Machine Learning

Classification and Regression



Classification and Regression *Problems*

1. Binary classification
   Classifying movie reviews as positive or negative
2. Multiclass classification
   Classifying news wires by topic
3. Scalar regression
   Estimating the price of a house, given real-estate data

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. For example, when classifying cat and dog pictures, "dog" and "cat" are the two classes.
- *Label* —A specific instance of a class annotation in a classification problem. For instance, if picture #1234 is annotated as containing the class "dog," then "dog" is a label of picture #1234.

Classification and Regression *Glossary*

- *Ground-truth* or *annotations*—All targets for a dataset, typically collected by humans.
- *Binary classification*—A classification task where each input sample should be categorized into two exclusive categories.
- *Multiclass classification*—A classification task where each input sample should be categorized into more than two categories: for instance, classifying handwritten digits.
- *Multilabel classification*—A classification task where 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. The number of labels per image is usually variable.

Classification and Regression *Glossary*

- *Scalar regression*—A task where the target is a continuous scalar value. Predicting house prices is a good example: the different target prices form a continuous space.
- *Vector regression*—A task where the target is a set of continuous values: for example, a continuous vector. If you're doing regression against multiple values (such as the coordinates of a bounding box in an image), then you're doing vector regression.
- *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.

Binary Classification: IMDB *dataset*

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.

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

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

Binary Classification: IMDB *dataset*

Preparing the data

```
import numpy as np
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))   ❶
    for i, sequence in enumerate(sequences):
        for j in sequence:
            results[i, j] = 1.                        ❷
    return results
x_train = vectorize_sequences(train_data)             ❸
x_test = vectorize_sequences(test_data)               ❹
```

