simple-cnn

October 7, 2023

1 Tugas Besar IF4074 - Pembelajaran Mesin Lanjut

2 Implementasi Convolutional Neural Network

3 Simple CNN

Simple CNN is a convolutional neural network implemented in Python and fine-tuned using backpropagation algorithm.

3.1 Setup

Assuming you've installed the latest version of Python (if not, guides for it are widely available), 1. ensure pip is installed by running `python -m ensurepip --upgrade`; 2. install the Python dependencies by running `pip install -r requirements.txt`.

3.2 Contribution (Milestone 1)

NIM	Name	Contribution(s)
13520041	Ilham Pratama	Dataset handling; Detector, Pooling, Dense, and Flatten layer; Report
13520042	Jeremy S.O.N. Simbolon	Class model; Convolutional layer; Report

3.3 Contribution (Milestone 2)

NIM	Name	Contribution(s)
13520041	Ilham Pratama	Model training; Detector, Pooling, Dense, and Flatten layer; Report
13520042	Jeremy S.O.N. Simbolon	Model training; Model loading and saving; Convolutional layer; Report

3.3.1 Library Import

The following external library is used for the building of this model. 1. `cv2` for preprocessing the image dataset 2. `jsonpickle` for saving and loading the model 3. `numpy` for performing model-related calculations 4. `scipy` for performing a suppressed version of the logistic sigmoid function 5. `sklearn` for computing the model evaluation metrics

```
[1]: import math
     import os

     from typing import Any

     import cv2
     import jsonpickle
     import jsonpickle.ext.numpy
     import numpy as np
     import numpy.typing as npt

     from scipy.special import expit
     from sklearn import metrics
```

3.3.2 Dataset Loading

```
[2]: class Utils:
         """
         Module related utility functions.

         This class is used to prepare the image dataset for the CNN model. In
         addition, this class is also used to save and load the CNN model.
         """

         @staticmethod
         def load_dataset(dataset_path: str) -> tuple[npt.NDArray, npt.NDArray,␣
      ↪dict]:
             """
             Preprocess the dataset and return useful information for further␣
      ↪processing.

             :param dataset_path: A string representation of the path pointing to
                                  the dataset.
             :return: A tuple consisted of an ndarray of dataset image path, an
                      ndarray of image labels, and a dictionary that maps class
                      labels to folder name.
             """
             folder_list = sorted(os.listdir(dataset_path))
             image_path = []
             image_label = np.array([], dtype=np.int16)
             image_dictionary = {}
             for i, folder_name in enumerate(folder_list):
                 class_folder_path = os.path.join(dataset_path, folder_name)
                 list_image_name = sorted(os.listdir(class_folder_path))
                 temp_folder_path = [os.path.join(class_folder_path, image_name) for␣
      ↪image_name in list_image_name]
```

```
            image_path += temp_folder_path
            temp_class_label = np.full(len(list_image_name), i, dtype=np.int16)
            image_label = np.concatenate((image_label, temp_class_label),␣
↪axis=0)
            image_dictionary[str(i)] = folder_name

        return np.asarray(image_path), image_label, image_dictionary

    @staticmethod
    def convert_image_to_matrix(path: npt.NDArray) -> npt.NDArray:
        """
        Convert the image dataset into a list of ndarray.

        Each ndarray is an RGB representation of each image in the dataset.

        :param path: An ndarray of string representation of the path pointing
                     to each image entry in the dataset.
        :return: A list of ndarray representation of the image in the dataset.
        """
        list_of_image_matrix = []
        size = (256, 256)

        for file_img in path:
            image = cv2.imread(file_img, 1)
            matrix = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
            matrix = cv2.resize(matrix, size)
            list_of_image_matrix.append(matrix)

        return np.array(list_of_image_matrix)

    @staticmethod
    def save_model(model_object: "Model", file_name: str = "model.json") ->␣
↪None:
        """
        Save the specified model into a JSON file.

        :param model_object: The model to be saved.
        :param file_name: A string specifying the file name of the saved model.
        """
        jsonpickle.ext.numpy.register_handlers()
        with open(file_name, "w") as file:
            json = jsonpickle.encode(model_object, indent=4)
            file.write(json)

    @staticmethod
    def load_model(file_name: str = "model.json") -> "Model":
        """
```

```
        Load a model from the specified JSON file name.

        :param file_name: A string specifying the file name of the model to be
                          loaded.
        :return: The loaded model from the specified file.
        """
        jsonpickle.ext.numpy.register_handlers()
        with open(file_name, "r") as file:
            json = file.read()
            return jsonpickle.decode(json)
```

