Real-time coins detection with ML based approach on iOS device

Nataliya Didukh, Ihor Zhvanko

Hamburg University of Applied Sciences

Abstract. Machine learning model on iOS device is used to recognise euro coins in real-time. YOLO architecture paper is used to implement the model from scratch in Keras. The classifier as backbone neural network for object detection based on DenseNet architecture with different hyperparameter is trained and evaluated on public dataset. The pre-trained models were converted to perform detection, trained with custom loss function and evaluated. At the end a developed iOS application uses detector model to perform real-time detection.

Key words: machine learning, classification, detection, yolo, keras, tensorflow, tf, data augmentation, custom loss, coremltools, ios, swift, coremL

1 Introduction

Numismatics is the study or collection of currency, including coins, tokens, paper money, medals and related objects [1]. While the numismatists deal with coins from different countries and epochs, regular people use one sort of coins of national currency with different denomination in their day-to-day payments.

Last year has challenged the assumptions and priorities of many businesses, and consequences of COVID-19 lock-downs with growing economic downturn will continue to make a tough climate for many industries. Yet pandemic has positive influence on tech investments and strategy which determine if business will remain viable and competitive. [2]

According to IT Investment Survey 2020 (Figure 1) cloud computing and remote collaboration are the major technologies that helps to profit from lock-downs, and will remain the top investment areas for 2021. At the same time top 10 investment priorities includes Data Analytic, Machine Learning and AI. Mostly a year ago most businesses tried to adopt the machine learning and AI in their business processes experimenting with possibilities for optimisations. With the pandemic majority turned to the technology to aid their responses, applying it in different areas e.g. demand forecasting and automation. [2]

Even before the pandemic, the future of cash was being discussed as the use of digital payments accelerated. Among the main reasons why people change their payment behaviour during COVID-19 in favor of digital currency are: convenience, the risk of being infected via the banknotes, government recommendations to pay cashless and other reasons. [3]

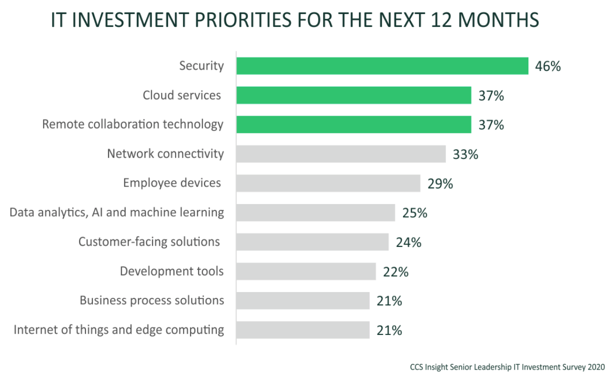

Fig. 1. CCS Insight Senior Leadership IT Investment Survey 2020

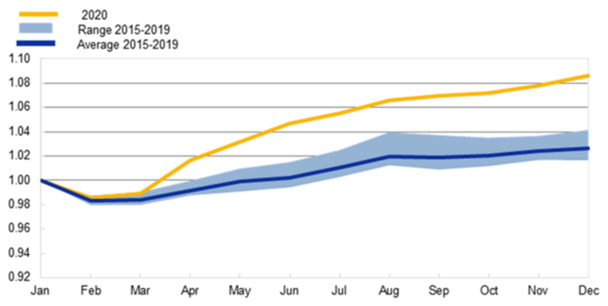

Fig. 2. Total value of banknote circulation in 2020 compared with the previous five (non-crisis) years (2015-19) [3]

During the COVID-19 pandemic arise an unexpected paradox (see Figure 2) - increasing demand for banknotes while cash payment is falling down. The possible explanations for this is that during the pandemic people with low income cut spending and hold the liquid assets to manage uncertainty. [3]

1.1 Motivation and problem statement

What can do a business to encourage people with accumulated physical currency during the pandemic to spend them in supermarkets, stores etc. without having fear to be infected? How can stores maximize the convenience of paying with cash?

As backbone of potential solution machine learning based approach for detecting and recognising currency in real-time is suggested. The system is constrained as proof of concept to recognize only euro coins with denomination: 1, 2, 5, 10, 20, 50 cents and 1, 2 euro, but has the potential to be extended to detect banknotes. The goal is to create an App with machine learning model using modern frameworks and be able to deploy it on any embedded device with constrained computational power.

1.2 Architecture and related works

Under object detection task we understand building the machine learning model for detecting the presence, locating with bounding box and classifying the object(s) on the image. [4] The major complication of detection task is dual prime concern: we want know exact location and concomitantly which class it belongs to. Almost a decade ago it was the most challenging task in Computer Vision before first successful machine learning model R-CNN (Regions with CNN features) was published. [5]

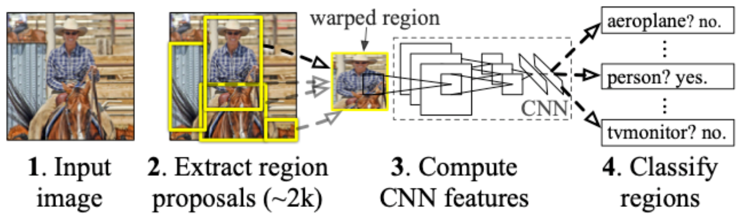

Fig. 3. R-CNN Pipeline [5]

R-CNN model family separates the problem into two main parts: region proposal and classification (see Figure 3). The models are flexible enough to use different region proposal methods and classification architectures. The prime advantage is a state-of-the-art accuracy on benchmark datasets: ILSVRC2013, PASCAL VOC. The major disadvantage is feasibility to use it in real-time object detection due to two stage processing that leeds to dramatic slow down. The pinnacle of model family in context of performance is Faster R-CNN architecture that tackles up to 18 FPS with modern hardware, but despite this videos are typically shot with at least 24 FPS. It means that Faster R-CNN without hardware acceleration will likely not keep pace. [6]

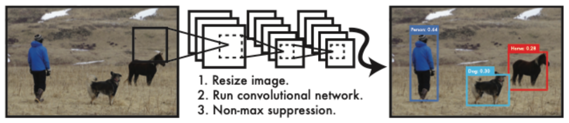

Fig. 4. YOLO Pipeline [8]

Given this was obvious that massive bottleneck in problem separation tends to be somewhat inefficient and focus moved away from two stage detectors. The researchers paid their attention to CNN models that relies on an unified one-state systems. The two main architectures are SSD (Single-shot Detector) [7] and YOLO (You Look Only Once) [8]. The approach involves a single deep CNN trained end-to-end, meaning it takes an input and outputs bounding boxes and corresponding class probability for each detected object. These detectors may be less accurate in comparison to R-CNN family, but it overcomes region-based models inference speed. The modern versions of YOLO can detect objects at 45 FPS, and speed optimized Fast YOLO operate at up to 155 FPS on Titan X GPU. However, the AP (Average Precision) and mAP (mean Average Precision) drops off on vast scale at this elevated speed. [8]

As basic architecture we decided to use YOLO detector architecture. As starting point we use YOLO paper to build the suggested in [8] pipeline from scratch using TensorFlow library and train the model on own designed dataset.

1.3 Target system

On the whole under embedded system we understand the one places into operation for narrow purpose and having the lack of general user interface you might find on an ordinary PC. Older cellphones would rather have a lot in common with embedded systems, but obviously contemporary mobile devices are far away in computational power and versatility. [10]

Our target system iPhone XR based on iOS (Apple Operation System for smartphones) can run machine learning and general purpose apps developed by end-users allowing them to perform tasks not determined by the manufacturer. This generation of iPhone's has front 12 MPX camera module and system can provide up to 60 frames per second. In addition the Apple A11 Bionic is a 64-bit ARM-based system on a chip designed by Apple Inc. that hold first dedicated neural network hardware that Apple calls a "Neural Engine". This piece of hardware can perform up to 600 billion operations per second and can be used by developers for machine learning tasks. [11]

We are going to use iPhone device for real-time inference. Besides inference Apple provides dedicated API for training custom model on device to perform typical machine learning tasks. Online training gives possibility to train the model and save it in the Core ML model format ready to use directly in app. [12] In our project we use ICC Cloud of HAW Hamburg to train and evaluate the model, and after covert it to Core ML format using Apple Python tools to perform object detection. [13]

2 Approach

As we mentioned before in Section 1.2 our decision was to use YOLO detector architecture. According to YOLO paper authors used pretrained convolutional layers on the ImageNet 1000-class competition dataset. Afterwards authors use transfer-learning principle and convert the model to perform a detection. The final model is able to predict several bounding boxes class probabilities and objectness score for each detected object. In post-processing step all boxes are processed with non-maximal suppression algorithm to fix multiple detection (see Figure 4).

Figure 5 summarizes the overall steps to reproduce YOLO model from scratch.

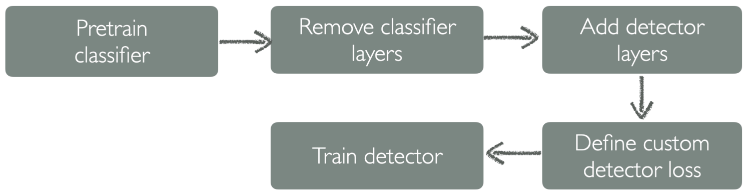

Fig. 5. Project structure

3 Classifier

3.1 Dataset

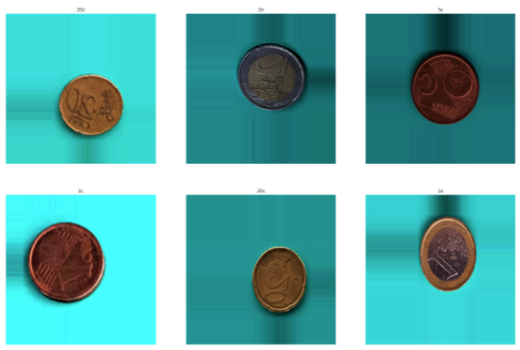

Fig. 6. Dataset images example with augmentation

Three-channel RGB images are used as the input to the convolutional neural network. We expect that background of images are homogeneous. For training and test we used a public dataset accessible on GitHub [14]. This dataset contains 7022 images belonging to 8 classes. (see Figure 6) Every image has already been organized into directories expected by Keras ImageDataGenerator class. This class generates batches flow of tensor image data with real-time data augmentation from directory. The only requirement is sub-directories containing images per class. The major advantages of using ImageDataGenerator are real-time random data augmentation, up- or downsample of images and low memory consumption during training due to one batch in memory at once. [15]

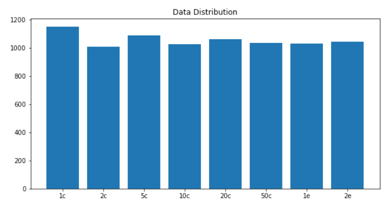

Fig. 7. Data distribution by coins

All data classes are almost equally distributed in raw dataset. (see Figure 7). When splitting up on train and validation dataset it's strongly recommended to preserve distribution inside each data class. [16] While doing test/validation split we took into account this distribution (see Figure 8).

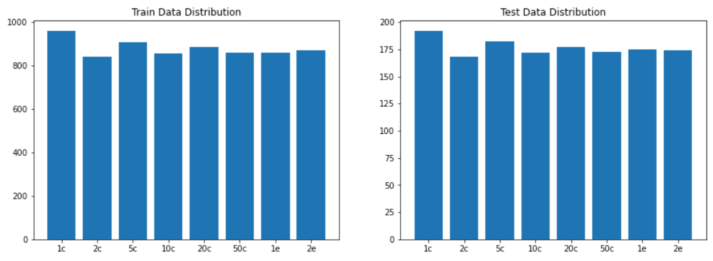

Fig. 8. Data distribution in train and test dataset

All training and test data are passed through augmentation pipeline to increase the overall amount of images. Data augmentation is a remedy for overfitting problem and surpass model generalization skill. [17] The pipeline includes random width/height shift, brightness change and zooming in/out. At the end of pipeline the image is normalized in every channel in range from -1 to 1. (see Listing 1)

```
def preprocess_image(tensor):
    return tensor / 127. - 1.0

train_data_flow = ImageDataGenerator(
    width_shift_range=.3, height_shift_range=.3,
    zoom_range=(1.5, 2.5), brightness_range=(0.7,1.3),
    preprocessing_function=preprocess_image
).flow_from_directory(IMAGES_TRAIN_DIR,
    classes=CLASSES_LIST, target_size=(IMAGE_SIZE, IMAGE_SIZE),
    color_mode='rgb', class_mode='categorical',
    batch_size=BATCH_SIZE, shuffle=True
)
```

Listing 1: Data augmentation configuration

3.2 Model

The invention of deep convolutional network has lead to a great breakthrough in image classification task. [18] The building block of modern classification models are convolutional layers that take into account local patterns in image. The essence of training of a such network is to find the set of filters that generate a feature maps that are passed to next filter set. The stacked convolutional layers naturally integrate low/mid/high-level features and classifiers at the end use this information to do a prediction. [19]

With a deep of network arise new problems: vanishing/exploding gradient that slows down convergence or leads to divergence of the model from the beginning; overfitting that affects poor generalization capability. To prevent a potential issues we used:

- **Rectified Linear Unit**. ReLU is an activation function defined as a positive part or it's argument $f(x) = max(0, x)$. The major advantage is better gradient propagation, faster and effective training in deep neural networks architectures. [20] In our project we used modified version of this function - leaky ReLU:

$$f(x) = \begin{cases} x & \text{if x > 0} \\ 0.01x & \text{otherwise} \end{cases}$$

- **Kernel initialization and regularizer**. He initialization introduced by Kumar in paper is proven mathematically to converge faster and helps to avoid exploding gradient at the beginning. [21] In our project we also used l2-regularizer to penalize higher weight terms during the training process. This technique also helps with exploding gradient problem in a way the weights at the beginning will not grow so fast. [22]
- **Overfitting**. Image augmentation and batch normalization prevent model overfitting and help model to improve generalization capability. [17] In addition batch normalization eliminates the need for Dropout, because it acts as a regularizer. [23]

Researching the history of best classification models from LeNet-5, AlexNet, VGG-16, VGG-19 is clear that with going deeper accuracy gets saturated (which might be unsurprising). All subsequent attempts have shown that with scaling in deep the model unexpectedly starts rapidly to degrade. The degradation problem was solved by ResNet architecture. The paper introduces the idea of identity layers and shortcut connections. The main idea of ResNet is to bypass a signal from one layer to the next via shortcut connections. [24] The "highway" from previous layers approach was used by another less popular architecture FractalNet. The main key characteristic is that they create short paths from early layers to later layers to guarantee better information flow.

In this project, we decided to use DenseNet architecture that makes use of a simple connectivity pattern to ensure maximum information flow between layers in the network. Every dense block is a set of subsequent layers. Each layer provides it's feature maps as input to all subsequent layers through concatenation. As described in paper this kind of networks can achieve the same accuracy with less parameters, and that's the most important property for system with constrained amount of computational power. The efficiency of a such setup are explained by authors as principle of "collective knowledge", because each layer has access to all the preceding feature-maps. [25]

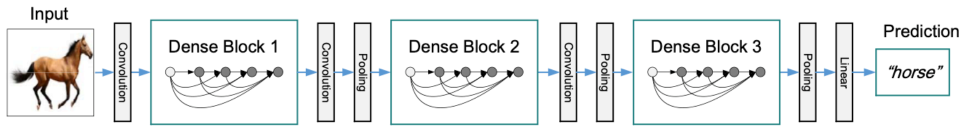

Fig. 9. DenseNet architecture

DenseNet has several hyperparameters that defines the deepness, wideness of the network and compression rate for bottleneck layers. Changing this hyperparameters influence network parameters, accuracy, precision and recall. Without getting into details we list this hyperparameters and values we used in experiments.

- **Growth rate**. If each layer produces k feature-maps, it follows that the subsequent layers have $k_0 + k \cdot (l-1)$ input feature-maps, where k_0 is the number of channels fed in the initial layer and l layer index beginning from 0. For experiments we used $k = 16, 24, 32, 48$.
- **Deepness**. Paper introduce several architectures with the same amount of Dense Blocks, but different repetitions inside the block: 6-12-24-16, 6-12-32-32, 6-12-48-32, 6-12-64-32, thereby the network grow in deepness. We experimented with fixed growth rate $k = 24$ and different setup: 3-6-16-16, 6-12-32-32, 6-12-48-32, 6-12-64-48, but results weren't included in paper.
- **Compression rate**. DenseNet uses so called Bottleneck Layers. Bottleneck layer is a 1x1 convolution introduced to reduce the number of input feature-maps. Authors put this layer right before every 3×3 convolutional layers. The parameter is denoted as Θ and is equal to 0.5 in paper. The same values is used in our experiments.

For training all our models we used initial learning rate 0.01 and categorical crossentropy loss function. For reducing learning rate when model validation loss does not improve we add scheduler that reduces the learning rate after 7 epochs. If model does not improve after 15 epochs we stop training process. (see Listing 2)

```python
# When to save the model
checkpointer = ModelCheckpoint(filepath=MODEL_PATH, verbose=1, save_best_only=True)
# Reduce learning rate when loss doesn't improve after n epochs
scheduler = ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=7, min_lr=1e-8, verbose=1)
# Stop early if model doesn't improve after n epochs
early_stopper = EarlyStopping(monitor='val_loss', patience=15, verbose=0, restore_best_weights=True)
```

Listing 2: Keras callback used in training process

3.3 Experiments and Results

As was mentioned in previous section DenseNet has several parameters, and we tried empirically understand what are the best parameters for our model. We present our experiments with changing growth rate and use DenseNet with repetitions: 6-12-32-32. All metrics we used are very detailed described in [26].

At first we plot confusion matrix to get more visual information where the model mix up the classes. By definition the columns are predicted classes and rows - actual classes. If model has a good classification skill then the diagonal values get saturated.

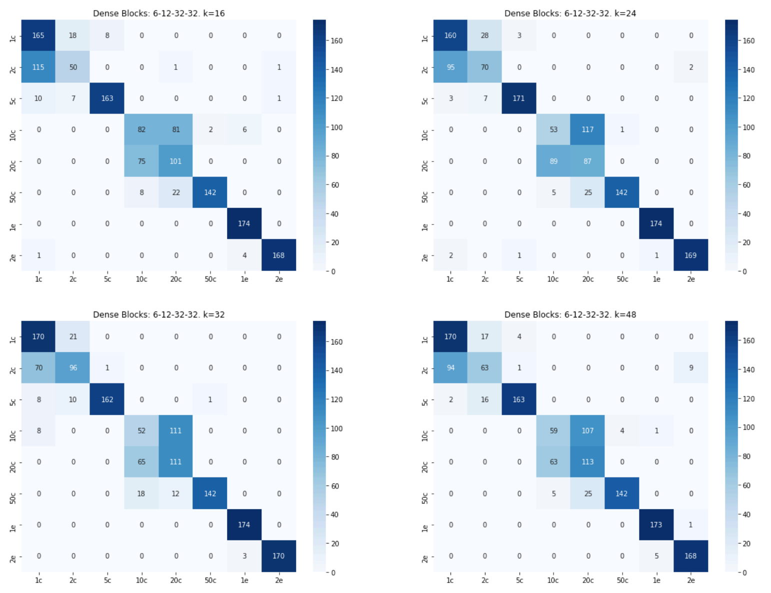

Fig. 10. Confusion matrix for $k = 16, 24, 32, 48$

As expected the most critical groups are 1,2,5 cents and 10,20,50 cents due to great similarity of given coins. In Figure 10 every model has quite noticeable complications with distinguishing 1, 2 cent and 10, 20 cent coins. Our assumption about a such massive confusion is similarity in color and size. We suggest to make image augmentation in HSV or HSL color space to force model learn more structures. This is assumption is based on fact that the smallest and the biggest models have the same behaviour in critical groups.

Also the interesting observation is that low confusions between 1, 2 and 5 cents so as between 10,20 and 50 cent. We assume that both coins 5 and 50 cent inscriptions have noticeable bigger size relative to coin radius.

Basically from initial view on confusion matrix all models classify quite good, and third model with $k = 32$ has shown the best result in critical groups.

To get better feel about overall model performance we also provide accuracy, precision, recall and F1-score for each model and highlight the best scores. F1-Score is a harmonic mean between precision and recall. (see Tables 1, 2, 3, 4).

Growth Rate	Params	Accuracy
16	3.6M	74.22%
24	7.9M	74.8%
32	14M	**78.71%**
48	31.3M	75.2%

Table 1. Model performance in terms of accuracy

Growth Rate	1c	2c	5c	10c	20c	50c	1e	2e
16	0.86	0.30	0.90	**0.48**	0.57	**0.83**	**1.00**	0.97
24	0.84	0.42	**0.94**	0.31	0.49	**0.83**	**1.00**	**0.98**
32	**0.89**	**0.57**	0.90	0.30	0.63	**0.83**	**1.00**	**0.98**
48	**0.89**	0.38	0.90	0.35	**0.64**	**0.83**	0.99	0.97

Table 2. Model precision per class for different growth rate

Growth Rate	1c	2c	5c	10c	20c	50c	1e	2e
16	0.57	0.67	0.95	**0.50**	**0.49**	**0.99**	0.95	0.99
24	0.62	0.67	0.98	0.36	0.38	**0.99**	**0.99**	0.99
32	**0.66**	**0.76**	**0.99**	0.39	0.47	**0.99**	0.98	**1.00**
48	0.64	0.66	0.97	0.46	0.46	**0.97**	0.97	0.94

Table 3. Model recall per class for different growth rate

Growth Rate	1c	2c	5c	10c	20c	50c	1e	2e
16	0.68	0.41	0.93	**0.49**	0.53	**0.90**	0.97	0.98
24	0.71	0.51	**0.96**	0.33	0.43	**0.90**	**0.99**	0.98
32	**0.76**	**0.65**	0.94	0.34	**0.54**	**0.90**	**0.99**	**0.99**
48	0.74	0.48	0.93	0.40	0.54	0.89	0.98	0.96

Table 4. Harmonic mean of precision and recall: F1-Score

4 Detector

4.1 Dataset

Detector network is fed with 3-channel RGB image input containing multiple objects belonging to different classes. Such as classifier dataset all image mainly have homogeneous background. Images are a part of videos recorded by our smartphone, resized and hand-annotated with *labelImg* application (see Figure 12) [27]. During the labeling process this application stores annotated objects in "*.txt"-file for each image in following format: **class**, **x** coordinate of center, **y** coordinate of center, **width**, **height** of bounding box. Bounding box **x**, **y**, **width** and **height** are relative to image size and defined between 0 and 1.

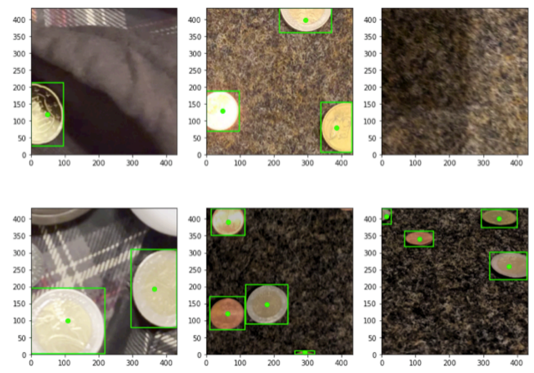

Fig. 11. Dataset images example with augmentation for detection

All training and test data are passed through augmentation pipeline to increase the overall amount of images, avoid overfitting and improve model generalization capability. The pipeline includes random horizontal/vertical flip or both and brightness variation. At the end of pipeline the image is normalized in every channel in range from -1 to 1.

4.2 Model

According to approach described in Section 2 we use our pretrained model in previous section on coin dataset and convert it to perform detection. We remove last two layers: flattening and fully connected layer with *softmax* activation, and replace with a convolutional layer having $(1+4+8) = 13$ filters (see "Output" subsection below). Eventually, network outputs a $7 \times 7 \times 13$ tensor as prediction (see Listing 3).

```
input_tensor = pretrained_model.input

# --------------------------------------------------------------------
# Pretrained Feature Extractor. Required to produce 7x7 feature-maps
# --------------------------------------------------------------------
x = pretrained_model.layers[-3].output

# ------------------------------------
# Detector head
# ------------------------------------
output_tensor = Conv2D(13,
    name='detector_output', kernel_size=(1, 1),
    kernel_regularizer=l2(0.001), kernel_initializer='he_normal',
    padding='same')(x)

model = Model(inputs=input_tensor, outputs=output_tensor)
```

Listing 3: Detector model

Output YOLO architecture has several hyperparameters that characterize network output and spatial capability to detect objects: grid size and amount of bounding boxes.

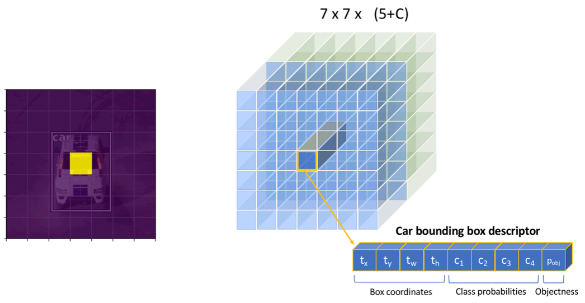
7 x 7 x (5+C)
car
Car bounding box descriptor
t_x t_y t_w t_h c_1 c_2 c_3 c_4 p_{obj}
Box coordinates
Class probabilities
Objectness

Fig. 12. YOLO output encoding

– **Grid size**. YOLO divides the image into an $S \times S$ grid. If bounding box center of an objects falls into grid cell, that grid cell is responsible for detecting that

object. To get lables in YOLO format we need to convert **x**, **y** coordinates relative to grid cell while **width and height remain relative whole image**. In our project we use 7×7 grid (see Figure 12).
– **Amount of bounding boxes**. In real life several objects can occupy one grid cell, therefore model can be extended to output tensor that contains more than one bounding box and class probabilities. Other possibility is to predict more than one bounding box of the same class, e.g. the flock of birds most likely contains densely several objects inside one grid cell. In our project for simplicity we predict only one bounding box per grid cell.

To sum up, the output is $7 \times 7 \times (1 + 4 + 8)$ tensor: 7×7 is a grid, 1 is an objectness score from 0 to 1, 4 is a bounding box and 8 is class probabilities.

As you can notice in Listing 3, the output layer produces a tensor with numbers that might be less than 0 and greater than 1, although we expect from 0 to 1. To obtain the the final predictions an additional post-processing must be performed: calculate logistic function value (*sigmoid*) for objectness score, bounding box and *softmax* for class probabilities. During the training process we take into account additional actions to be taken before computing loss.

Loss During training YOLO authors optimize the following, multi-part loss function:

$$\begin{gathered}
\lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
+ \lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\
+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2 \\
+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2 \\
+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2
\end{gathered}$$

Fig. 13. YOLO custom loss function [8]

First two terms compute the sum of loss for bounding boxes taking into account if grid cell is responsible for object, therefore function penalizes bounding box and classification error if an object is present in grid cell. Before several error terms are weight coefficients which defines the importance of error are described in details in [8].

Also we provide implementation details with TensorFlow, however, it's possible to implement loss function with any Keras backend (see Listing 4).

```
def my_loss(y_true, y_pred):
    # Hyperparameters for loss function
    lambda_coord = 5.0
    lambda_class = 1.0
    lambda_obj = 5.0
    lambda_noobj = 1.0

    # ? - denotes the batch size
    xy_true = y_true[...,:2]     # ? * 7 * 7 * 2 - x,y center
    wh_true = y_true[...,2:4]    # ? * 7 * 7 * 2 - width, height
    xywh_true = y_true[...,:4]   # ? * 7 * 7 * 4 - x,y,width,height
    sure_true = y_true[..., 4:5] # ? * 7 * 7 * 1 - objectness
    class_true = y_true[..., 5:] # ? * 7 * 7 * 8 - class probabilities

    # interpreting predictions
    xy_pred = tf.sigmoid(y_pred[..., :2])      # ? * 7 * 7 * 2 - center
    wh_pred = tf.sigmoid(y_pred[..., 2:4])     # ? * 7 * 7 * 2 - width, height
    xywh_pred = tf.sigmoid(y_pred[..., :4])    # ? * 7 * 7 * 4 - x,y,width,height
    sure_pred = tf.sigmoid(y_pred[..., 4:5])   # ? * 7 * 7 * 1 - confidence
    class_pred = tf.nn.softmax(y_pred[...,5:]) # ? * 7 * 7 * 8 - classes

    loss_xywh = lambda_coord * sure_true * tf.reduce_sum(
        (xy_pred - xy_true)**2 + (tf.sqrt(wh_pred) - tf.sqrt(wh_true))**2,
        axis=-1, keepdims=True
    ) # ? * 7 * 7 * 1 - loss for bounding boxes

    loss_p = lambda_class * sure_true * tf.reduce_sum(
        (class_true - class_pred)**2,
        axis=-1, keepdims=True
    ) # ? * 7 * 7 * 1 - loss for objectness score

    # loss or objectness score taking into account if object is present or not
    # if present then first term won't be 0, otherwize (1.0 - sure_trure) won't be 0
    loss_c =  lambda_obj * sure_true * (sure_true - sure_pred)**2 +
        lambda_noobj * (1.0 - sure_true) * (sure_true - sure_pred)**2

    return tf.reduce_sum(loss_xywh + loss_p + loss_c)
```

Listing 4: Custom loss function implementation with Tensorflow [9]

Training In training process we used the same setup as in classifier training. Initial learning rate is 0.001 (is lower due to model training instability) and is reduced, when model doesn't improve after 7 epochs. After 15 epochs we stop training process if model doesn't improve.

4.3 Experiments and Results

In Section 3.3 we used different hyperparameters for DenseNet and trained several models. For detection task we transfered all 4 models and for each did evaluation. Metrics used for evaluations are described in detail in [28] repository.

Popular way to compare the performance of object detectors is to calculate the area under the curve (AUC) of the Precision x Recall curve (see Figure 14). As AP curves are often zigzag curves going up and down, comparing different curves (different detectors) in the same plot usually is not an easy task - because the curves tend to cross each other much frequently. That's why Average Precision (AP), a numerical metric, can also help to compare different detectors. In practice AP is the precision averaged across all recall values between 0 and 1.

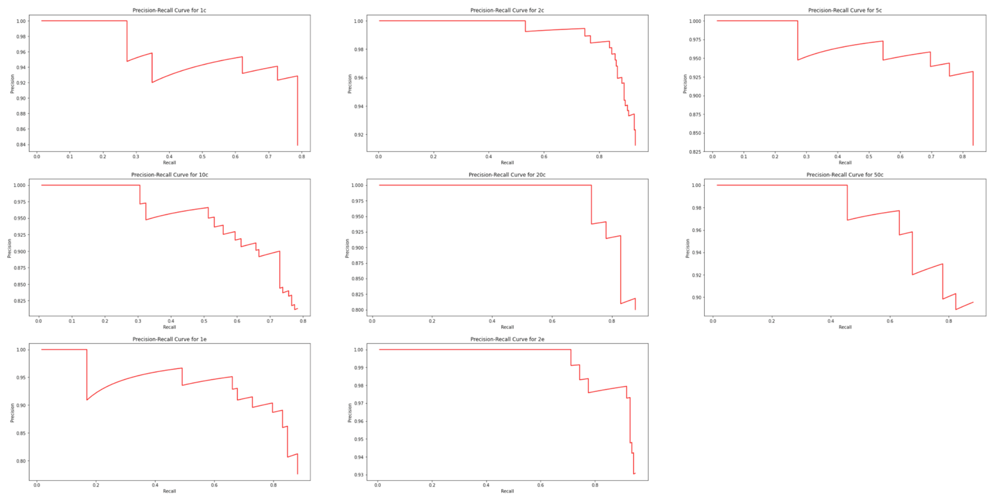

Fig. 14. An example of Precision×Recall curves of detector model for each class based on DenseNet $k = 16$

From 2010 on, the method of computing AP by the PASCAL VOC challenge has changed. Currently, the interpolation performed by PASCAL VOC challenge uses all data points, rather than interpolating only 11 equally spaced points. [28]

Mean Average Precision (mAP) is simply the mean value of each AP for class. Best results are highlighted.

k	mAP	1c	2c	5c	10c	20c	50c	1e	2e
16	**0.83**	**0.74**	**0.92**	0.79	0.74	**0.84**	**0.84**	0.81	**0.94**
24	0.79	0.67	**0.92**	0.73	0.73	0.77	0.71	0.84	0.93
32	0.79	**0.74**	0.91	0.75	**0.76**	0.71	0.76	0.82	0.90
48	0.82	**0.74**	0.89	**0.84**	**0.76**	0.79	0.75	**0.88**	0.92

Table 5. mAP score and AP scores per class

5 Model deployment

As we mentioned in Section 1.3 we use iOS-based device for inference, therefore to develop an application we have to use either Objective-C or Swift. We decided on Swift, because this language is much easier to learn and understand in contrast to Objective-C. Nevertheless, Apple provide additional tools to use both languages in same project to ensure backward compatibility with older libraries.

The first step is to convert Keras TensorFlow model into Apple ML format **.coreml*. For converting the model Apple provides own python helper library *coremltools*. [13] The important points here are:

– `ImageType('input_1', shape=(1,432,432,3) ... )`. It's mandatory to specify what input layer type has the neural network. Shape defines the input tensor and tells *coremltools* that we want feed in exactly one image of 432 × 432 size and 3 channels. If not defined the iOS will interpret the input layer as multidimensional array of undefined shape.
– `bias=[-1,-1,-1], scale=1.0/127.0)`. During the training every image was normalized from range -1 to 1, hence this must be specified in converter. Some ML models use normalization from 0 to 1, accordingly you don't need the bias and scale will be 1/255.

```python
import coremltools

coreml_model = coremltools.convert(
    pretrained_model,
    inputs=[coremltools.ImageType('input_1', shape=(1,432,432,3), bias=[-1,-1,-1], scale=1.0/127.0)],
    source='tensorflow'
)
coreml_model.save(COREML_DETECTOR_MODEL)
```

Listing 5: Converting the model into *coreml*-format

To setup the project and start fetching frames from front camera we used tips described in [29]. When the project was setup and model imported, device started to provide frames from front camera. All frames are directly fed in neural network and developer gets callback with result or error.(see Listing 6)

```swift
func handleMLRequest(request: VNRequest, error: Error?) {
        guard let results = request.results as? [VNCoreMLFeatureValueObservation] else { return }
        guard let observation = results.first else { return }
        guard let multiArray = observation.featureValue.multiArrayValue else { return }

        let output = CoinModelOutput.init(multiArray: multiArray)

        // bounding box rendering from output
        //...
}
```

Listing 6: Handling CoreML callback with model prediction result

The key point here is getting *multiArray* from observation. As described in Section 4.2 the output of YOLO model is a $7 \times 7 \times 13$ tensor, and iOS has no idea how to interpret this result. For interpreting the result we created CoinModelOutput class, which converts a YOLO encoded tensor into the list of bounding boxes. Finally the all bounding boxes are rendered on the top of camera frame (see Figure 15).

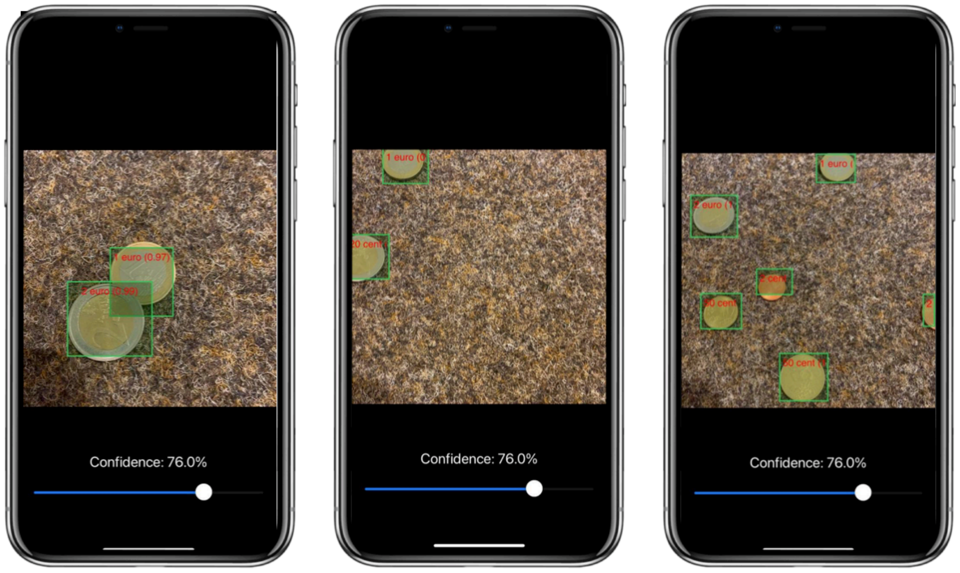

Fig. 15. iPhone App does real-time inference at 30 FPS

We didn't measure the performance, because every our model was able to run at 30 FPS, what is enough for real-time inference.

6 Conclusion

Suggested solution based on machine learning as backbone is feasible for real-time currency recognition. Despite the fact that developed model perform detection only on coins, it can be extended to recognize banknotes and can keep pace with common video frame rate.

In Section 1.2 was described what are possible ML based solutions and their pros and cons in terms of accuracy and performance. R-CNN model family apparently is good for offline recognition due to high accuracy and low performance. SSD and YOLO architectures have shown outstanding inference speed, what is ultimate match for embedded devices.

Considering our goals we have chosen YOLO architecture and implemented from scratch to optimize for our needs. In Section 2 is described very roughly how we proceeded in project step by step to build YOLO model.

In Section 3 we pretrained our classifier model on public coins dataset using DenseNet architecture. Due to high flexibility of DenseNet architecture we experimented with different hyperparameters and reflected all results using different metrics. The evaluation metrics have shown massive flaw present in every model, model confuses 1,2 cents and 10,20 cents and poorly distinguishes them. The possible solutions are suggested.

In Section 4 we use our pretrained classifier model and convert it to perform detection. After converting the model outputs a $7 \times 7 \times 13$ tensor as prediction. For training the model we used custom loss function that takes into account errors from object localization and classification at the same time. Finally we did evaluations for all pretrained models and presented using Average Precision (AP) and mean AP (mAP) metrics.

The very last section 5 describes step-by-step how to deploy the model on iOS device and several significant implementation details.

References

1. Wikipedia. Numismatics. (Access date: 04.08.2021) [Online]. Available: Link
2. Five Highlights from Our C-Suite IT Investment Survey 2020. (Access date: 01.09.2021). [Online]. Available: Link
3. Keynote speech by Fabio Panetta, Member of the Executive Board of the ECB, at the Deutsche Bundesbank's 5th International Cash Conference – "Cash in times of turmoil". (Access date: 01.09.2021) [Online]. Available: Link
4. Object Detection and Recognition Problems. (Access date: 02.09.2021). [Online]. Available Link
5. Ross Girshick and Jeff Donahue and Trevor Darrell and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. arXiv:1311.2524 [cs.CV], 2014.
6. Sánchez Hernández, Sergio & Romero, H & Morales, A. (2020). A review: Comparison of performance metrics of pretrained models for object detection using the TensorFlow framework. IOP Conference Series: Materials Science and Engineering. 844. 012024. 10.1088/1757-899X/844/1/012024.
7. Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, Alexander C. Berg. SSD: Single Shot MultiBox Detector. arXiv:1512.02325v5 [cs.CV], 2016.
8. Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. arXiv:1506.02640v5 [cs.CV], 2016.
9. Tensorflow. API Docs. (Access date: 15.06.2021) [Online]. Available: Link
10. Wikipedia. Embedded system. (Access date: 05.09.2021) [Online]. Available: Link
11. iPhone XR - Technical Specifications. (Access date: 05.09.2021) [Online]. Link
12. Getting a Core ML Model. (Access date: 05.09.2021) [Online]. Link
13. CoreML Tools. (Access date: 05.09.2021) [Online]. Link
14. GitHub. Coin Dataset. (Access date: 05.09.2021) [Online]. Link
15. Keras. ImageDataGenerator. (Access date: 06.09.2021) [Online]. Link
16. What to do when your training and testing data come from different distributions. (Access date: 06.09.2021) [Online]. Link
17. Luis Perez, Jason Wang. The Effectiveness of Data Augmentation in Image Classification using Deep Learning. arXiv:1712.04621v1 [cs.CV], 2017.
18. LeCun, Y.; Boser, B.; Denker, J. S.; Henderson, D.; Howard, R. E.; Hubbard, W. & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. Neural Computation, 1(4):541-551. Available: Paper Link
19. Matthew D Zeiler, Rob Fergus. Visualizing and Understanding Convolutional Networks. arXiv:1311.2901v3 [cs.CV], 2013.
20. Abien Fred Agarap. Deep Learning using Rectified Linear Units (ReLU). arXiv:1803.08375v2 [cs.NE], 2019.
21. Siddharth Krishna Kumar. On weight initialization in deep neural networks. arXiv:1704.08863v2 [cs.LG], 2017.
22. Corinna Cortes, Mehryar Mohri, Afshin Rostamizadeh. L2 Regularization for Learning Kernels. arXiv:1205.2653v1 [cs.LG], 2012.
23. Sergey Ioffe, Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv:1502.03167v3 [cs.LG], 2015.
24. Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Deep Residual Learning for Image Recognition. arXiv:1512.03385v1 [cs.CV], 2015.
25. Gao Huang, Zhuang Liu, Laurens van der Maaten, Kilian Q. Weinberger. Densely Connected Convolutional Networks. arXiv:1608.06993v5 [cs.CV], 2018.

26. Hugo Ferreira. Confusion matrix and other metrics in machine learning. Medium, 2018. (Access date: 09.09.2021) [Online]. Available: Link
27. GitHub. labelImg Application. (Access Date: 10.09.2021) [Online]. Available: Link
28. Padilla, Rafael and Passos, Wesley L. and Dias, Thadeu L. B. and Netto, Sergio L. and da Silva, Eduardo A. B. A Comparative Analysis of Object Detection Metrics with a Companion Open-Source Toolkit. Electronics, Volume 10, 2021. ISSN: 2079-9292. URL: https://www.mdpi.com/2079-9292/10/3/279. GitHub: Link
29. CoreML: Real Time Camera Object Detection with Machine Learning. (Access date: 15.09.2021). Available: Link