Politechnika Krakowska
im. Tadeusza Kościuszki

Zastosowanie modelu YOLO w problemie wykrywania obiektów (ang. object detection)

Grupa P2:

Michał Mistarz

Numer albumu: 149122

Hubert Kordula

Numer albumu: 147605

Sztuczna inteligencja

Projekt 2025/2026

Wydział Inżynierii Elektrycznej i Komputerowej
Politechnika Krakowska

Spis treści

1. Wprowadzenie teoretyczne i matematyczne do modelu YOLO

1.1 Problemy detekcji obiektów i rola Sieci Konwolucyjnych

Problem detekcji obiektów stanowi jedno z fundamentalnych zagadnień w dziedzinie widzenia komputerowego. W odróżnieniu od klasyfikacji obrazu, której celem jest przypisanie jednej etykiety do całego obrazu, detekcja obiektów wymaga jednoczesnego rozwiązania dwóch problemów: klasyfikacji („Co to jest?") i lokalizacji („Gdzie to się znajduje?")

1.1.1 Fundament technologiczny: Konwolucyjne Sieci Neuronowe (CNN)

Współczesne systemy detekcji, w tym modele YOLO, opierają się na architekturze Konwolucyjnych Sieci Neuronowych (CNN – Convolutional Neural Network). W przeciwieństwie do klasycznych sieci typu MLP (Multi-Layer Perceptron), które traktują obraz jako spłaszczony wektor pikseli, sieci CNN zachowują przestrzeną strukturę danych wejściowych.

Jak zauważył Ian Goodfellow w swojej fundamentalnej pracy Deep Learning:

„Sieci konwolucyjne to wyspecjalizowany rodzaj sieci neuronowych, które wykorzystują splotu zamiast ogólnego mnożenia macierzy w co najmniej jednej z ich warstw." [1]

Matematycznie operacja splotu dyskretnego dla dwuwymiarowego obrazu I oraz filtra (jądra) K o wymiarach m x n jest definiowana jako:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n).$$

Gdzie:

• S(i, j) – (ang. Source/Signal) – wartość piksela na wyjściowej mapie cech w punkcie o współrzędnych (i, j).

- I – (ang. Input) - obraz wejściowy lub mapa cech z poprzedniej warstwy.

• K – (ang. Kernel) – jądro splotu. Jest to mała macierz, której wagi są uczone przez sieć w procesie treningu. Wykrywa ona wzorce (krawędzie, kształty)

- i, j – współrzędne piksela w obrazie wyjściowym, dla którego obliczamy wartość

- m, n – zmienne iteracyjne sumowania, które pozwalają przesuwać się po wymiarach jądra splotu K. Oznaczają one przesunięcie względem aktualnego punktu (i, j)

- * - operator splotu(konwolucji).

Kluczową cechą CNN jest ich zdolność do automatycznej ekstrakcji cech. W tradycyjnym przetwarzaniu obrazów inżynierowie musieli ręcznie projektować algorytmy wykrywające

[1] Goodfellow Ian, Deep Learning, Cambridge 2016, s.330

krawędzie czy tekstury (np. algorytm Sobela czy deskryptor SIFT). Sieci CNN uczą się tych filtrów samodzielnie w procesie propagacji wstecznej.

Architektura CNN składa się zazwyczaj z naprzemiennych bloków:

1. **Warstwy Konwolucyjne:** Wykrywają lokalne wzorce (krawędzie w pierwszych warstwach, kształty i obiekty w głębszych warstwach). Francois Chollet określa ten proces budowania jako „hierarchię reprezentacji wizualnych" [2], która pozwala komputerom rozumieć złożone obrazy.
2. **Warstwa Pooling (Łączące):** Najczęściej Max Pooling. Służą do redukcji wymiarowości danych (downsampling) oraz wprowadzają niezmienniczość na małe przesunięcia obiektu(ang. tranlation invariance).
3. **Warstwy Fully Connected:** W klasycznych architekturach znajdują się na końcu sieci i pełnią rolę klasyfikatora, agregując cechy zebrane przez wcześniejsze warstwy.

1.1.2 Podejście oparte na regionach

Zanim wprowadzono model YOLO, dominującym paradygmatem w detekcji obiektów było podejście – Region-based Approach. Reprezentantem tej grupy jest rodzina modeli R-CNN (Region-based Convolutional Neural Networks).

Ross Girshick w swojej pracy opisuje R-CNN jako „…prosty i skalowalny algorytm detekcji, który łączy propozycje regionów z cechami CNN"[3]. Mimo wysokiej skuteczności podejście to było jednak kosztowne obliczeniowo i składało się z wielu niezależnych etapów:

1. **Generowanie propozycji:** Algorytm generował tysiące potencjalnych fragmentów obrazu, w których mógł znajdować się obiekt.
2. **Ekstrakcja cech:** Każdy z tych fragmentów był skalowany i przepuszczany przez sieć CNN w celu wyodrębnienia wektora cech.
3. **Klasyfikacja:** Support Vector Machine (SVM) klasyfikowała, co znajduję się w danym regionie.
4. **Regresja ramki:** Osobny model poprawiał współrzędne ramki.

Główną wadą tego podejścia była jego złożoność obliczeniowa. Konieczność wielokrotnego uruchamiania sieci CNN dla tysięcy wycinków tego samego obrazu sprawiała, że detekcja w czasie rzeczywistym była niemożliwa na standardowym sprzęcie. Jak zauważa Aurelien Geron:

„Możemy potraktować detekcję jako problem regresji, w którym model przewiduje współrzędne ramki oraz prawdopodobieństwo klasy bezpośrednio z pikseli obrazu."[4]

To właśnie ta obserwacja doprowadziła do powstania architektur jednostopniowych, takich jak YOLO.

---

[2] Chollet Francois, Deep Learning with Python, Shelter Island 2017, s. 120

[3] Girshick Ross, Rich feature hierarchies for accurate object detection and sematic segmentation, Columbus 2014, s. 580

[4] Geron Aurelien, Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, Sebastopol 2019, s.466

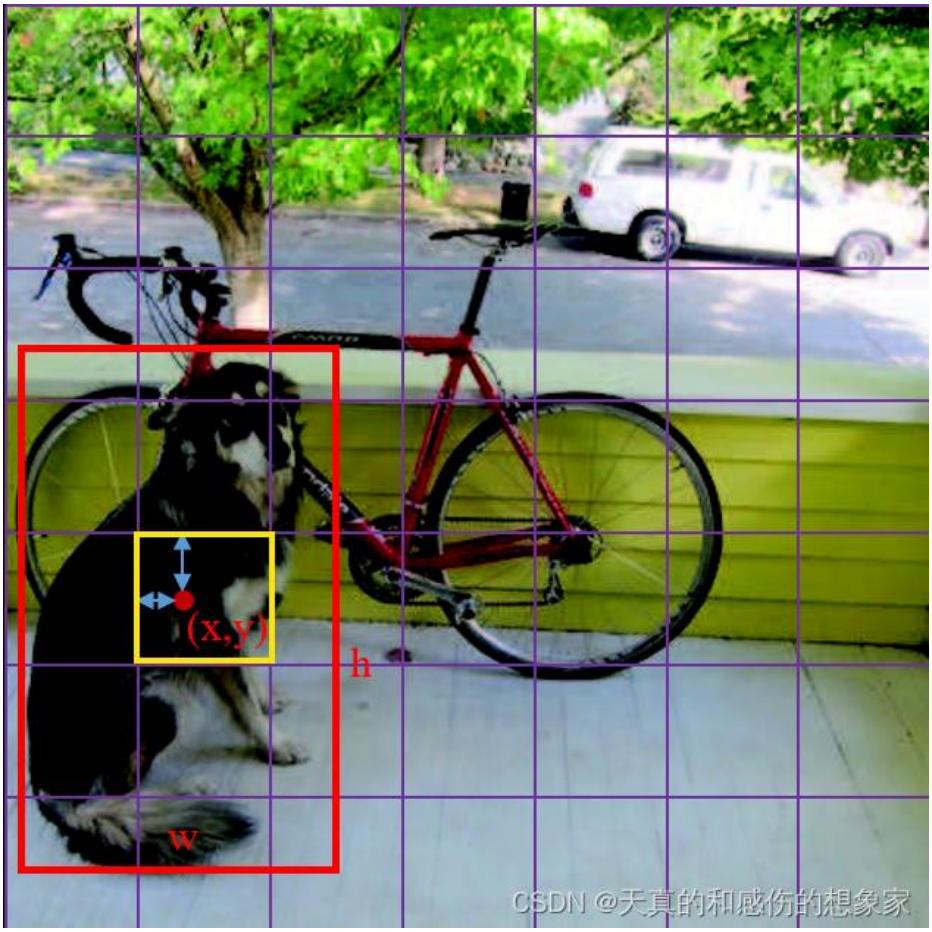
1.2 Architektura sieci i dyskretyzacji przestrzeni obrazu

Fundamentem działania modelu YOLO jest podział obrazu. Jak pisze twórca metody, Joseph Redmon: „Dzielimy obraz wejściowy na siatkę S x S. Jeśli środek obiektu wpada do komórki siatki, to ta komórka jest odpowiedzialna za jego wykrycie”[5], było to odejście od analizy „przesuwanym oknem” na rzecz jednorazowej analizy całego obrazu. Jest to możliwe dzięki specyficznej metodzie dyskretyzacji przestrzeni wejściowej.

