root_dir = root_dir
       self.transform = transform
   def _len_(self):
       return len(self.image_frame)
    def __getitem_ (self, idx):
       img_name = os.path.join(self.root_dir, self.image_frame.iloc[idx])
       image = Image.open(img_name)
        labels = self.labels_frame.iloc[idx].values.astype(np.float32)
        if self.transform:
            image = self.transform(image)
       return image, labels
```

Kod odpowiedzialny za one-hot encoding







```
Architektura
                         model = EfficientNet.from_name('efficientnet-b6')
   Model
                         num_ftrs = model._fc.in_features
                                                                                          Sieci
                         model._fc = nn.Linear(num_ftrs, len(train_dataset[0][1]))
                                                                                          Neuronalnej
           Optymalizator
                                     optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-5)
                         if epoch_val_loss < best_val_loss:
  Unikanie
                            best_val_loss = epoch_val_loss
przetrenowania
                            early_stop_counter = 0
                         else:
                            early_stop_counter += 1
                         if early_stop_counter >= patience:
                            print("Early Stopping: No improvement in validation loss. Training stopped.")
```



EfficientNet-B6

Mode1

ResNet (Residual Neural Network)
to model, który wprowadza
koncepcję połączeń skrótu, które
pomagają przeciwdziałać
problemowi zanikającego gradientu
w głębokich sieciach. ResNet-152
jest jednym z większych
wariantów, z 152 warstwami.
Dzięki swojej strukturze ResNet
jest w stanie efektywnie uczyć
się z dużej liczby warstw bez
utraty wydajności.

EfficientNet to seria modeli zaprojektowanych z myślą o skali. Zamiast dodawać więcej warstw, jak w przypadku ResNet, EfficientNet skupia się na równoczesnym zwiększaniu głębokości (liczby warstw), szerokości (liczby kanałów) i rozdzielczości (rozmiar wejścia). EfficientNet-B6 to szósta iteracja tej serii. Zasadniczo EfficientNet dąży do osiągnięcia wyższej wydajności przy niższym zużyciu zasobów.





Wady

Złożoność: ResNet-152, mając 152 warstwy, jest dosyć złożonym modelem, który może wymagać dużo mocy obliczeniowej i pamięci.

Czas treningu: Ze względu na swoją złożoność, ResNet-152 może potrzebować więcej czasu na trening w porównaniu do mniejszych modeli.

Optymalizacja: Mimo że ResNet radzi sobie z problemem zanikającego gradientu, może być trudno zoptymalizować sieć do osiągnięcia najlepszych możliwych wyników, szczególnie gdy model staje się coraz głębszy.

Potrzeba dużych zbiorów danych: Podobnie jak inne głębokie sieci neuronowe, ResNet-152 może potrzebować dużych, dobrze oznaczonych zbiorów danych do skutecznego uczenia się.

ResNet-152

Złożoność: ResNet-152, mając 152 warstwy, jest dosyć złożonym modelem, który może wymagać dużo mocy obliczeniowej i pamięci.

Czas treningu: Ze względu na swoją złożoność, ResNet-152 może potrzebować więcej czasu na trening w porównaniu do mniejszych modeli.

Optymalizacja: Mimo że ResNet radzi sobie z problemem zanikającego gradientu, może być trudno zoptymalizować sieć do osiągnięcia najlepszych możliwych wyników, szczególnie gdy model staje się coraz głębszy.

Potrzeba dużych zbiorów danych: Podobnie jak inne głębokie sieci neuronowe, ResNet-152 może potrzebować dużych, dobrze oznaczonych zbiorów danych do skutecznego uczenia się.

ResNet-152





Zalety

Skuteczność uczenia: ResNet, wprowadzając pojęcie połączeń skrótu (lub "residual connections"), skutecznie radzi sobie z problemem zanikającego gradientu, który jest typowy dla głębokich sieci neuronowych.

Skalowalność: Architektura ResNet jest skalowalna, co oznacza, że można ją łatwo rozszerzyć na większą liczbę warstw bez utraty wydajności.

Wszechstronność: ResNet może być używany do różnorodnych zadań, takich jak klasyfikacja obrazów, detekcja obiektów i segmentacja.

ResNet-152

Skuteczność uczenia: ResNet, wprowadzając pojęcie połączeń skrótu (lub "residual connections"), skutecznie radzi sobie z problemem zanikającego gradientu, który jest typowy dla głębokich sieci neuronowych.

Skalowalność: Architektura ResNet jest skalowalna, co oznacza, że można ją łatwo rozszerzyć na większą liczbę warstw bez utraty wydajności.

Wszechstronność: ResNet może być używany do różnorodnych zadań, takich jak klasyfikacja obrazów, detekcja obiektów i segmentacja.

ResNet-152





EfficientNet-B6

Mode1

ResNet (Residual Neural Network)
to model, który wprowadza
koncepcję połączeń skrótu, które
pomagają przeciwdziałać
problemowi zanikającego gradientu
w głębokich sieciach. ResNet-152
jest jednym z większych
wariantów, z 152 warstwami.
Dzięki swojej strukturze ResNet
jest w stanie efektywnie uczyć
się z dużej liczby warstw bez
utraty wydajności.

EfficientNet to seria modeli zaprojektowanych z myślą o skali. Zamiast dodawać więcej warstw, jak w przypadku ResNet, EfficientNet skupia się na równoczesnym zwiększaniu głębokości (liczby warstw), szerokości (liczby kanałów) i rozdzielczości (rozmiar wejścia). EfficientNet-B6 to szósta iteracja tej serii. Zasadniczo EfficientNet dąży do osiągnięcia wyższej wydajności przy niższym zużyciu zasobów.





Mady

Złożoność: EfficientNet-B6, będąc jednym z większych wariantów EfficientNet, jest dość złożonym modelem, który może wymagać dużo mocy obliczeniowej i pamięci.

