# 2.4. Regularizing and optimizing neural networks

- We can remember that regularization is a suite of strategies used to reduce the generalization error of machine learning models, in particular deep learning models.
- Typical measures for regularizing neural networks are: dropout, batch normalization, weight regularization and initialization.

## 2.4.1. IMDB dataset

- The IMDB (Internet Movie Data Base) dataset is a set of reviews on different movies collected from the popular internet site.
- There are 25,000 reviews of which half will be used for training and half for testing. Both halves are composed of 50% positive reviews and 50% negative reviews.
- The data consists of a set of words (the criticism itself), coded as integers (each different word is assigned a number) and a label indicating whether the criticism is positive or negative.
- The data is included in Keras so it is easy to load and use.

In [1]:
```
import tensorflow as tf
from tensorflow.keras.datasets import imdb

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

- The parameter `num_words = 10000` indicates that we will keep the 10,000 most frequent words used in the training set, the rest will be ignored. This allows the data to have a more manageable size.
- Each entry in the training set is a list of integers representing the words available in the critique.

In [2]:
```
print(train_data[0])
```

```
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 17
3, 36, 256, 5, 25, 100, 43, 838, 112, 50, 670, 2, 9, 35, 480, 284, 5, 15
0, 4, 172, 112, 167, 2, 336, 385, 39, 4, 172, 4536, 1111, 17, 546, 38, 1
3, 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, 1
2, 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, 10
7, 117, 5952, 15, 256, 4, 2, 7, 3766, 5, 723, 36, 71, 43, 530, 476, 26, 4
00, 317, 46, 7, 4, 2, 1029, 13, 104, 88, 4, 381, 15, 297, 98, 32, 2071, 5
6, 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]
```

- Each case is labeled as a zero or a one indicating a negative or positive criticism respectively.

In [3]:
```
train_labels[0]
```

Out[3]:
```
1
```

- We can take a look at the original review with this little code that converts the numerical indexes back into words (assigning `?` to the word we have not considered because it is not among the 10.000 most common.

In [4]:
```
# word_index is a dictionary mapping words to integer index
word_index = imdb.get_word_index()

# Reverses it, mapping integer indices to words
reverse_word_index = dict([(value, key) for (key, value) in word_index.ite

# Decodes de review. Note that the indices are offset by 3 because 0, 1, a
# indices por "padding", "start of sequence" and "unknown"
decoded_review = ' '.join([reverse_word_index.get(i - 3, '?') for i in tra
```

In [5]:
```
decoded_review
```

Out[5]:
```
"? 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 ? is an amazing actor and now the same being director
? 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 t
he film were great it was just brilliant so much that i bought the film a
s soon as it was released for ? and would recommend it to everyone to wat
ch 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 an
d this definitely was also ? to the two little boy's that played the ? of
norman and paul they were just brilliant children are often left out of t
he ? 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 shoul
d 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 s
hared with us all"
```

**IMDB: Preparing the data**

- In order to feed the neural network we need to convert the list of integers into a more manageable representation.
- As each review will be composed of words whose indexes range from zero to 9,999, it is easiest to encode our review as an array of 10,000 positions (from 0 to 9,999). In each position there will be a 1 if the word is present in the critique or a zero otherwise.
- This is called a *multi-hot encoding*.
- For example, a criticism that is the sequence of only two words `[3, 5]` will be encoded as an array of 10 000 positions in which only positions `3` and `5` store a one, the rest will be zeros.
- The 1s and 0s are represented as floating values since neural networks are fed with floating values (for various reasons, including efficiency in making the various calculations).

In [6]:
```
import numpy as np
def vectorize_sequences(sequences, dimension = 10000):
    # Creates an all-zero matrix of shape (len(sequences), dimension)
    results = np.zeros((len(sequences), dimension))

    for i, sequence in enumerate(sequences):
        # Sets specific indices of results[i] to 1s
        results[i, sequence] = 1.
    return results
```

- Let's see how `vectorized_sequences()` works with a simple example.
- We have the array `a` which we can understand as three lists of words identified by numbers.

In [7]:
```
a = [[1,2,3],
     [4,5,6],
     [7,8,9]]
```

- Therefore the end result of calling `vectorize_sequences()` over `a` is:

In [8]:
```
result = vectorize_sequences(a, 10)
print(result)
```

```
[[0. 1. 1. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 1. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]]
```

- Let us now apply the `vectorize_sequences` function to our training and test data.

In [9]:
```
# Vectorized training data
x_train = vectorize_sequences(train_data)

# Vectorized test data
x_test = vectorize_sequences(test_data)

print(x_train[0])
```

```
[0. 1. 1. ... 0. 0. 0.]
```

- The labels must also be converted to `np.array` but in this case the conversion is simple.
- As we can see, the ones and zeros of the labels are also converted to float values.

In [10]:
```
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

**IMDB: Building the neural network**

- We will build the network using a sequential Keras model composed of a sequence of densely connected layers (all neurons in one layer connected to all neurons in the next).
- We will have an input layer with 10,000 entries (one for each possible word).
- There are two key architecture decisions to be made about such a stack of Dense layers: (1) How many layers to use and (2) How many units to choose for each layer.
- A large network has a bigger representational power but it is more prone to overfitting.
- Unfortunately, there is no magical formula to determine the right number of layers or the right size for each layer.
- The general workflow for finding an appropriate model size is to start with relatively few layers and parameters, and increase the size of the layers or add new layers until you see diminishing returns with regard to validation loss.
- We decided to include two hidden layers with 16 neurons each with a `relu` activation function.
- Finally we will have an output layer with a single neuron. The activation of this neuron will be a `sigmoid` function that offers values between zero and one that indicate the probability that the case studied is a zero (negative review) or a one (positive review).

In [11]:
```
from tensorflow.keras import models
from tensorflow.keras import layers

# we are using the name parameter to identify better model and layers
model = models.Sequential(name='imdb')

# using the input_shappe parameter Keras will create an input layer to ins
model.add(layers.Dense(16, name='hidden_1', activation='relu', input_shape
model.add(layers.Dense(16, name='hidden_2',activation='relu'))
model.add(layers.Dense(1, name='output',  activation='sigmoid'))

model.summary()
```

```
Model: "imdb"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 hidden_1 (Dense)            (None, 16)                160016

 hidden_2 (Dense)            (None, 16)                272

 output (Dense)              (None, 1)                 17

=================================================================
Total params: 160305 (626.19 KB)
Trainable params: 160305 (626.19 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

- After creating the network structure we have to indicate the learning parameters that we are going to use.
- First of all we have to indicate an error function.
- As we are working with a model where the final output is a probability the most appropriate function is `binary_crossentropy` .
- We could also have applied `mean_squared_error` but as a general rule `binary_crossentropy` works better with outputs that represent probabilities.

- As an optimizing algorithm we will use `rmsprop` .
- As a validation measure in our test suite we will use `accuracy` .

In [12]:
```
model.compile(optimizer = 'rmsprop',
              loss = 'binary_crossentropy',
              metrics = ['accuracy'])
```

**IMDB with training, validation and test sets**

- To monitor the accuracy of the model during training on data that you have not seen before we will divide our training set ( `x_train` ) into two parts:
  - `x_val` : will be the first 10,000 values of the training set and `y_val` will be their corresponding labels, i.e., the validation set.
  - `partial_x_train` : will be the remaining 15,000 values and `partial_y_train` will be their corresponding labels, i.e., the new training set.

In [13]:
```
x_val = x_train[:10000] # Copy of x_train from the begining (0) to 9999
partial_x_train = x_train[10000:] # Copy of x_train from 10000 to the end

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

- Now we train the model using `partial_x_train` and `partial_y_train` as training data.
- We repeat the training with these data 20 times ( `epoch` ).
- The calculations with the loss function and the update of weights is done every 512 cases ( `batch_size` ).
- Finally the validation data are `x_val` and `y_val` and are passed to the `fit` method using the `validation_data` parameter.

In [14]:
```
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 28ms/step - loss: 0.5390 - ac
curacy: 0.7797 - val_loss: 0.4546 - val_accuracy: 0.8091
Epoch 2/20
30/30 [==============================] - 0s 11ms/step - loss: 0.3556 - ac
curacy: 0.8890 - val_loss: 0.3401 - val_accuracy: 0.8774
Epoch 3/20
30/30 [==============================] - 0s 11ms/step - loss: 0.2696 - ac
curacy: 0.9125 - val_loss: 0.2977 - val_accuracy: 0.8867
Epoch 4/20
30/30 [==============================] - 0s 11ms/step - loss: 0.2177 - ac
curacy: 0.9266 - val_loss: 0.2859 - val_accuracy: 0.8841
Epoch 5/20
30/30 [==============================] - 0s 11ms/step - loss: 0.1821 - ac
curacy: 0.9387 - val_loss: 0.2786 - val_accuracy: 0.8876
Epoch 6/20
30/30 [==============================] - 0s 11ms/step - loss: 0.1560 - ac
curacy: 0.9482 - val_loss: 0.2811 - val_accuracy: 0.8871
Epoch 7/20
30/30 [==============================] - 0s 11ms/step - loss: 0.1347 - ac
curacy: 0.9578 - val_loss: 0.3034 - val_accuracy: 0.8823
Epoch 8/20
30/30 [==============================] - 0s 11ms/step - loss: 0.1175 - ac
curacy: 0.9644 - val_loss: 0.2964 - val_accuracy: 0.8830
Epoch 9/20
30/30 [==============================] - 0s 11ms/step - loss: 0.1023 - ac
curacy: 0.9695 - val_loss: 0.3222 - val_accuracy: 0.8766
Epoch 10/20
30/30 [==============================] - 0s 11ms/step - loss: 0.0889 - ac
curacy: 0.9751 - val_loss: 0.3399 - val_accuracy: 0.8747
Epoch 11/20
30/30 [==============================] - 0s 11ms/step - loss: 0.0784 - ac
curacy: 0.9782 - val_loss: 0.3433 - val_accuracy: 0.8778
Epoch 12/20
30/30 [==============================] - 0s 11ms/step - loss: 0.0674 - ac
curacy: 0.9834 - val_loss: 0.3584 - val_accuracy: 0.8818
Epoch 13/20
30/30 [==============================] - 0s 11ms/step - loss: 0.0580 - ac
curacy: 0.9860 - val_loss: 0.3907 - val_accuracy: 0.8772
Epoch 14/20
30/30 [==============================] - 0s 11ms/step - loss: 0.0498 - ac
curacy: 0.9887 - val_loss: 0.3962 - val_accuracy: 0.8758
Epoch 15/20
30/30 [==============================] - 0s 12ms/step - loss: 0.0426 - ac
curacy: 0.9915 - val_loss: 0.4210 - val_accuracy: 0.8759
Epoch 16/20
30/30 [==============================] - 0s 11ms/step - loss: 0.0359 - ac
curacy: 0.9936 - val_loss: 0.4540 - val_accuracy: 0.8720
Epoch 17/20
30/30 [==============================] - 0s 12ms/step - loss: 0.0333 - ac
curacy: 0.9932 - val_loss: 0.4684 - val_accuracy: 0.8708
Epoch 18/20
30/30 [==============================] - 0s 11ms/step - loss: 0.0255 - ac
curacy: 0.9969 - val_loss: 0.5027 - val_accuracy: 0.8663
Epoch 19/20
30/30 [==============================] - 0s 12ms/step - loss: 0.0229 - ac
curacy: 0.9973 - val_loss: 0.5139 - val_accuracy: 0.8693
Epoch 20/20
30/30 [==============================] - 0s 11ms/step - loss: 0.0179 - ac
curacy: 0.9984 - val_loss: 0.5418 - val_accuracy: 0.8661
```

- We can use the `validation_split` parameter of the `fit` method for automatically divide the train test into train and validation
- The value passed is the percentage of training data reserved for validation

```
history = model.fit(x_train,                    y_train,
epochs = 20,                    batch_size = 512,
validation_split = 0.4)
```

- The `fit` method returns a `History` object.
- This object has a `history` property which is a dictionary containing the data of everything that has happened during the training.
- It has four keys: `loss`, `accuracy`, `val_loss`, `val_accuracy`.
- For each of these keys it has a list of the different values that the parameter has been taking during the training.

In [15]:
```
history_dict = history.history
history_dict.keys()
```

Out[15]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

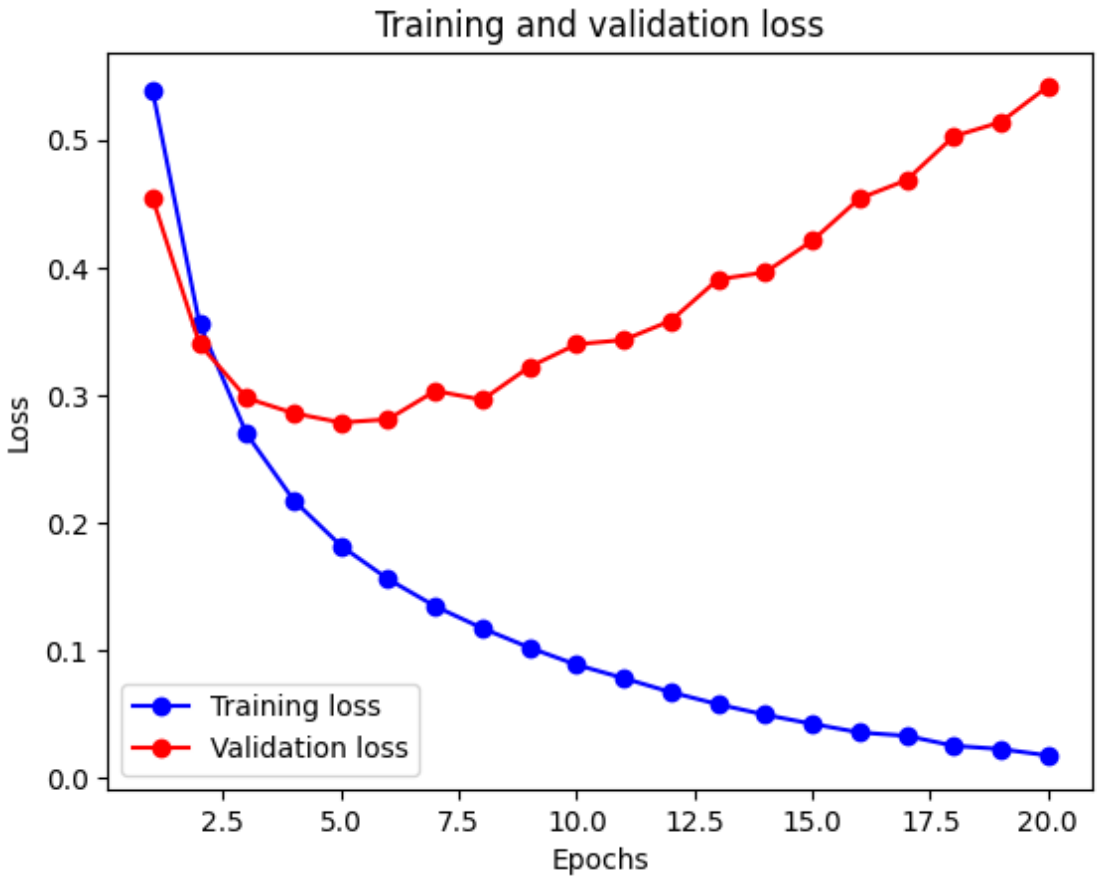
In [16]:
```
history_dict['loss']
```

Out[16]:
```
[0.5390281081199646,
 0.3556240499019623,
 0.2696126103401184,
 0.21766138076782227,
 0.1820698380470276,
 0.15601542592048645,
 0.13468341529369354,
 0.11750789731740952,
 0.10233305394649506,
 0.0888783186674118,
 0.07840856164693832,
 0.06744665652513504,
 0.057984646409749985,
 0.049764230847358704,
 0.04263148084282875,
 0.03591940551996231,
 0.03329065442085266,
 0.02547910064458847,
 0.02289753220975399,
 0.01790696755051613]
```

- Vamos a dibujar ahora sobre una gráfica como han ido variando, durante el entrenamiento, los datos de `loss` y de `val_loss`.
- En el eje X tendremos los distintos *epochs* numerados empezando en 1.

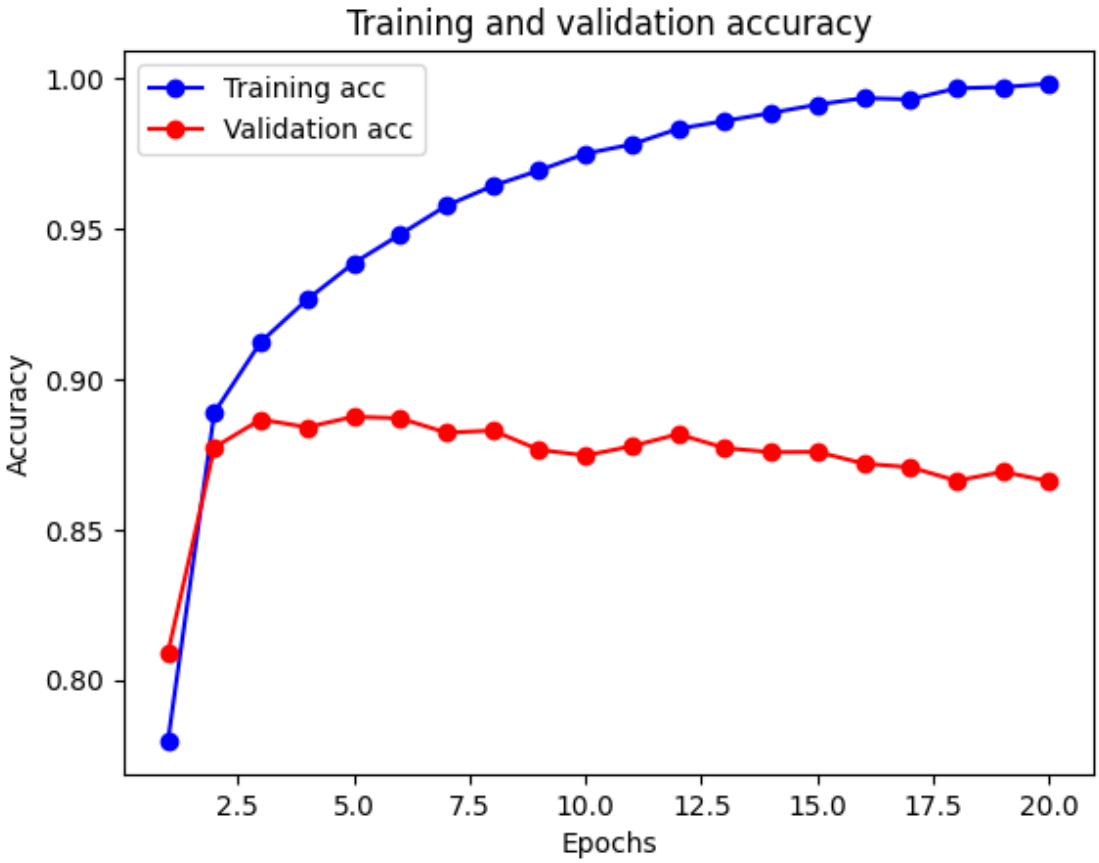
In [17]:
```
import os
os.environ['KMP_DUPLICATE_LIB_OK'] = 'True'
```

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

history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']

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

plt.plot(epochs, loss_values, 'b-o', label='Training loss')
plt.plot(epochs, val_loss_values, 'r-o', label='Validation loss')

plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```




- Now we represent the same thing but taking into account the values of accuracy

In [19]:
```
plt.clf()
acc = history_dict['accuracy']
val_acc = history_dict['val_accuracy']

plt.plot(epochs, acc, 'b-o', label='Training acc')
plt.plot(epochs, val_acc, 'r-o', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```




- As we can see during training the error is reduced and the accuracy of the model increases for the cases used in training.
- However, in the data we have set aside for validation we can see how we have a peak in accuracy at the third *epoch* but from there on the accuracy drops.
- What is happening is **overfitting**, the system is adapting too much to the training data, so it is not able to generalize with the validation data.

- We will evaluate the performance of the network using the test data `x_test` and `y_test` .

In [20]:
```
results = model.evaluate(x_test, y_test)
print(results)
```

```
782/782 [==============================] - 1s 915us/step - loss: 0.5901 -
accuracy: 0.8569
[0.590122401714325, 0.8569200038909912]
```

- The first value corresponds to the loss, the second to the accuracy.

## 2.4.2. Dropout

#### Definition

- Dropout is a technique where randomly selected neurons are ignored during training. They are "dropped out" randomly.
- This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass, and any weight updates are not applied to the neuron on the backward pass.
- It was developed by Geoff Hinton and his students at the University of Toronto.

#### Application

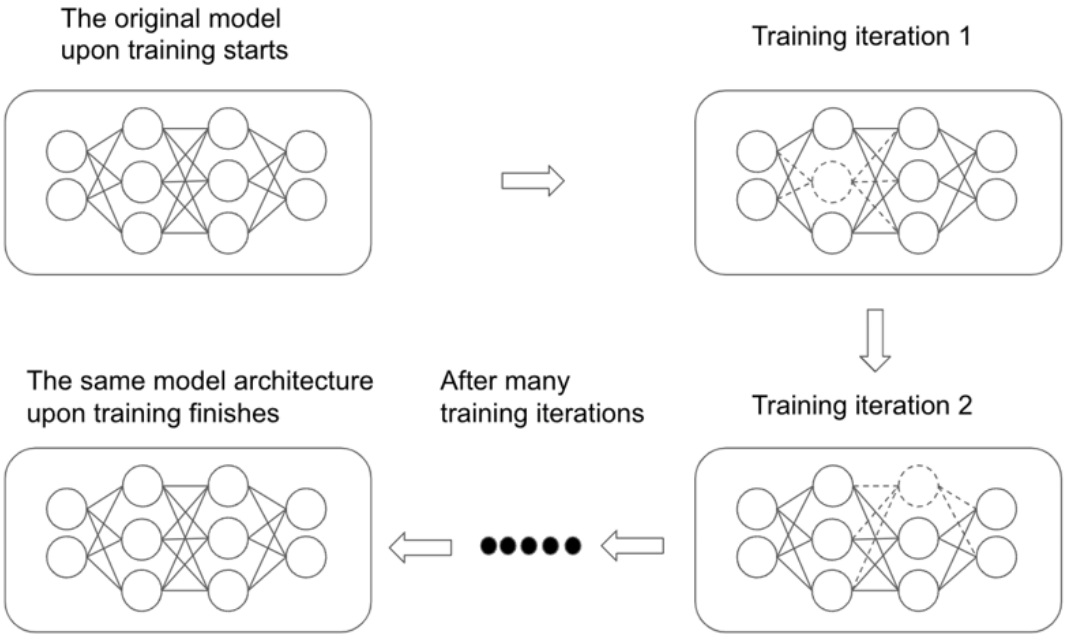
- The *dropout* is applied to a layer and consists of randomly zeroing a series of outputs of the layer during training.
- So if the output of a layer were the following vector: `[0.2, 0.5, 1.3, 0.8, 1.1]` after applying the *dropout* we would find some randomly distributed zero values: `[0, 0.5, 1.3, 0, 1.1]` .
- The *dropout* rate is the percentage of values to be set to zero and is usually set between `0.2` and `0.5` .

#### Dropout rationale

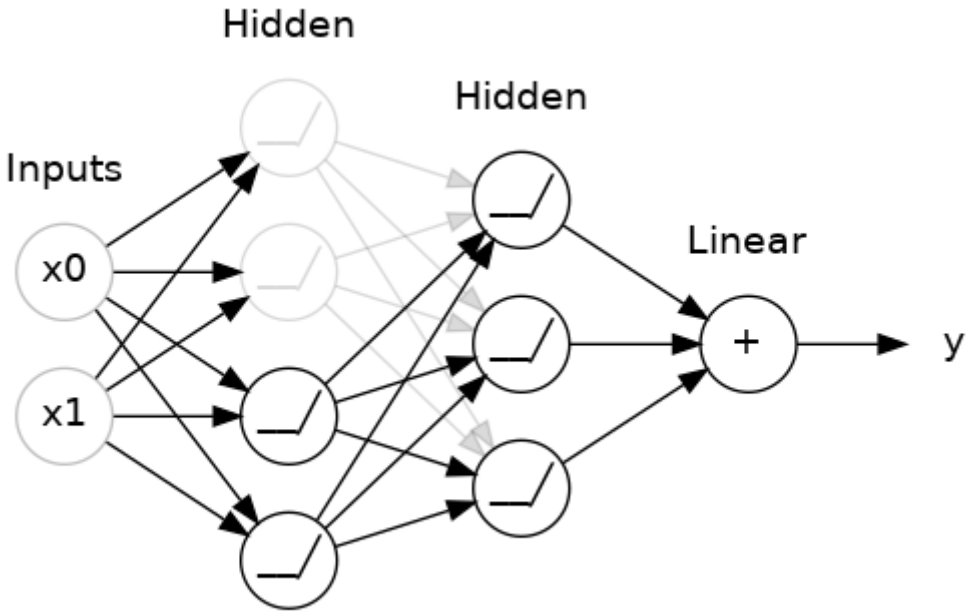
- The motivation for the *dropout* is to eliminate the co-adaptation between neurons that causes them to become highly specialized and lose the ability to generalize.
- The central idea of dropout is to approximate the ensemble learning as if we were to train many different models and average their predictions, since explicitly building multiple models is computationally expensive when the dataset gets large and the model gets complex.
- The approximation happens by randomly disabling certain nodes in the network (multiplied by zero), making it look like a different model for the current training iteration.
- Doing so will prevent the model from developing too much reliance on a particular node (as represented by its large weights), since it may just randomly get dropped out of the network.
- When a seemingly important node is dropped out, the other adjacent nodes will assume the additional shared responsibility for the current training task and learn to be more robust and less prone to co-adaptation, where adjacent nodes exhibit similar behavior and thus hurt generalization performance due to the high correlation.

**Dropout operation when training**

- Let's see how the dropout operation works when training a simple fully-connected neural network with two hidden layers.
- The probabilistic removal of hidden nodes happens during the training iterations.
- For example, the second node of the first hidden layer is removed at iteration 1, and the first node of the second hidden layer is removed at iteration 2.
- Note that muting a node also means silencing (setting to zero) all the connecting weights, as represented by the dashed line.
- We would end up with the same network architecture as the original version since the dropout operation only happens during training iterations.




- In the following image, we can see an animation in which a 50% dropout has been added between the two hidden layers.



### Dropout en Keras

- In Keras the *dropout* is applied by adding `Dropout` layers between the normal ( `Dense` ) layers of the network.
- The following example shows the operation of the IMDB network to which two *dropout* layers have been added after each of the hidden layers.

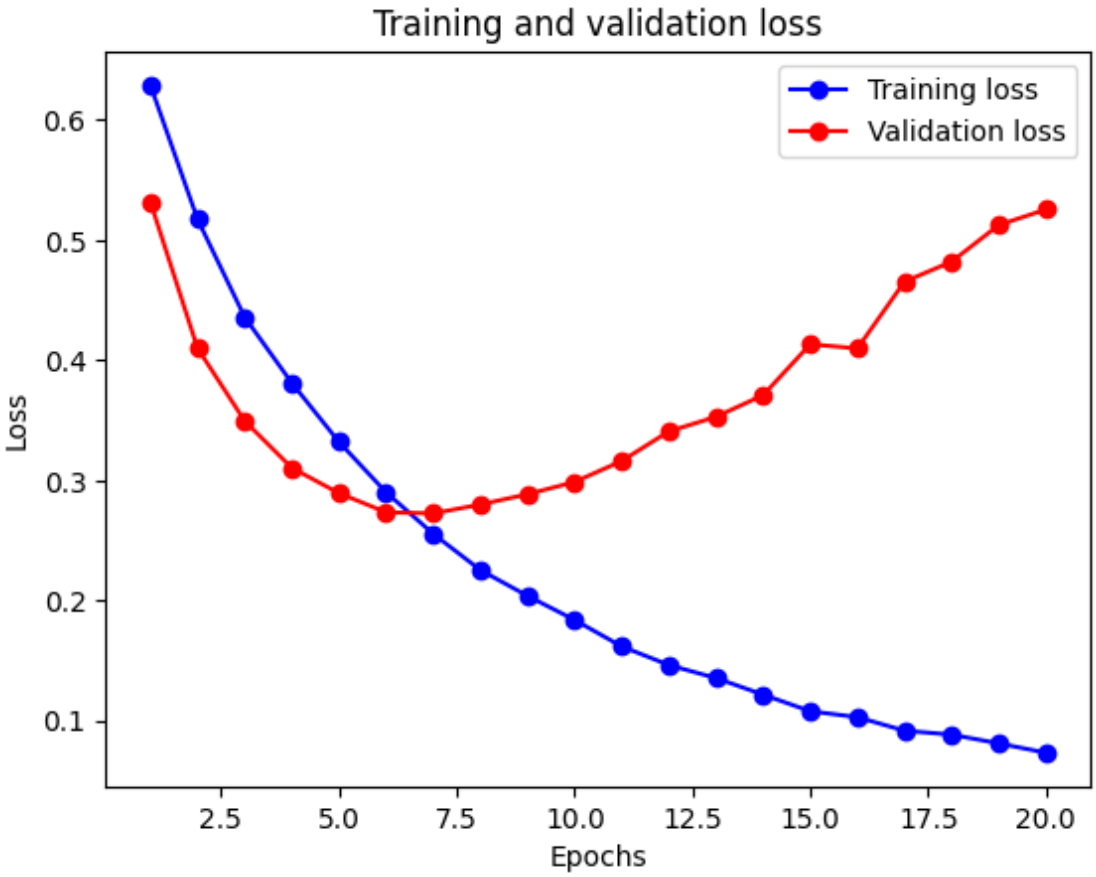
In [21]:
```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 16)                160016

 dropout (Dropout)           (None, 16)                0

 dense_1 (Dense)             (None, 16)                272

 dropout_1 (Dropout)         (None, 16)                0

 dense_2 (Dense)             (None, 1)                 17

=================================================================
Total params: 160305 (626.19 KB)
Trainable params: 160305 (626.19 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

In [22]:

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

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

history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']

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

plt.plot(epochs, loss_values, 'b-o', label='Training loss')
plt.plot(epochs, val_loss_values, 'r-o', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
Epoch 1/20
30/30 [==============================] - 1s 28ms/step - loss: 0.6291 - ac
curacy: 0.6366 - val_loss: 0.5318 - val_accuracy: 0.8015
Epoch 2/20
30/30 [==============================] - 0s 12ms/step - loss: 0.5174 - ac
curacy: 0.7578 - val_loss: 0.4100 - val_accuracy: 0.8712
Epoch 3/20
30/30 [==============================] - 0s 12ms/step - loss: 0.4359 - ac
curacy: 0.8178 - val_loss: 0.3495 - val_accuracy: 0.8821
Epoch 4/20
30/30 [==============================] - 0s 12ms/step - loss: 0.3811 - ac
curacy: 0.8463 - val_loss: 0.3104 - val_accuracy: 0.8819
Epoch 5/20
30/30 [==============================] - 0s 12ms/step - loss: 0.3320 - ac
curacy: 0.8727 - val_loss: 0.2894 - val_accuracy: 0.8856
Epoch 6/20
30/30 [==============================] - 0s 14ms/step - loss: 0.2901 - ac
curacy: 0.8953 - val_loss: 0.2735 - val_accuracy: 0.8874
Epoch 7/20
30/30 [==============================] - 0s 13ms/step - loss: 0.2555 - ac
curacy: 0.9088 - val_loss: 0.2725 - val_accuracy: 0.8868
Epoch 8/20
30/30 [==============================] - 0s 13ms/step - loss: 0.2254 - ac
curacy: 0.9238 - val_loss: 0.2800 - val_accuracy: 0.8857
Epoch 9/20
30/30 [==============================] - 0s 12ms/step - loss: 0.2039 - ac
curacy: 0.9333 - val_loss: 0.2885 - val_accuracy: 0.8866
Epoch 10/20
30/30 [==============================] - 0s 12ms/step - loss: 0.1838 - ac
curacy: 0.9439 - val_loss: 0.2987 - val_accuracy: 0.8880
Epoch 11/20
30/30 [==============================] - 0s 12ms/step - loss: 0.1618 - ac
curacy: 0.9498 - val_loss: 0.3162 - val_accuracy: 0.8851
Epoch 12/20
30/30 [==============================] - 0s 12ms/step - loss: 0.1463 - ac
curacy: 0.9541 - val_loss: 0.3408 - val_accuracy: 0.8854
Epoch 13/20
30/30 [==============================] - 0s 12ms/step - loss: 0.1355 - ac
curacy: 0.9591 - val_loss: 0.3530 - val_accuracy: 0.8818
Epoch 14/20
30/30 [==============================] - 0s 12ms/step - loss: 0.1216 - ac
curacy: 0.9644 - val_loss: 0.3710 - val_accuracy: 0.8833
Epoch 15/20
30/30 [==============================] - 0s 12ms/step - loss: 0.1079 - ac
curacy: 0.9665 - val_loss: 0.4134 - val_accuracy: 0.8840
Epoch 16/20
30/30 [==============================] - 0s 12ms/step - loss: 0.1029 - ac
curacy: 0.9682 - val_loss: 0.4099 - val_accuracy: 0.8836
Epoch 17/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0916 - ac
curacy: 0.9712 - val_loss: 0.4655 - val_accuracy: 0.8846
Epoch 18/20
30/30 [==============================] - 0s 13ms/step - loss: 0.0884 - ac
curacy: 0.9728 - val_loss: 0.4822 - val_accuracy: 0.8824
Epoch 19/20
30/30 [==============================] - 0s 12ms/step - loss: 0.0811 - ac
curacy: 0.9762 - val_loss: 0.5126 - val_accuracy: 0.8816
Epoch 20/20
30/30 [==============================] - 0s 12ms/step - loss: 0.0733 - ac
curacy: 0.9767 - val_loss: 0.5256 - val_accuracy: 0.8838
```




- We can see how the model takes longer to overfit and this overfit is smaller than if the *dropout* did not exist.

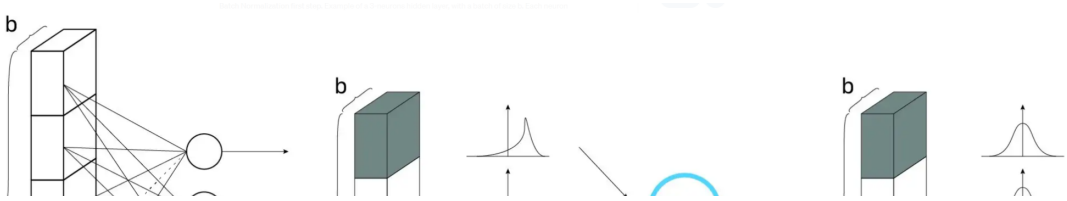
## 2.4.3. Batch normalization

#### Definition

- Batch normalization is a method used to make training of artificial neural networks faster and more stable through normalization of the layers' inputs by re-centering and re-scaling.
- Also it is a form of regularization that allows the network to reduce the chances of overfitting.

#### Batch normalization functioning

- Batch normalization works by normalizing the input to each layer of the network.
- This is done by first calculating the mean and standard deviation of the input, and then scaling the input so that it has a mean of 0 and a standard deviation of 1.
- Batch normalizaton also puts the data on a new scale with two trainable rescaling parameters.




**Tips for using batch normalization**

- Batch normalization may be used on the inputs to the layer before or after the activation function in the previous layer.
- Using batch normalization makes the network more stable during training. This may require the use of much larger than normal learning rates, that in turn may further speed up the learning process.
- Deep neural networks can be quite sensitive to the technique used to initialize the weights prior to training. The stability to training brought by batch normalization can make training deep networks less sensitive to the choice of weight initialization method.
- Batch normalization offers some regularization effect, reducing generalization error, reducing the need to use of dropout for regularization.
    - Also it is advisable not to use dropout with batch normalization because the statistics used to normalize the activations of the prior layer may become noisy given the random dropping out of nodes during the dropout procedure.

**Batch normalization in Keras**

- Batch normalization is also a layer that can be used after a layer or between a layer and its activation function:
- See: https://keras.io/api/layers/normalization_layers/batch_normalization/ (https://keras.io/api/layers/normalization_layers/batch_normalization/)

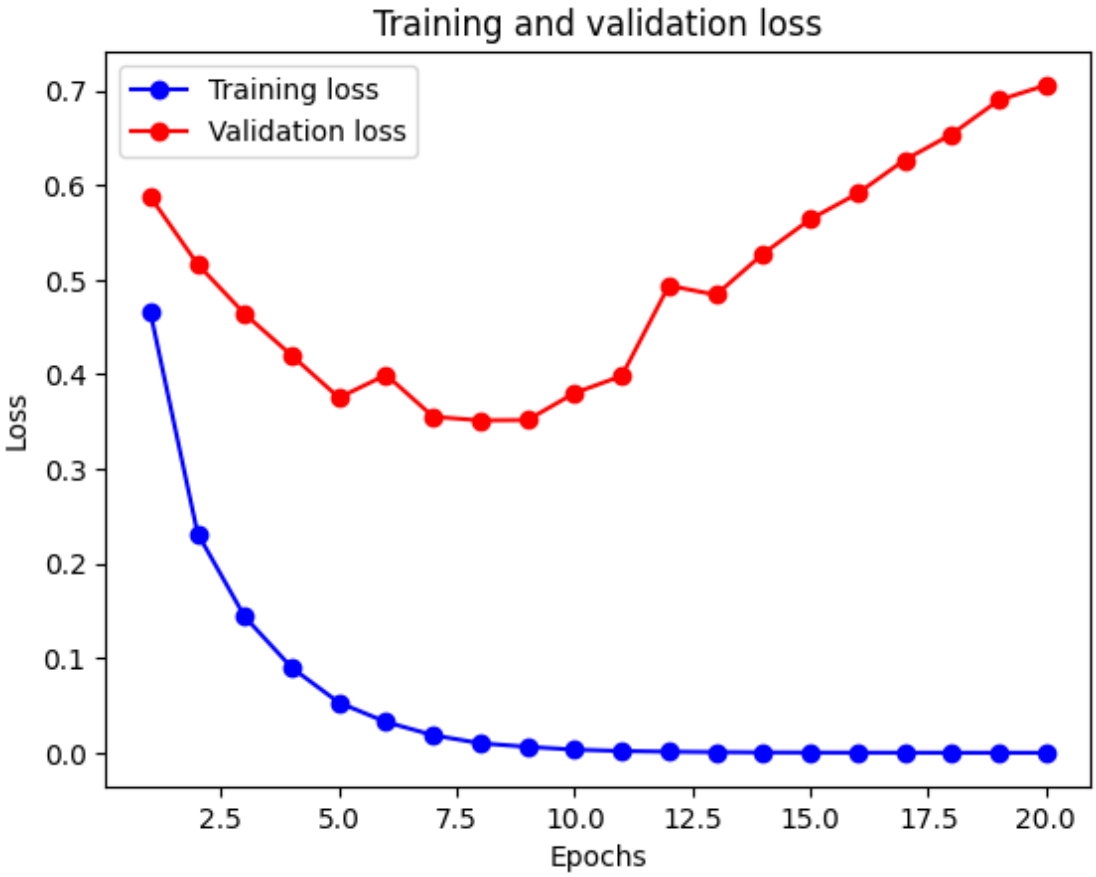
In [23]:
```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.BatchNormalization())
model.add(layers.Dense(16, activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.Dense(1, activation='sigmoid'))

model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_3 (Dense)             (None, 16)                160016

 batch_normalization (Batch  (None, 16)                64
 Normalization)

 dense_4 (Dense)             (None, 16)                272

 batch_normalization_1 (Bat  (None, 16)                64
 chNormalization)

 dense_5 (Dense)             (None, 1)                 17

=================================================================
Total params: 160433 (626.69 KB)
Trainable params: 160369 (626.44 KB)
Non-trainable params: 64 (256.00 Byte)
_________________________________________________________________
```

In [24]:

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

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

history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']

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

plt.plot(epochs, loss_values, 'b-o', label='Training loss')
plt.plot(epochs, val_loss_values, 'r-o', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
Epoch 1/20
30/30 [==============================] - 2s 23ms/step - loss: 0.4652 - ac
curacy: 0.7822 - val_loss: 0.5876 - val_accuracy: 0.8013
Epoch 2/20
30/30 [==============================] - 0s 9ms/step - loss: 0.2319 - acc
uracy: 0.9229 - val_loss: 0.5165 - val_accuracy: 0.8427
Epoch 3/20
30/30 [==============================] - 0s 9ms/step - loss: 0.1438 - acc
uracy: 0.9593 - val_loss: 0.4639 - val_accuracy: 0.8776
Epoch 4/20
30/30 [==============================] - 0s 9ms/step - loss: 0.0899 - acc
uracy: 0.9789 - val_loss: 0.4199 - val_accuracy: 0.8543
Epoch 5/20
30/30 [==============================] - 0s 9ms/step - loss: 0.0531 - acc
uracy: 0.9918 - val_loss: 0.3754 - val_accuracy: 0.8582
Epoch 6/20
30/30 [==============================] - 0s 9ms/step - loss: 0.0326 - acc
uracy: 0.9965 - val_loss: 0.3995 - val_accuracy: 0.8313
Epoch 7/20
30/30 [==============================] - 0s 9ms/step - loss: 0.0186 - acc
uracy: 0.9986 - val_loss: 0.3552 - val_accuracy: 0.8555
Epoch 8/20
30/30 [==============================] - 0s 9ms/step - loss: 0.0101 - acc
uracy: 0.9999 - val_loss: 0.3514 - val_accuracy: 0.8531
Epoch 9/20
30/30 [==============================] - 0s 9ms/step - loss: 0.0063 - acc
uracy: 0.9999 - val_loss: 0.3517 - val_accuracy: 0.8584
Epoch 10/20
30/30 [==============================] - 0s 9ms/step - loss: 0.0034 - acc
uracy: 0.9999 - val_loss: 0.3804 - val_accuracy: 0.8526
Epoch 11/20
30/30 [==============================] - 0s 9ms/step - loss: 0.0019 - acc
uracy: 1.0000 - val_loss: 0.3984 - val_accuracy: 0.8563
Epoch 12/20
30/30 [==============================] - 0s 10ms/step - loss: 0.0012 - ac
curacy: 1.0000 - val_loss: 0.4938 - val_accuracy: 0.8404
Epoch 13/20
30/30 [==============================] - 0s 10ms/step - loss: 7.7680e-04
- accuracy: 1.0000 - val_loss: 0.4841 - val_accuracy: 0.8553
Epoch 14/20
30/30 [==============================] - 0s 9ms/step - loss: 4.3371e-04 -
accuracy: 1.0000 - val_loss: 0.5275 - val_accuracy: 0.8575
Epoch 15/20
30/30 [==============================] - 0s 9ms/step - loss: 2.7291e-04 -
accuracy: 1.0000 - val_loss: 0.5639 - val_accuracy: 0.8545
Epoch 16/20
30/30 [==============================] - 0s 10ms/step - loss: 2.0126e-04
- accuracy: 1.0000 - val_loss: 0.5911 - val_accuracy: 0.8529
Epoch 17/20
30/30 [==============================] - 0s 11ms/step - loss: 1.6572e-04
- accuracy: 1.0000 - val_loss: 0.6265 - val_accuracy: 0.8514
Epoch 18/20
30/30 [==============================] - 0s 11ms/step - loss: 1.1808e-04
- accuracy: 1.0000 - val_loss: 0.6538 - val_accuracy: 0.8521
Epoch 19/20
30/30 [==============================] - 0s 11ms/step - loss: 9.6670e-05
- accuracy: 1.0000 - val_loss: 0.6898 - val_accuracy: 0.8533
Epoch 20/20
30/30 [==============================] - 0s 10ms/step - loss: 7.8548e-05
- accuracy: 1.0000 - val_loss: 0.7057 - val_accuracy: 0.8530
```




## 2.4.4. Weight Regularization

- A simpler network may give better results than a more complex network (which would over-fit the learning data).
- A simpler network may also mean having "simpler" weights.
- Here by "simple" we mean having less entropy, i.e., restricting the values of the weights to small numbers making their distribution more regular.
- This is known as **regularization of weights**.
- To achieve this, what is done is to add a cost to the error function associated with having large values in the weights.
- There are two ways of doing this, the **L1** regularization and the **L2** regularization.

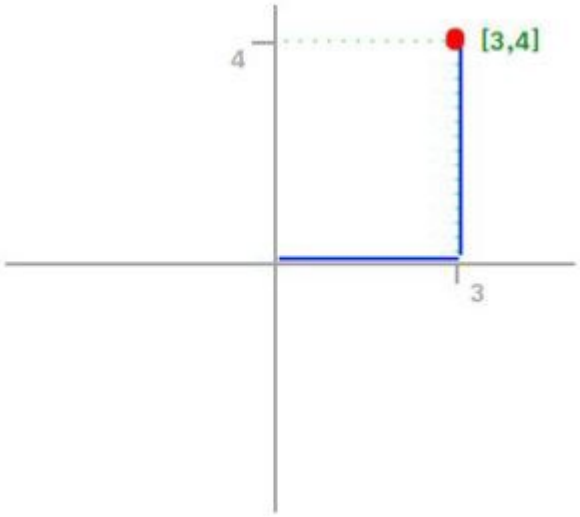
#### L1 norm (Lasso)

- The **L1** norm of a vector consists of the sum of the absolute values of the elements of the vector.

$$||\mathbf{w}||_1 = |w_1| + |w_2| + ... + |w_N|$$

1-norm (also known as L1 norm)

- The L1 norm is also known as the **Manhattan distance** or **Taxi norm** because it represents the distance to the origin of coordinates following a grid route (like the routes followed by cabs in Manhattan).



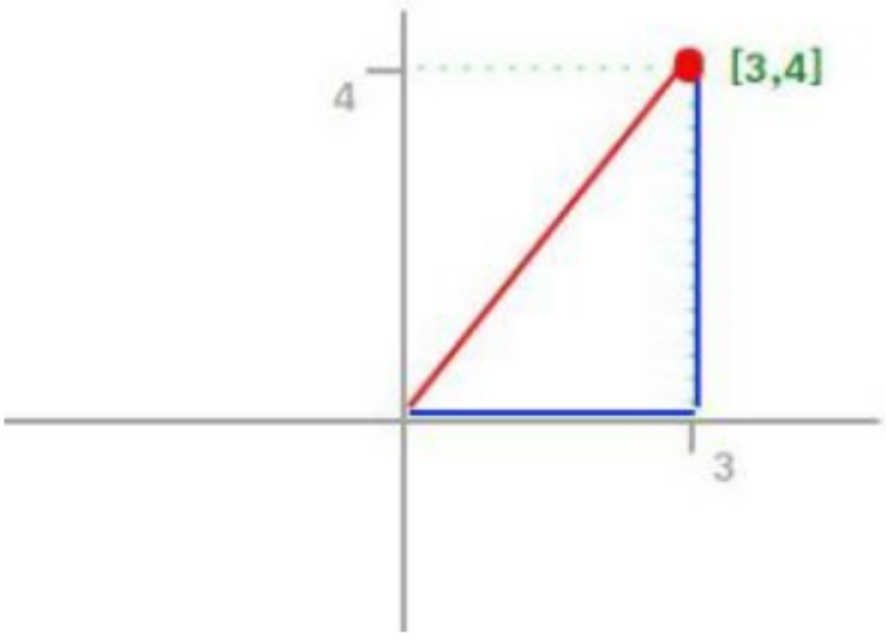

**L2 norm (Ridge)**

- The **L2 norm** of a vector consists of adding all the elements of the vector squared and then taking the square root of the result.

$$||\mathbf{w}||_2 = (w_1^2 + w_2^2 + ... + w_N^2)^{\frac{1}{2}}$$

2-norm (also known as L2 norm or Euclidean norm)

- It is also called Euclidean norm because it represents the distance to the origin of coordinates following the shortest distance.




$$||X||_2 = \sqrt{(|3|^2 + |4|^2)} = \sqrt{9+16} = \sqrt{25} = 5$$

- These norms can be generalized as *p-norms*:

$$||\mathbf{w}||_p = (w_1^p + w_2^p + ... + w_N^p)^{\frac{1}{p}}$$

**Modifying the loss function**

- We have that y is the desired output we expect.
- We can calculate the network output (assuming a single neuron) as:

$$\hat{y} = w_1 x_1 + w_2 x_2 + ... + w_N x_N + b$$

- Then we would have a loss function that would calculate the difference between the obtained value and the desired value:

$$Loss = Error(y, \hat{y})$$

Loss function with no regularisation

- Regularization consists of adding to the loss function a cost associated with the weights standards.
- The **L1** regularization consists in adding a cost to this loss function associated to the absolute value of the network weights.
- This cost is controlled by a parameter *lambda* as shown below:

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} |w_i|$$

Loss function with L1 regularisation

- The **L2** regularization consists of adding a cost to the loss function associated with the square of the network weights. In other words, the L2 rule is applied but without using the final square root.

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} w_i^2$$

Loss function with L2 regularisation

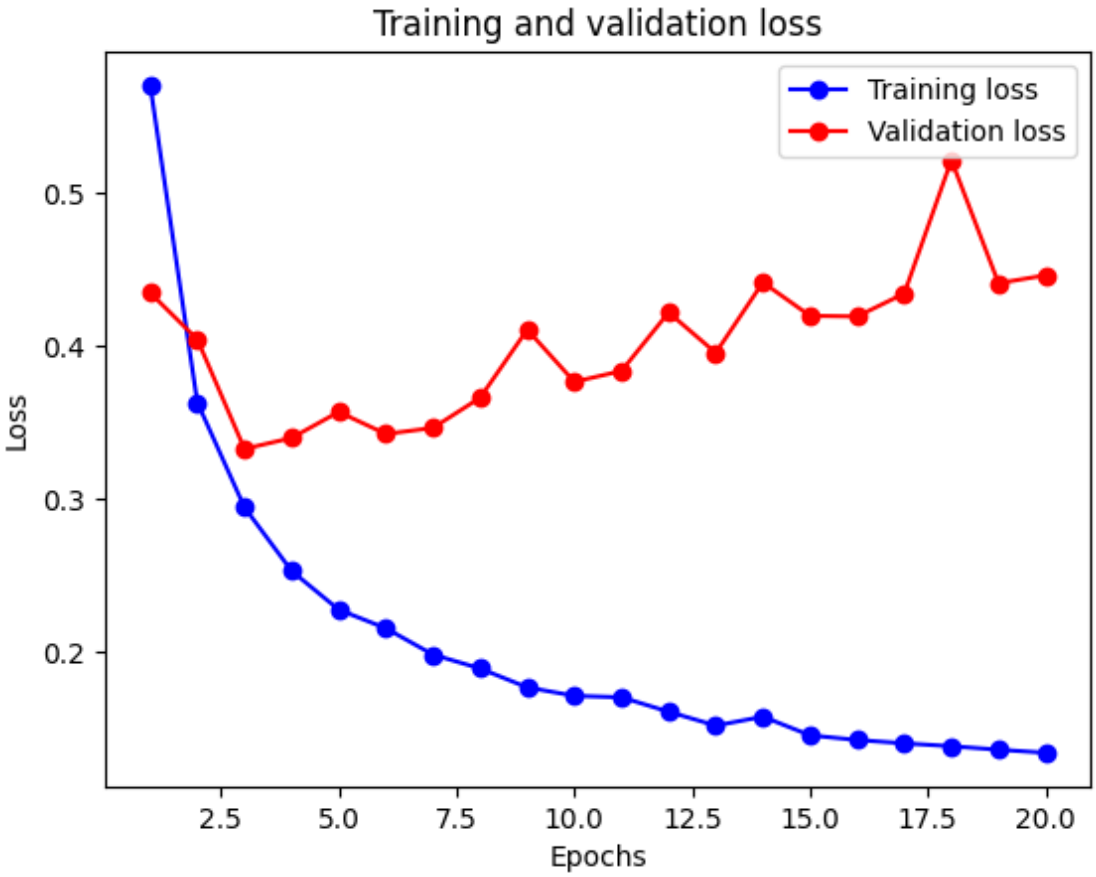
**Weight regularization in Keras**

- In Keras, weight regularizations are made by passing a regularizer parameter to the model.
- This is done when creating a layer using the `kernel_regularizer` parameter to which we pass an instance of a regularizer, which can be L1 or L2, with its corresponding *lambda* value.
- Let's see the effect of adding such a regularizer to the imdb network we have seen before.

In [25]:

```
from tensorflow.keras import regularizers

model = models.Sequential(name='imdb')
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001), acti
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001), acti
model.add(layers.Dense(1, name='output',  activation='sigmoid'))

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

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

history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']

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

plt.plot(epochs, loss_values, 'b-o', label='Training loss')
plt.plot(epochs, val_loss_values, 'r-o', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
Epoch 1/20
30/30 [==============================] - 1s 27ms/step - loss: 0.5707 - ac
curacy: 0.7771 - val_loss: 0.4350 - val_accuracy: 0.8648
Epoch 2/20
30/30 [==============================] - 0s 11ms/step - loss: 0.3634 - ac
curacy: 0.8935 - val_loss: 0.4046 - val_accuracy: 0.8531
Epoch 3/20
30/30 [==============================] - 0s 13ms/step - loss: 0.2948 - ac
curacy: 0.9149 - val_loss: 0.3331 - val_accuracy: 0.8880
Epoch 4/20
30/30 [==============================] - 0s 14ms/step - loss: 0.2533 - ac
curacy: 0.9319 - val_loss: 0.3402 - val_accuracy: 0.8817
Epoch 5/20
30/30 [==============================] - 0s 11ms/step - loss: 0.2280 - ac
curacy: 0.9405 - val_loss: 0.3572 - val_accuracy: 0.8756
Epoch 6/20
30/30 [==============================] - 0s 11ms/step - loss: 0.2158 - ac
curacy: 0.9445 - val_loss: 0.3428 - val_accuracy: 0.8854
Epoch 7/20
30/30 [==============================] - 0s 11ms/step - loss: 0.1986 - ac
curacy: 0.9528 - val_loss: 0.3468 - val_accuracy: 0.8845
Epoch 8/20
30/30 [==============================] - 0s 11ms/step - loss: 0.1896 - ac
curacy: 0.9566 - val_loss: 0.3665 - val_accuracy: 0.8758
Epoch 9/20
30/30 [==============================] - 0s 14ms/step - loss: 0.1772 - ac
curacy: 0.9633 - val_loss: 0.4110 - val_accuracy: 0.8648
Epoch 10/20
30/30 [==============================] - 0s 13ms/step - loss: 0.1717 - ac
curacy: 0.9647 - val_loss: 0.3769 - val_accuracy: 0.8811
Epoch 11/20
30/30 [==============================] - 0s 11ms/step - loss: 0.1707 - ac
curacy: 0.9640 - val_loss: 0.3840 - val_accuracy: 0.8764
Epoch 12/20
30/30 [==============================] - 0s 14ms/step - loss: 0.1613 - ac
curacy: 0.9702 - val_loss: 0.4222 - val_accuracy: 0.8713
Epoch 13/20
30/30 [==============================] - 0s 11ms/step - loss: 0.1522 - ac
curacy: 0.9750 - val_loss: 0.3958 - val_accuracy: 0.8803
Epoch 14/20
30/30 [==============================] - 0s 11ms/step - loss: 0.1581 - ac
curacy: 0.9692 - val_loss: 0.4419 - val_accuracy: 0.8646
Epoch 15/20
30/30 [==============================] - 0s 11ms/step - loss: 0.1459 - ac
curacy: 0.9754 - val_loss: 0.4200 - val_accuracy: 0.8771
Epoch 16/20
30/30 [==============================] - 0s 12ms/step - loss: 0.1428 - ac
curacy: 0.9774 - val_loss: 0.4196 - val_accuracy: 0.8758
Epoch 17/20
30/30 [==============================] - 0s 12ms/step - loss: 0.1407 - ac
curacy: 0.9781 - val_loss: 0.4346 - val_accuracy: 0.8742
Epoch 18/20
30/30 [==============================] - 0s 11ms/step - loss: 0.1388 - ac
curacy: 0.9799 - val_loss: 0.5218 - val_accuracy: 0.8489
Epoch 19/20
30/30 [==============================] - 0s 11ms/step - loss: 0.1365 - ac
curacy: 0.9790 - val_loss: 0.4414 - val_accuracy: 0.8744
Epoch 20/20
30/30 [==============================] - 0s 10ms/step - loss: 0.1345 - ac
curacy: 0.9787 - val_loss: 0.4465 - val_accuracy: 0.8746
```




- We can see that although the two networks have the same number of parameters the overfitting is more contained in this second network.
- Using regularizers we can also apply the L1 regularization in Keras or even both at the same time.

In [26]:
```
from tensorflow.keras import regularizers

regularizers.l1(0.001)
regularizers.l1_l2(l1=0.001, l2=0.001)
```

Out[26]:
```
<keras.src.regularizers.L1L2 at 0x1ed6b088310>
```

**Interpretation of weight regularization**

- L1 regularization is generally considered to generate models that are simple and interpretable, but not capable of learning complex patterns.
- On the other hand, L2 regularization gives rise to more complex models (no zero weights) but is more capable of learning complex patterns.
- Applying both at the same time, you can obtain an intermediate effect between the two.

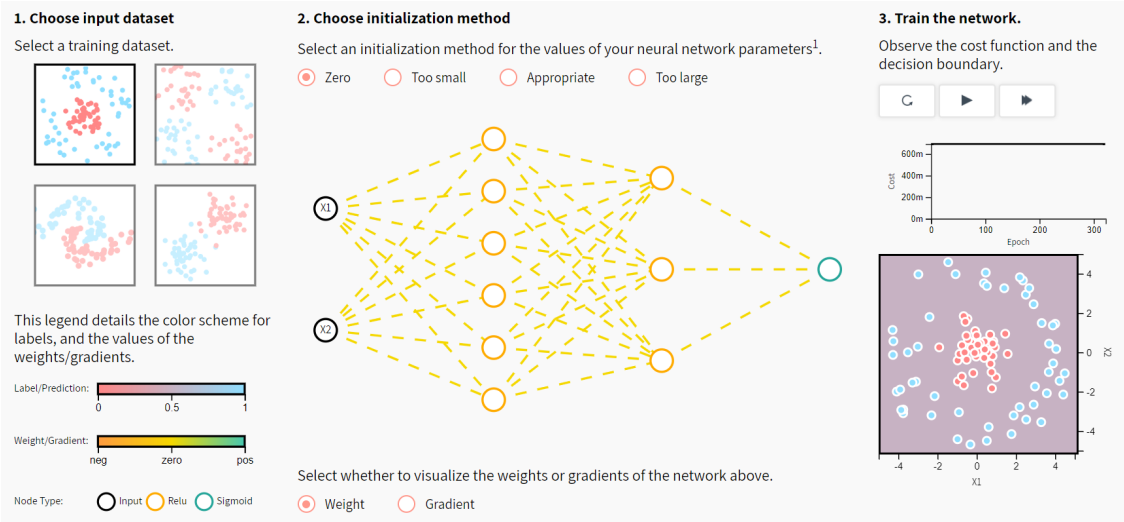
## 2.4.5. Weight initialization

- In order to be able to train the network, it has to offer, from the very beginning, results for a given input.

- The error function will then determine how to vary the weights so that the network learns to recognize the different input cases.
- But this requires that the weights have a certain initial value. Let's look at several

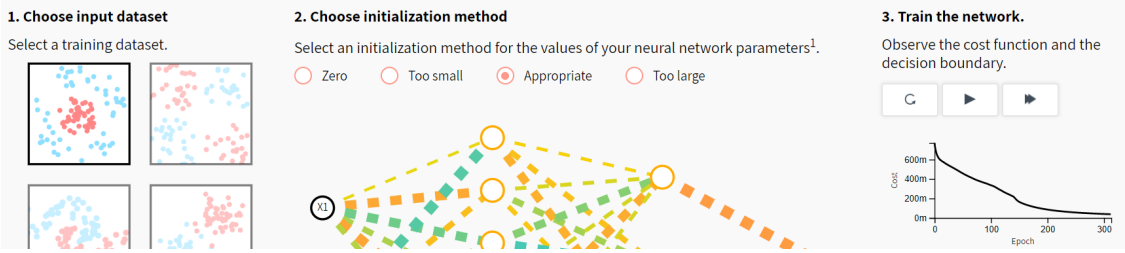
### Zero weight initialization

- One possibility we can think of is to initialize the weights to zero.
- But this is not a good idea because it turns our model into a model equivalent to a linear model.
- The derivative with respect to the error function is always the same for each weight, so all weights will have the same values in the next and subsequent iterations.
- In general any initialization of the weights to a constant value will have lousy performance.
- There is no problem with setting the bias (*bias*) to zero as long as the weights are not. Moreover, it is standard practice. Non-zero weights will break the symmetry and the results for each neuron will be different.
- At [https://www.deeplearning.ai/ai-notes/initialization/ (https://www.deeplearning.ai/ai-notes/initialization/)](https://www.deeplearning.ai/ai-notes/initialization/) we have a great simulator that allows us to see this.
- For example if we set the weights to zero we see how the network is unable to learn anything and all the weights in the network have the same values after multiple iterations.



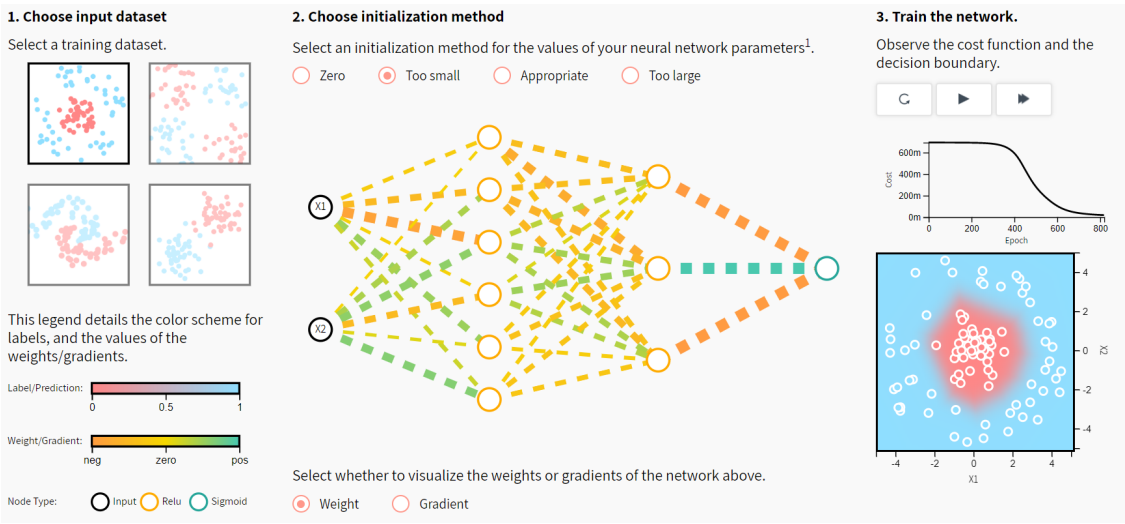

### Random weight initialization

- If we cannot initialize the weights to zero we can initialize them to a random value.
- Following the previous example we see how the network starts to converge quickly and reaches minimum values after three hundred iterations.



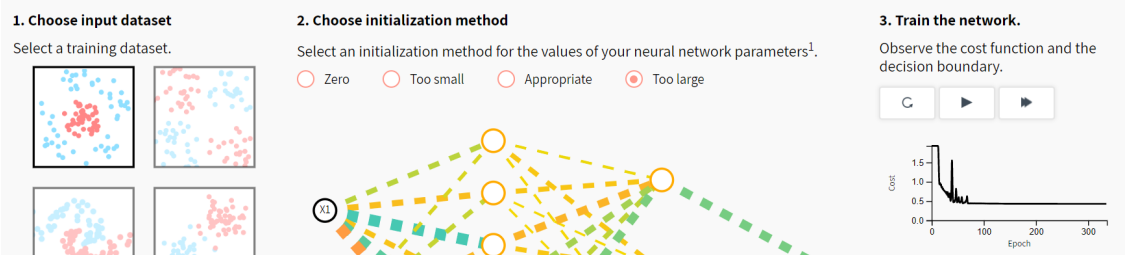

## Vanishing gradients

- Vanishing gradients are a problem that occurs when the variations in the weights after each iteration are very small, causing the optimization of the loss function to be very slow or, in the worst case, to stop the learning of the neural network altogether.
- This occurs for example with activation functions such as the hyperbolic tangent or the sigmoid with very large weights that cause the activation function to always present values close to one.
- Functions such as ReLU are more immune to this problem, but may fall into gradient vanishing if the values of the weights are very small.
- In the image you can see ReLU neurons initialized with very small weights. We can see that the network takes a long time to converge and until iteration 800 it does not reach levels that we previously reached with 300 iterations.




## Exploding gradients

- Exploding gradients are the opposite case. Very large changes in the values of the weights cause large changes in the error function. The gradients are then very large, causing it to oscillate around the value without falling into it and the model is unable to learn.
- With a ReLU activation function this can happen if the weights have too large values. In the following example you can see how the error function makes very large oscillations and then is unable to find the minimum value.




**Different types of weight initialization**

- Using the ReLU activation function is a way to avoid vanishing gradients or exploding gradients.
- To avoid the problem, the initialization of the weights must be done according to the following rules:
  - The mean of the activations must be zero.
  - The variance of the activations must be the same across each of the layers.
- This way the gradient signal to be backpropagated will not be multiplied by too large or too small values avoiding the problems of explosion or vanishing.
- **A suitable weight initialization algorithm follows the following steps**:

  1. The biases are initialized to zero.
  2. The weights are randomly chosen from a zero-mean normal distribution.
  3. To choose the standard deviation of the normal distribution we have several possibilities. In them `fan_in` refers to the number of incoming connections (neurons of the previous layer) and `fan_out` refers to the number of outgoing connections (neurons of the next layer).
- **Types of weight initialization**
  - *Inicialización He*
    - Propuesta por Kaiming He, investigador de Microsoft: https://arxiv.org/pdf/1502.01852.pdf (https://arxiv.org/pdf/1502.01852.pdf)
    - La desviacion estándar se calcula como `stddev = sqrt(2 / fan_in)`
  - *Inicialización Xavier*
    - Debe su nombre a Xavier Glorot, investigador de la Universidad de Montreal junto a Yoshua Bengio. (ref. http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf (http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf)).
    - La desviacion estándar se calcula como `stddev = sqrt(1 / fan_in)`
  - *Inicialización Glorot*
    - También propuesta por Xavier Glorot
    - La desviacion estándar se calcula como `stddev = sqrt(2 / (fan_in + fan_out))`
  - *Inicialización Glorot uniforme*
    - En este caso los pesos se obtienen de una distribución uniforme (todos los valores son igualmente probables) en el intervalo `[-limit, limit]` en donde `limit` es `sqrt(6 / (fan_in + fan_out))` .
- Los distintos inicializadores de Keras pueden consultarse en: https://keras.io/initializers/ (https://keras.io/initializers/)

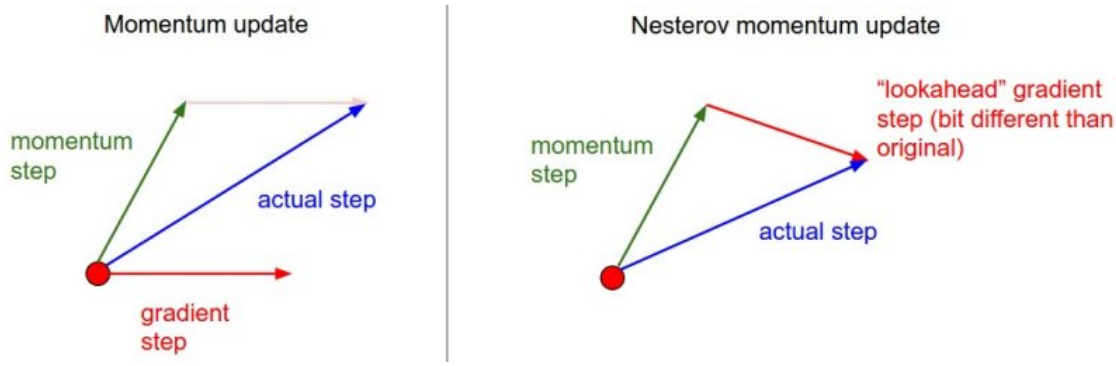
In [30]:
```
>>> # Usage in a Keras layer:
>>> initializer = tf.keras.initializers.HeNormal()
>>> layer = tf.keras.layers.Dense(3, kernel_initializer=initializer)
```

## 2.2.6. Gradient descent optimizers

- Stochastic gradient descent (SGD) can be very slow, so many optimizers has been proposed to improve its performance.

### Momentum

- A parameter that we can add to the gradient descent operation is the *momentum*.
- Momentum in physics is defined as the *amount of motion* calculated as the multiplication of mass times velocity.
- In the case of gradient descent the momentum is a parameter that adds *inertia* to the weight update process, making past updates in a certain direction indicate that this direction will continue in the future.
- In the following image you can see how the momentum works in gradient descent.
  - The image on the left represents the pure momentum, the changes in the weights have one direction (momentum) and the gradient indicates another, so the final direction is a combination of both.
  - The Nesterov momentum is a modification of the previous one in which the gradient step is calculated from the position where the momentum would take us (not at the current position), so the result is slightly different.

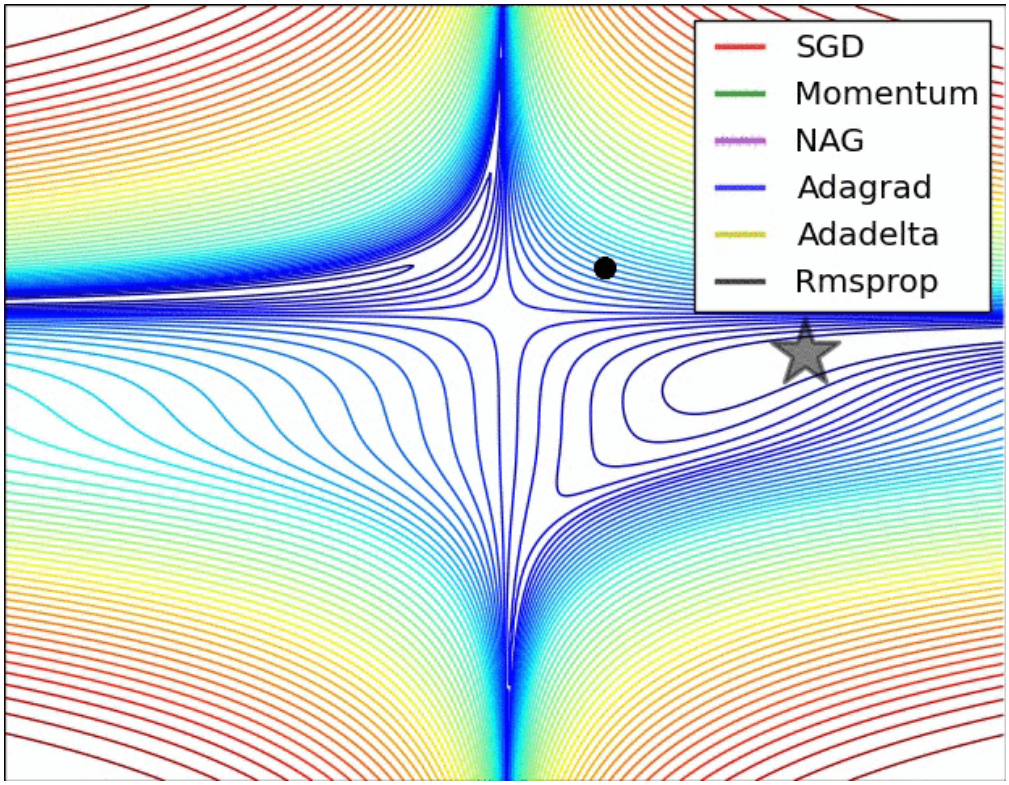


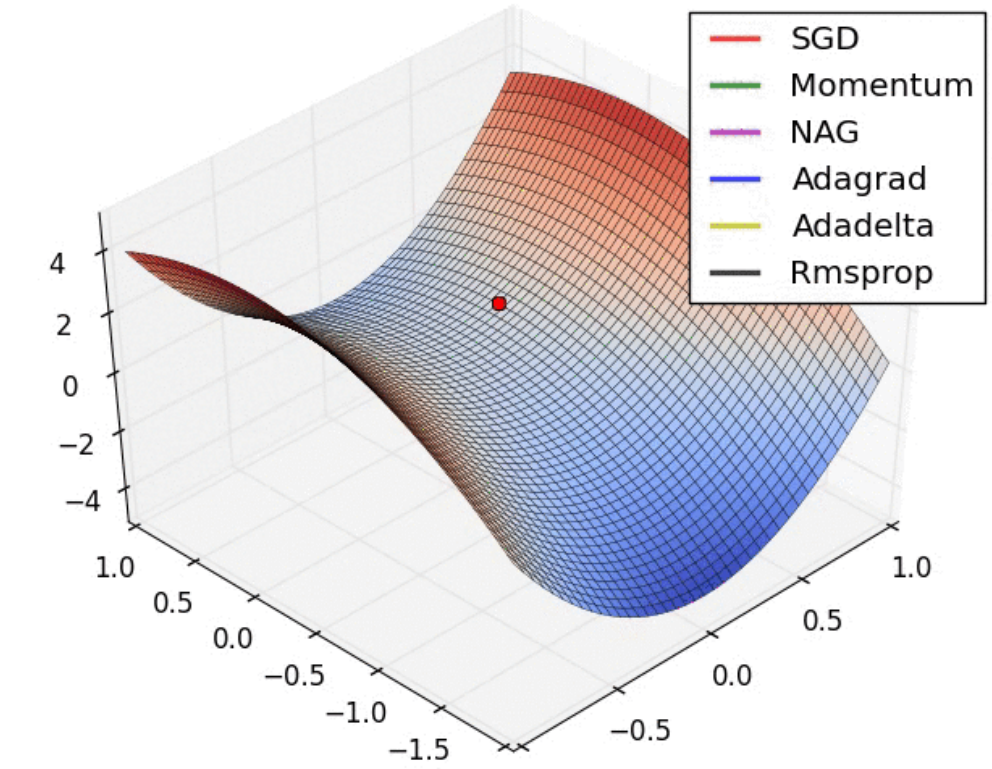

### Adaptive gradient descent algorithms

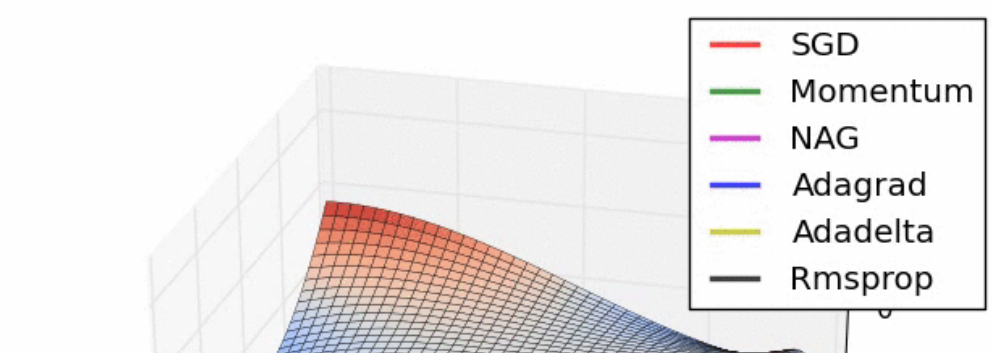
- Adaptive gradient descent algorithms are an alternative to stochastic gradient descent (SGD).
- Below is a table summarizing the most common gradient descent optimizers:

| Algorithm | Description |
| --- | --- |
| Stochastic gradient descent (SGD) | Gradient is computed based on a mini-batch of training examples. |
| Momentum | Addition to SGD where weight adjustment depends on gradient from previous adjustments as well as the current gradient. |
| AdaGrad | Variation on SGD that adaptively adjusts the learning rate during training. |

- Performance comparison of different gradient descent optimizers:






**Keras**

- In Keras these values can be passed directly as *strings* since Keras will recognize them

In [31]:
```
model = models.Sequential()
model.compile(optimizer = 'rmsprop',
              loss = 'binary_crossentropy',
              metrics = ['accuracy'])
```

- But we can also pass objects if we want to add additional parameters to them.
- So we can create the optimizer as an object of class `RMSprop` to which we indicate that the *learning rate* is `0.001` .
- And use the loss functions and the test metric using the functions already created.
- The following `compile` code would be equivalent to the previous one.

In [32]:
```
from tensorflow.keras import optimizers
from tensorflow.keras import losses
from tensorflow.keras import metrics

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

- The different Keras optimizers can be consulted at the following address: https://keras.io/optimizers/ (https://keras.io/optimizers/)