SLP2_Developing a model for sentiment analysis using tensorflow

April 7, 2021

[ ]:
```
student_name = "Nikki Fitzherbert"
student_id = "13848336"
```

Step 1: Import the necessary libraries

[1]:
```
from __future__ import absolute_import, division, print_function,␣
 ↪unicode_literals
import tensorflow as tf
from tensorflow import keras
import numpy as np
```

[2]:
```
print(tf.__version__)
```

```
2.4.1
```

Step 2: Download the IMDB dataset The IMDB dataset comes packaged with TensorFlow. It has already been pre-processed such that the reviews (sequences of words) have been converted to sequences of integers, where each integer represents a specific word in a dictionary.

The following code downloads the IMDB dataset to your machine (or uses a cached copy if it has already been downloaded). The argument num_words = 10000 keeps the top 10,000 most frequently occurring words in the training data. The rare words are discarded to keep the size of the data manageable.

[3]:
```
imdb = keras.datasets.imdb

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

```
<__array_function__ internals>:5: VisibleDeprecationWarning: Creating an ndarray
from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or
ndarrays with different lengths or shapes) is deprecated. If you meant to do
this, you must specify 'dtype=object' when creating the ndarray
C:\Users\nikki\.conda\envs\python_tfgpu\lib\site-
packages\tensorflow\python\keras\datasets\imdb.py:159:
VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences
(which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths
or shapes) is deprecated. If you meant to do this, you must specify
'dtype=object' when creating the ndarray
  x_train, y_train = np.array(xs[:idx]), np.array(labels[:idx])
```

```
C:\Users\nikki\.conda\envs\python_tfgpu\lib\site-
packages\tensorflow\python\keras\datasets\imdb.py:160:
VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences
(which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths
or shapes) is deprecated. If you meant to do this, you must specify
'dtype=object' when creating the ndarray
  x_test, y_test = np.array(xs[idx:]), np.array(labels[idx:])
```

Step 3: Explore the data The dataset comes pre-processed: each example is an array of integers representing the words of the movie review. Each label is an integer value of either 0 or 1, where 0 is a negative review, and 1 is a positive review.

[4]:
```
print("Training entries: {}, labels: {}".format(len(train_data),␣
↪len(train_labels)))
```

```
Training entries: 25000, labels: 25000
```

The text of reviews has been converted to integers, where each integer represents a specific word in a dictionary. This is what the first review looks like:

[5]:
```
print(train_data[0])
```

```
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 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, 4536, 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, 76, 15, 13, 1247, 4, 22, 17, 515, 17, 12, 16, 626, 18, 2, 5, 62, 386, 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, 33, 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, 98,
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]
```

Movie reviews may be different lengths. The following code shows the number of words in the first and second reviews. Since inputs to a neural network must be the same length, we'll need to resolve these length discrepancies.

[6]:
```
len(train_data[0]), len(train_data[1])
```

[6]: (218, 189)

Step 4: Convert the integers back to words It may be useful to know how to convert integers back to text. This code creates a helper function to query a dictionary object that contains the integer to string mapping.

[7]:
```
# A dictionary mapping words to an integer index
word_index = imdb.get_word_index()
```

```
# The first indices are reserved
word_index = {k:(v+3) for k,v in word_index.items()}
word_index["<PAD>"] = 0
word_index["<START>"] = 1
word_index["<UNK>"] = 2  # unknown
word_index["<UNUSED>"] = 3

reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])

def decode_review(text):
    return ' '.join([reverse_word_index.get(i, '?') for i in text])