3.3.3 Model Representation

The convolutional model is represented by a class named `Model`. The `Model` class contains several inner class that represents all possible layers that the model can have. Such layers include the convolution layer (represented by the `ConvolutionLayer` class), the detector layer (represented by the `DetectorLayer` class), the pooling layer (represented by the `PoolingLayer` class), the dense layer (represented by the `DenseLayer` class), and the flatten layer (represented by the `FlattenLayer` class).

[3]:
```
class Model:
    """
    The convolutional neural network model used to classify images.
    """

    def __init__(self) -> None:
        """
        Instantiate the convolutional neural network model.
        """
        self._layers = []
        self._forward_result = None
        self._backward_result = None

    class Layer:
        """
        Base representation of the layer used as part of the convolutional
        neural network architecture.
        """

        def __init__(self, name) -> None:
            """
            Instantiate the base layer.

            :param name: Name of the layer.
            """
            self._name = name
```

```python
    def forward_propagate(self) -> None:
        """Indicate the forward propagation is being performed."""
        print(f"Performing forward propagation on {self._name} layer...")
        print()

    def backward_propagate(self) -> None:
        """Indicate the backward propagation is being performed."""
        print(f"Performing backward propagation on {self._name} layer...")
        print()

class ConvolutionLayer(Layer):
    """
    The convolutional layer in convolutional neural network.

    This class is inherited from the ``Layer`` class. This layer is used
    to perform the convolution operation on the input weights.
    """

    def __init__(
        self,
        filter_count: int,
        filter_size: tuple[int, int] = (32, 32),
        padding_size: int = 0,
        stride_size: tuple[int, int] = (1, 1),
    ) -> None:
        """
        Instantiate the convolutional layer.

        :param filter_count: An integer specifying the amount of feature
                             to be extracted in the form of the amount of
                             filters.
        :param filter_size: A tuple of two integers specifying the height
                            and width of the convolution filter.
        :param padding_size: An integer specifying the dimension of 0's to
                             be added around the weight.
        :param stride_size: A tuple of two integers specifying the pixel
                            step size along the height and width of the
                            input weight.
        """
        super().__init__("convolution")
        self._filter_count = filter_count
        self._filter_dimension = 0
        self._filter_height, self._filter_width = filter_size
        self._filter_weights = None
        self._padding_size = padding_size
        self._stride_height, self._stride_width = stride_size
        self._output_height = 0
```

```python
        self._output_width = 0
        self._weight_dimension = 0
        self._weight_height = 0
        self._weight_width = 0
        self._weights = None
        self._biases = None
        self._filter_gradients = []
        self._bias_gradients = []

    def _pad_weights(
        self, weights: npt.NDArray[npt.NDArray[npt.NDArray[float]]], 
↪padding_size: int, forward: bool = True
    ) -> npt.NDArray[npt.NDArray[npt.NDArray[float]]]:
        """
        Pad the specified weights with 0's around it.

        :param weights: The ndarray of weights to be padded with 0's.
        :param padding_size: An integer specifying the dimension of 0's to
                             be added around the weight.
        :param forward: A boolean specifying whether the padding is
                        performed during forward propagation.
        :return: An ndarray of weights padded with 0's.
        """
        weight_dimension = len(weights)
        weight_height = len(weights[0])
        weight_width = len(weights[0][0])

        if forward:
            self._weight_dimension = weight_dimension
            self._weight_height = weight_height + 2 * padding_size
            self._weight_width = weight_width + 2 * padding_size

        padded_weights = [
            [
                [
                    0.0
                    if k < padding_size
                    or k >= weight_width + padding_size
                    or j < padding_size
                    or j >= weight_height + padding_size
