Politechnika
Wrocławska

Głębokie Sieci Neuronowe – Projekt

Własna implementacja konwolucyjnej sieci neuronowej

Autorzy:
Patryk Marciniak 248978
Wojciech Serewis 244117

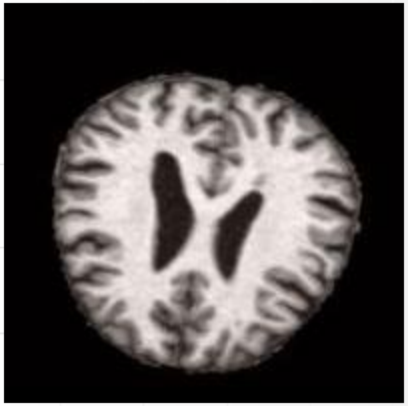
Prowadzący:
dr hab. inż. Andrzej Rusiecki

Plan prezentacji

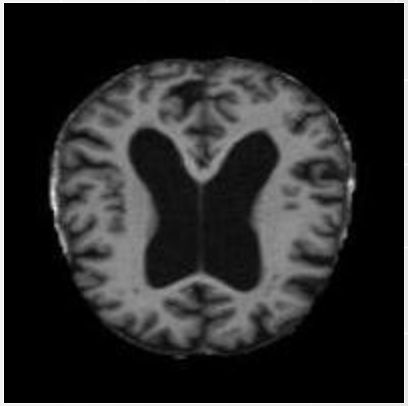
- Cel projektu
- Założenia projektowe
- Projekt i implementacja sieci
- Wyniki i podsumowanie

Cel projektu

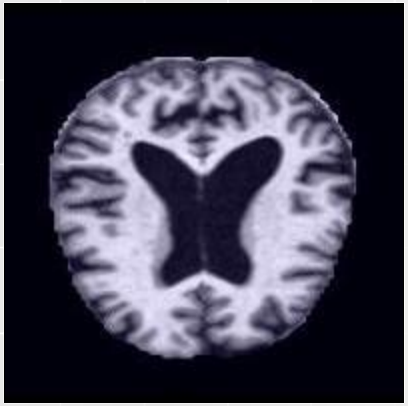
- Głównym celem projektu było samodzielne zaprojektowanie i zaimplementowanie konwolucyjnej sieci neuronowej, w celu klasyfikacji obrazów rezonansu magnetycznego, podzielonych na klasy w zależności od stopnia rozwinięcia choroby Alzheimera.



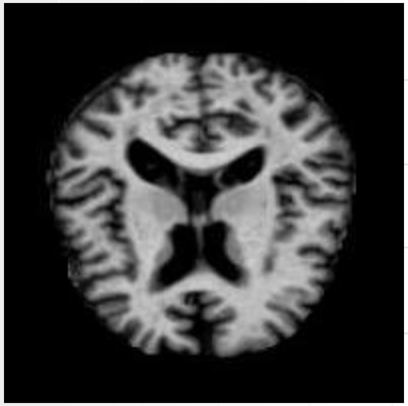
NonDemented



VeryMildDemented



MildDemented



ModerateDemented

Założenia projektowe

- Sieć powinna nauczyć się klasyfikować obrazy MRI i jako wyjście ma podawać prawdopodobieństwa przynależności obrazu do danych klas
- Sieć ma zawierać w sobie następujące warstwy:
  - Convolutional
  - Pooling (Max Pooling)
  - Flattening
  - Dropout
  - Dense (Fully Connected)
- Sieć powinna umożliwiać modyfikację najważniejszych parametrów warstw
- Sieć powinna umożliwiać zapis modelu do pliku oraz jego odczyt

Architektura sieci

- Architektura zaimplementowanej sieci jest luźno wzorowana jest na architekturze VGG, jest jednak znacznie uproszczona, głownie z powodu ograniczeń czasowych i sprzętowych
- Zaplanowano, aby następujące warstwy wchodziły w skład sieci:
  - Conv3x3 (8 filters)
  - MaxPooling (2x2)
  - Conv3x3 (12 filters)
  - MaxPooling (2x2)
  - Conv3x3 (16 filters)
  - MaxPooling (2x2)
  - Flattening
  - Dense (1024)
  - Dropout (25%)
  - Dense (4)
  - SoftMax

