Supporting Posits in Deep Learning Frameworks: A PyTorch Implementation

Afonso Vieira da Silva Luz
Instituto Superior Técnico, Universidade de Lisboa, Portugal
afonso.luz@tecnico.ulisboa.pt

***Abstract*—Reducing the energy consumption of computationally intensive deep learning implementations has received a growing interest in the last years. This is particularly relevant in applications where there are strict energy restrictions, such as space and aerial devices. To this end, the posit number format has shown promising results as a more energy efficient replacement to the standard IEEE-754 floating-point for deep learning computations. Recent research suggests that 16-bit posits achieve similar results as 32-bit floating-point and even smaller posits can be used to train and evaluate deep learning models. However, to study the use of posits for deep learning, researchers have to develop customized functions, since the most popular deep learning frameworks do not yet support posits. This work aims at bridging this gap, by integrating posits natively in PyTorch, the most popular framework for deep learning research. The proposed implementation makes posits a built-in data type in the framework, which means that they can be used in the same way as any other data type that the framework already supports. To validate the implementation, the convolutional neural network LeNet-5 was trained and tested using posits on the MNIST and FashionMNIST datasets. The obtained results with 16-bit posits were similar to those with 32-bit floating-point, suggesting that the implementation of the considered posit operators is correct. To disseminate this contribution, the produced code and documentation was made available on a public GitHub repository.**

***Index Terms*—Posit numerical format, Deep Learning (DL), Neural Networks (NNs), PyTorch**

I. Introduction

Deep Learning (DL) is, currently, one of the hottest topics of research, with applications ranging from Natural Language Processing to Computer Vision [1]. The most popular DL model types are Deep Artificial Neural Networks (NNs) [2]. These are comprised of a variable number of layers, each consisting of neurons holding weights that get updated with the labeled input data. In complex networks, the number of weights can reach hundreds of millions [3], representing very high computational demands.

To design and train NNs, there are several available frameworks that abstract the implementation details from the end user. The most popular are PyTorch [4], developed by Facebook, and TensorFlow [5], developed by Google. The first is currently the most popular among researchers, while the latter is the most used in production [6].

Given its computationally intensive nature, there is significant research interest in reducing the memory footprint and energy consumption of the operations performed by DL models [7]. This is particularly relevant in domains where there are strict constraints on processing capacity and energy consumption, such as space applications [8].

One possible approach to reduce this footprint is by searching for alternative formats to the traditional IEEE-754 32-bit floating-point numbers for computations, such as integers [9] and fixed-point numbers [10]. The introduction of the posit number format in 2017, with claims that it offers a higher accuracy and requires simpler hardware and exception handling structures than IEEE-754 floating-point numbers [11], has led to an increasing interest in using it within DL applications [12]–[14]. Recent research as shown that the same accuracy can be achieved with 16-bit posits as with 32-bit floats, and even smaller posit configurations have achieved state-of-the-art results [15] [16]. This means that, by using less bits, there is the potential of reducing hardware resources, energy consumption and memory footprint.

However, in order to train and test DL models with Posits, researchers have to develop all the functions and operators used for DL computations from scratch [15] [16]. This constitutes a significant overhead in the research process, which could only be mitigated if the most common DL frameworks supported the posit number format.

This premise constituted the motivation for this work: to introduce support of posits in the most widely used DL framework for research, PyTorch. This way, the main objectives of this work can be summarized as:

- Provide posit support to the main layers and functions of NNs in PyTorch;
- Expose these functions in the Python frontend in the same way as other native PyTorch data types;
- Document the process of extending posit support for further operators;
- Provide a tool to test future developments of hardware for posits within PyTorch.

The rest of this paper is organized as follows: Section II presents the posit number format, the core concepts behind DL and the most popular frameworks to design DL models. Given the choice of the PyTorch framework for this work, section III presents its frontend API. Then, in section IV, the internal structure of the framework is described. Following this structure, section V details the process of supporting NN operators for posits in the framework. Section VI presents the experimental evaluation conducted to validate this implementation. Finally, Section VII concludes with some final remarks about this work and future studies.

II. Background

Given that the purpose of this work is to integrate a novel numerical format into an established Deep Learning framework, it is important to understand its two main topics: the posit number format and Deep Learning. This section briefly explains these topics.

A. Posit Number Format

Despite the current ubiquity of IEEE's floating point format in computer systems, it has some problems and limitations, such as breaking linear algebra laws (e.g. due to the rounding process, $(a+b)+c \neq a+(b+c)$), overflowing to infinity and underflowing to 0, or the complexity of its manipulation, especially due to the multiple NaN values, leading to complex hardware to support it [17].

From the desire to overcome some of these problems, John L. Gustafson has recently proposed a set of alternative formats, denoted by Universal numbers (Unums). Firstly, Type I unum [18] introduced an extra bit to IEEE's floating point to assert whether the represented number was exact or whether it represented the lower end of an interval containing that number. With the introduction of Type II unum, compatibility with IEEE's standard was broken, in order to have more freedom to make it more hardware friendly [19].

In 2017, Type III unum - also known as Posit - was introduced, having in mind the complete replacement of IEEE's floating point. Gustafson states that this new format offers a larger dynamic range, a higher accuracy and requires simpler hardware and exception handling structures than traditional IEEE-754 standard floating-point numbers [11].

A posit number is defined by its total number of bits ($nbits$) and by the size of the exponent field (es). This way, a posit configuration is usually represented in the form posit($nbits$, es). The layout of a posit number (p) is represented in Figure 1, and its decoding is shown in equation (1).

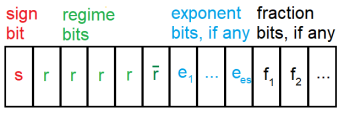

Fig. 1. Generic representation of an $nbits$-bit posit with es exponent bits.

$$x = \begin{cases} 0, & p = 000...0, \\ \pm\infty = \text{NaR}, & p = 100...0, \\ (-1)^s \times useed^k \times 2^e \times f, & \text{all other } p. \end{cases} \quad (1)$$

The meaning of the sign bit is similar to other formats: 0 for positive numbers and 1 for negative numbers. If the number is negative, the 2's complement of the other fields must be taken before decoding. The regime field is characterized by a run of identical bits (r) that is either terminated by an opposite bit ($\bar{r}$) or by using up all the $nbits-1$ bits. The numerical value (k) of the regime is given by the count of identical bits in this run. For a run of m bits, all equal to 0, $k=-m$; if they are 1s then $k=m-1$.

As represented in equation (1), the numerical value of the regime, k, represents an additional exponent, applied over $useed = 2^{2^{es}}$. Hence, this scale factor is given by $useed^k$. The value encoded by the exponent bits (e) represents a scaling by 2^e. Finally, just as in normalized IEEE's floats, the value encoded in the fraction bits (f) has a hidden bit at 1. Therefore, the scale factor is $1.f$, as shown in equation (1).

There are two special encodings that represent 0 (all bits at 0) and Not a Real (NaR) or $\pm\infty$ (first bit at 1 followed by all bits at 0). Having only two exception values with simple bit patterns makes their manipulation at the hardware level easier than the several NaN values of floats [11].

Contrary to IEEE floats, there is no bias for the exponent field, since negative values of k already lead to negative exponents. The variable size of the fraction field - determined by the size of the regime - leads to a tapered precision, that is, numbers with magnitude near 1 have more fraction bits therefore greater precision, whereas very small and very large numbers have smaller precision. This is illustrated in Figure 2 where the decimal accuracy (i.e. how many digits to the right of the decimal point are correctly represented) is compared for 8-bit fixed point, 8-bit floatint-point and different configurations of 8-bit Posits.

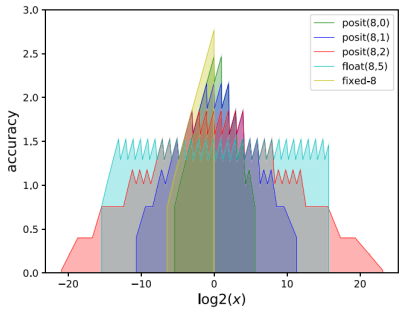

Fig. 2. Decimal accuracy of 8-bit fixed-point, 8-bit floating-point and several 8-bit Posits, obtained from [14].

The dynamic range of a positive ($nbits$, es) posit number is $[useed^{nbits-2}, useed^{2-nbits}]$, which means that, for a given posit size (nbits), the range increases only with the increase of the number of exponent bits, since $useed = 2^{2^{es}}$. Just as in floats, increasing the size of the exponent field comes at the cost of decreasing the size of the fraction, hence reducing precision.

B. Deep Learning - Neural Networks

Deep Learning (DL) is a subfield of Artificial Intelligence characterized by models with multiple layers, where each layer builds knowledge on top of the previous one [1]. These models are fed with input data and, by iterating over this data, they update their parameters, effectively learning from experience. The advantage of DL models over simpler machine learning techniques is their ability to capture complex non-linear behaviours by decomposing them into simpler tasks addressed by each layer.

Currently, the most widespread DL models are Deep Artificial Neural Networks (NNs). Drawing inspiration from the synapses that occur between neurons in the human brain, NNs have weights connecting neurons in consecutive layers. These weights represent the transmission of a signal between neurons, which, together with an associated bias, pass through a non-linear activation function to determine if the neuron is "fired" or not.

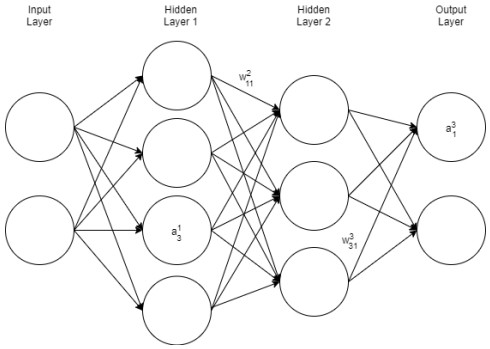

Fig. 3. Example of a Neural Network with 2 hidden layers

Figure 3 shows an example NN with 3 layers (the input layer is not counted). In order for a NN to be considered deep, it should have at least 2 hidden layers. The input layer is simply the input fed to the network: the pixels that constitute an image, for example. The output layer is responsible for connecting back to the real world problem at hand. If, for example, the problem is that of determining the class of an input image, each output neuron will represent the probability of the image belonging to a given class, hence there will be as many output neurons as there are classes of images.

Each circle in Figure 3 represents a neuron, and the arrows between them are the connections, each of them with an associated weight. Each neuron is connected to all the neurons of the previous layer and to all the neurons of the next layer.

The output value of neuron i in layer $l > 0$, a_i^l, is given by equation (2).

$$a_i^l = f(z_i^l), \text{ with } \quad z_i^l = b_i^l + \sum_{j \epsilon \text{previous layer}} w_{ij}^l a_j^{l-1} \tag{2}$$

z_i^l is the intermediate output, b_i^l the bias, w_{ij}^l the weight connecting neuron j of the previous layer with this neuron and f is a non-linear activation function. For the input layer, a_i^0 is simply equal to the the content at index i of the input vector.

The ultimate goal of a NN is to receive an input (an image, for example) and give a meaningful output (predict if the image is of a dog, a person, etc., in an image classification problem). This process is called **inference**. However, in order for the network to be able to make correct predictions, it must first learn from experience, just as humans do, in a process called **training**.

Training a NN consists of updating its parameters (weights) in order to obtain a model that can address as accurately as possible the problem at hand. During the training procedure, the measure that is used to evaluate whether the model is accurate is a **loss function** (L). This function compares the predicted value, $\hat{y}$, and the correct label, y_{true}, penalizing wrong predictions, $L = f(\hat{y}, y_{true})$.

Since the predicted value is a function of all the weights of the network, the loss will also be a function of these weights: $L = g(w_{11}^1, w_{12}^1, ..., w_{ij-1}^l, w_{ij}^l, y_{true})$. Therefore, the goal of the training procedure is to find the weights that minimize this loss function.

Among the several techniques for NN training, the **gradient descent** algorithm is the most popular [20]. The concept of gradient descent is illustrated in Figure 4 for the simple case of a single variable function where, for each step, the weight is updated in the direction opposite to that of the gradient.

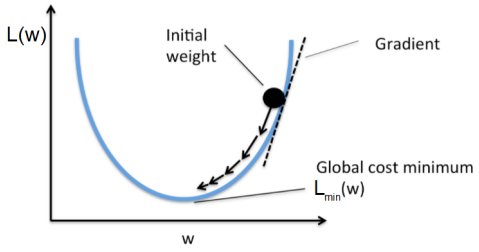

Fig. 4. Gradient Descent illustrated for a single variable function, obtained from [21]

Mathematically, this can be formulated as

$$w(t+1) = w(t) - \eta \frac{\partial L}{\partial w}\bigg|_{w(t)}, \tag{3}$$

where t is the step being considered and η is a hyperparameter called the **learning rate**, used to adjust the magnitude of the weight updates.

In order to calculate the derivative of the loss with respect to a certain weight, the chain rule is usually used, following equation (2) for the activation of each neuron. The influence on the loss is felt through different paths in the network. Taking as an example Figure 3, the weight w_{11}^2 influences the first neuron of the second hidden layer, which in turn affects both output neurons, effectively affecting the loss in two ways. This way, the derivative of the loss with respect to this weight will be the sum of these two paths. The general formulation is, then:

$$\begin{cases} \frac{\partial L}{\partial w_{ij}^l} = \frac{\partial L}{\partial a_j^l} f'(z_j^l) a_i^{l-1} \\ \frac{\partial L}{\partial a_j^l} = \sum\limits_{m \epsilon \text{next layer}} \frac{\partial L}{\partial a_m^{l+1}} f'(z_m^{l+1}) w_{mj}^{l+1}. \end{cases} \tag{4}$$

From equation (4), the sum used to calculate $\frac{\partial L}{\partial a_j^l}$ refers to the next layer, which means that to calculate the gradients it is more practical to do the computation in reverse: start from the loss in the last layer, going backwards up to the input layer. As a result, this algorithm is know as **backpropagation**, the most commonly used to train NN [22].

In summary, the training of a neural network has 4 main phases:

- **forward pass**: feeding an input to the network and propagating forward to obtain an output;
- **loss calculation**: with the predicted output and the true value, calculating the loss according to the chosen loss function;
- **backward pass**: calculating the gradients with respect to the weights, starting from the last layer up to the beginning;
- **weight update**: following the principle of gradient descent, update the weights to minimize the loss function.

C. Deep Learning Frameworks

In order to build, train and perform inference with NNs, one can either implement all the mathematical operations for the different layers, derivative calculations for backpropagation, etc. from scratch, or leverage existing frameworks that provide higher level APIs for these operations. Some of the most popular frameworks are PyTorch, TensorFlow, Keras (built on top of TensorFlow), Caffe and CNTK.

Among these, PyTorch [4] and TensorFlow [5] are the most established [6]. The first is developed by Facebook and the second by Google, but both of them are open source with very active communities. Google's TensorFlow was released earlier with a stable API. It provided solutions both for server and mobile and thus became dominant in production. However, given PyTorch's shallower learning curve and the easiness to experiment and debug novel architectures, researchers have been switching to PyTorch, and it has surpassed TensorFlow in popularity, as shown in Figure 5.

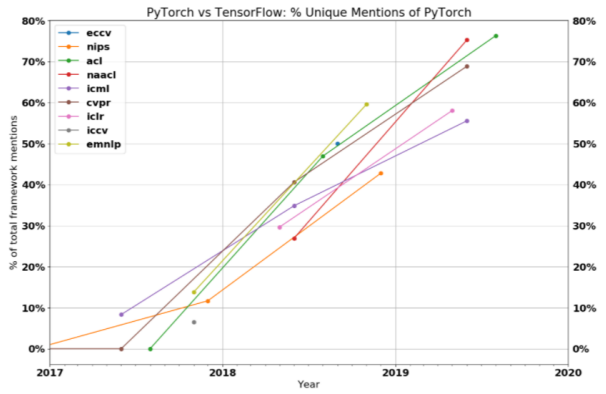

Fig. 5. Unique mentions of PyTorch in DL articles that either mention PyTorch or TensorFlow, obtained from [6].

Both frameworks offer similar functionalities, namely high level Python APIs for all NN layers and associated functions, along with automatic differentiation, responsible for calculating the gradients needed for backpropagation. From the growing popularity of PyTorch within the research community, as shown in Figure 5, this was the chosen framework to be extended for posits in this work.

III. PyTorch Framework API

As mentioned in the previous section, PyTorch is currently one of the most popular frameworks to build DL models. Its Application Programming Interface (API) provides functions for the most common DL operations, while abstracting their implementation details from the end user. This section describes the core data structure within PyTorch and presents an example of using the frontend API for the end to end design and train of a NN model.

A. Tensors as the base data structure

Most operations within NN design and training can be represented by a matrix multiplication. Hence, PyTorch uses Tensors (multi-dimensional arrays) to represent all data within NNs.

As an example, gray-scale images can be represented by a 2-D array, where each row and column position maps a pixel in a certain height and width position on the image. If the image is coloured, a new dimension representing each of the 3 channels is added. If images are grouped (within a batch, for example), an extra dimension is added, where the index denotes the image's position in the batch. A 4-D tensor is, thus, the adequate data structure to store a batch of coloured images.

Along with the stored data, PyTorch's Tensor class also stores metadata that is relevant for NN design, namely:

- **dtype**: the type of the stored data (32-bit float, 16-bit integer, 64-bit complex, etc);
- **device**: whether the tensor is allocated on the CPU or on the GPU, and in which core;
- **layout**: How the tensor's data is stored in memory (rows concatenated contiguously, a sparse matrix representation, etc);
- **requires_grad**: if, when integrated in a NN, this tensor will need to have its gradient calculated for backpropagation.

From these, the last attribute is particularly important. It constitutes the main difference between Tensors dedicated to NNs and multi-dimensional arrays. If this attribute is set to *True*, PyTorch will automatically store the gradient of the loss with respect to this tensor. Moreover, if other tensors are constructed through mathematical operations on a tensor with *requires_grad=True*, these tensors will also keep track of their gradient. With this, end users do not need to compute gradients, everything is done in the background once the network's design is established.

B. Model Training Example

PyTorch provides a base class *Module* from which the class representing the network should be derived. In the *__init__* method of the network's class, the design of the network should be specified, through the layers that constitute it. This class should also implement the method *forward*, responsible for passing the input through the network's layers and returning the output.

After the definition of the model, the 4 phases of the training process described in section II-B (forward pass, loss calculation, backward pass and weight update) are done for each batch of the training data. After going over the entire training data, an **epoch** is said to be completed. For a neural network to achieve a desirable precision on unseen data, the training is repeated for several epochs. However, if the number of epochs is too large, there is the risk of overfitting the training data and thus perform poorly on unseen data.

Combining all these concepts, listing 1 provides an end-to-end example of training the LeNet-5 network over the MNIST dataset.

```
class LeNet5(nn.Module):
    def __init__(self, n_classes):
        super(LeNet5, self).__init__()

        self.feature_extractor = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=6,
     kernel_size=5, stride=1),
            nn.Tanh(),
            nn.AvgPool2d(kernel_size=2),
            nn.Conv2d(in_channels=6, out_channels
    =16, kernel_size=5, stride=1),
            nn.Tanh(),
            nn.AvgPool2d(kernel_size=2),
            nn.Conv2d(in_channels=16, out_channels
    =120, kernel_size=5, stride=1),
            nn.Tanh()
        )
        self.classifier = nn.Sequential(
            nn.Linear(in_features=120, out_features
    =84),
            nn.Tanh(),
            nn.Linear(in_features=84, out_features=
    n_classes),
        )

    def forward(self, x):
        x = self.feature_extractor(x)
        x = torch.flatten(x, 1)
        logits = self.classifier(x)
        return logits

model = LeNet5(n_classes=10)
optimizer = torch.optim.Adam(model.parameters())
loss_criterion = nn.CrossEntropyLoss()

train_dataset = datasets.MNIST(root='mnist_data',
    train=True, transform=transforms.ToTensor(),
    download=True)
train_loader = DataLoader(dataset=train_dataset,
    batch_size=32, shuffle=True)

for epoch in range(0, epochs):
    for X, y_true in train_loader:
        optimizer.zero_grad()
        y_hat  = model(X)
        loss = criterion(y_hat, y_true)
        loss.backward()
        optimizer.step()
```

Listing 1. Training the LeNet-5 network on the MNIST dataset

IV. PyTorch Internal Structure

In order to understand how posits were supported in PyTorch as one of the native data types, it is first important to understand the internal structure of PyTorch's codebase and the main abstraction layers in it. Even though the codebase is constantly mutating and has undergone a major design change throughout 2019 [23], this section addresses its structure at the time of writing of this paper.

A. Codebase Structure

The PyTorch codebase has over 8000 files with over 2 million lines of code. Of this, 60% is C++ code and 35% Python code, the remaining 5% being comprised of C++ CUDA (gpu specific code) and mostly deprecated C code. [23]. The codebase is divided in 3 main directories that encompass all the core functionalities of the framework:

- **c10**: The core library files that are used in every other part of the code, common to both server and mobile;
- **aten**: The C++ tensor library, that is, where all the tensor definitions and operators are implemented;
- **torch**: The translation from C++ code to the Python frontend, with both C++ and Python files.

In order to support Posit, files from all of the 3 directories must be altered: *c10* to define the new data type for Posits; *aten* to define the operators on tensors with posit data and *torch* to expose posit tensors in the Python frontend. From these, aten is the most relevant, since it is where every function actuating on posits will be defined. This directory is further subdivided in subdirectories comprising deprecated code (previous to the major refactor undergone in 2019), and the more recent code under *aten/src/ATen*. Within *aten/src/ATen* there are several subdirectories (for CPU operators, cuda operators, operators that use external libraries, etc.) making the code organized in a way that makes it intuitive to navigate.

Besides those 3 core directories, there are others that address the different stages of development of the codebase: a directory with the scripts for the build process; a directory with the unit tests of the Python frontend and another dedicated to the tool used for integration of new code.

B. Tensor Implementation

As mentioned in section III-A, PyTorch uses tensors as the core data structure to represent all data within NNs. A tensor's data is stored in memory, usually with each row contiguous to the previous. This way, the tensor's metadata should hold not only the size of each dimension, but also the stride associated with it. The stride is what the logical index should be multiplied with in order to get to the physical index where the element is stored. Moreover, there is an offset, representing the position in memory where the count starts. Equation (5) provides the translation between the logical indexes (i) and strides (s) and the physical position in memory (mp) for a tensor with n dimensions.

$$\text{mp} = \text{i}[n-1] \times \text{s}[n-1] + \cdots + \text{i}[0] \times \text{s}[0] + \text{offset} \quad (5)$$

From here, it follows that for each tensor some metadata needs to be stored, namely the sizes and strides. Besides these, three main parameters are stored as metadata:

- **device**: represents where the tensor's physical memory is stored: CPU, GPU, TPU, etc.;

- **layout**: describes how to logical interpret the physical data. The most common is a strided tensor, but PyTorch supports other types of memory configurations;
- **dtype**: corresponds to the type of the tensor's data: 32-bit floating-point, 64-bit integer, brain floating-point, etc..

However, the storing of this metadata is divided according to whether it is metadata corresponding to the logical interpretation of a tensor or to the physical storage of data. This way, tensors have a level of indirection: the Tensor structure that holds the metadata relative to the logical interpretation (layout, sizes, strides, offset) and the Storage structure that records the device and data type of the tensor as well as a pointer to the raw data.

With this level of indirection, PyTorch effectively supports different views of a Tensor, that is, different logical interpretations of the same underlying data.

The number of possible types of tensors comes from the cartesian product of the 3 parameters presented above: *device*, *layout* and *dtype*. In order to extend tensors to support Posit, it is enough to add another *dtype* representing Posit. This is detailed in section V.

Tensor Iterator Utility: A lot of operations involve iterating over all the elements of a tensor (or more than one tensor). This is the case not only in all point-wise operations, where a certain function is applied to all the elements of the tensor individually, but also for operations involving two tensors, (e.g. addition). In order to facilitate these operations, the *TensorIterator* API offers a standardized way to iterate over elements of a tensor, automatically parallelizing operations, while abstracting device and data type details.

A *TensorIterator* should first be built through *TensorIteratorConfig*, where the input and output tensors are specified, along with more information such as whether there should be checks on the data types or dimensions, among other utilities. Once built, it can be iterated over using the *for_each* function, that can loop over 1 or 2 dimensions at once. There are also built-in kernels that take a *TensorIterator* as argument and apply an operation over each element. Listing 2 provides an example of building a *TensorIterator* with two inputs and using the built-in *cpu_kernel* to perform addition. This implementation works for all data types (as long as the + operator is overloaded), since the *dtype* is abstracted in the *TensorIterator*.

```
at::TensorIteratorConfig iter_config;
iter_config
  .add_output(c)
  .add_input(a)
  .add_input(b);

auto iter = iter_config.build();
auto datatype = iter.dtype()
at::native::cpu_kernel(iter, [] (datatype a,
    datatype b) -> datatype {
  return a + b;
});
```

Listing 2. Example of usage of the *TensorIterator* utility

C. Dispatcher

Operators to actuate on tensors need to be different depending on the type of the tensor. The implementation of matrix multiplication, for instance, is different if tensors are stored on the CPU or the GPU, if the data type is integers or complex reals, etc.. For this purpose, PyTorch has an abstraction layer, working similarly to a virtual table, that addresses this issue, called **dispatcher**.

The dispatcher is implemented through a bitset, called *DispatchKeySet*, where each relevant metadata contributes with a key to the set. These keys are ordered in decreasing order of priority. A kernel is called for the first key of the set, adding that key to the exclude set. In the next pass through the dispatcher, the next key is used to identify the kernel to be called. With this logic, the same dispatch key set can be used for multiple calls, which is useful for PyTorch's automatic differentiation (autograd), for example.

Figure 6 provides an example where an operation is to be performed on 2 tensors, one on the CPU and the other on the GPU. The key for autograd is excluded from the bitset - since it has already been handled in a previous dispatch - and the backend select key is the one used to call the kernel that will address this operation.

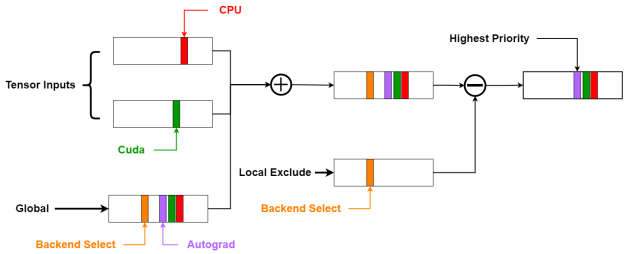

Fig. 6. Illustration of a dispatcher call.

The main advantage of the dispatcher is its decentralized nature. Once the dispatch keys are defined, all the kernels corresponding to a given operator can be defined independently, without the need for a centralized if statement.

D. Kernels

For each dispatch key, there should be a kernel that implements the intended operation. In order to register these kernels, PyTorch provides the Registration API, which provides 4 endpoints in the form of C++ macros:

- **DECLARE_DISPATCH**: where the function signature is associated with the name of the dispatch registry;
- **DEFINE_DISPATCH**: defines a function to be the dispatch registry;
- **TORCH_IMPL_FUNC**: calls the dispatch registry function through the function that is exposed to higher-level APIs;
- **REGISTER_DISPATCH**: associates the dispatch registry function with the kernel that implements the operation.

Even though the API is not intuitive at first and implies changing multiple files, it provides flexibility and speed when calling dispatched functions [24]. Listing 3 provides an example for the *add* function.

```
// file aten/src/ATen/native/BinaryOps.h
DECLARE_DISPATCH(void(*)(TensorIteratorBase&, const
    Scalar& alpha), add_stub);

// file aten/src/ATen/native/BinaryOps.cpp
DEFINE_DISPATCH(add_stub);

TORCH_IMPL_FUNC(add_out) (
  const Tensor& self, const Tensor& other, const
    Scalar& alpha, const Tensor& result
) {
  add_stub(device_type(), *this, alpha);
}

// file aten/src/ATen/native/cpu/BinaryOpsKernel.cpp
void add_kernel(TensorIteratorBase& iter, const
    Scalar& alpha_scalar) {
    //implementation of the add function on CPU
}

REGISTER_DISPATCH(add_stub, &add_kernel);

// file aten/src/ATen/native/cuda/BinaryAddSubKernel
    .cu
void add_kernel_cuda(TensorIteratorBase& iter, const
     Scalar& alpha_scalar) {
    //implementation of the add function on gpu
}

REGISTER_DISPATCH(add_stub, &add_kernel_cuda);
```

Listing 3. Registration API to associate function calls to kernel implementations

In this example, *add_stub* is the dispatch registry function, *add_out* is the function that is exposed to higher level API's, *add_kernel* is the CPU implementation of the add operation and *add_kernel_cuda* the gpu implementation.

V. Posit Integration in PyTorch

With an understanding of the main structure of the backend as exposed in the previous section, adding support for posits must make use of these tools. This section presents the different stages of this work, starting from adding the posit data type to Pytorch up to the support of the relevant operations on posit tensors and the exposure in the Python frontend.

A. Posit Data Type

The adopted strategy to integrate posits was to add them as one of the built-in data types supported by PyTorch. This approach has the advantage of exposing posits in the frontend as any other data type, avoiding the need for special representations. To emulate the operations with Posits, the Universal library was the one chosen. Hence, in order for the posit manipulation functions to be available in the namespaces of PyTorch, two files were used as translation: *c10/util/Posit16es2.h* for the posit declaration and *c10/util/Posit16es2-math.h* for the math functions. A snippet of the code in these two files is presented in listing 4.

```
// file c10/util/Posit16es2.h
#include <universal/number/posit/posit.hpp>
namespace c10 {
  using posit16es2 = sw::universal::posit<16,2>;
}

// file c10/util/Posit16es2-math.h
namespace std {
    inline c10::posit16es2 tanh(c10::posit16es2 a) {
      return sw::universal::tanh(a);
    }
}
```

Listing 4. Conversions from the Universal library functions to those in PyTorch's namespaces

PyTorch implements all the data types in the *c10/core* directory (since this is the common directory to the whole project) through the macro *AT_FORALL_SCALAR_TYPES _WITH_COMPLEX_AND_QINTS*. In it, a data type is associated with an alias (with a number associated), to be used throughout the codebase for comparison purposes. This file also contains a macro that encompasses all the data types belonging to a given set (integers, floats, complex, etc.), along with a helper function to determine if a type belongs to a given set. These were extended for posits with $nbits = 16$ and $es = 2$., as listing 5 shows.

```
#include <c10/util/Posit16es2.h>
// file c10/util/Posit16es2.h

// file c10/core/ScalarType.h
#define
    AT_FORALL_SCALAR_TYPES_WITH_COMPLEX_AND_QINTS(_)
     \
  _(uint8_t, Byte) /* 0 */  \
  _(int8_t, Char) /* 1 */   \
  _(int16_t, Short) /* 2 */ \
  _(int, Int) /* 3 */       \
  _(int64_t, Long) /* 4 */  \
  _(at::Half, Half) /* 5 */ \
  _(float, Float) /* 6 */   \
  _(double, Double) /* 7 */ \
  _(c10::complex<c10::Half>, ComplexHalf) /* 8 */
        \
  _(c10::complex<float>, ComplexFloat) /* 9 */
        \
  _(c10::complex<double>, ComplexDouble) /* 10 */
        \
  _(bool, Bool) /* 11 */                \
  _(c10::qint8, QInt8) /* 12 */         \
  _(c10::quint8, QUInt8) /* 13 */       \
  _(c10::qint32, QInt32) /* 14 */       \
  _(at::BFloat16, BFloat16) /* 15 */    \
  _(c10::quint4x2, QUInt4x2) /* 16 */   \
  _(c10::posit16es2, Posit16es2) /* 17 */

#define AT_FORALL_POSIT_TYPES(_)       \
  _(c10::posit16es2, Posit16es2)

static inline bool isPositType(ScalarType t) {
  return t == ScalarType::Posit16es2;
}
```

Listing 5. Supported data types in PyTorch, including posit(16,2)

Since a macro is a pre-processor directive that is only expanded at compile time, it is not possible to define a generic data type for Posits. Despite the flexibility provided by the templates of the Universal library to implement Posits, these must be instantiated in the macro at compile time.

As a consequence, only a posit datatype with $nbits = 16$ and $es = 2$ was added, since it is the most used to re-

place 32-bit floating-point numbers for NN training [15] [16]. Nonetheless, in order to support more posit configurations, it is enough to add them to the posit macros exposed throughout this section.

B. Dispatcher for Posit Types

As presented in section IV-C, PyTorch implements a cascading dispatcher system for operations on tensors. As such, it was necessary to define a dispatcher for the posit data types.

This dispatcher receives 3 arguments:

- **TYPE**: the type of the tensor;
- **NAME**: the name of the function being called;
- a C++ lambda function for the operation in question.

The dispatcher stores the type of the data, gets its alias and calls the passed function with the correct data type. Listing 6 shows the code for the newly created posit(16,2) datatype. To support more posit types, they should be added to the switch statement.

The amount of indirection levels is due to PyTorch's structure, where the usage of macros is maximised to increase the execution speed [25]. This comes at the cost of readability of the code and increase in complexity and build times when altering these core files.

```
#define AT_DISPATCH_POSIT_TYPES(TYPE, NAME, ...)
          \
  [&] {                           \
    const auto& the_type = TYPE;                \

    at::ScalarType _st = ::detail::scalar_type(
    the_type);          \
    RECORD_KERNEL_FUNCTION_DTYPE(NAME, _st);
          \
    switch (_st) {
                                          \
      AT_PRIVATE_CASE_TYPE(NAME, at::ScalarType::
    Posit16es2, c10::posit16es2, __VA_ARGS__)       \
      default:                             \
          AT_ERROR(#NAME, " not implemented for '",
    toString(TYPE), "'");        \
    }                          \
  }()
```

Listing 6. Supported data types in PyTorch, including posit(16,2)

The next section will present the operations that were extended to support Posit. In all of them, this dispatcher is used, given the structure of the indirection levels of PyTorch.

C. NN Operators for Posits

PyTorch contains thousands of operators, and supporting all of them for a new data type results in a workload above the scope of this work. This is evidenced by the two years taken to have full support for complex tensors in PyTorch [23]. As a result, the followed approach consisted of supporting the subset of operators needed to train a NN solely with Posits.

The binding from a C++ operator implementation and the corresponding Python frontend functions is mostly done by auto-generated code. Adding this to the size of the codebase, it is not recommended to navigate the stack trace to find the C++ function that needs to be extended for the new data type [25]. The most efficient way to locate the relevant function is to call it, record the error message and search through the codebase for that message. Listing 7 provides an example for the MSE loss function, where the error message *"mse_cpu" not implemented for 'Posit16es2'* can be searched through the codebase to find the kernel that implements this operation.

```
>>> loss = nn.MSELoss()
>>> input = torch.tensor ([[1.2 , 0.4, 2.3], [4.3,
    6,  -5.1]], dtype=torch.posit16es2)
>>> target = torch.tensor ([[1.2 ,  -1.2, 1], [4,
    7.2,  -5.4]], dtype=torch.posit16es2)
>>> output = loss(input, target)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/afonsoluz/pytorch/torch/nn/modules/
    module.py", line 1102, in _call_impl
    return forward_call(*input, **kwargs)
  File "/home/afonsoluz/pytorch/torch/nn/modules/
    loss.py", line 520, in forward
    return F.mse_loss(input, target, reduction=self.
    reduction)
  File "/home/afonsoluz/pytorch/torch/nn/functional.
    py", line 3112, in mse_loss
    return torch._C._nn.mse_loss(expanded_input,
    expanded_target, _Reduction.get_enum(reduction))
RuntimeError: "mse_cpu" not implemented for '
    Posit16es2'
```

Listing 7. Calling the MSE loss function before it was supported for Posits

Once the kernel is located, extending it for posits can either consist of simply using the code that is being used for other dtypes or writing a specific kernel. The second happens when the implementation for other data types is making use of hardware optimizations not available for Posits, such as vectorization through the AVX2 instruction set.

Listing 8 provides an example of supporting the average pooling kernel through the same operation that was being done for the other data types. Since the kernel *cpu_avg_pool* only performs operations that are supported by the Universal library (and hence work with the posit datatype), it is enough to add it to the dispatcher that was already being used for the other data types. This is done by adding the alias for the posit(16,2) type *at::ScalarType::Posit16es2* as an argument to the dispatcher.

```
AT_DISPATCH_FLOATING_TYPES_AND2(at::ScalarType::Long
    , at::ScalarType::Posit16es2, input.scalar_type
    (), "avg_pool2d", [&] {
        cpu_avg_pool<scalar_t>(output, input, kW, kH
    , dW, dH, padW, padH, count_include_pad,
    divisor_override);
      });
```

Listing 8. Extending the average pooling operator for posits through the same kernel as for other data types

Listing 9 provides an example of supporting the addition with scaling operation, where a different kernel should be used. Here, the *TensorIterator* API is explored, by using the *cpu_kernel*, which takes an iterator with two input tensors and applies the lambda function passed to them. This is an example of the case where for the other data types an hardware optimization was being used.

```
if (isPositType(dtype)) {
    AT_DISPATCH_POSIT_TYPES(dtype, "addcmul_cpu_out"
    , [&] {
      scalar_t scalar_val = value.to<scalar_t>();
      cpu_kernel(
```

```
        iter,
        [=](scalar_t self_val, scalar_t t1_val,
    scalar_t t2_val) -> scalar_t {
          return self_val + scalar_val * t1_val *
    t2_val;
        }
      );
    });
}
```

Listing 9. Supporting addition with scaling by calling a separate kernel for Posits

Table I lists all the operators whose support for posits was implemented in the scope of this work. It presents the name of the operator, its role and whether the extension was using the existing kernel for other data types or through a custom kernel for Posits. In the latter case, the *TensorIterator* API was used, except for the softmax functions. Operators that call others on this table, such as the cross entropy loss function (combination of the log softmax function and the NLL loss function), are automatically supported. Table II presents the frontend functions resulting from the operators implemented.

TABLE I
OPERATORS THAT WERE EXTENDED TO SUPPORT POSITS, ALONG WITH THEIR ROLE AND THE TYPE OF EXTENSION.

Name of the operator	Role	Extension type
scalar_fill	Tensor Implementation	Same Kernel
local_scalar_dense	Tensor Implementation	Same Kernel
copy	Tensor Implementation	Specific Kernel
uniform	Tensor Implementation	Same Kernel
fill	Tensor Implementation	Specific Kernel
add	Addition	Specific Kernel
sub	Subtraction	Specific Kernel
sum	Reduction addition	Specific Kernel
max	Maximum Tensor Value	Specific Kernel
sqrt	Square Root	Specific Kernel
add_mm	Matrix Multiplication	Same Kernel
addcmul	Multiplication and Addition	Specific Kernel
addcdiv	Division and Addition	Specific Kernel
unfolded2d_acc	Convolution	Same Kernel
avg_pool2d	Average Pooling	Same Kernel
avg_pool2d_backward	Gradient Computation	Same Kernel
tanh	Activation function	Specific Kernel
tanh_backward	Gradient Computation	Specific Kernel
softmax	Activation Function	Specific Kernel
softmax_backward	Gradient Computation	Specific Kernel
log_softmax	Activation Function	Specific Kernel
log_softmax_backward	Gradient Computation	Specific Kernel
nll_loss	Loss Function	Same Kernel
nll_loss_backward	Gradient Computation	Same Kernel
mse	Loss Function	Specific Kernel

VI. EXPERIMENTAL EVALUATION

To validate the implementation of the developed operators for posits within PyTorch, the CNN LeNet-5 was trained with posits with both the MNIST dataset and the more complex Fashion MNIST dataset. Given the software emulated nature of the posit library, computations take longer than with floats. Thus, only a portion of the samples of the datasets were used to have a manageable total training time. A block diagram representing the layers and parameters of the network is presented in Figure 7.

TABLE II
SUPPORTED NN FUNCTIONS AND LAYERS.

Function Name	Function Type
Convolution	Layer
Average Pooling	Layer
Maximum Pooling	Layer
Fully Connected	Layer
Dropout	Layer
TanH	Activation Function
ReLU	Activation Function
Softmax	Activation Function
Mean Squared Error (MSE)	Loss Function
Cross Entropy Loss	Loss Function
Adam	Optimizer

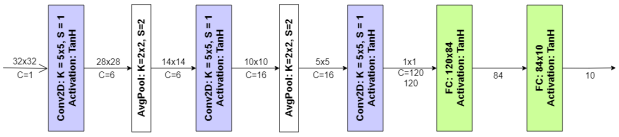

Fig. 7. Block Diagram representing the architecture of the evaluated LeNet-5. K stands for kernel, S for stride and FC for fully connected.

In addition to the architecture of the network, it was necessary to define the loss function and the optimizer algorithm. The chosen loss function was the Cross Entropy Loss.

The training was conducted in a system with an Intel Xeon E312xx CPU with 8 cores, operating at 2.4GHz and with 32 GB of RAM.

Figure 8 shows the evolution of the accuracy after each epoch for 32-bit floats and 16-bit posits on the MNIST dataset.

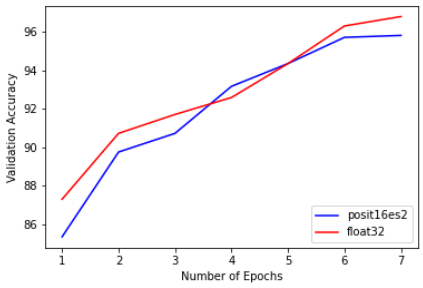

Fig. 8. Comparison of the obtained accuracy of LeNet-5 training with floats and posits on the MNIST dataset.

As it can be observed, the obtained accuracy with 16-bit posits is similar to that of 32-bit floats, with a difference of around 1% after 7 epochs. This is in line with the results obtained in other works that evaluate training with small precision posits [15] [16].

Figure 9 shows the evolution of the obtained accuracy after the execution of the same experiment on the FashionMNIST dataset.

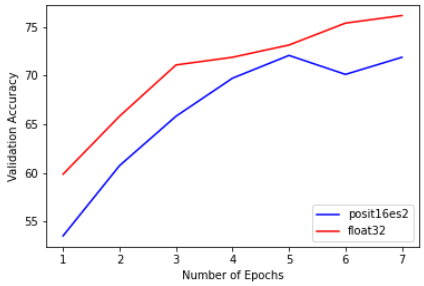

Fig. 9. Comparison of the accuracy of LeNet-5 training with 32-bit floats and 16-bit posits on the FashionMNIST dataset.

The FashionMNIST dataset is more complex, as it consists of clothes' items rather than digits. This leads to poorer performances, which is evidenced by the lower accuracy attained with 32-bit floats after 7 epochs of around 76%, contrasting with the 97% obtained when evaluating MNIST. Nonetheless, with posits with $nbits = 16$ and $es = 2$ the attained accuracy of around 72% is 4% smaller than that of floats. This gap is likely due to the reduced number of samples used for training, which prevents the network from generalizing adequately.

VII. Conclusion

Up until now, to perform experiments with posits in the context of DL, researchers had to build their own customized functions and operators. As a result of this work, the main NN operators and functions are supported in PyTorch, the most popular DL framework for research. The code, along with documentation on the process of supporting posits, was made available on a public GitHub repository [26]. This means further contributions can be made to extend the present work. Moreover, the modular nature of this work allows for future hardware implementations of the posit arithmetic to be integrated and tested in PyTorch.

References

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[2] A. Mosavi, S. Ardabili, and A. Varkonyi-Koczy, "List of deep learning models," in *Engineering for Sustainable Future*. Springer International Publishing, 01 2020, pp. 202–214, doi:10.1007/978-3-030-36841-8_20.

[3] N. C. Thompson, K. Greenewald, K. Lee, and G. F. Manso, "The computational limits of deep learning," 2020, arXiv:2007.05558.

[4] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.

[5] TensorFlow, "tensorflow/tensorflow: An open source machine learning framework for everyone - github, v2.4.3," 2021, accessed on 06/10/2021. [Online]. Available: https://github.com/tensorflow/tensorflow

[6] H. He, "The state of machine learning frameworks in 2019," 2019, accessed on 06/10/2021. [Online]. Available: https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/

[7] J. Johnson, "Rethinking floating point for deep learning," *CoRR*, vol. abs/1811.01721, 2018.

[8] A. Mcgovern and K. Wagstaff, "Machine learning in space: Extending our reach," *Machine Learning*, vol. 84, pp. 335–340, 09 2011.

[9] S. Wu, G. Li, F. Chen, and L. Shi, "Training and inference with integers in deep neural networks," 2018, arXiv:1802.04680.

[10] S. H. Fatemi Langroudi, T. Pandit, and D. Kudithipudi, "Deep learning inference on embedded devices: Fixed-point vs posit," in *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*, 2018, pp. 19–23, doi:10.1109/EMC2.2018.00012.

[11] J. L. Gustafson and I. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, 2017.

[12] H. F. Langroudi, Z. Carmichael, J. L. Gustafson, and D. Kudithipudi, "Positnn framework: Tapered precision deep learning inference for the edge," in *2019 IEEE Space Computing Conference (SCC)*, 2019, pp. 53–59.

[13] Z. Carmichael, S. H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi, "Deep positron: A deep neural network using the posit number system," *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1421–1426, 2019.

[14] J. Lu, C. Fang, M. Xu, J. Lin, and Z. Wang, "Evaluations on deep neural networks training using posit number system," *IEEE Transactions on Computers*, vol. 70, no. 2, pp. 174–187, 2021.

[15] G. E. C. Raposo, P. Tomás, and N. Roma, "Positnn: Training deep neural networks with mixed low-precision posit," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP2021)*. IEEE, June 2021.

[16] R. Murillo, A. A. Del Barrio, and G. Botella, "Deep pensieve: A deep learning framework based on the posit number system," *Digital Signal Processing*, vol. 102, p. 102762, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S105120042030107X

[17] J. L. Gustafson, "Beyond floating point: Next generation computer arithmetic. stanford seminar," 2016, accessed on 20/09/2021. [Online]. Available: https://www.youtube.com/watch?v=aP0Y1uAA-2Y&ab_channel=stanfordonline

[18] ——, *The End of Error, Unum Computing*, 2nd ed. Chapman and Hall/CRC, 2015, iSBN: 9781482239867.

[19] W. Tichy, "Unums 2.0: An interview with john l. gustafson," *Ubiquity*, September 2016.

[20] L. Bottou and Y. LeCun, "Large scale online learning," in *Advances in Neural Information Processing Systems*, vol. 16. MIT Press, 2004.

[21] S. Raschka, "Fitting a model via closed-form equations vs. gradient descent vs stochastic gradient descent vs mini-batch learning. what is the difference?" 2021, accessed on 27/09/2021. [Online]. Available: https://sebastianraschka.com/faq/docs/closed-form-vs-gd.htmlfitting-a-model-via-closed-form-equations-vs-gradient-descent-vs

[22] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533 – 536, 1986.

[23] PyTorch, "pytorch/pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration - github, v1.9.1," 2021, accessed on 14/10/2021. [Online]. Available: https://github.com/pytorch/pytorch

[24] E. Yang, "Let's talk about the pytorch dispatcher," 2020, accessed on 19/10/2021. [Online]. Available: http://blog.ezyang.com/2020/09/lets-talk-about-the-pytorch-dispatcher/

[25] ——, "Pytorch internals," 2019, accessed on 18/10/2021. [Online]. Available: http://blog.ezyang.com/2019/05/pytorch-internals/

[26] A. Luz, "Afonso-2403/pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration - posit support," 2021, accessed on 25/10/2021. [Online]. Available: https://github.com/Afonso-2403/pytorch