

Recoinize - Coin recognition made easy

#keras

#tensorflow

#machine-learning

#coin-recognition

Motivation

Computer vision has advanced significantly in recent years, making it possible for machines to recognize and understand visual content better than ever before. One challenging aspect of this field is recognizing and categorizing objects in digital images, such as coins. Despite state-of-the-art algorithms and techniques, coin recognition remains complex due to their variability.

Recognizing coins has practical applications in vending machines, banking, and finance. Accurate identification and classification of coins can increase efficiency, reduce errors, and improve accessibility for visually impaired individuals. Our proposed approach uses AI to recognize euro coins by training a neural network with a siamese or triplet architecture. This approach provides accurate and efficient results while overcoming the complexity and variability of coin recognition.

Our work has the potential to make significant contributions to the field of computer vision and improve practical applications in various industries.

Dataset creation

The main struggle that this project came with, was the absolute lack of usable pictures of coins we had for three main reasons:

- Firstly, **cash usage is increasingly uncommon** in daily life, with most people opting to pay using credit cards or mobile devices. We thought about asking a cashier at a store for change, but this came with the downside of having to initiate a social interaction with a person we didn't know, which as a computer scientist is known to be close to impossible.
- Secondly, our goal was to **identify one-of-a-kind types of coins**, since our tool targeted collectors and specialists, which would not be readily available in circulation, and **could neither be acquired easily nor for reasonable amounts of money**. For reference, buying about 100 of these limited edition coins would've costed us about 30000€.
- Finally, the process of capturing well-lit and high-quality images of coins demanded **considerable time and effort**, requiring the use of some sort of automated contraption, which we lacked the time or the money to build.

After careful consideration of the pros and cons of each alternative which came to mind, it was deemed best to generate images using a **rendering software** such as Blender. We scrapped hundreds of coins from the internet (more specifically from the European Central bank's website), cut out the excess background, and programmatically generated textures for our coin 3d models using **Hough circle detection** to crop the images to the correct shape.



Fig. 1: Original image scrapped from the ECB website

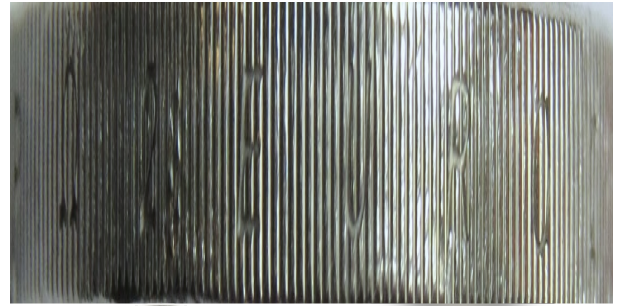


Fig. 2: Programmatically generated cylinder texture

A blender scene consisting of a high resolution cylinder (representing the coin) and a rectangular parallelepiped (which acts as the ground/table upon which the coin lies) will be used to generate the different shots.

