MGU - hyper-parameter self-learning using Multi Gated Units in deep neural graphs

Experimental Protocol for XCS229ii Winter 2020

Andrei Damian
Lummetry.AI
email: andrei.damian@lummetry.ai

Laurentiu Piciu
Lummetry.AI
email: laurentiu.piciu@lummetry.ai

December 6, 2020

1 The Hypothesis

1.1 Hypothesis introduction

The main proposed hypothesis behind this work is related to automatic neural graph topology architecture based on end-to-end training by directly using the task objective function. Thus we propose the *MultiGatedUnit* or shortly *MGU* - a graph module, or more intuitively said a complex neural network layer, that encapsulates any kind of neural computational layer (linear, convolutional, etc.) and it is able to self-learn the optimal sequencing configuration for the encapsulated layer through self-gating mechanisms.

Through direct optimization experiments on multiple datasets we argue that the proposed architecture drastically eliminates the need for grid search approaches required to find the optimal architecture of a deep neural graph. As mentioned, using just one end-to-end optimization process, the deep neural graph using *MGUs* will learn a *layer-level* unique individual module structure for all modules within the graph.

As an indirect result of this hypothesis we also argue that finding the *potentially ideal* architecture just by using one single end-to-end optimization process drastically reduce de carbon footprint of the whole model generation process by eliminating the need for multiple end-to-end optimizations. In terms of clear objectives for the experiments, our aim is to:

- demonstrate that our *MGU* based deep neural graph has similar or better performance with the best models resulted from a classic grid-search operation
- compare and contrast the compute time and thus carbon footprint of the whole grid-search process compared with the proposed *MGU* based deep neural graph optimization

1.2 Short related work

In terms of potentially similar work our proposed *MGU* is directly related to the general area of neural graph topology architecturing, a long researched subject and still a hot one. The area of neural graph topology architecture has a multitude of different directions and techniques such as evolutionary methods, reinforcement learning based network architecture search.

The direction of evolutionary approaches is probably the longest lasting research area in this subject - from early work of Miller et al in 1989 [1], Stanley et al in 2002 [2] up to recent work such as that of Real et al from Google Brain [3] and their successful architecture *AmoebaNet* .

Related to reinforcement learning based NAS we will mention probably one of the most important work in this area by Zoph et al [4] a research that most likely fuelled multiple teams to further deepen this direction.

Worth mentioning is the fact that both above mentioned directions are *intelligent* approaches of reducing the search-space *crawling* however both are nonetheless based on actual grid-searching.

Finally, our most closely related researches are those that focus on direct optimization of topologies based on the task objective function. Thus, our proposed research relates directly to the early and important work of Hochreiter and Schmidhuber [5], the more recent work of Schmidhuber et al in the area of self-gating mechanisms [6] and even more recent work of Hu et al [7] in the same direction.

2 The Data

2.1 Overview

For the testing of our hypothesis we propose 3 different public datasets ranging from the simple MNIST then growing the complexity to CIFAR and CERVIGRAM datasets. CERVIGRAM [8] - a little known yet public dataset provided by NIH - is a collection of data for cervical cancer classification based on colposcopy images.

2.2 The warm-up

In order to quickly iterate our experimentation protocol we decided to start with MNIST due to its simplicity and thus the correlated simplicity of the potential deep neural network architectures.

2.3 CERVIGRAM

This particular dataset, made public through the work of Xu et al [8], consists in a few hundreds of medical cases where the patients have been either suspect or have been diagnosed with various stages of cervical dysplasia. Although limited in the number of cases, the dataset is thorough in terms of cervical dysplasia medical diagnosis protocol such as: - each case is defined by at least 5 different colposcopies taken at different times - beside the acetic-acid colposcopies and their corresponding images a 6th image is taken using green lens as well as a 7th image is added to the case after applying iodine solution to the target colposcopy area. - the analysis is done individually on each cervigram as well as a whole. There are 4 different classes: normal cervix (no dysplasia), very early lesion type 1 that requires only monitoring, more advanced lesions type 2 and type 3 that require treatment and the final class of cervical cancer.

While the dataset aims to analyze each case individually and for each case there are 7 different colposcopy images (or cervigrams) as mentioned above, we decided to us only 5 images out of each case - namely the images that only have acetic acid applied on target cervix area during the colposcopy process.

3 Metrics

For our experimentation we decided to augment the classic hypothesis testing approaches based on often-used metrics such as accuracy, recall, precision with two other metrics that have the main purpose of measuring and comparing the carbon footprint of the *MGU* based topology search and optimization process with the one based on grid-search.

3.1 Time-to-solution

The "time-to-solution" metric purpose is to measure the time it takes to obtain the *best-model* both with a grid-search approach as well as based on the optimization process of the *MGU* based deep neural graph. In order to have a realistic comparison of the *MGU* approach with that of a hyper-parameter space search we opted to analyze the potential search-space and limit it to the lowest number of dimensions possible. As a result, our hyper-parameters search algorithm performs a minimal number of iterations within the grid-search process. Finally we compare this total time with the time it takes to train, evaluate and *self-analyze* the *MGU* deep neural graph.

3.2 Accuracy performance

Accuracy is a metric that needs not a special introduction and in the context of the three proposed public datasets classification accuracy is the straightforward used metric.

3.3 FLOPs/energy usage evaluation

The actual energy usage evaluation is directly correlated with the time-to-solution metric.For this particular metric we decided to train all the models with a single GPU sequentially and **monitor the GPU loading for each end-to-end optimization process**. The main idea behind this approach is that the energy consumption of various architectures can drastically vary based on the numerical computation parallelization. While the *MGU* training process is done in a single end-to-end optimization procedure it is nevertheless residing on heavy parallel computation and thus loads the GPU compute engine to a higher degree than some classic architectures (such as sequential convolutional deep neural graph architectures).

4 Models and approach

In terms of the actual model we already prepared a draft version of the *MGU* module in PyTorch as well as a Tensorflow one. While both versions will be published on GitHub, the Tensorflow version already has an extended set of features including the self-diagnosis presented in the next main section.

The main building block of *MGU* is the gate-activation block that is composed of a linear transformation and a squash function - *sigmoid* - that will constrain the gate-activation block output values between 0 and 1. Intuitively, the gate-activation output acts like a *pass* or *not pass* signal for a feature processor of the *MGU*'s input . In other words, the linear transformation weights encapsulates whether a feature processor is a good one or not to be *employed* in the computational module.

The current state of our work is focused towards automatic generation of graph topologies using the deep neural network feature processors that are currently most used both in convolutional and feed-forward neural networks literature such as *Batch Normalization* [9], *Layer Normalization* [10] and residual connections [11]. Each gate-activation block inputs two feature processors and computes the *pass / no pass* signal given the *MGU*'s input. Thus, it has the purpose of choosing between one a certain feature processing method or pathways within the graph.

4.1 MGU Model formalization

In the following equations we will present formalization of the *MultiGatedUnit* information pipeline. We will denote $L_x = f_{lyr}(x)$, where f_{lyr} is the *MGU* encapsulated transformation and x is the input of the *MGU*. Also, the encapsulated activation is denoted by $A(z)$ and a linear transofrmation of the input is denoted by $t(x)$. Please note that in below equations each individual gate *replicates* the linear transformation $t(x)$ - such as $t_{bnpp}(x)$ for the first gate below - of the encapsulated layer in order to obtain a gating tensor similar to that resulted from the actual encapsulated operations.

The first gate - g_{bnpp} - is responsible for choosing whether to use pre-activation or post-activation *Batch Normalization*.

$$l_{bnpp}(x) = t_{bnpp}(x) \tag{1}$$

$$g_{bnpp}(x) = \sigma(l_{bnpp}(x)) \tag{2}$$

$$f_{bn1}(x) = \mathrm{BN}(\mathrm{A}(L_x)) \tag{3}$$

$$f_{bn2}(x) = \mathrm{A}(\mathrm{BN}(L_x)) \tag{4}$$

$$f_{bnpp}(x) = g_{bnpp}(x) * f_{bn1}(x) + (1 - g_{bnpp}(x)) * f_{bn2}(x) \tag{5}$$

The second g_{bnln} gate chooses the best pathway between *Batch Normalization* or *Layer Normalization*.

$$l_{bnln}(x) = t_{bnln}(x) \tag{6}$$

$$g_{bnln}(x) = \sigma(l_{bnln}(x)) \tag{7}$$

$$f_{ln}(x) = \mathrm{A}(\mathrm{LN}(L_x)) \tag{8}$$

$$f_{bnln}(x) = g_{bnln}(x) * f_{bnpp}(x) + (1 - g_{bnln}(x)) * f_{ln}(x) \tag{9}$$

The third gate decides whether it is better to use any kind of feature normalization or not.

$$l_{non}(x) = t_{non}(x) \tag{10}$$

$$g_{non}(x) = \sigma(l_{non}(x)) \tag{11}$$

$$f_{non}(x) = g_{non}(x) * f_{bnln}(x) + (1 - g_{non}(x)) * \mathrm{A}(L_x) \tag{12}$$

The fourth gate is able to choose whether to use or not residual connections.

$$l_{ron}(x) = t_{ron}(x) \tag{13}$$

$$g_{ron}(x) = \sigma(l_{ron}(x)) \tag{14}$$

$$f_{res}(x) = f_{non}(x) + t(x) \tag{15}$$

$$f_{ron}(x) = g_{ron}(x) * f_{non}(x) + (1 - g_{ron}(x)) * f_{res}(x) \tag{16}$$

Finally, the last gate can decide to skip all the feature processors and use the transformed input. The output of the last gate represents also the output of the *MultiGatedUnit*

$$l_{skip}(x) = t_{skip}(x) \tag{17}$$

$$g_{skip}(x) = \sigma(l_{skip}(x)) \tag{18}$$

$$o(x) = g_{skip}(x) * f_{ron}(x) + (1 - g_{skip}(x)) * t(x) \tag{19}$$

4.2 MGU model code

In the following section a short snippet from the *MGU* python package (Tensorflow version) is p-resented containing the *forward* propagation method. Below is part of the MultiGatedUnit module - for full definition will be available soon in the GitHub repository, prior to the presentation of the final paper. First we start with the GatingUnit that is basically a learnable gate layer. The name of the class has been modified to avoid confusions

```
In [1]: import matplotlib.pyplot as plt
        import numpy as np
        %matplotlib inline
        print('Numpy', np.__version__)
        import tensorflow as tf
        print("Tensorflow", tf.__version__)

        class GatingUnit_Snippet(tf.keras.layers.Layer):
          # the call, build, __init__ methods are not shown here
          def _forward(self, inputs, for_model=False):
```

```
          tf_source = inputs[0]
          tf_value1 = inputs[1]
          tf_value2 = inputs[2]

          tf_gate_x = self.gate_activ(
            self.gate_trans(
              tf_source
              ))
          tf_out = tf_gate_x * tf_value1 + (1 - tf_gate_x) * tf_value2
          return tf_out
```

```
Numpy 1.19.2
Tensorflow 2.0.0
```

And now the main class. Again the name of the class has been modified to `MultiGatedUnit_Snippet` to avoid confusions with the actual class `MultiGatedUnit` that will be available in next two weeks on GitHub.

```
In [3]: class MultiGatedUnit_Snippet(tf.keras.layers.Layer):
          # the call, build, __init__ methods are not shown here
          def _forward(self, inputs, for_model=False):
            tf_bypass = self.bypass(inputs)
            tf_x = self.layer(inputs)
            tf_x_act = self.act_pre_bn(tf_x)

            tf_x_bn_act = self.act_post_bn(self.bn_pre_act(tf_x))
            tf_x_act_bn = self.bn_pos_act(tf_x_act)
            tf_x_act_ln = self.ln_pos_act(tf_x_act)

            tf_bpre_bpos = self.g_bpre_bpos([inputs, tf_x_bn_act, tf_x_act_bn])
            tf_bn_ln = self.g_bn_ln([inputs, tf_bpre_bpos, tf_x_act_ln])
            tf_norm_non = self.g_norm_non([inputs, tf_bn_ln, tf_x_act])
            tf_processed = tf_norm_non

            tf_processed_res = tf_processed + tf_bypass
            tf_final_processed = self.g_residual([inputs, tf_processed, tf_processed_res])

            tf_proc_noproc = self.g_proc_skip([inputs, tf_bypass, tf_final_processed])
            return tf_proc_noproc
```

5 Experimental pipeline and reasoning

In order to demonstrate the neural graph architecture self-learning capabilities of *MGU*, we defined a hyperparameter search space that is, based on our hypothesis, currently covered by our neural graph.

As described by the equations and the code in the above code snippet, our hypothesis is that the *MGU* can automatically choose for each encapsulated computational layer which kind of normalization to do - pre-activation *Batch Normalization*, post-activation *Batch Normalization*, *Layer Normalization* or no normalization. The *MGU* can automatically choose for each computational layer whether it's better ot not to use residual connections and finally can *decide* if the whole sub-graph should be skipped.

In a basic setup without *MGU* there would be 8 grid search resulted models (4 normalizations x 2 residual types) that need to be trained. However, finding the best deep neural model for a particular task requires grid searching through the number of computational elements that are stacked. Let's suppose that the grid search contains L possibilities for the number of computational elements. Thus, the grid search space grows to $8 * L$ model architectures possibilities. But, as already specified, the *MGU* possesses also a *skip* mechanism that allows the information to flow through the computational element unprocessed. In other words, we can stack a maximum number of *MGU* s - i.e. the maximum number of the L possibilities - and the neural network will self-learn how many of them should be used.

Therefore, *MGU discards* $8 * L$ grid search resulted models and the search space that the *MGU* currently covers can be defined as following:

```
In [4]: dct_mgu_search_space = {
          'norm': ['bn_pre', 'bn_post', 'ln', 'no_norm'],
          'residual': [True, False],
          'nr_computational_elements' : [3, 7, 10, 20]
        }
```

Another aspect that should be mentioned is that in a basic grid search approach, for each grid search combination the resulting computational element is copied a certain number of times. This architectural constraint implies a clear limitation to the class of hypotheses. Considering the possibility that the optimal hypothesis might need unique structures for each individual module throughout the whole deep neural graph, this ideal hypothesis will never be found. ***MGU* efficiently addresses this limitation because each individual *MGU* computational element has its unique topology**.

It's worth mentioning again that the gating mechanism uses the *sigmoid* function. Thus, each gate does not impose a hard *pass* or *no pass*. This aspect implies that each computational element can use fractions of information from all feature processors, which is (almost) impossible in a standard grid search process.

With our proposed approach we do not aim to self-learn and bypass the need for identifying hyper-parameters such as learning rates, weight initialization strategies, the number of units for a *Dense* layer, or the number of filters for a *Convolutional* layer etc.

5.1 The grid search

The hypothesis was tested for each individual dataset, using convolutional neural networks. A single *MGU* versus other 24 models (4 normalizations x 2 residual types x 3 possibilities for the layout of the neural network) were benchmarked. We did not do an extensive grid search because we concentrated our experimentation to benchmark a single *MGU* with the models resulted from the search space that the *MGU* covers.

The parameters used in the experimentation, but not grid searched, are:

```
- activation : 'selu'
- pre_redout : 'avg_pool'
- end_dropout : 0.3
- kernel_size : 3
- start_filters
   - MNIST: 16
   - CIFAR: 32
   - CERVIGRAM: 32
- max_filters
   - MNIST: 512
   - CIFAR: 1024
   - CERVIGRAM: 512
```

In order to get a valid shape of the feature map resulted after the convolution before the average pooling, the number of computational elements should come together with: `max_strides` (the maximum nr of strides applied to a convolution), `nr_reductions` (the count of the first convolutional layers on which are applied the maximum nr of strides), `scale_filters` (the factor applied to the previous convolutional layer's number of filters), `min_layer_padding_same` (which is the first convolutional layer where the padding changes from `valid` to `same`). All these parameters are denoted as `layout` in the presented grid search. We chose 3 different layouts for each individual dataset. The layout is presented as a list of items which can be decoded as following: `nr_layers, nr_reductions, max_strides, min_layer_padding_same, scale_filters = layout`.

The resulted grid search dictionaries for each dataset are:

```
In [5]: grid_search_mnist_basic = {
         'layout': [[2, 2, 4, np.inf, 2], [3, 3, 2, np.inf, 2], [4, 2, 2, np.inf, 2]],
         'norm': ['bn_pre', 'bn_post', 'ln', 'no_norm'],
         'residual': [True, False],
         'activation': ['selu'], 'pre_readout': ['avg_pool'], 'end_dropout': [0.3],
        'kernel_size': [3],
         'start_filters': [16], 'max_filters': [512]
        }

        grid_search_mnist_mgu = {
         'layout': [[4, 2, 2, np.inf, 2]],
         'activation': ['selu'], 'pre_readout': ['avg_pool'], 'end_dropout': [0.3],
        'kernel_size': [3],
         'start_filters': [16], 'max_filters': [512]
        }

        grid_search_cifar_basic = {
         'layout': [[15, 1, 2, 8, 1.32], [7, 2, 2, 5, 1.32], [10, 1, 2, 8, 1.32]],
         'norm': ['bn_pre', 'bn_post', 'ln', 'no_norm'],
         'residual': [True, False],
         'activation': ['selu'], 'pre_readout': ['avg_pool'], 'end_dropout': [0.3],
        'kernel_size': [3],
         'start_filters': [32], 'max_filters': [1024]
        }

        grid_search_cifar_mgu = {
         'layout': [[15, 1, 2, 8, 1.32]],
         'activation': ['selu'], 'pre_readout': ['avg_pool'], 'end_dropout': [0.3],
        'kernel_size': [3],
         'start_filters': [32], 'max_filters': [1024]
        }

        grid_search_cervigram_basic = {
         'layout': [[8, 6, 2, np.inf, 1.32], [11, 5, 2, np.inf, 1.24], [17, 4, 2, np.inf,
        1.14]],
         'norm': ['bn_pre', 'bn_post', 'ln', 'no_norm'],
         'residual': [True, False],
         'activation': ['selu'], 'pre_readout': ['avg_pool'], 'end_dropout': [0.3],
        'kernel_size': [3],
         'start_filters': [32], 'max_filters': [512]
        }

        grid_search_cervigram_mgu = {
         'layout': [[17, 4, 2, np.inf, 1.14]],
         'activation': ['selu'], 'pre_readout': ['avg_pool'], 'end_dropout': [0.3],
        'kernel_size': [3],
         'start_filters': [32], 'max_filters': [512]
        }

        def nr_combinations(dct_grid):
            n = 1
            for k,v in dct_grid.items():
```

```
        n*=len(v)
    return n

print("Grid search nr options:")
print("  - MNIST - basic: {} / MGU: {}".format(nr_combinations(grid_search_mnist_basic),
nr_combinations(grid_search_mnist_mgu)))
print("  - CIFAR - basic: {} / MGU: {}".format(nr_combinations(grid_search_cifar_basic),
nr_combinations(grid_search_cifar_mgu)))
print("  - CERVI - basic: {} / MGU:
{}".format(nr_combinations(grid_search_cervigram_basic),
nr_combinations(grid_search_cervigram_mgu)))
```

```
Grid search nr options:
  - MNIST - basic: 24 / MGU: 1
  - CIFAR - basic: 24 / MGU: 1
  - CERVI - basic: 24 / MGU: 1
```

The code snippet that interprets each resulted architecture can be inspected below:

```
In [1]: def create_model(mgu, shape, nr_classes, **kwargs):
          tf.keras.backend.clear_session()
          layout = kwargs.get('layout')
          model_name = kwargs.get('model_name', '')
          norm = kwargs.get('norm')
          residual = kwargs.get('residual')
          activation = kwargs.get('activation')
          pre_readout = kwargs.get('pre_readout')
          end_dropout = kwargs.get('end_dropout')
          start_filters = kwargs.get('start_filters')
          max_filters = kwargs.get('max_filters')
          kernel_size = kwargs.get('kernel_size')

          nr_layers, nr_reductions, max_strides,\
            min_layer_padding_same, scale_filters = layout

          if mgu and 'MGU' not in model_name:
            model_name = model_name + '_MGU'

          conv_func = tf.keras.layers.Conv2D

          tf_inp = tf.keras.layers.Input(shape)
          tf_x = tf_inp

          conv_layers = []
          conv_layers_kwargs = []

          for i in range(nr_layers):
            filters = min(max_filters, int(start_filters * (scale_filters**i)))
            strides = max_strides if i < nr_reductions else 1
            padding = 'valid'
            if i >= min_layer_padding_same-1:
              padding = 'same'

            conv_kwargs = dict(
              filters = filters, kernel_size = kernel_size,
              strides = strides, padding = padding
            )
            conv_layers.append(conv_func(**conv_kwargs))
            conv_layers_kwargs.append(conv_kwargs)
          #endfor

          tf_residuals = [tf_x]
          for i,layer in enumerate(conv_layers):

            if mgu:
              tf_x = MultiGatedUnit(layer, activation=activation)(tf_x)
```

```
      continue

    tf_x = layer(tf_x)
    conv_kwargs = conv_layers_kwargs[i]

    if norm == 'bn_pre':
      tf_x = tf.keras.layers.BatchNormalization()(tf_x)
      tf_x = tf.keras.layers.Activation(activation)(tf_x)

    if norm == 'bn_post':
      tf_x = tf.keras.layers.Activation(activation)(tf_x)
      tf_x = tf.keras.layers.BatchNormalization()(tf_x)

    if norm == 'ln':
      tf_x = tf.keras.layers.Activation(activation)(tf_x)
      tf_x = tf.keras.layers.LayerNormalization()(tf_x)

    if norm == 'no_norm':
      tf_x = tf.keras.layers.Activation(activation)(tf_x)

    if residual and i >= 1:
      lyr_transform = conv_func(**conv_kwargs)
      tf_x = tf.keras.layers.add([tf_x, lyr_transform(tf_residuals[-1])])

    tf_residuals.append(tf_x)
  #endfor

  if pre_readout == 'avg_pool':
    tf_x = tf.keras.layers.GlobalAveragePooling2D()(tf_x)
  else:
    raise ValueError("Not implemented")

  tf_x = tf.keras.layers.Dropout(end_dropout)(tf_x)
  tf_out = tf.keras.layers.Dense(nr_classes, activation='softmax')(tf_x)

  model = tf.keras.models.Model(tf_inp, tf_out, name=model_name)

  model.compile(
    loss='sparse_categorical_crossentropy',
    optimizer='nadam',
    metrics=['acc'],
  )

  print(model.summary())

  return model
```

5.2 Optimization approach

Each model was trained on GPU on a NVidia GeForce RTX 2080 Ti, using the same optimization approach - 100 epochs, 64 batch size for MNIST and CIFAR and 8 batch size for CERVIGRAM, *Nadam* optimizer with an initial 0.01 learning rate that is time based decayed and early stopping on validation accuracy with patience of 10 epochs.

6 Progress so far

At the particular moment of research and experimentation process we managed to obtain results from the MNIST and CIFAR experiments, while implementing self-analysis features within the *MGU* unit.

6.1 MNIST results and metrics analysis

The results of the grid-searching experiment on MNIST can be inspected in the below table. The results are promising, as the *MGU* based model surpasses all basic models in terms of validation accuracy - 99.32% for the *MGU* vs 99.14% for the best non *MGU*. Moreover, the time to solution (`tts` column) is 876 seconds for the *MGU* based model, whereas the basic grid search runs in approximately 8611 seconds - 9.84x speedup. This result drastically improves the energy consumption and the carbon footprint.

*The model size column - `model_sz (K)` - is represented in thousands of parameters. Please note that the MGU model size is always **expanded** from the size of the original neural graph due to the addition of all the gate transformation matrices.*

	model_name	layout	norm	residual	model_sz (K)	tts	train_acc	val_acc
0	MNI_MGU_001	[4, 2, 2, inf, 2]	N/A	N/A	586.6	876	99.93	99.32
1	MNI_001	[4, 2, 2, inf, 2]	ln	0	98.92	407	99.95	99.14
2	MNI_007	[4, 2, 2, inf, 2]	bn_pre	0	99.4	248	99.76	99.13
3	MNI_003	[4, 2, 2, inf, 2]	bn_post	0	99.4	197	99.79	99.12
4	MNI_004	[4, 2, 2, inf, 2]	bn_post	1	196.39	363	99.73	99.07
5	MNI_005	[4, 2, 2, inf, 2]	no_norm	0	98.44	154	99.57	98.97
6	MNI_015	[3, 3, 2, inf, 2]	bn_pre	0	24.39	404	99.7	98.97
7	MNI_008	[4, 2, 2, inf, 2]	bn_pre	1	196.39	360	99.43	98.92
8	MNI_002	[4, 2, 2, inf, 2]	ln	1	195.91	323	99.59	98.82
9	MNI_006	[4, 2, 2, inf, 2]	no_norm	1	195.43	298	99.59	98.82
10	MNI_011	[3, 3, 2, inf, 2]	bn_post	0	24.39	314	99.71	98.79
11	MNI_013	[3, 3, 2, inf, 2]	no_norm	0	23.95	332	99.73	98.77
12	MNI_009	[3, 3, 2, inf, 2]	ln	0	24.17	433	99.87	98.77
13	MNI_012	[3, 3, 2, inf, 2]	bn_post	1	47.53	325	99.63	98.68
14	MNI_014	[3, 3, 2, inf, 2]	no_norm	1	47.08	305	99.34	98.52
15	MNI_010	[3, 3, 2, inf, 2]	ln	1	47.31	300	99.4	98.45
16	MNI_016	[3, 3, 2, inf, 2]	bn_pre	1	47.53	383	99.46	98.44
17	MNI_024	[2, 2, 4, inf, 2]	bn_pre	1	9.96	494	93.61	93.12
18	MNI_020	[2, 2, 4, inf, 2]	bn_post	1	9.96	482	93.62	93.1
19	MNI_018	[2, 2, 4, inf, 2]	ln	1	9.87	597	93.32	93.06
20	MNI_022	[2, 2, 4, inf, 2]	no_norm	1	9.77	338	93.18	92.99
21	MNI_019	[2, 2, 4, inf, 2]	bn_post	0	5.32	349	92.88	92.73
22	MNI_021	[2, 2, 4, inf, 2]	no_norm	0	5.13	437	92.61	92.61
23	MNI_023	[2, 2, 4, inf, 2]	bn_pre	0	5.32	339	92.24	92.05
24	MNI_017	[2, 2, 4, inf, 2]	ln	0	5.23	430	91.76	91.61

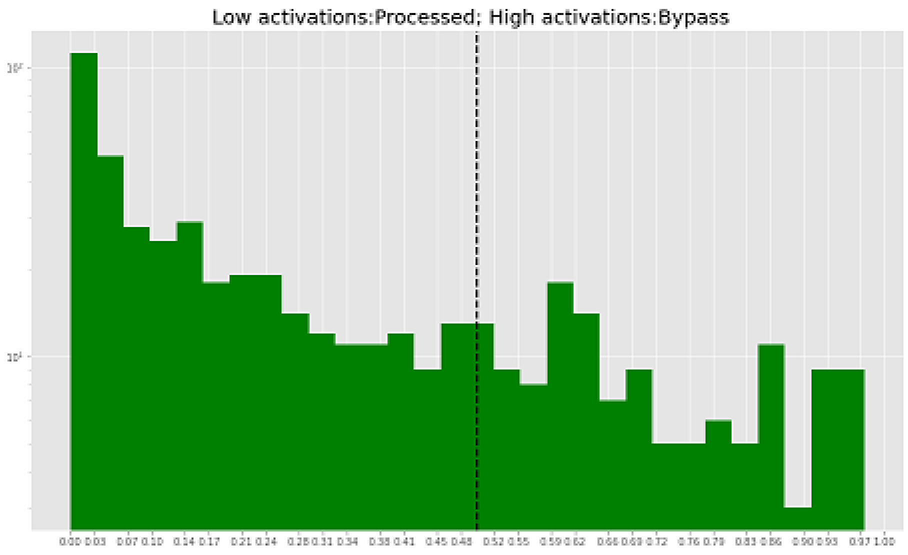
6.2 MGU self-diagnosis output

One of the most important features that we managed to add at this time in the *MGU* is a simple **self-explain-ability** method that enables the analysis of the gates within a *MGU* module and presents the most-likely feature data-flow. For this particular feature we perform forward-passes of data throughout our model

```
Self-explainability of MGU layers in 6 layer graph
```

```
...
  Layer: MGU_conv2d_1 self-explainability analysis based on (12, 12, 16) input
...
    Analysing 'Gate3 - any norm or no norming at all'
      Gate rule: `gate * Norming  +  (1 - gate) * No-norming`
      Gate mean: 0.75, median: 0.87, min/max: 0.01/1.00
      Gate opened for: Norming
...
    Analysing 'Gate5 - direct linear bypass or processed'
      Gate rule: `gate * Bypass  +  (1 - gate) * Processed`
      Gate mean: 0.32, median: 0.20, min/max: 0.00/0.99
      Gate opened for: Processed
```

From the above partial explain-ability log output we see that for each individual gate the *MGU* gives a *hint* regarding the most likely path that the data flows throughout the graph. For example for the second *MGU* `MGU_conv2d_1` the 5th gate blocks non-processed information from the bypass transformation while allowing the processed information to be passed forward - the actual activation distribution of the gate unit can be observed in below picture.



6.3 CIFAR-10 results and metrics analysis

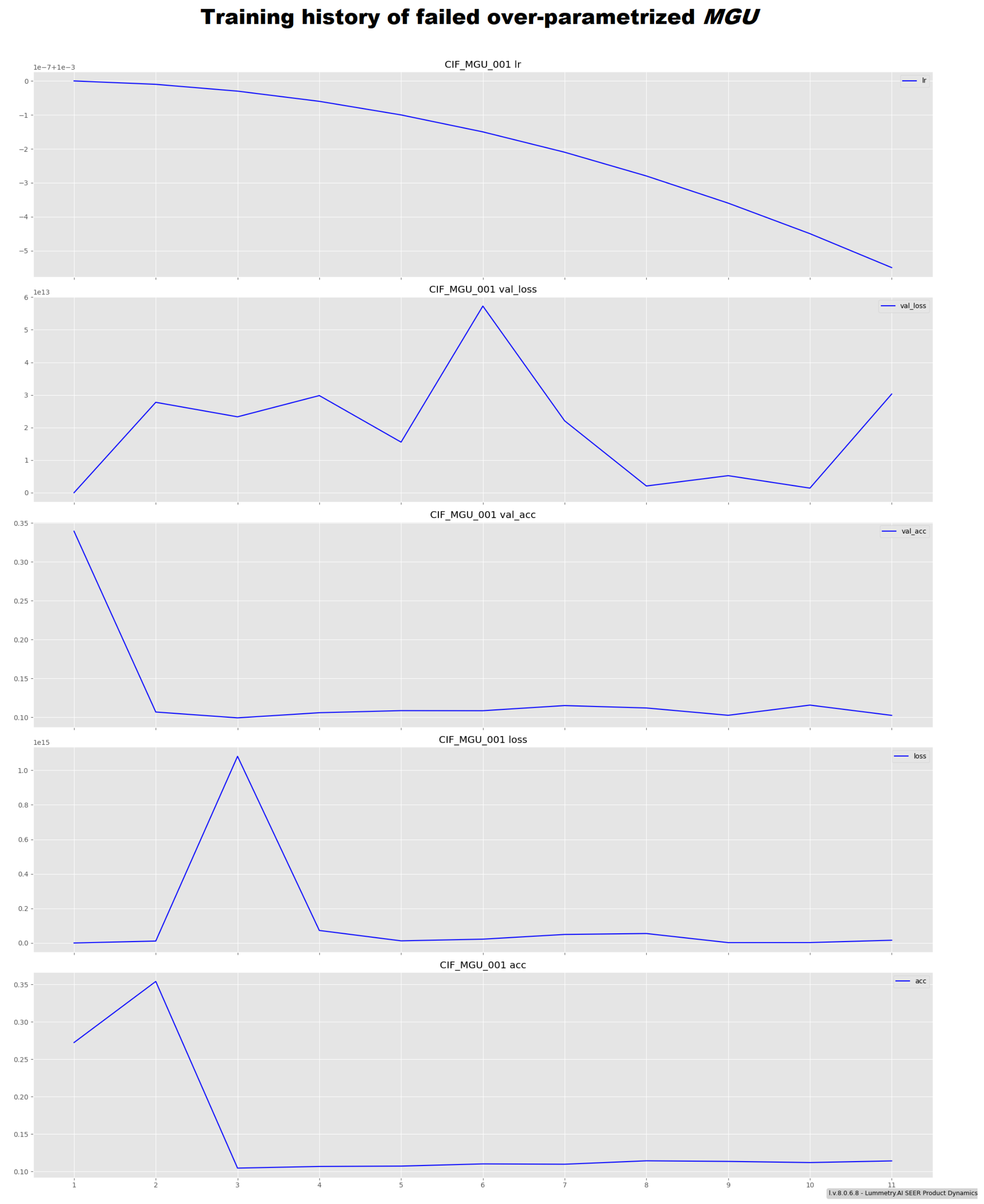
The results of the grid-searching experiment on CIFAR can be inspected in the below table. The results are also promising, as the *MGU* based model achieves comparable validation accuracy with the best non *MGU* model. The running time (`tts` column) for the *MGU* based model is 1089 seconds, whereas the basic grid search runs in approximately 13450 seconds - 12.35x speedup. This result drastically improves the energy consumption and the carbon footprint.

The model size column - `model_sz (M)` - is represented in millions of parameters.

	model_name	layout	norm	residual	model_sz (M)	tts	train_acc	val_acc
0	CIF_016	[10, 1, 2, 8, 1.32]	bn_pre	0	2.41	494	88.72	74.38
1	CIF_008	[15, 1, 2, 8, 1.32]	bn_pre	0	30.51	1456	93.06	74.13

	model_name	layout	norm	residual	model_sz (M)	tts	train_acc	val_acc
2	CIF_MGU_002	[7, 2, 2, 5, 1.32]	N/A	N/A	3.09	1089	95.02	73.96
3	CIF_006	[15, 1, 2, 8, 1.32]	bn_post	0	30.51	910	92.71	73.83
4	CIF_014	[10, 1, 2, 8, 1.32]	bn_post	0	2.41	347	92.2	73.2
5	CIF_024	[7, 2, 2, 5, 1.32]	bn_pre	0	0.44	386	87.55	71.18
6	CIF_022	[7, 2, 2, 5, 1.32]	bn_post	0	0.44	244	86.52	70.24
7	CIF_011	[10, 1, 2, 8, 1.32]	ln	1	4.81	689	82.37	68.96
8	CIF_021	[7, 2, 2, 5, 1.32]	bn_post	1	0.88	319	81.82	68.74
9	CIF_020	[7, 2, 2, 5, 1.32]	ln	0	0.44	251	84.79	68.68
10	CIF_018	[7, 2, 2, 5, 1.32]	no_norm	0	0.44	235	86.51	67.78
11	CIF_019	[7, 2, 2, 5, 1.32]	ln	1	0.88	548	91.74	67.34
12	CIF_013	[10, 1, 2, 8, 1.32]	bn_post	1	4.81	699	85.83	66.92
13	CIF_015	[10, 1, 2, 8, 1.32]	bn_pre	1	4.81	715	75.75	66.08
14	CIF_023	[7, 2, 2, 5, 1.32]	bn_pre	1	0.88	344	73	65.29
15	CIF_010	[10, 1, 2, 8, 1.32]	no_norm	0	2.4	258	83.32	61.09
16	CIF_017	[7, 2, 2, 5, 1.32]	no_norm	1	0.88	270	49.24	47.91
17	CIF_009	[10, 1, 2, 8, 1.32]	no_norm	1	4.8	213	47.26	44.85
18	CIF_007	[15, 1, 2, 8, 1.32]	bn_pre	1	60.99	1579	46.4	43.47
19	CIF_MGU_001	[15, 1, 2, 8, 1.32]	N/A	N/A	182.93	2115	34.98	33.95
20	CIF_005	[15, 1, 2, 8, 1.32]	bn_post	1	60.99	777	26.25	25.82
21	CIF_003	[15, 1, 2, 8, 1.32]	ln	1	60.98	733	20.14	20.2
22	CIF_001	[15, 1, 2, 8, 1.32]	no_norm	1	60.96	1058	17.67	17.71
23	CIF_002	[15, 1, 2, 8, 1.32]	no_norm	0	30.49	366	10	10
24	CIF_012	[10, 1, 2, 8, 1.32]	ln	0	2.41	187	10	10
25	CIF_004	[15, 1, 2, 8, 1.32]	ln	0	30.5	417	10	10

The first *MGU* based model - *CIF_MGU_001* that used the 15 stacked computational modules which is the maximum number of computational modules in the chosen CIFAR-10 grid search. As a result, this model is over-parametrized and thus, its optimization diverges after the first epoch. The over-parametrization comes from the added gate transformations for *Convolutional* layers. We train another *MGU* based model that uses less computational elements - *CIF_MGU_002* - and it behaves as expected.



7 The next steps

So far we managed to obtain a stable version of the *MGU* module including the self-explanation feature for Tensorflow. We also prepared a lighter version of *MGU* for PyTorch framework. We also experimented and evaluated the behavior of the MGU on two initial public datasets and clearly identified the need to have more efficient gating transformations. Beside the public dataset experimental work we already employed the *MGU* on private production-grade systems.

7.1 CERVIGRAM experiment

More work is required to obtain consistent evidence regarding the application of our proposed innovation. Currently experimentation is underway with CERVIGRAM dataset.

7.2 Predictive analytics experiment

Moreover, we will prepare for the final paper experiments proof that *MGU* can be applied to other *layer* types such as *linear* layers in predictive analytics experiments. The application of *MGU* on private datasets already has yielded good results and we are preparing the report on those experiments as well.

7.3 Over-parametrization

Moreover, we will address the over parametrization problem efficiently either with heuristical approaches such as using depth-wise convolutions transformations for convolutional gates or with low-dimensionality embedding generation similar to [7]. Last but not least we will explore further feature processors possibilities to include in the *MultiGatedUnit*.

```
In [ ]:
```

References

[1] Geoffrey F Miller, Peter M Todd, and Shailesh U Hegde. Designing neural networks using genetic algorithms. In *ICGA*, volume 89, pages 379–384, 1989.

[2] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[3] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

[4] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.

[5] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[6] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks, 2015. arXiv:1505.00387.

[7] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, and Enhua Wu. Squeeze-and-excitation networks, 2019.

[8] Tao Xu, Cheng Xin, L Rodney Long, Sameer Antani, Zhiyun Xue, Edward Kim, and Xiaolei Huang. A new image data set and benchmark for cervical dysplasia classification evaluation. In *International Workshop on Machine Learning in Medical Imaging*, pages 26–35. Springer, 2015.

[9] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *32nd International Conference on Machine Learning, ICML 2015*, 2015.

[10] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016.