# Lab2 (5IR - 2024): Fall Detection - Supervised learning

**Group name (2 people !):** 2

**Names:** Jumin, Gauche

**First names:** Noël, Clément

The objectives of this lab are

- Development of a classification model based on artificial neural networks for the fall detection dataset CAUCA,
- Use TensorFlow to determine a performant CNN architecture,
- Study of the architecture of a neural network and the sensitivity of its parameters,
- Use TensorFlow Lite to compress the CNN,
- (Optional) Implementation of Transfer Learning using an efficient architecture (e.g. MobileNetV2).

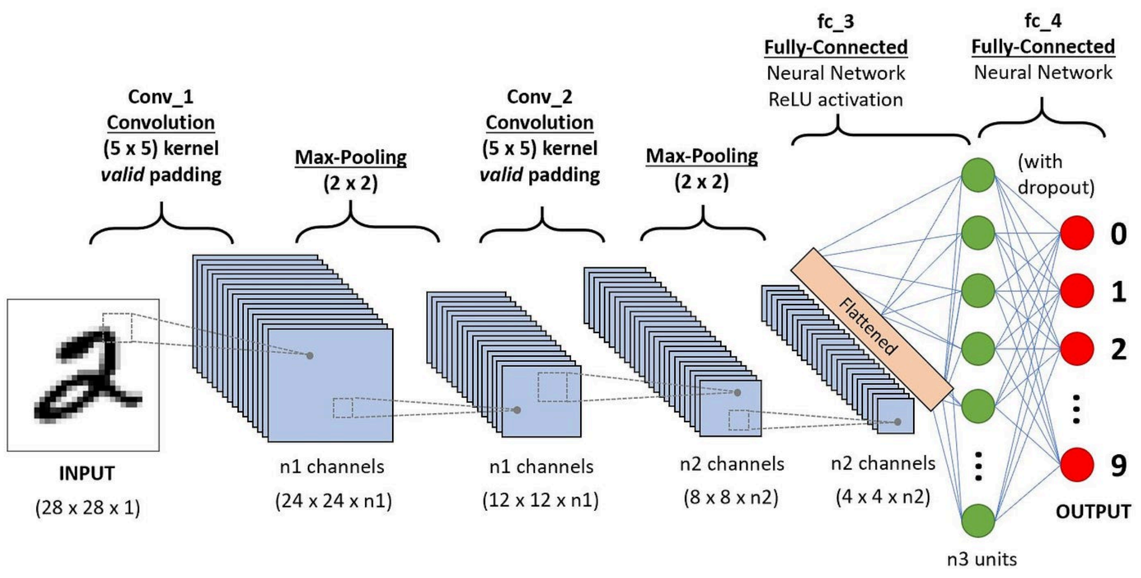
In [28]:
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.datasets import mnist
from keras.models import Sequential, save_model, clone_model, load_model
from keras.layers import Dense, Conv2D, Flatten, MaxPooling2D, BatchNormaliz
from keras.utils import to_categorical
from keras.saving import load_model

# TensorFlow Optimization
import tensorflow_model_optimization as tfmot
from tensorflow_model_optimization.sparsity.keras import prune_low_magnitude

import numpy as np
import matplotlib.pyplot as plt
import random
import seaborn as sns
import numpy as np
import tempfile
import os
import zipfile
import math
import pickle
from sklearn.metrics import confusion_matrix
import cv2
from sklearn.model_selection import train_test_split
```

## 1. Classification of the CAUCA dataset using Convolutional Neural Netowrks (CNN)

TensorFlow is an open-source machine learning library developed by Google, primarily used for deep learning applications. It provides a comprehensive, flexible ecosystem of tools, libraries, and community resources that enables researchers to develop and deploy machine learning powered applications easily. We use the Python version of the library which can be installed in Python with: *pip install tensorflow*

We will use the library **tensorflow.keras** to develop ANN of type Convolutional Neural Netowrks (CNN). We use the generic class **Sequential** combined with layers of types:

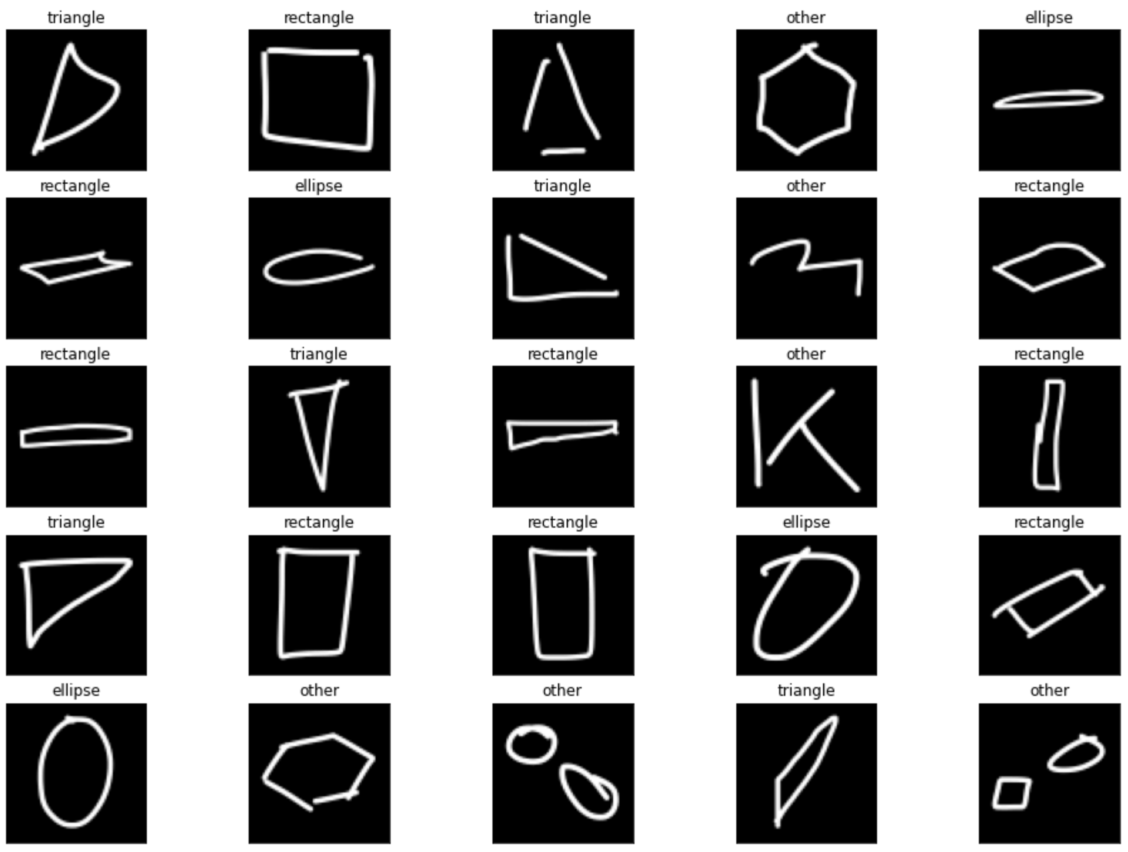
- **Conv2D**,
- **MaxPooling2D**,
- **BatchNormalization**,
- **Dropout**,
- **Flatten**,
- **Dense**.



The documentation for the Tensorflow layers is available at https://www.tensorflow.org/api_docs/python/tf/keras/layers. Since an example is better than 1000 words, below is a fully working example code for a toy classification problem.

## 1.1 Hand-drawn shapes (HDS) dataset

Some datasets are available on the internet (search Google, Kaggle, and scientific articles). Here, we focus on a small example dataset extracted from the repository: https://github.com/frobertpixto/hand-drawn-shapes-dataset/tree/main. This dataset contains 70x70px images (in .png format) in grayscale of hand-drawn shapes. The images can be of 4 classes: ellipse, triangle, rectangle, others. There are around 50 000 images available. Processing codes are available on github to preprocess the pictures, and build a CNN model for classification.



We have modified the processing code from the repository in order to reduce the dataset for our purpose:

- The images are resized to 14x14px,
- We only keep the classes ellipse, triangle, and rectangle,
- We keep 4 500 images.

The dataset is stored in a pickle file which allows to drop Python data structures to a binary file. The structure is a dictionnary with the data X at key "train_data", and the labes y at key "train_labels":

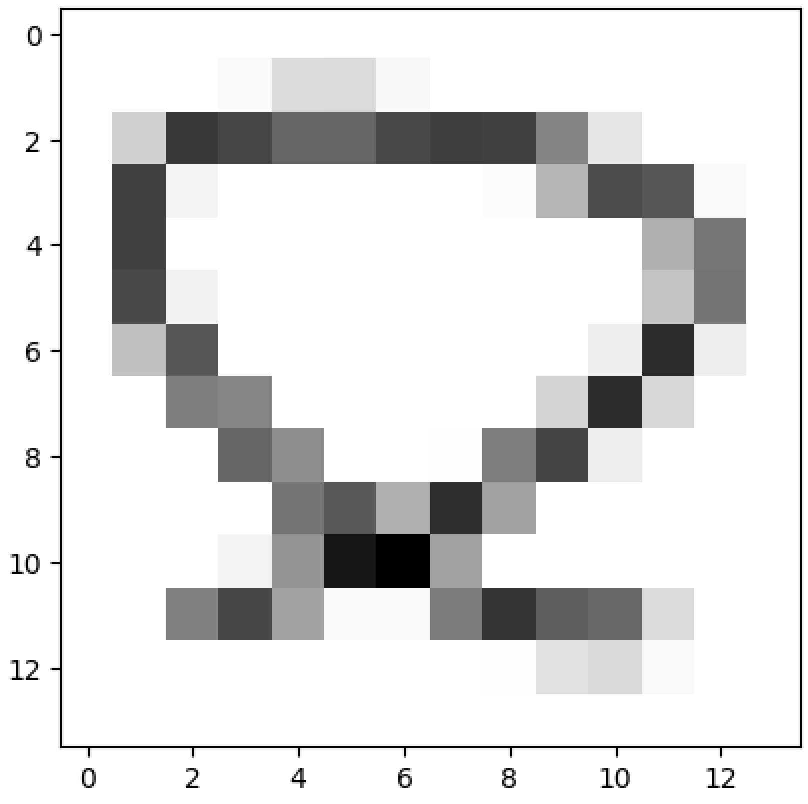
- Storing: *save_to_pickle( "train.pickle", {*
  `*'train_data': train_data,*`
  `*'train_labels': train_labels*`
  *})*
- Loading *with open("train.pickle", 'rb') as file: train_dict = pickle.load(file)*

- Load data
- N.B.: In the obtained labels y, class 1="ellipse", class 2="rectangle", class 3="triangle"

In [29]:
```
with open("hds_data.pickle", 'rb') as file:
    train_dict = pickle.load(file)
X = train_dict['train_data']
y = train_dict['train_labels'] - 1
```

In [30]:
```
ind = random.randint(0, len(X))
print("Image: ",ind)
plt.imshow(X[ind],cmap=plt.cm.gray_r,interpolation="nearest")
plt.show()
```

