

Deep Learning with Python

Chapter 3: Getting started
with neural networks

Loss functions and optimizers: keys to configuring the learning process

Choosing the right objective function for the right problem is extremely important

Imagine a stupid, omnipotent AI trained via SGD, with this poorly chosen objective function: “maximizing the average well-being of all humans alive.” To make its job easier, this AI might choose to kill all humans except a few and focus on the well-being of the remaining ones—because average well-being isn’t affected by how many humans are left. That might not be what you intended!

You'll use **binary crossentropy** for a two-class classification problem, **categorical crossentropy** for a many-class classification problem, **meansquared error** for a regression problem, **connectionist temporal classification (CTC)** for a sequence-learning problem, and so on. Only when you're working on truly new research problems will you have to develop your own objective functions.

Introduction to Keras

Keras has the following key features:

- It allows the same code to run seamlessly on CPU or GPU.
- It has a user-friendly API that makes it easy to quickly prototype deep-learning models.
- It has built-in support for convolutional networks (for computer vision), recurrent networks (for sequence processing), and any combination of both.
- It supports arbitrary network architectures: multi-input or multi-output models, layer sharing, model sharing, and so on. This means Keras is appropriate for building essentially any deep-learning model, from a generative adversarial network to a neural Turing machine.

Introduction to Keras

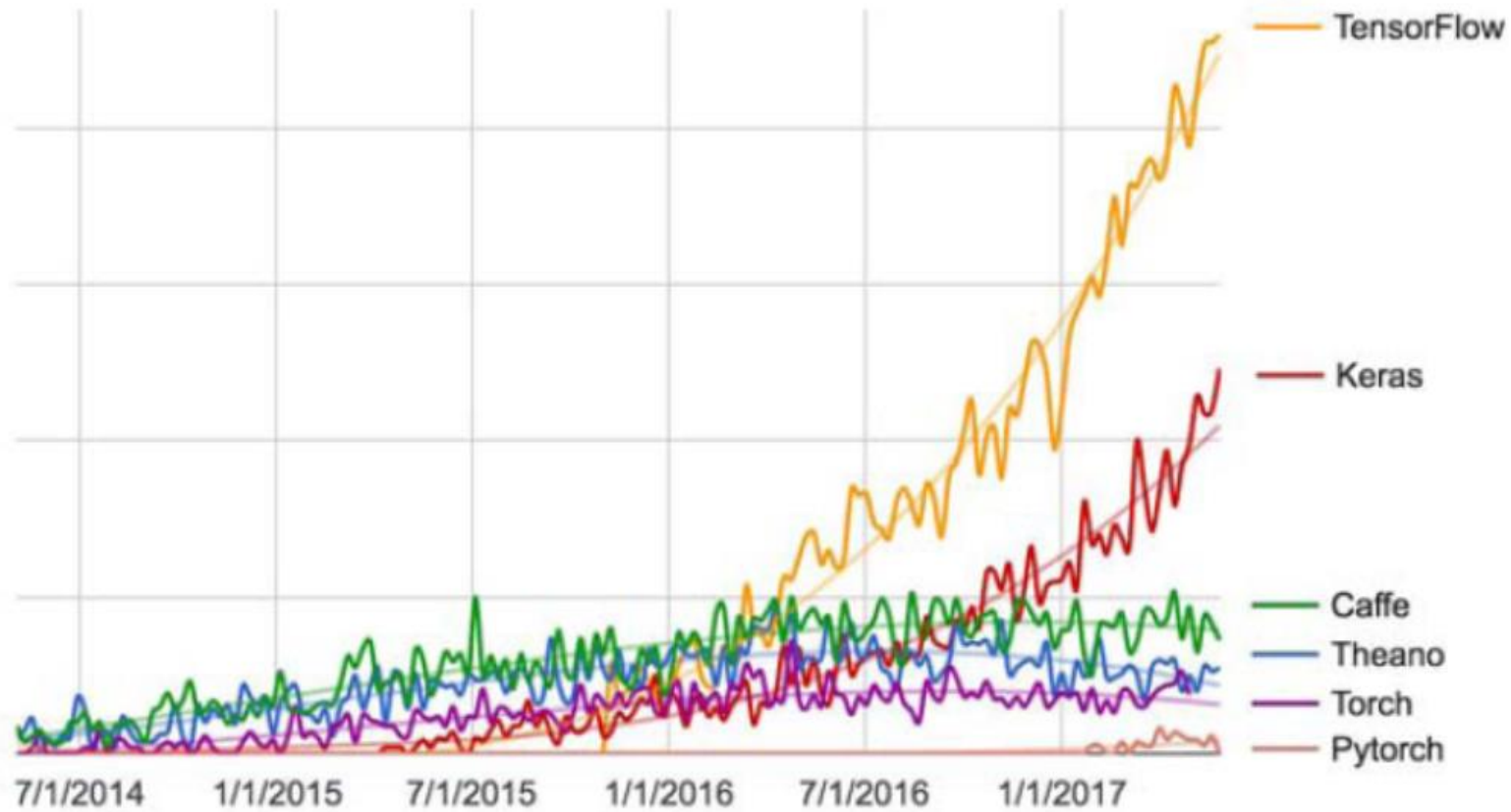


Figure 3.2 Google web search interest for different deep-learning frameworks over time

