HU, Ruoyu (rh4618)

Imperial College London

Department of Computing

Academic Year **2020-2021**

Page created Fri Nov 27 23:16:04 GMT 2020

CATE

70050 mrei 2

c3 rh4618 v7

0 0 7 2 0 0 0 2 1 4 7 7

Electronic submission

Fri - 27 Nov 2020 14:24:41

rh4618

Exercise Information

Module:	70050 Introduction to Machine Learning (Term1)	**Issued:**	Mon - 09 Nov 2020
Exercise:	2 (CBC)	**Due:**	Fri - 27 Nov 2020
Title:	Neural Networks	**Assessment:**	Group
FAO:	Rei, Marek (mrei)	**Submission:**	Electronic

Student Declaration - Version 7

- I declare that this final submitted version is my unaided work.

Signed: (electronic signature) Date: 2020-11-26 22:15:36

For Markers only: (circle appropriate grade)

HU, Ruoyu (rh4618) LEADER	01494690	c3	2020-11-26 22:15:36	A* A B C D E F
DEJL, Adam (ad5518)	01561662	c3	2020-11-26 22:16:35	A* A B C D E F
DING, Ke (kd120)	01328574	t5	2020-11-27 12:33:50	A* A B C D E F
MANGAL, Pranav (prm2418)	01487364	c3	2020-11-27 04:41:02	A* A B C D E F

COURSEWORK FEEDBACK FORM - Rh4618

TOTAL GRADE
94%

	GRADE	COMMENTS
Part 1	**50 / 50**	The submission successfully passed all the LabTS tests. The code is well-written, easily readable and efficient. Really well-done !
Implement an architecture for regression	**15 / 15**	A sensible regression architecture with input, hidden layers and output layer without activation function (since it is a regression task) is described in the report. The code submission successfully passed all the LabTS tests with an excellent RMSE error on the training data.
Evaluate your architecture	**8 / 10**	The preprocessing step is correct and nicely described. The NaNs are correctly filled with the mean values (an alternative would be median value since its more resilient to outliers) and one-hot encoding is used for the categorical ocean proximity. The data are correctly normalised using the MinMaxScaler. A more detailed analysis of the dataset would have shown you some highly correlated features that you could potentially combine. The RMSE as the evaluation metric and Cross-Validation as the evaluation method are correctly chosen. In the "Evaluation & Results" section, it is not clear what model architecture the experiments are conducted with. Although, the theoretical principles about dropout, regularization and early-stopping are correct, it is not clear to the reader the model / models that these findings refer to.
Fine tune your architecture	**9 / 10**	Generally, the hyper-parameter space is well-defined and the parameter choices are correctly justified. The step-by-step hyper-parameter space exploration is nicely done with the parameter choices to be well-justified. Instead of just stating that Adam and SGD would have no difference, some experiments could help you back this claim with empirical evidence. Excellent for including the learning curves of the final model and nicely stating future exploration paths.
Report quality	**12 / 15**	The report is well-structured and within the page limit. All the graphs and tables are well-presented. Some minor mistakes for example Figure 2 has no explanation about what each line represents. The three tables in the Appendix are of high importance as they contain your hyperparemeter selection experiments and they should have been in the main body of the report. In general, try to include all the important details in the main body and use the Appendix section for extra graphs, experiments, tables, proofs etc. that won't affect the reader's understanding of the discussed topic if they skip the Appendix section. Finally, some sections in the report seemed like a code walkthrough discussing the functions and methods implemented in your code which needs to be avoided. The report should be a document independent from the code where we document our approach, the steps we took to solve a problem rather than a code walkthrough.

```
Final Tests: Summary for  of c3
-------------------------------

  Hidden Tests:
    Part 1 -- Activations: relu backward:                 1 / 1
    Part 1 -- Activations: relu forward:                  1 / 1
    Part 1 -- Activations: sigmoid backward:              1 / 1
    Part 1 -- Activations: sigmoid forward:               1 / 1
    Part 1 -- Linear layer: backward (Private only):      1 / 1
    Part 1 -- Linear layer: forward (Private only):       1 / 1
    Part 1 -- Linear layer: shape mismatch:               1 / 1
    Part 1 -- Linear layer: smoke test:                   1 / 1
    Part 1 -- Linear layer: weight update (Private only): 1 / 1
    Part 1 -- Linear layer: zero update:                  1 / 1
    Part 1 -- Network: smoke test:                        1 / 1
    Part 1 -- Network: zero update:                       1 / 1
    Part 1 -- Preprocess: different dataset:              1 / 1
    Part 1 -- Preprocess: min-max scale:                  1 / 1
    Part 1 -- Preprocess: revert:                         1 / 1
    Part 1 -- Preprocess: shape mismatch:                 1 / 1
    Part 1 -- Trainer: 1D regression:                     1 / 1
    Part 1 -- Trainer: shuffle preserves pairs:           1 / 1
    Part 1 -- Trainer: shuffle removes correlations:      1 / 1
    Part 2: Regressor:                                    1 / 1
    Part 2: Preprocessing:                                1 / 1
    Part 2: Performance:                                  1 / 1

Git Repo: git@gitlab.doc.ic.ac.uk:lab2021_autumn/neural_networks_65.git
Commit ID: 7b588
```

```python
import copy

import torch
import torch.nn as nn

import pickle
import numpy as np
import pandas as pd

from sklearn.utils import shuffle
from sklearn.preprocessing import LabelBinarizer, MinMaxScaler
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import GridSearchCV, train_test_split, \
    cross_val_score
from sklearn.base import BaseEstimator

from part2_network import Network


class Regressor(BaseEstimator):

    def __init__(self,
                 x=None,
                 nb_epoch=500,
                 n_hidden=1,
                 n_nodes=None,
                 activations=None,
                 optimiser=torch.optim.Adam,
                 early_stopping=True,
                 earliest_stop=100,
                 patience=3,
                 dropout=False):
        """
        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 epoch to train the network.
            - n_hidden {int} -- number of layers with activation functions,
                includes the input layer as it requires activation function
            - n_nodes {List[int]} -- number of nodes in each layer except
                input layer, that is generated automatically
            - activations {List[nn.modules.activation]} -- list of activation
                functions for our model
            - optimiser {torch.optim} -- optimiser to be used for the model
            - early_stopping {boolean} -- whether early stopping is enabled for
                traininig the model
            - earliest_stop {int} -- earliest epoch to begin to consier early
                stopping, used to prevent stopping too early before model starts
                converging
            - patience {int} -- number of consecutive epochs without a better
                model before stopping
            - dropout {boolean} -- whether dropout is enabled during training

        """

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

        self.binariser_labels = None
        self.y_scaler = None
        self.learning_rate = 0.01

```



```python
        self.x = x
        self.nb_epoch = nb_epoch
        self.n_hidden = n_hidden
        self.n_nodes = n_nodes

        self.output_size = 1
        if x is not None:
            X, _ = self._preprocessor(x, training=True)
            self.input_samples, self.input_size = X.shape
        else:
            self.input_samples = 0
            self.input_size = 13

        # Pre-set or placeholder model variables
        if activations is not None:
            self.activations = activations
        else:
            self.activations = [nn.ReLU() for i in range(self.n_hidden)]

        self.n_inputs = None
        self.model = None
        self.criterion = nn.MSELoss()
        self.optimiser = optimiser

        # Optimisation features
        self.early_stopping = early_stopping
        self.earliest_stop = earliest_stop
        self.patience = patience
        self.dropout = dropout

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

    def _build_optimiser(self):
        """
        Builds an optimiser from the set optimiser types

        Returns: {nn.optim} instantiated optimiser
        """
        # Use given model optimiser
        if self.optimiser == torch.optim.Adam:
            return self.optimiser(self.model.parameters(),
                                  lr=self.learning_rate,
                                  weight_decay=1e-4)

        # Use classic SGD
        return self.optimiser(self.model.parameters(), lr=self.learning_rate)

    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} -- Preprocessed input array of size
                (batch_size, input_size).
            - {torch.tensor} -- Preprocessed target array of size
                (batch_size, 1).