```
Image:  1118
```



- **Split into train and test datasets**

In [31]:
```
xtrain, xtest, ytrain, ytest = train_test_split(X, y, train_size=0.8)

# Reshape and normalize the images
xtrain = xtrain.reshape((xtrain.shape[0], 14, 14, 1)).astype('float32')
xtest = xtest.reshape((xtest.shape[0], 14, 14, 1)).astype('float32')
# Convert labels to categorical (one-hot encoding)
ytrain_to_print1 = ytrain
ytrain = to_categorical(ytrain)
ytrain_to_print2 = ytrain
ytest = to_categorical(ytest)
```

- **Create the model**

In [32]:
```
# Building the CNN model
model = Sequential([
    Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(14, 14, 1
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(32, activation='relu'),
    Dense(3, activation='softmax')
])

model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d_4 (Conv2D)           (None, 12, 12, 32)        320       
                                                                 
 batch_normalization_4 (Bat  (None, 12, 12, 32)        128       
 chNormalization)                                                
                                                                 
 max_pooling2d_4 (MaxPoolin  (None, 6, 6, 32)          0         
 g2D)                                                            
                                                                 
 flatten_2 (Flatten)         (None, 1152)              0         
                                                                 
 dense_4 (Dense)             (None, 32)                36896     
                                                                 
 dense_5 (Dense)             (None, 3)                 99        
                                                                 
=================================================================
Total params: 37443 (146.26 KB)
Trainable params: 37379 (146.01 KB)
Non-trainable params: 64 (256.00 Byte)
_________________________________________________________________
```

- **Compile and train the model**

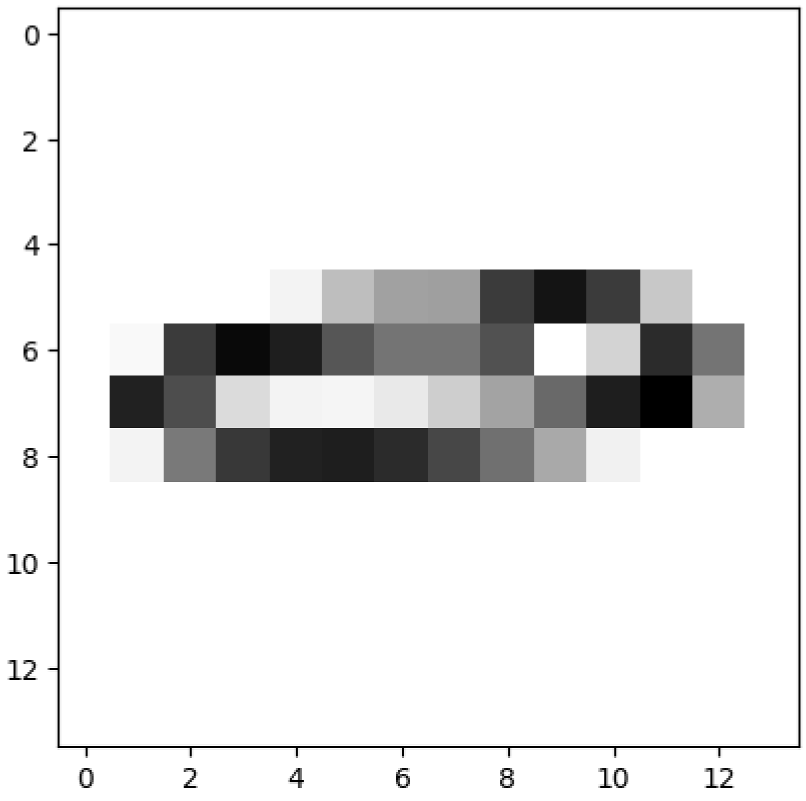
In [33]:
```python
# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['a

# Train the model
history = model.fit(xtrain, ytrain, validation_data=(xtest, ytest), epochs=1
```

```
Epoch 1/10
29/29 [==============================] - 1s 7ms/step - loss: 0.8333 - accur
acy: 0.6314 - val_loss: 1.0438 - val_accuracy: 0.6767
Epoch 2/10
29/29 [==============================] - 0s 4ms/step - loss: 0.4992 - accur
acy: 0.8081 - val_loss: 1.0027 - val_accuracy: 0.6822
Epoch 3/10
29/29 [==============================] - 0s 5ms/step - loss: 0.3990 - accur
acy: 0.8508 - val_loss: 0.9577 - val_accuracy: 0.7389
Epoch 4/10
29/29 [==============================] - 0s 4ms/step - loss: 0.3351 - accur
acy: 0.8775 - val_loss: 0.9133 - val_accuracy: 0.6944
Epoch 5/10
29/29 [==============================] - 0s 4ms/step - loss: 0.2881 - accur
acy: 0.8944 - val_loss: 0.8739 - val_accuracy: 0.6567
Epoch 6/10
29/29 [==============================] - 0s 5ms/step - loss: 0.2640 - accur
acy: 0.9050 - val_loss: 0.8226 - val_accuracy: 0.7600
Epoch 7/10
29/29 [==============================] - 0s 5ms/step - loss: 0.2156 - accur
acy: 0.9275 - val_loss: 0.7631 - val_accuracy: 0.7744
Epoch 8/10
29/29 [==============================] - 0s 5ms/step - loss: 0.1976 - accur
acy: 0.9356 - val_loss: 0.7068 - val_accuracy: 0.7978
Epoch 9/10
29/29 [==============================] - 0s 5ms/step - loss: 0.1705 - accur
acy: 0.9436 - val_loss: 0.6416 - val_accuracy: 0.8689
Epoch 10/10
29/29 [==============================] - 0s 5ms/step - loss: 0.1533 - accur
acy: 0.9506 - val_loss: 0.5813 - val_accuracy: 0.8844
```

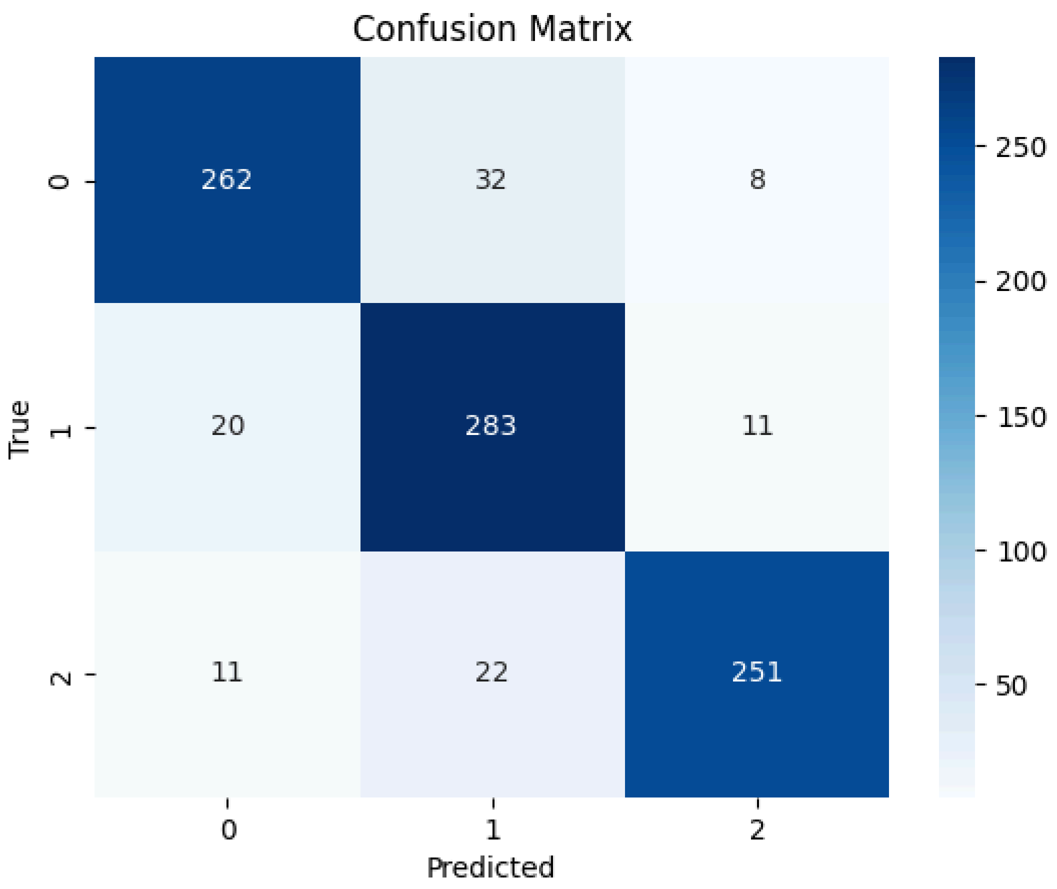
- **TODO: Evaluate the model** (this will help you later):
  - display a random image in the test set with its true and predicted labels,
  - display the accuracy of the CNN on the train and test sets,
  - compute and display the confusion matrix of the model,
  - display the evolution of the accuracy over the training iterations of the CNN.

In [34]:
```python
# Display a random image in the test set with its true and predicted labels
ind = random.randint(0, len(xtest) - 1)
print("Image: ",ind)
plt.imshow(xtest[ind],cmap=plt.cm.gray_r,interpolation="nearest")
plt.show()

# Display the accuracy of the CNN on the train and test sets
loss, accuracy = model.evaluate(xtest, ytest)
print(f"Test Accuracy: {accuracy * 100}%")

# Compute and display the confusion matrix of the model
y_pred = model.predict(xtest)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = np.argmax(ytest, axis=1)
conf_matrix = confusion_matrix(y_true, y_pred_classes)
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()