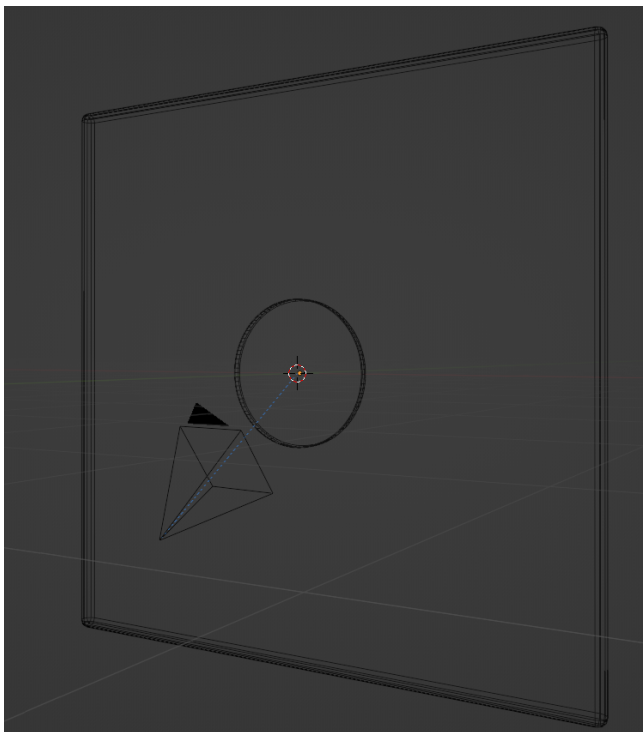


Fig. 3: Wireframe view of the scene

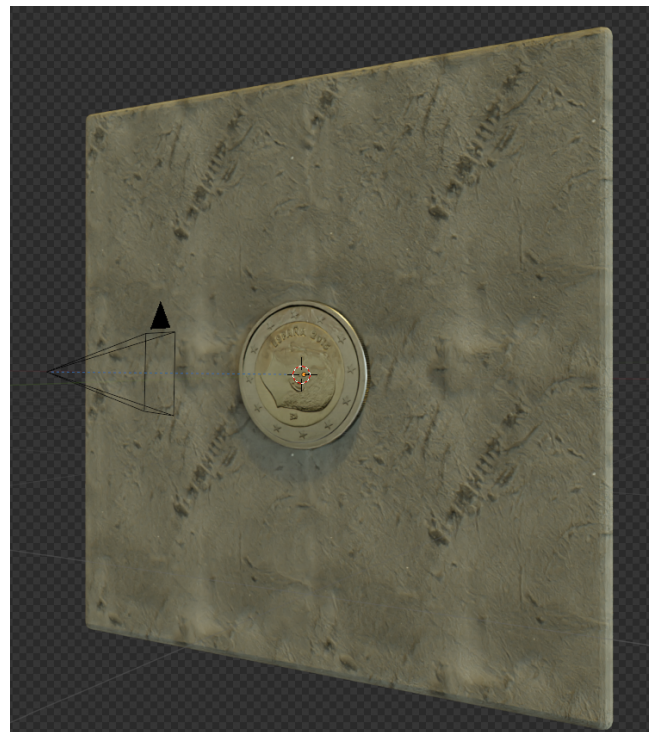


Fig. 4: Rendered view of the scene

The rendering process is fully automated using a python script, which single-handedly **randomizes** the **camera position** (whose frustum is visible in *Fig. 3* & *Fig. 4*), the **lighting**, the

concrete texture in the **background**, and the amount of rust on the surface of the coin, using shaders. This process of randomization and rendering is executed X amount of times for each coin, where X is a predefined constant, which enables the generation of a nearly infinite amount of images from just a couple of pictures downloaded from the Internet.

However, one of our major concern was, and still is, that rendering images using a 3D engine may produce results which would be different from real pictures, in a way that is too subtle for humans to distinguish (missing artefacts, noise or other influencing factors). We will see later how this may have affected our results.




Example code snippets used to automate the scene alteration and rendering process:

```
# Set the camera's position around the coin
bpy.data.objects["Camera"].location = (random.uniform(-2, 2), random.uniform(-2.5, -3),
random.uniform(-2, 2))

# Set the coin's rotation around the y-axis
coin.rotation_euler = (coin.rotation_euler[0], random.uniform(0, 6.28),
coin.rotation_euler[2])

# Render the image
bpy.context.scene.render.filepath = os.path.join(
    OUTPUT_DIR, f"{name}_{i}")
bpy.ops.render.render(write_still=True)
```

Some examples of coins rendered in blender can be seen below, along their associated contour masks, also generated by the script to prevent data loss if we ever needed to crop the image to the exact dimensions of the coin:

	Andorra 5 cents	Austria 2 euros	Andorra 10 cents
Render			
Mask			

Once created and correctly formatted, the dataset is then compiled and sent to Huggingface, where it is now publicly available for download[1]. This allows for a complete separation of the machines that generate the dataset from the ones that use it, thus achieving a dynamic and clean overall pipeline architecture.

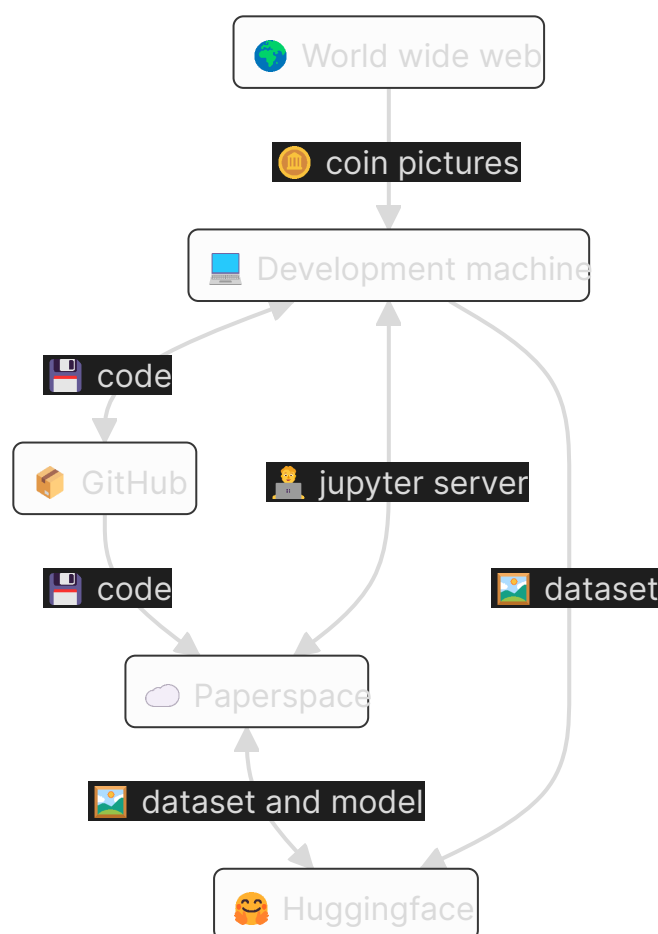
The flowchart below illustrates the overall layout of the infrastructure set up for this project, featuring **GitHub** (Code hosting), **Paperspace** (GPUs and cloud computing) and **Huggingface** (Dataset and model storage).

Workflow

#machine-learning

#coin-recognition

#flowchart



Theoretical background

Coin recognition is a challenging problem in computer vision due to the complexity and variability of coins. Coins come in different sizes, shapes, and colours, and can exhibit a wide range of visual characteristics such as scratches, dents, and dirt. Therefore, developing an efficient and reliable system for coin recognition requires the use of advanced algorithms and techniques in computer vision.

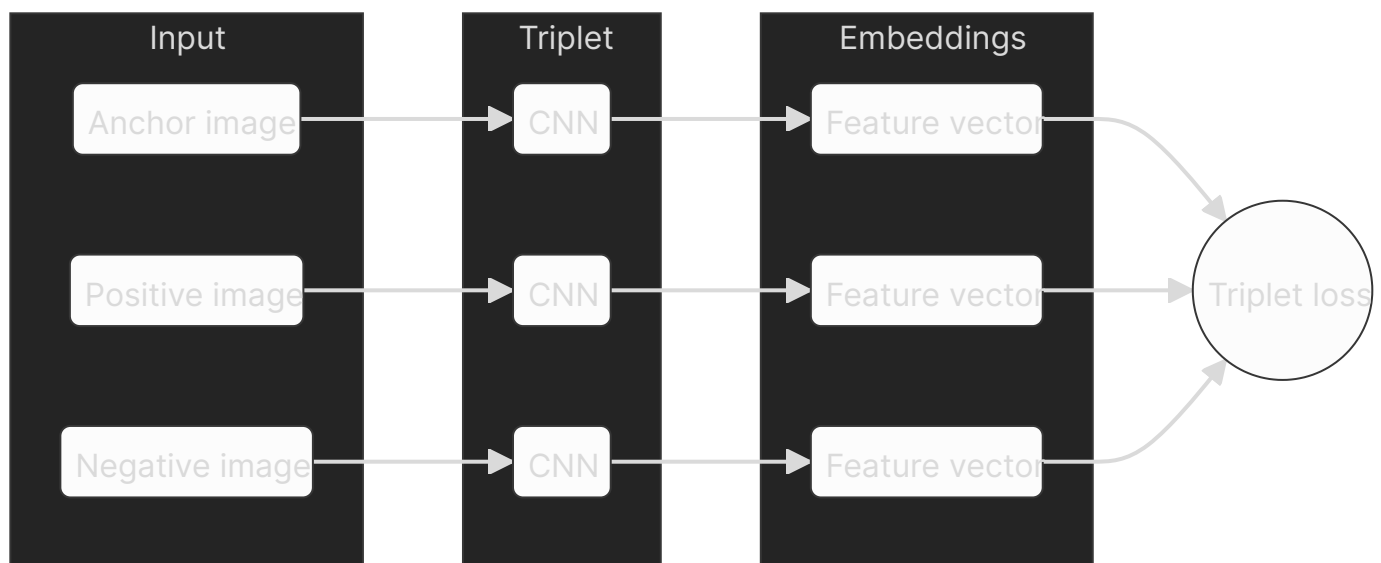
One popular approach to coin recognition is the use of convolutional neural networks (CNNs), which have achieved state-of-the-art performance in various computer vision tasks. However,

traditional CNNs are not well-suited for such a use case because they require large amounts of labelled data to train effectively, which is often not available in this domain. Furthermore, we expect new coins to be released every year, sometimes even at higher frequencies, which would require an update to the model every time an addition is made to the database.

To overcome this challenge, we propose the use of a siamese or triplet architecture, which is a variant of CNNs that are designed to work with small datasets by learning to compare or contrast pairs or triplets of images rather than classifying individual images. This approach allows us to train a network with a smaller amount of labelled data and achieve better performance on the task of coin recognition.

This report will be focusing on the triplet approach, which is based upon the work of the **Machine Perception** team at Google Research. As a matter of fact, triplet loss was first introduced in **FaceNet: A unified Embedding for Face Recognition and Clustering** [2], a paper by Florian Schroff, Dmitry Kalenichenko and James Philbin published in 2015. It supposedly offers better results than siamese training structures thanks to its ability to learn more fine-grained features, which leads to a better discrimination between similar classes.

The triplet architecture consists three identical subnetworks that share the same weights and are trained jointly using triplets of images as inputs. The subnetworks learn to extract features from the images, and the output of each subnetwork is a high-dimensional feature vector that represents the image, its "fingerprint" in a way. The similarity between two images is then computed by comparing the feature vectors using a distance metric such as cosine similarity or Euclidean distance. The network is trained by minimizing the distance between similar pairs or maximizing the distance between dissimilar pairs, as shown below:



In the context of coin recognition, we can use the triplet architecture to train a network to learn the similarity between different images of the same coin or the differences between images of different coins. This approach allows us to overcome the challenges of limited labelled data

and the complexity and variability of coins, as the network can learn to recognize coins based on their unique visual characteristics.

Overall, the use of a siamese or triplet architecture for coin recognition has the potential to significantly improve the accuracy and efficiency of coin recognition systems, making them more accessible and useful in various industries and for individuals who are visually impaired.

Implementation

Preprocessing

Training our model requires prior preprocessing of the images generated in Blender, such as resizing and conversion to grayscale. This is achieved in the `dataset` module, where the `creat_triplets` function reads the raw dataset from Huggingface and organizes it to match the triplet model's structural requirements. The code below illustrates the operation that all **anchor**, **positive** and **negative** images go through while being processed.

```
# remove the alpha channel
anchor_image = anchor_image.convert("RGB")
positive_img = positive_img.convert("RGB")
negative_img = negative_img.convert("RGB")

# resize the images
anchor_image = anchor_image.resize(size)
positive_img = positive_img.resize(size)
negative_img = negative_img.resize(size)

# convert the images to numpy arrays
anchor_image = np.array(anchor_image)
positive_img = np.array(positive_img)
negative_img = np.array(negative_img)

# scale down the images
anchor_image = anchor_image / 255
positive_img = positive_img / 255
negative_img = negative_img / 255
```

Model architecture

As a triplet model, the network is composed of three identical subnetworks that share the same weights and are trained jointly using triplets of images as inputs. Each subnetwork is a CNN whose architecture is defined by the `make_embedding` function, whose code is detailed below.

```
def make_embedding():
    inp = Input(shape=(105, 105, 3), name='input_image')
```

```

# First block
c1 = Conv2D(64, (10, 10), activation='relu')(inp)
m1 = MaxPooling2D(64, (2, 2), padding='same')(c1)

# Second block
c2 = Conv2D(128, (7, 7), activation='relu')(m1)
m2 = MaxPooling2D(64, (2, 2), padding='same')(c2)

# Third block
c3 = Conv2D(128, (4, 4), activation='relu')(m2)
m3 = MaxPooling2D(64, (2, 2), padding='same')(c3)

# Final embedding block
c4 = Conv2D(256, (4, 4), activation='relu')(m3)
f1 = Flatten()(c4)
d1 = Dense(4096, activation='sigmoid')(f1)

return Model(inputs=[inp], outputs=[d1], name='embedding')

```

The triplet structure is then assembled by the `make_triplet_model` function, which handles the calculation of the distance between the feature vectors that the CNN outputs. It is defined in the same module as `make_embedding` and consists of the following instructions:

```

def make_triplet_model(embedding):
    # Anchor image input in the network
    input_image = Input(name='input_img', shape=(105, 105, 3))

    # Positive image in the network
    positive_image = Input(name='positive_img', shape=(105, 105, 3))

    # Negative image in the network
    negative_image = Input(name='negative_img', shape=(105, 105, 3))

    # Combine triplet distance components
    positive_distance = L1Dist()
    positive_distance._name = 'positive_distance'
    positive_distance = positive_distance(embedding(input_image),
                                         embedding(positive_image))

    negative_distance = L1Dist()
    negative_distance._name = 'negative_distance'
    negative_distance = negative_distance(embedding(input_image),
                                         embedding(negative_image))

    # Classification layer
    classifier = Dense(1, activation='sigmoid')(subtract([positive_distance,
                                                         negative_distance]))

    return Model(inputs=[input_image, positive_image, negative_image],
                outputs=classifier, name='TripletNetwork')

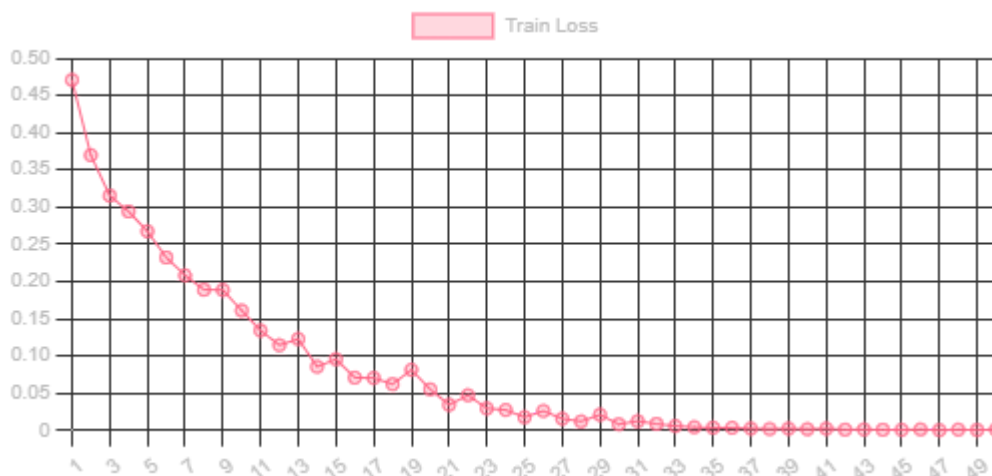
```

Once the model is properly built, and its training structure fully assembled, it is necessary to train it on the previously created dataset. A step which we will be exploring in the following paragraph.

Evaluation

Let us take a first look at the achieved training performance. The following chart illustrates that the training loss consistently decreased throughout the 50 epochs, with a minimum value of 0.00038259 at epoch 46. Despite some slight increases at epochs 19 or 22, among others, the convergence still indicates successful model training. However, it's important to evaluate the model's performance on unseen data, not just the training data.

Training performance



With that goal in mind, we will now take a look at the results obtained during the evaluation of the triplet model. These are calculated by feeding the model with image triplets (anchor, positive and negative as before) selected among a dataset unseen during training, and observing its performance, *ie.* its ability to differentiate coins. It is important to keep in mind that although this indicator is reliable for most cases, our neural network might behave differently if we decide to change the way we determine the distance between two images.

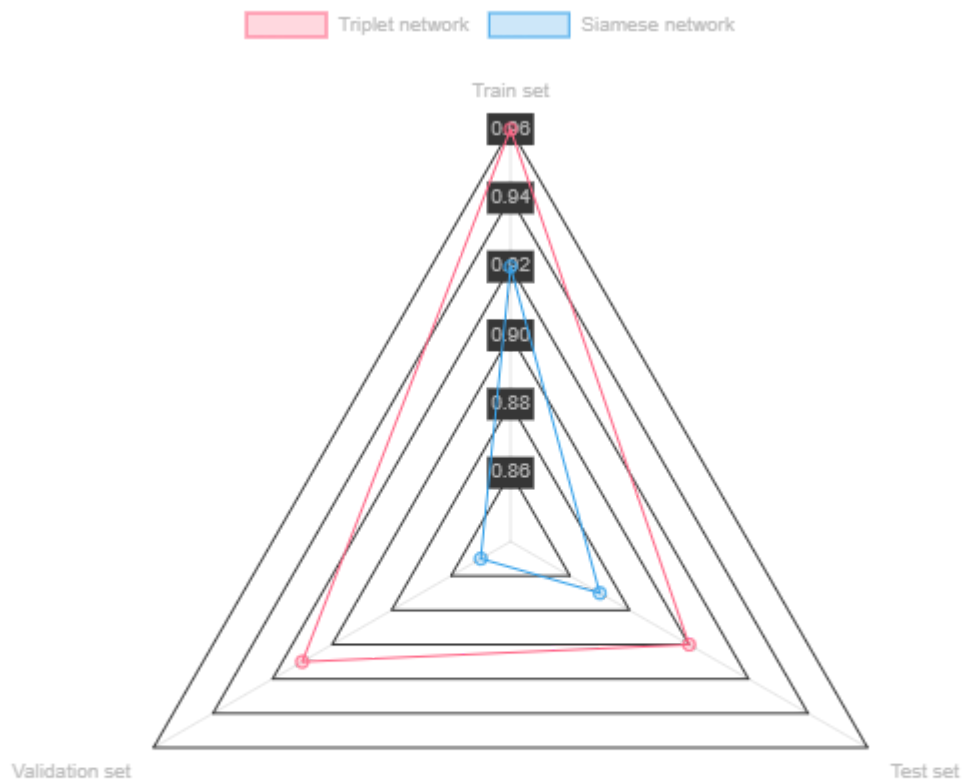
An interesting implementation, that is in addition very relevant to our problem, is the K-nearest neighbours approach. It offers the advantage of being way faster than the conventional method, which consists in iterating over all the already identified pictures of coins in order to find the ones that might be part of the same cluster, by only iterating, as its name indicates on the K nearest neighbours.

Although I did work on an early implementation of this algorithm for a hypothetical integration into a standalone app, I deemed unnecessary and irrelevant to include it here, mostly for code clarity reasons.

The Triplet model's performance can be seen below, along with an implementation of a Siamese model's results, for comparison. It is clearly visible that this learning structure brings

some major improvements, and is able to identify subtle details which were until now undistinguishable by other approaches.

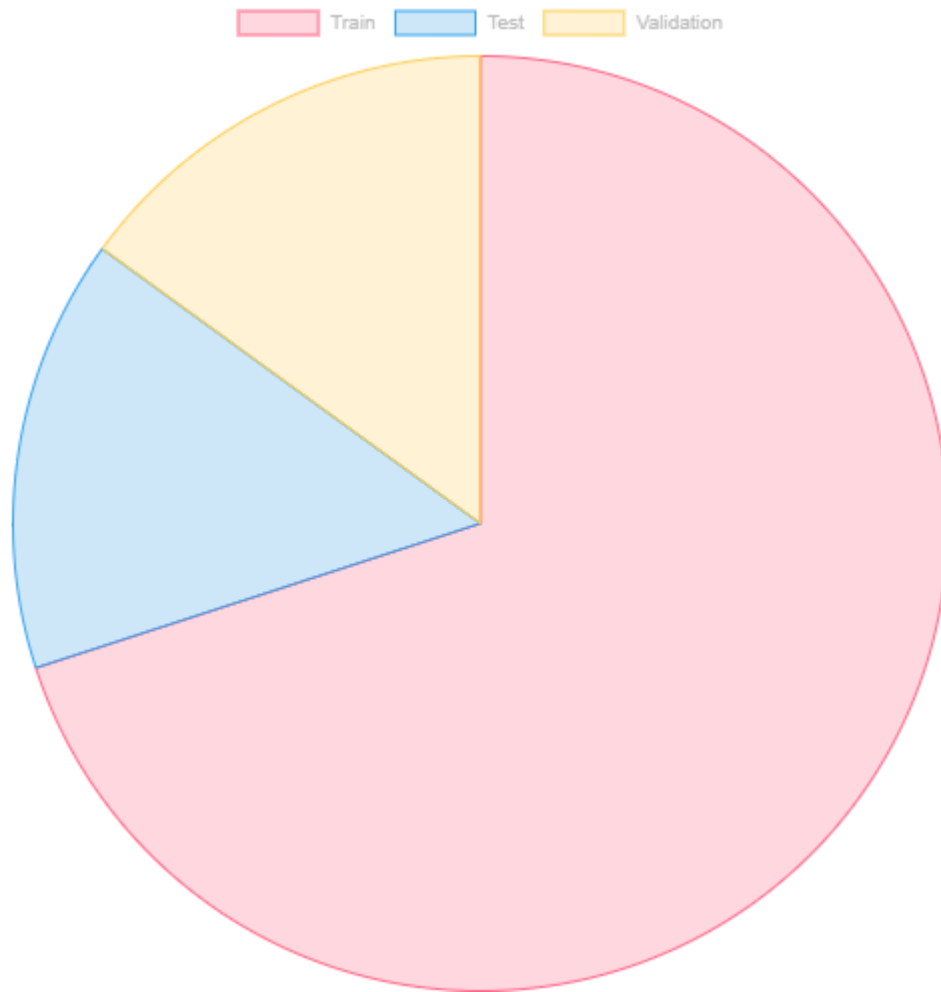
Model accuracy



For reference, the dataset was distributed into **train**, **test** and **validation** sets in the following proportions.

Dataset distribution

Train	Test	Validation
1932	414	414



Note

In this approach, the loss function for the triplet training structure is as follows:

$$L(a, p, n) = \max(0, d(a, p) - d(a, n) + \alpha)$$

where a is the anchor input, p is the positive input, n is the negative input, d is a distance function (such as Euclidean distance), and α is a margin parameter, which can be considered equal to zero for simplicity.

Conclusion

The results obtained may not be nearly as perfect as expected from a production-ready service, but they still do reflect the high potential of this project, which could, with some more tweaking of hyperparameters, make for a handy tool for banking services, collectors and visually impaired people.

A logical evolution of this study would consist in **researching** a better distance and comparison system for the feature vectors associated with the input images, as well as **building** a fully featured phone app along with a cloud-hosted AI inference, for users to be able to use this

service on their mobile device. Of course, such an evolution is purely hypothetical, and does at no point take into account financial constraints, nor does it rely on any prior market research.

1. [photonsquid/coins-euro on huggingface](#)↩
2. [2015 IEEE Conference on Computer Vision and Pattern Recognition \(CVPR\)](#)[2015 IEEE Conference on Computer Vision and Pattern Recognition \(CVPR\)](#)↩