1.2.1 Podział na siatkę (Grid System)

Obraz wejściowy X o wymiarach W_img x H_img zostaje podzielony na siatkę S x S komórek. W oryginalnej architekturze Josepha Redmona, przyjęto parametr S = 7.

Mechanizm detekcji opiera się na koncepcji odpowiedzialności komórki. Jeżeli środek rzeczywistego obiektu (ang. Ground Truth) o współrzędnych (x_c, y_c) wpada do wnętrza komórki siatki o indeksach (i, j), to ta konkretna komórka jest odpowiedzialna za jego wykrycie.



[5] Redmon Joseph, You Only Look Once: Unfified, Real-Time Object Detection, Las Vegas 2016, s.779

Jest to istotne ograniczenie architektury YOLO v1: jedna komórka może wykryć tylko jeden obiekt. Oznacza to, że w przypadku wystąpienia dwóch obiektów, których środki leżą w tej samej komórce, model napotka trudność tzw. „Problem bliskich obiektów” (Crowding problem).

1.2.2 Struktura tensora wyjściowego

W wyniku przejścia obrazu przez warstwy konwolucyjne i w pełni połączone, sieć zwraca tensor wyjściowy. W przeciwieństwie do zadań klasyfikacji, gdzie wyjściem jest wektor prawdopodobieństwa, tutaj wyjściem jest trójwymiarowa struktura przestrzenna.

Dla siatki S = 7, liczby ramek na komórkę B = 2 oraz liczby klas C = 20, tensor ma wymiar:

```
# reshaped:     (batch, 7, 7, 30) → grid structure
predictions = predictions.reshape(-1,S,self.split_size,S,self.split_size,C,self.num_classes + B,self.num_boxes * 5)
```

Każdy wektor głębokości (o długości 30) w pozycji (i, j) tensora zawiera:

1. Prawdopodobieństwa klas – P(Class_c | Object) – jest to pierwsze 20 wartości, które określają czym jest obiekt, zakładając, że on tam istnieje.
2. Parametry ramki 1 – (x, y, width, height, conf) – 5 wartości opisujących pierwsze przybliżenie położenia.
3. Parametry ramki 2 – (x, y, width, height, conf) – 5 wartości opisujących drugie przybliżenie położenia.

1.2.3 Transformacja współrzędnych

Kluczowym aspektem implementacyjnym jest przeliczenie współrzędnych środka obiektu z układu globalnego (cały obraz) na układ lokalny (konkretna komórka). Jest to niezbędne dla stabilności uczenia.

Niech (x, y) będą znormalizowanymi współrzędnymi globalnymi obiektu (wartości 0…1).

Krok 1: Identyfikacja indeksu komórki - Należy wyznaczyć, w którym wierszu (i) i kolumnie (j) siatki znajduję się obiekt.

*j = floor(x * S)* *i = floor(y * S)*

```
#--------------------------------
# MATH: GRID POSITION (i, j)
#--------------------------------
# We need to find which 7x7 cell "owns" this object
#
# Example: x = 0.55 (right half), y = 0.55 (bottom half)
# j = fllor(7 * 0.55) = floor(3.85) = 3 (4th column)
# i = fllor(7 * 0.55) = floor(3.85) = 3 (4th row)
i, j = int(self.split_size * y), int(self.split_size * x)
```

Krok 2: Obliczanie współrzędnych lokalnych – Sieć ma przewidywać przesunięcie środka obiektu względem lewego górnego rogu komórki, w której się on znajduje.

*x_cell = (x * S) – j* *y_cell = (y * S) – i*

```
#--------------------------------
# MATH: CELL RELATIVE COORDINATES (x_cell, y_cell)
#--------------------------------
# The model predicts x, y relative to the top_left corner
# of the 'specific cell, NOT the top-left of the whole img
#
# Formula: x_cell = (S * x) - floor(S * x)
# Example: 3.85 - 3 = 0.85 (object is 85% to the right of cell 3)
x_cell, y_cell = self.split_size * x - j, self.split_size * y - i
```

* wartości te zawsze mieszczą się w przedziale [0, 1)

Krok 3: Skalowanie wymiarów ramki – Szerokość i wysokość są skalowane względem wielkości komórki siatki, co pozwala sieci łatwiej uczyć się proporcji obiektów względem siatki.

*w_cell = w * S* *h_cell = h * S*

```
#--------------------------------
# MATH: CELL RELATIVE DIMENSIONS(w_cell, h_cell)
#--------------------------------
# We scale width/height by S so they are comparable to cell units
#
# if w=0.5 (half img), width_cell = 3.5(spans 3.5 cells)
width_cell, height_cell = (
    width * self.split_size,
    height * self.split_size,
)
```

1.2.4 Kodowanie tensora uczącego (Ground Truth)

Ostatnim etapem przygotowania danych jest wpisanie obliczonych wartości do macierzy Target (tensora docelowego). Zgodnie z teorią YOLO, jeśli w komórce (i, j) znajduje się obiekt, ustawiamy wskaźnik obecności obiektu na 1.

$$Target_{i,j} = [\underbrace{x_{cell}, y_{cell}, w_{cell}, h_{cell}}_{\text{box coords}}, \underbrace{1}_{\text{conf}}, \underbrace{0, \ldots, 1, \ldots, 0}_{\text{one-hot class}}]$$

```
#-------------------------------
# ONE OBJECT PER CELL LIMITATION
#-------------------------------
# Index 20 is the "objectness" score (confidence)
#
# if it is 0, the cell is empty, we fill it.
# if it is 1, the cell is already taken by another object, we ignore new one.
if label_matrix[i, j, 20] == 0:
    # set object conficence to 1
    label_matrix[i, j, 20] = 1

    # set box coordinates (x_cell, y_cell, w_cell, h_cell)
    # map to indices 21, 22, 23, 24
    box_coordinates = torch.tensor(
        [x_cell, y_cell, width_cell, height_cell]
    )
    label_matrix[i, j, 21:25] = box_coordinates

    # set class label (one-hot encoding)
    label_matrix[i, j, class_label] = 1
```

Takie podejście do implementacji zapewnia, że model optymalizuje parametry lokalne, co znacznie przyspiesza zbieżność algorytmu w porównaniu do prób bezpośredniej regresji współrzędnych globalnych.

1.3 Matematyczna definicja funkcji straty

Główną częścią uczenia modelu YOLO jest złożona funkcja kosztu (ang. Loss Function), która musi jednocześnie optymalizować błędy lokalizacji ramki, pewności detekcji oraz klasyfikacji. W przeciwieństwie do wielu współczesnych detektorów używających entropii krzyżowej, model wykorzystuje w całości Sumę Błędów Kwadratowych (SSE – Sum Squared Error). Christopher Bishop w swojej analizie metod uczenia maszynowego zaznacza, że „…suma kwadratów błędów jest naturalnym wyborem dla problemów regresji przy założeniu gaussowskiego rozkładu szumu”[6], co uzasadnia jej użycie w predykcji współrzędnych. Wybór ten podyktowany jest łatwością optymalizacji, choć wymaga wprowadzenia wag ważących, aby zrównoważyć poszczególne składowe błędu.

Implementacja funkcji straty znajduję się w klasie YoloLoss w pliku loss.py i realizuje poniższy model matematyczny.

1.3.1 Ważenie składowych funkcji kosztu

Ponieważ większość komórek na obrazie nie zawiera żadnego obiektu (stanowi tło), błąd wynikający z „braku obiektu” mógłby zdominować gradienty, uniemożliwiając zbieżność modelu. Aby temu zapobiec oraz aby nadać priorytet precyzji lokalizacji, wprowadzono dwa hiperparametry lambda.

Lambda_coord = 5 # wysoka waga dla błędu współrzędnych (priorytet lokalizacji)

Lambda_noobj = 0.5 # niska waga dla błędu pewności tła (tłumienie tła)

[6] Bishop Chritopher, Pattern Recognition and Machine Learning, Nowy Jork 2006, s. 140

```python
class YoloLoss(nn.Module):
    def __init__(self, split_size=7, num_boxes=2, num_classes=20):
        super(YoloLoss, self).__init__()
        self.mse = nn.MSELoss(reduction="sum")
        self.split_size = split_size
        self.num_boxes = num_boxes
        self.num_classes = num_classes

        # --- HYPERPARAMETERS ---
        # "background" cells (no object) overwhelm "object" cells
        # We weigh "no object" loss down (0.5) to prevent the models from always prediciting 0
        self.lambda_noobj = 0.5

        # We value coordinate accuracy (x, y, w, h) more than class accuracy
        self.lambda_coord = 5
```

1.3.2 Wybór ramki odpowiedzialnej (IoU)

Dla każdej komórki siatki sieć przewiduje B = 2 ramki. Jednakże, w procesie uczenia tylko jedna z nich powinna być „karana” za błąd współrzędnych – ta, która najlepiej pasuje do rzeczywistego obiektu.

