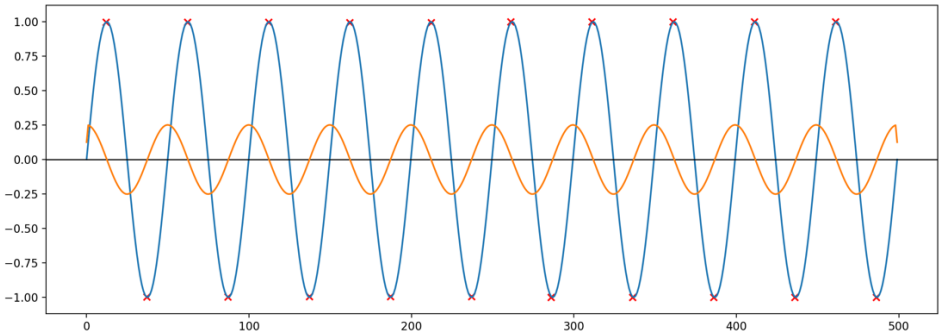
SHS solutions 11: Speech classification with deep networks

Solution 11.1. Convolution.

```
import numpy as np
import matplotlib.pyplot as plt

def convolve(signal, convolution):
     # Flipping the convolution mask
     convolution = convolution[::-1]

     # Half of the convolution window
     pad = int(np.floor(len(convolution) / 2))

     # Signal each point is an anchor, so we extend 2*int(pad)
     padded_signal = np.zeros(len(signal) + 2 * int(pad) + 1)
     padded_signal[1:-2] = signal

     padded_signal= np.pad(signal,(pad,pad + 1), mode="wrap")
     # Generating indices for the window
     window = np.arange(-pad, pad + 1, 1)

     return [np.sum(padded_signal[window+i]*convolution)
               for i in range(0, len(padded_signal) - 1)]

signal = np.sin(np.linspace(0, 2*np.pi*10, 500))
plt.figure(figsize= (15,5))

def peak_detection(signal):
     plt.plot(signal)
     result = convolve(signal, [1, 0, -1])
     plt.axhline(linewidth=1, color='black')
     plt.plot(range(len(signal)), result[1:-1])
     zero_crossings = np.where(np.diff(np.sign(result)))[0]
     plt.scatter(zero_crossings[1:] - 1, signal[zero_crossings[1:]],
               marker='x',color='red')
     plt.show()
     return zero_crossings

peak_detection(signal)
```

The output plot is:



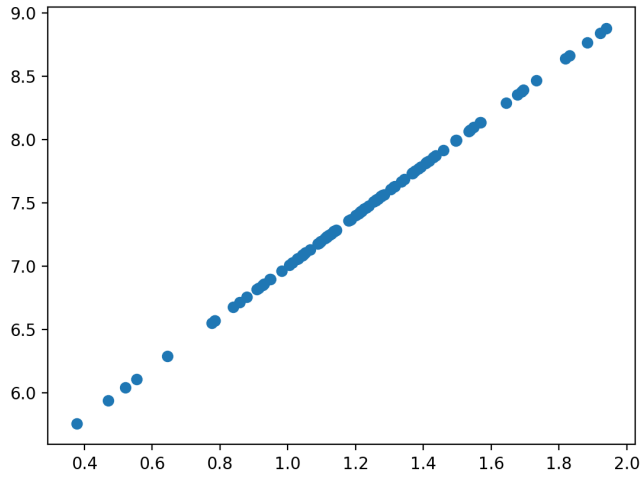
Solution 11.2. Linear regression.

When generating (x, y) data, we keep the x data and the y data together in a dictionary with two entries (namely x and y). We generate a dataset three times, and we put the three datasets together in another dictionary, so that we don't lose anything.

```
import numpy as np
import matplotlib.pyplot as plt

def generate_data(sigma_error):
     data = dict()
     number_of_data_points = 100
     mu_x = 1.2
     sigma_x = 0.3
     data['x'] = np.random.normal(mu_x, sigma_x, number_of_data_points)
     mu_error = 0.0
     data['y'] = 2.0 * data['x'] + 5.0 + np.random.normal(
               mu_error, sigma_error, number_of_data_points)
     return data

data = dict()
data[1] = generate_data(0.001)
data[2] = generate_data(0.1)
data[3] = generate_data(1.0)
```

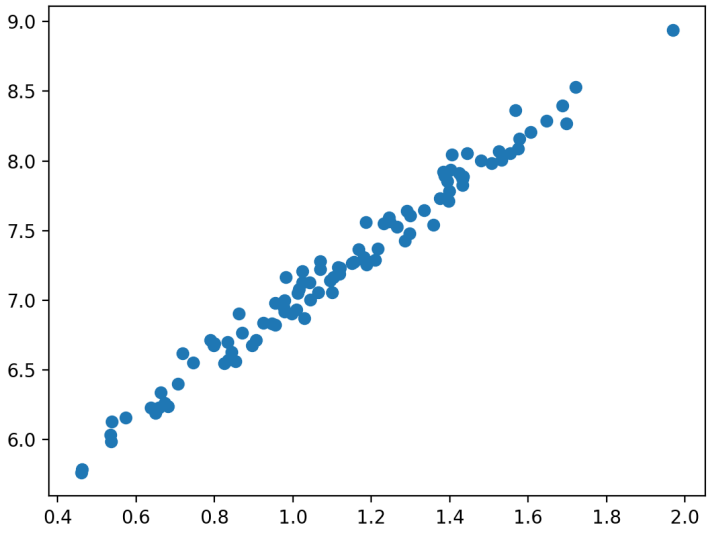
The plot for the low-σ data is almost straight:

```
plt.scatter(data[1]['x'], data[1]['y'])
plt.show()
```



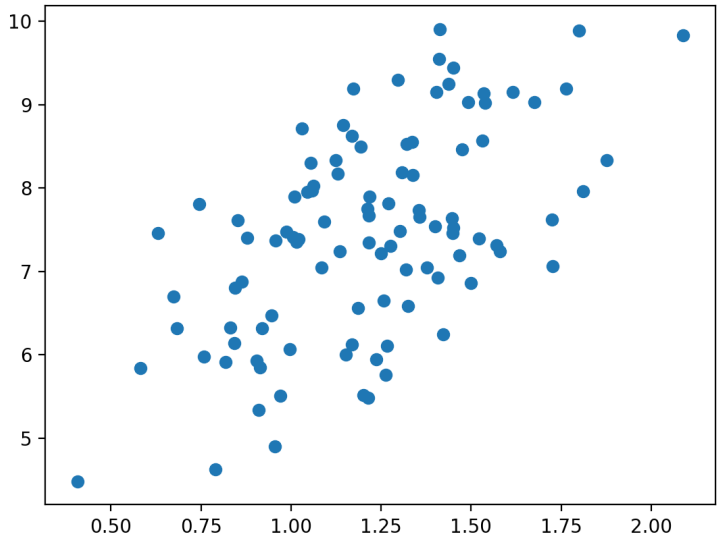
The plot for the mid-σ data shows more variation around the line $y = 2x + 5$:

```
plt.scatter(data[2]['x'], data[2]['y'])
plt.show()
```



The plot for the high-σ data shows large variation around the line:

```
plt.scatter(data[3]['x'], data[3]['y'])
plt.show()
```



In general, a fitted line has the formula $y = ax + b$. We define a linear regression function that takes an equal number of *x* and *y* values, and returns the two fitted (i.e. estimated) parameters *a* and *b*:

```
def linear_regression(x_observed, y_observed):
     assert len(x_observed) == len(y_observed)
     a = np.random.normal(0.0, 1.0)
     b = np.random.normal(0.0, 1.0)
     learning_rate = 0.01
     number_of_epochs = 100000
     for idatum in range(number_of_epochs):
          y_predicted = a * x_observed + b
          loss = np.mean((y_observed - y_predicted)**2)   # not used
          dloss_da = -2 * np.mean(x_observed * (y_observed - y_predicted))
          dloss_db = -2 * np.mean(y_observed - y_predicted)
          a -= learning_rate * dloss_da
          b -= learning_rate * dloss_db
     return a, b

print('Fitted parameters:')
print(linear_regression(data[1]['x'], data[1]['y']))
print(linear_regression(data[2]['x'], data[2]['y']))
print(linear_regression(data[3]['x'], data[3]['y']))

## Fitted parameters:
## (2.000090967400371, 4.999851772215268)
## (2.0530241344424316, 4.938034643483894)
## (2.2808150615212375, 4.672172135694904)
```

The procedure has succeeded: the estimated *a* and *b* are close to the underlying true *a* and *b* that generated the data. The third dataset yields estimates that are more removed from these true *a* and *b*; this is not because the linear regression failed in some way, but because the data themselves are more variable. If you want to know about the details behind this (e.g. how to compute the *confidence intervals* around the estimated values), you can take a statistics course.

In the above, each data point $x_{observed}$ can be regarded as the input of a "network" with one input node and one output node, which is truly the simplest fully connected ("dense") network that once can imagine. This network has at any point in time (during learning and after) the parameters *a* and *b*. The output of the network is therefore always given by

$$y_{predicted} = ax_{observed} + b$$

This prediction makes an error ("loss") of

$$\mathcal{L} = \left(y_{predicted} - y_{observed}\right)^2$$

which in terms of *a* and *b* is

$$\mathcal{L} = (ax_{observed} + b - y_{observed})^2$$

If we have a batch of observations (x_i, y_i), where *i* runs from 1 to *N*, then the mean loss over those *N* observations is

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(ax_i + b - y_i)^2$$

This formula appears in the "unused" expression for *loss* in the code above. Note that `x_observed`, `y_predicted` and `y_observed` in the code are arrays of length *N*.

To be able to learn *a* and *b*, we need the gradient of the loss with respect to *a* and *b*:

$$\frac{\partial \mathcal{L}}{\partial a} = \frac{2}{N}\sum_{i=1}^{N}(ax_i + b - y_i)x_i$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{2}{N}\sum_{i=1}^{N}(ax_i + b - y_i)$$

These formulas are in the code as well (the minus sign in the code comes from having the predicted and observed values in opposite order).

We can now update *a* and *b* along these gradients:

$$a_{new} = a_{old} - \eta\frac{\partial \mathcal{L}}{\partial a}$$

$$b_{new} = b_{old} - \eta\frac{\partial \mathcal{L}}{\partial b}$$

where η is a small learning rate. These changes in *a* and *b* usually take the values of $y_{predicted}$, when these are newly computed from a_{new} and b_{new}, closer to $y_{observed}$, on average. That is what we want from a learning algorithm!

Solution 11.3. Two-class classification in Keras

The following code actually runs.

```
from tensorflow import keras
import numpy as np

NUMBER_OF_DATA_POINTS = 100   # number of training examples
NUMBER_OF_INPUT_FEATURES = 10   # e.g. number of MFCC coefficients
NUMBER_OF_CLASSES = 2
NUMBER_OF_HIDDEN1 = 200
NUMBER_OF_HIDDEN2 = 50

input_data = np.zeros(shape=(NUMBER_OF_DATA_POINTS, NUMBER_OF_INPUT_FEATURES))
output_data = np.zeros(shape=(NUMBER_OF_DATA_POINTS, NUMBER_OF_CLASSES))

# Create a model with two hidden layers.
input_layer = keras.Input(shape=(NUMBER_OF_INPUT_FEATURES,))
hidden_layer1 = keras.layers.Dense(NUMBER_OF_HIDDEN1,
          activation="sigmoid")(input_layer)
hidden_layer2 = keras.layers.Dense(NUMBER_OF_HIDDEN2,
          activation="sigmoid")(hidden_layer1)
output_layer = keras.layers.Dense(NUMBER_OF_CLASSES,
          activation="softmax")(hidden_layer2)
model = keras.models.Model(input_layer,output_layer)

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

model.fit(input_data, output_data, epochs=10, batch_size=32)
```

The five lines that create the model show that the model is sequential: `keras.Input()` declares that there is an input layer consisting of `number_of_input_features` nodes; the first `keras.layers.Dense()()` declares that the input layer is followed by a layer of `number_of_hidden1` nodes that are fully (densely) connected to the input layer (i.e., each of the `number_of_hidden1` nodes on this hidden layer is connected to each of the `number_of_input_features` nodes of the input layer); the second `keras.layers.Dense()()` declares that the first hidden layer is followed by a layer of `number_of_hidden2` nodes that are fully connected to the first hidden layer; the third `keras.layers.Dense()()` declares that the second hidden layer is followed by a softmax layer. The call to `keras.models.Model()` declares that this last level is the output layer, and that the first level we just mentioned is the input layer. Thus, the four levels are in sequential order, so that we could have combined them in one statement:

```
model = keras.Sequential([
     keras.layers.Dense(NUMBER_OF_HIDDEN1, activation="sigmoid",
               input_shape=(NUMBER_OF_INPUT_FEATURES,)),
     keras.layers.Dense(NUMBER_OF_HIDDEN2, activation="sigmoid"),
     keras.layers.Dense(NUMBER_OF_CLASSES, activation="softmax")
])
```

Softmax. In the above example, the output of the network is a layer of `number_of_classes` nodes: the node that is activated most strongly on a give input is considered the winner. In

our example of two classes, the network is said to have recognized the input as belonging to class 1 if the input causes the first output node to be more strongly activated than the second output node, and the network is said to have recognized class 2 if the second output node is more strongly activated than the first. The "output data" for the model has to contain the "correct" class (label) in one-hot encoding: if the correct class is class 1, the current row of the output data has to have a 1 in the first column (column 0) and a 0 in the second column (column 1), while if the correct class is class 2, the current row of the output data has to have a 0 in the first column and a 1 in the second.

One binary output node. In the special case of two classes, we can also do just one node on the last level. The activation function at this last level is then just sigmoid, so that values lower than 0.5 can be regarded as meaning one class, and values higher than 0.5 can be regarded as meaning the other class.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import numpy as np

NUMBER_OF_DATA_POINTS = 100   # number of training examples
NUMBER_OF_INPUT_FEATURES = 10   # e.g. number of MFCC coefficients
NUMBER_OF_HIDDEN1 = 30
NUMBER_OF_HIDDEN2 = 20

input_data = np.zeros(shape=(NUMBER_OF_DATA_POINTS, NUMBER_OF_INPUT_FEATURES))
output_data = np.zeros(shape=(NUMBER_OF_DATA_POINTS,))

# Create a model with two hidden layers.
model = Sequential([
     Dense(NUMBER_OF_HIDDEN1, activation="sigmoid",
               input_shape=(NUMBER_OF_INPUT_FEATURES,)),
     Dense(NUMBER_OF_HIDDEN2, activation="sigmoid"),
     Dense(1, activation="sigmoid")
])

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

model.fit(input_data, output_data, epochs=10, batch_size=32)
```

Note that the loss function is now "binary cross-entropy" instead of "categorical cross-entropy". Also note that the shape of the output data has changed: the output data should just be a series of ones and zeroes, depending on whether the correct class is class 1 or class 2 (the construct `(NUMBER_OF_DATA_POINTS,)` with its strange comma is the Python way to indicate a tuple with one element, and the `shape` argument has to be a tuple).