

# Projekt CRoadA - generator siatek dróg miejskich

Konrad Ćwięka, Szymon Kowalski, Michał Król,  
Grzegorz Lenarski, Jakub Łabuz

20 lutego 2026

## Spis treści

<b>1 Lider grupy</b>	<b>3</b>
<b>2 Repozytorium z kodem źródłowym</b>	<b>3</b>
<b>3 Opis projektu</b>	<b>3</b>
<b>4 Sposób realizacji projektu</b>	<b>3</b>
4.1 Zbiór uczący . . . . .	3
4.1.1 Sposób reprezentacji danych wejściowych . . . . .	3
4.1.2 Informacje zawarte w zbiorze i ich źródła . . . . .	5
4.1.2.1 Obecność drogi . . . . .	6
4.1.2.2 Wysokość nad poziomem morza . . . . .	6
4.1.2.3 Status drogi osiedlowej . . . . .	6
4.2 Model . . . . .	7
4.2.1 Model przycinający (ClippingModel) . . . . .	7
4.2.2 Sposób tworzenia batchy . . . . .	8
4.2.2.1 implementacja końcowa . . . . .	8
4.2.2.2 implementacja pierwotna . . . . .	10
4.2.3 Statystyki procesu uczenia . . . . .	10
4.2.4 Architektura U-Net . . . . .	12
4.2.5 Architektura spłyconego U-Netu . . . . .	13
4.3 Algorytm uciągający <i>siatkę punktów</i> do grafu OSMnx . . . . .	13
4.4 Aplikacja demonstracyjna . . . . .	18
4.4.1 Funkcjonalności aplikacji: . . . . .	18
4.4.2 Aspekt wizualny . . . . .	19
<b>5 Wyniki</b>	<b>19</b>
5.1 Wielkość wycinka . . . . .	20
5.1.1 Model o wycinku 92 i nadmiarze 64 . . . . .	22
5.1.2 Model o wycinku 128 z nadmiarem 96 . . . . .	25

5.1.3	Model o wycinku 256 z nadmiarem 192	28
5.2	Paremtry wejściowe i wyjściowe	31
5.2.1	Model z pełnymi parametrami	31
5.3	Achitektury	32
5.4	Inne eksperymenty	33
<b>6</b>	<b>Wnioski</b>	<b>33</b>
6.1	Reprezentacja problemu	33
6.2	Przekazywanie kontekstu pomiędzy predykcjami poszczególnych wycinków	33
6.3	Zbiór danych	34
<b>7</b>	<b>Podział pracy</b>	<b>35</b>

# 1 Lider grupy

Grzegorz Lenarski (glenarski@student.agh.edu.pl)

## 2 Repozytorium z kodem źródłowym

<https://github.com/CRoadA/CRoadA>

## 3 Opis projektu

Celem projektu było stworzenie modelu generatywnego pozwalającego na otrzymywanie miejskich siatek dróg podobnych do rzeczywistych. Model taki mógłby okazać się użyteczny do poszerzania zbioru uczącego dla między innymi:

- modeli kierujących autonomicznymi pojazdami,
- modeli optymalizujących siatkę dróg miasta.

## 4 Sposób realizacji projektu

Projekt został zrealizowany w języku Python. Wykorzystane biblioteki można znaleźć w pliku requirements.txt w głównym katalogu projektu.

### 4.1 Zbiór uczący

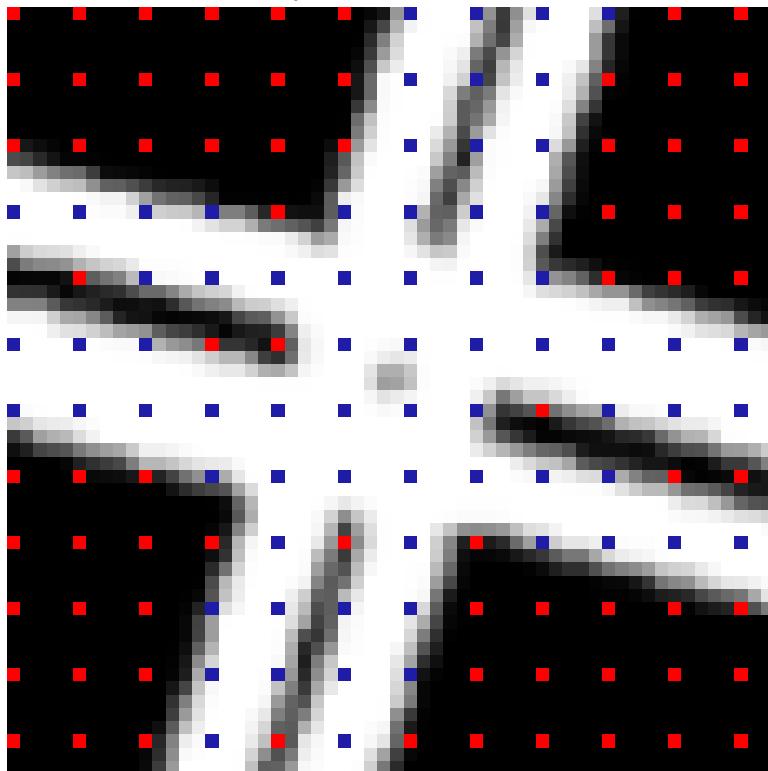
#### 4.1.1 Sposób reprezentacji danych wejściowych

Zbiór uczący składał się z informacji (wysokości, obecności drogi, statusu drogi osiedlowej) związanych z punktami na dwuwymiarowej kwadratowej *siatce punktów*, która jest zbiorem punktów wyznaczanych przez przecięcia linii poziomych (indeksowanych przez  $i$ ) oraz linii pionowych (indeksowanych przez  $j$ ). Kolejne linie zarówno pionowe, jak i poziome oddalone są od siebie o odległość  $d$  zwaną dalej gęstością siatki. Każda wielkość osadzana na tej siatce tworzy zatem macierz. Na przykładzie parametru obecności/nieobecności drogi związek pomiędzy topologią przestrzeni, a danymi został zobrazowany na rysunku 1b. Na tym etapie warto też rozróżnić dwa pojęcia:

- sieć dróg — opis przebiegu dróg w szczególności w formacie *Open Street Map*.
- siatka punktów — macierz wartości związanych z punktami w przestrzeni (w układzie opisanym powyżej).



(a) Pobrań z OpenStreetMap przykładowa siatka dróg wokół Akademii Górnictwo-Hutniczej w Krakowie.



(b) Skrzyżowanie dróg dwujezdniowych z lewego-górnego rogu sieci dróg zobrazowanej powyżej. Punkty zaznaczone na niebiesko ozna- czają obecność drogi, a na czerwono — jej brak.

Rysunek 1: Przykład dyskretyzacji pobranych danych z *Open Street Map*.

W takiej macierzy może zostać umieszczonych wiele wartości związań z tym samym położeniem, zatem każdy punkt  $(i, j)$  jest reprezentowany przez wektor wartości  $\mathbf{x}_{i,j}$ . Zbiór wszystkich wektorów  $\mathbf{x}_{i,j}$  tworzy macierz:

$$M = \begin{bmatrix} \mathbf{x}_{1,1} & \dots & \mathbf{x}_{1,j} \\ \vdots & \ddots & \vdots \\ \mathbf{x}_{i,1} & \dots & \mathbf{x}_{i,j} \end{bmatrix}, \quad (1)$$

która posłużyła jako wejście do modelu.

Ten sposób reprezentacji danych skutkował jednak bardzo dużym rozmiarem przetwarzanych danych, co skłoniło nas do przechowywania tych danych w plikach na dysku twardym na każdym etapie przetwarzania. Na potrzeby projektu zostały w związku z tym zdefiniowane dwa pojęcia:

- Segment — części *siatki punktów* na tyle małej, że mieści się w pamięci nawet w kilku kopiiach i mimo tego jej przetwarzanie samo w sobie nie wymaga szczególnie uwagi dotyczącej użytkowania pamięci. Wielkość tej początkowo ustalono na około  $5000 \times 5000$  komórek ważących ok. 300 MB. W celu przyspieszenia uczenia, którego jednym z problemów była wysoka intensywność pobierania danych, w dalszych testach powyższą wartość zmniejszono na obszary komórek  $500 \times 500$ . W przeciwnym wypadku predykcja pojedynczego *wycinka* wielkości  $256 \times 256$  (768 kB) wymagałaby pobrania 1 – 4 segmentów każdy o wielkości ok. 300 MiB.
- Obszar siatki punktów — dowolnie duża część *siatki punktów*, która na żadnym etapie nie jest przetwarzana jako całość lecz jedynie *segmentami*.

Wszelkie operacje w projekcie zostały zaprojektowane z uwzględnieniem podziału *siatki punktów* na *segmenty*, dzięki czemu wielkość dostępnej pamięci nie ogranicza wielkości przetwarzanego *obszaru siatki punktów*. Dla ułatwienia problemu, skonstruowano także narzędzia do ewentualnej resegmentacji danych - tj. zmiany wielkości segmentów bez konieczności ich ponownego pobierania - przydatne w przypadku dużej wielkości plików (dla przykładu: przekraczającej w sumie 24 GB przy jedenastu pobranych siatek miast).

#### 4.1.2 Informacje zawarte w zbiorze i ich źródła

Modelowi przekazano następujące dane:

- obecność drogi (wartość boolowska),
- wysokość nad poziomem morza (liczba zmiennoprzecinkowa),
- status drogi osiedlowej (wartość boolowska).

W rzeczywistej implementacji do zapisu każdej z wartości wykorzystano typ 32-bitowej liczby zmiennoprzecinkowej. Decyzja ta została podjęta w celu uproszczenia kodu źródłowego (ma to szczególne znaczenie w kontekście *obszarów siatki punktów i segmentów*).