```

```
        """

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

        # Fill NaN values within the numerical columns of the given input data
        numerical_x = x.drop("ocean_proximity", axis=1)
        fill_keys = numerical_x.median().to_dict()
        numerical_x = numerical_x.fillna(value=fill_keys)

        # Normalise numerical values to scale to values between 0 and 1
        numerical_labels = numerical_x.keys()
        x_scaler = MinMaxScaler()
        numerical_x = pd.DataFrame(data=x_scaler.fit_transform(numerical_x),
                                   columns=numerical_labels)

        # Perform one-hot encoding on textual values "ocean_proximity" replace
        # with columns of x_ij in [0, 1] for each label
        binariser = LabelBinarizer()
        ocean_proximity = x["ocean_proximity"]

        if training:
            # Create new binariser parameters for one-hot encoding
            binarised_data = binariser.fit_transform(ocean_proximity)
            self.binariser_labels = binariser.classes_
        else:
            # Use existing binariser parameters for one-hot encoding
            binariser.fit(self.binariser_labels)
            binarised_data = binariser.transform(ocean_proximity)

        # One-hot encoding parameters
        op_frame = pd.DataFrame(data=binarised_data,
                                columns=binariser.classes_)

        # Combine numerical and one-hot encoded textual data frames
        x = pd.concat([numerical_x, op_frame], axis=1)

        # Process on CPU, as Lab computers throw CUDA error
        device = torch.device('cpu')

        # Create tensor from preprocessed x data
        x_tensor = torch.tensor(x.values, device=device, requires_grad=True)
        y_tensor = None

        if isinstance(y, pd.DataFrame):
            # Normalise y values if available, store scaler to be used to undo
            # scaling
            self.y_scaler = MinMaxScaler()
            y = pd.DataFrame(data=self.y_scaler.fit_transform(y),
                             columns=y.keys())
            y_tensor = torch.tensor(y.values, device=device, requires_grad=True)

        # Return preprocessed x and y, return None for y if it was None
        return x_tensor, y_tensor

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

    def fit(self, x, y):
        """
        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.

        """

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

        # In order to match sklearn estimator API convention, the model is built
        # here instead of in the constructor
        if self.n_nodes is not None:
            self.n_inputs = [self.input_size] + self.n_nodes
        else:
            self.n_inputs = [self.input_size] +\
                            [self.input_size for i in range(self.n_hidden)]

        # Neural network model
        self.model = Network(n_layers=self.n_hidden,
                             n_inputs=self.n_inputs)

        self.optimiser = self._build_optimiser()

        # Split data into train-validation if early stopping
        if self.early_stopping:
            x, val_x, y, val_y = train_test_split(x,
                                                  y,
                                                  train_size=0.8,
                                                  random_state=42)

        # Preprocess training data
        X_train, Y_train = self._preprocessor(x, y=y, training=True)

        if self.early_stopping:
            # Also preprocess validation data
            X_val, Y_val = self._preprocessor(val_x, val_y)

            # Values for determining and storing optimal model
            best_model = None
            min_loss = 999
            strikes = 0

        # Set model mode to train
        self.model.train()
        for epoch in range(self.nb_epoch):
            # Clear gradient for this iteration
            self.optimiser.zero_grad()

            # Forward pass and loss calculation
            y_predicted = self.model(X_train,
                                     self.activations,
                                     dropout=self.dropout)

            loss = self.criterion(y_predicted, Y_train)

            # Backpropagation
            loss.backward()
            self.optimiser.step()

            if self.early_stopping:
                # Predict off validatation data and calculate loss
                val_y_pred = self.model(X_val, self.activations)
                val_loss = self.criterion(val_y_pred, Y_val)
```

```
                # print(f'val loss: {val_loss:.5f} best loss: {min_loss:.5f}')

                # Determine if training should stop early
                if val_loss.item() > min_loss:
                    if epoch >= self.earliest_stop:
                        if strikes == self.patience:
                            # Early stop
                            self.model = best_model
                            return self
                        else:
                            strikes += 1
                else:
                    # Clear strikes and save optimal model
                    strikes = 1
                    min_loss = val_loss.item()
                    best_model = copy.deepcopy(self.model)

            if (epoch + 1) % 10 == 0:
                print(f'epoch {epoch + 1} / {self.nb_epoch} loss = {loss.item():.8f}')

        if self.early_stopping:
            # Return optimal model if available
            self.model = best_model
        return self

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

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

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

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

        """

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

        if self.model is not None:
            X, _ = self._preprocessor(x, training=False)  # Do not forget

            # Set model to evaluate mode
            self.model.eval()

            # Predict from input and scale result using y scaler from training
            y_predicted = self.model(X, self.activations).detach().numpy()
            y_predicted_scaled = self.y_scaler.inverse_transform(y_predicted)

            return y_predicted_scaled

        return None

        #######################################################################
        #                       ** 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 ouput array of shape (batch_size, 1).

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

        """

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

        y_predicted = self.predict(x)
        mse = mean_squared_error(y, y_predicted)
        return np.sqrt(mse)

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


def save_regressor(trained_model):
    """
    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
    with open('part2_model.pickle', 'wb') as target:
        pickle.dump(trained_model, target)
    print("\nSaved model in part2_model.pickle\n")


def load_regressor():
    """
    Utility function to load the trained regressor model in part2_model.pickle.
    """
    # If you alter this, make sure it works in tandem with save_regressor
    with open('part2_model.pickle', 'rb') as target:
        trained_model = pickle.load(target)
    print("\nLoaded model in part2_model.pickle\n")
    return trained_model


def RegressorHyperParameterSearch(x, y):
    # 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:
        - x {pd.DataFrame} -- Raw input array of shape
            (batch_size, input_size).
        - y {pd.DataFrame} -- Raw target array of shape (batch_size, 1)

    Returns:
        - {dict{str, any}} optimised hyper-parameters.

    """

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

```python

    param_grid = [
        {
            'n_hidden': [1],
            'n_nodes': [[13], [24], [32]],
            'activations': [
                [nn.Sigmoid()],
                [nn.ReLU()],
                [nn.Tanh()]
            ]
        },
        {
            'n_hidden': [2],
            'n_nodes': [[13, 13], [20, 20], [24, 32], [24, 64]],
            'activations': [
                [nn.Sigmoid(), nn.Sigmoid()],
                [nn.ReLU(), nn.ReLU()],
                [nn.Tanh(), nn.Tanh()]
            ]
        },
        {
            'n_hidden': [3],
            'n_nodes': [[13, 13, 13], [32, 64, 32], [64, 128, 64], [512, 512, 128]],
            'activations': [
                [nn.Sigmoid(), nn.Sigmoid(), nn.Sigmoid()],
                [nn.Sigmoid(), nn.ReLU(), nn.ReLU()],
                [nn.ReLU(), nn.ReLU(), nn.ReLU()],
                [nn.ReLU(), nn.Sigmoid(), nn.ReLU()]
            ],
            'patience': [1, 3, 5, 8],
            'dropout': [True, False]
        },
        {
            'n_hidden': [4],
            'n_nodes': [[32, 64, 64, 32]],
            'activations': [
                [nn.Sigmoid(), nn.Tanh(), nn.ReLU(), nn.Sigmoid()],
                [nn.ReLU(), nn.ReLU(), nn.ReLU(), nn.ReLU()]
            ]
        },
    ]

    regressor = Regressor()
    grid_search = GridSearchCV(regressor,
                               param_grid,
                               cv=5,
                               scoring="neg_mean_squared_error")

    grid_search.fit(x, y)

    print(grid_search.best_score_)
    print(grid_search.best_params_)
    print(grid_search.best_estimator_)

    cvres = grid_search.cv_results_

    for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
        print(np.sqrt(-mean_score), params)

    # Return the chosen hyper parameters
    return grid_search.best_params_

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



```python


def example_main():
    output_label = "median_house_value"

    # 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
    data = pd.read_csv("housing.csv")

    # Set manual seed for result replication
    torch.manual_seed(42)

    # Shuffle data
    data = shuffle(data, random_state=42)
    data.reset_index(inplace=True, drop=True)

    # Spliting input and output
    x_raw = data.loc[:, data.columns != output_label]
    y_raw = data.loc[:, [output_label]]

    x_train, x_test, y_train, y_test \
        = train_test_split(x_raw, y_raw, test_size=0.2, random_state=42)

    # Training
    # Build Regressor
    acitivations = [nn.ReLU(), nn.ReLU(), nn.ReLU()]
    n_nodes = [512, 512, 128]
    regressor = Regressor(x_train,
                          nb_epoch=500,
                          n_hidden=3,
                          n_nodes=n_nodes,
                          activations=acitivations,
                          patience=5,
                          dropout=False)
    # regressor.fit(x_train, y_train)
    # save_regressor(regressor)

    # Perform cross validation and obtain average error
    nmse_score = -cross_val_score(regressor,
                                  x_raw,
                                  y_raw,
                                  scoring="neg_mean_squared_error",
                                  cv=5)

    scores = np.sqrt(nmse_score)

    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard Deviation:", scores.std())

    # RegressorHyperParameterSearch(x_raw, y_raw)

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


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

```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)
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        sigmoid = 1 / (1 + np.exp(-x))
        self._cache_current = sigmoid

        return sigmoid

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

    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 repect to layer
                input, of shape (batch_size, n_in).
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        a = self._cache_current
        sigmoid_derivative = a * (1 - a)

        return grad_z * sigmoid_derivative

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


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)
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        self._cache_current = x

        return np.maximum(0, x)

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

    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 repect to layer
                input, of shape (batch_size, n_in).
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        z = self._cache_current
        relu_derivative = np.int64(z > 0)

        return grad_z * relu_derivative

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


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

    def __init__(self, n_in, n_out):
        """
        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

        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        self._W = xavier_init((n_in, n_out))
        self._b = np.zeros(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)
        """
        #######################################################################
```

```
        #                       ** START OF YOUR CODE **
        #######################################################################
        self._cache_current = x
        return x @ self._W + self._b

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

    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 repect to layer
                input, of shape (batch_size, n_in).
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        self._grad_W_current = self._cache_current.transpose() @ grad_z
        self._grad_b_current = np.ones((1, np.shape(grad_z)[0])) @ grad_z
        return grad_z @ self._W.transpose()

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

    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.
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        self._W = self._W - learning_rate * self._grad_W_current
        self._b = self._b - learning_rate * self._grad_b_current

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


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).
```



```
Â Â Â Â Â Â  - neurons {list} -- Number of neurons in each linear layer
                represented as aÂ 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

        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        ACTIVATIONS_MAP = {
            "sigmoid": lambda size: SigmoidLayer(),
            "relu": lambda size: ReluLayer(),
            "identity": lambda size: LinearLayer(size, size)
        }

        self._layers = []
        prev_dim = input_dim
        for num_neurons, activation in zip(neurons, activations):
            self._layers.append(LinearLayer(prev_dim, num_neurons))
            self._layers.append(ACTIVATIONS_MAP[activation](num_neurons))
            prev_dim = num_neurons

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

    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)
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        prev = x
        for layer in self._layers:
            prev = layer.forward(prev)
        return prev

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

    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 (1,
                #_neurons_in_final_layer).

        Returns:
            {np.ndarray} -- Array containing gradient with repect to layer
```

```
                input, of shape (batch_size, input_dim).
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        prev = grad_z
        for layer in reversed(self._layers):
            prev = layer.backward(prev)
        return prev

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

    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.
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        for layer in self._layers:
            layer.update_params(learning_rate)

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


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


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,
                bce.
            - 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

        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        LOSS_FUN_MAP = {
            "mse": MSELossLayer,
            "cross_entropy": CrossEntropyLossLayer,
        }

        self._loss_layer = LOSS_FUN_MAP[loss_fun]()
        #######################################################################
        #                       ** END OF YOUR CODE **
        #######################################################################

    @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).
            - target_dataset {np.ndarray} -- Array of corresponding targets, of
                shape (#_data_points, #output_neurons).

        Returns:
            - {np.ndarray} -- shuffled inputs.
            - {np.ndarray} -- shuffled_targets.
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        assert len(input_dataset) == len(target_dataset)
        perm = np.random.permutation(len(input_dataset))
        return input_dataset[perm], target_dataset[perm]

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

    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).
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        for i in range(self.nb_epoch):
            if self.shuffle_flag:
                input_dataset, target_dataset = self.shuffle(
                    input_dataset,
                    target_dataset
                )

            input_batches = np.array_split(input_dataset, self.batch_size)
            target_batches = np.array_split(target_dataset, self.batch_size)

            for input_batch, target_batch in zip(input_batches, target_batches):
                network_output = self.network(input_batch)
                self._loss_layer(network_output, target_batch)
                grad_loss_wrt_outputs = self._loss_layer.backward()
                self.network.backward(grad_loss_wrt_outputs)
                self.network.update_params(self.learning_rate)

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

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

        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).
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        network_output = self.network(input_dataset)
        return self._loss_layer(network_output, target_dataset)

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


class Preprocessor(object):
    """
    Preprocessor: Object used to apply "preprocessing" operation to datasets.
    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.
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        self.min = np.min(data, axis=0)
        self.max_min_diff = np.max(data, axis=0) - self.min

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

    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.
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        return (data - self.min) / self.max_min_diff

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

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

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

        Returns:
            {np.ndarray} reverted dataset.
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        return data * self.max_min_diff + self.min

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


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:


Loaded model in part2_model.pickle


Expected RMSE error on the test data: 90000
Obtained RMSE error on the test data: 65548.515625
Succesfully reached the minimum performance threshold. Well done!

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

```

```
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:             1 / 1
    Part 2: Preprocessing:         1 / 1
    Part 2: Performance:           1 / 1

Git Repo: git@gitlab.doc.ic.ac.uk:lab2021_autumn/neural_networks_65.git
Commit ID: 7b588
```

```
import copy

import torch
import torch.nn as nn

import pickle
import numpy as np
import pandas as pd

from sklearn.utils import shuffle
from sklearn.preprocessing import LabelBinarizer, MinMaxScaler
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import GridSearchCV, train_test_split, \
    cross_val_score
from sklearn.base import BaseEstimator

from part2_network import Network


class Regressor(BaseEstimator):

    def __init__(self,
                 x=None,
                 nb_epoch=500,
                 n_hidden=1,
                 n_nodes=None,
                 activations=None,
                 optimiser=torch.optim.Adam,
                 early_stopping=True,
                 earliest_stop=100,
                 patience=3,
                 dropout=False):
        """
        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 epoch to train the network.
            - n_hidden {int} -- number of layers with activation functions,
                includes the input layer as it requires activation function
            - n_nodes {List[int]} -- number of nodes in each layer except
                input layer, that is generated automatically
            - activations {List[nn.modules.activation]} -- list of activation
                functions for our model
            - optimiser {torch.optim} -- optimiser to be used for the model
            - early_stopping {boolean} -- whether early stopping is enabled for
                traininig the model
            - earliest_stop {int} -- earliest epoch to begin to consier early
                stopping, used to prevent stopping too early before model starts
                converging
            - patience {int} -- number of consecutive epochs without a better
                model before stopping
            - dropout {boolean} -- whether dropout is enabled during training

        """

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

        self.binariser_labels = None
        self.y_scaler = None
        self.learning_rate = 0.01

```



```
        self.x = x
        self.nb_epoch = nb_epoch
        self.n_hidden = n_hidden
        self.n_nodes = n_nodes

        self.output_size = 1
        if x is not None:
            X, _ = self._preprocessor(x, training=True)
            self.input_samples, self.input_size = X.shape
        else:
            self.input_samples = 0
            self.input_size = 13

        # Pre-set or placeholder model variables
        if activations is not None:
            self.activations = activations
        else:
            self.activations = [nn.ReLU() for i in range(self.n_hidden)]

        self.n_inputs = None
        self.model = None
        self.criterion = nn.MSELoss()
        self.optimiser = optimiser

        # Optimisation features
        self.early_stopping = early_stopping
        self.earliest_stop = earliest_stop
        self.patience = patience
        self.dropout = dropout

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

    def _build_optimiser(self):
        """
        Builds an optimiser from the set optimiser types

        Returns: {nn.optim} instantiated optimiser
        """
        # Use given model optimiser
        if self.optimiser == torch.optim.Adam:
            return self.optimiser(self.model.parameters(),
                                  lr=self.learning_rate,
                                  weight_decay=1e-4)

        # Use classic SGD
        return self.optimiser(self.model.parameters(), lr=self.learning_rate)

    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} -- Preprocessed input array of size
                (batch_size, input_size).
            - {torch.tensor} -- Preprocessed target array of size
                (batch_size, 1).

```

```
        """

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

        # Fill NaN values within the numerical columns of the given input data
        numerical_x = x.drop("ocean_proximity", axis=1)
        fill_keys = numerical_x.median().to_dict()
        numerical_x = numerical_x.fillna(value=fill_keys)

        # Normalise numerical values to scale to values between 0 and 1
        numerical_labels = numerical_x.keys()
        x_scaler = MinMaxScaler()
        numerical_x = pd.DataFrame(data=x_scaler.fit_transform(numerical_x),
                                   columns=numerical_labels)

        # Perform one-hot encoding on textual values "ocean_proximity" replace
        # with columns of x_ij in [0, 1] for each label
        binariser = LabelBinarizer()
        ocean_proximity = x["ocean_proximity"]

        if training:
            # Create new binariser parameters for one-hot encoding
            binarised_data = binariser.fit_transform(ocean_proximity)
            self.binariser_labels = binariser.classes_
        else:
            # Use existing binariser parameters for one-hot encoding
            binariser.fit(self.binariser_labels)
            binarised_data = binariser.transform(ocean_proximity)

        # One-hot encoding parameters
        op_frame = pd.DataFrame(data=binarised_data,
                                columns=binariser.classes_)

        # Combine numerical and one-hot encoded textual data frames
        x = pd.concat([numerical_x, op_frame], axis=1)

        # Process on CPU, as Lab computers throw CUDA error
        device = torch.device('cpu')

        # Create tensor from preprocessed x data
        x_tensor = torch.tensor(x.values, device=device, requires_grad=True)
        y_tensor = None

        if isinstance(y, pd.DataFrame):
            # Normalise y values if available, store scaler to be used to undo
            # scaling
            self.y_scaler = MinMaxScaler()
            y = pd.DataFrame(data=self.y_scaler.fit_transform(y),
                             columns=y.keys())
            y_tensor = torch.tensor(y.values, device=device, requires_grad=True)

        # Return preprocessed x and y, return None for y if it was None
        return x_tensor, y_tensor

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

    def fit(self, x, y):
        """
        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.

        """

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

        # In order to match sklearn estimator API convention, the model is built
        # here instead of in the constructor
        if self.n_nodes is not None:
            self.n_inputs = [self.input_size] + self.n_nodes
        else:
            self.n_inputs = [self.input_size] +\
                        [self.input_size for i in range(self.n_hidden)]

        # Neural network model
        self.model = Network(n_layers=self.n_hidden,
                             n_inputs=self.n_inputs)

        self.optimiser = self._build_optimiser()

        # Split data into train-validation if early stopping
        if self.early_stopping:
            x, val_x, y, val_y = train_test_split(x,
                                                  y,
                                                  train_size=0.8,
                                                  random_state=42)

        # Preprocess training data
        X_train, Y_train = self._preprocessor(x, y=y, training=True)

        if self.early_stopping:
            # Also preprocess validation data
            X_val, Y_val = self._preprocessor(val_x, val_y)

            # Values for determining and storing optimal model
            best_model = None
            min_loss = 999
            strikes = 0

        # Set model mode to train
        self.model.train()
        for epoch in range(self.nb_epoch):
            # Clear gradient for this iteration
            self.optimiser.zero_grad()

            # Forward pass and loss calculation
            y_predicted = self.model(X_train,
                                     self.activations,
                                     dropout=self.dropout)

            loss = self.criterion(y_predicted, Y_train)

            # Backpropagation
            loss.backward()
            self.optimiser.step()

            if self.early_stopping:
                # Predict off validatation data and calculate loss
                val_y_pred = self.model(X_val, self.activations)
                val_loss = self.criterion(val_y_pred, Y_val)
```

```
                # print(f'val loss: {val_loss:.5f} best loss: {min_loss:.5f}')

                # Determine if training should stop early
                if val_loss.item() > min_loss:
                    if epoch >= self.earliest_stop:
                        if strikes == self.patience:
                            # Early stop
                            self.model = best_model
                            return self
                        else:
                            strikes += 1
                else:
                    # Clear strikes and save optimal model
                    strikes = 1
                    min_loss = val_loss.item()
                    best_model = copy.deepcopy(self.model)

            if (epoch + 1) % 10 == 0:
                print(f'epoch {epoch + 1} / {self.nb_epoch} loss = {loss.item():.8f}')

        if self.early_stopping:
            # Return optimal model if available
            self.model = best_model
        return self

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

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

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

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

        """

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

        if self.model is not None:
            X, _ = self._preprocessor(x, training=False)  # Do not forget

            # Set model to evaluate mode
            self.model.eval()

            # Predict from input and scale result using y scaler from training
            y_predicted = self.model(X, self.activations).detach().numpy()
            y_predicted_scaled = self.y_scaler.inverse_transform(y_predicted)

            return y_predicted_scaled

        return None

        #######################################################################
        #                       ** 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 ouput array of shape (batch_size, 1).

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

        """

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

        y_predicted = self.predict(x)
        mse = mean_squared_error(y, y_predicted)
        return np.sqrt(mse)

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


def save_regressor(trained_model):
    """
    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
    with open('part2_model.pickle', 'wb') as target:
        pickle.dump(trained_model, target)
    print("\nSaved model in part2_model.pickle\n")


def load_regressor():
    """
    Utility function to load the trained regressor model in part2_model.pickle.
    """
    # If you alter this, make sure it works in tandem with save_regressor
    with open('part2_model.pickle', 'rb') as target:
        trained_model = pickle.load(target)
    print("\nLoaded model in part2_model.pickle\n")
    return trained_model


def RegressorHyperParameterSearch(x, y):
    # 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:
        - x {pd.DataFrame} -- Raw input array of shape
            (batch_size, input_size).
        - y {pd.DataFrame} -- Raw target array of shape (batch_size, 1)

    Returns:
        - {dict{str, any}} optimised hyper-parameters.

    """

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

```python

    param_grid = [
        {
            'n_hidden': [1],
            'n_nodes': [[13], [24], [32]],
            'activations': [
                [nn.Sigmoid()],
                [nn.ReLU()],
                [nn.Tanh()]
            ]
        },
        {
            'n_hidden': [2],
            'n_nodes': [[13, 13], [20, 20], [24, 32], [24, 64]],
            'activations': [
                [nn.Sigmoid(), nn.Sigmoid()],
                [nn.ReLU(), nn.ReLU()],
                [nn.Tanh(), nn.Tanh()]
            ]
        },
        {
            'n_hidden': [3],
            'n_nodes': [[13, 13, 13], [32, 64, 32], [64, 128, 64], [512, 512, 128]],
            'activations': [
                [nn.Sigmoid(), nn.Sigmoid(), nn.Sigmoid()],
                [nn.Sigmoid(), nn.ReLU(), nn.ReLU()],
                [nn.ReLU(), nn.ReLU(), nn.ReLU()],
                [nn.ReLU(), nn.Sigmoid(), nn.ReLU()]
            ],
            'patience': [1, 3, 5, 8],
            'dropout': [True, False]
        },
        {
            'n_hidden': [4],
            'n_nodes': [[32, 64, 64, 32]],
            'activations': [
                [nn.Sigmoid(), nn.Tanh(), nn.ReLU(), nn.Sigmoid()],
                [nn.ReLU(), nn.ReLU(), nn.ReLU(), nn.ReLU()]
            ]
        },
    ]

    regressor = Regressor()
    grid_search = GridSearchCV(regressor,
                               param_grid,
                               cv=5,
                               scoring="neg_mean_squared_error")

    grid_search.fit(x, y)

    print(grid_search.best_score_)
    print(grid_search.best_params_)
    print(grid_search.best_estimator_)

    cvres = grid_search.cv_results_

    for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
        print(np.sqrt(-mean_score), params)

    # Return the chosen hyper parameters
    return grid_search.best_params_

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



```python


def example_main():
    output_label = "median_house_value"

    # 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
    data = pd.read_csv("housing.csv")

    # Set manual seed for result replication
    torch.manual_seed(42)

    # Shuffle data
    data = shuffle(data, random_state=42)
    data.reset_index(inplace=True, drop=True)

    # Spliting input and output
    x_raw = data.loc[:, data.columns != output_label]
    y_raw = data.loc[:, [output_label]]

    x_train, x_test, y_train, y_test \
        = train_test_split(x_raw, y_raw, test_size=0.2, random_state=42)

    # Training
    # Build Regressor
    acitivations = [nn.ReLU(), nn.ReLU(), nn.ReLU()]
    n_nodes = [512, 512, 128]
    regressor = Regressor(x_train,
                          nb_epoch=500,
                          n_hidden=3,
                          n_nodes=n_nodes,
                          activations=acitivations,
                          patience=5,
                          dropout=False)
    # regressor.fit(x_train, y_train)
    # save_regressor(regressor)

    # Perform cross validation and obtain average error
    nmse_score = -cross_val_score(regressor,
                                  x_raw,
                                  y_raw,
                                  scoring="neg_mean_squared_error",
                                  cv=5)

    scores = np.sqrt(nmse_score)

    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard Deviation:", scores.std())

    # RegressorHyperParameterSearch(x_raw, y_raw)

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


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

```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)
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        sigmoid = 1 / (1 + np.exp(-x))
        self._cache_current = sigmoid

        return sigmoid

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

    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 repect to layer
                input, of shape (batch_size, n_in).
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        a = self._cache_current
        sigmoid_derivative = a * (1 - a)

        return grad_z * sigmoid_derivative

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


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)
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        self._cache_current = x

        return np.maximum(0, x)

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

    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 repect to layer
                input, of shape (batch_size, n_in).
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        z = self._cache_current
        relu_derivative = np.int64(z > 0)

        return grad_z * relu_derivative

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


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

    def __init__(self, n_in, n_out):
        """
        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

        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        self._W = xavier_init((n_in, n_out))
        self._b = np.zeros(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)
        """
        #######################################################################
```

```
        #                       ** START OF YOUR CODE **
        #######################################################################
        self._cache_current = x
        return x @ self._W + self._b

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

    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 repect to layer
                input, of shape (batch_size, n_in).
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        self._grad_W_current = self._cache_current.transpose() @ grad_z
        self._grad_b_current = np.ones((1, np.shape(grad_z)[0])) @ grad_z
        return grad_z @ self._W.transpose()

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

    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.
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        self._W = self._W - learning_rate * self._grad_W_current
        self._b = self._b - learning_rate * self._grad_b_current

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


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).
```



```
Â Â Â Â Â Â  - neurons {list} -- Number of neurons in each linear layer
                represented as aÂ 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

        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        ACTIVATIONS_MAP = {
            "sigmoid": lambda size: SigmoidLayer(),
            "relu": lambda size: ReluLayer(),
            "identity": lambda size: LinearLayer(size, size)
        }

        self._layers = []
        prev_dim = input_dim
        for num_neurons, activation in zip(neurons, activations):
            self._layers.append(LinearLayer(prev_dim, num_neurons))
            self._layers.append(ACTIVATIONS_MAP[activation](num_neurons))
            prev_dim = num_neurons

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

    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)
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        prev = x
        for layer in self._layers:
            prev = layer.forward(prev)
        return prev

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

    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 (1,
                #_neurons_in_final_layer).

        Returns:
            {np.ndarray} -- Array containing gradient with repect to layer
```

```python
                input, of shape (batch_size, input_dim).
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        prev = grad_z
        for layer in reversed(self._layers):
            prev = layer.backward(prev)
        return prev

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

    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.
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        for layer in self._layers:
            layer.update_params(learning_rate)

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


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


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.
```



```python

        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,
                bce.
            - 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

        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        LOSS_FUN_MAP = {
            "mse": MSELossLayer,
            "cross_entropy": CrossEntropyLossLayer,
        }

        self._loss_layer = LOSS_FUN_MAP[loss_fun]()
        #######################################################################
        #                       ** END OF YOUR CODE **
        #######################################################################

    @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).
            - target_dataset {np.ndarray} -- Array of corresponding targets, of
                shape (#_data_points, #output_neurons).

        Returns:
            - {np.ndarray} -- shuffled inputs.
            - {np.ndarray} -- shuffled_targets.
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        assert len(input_dataset) == len(target_dataset)
        perm = np.random.permutation(len(input_dataset))
        return input_dataset[perm], target_dataset[perm]

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

    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).
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        for i in range(self.nb_epoch):
            if self.shuffle_flag:
                input_dataset, target_dataset = self.shuffle(
                    input_dataset,
                    target_dataset
                )

            input_batches = np.array_split(input_dataset, self.batch_size)
            target_batches = np.array_split(target_dataset, self.batch_size)

            for input_batch, target_batch in zip(input_batches, target_batches):
                network_output = self.network(input_batch)
                self._loss_layer(network_output, target_batch)
                grad_loss_wrt_outputs = self._loss_layer.backward()
                self.network.backward(grad_loss_wrt_outputs)
                self.network.update_params(self.learning_rate)

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

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

        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).
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        network_output = self.network(input_dataset)
        return self._loss_layer(network_output, target_dataset)

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


class Preprocessor(object):
    """
    Preprocessor: Object used to apply "preprocessing" operation to datasets.
    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.
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        self.min = np.min(data, axis=0)
        self.max_min_diff = np.max(data, axis=0) - self.min

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

    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.
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        return (data - self.min) / self.max_min_diff

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

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

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

        Returns:
            {np.ndarray} reverted dataset.
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        return data * self.max_min_diff + self.min

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


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:

Loaded model in part2_model.pickle


Expected RMSE error on the training data: 90000
Obtained RMSE error on the training data: 65367.3046875
Succesfully reached the minimum performance threshold. Well done!

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

```

Introduction to ML - Artificial Neural Networks Report

rh4618, kd120, ad5518, prm2418

November 27, 2020

1 Regressor Architecture & Methodology

1.1 The Preprocessor Design

The `preprocessor` is designed to perform preprocessing on raw data for both training and prediction. For input raw features `x`, the first step is to fill missing values in numerical data in each column(feature) with the medians of the present instances of said feature, using `Pandas` function `fillna`. Then all the numerical feature values are normalised to values between 0 and 1, using `sklearn.preprocessing`'s `MinMaxScaler`.

Special treatment is given to the *ocean_proximity* feature column, as it contains textual data. One-hot encoding is performed using `sklearn.preprocessing`'s `LabelBinarizer` tool. First mapping each unique textual value to a list of 0s and 1s, then replacing the original column with new columns each corresponding to an encoding. The reason we do this is to dissociate the magnitude of the resulting mapped variable from the result. A possible alternative would have been to use integer mapping to map *ocean_proximity* values by distance to the ocean, though the labels in the given dataset can be a little difficult to discern an ordering and one-hot encoding was sufficient.

Finally, for input labels `y`, the same normalization method is applied using `MinMaxScaler`. We did this due to value of `y` being quite large by default, and we wanted to avoid having large weight values within our model, thus we down-scale the `y` values in our preprocessor, and up-scale the predicted value.

The values generated in the preprocessor, such as the mapping used in `LabelBinarizer` and the scalers used are then stored in the `Regressor` object to ensure consistency when training and using the model.

1.2 The Neural Network Architecture

The `Network` class extends the `nn.Module` class to allow for custom constructions of neural network models. It is designed to be flexible, allowing us to build networks of differing number of hidden layers and activation functions. Additionally, we are able to use it seamlessly with `Pytorch`'s loss function and optimiser components to train the model.

The `Network` class takes a number of layers and a list of number of neurons in each layer. A `torch tensor` is generated for each entry, an output layer is constructed separately, before all created layers are added as the `Network`'s parameters. The `tensor` layers are added using `nn.ModuleList(layers)` to the object to ensure that the layers are individually added as `parameters` of the module in adherence to `Pytorch` conventions.

For the forward pass, the `Network` class overwrites `nn.Module`'s default `forward()` function, taking the input data, a list of activation functions and an optional boolean for whether we perform dropout on our input and hidden layers. The input layer is applied to the data, and its corresponding activation function is applied, the result is then propagated through the network with each hidden layer and its activation function being applied in turn. Finally, the output layer is applied, without an explicit activation function, to preserve linearity as the given task is prediction over an unbounded space.

1.3 The Training Process

The `Regressor` class's `fit()` function includes all the training functionalities. For the sake of performing grid search, which we'll discuss further on, the model and its related loss function and optimiser are defined here, such that we create a new model every time we fit to new data.

We first call `_preprocessor()` on the given data, if early stopping is enabled, we split the given data into training and validation sets in a 4 : 1 split. The model is set to training mode, and for `nb_epoch` number of iterations, it performs a forward pass using the training data, calculates its loss from the resulting prediction and calls `optimiser.step()` to update the model weights. By default, our models use the `torch.optim.Adam` optimiser which implements an adaptive gradient descent strategy, additionally related to section 2.2.1.

Early stopping is also implemented during the training process, the specifics will be discussed in section 2.2.3.

1.4 The Regressor Architecture

Our `Regressor` class extends `sklearn.base.BaseEstimator`, which allows for it to be used as a `sklearn` estimator in both performing grid search using `sklearn.model_selection.GridSearchCV` or cross-validation using `sklearn.model_selection.cross_val_score()`. The significance of this is shown when we talk about fine tuning our model.

The impact of implementing our `Regressor` in this way is that our model instance is not created in the constructor as it was initially, but rather, in the `fit()` function. This allows the `Regressor` to create a new model every time it is fitted to data, which allows `GridSearchCV` to create new instances of the `Regressor` class by cloning another instance and changing the parameters. Otherwise the old model would also be copied over.

2 Evaluation & Results

2.1 Evaluating a Model

To evaluate the performance of a particular approach, we perform 5 fold cross-validation using `sklearn`'s `model_selection.cross_val_score()`. To evaluate the performance of a particular model, we call `score()` which in turn calls `predict()` which performs a single forward pass through the neural network and then calculates the RMSE of the result.

2.2 Methods to Prevent Overfitting

During the initial phases of our implementation, we saw a high performance from our model on the training data set, yet a very low performance (high RMSE value) on the test data set. This is emblematic of overfitting to the training data, thus we implemented measures to prevent overfitting and allow the model to generalise better to unseen data.

2.2.1 Regularisation

The use of `torch.optim.Adam` as our optimiser by default implements L2 regularisation to punish large weights. This was one of the reasons we decided to apply `MinMaxScaler` to our `y` values within the preprocessor, as to avoid having large weights on the output layer.

2.2.2 Dropout

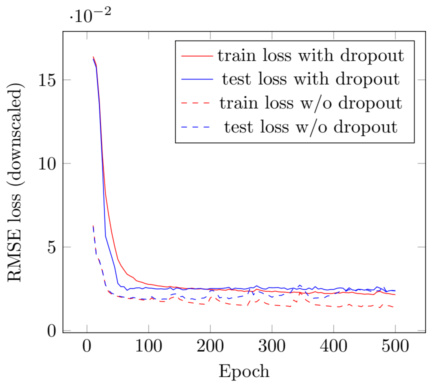

Figure 1: Effect of dropout on training and testing loss

We added this as an option the `Regressor`, toggled by the `dropout` parameter. Is not active by default for reasons we'll discuss in 2.2.3.

Dropout is implemented by passing the option as a `boolean` to the neural network model when calling the model's `forward()` function. This is only active during training, and applies `nn.Dropout` over each layer in the same manner as an activation function, causing 50% of outputs from a layer to be zeroed.

It can be observed from Figure 2 that the model without any significant measure to counteract overfitting begins to overfit to the training data, as indicated by the dashed lines becoming further apart. Conversely, it can also be observed that the performance of the model on both training and testing data remains fairly consistent with one another, though the training loss with dropout remains higher than without, whilst the loss on the test set without dropout increases past the test error with dropout.

2.2.3 Early Stopping

As seen in Figure 2, a standard model would overfit to the training data past a certain number of epochs, this results in the model performing badly on unseen data. Another means we have implemented to counteract this is early stopping.

This involves 3 variables passed to a `Regressor` instance: `early_stopping` - a `boolean` toggle for whether early stopping is enabled for the current `Regressor`, `patience` - the number of consecutive epochs without an improvement on the validation set before training is stopped, and `earliest_stop` - the earliest epoch at which we start considering early stopping.

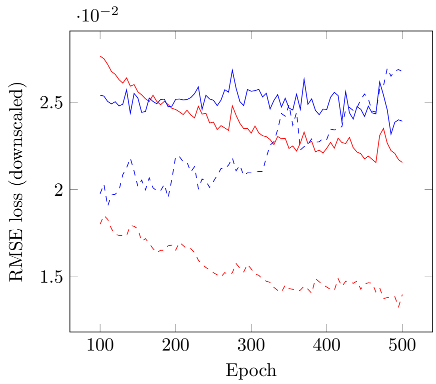

Figure 2: Final 400 Epochs

We first split the given training set into training and validation sets, at each epoch, the current model is trained on the training set, and evaluated on the validation set, if the performance on the validation set is better than some previous optimal performance, the current model is stored as the best model. Otherwise, it increments a number `strike` for the number of consecutive epochs without an improvement. If `strike` exceeds patience, then the training process is stopped, and the best model is returned.

In Figure 3 we can observe the training error, best validation error and the testing error. Note that there are sudden spikes in the values of the training error and testing error, this is likely a result of our use of `torch.optim.Adam`, which when the gradient becomes small, the result often spikes in this manner. A way we can circumvent its effects is to increase the patience of the model.

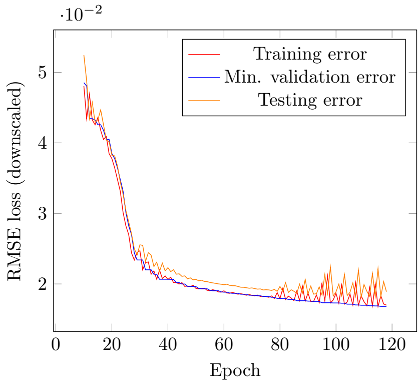

Figure 3: Effect of early stopping on train, test and validation loss

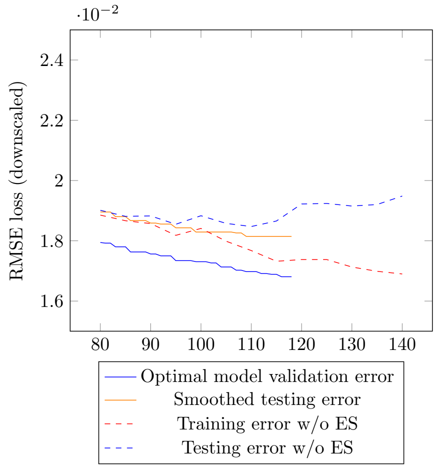

Figure 4: Smoothed results of early stopping with error values from another model without early stopping

For the sake of illustration, we've shown a smoothed version of both training and testing errors on Figure 4 along with the training and testing errors from another separately trained model, with its overfitting evident.

Through repeated cross-validation experiments, we found that effects of dropout and early stopping overlapped, in fact, it appeared that dropout slightly, though negatively, impacted the performance of early stopping, and thus we opted for early stopping to be enabled by default in lieu of dropout.

3 Hyperparameter Search

3.1 The overall search strategy

Now, there are three main sets of parameters that heavily affect a neural network's performance: the number of hidden layers, the neuron number of neurons in each layer(size of each layer), and the types of activations functions used in each layer. To obtain an optimal combination of these hyperparameters with a single set of experiments is not only computationally expensive, but also less robust. Instead, a search strategy is used, in which two major sets of experiments on two levels are conducted, one precedes the other. The first set will determine the optimal range of layer number. The second set then performs search within the optimal range of layer number, varying hyperparameters for each layer like the layer size and activations, to arrive at a best combination. For this experiment we did not run separate parameter setups for `torch.optim.SGD` optimisers, as its functionality was implemented in our choice of `torch.optim.Adam`, and we did not expect significantly different performance as a result.

Additionally for larger numbers of hidden layers, there were a significantly larger number of possible combinations of activation functions available, which would be computationally expensive and also bloat the number of samples at higher layer counts. Therefore we opted to select a few combinations for each layer count that were similar in structure across different setups to gain a general insight into the effect that the number of layers had on performance, and to narrow down a range of layer counts to explore further.

3.2 Search for optimal range of hidden layer number

Table 1 displays the results from the first set of experiments, where all the layers have the same neuron number, for a easier comparison on layer number. It can be seen that the lowerst RMSE value of all is highlighted red, this occurs at a hidden layer number of 4. Investigating further, one can conclude that RMSE values are on average lower with hidden layer number ranging from 2 to 4, RMSE values with hidden layer number beyond 5 increases considerably. Within the same layer number, neuron number and activations are also varied, however their effect is deemed less significant. One exception is at layer number 5, when the activations at last two layers are swapped, the RMSE values rise massively. The suspected reason for the high RMSE values at higher layers counts is attributed to the small delta values within the network, as a result, past a certain number of epochs the gradient between layers effectively become zero, causing the model to not update, thereby never converging towards a lower optimum value. Our solution to this issue was therefore to use lower layer counts.

3.3 A more detailed search into the network neuron configurations

The next set of experiments is focused on varying activations and neuron number at each layer more closely, under the same hidden layer number. Table 2 shows results for a constant hidden layer count of 3, while Table 3 shows results for layer number of 4. In both tables, the lowest RMSE values have been highlighted red. In the 3 layer case (Table 2), a neuron size combination of 512-512-128 is found to be optimal, with RMSE of 67602 using all `ReLU`, RMSE of 67443 using all `ReLU` except for the middle layer using `Sigmoid`. Both RMSE are among the lowest in the table, and the difference between the two is not significant. Thus it is concluded that for a 3-layer network, using a 512-512-128 configuration with at least `ReLU` at the first or final layer can yield the best model performance.

For the 4-layer case(Table 3), it can be seen that the 128-512-512-128 configuration yields the lowest RMSE. Furthermore, activations using `ReLU` are deemed optimal, and substituting `ReLU` with `Sigmoid` in the middle layers has devastating effect on the RMSE (shoots up from 68183 to 84121). Therefore 128-512-512-128 with all `ReLU` is the best combination.

Given all above, using a large neuron count like 512 at each layer is recommended, except for the beginning and ending layers the layer size should be tapered to yield the best performance. Activations at each layer should be kept constant, and `ReLU` is considered the best choice.

4 Final Evaluation

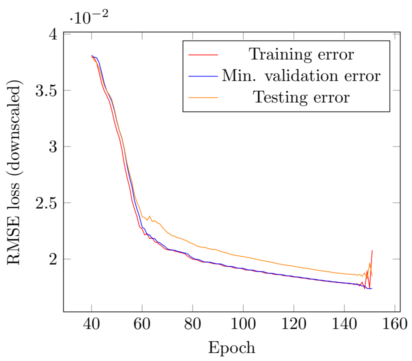

Figure 5: Performance of best model during training starting from epoch 40

As a result of our experimentation in section 3, we opted to use a 3 hidden layer neural network for our model, with 512-512-128 neuron configuration for the hidden layers. Our choice of activation functions was to use `ReLU` for all hidden layers, as it demonstrated the most consistently high performance across different configuration.

The model, using the methods detailed in the previous sections, performed comparatively well against our other models, with 65056 on the training data set, and 65056 on the test set, which was not used in the training of the model. This model has an RMSE error value of 65367 on LabTS.

Though this is still quite a high error rate considering the context of this task. It can be argued that perhaps a linear model is not a good fit for the relation between these features and the median house value. A potential future point of exploration is to add n-th polynomial terms to the features during preprocessing.

Appendices

A Results Tables

A.1

Varied hidden layer count, with drop-out and early-stopping							
Error in RMSE	**No. of activation layers**	**Layer size**	**Activation at each layer**				
			at layer 1	at layer 2	at layer 3	at layer 4	at layer 5
72018	1	32	Sigmoid	n/a	n/a	n/a	n/a
71710	1	64	Sigmoid	n/a	n/a	n/a	n/a
71012	2	32	Sigmoid	Tanh	n/a	n/a	n/a
71030	2	64	Sigmoid	Tanh	n/a	n/a	n/a
71688	3	13	Sigmoid	Tanh	Sigmoid	n/a	n/a
71081	3	32	Sigmoid	Tanh	Sigmoid	n/a	n/a
71352	3	32	Sigmoid	ReLU	Sigmoid	n/a	n/a
69988	4	13	Sigmoid	Tanh	ReLU	Sigmoid	n/a
72907	4	24	Sigmoid	Tanh	ReLU	Sigmoid	n/a
70948	4	32	Sigmoid	Tanh	ReLU	Sigmoid	n/a
73178	5	13	Sigmoid	Tanh	ReLU	ReLU	Sigmoid
77174	5	24	Sigmoid	Tanh	ReLU	ReLU	Sigmoid
75600	5	32	Sigmoid	Tanh	ReLU	ReLU	Sigmoid
115738	5	13	Sigmoid	Tanh	ReLU	Sigmoid	ReLU
70092	5	24	Sigmoid	Tanh	ReLU	Sigmoid	ReLU
115805	5	32	Sigmoid	Tanh	ReLU	Sigmoid	ReLU

Table 1: Performances under varied hidden layer count, with constant neuron(nodes) number per layer, with drop-out and early-stopping applied.

A.2

Hidden layer count=3, with drop-out and early-stopping						
Error in RMSE	Activation at each layer			number of nodes at each layer		
	at layer 1	at layer 2	at layer 3	at layer 1	at layer 2	at layer 3
74809	ReLU	ReLU	ReLU	13	13	13
70708	ReLU	ReLU	ReLU	32	64	32
69054	ReLU	ReLU	ReLU	64	128	64
67602	ReLU	ReLU	ReLU	512	512	128
67451	ReLU	ReLU	ReLU	512	512	512
77506	ReLU	Sigmoid	ReLU	13	13	13
70401	ReLU	Sigmoid	ReLU	32	64	32
68852	ReLU	Sigmoid	ReLU	64	128	64
67443	ReLU	Sigmoid	ReLU	512	512	128
68560	ReLU	Sigmoid	ReLU	512	512	512

Table 2: Performances under varied neuron configurations and activations, with a constant hidden layers number of 3.

A.3

Hidden layer count=4, with drop-out and early-stopping								
Error in RMSE	Activation at each layer				number of nodes at each layer			
	at layer 1	at layer 2	at layer 3	at layer 4	at layer 1	at layer 2	at layer 3	at layer 4
71129	ReLU	ReLU	ReLU	ReLU	64	128	128	64
68183	ReLU	ReLU	ReLU	ReLU	128	512	512	128
68874	ReLU	ReLU	ReLU	ReLU	256	512	512	256
72115	ReLU	Sigmoid	ReLU	ReLU	64	128	128	64
84121	ReLU	Sigmoid	ReLU	ReLU	128	512	512	128
69699	ReLU	Sigmoid	Sigmoid	ReLU	64	128	128	64
84121	ReLU	Sigmoid	Sigmoid	ReLU	128	512	512	128

Table 3: Performances under varied neuron configurations and activations, with a constant hidden layers number of 4.