Notebook-2: Electricity Price Prediction - Window Size 5

by Qianjing Liang (8434350), Daniel Barbie (4939697), Jacob Umland (8436406), Fan Jia (8436217), Jan Faulstich (8439328)

1.Preparations

Firstly, we downloaded the data, imported necessary packages and loaded the data into DataFrames. We have also created a dataframe for summarizing the models' performance.

In [1]:
```
pip install tensorflow-addons
```

```
Requirement already satisfied: tensorflow-addons in /usr/local/lib/python3.7/dist-pa
ckages (0.14.0)
Requirement already satisfied: typeguard>=2.7 in /usr/local/lib/python3.7/dist-packa
ges (from tensorflow-addons) (2.7.1)
```

In [2]:
```
from google.colab import drive
drive.mount('/content/gdrive')
%cd /content/gdrive/MyDrive/Electricity_Data
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call driv
e.mount("/content/gdrive", force_remount=True).
/content/gdrive/MyDrive/Electricity_Data
```

In [3]:
```
#Imports basic packages for our tasks
import pandas as pd
import numpy as np
import pickle
from math import sqrt

#Imports visualization packages
import matplotlib.pyplot as plt
import seaborn as sns

#Import ML packages
from sklearn.linear_model import LinearRegression
from sklearn.dummy import DummyRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.svm import SVR
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_absolute_error, r2_score, make_scorer, mean_squared
from sklearn.preprocessing import MinMaxScaler, PolynomialFeatures
from sklearn.model_selection import GridSearchCV, cross_val_score
import lightgbm as lgb

#Import DL packages
import tensorflow as tf
import tensorflow.keras as k
from keras.models import *
from keras.layers import merge
from keras.layers.core import *
from tensorflow.keras import Model, Sequential
from tensorflow.keras import backend as be
from tensorflow.keras.layers import *
from tensorflow.keras.layers import Dense, Dropout, Input, Conv1D, MaxPool1D, Embedd
from tensorflow.keras.callbacks import ModelCheckpoint, LearningRateScheduler, Early
from tensorflow.keras.optimizers import Adadelta, Adam, RMSprop
from tensorflow.keras.regularizers import *
from tensorflow.math import exp
import tensorflow_addons as tfa

import warnings
warnings.filterwarnings('ignore')
```

```
/usr/local/lib/python3.7/dist-packages/tensorflow_addons/utils/ensure_tf_install.py:
67: UserWarning: Tensorflow Addons supports using Python ops for all Tensorflow vers
ions above or equal to 2.4.0 and strictly below 2.7.0 (nightly versions are not supp
orted).
 The versions of TensorFlow you are currently using is 2.7.0 and is not supported.
Some things might work, some things might not.
If you were to encounter a bug, do not file an issue.
If you want to make sure you're using a tested and supported configuration, either c
hange the TensorFlow version or the TensorFlow Addons's version.
You can find the compatibility matrix in TensorFlow Addon's readme:
https://github.com/tensorflow/addons
  UserWarning,
```

In [4]:
```
#Load the datasets with a sliding window size of 5
X_train_window_size_5 = pd.read_csv('X_train_window_size_5.csv')
X_valid_window_size_5 = pd.read_csv('X_valid_window_size_5.csv')
X_test_window_size_5 = pd.read_csv('X_test_window_size_5.csv')
X_train_window_size_5_tree = pd.read_csv('X_train_window_size_5_tree.csv')
X_valid_window_size_5_tree = pd.read_csv('X_valid_window_size_5_tree.csv')
X_test_window_size_5_tree = pd.read_csv('X_test_window_size_5_tree.csv')
y_train_window_size_5 = pd.read_csv('y_train_window_size_5.csv')
y_valid_window_size_5 = pd.read_csv('y_valid_window_size_5.csv')
y_test_window_size_5 = pd.read_csv('y_test_window_size_5.csv')
```

In [5]:
```
y_train_window_size_5 = y_train_window_size_5['y']
y_valid_window_size_5 = y_valid_window_size_5['y']
y_test_window_size_5 = y_test_window_size_5['y']
```

In [6]:
```
# Prepare a dataframe to store the performance of all models
performance_df = pd.DataFrame(columns=['Model', 'MAE score (on test set)', 'MSE scor
```

2.Conventional ML models

In the second part, conventional ML models are used to make predictions, serving as a baseline to be compared with DL approaches.

Evaluation Functions: MAE

In [7]:
```
def evaluate_model(model, X_test, y_test_true):
    predictions = model.predict(X_test)
    mae = mean_absolute_error(y_test_true, predictions)
    mse = mean_squared_error(y_test_true, predictions)
    print("Mean absolut error on test:", mae)
    print("Mean squared error on test:", mse)
    return mae, mse
```

In [8]:

```python
def evaluate_3Dmodel(model, X_test, y_test_true):
    predictions = model.predict(X_test)
    predictions = predictions[:, -1]
    mae = mean_absolute_error(y_test_true, predictions)
    mse = mean_squared_error(y_test_true, predictions)
    print("Mean absolut error on test:", mae)
    return mae, mse
```

2.1.Dummy regressor

Takeaways:

In [9]:

```python
# Modeling
dummy_model = DummyRegressor()
dummy_model.fit(X_train_window_size_5, y_train_window_size_5)

# Evaluate model performance and store it
mae, mse = evaluate_model(dummy_model, X_test_window_size_5, y_test_window_size_5)
performance_df = performance_df.append({'Model': 'Dummy Regressor',
                                        'MAE score (on test set)': round(mae, 4),
                                        'MSE score (on test set)': round(mse, 4)},
                                       ignore_index=True)
```

```
Mean absolut error on test: 2.258751650277334
Mean squared error on test: 21.15889779728792
```

2.2.Linear Regression

As expected, the linear models performed poorly for our data, as indicated by the R2 scores for both degree 1 and degree 2 models. The near 0 R2 scores for both validation and testing suggest that dependent variable cannot be explained by our data. Even though in the plots, the predicted y basically covered the actual y, the non-linear nature of our data cannot be captured by linear models. Due to the computational limitation, we only include the degree 2 polynomial regression, which has already served our purpose.

Helper functions

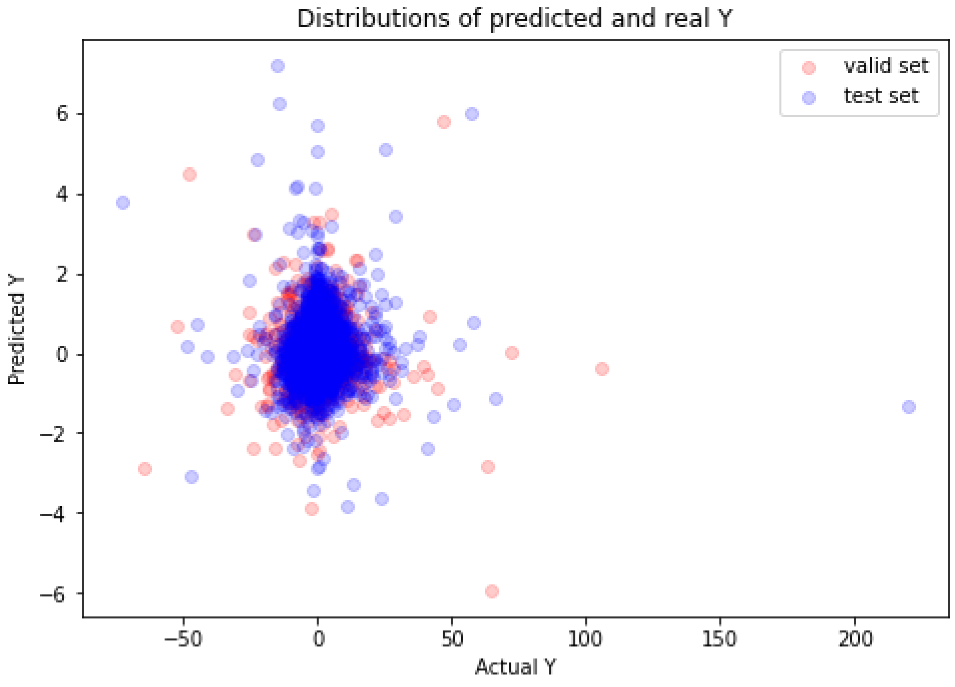
In [10]:

```python
# Modify the helper functions specifically for the linear models
# Function to perform Cross Validation on Linear Models
def get_cv_scores(model, X_train, y_train):
    scores = cross_val_score(model, X_train, y_train, cv=5, scoring='r2')
    print('CV Mean of R2: ', np.mean(scores))
    print('CV STD of R2: ', np.std(scores))

# Function to evaluate Linear Models with plots
def evaluate_valid(model, X_valid, y_valid, X_test, y_test):
    predictions_valid = model.predict(X_valid)
    mae_valid = mean_absolute_error(y_valid, predictions_valid)
    r2_valid = r2_score(y_valid, predictions_valid)
    print("Validation MAE:", mae_valid)
    print("Validation R2:", r2_valid)

    predictions_test = model.predict(X_test)
    mae_test = mean_absolute_error(y_test, predictions_test)
    mse_test = mean_squared_error(y_test, predictions_test)
    r2_test = r2_score(y_test, predictions_test)
    print("Test MAE:", mae_test)
    print("Test R2:", r2_test)

    fig=plt.figure()
    ax=fig.add_axes([0,0,1,1])
    ax.scatter(y_valid, predictions_valid, color='r', label='valid set', alpha=0.2)
    ax.scatter(y_test, predictions_test, color='b', label='test set', alpha=0.2)
    ax.set_title('Distributions of predicted and real Y')
    ax.set_xlabel('Actual Y')
    ax.set_ylabel('Predicted Y')
    ax.legend()

    return mae_test, mse_test
```

In [11]:

```
# Direct Linear Modling on Original Data
model_lr = LinearRegression(copy_X=True, n_jobs=100)
model_lr.fit(X_train_window_size_5, y_train_window_size_5)

# Evaluate model performance and store it
get_cv_scores(model_lr, X_train_window_size_5, y_train_window_size_5)
mae, mse = evaluate_valid(model_lr, X_valid_window_size_5, y_valid_window_size_5, X_

performance_df = performance_df.append({'Model': 'Linear Regression',
                                        'MAE score (on test set)': round(mae, 4),
                                        'MSE score (on test set)': round(mse, 4)},
                                       ignore_index=True)
```

```
CV Mean of R2:  -0.0024990875674780976
CV STD of R2:  0.011901930927789132
Validation MAE: 2.5597569293787914
Validation R2: -0.007536007631969 71
Test MAE: 2.278277126690864
Test R2: -0.004706594876143555
```

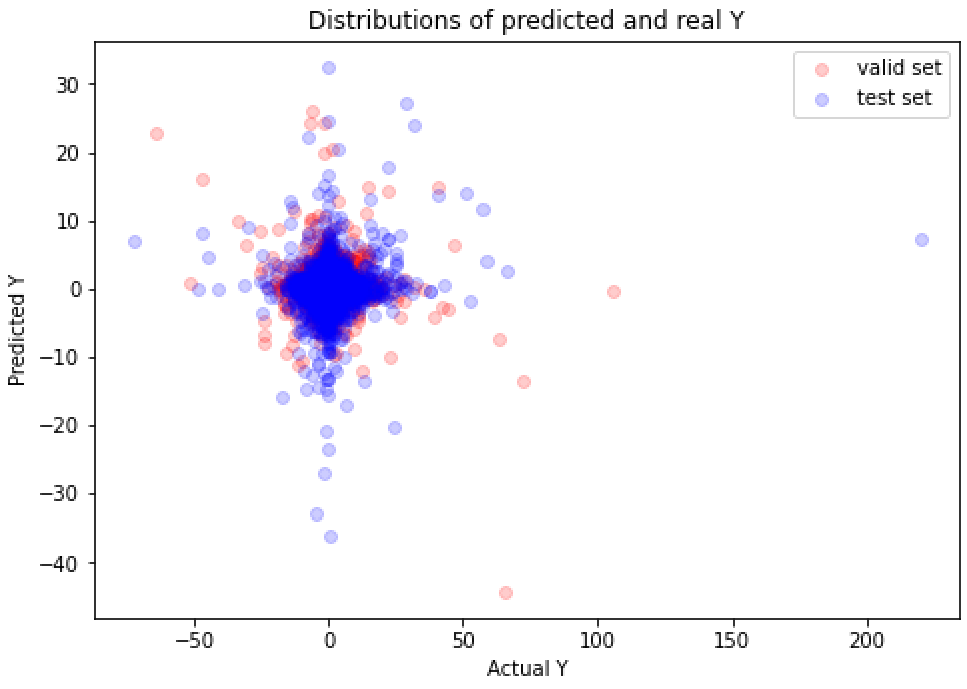


In [12]:

```
# Polynomial Regression (Degree=2)
# Transformation of X_train/valid/test datasets
polynomial_features= PolynomialFeatures(degree=2)
X_train_5_poly = polynomial_features.fit_transform(X_train_window_size_5)
X_valid_5_poly = polynomial_features.fit_transform(X_valid_window_size_5)
X_test_5_poly = polynomial_features.fit_transform(X_test_window_size_5)
```

```
# Modeling
model_lr = LinearRegression(normalize=True, copy_X=True, n_jobs=100)
model_lr.fit(X_train_5_poly, y_train_window_size_5)
get_cv_scores(model_lr, X_train_5_poly, y_train_window_size_5)

# Evaluate model performance and store it
mae, mse = evaluate_valid(model_lr, X_valid_5_poly, y_valid_window_size_5, X_test_5_

performance_df = performance_df.append({'Model': 'Linear Regression (poly transforme
                                        'MAE score (on test set)': round(mae, 4),
                                        'MSE score (on test set)': round(mse, 4)},
                                       ignore_index=True)
```

```
CV Mean of R2:  -1.6558473392156838e+16
CV STD of R2:  3.218491706856336e+16
Validation MAE: 2.811697086257412
Validation R2: -0.23269828315318963
Test MAE: 2.484765136958238
Test R2: -0.07257177828641947
```



2.3.Random Forrest

- **Intro**:

  The Random Forest method fits on the training data using an ensemble of decision trees. An important feature of the model is that it helps avoid overfitting by building each of the tree with a sample of data points as well as a sample of the features of the original training dataset.

- **Input data**:

  > Training dataset: `X_train_window_size_5_tree` and `y_train_window_size_5` \ Validation dataset: `X_valid_window_size_5_tree` and `y_valid_window_size_5` \ Testing dataset: `X_test_window_size_5_tree` and `y_test_window_size_5`

Apart from normalization, the datasets used for Random Forest modelling have also been transformed in terms of their datetime columns, such as for the variable `lasttrade_weekday`. These columns originally consisted of two sets of variables to express a certain datetime point, which has been preprocessed and converted back to their original value.

- **Architecture**:

  After comparing the models' performance on the validation dataset using GridSearchCV, the best combination of parameters for the Random Forest Regressor is:

  | Parameter name | Value | |-------------------|--------| | n_estimators | 500 | | max_features | 'sqrt' | | min_samples_split | 10 | | max_depth | 6 |

It is worth noting that our group has also tried dropping features that are deemed the least important by the Random Forest model after performing the Grid Search, and there seemed to be little improvement in the model's performance, which is why this part has been excluded here.

In [13]:
```python
# Modeling
rfmodel = RandomForestRegressor(random_state=3315,
                                max_depth=6,
                                n_estimators=500,
                                max_features='sqrt',
                                min_samples_split=10,
                                criterion='mse',
                                n_jobs=-1)
rfmodel.fit(X_train_window_size_5_tree, y_train_window_size_5)

# Evaluate model performance and store it
mae, mse = evaluate_model(rfmodel, X_test_window_size_5_tree, y_test_window_size_5)

performance_df = performance_df.append({'Model': 'Random Forest',
                                        'MAE score (on test set)': round(mae, 4),
                                        'MSE score (on test set)': round(mse, 4)},
                                       ignore_index=True)
```

```
Mean absolut error on test: 2.2755474195799463
Mean squared error on test: 21.846742385558375
```

- **Brief summary**:

  > As shown in the cell above, the Random Forest model's performance on the test dataset is 2.2755 in terms of MAE.

  Limitations:

  While Random Forest takes advantage of the power of ensemble, which helps avoid overfitting, it simply aggregates all the individual decision trees by training them in parallel. In other words, the trees within the Random Forest model are independent from each other and thus can't learn from each other. To address this disadvantage, we have also tried ensemble of boosted trees as shown in the following section.

2.4.Gradient Boosting

Takeaways:

LightGBM is a gradient boosting framework using tree based approaches. Gradient boosting comes from the idea whether a weak learner can modified to become even better. Additionally, we can state that a Gradient Boosting algorithm consists of 3 elements: 1. a loss function, which needs to be optimized. 2. A weak learner to make predictions of the input data and 3. An additive model which adds weak learners to minimize the loss function. Basically, this means each new added weak learner is trained one minimizing the errors of the previous models.

In the beginning we stated the LightGBM is using tree based approaches. This also means for us that we have to conserve the sin/cos features of the time series into real integers because a tree will split its decision after one feature even though the sin/cos of the columns is one feature. In the previous preprocessing notebook we encoded these time series sin/cos features to use LightGBM properly. Gradient Boosting algorithms are aggressive learners which means they overfit fast. That's why we first have to drop all features with a high correlation.

The architecture is really simple since we only need to define the right parameters. Since our group decided to optimize the model for the mean absolute error. The objective was set to regression_l1 and the metric is ‚mae' (mean absolute error). The parameters num_leaves and num_round are the most important parameters to tune because these regulate the over-/underfitting of our model. After several trials we identified a small number of leaves with 45 and 17 num of rounds.

Performance:

The LightGBM model is our best model and was able to achieve a slightly better mean absolute error than the DummyRegressor which was predicting the mean. By having a small number of leaves we regulated the overfitting of the model so it was able to capture more information of the training set to make better predictions on the test set. However, gradient boosting lacks the ability to capture time series information because it is a tree approach. In our opinion we were able to minimize the main disadvantage of overfitting but we think other models can be even better by also using the time series information correctly.

LightGBM Model

In [14]:
```
# Drop some features
X_train_window_size_5_tree = X_train_window_size_5_tree.drop(['total_hours','minutes
X_valid_window_size_5_tree = X_valid_window_size_5_tree.drop(['total_hours','minutes
X_test_window_size_5_tree = X_test_window_size_5_tree.drop(['total_hours', 'minutes_
```

In [15]:
```
# Process data
train_data = lgb.Dataset(X_train_window_size_5_tree, label=y_train_window_size_5)
valid_data = lgb.Dataset(X_valid_window_size_5_tree, label=y_valid_window_size_5)
test_data = lgb.Dataset(X_test_window_size_5_tree, label=y_test_window_size_5)

# Modeling
param = {'num_leaves': 60, 'objective': 'regression_l1', 'metric': 'mae'}
num_round = 20
bst = lgb.train(param, train_data, num_round, valid_sets=[valid_data])

# Evaluate model performance and store it
mae, mse = evaluate_model(bst, X_test_window_size_5_tree, y_test_window_size_5)
```

```
performance_df = performance_df.append({'Model': 'LightGBM',
                                        'MAE score (on test set)': round(mae, 4),
                                        'MSE score (on test set)': round(mse, 4)},
                                       ignore_index=True)
```

```
[1]	valid_0's l1: 2.55502
[2]	valid_0's l1: 2.55424
[3]	valid_0's l1: 2.55366
[4]	valid_0's l1: 2.55321
[5]	valid_0's l1: 2.55267
[6]	valid_0's l1: 2.55254
[7]	valid_0's l1: 2.55169
[8]	valid_0's l1: 2.55115
[9]	valid_0's l1: 2.55127
[10]	valid_0's l1: 2.55024
[11]	valid_0's l1: 2.54987
[12]	valid_0's l1: 2.54873
[13]	valid_0's l1: 2.54836
[14]	valid_0's l1: 2.54794
[15]	valid_0's l1: 2.54819
[16]	valid_0's l1: 2.54801
[17]	valid_0's l1: 2.54927
[18]	valid_0's l1: 2.54936
[19]	valid_0's l1: 2.54924
[20]	valid_0's l1: 2.54965
Mean absolut error on test: 2.2494811412930393
Mean squared error on test: 21.102277500657202
```

2.5.Support Vector Regression

- **Intro**:

  The Support Vector Regression model utilizes the kernel trick to project the original dataset to a higher dimension in an efficient way. A main advantage of the model is its ability to tolerate errors within a certain range, which allows the model to become robust.

- **Input data**:

  The input data of the model has been normalized.

  > Training dataset: `X_train_window_size_5` and `y_train_window_size_5` \ Validation dataset: `X_valid_window_size_5` and `y_valid_window_size_5` \ Testing dataset: `X_test_window_size_5` and `y_test_window_size_5`

- **Architecture**:

  After comparing the models' performance on the validation dataset using GridSearchCV, the best combination of parameters for the Support Vector Regression model is: | Parameter name | Value | |-------------------|--------| | kernel | 'poly' | | C | 20' |

In [16]:
```
# Modeling
svrmodel = SVR(kernel='poly', C=20)
svrmodel.fit(X_train_window_size_5, y_train_window_size_5)

# Evaluate model performance and store it
mae, mse = evaluate_model(svrmodel, X_test_window_size_5, y_test_window_size_5)
```

```
performance_df = performance_df.append({'Model': 'Support Vector Regressor',
                                        'MAE score (on test set)': round(mae, 4),
                                        'MSE score (on test set)': round(mse, 4)},
                                       ignore_index=True)
```

```
Mean absolut error on test: 2.255148286587747
Mean squared error on test: 21.162448708317868
```

- **Brief summary**:

  > As shown in the cell above, the Support Vector Regression model's performance on the test dataset is 2.255 in terms of MAE.

  Limitations:

  While Support Vector Regressor manages to introduce non-linearity by projecting the original dataset to a higher dimension space and does it in an effective way, it is relatively time consuming and requires much more resources compared to other models, especially when training on large datasets.

2.6.KNN

The optimal k was found doing a grid search for `range(3, 25)` .

