Deep Machine Learning
&
Regression Example

預測房價：迴歸範例
3.7-predicting-house-prices.ipynb

Predicting house prices: a regression example

- Another common type of machine-learning problem is *regression*（預測）, which consists of predicting a continuous value instead of a discrete label.

- ***The Boston Housing Price dataset***
  - It has relatively few data points: only 506,
  - Split between 404 training samples and 102 test samples.
  - And each ***feature*** in the input data (for example, the crime rate) 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.

Loading the Boston housing dataset

```
from keras.datasets import boston_housing
(train_data, train_targets), (test_data, test_targets) =
boston_housing.load_data()
```

- It contains, 13 numerical features, such as per capita crime rate, average number of rooms per dwelling, accessibility to highways, and so on.

```
>>> train_data.shape
(404, 13)
>>> test_data.shape
(102, 13)
>>> train_targets
[ 15.2, 42.3, 50. ... 19.4, 19.4, 29.1]
```

Attributes

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.

Output of train_targets

```
In [0]: train_targets   #訓練資料的標籤

Out[56]: 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,
                21.8, 26.4, 14.9, 24.1, 23.8, 12.3, 29.1, 21. , 19.5, 23.3, 23.8,
                17.8, 11.5, 21.7, 19.9, 25. , 33.4, 28.5, 21.4, 24.3, 27.5, 33.1,
                16.2, 23.3, 48.3, 22.9, 22.8, 13.1, 12.7, 22.6, 15. , 15.3, 10.5,
                24. , 18.5, 21.7, 19.5, 33.2, 23.2,  5. , 19.1, 12.7, 22.3, 10.2,
                13.9, 16.3, 17. , 20.1, 29.9, 17.2, 37.3, 45.4, 17.8, 23.2, 29. ,
                22. , 18. , 17.4, 34.6, 20.1, 25. , 15.6, 24.8, 28.2, 21.2, 21.4,
                23.8, 31. , 26.2, 17.4, 37.9, 17.5, 20. ,  8.3, 23.9,  8.4, 13.8,
                 7.2, 11.7, 17.1, 21.6, 50. , 16.1, 20.4, 20.6, 21.4, 20.6, 36.5,
                 8.5, 24.8, 10.8, 21.9, 17.3, 18.9, 36.2, 14.9, 18.2, 33.3, 21.8,
                19.7, 31.6, 24.8, 19.4, 22.8,  7.5, 44.8, 16.8, 18.7, 50. , 50. ,
                19.5, 20.1, 50. , 17.2, 20.8, 19.3, 41.3, 20.4, 20.5, 13.8, 16.5,
                23.9, 20.6, 31.5, 23.3, 16.8, 14. , 33.8, 36.1, 12.8, 18.3, 18.7,
                19.1, 29. , 30.1, 50. , 50. , 22. , 11.9, 37.6, 50. , 22.7, 20.8,
                23.5, 27.9, 50. , 19.3, 23.9, 22.6, 15.2, 21.7, 19.2, 43.8, 20.3,
                33.2, 19.9, 22.5, 32.7, 22. , 17.1, 19. , 15. , 16.1, 25.1, 23.7,
                28.7, 37.2, 22.6, 16.4, 25. , 29.8, 22.1, 17.4, 18.1, 30.3, 17.5,
                24.7, 12.6, 26.5, 28.7, 13.3, 10.4, 24.4, 23. , 20. , 17.8,  7. ,
                11.8, 24.4, 13.8, 19.4, 25.2, 19.4, 19.4, 29.1])
```

Preparing the data

- **Normalizing**（常態化）the data (Numpy lib)

```
mean = train_data.mean(axis=0)  #average or mean
train_data -= mean #每一個值減去平均數  x-mean
std = train_data.std(axis=0)  #計算 standard derivation
train_data /= std # normalization

test_data -= mean
test_data /= std
```

Building Neural Network

- The network ends with a single unit (單一輸出) and **no activation** (it will be a linear layer) for scalar regression.
- Because the last layer is purely linear, the network is free to learn to predict values in any range. (**softmax [0, 1]**)
- **MSE** loss function— ***mean squared error*(均方誤差)**, the square of the difference between the predictions and the targets. This is a widely used loss function for regression problems.

$$\mathrm{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

- **RMSE : Root Mean Square Error / 均方根誤差**

$$\mathrm{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2} = \sqrt{\mathrm{MSE}}$$

Model Definition

```
from keras import models
from keras import layers

def build_model():
    model = models.Sequential()  #建構一個 sequential 模型
    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
```

$$\mathrm{MAE} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

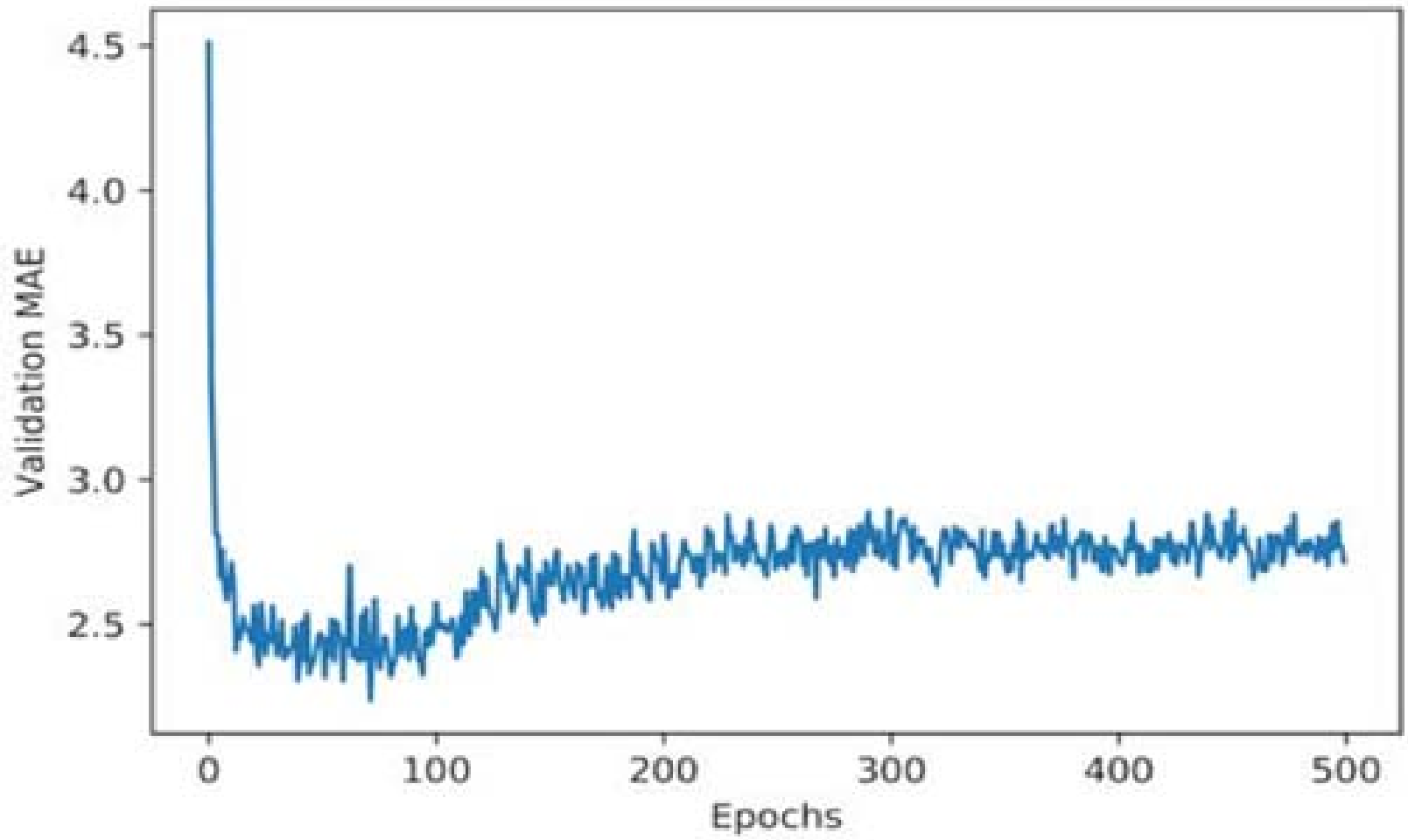
- **MAE : Mean Absolute Error / 平均絕對誤差**
  - 常用在財務方面, 又稱 L1 損失, MAE 就是將每次測量的絕對誤差取絕對值後求的平均值, 用來觀察預測值誤差的測量是否要調整。
- **rmsprop**:
  - 每一次更新learning rate時，分母所除的σ都與前一次的有關係，調整上面多了一個參數 α，可以自由調整新舊gradient的比重（影響力）

RMSProp

- **RMSprop**
  - **學習演算法用以處理複雜 error surface**
  - 現實中常會碰到的 Loss function 並非都是平穩、簡單的，甚至絕大多數我們遇到的 Error surface 都非常複雜。
  - 即使在同一個維度上，學習率都有可能必須要能夠快速的反應、變動
  - Hinton 提出
  - RMSprop 在 學習率調整上面多了一個參數 α，可以在新舊梯度上面做調節

- **優點**
  - 有效改善 Adagrad 提前結束訓練的問題。
  - 適合處理複雜的、non-convex 的 error surface。
- **缺點**
  - 仍然需要先設置一個全局學習率 η

Validation MAE by epoch



Training the final result

```
model = build_model()  #建立一個用最佳參數 compile 過的新模型
model.fit(train_data, train_targets,
epochs=80, batch_size=16, verbose=0)
test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)
#以整個資料進行訓練
```

- Result

```
>>> test_mae_score
2.5532484335057877
```

Evaluating machine-learning models

- We split the data into a **training set**, a **validation set**, and a **test set**.
- The reason not to evaluate the models on the same data they were trained on quickly became evident: after just a few epochs, all three models began to *overfit*.
- In machine learning, the goal is to achieve models that *generalize* —that perform well on **never-before-seen data** —and overfitting is the central obstacle.
- It's crucial to be able to reliably measure the generalization power of your model.
- **How to measure generalization: how to evaluate machine-learning models**

training, validation, and test

- Evaluating a model always boils down to splitting the available data into three sets: **training**, **validation**, and **test**.

- You **train** on the training data and **evaluate** your model on the validation data. Once your model is ready for prime time, you **test** it one final time on the test data.

- Developing a model always involves tuning its configuration: for example, choosing the **number of layers** or the **size of the layers** (called the *hyperparameters* of the model, to distinguish them from the *parameters*, which are the network's weights).

Information leaks

- Why two sets? (trainning and validation sets) You do this tuning by using as a feedback signal the performance of the model on the validation data.

- In essence, this tuning is a form of *learning*: a search for a good configuration in some parameter space. As a result, tuning the configuration of the model based on its performance on the validation set can quickly result in *overfitting to the validation set*, even though your model is never directly trained on it. Central to this phenomenon is the notion of ***information leaks***.

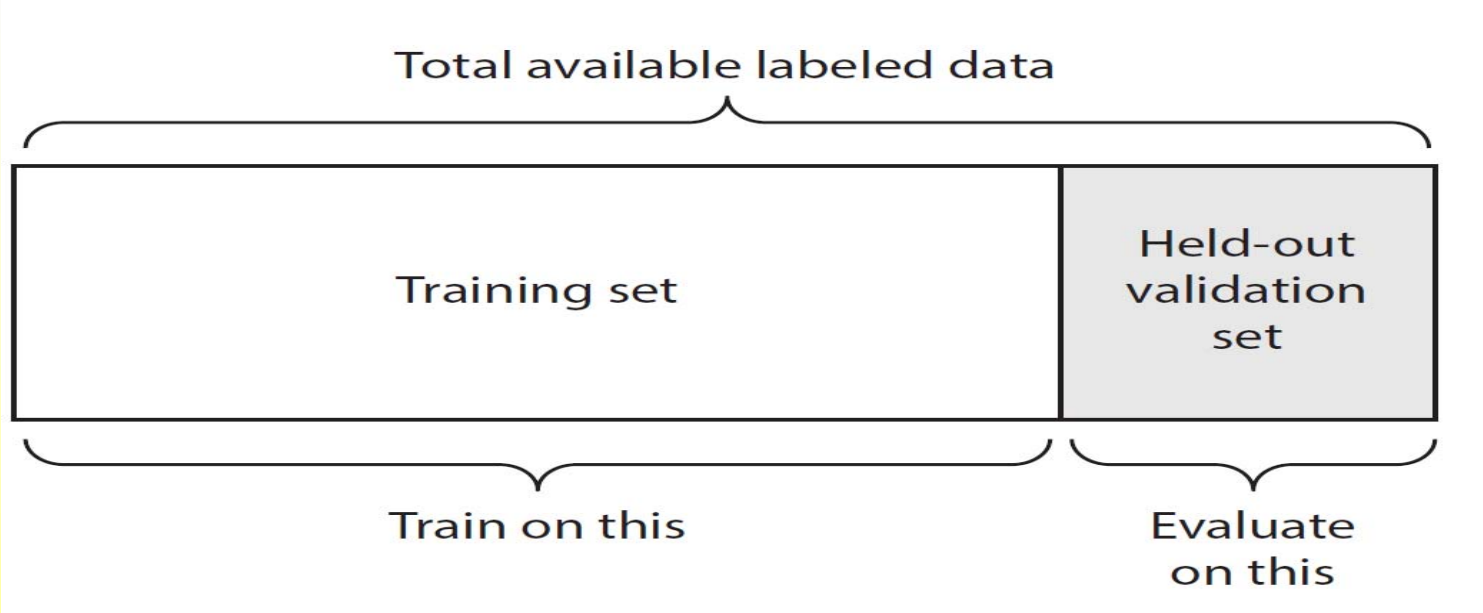
Validating Approach

Evaluation Recipes

- Three classic evaluation recipes:
  - Simple hold-out validation,
  - K-fold validation,
  - Iterated K-fold validation with shuffling.

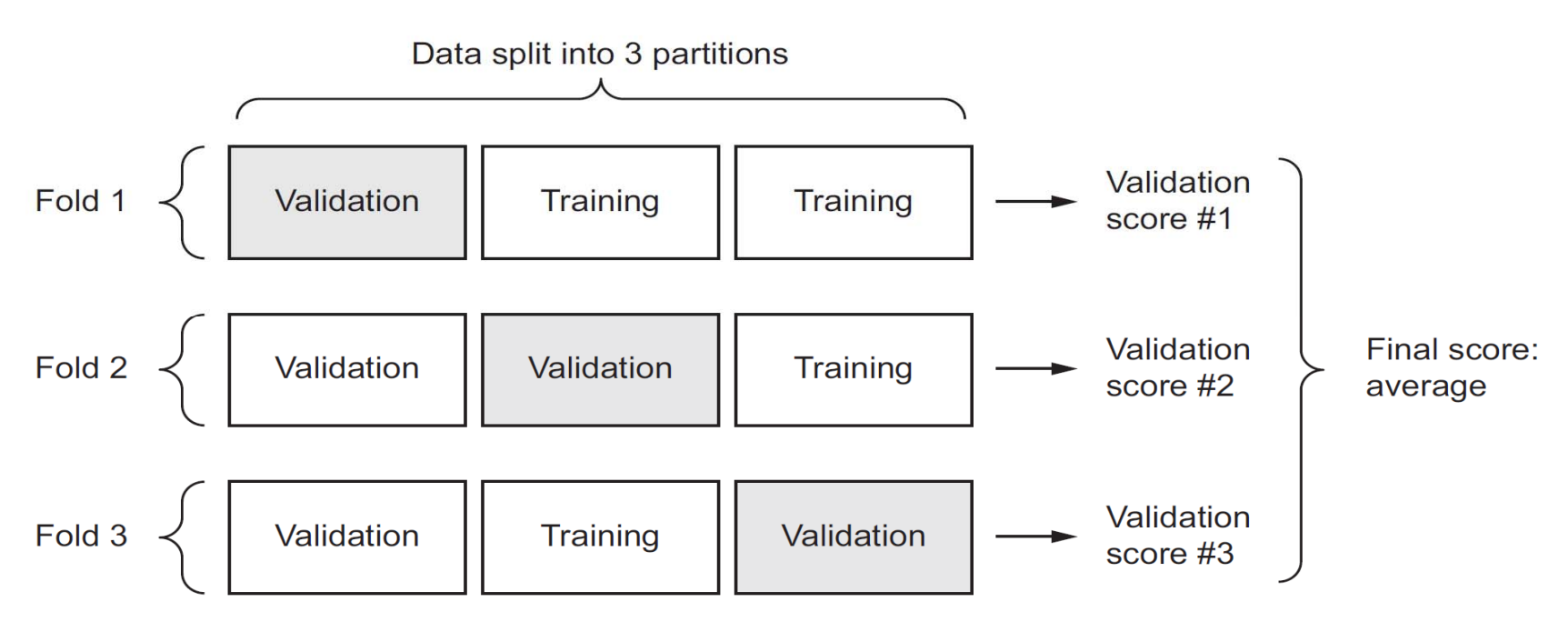
Simple Hold-out Validation

- **Simple Hold-out Validation**
  - Set apart some fraction of your data as your test set.
  - simplest evaluation protocol
  - if little data is available, then your validation and test sets may contain too few samples to be statistically representative of the data at hand.



K-fold Validation

- Split your data into ***K*** partitions of equal size.
- For each partition **i**, train a model on the remaining ***K* – 1** partitions, and evaluate it on partition **i**.
- Your final score is then the **averages of the *K* scores obtained**.



Validating your approach using K-fold validation

- To evaluate your network while you keep adjusting its parameters you could split the data into a **training set** and a **validation set**.
- But because you have so few data points, the validation set would end up being very small.
- As a consequence, the **validation scores** might change a lot depending on which data points you chose to use for validation and which you chose for training: the validation scores might have a **high *variance*** with regard to the validation split.
- 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**.

Iterated K-fold Validation with Shuffling

- This one is for situations in which you have relatively **little data** available and you need to evaluate your model as precisely as possible.
- It is helpful in Kaggle competitions.
- It consists of **applying K-fold validation multiple times**, shuffling (洗牌) the data every time before splitting it *K* ways.
- The final score is the average of the scores obtained at each run of K-fold validation.
- Note that you end up training and evaluating $P \times K$ models (where P is the **number of iterations** you use), which can very expensive.

Notes for Iterated K-fold

- ***Data representativeness***
  - You want both your training set and test set to be representative of the data at hand.
  - For this reason, you usually should *randomly shuffle* your data before splitting it into training and test sets.
- ***The arrow of time***
  - If you're trying to predict the future given the past (for example, tomorrow's weather, stock movements, and so on), you should *not* randomly shuffle your data before splitting it, because doing so will create a *temporal leak*:
  - You should always make sure all data in your test set is *posterior* to the data in the training set.
- ***Redundancy in your data***
  - If some data points in your data appear twice (fairly common with real-world data)
  - Make sure your training set and validation set are disjoint.

Data preprocessing, feature engineering, and feature learning

Data preprocessing, feature engineering, and feature learning

- How do you prepare the input data and targets before feeding them into a neural network?
- Many **data-preprocessing** and **feature-engineering** techniques are domain specific.

- ***Data preprocessing for neural networks*** includes
  - **vectorization,**
  - **normalization,**
  - **handling missing values, and**
  - **feature extraction.**

Vectorization

- All inputs and targets in a neural network must be tensors of floating-point data (or, in specific cases, tensors of integers).
- Whatever data you need to process—**sound**, **images**, **text**—you must first turn into tensors, a step called *data vectorization*.
- **Text-classification examples**: we started from text represented as lists of integers (standing for sequences of words), and we used one-hot encoding to turn them into a tensor of float32 data.

Value Normalization

- **digit-classification example**
  - you started from image data encoded as integers in the 0 - 255 range, encoding grayscale values.
  - Before you fed this data into your network, you had to cast it to float32 and divide by 255 so you'd end up with floating point values in the 0 - 1 range.
- **predicting house prices example**
  - Had to **normalize** each feature independently so that it had a standard deviation of 1 and a mean of 0.
- To make learning easier for your network, your data should have the following characteristics:
  - *Take small values* —Typically, most values should be in the 0 - 1 range.
  - *Be homogenous* —That is, all features should take values in roughly the same range.

```
x -= x.mean(axis=0)
x /= x.std(axis=0)
```

Assuming x is a 2D data matrix of shape (samples, features)

Handling Missing Values

- You may sometimes have missing values in your data.
- What if this feature wasn't available for all samples? You'd then have missing values in the training or test data.
- In general, with neural networks, **it's safe to input missing values as 0**, with the condition that 0 isn't already a meaningful value.
- The network will learn from exposure to the data that the value 0 means *missing data* and will start ignoring the value.

Feature Engineering

- *Feature engineering* is the process of using your own knowledge about the data and about the machine-learning algorithm at hand (in this case, a neural network) to make the algorithm work better by applying hardcoded (nonlearned) transformations to the data before it goes into the model.
- In many cases, it isn't reasonable to expect a machine learning model to be able to learn from completely arbitrary data.

Feature engineering

- Raw pixels of the image as input data and use CNN to solve it
- The black pixels of the clock hands and output the (x, y) coordinates of the tip of each hand. Then a simple machine-learning algorithm can learn to associate these coordinates with the appropriate time of day.

Raw data: pixel grid		
Better features: clock hands' coordinates	{x1: 0.7, y1: 0.7} {x2: 0.5, y2: 0.0}	{x1: 0.0, y2: 1.0} {x2: -0.38, 2: 0.32}
Even better features: angles of clock hands	theta1: 45 theta2: 0	theta1: 90 theta2: 140

Reducing The Network's Size

- The simplest way to **prevent overfitting** is to **reduce the size of the model**: the number of learnable parameters in the model (which is determined by the number of layers and the number of units per layer).
- Intuitively, a model with more parameters has more *memorization capacity* and therefore can easily learn a perfect **dictionary-like mapping** between training samples and their targets—a mapping **without any generalization power**.

- Deep learning models tend to be good at fitting to the training data, but the real challenge is generalization, not fitting.
- There is a compromise to be found **between *too much capacity* and *not enough capacity***.
- No magical formula to determine the right number of layers or the right size for each layer.

Comparing Models

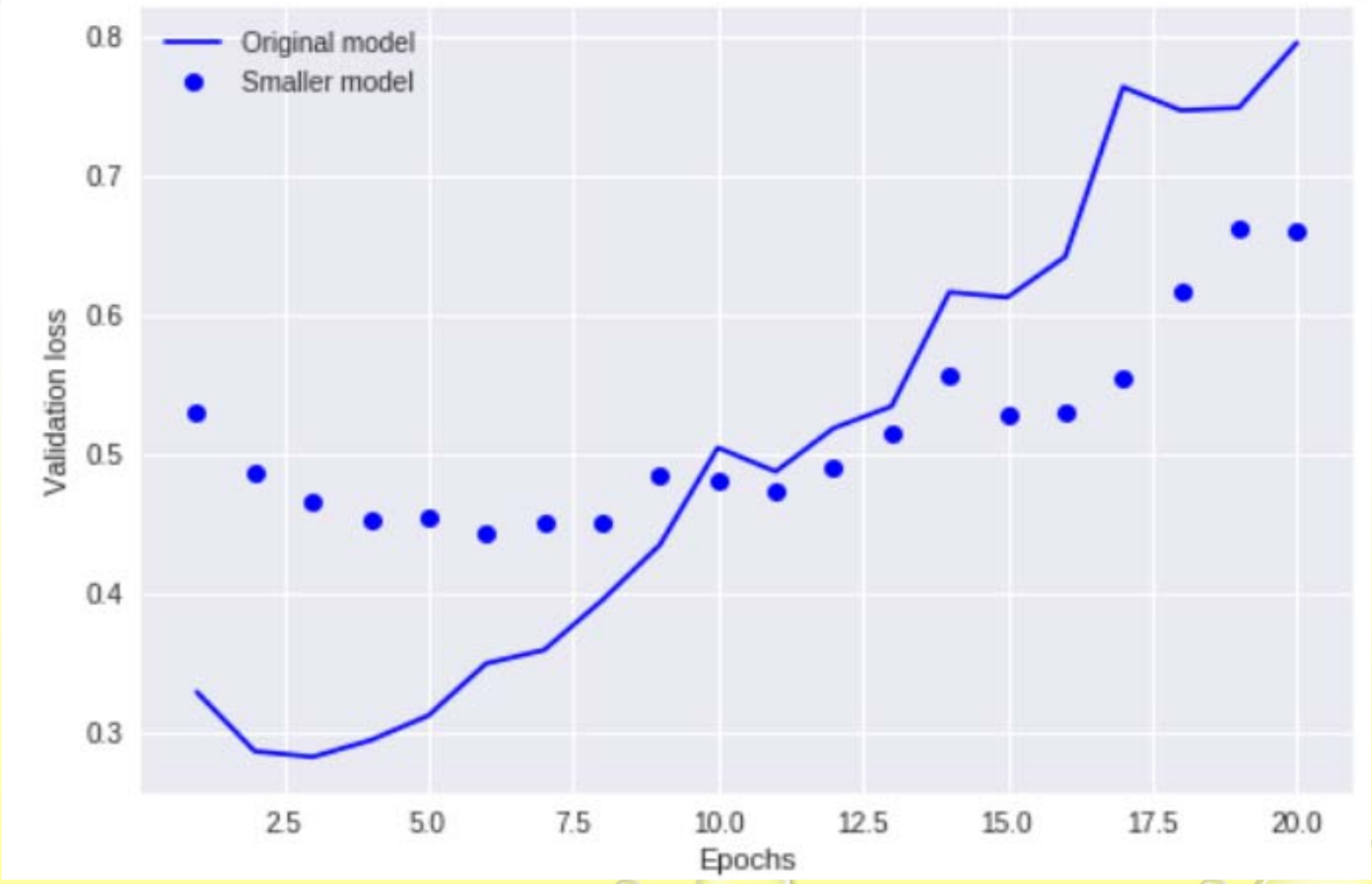
原始模型

```
from keras import models
from keras import layers

original_model = models.Sequential()
original_model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
#原始的為 16 個單元
original_model.add(layers.Dense(16, activation='relu'))
original_model.add(layers.Dense(1, activation='sigmoid'))

original_model.compile(optimizer='rmsprop',
                       loss='binary_crossentropy',
                       metrics=['acc'])
```

容量較低的模型

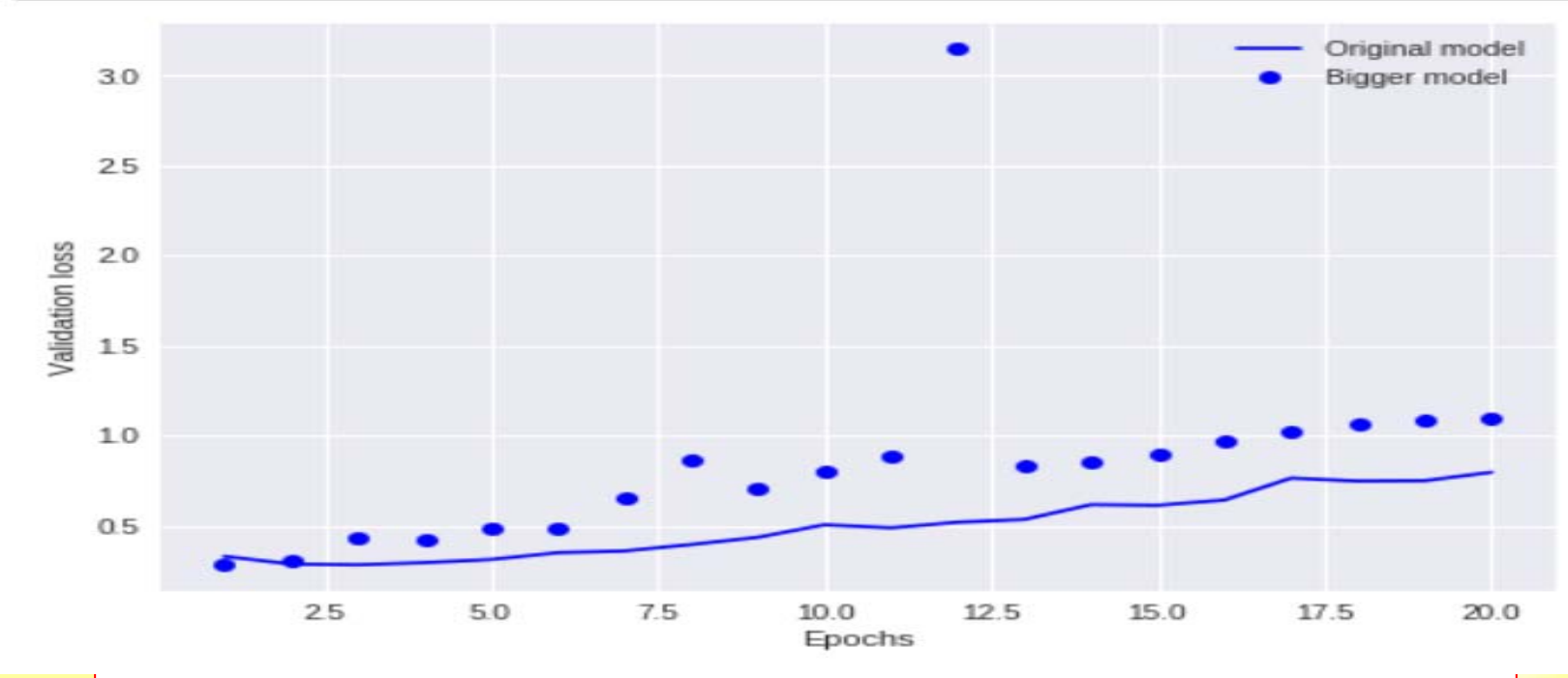
```
smaller_model = models.Sequential()
smaller_model.add(layers.Dense(4, activation='relu', input_shape=(10000,)))
#改成容量較低的 4 個單元
smaller_model.add(layers.Dense(4, activation='relu'))
smaller_model.add(layers.Dense(1, activation='sigmoid'))

smaller_model.compile(optimizer='rmsprop',
                      loss='binary_crossentropy',
                      metrics=['acc'])
```

Effect of model capacity on validation loss: trying a smaller model

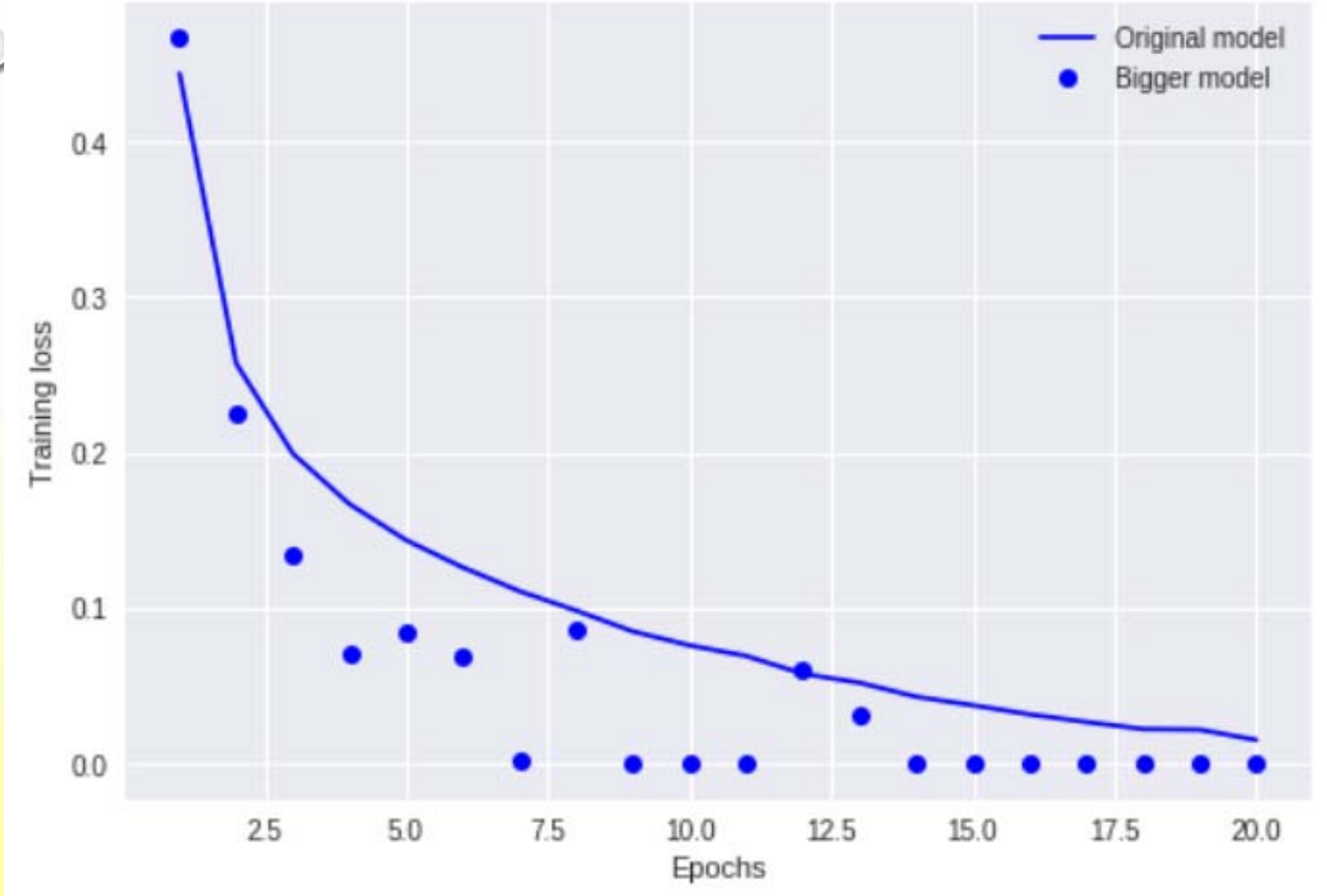


具有更高容量的模型

```
bigger_model = models.Sequential()
bigger_model.add(layers.Dense(512, activation='relu', input_shape=(10000,)))
#改以更高容量的 512 個輸出單位
bigger_model.add(layers.Dense(512, activation='relu'))
bigger_model.add(layers.Dense(1, activation='sigmoid'))

bigger_model.compile(optimizer='rmsprop',
                     loss='binary_crossentropy',
                     metrics=['acc'])
```



Training Loss



Adding Dropout

- ***Dropout*** developed by **Geoff Hinton** and his students at the University of Toronto.
- The most effective and most commonly used **regularization techniques**.
- Dropout, applied to a layer, consists of randomly *dropping out* (setting to zero) a number of output features of the layer during training.
- Return a **vector** [0.2, 0.5, 1.3, 0.8, 1.1] for a given input sample during training.
- After applying dropout, [0, 0.5, 1.3, 0, 1.1].
- The *dropout rate* is the fraction of the features that are zeroed out; it's usually set between 0.2 and 0.5.

將 Dropout 層添加到 IMDB 神經網路

```
dpt_model = models.Sequential()
dpt_model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
dpt_model.add(layers.Dropout(0.5))
dpt_model.add(layers.Dense(16, activation='relu'))
dpt_model.add(layers.Dropout(0.5))
dpt_model.add(layers.Dense(1, activation='sigmoid'))

dpt_model.compile(optimizer='rmsprop',
                  loss='binary_crossentropy',
                  metrics=['acc'])
```