# Display the evolution of the accuracy over the training iterations of the
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Accuracy evolution')
plt.show()
```

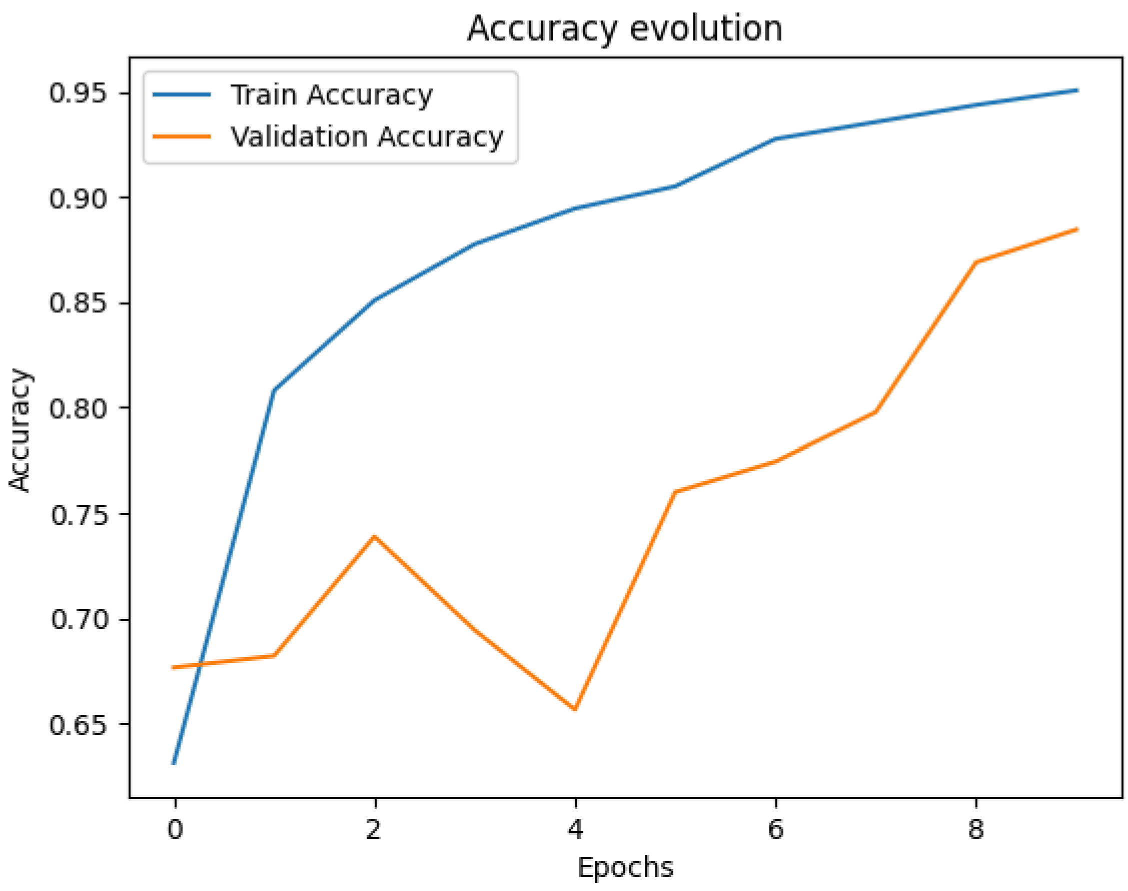
```
Image:  481
```



```
29/29 [==============================] - 0s 1ms/step - loss: 0.5813 - acc
uracy: 0.8844
Test Accuracy: 88.44444155693054%
29/29 [==============================] - 0s 1ms/step
```



|  | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 262 | 32 | 8 |
| 1 | 20 | 283 | 11 |
| 2 | 11 | 22 | 251 |



- **TODO: Questions:** (this will help you later)
  - look at the documentation of Tensorflow (or other) and explain briefly what is the a *Dropout* layer.

    As the TensorFlow documentation says, the dropout layer randomly sets imput units to 0 with a certain frequency at each step during training time to help prevent overfitting. This is automaticaly set when using model.fit without training=false.

  - what does the *to_categorical* function do and why do we modify the labels (ground truth) with it ?

    This function transform each data in an array of categories where each categorie is an index of the array and if its value is one, the data is in this categorie. For example, each data can be an ellipse, a rectangle or a triangle so: 0, 1 or 2 will become [1,0,0], [0,1,0] and [0,0,1], going from integers to binary.

In [35]:
```
ellipse_ind = np.where(ytrain_to_print1 == 0)[0][0]
rectangle_ind = np.where(ytrain_to_print1 == 1)[0][0]
triangle_ind = np.where(ytrain_to_print1 == 2)[0][0]
print(f'Value before categorization: (ellipse: {ytrain_to_print1[ellipse_inc
print(f'Value after categorization: (ellipse: {ytrain_to_print2[ellipse_ind]
```

```
Value before categorization: (ellipse: 0, rectangle: 1, triangle: 2)
Value after categorization: (ellipse: [1. 0. 0.], rectangle: [0. 1. 0.],
triangle: [0. 0. 1.])
```

- what is contained in the prediction for any example ?

There is the same array as after the categorization with each index representing a categorie. After the prediction we got the probability for the data to be in each classes.

In [36]:
```
model.save("best_CNN.keras")
model2 = load_model("best_CNN.keras")
```

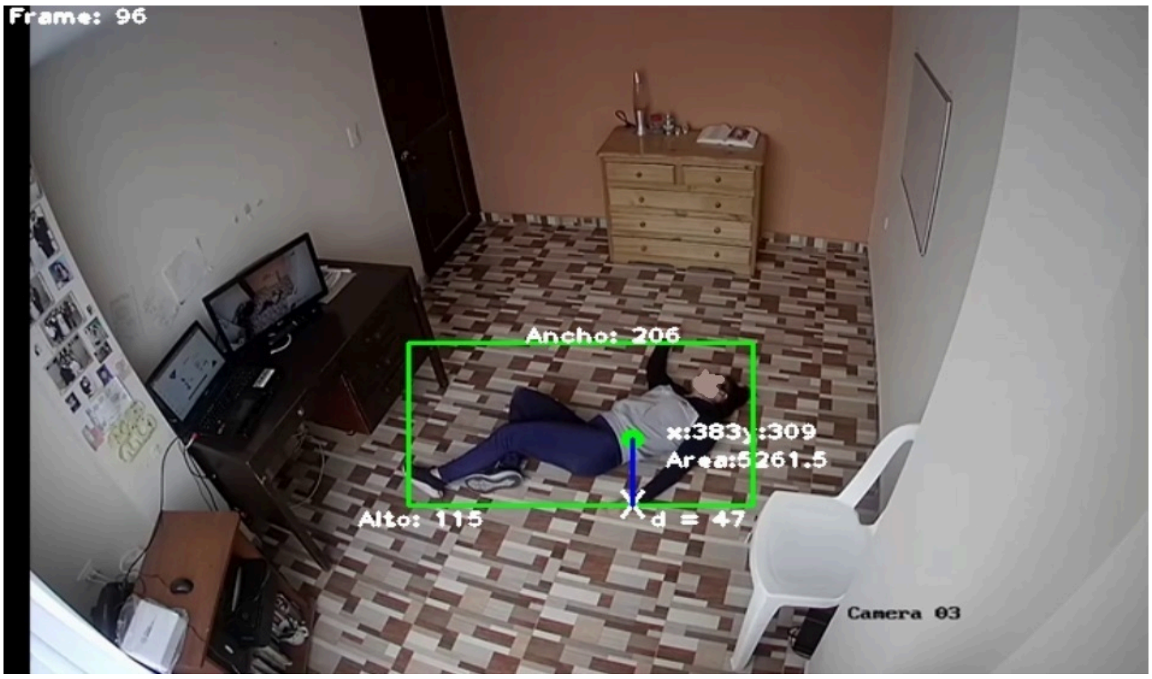
## 1.2 Classification of the CAUCA dataset

Goal: adapt the example of the HDS dataset classification above to build a performant classifier for the CAUCA dataset. Be creative and do not hesitate to explore CNN architectures !

For example, you can implement the CNN architecture shown above in the picture of the part 2 on CNN.

### CAUCA fall detection dataset

The CAUCA dataset is avalaible at: https://data.mendeley.com/datasets/7w7fccy7ky/4 The dataset of falls contains data from 10 different activities: Fall backwards, fall forward, fall left, fall right, fall sitting, hop, kneel, pick up object, sit down, walk. The 20000 labelled images were obtained from videos taken in conditions of an uncontrolled home environment (occlusions, lights, clothes, etc.). Each of the 10 subjects had to follow a specific protocol, containing the 10 different "activities".



We have preprocessed the images for a better classification:

- A YOLO model was used to segment the person in each picture, resulting in square images of size ranging between 200 and 350,
- The images were resized to be 96 × 96 or 224 x 224,
- Labels are merged to get : fall vs. non-fall activity.

The datasets are stored in pickle file which allows to drop Python data structures to a binary file. The structure is a dictionnary with the data X at key "train_data", and the labes y at key "train_labels":

- Storing: *save_to_pickle( "train.pickle", {*
  ```
  *'data': train_data,*
  *'labels': train_labels,*
  *'width': width,*
  *'height': height,*
  *'channels': channels*
  ```
  *})*
- Loading *with open("train.pickle", 'rb') as file: train_dict = pickle.load(file)*

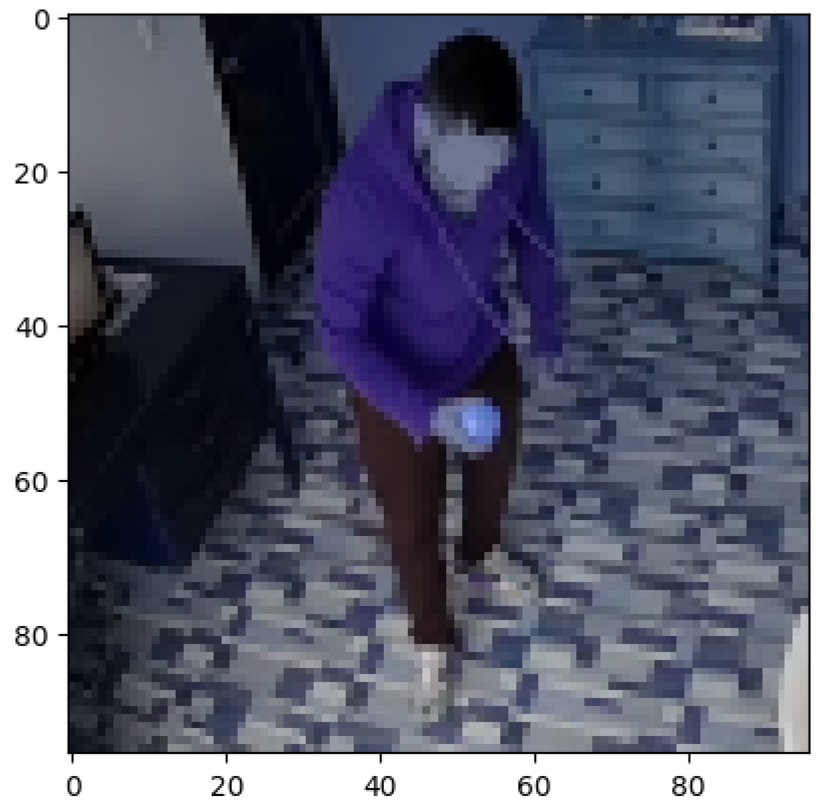
Four sets are available

- Training set (data_train_w_h_c.pickle): images of subjects 1-8,
- Final est set (data_test_w_h_c.pickle): images of subjects 9 and 10.

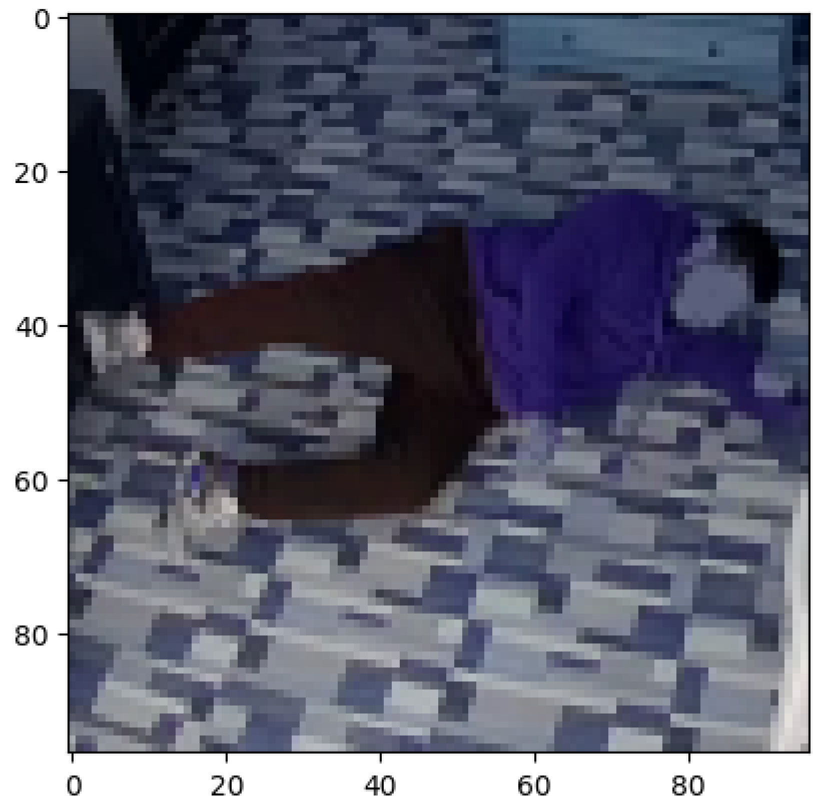
- **TODO: Explore the training data of images 96 x 96 px**
  - load the data from the corresponding pickle file
  - display an image of the train set for each label (0: non-fall, 1: fall) and display the label
  - split the data into train and validation set
  - transform the labels with *to_categorical*

In [37]:
```
# Load the data from the corresponding pickle file
with open("data_train_96_96_3.pickle", 'rb') as file:
    train_dict = pickle.load(file)
X = train_dict['data']
y = train_dict['labels']

# Display an image of the train set for each label (0: non-fall, 1: fall) ar
ind_standup = y.index(0)
print("Image Not Falling: ",ind_standup)
plt.imshow(X[ind_standup],cmap=plt.cm.gray_r,interpolation="nearest")
plt.show()
ind_fall = y.index(1)
print("Image Falling: ",ind_fall)
plt.imshow(X[ind_fall],cmap=plt.cm.gray_r,interpolation="nearest")
plt.show()

# split the data into train and validation set
xtrain, xtest, ytrain, ytest = train_test_split(X, y, train_size=0.8)
xtrain = xtrain.reshape((xtrain.shape[0], 96, 96, 3)).astype('float32') # 96
xtest = xtest.reshape((xtest.shape[0], 96, 96, 3)).astype('float32')

# Transform the labels with to_categorical
ytrain = to_categorical(ytrain)
ytest = to_categorical(ytest)
```

```
Image Not Falling:  0
```



```
Image Falling:  138
```



- **TODO: Questions:**
  - which activation function is generally used for CNN ?

    The relu function for *Rectified Linear Unit* is used for CNNs because it helps to add non-linearity and it is also less long to compute as we have seen on Lab1.

  - what purpose serve the BatchNormalisation layer ?

The BatchNormalization layer helps to normalize the inputs of between each layer to:

1. Get gradients on the same scale.
2. Accelerate convergence.
3. Reduce numerical issues.

- **TODO: Build a CNN classifier for fall detection**
  - Construct as a *Sequential* model the CNN which structure is given as an image in introduction with n1=n2=32, n3=64.
  - Show the number of parameters, and compile the model

In [38]:
```
# Building the CNN model
final_model = Sequential([
    Conv2D(32, kernel_size=(5, 5), activation='relu', input_shape=(96, 96, 3
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(32, kernel_size=(5, 5), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(64, kernel_size=(5, 5), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(64, activation='relu'),  # 64 because the number of neurons in out
    Dense(2, activation='softmax') # 2 because there is two possible output.
])

final_model.summary()
```

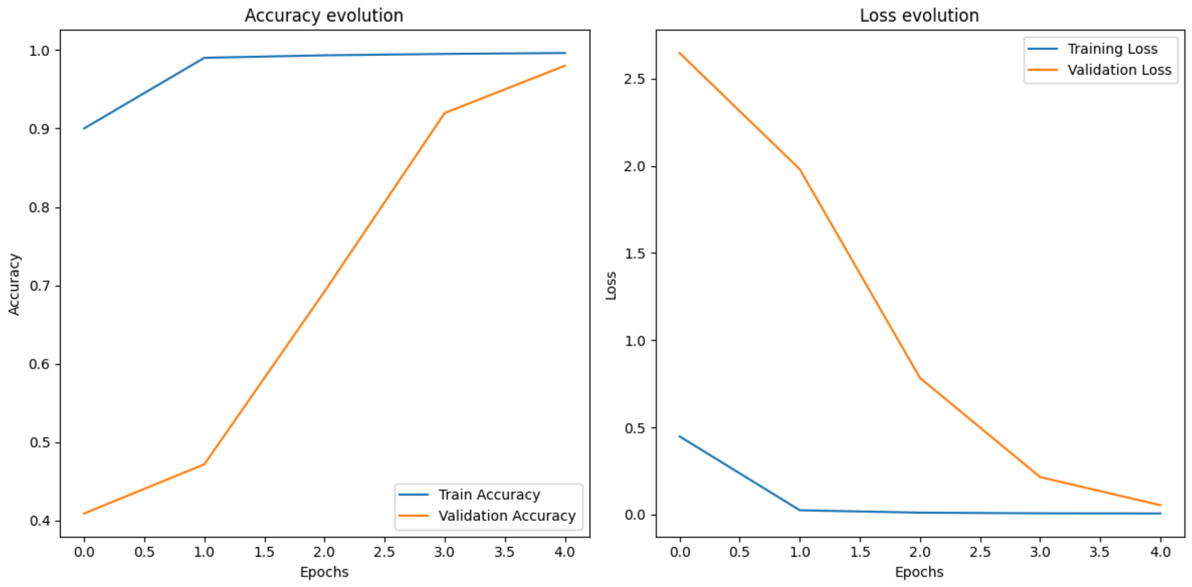
```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d_5 (Conv2D)           (None, 92, 92, 32)        2432      
                                                                 
 batch_normalization_5 (Bat  (None, 92, 92, 32)        128       
 chNormalization)                                                
                                                                 
 max_pooling2d_5 (MaxPoolin  (None, 46, 46, 32)        0         
 g2D)                                                            
                                                                 
 conv2d_6 (Conv2D)           (None, 42, 42, 32)        25632     
                                                                 
 batch_normalization_6 (Bat  (None, 42, 42, 32)        128       
 chNormalization)                                                
                                                                 
 max_pooling2d_6 (MaxPoolin  (None, 21, 21, 32)        0         
 g2D)                                                            
                                                                 
 conv2d_7 (Conv2D)           (None, 17, 17, 64)        51264     
                                                                 
 batch_normalization_7 (Bat  (None, 17, 17, 64)        256       
 chNormalization)                                                
                                                                 
 max_pooling2d_7 (MaxPoolin  (None, 8, 8, 64)          0         
 g2D)                                                            
                                                                 
 flatten_3 (Flatten)         (None, 4096)              0         
                                                                 
 dense_6 (Dense)             (None, 64)                262208    
                                                                 
 dense_7 (Dense)             (None, 2)                 130       
                                                                 
=================================================================
Total params: 342178 (1.31 MB)
Trainable params: 341922 (1.30 MB)
Non-trainable params: 256 (1.00 KB)
_________________________________________________________________
```

- **Training the model**
  - Actual training = fit with 5 epochs (can take some time depending on your architecture),
  - Do not forget to pass the argument *validation_data*,
  - Store the outputs (*history*) !

In [39]:
```python
# Compile the model
final_model.compile(optimizer='adam', loss='categorical_crossentropy', metri

# Train the model
history = final_model.fit(xtrain, ytrain, validation_data=(xtest, ytest), ep
Epoch 1/5
40/40 [==============================] - 10s 237ms/step - loss: 0.4483 -
accuracy: 0.9002 - val_loss: 2.6471 - val_accuracy: 0.4087
Epoch 2/5
40/40 [==============================] - 10s 262ms/step - loss: 0.0246 -
accuracy: 0.9903 - val_loss: 1.9802 - val_accuracy: 0.4714
Epoch 3/5
40/40 [==============================] - 11s 266ms/step - loss: 0.0108 -
accuracy: 0.9934 - val_loss: 0.7832 - val_accuracy: 0.6921
Epoch 4/5
40/40 [==============================] - 11s 270ms/step - loss: 0.0071 -
accuracy: 0.9952 - val_loss: 0.2147 - val_accuracy: 0.9198
Epoch 5/5
40/40 [==============================] - 11s 277ms/step - loss: 0.0062 -
accuracy: 0.9964 - val_loss: 0.0537 - val_accuracy: 0.9802
```

- **TODO: Evaluate the model:**
  - display the evolution of the accuracy over the training iterations of the CNN
  - display the accuracy of the model on the train and validation sets,
  - compute and display the confusion matrix of the model.

In [40]:
```
# Display the evolution of the accuracy over the training iterations of the
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Accuracy evolution')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Loss evolution')

plt.tight_layout()
plt.show()

# Display the accuracy of the model on the train and validation sets
y_pred = final_model.predict(xtest)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = np.argmax(ytest, axis=1)
conf_matrix = confusion_matrix(y_true, y_pred_classes)
print(conf_matrix)

# Display the accuracy of the model on the train and validation sets
loss, accuracy = final_model.evaluate(xtest, ytest)
print(f"Test Accuracy: {accuracy * 100}%")
```



```
40/40 [==============================] - 1s 24ms/step
[[757  19]
 [  6 478]]
40/40 [==============================] - 1s 23ms/step - loss: 0.0537 - accu
racy: 0.9802
Test Accuracy: 98.01587462425232%
```

- **TODO: Final evaluation of the model:**
  - load the data (pickle file) from the test set with images 96 x 96 px
  - transform the labels with *to_categorical*
  - evaluate the performance of the model you selected on this final test set
  - are you satisfied?

    The model has good accuracy, very close to the accuracy with the test set.

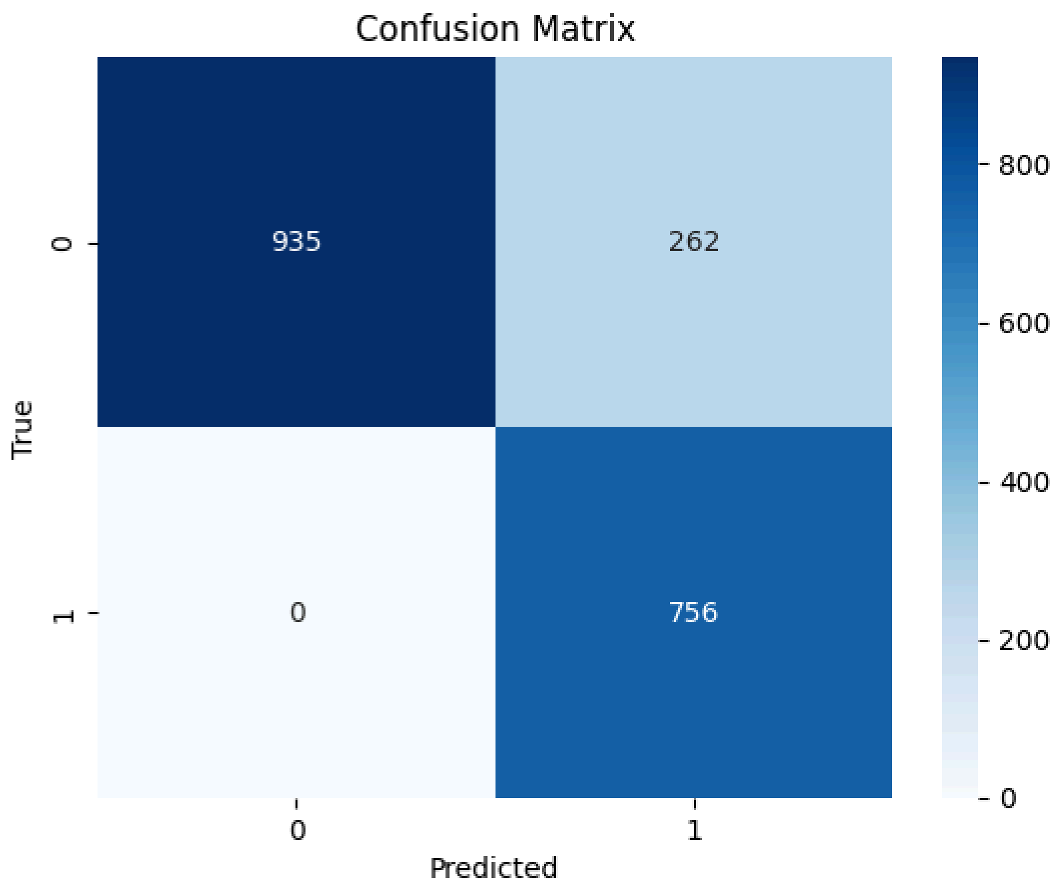
  - do you observe underfitting/overfitting on some models?

    No, the model is neither underfitting nor overfitting.

In [41]:
```python
# Load the data from the corresponding pickle file
with open("data_test_96_96_3.pickle", 'rb') as file:
    train_dict = pickle.load(file)
X_finaltest = train_dict['data']
y_finaltest = train_dict['labels']

# Transform the labels with to_categorical
y_finaltest = to_categorical(y_finaltest)

# Evaluate the performance of the model you selected on this final test set
y_pred = final_model.predict(X_finaltest)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = np.argmax(y_finaltest, axis=1)
conf_matrix = confusion_matrix(y_true, y_pred_classes)
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()

# Calculate the difference in accuracy between test data and final test data
test_accuracy = history.history['val_accuracy'][-1]
```

```
final_test_accuracy = accuracy
accuracy_difference = final_test_accuracy - test_accuracy
print(f"Difference in accuracy between test data and final test data: {accur
```

```
62/62 [==============================] - 2s 25ms/step
```



```
Difference in accuracy between test data and final test data: 0.0%
```

- save your model to the file "best_CNN.keras"

In [42]:
```
final_model.save("best_CNN.keras")
```

In [43]:
```
model = tf.keras.models.load_model("best_CNN.keras")
```

## 2. Compression of the CNN with a combination of pruning and quantization

TensorFlow and TensorFlow Lite include compression mechanisms to decrease the load of a CNN in terms of memory and computation.

We will use:

- the library **tensorflow_model_optimization** for network pruning without and with fine-tuning.
- the library **tf.lite.TFLiteConverter** to convert our model with TensorFlow Lite, with an application of quantization.

Search the web for the documentation, it can be a little hard for these aspects so **don't give up**.

## Given funtions

We give you a small set of functions that will help you during the following of the lab. These functions mainly serve

- to load models, write them to files, and estimate their size (see the doc of each function).
- to define specific extensions of the *tensorflow_optimization* package

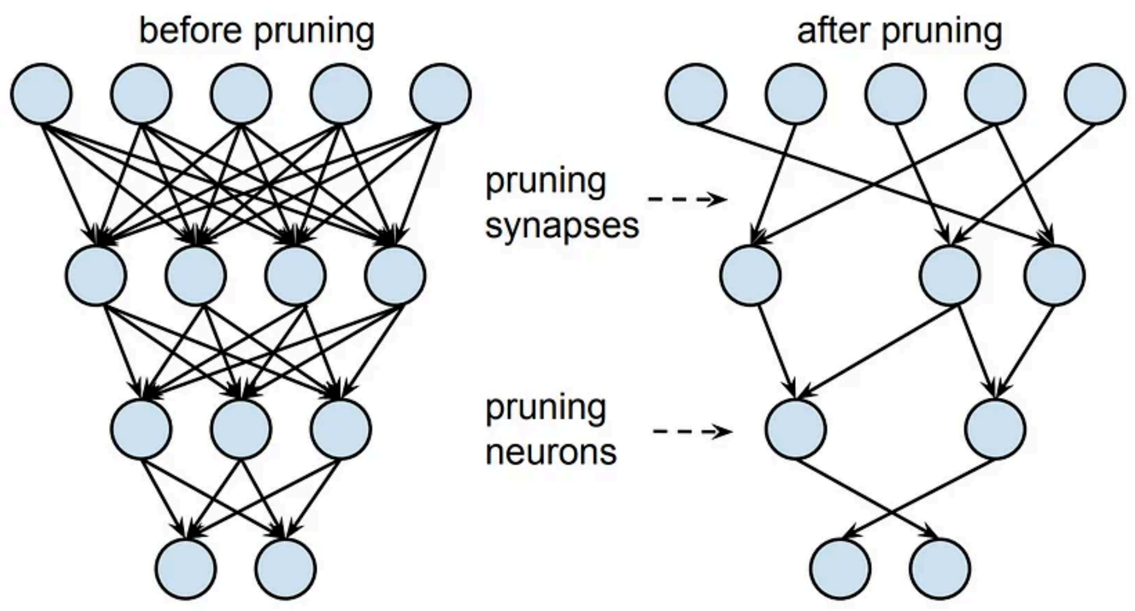
In [44]:
```
def setup_model(model_file):
    """Setup a model from a file where is was written using e.g. model.save(

    Parameters
    ----------
    model_file : str
        The file containing the model

    Returns
    -------
    tf_model
        the loaded model
    """

    model = load_model(model_file)
    return model

def save_pruned_model(model, model_file=None):
    """Save a pruned model after preprocessing it (strip pruneLowMagnitude l

    Parameters
    ----------
    model : tf_model
        The pruned model to save.
    model_file : str (Optional)
        The file where the model should be saved

    Returns
    -------
    tf_model
        The preprocessed model for export
    str
        The file where the model has been saved
    """
    model_for_export = tfmot.sparsity.keras.strip_pruning(model)

    if model_file is None:
        _, model_file = tempfile.mkstemp('.h5')
    tf.keras.models.save_model(model_for_export, model_file, include_optimiz

    return model_for_export, model_file

def get_gzipped_model_size(model=None, keras_file=None, zipped_file=None):
    """Model size estimation:
    Write a model to a file, compress it using gzip and return the file size

    Parameters (at least one of the 3)
    ----------
    model : tf_model (optional)
        The model to estimate.
    keras_file : str (optional)
```

```
        The name of the file where the model is saved.
    zipped_file : str (optional)
        The name of the file containing the compressed model.

    Returns
    -------
    str
        The name of the file where the model is saved.
    str
        The name of the file containing the compressed model.
    int
        The size of the compressed model
    """
    if model is not None:
        if keras_file is None:
            _, keras_file = tempfile.mkstemp('.h5')
        model.save(keras_file, include_optimizer=False)

        _, zipped_file = tempfile.mkstemp('.zip')

        with zipfile.ZipFile(zipped_file, 'w', compression=zipfile.ZIP_DEFLA
            f.write(keras_file)
    elif keras_file is not None:
        _, zipped_file = tempfile.mkstemp('.zip')

        with zipfile.ZipFile(zipped_file, 'w', compression=zipfile.ZIP_DEFLA
            f.write(keras_file)
    elif zipped_file is None:
        raise ValueError("You must provide at least model, keras_file, or zi

    return keras_file, zipped_file, math.ceil(os.path.getsize(zipped_file))

def print_model_weights_sparsity(model):
    """Display the sparsity of each layer in a model

    Parameters
    ----------
    model : tf_model (optional)
        The model to analyze.
    """
    stripped_pruned_model = tfmot.sparsity.keras.strip_pruning(model)
    for layer in stripped_pruned_model.layers:
        if isinstance(layer, tf.keras.layers.Wrapper):
            weights = layer.trainable_weights
        else:
            weights = layer.weights
        for weight in weights:
            if "kernel" not in weight.name or "centroid" in weight.name:
                continue
            weight_size = weight.numpy().size
            zero_num = np.count_nonzero(weight == 0)
            print(
                f"{weight.name}: {zero_num/weight_size:.2%} sparsity ",
                f"({zero_num}/{weight_size})",
            )

def apply_constant_prune(model, sparsity, only_dense=1):
    """Calls tfmot.sparsity.keras.prune_low_magnitude with the possibility t

    Parameters
    ----------
    model : tf_model
        The model to prune.
    sparsity : float
```

```
        The sparsity level to apply ConstantSparsity
    only_dense : int (default=1)
        Do we only apply pruning on the dense layers ?

    Returns
    -------
    tf_model
        the pruned model
    """
    def apply_pruning_to_dense(layer):
        if only_dense and isinstance(layer, tf.keras.layers.Dense):
            return tfmot.sparsity.keras.prune_low_magnitude(layer, ConstantS
        return layer
    model_for_pruning = clone_model(model, clone_function=apply_pruning_to_d
    return model_for_pruning
```

## 1.1 Network pruning

First, we will compress our CNN using a particular approach to network pruning using the function *ConstantSparsity* from the package **tfmot.sparsity.keras**:
https://www.tensorflow.org/model_optimization/api_docs/python/tfmot/sparsity/keras/ConstantSp



This process in done in 3 steps:

- decide on the pruning function: pruning_schedule = tfmot.sparsity.keras.ConstantSparsity(...)
- build your pruning model: pruned_model = apply_constant_prune(model, sparsity, only_dense=1)
- compile and fit: the pruning model is applied during training ! => the new model can be used for prediction, etc.

*Remark*: these steps could be repeated in an iterative process (see Lecture), but we will not do it here to decrease computation time.

- **TODO: Load the data** (yes, again)

- training set (do not forget *to_categorical*) and split it as train/validation sets
- final test set (no splitting)

In [45]:
```
## TRAINING
# Load the data from the corresponding pickle file
with open("data_train_96_96_3.pickle", 'rb') as file:
    train_dict = pickle.load(file)
X = train_dict['data']
y = train_dict['labels']

# split the data into train and validation set
xtrain, xtest, ytrain, ytest = train_test_split(X, y, train_size=0.8)
xtrain = xtrain.reshape((xtrain.shape[0], 96, 96, 3)).astype('float32') # 96
xtest = xtest.reshape((xtest.shape[0], 96, 96, 3)).astype('float32')

# Transform the labels with to_categorical
ytrain = to_categorical(ytrain)
ytest = to_categorical(ytest)

## TEST
# Load the data from the corresponding pickle file
with open("data_test_96_96_3.pickle", 'rb') as file:
    train_dict = pickle.load(file)
X_finaltest = train_dict['data']
y_finaltest = train_dict['labels']

# Transform the labels with to_categorical
y_finaltest = to_categorical(y_finaltest)
```

- Load the model obtained before (done)

In [46]:
```
base_model = setup_model("best_CNN.keras")
baseline_model_loss, baseline_model_accuracy = base_model.evaluate(X_finalte
print('Baseline test accuracy:', baseline_model_accuracy)

base_model.summary()
```

```
Baseline test accuracy: 0.8658474087715149
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d_5 (Conv2D)           (None, 92, 92, 32)        2432      
                                                                 
 batch_normalization_5 (Bat  (None, 92, 92, 32)        128       
 chNormalization)                                                
                                                                 
 max_pooling2d_5 (MaxPoolin  (None, 46, 46, 32)        0         
 g2D)                                                            
                                                                 
 conv2d_6 (Conv2D)           (None, 42, 42, 32)        25632     
                                                                 
 batch_normalization_6 (Bat  (None, 42, 42, 32)        128       
 chNormalization)                                                
                                                                 
 max_pooling2d_6 (MaxPoolin  (None, 21, 21, 32)        0         
 g2D)                                                            
                                                                 
 conv2d_7 (Conv2D)           (None, 17, 17, 64)        51264     
                                                                 
 batch_normalization_7 (Bat  (None, 17, 17, 64)        256       
 chNormalization)                                                
                                                                 
 max_pooling2d_7 (MaxPoolin  (None, 8, 8, 64)          0         
 g2D)                                                            
                                                                 
 flatten_3 (Flatten)         (None, 4096)              0         
                                                                 
 dense_6 (Dense)             (None, 64)                262208    
                                                                 
 dense_7 (Dense)             (None, 2)                 130       
                                                                 
=================================================================
Total params: 342178 (1.31 MB)
Trainable params: 341922 (1.30 MB)
Non-trainable params: 256 (1.00 KB)
_________________________________________________________________
```

- Pruning the model: the parameter *sparsity* for *ConstantSparsity* allows to choose the level of pruning applied.
  - The model should be trained (fit 2 epochs) in order to actually prune the network.
  - There is a callback parameter in the fit function which we provide below:
    - *UpdatePruningStep* is the option to apply pruning at each step of gradient descent,
    - *tfmot.sparsity.keras.PruningSummaries* allows to follow the evolution of the sparsity,
    - launch in a new terminal *tensorboard --logdir={log_dir}* where log_dir is the one generated below.

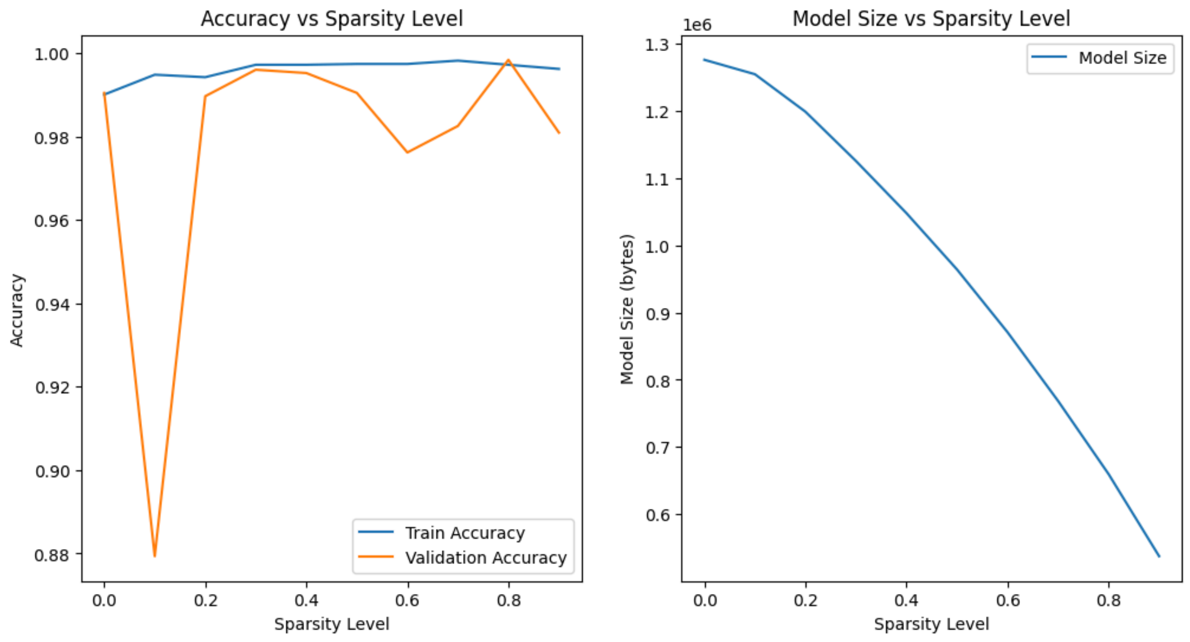
In [47]:
```
log_dir = tempfile.mkdtemp()
print(log_dir)
callbacks = [
    tfmot.sparsity.keras.UpdatePruningStep(),
    tfmot.sparsity.keras.PruningSummaries(log_dir=log_dir)
]
```

```
/tmp/tmp4ibsw7r8
```

- **TODO: optimal pruning of the model**:
  - vary the level of sparsity and plot the evolution of
    - the accuracy for the pruned model (on train and validation),
    - the size of the original and new models as estimated by the provided function get_gzipped_model_size,
  - which choice of sparsity is the best ?
  - Save the corresponding model to the file "constant_pruning.h5" using save_pruned_model.

In [48]:
```
sparsity_levels = np.arange(0., 1., 0.1)
train_loss = []
val_loss = []
train_accuracies = []
val_accuracies = []
model_sizes = []
pruned_models = []

_, _, base_model_size = get_gzipped_model_size(base_model)

for sparsity in sparsity_levels:
    base_model = setup_model("best_CNN.keras")
    pruned_model = apply_constant_prune(base_model, sparsity)
    pruned_model.compile(optimizer='adam', loss='categorical_crossentropy',
    history = pruned_model.fit(xtrain, ytrain, validation_data=(xtest, ytest
    train_loss.append(history.history['loss'])
    val_loss.append(history.history['val_loss'])
    train_accuracies.append(history.history['accuracy'][-1])
    val_accuracies.append(history.history['val_accuracy'][-1])
    _, _, model_size = get_gzipped_model_size(pruned_model)
    model_sizes.append(model_size)
    pruned_models.append(pruned_model)

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(sparsity_levels, train_accuracies, label='Train Accuracy')
plt.plot(sparsity_levels, val_accuracies, label='Validation Accuracy')
plt.xlabel('Sparsity Level')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Accuracy vs Sparsity Level')

plt.subplot(1, 2, 2)
plt.plot(sparsity_levels, model_sizes, label='Model Size')
plt.xlabel('Sparsity Level')
plt.ylabel('Model Size (bytes)')
plt.legend()
plt.title('Model Size vs Sparsity Level')

plt.show()

best_sparsity_index = np.argmax(val_accuracies)
print(f"The best sparsity level is {sparsity_levels[best_sparsity_index]} wi

print(f"The base model size is {base_model_size}bytes and the best pruned mo

compressed_model = pruned_models[best_sparsity_index]
save_pruned_model(compressed_model, "constant_pruning.h5")
```

```
/home/noel/Github/Embedded_IA_TP_INSA/5ISS_ML2/lib/python3.11/site-package
s/keras/src/engine/training.py:3103: UserWarning: You are saving your model
as an HDF5 file via `model.save()`. This file format is considered legacy.
We recommend using instead the native Keras format, e.g. `model.save('my_mo
del.keras')`.
  saving_api.save_model(
Epoch 1/2
158/158 [==============================] - 13s 76ms/step - loss: 0.0458 -
accuracy: 0.9847 - val_loss: 0.2332 - val_accuracy: 0.9214
Epoch 2/2
158/158 [==============================] - 12s 76ms/step - loss: 0.0267 -
accuracy: 0.9901 - val_loss: 0.0228 - val_accuracy: 0.9905
Epoch 1/2
158/158 [==============================] - 13s 77ms/step - loss: 0.0555 -
accuracy: 0.9855 - val_loss: 0.1055 - val_accuracy: 0.9603
Epoch 2/2
158/158 [==============================] - 12s 75ms/step - loss: 0.0200 -
accuracy: 0.9948 - val_loss: 0.9491 - val_accuracy: 0.8794
Epoch 1/2
158/158 [==============================] - 13s 76ms/step - loss: 0.0466 -
accuracy: 0.9831 - val_loss: 0.0676 - val_accuracy: 0.9659
Epoch 2/2
158/158 [==============================] - 12s 76ms/step - loss: 0.0219 -
accuracy: 0.9942 - val_loss: 0.0299 - val_accuracy: 0.9897
Epoch 1/2
158/158 [==============================] - 13s 75ms/step - loss: 0.0447 -
accuracy: 0.9871 - val_loss: 0.0249 - val_accuracy: 0.9937
Epoch 2/2
158/158 [==============================] - 12s 77ms/step - loss: 0.0074 -
accuracy: 0.9972 - val_loss: 0.0091 - val_accuracy: 0.9960
Epoch 1/2
158/158 [==============================] - 13s 75ms/step - loss: 0.0467 -
accuracy: 0.9829 - val_loss: 0.5637 - val_accuracy: 0.8500
Epoch 2/2
158/158 [==============================] - 12s 76ms/step - loss: 0.0080 -
accuracy: 0.9972 - val_loss: 0.0208 - val_accuracy: 0.9952
Epoch 1/2
158/158 [==============================] - 13s 78ms/step - loss: 0.0514 -
accuracy: 0.9839 - val_loss: 0.1753 - val_accuracy: 0.9476
Epoch 2/2
158/158 [==============================] - 13s 81ms/step - loss: 0.0114 -
accuracy: 0.9974 - val_loss: 0.0228 - val_accuracy: 0.9905
Epoch 1/2
158/158 [==============================] - 11s 66ms/step - loss: 0.0391 -
accuracy: 0.9865 - val_loss: 0.0614 - val_accuracy: 0.9849
Epoch 2/2
158/158 [==============================] - 12s 78ms/step - loss: 0.0074 -
accuracy: 0.9974 - val_loss: 0.0822 - val_accuracy: 0.9762
Epoch 1/2
158/158 [==============================] - 13s 79ms/step - loss: 0.0562 -
accuracy: 0.9843 - val_loss: 0.0554 - val_accuracy: 0.9825
Epoch 2/2
158/158 [==============================] - 13s 83ms/step - loss: 0.0050 -
accuracy: 0.9982 - val_loss: 0.0471 - val_accuracy: 0.9825
Epoch 1/2
158/158 [==============================] - 14s 82ms/step - loss: 0.0453 -
accuracy: 0.9853 - val_loss: 0.1456 - val_accuracy: 0.9302
Epoch 2/2
158/158 [==============================] - 13s 81ms/step - loss: 0.0084 -
accuracy: 0.9972 - val_loss: 0.0063 - val_accuracy: 0.9984
Epoch 1/2
158/158 [==============================] - 14s 82ms/step - loss: 0.0596 -
accuracy: 0.9823 - val_loss: 0.0757 - val_accuracy: 0.9746
Epoch 2/2
158/158 [==============================] - 13s 81ms/step - loss: 0.0158 -
accuracy: 0.9962 - val_loss: 0.0640 - val_accuracy: 0.9810
```



```
The best sparsity level is 0.8 with validation accuracy of 99.84126687049
866%
The base model size is 1270414bytes and the best pruned model is 659528by
tes
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics ha
ve yet to be built. `model.compile_metrics` will be empty until you train
or evaluate the model.
```

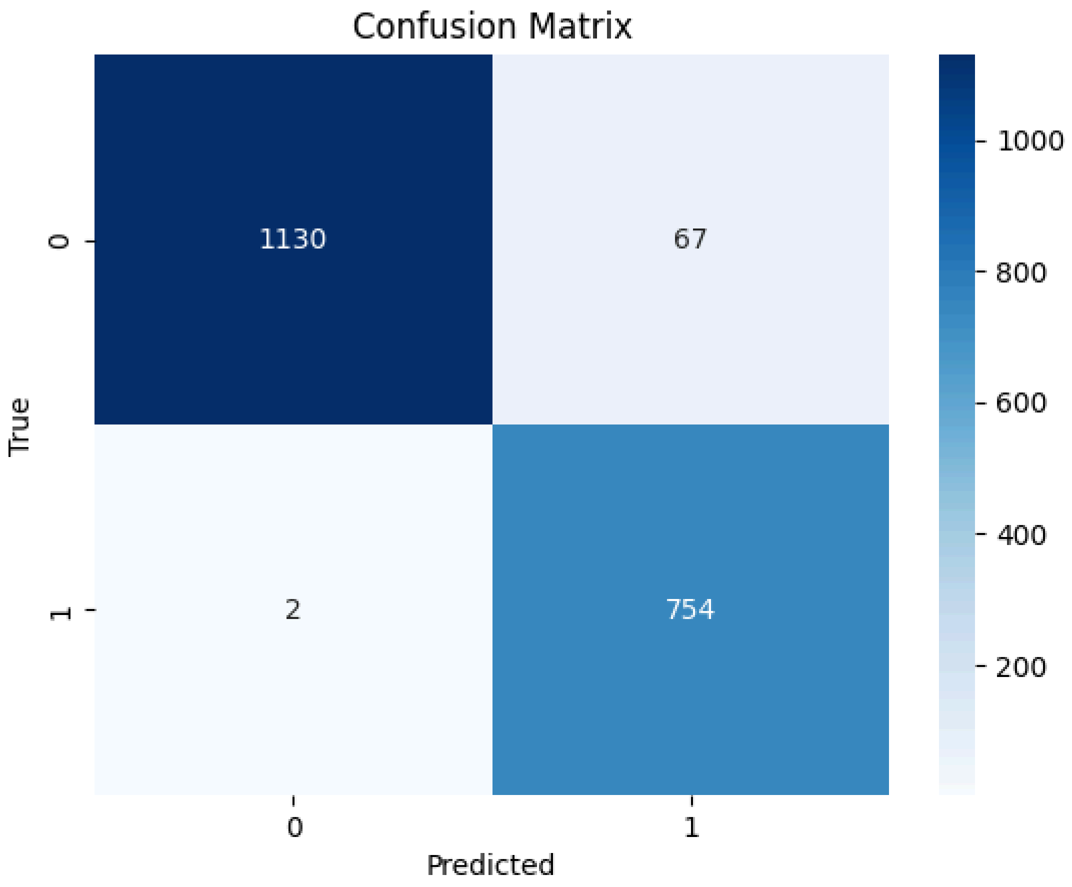
```
/tmp/ipykernel_11955/1507560165.py:39: UserWarning: You are saving your m
odel as an HDF5 file via `model.save()`. This file format is considered l
egacy. We recommend using instead the native Keras format, e.g. `model.sa
ve('my_model.keras')`.
  tf.keras.models.save_model(model_for_export, model_file, include_optimi
zer=False)
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics ha
ve yet to be built. `model.compile_metrics` will be empty until you train
or evaluate the model.
```

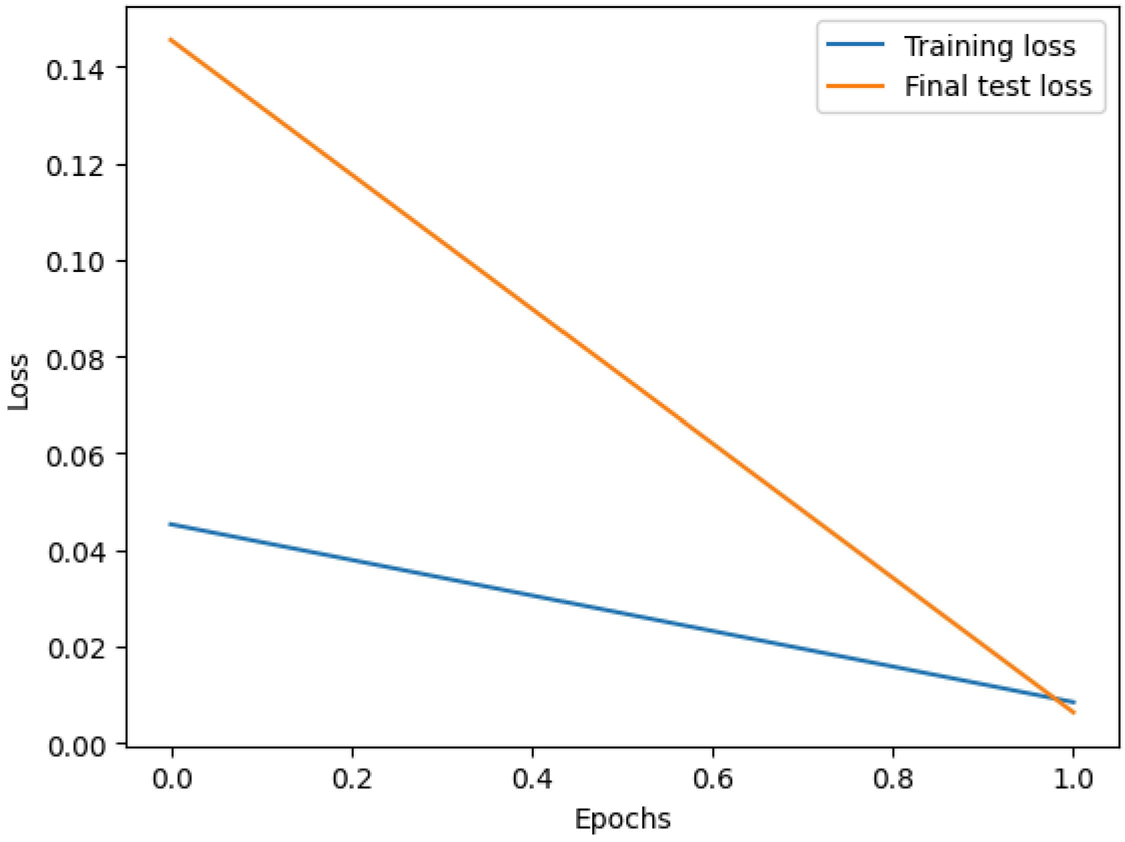
Out[48]:
```
(<keras.src.engine.sequential.Sequential at 0x7fb82b5e9d90>,
 'constant_pruning.h5')
```

- **TODO: For the selected model, evaluate the performance on the final test set**

In [49]:
```
# Evaluate the performance of the model you selected on this final test set
y_pred = compressed_model.predict(X_finaltest)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = np.argmax(y_finaltest, axis=1)
conf_matrix = confusion_matrix(y_true, y_pred_classes)
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()

# plot
plt.plot(train_loss[best_sparsity_index], label='Training loss')
plt.plot(val_loss[best_sparsity_index], label='Final test loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

```
62/62 [==============================] - 2s 28ms/step
```





## 1.2 Quantization

Secondly, we will compress our CNN using quantization optimization as implemented in TensorFlow Lite. The quantization is directly performed during the conversion of the TensorFlow model in a file .tflite.

This file can be used directly by a TensorFlow Lite interpreter on an embedded device (if small enough) !

## Given funtions

We give you a small set of functions that will help you during the following of the lab. These functions mainly serve

- to write them to files (see the doc of each function),
- to evaluate the accuracy of the tensorflow lite model.

In [50]:
```
def save_tflite_model(model, model_file=None):
    """Save a TensorFlow Lite model to a file

    Parameters
    ----------
    model : tflite_model
        The tflite model to save.
    model_file : str (Optional)
        The file where the model should be saved

    Returns
    -------
    str
        The file where the model has been saved
    """
    if model_file is None:
        _, model_file = tempfile.mkstemp('.tflite')
    with open(model_file, 'wb') as f:
        f.write(model)

    return model_file

def tflite_evaluate(X, y, model=None, model_file=None):
    """Evaluate a TensorFlow Lite model

    Parameters
    ----------
    X, y : list, list
        The labeled dataset
    model : tf_model_lite (Optional)
        The model to evaluate
    model_file : str (Optional)
        The file where the model is saved

    Returns
    -------
    float
        The accuracy of the model on the dataset
    """
    if model is None and model_file is None:
        raise ValueError("tflite_evaluate: give either model or model_file")

    # Load and run the TensorFlow Lite model on test data
    if model is not None:
        interpreter = tf.lite.Interpreter(model_pcontent=model)
    else:
        interpreter = tf.lite.Interpreter(model_path=model_file)
    interpreter.allocate_tensors()
    input_index = interpreter.get_input_details()[0]["index"]
    output_index = interpreter.get_output_details()[0]["index"]

    # Evaluate the model
    prediction_digits = []
    for test_image in X:
        test_image = np.expand_dims(test_image, axis=0).astype(np.float32)
        interpreter.set_tensor(input_index, test_image)

        interpreter.invoke()

        output = interpreter.tensor(output_index)
        digit = np.argmax(output()[0])
        prediction_digits.append(digit)

    tflite_qaware_accuracy = (np.array(prediction_digits) == np.argmax(y, ax
    return tflite_qaware_accuracy

def convert_to_TFLite(model, optimizations=None):
    """Convert a model to TensorFlow Lite format with possible optimizations

    Parameters
    ----------
    model : tf_model
        The model to convert
    optimizations : list (Optional)
        The optimization options

    Returns
    -------
    tflite_model
        The converted model
    """
    converter = tf.lite.TFLiteConverter.from_keras_model(model)
    if optimizations is not None:
        converter.optimizations = optimizations
    quant_tflite_model = converter.convert()
    return quant_tflite_model
```

- Load the pruned model obtained before

In [51]:
```
pruned_model = setup_model("constant_pruning.h5")
```

```
WARNING:tensorflow:No training configuration found in the save file, so the
model was *not* compiled. Compile it manually.
```

```
WARNING:tensorflow:No training configuration found in the save file, so the
model was *not* compiled. Compile it manually.
```

- **TODO: convert the model to the TensorFlow Lite format**
  - use *convert_to_TFLite* for the conversion: look inside this function and explain very briefly the steps it follows,
  - use *save_tflite_model* to save the obtained model to the file "pruning_quant.tflite".

In [52]:
```
pruned_model_lite = convert_to_TFLite(pruned_model, optimizations=[tf.lite.O
save_tflite_model(pruned_model_lite, "pruning_quant.tflite")
```

```
INFO:tensorflow:Assets written to: /tmp/tmpjyrv0qgm/assets
```

```
INFO:tensorflow:Assets written to: /tmp/tmpjyrv0qgm/assets
2025-01-22 12:01:31.294396: W tensorflow/compiler/mlir/lite/python/tf_tfl
_flatbuffer_helpers.cc:378] Ignored output_format.
2025-01-22 12:01:31.294424: W tensorflow/compiler/mlir/lite/python/tf_tfl
_flatbuffer_helpers.cc:381] Ignored drop_control_dependency.
2025-01-22 12:01:31.294830: I tensorflow/cc/saved_model/reader.cc:83] Rea
ding SavedModel from: /tmp/tmpjyrv0qgm
2025-01-22 12:01:31.295939: I tensorflow/cc/saved_model/reader.cc:51] Rea
ding meta graph with tags { serve }
2025-01-22 12:01:31.295950: I tensorflow/cc/saved_model/reader.cc:146] Re
ading SavedModel debug info (if present) from: /tmp/tmpjyrv0qgm
2025-01-22 12:01:31.299031: I tensorflow/cc/saved_model/loader.cc:233] Re
storing SavedModel bundle.
2025-01-22 12:01:31.326223: I tensorflow/cc/saved_model/loader.cc:217] Ru
nning initialization op on SavedModel bundle at path: /tmp/tmpjyrv0qgm
2025-01-22 12:01:31.336030: I tensorflow/cc/saved_model/loader.cc:316] Sa
vedModel load for tags { serve }; Status: success: OK. Took 41199 microse
conds.
Summary on the non-converted ops:
---------------------------------
 * Accepted dialects: tfl, builtin, func
 * Non-Converted Ops: 13, Total Ops 36, % non-converted = 36.11 %
 * 13 ARITH ops

- arith.constant:   13 occurrences  (f32: 12, i32: 1)



  (f32: 3)
  (f32: 3)
  (f32: 2)
  (f32: 3)
  (f32: 3)
  (uq_8: 4)
  (f32: 1)
  (f32: 1)
```

Out[52]:
```
'pruning_quant.tflite'
```

- **TODO: evaluate your results**: for the base model, the pruned model, and the tflite model
  - display the size of the model using *get_gzipped_model_size*. For the tflite model, input the *.tflite* directly with the parameter keras_file.
  - display the accuracy obtained on the final test set. For the tflite model:
    - use the function *tflite_evaluate* to get the accuracy,
    - look inside this function and explain very briefly the steps it follows.

In [53]:
```python
base_model = setup_model("best_CNN.keras")
pruned_model = setup_model("constant_pruning.h5")
# Display the size of the model using get_gzipped_model_size
_, _, base_model_size = get_gzipped_model_size(base_model)
_, _, pruned_model_size = get_gzipped_model_size(pruned_model)
_, _, tflite_model_size = get_gzipped_model_size(keras_file="pruning_quant.t

print(f"Base Model Size: {base_model_size} bytes")
print(f"Pruned Model Size: {pruned_model_size} bytes")
print(f"TFLite Model Size: {tflite_model_size} bytes")

# Display the accuracy obtained on the final test set
_, base_model_accuracy = base_model.evaluate(X_finaltest, y_finaltest, verbo
_, pruned_model_accuracy = compressed_model.evaluate(X_finaltest, y_finaltes
tflite_model_accuracy = tflite_evaluate(X_finaltest, y_finaltest, model_file

print(f"Base Model Accuracy: {base_model_accuracy * 100}%")
print(f"Pruned Model Accuracy: {pruned_model_accuracy * 100}%")
print(f"TFLite Model Accuracy: {tflite_model_accuracy * 100}%")
```

```
WARNING:tensorflow:No training configuration found in the save file, so the
model was *not* compiled. Compile it manually.
WARNING:tensorflow:No training configuration found in the save file, so the
model was *not* compiled. Compile it manually.
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics ha
ve yet to be built. `model.compile_metrics` will be empty until you train
or evaluate the model.
/home/noel/Github/Embedded_IA_TP_INSA/5ISS_ML2/lib/python3.11/site-packag
es/keras/src/engine/training.py:3103: UserWarning: You are saving your mo
del as an HDF5 file via `model.save()`. This file format is considered le
gacy. We recommend using instead the native Keras format, e.g. `model.sav
e('my_model.keras')`.
  saving_api.save_model(
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics ha
ve yet to be built. `model.compile_metrics` will be empty until you train
or evaluate the model.
Base Model Size: 1270414 bytes
Pruned Model Size: 607041 bytes
TFLite Model Size: 167536 bytes
Base Model Accuracy: 86.58474087715149%
Pruned Model Accuracy: 96.46697640419006%
TFLite Model Accuracy: 96.415770609319%
```

- **TODO: Questions**:
  - Are you satisfied with the results of the compression ?

    The two versions of compression are really interesting due to the realy low lost in accuracy.

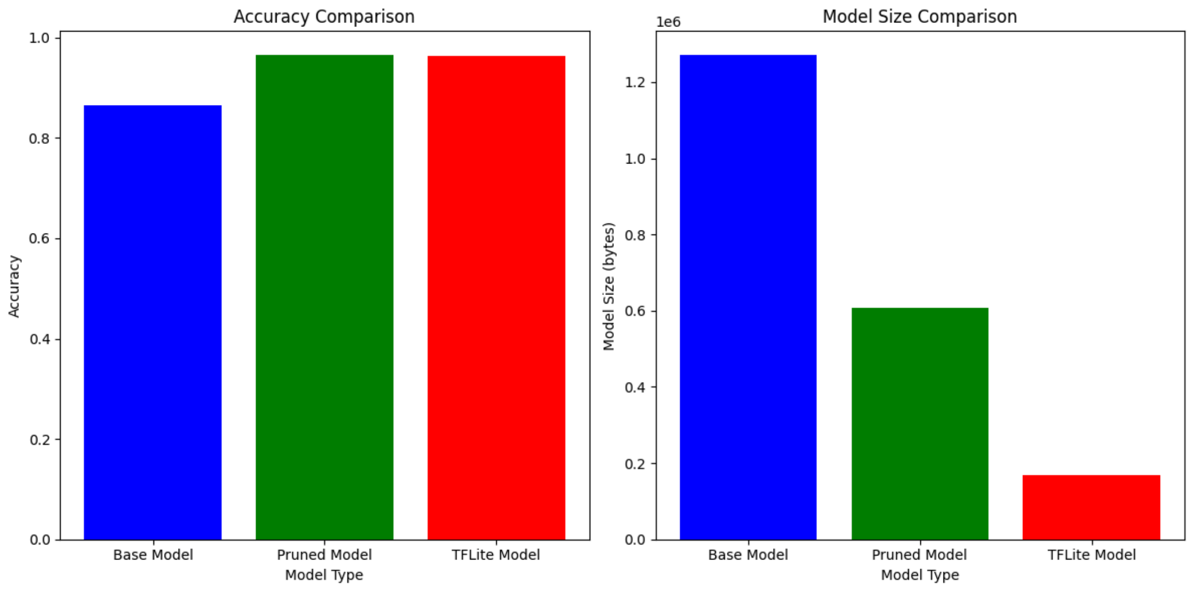
  - Analyse in terms of accuracy, model size.

    Concerning the accuracy, we gain some accuracy for each of the compression which is due to the fact that we removed some neural link that were noises maybe so it's really good when we look at the model size that has been reduced of more than 63%.

In [54]:
```
print(f"Gain of accuracy with pruned model is of {((pruned_model_accuracy-ba
print(f"Gain of accuracy with tfLite model is of {((tflite_model_accuracy-ba

# Plotting the accuracy comparison
plt.figure(figsize=(12, 6))

# Accuracy plot
plt.subplot(1, 2, 1)
accuracies = [base_model_accuracy, pruned_model_accuracy, tflite_model_accur
labels = ['Base Model', 'Pruned Model', 'TFLite Model']
plt.bar(labels, accuracies, color=['blue', 'green', 'red'])
plt.xlabel('Model Type')
plt.ylabel('Accuracy')
plt.title('Accuracy Comparison')

# Model size plot
```

```
plt.subplot(1, 2, 2)
model_sizes = [base_model_size, pruned_model_size, tflite_model_size]
plt.bar(labels, model_sizes, color=['blue', 'green', 'red'])
plt.xlabel('Model Type')
plt.ylabel('Model Size (bytes)')
plt.title('Model Size Comparison')

plt.tight_layout()
plt.show()
```

```
Gain of accuracy with pruned model is of 11.413368483783682%, for a loss
in size of -52.217072544855455%
Gain of accuracy with tfLite model is of 11.354228969878202%, for a loss
in size of -86.81248789764597%
```



- What possible benefits can you hope for embedding in a device? Give at least 3.

  I will base my answer on the BLERP analysis:

  1. Lower bandwidth usage: Local processing reduces the size of the data to transmirt, taking less bandwidth during the data transmission.
  2. Reduced latency: Embedding the model directly on the device allows real-time processing and fast decision making because the output not need to be pull from a server.
  3. Economics transmition: Less data to transmit means a shorter transmission time, resulting in lower power consumption.
  4. Increased reliability: The system could operate independently of network connectivity, ensuring continuous operation even in areas with connection issues.
  5. More privacy: By processing data locally on the device, sensitive data does not need to be transmitted.