VRIJE
UNIVERSITEIT
BRUSSEL

NEURAL LEGO: GENERATING NEURAL NETWORKS WITH REACTIVE BUILDING BLOCKS

Bachelorproef Wetenschappen

Lennert Saerens

Academiejaar 2022-2023

Promotoren: Prof. Dr. Wolfgang De Meuter, Prof. Dr. Coen De Roover

Begeleiders: Bjarno Oeyen, Camilo Velázquez-Rodríguez

Wetenschappen & Bio-ingenieurswetenschappen

Contents

1 Problem Statement

In recent years, there has been a huge growth in interest in deep learning. Because of this, the amount of neural networks currently in production is bigger than ever before.

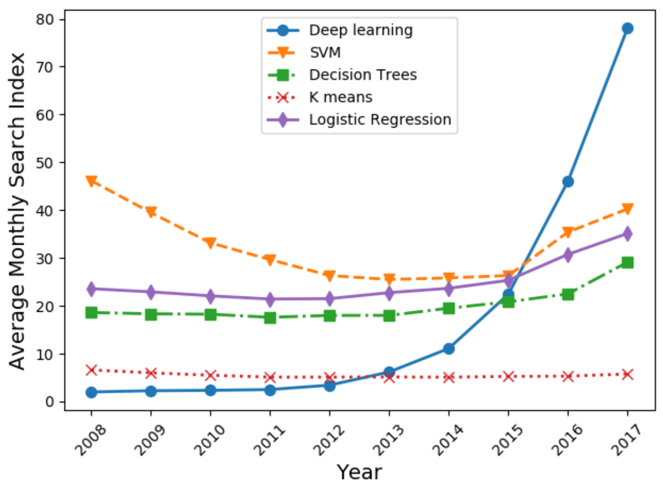

Figure 1: Google Trend showing more attention toward deep learning in recent years. (From [21])

Students are often presented with graph based representations of various kinds of neural networks when being taught about them. In these graphs, the nodes represent the network's neurons and the edges represent the connections between neurons. The weights of the connections are often abstracted away for simplicity's sake. These graphical representations offer an intuitive way for the students to understand the way data flows through the network.

Looking at the code for these seemingly simple introductory examples such as the one in Figure 2, it is still relatively readable and easy to understand.

```python
input = keras.layers.Input(shape=(784,))
n1 = keras.layers.Dense(64)(input)
n2 = keras.layers.Dense(64)(input)
n3 = keras.layers.Dense(64)(input)
added = keras.layers.Add()([n1, n2])
x = keras.layers.Average()([added, n3])
x = keras.layers.Dropout(0.5)(x)
x = keras.layers.Dense(1, activation="softmax")(x)
model = keras.models.Model(inputs=input, outputs=x)
```

Listing 1: Code for the neural network in Figure 2.

However, the presented way of defining neural networks using the *Keras Functional API* in Listing 1 can quickly lead to verbose and unintuitive code with no meaningful parallels to the graph based visual way used to represent neural networks. The fact that the code defining the network's architecture is not grouped together inside a single constructs only worsens the matter since this allows other snippets of Python code to be interleaved with the definition of the network. This discrepancy makes it harder for students, programmers and even experienced data scientists to transfer the intuition they have gained by looking at graphical representations into practice.

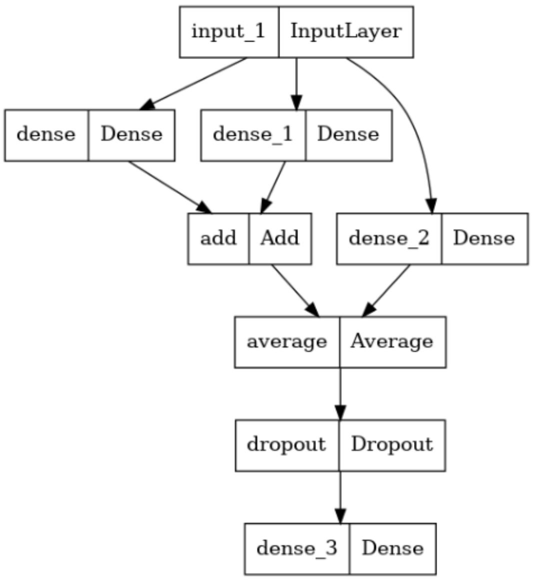

Figure 2: Graph based representation of a neural network. Each box represents a layer of the artificial neural network and is made up of two parts: the layer's name and its type.

Furthermore, when using this style of programming to define neural networks, the programmer easily reverts to using *destructive updates* to build the network and they are even incentivised to do so by the Keras Functional API's documentation. However, since values might be changed behind the programmers back, *destructive updating* can be considered a harmful programming practice [34, 19]. This is especially worrisome since a recent study by Yida Tao et al. found that debugging machine learning applications using libraries written in Python is often considered extra difficult due to the error messages being perceived as "cryptic", "unclear", "confusing", "uninformative", "misleading", "unhelpful", "poorly written", and "hard to digest" [28]. The fact that these errors are caught at runtime while machine learning applications often take a large amount of time to load big libraries and huge amounts of data, only worsens the matter.

When considering the previous, two orthogonal issues with the definition of neural networks can be identified:

1. The lack of parallels between the graph based visual representation of a neural network, and the way it is defined.

2. Errors are only caught at runtime and have uninformative error messages.

The first problem will from now on be referred to as the *intuitive discrepancy problem.* It can be solved by defining the neural network using a *reactive programming language.* These languages are based on the *reactive programming paradigm* and programs written in them can be visually represented using a directed acyclic graph [24]. Therefore, defining neural networks in a reactive programming language should provide a strong link to their graph based visual representation and allow for cleaner, more intuitive code.

The second problem is a more extensive one to solve because the amount of errors that could occur when defining and using a neural network is very large. From this point onward, it will be referred to as the *lazy error problem.* One solution could be to provide more intuitive error messages to the programmer, ideally at compile-time wherever possible because this provides the programmer with earlier feedback about the validity of the program.

Thesis objective:

Taking these two solutions into account, the goal of this bachelor's thesis will be to further investigate the similarities between the reactive programming paradigm and artificial neural networks. This will be achieved by creating a reactive *domain specific language* (DSL) that wraps around Keras and aims to solve both previously mentioned problems.

To build such a reactive DSL, two things are needed: a programming language suitable for building the DSL, and an overview of the typical workflow of creating a neural network. The former will be described in Section 2, while the latter will be discussed in greater detail in Section 3. After the DSL and its functionalities have been discussed in Section 3, the implementation of the DSL will be covered in detail in Section 4.

2 Hy as a Supporting Runtime

Two main things had to be taken into consideration when choosing a language to write the DSL: interoperability with Python, and support for macros. The first point is important so that the parts of the functionality provided by Keras can be reused. The second point enables easier addition of syntactical constructs to the language, allowing for the creation of a DSL.

Hy is a Lisp dialect that's embedded in Python. Since Hy transforms its Lisp code into Python *abstract syntax tree* (AST) objects, all of the functionality that is available in Python and libraries that are written in it, are also accessible in Hy [31].

The first thing a programmer will notice about Hy, is that it has Lisp's traditional parenthesis-heavy prefix syntax in place of Python's C-like infix syntax. As in other Lisps, the value of a simplistic syntax is that it facilitates Lisp's signature feature: *metaprogramming* through an extensive macro system [33].

Since both needs for creating the DSL are satisfied by Hy, it is the ideal candidate to use. In the next section we will look at the actual DSL.

3 The Brackeras Domain Specific Language

In this section *Brackeras* (Keras with brackets) will be introduced. Brackeras is a reactive domain specific language. It was specifically created for the purpose of creating artificial neural networks while solving the *visual discrepancy problem* and the *lazy error problem.* Brackeras compiles its programs into library calls to Keras to actually create highly performant and flexible neural networks.

It is important to have a clear view of the typical workflow of creating a neural network [35] so that the DSL satisfies the expectations of programmers wanting to use it for defining their own neural networks. The process can be divided into four distinct phases.

1. Defining the network
2. Compiling the network into a model
3. Training the model
4. Evaluating the trained model

5. Deploying the evaluated model

This fifth step is often included in other overviews of building a neural network but is beyond the scope of this bachelor's thesis. The workflow mentioned above is also assuming good, pre-processed training data is available. Otherwise, the first step would be to preprocess the training data.

The rest of this section provides a brief overview of how the Brackeras language operates.

3.1 Defining Networks

The first step programmers take when creating a neural network is the defining of a network *structure* or *architecture*. Layers of neurons are the basic building block when building such network architectures. An example of the composing of layers into a network can be seen in Figure 2. Because layers play such an important role in the composition of network architectures, Brackeras has its own *layer expressions*. Brackeras also provides the programmer with a construct called `network`, that can be used to combine layers into a network architecture. These constructs are discussed in the next sections.

3.1.1 Layer Expressions

The layer expressions in Brackeras are a direct translation of the layers provided by Keras. This means each different kind of layer in Keras has its own equivalent expression in Brackeras. The table below contains an overview of all layers currently supported by Brackeras and their Keras equivalent.

Layer Type	**Keras**	**Brackeras**
Core	Input	input
	Dense	dense
Convolutional	Conv2D	conv2d
	Conv3D	conv3d
Pooling	MaxPooling2D	maxpool2d
	MaxPooling3D	maxpool3d
	GlobalAveragePooling3D	globalavgpool3d
Merging	Concatenate	concat
	Average	avg
	Maximum	max
	Minimum	min
	Add	add
	Subtract	sub
	Multiply	mul
	Dot	dot
Normalization	BatchNormalization	batchnormalization
Regularization	Dropout	dropout
Reshaping	Flatten	flat

Table 1: Brackeras layers and their Keras equivalent.

Keras layers are extremely customizable. For example, the Keras `Dense` layer has ten arguments, providing defaults for eight of them. The programmer is then able to tweak each of these arguments to their liking by using *named parameters*. Because supporting all these named parameters is beyond the scope of this bachelor's thesis, Brackeras only supports a subset of the arguments for each of its layer expressions. However, as a proof of concept, the Brackeras `dense` layer does support exactly the same ten arguments as the Keras `Dense` layer. This is shown in Listing 6. A complete documentation of each Brackeras layer expression and the arguments they have, can be found in Appendix A.

3.1.2 Combining Layers into a Network

Since the definition of the network architecture is such an import step in neural networks' creation process, Brackeras has a specialised construct called `network` which can be used to create a network through the combining of layers. An example of the usage of `network` to define the structure of a neural network can be found in Figure 3.

```
(setv my-network
    (network
        (input (, 16))
        (dense 8)
        (dense 4)
        (dense 1)))
```

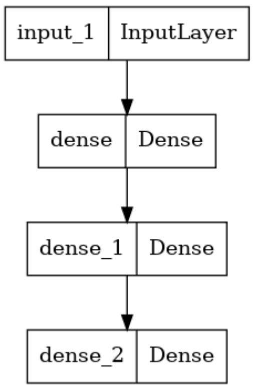

Figure 3: Example usage of `network`.

`network` takes a sequence of layers and accumulates then into a network structure. The example in Figure 3 is an extremely simple one where each layer uses the output of the previous layer in the sequence as its input. From now on, these type of networks will be referred to as having a *linear topology*.

However, popular machine learning libraries like Keras also provide the ability to its users to create neural networks with *non-linear* topologies. Brackeras' `network` also allows for the creation of non-linear network architectures. This functionality will be discussed in Section 3.1.3.

Apart from allowing the user to create network architectures, `network` also conducts a compile-time analysis of the network the user is trying to generate. This compile-time analysis allows Brackeras to check the following conditions:

- The first layer should be an input layer.
- A network should have at least two layers.
- When a layer uses the output of another layer as its input, the layers should be of compatible types. The compatibility of layers is discussed in detail in Sections 4.1.4 and 5.2.

The first point being a requirement of the Keras Functional API and the latter two being general rules for composing valid network architectures. The way this compile-time analysis contributes to the solution of the *lazy error problem* is discussed in detail in Section 5.2.

3.1.3 Non-Linear Topologies

When designing a neural network architecture, programmers often want to use a non-linear topology. However, the default behaviour of `network` is to, for each layer in the provided sequence, use the output of the previous layer as the input for the current layer. This implies that, by default, only networks with a linear topology can be created using `network`. This default behaviour is extremely similar to the *Keras Sequential API* which provides an intuitive syntax to convert a plain stack of layers into a linear network architecture [10]. An example of this can be seen in Listing 2.

```
model = keras.Sequential(
    [
        layers.Dense(2, activation="relu", name="layer1"),
        layers.Dense(3, activation="relu", name="layer2"),
        layers.Dense(4, name="layer3"),
    ]
)
```

Listing 2: Example usage of the Keras Sequential API. (From [10])

This sequential model is however extremely limited by the fact that it can only generate linear architectures. Since the goal of Brackeras is to also provide the programmer with the ability to create non-linear architectures, a way to circumvent the drawback of the Sequential Model has to be implemented.

Brackeras supports non-linear topologies by naming layers. Towards this end, programmers can make use of the `defl` form that is valid within `network` expressions. `defl` is short for "define layer" and allow the programmer to assign some name to a layer. An example of the usage of `defl` can be found in Figure 4.

`defl` takes two arguments, a name and a layer expression, and binds the layer expression to the name. The introduction of this naming mechanism allows the programmer to refer back to a certain named layer and use this specified layer as the input for the current layer, thus enabling the creation of non-linear network architectures.

The final piece in the puzzle of allowing both linear and non-linear topologies to be created is the ability for programmers to specify the input of a certain layer by using the name of another specified layer.

Brackeras provides the programmer with this functionality through a specialised construct called `with-input`. `with-input` can be used as seen in Figure 4 to specify that a certain layer should use another layer as its input instead of using the output of the preceding layer in the sequence. It takes two arguments, a layer and a name, and makes it so that the layer will use the output of the layer with the provided name as its input.

Another way input is, implicitly, specified is when using *merging* layers. A merging layer is a layer that takes the outputs of a set of layers and combines them in some way. An example of such a layer is the `add` layer in Figure 4. Because of its function, merging layers take other layers as their input. In Brackeras, layers that have been given a name using `defl` can be used as arguments for these merging layers. All merging layers in Keras are supported in Brackeras. The way the syntax for merging layers differs between the Keras Functional API and Brackeras is discussed in Section 5.1.1.

Now that the definition of different types of network architectures has been discussed, the compilation of networks into models can now be looked at in detail in the next section.

```
(setv my-network
    (network
        (defl in (input (, 16)))
        (defl x1 (dense 8)) (defl x2
     (with-input (dense 8) in))
        (add x1 x2)
        (dense 4)))
```

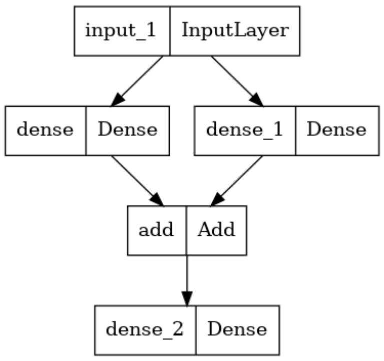

Figure 4: Example usage of `defl` and `with-input` to create a non-linear network architecture.

3.2 Creating Models

After designing a neural network architecture, programmers usually want to configure the network for training. This is done by combining the network with:

- An optimizer
- A loss function
- Metrics to be evaluated by the model during training and testing.

In Keras, this is done by calling the `compile` method on a network, passing the optimizer, loss function, and metrics as parameters [5]. Keras also allows some more advanced parameters to be passed but these are beyond the scope of this bachelor's thesis. Brackeras also provides a specialised construct for converting networks into trainable models: `model`. `model` has one required argument, mainly the network to be compiled into a trainable model. Brackeras provides the same default values as Keras for the optimizer, loss function and metrics as presented in the table below.

Argument	**Default Value**
optimizer	rmsprop
loss	None
metrics	None

Table 2: `model` arguments and their default value

An example of how `model` can be used to compile a network into a trainable model can be seen in Listing 3. Here, the default values for the optimizer, loss function and the metrics are overridden with the values *adam*, *categorical crossentropy* and a list containing *accuracy*. Notice that the different metrics to be used need to be packed together into a list when being passed as a parameter.

```
; Creating the network
(setv mnist-network
    (network
        (input input-shape)
        (conv2d 32 [3, 3])
        (maxpool2d [2, 2])
        (conv2d 64 [3, 3])
        (maxpool2d [2, 2])
        (flat)
        (dropout 0.5)
        (dense 10 "softmax")))

; Creating the model
(setv mnist-model (model mnist-network "adam" "categorical_crossentropy" ["
    accuracy"]))
```

Listing 3: Example usage of `model` for the creation of a simple *MNIST* model.

`model` returns a *reactive node*, being the root of the directed acyclic dataflow graph with which the network that was compiled by calling `model` corresponds to. What makes this node reactive is discussed in Section 4.2. This root node will be used as the input to the provided training and evaluation functions discussed in Sections 3.3 and 3.4.

3.3 Training Models

The next logical step after converting a network architecture into a trainable model is the actual training of the model. Brackeras provides a new syntactical structure called `fit` that can be used to train models. `fit` takes six arguments, three of which are required:

1. The model to be trained
2. The training data
3. The target data

The other three arguments are the *batch size*, the *epochs* and the *validation split*. These are not required because Brackeras provides them with a default value, as can be seen in the table below.

Argument	**Default Value**	**Possible Values**	**Explanation**
batch size	None	Integer or None	Number of samples per gradient update.
epochs	1	Integer	Number of iterations over the entire x and y data provided.
validation split	0.0	[0, 1]	Fraction of the training data to be used as validation data.

An example of how a trainable model can be trained using `fit` can be seen in Listing 4. Here, the default values of the batch size, epochs, and validation split are overridden with 128, 5, and 0.1 respectively.

```
(setv trained-mnist (fit mnist-model x-train y-train 128 5 0.1)
```

Listing 4: Example usage of `fit`

`fit` returns a *new* reactive node. Not just a destructively updated version of the reactive node returned by `model`. This is once again the root of the directed acyclic dataflow graph with which the model corresponds.

3.4 Evaluating Trained Models

The last step in the typical workflow of a programmer using Keras to create a neural network is to evaluate the network after training it. In Brackeras, this can be done using `evaluate` as can be seen in Listing 5.

```
(evaluate trained-mnist x-test y-test)

; Output in terminal
Test loss: 0.046185459941625595
Test accuracy: 0.9850000143051147
```

Listing 5: Example usage of `evaluate`.

`evaluate` takes two mandatory arguments, the input data, and the target data, and has one optional argument called *verbose*. *verbose* has 4 possible values:

- 0 = silent
- 1 = progress bar
- 2 = single line
- auto = 1 in most cases [5]

When the programmer uses `evaluate`, the *test loss* and *test score* are automatically printed to the terminal. The syntactical differences when evaluating models using Keras en Brackeras are discussed in Section 5.1.1.

With that, all different steps a programmer might take when using Brackeras have been discussed. In the next section, the way each of these four steps are implemented in Hy is discussed. Finally, a complete example showing all steps of the Brackeras workflow can be found in Appendix B. The code shown there implements a neural network for recognising hand-written digits using the MNIST dataset.

4 Brackeras' Implementation

In this section, the Hy (Section 2) implementation of each of the different syntactical constructs in Brackeras is discussed. Brackeras' complete implementation can be found at `https://gitlab.soft.vub.ac.be/bachelor-thesis-lennert/bachelorproef`.

4.1 Defining Networks

4.1.1 Layer Expressions

As mentioned in Section 3.1.1, each different kind of layer in Keras has its own equivalent expression in Brackeras that is added to the DSL through a macro. An example of such a macro can be seen in Listing 6.

```
(defmacro dense [units [activation None]
                       [use-bias True]
                       [kernel-initializer "glorot_uniform"]
                       [bias-initializer "zeros"]
                       [kernel-regularizer None]
                       [bias-regularizer None]
                       [activity-regularizer None]
                       [kernel-constraint None]
                       [bias-constraint None]]
    `["dense" ~units ~activation ~use-bias ~kernel-initializer ~bias-initializer ~
    kernel-regularizer ~bias-regularizer ~activity-regularizer ~kernel-constraint
    ~bias-constraint])
```
Listing 6: Implementation of the macro adding the dense layer.

This particular macro is called `dense` and has ten arguments. The arguments between square brackets have a default value that will be used when the programmer does not specify a value for them. Just like the `dense` macro presented in Listing 6, all other layer macros also have a name that specifies the layer kind to be created and take some arguments that serve to customize the layer. This information is then converted to a list. This process is also shown in Figure 5.

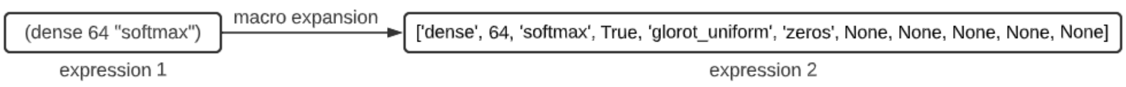
(dense 64 "softmax")
macro expansion
['dense', 64, 'softmax', True, 'glorot_uniform', 'zeros', None, None, None, None, None]
expression 1
expression 2

Figure 5: Expansion of the `dense` macro.

4.1.2 Naming Layers

The naming mechanism for layers, through the usage of `defl`, is also added to the DSL by using a macro. The macro's name is `defl` and it is depicted in Listing 7.

```
(defmacro defl [layer-name layer]
    (setv str-name (str layer-name))
    `(do (setv ~layer-name ~str-name)
         (, ~str-name ~layer)))
```
Listing 7: Implementation of the `defl` macro.

This macro takes the name the programmer wants to give to a layer, and the layer to be named as its input. First, the name to give is converted from an expression to a string. The macro is then expanded to an expression in which two things happen:

1. First the `layer-name` variable is bound to refer to the string that was created in the previous step.
2. Then a tuple containing the name and the layer is created.

The created tuple is the return value of the expanded expression. The complete expansion of a `defl` expression can be seen in Figure 6.

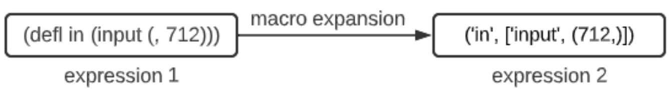

Figure 6: Expansion of the `defl` macro.

4.1.3 Specifying Input

The final way a programmer can manipulate layers in Brackeras is by specifying an input for them. This is implemented through the `with-input` macro. The definition of this macro is shown in Listing 8.

```
(defmacro with-input [layer input-layer]
    `["with-input" ~layer ~input-layer])
```

Listing 8: Implementation of the `with-input` macro.

When expanded, this macro simply wraps the layer and its specified input together in a list that is tagged with the string *with-input*. An example of this expansion process is shown in Figure 7. In this figure, *expression 1* is part of the network defined in Figure 4.

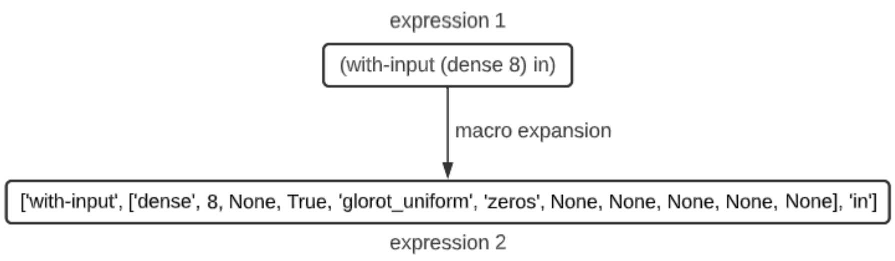
expression 1
(with-input (dense 8) in)
macro expansion
['with-input', ['dense', 8, None, True, 'glorot_uniform', 'zeros', None, None, None, None, None], 'in']
expression 2

Figure 7: Expansion of the `with-input` macro.

4.1.4 Combining Layers into a Network

Now that all the macros handling the layer expressions have been discussed, the combination of layers into a network architecture can be looked at. This is handled in the `network` macro, the implementation of which will be discussed in detail in this section.

The first that happens in the definition of `network` is the definition of three predicates to check whether a specific layer expression is a named layer, a layer that has a specified input, or the layer is an input expression. These work by checking the structure of the expression.

Afterwards, a tuple named `incompatibilities` and an empty dictionary named `name->kind` are defined. `incompatibilities` stores a collection of incompatible layers and `name->kind` is used to store pairs of layer names and their kind. This is used to check the incompatibilities between layers and their specified input.

Finally, two functions for getting information about layer expressions, named `expr-kind` and `layer-kind`, are defined. These can be used to get the expression kind (*input, named, or normal*), and the kind of layer the expression represents respectively. These functions exploit the distinct structure of each kind of expression to get this information about them.

With these definitions out of the way, the implementation of the `network` macro's different functionalities can now be looked at in detail. Since the `network` macro has lots of different tasks, an overview of all its functions are listed below:

- Compile-time analysis of the network structure as discussed in Section 3.1.
- Converting the list representation of layers into an actual Keras network by first *parsing* and then *building* the network.

The **network** macro's first function is to conduct a compile-time analysis of the network the programmer is trying to generate. This is done in three steps, the first of which being checking whether the network has at least two layers and throwing an exception if this requirement is not met. The implementation of this step can be seen in Listing 9. Here, all parts of the implementation of **network** that are not relevant for verifying whether the network has at least two layers have been left out to increase readability.

```
(defmacro network [#*layers]
    ; Irrelevant internals ...

    (defn check-amount [layer-exps]
        (if (< (len layer-exps) 2)
            (raise (Exception "Structural error: Cannot create a network with
    fewer than 2 layers."))))

    (check-amount layers)
    (check-input ...)
    (setv shape ...)
    (check-incompatibilities ...)
    `(do ...))
```

Listing 9: Compile-time analysis of the amount of layers.

To check whether the amount of layer expressions is not fewer than two, the layer expressions are passed into the **check-amount** function. This function simply verifies whether the length of the sequence of layer expressions is greater than or equal to two. If this is not the case, an exception is thrown and the running of the program is completely halted.

The second part of the compile-time analysis of the network structure is checking whether the first layer expression is one resulting in the creation of an input layer. This is necessary because, under the hood, the Keras Functional API is used to create the actual network. How this is implemented is shown in Listing 10.

```
(defmacro network [#*layers]
    ;Irrelevant internals

    (defn check-input [layer-exps]
        (setv fst (get layer-exps 0))
        (if (not (input? fst))
            (raise (Exception "Structural error: Network architecture's first
    layer must be an input layer."))))

    (check-amount ...)
    (check-input layers)
    (setv shape ...)
    (check-incompatibilities ...)
    `(do ...))
```

Listing 10: Compile-time checking whether the first layer is an input layer.

To check whether the first expression in the sequence of layer expressions is one resulting in the creation of an input layer, the layer expressions are passed into the **check-input** procedure. This procedure checks whether the first layer expression results in the creation of an **input** layer. If the first layer is not an input layer, an exception informing the user of the error is thrown and the execution of the program is halted.

After it has been verified that the first layer is indeed an input layer, the shape of the input layer can then be stored into a variable called **shape**. This variable will be stored globally once the code the **network** macro expands into, is executed. This way, it can be used later on to verify whether the input shape of the network and the shape of the training data are the same. Further explanation is provided in Section 4.3.

The final part of the compile-time analysis conducted by `network` is checking that, when a layer uses the output of another layer as its input, the layers are be of compatible types. The implementation of this step is shown in Listing 11.

```
(defmacro network [#*layers]
    ; Irrelevant internals ...

    (setv 1d-regex (re.compile r"1d"))
    (setv 2d-regex (re.compile r"2d"))
    (setv 3d-regex (re.compile r"3d"))
    (setv name->kind {})

    (defn check-incompatibility [le1 le2]
        (let [k1 (layer-kind le1 (expr-kind le1))
              k2 (layer-kind le2 (expr-kind le2))
              merged (+ k1 k2)]
            (setv 1d? (1d-regex.search merged))
            (setv 2d? (2d-regex.search merged))
            (setv 3d? (3d-regex.search merged))
            (if (or (and 1d? 2d?)
                    (and 2d? 3d?)
                    (and 1d? 3d?))
                (raise (Exception f"Structural error: Layer {k2} cannot use a {k1}
     layer as a compatible input.")))))

    (defn check-w-input-incompatibility [w-input-exp]
        (let [own-kind (layer-kind w-input-exp "input")
              name (get w-input-exp 1)
              inp-kind (get name->kind (get (get w-input-exp 2) 2))
              merged (+ own-kind inp-kind)]
            (setv (get name->kind name) own-kind)
            (setv 1d? (1d-regex.search merged))
            (setv 2d? (2d-regex.search merged))
            (setv 3d? (3d-regex.search merged))
            (if (or (and 1d? 2d?)
                    (and 2d? 3d?)
                    (and 1d? 3d?))
                (raise (Exception f"Structural error: Layer {k1} cannot use a {k2}
     layer as a compatible input.")))))

    (defn check-incompatibilities [layer-exps]
        (setv idx 0)
        (while (< idx (- (len layer-exps) 1))
            ; Get current and next layer
            (setv cur-exp (get layer-exps idx))
            (setv nxt-exp (get layer-exps (+ idx 1)))
            (let [ek (expr-kind cur-exp)]
                (cond
                    ; Layer has some layer specified as its input
                    [(= ek "input") (check-w-input-incompatibility cur-exp)]
                    ; Layer has a name --> may later be used as input to other
    layer --> save name-kind kv-pair
                    [(= ek "named") (do (add-name-kind cur-exp) (
    check-incompatibility cur-exp nxt-exp))]
                    [True (check-incompatibility cur-exp nxt-exp)]))
            (setv idx (+ idx 1))))

    (check-incompatibilities layers)
    `(do ...))
```

Listing 11: Compile-time checking whether layers are compatible.

To check whether all layers are compatible with the ones they use as their input(s), the layer expressions are passed into the `check-incompatibilities` procedure. This procedure iterates over the sequence of layer expressions. In each iteration, the current and the next expression in the sequence are looked at. First, the expression type of the current layer expression is determined. This is the case because each different kind of layer expression (specified input, named, or normal) has their own specific steps that need to be taken to check their compatibility.

If the current layer expression is one that has a specified input, the current expression is passed to the `check-w-input-incompatibility` procedure. This procedure gets the type of the current layer and the type of the layer that is specified as its input. The latter can be found through usage of the `name->kind` dictionary. Once both types have been acquired, regular expressions are used to check whether the types are compatible. If the layer and its specified input are incompatible, an exception alerting the user of this error, is thrown.

If the current layer expression is one binding a name to a layer, two things have to be done. First, an entry has to be added to the `name->kind` dictionary because, later on in the sequence, this name might be used to specify the current layer as the input for another layer. Secondly, the compatibility of the current and the next layer has to be verified. To achieve this, both expressions are passed to the `check-incompatibility` procedure.

This second step is also what happens when the current expression is not one specifying input for the current layer or binding a name to the current layer. `check-incompatibility` takes two layer expressions, gets their types, and finally checks whether they are compatible. If they are incompatible, an exception informing the user of this error, is thrown.

With that, the first function of `network`, mainly the compile-time analysis of the network structure, has been discussed. `network`'s second responsibility is to convert the sequence of layer expressions to an actual Keras network. This is done at runtime through the code `network` expands to. When evaluated, this code

- introduces a global variable called `input-shape-1337`
- binds the previously calculated input shape to this variable
- calls `parse-network` with the layer expression to start the process of parsing and then building the network.

Figure 8 graphically shows the expansion of the `network` macro. Since the call to `parse-network` is the last statement inside the `do`'s body, its result will be the result of the entire `network` expression. Listing 12 shows the implementation of `parse-network`.

```
(setv named-layers {})
(setv merging #{"concat" "avg" "max" "min" "add" "sub" "mul" "dot"})

(defn parse-network [layers]
    (setv named-layers {})
    (setv curr-idx 0)
    (setv parsed [])
    (for [layer layers]
        (setv parsed (+ parsed [(process-layer layer curr-idx)]))
        (setv curr-idx (+ curr-idx 1)))
    (build-network parsed))
```

Listing 12: Implementation of `parse-network`.

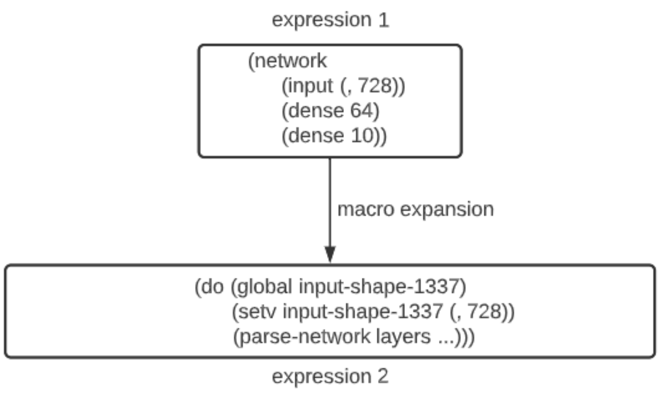
expression 1
(network
(input (, 728))
(dense 64)
(dense 10))
macro expansion
(do (global input-shape-1337)
(setv input-shape-1337 (, 728))
(parse-network layers ...)))
expression 2

Figure 8: Expansion of the `network` macro.

As mentioned, `parse-network` takes a list of layers as it's input. At this point, a layer is either a list tagged with *with-input*, a tuple meaning that the layer has a name, or an ordinary list. `parse-network` iterates over the list of layers. In each iteration, a single layer is processed by calling `process-layer` on the layer and also passing the current index. The implementation of `process-layer` can be seen in Listing 13. The processed layer is then added to the list of processed layers and the current index is incremented. When the parsing of the network has been completed, `build-network` is called on the list of parsed layers. Before moving on to the implementation of `build-network`, first the processing of a single layer needs to be looked at.

```
(defn process-layer [layer idx]
    (if (isinstance layer tuple)
        (let [name (get layer 0)]
            (setv (get named-layers name) idx)
            (setv layer (get layer 1))))
    (if (with-input? layer)
        (let [actual-layer (get layer 1)
              input (get layer 2)]
            (+ [(kind->layer actual-layer)] [input]))
        (kind->layer layer))) ; Layer has no specified input
```

Listing 13: Implementation of `process-layer`.

The first thing that happens in `process-layer` is checking whether the received layer is a tuple. If this is the case, the layer has a specified name. The name, together with the index at which the layer occurred are then added to the `named-layers` dictionary for later use. Secondly, it is checked whether the layer has a specified input. If this is the case, the layer is destructured into the actual layer and its specified input. Processing the layer then boils down to returning a list containing `kind->layer` called on the actual layer and the input layer. If the layer does not have a specified input, processing it is the same as calling `kind->layer` on it.

```
(defn kind->layer [layer]
    (let [kind (layer-kind layer)]
        (cond
            [(= kind "input") (layers.Input (shape layer))]
            ; Other types of layers ...
            ; Merging layers
            [(in kind merging) layer])))
```

Listing 14: Implementation of `kind->layer`.

The purpose of `kind->layer` is to convert a layer in list form to an actual Keras layer, based on the layer's kind. This means `kind->layer` dispatches based on the type and returns a Keras layer of the correct type that is parameterized in the way specified by the programmer. The only exception to this are the merging layers. Whenever a merging layer is encountered, the layer is simply returned. This is due to the way the Keras Functional API functions and is explained in detail in Section 5.1.1.

When the network has been parsed as discussed above, it is then passed to `build-network`. `build-network` takes in a list of parsed layers and converts this list into an actual Keras network. Its implementation can be found in Listing 15.

```
(defn build-network [layers]
    ; variabele for DAG creation
    (global DAG-data)
    (setv DAG-data [])
    ; process input
    (setv input (get layers 0))
    (setv previous-layer input)
    (setv DAG-data (+ DAG-data [input.name]))
    (setv idx 1)
    (for [layer (except-first layers)]
        ; Merging layers or layers with a specified input are present as lists
        (if (isinstance layer list)
            ; Check wheter the layer is a merging layer
            (if (in (get layer 0) merging)
                (do (setv applied (create-merge layer layers))
                    (setv DAG-data (+ DAG-data [(create-int-merge-data applied
    layer)]))
                    (setv previous-layer applied)
                    (setv (get layers idx) applied))
                ; Otherwise it is a layer with a specified input
                (let [actual-layer (get layer 0)
                      input (name->layer (get layer 1) layers)]
                    (do (setv applied (actual-layer input))
                        (setv DAG-data (+ DAG-data [[applied.name (get
    named-layers (get layer 1))]]))
                        (setv previous-layer applied)
                        (setv (get layers idx) applied))))
            ; Unnamed layer, no specified input -> use previous layer as input
            (do (setv applied (layer previous-layer))
                (setv DAG-data (+ DAG-data [applied.name]))
                (setv previous-layer applied)
                (setv (get layers idx) applied)))
        (setv idx (+ idx 1)))
    (setv DAG-data (cut-names DAG-data))
    (keras.Model input previous-layer))
```

Listing 15: Implementation of `build-network`.

The first thing that happens is the declaration and initialisation of a global variable `DAG-data` This variable will be filled with information needed to construct a directed acyclic dataflow graph that has the same topology as the network. This is explained in detail in Section 4.2. `build-network` then starts iterating over the parsed layers. In each iteration, one layer is processed. The different steps this processing phase consists of is once again based on the type of layer that has to be processed.

If the current layer is a merging layer, an actual Keras merging layer has to be created. This is done in `create-merge`, the implementation of which can be seen in Listing 16. This procedure first builds a list containing the actual Keras layers to be merged in the merging layer. Afterwards, it dispatches based on the type of merging layer to be created and it returns an actual Keras merging layer that merges the correct layers.

```
(defn create-merge [merge layers]
    (setv to-merge [])
    (for [name (except-first merge)]
        (setv to-merge (+ to-merge [(name->layer name layers)])))
    (let [kind (merge-kind merge)]
        (cond
            [(= kind "concat") ((keras.layers.Concatenate) to-merge)]
            ; Other merging layers ...
            [(= kind "dot") ((keras.layers.Dot) to-merge)])))
```

Listing 16: Implementation of `create-merge`.

After the applied merging layer has been created, some information is added to the `DAG-data`. This data is created by calling `create-int-merge-data`. The information added to `DAG-data` consists of the name of the created merging layer and the indexes in the `parsed` list at which the layers it merges can be found.

The final steps taken to process a merging layer are to update `previous-layer` to contain the value of the applied Keras merging layer that was just created, and to update the value of the layer in the `parsed` list to also contain the actual Keras applied layer that was just created.

The next type of layer that could be encountered when building the network is a layer that has a specified input. Layers that have a specified input should not be applied to the previous layer to connect them together but should instead be applied to the layer they have specified as their input. This is exactly what happens in `build-network`. Afterwards, information about this specific input is added to `DAG-data`. Finally, the `previous-layer` and entry in `parsed` are updated just like was the case when processing a merging layer.

If the layer that is currently being processed is not a merging layer, nor a layer with a specified input, it just uses the output of the previous layer as its input. The current layer is simply applied to the previous layer, information is added to the `DAG-data` and the `previous-layer` and entry in `parsed` are updated.

Finally, the last step in the creation of the actual Keras network can be discussed. This last step consists of calling `keras.model` with both the input layer, and final layer of the network. This makes it so that `network` returns an actual Keras network that has the topology specified by the programmer.

4.2 Creating Models

Brackeras also offers the programmer with a way to convert their newly created network architecture to a trainable model. This is supported through the `model` macro, the implementation of which can be seen in Listing 17.

```
(defmacro model [network [optimizer "rmsprop"] [loss None] [metrics None]]
    (setv already-compiled-message f"{network} is already a model. A model cannot
    be compiled more than once.")
    `(if (isinstance ~network reac.r-node)
         (raise (Exception ~already-compiled-message))
         (if (not (= "<class 'keras.engine.functional.Functional'>" (str (type ~
    network))))
             (raise (Exception f"Model can only be called on a network and is
    currently being called on {~network} which is not a network."))
             (do (.compile ~network ~optimizer ~loss ~metrics)
                 (dagify ~network)))))
```

Listing 17: Implementation of the `model` macro.

As can be seen in the code above, the `model` macro has four arguments, three of which have a default value for when they are not specified by the programmer. The first thing that happens when the code which `model` expands into is evaluated is checking whether `model` is being called on an existing model. The second verification that happens is checking whether the parameter that was passed as `network` is indeed a network. This is to ensure that the user is only converting networks into trainable models. These two checks are explained in further detail in Section 5. After these two checks, the conversion process can be started. This conversion phase consists of two parts.

1. Compiling the underlying Keras network into a trainable model with the correct optimizer, loss function, and metrics.
2. Generating the directed acyclic dataflow graph that represents the model and returning this to the user.

The first step is handled by using the Keras `.compile` method on the network, passing the arguments specified by the programmer or their default value. The second step is more extensive and is as such abstracted away in the `dagify` procedure. The implementation of `dagify` is shown in Listing 18.

```
(defn dagify [model]
    (setv name->node {})
    (setv l model.layers)
    (setv input (get l 0))
    ; source has reference to the model can be used for training
    (setv source (create-source input model input.name))
    (setv (get name->node input.name) source)
    (setv previous-node source)
    (for [layer (except-first l)]
        (let [depcies (lookup-depcies layer.name name->node previous-node)]
            (setv new-node (reac.r-node layer depcies layer.name))
            (setv (get name->node layer.name) new-node)
            (setv previous-node new-node)))
    source)
```

Listing 18: Implementation of `dagify`.

Firstly, a source node encapsulating the `input` layer is created. This is shown in Listing 19.

```
(defn create-source [layer network name]
    (reac.r-node layer [] name network))
```

Listing 19: Implementation of `create-source`.

After the initialization phase, `dagify` starts iterating over the model's layers. In each iteration a single layer is processed by first calculating the nodes that encapsulate the layers that the current layer uses as its input. This process is handled by `lookup-depcies`, the implementation of which can be seen in Listing 20. When all dependencies of the node that is being created, are found, a new node encapsulating the current layer is defined.

```
(defn lookup-depcies [name node-dict prev]
    (setv idx 0)
    (while (< idx (len DAG-data))
        (setv cur (get DAG-data idx))
        ; if the entry in the DAG-data is a list, the node layer is a merging
    layer or a layer with
        ; a specified input --> the indexes of the nodes on which it will be
    dependant are in the list
        (if (and (isinstance cur list) (= (get cur 0) name))
            (return (idxs->nodes (except-first cur) node-dict))
            ; if the entry in the DAG-data is equal to the current
            (if (= cur name) (return [prev])))
        (setv idx (+ idx 1))))

(defn idxs->nodes [idxs node-dict]
    (setv nodes [])
    (for [idx idxs]
        (let [found (get DAG-data idx)]
            ; entry in DAG data might be a list
            (if (isinstance found list)
                (setv nodes (+ nodes [(get node-dict (get found 0))]))
                (setv nodes (+ nodes [(get node-dict found)])))))
    nodes)
```

Listing 20: Implementation of `lookup-depcies` and `idxs->nodes`.

To look up the nodes on which another node is dependant, the entry in the `DAG-data` that corresponds to the current node needs to be consulted. `lookup-depcies` iterates over the `DAG-data`'s entries, constantly checking whether the current entry is the correct one. If the correct entry is a list, this means the layer a node is being built for is a merging layer or a layer with a specified input. In this case the indexes at which the dependencies can be found in the `DAG-data` are also present in the list. This list is then passed on to `idxs->nodes` where a list containing the nodes on which the current node is dependant, is built and returned. If the correct entry is not a list, the layer a node is being built for is a layer that uses the previous layer's output as its input and the node that should be built is thus only dependant on the previous node.

When the entire directed acyclic dataflow graph has been constructed, the root node created at the beginning is returned to the user for further use as a model.

At this point, the only implementation that still has to be discussed is the one of the nodes in the dataflow graph.

Attribute	**Explanation**
depcies	A list of nodes on which this node is dependant.
callable	A callable object (such as a Keras layer).
depants	A list of nodes on that depend on this node.
network	A reference to an underlying neural network.
name	The node's name.
val	The node's current value.
h	The node's height in the dataflow graph.

Table 3: Attributes of the `r-node` class.

Reactive nodes are implemented as a class that encapsulates seven attributes. These attributes are initialised in the `__init__` method that is, just like is the case in Python, automatically called when creating a new instance of a reactive node. `__init__`'s implementation can be found in Listing 21.

```
(defclass r-node []
    (defn __init__ [self callable depcies name #* network]
        (setv self.depcies depcies)
        (setv self.callable callable)
        (setv self.depants [])
        (if network
            (setv self.network (get network 0))
            (setv self.network False))
        (setv self.name name)
        (setv self.val default-val)
        (setv self.h (.set-correct-height! self))
        (for [dependency depcies]
            (dependency.add-dependant!  self)))

    (defn add-dependant! [self dependant]
        (setv self.depants (+ self.depants [dependant])))

    (defn set-correct-height! [self]
        (setv cur-max 0)
        (for [dependency self.depcies]
            (setv cur-h dependency.h)
            (when (> cur-h cur-max)
                (setv cur-max cur-h)))
        (+ 1 cur-max))

    ; Irrelevant internals)
```

Listing 21: Initialisation of the `r-node` class.

Two things that need to be discussed in detail are the `set-correct-height!` method and the way the nodes on which the current node is dependant are updated. *Glitch prevention* is crucial when updating the values in a dataflow graph representing a reactive program. One way this could be achieved was by assigning each node a *height* that is one more than the maximum height of all the nodes it depends on. When an update is propagated through the network, the nodes are enqueued into a priority queue where a lower height equals a higher priority. Nodes are then dequeued and updated based on their height so that all nodes with a height i are processed before nodes with a height $i+1$ can be updated. This mechanism is also used in the implementation of reactive nodes.

As can be seen in Listing 21, the list of dependant nodes starts empty. Whenever a node is dependant on another node, it calls their `add-dependant!` method to add themselves to their list of dependant nodes.

Once a DAG of nodes has been created, updates to values of nodes will cause an update of all nodes that depend on them to be propagated through the network. In Brackeras, such an update to the values of the nodes can be caused by calling the model on an input. Just like in Python, this causes the `__call__` method to be invoked. The implementation `__call__` and other methods that are subsequently invoked can be found in Listing 22

```
(setv pq (PriorityQueue))

(defn process-pq! []
    (while (not (.empty pq))
        (setv to-be-updated (get (.get pq) 1))
        (.update-value! to-be-updated)
        (.update-dependants to-be-updated)
        (process-pq!)))

(defclass r-node []
    ; Irrelevant internals

    (defn update-value! [self]
        (setv args (self.get-args))
        (setv self.val ((fn [input] (self.callable.output #* input)) args)))

    (defn set-value! [self val]
        (setv self.val val)
        (self.update-dependants))

    (defn get-args [self]
        (setv args [])
        (for [dependency self.depcies]
            (.append args dependency.val))
        args)

    (defn update-dependants [self]
        (for [dependant self.depants]
            (.put pq (, dependant.h dependant)))
        (process-pq!))

    (defn __call__ [self input]
        (if self.network
            (do (print "called network")
                (setv self.val (self.callable.output input))
                (self.update-dependants))
            (raise (Exception "Only the root of the network can be called")))))
```

Listing 22: Implementation of `__call__`.

Since Brackeras only returns reactive nodes that represent models, it will be assumed from now on that whenever a reactive node is being applied to an input, it is a model being applied to process a given input. The first thing that happens when calling a model is a check whether it is the root that is being called and not some other node of the DAG. If this is not the case, an exception is thrown. In the other case, the layer (callable) that is encapsulated inside the reactive node is called on the input to update the node's value. Afterwards, the `update-dependants` method is called to propagate the change through the dataflow graph.

As long as the priority queue is not empty, `process-pq!` continuously dequeues reactive nodes from the queue and invokes their `update-value!` method, causing the node to update its value based on the values of the nodes it depends upon. After the value has been updated, the nodes `update-dependants` method is invoked, further propagating the update through the dataflow graph.

With that, every aspect of the implementation of providing the user with a trainable model, in the form of a directed acyclic dataflow graph, has been explained and the training of models can now be looked at.

4.3 Training Models

Just like Keras, Brackeras provides the user with a specialised construct to train their models. In Brackeras, this construct is called `fit`. The implementation of the macro implementing `fit` can be seen in Listing 23.

```
(defmacro fit [model x y [batch-size None] [epochs 1] [validation-split 0.0]]
    (setv line model.start_line)
    (setv notmodel-msg f"Usage error at line {line}: Fit can only be called on a model.
     It is currently being called on {model} which is not a model. Try calling (model {model}) first.")
    `(if (not (isinstance ~model reac.r-node))
         (raise (Exception ~notmodel-msg))
         (let [x-shape (except-first (. ~x shape))]
            (if (not (= x-shape input-shape-1337))
                (raise (Exception f"Usage error at line {~line}: Model input shape {input-shape-1337} and training data shape {x-shape} are not the same but should be."))
                (do (setv keras-network (. ~model network))
                    (.fit keras-network ~x ~y ~batch-size ~epochs ~validation-split)
```

```
                        (update-layers! ~model keras-network)
                        ~model)))))
```
Listing 23: Implementation of the `fit` macro.

As discussed in Section 3.3, `fit` accepts six arguments, three of which have a default value. The first thing that happens when the code `fit` evaluates into is executed, is checking whether `fit` is being called with a trainable model. If this is not the case, an exception alerting the user of this mistake is thrown. Next, the shape or *dimensionality* of the training data `x` is determined. This shape is then compared against the previously instantiated variable `input-shape-1337`. If the shapes do not match, the training data is incompatible with the model and an exception is thrown. If the shapes do however match, `fit` moves on to the actual training of the model. For this step, the Brackeras model's underlying Keras model is fetched and the its `fit` method provided by Keras is invoked, passing the arguments passed to the `fit` macro as parameters to the method call. Afterwards, the layers of the Brackeras model, i.e. nodes in the dataflow graph, need to be updated to have the same weight values as their corresponding layer in the underlying Keras network. This is implemented in `update-layers!`.

`update-layers` starts by creating a queue and putting the source node of the directed acyclic dataflow graph into the queue. Then , a `while` loop is entered. This loops runs for as long as there are nodes in the queue. In the first iteration, the source node will be dequeued and the underlying Keras model's trained layer is fetched. The callable inside the Brackeras node (the untrained layer) can just be replaced with the trained layer that was just fetched. After this is done, all nodes dependant on the current node are added to the queue, resulting in a breadth first traversal of the entire dataflow graph.

Finally, `fit` returns the trained Brackeras model to the user, leaving the untrained model unchanged. At this point the user could evaluate how good the model performs on test data. How this is implemented in Brackeras will be discussed in the next section.

4.4 Evaluating Trained Models

The final part in the workflow of creating an artificial neural network with Brackeras is the evaluation of a trained model. To this end, Brackeras provides the user with a specialised construct called `evaluate`, the implementation of which can be seen in Listing 24.

```
(defmacro evaluate [model x-test y-test [verbosity 0]]
    (setv notmodel-msg f"Usage error at line {model.start_line}: Evaluate can only
     be called on a model.
     It is currently being called on {model} which is not a model. Try calling (
    model {model}) first.")
    `(if (not (isinstance ~model reac.r-node))
         (raise (Exception notmodel-msg))
         (do (setv score (.evaluate (. ~model network) ~x-test ~y-test ~verbosity)
    )
             (setv idx 0)
             (for [name (. (. ~model network) metrics_names)]
                 (print f"Test {name}: {(get score idx)}")
                 (setv idx (+ idx 1))))))
```
Listing 24: Implementation of the `evaluate` macro.

Brackeras' `evaluate` invokes the `evaluate` method of the underlying Keras model to make optimal use of the facilities provided by Keras. This method invocation returns a list of numeric values. `evaluate` loops over all these values and prints an informative prompt to the terminal. Why `evaluate` is implemented this way is discussed in detail in Section 5.1.1.

Now that Brackeras' usage and the way it is implemented have been thoroughly discussed, it can now be evaluated to what degree it solves the *visual discrepancy problem* and the `lazy error problem`. This will be examined in the next section.

5 Evaluation

In Section 1, two orthogonal problems that arise when using Keras to define artificial neural networks were identified:

1. The *visual discrepancy problem*: The lack of parallels between the graph based visual representation of a neural network, and the way it is defined.
2. The *lazy error problem*: Errors are only caught at runtime and have uninformative error messages.

Afterwards, the Brackeras domain specific programming language was introduced. This language was specifically created to solve both these issues. The aim of this section is to evaluate whether Brackeras succeeds in its goal, and to identify the specific areas in which it improves on the way neural networks are defined in Keras. Sections 5.1 and 5.2 discuss these point for the *visual discrepancy problem* and *lazy error problem* respectively.

5.1 Discrepancy between Code and Visual Representation

The first problem that was identified with the way neural networks are defined in Keras, is the lack of meaningful, easy to see parallels between the way the neural network is coded, and the graph based visual representation that is used to represent the network. Since Brackeras is inspired by the reactive programming paradigm, defining neural networks in Brackeras should provide a stronger link to their graph based visual representation, while having a simpler and more intuitive syntax.

5.1.1 Simple and Intuitive Syntax

To analyse the syntactical differences between Keras and Brackeras, both languages' implementation of the example from Section 1 will be looked at. The visual representation of this network can be seen in Figure 9.

As is clearly depicted by this figure, this network has a non-linear topology. This means a programmer that wants to use Keras to define this network, cannot use the Sequential API but is forced to use the Functional API. Listing 25 shows the way this network would then be implement when following the same programming style used in the Keras Functional API's documentation [9].

```
input = keras.layers.Input(shape=(784,))
n1 = keras.layers.Dense(64)(input)
n2 = keras.layers.Dense(64)(input)
n3 = keras.layers.Dense(64)(input)
added = keras.layers.Add()([n1, n2])
x = keras.layers.Average()([added, n3])
x = keras.layers.Dropout(0.5)(x)
x = keras.layers.Dense(1, activation="softmax")(x)
my_network = keras.models.Model(inputs=input, outputs=x)
```

Listing 25: Keras code for the neural network in Figure 9.

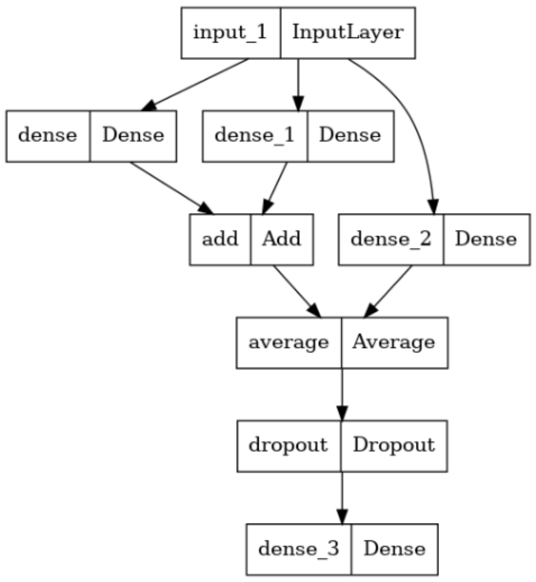

Figure 9: Graph based representation of the neural network in Section 1

As was discussed in Section 1, a number of issues with this way of defining neural networks can be identified:

1. Destructive updating. This can be considered a harmful programming practice since the values of variables may be chanced somewhere behind the programmers back, greatly increasing the debugging complexity [34, 19].

2. Imperative programming. This programming paradigm has a number of weaknesses such as the code quickly becomes very extensive and thus confusing, and the higher risk of errors when editing it [13, 12, 1]. Completely rejecting the imperative programming paradigm would be extremely naive since all programming paradigms have their strengths and weaknesses. However, since this is one of the sources for the discrepancy between the way the neural network is coded and its graph based visual representation, and debugging machine learning applications is notoriously difficult [28], in this case it can be considered suboptimal.

3. The explicit call to `keras.models.Model`. Since creating layers is part of the process of creating a model, they should be grouped together logically instead of being split up into two steps.

The Brackeras implementation of the same exact neural network can now be looked at:

```
(setv my-network
    (network
        (defl in (input (, 784)))
        (defl n1 (dense 64))
        (defl n2 (with-input (dense 64) in))
        (defl n3 (with-input (dense 64) in))
        (defl added (add n1 n2))
        (avg added n3)
        (dropout 0.5)
        (dense 1 "softmax")))
```

Listing 26: Brackeras implementation of the neural network depicted in Figure 9

With this second way of defining the network in mind, the three previously discussed issues can now be looked at once again:

1. The Brackeras code does not contain any destructive updates. This technique often used when using the Keras Functional API has been completely abstracted away from the user since it is handled automatically behind the scenes by `network`. This should result in less errors that are a consequence of the mutation of state somewhere in the program.

2. The Brackeras code is declarative. Brackeras essentially supplies the programmer with a declarative interface, built on top of the imperative Keras API. Enforcing the declarative programming paradigm has a number of advantages such as minimizing data mutability, and making the definition of the network more readable. The first advantage makes it so that errors regarding the mutation of data should occur less when using Brackeras. Secondly, increasing the readability of the code makes it easier for the user to see the parallels between the definition of the neural network and its graph based visual representation. In Brackeras, this can even be taken to the extreme such as shown in Figure 4. There, the sequence of layers provided to `network` is ordered visually to represent the topology of the network, resulting in a striking resemblance between the code and the network's visual representation.

3. In Brackeras, the definition of the layers of neurons of which the network consists, is an inherent part of the definition of the network. This makes it so that they are physically and logically grouped together instead of being two distinct steps. This is a much more logical approach since they always occur together.

This step of defining the network is only the first of the four steps in the workflow identified in Section 3. However, some of the problems discussed are also present in the other three steps. In a normal workflow, the next logical step would be to compile the network into a trainable model. How this would be done in Keras is shown in Listing 27.

```
my_network.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
# Wanting to use a different loss function.
my_network.compile(loss="rmsprop", optimizer="adam", metrics=["accuracy"])
```

Listing 27: Compilation of a network into a trainable model using Keras

As is visible from this example, Keras again reverts to destructive updating when the user wants to change a parameter, such as the loss function, of their trainable model. This could once again result in the user losing track of which parameters are currently being used in the model and could prove to be the source of unwanted behaviour and bugs.

```
(setv my-catcross-model (model my-network "categorical_crossentropy" "adam" ["accuracy"]))
; Wanting to use a different loss function.
(setv my-rms-model (model my-network "rmsprop" "adam" ["accuracy"]))
```

Listing 28: Compilation of a network into a trainable model using Brackeras.

Listing 28 shows how a user would need to convert the network to a trainable model in Brackeras. Again, the destructive updating that was present in the Keras code is completely gone. Calling the Brackeras `model` returns a trainable model. Brackeras' `model` even goes a step further and provides a type-safe API: when the user would try to call `model` with a trainable model, an error is thrown. This means the user is forced to create a completely new trainable model when wanting to use the same network architecture but a different set of parameters, resulting in less error-prone code by making it easier for the programmer to remember the parameters currently being used.

Usually, the next step would be to train the trainable model. Listing 29 shows the way this would be done in Keras.

```
batch_size = 128
epochs = 15
my_network.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
    validation_split=0.1)
```

Listing 29: Training of a trainable model using Keras.

As was discussed in the research training, the training step modifies the weights of the weighted connections between neurons of different layers. In Keras, destructive updating is once again used to update the values of these weights. When a model is trained more than once in an application, this could result in confusion.

```
(setv batch-size 128)
(setv epochs 15)
(setv trained-model (fit my-model x-train y-train batch-size epochs 0.1)
```

Listing 30: Training of a trainable model using Brackeras.

As can be seen in Listing 30, the Brackeras once again has no destructive updates. Instead `fit` returns a new model with layers that have updated weights due to the training, while leaving the untrained model unaffected. This lack of mutation of state should result in code that is less error prone and more easily readable.

Finally, the last step in the workflow of creating a neural network is the evaluation of the trained model. In Keras, when transforming a network to a trainable model, the user can specify a list of metrics to be measured during the evaluation phase. Listing 31 shows how the user would evaluate their network and access the computed values for the specified metrics when using Keras.

```
score = my_network.evaluate(x_test, y_test, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

Listing 31: Evaluation of a trained model using Keras.

The Keras `.evaluate` method returns either a scalar or a list of scalars depending on whether or not the user specified extra metrics to be measured during the evaluation [30]. This means that to access the measured values for each of the metrics, the user needs to look up which metrics they specified and in which order they did. They then need to manually use this information to access the correct entry in the list of scalars returned by `.evaluate`. Once they finally have access to the measurements, they then need to write their own code to print them to the terminal in a nice formatted way.

```
(evaluate trained-model x-test y-test)
```

Listing 32: Evaluation of a trained model using Brackeras.

Listing 32 shows how the user would evaluate their trained model in Brackeras. Brackeras condenses three lines of Keras code into a single line. It also takes away all the burden of the user to remember which metrics they specified and in which order. Instead, the values for each metric are automatically printed in the terminal in a meaningful format.

Brackeras' more readable syntax automatically makes it easier for the user to see the parallels between the definition of the network and its visual representation. However, the real similarities between the code and the programmer's mental model occur through Brackeras' data model. This is discussed in detail in the next section.

5.1.2 Efficient Dataflow Graph

In this section, the differences between Keras' and Brackeras' data model will be discussed.

Keras' data model has a few peculiarities. For example, in Keras a distinction is made between an `Input` layer and all other layers. When using the Keras Functional API, the user is obliged to use an `Input` layer as their network architecture's first layer. This is the case because a Keras `Input` layer instantiates a Keras tensor, which is a symbolic tensor-like object that is augmented with certain attributes. This allows for the creation of a Keras model just by knowing the inputs and outputs of the model [29]. All other Keras layers however are initially not Keras tensors but instances of the Keras `layer` class. In this stage of their lifecycle, they represent callable objects that take as input one or more tensors and that output one or more tensors [8]. This difference is illustrated further in Listing 33.

```
# Input layer
input = keras.layers.Input(shape=(784,))
print(type(input))
# Prints "<class 'keras.engine.keras_tensor.KerasTensor'>"

# Other layers
d = keras.layers.Dense(64)
print(type(d))
# Prints "<class 'keras.layers.core.dense.Dense'>"
```

Listing 33: Difference between the Keras `Input` layer and its other layers.

Layers are however just the basic building blocks of network architectures. To combine layers into a network when using the Keras Functional API, the users needs to sequentially "apply" layers to one another, starting from the `Input` layer. Listing 34 shows this process.

```
# Input layer
input = keras.layers.Input(shape=(784,))
print(type(input))
# Prints "<class 'keras.engine.keras_tensor.KerasTensor'>"

# Other layers
d = keras.layers.Dense(64)
print(type(d))
# Prints "<class 'keras.layers.core.dense.Dense'>"

# Combining layers into a network
combined = d(input)
print(type(combined))
# Prints "<class 'keras.engine.keras_tensor.KerasTensor'>"
```

Listing 34: Combining of Keras layers into a network architecture.

As as visible from this code, when an instance of the Keras `layer` class is called with a Keras tensor, the result will be a new Keras tensor that logically groups together the layers that were connected by the application. Keras basically requires their users to manually accumulate all the layers of their network into a single Keras tensor. Figure 10 illustrates this process. The accumulated Keras tensor can then be passed to a `keras.Model.model` call to create a network architecture. This can be seen in Listing 25.

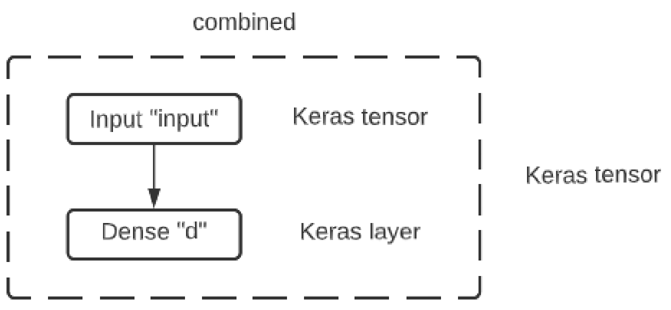

Figure 10: Combining of layers `input` and dense layer `d` into a single Keras tensor `combined`.

However, `keras.Model.model` returns an instance of the Keras `model` class. Calling the returned network architecture a "model" is however completely illogical since the returned value cannot even be trained at this point, let alone be used as an actual machine learning model, i.e. to find patterns or make decisions from a previously unseen dataset [17]. This is shown in Listing 35.

```
input = keras.layers.Input(shape=input_shape)
x = keras.layers.Dense(64)(input)
x = keras.layers.Conv2D(32, kernel_size=(3, 3), activation="relu")(x)
x = keras.layers.MaxPooling2D(pool_size=(2, 2))(x)
x = keras.layers.Conv2D(64, kernel_size=(3, 3), activation="relu")(x)
x = keras.layers.MaxPooling2D(pool_size=(2, 2))(x)
x = keras.layers.Flatten()(x)
x = keras.layers.Dropout(0.5)(x)
x = keras.layers.Dense(num_classes, activation="softmax")(x)
model = keras.models.Model(inputs=input, outputs=x)

# Training the "model"
model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split
    =0.1)
# Output in terminal: RuntimeError: You must compile your model before training/
    testing. Use `model.compile(optimizer, loss)`.
```

Listing 35: Training a Keras "model". (From [7])

This creates some confusing situations for first time users and is stylistically horrendous since a value that does not represent a model should not be called a model. It is only after the user invokes the "model's" `compile` function, that the model can be trained and may be considered a model in machine learning terms. Furthermore, the Keras documentation claims a Keras model groups together layers while in reality still burdening the user with manually grouping together the Keras layers into a single Keras tensor [5]. With that, two problems with the Keras data model can be identified:

1. The fact that the types of all layers, except for the `Input` layer, change during their lifecycle. This change depends on whether or not they are being used to create a network architecture while layers do not have a reason to exist outside the context of the creation of a network architecture.

2. Network architectures and trainable models both being instances of the keras `model` class while they are fundamentally different things.

Meanwhile in Brackeras:

1. Layers are implemented as macros that expand to lists. These lists are of little use outside the `network` macro, where they are used to construct a network architecture. Furthermore, Brackeras' layer expressions cannot be accessed when they are being used within the context of network creation, avoiding the problem Keras has.

2. In Brackeras, network architectures and trainable models have different data types, emphasizing to the user that these are two different things. Network architectures are instances of the Keras `model` class of which the `compile` method has not been called yet. Brackeras trainable models are a reactive node functioning as a handle to the DAG representing the model. This should make it so that the user makes less mistakes confusing the two and should result in more robust code.

Furthermore, representing the model with a directed acyclic dataflow graph like in Brackeras has some other advantages. First, Brackeras dataflow graph has been made more efficient by limiting the amount of updates that are propagated through the network. How this has been achieved can be seen in Listing 36

```
(defn process-pq! []
    (while (not (.empty pq))
        (setv to-be-updated (get (.get pq) 1))
        (setv changed? (.update-value! to-be-updated))
        (if changed? (.update-dependants to-be-updated))
        (process-pq!)))

(defclass r-node []
    ...
    (defn update-value! [self]
        (setv args (self.get-args))
        (setv new-val ((fn [input] (self.callable.output #* input)) args))
        (setv self.val new-val)
        (not (= new-val self.val)))
    ...)
```

Listing 36: Efficiency measures in the Brackeras model DAG.

As is visible from the code, the updating of the value of a certain node returns a Boolean representing whether the value contained in the node has changed. If the node's value does not change, neither will the values of the nodes that depend on the current node. This means the update does not need to be propagated to the nodes dependant on the current node.

Secondly, since the user is passed a handle to the DAG representing the model and the DAG's nodes store their previously computed value, the user has access to these intermediate values. These values might prove useful for fine-tuning the layer or even help explain the model. When using Keras, the user has no access to these intermediate values. There, the user can access a model's layers, but the individual layers do not have an attribute that contains their value [8].

Finally, increasing the size of deep learning models (layers and parameters) yields better accuracy for complex tasks such as computer vision and natural language processing. However, there is a limit to the maximum model size you can fit in the memory of a single GPU [14]. Furthermore, depending on the dimensionality of the input and the model's complexity, the computations happening inside the model's layers might require billions of flops [3]. Because this is the case, neural networks can benefit greatly from being distributed across multiple devices. There are generally two ways to distribute a neural networks computations across multiple devices:

- *Data parallelism*, where a single model gets replicated on multiple devices or multiple machines. Each of them processes different batches of data, then they merge their results.

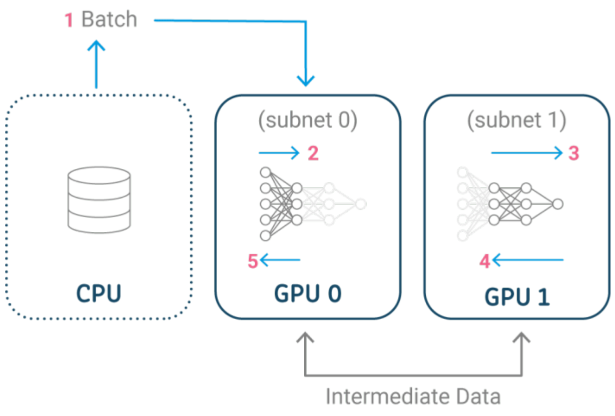

Figure 11: Visual representation of model parallelism. (From [15])

There exist many variants of this setup, that differ in how the different model replicas merge results, in whether they stay in sync at every batch or whether they are more loosely coupled, etc [6].

- *Model parallelism*, where different parts of a single model run on different devices, processing a single batch of data together. This works best with models that have a naturally-parallel architecture, such as models that feature multiple branches [6]. This is visualized in Figure 11.

These points have been directly taken from the Keras documentation and the point on model parallelism specifically is the only mention of model parallelism in the entire Keras documentation apart from a small example in the *Frequently Asked Questions* section. This describes how to distribute training over multiple GPU's of the same device. This leads to the suspicion that Keras does not have a fleshed out way of providing the user with the possibility to deploy their trained networks in a distributed way like is depicted in Figure 12.

However, in Brackeras models are implemented as directed acyclic dataflow graphs that are made out of reactive nodes. This inherent modularity makes Brackeras models particularly well suited for model parallelism across different devices, where each device could encapsulate a single reactive node. A single device would then only be responsible for the computations happening inside of a single layer of the model. However, it still needs to be asserted that no glitches occur within this distributed environment. The problem being that true distributed reactive programming cannot be achieved by naively connecting the dots among single (individually glitch-free) reactive applications. On the contrary, dedicated logic must ensure glitch freedom as a global property of the entire system. Re-using topological sorting in the distributed setting would however force a singlethreaded, centralized execution of updates across the entire application – an approach that can be consider unacceptable [27]. However, change propagation algorithms that works without centralized knowledge about the topology of the dependency structure among reactive nodes and avoids unnecessary propagation of changes, while retaining safety guarantees (glitch freedom), are being researched and are already described in publications such as [11, 18].

Actually implementing the ability for users to distribute Brackeras, and a distributed change propagation algorithm and combining the two to support distributed neural networks in Brackeras would however be beyond the scope of this bachelor's thesis. It can however be concluded that the data model proposed by Brackeras is much more suitable for distributed contexts than

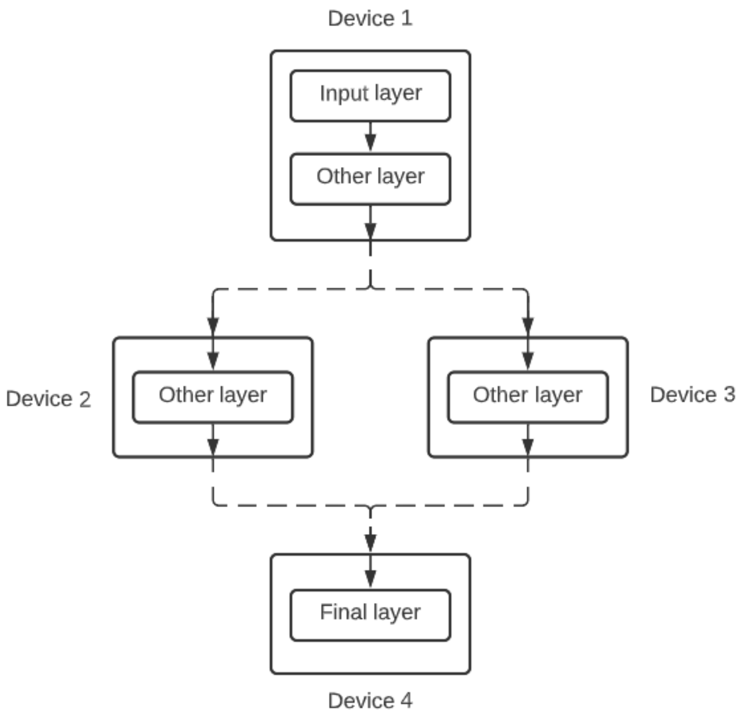

Figure 12: Visual representation of model distribution across different devices.

Keras' data model.

In short, Brackeras' data model discourages the user to create layers outside of the context of network creation and, unlike Keras, it emphasizes the difference between a network architecture, and a trainable model. Furthermore, it enables the user to access the computation's intermediate values and proves more suitable for distributed contexts than Keras' data model. Finally, and most importantly, Brackeras models are implemented as directed acyclic dataflow graphs. This makes it so that the program's data model directly represents the programmers mental model of the neural network, i.e. its graph based visual representation. Consequently, this might make it easier for the programmer to reason about the program and it diminished the discrepancy between the code used to define the network and its visual representation first identified in Section 1.

5.2 Better Errors

The second problem that was identified with the way artificial neural networks are defined in Keras, was the fact that errors are often only caught at runtime. Furthermore the error messages are often found uninformative and cryptic, while they should be as informative and clear as possible since they are the first tool provided to the programmer to help solve the problem. The next two sections will provide an insight into what errors are often made by programmers, and compare Keras' and Brackeras' error messages respectively.

5.2.1 Error Overview

In order to evaluate in what ways Brackeras' error messages improve on Keras' ones, the frequency and severity of the errors made by programmers using Keras need to be inquired into. To aid in this process, a distinction between different kinds of errors is made and new terminology regarding these different kinds of errors is introduced:

- *Structural errors.* These are errors the programmer might make regarding the *structure* of the network architecture. These errors include but are not limited to:
    1. Not having an input layer as the network architecture's first layer / errors regarding the accumulation of layers into a single Keras tensor and the difference in type between instances of the `layer` class and Keras tensors.
    2. Trying to create a network architecture with less than two layers.
    3. Having a certain layer use the output of another layer it is incompatible with as its input / errors regarding data of different shape to the layer's shape being used as input.
- *Usage errors.* These are errors regarding the *usage* of the network architecture and the model. These errors include but are not limited to:
    1. Trying to train a network architecture/model that has not been compiled yet.
    2. Using training data that has a different shape compared to the model's input layer.

The frequency and severity of the mentioned structural errors will be discussed first.

Structural Error	**Google Search Results**
Not having an input layer as the network architecture's first layer / errors regarding the accumulation of layers into a single Keras tensor and the difference in type between instances of the `layer` class and Keras tensors.	8
Trying to create a network architecture with less than two layers.	
Having a certain layer use the output of another layer it is incompatible with as its input / errors regarding data of a different shape to the layer's shape being used as input.	35

When googling the error messages they throw, both the first and third structural error seem quite frequent. Especially the third structural error, seems severe and widespread. The amount of occurrences of second structural error is difficult to determine. This is because Keras allows for the creation of network architectures with a single layer, and even allow this network architecture to be converted to a trainable model, only throwing a vague error once the user tries to train the trainable model by calling `fit`. This makes it virtually impossible to find specific occurrences of this error on the web. In some ways, this is even more worrisome then the frequent occurrences of the first and third structural error that are relatively easy to find and thus acquire extra information that might aid in debugging.

The frequency and severity of the mentioned usage errors will be discussed next.

Usage Error	Google Search Results
Trying to train a network architecture/model that has not been compiled yet.	13
Using training data that has a different shape compared to the model's input layer.	

When googling the error messages respectively thrown by these two usage errors, it is visible that the first error occurs quite often. This means the theory about network architectures being called models in Keras causing confusion is probable and this is a severe error. Just like was the case for the second structural error, specific occurrences the second usage error are difficult to track down. This time it is due to the fact that the error messages for the shape of the training data not matching the shape of the input layer, and the shapes of two consecutive layers in the model not matching, are the same.

Further adding to the severity of these errors is the fact that, in Keras, all of them are caught at runtime. All of this while machine learning applications often spend a large portion of their runtime loading big libraries and huge amounts of data, and training model. This has the potential to greatly diminish developers' productivity while error messages should instead increase developers' productivity.

Now that the frequency and severity of the different structural and usage errors has been discussed, the second part of this section can focus on what makes error messages good or bad. It is crucial to understand this in order to be able to make a true qualitative comparison of Keras' and Brackeras' error messages in Section 5.2.2. First, a distinction between runtime error messages and compile-time error messages needs to be made.

- *Compile-time errors* are errors that are detected by the compiler during the development of the code. They are either lexical, syntactic, or semantic errors. A compile-time error will hinder the execution or running of the complete code until the error is resolved [2, 26].
- *Runtime errors* on the other hand are not detected by the compiler but are caught when executing or running the complete code. These errors are often logical errors in the code [2, 26].

When looking at these two definitions, some clear advantages of compile-time errors over runtime errors can be identified. Unlike runtime errors, they are caught much earlier, which is a big advantage considering the often long run times of machine learning applications. Furthermore, they completely stop the developer from executing faulty code, while at the same time offering the developer the possibility to fix the problem during the development of the code. Taking these advantages into consideration, it can be concluded that compile time errors are generally better than runtime errors.

With that, what makes an error message "good" can now be discussed. Just as there are principles for programming language design [4], or software can be characterized by its inherent nature, it seems natural to think of a set of principles to guide error message design. [32] proposes a set of desirable characteristics of error messages. The proposed principles are inspired by the body of knowledge in HCI and were derived from examples of actual errors and the author's experience as a programmer and as an educator. In particular, the guidelines of heuristic evaluation provide good insight to define how errors should be. One of these general principles behind the heuristic evaluation has just to do with providing good error messages, so that the user can easily diagnose and recover from errors. The rest of the heuristics provide rich information that can be applied to consider how the compiler-programmer interaction should be and, since most of this interaction is in the form of the compiler error messages, these other heuristics can in turn also guide the design of the error messages themselves. The following is the set of proposed principles and heuristics from [32]. Guidelines about writing good error messages from other sources are put together with the principle they enforce the most.

- **Clarity and Brevity** (aesthetic and minimalist design, recognition rather than recall).
  - Do not print out information a developer could easily find in their code. [20]
  - Use language the developer will understand, not compiler-speak. [20, 23, 16, 25]
- **Specificity** (recognition rather than recall; help user recognize, diagnose and recover from errors).
  - Provide the user with information about what led to the error? What was the code trying to do when it failed? [22, 25]
  - Provide the user with information about the error itself. What exactly failed? [22, 23, 16, 25]
  - Provide the user with information about what needs to be done in order to overcome the error. [22, 23, 25]
- **Context Insensitivity** (consistency and standards).
- **Locality** (flexibility and efficiency of use)
  - The error location is so important since that's where the red squiggly goes in an IDE. Pick the smallest possible location at the operation which triggered the error. If the user is writing a new operation the error will point to where they are working. If the user is refactoring the error will point to all the operations which need to change. [20]
- **Proper Phrasing** (match between system and the real world).
- **Consistency** (consistency and standards).
  - Use correct English grammar. [20]
  - Uniform voice and style: The specific style chosen doesn't matter too much, but one should settle on either active vs. passive voice, apply consistent casing, either finish or not finishes messages with a dot, etc. [22]
- **Visual Design** (aesthetic and minimalist design; error prevention).
  - Keep error messages short. Preferably a single, clear, sentence. This format works best in an IDE context. [20, 16]
- **Extensible Help** (help and documentation).

These principles provide a solid base for evaluating error messages' quality and will be used to that effect in the next section where Keras' and Brackeras' error messages will by directly compared.

5.2.2 Comparison to Keras' Errors

In this section, Keras' and Brackeras' handling of the structural and usage errors discussed in the previous section will be directly compared and evaluated using the eight previously discussed *principles for writing good error messages.*

The first errors that will be discussed are the structural errors. More specifically, the structural error where the network does not have an input layer as the first layer / errors regarding the accumulation of layers into a single Keras tensor and the difference in type between instances of the layer class and Keras tensors, will be looked at first.

```
# Minimal example code causing the error
x = keras.layers.Dense(64)
x = keras.layers.Dense(32)(x)

# Output in terminal
Traceback (most recent call last):
  File ".../Desktop/bachelorproef/experiments/experiments.py", line 88, in <module
    >
    x = keras.layers.Dense(32)(x)
  File ".../.local/lib/python3.9/site-packages/keras/utils/traceback_utils.py",
    line 67, in error_handler
    raise e.with_traceback(filtered_tb) from None
  File ".../.local/lib/python3.9/site-packages/keras/engine/input_spec.py", line
    197, in assert_input_compatibility
    raise TypeError(f'Inputs to a layer should be tensors. Got: {x}')
TypeError: Inputs to a layer should be tensors. Got: <keras.layers.core.dense.
    Dense object at 0x7f0a4ca1c220>
```

Listing 37: Not using an `Input` layer as the first layer of the network architecture in Keras.

Listing 37 shows what happens in Keras when the user does not have an input layer as the first layer of their network architecture. As is visible from the output in the terminal, the error was caught at runtime somewhere deep inside the workings of the Keras library implementation.

This error message does not conform to the "Clarity and brevity" principle since this principle requires error messages to be short since it is a relatively common student behavior not to fully read error messages, which may lead the students to misinterpret or not to follow the error messages, even if they are informative and helpful. This is the case in this specific example where the developer's eye is attention is automatically drawn to the `TypeError`. This means they might miss import information, such as the line the error originated from, from the `Traceback` section of the error message. The `Traceback` section itself is too long, showing calls made to functions that are internal to Keras' implementation, needlessly exposing the programmer to this information.

Because the error does not specifically mention that is was caused by not having an `Input` layer as the first layer of the network architecture, it also does not conform to the "Specificity" principle.

Since this is the only context in which this specific error can occur, it automatically complies with the "Context-insensitivity" principle.

When strictly following the proposals of [32], the "Locality" principle does not apply to runtime errors. However, since Brackeras catches this error at compile time, the locality of the Keras error will also be discussed. In this case, the Keras error does not comply with the "Locality" principle since the error is caught deep within the Keras library implementation.

In some ways the Keras error complies with the "Proper phrasing" principle. It has a neutral tone and does not blame or condemn programmers. However, the inclusion of `< keras.layers.core.dense.Dense object at 0 x7f0a4ca1c220 >` is technical and could have been made less technical by instead saying `found a dense layer`.

The error has a consistent style, thus conforming to the “Consistency” principle.

The chaotic look of the `Traceback` section and the inclusion of technical details make it so that the error message does not conform to the “Suitable visual design” principle.

Finally, the “Extensible help” principle is more of a vision for having “layered” error messages than a principle that can be used to evaluate the quality of existing error messages. Because this is the case, the conformity to this principle will not be discussed.

```
; Minimal example code causing the error
(setv my-network
    (network
        (dense 64)
        (dense 32)))

; Output in terminal
Traceback (most recent call last):
hy.errors.HyMacroExpansionError:
  File ".../Desktop/bachelorproef/language/errortest.hy", line 107
    (network
        (dense 64)
        (dense 32)))
    ^^
expanding macro network
  Exception: Structural error: Network architecture's first layer must be an input
     layer.
```

Listing 38: Not using an `Input` layer as the first layer of the network architecture in Brackeras.

Listing 38 shows what happens in Brackeras when the user does not specify an input layer as the first layer of their network architecture. As is visible from the output in the terminal, the error was caught at macro-expansion time, thus immediately alerting the user of their mistake. This is in stark contrast to Keras which allows the user to make this mistake and only alerts the user at runtime.

For its macro-expansion time errors, Brackeras automatically limits the length of the `Traceback` section that is inevitable when building a DSL on top of a Python-based language. Because of this, the Brackeras error message is a lot more brief than the one in Keras. It only shows the line in the developer’s code that causes the error, the complete definition of the network architecture to make it easier for the user to identify the network architecture where the problem occurs, and a short description describing the nature of the error to the user. Because of this, the Brackeras error conforms to the “Clarity and brevity” principle.

This error specifically alerts that the lack of an input layer as the network architecture’s first layer is the cause of the error. This means the Brackeras error complies to the Specificity principle.

Since this is the only context in which this specific error can occur, it automatically complies with the Context-insensitivity principle.

The Brackeras error conforms to the Locality principle since the error is caught at the place where it occurs in the user’s code. The fact that the actual location at which the error occurs, both line number and actual context, is shown in the error message adds to the conformity to this principle.

Next, the error has a neutral tone, and is not overly technical. This means it conforms to the Proper Phrasing principle.

Since the error is consistent stylistically it complies with the Consistency principle.

The Brackeras error is split into two parts: Information about where the error occurred, directing the user to the source of the error, and information about the nature of the error. This makes the visual design of the Brackeras error better than that of the Keras error. It could however still be made less cluttered.

Principle	Keras	Brackeras
Clarity and Brevity	☆	★
Specificity	☆	★
Context Insensitivity	★	★
Locality	☆	★
Proper Phrasing	⯪	★
Consistency	★	★
Visual Design	☆	⯪

Table 4: Keras' and Brackeras' conformity to the design principles for writing good error messages. A fuller star means a higher degree of conformity with the respective design principle.

The degree to which both the Keras and Brackeras error message that is displayed when the user tries to use a different layer than an input layer as the first layer of their network architecture, is summarised in Table 4.

The root cause of this error is the distinction made by Keras between instances of the `layer` class and Keras tensors, and burdening the user with manually converting layers into Keras tensors. Another error that might occur due to this difference in types, is trying to instantiate a merging layer, merging two instances of the `layer` class, rather than Keras tensors. Code to reproduce this error can be seen in Listing 39.

```
# Minimal example code causing the error
x1 = keras.layers.Dense(64)
x2 = keras.layers.Dense(64)
added = keras.layers.Add()([x1, x2])

# Output in terminal
Traceback (most recent call last):
  File ".../Desktop/bachelorproef/experiments/experiments.py", line 90, in <module>
    added = keras.layers.Add()([x1, x2])
  File ".../.local/lib/python3.9/site-packages/keras/utils/traceback_utils.py", line 67, in error_handler
    raise e.with_traceback(filtered_tb) from None
  File ".../.local/lib/python3.9/site-packages/keras/layers/merge.py", line 87, in build
    if not isinstance(input_shape[0], tuple):
TypeError: 'NoneType' object is not subscriptable
```

Listing 39: Merging two layers instead of two Keras tensors.

This error is once again caught at runtime somewhere deep within the Keras library implementation. It is clear pretty quickly that it does not comply to the Clarity and Brevity, Specificity, Locality, and Visual Design principles. Just like the previous error, it has some elements conforming to the Proper Phrasing principle and it is consistent. However, unlike the previously discussed error, this one does not conform to the Context Insensitivity principle because the code in Listing 40 gives rise to a different error message than the code in Listing 39, while they have the same cause.

```
# Minimal example code causing the error
x1 = keras.layers.Dense(64)
x2 = keras.layers.Dense(64)(x1) # Keras tensor
added = keras.layers.Add()([x1, x2])

# Output in terminal
Traceback (most recent call last): ...
```

```
TypeError: Inputs to a layer should be tensors. Got: <keras.layers.core.dense.
    Dense object at 0x7f02dfcea220>
```

Listing 40: Merging a layer and a Keras tensor instead of two Keras tensors.

This means this error message is even less compliant to the design principles than the last one. Meanwhile, in Brackeras it is impossible for developers to cause this error since Brackeras automatically does the necessary conversions of layers into Keras tensors in the background. This is a big advantage of Brackeras over Keras because, of course, the impossibility for the error to occur is better than letting the error occur and displaying a badly designed error message.

The next structural error that will be discussed is trying to create a network architecture with fewer than two layers. Listing 41 shows how to recreate this error in Keras, together with the displayed error message.

```
# Minimal example code causing the error
layer = keras.layers.Input(shape=(28, 28, 1))
my_model = keras.models.Model(inputs=layer, outputs=layer)
my_model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["
    accuracy"])
my_model.fit(x_train, y_train, batch_size=128, epochs=1, validation_split=0.1)

# Output in terminal
Traceback (most recent call last):
  File ".../Desktop/bachelorproef/experiments/experiments.py", line 83, in <module
    >
    my_model.fit(x_train, y_train, batch_size=128, epochs=1, validation_split=0.1)
  File ".../.local/lib/python3.9/site-packages/keras/utils/traceback_utils.py",
    line 67, in error_handler
    raise e.with_traceback(filtered_tb) from None
  File ".../.local/lib/python3.9/site-packages/tensorflow/python/framework/
    func_graph.py", line 1147, in autograph_handler
    raise e.ag_error_metadata.to_exception(e)
ValueError: in user code:

    File ".../.local/lib/python3.9/site-packages/keras/engine/training.py", line
    1021, in train_function  *
        return step_function(self, iterator)
    File ".../.local/lib/python3.9/site-packages/keras/engine/training.py", line
    1010, in step_function  **
        outputs = model.distribute_strategy.run(run_step, args=(data,))
    File ".../.local/lib/python3.9/site-packages/keras/engine/training.py", line
    1000, in run_step  **
        outputs = model.train_step(data)
    File ".../.local/lib/python3.9/site-packages/keras/engine/training.py", line
    860, in train_step
        loss = self.compute_loss(x, y, y_pred, sample_weight)
    File ".../.local/lib/python3.9/site-packages/keras/engine/training.py", line
    918, in compute_loss
        return self.compiled_loss(
    File ".../.local/lib/python3.9/site-packages/keras/engine/compile_utils.py",
    line 201, in __call__
        loss_value = loss_obj(y_t, y_p, sample_weight=sw)
    File ".../.local/lib/python3.9/site-packages/keras/losses.py", line 141, in
    __call__
        losses = call_fn(y_true, y_pred)
    File ".../.local/lib/python3.9/site-packages/keras/losses.py", line 245, in
    call  **
        return ag_fn(y_true, y_pred, **self._fn_kwargs)
    File ".../.local/lib/python3.9/site-packages/keras/losses.py", line 1789, in
    categorical_crossentropy
        return backend.categorical_crossentropy(
    File ".../.local/lib/python3.9/site-packages/keras/backend.py", line 5083, in
    categorical_crossentropy
        target.shape.assert_is_compatible_with(output.shape)

    ValueError: Shapes (None, 10) and (None, 28, 28, 1) are incompatible
```

Listing 41: Creating a network architecture with fewer than two layers.

As is visible from the minimal code example causing the error, Keras lets the developer create a network architecture with a single layer, compile it into a trainable model, and only catches the error while attempting to train the model. Since the error is only caught during training, the `Traceback` section of the error message is extremely long. This exposes huge amount of needless information to the developer and makes it so that this error does not comply with the Clarity and Brevity principle. Since the error does not communicate the source of the problem to the programmer, nor what needs to be done to overcome the error, it does not comply with the Specificity principle. It does however comply with the Context insensitivity principle since invoking the model's `evaluate` method produces the same error message. As mentioned, the error is only caught during training instead of at the place where it is created. This means the error does not comply with the Locality principle. Just like the previous two Keras errors that were discussed, it complies to some element of the Proper Phrasing argument such as not having a negative tone. The extremely long `Traceback` section however exposes a large quantity of technical information to the developer. Finally, the error message itself is consistent with the other Keras errors discussed thus far and it does not comply with the Visual Design principle since it is extremely long.

Listing 42 shows how to cause such an error in Brackeras and the error message that is displayed in the terminal.

```
; Minimal example code causing the error
(setv my-network
    (network
        (input (, 28 28 1))))

; Output in terminal
Traceback (most recent call last):
hy.errors.HyMacroExpansionError:
  File ".../Desktop/bachelorproef/language/errortest.hy", line 107
    (network
        (input (, 28 28 1))))
    ^^
expanding macro network
  Exception: Structural error: Cannot create a network with fewer than 2 layers.
```

Listing 42: Creating a network architecture with fewer than two layers in Brackeras.

In Brackeras, this error is caught at macro expansion time instead of at runtime. Brackeras also does not allow the user to create this faulty network architecture and compile it to a trainable model, only to alert the user of the error when they attempt to train the model. The Brackeras error conforms to the Clarity and Brevity principle since it is short and does not expose the developer to overly technical information about the error. Secondly, it also complies with the Specificity principle since it explicitly communicates to the user that having fewer than two layers in their network architecture is the error's source. Because exactly the same error is thrown when a developer tries to create a network architecture with no layers, the Brackeras error conforms to the Context Insensitivity principle. Furthermore, the error is immediately caught at the place where it occurs, thus complying with the Locality principle. Since the error message explains the error's cause to the user, while informing them how to resolve the error, and does not expose the user to unnecessary technical information, it conforms to the Proper

Phrasing principle. Finally the error is consistent with the other Brackeras structural errors and has a clear visual design showing the user the error's location and nature. It could however still be made more minimal. Table 5 shows a summary of both Keras' and Brackeras' errors to the previously discussed principles.

Principle	**Keras**	**Brackeras**
Clarity and Brevity	☆	★
Specificity	☆	★
Context Insensitivity	★	★
Locality	☆	★
Proper Phrasing	⯪	★
Consistency	★	★
Visual Design	☆	⯪

Table 5: Conformity to the design principles of writing good error messages of Keras' and Brackeras' error messages concerning network architectures with less than two layers.

With that, the final Structural error discussed in the previous section can be looked at. This error is caused by having a certain layer use the output of another layer it is incompatible with as its input or, more generally, by having data of a different shape to the layer's shape being used as input. Listing 43 shows how such an error might occur in Keras, together with the displayed error message.

```
# Minimal example code causing the error
input = keras.layers.Input(shape=input_shape)
x = keras.layers.Conv2D(32, kernel_size=(3, 3), activation="relu")(input)
x = keras.layers.MaxPooling3D(pool_size=(2, 2, 2))(x)

# Output in terminal
Traceback (most recent call last):
  File ".../Desktop/bachelorproef/experiments/experiments.py", line 43, in <module>
    x = keras.layers.MaxPooling3D(pool_size=(2, 2, 2))(x)
  File ".../.local/lib/python3.9/site-packages/keras/utils/traceback_utils.py", line 67, in error_handler
    raise e.with_traceback(filtered_tb) from None
  File ".../.local/lib/python3.9/site-packages/keras/engine/input_spec.py", line 214, in assert_input_compatibility
    raise ValueError(f'Input {input_index} of layer "{layer_name}" '
ValueError: Input 0 of layer "max_pooling3d" is incompatible with the layer: expected ndim=5, found ndim=4. Full shape received: (None, 26, 26, 32)
```

Listing 43: Creating a network architecture with fewer than two layers.

As was the case with the three previously discussed Keras errors, this error is once again caught at runtime. Once again, the `Traceback` section is extremely long, needlessly exposing the programmer to information that would not help them resolve the error. This means the error does not comply with the Clarity and Brevity principle. Furthermore, it does not conform to the Specificity principle since it does not communicate to the user which sequence of layers exactly causes the error. This also means the developer does not necessarily know how to resolve the error after reading the message. Because this error only occurs during the accumulation of layers into a single Keras tensor, it complies to the Context Insensitivity principle. It also complies with the Locality principle since the error is caught at the place where it occurs, albeit at runtime. However, it does not comply with the Proper Phrasing principle since the wording of the error leaves the developer guessing and is extremely technical. To make matters worse, it relies on Keras' internal naming of layers to construct the error message as can be seen by the inclusion of "max_pooling3d". This is illustrated in Listing 44.

```
# Minimal example code causing the error
input = keras.layers.Input(shape=(28,28,28,1))
x = keras.layers.Conv3D(32, kernel_size=(3, 3, 3), activation="relu")(input)
x = keras.layers.MaxPooling3D(pool_size=(2, 2, 2))(x)

input2 = keras.layers.Input(shape=(28,28,1))
y = keras.layers.Conv2D(32, kernel_size=(3, 3), activation="relu")(input2)
y = keras.layers.MaxPooling3D(pool_size=(2, 2, 2))(y)

# Output in terminal
Traceback (most recent call last): ...
ValueError: Input 0 of layer "max_pooling3d_1" is incompatible with the layer:
    expected ndim=5, found ndim=4. Full shape received: (None, 26, 26, 32)
```

Listing 44: Dependency of error message on internal Keras name.

Because the error now occurs when combining the second MaxPooling3D layer with the Conv2D layer, the error message now refers to "max_pooling3d_1" since that is the name that is internally used for this layer. However the user usually has no idea of the implementation detail. This means the user has no idea which MaxPooling3D layer is causing the error. Finally, the error is consistently designed and does not comply to the Visual Design principle due to its length and cluttered look.

Now the same error in Brackeras can be looked at. The code to recreate it, together with the output in the terminal can be seen in Listing 45.

```
; Minimal example code causing the error
(setv my-network
    (network
        (input (. 28 28 1))
        (conv2d 32 [3, 3])
        (maxpool3d [2, 2, 2])))

; Output in terminal
Traceback (most recent call last):
hy.errors.HyMacroExpansionError:
  File ".../Desktop/bachelorproef/language/errortest.hy", line 107
    (network
        (input (. 28 28 1))
        (conv2d 32 [3, 3])
        (maxpool3d [2, 2, 2])))
    ^^
expanding macro network
  Exception: Structural error: Layer maxpool3d cannot use a conv2d layer as a
    compatible input.
```

Listing 45: Incompatible layers in Brackeras.

Just like the previously discussed Brackeras errors, this error conforms to the Clarity and Brevity principle, the Specificity principle, the Context Insensitivity principle, the Locality principle, the Proper Phrasing principle, and the consistency principle. Once again, the visual design could be improved upon.

```
; Minimal example code causing the error
(setv mnist-network
    (network
        (input input-shape)
        (conv2d 32 [3, 3])
        (maxpool2d [2, 2])
```

Principle	**Keras**	**Brackeras**
Clarity and Brevity	☆	★
Specificity	☆	★
Context Insensitivity	★	★
Locality	★	★
Proper Phrasing	☆	★
Consistency	★	★
Visual Design	☆	⯪

Table 6: Conformity to the design principles of writing good error messages of Keras' and Brackeras' error messages concerning incompatible layers.

```
        (conv2d 64 [3, 3])
        (maxpool3d [2, 2, 2])))

; Output in terminal
...
    (network
        (input input-shape)
        (conv2d 32 [3, 3])
        (maxpool2d [2, 2])
        (conv2d 64 [3, 3])
        (maxpool3d [2, 2, 2])))
    ^^
...
```

Listing 46: Incompatible layers in Brackeras not suffering from unnecessary dependency.

The Brackeras error also does not suffer from the problem of having the error message be dependant on some internal name. Instead, the part of the macro upto the point of the error is printed out as part of the error message, showing the user exactly where the error occurs. This is shown in Listing 46. Table 6 summarises the conformity of the discussed Keras and Brackeras errors to the design principles for writing good error messages.

With that, all Keras and Brackeras errors concerning the Structural errors identified in this bachelor's thesis have been compared and the Usage errors can now be looked at. The first usage error that will be inquired into is trying to train a network architecture before transforming it to a trainable error. A minimal example to cause this error together with the error message displayed in the terminal can be seen in Listing 47.

```
# Minimal example code causing the error
input = keras.layers.Input(shape=(28,28,1))
x = keras.layers.Conv2D(32, kernel_size=(3, 3), activation="relu")(input)
x = keras.layers.MaxPooling2D(pool_size=(2, 2))(x)
my_model = keras.models.Model(inputs=input, outputs=x)

my_model.fit(x_train, y_train, batch_size=128, epochs=1, validation_split=0.1)

# Output in terminal
Traceback (most recent call last):
  File ".../Desktop/bachelorproef/experiments/experiments.py", line 46, in <module
    >
    my_model.fit(x_train, y_train, batch_size=128, epochs=1, validation_split=0.1)
  File ".../.local/lib/python3.9/site-packages/keras/utils/traceback_utils.py",
    line 67, in error_handler
    raise e.with_traceback(filtered_tb) from None
  File ".../.local/lib/python3.9/site-packages/keras/engine/training.py", line
    3059, in _assert_compile_was_called
    raise RuntimeError('You must compile your model before '
RuntimeError: You must compile your model before training/testing. Use `model.
    compile(optimizer, loss)`.
```

Listing 47: Trying to call `fit` on an instance of the `model` class before its `compile` method was invoked.

As is visible from the code, this error was caught at runtime. Once again, the extremely long `Traceback` section exposes a lot of needlessly complicated information to the developer. This makes it so that the error does not conform to the Clarity and Brevity principle. It does however comply to the Specificity, Context Insensitivity and Locality principles. It complies to the Specificity principle since it clearly communicates the source of the error and a possible way to resolve it to the user. Invoking the `evaluate` procedure of a network architecture that has not been transformed to a trainable model should throw the same error and does so, making it so that the error complies to the Context Insensitivity principle. Since the error is immediately caught at the place where it occurs, albeit at runtime, it conforms to the Locality principle. Since it exposes so many technical details to the programmer, it does not comply to the Proper Phrasing principle. Finally, the error is consistently designed and does not conform to the Visual design principle since due to the message's length.

```
; Minimal example code causing the error
(setv my-network
    (network
        (input input-shape)
        (conv2d 32 [3, 3])
        (maxpool2d [2, 2])))

(setv trained-model (fit my-network x-train y-train batch-size 1 0.1))

; Output in terminal
Traceback (most recent call last):
Exception: Usage error at line 8: Fit can only be called on a model.
     It is currently being called on my-network which is not a model. Try calling
    (model my-network) first.
```

Listing 48: Trying to call `fit` on a network architecture in Brackeras.

Listing 48 shows how this same exact error could be recreated in Brackeras, together with the output in the terminal. Brackeras once again limits the length of the `Traceback` section in an effort not to overwhelm the developer with large amounts of technical details. Useful information such as the line number at which the error occurs are except offered to the programmer through the exception message. This makes is so that the Brackeras error complies to the Clarity and Brevity principle. Since the error also communicates to the developer what went wrong and provides the user with a way to resolve the error, it complies with the Specificity principle. Since trying to call `evaluate` on the network architecture results in the same error message, the error conforms to the Context Insensitivity principle. Furthermore, the error is caught immediately at the place where it occurs, resulting in conformity to the Locality principle. Unlike the Keras error, the Brackeras error does not communicate needlessly technical information to the user, making it so that the error complies with the Proper Phrasing principle. Finally, the error is consistent and complies to the Visual Design principle due to its brevity. Table 7 summarises the conformity of the Keras and Brackeras error messages. Although in this case, Brackeras' error message is not a huge improvement over the one in Keras, it is important to remember that Keras does not make a distinction between network architectures and models while Brackeras does. Because of this distinction, this error should occur way less in Brackeras.

The second and final Usage error that will be discussed in this report is using training data

Principle	Keras	Brackeras
Clarity and Brevity	☆	★
Specificity	★	★
Context Insensitivity	★	★
Locality	★	★
Proper Phrasing	☆	★
Consistency	★	★
Visual Design	☆	★

Table 7: Conformity to the design principles of writing good error messages of Keras' and Brackeras' error messages concerning training of a network architecture.

that has a different shape to the model's input layer. How such an error can be caused in Keras, can be seen in Listing 49.

```
# Minimal example code causing the error
input = keras.layers.Input(shape=(28,28,28,1))
x = keras.layers.Dense(64)(input)
x = keras.layers.Dense(32)(x)
my_model = keras.models.Model(inputs=input, outputs=x)

my_model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])

my_model.fit(x_train, y_train, batch_size=128, epochs=1, validation_split=0.1) # x_train.shape = (28,28,1)

# Output in terminal
Traceback (most recent call last):
  File ".../Desktop/bachelorproef/experiments/experiments.py", line 105, in <module>
    my_model.fit(x_train, y_train, batch_size=128, epochs=1, validation_split=0.1)
  File ".../.local/lib/python3.9/site-packages/keras/utils/traceback_utils.py", line 67, in error_handler
    raise e.with_traceback(filtered_tb) from None
  File ".../.local/lib/python3.9/site-packages/tensorflow/python/framework/func_graph.py", line 1147, in autograph_handler
    raise e.ag_error_metadata.to_exception(e)
ValueError: in user code:

    File ".../.local/lib/python3.9/site-packages/keras/engine/training.py", line 1021, in train_function  *
        return step_function(self, iterator)
    File ".../.local/lib/python3.9/site-packages/keras/engine/training.py", line 1010, in step_function  **
        outputs = model.distribute_strategy.run(run_step, args=(data,))
    File ".../.local/lib/python3.9/site-packages/keras/engine/training.py", line 1000, in run_step  **
        outputs = model.train_step(data)
    File ".../.local/lib/python3.9/site-packages/keras/engine/training.py", line 860, in train_step
        loss = self.compute_loss(x, y, y_pred, sample_weight)
    File ".../.local/lib/python3.9/site-packages/keras/engine/training.py", line 918, in compute_loss
        return self.compiled_loss(
    File ".../.local/lib/python3.9/site-packages/keras/engine/compile_utils.py", line 201, in __call__
        loss_value = loss_obj(y_t, y_p, sample_weight=sw)
    File ".../.local/lib/python3.9/site-packages/keras/losses.py", line 141, in
    __call__
        losses = call_fn(y_true, y_pred)
    File ".../.local/lib/python3.9/site-packages/keras/losses.py", line 245, in
    call  **
        return ag_fn(y_true, y_pred, **self._fn_kwargs)
    File ".../.local/lib/python3.9/site-packages/keras/losses.py", line 1789, in
    categorical_crossentropy
        return backend.categorical_crossentropy(
    File ".../.local/lib/python3.9/site-packages/keras/backend.py", line 5083, in
    categorical_crossentropy
        target.shape.assert_is_compatible_with(output.shape)

    ValueError: Shapes (None, 10) and (None, 28, 28, 1, 32) are incompatible
```

Listing 49: Using training data that has a different shape to the shape of the model's `Input` layer.

As can be seen in the code above, Keras catches this error somewhere deep within the library implementation while attempting to train the model. Just like all of the previous Keras errors, this error also has a terribly long `Traceback` section, making it so that the error does not conform with the Clarity and Brevity principle. It also fails to mention a way for the user to resolve the error, thus not conforming to the Specificity principle. This error message is also completely the same as the one that was displayed when the user tried to create a network architecture with fewer than two layers. The result of this is that the error does not conform to the Context Insensitivity principle. Since the error is caught in the invocation of `fit` it conforms to the Locality principle. It does however not conform to the Proper Phrasing, and Visual design principles due to its extremely long, technical, and cluttered nature. Finally, it is consistent.

The way this same error can be reproduced in Brackeras can be seen in Listing 50. This listing also shows the error message that is displayed in the user's terminal.

```
; Minimal example code causing the error
(setv my-network
    (network
        (input (, 128 28 1))
        (conv2d 32 [3, 3])
        (maxpool2d [2, 2])))

(setv my-model (model my-network "adam" "categorical_crossentropy" ["accuracy"]))

(setv trained-model (fit my-model x-train y-train batch-size 1 0.1))

; Output in terminal
Traceback (most recent call last):
Exception: Usage error at line 10: Model input shape (128, 28, 1) and training
    data shape (28, 28, 1) are not the same but should be.
```

Listing 50: Using training data with a different shape to the `input` layer's shape in Brackeras.

In Brackeras, this error is also caught at runtime. Brackeras once again limits the length of the `Traceback` section in an effort not to overwhelm the developer with large amounts of technical details. Useful information such as the line number at which the error occurs are except offered to the programmer through the exception message. This makes is so that the Brackeras error complies to the Clarity and Brevity principle. This error also conforms to the Specificity principle since it informs the programmer of what exactly caused the problem and indirectly proposes a way for the programmer to resolve the error. This error is also context insensitive because calling `evaluate` with data that has a different shape to the model's input layer's shape causes exactly the same error. Since the error is caught at the place where it occurs, it complies with the Locality principle. Due to not exposing technical details like the Keras error, it also complies with the Proper Phrasing principle. Finally, it also clearly complies with the Consistency and Visual Design principles. Table 8 summarises both Keras' and Brackeras' error message's conformity to the design principles.

Principle	**Keras**	**Brackeras**
Clarity and Brevity	☆	★
Specificity	☆	★
Context Insensitivity	☆	★
Locality	★	★
Proper Phrasing	☆	★
Consistency	★	★
Visual Design	☆	★

Table 8: Conformity to the design principles of writing good error messages of Keras' and Brackeras' error messages concerning using training data that has a wrong shape.

Now that Brackeras' functionality and implementation have been discussed in detail and Keras and Brackeras have been thoroughly compared, the next section will be the conclusion of this bachelor's thesis.

6 Conclusion

This bachelor's thesis started with the identification of two problems with the way neural networks are currently programmed:

1. The *visual discrepancy problem*: The lack of parallels between the graph based visual representation of a neural network, and the way it is defined.

2. The *lazy error problem*: Errors are only caught at runtime and have uninformative error messages.

To explore possible solutions to these two problems, the Brackeras domain specific language was introduced. Brackeras had as a goal to solve these two problems by taking inspiration from the Reactive programming paradigm and by introducing more informative, compile-time errors wherever possible.

Afterwards, Brackeras' functionality and implementation where discussed. Brackeras currently does not support the complete set of functionalities provided by Keras. It supports the creation of network architectures with both linear and non-linear topologies. These network architectures can be transformed into models which can be trained and evaluated. Adding support for more of Keras' layers is easy and only asks for some extra programming work. Furthermore, Brackeras' implementation aims to be as extensible and efficient as possible.

To evaluate to what degree Brackeras solves the two identified problems, Keras and Brackeras were directly compared with regards to these two problems in Section 5. Brackeras' syntax is more simple and intuitive than Keras' which should allow the programmer to more easily see the parallels between the code that defines the neural network and the graph based visual representation of the network. In contrary to Keras, Brackeras internally represents neural networks as directed acyclic dataflow graphs. This makes it so that the user's mental model exactly matches the program's internal data model, really driving home the similarities between code and visual representation in the process.

Regarding the *lazy error problem*, two separate kinds of errors were identified: Structural errors and Usage errors. In Brackeras, these Structural errors are caught at compile-time, which represents a huge improvement over Keras' runtime errors. Furthermore, a qualitative analysis of both Keras' and Brackeras' error messages was conducted. In this analysis, Brackeras showed big improvements in relation to the Keras errors across the board.

With that, it can be concluded that Brackeras succeeds in what is set out to do: it drastically reduces the discrepancy between the code defining the network and its visual representation, and it poses a big improvement to Keras' errors.

A Brackeras Layers and their Arguments

Brackeras Layer	**Arguments [8]**
input	*shape*: A shape tuple (integers), not including the batch size. For instance, shape=(32,) indicates that the expected input will be batches of 32-dimensional vectors.
dense	*units*: Positive integer, dimensionality of the output space *activation*: Activation function to use. If none is specified, ReLu is used.
conv2d	*filters*: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution). *kernel-size*: An integer or tuple/list of 2 integers, specifying the height and width of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
conv3d	*filters*: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution). *kernel-size*: An integer or tuple/list of 3 integers, specifying the height and width of the 3D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
maxpool2d	*pool-size*: integer or tuple of 2 integers, window size over which to take the maximum. (2, 2) will take the max value over a 2x2 pooling window. If only one integer is specified, the same window length will be used for both dimensions.
maxpool3d	*pool-size*: Tuple of 3 integers, factors by which to downscale (dim1, dim2, dim3). (2, 2, 2) will halve the size of the 3D input in each dimension.
dropout	*rate*: Float between 0 and 1. Fraction of the input units to drop.
concat avg max min add sub mul dot	Use a sequence of names as their input.
globalavgpool3d	
batchnormalization	
flat	

B Brackeras Implementation of a Simple MNIST Convnet

```
;; -------------------------------------------------------------------------- ;;
;; Neural Network for recognising hand-written digits using the MNIST dataset ;;
;; -------------------------------------------------------------------------- ;;

; source for network architecture and data: https://keras.io/examples/vision/mnist_convnet/

(require language *)
(import language *
        numpy
        tensorflow [keras]
        keras [layers])

;;
;; Constants
;;

(setv num-classes 10)
(setv input-shape (, 28, 28, 1))
(setv batch-size 128)
(setv epochs 5)

;;
;; Preparing test data
;;

(setv [[x-train y-train] [x-test y-test]] (keras.datasets.mnist.load_data))

(setv x-train (/ (.astype x-train "float32") 255))
(setv x-test (/ (.astype x-test "float32") 255))

(setv x-train (numpy.expand_dims x-train -1))
(setv x-test (numpy.expand_dims x-test -1))

(setv y-train (keras.utils.to_categorical y-train num_classes))
(setv y-test (keras.utils.to_categorical y-test num_classes))

;;
;; Creating the network
;;

(setv mnist-network
    (network
        (input input-shape)
        (conv2d 32 [3, 3])
        (maxpool2d [2, 2])
        (conv2d 64 [3, 3])
        (maxpool2d [2, 2])
        (flat)
        (dropout 0.5)
        (dense num-classes "softmax")))

;;
;; Compiling the network into a trainable model
;;

(setv mnist-model (model mnist-network "adam" "categorical_crossentropy" ["accuracy"]))

;;
;; Training the model
;;

(setv trained-mnist (fit mnist-model x-train y-train batch-size epochs 0.1))

;;
;; Evaluating the model
;;

(evaluate trained-mnist x-test y-test)
```

References

[1] Romain B. *Pros and cons of imperative and functional programming paradigms to solve the same technical problems.* URL: https://romain-b.medium.com/pros-and-cons-of-imperative-and-functional-programming-paradigms-to-solve-the-same-technical-1511ac2f654c. (accessed: 21.07.2022).

[2] Anjnee Bhatnagar. *What is Difference Between Compile Time Errors and Runtime Errors?* URL: https://byjusexamprep.com/difference-between-compile-time-errors-and-runtime-errors-i. (accessed: 24.07.2022).

[3] Abhijeet Bhilare. *Complexity of CNN using MACC and FLOPS.* URL: https://www.kaggle.com/general/240788. (accessed: 23.07.2022).

[4] I. J. Blain. “”The Psychology of Computer Programming, ” by Gerald M. Weinberg (Book Review)”. In: *Int. J. Man Mach. Stud.* 6.2 (1974), pp. 283–284. DOI: 10.1016/S0020-7373(74)80009-X. URL: https://doi.org/10.1016/S0020-7373(74)80009-X.

[5] François Chollet. *Model training APIs.* URL: https://keras.io/api/models/model_training_apis/. (accessed: 15.07.2022).

[6] François Chollet. *Multi-GPU and distributed training.* URL: https://keras.io/guides/distributed_training/. (accessed: 23.07.2022).

[7] François Chollet. *Simple MNIST convnet.* URL: https://keras.io/examples/vision/mnist_convnet/. (accessed: 22.07.2022).

[8] François Chollet. *The base Layer class.* URL: https://keras.io/api/layers/base_layer/. (accessed: 22.07.2022).

[9] François Chollet. *The Functional API.* URL: https://keras.io/guides/functional_api/. (accessed: 09.07.2022).

[10] François Chollet. *The Sequential model.* URL: https://keras.io/guides/sequential_model/. (accessed: 14.07.2022).

[11] Joscha Drechsler et al. “Distributed REScala: An Update Algorithm for Distributed Reactive Programming”. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages Applications.* OOPSLA ’14. Portland, Oregon, USA: Association for Computing Machinery, 2014, pp. 361–376. ISBN: 9781450325851. DOI: 10.1145/2660193.2660240. URL: https://doi.org/10.1145/2660193.2660240.

[12] Vaibhav Gharge. *Imperative vs Declarative programming. Your enemy is not object-oriented programming.* URL: https://dev.to/vaibsgharge/imperative-vs-declarative-programming-your-enemy-is-not-object-oriented-programming-52fe. (accessed: 21.07.2022).

[13] *Imperative programming: Overview of the oldest programming paradigm.* URL: https://www.ionos.com/digitalguide/websites/web-development/imperative-programming/. (accessed: 21.07.2022).

[14] *Introduction to Model Parallelism.* URL: https://docs.aws.amazon.com/sagemaker/latest/dg/model-parallel-intro.html. (accessed: 23.07.2022).

[15] *Keras Multi GPU, A Practical Guide.* URL: https://www.run.ai/guides/multi-gpu/keras-multi-gpu-a-practical-guide. (accessed: 23.07.2022).

[16] Meg Long. *Best 10 Examples And Guidelines For Error Messages.* URL: https://uxwritinghub.com/error-message-examples/. (accessed: 24.07.2022).

[17] *Machine Learning Models.* URL: `https://databricks.com/glossary/machine-learning-models`. (accessed: 22.07.2022).

[18] Alessandro Margara and Guido Salvaneschi. "We Have a DREAM: Distributed Reactive Programming with Consistency Guarantees". In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems.* DEBS '14. Mumbai, India: Association for Computing Machinery, 2014, pp. 142–153. ISBN: 9781450327374. DOI: `10.1145/2611286.2611290`. URL: `https://doi.org/10.1145/2611286.2611290`.

[19] C. A. McCann. *What is a destructive update?* URL: `https://stackoverflow.com/questions/6964233/what-is-a-destructive-update`. (accessed: 21.07.2022).

[20] Caleb Meredith. *Writing Good Compiler Error Messages.* URL: `https://calebmer.com/2019/07/01/writing-good-compiler-error-messages.html`. (accessed: 24.07.2022).

[21] Mehdi Mohammadi et al. "Deep Learning for IoT Big Data and Streaming Analytics: A Survey". In: *IEEE Communications Surveys Tutorials* PP (Dec. 2017). DOI: `10.1109/COMST.2018.2844341`.

[22] Gunnar Morling. *What's in a Good Error Message?* URL: `https://www.morling.dev/blog/whats-in-a-good-error-message/`. (accessed: 24.07.2022).

[23] Jakob Nielsen. *Error Message Guidelines.* URL: `https://www.nngroup.com/articles/error-message-guidelines/`. (accessed: 24.07.2022).

[24] Bjarno Oeyen, Sam Van den Vonder, and Wolfgang De Meuter. "Trampoline variables: a general method for state accumulation in reactive programming". In: *REBLS 2021: Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, Chicago, IL, USA, 18 October 2021.* Ed. by Louis Mandel. ACM, 2021, pp. 27–40. DOI: `10.1145/3486605.3486787`. URL: `https://doi.org/10.1145/3486605.3486787`.

[25] Hein Pieterse and Helene Gelderblom. "Guidelines for error message design". In: *International Journal of Technology and Human Interaction* 14.1 (2018), pp. 80–98. DOI: `10.4018/ijthi.2018010105`.

[26] Gourav R. *Difference Between Compile Time and Run Time in C.* URL: `https://www.scaler.com/topics/c/difference-between-compile-time-and-run-time/#examples-of-compile-time-errors-and-run-time-errors`. (accessed: 24.07.2022).

[27] Guido Salvaneschi, Joscha Drechsler, and Mira Mezini. "Towards Distributed Reactive Programming". In: *Coordination Models and Languages, 15th International Conference, COORDINATION 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings.* Ed. by Rocco De Nicola and Christine Julien. Vol. 7890. Lecture Notes in Computer Science. Springer, 2013, pp. 226–235. DOI: `10.1007/978-3-642-38493-6\_16`. URL: `https://doi.org/10.1007/978-3-642-38493-6%5C_16`.

[28] Yida Tao et al. "Demystifying "bad" error messages in data science libraries". In: *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021.* Ed. by Diomidis Spinellis et al. ACM, 2021, pp. 818–829. DOI: `10.1145/3468264.3468560`. URL: `https://doi.org/10.1145/3468264.3468560`.

[29] *tf.keras.Input.* URL: `https://www.tensorflow.org/api_docs/python/tf/keras/Input`. (accessed: 22.07.2022).

[30] *tf.keras.Model*. URL: `https://www.tensorflow.org/api_docs/python/tf/keras/Model#evaluate`. (accessed: 21.07.2022).

[31] *The Hy Manual*. URL: `https://docs.hylang.org/en/stable/`. (accessed: 09.07.2022).

[32] V. Javier Traver. "On Compiler Error Messages: What They *Say* and What They *Mean*". In: *Adv. Hum. Comput. Interact.* 2010 (2010), 602570:1–602570:26. DOI: `10.1155/2010/602570`. URL: `https://doi.org/10.1155/2010/602570`.

[33] *Why Hy?* URL: `https://docs.hylang.org/en/stable/whyhy.html#hy-versus-python`. (accessed: 09.07.2022).

[34] Kai Witte. *Avoid destructive updates*. URL: `http://witte-consulting.com/blog/avoid-destructive-updates/`. (accessed: 21.07.2022).

[35] *Workflow for Neural Network Design*. URL: `https://nl.mathworks.com/help/deeplearning/ug/workflow-for-neural-network-design.html`. (accessed: 07.08.2022).