                    else weights[i][j - padding_size][k - padding_size]
                    for k in range(weight_width + 2 * padding_size)
                ]
                for j in range(weight_height + 2 * padding_size)
            ]
            for i in range(weight_dimension)
        ]
```

```python
        return np.array(padded_weights)

    def _convolute(
        self,
        weights: npt.NDArray[npt.NDArray[npt.NDArray[float]]],
    ) -> npt.NDArray[npt.NDArray[npt.NDArray[float]]]:
        """
        Perform the convolution operation on the input weights.

        :param weights: An ndarray of input weights.
        :return: An ndarray of features extracted from the weights.
        """
        self._weights = np.array(weights)
        self._filter_dimension = len(weights)
        self._output_height = (
            math.ceil((len(weights[0]) - self._filter_height + 2 * self.
↪_padding_size) / self._stride_height) + 1
        )
        self._output_width = (
            math.ceil((len(weights[0][0]) - self._filter_width + 2 * self.
↪_padding_size) / self._stride_width) + 1
        )

        if self._filter_weights is None:
            self._filter_weights = np.random.rand(
                self._filter_count,
                self._filter_dimension,
                self._filter_height,
                self._filter_width,
            )
        if self._biases is None:
            self._biases = np.random.rand(self._filter_count, self.
↪_output_height, self._output_width)

        feature_maps = np.copy(self._biases)
        weights = self._pad_weights(weights, self._padding_size)
        for i in range(self._filter_count):
            for j in range(0, self._weight_height - self._filter_height +␣
↪1, self._stride_height):
                for k in range(0, self._weight_width - self._filter_width +␣
↪1, self._stride_width):
                    for l in range(self._filter_dimension):
                        field = weights[l, j : j + self._filter_height, k :␣
↪k + self._filter_width]
                        feature = field * self._filter_weights[i][l]
```

```python
                            feature_maps[i][j][k] += np.sum(feature)
        return feature_maps

    def _calculate_gradient(self, output_gradient: npt.NDArray) -> npt.
↪NDArray:
        """
        Calculate the gradient used for updating the weight of the
        convolution layer.

        :param output_gradient: The gradient of the model's output with
                                respect to the layer ahead.
        :return: The gradient of the model's output with respect to this
                 convolutional layer.
        """
        output_gradient_height = len(output_gradient[0])
        output_gradient_width = len(output_gradient[0][0])

        filter_gradient = np.zeros(
            (self._filter_count, self._filter_dimension, self.
↪_filter_height, self._filter_width)
        )
        input_gradient = np.zeros((self._weight_dimension, self.
↪_weight_height, self._weight_width))
        padded_output_gradient = self._pad_weights(output_gradient, 2,␣
↪forward=False)

        for i in range(self._filter_count):
            for j in range(self._filter_dimension):
                for k in range(0, self._weight_height -␣
↪output_gradient_height + 1, self._stride_height):
                    for l in range(0, self._weight_width -␣
↪output_gradient_width + 1, self._stride_width):
                        field = self._weights[j, k : k +␣
↪output_gradient_height, l : l + output_gradient_width]
                        gradient = field * output_gradient[i]
                        filter_gradient[i][j][k][l] = np.sum(gradient)
                for k in range(0, output_gradient_height - self.
↪_filter_height + 1, self._stride_height):
                    for l in range(0, output_gradient_width - self.
↪_filter_width + 1, self._stride_width):
                        field = padded_output_gradient[i, k : k + self.
↪_filter_height, l : l + self._filter_width]
                        gradient = field * np.rot90(self.
↪_filter_weights[i][j], k=2)
                        input_gradient[j][k][l] += np.sum(gradient)
```

```
            self._filter_gradients.append(filter_gradient)
            self._bias_gradients.append(output_gradient)

            return input_gradient

        def forward_propagate(
            self, weights: npt.NDArray[npt.NDArray[npt.NDArray[float]]]
        ) -> npt.NDArray[npt.NDArray[npt.NDArray[float]]]:
            """
            Indicate and perform the convolution process on the input weights.

            :param weights: The ndarray of weights to be convoluted.
            :return: An ndarray of convoluted weights.
            """
