Memory and inference time considerations in a C translated QKeras model

Adamantia Dotsika (s2410583)
Supervisors: Dr. C.G. Zeinstra, Dr. N. Alachiotis , S. Bunda
Department: Datamanagement and Biometrics

***Abstract*—With the growth of applications using neural networks, there is an increase in need for compact C models. The work of [1] presents interesting results on QKeras models and its impact on memory footprint. With that in mind this paper presents a modified version of the keras2c library to adapt to the needs of QKeras models. The modified library is used for studying the influence of data representation on memory and inference time in C-translated Qkeras models. The results show a memory reduction of 2.5x in case of the fixed-point representation with no loss in inference time. Even though the output of the inference could not be studied in accuracy, the study shows interesting and promising results that need further investigation.**

I. Introduction

Machine Learning is widely used in a multitude of applications (some examples include medicine, spam sorting, speech and face recognition, translation, automation etc.) [7]. Multiple reasons are involved in the increasing need of neural networks to be deployed in mobile devices or limited resource systems. For some applications new data privacy laws require a local storage [1]. Other applications require easy deployment on existing models or real time inferences [3]. In any case it is interesting to study possibilities of memory size reduction.

Translating a machine learning model from Python to C is not a new concept. There are multiple articles that have focused on solving the issue of real time inference on an edge computing. The majority of those methods require large libraries or are time consuming and thus making them inefficient [3]. The work of [3] created a new C library for Keras API for real time inference. The library is simple and small in size compared to other solutions and puts out interesting results for Keras models [3]. The paper of [1] successfully reduced the memory footprint of a face recognition neural network from 32-bits down to 8-bits and 4-bits using QKeras.

A. Research Question

This paper aims to answer the following question: *How does number representation influence the memory footprint and inference time of a C-translated QKeras model?*

To answer this question a workflow has been developed aiming to answer the following subquestions chronologically:

- What are the possible number representations?
- How can keras2c methodology be adapted for quantized QKeras models?
- How are memory footprint and inference time influenced by the number representations of QKeras model in C?

II. Related Work

A. Number Representations

The definition of data representation is the way information is stored and described in computers [8]. There are two types of data representation, fixed-point and floating-point representation. The choice of the representation type depends on the application and needs of the programmer (storage available, precision etc.) [8]. Arithmetic and operations are influenced by the representation and it is therefore important to understand them. This paper will only focus on signed numbers therefore only the signed cases will be studied

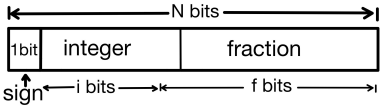

Fig. 1. N-bits fixed-point representation

1) Fixed-Point Representation: Figure 1 shows the N-bit fixed point representation. This type of data representation has a fixed amount of bits for the integer and fractional part of the number, meaning the radix point is always at the same place, hence its name 'fixed-point representation' [8]. The only important information that needs to be stored is the total number of bits (N) and either the fractional bits (f) or integer bits (i).

The integer representation of a N-bit number is given by [1]:

$$(-1)^s \cdot m \cdot 2^{-f} \tag{1}$$

In equation 1, s is the signed bit, m is the mantissa ($N-1$ with N the total number of bits) and f is the number of bits for the fractional part (the part after the radix point) which is given as $N-1-i$ (i is the amount of bits for the integer) [1]. 2^{-f} is a scaling factor and depending on the f parameter the representation will change.

When using a dynamic fixed point representation f can take a range of values, this will influence the position of the decimal point. Thus by regulating f, the amount of bits for the integer representation can vary [1]. The full fractional bit representation has the most promising results according to the results obtained by [1]. This means that no bits are given for the integer part of the representation and all bits of the mantissa are used for the fractional part, meaning it normalizes the integer between $[1, -1]$.

When using fixed-point representation two characteristics are important to keep in mind, range and precision. The range is defined as all the possible numbers that the data can have and the precision as the difference between two consecutive

numbers [8]. These two values have a trade-off, increasing one may result in a reduction of the other due to the limitation of the fixed-point representation.

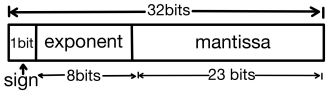

Fig. 2. 32 bit floating point representation

2) Floating-Point Representation: Figure2 shows an example of 32 bits floating point representation. Floating-point representation allows for a greater range compared to fixed-point representation [8]. It achieves this by using parts of the mantissa for exponent bits. This reduces the precision, as now parts of the mantissa are no longer accessible, but it increases the range considerably. In floating point representation the mantissa is the fixed point part of the number [8].

The floating point representation of a 32-bit number is given by [1]:

$$(-1)^s \cdot m \cdot 2^{e-127} \tag{2}$$

In equation 2, s is the sign bit, m is the mantissa of 23-bits and e is the exponent of 8-bits [1].

B. Quantization

The process of storing and operating data at lower bit-widths than floating point representation is defined as quantization [10]. Quantization is very useful when applications require a compact neural network. [4] and [10] explain in detail how quantization schemes work. In summary: the mapping of the floating point to a quantized bit integer is defined as [10]:

$$q = round(\frac{1}{S}r - Z) \tag{3}$$

In equation 3 q is the quantization (an N-bit integer depending on the quantization), r is a floating point of range $r \in [\alpha, \beta]$, S and Z are quantization parameters. The dequantization equation is shown in the equation below [10], [4]:

$$r = S(q + Z) \tag{4}$$

In order to find the quantization parameters the detail derivation can be found in [10] and in the Appendix A. The important information to remember is that S is the scale and is represented as a float and Z is the zero point with a bit representation identical to q. Z is an important requirement of quantization as zero needs to be represented without any error after the conversion as neural networks use zero-padding [10],[4].

C. Qkeras

Keras is a high level API that is part of tensorflow. Some of the reasons Keras is widely used is due to it's user friendly interface, it is modular and extensible, this makes Keras simple to use [6] [3]. Qkeras, is a quantized extension of Keras [1] [2] [9]. The benefits of using QKeras is that it is based on the same idea and principles as Keras and is intended to extend those functionalities that make Keras so widely used [2] [9]. In addition, this last point offers the possibility for QKeras to be used easily with Keras or replace Keras [1] [9] [2]. The layers of QKeras are similar to Keras and the main difference is that they label all variables with the quantization functions in order to create a shell and perform the necessary tasks [1] [2]. QKeras provides quantization-aware training (QAT), this has better accuracy results in comparison to post training quantization [2] [5]. One more key element of QKeras is that it allows a customized quantization per layer, meaning a potential heterogeneous quantization. Compared to homogeneous quantization where all the model is quantized post training by the same fixed point precision, heterogeneous quantization allows each layer to be quantized independently from the other layers [2]. The real benefits of heterogeneous quantization is further reduction of the model while simultaneously assuring a high accuracy [2].

QKeras can have three possible quantization representation: *dynamic fixed-point, exponent, sub-byte* [1]. Another addition to the possibilities of the library is the options concerning the scaling factor. The scaling factor (QKeras use the notation α) has the option of *None*, *Automatic* and *Auto po2* [1].

D. Keras2c [3]

Keras2c is a library that was introduced by [3]. It targets to translate a hdf5 file of a trained model into a callable C network. Hdf5 is a format used to store data efficiently [11]. Keras and QKeras use this type of file format to save the weights, parameters and model architecture. The idea of the library is based on the same design idea as Keras. The idea is that the layer stays as a simple callable function in Keras, making the forward pass simple [3]. Each layer will extract the parameters from the hdf5 file and write it on the C extension file. One of the key ideas of keras2c is the k2c tensor [3] introduced. All information will use the tensor datatype that they created in order to save the weights either on the stack or in memory [3].

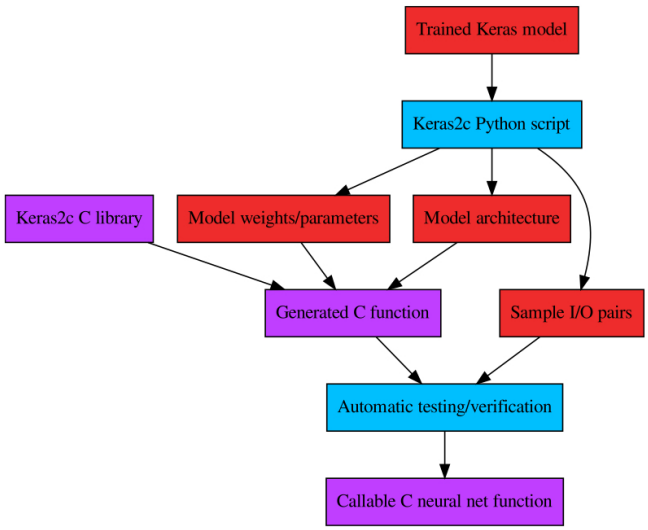

Fig. 3. Keras2c workflow [3]

Figure 3 shows the workflow of keras2c project [3]. The hdf5 file of the trained model is passed with all information in the python script of the Keras2c. From the script the weights and architecture are extracted and used to generate the C file. The Keras2c C library is written and will be compiled along

with the C file. The python script also generates a test file to verify the C model.

E. Summary

This section provides a lot of information that will be needed later on during the implementation of this project. The first subquestion can be already be answered. There are two major data representations that can be used for storing the weights and parameters. The floating point representation is the default 32-bits implementation. The fixed point representation provides more compact options. Depending on the number of bits used for the storage the fixed point representation can be more flexible as the number of fractional bits can change. The previous statement allows for dynamic fixed point representation. The choice of representation relies fully on the type of application and the programmer. A resource limited application will require a fixed point representation. An application with large computer storage could use floating point representation.

III. Method

Figure 4 shows the workflow that will serve as a guidance for the implementation of the goal of the project. The figure is separated in two parts the, the python part on the left and the C part on the right. The green boxes are the python scripts that the Mnist neural network is written. The light pink are the output of the scripts. Between the Mnist Qkeras and the HDF5 file there is an additional function that will be added (the saving function) in order to meet the requirements of the input of the qkeras2c script. Blue boxes represent the parts that are written using the keras2c model. The red boxes are the keras2c library that are used with no changes added.

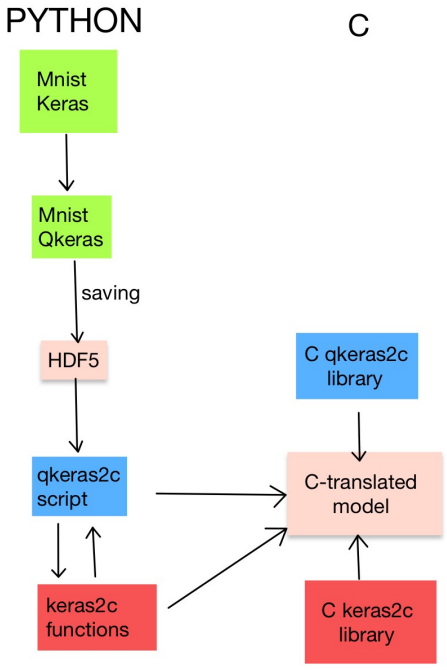

Fig. 4. Project workflow

The workflow of figure 4 can be separated in three different sections: *Model coding*, *Qkeras C-translator* and *C model testing*

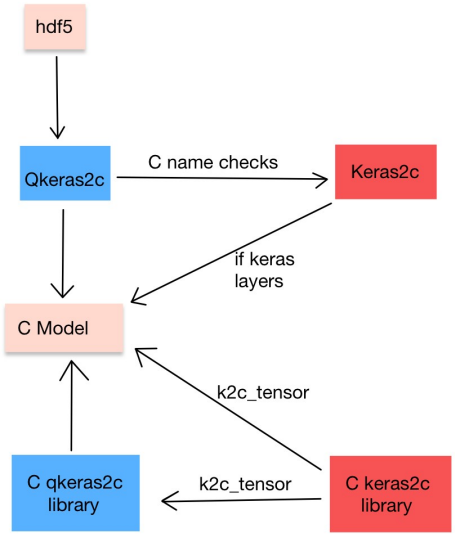

Fig. 5. Qkeras2c workflow and connections with the Keras2c library

A. Model Coding

Before the Qkeras C-translator is implemented a simple neural network application is implemented using Keras and QKeras in Python. The application that is chosen is the Mnist neural network. The structure of this neural network can be found in Appendix B, C). The model architecture and weights/parameters is saved in an HDF5 file. Currently Qkeras only support saving of model weights, therefore an additional saving function is written to ensure model architecture and quantization information are stored in the file. This type of information is required at the input of the Qkeras C-translator.

B. Qkeras C-translator

The Qkeras C-translator is based on the design of the Keras2c library. Figure 5 shows the Qkeras2c structure and links with the existing Keras2c library. For consistency the colors have similar meanings as for figure 4. Blue boxes are the contributions of this paper, red boxes are unchanged existing libraries (keras2c) and light pink represent the outputs. The translator has three steps: *name compatibility*, *layer & weight C generation* and *C library*.

1) Name Compatibility: In C there is a strict requirements concerning names given to variables, functions and etc. Therefore the first step of the translator is a name check for each layer and the output C name file. To do so the existing functions of keras2c library are called from the qkeras2c.

2) Layer & Weight C generation: This step consist of the extraction of the model architecture and weight/parameters. Depending on the application the programmer can choose to write the weights in the heap or in memory. This option exist in keras2c and is adapted for the needs of the qkeras2c. The weights is then written depending on the layer. The last step is to write the layer itself and to do so the input variables needed for inference are written in the C file. If the model uses some Keras layers qkeras2c will call the keras2c to write those functions.

The previous section showed that there is two different ways of representing data. In Python data are stored in

`float32` by default, this is not the case for C. In C every variable, function need to be explicitly define the data type. For Qkeras the translator aims to output the weights in a fixed-point representation. To do so the weights are converted to their fixed-point representation and then written in the C file as an integer.

3) C library: For the C libraries the `k2c_tensor` datatype is used for the structure. Information are passed through this datatype. This datatype is define in the keras2c C library. A new qkeras2c C library is defined in order to allow operations of the QKeras model. Currently only the QDense, QActivation and QConv layers is going to be implemented. For the qactivation only the `quantized_relu` will be possible.

C. C Model Testing

The keras2c outputs a `test_suite` C file in order to test the output C model. There is two things that the test file checks, the time required for inference and the error of the model output. As the qkeras2c is based on the keras2c, the `test_suite` will be implemented to fit the needs of QKeras.

D. Arithmetic

This subsection concerns the arithmetic that will be used. Depending on the data representation the implementation of arithmetic differ. As the weights are stored using integer datatype the arithmetic will require integer only arithmetic. In order to avoid this (integer only arithmetic is harder to be implemented) the weights will be converted back to floating point for calculations. Before storage the weights will be converted to an integer once again.

E. Summary

This section explains the implementation workflow and method. The subquestion: *How can keras2c methodology be adapted for quantized QKeras models?* can be answered at this point. The keras2c targets Keras models, in order to achieve a QKeras translator the QKeras includes functions that translate Qlayers. In some cases the qkeras2c uses functions of the keras2c python script. Additional utilities functions are added in order to allow fixed-point representation. The C libraries of qkeras2c are programmed to allow Qkeras layers to do inference with the same design structure as keras2c.

IV. Results & Discussion

The last subquestion to be answered is: *How are memory footprint and inference time influenced by the number representations of QKeras model in C?*. This section tries to answer this in order to fully answer the main research question.

In the previous section it was mentioned that a `test_suite` is automatically created, in order to calculate the error and run time of the inference. A number of tests were run in order to have a clear comparison. The Keras Mnist model was used to output a Keras C file. The results of this file will be used to compare with the Qkeras file.

The results of the test are summarise in the figure 6 below:

Model Type	C File Size	Executable File Size	Max Absolute Error	Average Time
Keras	2012Kb	937Kb	1.19e-07	9e-04s
Keras with CSV	4Kb	476Kb & 1914Kb CSV	8.84e-07	1.3e-03s
QKeras int	373Kb			
QKeras int with CSV	6Kb	506Kb & 445Kb CSV	8.88e-01	9e-04s

Fig. 6. Result table for the different types of models

The table of figure 6 has four different case studies. The first consist of the Keras C model (generated using the keras2c library) with weights saved on the heap. The second case is the Keras C model with weights saved in memory using csv files. The third case is the Qkeras C model (generated using the new library) with weights saved in integer representation in the heap. In this case the arithmetic was implemented using integer and floating point conversions. The last case is the QKeras C model with integer representation and weights saved in memory. This case uses integer datatype.

The results of figure 6 show that storing the weights with fixed-point representation in memory, has a clear size reduction compared to the Keras model. The Keras model uses 1914Kb of memory in total to store the weights in a CSV file and 476Kb for the executable. The executable requires the CSV files to perform the inference. In the QKeras model which is stored in integer form, requires 445Kb of memory to store the weights and 506Kb for the executable file. This means that there is more than 4x reduction in the CSV file for memory in fixed-point representation. In total there is a 2.5x reduction in the total memory footprint. The inference time is not reduced compared to the Keras model. This shows a clear improvement in memory footprint, with no loss in inference time.

The table above also shows a high error percentage for the Qkeras model. The reason for this error is due to the integer arithmetic implementation. These results were obtained, using an initial integer arithmetic which was not optimal. The model using integer and floating conversion to perform arithmetic, was not able to work properly and could not output a time of inference nor error at this point.

The last subquestion can now be answered. The memory is influenced by the usage of the fixed-point representation. Even though this is a good achievement, unfortunately the current results don't show anything interesting with respect to the inference itself. It is important for a neural network to have a high accuracy. Therefore further investigation is required to study the influence on the accuracy of the inference, with more optimal and working algorithms.

A. Improvements & Future Work

Due to time constraints the qkeras2c was implemented to translate a very simple model. Currently floating-point arithmetic is supported, which is useful for mobile devices. If the application requires the model to be used in a micro-processor, integer only arithmetic should be implemented to further reduce the memory footprint. The qkeras2c library is a project that I will keep investigating further after the end of this project. Some of the future improvements concern the efficiency of the algorithms, the current version was implemented to have a proof of concept. Another idea that would be studied is the possibility of conversion of the keras2c C library to support integer arithmetic to allow for a fully independent library that could potentially import functions and convert them to it's needs. For the integer arithmetic,

the algorithms were written but errors were not allowing the files to compile, further work is required in order to allow for a successful use of integer arithmetic. Lastly the idea of possibly choosing the type of arithmetic during the inference is interesting as it would allow for the converted model to be used in a larger range of applications.

V. Conclusion

This paper aimed to study the impact of data representation in the memory footprint and inference time. The output of this thesis shows that there is a clear impact in memory size, when using fixed-point representation. In addition the inference time is not influenced. Despite this, it was not possible to prove that this method produces accurate inference. The reasons for this has been identified in the discussion. Even though the accuracy doesn't show satisfying results, this work shows interesting outcomes. The methodology is not a failure but rather a start for further studies.

References

[1] Sebastian Bunda. *Limited Resource Optimization for Face Recognition Convolutional Neural Networks Sub-byte quantization of MobileFaceNet using QKeras*. Tech. rep. 2022.

[2] Claudionor N. Coelho et al. "Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors". In: (June 2020). DOI: 10.1038/s42256-021-00356-5. URL: http://arxiv.org/abs/2006.10159%20http://dx.doi.org/10.1038/s42256-021-00356-5.

[3] Rory Conlin et al. "Keras2c: A library for converting Keras neural networks to real-time compatible C". In: *Engineering Applications of Artificial Intelligence* 100 (Apr. 2021), p. 104182. ISSN: 0952-1976. DOI: 10.1016/J.ENGAPPAI.2021.104182. URL: https://collaborate.princeton.edu/en/publications/keras2c-a-library-for-converting-keras-neural-networks-to-real-ti.

[4] Benoit Jacob et al. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference". In: (Dec. 2017). URL: http://arxiv.org/abs/1712.05877.

[5] Chutian Jiang. "Efficient Quantization Techniques for Deep Neural Networks". In: *Proceedings - 2021 International Conference on Signal Processing and Machine Learning, CONF-SPML 2021*. Institute of Electrical and Electronics Engineers Inc., 2021, pp. 271–277. ISBN: 9781665417341. DOI: 10.1109/CONF-SPML54095.2021.00059.