Identyfikator:

$$\mathbb{1}_{ij}^{\text{obj}}$$

przyjmuję wartość 1, gdy j-ta ramka w komórce i posiada najwyższy współczynnik Intersection over Union (IoU) z ramką Ground Truth.

```python
# 2. CALCULATE IoU
# We calculate IoU for both predicted boxes to find which one is 'responsible'
"""
                area_overlap
    Math: IoU = ------------
                 arem_union

"""
iou_b1 = intersection_over_union(predictions[..., 21:25], target[..., 21:25])
iou_b2 = intersection_over_union(predictions[..., 26:30], target[..., 21:25])

# Stack them to compare: shape (2, batch, 7, 7, 1)
ious = torch.cat([iou_b1.unsqueeze(0), iou_b2.unsqueeze(0)], dim=0)

# Get the best box: is the index j (0 or 1) that has the hightes IoU
iou_maxes, best_box = torch.max(ious, dim=0)
```

1.3.3 Składowa 1: Błąd Lokalizacji (Coordinate Loss)

Jest to suma kwadratów różnic między przewidywanymi a rzeczywistymi współrzędnymi środka (x, y) oraz pierwiastkami wymiarów (sqrt(w), sqrt(h))

Wzór matematyczny:

$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right]$$

$$+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]$$

Zastosowanie pierwiastka kwadratowego dla szerokości i wysokości ma na celu zmniejszenie kary dla błędów w dużych ramkach względem małych ramek (np. błąd w postaci 5 pikseli jest mniej istotny dla dużego obiektu niż dla małego)

```
# ========================
#   PART 1: FOR BOX COORDINATES
# ========================
"""
                              s^2  B
    loss_cord = lambda_coord * SUM SUM 1_obj_ij [(x_i - x_hat_i)^2 + (x_i - y_hat_i)^2]
                              i=0 j=0

                              s^2  B
              + lambda_coord * SUM SUM 1_obj_ij [(sqrt(w_i) - sqrt(w_hat_i))^2 + (sqrt(h_i) - sqrt(h_hat_i))^2]
                              i=0 j=0
"""
# Get the coordinates of the responsible box (either box 1 or box 2)
box_predictions = exists_box * (
    (best_box * predictions[..., 26:30] + (1 - best_box) * predictions[..., 21:25])
)

box_targets = exists_box * target[..., 21:25]

# --- SQRT TRANSFORMATION ---
# We use sqrt root because we want to flatten the curve
# ⇒ penalizing errors on small objects more
# (Without this deviations in large boxes matter less than in small boses)

# 1. Calculate safe width/height (predictions)
# CRITICAL: We use torch.abs + 1e-6 to prevent NaN gradients
# (derivative of 0 is infinite)
box_predictions_xy = box_predictions[..., 0:2] # x, y
box_predictions_wh = torch.sign(box_predictions[..., 2:4]) * torch.sqrt(
    torch.abs(box_predictions[..., 2:4]) + 1e-6
)

# 2. Concatenate them back together
# This preserves the gradient history correctly
box_predictions = torch.cat([box_predictions_xy, box_predictions_wh], dim=-1)

# 3. Square root the targets as well
box_targets_xy = box_targets[..., 0:2]
box_targets_wh = torch.sqrt(box_targets[..., 2:4])
box_targets = torch.cat([box_targets_xy, box_targets_wh], dim=-1)

# Calculate Sum Squared Error
box_loss = self.mse(
    torch.flatten(box_predictions, end_dim=-2),
    torch.flatten(box_targets, end_dim=-2),
)
```

1.3.4 Składowa 2: Błąd Pewności (Confidence Loss)

Model uczy się przewidywać wartość C (Confidence Score). Jeśli w komórce jest obiekt, C powinno dążyć do 1 (lub wartości IoU). Jeśli obiektu nie ma, C powinno dążyć do 0.

Wzór matematyczny:

$$+\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{1}_{ij}^{\text{obj}}\left(C_i-\hat{C}_i\right)^2$$

$$+\lambda_{\text{noobj}}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{1}_{ij}^{\text{noobj}}\left(C_i-\hat{C}_i\right)^2$$

```
# ========================
#   PART 2: OBJECT LOSS (Confidence)
# ========================
"""
                s^2  B
    loss_obj =  SUM SUM 1_obj_ij (C_i - C_hat_i)^2
                i=0 j=0
"""

pred_box = (
    best_box * predictions[..., 25:26] + (1 - best_box) * predictions[..., 20:21]
)

object_loss = self.mse(
    torch.flatten(exists_box * pred_box),
    torch.flatten(exists_box * target[..., 20:21]),
)

# ========================
#   PART 3: NO OBJECT LOSS (Background)
# ========================
# Penalize the model if it predicts an object in an empty cell
"""
                  s^2  B
    loss_noobj =  SUM SUM 1_noobj_ij (C_i - C_hat_i)^2
                  i=0 j=0
"""

# (1 - exists_box) creates a mask for Empty Cells
no_object_loss = self.mse(
    torch.flatten((1 - exists_box) * predictions[..., 20:21], start_dim=1),
    torch.flatten((1 - exists_box) * target[..., 20:21], start_dim=1),
)

no_object_loss += self.mse(
    torch.flatten((1 - exists_box) * predictions[..., 25:26], start_dim=1),
    torch.flatten((1 - exists_box) * target[..., 20:21], start_dim=1),
)
```

1.3.5 Składowa 3: Błąd klasyfikacji (Classification Loss)

Jeśli w komórce znajduję się obiekt, sieć jest karana za błędne predykcje prawdopodobieństw klas.

Wzór matematyczny:

$$+\sum_{i=0}^{S^2} \mathbb{1}_i^{\text{obj}} \sum_{c \in \text{classes}} \left(p_i(c) - \hat{p}_i(c)\right)^2$$

```
# ========================
#   PART 4: CLASS LOSS
# ========================
# Predict the class (Dog, Cat, Bike etc.)
"""
                  s^2
    loss_class =  SUM 1_obj_i SUM (p_i(c) - p_hat_i(c))^2
                  i=0         c in classes
"""
class_loss = self.mse(
    torch.flatten(exists_box * predictions[..., :20], end_dim=-2),
    torch.flatten(exists_box * target[..., :20], end_dim=-2),
)
```

1.3.6 Całkowita funkcja straty

Ostatecznie wartość straty, która jest poddawana propagacji wstecznej w metodzie loss.backward() w pętli treningowej, jest ważoną sumą powyższych składników.

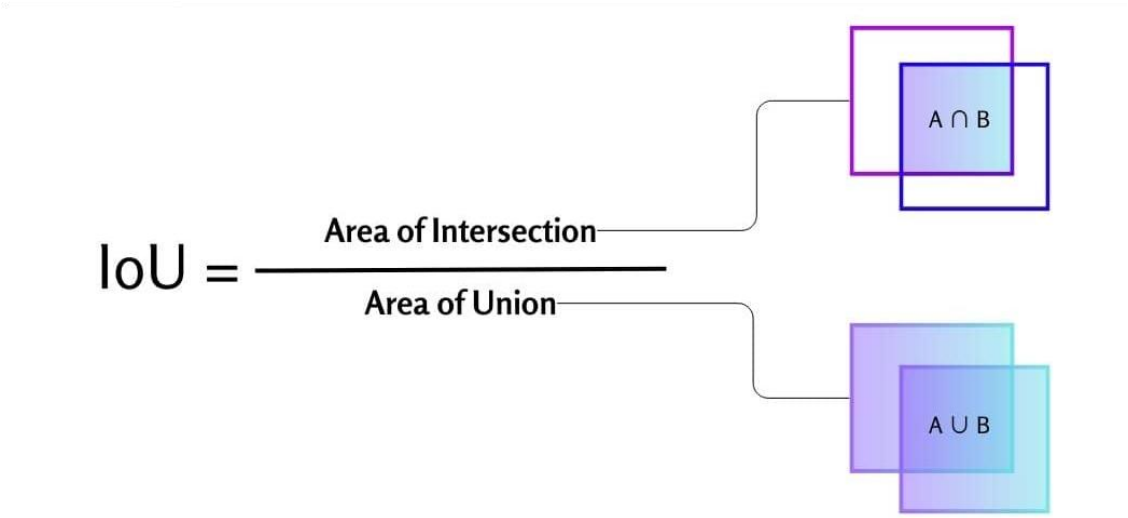
```
# ========================
#   TOTAL LOSS
# ========================
loss = (
    self.lambda_coord * box_loss            # weights = 5.0
    + object_loss                           # weights = 1.0
    + self.lambda_noobj * no_object_loss    # weights = 0.5
    + class_loss                            # weights = 1.0
)
```

1.4 Metryki oceny jakości detekcji i algorytmy post-processingu

W zadaniach detekcji obiektów proste metryki trafności, takie jak np. w klasyfikacji, nie są wystarczające ze względu na niezbalansowanie klas (dominacja tła nad obiektami) oraz konieczność oceny precyzji geometrycznej ramki. W projekcie zastosowano standardowe metryki dla zbioru PASCAL VOC: Intersection over Union (IoU) oraz Mean Average Precision (mAP), zaimplementowane w module utils.py.

1.4.1 Intersection over Union (IoU)

IoU, zwane też indeksem Jaccarda, jest fundamentalną miarą określającą stopień nakładania się dwóch obszarów: ramki przewidywanej(A) oraz ramki wzorcowej(B):



Wartość IoU mieści się w przedziale [0, 1], gdzie 1 oznacza idealne pokrycie, W projekcie IoU pełni dwie funkcje:

1. **Podczas treningu:** Decyduję, która ramka jest odpowiedzialna za obiekt.
2. **Podczas walidacji:** Pozwala zaklasyfikować predykcję jako Prawdziwą Pozytywną (True Positive), jeśli IoU > próg (np.: 0.5)

```
# --- GEOMETRY OF INTERSECTION ---
# The Top-Left of the intersection is the MAX of the two Top-Lefts
x1 = torch.max(box1_x1, box2_x1)
y1 = torch.max(box1_y1, box2_y1)

# The Bottop-Right of the intersection is the MIN of the two Botton-Rights
x2 = torch.min(box1_x2, box2_x2)
y2 = torch.min(box1_y2, box2_y2)

# CRITICAL: .clamp(0) is for the case when they do not intersect
# If boxes do not overlap, (x2 - x1) will be negative
# area cannot be negative, so we clamp it to 0
intersection = (x2 - x1).clamp(0) * (y2 - y1).clamp(0)

box1_area = abs((box1_x2 - box1_x1) * (box1_y2 - box1_y1))
box2_area = abs((box2_x2 - box2_x1) * (box2_y2 - box2_y1))

# Add epsilon to prevent division by zero
return intersection / (box1_area + box2_area - intersection + 1e-6)
```

1.4.2 Non-Max Suppression (NMS)

Model YOLO często generuje wiele ramek dla tego samego obiektu (z różnym prawdopodobieństwem). Aby otrzymać ostateczny wynik, stosuje się algorytm tłumienia niemaksymalnego (NMS).

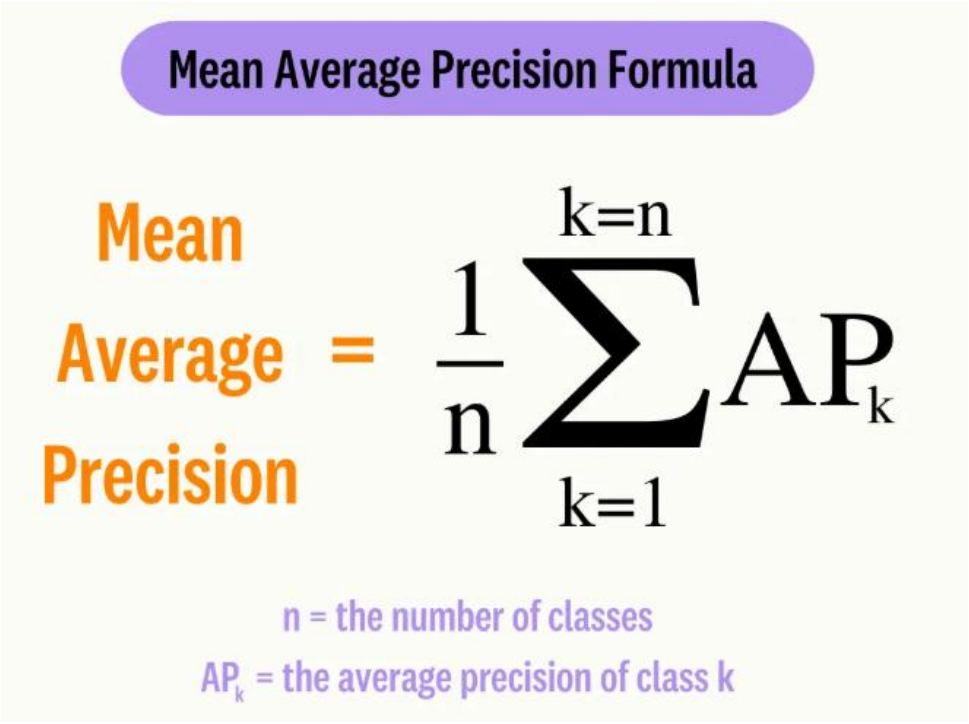
Algorytm:

1. Odrzuć ramki z pewnością poniżej progu detekcji.
2. Wybierz ramkę z najwyższym prawdopodobieństwem.
3. Usuń wszystkie pozostałe ramki, które mają z nią wysokie IoU, zakładając, że dotyczą tego samego obiektu.
4. Powtarzaj kroki 2-3 do wyczerpania listy

```
while bboxes:
    chosen_box = bboxes.pop(0)

    bboxes = [
        box
        for box in bboxes
        if box[0] != chosen_box[0]
        or intersection_over_union(
            torch.tensor(chosen_box[2:]),
            torch.tensor(box[2:]),
            box_format=box_format,
        )
        < iou_threshold
    ]

    bboxes_after_nms.append(chosen_box)
```

1.4.3 Mean Average Precision (mAP)

Jest to standard oceny detektorów. mAP to średnia wartość AP (ang. Average Precision) obliczona dla wszystkich klas C.



Aby obliczyć AP dla danej klasy, analizuje się krzywą Precyzji-Czułości (Precision-Recall Curve).

1. **Precision:** Jaki procent wykrytych obiektów poprawny?
2. **Recall:** Jaki procent wszystkich istniejących obiektów został wykryty?

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

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

TP = True positive

TN = True negative

FP = False positive

FN = False negative

Average Precision (AP) jest polem pod wykresem krzywej Precision-Recall p®:

$$\text{PR-AUC} = \int_0^1 prec(rec)\, d(rec)$$

Podczas obliczeń komputerowych całkę tę przybliża się sumowaniem numerycznym (np. metoda trapezów).

```
    # add start points (precision 1, recall 0) for integration
    precisions = torch.cat((torch.tensor([1]), precisions))
    recalls = torch.cat((torch.tensor([0]), recalls))

    # torch.trapz for numerical integration (area under curve)
    average_precisions.append(torch.trapz(precisions, recalls))

return sum(average_precisions) / len(average_precisions)
```

2. Opis danych wejściowych i analiza eksploracyjna

Skuteczność modelu uczenia maszynowego jest bezpośrednio skorelowana z jakością i strukturą danych treningowych. W projekcie wykorzystano standardowy zbiór danych PASCAL VOC. Jak podkreśla Mark Everingham, celem tego zbioru jest „… dostarczenie standardowego benchmarku do oceny algorytmów rozpoznawania i detekcji kategorii obiektów wizualnych”[7].

2.1 Charakterystyka zbioru PASCAL VOC

Zbiór PASCAL VOC składa się z obrazków przypisanych do 20 kategorii semantycznych. Kategorie te to: osoba, ptak, kot, krowa, pies, koń, owca, samolot, rower, łódź, autobus, samochód, motocykl, pociąg, butelka, krzesło, stół, roślina doniczkowa, sofa, monitor.

Każdy obraz w zbiorze zawiera adnotacje w postaci ramek ograniczających (bounding boxes) oraz etykiet klas. Istotną cecha tego zbioru jest duża zmienność wewnątrzklasowa (różne pozy, oświetlenie, zasłonięcia) oraz niezbalansowanie klas (np. znacznie więcej przykładów klasy „osoba” niż „sofa”), co stanowi wyzwanie dla modelu.

2.2 Struktura danych i format etykiet

W ramach implementacji projektu (plik dataset.py), dane [obrazy .jpg i adnotacje .txt] zostały przekonwertowane do formatu tekstowego, bardzie przyjaznego dla modelu YOLO.