Introduction to Keras

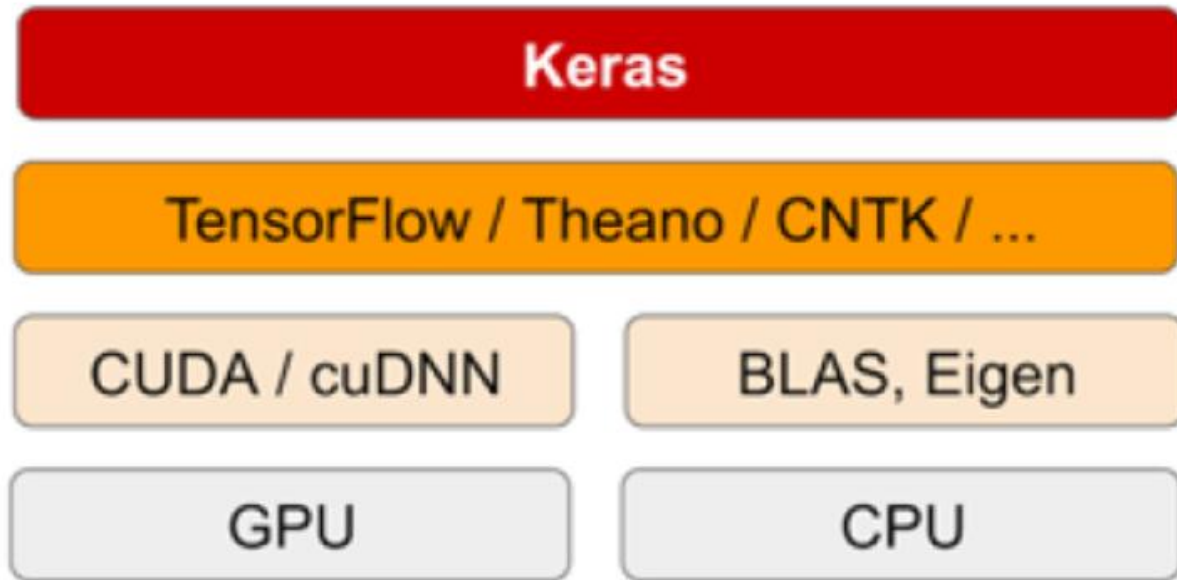


Figure 3.3 The deep-learning software and hardware stack

Introduction to Keras

There are two ways to define a model: using the `Sequential` class (only for linear stacks of layers, which is the most common network architecture by far) or the *functional API* (for directed acyclic graphs of layers, which lets you build completely arbitrary architectures).

Introduction to Keras

Sequential

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(784,)))
model.add(layers.Dense(10, activation='softmax'))
```

Functional

```
input_tensor = layers.Input(shape=(784,))
x = layers.Dense(32, activation='relu')(input_tensor)
output_tensor = layers.Dense(10, activation='softmax')(x)

model = models.Model(inputs=input_tensor, outputs=output_tensor)
```


Introduction to Keras

The learning process is configured in the compilation step, where you specify the optimizer and loss function(s) that the model should use, as well as the metrics you want to monitor during training. Here's an example with a single loss function, which is by far the most common case:

```
from keras import optimizers

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

model.fit(input_tensor, target_tensor, batch_size=128, epochs=10)
```

Classifying movie reviews: a binary classification example

The IMDB dataset

You'll work with the IMDB dataset: a set of 50,000 highly polarized reviews from the Internet Movie Database. They're split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting of 50% negative and 50% positive reviews.

Classifying movie reviews: a binary classification example

Listing 3.1 Loading the IMDB dataset

```
from keras.datasets import imdb

(train_data, train_labels), (test_data, test_labels) = imdb.load_data(
    num_words=10000)
```

The argument `num_words=10000` means you'll only keep the top 10,000 most frequently occurring words in the training data. Rare words will be discarded. This allows you to work with vector data of manageable size.

The variables `train_data` and `test_data` are lists of reviews; each review is a list of word indices (encoding a sequence of words). `train_labels` and `test_labels` are lists of 0s and 1s, where 0 stands for *negative* and 1 stands for *positive*:

```
>>> train_data[0]
[1, 14, 22, 16, ... 178, 32]
```

```
>>> train_labels[0]
1
```

Preparing the data

- One-hot encode your lists to turn them into vectors of 0s and 1s. This would mean, for instance, turning the sequence [3, 5] into a 10,000-dimensional vector that would be all 0s except for indices 3 and 5, which would be 1s. Then you could use as the first layer in your network a Dense layer, capable of handling floating-point vector data.

Listing 3.2 Encoding the Integer sequences into a binary matrix

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results

x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

Creates an all-zero matrix of shape (len(sequences), dimension)

Sets specific indices of results[i] to 1s

Vectorized training data

Vectorized test data

Here's what the samples look like now:

```
>>> x_train[0]
array([ 0.,  1.,  1., ...,  0.,  0.,  0.])
```

You should also vectorize your labels, which is straightforward:

```
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

Now the data is ready to be fed into a neural network.

