solution06.2

November 10, 2020

Exercise Sheet 6.2 **Long short-term memory (LSTM)**

```
[ ]: import numpy as np
     from keras import Sequential
     from keras.callbacks import Callback
     from keras import layers
     from keras import optimizers
     import matplotlib.pyplot as plt
```

```
[2]: class LogHistory(Callback):
         def on_train_begin(self, logs={}):
             self.acc = []
             self.loss = []

         def on_batch_end(self, batch, logs={}):
             self.acc.append(logs.get('acc'))
             self.loss.append(logs.get('loss'))
```

```
[3]: def input_maker():
         # input data and labels setup as a natural selection
         input_data = np.random.randint(low=0, high=10, size=(10000, 1, 30))
         true_labels = np.array([np.sum(input_data, axis=2) >= 100]).astype(int)[0, :
      ↪, 0].T

         # slicing for training and validation
         training_input = input_data[0:8000, :, :]
         validation_input = input_data[8000:, :, :]
         training_labels = true_labels[0:8000]
         validation_labels = true_labels[8000:]

         # checking the ratios
         i_number_of_ones = np.count_nonzero(true_labels)
         print('Number of series which add up to 100 or more = %d' %␣
      ↪i_number_of_ones)
         i_number_of_zeros = len(true_labels) - i_number_of_ones
         print('Number of series which add up to less than 100 = %d' %␣
      ↪i_number_of_zeros)
         i_ratio = i_number_of_ones / i_number_of_zeros
```

```
    print('the ratio of ones to zeroes in the labels = %d' % int(i_ratio))

    return training_input, validation_input, training_labels, validation_labels
```

[4]:
```
def input_maker_rnd():
    # a bad implementation of random selection of series
    input_data = np.zeros((10000, 1, 30))
    i = 0
    while i < 5000:
        temp_array = np.random.randint(low=0, high=10, size=(1, 1, 30))
        if np.sum(temp_array) >= 100:
            input_data[i,0,:] = temp_array
            i+=1
    while i < 10000:
        temp_array = np.random.randint(low=0, high=10, size=(1, 1, 30))
        if np.sum(temp_array) < 100:
            input_data[i,0,:] = temp_array
            i+=1
    np.random.shuffle(input_data)

    # true labels
    true_labels = np.array([np.sum(input_data, axis=2) >= 100]).astype(int)[0,:
 ↪,0].T

    # checking the ratios
    i_number_of_ones = np.count_nonzero(true_labels)
    print('Number of series which add up to 100 or more = %d' %␣
 ↪i_number_of_ones)
    i_number_of_zeros = len(true_labels) - i_number_of_ones
    print('Number of series which add up to less than 100 = %d' %␣
 ↪i_number_of_zeros)
    i_ratio = i_number_of_ones / i_number_of_zeros
    print('the ratio of ones to zeroes in the labels = %d' % int(i_ratio))

    # slicing for training and validation
    training_input = input_data[0:8000, :, :]
    validation_input = input_data[8000:, :, :]
    training_labels = true_labels[0:8000]
    validation_labels = true_labels[8000:]

    return training_input, validation_input, training_labels, validation_labels
```

[5]:
```
# creating the random inputs
training_input, validation_input, training_labels, validation_labels =␣
 ↪input_maker()
```

```
Number of series which add up to 100 or more = 9874
Number of series which add up to less than 100 = 126
```

```
the ratio of ones to zeroes in the labels = 78
```

[6]:
```
# defining the model
model = Sequential()

model.add(layers.LSTM(units=200,
                      activation='tanh',
                      recurrent_activation='hard_sigmoid',
                      use_bias=True,
                      unit_forget_bias=True))

model.add(layers.Dropout(0.5))

model.add(layers.Dense(1, activation='sigmoid'))
```

