Deep Learning

Practical 2a - Classifying Movie Reviews

AY2020/21 Semester

In [1]:
```
from tensorflow import keras
print('keras: ', keras.__version__)
```

```
keras:  2.2.4-tf
```

In [2]:
```
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:95% !important; }</style>"))
```

Objectives ¶

After completing this practical exercise, students should be able to:

1. Build a neural network model to classify movie reviews
2. Exercise- tuning several model parameters

1. Classifying movie reviews (a binary classification example)

In this example, we will learn to classify movie reviews into "positive" reviews and "negative" reviews, just based on the text content of the reviews.

1.1 The IMDB dataset

The IMDB dataset has 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 IMDB dataset comes packaged with Keras. It has already been preprocessed: the reviews (sequences of words) have been turned into sequences of integers, where each integer stands for a specific word in a dictionary.

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 [3]:
```
from tensorflow.keras.datasets import imdb

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

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 [4]:
```
import numpy as np

print(train_data[100])
print('\nthe length of this training sample is: ',len(train_data[100]))
```

```
[1, 13, 244, 6, 87, 337, 7, 628, 2219, 5, 28, 285, 15, 240, 93, 23, 288, 549, 18, 1455, 673, 4, 241, 534, 363
5, 8448, 20, 38, 54, 13, 258, 46, 44, 14, 13, 1241, 7258, 12, 5, 5, 51, 9, 14, 45, 6, 762, 7, 2, 1309, 328, 5,
428, 2473, 15, 26, 1292, 5, 3939, 6728, 5, 1960, 279, 13, 92, 124, 803, 52, 21, 279, 14, 9, 43, 6, 762, 7, 59
5, 15, 16, 2, 23, 4, 1071, 467, 4, 403, 7, 628, 2219, 8, 97, 6, 171, 3596, 99, 387, 72, 97, 12, 788, 15, 13, 1
61, 459, 44, 4, 3939, 1101, 173, 21, 69, 8, 401, 2, 4, 481, 88, 61, 4731, 238, 28, 32, 11, 32, 14, 9, 6, 545,
1332, 766, 5, 203, 73, 28, 43, 77, 317, 11, 4, 2, 953, 270, 17, 6, 3616, 13, 545, 386, 25, 92, 1142, 129, 278,
23, 14, 241, 46, 7, 158]

the length of this training sample is:  158
```

In [5]:
```
train_labels[100]
```

Out[5]:
```
0
```

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

In [6]:
```
max(np.array([max(sequence) for sequence in train_data]))
```

Out[6]:
```
9999
```

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

In [7]:
```
# 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.items()])
# 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", and "unknown".
decoded_review = ' '.join([reverse_word_index.get(i - 3, '?') for i in train_data[100]])
```

In [8]:
```
decoded_review
```

Out[8]:
```
"? i am a great fan of david lynch and have everything that he's made on dvd except for hotel room the 2 hour
twin peaks movie so when i found out about this i immediately grabbed it and and what is this it's a bunch of
? drawn black and white cartoons that are loud and foul mouthed and unfunny maybe i don't know what's good but
maybe this is just a bunch of crap that was ? on the public under the name of david lynch to make a few bucks
too let me make it clear that i didn't care about the foul language part but had to keep ? the sound because m
y neighbors might have all in all this is a highly disappointing release and may well have just been left in t
he ? box set as a curiosity i highly recommend you don't spend your money on this 2 out of 10"
```

1.2 Preparing the data

We cannot feed lists of integers into a neural network. We have to turn our lists into tensors. We use one-hot-encode to turn our lists into vectors of 0s and 1s. For example, 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. Let's vectorize our data, which we will do manually for maximum clarity:

In [9]:
```
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 1s
    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 [10]:
```
x_train.shape
```

Out[10]:
```
(25000, 10000)
```

In [11]:
```
x_train[100]
```

Out[11]:
```
array([0., 1., 1., ..., 0., 0., 0.])
```

In [12]:
```
print('train_data[100] is: \n', train_data[100])
print('\n After one-hot-encode, train_data[100] is encoded to x_train[100]: \n', x_train[100])
print('\n The index of non-zero elements in x_train[100] are : \n', np.nonzero(x_train[100]))
```

```
train_data[100] is:
 [1, 13, 244, 6, 87, 337, 7, 628, 2219, 5, 28, 285, 15, 240, 93, 23, 288, 549, 18, 1455, 673, 4, 241, 534,
3635, 8448, 20, 38, 54, 13, 258, 46, 44, 14, 13, 1241, 7258, 12, 5, 5, 51, 9, 14, 45, 6, 762, 7, 2, 1309,
328, 5, 428, 2473, 15, 26, 1292, 5, 3939, 6728, 5, 1960, 279, 13, 92, 124, 803, 52, 21, 279, 14, 9, 43, 6,
762, 7, 595, 15, 16, 2, 23, 4, 1071, 467, 4, 403, 7, 628, 2219, 8, 97, 6, 171, 3596, 99, 387, 72, 97, 12,
788, 15, 13, 161, 459, 44, 4, 3939, 1101, 173, 21, 69, 8, 401, 2, 4, 481, 88, 61, 4731, 238, 28, 32, 11, 3
2, 14, 9, 6, 545, 1332, 766, 5, 203, 73, 28, 43, 77, 317, 11, 4, 2, 953, 270, 17, 6, 3616, 13, 545, 386, 2
5, 92, 1142, 129, 278, 23, 14, 241, 46, 7, 158]

 After one-hot-encode, train_data[100] is encoded to x_train[100]:
 [0. 1. 1. ... 0. 0. 0.]

 The index of non-zero elements in x_train[100] are :
 (array([   1,    2,    4,    5,    6,    7,    8,    9,   11,   12,   13,
         14,   15,   16,   17,   18,   20,   21,   23,   25,   26,   28,
         32,   38,   43,   44,   45,   46,   51,   52,   54,   61,   69,
         72,   73,   77,   87,   88,   92,   93,   97,   99,  124,  129,
        158,  161,  171,  173,  203,  238,  240,  241,  244,  258,  270,
        278,  279,  285,  288,  317,  328,  337,  386,  387,  401,  403,
```

We should also vectorize our labels, which is straightforward:

In [13]:
```
train_labels[100]
```

Out[13]:
```
0
```

In [14]:
```
# Our vectorized labels
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

In [15]:
```
y_train[100]
```

Out[15]:
```
0.0
```

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

1.3 Building our network

Our input data is simply vectors, and our labels are scalars (1s and 0s). Let's implement it in Keras.

In [16]:
```
from tensorflow.keras import models
from tensorflow.keras import 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(1, activation='sigmoid'))
```

In [17]:
```
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 configure our model with the `rmsprop` optimizer and the `binary_crossentropy` loss function. Note that we will also monitor accuracy during training.

In [18]:
```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

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.

1.4 Validating our approach

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 [19]:
```
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, in mini-batches of 512 samples. At 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 [20]:
```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

