Uniwersytet Zielonogórski
Instytut Sterowania i Systemów Informatycznych

Rozpoznawanie Obrazów

Semestr II, Informatyka

mgr inż. Marcin Skobel

2023

Laboratorium nr 9: Segmentacja obrazu - splotowe sieci neuronowe

I. Zagadnienia teoretyczne

Wstęp

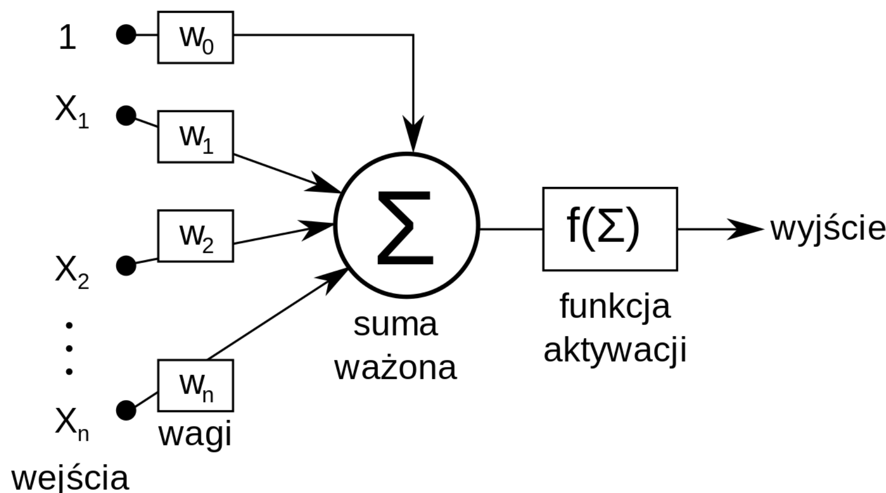
Segmentacja obrazu współcześnie najczęściej jest wykonywana przy zastosowaniu splotowych sieci neuronowych. Zanim jednak dojdziemy do praktycznych zastosowań warto by było przypomnieć sobie podstawowe zagadnienia związane ze sztucznymi sieciami neuronowymi. Ostatecznie do budowy i zrozumienia działania splotowych sieci neuronowych należy znać pojęcia takie jak: architektura sieci neuronowej, uczenie sieci oraz funkcja aktywacji i jej rodzaje. Niestety ramy czasowe laboratorium nie pozwalają na większe rozwinięcie rozległego tematu zastosowań sztucznych sieci neuronowych w segmentacji obrazów, dlatego skupimy się na najważniejszych aspektach bieżącego tematu.

Pojęcia podstawowe

Neuron

Sieć neuronowa jak sama nazwa wskazuje jest zbudowana z podstawowych jednostek zwanych neuronami. Często w różnych opracowaniach sugeruje się, że sztuczny neuron jest odpowiednikiem neuronu biologicznego. Prawda jest jednak taka, że neuron sztuczny jest bardzo luźno inspirowany neuronem biologicznym i tak naprawdę jest jedynie matematyczną ramą służącą do przetwarzania danych.

Na rysunku 1 przedstawiono podstawowy schemat budowy sztucznego neuronu. W prezentowanym przykładnie neuron posiada wiele wejść i jedno wyjście. Oprócz wejść i wyjąć występują również wagi, które podlegają uczeniu na podstawie sygnałów wejściowych w celu wyznaczenia liniowej granicy decyzyjnej. Zatem pojedynczy neuron jest w stanie poradzić sobie z problemem separowalnym liniowo. Reguła uczenia się perceptronu stwierdza, że algorytm automatycznie



Rysunek 1: Przykładowy schemat sztucznego neuronu, źródło: wikipedia.org

nauczy się optymalnych współczynników wagi. Cechy wejściowe są następnie mnożone przez te wagi, aby określić, czy neuron jest aktywowany, czy też nie. Perceptron odbiera wiele sygnałów wejściowych, a jeśli suma sygnałów wejściowych przekracza pewien próg, wtedy albo wysyła sygnał, albo nie zwraca sygnału wyjściowego. Na podstawie różnych danych wejściowych określić można wartość aktywacji neuronu na podstawie wzoru:

$$a(x) = \sum_{i=1}^{m} x_i w_i \tag{1}$$

gdzie x_i jest wartością wejścia neuronu, natomiast w_i oznacza wartość połączenia pomiędzy neuronem i a wyjściem. Neuron może być jednocześnie podłączony do dodatkowego wejścia w postaci biasu, który otrzymuje stałą wartość na przykład równą jeden co z kolei przekłąda sie na ostateczny wzór:

$$a(x) = \sum_{i=1}^{m} x_i w_i + w_0 \tag{2}$$

Sieć głęboka

Przedstawiony powyżej schemat najprostszej sieci (Preceptronu) wskazuje, że sieć ta składa się jedynie z pojedynczego neuronu pozwalającego na rozwiązanie problemu liniowo separowalnego. Rozwinięciem tej koncepcji jest zbudowanie sieci złożonej z kilku neuronów. Sieć taka posiada znacznie większe zdolności uczące i może być stosowane do bardziej skomplikowanych problemów. Najprostszym przykładem sieci złożonej może być sieć złożona kilku równoległych neuronów do których dołączone są wszystkie wejścia. Dalsze rozwinięcie polega na dodawaniu kolejnych warstw neuronów przy czym warstwy nie mające bezpośredniego kontaktu z wejściami oraz wyjściami nazywa się warstwami ukrytymi, taka sieć z kolei nazywana jest siecią głęboką, a trening tej sieci uczeniem głębokim (deep learning).

Funkcja aktywacji

Funkcja aktywacji jest to rodzaj funkcji, której zadaniem jest obliczenie wartości wyjścia neuronów sieci neuronowej. Najprostszy wynik funkcji aktywacji można porównać do przełącznika

”włącz”/”wyłącz” jednak rzeczywiste problemy jakie jesteśmy w stanie rozwiązać przy użyciu sztucznych sieci neuronowych czasem wymagają użycia bardziej skomplikowanych a nawet nieliniowych funkcji aktywacji. Możemy zatem zauważyć, że funkcja aktywacji może przyjmować różne formy, przykładowo:
- funkcja progowa (zakres wartości {0, 1})

$$fx = \begin{cases} 0 & \text{dla} \quad x < 0 \\ 1 & \text{dla} \quad x \geqslant 0 \end{cases} \tag{3}$$

- funkcja sigmoidalna/logistyczna (zakres wartości (0, 1))

$$fx = \frac{1}{1 + e^{-x}} \tag{4}$$

- funkcja hiperboliczna (zakres wartości (-1, 1))

$$fx = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{5}$$

- funkcja ReLU (zakres wartości (0, ∞))

$$fx = \begin{cases} 0 & \text{dla} \quad x \leqslant 0 \\ x & \text{dla} \quad x > 0 \end{cases} \tag{6}$$

- funkcja softmax (zakres wartości (0, 1))

$$fx = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} \tag{7}$$

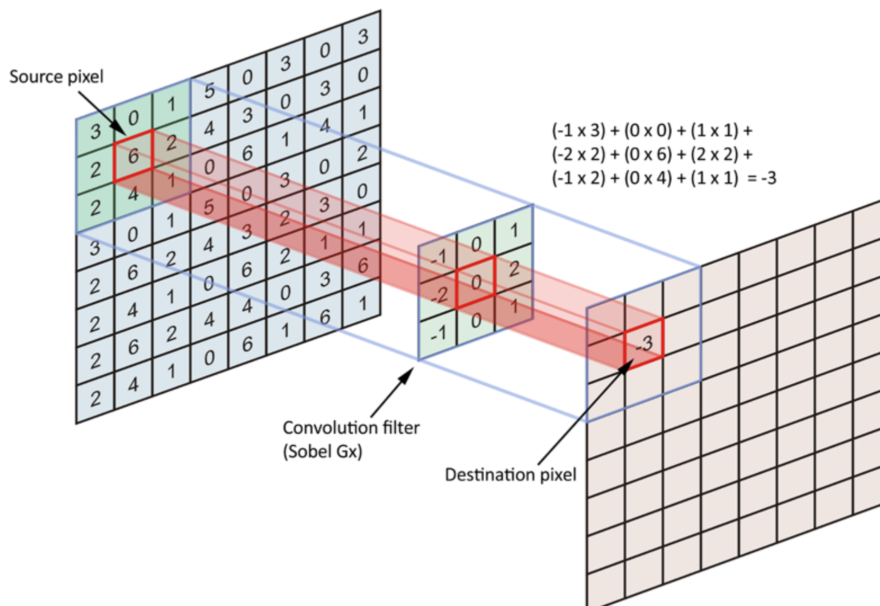
Lista ta nie wyczerpuje wszystkich przykładów funkcji aktywacji, jedynie wymienia najważniejsze z nich.

Warstwy konwolucyjne (splotowe)

Sztuczne splotowe sieci neuronowe (CNN) to przykład sieci głębokiej zorganizowanej w sposób hierarchiczny. Kolejne warstwy dokonują przetwarzania danych z niższego poziomu. Dane uzyskane na niższym poziomie przekazywane są do kolejnej warstwy, dzięki temu sieć stopniowo uzyskuje bardziej złożone informacje. Sieci CNN znajdują przede wszystkim zastosowanie w semantycznej segmentacji obrazów oraz ich klasyfikacji. Ponadto sieci CNN używane są w zagadnieniu transfer learningu, czyli problemie badawczym polegającym na zastosowaniu sieci wyspecjalizowanych w rozwiązywaniu jednego zagadnienia do rozwiązywanie innego problemu.

Podstawową strukturą służącą do uczenia sieci splotowej jest filtr który przyjmuje formę macierzy kwadratowej z różnymi wartościami. Filtr stanowi w sieci splotowej odpowiednik zbioru wag aktywacji neuronu. Innymi słowy to właśnie filtry oraz ich modyfikacja stanowią źródło treningu splotowej sieci neuronowej. Przetwarzanie obrazu przy użyciu filtra odbywa się za pośrednictwem operacji splotu. Operacja ta była przedstawiana na laboratorium związanym z filtracją obrazów, więc warto teraz sobie krótko przypomnieć czym jest splot.

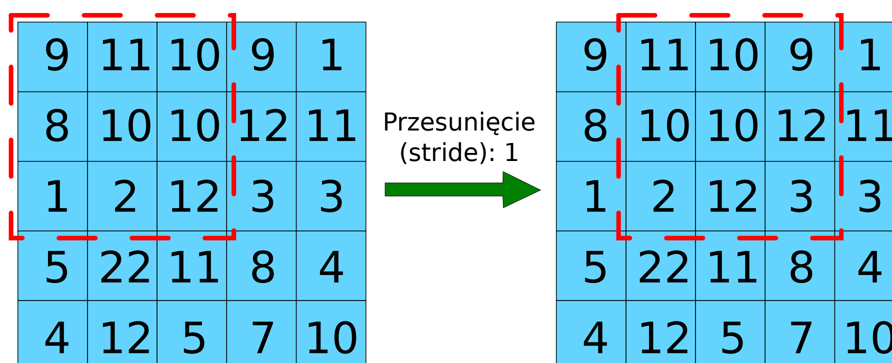
Operacja splotu polega na wykonaniu iloczynu skalarnego filtra oraz fragmentu macierzy o odpowiedniej wielkości. Z reguły przetwarzane są macierze o rozmiarze 3x3. Operacja splotu



Rysunek 2: Operacja splotu

została zilustrowana na rysunku 2. Wykonanie operacji splotu obrazu z filtrem powoduje powstanie obrazu wynikowego zwanego mapą cech. Charakterystyczną cechą sieci CNN jest fakt, że neurony z pierwszej warstwy nie są połączone z każdym pikselem obrazu wejściowego lecz jedynie ze zbiorem pikseli znajdujących się w polu recepcyjnym neuronu.

Operacja splotu wymaga zdefiniowania dwóch podstawowych rzeczy. Pierwsza z nich to wielkość filtra i jego wartości startowe. Druga natomiast informacja to wielkość przesunięcia filtra przetwarzającego. Przesunięcie możemy zdefiniować przy użyciu rysunku 3. Przesunięcie

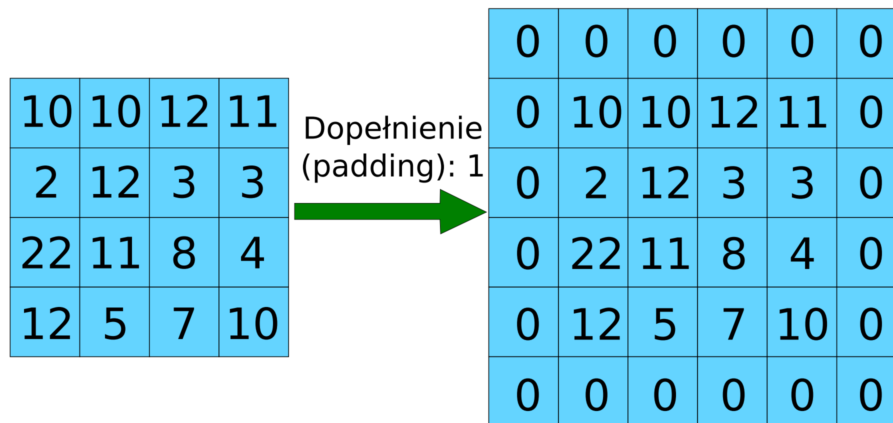


Rysunek 3: Model przesunięcia

o wartości 1 powoduje przesunięcie filtra przetwarzania o jeden piksel, co zostało na rysunku 3 oznaczone kolorem czerwonym.

Operacja splotu powoduje, że obraz ulega redukcji na krawędzi, ponieważ w wyniku operacji splotu np. macierzy 3x3 powstaje tylko jedna wartość zatem 9 pikseli zostaje zredukowane do 1 wartości. Ponadto jeśli przesunięcie wynosiło 1 wówczas w takiej konfiguracji po operacji splotu stracimy po jednym pikselu dookoła obrazu. Aby poradzić sobie z problemem redukcji

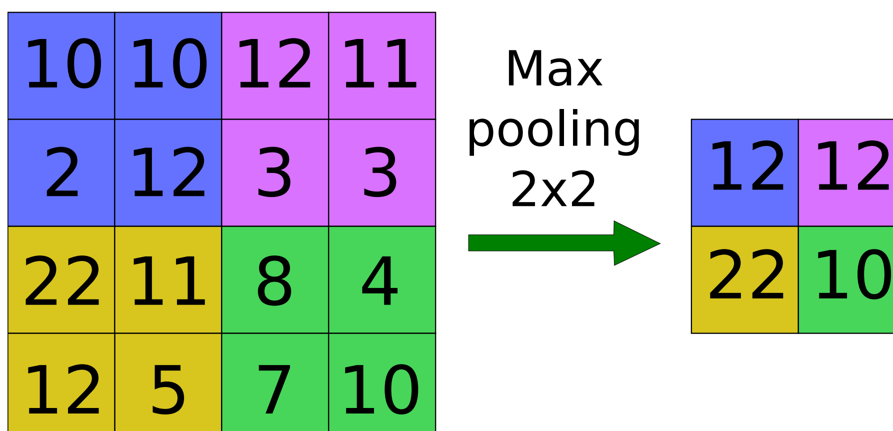
pikseli można zastosować operację dopełnienia. Operacje tą można stosunkowo łatwo zaprezentować graficznie, zatem dopełnienie wykonywane jest podobnie jak na rysunku 4. Najczęściej



Rysunek 4: Dopełnienie

dopełnienie po operacji splotu wykonuje się poprzez dodanie brakujących pikseli o wartości 0.

Bardzo użyteczną własnością wynikającą z użycia warstw splotowych jest redukcja liczby parametrów obrazu wejściowego, a co za tym idzie redukuja złożoności obliczeniowej badanego problemu. Liczba parametrów obrazu wejściowego w odcieniach szarości o rozmiarze 256x256 wynosi dokładnie 256^2 czyli całkiem sporo oraz na tyle dużo żeby skutecznie uniemożliwić kompletne przeanalizowanie całego obrazu piksel po pikselu. Splotowa sieć neuronowa w uproszczeniu analizuje wybrane fragmenty obrazu w celu przeanalizowania jego zawartości. W zależności od potrzeb CNN może być stosowana jako narzędzie do klasyfikacji obrazów lub segmentacji. Oprócz tego że warstwy splotowe powodują redukcję liczby parametrów dodatkowo zazwyczaj stosuje się jeszcze dalszą redukcję w postaci tworzenia warstw pooling. Przykładem takiej warstwy dość powszechnie stosowanym jest MaxPooling. Warstwa ta powstaje w wyniku wyboru maksymalnej wartości z pól na które został podzielony obraz. Przykładowo mogą to być pola o rozmiarze 2x2 czyli łącznie 4 pola spośród, których wybierane jest pole o najwyższej wartości. To z kolei powoduje, że przetwarzany obraz zmniejsza się o połowę (rys. 5). Oczywiście

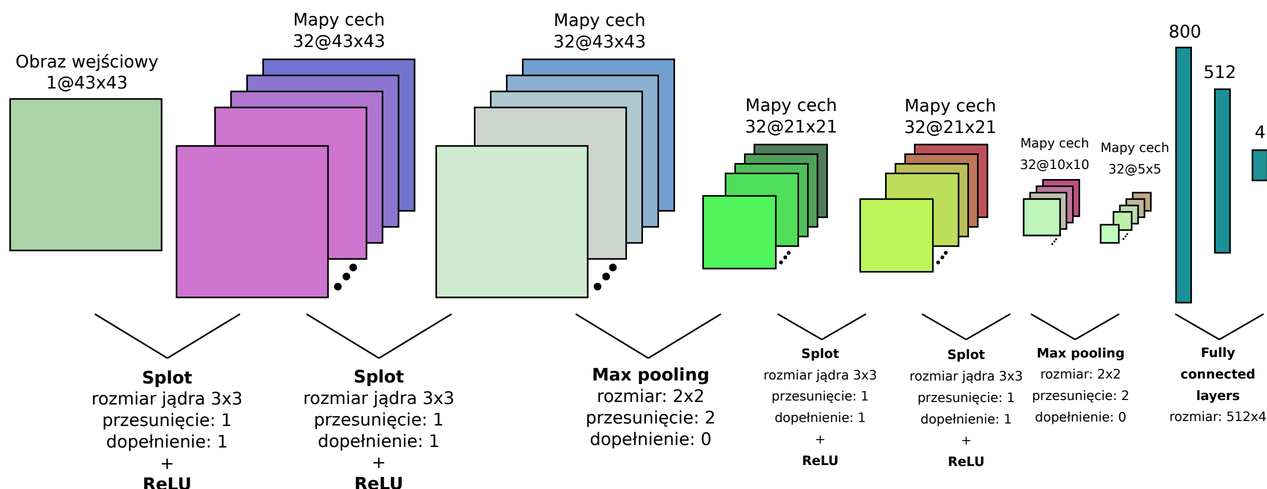


Rysunek 5: Warstwa MaxPooling

występują inne metody poolingu jednak w praktyce najczęściej można się spotkać właśnie z MaxPoolingiem.

Przykładowe splotowe sieci neuronowe

Budowę splotowej sieci neuronowej łatwiej przedstawić na modelu służącym do klasyfikacji obrazów niż do segmentacji, więc rozpoczniemy od prostszego przypadku. Splotowa sieć neuronowa służąca do klasyfikacji może przybrać poniższą formę:

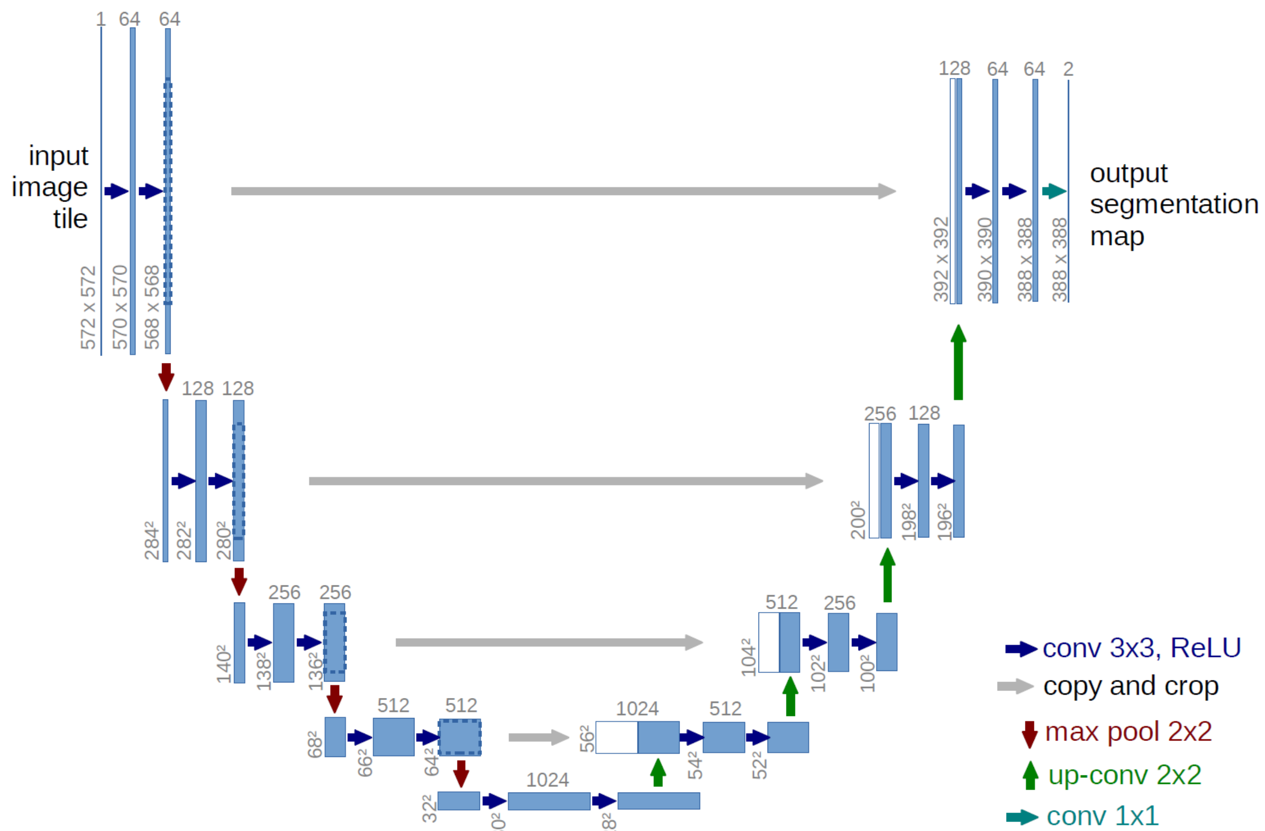


Rysunek 6: Klasyfikacyjna splotowa sieć neuronowa

Zaprezentowana sieć neuronowa rozpoczyna się od obrazu wejściowego o wielkości 43x43 piksele na jednym kanale. W pierwszym kroku następuje przetwarzanie obrazu przy użyciu filtra o rozmiarze 3x3 z przesunięciem o wartości 1 i dopełnieniem zerami o wartości 1. W wyniku splotu powstał tensor map cech o wymiarach 43x43 na 32. Co oznacza, że w pierwszym kroku było 32 filtry w pierwszej warstwie. W drugim kroku wykonywana jest kolejna konwolucja, lecz tym razem wszystkie filtry (32) tworzą tensor o wymiarze 3x3x32 i jednocześnie mogą przetwarzać tensor mapy cech o rozmiarze 43x43x32. W kolejnym kroku występuje warstwa MaxPooling, z rozmiarem pól 2x2 i przesunięceim równym 2 co oznacza, że w wielkość map cech ulegnie redukcji o połowę do rozmiaru 21x21x32. W kolejnych dwóch krokach następują kolejne warstwy splotowe oraz kolejne dwie warstwy MaxPoolingu. Na końcu powstałe 32 mapy cech o rozmiarze 5x5 są łączone do 800 neuronów w pierwszej warstwie fully connected. Następnie pierwsza warstwa fully connected jest połączona jest z 512 neuronami drugiej warstwy fully connected, a ta z kolei jest połączona z 4 neuronami w ostatniej warstwie fully connected. Oznacza to, że na wyjściu sieci mamy 4 klasy obiektów. Przykładowo za pomocą takiej sieci moglibyśmy sklasyfikować na przykład gatunki 4 zwierząt na obrazach.

Przyjrzyjmy się teraz drugiemu przykładowi, tym razem splotowej sieci neuronowej przystosowanej do segmentacji obrazów cyfrowych. Klasycznym już przykładem jest przygotowane w 2015 roku przez zespół w składzie: Olaf Ronneberger, Philipp Fischer, Thomas Brox, sieć o nazwie U-Net. Sieć ta została zaprojektowana do śledzenia obiektów na obrazach cyfrowych jednak wkrótce okazało się, że jest też doskonałym narzędziem do segmentacji obrazów cyfro-

wych. W oryginalnej formie sieć ta prezentuje się następująco:



Rysunek 7: Sieć splotowa U-Net (Ronneberger, Fisher, Brox, 2015)

Jak widać konstrukcja sieci jest już znacznie skomplikowana i oprócz znanych warstw MaxPoolingu i Splotowej pojawia się warstwa up-conv. Wynika to z faktu, że ta splotowa sieć nie kończy swojego działania na klasyfikacji poszczególnych obrazów lecz przywraca obraz do mapy po segmentacji która zawiera mapy obiektów po segmentacji. Widać też, że w pierwotnej wersji poszukiwano dwóch warstw czyli tła oraz obiektu/obiektów i taka jest też ostateczna mapa cech. Ponadto w pierwotnej wersji U-Net mapa wyjściowa (388x388) jest wyraźnie mniejsza od obrazu wejściowego (572x572). Niestety w przypadku wielu praktycznych zastosowań takie ograniczenie pola widzenia sieci nie jest dopuszczalne. Od 2015 roku powstało jednak sporo modyfikacji sieci U-Net dzięki którym możliwe jest odtworzenie obrazu po segmentacji do pierwotnych wymiarów. Jedną z takich modyfikacji omawialiśmy na zajęciach projektowych a dziś przyjrzymy się jej ponownie ale tym razem na solidnej podstawie teoretycznej.

II. Przykład praktyczny

Przykład praktyczny składa się z obrazów zawierających sylwetki zwierząt domowych i zadaniem sieci jest poprawna segmentacja maski sylwetki zwierzęcia, krawędzi tej maski oraz tła. Czyli mają powstać 3 odrębne klasy obiektów na pojedynczym obrazie. Zadanie zaczynamy od zainstalowania przykładu:

```
!pip install git+https://github.com/tensorflow/examples.git
!pip install -U tfds-nightly
```

W kolejnym kroku importujemy bibliotekę to sztucznych sieci neuronowych czyli tensorflow od Google:

```
import tensorflow as tf
```

Następnie importujemy wszystkie niezbędne do obliczeń biblioteki oraz przykłady. Biblioteka pix2pix to zbiór przykładów, tensorflow_datasets to zestaw zbiorów danych:

```
from tensorflow_examples.models.pix2pix import pix2pix

import tensorflow_datasets as tfds

from IPython.display import clear_output
import matplotlib.pyplot as plt
```

Zestaw danych który zostanie użyty w eksperymencie oczywiście jest już dostępny w bibliotece tensorflow_datasets:

```
dataset, info = tfds.load('oxford_iiit_pet:3.*.*', with_info=True)
```

W kolejnym kroku zdefiniujemy trzy bardzo ważne funkcje. Pierwsza służy do normalizacji obrazu, która pozwala na uzyskanie wartości pikseli w przedziale od 0 do 1. Następnie utworzymy dwie funkcje służące do budowania modelu treningowego i testowego:

```
def normalize(input_image, input_mask):
  input_image = tf.cast(input_image, tf.float32) / 255.0
  input_mask -= 1
  return input_image, input_mask

@tf.function
def load_image_train(datapoint):
  input_image = tf.image.resize(datapoint['image'], (128, 128))
  input_mask = tf.image.resize(datapoint['segmentation_mask'], (128, 128))

  if tf.random.uniform(()) > 0.5:
    input_image = tf.image.flip_left_right(input_image)
    input_mask = tf.image.flip_left_right(input_mask)

  input_image, input_mask = normalize(input_image, input_mask)

  return input_image, input_mask

def load_image_test(datapoint):
  input_image = tf.image.resize(datapoint['image'], (128, 128))
  input_mask = tf.image.resize(datapoint['segmentation_mask'], (128, 128))

  input_image, input_mask = normalize(input_image, input_mask)

  return input_image, input_mask
```

Po utworzeniu zbiorów danych przechodzimy do ustawienia podstawowych parametrów sieci. Wśród nich warto zwrócić uwagę na wielkość BATCH_SIZE, która oznacza wielkość pojedynczej liczby próbek na wejściu do sieci. Jeśli podamy zbyt wysoką liczbę wówczas colab może nie podołać zadaniu, więc jeśli mamy duże obrazki np. 512x512 wówczas dobrze jest odpowiednio zmniejszyć BATCH_SIZE:

```
TRAIN_LENGTH = info.splits['train'].num_examples
BATCH_SIZE = 64
BUFFER_SIZE = 1000
STEPS_PER_EPOCH = TRAIN_LENGTH // BATCH_SIZE

train = dataset['train'].map(load_image_train, num_parallel_calls=tf.data.experimental.AUTO
test = dataset['test'].map(load_image_test)

train_dataset = train.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE).repeat()
train_dataset = train_dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
test_dataset = test.batch(BATCH_SIZE)
```

W kolejnym kroku definiujemy funkcję służącą do wyświetlania podglądu obrazka testowego w celu określenia jak dobrze model zaczyna działać w trakcie treningu:

```
def display(display_list):
  plt.figure(figsize=(15, 15))

  title = ['Input Image', 'True Mask', 'Predicted Mask']

  for i in range(len(display_list)):
    plt.subplot(1, len(display_list), i+1)
    plt.title(title[i])
    plt.imshow(tf.keras.preprocessing.image.array_to_img(display_list[i]))
    plt.axis('off')
  plt.show()

for image, mask in train.take(1):
  sample_image, sample_mask = image, mask
display([sample_image, sample_mask])
```

Do eksperymentu zostanie użyta zmodyfikowana sieć typu U-Net. Sieć U-Net składa się z enkodera (downsampler) oraz dekodera (upsampler). W celu redukcji liczby parametrów jako enkoder do uczenia wykorzystany zostanie wstępnie wytrenowany model o nazwie MobileNetV2 natomiast dekoder jest zaimplementowany jako unsample w tutorialu pod nazwą pix2pix. Wyjściowe klasy obiektów są trzy i wynika to z faktu, że proces segmentacji w przypadku sieci neuronowej będzie to klasyfikacja pojedynczego piksela do jednej z 3 badanych klas:

```
OUTPUT_CHANNELS = 3
```

Jak wspomniano, koder będzie wstępnie wyszkolonym modelem MobileNetV2, który jest przygotowany i gotowy do użycia w aplikacjach tf.keras.applications. Koder składa się z określonych

wyjść z warstw pośrednich w modelu. Należy pamiętać, że koder nie zostanie wytrenowany podczas procesu uczenia:

```
base_model = tf.keras.applications.MobileNetV2(input_shape=[128, 128, 3], include_top=Fals

# Use the activations of these layers
layer_names = [
    'block_1_expand_relu',   # 64x64
    'block_3_expand_relu',   # 32x32
    'block_6_expand_relu',   # 16x16
    'block_13_expand_relu',  # 8x8
    'block_16_project',      # 4x4
]
layers = [base_model.get_layer(name).output for name in layer_names]

# Create the feature extraction model
down_stack = tf.keras.Model(inputs=base_model.input, outputs=layers)

down_stack.trainable = False
```

Dekoder / upsampler to po prostu seria bloków upsamplingu zaimplementowanych w przykładach TensorFlow:

```
up_stack = [
    pix2pix.upsample(512, 3),  # 4x4 -> 8x8
    pix2pix.upsample(256, 3),  # 8x8 -> 16x16
    pix2pix.upsample(128, 3),  # 16x16 -> 32x32
    pix2pix.upsample(64, 3),   # 32x32 -> 64x64
]

def unet_model(output_channels):
  inputs = tf.keras.layers.Input(shape=[128, 128, 3])
  x = inputs

  # Downsampling through the model
  skips = down_stack(x)
  x = skips[-1]
  skips = reversed(skips[:-1])

  # Upsampling and establishing the skip connections
  for up, skip in zip(up_stack, skips):
    x = up(x)
    concat = tf.keras.layers.Concatenate()
    x = concat([x, skip])

  # This is the last layer of the model
  last = tf.keras.layers.Conv2DTranspose(
      output_channels, 3, strides=2,
      padding='same')  #64x64 -> 128x128
```

```
  x = last(x)

  return tf.keras.Model(inputs=inputs, outputs=x)
```

Teraz pozostaje tylko skompilować i wytrenować model. Wykorzystywana tutaj strata to *losses.SparseCategoricalCrossentropy(from_logits=True)*. Powodem używania tej funkcji utraty jest to, że sieć próbuje przypisać każdemu pikselowi etykietę, podobnie jak w przypadku przewidywania wieloklasowego. W prawdziwej masce segmentacji każdy piksel ma wartość 0,1,2. Sieć tutaj wysyła trzy kanały. Zasadniczo każdy kanał próbuje nauczyć się przewidywać klasę i *losses.SparseCategoricalCrossentropy(from_logits=True)* jest zalecaną stratą dla takiego scenariusza. Korzystając z wyjścia sieci, etykieta przypisana do piksela to kanał o najwyższej wartości. To właśnie robi funkcja create_mask:

```
model = unet_model(OUTPUT_CHANNELS)
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Można teraz spojrzeć na architekturę utworzonego modelu:

```
tf.keras.utils.plot_model(model, show_shapes=True)
```

Dodatkowo można też wypróbować model przed treningiem aby sprawdzić jakiej predykcji dokonał:

```
def create_mask(pred_mask):
  pred_mask = tf.argmax(pred_mask, axis=-1)
  pred_mask = pred_mask[..., tf.newaxis]
  return pred_mask[0]

def show_predictions(dataset=None, num=1):
  if dataset:
    for image, mask in dataset.take(num):
      pred_mask = model.predict(image)
      display([image[0], mask[0], create_mask(pred_mask)])
  else:
    display([sample_image, sample_mask,
             create_mask(model.predict(sample_image[tf.newaxis, ...]))])

show_predictions()
```

W kolejnym kroku możemy dokonać weryfikacji czy model ulega poprawie a także rozpocząć proces uczenia sieci. Należy obserwować przebieg zmian wartości skuteczności segmentacji na danych treningowych oraz walidacyjnych lub wartość błędu. Jeśli skuteczność na danych treningowych wzrasta natomiast na danych walidacyjnych się pogarsza oznacza to, że model uległ przetrenowaniu i naszym zadaniem będzie znalezienie takiej epoki w której wartość błędu jest najniższa dla danych treningowych i walidacyjnych:

```
class DisplayCallback(tf.keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs=None):
    clear_output(wait=True)
    show_predictions()
    print ('\nSample Prediction after epoch {}\n'.format(epoch+1))

EPOCHS = 20
VAL_SUBSPLITS = 5
VALIDATION_STEPS = info.splits['test'].num_examples//BATCH_SIZE//VAL_SUBSPLITS

model_history = model.fit(train_dataset, epochs=EPOCHS,
                          steps_per_epoch=STEPS_PER_EPOCH,
                          validation_steps=VALIDATION_STEPS,
                          validation_data=test_dataset,
                          callbacks=[DisplayCallback()])

loss = model_history.history['loss']
val_loss = model_history.history['val_loss']

epochs = range(EPOCHS)

plt.figure()
plt.plot(epochs, loss, 'r', label='Training loss')
plt.plot(epochs, val_loss, 'bo', label='Validation loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss Value')
plt.ylim([0, 1])
plt.legend()
plt.show()
```

Zróbmy kilka prognoz. W celu zaoszczędzenia czasu, liczba epok była niewielka, ale można ustawić ją na wyższą, aby uzyskać dokładniejsze wyniki (możliwe jest też spowodowanie przeuczenia sieci):

```
show_predictions(test_dataset, 3)
```

III. Uwagi

Plik z przykładami można pobrać ze strony: `https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/images/segmentation.ipynb#scrollTo=DeFwFDN6EVoI`

IV. Lista zadań

1. Wykonaj przykład z zadania i ustal czy sieć nie uległa przetrenowaniu, jeśli tak się stało ustal numer epoki dla której dane walidacyjne miały najlepszy rezultat w przeciwnym razie zwiększ liczbę epok obliczeniowych.