Czas treningu: Podobnie jak w przypadku ResNet-152, trening EfficientNet-B6 może zająć więcej czasu, zwłaszcza gdy używamy dużych zbiorów danych.

Potrzeba dużych zbiorów danych: Aby w pełni wykorzystać potencjał EfficientNet-B6, może być konieczne użycie dużych, dobrze oznaczonych zbiorów danych.

Trudność w dostosowaniu: Ze względu na złożoność architektury, EfficientNet może być trudniejszy do dostosowania i zoptymalizowania w porównaniu do prostszych modeli.

EfficientNet-B6







Zalety

Skuteczność: EfficientNets osiąga wyższe wyniki na benchmarkach, takich jak ImageNet, w porównaniu do wielu innych modeli, takich jak ResNets czy DenseNets.

Efektywność: Architektura EfficientNet została zaprojektowana tak, aby zwiększać jednocześnie głębokość (liczbę warstw), szerokość (liczbę kanałów) i rozdzielczość (rozmiar wejścia), co prowadzi do większej efektywności niż tradycyjne podejście do skalowania sieci.

Skalowalność: Tak jak ResNets, EfficientNets są skalowalne, ale z większą efektywnością. Można je łatwo skalować do większych rozmiarów, co może prowadzić do jeszcze lepszej wydajności.

EfficientNet-B6







EfficientNet-B6

Mode1

ResNet (Residual Neural Network)
to model, który wprowadza
koncepcję połączeń skrótu, które
pomagają przeciwdziałać
problemowi zanikającego gradientu
w głębokich sieciach. ResNet-152
jest jednym z większych
wariantów, z 152 warstwami.
Dzięki swojej strukturze ResNet
jest w stanie efektywnie uczyć
się z dużej liczby warstw bez
utraty wydajności.

EfficientNet to seria modeli zaprojektowanych z myślą o skali. Zamiast dodawać więcej warstw, jak w przypadku ResNet, EfficientNet skupia się na równoczesnym zwiększaniu głębokości (liczby warstw), szerokości (liczby kanałów) i rozdzielczości (rozmiar wejścia). EfficientNet-B6 to szósta iteracja tej serii. Zasadniczo EfficientNet dąży do osiągnięcia wyższej wydajności przy niższym zużyciu zasobów.

```
Architektura
                         model = EfficientNet.from_name('efficientnet-b6')
   Model
                         num_ftrs = model._fc.in_features
                                                                                          Sieci
                         model._fc = nn.Linear(num_ftrs, len(train_dataset[0][1]))
                                                                                          Neuronalnej
           Optymalizator
                                     optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-5)
                         if epoch_val_loss < best_val_loss:
  Unikanie
                            best_val_loss = epoch_val_loss
przetrenowania
                            early_stop_counter = 0
                         else:
                            early_stop_counter += 1
                         if early_stop_counter >= patience:
                            print("Early Stopping: No improvement in validation loss. Training stopped.")
```



SGD jest jednym z najprostszych algorytmów optymalizacji. Działa on poprzez aktualizację parametrów modelu po każdej iteracji, minimalizując błąd wyjścia. W każdej iteracji używa losowo wybranego podzbioru danych treningowych (zwanego mini-batch) do obliczenia gradientu funkcji bledu. Niestety, SGD ma tendencje do "wedrowania" wzdłuż dolin, co powoduje, że może być trudne osiągnięcie optymalnego punktu. Może także mieć problemy ze zbieganiem, jeśli funkcja błędu jest bardzo

nieregularna.

ADAM

ADAM jest bardziej zaawansowanym algorytmem optymalizacji, który jest skonstruowany tak, aby radzić sobie z niektórymi problemami SGD. ADAM używa adaptacyjnych współczynników uczenia się dla różnych parametrów, co oznacza, że niektóre parametry mogą być aktualizowane szybciej, a inne wolniej. ADAM zazwyczaj zbiega szybciej i jest bardziej stabilny niż SGD. Ponadto, ADAM jest mniej wrażliwy na wybór początkowego współczynnika uczenia się.

Optymalizator





Wady

Wędrowanie: SGD ma tendencję do "wędrowania" wzdłuż dolin, co sprawia, że może być trudne osiągnięcie optymalnego punktu.

Czułość na hiperparametry: SGD jest bardzo wrażliwy na dobór odpowiedniego współczynnika uczenia się. Zbyt duży może prowadzić do niestabilnych wyników, zbyt mały może spowodować bardzo wolne zbieganie.

Brak adaptacyjnej stopy uczenia się: SGD aktualizuje wszystkie wagi jednym i tym samym współczynnikiem uczenia się, co może nie być optymalne, jeśli niektóre wagi powinny być aktualizowane szybciej, a inne wolniej.

Problemy ze zbieganiem: SGD może mieć problemy ze zbieganiem, jeśli funkcja błędu jest bardzo nieregularna lub zawiera wiele minimów lokalnych. SGD







Zalety

Prostota: SGD jest relatywnie prosty do zrozumienia, zaimplementowania i dostosowania. Algorytm jest prosty, co czyni go doskonałym wyborem dla osób rozpoczynających pracę z uczeniem maszynowym.

Skalowalność: SGD jest efektywny obliczeniowo, dzięki czemu jest dobrym wyborem dla dużych zestawów danych.

Efektywność pamięciowa: SGD nie wymaga
przechowywania wszystkich danych treningowych w
pamięci na raz, co czyni go bardziej efektywnym
pamięciowo w porównaniu do algorytmów optymalizacji
działających na pełnym zestawie danych.

Właściwości statystyczne: Ze względu na swoją stochastyczną naturę, SGD ma tendencję do "ucieczki" z płytkich minimów lokalnych, które mogą utrudniać proces optymalizacji.









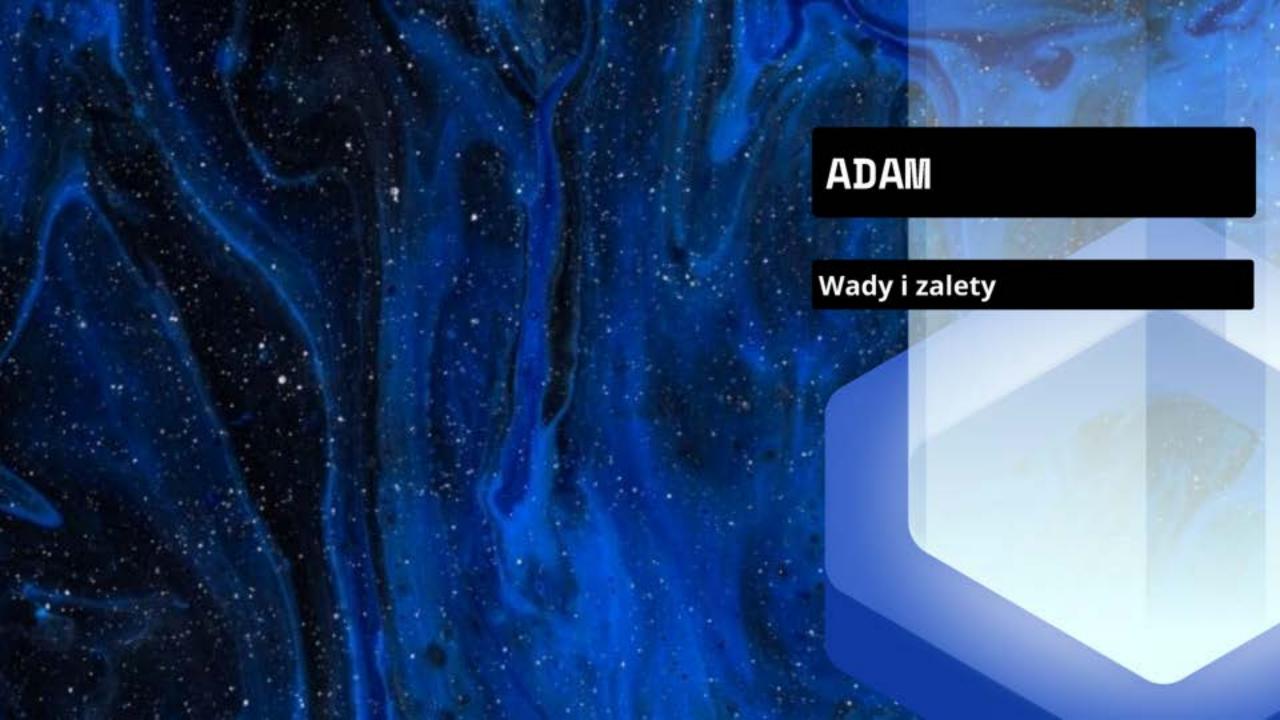
SGD jest jednym z najprostszych algorytmów optymalizacji. Działa on poprzez aktualizację parametrów modelu po każdej iteracji, minimalizując błąd wyjścia. W każdej iteracji używa losowo wybranego podzbioru danych treningowych (zwanego mini-batch) do obliczenia gradientu funkcji bledu. Niestety, SGD ma tendencje do "wedrowania" wzdłuż dolin, co powoduje, że może być trudne osiągnięcie optymalnego punktu. Może także mieć problemy ze zbieganiem, jeśli funkcja błędu jest bardzo

nieregularna.

ADAM

ADAM jest bardziej zaawansowanym algorytmem optymalizacji, który jest skonstruowany tak, aby radzić sobie z niektórymi problemami SGD. ADAM używa adaptacyjnych współczynników uczenia się dla różnych parametrów, co oznacza, że niektóre parametry mogą być aktualizowane szybciej, a inne wolniej. ADAM zazwyczaj zbiega szybciej i jest bardziej stabilny niż SGD. Ponadto, ADAM jest mniej wrażliwy na wybór początkowego współczynnika uczenia się.

Optymalizator





Mady

Złożoność: ADAM jest bardziej złożony od SGD pod względem matematycznym i implementacyjnym.

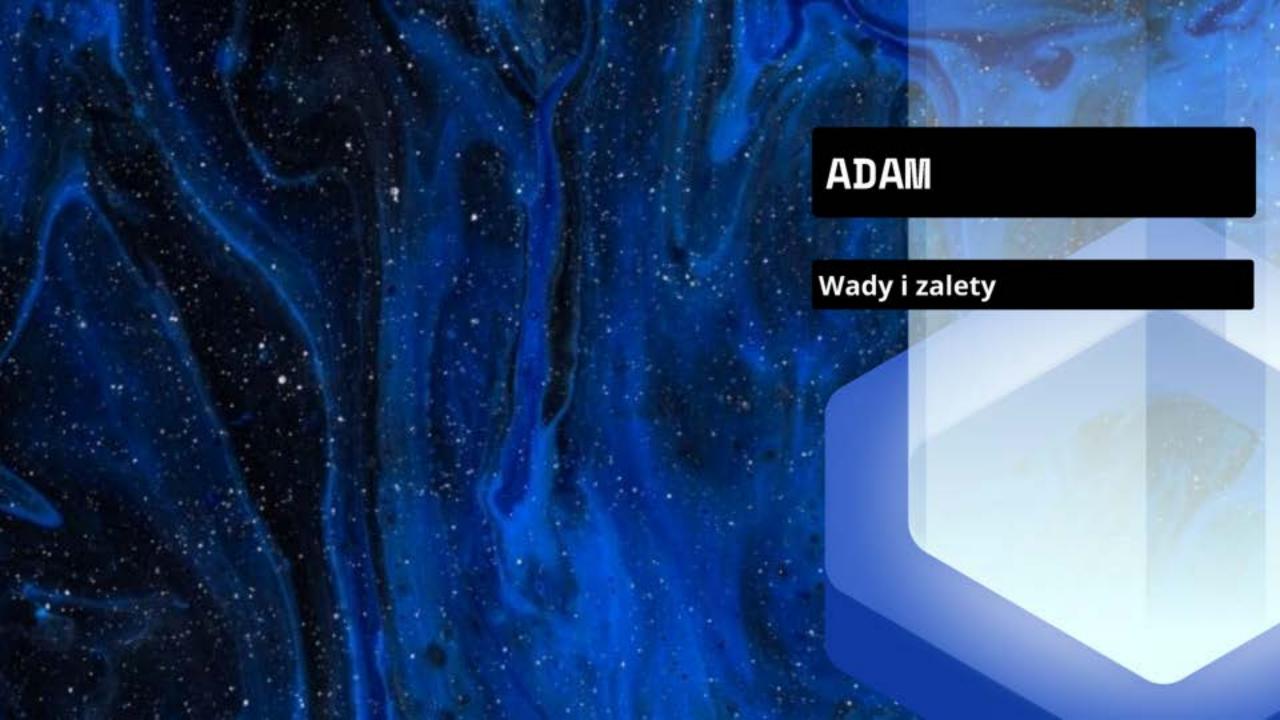
Wielkość pamięci: ADAM przechowuje oszacowania momentów pierwszego i drugiego rzędu dla każdego parametru, co zwiększa wymagania pamięci.

Bias korekta: Choć bias korekta używana w ADAM pomaga stabilizować estymacje momentów, niektóre badania sugerują, że może to prowadzić do nieoptymalnych wyników w niektórych sytuacjach.

Problem z generalizacją: Niektóre badania wykazały, że mimo szybkiego zbiegania, ADAM może mieć problem z generalizacją na niektórych zestawach danych w porównaniu do innych optymalizatorów, takich jak SGD.

ADAM







Zalety

Adaptacyjny współczynnik uczenia się: ADAM automatycznie dostosowuje współczynnik uczenia się dla różnych parametrów, co oznacza, że niektóre parametry mogą być aktualizowane szybciej, a inne wolniej. Dzięki temu jest bardziej elastyczny i skuteczny w różnych scenariuszach.

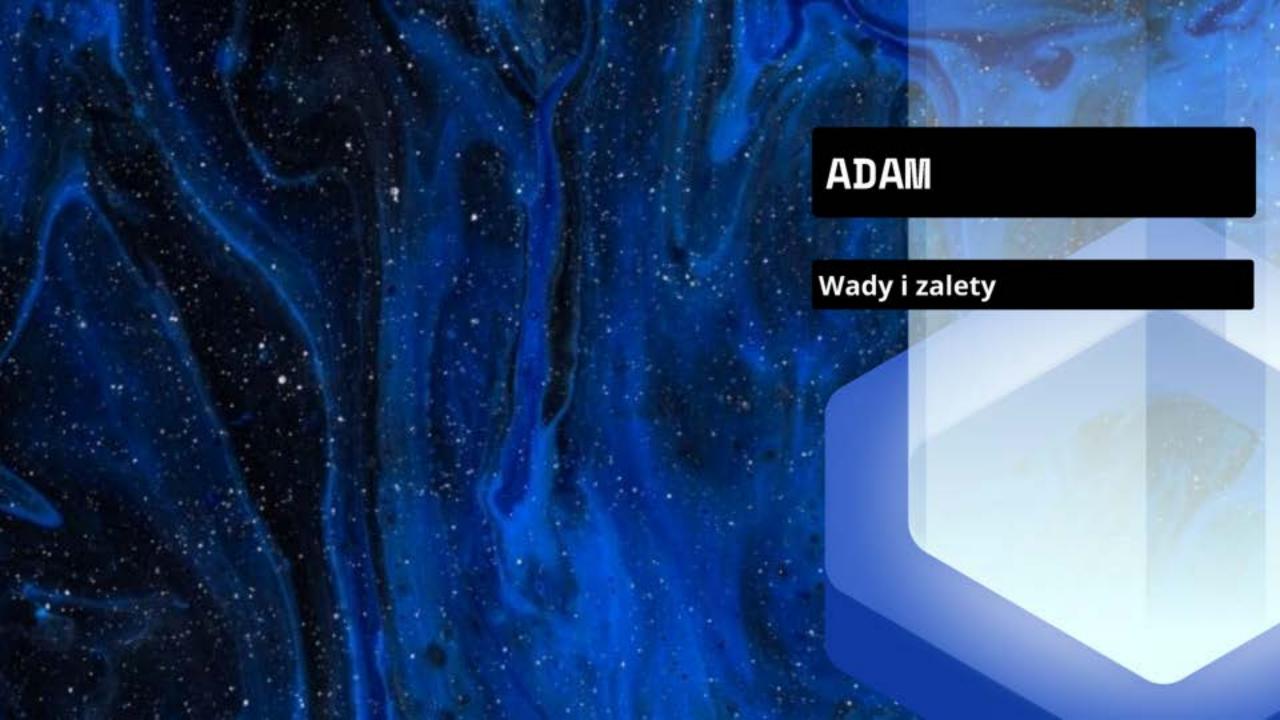
Elastyczność: ADAM działa dobrze z małymi, dużymi i / lub rzadkimi danymi. Jest skuteczny dla różnego rodzaju funkcji błędu i architektur modeli.

Szybkość zbiegania: ADAM zazwyczaj zbiega szybciej niż wiele innych algorytmów optymalizacji, takich jak SGD.

Stabilność: ADAM jest mniej wrażliwy na zmiany współczynnika uczenia się, co czyni go bardziej stabilnym w praktyce.









SGD jest jednym z najprostszych algorytmów optymalizacji. Działa on poprzez aktualizację parametrów modelu po każdej iteracji, minimalizując błąd wyjścia. W każdej iteracji używa losowo wybranego podzbioru danych treningowych (zwanego mini-batch) do obliczenia gradientu funkcji bledu. Niestety, SGD ma tendencje do "wedrowania" wzdłuż dolin, co powoduje, że może być trudne osiągnięcie optymalnego punktu. Może także mieć problemy ze zbieganiem, jeśli funkcja błędu jest bardzo

nieregularna.

ADAM

ADAM jest bardziej zaawansowanym algorytmem optymalizacji, który jest skonstruowany tak, aby radzić sobie z niektórymi problemami SGD. ADAM używa adaptacyjnych współczynników uczenia się dla różnych parametrów, co oznacza, że niektóre parametry mogą być aktualizowane szybciej, a inne wolniej. ADAM zazwyczaj zbiega szybciej i jest bardziej stabilny niż SGD. Ponadto, ADAM jest mniej wrażliwy na wybór początkowego współczynnika uczenia się.

Optymalizator

```
Architektura
                         model = EfficientNet.from_name('efficientnet-b6')
   Model
                         num_ftrs = model._fc.in_features
                                                                                          Sieci
                         model._fc = nn.Linear(num_ftrs, len(train_dataset[0][1]))
                                                                                          Neuronalnej
           Optymalizator
                                     optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-5)
                         if epoch_val_loss < best_val_loss:
  Unikanie
                            best_val_loss = epoch_val_loss
przetrenowania
                            early_stop_counter = 0
                         else:
                            early_stop_counter += 1
                         if early_stop_counter >= patience:
                            print("Early Stopping: No improvement in validation loss. Training stopped.")
```

Co to?

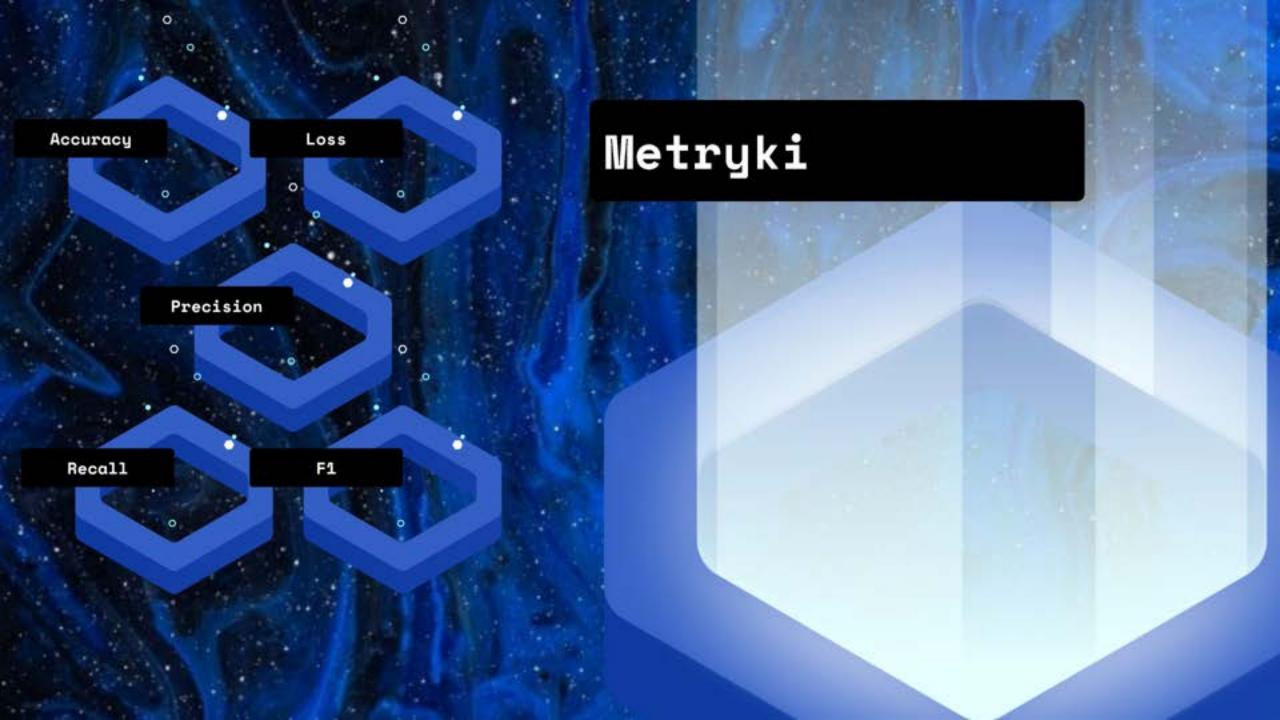
Early-early stopping jest stosowany w celu uniknięcia przeuczenia modelu, poprawy zdolności do generalizacji oraz zmniejszenia czasu i zasobów potrzebnych do uczenia. Dzięki tej technice możemy wybrać optymalny punkt zatrzymania uczenia modelu, który zapewnia najlepszą wydajność na danych nieznanych.

Early-stopping

```
Architektura
                         model = EfficientNet.from_name('efficientnet-b6')
   Model
                         num_ftrs = model._fc.in_features
                                                                                          Sieci
                         model._fc = nn.Linear(num_ftrs, len(train_dataset[0][1]))
                                                                                          Neuronalnej
           Optymalizator
                                     optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-5)
                         if epoch_val_loss < best_val_loss:
  Unikanie
                            best_val_loss = epoch_val_loss
przetrenowania
                            early_stop_counter = 0
                         else:
                            early_stop_counter += 1
                         if early_stop_counter >= patience:
                            print("Early Stopping: No improvement in validation loss. Training stopped.")
```

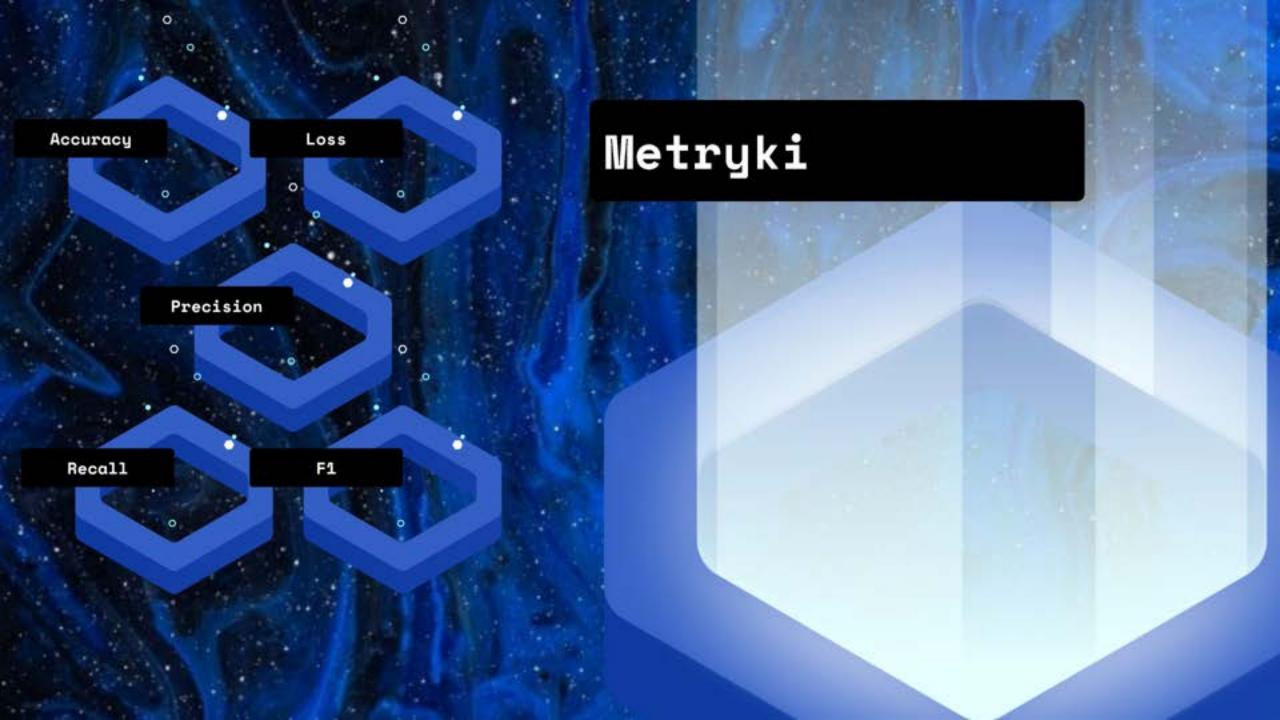






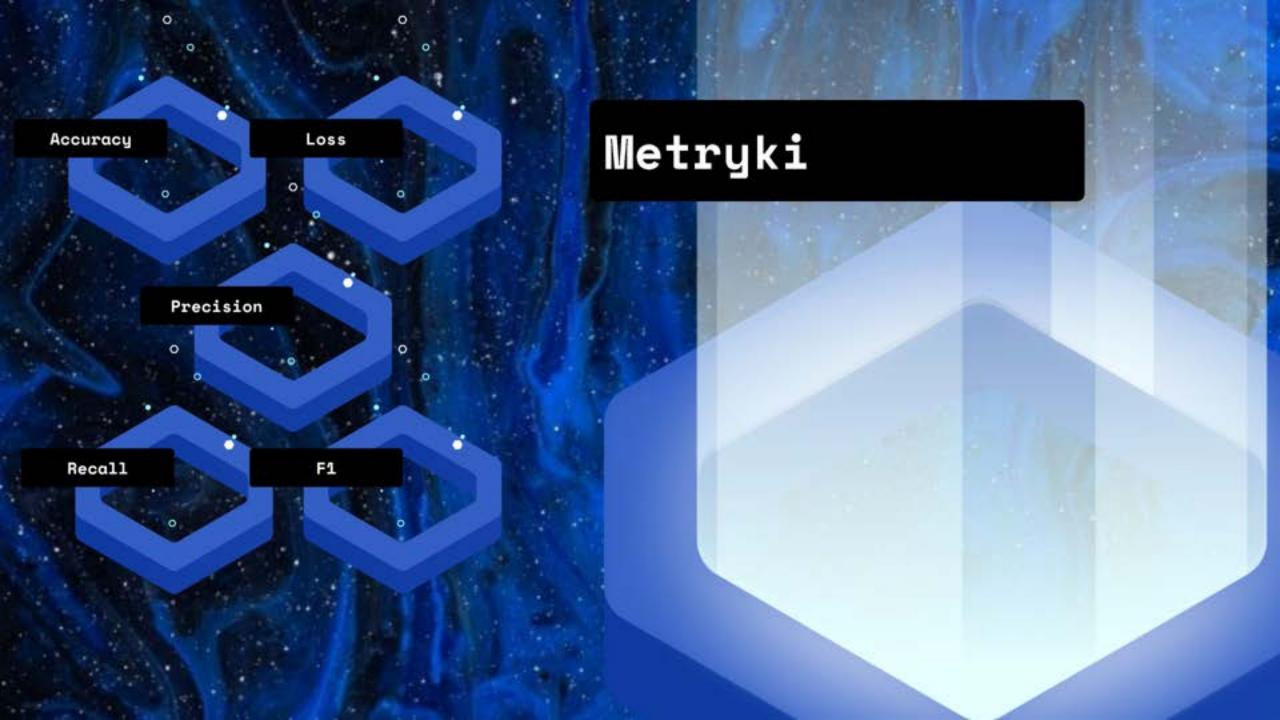
Accuracy

To jest podstawowa metryka, która jest używana do oceny modeli klasyfikacji. Accuracy definiuje stosunek prawidłowo przewidzianych obserwacji do całkowitej liczby obserwacji. Czyli, jest to procent prawidłowych prognoz.



