LambdaNetworks: Efficient & accurate, but also accessible? A reproducibility project with CIFAR-10

de Alvear Cárdenas, J.I.*

4463196
Department of Control & Simulation,
Faculty of Aerospace Engineering,
Delft University of Technology, Delft

de Vries, W.A.J.G.†

4296869
Department of Space Systems Engineering,
Faculty of Aerospace Engineering,
Delft University of Technology, Delft

Dated: April 17, 2021

Abstract

LambdaNetworks is a novel machine learning architecture by Irwan Bello that promises superior performance with faster computation and lower memory footprint compared to attention and convolutions. However, the original paper validates its performance in a ResNet-50 architecture on ImageNet running on 32 TPUv3s. We present the implementation and the results of our reproduction with the smaller and lower dimensional CIFAR-10 dataset. With respect to the vanilla ResNet-50 baseline, the throughput with lambda layers is increased by 31% and training time decreased by 18.5%. However, accuracy suffers from a 3.2% reduction. Our implementation can be found at:
`https://github.com/joigalcar3/LambdaNetworks`

1 The author of the original LambdaNetworks article: Irwan Bello

Irwan Bello, the author of LambdaNetworks: Modeling long-range interactions without attention is a machine learning and artificial intelligence expert who

*Electronic address: `j.i.dealvearcardenas@student.tudelft.nl`; Corresponding author
†Electronic address: `w.devries-1@student.tudelft.nl`

focuses on self-attention and closely related topics. He has been working at Google Brain as a research scientist since 2016. Recently, Bello has published two articles, the article about LambdaNetworks and a very closely related paper concerning "Revisiting ResNets: Improved Training and Scaling Strategies".

2 Introduction

This article builds on the foundations laid by Irwan Bello in "LambdaNetworks: Modeling long-range interactions without attention" (Bello 2021). Bello proposes a method where long-range interactions are modelled by layers which transform contexts into linear functions called lambdas, in order to avoid the use of attention maps. The great advantage of lambda layers is that they require much less compute than self-attention mechanisms according to the original paper by Bello. This is fantastic, because it does not only provide results faster, but also saves money and has a more favorable carbon footprint! However, Bello still uses 32 TPUv3s [1] and the 200 GB sized ImageNet classification dataset. Therefore, we started this reproducibility project wondering: Could lambda layers be scaled to mainstream computers while keeping its attractive properties?

In 2021 the world did not only have to deal with the COVID-19 epidemic [2] but was struck by chip shortages [3] as well due to increase in consumer electronics for working at home, shut down factories in China and the rising prices of crypto-currencies. This has decreased supply to record lows and prices to record highs. Resulting in a situation, whereby researchers, academics, and students (who are all usually on a budget) are no longer able to quickly build a cluster out of COTS (commercial off-the-shelf) GPUs resulting in having to deal with older, less, and less efficient hardware.

No official code was released at the time of starting the project mid-March. Therefore, in order to answer the aforementioned question, it is up to us to reproduce the paper by Bello as accurately as possible while trying to scale it down such that it can be run on an average consumer computer. The produced code is made publicly available

3 Existing LambdaNetworks paper reviews and reproducibility projects

The paper by Bello has been published on February 2, 2021 and has been cited 7 times at the time of writing this article. This means that the article is brand new and therefore has not been combed through yet by the academic community.

However, some researchers and members of the machine learning community have already read the article and provided a review of the paper - these will be

[1] https://cloud.google.com/tpu

[2] https://www.who.int/emergencies/diseases/novel-coronavirus-2019

[3] https://www.cnbc.com/2021/02/10/whats-causing-the-chip-shortage-affecting-ps5-cars-and-more.html

mentioned hereafter. Yannic Kilcher [4] has published "LambdaNetworks: Modeling long-range interactions without Attention (Paper Explained)" on YouTube 4 months prior to the publication of the article by Bello. Kilcher goes over the preliminary version of the paper and explains them to listeners and provides recommendations to the author. Carlos Ledezma [5] published a video along the lines of Kilcher but takes more time to extensively clarify the distinction between the structure of an attention layer and a lambda layer.

Furthermore, Phil Wang [6] has published unofficial code using Pytorch about the lambda layer. Additionally, Myeongjun Kim [7] has not only reproduced the lambda layer code, but this member of the community has also applied it to different ResNet versions and different datasets, as well as performed an ablation study. The code from the aforementioned data scientists is not used for generating our code. However, we will briefly compare our code to theirs in order to see why they made specific choices for their implementations and justify ours.

Luckily, Bello has clarified that he will publish the code corresponding to the LambdaNetworks paper soon. This will most-likely enhance everyone's understanding of the LambdaNetworks.

4 Goals of this reproducibility project

Originally, the scientific goal of this paper is to reproduce two particular results from Table 3 from Bello's paper, which can be seen below in Figure 1. These results are Conv (He et al., 2016) - 25.6 - 76.9 and Lambda layer 15.0 - 78.4 (+1.5). This was set in order to find out whether we would be able to reproduce this without prior code implementations.

Layer	Params (M)	top-1
Conv (He et al., 2016)†	25.6	76.9
Conv + channel attention (Hu et al., 2018c)†	28.1	77.6 (+0.7)
Conv + linear attention (Chen et al., 2018)	33.0	77.0
Conv + linear attention (Shen et al., 2018)	-	77.3 (+1.2)
Conv + relative self-attention (Bello et al., 2019)	25.8	77.7 (+1.3)
Local relative self-attention (Ramachandran et al., 2019)	18.0	77.4 (+0.5)
Local relative self-attention (Hu et al., 2019)	23.3	77.3 (+1.0)
Local relative self-attention (Zhao et al., 2020)	20.5	78.2 (+1.3)
Lambda layer	**15.0**	**78.4 (+1.5)**
Lambda layer ($\|u\|$=4)	**16.0**	**78.9 (+2.0)**

Figure 1: Comparison of the lambda layer and attention mechanisms on ImageNet classification with a ResNet-50 architecture Bello 2021

However, as one can read in section 7, the 1-to-1 reproduction of these results

[4] https://www.youtube.com/watch?v=3qxJ2WD8p4w
[5] https://www.youtube.com/watch?v=awclKwGO_sM
[6] https://github.com/lucidrains/lambda-networks
[7] https://github.com/leaderj1001/LambdaNetworks

is not possible for the average student and researcher due to the large dataset employed by Bello in his research, namely ImageNet, and the compute resources he exploited to obtain those results in reasonable time, namely 8-128 TPUv3; resources beyond what we, as students, have at our disposal. Therefore, the scope was shifted slightly. No longer was it the goal to exactly reproduce the aforementioned results without published code, but rather to find out if these results are attainable with a much smaller dataset as well.

The personal goal of this reproducibility project is twofold. Firstly, attention seems to be state-of-the-art at the moment with a lot of ongoing research. However, attention has some shortcomings that lambda layers aims to fix while, at the same time, slightly increasing the accuracy. Therefore, reproducing the lambda layers can contribute to the early future adoption of this potentially superior algorithm by the community. Additionally, its implementation on a lower dimensional dataset proves the robustness of the algorithm, as well as its potential implementation on resource constrained devices (TinyML).

The second reason is the enhancement and broadening of the deep learning knowledge and skills of the authors of the here presented reproducibility project, which is part of the Deep Learning course at Delft University of technology taught by Dr. Jan van Gemert [8]. In order to get a feel for Deep-Learning it is not only important to read from the great number of online resources, but also to have hands-on experience with some groundbreaking papers. After having read the well-received paper called "Attention Is All You Need" (Vaswani et al. 2017), we were excited by the whole pool of papers that could follow-up. From the abstract, lambda layers promised to be a great advancement from Transformers, targeting multiple weakness and leading to greater performance and lower computational load. Eager to learn more about attention and thrilled by the potential of lambda layers, choosing this reproducibility project was an obvious choice for us.

5 Why lambda layers?

Lambda layers are closely related to self-attention mechanisms as they allow for modeling long-range interactions. However, self-attention mechanisms have a big drawback which has to do with the fact that they require attention maps for modeling the relative importance of layer activations, which require additional compute and are hungry for RAM (Random Access Memory). This makes them less applicable for use in machine vision applications which heavily rely on images (consisting of a grid of pixels), due to compute and RAM requirements for modeling long-range interactions between each of these pixels. Therefore, it is evident that this problem should be solved in order to decrease the training and inference times of any attention based vision task.

As Bello (Bello 2021) says it himself in his article: "We propose *lambda layers* which model long-range interactions between a query and a *structured* set of context elements at a reduced memory cost. Lambda layers transform each available context into a linear function, termed a lambda, which is then

[8] `https://scholar.google.com/citations?user=JUdMRGcAAAAJ`

directly applied to the corresponding query." This set of context elements is consequently summarized by the lambda layer into a linear function.

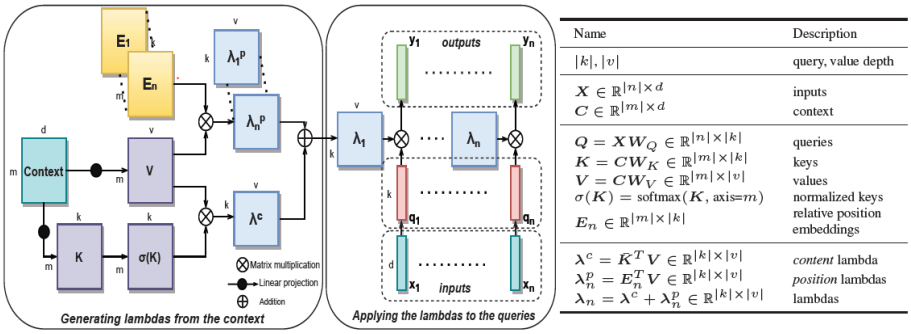
Linear attention mechanisms (Li et al. 2020) have posed a solution to the problem of high memory usage. However, these methods do not capture positional information between query and context elements (e.g. where pixels are on an image). Lambda layers, in contrast, have low memory usage and capture position information. The latter even results in increased performance, such that it outperforms convolutions with linear attention and local relative self-attention on the ImageNet dataset.

6 LambdaNetworks explained

The main advantage of the LambdaNetworks over Transformers is that content and position-based interactions between all components of the context, which can be chosen to be global or local, are computed without producing expensive and memory intensive attention maps. To achieve this, the lambda layers feature 3 main steps:

1. Computation content lambda which encapsulates the context content.
2. Computation of the position lambda which encapsulate the relative position information between the query and the elements of the context.
3. Application of the content and position lambdas to the queries for the computation of the output.

Figure 2 provides a great overview of the complete process. In the following subsections we will explain each of the steps in detail, constantly referring back to the presented lambda layer computational graph.



Name	Description
$\|k\|, \|v\|$	query, value depth
$\boldsymbol{X} \in \mathbb{R}^{\|n\| \times d}$	inputs
$\boldsymbol{C} \in \mathbb{R}^{\|m\| \times d}$	context
$\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}_Q \in \mathbb{R}^{\|n\| \times \|k\|}$	queries
$\boldsymbol{K} = \boldsymbol{C}\boldsymbol{W}_K \in \mathbb{R}^{\|m\| \times \|k\|}$	keys
$\boldsymbol{V} = \boldsymbol{C}\boldsymbol{W}_V \in \mathbb{R}^{\|m\| \times \|v\|}$	values
$\sigma(\boldsymbol{K}) = \text{softmax}(\boldsymbol{K}, \text{axis}{=}m)$	normalized keys
$\boldsymbol{E}_n \in \mathbb{R}^{\|m\| \times \|k\|}$	relative position embeddings
$\boldsymbol{\lambda}^c = \boldsymbol{K}^T\boldsymbol{V} \in \mathbb{R}^{\|k\| \times \|v\|}$	*content* lambda
$\boldsymbol{\lambda}_n^p = \boldsymbol{E}_n^T\boldsymbol{V} \in \mathbb{R}^{\|k\| \times \|v\|}$	*position* lambdas
$\boldsymbol{\lambda}_n = \boldsymbol{\lambda}^c + \boldsymbol{\lambda}_n^p \in \mathbb{R}^{\|k\| \times \|v\|}$	lambdas

Figure 2: Computational graph of the lambda layer (Bello 2021).

6.1 Content lambda

For the computation of the content lambda, the global context is used. In the case of a single square image, the global context consists of all the pixels.

Therefore, if the image shape is d×n×n, being n the number of pixels along the width and length of the image and d the image dimensions (3 in the case of a color image), then the context is of shape |n|×d, where |n|=n^2. Figure 2 can be misleading when understanding the dimensions of the context, since it shows that it is of shape |m|×d. In contrast with content lambdas, positional lambdas use local contexts and the author decided to express the dimensions as required for the computation of the latter. In reality, the content lambdas are computed first with the global context and then the positional lambdas are computed with local contexts. Unfortunately, this sequential computation of the lambdas and different matrix dimensions are not reflected clearly in the computational graph.

From the global context, the values and the keys are computed as shown in Equation 1 and Equation 2. Then the keys are normalised with softmax along the |m| dimension (Equation 3), whereas the values are batch normalised. Finally, the content lambda is obtained from the matrix multiplication of the normalised values and keys, as can be observed in Equation 4.

$$\boldsymbol{V} = \boldsymbol{C}\boldsymbol{W}_V \ \in \mathbb{R}^{|m|\times|v|} \tag{1}$$

$$\boldsymbol{K} = \boldsymbol{C}\boldsymbol{W}_K \ \in \mathbb{R}^{|m|\times|k|} \tag{2}$$

$$\bar{\boldsymbol{K}} = \text{softmax}(\boldsymbol{K}, \text{axis} = m) \ \in \mathbb{R}^{|m|\times|k|} \tag{3}$$

$$\boldsymbol{\lambda}^c = \bar{\boldsymbol{K}}^T\boldsymbol{V} \ \in \mathbb{R}^{|k|\times|v|} \tag{4}$$

The benefit of this approach is that the content lambda encodes how to transform the queries based solely on the context content, independently of the each query. As a result, the content lambda is shared among all queries/pixels of the image.

6.2 Position lambda

For the computation of the content lambda, the user can choose whether using local contexts of size |m|×d or a global context by making |m| equal to |n|. For the purpose of this reproducibility project, the global context is used for the computation of the position lambda due to the low |n| value of the inputs fed to the lambda layers, namely |n| ⊂ [8, 4, 2, 1]. This small input dimension is caused by the dataset used, namely CIFAR-10, whose images are 6 times smaller than ImageNet. With this reduced dataset, the potential acceleration gained from computing the position lambdas from smaller contexts is insignificant when compared to the extra computations required to extract the local context from the input.

As can be observed in Figure 2, the position lambdas are the result of the product between the value matrix and position embeddings (Equation 5). The latter are n learnt |m|×k matrices that encapsulate the positional relation between each of the n queries with the context. As a result, the embeddings are a |n| × |m|×k block that generates n positional lambdas.

$$\boldsymbol{\lambda}_n^p = \boldsymbol{E}_n^T \boldsymbol{V} \ \in \mathbb{R}^{|k| \times |v|} \tag{5}$$

Thanks to the separation of the content and the positional information, it is possible to share the position embeddings among lambda layers that receive the same input size, leading to a lower memory footprint. The position lambdas encode how the queries need to be transformed solely based on their relative position with respect to each of the components of its context.

6.3 Lambdas applied to queries

Once the content and position lambdas have been computed, it is possible to compute the final lambda matrices that will be applied to the queries by summing both components, as can be seen in Equation 6. Furthermore, Equation 7 shows how the queries are computed from the input.

$$\boldsymbol{\lambda}_n = \boldsymbol{\lambda}^c + \boldsymbol{\lambda}_n^p \ \in \mathbb{R}^{|k| \times |v|} \tag{6}$$

$$\boldsymbol{Q} = \boldsymbol{X} \boldsymbol{W}_Q \ \in \mathbb{R}^{|n| \times |k|} \tag{7}$$

Once the queries and their respective lambdas are computed, the lambdas transform the queries to the output by a simple matrix multiplication, as shown in Equation 8, where $\boldsymbol{q}_n$ corresponds to the queries generated from every single pixel; each row in the $\boldsymbol{Q}$ matrix.

$$\boldsymbol{y}_n = \boldsymbol{\lambda}_n^T \boldsymbol{q}_n \ \in \mathbb{R}^{|v|} \tag{8}$$

6.4 Multi-query lambda layer

In the LambdaNetworks paper (Bello 2021) it was observed that the reduction of the value dimension |v| could greatly reduce the computational cost, as well as the space and time complexities. Therefore, the author decides to decouple these complexities from this dimension by manipulating its value at will.

For that purpose, he proposes using |h| queries $\boldsymbol{q}_n^h$ for each (pixel) input to which the same lambda is applied. Then the output for a single (pixel) input is the result of the concatenation of each of the h outputs as $\boldsymbol{y}_n = \text{concat}(\boldsymbol{\lambda}_n \boldsymbol{q}_n^1, \cdots, \boldsymbol{\lambda}_n \boldsymbol{q}_n^{|h|})$. Consequently, |v| is now equal to d/|h|, which reduces the complexity by a factor of |h|. This reduction in the dimensionality of the values is called by the author multi-query lambda layers. It is important to note that there is a trade-off between the size of the lambdas $\boldsymbol{\lambda}_n \in \mathbb{R}^{|k| \times |d|/|h|}$ and the size of the queries $\boldsymbol{q}_n \in \mathbb{R}^{|hk|}$.

7 Constraints, dataset and parameters

In this section we will explain how constraints in compute lead to the usage of a different dataset and how and why some parameters were adjusted from the original paper by Bello. We would however like to stress, that most of the parameters are kept exactly the same as the ones used by Bello for a fair comparison later in this article.

7.1 Constraints

There are some computational constraints in comparison to Bello's setup. Bello, as a Google researcher, was able to use 32 TPUv3 units in his research project with the ImageNet classification dataset. Unfortunately, we do not have this compute to our disposal. TU Delft was kind enough to provide us with 50 Euros of Google Cloud credit, however implementing the changes in the code such that it could efficiently be run on Google Cloud would be too time consuming. Additionally, 50 Euros may would most-likely not have been enough to run the 200 GB sized ImageNet a few times. ImageNet consists of 130 million filtered and balanced JFT images with pseudo-labels generated by an EfficientNet-L2 with 88.4% accuracy, according to the set used by Bello. Therefore, a different dataset was chosen as explained in subsection 7.2.

The data was trained and tested on Google Colab and a high-end laptop with mediocre graphics card from 2020. The local laptop provided comparable results as check: since the availability of components is more or less fixed, while Colab assigns different configurations based on a (secret) algorithm.

The primary environment consisted of a free Google Colab environment[9]. This environment is available to almost everyone on the planet and therefore deemed accessible for use in this paper. In this environment the resources are not guaranteed as they are shared among users of the platform and therefore it is hard to say whether the obtained results will be consistent in terms of running time. This is due to the fact that not only the amount of RAM changes, but also the available type of GPU can change between the Nvidia K80, T4, P4 and P100.[10] The environment was set up by using these GPUs to perform the calculations. The algorithm however determines on a case-by-case basis how much RAM will most-likely be used and allocate more to heavy users, most-likely up to 12 GB and the maximum allowed time for a session is 12 hours. The allocated resources however, do not vary throughout a session which makes intersession results reliable to compare between each other.

The secondary training environment consisted of a laptop with an Intel i7-10750H on stock speed with adequate cooling, Nvidia Quadro T1000 Max-Q 4 GB DDR5 (stock speed) and 16 gigabytes of DDR4 ram in single-slot configuration at a stock speed of 2400 MHz. Due to the fact this runs fully independent, all results are reproducible; unlike in Google Colab (where performance is approximately twofold better.)

7.2 Dataset

After consulting with Dr. Jan van Gemert [11] and Robert-Jan Bruintjes [12] of TUDelft, it was decided to go for a smaller dataset, namely CIFAR-10 [13]. CIFAR-10 is a dataset with 60,000 32x32 colour images in 10 classes, with 6,000

[9] https://colab.research.google.com/

[10] https://research.google.com/colaboratory\/faq.html#usage-limits

[11] https://scholar.google.com/citations?user=JUdMRGcAAAAJ

[12] https://scholar.google.com/citations?user=RXVnqgcAAAAJ&hl=en

[13] https://www.cs.toronto.edu/~kriz/cifar.html

images per class, split into 5:1 train-test ratio. An example of what these images look like can be found in Figure 3 below.

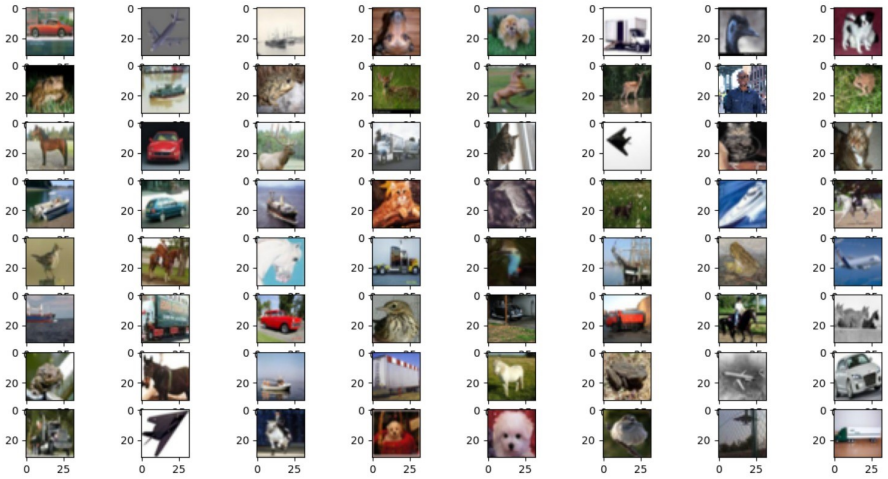


Figure 3: A sample of the CIFAR10 dataset with 32 x 32 colour images.

7.3 Parameters

In order to find out how the author of the paper has exactly compiled the results, it is important to look at the initial conditions which were used to obtain the results. Next, we compile a list of all the most important parameters used with their corresponding definitions and values. They are split up in three tables:

1. Table 1 contains the parameters that can readily be changed by the user. They all can be found in `user_input.py` within our code. These parameters include among others batch size, input size and initial learning rate.

2. Table 2 contains other parameters used within the implementation of the ResNet-50 and the lambda layer. They are defined in `lambda_layer.py`, `resnet.py` and `resnet_lambda.py`.

3. Table 3 contains more information about the implementation of the algorithms. For more information, we would like to refer the reader to the documentation in the code and later sections.

Table 1: The most-important variables as defined in `user_input.py`.

Variable	**Variable (code)**	**Value**	**Description**
B, $\|b\|$	`b_size`	128	Batch size $B = 128$, in the paper a batch size B of 4096 is distributed over 32 TPUv3 cores. As we only train on 1 GPU, this number should be divided by 32 resulting in 128.
n	`input_size`	8 x 8	The size of the input is 8 by 8 pixels.

m, $\|m\|$	`context_size`	8 x 8	The size of the global context equals 8 by 8 pixels, just like the size of the input. Bello, with ImageNet, uses local contexts of size $\|m\| = 23 \times 23$ to generate local position lambdas.
k	`qk_size`	16	Key size
q	`qk_size`	16	Query size
$\|h\|$	`heads`	4	Heads
Epochs	`epochs`	90	Epochs
Weight decay	`weight_decay`	1e-4	Weight decay for Adam
Initial learning rate	`initial_lr`	0.0005	The learning rate (α) from epoch 0 to 5, is defined by a linear relation over these epochs and determined by $\alpha_{max} = 0.1 \cdot B/256$. Even though the batch size is 128 per GPU, the batch size the author was using consisted of 4096 samples. Using the formula as provided in the paper, the maximum learning rate was determined to be 0.05 resulting in a initial learning rate of 0.01. Using an equally spaced grid, this resulted in the following learning rates: [0.01,0.02,0.03,0.04,0.05]. However, the obtained learning rate did not yield good results. Therefore, a learning rate sensitivity analysis was performed, yielding 0.0005 as optimal result, as is determined in subsection 11.3. This is used throughout this project, unless stated otherwise.
γ	`BN_gamma`	0	Gamma is the scaling parameter, an argument in the batch normalization. Only for the last batch normalization layer of the bottleneck blocks (Ioffe and Szegedy 2015).

Table 2: The most-important variables as used in `lambda_layer.py`, `resnet.py` and `resnet_lambda.py`.

Variable	**Variable (code)**	**Value**	**Description**
D	-	50	Depth of the ResNet: 50 layers (referred to as ResNet-50).
v	`v`	$v = \frac{d}{h}$	Value depth, the dimension of the value matrix which is layer dependent.
d_{in}, d_c, d_{out}	`d`	-	d_{in} = input channels, d_c = context channels, d_{out} = output channels. Its value is layer dependent since it is the number of channels of the layer inputs.
$\|k\|$	`k`	16	Key depth, query depth
E_n	`embedding`	-	Relative position embeddings initialised with $\mathcal{N}(0, 1)$.
K	`keys`	-	Keys
V	`values`	-	Values
Q	`queries`	-	Queries, The queries **Q** are normalized around the query depth k. Then `BatchNorm2D` is applied to the queries along context **C** (2^{nd} dimension), resulting in a shift of k to the second dimension, then batch normalization is applied and it is transposed back to the original shape.
W_K	tokeys	-	Projection to compute **K** initialised with $\mathcal{N}(0, \|d\|^{-\frac{1}{2}})$.

W_V	tovalues	-	Projection to compute **V** initialised with $\mathcal{N}(0, \lvert d \rvert^{-\frac{1}{2}})$.
W_Q	toqueries	-	Projection to compute **Q** initialised with $\mathcal{N}(0, \lvert kd \rvert^{-\frac{1}{2}})$.
CUDA benchmark	`cudnn.benchmark`	False	`cudnn.benchmark = True`[14], which could speed up the covergence rate is not used. The reason for this is twofold. Firstly, Bello does not mention that he used this (though it could be a common convention that this is always used). Secondly, as the benchmark tries multiple convolution algorithms it could be that the convolution algorithms between the ResNet-50 and the ResNet-50 with lambda layers are not equal, and therefore the results could not be compared.

Table 3: Additional general implementation details

Parameter	**Implementation**	**Description**
Batch-normalization	BatchNorm2d	Batch-normalization is applied with a decay of 0.9999 over trainable parameters.
Optimizer	Adam	Networks were trained via back-propagation using Adam with the weight decay and the initial learning rate mentioned in Table 1, and betas = (0.9, 0.9999).
Data augmentation	Random crops & horizontal flips	Standard training data augmentation (random crops and horizontal flip with 50% probability). Quoted from (He et al. 2016): "4 pixels are padded on each side and a 32 × 32 crop is randomly sampled from the padded image or its horizontal flip."
Label-smoothing	0.1	ResNet-50 does not automatically apply label-smoothing, so it had to added. More about this can be read in subsection 8.3.
Learning rate scheduler (epoch 5 - 90)	CosineAnnealingLR	The learning rate from epoch 5 to 90, is defined by a `CosineAnnealingLR`.

8 Implementation details: ResNet-50

8.1 Clarification of ambiguities

Upon implementing the lambda layers for the first time and executing the first trial runs, we realised there exist a few ambiguities in the paper that limit a successful implementation.

8.1.1 ResNet-50 vs ResNet-RS

When reading the original lambda layer paper (Bello 2021) the reader can be confused whether ResNet-50 was used for obtaining the results presented in Table 3 or ResNet-RS, as mentioned in page 29. The problem is that the term "LambdaResNets" has been used for both cases in different sections of the paper. We concluded that ResNet-50 was used instead of the modified version

[14] `https://pytorch.org/docs/stable/backends.html`

ResNet-RS thanks to two observations.

First, it is very unlikely that the author used as baseline for the lambda layers paper a modified version of ResNet-50 that he proposes on a paper published around the same time Bello et al. 2021.

The second observation consists of comparing the results of multiple tables, namely Table 3, Table 4 and Table 12 of the original paper (Bello 2021). Table 3 (Figure 1) shows that for the Lambda layer, the values for "Params (M)" and "top-1" are 15.0 and 78.4, respectively. The value of 78.4 can be recognized in Table 4 shown in Figure 4 for the lambda layer, where it is stated that the "Throughput" equals 1160ex/s. Both of the aforementioned values for the accuracy and the throughput can be seen in Table 12, shown in Figure 5, for the $\mathrm{L} \rightarrow \mathrm{L} \rightarrow \mathrm{L} \rightarrow \mathrm{L}$ architecture. In the caption of this table, it is clearly stated that the ResNet-50 architecture was used. The 3x3 convolutions in the last 4 stages of the ResNet-50 were changed by lambda layers. These two observations allowed us to safely discard the ResNet-RS as the baseline.

Layer	Space Complexity	Memory (GB)	Throughput	top-1
Global self-attention	$\Theta(blhn^2)$	120	OOM	OOM
Axial self-attention	$\Theta(blhn\sqrt{n})$	4.8	960 ex/s	77.5
Local self-attention (7x7)	$\Theta(blhnm)$	-	440 ex/s	77.4
Lambda layer	$\Theta(lkn^2)$	1.9	1160ex/s	**78.4**
Lambda layer ($\|k\|$=8)	$\Theta(lkn^2)$	0.95	**1640** ex/s	77.9
Lambda layer (shared embeddings)	$\Theta(kn^2)$	**0.63**	1210 ex/s	78.0
Lambda convolution (7x7)	$\Theta(lknm)$	-	1100 ex/s	78.1

Figure 4: The lambda layer reaches higher ImageNet accuracies while being faster and more memory-efficient than self-attention alternatives (Bello 2021)

Architecture	Params (M)	Throughput	top-1
$\mathbf{C} \rightarrow \mathbf{C} \rightarrow \mathbf{C} \rightarrow \mathbf{C}$	25.6	7240 ex/s	76.9
$\mathbf{L} \rightarrow \mathbf{C} \rightarrow \mathbf{C} \rightarrow \mathbf{C}$	25.5	1880 ex/s	77.3
$\mathbf{L} \rightarrow \mathbf{L} \rightarrow \mathbf{C} \rightarrow \mathbf{C}$	25.0	1280 ex/s	77.2
$\mathbf{L} \rightarrow \mathbf{L} \rightarrow \mathbf{L} \rightarrow \mathbf{C}$	21.7	1160 ex/s	77.8
$\mathbf{L} \rightarrow \mathbf{L} \rightarrow \mathbf{L} \rightarrow \mathbf{L}$	15.0	1160 ex/s	78.4
$\mathbf{C} \rightarrow \mathbf{L} \rightarrow \mathbf{L} \rightarrow \mathbf{L}$	15.1	2200 ex/s	78.3
$\mathbf{C} \rightarrow \mathbf{C} \rightarrow \mathbf{L} \rightarrow \mathbf{L}$	15.4	4980 ex/s	78.3
$\mathbf{C} \rightarrow \mathbf{C} \rightarrow \mathbf{C} \rightarrow \mathbf{L}$	18.8	7160 ex/s	77.3

Figure 5: Hybrid models achieve better speed-accuracy trade-off (Bello 2021)

After having decided that ResNet-50 was the correct baseline, multiple implementations were found using different machine learning libraries and versions. The highlighted implementation was the one chosen for this work.

1. Torch: Facebook AI Research has released the code in this machine learning library including pre-trained models. They have written a blog about

the implementation [15] and released the code in GitHub [16].

2. PyTorch: more recent implementation of the previous contribution [17].

3. **PyTorch**: implementation of the PyTorch team of multiple computer vision architectures [18], including ResNet-50.

4. Keras 1.0: third-party implementation [19].

8.1.2 Learning rate

The first run was performed using the parameters as described above with momentum equal to 0.9999 for the `BatchNorm2D` for 90, 180, and 350 epochs. Unfortunately, the performance was not higher than 45% and it did not lead to a converging result. Therefore, the paper was looked at again with a fresh mindset and it was determined that the learning rate as obtained via the method of the paper was way too high, as it reached a maximum of 1.6 while the default for Adam in Pytorch is 0.001. Therefore, we decided to modify the learning rate in order to find the optimal performance for the CIFAR-10 dataset. The performance variations when carrying out the learning rate sensitivity analysis are reported later in the results.

The final calculation which led to the best results was performed with $\alpha_{max} = 0.0005$. This resulted in a learning rate of [0.0005, 0.001, 0.0015, 0.002, 0.0025] over the first 5 epochs before it was scaled down using the `CosineAnnealingLR` method over the remaining epochs. As a result, accuracies around 75% were reached after 12 epochs already. This took between 42 seconds per epoch.

8.2 Data pre-processing

Before the data can be fed to the algorithms, it is pre-processed. Here, the data is first downloaded, augmented and normalised. For that purpose, the original images are randomly cropped maintaining the original size by padding all image sides with four black pixels. Additionally, images are randomly flipped with 50% probability and are normalised with means [0.4914, 0.4822, 0.4465] and variances [0.2023, 0.1994, 0.2010]; values which correspond to the means and variances of the CIFAR-10 data along each of the 3 image dimensions. The transformation for the training dataset can be observed in the next code snippet.

```
train_transform = torchvision.transforms.Compose([
    torchvision.transforms.RandomCrop(32, padding=4),
    torchvision.transforms.RandomHorizontalFlip(p=0.5),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.4914, 0.4822, 0.4465), (0.
                                      2023, 0.1994, 0.2010))])
```

[15] http://torch.ch/blog/2016/02/04/resnets.html
[16] https://github.com/facebookarchive/fb.resnet.torch
[17] https://github.com/pytorch/examples/tree/master/imagenet
[18] https://pytorch.org/hub/pytorch_vision_resnet/
[19] https://github.com/raghakot/keras-resnet

After that, the training and test data are fed to iterators (DataLoader) and a sample of training images is plotted for visual verification.

8.3 Model preparation

Before executing the training and testing with the pre-processed data, the model needs to be created. For that purpose, the loss function and the optimizer need to be defined, as well as the learning rate scheduler.

For the definition of the loss function or criterion, the author of the LambdaNetworks paper mentions that label-smoothing with a smoothing value of 0.1 was used. However, this could not be found by default in the chosen ResNet-50 architecture implementation. As a result, the label smoothing presented by Christian Szegedy in the paper "Rethinking the Inception Architecture for Computer Vision" (Szegedy et al. 2016) had to be implemented. For that purpose, we adjusted the label smoothing proposed by Suvojit Manna [20], which lead to the final compact form shown in the code-snippet below. As can be seen, the loss applied before the label smoothing is cross-entropy with log softmax activations.

```
class LabelSmoothing(nn.Module):
    """
    Negative log-likelihood loss with label smoothing.
    """
    def __init__(self, smoothing=0.0):
        """
        Constructor for the LabelSmoothing module.
        :param smoothing: label smoothing factor
        """
        super(LabelSmoothing, self).__init__()
        self.confidence = 1.0 - smoothing
        self.smoothing = smoothing

    def forward(self, x, target):
        """
        Function of label smoothing according to the equations in
                                        page 7 of the paper: '
                                        Rethinking the Inception
        Architecture for Computer Vision'
        """
        # Computes the logsoftmax of the predictions. For each
                                        datapoint, there are 10
                                        numbers, each represents
                                        a label
        logprobs = nn.functional.log_softmax(x, dim=-1)

        # Obtain the prediction obtained for the right labels. They
                                         should be as close to 1
                                        as possible
        nll_loss = -logprobs.gather(dim=-1, index=target.unsqueeze(
                                        1))

        nll_loss = nll_loss.squeeze(1)        # Compute the first
                                        term of the equation: the
                                         predicted value of the
                                        labels
```

[20] https://gist.github.com/suvojit-0x55aa/0afb3eefbb26d33f54e1fb9f94d6b609

```
        smooth_loss = -logprobs.mean(dim=-1)  # Compute the second
                                              term: average among all
                                              label predictions
        loss = self.confidence * nll_loss + self.smoothing *
                                              smooth_loss    # Sum both
                                               terms weighted with the
                                              smoothing
        return loss.mean()    # Compute the batch loss
```

Furthermore, the Adam optimizer was used with the hyperparameters (weight decay, initial learning rate, etc.) presented in subsection 7.3. The learning rate scheduler is based on a combination of the linear and cosine learning rate schedulers presented in subsection 7.3 and subsubsection 8.1.2.

In the model preparation, it is further checked whether the model can be run in the host GPU with CUDA and, in the case that is specified by the user, it fills the model parameters with those stored in the user specified checkpoint of the previous run. As design choice, the model automatically stores checkpoints every 5 epochs.

8.4 Training and testing

Once the model has been defined and the data pre-processed, the training and testing were executed simultaneously for 90 epochs on a Nvidia Quadro T1000 max-Q 4GB DDR5 (more details in subsection 7.1). During this process, two measures have been implemented in order to guarantee a smooth analysis and comparison of the results. First, the train and test accuracy, loss and computation time, as well as the learning rate for every epoch, are stored in log text files for posterior analysis. Additionally, the accuracy and loss information is stored such that it can be plotted in Tensorboard.

9 Implementation details: LambdaNetworks

9.1 Single lambda layer implementation

Even though some code snippets are provided in the paper, it assumes that the keys, values and queries have already been computed, no layer parameter initialisation is provided and it is not clear how the position embedding are defined. For the implementation of the lambda layer and its smooth later integration within the ResNet-50 architecture, a class was created with 3 main methods:

1. `__init__`: initialisation method that receives as input the input size (|n|), the context size (|m|), the value size (|v|), the output size (d), the number of heads (h) and the position embeddings ($\boldsymbol{E}$). The position embeddings are not created within the class such that they can be easily shared among multiple layers. Outside of the class, the embeddings are created and fed as input to different lambda layers. The instantiation of the position embeddings and their initialisation with $\mathcal{N}(0,1)$ outside of the lambda layer is defined as follows:

```
# Initialisation of the embedding matrices
```

```
embedding = nn.Parameter(torch.Tensor(self.input_size, self.
                                      context_size, self.qk_size),
                                       requires_grad=True)
torch.nn.init.normal_(embedding, mean=0.0, std=1.0)
```

For the computation of the keys, values and queries, single linear transformations are applied without biases. In the case of the queries, the output of the linear transformation would be of size $|\mathrm{n}| \times (kh)$, as discussed in subsection 6.4. These transformations can be observed in the following code:

```
self.toqueries = nn.Linear(self.d, self.k * self.h, bias=False
                                   )
self.tokeys    = nn.Linear(self.d, self.k, bias=False)
self.tovalues  = nn.Linear(self.d, self.v, bias=False)
```

Additionally, 1D batch normalisation layers are instantiated for the values and the queries, as well as a softmax function for the keys.

```
# Create batch normalization layers
self.bn_values = nn.BatchNorm1d(self.m)
self.bn_queries = nn.BatchNorm2d(self.k)

# Keys softmax function for the keys
self.softmax = nn.Softmax(dim=1)
```

This function ends calling the "reset_params" method, which is explained next.

2. `reset_params`: initialises the learnt matrices of the lambda layer, namely the key, value and query projection matrices with the same normal distributions mentioned in the paper. The position embedding are initialised outside of the lambda layer as mentioned before.

```
td_kv = 1/np.sqrt(self.d)  # standard deviation for the key
                                      and value
std_q = 1/np.sqrt(self.d * self.k)  # standard deviation for
                                      the query
torch.nn.init.normal_(self.toqueries.weight, mean=0.0, std=
                                      std_q)  # initialise of the
                                      query projection matrix
torch.nn.init.normal_(self.tokeys.weight, mean=0.0, std=std_kv
                                      )  # initialise of the keys
                                      projection matrix
torch.nn.init.normal_(self.tovalues.weight, mean=0.0, std=
                                      std_kv)  # initialise of the
                                       values projection matrix
```

3. `forward`: function that is run during the forward propagation of the neural network. First, since the lambda layer requires to compress the input image height and width into a single dimension, namely $|\mathrm{n}|$, the input x is reshaped as follows:

```
# Obtain the batch_size
b, d, n1, n2 = x.size()             # b-d-n1-n2
n = n1*n2                           # compute the number of
                                    pixels in the input
x = torch.reshape(x, [b, n, d])     # b-n-d
```

Furthermore, since the global context is used for the current reproducibility project, the context is also obtained from resizing the input accordingly:

```
# Reshape the context
c = torch.reshape(x, [b, self.m, d])
```

The next step is to compute the keys, queries and values using the linear transformations and normalisations defined in `__init__`:

```
# Compute the keys
keys = self.tokeys(c)                    # b-m-k
softmax_keys = self.softmax(keys)        # b-m-k

# Compute the values
values = self.bn_values(self.tovalues(c))    # b-m-v

# Compute the queries
queries = torch.reshape(self.toqueries(x), [b, n, self.k, self
                                    .h])  # b-n-k-h
queries = torch.transpose(queries, 1, 2)  # b-k-n-h
queries = self.bn_queries(queries)  # b-k-n-h
```

Finally, the lambdas and outputs are computed by using the torch.einsum function and following the equations outlined in section 6. The output is reshaped at the end such that it has the same number of dimensions as the input fed to the lambda layer

```
# Compute content lambda
content_lambda = torch.einsum('bmk, bmv->bkv', softmax_keys,
                                    values)     # b-k-v

# Compute position lambda
position_lambdas = torch.einsum('nmk, bmv->bnkv', self.E,
                                    values)        # b-n-k-v

# Compute content output
content_output = torch.einsum('bknh, bkv->bhvn', queries,
                                    content_lambda)    # b-h-v-n

# Compute position output
position_output = torch.einsum('bknh, bnkv->bhvn', queries,
                                    position_lambdas)    # b-h-v-
                                    n

# Compute output
output = torch.reshape(content_output + position_output, [b, d
                                    , n])    # b-d-n

# Reshape as an image
output = torch.reshape(output, [b, d, n1, n2])    # b-d-n1-n2
```

As can be observed, all the computations required to reproduce the lambda layer can be compressed in less than 20 lines of code.

9.2 Lambda layer integration within ResNet-50

When reading the original ResNet paper from Facebook AI Research (He et al. 2016), different architectures were proposed in Figure 6, whose main difference was the number of layers. As can be observed in the first column, the ResNet architectures contain 5 blocks (conv_1, conv_2, conv_3, conv_4 and conv_5) which are referred as cx stages in the lambda paper.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 6: ResNet architectures (He et al. 2016)

In Bello 2021 it is stated that the (non-hybrid) LambdaResNets are obtained by replacing the 3x3 convolutions in the bottleneck blocks, namely conv_2, conv_3, conv_4 and conv_5 of the ResNet architectures, by lambda layers. Similar information is conveyed by Figure 5 that shows that the last 4 stages are substituted by lambda layers. Therefore, the main change in the original ResNet-50 architecture can be found in the definition of the bottleneck layers, as well as the initialisation of the ResNet where the position embeddings are initialised; as it was previously discussed in subsection 9.1.

The following two code snippets shows part of the original ResNet-50 bottleneck layer initialisation and its corresponding lines of code in the ResNet-50 with integrated lambda layers. It clearly shows how the 3×3 CNN layers are exchanged for the lambda layers while the rest of the code remains intact.

```
self.conv1 = conv1x1(inplanes, width)
self.bn1 = norm_layer(width, momentum=0.9999)

self.conv2 = conv3x3(width, width, stride, groups, dilation)
self.bn2 = norm_layer(width, momentum=0.9999)

self.conv3 = conv1x1(width, planes * self.expansion)
self.bn3 = norm_layer(planes * self.expansion, momentum=0.9999)

self.relu = nn.ReLU(inplace=True)
```

```
self.conv1 = conv1x1(inplanes, width)
self.bn1 = norm_layer(width, momentum=0.9999)

self.conv2 = LambdaLayer(context_size, value_size, qk_size,
                                    output_size, heads, E)
self.bn2 = norm_layer(width, momentum=0.9999)

self.conv3 = conv1x1(width, planes * self.expansion)
self.bn3 = norm_layer(planes * self.expansion, momentum=0.9999)

self.relu = nn.ReLU(inplace=True)
```

Furthermore, as can be seen from the architecture presented in section 6, the output of the lambda layer has the same dimensions as its input. However, in the original ResNet-50 architecture, some of the 3×3 CNN layers decreased the size of the input image by using a stride higher than 1. In order to establish a fair comparison between the baseline and the lambda layer implementation, the images also need to be downsized in the lambda layer implementation at those stages of the ResNet-50 architecture where the original 3×3 CNN layers used a stride higher than 1. For that purpose, at those stages of the network, an average 2D pooling layer is introduced with a kernel size of 3×3, (1, 1) padding and the corresponding stride. The benefit of a pooling layer when compared to a 1×1 convolution is that no additional parameters need to be learnt by the network.

```
# When the stride is different than one, downsize the LambdaLayer
                                    output accordingly
downsample_output = nn.AvgPool2d(kernel_size=(3, 3), stride=stride,
                                     padding=(1, 1))
```

Whenever the image is downsampled, new position embeddings are generated that will be shared with the next lambda layers until the image is newly downsampled.

```
# If stride is not equal to one, downsize the LambdaLayer output
                                    accordingly
self.input_size = int(self.input_size/stride**2)
self.context_size = int(self.context_size / stride ** 2)

# Resize the embeddings according to the downsizing
E = nn.Parameter(torch.Tensor(self.input_size, self.context_size,
                                    self.qk_size), requires_grad=True
                                    )
torch.nn.init.normal_(E, mean=0.0, std=1.0)
```

Finally, Figure 7 shows the current architecture of a layer within a bottleneck layer of the ResNet-50 without and with lambda layers. In the scenario presented, the stride used by the layer is 2. In the original ResNet-50, there already existed a downsampling block made of 1×1 convolutions that was applied to the residual connection and that increased the dimensions of the bottleneck layer input to match its output. Now, the afore mentioned average pooling downsampling block is added at the end of the bottleneck layer. The implementation of this downsampling leads to a reduction in required GPU RAM from 5GB to 2GB.

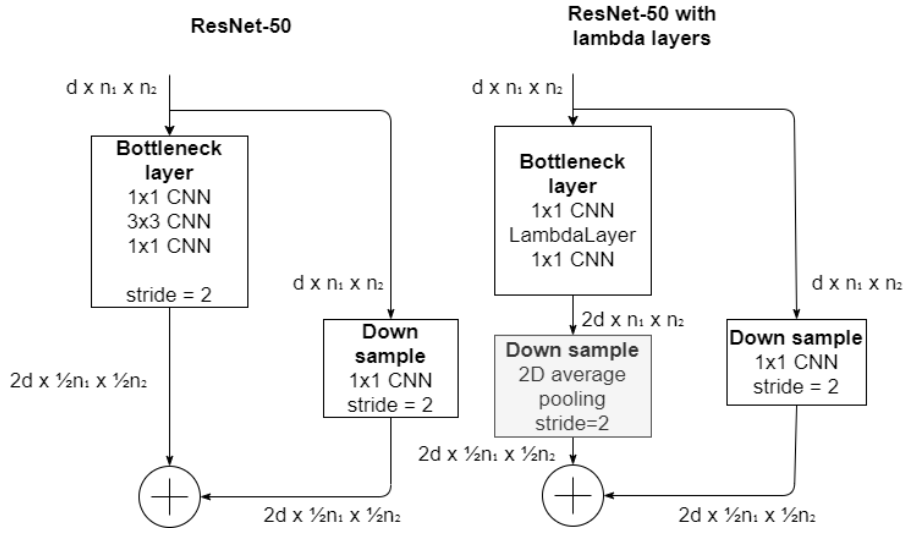


Figure 7: Block diagram showing the implementation of an additional downsampling block in the ResNet-50 with lambda layers based on 2D average pooling.

10 Implementation comparison

Myeongjun Kim [21] also implemented the lambda layers in the ResNet-50 architecture. In the following list we enumerate the main discrepancies between both implementations and the reason why we believe ours is a more faithful representation of the original LambdaNetworks paper:

- Kim does not use smoothing after applying cross-entropy loss, even though its application is stated in the paper.
- The original LambdaNetworks paper (Bello 2021) mentions the use of the cosine scheduler for the learning rate, but it does not specify the parameter used within the scheduler, such as the number of iteration after the scheduler restart (T_0) [22]. The paper that proposed this type of scheduler (Loshchilov and Hutter 2017) shows that the lowest test error is achieved when there are no restarts. Therefore, after the linear scheduler, we chose to implement the cosine scheduler such that it operates for the remaining 85 epochs without restarting. In contrast, Kim implemented the scheduler with $T_0 = 10$ and a multiplier factor after every restart (T_{mult}) of 2.
- Kim uses gradient clipping. However, this mechanism is not mentioned in the original LambdaNetworks paper (Bello 2021).
- Kim uses 2D convolutions for the computations of the keys, queries and values. However, in practice, the computations carried out by both implementations (kim's and ours) are identical.

[21] `https://github.com/leaderj1001/LambdaNetworks`
[22] `https://pytorch.org/docs/stable/optim.html`

- Kim uses embeddings of size $k \times u \times 1 \times m$ in the case that local contexts are used and ku in the case that the global counterpart is exploited. In Figure 2 and 3 of the original LambdaNetworks paper (Bello 2021) it can be observed that the embedding should be of size $n \times m \times k$.
- Kim applies dropout of 0.3 in the last classification layer, whereas it is not mentioned in the paper.

11 Results

In this section, the results of the reproducibility project are summarised. The main claims of the original paper on lambda layers are their superior performance and higher computationally efficiency when compared to their convolutional and attention counterparts. Therefore, subsection 11.1 and subsection 11.2 compare the accuracy of the original ResNet-50 to its modified version with lambda layers (section 9), as well as their required training computation times and throughput, respectively. Then, subsection 11.3 provides a brief sensitivity analysis performed on the initial learning rate in order to tune this hyperparameter to the new architecture-dataset combination.

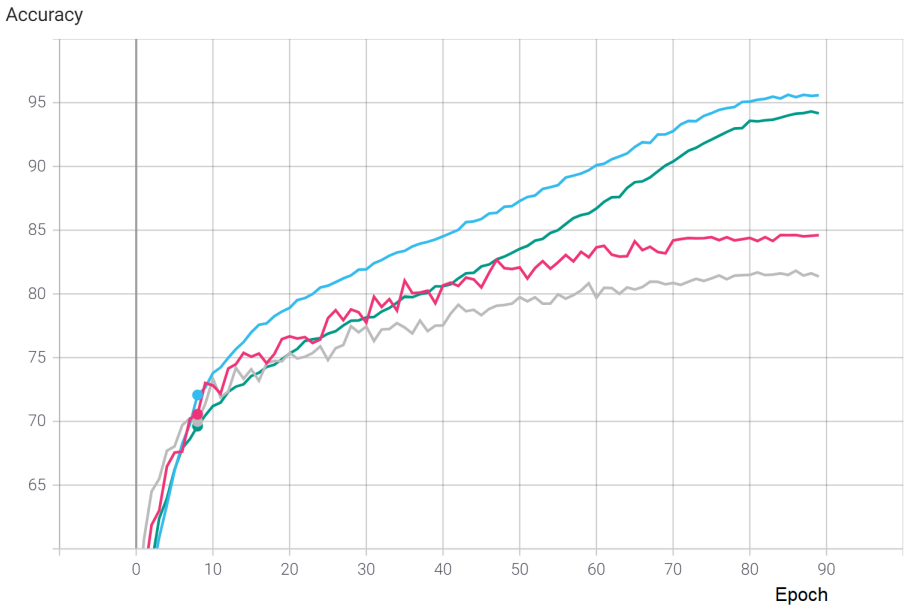
All the results were obtained with the parameters and values defined in subsection 7.3, except that the initial learning rate used was 0.0005; as it is discovered in subsection 11.3 to be the best choice.

11.1 Accuracy and model complexity

Figure 8 and Figure 9 present the accuracy and the loss of the training and test data splits for the ResNet-50 and its lambda layer modified version with respect to the number of epochs. It shows the top-1 accuracy, which is the percentage of datapoints for which their top class (the class with the highest probability after softmax) is the same as their corresponding targets. As can be observed, the training and testing accuracy performance measure of the ResNet-50 is higher than that of the version with lambda layers. In the case of the test data set, it is 3.2% higher. As a result, when trained on a lower dimensional dataset as CIFAR-10, lambda layers do not outperform the convolutional counterparts; however, they still reach competitive results.

On the original ImageNet dataset, Bello reports an accuracy of the convolutional baseline of 76.9%, whereas the lambda layers version acquires an accuracy of 78.4%. When compared to the results obtained in the CIFAR-10 dataset, the relation between both architectures has been reversed. Besides that, it has been observed that the accuracy of both architectures increases on CIFAR-10. This observation is alluded to the lower difficulty of classifying an image among 10 classes instead of 1000.

Finally, Bello reported that the baseline and the lambda layer models have 25.6M and 15M trainable parameters, respectively. In our case, they have 23.5M and 12.8M respectively. Since both models miss approximately the same number of parameters, namely 2M, the authors of this reproducibility project hypothesize that the missing parameters are from the borrowed ResNet-50 im-



Colour	Name	Top-1	Total time
	Top-1 test ResNet-50	84.6	1h 17m 29s
	Top-1 train ResNet-50	95.6	1h 17m 29s
	Top-1 test ResNet-50 with lambda layer	81.4	1h 3m 6s
	Top-1 train ResNet-50 with lambda layer	94.2	1h 3m 6s

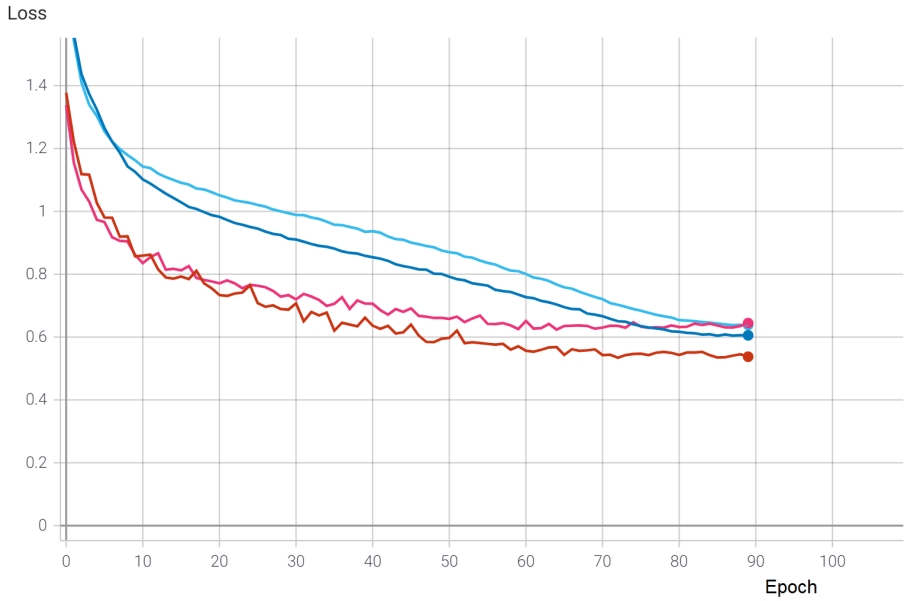
Figure 8: The accuracy diagram with the top-1 vs epochs, there are 90 epochs running from epoch 0 to epoch 89. The initial learning rate equals 0.0005, as defined in subsection 7.3.

plementation [23] and not from the lambda layer implementation presented in subsection 9.1.

11.2 Training time and throughput

Not only do autoreffig:accplot and Figure 9 show the accuracy, they also show the training time per model. For ResNet-50 this was 1 hour 17 minutes 29 seconds resulting in an average time of 51.6 seconds per epoch. For ResNet-50 with implemented lambda layer the total run time was with 1 hour 3 minutes 6 seconds, resulting in 42.1 seconds per epoch. Therefore, it may seem that the training and testing time of ResNet-50 is approximately 18.5% longer with only a mere 3.2% increase in accuracy and that the implementation of the lambda layers pays off. However, one does not only have access to the final result obtained at epoch 90, but also at intermediate results even though the timescale is different. Yet, from Figure 8 one can see that the top-1 test accuracy of the ResNet-50 with lambda layer at epoch 90 is already reached around epoch 41 for the baseline ResNet-50. In this case, epoch 41 for the baseline ResNet-50

[23] `https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py`



Colour	Name	Loss	Total time
	Loss test ResNet-50	0.54	1h 17m 29s
	Loss train ResNet-50	0.61	1h 17m 29s
	Loss test ResNet-50 with lambda layer	0.64	1h 3m 6s
	Loss train ResNet-50 with lambda layer	0.64	1h 3m 6s

Figure 9: The loss diagram with the loss vs epochs, there are 90 epochs running from epoch 0 to epoch 89. The initial learning rate equals 0.0005, as defined in subsection 7.3.

translates to about 35 minutes and 20 seconds and is therefore almost two times faster than its equivalent with lambda layers.

When comparing the throughput results of different architectures, it is necessary that they are run within the same platform. Given that Bello ran his algorithms on multiple TPUv3s, it is not possible to compare the throughput results of this reproducibility project with those of Bello. However, the throughput between the baseline and the lambda layer version on the CIFAR-10 could be compared. In the case of the baseline, a training epoch takes, in average, 50.92 s. Given that there are 50000 training samples, the throughput of the baseline is 981.93 ex/s. In the case of the lambda layers, a training epoch takes approximately 38.96 s. As a result, the lambda layer model has a throughput of 1283.26 ex/s. From these values, it can be seen that the lambda layer has a higher throughput than the convolutional counterpart, namely 31% higher.

11.3 Learning rate sensitivity analysis

Finally, in order to determine a good learning rate for the architecture-dataset combinations proposed in this reproducibility project, the initial learning rate (*initial_lr*) was modified. However, the learning rate scheduler was maintained constant. As a result, the learning rate of the first 5 epochs is defined as [*initial_lr*, $2 \cdot$ *initial_lr*, $3 \cdot$ *initial_lr*, $4 \cdot$ *initial_lr*, $5 \cdot$ *initial_lr*] and increases whereas the learning rates of the later epochs follow a cosine scheduler and decreases.

Top-1 test accuracies were obtained for the baseline and lambda models for the following initial learning rates: 0.01, 0.005, 0.001, 0.0005 and 0.0001. The highest learning rate was obtained using the formula proposed by Bello, namely $initial\_lr = \frac{0.1 \cdot B/256}{5}$, with B as our batch size of 128 samples. The following learning rates were obtained by successively halving the pre-computed highest learning rate. In both cases, a maximum in accuracy can be observed at 0.0005. The results can be seen in Figure 10.

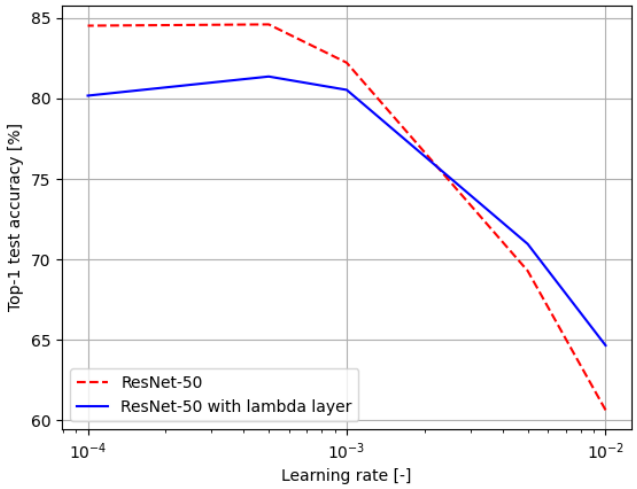


Figure 10: Accuracy as a function of the initial learning rate [0.01, 0.005, 0.001, 0.0005, 0.0001] for the ResNet-50 and implemented lambda layer architectures.

12 Conclusion

LambdaNetworks promise superior performance to convolutional and attention alternatives, as well as a lower memory footprint and a substantial speed-up during training and inference. The greater accuracy is mainly attributed to the combination of extracted position- and context-based interactions, whereas the lower memory usage and higher speed is achieved by bypassing the memory-intensive attention maps, sharing context information among the elements of the

batch and sharing the position embeddings among lambda layers. In the original paper (Bello 2021), Bello validates all these claims by training on ImageNet a ResNet-50 with lambda layers in exchange of its standard 3×3 convolutions. This choice of dataset difficults the reproducibility of the paper since the author uses 32 TPUv3 for training, resource far beyond the reach of most students and researchers. Therefore, the work here presented assessed the accuracy and speed of the lambda layers by training on a lower dimensional dataset, namely CIFAR-10. For that purpose, we have not only summarised the original paper on LambdaNetworks and presented our implementation, but we have also discussed those aspects that we considered ambiguous for understanding or reproducing it.

From the reproducibility project, we can highlight 4 main conclusions:

1. When trained on a lower dimensional dataset as CIFAR-10, lambda layers do not outperform the convolutional counterparts; however, they still reach competitive results.
2. On the ImageNet dataset, Bello reports a baseline accuracy of 76.9% and a lambda layer accuracy of 78.4%. The accuracy of both architectures increases on CIFAR-10. This observation is alluded to the lower difficulty of classifying an image among 10 classes versus 1000.
3. The lambda layer has a higher throughput than the convolutional counterpart, namely 31% higher. This results in lower training times.
4. The best initial learning rate is 0.0005 for both architectures.

The results from this work show that LambdaNetworks can be applied to smaller lower dimensional datasets with performances comparable to those presented in the original paper and within an affordable computational time budget, namely a few hours and not days.

The next steps of this work would be the implementation of the position lambda with local contexts in which $|m| \neq |n|$. Previous reproducibility attempts have used 3D convolutions for this purpose. However, the dimensions of the resulting relative position embeddings are $|m| \times |k|$ instead of $|n| \times |m| \times |k|$. As a result, they are not computing a relative position embedding for every pixel, but instead a single one for the complete input.

Finally, we hope that this work will contribute to the discussion and potential adoption of this novel architecture by the research community, and that our implementation will accelerate its integration by the industry.

13 Recommendations

In order to get a more sound conclusion on the amount of time it takes to obtain the train and test accuracy results, these runs have to be performed multiple times and post-processed by taking their mean or median. The importance of multiple runs increases when performing them in (free) Google Colab, since the compute availability can vary over time and one can not properly determine

which hardware configuration was used at each point in time. The results within a session however, are comparable, yet it would be advised to run the algorithm on a more stable platform in order to obtain more accurate running time results.

The maximum learning rate that Bello used with ImageNet differs substantially from what we used with CIFAR-10. While he used a specific formula to obtain the maximum learning rate, we tuned the learning rate using a coarse 1D-grid. It may not be fair to compare accuracies between his results and our results because he did not state how he obtained this formula. Perhaps the behavior of his trained ResNet-50 and ResNet-50 with lambda layer networks change significantly from ours as we optimized the learning rate. Therefore, it could be that there is much more potential in his network than shown. Additionally, the "optimal" initial learning rate of 0.0005 that we obtained was used for both networks: ResNet-50 and ResNet-50 with lambda layers. It could very well be that both could have a slightly different optimum learning rate and that these should be used instead. Therefore, it is recommended to perform further research in finding the optimum learning rate. In this light, it could also prove interesting to see whether the tuning or removal of warm restarts and the type of scheduler could have a significant influence on the accuracy.

14 View of the authors on the LambdaNetworks paper

The paper on LambdaNetworks promised great advances in terms of accelerating training and inference, as well as reducing the memory footprint, while maintaining or slightly increasing the performance when compared to existing attention-based alternatives. Hereby we would like to present to you our view on the paper with some aspects that made it stand out from existing literature, as well as some points that could be improved.

Always good news first: Even though the paper is very long, namely 31 pages, we were glad that the author was very complete and detailed when discussing the algorithm. Bello did not only include compact code-snippets with part of the implementation of the lambda layers but he also included all the information required to exactly reproduce the results (ambiguities have been discussed in this document), even the initialization used for the learnable parameters. Additionally, he included an ablation study to support his choice of architecture.

We were also positively surprised by Appendix A of the paper. It presents in a "Frequently Asked Questions" format theoretical and practical questions that the reader could ask himself/herself. Some of the entries of this appendix helped us to better understand the paper and implement it. Therefore, we believe that it is a great addition to the paper, as well as a successful format that the community could consider adopting in future publications. Upon submission to a conference, there are always questions posed by reviewers that the author responds in order to have his/her submission accepted. In the case that those questions do not lead to a change in the final document, they could be included

with the author response as a "Q&A" appendix.

Although it is true that for the reproduction of the paper it contains all the required information, we consider that the author presented too many variations of his model that makes its reading difficult and sometimes confusing. Therefore, we believe that the paper could have been split into multiple papers. For instance, in multiple parts of the paper, Bello mentions the use of the ResNet-RS with and without squeeze-and-excitation, an architecture that has been published at the same time as the lambda layers paper. The incorporation of this information scattered throughout the whole paper only contributes to the disorientation of the reader, even for a specialist in the field of attention since the information has only become recently available. Also, it is not very clear what is the difference between LambdaResnets and the ResNet architecture with lambda layers. We believe that the latter is a specific case of the former, but this is not clear specified in the paper.

Besides that, we are surprised by the difficulty and complexity of the LambdaNetworks paper when compared to the "Attention Is All You Need" paper (Vaswani et al. 2017). Even though the architectures and the general concept have many points in common, we found that the LambdaNetworks paper was an order of magnitude more difficult to read and to visualize. Given the huge potential of this architecture, we believe that some lessons could be learnt from the original papers that boosted the field of attention in machine learning in 2017.

Additionally, we believe Figure 2 in the original paper can be very misleading since it does not clearly reflect that the global context is available to the content lambda whereas a local context is used for the position lambda. Also, it does not show how the context is obtained from the layer input. Being it the only graphic displaying the concept, ambiguities in this figure can have a detrimental effect in the acceptance and diffusion of the lambda layer by the research community.

To conclude, even though the paper has multiple ambiguities, we have experienced first-hand the huge potential of this framework. Therefore, we hope that the author will base his future work on the lambda layers and will ease the accessibility to this novel method by providing a Google Colab tutorial and a step-by-step explanation of the concept. With the here presented reproducibility project and open-source implementation we hope to facilitate the access to this novel framework for the research community and industry.

References

Bello, Irwan (2021). "LambdaNetworks: Modeling long-range Interactions without Attention". In: *International Conference on Learning Representations*.

Bello, Irwan et al. (2021). "Revisiting ResNets: Improved Training and Scaling Strategies". In: arXiv: `2103.07579 [cs.CV]`.

He, Kaiming et al. (2016). "Deep residual learning for image recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. DOI: 10.1109/cvpr.2016.90.

Ioffe, Sergey and Christian Szegedy (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France: JMLR.org, pp. 448–456.

Li, Rui et al. (2020). *Linear Attention Mechanism: An Efficient Attention for Semantic Segmentation*. arXiv: 2007.14902 [cs.CV].

Loshchilov, Ilya and Frank Hutter (2017). "SGDR: Stochastic Gradient Descent with Warm Restarts". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

Szegedy, Christian et al. (2016). "Rethinking the Inception Architecture for Computer Vision". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2818–2826.

Vaswani, Ashish et al. (2017). "Attention is all you need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Long Beach, CA: Curran Associates, Inc.