[7]:
```
# call back function
log_history = LogHistory()

# the adam optimizer
adam = optimizers.Adam(lr=0.001,
                       beta_1=0.9,
                       beta_2=0.999,
                       epsilon=1e-8)

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

training_log = model.fit(training_input,
                         training_labels,
                         batch_size=50,
                         epochs=60,
                         shuffle=True,
                         validation_data=(validation_input, validation_labels),
                         callbacks=[log_history])
```

```
Train on 8000 samples, validate on 2000 samples
Epoch 1/60
8000/8000 [==============================] - 4s 510us/step - loss: 0.0651 -
binary_accuracy: 0.9868 - val_loss: 0.0573 - val_binary_accuracy: 0.9860
Epoch 2/60
8000/8000 [==============================] - 2s 222us/step - loss: 0.0488 -
binary_accuracy: 0.9878 - val_loss: 0.0584 - val_binary_accuracy: 0.9860
Epoch 3/60
8000/8000 [==============================] - 2s 245us/step - loss: 0.0446 -
binary_accuracy: 0.9878 - val_loss: 0.0515 - val_binary_accuracy: 0.9860
Epoch 4/60
8000/8000 [==============================] - 2s 275us/step - loss: 0.0402 -
```

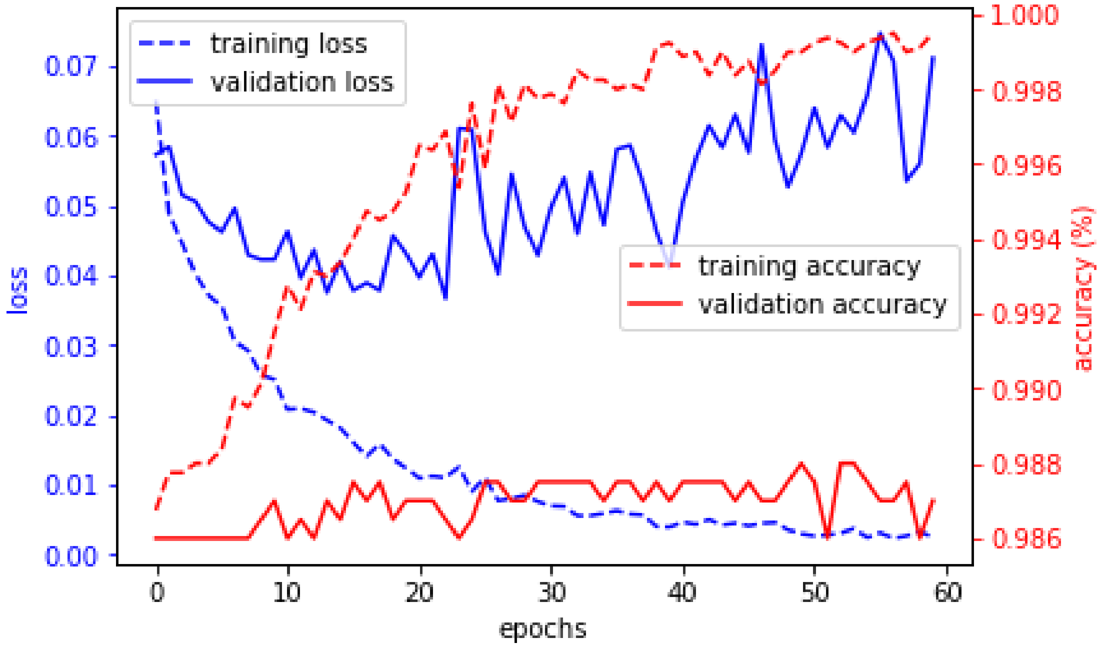
```
binary_accuracy: 0.9880 - val_loss: 0.0506 - val_binary_accuracy: 0.9860
Epoch 5/60
8000/8000 [==============================] - 2s 229us/step - loss: 0.0372 -
binary_accuracy: 0.9880 - val_loss: 0.0477 - val_binary_accuracy: 0.9860
Epoch 6/60
8000/8000 [==============================] - 2s 290us/step - loss: 0.0356 -
binary_accuracy: 0.9884 - val_loss: 0.0463 - val_binary_accuracy: 0.9860
Epoch 7/60
8000/8000 [==============================] - 2s 308us/step - loss: 0.0305 -
binary_accuracy: 0.9898 - val_loss: 0.0497 - val_binary_accuracy: 0.9860
Epoch 8/60
8000/8000 [==============================] - 2s 236us/step - loss: 0.0292 -
binary_accuracy: 0.9895 - val_loss: 0.0430 - val_binary_accuracy: 0.9860
Epoch 9/60
8000/8000 [==============================] - 2s 226us/step - loss: 0.0257 -
binary_accuracy: 0.9901 - val_loss: 0.0423 - val_binary_accuracy: 0.9865
Epoch 10/60
8000/8000 [==============================] - 2s 228us/step - loss: 0.0251 -
binary_accuracy: 0.9915 - val_loss: 0.0423 - val_binary_accuracy: 0.9870
Epoch 11/60
8000/8000 [==============================] - 2s 228us/step - loss: 0.0208 -
binary_accuracy: 0.9928 - val_loss: 0.0463 - val_binary_accuracy: 0.9860
Epoch 12/60
8000/8000 [==============================] - 2s 230us/step - loss: 0.0209 -
binary_accuracy: 0.9921 - val_loss: 0.0397 - val_binary_accuracy: 0.9865
Epoch 13/60
8000/8000 [==============================] - 2s 220us/step - loss: 0.0204 -
binary_accuracy: 0.9931 - val_loss: 0.0436 - val_binary_accuracy: 0.9860
Epoch 14/60
8000/8000 [==============================] - 2s 248us/step - loss: 0.0193 -
binary_accuracy: 0.9930 - val_loss: 0.0375 - val_binary_accuracy: 0.9870
Epoch 15/60
8000/8000 [==============================] - 3s 313us/step - loss: 0.0181 -
binary_accuracy: 0.9934 - val_loss: 0.0420 - val_binary_accuracy: 0.9865
Epoch 16/60
8000/8000 [==============================] - 3s 314us/step - loss: 0.0161 -
binary_accuracy: 0.9940 - val_loss: 0.0378 - val_binary_accuracy: 0.9875
Epoch 17/60
8000/8000 [==============================] - 3s 344us/step - loss: 0.0141 -
binary_accuracy: 0.9948 - val_loss: 0.0390 - val_binary_accuracy: 0.9870
Epoch 18/60