```
Train on 15000 samples, validate on 10000 samples
Epoch 1/20
15000/15000 [==============================] - 4s 277us/sample - loss: 0.5178 - accuracy: 0.7959 - val_los
s: 0.4149 - val_accuracy: 0.8475
Epoch 2/20
15000/15000 [==============================] - 2s 145us/sample - loss: 0.3201 - accuracy: 0.9006 - val_los
s: 0.3216 - val_accuracy: 0.8761
Epoch 3/20
15000/15000 [==============================] - 2s 137us/sample - loss: 0.2390 - accuracy: 0.9240 - val_los
s: 0.2896 - val_accuracy: 0.8870
Epoch 4/20
15000/15000 [==============================] - 2s 144us/sample - loss: 0.1888 - accuracy: 0.9399 - val_los
s: 0.2823 - val_accuracy: 0.8872
Epoch 5/20
15000/15000 [==============================] - 2s 139us/sample - loss: 0.1533 - accuracy: 0.9519 - val_los
s: 0.2745 - val_accuracy: 0.8886
Epoch 6/20
15000/15000 [==============================] - 2s 155us/sample - loss: 0.1235 - accuracy: 0.9633 - val_los
s: 0.2919 - val_accuracy: 0.8838
```

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:

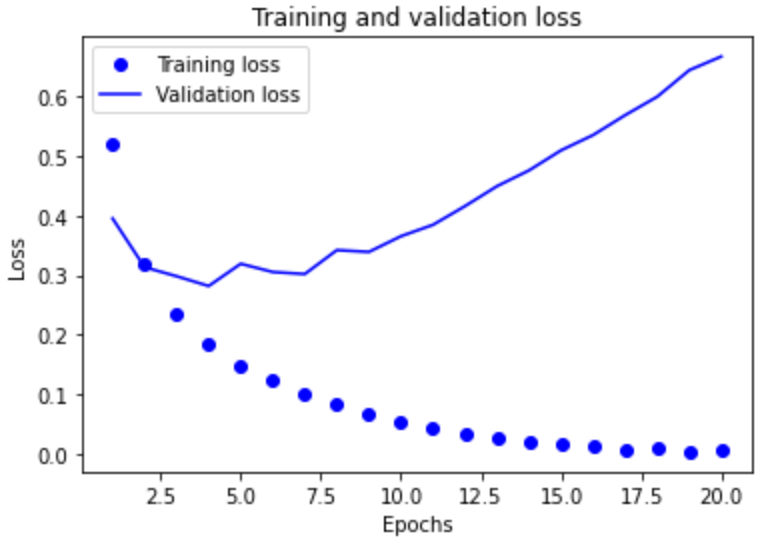
In [21]:
```
history_dict = history.history
history_dict.keys()
```

Out[21]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

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:

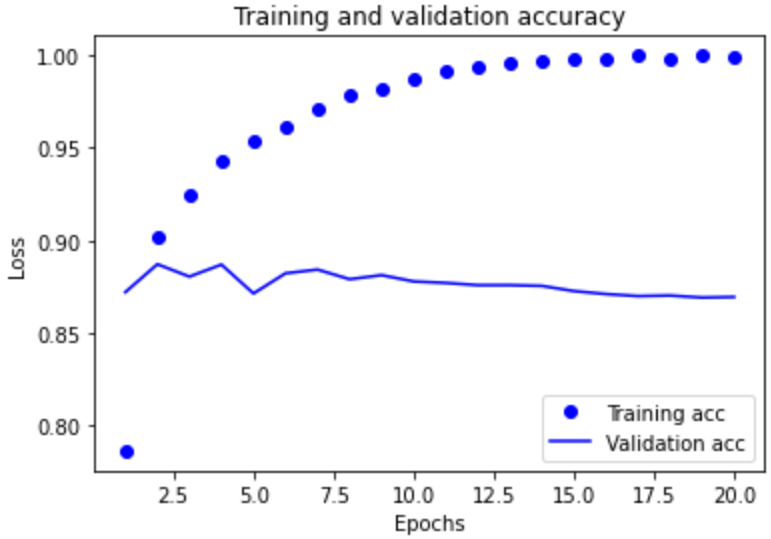
In [22]:

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

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
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 [23]:

```
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('Loss')
plt.legend()

plt.show()
```



The dots are the training loss and accuracy, while the solid lines are the validation loss and accuracy. 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. 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. 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. Let's train a new network from scratch for four epochs, then evaluate it on our test data:

In [24]:
```
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)
```

```
Train on 25000 samples
Epoch 1/4
25000/25000 [==============================] - 5s 200us/sample - loss: 0.5187 - accuracy: 0.7798
Epoch 2/4
25000/25000 [==============================] - 3s 127us/sample - loss: 0.3013 - accuracy: 0.9027
Epoch 3/4
25000/25000 [==============================] - 3s 118us/sample - loss: 0.2215 - accuracy: 0.9254
Epoch 4/4
25000/25000 [==============================] - 3s 128us/sample - loss: 0.1788 - accuracy: 0.9388
```

Out[24]:
```
<tensorflow.python.keras.callbacks.History at 0x20e4f479b48>
```

In [25]:
```
results = model.evaluate(x_test, y_test, verbose=2)
results # returns the loss value and accuracy
```

```
25000/1 - 4s - loss: 0.2762 - accuracy: 0.8856
```

Out[25]:
```
[0.2861841391658783, 0.88564]
```

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

1.5 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 [26]:
```
x_test
```

Out[26]:
```
array([[0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 1., ..., 0., 0., 0.],
       ...,
       [0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 1., ..., 0., 0., 0.]])
```

In [27]:
```
model.predict(x_test)
```

Out[27]:
```
array([[0.16669199],
       [0.99419415],
       [0.87037385],
       ...,
       [0.13370472],
       [0.08534437],
       [0.52047276]], 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).

2. Exercise - tuning model parameters

Please try below two scenerios, make changes on the model, train the models for 20 epochs (keep the same optimizer, loss, metrics and batch size).

Scenerio A:

- In the first two layers, change activation function from "relu" to "sigmoid".

Scenerio B:

- Remove one hidden layer
- Only use 2 units in the first layer

Observe the training and validation loss & accuracy curves for both scenerios.

Provide your codes & comments in the below boxes.

2.1 Scenerio A

In [28]:

```
#Task 1: Build the model and change activation function from "relu" to "sigmoid"
modelA = models.Sequential()
modelA.add(layers.Dense(16, activation='sigmoid', input_shape=(10000,)))
modelA.add(layers.Dense(16, activation='sigmoid'))
modelA.add(layers.Dense(1, activation='sigmoid'))
modelA.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 16)                160016
_________________________________________________________________
dense_7 (Dense)              (None, 16)                272
_________________________________________________________________
dense_8 (Dense)              (None, 1)                 17
=================================================================
Total params: 160,305
Trainable params: 160,305
Non-trainable params: 0
_________________________________________________________________
```

In [29]:

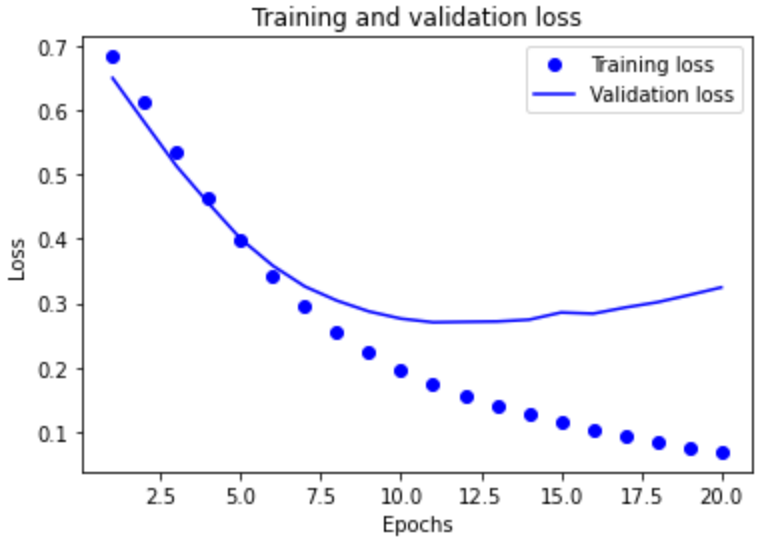
```
#Task 2: Compile and Fit the model
modelA.compile(optimizer='rmsprop',
               loss='binary_crossentropy',
               metrics=['accuracy'])
history = modelA.fit(partial_x_train,
                     partial_y_train,
                     epochs=20,
                     batch_size=512,
                     validation_data=(x_val, y_val))
```

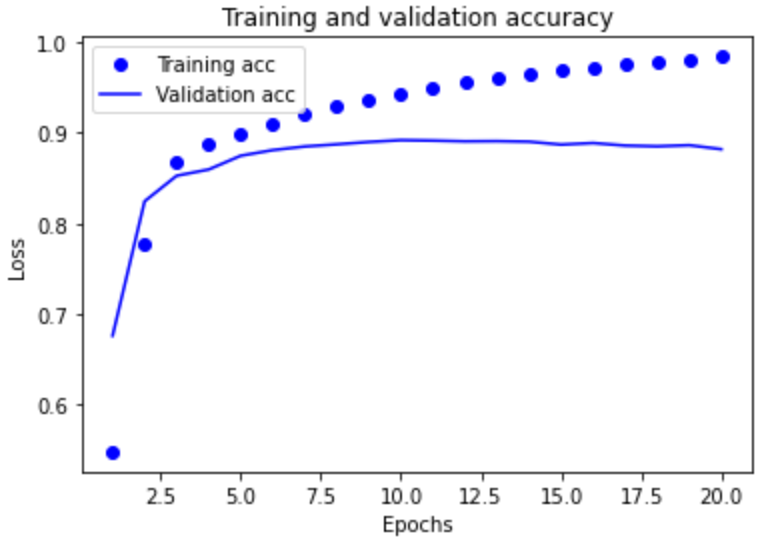
```
Train on 15000 samples, validate on 10000 samples
Epoch 1/20
15000/15000 [==============================] - 5s 345us/sample - loss: 0.6828 - accuracy: 0.5478 - val_loss:
0.6494 - val_accuracy: 0.6760
Epoch 2/20
15000/15000 [==============================] - 3s 193us/sample - loss: 0.6107 - accuracy: 0.7769 - val_loss:
0.5804 - val_accuracy: 0.8244
Epoch 3/20
15000/15000 [==============================] - 3s 193us/sample - loss: 0.5352 - accuracy: 0.8675 - val_loss:
0.5125 - val_accuracy: 0.8526
Epoch 4/20
15000/15000 [==============================] - 3s 209us/sample - loss: 0.4628 - accuracy: 0.8866 - val_loss:
0.4547 - val_accuracy: 0.8595
Epoch 5/20
15000/15000 [==============================] - 3s 192us/sample - loss: 0.3983 - accuracy: 0.8981 - val_loss:
0.3997 - val_accuracy: 0.8747
Epoch 6/20
15000/15000 [==============================] - 3s 182us/sample - loss: 0.3419 - accuracy: 0.9105 - val_loss:
0.3582 - val_accuracy: 0.8809
Epoch 7/20
15000/15000 [==============================] - 3s 180us/sample - loss: 0.2948 - accuracy: 0.9209 - val_loss:
0.3260 - val_accuracy: 0.8849
Epoch 8/20
15000/15000 [==============================] - 3s 185us/sample - loss: 0.2552 - accuracy: 0.9295 - val_loss:
0.3040 - val_accuracy: 0.8873
Epoch 9/20
15000/15000 [==============================] - 3s 183us/sample - loss: 0.2230 - accuracy: 0.9365 - val_loss:
0.2872 - val_accuracy: 0.8899
Epoch 10/20
15000/15000 [==============================] - 3s 204us/sample - loss: 0.1964 - accuracy: 0.9432 - val_loss:
0.2760 - val_accuracy: 0.8919
Epoch 11/20
15000/15000 [==============================] - 3s 189us/sample - loss: 0.1746 - accuracy: 0.9502 - val_loss:
0.2703 - val_accuracy: 0.8914
Epoch 12/20
15000/15000 [==============================] - 3s 188us/sample - loss: 0.1558 - accuracy: 0.9550 - val_loss:
```

```
0.2710 - val_accuracy: 0.8906
Epoch 13/20
15000/15000 [==============================] - 3s 185us/sample - loss: 0.1405 - accuracy: 0.9600 - val_loss:
0.2716 - val_accuracy: 0.8908
Epoch 14/20
15000/15000 [==============================] - 3s 188us/sample - loss: 0.1270 - accuracy: 0.9643 - val_loss:
0.2744 - val_accuracy: 0.8902
Epoch 15/20
15000/15000 [==============================] - 3s 197us/sample - loss: 0.1149 - accuracy: 0.9683 - val_loss:
0.2856 - val_accuracy: 0.8870
Epoch 16/20
15000/15000 [==============================] - 3s 207us/sample - loss: 0.1041 - accuracy: 0.9720 - val_loss:
0.2837 - val_accuracy: 0.8887
Epoch 17/20
15000/15000 [==============================] - 3s 187us/sample - loss: 0.0940 - accuracy: 0.9762 - val_loss:
0.2933 - val_accuracy: 0.8859
Epoch 18/20
15000/15000 [==============================] - 3s 205us/sample - loss: 0.0847 - accuracy: 0.9789 - val_loss:
0.3014 - val_accuracy: 0.8851
Epoch 19/20
15000/15000 [==============================] - 3s 196us/sample - loss: 0.0767 - accuracy: 0.9809 - val_loss:
0.3127 - val_accuracy: 0.8863
Epoch 20/20
15000/15000 [==============================] - 3s 211us/sample - loss: 0.0693 - accuracy: 0.9843 - val_loss:
0.3245 - val_accuracy: 0.8819
```

In [30]:

```
#Task 3: Plot the loss (train & test) and accuracy (train & test) curves
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
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()

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('Loss')
plt.legend()

plt.show()
```





Task 4: Comments We can see the loss value is descreasing slowly and the accuracy is increasing slowly. Model only starts to overfitting at a later stage.

2.2 Scenerio B

In [31]:

```
#Task 1: Build the model, remove one hidden layer and Only use 2 units in the first layer
modelB = models.Sequential()
modelB.add(layers.Dense(2, activation='relu', input_shape=(10000,)))
modelB.add(layers.Dense(1, activation='sigmoid'))
modelB.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_9 (Dense)              (None, 2)                 20002
_________________________________________________________________
dense_10 (Dense)             (None, 1)                 3
=================================================================
Total params: 20,005
Trainable params: 20,005
Non-trainable params: 0
_________________________________________________________________
```

In [32]:

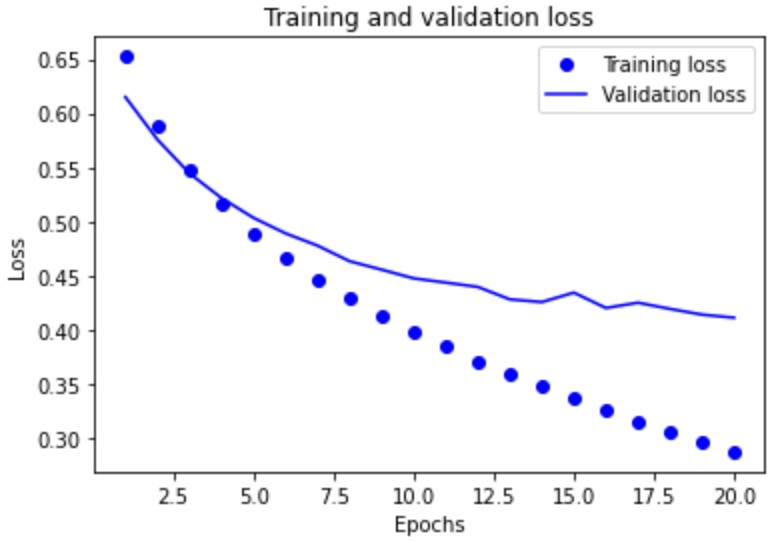
```
#Task 2: Compile and Fit the model
modelB.compile(optimizer='rmsprop',
               loss='binary_crossentropy',
               metrics=['accuracy'])
history = modelB.fit(partial_x_train,
                     partial_y_train,
                     epochs=20,
                     batch_size=512,
                     validation_data=(x_val, y_val))
```

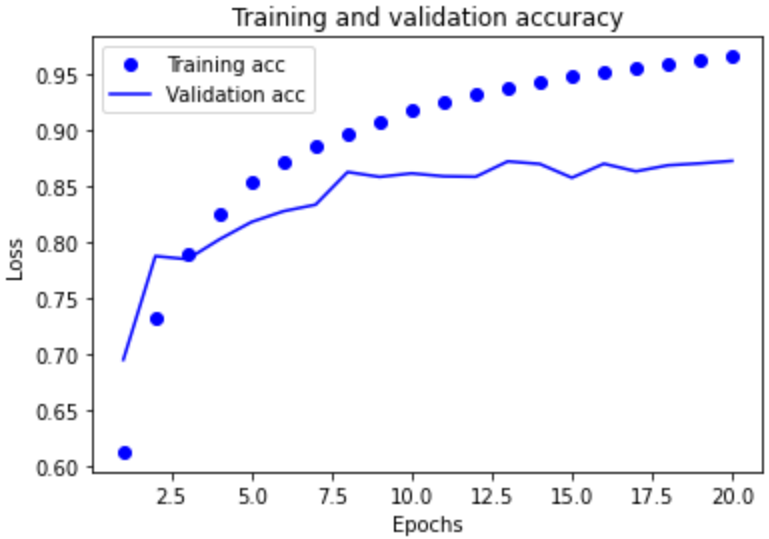
```
Train on 15000 samples, validate on 10000 samples
Epoch 1/20
15000/15000 [==============================] - 4s 280us/sample - loss: 0.6524 - accuracy: 0.6131 - val_loss:
0.6150 - val_accuracy: 0.6952
Epoch 2/20
15000/15000 [==============================] - 3s 172us/sample - loss: 0.5884 - accuracy: 0.7318 - val_loss:
0.5758 - val_accuracy: 0.7875
Epoch 3/20
15000/15000 [==============================] - 3s 189us/sample - loss: 0.5480 - accuracy: 0.7901 - val_loss:
0.5446 - val_accuracy: 0.7847
Epoch 4/20
15000/15000 [==============================] - 3s 203us/sample - loss: 0.5160 - accuracy: 0.8244 - val_loss:
0.5218 - val_accuracy: 0.8024
Epoch 5/20
15000/15000 [==============================] - 3s 194us/sample - loss: 0.4895 - accuracy: 0.8527 - val_loss:
0.5035 - val_accuracy: 0.8179
Epoch 6/20
15000/15000 [==============================] - 3s 181us/sample - loss: 0.4669 - accuracy: 0.8705 - val_loss:
0.4892 - val_accuracy: 0.8275
Epoch 7/20
15000/15000 [==============================] - 3s 168us/sample - loss: 0.4470 - accuracy: 0.8856 - val_loss:
0.4779 - val_accuracy: 0.8333
Epoch 8/20
15000/15000 [==============================] - 2s 166us/sample - loss: 0.4294 - accuracy: 0.8963 - val_loss:
0.4635 - val_accuracy: 0.8624
Epoch 9/20
15000/15000 [==============================] - 3s 174us/sample - loss: 0.4133 - accuracy: 0.9075 - val_loss:
0.4557 - val_accuracy: 0.8581
Epoch 10/20
15000/15000 [==============================] - 3s 167us/sample - loss: 0.3982 - accuracy: 0.9172 - val_loss:
0.4477 - val_accuracy: 0.8612
Epoch 11/20
15000/15000 [==============================] - 3s 181us/sample - loss: 0.3844 - accuracy: 0.9250 - val_loss:
0.4438 - val_accuracy: 0.8586
Epoch 12/20
```

```
15000/15000 [==============================] - 3s 188us/sample - loss: 0.3713 - accuracy: 0.9316 - val_loss:
0.4398 - val_accuracy: 0.8582
Epoch 13/20
15000/15000 [==============================] - 3s 167us/sample - loss: 0.3592 - accuracy: 0.9367 - val_loss:
0.4283 - val_accuracy: 0.8719
Epoch 14/20
15000/15000 [==============================] - 3s 177us/sample - loss: 0.3475 - accuracy: 0.9433 - val_loss:
0.4258 - val_accuracy: 0.8696
Epoch 15/20
15000/15000 [==============================] - 3s 172us/sample - loss: 0.3367 - accuracy: 0.9471 - val_loss:
0.4345 - val_accuracy: 0.8573
Epoch 16/20
15000/15000 [==============================] - 3s 179us/sample - loss: 0.3260 - accuracy: 0.9524 - val_loss:
0.4203 - val_accuracy: 0.8698
Epoch 17/20
15000/15000 [==============================] - 3s 186us/sample - loss: 0.3157 - accuracy: 0.9558 - val_loss:
0.4252 - val_accuracy: 0.8630
Epoch 18/20
15000/15000 [==============================] - 3s 185us/sample - loss: 0.3061 - accuracy: 0.9589 - val_loss:
0.4195 - val_accuracy: 0.8684
Epoch 19/20
15000/15000 [==============================] - 3s 169us/sample - loss: 0.2968 - accuracy: 0.9629 - val_loss:
0.4142 - val_accuracy: 0.8701
Epoch 20/20
15000/15000 [==============================] - 3s 167us/sample - loss: 0.2876 - accuracy: 0.9653 - val_loss:
0.4114 - val_accuracy: 0.8723
```

In [33]:

```
#Task 3: Plot the loss (train & test) and accuracy (train & test) curves
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
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()

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('Loss')
plt.legend()

plt.show()
```





Task 4: Comments We can see the loss value is descreasing slowly and the accuracy is increasing slowly. Model only starts to overfitting at a later stage.

In [ ]: