Google Summer of Code 2025

CODING PROJECT PROPOSAL

Support for Logarithmic Number Systems in a Deep-Learning Framework.

Name: Alex Krentz

Potential Mentors: Mark Arnold, Ed Chester, and Pradeeban Kathiravelu

Email: alexkrentz2@gmail.com

Github: https://github.com/akrentz6/

Code Repository: https://github.com/akrentz6/xlns-gsoc-application/

Project Length: 350 hours

1 Abstract

This project aims to integrate Logarithmic Number Systems (LNS) into a deep-learning framework, offering an alternative to traditional floating-point (FP) arithmetic. LNS simplifies multiplication and division, potentially reducing power consumption and improving efficiency in applications like deep learning, where approximate results are acceptable. However, current deep-learning frameworks, such as PyTorch and TensorFlow, are hardcoded to use FP operations, limiting the adoption of LNS.

We seek to bridge that gap by developing a system that supports LNS arithmetic while maintaining compatibility with existing frameworks. By creating custom layers, optimizers, and benchmarking tools, the project will enable the simulation of LNS arithmetic in FP hardware for neural network training and inference. The expected outcome is a robust, open-source solution to make research and exploration of LNS in deep-learning applications easier, paving the way for future hardware implementations.

Contents

2 Resume

I'm an undergraduate Mathematics student at University College London, graduating in 2027. My studies have given me a strong grasp in many fields that will be useful in this project. My coursework in numerical analysis has equipped me with the tools to assess numerical stability, examine how rounding errors propagate, and develop accurate function approximations. I've also taken several optimisation and linear algebra modules which is foundational for a strong understanding of deep learning. In addition, my modules in real analysis and discrete math will help when analysing the properties of functions used in LNS arithmetic and in optimising the internal representations of LNS values.

I have been very interested in machine learning for a while, having written a research paper last year comparing first and second order optimisation methods for training neural networks. This required a deep understanding of PyTorch and I implemented various neural network architectures as well as my own optimizers, with a benchmarking suite to analyse convergence rates, accuracy and training speed. In addition, I have worked on several projects involving machine learning, most recently a chess engine with a neural network backend to evaluate positions (C++) and an AI trading bot that analysed time series data of stock prices and other metrics to predict future trends (Python).

These experiences give me an excellent foundation for this project, demonstrating my confidence in the math required to implement LNS arithmetic in neural networks (especially in backpropagation). My knowledge of PyTorch will be invaluable in adapting existing frameworks to incorporate LNS-based computations, and handle the intricacies involved in a technical project like this.

I have also built a diverse portfolio of coding projects that demonstrate my technical skills. For instance, I recently developed an open source Python library, Redshot, which automates WhatsApp web workflows by managing, sending, and receiving messages. Additionally, I created an economy Minecraft plugin in Java that has amassed over 50,000 downloads. These projects have not only reinforced my ability to write clear, well-documented, and robust code, but also demonstrate my ability to deliver practical solutions at scale. Such experience will be very useful in ensuring that this project's codebase is both functional and maintainable.

3 Coding Challenges

As part of the application, we were asked to complete three coding challenges. These were intended to illustrate the wide array of issues to be tackled in this project and would help demonstrate which library is more suited to injecting LNS arithmetic. My code for each challenge can be found in the appendix at the end of this document or at https://github.com/akrentz6/xlns-gsoc-application/.

3.1 Challenge 1: XLNS Incompatibility with TensorFlow/PyTorch

In the first challenge, we wanted to show the incompatibility of xlns with Tensorflow and PyTorch. Indeed, from the code output in the appendix, you can see that passing an `xlns.xlnsnp` array as the argument to a `tf.Tensor` or `torch.Tensor` raises an error.

Unlike with numpy, we are unable to overload the TensorFlow/PyTorch operators and so we are forced to convert each number's `int64` internal LNS representation back to its standard floating point representation before performing any operations on it. The only other option we have is to pass the number's internal representation to Tensorflow/PyTorch and do operations on that (as we will see in challenge 3).

3.2 Challenge 2: LNS vs. FP convergence comparison

The next challenge asked us to implement a fully connected network in TensorFlow and PyTorch, trained on the MNIST dataset. Each model featured 28×28 input nodes, a hidden layer with 100 neurons using a ReLU activation function, and a 10 neuron output using the softmax function. Each layer includes an extra bias weight, consistent with the original design.

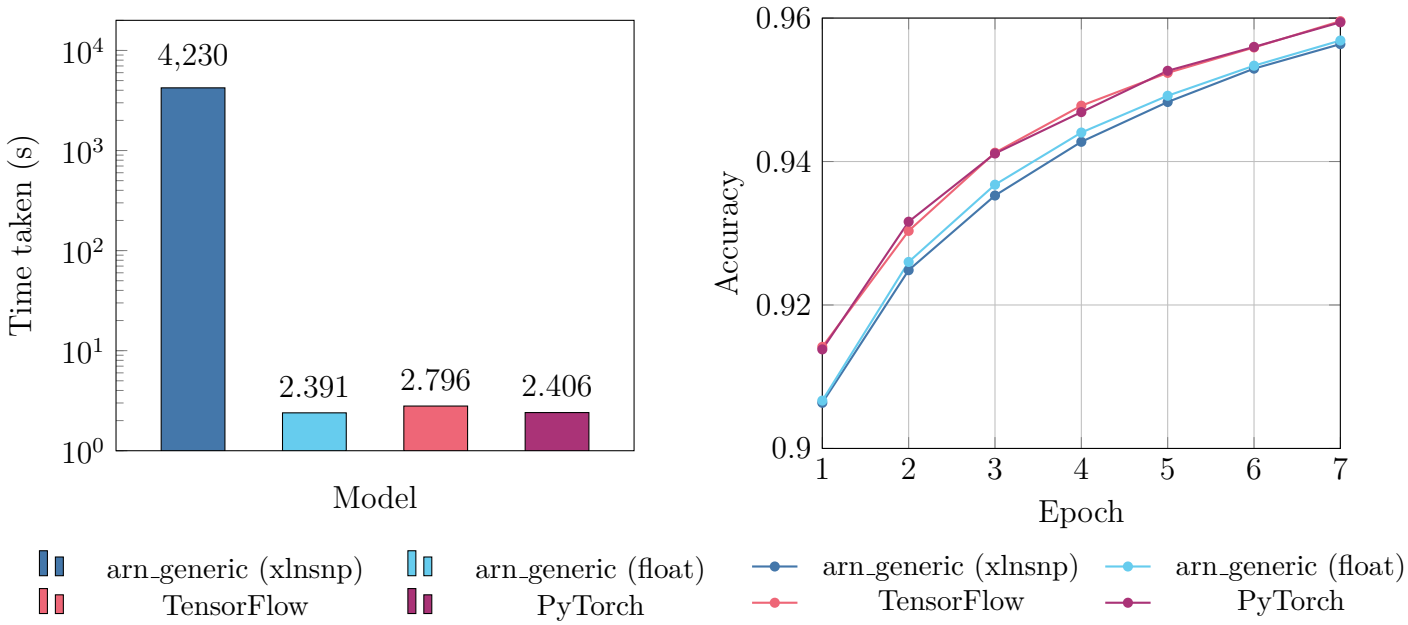
We were then tasked with comparing the convergence of these models with xlns' example program `arn_generic.py` which implements a network of the same structure but employs manual differentiation. For our comparison, the program was run separately using standard FP arithmetic and then with xlns' `xlns.xlnsnp` class which uses LNS arithmetic. In all tests, models were run ten times consecutively for 7 epochs each with a batch size of 128 and a learning rate of 0.1. Weights were randomly generated from a normal distribution each iteration.

Model	Ep. 1	Ep. 2	Ep. 3	Ep. 4	Ep. 5	Ep. 6	Ep. 7
arn_generic (xlnsnp)	0.9064	0.9249	0.9353	0.9428	0.9483	0.9530	0.9564
arn_generic (float)	0.9067	0.9260	0.9368	0.9440	0.9492	0.9534	0.9569
TensorFlow	0.9142	0.9303	0.9412	0.9479	0.9524	0.9559	0.9596
PyTorch	0.9138	0.9316	0.9411	0.9469	0.9527	0.9560	0.9594

Table 1: The average accuracy (to 4 s.f.) of each model after a given number of epochs.

Model	Time taken (s)
arn_generic (xlnsnp)	4230
arn_generic (float)	2.391
TensorFlow	2.796
PyTorch	2.406

Table 2: The average time taken (to 4 s.f.) to train each model.



The results indicate that while all four implementations converged comparably in terms of accuracy, differences in execution time and slight variations in per-epoch accuracy underscore important trade-offs in implementation design. Notably, the two `arn_generic.py` models demonstrated nearly identical convergence behavior. This similarity reinforces the central hypothesis that LNS can achieve comparable training and inference accuracy to traditional floating-point methods, even though the LNS implementation (arn_generic xlnsnp) incurred a considerably longer runtime (4230s) compared to the floating-point version (2.391s). Given that the xlns library is not optimized for speed, the substantial overhead is anticipated and acceptable for the purposes of evaluating precision and feasibility.

Turning to the higher-level frameworks, both TensorFlow and PyTorch exhibited fast runtimes (2.796s and 2.406s respectively), with the PyTorch implementation running very close to the arn_generic float implementation. The slight delay in TensorFlow's runtime might be attributed to its approach to organizing and optimizing its operations before actual execution. In simpler terms, TensorFlow spends a bit of extra time setting up a detailed plan (or "map") of what it needs to do, which sometimes adds a small delay. Although these extra steps don't significantly affect the overall learning process, they do show that even small design choices in a framework can have an impact on performance.

The roughly 0.5% improvement in accuracy at each epoch with TensorFlow and PyTorch

may be partly explained by the fact that these frameworks used built-in stochastic gradient descent (SGD) optimizers. In contrast, the arn_generic models relied on manual backpropagation. This difference in training methods could contribute to the slight edge in accuracy, as Tensorflow and PyTorch's optimizers benefit from extensive tuning and refinement over time.

Overall, the experiment demonstrates that LNS-based computations are viable for achieving high-accuracy outcomes in machine learning, at least at the level of software simulation. Although the software-based LNS implementation is not competitive in terms of speed, its convergence behavior mirrors that of more conventional floating-point methods. This finding supports further exploration into LNS, especially in contexts where dedicated hardware could offset the performance penalties seen in a software environment.

3.3 Challenge 3: Tensor-Based LNS Addition Prototype

The final challenge asked us to implement a toy version of LNS arithmetic for addition. Instead of overloading numpy functions as in the xlns library, the goal was to use tensor operations from PyTorch and TensorFlow to perform the same kind of arithmetic done in the provided example.

When working with frameworks such as these, it's important to use their native functions rather than NumPy's because these frameworks are built to operate on tensors, providing GPU acceleration, parallel processing, and ensuring that all parts of the computation are compatible with the rest of the framework. This challenge reveals the difficulties with injecting LNS arithmetic into a machine learning framework based in FP arithmetic. The following is a derivation of the formulae used to compute addition.

Suppose we want to represent a non-zero real number x in the LNS using the following scheme. Let B denote the underlying base and let

$$x = (-1)^{s_x} \cdot B^X \tag{1}$$

where

$$X = \log_B|x| \quad \text{and} \quad s_x = \begin{cases} 0, & \text{if } x > 0, \\ 1, & \text{if } x < 0. \end{cases}$$

For another non-zero real number y, let

$$y = (-1)^{s_y} \cdot B^Y \tag{2}$$

where Y and s_y are defined analogously.

In a fixed precision implementation, we can "pack" the logarithm and the sign into one `int64` value. With, for example, p bits reserved for the logarithm, the internal representation is chosen to be

$$x' = (\text{round}(X) \ll 1) + s_x \tag{3}$$

where we left-shift by one (i.e., multiply by 2) to make room for the sign bit in the least significant bit (LSB). It's useful to use the LSB here as Python integers are of variable size so don't have a most significant bit that acts as the sign bit (as in languages like C or Java).

Addition in the LNS Domain.

Given x and y as in (1)–(3), we wish to compute a non-zero

$$z = x + y \tag{4}$$

and represent it in the same LNS format. Without loss of generality assume that $|x| > |y|$, so that $X > Y$. Then,

$$\begin{aligned} x + y &= (-1)^{s_x} \cdot B^X + (-1)^{s_y} \cdot B^Y \\ &= B^X \left((-1)^{s_x} + (-1)^{s_y} \cdot B^{Y-X} \right). \end{aligned}$$

The behavior of the summation depends on whether the signs s_x and s_y are the same or different, so we consider two cases:

Case 1: Same Signs ($s_x = s_y$)

In this case,

$$(-1)^{s_x} + (-1)^{s_y} B^{Y-X} = (-1)^{s_x} \left(1 + B^{Y-X} \right)$$

Thus,

$$x + y = (-1)^{s_x} B^X \left[1 + B^{Y-X} \right]$$

and taking logarithms yields

$$\log_B |x + y| = X + \log_B \left(1 + B^{Y-X} \right) \tag{5}$$

In addition, it is clear that $s_{x+y} = s_x = s_y$ as the sum of two positive numbers is positive and the sum of two negative numbers is negative.

Case 2: Opposite Signs ($s_x \neq s_y$)

For opposite signs, we have

$$x + y = (-1)^{s_x} B^X \left(1 - B^{Y-X} \right)$$

Hence,

$$\log_B |x + y| = X + \log_B \left| 1 - B^{Y-X} \right| \tag{6}$$

provided that $B^{Y-X} < 1$ (which is automatically true since $X \geq Y$, so that $Y - X \leq 0$). The overall sign of the sum is determined by the number with the greater magnitude, i.e. x, so in this case $s_{x+y} = s_x$.

A Uniform Treatment of Both Cases.

To write both cases in a uniform way, define

$$d = Y - X \quad (\text{so that } d < 0, \text{ i.e. } X > Y)$$

and introduce a binary flag

$$s_{x,y} = \begin{cases} 0, & \text{if } s_x = s_y, \\ 1, & \text{if } s_x \neq s_y. \end{cases}$$

Then a unified formula for the logarithm of the absolute value of the sum is

$$\log_B |x + y| = X + \delta, \quad \text{with } \delta = \log_B \left|1 - 2s_{x,y} + B^d\right| \tag{7}$$

and a unified formula for the sign of the sum is

$$s_{x+y} = s_x \tag{8}$$

Finally, we can express the internal representation of $x + y$ as in (3) by

$$\begin{aligned} (x+y)' &= (\text{round}\,(X + \delta) \ll 1) + s_x \\ &= ((\text{round}\,(X) + \text{round}\,(\delta)) \ll 1) + s_x \\ &= (\text{round}\,(X) \ll 1) + s_x + (\text{round}\,(\delta) \ll 1) \\ &= x' + (\text{round}\,(\delta) \ll 1) \end{aligned} \tag{9}$$

where we are able to make the approximation that $\text{round}\,(X + \delta) = \text{round}\,(X) + \text{round}\,(\delta)$ since our internal representation of x already fixes X as an integer, so $\text{round}\,(X + \delta)$ reduces to simply adding the integer X to $\text{round}\,(\delta)$.

Handling the Zero Case

As x approaches 0, its logarithm tends towards $-\infty$. Because 0 itself can't be directly represented in the logarithmic domain (its logarithm is undefined), a practical solution is to define the representation of 0 using the largest (most negative) number available in the system. This convention has several advantages:

1. It reflects the inherent limit behaviour of the logarithm near 0.
2. It ensures that every positive number is assigned a logarithmic value higher than that of zero, so has a consistent ordering (although depending on the resolution, nearby numbers will share a quantised logarithmic representation).

The Implementation

I implemented this ideal addition function in both TensorFlow and PyTorch to demonstrate how LNS arithmetic can be integrated into one of these frameworks. This challenge underscores the difficulties inherent in using LNS in a framework designed for FP computations. In particular, bit manipulations and handling of special cases had to be translated into tensor operations with framework specific nuances. Note that in practice, we would implement an approximate function since this is computationally intensive.

4 Project Goals

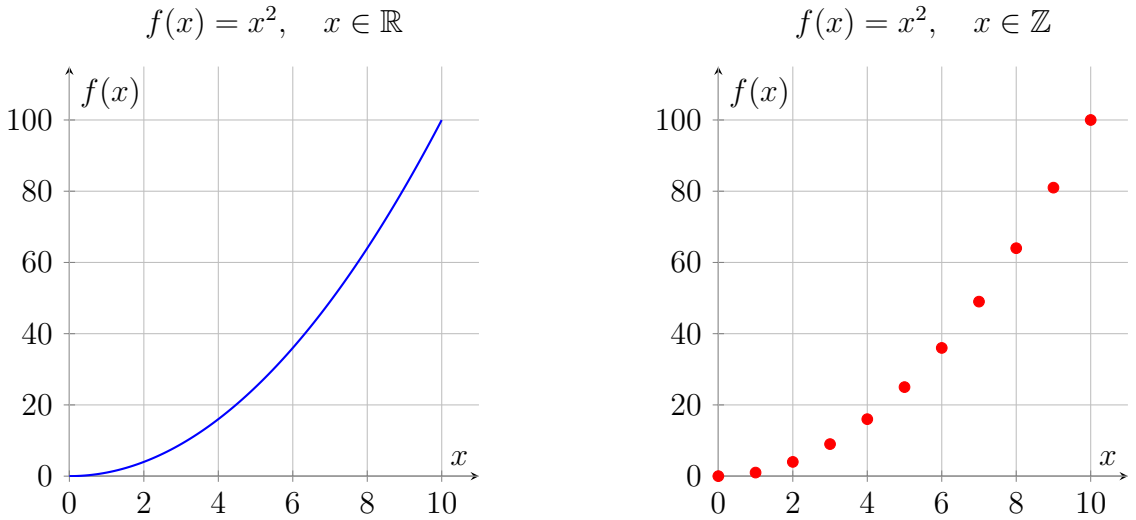
One intuitive approach is to create a wrapper class for Tensor that holds an integer logarithmic encoding of its elements and defines custom operators for log-domain arithmetic. Custom gradient functions can then be overloaded to apply LNS-based derivative rules, allowing both forward and backpropagation to use LNS arithmetic. Finally, we can implement custom layers and optimizers that use our wrapper class. This will let users work with LNS as easily as standard floating-point arithmetic. In this section, I'll give proof of concepts and demos for all the major features I plan to implement. This will demonstrate my ability to complete the project and will allow me to work more efficiently in the coding period. If you prefer, you can follow along with the demos in the github repository (which will contain the full working scripts rather than snippets) at https://github.com/akrentz6/xlns-gsoc-application/.

4.1 PyTorch vs Tensorflow

The most important design choice for this project is the machine learning framework. I propose that PyTorch is the better option for several reasons that I will outline in this section: a research-friendly econsystem, flexibility in overloading operators, and ease of use for defining custom gradients.

4.2 Problems with Integer Tensor Differentiation

A key challenge with this project is that natively, both TensorFlow and PyTorch allow only `float` and `complex` tensor types for automatic differentiation. In fact, this is a reasonable assumption to make when considering standard practices in machine learning. Consider the following functions:



We can see that there is no meaningful geometric interpretation of differentiating the discrete function on the right. However, in our case, we would ideally represent reals internally with an integer tensor. For us, taking derivatives of functions involving integers would be very useful. Take a look at the following example:

```python
import torch

x = torch.tensor(1, requires_grad=True)
y = torch.tensor(2, requires_grad=True)

z = x + y
z.backward()
```

```
RuntimeError: Only Tensors of floating point and complex dtype can require
 gradients
```

PyTorch is unwilling to compute derivatives of z with respect to x and y since they are both integer tensors. At first glance, TensorFlow does appear to handle integer tensors as it raises no errors, but when we try to compute these derivatives we get `None` values:

```python
import tensorflow as tf

x = tf.Variable(1)
y = tf.Variable(2)

with tf.GradientTape() as tape:
    z = x + y

grad = tape.gradient(z, [x, y])
print("Gradients w.r.t. x and y:", grad)
```

```
Gradients w.r.t. x and y: [None, None]
```

So how can we integrate LNS arithmetic into one of these frameworks without this functionality? My first idea was to create a fork of PyTorch, modified to support integer tensor differentiation. This would require several changes to PyTorch's code base which you can find in the appendix and would treat integer tensors similar to floating point tensors when performing backpropagation. Although this would be the most optimal solution, it requires us to maintain and merge changes made to PyTorch which, if they conflict with the changes we make, would be a nuisance. Users would also have to install our PyTorch fork rather than the official library.

Note that neither PyTorch nor TensorFlow are likely to support this kind of integer tensor differentiation anytime soon (see #31880, #76008, and #78436), so it is unlikely that this will be a viable approach. In the Experimental Features section, I discuss potential future ideas that would involve forking PyTorch as a side project and using that as an extension of this library (all main functionality would still work with the official PyTorch library).

The alternative to forking PyTorch is to represent these integer values using 64-bit floating point numbers (doubles). Doubles can exactly represent all integers from -2^{53} to 2^{53} since they have a 52 bit significand (mantissa) as defined by IEEE 754. While this doesn't provide the full range of a 64-bit integer, for many applications - especially in deep learning where extreme precision isn't necessary - this trade-off is perfectly acceptable. With this

approach, we can take advantage of PyTorch's existing autograd framework without having to customise it for integer differentiation. Consider the following example with doubles (i.e. `torch.float64`) where I implement a toy multiplication example.

```
import torch

class LNSMulFunction(torch.autograd.Function):

    @staticmethod
    def forward(ctx, a, b):
        ctx.save_for_backward(a, b)
        # Multiplication in the log domain becomes addition
        out = a + b
        return out

    @staticmethod
    def backward(ctx, grad_output):
        a, b = ctx.saved_tensors
        # The partial derivatives with respect to a and b are:
        # d/da(ab) = b
        # d/db(ab) = a
        # We are still working in the log domain during
        # backpropagation so multiplication once again
        # becomes addition.
        grad_a = grad_output + b
        grad_b = grad_output + a
        return grad_a, grad_b

# The internal representations of two decimal numbers
x = torch.tensor(2, dtype=torch.float64, requires_grad=True)
y = torch.tensor(3, dtype=torch.float64, requires_grad=True)

z = LNSMulFunction.apply(x, y)
print("x * y =", z)

# We supply a zero tensor for the gradient argument (the
# inital gradient with respect to the output tensor) since
# in our LNS, the number 1 is internally represented by a 0.
z.backward(gradient=torch.zeros_like(z))
print("Derivative w.r.t x:", x.grad)
print("Derivative w.r.t y:", y.grad)
```

```
x * y = tensor(5., dtype=torch.float64, grad_fn=<LNSMulFunctionBackward>)
Derivative w.r.t x: tensor(3., dtype=torch.float64)
Derivative w.r.t y: tensor(2., dtype=torch.float64)
```

In this demo, I subclass `torch.autograd.Function` which allows me to define custom gradients for the backwards pass. Taking two reals, 4 and 8, that would internally be represented as 2 and 3, their product (the sum of the internal representations) is $32 = 2^5$ and their derivatives are $8 = 2^3$ and $4 = 2^2$ respectively. I pass a zero tensor to the backward function since this tensor is supposed to represent the gradient w.r.t. the output tensor. The function's default is a ones tensor which would be a zeros tensor in the LNS domain.

4.3 Overloading Operators

Consider the following `LNSTensor` wrapper class:

```
import numpy as np
import torch

f = 23
base = torch.tensor(2.0**(2**(-f)), dtype=torch.float64)

class LNSTensor(object):

    def __init__(self, data, from_lns=False):

        # Convert data to a torch.Tensor of the correct type
        if not isinstance(data, torch.Tensor):
            if isinstance(data, np.ndarray):
                input_data = torch.from_numpy(data).to(torch.float64)
            else:
                # For scalars, lists, or tuples
                input_data = torch.tensor(data, dtype=torch.float64)
        else:
            input_data = data.to(torch.float64)

        if from_lns:
            self._lns = input_data
        # We use 'with torch.no_grad()' here because the conversion
        # from floating-point to LNS shouldn't be tracked by autograd.
        else:
            with torch.no_grad():
                log_data = (torch.log(input_data)/torch.log(base))
                quant_log_data = log_data.round().to(torch.int64)
                sign_bit = (input_data <= 0).to(torch.int64)
                self._lns = torch.bitwise_left_shift(quant_log_data, 1)
                self._lns += sign_bit
                self._lns = self._lns.to(torch.float64)

        self._lns.requires_grad_(True)
```

The purpose of this class is to manage and store the internal LNS representation of a tensor. It can either convert a floating-point tensor into its corresponding LNS representation or be initialised directly from an existing LNS representation. We'd like to be able to use PyTorch's operators (e.g. `torch.mul`) with this class and implement custom behaviour, but this will raise error messages since the operators haven't been well-defined for `LNSTensor`:

```
x = LNSTensor(1)
y = LNSTensor(2)
z = torch.mul(x, y)
```

```
TypeError: mul(): argument 'input' (position 1) must be Tensor, not
LNSTensor
```

PyTorch, unlike TensorFlow, provides methods for extending the torch library with Tensor-like classes such as `LNSTensor`. By defining the `LNSTensor` class method `__torch_function__`, we can override any PyTorch API function as follows:

```
@classmethod
def __torch_function__(cls, func, types, args=(), kwargs=None):
    if kwargs is None:
        kwargs = {}
    if func not in HANDLED_FUNCTIONS or not all(
        issubclass(t, LNSTensor)
        for t in types
    ):
        return NotImplemented
    return HANDLED_FUNCTIONS[func](*args, **kwargs)
```

Here, `HANDLED_FUNCTIONS`, a dict object, acts as a global dispatch table to store custom implementations of any torch function we choose. Using the `LNSMulFunction` class we defined before and a decorator to add our custom implementations to the `HANDLED_FUNCTIONS` dispatch table, we get:

```
HANDLED_FUNCTIONS = {}

def implements(torch_function):
    """Register a torch function override for LNSTensor."""
    def decorator(func):
        functools.update_wrapper(func, torch_function)
        HANDLED_FUNCTIONS[torch_function] = func
        return func
    return decorator

@implements(torch.mul)
def multiply(input, other):
    out_lns = LNSMulFunction.apply(input._lns, other._lns)
    return LNSTensor(out_lns, from_lns=True)
```

All that remains is to call `torch.mul` on our `LNSTensor` objects and all the work is done under the hood to handle the LNS arithmetic in both the forward and backpropagation.

```
x = LNSTensor(1) # Internal representation is Tensor(0)
y = LNSTensor(2) # Internal representation is Tensor(16777216)

z = torch.mul(x, y)
print("x * y =", z._lns.item()) # Internal representation is 1*2=2:
Tensor(16777216)

z._lns.backward(gradient=torch.zeros_like(z._lns))
print("Derivative w.r.t x:", x._lns.grad.item())
print("Derivative w.r.t y:", y._lns.grad.item())
```

```
x * y = 16777216.0
Derivative w.r.t x: 16777216.0
Derivative w.r.t y: 0.0
```

And finally, to clean up the code we can write some quality of life wrapper methods in `LNSTensor` so that users aren't required to interact directly with the internal representations. This gives functionality that would look something like this:

```
x = LNSTensor(1)
y = LNSTensor(2)

z = torch.mul(x, y)
print("x * y =", z)

z.backward()
print("Derivative w.r.t x:", x.grad)
print("Derivative w.r.t y:", y.grad)
```

```
x * y = LNSTensor(16777216)
Derivative w.r.t x: LNSTensor(16777216)
Derivative w.r.t y: LNSTensor(0)
```

This is a clear way in which PyTorch's core design is better suited to this project than TensorFlow. Defining and overriding operations directly in the `LNSTensor` class allows for far smoother integration of LNS arithmetic into the library. Comparatively, integrating custom operator overrides is less intuitive in TensorFlow's ecosystem and would be far more complicated.

4.4 Alternative Operator Implementations

So far, I've only discussed the ideal addition function. This is a computationally intense operation, so `xlnsconf` provides more efficient approximations (for addition and other functions) as alternatives. It would be very useful to support these in this project and I will provide two methods of doing so. First, we need to support multiple implementations for the same torch function.

```
def implements(torch_function, key=None, default=False):
    """Register a torch function override for LNSTensor."""
    def decorator(func):
        function_key = key or func.__name__
        functools.update_wrapper(func, torch_function)
        if torch_function not in HANDLED_FUNCTIONS:
            HANDLED_FUNCTIONS[torch_function] = {}
        HANDLED_FUNCTIONS[torch_function][function_key] = func
        if default:
            DEFAULT_IMPLEMENTATIONS[torch_function] = key
        return func
    return decorator
```

This new `implements` decorator takes optional parameters `key` and `default`. The dictionary `HANDLED_FUNCTIONS` now contains nested dictionaries, each of which stores all the different implementations for a function. In addition, we create a `DEFAULT_IMPLEMENTATIONS` dictionary which, as its name says, stores the default implementation for each function. We also need to add support for this in the `__torch_function__` method.

```
@classmethod
def __torch_function__(cls, func, types, args=(), kwargs=None):
    if kwargs is None:
        kwargs = {}
    if func not in HANDLED_FUNCTIONS or not all(issubclass(t, LNSTensor)
    for t in types):
        return NotImplemented
    chosen_impl = DEFAULT_IMPLEMENTATIONS.get(func)
    if chosen_impl is None:
        raise RuntimeError(f"No default implementation has been set for
        {func}.")
    if chosen_impl not in HANDLED_FUNCTIONS[func]:
        raise ValueError(f"Implementation key '{chosen_impl}' is not
        registered for {func}.")
    return HANDLED_FUNCTIONS[func][chosen_impl](*args, **kwargs)
```

Finally, creating a function `set_default` and a context manager `override_impl` would allow users to choose between implementations as follows:

```
set_default(torch.mul, "alternative_1") # switches the default to some
alternative implementation

x = LNSTensor(1)
y = LNSTensor(2)

z = torch.mul(x, y) # uses alternative_1
with override_impl(torch.mul, "default"):
    w = torch.mul(x, y) # uses the original default implementation
```

This will allow me to implement each of the `xlnsconf` implementations using native PyTorch operations. In addition, it would be useful to support the implementations already defined in `xlnsconf`. In particular, if an implementation is added to `xlnsconf` but has not been ported to the PyTorch side of the library, users won't be able to use it so it would be useful to provide functionality for this.

Support for this could come in the form of functions, e.g. `set_sbdb_ufunc` and others. This would take a given `sbdb_ufunc` implementation from `xlnsconf` as a parameter. Then when performing addition, if this function has been called, the addition function would convert the `LNSTensor` internal representations to `numpy.ndarray` objects and pass them to the corresponding `sbdb_ufunc`. It would then wrap the output in an `LNSTensor` again and return to the user, doing all the conversions under the hood.

4.5 Support for XLNS Objects

We can also go a step further and define operations between `LNSTensor` and the `xlns` objects. For now, I will only discuss the non-redundant objects with a global base and precision: `xlns`, `xlnsnp` (and `xlnsud`). The other cases will be considered later. We should first note that trying to pass `xlns` objects to PyTorch functions like `torch.mul` will raise unavoidable errors since none of the `xlns` objects implement `__torch_function__` so PyTorch doesn't treat them as "tensor-like".

```
x = LNSTensor(1)
y = xl.xlns(2)
z = torch.mul(x, y)
```

```
TypeError: mul(): argument 'other' (position 2) must be Tensor, not xlns
```

Thankfully, in most settings, users won't require the majority of PyTorch functions to work directly with xlns objects. In practice, the operations that matter - like addition, subtraction, multiplication, and division - between `xlns` and `LNSTensor` can be sorted rather easily. To do this, we can simply define the appropriate dunder methods in the `LNSTensor` class, ensuring that xlns objects (or any other standard Python objects) play nicely with `LNSTensor`.

```
def lnstensor(data, from_lns=False):
    if isinstance(data, LNSTensor):
        input_data = data._lns
        from_lns = True
    elif isinstance(data, torch.Tensor):
        input_data = data.to(torch.float64)
    elif isinstance(data, np.ndarray):
        input_data = torch.from_numpy(data).to(torch.float64)
    # For xlns non-redundant, global precision scalars
    elif isinstance(data, (xl.xlns, xl.xlnsud)):
        input_data = torch.tensor(data.x << 1 + data.s,
        dtype=torch.float64)
        from_lns = True
    # For xlns non-redundant, global precision numpy-like arrays
    elif isinstance(data, xl.xlnsnp):
        input_data = torch.tensor(data.nd, dtype=torch.float64)
        from_lns = True
    # For scalars, lists, or tuples, etc
    else:
        input_data = torch.tensor(data, dtype=torch.float64)
    return LNSTensor(input_data, from_lns=from_lns)
```

In the above code, I move the type-checking logic into a dedicated function called `lnstensor`, much like `torch.tensor`, in order to support global-precision xlns objects. This lets us clean up the `__init__` method by removing explicit type checks. We can now also add a custom `__mul__` dunder method as follows:

```
def __mul__(self, other):
    return torch.mul(self, lnstensor(other))
```

And finally, putting this together we have a toy multiplication example between `LNSTensor` objects and the existing `xlns` objects. This also handles operations between `LNSTensor` and scalars or other python and numpy objects.

```
x = lnstensor([1, 2])
y = xl.xlnsnp([2, 1])
z = xl.xlns(2)

u = x * y
v = x * z
w = x * 2

print("Multiply x and y element wise:", u)
print(f"Multiply x by 2 (xlns and scalar):\n", v, w)
```

```
Multiply x and y element wise: LNSTensor([16777216 16777216])
Multiply x by 2 (xlns and scalar):
LNSTensor([16777216 33554432]) LNSTensor([16777216 33554432])
```

4.6 Redundant XLNS Objects

Xlns also offers support for (dual) redundant LNS with global precision. We can represent any real number x in the redundant LNS with the following scheme. Let B denote the underlying base and let

$$x = B^{X_p} - B^{X_n} \tag{10}$$

where X_p and X_n are real numbers. Note that this representation isn't unique. We can see that we are more easily able to represent zero (with $X_p = X_n$) and the sign of x (with $X_p > X_n$ or $X_p < X_n$ respectively). Whilst addition (and subtraction) in redundant LNS is a similar process to its non-redundant counterpart, multiplication, division, square rooting, etc are far harder which defeats much of the purpose of working with this LNS arithmetic.

$$\begin{aligned} z = x \cdot y &= \left(B^{X_p} - B^{X_n}\right) \cdot \left(B^{Y_p} - B^{Y_n}\right) \\ &= B^{X_p+Y_p} + B^{X_n+Y_n} - \left(B^{X_n+Y_p} + B^{X_p+Y_n}\right) \\ &= B^{Z_p} - B^{Z_n} \end{aligned} \tag{11}$$

Where Z_p and Z_n can be computed using an analogous process to addition for non-redundant LNS as in equations (4)-(9). For this reason, `xlns` doesn't support these kinds of operations for its redundant LNS objects `xlnsr` and `xlnsnpr`. Consequently, I have opted not to include support for these redundant LNS objects in my implementation, as doing so would unnecessarily complicate the codebase without delivering meaningful benefits.

4.7 Variable Precision XLNS Objects

So far, I have only dealt with the global precision LNS implementations. I have set this manually for my code examples but in the real implementation, this will default to the global `xlns.xlnsB`. However, `xlns` also supports objects with a per instance precision (and corresponding base), `xlnsv` and `xlnsnpv`, as well as those with a per instance base, `xlnsb` and `xlnsnpb`.

To address these cases, I'll extend the `LNSTensor` class by adding an optional `base` parameter. When using the `lnstensor` function, users can optionally provide either an `f` or `b` argument. If neither is provided, the base defaults to `xlnsB`. If `f` is provided, a corresponding base is calculated and passed to `LNSTensor`. Likewise, if `b` is provided without an accompanying `f`, that value is used as the base. This would look something like this:

```
x = lnstensor(1) # base defaults to xlnsB=2^(2^-f) where by default, f=23
y = lnstensor(2, f=10) # base is 2^(2^-10)
z = lnstensor(3, b=1.01) # base is 1.01
w = lnstensor(4, f=10, b=1.01) # base is still 1.01
```

Where `LNSTensor` now includes a `base` variable. The critical challenge is ensuring that if a precision/base object (such as `xlnsv`, `xlnsnpb`, etc.) is supplied along with a `b` or `f` parameter, the object's value is converted to use the new base or precision. For example:

```
x = xl.xlnsv(1, setF=10) # base is 2^(2^-10)
y = xl.xlnsnpb(1, setB=1.001) # base is 1.001
z = lnstensor(x, b=1.01) # we must convert the base from 2^(2^-10) to 1.01
w = lnstensor(y) # we must convert the base from 1.001 to 2^(2^-23)
```

This conversion is performed as follows. Consider a nonzero real number x originally represented with base B as in (1):

$$X = \log_B |x|, \quad \Rightarrow \quad |x| = B^X.$$

We wish to convert this internal representation to a new base C. To do this, we compute the logarithm of $|x|$ in base C:

$$\begin{aligned} \log_C |x| &= \log_C \left(B^X\right) \\ &= X \cdot \log_C B \\ &= X \cdot \frac{\ln B}{\ln C}, \end{aligned} \tag{12}$$

where the final step follows from the change-of-base formula. Note that the sign of x remains unchanged, so the associated sign s_x is unaffected by this conversion. I've also excluded the changes in code to `lnstensor` for the sake of brevity. We require equation (12) in terms of the natural logarithm since libraries like NumPy and PyTorch only support specific bases (2, 10, e) for their logarithm functions.

Now we can handle support for operations between objects with mismatched bases. In `xlns`, when performing an operation with two operands, the output's base (or precision) is determined by the primary input. In other words, the base associated with the left-hand operand takes priority, and if the other operand has a different base, it is automatically converted to match the input's base; I will follow this convention for the project. This means the custom `torch.mul` implementation will become:

```
@implements(torch.mul, key="default_mul", default=True)
def multiply(input, other):
    other_aligned = lnstensor(other, b=input.base.item())
    out_lns = LNSMulFunction.apply(input._lns, other_aligned._lns)
    return LNSTensor(out_lns, from_lns=True, base=input.base)
```

And this lets us finally perform operations between `LNSTensor` and variable precision objects as follows:

```
x = lnstensor(2) # f=23
y = xl.xlnsv(2, setF=10)
print(x * y)
```

```
LNSTensor(33554432)
```

4.8 Building Models

Now that we've fleshed out the `LNSTensor` we need to see how to actually write layers, optimizers, loss functions, etc using LNS arithmetic. The easiest of these is loss functions. PyTorch provides classes like `MSELoss` which can be used as follows:

```
loss = torch.nn.MSELoss()

input = torch.randn(3, 5, requires_grad=True)
target = torch.randn(3, 5)

output = loss(input, target)
print(output)
```

```
tensor(2.4310, grad_fn=<MseLossBackward0>)
```

These loss classes can be thought of as wrappers for the loss functions themselves which are defined in `torch.nn.functional` (like `torch.nn.functional.mse_loss`). Luckily for us, we know how to overload PyTorch functions to support `LNSTensor` objects; in the same way we overload addition, multiplication, etc, we can use our `implements` decorator. In fact, most of the loss functions themselves use basic torch operations that I'll have already implemented so we won't need to make any changes here to support them. Most of the work will involve tests to make sure the loss functions work as expected.

Layers, however, are a bit more challenging. Since all PyTorch layers use normal `Tensor` objects, I'll be forced to reimplement them using `LNSTensor`. Another problem is the way in which PyTorch registers trainable variables so that optimizers know how to update weights. The `torch.nn.Parameter` class wraps a `Tensor` and automatically registers it as a model's learnable parameter. Since `LNSTensor` isn't a subclass of `Tensor`, we can't directly use it as a `Parameter` (see #73459).

To fix this, I store the internal representation `Tensor` of an `LNSTensor` as a `Parameter`. Since learnable parameters must be stored as attributes of a layer so that PyTorch can automatically discover, register, and manage them, I store a reference to the `Tensor` within the custom layer.

```
class TestLayer(torch.nn.Module):

    def __init__(self):
        super().__init__()

        # generate random weights
        self.weights = lnstensor(torch.randn())

        # Wrap the internal representation in a Parameter
        self.weights._lns = torch.nn.Parameter(self.weights._lns)
        # Store a reference to the parameter so it's visible
        self.weights_param = self.weights._lns
```

Where in practice there would be a method like `LNSTensor.set_param` to handle this internal logic - this is just to demonstrate what will happen under the hood.

Finally, we need to support PyTorch's optimizers. Since we are forced to declare each parameter using the internal representation of an `LNSTensor`, a 64-bit float, the optimizer will perform standard floating point operations on this. We want to perform LNS arithmetic, since this tensor is in the log domain. To do this, we can create custom optimizers, like the custom layers, which will wrap the internal representation back in an `LNSTensor` and perform the correct operations. Here's a simplified example for gradient descent:

```
class TestOptimizer(torch.optim.Optimizer):

    def __init__(self, params, lr=0.1):
        defaults = dict(lr=lr)
        super(TestOptimizer, self).__init__(params, defaults)

    # For this toy example, ignore the closure parameter
    def step(self, closure=None):

        # Iterate over each parameter group
        for group in self.param_groups:

            lr = group["lr"]
            for p in group["params"]:
```

```
15
16                  if p.grad is None:
17                      continue
18
19                  grad = p.grad.data
20                  with torch.no_grad():
21
22                      # Wrap variables in the log-domain in LNSTensors
23                      temp_data = lnstensor(p.data, from_lns=True)
24                      temp_grad = lnstensor(grad, from_lns=True)
25
26                      # Now uses LNSTensor operations
27                      temp_data = temp_data - temp_grad * lr
28                      p.data = temp_data._lns.data
```

Where once again, I will add quality of life methods to `LNSTensor` to handle this more smoothly. Once I have added LNS support in these three areas, users will have access to the vast majority of PyTorch's features with LNS.

4.9 Research-Friendly Ecosystem

One of the key factors in favor of PyTorch over TensorFlow is its inherently research-friendly ecosystem. The majority of published studies use PyTorch, which reflects the advantages it offers. Its dynamic computational graph allows for faster prototyping and debugging, providing immediate execution of operations that grants researchers greater control over custom implementations. In contrast, TensorFlow's historically static, declarative graph-building approach can constrain flexibility during research and development. In addition, I have already laid out how the PyTorch ecosystem makes it easier to integrate and test unconventional or experimental data types and operations.

4.10 Benchmarking and Visualisation Tools

An important feature that I'd like to implement in this project is a benchmarking suite. This could include detailed memory profiling to track the memory usage of objects like `LNSTensor` and compare memory footprints between `LNSTensor` objects with different bases as well as with the standard PyTorch `Tensor` object in order to analyse tradeoffs.

In addition, we could build a dashboard/logging system (potentially integrated into TensorBoard or a similar tool) showing histograms of numerical errors per layer over time during training. This would allow us to evaluate the accuracy of different implementations of, for example, custom addition and subtraction functions and analyse the tradeoffs of efficiency with convergence. Visualising this with graphs and heatmaps as well as tracking mean absolute errors, average errors, variance and comparing these with the baseline of a high precision floating point implementation will be very useful.

The easiest benchmarking we can add is tracking the number of calls to each function, and the total time elapsed. This can be done in `__torch_function__` using just the standard Python `time` library:

```
@classmethod
def __torch_function(cls, func, types, args=(), kwargs=None):
    ... # skipping start of the function
    call_impl = HANDLED_FUNCTIONS[func][chosen_impl]
    if BENCHMARKING_ENABLED:
        start_time = time.perf_counter()
        result = call_impl(*args, **kwargs)
        elapsed = time.perf_counter() - start_time
        stats = BENCHMARK_STATS.setdefault(func.__name__, {"calls": 0,
        "total_time": 0.0})
        stats["calls"] += 1
        stats["total_time"] += elapsed
        return result
    else:
        return call_impl(*args, **kwargs)
```

We can also add support for memory profiling with PyTorch's `torch.profiler` library. This allows us to track the allocation and consumption of CUDA memory if available, as well as record shapes and sizes of tensors processed during computation. This information is invaluable when looking for memory bottlenecks or inefficiencies.

```
@classmethod
def __torch_function(cls, func, types, args=(), kwargs=None):
    ... # skipping start of the function
    call_impl = HANDLED_FUNCTIONS[func][chosen_impl]
    if BENCHMARKING_ENABLED:
        start_time = time.perf_counter()
        if MEMORY_BENCHMARKING_ENABLED:
            with torch.profiler.profile(
                    enable_cuda=torch.cuda.is_available(),
                    profile_memory=True,
                    record_shapes=True) as prof:
                result = call_impl(*args, **kwargs)
            # then do something with the memory data
        else:
            result = call_impl(*args, **kwargs)
        elapsed = time.perf_counter() - start_time
        stats = BENCHMARK_STATS.setdefault(func.__name__, {"calls": 0,
        "total_time": 0.0})
        stats["calls"] += 1
        stats["total_time"] += elapsed
        return result
    else:
        return call_impl(*args, **kwargs)
```

We also want to be able to compare an LNS model as a whole to its floating point counterpart or to another base. To do this, we need to be able to parse a model to extract its layers, activations and other features. Whilst this is infeasable for more complicated models which will need to be specified manually, we can automate the process for the majority of models as follows:

```python
import torch

class TestModel(torch.nn.Module):

    def __init__(self):
        super(TestModel, self).__init__()
        self.linear1 = torch.nn.Linear(100, 200)
        self.activation = torch.nn.ReLU()
        self.linear2 = torch.nn.Linear(200, 10)
        self.softmax = torch.nn.Softmax()

    def forward(self, x):
        x = self.linear1(x)
        x = self.activation(x)
        x = self.linear2(x)
        x = self.softmax(x)
        return x

def get_children(model):
    children = list(model.children())
    flat_children = []
    if not children:
        return model
    for child in children:
        # get_children returns either a torch.nn.Module or a list
        try:
            flat_children.extend(get_children(child))
        except TypeError:
            flat_children.append(get_children(child))
    return flat_children

print(*get_children(TestModel()), sep="\n")
```

```
Linear(in_features=100, out_features=200, bias=True)
ReLU()
Linear(in_features=200, out_features=10, bias=True)
Softmax(dim=None)
```

Once we know the architecture of an LNS model, we can build an analogous floating point model (or model with a different base) using the standard PyTorch functions. This can then be used to compare convergence and numerical accuracy. We'll also be able to identify in particular which LNS layers perform worse and from there, which operations are the least numerically accurate.

Finally, to visualise all of the useful metrics that we'll track, we can support TensorBoard and MatPlotLib. We can plot graphs in MatPlotLib in the usual way and also stream live metrics and summaries to TensorBoard, allowing for real-time monitoring and more interactive exploration of our data.

4.11 Documentation and Testing

It'll be key to document the architecture of the code throughout since interfacing with PyTorch might get complicated. Also, writing clear comments on the maths involved in LNS arithmetic will be very useful to people unfamiliar with the specifics. I will make sure to create beginner friendly user guides and tutorials for each feature I implement with code snippets and other helpful resources to understand how to use the library.

For generating documentation, I'll use Sphinx, the industry standard for Python libraries (and what PyTorch uses too). Sphinx can automatically extract docstrings from source code as well as produce documentation in HTML, PDF, ePub, and more. We can also set up workflows in Github Actions to automatically build the documentation and render it with Github Pages. We'll be able to use this in the future to document the main `xlns` codebase.

Also, I will use PyTest, a widely used testing framework, to test each custom LNS operation, its autograd integration, and each custom layer and optimizer to ensure that they are compatible with PyTorch's entire framework. PyTest is a good choice for this since it's minimalistic, making tests easier to write and maintain, whilst still being powerful and allowing for tests on complex functionality. In addition, it has an extensive plugin ecosystem giving access to hundreds of useful testing tools including parallel testing, code coverage, etc. We can also integrate these tests into the repository with Github Actions so that any pull requests must pass the tests to be committed. As with Sphinx, we can use PyTest in the future to improve the tests in the main `xlns` codebase.

4.12 Experimental/Future Features

If I'm ahead of schedule near to the end of the project, I have some ideas for other experimental features to implement. I am also interested in working on these features after Google Summer of Code is over.

It would be interesting to look into creating a tool that could seamlessly convert standard PyTorch code and model architectures into LNS-based equivalents. If we could analyse Python code, similar to how TorchScript does, we could inspect a model's layers and operations and and automatically substitute in their LNS counterparts. This would allow developers and researchers to experiment with LNS arithmetic with limited to no rewriting of their code. This is, like the other experimental features, a large project and would potentially be better suited to a future GSoC project.

Another potential feature is a custom `dtype`. Internally, this `dtype` would be stored as a 64-bit word - similar to the internal representation used in our LNSTensor wrapper - but we could subclass `Tensor` objects of this `dtype` to provide custom operations. This design offers greater flexibility, allowing us to experiment with alternative LNS representations (for example, incorporating a dedicated zero bit). It would also allow others to implement and test their own arithmetic systems. This would, however, require extensive knowledge and understanding of the PyTorch codebase as well as a lot of planning. It would be challenging to implement and merge new updates from PyTorch since it would require changes to many

different sections of PyTorch's source code. However, it could be interesting as an additional side project and would help significantly with allowing researchers to experiment.

Finally, a third feature I'd like to work on on the side is compatibility with TorchScript. TorchScript is a way to create serialisable models from PyTorch code to run them outside of Python - it captures the architecture and structure of a model. Since we will need to define custom types like `LNSTensor` and custom layers to support arithmetic with `LNSTensor`, We might need to define these types as well as various functions in PyTorch's `jit` subfolder. This has the same drawbacks of creating a custom `dtype` in that it involves in depth changes to PyTorch's C++ code and would make it harder to maintain.

5 Project Schedule

5.1 Community Bonding Period

Over these three weeks, I will begin to prepare for the project. Firstly, I will set up an environment to work on the `xlns` library (either a fork that will be merged at the end or a branch of `xlns`). I'll also join PyTorch community spaces to learn more about better coding practices with PyTorch and look for better ways to implement my proposed features. In addition, I'll communicate with the mentors to further develop ideas for the project and ensure that our goals are aligned. Ideally, I'll start coding in this period - even though this isn't the development period, I feel it's important with a project like this to work on as many proof of concept/example code snippets as possible to get a feel for the difficulties of the later stages of the project, particularly the benchmarking suite. This will allow me to more confidently work during the development period and help me stay on schedule.

5.2 Development Period

Weeks 1-2: LNSTensor Wrapper Class (2nd June - 15th June)

1. Create helper functions (converting to and from `xlns` objects) and wrap tensor methods (e.g. backward, grad, etc) so that users never need to access the internal representation directly.
2. Implement the dispatch table to override torch functions.
3. Handle the majority of custom operations and their autograd functions.
4. Add support for `xlnsconf` and user-defined implementations with an easy system to choose between these.
5. Implement detailed tests to ensure these work as expected.
6. Create some user examples and demos.

Weeks 3-5: Custom Layers, Optimizers, and Building Blocks (16th June - 6th July)

1. Recreate the most popular types of layers (Linear, Convolutional, Recurrent, etc) using `LNSTensor` for the parameters.
2. Ensure that optimizers, activation functions, and loss functions work as expected with these models. For those that don't (likely many since under the hood most have C++ implementations) I will need to implement custom versions.

Note: This section will require the most documentation, examples and test cases (and likely the most lines of code) so it's important to set aside longer to make sure it's done right.

Weeks 6-8: Benchmarking and Visualisation Tools (7th July - 27th July)

1. Create tools to analyse the accuracy and speeds of different `LNSTensor` operations. We can compare these to baseline floating point metrics.
2. Extend this to create a wrapper class for LNS models. This will track numerical errors during the forward pass and/or backpropagation and in particular, which custom functions cause the greatest errors.
3. Visualise all this information in TensorBoard and/or MatPlotLib. This could include:
    - Time-series plots for training loss and accuracy, comparing FP and LNS runs side-by-side.
    - Bar charts or histograms showing per-operation latency breakdowns.
    - Heatmaps demonstrating how error levels vary across different layers or operations in the model.
    - Scatter plots correlating error magnitudes with operation execution times to show performance trade-offs.
4. Create profiling tools to inspect model internals and gradients.

Weeks 9-10: Returning to the LNSTensor and Layers (28th July - 10th August)

1. Ensure/add support for changing the device and make this work with the benchmarking suite (e.g. to CUDA or the CPU).
2. With the help of the benchmarking suite, implement more alternatives for addition and subtraction (as in `xlnsconf`).
3. Implement any missing torch operations for `LNSTensor`. The work/time put into this will depend on the schedule and how on track I am.
4. Work on additional custom LNS layers - create templates or other tools that make it easy for others in the future to do this.
5. Write detailed tests for all of the above and lots of tutorials/examples.

Week 11: Extra Features (11th August - 17th August)

Note: This week involves less important/experimental features that can be substituted out closer to the deadline if I am behind schedule. I want to keep this open in case the work in weeks 9-10 takes longer than expected.

1. Work on and research ways to improve the design of the library to make the user experience more seamless. The fewer calls to LNS specific functions or classes rather than PyTorch ones the better.
2. Add support for data preprocessing and pipelines. For example, PyTorch provides utilities for datasets and dataloaders so I would wrap these to support LNS.

3. Add more utility functions for saving/loading models, device management (CPU/GPU), etc.

Week 12: Final Documentation, Testing, and Admin Work (18th August - 24th August)

1. Finish any remaining documentation and test cases.
2. Convert existing PyTorch models into LNS based equivalents. Ensure that everything works well in larger, end-to-end programs.
3. Final checks and ensure compatibility with the rest of xlns and produce a working final build.

Submission Week: (25th August - 31st August)

Submit and finalise all code and any other required documents e.g. evaluations.

5.3 Final Thoughts

I will make several commits each day (or few days) to a branch of the xlns github repository or a fork of it to let mentors track my progress. In addition, I'll maintain a changelog and have regular zoom or skype meetings with my mentor to discuss my progress and any challenges I come across. Since I'll have no other commitments (vacations, lectures, work, etc) over the course of this project, I will consider this a full time job and will commit to a five day (at least 35 hour) work week. Also, note that I live in London so I'm in the BST time zone. I'm flexible with my schedule and can work around the mentors' schedules for meeting times.

6 Appendix

6.1 Challenge 1 Source Code

Tensorflow XLNS Data Demo

```
import tensorflow as tf
import xlns as xl

# Create an xlns object
xlns_data = xl.xlnsnp([2.0, 4.0, 5.0])
print("xlns data:", xlns_data)

# Try to create a TensorFlow constant directly from an xlns object
try:
    tf_data = tf.constant(xlns_data)
except ValueError as e:
    # Display only the first line of the error message
    error_str = str(e).splitlines()[0]
    print("\nTensorFlow error when passing xlns object:", error_str)

# The correct approach is to convert to floating point first:
fp_data = [float(x) for x in xlns_data.xlns()]
tf_data = tf.constant(fp_data)
print("\nConverted TensorFlow constant:", tf_data)
```

```
xlns data: [xlns(1.9999999986889088) xlns(3.9999999947556355) xlns
(5.00000016087236)]

TensorFlow error when passing xlns object: AttributeError: 'numpy.int64'
object has no attribute '__len__'

Converted TensorFlow constant: tf.Tensor([2. 4. 5.], shape=(3,), dtype=
float32)
```

PyTorch XLNS Data Demo

```
import torch
import xlns as xl

# Create an xlns object
xlns_data = xl.xlnsnp([2.0, 4.0, 5.0])
print("xlns data:", xlns_data)

# Try to create a PyTorch tensor directly from an xlns object
try:
    pt_data = torch.tensor(xlns_data)
except AttributeError as e:
    # Display only the first line of the error message
    error_str = str(e).splitlines()[0]
    print("\nPyTorch error when passing xlns object:", error_str)

# The correct approach is to convert to floating point first:
```

```
fp_data = [float(x) for x in xlns_data.xlns()]
pt_data = torch.tensor(fp_data)
print("\nConverted PyTorch tensor:", pt_data)
```

```
xlns data: [xlns(1.9999999986889088) xlns(3.9999999947556355) xlns
(5.00000016087236)]

PyTorch error when passing xlns object: 'numpy.int64' object has no
attribute '__len__'

Converted PyTorch tensor: tensor([2., 4., 5.])
```

6.2 Challenge 2 Source Code

TensorFlow Fully-Connected Network

```
import tensorflow as tf
import keras
import time

# Load and preprocess MNIST data:
#   - Flatten 28x28 images into 784-length vectors.
#   - Normalize pixel values to [0, 1]
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train = x_train.reshape(-1, 784).astype("float32") / 255.0
x_test = x_test.reshape(-1, 784).astype("float32") / 255.0

# Build the model as a Sequential Keras model:
model = tf.keras.models.Sequential([
    tf.keras.layers.InputLayer(shape=(784,)),
    tf.keras.layers.Dense(100, activation="relu",
                          kernel_initializer=tf.keras.initializers.RandomNormal(
                              mean=0.0, stddev=0.1),
                              bias_initializer="zeros"),
    tf.keras.layers.Dense(10, activation="softmax",
                          kernel_initializer=tf.keras.initializers.RandomNormal(
                              mean=0.0, stddev=0.1),
                              bias_initializer="zeros")
])

# Compile the model:
#   - Use sparse categorical crossentropy since labels are
integer-encoded.
#   - The SGD optimizer mimics the update rule in arn_generic.py.
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.1),
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])

# Train the model for seven epochs:
start = time.time()
history = model.fit(x_train, y_train, epochs=7, batch_size=128,
validation_data=(x_test, y_test))
```

```
elapsed = time.time() - start

# Evaluate on test data:
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
print("Test accuracy (TensorFlow):", test_acc)
print("Elapsed time:", elapsed)
```

PyTorch Fully-Connected Network

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import TensorDataset, DataLoader
from torchvision import datasets, transforms
import time

# Define the fully connected network as a subclass of nn.Module:
class FCNet(nn.Module):
    def __init__(self):

        super(FCNet, self).__init__()

        self.fc1 = nn.Linear(784, 100)
        self.fc2 = nn.Linear(100, 10)

        # Initialize: weights ~ N(0, 0.1) and biases = 0.
        nn.init.normal_(self.fc1.weight, mean=0.0, std=0.1)
        nn.init.zeros_(self.fc1.bias)
        nn.init.normal_(self.fc2.weight, mean=0.0, std=0.1)
        nn.init.zeros_(self.fc2.bias)

    def forward(self, x):

        x = x.view(-1, 784)
        x = F.relu(self.fc1(x))

        # We don't apply softmax here as the CrossEntropyLoss function
        # internally applies the appropriate log softmax transformation.
        x = self.fc2(x)

        return x

# Set up MNIST datasets with basic transforms (converting images to
tensors)
train_transform = transforms.ToTensor()
raw_train_dataset = datasets.MNIST('./data', train=True, download=True,
transform=train_transform)
raw_test_dataset = datasets.MNIST('./data', train=False, download=True,
transform=train_transform)

# Convert the datasets into in-memory tensors as they are loaded on the
fly by default.
# Note: raw_train_dataset.data is of shape [60000, 28, 28] and is of type
```

```
torch.uint8.
# We unsqueeze to add a channel dimension and convert to float, scaling to [0,1].
train_data = raw_train_dataset.data.unsqueeze(1).float() / 255.0
train_targets = raw_train_dataset.targets
test_data = raw_test_dataset.data.unsqueeze(1).float() / 255.0
test_targets = raw_test_dataset.targets

# Create TensorDatasets and DataLoaders based on the in-memory data.
train_dataset = TensorDataset(train_data, train_targets)
test_dataset = TensorDataset(test_data, test_targets)

train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
test_loader  = DataLoader(test_dataset, batch_size=128, shuffle=False)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = FCNet().to(device)

# Use the built-in cross entropy loss (fused log softmax + NLL).
loss_func = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)

start = time.time()
num_epochs = 7
for epoch in range(1, num_epochs + 1):

    # Training phase
    model.train()

    # Track cumulative loss and accuracy for the training epoch
    running_train_loss = 0.0
    train_correct = 0
    train_total = 0

    for data, target in train_loader:

        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()

        outputs = model(data)
        loss = loss_func(outputs, target)

        loss.backward()
        optimizer.step()

        # Accumulate the loss (multiplied by the batch size for averaging later)
        running_train_loss += loss.item() * data.size(0)

        _, predicted = torch.max(outputs, dim=1)
        train_total += target.size(0)
        train_correct += (predicted == target).sum().item()

    # Calculate the average training loss and accuracy for the epoch
```

```
93      train_epoch_loss = running_train_loss / train_total
94      train_epoch_acc = train_correct / train_total
95
96      # Validation phase
97      model.eval()
98
99      # Track cumulative loss and accuracy for the validation epoch
100     running_val_loss = 0.0
101     val_correct = 0
102     val_total = 0
103
104     # Disable gradient calculation for validation
105     with torch.no_grad():
106         for data, target in test_loader:
107
108             data, target = data.to(device), target.to(device)
109
110             outputs = model(data)
111             loss = loss_func(outputs, target)
112
113             # Accumulate the loss (multiplied by the batch size for
                averaging later)
114             running_val_loss += loss.item() * data.size(0)
115
116             _, predicted = torch.max(outputs, 1)
117
118             val_total += target.size(0)
119             val_correct += (predicted == target).sum().item()
120
121     # Calculate the average validation loss and accuracy for the epoch
122     val_epoch_loss = running_val_loss / val_total
123     val_epoch_acc = val_correct / val_total
124
125     # Print epoch summary
126     print(f"Epoch {epoch}:")
127     print(f"  Training   - Loss = {train_epoch_loss:.4f}, Accuracy =
        {train_epoch_acc:.4f}")
128     print(f"  Validation - Loss = {val_epoch_loss:.4f}, Accuracy =
        {val_epoch_acc:.4f}")
129
130 elapsed = time.time() - start
131
132 # Final test accuracy (optional, as we've already been validating each
    epoch)
133 model.eval()
134 correct = 0
135 total = 0
136
137 with torch.no_grad():
138     for data, target in test_loader:
139
140         data, target = data.to(device), target.to(device)
141
142         outputs = model(data)
```

```
        _, predicted = torch.max(outputs, 1)

        total += target.size(0)
        correct += (predicted == target).sum().item()

print("\nFinal Test Accuracy:", correct / total)
print("Elapsed time:", elapsed)
```

6.3 Challenge 3 Source Code

TensorFlow LNS Addition Function

```
import tensorflow as tf
import xlns as xl

# Constants for the base and zero internal representation
xlnsB_tf = tf.constant(xl.xlnsB, dtype=tf.float64)
xlns0_tf = tf.constant(-0x7fffffffffffffff, dtype=tf.int64)

def xlns_to_tf_const(xlns_array):
    return tf.constant(xlns_array.nd, dtype=tf.int64)

def tf_const_to_xlns(tf_tensor):

    internal_data = tf_tensor.numpy()
    fp_values = []

    # Set the internal data of each xlns object
    for value in internal_data:
        x_obj = xl.xlns(0)
        x_obj.x = value >> 1
        x_obj.s = value & 1
        fp_values.append(x_obj)

    return xl.xlnsnp(fp_values)

# Logarithm term of addition equation
@tf.function
def tf_sbdb_ideal(d, s, B=None):

    d_f = tf.cast(d, tf.float64)
    s_f = tf.cast(s, tf.float64)

    base = xlnsB_tf if B is None else tf.cast(B, tf.float64)
    term = tf.pow(base, d_f)

    v = tf.abs(1.0 - 2.0 * s_f + term)
    log_val = tf.math.log(v) / tf.math.log(base)

    result = tf.bitwise.left_shift(tf.cast(tf.round(log_val), tf.int64),
    1)
    return result

```

```
@tf.function
def tf_logadd(x, y):

    # Our formulae suppose the log part of x is greater than that of y.
    # We are able to take the maximum of the full internal representation
    # since the sign bit is the least significant bit so doesn't matter.
    part1 = tf.math.maximum(x, y)

    d = -tf.abs(tf.bitwise.right_shift(x, 1) - tf.bitwise.right_shift(y,
    1))
    s = tf.bitwise.bitwise_and(tf.bitwise.bitwise_xor(x, y), 1)

    part2 = tf_sbdb_ideal(d, s)

    return part1 + part2

@tf.function
def tf_myadd(x, y):

    # Handles logic for when their sum equals 0. In this case, the
    # internal representations differ only in their LSB, i.e. x^y=1.
    sum_to_zero_cond = tf.where(tf.equal(tf.bitwise.bitwise_xor(x, y), 1),
                                xlns0_tf, tf_logadd(x, y))

    # Handles logic for when either term equals 0.
    y_equals_zero_cond = tf.where(tf.equal(tf.bitwise.bitwise_or(y, 1),
                                           xlns0_tf),
                                  x, sum_to_zero_cond)
    x_equals_zero_cond = tf.where(tf.equal(tf.bitwise.bitwise_or(x, 1),
                                           xlns0_tf),
                                  y, y_equals_zero_cond)

    return x_equals_zero_cond

if __name__ == "__main__":

    # Test data including edge cases (negatives and numbers that sum to 0)
    # x = tf.Tensor([16777216 26591258 -9223372036854775808 0])
    # y = tf.Tensor([0 1 16777216 1])
    x = xlns_to_tf_const(xl.xlnsnp([2,  3,  0,  1]))
    y = xlns_to_tf_const(xl.xlnsnp([1, -1,  2, -1]))

    result = tf_myadd(x, y)
    print("TensorFlow myadd result:", tf_const_to_xlns(result))
```

```
TensorFlow myadd result: [xlns(2.9999999688096786) xlns
(1.9999999986889088) xlns(1.9999999986889088) xlns(0)]
```

PyTorch LNS Addition Function

```
import torch
import xlns as xl

# Constants
```

```
xlnsB_pt = torch.tensor(xl.xlnsB, dtype=torch.float64)
xlns0_pt = torch.tensor(-0x8000000000000000, dtype=torch.int64)

def xlns_to_torch(xlns_array):
    return torch.tensor(xlns_array.nd, dtype=torch.int64)

def torch_to_xlns(torch_tensor):

    internal_data = torch_tensor.cpu().numpy()
    fp_values = []

    # Set the internal data of each xlns object
    for value in internal_data:
        x_obj = xl.xlns(0)
        x_obj.x = value >> 1
        x_obj.s = value & 1
        fp_values.append(x_obj)

    return xl.xlnsnp(fp_values)

# Logarithm term of addition equation
def torch_sbdb_ideal(d, s, B=None):

    d_f = d.to(torch.float64)
    s_f = s.to(torch.float64)

    base = xlnsB_pt if B is None else torch.tensor(B,
    dtype=torch.float64, device=d.device)
    term = torch.pow(base, d_f)

    v = torch.abs(1.0 - 2.0 * s_f + term)
    log_val = torch.log(v) / torch.log(base)

    result =
    torch.bitwise_left_shift(torch.round(log_val).to(torch.int64), 1)
    return result

def torch_logadd(x, y):

    # Our formulae suppose the log part of x is greater than that of y.
    # We are able to take the maximum of the full internal representation
    # since the sign bit is the least significant bit so doesn't matter.
    part1 = torch.maximum(x, y)

    d = -torch.abs(torch.bitwise_right_shift(x, 1) -
    torch.bitwise_right_shift(y, 1))
    s = torch.bitwise_and(torch.bitwise_xor(x, y), 1)
    part2 = torch_sbdb_ideal(d, s)

    return part1 + part2

def torch_myadd(x, y):

    # Handles logic for when their sum equals 0. In this case, the
```

```
    # internal representations differ only in their LSB, i.e. x^y=1.
    sum_to_zero_cond = torch.where(torch.eq(torch.bitwise_xor(x, y), 1),
                                   xlns0_pt, torch_logadd(x, y))

    # Handles logic for when either term equals 0.
    y_equals_zero_cond = torch.where(torch.eq(torch.bitwise_or(y, 1),
                                              xlns0_pt),
                                     x, sum_to_zero_cond)
    x_equals_zero_cond = torch.where(torch.eq(torch.bitwise_or(x, 1),
                                              xlns0_pt),
                                     y, y_equals_zero_cond)

    return x_equals_zero_cond

if __name__ == '__main__':

    # Test data including edge cases (negatives and numbers that sum to 0)
    # x = torch.Tensor([16777216 26591258 -9223372036854775808 0])
    # y = torch.Tensor([0 1 16777216 1])
    x = xlns_to_torch(xl.xlnsnp([2,  3,  0,  1]))
    y = xlns_to_torch(xl.xlnsnp([1, -1,  2, -1]))

    result = torch_myadd(x, y)
    print("PyTorch myadd result:", torch_to_xlns(result))
```

```
PyTorch myadd result: [xlns(2.9999999688096786) xlns(1.9999999986889088)
xlns(1.9999999986889088) xlns(0)]
```

6.4 PyTorch Integer Tensor Differentiation

Below is an explanation of the key changes made to PyTorch's internals to support integer tensors in the autograd library, along with the corresponding code snippets. See https://github.com/pytorch/pytorch/blob/main/CONTRIBUTING.md for how to fork or make changes to the PyTorch library.

Adding an Integer Type Check in ScalarType.h

To enable treating integer tensors as differentiable, a new helper function was added. The function `isIntType()` checks whether a given `ScalarType` corresponds to any of the supported integer types (`Char`, `Byte`, `Short`, `Int`, `Long`). This function lays the groundwork by identifying integer types and will be used elsewhere in the code to ensure that integer tensors can be processed by autograd.

File: pytorch/c10/core/ScalarType.h

```
inline bool isIntType(ScalarType t) {
  return t == ScalarType::Char || t == ScalarType::Byte ||
      t == ScalarType::Short || t == ScalarType::Int ||
      t == ScalarType::Long;
}
```

Extending Differentiability in Variable.h

Previously, the system considered only floating and complex types as differentiable. The update in `isDifferentiableType()` adds integer types as differentiable by including a call to `isIntType()`.

File: pytorch/torch/csrc/autograd/variable.h (before)

```
48 static inline bool isDifferentiableType(at::ScalarType t) {
49   return isFloatingType(t) || isComplexType(t);
50 }
```

File: pytorch/torch/csrc/autograd/variable.h (after)

```
48 static inline bool isDifferentiableType(at::ScalarType t) {
49   return isFloatingType(t) || isComplexType(t) || isIntType(t);
50 }
```

Exposing Integer Type Information via Dtype in C++

In PyTorch, many parts of the framework are implemented in C++ for performance and interfaced with Python to allow users to interact with complex operations efficiently in an easy-to-use manner. In particular, the `dtype` class is written in C++, so we define a new class method here that calls `isIntType()`.

File: pytorch/torch/csrc/Dtype.cpp

```
56 static PyObject* THPDtype_is_int(THPDtype* self, PyObject* noargs) {
57   HANDLE_TH_ERRORS
58   if (at::isIntType(self->scalar_type)) {
59     Py_RETURN_TRUE;
60   } else {
61     Py_RETURN_FALSE;
62   }
63   END_HANDLE_TH_ERRORS
64 }
```

Updating the Dtype Interface in Python

The type hinting and interface for `dtype` (in the __init__.pyi.in file) must be updated to include an `is_int` attribute. This reflects that integer types are now supported by autograd and allows checks for integer tensors to be done at a high level in the python code.

File: pytorch/torch/_C/__init__.pyi.in (before)

```
189 # Defined in torch/csrc/Dtype.cpp
190 class dtype:
191     # TODO: __reduce__
192     is_floating_point: _bool
193     is_complex: _bool
```

```
    is_signed: _bool
    itemsize: _int
    def to_real(self) -> dtype: ...
    def to_complex(self) -> dtype: ...
```

File: pytorch/torch/_C/__init__.pyi.in (after)

```
# Defined in torch/csrc/Dtype.cpp
class dtype:
    # TODO: __reduce__
    is_floating_point: _bool
    is_int: _bool
    is_complex: _bool
    is_signed: _bool
    itemsize: _int
    def to_real(self) -> dtype: ...
    def to_complex(self) -> dtype: ...
```

Allowing Gradient Computation for Integer Outputs in autograd.cpp

The autograd mechanism must be modified to allow gradient computation for outputs with integer types. The check now verifies that the scalar type is either floating point or integer in the internal `_make_grads()` function.

File: pytorch/torch/csrc/autograd/autograd.cpp (before)

```
TORCH_CHECK(
    c10::isFloatingType(output.scalar_type()),
    "grad can be computed only for real scalar outputs but got ",
    output.scalar_type());
```

File: pytorch/torch/csrc/autograd/autograd.cpp (after)

```
TORCH_CHECK(
    c10::isFloatingType(output.scalar_type()) ||
    c10::isIntType(output.scalar_type()),
    "grad can be computed only for real or integer scalar outputs but got
    ",
    output.scalar_type());
```

Allowing Gradient Computation for Integer Outputs in Python

We must also change the analogous `_make_grads()` function in the python section of the library where we can finally use our `dtype.is_int` attribute.

File: pytorch/torch/csrc/autograd/__init__.py (before)

```
if not out_dtype.is_floating_point:
    msg = (
        "grad can be implicitly created only for real scalar outputs"
        f" but got {out_dtype}"
    )
    raise RuntimeError(msg)
```

File: pytorch/torch/csrc/autograd/__init__.py (after)

```
202 if not (out_dtype.is_floating_point or out_dtype.is_int):
203     msg = (
204         "grad can be implicitly created only for real or integer scalar
            outputs"
205         f" but got {out_dtype}"
206     )
207     raise RuntimeError(msg)
```