8000/8000 [==============================] - 3s 322us/step - loss: 0.0159 -
binary_accuracy: 0.9945 - val_loss: 0.0378 - val_binary_accuracy: 0.9875
Epoch 19/60
8000/8000 [==============================] - 2s 266us/step - loss: 0.0137 -
binary_accuracy: 0.9948 - val_loss: 0.0457 - val_binary_accuracy: 0.9865
Epoch 20/60
8000/8000 [==============================] - 2s 224us/step - loss: 0.0124 -
```

```
binary_accuracy: 0.9953 - val_loss: 0.0432 - val_binary_accuracy: 0.9870
Epoch 21/60
8000/8000 [==============================] - 2s 220us/step - loss: 0.0109 -
binary_accuracy: 0.9965 - val_loss: 0.0397 - val_binary_accuracy: 0.9870
Epoch 22/60
8000/8000 [==============================] - 2s 225us/step - loss: 0.0113 -
binary_accuracy: 0.9964 - val_loss: 0.0431 - val_binary_accuracy: 0.9870
Epoch 23/60
8000/8000 [==============================] - 2s 225us/step - loss: 0.0110 -
binary_accuracy: 0.9969 - val_loss: 0.0367 - val_binary_accuracy: 0.9865
Epoch 24/60
8000/8000 [==============================] - 2s 217us/step - loss: 0.0126 -
binary_accuracy: 0.9954 - val_loss: 0.0610 - val_binary_accuracy: 0.9860
Epoch 25/60
8000/8000 [==============================] - 2s 209us/step - loss: 0.0091 -
binary_accuracy: 0.9976 - val_loss: 0.0608 - val_binary_accuracy: 0.9865
Epoch 26/60
8000/8000 [==============================] - 2s 216us/step - loss: 0.0110 -
binary_accuracy: 0.9959 - val_loss: 0.0462 - val_binary_accuracy: 0.9875
Epoch 27/60
8000/8000 [==============================] - 2s 206us/step - loss: 0.0077 -
binary_accuracy: 0.9981 - val_loss: 0.0402 - val_binary_accuracy: 0.9875
Epoch 28/60
8000/8000 [==============================] - 2s 214us/step - loss: 0.0080 -
binary_accuracy: 0.9971 - val_loss: 0.0545 - val_binary_accuracy: 0.9870
Epoch 29/60
8000/8000 [==============================] - 2s 205us/step - loss: 0.0085 -
binary_accuracy: 0.9981 - val_loss: 0.0470 - val_binary_accuracy: 0.9870
Epoch 30/60
8000/8000 [==============================] - 2s 204us/step - loss: 0.0076 -
binary_accuracy: 0.9978 - val_loss: 0.0429 - val_binary_accuracy: 0.9875
Epoch 31/60
8000/8000 [==============================] - 2s 197us/step - loss: 0.0070 -
binary_accuracy: 0.9979 - val_loss: 0.0497 - val_binary_accuracy: 0.9875
Epoch 32/60
8000/8000 [==============================] - 2s 216us/step - loss: 0.0069 -
binary_accuracy: 0.9976 - val_loss: 0.0540 - val_binary_accuracy: 0.9875
Epoch 33/60
8000/8000 [==============================] - 2s 200us/step - loss: 0.0056 -
binary_accuracy: 0.9985 - val_loss: 0.0460 - val_binary_accuracy: 0.9875
Epoch 34/60
8000/8000 [==============================] - 2s 206us/step - loss: 0.0056 -
binary_accuracy: 0.9983 - val_loss: 0.0548 - val_binary_accuracy: 0.9875
Epoch 35/60
8000/8000 [==============================] - 2s 208us/step - loss: 0.0059 -
binary_accuracy: 0.9983 - val_loss: 0.0472 - val_binary_accuracy: 0.9870
Epoch 36/60
8000/8000 [==============================] - 2s 203us/step - loss: 0.0063 -
```

```
binary_accuracy: 0.9980 - val_loss: 0.0581 - val_binary_accuracy: 0.9875
Epoch 37/60
8000/8000 [==============================] - 2s 192us/step - loss: 0.0058 -
binary_accuracy: 0.9981 - val_loss: 0.0586 - val_binary_accuracy: 0.9875
Epoch 38/60
8000/8000 [==============================] - 1s 187us/step - loss: 0.0057 -
binary_accuracy: 0.9980 - val_loss: 0.0532 - val_binary_accuracy: 0.9870
Epoch 39/60
8000/8000 [==============================] - 2s 194us/step - loss: 0.0040 -
binary_accuracy: 0.9991 - val_loss: 0.0463 - val_binary_accuracy: 0.9875
Epoch 40/60
8000/8000 [==============================] - 2s 199us/step - loss: 0.0040 -
binary_accuracy: 0.9993 - val_loss: 0.0411 - val_binary_accuracy: 0.9870
Epoch 41/60
8000/8000 [==============================] - 2s 209us/step - loss: 0.0047 -
binary_accuracy: 0.9989 - val_loss: 0.0505 - val_binary_accuracy: 0.9875
Epoch 42/60
8000/8000 [==============================] - 2s 189us/step - loss: 0.0043 -
binary_accuracy: 0.9990 - val_loss: 0.0568 - val_binary_accuracy: 0.9875
Epoch 43/60
8000/8000 [==============================] - 2s 188us/step - loss: 0.0050 -
binary_accuracy: 0.9984 - val_loss: 0.0615 - val_binary_accuracy: 0.9875
Epoch 44/60
8000/8000 [==============================] - 1s 182us/step - loss: 0.0042 -
binary_accuracy: 0.9990 - val_loss: 0.0584 - val_binary_accuracy: 0.9875
Epoch 45/60
8000/8000 [==============================] - 2s 207us/step - loss: 0.0045 -
binary_accuracy: 0.9984 - val_loss: 0.0631 - val_binary_accuracy: 0.9870
Epoch 46/60
8000/8000 [==============================] - 2s 191us/step - loss: 0.0041 -
binary_accuracy: 0.9988 - val_loss: 0.0576 - val_binary_accuracy: 0.9875
Epoch 47/60
8000/8000 [==============================] - 2s 194us/step - loss: 0.0045 -
binary_accuracy: 0.9981 - val_loss: 0.0731 - val_binary_accuracy: 0.9870
Epoch 48/60
8000/8000 [==============================] - 1s 182us/step - loss: 0.0046 -
binary_accuracy: 0.9985 - val_loss: 0.0594 - val_binary_accuracy: 0.9870
Epoch 49/60
8000/8000 [==============================] - 1s 180us/step - loss: 0.0036 -
binary_accuracy: 0.9990 - val_loss: 0.0527 - val_binary_accuracy: 0.9875
Epoch 50/60
8000/8000 [==============================] - 1s 184us/step - loss: 0.0030 -
binary_accuracy: 0.9990 - val_loss: 0.0575 - val_binary_accuracy: 0.9880
Epoch 51/60
8000/8000 [==============================] - ETA: 0s - loss: 0.0027 -
binary_accuracy: 0.999 - 2s 210us/step - loss: 0.0027 - binary_accuracy: 0.9993
- val_loss: 0.0639 - val_binary_accuracy: 0.9875
Epoch 52/60
```

```
8000/8000 [==============================] - 2s 215us/step - loss: 0.0028 -
binary_accuracy: 0.9994 - val_loss: 0.0583 - val_binary_accuracy: 0.9860
Epoch 53/60
8000/8000 [==============================] - 2s 217us/step - loss: 0.0031 -
binary_accuracy: 0.9993 - val_loss: 0.0629 - val_binary_accuracy: 0.9880
Epoch 54/60
8000/8000 [==============================] - 2s 217us/step - loss: 0.0037 -
binary_accuracy: 0.9990 - val_loss: 0.0604 - val_binary_accuracy: 0.9880
Epoch 55/60
8000/8000 [==============================] - 2s 218us/step - loss: 0.0025 -
binary_accuracy: 0.9993 - val_loss: 0.0658 - val_binary_accuracy: 0.9875
Epoch 56/60
8000/8000 [==============================] - 2s 221us/step - loss: 0.0031 -
binary_accuracy: 0.9994 - val_loss: 0.0747 - val_binary_accuracy: 0.9870
Epoch 57/60
8000/8000 [==============================] - 2s 216us/step - loss: 0.0024 -
binary_accuracy: 0.9995 - val_loss: 0.0706 - val_binary_accuracy: 0.9870
Epoch 58/60
8000/8000 [==============================] - 2s 218us/step - loss: 0.0027 -
binary_accuracy: 0.9990 - val_loss: 0.0535 - val_binary_accuracy: 0.9875
Epoch 59/60
8000/8000 [==============================] - 2s 272us/step - loss: 0.0033 -
binary_accuracy: 0.9991 - val_loss: 0.0560 - val_binary_accuracy: 0.9860
Epoch 60/60
8000/8000 [==============================] - 3s 315us/step - loss: 0.0025 -
binary_accuracy: 0.9995 - val_loss: 0.0711 - val_binary_accuracy: 0.9870
binary_accurac
```

[9]:
```python
# plotting history for every batch
fig, ax1 = plt.subplots()
ax1.plot(training_log.history['loss'], '--', color='b', label='training loss')
ax1.plot(training_log.history['val_loss'], color='b', label='validation loss')
ax1.set_xlabel('epochs')
ax1.set_ylabel('loss', color='b')
ax1.tick_params('y', colors='b')
ax1.legend(loc='upper left')
ax2 = ax1.twinx()
ax2.plot(training_log.history['binary_accuracy'], '--', color='r',␣
 ↪label='training accuracy')
ax2.plot(training_log.history['val_binary_accuracy'], color='r',␣
 ↪label='validation accuracy')
ax2.set_ylabel('accuracy (%)', color='r')
ax2.tick_params('y', colors='r')
ax2.legend(loc='center right')
plt.show()
```



The distribution has skew in the sample distribution (specifically, toward the majority class/class 1 where the sum of 30 digits being >= 100). So, when the network learns, it considers the majority class more heavily than the minority class/class 0(series sum < 100), and adjusts accordingly. If we view class 1 as positive and class 0 as negative, it will have higher sensitivty than specificity. The network could learn to only output class 1 and it would still very high accuracy results! So, we could say the network isn't discriminiative. Rather, its accuracy is a measure of the underlying class distribution.

To prove this we could compare accuracy of the network on each individual class. Surely, the accuracy for the majority class will be better than that of the minority class.

As a solution to this we could create synthetic examples (made by duplicating or tranforming data in the minority class) to match the number of examples majority class, thus oversampling the minority class. Or, we could undersample the majority class, and train the network on all n series in the minority class and only n series from the majority class, taken at random.

This question has made us consider a very common problem in deep learning: the problem of *imbalanced data*.