Deep Learning

Lecture Topic:
Getting Started with Neural Networks: Classification and Regression

Anxiao (Andrew) Jiang

Learning Objectives:

1. Understand deep learning for binary classification.

2. Understand deep learning for multi-class classification.

3. Understand deep learning for regression.

Roadmap of this lecture:

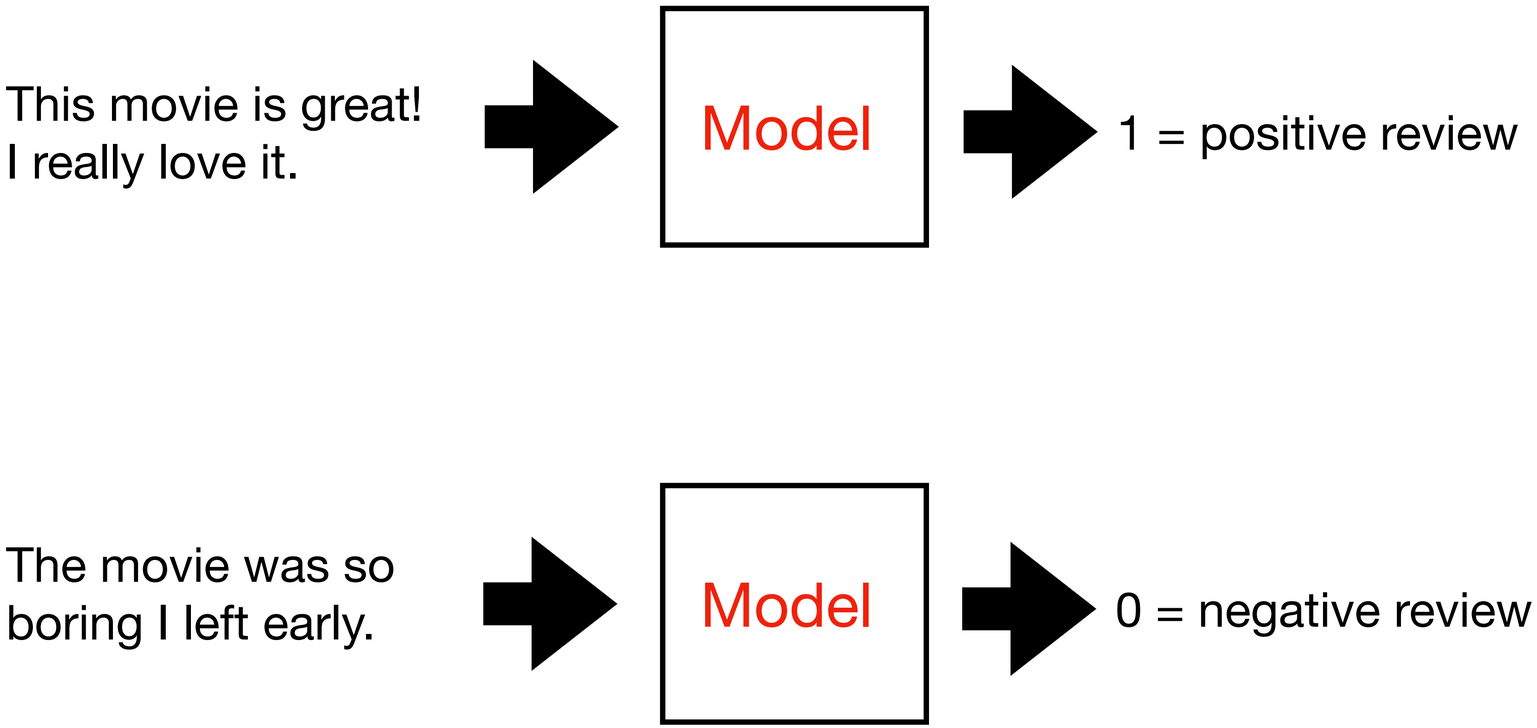
1. Binary classification of movie reviews.

2. Multi-class classification of newswires.

3. Regression for predicting house prices.

Binary Classification

Example of Binary Classification: Classifying Movie Reviews



Get dataset

IMDB dataset: 50,000 highly polarized reviews on movies.

Balanced dataset:

25,000 reviews for training, and 25,000 reviews for testing.
Each has 50% positive reviews, and 50% negative reviews.

3)

Samples of IMDB dataset:

One of the other reviewers has mentioned that after watching just 1 Oz episode you'll be hooked. The...

Label: positive

A wonderful little production.The filming technique is very unassuming- very old-time-B...

Label: positive

Phil the Alien is one of those quirky films where the humour is based around the oddness of everything...

Label: negative

Get dataset

Listing 4.1 Loading the IMDB dataset

```
from tensorflow.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 us to work with vector data of manageable size. If we didn't set this limit, we'd be working with 88,585 unique words in the training data, which is unnecessarily large. Many of these words only occur in a single sample, and thus can't be meaningfully used for classification.

Get dataset

Listing 4.1 Loading the IMDB dataset

```
from tensorflow.keras.datasets import imdb
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(
    num_words=10000)
```

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
```

Each word has been mapped to an integer.

Prepare the data

Make input samples have the same size.

- *Multi-hot encode* your lists to turn them into vectors of 0s and 1s. This would mean, for instance, turning the sequence `[8, 5]` into a 10,000-dimensional vector that would be all 0s except for indices 8 and 5, which would be 1s.

Prepare the data

Listing 4.3 Encoding the integer sequences via multi-hot encoding

```
import numpy as np
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        for j in sequence:
            results[i, j] = 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.])
```

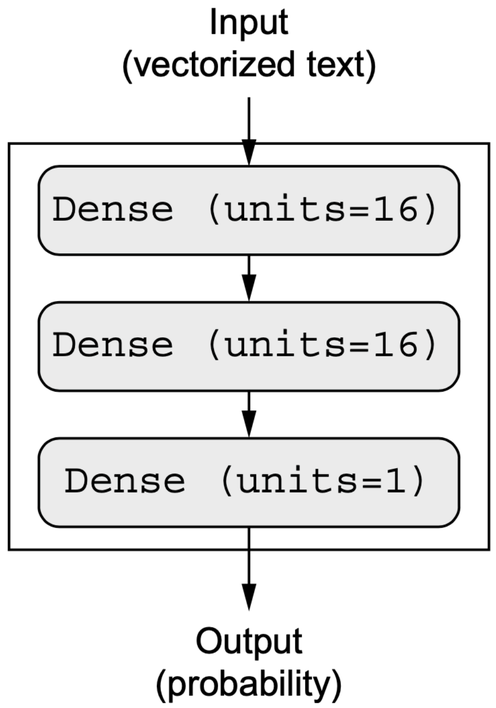
x_train and x_test: 2-d arrays.

