Améliorer notre classifieur de chats et chiens grâce à un réseau préentraîné

Dans l'exercice précédent, nous avons créé un réseau, que nous avons entraîné à distinguer des photos de chiens de photos de chats.

Nous allons maintenant utiliser une autre approche, et adapter un réseau pré-entraîné.

Les images que nous allons utiliser sont les mêmes que celles utilisées pour entraîner le classifieur linéaire et le premier réseau convolutionnel.

Elles sont issues d'une compétition de la plateforme Kaggle (https://www.kaggle.com/c/dogs-vs-cats/overview). Cette compétition, datant de 2013, a été remportée par un français, Pierre Sermanet (qui travaille maintenant pour Google Brain), qui a obtenu une précision de 98.9% en utilisant un réseau de neurones convolutionnel. Cependant, comme nous n'allons considérer ici qu'un petit sous-ensemble de ces données, nous n'aurons probablement qu'une précision beaucoup plus faible.

Voici la structure de ce notebook :

Imports

Nous commençons par importer les paquets nécessaires pour ce notebook.

La commande `%matplotlib inline` permet d'afficher les graphiques de matplotlib avec la bibliothèque graphique intégrée à Notebook. Sans cette commande, les graphiques générés avec `matplotlib` ne s'afficheront pas.

Le bibliothèque suivantes sont importées :

- La bibliothèque `OpenCV` , qui s'importe avec la commande `import cv2` , nous servira à lire et à prétraiter les images avant qu'elles n'entrent dans le classifieur ;
- `Matplotlib` est utilisée pour afficher des images et tracer des courbes ;
- `Numpy` sert à faire des calculs matriciels. Les images lues par `OpenCV` sont stockées

comme des matrices `Numpy` ;
- `Os` permet d'utiliser des fonctionnalités dépendantes du système d'exploitation, comme créer des dossiers, ou spécifier des chemins à partir de noms de dossiers ;
- `Random` permet de générer des nombres aléatoires. Nous nous en servirons pour initialiser les paramètres du classifieur ;
- La bibliothèque `Scikit-Learn`, qui s'importe avec la commande `import sklearn`, contient une grande quantité d'outils pour l'apprentissage automatique. Cette librairie nécessite d'importer individuellement tous les sous-paquets qui seront utilisés. Dans ce notebook, nous l'utiliserons pour calculer diverses métriques permettant d'évaluer la performance de notre classifieur, nous importons donc son sous-paquet `metrics`. Nous l'utiliserons également pour séparer les données en sous-ensembles d'entraînement, validation et test, avec le sous-paquet `model_selection` ;
- La bibliothèque `TensorFlow` et son module `Keras` nous permettront de créer et d'entraîner le réseau de neurones.

In [6]:
```
# Imports
%matplotlib inline
import cv2
from matplotlib import pyplot as plt
import numpy as np
import os
import random
import sklearn.model_selection
import tensorflow as tf
from tensorflow import keras
```

Ce notebook a été créé avec les versions de paquets suivantes :

In [7]:
```
print("Version d'OpenCV", cv2.__version__)
print("Version de Numpy", np.__version__)
print("Version de Scikit-Learn", sklearn.__version__)
print("Version de TensorFlow", tf.__version__)
print("Version de Keras", keras.__version__)
```

```
Version d'OpenCV 4.8.0
Version de Numpy 1.23.4
Version de Scikit-Learn 1.1.2
Version de TensorFlow 2.6.0
Version de Keras 2.6.0
```

Certains de ces paquets sont installés par défaut à l'installation de Python. D'autres doivent être installés séparément.

Si l'import d'un de ces paquets échoue en renvoyant un message d'erreur spécifiant `ModuleNotFoundError`, c'est probablement que le paquet n'a pas été installé. Dans ce cas, installez le paquet manquant en exécutant la commande `!pip install <paquet manquant>`.

La cellule qui suit installe tous les paquets nécessaires à ce notebook. Commentez les lignes correspondant aux paquets qui sont déjà installés, exécutez la cellule pour installer les paquets restants, et exécutez à nouveau la cellule effectuant l'import des modules nécessaires.

In [8]:
```
!pip install opencv-python
!pip install matplotlib
!pip install numpy
!pip install scikit-learn
!pip install tensorflow
!pip install keras
```

```
Defaulting to user installation because normal site-packages is not writea
ble
Requirement already satisfied: opencv-python in c:\users\ispri\appdata\roa
ming\python\python39\site-packages (4.8.0.76)
Requirement already satisfied: numpy>=1.19.3 in c:\users\ispri\appdata\roa
ming\python\python39\site-packages (from opencv-python) (1.24.3)

WARNING: You are using pip version 22.0.4; however, version 23.2.1 is avai
lable.
You should consider upgrading via the 'C:\Program Files\Python39\python.ex
e -m pip install --upgrade pip' command.

Defaulting to user installation because normal site-packages is not writea
ble
Requirement already satisfied: matplotlib in c:\users\ispri\appdata\roamin
g\python\python39\site-packages (3.7.2)
Requirement already satisfied: cycler>=0.10 in c:\users\ispri\appdata\roam
ing\python\python39\site-packages (from matplotlib) (0.11.0)
Requirement already satisfied: packaging>=20.0 in c:\users\ispri\appdata\r
oaming\python\python39\site-packages (from matplotlib) (23.1)
```

Création d'ensembles de données

Nous allons utiliser ici exactement les mêmes ensembles de données que dans le notebook correspondant au chapitre 5.

Ainsi, il ne sera pas nécessaire d'exécuter certaines cellules de cette partie (qui seront clairement indiquées), comme le téléchargement de l'ensemble de données par exemple, si vous les avez déjà exécutées dans le notebook correspondant aux chapitres précédents. D'autres, cependant, comme la création des matrices de données d'entraînement, validation et test, devront être obligatoirement exécutées.

Comme mentionné dans les notebooks correpondant aux chapitres 3 et 5, il existe plusieurs manières d'obtenir cet ensemble de données :

- il peut être téléchargé depuis la plateforme du concours, Kaggle (https://www.kaggle.com/c/dogs-vs-cats/overview). Ceci nécessite de s'inscrire et de créer un compte sur la plateforme ;
- il peut être téléchargé en utilisant la fonction `load` du paquet `tfds` de tensorflow (https://www.tensorflow.org/datasets/catalog/cats_vs_dogs?hl=en). Les donnnées sont téléchargées au format TFRecord, un format binaire qui permet de stocker des images de manière beaucoup plus compacte et rapidement utilisable que les formats d'images classiques (png, jpeg, etc.), mais qui a l'inconvénient de ne pas permettre une visualisation immédiate du contenu ;
- il peut également être téléchargé depuis le site de Microsoft (https://www.microsoft.com/en-us/download/details.aspx?id=54765).

C'est cette dernière solution qui sera utilisée ici.

Nous allons commencer par télécharger l'ensemble des données, mais nous n'en utiliserons qu'une fraction pour l'entraînement et les tests, afin de limiter les ressources (mémoire et calculs) requises.

Une fois les données pour l'entraînement et les tests sélectionnées, nous aurons besoin d'uniformiser leurs tailles avant de les utiliser.

Nous ne leur appliquerons pas d'autre traitement : chaque réseau pré-entraîné attend des images différentes en entrée (images en niveaux de gris ou avec trois canaux de couleur, intensités comprises entre 0 et 1 o entre -1 et 1, etc.).

Nous leur appliquerons donc un prétraitement supplémentaire uniquement après avoir

Téléchargement de l'ensemble des données

Si vous avez déjà téléchargé et décompressé l'ensemble de données dans le notebook précédent, vous pouvez sauter toutes les cellules correspondant à cette partie.

Nous allons commencer par télécharger l'ensemble des données dans l'archive cats-and-dogs.zip. Nous allons ensuite décompresser l'archive dans le dossier tmp/PetImages, contenant les sous-dossiers 'Cat' et 'Dog'.

Si le lien de téléchargement ne fonctionne pas, vous pouvez exécuter toutes ces étapes manuellement, en commençant par télécharger les images depuis l'adresse https://www.microsoft.com/en-us/download/confirmation.aspx?id=54765 (https://www.microsoft.com/en-us/download/confirmation.aspx?id=54765).

Pour pouvoir télécharger les données, nous avons besoin d'installer le paquet `wget`, qui permet de télécharger des fichiers depuis internet.

Le paquet `zipfile38` nous servira à décompresser l'archive.

In [9]:
```
!pip install wget
!pip install zipfile38
```

```
Defaulting to user installation because normal site-packages is not writea
ble
Requirement already satisfied: wget in c:\users\ispri\appdata\roaming\pyth
on\python39\site-packages (3.2)

WARNING: You are using pip version 22.0.4; however, version 23.2.1 is avai
lable.
You should consider upgrading via the 'C:\Program Files\Python39\python.ex
e -m pip install --upgrade pip' command.

Defaulting to user installation because normal site-packages is not writea
ble
Requirement already satisfied: zipfile38 in c:\users\ispri\appdata\roamin
g\python\python39\site-packages (0.0.3)

WARNING: You are using pip version 22.0.4; however, version 23.2.1 is avai
lable.
You should consider upgrading via the 'C:\Program Files\Python39\python.ex
e -m pip install --upgrade pip' command.
```

Nous pouvons maintenant télécharger les images dans l'archive `cats-and-dogs.zip` avec `wget` .

In [10]:
```
import wget

url = "https://download.microsoft.com/download/3/E/1/3E1C3F21-ECDB-4869
local_zip_path = "cats-and-dogs.zip"
wget.download(url, local_zip_path)
```

```
  0% [
]   1884160 / 824887076

  0% [
]   5562368 / 824887076

  1% [
]   8970240 / 824887076

  1% [.
]  12877824 / 824887076

  2% [.
]  16719872 / 824887076

  2% [.
]  21315584 / 824887076

  2% [.
]  23085056 / 824887076
```

Ensuite, nous décompressons l'archive grâce au paquet `zipfile` .

In [11]:
```
import zipfile

local_zip_path = "cats-and-dogs.zip"
zip_ref = zipfile.ZipFile(local_zip_path, 'r')
zip_ref.extractall()
zip_ref.close()
```

Le dossier décompressé s'appelle `PetImages` .

Si vous l'ouvrez, vous découvrirez qu'il comporte deux sous-dossiers : `Cat` , qui contient 12 501 images de chats, et `Dog` , qui contient 12 501 images de chiens.

Pré-traitement des images

Chacune de ces images a une taille différente.

Nous allons donc harmoniser leurs tailles.

Notons que nous pouvons choisir n'importe quelle taille pour nos images d'entrée. En effet, seules les couches complètement connectées requièrent une taille d'entrée fixée, les couches de convolution sont indépendantes de la taille des images d'entrée. Pour ré-entraîner un réseau, nous allons justement supprimer toutes ses couches complètement connectées, et ne conserver que les couches de convolution. Nous pouvons donc choisir la

taille d'images que nous voulons.

Ici, et comme dans le notebook précédent, nous allons utiliser des images de taille 128 x 128. Pour des raisons d'efficacité des calculs, les tailles choisies en entrée sont souvent des puissances de 2.

Nous allons tout d'abord créer une fonction qui redimensionne une image individuelle.

Pour cela, nous définissons une fonction, `resize_images_in_folder`, qui prend en entrée le chemin vers une image. Cette image est lue et redimensionnée.

Nous allons appliquer un redimensionnement simple, qui modifiera le ratio hauteur/largeur de la plupart des images. Il est possible que ceci affecte légèrement la performance de la classification, mais, comme c'est la manière la plus simple de redimensionner une image, c'est généralement celle qui est appliquée lors des premiers tests.

Libre à vous de faire ensuite des expériences avec des images de tailles plus grandes ou plus petites, ainsi qu'avec des redimensionnements qui permettent de conserver le ratio hauteur/largeur des images.

In [12]:
```
# Données
root_dir = 'PetImages'
cat_dir = 'Cat'
dog_dir = 'Dog'
```

In [13]:
```
img_size = 128
```

In [14]:
```
def resize_image(input_image_path, output_image_size):
    # On lit l'image
    image_content = cv2.imread(input_image_path)
    # On vérifie que l'image peut être lue.
    if image_content is None:
        print("could not read image ", input_image_path)
        return image_content
    else:
        # On la redimensionne
        square_image = cv2.resize(image_content, (output_image_size, ou

        return square_image
```

Commençons par appliquer cette fonction à une image pour voir exactement ce qu'elle fait.

Nous choisissons une image au hasard du dossier `Cat`, affichons sa taille et ses intensités min et max.

In [15]:
```
# Sélection d'une image dans le dosser Cat
train_cat_dir = os.path.join(root_dir, cat_dir)
img_name = os.listdir(train_cat_dir)[0]
img_path = os.path.join(train_cat_dir, img_name)
original_img = cv2.imread(img_path)
print("Taille de l'image originale :", original_img.shape)
print("Intensités min et max de l'image originale :", np.min(original_i
```

```
Taille de l'image originale : (375, 500, 3)
Intensités min et max de l'image originale : 0 255
```

Nous appliquons ensuite la fonction `resize_image` à cette image, et affichons sa taille pour vérifier que le redimensionnement fonctionne bien.

In [16]:
```
# Conversion de l'image grâce à notre fonction grayscale_and_resize_ima
processed_img = resize_image(img_path, img_size)
print("Taille de l'image prétraitée :", processed_img.shape)
```

```
Taille de l'image prétraitée : (128, 128, 3)
```

Affichons maintenant ces deux images.

Notons que `OpenCV` lisant les images en utilisant les canaux de couleur dans l'ordre (B, G, R), il est nécessaire de remettre les canaux dans l'ordre habituel (R, G, B) avant de les afficher avec `matplotlib` .

In [17]:
```
# Affichage de l'image originale
rgb_img = cv2.cvtColor(original_img, cv2.COLOR_BGR2RGB) # Changement d'
# Affichage avec matplotlib
plt.imshow(rgb_img)
plt.title("original image " + img_path)
plt.show()

# Affichage de l'image convertie
rgb_processed_img = cv2.cvtColor(processed_img, cv2.COLOR_BGR2RGB) # Ch
plt.imshow(rgb_processed_img)
plt.title("processed image " + img_path)
plt.show()
```

Construction des données d'entraînement, validation et test

Comme dans le notebook précédent, nous allons directement créer les matrices d'images et d'étiquettes qui seront utilisées par le modèle. Ceci nous permettra d'utiliser la fonction `train_test_split` de la librairie `sklearn` .

La première étape est de construire des matrices X et y contenant toutes les images

prétraitées et les étiquettes. Nous utiliserons ensuite la fonction `train_test_split` pour les séparer en images à utiliser pour l'entraînement, la validation et les tests.

La **matrice X** contient toutes les images. Elle est de dimension nombre d'images * hauteur * largeur * nombre de canaux.

La **matrice y** contient les caractéristiques de chaque image. Nous allons représenter chaque caractéristique par un vecteur de taille 2 : (1, 0) pour les chats, et (0, 1) pour les chiens. Cette représentation, nommée "encodage à chaud" ("one hot encoding" en anglais), est fréquemment utilisée en apprentissage automatique. Cette matrice est de dimensions nombre d'images * nombre de classes

In [18]:
```
nb_images_per_pet = 330
source_dir = 'PetImages'

# On initialise la liste X_list, qui contiendra chaque image sous forme
# et la liste y, qui contiendra chaque étiquette sous forme d'un entier
X_list = []
y_list = []

# On considère chacun des animaux successivement
for class_id, pet_dir in enumerate([cat_dir, dog_dir]):
    print("Traitement des images du dossier", pet_dir)
    print("Ces images ont l'étiquette", [class_id, 1-class_id])

    # Création d'une liste contenant les noms de toutes les images de c
    all_pet_filenames = os.listdir(os.path.join(source_dir, pet_dir))

    # On mélange la liste avant d'en extraire une portion. La fonction
    random.shuffle(all_pet_filenames)

    # On extrait les nb_images_per_pet premières images de la liste
    subset = all_pet_filenames[:nb_images_per_pet]
    for file_name in subset:
        img_path = os.path.join(source_dir, pet_dir, file_name)
        square_image = resize_image(img_path, img_size)
        if square_image is not None:
            # On l'ajoute à la liste
            X_list.append(square_image)
            y_list.append(class_id)

# On mélange les deux listes de la même manière
xy = list(zip(X_list, y_list))
random.shuffle(xy)
X_tuple, y_tuple = zip(*xy)

# On convertit X_tuple depuis un tuple de vecteurs en une matrice numpy
X = np.stack(X_tuple, axis=0)

# On encode y_tuple avec un encodage à chaud.
y = np.asarray(y_tuple)
y = np.stack([y, 1-y], axis=-1)
```

```
Traitement des images du dossier Cat
Ces images ont l'étiquette [0, 1]
Traitement des images du dossier Dog
Ces images ont l'étiquette [1, 0]
could not read image  PetImages\Dog\2688.jpg
```

In [19]:
```
# Vérification des dimensions de X et de y
print("X.shape", X.shape)
print("y.shape", y.shape)
```

```
X.shape (659, 128, 128, 3)
y.shape (659, 2)
```

On peut maintenant utiliser la fonction `train_test_split` de la librairie `sklearn` .

Cette fonction permet de séparer une matrice d'images et la matrice d'étiquettes correspondante en deux ensembles de données, dans une proportion donnée en argument. Les arguments obligatoire de la fonction sont les suivants :

- matrice contenant les images,
- matrice ou vecteur contenant les étiquettes.

Il est également possible de spécifier plusieurs arguments optionnels :

- `test_size` : proportion de données à utiliser dans le second ensemble de données,
- `shuffle` : ce booléen détermine s'il faut, ou non, mélanger les données avant de les séparer en deux groupes. Par défaut, il est mis à `True` ,
- `random_state` : ce paramètre permet d'initialiser le générateur de nombre aléatoires d'une manière reproductible, ce qui permet d'obtenir le même résultat lorsqu'on exécute plusieurs fois la fonction,
- `stratify` : si cet argument vaut `None` (ce qui est la valeur par défaut), alors les données sont séparées de manière aléatoire. Si on donne la liste des étiquettes comme valeur à cet argument, les données seront séparées de manière stratifiées, c'est à dire que chaque ensemble de données contiendra la même proportion d'images de chacune des classes.

La fonction `train_test_split` ne permettant de séparer un ensemble de données qu'en deux parties, nous l'exécutons deux fois :

- une première fois pour séparer les images *entraînement* + *validation* des images de *test*, avec une proportion de 80% pour le premier ensemble et 20% pour le second,
- une deuxième fois pour séparer les images *entraînement* + *validation* renvoyées par la première exécution en images d'*entraînement* et images de *validation*, avec encore une fois une proportion de 80% pour le premier ensemble et 20% pour le second.

In [20]:
```
# Séparation des images en train + val et test
X_train_val, X_test, y_train_val, y_test = sklearn.model_selection.trai



```

In [21]:
```
# Affichage des dimensions des matrices créées
print("X_train_val.shape", X_train_val.shape)
print("y_train_val.shape", y_train_val.shape)
print("X_test.shape", X_test.shape)
print("y_test.shape", y_test.shape)
```

```
X_train_val.shape (527, 128, 128, 3)
y_train_val.shape (527, 2)
X_test.shape (132, 128, 128, 3)
y_test.shape (132, 2)
```

L'ensemble "entraînement + validation" contient 80% des 660 images de départ, soit 528 images. La matrice X_train_val a donc effectivement pour taille nombre d'images * hauteur * largeur * nombre de canaux, i.e. 421 * 128 * 128 * 1.

Les étiquettes étant encodées à chaud, la matrice y_train_val a pour taille nombre d'images * nombre de classes, i.e. 421 * 2.

L'ensemble de test contient 20% des 660 images de départ, soit 132 images. La matrice X_test a donc effectivement pour taille nombre d'images * hauteur * largeur * nombre de canaux, i.e. 132 * 128 * 128 * 1.

Les étiquettes étant encodées à chaud, la matrice y_test a pour taille nombre d'images * nombre de classes, i.e. 132 * 2.

In [22]:
```
X_train, X_val, y_train, y_val = sklearn.model_selection.train_test_spl

print("X_train.shape", X_train.shape)
print("y_train.shape", y_train.shape)
print("X_val.shape", X_val.shape)
print("y_val.shape", y_val.shape)
```

```
X_train.shape (421, 128, 128, 3)
y_train.shape (421, 2)
X_val.shape (106, 128, 128, 3)
y_val.shape (106, 2)
```

L'ensemble d'entraînement contient 80% des 528 images de l'ensemble "entraînement + validation", soit 422 images. La matrice X_train a donc effectivement pour taille nombre d'images * hauteur * largeur * nombre de canaux, i.e. 422 * 128 * 128 * 1. La matrice y_train a pour taille nombre d'images * nombre de classes, i.e. 422 * 2.

L'ensemble de validation contient 20% de ces images, soit 105 images. La matrice X_val a donc effectivement pour taille nombre d'images * hauteur * largeur * nombre de canaux, i.e. 105 * 128 * 128 * 1. La matrice y_train a pour taille nombre d'images * nombre de classes, i.e. 105 * 2.

Choisir et charger un modèle

`Keras` propose plusieurs modèles préentraînés sur ImageNet. Un modèle préentraîné est composé de deux parties : l'architecture du modèle, et les poids. Les fichiers de poids étant très lourds, ils ne sont pas directement inclus dans Keras, mais sont téléchargés lors de l'instanciation d'un modèle préentraîné.

Les modèles suivants sont disponibles :

- `VGG16` ;
- `InceptionV3` ;
- `ResNet` ;
- `MobileNet` ;
- `Xception` ;
- `InceptionResNetV2` .

Tous ces réseaux sont entraînés sur ImageNet, ils sont donc entraînés à prédire l'appartenance d'une image à une classe parmi les 1000 classes de la base de données. La dernière couche de chacun de ces réseaux est une couche complètement connectée contenant 1000 neurones.

Afin d'adapter ces réseaux pour notre usage, nous allons charger toutes les couches, sauf la dernière couche complètement connectée. Nous remplacerons cette dernière par une couche complètement connecté à deux neurones, car nous avons deux classes.

Dans ce notebook, nous allons utiliser le réseau `InceptionV3` .

Commençons par charger ce réseau, afin de visualiser son architecture.

Pour cela, nous utilisons la fonction `keras.applications.inception_v3.InceptionV3` , qui accepte les arguments suivants :

- `weights` : si cet argument est `imagenet` (ce qui est la valeur par défaut), alors les poids du réseau préentraîné sur ImageNet sont chargés. Si cet argument est `None` , alors les poids sont initialisés aléatoirement. On peut aussi charger d'autres poids, on donnant le chemin vers ces poids dans cet argument ;
- le booléen `include_top` vaut `True` si la dernière couche complètement connectée du réseau doit être chargée, `False` sinon. Notons que si la dernière couche est chargée (c'est alors le réseau dans sa totalité qui est chargé), ceci impacte les entrées et les sorties du réseau : les images d'entrée doivent obligatoirement avoir la taille prévue par le réseau, et la sortie sera un vecteur de 1000 éléments.
- l'argument `input_shape` , qui permet de spécifier la taille des images d'entrée, ne doit être utilisé que si l'argument `include_top` vaut `False` .

