Głębokie sieci neuronowe - Projekt

Samodzielna implementacja konwolucyjnej sieci neuronowej wykrywającej chorobę Alzheimera na podstawie obrazów rezonansu magnetycznego

Prowadzący	Autorzy	Nr. Indeksów	Data
Dr hab. inż. Andzej Rusiecki	Patryk Marciniak Wojciech Serewis	248978 244117	30.01.2023

1 Cel projektu

Celem projektu było zaprojektowanie oraz zaimplementowanie od zera konwolucyjnej sieci neuronowej, której zadanie sprowadzałoby się do klasyfikacji obrazów rezonansu magnetycznego do jednej z czterech klas, w zależności od stopnia zaawansowania choroby Alzheimera. Ponadto założono implementację pomocnych funkcjonalności, które ułatwiłyby korzystanie z modelu. W projekcie nie używano żadnych gotowych funkcji z bibliotek związanych z sieciami neuronowymi, głównym narzędziem służącym w implementacji była biblioteka *numpy* umożliwiająca efektywne operacje na macierzach.

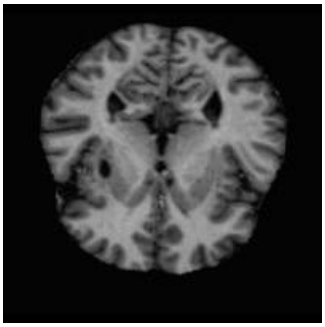
2 Założenia projektowe

2.1 Wykorzystane narzędzia

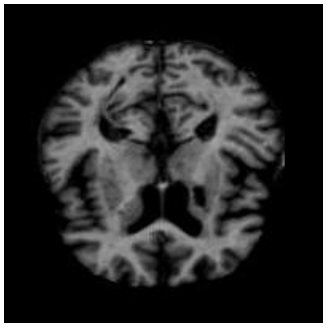
- Python
- JetBrains PyCharm
- GitHub
- numpy

2.2 Użyty zbiór danych

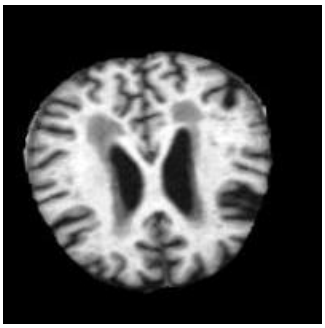
W celu realizacji postawionego celu użyto zbioru obrazów ze strony *kaggle.com*. Obrazy miały różne wymiary, dlatego dla systematyczności wstępnie przetworzono je tak, aby każdy obraz był rozmiarów 200x200 pikseli. Zostało to osiągnięte poprzez dodawanie wierszy i kolumn czarnych pikseli przy granicach obrazów, do osiągnięcia przyjętych wymiarów.



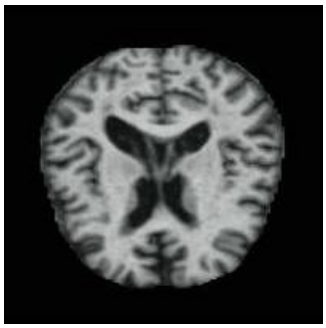
(a) NonDemented



(b) VeryMildDemented



(a) MildDemented



(b) ModerateDemented

2.3 Założenia funkcjonalne

- Zaimplementowana sieć powinna uczyć się na zbiorze danych w celu przeprowadzania klasyfikacji do danych czterech klas
- Sieć powinna w miarę możliwości korzystać z jak największej liczby próbek ze zbioru obrazów do uczenia (po wydzieleniu zbioru walidującego)
- Sieć ma zawierać w sobie następujące warstwy:
  - Konwolucyjna (Convolutional)
  - Poolingująca (Pooling, a konkretniej MaxPooling)
  - Spłaszczająca (Flattening)
  - Dropout
  - Gęsta, w pełni połączona (Dense, Fully Connected)
- Możliwa powinna być odpowiednia parametryzacja warstw:
  - Konwolucyjna — liczba filtrów oraz ich wymiary
  - Poolingująca — rozmiar okna wykonującego operację
  - Dropout — prawdopodobieństwo odrzucenia neuronu
  - Gęsta — liczba neuronów znajdująca się w warstwie

  Ponadto dla warstwy konwolucyjnej oraz gęstej powinno być możliwe wybranie funkcji aktywacji wraz z jej pochodną. W przypadku warstwy konwolucyjnej postanowiono pominąć parametryzację stride'a oraz paddingu
- Wytrenowaną sieć powinno się dać zapisać w pliku, a także powinno być możliwe odczytanie takiego pliku i wczytanie modelu do programu - w celu dalszego uczenia lub wykorzystania do klasyfikacji

2.4 Architektura sieci

Założono następującą architekturę sieci (kierunek przepływu sygnału - z góry do dołu):

1. Convolutional (3x3 kernel, 8 filters) + ReLU
2. MaxPooling (2x2)
3. Convolutional (3x3 kernel, 12 filters) + ReLU
4. MaxPooling (2x2)
5. Convolutional (3x3 kernel, 16 filters) + ReLU
6. MaxPooling (2x2)
7. Flattening
8. Dense (512 neurons) + ReLU
9. Dropout (25%)
10. Dense (4 neurons) + SoftMax

3 Implementacja sieci

Sieć została zaimplementowana obiektowo, każda warstwa została zaprojektowana jako odmienna klasa z konstruktorem oraz metodą inicjalizującą parametry na podstawie rozmiarów sygnału wejściowego oraz parametrów zadanych przy tworzeniu obiektu.

```
def init_params(self, input_shape):
    if len(input_shape) == 4:
        self.input_shape = input_shape

    elif len(input_shape) == 3:
        self.input_shape = input_shape + (1, )

    elif len(input_shape) == 2:
        self.input_shape = input_shape + (1, 1)

    else:
        raise ValueError('ConvolutionalLayer input must be of dimensions 2, 3 or 4.')

    self.kernel_shape = (self.n_filters,  # Number of filters
                         self.input_shape[2],  # Number of channels in images
                         self.filter_shape[0],  # Kernel height
                         self.filter_shape[1])  # Kernel width

    # Prepare padded input
    self.padding = self.kernel_shape[2] // 2

    # Initialization of filters and biases using normal distribution
    standard_dev = 1 / np.sqrt(np.prod(self.kernel_shape))
    self.weights = np.random.normal(0, standard_dev, self.kernel_shape)
    self.biases = np.random.randn(self.n_filters)

    # Layer output and its shape assuming (1, 1) strides and padding that won't cause size change
    self.output_shape = (self.input_shape[0],  # Input height
                         self.input_shape[1],  # Input width
                         self.n_filters,  # Number of filters (channels)
                         self.input_shape[3])  # Number of inputs

    # Prepare delta variables for backpropagation
    self.delta_weights = np.zeros(self.weights.shape)
    self.delta_biases = np.zeros(self.biases.shape)
```

Rysunek 3: Fragment kodu przykładowej funkcji inicjującej parametry

3.1 Warstwa konwolucyjna — Convolutional

Do implementacji najważniejszej warstwy, od której bierze się nazwa konwolucyjnych sieci neuronowych użyto odpowiednich operacji na macierzach w celu przyspieszenia działania. Metoda ta pozwala na przekształcenie dowolnej macierzy lub obszaru macierzy w kolumną. Co za tym idzie możliwej jest przedstawienie wielu macierzy o jednakowych rozmiarach (np. obrazu i jego kanałów) za pomocą jednej. Dzięki temu całą operację konwolucji wszystkich kanałów obrazu ze wszystkimi filtrami można przeprowadzić za pomocą pojedynczego mnożenia macierzy. W warstwie konwolucyjnej dostrajane w procesie uczenia będą wartości znajdujące się w poszczególnych filtrach, oraz wartości polaryzacji (biasu, po jednej na każdy filtr).

```python
def forward_prop(self, layer_input):
    h, w, c, n = self.input_shape

    if len(layer_input.shape) == 2:  # First layer case
        self.input = layer_input[:, :, np.newaxis, np.newaxis]

    else:  # Not first layer case
        self.input = layer_input.reshape(self.input_shape)

    input_reshaped = self.input.transpose(3, 2, 0, 1)
    convo_result_shape = self.n_filters, h, w, n

    kernel_h = self.kernel_shape[2]  # Kernel height
    kernel_w = self.kernel_shape[3]  # Kernel width

    # Convert images into columns
    self.input_col = im2col_indices(input_reshaped, kernel_h, kernel_w, padding=self.padding, stride=1)
    weights_col = self.weights.reshape(self.n_filters, -1)

    # Perform convolution
    out = weights_col @ self.input_col + self.biases[:, np.newaxis]
    out = out.reshape(convo_result_shape).transpose(1, 2, 0, 3)

    # Activate outputs
    self.output = self.activation(out)

    return self.output
```

Rysunek 4: Propagacja sygnału w przód w warstwie konwolucyjnej

```python
def backward_prop(self, next_layer):
    h, w, c, n = self.input_shape
    shape_for_c2i = n, c, h, w

    if len(next_layer.delta.shape) == 3:
        delta_nl = next_layer.delta[:, :, :, np.newaxis]

    else:
        delta_nl = next_layer.delta

    kernel_h = self.kernel_shape[2]  # Kernel height
    kernel_w = self.kernel_shape[3]  # Kernel width

    # Get delta term from next layer in needed shapes
    delta_nxt_layer = delta_nl.transpose(3, 2, 0, 1)
    delta_result_shaped = delta_nxt_layer.transpose(1, 2, 3, 0).reshape(self.n_filters, -1)

    # Determine delta term of biases
    self.delta_biases += np.sum(delta_nxt_layer, axis=(0, 2, 3))

    # Determine delta term of weights
    col_delta_weights = delta_result_shaped @ self.input_col.T
    self.delta_weights += col_delta_weights.reshape(self.weights.shape)

    # Get weights in needed shape
    weights_reshaped = self.weights.reshape(self.n_filters, -1)

    # Determine delta term of inputs
    delta_col = weights_reshaped.T @ delta_result_shaped
    delta = col2im_indices(delta_col, shape_for_c2i, kernel_h, kernel_w, padding=self.padding)
    self.delta = self.activation_deriv(delta.transpose(2, 3, 1, 0))
```

Rysunek 5: Propagacja wsteczna błędów w warstwie konwolucyjnej

3.2 Warstwa poolingująca — MaxPooling

Podobnie, jak w przypadku warstwy konwolucyjnej, warstwa poolingująca korzysta z podobnej "sztuczki" pozwalającej na znaczne przyspieszenie obliczeń Poprzez ustawienie wszystkich pikseli z danego obszaru 2x2 (okno poolingu) wzdłuż jednej osi, wykonać można pojedynczy wybór wartości maksymalnych. Warstwa poolingująca nie ma parametrów, które będą się zmieniać pod wpływem uczenia.

Zaimplementowana warstwa realizuje założenie warstwy MaxPooling - z okna 2x2 wybierana jest wartość maksymalna. Warto tu jednak zaznaczyć, że w niniejszej implementacji wybierany jest zawsze jeden piksel, niezależnie od tego ile pikselów ma wartość maksymalną.

```
def forward_prop(self, layer_input):
    self.input = layer_input.reshape(self.input_shape)

    input_reshaped = self.input.transpose(3, 2, 0, 1)

    # Get input dimensions (shape)
    h, w, c, n = self.input_shape

    # Get shape of pooling kernel
    h_pool = self.kernel_shape[0]
    w_pool = self.kernel_shape[1]

    # Determine output shape
    h_out = (h - h_pool) // h_pool + 1
    w_out = (w - w_pool) // w_pool + 1

    input_split = input_reshaped.reshape(n * c, 1, h, w)
    self.input_cols = im2col_indices(input_split, h_pool, w_pool, padding=0, stride=h_pool)

    # Get indices of max values and save those values
    input_cols_argmax = np.argmax(self.input_cols, axis=0, keepdims=True)
    input_cols_max = self.input_cols[input_cols_argmax, np.arange(input_cols_argmax.shape[1])]

    # Get a proper shape
    self.output = input_cols_max.reshape(h_out, w_out, n, c).transpose(0, 1, 3, 2)

    return self.output
```