Prepare the data

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")
```

Build the model



- Two intermediate layers with 16 units each
- A third layer that will output the scalar prediction regarding the sentiment of the current review

Build the model

Listing 4.4 Model definition

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
```

RELU and Sigmoid

$$\text{relu}(x) = \max(x, 0)$$

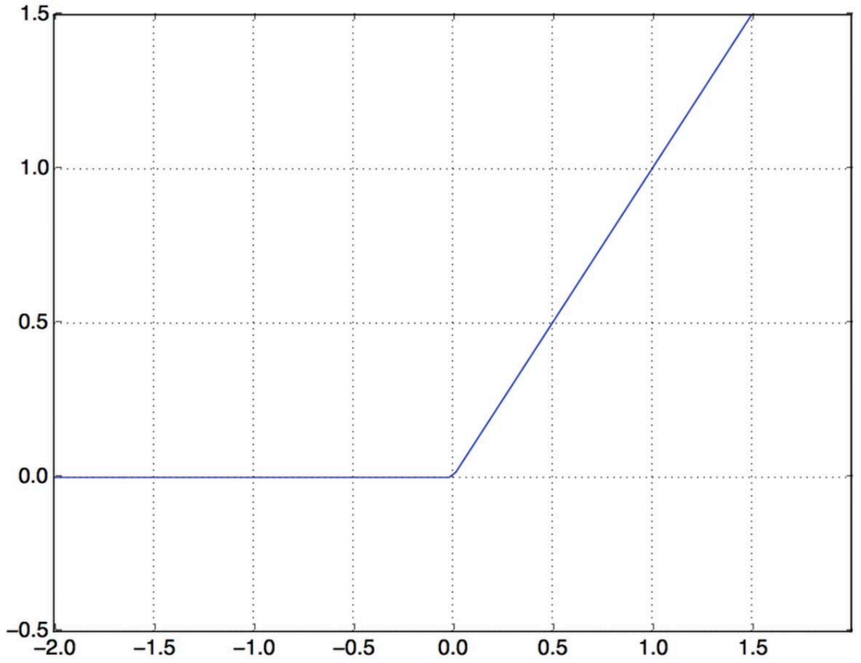


Figure 4.2 The rectified linear unit function

RELU and Sigmoid

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

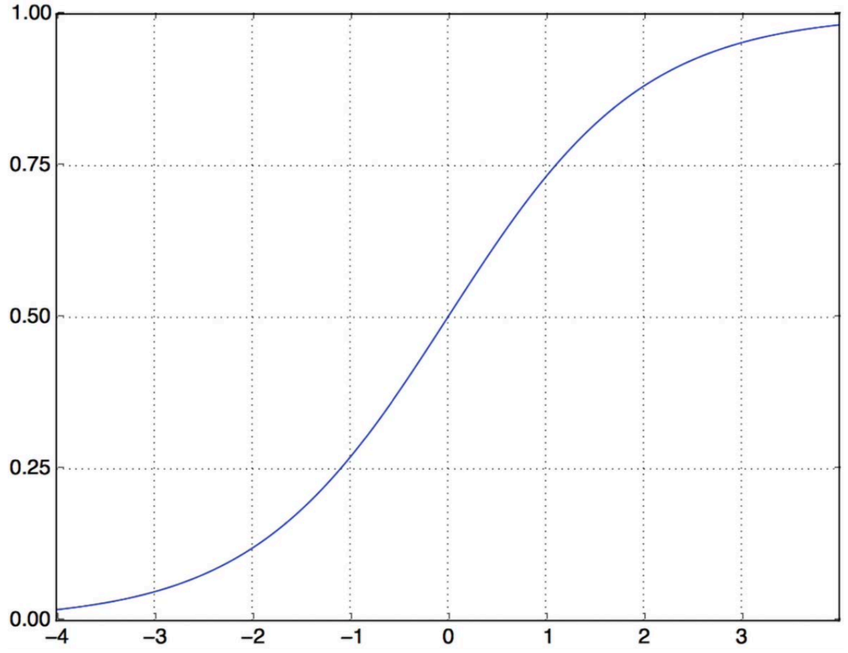
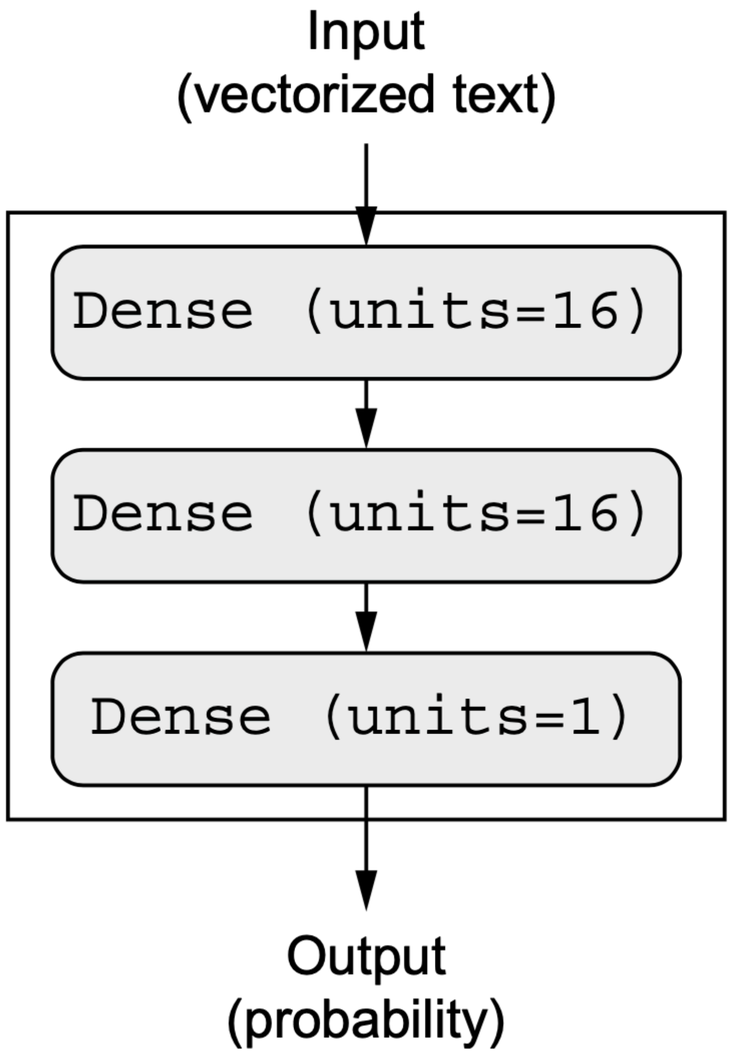


Figure 4.3 The sigmoid function

Build the model



$$X_{(10000,)}$$

$$A_{(16,)} = X_{(10000,)} W_{(10000,16)} + b_{(16,)}$$

$$B_{(16,)} = \text{relu}(A_{(16,)})$$

$$C_{(16,)} = B_{(16,)} \hat{W}_{(16,16)} + \hat{b}_{(16,)}$$

$$D_{(16,)} = \text{relu}(C_{(16,)})$$

$$E_{(1,)} = D_{(16,)} \bar{W}_{(16,1)} + \bar{b}_{(1,)}$$

$$F_{(1,)} = \text{sigmod}(E_{(1,)})$$

Build the model

Listing 4.4 Model definition

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
```

Compile the model

Listing 4.5 Compiling the model

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

Compile the model

$$\text{binary crossentropy} = -\frac{1}{N}\sum_{i=1}^{N} L_i \log y_i + (1 - L_i)\log(1 - y_i)$$

N : the number of samples in a mini-batch

$L_i \in \{0,1\}$: label of the i-th sample

$y_i \in (0,1)$: model's output for the i-th sample

The binary crossentropy is minimized when $y_i = L_i$ for all i

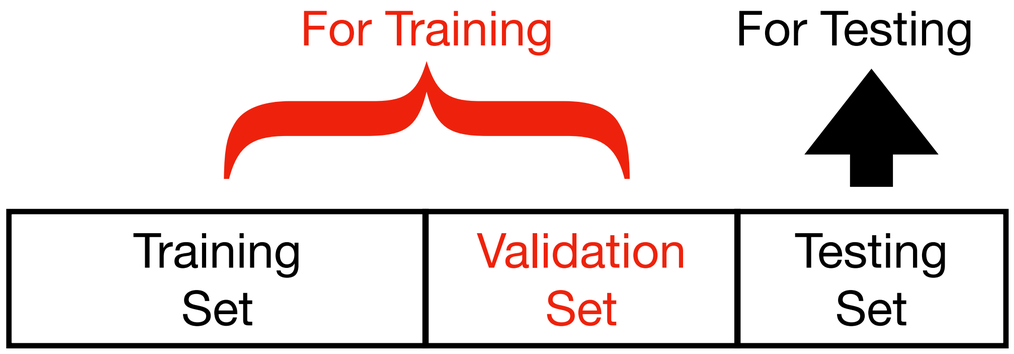
Compile the model

Listing 4.5 Compiling the model

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

Accuracy: the fraction of samples that are correctly classified.

Validation Set



Listing 4.6 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:]
```

Train the model with Validation Set

Listing 4.7 Training your model

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

Check performance of training

Listing 4.7 Training your model

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

Note that the call to `model.fit()` returns a `History` object, as you saw in chapter 3. This object has a member `history`, which is a dictionary containing data about everything that happened during training. Let's look at it:

```
>>> history_dict = history.history
>>> history_dict.keys()
[u"accuracy", u"loss", u"val_accuracy", u"val_loss"]
```

Plot loss function and accuracy

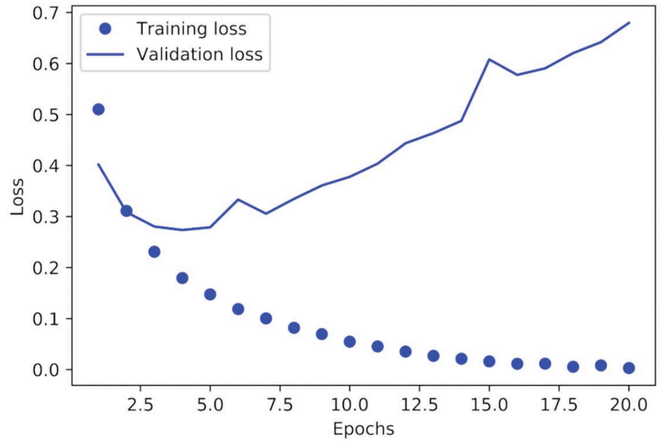


Figure 4.4 Training and validation loss

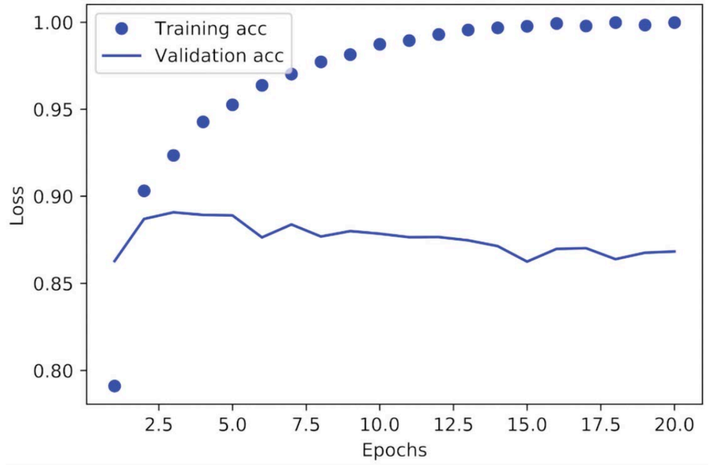


Figure 4.5 Training and validation accuracy

Overfit

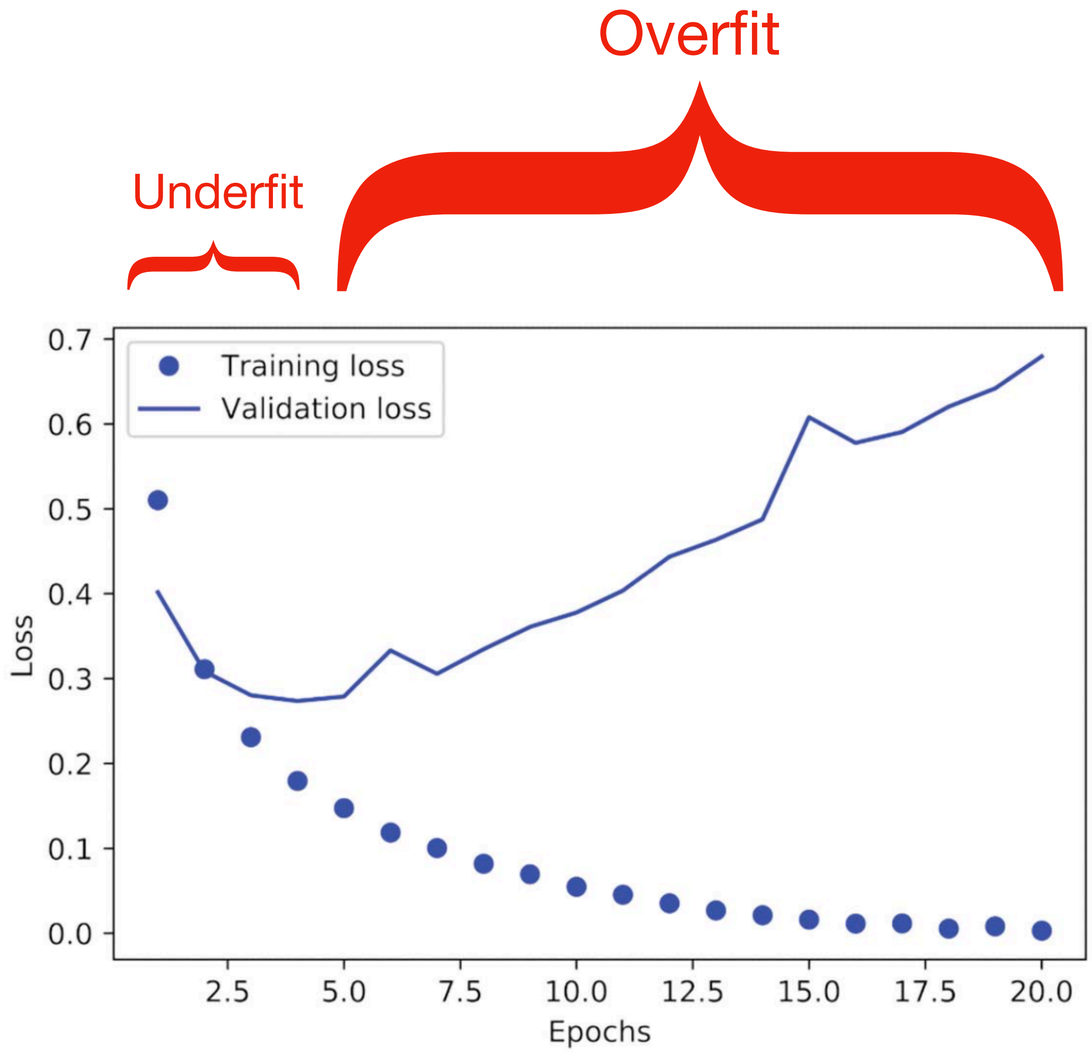


Figure 4.4 Training and validation loss

Overfit

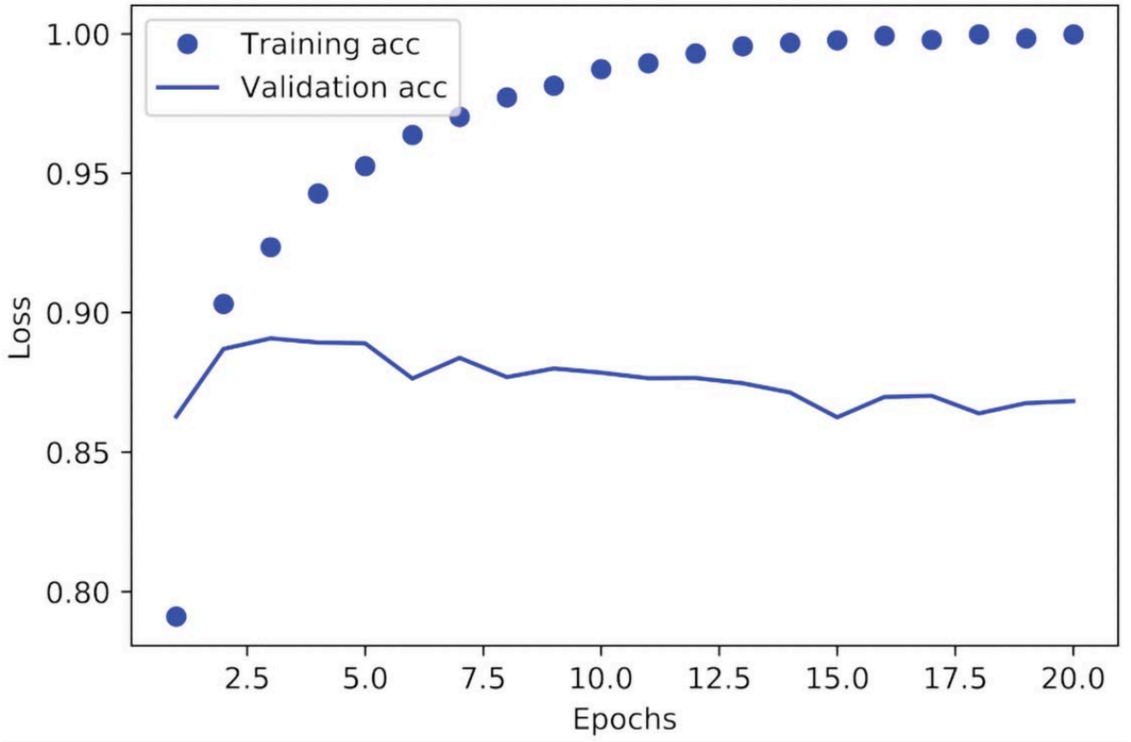


Figure 4.5 Training and validation accuracy

Overfitting happened after 4 epochs, so we re-train the model for 4 epochs.

We can use validation data in training set, too, this time.

Listing 4.10 Retraining a model from scratch

```
model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    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, 0.88327999999999995]
```

The first number, 0.29, is the test loss, and the second number, 0.88, is the test accuracy.

Use the trained model to generate prediction on new data

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

Quiz questions:

1. What is the importance of having a balanced dataset?

2. What is the use of multi-hot encoding?

3. How to tell if overfitting happened during training?

Roadmap of this lecture:

1. Binary classification of movie reviews.

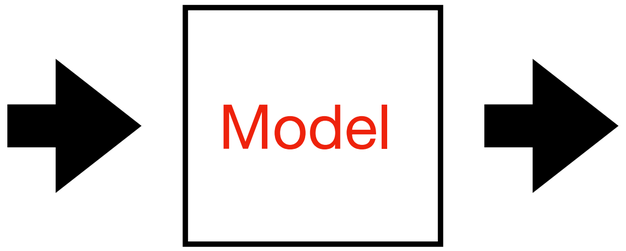
2. Multi-class classification of newswires.

3. Regression for predicting house prices.

Multi-Class Classification

Example of Multi-Class Classification: Classifying Newswires

The copper price
just went up again
for the third straight
day. The stock market
…



Topic 0: cocoa
Topic 1: grain
Topic 2: veg-oil
Topic 3: earn
Topic 4: acq
Topic 5: wheat
Topic 6: copper
Topic 7: housing
Topic 8: money-supply
Topic 9: coffee
Topic 10: sugar
Topic 11: trade
Topic 12: reserves
Topic 13: ship
Topic 14: cotton
Topic 15: carcass
…
Topic 45: lead

Get 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.

Listing 4.11 Loading the Reuters dataset

```
from tensorflow.keras.datasets import reuters
(train_data, train_labels), (test_data, test_labels) = reuters.load_data(
    num_words=10000)
```

Get Dataset

You have 8,982 training examples and 2,246 test examples:

```
>>> 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]
```

The label associated with an example is an integer between 0 and 45—a topic index:

```
>>> train_labels[10]
3
```

Prepare the data: vectorize data using Multi-Hot Encoding

You can vectorize the data with the exact same code as in the previous example.

Listing 4.13 Encoding the input data

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

Vectorized training data

Vectorized test data

Prepare the data: vectorize labels using One-Hot Encoding

Listing 4.14 Encoding the labels

```
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
y_train = to_one_hot(train_labels)
y_test = to_one_hot(test_labels)
```

Vectorized training labels

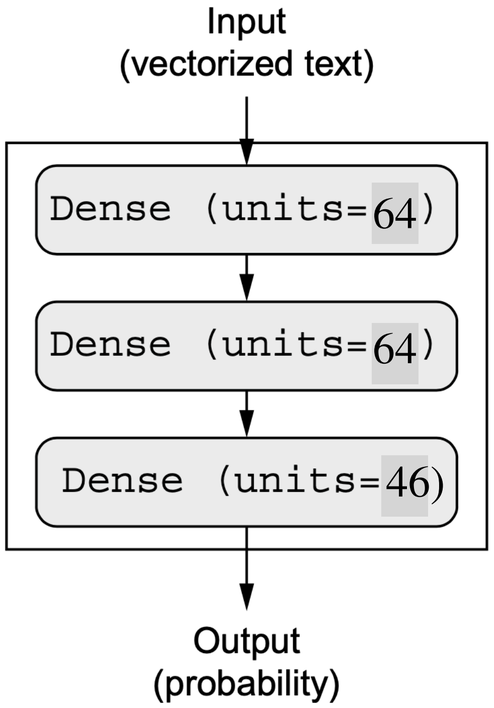
Vectorized test labels

One-Hot Encoding is also called "categorical encoding".

Note that there is a built-in way to do this in Keras:

```
from tensorflow.keras.utils import to_categorical
y_train = to_categorical(train_labels)
y_test = to_categorical(test_labels)
```

Build the model



Listing 4.15 Model definition

```
model = keras.Sequential([
    layers.Dense(64, activation="relu"),
    layers.Dense(64, activation="relu"),
    layers.Dense(46, activation="softmax")
])
```

RELU and Softmax

$$\text{relu}(x) = \max(x, 0)$$

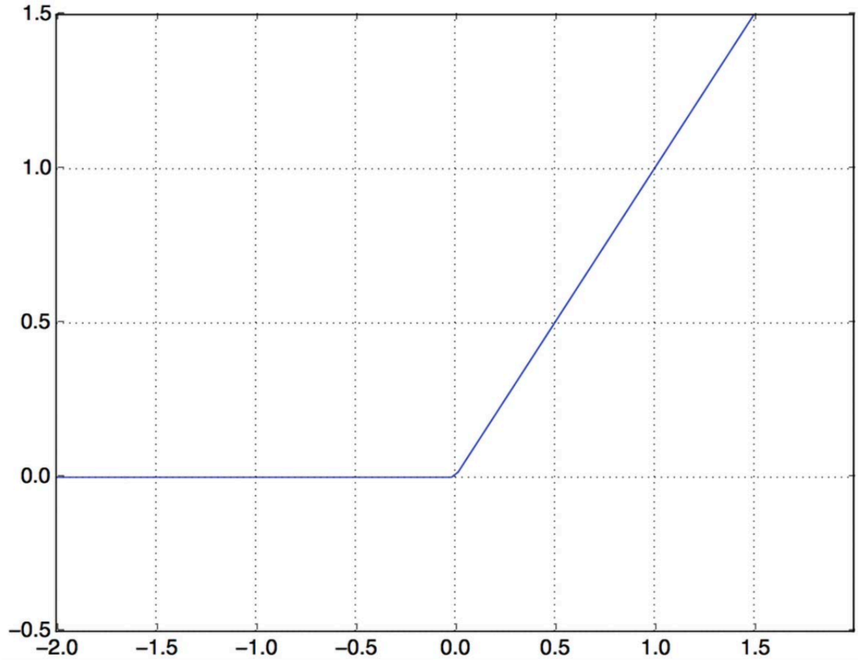
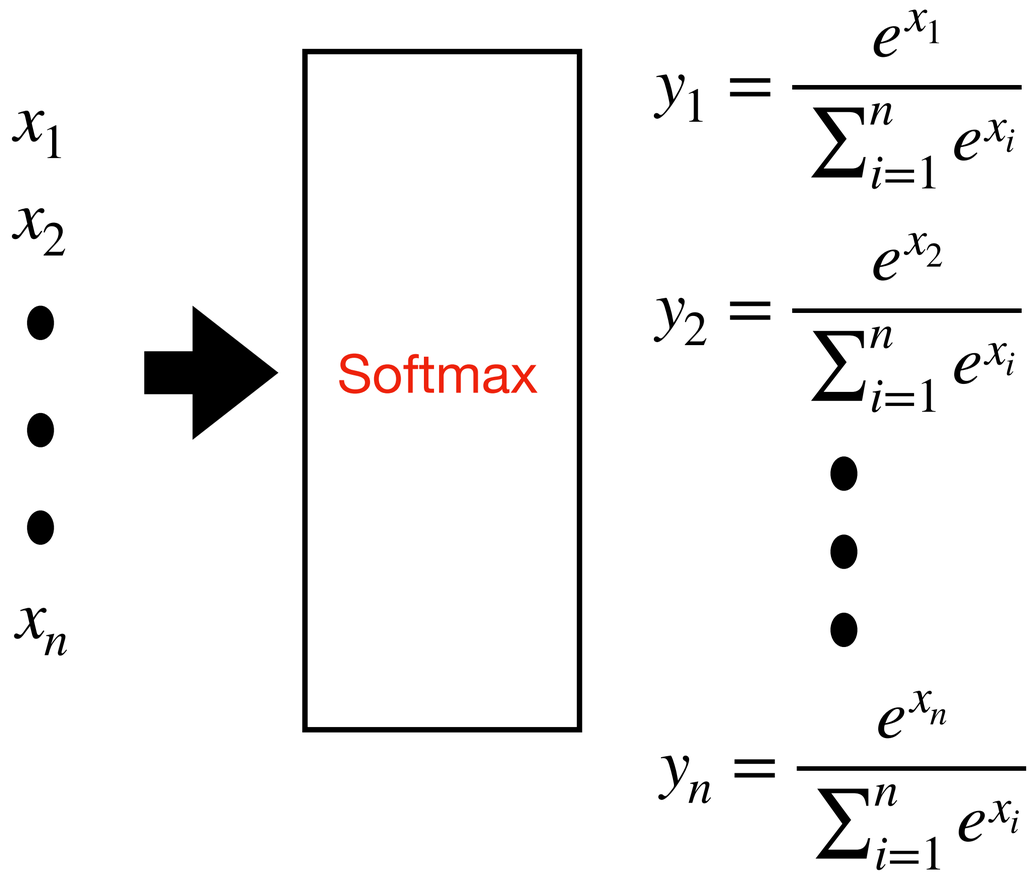


Figure 4.2 The rectified linear unit function

RELU and Softmax



Compile the model

Listing 4.16 Compiling the model

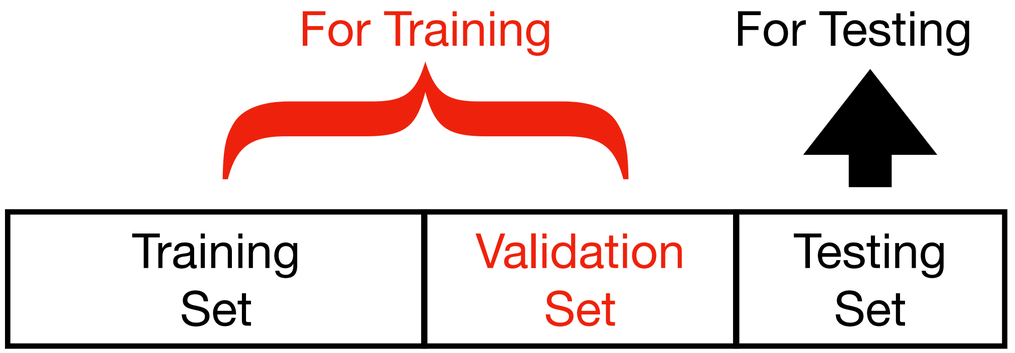
```
model.compile(optimizer="rmsprop",
              loss="categorical_crossentropy",
              metrics=["accuracy"])
```

categorical crossentropy =

$$-\frac{1}{N}\sum_{i=1}^{N}\left(L_{i,1}\log y_{i,1} + L_{i,2}\log y_{i,2} + \cdots + L_{i,n}\log y_{i,n}\right)$$

The categorical crossentropy is minimized when $y_{i,j} = L_{i,j}$ for all i, j

Validation Set, and Train the model



Listing 4.17 Setting aside a validation set

```
x_val = x_train[:1000]
partial_x_train = x_train[1000:]
y_val = y_train[:1000]
partial_y_train = y_train[1000:]
```

Validation Set, and Train the model

Listing 4.18 Training the model

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

Check performance of training

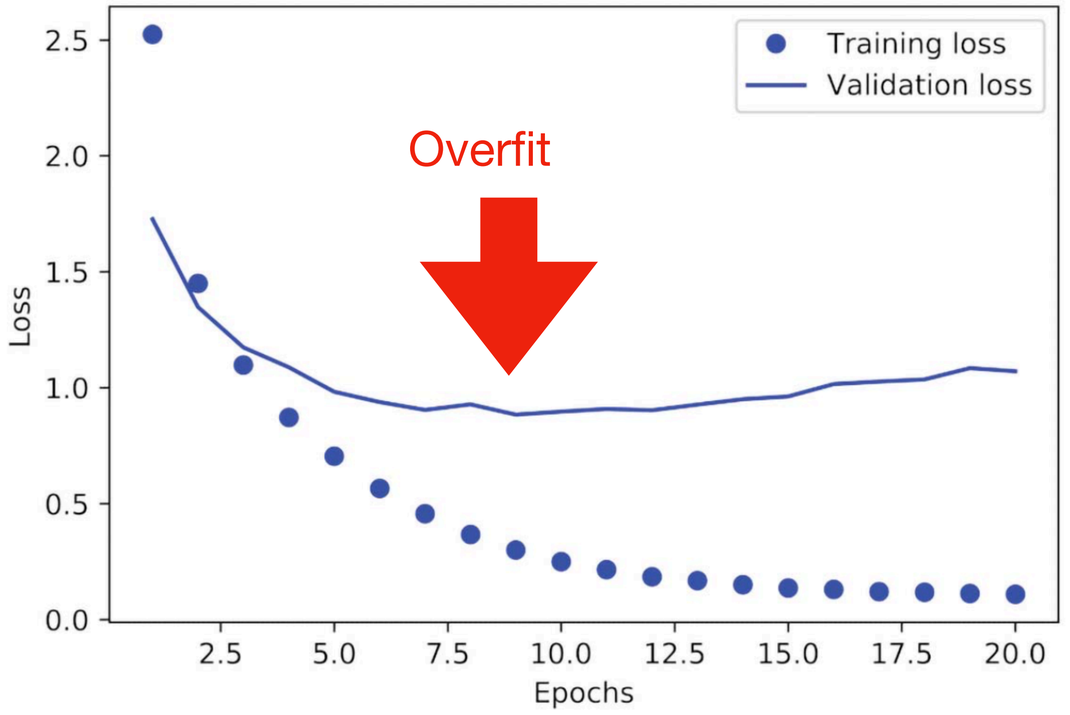


Figure 4.6 Training and validation loss

Check performance of training

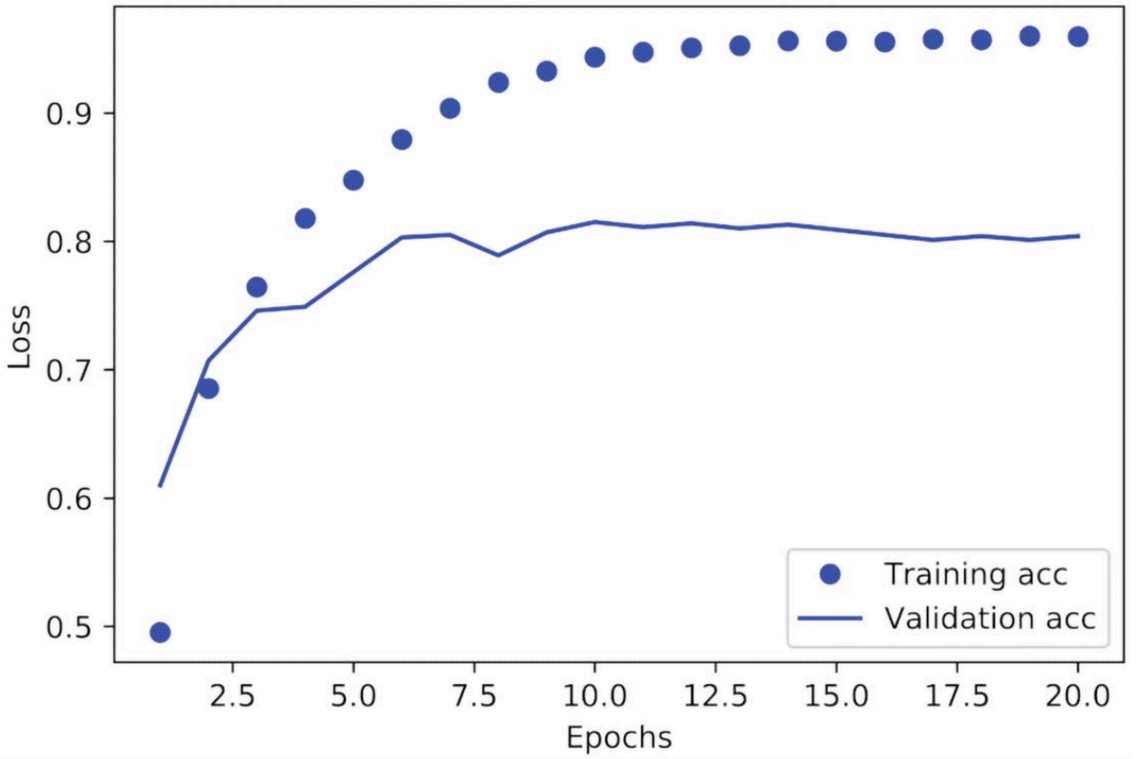


Figure 4.7 Training and validation accuracy

Re-train the model (also using validation data) for 9 epochs

Listing 4.21 Retraining a model from scratch

```
model = keras.Sequential([
    layers.Dense(64, activation="relu"),
    layers.Dense(64, activation="relu"),
    layers.Dense(46, activation="softmax")
])
model.compile(optimizer="rmsprop",
              loss="categorical_crossentropy",
              metrics=["accuracy"])
model.fit(x_train,
          y_train,
          epochs=9,
          batch_size=512)
results = model.evaluate(x_test, y_test)
```

Here are the final results:

```
>>> results
[0.9565213431445807, 0.7969723953695458]
```

Loss Accuracy

Use the trained model to generate predictions on new data

Calling the model's `predict` method on new samples returns a class probability distribution over all 46 topics for each sample. Let's generate topic predictions for all of the test 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, as they form a probability distribution:

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

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

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

A “lazy” way to handle the labels (without calling one-hot encoding)

Each label is an integer, not a one-hot encoded vector

Array List

```
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 4.21, `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=["accuracy"])
```

This new loss function is still mathematically the same as `categorical_crossentropy`; it just has a different interface.

Quiz questions:

1. Why do we use cross-entropy as the loss function?

2. What is the use of the “softmax” activation function?

Roadmap of this lecture:

1. Binary classification of movie reviews.

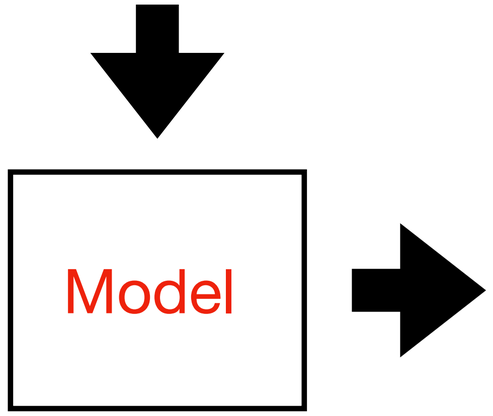
2. Multi-class classification of newswires.

3. Regression for predicting house prices.

Regression

Example of Regression: Predict house price

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



Average housing price:
$42.3K

Get Dataset

Boston Housing Price Dataset: only 506 samples, where each sample has 13 numerical features.

Listing 4.23 Loading the Boston housing dataset

```
from tensorflow.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)
```

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]
```

Prepare the data

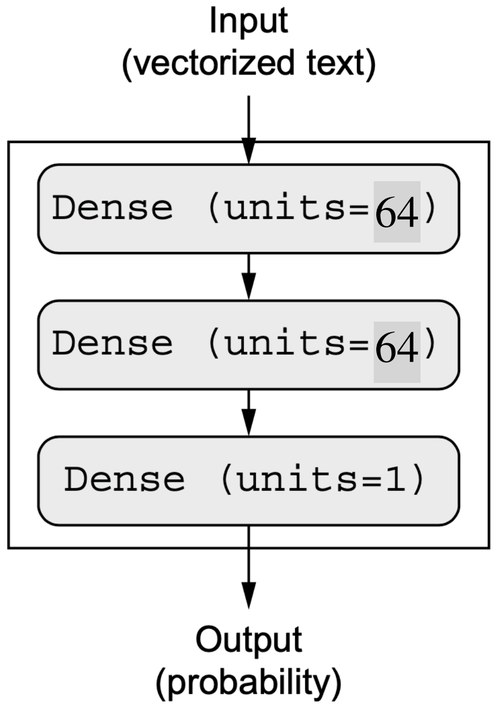
Normalize the data so that each feature has mean 0 and standard deviation 1.

Listing 4.24 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
```

Build the model, and compile the model



Build the model, and compile the model

Listing 4.25 Model definition

```
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
```

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

MSE and MAE

MSE: mean square error

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(L_i - y_i)^2$$

MAE: mean absolute error

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|L_i - y_i|$$

K-fold validation for small dataset

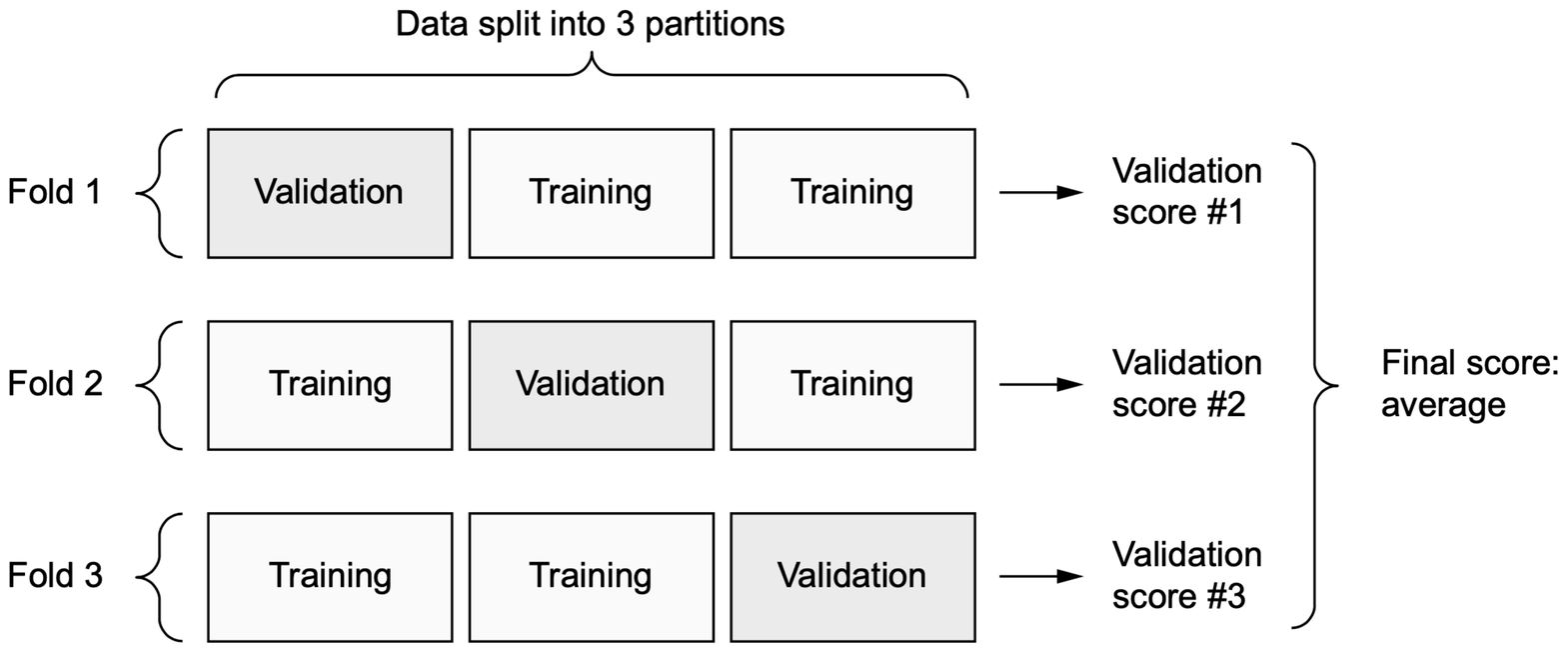


Figure 4.8 K-fold cross-validation with K=3

K-fold validation for small dataset

Listing 4.26 K-fold validation

```
k = 4
num_val_samples = len(train_data) // k
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)
    val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
    all_scores.append(val_mae)
```

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)

Evaluates the model on the validation data

Train the model

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

```
>>> all_scores
[2.112449, 3.0801501, 2.6483836, 2.4275346]
>>> np.mean(all_scores)
2.5671294
```

Train the model with more epochs, and find out when overfitting happens

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.

Listing 4.27 Saving the validation logs at each fold

```
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)
```

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)

Plot performance

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

Listing 4.28 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)]
```

Let's plot this; see figure 4.9.

Listing 4.29 Plotting validation scores

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

Plot performance

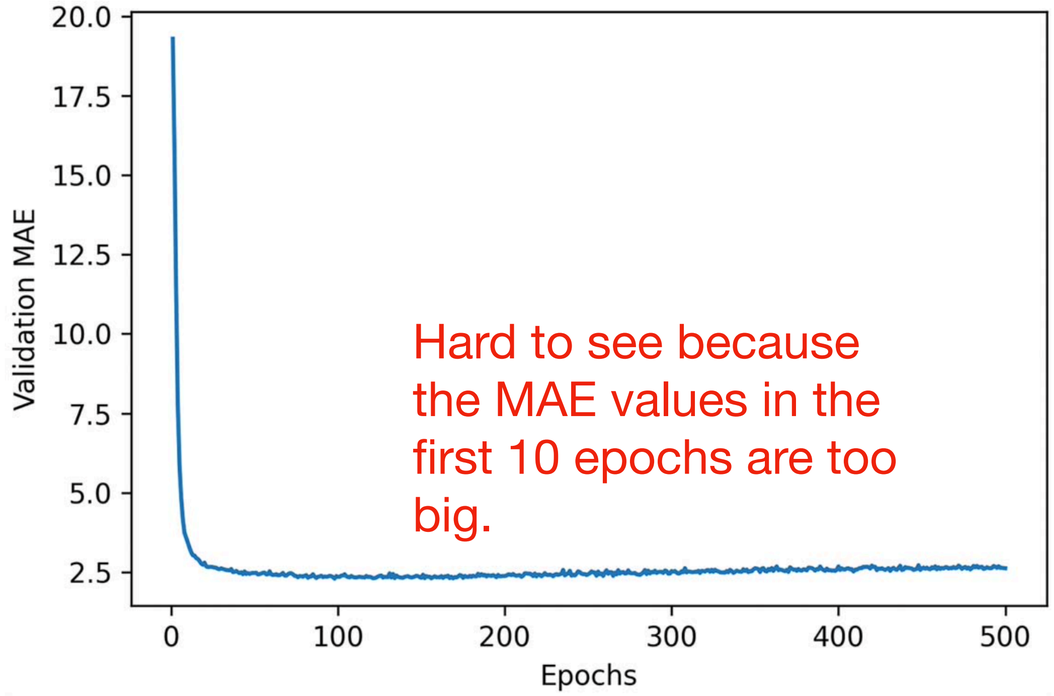


Figure 4.9 Validation MAE by epoch

Skip the first 10 epochs and plot the loss again

Listing 4.30 Plotting validation scores, excluding the first 10 data points

```
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()
```

Skip the first 10 epochs and plot the loss again

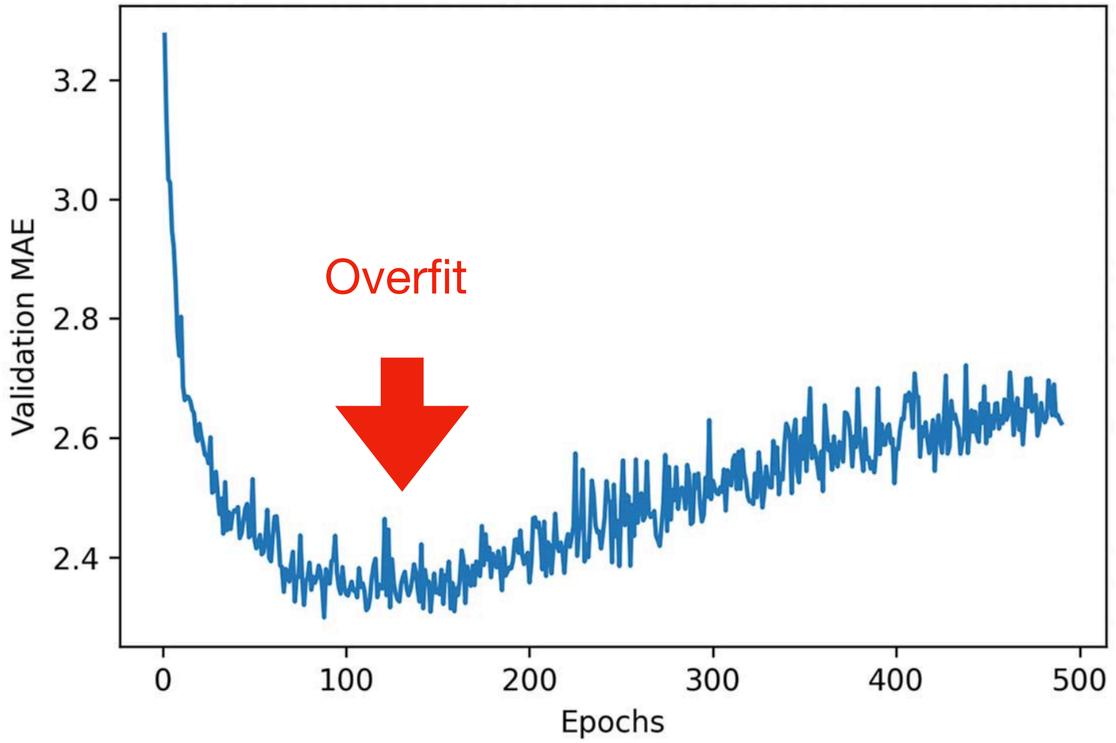


Figure 4.10 Validation MAE by epoch, excluding the first 10 data points

Re-train the model using all data, then test it

Listing 4.31 Training the final model

```
model = build_model()
model.fit(train_data, train_targets,
          epochs=130, 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
2.4642276763916016
```

Use the trained model to generate predictions on new data

```
>>> predictions = model.predict(test_data)
>>> predictions[0]
array([9.990133], dtype=float32)
```

Quiz questions:

1. What is the use of K-fold validation?

2. What are the differences between regression and classification?