```
Test Preview: Summary for  of c3
--------------------------------

  Public Tests:
    Part 1: Linear layer:          3 / 3
    Part 1: Network:               2 / 2
    Part 1: Trainer:               3 / 3
    Part 1: Preprocessor:          4 / 4
    Part 1: Activation functions:  4 / 4
    Part 2: Regressor:             0 / 1
    Part 2: Preprocessing:         1 / 1
    Part 2: Performance:           1 / 1

Git Repo: git@gitlab.doc.ic.ac.uk:lab2223_autumn/Neural_Networks_078.git
Commit ID: f1b9e
```

```python
import math
from sklearn.metrics import mean_squared_error
import torch.nn as nn
import torch
import pickle
import numpy as np
import pandas as pd

from pandas import DataFrame, concat
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder, PowerTransformer
from torch import Tensor, tensor, float32
from typing import Callable, Union, List

def as_torch_tensor(func: Callable[..., DataFrame]):
    def wrapper(*args, **kwargs):
        df: DataFrame = func(*args, **kwargs)
        return tensor(df, dtype=float32)
    return wrapper

class Preprocessor():
    __input_transformer: ColumnTransformer
    __target_transformer: ColumnTransformer

    def __init__(self):
        self.__input_transformer = ColumnTransformer(
            transformers=[
                ('std', StandardScaler(), [
                    'longitude',
                    'latitude',
                ]),
                ('pwr', PowerTransformer(method="box-cox", standardize=True), [
                    'housing_median_age',
                    'total_rooms',
                    'total_bedrooms',
                    'population',
                    'households',
                    'median_income',
                ]),
                ('one_hot', OneHotEncoder(), ['ocean_proximity']),
            ],
            remainder='drop',
        )
        self.__target_transformer = ColumnTransformer(
            transformers=[
                ('std', StandardScaler(), ['median_house_value'])
            ],
            remainder='drop',
        )

    def fill_missing(self, x: DataFrame):
        x = x.copy()

        total_bedrooms_mean = x['total_bedrooms'].mean()
        x['total_bedrooms'] = x['total_bedrooms'].fillna(total_bedrooms_mean)

        return x

    def fit_input(self, input_df: DataFrame):
        self.__input_transformer.fit(input_df)

    def fit_target(self, target_df: DataFrame):
        self.__target_transformer.fit(target_df)

    @as_torch_tensor
    def transform_input(self, input_df: DataFrame) -> Tensor:
```



```python
        return self.__input_transformer.transform(input_df)

    @as_torch_tensor
    def transform_target(self, target_df: DataFrame) -> Tensor:
        return self.__target_transformer.transform(target_df)

    def inverse_transform_target(self, target_tensor: Tensor):
        target_np = target_tensor.detach().numpy()

        return self.__target_transformer \
            .named_transformers_['std'] \
            .inverse_transform(target_np)


class Regressor():
    __preprocessor: Preprocessor
    __loss_history: List[float]
    __loss_history_eval: List[float]

    def __init__(self, x, nb_epoch=1000, model=None, batch_size=32, learning_rate=0.01, loss_fn=nn.MSELoss()):
        # You can add any input parameters you need
        # Remember to set them with a default value for LabTS tests
        """
        Initialise the model.

        Arguments:
            - x {pd.DataFrame} -- Raw input data of shape
                (batch_size, input_size), used to compute the size
                of the network.
            - nb_epoch {int} -- number of epochs to train the network.

        """

        self.nb_epoch = nb_epoch
        self.batch_size = batch_size
        self.learning_rate = learning_rate
        self.loss_fn = loss_fn

        # Initialise Loss History
        self.__loss_history = []
        self.__loss_history_eval = []

        # Construct Preprocessor
        self.__preprocessor = Preprocessor()

        # Initialise Preprocessor
        X, _ = self._preprocessor(x, training=True)

        self.input_size = X.shape[1]
        self.output_size = 1

        # default configuration
        if model is None:
            self.model = nn.Sequential(
                nn.Linear(self.input_size, 64),
                nn.SiLU(),
                nn.Linear(64, 64),
                nn.SiLU(),
                nn.Linear(64, self.output_size),
            )
        else:
            self.model = model

    def _preprocessor(self, x, y=None, training=False):
        """
```

```
        Preprocess input of the network.