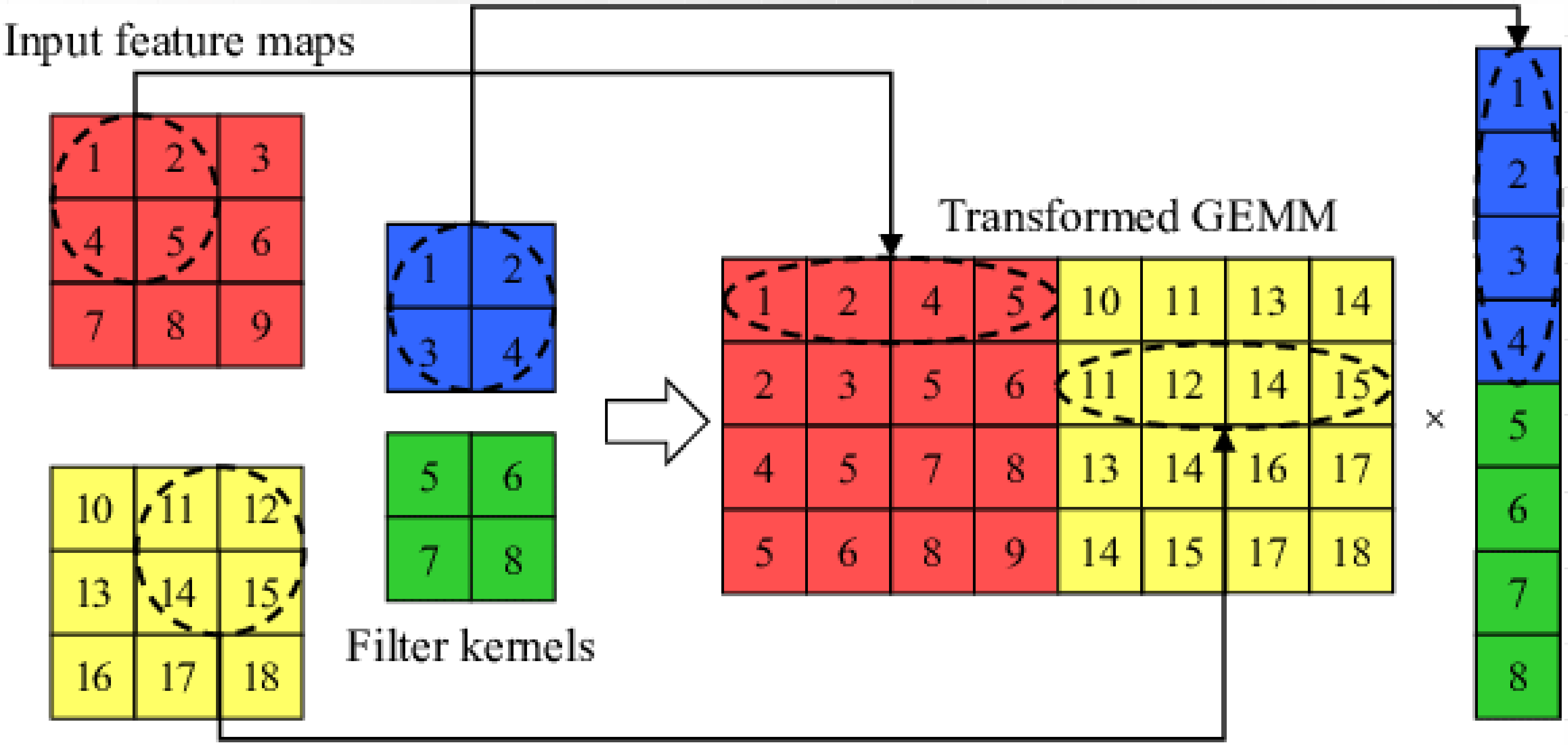
Warstwa konwolucyjna – podejście klasyczne

```python
def forward_prop(self, layer_input):
    self.input = np.atleast_3d(layer_input)

    # Apply zero padding
    self.padded_input[self.padding: -self.padding, self.padding: -self.padding] = self.input

    # For each filter in layer
    for f in range(self.n_filters):
        # For each row
        for r in range(self.input_shape[0]):
            r_end = r + self.kernel_shape[0]

            # For each column
            for c in range(self.input_shape[1]):
                c_end = c + self.kernel_shape[1]

                # Get a chunk of the padded input array
                chunk = self.padded_input[r: r_end, c: c_end]

                # Perform convolution
                convolution_output = (chunk * self.weights[:, :, :, f]).sum() + self.biases[f]
                self.output[r, c, f] = convolution_output

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

    return self.output
```

