Neural Networks: Cats vs. Dogs

Statistical Methods for Machine Learning

Simone Giachetta - 956536

Università di Milano - Statale

February 5, 2023

Contents

1 Abstract

This work shows a possible implementation of a Neural Network in an image recognition use case. Specifically, the Neural Network will be a convolutional one. The goal of this work is build a model which predict if an image shows a cat or a dog. Different network architectures and training parameters will be taken in consideration, explaining their influences in the results. 5-fold cross validation will be the procedure to compute the risk estimates while the reported cross-validated estimates is computed according to the zero-one loss. After a theoretical introduction of the algorithm used, there will be an implementation using Python and the TensorFlow deep learning framework: in particular, Keras, an API running on top of TensorFlow, helps to exploit the scalability and capabilities of the framework.

2 Introduction

Neural networks are algorithms that were specifically designed to be inspiration for biological neural networks. Neurons interconnected according to network type form the basis of neural networks. The technique is named after the human neural networks that inspired it. The goal was to create an artificial system that functioned exactly like the human brain. Neural Networks are one of the most well-known techniques in the machine learning field, if not the most popular in the Deep Learning subfield. The neural networks are made up of levels of nodes, each of which has a level of outputs and one or more hidden levels of outputs. Each node connects to another node and is assigned a weight and a threshold. If the output of any single node exceeds its threshold, that node is activated and begins sending data to the next level of nodes. If the value is less than the threshold, no data will be sent. To learn and improve accuracy, neural networks rely on training data which are the data used to train an algorithm or machine learning model to predict the outcome.

Based on the use case, Neural Networks can be classified in different types. The most common ones are:

- **Feedforward Neural Networks**: These networks were created to answer issues that linear regression approaches failed to handle. When the perceptron, the parent of neural networks, was constructed, regression models were already well established in the industry and and permitted the extraction of linear correlations between variables. Feedforward neural networks are node networks that transfer a linear combination of inputs from one layer to the next. The nodes determine how to change their inputs while doing so, using a particular activation function. The activation function of a neuron is crucial in this case as neural network can include non-linearity into its functioning by using nonlinear activation functions.
- **Convolutional Neural Network**: it is similar to feedforward, but it is better at image recognition since they employ linear algebra and matrices

to identify a picture.

- **Recurrent Neural Network**: it is a sort of artificial neural network in which node connections can create a cycle, enabling one node's output to impact subsequent input to that same node. Using their internal state, RNNs, which are built from feedforward neural networks, can manage variable length sequences of inputs. As a result, they are highly suited to tasks such as speech recognition.

After we train the neural network it is necessary to check how correct it is. To assess the method's quality, they are taken in consideration metrics that depicts the solution errors or accuracy. In classification problems, the following metrics are often utilized: Accuracy, Precision, Recall, and F1. Precision is calculated as the ratio between the number of Positive samples correctly classified to the total number of samples classified as Positive.
The recall measures the model's ability to detect Positive samples. The higher the recall, the more positive samples detected.
The accuracy of a machine learning classification algorithm is one way to measure how often the algorithm classifies a data point correctly. Its formula is the following:

$$Accuracy = \frac{TrueNegative + TruePositive}{TN + TP + FN + FP}$$

Following the selection of the metric, the validation technique must be established. One traditional method is to divide the entire data set into training and test sets. It is crucial to note that picking the model with the best accuracy on the training set does not ensure that it will perform similarly with fresh data in the future. Thus, the goal of validation is to offer at least an approximation of the model's performance for future data. Furthermore, it is critical to strike a balance between underfitting and overfitting. Underfitting indicates that the model performs poorly on both the training and test sets. The most likely cause of underfitting is because the model was not well-tuned on the training set or was not sufficiently trained. As a result, there is a high bias and a low variance. Overfitting indicates that the model is overly tailored to the training data. As a result, the model excels on the training set but fails on the test set. As a result, there is low bias and high variance.

The most major drawback of separating the data into one training and one test set is that the test set may not have the same distribution of classes as the data in general. Furthermore, some numerical characteristics in the training and test sets may not have the same distribution. This is solved using k-fold cross validation. Essentially, it generates a procedure in which every sample in the data is included in the test set at some point. The k denotes a positive integer used to divide in equal part the data, composing equal folds. Using this method, the model is trained separately k-1 times having k-1 scores measured by some of the specified metrics.

3 Image Recognition

Image recognition is the challenge of feeding an image into a neural network and having in return a label. The label that the network produces will correlate to a previously established class that was introduced prior to training. The components of data that you care about that will be supplied over the network are referred to as features.

An essential part is the feature recognition which is the technique of extracting relevant characteristics from an input image so that these features may be examined.

The first layer of a neural network incorporates all of the pixels in a picture. After all of the data has been put into the network, various filters are applied to the picture, resulting in representations of various regions of the image. This is called feature extraction, and it produces feature maps. A convolutional layer is used to extract features from an image, and convolution is simply producing a representation of a portion of an image. A filter is what the network uses to create an image representation. While the filter size specifies the filter's height and width, the filter's depth must also be supplied. Digital pictures are presented as height, width, and some RGB value that identifies the color of the pixel, hence the "depth" being monitored is the number of color channels in the image. Color pictures contain three depth channels, but grayscale images have only one. All of this computation results in a feature map. This technique is often carried out using many filters, which helps to maintain the image's intricacy. Following the creation of the image's feature map, the image's values are sent via an activation function or activation layer. The activation function takes picture values that are in a linear form (i.e. merely a list of numbers) owing to the convolutional layer and enhances their non-linearity since images are non-linear.

The most common activation function used to do this is a Rectified Linear Unit (ReLU), while other activation functions are utilized on occasion. After activation, the data is sent to a pooling layer which compresses the information that describes the image, making it smaller. The act of pooling makes the network more adaptable and competent at detecting images based on relevant attributes.

When looking at an image, the concern is on the things that object of the research, in this example, dogs and cats.

Similarly, in a Neural Network, a pooling layer will abstract away the unnecessary portions of the image, leaving just the parts of the image that it deems are relevant, as decided by the pooling layer's size.

Because the network must make decisions about the most essential elements of the image, it is anticipated to learn just those parts of the image that properly portray the item in issue, helping avoiding overfitting. The maximum values of the pixels are utilized to account for any picture distortions, and the image parameters are lowered to compensate for overfitting.

Other pooling types, such as average pooling and sum pooling, are used less commonly because max pooling produces superior accuracy. The tightly linked layers of our CNN demand that the data be in the form of a vector in order to be processed. As a result, the data must be "flattened." The values are compressed into a lengthy vector or a column of integers in sequential sequence.

The primary purpose of CNN is to evaluate the incoming data and incorporate it into several features that will assist in classification. These layers essentially generate clusters of neurons that reflect distinct aspects of the item in question. An image is classified as an object when a sufficient number of these neurons activate in response to an input picture. The error, or the difference between the calculated and predicted values in the training set, is computed by the Network Network.

The network is then exposed to backpropagation, which calculates and changes the influence of a certain neuron on a neuron in the next layer. This is done to increase the performance of the model. This procedure is then repeated multiple times. By training on data, the network learns correlations between input characteristics and output classifications.

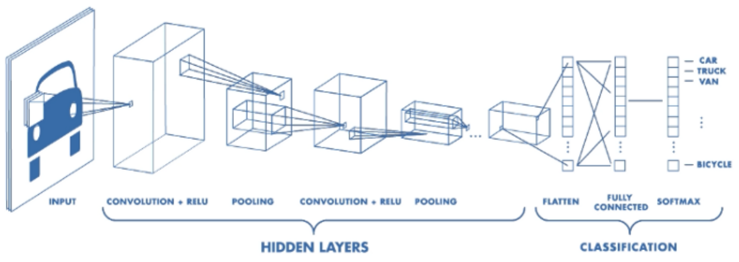
The neurons in the center's completely connected layers will generate binary values according to the available classes. The neuron will assign a 1 to the class it believes the image represents and a 0 to the others. The final fully connected layer will receive the preceding layer's output and offer a probability for each class that adds up to one. The image classifier has now been trained, and the photos may be used to estimate which label to attach.

4 Theoretical Background

After the previous overview, it is possible to describe the different layers:

- Convolution and ReLu
- Pooling
- Flatten
- Fully Connected
- SoftMax

Figure 1: Architecture of a CNN (Image Source)



Convolution

Convolutional layers has as input an image with m-1 channels and compute an output image with m channels. The layer is made up of a rectangular grid of neurons, with each neuron receiving input from a rectangular portion of the previous layer, and the weights for this rectangle area are the same for each neuron in the convolutional layer.

ReLU

ReLu is a non-linear activation function that can be represented as:
Relu(x) = max(0,x)

The largest value between zero and the input value is the output of ReLu. When the input value is negative, the output is equal to zero, and when the input value is positive, the output is equal to the input value.

The major purpose for utilizing ReLU instead of sigmoid and hyperbolic functions is to improve computing performance. Because the derivative of ReLu is 1 for positive input, it can enhance deep neural network training speed when compared to typical activation functions. Deep neural networks do not need to spend extra time during the training phase because of a constant.

The ReLU deep learning function is simple and does not require any intensive processing because there is no complicated mathematics. As a result, the model can train or run faster.
Sparsity is another important feature that we view to be a benefit of using the ReLU activation function. The majority of the entries in a sparse matrix are zero, and we desire this attribute in our ReLU neural networks where part of the weights are zero. Sparsity results in compact models that are more predictive and have less overfitting and noise. Neurons in a sparse network are more likely to be processing critical components of the problem.

Pooling
The pooling layer subsamples tiny rectangular blocks from the convolutional layer to generate a single output from that block. There are numerous methods for pooling neurons in a block, such as taking the average or maximum, or a learnt linear combination of the neurons in the block. They just take a kk area and produce a single number that is the largest value in that region.For instance, if their input layer is a N×N layer, they will then output a $N_k \times N_k$ layer, as each k×k block is reduced to just a single value via the max function.

Flatten
Flattening is the process of turning data into a one-dimensional array for input into the following layer. We flatten the convolutional layer output to form a single long feature vector. It is also linked to the final categorization model.

Fully connected
Every neuron in one layer is connected to every neuron in the next layer via fully connected layers. To identify the photos, the flattened matrix is passed through a fully linked layer. The neural network's high-level reasoning is done via fully connected layers after numerous convolutional and max pooling layers. A completely linked layer connects all neurons in the preceding layer to every neuron it possesses. Because fully linked layers are no longer spatially localized (they may be seen as one-dimensional), there can be no convolutional layers following them.

Sofmax
The softmax function converts a vector of K real values to a vector of K real values that sum to one. The softmax translates input values that are positive, negative, zero, or higher than one into values between 0 and 1, allowing them to be understood as probabilities. If one of the inputs is tiny or negative, the softmax converts it to a small probability; if an input is high, it converts it to a large probability, but it always remains between 0 and 1.

$$\sigma(\vec{z})_i = \frac{e_i^z}{\sum_{j=1}^{K} e_j^z}$$

There are different CNN architecture that are using these layers in different ways and with different parameters. Some of them are:

LeNet-5

LeNet-5 t was developed in the 1990s. It consists of 2 convolutional layers, 3 fully-connected layers, and an average-pooling layer. It was the first successful application of convolutional neural networks to the task of recognizing handwritten digits. It has around 60,000 parameters.

AlexNet

AlexNet was developed in 2012. It consists of 8 layers, including 5 convolutional layers and 3 fully-connected layers. It was the first architecture to win the ImageNet Large Scale Visual Recognition Challenge, and it has been used in many applications since then. It has around 60 million parameters.

ResNet-50

ResNet-50 is a convolutional neural network architecture developed in 2015. It consists of 50 layers, including convolutional layers, fully-connected layers, and residual blocks. It was the first architecture to win the ImageNet Large Scale Visual Recognition Challenge, and it has been used in many applications since then. It has around 25 million parameters.

5 Implementation

Explain the packages

```
import numpy as np
import pandas as pd
import os
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
import tensorflow.keras as keras
import cv2
from tqdm import tqdm
from sklearn.metrics import confusion_matrix
import seaborn as sn
from sklearn.utils import shuffle
from sklearn.model_selection import KFold
import pandas as pd
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.layers import Dense, Flatten, GlobalAveragePooling2D, BatchNormalizati
from tensorflow.keras.applications.resnet50 import preprocess_input
```

Create classes and set the image size

```
class_names = ['Dogs', 'Cats']
class_names_label = {class_name:i for i, class_name in enumerate(class_names)}

nb_classes = len(class_names)

IMAGE_SIZE = (64, 64)
```

Create the function to load the images, setting the color format and the size. The image size is set as 64x64 (to help the machine handling the dataset) while the color is set as RBG. This means images have 3 channels. If they where greyscale, the channel would be 1.

```
def load_data():

    datasets = ['C:/Users/simon/Desktop/CatsDogsSeparate/train',
    'C:/Users/simon/Desktop/CatsDogsSeparate/test']
    output = []
    for dataset in datasets:

        images = []
        labels = []

        print("Loading {}".format(dataset))

        for folder in os.listdir(dataset):
            label = class_names_label[folder]

            for file in tqdm(os.listdir(os.path.join(dataset, folder))):

                img_path = os.path.join(os.path.join(dataset, folder), file)

                image = cv2.imread(img_path)
                image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
                image = cv2.resize(image, IMAGE_SIZE)

                images.append(image)
                labels.append(label)

        images = np.array(images, dtype = 'float32')
        labels = np.array(labels, dtype = 'int32')

        output.append((images, labels))

    return output
```

Load and split the images into training (90%) and test (10%) sets

```
(train_images, train_labels), (test_images, test_labels) = load_data()
```

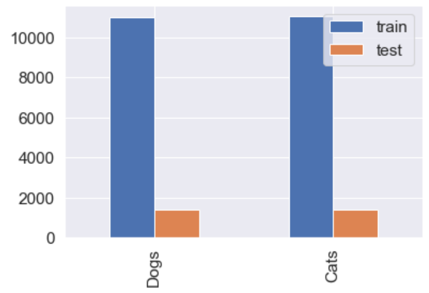
The following function will allow to shuffle arrays in a consistent way. It required as input the arrays (the training images and labels in this case), the random state (which determine random number generation for shuffling the data) and the number of sample to generate.

```
train_images, train_labels = shuffle(train_images, train_labels, random_state=25)
n_train = train_labels.shape[0]
n_test = test_labels.shape[0]

print ("Number of training examples: {}".format(n_train))
print ("Number of testing examples: {}".format(n_test))
print ("Each image is of size: {}".format(IMAGE_SIZE))
```

Number of training examples: 22075 Number of testing examples: 2795 Each image is of size: (64, 64)

It is useful to Visualise the compositions of our sets, to have a visual representation of the proportions.

```
_, train_counts = np.unique(train_labels, return_counts=True)
_, test_counts = np.unique(test_labels, return_counts=True)
pd.DataFrame({'train': train_counts,
                    'test': test_counts},
             index=class_names
            ).plot.bar()
plt.show()
```

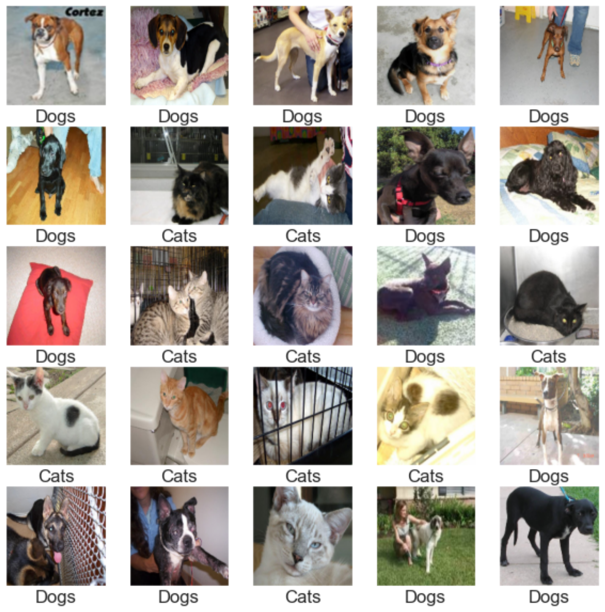


```
train_images = train_images / 255.0
test_images = test_images / 255.0
```

At this point, it is possible to display the image part of the data.

```
def display_examples(class_names, images, labels):

    fig = plt.figure(figsize=(10,10))
    fig.subtitle("Examples", fontsize=16)
    for i in range(25):
        plt.subplot(5,5,i+1)
        plt.xticks([])
        plt.yticks([])
        plt.grid(False)
        plt.imshow(images[i], cmap=plt.cm.binary)
        plt.xlabel(class_names[labels[i]])
    plt.show()

    display_examples(class_names, train_images, train_labels)
```

Some examples of images of the dataset



Create a model

- Sequential: it is one of the model that is used to investigate varied types of neural networks where the model gets in one input as feedback and expects an output as desired. Keras class is one of the important class as part of the entire Keras sequential model. This class helps in creating a cluster where a cluster is formed with layers of information or data that flows with top to bottom approach having a lot of layers incorporated with tf.Keras. a model where most of its features are trained with algorithms that provide a lot of sequence to the model.

- COnv2D:This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. If use bias is True, a bias vector is created and added to the outputs. Finally, if activation is not None, it is applied to the outputs as well.

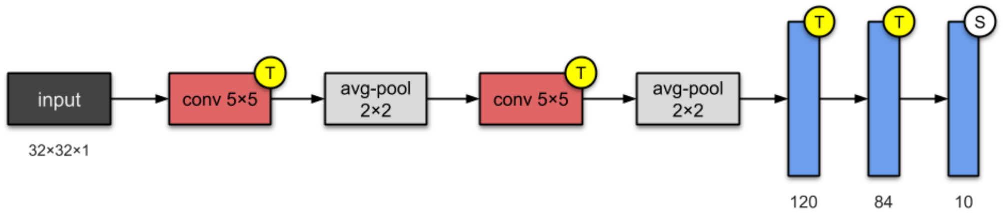
- activation: specify the activation function for the layer. Relu Applies the rectified linear unit activation function. The tanh activation function is Zero centered; so the result ca be interpreted as strongly negative, neutral, or strongly positive. Other functions are sigmoid, softmax, softplus, softsign and more.

- AveragePooling2D: Downsamples the input along its spatial dimensions (height and width) by taking the average value over an input window (of size defined by pool size) for each channel of the input. The window is shifted by strides along each dimension.

- Flatten: Return a copy of the array collapsed into one dimension.
- Dense: Dense implements the operation: output = activation(dot(input, kernel) + bias) where activation is the element-wise activation function passed as the activation argument, kernel is a weights matrix created by the layer, and bias is a bias vector created by the layer (only applicable if use bias is True). These are all attributes of Dense.

LeNet-5

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (5, 5), activation = 'tanh', input_shape = (64, 64, 3)),
    tf.keras.layers.AveragePooling2D(2,2),
    tf.keras.layers.Conv2D(32, (5, 5), activation = 'tanh'),
    tf.keras.layers.AveragePooling2D(2,2),
    keras.layers.Flatten(),
    keras.layers.Dense(1024,activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(1024,activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10,activation='softmax')
])
```



It is now the time to configure the model with the losses and metrics

- optimizer: Name of optimizer or optimizer instance. E.g.: Adam, Adamax, Adagrad, FTRL, RMSprop. SGD. Adam is suggested as computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters
- loss: Name of loss function). The sparse_categorical_crossentropy computes the cross-entropy loss between true labels and predicted labels.

- metrics: List of the metrics to be evaluated by the model during training and testing.

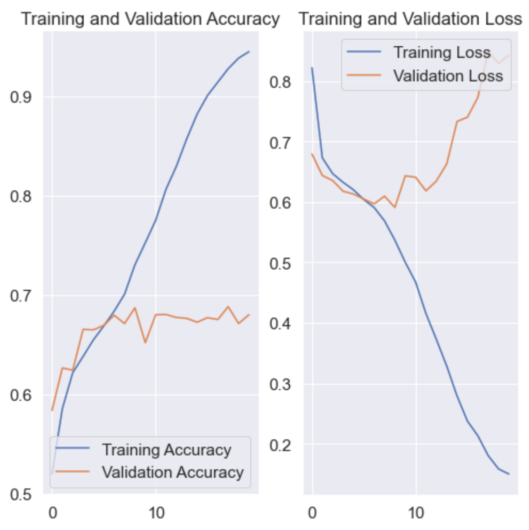
```
model.compile(optimizer = 'adam',
loss = 'sparse_categorical_crossentropy', metrics=['accuracy'])
```

Fit train the model for a fixed number of epochs (which are the iteration in the dataset). The batch are the number of samples per gradient update. On validation data, neurons using drop out do not drop random neurons. The reason is that during training we use drop out in order to add some noise for avoiding over-fitting ehile during calculating cross-validation, we are in the recall phase and not in the training phase (using all the capabilities of the network).

