Zastosowanie sieci spolotowej w wybranym problemie klasyfikacji wizyjnej

Mikołaj Michalczyk

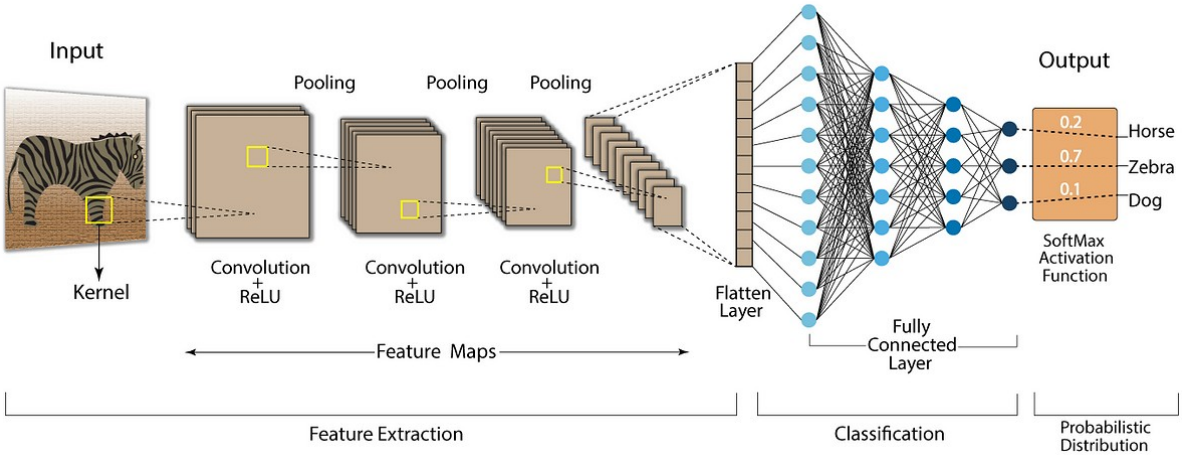
Piotr Tokarczyk

Sztuczna Inteligencja
Projekt 2023/2024

Wydział Inżynierii Elektrycznej i Komputerowej
Politechnika Krakowska

Wprowadzenie teoretyczne (matematyczne)

Sieci splotowe (ang. Convolutional Neural Networks, CNN) to klasa sztucznych sieci neuronowych zaprojektowanych z myślą o przetwarzaniu danych o strukturze siatki, takich jak obrazy. CNN zdobyły popularność dzięki swojej wyjątkowej zdolności do automatycznego ekstraktowania i hierarchizowania cech z surowych danych wejściowych, co czyni je szczególnie efektywnymi w zadaniach związanych z analizą obrazów.



Podstawowe Pojęcia i Operacje

Splot (ang. Convolution): Operacja splotu jest podstawą działania CNN. Polega ona na przesuwaniu filtra (jądra) po całym obrazie wejściowym i obliczaniu sumy elementów iloczynu punktowego filtra i fragmentu obrazu. Wartości te są zapisywane w nowej macierzy zwanej mapą cech (ang. feature map). Filtry w CNN uczą się wykrywać różne cechy obrazu, takie jak krawędzie, tekstury czy bardziej złożone wzory. Matematycznie, operację splotu dla obrazu I o wymiarach W x H i filtra F o wymiarach K x K można zapisać jako:

$$(I * F)(x, y) = \Sigma_{i=0}^{K-1} \Sigma_{j=0}^{K-1} I(x + i, y + j) \cdot F(i, j)$$

gdzie x i y to współrzędne piksela w obrazie wyjściowym.

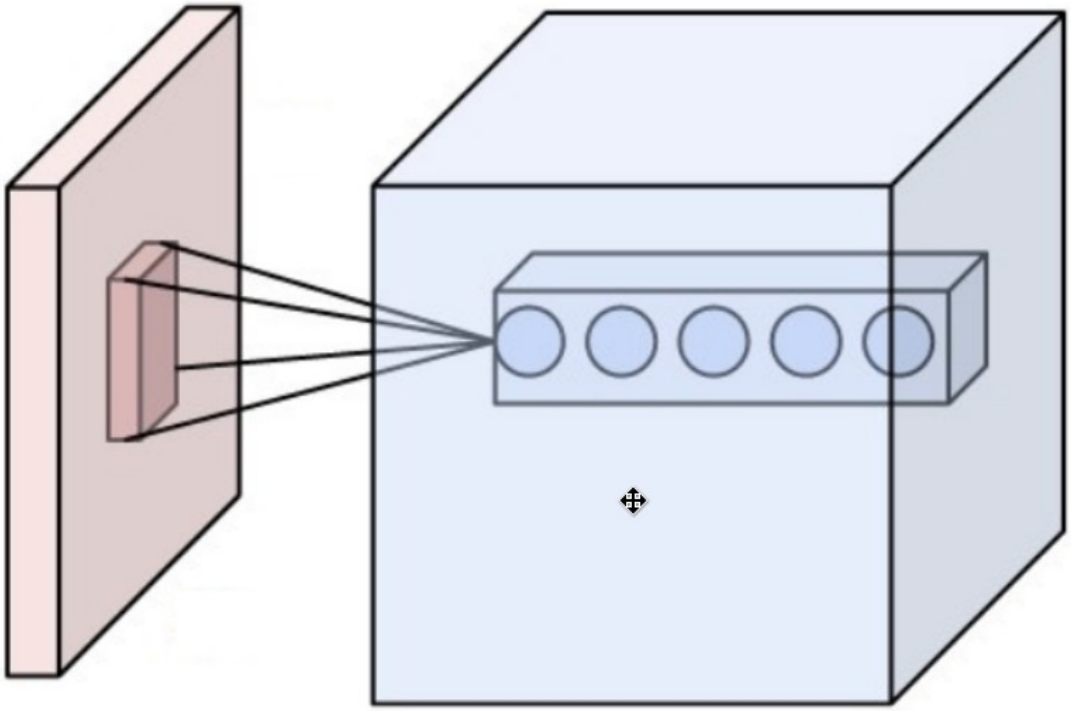
Padding: Aby zachować rozmiar obrazu po operacji splotu, stosuje się padding, czyli dodawanie zerowych ramek wokół obrazu wejściowego. Dla filtra F o wymiarach K x K, padding p zwykle wynosi:

$$p = \frac{K-1}{2}.$$

Stride: Stride to krok, o jaki przesuwany jest filtr po obrazie. Standardowo stride wynosi 1, co oznacza, że filtr przesuwany jest o jeden piksel na raz. Zwiększenie stride powoduje zmniejszenie rozmiaru mapy cech.

Warstwa Splotowa (ang. Convolutional Layer): Składa się z zestawu filtrów stosowanych do obrazu wejściowego. Każdy filtr generuje swoją mapę cech, a zestaw tych map tworzy wyjście warstwy splotowej. Liczba filtrów oraz ich rozmiary są hiperparametrami sieci.

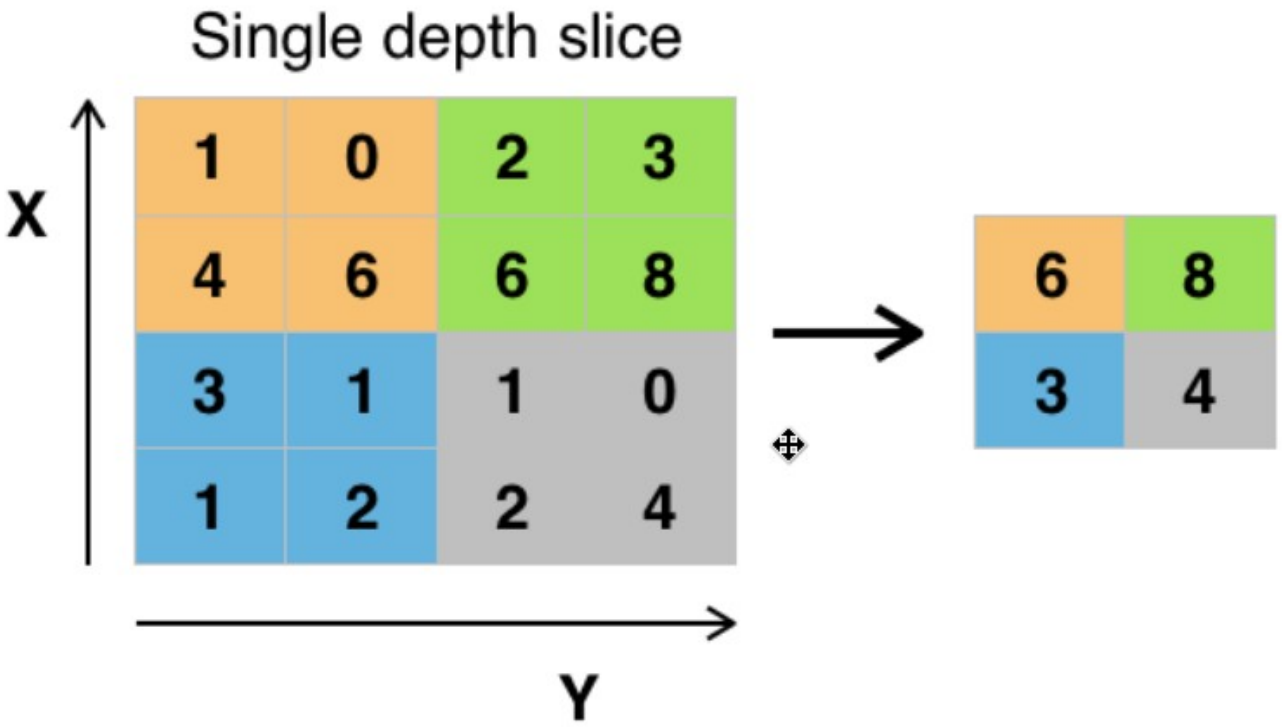
(Przykład propagacji w przód w 1-wymiarowej splotowej sieci neuronowej)