Warstwa konwolucyjna – podejście klasyczne

```
def backward_prop(self, next_layer):
    self.delta = np.zeros(self.input_shape)

    # For every filter
    for f in range(self.n_filters):
        # For every row
        for r in range(self.kernel_shape[0], self.input_shape[0] + 1):
            r_start = r - self.kernel_shape[0]

            # For every column
            for c in range(self.kernel_shape[1], self.input_shape[1] + 1):
                c_start = c - self.kernel_shape[1]

                # Get a chunk of the input array
                chunk = self.input[r_start: r, c_start: c]

                # Determine delta terms for weights and biases
                self.delta_weights[:, :, :, f] += chunk * next_layer.delta[r_start, c_start, f]
                self.delta[r_start: r, c_start: c] += next_layer.delta[r_start, c_start, f] * self.weights[:, :, :, f]

        self.delta_biases[f] = np.sum(next_layer.delta[:, :, f])
    self.delta = self.activation_deriv(self.delta)
```

Warstwa konwolucyjna – wykorzystanie operacji na macierzach



Warstwa konwolucyjna – operacje na macierzach

```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
```

Warstwa konwolucyjna – operacje na macierzach

```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))
```

Warstwa MaxPooling

```python
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
```

Warstwa MaxPooling

```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)
```

Warstwa Dense

```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

AppleSoju
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
```

Uczenie sieci – metoda *train()*

```
for epoch in range(epochs):
    errors = []

    for batch_num, batch in enumerate(batches):
        batch_loss = 0

        xs, ys = inputs[batch], correct_outputs[batch]

        batch_t = time.time()
        for i, xy in enumerate(zip(xs, ys)):
            x, y = xy

            output = self.forward_propagation(x)
            out = np.squeeze(output)
            loss, error = self.cross_entropy_loss(correct_output=y, network_output=out)

            batch_loss += loss
            errors.append(error)

            update = False
            if i == batch_size - 1:
                update = True
                loss = batch_loss / batch_size

            self.backward_propagation(loss, update)

        print(f'Batch {batch_num + 1} of Epoch {epoch + 1} done in {round(time.time() - batch_t, 2)}.')
        batch_t = time.time()

    train_output = self.classify(inputs[indices])
    train_loss, train_error = self.cross_entropy_loss(correct_outputs[indices], train_output)

    train_accuracy = np.squeeze(train_output.argmax(axis=2)) == correct_outputs[indices].argmax(axis=1)

    self.training_loss[epoch] = round(train_error.mean(), 4)
    self.training_accuracy[epoch] = round(train_accuracy.mean() * 100, 4)
```

Uczenie sieci – optymalizator ADAM

```python
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
```

Dodatkowe funkcje – podsumowanie

Layer Name	Input	Output	Activation Function
Convolutional Layer	(200, 200, 1, 1)	(200, 200, 8, 1)	relu
MaxPooling Layer	(200, 200, 8, 1)	(100, 100, 8, 1)	None
Convolutional Layer-2	(100, 100, 8, 1)	(100, 100, 12, 1)	relu
MaxPooling Layer-2	(100, 100, 12, 1)	(50, 50, 12, 1)	None
Convolutional Layer-3	(50, 50, 12, 1)	(50, 50, 16, 1)	relu
MaxPooling Layer-3	(50, 50, 16, 1)	(25, 25, 16, 1)	None
Flattening Layer	(25, 25, 16, 1)	(1, 10000)	None
Dense Layer	10000	(1, 512)	relu
Dropout Layer	(1, 512)	(1, 512)	None
Dense Layer-2	512	(1, 4)	softmax

Total number of images: 1024

Number of training samples: 768

Number of validation saples: 256

Number of batches: 24

Size of one batch: 32

Dodatkowe funkcje – zapis do pliku

```python
def save_to_json(self, path='model.json'):
    if not os.path.exists('/'.join(path.split('/')[:-1])):
        raise FileNotFoundError('Given path does not exist')

    dict_model = {'model': str(type(self).__name__)}

    to_save = ['name', 'n_neurons', 'input_shape', 'output_shape',
               'weights', 'biases', 'activation', 'activation_deriv', 'n_filters',
               'kernel_shape', 'probability']

    for layer in self.layers:
        current_layer = vars(layer)

        values = {'type': str(type(layer).__name__)}
        for key, val in current_layer.items():
            if key in to_save:
                if key in ['weights', 'biases']:

                    try:
                        val = val.tolist()
                    except:
                        val = float(val)

                if type(val) == np.int32:
                    val = float(val)

                if key == 'input_shape' or key == 'output_shape':
                    try:
                        val = tuple(val)
                    except:
                        pass
                if key == 'activation' or key == 'activation_deriv':
                    val = val.__name__
                values[key] = val
        dict_model[layer.name] = values

    json_dict = json.dumps(dict_model)
```

Dodatkowe funkcje – odczyt z pliku

```
for layer, params in dict_model.items():
    if layer != 'model':
        layer_type = layers[params['type']]

        if layer_type == ConvolutionalLayer:
            lay = layers[params['type']](
                input_shape=params['input_shape'],
                n_filters=params['n_filters'],
                kernel_shape=(params['kernel_shape'][2], params['kernel_shape'][3]),
                activation=functions[params['activation']],
                activation_deriv=functions[params['activation_deriv']]
            )
            lay.weights = np.array(params['weights'])
            lay.biases = np.array(params['biases'])

        elif layer_type == MaxPoolingLayer:
            lay = layers[params['type']](input_shape=params['input_shape'])

        elif layer_type == FlatteningLayer:
            lay = layers[params['type']](input_shape=params['input_shape'])

        elif layer_type == DenseLayer:
            lay = layers[params['type']](input_shape=(1, params['input_shape']),
                                         n_neurons=params['n_neurons'],
                                         activation=functions[params['activation']],
                                         activation_deriv=functions[params['activation_deriv']])
            lay.weights = np.array(params['weights'])
            lay.biases = np.array(params['biases'])

        elif layer_type == DropoutLayer:
            lay = layers[params['type']](input_shape=params['input_shape'],
                                         probability=params['probability'])
            lay.n_neurons = params['n_neurons']

        else:
            raise TypeError('Unknown Layer type detected.')

        model.add(lay)
print(f'Model loaded from {path}')
```

Działanie sieci – zdefiniowanie warstw i modelu

```python
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)
]
```

```python
x, y = prepare_data('images/augmented', 256)
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=20,
          batch_size=32,
          shuffle=False,
          validation_split=0.25)

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

Działanie sieci – wyniki

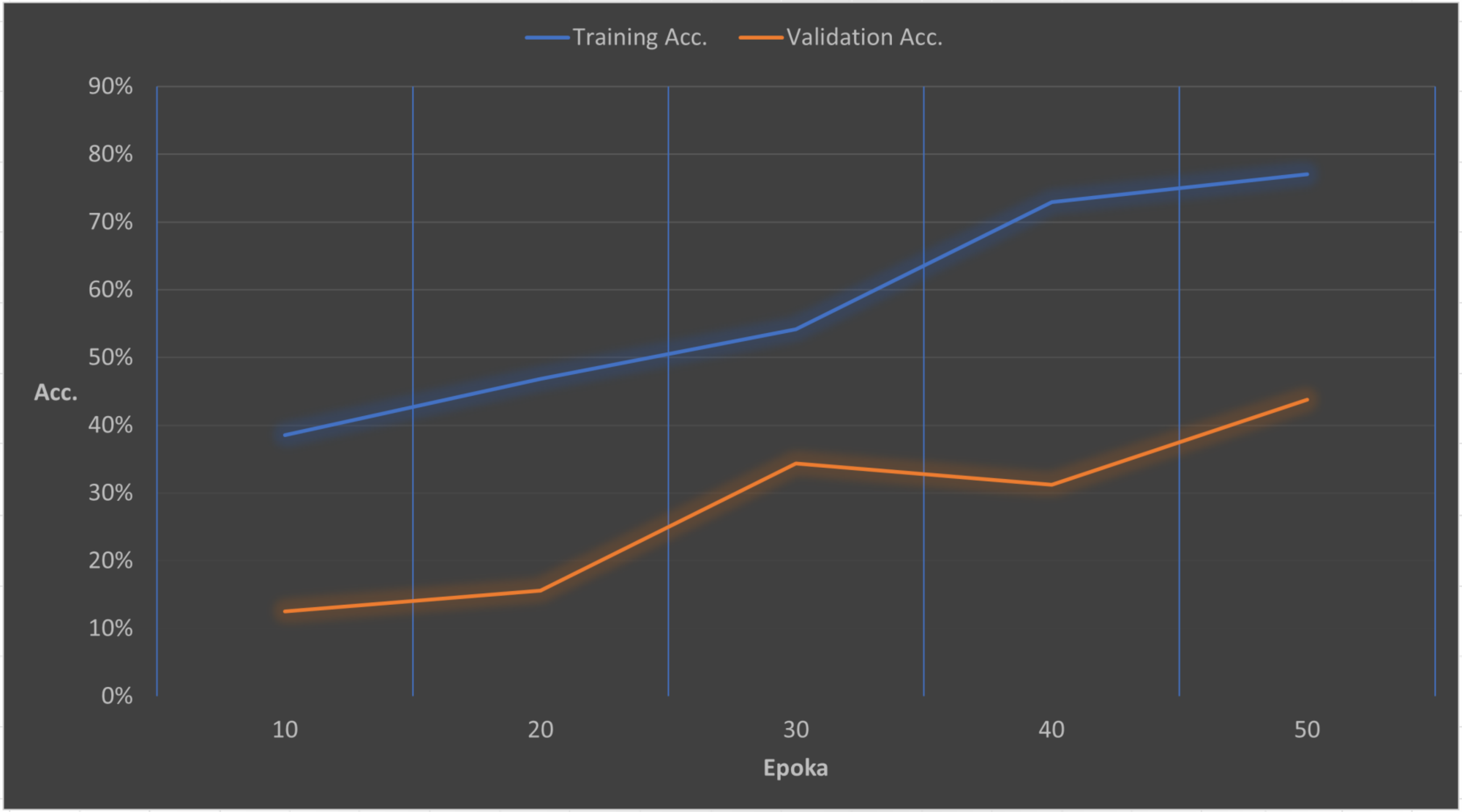
- Wyniki uzyskane dla: batch_size = 16, learning_rate = 0.0001 oraz liczby próbek = 32 i 50 epok.

Epoch	Training accuracy	Validation accuracy	Time [s]
10	38.54%	12.50%	35.64
20	46.87%	15.62%	38.20
30	54.16%	34.37%	35.84
40	72.91%	31.25%	36.33
50	**77.08%**	**43.75%**	36.88

- Przykładowe komunikaty w programie