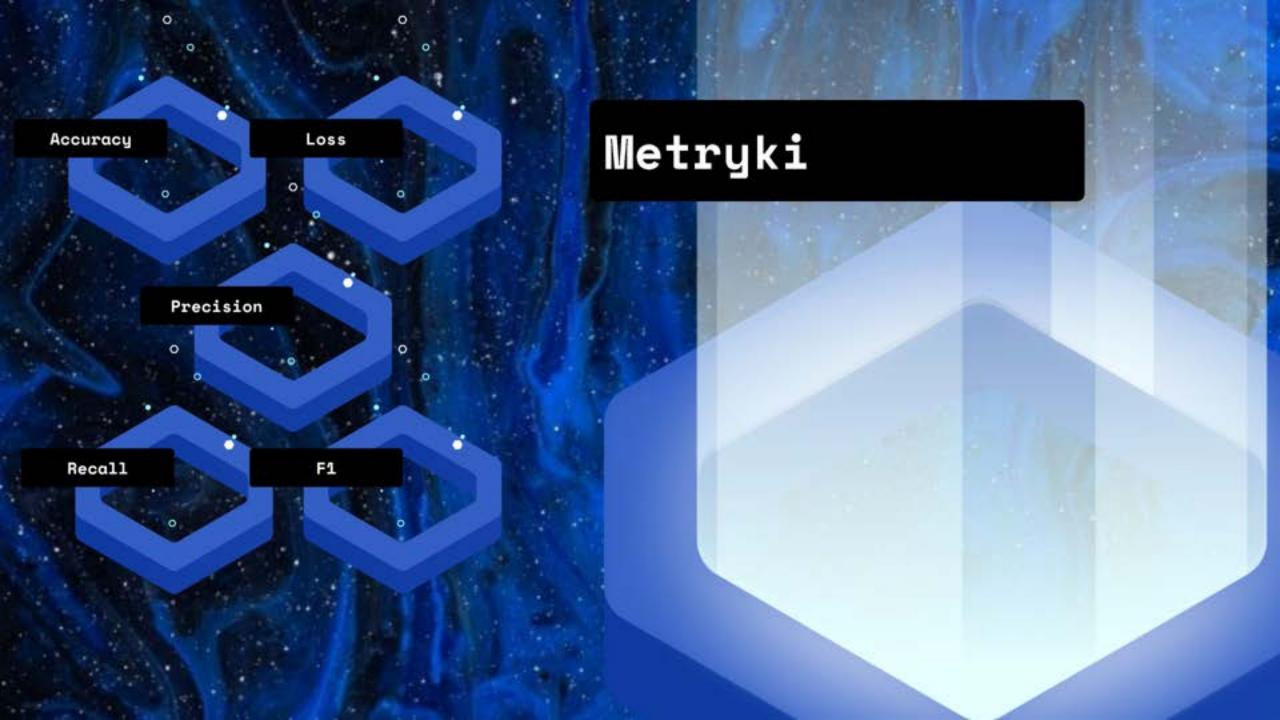
Loss

Loss (inaczej koszt) to funkcja, którą chcemy zminimalizować podczas trenowania modelu. W kontekście uczenia maszynowego, loss jest sumą błędów dla każdego przykładu treningowego. Im niższa wartość loss, tym lepszy model.



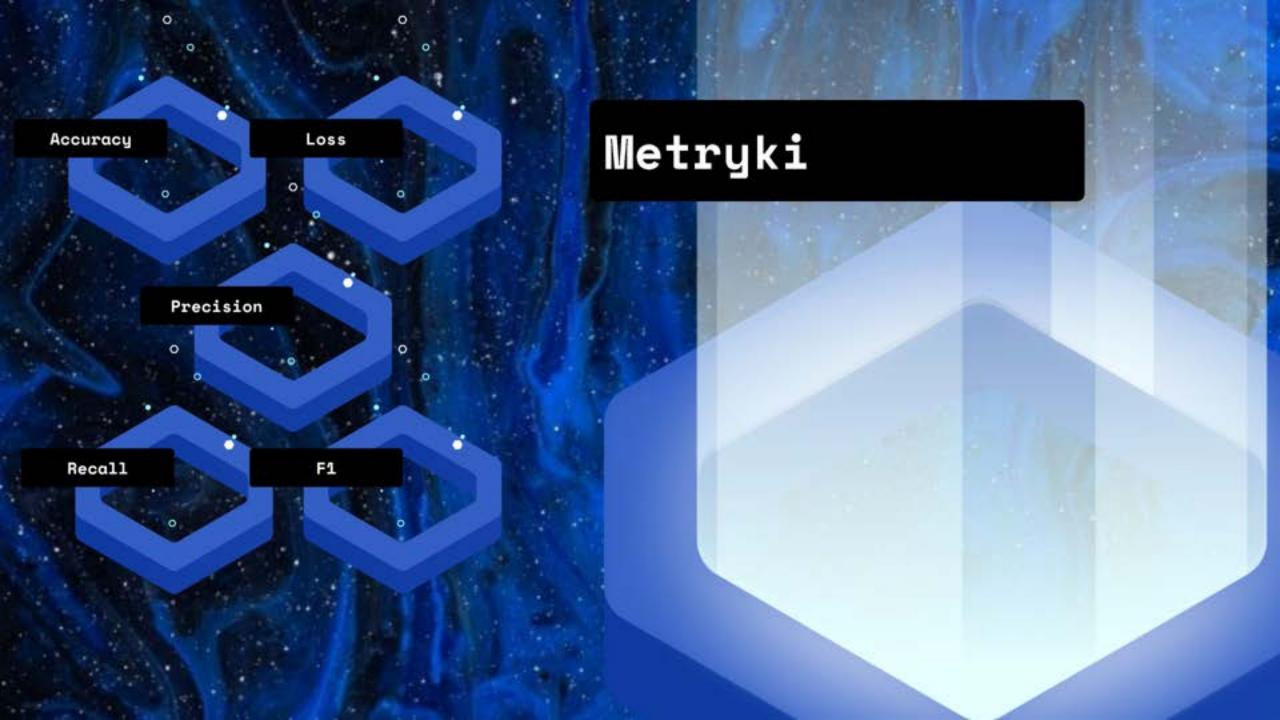
Precision

Precision (precyzja) to stosunek prawidłowo przewidzianych pozytywnych obserwacji do całkowitej liczby przewidzianych pozytywnych obserwacji. Jest to miara, jak wiele z naszych pozytywnych prognoz jest faktycznie prawdziwych.