**4.1.2.1 Obecność drogi** Parametr jest kluczowy do uczenia modelu realizacji założonego zadania, gdyż jest to główny oczekiwany rezultat predykcji. Dane dotyczące rozmieszczenia dróg zostały pobrane z *Open Street Map* za pomocą biblioteki OSMnx [[2]]. Biblioteka ta reprezentuje drogi jako zbiory wierzchołków grafu o zadanych współrzędnych, co wymagało nałożenia na nią *siatki punktów* i odpowiedniego wyliczenia wartości w punktach. Pobrane dane zawierały jednak wiele nieścisłości co do szerokości dróg, więc dla jezdni niezawierających atrybutu 'width' ani 'lanes' uśredniono szerokość drogi na podstawie jej typu. W uzyskanych danych znalazło się jednak bardzo dużo dróg osiedlowych, które charakteryzowały się promieniami skrętu mniejszymi niż 30 m. Na początku projektu, poczyniono założenie dotyczące późniejszej walidacji modelu, gdyż drogi o ograniczeniu prędkości większym bądź równym 30 km/h nie mogą wykazywać promieni skrętu poniżej 30 m. Celem umożliwienia sprawdzenia poprawności działania modelu zdecydowano się rozszerzyć dane wyjściowe z modelu o dodatkową wartość *statusu drogi*, w której model będzie klasyfikował drogi jako nie podlegające temu ograniczeniu. Wobec ostatecznych wyników modelu zdaje się to jednak niestety nie mieć większego znaczenia.

**4.1.2.2 Wysokość nad poziomem morza** Wielkość ta jest istotna w procesie uczenia modelu dopasowania *sieci dróg* do ukształtowania terenu oraz przekształcania terenu pod drogi tak, by zachować parametry. Dane pozyskano ze zbioru *SRTM* za pomocą biblioteki srtm [[1]].

**4.1.2.3 Status drogi osiedlowej** Wartość ta informuje o tym, czy model klasyfikuje daną drogę, jako osiedlową, co wyłącza ją z ograniczenia minimalnego promienia skrętu. Dane te zostały uzyskane z rozkładu przestrzennego *obecności drogi* — drogi o dostatecznie małym promieniu skrętu zostały tak zaklasyfikowane. Rozważano też odczyt tych danych z *Open Street Map*, jednak okazało się, że liczba dróg zaklasyfikowanych jako osiedlowe jest tak duża, że znacząco rozrzedziłoby to zbiór danych uczących, co w konsekwencji prawdopodobnie skutkowałoby problemami z uczeniem modelu. Wobec ostatecznych wyników modelu parametr ten nie był jednak wykorzystywany w prezentowanych poniżej modelach.

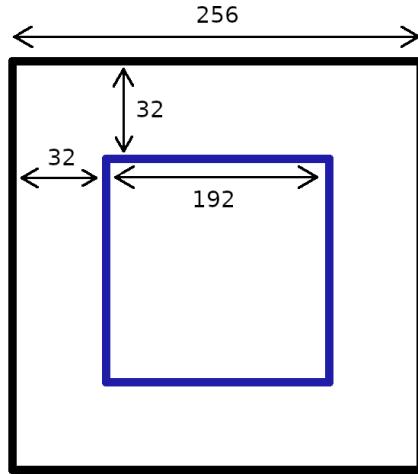
## 4.2 Model

Model został zaprojektowany jako interfejs służący do szkolenia i predykcji w oparciu o *obszary siatki punktów*. Ostatecznie jednak zaimplementowano tylko jedną klasę, która realizuje jego funkcjonalność — *model przycinający*.

### 4.2.1 Model przycinający (ClippingModel)

Realizacja interfejsu oferującego możliwość predykcji dla dowolnego rozmiaru wejściowego wymagało przetworzenia wejścia przed przewidywaniem w sposób silnie zależny od architektury sieci (a właściwie od rozmiaru jej wejścia). W modelu przycinającym zostało to zrealizowane poprzez cięcie wejściowego *obszaru siatki punktów* na wycinek, które są spójnymi, kwadratowymi fragmentami *obszaru siatki punktów* o wymiarach zgodnych z wejściem do sieci neuronowej. Predykcja danego *wycinka* skutkuje jednak zwróceniem macierzy o boku pomniejszonym o nadmiar modelu. Nadmiar modelu jest kontekstem przekazywanym sieci neuronowej, dzięki któremu może ona zapewnić spójność pomiędzy *wycinkami*. Podczas predykcji na wejściu podawanego są fragmenty z już przewidzianych *wycinków* powyżej i z lewej strony obecnie przewidywanego *wycinka*, dzięki czemu model może „kontynuować” wcześniej prowadzoną predykcję. *Wycinki* z ostatnich rzędów i kolumn są wyrównywane odpowiednio dolnymi krawędziami do dolnej krawędzi *obszaru siatki punktów* i prawymi krawędziami do prawej krawędzi *obszaru siatki punktów*. Dzięki temu unika się problemu paddingu danych, choć odbywa się to za cenę podwójnej predykcji fragmentu, gdzie sąsiednie *wycinki* się nakładają.

Przykładowo model o *wycinku* szerokości 256 i *nadmiarze* 64 przyjmuje wejście o wielkości 256x256 oraz wyjście o wielkości 192x192. Wyjście to odpowiada fragmentowi wejścia zawartemu pomiędzy punktami o współrzędnych (32, 32) i (224, 224), traktując pozostały brzeg jako kontekst (jak na rysunku 2).



Rysunek 2: Ilustracja relacji przestrzennych pomiędzy wejściem, a wyjściem z modelu.

Dodatkowo na wejściu model przyjmuje dodatkową wartość boolowską czy przewidywane, która wskazuje modelowi, czy w danym punkcie oczekiwana jest predykcja. Rozwiążanie to powinno dać modelowi możliwość predykcji obszarów o bardziej złożonych kształtach niż prostokątny. Wobec wyników modelu wszystkie zastosowania tego parametru sprowadziły się do ustawienia wartości „prawda” dla każdego punktu wejścia odpowiadającego punktowi wyjścia (wyjście modelu nie było nigdy zauważane tą metodą). Fakt ten dotyczył zarówno uczenia, jak i walidacji modelu.

Modelem dokonującym predykcji pojedynczego *wycinka* była konwolucyjna sieć neuronowa. Była ona uczona w procesie uczenia samonadzorowanego (właściwie ang. *Autoassociative self-supervised learning*).

#### 4.2.2 Sposób tworzenia batchy

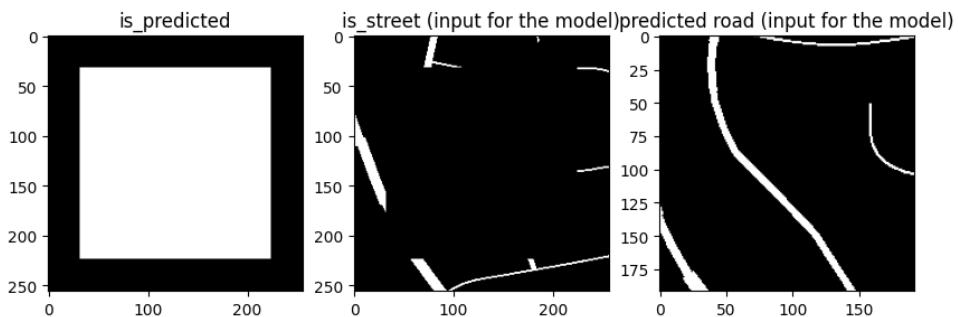
**4.2.2.1 implementacja końcowa** Użycie zaimplementowanego generatora danych sprowadza się do wykonania kodu zaprezentowanego na listingu

```
data_gen = clipping_sample_generator(
    grid_managers=files ,
    cut_sizes=[(256, 256)],
    clipping_size=256,
    input_surplus=64,
    input_third_dimension=2,
    output_third_dimension=1,
)
X_batch, y_batch = next(data_gen)
```

Generatorowi, jako argumenty, podawane są kolejno:

- pliki w formacie klasy GridManager (*segmentowane* — jak opisano sekcji 4.1.1)
- wymiary *obszarów siatki punktów* które są następnie dzielone na *wycinki*, na których model się uczy (w wersji finalnej używano jedynie rozmiaru *wycinka* — zrównując jego wymiar z *obszarem siatki punktów*).
- wymiary *wycinków* wejścia do modelu
- wielkość *nadmiaru* (brzeg z kontekstem ramki)
- liczba kanałów w trzecim wymiarze wejścia (dla wartości 2 — tylko wartości *czy przewidywane* oraz *obecność drogi*). Odpowiednie kanały są dodawane albo pomijane w procesie generowania danych. Kolejnymi kanałami są *wysokość nad poziomem morza* oraz *status drogi osiedlowej*.
- liczba kanałów w trzecim wymiarze wyjścia (dla wartości 1 — tylko wartość *obecność drogi*).

Przedstawiony powyżej generator jest przekazywany metodzie `data.Dataset.from_generator()` z modułu Tensorflow używanego do pobierania batchy z generatora i dalszego szkolenia modelu. Taka implementacja umożliwia zastosowanie wbudowanego mechanizmu prefetchingu, a więc generowania danych z wyprzedzeniem momentu ich przydatności. Przykładowe dane generowane w wersji końcowej przedstawiono na rysunku 3. Jak widać, w generatorze przeprowadzony jest cały proces obróbki danych — a więc i zerowanie odpowiednich fragmentów wycinka.