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

❷ Sets specific indices of results[i] to 1s

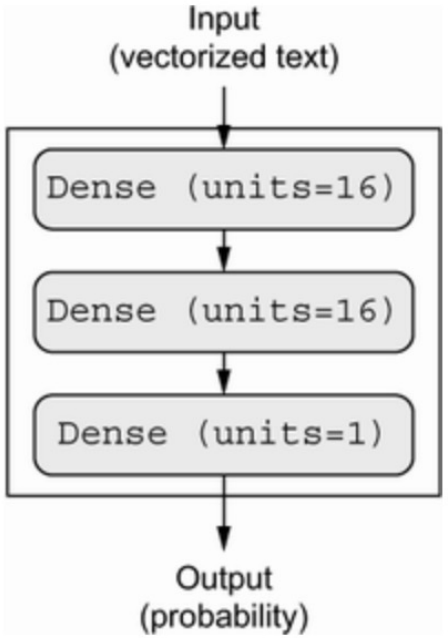
❸ Vectorized training data

❹ Vectorized test data

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

Binary Classification: IMDB *dataset*

Building the Model

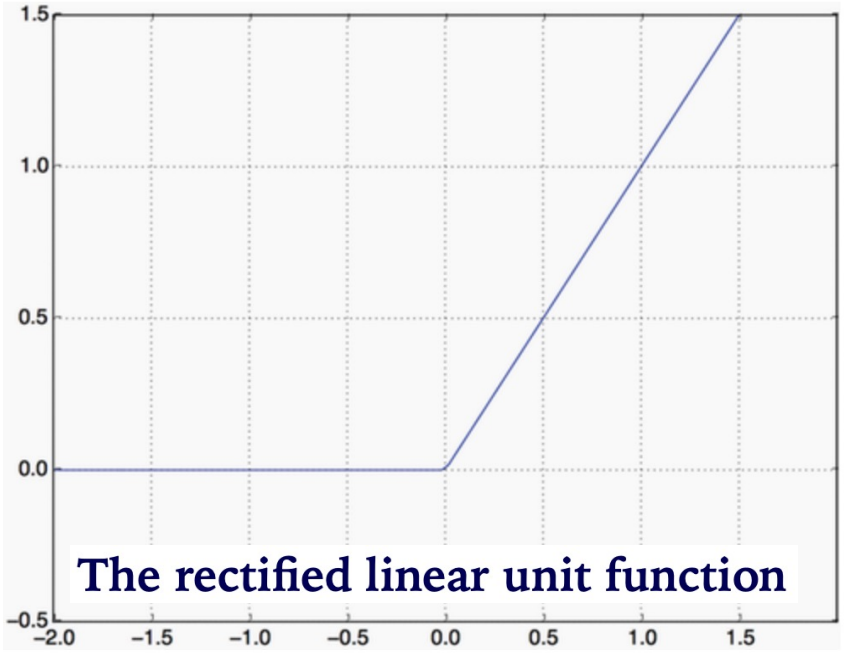


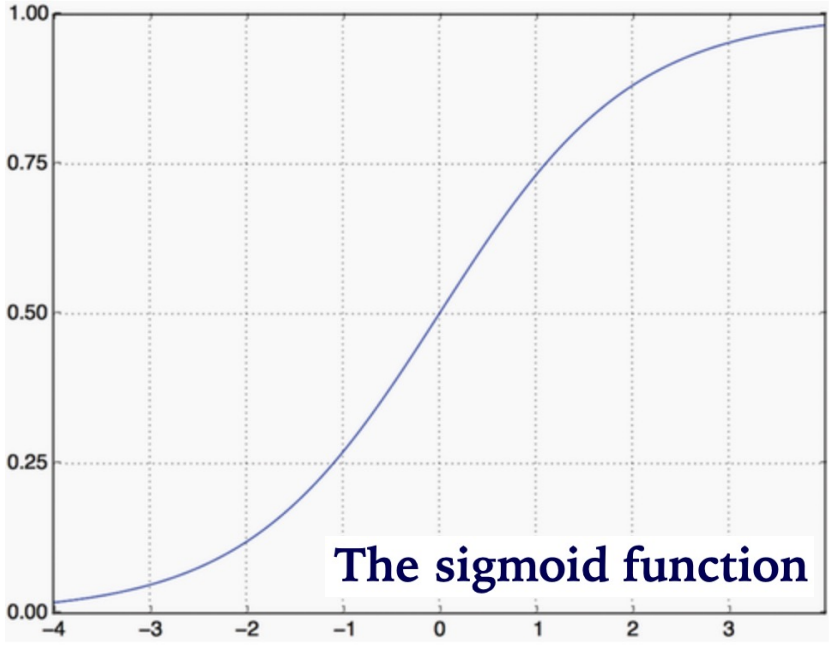

```
from tensorflow import keras
from tensorflow.keras import layers
```

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

```
output = relu(dot(input, W) + b)
```

Activation Function, *why is it necessary*?





Without an activation function like relu (also called a non-linearity),
the Dense layer would consist of two linear operations—a dot product and an addition

```
output = relu(dot(input, W) + b)
```

