Lecture_4_Boston housing price_regression

October 27, 2022

1 Predicting house prices: A regression example

Previous examples considered classification problems (MNIST and IMDB), where the goal was to predict a single discrete label of an input data point. Another common type of machine learning problem is **regression**, which consists of *predicting a continuous value* instead of a discrete label. For instance, we can predict the temperature tomorrow, given meteorological data or predict the time that a software project will take to complete, given its specifications.

1.1 The Boston Housing Price dataset

In this notebook, we'll address predicting the median price of homes in a given Boston suburb in the mid-1970s, given data points about the suburb at the time, such as the crime rate, the local property tax rate, and so on.

The dataset we'll use has an interesting difference from MNIST and IMDB. It has relatively few data points: only 506, split between 404 training samples and 102 test samples.

Input features are as follows:

- CRIM - per capita crime rate by town
- ZN - proportion of residential land zoned for lots over 25,000 sq.ft.
- INDUS - proportion of non-retail business acres per town.
- CHAS - Charles River dummy variable (1 if tract bounds river; 0 otherwise)
- NOX - nitric oxides concentration (parts per 10 million)
- RM - average number of rooms per dwelling
- AGE - proportion of owner-occupied units built prior to 1940
- DIS - weighted distances to five Boston employment centres
- RAD - index of accessibility to radial highways
- TAX - full-value property-tax rate per 10,000 \$
- PTRATIO - pupil-teacher ratio by town
- B - 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
- LSTAT - % lower status of the population
- MEDV - Median value of owner-occupied homes in 1000's \$

Input data contain 13 numerical features and each feature has a different scale. For instance, some values are proportions, which take values between 0 and 1, others take values between 1 and 12, others between 0 and 100, and so on.

[1]:
```
import numpy as np
from tensorflow import keras
```

```
from tensorflow.keras import layers
from tensorflow.keras.datasets import boston_housing
from matplotlib import pyplot as plt
```

Loading the Boston housing dataset

[2]:
```
(train_data, train_targets), (test_data, test_targets) = boston_housing.
 ↪load_data()
```

[3]:
```
train_data.shape
```

[3]:
```
(404, 13)
```

[4]:
```
test_data.shape
```

[4]:
```
(102, 13)
```

[5]:
```
train_targets[:50]  # prices in thousands of dollars
```

[5]:
```
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])
```

1.2 Preparing the data

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

Normalizing the data

[6]:
```
mean = train_data.mean(axis=0)
train_data -= mean
std = train_data.std(axis=0)
train_data /= std
test_data -= mean
test_data /= std
```

The quantities used for normalizing the test data are computed using the training data. You should **never use any quantity computed on the test data in your workflow**, even for something as simple as data normalization.

1.2.1 Building the model

Model definition

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

[7]:
```
def build_model():
    model = keras.Sequential([
        layers.Dense(64, activation="relu"),
        layers.Dense(64, activation="relu"),
        layers.Dense(1)
    ])
    model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
    return model
```

The model ends with a single unit and no activation (a linear layer). This is a typical setup for scalar regression (a regression where you're trying to predict a single continuous value). Applying an activation function would constrain the range the output can take; for instance, if we applied a sigmoid activation function to the last layer, the model could only learn to predict values between 0 and 1. Here, because the last layer is purely linear, the model is free to learn to predict values in any range.

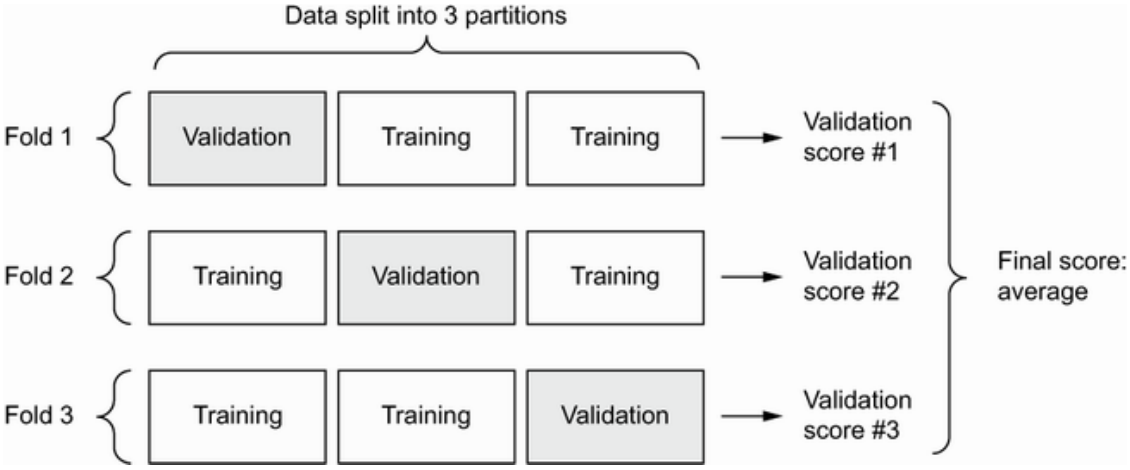
Note that we compile the model with the `mse` loss function — mean squared error, the square of the difference between the predictions and the targets. The MSE is a widely used loss function for **regression problems**.

We're also monitoring a new metric during training: mean absolute error (MAE). It's the absolute value of the difference between the predictions and the targets. For instance, an MAE of 0.5 on this problem would mean that our predictions are off by $500 on average.

1.3 Validating our approach using K-fold validation

To evaluate our model while we keep adjusting its parameters (such as the number of epochs used for training), we could split the data into a training set and a validation set, as we did in the previous examples. But because we have so few data points, the validation set would end up being very small (for instance, about 100 examples). As a consequence, the validation scores might change a lot depending on which data points we chose for validation and which we chose for training: the validation scores might have a high variance with regard to the validation split. This would prevent us from reliably evaluating our model.

The best practice in such situations is to use **K-fold cross-validation**.



It consists of splitting the available data into K partitions (typically $K = 4$ or 5), 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 is then **the average of the K validation scores** obtained. The corresponding code is given below.

K-fold validation

[8]:
```
k = 4
num_val_samples = len(train_data) // k      # // floor division operator,␣
 ↪rounds the result down to the nearest integer
num_epochs = 100
all_scores = []
for i in range(k):
    print(f"Processing fold #{i}")
    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]

    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)

    model = build_model()
    model.fit(partial_train_data,
              partial_train_targets,
              epochs=num_epochs,
              batch_size=16,
              verbose=0)            # verbose=0 - no progress messages will be␣
 ↪shown during the training
```

```
    val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
    all_scores.append(val_mae)
```

```
Processing fold #0
Processing fold #1
Processing fold #2
Processing fold #3
```

[9]:
```
all_scores
```

[9]:
```
[1.7818734645843506, 2.5124921798706055, 2.7510006427764893, 2.439776659011841]
```

[10]:
```
np.mean(all_scores)
```

[10]:
```
2.3712857365608215
```

The different runs do indeed show rather different validation scores. The average is a much more reliable metric than any single score—that's the entire point of K-fold cross-validation. In this case, we're off by 2300\$ on average, which is significant considering that the prices range from 10000\$ to 50000\$.

Saving the validation logs at each fold

Let's try training the model a bit longer: 500 epochs. To keep a record of how well the model does at each epoch, we'll modify the training loop to save the per-epoch validation score log for each fold.

[11]:
```
num_epochs = 500
all_mae_histories = []
for i in range(k):
    print(f"Processing fold #{i}")
    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]

    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)

    model = build_model()
    history = model.fit(partial_train_data,
                        partial_train_targets,
                        validation_data=(val_data, val_targets),
                        epochs=num_epochs,
                        batch_size=16,
```

```
                         verbose=0)
    mae_history = history.history["val_mae"]
    all_mae_histories.append(mae_history)
```

```
Processing fold #0
Processing fold #1
Processing fold #2
Processing fold #3
```

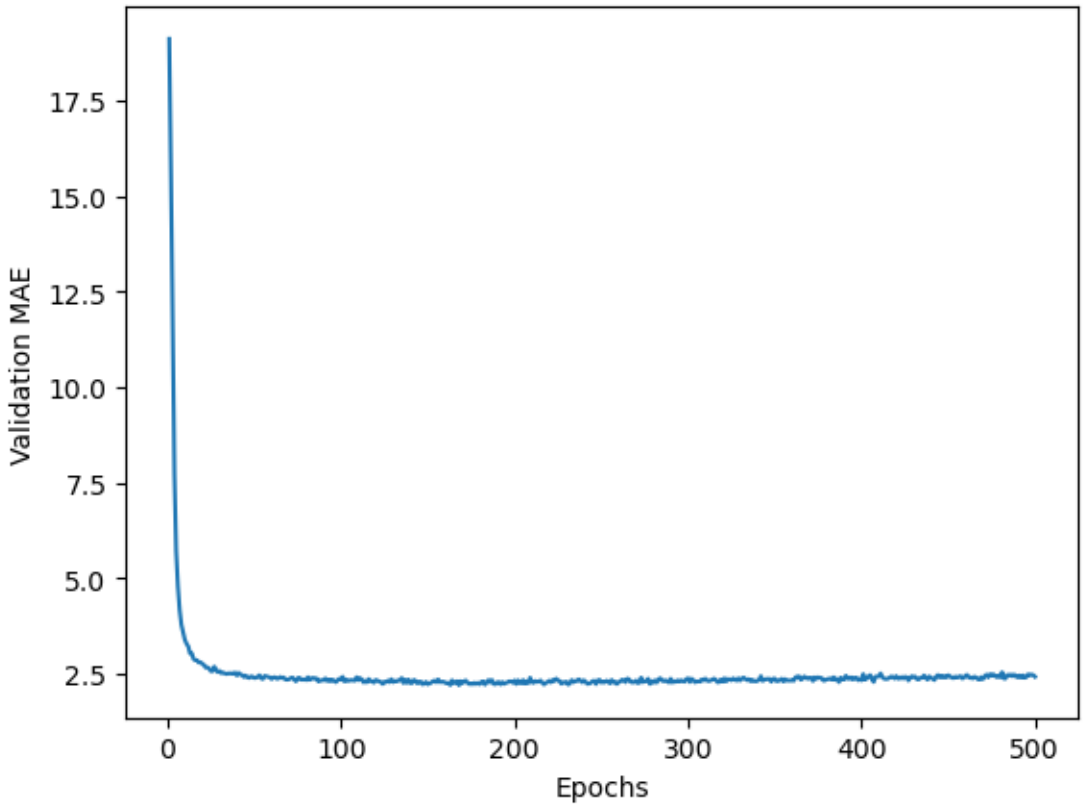
Building the history of successive mean K-fold validation scores

We can now compute the average of the per-epoch MAE scores for all folds.

[12]:
```
average_mae_history = [
    np.mean([x[i] for x in all_mae_histories]) for i in range(num_epochs)]
```

Plotting validation scores

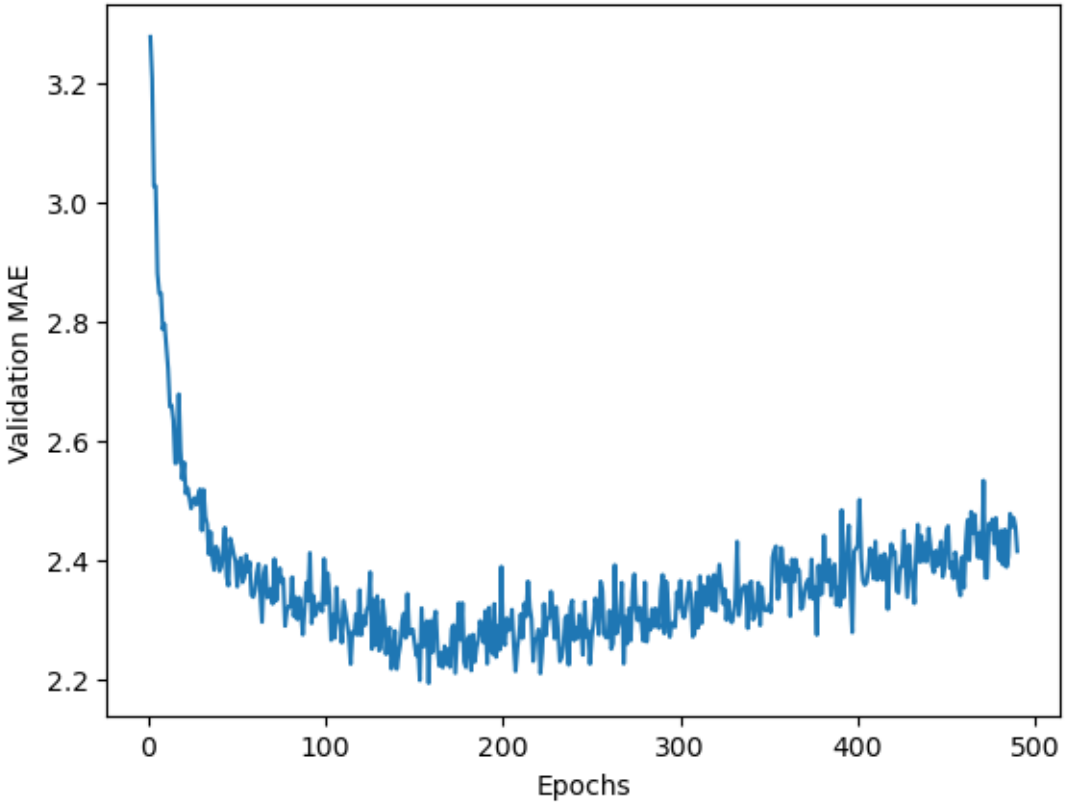
[13]:
```
plt.plot(range(1, len(average_mae_history) + 1), average_mae_history)
plt.xlabel("Epochs")
plt.ylabel("Validation MAE")
plt.show()
```



It's a bit difficult to read the plot, due to a scaling issue: the validation MAE for the first few epochs is dramatically higher than the values that follow.

Let's omit the first 10 data points, which are on a different scale than the rest of the curve.

[14]:
```
truncated_mae_history = average_mae_history[10:]
plt.plot(range(1, len(truncated_mae_history) + 1), truncated_mae_history)
plt.xlabel("Epochs")
plt.ylabel("Validation MAE")
plt.show()
```



As we can see from the above figure, validation MAE stops improving after 120–140 epochs. Past that point, we start overfitting.

Once we've finished tuning other parameters of the model (in addition to the number of epochs, we could also adjust the size of the intermediate layers), we can train a final production model on **all of the training data**, with the best parameters, and then look at its **performance on the test data**.

Training the final model

[15]:
```
model = build_model()
model.fit(train_data,
```

```
          train_targets,
          epochs=130,      # at approx epochs=130 Validation MAE reaches minimum
          batch_size=16,
          verbose=0)
test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)
```

```
4/4 [==============================] - 0s 2ms/step - loss: 15.1333 - mae: 2.5278
```

[16]:
```
test_mae_score
```

[16]:
```
2.527799129486084
```

1.3.1 Generating predictions on new data

[17]:
```
predictions = model.predict(test_data)
print("Actual:  Predicted:")
for i in range(20):
    print(f'{predictions[i][0]:<8.2f} {test_targets[i]:.2f}')
```

```
4/4 [==============================] - 0s 1ms/step
Actual:  Predicted:
8.14     7.20
18.23    18.80
21.12    19.00
35.35    27.00
24.23    22.20
22.12    24.50
26.44    31.20
21.02    22.90
18.53    20.50
22.13    23.20
17.89    18.60
16.21    14.50
15.78    17.80
42.02    50.00
20.56    20.80
20.27    24.30
25.00    24.20
18.19    19.80
19.42    19.10
22.92    22.70
```