decode_review(train_data[0])
```

[7]:
```
"<START> this film was just brilliant casting location scenery story direction
everyone's really suited the part they played and you could just imagine being
there robert <UNK> is an amazing actor and now the same being director <UNK>
father came from the same scottish island 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
released for <UNK> 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 good and this definitely was also
<UNK> to the two little boy's that played the <UNK> of norman and paul they were
just brilliant children are often left out of the <UNK> list i think because the
stars that play 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 have 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"
```

Step 5: Prepare the data The reviews – the arrays of integers – must be converted to tensors before being fed into the neural network. This conversion can be done a couple of ways: - Convert the arrays into vectors of 0s and 1s indicating word occurrence, similar to a one-hot encoding. For example, the sequence [3, 5] would become a 10 000-dimensional vector that is all zeros except for indices 3 and 5, which are ones. Then, make this the first layer in our network – a Dense layer – that can handle floating point vector data. This approach is memory intensive, though, requiring a num_words * num_reviews size matrix. - Alternatively, we can pad the arrays so they all have the same length, then create an integer tensor of shape max_length * num_reviews. We can use an embedding layer capable of handling this shape as the first layer in our network.

This tutorial uses the second approach. Since the movie reviews must be the same length, the pad_sequences function is used to standardise the lengths:

[8]:
```
train_data = keras.preprocessing.sequence.pad_sequences(train_data,
                                                        ␣
↪value=word_index["<PAD>"],
```

```
                                                        padding='post',
                                                        maxlen=256)

test_data = keras.preprocessing.sequence.pad_sequences(test_data,
                                                       ␣
 ↪value=word_index["<PAD>"],
                                                        padding='post',
                                                        maxlen=256)
```

Let's take a look again at the length of the two example datasets we looked at in step 3. Now that the length is different from before, we will have to train the data again:

[9]:
```
len(train_data[0]), len(train_data[1])
```

[9]: (256, 256)

And inspect the padded first review:

[10]:
```
print(train_data[0])
```

```
[   1   14   22   16   43  530  973 1622 1385   65  458 4468   66 3941
    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 4536 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   76   15   13 1247    4   22   17  515   17   12   16
  626   18    2    5   62  386   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   33    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   98   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    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0]
```

Step 6: Build the model The neural network is created by stacking layers – this requires two main architectural decisions: - How many layers to use in the model? - How many hidden units to use for each layer?

In this example, the input data consists of an array of word-indices. The labels to predict are either 0 or 1.

When you have only two classes (binary classification), your model should output a single probability score. For instance, outputting 0.2 for a given input sample means '20% confidence that this

sample is in the first class (class 1), 80% that it is in the second class (class 0)'. To output such a probability score, the activation function of the last layer should be a sigmoid function.

[11]:
```
#input shape is the vocabulary count used for the movie reviews (10,000 words)
vocab_size = 10000
model = keras.Sequential()
model.add(keras.layers.Embedding(vocab_size, 16))
model.add(keras.layers.GlobalAveragePooling1D())
model.add(keras.layers.Dense(16, activation=tf.nn.relu))
model.add(keras.layers.Dense(1, activation=tf.nn.sigmoid))
```

The layers are stacked sequentially to build the classifier: - The first layer is an Embedding layer. This layer takes the integer-encoded vocabulary and looks up the embedding vector for each word-index. These vectors are learned as the model trains. The vectors add a dimension to the output array. The resulting dimensions are: (batch, sequence, embedding). - Next, a GlobalAveragePooling1D layer returns a fixed-length output vector for each example by averaging over the sequence dimension. This allows the model to handle input of variable lengths, in the simplest way possible. - This fixed-length output vector is piped through a fully connected (Dense) layer with 16 hidden units. - The last layer is densely connected with a single output node. Using the sigmoid activation function, this value is a float between 0 and 1, representing a probability, or confidence level.

[12]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 16)          160000
_________________________________________________________________
global_average_pooling1d (Gl (None, 16)                0
_________________________________________________________________
dense (Dense)                (None, 16)                272
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 17
=================================================================
Total params: 160,289
Trainable params: 160,289
Non-trainable params: 0
_________________________________________________________________
```

Hidden units The above model has two intermediate or 'hidden' layers, between the input and output. The number of outputs (units, nodes, or neurons) is the dimension of the representational space for the layer. In other words, the amount of freedom the network is allowed when learning an internal representation.

If a model has more hidden units (a higher-dimensional representation space), and/or more layers, then the network can learn more complex representations. However, it makes the network more computationally expensive and may lead to learning unwanted patterns – patterns that improve performance on training data but not on the test data. This is called 'overfitting', and you might

have explored it before.

Step 8: Loss function and optimiser A model needs a loss function and an optimizer for training. Since this is a binary classification problem and the model outputs a probability (a single-unit layer with a sigmoid activation), we'll use the binary_crossentropy loss function.

This isn't the only choice for a loss function, you could, for instance, choose mean_squared_error. But, generally, binary_crossentropy is better for dealing with probabilities—it measures the "distance" between probability distributions, or in this case, between the ground-truth distribution and the predictions. If you are exploring regression problems (say, to predict the price of a house), you may learn how to use another loss function called mean squared error.

[13]:
```
# configure the model to use a loss function and optimiser
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['acc'])
```

Step 9: Create a validation set When training, you want to check the accuracy of the model on data it hasn't seen before. Create a validation set by setting apart 10,000 examples from the original training data.

[14]:
```
x_val = train_data[:10000]
partial_x_train = train_data[10000:]

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

Step 10: Train the model Train the model for 40 epochs in mini-batches of 512 samples. This is 40 iterations over all samples in the x_train and y_train tensors. While training, monitor the model's loss and accuracy on the 10,000 samples from the validation set.

[15]:
```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=40,
                    batch_size=512,
                    validation_data=(x_val, y_val),
                    verbose=1)
```

```
Epoch 1/40
30/30 [==============================] - 3s 46ms/step - loss: 0.6922 - acc:
0.5713 - val_loss: 0.6881 - val_acc: 0.6639
Epoch 2/40
30/30 [==============================] - 1s 19ms/step - loss: 0.6853 - acc:
0.7164 - val_loss: 0.6779 - val_acc: 0.7507
Epoch 3/40
30/30 [==============================] - 1s 19ms/step - loss: 0.6722 - acc:
0.7512 - val_loss: 0.6597 - val_acc: 0.7528
Epoch 4/40
```

```
30/30 [==============================] - 1s 19ms/step - loss: 0.6505 - acc:
0.7720 - val_loss: 0.6329 - val_acc: 0.7715
Epoch 5/40
30/30 [==============================] - 1s 19ms/step - loss: 0.6175 - acc:
0.7940 - val_loss: 0.5970 - val_acc: 0.7905
Epoch 6/40
30/30 [==============================] - 1s 19ms/step - loss: 0.5753 - acc:
0.8170 - val_loss: 0.5565 - val_acc: 0.8063
Epoch 7/40
30/30 [==============================] - 1s 19ms/step - loss: 0.5308 - acc:
0.8298 - val_loss: 0.5156 - val_acc: 0.8211
Epoch 8/40
30/30 [==============================] - 1s 19ms/step - loss: 0.4865 - acc:
0.8452 - val_loss: 0.4754 - val_acc: 0.8344
Epoch 9/40
30/30 [==============================] - 1s 19ms/step - loss: 0.4437 - acc:
0.8584 - val_loss: 0.4409 - val_acc: 0.8439
Epoch 10/40
30/30 [==============================] - 1s 20ms/step - loss: 0.4006 - acc:
0.8783 - val_loss: 0.4112 - val_acc: 0.8503
Epoch 11/40
30/30 [==============================] - 1s 20ms/step - loss: 0.3711 - acc:
0.8801 - val_loss: 0.3866 - val_acc: 0.8573
Epoch 12/40
30/30 [==============================] - 1s 20ms/step - loss: 0.3462 - acc:
0.8847 - val_loss: 0.3670 - val_acc: 0.8627
Epoch 13/40
30/30 [==============================] - 1s 19ms/step - loss: 0.3164 - acc:
0.8941 - val_loss: 0.3514 - val_acc: 0.8660
Epoch 14/40
30/30 [==============================] - 1s 19ms/step - loss: 0.2990 - acc:
0.9004 - val_loss: 0.3376 - val_acc: 0.8719
Epoch 15/40
30/30 [==============================] - 1s 19ms/step - loss: 0.2776 - acc:
0.9079 - val_loss: 0.3268 - val_acc: 0.8750
Epoch 16/40
30/30 [==============================] - 1s 19ms/step - loss: 0.2601 - acc:
0.9102 - val_loss: 0.3176 - val_acc: 0.8786
Epoch 17/40
30/30 [==============================] - 1s 19ms/step - loss: 0.2405 - acc:
0.9211 - val_loss: 0.3103 - val_acc: 0.8790
Epoch 18/40
30/30 [==============================] - 1s 19ms/step - loss: 0.2330 - acc:
0.9207 - val_loss: 0.3050 - val_acc: 0.8783
Epoch 19/40
30/30 [==============================] - 1s 19ms/step - loss: 0.2257 - acc:
0.9210 - val_loss: 0.2997 - val_acc: 0.8805
Epoch 20/40
```

```
30/30 [==============================] - 1s 19ms/step - loss: 0.2128 - acc:
0.9286 - val_loss: 0.2958 - val_acc: 0.8836
Epoch 21/40
30/30 [==============================] - 1s 19ms/step - loss: 0.2043 - acc:
0.9313 - val_loss: 0.2926 - val_acc: 0.8828
Epoch 22/40
30/30 [==============================] - 1s 21ms/step - loss: 0.1907 - acc:
0.9377 - val_loss: 0.2902 - val_acc: 0.8839
Epoch 23/40
30/30 [==============================] - 1s 19ms/step - loss: 0.1830 - acc:
0.9401 - val_loss: 0.2882 - val_acc: 0.8844
Epoch 24/40
30/30 [==============================] - 1s 19ms/step - loss: 0.1755 - acc:
0.9433 - val_loss: 0.2870 - val_acc: 0.8846
Epoch 25/40
30/30 [==============================] - 1s 19ms/step - loss: 0.1759 - acc:
0.9442 - val_loss: 0.2862 - val_acc: 0.8850
Epoch 26/40
30/30 [==============================] - 1s 19ms/step - loss: 0.1628 - acc:
0.9513 - val_loss: 0.2862 - val_acc: 0.8854
Epoch 27/40
30/30 [==============================] - 1s 19ms/step - loss: 0.1508 - acc:
0.9537 - val_loss: 0.2860 - val_acc: 0.8849
Epoch 28/40
30/30 [==============================] - 1s 19ms/step - loss: 0.1468 - acc:
0.9552 - val_loss: 0.2873 - val_acc: 0.8855
Epoch 29/40
30/30 [==============================] - 1s 19ms/step - loss: 0.1387 - acc:
0.9592 - val_loss: 0.2879 - val_acc: 0.8860
Epoch 30/40
30/30 [==============================] - 1s 19ms/step - loss: 0.1367 - acc:
0.9598 - val_loss: 0.2893 - val_acc: 0.8846
Epoch 31/40
30/30 [==============================] - 1s 19ms/step - loss: 0.1343 - acc:
0.9593 - val_loss: 0.2900 - val_acc: 0.8860
Epoch 32/40
30/30 [==============================] - 1s 19ms/step - loss: 0.1291 - acc:
0.9622 - val_loss: 0.2915 - val_acc: 0.8866
Epoch 33/40
30/30 [==============================] - 1s 18ms/step - loss: 0.1193 - acc:
0.9666 - val_loss: 0.2939 - val_acc: 0.8845
Epoch 34/40
30/30 [==============================] - 1s 19ms/step - loss: 0.1214 - acc:
0.9665 - val_loss: 0.2964 - val_acc: 0.8853
Epoch 35/40
30/30 [==============================] - 1s 19ms/step - loss: 0.1117 - acc:
0.9674 - val_loss: 0.2980 - val_acc: 0.8845
Epoch 36/40
```

```
30/30 [==============================] - 1s 19ms/step - loss: 0.1076 - acc:
0.9705 - val_loss: 0.3013 - val_acc: 0.8850
Epoch 37/40
30/30 [==============================] - 1s 20ms/step - loss: 0.1000 - acc:
0.9727 - val_loss: 0.3030 - val_acc: 0.8842
Epoch 38/40
30/30 [==============================] - 1s 20ms/step - loss: 0.0998 - acc:
0.9746 - val_loss: 0.3068 - val_acc: 0.8833
Epoch 39/40
30/30 [==============================] - 1s 20ms/step - loss: 0.0938 - acc:
0.9758 - val_loss: 0.3095 - val_acc: 0.8825
Epoch 40/40
30/30 [==============================] - 1s 19ms/step - loss: 0.0927 - acc:
0.9761 - val_loss: 0.3153 - val_acc: 0.8811
```

Step 11: Evaluate the model Let's see how the model performs. Two values will be returned: loss (a number which represents our error, lower values are better), and accuracy.

This fairly naive approach achieves an accuracy of about 87 per cent. With more advanced approaches, the model should get closer to 95 per cent.

[16]:
```
results = model.evaluate(test_data, test_labels)
print(results)
```

```
782/782 [==============================] - 2s 3ms/step - loss: 0.3383 - acc:
0.8693
[0.33829477429389954, 0.8693199753761292]
```

Step 12: Plot accuracy and loss over time model.fit() returns a history object that contains a dictionary with everything that happened during training.

[17]:
```
history_dict = history.history
history_dict.keys()
```

[17]:
```
dict_keys(['loss', 'acc', 'val_loss', 'val_acc'])
```

There are four entries: one for each monitored metric during training and validation. The val_acc, acc, val_loss and loss parameters can be used to plot the training and validation loss for comparison, as well as the training and validation accuracy.

With the increment of epochs the model gets lower loss function value, which means model overfitting could become an issue.

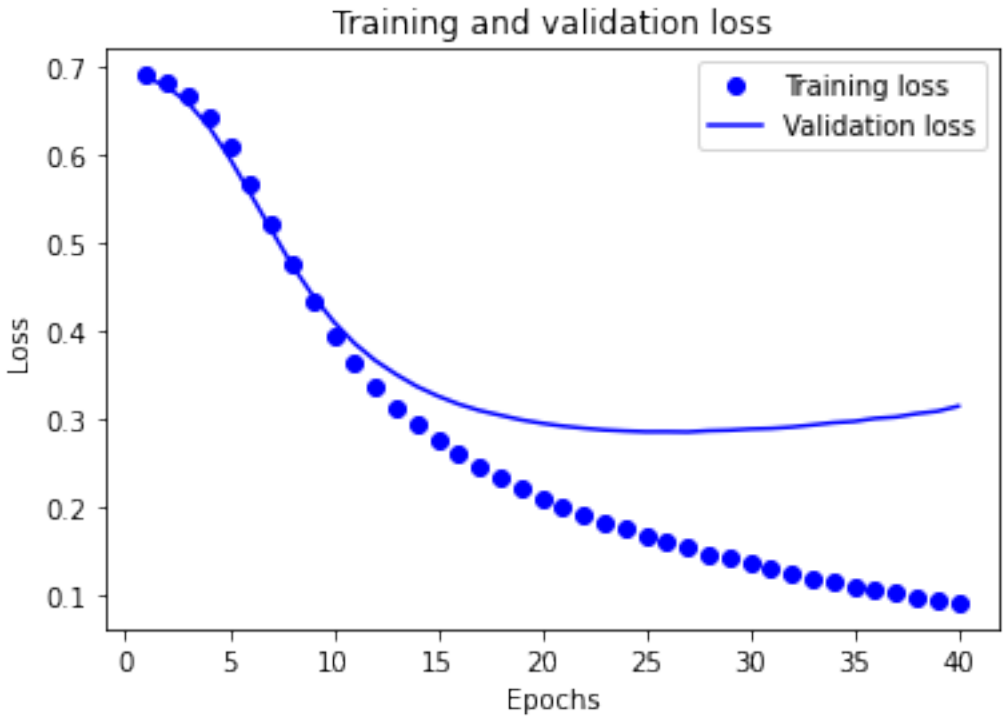
[18]:
```
import matplotlib.pyplot as plt

acc = history_dict['acc']
val_acc = history_dict['val_acc']
loss = history_dict['loss']
val_loss = history_dict['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 this plot, the dots represent the training loss and accuracy, and the solid lines are the validation loss and accuracy.

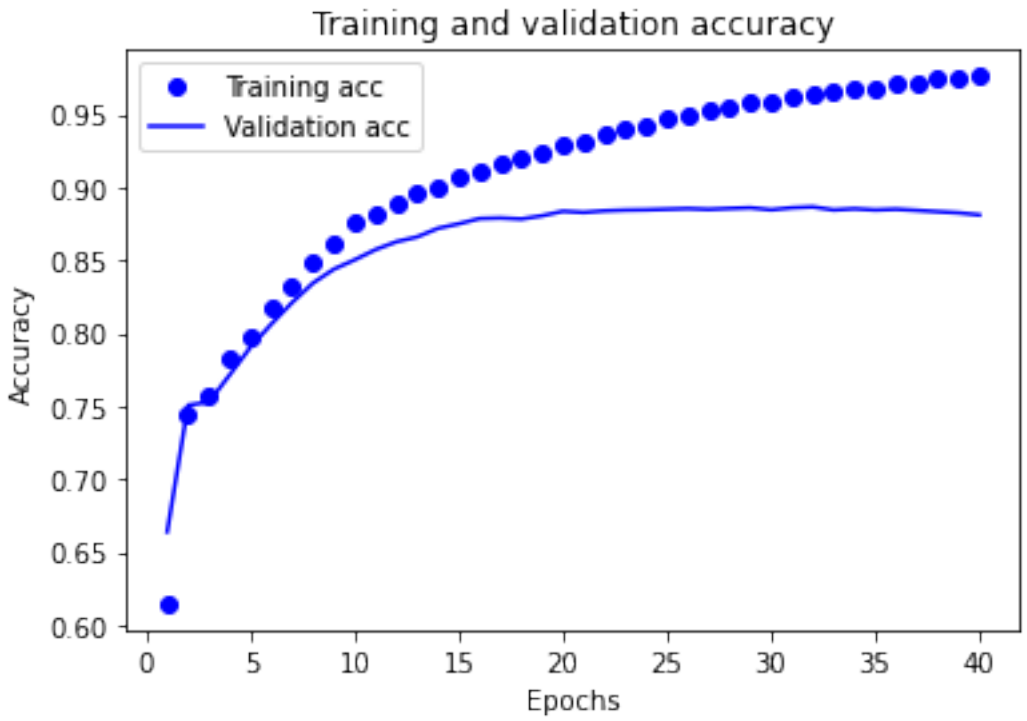
Notice the training loss decreases with each epoch and the training accuracy increases with each epoch. This is expected when using a gradient descent optimization—it should minimize the desired quantity on every iteration.

This isn't the case for the validation loss and accuracy—they seem to peak after about twenty epochs. This is an example of overfitting: the model performs better on the training data than it does on data it has never seen before. After this point, the model over-optimizes and learns representations specific to the training data that do not generalize to test data.

For this particular case, you could prevent overfitting by simply stopping the training after twenty or so epochs. This can be done automatically with a callback.

[19]:
```
plt.clf()   # clear figure

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()
```



[ ]: