70050 ExerciseTypes.CW2

Neural Networks

Submitters

cf23	Chen Fan
ky523	Kangle Yuan
glg23	Gordian Gruentuch
vm23	Vinayak Modi

good job, left some comments 98/100

Imperial College London

70050 Intro to Machine Learning

Coursework 2

Artificial Neural Networks

Authors:

Chen Fan

Yuan Kangle

Vinayak Modi

Gordian Grüntuch

November 26, 2023

1 Description of model, justification of choices

Our final model consists of a neural network with two hidden layers, with 512 neurons in each layer and a single output. The hyperparameters of the model were chosen through a series of tuning steps, see 3, that justified our final choices. The hyperparameters of the final model are presented in Table 1.

We took the following steps in our preprocessing: We replaced all missing numerical values with the median of the corresponding numerical columns, while we replace missing values in categorical columns with the most frequent value. In analyzing our dataset, we noticed significant outliers in the positive direction for each numerical feature. Consequently, we chose the median over the mean as the replacement value as it better represents the dataset's core distribution. We confirmed that the median outperforms the mean as a replacement value in section 3. The most frequent value is assumed to be the representative value for categorical data.

Min-Max Normalization was employed because the attributes have different scales and units. Normalization ensures that attributes contribute equally to the prediction, preventing features with larger scales from dominating the learning process. We stored the Min-max normalization constants from the training in order to normalize the validation data exactly the same.

Hyperparameter	**Value**	**Justification**
Batch Size	64	Peformed great in hyperparameter tuning; balances convergence and computational efficiency.
Network Architecture	[512, 512]	Two hidden layers with 512 units each, found to be perform best in the hyperparameter tuning; it was able to capture the complexity of the task without overfitting.
Optimizer	Adadelta	It dynamically adapts the learning rates for each parameter during training, eliminating the need for manually tuning the learning rate. Performed best in hyperparameter tuning.
Loss Function	MSE	Common choice recommended by theory. Also used for the grading of this coursework. For further explanations, see 2.
Ativation Functions	Relu	Unlike sigmoid or tanh functions, ReLU does not saturate for positive inpt values. Instead, it remains active, preventing the vanishing gradient problem and is thus suitable for regression problems.
Dropout implementation	0.1	Dropout was implemented in the neural network to avoid overfitting of the data. A dropout rate of 0.1 was selected after hyperparameter tuning to avoid overfitting.

Table 1: Choice of hyperparameters for Neural Network.

2 Description of the evaluation setup

Our `predict` method efficiently handles our input data by first applying preprocessing, and then generating predictions with our model inside a `with torch.no_grad():` block to improve efficiency. The use of `torch.no_grad()` is crucial as it disables gradient computation, which is not necessary during the prediction phase, thus significantly reducing memory usage and speeding up computations. We avoided Python loops by leveraging PyTorch's tensor operations for batch processing, which enables simultaneous handling of multiple data instances, significantly enhancing prediction speed and reducing computational overhead.

The `score` function in our model is designed to evaluate model accuracy on a validation dataset. It takes two pandas DataFrames, `x` (input data) and `y` (true output values), each with appropriate batch sizes. The function first generates predictions using the `predict` method, then converts these predictions and the true values to PyTorch tensors, and finally returns the Mean Square Error (MSE) between them, providing a quantifiable measure of the model's efficiency.

Our primary metric for evaluation was the Mean Square Error Loss. This metric is widely regarded as the standard for regression problems due to its effectiveness in quantifying the variance between predicted and actual values. The choice of MSE Loss is driven by its ability to penalize larger errors more severely, which is crucial in regression analyses.

Upon analyzing our dataset, we observed that both the numerical features and label data are not normally distributed and exhibit multiple outliers in the positive direction. This non-normal distribution led us to consider the log Mean Square Error (log MSE) as a potentially more suitable metric. Log MSE can be particularly advantageous in dealing with skewed data by reducing the disproportionate impact of outliers.

Another factor influencing our decision to use MSE Loss is its alignment with the performance assessment criteria of our coursework. Employing the same metric used for performance evaluation ensures consistency and relevance in our model assessment approach. Moreover, the square root of the MSE Loss, the Root Mean Square Error, provides an intuitive understanding of the error magnitude in the same units as the original data. This characteristic of RMSE makes it a valuable metric for a more comprehensible interpretation of the model's performance and thus we use it as in our reporting, but not in the actual training.

3 Hyperparameter tuning

We performed multiple extensive grid searches and a series of experiments to optimize our hyperparameters, which can be found in Table 1. In the following, we will focus on three hyperparameters in particular and illustrate our findings using figures. Other hyperparameters, such as number of neurons, optimizer class, or activation functions were similarly tuned, but will be omitted from the detailed discussion due to space limitations.

Using the median as a replacement for missing numerical values, seemed to yield a lower Validation Loss, as seen in Fig. 1. We performed a range of experiments comparing the two, and the absolute differences in the performance were consistent and small in all cases.

The impact of dropout rate on model performance is clearly demonstrated by Figure 2. The model utilizing a 0.1 dropout rate exhibits superior generalization, as evidenced by lower validation loss, in comparison to the models with dropout rates of 0.3 and 0. The latter, with a dropout rate of 0, clearly indicates overfitting, as evidenced by a significantly lower training loss yet higher validation loss. Conversely, the model with a 0.3 dropout rate shows both a higher validation loss and a higher testing loss, implying insufficient model complexity or inadequate learning. Thus, a dropout rate of 0.1 emerges as the optimal choice in balancing model fit and generalization capabilities.