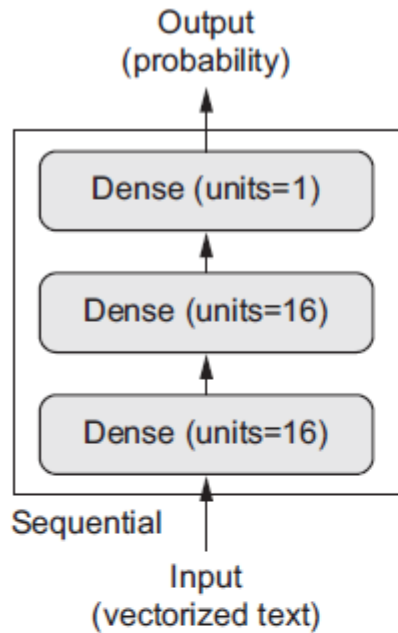


Figure 3.6 The three-layer network

Figure 3.6 shows what the network looks like. And here's the Keras implementation, similar to the MNIST example you saw previously.

Listing 3.3 The model definition

```
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 3.4 Compiling the model

```
model.compile(optimizer='rmsprop',  
              loss='binary_crossentropy',  
              metrics=['accuracy'])
```

Listing 3.5 Configuring the optimizer

```
from keras import optimizers  
  
model.compile(optimizer=optimizers.RMSprop(lr=0.001),  
              loss='binary_crossentropy',  
              metrics=['accuracy'])
```

Listing 3.6 Using custom losses and metrics

```
from keras import losses  
from keras import metrics  
  
model.compile(optimizer=optimizers.RMSprop(lr=0.001),  
              loss=losses.binary_crossentropy,  
              metrics=[metrics.binary_accuracy])
```

Listing 3.7 Setting aside a validation set

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]

y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

Listing 3.8 Training your model

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

history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```


Listing 3.9 Plotting the training and validation loss

```
import matplotlib.pyplot as plt

history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']

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

plt.plot(epochs, loss_values, 'bo', label='Training loss')
plt.plot(epochs, val_loss_values, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

“bo” is for
“blue dot.”

“b” is for “solid
blue line.”

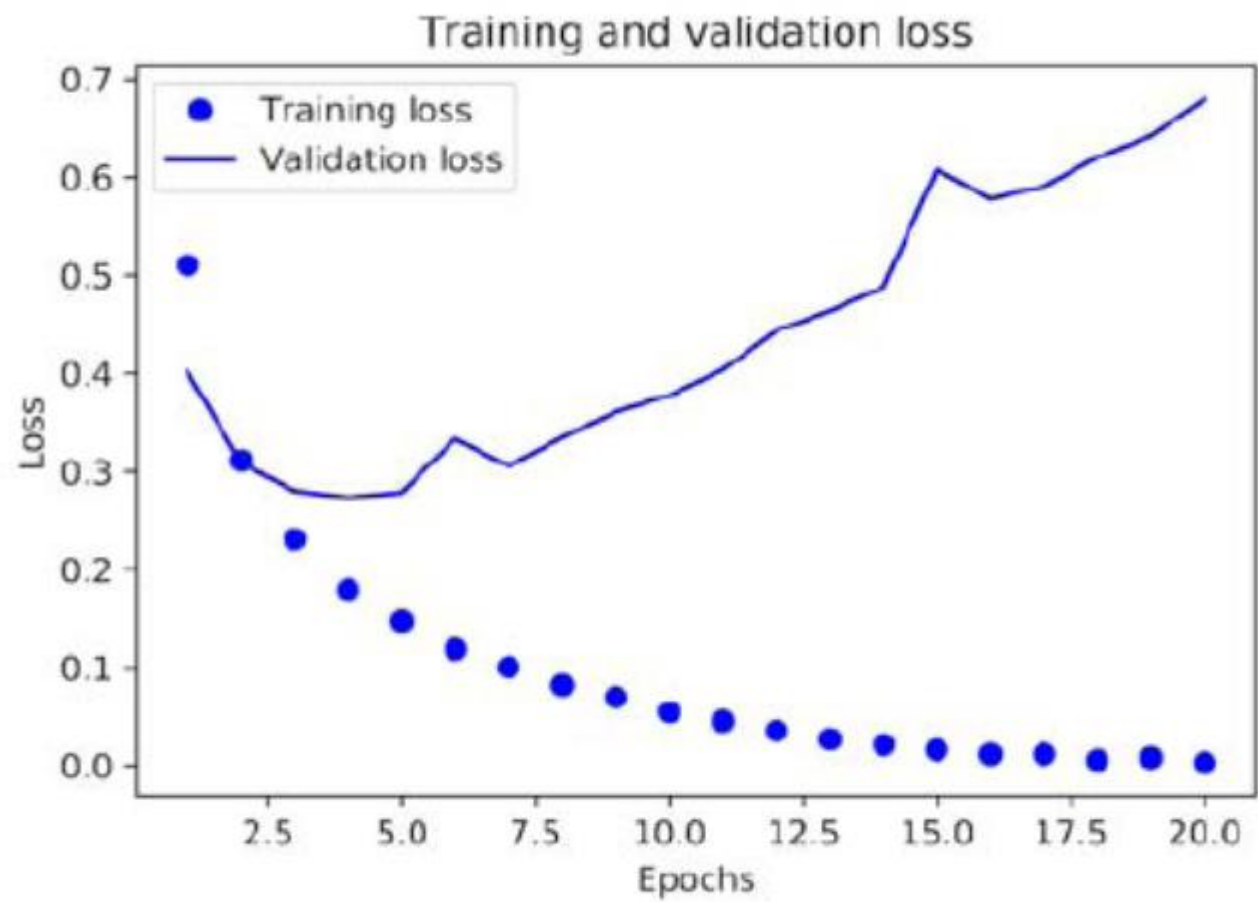


Figure 3.7 Training and validation loss

Listing 3.10 Plotting the training and validation accuracy

```
plt.clf()           ← Clears the figure
acc_values = history_dict['acc']
val_acc_values = history_dict['val_acc']

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

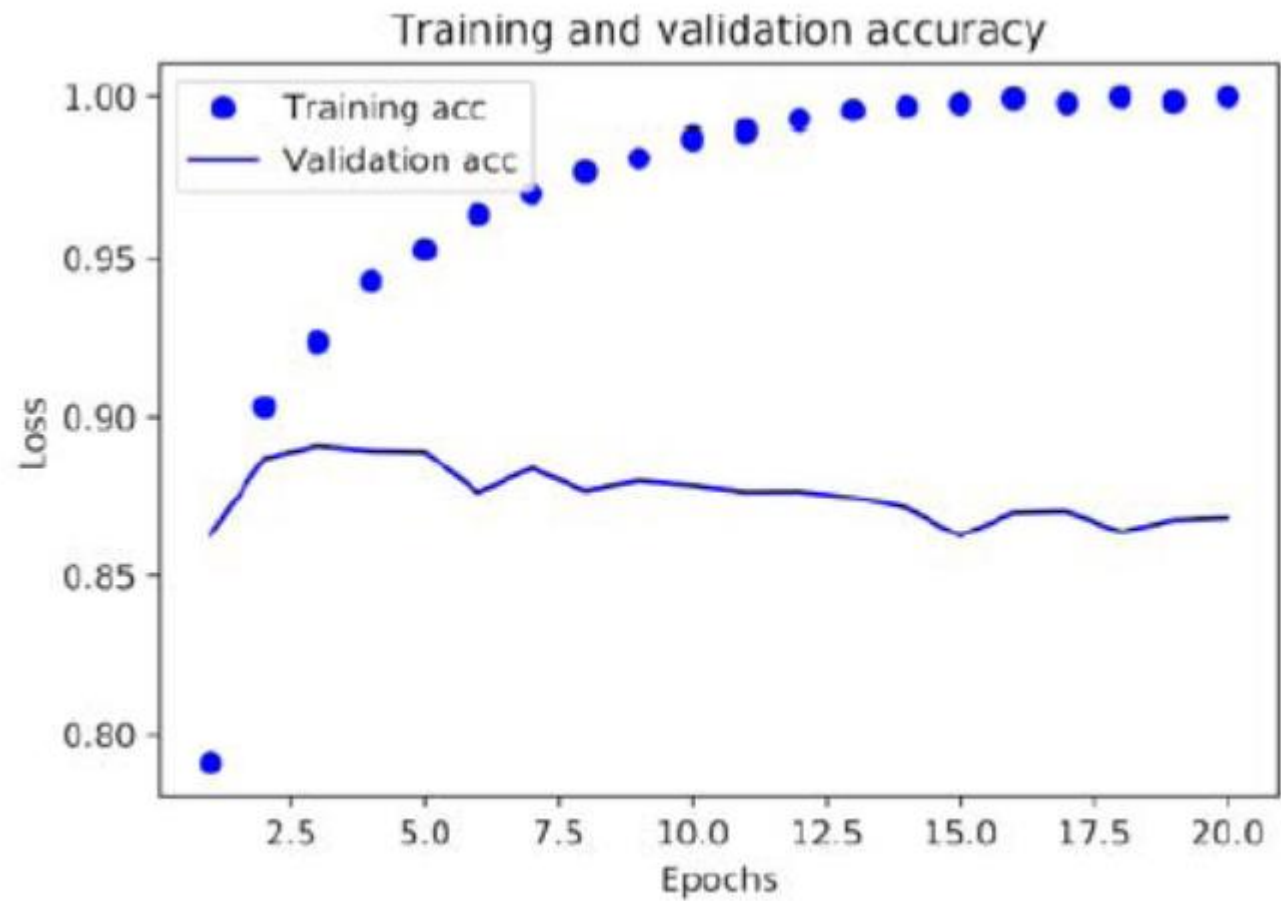


Figure 3.8 Training and validation accuracy

Listing 3.11 Retraining a model from scratch

```
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'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=4, batch_size=512)
results = model.evaluate(x_test, y_test)
```

The final results are as follows:

```
>>> results
[0.2929924130630493, 0.8832799999999999]
```

3.4.5 *Using a trained network to generate predictions on new data*

After having trained a network, you'll want to use it in a practical setting. You can generate the likelihood of reviews being positive by using the `predict` method:

```
>>> model.predict(x_test)
array([[ 0.98006207]
       [ 0.99758697]
       [ 0.99975556]
       ...,
       [ 0.82167041]
       [ 0.02885115]
       [ 0.65371346]], dtype=float32)
```

3.5 *Classifying newswires: a multiclass classification example*

3.5.1 The Reuters dataset

You'll work with the *Reuters dataset*, a set of short newswires and their topics, published by Reuters in 1986. It's a simple, widely used toy dataset for text classification. There are 46 different topics; some topics are more represented than others, but each topic has at least 10 examples in the training set.

Like IMDB and MNIST, the Reuters dataset comes packaged as part of Keras. Let's take a look.

Listing 3.12 Loading the Reuters dataset

```
from keras.datasets import reuters

(train_data, train_labels), (test_data, test_labels) = reuters.load_data(
    num_words=10000)
```



```
>>> len(train_data)
8982
>>> len(test_data)
2246
```

As with the IMDB reviews, each example is a list of integers (word indices):

```
>>> train_data[10]
[1, 245, 273, 207, 156, 53, 74, 160, 26, 14, 46, 296, 26, 39, 74, 2979,
3554, 14, 46, 4689, 4329, 86, 61, 3499, 4795, 14, 61, 451, 4329, 17, 12]
```

Listing 3.14 Encoding the data

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results

x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

← Vectorized training data

← Vectorized test data

```
def to_one_hot(labels, dimension=46):
    results = np.zeros((len(labels), dimension))
    for i, label in enumerate(labels):
        results[i, label] = 1.
    return results

one_hot_train_labels = to_one_hot(train_labels)
one_hot_test_labels = to_one_hot(test_labels)
```

← **Vectorized training labels**

← **Vectorized test labels**

Note that there is a built-in way to do this in Keras, which you've already seen in action in the MNIST example:

```
from keras.utils.np_utils import to_categorical

one_hot_train_labels = to_categorical(train_labels)
one_hot_test_labels = to_categorical(test_labels)
```

Listing 3.15 Model definition

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))
```

Listing 3.16 Compiling the model

```
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Listing 3.17 Setting aside a validation set

```
x_val = x_train[:1000]
partial_x_train = x_train[1000:]

y_val = one_hot_train_labels[:1000]
partial_y_train = one_hot_train_labels[1000:]
```

Listing 3.18 Training the model

```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

Listing 3.19 Plotting the training and validation loss

```
import matplotlib.pyplot as plt

loss = history.history['loss']
val_loss = history.history['val_loss']

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

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

Listing 3.20 Plotting the training and validation accuracy

```
plt.clf() ← Clears the figure

acc = history.history['acc']
val_acc = history.history['val_acc']

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

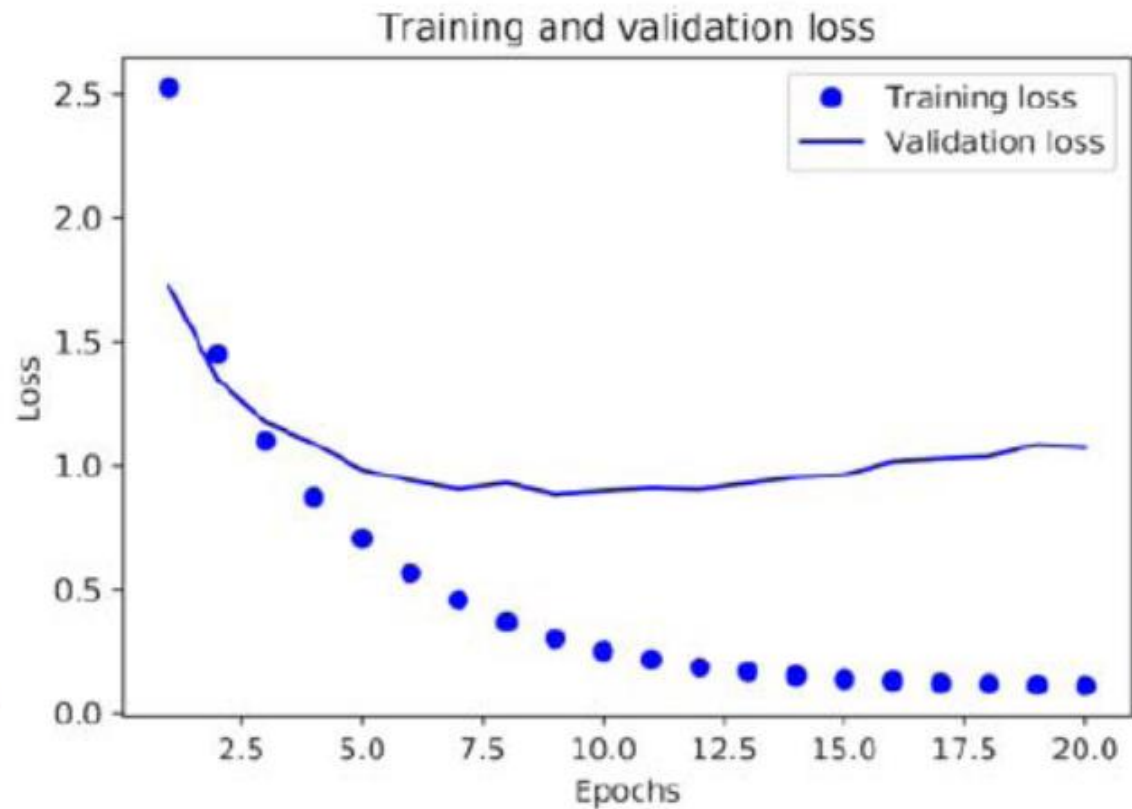


Figure 3.9 Training and validation loss

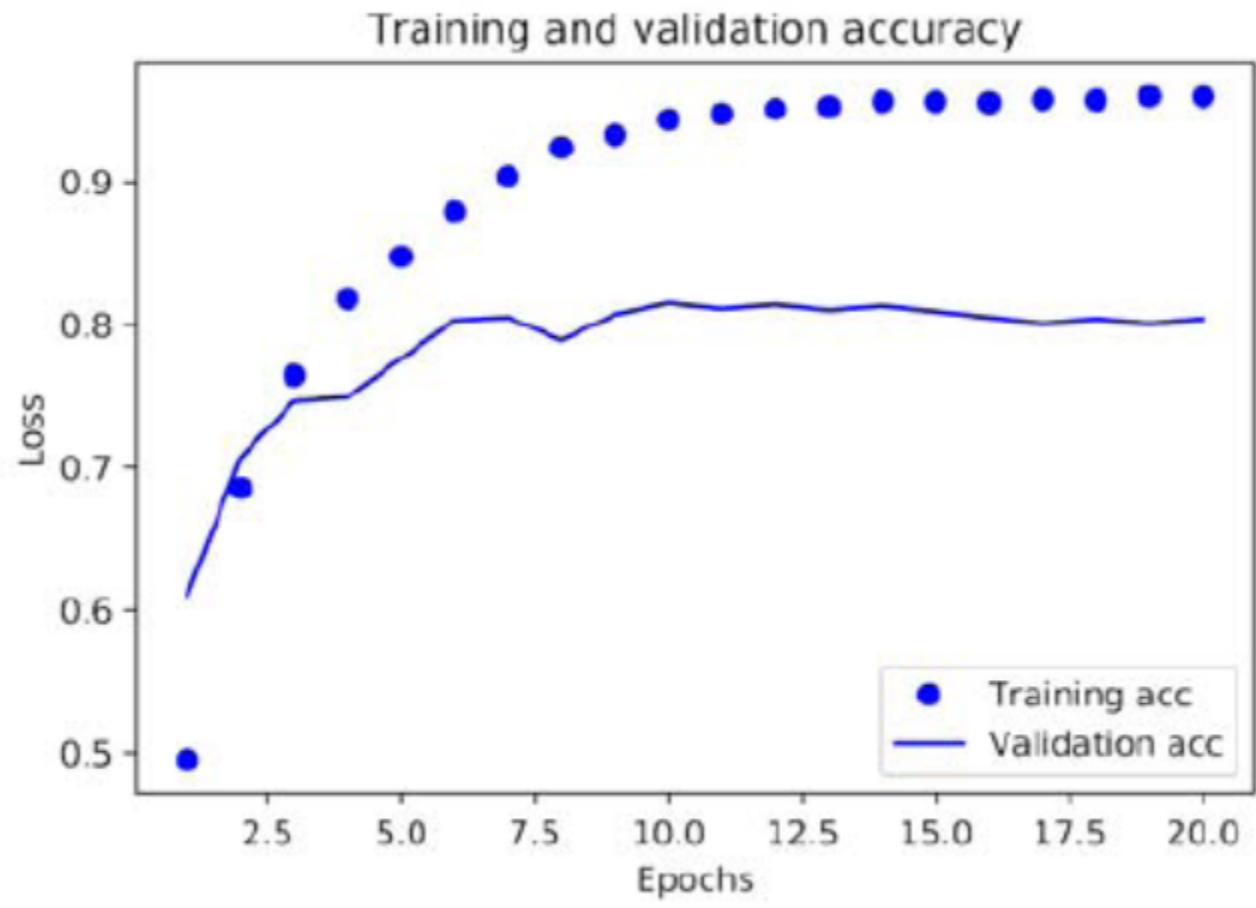


Figure 3.10 Training and validation accuracy

Listing 3.21 Retraining a model from scratch

```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(partial_x_train,
          partial_y_train,
          epochs=9,
          batch_size=512,
          validation_data=(x_val, y_val))
results = model.evaluate(x_test, one_hot_test_labels)
```

Here are the final results:

```
>>> results
[0.9565213431445807, 0.79697239536954589]
```

Listing 3.22 Generating predictions for new data

```
predictions = model.predict(x_test)
```

Each entry in `predictions` is a vector of length 46:

```
>>> predictions[0].shape  
(46,)
```

The coefficients in this vector sum to 1:

```
>>> np.sum(predictions[0])  
1.0
```

The largest entry is the predicted class—the class with the highest probability:

```
>>> np.argmax(predictions[0])  
4
```

3.5.6 *A different way to handle the labels and the loss*

We mentioned earlier that another way to encode the labels would be to cast them as an integer tensor, like this:

```
y_train = np.array(train_labels)
y_test = np.array(test_labels)
```

The only thing this approach would change is the choice of the loss function. The loss function used in listing 3.21, `categorical_crossentropy`, expects the labels to follow a categorical encoding. With integer labels, you should use `sparse_categorical_crossentropy`:

```
model.compile(optimizer='rmsprop',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])
```

Listing 3.23 A model with an information bottleneck

```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(partial_x_train,
        partial_y_train,
        epochs=20,
        batch_size=128,
        validation_data=(x_val, y_val))
```

The network now peaks at ~71% validation accuracy, an 8% absolute drop. This drop is mostly due to the fact that you're trying to compress a lot of information (enough information to recover the separation hyperplanes of 46 classes) into an intermediate space that is too low-dimensional. The network is able to cram *most* of the necessary information into these eight-dimensional representations, but not all of it.

3.6 Predicting house prices: a regression example

Listing 3.24 Loading the Boston housing dataset

```
from keras.datasets import boston_housing

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

Let's look at the data:

```
>>> train_data.shape
(404, 13)
>>> test_data.shape
(102, 13)
```

As you can see, you have 404 training samples and 102 test samples, each with 13 numerical features, such as per capita crime rate, average number of rooms per dwelling, accessibility to highways, and so on.

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

```
>>> train_targets
[ 15.2,  42.3,  50. ...  19.4,  19.4,  29.1]
```

Listing 3.25 Normalizing the data

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

test_data -= mean
test_data /= std
```

Listing 3.26 Model definition

```
from keras import models
from keras import layers

def build_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))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    return model
```

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

Validating your approach using K-fold validation

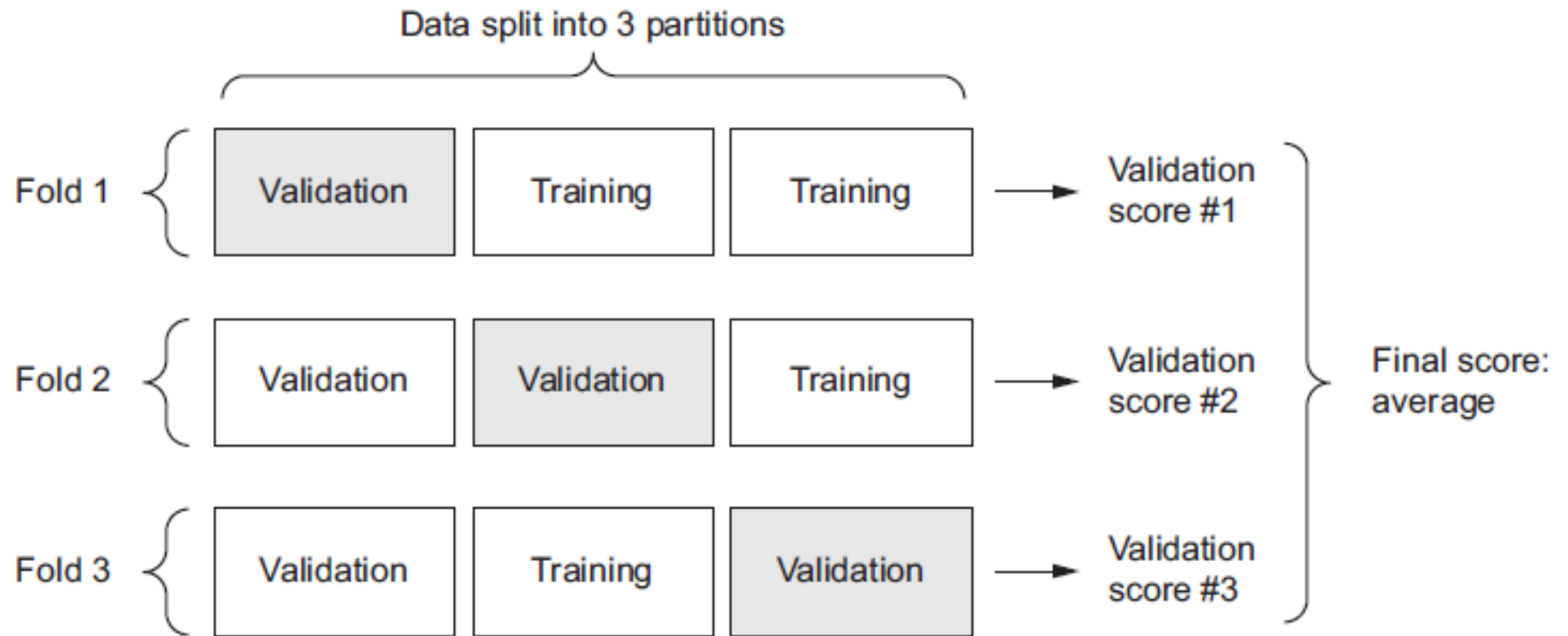


Figure 3.11 3-fold cross-validation

Listing 3.27 K-fold validation

```
import numpy as np

k = 4
num_val_samples = len(train_data) // k
num_epochs = 100
all_scores = []
```

**Prepares the validation data:
data from partition #k**

```
for i in range(k):
    print('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]
```

**Prepares 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)
```

**Builds the Keras model
(already compiled)**

```
model = build_model()
model.fit(partial_train_data, partial_train_targets,
          epochs=num_epochs, batch_size=1, verbose=0)
```

**Trains the model
(in silent mode,
verbose = 0)**

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

**Evaluates the model
on the validation data**

Running this with `num_epochs = 100` yields the following results:

```
>>> all_scores
[2.588258957792037, 3.1289568449719116, 3.1856116051248984, 3.0763342615401386]
>>> np.mean(all_scores)
2.9947904173572462
```

The different runs do indeed show rather different validation scores, from 2.6 to 3.2. The average (3.0) is a much more reliable metric than any single score—that's the entire point of K-fold cross-validation. In this case, you're off by \$3,000 on average, which is significant considering that the prices range from \$10,000 to \$50,000.

Listing 3.28 Saving the validation logs at each fold

```
num_epochs = 500
all_mae_histories = []
for i in range(k):
    print('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=1, verbose=0)
    mae_history = history.history['val_mean_absolute_error']
    all_mae_histories.append(mae_history)
```

**Prepares the validation data:
data from partition #k**

**Prepares the training
data: data from all
other partitions**

**Builds the Keras model
(already compiled)**

**Trains the model
(in silent mode, verbose=0)**

Listing 3.29 Building the history of successive mean K-fold validation scores

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

Listing 3.30 Plotting validation scores

```
import matplotlib.pyplot as plt

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

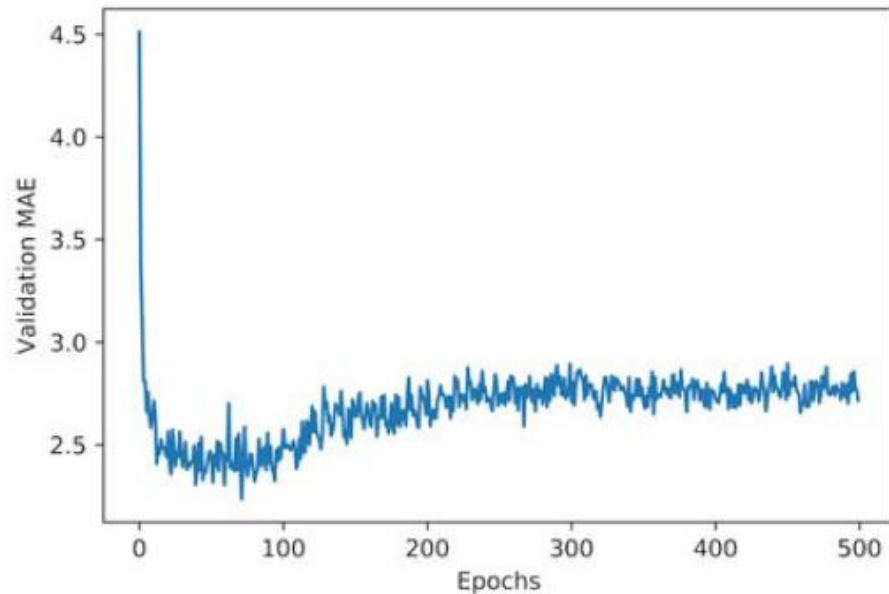


Figure 3.12 Validation MAE by epoch

It may be a little difficult to see the plot, due to scaling issues and relatively high variance. Let's do the following:

- Omit the first 10 data points, which are on a different scale than the rest of the curve.
- Replace each point with an exponential moving average of the previous points, to obtain a smooth curve.

Listing 3.31 Plotting validation scores, excluding the first 10 data points

```
def smooth_curve(points, factor=0.9):
    smoothed_points = []
    for point in points:
        if smoothed_points:
            previous = smoothed_points[-1]
            smoothed_points.append(previous * factor + point * (1 - factor))
        else:
            smoothed_points.append(point)
    return smoothed_points

smooth_mae_history = smooth_curve(average_mae_history[10:])

plt.plot(range(1, len(smooth_mae_history) + 1), smooth_mae_history)
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
plt.show()
```

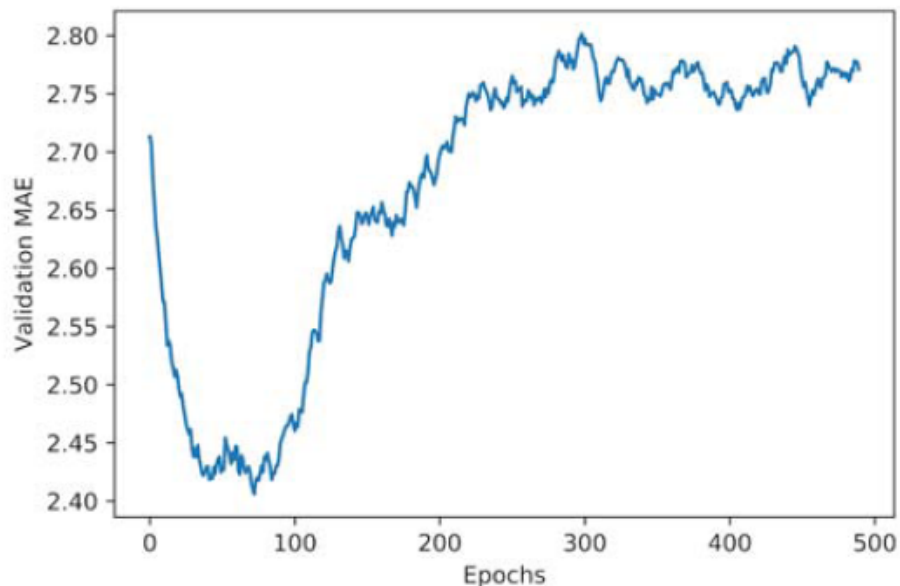


Figure 3.13 Validation MAE by epoch, excluding the first 10 data points

Listing 3.32 Training the final model

```
model = build_model()
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)
```

← Gets a fresh, compiled model

← Trains it on the entirety of the data

Here's the final result:

```
>>> test_mae_score
2.5532484335057877
```

You're still off by about \$2,550.

Wrapping up

Here's what you should take away from this example:

- Regression is done using different loss functions than what we used for classification. Mean squared error (MSE) is a loss function commonly used for regression.
- Similarly, evaluation metrics to be used for regression differ from those used for classification; naturally, the concept of accuracy doesn't apply for regression. A common regression metric is mean absolute error (MAE).
- When features in the input data have values in different ranges, each feature should be scaled independently as a preprocessing step.
- When there is little data available, using K-fold validation is a great way to reliably evaluate a model.
- When little training data is available, it's preferable to use a small network with few hidden layers (typically only one or two), in order to avoid severe overfitting.