Comparison between posit and float representation in a computer vision task using YOLOv5

Erica Dallatomasina[1] and Castrignano Matteo[2]
Federico Rossi[3]

[1]e.dallatomasinia@studenti.unipi.it
[2]m.castrignano@studenti.unipi.it
[3]federico.rossi@ing.unipi.it

ABSTRACT. YOLOv5 is a deep neural network (DNN) used for computer vision tasks, capable of obtaining excellent results, with high accuracy, in object detection operations. Like all DNNs, YOLOv5 has a high computational cost making it difficult to deploy in real-time applications and embedded systems. The computational cost can be lowered by exploiting more efficient representations of the information, such as the posit format. This work will aim to test YOLOv5 performance when using different number representations, using the posit representation directly in YOLOv5 and then comparing the results with the ones obtained with the float representation.

Keywords: *YOLOv5 Posit TensorFlow*

1 Introduction

Real-time applications of Machine Learning (ML) and Deep Neural Networks (DNN) are complex applications and require high computational computing. They can also be implemented in embedded systems, without relying on the internet connection or remote services, and they can reach an high accuracy but at the cost of high computational complexity.

To reduce the computational cost, one can think to some optimizations. One important optimization to be done regards the numerical representation of the information, which affects the energy consumption and the use of resources, resulting in an impact on the hardware.

The standard for real numbers representation is IEEE 754 Floating Point format, but IEEE Floats are starting to get older and to show their limitations. Some alternative formats has been proposed, among these, we find the Posit format, specifically designed to be able to replace the IEEE Floats in ML applications.

The work we are going to present aims to compare the results obtained by exploiting YOLOv5 DNN in an object detection task with posit format and with the standard float representation.

2 Real Number Representations

In this section we will analyze the IEEE 754 Floating Point and Posit formats, which will be used in our experiments.

2.1 IEEE 754 Floating Point standard

The IEEE standard for Floating Point calculation (IEEE 754)[1] is the most widely used standard in the field of automatic calculation. It was established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE) and defines the format for the representation of floating point numbers and a set of operations that can be performed on them. It is a fixed-lenght format, where different sizes are possible:

- half precision format, on 16 bits (FP16)
- single precision format, on 32 bits (FP32)
- double precision format, on 64 bits (FP64)
- quadruple precision format, on 128 bits (FP128)

The IEEE 754 Floating Point representation consists of three fields:

- **Sign**, for which 1 bit is allocated.
- **Exponent**, for which 5 bits are allocated in FP16, 8 bits in FP32, 11 bits in FP64 and 15 bits in FP128.
- **Mantissa**, for which 10 bits are allocated in FP16, 23 bits in FP32, 52 in FP64 and 112 bits in FP128.

In our work, we are going to exploit FP16, FP32 and FP64.

As we mentioned above, the IEEE 754 floating point standard is starting to get older and some problems are becoming more and more evident:

- It is not portable between architectures
- The hardware (the FPU, floating point unit) is very complex and resource costing
- The IEEE 754 standard is never fully implemented in any architecture
- Exception handling is problematic
- They are too general and not optimal for specific applications such as ML

These problems led to the proposal of new formats such as Posit.

[1] https://irem.univ-reunion.fr/IMG/pdf/ieee-754-2008.pdf

2.2 Posit format

The posit standard was introduced in in 2017 by John L. Gustafson[2] to replace the floating point numbers of the IEEE 754 standard.

The Posit numbers allow us to cope with the problems mentioned above with the floating point format.

A posit number has two configuration parameters, *nbits* and *esbits*. The first one defines the length of the representation, which remains fixed, the latter is the exponent length.

The posit number consists of four fields:

- **Sign field**, on 1 bit
- **Regime field**, of variable length (run-length encoding)
- **Exponent field**, of variable lenght (at maximum on *esbit* bits)
- **Fraction field**, of variable length

Just for completeness, we report here the formula used to decode a number in the posit format:

$$x = \begin{cases} 0, & p = 0, \\ \pm\infty, & p = -2^{n-1}, \\ sign(p) \times useed^k \times 2^e \times f, & \text{otherwise} \end{cases} \tag{1}$$

were p is the bit string representing the posit (view as a signed integer) and e and f are respectively the exponent value and the fraction value. k is obtained from the run-lenght l and it is called 'super-exponent'. $useed^k$ is obtained as $2^{(2^{esbits})}$.

[2] http://www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf [Joh]

3 YOLOv5

YOLO (You Only Look Once) [al] is a family of networks used for object detection. As the name suggest, it performs the detection in a single stage, making it possible to reach a great accuracy at high speed.

In our work, we exploited the YOLOv5 model[3]. It has been release by Glenn Jocher, the founder and CEO of Ultralytics, shortly after the release of YOLOv4, using the PyTorch framework in order to simplify its use and speed up its performance. It has been trained on the pre-trained on the MS COCO Dataset (which we briefly describe in section 4).

4 MS COCO Dataset

The MS COCO (Microsoft Common Objects in Context) dataset[4] is a large-scale object detection, segmentation, key-point detection, and captioning dataset. The dataset consists of 328K images.

There exist some different version, but for our tests we have used the 2017 version, given also that YOLOv5 is trained on it. In particular, in the 2017 version the training and the validation sets are provided, with respectively 118K and 5K images. The objects in the images belong to 90 different classes, but only 80 of them have been used for labelling the objects during the YOLOv5 training.

For our work, we have decided to download and use the COCO 2017 validation dataset, which contains 5000 images. Each image comes with an associated text file containing the coordinates of the bounding-boxes and the corresponding label. With these annotations, we can evaluate the performance of the model in terms of predicted labels and bounding boxes coordinates.

[3] https://ultralytics.com/yolov5
[4] https://cocodataset.org/home

5 Set-up and installation

Below will be listed the steps performed to adapt the Posit library in YOLOv5 and to implement the object detection mechanism:

1. Firstly it was necessary to install the Posit library, implemented in Tensorflow and compiled using Python 3.6. So, using a WSL (Windows Subsystem for Linux) we downloaded Python 3.6 and then we used it to install the Posit library.
2. Then, we cloned the YOLOv5[5] repository.
3. After that, it was necessary to instantiate nd export the model. Yolov5 repository provides a file (`export.py`) to do that, but for our purposes it was necessary to have some modifications:
   (a) Even though with the `export.py` file it was possible to export the model as a *SavedModel*, initially it was not possible to export it as a *keras* model. We reported the bug to the Yolov5 autohrs ([6]), who provided to add a flag to make it possible. So after this, it is sufficient to call the `export.py` file with the `--keras` option to export the model as a *keras* model.
   (b) Then we tried to change the weights of the model, to set them in the desired format. We noticed that, even if explicitly setting them in *posit* type, the change had no effect (in fact printing the weights type always returned *float32*).
   (c) Once observed the problem reported at the previous point, we realized that it was necessary to change the model creation and export. So we followed the steps that are done in order to export the model, and we modified the `export.py` and the `tf.py` files [7]. The main changes we did are listed below:
      - firstly we changed the parameters that we have when calling from command line the `export_modificato.py` file, by adding the `custom_type` parameter, which can be equal to 'posit160', 'float16', 'float32' and 'float64'. Depending on this parameter, the model instantiation is done in the desired format.
      - then we modified the instantiation of the layers of the model (in the `tf_modificato.py` file). In particular, the `TFModel` class is responsible of the model creation, and does that by reading a *yaml* file in which the network structure is defined. Then, it calls the `parse_model` function to instantiate all the layers by calling the constructor of the respective classes (each layer of the network is defined as a class which extends the `keras.layers.Layer` class).
        Therefore we added the argument `"dtype = custom_type"` to the `super.__init__()` constructor call of each layer of the network, so that it's possible to change the layer's type.

[5] https://github.com/ultralytics/yolov5
[6] https://github.com/ultralytics/yolov5/issues/7911#issuecomment-1133671255
[7] obtaining the `export_modificato.py` file and the `tf_modificato.py`

After doing those changes, we verify that the model is correctly exported in the desired format.

4. Since YOLOv5 is developed in PyTorch and we work in Tensorflow and keras, we went to integrate and modify the existing code to carry out the loading of the model, the detection of objects in the images and the evaluation of the performances. In particular, the most import changes are the following:

   (a) we modified the `val.py` file [8], by adding the parameter `--float_type` that we can pass from command line. This parameter allows to specify which is the numeric format of the model we are loading (and, as usual, it can be equal to 'posit160', 'float16', 'float32' or 'float64').

   (b) we modified the `common.py` file [9]. In this file, the `DetectMultiBackend` class is responsible of the model prediction, and so we modified it by exploiting the `keras.backend.set_floatx()` function, which allows us to specify with which type of numeric representation we are considering.

