UNIVERSITY OF SOUTHAMPTON
Faculty of Engineering and Physical Sciences
School of Electronics and Computer Science

A project report submitted for the award of
BEng Computer Science

Supervisor: Jonathon Hare
Examiner: Sheng Chen

Novel Implementation of Training Control in TensorFlow and its Demonstration on iNALU

by Filip Kubacki

April 27, 2023

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

A project report submitted for the award of BEng Computer Science

by Filip Kubacki

The aim of this work is to engineer and test a new approach towards the TensorFlow training API. Embedding custom training procedures in the present TensorFlow distribution relies on training loop reimplementations. It is not portable, and we miss a lot of great TensorFlow default functionalities. Therefore, we successfully built and tested the wrapper to customise the training procedure based on the easy-to-read and modified Python configuration dictionaries. It keeps the default TensorFlow training API by adjusting the compilation of the data-flow graph. Utilising our novel approach, we successfully trained deep models of stacked improved Neural Arithmetic Logic Units (iNALU) layers. We discuss the architectural and technical decisions to deliver versatile and fresh functionality in the TensorFlow environment.

Contents

List of Figures

List of Symbols

$\cdot$	Multiplication
$\div$	Division
$+$	Element-wise addition
$-$	Element-wise subtraction
$\odot$	Element-wise multiplication
σ	Sigmoid function
sign	function returning an element-wise indication of the sign of a number
I	Input dimension size
O	Output dimension size
I	Element-wise multiplication
$\widehat{W}, \widehat{M}$	NALU learnt matrices of the size $\mathrm{R}^{I \times O}$
$\widehat{W^A}, \widehat{M^A}$	iNALU learnt summative matrices of the size $\mathrm{R}^{I \times O}$
$\widehat{W^M}, \widehat{M^M}$	iNALU learnt multiplicative matrices of the size $\mathrm{R}^{I \times O}$
$\widehat{G}$	NALU learnt gating matrice
NAC_+	Summative sub-unit/path in NALU
$iNAC_+$	Summative sub-unit/path in iNALU
$NAC_\cdot$	Multiplicative sub-unit/path in NALU
$iNAC_\cdot$	Multiplicative sub-unit/path in iNALU
ω	Learnable parameter for clipping the multiplicative weights
t	Learnable parameter for a regularisation loss

Chapter 1

Introduction

One piece that needs to be added to the machine learning landscape is the ability to work reliably on out-of-distribution data (OOD). Simple mathematical operations like $+ - \div \cdot$ are perfect for tackling this issue. The composition of such functions is called arithmetic compositional generalisation [2]. For instance, if a model learns to compute $(x1 + x2) \cdot (x3 + x4)$ on training data between 0-100, then we might be confident it will perform well for inference on data between 101 and 1000. Architecture that learns systematic generalisation for arithmetic computations was proposed in Neural Arithmetic Logic Modules (NALM) [11]. Since then, many variations and improvements have been proposed, out of which improved Neural Arithmetic Logic Modules (iNALU) [10] looks the most robust at the time of writing.

Nevertheless, training such novel architectures is complex and requires many custom actions during training. For this reason, it is not "plug-and-play" code that everyone can understand and run in a couple of minutes. The current implementation of iNALU relies on either graph-based TensorFlow 1 execution or a custom training code implemented in TensorFlow 2. We identified the hurdles related to building and training iNALU and wrapped most of them in a single callback that controls the training procedure.

We engineered and tested the versatile TenosrFlow callback to control the training of the feedforward AI architectures. Our work minimises the workload required to customise training procedures. Our controller converts custom training logic into the native TensorFlow functions acting directly on the model. We can think about it as the wrapper of the TensorFlow compiler that embeds our custom logic on the data-flow graph produced at the time of the model's compilation.

The proposed architecture is flexible enough to be successfully utilised outside the scope of NALM-based models.

The primary purpose of this work is to show and explain our novel approach towards custom training logic in TensorFlow. The user-facing training configuration file is easy to reuse, modify, and read, making models like iNALU much more attainable for AI newcomers. Moreover, we investigate and elaborate on iNALU architecture as an example of using the tool of our invention.

Chapter 2

Background

2.1 NALM - Motivation

Definition: A Neural Arithmetic Logic Module (NALM) is a neural network that performs arithmetic and/or logic-based expressions which can extrapolate to out-of-distribution (OOD) data when parameters are appropriately learnt whilst expressing an interpretable solution [7].

NALM has successfully incorporated various operations, such as arithmetic/logic-based operations, and selected relevant inputs. NALM is a supervised learned neural network architecture. The learning process involves selecting significant elements of the presented data and finding a combination of proper operations to produce the output. Neural Algebraic Logical Unit (NALU) architecture consumes data as vectors. Figure 2.1 depicts a high-level view of the working principle of NALU.

NALU is interpretable by its mathematical nature. To be more precise, NALU is transparently decomposable [5]. Arithmetic or logic-based operations were invented centuries before the rise of the first ML; therefore, it is the preferred way of the world description for scientists and engineers.

Due to the sparse and straightforward mathematics hidden in the successfully trained NALU, generalizable properties are of primary interest in the NALU architecture. This property makes NALU a perfect candidate for scenarios where OOD data can appear in production. Nevertheless, training NALU is complex, and sometimes simple equations cannot be derived to describe real-world scenarios

accurately. Therefore, NALU is usually part of the more extensive architecture [7].

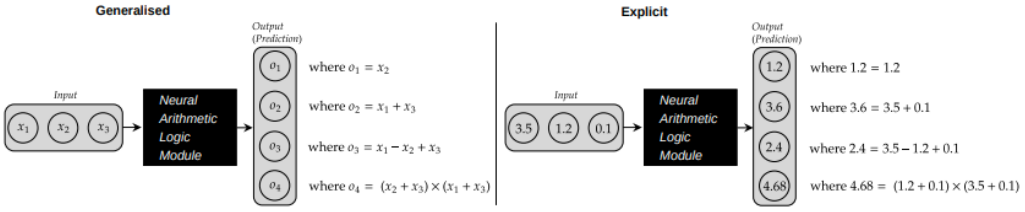


FIGURE 2.1: High-level example of the input-output structure of a NALM. Both networks are the same. The generalised network defines the notation of each element in the input and output. The explicit network is an example of valid input and output values [7].

Learning simple arithmetical operations might seem trivial compared to the super-human capabilities of RNN or CNN. Nevertheless, learning such simplistic representations of the problem by a network has been elusive until recently. In 2018 Trask proposed NALM [11], which proved that our ML's understating is mature enough to tackle problems in a way humans have been doing for centuries.

The idea is still relatively new and needs to be tested more extensively in real-world situations. To improve extrapolation abilities, NALU layers are currently attached to battle-tested architectures like CNN [8]. Our research found no information about using NALU in production environments. However, it might change with more research done in this niche field.

2.2 NALU

The structure of NALM proposed by Trask [7] is in Figure 2.2. It is a graphical representation of the mathematics governing the NALU layer presented below.

$$W_{i,o} = \tanh(\widehat{W_{i,o}}) \odot \sigma(\widehat{M_{i,o}}) \tag{2.1}$$

$$NAC_+ : a_o = \sum_{i=1}^{I} W_{i,o} \cdot x_i \tag{2.2}$$

$$NAC. : m_o = \exp(\sum_{i=1}^{I} W_{i,o} \cdot \ln(|x_i| + \epsilon)) \tag{2.3}$$

$$g_o = \sigma(\sum_{i=1}^{I} G_{i,o} \cdot x_o) \tag{2.4}$$

$$NALU : \hat{y_o} = g_o \cdot a_o + (1 - g_o) \cdot m_o \tag{2.5}$$

$\widehat{W}$ and $\widehat{M}$ are trainable matrices. Equation 2.1 induces bias towards value $-1, 0, 1$, which means selecting one of the paths. We can easily translate quantitative information into a qualitative understanding, another side of transparency. In multiplicative path 2.3 -1 is division, 0 is element ignorance, and $+1$ is multiplication. We combine both paths 2.5 via the gate mechanism 2.4. Ideally, the gate's value should be 0 (close) or 1 (open) when the training finishes. Then NALU will not switch between summative and multiplicative paths. To use both paths, we can stack two NALUs such that the output of one NALU is the input of another.

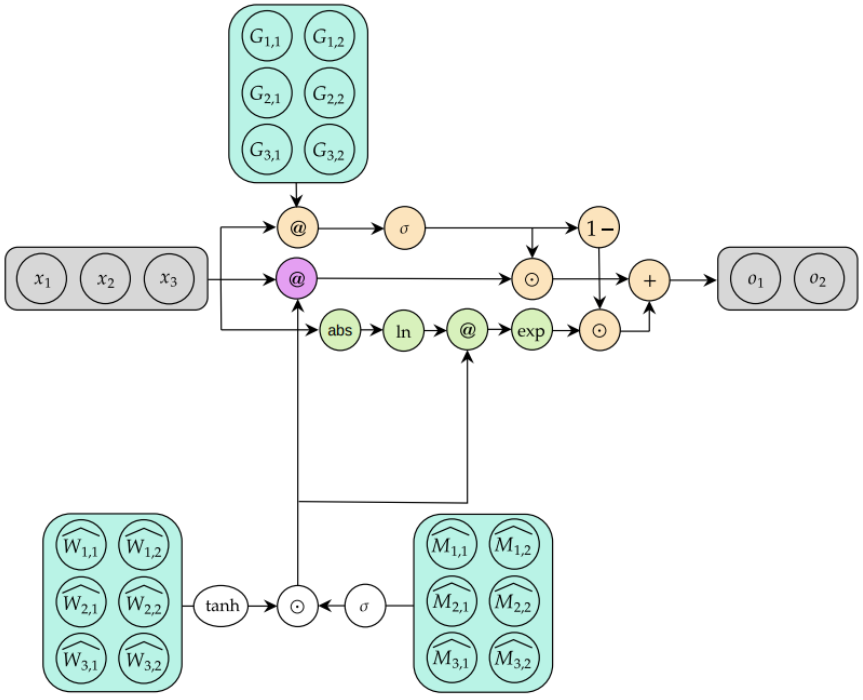


FIGURE 2.2: NALU architecture for 3-feature input and 2-feature output layer [7].

Such bold and novel ideas like NALU are inherently related to some challenges. Exploding intermediate results is one of the most significant problems in the above-mentioned architecture. It is evident when stacking NALUs to form deeper networks. Another hurdle of NALU is its inability to operate with data leading to numerically-negative outcomes. In the multiplicative path, calculations are shifted

to absolute log space. Therefore, it trims information about the sign of the presented data. Another issue is NALU's sensitivity to initialization, making the training success rate relatively low, as initialization depends on the problem and input data distribution. Lastly, the gating mechanism to switch between multiplicative and summation paths is leaky. In practice, g variables are often half-open (value around 0.5). It makes sense as a half-open gate can cover a broader range of outputs. It leads to memorizing data instead of learning the governing rules hidden inside the data.

2.3 iNALU

Improved Neural Arithmetic Logic Unit (iNALU) is one of many members of NALU-influenced models 2.4. It addresses many issues mentioned above. It was proposed in the form depicted in Figure 2.3 [10]. Our iNALU layer implementation is available in the project repository.

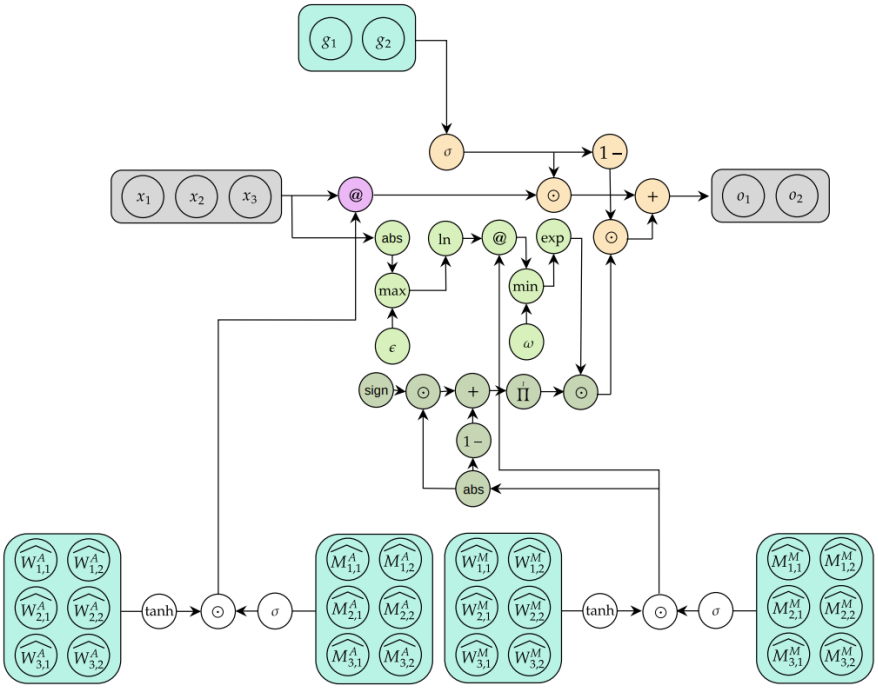


FIGURE 2.3: Architecture of the improved Neural Arithmetic Logic Unit (iNALU) [7].

2.3.1 Structural Improvements

INALU overcome some issues by weight independence. It separates trainable matrices for summative and multiplicative paths: equations 2.6, 2.7. A model with weights separated can learn optimal values for two paths independently and then select the correct path using a gating mechanism. For example, when equations governing the data rely on division, there will be a problem of small number division. It will empower errors coming from the leaky gate. In iNALU, W_m can be optimized to smaller values to mitigate the scenario mentioned above.

INALU solves issues related to exploding intermediate results in deep models by values clipping: equation 2.9. In the back transformation from log space, iNALU clips exploding weights by ω. Gradient clipping ϵ improves training stability, which may occur when values oscillate near zero.

NALU, proposed in the previous section, cannot operate with data of negative results in a multiplicative path. This structural limitation makes NALU fail to multiply and divide mixed signed data. iNALU resolves this issue by introducing multiplication sign matrix msv 2.11 that restores information about the output sign. If W contains values belonging to the $-1, 0, 1$ then msv_o values are discrete and belong to the $-1, 1$. The result of column multiplication of msm gives the sign vector msv 2.11 that can retrieve information about the sign from log-space transformation.

In many cases, the task determines which operation path is best selected. With such an assumption, the gating mechanism should not depend on the input data and should be in the vector form: equation 2.12. The most significant advantage of a vector over a scalar form of the gate is the ability to select the operation for each output independently. We can consider the gating mechanism as a separate part of the model that does not rely on the data but on the paths' performance.

$$W^A_{i,o} = \tanh(\widehat{W^A_{i,o}}) \odot \sigma(\widehat{M^A_{i,o}}) \tag{2.6}$$

$$W^M_{i,o} = \tanh(\widehat{W^M_{i,o}}) \odot \sigma(\widehat{M^M_{i,o}}) \tag{2.7}$$

$$iNAC_+ : a_o = \sum_{i=1}^{I} W^A_{i,o} \cdot x_i \tag{2.8}$$

$$iNAC. : m_o = \exp(\min(\ln(\max(|x_i|, \epsilon)) \cdot W^M_{i,o}, \omega)) \tag{2.9}$$

$$msv_{i,o} = \text{sign}(x_i) \odot |W^M_{i,o}| + 1 - |W^M_{i,o}| \tag{2.10}$$

$$msv_o = \prod_{i=1}^{I} msm_{i,o} \tag{2.11}$$

$$g_o = \sigma(G_o) \tag{2.12}$$

$$iNALU : \hat{y}_o = g_o \cdot a_o + (1 - g_o) \cdot m_o \cdot msv_o \tag{2.13}$$

2.3.2 Training Improvements

INALU works best if W, M, and g are discrete values. It is crucial as, without this property, iNALU will only approximate the solution. We can induce dynamic constraints, promoting such properties using the regularization technique given by the equation 2.15. According to the [10], $t = 20$ works sufficiently well. The form of the regularization 2.15 does not guarantee that gradient directions will work in the same direction as the gradient direction of the loss without penalty term. It is highly initialization dependant; therefore, regularization can be delayed till the loss falls below a certain threshold. Our experiments prove that regularization is crucial for iNALU to learn global patterns and fully exploit the network's arithmetic properties. The regularization effect is visible in Figure 6.7 as a considerable drop in loss value.

$$\mathcal{L}_{reg}(\omega) = \frac{\sum_o^O \sum_i^I \max(\min(-\omega_{i,o}, \omega_{i,o}) + t, 0)}{O \cdot I} \tag{2.14}$$

$$\mathcal{R}_{sparse} = \frac{1}{t} \sum_{\omega \in X} \mathcal{L}_{reg}(\omega) : X = \{\widehat{W^A}, \widehat{M^A}, \widehat{W^M}, \widehat{M^M}, g\} \tag{2.15}$$

NALU tends to get stuck in local optima. Therefore, iNALU overcomes this issue with a reinitialisation strategy. It inspects the losses from the last n-th epochs,

and if the averaged loss does not improve significantly and is still relatively high, then it randomly reinitialises all iNALU's weights

2.4 NALU Influenced Modules

The NALU model family is surprisingly diversified. For a comprehensive overview of this field, read section 4 of A Primer for Neural Arithmetic Logic Modules [7]. We will outline only a couple of models influenced by NALU besides iNALU, on which we elaborated above.

2.4.1 NAU and NMU

As was mentioned in section 2.2, NALU can either follow summative or multiplicative paths. Gating mechanism controls which path our model selects to fit the presented data. However, we can extract summative and multiplicative paths as single entities, Neural Addition Unit (NAU) and Neural Multiplication Unit (NMU), respectively [6]. Then there is no need for a gating mechanism that greatly simplifies the training procedure. We can use the field knowledge to use NAU or NMU intentionally. For instance, stock volatility prediction is an excellent example of where we can benefit from using NAU. On the other hand, NMU might not bring any advantages in this scenario.

2.4.2 NPU

Neural Power Unit (NPU) [4] aims to improve the division ability of the multiplicative path of NALU. Moreover, it models products of arbitrary powers. It tries to achieve it by taking a complex log transformation and using real and complex weight matrices. After such transformation, we can freely operate in the space of nondiscreet values of the learnable weights.

2.4.3 NLRL

Neural Logic Rule Layer (NLRL) [9] is a module to express simple arithmetic operations and boolean logic rules. We can express more sophisticated boolean

operators like implication or equivalence by stacking multiple NLRLs. The limitation of this module is the boolean range of the input data.

2.4.4 NSR

Neural Status Register (NSR) [3] learns to compare control logic: $<$, $>$, $!=$, $=$, $>=$, $<=$. We can perceive it as a module for quantitative reasoning about elements selection for comparison and the way of comparison. A NSR module returns two elements. The first element carries information about if the comparison is true or false, and the second element is the negation of the first. This mechanism aims to give downstream task access to information from the two possible scenarios of evaluations.

2.5 NALU Further Work

2.5.1 Division

At the time of writing, NPU 2.4.2 is the most robust architecture to work with division. Nevertheless, it still struggles with the division of small numbers. It was hypothesized that this issue could not be solved [6] due to the singularity of division by small numbers. An alleviation was proposed to return a flag indicating invalid output [9].

2.5.2 Robustness

One of the most significant advantages of NALU is its ability to extrapolate. A model should reliably infer while being trained on any input range. Unfortunately, achieving this is challenging when we train a model on a relatively small range of data. It was first observed and described in work [6], which investigated the training failure of stacked NAU and NMU. It failed to generalise if trained on a small range of values.

2.5.3 Compositionality

Well-trained NALU models should be versatile enough to tackle the selection of various types of operations and to model complex expressions. NALU's ability to manage composition relies on the working principle of either gating or stacking. The gating mechanism can select operations from a bigger group, but it leads to convergence issues. On the other hand, stacking works more reliably, but it limits the number of operations it selects.

Chapter 3

Training Procedure - Overview

3.1 TensorFlow 2 - Training Loop Revision

The contemporary distribution of TensorFlow hides most of the technical hurdles in compact functions such as Model.fit(), Model.evaluate() and Model.predict(). The biggest advantage of using these functions is the code compactness, readability, and utilization of more advanced functionalities like distributed training or TPU acceleration.

For most users, training feedforward neural networks in TensorFlow 2 can be as easy as in Figure 3.1. Passing more parameters like **validation_split** to the *Model.fit* function, we can easily incorporate new logic inside the training loop 3.2.

TensorFlow does all the heavy lifting for us. It builds a graph of computation for speed, runs automatic differentiation, calculates gradient, updates trainable weights, preprocesses data (for instance, runs vectorization and batching), or configures metrics tracker, loss tracker, or updates progress bar at the time of training.

Nevertheless, such a user-friendly API can only serve a limited number of models with a straightforward training procedure. If a user wants to customize what happens inside the training loop, we must implement the whole training loop from scratch. It is not very demanding, but the code readability and availability of advanced features like the abovementioned distributed training deteriorate significantly. The proposed solution fills the gap between the default training procedure and hand-crafted custom training loops. We elaborate on it in the Section 3.2 *TensorFlow 2 - Training Loop Redesign*

```
# load data to the local scope
tf_data = tf.data.Dataset.from_tensor_slices((data, labels))

# define model architecture
model = tf.keras.Sequential([
    tf.keras.layers.InputLayer(INPUT_SHAPE),
    ...,
    tf.keras.layers.Dense(OUTPUT_SHAPE)
])

# compile model setting loss function and metric
model.compile(optimizer=
    tf.keras.optimizers.RMSprop(learning_rate=0.01),
    loss="mse",
    metrics=["mae"])

# train model on batches given by batch_size
# for EPOCHS_NUM epochs.
history = model.fit(
    tf_data,
    batch_size = BATCH_SIZE,
    epochs = EPOCHS_NUM
    )
```

FIGURE 3.1: Following the schema of data loading, model definition, model compilation and model training, we can fully utilise the potential of default TF functionality for training.

```
# use 20% of data for validation
history = model.fit(
    tf_data,
    validation_split = 0.2,
    batch_size = BATCH_SIZE,
    epochs = EPOCHS_NUM
    )
```

FIGURE 3.2: Adding seemingly complex logic to the training procedure, like validation check, is as easy as passing a value to one of *tf.fit* arguments.

3.1.1 Eager vs Graph Execution

The major shift between the first and second generations of TensorFlow is the chance to execute code eagerly. In practice, it is an environment that evaluates operations immediately. From the user's perspective, the whole ecosystem works as a standard set of Python classes. To be more precise, in eager execution, TensorFlow execute one operation after another in the native Python environment. It makes code and execution intuitive and easy to debug. Nevertheless, in such an execution mode, we cannot incorporate much of the performance optimisation.

An alternative execution model is called *Graph* execution. It builds and optimises the dataflow graph before evaluation. It is a directed graph where nodes represent operations, and direct edges represent tensors. An example of such a graph is in figure 3.3. Models in such a form can be profoundly optimised and deployed easily to various services (e.g. TensorFlow light for embedded devices). For instance, graph compilers consider the system's physical limitations to which the graph is optimised (e.g. number of threads, cash, or GPU/TPU support). These improvements come at the cost of a bunch of technical constraints and a slower pace of development. Luckily **tf.function** is a bridge between these two models of execution that lets us use the best out of these two world.

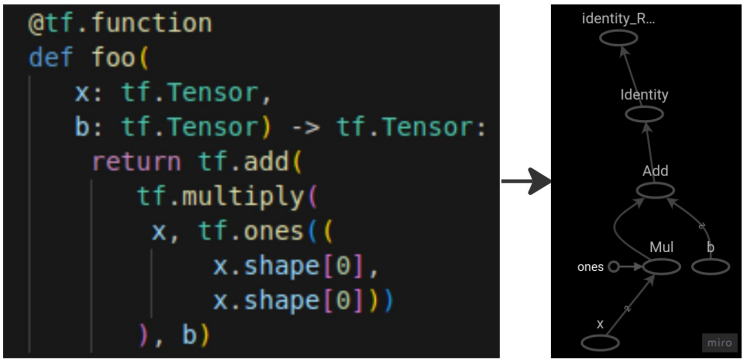


FIGURE 3.3: The ordinary Python function can be transformed into the TensorFlow graph. All of these happen in a very convenient way by **@tf.functoin** decorator.

3.1.2 TensorFlow Function

The TensorFlow function lies between the human-readable code and compiled file the computing engine consumes. Tf.function makes graphs out of our code. It transforms it into Python-independent dataflow graphs. It makes our code portable, reproducible, and deployable. Understanding the TensorFlow function is crucial to understand most of the technical aspects of TensorFlow Training Control proposed in chapter 4.

Graph building procedure by **tf.function** starts from tracing. In general, python functions incorporate polymorphism which prevents **tf.Graph** from being built due to the numerous possibilities related to the different types of function arguments. For this reason, traceable functions must have all arguments of the native TensorFlow type. Nevertheless, in practice, this rule is relaxed. Tracing scans a function's body, and all TensorFlow operations are deferred to the captured

tf.Graph. When a non-TensorFlow native type appears in the function declaration, part of the not-deferred function is retraced on each call. Successful tracing captures a callable graph of optimised computations for better performance. It is essential due to the hardware-native accelerations like TPU or distributed strategies in computing hubs. Nevertheless, it comes at the price of meticulous engineering of function bodies that can be safely converted into dataflow graphs.

Generally, **tf.functions** would not have access to global and free variables. Another limitation relates to working with Python objects. There is a workaround for creating custom *TraceType*, but it is still in the experimental phase. It explains why we first evaluate and capture values of function passing to the configuration file inside our code. This issue is elaborated in the section 4.1.5.

Another **tf.function** technicality we used in our project referees to the Python *if* condition. Tracing it would capture only a single branch, which might confuse and make debugging cumbersome. Suppose we want to build conditional expressions inside **tf.function**, then we must use **tf.cond**, which is the native-TensorFlow *if* counterpart. **Tf.cond** must return callables from both branches; otherwise, it will raise an error for dataflow graph execution. Similarly, the predicate must be of the native-TensorFlow type. It explains the form of the capturing **slave_steps losses** in **training_step**: section 3.2.2

3.2 TensorFlow 2 - Training Loop Redesign

3.2.1 Control Variables

The TensorFlow team followed the best programming practice and built TensorFlow as the decoupled set of components that improve readability, maintenance, testing, and simplifying the logic. Nevertheless, it also implies that from the perspective of a TensorFlow model and **train_step** (see section 3.2.2), we have no access to the variables/data related to the training itself. Therefore, we started engineering our Train Control by building coupling between the default TensorFlow training procedure (see **tf.fit** at section 3.2.2) and a model's state. We aimed to find a generalisable way to project logic related to training procedures on the model's training. We can build training predicates using epoch numbers, batch numbers, or training loss history. The projection of them onto the model can control its behaviour. It circumvents the decoupling between the TensorFlow

model and TensorFlow training. It allows controlling the model training from the position of the training loop still using default **tf.fit** function.

Our solution is to add non-trainable boolean variables to the model instance, which can be incorporated into the dataflow graph. An engineer might ask, why do we need **tf.Variable** to keep a single boolean value? As discussed in section 3.1.2, TensorFlow will compile all non-variables attributes like tensors or native booleans into static values in the dataflow graph. It prevents us from controlling what happens inside the model as the whole logic would be captured from its state at tracing time. For this reason, we assign non-trainable boolean Variables to the model's instance that are not-captured as static values inside **tf.functon**. Another advantage of this architecture is the accessibility of such control variables due to their exposure as an ordinary attribute of the model instance.

3.2.2 Master Step

To control the training procedure, we must start by understanding **train_step** or, more precisely, **tf.keras.model.Model.train_step**. As the name suggests, it implements mathematical logic for one training step. It usually includes the forward pass, loss calculation, backpropagation, and metric updates. **Train_step** is utilised by **tf.fit** function that cope with the training loop for us. It has enormous numbers of parameters that can easily control things like batching, data shuffle, validation, distributed training, or multiprocessing. **Tf.fit** iterate through batches of data and call **train_step** on each batch. Therefore, if we can control the logic of **train_step**, then we can also control **tf.fit** and what follows the whole training procedure.

Train_step comprises multiple smaller steps implementing the training logic. For this reason, we rearranged **train_step** into the master control step that conditionally calls **train_substep** what we call **slave steps** (naming convention from a communication protocol). Conditional executions are manipulated by the control variables, which is explained in figure 3.4. Similarly, the graphical representation of this process is depicted in figure 3.7.

```python
@tf.function
def train_step(self, data):
    slave_loss = {
        "loss_gate": tf.cond(
            # control variables for gating
            self.control_variables["gate"],
            # train_substep called for positive
            # evaluation of the condition
            lambda: self.gate_on(data),
            # train_substep called for negative
            # evaluation of the condition
            lambda: self.gate_off(data),
        ),
        # remaining conditions
    }
    # remaining train step logic
```

FIGURE 3.4: *Slave-loss* is a map of substeps names (in this case, it is gating action) to the related losses. The master step will call **self.gate_on** or **self.gate_off** what is controlled by the control variable stored in **self.control_variables** under *gate* key.

3.2.3 Slave Steps and Training Substeps

Slave_steps are the building blocks of the training logic. Each slave step consists of three elementary pieces: control variable, **train_substep** called when the control variable is true, and **train_substep** called when the control variable is false. On the figure 3.4 **self.gate_on** and **self.gate_off** are **train_substeps**, while their combination with *tf.cond* is a single **slave_step**.

Each **train_substep** implements similar logic to the default **train_step**, creating *GraidentTape*, calculating losses, updating model weights or updating metrics like in the figure 3.5. From the position of our train control, we can control what happens inside a slave step. It is the unobvious advantage of first generating a function as a string and then binding it to the object instance. It is possible due to the effortlessness of string manipulation in the function body generator. We elaborate on this in the section 4.2.1.

3.2.4 Training Control Callback

TensorFlow Callbacks is a functionality that can be hooked into the various stages of the model training and inference lifecycle. As it is the only part of the training procedure that is aware of the information, such as the number of epochs since

```python
@tf.function
def gate_on(self, data):
    x, y = data
    with tf.GradientTape(watch_accessed_variables=True) as tape:
        # forward pass
        logits = self(x, training=True)
        # loss calculation
        loss_value =  loss(y, logits)
    # gradient calculation
    grads = tape.gradient(loss_value, tape.watched_variables())
    # updating models weights
    self.optimizer.apply_gradients(grads, tape.watched_variables())
    # updating metrics
    self.compiled_metrics.update_state(y, logits)
    return loss_value
```

FIGURE 3.5: Each **train_substep** performs an independent training update for the model. Combining multiple such steps inside the master step allows us to control how the model is updated depending on the state of control variables.

the beginning of training, it is the perfect place to start engineering our training control. TensorFlow Callbacks are single-standing entities hooked to the model's training loop. From their position, we can call their internal function at various stages such as: **on_epoch_end** or **on_predict_batch_begin**. What is important from the position of the callback, we have access to the instance of the model that the callback is hooked. We can access the model instance via **self.model**. What is essential, the model is assigned to the callback instance at the beginning of the training loop to which the callback is hooked. For this reason, all the logic acting on the model must be executed inside **on_train_begin** instead of the constructor of the callback.

To make the instance of the model follow our custom training procedure, we have to rebuild the model instance from the callback position. First, we assign control variables to control **train_substeps**. Then we generate **train_substeps** and new **train_step** function bodies in the form of strings. The generated function bodies are in the same form as if they are functions of the model instance. We execute these functions to bring them into the local scope of **on_train_begin**. Then we bind them to the model instance. We elaborate more on this in the section 4.2.1. Ultimately, we compile the model that makes it ready to enter our custom training logic. All steps before training are depicted in figure 3.6.

iNALU implementation relies on the reinitialisation technique. INALU is susceptible to being stuck in local minima. Therefore, if we detect it, we can reinitialise the models' weights to start training again. We added an extra argument to inject this logic into the training procedure. Similarly to the scheme above, we include a map of actions, each holding a callable condition and function binding to the

model instance. This function is then called on each positive evaluation of the condition. By now, conditions evaluate at the end of each training epoch.

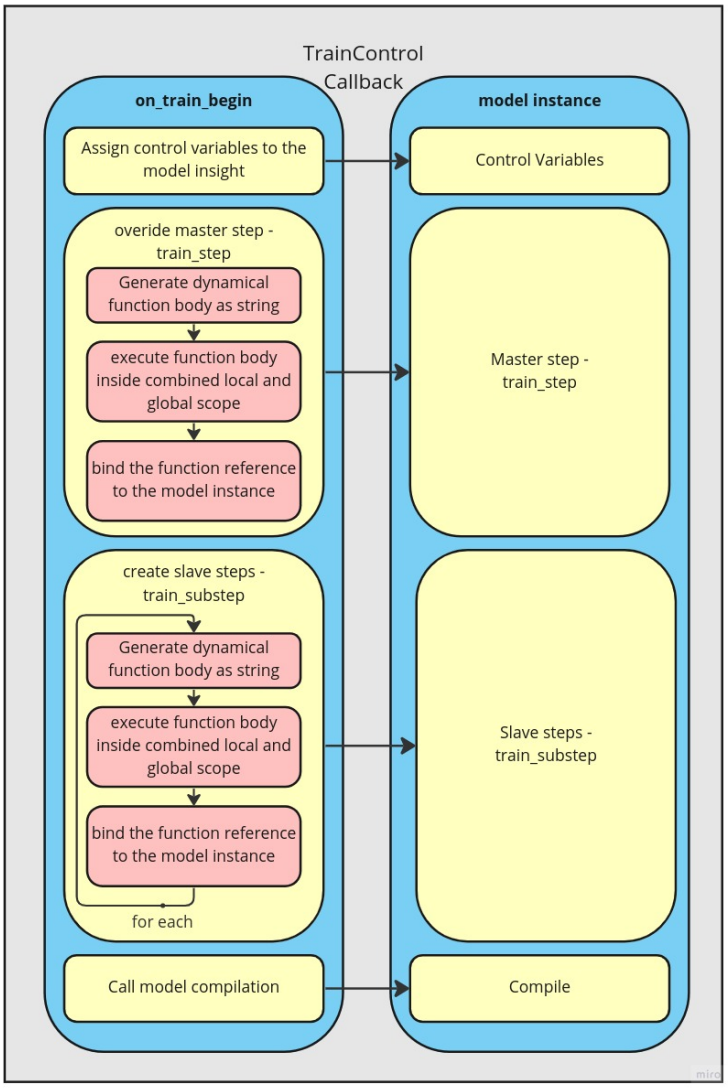


FIGURE 3.6: We must remember that both Train Control Callback and the model instance are consumed inside the body of the **model.fit** function. Therefore, we can consider callback as a **model.fit** argument. Before our model starts learning, we configure it in 4 steps according to the values passed as the constructor arguments of the callback. First, we assign control variables to the model instance from the callback position. Then we generate, execute, and bind the master and slave steps. Eventually, we recompile the model using the same compiler and compiler's setting as at the time of model compilation before calling **model.fit** function.

After the execution of **on_train_begin** TensorFlow starts to iterate over data batches of data and calls **train_step**. As the training progresses, we must update the values of control variables assigned to the model instance. Setting values can be freely controlled from the callback position utilizing the whole function branch

like **on_epoch_end** or **on_predict_batch_begin**. A TensorFlow callback is not transformed into the dataflow graph; therefore, all restrictions mentioned in the section 3.1.2 are relaxed. Figure 3.7 shows a high-level overview of the training procedure for a model with train control callback. For a proof of concept, see appendix B.

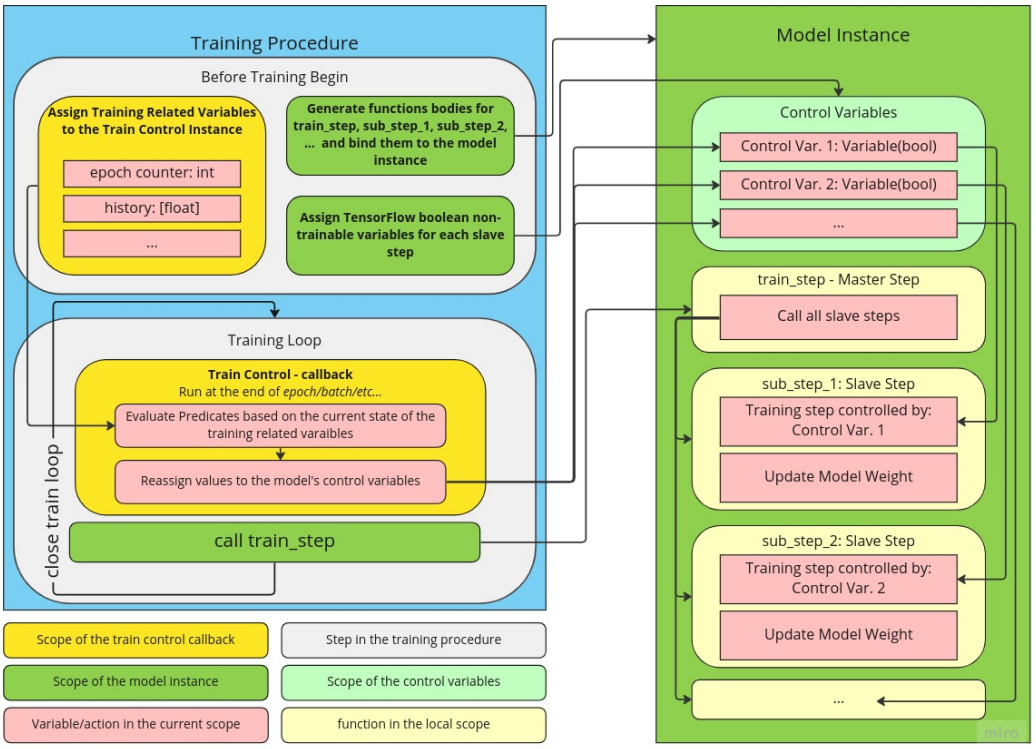


FIGURE 3.7: **Before Training Begin** referees to the sequence of steps depicted in the figure 3.6. Three things happen sequentially when the training loop starts iterating over batches (**Training Loop** rectangle). First, training control predicates are evaluated based on the training state (e.g. epoch, batches, or loss). Then we assign these values to the model control variables. The model's **train_step** step is called at the end of the single interaction step. We can spot that the state of the model's control variables controls **train_step** that aggregate calls to all salve steps. This way, we superimpose training logic to the model instance, unaware of the training progress. We can consider it a small coupling between the model and the models' training or a dynamical logic projection onto the model.

Chapter 4

Training Procedure - Implementation

4.1 Configuration of Training Control Callback

We decided to outsource the training logic into the Python dictionaries used as the argument of train control callback. The main reason for such an architectural decision was the convenience of passing Python callables. To understand the logic behind the keys of the configuration dictionary, we need to analyse the elementary blocks of the supervised training steps. Each training step consists of the following:

- auto-differentiation configuration
- forward pass
- computing loss
- calculating gradient
- modifying gradient - optionally
- updating model weights

Therefore, by controlling the abovementioned parameters, we can reach sufficient flexibility of training control for feed-forward architectures.

4.1.1 Tracking Values by Callback

By default, training loop callback track four meta attributes listed in the figure 4.1. The attributes should be self-explanatory. **History** saves the values of all losses and metrics along with the states of control variables. We can build control functions based on these values. An example **gate_open** is in the figure 6.3.

```
self.epochs: int = 0 → epoch counter from the training begin
self.batches: int = 0 → global batch counter from the training begin
self.batches_in_epochs: int = 0 → counter of batches in each epoch
self.history: List[Any] = [] → save state of training on each train step
```

FIGURE 4.1: Four meta attributes tracking by the training loop callback at the end of each training step.

4.1.2 Configuration File

The training procedure is controlled via a Python dictionary. At the top level, we have a map of action names, represented by strings, to another dictionary with the configuration for this particular action. As explained in section 3.2.3, an action with its internal configuration can be considered a slave step. An example of configuration with two slave steps: *action_1* and *action_2* is on the figure 4.2.

```
config = {
    # dictionary holding training configuration
    # for steps taken in action_1
    "action_1": action_1_config,
    # dictionary holding training configuration
    # for steps taken in action_2
    "action_2": action_2_config,
    # more actions
    }
```

FIGURE 4.2: The high-level view of the configuration with two actions (two slave steps according to the naming convention from the previous chapter). Both **action_1_config** and **action_2_config** are dictionaries explained below.

Action_1_config and **action_2_config** must be Python dictionaries with specific keys. Each configuration dictionary must have at least a map from **"cond"** to python callable returning bool and either **True** or **False** or both keys mapping to another configuration dictionary 4.3.

The sections below explains **action_1_config_true** and **action_1_config_false**. For a complete example, go to the section 4.9.

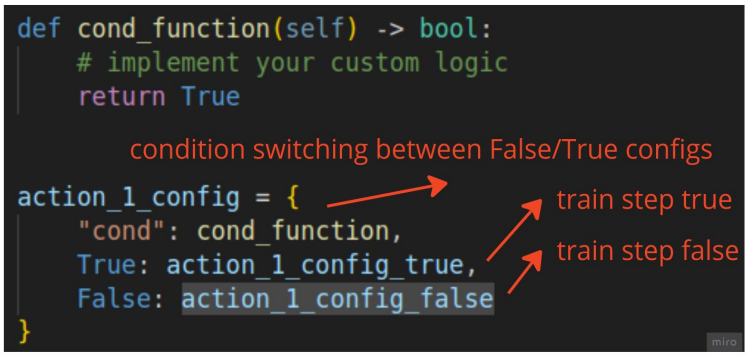


FIGURE 4.3: Callable passed to the **"cond"** binds to the callback instance and is executed from that position. Therefore, we use self as the initial argument, and we have access to tracking variables inside the scope of this function. Values passed to **True** and **False** are Python dictionaries explained below.

4.1.3 Loss Functions

To control the loss function, we can either pass a callable TensorFlow loss instance, leave it blank, or not include it. It is a straightforward solution; nevertheless, dynamically generating and binding such a loss function is not trivial. We elaborate on this in section 4.2.1. Figure 4.4 explains how we can control loss used in **train_substep** from the position of configuration passed as the argument of the train control callback.

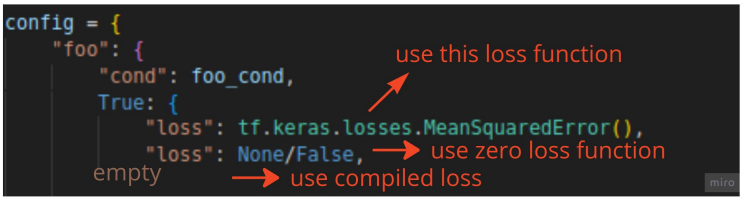


FIGURE 4.4: We can freely customize the loss function by passing an instance of an object subclassing **tf.keras.losses.Loss**. If we explicitly pass **None** or **False**, then the zero loss function would be used, meaning that training would not rely on the presented data. We will use compiled loss if we omit **"loss"**.

4.1.4 Regularisation

Regularisation is not included in the training procedure on its own. It is part of the TensorFlow layer. Therefore, from the position of the training control, we can only switch it on or off globally. Nevertheless, extending control to individual layers is the obvious extension of our work. A code example of including and excluding regularisation loss in **train_substeps** is in the section 4.9.

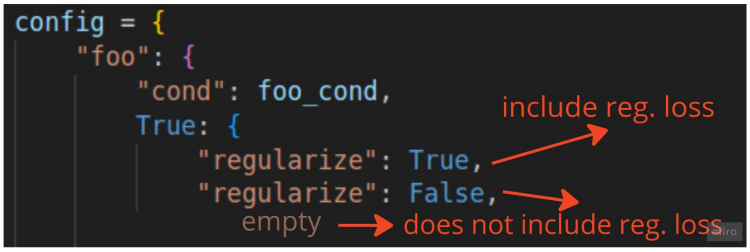


FIGURE 4.5: Regularisation can be either on or off. It adds regularisation loss for gradient computation. It happens independently from the loss function.

4.1.5 Limiting Trainable Variables

For the versatility of train control callback, we need to control variables passed to the Gradient Tape that ensures auto-differentiation. In most cases, we want to update all trainable weights. Nevertheless, for iNALU training (it inspired this work), we want to switch between two disjoint subsets of trainable variables constantly. We elaborate on the details of iNALU training in section 6.2. We cannot pass an explicit array of variables due to the nature of code execution in TensorFlow. In TensorFlow, callbacks are interpreted before the model-building phase. Therefore, creating an explicit array of variables results in an error because variables still need to be initialised at this stage of execution. The workaround is to use external callables pointing to the function written in a way as if they belong to the model instance. We elaborate on this in section 4.2.1.

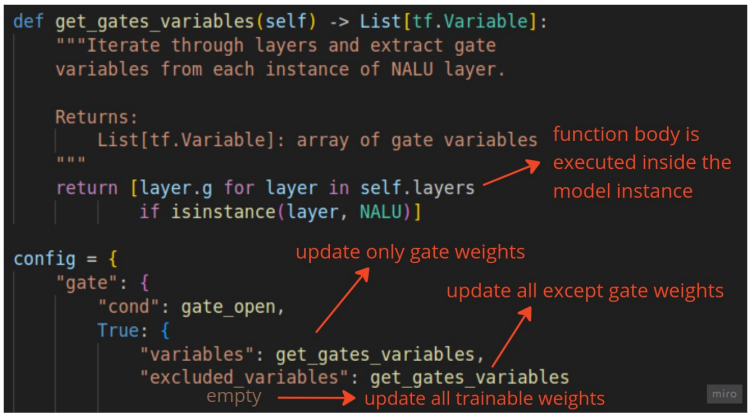


FIGURE 4.6: Adding a callable returning array of the model's weights to **"variables"** makes a training run only on the variables from the array. Logically, the same array passed to **excluded_varaiables** updates all trainable weights besides those from the array. Not specifying either makes the **train_substep** update all trainable weights. The peculiarity of the function returning the array of variables is explained in the section 4.2.1. To start using train control callback, writing this function as if it is a function of the model instance is enough.

4.1.6 Gradient Clipping

To complete the scope of the training step listed in 4.1, we need to control how to modify the values of the calculated gradient. An example of such modification can be gradient clipping, where we clip the error derivative according to a threshold. When **clipping** is present, **tf.clip_by_value** modifies the gradient according to the passed tuple. The natural extension to this idea would be to extend the range of gradient modification to work on callables with specifying function signatures.

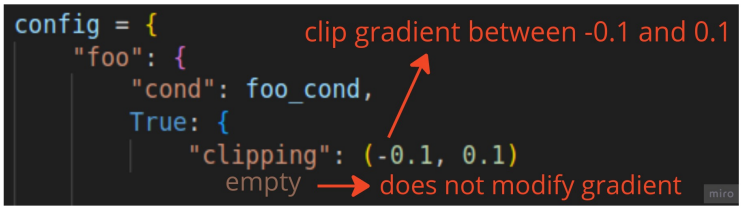


FIGURE 4.7: **Clipping** pointing to the tuple of floats modifies gradient using **tf.clip_by_value**. If **clipping** is omitted, then the gradient is not modified.

4.1.7 Asynchronious Configuration File

Training procedures sometimes require model modification outside of the main training loop. For instance, we might reinitialise the model's weights if we detect it stucks in the local minima or reach a plate. Similarly, we can think about early stopping supported by the **tf.keras.callbacks.EarlyStopping**. We want to have control over such functionality. Therefore, in addition to the training configuration explained in the section 4.1.2, we have a similar configuration to control the model out of the training loop. Configuration of this behaviour is depicted in the figure 4.8.

4.1.8 Extended Example

This example 4.9 shows how to configure custom training control. For the meaning of each argument, read explanations pointed out by arrows.

FIGURE 4.8: Similar to the configuration file from section 4.1.2, we rely on a map from strings, representing action name, to dictionaries with callable **"cond"** and two keys **"model"** and **"callback"**. Callables from these two keys bind to the model and callback instance, respectively. These callables are executed only at the positive evaluation of the callable pointing by **"cond"**. Condition is evaluated at the end of each training epoch.

4.2 Technical Difficulties

4.2.1 Binding Dynamically Created Functions

We first generate a function body as a string, then execute it in the local scope, and finally bind the function to another instance. It is very versatile due to the effortlessness related to string manipulation. Nevertheless, the execution of the function in the local scope is challenging because this function does not belong to the local scope but to a different instance. When the function body is interpreted by the python command **exec**, it needs access to global objects like packages and their aliases and to the local objects like a loss function. It explains why controlling values in the local scope can manage the train substeps without using unique function names.

We can execute the function first and then bind it to another object using the properties of Python code execution. Python is interpreted instead of compiled

FIGURE 4.9: A simple example shows the possibilities of training loop customisation via configuration files. Conditional functions can be considered to bind to the instance of control callback (as they have access to the attributes it tracks). Functions working with variable selection bind to the model instance, having access to the model's internal properties.

language; therefore, it does not check run time errors when it binds the function to the local scope. For instance, we provide package aliases without worrying about the execution object's methods. Exploiting this feature, we can freely re-bind functions within the Python code.

On figure 4.10, only a loss is added to the **train_substep**. The whole string generation is much more complicated and can be inspected in the appendix C.

4.2.2 Selecting Subsets of Variables for Slave Step

Due to the compilation nature of **tf.function**, we need to use particular architecture to modify variables passed to the **GradientTape**. As depicted in figure 4.6, we pass callables to return the array of model weights instead of explicitly passing these arrays. The reason standing behind it is the dynamic building of TensorFlow models.

Layers and their variables are initialised on the first call on a batch of data. Therefore, we cannot pass model weights directly to the callback configuration,

FIGURE 4.10: First, declare and extend the local scope with the **train_substep** configuration. We must use the double asterisks operator (**) to inject configuration as local scope objects. **Fn_config** contains *"loss"* key pointing to the callable in the local scope. It is interpreted later as the **loss** object representing the TensorFlow loss function. Then, we generate and execute a function body using the python **exec** function. It is important to pass concatenated global and local scope as the background for execution (from global scope, we acquire Python modules with their aliases, and local scope provides things like a loss function). The execution result is saved in the local scope from which we withdraw it to bind to the model instance.

as these steps happen before the training begin. Nevertheless, we can postpone it using callable returning an explicitly defined array of variables. Nevertheless, we cannot use the callback directly inside the TensorFlow function because it cannot be traced, causing TensorFlow to crush or trace back on each call. To overcome this, we can call the callable to get an array of variables and store them in the model instance attribute. TensorFlow function tracing will interpret these as static values and explicitly incorporate them in the dataflow graph.

As it was explained in the section 4.1.5, we can use **variables** and **excluded_varaibles** to either define variables passed or excluded from the **GradientTape**. In the case of **variables**, we execute callable and save the result as the attribute to the model

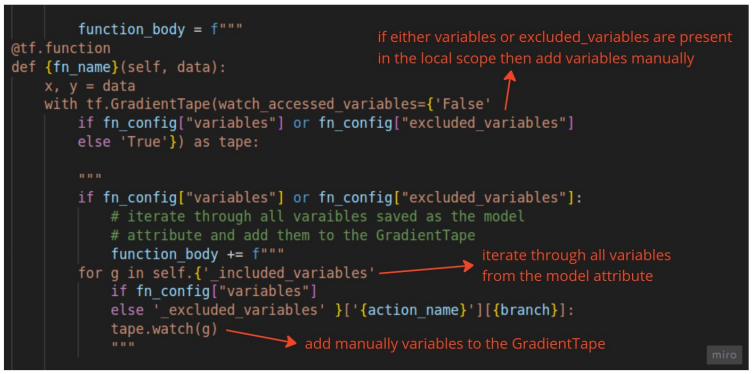


FIGURE 4.11: If either **variables** or **excluded_varaibles** is present in the train_substep configuration, we iterate through variables and manually add them to the GradientTape. Depending on the present key, we switch between **self._included_variables** and **self._excluded_variables**. They store arrays of variables defined by **variables** and **excluded_varaibles**, respectively. Using the action name and boolean flag, we can pinpoint if we are in the positive and negative evaluation of the slave_step condition.

instance. For **excluded_varaibles**, we iterate through all trainable weights of the model, saving all weights not present in the result of the callable. To keep model attributes clean, we store all arrays of variables in two dictionaries, one for **variables** and one for **excluded_varaibles**. The scrupulous inspection of variables arrays in the appendix C and their practical reference is in the figure 4.11.

4.2.3 Optimiser Build

Error Message

KeyError: 'The optimizer cannot recognize variable nalu/g:0. This usually means you are trying to call the optimizer to update different parts of the model separately. Please call ‘optimizer.build(variables)‘ with the full list of trainable variables before the training loop or use legacy optimizer ‘tf.keras.optimizers.legacy.self.__class__.__name__.’

The last step before training begins is model recompilation (see figure 3.6). Due to the structural changes, we must recompile the model to be aware of the variables and new training steps. Otherwise, the error message would display an error (like the one above) that the optimizer is unaware of the new model's variables.

Chapter 5

Testing

5.0.1 Variables Inclusion

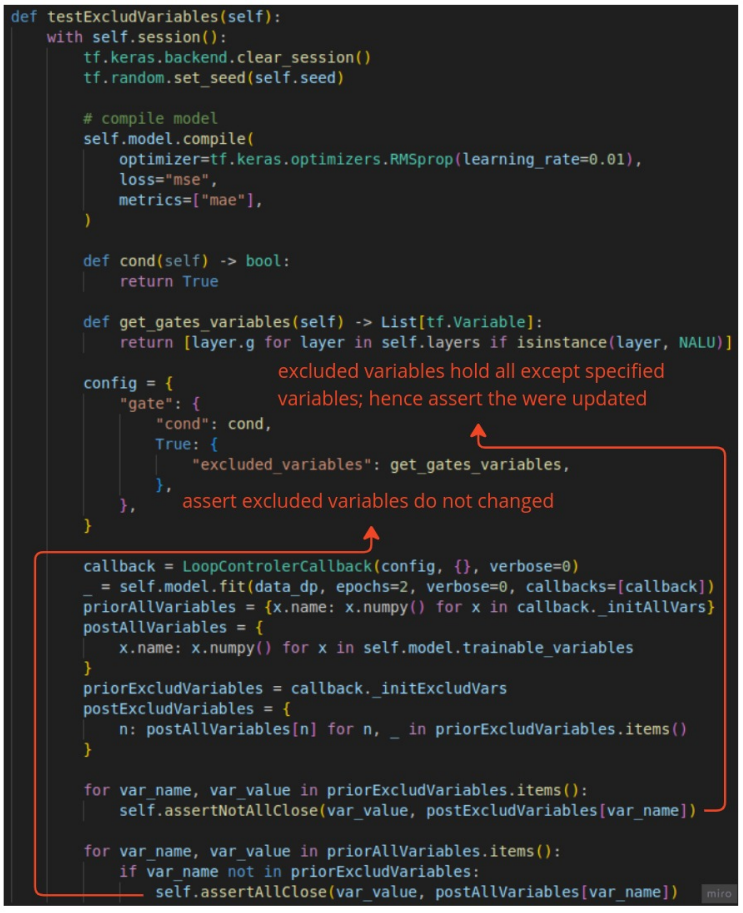
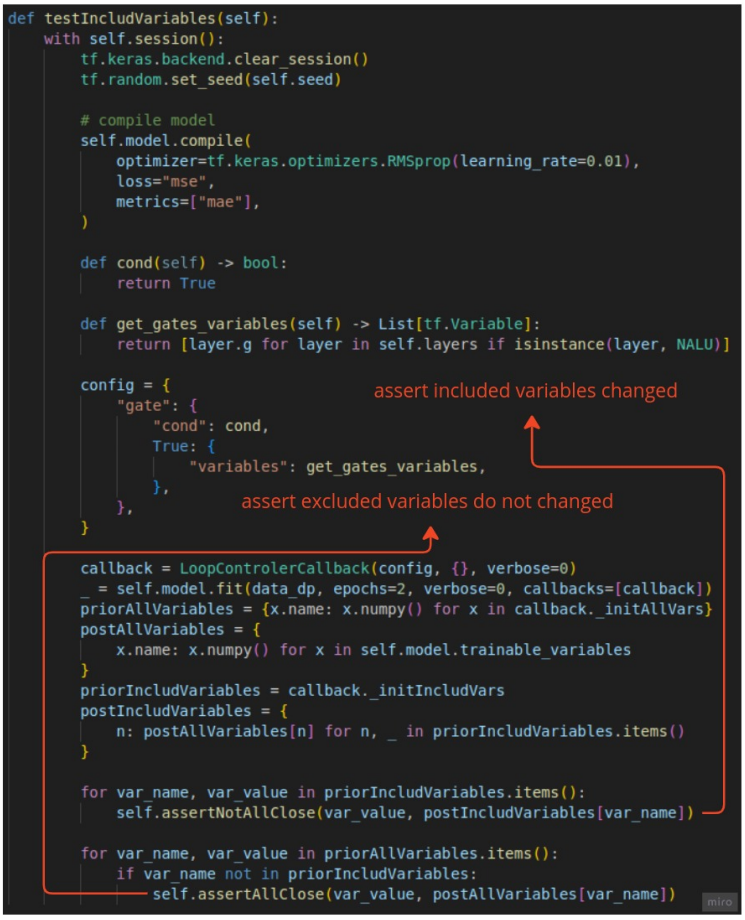


FIGURE 5.1: We tested the variables training inclusion via their comparison with the same variables' values after multiple training steps.

5.0.2 Variables Exclusion




```
def testIncludVariables(self):
    with self.session():
        tf.keras.backend.clear_session()
        tf.random.set_seed(self.seed)

        # compile model
        self.model.compile(
            optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.01),
            loss="mse",
            metrics=["mae"],
        )

        def cond(self) -> bool:
            return True

        def get_gates_variables(self) -> List[tf.Variable]:
            return [layer.g for layer in self.layers if isinstance(layer, NALU)]

        config = {
            "gate": {
                "cond": cond,
                True: {
                    "variables": get_gates_variables,
                },
            },
        }

        callback = LoopControlerCallback(config, {}, verbose=0)
        _ = self.model.fit(data_dp, epochs=2, verbose=0, callbacks=[callback])
        priorAllVariables = {x.name: x.numpy() for x in callback._initAllVars}
        postAllVariables = {
            x.name: x.numpy() for x in self.model.trainable_variables
        }
        priorIncludVariables = callback._initIncludVars
        postIncludVariables = {
            n: postAllVariables[n] for n, _ in priorIncludVariables.items()
        }

        for var_name, var_value in priorIncludVariables.items():
            self.assertNotAllClose(var_value, postIncludVariables[var_name])

        for var_name, var_value in priorAllVariables.items():
            if var_name not in priorIncludVariables:
                self.assertAllClose(var_value, postAllVariables[var_name])
```

FIGURE 5.2: We tested the variables training exclusion via their comparison with the same variables' values after multiple training steps. In the scope of the GradientTape, we cannot directly remove variables from the tape. For this reason, we simulate such functionality by filtering out excluded variables from the same trainable variables. It explains why the code structure is similar to the one in figure 5.4. We assert that the values of excluded variables (remember the reverse logic here) are not similar to those before training. Similarly, we assert that non-excluded variables have similar values to their state before training.

5.0.3 Loss Functions

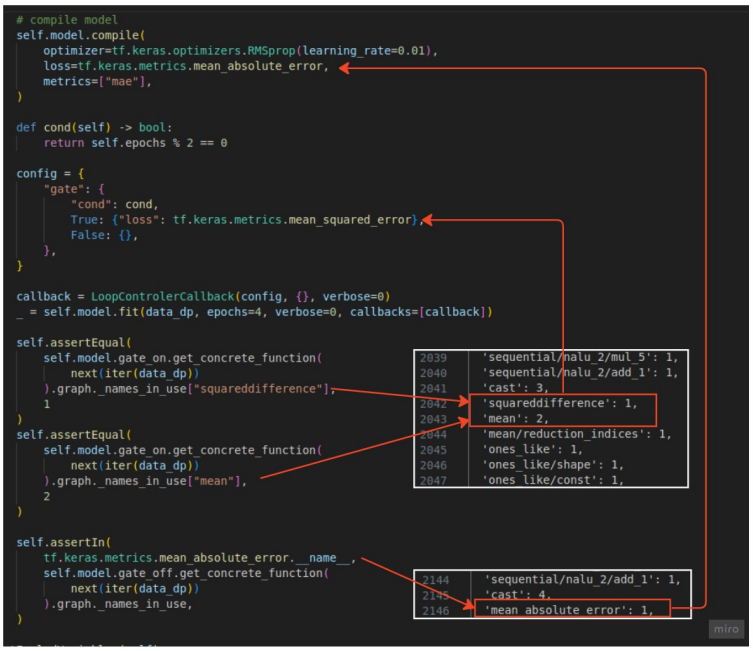


FIGURE 5.3: Unfortunately, testing the loss function used in the data-flow graphs is not straightforward. TensorFlow compiler brakes down loss function into fundamental steps. For this reason, we have to lookup for all components of the loss function being stacked together. The situation is different when we use compiled loss function. Then we can directly search for a loss function class name.

5.0.4 Control Variables Reassignment

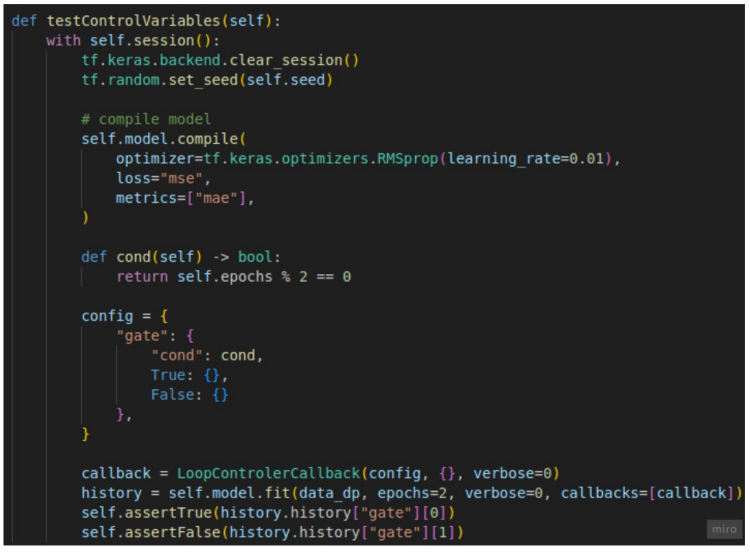


FIGURE 5.4: We inspect values of control variables returned as part of the tracked metric by the training model. Even though the default logger for training progress counts epochs from 1, we decided to follow the convention of counting from 0.

5.0.5 Regularisation

FIGURE 5.5: We were unable to automate regularisation testing reliably. We inspected the compiled graph by hand. For details inspect the graph in GitHub.

5.0.6 Gradient Clipping

Unfortunately, we cannot easily depicts changes done by the gradient clipping. The results of our inspection can be found here.

Chapter 6

Results

Code is available in the GitHub repository.

6.1 Prerequisites

6.1.1 Datasets

We decided to train our iNALU model on the Arithmetic Dataset Task [6]. We generated three datasets, each of the size 64000 samples. One for training and two for evaluation of extrapolation and interpolation performance. Interpolation and extrapolation datasets are drawn from a distribution with a different value range. We used range $(-3, 3)$ and $(10, 15)$ respectively.

6.1.2 Arithmetic Task

We designed a task to test the composition and selection capabilities of a deep iNALU model. We encoded the equation: $(x_1 - x_2) \cdot (x_3 - x4) + (x_5 \cdot x_6)$ in the vector of 7 elements. The composition of this equation is depicted in figure 6.11.

6.1.3 Training

We used the RMSProp optimiser in mini-batch training with a batch size of 64 and a learning-rate of 0.01. Using our training control callback, we attach a

reinitialisation strategy to reinitialise the trainable weight of iNALU layers in the model. Reinitialisation occurs when the model loss exceeds a threshold after 10000 training steps. Early stopping terminates the training when the average of the last 3000 training step is below a certain threshold. This configuration is illustrated in the figure 6.5. We elaborated on the training process in 2.3.2, and the final form of iNALU training is depicted in figure 6.4.

6.1.4 Evaluation

We evaluated extrapolation and interpolation performance using mean squared error (MSE). We used the model inference on the data from interpolation and extrapolation datasets. We evaluated the model's performance at the end of each training epoch.

6.2 Training iNALU

Training iNALU is complicated in comparison to most of the feed-forward models. It requires weights and gradient clipping, delaying regularization, custom reinitialization technique, and independent gate training.

6.2.1 iNALU Training Configuration

We start by importing the TensorFlow package and iNALU layer from the NALU package. This package is part of this project. Then we build the model by stacking iNALU layers using **tf.keras.Sequential**. TensorFlow model subclassing **tf.keras.Model** works as well.

The next step is to compile the model. It can be seen as decoupling the model's architecture from the training loop. The model itself has no information about training parameters like learning rate.

We define a python function that will control the details of the training procedure. Functions passed as conditions will belong to the instance of the control callback, while all others will belong to the model's instance.

```python
import tensorflow as tf
from iNALU import NALU

# define model
model = tf.keras.Sequential([
        tf.keras.layers.InputLayer(INPUT_DIM),
        NALU(3, clipping=20),
        NALU(2, clipping=20),
        NALU(1, clipping=20)]
)
```

FIGURE 6.1: We can freely mix iNALU with other layers without the need for any adjustments due to the training control callback.

```python
# compile model
model.compile(
    optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.01),
    loss="mse",
    metrics=["mae"],
)
```

FIGURE 6.2: The model compilation is crucial, as it configures loss function (which we can control by training loop callback), learning rate, or metrics.

```python
def gate_open(self) -> bool:
    return self.batches % 10 < 8

def delay_reg(self) -> bool:
    return (
        self.reinit_epochs > 10
        and self.reinit_history
        and self.reinit_history[-1]["loss"] < 1.0
    )

def get_gates_variables(self) -> List[tf.Variable]:
    return [
        layer.g for layer in self.layers
            if isinstance(layer, NALU)]
```

FIGURE 6.3: **Gate_open** controls gating in iNALU training. Gates are trained separately from the remaining part of the model and do not depend on the input data. We elaborated on this in section 2.3.1. To control gate variables, we extract them in function **get_gates_varialbes** where we iterate through all model's layers and extract gate variables of all **NALU** layers. This function returns the list of gate variables crucial to train gates separately. **Delay_reg** conditionally delay regularization loss for 10 epochs and a loss threshold (in this case, 1.0).

Now, we are ready to combine everything above in a single configuration file to control the model's training. Appendix C summarises dynamically generated functions.

```
config = {
    "gate": {
        "cond": gate_open,
        True: {"clipping": (-0.1, 0.1),
               "excluded_variables": get_gates_variables
               },
        False: {
            "clipping": (-0.1, 0.1),
            "variables": get_gates_variables,
        },
    },
    "delay": {
        "cond": delay_reg,
        True: {
            "loss": False,
            "regularize": True}
            },
}
```

FIGURE 6.4: Both **gate_open** and **delay_reg** bind to the callback's instance, while **get_gates_varialbes** bind to the model's instance. For **gate** action, we clip gradient values between -1.0 and 1.0. In this step, we constantly switch between gates and all other weights. Switching between two is controlled by **gate_open**. In **delay**, we only take actions when **delay_reg** evaluates to true. We relay the whole training_substep only on the regularization loss.

It is the configuration dictionary to control the gating mechanism and regularisation delay. However, we need out-of-the-training loop control to implement reinitialisation and early stopping. We can see the logic from the training prerequisites 6.1.3 enforced in the asynchronous configuration dictionary 6.5.

We created another callback instance to evaluate interpolation and extrapolation capabilities of the trained model. We can find the code for **InExtrapolationEvaluationCallback** in the appendix D.

Having everything in place, we can run the training of the deep iNALU model using the default TensorFlow API 6.6. By doing so, we benefit from all convenience of TensorFlow training.

```python
def reinit_cond(self):
    reinit_loss = [x["loss"] for x in self.history]
    return (len(reinit_loss) > 10000) and (
        tf.math.reduce_mean(reinit_loss[-3000:]) > 0.2
    )

def reinitialise_fn(self):
    for layer in self.layers:
        if isinstance(layer, NALU):
            layer.reinitialize()

def reinitialise_metrics(self):
    self.history = []
    self.epochs = 0
    self.batches = 0

def early_stopping(self):
    return tf.experimental.numpy.log10(
        tf.math.reduce_mean([x["loss"] for x in self.history][-3000:])
        ) < -10

def stop_training(self):
    self.model.stop_training = True

async_config = {
    "reinitialize": {
        "cond": reinit_cond,
        "model": reinitialise_fn,
        "callback": reinitialise_metrics,
    },
    "eraly_stopping": {
        "cond": early_stopping,
        "callback": stop_training,
    }
}
```

FIGURE 6.5: Reinitialisation call **reinitialise** function on the all layers of iNALU instance. The callable doing this binds to the model instance, while the tracking metrics reinitialisation happens from the position of the callback. Hence, we pass the callable reinitialising history, batches, and epochs to the **"callback"** key. What is worth mentioning is that we do not have to implement early stopping using asynchronous configuration. TensorFlow implements it in the predeclared instance of **tf.keras.callbacks.EarlyStopping**. Nevertheless, we found our implementation more concise.

```python
loopControl = LoopControlerCallback(config, async_config, verbose=1)
inExEvaluation = InExtrapolationEvaluationCallback(ext_dp, int_dp)
history = model.fit(data_dp, epochs=120, verbose=1, callbacks=[
    loopControl,
    inExEvaluation
    ])
```

FIGURE 6.6: It is the final form of the user-facing API to train the deep iNALU model. We can freely use other callbacks with the loop control. The structure of the configuration dictionaries is easy to understand and modify. We hope it will make working with more sophisticated ML architecture, requiring customisation of the training process, less intimidating for young scientists and developers.

6.3 Training Results

We can analyse the results of iNALU training from two perspectives. The first is the perspective of the whole training process. Another perspective is the plot,

where we show loss evaluation at every training step before the model converges. Even though these two models represent the same phenomenon of training multiple iNALU models, we can extract different information from each. All losses are presented on a logarithmic scale. Without it, we would be unable to inspect the effect of regularisation.

The obvious difference between the plots from different perspectives is their smoothness. By default, TensorFlow smooths the recorded loss after each training epoch. It is why we cannot see the effect of gate training on the plot 6.7. The loss inspection 6.8 at each training step reveals how much the loss oscillates due to the switching between the training model and gates.

Figure 6.7 shows iNALU initialisation dependence. For some seeds, early stopping terminated training in 14 epochs, while others made the model go through multiple reinitialisations to converge. Huge jumps in loss values indicate regularisation loss being added to the training. As we can see, when regularisation starts, the model converges within one epoch afterwards. Unfortunately, the model needs to reach the local minimum before regularisation starts.

Furthermore, figure 6.8 depicts asynchronous logic encoded in the training control callback. Due to the tracking reinitialisation, we can precisely inspect the converged training loss trajectories.

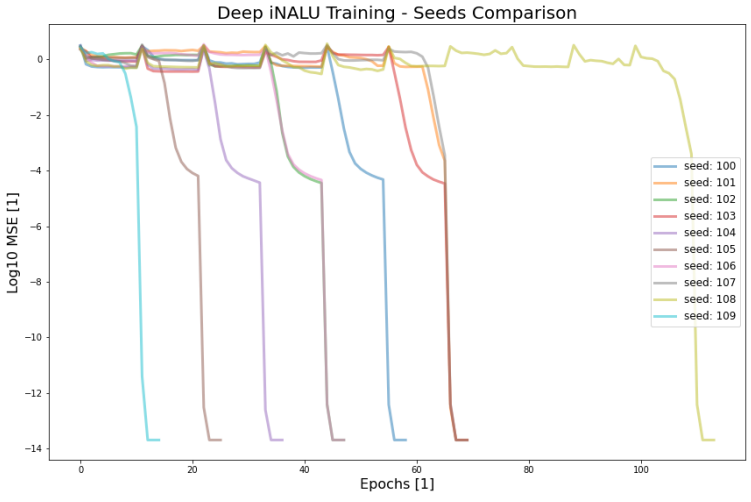


FIGURE 6.7: In the log space, we can see the whole training process. We can easily understand the significance of regularization by the many orders of magnitude loss decrease. In all cases, the span of one epoch was enough to utilize the full regularization potential.

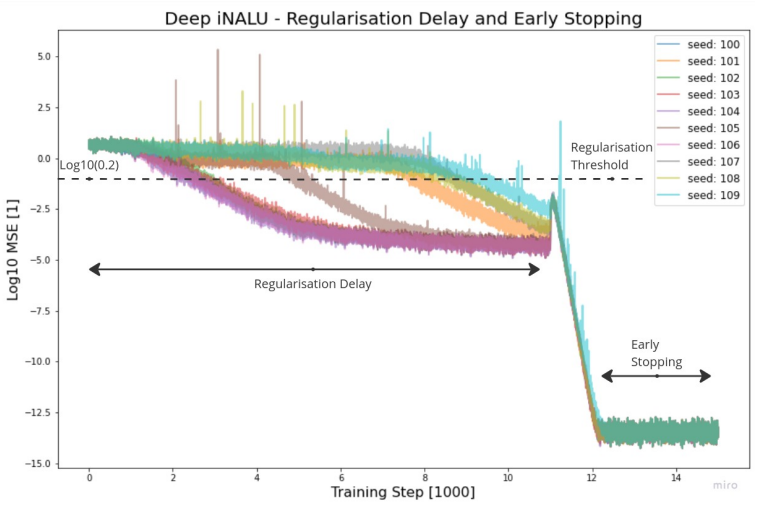


FIGURE 6.8: These plots represent tracking loss by training control callback. Due to the reinitialisation strategy 6.5, we can see only training steps leading to the training termination due to the early stopping mechanism. The regularisation effect and early stopping are represented by arrows on the plot. Additionally, the training loss oscillates within the single training step due to the gating mechanism. We can generally expect it from training loop control because we will encounter various training steps with different loss functions or configurations.

6.4 iNALU Extrapolation

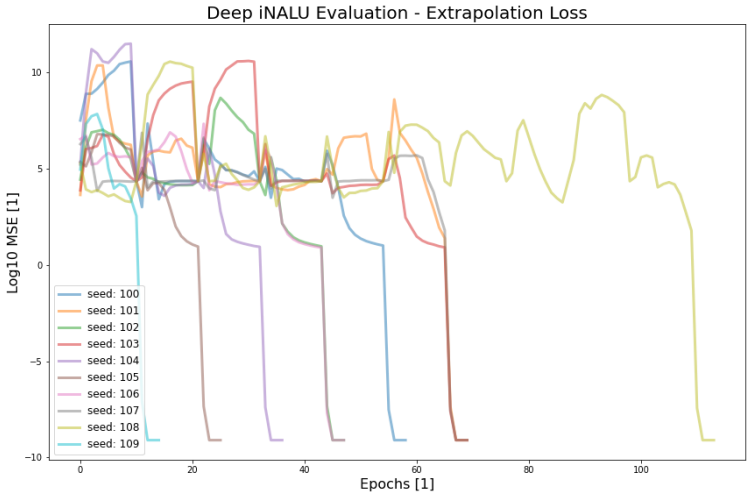


FIGURE 6.9: The loss evaluated on the extrapolated dataset is a couple of orders of magnitude greater than the loss calculated on the interpolated dataset 6.10. It is the indication of no generalisability. It is normal for most ML models, which struggle to work on the out-of-distribution (OOD) data. INALU model converges very quickly after regularisation.

6.5 iNALU Interpolation

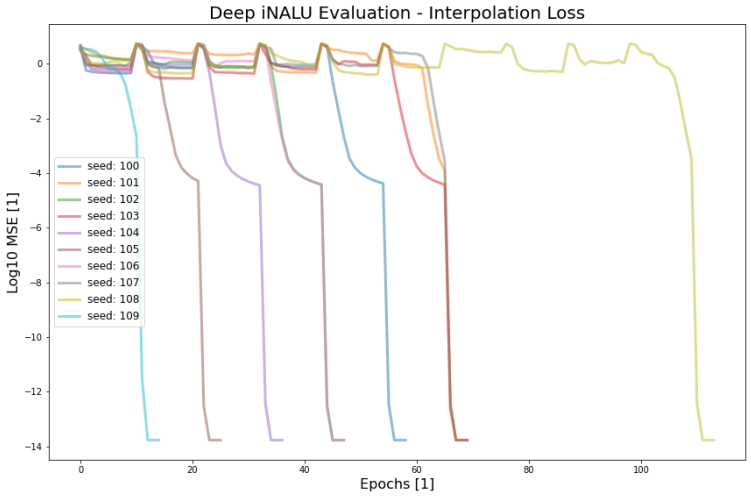


FIGURE 6.10: Evaluation of the interpolation dataset resembles the training plot 6.7 closely. Evaluation of the data drawn from the same distribution carries no information about the generalisability. It is obvious comparing the vertical scales of this plot with extrapolated loss plot 6.9.

6.6 Trained iNALU Explanation

In chapter 2.3, we mention that iNALU architecture is explainable and generalisable. The combination of fundamental mathematical operations achieves it. A fully trained model with multiple iNALU layers can be analysed graphically by layers' weights inspection. A diagram 6.11 explain the role of each layer. We can see that even the shallow model with three layers can capture the complex combination of fundamental mathematical operations.

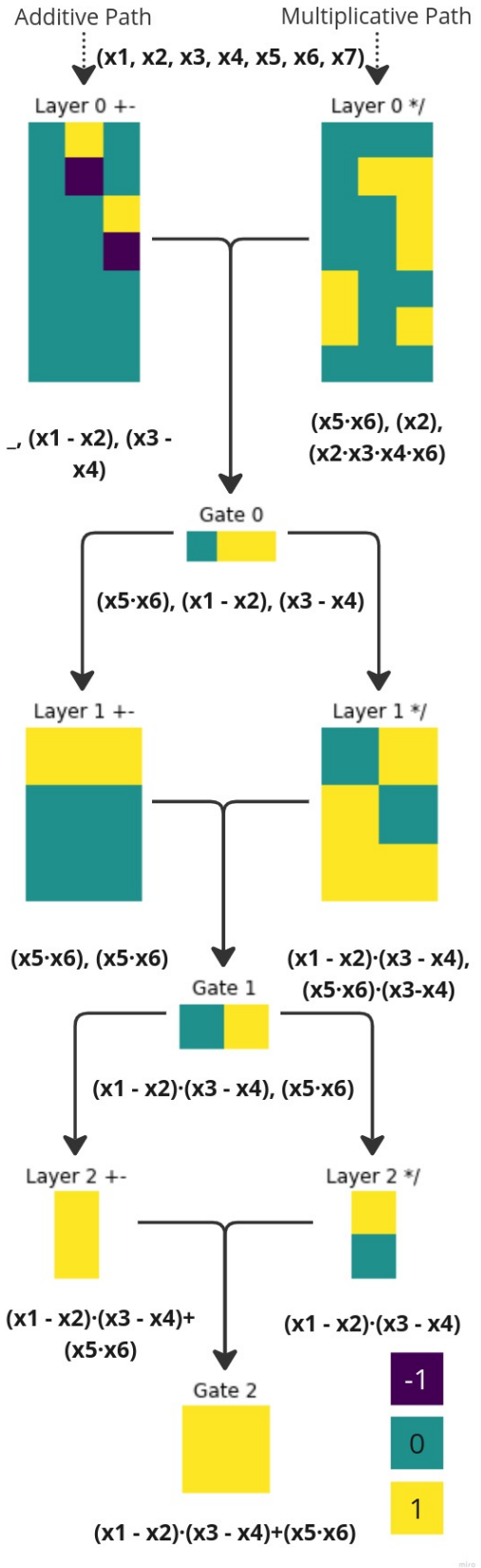


FIGURE 6.11: Diagram explaining the role of each layer. We can see the operations composition in action and the effect of the gate as a vector. We follow the convention that 1 is $+$ or $\cdot$, and -1 means $-$ or $\div$ in summative and multiplicative paths, respectively. This diagram shows the need for deep networks as the model of depth two could not capture this particular composition.

Chapter 7

Further Work

7.1 Gradient Modification

For now, we revise the gradient only by using gradient clipping. We manage it by passing the tuple of floats. Nevertheless, we can transform it to accept callable that will consume gradient and return the list of altered tensors. Callables passed as the gradient modification shall follow specific function scope referring to the function's arguments and return type. Function at the image 7.1 depicts our proposal for gradient manipulation.

```python
def modifyGradient(gradient: List[tf.Tensor]) -> List[tf.Tensor]:
  # your custom logic to modify gradient
  modifiedGradient = gradient
  if len(modifiedGradient) != len(gradient):
    raise ValueError("Modified gradient must be of the same length")
  for i, (mg, g) in enumerate(zip(modifiedGradient, gradient)):
    # ensure elements of tensors are the same shape
    if tf.shape(mg) != tf.shape(g):
      raise ValueError(f"gradients at position {i} have different shapes")
  return gradient
```

FIGURE 7.1: Example of callable to modify gradient. This function must return the list of tensors of the same length, and all elements must have the same shapes.

7.2 Graph Optimisation

Our solution is built upon the strong decoupling of each training sub-step. We did it to keep the code human-inspectable; nevertheless, it comes at the cost of a lot of repeatable code captured in the dataflow graph. It makes the model's memory

footprint larger but guarantees training steps decoupling from each other. This property makes the code easy to debug. In some situations, we should sacrifice the code's readability and effortlessness of debugging to achieve better performance. We can optimise the structure of the dataflow graph according to the logic imposed by the configuration dictionary. It requires a new approach towards code recompilation and is out of the scope of the presented work.

Chapter 8

Conclusions

We successfully designed and engineered the novel approach toward TensorFlow training. We control the training procedure by passing the Python dictionary to the training loop callback. Prior to training, we bind newly-created functions to the model instance together with controlling variables. Then we recompile the model to follow a custom training process. All of these occur from the callback's position; hence the user-facing API is the same as in the case of the default training in TensorFlow. We tested the code on the deep iNALU models. We verified that our code is versatile enough to tackle the peculiarities of iNALU training. We explained architectural and technical decisions which formed the final shape of the code. We rely on dynamic functions' body generation and their execution and binding to external instances. We focused on the dynamic function readability; hence the generated code is human-readable. We run multiple integration tests to inspect various configurable parameters in our training control callback. We compressed the code to a single Python file with TensorFlow dependency. It makes it portable and easy to reuse. We delineated a path towards extending the proposed solution to tackle the broader range of training customisation.

Appendix A

Project Planning

A.1 Project Brief

A.1.1 Title

Implemention of iNALU as Tensorflow Addons and Its Applications of Exploring HIFF Problem Space

A.1.2 Content

A.1.2.1 Problem

Using the iNALU layer in Tensorflow is cumbersome, and its implementation is neither rigorously tested nor optimised for performance.

A.1.3 Goal

The primary goal is to add iNALU layer to the official distribution of TensorFlow Addons. Then with well-tested and user-friendly implementation, investigate if we can explore the HIFF problem space relying on iNALU transparency.

A.1.3.1 Scope

- Rise feature request issue on Tensorflow Addons to add iNALU to the official distribution.
- Configure an environment for implementation and testing.
- Implement iNALU according to the community's coding standards.
- Repeat results from scientific papers.
- Investigate inference optimizations for CPU and GPU.
- Investigate how iNALU can help to investigate HIFF problem space.

A.1.3.2 Successful outcome

- Create pull request to the official tf-addons repository.
- Successful demonstration of using iNALU from the locally built tf.addons package.
- Summary of the investigation: Can we use iNALU architecture to explore HIFF problem space.

A.2 Gantt Charts

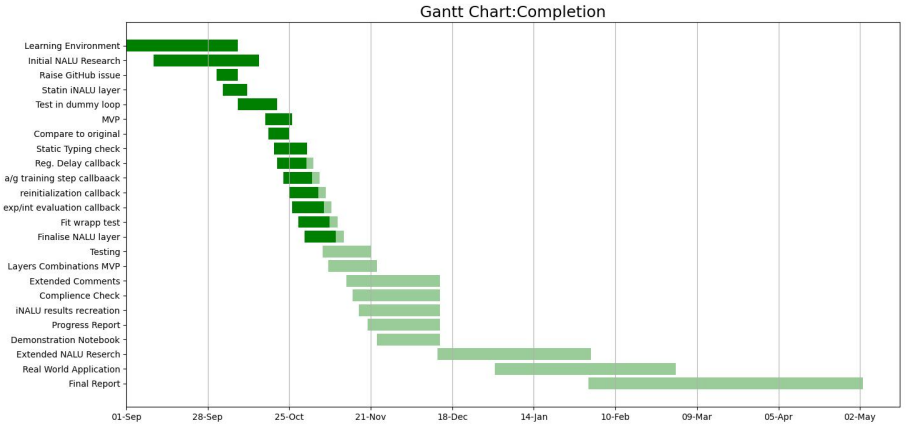


FIGURE A.1: Gantt chart from October.

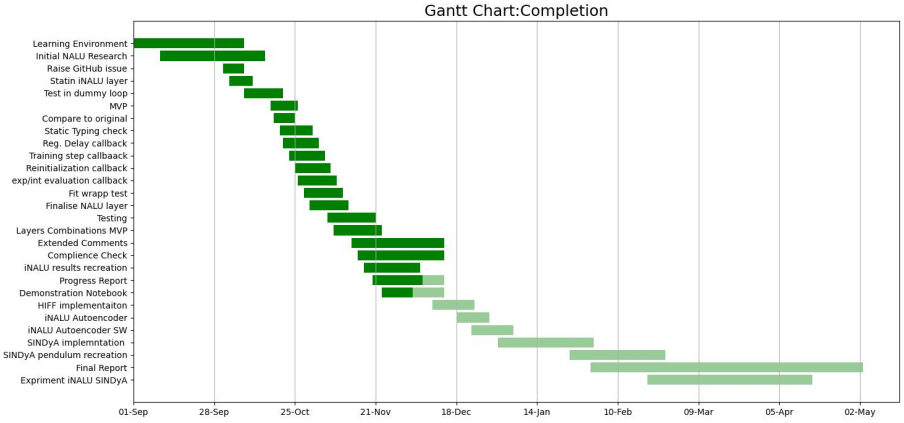


FIGURE A.2: Gantt chart from December.

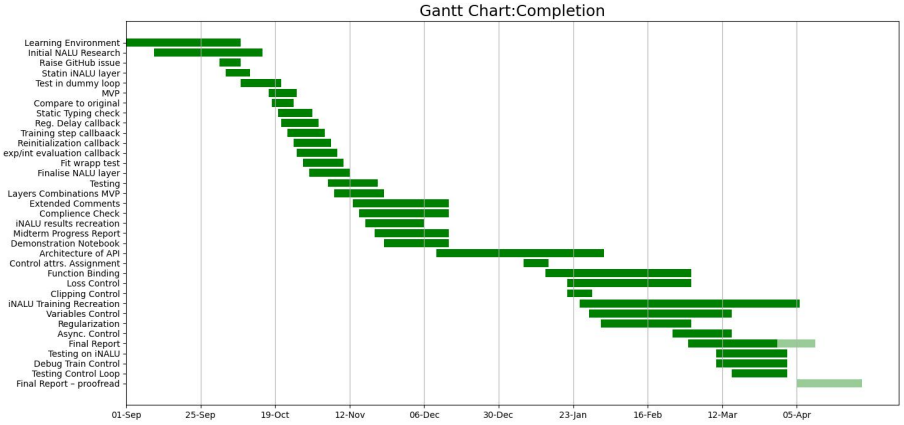


FIGURE A.3: Gantt chart from April. In January, we decided to start working on training loop control instead of SINDY iNALU Autoencoder.

A.3 Comparision

A.3.1 Initial Goal

Our initial goal was to investigate the possibility of using iNALU architecture in the SINDY autoencoder. We thought iNALU could replace the predetermined matrix of elements combinations and sparse regularisation. Firstly, we wanted to experiment with the iNALU autoencoder to explore the HIFF problem space. We thought it could improve deep optimisation [1] by guaranteeing transparency of the successfully trained layers. After successfully passing this stage, we wanted to experiment with the iNALU layer in SINDY autoencoder architecture.

A.3.2 Risk Assessment and Goals Adjustments

We started our work on implementing the iNALU layer in TensorFlow. We pinpointed the difficulties of training models with iNALU layers due to the training peculiarities. Then we created a function wrapping iNALU model to control the training procedure. At the same time, we started doing background research on the chances of using iNALU in autoencoder architecture. We tried to qualitatively describe the benefits of using iNALU architecture in the DeepOptimisation architecture.

Nevertheless, early testing showed the difficulties of training iNALU models on convenient problems. We have realised that integrating iNALU into an autoencoder architecture carries the inherent risk of yielding an unsuccessful outcome. Due to the training difficulties and the vague understanding of the possible benefits of using iNALU in DeepOptimisation, we decided to adjust the goals of our work. We changed the project character from research to software engineering. We analysed the hurdles of using iNALU in TensorFlow and concluded that we should be able to use the default API for training instead of handcrafting a custom training loop.

A.3.3 Final Report

We started by building a proof of concept (appendix B) of controlling the training by a single configuration dictionary. After successfully testing, we design a configuration file structure for training control. Then, we found a way of binding external functions to other instances. Then, we progressively incorporated adjustable parameters from the configuration dictionary. In the process, we solved issues mentioned in the section 4.2. We trained iNALU models on multiple seeds using our approach towards the custom training procedure 6. Finally, we analysed one of the successfully trained deep iNALU models and graphically showed its explainability properties.

Appendix B

Training Control Static Code Demonstration

Proof of concept of controling training from the callback's position using control variables (code).

```
import tensorflow as tf
class LoopControlableModel(tf.keras.Model):

    def __init__(self, *args, **kwargs):
        super(LoopControlableModel, self).__init__(*args, **kwargs)
        self.gate = tf.Variable(False, trainable=False) # gate
        ↪  control variable

    @tf.function
    def train_step(self, data):
        train_metrics = tf.cond(
            self.gate,
            lambda: self.train_step_active(data),
            lambda: self.train_step_passive(data)
        )

        return train_metrics

    @tf.function
    def train_step_active(self, data):
```

```
        x, y = data
        with tf.GradientTape(watch_accessed_variables=True) as
        ↪  tape:
            logits = self(x, training=True)
            loss_value = self.compiled_loss(y, logits)
        grads = tape.gradient(loss_value, tape.watched_variables())
        self.optimizer.apply_gradients(zip(grads,
        ↪  tape.watched_variables()))
        return {**{m.name: m.result() for m in self.metrics},
            **{"active": True, "passive": False}}

    @tf.function
    def train_step_passive(self, data):
        x, y = data
        with tf.GradientTape(watch_accessed_variables=True) as
        ↪  tape:
            logits = self(x, training=True)
            loss_value = self.compiled_loss(y, logits)
        grads = tape.gradient(loss_value, tape.watched_variables())
        self.optimizer.apply_gradients(zip(grads,
        ↪  tape.watched_variables()))
        return {**{m.name: m.result() for m in self.metrics},
            **{"active": False, "passive": True}}

class LoopControlerCallback(tf.keras.callbacks.Callback):

    def __init__(self, gating_frequency: int, *args, **kwargs) ->
    ↪  None:
        super(LoopControlerCallback, self).__init__(*args,
        ↪  **kwargs)
        self.gating_frequency = gating_frequency

    def on_epoch_end(self, epoch, logs):
        """Control gating variable from the level of callback
        ↪  which can work on epoch/batch level."""
        # tf.variable.assign is different than tf.variable =
        ↪  <sth>. The second option is compiled to static
```

```
        # value in TF graph of computation as the result of
        ↪  @tf.function decorators in LoopControlableModel
        self.model.gate.assign(epoch % self.gating_frequency ==0)

class LoopControledModel(LoopControlableModel):

    def __init__(self, output_size, *args, **kwargs):
        super(LoopControledModel, self).__init__(*args, **kwargs)
        # define architecture of the model
        self.layer = tf.keras.layers.Dense(output_size)

    def call(self, inputs):
        return self.layer(inputs)

DATASET_SIZE, INPUT_SIZE, OUTPUT_SIZE = 1000, 2, 1
BATCH_SIZE = 64
data = tf.data.Dataset.from_tensor_slices(
    (tf.random.uniform((DATASET_SIZE, INPUT_SIZE)),
    tf.random.uniform((DATASET_SIZE, OUTPUT_SIZE)))
    ).batch(BATCH_SIZE)

model = LoopControledModel(OUTPUT_SIZE)

# compile model
model.compile(optimizer=
    tf.keras.optimizers.RMSprop(learning_rate=0.01),
    loss="mse",
    metrics=["mae"])

# start training
history = model.fit(data, epochs = 7, verbose = 1,
    callbacks=[LoopControlerCallback(2)])
```

Terminal Output

```
Epoch 1/7 16/16 [==============================] -
1s 2ms/step - loss: 0.1328 - active: 0.0000e+00 - passive: 1.0000
Epoch 2/7 16/16 [==============================] -
0s 3ms/step - loss: 0.1042 - active: 1.0000 - passive: 0.0000e+00
Epoch 3/7 16/16 [==============================] -
0s 2ms/step - loss: 0.0980 - active: 0.0000e+00 - passive: 1.0000
Epoch 4/7 16/16 [==============================] -
0s 3ms/step - loss: 0.0933 - active: 1.0000 - passive: 0.0000e+00
Epoch 5/7 16/16 [==============================] -
0s 3ms/step - loss: 0.0902 - active: 0.0000e+00 - passive: 1.0000
Epoch 6/7 16/16 [==============================] -
0s 3ms/step - loss: 0.0881 - active: 1.0000 - passive: 0.0000e+00
Epoch 7/7 16/16 [==============================] -
0s 3ms/step - loss: 0.0868 - active: 0.0000e+00 - passive: 1.0000 ...
```

Appendix C

Training Control Callback - Function Bodies Generation

C.1 Configuration File

```
config = {
    "gate": {
        "cond": gate_open,
        True: {"clipping": (-0.1, 0.1),
               "excluded_variables": get_gates_variables
               },
        False: {
            "clipping": (-0.1, 0.1),
            "variables": get_gates_variables,
        },
    },
    "delay": {
        "cond": delay_reg,
        True: {
            "loss": False,
            "regularize": True}
            },
}
```

FIGURE C.1: This configuration dictionary is the same as on image 6.4.

C.2 Master Step

```
@tf.function
def train_step(self, data):
```

```
    slave_loss = {'loss_gate' :
    ↪  tf.cond(self.control_variables['gate'], lambda:
    ↪  self.gate_on(data), lambda:
    ↪  self.gate_off(data)),'loss_delay' :
    ↪  tf.cond(self.control_variables['delay'], lambda:
    ↪  self.delay_on(data), lambda: 0.0)}
    losses = {**{'loss': slave_loss['loss_gate']}, **slave_loss}
    metrics = {m.name : m.result() for m in self.metrics}
    metrics.pop("loss")
    control_states = {
        control_name: tf.cond(
            control_value,
            lambda: tf.constant(True),
            lambda: tf.constant(False),
        )
        for control_name, control_value in
        ↪  self.control_variables.items()
    }

    return {**losses, **metrics, **control_states}
```

C.3 Slave Steps

C.3.1 Gate_on

```
@tf.function
def gate_on(self, data):
    x, y = data
    with tf.GradientTape(watch_accessed_variables=False) as tape:

        for g in self._excluded_variables['gate'][True]:
            tape.watch(g)

        logits = self(x, training=True)
        loss_value = loss(y, logits)
```

```
    grads = tape.gradient(loss_value, tape.watched_variables())
    self.optimizer.apply_gradients(zip([tf.clip_by_value(g, -0.1,
    ↪  0.1) for g in grads], tape.watched_variables()))
    self.compiled_metrics.update_state(y, logits)
    return loss_value
```

C.3.2 Gate_off

```
@tf.function
def gate_off(self, data):
    x, y = data
    with tf.GradientTape(watch_accessed_variables=False) as tape:

        for g in self._included_variables['gate'][False]:
            tape.watch(g)

        logits = self(x, training=True)
        loss_value = loss(y, logits)
    grads = tape.gradient(loss_value, tape.watched_variables())
    self.optimizer.apply_gradients(zip([tf.clip_by_value(g, -0.1,
    ↪  0.1) for g in grads], tape.watched_variables()))
    self.compiled_metrics.update_state(y, logits)
    return loss_value
```

C.3.3 Delay_on

```
@tf.function
def delay_on(self, data):
    x, y = data
    with tf.GradientTape(watch_accessed_variables=True) as tape:

        logits = self(x, training=True)
        loss_value = tf.constant(0, dtype=tf.float32)
        loss_value += tf.math.reduce_sum(self.losses)
```

```
    grads = tape.gradient(loss_value, tape.watched_variables())
    self.optimizer.apply_gradients(zip(grads,
    ↪  tape.watched_variables()))
    self.compiled_metrics.update_state(y, logits)
    return loss_value
```

Appendix D

Extrapolation and Interpolation Evaluation Callback

```
class
↪ InExtrapolationEvaluationCallback(tf.keras.callbacks.Callback):
    def __init__(self, ext_data, int_data, *args, **kwargs):
        super(InExtrapolationEvaluationCallback,
        ↪  self).__init__(*args, **kwargs)
        self.ext_data, self.int_data = ext_data, int_data
        self.ext_results, self.int_results = {}, {}
        self.test_per_epochs = 1
        self.epochs = 0

    def on_epoch_end(self, epochs, log=None):
        if self.epochs % self.test_per_epochs==0:
            self.int_results[self.epochs] =
            ↪  tf.reduce_mean(tf.keras.losses.get("mse")(self
            ↪  .model.predict(self.int_data[0], verbose = 0),
            ↪  self.int_data[1])).numpy()
            self.ext_results[self.epochs] =
            ↪  tf.reduce_mean(tf.keras.losses.get("mse")(self
            ↪  .model.predict(self.ext_data[0], verbose = 0),
            ↪  self.ext_data[1])).numpy()
        self.epochs+=1
```

Bibliography

[1] Jamie Caldwell, Joshua Knowles, Christoph Thies, Filip Kubacki, and Richard Watson. Deep optimisation: Transitioning the scale of evolutionary search by inducing and searching in deep representations. *SN Computer Science*, 3(3):1–26, 2022.

[2] Kathleen Champion, Bethany Lusch, J Nathan Kutz, and Steven L Brunton. Data-driven discovery of coordinates and governing equations. *Proceedings of the National Academy of Sciences*, 116(45):22445–22451, 2019.

[3] Lukas Faber and Roger Wattenhofer. Neural status registers. *arXiv preprint arXiv:2004.07085*, 2020.

[4] Niklas Heim, Tomas Pevny, and Vasek Smidl. Neural power units. *Advances in Neural Information Processing Systems*, 33:6573–6583, 2020.

[5] Zachary C Lipton. The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery. *Queue*, 16(3):31–57, 2018.

[6] Andreas Madsen and Alexander Rosenberg Johansen. Neural arithmetic units. *arXiv preprint arXiv:2001.05016*, 2020.

[7] Bhumika Mistry, Katayoun Farrahi, and Jonathon Hare. A primer for neural arithmetic logic modules. In *JMLR: Workshop and Conference Proceedings*, volume 23, pages 1–58, 2022.

[8] Shangeth Rajaa and Jajati Keshari Sahoo. Convolutional feature extraction and neural arithmetic logic units for stock prediction. In *Advances in Computing and Data Sciences: Third International Conference, ICACDS 2019, Ghaziabad, India, April 12–13, 2019, Revised Selected Papers, Part I 3*, pages 349–359. Springer, 2019.

[9] Jan Niclas Reimann, Andreas Schwung, and Steven X Ding. Neural logic rule layers. *Information Sciences*, 596:185–201, 2022.

[10] Daniel Schlör, Markus Ring, and Andreas Hotho. inalu: Improved neural arithmetic logic unit. *GitHub repository*, abs/2003.07629.

[11] Andrew Trask, Felix Hill, Scott E Reed, Jack Rae, Chris Dyer, and Phil Blunsom. Neural arithmetic logic units. *Advances in neural information processing systems*, 31, 2018.