(Neurony warstwy splotowej (niebieskie), połączone ze swoim polem recepcyjnym (czerwone))

Warstwa Poolingu (ang. Pooling Layer): Warstwy poolingowe zmniejszają wymiarowość map cech poprzez subsampling, co redukuje liczbę parametrów i obliczeń, a także pomaga w kontroli nadmiernego dopasowania. Najpopularniejszym typem poolingu jest max pooling, który wybiera maksymalną wartość w każdym oknie poolingowym. Matematycznie, operację max poolingu dla mapy cech A o wymiarach W x H z oknem poolingowym o wymiarach P x P i stride s można zapisać jako:

$$A'_{i,j} = max\{A_{m,n} : m \in [i \cdot s,\ i \cdot s + P],\ n \in [j \cdot s,\ j \cdot s + P]\}$$



(max pooling z filtrem F 2 x 2 i stride = 2)

Warstwa Całkowicie Połączona (ang. Fully Connected Layer): Po kilku warstwach splotowych i poolingowych, mapy cech są spłaszczane do jednowymiarowego wektora i podawane do klasycznych warstw neuronowych, które wykonują ostateczną klasyfikację. Każdy neuron w warstwie jest połączony z każdym neuronem w poprzedniej warstwie.

Proces Klasyfikacji Obrazów za Pomocą CNN

1. Ekstrakcja Cech: Warstwy splotowe uczą się wykrywać różne poziomy cech na obrazach. Pierwsze warstwy mogą wykrywać podstawowe cechy, takie jak krawędzie, podczas gdy kolejne warstwy mogą identyfikować bardziej złożone struktury.

2. Redukcja Wymiarowości: Warstwy poolingowe zmniejszają wymiarowość map cech, co pozwala na redukcję liczby parametrów i obliczeń oraz zwiększa efektywność sieci.

3. Klasyfikacja: Wyjście z ostatniej warstwy splotowej jest spłaszczane i podawane do warstw całkowicie połączonych, które przekształcają dane w wektor prawdopodobieństw klas. Wykorzystuje się funkcję softmax, aby uzyskać rozkład prawdopodobieństwa dla różnych klas:

$$softmax(z_i) = \frac{e^{z_i}}{\Sigma_j e^{z_j}}$$

gdzie z_i to wyjściowa aktywacja dla klasy i.

Przykład Zastosowania: Klasyfikacja Obrazów Zwierząt

Rozważmy problem klasyfikacji obrazów różnych gatunków zwierząt (np. psy, koty, ptaki). CNN można skonfigurować następująco:

1. Warstwy Splotowe: Kilka warstw splotowych z różnymi filtrami pozwala na nauczenie się rozpoznawania różnych cech zwierząt, takich jak futro, uszy, czy kształt dzioba. Na przykład, pierwsza warstwa splotowa może składać się z 32 filtrów o rozmiarze 3 x 3, co oznacza, że każdy filtr będzie próbował wykryć różne cechy na podstawie małych fragmentów obrazu.

2. Warstwy Poolingowe: Co kilka warstw splotowych stosujemy warstwy max poolingowe, aby zmniejszyć wymiarowość map cech i skupić się na najważniejszych informacjach. Na przykład, po dwóch warstwach splotowych możemy zastosować warstwę max poolingu o rozmiarze 2 x 2 i stridzie 2, co zmniejszy wymiary map cech o połowę.

3. Warstwy Całkowicie Połączone: Na końcu stosujemy jedną lub więcej warstw całkowicie połączonych, które przekształcają wyekstrahowane cechy w końcową decyzję klasyfikacyjną. Przykładowo, po spłaszczeniu map cech możemy zastosować dwie warstwy całkowicie połączone z odpowiednio 128 i 64 neuronami, a następnie warstwę wyjściową z liczbą neuronów równą liczbie klas (np. 3, jeśli klasyfikujemy psy, koty i ptaki).

Sieci splotowe zrewolucjonizowały dziedzinę wizji komputerowej dzięki swojej zdolności do automatycznego ekstraktowania cech i skutecznej klasyfikacji obrazów. Dzięki swojej strukturze hierarchicznej, CNN mogą efektywnie przetwarzać złożone dane obrazowe, co czyni je idealnym narzędziem do rozwiązywania problemów klasyfikacji wizyjnej. Struktura sieci, parametry filtrów oraz sposób przeprowadzania splotów i poolingów są kluczowymi elementami, które wpływają na efektywność i dokładność modeli CNN.

2. **Opis danych wejściowych i ich analiza statystyczna/wizualizacja**

W celu opisania danych, z których korzystaliśmy podczas prowadzenia projektu, konieczne jest najpierw ich wgranie do notatnika:

```
import pandas as pd

# Wczytanie danych
df = pd.read_csv('D:\SI_projekt_pieski\labels.csv')

# Podgląd danych
print(df.head())
```

```
                                 id             breed
0  000bec180eb18c7604dcecc8fe0dba07       boston_bull
1  001513dfcb2ffafc82cccf4d8bbaba97             dingo
2  001cdf01b096e06d78e9e5112d419397          pekinese
3  00214f311d5d2247d5dfe4fe24b2303d          bluetick
4  0021f9ceb3235effd7fcde7f7538ed62  golden_retriever
```

(Wczytanie i przegląd danych)

Po udanym wprowadzeniu danych, możliwe jest przeprowadzenie podstawowej analizy statystycznej:

```python
# Opis kolumn
print(df.info())

# Podstawowe statystyki opisowe
print(df.describe())

# Rozkład liczności poszczególnych ras psów
print(df['breed'].value_counts())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10222 entries, 0 to 10221
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   id      10222 non-null  object
 1   breed   10222 non-null  object
dtypes: object(2)
memory usage: 159.8+ KB
None
                                    id                breed
count                            10222                10222
unique                           10222                  120
top     000bec180eb18c7604dcecc8fe0dba07  scottish_deerhound
freq                                 1                  126
breed
scottish_deerhound      126
maltese_dog             117
afghan_hound            116
entlebucher             115
bernese_mountain_dog    114
                       ...
golden_retriever         67
brabancon_griffon        67
komondor                 67
eskimo_dog               66
briard                   66
Name: count, Length: 120, dtype: int64
```

Z analizy statystycznej wynika, że zbiór posiada dwie kolumny: id, breed (id, rasa). Każda z kolumn posiada 10222 wartości. Kolumna id posiada wszystkie wartości unikatowe, natomiast kolumna breed posiada 120 unikatowych wartości. Rozkład liczebności mówi nam ile unikatowych id przypada na poszczególna rase psa np. Scottish_deerhound ma przypisane 126 unikatowych id i jest to rasa o największej ilości przypisanych id. Rasy Briard i Eskimo_dog posiadają za to już tylko po 66 przypisanych unikatowych id co stanowi prawie połowę liczby id dla rasy Scottish_deerhound i może wpłynąć na proces nauki rozpoznawania akurat tych ras przez zaprojektowany model.

Część dalsza analizy statystycznej:

```
# Liczba unikalnych ras
num_breeds = df['breed'].nunique()
print(f'Liczba unikalnych ras: {num_breeds}')

# Średnia liczba zdjęć na rasę
mean_images_per_breed = df['breed'].value_counts().mean()
print(f'Średnia liczba zdjęć na rasę: {mean_images_per_breed:.2f}')
```

```
Liczba unikalnych ras: 120
Średnia liczba zdjęć na rasę: 85.18
```

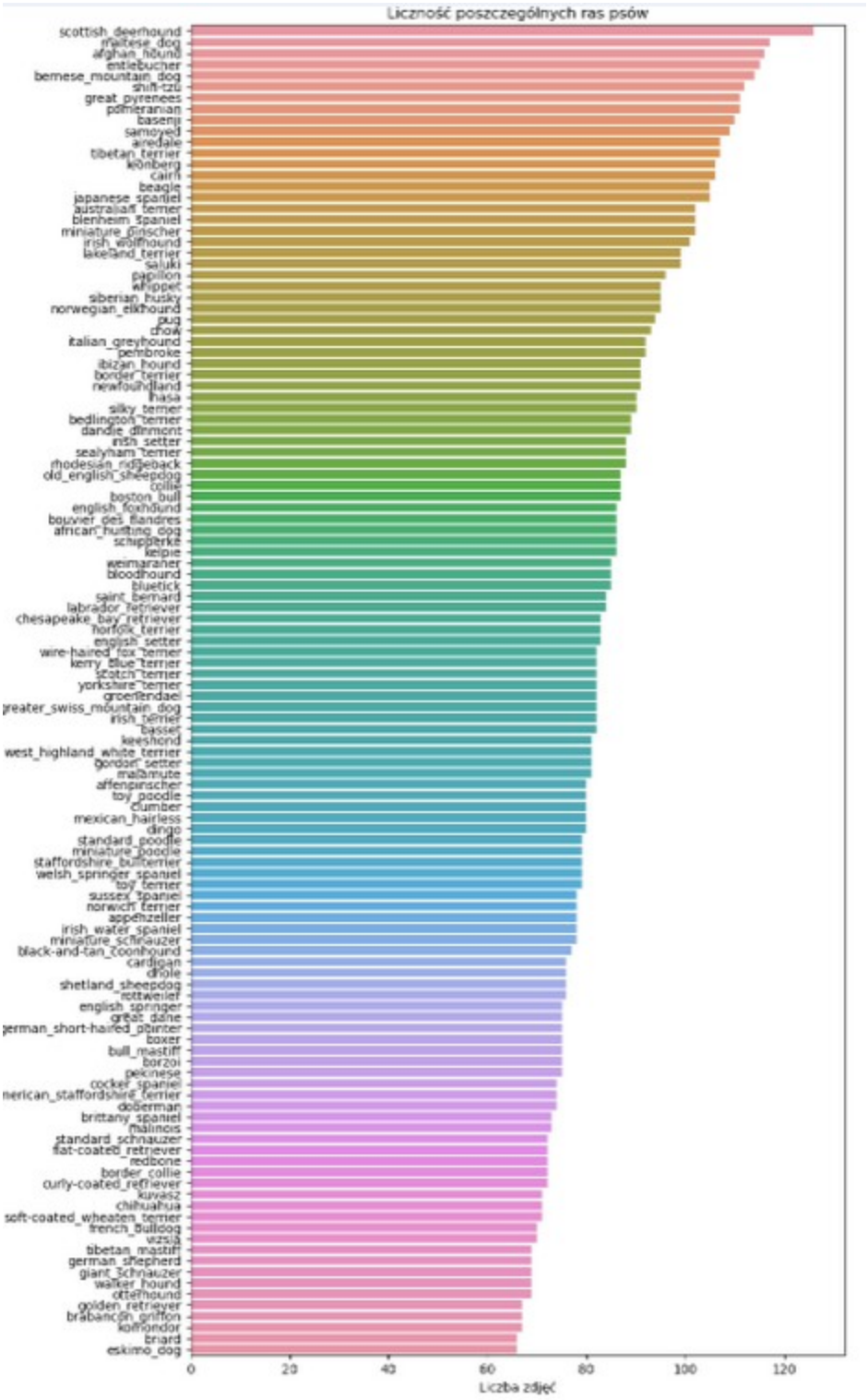
(Średnia liczba zdjęć na rasę)

Dzięki temu obliczeniu wiemy ile statystycznie wynosi średnia ilość zdjęć przypisanych do konkretnej rasy psów.

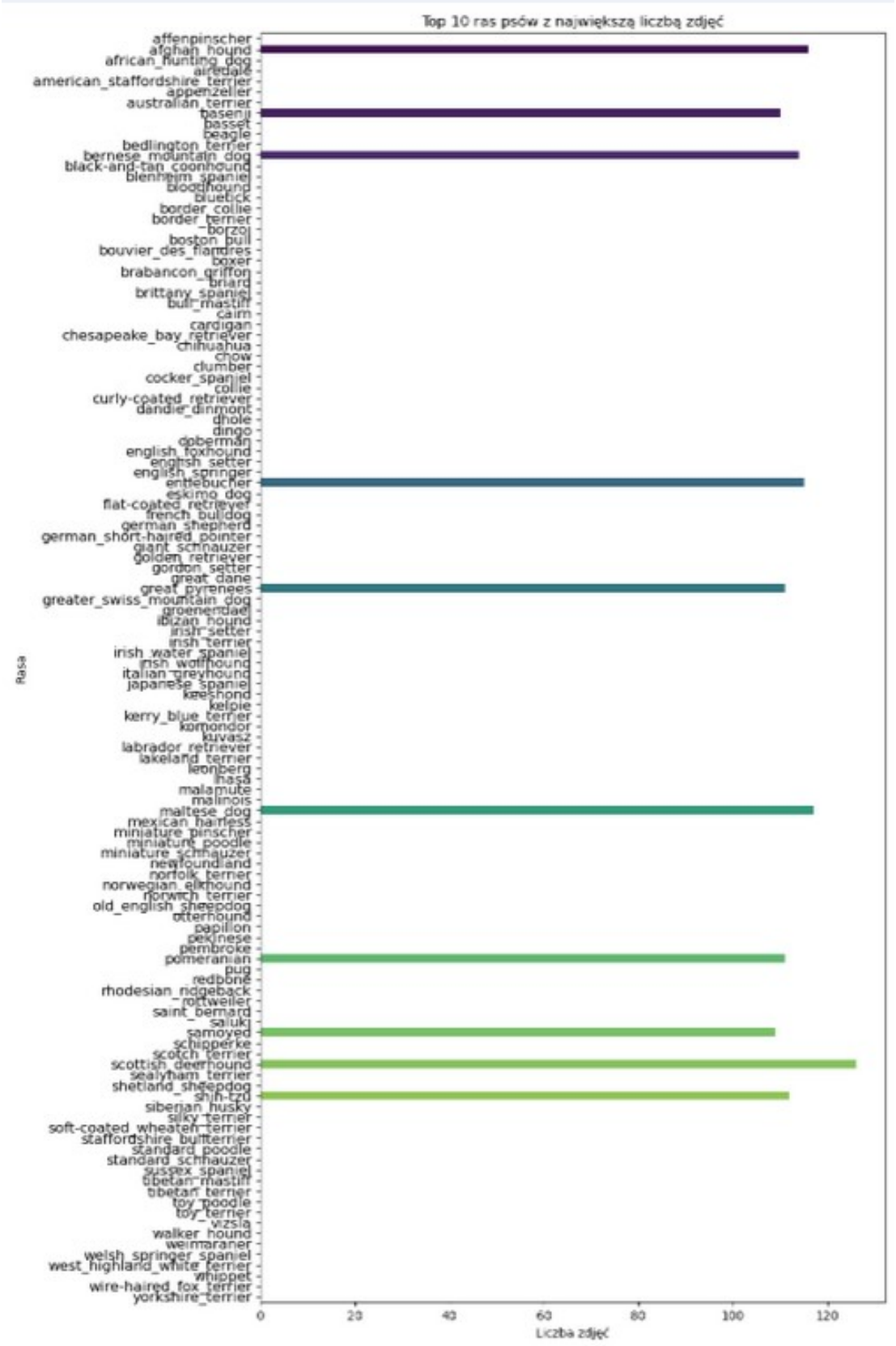
Wizualizacja danych pomoże nam lepiej zrozumieć rozkład ras oraz inne istotne cechy. Wizualizacja danych na wykresach:

```
import matplotlib.pyplot as plt
import seaborn as sns

# Konwersja kolumny breed do typu kategorycznego
df['breed'] = df['breed'].astype('category')

# Histogram liczności poszczególnych ras
plt.figure(figsize=(12, 19))
sns.countplot(y='breed', data=df, order=df['breed'].value_counts().index)
plt.title('Liczność poszczególnych ras psów')
plt.xlabel('Liczba zdjęć')
plt.ylabel('Rasa')
plt.yticks(fontsize=10)
plt.subplots_adjust(left=0.3)
plt.show()

# Histogram liczby zdjęć dla każdej rasy (top 10)
top_breeds = df['breed'].value_counts().nlargest(10)
plt.figure(figsize=(12, 19))
sns.barplot(x=top_breeds.values, y=top_breeds.index, palette='viridis')
plt.title('Top 10 ras psów z największą liczbą zdjęć')
plt.xlabel('Liczba zdjęć')
plt.ylabel('Rasa')
plt.yticks(fontsize=12)
plt.subplots_adjust(left=0.3)
plt.show()

# Histogram liczby zdjęć dla każdej rasy (bottom 10)
bottom_breeds = df['breed'].value_counts().nsmallest(10)
plt.figure(figsize=(12, 19))
sns.barplot(x=bottom_breeds.values, y=bottom_breeds.index, palette='viridis')
plt.title('Top 10 ras psów z najmniejszą liczbą zdjęć')
plt.xlabel('Liczba zdjęć')
plt.ylabel('Rasa')
plt.yticks(fontsize=12)
plt.subplots_adjust(left=0.3)
plt.show()
```

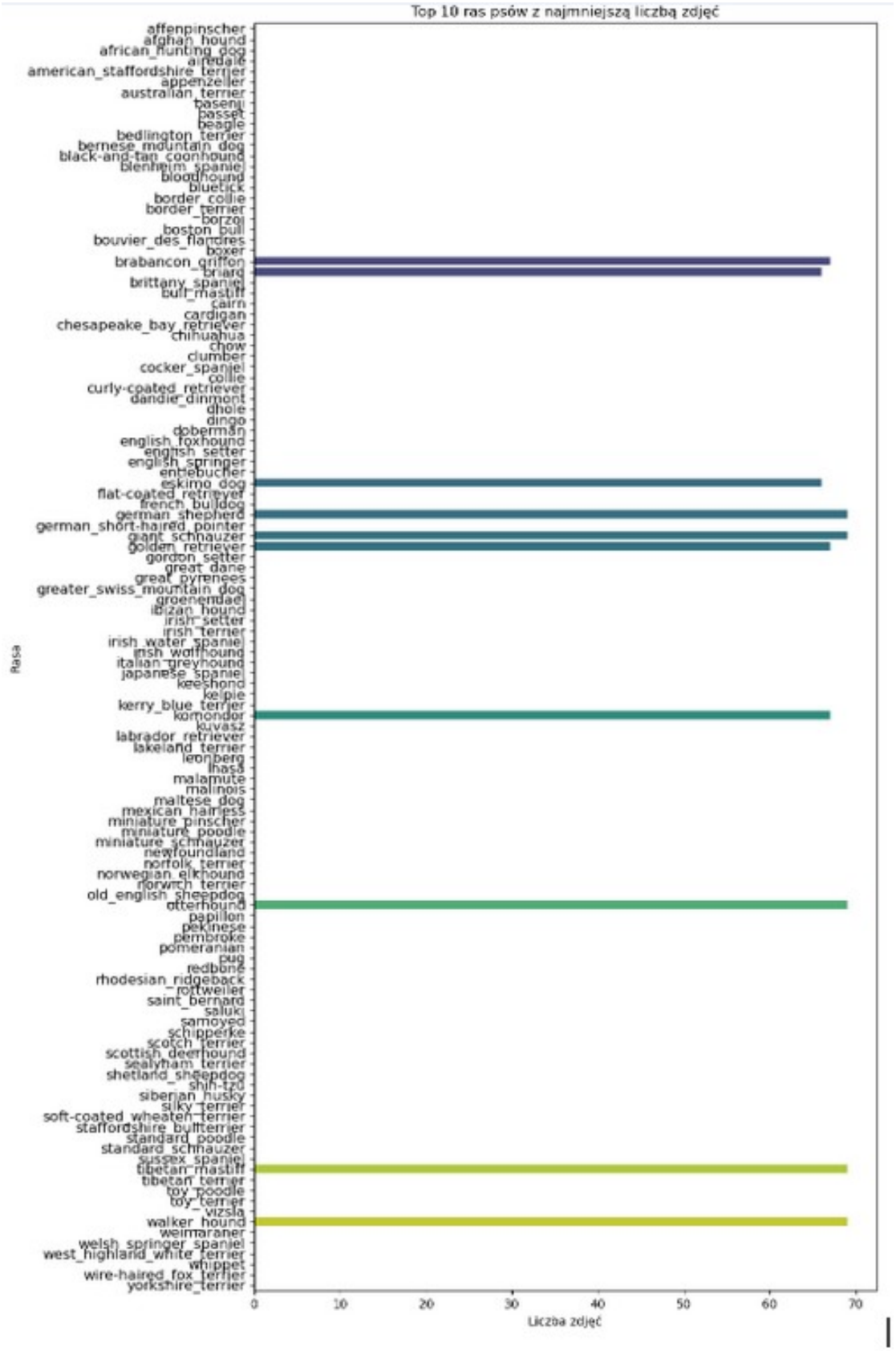
(Histogramy liczebności ras, liczby zdjęc dla każdej rasy: top 10 i bottom 10)



(Liczebność poszczególnych ras psów w datasetcie)



(Top 10 ras psów z największą liczbą zdjęć)



(Top 10 ras psów z najmniejszą liczbą zdjęć)

Podsumowanie:

```
print(f'Liczba unikalnych ras: {num_breeds}')
print(f'Średnia liczba zdjęć na rasę: {mean_images_per_breed:.2f}')
print(f'Top 10 ras z największą liczbą zdjęć:\n{top_breeds}')
print(f'Top 10 ras z najmniejszą liczbą zdjęć:\n{bottom_breeds}')
```

```
Liczba unikalnych ras: 120
Średnia liczba zdjęć na rasę: 85.18
Top 10 ras z największą liczbą zdjęć:
breed
scottish_deerhound      126
maltese_dog             117
afghan_hound            116
entlebucher             115
bernese_mountain_dog    114
shih-tzu                112
great_pyrenees          111
pomeranian              111
basenji                 110
samoyed                 109
Name: count, dtype: int64
Top 10 ras z najmniejszą liczbą zdjęć:
breed
briard                66
eskimo_dog            66
golden_retriever      67
brabancon_griffon     67
komondor              67
tibetan_mastiff       69
german_shepherd       69
giant_schnauzer       69
walker_hound          69
otterhound            69
Name: count, dtype: int64
```

(Podsumowanie analizy danych wejściowych)

Dzięki powyższym krokom uzyskaliśmy kompleksowy opis danych wejściowych, przeprowadziliśmy podstawową analizę statystyczną i zwizualizowaliśmy kluczowe cechy naszego zbioru danych. Taka analiza jest kluczowym krokiem przed przystąpieniem do budowania modeli predykcyjnych.

1. Liczba unikalnych ras: Pozwala określić różnorodność w zbiorze danych.
2. Średnia liczba zdjęć na rasę: Informuje o równomierności reprezentacji ras.
3. Histogram liczności poszczególnych ras: Wizualizuje, które rasy są najczęściej reprezentowane.
4. Histogram liczności poszczególnych ras: Wizualizuje, które rasy są najrzadziej reprezentowane w zbiorze danych.

3. Normalizacja i standaryzacja danych

Zastosowaliśmy transformacje danych poprzez normalizacje:

Normalizacja Danych

Normalizacja obrazów jest kluczowym krokiem w przetwarzaniu wstępnym danych przed treningiem modelu uczenia maszynowego. Celem normalizacji jest przekształcenie wartości pikseli obrazu tak, aby miały określoną średnią (mean) i odchylenie standardowe (std). Jest to pomocne, ponieważ stabilizuje proces uczenia się i sprawia, że optymalizacja staje się bardziej efektywna.

Definicja Normalizacji:

```
# Definicja transformacji normalizującej
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
```

Transformacja transforms.Normalize przyjmuje dwie listy wartości:

- mean (średnie): [0.485, 0.456, 0.406]
- std (odchylenia standardowe): [0.229, 0.224, 0.225]

Te wartości zostały wybrane na podstawie średnich i odchyleń standardowych obrazów w zbiorze danych ImageNet, który jest często używany jako baza danych w zadaniach związanych z rozpoznawaniem obrazów. Zakładając, że obrazy mają trzy kanały (RGB), wartości te są stosowane osobno do każdego kanału.

Składniki transformacji:

- transforms.RandomAffine() - losowo obraca obraz w zakresie od -30 do 30 stopni i przesuwa go w poziomie i pionie do 20% szerokości/wysokości obrazu. Zastosowanie tej techniki ma zalety, takie jak: sztuczne zwiększenie różnorodności zbioru danych treningowych, redukcja przeuczenia itd.

- Transforms.RandomHorizontalFlip() - losowo obraca obraz w poziomie z prawdopodobieństwem 0.5.

- transforms.Resize((IMAGE_HEIGHT, IMAGE_WIDTH)) – zmienia rozmiar obrazu do wymiarów 128x128 pikseli.

- Transforms.ToTensor() - Konwertuje obraz do tensora PyTorch.

- Normalize – stosuje wcześniej opisaną normalizace.

```
training_transform = transforms.Compose([
    transforms.RandomAffine(degrees=(-30, 30), translate=(0.0, 0.2)),
    transforms.RandomHorizontalFlip(),
    transforms.Resize((IMAGE_HEIGHT, IMAGE_WIDTH)),
    transforms.ToTensor(),
    normalize,
])
```

Transformacje Danych Walidacyjnych/Testowych:

Podobnie jak poprzednio najpierw skaluje do rozmiaru 128x128, a następnie konwertuje na tensor:

```
# Definicja transformacji danych walidacyjnych/testowych
testing_transform = transforms.Compose([
    transforms.Resize((IMAGE_HEIGHT, IMAGE_WIDTH)),
    transforms.ToTensor(),
    normalize,
])
```

Przygotowanie danych do trenowania:

- Losowo dzielimy nasz zbiór danych na zbiór treningowy i uczący w stosunku 80-20. Zapisujemy dane jako obiekty naszej klasy DogDataset.

```
# Ustawienie ziarna generatora liczb losowych dla powtarzalności
np.random.seed(42)
# Stworzenie maski dla podziału 80-20 na zbiór treningowy i walidacyjny
msk = np.random.rand(len(data_df)) < 0.8

# Podział DataFrame na zbiór treningowy i walidacyjny
train_df = data_df[msk].reset_index()
val_df = data_df[~msk].reset_index()

# Stworzenie zbiorów danych treningowych i walidacyjnych
train_ds = DogDataset(train_df, TRAIN_DIR, transform=training_transform)
val_ds = DogDataset(val_df, TRAIN_DIR, transform=testing_transform)
```

Wydrukowanie liczby dostępnych kart graficznych i ich pamięci:

Następnym krokiem jest upewnienie się czy nasz program korzysta z wszystkich dostępnych kart graficznych i czy pamięć tych kart jest całkowicie wykorzystywana. Oczywiste jest, że im więcej kart graficznych i pamięci tym większa jest potencjalna wydajność trenowania naszego modelu:

```python
# Wydrukowanie liczby dostępnych GPU i ich pamięci
for i in range(torch.cuda.device_count()):
    print(f"GPU {i}: {torch.cuda.get_device_properties(i).total_memory / (1024 ** 3):.2f} GB")
```

Załadowanie danych do naszych zbiorów danych za pomocą klasy DataLoader:

```python
BATCH_SIZE = 128
# Stworzenie DataLoaderów dla zbiorów danych treningowych i walidacyjnych
train_dataloader = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=True, drop_last=True, num_workers=0, pin_memory=True)
val_dataloader = DataLoader(val_ds, batch_size=BATCH_SIZE, shuffle=True, drop_last=True, num_workers=0,  pin_memory=True)
```

Klasa DataLoader w bibliotece PyTorch jest używana do ładowania danych w modelu uczenia maszynowego. W powyższym kodzie, DataLoader jest wykorzystywany do przygotowania danych treningowych i walidacyjnych, które będą używane do trenowania i walidacji modelu.

Oto szczegóły dotyczące używanych argumentów:

1. train_ds i val_ds: Są to obiekty datasetów, które zawierają dane treningowe i walidacyjne odpowiednio.

2. batch_size: Określa rozmiar batcha, czyli liczbę próbek, które będą przetwarzane jednocześnie przez model podczas jednej iteracji uczenia.

3. shuffle: Określa, czy dane mają być przetasowane przed podziałem na batche. Jest to ważne, aby uniknąć uczenia modelu na kolejności danych, co może prowadzić do wpadania w lokalne minimum.

4. drop_last: Określa, czy należy odrzucić ostatni niekompletny batch, jeśli rozmiar zbioru danych nie jest podzielny przez rozmiar batcha. Jest to często stosowane, aby zapewnić, że wszystkie batche mają taki sam rozmiar.

5. num_workers: Określa liczbę wątków używanych do wczytywania danych. Ustawienie tej wartości na wartość większą niż zero pozwala na wczytywanie danych w tle, co może przyspieszyć proces wczytywania, zwłaszcza dla dużych zbiorów danych.

6. pin_memory: Określa, czy tensory danych powinny być przechowywane w pamięci przypiętej na GPU. Ustawienie tej wartości na True może przyspieszyć transfer danych z pamięci hosta do pamięci urządzenia GPU.

DataLoader jest używany w powyższym kodzie do przygotowania danych treningowych i walidacyjnych w formie, która jest gotowa do przekazania do modelu uczenia maszynowego, z uwzględnieniem podziału na batche, tasowania danych i ewentualnego odrzucenia ostatnich niekompletnych batchy.

Sprawdzenie partii danych (batch):

```
]: # Pobranie partii danych treningowych i wydrukowanie kształtów obrazów i etykiet
(image_batch, label_batch) = next(iter(train_dataloader))
print(image_batch.shape)
print(label_batch.shape)
```

```
torch.Size([128, 3, 128, 128])
torch.Size([128])
```

Termin "batch" odnosi się do zbioru próbek danych, które są przetwarzane jednocześnie przez model w jednej iteracji uczenia. W kontekście uczenia maszynowego, dane są zazwyczaj podzielone na mniejsze grupy nazywane batchami, aby umożliwić efektywne przetwarzanie danych przez model.

W powyższym kodzie, train_dataloader jest obiektem klasy DataLoader, który wczytuje dane treningowe. Wywołanie funkcji next(iter(train_dataloader)) służy do pobrania jednej partii danych treningowych z train_dataloader. Ta partia danych jest następnie przypisana do zmiennych image_batch i label_batch.

Następnie wyświetlane są kształty (shape) obrazów (image_batch) i etykiet (label_batch). Kształt obrazów (image_batch.shape) informuje nas o wymiarach obrazów w partii, np. (batch_size, channels, height, width), gdzie batch_size oznacza liczbę obrazów w partii, channels to liczba kanałów obrazu (np. 3 dla obrazów kolorowych RGB), a height i width to odpowiednio wysokość i szerokość obrazu.

Podobnie, kształt etykiet (label_batch.shape) informuje nas o wymiarach etykiet w partii. Dla typowego problemu klasyfikacji, kształt etykiet zazwyczaj wynosi (batch_size,), gdzie batch_size oznacza liczbę etykiet w partii.

Definicja modelu sieci neuronowej:

Model sieci neuronowej zdefiniowaliśmy jako klasę z metodami:

- __init__: Inicjalizuje warstwy sieci neuronowej, w tym warstwy konwolucyjne, batch normalization, ReLU, max pooling oraz warstwy klasyfikacyjne (fully connected).

- forward: Definiuje przepływ danych przez sieć - najpierw przez warstwy konwolucyjne (features), a następnie przez warstwy klasyfikacyjne (classifier).

Składa się z dwóch głównych części:

1. **Część ekstrakcji cech**: Zawiera zestaw warstw konwolucyjnych, normalizacyjnych, funkcji aktywacji ReLU oraz warstw MaxPooling, które wyodrębniają istotne cechy z obrazu.

2. **Część klasyfikacyjna**: Składa się z dwóch warstw w pełni połączonych, które przetwarzają cechy na wyjście klasyfikacji.

Sieć przyjmuje na wejściu obrazy o rozmiarze 3x128x128 (RGB) i przetwarza je w celu uzyskania prawdopodobieństw przynależności do różnych klas.

Kod w Pythonie:

```
class VGG19(nn.Module):
    def __init__(self, num_classes):
        super(VGG19, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 8, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(8),
            nn.ReLU(inplace=True),
            nn.Conv2d(8, 8, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(8),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(8, 16, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(16),
            nn.ReLU(inplace=True),
            nn.Conv2d(16, 16, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(16),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(16, 32, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.Conv2d(32, 32, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(32, 64, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.classifier = nn.Sequential(
            nn.Linear(64 * 8 * 8, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(512, num_classes),
        )
```

Struktura sieci:

1. **Warstwy konwolucyjne (nn.Conv2d):**

- Pierwsza warstwa konwolucyjna: 3 wejścia (RGB), 8 wyjść, jądro 3x3, padding 1, bez obciążenia (bias), bez zmian inplace (ReLU).
- Druga warstwa konwolucyjna: 8 wejść (z poprzedniej warstwy), 8 wyjść, jądro 3x3, padding 1, bez obciążenia (bias), bez zmian inplace (ReLU).
- Trzecia warstwa konwolucyjna: 8 wejść (z poprzedniej warstwy), 16 wyjść, jądro 3x3, padding 1, bez obciążenia (bias), bez zmian inplace (ReLU).
- Czwarta warstwa konwolucyjna: 16 wejść (z poprzedniej warstwy), 16 wyjść, jądro 3x3, padding 1, bez obciążenia (bias), bez zmian inplace (ReLU).
- Piąta warstwa konwolucyjna: 16 wejść (z poprzedniej warstwy), 32 wyjścia, jądro 3x3, padding 1, bez obciążenia (bias), bez zmian inplace (ReLU).
- Szósta warstwa konwolucyjna: 32 wejścia (z poprzedniej warstwy), 32 wyjścia, jądro 3x3, padding 1, bez obciążenia (bias), bez zmian inplace (ReLU).
- Siódma warstwa konwolucyjna: 32 wejścia (z poprzedniej warstwy), 64 wyjścia, jądro 3x3, padding 1, bez obciążenia (bias), bez zmian inplace (ReLU).
- Ósma warstwa konwolucyjna: 64 wejścia (z poprzedniej warstwy), 64 wyjścia, jądro 3x3, padding 1, bez obciążenia (bias), bez zmian inplace (ReLU).

Batch Normalization to technika normalizacji danych wejściowych dla każdej warstwy sieci neuronowej w batchu danych. Podstawowym jej celem jest utrzymanie stabilności i przyspieszenie uczenia poprzez normalizację rozkładu aktywacji w każdej warstwie. W programie używamy tej normalizacji przez funkcję nn.BatchNorm2d().

Funkcja ReLU to jedna z najczęściej stosowanych funkcji aktywacji w sieciach neuronowych. Jej działanie polega na przekazywaniu wartości nieujemnych niezmienionych (przechodzi przez funkcję bez zmiany), a wartości ujemne są ustawiane na zero. Realizowana przez funkcję nn.ReLU().

Redukcja wymiarowości: MaxPooling redukuje wymiarowość obrazu lub cechy poprzez podział na mniejsze obszary i wybór maksymalnej wartości w każdym z nich. Na przykład w warstwie MaxPooling o rozmiarze (2x2) wybierana jest maksymalna wartość spośród każdych czterech sąsiednich pikseli, co prowadzi do zmniejszenia wymiarowości obrazu o połowę w każdym wymiarze. Realizowana przez funkcję nn.MaxPool2d().

2. Warstwy w pełni połączone – część klasyfikacyjna:

- nn.Linear(64 * 8 * 8, 512): Pierwsza warstwa to warstwa liniowa, która wykonuje operację transformacji liniowej danych wejściowych o wymiarze 64 * 8 * 8 (wynik z poprzedniej warstwy konwolucyjnej) na wymiar 512.

- nn.BatchNorm1d(512): Warstwa normalizacji wsadowej jest stosowana po warstwie liniowej w celu przyspieszenia uczenia się sieci oraz poprawy jej stabilności.

- nn.ReLU(inplace=True): Funkcja aktywacji ReLU jest stosowana po warstwie normalizacji wsadowej. Parametr inplace=True oznacza, że ReLU jest stosowane bezpośrednio do danych wejściowych, co zmniejsza zużycie pamięci.

- nn.Dropout(0.5): Warstwa Dropout jest stosowana w celu regularyzacji modelu poprzez losowe wyłączanie połowy neuronów w warstwie w trakcie treningu, co pomaga w zapobieganiu przeuczeniu.

- nn.Linear(512, num_classes): Ostatnia warstwa liniowa przekształca dane z wymiaru 512 na wymiar odpowiadający liczbie klas w problemie klasyfikacji (parametr num_classes). Ta warstwa generuje końcowe wyniki klasyfikacji.

Metoda forward klasy sieci neuronowej:

```
def forward(self, x):
    x = self.features(x)
    x = x.view(x.size(0), -1)
    x = self.classifier(x)
    return x
```

Metoda forward w klasie My_NeuralNetwork odpowiada za przepływ danych przez sieć neuronową. Jest to kluczowa metoda, która definiuje sposób, w jaki dane wejściowe są przetwarzane przez warstwy sieci, a następnie generowane są końcowe wyniki.

1. self.features(x): Dane wejściowe x są przekazywane przez sekwencję warstw konwolucyjnych i warstw normalizacji wsadowej zawartych w atrybucie features.

2. x = x.view(x.size(0), -1): Wynik z warstw konwolucyjnych jest przekształcany z tensora wielowymiarowego na jednowymiarowy za pomocą metody view, aby mógł zostać przekazany do warstw w pełni połączonych.

3. x = self.classifier(x): Przetworzony jednowymiarowy tensor x jest przekazywany przez sekwencję warstw w pełni połączonych zawartych w atrybucie classifier, która generuje końcowe wyniki klasyfikacji.

4. return x: Metoda zwraca wyjściowy tensor x, który zawiera wyniki klasyfikacji uzyskane przez sieć neuronową.

Inicjalizacja modelu i ustawienia treningowe:

11. Inicjalizacja modelu i ustawienia treningowe:

- Inicjalizacja modelu: Tworzy instancję modelu VGG19 z 120 klasami (num_classes).
- Konfiguracja urządzenia: Ustawia model na GPU, jeśli jest dostępne.
- Funkcja straty i optymalizator: Definiuje funkcję straty jako CrossEntropyLoss i optymalizator jako Adam.
- AMP Scaler: Inicjalizuje skaler do automatycznego mieszania precyzji.

```
import torch.optim as optim
NUM_CLASSES = 120
model = VGG19(NUM_CLASSES)

# Ustawienie modelu na GPU, jeśli jest dostępnetorch.backends.cudnn.benchmark = True
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model.to(device)
# Definicja funkcji straty i optymalizatora
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Inicjalizacja skaleru AMP (Automatic Mixed Precision)
scaler = torch.cuda.amp.GradScaler()
```

- **CrossEntropyLoss()**: Jest to funkcja kosztu, która jest często stosowana w zadaniach klasyfikacji wieloklasowej. Oblicza ona stratę między przewidywaniami modelu a rzeczywistymi etykietami, wykorzystując technikę entropii krzyżowej. W skrócie, im mniejsza wartość straty, tym lepiej model radzi sobie z klasyfikacją.

- **optim.Adam()**: Jest to algorytm optymalizacji gradientowej używany do aktualizacji wag modelu w procesie uczenia się. Algorytm Adam jest ulepszeniem popularnego algorytmu SGD (Stochastic Gradient Descent). Działa on przez adaptacyjne dostosowywanie współczynnika uczenia się dla każdego parametru na podstawie średnich pierwszego i drugiego momentu gradientu.

- **Współczynnik uczenia (learning rate)**: Jest to parametr, który kontroluje, jak bardzo wagi modelu są aktualizowane w każdym kroku podczas procesu uczenia się. Wartość współczynnika uczenia się decyduje o tym, jak duże są kroki, jakie podejmuje algorytm optymalizacji w kierunku minimalizacji funkcji kosztu. Odpowiedni wybór współczynnika uczenia się jest istotny dla efektywnego uczenia modelu. Zbyt mały współczynnik uczenia się

może prowadzić do wolnego postępu lub utknięcia w minimum lokalnym, podczas gdy zbyt duży może prowadzić do oscylacji lub niezbieżności procesu uczenia się.

- **torch.cuda.amp.GradScaler()** jest narzędziem służącym do skalowania gradientów podczas szkolenia modeli w PyTorch, szczególnie przy użyciu funkcji autokastowania (autocast). Pomaga to zapobiegać problemom z niedokładnością numeryczną i przepełnieniem przy obliczaniu gradientów, szczególnie w przypadku modeli o dużych wagach lub przy niskiej precyzji numerycznej. Skalowanie gradientów pozwala na zachowanie stabilności procesu uczenia się i poprawę jego efektywności.

Funkcja zapisu stanu modelu:

Służy do zapisania stanu modelu, optymalizatora, skalera AMP oraz dodatkowych informacje, takich jak numer epoki i wartość straty, do pliku o nazwie filename.

```python
def save_checkpoint(model, optimizer, epoch, loss, filename, scaler):
    checkpoint = {
        'epoch': epoch,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'scaler_state_dict': scaler.state_dict(),
        'loss': loss,
    }
    torch.save(checkpoint, filename)
```

Definicja funkcji do trenowania modelu:

```python
from  torch.cuda.amp import autocast

model_path = '/home/mikolajmichalczyk/Documents/SI/projekt/model_checkpoint.pth'

def train_model(model, criterion, optimizer, num_epochs=10):
    model.to(device)
    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0
        progress_bar = tqdm(enumerate(train_dataloader), total=len(train_dataloader), desc=f'Epoch {start_epoc

        for i, (images, labels) in progress_bar:
            images = images.to(device)  # Przeniesienie obrazów na odpowiednie urządzenie
            labels = labels.to(device)  # Przeniesienie etykiet na odpowiednie urządzenie
            optimizer.zero_grad(set_to_none=True)  # Wyzerowanie gradientów optymalizatora
            with autocast():  # Użycie automatycznego skalowania mieszanej precyzji
                outputs = model(images)  # Obliczenie wyjść modelu
                loss = criterion(outputs, labels)  # Obliczenie straty

            # Obliczenie gradientów i aktualizacja wag modelu przy użyciu skalowania AMP
            scaler.scale(loss).backward()
            scaler.step(optimizer)
            scaler.update()
```

```
        running_loss += loss.item()  # Aktualizacja skumulowanej straty
        _, predicted = torch.max(outputs, 1)  # Pobranie predykcji
        total += labels.size(0)  # Aktualizacja całkowitej liczby próbek
        correct += (predicted == labels).sum().item()  # Aktualizacja liczby poprawnych predykcji
        # Aktualizacja paska postępu z bieżącą stratą i dokładnością
        progress_bar.set_postfix(loss=running_loss / (i + 1), accuracy=correct / total)

    epoch_loss = running_loss / len(train_dataloader)
    epoch_acc = correct / total
    print(f'Epoch {epoch + start_epoch}/{num_epochs+start_epoch}, Loss: {epoch_loss:.4f}, Accuracy: {epoch
    # zapisanie modelu
    save_checkpoint(model, optimizer, epoch = epoch, loss = epoch_loss, filename=model_path, scaler=scale
```

Oto kroki podjęte w funkcji do trenowania modelu sieci neuronowej:

- Przeniesienie modelu na odpowiednie urządzenie (CPU lub GPU).
- Iteracja przez każdą epokę treningową.
- Ustawienie modelu w tryb treningu. - model.train()
- Inicjacja liczników dla bieżącej straty, liczby poprawnych predykcji i całkowitej liczby próbek.
- Iteracja przez dane treningowe za pomocą paska postępu.
- Przeniesienie obrazów i etykiet na odpowiednie urządzenie.
- Zerowanie gradientów optymalizatora.
- Użycie automatycznego skalowania mieszanej precyzji (autocast) do obliczenia straty.
- Obliczenie predykcji modelu i straty.
- Obliczenie gradientów i aktualizacja wag modelu przy użyciu skalowania AMP.
- Aktualizacja skumulowanej straty, liczby poprawnych predykcji i całkowitej liczby próbek.
- Wyświetlenie bieżącej straty i dokładności na pasku postępu.
- Zapisanie modelu po zakończeniu epoki.

Funkcja do załadowania stanu modelu z pliku:

```
def load_checkpoint(model, optimizer, filename, scaler):
    checkpoint = torch.load(filename)
    model.load_state_dict(checkpoint['model_state_dict'])
    model.to(device)
    optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    scaler.load_state_dict(checkpoint['scaler_state_dict'])
    epoch = checkpoint['epoch']
    loss = checkpoint['loss']
    return model, optimizer, epoch, loss, scaler
```

Działanie funkcji:

- Funkcja load_checkpoint ładuje stan modelu, optymalizatora, skalera AMP, numer epoki oraz wartość straty z pliku checkpoint.
- Przenosi model na odpowiednie urządzenie po załadowaniu stanu.
- Zwraca model, optymalizator, epokę, stratę i skaler AMP.

Sprawdzenie, czy plik ze stanem modelu nie jest pusty:

```
if os.path.getsize(model_path) > 0:
    model, optimizer, start_epoch, loss, scaler = load_checkpoint(model, optimizer, model_path, scaler)
else:
    start_epoch = 0
    loss = None
```

Działanie kodu:

- Używa funkcji os.path.getsize, aby sprawdzić, czy plik checkpoint nie jest pusty.
- Jeśli plik nie jest pusty, ładuje stan modelu, optymalizatora i skalera AMP.
- Jeśli plik jest pusty, ustawia start_epoch na 0 i loss na None.

Trenowanie modelu i pomiar czasu:

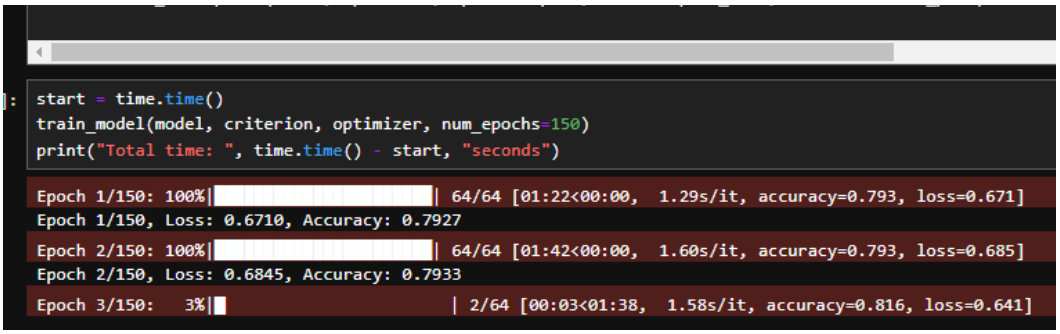
```
start = time.time()
train_model(model, criterion, optimizer, num_epochs=2)
print("Total time: ", time.time() - start, "seconds")
```

```
Epoch 10/11: 100%|████████████████████| 64/64 [03:03<00:00,  2.87s/it, accuracy=0.217, loss=3.19]
Epoch 9/11, Loss: 3.1864, Accuracy: 0.2174
Epoch 11/11: 100%|████████████████████| 64/64 [03:07<00:00,  2.94s/it, accuracy=0.225, loss=3.13]
Epoch 10/11, Loss: 3.1346, Accuracy: 0.2251
Total time:  371.682147026062 seconds
```

Poprzez wywołanie metody train_model() program wykonuje trenowanie sieci neuronowej. W trakcie trenowania w notatniku wyświetla się pasek informujący o numerze epoki w stosunku do wszystkich planowanych do wykonania epok, procencie wykonania epoki, aktualnej dokładności modelu (accuracy), koszcie(loss), czasie wykonania, iteracji na sekundę. Funkcja wykonuje się do czasu wykonania ostatniej założonej epoki. Na końcu wyświetla się ostateczna dokładność i koszt straty modelu wraz z całkowitym czasem trenowania.

Powyższy zrzut ekranu wykonywany był na laptopie bez dostępu do karty graficznej, przez co korzystał z procesora. Możemy porównać średni czas trenowania epoki laptopa z komputerem korzystającym z karty graficznej do trenowania. Na procesorze epoka wykonuje się mniej więcej od 3 do 4 minut w zależności od przeciążenia procesora.

Zrzut ekranu z komputera z kartą graficzną o pamięci 4GB:



Możemy zauważyć, że czas wykonania epoki to już od 1:20 min do 1:42 min, zatem conajmniej 2x krótszy. Wniosek jest taki, że trenowanie modeli sztucznej inteligencji zachodzi znacznie szybciej na kartach graficznych.

W naszym przypadku udało się wytrenować model do średniej skuteczności około 80% predykcji wykorzystując kartę graficzną. Czas jaki potrzebowaliśmy to około 7-8 godzin.

Definicja funkcji ewaluacji modelu:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
def test_model(model, criterion, test_dataloader, device):
    model.eval()  # Set the model to evaluation mode
    model.to(device)
    correct = 0
    total = 0
    # Pobranie jednego batcha danych z dataloadera walidacyjnego
    images, labels = next(iter(test_dataloader))
    images = images.to(device)  # Przeniesienie obrazów na odpowiednie urządzenie
    labels = labels.to(device)  # Przeniesienie etykiet na odpowiednie urządzenie

    with torch.no_grad():  # Wyłączenie obliczeń gradientów w celu przyspieszenia procesu ewaluacji
        for images, labels in test_dataloader:
            images = images.to(device)  # Przeniesienie obrazów na odpowiednie urządzenie
            labels = labels.to(device)  # Przeniesienie etykiet na odpowiednie urządzenie

            # Przekazanie danych przez model (forward pass)
            outputs = model(images)

            # Obliczenie metryk
            _, predicted = torch.max(outputs, 1)  # Pobranie predykcji
            total += labels.size(0)  # Aktualizacja całkowitej liczby próbek
            correct += (predicted == labels).sum().item()  # Aktualizacja liczby poprawnych predykcji

    # Obliczenie dokładności na zbiorze walidacyjnym
    accuracy = correct / total
    print(f'Test Accuracy: {accuracy:.4f}')  # Wyświetlenie dokładności
```

Wywołanie metody:

```
test_model(model, criterion, val_dataloader, device)
```

```
Test Accuracy: 0.2531
```

Zastosowanie funkcji polega na ocenie dokładności predykcji modelu na podstawie ewaluacji z udziałem zbioru testowego.

Działanie funkcji:

Funkcja test_model testuje model na zbiorze walidacyjnym (test_dataloader).

- Ustawia model w tryb ewaluacji (model.eval()), co wyłącza obliczenia gradientów i inne mechanizmy specyficzne dla trybu treningu, takie jak dropout.
- Przenosi model na odpowiednie urządzenie (model.to(device)).
- Inicjalizuje zmienne do śledzenia liczby poprawnych predykcji (correct) i całkowitej liczby próbek (total).
- images, labels = next(iter(test_dataloader)) pobiera jeden batch danych z dataloadera walidacyjnego.
- Przenosi obrazy i etykiety na odpowiednie urządzenie (images.to(device), labels.to(device)).
- with torch.no_grad(): wyłącza obliczenia gradientów, co przyspiesza proces ewaluacji i zmniejsza zużycie pamięci.
- Dla każdej partii danych w dataloaderze walidacyjnym przenosi obrazy i etykiety na odpowiednie urządzenie.
- Przekazuje dane przez model (outputs = model(images)), oblicza predykcje (_, predicted = torch.max(outputs, 1)), aktualizuje całkowitą liczbę próbek (total) i liczbę poprawnych predykcji (correct).
- Oblicza dokładność jako stosunek liczby poprawnych predykcji do całkowitej liczby próbek (accuracy = correct / total).

Funkcja wizualizująca działanie modelu predykcji rasy psów:

Zaimplementowaliśmy funkcję, która obrazuje działanie naszego modelu. Jej działanie polega na wyświetleniu 8 zdjęć psów w 2 rzędach po 4 zdjęcia wraz z ich prawdziwą rasą i przewidywaną rasą przez nasz model. Funkcja przyjmuje jako argumenty model, dataloader testowy, urządzenie oraz listę nazw klas.

```python
import matplotlib.pyplot as plt
import numpy as np
import torch

def show_random_image_prediction(model, test_dataloader, device, class_names):
    """
    Funkcja wybiera losowe obrazy z zestawu testowego, przewiduje ich klasy za pomocą modelu
    i wyświetla te obrazy wraz z informacjami o predykcji.

    Args:
        model (torch.nn.Module): Wytrenowany model sieci neuronowej.
        test_dataloader (torch.utils.data.DataLoader): Dataloader zestawu testowego.
        device (torch.device): Urządzenie (CPU lub GPU) do przeprowadzenia predykcji.
        class_names (list): Lista nazw klas.
    """
    # Ustawienie modelu w tryb ewaluacji
    model.eval()
    model.to(device)  # Przeniesienie modelu na GPU (jeśli dostępne)

    # Pobranie obrazów i etykiet z dataloadera
    images, labels = next(iter(test_dataloader))
    images = images.to(device)
    labels = labels.to(device)

    # Wybranie losowych indeksów obrazów
    indices = np.random.choice(len(images), 8, replace=False)
    selected_images = images[indices]
    selected_labels = labels[indices]

    # Tworzenie figure do wyświetlania obrazów
    fig, axes = plt.subplots(2, 4, figsize=(15, 7))
```

Działanie powyższego kodu funkcji:

- model.eval() wyłącza tryb treningowy modelu.
- images, labels = next(iter(test_dataloader)) pobiera jeden batch danych.
- przenosi obrazy i etykiety na odpowiednie urządzenie.
- wybiera losowe obrazy
- tworzy figure z 2 rzędami i 4 kolumnam

Przewidywanie ras psów na wylosowanych zdjęciach:

```
    with torch.no_grad():
        for i, ax in enumerate(axes.flat):
            # Przekazanie pojedynczego obrazu przez model
            img = selected_images[i].unsqueeze(0)
            output = model(img)
            _, predicted = torch.max(output, 1)
            prediction = predicted.item()

            # Konwersja obrazu do formatu wyświetlanego przez matplotlib
            image = selected_images[i].cpu().numpy().transpose((1, 2, 0))
            label = selected_labels[i].item()

            # Wyświetlenie obrazu wraz z informacją o prawdziwej i przewidzianej klasie
            ax.imshow(image)
            ax.axis('off')
            ax.set_title(f'Prawdziwa: {class_names[label]}\nPrzewidziana: {class_names[prediction]}')

            # Sprawdzenie poprawności predykcji
            if prediction == label:
                ax.set_title(f'Prawdziwa: {class_names[label]}\nPrzewidziana: {class_names[prediction]}', color='green')
            else:
                ax.set_title(f'Prawdziwa: {class_names[label]}\nPrzewidziana: {class_names[prediction]}', color='red')

    # Wyświetlenie wszystkich obrazów na figure
    plt.tight_layout()
    plt.show()

# Przykład użycia:
show_random_image_prediction(model, val_dataloader, device, labels_names)
```

Działanie powyższego kodu:

- Iteruje przez wszystkie osie (miejsca) na wykresie (axes.flat spłaszcza tablicę osi 2x4 do jednej listy). i to indeks, a ax to pojedyncza oś (miejsce na wykresie).
- Pobiera pojedynczy obraz z selected_images i dodaje wymiar na początku (batch size) za pomocą unsqueeze(0), aby kształt obrazu był zgodny z oczekiwaniami modelu (np. z [C, H, W] na [1, C, H, W]).
- przekazuje obraz do modelu i uzyskuje wynik predykcji dla zdjęcia.
- Przenosi obraz na CPU i konwertuje do tablicy numpy. transpose((1, 2, 0)) zmienia kolejność osi z [C, H, W] na [H, W, C], co jest wymagane do poprawnego wyświetlenia obrazu przez matplotlib.
- Pobiera prawdziwą etykietę dla bieżącego obrazu i konweruje na zwykłą liczbę.
- wyświetla obraz na wykresie.
- Wyświetla informacje o wyniku predykcji. Dla prawidłowej predykcji informacja wyświetla się na zielono, a dla nieprawidłowej na czerwono.

Wynik działania funkcji dla modelu o dokładności szacowanej na 25%:

Wniosek: Wynik statystycznie najbardziej prawdopodobny, jeżeli model ma dokładność 25%, ponieważ 2 obrazki na 8 udało się dobrze zklasyfikować co daje nam w tym przypadku skuteczność dokładnie 25% dla tych wylosowanych zdjęć.

Końcowe wnioski:

Najwyższa dokładność do której udało nam się doprowadzić model to 80%, jednak wymagało to od nas dużej mocy obliczeniowej i długiego okresu trenowania. Gdybyśmy mieli więcej czasu na wykonanie projektu przeprowadzilibyśmy więcej eksperymentów z parametrami modelu takimi jak struktura sieci neuronowej lub różne parametry współczynnika uczenia. W przypadku gdybyśmy mieli szybsze karty graficzne proces trenowania również mógłby być szybszy, co doprowadziłoby też do większej dokładności modelu przy takim samym czasie poświęconym na trenowanie go. Innym sposobem na zwiększenie wydajności modelu byłaby eliminacja trudnych do klasyfikowania zdjęć, które przykładowo charakteryzują się znikomą obecnością psa na zdjęciu tak jak w przypadku drugiego zdjęcia w drugim wierszu na powyższym zestawieniu zdjęć.

Kod źródłowy: https://github.com/Promess02/DogsCNN

Źródła:

https://www.kaggle.com/code/nilaychauhan/dog-breed-classification-using-jax-and-flax

https://pytorch.org/docs/stable/index.html

https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html

https://en.wikipedia.org/wiki/Convolutional_neural_network

https://www.analyticsvidhya.com/blog/2018/12/guide-convolutional-neural-network-cnn/

https://www.freecodecamp.org/news/convolutional-neural-network-tutorial-for-beginners/

https://learnopencv.com/understanding-convolutional-neural-networks-cnn/