Binary Classification: IMDB *dataset*

Compiling the Model

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

Validating the Model

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

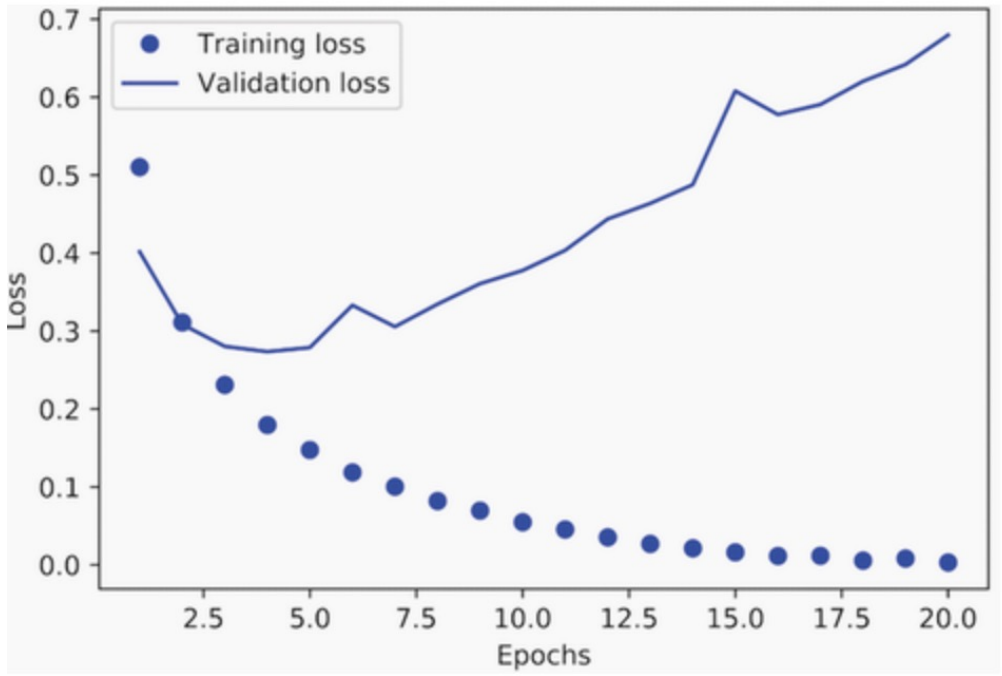
Training the Model

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

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

Binary Classification: IMDB *dataset*

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

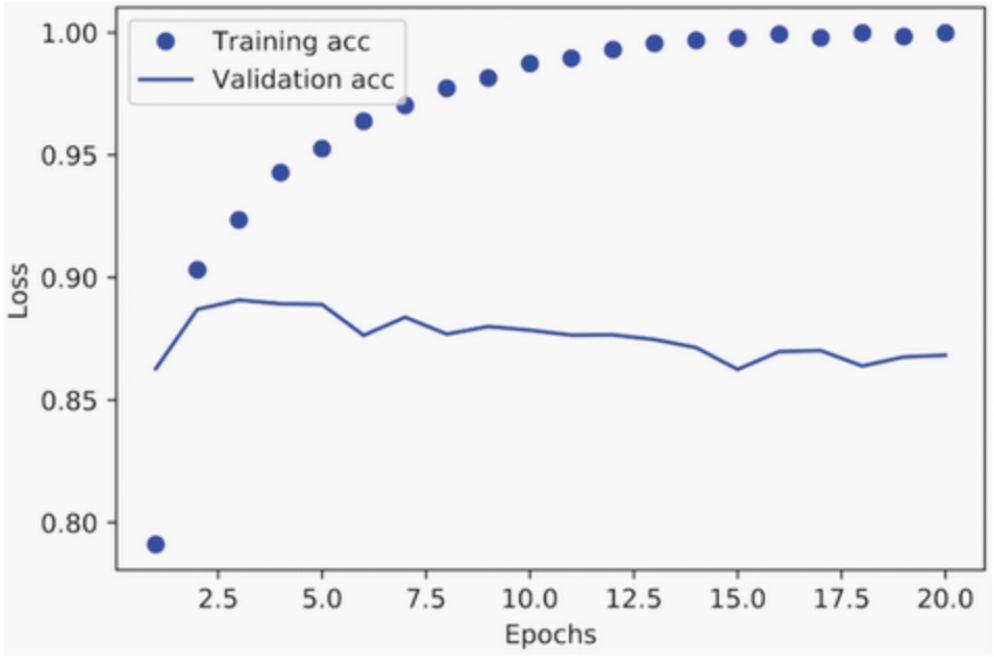



```
import matplotlib.pyplot as plt
history_dict = history.history
loss_values = history_dict["loss"]
val_loss_values = history_dict["val_loss"]
```

```
epochs = range(1, len(loss_values) + 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()
```

Binary Classification: IMDB *dataset*

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




```
plt.clf()                                    ❶
acc = history_dict["accuracy"]
val_acc = history_dict["val_accuracy"]
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("Accuracy")
plt.legend()
plt.show()
```

❶ Clears the figure

Binary Classification: IMDB *dataset*

Retraining the 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)
```

```
>>> results
[0.2929924130630493, 0.8832799999999995]
```

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

This fairly naive approach achieves an accuracy of **88%**. With state-of-the-art approaches, you should be able to get close to **95%**

Binary Classification: IMDB *dataset*

Generating predictions on new data

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

The model is confident for some samples (0.99 or more, or 0.01 or less)
but less confident for others (0.6, 0.4)

A Multiclass Classification: Reuters *dataset*

- We will build a model to classify Reuters newswires into 46 mutually exclusive topics.
- As we have many classes, this problem is an instance of **multiclass classification**;
- Each data point should be classified into only one category, so the problem is more specifically an instance of single-label multiclass classification.

If each data point could belong to multiple categories (in this case, topics), we'd be facing a **multilabel multiclass classification problem**.

A Multiclass Classification: Reuters *dataset*

```
from tensorflow.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
```

```
>>> 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
```

A Multiclass Classification: Reuters *dataset*

Preparing the data

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

❶ Vectorized training data

❷ Vectorized test data

Encoding the labels

To vectorize the labels, there are two possibilities:

- Cast the label list as an integer tensor;
- Use *one-hot encoding* also called *categorical encoding*. In this case, one-hot encoding of the labels consists of embedding each label as an all-zero vector with a 1 in the place of the label index.

A Multiclass Classification: Reuters *dataset*

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)
```

❶ Vectorized training labels

❷ Vectorized test labels

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)
```

A Multiclass Classification: Reuters *dataset*

Building the Model

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

If one layer drops some information relevant to the classification problem, this information can never be recovered by later layers: each layer can potentially become an information bottleneck.
For this reason, we will use larger layers (64 units)

***Softmax* activation**: It means the model will output a probability distribution over the 46 different output classes—for every input sample, the model will produce a 46-dimensional output vector, where output[i] is the probability that the sample belongs to class i. The 46 scores will sum to 1."

A Multiclass Classification: Reuters *dataset*

Compiling the Model

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

The best loss function to use in this case is categorical_crossentropy. It measures the distance between two probability distributions: here, between the probability distribution output by the model and the true distribution of the labels.

Validating the approach

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

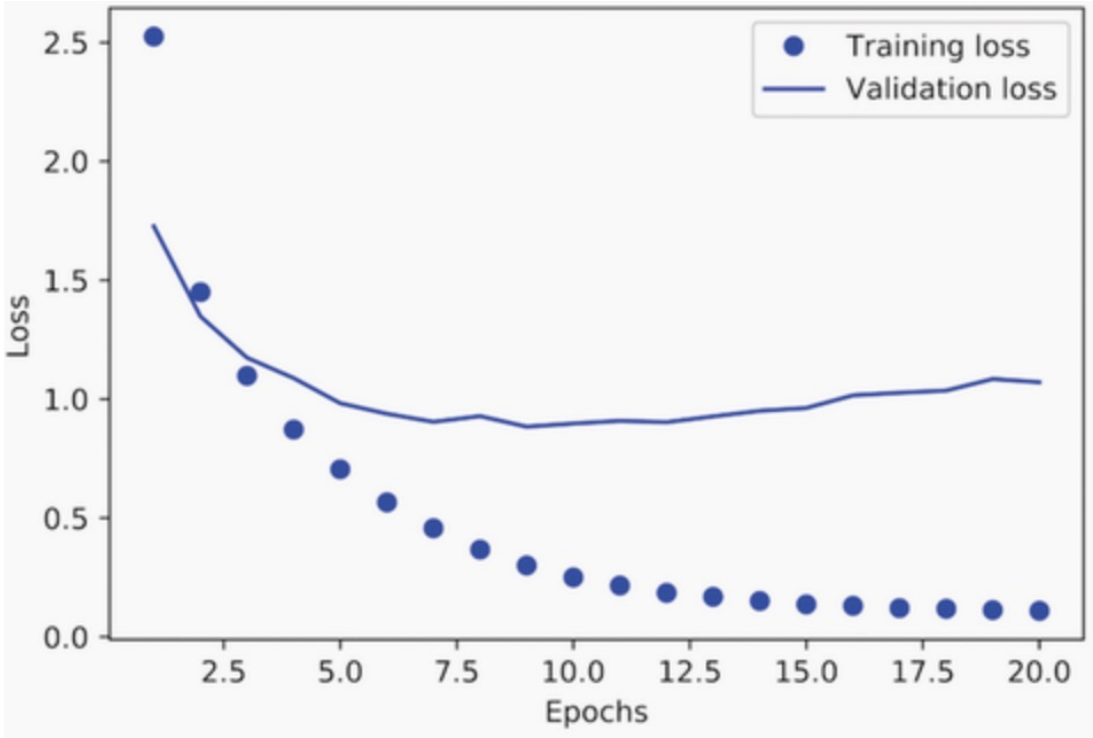
A Multiclass Classification: Reuters *dataset*

Training the Model

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

A Multiclass Classification: Reuters *dataset*

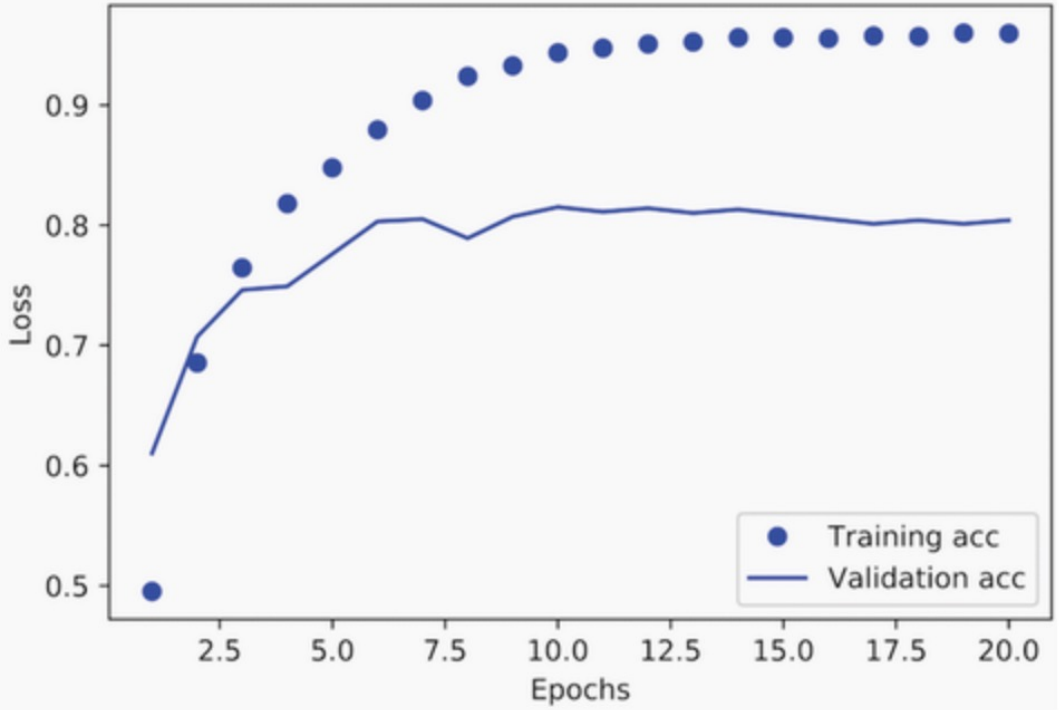
Training and Validation Loss Model




```
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()
```

A Multiclass Classification: Reuters *dataset*

Training and Validation Loss Model




```
plt.clf()                                                    ❶
acc = history.history["accuracy"]
val_acc = history.history["val_accuracy"]
plt.plot(epochs, acc, "bo", label="Training accuracy")
plt.plot(epochs, val_acc, "b", label="Validation accuracy")
plt.title("Training and validation accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```

A Multiclass Classification: Reuters *dataset*

Retraining the 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)
```

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

Loss Accuracy

A Multiclass Classification: Reuters *dataset*

Comparing the model with a random classifier

- This approach reaches an accuracy of ~80%.
- With a balanced binary classification problem, the accuracy reached by a purely random classifier would be 50%.
- In this case, we have 46 classes, and they may not be equally represented. What would be the accuracy of a random baseline?

```
>>> import copy
>>> test_labels_copy = copy.copy(test_labels)
>>> np.random.shuffle(test_labels_copy)
>>> hits_array = np.array(test_labels) == np.array(test_labels_copy)
>>> hits_array.mean()
0.18655387355298308
```

A random classifier would score around 19% classification accuracy, so the results of our model seem pretty good in that light."

A Multiclass Classification: Reuters *dataset*

Comparing the model with a random classifier

- This approach reaches an accuracy of ~80%.
- With a balanced binary classification problem, the accuracy reached by a purely random classifier would be 50%.
- In this case, we have 46 classes, and they may not be equally represented. What would be the accuracy of a random baseline?

```
>>> import copy
>>> test_labels_copy = copy.copy(test_labels)
>>> np.random.shuffle(test_labels_copy)
>>> hits_array = np.array(test_labels) == np.array(test_labels_copy)
>>> hits_array.mean()
0.18655387355298308
```

A random classifier would score around 19% classification accuracy, so the results of our model seem pretty good in that light."

A Multiclass Classification: Reuters *dataset*

Generating predictions on new data

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

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

```
>>> 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 Scalar Regression: Predicting House Prices

The Boston housing price dataset

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

We 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.

```
>>> 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]
```

A Scalar Regression: Predicting House Prices

Preparing the data

Feature-wise normalization: for each feature in the input data (a column in the input data matrix), we subtract the mean of the feature and divide by the standard deviation, so that the feature is centered around 0 and has a unit standard deviation.

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

A Scalar Regression: Predicting House Prices

Building the Model

```
def build_model():
    model = keras.Sequential([                    ❶
        layers.Dense(64, activation="relu"),
        layers.Dense(64, activation="relu"),
        layers.Dense(1)
    ])
    model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
    return model
```

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

More about rmse, mse and mae Metrics

- The model ends with a **single unit and no activation** (it will be a linear layer);
- This is a typical setup for scalar regression (this case predicting a **single continuous value**);
- Applying an activation function would constrain the range the output can take;
- Applying a sigmoid activation function to the last layer, the model could only learn to predict values between 0 and 1;
- As the the last layer is purely linear, the model is free to learn to predict values in any range.

A Scalar Regression: Predicting House Prices

Building the Model

```
def build_model():
    model = keras.Sequential([                    ❶
        layers.Dense(64, activation="relu"),
        layers.Dense(64, activation="relu"),
        layers.Dense(1)
    ])
    model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
    return model
```

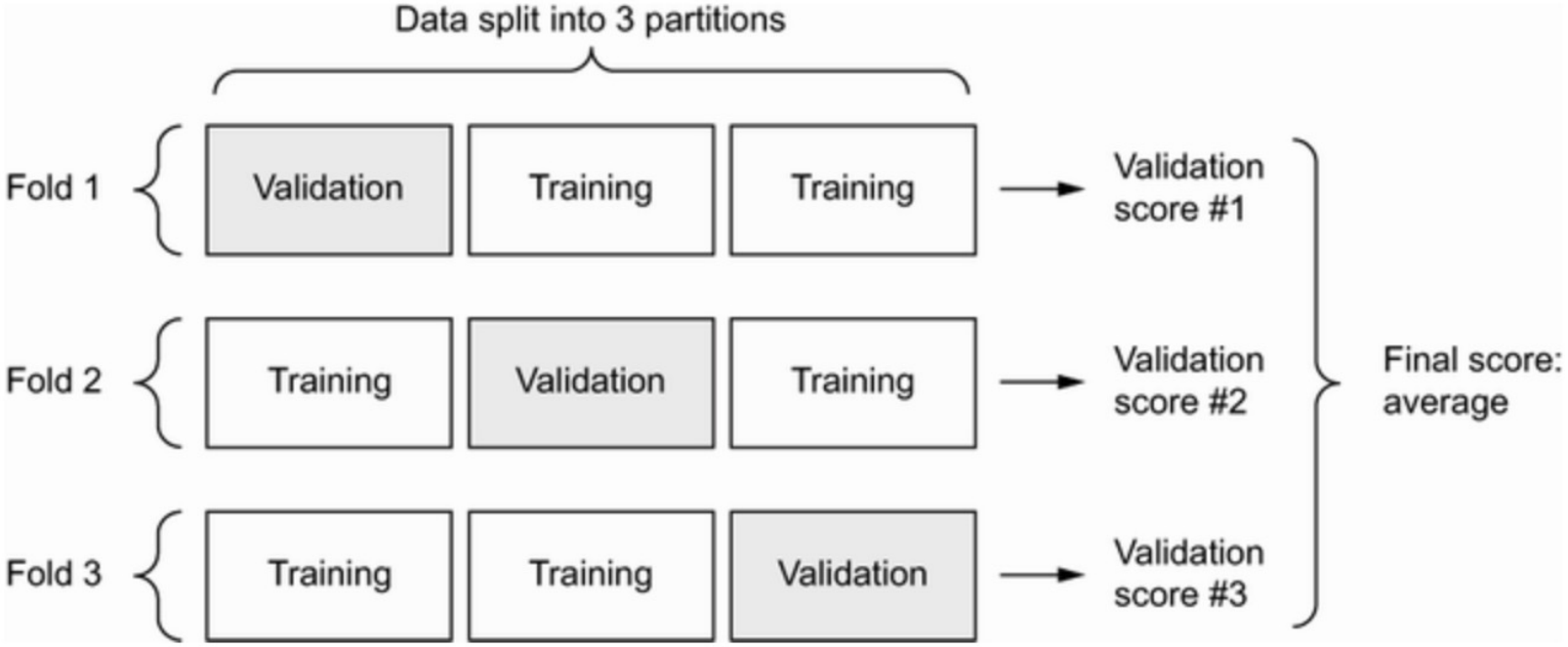
More about rmse, mse and mae Metrics

Validating the approach

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

A Scalar Regression: Predicting House Prices

Validating the approach using K-fold validation



K-fold cross-validation with K=3

A Scalar Regression: Predicting House Prices

Validating the approach using 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]
❶
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_sam-
ples]
       partial_train_data  =  np.concatenate(
❷
        [train_data[:i * num_val_samples],
         train_data[(i + 1) * num_val_samples:]],
        axis=0)
```

❶ Prepares the validation data: data from partition #k

❷ Prepares the training data: data from all other partitions

A Scalar Regression: Predicting House Prices

Validating the approach using K-fold validation

```
    partial_train_targets = np.concatenate(
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]],
        axis=0)
        model  =  build_model()
❸
        model.fit(partial_train_data,  partial_train_targets,
❹
                  epochs=num_epochs, batch_size=16, verbose=0)
      val_mse,  val_mae  =  model.evaluate(val_data,  val_targets,  verbose=0)
```

```
❺
    all_scores.append(val_mae)
```

❸ Builds the Keras model (already compiled)

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

❺ Evaluates the model on the validation data

A Scalar Regression: Predicting House Prices

Running with num_epochs = 100 yields the following results:

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