```
Batch 91 of Epoch 10 done in 20.07.
Batch 92 of Epoch 10 done in 20.06.
Batch 93 of Epoch 10 done in 20.0.
Batch 94 of Epoch 10 done in 20.03.
Batch 95 of Epoch 10 done in 20.02.
Batch 96 of Epoch 10 done in 20.01.
Epoch 10:
Time: 2241.003 seconds
Train loss: inf
Train accuracy: 28.3366%

Validation loss: inf
Validation Accuracy: 26.416%

Batch 1 of Epoch 11 done in 20.04.
Batch 2 of Epoch 11 done in 20.03.
Batch 3 of Epoch 11 done in 20.21.
Batch 4 of Epoch 11 done in 23.65.
Batch 5 of Epoch 11 done in 22.37.
```

Działanie sieci – wyniki



Działanie sieci – wyniki

- W celu sprawdzenia działania zaimplementowanej sieci CNN spróbowano kilku wariantów parametrów: batch_size, learning_rate oraz liczby zadanych próbek.
- Wyniki uzyskane dla: batch_size = 32, learning_rate = 0.0001 oraz liczby próbek = 256 i 20 epok.

Epoch	Training accuracy	Validation accuracy	Time [s]
2	31.90%	22.26%	289.94
4	34.37%	26.95%	291.02
6	29.42%	25.39%	295.24
8	31.38%	27.34%	293.33
10	34.63%	27.34%	290.92
12	34.89%	27.34%	292.70
14	36.32%	30.85%	291.84
16	36.97%	**32.42%**	290.08
18	36.19%	28.90%	291.41
20	**38.41%**	29.29%	298.90

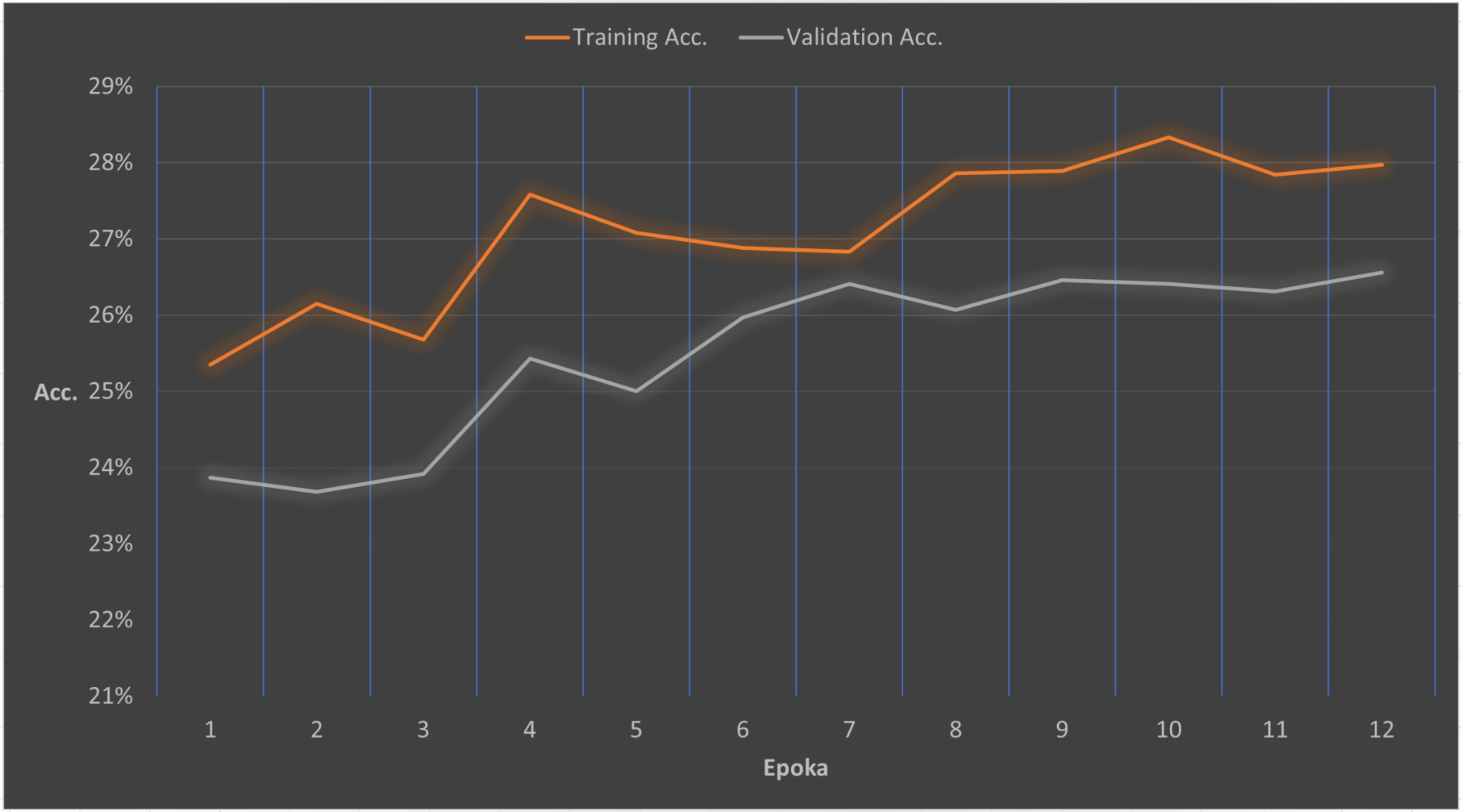
Działanie sieci – wyniki

Działanie sieci – wyniki

- Wyniki uzyskane dla: batch_size = 64, learning_rate = 0.0001 oraz liczby próbek = 2048 i 20 epok.

Epoch	Training accuracy	Validation accuracy	Time [s]
1	25.35%	23.87%	2211.21
2	26.15%	23.68%	2260.88
3	25.68%	23.92%	2264.81
4	27.58%	25.43%	2255.97
5	27.08%	25.00%	2241.18
6	26.88%	25.97%	2238.41
7	26.83%	26.41%	2243.93
8	27.86%	26.07%	2244.18
9	27.89%	26.46%	2244.42
10	28.33%	26.41%	2241.00
11	27.84%	26.31%	2249.68
12	27.97%	26.56%	2248.15

Działanie sieci – wyniki



Podsumowanie

- Sieć spełnia postawione założenia, jednak można powiedzieć, że jest mało efektywna – proces uczenia jest czasochłonny
- Dzięki zastosowaniu odpowiednich operacji na macierzach udało się przyspieszyć uczenie sieci
- Wyniki pokazują, że sieć jest w stanie się uczyć, co potwierdza poprawność implementacji
- Projekt i implementacja sieci neuronowej od zera jest dość trudnym zadaniem i zazwyczaj nie prowadzi do efektywniejszych rozwiązań, niż te powszechnie dostępne (PyTorch, TensorFlow)
- Szczególnie problematyczne okazało się zaimplementowanie propagacji wstecznej dla warstwy konwolucyjnej