Rysunek 3: Przykład wygenerowanego kanału z wartością *czy przewidywane* = 1 (białe piksele), *obecność drogi* z wyciętym środkiem na podstawie maski *czy przewidywane* oraz oczekiwanym wyjściem w procesie uczenia.

**4.2.2.2 implementacja pierwotna** W pierwotnie zaimplementowanej wersji, do powyżej zaprezentowanego zadania, używano dwóch oddzielnych klas — BatchSequence oraz ClippingSequence opartych na logice obecnej w rozwiązaniu keras.utils.Sequence z modułu Tensorflow. Rozwiązanie to jednak zawierało podwójną logikę generowania danych - najpierw losowano *obszar siatki punktów* z wybranych plików o wymiarach zadanych na wejściu inicjalizacji klasy BatchSequence — w celu podzielenia danych wejściowych modelu na fragmenty zdolne do jednocięsnego przetwarzania w pamięci RAM, aby następnie generować mniejsze wycinki interfejsem klasy ClippingSequence (której wejściem była odpowiednia instancja klasy BatchSequence). W momencie zrównania wymiarów podawanych do obu klas, postanowiono od nowa zaimplementować jeden spójny mechanizm generowania gotowych wycinków — zaprezentowany powyżej jako rozwiązanie końcowe. Wcześniejszego implementacji okazała się przesadnie skomplikowana i mało wydajna w generowaniu dużych liczb batchy. Blokowała proces uczenia, generując bardzo wysoki czas oczekiwania na wyszkolenie modelu — nawet przy testowych, ograniczonych parametrach szkolenia, uniemożliwiających jakkolwiek istotny efekt uczenia.

#### 4.2.3 Statystyki procesu uczenia

1. W modelu przycinającym użyto dla poszczególnych wartości następujących strat i metryk:

- Dla wartości **obecność drogi**:
  - **\_dice\_coef** - mierząca podobieństwo dwóch masek dla segmentacji binarnej - im bliższa wartość jedynce, tym lepiej. Jej zastosowanie wynika z faktu, że ma ona być ona użyteczna przy silnej nierównowadze klas (np. mało pikseli drogi w porównaniu do tła). Strata **\_dice\_loss** wyznaczana jest jako różnica jedynki z tym współczynnikiem.
  - Dla wartości **obecność drogi** użyto jednak straty nieco bardziej skomplikowanej: **FocalDiceLoss** będącej sumą dwóch składników: wspomnianej **\_dice\_loss** pomnożonej przez wagę podawaną na wejściu z wartością **Binary Focal Cross-Entropy** o parametrach  $\alpha = 0.75$  oraz  $\gamma = 2.0$ . Wagę straty na wejściu określono jako 10 dla zrównoważenia znacznie większych różnic w danych dotyczących **wysokości nad poziomem morza**.
  - Oprócz powyższej **\_dice\_coef** użyto wspólnie metryk:
    - \* **AUC pod krzywą Precision-Recall** - używane przy niezbalansowanych danych
    - \* **Precision z progiem 0.5** - traktujące piksele powyżej wartości 0.5 jako klasy pozytywne

\* Recall z progiem 0.5.

- Odmienny problem reprezentuje wyjście **wysokość nad poziomem morza**, gdyż jest to problem nie klasyfikacji, ale regresji. Tak więc:
    - za strategię przyjęto **Hubera** (połączenie MSE i MAE), a jej wagę określono jako wartość 1
    - posługiwano się dwoma metrykami:
      - \* **Mean Absolute Error** (MAE)
      - \* **Root Mean Squared Error** (RMSE).
  - Ze względu na podobieństwo problemu klasyfikacji, dla wartości **status drogi osiedlowej** przyjęto parametry identyczne jak wymienione wyżej dla wartości **obecność drogi**.
2. Dla wyjść sieci neuronowej, wewnętrzach architektury (o których niżej) przyjęto następujące funkcje aktywacji — typowe dla poszczególnych problemów:
- funkcja **sigmoid** dla wartości **obecność drogi** oraz **status drogi osiedlowej** — typowa dla problemu klasyfikacji
  - funkcja **linear** dla wartości **wysokość nad poziomem morza** — typowa dla problemu regresji bez ograniczeń.
3. W komplikacji modelu użyto optymalizatora **adam**.

#### 4.2.4 Architektura U-Net

Architektura U-Net jest znaną architekturą stworzoną do segmentacji obrazów. Składa się ona z zawiązających warstw enkodera, wąskiego gardła i rozszerzających warstw dekodera. Dodatkowo analogiczne warstwy enkodera i dekodera są powiązane połączeniami typu „skip”. Za jej opis prawdopodobnie najlepiej posłuży wykorzystany kod Python:

```
inputs = tf.keras.layers.Input(shape=(clipping_size, clipping_size, input_third_dimension))

# Encoder
c1 = tf.keras.layers.Conv2D(32, 3, activation="relu", padding="same")(inputs)
c1 = tf.keras.layers.Conv2D(32, 3, activation="relu", padding="same")(c1)
p1 = tf.keras.layers.MaxPooling2D()(c1)

c2 = tf.keras.layers.Conv2D(64, 3, activation="relu", padding="same")(p1)
c2 = tf.keras.layers.Conv2D(64, 3, activation="relu", padding="same")(c2)
p2 = tf.keras.layers.MaxPooling2D()(c2)

c3 = tf.keras.layers.Conv2D(128, 3, activation="relu", padding="same")(p2)
c3 = tf.keras.layers.Conv2D(128, 3, activation="relu", padding="same")(c3)
p3 = tf.keras.layers.MaxPooling2D()(c3)

# Bottleneck
b = tf.keras.layers.Conv2D(256, 3, activation="relu", padding="same")(p3)
b = tf.keras.layers.Conv2D(256, 3, activation="relu", padding="same")(b)
# Decoder
u3 = tf.keras.layers.UpSampling2D()(b)
u3 = tf.keras.layers.concatenate([u3, c3])
c4 = tf.keras.layers.Conv2D(128, 3, activation="relu", padding="same")(u3)
c4 = tf.keras.layers.Conv2D(128, 3, activation="relu", padding="same")(c4)

u2 = tf.keras.layers.UpSampling2D()(c4)
u2 = tf.keras.layers.concatenate([u2, c2])
c5 = tf.keras.layers.Conv2D(64, 3, activation="relu", padding="same")(u2)
c5 = tf.keras.layers.Conv2D(64, 3, activation="relu", padding="same")(c5)

u1 = tf.keras.layers.UpSampling2D()(c5)
u1 = tf.keras.layers.concatenate([u1, c1])
c6 = tf.keras.layers.Conv2D(32, 3, activation="relu", padding="same")(u1)
c6 = tf.keras.layers.Conv2D(32, 3, activation="relu", padding="same")(c6)

# Crop to remove surplus
crop = clipping_surplus // 2
c6 = tf.keras.layers.Cropping2D(cropping=((crop, crop), (crop, crop)))(c6)
```

#### 4.2.5 Architektura spłyconego U-Netu

Architektura powstała poprzez usunięcie jednej warstwy z obu stron względem architektury *U-Net* przez co dochodzi do zmniejszenia liczby filtrów w środku. Za jej opis prawdopodobnie najlepiej posłuży wykorzystany kod Pythona:

```
inputs = tf.keras.layers.Input(shape=(clipping_size, clipping_size, input_third_dimension))

# Encoder
c1 = tf.keras.layers.Conv2D(32, 3, activation="relu", padding="same")(inputs)
c1 = tf.keras.layers.Conv2D(32, 3, activation="relu", padding="same")(c1)
p1 = tf.keras.layers.MaxPooling2D()(c1)

c2 = tf.keras.layers.Conv2D(64, 3, activation="relu", padding="same")(p1)
c2 = tf.keras.layers.Conv2D(64, 3, activation="relu", padding="same")(c2)
p2 = tf.keras.layers.MaxPooling2D()(c2)

# Bottleneck
b = tf.keras.layers.Conv2D(128, 3, activation="relu", padding="same")(p2)
b = tf.keras.layers.Conv2D(128, 3, activation="relu", padding="same")(b)
# Decoder
u2 = tf.keras.layers.UpSampling2D()(b)
u2 = tf.keras.layers.concatenate([u2, c2])
c5 = tf.keras.layers.Conv2D(64, 3, activation="relu", padding="same")(u2)
c5 = tf.keras.layers.Conv2D(64, 3, activation="relu", padding="same")(c5)

u1 = tf.keras.layers.UpSampling2D()(c5)
u1 = tf.keras.layers.concatenate([u1, c1])
c6 = tf.keras.layers.Conv2D(32, 3, activation="relu", padding="same")(u1)
c6 = tf.keras.layers.Conv2D(32, 3, activation="relu", padding="same")(c6)

# Crop to remove surplus
crop = clipping_surplus // 2
c6 = tf.keras.layers.Cropping2D(cropping=((crop, crop), (crop, crop)))(c6)
```

### 4.3 Algorytm uciążlający siatkę punktów do grafu OSMnx

W związku ze stosowanym podziałem obszarów siatki punktów na segmenty, algorytm tworzący graf dróg został podzielony na dwie części:

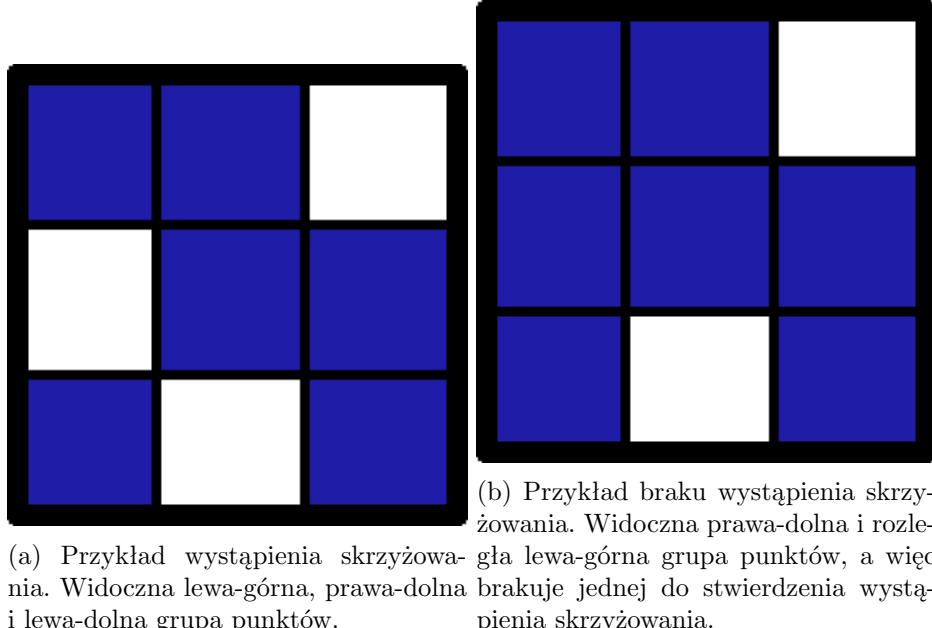
- dolna część algorytmu - dokonującą przetwarzania pojedynczego fragmentu siatki punktów na zagregowane dane wykrytych obiektów (dróg i skrzyżowań),
- górną część algorytmu - wyznaczającą fragmenty siatki punktów dla części dolnej oraz scalającą odtworzone z nich dane.

Fragmenty siatki punktów dla części dolnej są wyznaczane w taki sposób, by na siebie nachodziły pasem o grubości jednego punktu. Jest to użyteczne w dalszym scalaniu wyników. Nachylenia dróg są mierzone na siatce punktów, a więc przez dolną część, a promienie skrętu — przez górną

Dolna część algorytmu działa w następujących krokach:

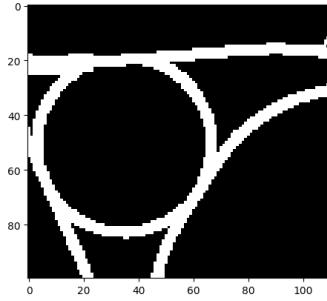
1. rozszerzenie fragmentu o obrys z punktów będących drogą (dzięki temu uzyskany później szkielet osiąga krawędzie rozpoznawanego fragmentu).
2. szkieletyzacja algorytmem Zhang [[3], [4]], który gwarantuje ciągły szkielet o grubości 1.
3. pozbawienie fragmentu ramki dodanej w kroku 1.
4. wykrycie skrzyżowań — odbywa się to poprzez znalezienie punktów będących drogami, które w sąsiedztwie Moore'a mają co najmniej 3 rozłączne grupy sąsiadujących punktów będących drogami (jak na rysunku 4). Jako skrzyżowanie kwalifikowane są wtedy wszystkie punkty otoczenia Moore'a.
5. rozdzielenie dróg poprzez usunięcie punktów należących do skrzyżowań od szkieletu
6. utworzenie odpowiednich powiązań pomiędzy obiektami dróg, a skrzyżowań.

Efekty działania dolnej części algorytmu zostały zaprezentowane na rysunkach 5 i 6.

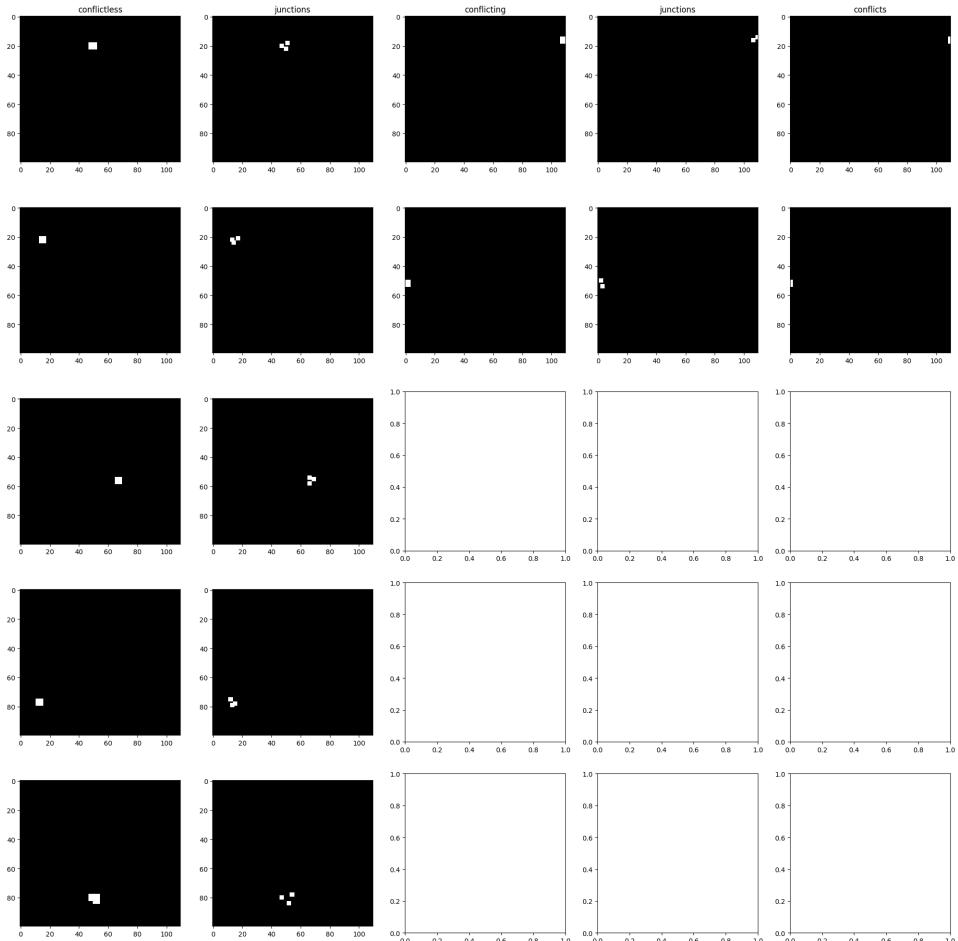


Rysunek 4: Przykłady otoczenia punktu weryfikowanego pod kątem wystąpienia skrzyżowania. Na biało zaznaczono piksele mające wartość *obecność drogi* jako fałsz, a na niebiesko — jako prawda.

W oparciu o te informacje *górną część* dokonuje scalenia części drogi znajdującej się w wielu segmentach, wykorzystując fakt, że obie części mają wspólne (lub bardzo blisko siebie położone) punkty, gdyż przetwarzane fragmenty nieznacznie na siebie nachodziły — obecnie używana „zakładka” ma grubość 1 piksela.

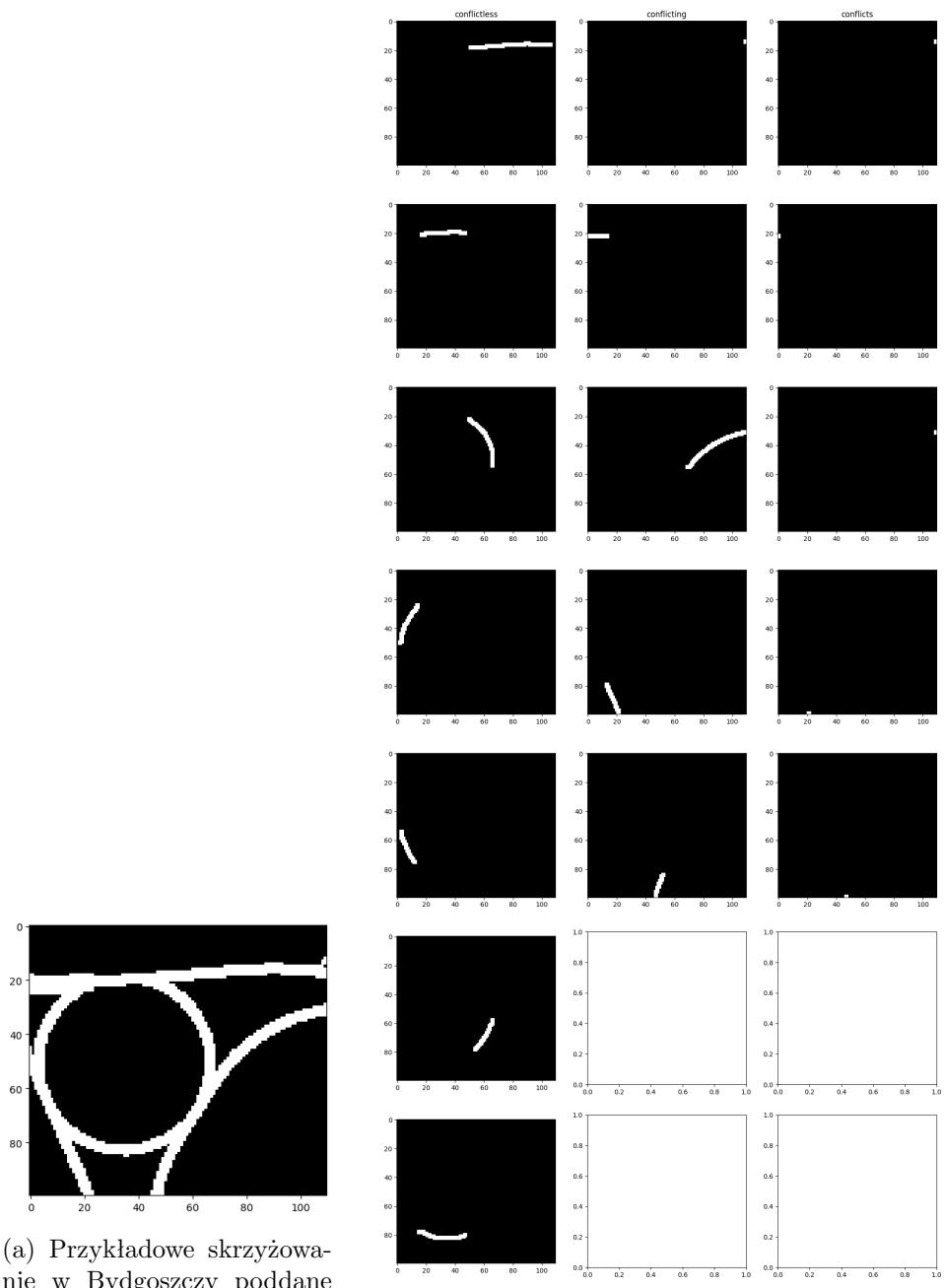


(a) Przykładowe skrzyżowanie w Bydgoszczy poddane analizie.



(b) Wykryte skrzyżowania.

Rysunek 5: Wykryte skrzyżowania dla przykładowego ronda. W kolejnych kolumnach: 1. skrzyżowania bez konfliktu z brzegiem, 2. punkty ich połączenia z drogami, 3. wykryte skrzyżowania konflikujące z brzegiem, 4. punkty ich połączenia z drogami, 5. punkty konfliktu z brzegiem. Obrazy na rysunku b) zostały poddane dylatacji dla lepszej widoczności.



Rysunek 6: Wykryte drogi dla przykładowego ronda. W kolejnych kolumnach: 1. wykryte drogi bez konfliktów z brzegiem, 2. wykryte drogi konflikujące z brzegiem, 3. ich punkty konfliktu. Obrazy na rysunku b) zostały poddane dylatacji dla lepszej widoczności.

## 4.4 Aplikacja demonstracyjna

Głównym punktem wejścia do aplikacji oraz elementem spajającym wszystkie komponenty projektu jest klasa `Application`, opierająca się na bibliotece PyQt6. Stanowi ona warstwę kontrolera, odpowiedzialną za inicializację zasobów, zarządzanie cyklem życia aplikacji oraz obsługę zdarzeń wywoływanych przez użytkownika.

Kluczowym aspektem architektonicznym rozwiązania jest zastosowanie biblioteki `qasync`, która powoduje, że czasochłonne operacje wejścia-wyjścia, takie jak pobieranie danych geograficznych czy operacje predykcji, nie powodują zamrażania interfejsu użytkownika, co znaczaco wpływa na wygodę korzystania z aplikacji.

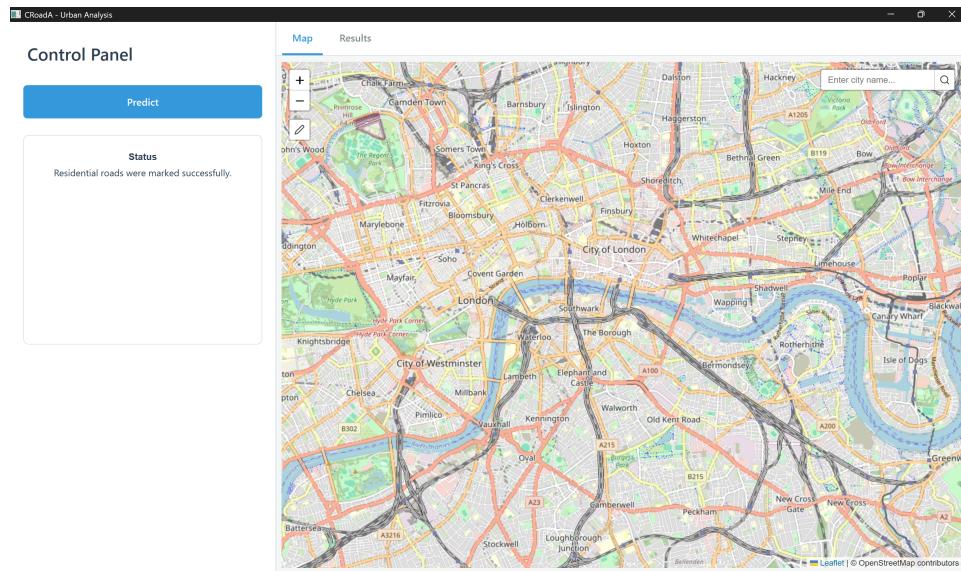
### 4.4.1 Funkcjonalności aplikacji:

Aplikacja wykorzystuje moduły, które wyróżnione były w poprzednich sekcjach do realizacji poniższych funkcjonalności:

- **Wyszukiwanie miast na interaktywnej mapie** – wykorzystanie silnika `Leaflet` do nawigacji po globalnej mapie. Funkcjonalność obejmuje dynamiczne centrowanie widoku na zadanym obszarze. Dane o lokalizacji są synchronizowane z obiektem `WebBridge`, co pozwala na automatyczne mapowanie zapytań tekstowych na współrzędne geograficzne.
- **Wybór obszaru predykcji** – użytkownik może ręcznie zdefiniować obszar na mapie, który jest następnie konwertowany na format `GeoJSON` i przesyłany do za pośrednictwem `QWebChannel`. Pobierana jest siatka punktów z danego obszaru i zapisywana w odpowiednim pliku. System zawiera mechanizm potwierdzania wyboru poprzez pop-up zintegrowany z warstwą graficzną mapy.
- **Predykcja** – uruchomienie predykcji wczytanego do aplikacji, wytrenowanego modelu. Możliwość predykcji następuje jedynie po zapisie pliku z danymi zaznaczonego obszaru.
- **Wizualizacja wyników analizy** – automatyczna konwersja macierzy jak wynik predykcji modelu na format graficzny, co pozwala na podgląd wygenerowanych siatek punktów.
- **System powiadomień i kontroli stanu** – wdrożenie wizualnego wskaźnika powiadomień informującego o gotowych wynikach oraz dynamiczne zarządzanie stanem przycisków kontrolnych w zależności od postępu prac.

#### 4.4.2 Aspekt wizualny

Interfejs graficzny aplikacji został zaprojektowany z naciskiem na przejrzystość i ergonomię użytkowania. Układ głównego okna składa się z panelu kontroli, umożliwiającego predykcję oraz śledzenie aktualnego statusu zrealizowanej zapisu pliku oraz działania modelu.



Rysunek 7: Widok aplikacji po zapisie zaznaczonego obszaru

## 5 Wyniki

W celu znalezienia optymalnego modelu sprawdzono wybrane wartości pod następującymi kątami:

- wielkość *wycinka* i jego *nadmiaru*,
- architektury modelu,
- danych na wejściu i wyjściu modelu,

Szkolenie było prowadzone przeprowadzone na zbiorach uczących generowanych z 11 miast:

- Gdańsk,
- Bydgoszcz,
- Lublin,

- Gdynia,
- Radom,
- Toruń,
- Świętochłowice,
- Koszalin,
- Mielec,
- Chorzów,
- Rybnik.

Z kolei zbiór testowy był generowany z 4 miast:

- Elbląg,
- Płock,
- Rzeszów,
- Słupsk.

### 5.1 Wielkość wycinka

Dla ustalenia warunków i możliwości badania samego wpływu wielkości *wycinka modelu* oraz *nadmiaru modelu* wszystkie modele zostały zbudowane w takimi samymi pozostały parametrami, tj.:

- w oparciu o architekturę *spłyconego U-Netu*,
- jedynie z danymi *obecności drogi*.

Aby uzyskać model jakkolwiek odnoszący się zadanego problemu sprawdzono kilka konfiguracji wymienionych w tabeli 1. Dzięki nim udało się znaleźć takie parametry, które zaczęły przynosić pewien (choć niestety niewielki) efekt i zostały zaprezentowane w sekcjach poniżej.

Tabela 1: Tabela weryfikowanych konfiguracji modelu, które nie spełniły założonych zadań.

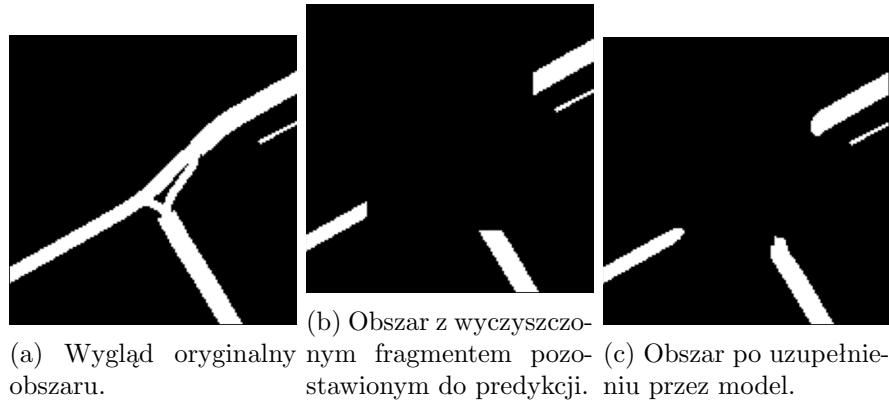
wielkość wycinka	nadmiar	przyjmowane wartości	liczba epok (po 100–150 predykcji)
64	32	obecność drogi, wysokość, status drogi osiedlowej	119
92	64	obecność drogi, wysokość, status drogi osiedlowej	156
92	64	obecność drogi	184
128	64	obecność drogi, wysokość, status drogi osiedlowej	132
128	64	obecność drogi	88
256	64	obecność drogi, wysokość, status drogi osiedlowej	312
256	64	obecność drogi	54
512	128	obecność drogi	166

W wyniku tych eksperymentów postanowiono, że należy sprawdzić, czy nie warto zwiększać *nadmiaru modelu*. Taka zmiana podejścia przyniosła efekt w postaci pewnych, niewielkich zdolności modelu w uciąglaniu dróg, które dostaje jako kontekst.

