KLASYFIKACJA OBRAZÓW HISTOPATOLOGICZNYCH TKANEK PŁUCNYCH: AUTOMATYCZNE ROZPOZNAWANIE TYPÓW NOWOTWORÓW ZA POMOCĄ KONWOLUCYJNYCH SIECI NEURONOWYCH (CNN)

SPRAWOZDANIE DO PROJEKTU Z PRZEDMIOTU SIECI NEURONOWE

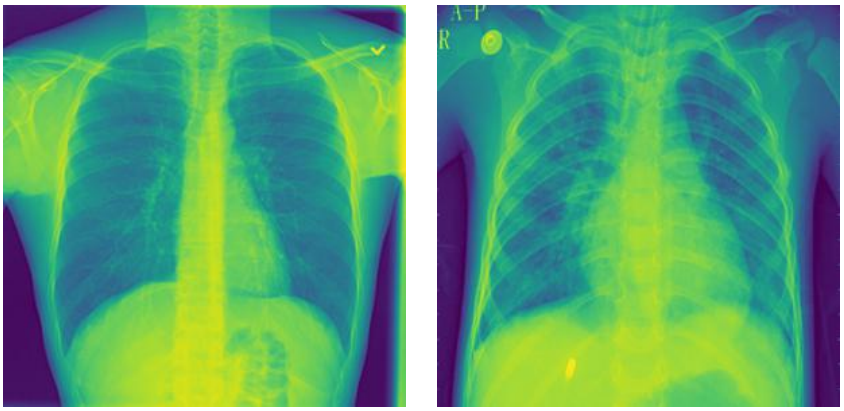
KLAUDIA STĘPIEŃ (S230062)

Spis treści

1

WPROWADZENIE

Klasyfikacja obrazów jest kluczowym zadaniem w medycynie, służącym do wykrywania różnych patologii, takich jak nowotwory, zakażenia, choroby przewlekłe oraz urazy w ciele.



(a) Rentgen osoby zdrowej (b) Rentgen osoby chorej

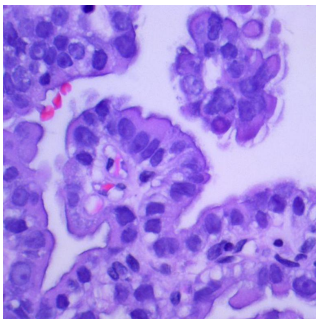
Najpopularniejszymi typami obrazów w medycynie są:

- Obrazy radiologiczne: rentgen, tomografia, rezonans magnetyczny
- Obrazy ultrasonograficzne (USG)
- Obrazy histopatologiczne, czyli obrazy mikroskopowe pobranych fragmentów tkanek, które są barwione specjalnymi odczynnikami, aby ułatwić rozpoznanie zmian patologicznych.

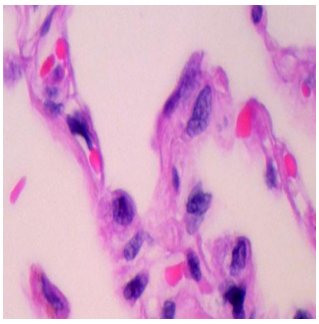
W ramach projektu skupię się na obrazach histopatologicznych dwóch rodzajów nowotworów płuc oraz łagodnej tkanki płuc (niegroźne zmiany, ale warto sprawdzić):

- gruczolakorak płuc
- rak płaskonabłonkowy płuc

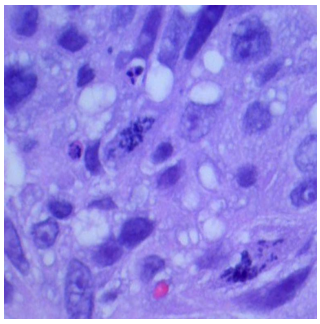
Poniżej znajdują się przykładowe zdjęcia histopatologiczne dla każdego rodzaju nowotworu oraz tkanki łagodnej.



