



4장 *Fundamentals of machine learning*

“지혜로운 자는 꿈을 확신하며 준비한다...”



This chapter covers



- Forms of machine learning beyond classification and regression
- Formal evaluation procedures for machine-learning models
- Preparing data for deep learning
- Feature engineering
- Tackling overfitting
- The universal workflow for approaching machine-learning problems

4.1 Four branches of machine learning

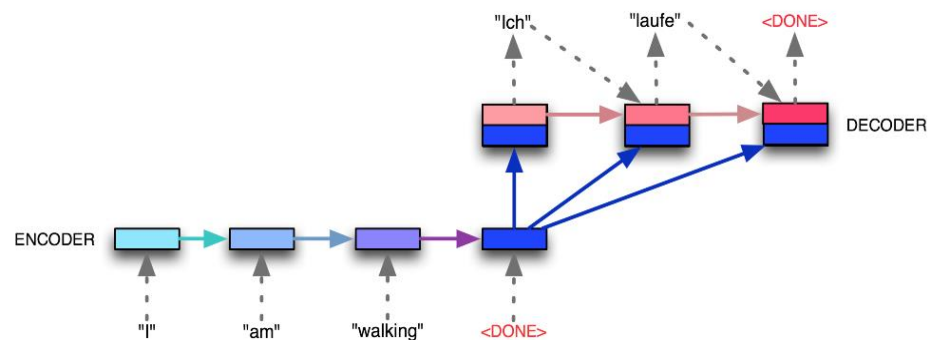
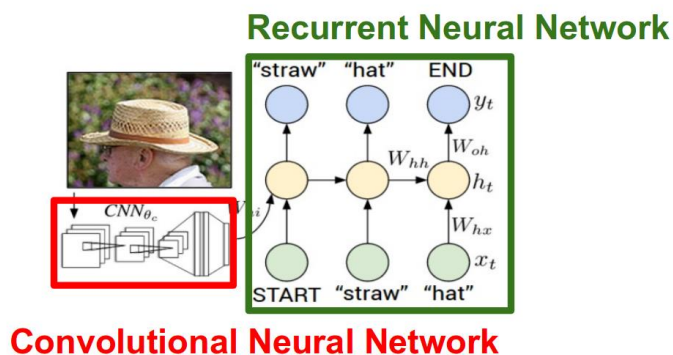
- ▶ In our previous examples: binary classification, multiclass classification, and scalar regression.
- ▶ All three are instances of *supervised learning*, where the goal is to learn the relationship between **training inputs and training targets**.
- ▶ Supervised learning is just the tip of the iceberg—machine learning is a vast field with a complex subfield taxonomy.
- ▶ Machine-learning algorithms generally fall into **four broad categories**, described in the following sections.

4.1 Four branches of machine learning

4.1.1 Supervised learning

- ▶ map input data to known targets (also called *annotations*), given a set of examples (often annotated by humans)
- ▶ optical character recognition, speech recognition, image classification, and language translation
- ▶ more exotic variants as well, including the following (with examples):
 - *Sequence generation* — predict a caption describing it, repeatedly predicting a word or token in a sequence

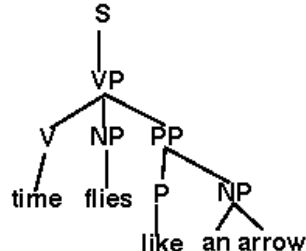
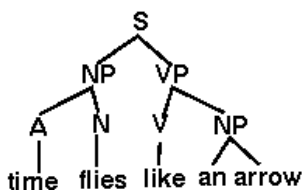
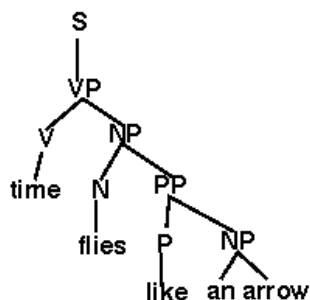
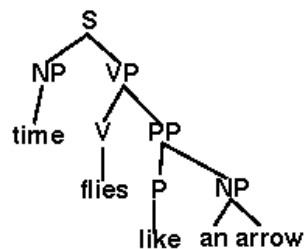
Describing images



4.1 Four branches of machine learning

4.1.1 Supervised learning

- *Syntax tree prediction*—Given a sentence, predict its decomposition into a syntax tree.
- *Object detection*—Given a picture, draw a **bounding box** around certain objects inside the picture. This can also be expressed as a **classification** problem or as a joint classification and regression problem, where the **bounding-box coordinates** are predicted via vector **regression**.
- *Image segmentation*—Given a picture, draw a pixel-level mask on a specific object.



Classification



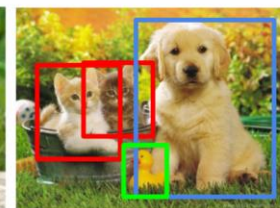
CAT

Classification
+ Localization



CAT

Object Detection



CAT, DOG, DUCK

Instance
Segmentation



CAT, DOG, DUCK

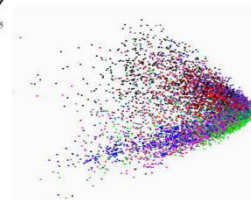
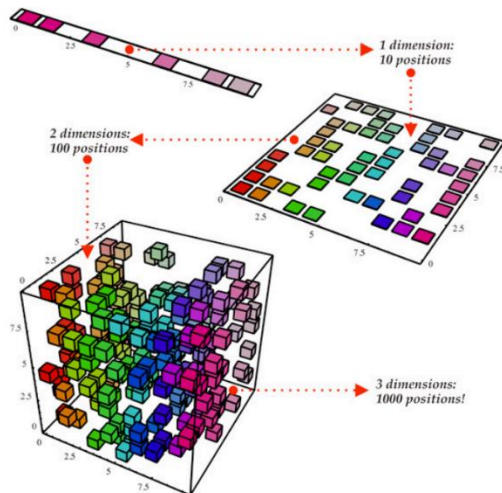
Single object

Multiple objects

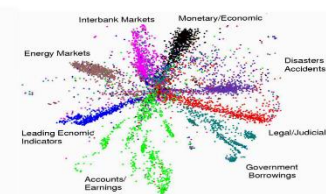
4.1 Four branches of machine learning

4.1.2 Unsupervised learning

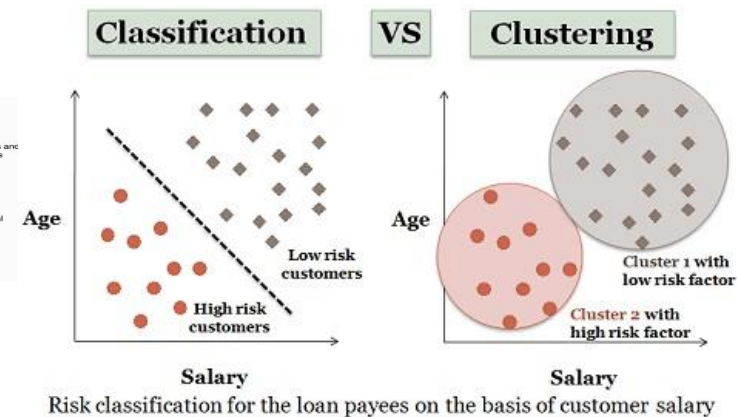
- ▶ input data without the help of any targets
- ▶ data visualization, data compression, or data denoising, or to better understand the correlations
- ▶ better understanding a dataset before attempting to solve a supervised-learning problem.
- ▶ *Dimensionality reduction* and *clustering* are well-known categories of unsupervised learning.



PCA



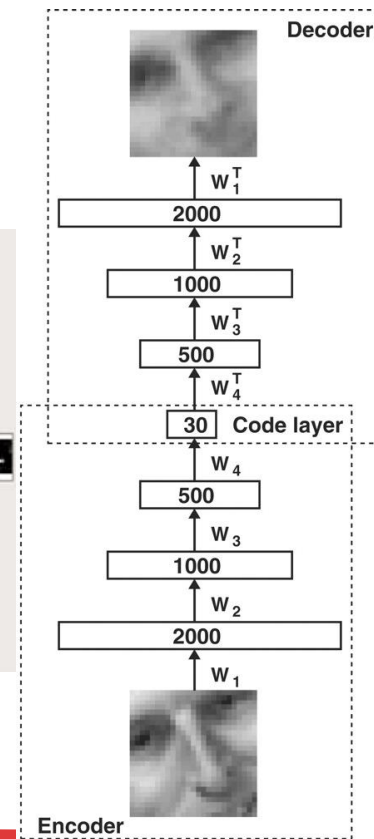
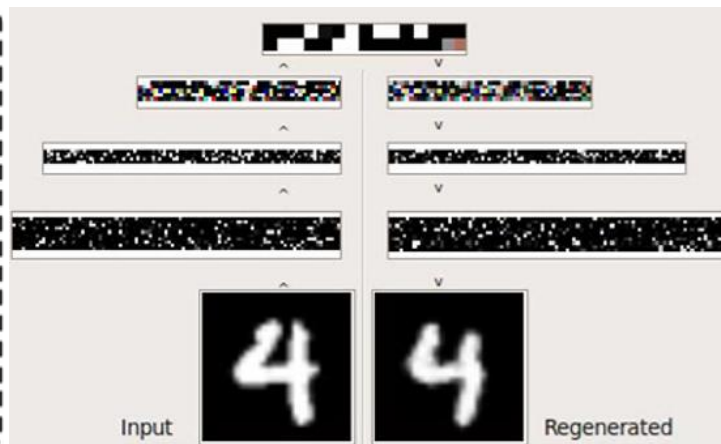
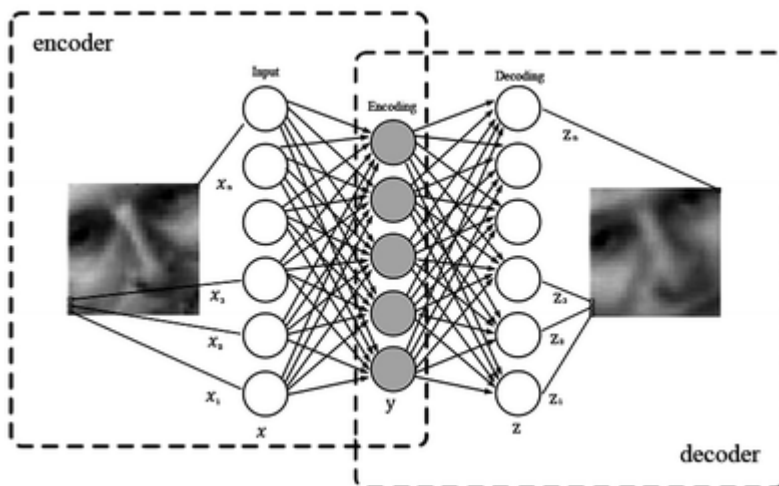
Auto Encoder



4.1 Four branches of machine learning

4.1.3 Self-supervised learning

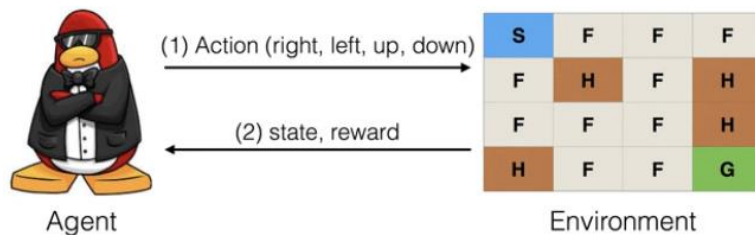
- ▶ **Self-supervised learning** is supervised learning without human-annotated labels but still **labels from the input data**.
- ▶ **autoencoders** - the generated **targets are the input**
- ▶ predict the next frame in a video, given past frames, or the next word in a text, given previous words



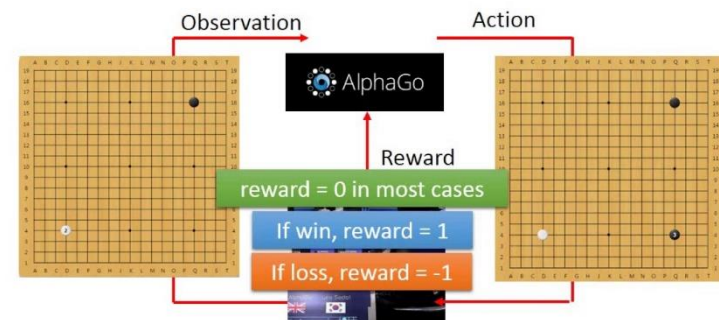
4.1 Four branches of machine learning

4.1.4 Reinforcement learning

- ▶ Google DeepMind successfully applied it to learning to play Atari games
- ▶ *agent* receives information about its *environment* and learns to choose actions that will maximize some reward.
- ▶ self-driving cars, robotics, resource management, education, and so on.



Learning to play Go



4.1 Four branches of machine learning

Classification and regression glossary

- **Sample or input**—One data point that goes into your model.
- **Prediction or output**—What comes out of your model.
- **Target**—The truth. What your model should ideally have predicted, according to an external source of data.
- **Prediction error or loss value**—A measure of the distance between your model's prediction and the target.
- **Classes**—A set of possible labels to choose from in a classification problem. (“dog” and “cat” classes)
- **Label**—A specific instance of a class annotation in a classification problem.
- **Ground-truth or annotations**—All targets for a dataset, typically collected by humans.
- **Binary classification**—each input sample should be categorized into **two exclusive categories**.
- **Multiclass classification**—each input sample should be categorized into **more than two categories**
- **Multilabel classification**—each input sample can be assigned multiple labels. For instance, a given image may contain both a cat and a dog and should be annotated both with the “cat” label and the “dog” label.
- **Scalar regression**—A task where the target is a continuous scalar value, Predicting house
- **Vector regression**—A task where the target is a set of continuous values (the coordinates of a bounding box in an image)
- **Mini-batch** or **batch**—A small set of samples (typically between 8 and 128) that are processed simultaneously by the model. The number of samples is often a power of 2, to facilitate memory allocation on GPU. When training, a mini-batch is used to compute a single gradient-descent update applied to the weights of the model.

4.2 Evaluating machine-learning models

- ▶ split the data into a **training** set, a **validation** set, and a **test** set
- ▶ after just a few epochs, all three models began to **overfit**
- ▶ the goal is **generalize**—perform well on never-before-seen
- ▶ The following sections look at strategies for **mitigating overfitting** and **maximizing generalization**.
- ▶ In this section, we'll focus on how to measure **generalization**: how to **evaluate** machine-learning models.

○ 4.2 ○ Evaluating machine-learning models ○

4.2.1 Training, validation, and test sets

- ▶ Evaluating a model: splitting the available data into **training, validation, and test**
- ▶ developing a model - tuning its configuration: for example, choosing the **number of layers** or the **size of the layers** (called the *hyper-parameters* of the model, to distinguish them from the *parameters, which are the network's weights*), form of *learning*: a search
- ▶ never-before-seen dataset to evaluate the model: the test dataset, *no* information about the test set

4.2 Evaluating machine-learning models

4.2.1 Training, validation, and test sets

▶ three classic evaluation recipes: simple hold-out validation, K-fold validation, and iterated K-fold validation with shuffling.

SIMPLE HOLD-OUT VALIDATION

▶ Set apart some fraction of your data as your test set. Train on the remaining data, and evaluate on the test set.

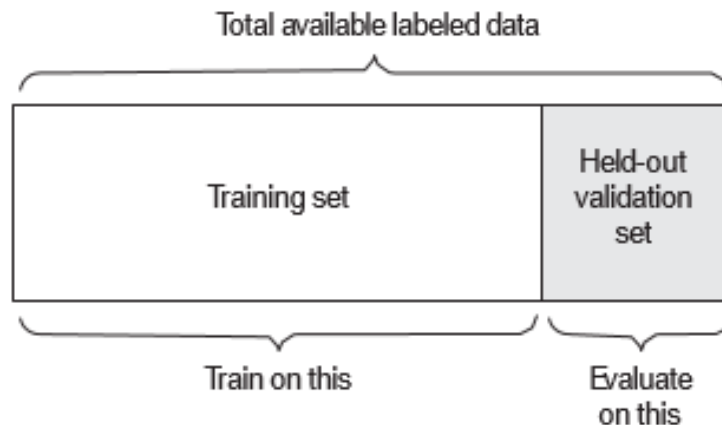


Figure 4.1 Simple hold-out validation split

4.2 Evaluating machine-learning models

4.2.1 Training, validation, and test sets

Listing 4.1 Hold-out validation

```
num_validation_samples = 10000
```

```
np.random.shuffle(data)
```

```
validation_data = data[:num_validation_samples] # 훈련
```

```
data = data[num_validation_samples:] # 검증
```

```
training_data = data[:]
```

```
model = get_model() # 미리 정의된 model 호출
```

```
model.train(training_data)
```

```
validation_score = model.evaluate(validation_data)
```

```
# At this point you can tune your model,
```

```
# retrain it, evaluate it, tune it again...
```

```
model = get_model() # 최종 테스트
```

```
model.train(np.concatenate([training_data, validation_data])) # 훈련 + 검증
```

```
test_score = model.evaluate(test_data) # test
```

```
In [36]: english_grade = [100,90,92,95,70,84,80,75,89]
E = np.array(english_grade)
print(E)

np.random.shuffle(E)
print(E)
```

```
[100  90  92  95  70  84  80  75  89]
[ 75 100  90  92  84  95  89  80  70]
```

4.2 Evaluating machine-learning models

4.2.1 Training, validation, and test sets

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 .
- ▶ Schematically, K-fold cross-validation looks like figure 4.2.

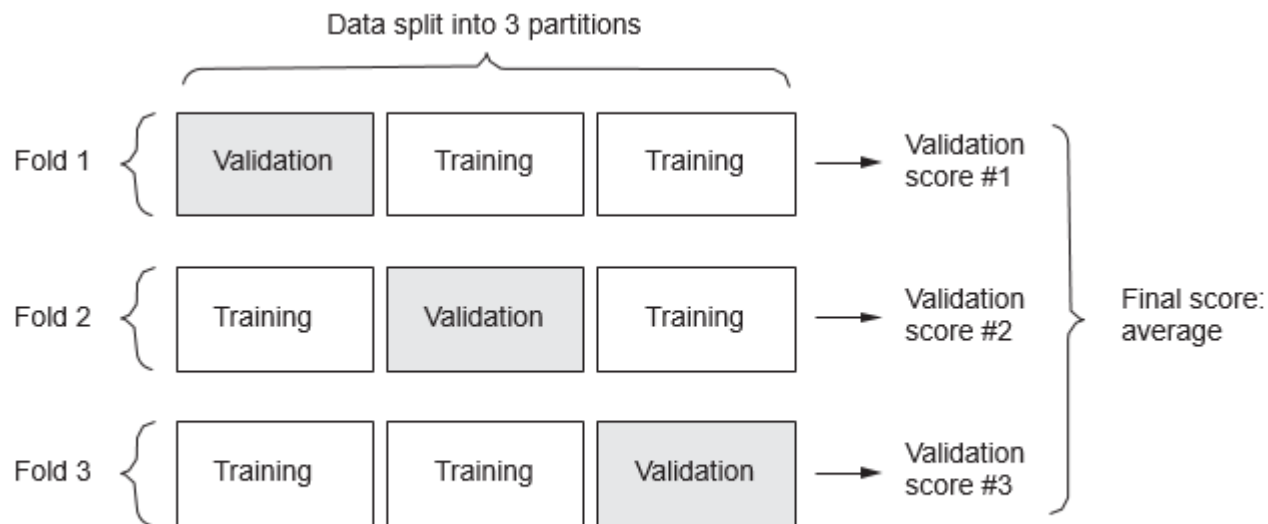


Figure 4.2 Three-fold validation

4.2 Evaluating machine-learning models

4.2.1 Training, validation, and test sets

▶ Listing 4.2 K-fold cross-validation

```
import numpy as np
k = 4
num_val_samples = len(train_data) // k # 나눗셈의 몫
np.random.shuffle(data)
validation_scores = []
for fold in range(k):
    validation_data = data[num_validation_samples * fold:num_validation_samples * (fold+1)]
    training_data = data[:num_validation_samples*fold] + data[num_validation_samples*(fold + 1):]
    model = get_model()
    model.train(training_data)
    validation_score = model.evaluate(validation_data)
    validation_scores.append(validation_score)

validation_score = np.average(validation_scores) # 책 오류
model = get_model()
model.train(data)
test_score = model.evaluate(test_data)
```

ITERATED K-FOLD VALIDATION WITH SHUFFLING

- ▶ $P \times K$ models (where P is the number of iterations you use)

4.2 Evaluating machine-learning models

4.2.2 Things to keep in mind

Keep an eye out for the following when you're choosing an evaluation protocol:

- *Data representativeness* — 80% of training set containing only classes 0–7, whereas 20% of test set contains only classes 8–9 (ridiculous mistake), *randomly shuffle* your data before splitting it into training and test sets.
- *The arrow of time* — predict the future given the past (for example, tomorrow's weather, stock movements, and so on), *not randomly shuffle* your data before splitting it, *test set* is *posterior* to the data in the *training set*.
- *Redundancy in your data* — Make sure your training set and validation set are *disjoint* (not redundancy between the training and validation sets).

4.3 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** (for example, specific to text data or image data).

4.3.1 Data preprocessing for neural networks

- ▶ Data preprocessing: vectorization, normalization, handling missing values, and feature extraction.

VECTORIZATION

- ▶ *data vectorization* - All inputs and targets in a neural network must be tensors of **floating-point data** (or, in specific cases, tensors of integers) for weights and a lot of functions. - sound, images, text
- ▶ one-hot encoding - tensor of `float32` data

4.3 Data preprocessing, feature engineering, and feature learning

VALUE NORMALIZATION

- ▶ digit-classification - 0–255 grayscale values → float32 values in the 0–1 range
- ▶ predicting house prices - variety of ranges → normalize each feature independently with a standard deviation of 1 and a mean of 0
 - *Take small values*—Typically, most values should be in the 0–1 range.
 - *Be homogenous*—all features should take values in roughly the same scale
- ▶ normalization practice:
 - Normalize each feature independently to have a mean of 0.
 - Normalize each feature independently to have a standard deviation of 1.

This is easy to do with Numpy arrays:

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

4.3 *Data preprocessing, feature engineering, and feature learning*

HANDLING MISSING VALUES

- ▶ missing values - input missing values as 0, with the condition that 0 isn't already a meaningful value, ignoring the value.
- ▶ missing values in the test data, but no missing values in the train data - copy some missing data samples to the train data

4.3 Data preprocessing, feature engineering, and feature learning

4.3.2 Feature engineering

- ▶ *Feature engineering* - the process of **transformations** of input data for better result before it goes into the model
- ▶ **model's job easier**
- ▶ image of a clock and can output the time of day (see figure 4.3).
- ▶ raw pixels $<$ (x, y) coordinates $<$ angle theta of each clock hand.
- ▶ making a problem easier by expressing it in a simpler way



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

Figure 4.3 Feature engineering for reading the time on a clock

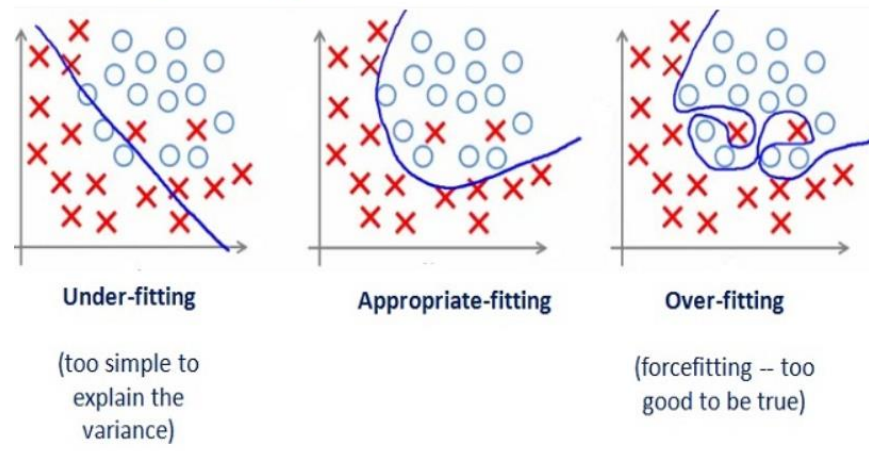
4.3 Data preprocessing, feature engineering, and feature learning

4.3.2 Feature engineering

- ▶ Before deep learning, feature engineering used to be critical
- ▶ MNIST digit-classification problem - number of loops in a digit image, the height of each digit in an image, a histogram of pixel values, and so on.
- ▶ Deep learning removes the need for most feature engineering by **automatic extraction** of useful features from raw data.
- ▶ Does this mean you don't have to worry about feature engineering as long as you're using deep neural networks? **No**, for two reasons:
 - Good features still allow you to solve problems more elegantly while using **fewer resources** (clock example).
 - Good features let you solve a problem with far **less data**.

4.4 Overfitting and underfitting

- ▶ In all three examples—predicting movie reviews, topic classification, and house-price regression—the model quickly started to **overfit** to the training data.
- ▶ **Optimization** refers to the process of adjusting a model to get the best performance possible on the **training data**.
- ▶ **Generalization** refers to how well the trained model performs on **test data**.
- ▶ **underfit**: the network hasn't yet modeled all relevant patterns in the training data.
- ▶ A model trained on **more data** will naturally **generalize better**.
- ▶ The processing of fighting overfitting this way is called **regularization**.



○ 4.4 ○ Overfitting and underfitting ○ ○ ○

4.4.1 Feature engineering Reducing the network's size

- ▶ **reduce** the size of the model: number of **layers** and the number of **units** per layer
- ▶ model's large *memorization capacity*: perfect dictionary-like mapping between training samples and their targets—a mapping without any generalization power.
- ▶ but the real challenge is **generalization**, not fitting.
- ▶ **no magical formula** to determine the right number of layers or the right size for each layer.
- ▶ start with relatively few layers and parameters, and increase the size of the layers

4.4 Overfitting and underfitting

4.4.1 Feature engineering Reducing the network's size

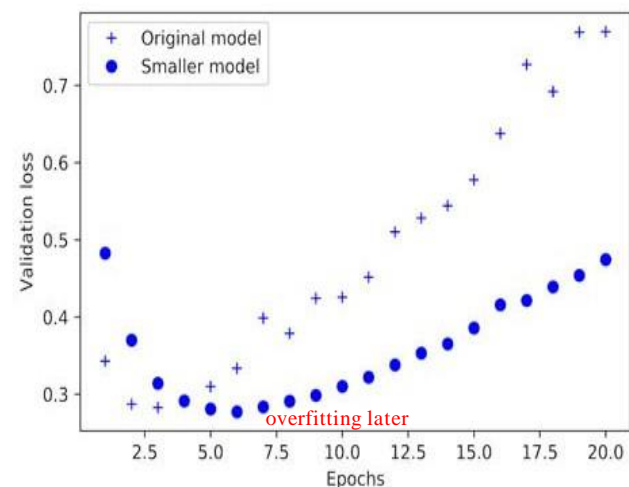
Listing 4.3 Original model

```
from keras import models
from keras import layers
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Listing 4.4 Version of the model with lower capacity

```
model = models.Sequential()
model.add(layers.Dense(4, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Figure 4.4 Effect of model capacity on validation loss: trying a smaller model



4.4 Overfitting and underfitting

4.4.1 Feature engineering Reducing the network's size

Listing 4.5 Version of the model with higher capacity

```
model = models.Sequential()  
model.add(layers.Dense(512, activation='relu', input_shape=(10000,)))  
model.add(layers.Dense(512, activation='relu'))  
model.add(layers.Dense(1, activation='sigmoid'))
```

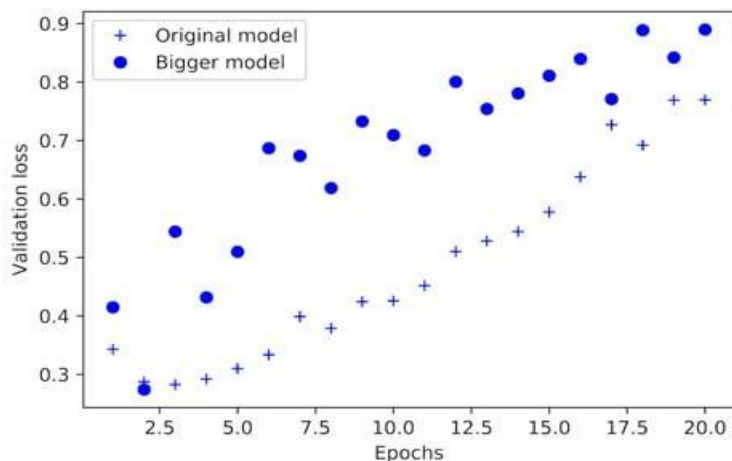


Figure 4.5 Effect of model capacity on **validation loss**: trying a bigger model

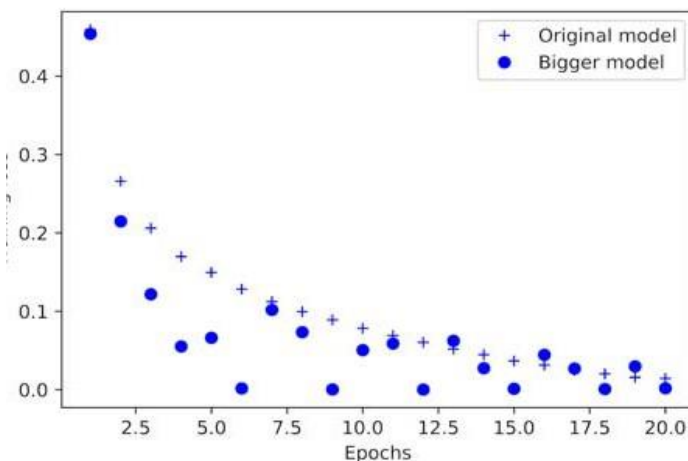


Figure 4.6 Effect of model capacity **on training loss**: trying a bigger model

4.4 Overfitting and underfitting

4.4.2 Adding weight regularization

- ▶ *Occam's razor* : given two explanations for something, the explanation most likely to be correct is the **simplest one**, 노트르담 대성당 화재 원인, 영화선택
- ▶ neural networks: **Simpler models** are less likely to overfit than complex ones.
- ▶ A *simple model* - fewer parameters
- ▶ **weight regularization** - adding to the loss function of the network a **cost** associated with having large weights:
 - **L1 regularization** —The cost added is proportional to the **absolute value** of the weight coefficients (the *L1 norm* of the weights) .
 - **L2 regularization**(**weight decay** —The cost added is proportional to the **square of the value** of the weight coefficients (the *L2 norm* of the weights).

4.4 Overfitting and underfitting

► L1 regularization (λ) with Back-Propagation

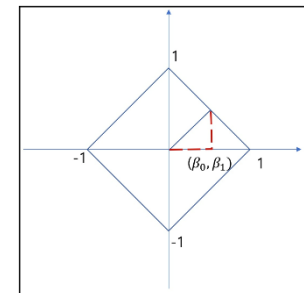
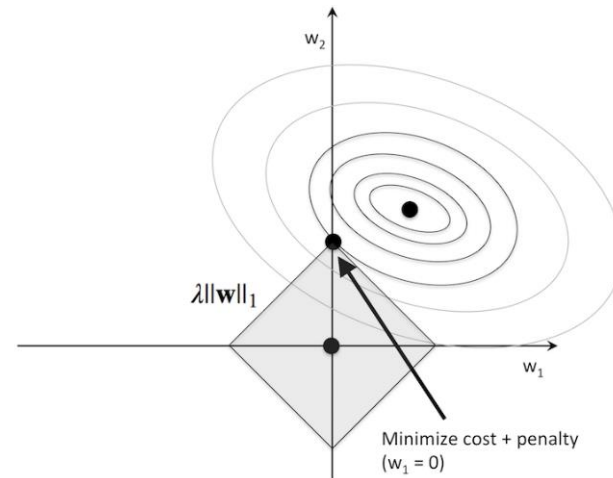
current values of the weights are (6.0, -2.0, 4.0), $\lambda=0.1$

add $0.1 * [|6.0| + | -2.0| + |4.0|] = 0.1 * (6.0 + 2.0 + 4.0) = 0.1 * 12.0 = 1.20$ to E

$$E = \underbrace{\frac{1}{2} * \sum (t_k - o_k)^2}_{\text{squared error}} + \underbrace{\lambda * \sum |w_i|}_{\text{L1 weight penalty}}$$

$$\Delta w_{jk} = -1 * \underbrace{\eta}_{\text{learning rate}} * \underbrace{\left[x_j * (o_k - t_k) * o_k * (1 - o_k) \right] \pm \lambda}_{\text{signal}} \quad \frac{\partial E}{\partial w_{jk}} \text{ gradient}$$

$$w_{jk} = w_{jk} + \Delta w_{jk}$$



$$\text{L1: } ||\beta||_1 = |\beta_0| + |\beta_1| = 1$$

4.4 Overfitting and underfitting

► L2 regularization (λ) with Back-Propagation

current values of the weights are (5.0, -3.0, 2.0), $\lambda=0.1$

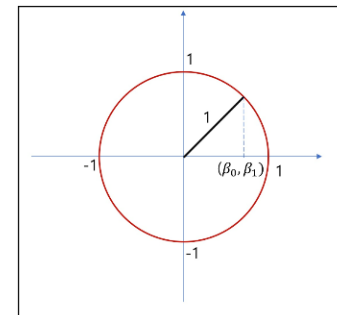
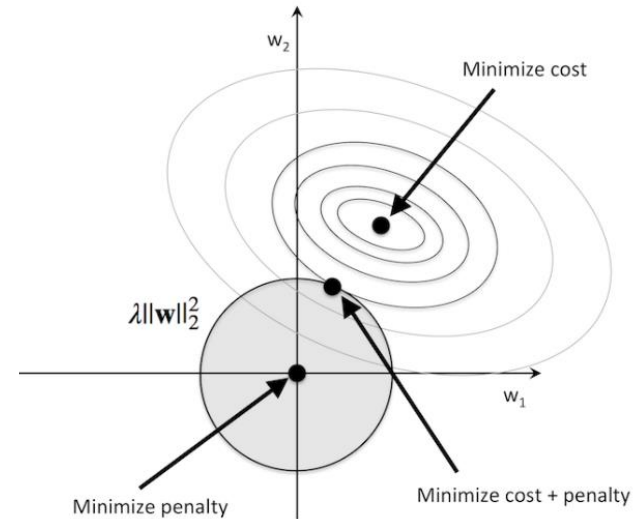
add $0.1 * [(5.0)^2 + (-3.0)^2 + (2.0)^2] = 0.1 * (25.0 + 9.0 + 4.0) = 0.1 * 38.0 = 3.80$ to E

$$E = \underbrace{\frac{1}{2} * \sum (t_k - o_k)^2}_{\text{plain error}} + \underbrace{\frac{\lambda}{2} * \sum w_i^2}_{\text{weight penalty}}$$

elegant math \downarrow simple math

$$\frac{\partial E}{\partial w_{jk}} \quad \text{gradient}$$

$$\Delta w_{jk} = \underbrace{\eta}_{\text{learning rate}} * \underbrace{[x_j * (o_k - t_k) * o_k * (1 - o_k)] + [\lambda * w_{jk}]}_{\text{signal}}$$



$$\text{L2: } \|\beta\|_2 = \sqrt{(\beta_0)^2 + (\beta_1)^2} = 1$$

4.4 Overfitting and underfitting

- ▶ L_1 은 경우에 따라 특정 Feature(Vector의 요소)없이도 같은 값을 낼 수 있다
- ▶ 이런 특징으로 L_1 은 **Feature Selection**을 하는 데 사용할 수 있고, 특정 Feature들을 0으로 처리해버리는 것이 가능하기 때문에 결과적으로 **Sparse**한 형태를 가질 수 있다.

$$a = (0.25, 0.25, 0.25, 0.25)$$

$$b = (-0.5, 0.5, 0.0, 0.0)$$

이 두 벡터의 L_1 norm을 구하면,

$$\|a\|_1 = |0.25| + |0.25| + |0.25| + |0.25| = 1$$

$$\|b\|_1 = |-0.5| + |0.5| + |0.0| + |0.0| = 1$$

과 같이 같은 1이라는 숫자가 나오지만,

L_2 norm을 구하면,

$$\|a\|_2 = \sqrt{0.25^2 + 0.25^2 + 0.25^2 + 0.25^2} = 0.5$$

$$\|b\|_2 = \sqrt{(-0.5)^2 + (0.5)^2 + 0^2 + 0^2} = 0.707$$

과 같이 다른 수가 나온다.

L1

SPARSITY: (1, 0, 0, 1, 0)

GOOD FOR FEATURE
SELECTION

L2

SPARSITY: (0.5, 0.3, -0.2, 0.4, 0.1)

NORMALLY BETTER FOR
TRAINING MODELS

$$(1, 0) \rightarrow (0.5, 0.5)$$

$$1^2 + 0^2 = 1 \quad 0.5^2 + 0.5^2 = 0.5$$