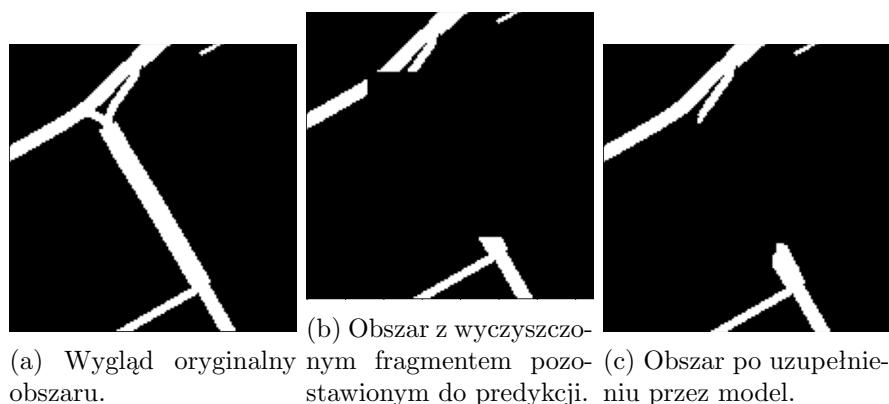
### 5.1.1 Model o wycinku 92 i nadmiarze 64

Pierwszym modelem, który przyniósł rezultaty warte wspomnienia był model o *wycinku 92 i nadmiarze 64*. Pierwsza predykcja (ilustracje 8) pokazuje jedynie niewielkie przedłużenie rozłączonych gałęzi skrzyżowania, jednak po przesunięciu obszaru predykcji udało się uzyskać znacznie lepszy wynik (ilustracje 9). W jej wypadku model odtworzył skrzyżowanie w dość nietypowy sposób, jednak udało się mu odtworzyć ciągłość drogi głównej. Analizując przebieg predykcji dla poszczególnych wycinków (pokazanych na ilustracjach 10), można dojść do wniosku, że sukces ten był prawdopodobnie efektem tego, że predykcja ta została zrealizowana w ramach pojedynczego wycinka.

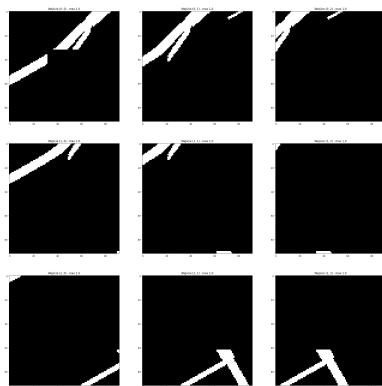
parametr modelu	wartość
architektura sieci neuronowej	<i>słycony U-Net</i>
wielkość wycinka	92
nadmiar modelu	64
wykorzystywane dane wejściowe	obecność drogi
zwracane dane	obecność drogi
czas szkolenia	135 epok po 150 kroków
końcowy współczynnik _dice_coef dla zbioru testowego	0,8825
końcowy współczynnik _dice_coef dla zbioru uczącego	0,9268
końcowa metryka AUC P-R dla zbioru testowego	0,9749
końcowa metryka Precision dla zbioru testowego	0,9220
końcowa metryka Recall dla zbioru testowego	0,9409



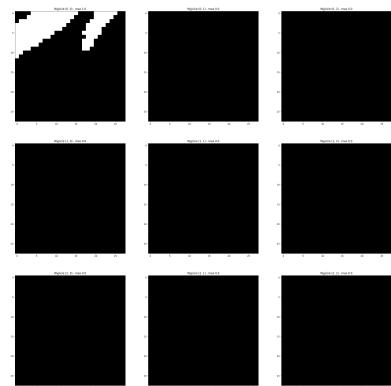
Rysunek 8: Pierwsza predykcja wykonana dla fragmentu Bydgoszczy, będącej elementem zbioru uczącego.



Rysunek 9: Pierwsza predykcja wykonana dla fragmentu Bydgoszczy, będącej elementem zbioru uczącego.



(a) Wejścia.



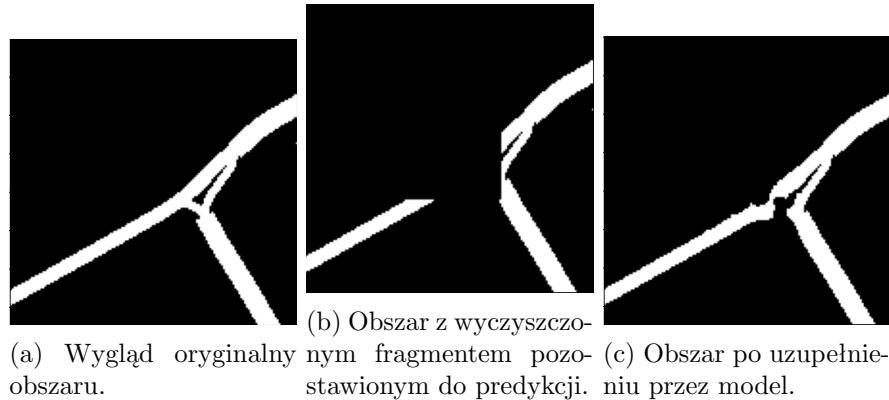
(b) Wyjścia.

Rysunek 10: Wejścia do i wyjścia z modelu związane z predykcjami poszczególnych wycinków w toku predykcji na fragmencie Bydgoszczy zobrazowanej na ilustracjach 9. Warto przypomnieć, że w *modelu przycinającym* wejścia nie są rozłączne, ale wyjścia już co do zasad tak.

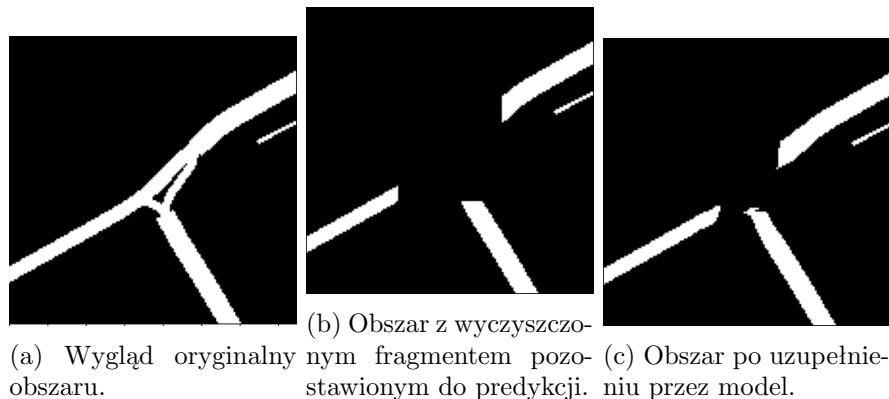
### 5.1.2 Model o wycinku 128 z nadmiarem 96

parametr modelu	wartość
architektura sieci neuronowej	<i>spłycony U-Net</i>
wielkość wycinka	128
nadmiar modelu	96
wykorzystywane dane wejściowe	obecność drogi
zwracane dane	obecność drogi
czas szkolenia	102 epoki po 150 kroków
końcowa metryka <code>_dice_coef</code> dla zbioru uczącego	0,9140
końcowa metryka <code>_dice_coef</code> dla zbioru testowego	0,8582
końcowa metryka AUC P-R dla zbioru testowego	0,9526
końcowa metryka Precision dla zbioru testowego	0,8943
końcowa metryka Recall dla zbioru testowego	0,9288

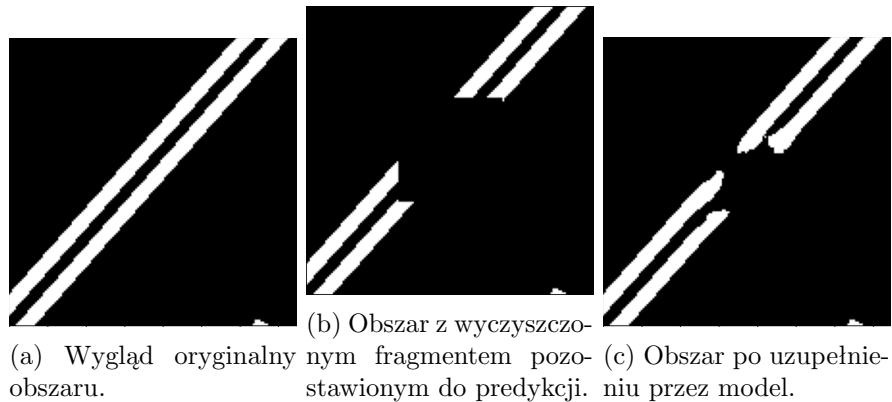
Wszystkie przedstawione fragmenty poddane predykcji mieszczą w sobie 2x2 *wycinki*. Na ilustracjach 11 można zaobserwować skuteczne uciążlenie skrzyżowania. Nie przypomina ono naturalnego wyglądu dróg miejskich, jednak w tym wypadku udało się uzyskać połoczenie, co nie jest oczywiste w tym kontekście, że jej obie strony były generowane w przy okazji innych *wycinków* (choć częściowo nakładających się). Jednak ilustracje 12 pokazują już predykcję obszaru przesuniętego o 20 metrów na wschód — tak mała zmiana warunków doprowadziła już rozerwanie połączenia pomiędzy dwoma przeciwnymi końcami tej samej drogi. Ostatnia predykcja została wykonana w oparciu o miasto Rzeszów, które nie było nigdy wykorzystywane w toku uczenia modelu. W tym przypadku model nie zdążył uciągnąć prostej, dwujezdniowej drogi, jednak ich przedłużenie zdaje się zmierzać w tym kierunku, a więc można mówić o pewnych zdolnościach do uogólniania. Wyniki tej predykcji zostały przedstawione na ilustracjach 13



Rysunek 11: Pierwsza predykcja wykonana dla fragmentu Bydgoszczy, będącej elementem zbioru uczącego.



Rysunek 12: Druga predykcja wykonana dla fragmentu Bydgoszczy, będącej elementem zbioru uczącego.

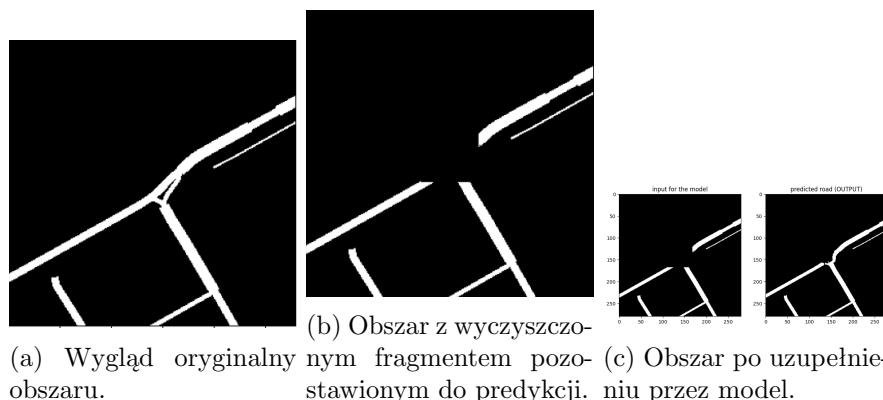


Rysunek 13: Predykcja wykonana dla fragmentu Rzeszowa, będącego elementem zbioru walidacyjnego.

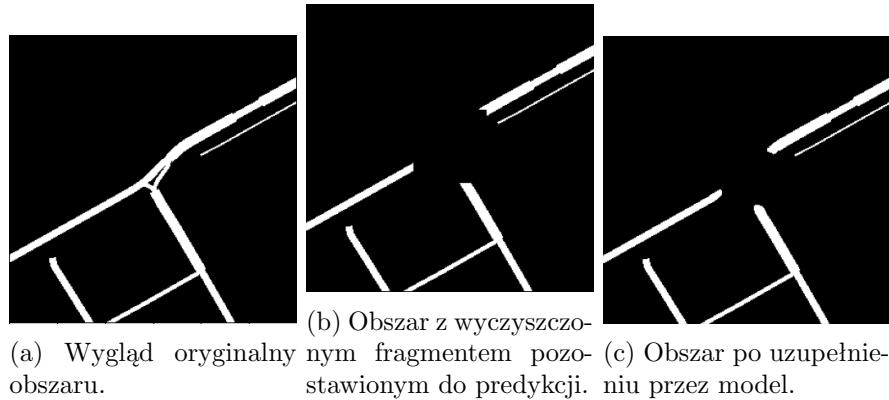
### 5.1.3 Model o wycinku 256 z nadmiarem 192

parametr modelu	wartość
architektura sieci neuronowej	<i>spłycony U-Net</i>
wielkość wycinka	256
nadmiar modelu	192
wykorzystywane dane wejściowe	obecność drogi
zwracane dane	obecność drogi
czas szkolenia	102 epoki po 150 kroków
końcowa metryka <code>_dice_coef</code> dla zbioru uczącego	0,8177
końcowa metryka <code>_dice_coef</code> dla zbioru testowego	0,8955
końcowa metryka AUC P-R dla zbioru testowego	0,9520
końcowa metryka Precision dla zbioru testowego	0,8818
końcowa metryka Recall dla zbioru testowego	0,9366

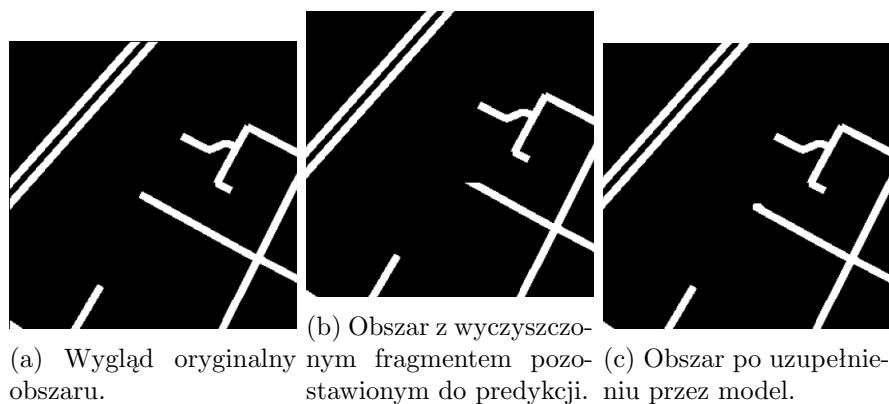
Model nie wydaje się być znacznie lepszy od poprzednich, choć jako jedyny był uczyony tylko do pierwszego early stoppingu. Poprzednie były uczone później dwukrotnie do zatrzymania według tego samego kryterium. Nie mniej jednak, tutaj warunek zatrzymania został spełniony wyjątkowo szybko, co sugeruje, że model mógłby już popaść w przeuczenie. W każdym razie na uwagę zasługuje chyba tylko przypadkiem wykonana predykcja przedstawiona na rysunku 17, która ilustruje przypadek, kiedy predykcje wykonywane przez model są uzasadnione.



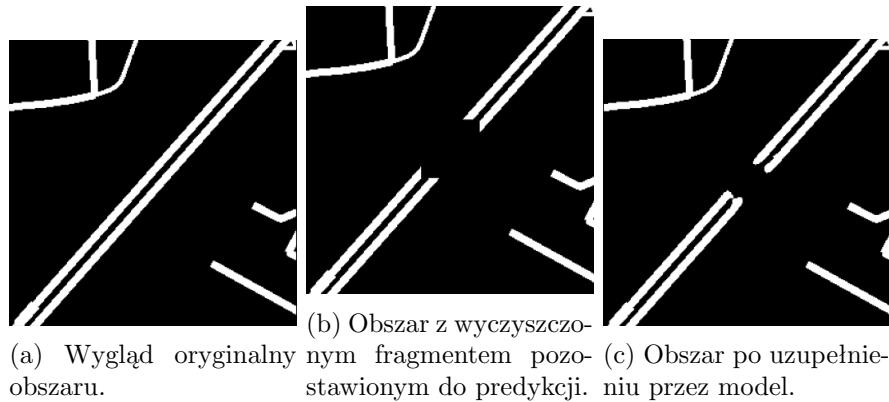
Rysunek 14: Pierwsza predykcja wykonana dla fragmentu Bydgoszczy, będącej elementem zbioru uczącego.



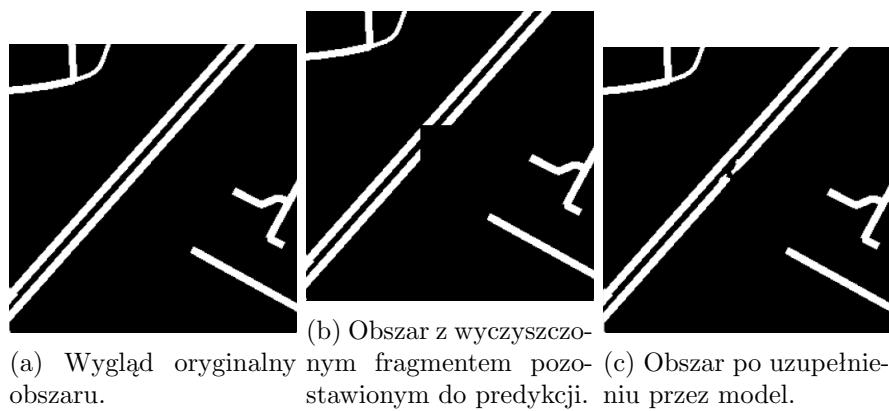
Rysunek 15: Druga predykcja wykonana dla fragmentu Bydgoszczy, będącej elementem zbioru uczącego.



Rysunek 16: Pierwsza predykcja wykonana dla fragmentu Rzeszowa, będącego elementem zbioru walidacyjnego.



Rysunek 17: Druga predykcja wykonana dla fragmentu Rzeszowa, będącego elementem zbioru walidacyjnego.



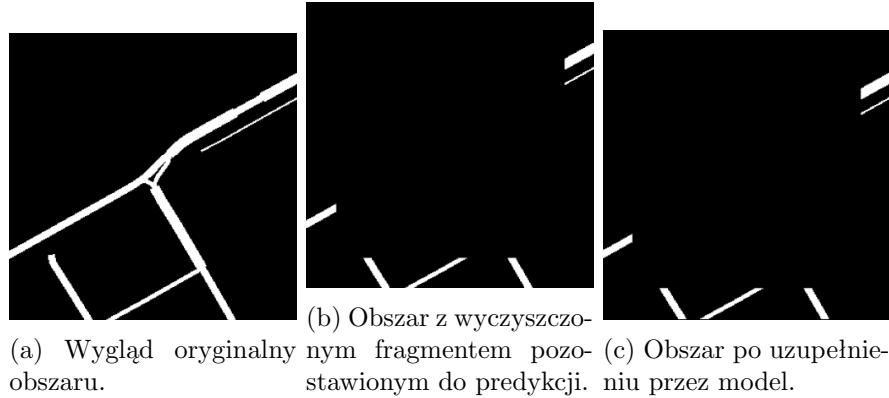
Rysunek 18: Trzecia predykcja wykonana dla fragmentu Rzeszowa, będącego elementem zbioru walidacyjnego.

## 5.2 Parametry wejściowe i wyjściowe

### 5.2.1 Model z pełnymi parametrami

parametr modelu	wartość
architektura sieci neuronowej	<i>spłycony U-Net</i>
wielkość wycinka	256
nadmiar modelu	64
wykorzystywane dane wejściowe	obecność drogi, wysokość n.p.m. status drogi osiedlowej
zwracane dane	obecność drogi, wysokość n.p.m. status drogi osiedlowej
czas szkolenia	184 epoki po 150 kroków
końcowa metryka <code>_dice_coef</code> dla zbioru uczącego	0,0248
końcowa metryka <code>_dice_coef</code> dla zbioru testowego	0,0298
końcowa metryka AUC P-R dla zbioru testowego	0,0184
końcowa metryka Precision dla zbioru testowego	0
końcowa metryka Recall dla zbioru testowego	0

W przeciwieństwie do prezentowanych wcześniej modeli ten nie był szkolony w wykorzystaniem early stopping. Zostało wykonane ponad 300 epok, po czym ręcznie wybrano tę z najlepszą metryką `_dice_coef`. Jak widać jednak po pozostałych metrykach, model ten w ogóle się nie uczył. Wyniki predykcji potwierdzają tę tezę. Po kilku eksperymentach tego typu na różnych wielkościach modeli zarzucono więc ten sposób uczenia na rzecz wejścia zawierającego jedynie *obecność drogi*.



Rysunek 19: Pierwsza predykcja wykonana dla fragmentu Bydgoszczy, będącej elementem zbioru uczącego.



Rysunek 20: Wejścia do i wyjścia z modelu związane z predykcjami poszczególnych wycinków w toku predykcji na fragmencie Bydgoszczy zobrazowanej na ilustracji 9. Warto przypomnieć, że w *modelu przycinającym* wejścia nie są rozłączne, ale wyjścia już co do zasady tak.

### 5.3 Achitektury

Podjęto również próbę wyszkolenia modelu o pełnej *architekturze U-Net*, jednak trenowane modele istotnie lepszych wyników. Modele zostały przedstawione w tabeli 2.

Tabela 2: Parametry sprawdzanych modeli o architekturze U-Net.

wielkość wycinka	nadmiar	czas szkolenia	metryka _dice_coef
256	64	48 epok po 150 kroków	0,513
256	128	62 epoki po 200 kroków	0,575

## 5.4 Inne eksperymenty

Oprócz powyższych, przeprowadzono także inne eksperymenty z nieco zmodyfikowanymi parametrami - takimi jak rozmiar wycinka i wymiar jego brzegu. Wszystkie z nich sprowadzają się jednak do tego samego wniosku - model przedłuża drogi od brzegu ramki w kierunku jej środka, zostawiając jednak zwykle puste fragmenty środka. Warto jednakże wspomnieć o próbie dodania dodatkowego wyjścia w testowanym modelu - wyjście „distance”, któremu nadano osobne funkcje straty, aby karać model za brak ciągłości dróg. Próby nie przyniosły widocznych pozytywnych zmian w predykcjach.

# 6 Wnioski

## 6.1 Reprezentacja problemu

Opisana w raporcie reprezentacja problemu z góry skazuje zbiór danych uczących na niezbilansowanie, co wymusza zastosowanie specjalnych funkcji straty oraz utrudnia uczenie. W zastosowanym podejściu problemem jest już samo uzyskanie ciągłości dróg na wyjściu, co sugeruje, że dobrym pomysłem może być reprezentacja, która gwarantuje tę własność. Czymś takim mogłaby być reprezentacja grafowa (być może uproszczona reprezentacja *OSMnx*). Jednak taka zmiana podejścia wymagałaby stworzenia zupełnie innego modelu.

## 6.2 Przekazywanie kontekstu pomiędzy predykcjami poszczególnych wycinków

Kolejność dokonywania predykcji *wycinków* jest prawdopodobnie błędna. Zamiast realizować to wierszami należało by raczej stosować przebieg, który by propagował informacje z brzegów otrzymanego fragmentu do jego centrum. Tę własność miałby np. zaprezentowany na rysunku 21 przebieg spiralny.

Innym zagadnieniem jest sposób realizacji tego zadania. Obecnie kontekst jest przekazywany w ramach generowanego zbioru uczącego, natomiast być może lepszym pomysłem byłoby wykorzystanie do tego celu sieci rekurencyjnej. Dzięki temu mogłoby dochodzić do propagacji gradientów związanych z kontekstem, co być może poprawiłoby efekty uczenia. Byłoby to jednak związane ze znacznym wzrostem wymagań obliczeniowych i pamięciowym uczenia modelu.

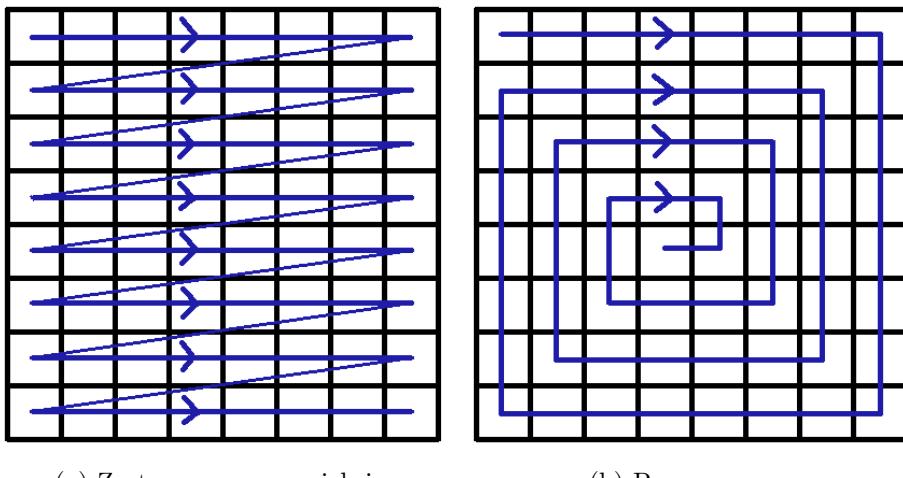
Jest również możliwe, że błędem było prowadzenie uczenia w oparciu o pojedyncze wycinki, co odbierało modelowi możliwość uczenia się odporności na własny szum. Być może już na etapie uczenia należało modelowi przekazywać jego własny wygenerowany kontekst lub Najpierw uczyć jedyne na kontekście rzeczywistym by potem kontynuować uczenie z kontekstem własnym.

### 6.3 Zbiór danych

Bardzo możliwe, że wykorzystany zbiór danych był zbyt mały. Nie mniej jednak zajmował ponad 30 GiB miejsca na dysku, zatem zwiększenie go o rząd wielkości sprawiałoby poważne problemy w związku z tym, że uczenie było prowadzone na zwykłych laptopach.

Innym aspektem jest równoważenie zbioru pod kątem przypadków w rodzaju tego zaprezentowanego na rysunku 17 — czyli ślepych ulic. Należało się zastanowić, jak wiele takich przypadków znajduje się w zbiorze uczącym i ewentualnie tę wartość modyfikować. Jednak w przypadku opisywanego projektu w procesie uczenia współrzędne wycinka były zupełnie losowe, co pozwalała sądzić, że prawdopodobieństwo „przerwania” drogi *wycinkiem* było znacznie większe niż trafienie na koniec ślepej ulicy. Natomiast bardziej prawdopodobne mogły być przypadki „trafienia” w brzeg ulicy bez jej przecięcia (jak lewa jezdnia na rysunku 18). Być może tego typu przypadki „sklaniają” model do nadużywania zakończeń dróg jako ślepa ulica. Pewnym pomysłem jest też dodanie parametru, który decydował by o tym, jak bardzo model ma być skłanny do ściągania, a jak — do zakończania dróg jako ślepe ulice.

Warto też zauważyć, że właściwie nie przeprowadzono klasycznego uczenia smonadzorowanego, w którym to do danych stopniowo dodaje się co raz więcej czumu. Zamiast tego usuwano zupełnie dane na wejściu do modelu podczas uczenia, co być może zaburzało ten proces, gdyż odebrało to możliwość stopniowego zwiększania poziomu trudności zadania.



(a) Zastosowany w projekcie.

(b) Proponowany.

Rysunek 21: Kierunki predykcji poszczególnych wycinków.

## 7 Podział pracy

- Konrad Ćwięka — przygotowanie zbioru danych (parametry *obecność drogi i status drogi osiedlowej*), aplikacja demonstracyjna, eksperymenty z architekturą U-Net.
- Szymon Kowalski — *część górną*, algorytm rekonstrukcji grafu z segmentów, analiza parametrów fizycznych wygenerowanych dróg,
- Michał Król — przygotowanie zbioru danych (parametr *wysokość nad poziomem morza*), wykrywanie nachyleń dróg na *siatce punktów*
- Grzegorz Lenarski — prowadzenie projektu, *część górną*, implementacja *segmentacji* zbioru danych, koncepcja *modelu przycinającego*, eksperymenty z różnymi rozmiarami *wycinków* modelu i różnymi parametrami wejściowymi.
- Jakub Łabuz — implementacja uczenia, strojenie modelu, eksperymenty z różnymi parametrami wejściowymi modelu.