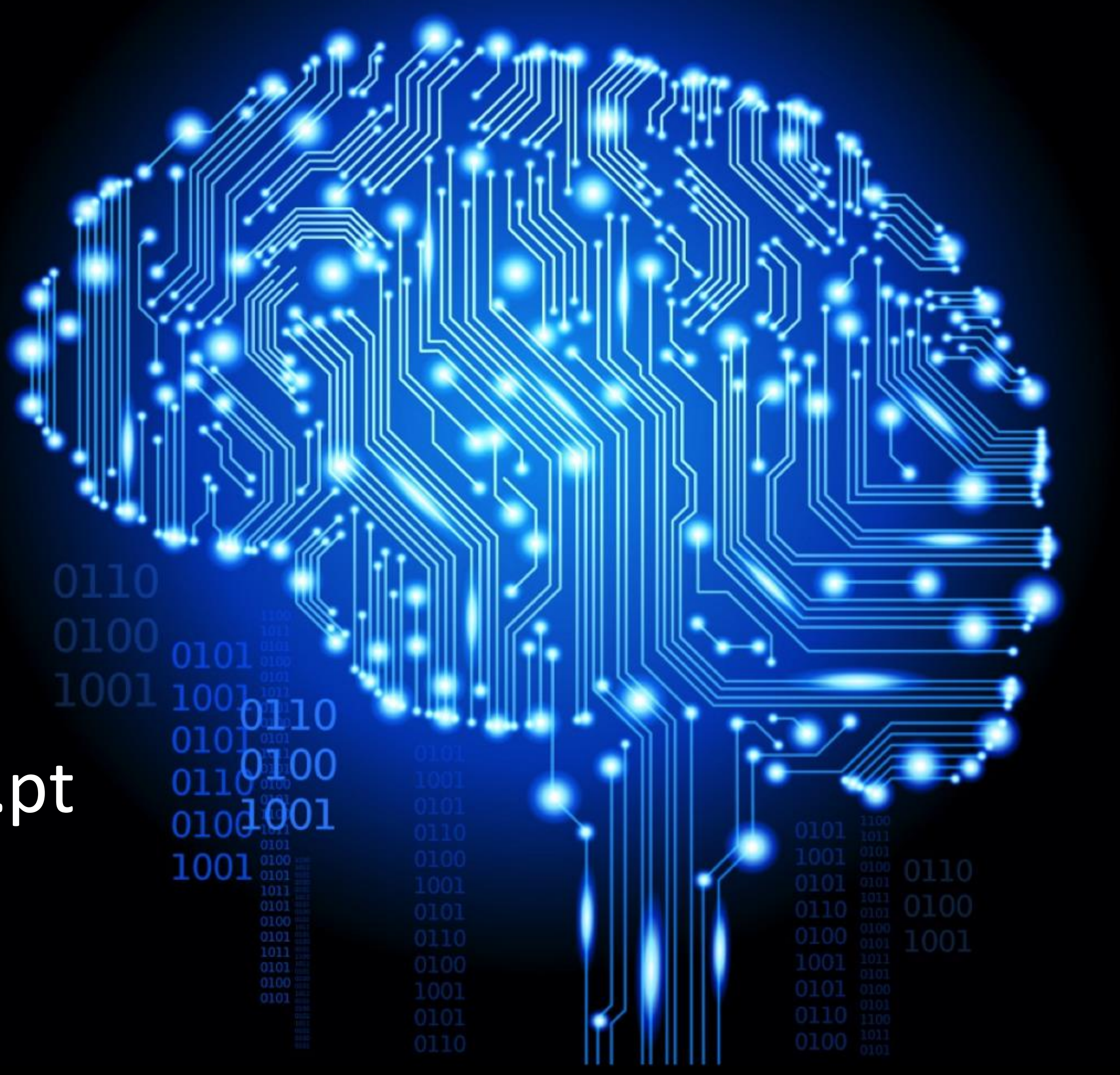
DEEP LEARNING

Spring 2021

MAURO CASTELLI
mcastelli@novaims.unl.pt



Multiclass Classification

Multiclass Classification

We now consider the case in which the target labels are greater than 2.

In this case we are facing a multiclass classification problem.

By using Keras, we will build a network to classify Reuters news into 46 mutually exclusive topics. Because you have many classes, this problem is an instance of *multiclass classification*; and because each data point should be classified into only one category, the problem is more specifically an instance of *single-label, multiclass classification*.

Multiclass Classification: Reuters Dataset

We’ll work with the Reuters dataset, a set of short newswires and their topics, published by Reuters in 1986. It’s a 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.

Multiclass Classification: Reuters Dataset

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

As with the IMDB dataset, the argument `num_words=10000` restricts the data to the 10000 most frequently occurring words found in the data.

You have 8982 training examples and 2246 test examples:

```
>>> len(train_data)
8982
>>> len(test_data)
2246
```

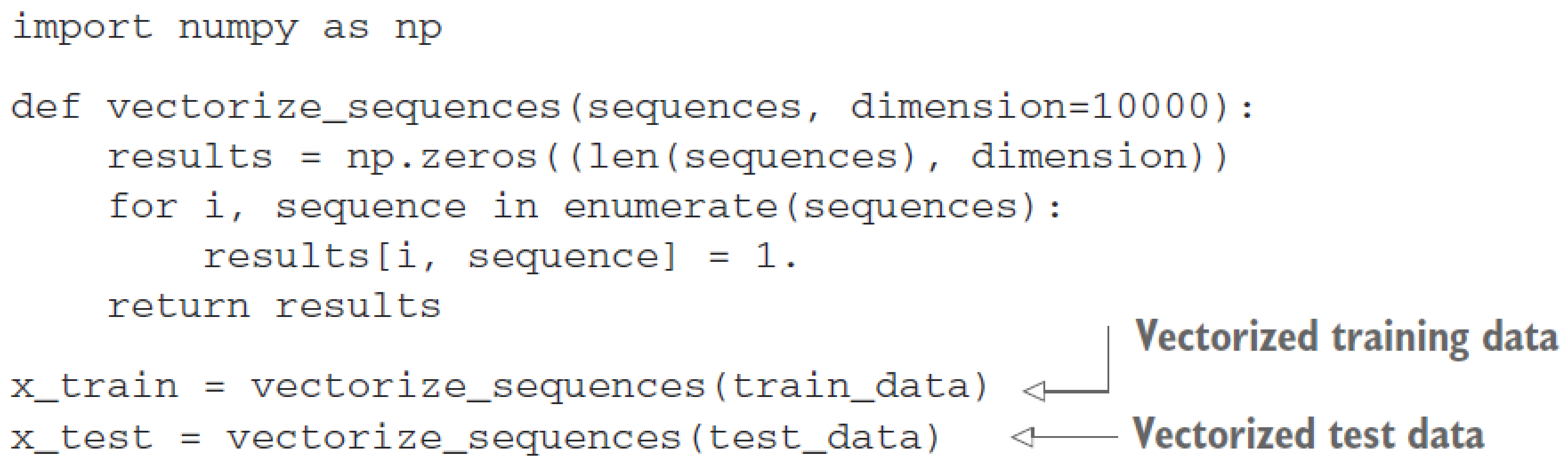
Multiclass Classification: Reuters Dataset

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

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



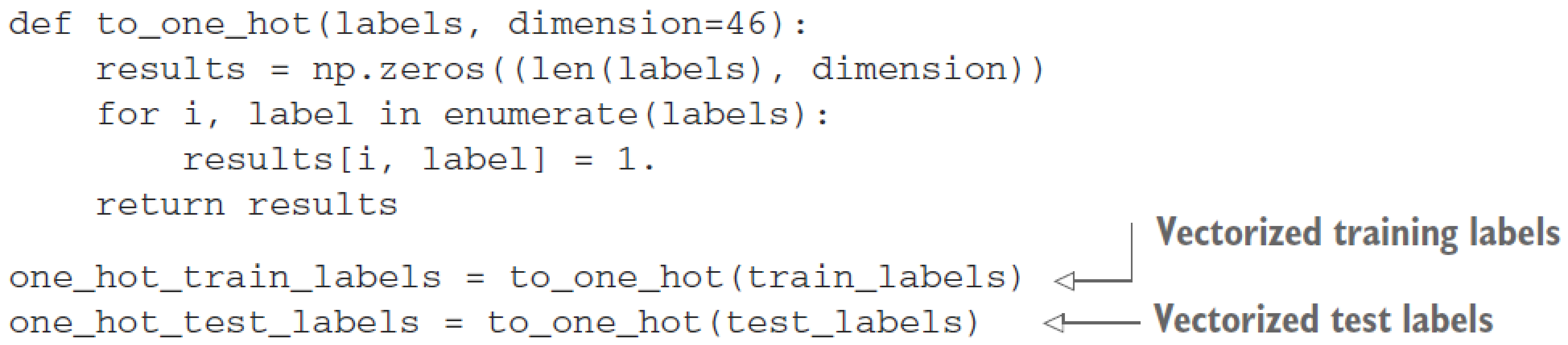
```
import numpy as np

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

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

Vectorized training data

Vectorized test data

To vectorize the labels we use one-hot encoding, a widely used format for categorical data, also called *categorical encoding*.

Preparing the data



```
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

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

Note that there is a built-in way to do this in Keras:

```
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 the Network

This topic-classification problem looks similar to the previous movie-review classification problem: in both cases, we're trying to classify short snippets of text.

But…the number of output classes has gone from 2 to 46. The dimensionality of the output space is much larger.

In a stack of `Dense` layers like that we've been 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.

Building the Network

In the previous example, we used 16-dimensional intermediate layers, but a 16-dimensional space may be too limited to learn to separate 46 different classes: such small layers may permanently drop relevant information.

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

```
from keras import models
from keras import layers
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_dim=10000))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))
```

Building the Network

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

- ✓ You end the network with a `Dense` layer of size 46. This means for each input sample, the network will output a 46-dimensional vector. Each entry in this vector (each dimension) will encode a different output class.
- ✓ The last layer uses a `softmax` activation. It means the network will output a *probability distribution* over the 46 different output classes—for every input sample, the network 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.

Building the Network

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 network and the true distribution of the labels.

By minimizing the distance between these two distributions, you train the network to output something as close as possible to the true labels.

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

Validation Set

Let’s set apart 1000 samples in the training data to use as a validation set.

```
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:]
```

And now let’s train our network!

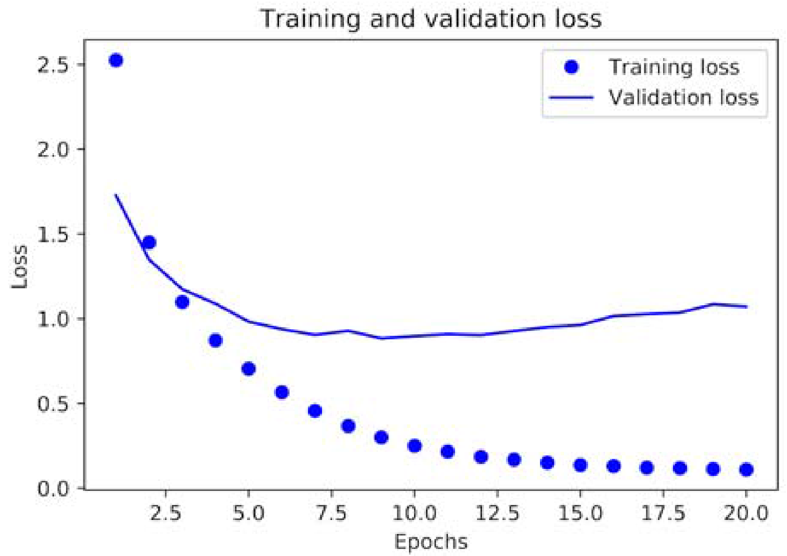
Training the model

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

And finally, let’s display its loss and accuracy curves (we use the same code as the IMDB example).

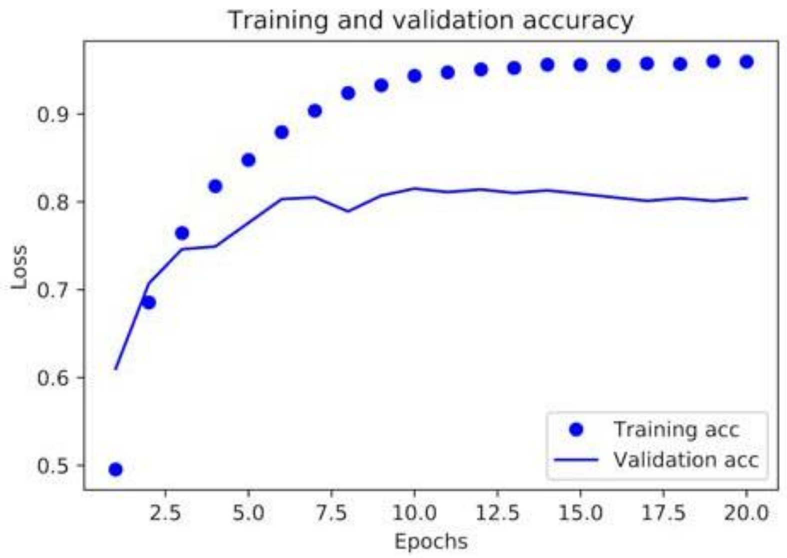
Plotting the training and validation loss

```
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()
```



Plotting the training and validation accuracy

```
plt.clf()      # clears the figure
acc = history.history['acc']
val_acc = history.history['val_acc']
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()
```



Observations

The network begins to overfit after nine epochs. Let’s train a new network from scratch for nine epochs and then evaluate it on the test set.

```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_dim=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=9, batch_size=512, validation_data=(x_val, y_val))
results = model.evaluate(x_test, one_hot_test_labels)
```

Here are the final results:

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

Generating predictions on new data

You can verify that the `predict` method of the model instance returns a probability distribution over all 46 topics. Let’s generate topic predictions for all of the test data.

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

Each entry in predictions is a vector of length 46:

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

The coefficients in this vector sum to 1:

```
>>> np.sum(predictions[0])
1.0
```

The largest entry is the predicted class—the class with the highest probability:

```
>>> np.argmax(predictions[0])
4
```

Labels and Loss

Another way to encode the labels would be to cast them as an integer tensor, like this:

```
y_train = np.array(train_labels)
y_test = np.array(test_labels)
```

The only thing this approach would change is the choice of the loss function. The loss function previously used (`categorical_crossentropy`), expects the labels to follow a categorical encoding.

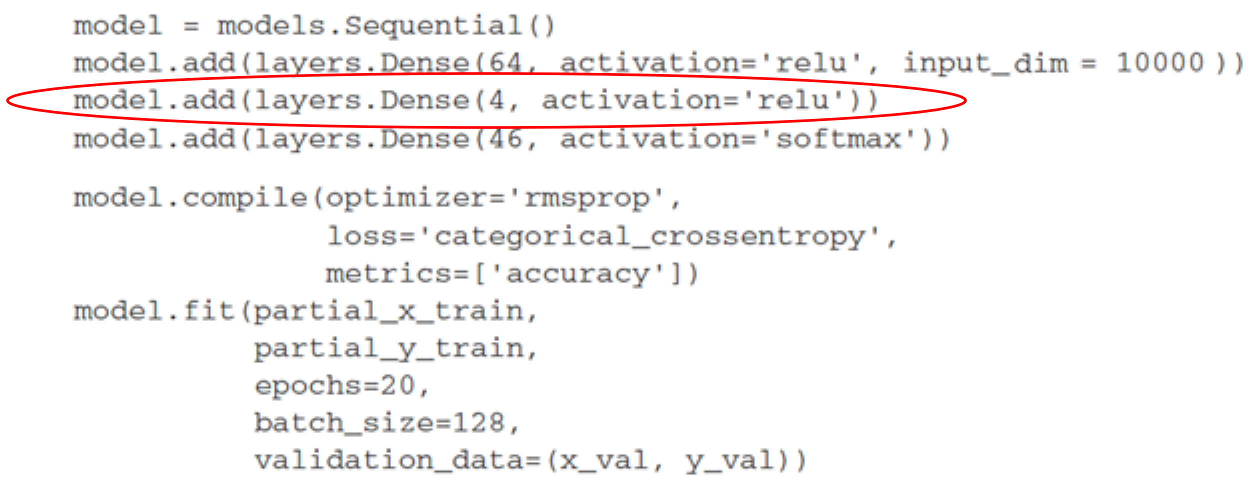
With integer labels, you should use `sparse_categorical_crossentropy`:

```
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.

The importance of having sufficiently large intermediate layers

Because the final outputs are 46-dimensional, you should avoid intermediate layers with many fewer than 46 hidden units. Now let’s see what happens when you introduce intermediate layers that are significantly less than 46-dimensional: for example, 4-dimensional.



```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_dim = 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))
```

The network now peaks at ~71% validation accuracy, an 8% absolute drop. This drop is mostly due to the fact that you’re 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.

Further experiments

- ➢ Try using larger or smaller layers: 32 units, 128 units, and so on.
- ➢ You used two hidden layers. Now try using a single hidden layer, or three hidden layers.

Take home messages

- ✓ If you’re trying to classify data points among *N* classes, your network should end with a Dense layer of size *N*.
- ✓ In a single-label, multiclass classification problem, your network 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. It minimizes the distance between the probability distributions output by the network and the true distribution of the targets.
- ✓ 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 in your network due to intermediate layers that are too small.