[6] *Keras io*. URL: https://keras.io/.

[7] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. *Deep learning*. May 2015. DOI: 10.1038/nature14539.

[8] Miles Murdocca and Vincent Heuring. *Computer Architecture and Organization*. 2007, p. 28.

[9] *QKeras Github*. URL: https://github.com/google/qkeras.

[10] *Quantization for Neural Networks - Lei Mao's Log Book*. URL: https://leimao.github.io/article/Neural-Networks-Quantization/.

[11] *The HDF Group*. URL: https://www.hdfgroup.org/solutions/hdf5/.

Appendix

A. Quantization

The following section can be found in [10] in more details, it is included in this paper for easy access on additional information concerning quantization and derivations.

1) Derivation of Quantization Parameters: The range of possible values for r is $r \in [\alpha, \beta]$ and for q is $q \in [\alpha_q, \beta_q]$. The range need to also follow the mapping requirements of r and q (equations 3 and 4). Solving the following system of equations will provide the quantization parameters.

$$\begin{cases} \beta = S(\beta_q + Z) \\ \alpha = S(\alpha_q + Z) \end{cases} \tag{5}$$

In equation 5 subtracting α from β will give the scale parameter S.

$$\beta - \alpha = S(\beta_q - Z) - S(\alpha_q - Z) = S(\beta_q - \alpha_q)$$

$$S = \frac{\beta - \alpha}{\beta_q - \alpha_q} \tag{6}$$

By substituting equation 6 in the first equation of 5 results in the zero-point.

$$\beta = \frac{\beta - \alpha}{\beta_q - \alpha_q}(\beta_q + Z)$$

$$\beta(\beta_q - \alpha_q) = (\beta - \alpha)(\beta_q + Z)$$

$$-\beta\alpha_q = (\beta - \alpha)Z - \alpha\beta_q$$

$$Z = \frac{\alpha\beta_q - \beta\alpha_q}{\beta - \alpha} \tag{7}$$

To prove that Z is the zero point the following equation needs to be solved:

$$q_0 = round(\frac{1}{S}0 - Z) = round(-Z) = -Z \tag{8}$$

B. MNIST model

The figure below 7 shows the Keras model for the MNIST application. The input is again 784 and there is two hidden layers of 128 nodes each. The activation is Rectified Linear Unit (RELU). Between the hidden layers and the output, the Dropout function of Keras is used to ensure that the model is not overfitting. In this simple model the Dropout layer is not really needed. The output layer has 10 nodes for the 10 different numbers from 0 to 10. The activation function for the last layer is the softmax in order to output a probability. This will output the probability of each number, the highest probability is the predicted number.

```
model = Sequential()

model.add(Dense(units=128, input_shape=(784,), activation='relu'))
model.add(Dense(units=128, activation='relu'))

model.add(Dropout(0.25))
model.add(Dense(units=10, activation='softmax'))

model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
```

Fig. 7. Keras MNIST model

C. QMNIST model

Figure 8 shows the Python code of the QKeras version of the MNIST model. The model consist of one Input layer like the Keras version with an input of 784 nodes. The hidden body of the model consist of 2 QDense layers, with 128 nodes and 8bits quantization and 0 bits for the integer part. The activation of those layers is quantized_relu (the quantized version of the relu function of Keras), the quantization bits are 8 bits for the total and 3 bits for the integer part. The output layer consist of a QDense layer with 10 nodes and same quantization pattern as the hidden layers. The output activation function is softmax as we want to have a probability as an output to classify the results.

```
# Fully Connected Neural Network
x = x_in = Input((784))
x = QDense(128,
           kernel_quantizer = quantized_bits(8,0, alpha=1),
           bias_quantizer = quantized_bits(8,0, alpha=1))(x)
x = QActivation('quantized_relu(8,3)')(x)
x = QDense(128,
           kernel_quantizer = quantized_bits(8,0, alpha=1),
           bias_quantizer = quantized_bits(8,0, alpha=1))(x)
x = QActivation('quantized_relu(8,3)')(x)
x = Dropout(0.25)(x)
x = QDense(10,
           kernel_quantizer = quantized_bits(8,0, alpha=1),
           bias_quantizer = quantized_bits(8,0, alpha=1))(x)
x = Activation('softmax')(x)

# Creates the Model
model = keras.Model(inputs=[x_in], outputs=[x])
model.summary()

# Compile Model
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Fig. 8. QKeras MNIST model

D. Save function for QKeras

The saving function implemented to save the weights/parameters, model architecture and quantization information of QKeras model follows the structure of the `model_save_quantized_weights` function. The reason QKeras currently only save the weights can be found in the previous mentioned function, it calls the Keras `save_weights` function. Therefore the new QKeras function only changes the call to the Keras function `save` that will save the full model information (including the quantization and model architecture).

E. Floating to Fixed point Representation in Python

A major difference between the keras2c and qkeras2c library is the fact that weights are stored in fixed-point representation. To do so a function that converts from floating point to fixed point.

```
def convert_array_to_fixed(layer):

    weight_array = layer.get_weights()
    new_weights=[]
    for weight in weight_array:
        (bits,integer,alpha) = get_quantized_bits(layer)
        fraction = bits-1-integer
        to_fix = NumpyFloatToFixConverter(True,bits, fraction)
        new_weights.append(to_fix(weight))

    return new_weights
```

Fig. 9. Convert to a fixed-point representation function

In order to ensure that the function work two packages need to be imported `qkeras` and `NumpyFloatToFixConverter` from the `rig.type_casts` package.

In order to print the weights in the C file as an integer using the following syntax: `s += ":".format(temp[i]) + ','` in the `qarray2c` function of the `qweights2c` python file.

F. C translated QKeras model

```
qk2c_qdense(&q_dense_24_output,input_10_input,&q_dense_24_kernel,
    &q_dense_24_bias,8,0,1,qk2c_linear,q_dense_24_fwork);
qk2c_quantized_relu(q_dense_24_output.array,q_dense_24_output.numel, 8,3);
qk2c_tensor q_activation_18_output;
q_activation_18_output.ndim = q_dense_24_output.ndim; // copy data into output struct
q_activation_18_output.numel = q_dense_24_output.numel;
memcpy(q_activation_18_output.shape,q_dense_24_output.shape,QK2C_MAX_NDIM*sizeof(size_t));
q_activation_18_output.array = &q_dense_24_output.array[0]; // rename for clarity
qk2c_qdense(&q_dense_25_output,&q_activation_18_output,&q_dense_25_kernel,
    &q_dense_25_bias,8,0,1,qk2c_linear,q_dense_25_fwork);
qk2c_quantized_relu(q_dense_25_output.array,q_dense_25_output.numel, 8,3);
qk2c_tensor q_activation_19_output;
q_activation_19_output.ndim = q_dense_25_output.ndim; // copy data into output struct
q_activation_19_output.numel = q_dense_25_output.numel;
memcpy(q_activation_19_output.shape,q_dense_25_output.shape,QK2C_MAX_NDIM*sizeof(size_t));
q_activation_19_output.array = &q_dense_25_output.array[0]; // rename for clarity
qk2c_tensor dropout_3_output;
dropout_3_output.ndim = q_activation_19_output.ndim; // copy data into output struct
dropout_3_output.numel = q_activation_19_output.numel;
memcpy(dropout_3_output.shape,q_activation_19_output.shape,QK2C_MAX_NDIM*sizeof(size_t));
dropout_3_output.array = &q_activation_19_output.array[0]; // rename for clarity
qk2c_qdense(&q_dense_26_output,&dropout_3_output,&q_dense_26_kernel,
    &q_dense_26_bias,8,0,1,qk2c_linear,q_dense_26_fwork);

k2c_softmax(q_dense_26_output.array,q_dense_26_output.numel);
activation_4_output->ndim = q_dense_26_output.ndim; // copy data into output struct
activation_4_output->numel = q_dense_26_output.numel;
memcpy(activation_4_output->shape,q_dense_26_output.shape,QK2C_MAX_NDIM*sizeof(size_t));
memcpy(activation_4_output->array,q_dense_26_output.array,activation_4_output->numel*sizeof(activation_4_output->array
```

Fig. 10. Model structure in the C file