Transfer Learning Between Different Architectures Via Weights Injection

Maciej A. Czyzewski
Poznan University of Technology, Poznan, Poland
maciejanthonyczyzewski@gmail.com

Abstract

This work presents a naive algorithm for parameter transfer between different architectures with a computationally cheap injection technique (which does not require data). The primary objective is to speed up the training of neural networks from scratch. It was found in this study that transferring knowledge from any architecture was superior to Kaiming and Xavier for initialization. In conclusion, the method presented is found to converge faster, which makes it a drop-in replacement for classical methods. The method involves: 1) matching: the layers of the pre-trained model with the targeted model; 2) injection: the tensor is transformed into a desired shape. This work provides a comparison of similarity between the current SOTA architectures (ImageNet), by utilising TLI (Transfer Learning by Injection) score.

1 Introduction

We propose a naive method of transferring knowledge between teacher and student neural network: computationally cheap injection technique that does not require any data samples. The primary objective is to speed up the learning from scratch of a neural network, if there is no previous pre-trained model. We name this the TLI[1] (Transfer Learning by Injection) family of operations. The work presented in this paper provides a minimal proof of concept. Further research is required.

Student networks after transferring knowledge from teacher networks - that may be different or pre-trained on different domains - are more likely to reach convergence faster than the same student networks initialized with Xavier/Kaiming methods. Furthermore, a relationship exists between teacher-student similarity and convergence times. During the research, minor revisions to the architecture are made on a continuous basis. Typically, each model is designed to improve upon the previous model in some way. With our method, the models practically retain their previous performance and continue to converge further. There are a number of research workflows, including Kaggle Competitions, that can be accelerated by the TLI method.

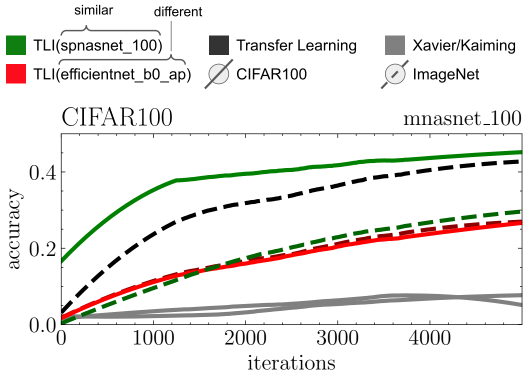


Figure 1: TLI (our) vs. standard methods. Training *mnasnet_100* on CIFAR100. Dashed line means that teacher was pre-trained on ImageNet.

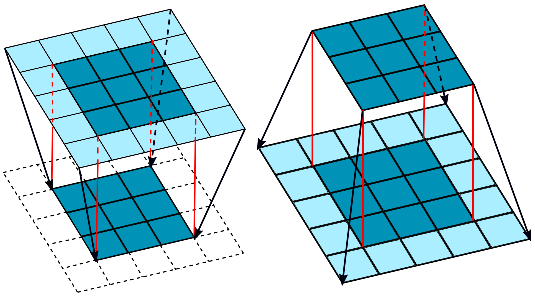


Figure 2: ComboInjection: is a mix of "center crop" (a) and "resize" (b), as tensor $x = \lambda a + (1 - \lambda)b$, where λ is strength of interpolation.

[1] Code: https://github.com/maciejczyzewski/tli-pytorch

The major contributions of this work:

1. Presenting the algorithm for transferring parameters between different architectures via computationally cheap injection technique (does not require data) - drop-in replacement for Xavier/Kaiming initialization(He et al., 2015).
2. Comparison of similarity between the current SOTA architectures (ImageNet), by utilising similarity score from presented method (Figure 7).

2 Related work

The term "parameter remapping" is used in (Fang et al., 2020), their work describes an efficient framework for neural architecture search (FNA++). Their method of transferring weights between different architectures is simple: their weights are transferred by matching layers on depth, width and kernel levels (crop center), which work only between same blocks. Therefore, this method is insufficient for more complex architectures.

There is also Net2Net described in (Chen et al., 2015) - in their work they presents a simple method to accelerate the training of larger neural networks by initializing them with parameters from a trained, smaller network. The random mapping algorithm for different layers was done manually. Developing a remapping algorithm would enable the Net2Net technique to be more general. This work further advances knowledge transfer by presenting a better remapping technique that generalises prior methods.

3 Weights Injection

This algorithm uses two models as inputs: a teacher model to transfer knowledge from, and a student model to transfer knowledge to. Different architectures are recommended, unless you are dealing with classic transfer learning (FT). The presented method operates on an execution graph and can be applied to a variety of tensor shapes.

The algorithm consists of two phases: 1) matching: the layers of the pre-trained model with the targeted model; 2) injection: the tensor is transformed into a desired shape. No data samples are used in the algorithm, and there is no mutual loss between layers. Moreover, the method can be extended to having multiple teachers, or to search for the best teacher from a library of pre-trained ones. Several different architectures may make use of the same blocks as student.

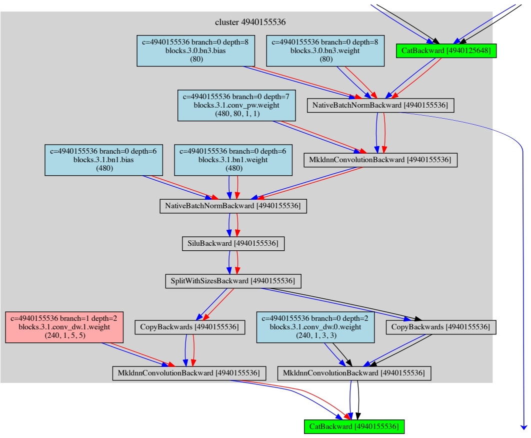

cluster 4940155536
c=4940155536 branch=0 depth=8
blocks.3.0.bn3.bias
(80)
c=4940155536 branch=0 depth=8
blocks.3.0.bn3.weight
(80)
CatBackward [4940125648]
c=4940155536 branch=0 depth=7
blocks.3.1.conv_pw.weight
(480, 80, 1, 1)
NativeBatchNormBackward [4940155536]
c=4940155536 branch=0 depth=6
blocks.3.1.bn1.bias
(480)
c=4940155536 branch=0 depth=6
blocks.3.1.bn1.weight
(480)
MkldnnConvolutionBackward [4940155536]
NativeBatchNormBackward [4940155536]
SiluBackward [4940155536]
SplitWithSizesBackward [4940155536]
c=4940155536 branch=1 depth=2
blocks.3.1.conv_dw.1.weight
(240, 1, 5, 5)
CopyBackwards [4940155536]
c=4940155536 branch=0 depth=2
blocks.3.1.conv_dw.0.weight
(240, 1, 3, 3)
CopyBackwards [4940155536]
MkldnnConvolutionBackward [4940155536]
MkldnnConvolutionBackward [4940155536]
CatBackward [4940155536]

Figure 3: The tensor with a red background indicates the analyzed weight, a red path indicates the execution path, and a green block indicates the operation (*CatBackward*).

3.1 Matching method: path algorithm & hashing

Both models provided as input (student/teacher) are parsed in the following way: 1) the execution graph is clustered into submodules, divided by operations *AddBackward0*, *MulBackward0*, *CatBackward* defined in PyTorch (Paszke et al., 2019)[2]; 2) for each tensor of weights, we need to find the path (list of operands) between one operation and another. 3) we iterate through the list of tensors of the model and the teacher, finding the most similar execution path (using scoring function). The Figure 3 illustrates this process.

[2] PyTorch 1.7.0 notation

In practice, this algorithm has O(nm) complexity - where n denotes the number of tensors containing the student weights, and m is the same as n but in the teacher model. In this work, we will not discuss any ways of increasing speed.

The following is considered during the scoring comparison of the two execution paths: depth; branch; used activations; submodule position from head; shape of tensors.[3]

3.2 Injection method: CenterCrop + Resize = ComboInjection

It is a combination of two operations: 1) resize to a new tensor size; and 2) crop center, it does not modify the weights (teacher shape unchanged). The strength of interpolation is controlled by the variable λ. Based on empirical data, it is best when λ is 0.75. The operation has a desirable quality since it does not alter the weights for the transferred tensor when the target and the input tensor have the same shape, mimicking classic transfer learning (loading parameters).

3.3 Multiple matches

In cases where we do not have a sure match, but a number of uncertain ones, the top K matches can be combined according to their weight according to the following:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \text{ for } i = 1, \dots, K \text{ and } \mathbf{z} = (z_1, \dots, z_K) \in \mathbb{R}^K \tag{1}$$

$$\boldsymbol{W}_s^{(j)} = \sum_{i=1}^{K} \sigma(\mathbf{z})_i T_i \tag{2}$$

To calculate our transformation mixing function (2), we will use our $\mathbf{z}$ score vector in conjunction with softmax (1).

4 Experiments

4.1 Datasets and implementation details

Optimizer: Adam (lr=0.003); batch_size=64; gradient accumulation (8 iterations) was used. A single iteration is defined as one batch fit. Every result is the average of three different runs from different seeds (series are normalized with savgol_filter). Besides image normalization (std/var), we did not use augmentation. Mixed precision is used for performance purposes. These models and their weights have been imported from (Wightman, 2019) library “PyTorch Image Models”.

4.2 Results and analysis

The TLI requires further study and rigorous experiments. The present work only involves simple experiments that can verify the proposed method only under some basic conditions. More studies are needed.

4.2.1 Initialisation on CIFAR100

Three different architectures were selected: mnasnet_100, spnasnet_100, tf_efficientnet_b0_ap. We choose mnasnet_100 as the base model to train on CIFAR100 (results in Figure 4). Knowledge was transfered from spnasnet_100 (green) and tf_efficientnet_b0_ap (red) - each in two options: a) pre-trained 5k iterations on CIFAR100 (normal line); b) original weights from pre-trained models on ImageNet (dashed line).

Assuming that classical transfer learning is not applicable in this experiment (black), we treat mnasnet_100 as a new architecture that has never been pre-trained before (for research purposes, we fine-tuned to compare). As can be seen any TLI is better than Xavier/Kamming initialization. A higher TLI score indicates that architectures are more similar to each other, resulting in faster convergence.

[3] This work is a draft, a thorough analysis of the formula and math will be presented in the final version.

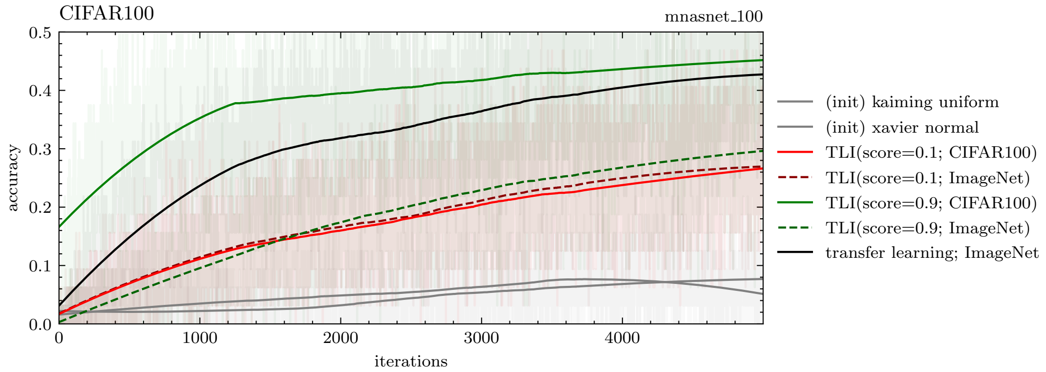
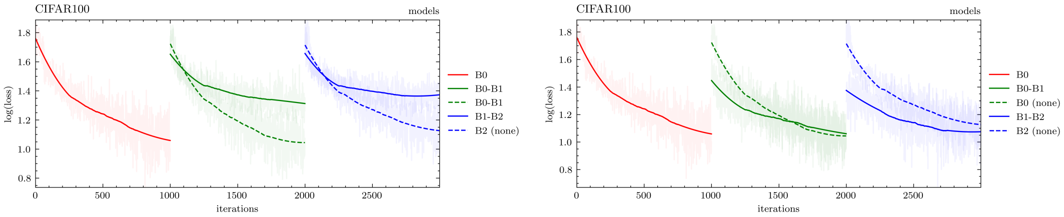


Figure 4: Compared with traditional initialization methods, TLI-based initialization converges faster after a few initial epochs (epoch=1.5k iterations). When an architecture is new or layers have been modified, transfer learning (FT) is not applicable. TLI(score=0.9) = *spnasnet_100*; TLI(score=0.1) = *tf_efficientnet_b0_ap* (TLI scores are in Figure 7)

4.2.2 Without/with BatchNorm injection



(a) without

(b) with

Figure 5: Comparison of impact of transferring BatchNorm weights.

This experiment will test whether it is worthwhile to transfer BatchNorm between different architectures (EfficientNet-B0, EfficientNet-B1, and EfficientNet-B2). The EfficientNets described in (Tan & Le, 2019) were selected because they are similar in structure and block architecture (high TLI score). Training each model involves 1000 iterations, following which TLI is applied to progressively larger models. We will perform the first experiment with BatchNorm, followed by a second experiment without BatchNorm.

When architectures are similar or pre-trained using a task dataset, transferring BatchNorm weights results in higher efficiency in most cases (in Figure 5).

4.2.3 Use case: Kaggle Competitions

Many competitions use pre-trained models such as EfficientNet as a baseline model. These models are then adapted for a new task (only output layers), and trained on the competition dataset. This phase can be called fine-tuning (FT). As a result, such a model, which has previously been trained on ImageNet, will often adapt to competition very quickly (e.g. 100 epochs). However, it is problematic to create a new architecture specifically for a competition problem. Because it requires a lot of research and it takes a lot of computational resources (training from scratch). In certain situations, increasing the size of filters or strides improves the performance of tasks with high-resolution images. Typically, manual weight assignment will make it unnecessary to undergo excessive training. This research proposes the TLI algorithm as an automated method of solving this problem.

This is the proposed pipeline:

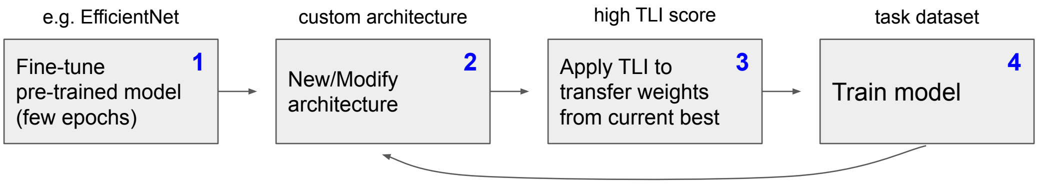


Figure 6: Pipeline.

1. train a few epochs model such as EfficientNet (generally, model with the most similar TLI score to targeted model) on our task dataset. This step can be omitted if there are public weights on Kaggle.
2. create a new or modify architecture with new features that will improve performance.
3. use TLI for transfer learning (from pre-trained model to new architecture).
4. train/fine-tune model, after a few epochs it accuracy should be equal to the result of the pre-trained fine-tuned model (like EfficientNet) chosen as teacher in step 1.

When modifying activation function or one layer, the model should not lose its accuracy from the very first epoch (repeat step 2/3/4 but as a teacher use last trained model).

5 Conclusions

The hypothesis was tested whether it is better to transfer knowledge from any architecture than to utilize Xavier/Kaiming as an initialization method. It turned out that the presented technique converges faster, making it a drop-in replacement.

Acknowledgments

This work will be developed further in collaboration with Kamil Piechowiak and Daniel Nowak as part of a bachelor's thesis at the Poznan University of Technology, Poznan, Poland.

References

Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015.

Jiemin Fang, Yuzhu Sun, Qian Zhang, Kangjian Peng, Yuan Li, Wenyu Liu, and Xinggang Wang. Fna++: Fast network adaptation via parameter remapping and architecture search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

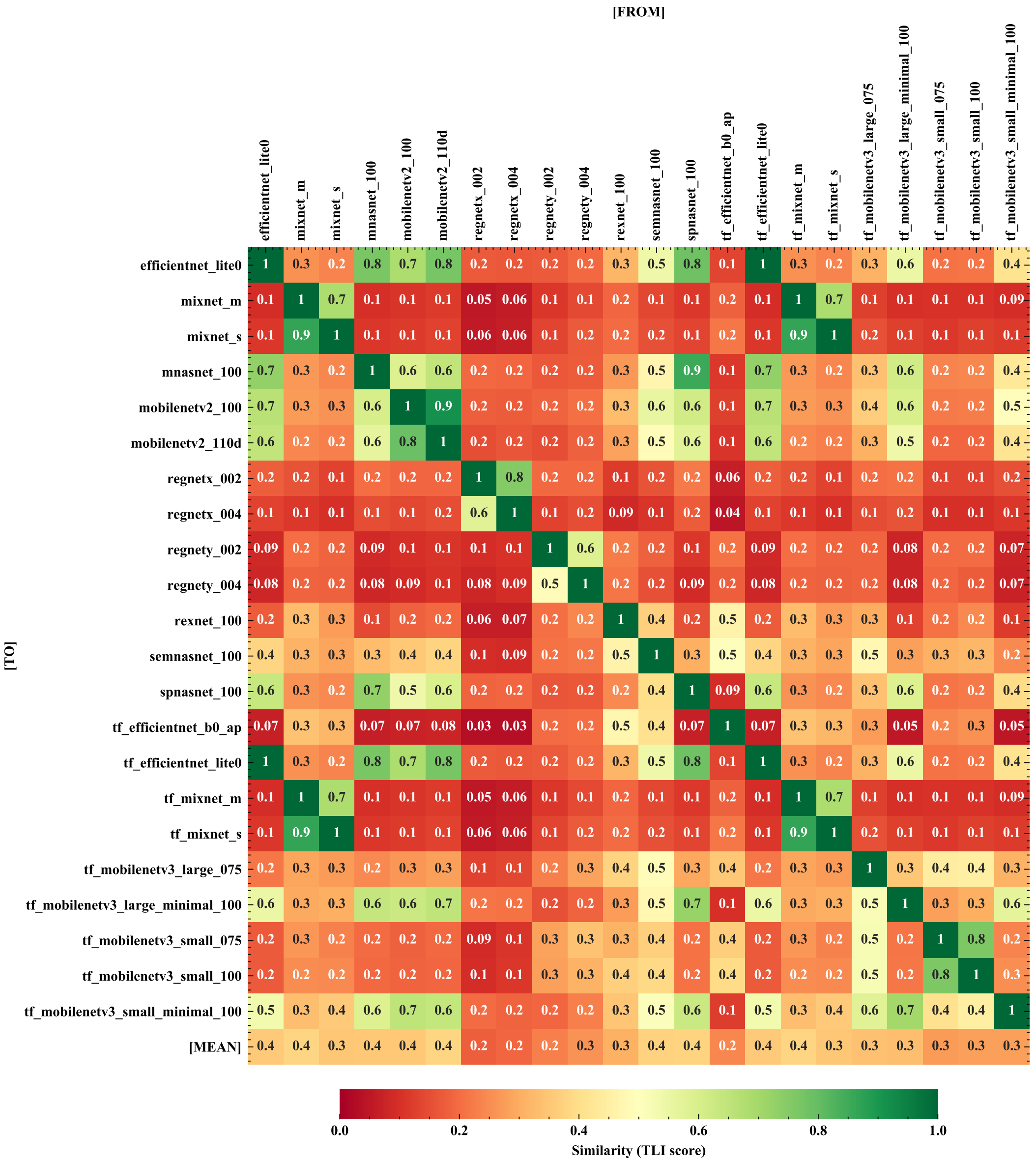
Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019.

Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

Ross Wightman. Pytorch image models. `https://github.com/rwightman/pytorch-image-models`, 2019.

A Appendix: similarity between architectures

A table below presents the scores for similarity in the range [0, 1], where 1 means they are identical, while a score below 0.5 indicates that they are significantly different from each other. There is no doubt that models like *tf_efficientnet_lite0* and B*efficientnet_lite0* generate similar results. Contrary to this, architectures such as RegNet and ResNet alternatives differ from one another in structure, as shown by the comparison.



	efficientnet_lite0	mixnet_m	mixnet_s	mnasnet_100	mobilenetv2_100	mobilenetv2_110d	regnetx_002	regnetx_004	regnety_002	regnety_004	rexnet_100	semnasnet_100	spnasnet_100	tf_efficientnet_b0_ap	tf_efficientnet_lite0	tf_mixnet_m	tf_mixnet_s	tf_mobilenetv3_large_075	tf_mobilenetv3_large_minimal_100	tf_mobilenetv3_small_075	tf_mobilenetv3_small_100	tf_mobilenetv3_small_minimal_100
efficientnet_lite0	1	0.3	0.2	0.8	0.7	0.8	0.2	0.2	0.2	0.2	0.3	0.5	0.8	0.1	1	0.3	0.2	0.3	0.6	0.2	0.2	0.4
mixnet_m	0.1	1	0.7	0.1	0.1	0.1	0.05	0.06	0.1	0.1	0.2	0.1	0.1	0.2	0.1	1	0.7	0.1	0.1	0.1	0.1	0.09
mixnet_s	0.1	0.9	1	0.1	0.1	0.1	0.06	0.06	0.1	0.2	0.2	0.2	0.1	0.2	0.1	0.9	1	0.2	0.1	0.1	0.1	0.1
mnasnet_100	0.7	0.3	0.2	1	0.6	0.6	0.2	0.2	0.2	0.2	0.3	0.5	0.9	0.1	0.7	0.3	0.2	0.3	0.6	0.2	0.2	0.4
mobilenetv2_100	0.7	0.3	0.3	0.6	1	0.9	0.2	0.2	0.2	0.2	0.3	0.6	0.6	0.1	0.7	0.3	0.3	0.4	0.6	0.2	0.2	0.5
mobilenetv2_110d	0.6	0.2	0.2	0.6	0.8	1	0.2	0.2	0.2	0.2	0.3	0.5	0.6	0.1	0.6	0.2	0.2	0.3	0.5	0.2	0.2	0.4
regnetx_002	0.2	0.2	0.1	0.2	0.2	0.2	1	0.8	0.2	0.2	0.1	0.2	0.2	0.06	0.2	0.2	0.1	0.2	0.2	0.1	0.1	0.2
regnetx_004	0.1	0.1	0.1	0.1	0.1	0.2	0.6	1	0.1	0.2	0.09	0.1	0.2	0.04	0.1	0.1	0.1	0.1	0.2	0.1	0.1	0.1
regnety_002	0.09	0.2	0.2	0.09	0.1	0.1	0.1	0.1	1	0.6	0.2	0.2	0.1	0.2	0.09	0.2	0.2	0.2	0.08	0.2	0.2	0.07
regnety_004	0.08	0.2	0.2	0.08	0.09	0.1	0.08	0.09	0.5	1	0.2	0.2	0.09	0.2	0.08	0.2	0.2	0.2	0.08	0.2	0.2	0.07
rexnet_100	0.2	0.3	0.3	0.1	0.2	0.2	0.06	0.07	0.2	0.2	1	0.4	0.2	0.5	0.2	0.3	0.3	0.3	0.1	0.2	0.2	0.1
semnasnet_100	0.4	0.3	0.3	0.3	0.4	0.4	0.1	0.09	0.2	0.2	0.5	1	0.3	0.5	0.4	0.3	0.3	0.5	0.3	0.3	0.3	0.2
spnasnet_100	0.6	0.3	0.2	0.7	0.5	0.6	0.2	0.2	0.2	0.2	0.2	0.4	1	0.09	0.6	0.3	0.2	0.3	0.6	0.2	0.2	0.4
tf_efficientnet_b0_ap	0.07	0.3	0.3	0.07	0.07	0.08	0.03	0.03	0.2	0.2	0.5	0.4	0.07	1	0.07	0.3	0.3	0.3	0.05	0.2	0.3	0.05
tf_efficientnet_lite0	1	0.3	0.2	0.8	0.7	0.8	0.2	0.2	0.2	0.2	0.3	0.5	0.8	0.1	1	0.3	0.2	0.3	0.6	0.2	0.2	0.4
tf_mixnet_m	0.1	1	0.7	0.1	0.1	0.1	0.05	0.06	0.1	0.1	0.2	0.1	0.1	0.2	0.1	1	0.7	0.1	0.1	0.1	0.1	0.09
tf_mixnet_s	0.1	0.9	1	0.1	0.1	0.1	0.06	0.06	0.1	0.2	0.2	0.2	0.1	0.2	0.1	0.9	1	0.2	0.1	0.1	0.1	0.1
tf_mobilenetv3_large_075	0.2	0.3	0.3	0.2	0.3	0.3	0.1	0.1	0.2	0.3	0.4	0.5	0.3	0.4	0.2	0.3	0.3	1	0.3	0.4	0.4	0.3
tf_mobilenetv3_large_minimal_100	0.6	0.3	0.3	0.6	0.6	0.7	0.2	0.2	0.2	0.2	0.3	0.5	0.7	0.1	0.6	0.3	0.3	0.5	1	0.3	0.3	0.6
tf_mobilenetv3_small_075	0.2	0.3	0.2	0.2	0.2	0.2	0.09	0.1	0.3	0.3	0.3	0.4	0.2	0.4	0.2	0.3	0.2	0.5	0.2	1	0.8	0.2
tf_mobilenetv3_small_100	0.2	0.2	0.2	0.2	0.2	0.2	0.1	0.1	0.3	0.3	0.4	0.4	0.2	0.4	0.2	0.2	0.2	0.5	0.2	0.8	1	0.3
tf_mobilenetv3_small_minimal_100	0.5	0.3	0.4	0.6	0.7	0.6	0.2	0.2	0.2	0.2	0.3	0.5	0.6	0.1	0.5	0.3	0.4	0.6	0.7	0.4	0.4	1
[MEAN]	0.4	0.4	0.3	0.4	0.4	0.4	0.2	0.2	0.2	0.3	0.3	0.4	0.4	0.2	0.4	0.4	0.3	0.3	0.3	0.3	0.3	0.3

Figure 7: Similarity between architectures.