(a) Gruczolakorak płuc



(b) Łagodna tkanka płuc



(c) Rak płaskonabłonkowy płuc

Sztuczne sieci neuronowe to algorytmy inspirowane budową i funkcjonowaniem ludzkiego mózgu. W ostatnich latach znalazły one szerokie zastosowanie w medycynie, gdzie rewolucjonizują sposób, w jaki diagnozujemy i leczymy choroby. Sieci neuronowe są trenowane na ogromnych zbiorach danych medycznych, takich jak obrazy radiologiczne, wyniki badań laboratoryjnych czy dane genetyczne. Dzięki temu uczą się rozpoznawać wzorce i zależności, które mogą być niezauważalne dla ludzkiego oka. Dzięki nim diagnostyka może stać się bardziej precyzyjna, szybsza i bardziej dostępna.
W tym momencie jednymi z najbardziej rozwijanych sieci neuronowych są konwolucyjne sieci neuronowe. Sieci te sprawdzają się bardzo dobrze m.in. do rozpoznawania obrazów oraz detekcji obiektów na nich.
Celem projektu jest zaprojektowanie konwolucyjnej sieci neuronowej za pomocą PyTorch w celu odpowiedniego sklasyfikowania obrazów histopatologicznych tkanek płucnych.

2

OPIS DANYCH

Dane, których użyłam do projektu pochodzą ze strony Kaggle i dotyczą zdjęć histopatologicznych tkanek płucnych z podziałem na 3 rodzaje:

- Tkanka łagodna płuc,
- Gruczolakorak płuc,
- Rak płaskonabłonkowy płuc.

Dane są od razu podzielone na 3 zbiory:

- dane treningowe,
- dane walidacyjne,
- dane testowe.

Na potrzeby projektu użyłam danych treningowych oraz testowych. Oba zbiory w sumie liczą 13507 obrazów. Zbiór danych treningowych to 12007 obrazów podzielonych na 3 klasy (około 4000 obrazów na klasę). Zbiór danych testowych zawiera 1500 obrazów podzielonych na 3 klasy (po 500 obrazów na każdą klasę).

Ze względu na tak dużą liczbę obrazów na potrzeby projektu (szybkość uczenia sieci) postanowiłam za pomocą biblioteki os wyodrębnić losowo co 3 plik z każdego zbioru. Za każdym uruchomieniem programu wszystkie pliki jpg są usuwane z katalogów, a następnie wybierane w sposób losowy i kopiowane z katalogów pierwotnych.

Poniżej znajdują się 3 funkcje, które odpowiadają za wybieranie danych i usuwanie ich.

```
def copy_random_data_from_one_folder(source_path, dest_path):
    all_files = os.listdir(source_path)
    random.shuffle(all_files)
    for i, file in enumerate(all_files):
      if i % 3 == 0:
        source_file = os.path.join(source_path, file)
        destination_file = os.path.join(dest_path, file)
        shutil.copy2(source_file, destination_file)

def copy_random_data_from_all_folders():
```

```
    copy_random_data_from_one_folder('pliki\Testing\
        Lung_adenocarcinoma','pliki_new\Testing\
        Lung_adenocarcinoma')
    copy_random_data_from_one_folder('pliki\Testing\
        Lung_benign_tissue','pliki_new\Testing\Lung_benign_tissue'
        )
    copy_random_data_from_one_folder('pliki\Testing\Lung_squamous
        cell_carcinoma','pliki_new\Testing\Lung_squamous
        cell_carcinoma')
    copy_random_data_from_one_folder('pliki\Training\
        Lung_adenocarcinoma','pliki_new\Training\
        Lung_adenocarcinoma')
    copy_random_data_from_one_folder('pliki\Training\
        Lung_benign_tissue','pliki_new\Training\Lung_benign_tissue
        ')
    copy_random_data_from_one_folder('pliki\Training\Lung_squamous
         cell_carcinoma','pliki_new\Training\Lung_squamous
        cell_carcinoma')

def delete_all_test_jpg_files(path):
    for root, dirs, files in os.walk(path):
        for file in files:
            if file.endswith(".jpg") or file.endswith(".jpeg"):
                os.remove(os.path.join(root, file))
```

Obrazy, które zostały wybrane są poddane prostemu przekształceniu w celu przygotowania danych wejściowych dla sieci neuronowej. Zarówno dane treningowe, jak i testowe są przekształcone w ten sam sposób. Główne przekształcenia opierają się na:

- normalizacji - standaryzacja danych wejściowych (pikseli) dla każdego kanału RGB do przedziału [-1,1]
- ujednolicenia rozmiarów obrazów do rozmianu 32x32 (w pikselach)
- zamiany obrazów na tensory pytorch

Poniżej znajduje się kod, który definiuje przekształcenia obrazów zarówno dla zbioru treningowego (transform_train), jak i dla zbioru testowego (transform_test).

```
transform_train = transforms.Compose([
    transforms.Resize((32,32)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

transform_test = transforms.Compose([
    transforms.Resize((32,32)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

Dane treningowe i testowe są wczytywane z folderów i przekształcane poprzez ImageFile. Następnie dla danych testowych i treningowych są tworzone iteratory za pomocą DataLoader. DataLoader służy do podzielenia danych na tzw. batche, czyli podzbiory danych wejściowych, które są przetwarzane przez model podczas jednej epoki w procesie uczenia. W moim projekcie, ze względu na ilość obrazów poddanych analizie, rozmiar takiego batcha to 25.

Poniżej kod wczytujący dane do programu.

```
train_data = ImageFolder(root = "pliki_new/Training", transform =
     transform_train)
test_data = ImageFolder(root = "pliki_new/Testing", transform =
    transform_test)

train_loader = DataLoader(train_data, batch_size=25, shuffle=True
    )
test_loader = DataLoader(test_data, batch_size=25, shuffle=False)
```

3

MODEL SIECI NEURONOWEJ

W ramach projektu stworzyłam sieć neuronową złożoną z :

- trzech warstw konwolucyjnych,
- warstwy spłaszczającej,
- warstwy "pool" (zmniejszającej wymiar),
- warstwy w pełni połączonej,
- warstwy wyjściowej

Struktura sieci określona jest w sposób następujący:

1. Na wejściu mamy tensor x, jest to tablica wielowymiarowa reprezentująca dane wejściowe (w tym przypadku jest to obraz w postaci liczbowej).
2. Następnie tensor x przechodzi do pierwszej warstwy konwolucyjnej
    a) Warstwa conv1 przyjmuje obraz 3 kanałami (obrazy histopatologiczne najlepiej analizować w modelu przestrzeni barw RGB)
    b) Następnie filtry o rozmiarze 3x3 skanują obraz, generując 32 mapy cech
    c) Na wyjściu dostajemy tensor x o wymiarach [25, 32, 32, 32]
3. Następnie tensor x przechodzi przez pooling
    a) Redukcja wymiaru map cech o połowę
    b) Na wyjściu dostajemy tensor o wymiarach [25, 32, 16, 16]
4. Następnie tensor x przechodzi do drugiej warstwy konwolucyjnej
    a) Warstwa conv2 przyjmuje obraz 32 kanałami
    b) Następnie filtry o rozmiarze 3x3 skanują obraz, generując 64 kanały wyjściowe
    c) Na wyjściu dostajemy tensor x o wymiarach [25, 64, 16, 16]

5. Następnie tensor x ponownie przechodzi przez pooling
    a) Redukcja wymiaru map cech o połowę
    b) Na wyjściu dostajemy tensor o wymiarach [25, 32, 8, 8]
6. Następnie tensor x przechodzi do trzeciej warstwy konwolucyjnej
    a) Warstwa conv3 przyjmuje obraz 64 kanałami
    b) Następnie filtry o rozmiarze 3x3 skanują obraz, generując 64 kanały wyjściowe
    c) Na wyjściu dostajemy tensor x o wymiarach [25, 64, 8, 8]
7. Następnie tensor x ponownie przechodzi przez pooling
    a) Redukcja wymiaru map cech o połowę
    b) Na wyjściu dostajemy tensor o wymiarach [25, 64, 4, 4]
8. Następnie tensor x przechodzi przez warstwę spłaszczającą (flatten)
    a) Spłaszczenie tensoru wielowymiarowego do jednowymiarowego wektora
    b) Na wejściu mamy [25, 64, 4, 4]
    c) Na wyjściu dostajemy tensor [25, 1024] (inaczej [25,64*4*4])
9. Następnie tensor x przechodzi przez warstwę w pełni połączoną
    a) Przekształcenie spłaszczonego wektora o wymiarach 1024 do wektora o wymiarach 64
    b) Na wejściu mamy [25, 1024]
    c) Na wyjściu dostajemy tensor [25, 64]
10. Następnie tensor x przechodzi do warstwy wyjściowej, która ostatecznie taki obraz klasyfikuje.
    a) Ostatecznie przekształcenie do tensoru [25,3] - gdzie każda wartość reprezentuje wynik dla jednej z trzech klas (z jakiego rodzaju zmianą mamy do czynienia)
    b) Na wyjściu dostajemy tensor [25, 3]

Wszystkie warstwy konwolucyjne oraz warstwa w pełni połączona posiadają funkcję aktywacji ReLU (Rectified Linear Unit). Funkcja ReLU jest zdefiniowana jako:

$$f(x) = \max(0, x) \tag{1}$$

Poniżej znajduje się pełna definicja klasy tworzącej CNN.

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, padding = 1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, 3, padding = 1)
        self.conv3 = nn.Conv2d(64, 64, 3, padding= 1)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(64 * 4 * 4, 64)
        self.dropout = nn.Dropout(0.2)
        self.fc2 = nn.Linear(64, 3)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = self.pool(x)
        x = self.flatten(x)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

4

TRENOWANIE SIECI NEURONOWEJ

Aby odpowiednio wytrenować sieć należy w pierwszej kolejności określić funkcję starty oraz optymalizator.
Funkcja straty jest miarą określającą, jak bardzo przewidywania modelu różnią się od rzeczywistych wartości. Im mniejsza wartość funkcji straty, tym model lepiej przewiduje wartości. Minimalizowanie funkcji straty pozwala na poprawę jakości klasyfikacji. W moim projekcie funkcją straty jest Cross Entropy Loss.
Optymalizator z kolei oblicza, jak zmieniać parametry modelu (np. wagi) w celu minimalizacji funkcji straty. Aktualizuje on wagi w kierunku przeciwnym do gradientu straty, przybliżając optymalne rozwiązanie. W projekcie używam jednego z najpopularniejszych optymalizatorów - adam (skrót od Adaptive Moment Estimation).
Optymalizator posiada również współczynnik uczenia (learning rate). Jest to współczynnik określający, jak duże kroki może wykonać optymalizator przy zmianie wag. Wartość tego współczynnika jest bardzo ważna. Jeżeli lr będzie za duże, to optymalizator może omijać najbardziej optymalne rozwiązanie. Z drugiej strony za mała wartość lr może spowodować wydłużenie czasu uczenia sieci oraz optymalne rozwiązanie może utknąć w minimum lokalnym (które nie jest równocześnie globalnym). W moim przypadku lr ma wartość 0.0015.

Poniżej znajduje się kod do definiowania funkcji straty oraz optymalizatora.

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.0015)
```

Proces trenowania stworzonego modelu rozpoczyna się od włączenia "trybu trenowania" sieci. Podczas tego etapu włączona jest dodatkowa warstwa w sieci (warstwa Dropout). Warstwa ta działa w ten sposób, że zeruje pewną część neuronów w każdej iteracji treningu (dla każdej epoki) - w moim przypadku "wyłącza" ona 20% neuronów. Zabieg ten stosowany jest w celu zapobiegania przetrenowaniu sieci (zbyt dobremu dopasowaniu modelu do danych treningowych).

Główną częścią uczenia sieci jest iteracja przez epoki. Podczas jednej epoki przechodzimy przez cały zbiór treningowy podzielony na batche. W ramach każdej z epok są wykonywane następujące kroki:

1. inicjalizacja sumy strat - na początku każdej epoki tworzona jest zmienna, która będzie służyła do zsumowania wartości funkcji

straty obliczonych na poszczególnych batchach. Dzięki temu na koniec epoki można obliczyć średnią stratę, przez co możemy monitorować postępy uczenia sieci (im mniejsza suma strat tym lepiej dla samego modelu).

2. iteracja przez batche - zbiór danych treningowych zostaje podzielony na batche a następnie dla każdej partii danych wykonywane są następujące czynności:

   a) pobranie partii danych wejściowych oraz odpowiadających im etykiet,

   b) przetwarzanie danych przez model i zwrócenie danych wyjściowych,

   c) obliczenie wartości funkcji straty,

   d) obliczenie gradientu - w którym kierunku zmienić parametry, aby zmniejszyć wartość funkcji straty,

   e) aktualizacja parametrów poprzez optymalizator

   f) aktualizacja sumy strat

3. po iteracji przez wszystkie batche w ramach epoki wyświetlana jest suma strat w danej epoce. Pozwala to na monitorowanie postępów uczenia modelu.

Poniżej znajduje się fragment skryptu do trenowania sieci.

```
net.train()
for epoch in range(25):
    loss_epoch = 0
    print(f"Epoka {epoch}")
    for i, batch in enumerate(train_loader, 0):
        inputs, labels = batch
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        loss_epoch += loss.item()
    print(f"epoch = {epoch}, loss = {loss_epoch}")
```

Poniżej znajdują się rezultaty z terminala (wartość starty dla każdej epoki). Jak widać model coraz lepiej się uczy z każdą epoką (z małymi wyjątkami, ale tutaj szukamy minimum funkcji strat).

```
Epoka 0
epoch = 0, loss = 73.20842385292053
Epoka 1
epoch = 1, loss = 48.781455509364605
Epoka 2
epoch = 2, loss = 42.693693444132805
Epoka 3
epoch = 3, loss = 36.75992916896939
Epoka 4
epoch = 4, loss = 34.36781061440706
Epoka 5
epoch = 5, loss = 33.06779783219099
Epoka 6
epoch = 6, loss = 28.889077458530664
Epoka 7
epoch = 7, loss = 24.66039218613878
Epoka 8
epoch = 8, loss = 23.42128170603246
Epoka 9
epoch = 9, loss = 20.201707900501788
Epoka 10
epoch = 10, loss = 19.423787505365908
Epoka 11
epoch = 11, loss = 17.154904041439295
Epoka 12
epoch = 12, loss = 14.13792124320753
Epoka 13
epoch = 13, loss = 12.724811535561457
Epoka 14
epoch = 14, loss = 12.347063398839964
Epoka 15
epoch = 15, loss = 9.486189419170842
Epoka 16
epoch = 16, loss = 8.301392335823039
Epoka 17
epoch = 17, loss = 7.2138326545245945
Epoka 18
epoch = 18, loss = 4.24669130628854
Epoka 19
epoch = 19, loss = 9.614344915332367
Epoka 20
epoch = 20, loss = 2.6458743156617857
Epoka 21
epoch = 21, loss = 2.0450080907230017
Epoka 22
epoch = 22, loss = 4.897275526789599
Epoka 23
epoch = 23, loss = 2.4715835214665276
Epoka 24
epoch = 24, loss = 2.295635212189154
```

5

TESTOWANIE SIECI NEURONOWEJ

Pierwszym krokiem w testowaniu sieci jest wyłączenie w niej "trybu trenowania" i włączenie "trybu testowania". Podczas testowania wyłączana jest warstwa Dropout, która jest przydatna w trybie trenowania sieci.

Następnie następuje ewaluacja, czyli przewidywanie, jak sklasyfikowany powinien zostać dany obraz. W tym przypadku również zbiór testowy zostaje podzielony na batche. Dla każdej partii danych są wykonywane poniższe operacje:

1. pobierane są dane testowe (tensory) i odpowiadające im rzeczywiste etykiety,
2. dane trafiają na wejście sieci i obliczane są dla nich wyjścia,
3. dla wyjść modelu przeprowadzana jest predykcja (dla każdego obrazu wybierana jest najbardziej prawdopodobna klasa),
4. następnie przewidywana etykieta jest porównywana z rzeczywistą
    a) jeżeli etykiety się zgadzają to zwiększamy licznik poprawnych predykcji o 1,
    b) jeżeli etykiety są różne to nie zwiększamy go.
5. ostatnim krokiem jest zapisanie wyjść modelu oraz przewidzianych etykiet do zmiennych output_all oraz labels_all.

Poniżej znajduje się fragment kodu źródłowego, który odpowiada za testowanie sieci.

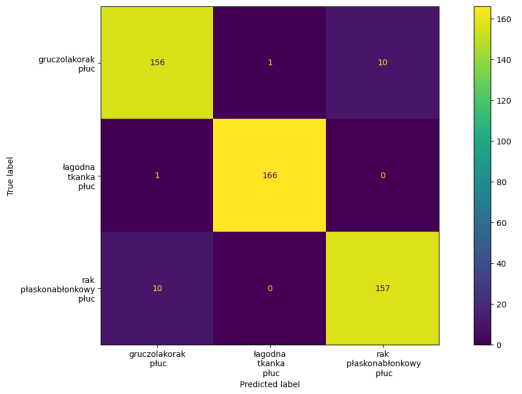
```
correct = 0
total = 0
net.eval()

outputs_all = []
labels_all = []

with torch.no_grad():
    for batch in test_loader:
        images, labels = batch
        outputs = net(images)
        predicted = torch.argmax(outputs, 1)
        total += labels.size(0)
```

```
            correct += (predicted == labels).sum().item()
            outputs_all.append(torch.softmax(outputs, dim=1).
                detach().cpu().numpy())
            labels_all.append(labels.numpy())
```

6

MIARY JAKOŚCI MODELU

Miary jakości modelu w moim projekcie oparte są na macierzy pomyłek (confision matrix). Macierz ta określa zależności pomiędzy etykietami rzeczywistymi a etykietami powstałymi na podstawie predykcji. W przypadku mojego projektu macierz pomyłek będzie rozmiaru 3x3, ponieważ obrazy są klasyfikowane na 3 różne klasy. Poniżej znajduje się macierz pomyłek dla modelu:



W wierszach znajdują się rzeczywiste etykiety danych, natomiast w kolumnach znajdują się etykiety uzyskane z predykcji. W każdej komórce macierzy znajduje się ilość przypadków, w których etykieta rzeczywista jest taka jak w danym wierszu oraz etykieta "predykcyjna" jest taka sama jak etykieta w kolumnie. Na przykład: element macierzy (powiedzmy, że macierz pomyłek ma oznaczenie A) A[1,1] oznacza liczbę obrazów, które rzeczywiście są w klasie gruczolakorak płuc oraz jednocześnie są sklasyfikowane jako gruczolakorak płuc.

Na podstawie macierzy można stwierdzić, ża model całkiem dobrze sklasyfikował obrazy. Istnieje niewiele przypadków (dokładnie 22), gdzie model źle sklasyfikował obrazy :

- istnieje 10 przypadków, kiedy model sklasyfikował gruczolakoraka płuc jako raka płaskonabłonkowego,
- istnieje 1 przypadek, kiedy model sklasyfikował gruczolakoraka płuc jako łagodną tkankę płuc,

- istnieje 1 przypadek, kiedy model sklasyfikował łagodną tkankę płuc jako gruczolakoraka płuc,
- istnieje 10 przypadków, kiedy model sklasyfikował raka płaskonabłonkowego jako gruczolakoraka płuc,

Na podstawie macierzy pomyłek został stworzony raport miar klasyfikacji. Obliczono dla każdej klasy:

- precyzję,

$$\text{Precyzja} = \frac{A[i,i]}{\sum_{j=1}^{n} A[i,j]} \tag{1}$$

- czułość,

$$\text{Czułość} = \frac{A[i,i]}{\sum_{j=1}^{n} A[j,i]} \tag{2}$$

- f1-score,

$$\text{F1-score} = 2 * \frac{Precyzja * \text{Czułość}}{Precyzja + \text{Czułość}} \tag{3}$$

Dodatkowo obliczono dokładność, średnią arytmetyczną powyższych miar oraz średnią ważoną powyższych miar.

	precision	recall	f1-score	support
gruczolakorak płuc	0.93	0.93	0.93	167
łagodna tkanka płuc	0.99	0.99	0.99	167
rak płaskonabłonkowy płuc	0.94	0.94	0.94	167
accuracy			0.96	501
macro avg	0.96	0.96	0.96	501
weighted avg	0.96	0.96	0.96	501

Jak widać powyżej wszystkie miary jakości przekracają 90%, zatem można stwierdzić, że model został całkiem dobrze dopasowany.

Poniżej znajduje się fragment kodu źródłowego do obliczania miar jakości modelu.

```
print(f'ACC: {100 * correct // total}%')


outputs_all = np.concatenate(outputs_all)
labels_all = np.concatenate(labels_all)

print(outputs_all)
print(labels_all)

print(outputs_all.shape)
```

```
print(labels_all.shape)

y_d = np.argmax(outputs_all, axis=1)

def get_label(x):
    return ['gruczolakorak pluc' if z == 0 else 'lagodna tkanka
        pluc' if z == 1 else 'rak plaskonablonkowy pluc' for z in
         x]

cm = confusion_matrix(y_true=get_label(labels_all), y_pred=
    get_label(y_d), labels=['gruczolakorak pluc','lagodna tkanka
    pluc', 'rak plaskonablonkowy pluc'])
print(classification_report(y_true=get_label(labels_all), y_pred=
    get_label(y_d), labels=['gruczolakorak pluc','lagodna tkanka
    pluc', 'rak plaskonablonkowy pluc']))
disp = ConfusionMatrixDisplay(confusion_matrix=cm,display_labels
    =['gruczolakorak\n pluc','lagodna\n tkanka\n pluc', 'rak\n
    plaskonablonkowy\n pluc'])
disp.plot()
plt.show()
```

7

PODSUMOWANIE

W ramach projektu zrealizowano klasyfikację obrazów rentgenowskich płuc na trzy rodzaje zmian: gruczolakorak, łagodne tkanki oraz rak płaskonabłonkowy. Do tego celu zaprojektowano i przeszkolono model konwolucyjnej sieci neuronowej (CNN).

Projekt pokazał, że sieci konwolucyjne dobrze radzą sobie z klasyfikacją obrazów w zadaniach medycznych, takich jak rozpoznawanie zmian nowotworowych. Choć model działał poprawnie, można by go jeszcze ulepszyć, np. przez zastosowanie większych zbiorów danych, rozbudowanie sieci lub użycie innych środków, np. innych funkcji aktywacji.

Realizacja projektu była wartościowym doświadczeniem, które pozwoliło na praktyczne wykorzystanie wiedzy z zakresu uczenia maszynowego i pogłębienie zrozumienia technik analizy obrazów.