< Deep Learning - PART3 TF2 RNNs >

Ch 6. RNNs Workshop 1 - NLP - IMDB : One-Hot Encoding

2021/10/01

[Reference] :

- Wikipedia, **自然語言處理 (Natural Language Processing, NLP)** https://reurl.cc/WdKl6Z (https://reurl.cc/WdKl6Z)
- François Chollet, **Deep Learning with Python**, Chapter 3, Section 5, Manning, 2018.
    - [Code] : https://github.com/fchollet/deep-learning-with-python-notebooks (https://github.com/fchollet/deep-learning-with-python-notebooks)

In [1]:

```
import tensorflow as tf
tf.__version__
```

Out[1]:

```
'2.4.1'
```

The IMDB dataset

We'll be working with "IMDB dataset", a set of 50,000 highly-polarized reviews from the Internet Movie Database. They are split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting in 50% negative and 50% positive reviews.

The following code will load the dataset (when you run it for the first time, about 80MB of data will be downloaded to your machine):

In [4]:

```
imdb = tf.keras.datasets.imdb

(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_w
```

The argument `num_words=10000` means that we will only keep the top 10,000 most frequently occurring words in the training data. Rare words will be discarded. This allows us to work with vector data of manageable size.

The variables `train_data` and `test_data` are lists of reviews, each review being 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":

In [5]:

```python
print(train_data[0])
```

```
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 394
1, 4, 173, 36, 256, 5, 25, 100, 43, 838, 112, 50, 670, 2, 9, 35,
480, 284, 5, 150, 4, 172, 112, 167, 2, 336, 385, 39, 4, 172, 453
6, 1111, 17, 546, 38, 13, 447, 4, 192, 50, 16, 6, 147, 2025, 19,
14, 22, 4, 1920, 4613, 469, 4, 22, 71, 87, 12, 16, 43, 530, 38, 7
6, 15, 13, 1247, 4, 22, 17, 515, 17, 12, 16, 626, 18, 2, 5, 62, 3
86, 12, 8, 316, 8, 106, 5, 4, 2223, 5244, 16, 480, 66, 3785, 33,
4, 130, 12, 16, 38, 619, 5, 25, 124, 51, 36, 135, 48, 25, 1415, 3
3, 6, 22, 12, 215, 28, 77, 52, 5, 14, 407, 16, 82, 2, 8, 4, 107,
117, 5952, 15, 256, 4, 2, 7, 3766, 5, 723, 36, 71, 43, 530, 476,
26, 400, 317, 46, 7, 4, 2, 1029, 13, 104, 88, 4, 381, 15, 297, 9
8, 32, 2071, 56, 26, 141, 6, 194, 7486, 18, 4, 226, 22, 21, 134,
476, 26, 480, 5, 144, 30, 5535, 18, 51, 36, 28, 224, 92, 25, 104,
4, 226, 65, 16, 38, 1334, 88, 12, 16, 283, 5, 16, 4472, 113, 103,
32, 15, 16, 5345, 19, 178, 32]
```

In [6]:

```python
train_labels[0]
```

Out[6]:

```
1
```

Since we restricted ourselves to the top 10,000 most frequent words, no word index will exceed 10,000:

In [7]:

```python
max([max(sequence) for sequence in train_data])
```

Out[7]:

```
9999
```

For kicks, here's how you can quickly decode one of these reviews back to English words:

In [8]:

```
# word_index is a dictionary mapping words to an integer index
word_index = imdb.get_word_index()
# We reverse it, mapping integer indices to words
reverse_word_index = dict([(value, key) for (key, value) in word_index.item
# We decode the review; note that our indices were offset by 3
# because 0, 1 and 2 are reserved indices for "padding", "start of sequence
decoded_review = ' '.join([reverse_word_index.get(i - 3, '?')
                           for i in train_data[0]])
```

In [9]:

```
decoded_review
```

Out[9]:

```
"? this film was just brilliant casting location scenery story di
rection everyone's really suited the part they played and you cou
ld just imagine being there robert ? is an amazing actor and now
the same being director ? father came from the same scottish isla
nd as myself so i loved the fact there was a real connection with
this film the witty remarks throughout the film were great it was
just brilliant so much that i bought the film as soon as it was r
eleased for ? and would recommend it to everyone to watch and the
fly fishing was amazing really cried at the end it was so sad and
you know what they say if you cry at a film it must have been goo
d and this definitely was also ? to the two little boy's that pla
yed the ? of norman and paul they were just brilliant children ar
e often left out of the ? list i think because the stars that pla
y them all grown up are such a big profile for the whole film but
these children are amazing and should be praised for what they ha
ve done don't you think the whole story was so lovely because it
was true and was someone's life after all that was shared with us
all"
```

Preparing the data - `One-hot-encoding`

We cannot feed lists of integers into a neural network. We have to turn our lists into tensors. There are two ways we could do that:

- We could pad our lists so that they all have the same length, and turn them into an integer tensor of shape `(samples, word_indices)` , then use as first layer in our network a layer capable of handling such integer tensors (the `Embedding` layer, which we will cover in detail later in the book).
- We could one-hot-encode our lists to turn them into vectors of 0s and 1s. Concretely, this would mean for instance turning the sequence `[3, 5]` into a 10,000-dimensional vector that would be all-zeros except for indices 3 and 5, which would be ones. Then we could use as first layer in our network a `Dense` layer, capable of handling floating point vector data.

We will go with the latter solution. Let's vectorize our data, which we will do manually for maximum clarity:

In [10]:

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    # Create an all-zero matrix of shape (len(sequences), dimension)
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.  # set specific indices of results[i] to
    return results

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

Here's what our samples look like now:

In [11]:

```
x_train[0]
```

Out[11]:

```
array([0., 1., 1., ..., 0., 0., 0.])
```

We should also vectorize our labels, which is straightforward:

In [12]:

```
# Our vectorized labels
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

In [14]:

```
y_train.shape
```

Out[14]:

```
(25000,)
```

Now our data is ready to be fed into a neural network.

Building our network

Our input data is simply vectors, and our labels are scalars (1s and 0s): this is the easiest setup you will ever encounter. A type of network that performs well on such a problem would be a simple stack of fully-connected (`Dense`) layers with `relu` activations:

```
Dense(16, activation='relu')
```

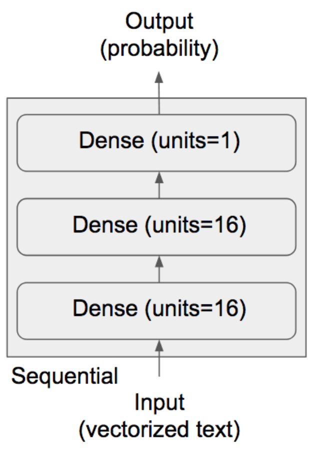
The argument being passed to each `Dense` layer (16) is the number of "hidden units" of the layer. What's a hidden unit? It's a dimension in the representation space of the layer. You may remember from the previous chapter that each such `Dense` layer with a `relu` activation implements the following chain of tensor operations:

```
output = relu(dot(W, input) + b)
```

Having 16 hidden units means that the weight matrix `W` will have shape `(input_dimension, 16)`, i.e. the dot product with `W` will project the input data onto a 16-dimensional representation space (and then we would add the bias vector `b` and apply the `relu` operation).

A `relu` (rectified linear unit) is a function meant to zero-out negative values, while a sigmoid "squashes" arbitrary values into the `[0, 1]` interval, thus outputting something that can be interpreted as a probability.

Here's what our network looks like:



Forward Propagation

In [15]:

```python
from tensorflow.keras import models, layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

In [16]:

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 16)                160016
_________________________________________________________________
dense_1 (Dense)              (None, 16)                272
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 17
=================================================================
Total params: 160,305
Trainable params: 160,305
Non-trainable params: 0
_________________________________________________________________
```

Lastly, we need to pick a loss function and an optimizer.

- Since we are facing a binary classification problem and the output of our network is a probability (we end our network with a single-unit layer with a sigmoid activation), is it best to use the `binary_crossentropy` loss.
- Crossentropy is a quantity from the field of Information Theory, that measures the "distance" between probability distributions, or in our case, between the ground-truth distribution and our predictions.

> We are passing our optimizer, loss function and metrics as strings, which is possible because `rmsprop` , `binary_crossentropy` and `accuracy` are packaged as part of Keras. Sometimes you may want to configure the parameters of your optimizer, or pass a custom loss function or metric function. This former can be done by passing an optimizer class instance as the `optimizer` argument:

Here's the step where we configure our model with the `rmsprop` optimizer and the `binary_crossentropy` loss function. Note that we will also monitor accuracy during training.

In [18]:

```python
from tensorflow.keras import optimizers

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

The latter can be done by passing function objects as the `loss` or `metrics` arguments:

In [19]:

```python
from tensorflow.keras import losses, metrics

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss=losses.binary_crossentropy,
              metrics=[metrics.binary_accuracy])
```

Validating our approach - `15000 Training Samples + 10000 Validating Samples`

In order to monitor during training the accuracy of the model on data that it has never seen before, we will create a "validation set" by setting apart 10,000 samples from the original training data:

In [20]:

```python
x_val = x_train[:10000]
partial_x_train = x_train[10000:]

y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

We will now train our model for 20 epochs (20 iterations over all samples in the `x_train` and `y_train` tensors), in mini-batches of 512 samples. At this same time we will monitor loss and accuracy on the 10,000 samples that we set apart. This is done by passing the validation data as the `validation_data` argument:

In [21]:

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

```
Epoch 1/20
30/30 [==============================] - 1s 26ms/step - loss: 0.5
902 - binary_accuracy: 0.7162 - val_loss: 0.3834 - val_binary_acc
uracy: 0.8729
Epoch 2/20
30/30 [==============================] - 0s 15ms/step - loss: 0.3
263 - binary_accuracy: 0.9016 - val_loss: 0.3072 - val_binary_acc
uracy: 0.8867
Epoch 3/20
30/30 [==============================] - 0s 14ms/step - loss: 0.2
380 - binary_accuracy: 0.9267 - val_loss: 0.2774 - val_binary_acc
uracy: 0.8925
Epoch 4/20
30/30 [==============================] - 0s 14ms/step - loss: 0.1
795 - binary_accuracy: 0.9414 - val_loss: 0.2773 - val_binary_acc
uracy: 0.8890
Epoch 5/20
30/30 [==============================] - 0s 14ms/step - loss: 0.1
434 - binary_accuracy: 0.9587 - val_loss: 0.2799 - val_binary_acc
uracy: 0.8882
Epoch 6/20
30/30 [==============================] - 0s 14ms/step - loss: 0.1
133 - binary_accuracy: 0.9694 - val_loss: 0.2950 - val_binary_acc
uracy: 0.8847
Epoch 7/20
30/30 [==============================] - 0s 13ms/step - loss: 0.0
992 - binary_accuracy: 0.9727 - val_loss: 0.3074 - val_binary_acc
uracy: 0.8845
Epoch 8/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0
793 - binary_accuracy: 0.9799 - val_loss: 0.3283 - val_binary_acc
uracy: 0.8837
Epoch 9/20
30/30 [==============================] - 1s 19ms/step - loss: 0.0
608 - binary_accuracy: 0.9859 - val_loss: 0.3830 - val_binary_acc
uracy: 0.8747
Epoch 10/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0
532 - binary_accuracy: 0.9877 - val_loss: 0.4068 - val_binary_acc
uracy: 0.8734
Epoch 11/20
30/30 [==============================] - 0s 16ms/step - loss: 0.0
413 - binary_accuracy: 0.9915 - val_loss: 0.4007 - val_binary_acc
uracy: 0.8772
Epoch 12/20
30/30 [==============================] - 0s 15ms/step - loss: 0.0
330 - binary_accuracy: 0.9938 - val_loss: 0.4391 - val_binary_acc
uracy: 0.8748
```

```
Epoch 13/20
30/30 [==============================] - 0s 17ms/step - loss: 0.0
293 - binary_accuracy: 0.9942 - val_loss: 0.4617 - val_binary_acc
uracy: 0.8732
Epoch 14/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0
203 - binary_accuracy: 0.9966 - val_loss: 0.4956 - val_binary_acc
uracy: 0.8740
Epoch 15/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0
148 - binary_accuracy: 0.9985 - val_loss: 0.5263 - val_binary_acc
uracy: 0.8713
Epoch 16/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0
105 - binary_accuracy: 0.9995 - val_loss: 0.6131 - val_binary_acc
uracy: 0.8628
Epoch 17/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0
103 - binary_accuracy: 0.9992 - val_loss: 0.6082 - val_binary_acc
uracy: 0.8638
Epoch 18/20
30/30 [==============================] - 1s 20ms/step - loss: 0.0
073 - binary_accuracy: 0.9994 - val_loss: 0.6475 - val_binary_acc
uracy: 0.8676
Epoch 19/20
30/30 [==============================] - 0s 13ms/step - loss: 0.0
052 - binary_accuracy: 0.9999 - val_loss: 0.6581 - val_binary_acc
uracy: 0.8685
Epoch 20/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0
058 - binary_accuracy: 0.9992 - val_loss: 0.6950 - val_binary_acc
uracy: 0.8656
```

On CPU, this will take less than two seconds per epoch -- training is over in 20 seconds. At the end of every epoch, there is a slight pause as the model computes its loss and accuracy on the 10,000 samples of the validation data.

Note that the call to `model.fit()` returns a `History` object. This object has a member `history`, which is a dictionary containing data about everything that happened during training. Let's take a look at it:

In [22]:

```python
history_dict = history.history
history_dict.keys()
```

Out[22]:

```
dict_keys(['loss', 'binary_accuracy', 'val_loss', 'val_binary_acc
uracy'])
```

It contains 4 entries: one per metric that was being monitored, during training and during validation. Let's use Matplotlib to plot the training and validation loss side by side, as well as the training and validation accuracy:

In [23]:

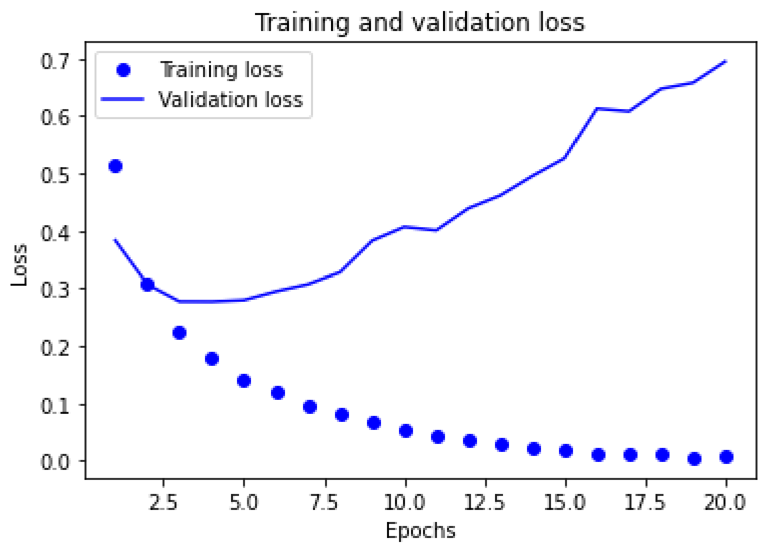
```
import matplotlib.pyplot as plt
%matplotlib inline

acc = history.history['binary_accuracy']          # 原始程式：此處為 'acc' =
val_acc = history.history['val_binary_accuracy']  # 原始程式：此處為 'val_ac
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

# "bo" is for "blue dot"
plt.plot(epochs, loss, 'bo', label='Training loss')
# b is for "solid blue line"
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()
```

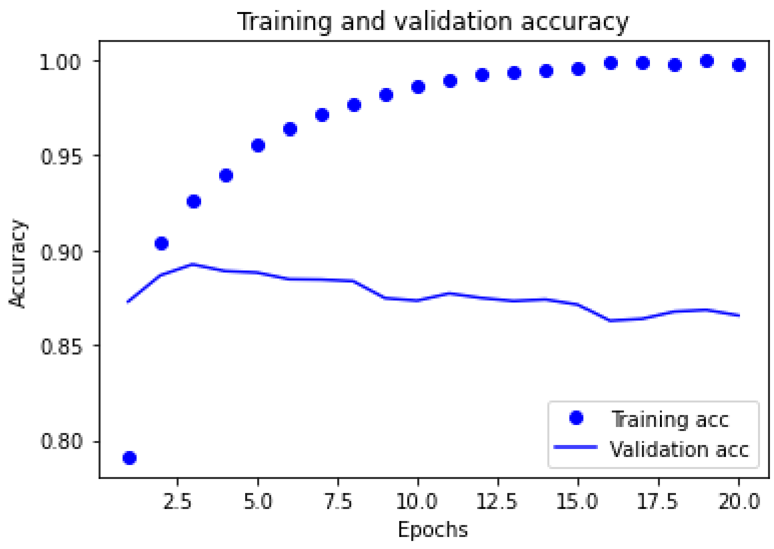


In [24]:

```
plt.clf()   # clear figure
acc_values = history_dict['binary_accuracy']          # 原始程式：此處為 'ac
val_acc_values = history_dict['val_binary_accuracy']  # 原始程式：此處為 'va

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()
```



The dots are the training loss and accuracy, while the solid lines are the validation loss and accuracy. Note that your own results may vary slightly due to a different random initialization of your network.

> As you can see, the training loss decreases with every epoch and the training accuracy increases with every epoch. That's what you would expect when running gradient descent optimization -- the quantity you are trying to minimize should get lower with every iteration. But that isn't the case for the validation loss and accuracy: they seem to peak at the fourth epoch. This is an example of what we were warning against earlier: 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. **In precise terms, what you are seeing is "overfitting": after the second epoch, we are over-optimizing on the training data, and we ended up learning representations that are specific to the training data and do not generalize to data outside of the training set.**

- **In this case, to prevent overfitting, we could simply stop training after three epochs. In general, there is a range of techniques you can leverage to mitigate overfitting, which we will cover in the next chapter.**

Let's train a new network from scratch for four epochs, then evaluate it on our test data:

In [25]:

```python
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(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, verbose=2)
```

```
Epoch 1/4
49/49 [==============================] - 1s 8ms/step - loss: 0.55
99 - accuracy: 0.7436
Epoch 2/4
49/49 [==============================] - 0s 8ms/step - loss: 0.28
16 - accuracy: 0.9071
Epoch 3/4
49/49 [==============================] - 0s 9ms/step - loss: 0.20
30 - accuracy: 0.9323
Epoch 4/4
49/49 [==============================] - 0s 9ms/step - loss: 0.16
44 - accuracy: 0.9432
782/782 - 1s - loss: 0.2921 - accuracy: 0.8834
```

In [26]:

```
results
```

Out[26]:

```
[0.29206517338752747, 0.8833600282669067]
```

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

Using a trained network to generate predictions on new data

After having trained a network, you will want to use it in a practical setting. You can generate the likelihood of reviews being positive by using the `predict` method:

In [25]:

```
model.predict(x_test)
```

Out[25]:

```
array([[0.21945101],
       [0.99947214],
       [0.881626  ],
       ...,
       [0.1477386 ],
       [0.06554589],
       [0.666247  ]], dtype=float32)
```

As you can see, the network is very confident for some samples (0.99 or more, or 0.01 or less) but less confident for others (0.6, 0.4).

Further experiments

- We were using 2 hidden layers. Try to use 1 or 3 hidden layers and see how it affects validation and test accuracy.
- Try to use layers with more hidden units or less hidden units: 32 units, 64 units...
- Try to use the `mse` loss function instead of `binary_crossentropy` .
- Try to use the `tanh` activation (an activation that was popular in the early days of neural networks) instead of `relu` .

These experiments will help convince you that the architecture choices we have made are all fairly reasonable, although they can still be improved!

Fighting Overfitting

> **[Ref]:** François Chollet, **Deep Learning with Python**, Chapter 4, Section 4, Manning, 2018.

The most common ways to prevent overfitting in neural networks:

> 1. Getting more training data.
> 2. Reducing the network's size
> 3. Adding `weight regularization`
> 4. Adding `Dropout`

Adding weight regularization

- A common way to mitigate overfitting is to put constraints on the complexity of a network by forcing its weights to only take small values, which makes the distribution of weight values more "regular".
- This is called "weight regularization", and it is done by adding to the loss function of the network a cost associated with having large weights. This cost comes in two flavors:

> - **L1 regularization**, where the cost added is proportional to the absolute value of the weights coefficients (i.e. to what is called the "L1 norm" of the weights).

- **L2 regularization**, where the cost added is proportional to the square of the value of the weights coefficients (i.e. to what is called the `"L2 norm" of the weights`).
  - L2 regularization is also called `weight decay` in the context of neural networks.

Adding dropout

Dropout is one of the most effective and most commonly used regularization techniques for neural networks, developed by Hinton and his students at the University of Toronto. Dropout, applied to a layer, consists of randomly "dropping out" (i.e. setting to zero) a number of output features of the layer during training. Let's say a given layer would normally have returned a vector `[0.2, 0.5, 1.3, 0.8, 1.1]` for a given input sample during training; after applying dropout, this vector will have a few zero entries distributed at random, e.g. `[0, 0.5, 1.3, 0, 1.1]`. The "dropout rate" is the fraction of the features that are being zeroed-out; it is usually set between 0.2 and 0.5. At test time, no units are dropped out, and instead the layer's output values are scaled down by a factor equal to the dropout rate, so as to balance for the fact that more units are active than at training time.

Consider a Numpy matrix containing the output of a layer, `layer_output`, of shape `(batch_size, features)`. At training time, we would be zero-ing out at random a fraction of the values in the matrix:

In Keras you can introduce dropout in a network via the `Dropout` layer, which gets applied to the output of layer right before it, e.g.:

In [ ]:

```
model.add(layers.Dropout(0.5))
```