```
history = model.fit(train_images, train_labels, batch_size=128,
epochs=20, validation_split = 0.2)

Receiving for each epoch the following results:
Epoch 20/20
138/138 [==============================] - 264s 2s/step -
loss: 0.1499 - accuracy: 0.9447 - val_loss: 0.8428 - val_accuracy: 0.6802
```

At this point, plotting the results is the faster way to have a good overview of the results

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(20)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```

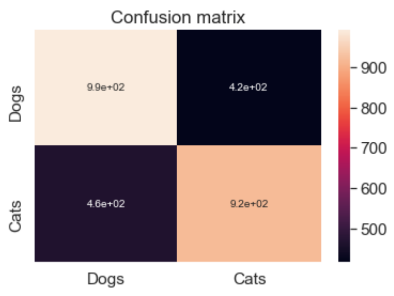


Another helpful step is to check the loss value metrics values for the model in test mode.

```
test_loss = model.evaluate(test_images, test_labels)
print(f'Test loss: {test_loss[0]} / Test accuracy: {test_loss[1]}')

88/88 [==============================] - 14s 123ms/step -
loss: 0.7813 - accuracy: 0.6844
Test loss: 0.781338632106781 / Test accuracy: 0.6844365000724792
```

As the last step to complete the analysis, it is good norm to check the misclassificated images against the correct ones in a graphic way. This is particularly useful when there are more than two labels.

```
CM = confusion_matrix(test_labels, pred_labels)
ax = plt.axes()
sn.heatmap(CM, annot=True,
           annot_kws={"size": 10},
           xticklabels=class_names,
           yticklabels=class_names, ax = ax)
ax.set_title('Confusion matrix')
plt.show()
```



As it is highlighted by the map, there is no large difference between the two misclassification (cats for dogs or dogs for cats).

K-Fold Cross-Validation

Another aspect to evaluate is the validation method. The K-Fold Cross-Validation is a method used to compare and select a model and detect overfitting. This is done by leveraging the subsetting of the same dataset and running the model in different conditions but using the same data.

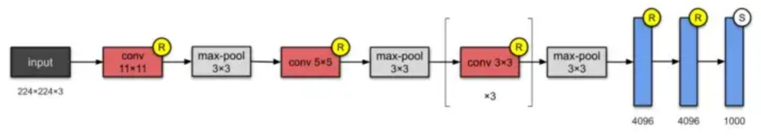
```
folds = 5

accuracy_per_fold = []
loss_per_fold = []

inputs = np.concatenate((train_imgs, test_imgs), axis=0)
targets = np.concatenate((train_labels, test_labels), axis=0)

kfold = KFold(n_splits=folds, shuffle=True)

fold_no = 1
for train, test in kfold.split(train_images, train_labels):
  model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (5, 5), activation = 'tanh', input_shape = (64, 64, 3)),
    tf.keras.layers.AveragePooling2D(2,2),
    tf.keras.layers.Conv2D(32, (5, 5), activation = 'tanh'),
    tf.keras.layers.AveragePooling2D(2,2),
    keras.layers.Flatten(),
    keras.layers.Dense(1024,activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(1024,activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10,activation='softmax')
  ])
  model.compile(loss='sparse_categorical_crossentropy',
                optimizer='Adam',
                metrics=['accuracy'])


  print('------------------------------------------------------------------------')
  print(f'Training for fold {fold_no} ...')

  # Fit data to model
  history = model.fit(inputs[train], targets[train],
              batch_size=50,
              epochs=25,
              verbose=1)

  scores = model.evaluate(inputs[test], targets[test], verbose=0)
  print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]}; {model.metrics_
  acc_per_fold.append(scores[1] * 100)
  loss_per_fold.append(scores[0])

  fold_no = fold_no + 1


# == Provide average scores ==
print('------------------------------------------------------------------------')
print('Score per fold')
for i in range(0, len(acc_per_fold)):
  print('------------------------------------------------------------------------')
  print(f'> Fold {i+1} - Loss: {loss_per_fold[i]} - Accuracy: {acc_per_fold[i]}%')
print('------------------------------------------------------------------------')
print('Average scores for all folds:')
print(f'> Accuracy: {np.mean(acc_per_fold)} (+- {np.std(acc_per_fold)})')
print(f'> Loss: {np.mean(loss_per_fold)}')
print('------------------------------------------------------------------------')

Score per fold
--------------------------------------------------------------------
> Fold 1: Loss of 0.6939; accuracy of 50.06%
--------------------------------------------------------------------
> Fold 2: Loss of 0.6932; accuracy of 49.885%
--------------------------------------------------------------------
> Fold 3 - Loss of 0.6935; accuracy of 49.49%
--------------------------------------------------------------------
> Fold 4 - Loss of 0.6937; accuracy of 49.17%
--------------------------------------------------------------------
> Fold 5 - Loss of 0.6932; accuracy of 48.954%
--------------------------------------------------------------------
```

```
The average accuracy is 49.512
```

AlexNet

One difference is the use of Max Pooling instead of the Average one. It helps to reduce the number of parameters and computation in the network. It is a form of non-linear down-sampling. Max pooling works by sliding a window over the input and taking the maximum value in each window. This reduces the size of the input and helps to make the network more robust to small changes in the input. The main difference between max pooling and average pooling is that max pooling takes the maximum value from each window, while average pooling takes the average value from each window. Max pooling is more effective at preserving the most important features of the input, while average pooling is more effective at smoothing out noise.

```
model=keras.models.Sequential([
    keras.layers.Conv2D(filters=128, kernel_size=(11,11), strides=(4,4), activation='relu',
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(2,2)),
    keras.layers.Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), activation='relu', pa
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(2,2)),
    keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu', pa
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu', pa
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu', pa
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(2,2)),
    keras.layers.Flatten(),
    keras.layers.Dense(1024,activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(1024,activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10,activation='softmax')
])
```

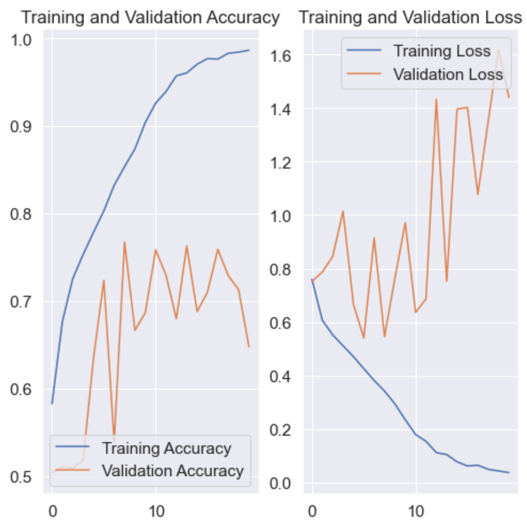


```
model.compile(optimizer = 'adam',
loss = 'sparse_categorical_crossentropy', metrics=['accuracy'])
```

Fit train the model for a fixed number of epochs (which are the iteration in the dataset). The batch are the number of samples per gradient update. On validation data, neurons using drop out do not drop random neurons. The reason is that during training we use drop out in order to add some noise for avoiding over-fitting ehile during calculating cross-validation, we are in the recall phase and not in the training phase (using all the capabilities of the network).

```
history = model.fit(train_images, train_labels, batch_size=128,
epochs=20, validation_split = 0.2)

Receiving for each epoch the following results:
Epoch 20/20
138/138 [==============================] - 264s 2s/step -
loss: 0.0372 - accuracy: 0.9866 - val_loss: 1.4399 - val_accuracy: 0.6476
```

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(20)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```



```
test_loss = model.evaluate(test_images, test_labels)
print(f'Test loss: {test_loss[0]} / Test accuracy: {test_loss[1]}')

88/88 [==============================] - 14s 123ms/step -
loss: 1.4749 - accuracy: 0.6490
Test loss: 1.474876880645752 / Test accuracy: 0.6490160822868347
```

```
CM = confusion_matrix(test_labels, pred_labels)
ax = plt.axes()
sn.heatmap(CM, annot=True,
           annot_kws={"size": 10},
           xticklabels=class_names,
           yticklabels=class_names, ax = ax)
ax.set_title('Confusion matrix')
plt.show()
```

K-Fold Cross-Validation

```
folds = 5

accuracy_per_fold = []
loss_per_fold = []
```

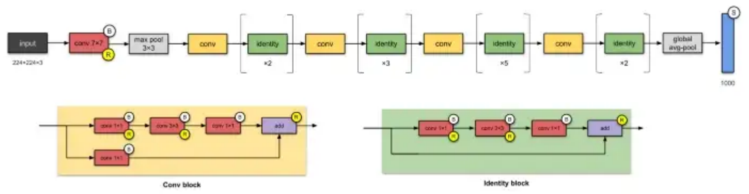
```
inputs = np.concatenate((train_imgs, test_imgs), axis=0)
targets = np.concatenate((train_labels, test_labels), axis=0)

kfold = KFold(n_splits=folds, shuffle=True)

fold_no = 1
for train, test in kfold.split(train_images, train_labels):
  model=keras.models.Sequential([
    keras.layers.Conv2D(filters=128, kernel_size=(11,11), strides=(4,4), activation='relu',
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(2,2)),
    keras.layers.Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), activation='relu', pa
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(2,2)),
    keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu', pa
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu', pa
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu', pa
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(2,2)),
    keras.layers.Flatten(),
    keras.layers.Dense(1024,activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(1024,activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10,activation='softmax')
  ])

  model.compile(loss='sparse_categorical_crossentropy',
                optimizer='Adam',
                metrics=['accuracy'])


  print('------------------------------------------------------------------------')
  print(f'Training for fold {fold_no} ...')

  # Fit data to model
  history = model.fit(inputs[train], targets[train],
              batch_size=50,
              epochs=25,
              verbose=1)

  scores = model.evaluate(inputs[test], targets[test], verbose=0)
  print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]}; {model.metrics_
  acc_per_fold.append(scores[1] * 100)
  loss_per_fold.append(scores[0])

  fold_no = fold_no + 1


# == Provide average scores ==
print('------------------------------------------------------------------------')
print('Score per fold')
for i in range(0, len(acc_per_fold)):
  print('------------------------------------------------------------------------')
  print(f'> Fold {i+1} - Loss: {loss_per_fold[i]} - Accuracy: {acc_per_fold[i]}%')
print('------------------------------------------------------------------------')
print('Average scores for all folds:')
print(f'> Accuracy: {np.mean(acc_per_fold)} (+- {np.std(acc_per_fold)})')
print(f'> Loss: {np.mean(loss_per_fold)}')
print('------------------------------------------------------------------------')
```

```
Score per fold
--------------------------------------------------------------------
> Fold 1: Loss of 1.425; accuracy of 76.467%
--------------------------------------------------------------------
> Fold 2: Loss of 1.6133; accuracy of 78.12%
--------------------------------------------------------------------
> Fold 3 - Loss of 1.284; accuracy of 76.7388%
--------------------------------------------------------------------
> Fold 4 - Loss of 1.165; accuracy of 76.8063%
--------------------------------------------------------------------
> Fold 5 - Loss of 1.78965; accuracy of 77.6217%
--------------------------------------------------------------------

The average accuracy is 77.15
```

It is interesting to note that the accuracy of a model can increase with the loss because the model is learning from the mistakes it makes. As the model learns from its mistakes, it is able to better identify patterns in the data and make more accurate predictions. This leads to an increase in accuracy. Saying that, it is important to note that a high loss and high accuracy can indicate that the model is overfitting the data. This means that the model is memorizing the data instead of learning from it. This can lead to poor generalization performance on unseen data. To understand if a model is overfitting, it is possible to compare the performance of the model on the training data and the test data. If the model performs significantly better on the training data than the test data, then it is likely overfitting. In this case, it is also possible to look at the loss and accuracy of the model to see if there is a large discrepancy between the two.

ResNet-50

```
model = Sequential()
model.add(ResNet50(include_top=False, pooling='avg', weights=None))
model.add(Flatten())
model.add(BatchNormalization())
model.add(Dense(2048, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(1024, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(2, activation='softmax'))
model.layers[0].trainable = False
```



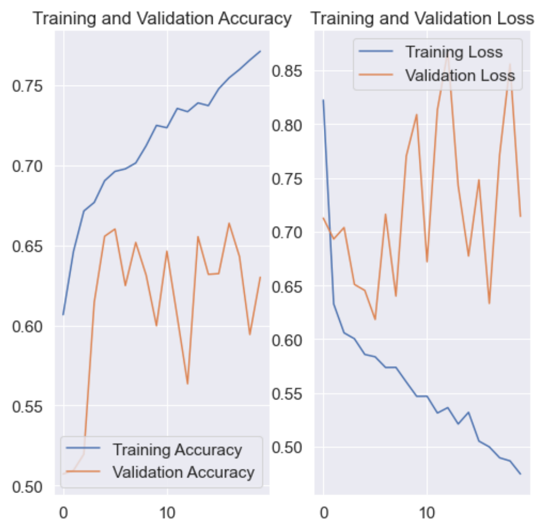
```
model.compile(optimizer = 'adam',
loss = 'sparse_categorical_crossentropy', metrics=['accuracy'])
```

Fit train the model for a fixed number of epochs (which are the iteration in the dataset). The batch are the number of samples per gradient update. On validation data, neurons using drop out do not drop random neurons. The reason is that during training we use drop out in order to add some noise for avoiding over-fitting ehile during calculating cross-validation, we are in the recall phase and not in the training phase (using all the capabilities of the network).

```
history = model.fit(train_images, train_labels, batch_size=128,
epochs=20, validation_split = 0.2)

Receiving for each epoch the following results:
Epoch 20/20
138/138 [==============================] - 264s 2s/step -
loss: 0.4746 - accuracy: 0.7713 - val_loss: 0.7142 - val_accuracy: 0.6301
```

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
```

```
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(20)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```



```
test_loss = model.evaluate(test_images, test_labels)
print(f'Test loss: {test_loss[0]} / Test accuracy: {test_loss[1]}')

88/88 [==============================] - 14s 123ms/step -
loss: 0.7006 - accuracy: 0.6501
Test loss: 0.7006387710571289 / Test accuracy: 0.65008944272995
```

```
CM = confusion_matrix(test_labels, pred_labels)
ax = plt.axes()
sn.heatmap(CM, annot=True,
           annot_kws={"size": 10},
           xticklabels=class_names,
           yticklabels=class_names, ax = ax)
ax.set_title('Confusion matrix')
plt.show()
```

K-Fold Cross-Validation

```
folds = 5

accuracy_per_fold = []
loss_per_fold = []

inputs = np.concatenate((train_imgs, test_imgs), axis=0)
targets = np.concatenate((train_labels, test_labels), axis=0)

kfold = KFold(n_splits=folds, shuffle=True)

fold_no = 1
for train, test in kfold.split(train_images, train_labels):
  model = Sequential()
  model.add(ResNet50(include_top=False, pooling='avg', weights=None))
  model.add(Flatten())
  model.add(BatchNormalization())
  model.add(Dense(2048, activation='relu'))
  model.add(BatchNormalization())
  model.add(Dense(1024, activation='relu'))
  model.add(BatchNormalization())
  model.add(Dense(2, activation='softmax'))
  model.layers[0].trainable = False

  model.compile(loss='sparse_categorical_crossentropy',
                optimizer='Adam',
                metrics=['accuracy'])

  model.compile(loss='sparse_categorical_crossentropy',
                optimizer='Adam',
                metrics=['accuracy'])
  print('------------------------------------------------------------------------')
  print(f'Training for fold {fold_no} ...')

  # Fit data to model
  history = model.fit(inputs[train], targets[train],
              batch_size=50,
              epochs=25,
              verbose=1)

  scores = model.evaluate(inputs[test], targets[test], verbose=0)
  print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]}; {model.metrics_
  acc_per_fold.append(scores[1] * 100)
  loss_per_fold.append(scores[0])

  fold_no = fold_no + 1


# == Provide average scores ==
print('------------------------------------------------------------------------')
print('Score per fold')
for i in range(0, len(acc_per_fold)):
  print('------------------------------------------------------------------------')
  print(f'> Fold {i+1} - Loss: {loss_per_fold[i]} - Accuracy: {acc_per_fold[i]}%')
print('------------------------------------------------------------------------')
print('Average scores for all folds:')
print(f'> Accuracy: {np.mean(acc_per_fold)} (+- {np.std(acc_per_fold)})')
print(f'> Loss: {np.mean(loss_per_fold)}')
print('------------------------------------------------------------------------')

Score per fold
--------------------------------------------------------------------
> Fold 1: Loss of 0.92598; accuracy of 66.5232%
--------------------------------------------------------------------
> Fold 2: Loss of 1.0522; accuracy of 64.91518%
--------------------------------------------------------------------
> Fold 3 - Loss of 0.9271; accuracy of 66.0929%
--------------------------------------------------------------------
> Fold 4 - Loss of 0.9267; accuracy of 65.3001%
--------------------------------------------------------------------
> Fold 5 - Loss of 0.97832; accuracy of 64.7565%
--------------------------------------------------------------------

The average accuracy is 65.52
```

Here the average results from the different validations:

LeNet-5: Loss: 0.6935 — Accuracy: 49.512
AlexNet: Loss: 1.455 — Accuracy: 77.15
ResNet-50: Loss: 0.9621 — Accuracy: 65.52

Choose the best model based on loss and accuracy depends on:

- Loss: Lower values of loss indicate that the model is making accurate predictions. You should choose the model with the lowest loss, as long as be mindful about over and under fitting
- Accuracy: Accuracy is a measure of how many of the predictions made by the model are correct. A high accuracy value is usually desirable, but you should also be mindful of imbalanced data where accuracy can be misleading.
- Validation Loss and Accuracy: It's important to use validation data (data the model has not seen before) to evaluate the model's performance and avoid over fitting. You should choose the model that has the lowest validation loss and the highest validation accuracy.

A LeNet-5 model with a loss of 0.69 and accuracy of 49% in image recognition might be considered under performing, depending on the specifics of the problem and the data.
Loss: A loss value of 0.69 is relatively high, indicating that the model is making a large number of incorrect predictions.
Accuracy: An accuracy of 49% is relatively low, and might indicate that the model is not accurately recognizing the objects in the images for the majority of the cases.
It's important to keep in mind that these results may be due to overfitting or underfitting, or they may indicate that the architecture of the LeNet-5 model is not well suited for the problem to solve.

For an image classification task, an AlexNet model with a loss of 1.45 and accuracy of 77.15% might be considered to be performing moderately.
Loss: A loss value of 1.45 indicates that the model is still making a significant number of incorrect predictions, so there is room for improvement.
Accuracy: An accuracy of 77.15% is above average, suggesting that the model is accurately recognizing the objects in the images for a relatively high percentage of the cases. However, accuracy is not the only metric to consider, as the specific problem, data distribution, and desired trade-off between false negatives and false positives, will influence the optimal value of accuracy.

Also a ResNet-50 model with a loss of 0.96 and accuracy of 65.52% might be considered as performing moderately.

In conclusion:

- Looking for the best accuracy, AlexNet with a loss of 1.45 and accuracy of 77.15% might be the best choice. However, it's important to keep in mind that a higher accuracy doesn't necessarily mean a better model, as the model might be overfitting or the data might have some issues.

- If you want a balance between accuracy and model complexity, ResNet-50 with a loss of 0.96 and accuracy of 65.52% might be a good choice. ResNet-50 is known for its good performance and relatively low complexity compared to other models, making it a good choice for many image classification tasks.

- LeNet-5 with a loss of 0.69 and accuracy of 49% is the model with the lowest accuracy, so it might not be the best choice for an image classification task. However, in case of working with limited computational resources or needing a faster training time, LeNet-5 might still be a good option.

Images of the different architectures are taken from the following article: Illustrated: 10 CNN Architectures