In [23]:

```
# Chargement et visualisation du réseau entier
inception_model_top = keras.applications.inception_v3.InceptionV3(weigh
    include_top=True)

# Visualisation de l'architecture du réseau
print(inception_model_top.summary())
```

```
Model: "inception_v3"
__________________________________________________________________________
_____________________
Layer (type)                    Output Shape         Param #     Connected
to
==========================================================================
=======================
input_1 (InputLayer)            [(None, 299, 299, 3) 0
__________________________________________________________________________
_____________________
conv2d (Conv2D)                 (None, 149, 149, 32) 864         input_1
[0][0]
__________________________________________________________________________
_____________________
batch_normalization (BatchNorma (None, 149, 149, 32) 96          conv2d
[0][0]
__________________________________________________________________________
_____________________
activation (Activation)         (None, 149, 149, 32) 0           batch_nor
```

On voit que la dernière couche, nommé `predictions (Dense)` donne en sortie un vecteur de 1000 éléments.

Chargeons maintenant le modèle sans cette dernière couche, afin de voir la différence.

In [24]:
```
# Chargement et visualisation du réseau sans la dernière couche
inception_model = keras.applications.inception_v3.InceptionV3(weights='
    input_shape=(img_size, img_size, 3),
    include_top=False)

# Visualisation de l'architecture du réseau
print(inception_model.summary())
```

```
Model: "inception_v3"
__________________________________________________________________________
_____________________
Layer (type)                    Output Shape         Param #     Connected
to
==========================================================================
=======================
input_2 (InputLayer)            [(None, 128, 128, 3) 0
__________________________________________________________________________
_____________________
conv2d_94 (Conv2D)              (None, 63, 63, 32)   864         input_2
[0][0]
__________________________________________________________________________
_____________________
batch_normalization_94 (BatchNo (None, 63, 63, 32)   96          conv2d_94
[0][0]
__________________________________________________________________________
_____________________
activation_94 (Activation)      (None, 63, 63, 32)   0           batch_nor
```

Ce réseau a deux couches de moins que le réseau complet : il lui manque la toute dernière couche, complètement connectée, ainsi que l'avant-dernière couche, qui réalise une agrégation par la moyenne.

A partir de ce réseau, recréons maintenant un réseau complet pour faire de la classification à deux classes.

Pour cela, on commence par geler les poids de ce modèle. En effet, dans un premier temps, nous n'allons entraîner que les couches additionnelles.

In [25]:
```
# Gel des poids du modèle de base inception_model
inception_model.trainable = False
```

Nous allons maintenant ajouter des couches à ce modèle.

La première couche à créer est la couche d'entrée. Elle est créée grâce à la fonction `keras.Input`, qui prend comme unique argument la taille des images.

In [26]:
```
# Création de la couche d'entrée
inputs = keras.Input(shape=(img_size, img_size, 3))
```

Le réseau `inception_model` attend en entrée des images dont les intensités sont comprises entre -1 et 1.

Nous faisons donc suivre la couche d'entrée d'une couche de prétraitement, qui va ajuster les intensités comme il convient. Cette couche utilise la fonction `keras.layers.Rescaling`, qui prend deux arguments :

- `scale` est un facteur multiplicatif à appliquer aux intensités,
- `offset` est un facteur additif.

L'intensité de chaque pixel en sortie de cette couche vaut : intensité_originale × `scale` + `offset`

Afin d'obtenir des intensités entre -1 et 1 à partir d'intensités entre 0 et 255, nous utilisons les valeurs suivantes :

- `scale` = 1/127.5
- `offset` = -1

La sortie de ce réseau est stockée dans la variable `scaled images` .

In [27]:
```
# Couche de prétraitement pour ajuster les intensités
scale_layer = keras.layers.Rescaling(scale=1 / 127.5, offset=-1)
scaled_images = scale_layer(inputs)
```

Nous pouvons maintenant ajouter les couches suivantes :

- le **réseau de base** `inception_model` , qui prend en entrée les images originales prétraitées stockées dans la variable `scaled_images` . Comme ce réseau contient des couches de normalisation par lot, nous lui donnons comme argument `training=False` . Le réseau est ainsi en mode "inférence", et ces couches ne seront pas modifiées. Ce réseau est stocké dans la variable `inception` ;
- une **couche d'agrégation par moyennage**, définie avec la fonction `keras.layers.GlobalAveragePooling2D` . Cette fonction ne nécessite aucun argument. On l'applique à la sortie de la couche précédente, et on stocke la couche ainsi définie dans la variable `average_pooling` ;
- une **couche de dropout**, afin de limiter le suraprentissage. Cette couche est définie avec la fonction `keras.layers.Dropout` , qui prend en argument la fraction de noeuds à désactiver à chaque époque. On choisit ici 0.2. On l'applique à la sortie de la couche précédente, et on stocke cette couche dans la variable `dropout` ;
- une **couche complètement connectée**, avec deux neurones, définie avec la fonction `keras.layers.Dense` . Cette fonction prend comme arguments le nombre de neurones, ici deux. On l'applique à la sortie de la couche précédente, et on stocke cette couche dans la variable `dense` ;
- enfin, une couche **Softmax** permet de convertir les sorties de la couche complètement connectée en probabilités. Cette couche est créée avec la fonction `tf.keras.layers.Softmax` , qui ne nécessite aucun argument en entrée. On l'applique à la sortie de la couche précédente, et on stocke cette couche dans la variable `outputs` .

Remarque : attention à ne pas confondre l'attribut `trainable` , qu'on peut appliquer à un réseau entier ou à une seule couche pour signifier que ses poids doivent être optimisés pendant l'apprentissage, et l'attribut `training` , qui ne s'applique qu'à un réseau entier et qui détermine si ce réseau fonctionne en mode "apprentissage" ou en mode "inférence".

In [28]:
```
inception = inception_model(scaled_images, training=False)
average_pooling = keras.layers.GlobalAveragePooling2D()(inception)
dropout = keras.layers.Dropout(0.2)(average_pooling)
dense = keras.layers.Dense(2)(dropout)
outputs = tf.keras.layers.Softmax()(dense)
```

Nous pouvons maintenant créer un modèle à partir de toutes ces couches.

La fonction `Model` de `Keras` prend deux arguments : la couche d'entrée, et la couche de sortie, et créé un modèle.

In [29]:
```
model = keras.Model(inputs, outputs)

# Affichage du modèle créé
model.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, 128, 128, 3)]     0
_________________________________________________________________
rescaling (Rescaling)        (None, 128, 128, 3)       0
_________________________________________________________________
inception_v3 (Functional)    (None, 2, 2, 2048)        21802784
_________________________________________________________________
global_average_pooling2d (Gl (None, 2048)              0
_________________________________________________________________
dropout (Dropout)            (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 2)                 4098
_________________________________________________________________
softmax (Softmax)            (None, 2)                 0
=================================================================
Total params: 21,806,882
Trainable params: 4,098
Non-trainable params: 21,802,784
_________________________________________________________________
```

On voit que seule une petite partie des paramètres est définie comme étant entraînable. Les premières couches du réseau Inception sont donc bien considérées comme des couches non entraînables.

Entraînement du réseau

L'entraînement du réseau va se faire en deux étapes.

On va commencer par entraîner seulement la dernière couche, i.e. la couche complètement connectée qu'on a ajoutée au réseau initial. On évaluera les performances du modèle après cette première étape.

On entraînera ensuite toutes les couches du réseau, en utilisant un taux d'apprentissage très faible, et on évaluera à nouveau les performances du modèle pour voir ce que cette seconde étape a apporté.

Entraînement de la dernière couche

Lors de la création du modèle, nous avons déjà spécifié quelles couches devaient être entraînées. Nous pouvons maintenant entraîner le modèle comme d'habitude ; seuls les paramètres de la dernière couche complètement connectée seront optimisés.

Pour cela, il faut tout d'abord compiler le modèle, afin de spécifier des informations telles que :

- La fonction d'optimisation,
- La fonction-coût,
- D'éventuelles métriques additionnelles à enregistrer à chaque époque.

Ici, nous utilisons la fonction Adam pour l'optimisation, avec un taux d'apprentissage de 10^-4, et la fonction-coût d'entropie croisée binaire, qui est le choix le plus courant pour réaliser une classification binaire. En plus de la fonction-coût, nous demandons au réseau de calculer la précision à chaque époque.

Après compilation, l'entraînement du modèle se fait en utilisant la fonction `fit` de Keras. Les paramètres de cette fonction sont :

- Les images de test,
- Les étiquettes de test,
- Le nombre d'époques pendant lequel réaliser l'entraînement,
- La taille de lot à utiliser.

Ici, nous allons entraîner le modèle pendant 20 époques, avec une taille de lot de 8.

In [30]:
```
# Compilation du modèle
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=1e-4),
    loss=keras.losses.CategoricalCrossentropy(),
    metrics=['accuracy']
)

epochs = 20
history_step1 = model.fit(X_train, y_train, validation_data=(X_val, y_v
```

```
Epoch 1/20
53/53 [==============================] - 26s 240ms/step - loss: 2.1312 - a
ccuracy: 0.4608 - val_loss: 1.2841 - val_accuracy: 0.4717
Epoch 2/20
53/53 [==============================] - 5s 90ms/step - loss: 1.0235 - acc
uracy: 0.6485 - val_loss: 0.6790 - val_accuracy: 0.6509
Epoch 3/20
53/53 [==============================] - 7s 140ms/step - loss: 0.7891 - ac
curacy: 0.7482 - val_loss: 0.4954 - val_accuracy: 0.7830
Epoch 4/20
53/53 [==============================] - 6s 115ms/step - loss: 0.5779 - ac
curacy: 0.8005 - val_loss: 0.3912 - val_accuracy: 0.7925
Epoch 5/20
53/53 [==============================] - 6s 120ms/step - loss: 0.5208 - ac
curacy: 0.8242 - val_loss: 0.3364 - val_accuracy: 0.8491
Epoch 6/20
53/53 [==============================] - 5s 89ms/step - loss: 0.5091 - acc
uracy: 0.8314 - val_loss: 0.3473 - val_accuracy: 0.8585
Epoch 7/20
```

La fonction `fit` renvoie en sortie l'historique des métriques évaluées pendant le processus d'optimisation, sous la forme d'un dictionnaire.

Commençons par afficher les différentes clefs sauvegardées.

In [31]:
```
# Liste des données enregistrées dans l'historique d'optimisation
print(history_step1.history.keys())
```

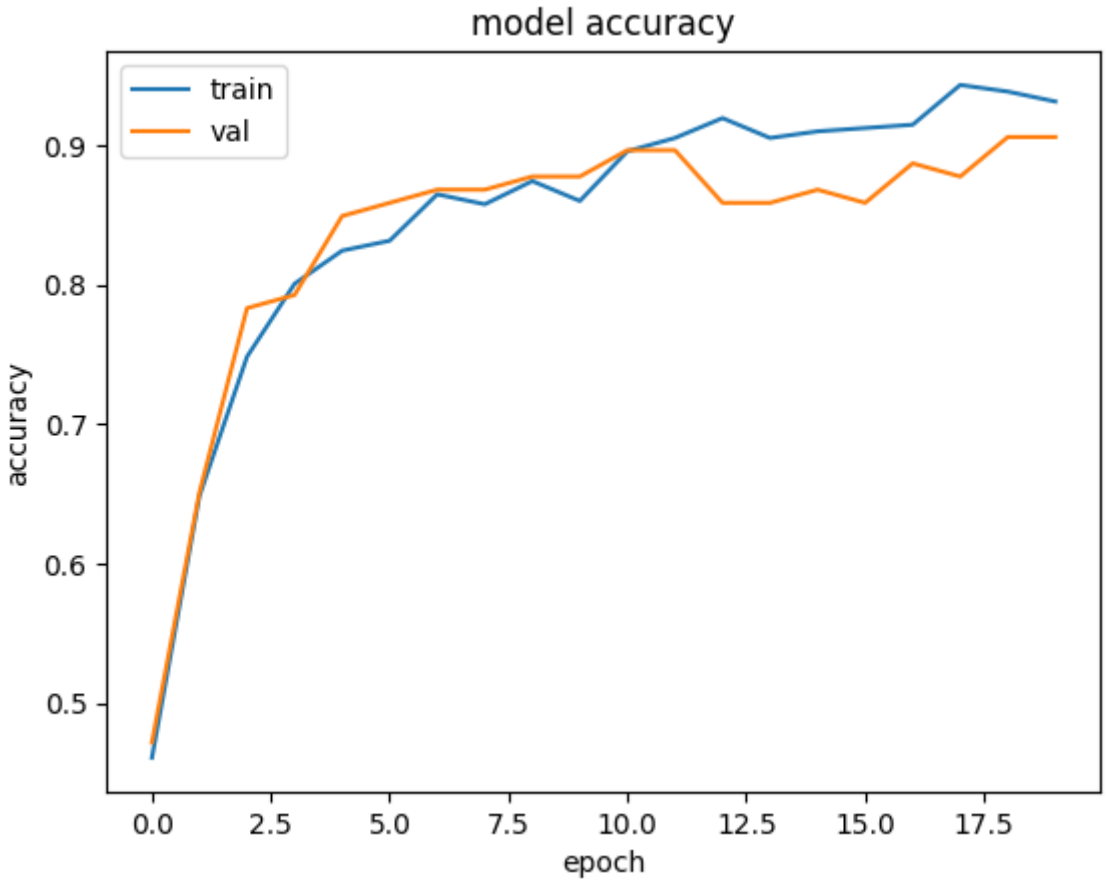
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

On a enregistré les valeurs de la fonction-coût et de la précision des données d'entraînement et de validation.

On peut tracer ces valeurs en fonction du nombre d'itérations pour mieux visualiser l'entraînement.

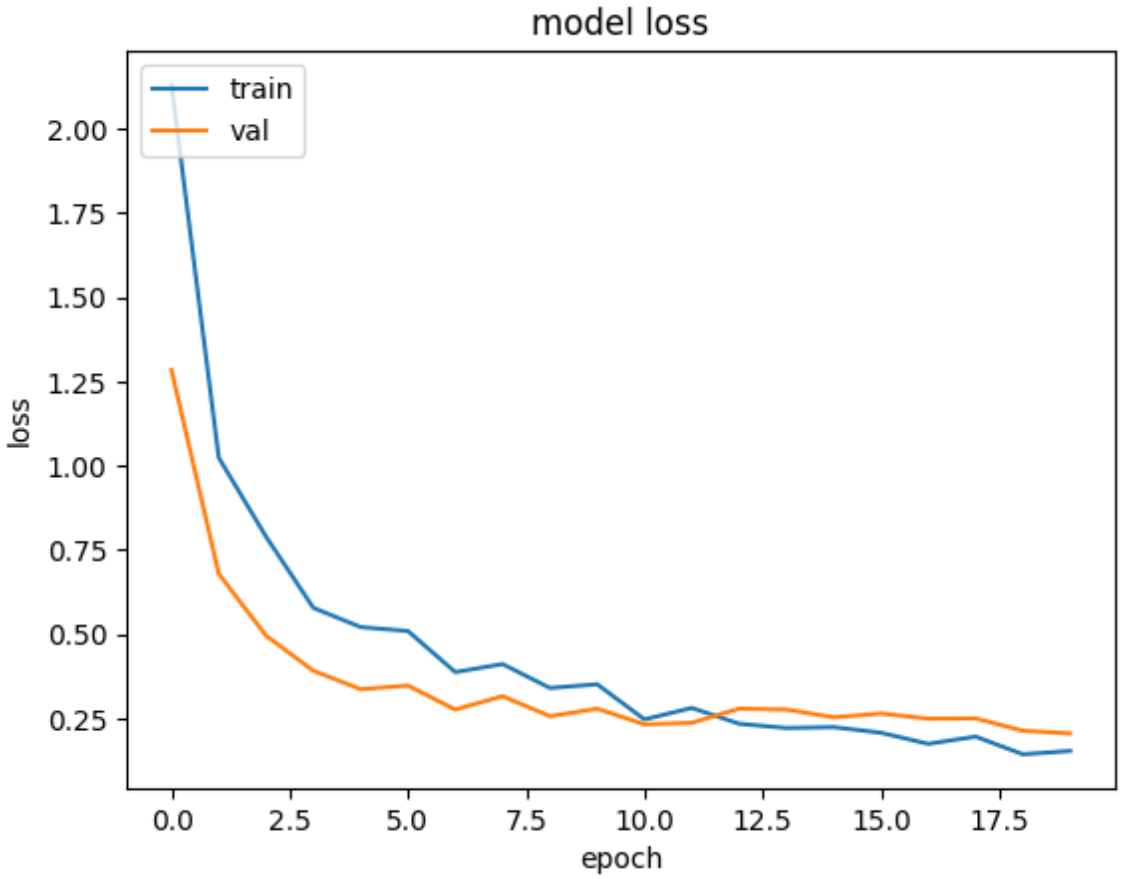
In [32]:

```
# Affichage de l'accuracy
plt.plot(history_step1.history['accuracy'])
plt.plot(history_step1.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```



La performance est systématiquement moins bonne sur les données de validation que sur les données d'entraînement, ce qui est attendu.

In [33]:
```
# Affichage de la fonction coût
plt.plot(history_step1.history['loss'])
plt.plot(history_step1.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```



Au cours de itérations, la fonction-coût des deux ensembles de données, entraînement et validation, décroît.

Evaluation du classifieur après entraînement de la dernière couche

Nous pouvons maintenant calculer les valeurs de prévision, spécificité et score F1 pour les images de test, en utilisant la fonction `classification_report` de la bibliothèque `sklearn` .

Cette fonction nécessite que les étiquettes soient encodées au format binaire (0 pour les chiens et 1 pour les chats). Nous devons donc convertir les étiquettes prédites et réelles depuis le format à chaud vers ce format. Pour cela, nous utilisons la fonction `argmax` de `numpy` , qui appliquée à un vecteur, renvoie l'indice qui contient la valeur maximale.

Par exemple, considérons un vecteur de prédiction $y = [0.67, 0.33]$, qui correspond à un chien avec l'encodage à chaud.

L'indice contenant la valeur maximale est l'indice 0, ce qui correspond bien à la classe du

chien avec l'encodage au format binaire.

Nous calculerons également une matrice de confusion pour visualiser les résultats

In [34]:
```
# Calcul des prédictions pour les images contenues dans X_train
y_pred = model.predict(X_test)

# Conversion des prédictions et des étiquette réelles au format binaire
y_pred_bin = np.argmax(y_pred, axis=-1)
y_test_bin = np.argmax(y_test, axis=-1)

# Calcul des métriques en utilisant la bibliothèque sklearn
print(sklearn.metrics.classification_report(y_test_bin, y_pred_bin))
```

```
              precision    recall  f1-score   support

           0       0.92      0.89      0.91        66
           1       0.90      0.92      0.91        66

    accuracy                           0.91       132
   macro avg       0.91      0.91      0.91       132
weighted avg       0.91      0.91      0.91       132
```

La bibliothèque `sklearn` a aussi une fonction qui permet de tracer la matrice de confusion.

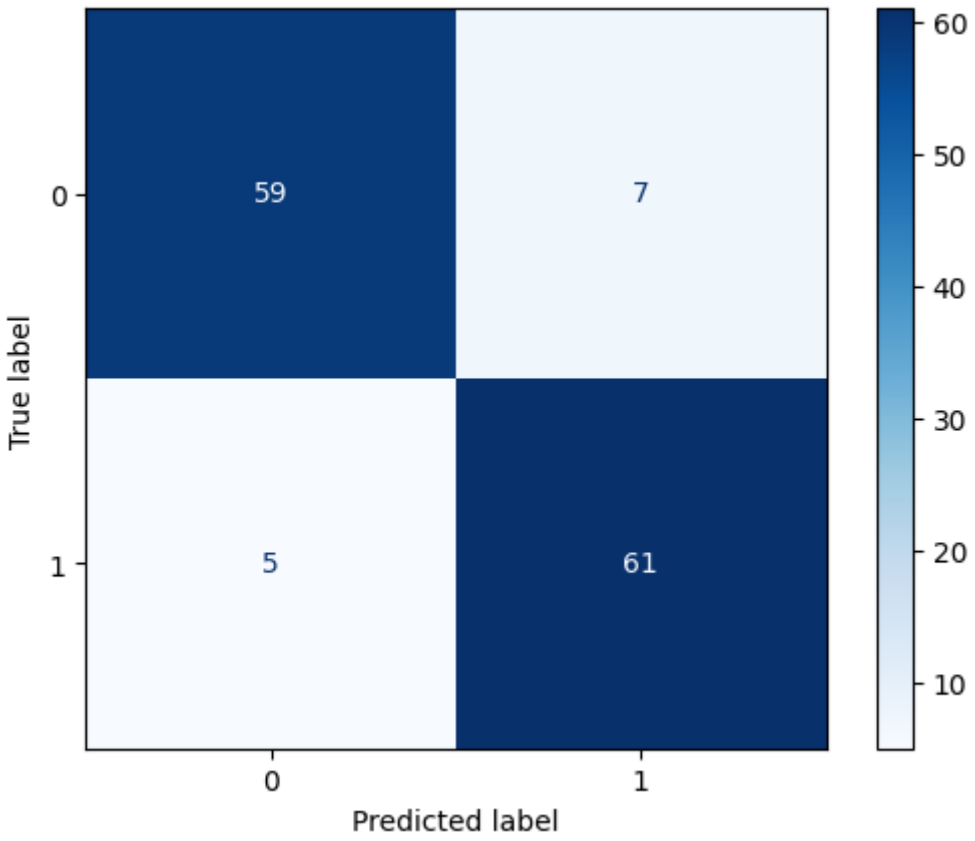
Il s'agit de la fonction `ConfusionMatrixDisplay.from_predictions` , qui prend en entrée deux arguments obligatoires :

- une liste d'étiquettes réelles au format binaire ;
- une liste d'étiquettes prédites au format binaire.

Cette fonction accepte également plusieurs arguments facultatifs. Ici, on précise une colormap bleue, qui est différente de celle choisie par défaut, mais conforme à la manière dont sont généralement affichées les matrices de confusion.

In [35]:
```
1 sklearn.metrics.ConfusionMatrixDisplay.from_predictions(y_test_bin, y_p
```

Out[35]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1bdc59
d5eb0>
```



Les résultats obtenus en réetraînant uniquement la dernière couche sont déjà très bons. Voyons si on peut encore les améliorer en réentraînant l'ensemble du modèle.

Entraînement de toutes les couches

Pour entraîner l'ensemble du modèle, nous devons spécifier que toutes les couches sont entraînables.

Ceci se fait avec l'instruction suivante :

In [36]:
```
1 # Dégel du modèle de base
2 inception_model.trainable = True
```

Remarque : Si nous le voulions, nous pourrions choisir de n'entraîner que les n dernières couches du réseau. L'attribut `trainable` peut également s'appliquer à une couche, plutôt qu'au réseau entier.

Affichons la structure du modèle pour vérifier que toutes les couches sont bien entraînables.

In [37]:
```
1 # Affichage de la structure du modèle
2 model.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, 128, 128, 3)]     0
_________________________________________________________________
rescaling (Rescaling)        (None, 128, 128, 3)       0
_________________________________________________________________
inception_v3 (Functional)    (None, 2, 2, 2048)        21802784
_________________________________________________________________
global_average_pooling2d (Gl (None, 2048)              0
_________________________________________________________________
dropout (Dropout)            (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 2)                 4098
_________________________________________________________________
softmax (Softmax)            (None, 2)                 0
=================================================================
Total params: 21,806,882
Trainable params: 21,772,450
Non-trainable params: 34,432
_________________________________________________________________
```

Le modèle contient plus de 21 000 000 paramètres entraînables, contre 4 098 à l'étape précédente. Les paramètre du modèle de base sont donc bien considérés comme des paramtères entraînables.

Par contre, nous ne voulons pas mettre à jour les couches de normalisation par lot du réseau de base. En effet, il est important qu'elles gardent en mémoire les statistiques d'ImageNet. Comme nous avons spécifié `training=False` lors de sa création, il reste en mode "inférence", et ces couches ne seront pas modifiées. Nous n'avons rien à modifier de plus.

Comme précédemment, nous pouvons maintenant compiler le modèle. C'est à cette étape que nous spécifions la fonction d'optimisation et le taux d'apprentissage utilisés.

Ici, nous utilisons la fonction Adam pour l'optimisation, avec un taux d'apprentissage de 10^-5. Il est courant de choisir un taux d'apprentissage dix fois plus petit pour l'optimisation de l'ensemble du réseau que pour l'optimisation de la dernière couche seule.

Après compilation, l'entraînement du modèle se fait en utilisant la fonction `fit` de Keras.

Ici, nous allons entraîner le modèle pendant 10 époques, avec une taille de lot de 8.

In [38]:

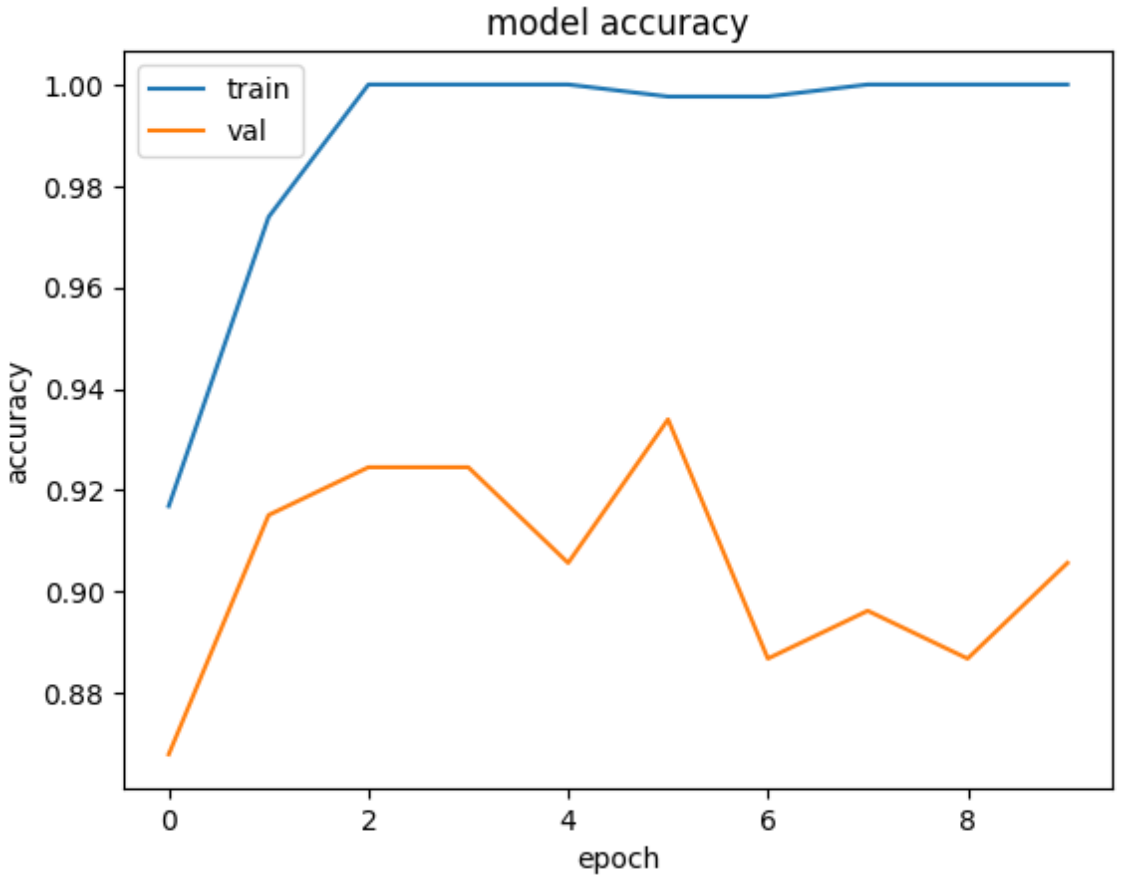
```
model.compile(
    optimizer=keras.optimizers.Adam(1e-5),  # Low learning rate
    loss=keras.losses.CategoricalCrossentropy(),
    metrics=['accuracy'],
)

epochs = 10
history_step2 = model.fit(X_train, y_train, validation_data=(X_val, y_v
```

```
Epoch 1/10
53/53 [==============================] - 29s 313ms/step - loss: 0.2430 - a
ccuracy: 0.9169 - val_loss: 0.3710 - val_accuracy: 0.8679
Epoch 2/10
53/53 [==============================] - 9s 171ms/step - loss: 0.0700 - ac
curacy: 0.9739 - val_loss: 0.2527 - val_accuracy: 0.9151
Epoch 3/10
53/53 [==============================] - 9s 172ms/step - loss: 0.0061 - ac
curacy: 1.0000 - val_loss: 0.2517 - val_accuracy: 0.9245
Epoch 4/10
53/53 [==============================] - 10s 181ms/step - loss: 0.0022 - a
ccuracy: 1.0000 - val_loss: 0.2225 - val_accuracy: 0.9245
Epoch 5/10
53/53 [==============================] - 9s 176ms/step - loss: 0.0040 - ac
curacy: 1.0000 - val_loss: 0.2903 - val_accuracy: 0.9057
Epoch 6/10
53/53 [==============================] - 10s 186ms/step - loss: 0.0046 - a
ccuracy: 0.9976 - val_loss: 0.1849 - val_accuracy: 0.9340
Epoch 7/10
```

Affichons maintenant l'historique des valeurs d'accuracy et de fonction coût évaluées pendant le processus d'optimisation.

In [39]:

```
# Affichage de l'accuracy
plt.plot(history_step2.history['accuracy'])
plt.plot(history_step2.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```

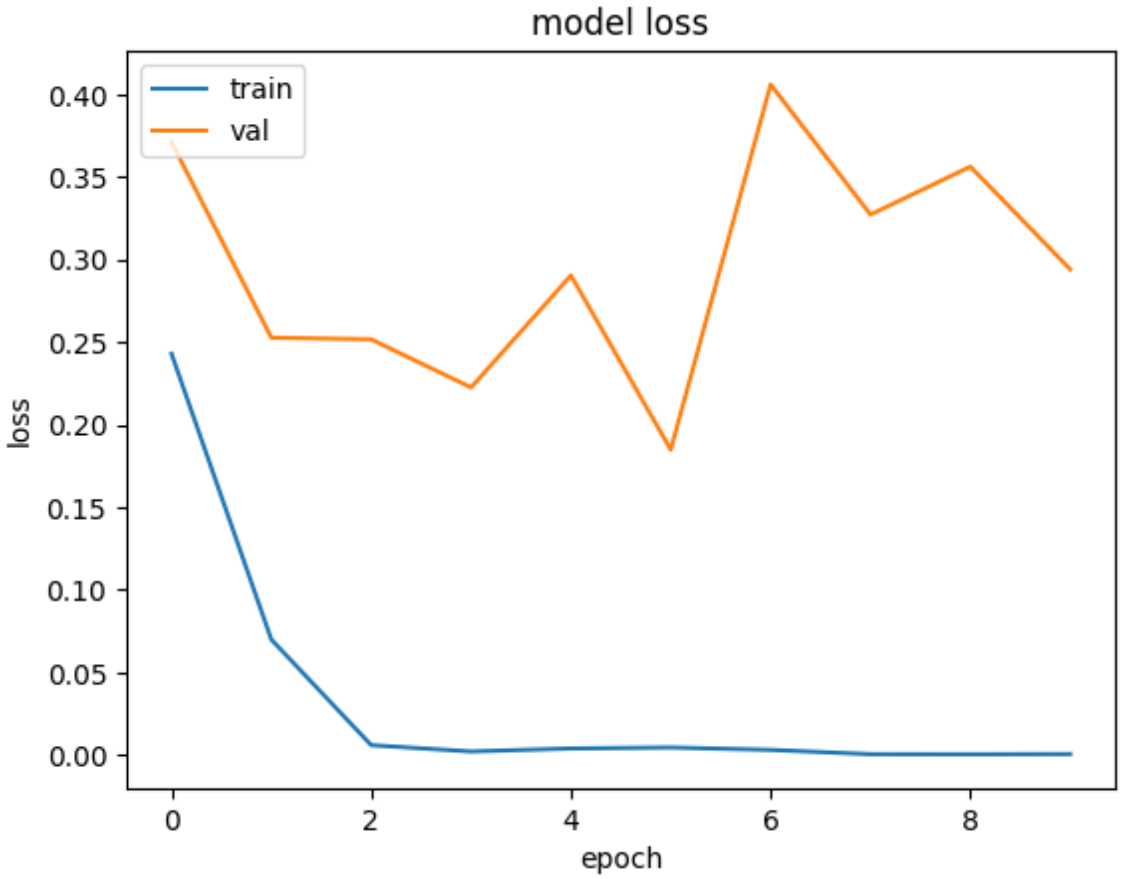


La performance est systématiquement moins bonne sur les données de validation que sur les données d'entraînement, ce qui est attendu.

Cependant, l'ampleur de l'écart nous indique que le réseau est probablement en situation de suraprentissage.

In [40]:

```
# Affichage de la fonction coût
plt.plot(history_step2.history['loss'])
plt.plot(history_step2.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```



Au cours de itérations, la fonction-coût des données d'entraînement décroît, tandis que celle des données de validation croît : il y a donc du surapprentissage.

Evaluation du classifieur après entraînement de toutes les couches

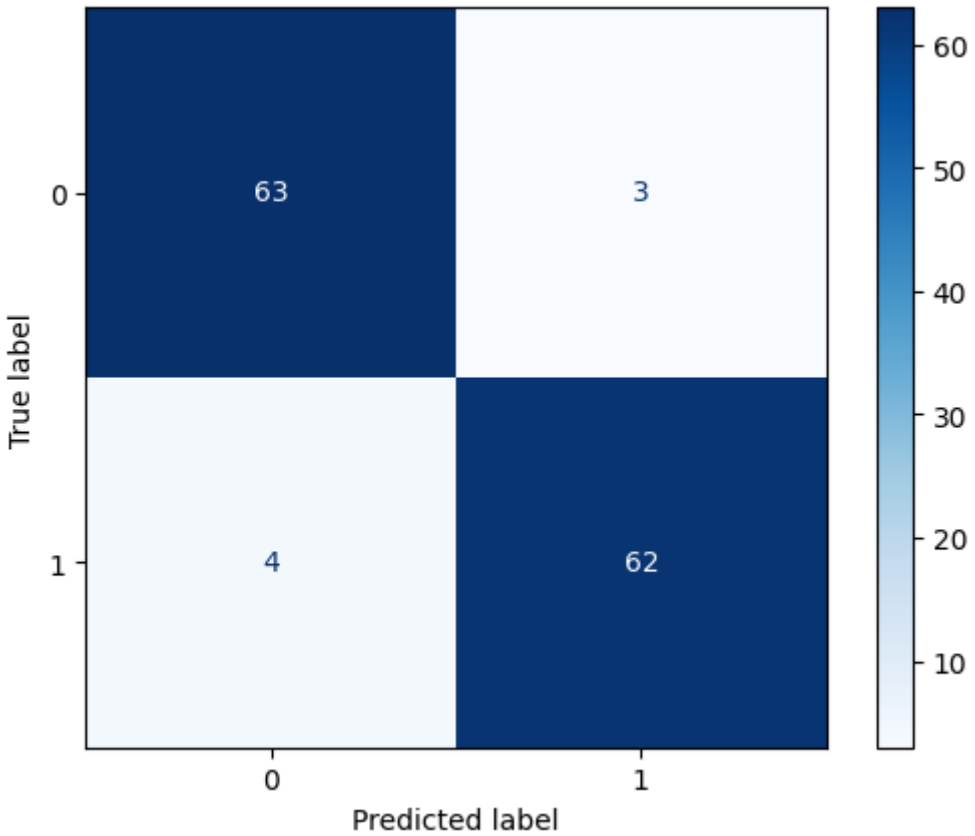
Nous pouvons maintenant calculer les valeurs de prévision, spécificité et score F1 pour les images de test, et calculer une matrice de confusion pour visualiser les résultats.

In [41]:
```
# Calcul des prédictions pour les images contenues dans X_test
y_pred = model.predict(X_test)

# Conversion des prédictions et des étiquette réelles au format binaire
y_pred_bin = np.argmax(y_pred, axis=-1)
y_test_bin = np.argmax(y_test, axis=-1)

# Calcul des métriques en utilisant la bibliothèque sklearn
print(sklearn.metrics.classification_report(y_test_bin, y_pred_bin))
```

```
              precision    recall  f1-score   support

           0       0.94      0.95      0.95        66
           1       0.95      0.94      0.95        66

    accuracy                           0.95       132
   macro avg       0.95      0.95      0.95       132
weighted avg       0.95      0.95      0.95       132
```

In [42]:
```
# Calcul et affichage de la matrice de confusion
sklearn.metrics.ConfusionMatrixDisplay.from_predictions(y_test_bin, y_p
```

Out[42]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1bdfa1
b3130>
```



Malgré le surapprentissage, réentraîner toutes les couches du réseau a permis d'augmenter ses performances.

Les résultats finaux sont très bons. Ceci s'explique par le fait que notre ensemble de données n'est pas très éloigné de la base de données sur laquelle est entraîné le réseau de base. En effet, ImageNet contient plusieurs classes d'espèces de chiens et de chats. Même

avant son adaptation, notre réseau de base était donc déjà entraîné pour reconnaître des chats et des chiens.

Conclusion

Cet exercice nous a permis d'aborder plusieurs sujets :

- Le chargement d'un réseau préentraîné ;
- La recréation d'une couche de classification adaptée à notre problème ;
- L'entraînement en deux étapes (dernière couche seule, puis toutes les couches).

A vous de jouer !

Voici quelques propositions d'exercices que vous pouvez réaliser pour être plus à l'aise avec TensorFlow et Keras :

- Utilisez un autre réseau pré-entraîné ;
- Modifiez les couches ajoutées à la fin du réseau de base : comment évolue la performance si vous ajoutez plusieurs couches complètement connectées, avec différents nombres de neurones dans chacune (remarquez que la dernière couche complètement connectée doit toujours contenir deux neurones) ?
- Comment se comporte le réseau si vous supprimez la couche de drop-out en sortie du réseau ?
- Dans la seconde étape de l'entraînement, réentrainez uniquement quelques couches du réseau, et pas la totalité ;
- Utilisez un autre ensemble de données pour y adapter le réseau.

In [ ]:
```
1
```