        Arguments:
            - x {pd.DataFrame} -- Raw input array of shape
                (batch_size, input_size).
            - y {pd.DataFrame} -- Raw target array of shape (batch_size, 1).
            - training {boolean} -- Boolean indicating if we are training or
                testing the model.

        Returns:
            - {torch.tensor} or {numpy.ndarray} -- Preprocessed input array of
              size (batch_size, input_size). The input_size does not have to be the same as the input_size for x above.
            - {torch.tensor} or {numpy.ndarray} -- Preprocessed target array of
              size (batch_size, 1).

        """

        # clean data
        x = self.__preprocessor.fill_missing(x)


        # If training flag set, fit preprocessor to training data.
        if training:
            # Remove rows who’s age are at maximum threshold.
            # age_filter = x[’housing_median_age’] <= 50
            # x = x[age_filter]
            # if y is not None: y = y[age_filter]

            # Remove rows who’s price is at maximum threshold.
            # if y is not None:
            #     price_filter = y[’median_house_value’] <= 500000
            #     x = x[price_filter]
            #     y = y[price_filter]

            self.__preprocessor.fit_input(x)
            if y is not None: self.__preprocessor.fit_target(y)

        return (
            self.__preprocessor.transform_input(x),
            self.__preprocessor.transform_target(y) if y is not None else None
        )

    def fit(self, x, y, x_eval = None, y_eval = None):
        """
        Regressor training function

        Arguments:
            - x {pd.DataFrame} -- Raw input array of shape
                (batch_size, input_size).
            - y {pd.DataFrame} -- Raw output array of shape (batch_size, 1).

        Returns:
            self {Regressor} -- Trained model.

        """

        print("A")
        input_data, target_data = self._preprocessor(
            x, y=y, training=True)  # Do not forget

        print("A")
        for epoch_n in range(self.nb_epoch):
            # Adam optimiser is sick
            print("A")
            optimiser = torch.optim.SGD(
```



```
                self.model.parameters(),
                lr=self.learning_rate,
                momentum=0.9,
            )

            # self.model.parameters(), lr=self.learning_rate)

            # shuffle data and split into batches using DataLoader
            print("B")
            rand_perm = np.random.permutation(len(input_data))
            shuffled_input_data = input_data[rand_perm]
            shuffled_target_data = target_data[rand_perm]

            # split into self.batch_size sized batches
            print("C")
            try:
                no_batches = int(x.shape[0] / self.batch_size)
                print("Start")
                print(no_batches)
                print(int(x.shape[0]))
                print(self.batch_size)
                print("End")
                input_batches = np.array_split(shuffled_input_data, no_batches)
                target_batches = np.array_split(shuffled_target_data, no_batches)
            except Exception as e:
                print(e)
                raise e

            print("D")
            for (input_batch, target_batch) in zip(input_batches, target_batches):
                optimiser.zero_grad()

                # Perform a forward pass through the model
                outputs = self.model(input_batch)

                # Compute loss
                # Compute gradient of loss via backwards pass
                loss = self.loss_fn(outputs, target_batch)
                loss.backward()

                # Change the weights via gradient decent
                optimiser.step()

            print("E")
            error_train = self.score(x, y)
            self.__loss_history.append(error_train)

            #   print(f"Epoch {epoch_n} error (train): {error_train}")

            print("F")
            if x_eval is not None and y_eval is not None:
                error_eval = self.score(x_eval, y_eval)
                self.__loss_history_eval.append(error_eval)

                # print(f"Epoch {epoch_n} error (eval): {error_eval}")

        print("G")

        return self.model

        #######################################################################
        #                        ** END OF YOUR CODE **
        #######################################################################

    def loss_history(self) -> List[float]:
```

```python
        return self.__loss_history

    def loss_history_eval(self) -> List[float]:
        return self.__loss_history_eval

    def predict(self, x):
        """
        Output the value corresponding to an input x.

        Arguments:
            x {pd.DataFrame} -- Raw input array of shape
                (batch_size, input_size).

        Returns:
            {np.ndarray} -- Predicted value for the given input (batch_size, 1).

        """

        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################

        X, _ = self._preprocessor(x, training=False)  # Do not forget

        O = self.model.forward(X)

        return self.__preprocessor.inverse_transform_target(O)

        #######################################################################
        #                       ** END OF YOUR CODE **
        #######################################################################

    def score(self, x, y):
        """
        Function to evaluate the model accuracy on a validation dataset.

        Arguments:
            - x {pd.DataFrame} -- Raw input array of shape
                (batch_size, input_size).
            - y {pd.DataFrame} -- Raw output array of shape (batch_size, 1).

        Returns:
            {float} -- Quantification of the efficiency of the model.

        """

        X_norm, Y_norm = self._preprocessor(x, y=y, training=False)  # Do not forget

        Y_pred_norm = self.model.forward(X_norm)

        Y_pred = self.__preprocessor.inverse_transform_target(Y_pred_norm)
        Y = self.__preprocessor.inverse_transform_target(Y_norm)

        return mean_squared_error(Y_pred, Y, squared=False)


def save_regressor(trained_model, model_name: Union[str, None] = None):
    """
    Utility function to save the trained regressor model in part2_model.pickle.
    """
    # If you alter this, make sure it works in tandem with load_regressor
    model_pickle_path = 'part2_model.pickle' if model_name is None \
        else f'assets/{model_name}-lr-{trained_model.learning_rate}-epch-{trained_model.nb_epoch}.pickle'
```



```python

    with open(model_pickle_path, 'wb') as target:
        pickle.dump(trained_model, target)
    print(f"\nSaved model in {model_pickle_path}")


def load_regressor(model_name: Union[str, None] = None):
    """
    Utility function to load the trained regressor model in part2_model.pickle.
    """
    model_pickle_path = 'part2_model.pickle' if model_name is None \
        else f'assets/{model_name}.pickle'

    # If you alter this, make sure it works in tandem with save_regressor
    with open(model_pickle_path, 'rb') as target:
        trained_model = pickle.load(target)
    print(f"\nLoaded model in {model_pickle_path}\n")
    return trained_model


def RegressorHyperParameterSearch():
    # Ensure to add whatever inputs you deem necessary to this function
    """
    Performs a hyper-parameter for fine-tuning the regressor implemented
    in the Regressor class.

    Arguments:
        Add whatever inputs you need.

    Returns:
        The function should return your optimised hyper-parameters.

    """

    #######################################################################
    #                       ** START OF YOUR CODE **
    #######################################################################

    return  # Return the chosen hyper parameters

    #######################################################################
    #                       ** END OF YOUR CODE **
    #######################################################################


def example_main():
    # Use pandas to read CSV data as it contains various object types
    # Feel free to use another CSV reader tool
    # But remember that LabTS tests take Pandas DataFrame as inputs
    train_data = pd.read_csv("housing.csv")
    eval_data = pd.read_csv("housing_eval.csv")

    data_main(train_data, eval_data)

def k_fold_main(k):
    data = pd.read_csv("housing.csv")

    chunk_size = data.shape[0] // k
    data_split = [data[i : i + chunk_size] for i in range(0, data.shape[0], chunk_size)]

    total = 0

    for i in range(0, k):
        # The eval data is this kth of the data.
        eval_data = data_split[i]
```

```python

        # The training data is all but the eval data
        train_data = data_split[:]
        del train_data[i]
        train_data = pd.concat(train_data)

        total += data_main(train_data, eval_data)

    print(f"Average score over {k} splits = {total / k}")


# Trains model with train_data,
# Evaluates with eval_data,
# Returns score
def data_main(train_data, eval_data) -> float:
    output_label = "median_house_value"

    # Splitting input and output
    x_train = train_data.loc[:, train_data.columns != output_label]
    y_train = train_data.loc[:, [output_label]]

    # Training
    # This example trains on the whole available dataset.
    # You probably want to separate some held-out data
    # to make sure the model isn't overfitting
    regressor = Regressor(x_train, nb_epoch=800, learning_rate=0.01)
    # regressor = load_regressor()
    regressor.fit(x_train, y_train)
    save_regressor(regressor)

    # Error
    error = regressor.score(x_train, y_train)
    print("\nRegressor error: {}".format(error))

    # Eval Error
    eval_x_train = eval_data.loc[:, eval_data.columns != output_label]
    eval_y_train = eval_data.loc[:, [output_label]]
    eval_error = regressor.score(eval_x_train, eval_y_train)
    print("\nRegressor error vs eval: {}\n".format(eval_error))

    return eval_error


if __name__ == "__main__":
    example_main()
    # k_fold_main(10)

```



```python
import numpy as np
import pickle


def xavier_init(size, gain = 1.0):
    """
    Xavier initialization of network weights.

    Arguments:
        - size {tuple} -- size of the network to initialise.
        - gain {float} -- gain for the Xavier initialisation.

    Returns:
        {np.ndarray} -- values of the weights.
    """
    low = -gain * np.sqrt(6.0 / np.sum(size))
    high = gain * np.sqrt(6.0 / np.sum(size))
    return np.random.uniform(low=low, high=high, size=size)


class Layer:
    """
    Abstract layer class.
    """

    def __init__(self, *args, **kwargs):
        raise NotImplementedError()

    def forward(self, *args, **kwargs):
        raise NotImplementedError()

    def __call__(self, *args, **kwargs):
        return self.forward(*args, **kwargs)

    def backward(self, *args, **kwargs):
        raise NotImplementedError()

    def update_params(self, *args, **kwargs):
        pass


class MSELossLayer(Layer):
    """
    MSELossLayer: Computes mean-squared error between y_pred and y_target.
    """

    def __init__(self):
        self._cache_current = None

    @staticmethod
    def _mse(y_pred, y_target):
        return np.mean((y_pred - y_target) ** 2)

    @staticmethod
    def _mse_grad(y_pred, y_target):
        return 2 * (y_pred - y_target) / len(y_pred)

    def forward(self, y_pred, y_target):
        self._cache_current = y_pred, y_target
        return self._mse(y_pred, y_target)

    def backward(self):
        return self._mse_grad(*self._cache_current)


class CrossEntropyLossLayer(Layer):
```

```python
    """
    CrossEntropyLossLayer: Computes the softmax followed by the negative
    log-likelihood loss.
    """

    def __init__(self):
        self._cache_current = None

    @staticmethod
    def softmax(x):
        numer = np.exp(x - x.max(axis=1, keepdims=True))
        denom = numer.sum(axis=1, keepdims=True)
        return numer / denom

    def forward(self, inputs, y_target):
        assert len(inputs) == len(y_target)
        n_obs = len(y_target)
        probs = self.softmax(inputs)
        self._cache_current = y_target, probs

        out = -1 / n_obs * np.sum(y_target * np.log(probs))
        return out

    def backward(self):
        y_target, probs = self._cache_current
        n_obs = len(y_target)
        return -1 / n_obs * (y_target - probs)


class SigmoidLayer(Layer):
    """
    SigmoidLayer: Applies sigmoid function elementwise.
    """

    def __init__(self):
        """
        Constructor of the Sigmoid layer.
        """
        self._cache_current = None

    def forward(self, x):
        """
        Performs forward pass through the Sigmoid layer.

        Logs information needed to compute gradient at a later stage in
        `_cache_current`.

        Arguments:
            x {np.ndarray} -- Input array of shape (batch_size, n_in).

        Returns:
            {np.ndarray} -- Output array of shape (batch_size, n_out)
        """

        self._cache_current = 1 / (1 + np.exp(-x))
        return self._cache_current


    def backward(self, grad_z):
        """
        Given `grad_z`, the gradient of some scalar (e.g. loss) with respect to
        the output of this layer, performs back pass through the layer (i.e.
        computes gradients of loss with respect to parameters of layer and
        inputs of layer).

        Arguments:
```



```python
            grad_z {np.ndarray} -- Gradient array of shape (batch_size, n_out).

        Returns:
            {np.ndarray} -- Array containing gradient with respect to layer
                input, of shape (batch_size, n_in).
        """

        derivative = self._cache_current * (1 - self._cache_current)
        return grad_z * derivative




class ReluLayer(Layer):
    """
    ReluLayer: Applies Relu function elementwise.
    """

    def __init__(self):
        """
        Constructor of the Relu layer.
        """
        self._cache_current = None

    def forward(self, x):
        """
        Performs forward pass through the Relu layer.

        Logs information needed to compute gradient at a later stage in
        `_cache_current`.

        Arguments:
            x {np.ndarray} -- Input array of shape (batch_size, n_in).

        Returns:
            {np.ndarray} -- Output array of shape (batch_size, n_out)
        """

        self._cache_current = np.where(x <= 0, 0, x)
        return self._cache_current

    def backward(self, grad_z):
        """
        Given `grad_z`, the gradient of some scalar (e.g. loss) with respect to
        the output of this layer, performs back pass through the layer (i.e.
        computes gradients of loss with respect to parameters of layer and
        inputs of layer).

        Arguments:
            grad_z {np.ndarray} -- Gradient array of shape (batch_size, n_out).

        Returns:
            {np.ndarray} -- Array containing gradient with respect to layer
                input, of shape (batch_size, n_in).
        """

        derivative = np.where(self._cache_current > 0, 1, self._cache_current)
        return grad_z * derivative


class LinearLayer(Layer):
    """
    LinearLayer: Performs affine transformation of input.
    """

    def __init__(self, n_in, n_out): #Â shake it all about
```

```python
        """
        Constructor of the linear layer.

        Arguments:
            - n_in {int} -- Number (or dimension) of inputs.
            - n_out {int} -- Number (or dimension) of outputs.
        """
        self.n_in = n_in
        self.n_out = n_out
        # shake it all about

        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################

        """
        Weights have the shape:

            (w_11, w_12, ..., w_1n_in)
            ...
            ...
            (w_n_out1, w_n_out2, ..., w_n_outn_in)

        where w_ij is the weight from the i-th input to the j-th output.

        Bias are initialized to 0, as a vector of size n_out.

        """

        self._W = xavier_init((n_in, n_out)) # shake it all about
        self._b = np.zeros((1, n_out))

        self._cache_current = None
        self._grad_W_current = None
        self._grad_b_current = None

        #######################################################################
        #                       ** END OF YOUR CODE **
        #######################################################################

    def forward(self, x):

        """
        Performs forward pass through the layer (i.e. returns Wx + b).

        Logs information needed to compute gradient at a later stage in
        `_cache_current`.

        Arguments:
            x {np.ndarray} -- Input array of shape (batch_size, n_in).

        Returns:
            {np.ndarray} -- Output array of shape (batch_size, n_out)
        """

        # store input array in cache for backpropagation
        self._cache_current = x
        return np.dot(x, self._W) + self._b


    def backward(self, grad_z):
        """
        Given `grad_z`, the gradient of some scalar (e.g. loss) with respect to
        the output of this layer, performs back pass through the layer (i.e.
        computes gradients of loss with respect to parameters of layer and
        inputs of layer).
```



```python

        Arguments:
            grad_z {np.ndarray} -- Gradient array of shape (batch_size, n_out).

        Returns:
            {np.ndarray} -- Array containing gradient with respect to layer
                input, of shape (batch_size, n_in).
        """

        # Compute gradient with respect to layer input
        self._grad_W_current = np.dot(self._cache_current.T, grad_z)

        # sum biases along columns
        self._grad_b_current = np.sum(grad_z, axis=0, keepdims=True)

        # Compute gradient with respect to layer parameters
        return np.dot(grad_z, self._W.T)

    def update_params(self, learning_rate):
        """
        Performs one step of gradient descent with given learning rate on the
        layer's parameters using currently stored gradients.

        Arguments:
            learning_rate {float} -- Learning rate of update step.
        """

        self._W -= learning_rate * self._grad_W_current
        self._b -= learning_rate * self._grad_b_current

class MultiLayerNetwork(object):
    """
    MultiLayerNetwork: A network consisting of stacked linear layers and
    activation functions.
    """

    def __init__(self, input_dim, neurons, activations):
        """
        Constructor of the multi layer network.

        Arguments:
            - input_dim {int} -- Number of features in the input (excluding
                the batch dimension).
            - neurons {list} -- Number of neurons in each linear layer
                represented as a list. The length of the list determines the
                number of linear layers.
            - activations {list} -- List of the activation functions to apply
                to the output of each linear layer.
        """
        self.input_dim = input_dim
        self.neurons = neurons
        self.activations = activations


        self._layers = []

        if (len(neurons) != len(activations)):
            raise ValueError("The number of layers and activations must be equal")

        for i in range(len(neurons)):
            if (i == 0):
                self._layers.append(LinearLayer(input_dim, neurons[i]))
            else:
                self._layers.append(LinearLayer(neurons[i-1], neurons[i]))

```

```python
            match activations[i]:
                case "relu":
                    self._layers.append(ReluLayer())
                case "sigmoid":
                    self._layers.append(SigmoidLayer())
                case "identity":
                    continue

    def forward(self, x):
        """
        Performs forward pass through the network.

        Arguments:
            x {np.ndarray} -- Input array of shape (batch_size, input_dim).

        Returns:
            {np.ndarray} -- Output array of shape (batch_size,
                #_neurons_in_final_layer)
        """

        for layer in self._layers:
            x = layer.forward(x)

        return x

    def __call__(self, x):
        return self.forward(x)

    def backward(self, grad_z):
        """
        Performs backward pass through the network.

        Arguments:
            grad_z {np.ndarray} -- Gradient array of shape (batch_size,
                #_neurons_in_final_layer).

        Returns:
            {np.ndarray} -- Array containing gradient with respect to layer
                input, of shape (batch_size, input_dim).
        """

        for layer in reversed(self._layers):
            grad_z = layer.backward(grad_z)

        return grad_z

    def update_params(self, learning_rate):
        """
        Performs one step of gradient descent with given learning rate on the
        parameters of all layers using currently stored gradients.

        Arguments:
            learning_rate {float} -- Learning rate of update step.
        """

        for layer in self._layers:
            layer.update_params(learning_rate)


def save_network(network, fpath):
    """
    Utility function to pickle `network` at file path `fpath`.
    """
    with open(fpath, "wb") as f:
        pickle.dump(network, f)

```



```python

def load_network(fpath):
    """
    Utility function to load network found at file path `fpath`.
    """
    with open(fpath, "rb") as f:
        network = pickle.load(f)
    return network


class Trainer(object):
    """
    Trainer: Object that manages the training of a neural network.
    """

    def __init__(
        self,
        network,
        batch_size,
        nb_epoch,
        learning_rate,
        loss_fun,
        shuffle_flag,
    ):
        """
        Constructor of the Trainer.

        Arguments:
            - network {MultiLayerNetwork} -- MultiLayerNetwork to be trained.
            - batch_size {int} -- Training batch size.
            - nb_epoch {int} -- Number of training epochs.
            - learning_rate {float} -- SGD learning rate to be used in training.
            - loss_fun {str} -- Loss function to be used. Possible values: mse,
                cross_entropy.
            - shuffle_flag {bool} -- If True, training data is shuffled before
                training.
        """
        self.network = network
        self.batch_size = batch_size
        self.nb_epoch = nb_epoch
        self.learning_rate = learning_rate
        self.loss_fun = loss_fun
        self.shuffle_flag = shuffle_flag

        match loss_fun:
            case "mse":
                self._loss_layer = MSELossLayer()
            case "cross_entropy":
                self._loss_layer = CrossEntropyLossLayer()


    @staticmethod
    def shuffle(input_dataset, target_dataset):
        """
        Returns shuffled versions of the inputs.

        Arguments:
            - input_dataset {np.ndarray} -- Array of input features, of shape
                (#_data_points, n_features) or (#_data_points,).
            - target_dataset {np.ndarray} -- Array of corresponding targets, of
                shape (#_data_points, #output_neurons).

        Returns:
            - {np.ndarray} -- shuffled inputs.
            - {np.ndarray} -- shuffled_targets.
        """
```

```python

        rand_perm = np.random.permutation(len(input_dataset))

        return (input_dataset[rand_perm], target_dataset[rand_perm])

    def train(self, input_dataset, target_dataset):
        """
        Main training loop. Performs the following steps `nb_epoch` times:
            - Shuffles the input data (if `shuffle` is True)
            - Splits the dataset into batches of size `batch_size`.
            - For each batch:
                - Performs forward pass through the network given the current
                batch of inputs.
                - Computes loss.
                - Performs backward pass to compute gradients of loss with
                respect to parameters of network.
                - Performs one step of gradient descent on the network
                parameters.

        Arguments:
            - input_dataset {np.ndarray} -- Array of input features, of shape
                (#_training_data_points, n_features).
            - target_dataset {np.ndarray} -- Array of corresponding targets, of
                shape (#_training_data_points, #output_neurons).
        """



        for _ in range(self.nb_epoch):

            if self.shuffle_flag:
                input_dataset, target_dataset = Trainer.shuffle(input_dataset, target_dataset)

            no_batches = int(input_dataset.shape[0] / self.batch_size)
            input_batches = np.array_split(input_dataset, no_batches)
            target_batches = np.array_split(target_dataset, no_batches)

            for i in range(no_batches):
                forward = self.network.forward(input_batches[i])
                self._loss_layer.forward(forward, target_batches[i])
                self.network.backward(self._loss_layer.backward())
                self.network.update_params(self.learning_rate)

    def eval_loss(self, input_dataset, target_dataset):
        """
        Function that evaluate the loss function for given data. Returns
        scalar value.

        Arguments:
            - input_dataset {np.ndarray} -- Array of input features, of shape
                (#_evaluation_data_points, n_features).
            - target_dataset {np.ndarray} -- Array of corresponding targets, of
                shape (#_evaluation_data_points, #output_neurons).

        Returns:
            a scalar value -- the loss
        """
        forward = self.network.forward(input_dataset)
        return self._loss_layer.forward(forward, target_dataset)



class Preprocessor(object):
    """
    Preprocessor: Object used to apply "preprocessing" operation to datasets.
```



```python
    The object can also be used to revert the changes.
    """

    def __init__(self, data):
        """
        Initializes the Preprocessor according to the provided dataset.
        (Does not modify the dataset.)

        Arguments:
            data {np.ndarray} dataset used to determine the parameters for
            the normalization.
        """
        self.min_range = 0
        self.max_range = 1

        self.min_data = np.min(data, axis=0)
        self.max_data = np.max(data, axis=0)

    def apply(self, data):
        """
        Apply the pre-processing operations to the provided dataset.

        Arguments:
            data {np.ndarray} dataset to be normalized.

        Returns:
            {np.ndarray} normalized dataset.
        """

        # Normalize the data using min-max normalization

        return ((data - self.min_data) * (self.max_range - self.min_range)) / (self.max_data - self.min_data)


    def revert(self, data):
        """
        Revert the pre-processing operations to retrieve the original dataset.

        Arguments:
            data {np.ndarray} dataset for which to revert normalization.

        Returns:
            {np.ndarray} reverted dataset.
        """

        return (data * (self.max_data - self.min_data)) / (self.max_range - self.min_range) + self.min_data


def example_main():
    input_dim = 4
    neurons = [16, 3]
    activations = ["relu", "identity"]
    net = MultiLayerNetwork(input_dim, neurons, activations)

    dat = np.loadtxt("iris.dat")
    np.random.shuffle(dat)

    x = dat[:, :4]
    y = dat[:, 4:]

    split_idx = int(0.8 * len(x))

    x_train = x[:split_idx]
    y_train = y[:split_idx]
```

```
    x_val = x[split_idx:]
    y_val = y[split_idx:]

    prep_input = Preprocessor(x_train)

    x_train_pre = prep_input.apply(x_train)
    x_val_pre = prep_input.apply(x_val)

    trainer = Trainer(
        network=net,
        batch_size=8,
        nb_epoch=1000,
        learning_rate=0.01,
        loss_fun="cross_entropy",
        shuffle_flag=True,
    )

    trainer.train(x_train_pre, y_train)
    print("Train loss = ", trainer.eval_loss(x_train_pre, y_train))
    print("Validation loss = ", trainer.eval_loss(x_val_pre, y_val))

    preds = net(x_val_pre).argmax(axis=1).squeeze()
    targets = y_val.argmax(axis=1).squeeze()
    accuracy = (preds == targets).mean()
    print("Validation accuracy: {}".format(accuracy))


if __name__ == "__main__":
    example_main()
```

```
-------- Test Output --------


PART 1 test output:


PART 2 test output:
A
A
A
B
C
Start
0
11
32
End
number sections must be larger than 0.

Exception thrown when creating and using an instance of Regressor.

Loaded model in part2_model.pickle


Expected RMSE error on the training data: 90000
Obtained RMSE error on the training data: 38608.01953125
Succesfully reached the minimum performance threshold. Well done!

-------- Test Errors --------

```