Our experiments with varying the number of hidden layers in the neural network, as demonstrated in Figure 3, indicated that a configuration with 2 layers was the most effective. This setup not only achieved the lowest validation loss, suggesting optimal generalization, but it also proved to be the least computationally intensive, striking a balance between performance and efficiency.

what's the validation split? should mention it

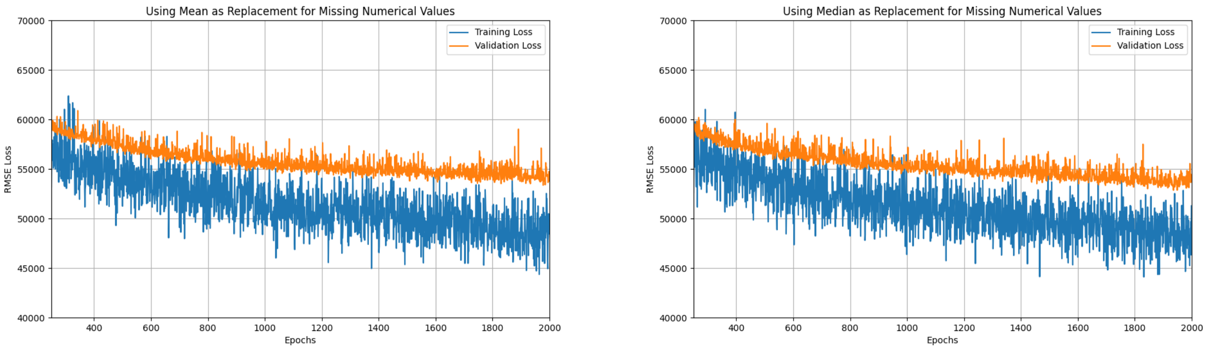

Figure 1: Training and Validation Loss Trends from 250 to 2000 Epochs using either the mean or the median as replacement for missing numerical feature values. For all other hyperparameters, the network is configured according to Table 1.

would be better to have them as a single plot to better capture the difference

same here

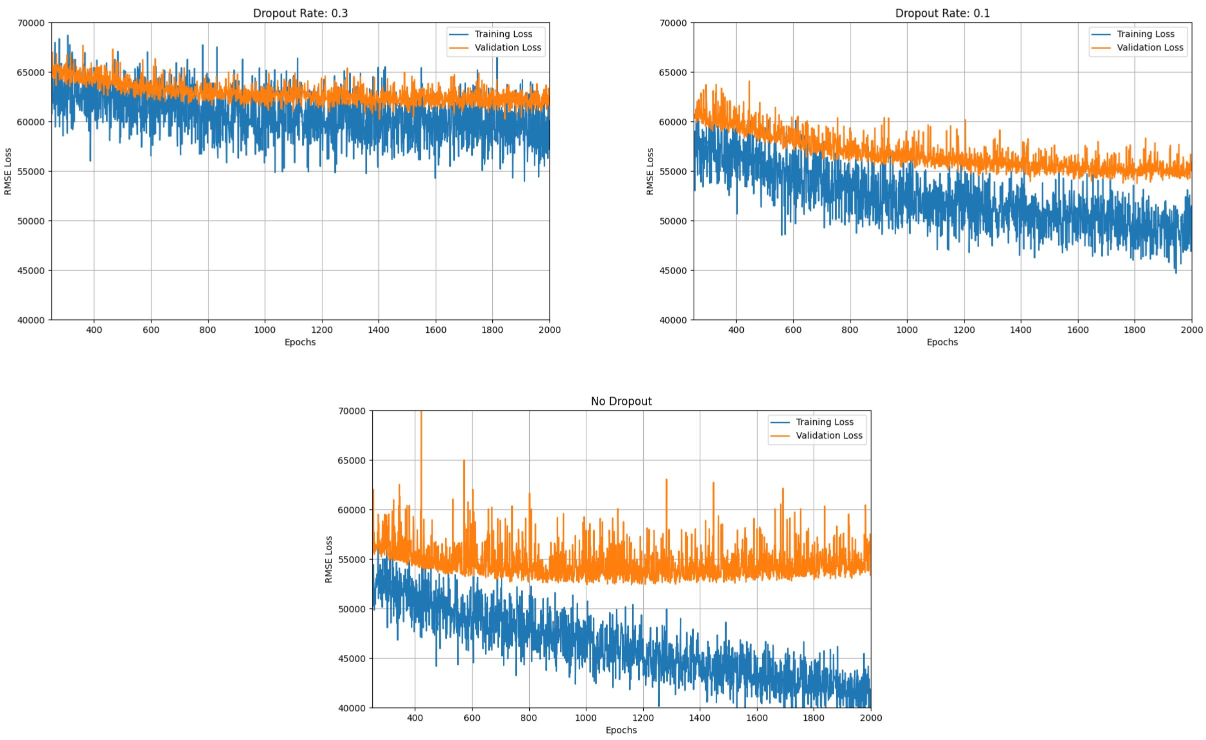

Figure 2: Training and Validation Loss Trends from 250 to 2000 Epochs Under Varying Dropout Rates. In this case, the network was configured with three hidden layers, each consisting of 128 neurons. Refer to Table 1 for configuration of all other hyperparameters.

