POSIT-BASED DEEP NEURAL NETWORKS FOR AUTONOMOUS DRIVING

Project for Symbolic and Evolutionary AI course

PROJECT CODE AND NOTEBOOKS AVAILABLE AT:
https://github.com/mattemugno/seai_project

A.Y: 2022/23

ARTIFICIAL INTELLIGENCE AND DATA ENGINEERNG

Filippo Puccini, Matteo Mugnai

1 SOMMARIO

2 INTRODUCTION

In this project, we have conducted computer vision tasks and object detection ones using Google Colab and TensorFlow on automotive datasets. Our primary objective was to compare the performance during testing and evaluate the differences in accuracy between using floating-point numbers (float) and using posit numbers.

Computer vision and object detection techniques play a crucial role in various domains, including autonomous driving, surveillance systems, and robotics. By accurately identifying and localizing objects in images or video streams, these technologies enable advanced applications such as object tracking, collision avoidance, and scene understanding.

To achieve our goal, we leveraged the power of Google Colab, a cloud based Jupyter notebook environment, and TensorFlow, a popular deep learning framework widely used in the field of computer vision. By utilizing these tools, we were able to train deep learning models for object detection on our automotive dataset.

One critical aspect we focused on was the choice of number representation in the models. Traditionally, floating-point numbers (float) have been widely used in deep learning frameworks for their flexibility and precision. However, recent advancements in hardware design and number representation have introduced posit numbers as an alternative option. Posits offer improved efficiency and reduced memory requirements while maintaining competitive accuracy.

Through extensive experimentation and testing, we aimed to quantify the impact of using float and posit numbers on the performance of our models. By comparing the accuracy achieved in testing, we sought to gain insights into the trade-offs between the two number representations and determine which approach is more suitable for automotive computer vision tasks.

We hope that this project documentation provides a comprehensive understanding of our research process, experimental setup, and the results obtained. By analysing the performance and differences between float and posit numbers, we aim to contribute to the ongoing discussion and understanding of the advantages and limitations of posits as a potential replacement for floats.

Posit numbers are a new way to represent real numbers for computers, an alternative to the standard IEEE floating point formats. The primary advantage of posits is the ability to get more precision or dynamic range out of a given number of bits. If an application can switch from using 32-bit IEEE floats to using 16-bit posits, for example, it can fit twice as many numbers in memory at a time. That can make a big difference in the performance of applications that process large amounts of data. Note that the precision of a posit number depends on its size. This is the sense in which posits have tapered precision. Numbers near 1 have more precision, while extremely big numbers and extremely small numbers have less. This is often what you want. Typically, most numbers in a computation are roughly on the order of 1, while with the largest and smallest numbers, you mostly want them to not overflow or underflow. [1]

Despite all the expected benefits of posit numbers, the floating-point format has one important advantage over posit, which is the fixed bit format for the exponent and fraction. As a result, parallel decoding may be used for extracting the exponent and fraction of floating-point numbers. Whereas, decoding the posit fields may only be done serially due to using dynamic ranges. However, not having NaNs and denormals simplifies the circuit to some extent. Nevertheless, unlike the floating-point system, posit is a newly proposed number system yet to be investigated thoroughly. [2]

We run experiments considering only the testing phase and not the training one because we observed that training a neural network with posit numbers tends to suffer from increased computational complexity and slower convergence compared to using float numbers. So, all training procedures have been performed by using standard *'float_32'.*
The slower training speed with posit numbers can be attributed to several factors. First, the more compact representation of posit numbers often requires additional computational operations, such as conversion or scaling, to accommodate the range of values encountered during training. These extra operations introduce overhead and increase the overall computational load. Moreover, the mathematical operations involved in training a neural network, such as matrix multiplications, gradients calculations, and weight updates, often rely on highly optimized algorithms and libraries specifically designed for floating-point computations. These algorithms and libraries may not be fully compatible or optimized for posit numbers, leading to suboptimal performance and slower execution.
It's worth noting that the challenges in training with posit numbers are an active area of research, and ongoing efforts are being made to improve their efficiency and compatibility with neural network training algorithms. However, at the current stage, using float numbers remains the preferred choice for training neural networks due to their extensive support, optimized implementations, and compatibility with existing frameworks and hardware architectures.

3 DATASETS

In this project we worked with different kind of datasets related to automotive field. We started from a simple and well-known dataset like **GTSRB** one. [3]

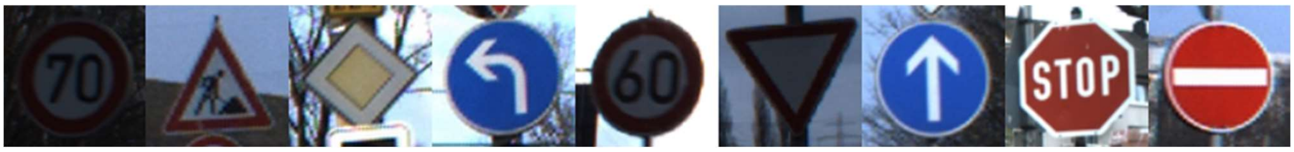


Figure 1 Samples from GTSRB dataset

The German Traffic Sign Recognition Benchmark is a multi-class, single-image classification challenge held at the International Joint Conference on Neural Networks (IJCNN) 2011. It contains more than 40,000 images in total splitted among 43 classes.
This dataset can be very useful for control systems of self-driving cars. In fact, it is paramount for such systems to be able to recognize traffic signals in order to take prompt actions that ensures the correct behaviour of a car on the road, avoiding accidents and dangerous moves.
Then we took in consideration another dataset: GTSDB [4]. It is similar to the previous one but now it contains common road traffic scenarios in which they can be present or not some traffic signals.

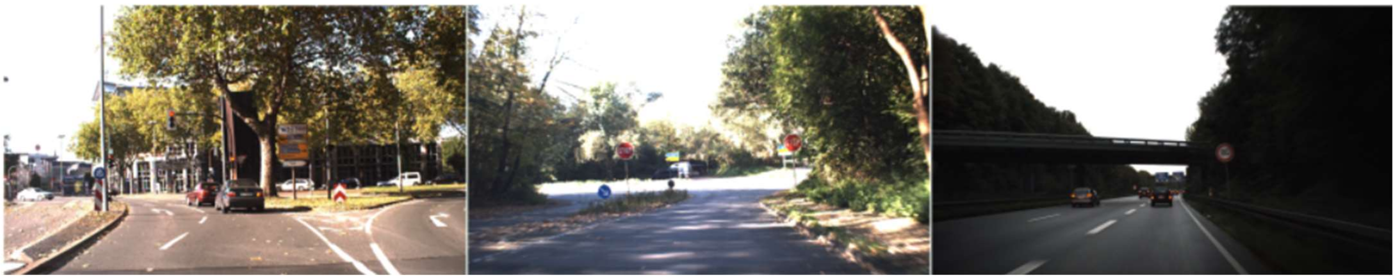


Figure 2 Samples from GTSDB dataset

The German Traffic Sign Detection Benchmark is a single-image detection assessment for researchers with interest in the field of computer vision, pattern recognition and image-based driver assistance. It contains 900 images (divided in 600 training images and 300 evaluation images) of various traffic scenarios that contains from 0 up to 6 signals.
As we can see they represent two different types of tasks: on one side we have classification task and on the other side we have an object detection one. These datasets can be a good starting point to investigate the behaviour of neural networks with posit number format and the differences with respect to float representation.
In GTSRB dataset we have a folder for each class with all related images of traffic signals in “. ppm” format.
In GTSDB dataset we have 900 images again in “. ppm” format and a gt.txt file containing a row for each traffic signals present inside a scenario with x1, y1, x2, y2 coordinates useful to draw bounding boxes and the corresponding class label of the signal itself. Let’s analyse some statistics of this dataset.

```
CLASS NAMES MAPPING ( {1: 'speed_limit_20', 2: 'speed_limit_30', 3: 'speed_limit_50',
4: 'speed_limit_60', 5: 'speed_limit_70', 6: 'speed_limit_80', 7: 'restriction_ends_80',
8: 'speed_limit_100', 9: 'speed_limit_120', 10: 'no_overtaking', 11: 'no_overtaking_trucks',
12: 'priority_at_next_intersection', 13: 'priority_road', 14: 'give_way', 15: 'stop',
16: 'no_traffic_both_ways', 17: 'no_trucks', 18: 'no_entry', 19: 'danger', 20: 'bend_left',
21: 'bend_right', 22: 'bend', 23: 'uneven_road', 24: 'slippery_road', 25: 'road_narrows',
26: 'construction', 27: 'traffic_signal', 28: 'pedestrian_crossing', 29: 'school_crossing',
30: 'cycles_crossing', 31: 'snow', 32: 'animals', 33: 'restriction_ends', 34: 'go_right', 35:
'go_left', 36: 'go_straight', 37: 'go_right_or_straight', 38: 'go_left_or_straight', 39: 'keep_right',
40: 'keep_left', 41: 'roundabout', 42: 'restriction_ends_overtaking',
43: 'restriction_ends_overtaking_trucks'}

```

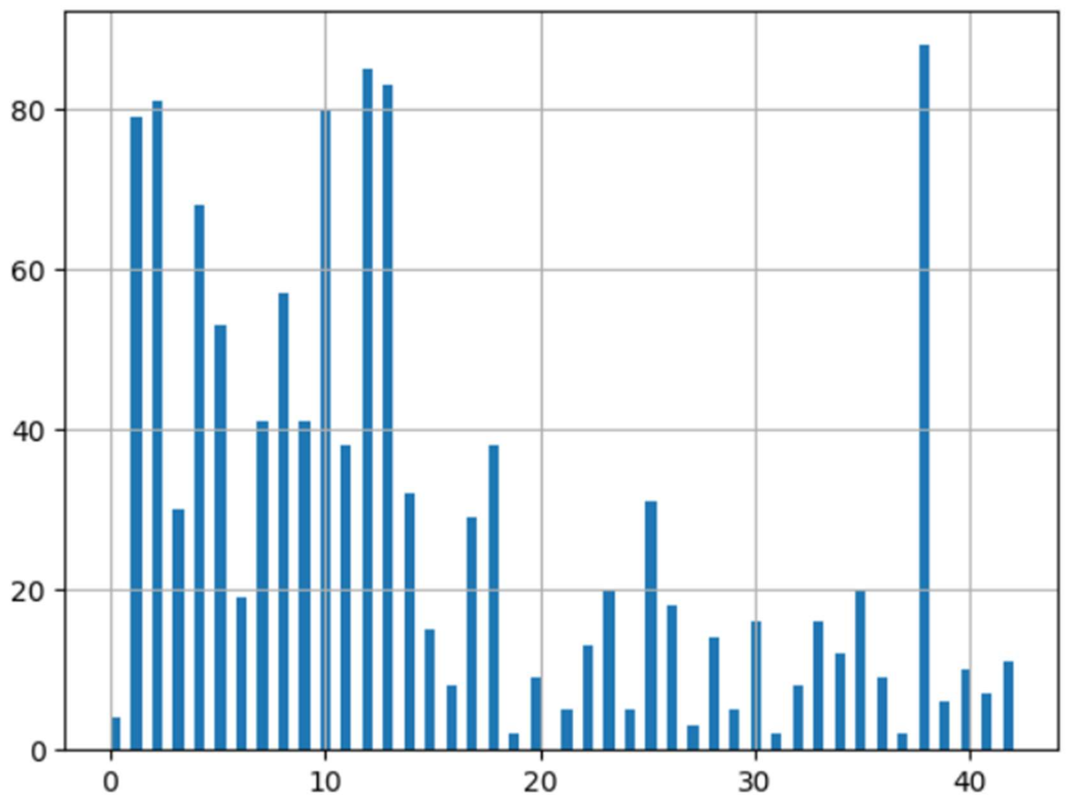
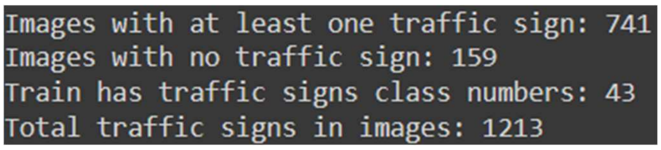


Figure 3 Class cardinality of GTSDB dataset



```
Images with at least one traffic sign: 741
Images with no traffic sign: 159
Train has traffic signs class numbers: 43
Total traffic signs in images: 1213
```

Figure 4 Dataset statistics

4 CLASSIFICATION TASK

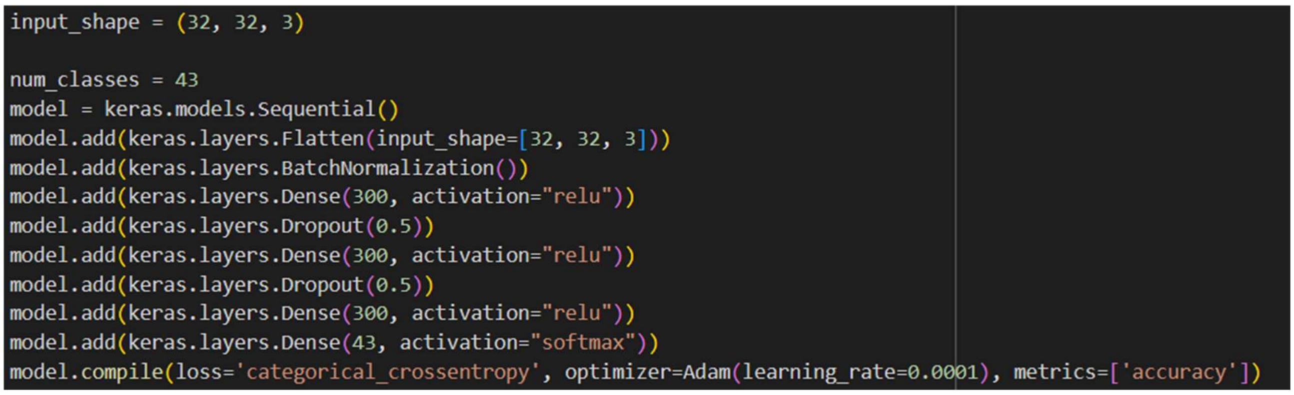
On this collection we performed a classification task of computer vision in which we basically recognize and classify properly a traffic signal represented in an image. For this purpose, we used some different CNNs: from scratch and pretrained.

4.1 CNN FROM SCRATCH

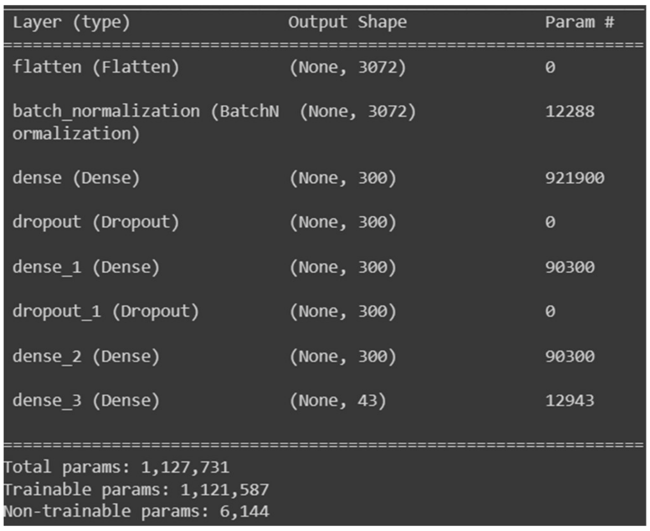
Both these models have been trained on Google Colab following an 80/10/10 strategy for what concerns training/validation/test sets. We run the training procedure with *model.fit()* function leaving the dataset unbalanced but assigning a weight to every class proportional to the number of samples.

4.1.1 Dense based model

We tried to run an experiment starting from a simple CNN from scratch. It has the following format:



```
input_shape = (32, 32, 3)

num_classes = 43
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[32, 32, 3]))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dropout(0.5))
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dropout(0.5))
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dense(43, activation="softmax"))
model.compile(loss='categorical_crossentropy', optimizer=Adam(learning_rate=0.0001), metrics=['accuracy'])
```



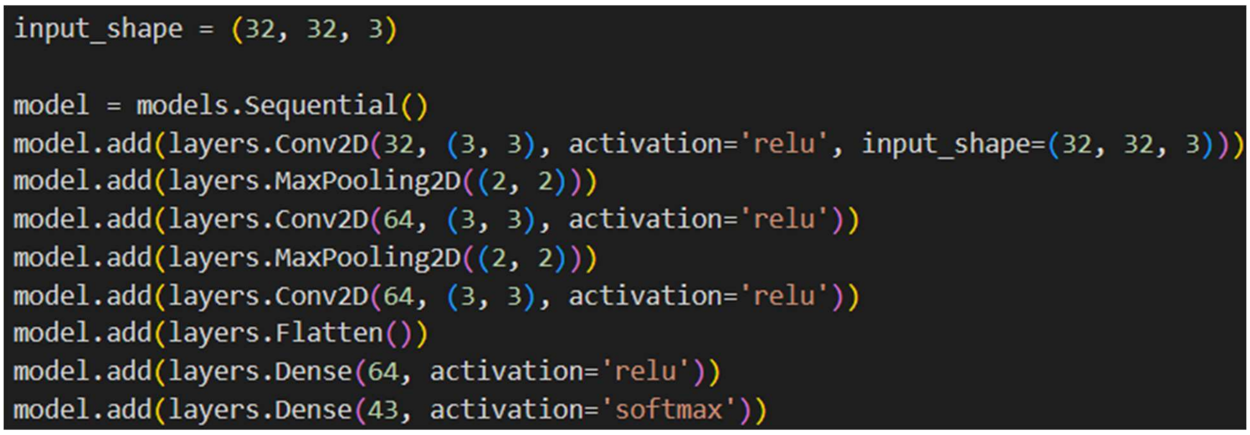
```
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 3072)              0

 batch_normalization (BatchN  (None, 3072)             12288
 ormalization)

 dense (Dense)               (None, 300)               921900

 dropout (Dropout)           (None, 300)               0

 dense_1 (Dense)             (None, 300)               90300

 dropout_1 (Dropout)         (None, 300)               0

 dense_2 (Dense)             (None, 300)               90300

 dense_3 (Dense)             (None, 43)                12943

=================================================================
Total params: 1,127,731
Trainable params: 1,121,587
Non-trainable params: 6,144
```

Figure 5 Model from scratch with only dense and dropout layers

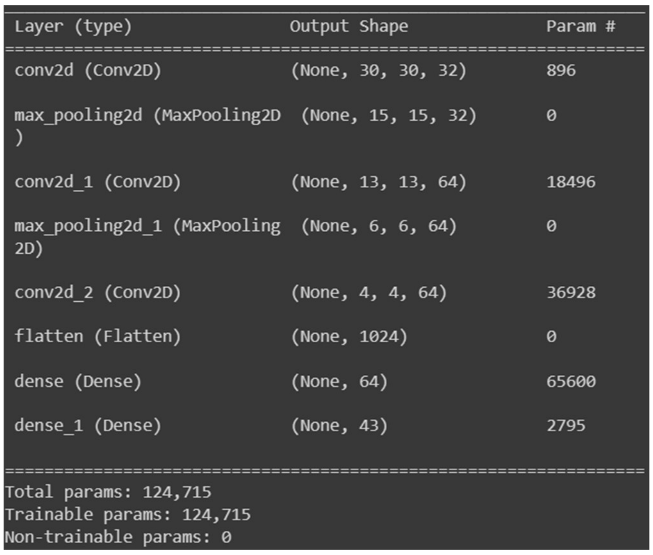
The given neural network is designed for image classification tasks. It takes input images of shape (32, 32, 3) representing RGB images with a width and height of 32 pixels and 3 colour channels. The goal is to classify the images into one of 43 possible classes.

The network architecture consists of a sequential model in Keras. The input layer flattens the 3D input images into a 1D vector. This is followed by a batch normalization layer, which helps in normalizing the input and improving the training process.
Next, there are three fully connected layers with 300 units each, using the ReLU activation function. These layers are intended to learn complex patterns and representations from the flattened input. Dropout layers with a rate of 0.5 are added after each fully connected layer to prevent overfitting by randomly deactivating some neurons during training.
Finally, the output layer consists of a dense layer with 43 units, using the softmax activation function. This layer produces the probability distribution over the 43 classes, allowing the network to predict the class of the input image.

4.1.2 Convolution based models



```
input_shape = (32, 32, 3)

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(43, activation='softmax'))
```



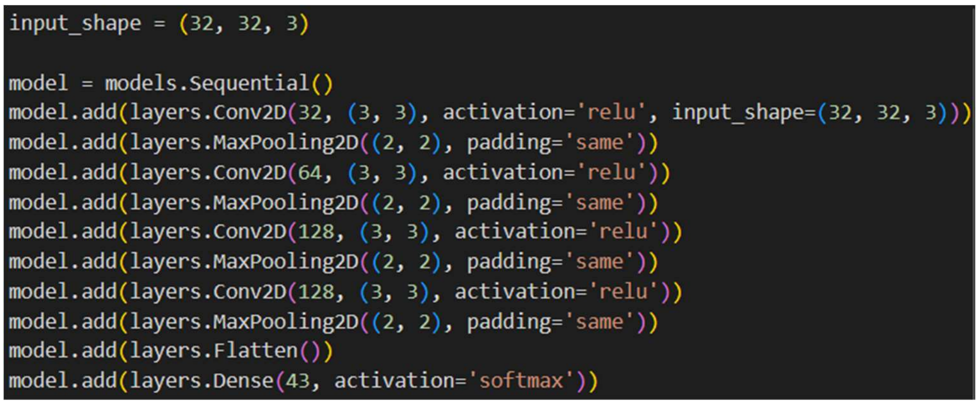
```
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 30, 30, 32)        896

 max_pooling2d (MaxPooling2D  (None, 15, 15, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 13, 13, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 6, 6, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 4, 4, 64)          36928

 flatten (Flatten)           (None, 1024)              0

 dense (Dense)               (None, 64)                65600

 dense_1 (Dense)             (None, 43)                2795

=================================================================
Total params: 124,715
Trainable params: 124,715
Non-trainable params: 0
```

Figure 6 Model from scratch with convolutional layers

The network starts with three convolutional layers. These layers are designed to detect various patterns and features within the input images. Each convolutional layer uses a set of learnable filters to perform feature extraction. The filters' sizes are 3x3, which allows the network to capture local information effectively.
After each convolutional layer, there is a max-pooling layer. Max-pooling reduces the spatial dimensions of the data while retaining essential features. It works by selecting the maximum value

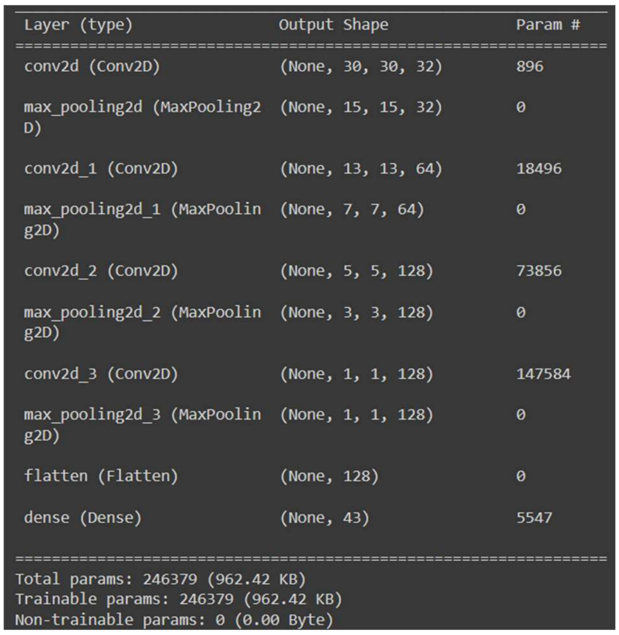
within a small window and moving it through the input, which helps in spatial reduction and abstraction.
Following the convolutional and max-pooling layers, the data is flattened into a one-dimensional vector. This step prepares the features for further processing in the fully connected layers.
There are two fully connected layers. The first fully connected layer consists of 64 neurons and uses the ReLU activation function. It serves as a feature extractor, learning complex patterns from the flattened data. The second fully connected layer is the output layer. It contains 43 neurons, each corresponding to a different class. The softmax activation function is applied here to convert the network's raw output into class probabilities. This layer assigns a probability to each class, and the class with the highest probability is the predicted class.

Then we built also another convolutional model similar to this presented above, but deeper (with more repeated convolutional layers and more parameters) to see whether there is a larger difference in metrics. The model crafted from scratch is this one:



```python
input_shape = (32, 32, 3)

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2), padding='same'))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2), padding='same'))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2), padding='same'))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2), padding='same'))
model.add(layers.Flatten())
model.add(layers.Dense(43, activation='softmax'))
```

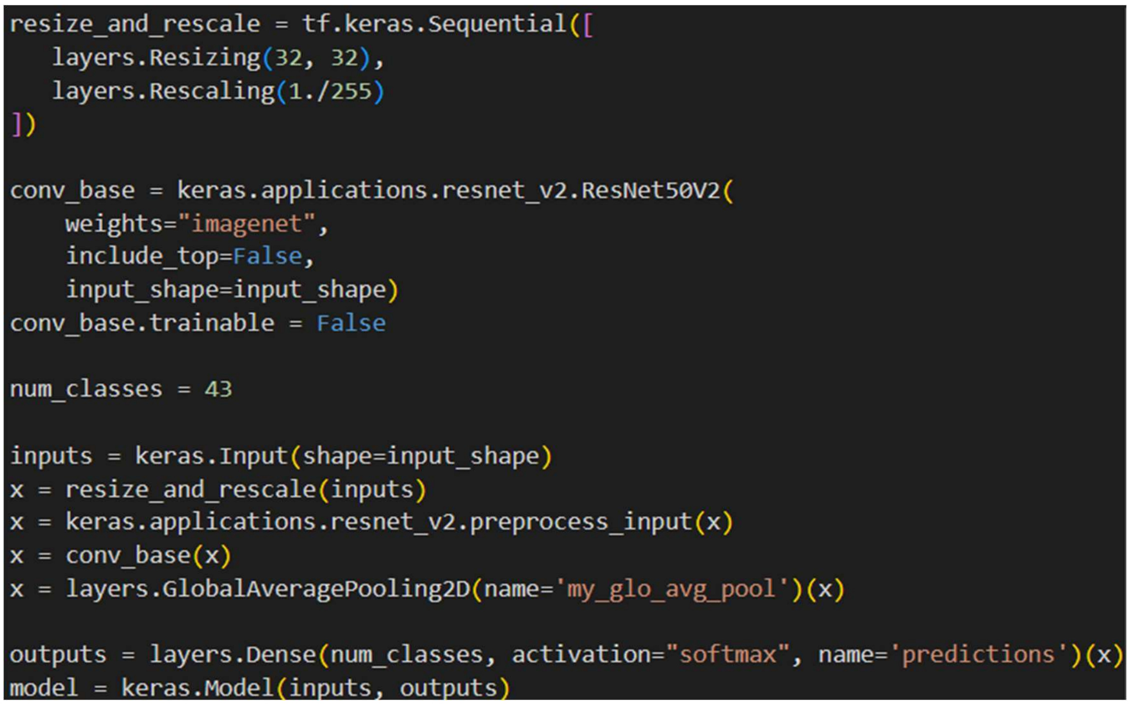
Figure 7 Extended from scratch convolution model



```
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 30, 30, 32)        896

 max_pooling2d (MaxPooling2  (None, 15, 15, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 13, 13, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 7, 7, 64)          0
 g2D)

 conv2d_2 (Conv2D)           (None, 5, 5, 128)         73856

 max_pooling2d_2 (MaxPoolin  (None, 3, 3, 128)         0
 g2D)

 conv2d_3 (Conv2D)           (None, 1, 1, 128)         147584

 max_pooling2d_3 (MaxPoolin  (None, 1, 1, 128)         0
 g2D)

 flatten (Flatten)           (None, 128)               0

 dense (Dense)               (None, 43)                5547

=================================================================
Total params: 246379 (962.42 KB)
Trainable params: 246379 (962.42 KB)
Non-trainable params: 0 (0.00 Byte)
```

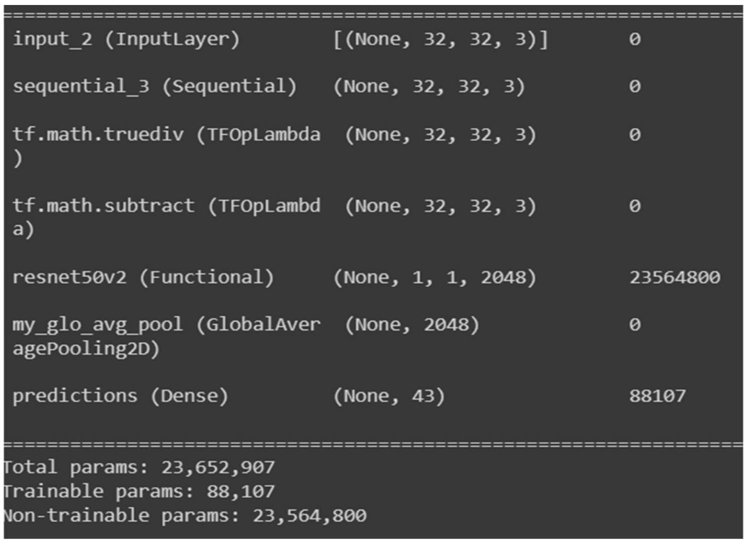
4.2 PRETRAINED CNN

Then we tried another approach by using the transfer learning technique on a pretrained CNN. For the first experiment we picked ResNet50v2 deep neural network.



```
resize_and_rescale = tf.keras.Sequential([
   layers.Resizing(32, 32),
   layers.Rescaling(1./255)
])

conv_base = keras.applications.resnet_v2.ResNet50V2(
    weights="imagenet",
    include_top=False,
    input_shape=input_shape)
conv_base.trainable = False

num_classes = 43

inputs = keras.Input(shape=input_shape)
x = resize_and_rescale(inputs)
x = keras.applications.resnet_v2.preprocess_input(x)
x = conv_base(x)
x = layers.GlobalAveragePooling2D(name='my_glo_avg_pool')(x)

outputs = layers.Dense(num_classes, activation="softmax", name='predictions')(x)
model = keras.Model(inputs, outputs)
```

Figure 8 ResNet architecture



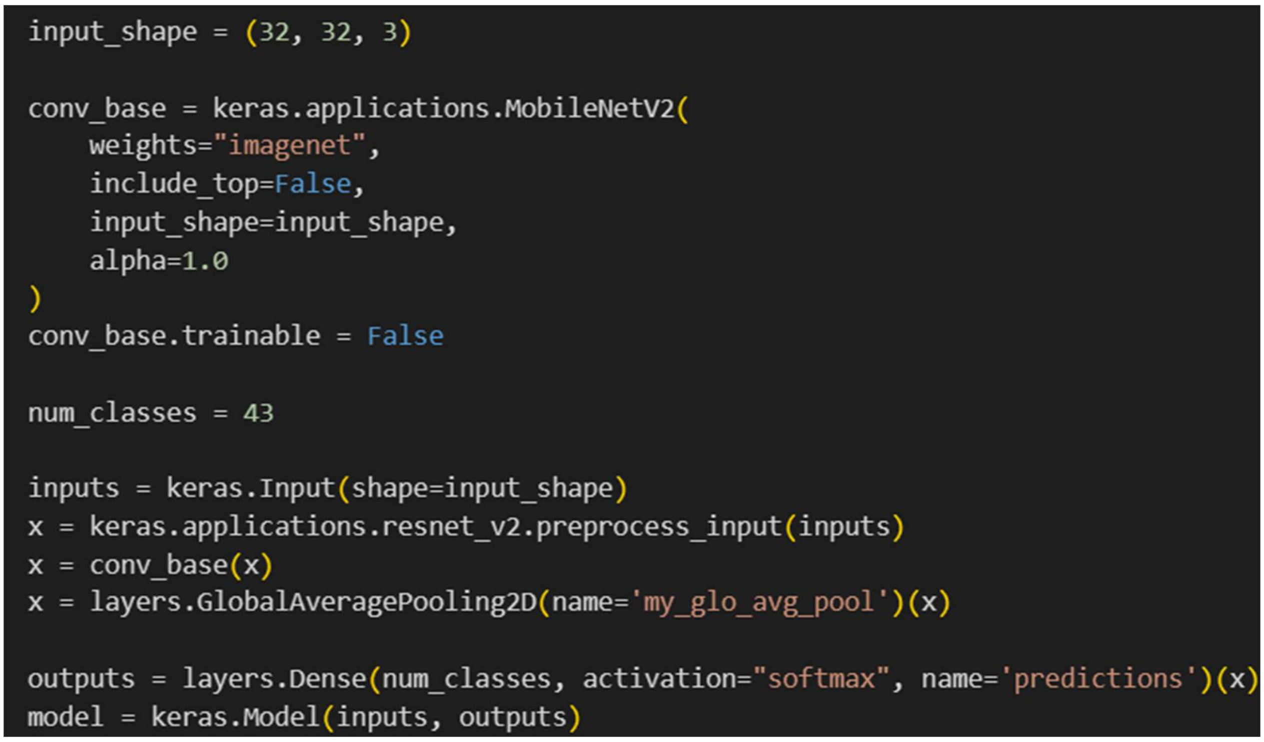
```
=================================================================
 input_2 (InputLayer)        [(None, 32, 32, 3)]       0

 sequential_3 (Sequential)   (None, 32, 32, 3)         0

 tf.math.truediv (TFOpLambda  (None, 32, 32, 3)        0
 )

 tf.math.subtract (TFOpLambd  (None, 32, 32, 3)        0
 a)

 resnet50v2 (Functional)     (None, 1, 1, 2048)        23564800

 my_glo_avg_pool (GlobalAver  (None, 2048)             0
 agePooling2D)

 predictions (Dense)         (None, 43)                88107

=================================================================
Total params: 23,652,907
Trainable params: 88,107
Non-trainable params: 23,564,800
```

Figure 9 Pretrained model using resnet50

The provided neural network architecture utilizes a pre-trained ResNet50V2 model for image classification. The network's purpose is to classify images into one of 43 possible classes.
The ResNet50V2 model is initialized with pre-trained weights from the ImageNet dataset. By freezing the weights of the convolutional base, the model retains the knowledge learned from the large dataset while allowing the addition of new layers for specific classification tasks.

The input layer is defined using the specified input shape. The input images are then processed by resizing and rescaling operations, followed by preprocessing according to the ResNet50V2 model's requirements.
The processed images are passed through the frozen convolutional base, extracting relevant features from the images. A global average pooling layer is applied to reduce the spatial dimensions of the extracted features, producing a fixed-length feature vector.
Finally, the feature vector is fed into a dense layer with the softmax activation function, generating the output probabilities for the 43 classes.
This neural network configuration enables leveraging the ResNet50V2 model's pre-trained weights and feature extraction capabilities to perform accurate image classification across 43 classes while allowing for efficient training and inference.
We tried to perform some kind of fine-tuning, also adding some additional layer (dense and dropout) without improving results, so we kept the structure unmuted.
The network without fine-tuning performed poorly, so we tried step by step to unfreeze and retrain some blocks of the model skeleton starting by the last ones. At the end we reached acceptable results, in particular with the fine tuning of one and two blocks.

In a second experiment we used *MobileNet* model to make a comparison of performances with the previous network. The architecture is the following one:



```
input_shape = (32, 32, 3)

conv_base = keras.applications.MobileNetV2(
    weights="imagenet",
    include_top=False,
    input_shape=input_shape,
    alpha=1.0
)
conv_base.trainable = False

num_classes = 43

inputs = keras.Input(shape=input_shape)
x = keras.applications.resnet_v2.preprocess_input(inputs)
x = conv_base(x)
x = layers.GlobalAveragePooling2D(name='my_glo_avg_pool')(x)

outputs = layers.Dense(num_classes, activation="softmax", name='predictions')(x)
model = keras.Model(inputs, outputs)
```

Figure 10 Mobilenet architecture

This model is much lighter if compared to ResNet, so we expect a small execution time, but we hope in similar performances.

4.3 Results and Conclusions

In the following table we can summarize all results that we obtained by running an evaluation script on Linux with Python on each model separately, through an environment in which has been installed the right wheel of TensorFlow needed to work with posits.

CNN MODEL	MODEL PARAMETERS	NUMERIC FORMAT	ACCURACY	% ACC. VARIATION	EX. TIME
FROM SCRATCH DENSE	1127131	FLOAT 16	0.9870	±0	26s
		FLOAT 32	0.9870	±0	1s
		FLOAT 64	0.9870	±0	1s
		POSIT<16,0>	0.9772	- 0.1%	108s
FROM SCRATCH CONV	124715	FLOAT 16	0.8743	- 0.06%	9s
		FLOAT 32	0.8748	±0	1s
		FLOAT 64	0.8748	±0	2s
		POSIT<16,0>	0.8528	- 2.5%	561s
FROM SCRATCH DEEPER	246379	FLOAT 16	0.9944	±0	11s
		FLOAT 32	0.9944	±0	1s
		FLOAT 64	0.9944	±0	3s
		POSIT<16,0>	0.9941	- 0.03%	592s
RESNET FINE TUNED 1 BLOCK	23652907 (4546603)	FLOAT 16	0.9278	+ 0.05%	1058s
		FLOAT 32	0.9273	±0	11s
		FLOAT 64	0.9273	±0	14s
		POSIT<16,0>	0.8980	- 3.16%	3364s
RESNET FINE TUNED 2 BLOCKS	23652907 (9005099)	FLOAT 16	0.9197	+ 0.09%	1058s
		FLOAT 32	0.9189	±0	8s
		FLOAT 64	0.9189	±0	16s
		POSIT<16,0>	0.9026	- 1.77%	3374s
MOBILENET FINE TUNED 1 BLOCK	2313067 (1848384)	FLOAT 16	0.9153	- 0.28%	209s
		FLOAT 32	0.9179	±0	4s
		FLOAT 64	0.9179	±0	9s
		POSIT<16,0>	0.8335	- 9.19%	726s

After completing all scheduled tests, in terms of accuracy we can conclude that CNN from scratch got better results than pretrained one. Moreover, we can notice that CNN from scratch without convolution layers outperforms all the others.

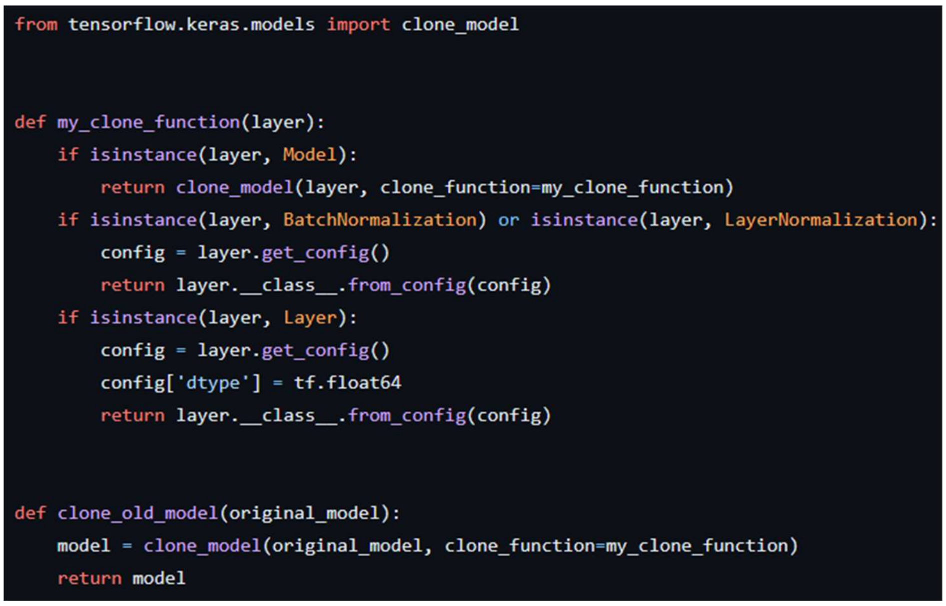
There can be several reasons why classification on the GTSRB (German Traffic Sign Recognition Benchmark) dataset might perform better in terms of test accuracy with a CNN (Convolutional Neural Network) trained from scratch compared to using a pretrained one:

- **Dataset/domain dissimilarity**: Pretrained models are typically trained on large-scale datasets such as ImageNet (in our case), which contains a wide variety of objects from different domains. However, the GTSRB dataset focuses specifically on traffic sign recognition. As a result, the pretrained model may not have learned features or patterns that are particularly relevant to traffic signs. Training a CNN from scratch on the GTSRB dataset allows the model to learn domain-specific features and patterns, potentially leading to better performance.

- **Complexity of the pretrained model**: Pretrained models are often quite deep and complex, with millions of parameters. Such models have a large capacity to capture a wide range of features and patterns. However, if the GTSRB dataset is relatively small, training a complex pretrained model can lead to overparameterization, making it difficult for the model to generalize effectively. In contrast, a CNN trained from scratch can be tailored to the dataset's size and complexity, potentially leading to better performance.
- **Transfer learning limitations**: While transfer learning using pretrained models can be highly effective in many scenarios, it is not a universal solution. The features learned by a pretrained model may not always be directly transferable to a new dataset, especially if the dataset has distinct characteristics or unique classes. In such cases, training a CNN from scratch can allow the model to learn specific representations and optimize its performance for the target dataset.

4.3.1 Numeric format influence

We run all test experiments on Linux with this TensorFlow wheel: *tensorflow-2.6.0-cp38-cp38-linux_x86_64.whl.* In this way we were able to convert all tensors of test images from *float_32* data type to the target one (***posit <16,0>***, *float_16* and *float_64*). We did the same thing with the model's weights that originally were in *float_32* format.
For the tensors we converted them by using this python instruction:
img = tf.convert_to_tensor(img, dtype='target_data_type').
To convert weights the only way was to clone the original model by building a new one layer by layer specifying the target data type. We used the following function:



```
from tensorflow.keras.models import clone_model


def my_clone_function(layer):
    if isinstance(layer, Model):
        return clone_model(layer, clone_function=my_clone_function)
    if isinstance(layer, BatchNormalization) or isinstance(layer, LayerNormalization):
        config = layer.get_config()
        return layer.__class__.from_config(config)
    if isinstance(layer, Layer):
        config = layer.get_config()
        config['dtype'] = tf.float64
        return layer.__class__.from_config(config)


def clone_old_model(original_model):
    model = clone_model(original_model, clone_function=my_clone_function)
    return model
```

Figure 11 Data type conversion function for layers

The *my_clone_function* is a custom function used with TensorFlow/Keras' *clone_model* method. It serves the purpose of cloning a model while customizing the cloning process for specific types of layers. Here's what it does:

1. If the layer is a model, it ensures that the internal model is correctly cloned.
2. If the layer is a *BatchNormalization* or *LayerNormalization*, it copies the layer configuration to create an identical new layer.
3. If the layer is a generic Layer, it modifies the layer's data type to the target one and duplicates it.

In summary, *my_clone_function* ensures that when a model is cloned, sets the data type of generic layers to target dtype in the cloned model, offering customization options for the cloning process. *BatchNormalization* layer behaves differently from the others, and it is impossible to convert it in the desired format. We ‘ll see again this point later in the documentation.

After checking that both the images and the weights were encoded properly, we run the inference on a test set of 3921 samples (10% of dataset, generated with a fixed seed).

From all tests stored in the table, independently from the model, we can notice that there is an always a small difference between posits and floats. This is the result that we expected and we hoped to reach, since we are evaluating posits as a viable alternative for floats. Since the network has been trained with *float_32*, during evaluation with *posit<16,0>,* in the best scenario we can reach at maximum the same accuracy than *float_32*.
Posit<16,0> is a numeric format with lower precision compared to *float_32*. When converting the model's weights and test data to this format, it is possible that some information is lost due to the limited numerical precision. This loss of precision could impact the model's accuracy during testing.
Almost in every experiment the difference is small, but in some cases is remarkable (second, fourth and last case).
The reason for similar results could stand in the fact that GTSRB is a simple dataset with images that are not “difficult” in terms of shapes and colours. So, decoding them with posits or floats might make minor difference, even if *posit<16,0>* has lower precision than *float_32*.

We can notice when the model’s weights are converted in another numeric format simply looking at the execution time: in all experiments we can see the significant difference in the time needed for the whole procedure of evaluation. In fact, when we deal with posits, we can see a huge increase in time spent. This is mainly since this numeric format lacks a special hardware support necessary to speed up the operations. In particular, when there are computationally heavy operations like convolution, even if the model is not too big and deep (like in the second or third experiment), it results in a remarkably high delay with respect to the case of *float_32.* Even in the case of *float_16* the execution time grows since also this format lack of GPU support, and it runs only on CPU.

We tried also to emphasize the difference between float and posit accuracy obtained in the experiment of the model from scratch with convolution, without having success. In the third test we added some other layers to deepen the model and to increase number of parameters and so also the dimension of the related ".h5" file. Unfortunately, we obtained only a very small worsening in the accuracy.

For what concerns float formats, with *float_64* we didn't obtain differences in terms of accuracy than *float_32*. This make sense since with this format number are represented with twice digits than the standard. In this way numbers don't need to be approximated or truncated and this results in same performance. The only difference stands in execution time.
With *float_16* we always get a different behaviour: sometimes accuracy improves, sometimes is the same and sometimes worsens.
We can't explain completely this result but what is sure is that execution times always increase. This is because *float_16* lacks GPU support, and it is directly executed on CPU.

Talking about pretrained network we can notice that *MobileNet* performed very similar to *ResNet* even if the number of parameters is much lower. The execution time is much lower, so with the proper hardware acceleration could be a valid model to be used with posits in alternative to *float_32*. Unfortunately, the accuracy drops more than *ResNet* losing a 9.2% in precision.

For this use case we can conclude that using *posit<16,0>* can be beneficial in testing phase (if we have at disposal the right hardware, like RISC-V architecture, for this format number to speed up operations thanks to acceleration) and it can be a good alternative that must be taken in consideration for several reasons (explained below) when we have to cope with real-time computing and embedding systems like in the automotive field.
First, *posit<16,0>* reduces memory requirements, making it an attractive choice for storage and data transmission. Its compact representation consumes fewer bits, addressing the ever-growing demand for efficient memory usage in modern computing systems.
Moreover, it promotes computational efficiency. Its reduced precision enables faster arithmetic operations, optimizing the speed of calculations. This advantage is particularly significant in applications requiring real-time processing, such as embedded systems like in our use case.
In addition, *posit<16,0>* aligns with energy-efficient computing. By utilizing fewer resources, it minimizes power consumption, a critical consideration in portable devices and green computing initiatives.
Furthermore, this numeric format enhances parallelism in certain hardware architectures, unlocking the potential for concurrent processing and accelerating complex tasks.

5 OBJECT DETECTION TASK

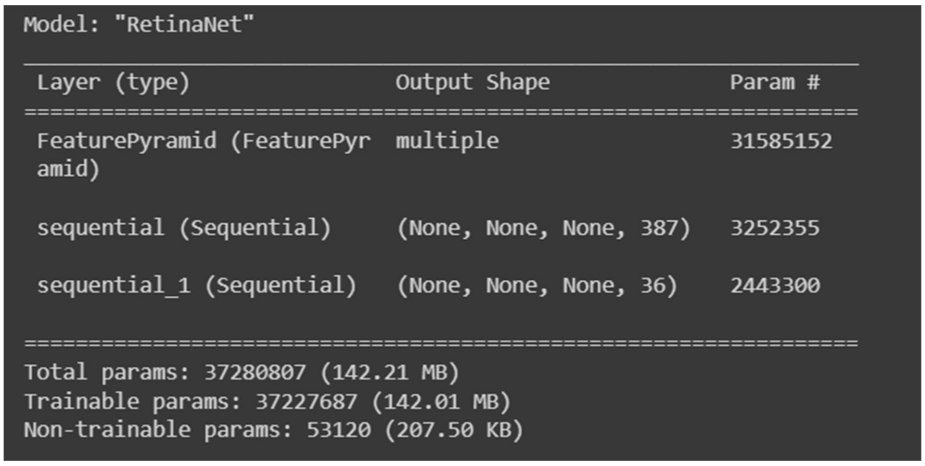
As already mentioned on this collection we performed an object detection task by using again TensorFlow.
At the beginning we tried various approach from YOLO models to R-CNN ones. We have to abandon these ideas because **YOLO** is not TensorFlow based and it's not trivial to convert the trained model in a format suitable to TensorFlow. **MASK R-CNN**, instead, is too heavy and it needs a lot of resources especially if it has to deal with images with shape 1360x800 like in the case of our dataset. So, it was impossible for our computers to complete the training procedure.
Then we opted to use a pretrained model from *TensorFlow model Zoo* [5].
We select two model from this library: **SSD + MobileNet v2 320x320** and **EfficientDet d0 512x512**.
We made this choice because we learned that SSD models are light and suitable to be used in environment like Google Colab.
In this case we were able to complete the full training procedure. At the end we got a file with the weights of the model, but it was in saved model format (". pb"). So, we tried to convert it in ".h5" (through ONNX converter and other methods) without having success because this model is not compatible with Keras ".h5" format.
Finally, we decided to use directly a model based on Keras. We complete a full training procedure on Google Colab by using a **RetinaNet** model through *Keras CV*, we were able to calculate some metrics like *IoU* and at the end we got also a ".h5" file containing the weights of the model. At this point we started performing an evaluation on Linux by using the proper TensorFlow wheel of posits. We were unable to complete the procedure due to an incompatibility between TensorFlow and Python versions used for training on Colab and the wheel's versions of these libraries.
As a last resort, we tried to train an R-CNN model directly on a remote machine of the university, but we encountered difficulties even in this case because the wheel of TensorFlow requires a version of *GLIBC* (an OS library) that was not supported on the current OS installed on that host.

5.1 RETINANET

Following a thorough assessment of the challenges mentioned earlier, we made the decision to exploit the *RetinaNet* model [7], constructing it from the ground up. This implementation of the *RetinaNet* network is developed in Keras and is composed by the following components:

- **Backbone**: The backbone of RetinaNet is typically a pre-trained convolutional neural network (CNN) architecture, such as ResNet or VGG (in our case ResNet50v2).
- **Feature Pyramid Network (FPN)**: RetinaNet incorporates a Feature Pyramid Network (FPN) to create a feature pyramid. This pyramid contains multi-scale features, enabling the model to handle objects of various sizes effectively. FPN enhances object detection across different scales.
- **Detection Head**: The detection head in RetinaNet consists of two parallel branches. One branch is responsible for object classification, determining the class of the detected object. The other branch handles object localization and regression, estimating the precise bounding box around the object. This part comprises multiple layers of neural networks, including a classification head and a regression head.

We proceeded to train the *RetinaNet* model using the German Traffic Sign Detection Benchmark (GTSDB) dataset, with the goal of establishing an object detection system. The GTSDB dataset serves as a dedicated benchmark for single-image detection, designed to cater to researchers interested in computer vision, pattern recognition, and image-based driver assistance. It comprises 900 images, distributed into 600 training images and 300 evaluation images.
We trained the model in float32 using Google Colab, saving only the weights at the end of the training. To perform inference on the model, we utilized the 'clone model' method, as described in Section 4.3.1, by converting the model to float16 and posit16. In Figures 12 and 13, you can see the models in float32 and float16, respectively. The difference in terms of storage space is significant, as we occupy only 80 MB in float16 compared to 140 MB in float32.

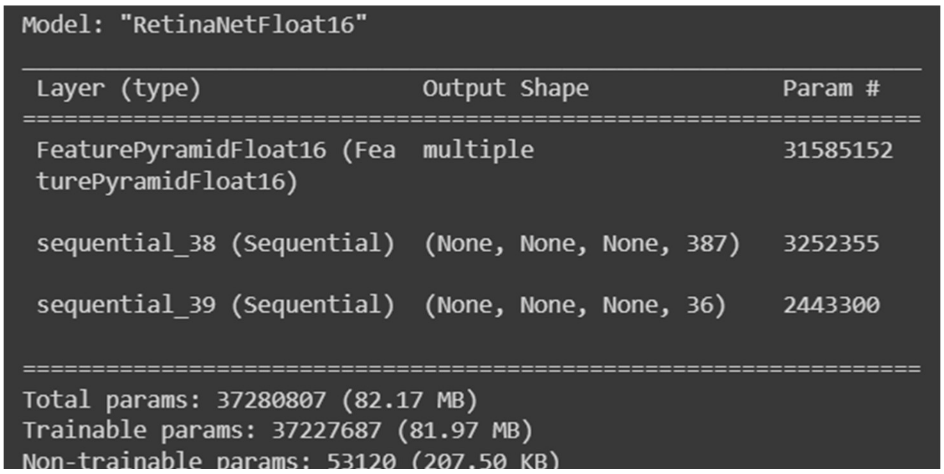


```
Model: "RetinaNet"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 FeaturePyramid (FeaturePyr  multiple                  31585152
 amid)

 sequential (Sequential)     (None, None, None, 387)   3252355

 sequential_1 (Sequential)   (None, None, None, 36)    2443300

=================================================================
Total params: 37280807 (142.21 MB)
Trainable params: 37227687 (142.01 MB)
Non-trainable params: 53120 (207.50 KB)
```

Figure 12 RetinaNet float32 model

5.1.1 Test and Results

At first, we were unable to load the model weights using a different numeric format. It appears that the instruction *K.set_floatx('posit160')* does not impact the representation of the model's weights. To solve this problem and load the model weights in a different numeric format, as already said above, such as float16 and posit160 in our case, we utilize the Keras method *'tf.keras.models.clone_model'* with a custom cloning function. This was necessary because we need to bypass the BatchNormalization layers due to an error that will be discussed later in the



```
Model: "RetinaNetFloat16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 FeaturePyramidFloat16 (Fea  multiple                  31585152
 turePyramidFloat16)

 sequential_38 (Sequential)  (None, None, None, 387)   3252355

 sequential_39 (Sequential)  (None, None, None, 36)    2443300

=================================================================
Total params: 37280807 (82.17 MB)
Trainable params: 37227687 (81.97 MB)
Non-trainable params: 53120 (207.50 KB)
```

Figure 13 RetinaNet float16 model

documentation. In this case, we chose to convert only the Feature Pyramid (and consequently the backbone it includes) to a different numerical format, while leaving the weights of the Detection Head in float32. This decision was made due to the limited computational power of the tools we used for testing, which would have significantly slowed down the calculation of the Intersection over Union (IOU), especially in the posit case, as they lacked native hardware acceleration. Additionally, we resized the images to further expedite the process.

To evaluate our model, we used the **Intersection Over Union (IOU)** as our metric. This metric calculates the ratio between the area of the predicted bounding box and the area of the ground truth bounding box. By averaging this value across all the images in the test set, we can gauge how accurate our network is in detecting subjects within the images. It should be noted that the model was trained on a limited number of images, and some of them do not contain any signs. We considered outputs with a confidence score greater than 0.3. A higher threshold resulted in too few predicted bounding boxes. We conducted tests using *float32*, *float16*, and *posit<16,0>* data types.

CNN MODEL	**NUMERIC FORMAT**	**AVERAGE IOU**	**TIME**
RETINANET	FLOAT 16	0.786	37 s/step
	FLOAT 32	0.78603023	1 s/step
	POSIT<16,0>	0.727051	199 s/step

The tabulated results underscore that employing various numeric formats in an object detection task yields remarkably similar performance outcomes. Notably, both FLOAT 16 and FLOAT 32 yield nearly identical average IOU scores, hovering around 0.786. This observation highlights the model's consistency in object detection capabilities, whether opting for the reduced precision of FLOAT 16 or the standard FLOAT 32. This flexibility can be particularly advantageous in scenarios where computational resources or memory limitations dictate the need for lower precision data types, such as FLOAT 16, without significantly compromising accuracy. However, it should be noted that for each image, prediction in the float16 case takes 37 seconds, while in the float32 case, it only takes one second. The significant reduction in model size in float16 results in much slower inference compared to the float32 model, which can leverage hardware acceleration.

The utilization of *Posit(16,0)* introduces a discrete reduction in the average IOU value, yet it consistently maintains a commendable level of performance. The time required for the execution of a single step, on the other hand, is much higher compared to the other two models. In this case, with only the Feature Pyramid in posit and the rest in float32, we reach 199 s/step, but in the case of the fully posit model, the time per step reaches 600 s/step. The lack of hardware support makes working with posits challenging, even though the performance is only slightly inferior to the other two models.

So, in terms of accuracy, posits can be used for more complex tasks like object detection, provided that there is hardware acceleration to speed up the required mathematical operations.

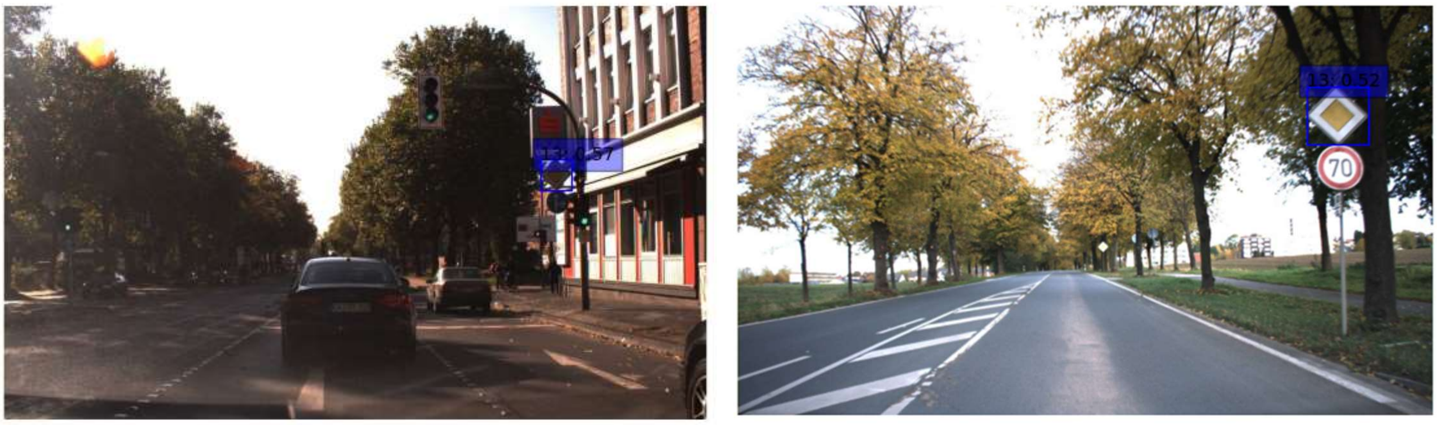


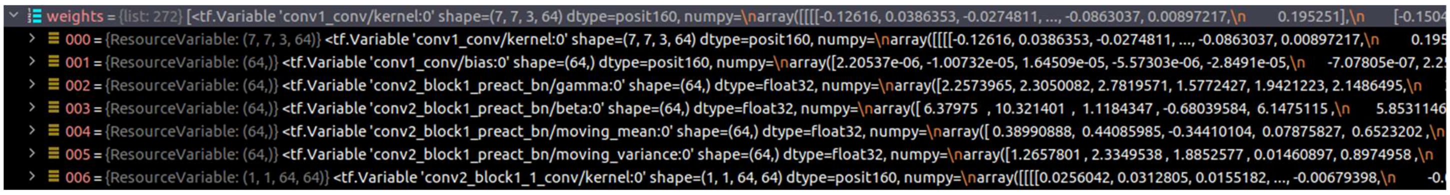
Figure 14 Detection with RetinaNet model on test set samples

In conclusion, posits emerge as a viable alternative to floats, within the limitations seen above about the execution time, within the context of object detection. In classification tasks, the foremost goal is the accurate assignment of class labels to input instances, necessitating precision due to the high stakes involved, such as in medical or security applications. Conversely, in object detection tasks, the emphasis shifts toward object localization within images, often through bounding boxes. While precision remains important, a slightly lower degree of precision compared to classification can be tolerated, especially when prioritizing rapid object detection over precise classification of specific object classes. Moreover, tasks like real-time object tracking can reap the benefits of improved computational efficiency.
Hence, in object detection tasks where precision may be marginally less critical than in classification but still demands reliability, the adoption of numeric formats like Posits can present a reasonable choice to reduce computational complexity and enhance efficiency. Nonetheless, the decision will always hinge on the specific application's requirements and the acceptable trade-offs between precision and efficiency.

5.1.2 Limitations and issues

As mentioned earlier, the cloning_model method we used to clone the model and assign weights with a different numerical format ignored batch normalization layers because the operations within it accept only float32, float16, or bfloat16 types (which are, nonetheless, cast to float32). In the case of the model in posit format, this meant that when a batch normalization layer was encountered, the input in posit was converted to float32, the operation was executed, and then it was converted back to posit, ready to be passed as input to the next layer. The batch normalization layer has four parameters (gamma, beta, moving mean, and moving variance) that remain as float32. This results in a decrease in precision due to numerous conversions and an overall performance decrease of the model. The situation that arises can be observed in Figure 15:

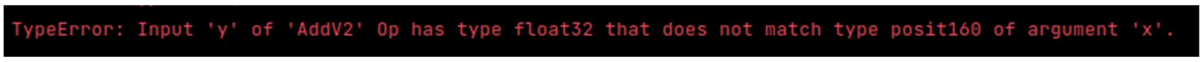
the four parameters of the batch normalization layers remain as float32, while all other layers are converted to posits.



```
weights = {list: 272} [<tf.Variable 'conv1_conv/kernel:0' shape=(7, 7, 3, 64) dtype=posit160, numpy=\narray([[[[-0.12616, 0.0386353, -0.0274811, ..., -0.0863037, 0.00897217,\n       0.195251],\n      [-0.1504
  000 = {ResourceVariable: (7, 7, 3, 64)} <tf.Variable 'conv1_conv/kernel:0' shape=(7, 7, 3, 64) dtype=posit160, numpy=\narray([[[[-0.12616, 0.0386353, -0.0274811, ..., -0.0863037, 0.00897217,\n       0.195
  001 = {ResourceVariable: (64,)} <tf.Variable 'conv1_conv/bias:0' shape=(64,) dtype=posit160, numpy=\narray([2.20537e-06, -1.00732e-05, 1.64509e-05, -5.57303e-06, -2.8491e-05,\n    -7.07805e-07, 2.2
  002 = {ResourceVariable: (64,)} <tf.Variable 'conv2_block1_preact_bn/gamma:0' shape=(64,) dtype=float32, numpy=\narray([2.2573965, 2.3050082, 2.7819571, 1.5772427, 1.9421223, 2.1486495,\n
  003 = {ResourceVariable: (64,)} <tf.Variable 'conv2_block1_preact_bn/beta:0' shape=(64,) dtype=float32, numpy=\narray([ 6.37975   , 10.321401  ,  1.1184347 , -0.68039584,  6.1475115 ,\n      5.8531146
  004 = {ResourceVariable: (64,)} <tf.Variable 'conv2_block1_preact_bn/moving_mean:0' shape=(64,) dtype=float32, numpy=\narray([ 0.38990888,  0.44085985, -0.34410104,  0.07875827,  0.6523202 ,\n
  005 = {ResourceVariable: (64,)} <tf.Variable 'conv2_block1_preact_bn/moving_variance:0' shape=(64,) dtype=float32, numpy=\narray([1.2657801 , 2.3349538 , 1.8852577 , 0.01460897, 0.8974958 ,\n
  006 = {ResourceVariable: (1, 1, 64, 64)} <tf.Variable 'conv2_block1_1_conv/kernel:0' shape=(1, 1, 64, 64) dtype=posit160, numpy=\narray([[[[0.0256042, 0.0312805, 0.0155182, ..., -0.00679398,\n      -0.
```

Figure 15 List of the first weights of the RetinaNet model

If we attempt to convert the model entirely to a numerical format not supported by the batch normalization layer, such as posit, the exception shown in Figure 16 is raised.



```
TypeError: Input 'y' of 'AddV2' Op has type float32 that does not match type posit160 of argument 'x'.
```

Figure 16 Exception raised from batch normalization layer

One potential improvement that could be applied in future projects is the implementation of a custom batch normalization layer, enabling full support for posit format (or other unsupported formats).

6 Conclusions

For what concerns classification tasks, we can conclude that *Posit<16,0>* can be a real alternative to *float_32* because we obtained similar performances in terms of accuracy and loss in every experiments. Unlike the object detection case, here we succeed in two experiments to convert both the weights of the model and tensors of test images. It would be interesting to conduct more experiments on a posit dedicated architecture (RISC-V) to see whether thanks to hardware acceleration we can reach similar results also in terms of execution time.

In the context of the object detection model, our results clearly demonstrate that Posits enable commendable outcomes without a significant precision decrease. The challenges we faced during implementation reveal the limited support for this format, even on widely used platforms like TensorFlow. Despite the promising advantages of Posits, we had to conduct our tests locally due to existing limitations, rather than on Colab. Our experience with Posits has emphasized their potential to balance precision and efficiency effectively, making them suitable for our object detection task. However, the limited integration of Posits into mainstream platforms underscores the urgent need for further development and comprehensive integration. This will ensure broader accessibility and facilitate their seamless incorporation into cutting-edge machine learning projects.

7 REFERENCES

[1] https://www.johndcook.com/blog/2018/04/11/anatomy-of-a-posit-number/

[2] https://www.sigarch.org/posit-a-potential-replacement-for-ieee-754/

[3] https://benchmark.ini.rub.de/gtsrb_news.html

[4] https://benchmark.ini.rub.de/gtsdb_news.html

[5]https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md

[6] https://neptune.ai/blog/how-to-train-your-own-object-detector-using-tensorflow-object-detection-api

[7] https://keras.io/examples/vision/retinanet/

Project code repository: https://github.com/mattemugno/seai_project