School of InfoComm Technology — Taking IT Higher

Deep Learning

Practical 2b - Predicting House Prices

AY2020/21 Semester

In [2]:
```
from tensorflow import keras
print('keras: ', keras.__version__)
```

```
keras:  2.2.4-tf
```

In [3]:
```
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:95% !important; }</style>"))
```

Objectives ¶

After completing this practical exercise, students should be able to:

1. Build a neural network model to predict house prices
2. Exercise- tuning several model parameters

1. Predicting house prices (a regression example)

1.1 The Boston Housing Price dataset

We will be attempting to predict the median price of homes in a given Boston suburb in the mid-1970s, given a few data points about the suburb at the time, such as the crime rate, the local property tax rate, etc.

The dataset has only 506 samples, split between 404 training samples and 102 test samples. Let's take a look at the data:

In [4]:
```python
from tensorflow.keras.datasets import boston_housing

(train_data, train_targets), (test_data, test_targets) =  boston_housing.load_data()
```

In [5]:
```python
train_data.shape
```

Out[5]:
```
(404, 13)
```

In [6]:
```python
print(train_data[100])
```

```
[6.1290e-02 2.0000e+01 3.3300e+00 1.0000e+00 4.4290e-01 7.6450e+00
 4.9700e+01 5.2119e+00 5.0000e+00 2.1600e+02 1.4900e+01 3.7707e+02
 3.0100e+00]
```

In [7]:
```python
test_data.shape
```

Out[7]:
```
(102, 13)
```

As you can see, we have 404 training samples and 102 test samples. The data comprises 13 features (details are shown below) and each "feature" in the input data (e.g. the crime rate is a feature) has a different scale. For instance some values are proportions, which take a values between 0 and 1, others take values between 1 and 12, others between 0 and 100...

1. Per capita crime rate.
2. Proportion of residential land zoned for lots over 25,000 square feet.
3. Proportion of non-retail business acres per town.
4. Charles River dummy variable (= 1 if tract bounds river; 0 otherwise).
5. Nitric oxides concentration (parts per 10 million).
6. Average number of rooms per dwelling.
7. Proportion of owner-occupied units built prior to 1940.
8. Weighted distances to five Boston employment centres.
9. Index of accessibility to radial highways.
10. Full-value property-tax rate per $10,000.
11. Pupil-teacher ratio by town.
12. 1000 * (Bk - 0.63) ** 2 where Bk is the proportion of Black people by town.
13. % lower status of the population.

The targets are the median values of owner-occupied homes, in thousands of dollars:

In [8]:
```
train_targets
```

Out[8]:
```
array([15.2, 42.3, 50. , 21.1, 17.7, 18.5, 11.3, 15.6, 15.6, 14.4, 12.1,
       17.9, 23.1, 19.9, 15.7,  8.8, 50. , 22.5, 24.1, 27.5, 10.9, 30.8,
       32.9, 24. , 18.5, 13.3, 22.9, 34.7, 16.6, 17.5, 22.3, 16.1, 14.9,
       23.1, 34.9, 25. , 13.9, 13.1, 20.4, 20. , 15.2, 24.7, 22.2, 16.7,
       12.7, 15.6, 18.4, 21. , 30.1, 15.1, 18.7,  9.6, 31.5, 24.8, 19.1,
       22. , 14.5, 11. , 32. , 29.4, 20.3, 24.4, 14.6, 19.5, 14.1, 14.3,
       15.6, 10.5,  6.3, 19.3, 19.3, 13.4, 36.4, 17.8, 13.5, 16.5,  8.3,
       14.3, 16. , 13.4, 28.6, 43.5, 20.2, 22. , 23. , 20.7, 12.5, 48.5,
       14.6, 13.4, 23.7, 50. , 21.7, 39.8, 38.7, 22.2, 34.9, 22.5, 31.1,
       28.7, 46. , 41.7, 21. , 26.6, 15. , 24.4, 13.3, 21.2, 11.7, 21.7,
       19.4, 50. , 22.8, 19.7, 24.7, 36.2, 14.2, 18.9, 18.3, 20.6, 24.6,
       18.2,  8.7, 44. , 10.4, 13.2, 21.2, 37. , 30.7, 22.9, 20. , 19.3,
       31.7, 32. , 23.1, 18.8, 10.9, 50. , 19.6,  5. , 14.4, 19.8, 13.8,
       19.6, 23.9, 24.5, 25. , 19.9, 17.2, 24.6, 13.5, 26.6, 21.4, 11.9,
       22.6, 19.6,  8.5, 23.7, 23.1, 22.4, 20.5, 23.6, 18.4, 35.2, 23.1,
       27.9, 20.6, 23.7, 28. , 13.6, 27.1, 23.6, 20.6, 18.2, 21.7, 17.1,
        8.4, 25.3, 13.8, 22.2, 18.4, 20.7, 31.6, 30.5, 20.3,  8.8, 19.2,
       19.4, 23.1, 23. , 14.8, 48.8, 22.6, 33.4, 21.1, 13.6, 32.2, 13.1,
       23.4, 18.9, 23.9, 11.8, 23.3, 22.8, 19.6, 16.7, 13.4, 22.2, 20.4,
```

The prices are typically between $10, 000-$50,000. If that sounds cheap, remember this was the mid-1970s, and these prices are not inflation-adjusted.

1.2 Preparing the data

It would be problematic to feed into a neural network values that all take wildly different ranges. The network might be able to automatically adapt to such heterogeneous data, but it would definitely make learning more difficult. A widespread best practice to deal with such data is to do feature-wise normalization: for each feature in the input data (a column in the input data matrix), we will subtract the mean of the feature and divide by the standard deviation, so that the feature will be centered around 0 and will have a unit standard deviation. This is easily done in Numpy:

In [9]:
```
mean = train_data.mean(axis=0)
std = train_data.std(axis=0)
train_data -= mean
train_data /= std

test_data -= mean
test_data /= std
```

Note that the quantities that we use for normalizing the test data have been computed using the training data. We should never use in our workflow any quantity computed on the test data, even for something as simple as data normalization.

In [10]:
```
print(train_data[100])
```

```
[-0.39914449  0.35890566 -1.14281587  3.89358447 -0.97702129  1.9437178
 -0.69198737  0.72576261 -0.51114231 -1.1428069  -1.62718308  0.23710757
 -1.34300395]
```

1.3 Building our network

Because so few samples are available, we will be using a very small network with two hidden layers, each with 64 units. In general, the less training data you have, the worse overfitting will be, and using a small network is one way to mitigate overfitting.

In [11]:
```
from tensorflow.keras import models
from tensorflow.keras import layers
from tensorflow.keras import optimizers

model = models.Sequential()
model.add(layers.Dense(64, activation='relu',
                       input_shape=(train_data.shape[1],)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(1))

model.compile(optimizer=optimizers.RMSprop(lr=0.001), loss='mse', metrics=['mae'])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 64)                896
_________________________________________________________________
dense_1 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 65
=================================================================
Total params: 5,121
Trainable params: 5,121
Non-trainable params: 0
_________________________________________________________________
```

Our network ends with a single unit, and no activation (i.e. linear layer). This is a typical setup for scalar regression. Because the last layer is purely linear, the network is free to learn to predict values in any range.

Note that we are compiling the network with the `mse` loss function -- Mean Squared Error, the square of the difference between the predictions and the targets, a widely used loss function for regression problems.

We are also monitoring a new metric during training: `mae` . This stands for Mean Absolute Error. It is simply the absolute value of the difference between the predictions and the targets. For instance, a MAE of 0.5 on this problem would mean that our predictions are off by $500 on average.

In [12]:
```
model.fit(train_data, train_targets, validation_split =0.2, epochs = 200, batch_size = 1)
```

```
Train on 323 samples, validate on 81 samples
Epoch 1/200
323/323 [==============================] - 2s 6ms/sample - loss: 203.2259 - mae: 10.9547 - val_loss: 64.39
10 - val_mae: 5.2544
Epoch 2/200
323/323 [==============================] - 1s 2ms/sample - loss: 29.3281 - mae: 3.5543 - val_loss: 29.5492
- val_mae: 3.5370
Epoch 3/200
323/323 [==============================] - 1s 3ms/sample - loss: 21.3152 - mae: 2.8764 - val_loss: 27.2706
- val_mae: 3.5433
Epoch 4/200
323/323 [==============================] - 1s 2ms/sample - loss: 18.5062 - mae: 2.6546 - val_loss: 20.0039
- val_mae: 3.0831
Epoch 5/200
323/323 [==============================] - 1s 2ms/sample - loss: 16.1202 - mae: 2.5003 - val_loss: 19.8854
- val_mae: 2.9145
Epoch 6/200
323/323 [==============================] - 1s 2ms/sample - loss: 14.7287 - mae: 2.4604 - val_loss: 18.8125
- val_mae: 3.0771
```

As you can see, the model very quickly overfits to the training data, so we should stop it before it overfits. Now the question is when to stop? We will use K-fold validation to figure out in the next secsion.

Because you'll need to instantiate the same model multiple times, you use a function to construct it.

In [13]:

```
from tensorflow.keras import models
from tensorflow.keras import layers

def build_model():
    # Because we will need to instantiate
    # the same model multiple times,
    # we use a function to construct it.
    model = models.Sequential()
    model.add(layers.Dense(64, activation='relu',
                           input_shape=(train_data.shape[1],)))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    return model
```

1.4 Validating our approach using K-fold validation

To evaluate our network while we keep adjusting its parameters (e.g. the number of epochs), we use K-fold cross-validation because we have so few data points. It splits the data into K partitions, then instantiating K identical models, and training each one on K-1 partitions while evaluating on the remaining partition. The validation score for the model used would then be the average of the K validation scores obtained.

In [14]:

```
from tensorflow.keras import backend as K
# Some memory clean-up
K.clear_session()
```

In [15]:

```python
import numpy as np

k = 5
num_val_samples = len(train_data) // k
num_epochs = 120
all_scores = []

all_mae_histories = []
for i in range(k):
    print('processing fold #', i)
    # Prepare the validation data: data from partition # k
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]

    # Prepare the training data: data from all other partitions
    partial_train_data = np.concatenate(
        [train_data[:i * num_val_samples],
         train_data[(i + 1) * num_val_samples:]],
        axis=0)
    partial_train_targets = np.concatenate(
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]],
        axis=0)

    # Build the Keras model (already compiled)
    model = build_model()
    # Train the model (in silent mode, verbose=0)
    history = model.fit(partial_train_data, partial_train_targets,
                        validation_data=(val_data, val_targets),
                        epochs=num_epochs, batch_size=1, verbose=1)
    mae_history = history.history['val_mae']
    all_mae_histories.append(mae_history)
```

```
processing fold # 0
Train on 324 samples, validate on 80 samples
Epoch 1/120
324/324 [==============================] - 1s 4ms/sample - loss: 171.5867 - mae: 9.6710 - val_loss: 33.167
4 - val_mae: 3.5690
Epoch 2/120
324/324 [==============================] - 1s 2ms/sample - loss: 28.2962 - mae: 3.6407 - val_loss: 23.5491
- val_mae: 2.8806
Epoch 3/120
324/324 [==============================] - 1s 2ms/sample - loss: 21.4015 - mae: 3.0764 - val_loss: 19.0381
- val_mae: 2.8802
Epoch 4/120
324/324 [==============================] - 1s 2ms/sample - loss: 18.1230 - mae: 2.7843 - val_loss: 16.3608
```

```
- val_mae: 2.5906
Epoch 5/120
324/324 [==============================] - 1s 2ms/sample - loss: 16.8374 - mae: 2.7547 - val_loss: 14.2632
- val_mae: 2.3483
Epoch 6/120
324/324 [==============================] - 1s 2ms/sample - loss: 15.0204 - mae: 2.6641 - val_loss: 13.1379
```

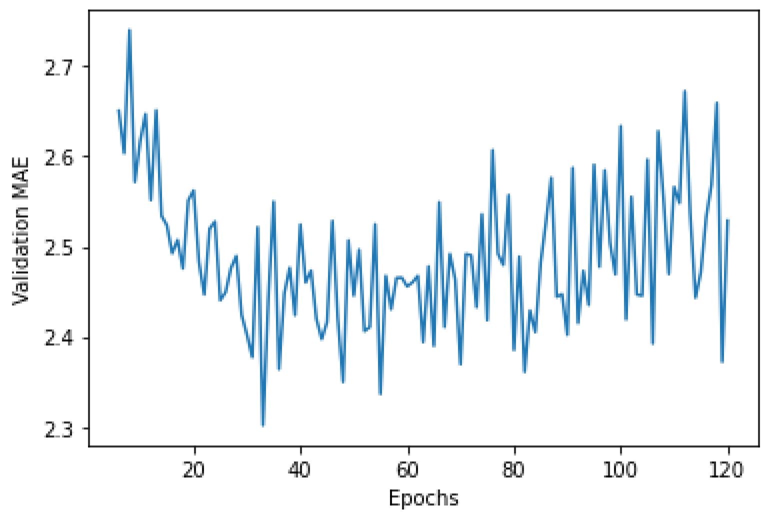
We can then compute the average of the per-epoch MAE scores for all folds:

In [16]:
```python
average_mae_history = [
    np.mean([x[i] for x in all_mae_histories]) for i in range(num_epochs)]
```

Let's plot this:

In [17]:
```python
import matplotlib.pyplot as plt
%matplotlib inline

plt.plot(range(1, len(average_mae_history) + 1)[5:], average_mae_history[5:])
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
plt.show()
```



It seems that validation MAE stops improving significantly after 50 epochs. We can now train a final "production" model on all of the training data, with the best parameters, then look at its performance on the test data:

In [18]:

```
# Get a fresh, compiled model.
model = build_model()
# Train it on the entirety of the data.
history = model.fit(train_data, train_targets, validation_split =0.2,
          epochs=50, batch_size=1, verbose=1)
```

```
Train on 323 samples, validate on 81 samples
Epoch 1/50
323/323 [==============================] - 1s 4ms/sample - loss: 199.6549 - mae: 10.9164 - val_loss: 54.94
77 - val_mae: 5.2122
Epoch 2/50
323/323 [==============================] - 1s 2ms/sample - loss: 31.0661 - mae: 3.6039 - val_loss: 28.2696
- val_mae: 3.7576
Epoch 3/50
323/323 [==============================] - 1s 2ms/sample - loss: 20.7907 - mae: 2.9457 - val_loss: 19.1979
- val_mae: 3.2109
Epoch 4/50
323/323 [==============================] - 1s 2ms/sample - loss: 18.1605 - mae: 2.6948 - val_loss: 15.4119
- val_mae: 3.0113
Epoch 5/50
323/323 [==============================] - 1s 2ms/sample - loss: 16.2347 - mae: 2.5258 - val_loss: 13.4556
- val_mae: 2.7274
Epoch 6/50
323/323 [==============================] - 1s 2ms/sample - loss: 14.6436 - mae: 2.4808 - val_loss: 12.1205
- val_mae: 2.6882
```

In [19]:

```
print(history.history.keys())
```

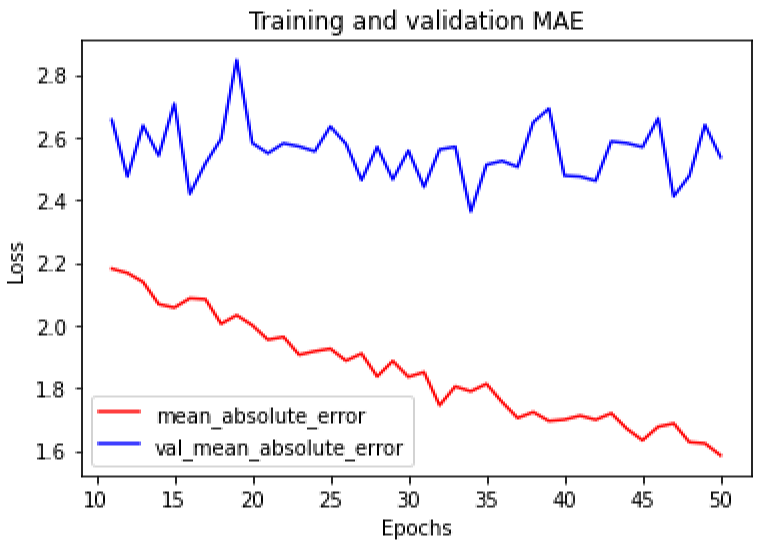
```
dict_keys(['loss', 'mae', 'val_loss', 'val_mae'])
```

In [20]:

```
import matplotlib.pyplot as plt
%matplotlib inline
mae = history.history['mae']
val_mae = history.history['val_mae']

epochs = range(1, len(mae) + 1)

plt.plot(epochs[10:], mae[10:], 'r', label='mean_absolute_error')
plt.plot(epochs[10:], val_mae[10:], 'b', label='val_mean_absolute_error')
plt.title('Training and validation MAE')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



In [21]:
```
test_mse_score, test_mae_score = model.evaluate(test_data, test_targets, verbose=2)
test_mae_score
```

```
102/1 - 0s - loss: 103.6781 - mae: 2.9203
```

Out[21]:
```
2.9203424
```

2. Exercise - tuning model parameters

Please train the above model in the below two scenerios: make the changes on the indicated training configurations (the rest no change). Train both models for 120 epochs.

Scenerio A:

- change the batch size from 1 to 128

Scenerio B:

- change the learning rate (`optimizers.RMSprop(lr=0.001)`) from 0.001 to 0.0002

Observe the training and validation MAE curves for both scenerios.

Provide your codes & observations in the below boxes.

2.1 Scenerio A

In [22]:
```
# Task 1: Build the model and no changes
model = models.Sequential()
model.add(layers.Dense(64, activation='relu',
                        input_shape=(train_data.shape[1],)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(1))
```

In [23]:

```python
# Task 2: Compile and Train the model for 120 epochs. Change the batch size from 1 to 128
model.compile(optimizer=optimizers.RMSprop(lr=0.001), loss='mse', metrics=['mae'])
model.summary()
history = model.fit(train_data, train_targets, validation_split =0.2,
          epochs=120, batch_size=128, verbose=1)
```

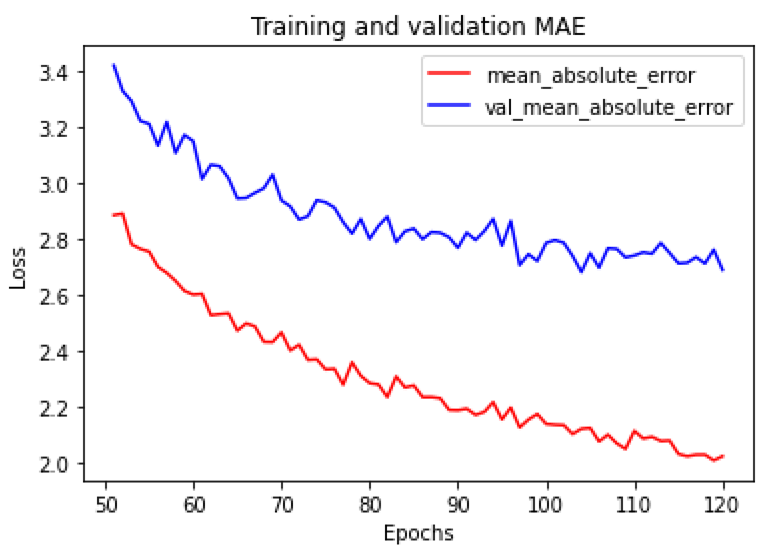
```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_18 (Dense)             (None, 64)                896       
_________________________________________________________________
dense_19 (Dense)             (None, 64)                4160      
_________________________________________________________________
dense_20 (Dense)             (None, 1)                 65        
=================================================================
Total params: 5,121
Trainable params: 5,121
Non-trainable params: 0
_________________________________________________________________
Train on 323 samples, validate on 81 samples
Epoch 1/120
323/323 [==============================] - 1s 3ms/sample - loss: 551.8048 - mae: 21.4818 - val_loss: 603.3
941 - val_mae: 22.6105
Epoch 2/120
```

In [24]:

```
#Task 3: Plot the MAE (train & test) curves
import matplotlib.pyplot as plt
%matplotlib inline
mae = history.history['mae']
val_mae = history.history['val_mae']

epochs = range(1, len(mae) + 1)

plt.plot(epochs[50:], mae[50:], 'r', label='mean_absolute_error')
plt.plot(epochs[50:], val_mae[50:], 'b', label='val_mean_absolute_error')
plt.title('Training and validation MAE')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



- Task 4 (Comments): Increasing the batch size help to reduce the noise or the fluctuation of MAE values during training phase.

2.2 Scenerio B

In [25]:

```python
#Task 1: Build the model
model = models.Sequential()
model.add(layers.Dense(64, activation='relu',
                        input_shape=(train_data.shape[1],)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(1))
```

In [26]:

```python
#Task 2: Compile and Fit the model. Change the learning rate from 0.001 to 0.0002
model.compile(optimizer=optimizers.RMSprop(lr=0.0002), loss='mse', metrics=['mae'])
model.summary()

history = model.fit(train_data, train_targets, validation_split =0.2,
          epochs=120, batch_size=1, verbose=1)
```

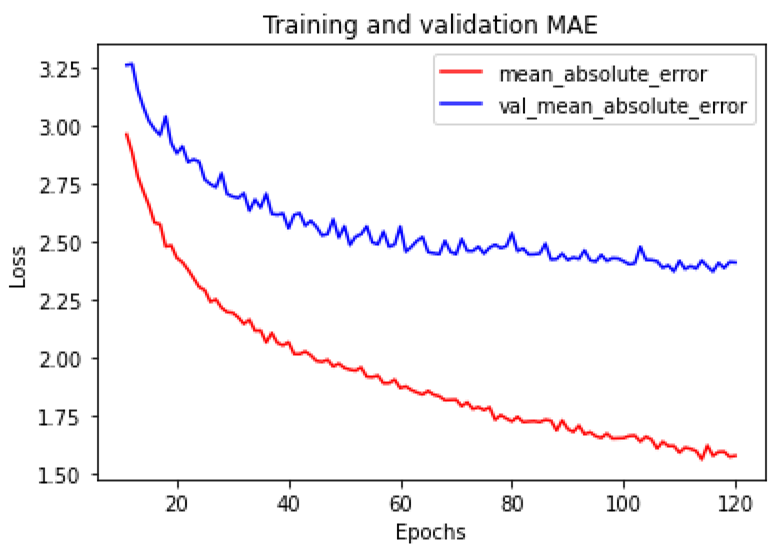
```
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_21 (Dense)             (None, 64)                896       
_________________________________________________________________
dense_22 (Dense)             (None, 64)                4160      
_________________________________________________________________
dense_23 (Dense)             (None, 1)                 65        
=================================================================
Total params: 5,121
Trainable params: 5,121
Non-trainable params: 0
_________________________________________________________________
Train on 323 samples, validate on 81 samples
Epoch 1/120
323/323 [==============================] - 1s 4ms/sample - loss: 460.2875 - mae: 19.3117 - val_loss: 419.8
981 - val_mae: 18.2980
Epoch 2/120
```

In [27]:

```
#Task 3: Plot the MAE (train & test) curves
import matplotlib.pyplot as plt
%matplotlib inline
mae = history.history['mae']
val_mae = history.history['val_mae']

epochs = range(1, len(mae) + 1)

plt.plot(epochs[10:], mae[10:], 'r', label='mean_absolute_error')
plt.plot(epochs[10:], val_mae[10:], 'b', label='val_mean_absolute_error')
plt.title('Training and validation MAE')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



- Task 4 (Comments): Decreasing learning rate in optimizer helps to reduce the noise or the fluctuation of MAE values during training phase.

In [ ]:

```

```