   (c) it's importand to mention here that during the overall procedure it was necessary also to implement some conversions between PyTorch tensors and Tensorflow tensors (and viceversa).

5. The last step was to fix some bugs in the tensorflow library. In the `tensor_util.py` file (located under the `tensorflow/python/framework` directory) a series of instructions have been added to recognize and handle the posit format.

It's worth to notice that there are some main steps that we have to do while carrying out the detection:

1. the input images must be resized to 640 x 640 pixels and converted to RGB format. Moreover the pixel values must be normalized.
2. Once the prediction is made, the net returns 25200 bounding boxes. To filter out the non relevant boxes and select just the most significant ones, the *Non-Maximum Suppression* (*NMS*) function must be applied.
3. Finally, the predicted bounding boxes must be re-scaled to the original image size.

Additionally, it's important to remember here that we have to set some parameters in order to carry out the object detection task (they can be specified from command line when calling the program). In particular, the most important ones are the *conf-thres* (representing the confidence theshold) and the *iou-thres* (representing the Intersection Over Union threshold [10]). They are both exploited by the `non_max_suppression` function when filtering out the bounding boxes predicted by YoloV5 network. For our experiments, we set *conf-thres* to 0.001 and iou-thres to 0.6 .

[8] obtaining the file that in our repository is called `val_modificato.py`
[9] obtaining the file that in our repository is called `common_modificato.py`
[10] see section 6.4 for a detailed explanation of IoU

6 Evaluation Metrics

To evaluate the performance of the model, we used the ***Mean Average Precision*** (mAP), which is a good performance indicator for object detection tasks since it incorporates the trade-off between precision and recall allowing also to consider different IoU thresholds.

The formula of the mAP is based on other important metrics:

- Confusion matrix
- Precision
- Recall
- Intersection over Union

In the following paragraphs, we briefly discuss each one of the metrics that we mentioned above and how it is interpreted. Finally, we will introduce the mAP formula (5).

6.1 Confusion Matrix

The confusion matrix is a tool for analyzing the errors made by a machine learning model. It is a contingency table with two dimensions ("predicted" and "actual"), that for each class counts the number of misclassified and correctly classified objects.

It is based on the following concepts:

- *True Positives* (TP) Objects correctly recognized as belonging to a specific class.
- *False Positives* (FP) Objects classified as belonging to a specific class, but that actually belongs to another class.
- *True Negatives* (TN) Objects correctly classified as not belonging to a specific class.
- *False Negatives* (FN) Objects classified as not belonging to a specific class, but that actually belong to that class.

6.2 Precision

Precision measures how accurate the model predictions are, that is the percentage of the predictions which are correct. The formula of the precision is reported in 2.

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

6.3 Recall

Recall measures how the model is able to correctly recognize the positive tuples, given all the positives. The formula of the recall is reported in 3.

$$Precision = \frac{TP}{P} \quad \text{where} \quad P = TP + FN \tag{3}$$

6.4 Intersection over Union (IoU)

Since we are dealing with an object detection task, our class predictions come together with predicted bounding boxes. To measure how well the model is capable of detecting an object, we use the IoU metric, which compares the predicted bounding box and the ground truth bounding box (see figure 1 for a visual representation). The formula of IoU is reported in 4).

$$IoU = \frac{Area\ of\ Overlap}{Area\ of\ Union} \tag{4}$$

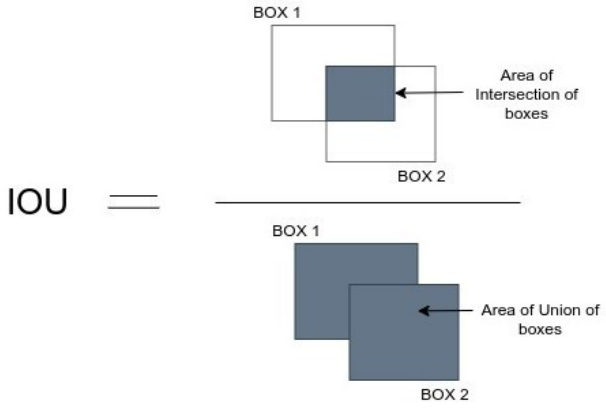

Figure 1: Graphical representation of IoU formula (source https://medium.com/analytics-vidhya/iou-intersection-over-union-705a39e7acef)

For object detection tasks, we calculate *Precision*, *Recall* and mAP using different thresholds, that corresponds to different IoU values.

6.5 Mean Average Precision

Finally, we can briefly discuss about how the mAP is calculated. In particular, the precision-recall curve is computed for different values of IoU thresholds, and then the mAP is calculated by averaging the *Aveage Precision* (AP) over all the classes [Hui]. The mathematical formula is reported in 5.

For the IoU thresholds values, we considered the same values used in the evaluation document of the COCO object detection challenge ([Dat]), that is 10 IoU thresholds ranging from 0.50 to 0.95 in steps of 0.05. With this approach, we consider an object as correctly classified if two conditions hold: the predicted class for the object is the same of the ground truth and the IoU of the predicted bounding box and the ground truth bounding box is greater than the threshold.

$$mAP = \sum_{t \in T} \frac{\sum_{c \in C} AP_c@t / |C|}{|T|} \tag{5}$$

$$T = \text{set of thresholds}$$

$$C = \text{set of classes}$$

$$AP_c@t = \text{average precision computed for class c at threshold t}$$

7 Tests

7.1 Yolov5 weights exploration

Since we will work on translating the YOLOv5 weights into the posit format, it's important to explore the interval in which the weights fall. In particular, the network has in total 7 225 885 weights and they are all contained in the interval [-11.749928, 23.540846], but they are highly concentrated around the zero (as we can see from figure 2).

This is good for our task since the network weights are mainly contained in the range of the posit ring where the resolution is higher. Also for this reason, the conversion error is small. As an example, in figure 3 we report the histogram of the value of the conversion error obtained converting the weights from float32 to posit160 (for which the values are all contained in the interval [-0.0038, 0.0095]).

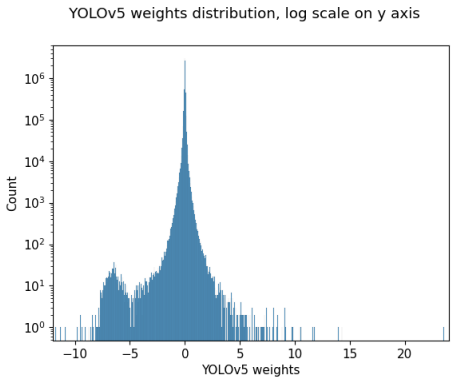

Figure 2: Histogram of the weights distribution of YOLOv5 model, with log scale on y axis to better visualize the counts. There are in total 7 225 885 weights and they are all contained in the interval [-11.749928, 23.540846].

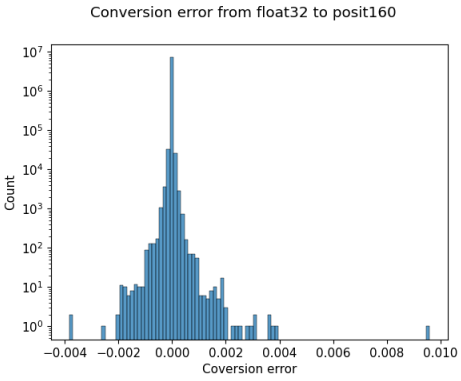

Figure 3: Histogram of the value of the conversion error of the YOLOv5 weights from float32 to posit160. The values are all contained in the interval [-0.0038, 0.0095].

7.2 Result on the COCOval2017 Dataset

7.2.1 Comparison taking into account the mAP and the time spent

The table 1 shows the value of *mAP*, *precision* and *recall* calculated on the COCOval2017 dataset, instead the table 2 reports the times to run the main phases of detection (preprocess, inference and non-max-suppression operations) and the total execution time.

From table 1 we can see that float32 and float64 achieved the same results in terms of mAP, precision and recall. Additionally, the float16 format has a slight decrease in performance (visible at the 4^{th} decimal digit for $mAP@0.5$ and at the 3^{rd} decimal digit for $mAP@0.5:0.95$) while for the posit160 format the decrease in performance is more evident (visible at the 3^{rd} decimal digit for $mAP@0.5$ and at the 2^{nd} decimal digit for $mAP@0.5:0.95$). We can expect these results since float numbers are represented with single precision (so float32) in both PyTorch ([11]) and Tensorflow ([12]). So by switching to smaller formats (like float16 and posit160) we are decreasing the dynamic range and the decimal accuracy, and this results in a loss of accuracy.

