Experiment 2

Fitting a Regression Model with Neural Networks

2.1 Regression Modeling of California Housing Prices

In this experiment, we'll build a neural network that predicts housing values using data for houses in California from the 1990 U.S. census. The California housing dataset, one among the several built-in datasets of Keras, contains median home value, in dollars, for $20,640$ districts (known as **block groups** by the U.S. Census Bureau) in California. A district is described by eight input features:

- latitude
- longitude
- median age of homes in the district
- population (average population is a little larger than 1400)
- number of households
- median income of households
- total number of rooms in all homes
- total number of bedrooms in all homes

Median homes

2.1.1 Conventions

Code cells are briefly described in the text cell that immediately precedes them. A code cell and its accompanying text cell are labeled as Block nn.

Comment lines beginning with `##` indicate places in code cells at which you should enter a comment as described in the procedure.

Write comments so that they are informative to someone with Python coding experience similar to your own.

2.1.2 Importing packages and defining functions

Run the code cells labeled Block 1, Block 2, and Block 3.

As in Experiment 1, the `import` statements are put into two code cells, labeled Block 1 and Block 2, for convenience. Add `import` statements for any additional packages to Block 1 to preserve this convenience.

Insert a code cell above Block 3. Using your exploration of the IMDB dataset as a guide, add one or more statements whose execution helps you get acquainted with the California housing dataset. Ignoring the dunder methods, what method or methods are shared by the two datasets? What method or methods are found in only one of the datasets? Answer these questions in a comment following the last statement of the code block you added. Are the similarities and differences between the two sets of methods consistent with the similarities and differences between the datasets? Why or why not?

Functions that we want to be available throughout the session are defined in Block 3. A sample in the dataset is a vector, so it is already in sequential form, like a sample in the IMDB dataset but unlike the images in the MNIST dataset. The output layer produces a scalar, like the output layer of the models in Experiment 1, but unlike the output layer in the models used to categorize MNIST images. Although both the models of this experiment and those of Experiment 1 produce a scalar, the scalar is processed differently. More will be said about this in the section describing the model. The arguments to `compile` all differ from those of Experiment 1, with `loss` and `metrics` determined by the nature of the input and target data. The difference between `RMSprop` and `adam` is that `adam` adjusts how large a step is taken as the model parameters are adjusted, which usually accelerates the rate at which the loss function decreases.

2.1.3 Loading the California housing dataset

Run the code cell labeled Block 4.

The `load_data` method of `california_housing` has `test_split` as a parameter. This parameter determines how the full dataset is divided between the training and testing dataset. Insert a code cell above Block 5. Write a code snippet that uses the `shape` attribute of `train_data` and `test_data` to

- print a statement that can be used to confirm that a sample in the dataset is described by eight features; and
- calculate and report the default value of `test_split`, expressed as a floating point number with one decimal place.

Explore `train_data` and `train_targets` using NumPy functions like `min`, `max`, `mean`, `median`, and `stdev`. Clean up your code snippet by eliminating any functions whose output you do not find informative and by embedding the remaining functions in print statements that, taken together, display a short summary of the data set.

2.1.4 Preparing the data

Your exploration of the dataset will have revealed that the features of the input data and the targets are different in both their nature and in their magnitude. This type of data is called **heterogeneous**. For example, latitude and longitude are angles, measured in degrees, which range from about 32.5° to 42° N and from about 114° to 124.5° W, respectively, in California, while median income is measured in dollars and ranges from under $10,000 to nearly $10,000,000.

You have seen some techniques for handling heterogeneous data in Machine Learning Fundamentals. Such techniques are known as **normalization**. A common normalization technique shifts each feature independently so it has a mean of zero (so it is centered at 0) and scaling each feature independently so that its distribution has a standard deviation of one. If the data before normalization followed a normal distribution, this shifting and scaling would result in the data following the standard normal distribution.

The mean and standard deviation of the training data are used to normalize the testing data as well. If the testing data were normalized against itself, information about the testing data would leak into the training set through the calculation of the loss, when the predictions of the model are compared to ground truth. It is essential to prevent such leakage. As you have seen, loss goes to zero and accuracy goes to 100 % if the model is trained for enough epochs. If testing data leaks into the training data and the model is trained long enough, the model will "learn" the features of both the training and the testing data and have unwarranted predictive power. Performance of the model on truly new data would be unexpectedly poor, which would not only cast doubt on the model (appropriately, in this case) but also cast doubt on the whole process (much less appropriately, as a general conclusion).

The normalized training and testing data will provide inputs to the model in a fairly narrow range around zero. The parameters of the model (the weights) begin with randomly generated small values. Under these circumstances, multiplying the parameters by the inputs will produce a small predicted values, at least early in the training process. The training and testing targets are between $60,000 and $500,000, which are much larger than the values of the normalized inputs. If the targets are not normalized, it will take many, many epochs to train the model. Dividing the targets by $100,000$ is a simple but effective normalization method that scales the targets from the range $60,000$ to $500,000$ to the range 0.6 to 5. Multiplying the predictions by $100,000$ restores the predictions to dollar values.

Run the code cell labeled Block 5. Insert a code cell above Block 6. Write a code snippet that confirms that the features of the input data have been normalized using `np.mean` and `np.std`. As for Block 4, clean up your code snippet by eliminating any functions whose output you do not find informative and by embedding the remaining functions in print statements that, taken together, will convince someone reading the notebook that the input data has been normalized.

Based on the discussion of data leakage, do you expect the testing dataset to be normalized? Is the testing dataset normalized? Write a comment that explains your expectation and what you did to confirm or refute it.

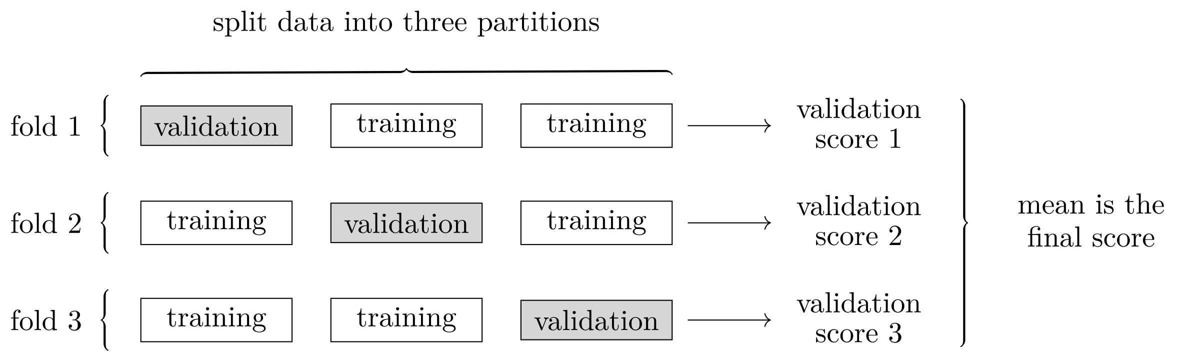
2.1.5 Building, training, and assesssing the model

Given the small size of the data set, a very small model is called for. As a general rule, overfitting is more likely when working with a small data set. Using a small model is the standard way to avoid this pitfall. As in Experiment 1, I have adopted the recommendations of Chollet, whose code in *Deep Learning with Python, 3e* is the inspiration for this The model has two hidden layers, each dense, each having 64 nodes, and each employing ReLU activation, and a dense output layer having a single node (the predicted home value) and no activation. We are not mapping the output onto categories, as is done in classification problems, and the single predicted value can take on a wide range because home values have a wide range, so allowing the predictions to pass through unaltered is appropriate.

2.1.6 Validating the model

The dataset is too small to use the validation techniques of Experiment 1 (recall that about 20 % of the training dataset is set aside as a validation dataset), so the validation dataset would have only about 100 samples for the small version of the California housing dataset. Using such a small validation dataset would result in validation scores that depend too strongly on the particular samples in the validation dataset. A model cannot be evaluated reliably if the validation scores have a large variance.

The cure is K-fold cross-validation, which is illustrated in the accompanying figure. The training dataset is divided into K subsets, called **partitions**, with one partition being used for validation, the remaining subsets being used for training, and a validation score being calculated. Each partition is used in turn as the validation dataset, leading to K validation scores. The final score used to evaluate the model is the mean of the validation scores.



Run the code cells labeled Block 6A and Block 6B. Look up **relative standard deviation** and add a comment at the end of the code cell in Block 6B that interprets the RSD of this model.

2.1.7 Building and training the model, slight return

It is worth checking whether increasing the number of epochs (training the model longer) produces a better model. The `fit` method of `model` maintains a log of the epoch The `fit` method of a Keras model creates a History object with a history dictionary whose keys are named following the metrics for the training and validation stages (e.g., `loss`) and whose values are lists of the metrics after each epoch. These values can be plotted versus the epoch number to visualize the progress of the fit.

Run the code cell labeled Block 7.

2.1.8 Summarizing and visualizing the training history

Run the code cells labeled Block 8 and Block 9.

The first execution of the code cell of Block 9 produces a graph that makes it difficult to locate the minimum on the curve due to scaling issues. The large drop in MAE during the first few epochs obscures the detail in the curve that must be seen to associate an epoch with the minimum loss. Vary `ifirst`, rerunning the code cell after each change, until you can make a reasonable estimate of the location of the minimum on the curve.

2.1.9 Retraining the model, from the ground up

Run the code cell labeled Block 10.

2.1.10 Making predictions using the retrained model

Run the code cell labeled Block 11. Vary the value of `plot_sf` to give appropriate prominence to the tick mark labels. Add a text cell following the figure commenting on your choice of `plot_sf`. Be sure to adjust the axes labels to account for any change in `plot_sf`.