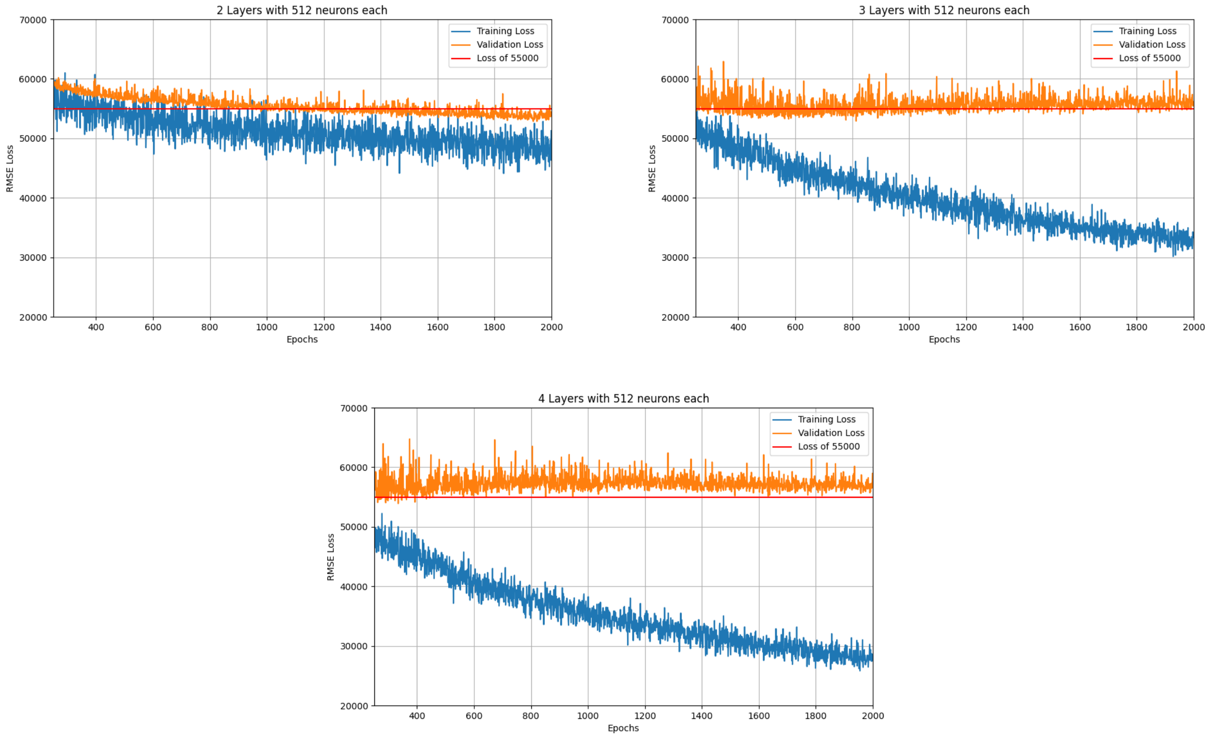

Figure 3: Training and Validation Loss Trends from 250 to 2000 Epochs Using Varying Number of Layers in the Network. The horizontal red line was added as a visual reference for a RMSE Loss level of 5500. Refer to Table 1 for configuration of all other hyperparameters.

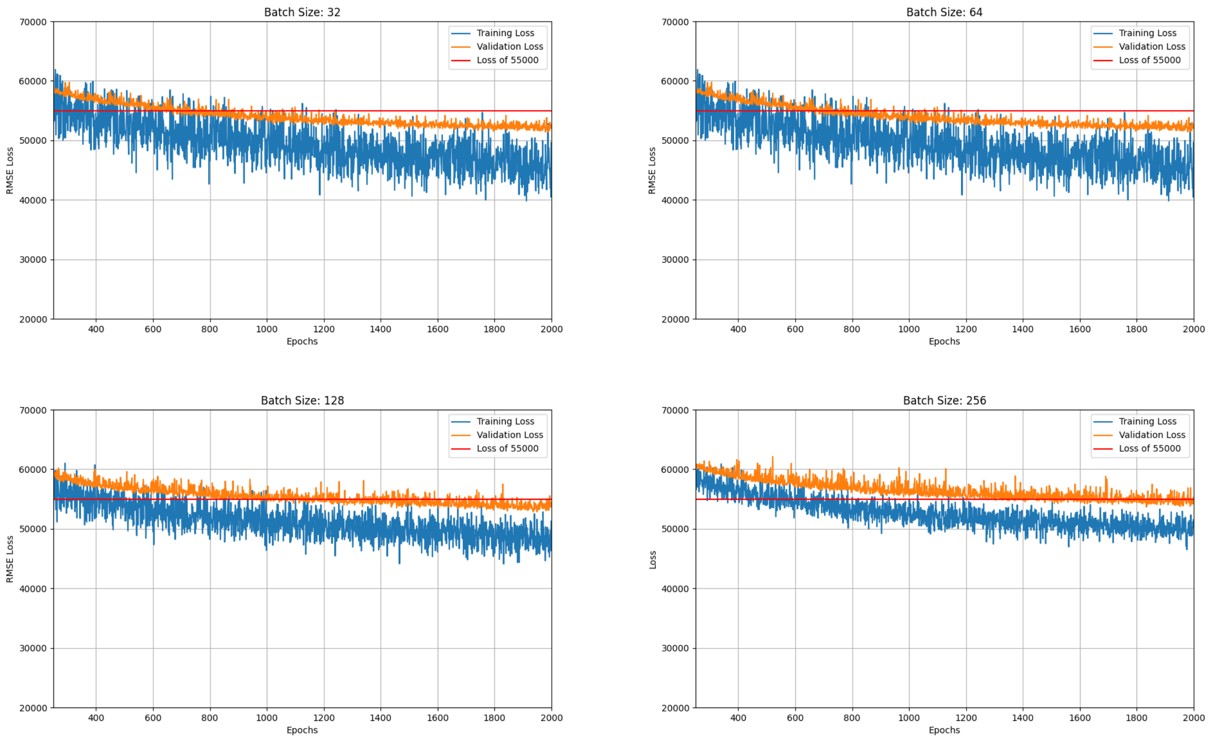

Figure 4: Training and Validation Loss Trends from 250 to 2000 Epochs Using Varying Batch Sizes. The horizontal red line was added as a visual reference for a RMSE Loss level of 5500. Refer to Table 1 for configuration of all other hyperparameters.

Upon careful evaluation of the training and validation loss trends depicted in Figure 4, our analysis spans across four distinct batch sizes: 32, 64, 128, and 256, over a range of 250 to 2000 epochs. An ideal batch size strikes a delicate balance between computational efficiency, which tends to increase with larger batch sizes due to enhanced parallel processing capabilities, and the stability and quality of the learning process, which can be favored by smaller batch sizes due to more frequent updates. While the batch size of 32 demonstrated the lowest validation loss, suggesting excellent generalization, it was batch size 64 that provided the best compromise. This batch size still benefited from relatively tight convergence and stable learning, as shown by the consistent loss trends, but with improved computational efficiency. The gains in processing speed without a significant penalty in performance metrics lead us to select a batch size of 64 as the most effective for our model, balancing computational demands with robust learning dynamics.

4 Final Evaluation of the Best Model

Our best model, fine-tuned as detailed in Section 3, features a 0.1 dropout rate, two hidden layers with 512 neurons each, and a 64 batch size, optimizing both computational efficiency and prediction accuracy. It achieved a competitive average RMSE of 49907.0352 on the test set, reflecting a solid fit given the dataset's complexity.

The final model's performance is not only a result of the optimal hyperparameter settings but also of the careful preprocessing of input data, feature engineering, and the selection of an appropriate loss function that guided the training process effectively.

Future work may involve exploring more sophisticated model architectures, including deep learning methods that can automatically extract high-level features from the data. Moreover, experimenting with different forms of regularization and loss functions could yield further improvements in model accuracy and generalization. Ensuring the model's interpretability, especially for high-stakes decision-making scenarios, remains a priority to facilitate trust and adoption of the model's predictions.

Furthermore, our reliance on the RMSE score as a performance metric may need reexamination. Given its tendency to overly penalize large errors, it could misrepresent the predictive accuracy for vastly differing house prices. For instance, a few $10,000 prediction error carries different significance for houses priced at $12,000 versus $400,000,000. The former indicates a severe model inaccuracy, whereas the latter could be deemed acceptable. Hence, an error metric like the logarithmic RMSE might be more appropriate, offering a more nuanced and interpretable measure that accounts for the relative scale of house prices.

In conclusion, the best model presents a significant improvement over baseline models and serves as a strong foundation for further research and practical applications. Its performance is a testament to the efficacy of systematic hyperparameter tuning and the thoughtful construction of neural network models.

```
Final Tests: Summary for vm23 of t5
-----------------------------------

  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:lab2324_autumn/Neural_Networks_70050_097.git
Commit ID: f8960
```

```python
import pandas
import torch
import pickle
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelBinarizer
import torch.nn as nn
from sklearn.metrics import mean_squared_error, r2_score, make_scorer
from sklearn.model_selection import GridSearchCV
#from skorch import NeuralNetRegressor
import torch.nn.functional as F
import torch.utils.data as u_data
from datetime import datetime

class CustomNet(nn.Module):
    """
    Custom torch module for better customisation on network layers
    """
    def __init__(self,
                 input_size,
                 hidden_layer_number = 1,
                 hidden_layer_feature = 64,
                 activation_function = nn.ReLU,
                 init_function = None,
                 apply_dropout = False,
                 dropout_rate = 0.2
                 ):
        """
        Construct the network and it’s linear and activation layers

        Args:
            input_size {int}: Number of attributes of input data
            hidden_layer_number {int}: number of hidden layers of the network
            hidden_layer_feature {int}: number of neurons of each hidden layer
            activation_function {function}: type of activation function to use
            init_function {function}: type of parameter initialization function to use
            apply_dropout {boolean}: apply dropout after each forward pass or not
            dropout_rate {float}: dropout rate if apply dropout
        """
        super().__init__()

        self.apply_dropout = apply_dropout

        # Use ModuleList allowing the network to be transferred to GPU
        # so I can train it with 500k epoch
        self.layers = nn.ModuleList()
        self.acts = nn.ModuleList()
        self.dropouts = nn.ModuleList()

        # Add input layer
        self.layers.append(nn.Linear(input_size, hidden_layer_feature))
        self.acts.append(activation_function())
        self.dropouts.append(nn.Dropout(dropout_rate))

        # Add hidden layers (if more than 1)
        if hidden_layer_number > 1:
            for i in range(hidden_layer_number - 1):
                self.layers.append(nn.Linear(hidden_layer_feature, hidden_layer_feature))
                self.acts.append(activation_function())
                self.dropouts.append(nn.Dropout(dropout_rate))

        # Add output layer
        self.output = nn.Linear(hidden_layer_feature, 1)
```



```python

        # If init function provided, fill in the init values
        if init_function:
            for layer in self.layers:
                init_function(layer.weight)
            init_function(self.output.weight)

    def forward(self, x):
        """
        Apply a forward pass to given data x

        Args:
            x {torch.tensor}: training data

        Returns:
            predictions from this pass
        """
        for layer, act, dropout in zip(self.layers, self.acts, self.dropouts):
            x = act(layer(x))
            # Apply dropout if we want to
            if self.apply_dropout:
                x = dropout(x)
        x = self.output(x)
        return x

class Regressor():

    def __init__(self, x, nb_epoch = 1000, apply_dropout = False, dropout_rate = 0.1):
        # You can add any input parameters you need
        # Remember to set them with a default value for LabTS tests
        """
        Initialise the model.

        Arguments:
            - x {pd.DataFrame} -- Raw input data of shape
                (batch_size, input_size), used to compute the size
                of the network.
            - nb_epoch {int} -- number of epochs to train the network.
            - apply_dropout {float} -- apply dropout after each forward pass or not
            - dropout_rate {float} -- dropout rate if apply dropout
        """

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

        # Prepare parameter dictionary for preprocessor
        self.pre_params = dict()

        # Fill the parameter dict and get input_size
        X, _ = self._preprocessor(x, training = True)

        # Fill other class fields
        self.input_size = X.shape[1]
        self.output_size = 1
        self.nb_epoch = nb_epoch

        # Creating custom network with the parameters
        self.net = CustomNet(
            self.input_size,
            2,
            512,
            apply_dropout= apply_dropout,
            dropout_rate= dropout_rate)
```

```python

        # Create a list to store loss over the training for plotting
        self.loss_data = []
        self.eval_data = []
        return

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

    def _preprocessor(self, x, y = None, training = False):
        """
        Preprocess input of the network.

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

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

        """

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

        # Reset the index so the LabelBinarizer concats correctly
        x.reset_index(drop= True, inplace=True)

        # Split numerical and categorical for processing
        numerical = x.select_dtypes(include=['float64', 'int64'])
        categorical = x.select_dtypes(include=['object', 'category'])

        # Fill the parameters dict to use in future non-training processing
        if training:
            self.pre_params['numerical_median'] = numerical.median()
            self.pre_params['categorical_most'] = []
            for column in categorical.columns:
                self.pre_params['categorical_most'].append(x[column].mode()[0])
            self.pre_params['numerical_min'] = numerical.min()
            self.pre_params['numerical_max'] = numerical.max()


        # filling missing numerical columns with median value
        numerical = numerical.fillna(self.pre_params['numerical_median'])

        # filling missing text columns with the most frequent value
        for column_index in range(len(categorical.columns)):
            column = categorical.columns[column_index]
            categorical[column] = categorical[column].fillna(
                self.pre_params['categorical_most'][column_index]
            )

        # binarize text column
        lb = LabelBinarizer()
        categorical_new = pd.DataFrame()
        for column in categorical.columns:
            # combines fit and transform into a single step
```



```python
            instance = lb.fit_transform(categorical[column])
            # covert from numpy to dataframe
            instance_df = pd.DataFrame(instance, columns=lb.classes_)
            # drop the original column
            categorical_new = pd.concat([categorical_new, instance_df], axis=1)

        # Normalize numerical value
        min_value = self.pre_params['numerical_min']
        max_value = self.pre_params['numerical_max']
        numerical = (numerical - min_value) / (max_value - min_value)

        # Concat the two parts back to data
        x = pd.concat([numerical, categorical_new], axis=1)

        # Convert to tensor
        x = torch.tensor(x.values, dtype=torch.float32)
        if y is None:
            return x, None
        y = torch.tensor(y.values, dtype=torch.float32)
        # Return preprocessed x and y, return None for y if it was None
        return x, y
        #######################################################################
        #                       ** END OF YOUR CODE **
        #######################################################################


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

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

        Returns:
            self {Regressor} -- Trained model.

        """

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

        # All mannual-set parameters here for easy modifying
        batch_size = 64
        learning_rate = 1

        # Preprocess
        X, Y = self._preprocessor(x, y = y, training = True) # Do not forget


        # !!!!!!!!! PICK ONE OF THE FOLLOWING
        # !!! USING GPU
        #X = X.cuda()
        #Y = Y.cuda()
        #net = self.net.cuda()

        # !!! USING CPU
        net = self.net

        # use torch.utils to batch data
        dataset = u_data.TensorDataset(X, Y)
        training_loader = u_data.DataLoader(dataset, batch_size = batch_size, shuffle=True)

```

```python
        # choose loss function
        loss = nn.MSELoss()

        # choose optimiser
        optimiser = torch.optim.Adadelta(
            net.parameters(),
            lr = learning_rate,
            weight_decay= 0
        )

        # Train all epochs
        for i in range(self.nb_epoch):
            print("Epoch {}:".format(i+1))
            running_loss = 0.
            last_loss = 0.

            # Train per epoch
            for batch_index, data in enumerate(training_loader):
                inputs, labels = data
                optimiser.zero_grad()
                pred_y = net(inputs)
                l = torch.sqrt(loss(pred_y, labels))
                l.backward()
                optimiser.step()

                # Print out the metrics
                running_loss += l.item()
                if batch_index % 10 == 9:
                    last_loss = running_loss / 10
                    print("batch {} loss: {}".format(batch_index + 1, last_loss))
                    running_loss = 0

            # Keep record of loss/eval data for plots
            self.loss_data.append(last_loss)
            if not (x_eval is None):
                score_epoch = self.score(x_eval, y_eval)
                self.eval_data.append(score_epoch)
                print("Average Loss on validation of epoch: {}".format(score_epoch))
            print("Average Loss of epoch: {}".format(last_loss))
        return self

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


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

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

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

        """

        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        X, _ = self._preprocessor(x, training=False)  # Do not forget

        # !!!!!!!!!!! COMMENT THIS IF NOT USING GPU
```



```python
        # !!! USING GPU
        #X = X.cuda()

        # disable gradient calculations.
        # In prediction mode, you don't need to compute gradients,
        # which are only necessary during training for backpropagation
        with torch.no_grad():
            predictions = self.net(X)

        # !!! AND THIS .cpu()
        #return predictions.cpu().numpy()
        return predictions.numpy()

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

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

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

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

        """

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

        pred_y = self.predict(x)
        pred_y = torch.from_numpy(pred_y)
        Y = torch.tensor(y.values, dtype=torch.float32)
        return rmse(pred_y, Y)

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


def save_regressor(trained_model, label):
    """
    Utility function to save the trained regressor model in part2_model.pickle.
    With extra label to identify them

    Args:
        trained_model {Regressor}: trained model
        label {str}: 'final' if the model is used for submission,
                     anything else to add a label to identify the models
    """
    # If you alter this, make sure it works in tandem with load_regressor
    if label == 'final':
        label = ''
    else:
        label = '_' + label
    with open(f'part2_model{label}.pickle', 'wb') as target:
        pickle.dump(trained_model, target)
    print(f"\nSaved model in part2_model{label}.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 save_loss_data(loss_data, time_str):
    """
    Utility function to save the loss data to a csv file with timestamp

    Args:
        loss_data {list}: List of loss during training per epoch
        time_str {string}: String of timestamp in the form of DD_HHMMSS
    """
    np.savetxt("loss_{}.csv".format(time_str),
               loss_data,
               delimiter=',',
               fmt='%f')
    print("\nSaved Loss data in loss_{}.csv".format(time_str))

def save_eval_data(eval_data, time_str):
    """
    Utility function to save the evaluation loss data to a csv file with timestamp

    Args:
        loss_data {list}: List of loss evaluated using an evaluation set
                          during training per epoch
        time_str {string}: String of timestamp in the form of DD_HHMMSS
    """
    np.savetxt("eval_{}.csv".format(time_str),
               eval_data,
               delimiter=',',
               fmt='%f')
    print("\nSaved Loss data in eval_{}.csv".format(time_str))


def rmse(pred_y, y):
    """
    Using rmse to get the error.

    Args:
        pred_y {torch.tensor}: Predicted label from the input
        y {torch.tensor}: True label of the input

    Returns:
        {float}: rmse score of this prediction (lower is better)
    """
    mse_loss = F.mse_loss(pred_y, y)

    # Calculate RMSE
    rmse = torch.sqrt(mse_loss)

    return rmse

def log_rmse(pred_y, y):
    """
    Use rmse for the logged value to get a better insight into relative error

    Args:
        pred_y {torch.tensor}: Predicted label from the input
        y {torch.tensor}: True label of the input

```



```
    Returns:
        {float}: log_rmse score of this prediction (lower is better)

    """
    # Log_rmse is currently broken so we're back to just rmse

    # Clamp the values to 1 to furthur stablise the result
    clamp_y = torch.clamp(pred_y, 1, float('inf'))

    # Calculating the log_rmse
    l = torch.sqrt(torch.mean((torch.log(clamp_y) - torch.log(y))**2))
    return l

def RegressorHyperParameterSearch(x, y):
    """
    Performs a hyper-parameter for fine-tuning the regressor implemented
    in the Regressor class.

    Lines using custom packages are commented out to enable the project
    to be tested on LabTS.

    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:
        The function should return your optimised hyper-parameters.

    """

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

    # Create a dummy model so we can use the preprocessor
    dummy_regressor = Regressor(x)
    X, Y = dummy_regressor._preprocessor(x=x, y=y, training=True)

    # Convert our custom torch model to sklearn model using skorch
    #model = NeuralNetRegressor(
    #    module=CustomNet,
    #    optimizer= torch.optim.Adam,
    #    max_epochs= 300
    #)

    # Define our parameter table for small-scaled tuning automation
    parameters_to_tune = {
        # this is fixed
        'module__input_size': [X.shape[1]],
        'module__hidden_layer_number' : [1],
        #'module__hidden_layer_number' : [1, 2, 3],
        'module__hidden_layer_feature' : [512],
        #'module__hidden_layer_feature' : [64, 128, 256, 512, 1024],
        'module__activation_function': [nn.ReLU],
        #'module__activation_function' : [nn.ReLU, nn.ReLU6, nn.Sigmoid, nn.Tanh],
        'module__init_function': [nn.init.normal_],
        #'module__init_function' : [nn.init.uniform_,
        #                           nn.init.xavier_uniform_,
        #                           nn.init.normal_,
        #                           nn.init.zeros_,
        #                           nn.init.kaiming_normal_,
        #                           nn.init.kaiming_uniform_,
        #                           nn.init.xavier_normal_],
        'optimizer': [torch.optim.Adadelta],
```

```python
        #'optimizer': [torch.optim.Adam,
        #              torch.optim.Adadelta,
        #              torch.optim.Adagrad,
        #              torch.optim.AdamW,
        #              torch.optim.Adamax,
        #              torch.optim.NAdam,
        #              torch.optim.RMSprop
        #              ],
        'optimizer__lr': [250],
        #'optimizer__lr': [100, 300, 500, 700],
        'optimizer__weight_decay': [0],
        'max_epochs': [1000],
        'module__apply_dropout': [True],
        #'module__apply_dropout': [True, False],
        'module__dropout_rate': [0.1],
        #'module__dropout_rate': [0.1, 0.2, 0.3, 0.4],
        'batch_size': [128]
        #'batch_size': [16, 32, 64, 128, 256]
    }

    # Perform the tuning
    # change n_jobs to less if the cpu is <16 cores
    #grid = GridSearchCV(
    #    estimator=model,
    #    param_grid=parameters_to_tune,
    #    scoring='neg_root_mean_squared_error',
    #    n_jobs=16,
    #    verbose=2
    #)
    #grid_result = grid.fit(X, Y)

    # Retrive the results
    #print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
    #means = grid_result.cv_results_['mean_test_score']
    #stds = grid_result.cv_results_['std_test_score']
    #params = grid_result.cv_results_['params']
    #for mean, stdev, param in zip(means, stds, params):
    #    print("%f (%f) with: %r" % (mean, stdev, param))

    return  #grid_result.best_params_ # Return the chosen hyper parameters

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


def training_main():
    """
    Primary function used to run the training and tuning

    """

    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")

    # Splitting training and evaluation
    data_train, data_evaluation = train_test_split(
        data,
        test_size=0.33,
        random_state=70050)

```



```python
    # Splitting input and output
    x_train = data_train.loc[:, data.columns != output_label]
    y_train = data_train.loc[:, [output_label]]

    x_eval = data_evaluation.loc[:, data.columns != output_label]
    y_eval = data_evaluation.loc[:, [output_label]]

    # Create our model
    regressor = Regressor(
        x_train,
        nb_epoch = 5000,
        apply_dropout= True,
        dropout_rate= 0.1
    )

    # Training
    regressor.fit(x_train, y_train, x_eval, y_eval)

    # Save model and data
    save_regressor(regressor, 'final')
    time_str = datetime.now().strftime('%d_%H%M%S')
    save_loss_data(regressor.loss_data, time_str)
    save_eval_data(regressor.eval_data, time_str)

    # An intuitive view of performance
    x_pred = regressor.predict(x_eval)
    print(x_pred[0:10])
    print(y_eval[0:10])
    print(x_pred[-10:])
    print(y_eval[-10:])

    # Error
    error = regressor.score(x_eval, y_eval)
    print("\nRegressor error: {}\n".format(error))

    #RegressorHyperParameterSearch(x_train, y_train)

def transfer_trained_to_cpu():
    '''
    Transfer the net back to cpu so it can pass the LabTS tests
    '''
    my_model = load_regressor()
    my_model.net = my_model.net.to('cpu')
    save_regressor(my_model, 'final')

def final_eval():
    """
    Print the final evaluation score to use in report

    """
    output_label = "median_house_value"
    data = pd.read_csv("housing.csv")

    # Splitting training and evaluation
    data_train, data_evaluation = train_test_split(
        data,
        test_size=0.33,
        random_state=70050)

    x_eval = data_evaluation.loc[:, data.columns != output_label]
    y_eval = data_evaluation.loc[:, [output_label]]

    my_model = load_regressor()

    # disable the dropout layers so it will get the fixed result
    my_model.net.eval()
```

```python

    final_score = my_model.score(x_eval, y_eval)
    print(final_score)

if __name__ == "__main__":
    training_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 **
        #######################################################################
        self._cache_current = x
        layer_output = 1 / (1 + np.exp(-x))
        return layer_output



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



```python

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

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

        Returns:
            {np.ndarray} -- Array containing gradient with respect to layer
                input, of shape (batch_size, n_in).
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        x = self._cache_current
        sigmoid_x = 1 / (1 + np.exp(-x))
        return grad_z * (sigmoid_x * (1 - sigmoid_x))

        #######################################################################
        #                       ** 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
        relu = np.maximum(0, x)
        return relu

        #######################################################################
        #                       ** 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 respect to layer
                input, of shape (batch_size, n_in).
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        x = self._cache_current
        deriv_x = np.where(x > 0, 1, 0)
        return grad_z * deriv_x

        #######################################################################
        #                       ** 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)) # a 2D array
        self._b = np.zeros(n_out)  # a 1D array

        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 **
        #######################################################################
        y_predicted = x @ self._W + self._b
        self._cache_current = x

        return y_predicted
        #######################################################################
        #                       ** 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 respect to layer
                input, of shape (batch_size, n_in).
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        x = self._cache_current
        self._grad_W_current = x.T @ (grad_z)
        self._grad_b_current = np.sum(grad_z, axis=0)
        grad_x_current = grad_z @ self._W.T

        return grad_x_current

        #######################################################################
        #                       ** 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):
        """
```

```python
        Constructor of the multi layer network.

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


        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        self.input_dim = input_dim
        self.neurons = neurons
        self.activations = activations
        self._layers = []


        ##################
        # Create all layers
        ##################

        n_of_input_neurons = input_dim

        # Iterate over each layer
        for n_of_output_neurons, activation_type in zip(neurons, activations):

            # Create Linear Layer
            current_linear_layer = LinearLayer(n_of_input_neurons, n_of_output_neurons)
            self._layers.append(current_linear_layer)

            # Create corresponding Activation Layer
            current_activation_layer = self.get_activation_layer(activation_type)

            # In case of Sigmoid or Relu, we add an activation layer
            # In case of Identity Layer, we add no further layers
            if not current_activation_layer is None:
                self._layers.append(current_activation_layer)

            n_of_input_neurons = n_of_output_neurons


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

    def get_activation_layer(self, activation_type):
        if activation_type == "relu":
            return ReluLayer()
        if activation_type == "sigmoid":
            return SigmoidLayer()
        if activation_type == "identity":
            return None

        error_message = (
            f"The activation function '{activation_type}'"
            " cannot be implemented."
            "The only accepted activation functions are:"
            "'relu', 'sigmoid', and 'identity."
                        )

```



```python
        raise ValueError(error_message)

    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 **
        #######################################################################

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


        return x

        #######################################################################
        #                       ** 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 (batch_size,
                #_neurons_in_final_layer).

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

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

        return grad_z

        #######################################################################
        #                       ** 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.
        """
        #######################################################################
```

```python
        #                        ** 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,
                cross_entropy.
            - shuffle_flag {bool} -- If True, training data is shuffled before
                training.
        """
        self.network = network
        self.batch_size = batch_size
        self.nb_epoch = nb_epoch
        self.learning_rate = learning_rate
        self.loss_fun = loss_fun
        self.shuffle_flag = shuffle_flag

        #######################################################################
        #                        ** START OF YOUR CODE **
        #######################################################################
        if loss_fun == "mse":
            self._loss_layer = MSELossLayer()
```



```python
        elif loss_fun == "cross_entropy":
            self._loss_layer = CrossEntropyLossLayer()
        else:
            error_message = (f"The loss function '{loss_fun}' cannot be"
                             " implemented. The only accepted loss functions "
                             " are 'mse', and 'cross_entropy.")
            raise ValueError( error_message )

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

        Returns:
            - {np.ndarray} -- shuffled inputs.
            - {np.ndarray} -- shuffled_targets.
        """
        #######################################################################
        #                        ** START OF YOUR CODE **
        #######################################################################
        assert len(input_dataset) == len(target_dataset)

        shuffled_indices = np.random.permutation(len(input_dataset))
        shuffled_input_dataset = input_dataset[shuffled_indices]
        shuffled_target_dataset = target_dataset[shuffled_indices]

        return shuffled_input_dataset, shuffled_target_dataset

        #######################################################################
        #                        ** 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 epoch in range(self.nb_epoch):
```

```
            if self.shuffle:

                input_dataset, target_dataset = self.shuffle(
                    input_dataset, target_dataset)


            # Split the dataset into mini-batches of size 'batch_size'.
            for i in range(0, len(input_dataset), self.batch_size):
                input_batch = input_dataset[i : i + self.batch_size]
                target_batch = target_dataset[i : i + self.batch_size]

                # Perform forward pass through the network given the current
                # batch of inputs.
                # Input_batch has size (batch_size, n_features)
                # Output has size (batch_size, #output_neurons)
                output = self.network.forward(input_batch)

                # Compute loss
                self._loss_layer.forward(output, target_batch)

                # Backward pass: Compute gradients
                grad_z = self._loss_layer.backward()
                self.network.backward(grad_z)

                # Update network parameters
                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. Returns
        scalar value.

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

        Returns:
            a scalar value -- the loss
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        # Forward pass through the network
        network_output = self.network.forward(input_dataset)

        # Calculate the loss
        loss = self._loss_layer.forward(network_output, target_dataset)

        return loss
        #######################################################################
        #                       ** 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.data = data

        # Needed for min-max-scaling
        self.min = None
        self.max = None

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

    def fit(self, data):
        """
        Compute the min and max values for each feature in the dataset.

        Arguments:
            data {np.ndarray} -- dataset to fit the preprocessor.
        """
        self.min = data.min(axis=0)
        self.max = data.max(axis=0)


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

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

        Returns:
            {np.ndarray} normalized dataset.
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################
        if self.min is None or self.max is None:
            self.fit(data)

        # Perform min-max-scaling
        return (data - self.min) / (self.max - self.min)

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


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

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

```

```
        Returns:
            {np.ndarray} reverted dataset.
        """
        #######################################################################
        #                       ** START OF YOUR CODE **
        #######################################################################

        return data * (self.max - self.min) + 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:

Epoch 1:
Average Loss of epoch: 0.0
Epoch 2:
Average Loss of epoch: 0.0

Loaded model in part2_model.pickle


Expected RMSE error on the test data: 90000
Obtained RMSE error on the test data: 52222.72265625
Succesfully reached the minimum performance threshold. Well done!

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

```