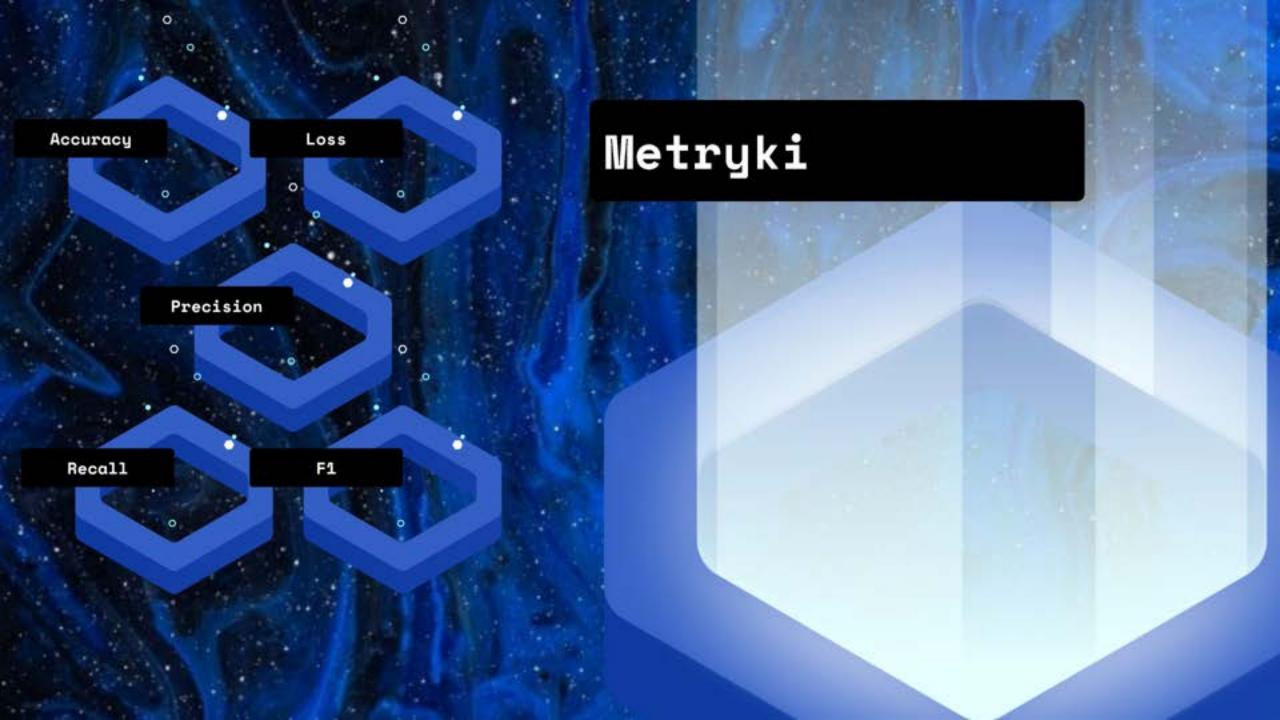
Recall

Recall (czułość) to stosunek prawidłowo przewidzianych pozytywnych obserwacji do wszystkich obserwacji w rzeczywistej klasy pozytywnej. Innymi słowy, jest to miara, jak wiele z rzeczywistych pozytywnych prognozowaliśmy poprawnie.



F1 Score

F1 Score to harmoniczna średnia z Precision i Recall. Przyjmuje wartości od 0 do 1, gdzie 1 oznacza doskonałą precyzję i czułość. Ta metryka jest użyteczna, gdy mamy nierównomierny rozkład klas, ponieważ bierze pod uwagę zarówno fałszywe pozytywne, jak i fałszywe negatywne.





Trening

```
C:\Users\pistr\Desktop\PPY-projekt*python main.gy
Using 1 GPU for training
True
11.8
1
NVIDIA GeForce NTX 3888 Ti
Epoch 1/56

Training 1884 | 60210/69219 [0:52:05=88:00, 1.951t/s, acc=03, f1=8.532, loss=8.201, precision=0.62, recall=0.532]
Training 1885: 0.2668, Training Accuracy: 92.97%, Training F1: 53.22%
Validation: 1885 | 60210/69219 [0:52:05=88:00, 1.951t/s, acc=03, f1=8.583, loss=8.281, prec=8.583]
Validation: 1885 | 60210/69219 [0:52:05=88:00, 1.951t/s, acc=03, f1=8.583, loss=8.281, prec=8.583]
Validation: 1885 | 60210/69219 [0:32:05=80, 03.03%, Validation Precision: 88.27%, Validation F2: 58.27%
Epoch 2/50

Training: 57% | 19250/69219 [0:32:01=2:06:00, 0.13it/s, acc=03, f1=8.54, loss=8.2, precision=0.017, recall=0.54]
```

Prezentacja Wyników

Test

```
File: 00000013_014.png, Predicted classes: ['No Finding'], True classes: ['No Finding']
File: 00000013_015.png, Predicted classes: ['No Finding'], True classes: ['No Finding']
File: 00000013_016.png, Predicted classes: ['No Finding'], True classes: ['No Finding']
File: 00000013_017.png, Predicted classes: ['No Finding'], True classes: ['No Finding']
```





Wnioski i dalsze kierunki

Mój projekt pokazuje, że zastosowanie konwolucyjnych sieci neuronowych w analizie obrazów medycznych ma duże możliwości.

W przyszłości projekt może być rozwinięty poprzez dodanie więcej chorób do klasyfikacji, poprawę dokładności modelu, czy zastosowanie innych technik uczenia maszynowego.