Rysunek 6: Propagacja sygnału w przód w warstwie poolingującej

```python
def backward_prop(self, next_layer):
    delta_nl_reshaped = next_layer.delta.transpose(3, 2, 0, 1)

    # Get input dimensions (shape)
    h, w, c, n = self.input_shape

    # Get shape of pooling kernel
    h_pool = self.kernel_shape[0]
    w_pool = self.kernel_shape[1]

    delta_nl_trans = delta_nl_reshaped.transpose(2, 3, 0, 1).flatten()
    delta_cols = np.zeros_like(self.input_cols)

    input_cols_argmax = np.argmax(self.input_cols, axis=0, keepdims=True)
    delta_cols[input_cols_argmax, np.arange(input_cols_argmax.shape[1])] = delta_nl_trans

    # Determine delta term for this layer
    delta = col2im_indices(delta_cols, (n * c, 1, h, w), h_pool, w_pool, padding=0, stride=h_pool)

    input_reshaped = self.input.transpose(3, 2, 0, 1)
    self.delta = delta.reshape(input_reshaped.shape).transpose(2, 3, 1, 0)
```

Rysunek 7: Propagacja wsteczna błędów w warstwie poolingującej

3.3 Warstwa spłaszczająca — Flattening

Warstwa spłaszczająca jest chyba najprostszą warstwą w modelu - polega jedynie na sprowadzeniu wszystkich wejść, niezależnie od wymiarów do pojedynczego wektora. Warstwa spłaszczająca nie ma parametrów, które będą się zmieniać pod wpływem uczenia.

```python
def forward_prop(self, layer_input):
    # Perform flattening
    self.input = layer_input
    self.output = self.input.flatten().reshape(1, -1)

    return self.output
```

Rysunek 8: Propagacja sygnału w przód w warstwie spłaszczającej

```python
def backward_prop(self, next_layer):
    # Compute error from downstream and determine this layers delta term
    self.error = np.dot(next_layer.weights, next_layer.delta.T).T
    self.delta = self.error * self.output
    self.delta = self.delta.reshape(self.input_shape)
```

Rysunek 9: Propagacja wsteczna błędów w warstwie spłaszczającej

3.4 Warstwa głęboka — Dense

Warstwa głęboka jest najprostszą warstwą stosowaną w sieciach neuronowych, posiada macierz wag, o wymiarach zależnych od wejścia oraz liczby neuronów. Wektor wejściowy o wymiarze n jest mnożony przez macierz wag $m \times n$ i po dodaniu polaryzacji (biasu) otrzymuje się wektor wyjściowy o wymiarze n (po użyciu funkcji aktywacji). Macierz wag oraz polaryzacja będą w tej warstwie parametrami, które będą dostrajane w procesie uczenia.

```python
def forward_prop(self, layer_input):
    self.input = layer_input

    # Dot product of input and neuron weights plus bias values
    dense_output = np.dot(layer_input, self.weights) + self.biases
    # Activate output using provided function
    self.output = self.activation(dense_output)

    return self.output
```

Rysunek 10: Propagacja sygnału w przód w warstwie gęstej

```python
def backward_prop(self, next_layer):
    # If the next layer is Dropout just get the error
    if type(next_layer).__name__ == 'DropoutLayer':
        self.error = next_layer.error

    # If not compute error from downstream
    else:
        self.error = np.dot(next_layer.weights, next_layer.delta.T).T

    # Determine this layers delta term
    self.delta = self.error * self.activation_deriv(self.output)

    # Determine delta terms for weights and biases
    self.delta_weights += self.delta * self.input.T
    self.delta_biases += self.delta
```

Rysunek 11: Propagacja wsteczna błędów w warstwie gęstej

3.5 Warstwa Dropout

Warstwa Dropout powoduje dezaktywowanie pewnej ustalonej części neuronów, przy pozostawieniu pozostałych neuronów w takim samym stanie, jak na wejściu. Warstwa jest ważnym elementem modelu i zmniejsza prawdopodobieństwo wystąpienia w nim przeuczenia, czyli zbyt mocnego przystosowania się do zbioru uczącego. Warstwa nie posiada żadnych parametrów, które będą dostrajane w procesie uczenia.

```
def forward_prop(self, layer_input):
    self.input = layer_input

    # Dot product of input and neuron weights plus bias values
    dense_output = np.dot(layer_input, self.weights) + self.biases
    # Activate output using provided function
    self.output = self.activation(dense_output)

    return self.output
```

Rysunek 12: Propagacja sygnału w przód w warstwie gęstej

```
def backward_prop(self, next_layer):
    # If the next layer is Dropout just get the error
    if type(next_layer).__name__ == 'DropoutLayer':
        self.error = next_layer.error

    # If not compute error from downstream
    else:
        self.error = np.dot(next_layer.weights, next_layer.delta.T).T

    # Determine this layers delta term
    self.delta = self.error * self.activation_deriv(self.output)

    # Determine delta terms for weights and biases
    self.delta_weights += self.delta * self.input.T
    self.delta_biases += self.delta
```

Rysunek 13: Propagacja wsteczna błędów w warstwie gęstej

3.6 Optymalizacja parametrów

Dostrajanie parametrów sieci odbywa się za pomocą optymalizatora ADAM. Został on zaimplementowany na podstawie ogólno dostępnych pseudokodów opisujących jego działanie.

```
def adam(self):
    for i, layer in enumerate(self.layers):
        if not hasattr(layer, 'weights'):
            continue

        if not self.training:
            self.reset_adam_params()

        else:
            self.ts[i] += 1
            t = self.ts[i]

            # Get weights' and biases' moments
            self.weights_adam1[i], self.biases_adam1[i] = self.compute_moment(self.beta1, i)
            self.weights_adam2[i], self.biases_adam2[i] = self.compute_moment(self.beta2, i)

            # Get needed parameters
            w_mcap, b_mcap = self.get_mcaps(i, t)
            w_vcap, b_vcap = self.get_vcaps(i, t)

            # Adjust weights
            layer.delta_weights = w_mcap / (np.sqrt(w_vcap) + self.eps)
            layer.weights += self.learning_rate * layer.delta_weights

            # Adjust biases
            layer.delta_biases = b_mcap / (np.sqrt(b_vcap) + self.eps)
            layer.biases += self.learning_rate * layer.delta_biases
```

Rysunek 14: Implementacja metody realizującej funkcjonalność optymalizatora ADAM

3.7 Przygotowanie danych wejściowych

W projekcie założone, że dane wejściowe znajdują się w pewnym katalogu, podzielonym na podkatalogi - po jednym na każdą klasę. W pojedynczym podkatalogu powinny znajdować się wszystkie obrazy należące do danej klasy.

W skrócie funkcja wczytuje określoną ilość zdjęć z pierwszego katalogu, poprzez najpierw pobranie danej liczby nazw, a następnie otworzenie tych obrazów za pomocą bilbioteki *opencv* jako obrazów w skali szarości i zapisanie ich w formie tablicy. Następnie tablica zapisywana jest na liście obrazów, a na odpowiadającej jej liście etykiet zapisywana jest odpowiednia etykieta. Proces powtarza się najpierw dla każdego obrazu, a potem dla każdego podkatalogu. Na koniec zwracane są obie listy, po przekonwertowaniu je na tablice (pary macierz-wektor odpowiadające parom obraz-etykieta).

```python
def prepare_data(dir_path_to_data, n_samples=0):
    if not os.path.exists(dir_path_to_data):
        raise FileNotFoundError(f'Directory {dir_path_to_data} does not exist.')

    # Get directories which correspond to classes
    classes = next(os.walk(dir_path_to_data))[1]

    label_list = []
    image_list = []

    for class_index, class_name in enumerate(classes):
        start_time = time.time()
        class_dir_path = f'{dir_path_to_data}/{class_name}'

        samples = os.listdir(class_dir_path)[:n_samples] if n_samples > 0 else os.listdir(class_dir_path)

        for file in samples:
            img_path = f'{class_dir_path}/{file}'
            img = np.asarray(cv2.imread(img_path, cv2.IMREAD_GRAYSCALE))

            label_list.append(class_index)
            image_list.append(img)

        print(f'Loading images from class {class_name} done in {round(time.time() - start_time, 3)} seconds.')

    image_array = np.array(image_list)

    if n_samples == 0:
        label_array = np.zeros((len(label_list), len(classes)))
    else:
        label_array = np.zeros((n_samples * len(classes), len(classes)))

    for i, j in enumerate(label_list):
        label_array[i, j] = 1

    return image_array, label_array
```

Rysunek 15: Funkcja przygotowująca dane do zadania na wejście sieci neuronowej

3.8 Budowa sieci

W celu zbudowania sieci w pierwszej kolejności należy stworzyć odpowiednie warstwy pamiętając o podaniu wartości odpowiednich parametrów, a następnie dodać je do modelu sieci. Możliwe również jest pojedyncze tworzenie i dodawanie warstw do sieci

```
final_model = [
    # Convolutional, 3x3, 8 fiters, ReLU
    ConvolutionalLayer(input_shape=(200, 200),
                       n_filters=8,
                       kernel_shape=(3, 3),
                       activation=funs.relu,
                       activation_deriv=funs.relu_prime),
    # Max Pooling, 2x2
    MaxPoolingLayer(),
    # Convolutional, 3x3, 12 filters, ReLU
    ConvolutionalLayer(n_filters=12,
                       kernel_shape=(3, 3),
                       activation=funs.relu,
                       activation_deriv=funs.relu_prime),
    # Max Pooling 2x2
    MaxPoolingLayer(),
    # Convolutional, 3x3, 16 filters, ReLU
    ConvolutionalLayer(n_filters=16,
                       kernel_shape=(3, 3),
                       activation=funs.relu,
                       activation_deriv=funs.relu_prime),
    # Max Pooling 2x2
    MaxPoolingLayer(),
    # Flattening to (1, n)
    FlatteningLayer(),
    # Dense, 512, ReLU
    DenseLayer(n_neurons=512,
               activation=funs.relu,
               activation_deriv=funs.relu_prime),
    # Dropout 25%
    DropoutLayer(probability=0.25),
    # Dense, 4, SoftMax
    DenseLayer(n_neurons=4,
               activation=funs.softmax,
               activation_deriv=funs.softmax_prime)
]
```

Rysunek 16: Stworzenie potrzebnych warstw na potrzeby implementacji

```
x, y = prepare_data('images/augmented', 1024)
layers_list = final_model

cnn = Network()

for lay in layers_list:
    cnn.add(lay)

cnn.compile()
cnn.summary()

cnn.train(inputs=x,
          correct_outputs=y,
          epochs=2,
          batch_size=8,
          shuffle=False,
          validation_split=0.25)

cnn.save_to_json('models/model_001.json')
```

Rysunek 17: Stworzenie modelu sieci, dodanie do niego warstw, kompilacja modelu, wyświetlenie podsumowania, trening sieci oraz zapis modelu do pliku

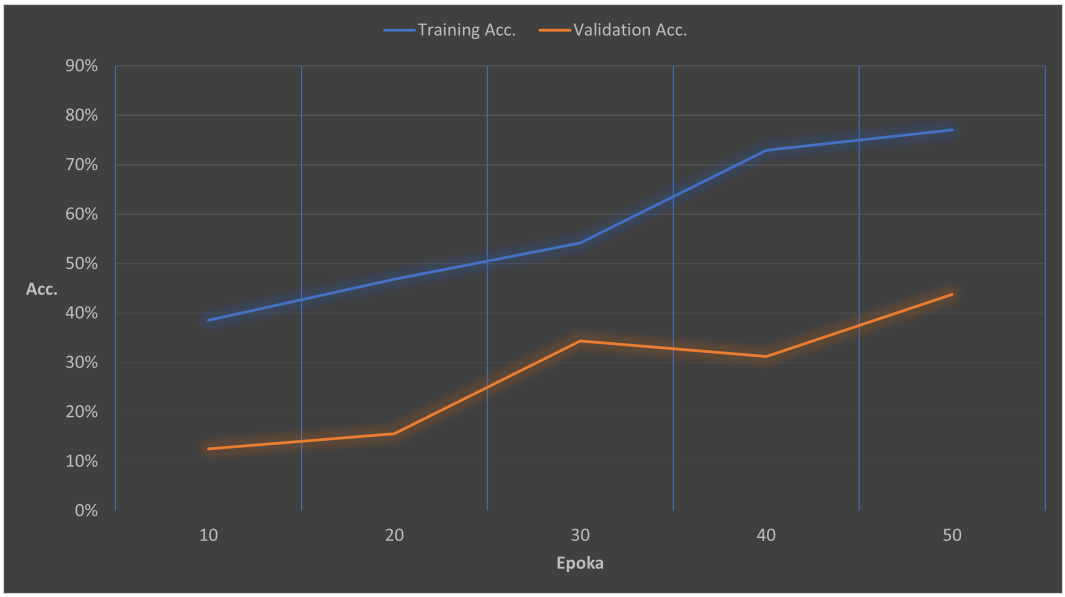
```
cnn = load_from_json('models/model_001.json')

test_imgs = [
    cv2.imread('images/augmented/MildDemented/ff951dd6-f361-41d0-b6c4-2de07ab87490.jpg', cv2.IMREAD_GRAYSCALE),
    cv2.imread('images/augmented/ModerateDemented/f41afea6-1e7c-4a4b-b7d2-5eb170fa43b4.jpg', cv2.IMREAD_GRAYSCALE),
    cv2.imread('images/augmented/NonDemented/db3edf65-9f53-4662-90c1-54765ec0d0c1.jpg', cv2.IMREAD_GRAYSCALE),
    cv2.imread('images/augmented/VeryMildDemented/e93ac360-2788-41e2-bfd3-420bdb8654e4.jpg', cv2.IMREAD_GRAYSCALE)
]

result = cnn.classify(input_for_classification=test_imgs)

print([np.round(i, 2) for i in np.squeeze(result)])
```

Rysunek 18: Przykład wykorzystania wczytania modelu z pliku i użycia go do przeprowadzenia klasyfikacji

4 Uzyskane wyniki

W celu sprawdzenia poprawności implementacji sieci konwolucyjnej przeprowadzono kilka eksperymentów, używając różnych wartości parametrów. Empirycznie wyznaczono, że najlepiej sprawdzającą się wartością parametru *learning_rate* jest 0.0001.

4.1 Batch size = 16, 50 epok, 32 obrazów z każdej klasy (128)

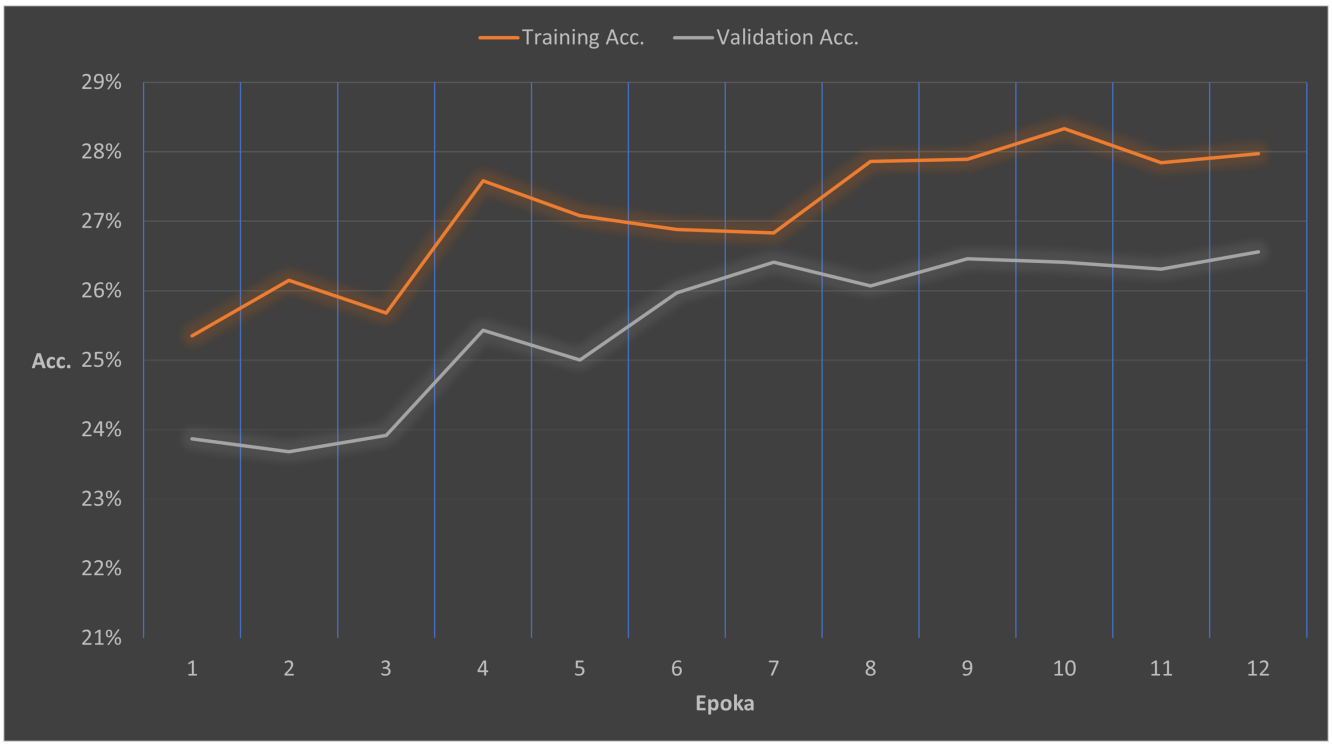


Rysunek 19: Wykres przedstawiający zmianę dokładności klasyfikacji dla zbioru treningowego oraz walidującego w zależności od epoki dla danych parametrów

4.2 Batch size = 32, 20 epok, 256 obrazów z każdej klasy (1024)

Rysunek 20: Wykres przedstawiający zmianę dokładności klasyfikacji dla zbioru treningowego oraz walidującego w zależności od epoki dla danych parametrów

4.3 Batch size = 64, 12 epok, 2048 obrazów z każdej klasy (8192)



Rysunek 21: Wykres przedstawiający zmianę dokładności klasyfikacji dla zbioru treningowego oraz walidującego w zależności od epoki dla danych parametrów

5 Wnioski

- Jak wynika z wykresów, sieć ogółem jest się w stanie uczyć, jednak jest to dość czasochłonny proces
- Udało się przyspieszyć uczenie sieci dzięki zastosowaniu operacji na macierzach konwertujących je na kolumny, w celu wykonania poolingu lub konwolucji za pomocą pojedynczego mnożenia macierzy
- Sieć spełnia postawione założenia, jednak można powiedzieć, że jest mało efektywna – proces uczenia jest czasochłonny
- Postawione zadanie zostało zrealizowane, choć zaimplementowane rozwiązanie jest mało efektywne, sprawdza się o wiele gorzej, niż znane biblioteki przeznaczone do tworzenia i uczenia sieci neuronowych (TensorFlow, PyTorch)
- Implementacja sieci neuronowej od podstaw była dość problematyczna i wymagała szerokiego rozeznania w temacie, najbardziej problematycznym elementem projektu była wsteczna propagacja błędów, a konkretnie jej implementacja w warstwie konwolucyjnej