Assignment 5.2 Implement the news classifier - classifying-newswires

October 3, 2020

0.0.1 Assignment 5.2

Implement the news classifier found in section 3.5 of Deep Learning with Python as a Luigi workflow. Example code and results can be found in dsc650/assignments/assignment05/.

[1]:
```
import keras
keras.__version__
```

[1]: '2.4.3'

0.0.2 Classifying newswires: a multi-class classification example

0.0.3 The Reuters dataset

We will be working with the Reuters dataset, a set of short newswires and their topics, published by Reuters in 1986. It's a very 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.

Like IMDB and MNIST, the Reuters dataset comes packaged as part of Keras. Let's take a look right away:

[2]:
```
from keras.datasets import reuters

(train_data, train_labels), (test_data, test_labels) = reuters.
↪load_data(num_words=10000)
```

Like with the IMDB dataset, the argument num_words=10000 restricts the data to the 10,000 most frequently occurring words found in the data.

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

[3]:
```
len(train_data)
```

[3]: 8982

[4]:
```
len(test_data)
```

[4]: 2246

As with the IMDB reviews, each example is a list of integers (word indices):

[5]:
```
train_data[10]
```

[5]:
```
[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]
```

[6]:
```
word_index = reuters.get_word_index()
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])

# Note that our indices were offset by 3
# because 0, 1 and 2 are reserved indices for "padding", "start of sequence",␣
↪and "unknown".

decoded_newswire = ' '.join([reverse_word_index.get(i - 3, '?') for i in␣
↪train_data[0]])
```

[7]:
```
decoded_newswire
```

[7]: '? ? ? said as a result of its december acquisition of space co it expects earnings per share in 1987 of 1 15 to 1 30 dlrs per share up from 70 cts in 1986 the company said pretax net should rise to nine to 10 mln dlrs from six mln dlrs in 1986 and rental operation revenues to 19 to 22 mln dlrs from 12 5 mln dlrs it said cash flow per share this year should be 2 50 to three dlrs reuter 3'

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

[8]:
```
train_labels[10]
```

[8]: 3

0.0.4 Preparing the data

We can vectorize the data with the exact same code as in our previous example:

[9]:
```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results

# Our vectorized training data
x_train = vectorize_sequences(train_data)
# Our vectorized test data
x_test = vectorize_sequences(test_data)
```

To vectorize the labels, there are two possibilities: we could just cast the label list as an integer tensor, or we could use a "one-hot" encoding. One-hot encoding is a widely used format for categorical data, also called "categorical encoding". For a more detailed explanation of one-hot encoding, you can refer to Chapter 6, Section 1. In our case, one-hot encoding of our labels consists in embedding each label as an all-zero vector with a 1 in the place of the label index, e.g.:

[10]:
```
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

# Our vectorized training labels
one_hot_train_labels = to_one_hot(train_labels)
# Our vectorized test labels
one_hot_test_labels = to_one_hot(test_labels)
```

Note that there is a built-in way to do this in Keras, which you have already seen in action in our MNIST example:

[11]:
```python
from keras.utils.np_utils import to_categorical

one_hot_train_labels = to_categorical(train_labels)
one_hot_test_labels = to_categorical(test_labels)
```

Building our network

This topic classification problem looks very similar to our previous movie review classification problem: in both cases, we are trying to classify short snippets of text. There is however a new constraint here: the number of output classes has gone from 2 to 46, i.e. the dimensionality of the output space is much larger.

In a stack of Dense layers like what we were using, 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: each layer can potentially become an "information bottleneck". In our previous example, we were using 16-dimensional intermediate layers, but a 16-dimensional space may be too limited to learn to separate 46 different classes: such small layers may act as information bottlenecks, permanently dropping relevant information.

For this reason we will use larger layers. Let's go with 64 units:

[12]:
```python
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))
```

There are two other things you should note about this architecture:

```
We are ending the network with a Dense layer of size 46. This means that for each input sample
The last layer uses a softmax activation. You have already seen this pattern in the MNIST examp
```

The best loss function to use in this case is categorical_crossentropy. It measures the distance between two probability distributions: in our case, between the probability distribution output by our network, and the true distribution of the labels. By minimizing the distance between these two distributions, we train our network to output something as close as possible to the true labels.

[13]:
```python
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

0.0.5 Validating our approach

Let's set apart 1,000 samples in our training data to use as a validation set:

[14]:
```python
x_val = x_train[:1000]
partial_x_train = x_train[1000:]
```

```
y_val = one_hot_train_labels[:1000]
partial_y_train = one_hot_train_labels[1000:]
```

Now let's train our network for 20 epochs:

[15]:
```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

```
Epoch 1/20
16/16 [==============================] - 1s 36ms/step - loss: 2.8142 - accuracy:
0.5158 - val_loss: 1.9043 - val_accuracy: 0.6320
Epoch 2/20
16/16 [==============================] - 0s 19ms/step - loss: 1.5327 - accuracy:
0.6904 - val_loss: 1.3879 - val_accuracy: 0.7010
Epoch 3/20
16/16 [==============================] - 0s 19ms/step - loss: 1.1276 - accuracy:
0.7561 - val_loss: 1.1811 - val_accuracy: 0.7490
Epoch 4/20
16/16 [==============================] - 0s 16ms/step - loss: 0.8892 - accuracy:
0.8108 - val_loss: 1.0713 - val_accuracy: 0.7700
Epoch 5/20
16/16 [==============================] - 0s 17ms/step - loss: 0.7144 - accuracy:
0.8485 - val_loss: 0.9865 - val_accuracy: 0.7870
Epoch 6/20
16/16 [==============================] - 0s 16ms/step - loss: 0.5716 - accuracy:
0.8867 - val_loss: 0.9395 - val_accuracy: 0.8010
Epoch 7/20
16/16 [==============================] - 0s 19ms/step - loss: 0.4618 - accuracy:
0.9068 - val_loss: 0.8940 - val_accuracy: 0.8150
Epoch 8/20
16/16 [==============================] - 0s 18ms/step - loss: 0.3783 - accuracy:
0.9207 - val_loss: 0.8982 - val_accuracy: 0.8090
Epoch 9/20
16/16 [==============================] - 0s 14ms/step - loss: 0.3107 - accuracy:
0.9341 - val_loss: 0.9203 - val_accuracy: 0.8000
Epoch 10/20
16/16 [==============================] - 0s 14ms/step - loss: 0.2636 - accuracy:
0.9422 - val_loss: 0.8883 - val_accuracy: 0.8150
Epoch 11/20
16/16 [==============================] - 0s 14ms/step - loss: 0.2265 - accuracy:
0.9478 - val_loss: 0.9101 - val_accuracy: 0.8060
Epoch 12/20
16/16 [==============================] - 0s 16ms/step - loss: 0.1993 - accuracy:
0.9514 - val_loss: 0.9290 - val_accuracy: 0.8200
Epoch 13/20
```

```
16/16 [==============================] - 0s 16ms/step - loss: 0.1795 - accuracy:
0.9515 - val_loss: 0.9199 - val_accuracy: 0.8220
Epoch 14/20
16/16 [==============================] - 0s 17ms/step - loss: 0.1598 - accuracy:
0.9536 - val_loss: 0.9784 - val_accuracy: 0.7960
Epoch 15/20
16/16 [==============================] - 0s 16ms/step - loss: 0.1472 - accuracy:
0.9551 - val_loss: 0.9445 - val_accuracy: 0.8170
Epoch 16/20
16/16 [==============================] - 0s 16ms/step - loss: 0.1369 - accuracy:
0.9567 - val_loss: 1.0156 - val_accuracy: 0.7940
Epoch 17/20
16/16 [==============================] - 0s 15ms/step - loss: 0.1313 - accuracy:
0.9562 - val_loss: 0.9840 - val_accuracy: 0.8070
Epoch 18/20
16/16 [==============================] - 0s 17ms/step - loss: 0.1217 - accuracy:
0.9572 - val_loss: 1.0551 - val_accuracy: 0.7970
Epoch 19/20
16/16 [==============================] - 0s 13ms/step - loss: 0.1208 - accuracy:
0.9573 - val_loss: 1.0524 - val_accuracy: 0.7980
Epoch 20/20
16/16 [==============================] - 0s 19ms/step - loss: 0.1121 - accuracy:
0.9579 - val_loss: 1.0218 - val_accuracy: 0.8150
```

To display its loss and accuracy curves:

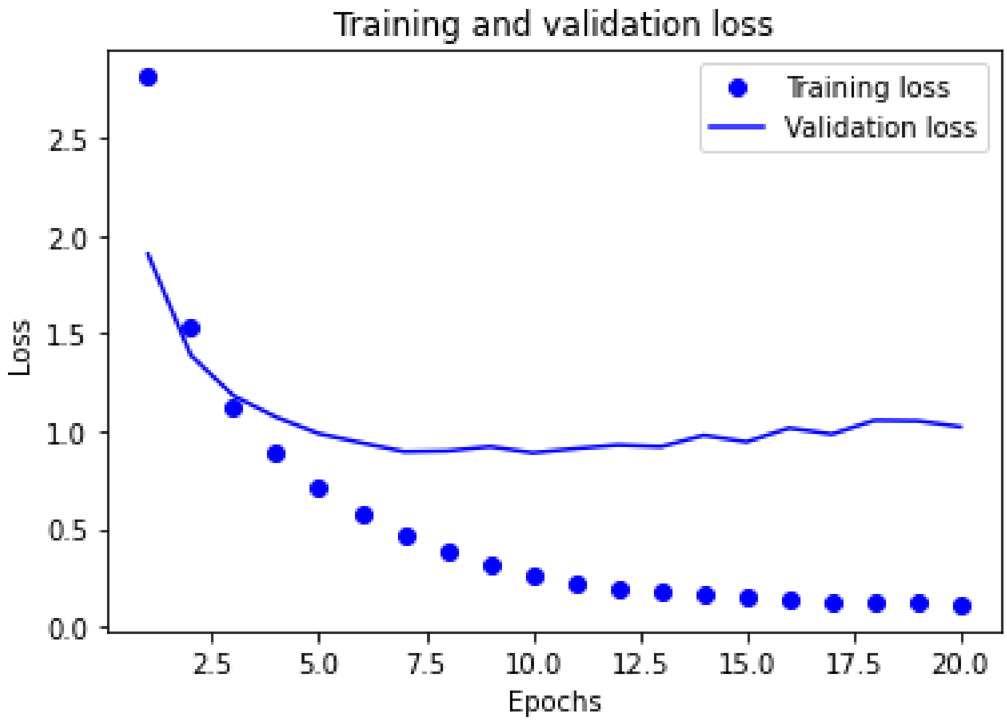
[16]:
```python
import matplotlib.pyplot as plt

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()
```

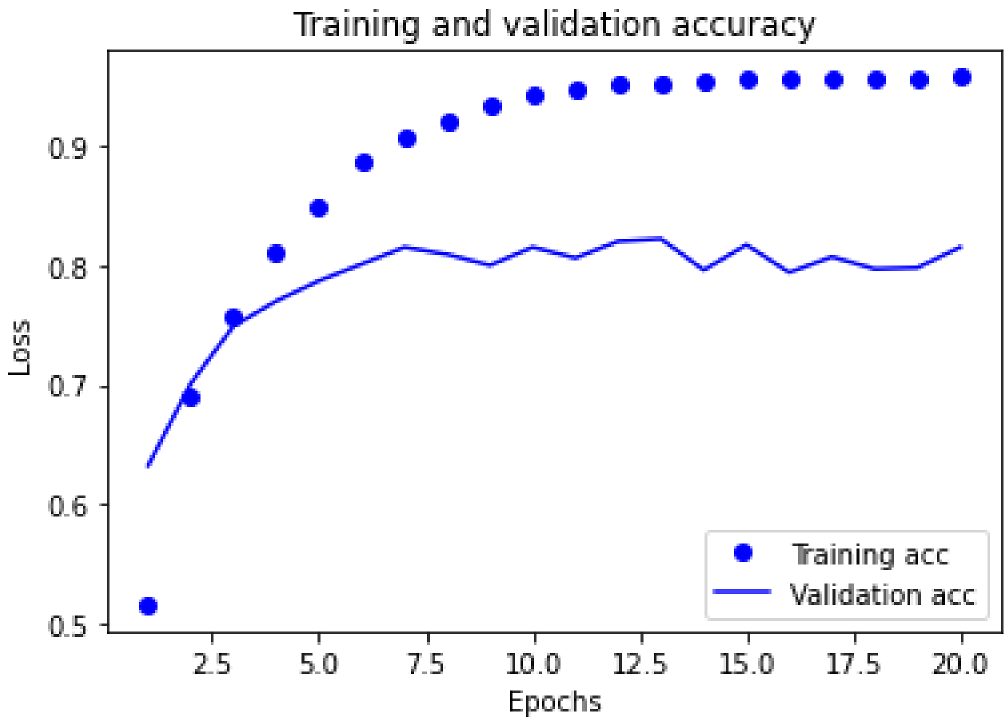


[17]:
```
plt.clf()   # clear figure

acc = history.history['accuracy']
val_acc = history.history['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('Loss')
plt.legend()

plt.show()
```



It seems that the network starts overfitting after 8 epochs. Let's train a new network from scratch for 8 epochs, then let's evaluate it on the test set:

[18]:
```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(partial_x_train,
          partial_y_train,
          epochs=8,
          batch_size=512,
          validation_data=(x_val, y_val))
results = model.evaluate(x_test, one_hot_test_labels)
```

```
Epoch 1/8
16/16 [==============================] - 0s 21ms/step - loss: 2.6389 - accuracy:
0.5028 - val_loss: 1.7080 - val_accuracy: 0.6550
Epoch 2/8
16/16 [==============================] - 0s 18ms/step - loss: 1.4033 - accuracy:
0.7072 - val_loss: 1.2917 - val_accuracy: 0.7000
```

```
Epoch 3/8
16/16 [==============================] - 0s 17ms/step - loss: 1.0398 - accuracy:
0.7767 - val_loss: 1.1159 - val_accuracy: 0.7560
Epoch 4/8
16/16 [==============================] - 0s 17ms/step - loss: 0.8206 - accuracy:
0.8226 - val_loss: 1.0611 - val_accuracy: 0.7730
Epoch 5/8
16/16 [==============================] - 0s 16ms/step - loss: 0.6531 - accuracy:
0.8656 - val_loss: 0.9670 - val_accuracy: 0.7970
Epoch 6/8
16/16 [==============================] - 0s 16ms/step - loss: 0.5239 - accuracy:
0.8925 - val_loss: 0.9174 - val_accuracy: 0.8120
Epoch 7/8
16/16 [==============================] - 0s 17ms/step - loss: 0.4209 - accuracy:
0.9137 - val_loss: 0.9073 - val_accuracy: 0.8100
Epoch 8/8
16/16 [==============================] - 0s 14ms/step - loss: 0.3443 - accuracy:
0.9297 - val_loss: 0.9643 - val_accuracy: 0.7950
71/71 [==============================] - 0s 2ms/step - loss: 1.0490 - accuracy:
0.7667
```

[19]:
```
results
```

[19]:
```
[1.0489784479141235, 0.7666963338851929]
```

Our approach reaches an accuracy of ~78%. With a balanced binary classification problem, the accuracy reached by a purely random classifier would be 50%, but in our case it is closer to 19%, so our results seem pretty good, at least when compared to a random baseline:

[20]:
```
import copy

test_labels_copy = copy.copy(test_labels)
np.random.shuffle(test_labels_copy)
float(np.sum(np.array(test_labels) == np.array(test_labels_copy))) /␣
↪len(test_labels)
```

[20]:
```
0.18432769367764915
```

0.0.6 Generating predictions on new data

We can verify that the predict method of our model instance returns a probability distribution over all 46 topics. Let's generate topic predictions for all of the test data:

[21]:
```
predictions = model.predict(x_test)
#predictions
```

Each entry in predictions is a vector of length 46:

[22]:
```
predictions[0].shape
```

[22]: (46,)

The coefficients in this vector sum to 1:

[23]:
```
np.sum(predictions[0])
```

[23]: 0.99999994

The largest entry is the predicted class, i.e. the class with the highest probability:

[24]:
```
np.argmax(predictions[0])
```

[24]: 4

0.0.7 A different way to handle the labels and the loss

We mentioned earlier that another way to encode the labels would be to cast them as an integer tensor, like such:

[25]:
```
y_train = np.array(train_labels)
y_test = np.array(test_labels)
```

The only thing it would change is the choice of the loss function. Our previous loss, categorical_crossentropy, expects the labels to follow a categorical encoding. With integer labels, we should use sparse_categorical_crossentropy:

[26]:
```
model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy',␣
 ↪metrics=['acc'])
```

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

0.0.8 On the importance of having sufficiently large intermediate layers

We mentioned earlier that since our final outputs were 46-dimensional, we should avoid intermediate layers with much less than 46 hidden units. Now let's try to see what happens when we introduce an information bottleneck by having intermediate layers significantly less than 46-dimensional, e.g. 4-dimensional.

[27]:
```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))

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

```
model.fit(partial_x_train,
          partial_y_train,
          epochs=20,
          batch_size=128,
          validation_data=(x_val, y_val))
```

```
Epoch 1/20
63/63 [==============================] - 1s 9ms/step - loss: 2.9928 - accuracy:
0.1958 - val_loss: 2.2332 - val_accuracy: 0.3480
Epoch 2/20
63/63 [==============================] - 0s 7ms/step - loss: 1.7848 - accuracy:
0.6346 - val_loss: 1.5289 - val_accuracy: 0.6650
Epoch 3/20
63/63 [==============================] - 0s 7ms/step - loss: 1.3237 - accuracy:
0.7066 - val_loss: 1.3489 - val_accuracy: 0.7050
Epoch 4/20
63/63 [==============================] - 0s 7ms/step - loss: 1.1386 - accuracy:
0.7359 - val_loss: 1.2911 - val_accuracy: 0.7040
Epoch 5/20
63/63 [==============================] - 0s 7ms/step - loss: 1.0234 - accuracy:
0.7502 - val_loss: 1.2698 - val_accuracy: 0.6970
Epoch 6/20
63/63 [==============================] - 0s 7ms/step - loss: 0.9363 - accuracy:
0.7641 - val_loss: 1.2451 - val_accuracy: 0.7220
Epoch 7/20
63/63 [==============================] - 1s 9ms/step - loss: 0.8625 - accuracy:
0.7826 - val_loss: 1.2508 - val_accuracy: 0.7140
Epoch 8/20
63/63 [==============================] - 1s 14ms/step - loss: 0.8006 - accuracy:
0.7968 - val_loss: 1.2626 - val_accuracy: 0.7120
Epoch 9/20
63/63 [==============================] - 1s 9ms/step - loss: 0.7454 - accuracy:
0.8121 - val_loss: 1.2833 - val_accuracy: 0.7190
Epoch 10/20
63/63 [==============================] - 0s 7ms/step - loss: 0.6938 - accuracy:
0.8274 - val_loss: 1.3043 - val_accuracy: 0.7220
Epoch 11/20
63/63 [==============================] - 0s 7ms/step - loss: 0.6520 - accuracy:
0.8340 - val_loss: 1.3333 - val_accuracy: 0.7160
Epoch 12/20
63/63 [==============================] - 0s 7ms/step - loss: 0.6130 - accuracy:
0.8401 - val_loss: 1.3830 - val_accuracy: 0.7090
Epoch 13/20
63/63 [==============================] - 0s 7ms/step - loss: 0.5821 - accuracy:
0.8449 - val_loss: 1.4149 - val_accuracy: 0.7080
Epoch 14/20
63/63 [==============================] - 0s 6ms/step - loss: 0.5529 - accuracy:
```

```
0.8494 - val_loss: 1.4806 - val_accuracy: 0.6970
Epoch 15/20
63/63 [==============================] - 0s 7ms/step - loss: 0.5267 - accuracy:
0.8530 - val_loss: 1.5140 - val_accuracy: 0.7030
Epoch 16/20
63/63 [==============================] - 0s 7ms/step - loss: 0.5023 - accuracy:
0.8569 - val_loss: 1.5629 - val_accuracy: 0.7140
Epoch 17/20
63/63 [==============================] - 1s 8ms/step - loss: 0.4857 - accuracy:
0.8598 - val_loss: 1.6084 - val_accuracy: 0.7150
Epoch 18/20
63/63 [==============================] - 0s 7ms/step - loss: 0.4654 - accuracy:
0.8647 - val_loss: 1.7012 - val_accuracy: 0.7000
Epoch 19/20
63/63 [==============================] - 1s 10ms/step - loss: 0.4469 - accuracy:
0.8715 - val_loss: 1.6981 - val_accuracy: 0.7090
Epoch 20/20
63/63 [==============================] - 0s 7ms/step - loss: 0.4325 - accuracy:
0.8773 - val_loss: 1.7539 - val_accuracy: 0.7150
```

```
[27]: <tensorflow.python.keras.callbacks.History at 0x7f5093009df0>
```

Our network now seems to peak at ~71% test accuracy, a 8% absolute drop. This drop is mostly due to the fact that we are now trying to compress a lot of information (enough information to recover the separation hyperplanes of 46 classes) into an intermediate space that is too low-dimensional. The network is able to cram most of the necessary information into these 8-dimensional representations, but not all of it.

0.0.9 Further experiments

```
-Try using larger or smaller layers: 32 units, 128 units...
-We were using two hidden layers. Now try to use a single hidden layer, or three hidden layers
```

0.0.10 Wrapping up

Here's what you should take away from this example:

1.If you are trying to classify data points between N classes, your network should end with a Dense layer of size N.

2.In a single-label, multi-class classification problem, your network should end with a softmax activation, so that it will output a probability distribution over the N output classes.

3.Categorical crossentropy is almost always the loss function you should use for such problems. It minimizes the distance between the probability distributions output by the network, and the true distribution of the targets.

4.There are two ways to handle labels in multi-class classification: Encoding the labels via "categorical encoding" (also known as "one-hot encoding") and using categorical_crossentropy as your loss function. Encoding the labels as integers and using the sparse_categorical_crossentropy loss function.

5.If you need to classify data into a large number of categories, then you should avoid creating information bottlenecks in your network by having intermediate layers that are too small.

End

https://github.com/fchollet/deep-learning-with-python-notebooks