Let us now analyze the time dedicated to preprocessing, inference and non-max suppression operations (2). The preprocessing time is mostly the same in all the tests performed, demonstrating that the representation chosen does not affect too much the time required to perform the operations in this phase (in the preprocessing phase we are just normalizing the image pixels and getting some image features like the width and the height). On the other hand, the inference time (which is the most time consuming interval time

[11] https://pytorch.org/docs/stable/tensors.html
[12] https://www.tensorflow.org/api$_d ocs/python/tf/dtypes/DType$

in the overall computation) varies a lot according to the numerical format adopted (4), affecting also the total execution time.

As we wanted to demonstrate, the float32 format has the lowest value compared to all other representations. Therefore, a float representation that is too large (float64) or too small (float16) penalizes the inference time and so we can consider float32 the best trade-off between the representations.

The posit format deserves a separate discussion. We must remember that the posit format is only simulated and has no hardware support, which instead floats have. Because of this, both the inference time and the non-max-suppression time increase considerably, almost doubling the execution time compared to the float32 representation.

Numeric format	mAP@0.5	mAP@0.5:0.95	Precision	Recall
float16	0.564679	0.370728	0.665805	0.522771
float32	0.564721	0.371199	0.665495	0.522367
float64	0.564721	0.371199	0.665495	0.522367
posit160	0.563775	0.366043	0.664919	0.521869

Table 1: *mAP*, *Precision* and *Recall* value measured on the COCOval2017 Dataset. mAP@0.5 is the mAP value calculated for a threshold equal to 0.5. mAP@0.5:0.95 is the mAP calculated by averaging the mAP for 10 different values of thresholds (that range from 0.5 to 0.95 in steps of 0.05).

Numeric format	Avg pre-process time (ms)	Avg inference time (ms)	Avg NMS time (ms)	Total execution time (mm:ss)
float16	0.5	193.6	4.3	19:16
float32	0.5	143.8	4.3	14:40
float64	0.6	199.9	4.4	18:23
posit160	0.6	320.2	5.1	30:12

Table 2: Table showing the average pre-process time, inference time, NMS (Non Max Suppression) time per image and the total execution time spent by the YOLOv5 model applied on the COCO2017val Dataset.

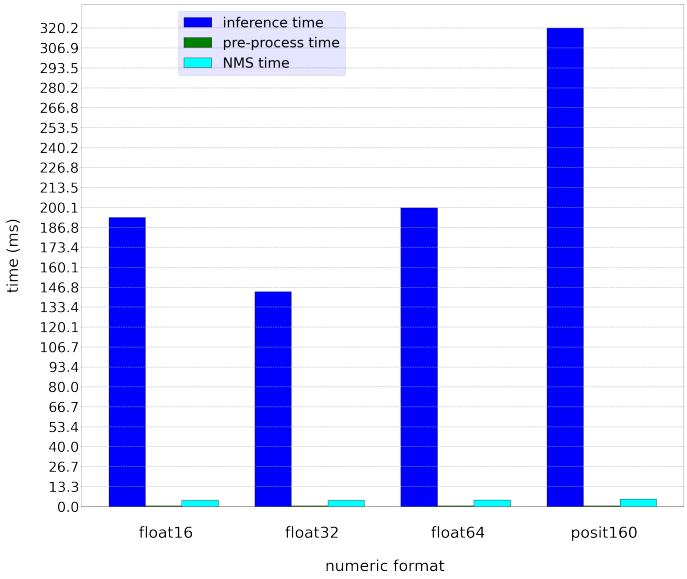

Figure 4: Comparison between different numeric formats in terms of average inference time per image, average pre-process time per image, and average NMS time per image. The differences in average NMS time and average preprocessing time are not visible at this scale. Instead, the impact that the numeric format adopted has is highly visible for the average inference time.

7.2.2 Comparison taking into account the number of predicted bounding boxes per image

Although it's important to compare the model predictions with the ground-truth (through the mAP), for our comparison between posit and floats it's useful also to directly compare the results obtained when using the different formats. To this aim, we further explored the annotations produced by the object detection model to analyze the differences in the predicted bounding boxes.

In particular, by calling the `validation_modificato.py` script with the `--save-json` option, at the end of the detection a json file is saved containing all the information that regard the predicted boxes (like the vertices coordinates, the area, the confidence etc.). By analyzing these files, we observed some interesting things:

- For each one of the numeric formats adopted, there is no image for which the YOLOv5 model is not able to detect any bounding box. In other words, there is at least one predicted bounding box for every image in the COCOval2017 dataset, and this is true for every numeric format adopted.
- There are some images for which the number of predicted bounding boxes is different. In particular, there is just one image for which the number of predicted bounding boxes is different (compared to float32 format) when adopting float64 format. Instead, for posit we have 2206 images for which the number of predicted bounding boxes is different from the number of predicted boxes that we have when using float32. Finally, for float16 format, there are 1789 images for which the number of predicted bounding boxes is different from the number of predicted boxes that we have when using float32.
- Despite the difference in the number of predicted bounding boxes, most of the times this difference is not so high. In particular, calculating mean and standard deviation of the differences between predicted bounding boxes when using float32 and one of the other numerical formats (posit or float16) [13], we obtain that this difference is equal to:

  a) 0.99 +/− 2.42 when considering float32 and posit, with the minimum difference being -14 (this means that there is at least one image for which with posit we predict 14 boxes more than the float32 case) and the maximum difference being 14 (this means that there is at least one image for which with float32 we predict 14 boxes more than the posit case)

  b) -0.39 +/− 2.02 when considering float32 and float16, with the minimum difference being -10 and the maximum difference being 8

  So when using the posit format the model predicts on average less boxes, instead when using float16 it predicts slightly more boxes than the float32 case. To better visualize these results, in figure 5 we report a box-plot showing the distribution of the difference of the number of predicted bounding boxes.

[13] For this calculation, we are considering just the images for which the number of predicted bounding boxes is different.

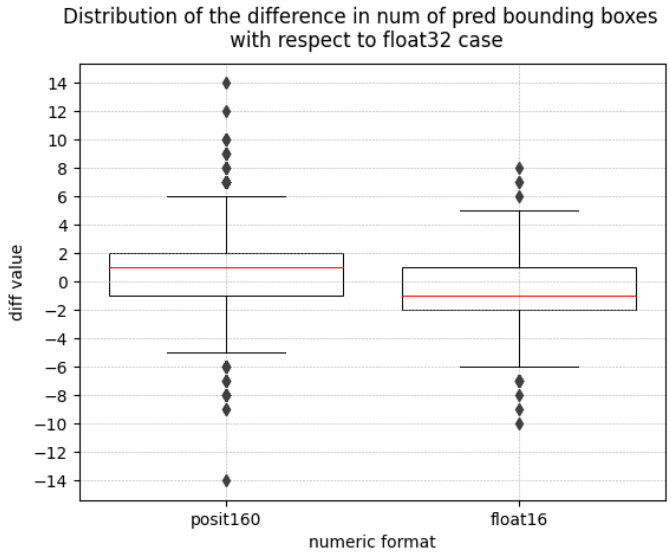

Figure 5: Box-plots showing the distribution of differences of the number of predicted bounding boxes for each image in float32 and posit formats, and in float32 and float16 formats. The red line represents the median. The box-plots are calculated by considering just the images for which the number of predicted bounding boxes is different.

8 Conclusion

To conclude, we can we can state that for the object detection task that we analyzed, with the posit representation we have both a loss of accuracy and an increase in time spent on the task (resulting in an overall time that is almost twice of the overall time we observe when adopting the float32 format). Despite the loss of accuracy, this loss is visible just at the 2nd decimal digit (for $mAP@0.5:0.95$) and only at the 3rd decimal digit (for $mAP@0.5$) .

It's worth to notice that we do not have hardware support for the posit representation and that all the tests performed were carried out by simulating the posit environment and making different conversions between different types of representation.

What if we had hardware support for posit representations? Would the performance have changed getting better or worse than the one observed for float32 format? Unfortunately we cannot answer this question, that could be answered only by carrying out the object detection task on a computer that has hardware support for posits.

References

[al] J.Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: (). DOI: https://arxiv.org/abs/1506.02640.

[Dat] Coco Dataset. *Detection Evaluation*. URL: https://cocodataset.org/#detection-eval. (accessed: 12.07.2022).

[Hui] Jonathan Hui. *mAP (mean Average Precision) for Object Detection*. URL: https://jonathan-hui.medium.com/map-mean-average-precision-for-object-detection-45c121a31173.

[Joh] Isaac Yonemoto John L. Gustafson. “Beating Floating Point at its Own Game: Posit Arithmetic”. In: (). DOI: http://www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf.