Coding Neural Networks: Classification and Regression

By

Dr. M. Shahidur Rahman

Professor, DoCSE, SUST

Outline

- Handling classification problems over vector data
  - Classifying movie reviews as positive or negative (binary classification)
  - Classifying news wires by topic (multiclass classification)
- Handling continuous regression problems over vector data
  - Estimating the price of a house, given real-estate data (scalar regression)

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.
- Ground-truth or annotations—All targets for a dataset, typically collected by humans.

Classification and regression glossary…

- 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.
- Scalar regression—A task where the target is a continuous scalar value. Predicting house prices is a good example.
- Vector regression—A task where the target is a set of continuous values: for example, doing regression against multiple values (such as 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.

Classifying movie reviews: Binary classification

IMDB Data Set

- 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.
- The IMDB dataset comes packaged with Keras. It has already been preprocessed: the reviews (sequences of words) have been turned into sequences of integers.
- 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.

Loading IMDB Data Set

```
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.
- 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.

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

Preparing the Data

- You can’t directly feed lists of integers into a neural network. They all have different lengths, but a neural network expects to process contiguous batches of data.
- You have to turn your lists into tensors. There are two ways to do that:
    - Pad your lists so that they all have the same length, turn them into an integer tensor of shape (samples, max_length)
    - 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.

Encoding the 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.])
```

Encoding the labels

- You should also vectorize your labels:

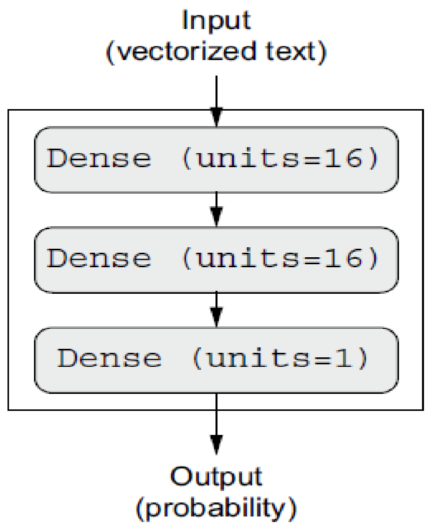
```
y_train = np.asarray(train_labels).astype("float32")
y_test = np.asarray(test_labels).astype("float32")
```

Building the Model

- The input data is vectors, and the labels are scalars (1s and 0s): this is one of the simplest problem setups.
- Model that performs well on such a problem is a plain stack of densely connected (Dense) layers with relu activations.
- There are two key architecture decisions to be made about such a stack of Dense layers:
  - How many layers to use
  - How many units to choose for each layer

Building the Model...

A tentative model for the task in hand:



The three-layer 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")
])
```

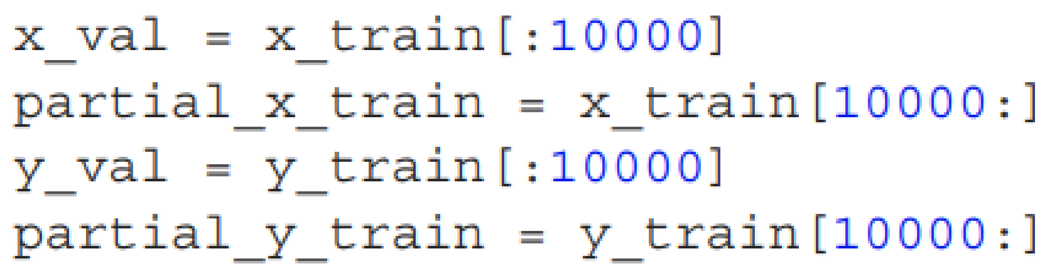
Compiling the Model

- For binary classification problem the output of the model is a probability (you end your model with a single-unit layer with a sigmoid activation), it’s best to use the binary_crossentropy loss.
- But crossentropy is usually the best choice when you’re dealing with models that output probabilities.

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

Validating the Approach

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:]
```

Training the Model

- The call to model.fit() returns a History object, which has a member history, containing a dictionary of data about everything that happened during training.
- We train the model for 20 epochs (20 iterations over all samples in the training data) in mini-batches of 512 samples.

```
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"]
```

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(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()
```

"bo" is for "blue dot."

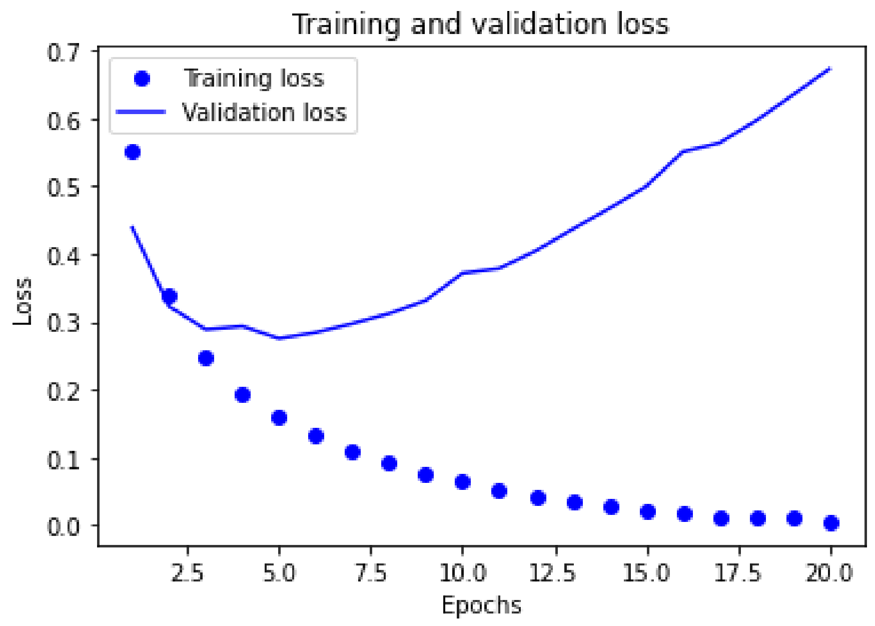
"b" is for "solid blue line."

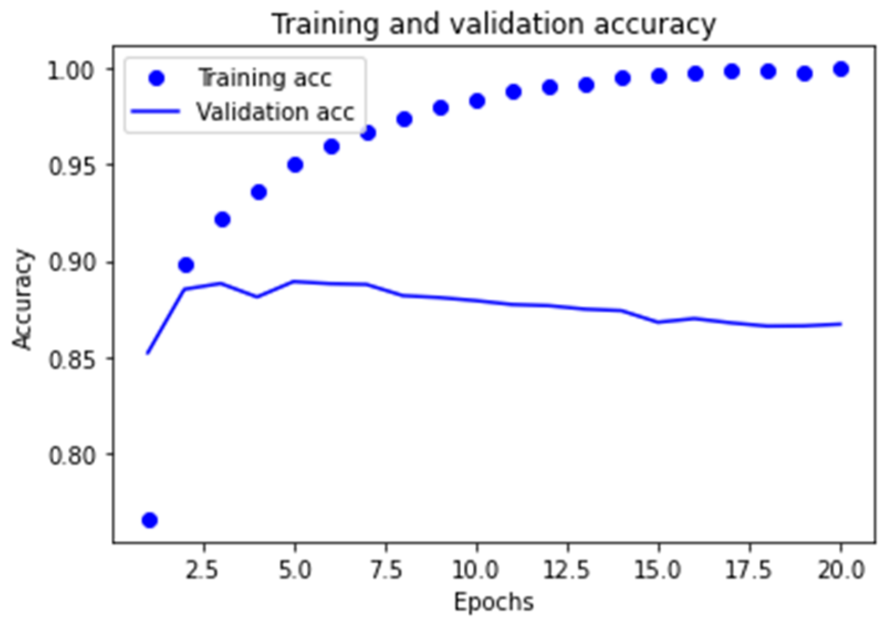
Plotting the training and validation accuracy

```
plt.clf()
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()
```

Clears the figure

Plotting the training and validation loss & accuracy





- The validation loss and accuracy seem to peak at the fourth epoch.
- A model that performs better on the training data isn't necessarily a model that will do better on data it has never seen before.
- Let's train a new model for four epochs and then evaluate it on the test data.

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:

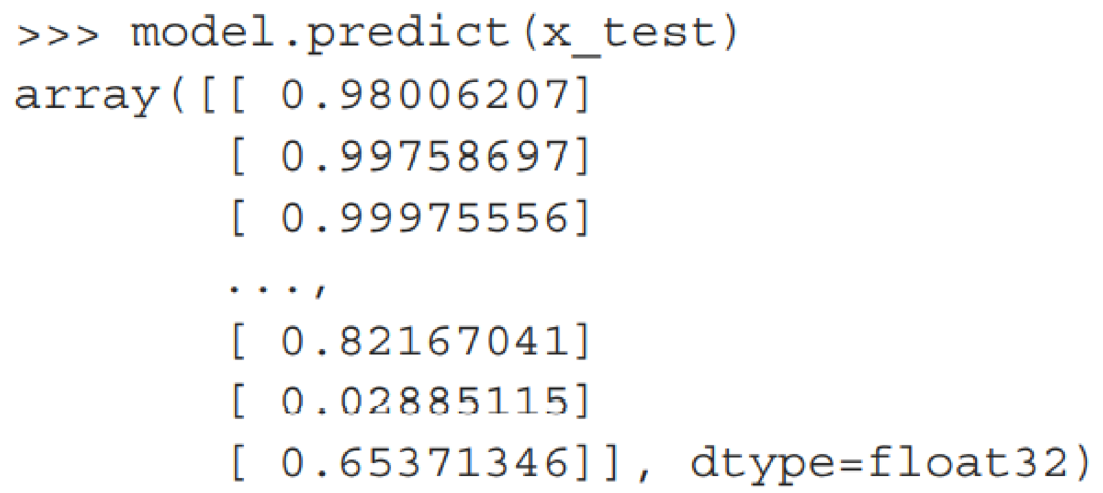
```
>>> results
[0.2929924130630493, 0.88327999999999995]
```

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

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

Predictions on new data

- Using a trained model to generate predictions on new data



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

- These are the the likelihood of reviews being positive by using the **predict** method. As you can see, 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).

Further Experiments

- Try using one or three representation layers, and see how doing so affects validation and test accuracy.
- Try using layers with more units or fewer units: 32 units, 64 units, and so on.
- Try using the MSE loss function instead of binary_crossentropy.
- Try using the tanh activation (an activation that was popular in the early days of neural networks) instead of relu.

Classifying newswires: A multiclass classification example

The Reuters dataset

- Reuters dataset, a set of short newswires and their topics, published by Reuters in 1986.
- It’s a simple 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.

Loading the Reuters dataset

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

- You have 8,982 training examples and 2,246 test examples.
- 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]
```

Preparing the data

- Encoding the input data

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

Vectorized training data

Vectorized test data

- To vectorize the labels, there are two possibilities:
  - you can cast the label list as an integer tensor,
  - you can use one-hot encoding. One-hot encoding is a widely used format for categorical data.
  - 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.
- 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)
```

Building your model

- The number of output classes is now 46. The dimensionality of the output space is much larger.
- In a stack of Dense layers, each layer can only access information present in the output of the previous layer. If one layer drops some information relevant to the classification problem, this information can never be recovered by later layers.
- As in the previous example, a 16-dimensional space may be too limited to learn to separate 46 different classes: such small layers permanently drop relevant information.
- For this reason we'll use larger layers with 64 units.

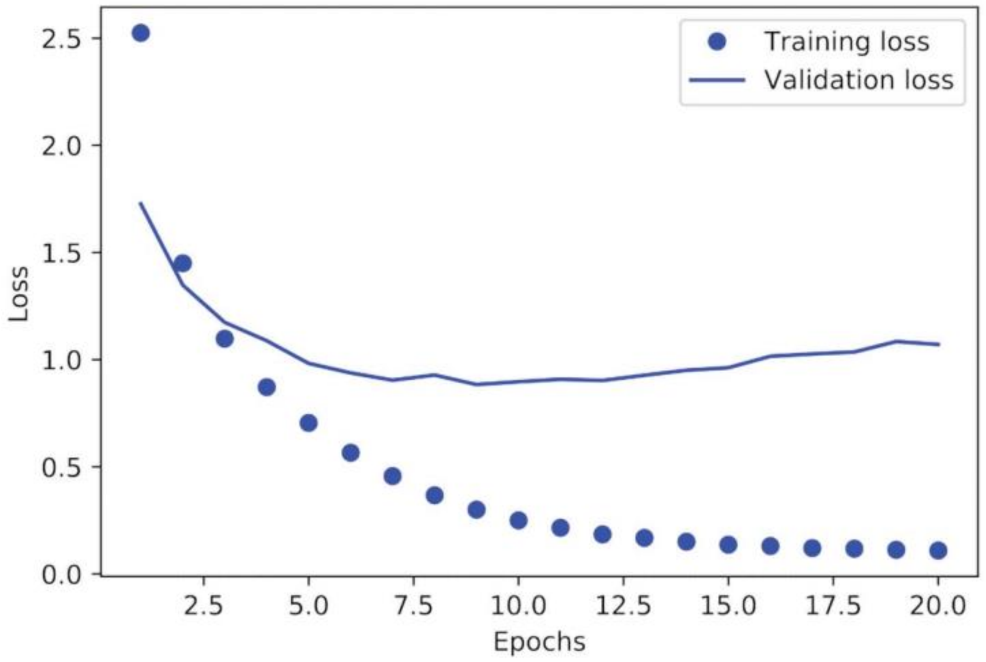
Model definition

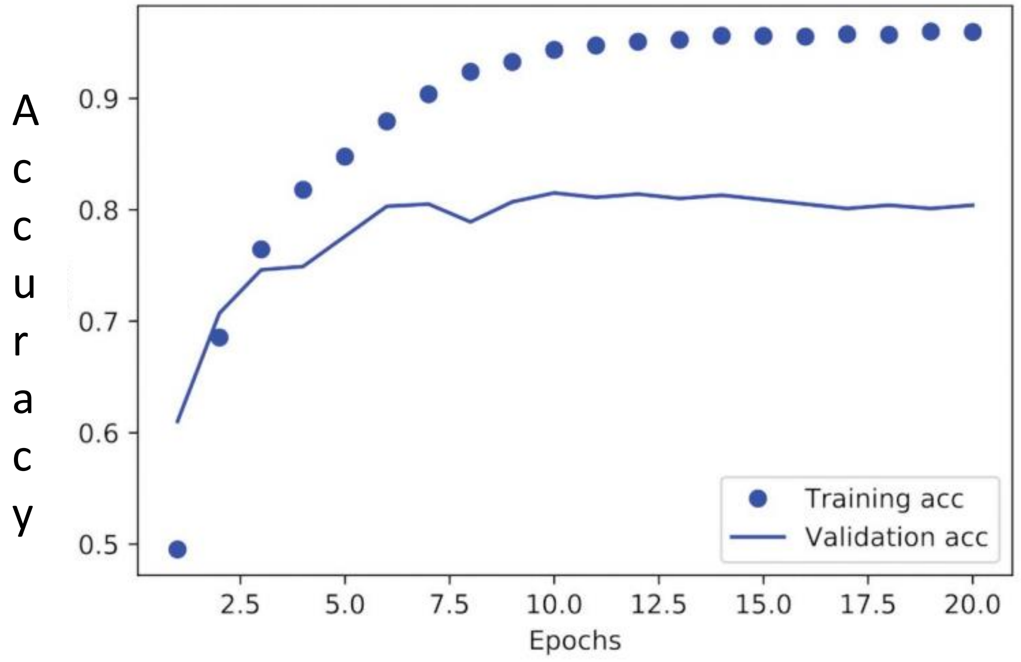
```
model = keras.Sequential([
    layers.Dense(64, activation="relu"),
    layers.Dense(64, activation="relu"),
    layers.Dense(46, activation="softmax")
])
```

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

- The model will output a probability distribution over the 46 different output, where output[i] is the probability that the sample belongs to class i. The 46 scores will sum to 1.
- The last layer uses a softmax activation for multiclass classification.
- The best loss function to use in this case is categorical_crossentropy. It measures the distance between the probability distribution output by the model and the true distribution of the labels.

The training and validation loss & accuracy





- You can now fit and evaluate the model on validation set as before.
- The model begins to overfit after nine epochs.
- Let’s train a new model from scratch for nine epochs for evaluation on test set.

Wrapping up

- If you’re trying to classify data points among N classes, your model should end with a Dense layer of size N.
- In a single-label, multiclass classification problem, your model should end with a softmax activation so that it will output a probability distribution over the N output classes.
- Categorical crossentropy is almost always the loss function you should use for such problems.
- There are two ways to handle labels in multiclass classification:
  - Encoding the labels via categorical encoding (also known as one-hot encoding) and using categorical_crossentropy as a loss function
  - Encoding the labels as integers and using the sparse_categorical_crossentropy loss function
- If you need to classify data into a large number of categories, you should avoid creating information bottlenecks due to intermediate layers that are too small.

Predicting house prices: A regression example

- Predicting a continuous value instead of a discrete label

The Boston housing price dataset

- The dataset contains the median price of homes in a given Boston suburb in the mid-1970s, given data points about the suburb at the time, such as the crime rate, the local property tax rate, and so on.
- The dataset has relatively few data points: only 506, split between 404 training samples and 102 test samples.
- Each feature in the input data has a different scale.
  - some values are proportions, which take values between 0 and 1, others take values between 1 and 12, others between 0 and 100, and so on.

Loading the Boston housing dataset

```
from 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
- It is problematic to feed values that all take wildly different ranges.
- A best practice for dealing with such data is to do 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.

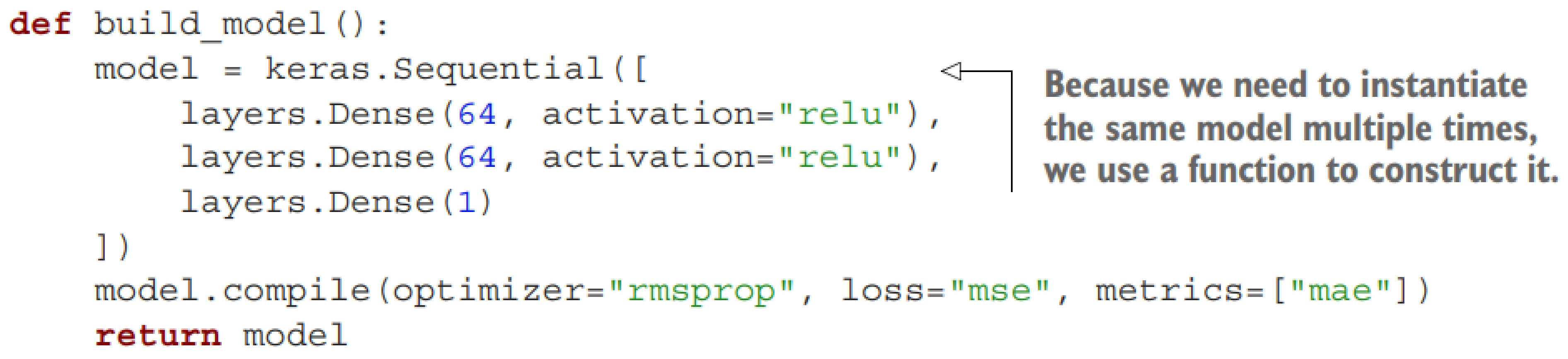
Normalizing the data

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

- Note that the quantities used for normalizing the test data are computed using the training data.
- You should never use any quantity computed on the test data in your workflow, even for something as simple as data normalization.

Building your model

- Because so few samples are available, we'll use a very small model with two intermediate layers, each with 64 units.
- In general, the less training data you have, the worse overfitting will be, and using a small model is one way to mitigate overfitting.



```
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
```

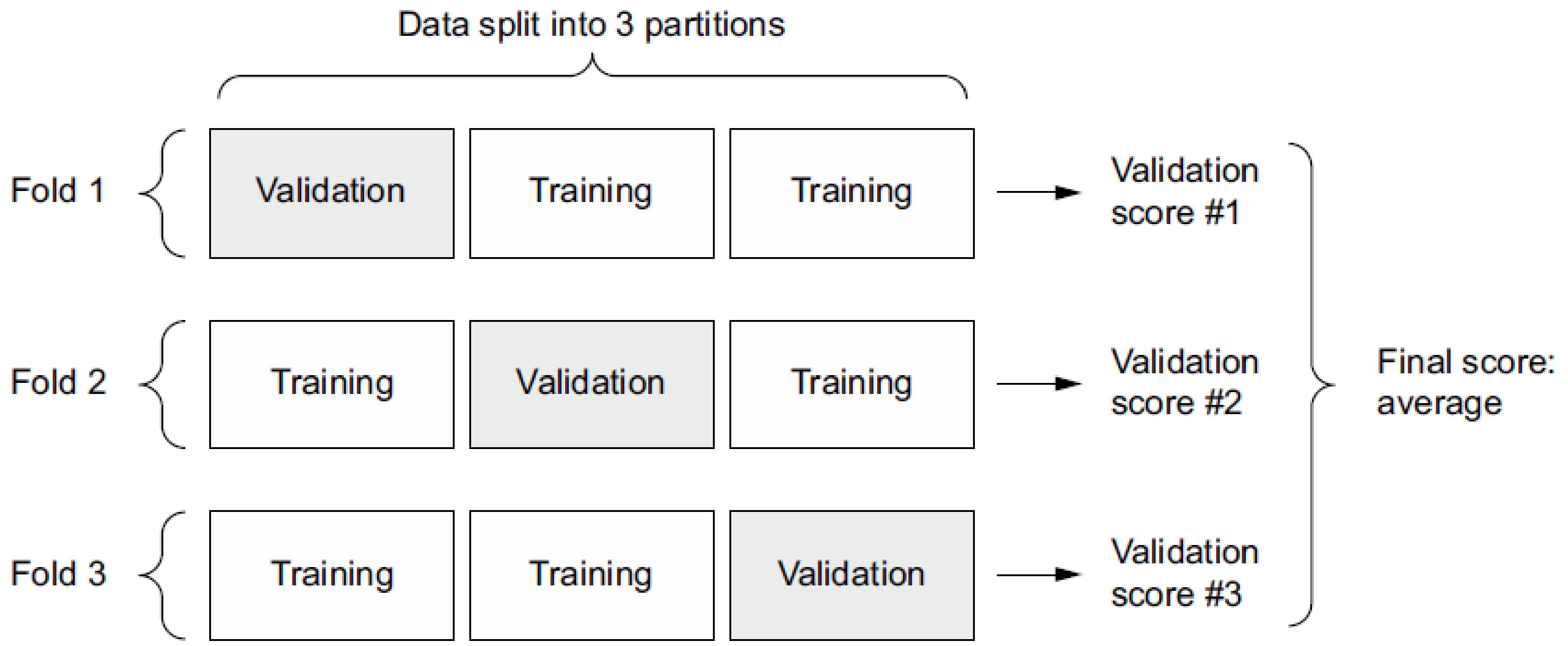
Because we need to instantiate the same model multiple times, we use a function to construct it.

- The model ends with a single unit and no activation. This is a typical setup for scalar regression.
  - The last layer is purely linear, the model is free to learn to predict values in any range.
  - Also, MSE is a widely used loss function for regression problems

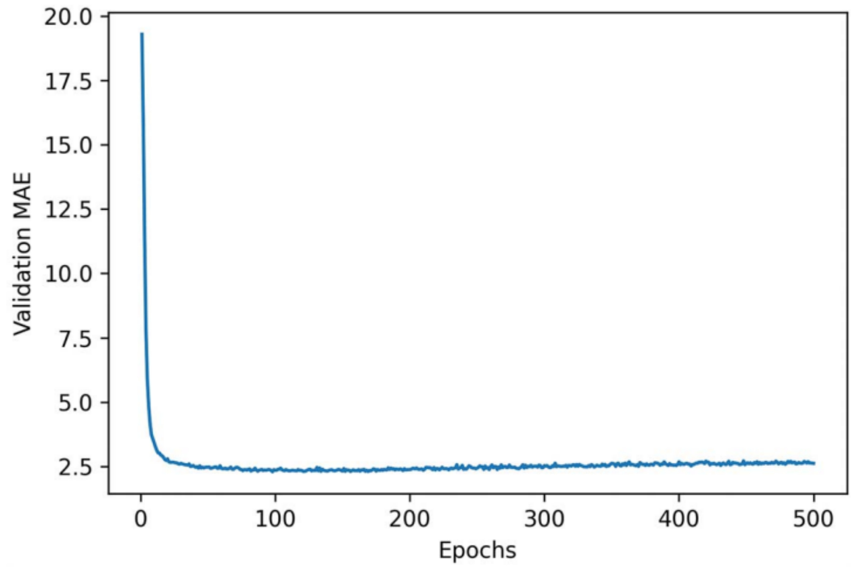
Validating your approach using K-fold validation

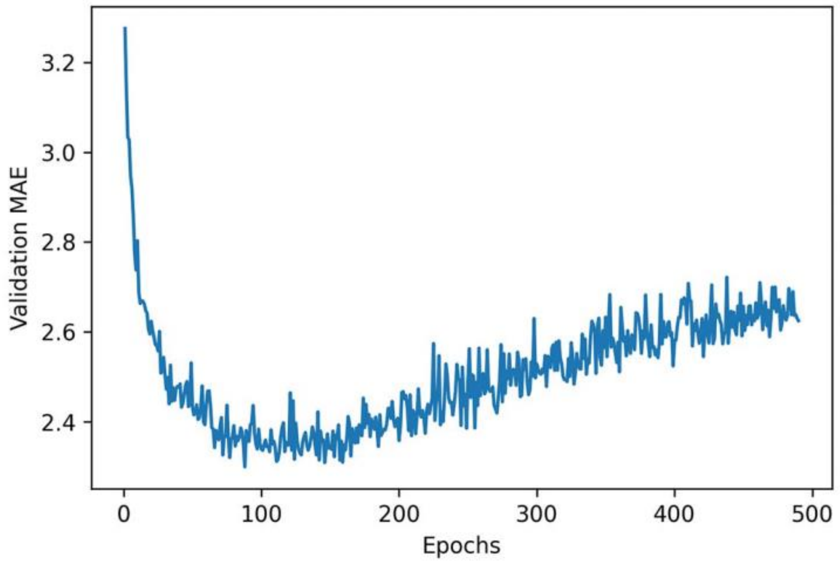
- To evaluate our model while we keep adjusting its parameters (such as the number of epochs used for training), we could split the data into a training set and a validation set.
- But because we have so few data points, the validation set would end up being very small. So, the validation scores might change a lot depending on which data points we chose for validation and which we chose for training.
- This would prevent us from reliably evaluating our model. The best practice in such situations is to use K-fold cross-validation

K-fold cross-validation with K=3



Validation MA E by epoch, including & excluding the first 10 data points





- The validation MAE for the first few epochs is dramatically higher than the values that follow. Let's omit the first 10 data points, which are on a different scale than the rest of the curve.
- Validation MAE stops improving significantly after 120–140 epochs (including the 10 epochs we omitted). Past that point, we start overfitting.

Training the final model & Predicting from new data

Gets a fresh, compiled model

Trains it on the entirety of the data

```
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)
```

```
>>> test_mae_score
2.4642276763916016
```

Predictions from New Data

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

- The first house in the test set is predicted to have a price of about $10,000.

Wrapping up

- Regression is done using different loss functions than 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 model with few intermediate layers (typically only one or two), in order to avoid severe overfitting.