            super().forward_propagate()
            result = self._convolute(weights)
            return result

        def backward_propagate(self, gradient) -> npt.NDArray:
            """
            Indicate and perform the backward propagation operation on the␣
↪model.
            """
            super().backward_propagate()
            output_gradient = self._calculate_gradient(gradient)
            return output_gradient

        def update_weight(self, learning_rate: float) -> None:
            """
            Update the filter weights of the convolution layer.

            :param learning_rate: A float specifying the learning rate of the␣
↪model.
            """
            self._filter_weights -= learning_rate * np.average(np.array(self.
↪_filter_gradients), axis=0)
            self._biases -= learning_rate * np.average(np.array(self.
↪_bias_gradients), axis=0)
            self._filter_gradients = []
            self._bias_gradients = []

    class DetectorLayer(Layer):
        """
        The detector layer in convolutional neural network.

        This class is inherited from the ``Layer`` class. This layer is used to
        introduce non-linearity to the learning process using the reLU
```

```
        activation function.
        """

    def __init__(self) -> None:
        """Instantiate the detector layer."""
        super().__init__("detector")
        self._weights = None

    def _detect(self, feature: npt.NDArray) -> npt.NDArray:
        """
        Apply the reLU activation function on the input weights.

        :param feature: An ndarray of input weights.
        :return: An ndarray of weights on which the reLU function has been
                 applied.
        """
        self._weights = feature
        return np.maximum(feature, 0)

    def _calculate_gradient(self, error: npt.NDArray) -> npt.NDArray:
        """
        Perform the backward propagation on the detector layer.
        Use reLu derivative: dreLU/dx = 1 if x > 0, otherwise 0.

        :param error: The gradient from the next layer.
        :return: The gradient for the previous layer.
        """
        dx = error * (self._weights > 0)
        return dx

    def forward_propagate(self, feature: npt.NDArray) -> npt.NDArray:
        """
        Indicate and perform the detector process on the input weights.

        :param feature: The ndarray of weights on which reLU function is
                        to be applied.
        :return: An ndarray of activated weights.
        """
        super().forward_propagate()
        result = self._detect(feature)
        return result

    def backward_propagate(self, gradient) -> npt.NDArray:
        """
        Indicate and perform the backward propagation operation on the␣
↪model.
        """
```

```
        super().backward_propagate()
        output_gradient = self._calculate_gradient(gradient)
        return output_gradient

    def update_weight(self, learning_rate: float) -> None:
        """
        Update the filter weights of the detector layer.

        Because the detector layer has no trainable weights, this method
        exist for the purpose of iteratively updating the weights of all
        the layers in the model. In practice, this method does nothing.

        :param learning_rate: A float specifying the learning rate of the␣
↪model.
        """
        pass

class PoolingLayer(Layer):
    """
    The pooling layer in convolutional neural network.

    This class is inherited from the ``Layer`` class. This layer is used to
    down-sample the input weights according to the specified pooling
    operation.
    """

    def __init__(self, filter_size: int, stride_size: int, mode: str =␣
↪"max") -> None:
        """
        Instantiate the pooling layer.

        :param filter_size: An integer specifying the dimension of the
                            pooling window.
        :param stride_size: An integer specifying the pixel step size along
                            the height and width of the input weight.
        :param mode: A string specifying the preferred pooling operation.
                     Must either be ``average`` or ``max``.
        """
        super().__init__("pooling")
        self._filter_size = filter_size
        self._stride_size = stride_size
        self._mode = mode
        self._weights = None

    def _average(self, input_matrix: npt.NDArray, d: int, h: int, w: int)␣
↪-> float:
        """
```

```
        Take the average of the input values over the pooling window.

        :param input_matrix: The ndarray of weights on which the operation
                             is applied.
        :param d: An integer specifying the depth location of the pooling
                  window.
        :param h: An integer specifying the height location of the pooling
                  window.
        :param w: An integer specifying the width location of the pooling
                  window.
        :return: The average of the input values.
        """
        h_start = h * self._stride_size
        w_start = w * self._stride_size
        h_end = h_start + self._filter_size
        w_end = w_start + self._filter_size
        return np.average(input_matrix[d, h_start:h_end, w_start:w_end])

    def _max(self, input_matrix: npt.NDArray, d: int, h: int, w: int) ->␣
↪float:
        """
        Take the maximum of the input values over the pooling window.

        :param input_matrix: The ndarray of weights on which the operation
                             is applied.
        :param d: An integer specifying the depth location of the pooling
                  window.
        :param h: An integer specifying the height location of the pooling
                  window.
        :param w: An integer specifying the width location of the pooling
                  window.
        :return: The maximum of the input values.
        """
        h_start = h * self._stride_size
        w_start = w * self._stride_size
        h_end = h_start + self._filter_size
        w_end = w_start + self._filter_size
        return np.max(input_matrix[d, h_start:h_end, w_start:w_end])

    def _pool(self, input_matrix: npt.NDArray) -> npt.NDArray:
        """
        Perform the pooling operation on the input weights.

        :param input_matrix: An ndarray of input weights.
        :return: An ndarray of down-sampled weights.
        """
        self._weights = input_matrix
```

```
        depth, height, width = input_matrix.shape
        filter_height = (height - self._filter_size) // self._stride_size +␣
↪1
        filter_width = (width - self._filter_size) // self._stride_size + 1
        pooled = np.zeros([depth, filter_height, filter_width], dtype=np.
↪double)
        for d in range(0, depth):
            for h in range(0, filter_height):
                for w in range(0, filter_width):
                    if self._mode == "average":
                        pooled[d, h, w] = self._average(input_matrix, d, h,␣
↪w)
                    elif self._mode == "max":
                        pooled[d, h, w] = self._max(input_matrix, d, h, w)
        return pooled

    def _calculate_gradient(self, error: npt.NDArray) -> npt.NDArray:
        f, w, h = self._weights.shape
        dx = np.zeros(self._weights.shape)
        for i in range(0, f):
            for j in range(0, w, self._filter_size):
                for k in range(0, h, self._filter_size):
                    input_slice = self._weights[i, j : j + self.
↪_filter_size, k : k + self._filter_size]
                    max_input_slice = np.argmax(input_slice)
                    max_idx = np.unravel_index(max_input_slice, (self.
↪_filter_size, self._filter_size))
                    if (j + max_idx[0]) < w and (k + max_idx[1]) < h:
                        dx[i, j + max_idx[0], k + max_idx[1]] = error[
                            i, int(j // self._filter_size), int(k // self.
↪_filter_size)
                        ]
        return dx

    def forward_propagate(self, input_matrix: npt.NDArray) -> npt.NDArray:
        """
        Indicate and perform the pooling operation on the input weights.

        :param input_matrix: An ndarray of input weights.
        :return: An ndarray of down-sampled weights.
        """
        super().forward_propagate()
        result = self._pool(input_matrix)
        return result

    def backward_propagate(self, gradient) -> npt.NDArray:
```

```
        """
        Indicate and perform the backward propagation operation on the␣
↪model.
        """
        super().backward_propagate()
        output_gradient = self._calculate_gradient(gradient)
        return output_gradient

    def update_weight(self, learning_rate: float) -> None:
        """
        Update the filter weights of the pooling layer.

        Because the pooling layer has no trainable weights, this method
        exist for the purpose of iteratively updating the weights of all
        the layers in the model. In practice, this method does nothing.

        :param learning_rate: A float specifying the learning rate of the␣
↪model.
        """
        pass

class DenseLayer(Layer):
    """
    The dense layer in convolutional neural network.

    This class is inherited from the ``Layer`` class. This layer is used to
    abstractly represent the input data using its weights.
    """

    def __init__(self, unit_count: int, activation: str = "sigmoid") ->␣
↪None:
        """
        Instantiate the dense layer.

        :param unit_count: An integer specifying the dimension of the
                           output space.
        :param activation: The activation function to be applied to each
                           node. Must either be ``sigmoid`` or ``relu``.
        """
        super().__init__("dense")
        self._unit_count = unit_count
        self._activation = activation
        self._bias = np.zeros(unit_count)
        self._dense_weight = []
        self._weights = None
        self._output = 0.0
        self._deltaW = np.zeros(unit_count)
```

```
    @staticmethod
    def _sigmoid_derivative(input_: npt.NDArray) -> npt.NDArray:
        """
        Take derivative value from input.
        """
        sigmoid = 1 / (1 + np.exp(-input_))
        return sigmoid * (1 - sigmoid)

    @staticmethod
    def _relu_derivative(input_: npt.NDArray) -> int:
        """
        Take derivative value from input.
        """
        if input_ >= 0:
            return 1
        else:
            return 0

    def _derivative_act_func(self, activation: str, input_: npt.NDArray) ->␣
↪npt.NDArray | float:
        """
        Take derivative value from activation function and input.
        """
        if activation == "sigmoid":
            return self._sigmoid_derivative(input_)
        else:
            return self._relu_derivative(input_)

    def _dense(self, input_matrix: npt.NDArray) -> npt.NDArray:
        """
        Perform the linear combination and activation of the input weights
        using the layer's weights.

        :param input_matrix: An ndarray of input weights.
        :return: An ndarray of linearly-combined and activated weights.
        """
        self._weights = input_matrix
        if len(self._dense_weight) == 0:
            self._dense_weight = np.random.randn(self._unit_count, len(self.
↪_weights))
        result = np.zeros(self._unit_count)

        for i in range(self._unit_count):
            input_weight = np.sum(self._dense_weight[i] * input_matrix)
            result[i] = input_weight + self._bias[i]
```

```python
        if self._activation == "sigmoid":
            self.output = expit(result)
        elif self._activation == "relu":
            self.output = np.maximum(result, 0)
        return self.output

    def _calculate_gradient(self, error: npt.NDArray) -> npt.NDArray:
        """
        Perform the backward propagation on the layer.

        :param error: The gradient from the next layer.
        :return: The gradient for the previous layer.
        """
        derivative_value = np.array([])
        for i in self.output:
            derivative_value = np.append(derivative_value, self.
↪_derivative_act_func(self._activation, i))

        self._deltaW += np.multiply(derivative_value, error)
        de = np.matmul(error, self._dense_weight)
        return de

    def forward_propagate(self, input_matrix: npt.NDArray) -> npt.NDArray:
        """
        Indicate and perform the linear combination and activation of the
        input weights using the layer's weights.

        :param input_matrix: An ndarray of input weights.
        :return: An ndarray of linearly-combined and activated weights.
        """
        super().forward_propagate()
        result = self._dense(input_matrix)
        return result

    def backward_propagate(self, gradient: npt.NDArray) -> npt.NDArray:
        """
        Indicate and perform the backward propagation operation on the␣
↪model.
        """
        super().backward_propagate()
        output_gradient = self._calculate_gradient(gradient)
        return output_gradient

    def update_weight(self, learning_rate: float) -> None:
        """
        Indicate and perform the update weight and bias on the model.
        """
```

```
            for i in range(self._unit_count):
                self._dense_weight[i] -= learning_rate * self._deltaW[i] * self.
↪_weights

            self._bias -= learning_rate * self._deltaW
            self._deltaW = np.zeros(self._unit_count)

    class FlattenLayer(Layer):
        """
        The flatten layer in convolutional neural network.

        This class is inherited from the ``Layer`` class. This layer is used to
        flatten the input weights.
        """

        def __init__(self) -> None:
            """Instantiate the flatten layer."""
            super().__init__("flatten")
            self._weights = None

        def _flatten(self, input_matrix: npt.NDArray) -> npt.NDArray:
            """
            Perform the flatten operation on the input weights.

            :param input_matrix: An ndarray of input weights.
            :return: An ndarray of flatten weights.
            """
            self._weights = input_matrix
            return input_matrix.flatten()

        def forward_propagate(self, input_matrix: npt.NDArray) -> npt.NDArray:
            """
            Indicate and perform the flatten operation on the input weights.

            :param input_matrix: An ndarray of input weights.
            :return: An ndarray of flatten weights.
            """
            super().forward_propagate()
            result = self._flatten(input_matrix)
            return result

        def backward_propagate(self, gradient: npt.NDArray) -> npt.NDArray:
            """
            Indicate and perform the backward propagation operation on the␣
↪model.
            """
            super().backward_propagate()
```

```python
            k, w, h = self._weights.shape
            return gradient.reshape(k, w, h)

        def update_weight(self, learning_rate: float) -> None:
            """
            Update the filter weights of the flatten layer.

            Because the flatten layer has no trainable weights, this method
            exist for the purpose of iteratively updating the weights of all
            the layers in the model. In practice, this method does nothing.

            :param learning_rate: A float specifying the learning rate of the␣
↪model.
            """
            pass

    def add_layer(self, name: str, **kwargs: Any) -> None:
        """
        Sequentially add the specified layer into the model.

        :param name: A string representation of the layer to be added.
        :param kwargs: Layer-related parameters in the form of key-value pairs.
        """
        match name:
            case "convolution":
                self._layers.append(self.ConvolutionLayer(**kwargs))
            case "detector":
                self._layers.append(self.DetectorLayer())
            case "pooling":
                self._layers.append(self.PoolingLayer(**kwargs))
            case "dense":
                self._layers.append(self.DenseLayer(**kwargs))
            case "flatten":
                self._layers.append(self.FlattenLayer())

    def forward_propagate(self, tensor: npt.NDArray) -> None:
        """
        Indicate and perform the forward propagation operation on the model.

        :param tensor: An ndarray of input weights representing the input
                       pictures.
        """
        for layer in self._layers:
            tensor = layer.forward_propagate(tensor)
        self._forward_result = tensor

    def backward_propagate(self, gradient: npt.NDArray) -> None:
```

```python
        """
        Indicate and perform the backward propagation operation on the model.
        """
        for layer in reversed(self._layers):
            gradient = layer.backward_propagate(gradient)
        self._backward_result = gradient

    def train(
        self,
        tensor: npt.NDArray[npt.NDArray],
        target: npt.NDArray,
        epochs: int = 1,
        batch_size: int = 5,
        learning_rate: float = 0.01,
    ) -> None:
        """
        Fit and train the CNN model.

        :param tensor: An ndarray of representations of the input pictures to
                       be fed into the model.
        :param target: An ndarray of representations of the target pictures to
                       be fed into the model.
        :param epochs: An integer specifying the number of training epochs.
        :param batch_size: An integer specifying the number of training batch.
        :param learning_rate: A float specifying the learning rate of the model.
        """
        out = np.array([])
        y_target = np.array([])
        for epoch in range(epochs):
            loss = 0
            print("Epoch : ", epoch + 1)
            for i in range(len(tensor)):
                self.forward_propagate(tensor[i])
                forward_result = self._forward_result
                curr_target = target[i]
                curr_output = forward_result[0]
                de = np.array([curr_target - curr_output]) * -1
                self.backward_propagate(de)
                loss += 0.5 * (curr_target - curr_output) ** 2
                out = np.rint(np.append(out, curr_output))
                y_target = np.append(y_target, curr_target)

                if (i + 1) % batch_size == 0:
                    for layer in reversed(self._layers):
                        layer.update_weight(learning_rate)

            avg_loss = loss / len(tensor)
```

```
            print("Loss: ", avg_loss)
            print("Accuracy: ", metrics.accuracy_score(y_target, out))
```

3.3.4 Test result

We shall test the model we have built above using a subset of the dataset provided.

[4]:
```
folder_path, class_label, class_dictionary = Utils.load_dataset("./dataset")
image_matrix = Utils.convert_image_to_matrix(folder_path).reshape((100, 1, 256,␣
 ↪256))[:10]

model = Model()
model.add_layer(
    "convolution",
    filter_count=32,
    filter_size=(3, 3),
    padding_size=0,
    stride_size=(1, 1),
)
model.add_layer("detector")
model.add_layer("pooling", filter_size=3, stride_size=2, mode="average")
model.add_layer("flatten")
model.add_layer("dense", unit_count=8, activation="relu")
model.add_layer("dense", unit_count=1, activation="sigmoid")
model.train(image_matrix, class_label)
```

```
Epoch :  1
Performing forward propagation on convolution layer…
Performing forward propagation on detector layer…

Performing forward propagation on pooling layer…
Performing forward propagation on flatten layer…

Performing forward propagation on dense layer…

Performing forward propagation on dense layer…

Performing backward propagation on dense layer…

Performing backward propagation on dense layer…

Performing backward propagation on flatten layer…

Performing backward propagation on pooling layer…
Performing backward propagation on detector layer…

Performing backward propagation on convolution layer…
Performing forward propagation on convolution layer…
```

```
Performing forward propagation on detector layer…

Performing forward propagation on pooling layer…
Performing forward propagation on flatten layer…

Performing forward propagation on dense layer…

Performing forward propagation on dense layer…

Performing backward propagation on dense layer…

Performing backward propagation on dense layer…

Performing backward propagation on flatten layer…

Performing backward propagation on pooling layer…
Performing backward propagation on detector layer…

Performing backward propagation on convolution layer…
Performing forward propagation on convolution layer…
Performing forward propagation on detector layer…

Performing forward propagation on pooling layer…
Performing forward propagation on flatten layer…

Performing forward propagation on dense layer…

Performing forward propagation on dense layer…

Performing backward propagation on dense layer…

Performing backward propagation on dense layer…

Performing backward propagation on flatten layer…

Performing backward propagation on pooling layer…
Performing backward propagation on detector layer…

Performing backward propagation on convolution layer…
Performing forward propagation on convolution layer…
Performing forward propagation on detector layer…

Performing forward propagation on pooling layer…
Performing forward propagation on flatten layer…

Performing forward propagation on dense layer…

Performing forward propagation on dense layer…
```

```
Performing backward propagation on dense layer…

Performing backward propagation on dense layer…

Performing backward propagation on flatten layer…

Performing backward propagation on pooling layer…
Performing backward propagation on detector layer…

Performing backward propagation on convolution layer…
Performing forward propagation on convolution layer…
Performing forward propagation on detector layer…

Performing forward propagation on pooling layer…
Performing forward propagation on flatten layer…

Performing forward propagation on dense layer…

Performing forward propagation on dense layer…

Performing backward propagation on dense layer…

Performing backward propagation on dense layer…

Performing backward propagation on flatten layer…

Performing backward propagation on pooling layer…
Performing backward propagation on detector layer…

Performing backward propagation on convolution layer…
Performing forward propagation on convolution layer…
Performing forward propagation on detector layer…

Performing forward propagation on pooling layer…
Performing forward propagation on flatten layer…

Performing forward propagation on dense layer…

Performing forward propagation on dense layer…

Performing backward propagation on dense layer…

Performing backward propagation on dense layer…

Performing backward propagation on flatten layer…

Performing backward propagation on pooling layer…
```

```
Performing backward propagation on detector layer…

Performing backward propagation on convolution layer…
Performing forward propagation on convolution layer…
Performing forward propagation on detector layer…

Performing forward propagation on pooling layer…
Performing forward propagation on flatten layer…

Performing forward propagation on dense layer…

Performing forward propagation on dense layer…

Performing backward propagation on dense layer…

Performing backward propagation on dense layer…

Performing backward propagation on flatten layer…

Performing backward propagation on pooling layer…
Performing backward propagation on detector layer…

Performing backward propagation on convolution layer…
Performing forward propagation on convolution layer…
Performing forward propagation on detector layer…

Performing forward propagation on pooling layer…
Performing forward propagation on flatten layer…

Performing forward propagation on dense layer…

Performing forward propagation on dense layer…

Performing backward propagation on dense layer…

Performing backward propagation on dense layer…

Performing backward propagation on flatten layer…

Performing backward propagation on pooling layer…
Performing backward propagation on detector layer…

Performing backward propagation on convolution layer…
Performing forward propagation on convolution layer…
Performing forward propagation on detector layer…

Performing forward propagation on pooling layer…
Performing forward propagation on flatten layer…
```

```
Performing forward propagation on dense layer…

Performing forward propagation on dense layer…

Performing backward propagation on dense layer…

Performing backward propagation on dense layer…

Performing backward propagation on flatten layer…

Performing backward propagation on pooling layer…
Performing backward propagation on detector layer…

Performing backward propagation on convolution layer…
Performing forward propagation on convolution layer…
Performing forward propagation on detector layer…

Performing forward propagation on pooling layer…
Performing forward propagation on flatten layer…

Performing forward propagation on dense layer…

Performing forward propagation on dense layer…

Performing backward propagation on dense layer…

Performing backward propagation on dense layer…

Performing backward propagation on flatten layer…

Performing backward propagation on pooling layer…
Performing backward propagation on detector layer…

Performing backward propagation on convolution layer…
Loss:  0.15
Accuracy:  0.7
```

3.3.5 Model Saving and Loading

[5]:
```
Utils.save_model(model, file_name="model.json")
loaded_model = Utils.load_model("model.json")
```