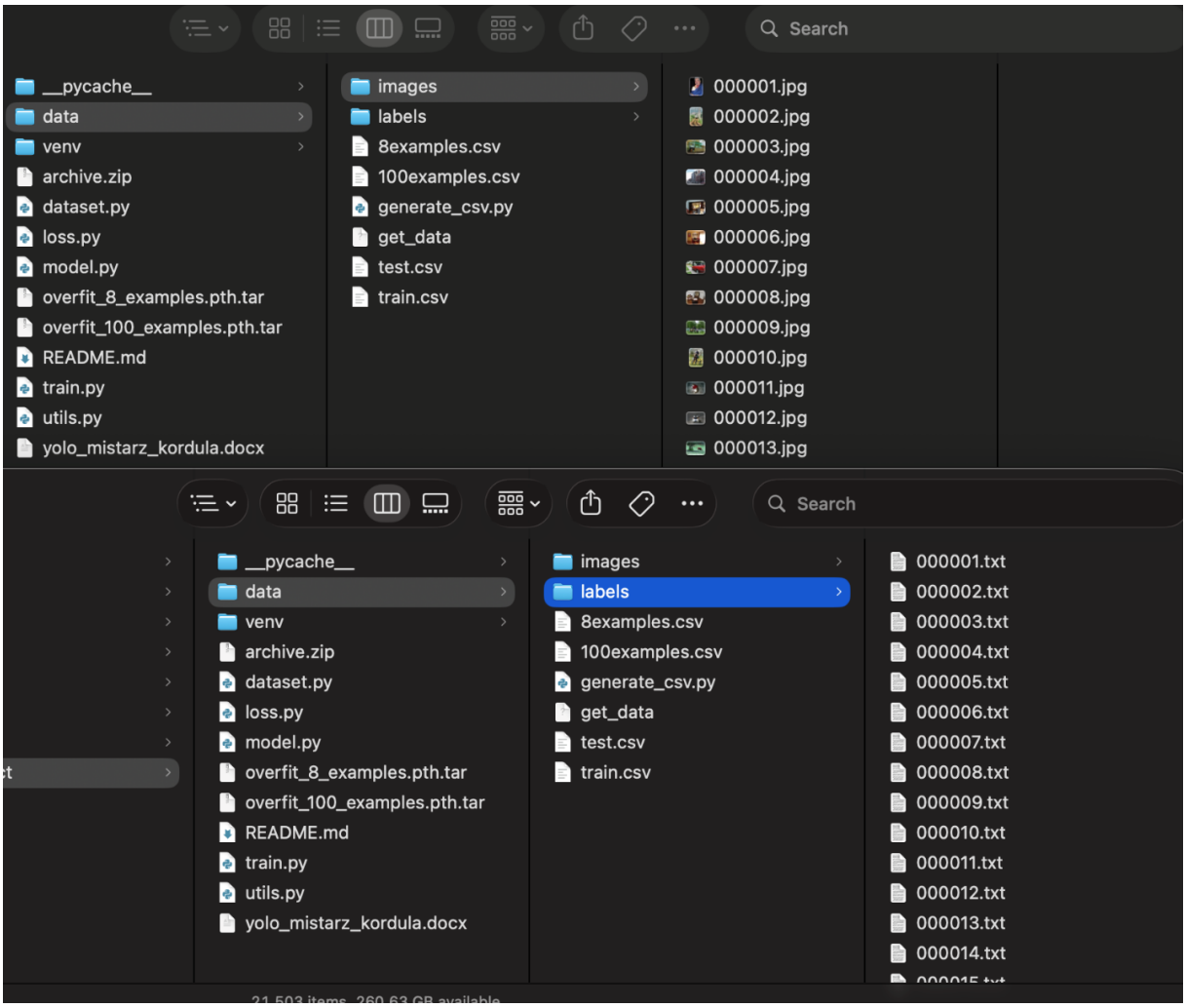
Struktura folderów w projekcie:

- /data/images/ - pliki graficzne

- /data/labes/ - pliki tekstowe z adnotacjami

- train.csv / test.csv – pliki mapujące obrazów do nazw plików z etykietami

[7] Everingham Mark, The Pascal Visual Object Classes (VOC) Challenge, Oxford 2010, s.303



2.2.1 Format pliku etykiet

Każdy plik tekstowy w katalogu /labels/ odpowiada jednemu obrazowi i zawiera wiersze opisujące obiekty, zgodnie ze schematem:

[class_id] [x_center] [y_center] [width] [height]

Wartości współrzędnych są znormalizowane względem wymiarów obrazu, przyjmując wartości z przedziału [0, 1].

Przykład (plik 0000028.txt):

000028.txt

```
7 0.5826666666666667 0.518 0.8293333333333333 0.964
```

Oznacza to obiekt klasy 7 („cat”), którego środek znajduję się w około 58.3 % szerokości i 51.8 % wysokości obrazu.

Przykład (000028.jpg):

Implementacja w kodzie (dataset.py): Klasa VOCDataset wczytuje dane w metodzie __getitem__:

```
def __getitem__(self, index):
    # 1. READ LABEL FILE
    # format: (class_id, center_x, center_y, width, height)
    # NOTE: all coordinates are normalized (0 to 1) relative to img size
    label_path = os.path.join(self.label_dir, self.annotations.iloc[index, 1])
    boxes = []
    with open(label_path) as f:
        for label in f.readlines():
            class_label, x, y, width, height = [
                float(x) if float(x) != int(float(x)) else int(x)
                for x in label.replace("\n", "").split()
            ]

            boxes.append([class_label, x, y, width, height])
```

2.3 Przetwarzanie wstępne i transformacje (Preprocessing)

Modele oparte na CNN wymagają stałego wymiaru wejściowego. Oryginalne obrazy w zbiorze PACAL VOC mają różne rozdzielczości, dlatego konieczne jest ich przeskalowanie.

W pliku train.py zdefiniowano pipeline przy użyciu klasy Compose:

```
# resize to 448x448 (YOLO requirement) and convert to Tensor
transform = Compose([transforms.Resize((448, 448)), transforms.ToTensor()])
```

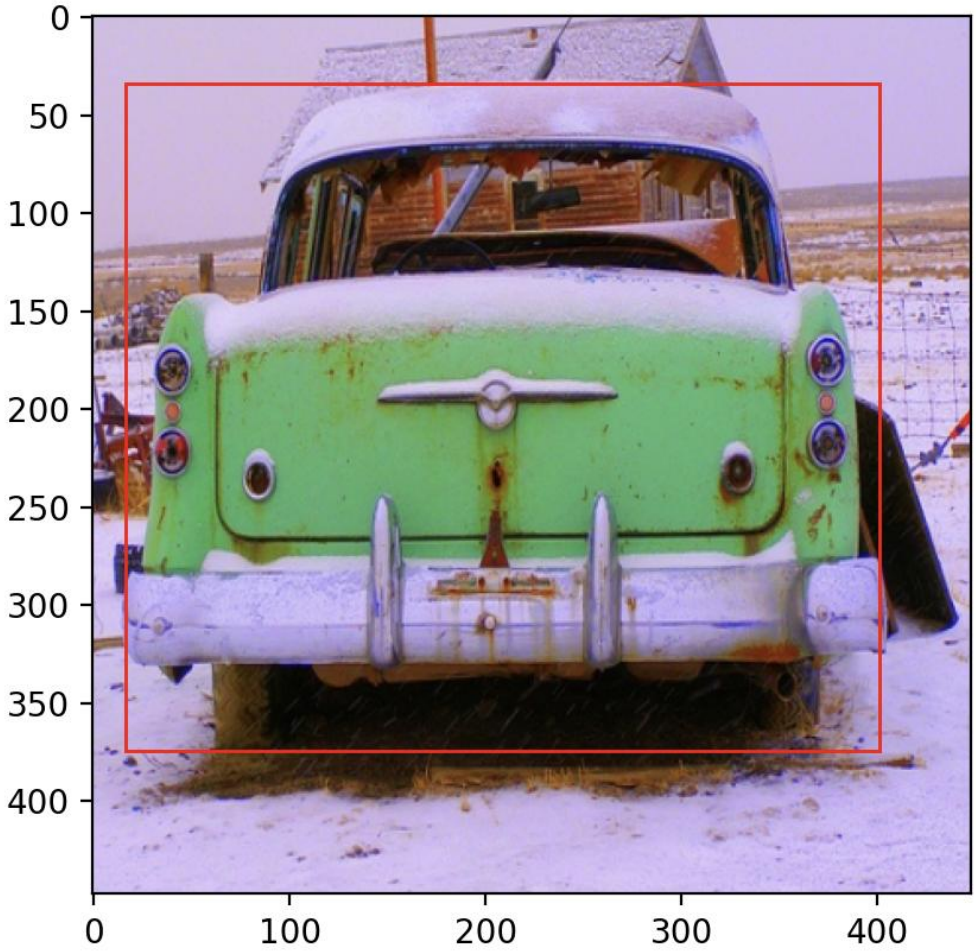
1. Resize ((448, 448)): Obrazy są skalowane do rozmiaru 448 x 448 pikseli. Jest to wymóg architektury YOLO (wynika z liczby warstw splotowych i redukcji wymiarowości, aby ostateczny tensor miał wymiar 7 x 7).
2. ToTensor: Konwersja obrazu z macierzy liczb całkowitych (0 – 255) na Tensor PyTorch (0.0 – 1.0)

*Skalowanie obrazu wymaga jednoczesnego skalowania ramek ograniczających, co jest obsługiwane przez klasę Compose.

2.4 Wizualizacja danych (Analiza Jakościowa)

Przed rozpoczęciem procesu uczenia przeprowadzono wizualizację przykładowych danych treningowych, aby zweryfikować poprawność wczytywania etykiet oraz konwersji współrzędnych. Wykorzystana do tego funkcje plot_image z modułu utils.py

```
def plot_image(image, boxes):
    """Plots predicted bounding boxes on the image"""
    im = np.array(image)
    height, width, _ = im.shape

    # Create figure and axes
    fig, ax = plt.subplots(1)      • "fig" is not accessed
    # Display the image
    ax.imshow(im)

    # box[0] is x midpoint, box[2] is width
    # box[1] is y midpoint, box[3] is height

    # Create a Rectangle potch
    for box in boxes:
        box = box[2:]
        assert len(box) == 4, "Got more values than in x, y, w, h, in a box!"

        # geometry: convert center (x,y) to top-left (x,y) for matplotlib
        upper_left_x = box[0] - box[2] / 2
        upper_left_y = box[1] - box[3] / 2
        rect = patches.Rectangle(
            (upper_left_x * width, upper_left_y * height),
            box[2] * width,
            box[3] * height,
            linewidth=1,
            edgecolor="r",
            facecolor="none",
        )
        # Add the patch to the Axes
        ax.add_patch(rect)

    plt.show()
```



Etykieta = (6, 0.4700000000003, 0457333333333333, 0.876, 0.7653333333333)

Wizualizacja ta pozwala potwierdzić, że:

1. Współrzędne (x, y) są poprawnie interpretowane jako środek ramki.
2. Wymiary (w, h) są adekwatne do wielkości obiektu.
3. Etykiety klas (reprezentowane przez numery ID) odpowiadają zawartości obrazu.

Tego typu analiza eksploracyjna pozwala wyeliminować błędy, w których model trenuję się na niepoprawnych danych wejściowych.

3. Badania symulacyjne i ocena jakości modelu

Celem badań symulacyjnych była weryfikacja poprawności zaimplementowanej architektury YOLO oraz funkcji straty poprzez proces uczenia nadzorowanego. Proces badawczy podzielono na dwa etapy: weryfikację wstępną (Sanity Check) polegającą na celowym przeuczeniu modelu na małej próbie danych, oraz omówienie procedury treningu właściwego z uwzględnieniem walidacji krzyżowej.

3.1 Środowisko eksperymentalne i konfiguracja hierparametrów

Implementację zrealizowano w języku Python(v3.13.9) z wykorzystaniem biblioteki PyTorch. Ze względu na wysoką złożoność obliczeniową operacji splotu, obliczenia były akcelerowane sprzętowo. W środowisku testowym wykorzystano akcelerator Apple MPS (Metal Performance Shaders):

3.1.1 Dobór optymalizatora

Do aktualizacji wag sieci wykorzystano algorytm Adam (Adaptive Moment Estimation)8. W przeciwieństwie do klasycznego stochastycznego spadku gradientu (SGD), Adam adaptuje tempo uczenia dla każdego parametru indywidualnie, wykorzystując estymacje pierwszego i drugiego momentu gradientów. Pozwala to na szybszą zbieżność modelu w początkowej fazie treningu. Diederik Kingma wskazuje, że główną zaletą tej metody jest to, iż „… oblicza ona indywidualne adaptacyjne współczynniki uczenia dla różnych parametrów na podstawie estymacji momentów gradientów”[8].

3.1.2 Hiperparametry treningowe

Podczas procesu uczenia przyjęto następującą konfigurację parametrów:

1. **Learning Rate** – Niska wartość zapewnia stabilną zbieżność i minimalizuje ryzyko oscylacji funkcji straty w pobliżu minimum globalnego.
2. **Batch size** – Wartość dobrana eksperymetnalnie.
3. **Weight Decay** – W fazie weryfikacji wyłączono regularyzację L2, aby umożliwić pełne dopasowanie do danych treningowych.
4. **Epochs** – Liczba epok wystarczająca do pełnego stabilizacji funkcji kosztu.

Kod train.py:

```
36 LEARNING_RATE = 2e-5
37 BATCH_SIZE = 16
38 WEIGHT_DECAY = 0
39 EPOCHS = 256
40 NUM_WORKERS = 2
```

[8] Kingma Diederik, Adam: A Method for Stochastic Optimization, San Diego 2015, s.1

3.2 Implementacja pętli treningowej i algorytm propagacji

Kluczowym elementem implementacji, znajdującym się w funkcji train_fn w pliku train.py, jest pętla realizująca proces uczenia nadzorowanego. Algorytm ten iteruje po wsadach danych (batches) dostarczonych przez obiekt DataLoader.

3.2.1 Forward pass

W pierwszej fazie tensor obrazów wejściowych X jest przekazywany przez warstwy sieci neuronowej. Model zwraca tensor predykcji Y_pred który zawiera surowe logity.

```
x, y = x.to(DEVICE), y.to(DEVICE)

# 1. FORWARD PASS
# calculate predictions y_hat = f(x)
out = model(x)
```

3.2.2 Obliczanie wartości funkcji starty (loss function)

Uzyskane predykcje (out) oraz etykiety(y) są przekazywane do funkcji kosztu YoloLoss. W tyk korku następuje agregacja błędów lokalizacji, pewności i klasyfikacji do jednej wartości skalarnej J.

```
# 2. CALCULATE LOSS
# calculate error J = MSE(y_hat, y)
loss = loss_fn(out, y)
mean_loss.append(loss.item())
```

Wartość jest następnie logowana (metoda mean_loss.append) w celu monitorowania przebiegu uczenia i wizualizacji krzywej uczenia.

3.2.3 Propagacja wsteczna (Backward Pass)

Jest to kluczowy moment uczenia sieci. Wywołanie metody backward() uruchamia mechanizm automatycznego różniczkowania (autograd). Biblioteka PyTorch oblicza gradienty funkcji kosztu względem każdego parametru (wagi) sieci.

• Należy pamiętać o jawnym wyzerowaniu gradientów z poprzedniej iteracji, aby uniknąć ich akumulacji, co jest domyślnym zachowaniem w PyTorch.

```
# 3. BACKWARD PASS (Backpropagation)
# reset gradients from previous step to 0
optimizer.zero_grad()

# calculate gradients dJ/d_theta
loss.backward()
```

3.2.4 Aktualizacja wag (Optimization Step)

W ostatnim kroku optymalizator aktualizuje parametry modelu, wykonując krok w kierunku przeciwnym do gradientu.

```
    # 4. OPTIMIZER STEP (Gradient Descent)
    # update weights: theta = theta - lr * gradient
    optimizer.step()
```

3.3 Badanie - Sanity Check

Przed rozpoczęciem długotrwałego treningu na pełnym zbiorze danych, przeprowadzono test poprawności typu „Overfit Sanity Check".

```
# Evalutate Performance
# Note: For the Sanity Check, we evaluate on the TRAINING set
# We want to see if the model has memorized the answers.
# !!! In real training, we should evalutae on test_loader !!!
pred_boxes, target_boxes = get_bboxes(
    train_loader,
    model,
    iou_threshold=0.5,
    threshold=0.4,
    device=DEVICE
)
```

3.3.1 Metodyka

Model trenowano na bardzo małym podzbiorze (100) przy wyłączonej regularyzacji. W tym scenariuszu zbiór testowy był identyczny ze zbiorem treningowym.

3.3.2 Cel

Test na małym zbiorze danych doprowadził do przeuczenia. Sebastian Raschka wyjaśnia to zjawisko stwierdzenie, że „… im mniejsza liczba parametrów modelu w stosunku do danych, tym mniejsza jest jego zdolność zapamiętywania szumu"[9], co w naszym przypadku (duży model, mało danych) zadziałało odwrotnie – model zapamiętał szum. Jest to dowód na to, że architektura sieci ma wystarczającą pojemność (capacity), a funkcja straty poprawnie kieruje gradient.

3.3.3 Kryterium sukcesu i wyniki eksperymentu

1. Spadek funkcji straty do wartości mniejszych niż 25.
2. Wzrost metryki mAP (Mean Average Precision) do wartości powyżej 0.95.

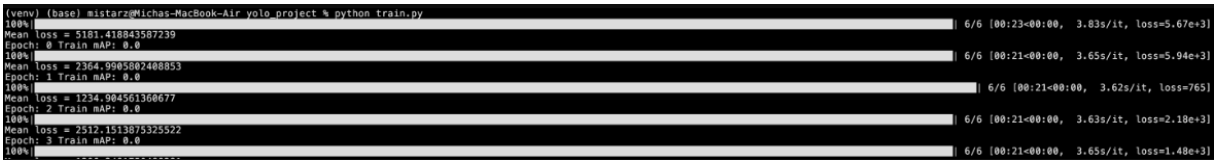
[9] Raschka Sebastian, Python Machine Learning, Birmingham 2017, s. 125

3.3.4 Wyniki eksperymentu

W przeprowadzonym badaniu zaobserwowano następującą dynamikę:

1. Faza początkowa (Epoki 0 – 30): Początkowa wartość funkcji straty była bardzo wysoka (MSE = 5181), co wynika z losowej inicjalizacji wag sieci, gdzie predykcje są całkowicie rozbieżne z rzeczywistością. W ciągu pierwszych 30 epok nastąpił gwałtowny spadek błędu do poziomu ok. 200. W tej fazie metryka była blisko 0.0, co oznacza, że mimo poprawy ogólnej, model nie predykował jeszcze ramek z IoU > 0.5.
2. Faza uczenia cech geometrycznych (Epoki 30-150): Po spadku Loss poniżej poziomu 200, sieć zaczęła skutecznie lokalizować obiekty. Metryka mAP zaczęła przyjmować wartości niezerowe, systematycznie rosnąc. Świadczy to o tym, że gradienty funkcji CoordLoss zaczęły dominować nad gradientami tła.
3. Faza precyzyjnego dostrajania (Epoki 150-184): W końcowej fazie eksperymentu funkcja straty ustabilizowała się na poziomie 20.0 – 30.0. Model osiągnął bardzo wysoką precyzję detekcji. W epoce 183 było to mAP około 96%.

Epoch 0 – 3:



Epoch 28-30:

Epoch 181-183:

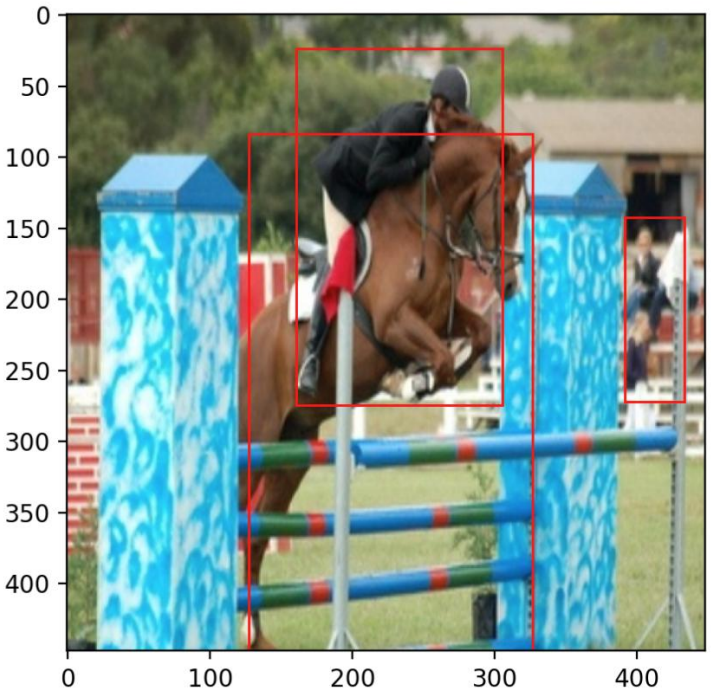
3.3.5 Wnioski

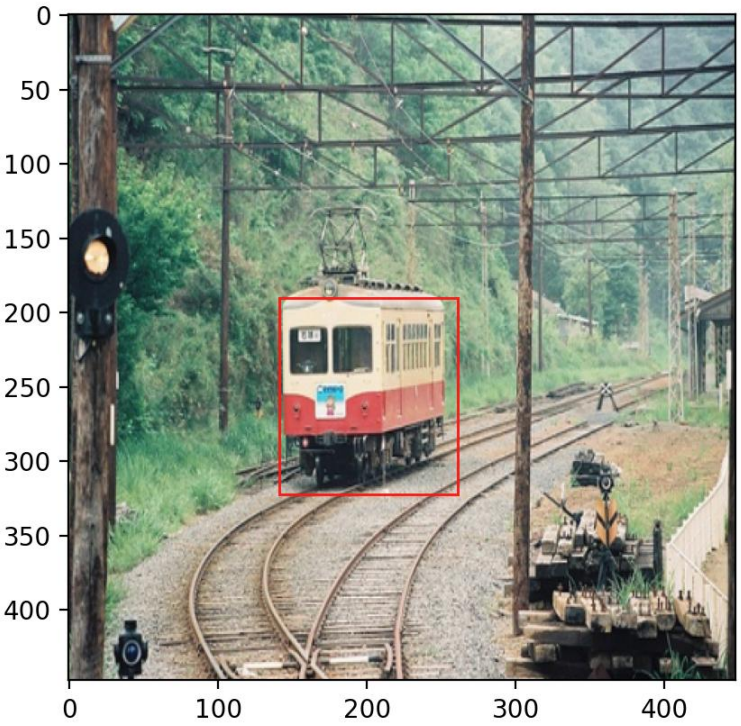
Osiągnięci mAP na poziomie 96% przy jednoczesnej redukcji błędu o dwa rzędy wielkości stanowi dowód na poprawność implementacji.

Eksperyment ten potwierdza, że:

1. Tensor wejściowy jest poprawnie dekodowany na współrzędne ramek.
2. Funkcja straty prawidłowo karze błędne predykcje.
3. Nie występują błędy w transformacji danych.

3.3.6 Wyniki – zdjęcia





3.4 Badanie - zdolność generalizacji

W celu zbadania wpływu wielkości zbioru treningowego na zdolność generalizacji modelu, przeprowadzono eksperyment polegający na treningu sieci na ewaluacji modelu wytrenowanego w sekcji 3.3 (na próbie 100 obrazów) na pełnym, niedzianym wcześniej zbiorze testowym (test.csv).

```
# Evalutate Performance
pred_boxes, target_boxes = get_bboxes(
    test_loader,
    model,
    iou_threshold=0.5,
    threshold=0.4,
    device=DEVICE
)
```

Wyniki eksperymentu:

```
(venv) (base) mistarz@Michas-MacBook-Air yolo_project % python train.py
=> Loading checkpoint
100%|| 6/6 [00:22<00:00,  3.67s/it, loss=26.4]
Mean loss = 50.74107392628988
Epoch: 0 Train mAP: 0.0013572729658335447
100%|| 6/6 [00:22<00:00,  3.70s/it, loss=37.5]
Mean loss = 29.661213556925457
100%|| 6/6 [00:22<00:00,  3.68s/it, loss=27.2]
Mean loss = 21.538923899332683
100%|| 6/6 [00:21<00:00,  3.64s/it, loss=16]
Mean loss = 19.106818517049152
100%|| 6/6 [00:21<00:00,  3.62s/it, loss=13.8]
Mean loss = 15.555009841918945
100%|| 6/6 [00:21<00:00,  3.66s/it, loss=16]
Mean loss = 16.203779220581055
100%|| 6/6 [00:24<00:00,  4.04s/it, loss=13.9]
Mean loss = 14.974305152893066
100%|| 6/6 [00:25<00:00,  4.28s/it, loss=16.1]
Mean loss = 13.780105749766031
100%|| 6/6 [00:26<00:00,  4.33s/it, loss=13.6]
Mean loss = 12.469396352767944
100%|| 6/6 [00:26<00:00,  4.44s/it, loss=8.21]
Mean loss = 11.263947168986002
100%|| 6/6 [00:24<00:00,  4.07s/it, loss=18.1]
Mean loss = 10.886321465174357
Epoch: 10 Train mAP: 0.0015653675654903054
```

Eksperyment ujawnił występowanie ogromnej luki generalizacji (Generalization Gap). Jest to klasyczny, empiryczny dowód na zjawisko przeuczenia (overfitting) wynikającego ze zbyt małej liczebności próby badawczej. Sieć neuronowa „zapamiętała" specyficzne układy pikseli dla 100 obrazów treningowych, jednak nie byłą w stanie nauczyć się ogólnych cech, które pozwoliłyby jej rozpoznawać obiekty na nowych zdjęciach, które nigdy wcześniej nie widziała.

3.5 Wnioski końcowe z badań symulacyjnych

Wnioski sformułowane na podstawie przeprowadzonych badań:

1. **Poprawność implementacji:** Osiągnięcie mAP na poziomie 96% na zbiorze treningowym oraz spadek funkcji starty z 5181 do 20 jednoznacznie potwierdzają, że zaimplementowana architektura YOLOv1, funkcja straty oraz algorytm propagacji wstecznej działają poprawnie pod względem matematycznym i programistycznym.
2. **Wymagania danych:** Eksperyment porównawczy (train vs test) wykazał, że głębokie sieci konwolucyjnenie nie są w stanie generalizować wiedzy na podstawie małej liczby przykładów (N=100). Aby uzyskać użyteczny detektor (Test mAP >0.5), konieczne jest przeprowadzenie treningu na pełnym zbiorze PASCAL VOC (ok. 5000 obrazów), co wymagałoby większych zasobów obliczeniowych i dłuższego treningu.

3. **Gotowość rozwiązania:** Zrealizowany projekt stanowi w pełni funkcjonalny szkielet systemu detekcji. Niski wynik na zbiorze testowym wynika wyłącznie z ograniczonej ilości danych treningowych użytych w trakcie uczenia modelu.

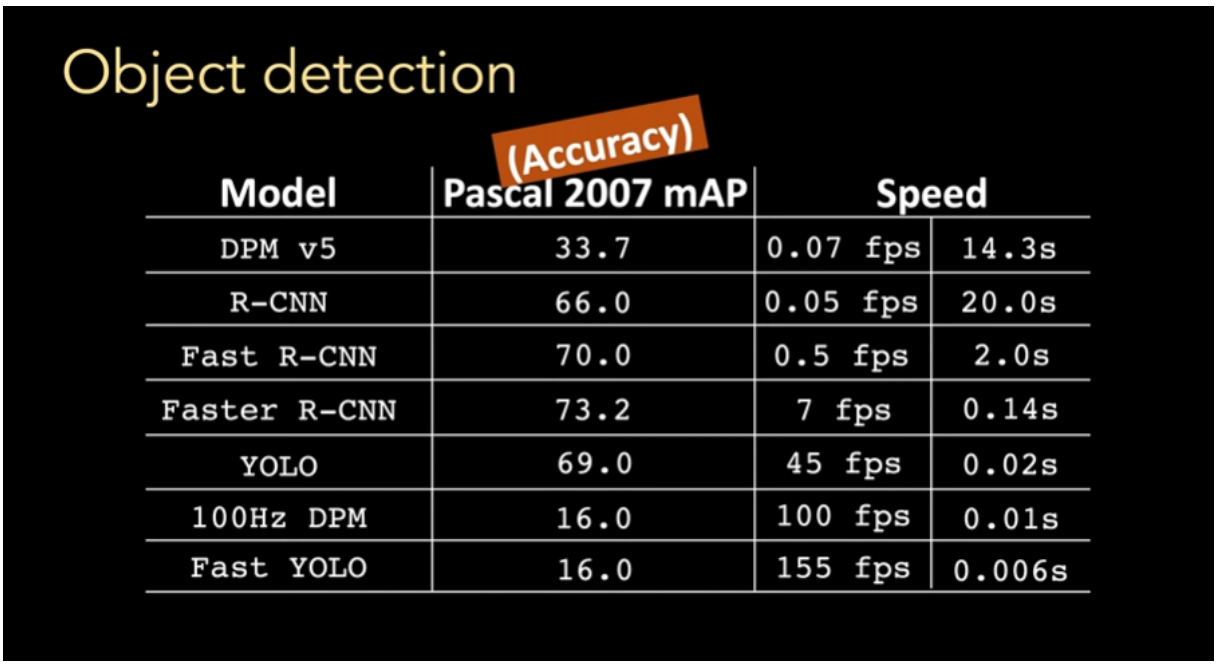
4. Analiza porównawcza architektury YOLO i rozwój metod detekcji

Zrealizowane w poprzednim rozdziale badania symulacyjne miały charakter weryfikacyjny. Potwierdziły one poprawność implementacji matematycznej modelu oraz funkcji straty, jednak ze względu na ograniczenia sprzętowe niemożliwe było przeprowadzenie pełnego treningu na zbiorze PASCAL VOC, który pozwoliłby uzyskać wyniki podobne do oryginalnego modelu.

W związku z tym, rozdział 4 koncentruje się na analizie teoretycznej samej architektury YOLO. Na podstawie danych literaturowych porównano ją z rozwiązaniami konkurencyjnymi (np. R-CNN) oraz omówiono ewolucję tej metody (od wersji v1 do v11), która doprowadziła do jej obecnej dominacji w systemach czasu rzeczywistego.

4.1 Analiza wydajności na tle rozwiązań konkurencyjnych

Główną przewagą modelu YOLO jest szybkość. Aby zrozumieć przewagę tego podejścia, należy odnieść się do wyników benchmarkowych opublikowanych przez twórców metody, porównujących ją do ówczesnych standardów.



Model	Pascal 2007 mAP	Speed	
DPM v5	33.7	0.07 fps	14.3s
R-CNN	66.0	0.05 fps	20.0s
Fast R-CNN	70.0	0.5 fps	2.0s
Faster R-CNN	73.2	7 fps	0.14s
YOLO	69.0	45 fps	0.02s
100Hz DPM	16.0	100 fps	0.01s
Fast YOLO	16.0	155 fps	0.006s

Tabela 1

Wnioski z analizy:

1. **Dominacja prędkości:** Architektura YOLO osiąga prędkość 45 klatek na sekundę (FPS). Jest to wynik rzędu wielkości lepszy od Faster R-CNN (7FPS) i blisko 1000 krotnie lepszy od R-CNN (0.05 FPS). Oznacza to, że zaimplementowana w pracy metoda jest jedyną z tego zestawienia która teoretycznie nada się do płynnego przetwarzania wideo w czasie rzeczywistym.
2. **Koszt precyzji:** Wysoka prędkość została okupiona nieznacznym spadkiem precyzji (69.0 mAP względem 73.2 mAP dla Faster R-CNN).

4.2 Ewolucja rodziny modeli YOLO (v1 – v11)

Projekt zrealizowany w ramach tej pracy opiera się na fundamentalnej wersji v1. Jednakże dynamiczny rozwój dziedziny wizji komputerowej sprawił, że architektura ta przeszła znaczącą ewolucję. Współczesne systemy wizyjne wykorzystują ulepszone warianty tego algorytmu, które eliminują ograniczenia z wersji pierwszej.

4.2.1 Era Darknet (v1 – v3)

Twórcą pierwszych wersji był Joseph Redmon. Modele te kładły podwaliny pod współczesną detekcję.

- YOLOv2 (YOLO9000): Rozwiązał problem sztywnej siatki, wprowadzając tzw. Anchor Boxes – predefiniowane kształty ramek, co znacznie ułatwiło sieci dopasowywanie się do obiektów o różnych proporcjach.
- YOLOv3: Wprowadził detekcję na trzech różnych skalach (Feature Pyramid Netowrks). Był to kluczowy moment, który pozwolił modelowi YOLO skutecznie wykrywać bardzo małe obiekty, z czym wersja v1 miała istotne probely.

4.2.2 Era optymalizacji (v4 – v7)

Rozwój przejęła społeczność Open Source (m. in. Firma Ultralytics).

- YOLOv4 i v5: Wersja v5 stała się standardem przemysłowym ze względu na łatwość użycia. Wprowadzono zaawansowane techniki agumentacji danych (np.. Mosaic), które pozwalają trenować skutecznie modele nawet na mniejszych zbiorach danych.
- YOLOv7: Skupił się na redukcji kosztu obliczeniowego, oferując najlepszy stosunek wydajności do zużycia energii, co jest kluczowe w systemach wbudowanych.

4.2.3 Stan obecny: Modele Anchor-Free (v8 – v11)

Najnowsze generacje uprościły architekturę.

- YOLOv8: Przejście na architekturę Anchor-Free (bez ramkową). Model nie musi już dopasowywać predefiniowanych ramek, lecz bezpośrednio przewiduje środek obiektu. Znacznie uprościło to proces treningu i doboru hiperparametrów.
- YOLOv11: Najnowsza odsłona opracowana przez Ultralytics. Wprowadza ulepszone moduły ekstrakcji cech, które pozwalają na osiągnięcie wyższej precyzji przy mniejszej liczbie parametrów. Wersja, v11 jest modelem multimodalnym, co pozwala nie tylko na detekcję, ale też segmentację, klasyfikację i śledzenie pozy.

5. Podsumowanie i wnioski końcowe

Celem projektu było praktyczne zastosowanie metod Deep Learningu w problemie detekcji obiektów poprzez implementację modelu YOLO. Projekt zrealizowano w środowisku PyTorch, przy wykorzystaniu zbioru danych: PASCAL VOC.

Rezultaty:

1. Weryfikacja Implementacji (Proof of Concept): Przeprowadzony eksperyment weryfikacyjny (Sanity Check w Rozdziale 3) zakończył się sukcesem. Model osiągnął mAP na poziomie 96 % na małym wycinku danych, a funkcja straty spadła z wartości początkowej ~5181 do ~20. Potwierdza to poprawność implementacji kluczowych komponentów. Architektury CNN, funkcji straty oraz algorytmów transformacji ramek.
2. Analiza ograniczeń: Eksperymenty wykazały, że trenowanie głębokich sieci detekcyjnych „od zera” wymaga zasobów sprzętowych przekraczających możliwości standardowych komputerów osobistych oraz większego zbioru danych, aby uniknąć zjawiska przeuczenia (overfittingu), które zaobserwowano przy walidacji na zbiorze testowym.
3. Wnioski praktyczne: Model YOLOv1 ma wiele ograniczeń, przez co w zastosowaniach komercyjnych należałoby skorzystać z nowszych wersji.

Realizacja projektu i wnioski potwierdziły, że – jak uważa Richard Szeliski – „rozpoznawanie wizualne jest prawdopodobnie najtrudniejszym problemem w widzeniu komputerowym”[10], wymagającym balansu między precyzją a szybkością.

[10] Szeliski Richard, Computer Vision: Algorithms and Applications, Springer, Londyn 2010, s. 605

6. Bibliografia

6.1 Pozycje zwarte i artykuły naukowe:

1. Bishop Christopher, Pattern Recognition and Machine Learning, Springer, Nowy Jork 2006.
2. Chollet François, Deep Learning with Python, Manning Publications, Shelter Island 2017.
3. Everingham Mark, The Pascal Visual Object Classes (VOC) Challenge, International Journal of Computer Vision, Oxford 2010.
4. Géron Aurélien, Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, O'Reilly Media, Sebastopol 2019.
5. Girshick Ross, Rich feature hierarchies for accurate object detection and semantic segmentation, CVPR, Columbus 2014.
6. Goodfellow Ian, Deep Learning, MIT Press, Cambridge 2016.
7. Kingma Diederik, Adam: A Method for Stochastic Optimization, ICLR, San Diego 2015.
8. Raschka Sebastian, Python Machine Learning, Packt Publishing, Birmingham 2017.
9. Redmon Joseph, You Only Look Once: Unified, Real-Time Object Detection, CVPR, Las Vegas 2016.
10. Szeliski Richard, Computer Vision: Algorithms and Applications, Springer, Londyn 2010.

6.2 Źródła internetowe:

1. Dokumentacja biblioteki PyTorch, Torch.nn.MSELoss, https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html, dostęp: 11.01.2026.
2. Strona Kaggle
3. https://www.kaggle.com/datasets/734b7bcb7ef13a045cbdd007a3c19874c2586ed0b02b4afc86126e89d00af8d2, dostęp: 15.12.2025.