In [17]:

```
# For this 'classic' approach, the train and valid set will be put together to make
X_train_window_size_5_classic = X_train_window_size_5.copy().append(X_valid_window_s
y_train_window_size_5_classic = y_train_window_size_5.copy().append(y_valid_window_s

minmax_transformer_classic = Pipeline(steps=[('minmax', MinMaxScaler())])

preprocessor_window_size_5_classic = ColumnTransformer(
        remainder='passthrough', #passthough features not listed
        transformers=[
            ('mm', minmax_transformer_classic , [X_train_window_size_5_classic.colum
                                                 *[*X_train_window_size_5_classic.co
        ])

preprocessor_window_size_5_classic.fit(X_train_window_size_5, y_train_window_size_5)
X_train_window_size_5_classic_norm = preprocessor_window_size_5_classic.transform(X_
X_test_window_size_5_classic_norm = preprocessor_window_size_5_classic.transform(X_t
```

In [18]:

```
class Knn:

    def __init__(self,
                 X_train: pd.DataFrame,
                 X_test: pd.DataFrame,
                 y_train: pd.DataFrame,
                 y_test: pd.DataFrame,
                 k: int):
        self.X_train = X_train
        self.X_test = X_test
        self.y_train = y_train
        self.y_test = y_test
        self.k = k
        self.model = KNeighborsRegressor(n_neighbors=self.k, n_jobs=-1)
        self.mae_scorer = make_scorer(mean_absolute_error)

    def get_cv_scores(self):
        return cross_val_score(self.model, self.X_train,
                               self.y_train, cv=5,
                               scoring=self.mae_scorer)

    def fit(self):
        self.model.fit(self.X_train, self.y_train)

    def predict(self):
        predictions = self.model.predict(self.X_test)
        return predictions

    def go(self):
        cv_scores = self.get_cv_scores()
        self.fit()
        predictions = self.predict()
        return cv_scores, predictions, self.model
```

In [19]:

```
knn_5 = Knn(
    X_train_window_size_5_classic_norm,
    X_test_window_size_5_classic_norm,
    y_train_window_size_5_classic,
    y_test_window_size_5,
    k=3
)

# Fit model and get cross validation scores
knn_cv_scores_5, knn_predictions_5, knn_model_5 = knn_5.go()
```

CV Scores for KNN

In [20]:

```
knn_cv_scores_5
```

Out[20]:

```
array([1.8916902 , 1.96246531, 2.06411225, 3.64786982, 2.38535877])
```

In [22]:

```
mae, mse = evaluate_model(knn_model_5, X_test_window_size_5, y_test_window_size_5)

performance_df = performance_df.append({'Model': 'K-Nearest Neighbor',
                                        'MAE score (on test set)': round(mae, 4),
                                        'MSE score (on test set)': round(mse, 4)},
                                       ignore_index=True)
```

```
Mean absolut error on test: 2.687419344793223
Mean squared error on test: 23.32592340869457
```

3.Creating DL models

In this part, we employed multiple DL methods.

3.1.Multi-Layer Perceptron

For the MLP, the architecture seemed to be rather irrelevant. Generally speaking, simpler architectures with less neurons seemed to work as well as more complex architectures with several hidden layers and/or a large number of neurons. In total, a few hundred neurons overall were enough. Dropout, LR Scheduling and Adam optimizer created the best results.
We also found that higher batch sizes decreased the MAE.

One interesting finding is that the performance on the validation set would not change after 2-3 epochs. Additionally, the performance on the test set was always better than the performance on the validation set.

In [23]:
```
class Mlp:
    """
    Creates a multilayer-perceptron model.
    """
    def __init__(self,
                 X_train: pd.DataFrame,
                 X_valid: pd.DataFrame,
                 X_test: pd.DataFrame,
                 y_train: pd.DataFrame,
                 y_valid: pd.DataFrame,
                 y_test: pd.DataFrame,
                 params: dict,
                 layers: list,
                 dropout: bool,
                 schedulerthresh: int,
                 optimizer,
                 earlystopping: int
                 ):
        self.X_train = X_train
        self.X_valid = X_valid
        self.X_test = X_test
        self.y_train = y_train
        self.y_valid = y_valid
        self.y_test = y_test
        self.params = params
        self.layers = layers
        self.dropout = dropout
        self.schedulerthresh = schedulerthresh
        self.optimizer = optimizer
        self.earlystopping = earlystopping
        self.model = Sequential()

    def compile_model(self):
        be.clear_session()
        self.model.add(Input(shape=(self.X_train.shape[1])))
        for i in range(len(self.layers)):
            self.model.add(Dense(self.layers[i], activation="relu"))
            if self.dropout:
                self.model.add(Dropout(rate=self.dropout))
        self.model.add(Dense(1, activation="linear"))
        optimizer = self.optimizer
        self.model.compile(loss='mean_absolute_error', optimizer=optimizer)
        return self.model

    def fit_model(self):
        callbacks = []
        if self.scheduler:
            callbacks.append(LearningRateScheduler(self.scheduler))
        if self.earlystopping:
            callbacks.append(EarlyStopping(monitor='loss', patience=self.earlystoppi
        history = self.model.fit(x=self.X_train,y=self.y_train,
                                 batch_size=self.params["BATCH_SIZE"],
                                 validation_data=(self.X_valid,self.y_valid),
                                 epochs=self.params["EPOCHS"],
                                 callbacks=callbacks,
                                 verbose=1,
                                 shuffle=False)
        return history, self.model
    def evaluate_model(self):
        eval_score = self.model.evaluate(self.X_test, self.y_test.to_numpy())
        return eval_score, self.model

    def predict(self):
        return self.model.predict(self.X_test)

    def go(self):
        self.compile_model()
        history, _ = self.fit_model()
        eval_score, _ = self.evaluate_model()
        predictions = self.predict()
        return history, eval_score, predictions, self.model

    def scheduler(self, epoch, lr):
        if self.schedulerthresh:
            thresh = self.schedulerthresh
        else:
            thresh = 5
        if epoch < thresh:
            return lr
        else:
            return lr*exp(-0.1)
```

Creating best model for MLP

In [24]:

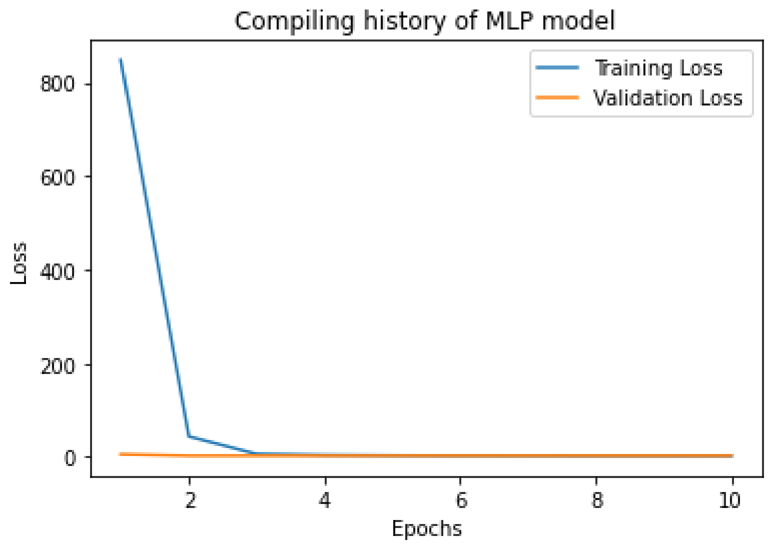
```
params = {
    "BATCH_SIZE": 2048,
    "EPOCHS": 10,
    "LEARNING_RATE": 0.0005}
layers = [300, 100]
dropout = 0.25
scheduler = 5
optimizer = Adam(learning_rate=0.0005)
earlystopping = 3

mlp_5 = Mlp(
    X_train_window_size_5,
    X_valid_window_size_5,
    X_test_window_size_5,
    y_train_window_size_5,
    y_valid_window_size_5,
    y_test_window_size_5,
    params,
    layers,
    dropout,
    scheduler,
    optimizer,
    earlystopping
)

# Fit the model
mlp_history_5, mlp_eval_score_5, mlp_predictions_5, mlp_model_5 = mlp_5.go()
```

```
Epoch 1/10
45/45 [==============================] - 3s 12ms/step - loss: 848.6163 - val_loss:
5.7091 - lr: 5.0000e-04
Epoch 2/10
45/45 [==============================] - 0s 6ms/step - loss: 43.8270 - val_loss: 2.5
556 - lr: 5.0000e-04
Epoch 3/10
45/45 [==============================] - 0s 6ms/step - loss: 6.3598 - val_loss: 2.55
52 - lr: 5.0000e-04
```

```
Epoch 4/10
45/45 [==============================] - 0s 6ms/step - loss: 4.0672 - val_loss: 2.55
51 - lr: 5.0000e-04
Epoch 5/10
45/45 [==============================] - 0s 6ms/step - loss: 3.2134 - val_loss: 2.55
51 - lr: 5.0000e-04
Epoch 6/10
45/45 [==============================] - 0s 6ms/step - loss: 2.7473 - val_loss: 2.55
51 - lr: 4.5242e-04
Epoch 7/10
45/45 [==============================] - 0s 6ms/step - loss: 2.6071 - val_loss: 2.55
52 - lr: 4.0937e-04
Epoch 8/10
45/45 [==============================] - 0s 6ms/step - loss: 2.3839 - val_loss: 2.55
53 - lr: 3.7041e-04
Epoch 9/10
45/45 [==============================] - 0s 6ms/step - loss: 2.3046 - val_loss: 2.55
54 - lr: 3.3516e-04
Epoch 10/10
45/45 [==============================] - 0s 6ms/step - loss: 2.2794 - val_loss: 2.55
54 - lr: 3.0327e-04
335/335 [==============================] - 1s 3ms/step - loss: 2.2550
```

In [25]:

```
loss_values = mlp_history_5.history['loss']
val_loss_values = mlp_history_5.history['val_loss']
epochs = range(1, len(loss_values)+1)

plt.plot(epochs, loss_values, label='Training Loss')
plt.plot(epochs, val_loss_values, label='Validation Loss')

plt.title("Compiling history of MLP model")
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```



Evaluation Score of MLP

In [26]:

```
# Evaluate model performance and store it
print(f"Mean absolut error on test: {mlp_eval_score_5}")
performance_df = performance_df.append({'Model': 'Mulit-Layer Perceptron',
                                        'MAE score (on test set)': round(mlp_eval_sc
                                       ignore_index=True)
```

```
Mean absolut error on test: 2.2549593448638916
```

3.2.Load 3D data for Recurrent Neural Networks

In [27]:

```
# Write code to load the pickles
pkl_file = open("X_train_unflatten_all_5", 'rb')
X_train_unflatten_all_5 = pickle.load(pkl_file)

pkl_file = open("X_valid_unflatten_all_5", 'rb')
X_valid_unflatten_all_5 = pickle.load(pkl_file)

pkl_file = open("X_test_unflatten_all_5", 'rb')
X_test_unflatten_all_5 = pickle.load(pkl_file)
```

3.3.Convolutional Neural Network

- **Intro**:

  A Convolutional Neural Network model involves multiple filters that can be used to capture complex patterns in the datasets. One of its distinct features is its parameter sharing, which can greatly increase its efficiency.

- **Input data**:

  The input data of the model has been normalized and rolled into a form of 3D data.

  > Training dataset: `X_train_unflatten_all_5` and `y_train_window_size_5` \ Validation dataset: `X_valid_unflatten_all_5` and `y_valid_window_size_5` \ Testing dataset: `X_test_unflatten_all_5` and `y_test_window_size_5`

- **Architecture**:

  When increasing the complexity of the model (whether by increasing the number of hidden layers or the number of hidden cells), its performance doesn't seem to change. With various experiments, we have arrived at a relatively simple structure that yields similar results to its more complicated alternatives: 2 pairs of Convolutional layers and Max Pooling layers, followed by 2 pairs of fully connected layers and dropout layers.

In [28]:

```
n_hidden_layers = 2
hidden_layer_size = 40
dropout_rate = 0

BATCH_SIZE = 100
EPOCHS = 20
```

In [29]:

```
tf.compat.v1.reset_default_graph()
be.clear_session()

# Define regularizer and initializer
regularizer = tf.keras.regularizers.L2(2.)
initializer = tf.keras.initializers.RandomUniform()

input_shape = np.shape(X_train_unflatten_all_5)
```

```
column_count = input_shape[2]

input_layer=Input(shape=(input_shape[1], column_count))
cur_last_layer = input_layer

for l in range(n_hidden_layers):
    cnn_layer=Conv1D(filters=5, kernel_size=2,
                     input_shape=input_shape[1:],
                     padding='same',
                     kernel_initializer=initializer,
                     kernel_regularizer=regularizer)(cur_last_layer)
    pool = MaxPool1D(pool_size=2, strides=1)(cnn_layer)
    cur_last_layer=pool

for l in range(n_hidden_layers):
    dense = Dense(100, activation='tanh')(cur_last_layer)
    dropout_layer = Dropout(dropout_rate)(dense)
    cur_last_layer=dropout_layer

predictions=Dense(1)(cur_last_layer)

cnn_model=Model(inputs=input_layer, outputs=predictions)
cnn_model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_1 (InputLayer)        [(None, 4, 22)]           0         
                                                                 
 conv1d (Conv1D)             (None, 4, 5)              225       
                                                                 
 max_pooling1d (MaxPooling1D  (None, 3, 5)             0         
 )                                                               
                                                                 
 conv1d_1 (Conv1D)           (None, 3, 5)              55        
                                                                 
 max_pooling1d_1 (MaxPooling  (None, 2, 5)             0         
 1D)                                                             
                                                                 
 dense (Dense)               (None, 2, 100)            600       
                                                                 
 dropout (Dropout)           (None, 2, 100)            0         
                                                                 
 dense_1 (Dense)             (None, 2, 100)            10100     
                                                                 
 dropout_1 (Dropout)         (None, 2, 100)            0         
                                                                 
 dense_2 (Dense)             (None, 2, 1)              101       
                                                                 
=================================================================
Total params: 11,081
Trainable params: 11,081
Non-trainable params: 0
_________________________________________________________________
```

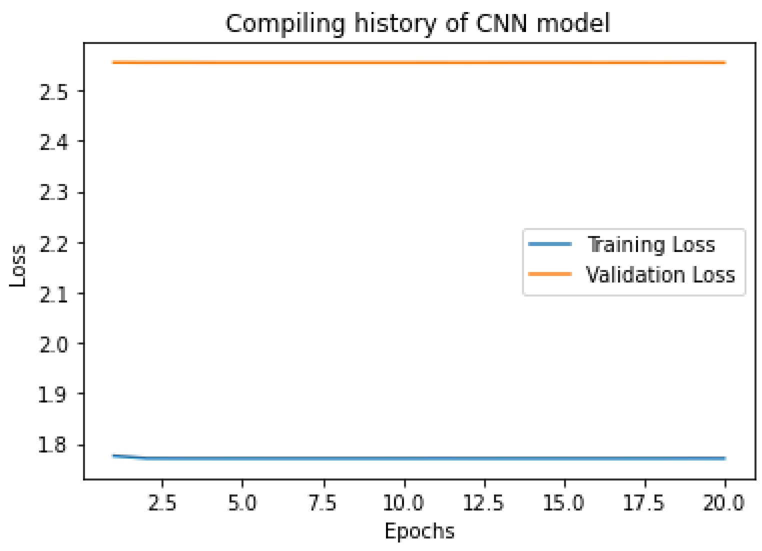
In [30]:
```
optimizer = Adam(learning_rate=0.0018)

cnn_model.compile(loss='mean_absolute_error', optimizer=optimizer)

cnn_history = cnn_model.fit(X_train_unflatten_all_5, y_train_window_size_5,
                            validation_data=(X_valid_unflatten_all_5, y_valid_window
                            epochs=EPOCHS,
                            batch_size=BATCH_SIZE)
```

```
Epoch 1/20
916/916 [==============================] - 35s 8ms/step - loss: 1.7759 - val_loss: 
2.5556
Epoch 2/20
916/916 [==============================] - 7s 7ms/step - loss: 1.7710 - val_loss: 2.
5552
Epoch 3/20
916/916 [==============================] - 7s 7ms/step - loss: 1.7709 - val_loss: 2.
5553
Epoch 4/20
916/916 [==============================] - 7s 7ms/step - loss: 1.7709 - val_loss: 2.
5552
Epoch 5/20
916/916 [==============================] - 7s 7ms/step - loss: 1.7708 - val_loss: 2.
5552
Epoch 6/20
916/916 [==============================] - 7s 7ms/step - loss: 1.7709 - val_loss: 2.
5552
Epoch 7/20
916/916 [==============================] - 7s 7ms/step - loss: 1.7708 - val_loss: 2.
5552
Epoch 8/20
916/916 [==============================] - 7s 7ms/step - loss: 1.7709 - val_loss: 2.
5552
Epoch 9/20
916/916 [==============================] - 7s 7ms/step - loss: 1.7709 - val_loss: 2.
5552
Epoch 10/20
916/916 [==============================] - 7s 7ms/step - loss: 1.7708 - val_loss: 2.
5552
Epoch 11/20
916/916 [==============================] - 7s 7ms/step - loss: 1.7709 - val_loss: 2.
5552
Epoch 12/20
916/916 [==============================] - 7s 7ms/step - loss: 1.7709 - val_loss: 2.
5551
Epoch 13/20
916/916 [==============================] - 7s 7ms/step - loss: 1.7709 - val_loss: 2.
5552
Epoch 14/20
916/916 [==============================] - 7s 7ms/step - loss: 1.7708 - val_loss: 2.
5552
Epoch 15/20
916/916 [==============================] - 7s 7ms/step - loss: 1.7709 - val_loss: 2.
5552
Epoch 16/20
916/916 [==============================] - 7s 8ms/step - loss: 1.7709 - val_loss: 2.
5551
Epoch 17/20
916/916 [==============================] - 7s 8ms/step - loss: 1.7708 - val_loss: 2.
5552
Epoch 18/20
916/916 [==============================] - 7s 8ms/step - loss: 1.7708 - val_loss: 2.
5552
Epoch 19/20
916/916 [==============================] - 7s 8ms/step - loss: 1.7709 - val_loss: 2.
5553
Epoch 20/20
916/916 [==============================] - 7s 8ms/step - loss: 1.7709 - val_loss: 2.
5552
```

In [31]:

```
loss_values = cnn_history.history['loss']
val_loss_values = cnn_history.history['val_loss']
epochs = range(1, len(loss_values)+1)
```

```
plt.plot(epochs, loss_values, label='Training Loss')
plt.plot(epochs, val_loss_values, label='Validation Loss')

plt.title("Compiling history of CNN model")
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```



In [35]:
```
# Evaluate model performance and store it
mae, mse = evaluate_3Dmodel(cnn_model, X_test_unflatten_all_5, y_test_window_size_5)
performance_df = performance_df.append({'Model': 'CNN',
                                        'MAE score (on test set)': round(mae, 4),
                                        'MSE score (on test set)': round(mse, 4)},
                                       ignore_index=True)
```

```
Mean absolut error on test: 2.255152949612842
```

- **Brief summary**:

> As shown in the cell above, the performance of the CNN on the test dataset is 2.255 in terms of MAE.

Limitations:

Generally, it is more common to use CNN in image related problems, therefore, it's not surprising that the model cannot achieve high performance here. Moreover, considering the dimensionality of the data fed into the neural network (with a window size of 5 and therefore 4 "time steps"), the size of kernel that can be used is rather limited.

3.4.Recurrent Neural Network

LSTM: Long- Short-Term Memory

- **Intro**:

A Long- Short-Term Memory model is a type of Recurrent Neural Network that is often used for sequence data such as in time-series and NLP problems. By introducing an additional vector, i.e., the cell state, the model is able to store important information of the past that is helpful for the prediction of the current state (or forget useless information when needed). Since the task we have involve time series data, LSTM seems to be a natural candidate for achieving high performance.

- **Input data**:

  The input data of the model has been normalized and rolled into a form of 3D data.

  > Training dataset: `X_train_unflatten_all_5` and `y_train_window_size_5` \ Validation dataset: `X_valid_unflatten_all_5` and `y_valid_window_size_5` \ Testing dataset: `X_test_unflatten_all_5` and `y_test_window_size_5`

- **Architecture**:

  When increasing the complexity of the model (whether by increasing the number of hidden layers or the number of hidden cells), its performance doesn't seem to change. Below is one of the structures that yields similar results to its alternatives, with 3 pairs of LSTM layers and dropout layers followed by 3 pairs of fully connected layers and dropout layers.

In [36]:
```
n_hidden_layers = 3
hidden_layer_size = 300
dropout_rate = 0.25

BATCH_SIZE = 500
EPOCHS = 20
```

In [37]:
```
tf.compat.v1.reset_default_graph()
be.clear_session()

input_shape[2]
column_count = np.shape(X_train_unflatten_all_5)[2]

input_layer=Input(shape=(np.shape(X_train_unflatten_all_5)[1], column_count))
cur_last_layer=input_layer

for l in range(n_hidden_layers):
    hidden_layer=LSTM(hidden_layer_size, return_sequences=True)(cur_last_layer)
    dropout_layer = Dropout(0.5)(hidden_layer)
    cur_last_layer=dropout_layer

for l in range(n_hidden_layers):
    dense = Dense(100, activation='tanh')(cur_last_layer)
    dropout_layer = Dropout(dropout_rate)(dense)
    cur_last_layer=dropout_layer

predictions=Dense(1)(cur_last_layer)

lstm_model=Model(inputs=input_layer, outputs=predictions)
lstm_model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 4, 22)]           0
 lstm (LSTM)                 (None, 4, 300)            387600

 dropout (Dropout)           (None, 4, 300)            0

 lstm_1 (LSTM)               (None, 4, 300)            721200

 dropout_1 (Dropout)         (None, 4, 300)            0

 lstm_2 (LSTM)               (None, 4, 300)            721200

 dropout_2 (Dropout)         (None, 4, 300)            0

 dense (Dense)               (None, 4, 100)            30100

 dropout_3 (Dropout)         (None, 4, 100)            0

 dense_1 (Dense)             (None, 4, 100)            10100

 dropout_4 (Dropout)         (None, 4, 100)            0

 dense_2 (Dense)             (None, 4, 100)            10100

 dropout_5 (Dropout)         (None, 4, 100)            0

 dense_3 (Dense)             (None, 4, 1)              101

=================================================================
Total params: 1,880,401
Trainable params: 1,880,401
Non-trainable params: 0
_________________________________________________________________
```

In [38]:

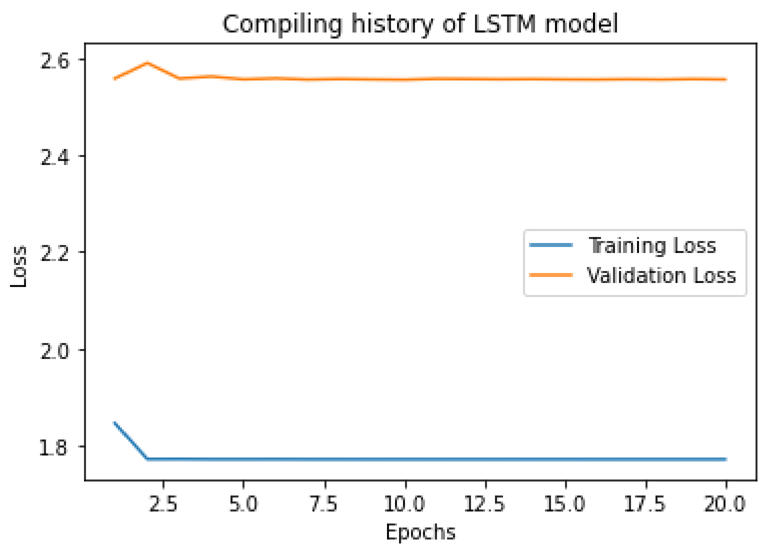
```
optimizer = RMSprop(learning_rate=0.005)
lstm_model.compile(loss='mean_absolute_error', optimizer=optimizer)

lstm_history = lstm_model.fit(X_train_unflatten_all_5, y_train_window_size_5,
                              validation_data=(X_valid_unflatten_all_5, y_valid_wind
                              epochs=EPOCHS,
                              batch_size=BATCH_SIZE)
```

```
Epoch 1/20
184/184 [==============================] - 16s 46ms/step - loss: 1.8461 - val_loss:
2.5578
Epoch 2/20
184/184 [==============================] - 7s 36ms/step - loss: 1.7717 - val_loss:
2.5898
Epoch 3/20
184/184 [==============================] - 7s 35ms/step - loss: 1.7719 - val_loss:
2.5576
Epoch 4/20
184/184 [==============================] - 7s 35ms/step - loss: 1.7715 - val_loss:
2.5619
Epoch 5/20
184/184 [==============================] - 7s 35ms/step - loss: 1.7715 - val_loss:
2.5559
Epoch 6/20
184/184 [==============================] - 7s 36ms/step - loss: 1.7716 - val_loss:
2.5580
Epoch 7/20
184/184 [==============================] - 7s 36ms/step - loss: 1.7714 - val_loss:
2.5555
Epoch 8/20
184/184 [==============================] - 7s 35ms/step - loss: 1.7714 - val_loss:
```

```
2.5566
Epoch 9/20
184/184 [==============================] - 7s 35ms/step - loss: 1.7713 - val_loss:
2.5558
Epoch 10/20
184/184 [==============================] - 7s 35ms/step - loss: 1.7714 - val_loss:
2.5552
Epoch 11/20
184/184 [==============================] - 7s 35ms/step - loss: 1.7713 - val_loss:
2.5570
Epoch 12/20
184/184 [==============================] - 7s 35ms/step - loss: 1.7714 - val_loss:
2.5567
Epoch 13/20
184/184 [==============================] - 7s 35ms/step - loss: 1.7714 - val_loss:
2.5560
Epoch 14/20
184/184 [==============================] - 6s 35ms/step - loss: 1.7714 - val_loss:
2.5564
Epoch 15/20
184/184 [==============================] - 6s 35ms/step - loss: 1.7714 - val_loss:
2.5558
Epoch 16/20
184/184 [==============================] - 6s 35ms/step - loss: 1.7713 - val_loss:
2.5555
Epoch 17/20
184/184 [==============================] - 7s 36ms/step - loss: 1.7714 - val_loss:
2.5561
Epoch 18/20
184/184 [==============================] - 7s 35ms/step - loss: 1.7714 - val_loss:
2.5555
Epoch 19/20
184/184 [==============================] - 7s 35ms/step - loss: 1.7713 - val_loss:
2.5565
Epoch 20/20
184/184 [==============================] - 6s 35ms/step - loss: 1.7714 - val_loss:
2.5557
```

In [39]:

```
loss_values = lstm_history.history['loss']
val_loss_values = lstm_history.history['val_loss']
epochs = range(1, len(loss_values)+1)

plt.plot(epochs, loss_values, label='Training Loss')
plt.plot(epochs, val_loss_values, label='Validation Loss')

plt.title("Compiling history of LSTM model")
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```



In [41]:

```
# Evaluate model performance and store it

mae, mse = evaluate_3Dmodel(lstm_model, X_test_unflatten_all_5, y_test_window_size_5
performance_df = performance_df.append({'Model': 'LSTM',
                                        'MAE score (on test set)': round(mae, 4),
                                        'MSE score (on test set)': round(mse, 4)},
                                       ignore_index=True)
```

```
Mean absolut error on test: 2.2557934026838153
```

- **Brief summary**:

  > As shown in the cell above, the LSTM model's performance on the test dataset is 2.256 in terms of MAE.

  Limitations:

  While the structure of LSTM, especially the introduction of the cell state, allows it to process sequence data better, again, as mentioned above, due to the limitation caused by dimensionality of the input data (with a window size of 5 and therefore 4 "time steps"), the maximum context information that the neural network can store in the cell state is restricted, which is possibly the reason why the network fails to learn much additional information and predict better compared to other models shown above.

Gated Recurrent Unit

- **Intro**:

  A Gated Recurrent Unit model is another type of Recurrent Neural Network. Compared to LSTM, a GRU neural network has fewer gates and parameters and therefore is more efficient to train.

- **Input data**:

  The input data of the model has been normalized and rolled into a form of 3D data.

  > Training dataset: `X_train_unflatten_all_5` and `y_train_window_size_5` \ Validation dataset:

> `X_valid_unflatten_all_5` and `y_valid_window_size_5` \ Testing dataset: `X_test_unflatten_all_5` and `y_test_window_size_5`

- **Architecture**:

  When increasing the complexity of the model (whether by increasing the number of hidden layers or the number of hidden cells), its performance doesn't seem to change. Below is one of the structures that yields similar results to its alternatives, with 2 GRU layers followed by 2 pairs of fully connected layers and dropout layers.

In [42]:
```
n_hidden_layers = 2
hidden_layer_size = 200
dropout_rate = 0.5

BATCH_SIZE = 300
EPOCHS = 20
```

In [43]:
```
tf.compat.v1.reset_default_graph()
be.clear_session()

column_count = np.shape(X_train_unflatten_all_5)[2]

input_layer=Input(shape=(np.shape(X_train_unflatten_all_5)[1], column_count))
cur_last_layer=input_layer

for l in range(n_hidden_layers):
    hidden_layer=GRU(hidden_layer_size, return_sequences=True)(cur_last_layer)
    cur_last_layer=hidden_layer

for l in range(n_hidden_layers):
    dense = Dense(100, activation='sigmoid')(cur_last_layer)
    dropout_layer = Dropout(dropout_rate)(dense)
    cur_last_layer=dropout_layer
predictions=Dense(1)(cur_last_layer)

gru_model=Model(inputs=input_layer, outputs=predictions)
gru_model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 4, 22)]           0

 gru (GRU)                   (None, 4, 200)            134400

 gru_1 (GRU)                 (None, 4, 200)            241200

 dense (Dense)               (None, 4, 100)            20100

 dropout (Dropout)           (None, 4, 100)            0

 dense_1 (Dense)             (None, 4, 100)            10100

 dropout_1 (Dropout)         (None, 4, 100)            0

 dense_2 (Dense)             (None, 4, 1)              101

=================================================================
Total params: 405,901
```

```
Trainable params: 405,901
Non-trainable params: 0
_________________________________________________________________
```

In [44]:

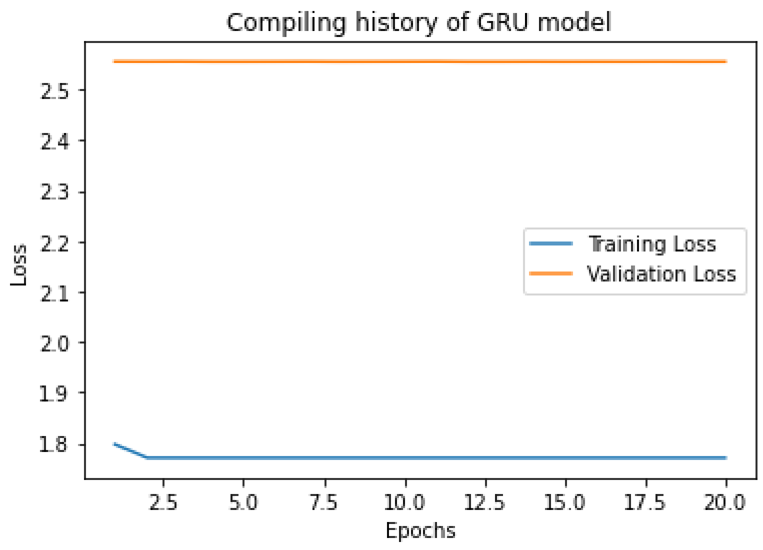
```
optimizer = RMSprop(learning_rate=0.0015)
gru_model.compile(loss='mean_absolute_error', optimizer=optimizer)

gru_history = gru_model.fit(X_train_unflatten_all_5, y_train_window_size_5,
                              validation_data=(X_valid_unflatten_all_5, y_valid_wind
                              epochs=EPOCHS,
                              batch_size=BATCH_SIZE)
```

```
Epoch 1/20
306/306 [==============================] - 11s 22ms/step - loss: 1.7977 - val_loss:
2.5554
Epoch 2/20
306/306 [==============================] - 6s 18ms/step - loss: 1.7709 - val_loss:
2.5553
Epoch 3/20
306/306 [==============================] - 6s 18ms/step - loss: 1.7709 - val_loss:
2.5555
Epoch 4/20
306/306 [==============================] - 6s 19ms/step - loss: 1.7709 - val_loss:
2.5552
Epoch 5/20
306/306 [==============================] - 6s 19ms/step - loss: 1.7709 - val_loss:
2.5552
Epoch 6/20
306/306 [==============================] - 6s 19ms/step - loss: 1.7709 - val_loss:
2.5553
Epoch 7/20
306/306 [==============================] - 6s 18ms/step - loss: 1.7709 - val_loss:
2.5554
Epoch 8/20
306/306 [==============================] - 6s 18ms/step - loss: 1.7709 - val_loss:
2.5551
Epoch 9/20
306/306 [==============================] - 6s 18ms/step - loss: 1.7709 - val_loss:
2.5553
Epoch 10/20
306/306 [==============================] - 6s 18ms/step - loss: 1.7709 - val_loss:
2.5554
Epoch 11/20
306/306 [==============================] - 6s 18ms/step - loss: 1.7709 - val_loss:
2.5555
Epoch 12/20
306/306 [==============================] - 6s 18ms/step - loss: 1.7709 - val_loss:
2.5551
Epoch 13/20
306/306 [==============================] - 6s 18ms/step - loss: 1.7709 - val_loss:
2.5551
Epoch 14/20
306/306 [==============================] - 6s 18ms/step - loss: 1.7709 - val_loss:
2.5553
Epoch 15/20
306/306 [==============================] - 6s 18ms/step - loss: 1.7709 - val_loss:
2.5552
Epoch 16/20
306/306 [==============================] - 6s 18ms/step - loss: 1.7709 - val_loss:
2.5553
Epoch 17/20
306/306 [==============================] - 6s 19ms/step - loss: 1.7709 - val_loss:
2.5552
Epoch 18/20
```

```
306/306 [==============================] - 6s 18ms/step - loss: 1.7709 - val_loss:
2.5553
Epoch 19/20
306/306 [==============================] - 6s 18ms/step - loss: 1.7709 - val_loss:
2.5552
Epoch 20/20
306/306 [==============================] - 6s 18ms/step - loss: 1.7709 - val_loss:
2.5553
```

In [45]:

```
loss_values = gru_history.history['loss']
val_loss_values = gru_history.history['val_loss']
epochs = range(1, len(loss_values)+1)

plt.plot(epochs, loss_values, label='Training Loss')
plt.plot(epochs, val_loss_values, label='Validation Loss')

plt.title('Compiling history of GRU model')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```



In [46]:

```
# Evaluate model performance and store it

mae, mse = evaluate_3Dmodel(gru_model, X_test_unflatten_all_5, y_test_window_size_5)
performance_df = performance_df.append({'Model': 'GRU',
                                        'MAE score (on test set)': round(mae, 4),
                                        'MSE score (on test set)': round(mse, 4)},
                                       ignore_index=True)
```

```
Mean absolut error on test: 2.2552665536389123
```

- **Brief summary**:

> As shown in the cell above, the GRU model's performance on the test dataset is 2.2553 in terms of MAE.

It is interesting to see that compared to LSTM, GRU yields a slightly better performance on the test set. However, the graphs of the compile histories of both models seem very similar--both show that there is little learning after the first 2 or 3 epochs.

Limitations:

With fewer parameters and gates in the neural network, while the speed of training has been increased, the complexity that can be captured by GRU is also reduced. While its performance does seem better than LSTM, the difference can also be caused by other random factors.

3.5.Transformers: Attention is all you need!

Self-Attention

- **Intro**:

The Attention mechanism is one of the state of the art architecture. To gain some experience with it our group decided to try the attention architecture on the our electricity data. However, our prediction problem is not a sequence thats why we have to apply the self-attention mechanism. In the self-attention mechanism both the source of the queries and the target of the attention are input embeddings and are learned projections.

During training we identified that high drop_out rate of 0.5 helps a lot to have a good convergence. Additionally, a batch size of 64 and and a dense layer size of 256 which enables the model to learn well.

Unluckily, the model was also not able to beat the Dummy Regressor but for our group we learned very well how to apply self attention to predict.

- **Input data**:

  The input data of the model has been normalized and rolled into a form of 3D data.

  > Training dataset: `X_train_window_size_5` and `y_train_window_size_5` \ Validation dataset: `X_valid_window_size_5` and `y_valid_window_size_5` \ Testing dataset: `X_test_window_size_5` and `y_test_window_size_5`

- **Architecture**:

1. Input Layer
2. Embedding Layer for the query
3. Self-Attention layer with the inputs and embedding layer
4. Concatenating the input and attention layer
5. 2x Dense Layers followed by a Dropout Layer
6. Output Layer with a linear activation function

In [47]:
```
#Dropping high correlation features to prohibit to overfitting a bit
X_train_window_size_5 = X_train_window_size_5.drop(['total_hours', 'minutes_4_3', 'm
X_valid_window_size_5 = X_valid_window_size_5.drop(['total_hours', 'minutes_4_3', 'm
X_test_window_size_5 = X_test_window_size_5.drop(['total_hours', 'minutes_4_3', 'min
```

In [48]:
```
EPOCHS = 10
BATCH_SIZE = 64
```

```
DENSE_LAYER_SIZE = 256
DROP_OUT = 0.5
```

In [49]:

```
tf.compat.v1.reset_default_graph()
be.clear_session()

def create_self_attention():


    inputs_q = Input(shape=(X_train_window_size_5.shape[1],))

    dense_embedding_layer_q = Dense(X_train_window_size_5.shape[1], activation='soft

    self_attention = tf.keras.layers.Attention()([inputs_q, dense_embedding_layer_q]

    attention_inputs = merge.Concatenate()([inputs_q, self_attention])

    Dense1 = Dense(DENSE_LAYER_SIZE, name='Dense1', activation='relu')(attention_inp

    Dropout1 = Dropout(DROP_OUT)(Dense1)

    Dense2 = Dense(DENSE_LAYER_SIZE, name='Dense2', activation='relu')(Dropout1)
    Dropout2 = Dropout(DROP_OUT)(Dense2)

    output = Dense(1, name='output', activation='linear')(Dropout2)

    model = Model(inputs=[inputs_q], outputs=output)
    print("Architecture of model:\n")
    model.summary()

    #Since we are using a 'state of the art'-model we also tried the 'state of the a
    radam = tfa.optimizers.RectifiedAdam()
    ranger = tfa.optimizers.Lookahead(radam, sync_period=6, slow_step_size=0.5)

    model.compile(optimizer=ranger, loss='mae')

    model_history = model.fit(X_train_window_size_5, y_train_window_size_5,
                              validation_data=(X_valid_window_size_5, y_valid_window
                              epochs=EPOCHS,
                              batch_size=BATCH_SIZE,
                              shuffle=False)

    return model, model_history
```

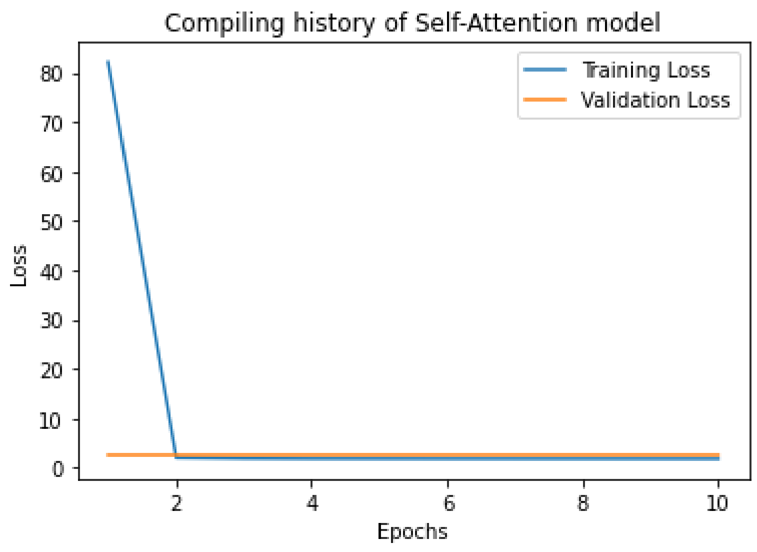
In [50]:

```
attention_model, attention_history = create_self_attention()
```

```
Architecture of model:

Model: "model"
____________________________________________________________________________________
______________
 Layer (type)                   Output Shape         Param #     Connected to
====================================================================================
==============
 input_1 (InputLayer)           [(None, 34)]         0           []

 dense (Dense)                  (None, 34)           1190        ['input_1[0][0]']

 attention (Attention)          (None, 34)           0           ['input_1[0][0]',
                                                                  'dense[0][0]']

 concatenate (Concatenate)      (None, 68)           0           ['input_1[0][0]',
                                                                        'attention[0][0]']

 Dense1 (Dense)              (None, 256)          17664       ['concatenate[0]
[0]']

 dropout (Dropout)           (None, 256)          0           ['Dense1[0][0]']

 Dense2 (Dense)              (None, 256)          65792       ['dropout[0][0]']

 dropout_1 (Dropout)         (None, 256)          0           ['Dense2[0][0]']

 output (Dense)              (None, 1)            257         ['dropout_1[0][0]']

==================================================================================
==============
Total params: 84,903
Trainable params: 84,903
Non-trainable params: 0
__________________________________________________________________________________
______________
Epoch 1/10
1430/1430 [==============================] - 18s 10ms/step - loss: 82.2027 - val_los
s: 2.5552
Epoch 2/10
1430/1430 [==============================] - 14s 10ms/step - loss: 1.9801 - val_los
s: 2.5551
Epoch 3/10
1430/1430 [==============================] - 14s 10ms/step - loss: 1.8514 - val_los
s: 2.5551
Epoch 4/10
1430/1430 [==============================] - 14s 10ms/step - loss: 1.8023 - val_los
s: 2.5551
Epoch 5/10
1430/1430 [==============================] - 14s 10ms/step - loss: 1.7924 - val_los
s: 2.5552
Epoch 6/10
1430/1430 [==============================] - 14s 10ms/step - loss: 1.7830 - val_los
s: 2.5551
Epoch 7/10
1430/1430 [==============================] - 14s 10ms/step - loss: 1.7795 - val_los
s: 2.5551
Epoch 8/10
1430/1430 [==============================] - 14s 10ms/step - loss: 1.7790 - val_los
s: 2.5551
Epoch 9/10
1430/1430 [==============================] - 14s 10ms/step - loss: 1.7764 - val_los
s: 2.5551
Epoch 10/10
1430/1430 [==============================] - 14s 10ms/step - loss: 1.7750 - val_los
s: 2.5551
```

In [51]:

```
loss_values = attention_history.history['loss']
val_loss_values = attention_history.history['val_loss']
epochs = range(1, len(loss_values)+1)

plt.plot(epochs, loss_values, label='Training Loss')
plt.plot(epochs, val_loss_values, label='Validation Loss')

plt.title("Compiling history of Self-Attention model")
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



In [52]:
```
mae, mse = evaluate_model(attention_model, X_test_window_size_5, y_test_window_size_
performance_df = performance_df.append({'Model': 'Self-Attention',
                                        'MAE score (on test set)': round(mae, 4),
                                        'MSE score (on test set)': round(mse, 4)},
                                       ignore_index=True)
```

```
Mean absolut error on test: 2.255134580546972
Mean squared error on test: 21.161776632786403
```

4.Summary

In this part, we summairzed the results of all the models above.

In [53]:
```
performance_df.sort_values('MAE score (on test set)')
```

Out[53]:

	Model	MAE score (on test set)	MSE score (on test set)
4	LightGBM	2.2495	21.1023
7	Mulit-Layer Perceptron	2.2550	NaN
5	Support Vector Regressor	2.2551	21.1624
11	Self-Attention	2.2551	21.1618
8	CNN	2.2552	21.1624
10	GRU	2.2553	21.1627
9	LSTM	2.2558	21.1615
0	Dummy Regressor	2.2588	21.1589
3	Random Forest	2.2755	21.8467
1	Linear Regression	2.2783	21.2577
2	Linear Regression (poly transformed)	2.4848	22.6935
6	K-Nearest Neighbor	2.6874	23.3259

Observations:

- Boosted decision trees have the best performance for this dataset, which is around 0.25% better than the second best model in terms of MAE.

  It is not surprising to see that LightGBM model has a higher performance when compared to Random Forest. As mentioned earlier, by training different decision trees in parallel, Random Forest can only learn from the training data. Boosted models such as LightGBM, however, build trees that are dependent on each other, which allows more room for the model to learn and explains the better performance seen here.

- The performance of all neural networks are very close to each other.

  Whether it is Recurrent Neural Network, which is more suitable for sequence data, or Convolutional Neural Network, which is more suitable for image data, their performance results seem very similar to each other. Moreover, as mentioned earlier, during the stage of hyperparameter tuning, we have discovered that the complexity of the model also did not make much difference.

  It is also worth noting that when training the neural networks, we have discovered that the validation and training loss values tend to fluctuate around a certain value, and regardless of the type of optimizer/learning rate (scheduler)/activation function/initializer that we use, the model can not seem to reach a lower value.

  Therefore, it is very likely that neural network is not suitable for solving the problem we have at hand. Generally, neural networks are helpful in identifying patterns in data. However, the dataset here involves not just one time series but multiple sequences of different contracts, which can add a lot of noise and, without a large amount of data to make up for this drawback, it can be one of the reasons why neural networks are unable to learn from it.

- Some additional note to the comparison between Neural Nets and Decision Trees:

  Even though Neural Networks are comapred to Decision Trees very often because both are able to handle non-linear relationships within the data, neural networks have some disadvantages in capturing categorical multiclass problems. Additionally, as stated in the introduction, our data consists of small time series sequences of electricity contracts with varying lengths, which makes it aslo hard for the state of the art LSTM model that is able to remember important features and information of a time series. In contrast to that, gradient boosted decision trees don't even consider the time series and the included auto-correlation. However, this kind of models tend to overfit on training data very fast, but we were able to mitigate this by dropping features with high correlation and increase the independence between the variables.

5.Error Analysis and Limitations

5.1.Error Analysis

One of the most important findings is that nearly all of our models had a hard time beating the baseline set by the dummy regressor, which only predicts the average value. To analyze this behaviour we will look into distribution of the train target variable, the test target variable and the prediction of our best model, the LightGBM model.

In [54]:
```
lightgbm_predictions = bst.predict(X_test_window_size_5)
dummy_model_predictions = dummy_model.predict(X_test_window_size_5)
```

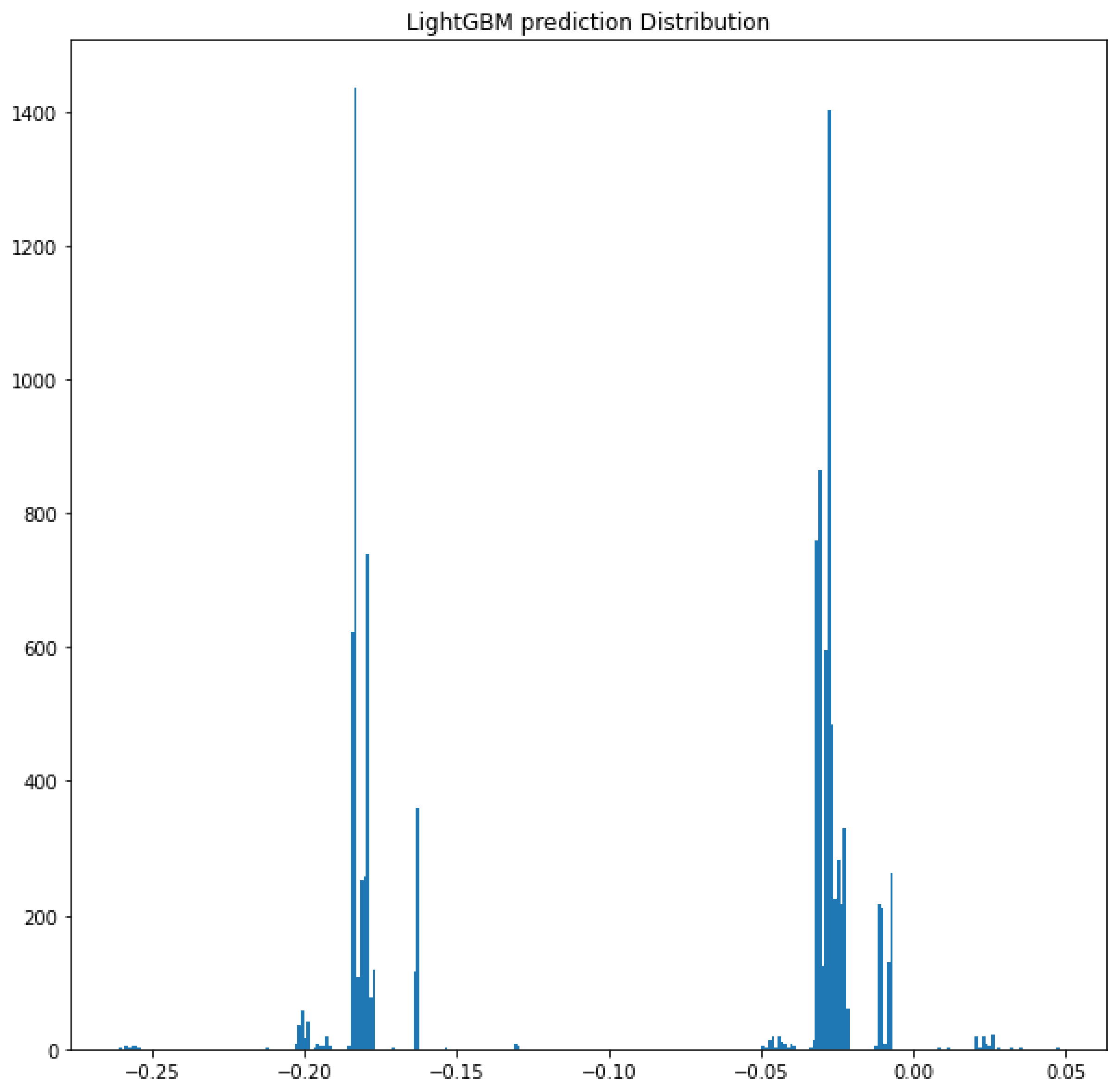
In [55]:
```
print(f'These are the predictions of lightgbtm: {lightgbm_predictions}')
print(f'These are the predictions of the dummy model:  {dummy_model_predictions}')
print(f'These are the real values: \n{y_test_window_size_5}')
```

```
These are the predictions of lightgbtm: [-0.19360566 -0.18433629 -0.18433629 ... -0.
03179153 -0.03179153
 -0.03179153]
These are the predictions of the dummy model:  [0.03707088 0.03707088 0.03707088 ...
0.03707088 0.03707088 0.03707088]
These are the real values: 
0        2.28
1       -6.57
2       -4.86
3       -0.68
4       -0.83
         ... 
10699    2.26
10700    3.05
10701   -5.33
10702   -1.30
10703    6.69
Name: y, Length: 10704, dtype: float64
```

In [56]:
```
print(f'These are the average prediction values of LightGBM {sum(lightgbm_prediction
print(f'This the average of the test target variable {sum(dummy_model_predictions) /
print(f'This the average of the train target variable {sum(y_train_window_size_5) /
print(f'This is the average predicted by the dummy model {sum(y_test_window_size_5)
```

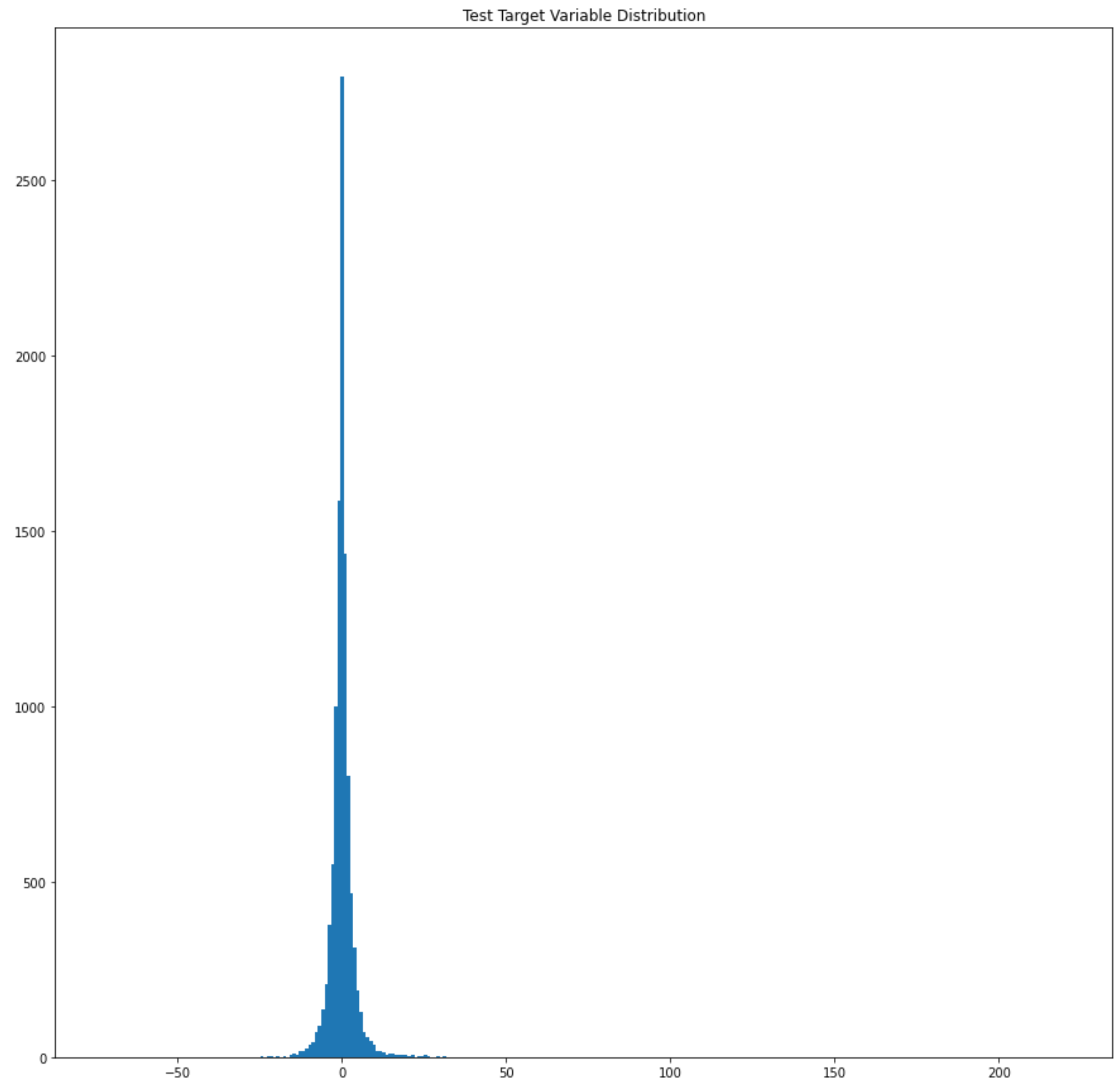
```
These are the average prediction values of LightGBM -0.08794192033593307
This the average of the test target variable 0.037070875950690634
This the average of the train target variable 0.03707087595069501
This is the average predicted by the dummy model 0.06584547832585931
```

In [57]:
```
col = "LightGBM prediction"
plt.figure(figsize=(10,10))
plt.hist(lightgbm_predictions, bins= 300)
plt.title(f"{col} Distribution")
plt.show()
```



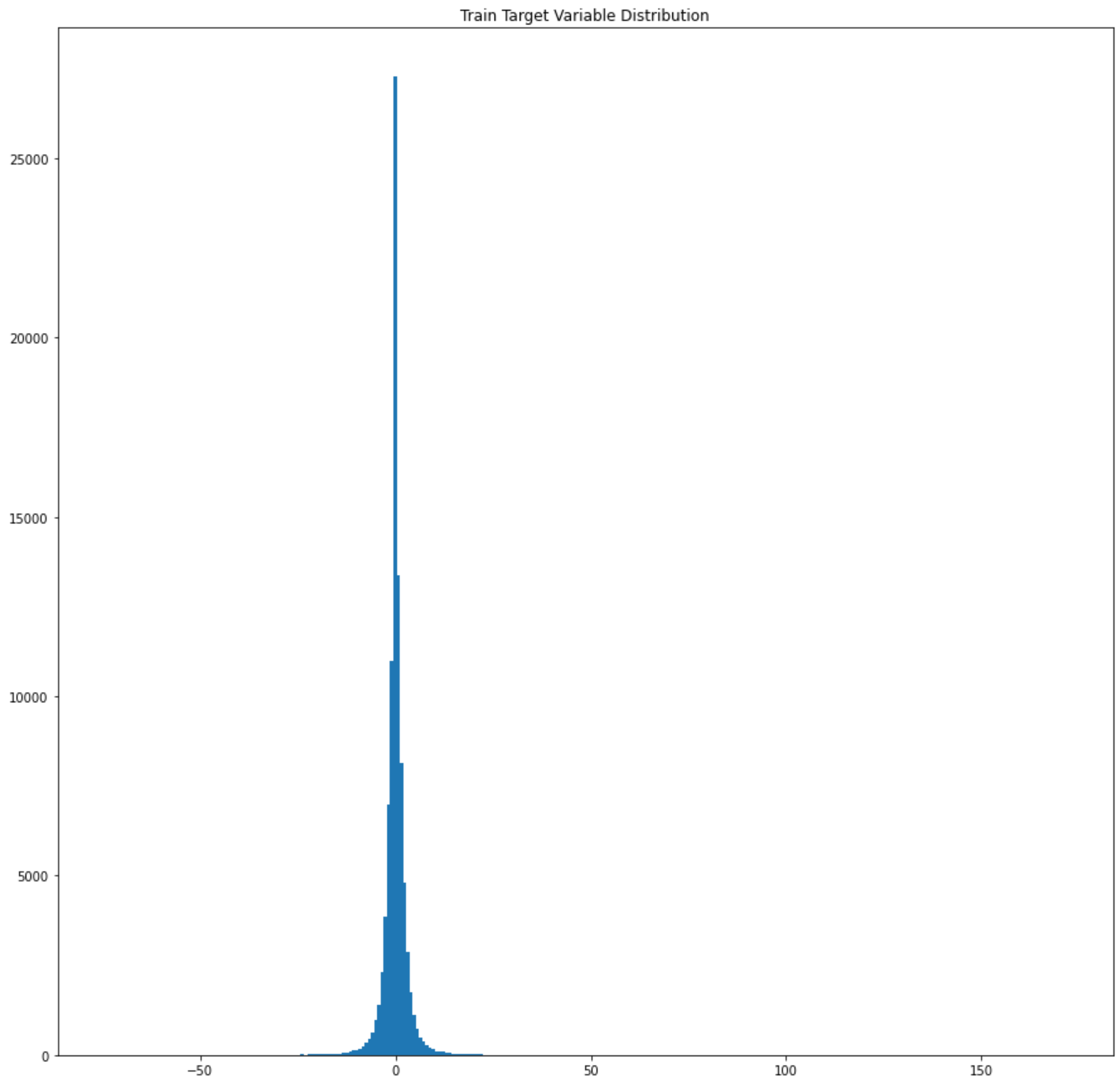
In [58]:

```
col = "Test Target Variable"
plt.figure(figsize=(15,15))
plt.hist(y_test_window_size_5, bins= 300)

plt.title(f"{col} Distribution")
plt.show()
```

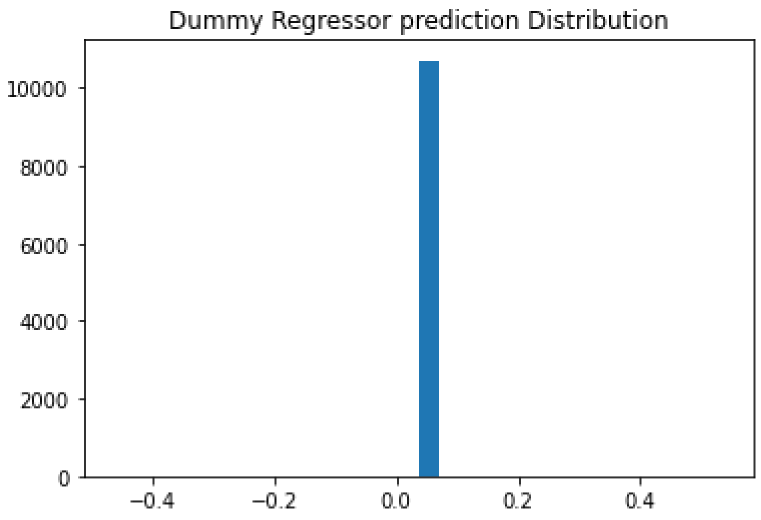


In [59]:

```
col = "Train Target Variable"
plt.figure(figsize=(15,15))
plt.hist(y_train_window_size_5, bins= 300)

plt.title(f"{col} Distribution")
plt.show()
```



```
In [60]: col = "Dummy Regressor prediction"

         plt.hist(dummy_model_predictions, bins= 30)
         plt.title(f"{col} Distribution")
         plt.show()
```



In summary, we can see that the LightGBM does not predict a constant value but values between -0.3 and 0.2 so in a very narrow space. The average of these predictions is roundabout

-0.06 so an opposite value to the +0.06 average of the Dummy Regressor's prediction. This makes more sense as we are looking into the distribution of the train and test target variable. Both are distributed very dense around 0, which also explains why the Dummy Regressor model is doing so well. Predicting only one value for a dense distribution also minimizes the mean abolsut error the best which gives us also an understanding why only the LightGBM model was able to beat the Dummy Regressor.

5.2.Limitations - shift in data distribution

Dataset shift can be the root of many problems in machine learning tasks, the most obvious being that trained models are not generalizing well beyond the training data.

More specifically, we are dealing with dataset shift, if the features' distributions are significantly different across training, validation and test set.

While identifying dataset shift itself does not solve its problems, it does offer a partial explanation of our models' accuracies and discrepancies in accuracy metrics across training, validation and test sets.

There are multiple types of dataset shift depending on which kind of data "shifts". We will be looking at **covariate shift** (do the input features shift?) and **prior probability shift** (does the target variable shift?).

There are two major explanations why the dataset is shifting:

1. Sample selection
2. "Shifting" or non-stationary environments

For our data it is most likely that both causes play a role, as the arbirtrary splitting into training, validation and test data represent differently sized samples (1.) and they exhibit a temporal order which might be accompanied by shifts in time (2.).

We will be looking at the unproprocessed window size 5 data set exclusively as findings should apply to the window size 15 data set, which is only being extended in the time steps columns.

In [61]:
```
# Load raw data for window size 5
X_train_window_size_5 = pd.read_csv('X_train_window_size_5_time_encoding_True.csv')
y_train_window_size_5 = pd.read_csv('y_train_window_size_5_time_encoding_True.csv')
X_valid_window_size_5 = pd.read_csv('X_valid_window_size_5_time_encoding_True.csv')
y_valid_window_size_5 = pd.read_csv('y_valid_window_size_5_time_encoding_True.csv')
X_test_window_size_5 = pd.read_csv('X_test_window_size_5_time_encoding_True.csv')
y_test_window_size_5 = pd.read_csv('y_test_window_size_5_time_encoding_True.csv')
```

Covariate shift

Let's take a look at how the distributions change across the training, validation and test set.

In order to make the distributions comparable, we use weights <1 (calculcated below) applied to the frequencies for the training and test set to bring them to the same count as the validation set, being the smallest data set.

In [62]:
```
# look at row count and identify smallest dataset
```

```
print(f"Row count of training set with window size 5: {X_train_window_size_5.shape[0
print(f"Row count of validation set with window size 5: {X_valid_window_size_5.shape
print(f"Row count of test set with window size 5: {X_test_window_size_5.shape[0]}\n"
```

```
Row count of training set with window size 5: 91512
Row count of validation set with window size 5: 4817
Row count of test set with window size 5: 10704
```

In [63]:

```
# calculate weights to make sure that distributions are comparable in the y-axis (w)
# for window size 5
train_valid_weight_5 = 1/(X_train_window_size_5.shape[0]/X_valid_window_size_5.shape
test_valid_weight_5 = 1/(X_test_window_size_5.shape[0]/X_valid_window_size_5.shape[0
# not necessary, but increases code understandability
valid_valid_weight_5 = 1/(X_valid_window_size_5.shape[0]/X_valid_window_size_5.shape
# fill np arrays with tespective weight values (same for all values within a dataset
train_valid_weights_5 = np.full((X_train_window_size_5.shape[0], ), train_valid_weig
valid_valid_weights_5 = np.full((X_valid_window_size_5.shape[0], ), valid_valid_weig
test_valid_weights_5 = np.full((X_test_window_size_5.shape[0], ), test_valid_weight_

print(f"These weights should be used to weigh the frequency for the train, test and 
      f"validations data sets with window size 5, respectively: {round(train_valid_w
```

```
These weights should be used to weigh the frequency for the train, test and validati
ons data sets with window size 5, respectively: (0.05, 0.45, 1.0)
```

In [64]:

```
# create function for changing the integer-based column names for window size 5
def rename_time_series_5(df):
    labels = ["open", "high", "low", "close", "volume", "minutes"]
    new_cols = list(df.columns[:17])
    for i in range(4, 0, -1):
        for label in labels:
            new_cols.append(label+f"_{i}_{i-1}")

    df_original = df.copy()
    df.columns = new_cols

    return df

# apply renaming functions to all X datasets
X_train_window_size_5 = rename_time_series_5(X_train_window_size_5)
X_valid_window_size_5 = rename_time_series_5(X_valid_window_size_5)
X_test_window_size_5 = rename_time_series_5(X_test_window_size_5)
```

In [65]:

```
# plot relative distributions for all features
datasets_5 = [X_train_window_size_5, X_valid_window_size_5, X_test_window_size_5]

bins = 50

fig, ax = plt.subplots(datasets_5[0].shape[1], 1, figsize = (50, 450))

for i in range(len(X_train_window_size_5.columns)):
    ax[i].hist(datasets_5[0][datasets_5[0].columns[i]], bins = bins, weights = train
    ax[i].hist(datasets_5[1][datasets_5[1].columns[i]], bins = bins, weights = valid
    ax[i].hist(datasets_5[2][datasets_5[2].columns[i]], bins = bins, weights = test_
    ax[i].legend(fontsize = 20)
    ax[i].set_title(f" Column: {datasets_5[0].columns[i]} // train mean: {round(data
    ax[i].tick_params(labelsize=20)
    ax[i].set_ylabel("relative frequency", fontsize=20)
    ax[i].set_xlabel("feature values", fontsize=20)
```

```
Output hidden; open in https://colab.research.google.com to view.
```

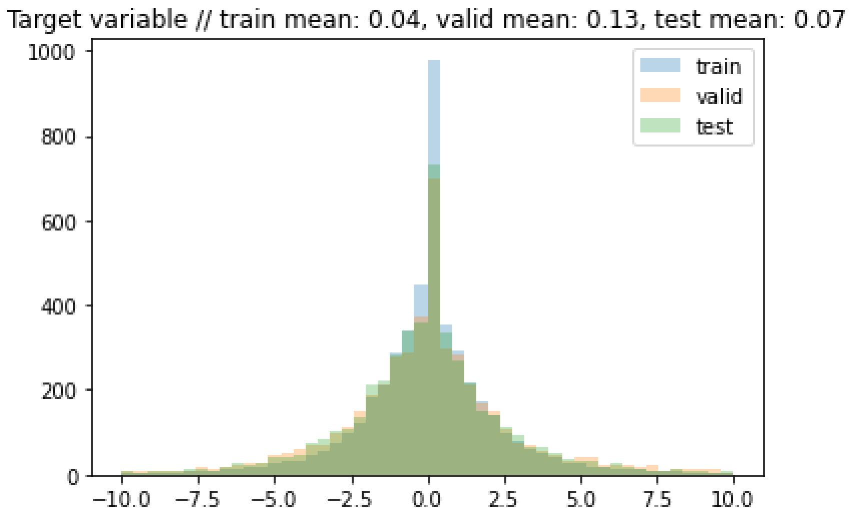
Main findings

- Significant shift in feature total_hours from training set compared to validation and test set (see means)
- No dlvry_bank_holiday seem to be only represented in the training set
- Drastic discrepancies between the similar distributions of train and test sets compared to the validation set in the dlvry_day features (see means), same holds for the lasttrade_day features
- Looking at the time series features, we can see that the open, high, low values at t-4 and t-1 are systematically different in validation and test set compared to the more 0-centric training data (see means)

Prior probability shift

Let's take a look at whether there are major distribution shifts among the **target values**.

In [66]:
```
# plot histogram for y values
plt.hist(y_train_window_size_5.y, bins = bins, weights = train_valid_weights_5, alph
plt.hist(y_valid_window_size_5.y, bins = bins, weights = valid_valid_weights_5, alph
plt.hist(y_test_window_size_5.y, bins = bins, weights = test_valid_weights_5, alpha=
plt.legend()
plt.title(f"Target variable // train mean: {round(float(y_train_window_size_5.mean()
plt.show()
```



Main findings

- While the target variable distributions are quite similar at first sight, we see that the training set's mean is dragged closer to zero via frequently occuring values close to zero
- There is a ~2x increase in mean from training to test set and another ~2x increase from test to valdiation set
- As this is the value the models are trying to predict based on training on the training set, we can see how generalization might be impeded when the distributions' means change that much across training and test sets

Conclusion

From looking at distributional shifts across training, validation and test sets for both the input features and the target variable, we can understand the generalization difficulties of our models (both from training to validation and from training to test sets).