©Copyright 2021
Sachin Mehta

Efficient Deep Learning for Visual and Textual Data

Sachin Mehta

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2021

Reading Committee:

Linda Shapiro, Chair

Hannaneh Hajishirzi, Chair

Ming-Ting Sun

Mohammad Rastegari

Program Authorized to Offer Degree:
Electrical Engineering

University of Washington

Abstract

Efficient Deep Learning for Visual and Textual Data

Sachin Mehta

Co-Chairs of the Supervisory Committee:
Professor Linda Shapiro
Electrical Engineering
Computer Science and Engineering

Assistant Professor Hannaneh Hajishirzi
Computer Science and Engineering

"*Less is more*" - Robert Browning, 1855

Efficient hardware, increased computational power, and smart sensors, are powering deep learning, and moving intelligence from the cloud to edge devices (e.g., smartphones, smart cameras, and wearables). However, deep neural networks are computationally expensive and are difficult to deploy on edge devices because of limited computational capabilities, including limited energy overhead. To enable on-device AI, we focus on developing efficient neural architectures for edge devices. We optimize the basic building blocks of deep neural networks (e.g., convolutional layers and linear transformation functions) and build light-weight, fast, and memory-efficient deep neural architectures. Besides efficiency, we also study the generalization capabilities of our architectures on different datasets and tasks.

We start by designing efficient architectures for computer vision tasks. In the first part of the dissertation, we introduce the Efficient Spatial Pyramid (ESP) unit, an efficient alternative to standard convolution layers in convolutional neural networks (CNNs). Compared to standard convolutions, the ESP unit allows networks to learn representations from a large receptive field with fewer parameters and operations. Our efficient architecture, `ESPNet`, that

is built using the ESP module is able to deliver similar or better performance than state-of-the-art efficient neural architectures, such as MobileNets and ShuffleNets, across different tasks (e.g., image classification and object detection) while being 2× more power efficient.

The second part is geared towards improving the performance of efficient CNNs. We introduce a novel and generic convolutional unit, the `DiCE` unit, that is built using dimension-wise convolutions and dimension-wise fusion. The dimension-wise convolutions apply light-weight convolutional filtering across each dimension of the input tensor, while dimension-wise fusion efficiently combines these dimension-wise representations; allowing the `DiCE` unit to efficiently encode spatial and channel-wise information contained in the input tensor. When `DiCE` units are stacked to build the `DiCENet` network, we observe better task-level generalization capabilities over state-of-the-art methods, including efficient architectures (e.g., MobileNetv3) that are constructed using neural architecture search.

Next, we focus on designing efficient architectures for natural language processing tasks. In the third part of this dissertation, we introduce the pyramidal recurrent unit (PRU), a drop-in replacement to widely used LSTMs. PRUs replace the linear transformations in LSTMs with pyramidal and group linear transformations; this enables learning representations in high dimensional space with fewer parameters and operations. Besides efficiency, our quantitative and qualitative analysis shows that PRUs have better gradient coverage as compared to LSTMs, which helps them deliver better performance.

In the fourth part, we introduce a deep and light-weight transformer (`DeLighT`) that delivers similar or better performance than standard transformer-based models with significantly fewer parameters and operations on sequence modeling tasks. `DeLighT` more efficiently allocates parameters both (1) within each Transformer block using the `DeLighT` transformation, a deep and light-weight transformation, and (2) across blocks using block-wise scaling, which allows for shallower and narrower `DeLighT` blocks near the input and wider and deeper `DeLighT` blocks near the output. Our empirical evaluation on machine translation and lan-

guage modeling tasks shows that `DeLighT` matches or improves the performance of baseline Transformers with 2 to 3 times fewer parameters on average.

Overall, this dissertation introduces neural architectures for learning generalizable representations from visual and textual data efficiently.

DEDICATION

To my parents and to Suman

ACKNOWLEDGMENTS

The journey toward this dissertation was, at times, torturous. It would not have been possible to complete without the support of the many nurturing people around me.

First and foremost, I am extremely grateful to my supervisors, Linda Shapiro and Hannaneh Hajishirzi, for their guidance, unparalleled support, and constructive criticism throughout my Ph.D. I thank them for their encouragement and for letting me choose my own research directions. Your impact on my life as a researcher and as a person is immense.

I would also like to extend my deepest gratitude to Mohammad Rastegari, whose expertise was invaluable in formulating research questions. His insightful suggestions pushed me to think deeper and brought my work to a higher level. I would also like to thank my other dissertation committee members, Ming-Ting Sun and John C. Kramlich, for the feedback they gave on my research proposal. I am also grateful to Rajarathnam Nallusamy, Balakrishnan Prabhakaran, Kanishka Lahiri, and Saumya Chandra for introducing me to research and for encouraging me to pursue a Ph.D. at the University of Washington. I also had the great pleasure of working with Anat Caspi, Luke Zettlemoyer, and Joann Elmore, all of whom inspired me to apply my work to impactful problems.

I would also like to thank my co-authors, especially Srinivasan Iyer and Ezgi Mercan, for their support and friendship. I am thankful to my friends from the UW GRAIL and UW NLP groups for their insightful research discussions and feedback on paper drafts. Some of these people include Nicholas Nuechterlein, Beibin Li, Rik Koncel-Kedziorski, Aida Amini, Dave Wadden, Collin Lockard, James Ferguson, Yi Luan, and Sewon Min. I would also like to acknowledge the assistance of the administrative staff at UW CSE and UW ECE.

I cannot leave UW without mentioning the continued and dependable support I receive

from my close friends. These include Gaurav Parashar, Michael Driscoll, Dhruv Gakkhar, Srinivasan Iyer, Rohan Patidar, Akshay Randhad, Gaurav Mahamuni, Sudipto Mukherjee, Mandar Joshi, Maaz Ahmed, Nikita Haduong, Wenjun Wu, Shima Nofallah, Deepali Nijhawan, Gagan Bansal, Bhargavi Paranjape, Guna Prasad, and Shalaka Bhawal. Thank you.

I give my deep and sincere gratitude to my family for their continuous and unparalleled love, for supporting me through all the ups and downs, and for their endless patience. I am forever indebted to my parents for giving me the opportunities and experiences that have made me who I am. I am deeply thankful to my brother, Saurabh, who has been a great support in my life. Lastly, I am thankful to my wife, Suman. This journey would not have been possible without your relentless support and encouragement. Thank you for everything.

TABLE OF CONTENTS

Page

List of Figures . . . iii

List of Tables . . . v

Chapter 1: Introduction . . . 1
- 1.1 Motivation . . . 4
- 1.2 Contributions and Dissertation Outline . . . 5

Chapter 2: Learning Visual Representations using Efficient Spatial Pyramidal Unit 8
- 2.1 Introduction . . . 8
- 2.2 Related Work . . . 11
- 2.3 `ESPNet` . . . 12
- 2.4 Experiments . . . 19
- 2.5 Ablation Studies on the ImageNet Dataset . . . 26
- 2.6 Summary . . . 29

Chapter 3: Dimension-wise Convolutions for Efficient Networks . . . 30
- 3.1 Introduction . . . 30
- 3.2 `DiCENet` . . . 32
- 3.3 Experimental Set-up . . . 38
- 3.4 Evaluating `DiCE` Unit on ImageNet . . . 40
- 3.5 Evaluating `DiCENet` on the ImageNet . . . 46
- 3.6 Task-level Generalization of `DiCENet` . . . 49
- 3.7 Summary . . . 53

Chapter 4: Efficient Recurrent Neural Network for Learning Textual Representations 54
- 4.1 Introduction . . . 54

4.2 Related work . . . 56
4.3 Pyramidal Recurrent Units . . . 57
4.4 Experiments . . . 61
4.5 Summary . . . 71

Chapter 5: Deep and Light-weight Transformer for Sequence Modeling . . . 72
5.1 Introduction . . . 72
5.2 Related Work . . . 73
5.3 DeLighT: Deep and Light-weight Transformer . . . 75
5.4 Experimental results . . . 82
5.5 Analysis and Discussions on Computational Efficiency . . . 89
5.6 Ablations on the task of Language Modeling . . . 91
5.7 Summary . . . 97

Chapter 6: Conclusion . . . 99
6.1 Future work . . . 101

Appendix A: Supplementary Material for the `DiCENet` Network . . . 124
A.1 Architecture of the `DiCENet` Network . . . 124

Appendix B: Supplementary Material for the `DeLighT` . . . 126
B.1 `DeLighT` Architectures for Language Modeling and Machine Translation . . . 126
B.2 Group linear transformation with Input-mixer connection . . . 128
B.3 Multiplication-Addition Operations in `DeLighT` . . . 128

LIST OF FIGURES

Figure Number Page

1.1 Comparison in terms of performance, battery discharge rate, power consumption, and inference speed of different semantic segmentation convolutional neural networks on a standard laptop with an NVIDIA GTX-960M GPU. . . 2
1.2 Thesis contribution: Efficient and generalizable neural architectures for visual and textual data. 6
2.1 Overview of the efficient spatial pyramid module. 9
2.2 Block diagrams of the ESP module. 14
2.3 Gridding artifact caused by dilated convolutions. 16
2.4 Strided ESP unit with shortcut connection to an input image for down-sampling. 17
2.5 Cyclic learning rate policy with linear learning rate decay and warm restarts. 20
2.6 Performance comparison of `ESPNet` with state-of-the-art efficient methods on the ImageNet dataset. 21
2.7 Performance improvement in F1-score of `ESPNet` over ShuffleNetv2 on MS-COCO multi-object classification task when tested at different image resolutions. 21
2.8 Performance analysis in terms of power consumption and inference speed of different efficient networks, including `ESPNet`. 22
2.9 `ESPNet`'s quantitative performance on the task of semantic segmentation. . . 24
2.10 `ESPNet`'s qualitative performance on *unseen images* on the task of semantic segmentation. 25
3.1 Separable convolutions vs. the `DiCE` unit. 31
3.2 Convolution-wise distribution of FLOPs for different networks with similar accuracy. 33
3.3 Overview of the `DiCE` unit. 34
3.4 `DiCE` unit for arbitrary sized input. 36
3.5 Integrating `DiCE` unit with different architecture designs for the task of image classification on the ImageNet dataset. 37
3.6 Implementation of dimension-wise convolution 39

3.7 Impact of MobileNet's hyper-parameters on the training of `DiCENet`. 45
3.8 `DiCENet`'s qualitative results on the task of semantic segmentation. 52

4.1 Training and validation curves for LSTM and pyramidal recurrent unit on the Penn Treebank dataset 55
4.2 Overview of pyramidal recurrent unit 55
4.3 Histogram of the entropies of next-token distributions predicted by the PRU and the LSTM on the Pentree bank validation set. 65
4.4 Variance of learned word embeddings for different categories of words on the PTB validation set. 66
4.5 Qualitative comparison between the LSTM and the PRU. 67
4.6 Impact of number of groups g and pyramidal levels K in PRU on the perplexity. 69

5.1 Block-wise comparison between the standard transformer block and the `DeLighT` block. 75
5.2 Illustration of the expansion phase in `DeLighT` transformation. 78
5.3 Block-wise scaling in `DeLighT`. 80
5.4 Comparison of `DeLighT` with Transformers and Evolved Transformers at different settings on the WMT'14 En-De corpus 85
5.5 Scaling up `DeLighT` models. 87
5.6 Impact of different transformations on the performance of `DeLighT`. 94
5.7 Impact of feature shuffling on the performance of `DeLighT`. 95
5.8 Impact of reduction factor r in light-weight FFN on the performance of `DeLighT`. 95
5.9 Uniform vs. block-wise scaling in `DeLighT`. 96
5.10 Scaling up `DeLighT` models 98

6.1 Summary of the dissertation 99

B.1 Sequence modeling with `DeLighT` 127
B.2 This figure visualizes different variants of group linear transformations that are used in the `DeLighT` transformation. 130

LIST OF TABLES

Table Number | Page

2.1 Comparison between different type of convolutions. 13

2.2 The `ESPNet` network at different computational complexities for classifying a 224×224 input into 1000 classes in the ImageNet dataset. 18

2.3 `ESPNet`'s quantitative and qualitative performance on the task of object detection. 27

2.4 Effect of different convolutions on the performance of `ESPNet`. 28

2.5 Performance of `ESPNet` under different settings, including hierarchical feature fusion and long-range shortcut connection with an input 28

3.1 Comparison between the `DiCE` unit and separable convolutions on the ImageNet dataset across different architectures. 41

3.2 Evaluating `DiCE` unit on the ImageNet dataset. 43

3.3 Impact of replacing all pointwise convolutions with `DimFuse`. Here, DWise denotes depth-wise convolution. 44

3.4 Comparison of `DiCENet` with state-of-the-art efficient methods on the ImageNet dataset. 47

3.5 Comparison of `DiCENet` with state-of-the-art methods in terms of inference speed. 48

3.6 Object detection using `DiCENet`. 50

3.7 `DiCENet`'s performance on the task of classifying multiple objects in an image. 53

4.1 Comparison of PRU with state-of-the-art methods on the task of language modeling. 63

4.2 Impact of different transformations used for processing input and context vectors in the pyramidal recurrent unit 70

4.3 Impact of different sub-sampling methods on the word-level perplexity. . . . 70

5.1 Comparison with baseline transformers on machine translation corpora. . . . 84

5.2 `DeLighT` networks are deep, light-weight and efficient as compared to transformers 85

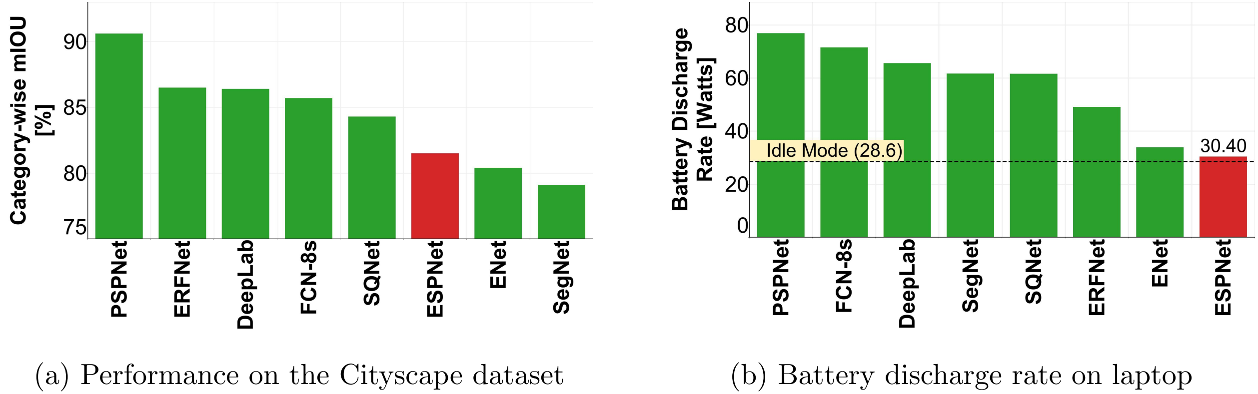
5.3 Comparison with state-of-the-art methods on machine translation corpora . . 86
5.4 DeLighT's quantitative performance on the WikiText-103 dataset. 88
5.5 Comparison of DeLighT with baseline transformers in terms of training speed and memory consumption. 90
5.6 DeLighT models require less regularization as compared to baseline transformers. 91
5.7 Ablations on different aspects of the DeLighT block 92
5.8 Effect of the position of DeLighT transformation on the performance of DeLighT. 97
A.1 Overall architecture of DiCENet at different network complexities for the ImageNet classification. 125

Chapter 1

INTRODUCTION

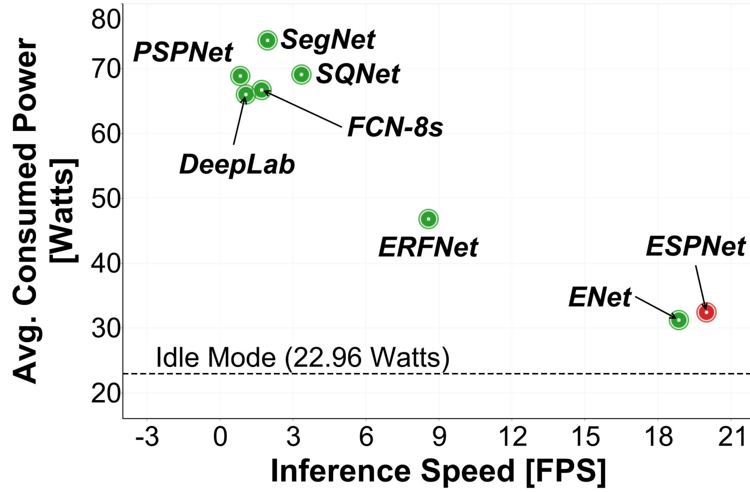
Deep neural networks (DNNs) have shown unprecedented success in the last few years. On the standard large scale benchmarking datasets for visual and textual recognition tasks, these networks can outperform traditional machine learning-based methods significantly and achieve human-level performance [1, 2]. For example, AlexNet [3], a DNN for large-scale object classification, outperformed the classification system of Lin el al. [4], which uses a histogram of oriented gradient (HOG) as features and a support vector machines (SVM) as a classifier on the ImageNet dataset [5] with 1000 object categories by 11% . DNNs are particularly more exciting today, because the advances in hardware technology and programming have made it feasible to train large DNNs with billions of network parameters and full-precision floating-point operations (FLOPs) on internet-scale data in days [6, 7].

Visual and textual recognition tasks provide core functionality for important real-world applications (e.g., self-driving cars, home monitoring systems, and machine translation). Large computational resources are required to power real-world applications with DNNs. Many of these real-world applications run on edge devices (e.g., wearables, smartphones, smart cameras, and tablets). Because these devices have limited computational resources (limited storage memory, RAM, and compute), running these DNNs on such devices is computationally not feasible. Additionally, many of these devices are battery-driven, therefore, running such computationally heavy DNNs on edge devices will quickly drain out the device's battery and pose deployment challenges. For instance, state-of-the-art semantic segmentation networks (e.g., PSPNet [14] and DeepLab [13]) are slow and power-hungry (see Figure 1.1) when executed on a laptop with a mobile GPU (NVIDIA GTX-960M). Similarly, the state-of-the-art sequence model, Transformer [17], learns 67 million parameters and requires



(a) Performance on the Cityscape dataset

(b) Battery discharge rate on laptop



(c) Power consumption vs. inference speed on a laptop

Figure 1.1: Comparison in terms of performance, battery discharge rate, power consumption, and inference speed of different semantic segmentation convolutional neural networks on a standard laptop with an NVIDIA GTX-960M GPU. The performance in (a) is *category-wise* mean intersection over union on the Cityscapes dataset [8]. The power and speed data is averaged over 200 trials, each trial processing an RGB image of size 1024×512. All networks (FCN-8s [9], SegNet [10], SQNet [11], ENet [12], DeepLab-v2 [13], PSPNet [14], ERFNet [15], and ESPNet (Ours) [16]) are converted to PyTorch for a fair comparison.

11 billion FLOPs to translate a sequence of 20 tokens (or words) in English to French [18].

To tackle the challenges and constraints imposed by these edge devices, a standard approach is to process the data generated by these devices on central servers or the cloud [19, 20]. However, communication between edge devices and servers increases the response time. Besides latency, such approaches also suffer from network bandwidth and coverage issues. This is especially problematic for applications that require a real-time response. Hence, there is a need for *on-device computing*.

Improving the efficiency of DNNs is an active area of research. The first line of research focuses on designing custom hardware accelerators (or AI accelerators) for different neural network operations (e.g., batch-wise matrix multiplication) [21–25]. Besides providing support for neural network operations, these accelerators also optimize the data and memory-related operations. These accelerators along with highly tuned software (e.g., cuDNN, The NVIDIA's CUDA® deep neural network library [26]) improve the latency and energy efficiency of DNNs significantly.

Quantization-based approaches [27–30] are further used to improve hardware efficiency, wherein full-precision floating-point operations in a pre-trained network are approximated with fewer bits (e.g., 1, 4, 8, or 16 bits). Most of the initial quantization methods focused on uniform quantization, i.e., the number of bits is fixed for all neural network operations. Though such approaches are effective in improving hardware efficiency, they result in performance degradation. Recent attempts are on mixed-precision operations [25]. Unlike previous quantization-based approaches, mixed-precision operations do not result in performance drop and have enabled the training of DNNs on internet-scale corpora [25, 31].

Though custom accelerators and quantization-based methods for DNNs are promising, they treat DNNs as a black box and optimizes only for hardware acceleration. It may be the case that the DNN architecture is not optimal at all. The second line of research focuses on improving the efficiency of DNNs using pruning-based methods [32–34]. These methods identify and remove redundant parameters in a network, which reduces the model size and latency. These methods also suggest that some neural network operations benefit more

from compression than others. For example, Li el al. [33] found that the 3×3 standard convolutional layers benefit the most from compression over other layers, including point-wise (1×1) convolutions and fully-connected layers. This suggests that there is plenty of opportunity in *designing efficient neural architectures*.

In this dissertation, we design neural architectures that are light-weight, fast, and energy-efficient for computer vision and natural language processing tasks. On the algorithm side, we develop novel transformation functions that allow us to learn deeper and wider representations efficiently. With these novel transformation functions, we can build very efficient and hardware friendly neural architectures that generalize to different tasks and datasets, and delivers similar or better performance than state-of-the-art networks with fewer parameters and operations. From the hardware perspective, efficient transformation functions have the potential to improve speed and energy efficiency, because they are light-weight and have fewer FLOPs. However, these transformation functions are not supported by most deep learning frameworks and hardware accelerators. Therefore, we also develop custom CUDA kernels to demonstrate the effectiveness of our efficient transformation functions.

1.1 Motivation

In this dissertation, we want to learn representations that can be generalized across different tasks and datasets using fewer parameters and operations. This is important because of the following:

Memory footprint: Efficient neural architectures have fewer parameters. Thus, these models have less overhead as they require less off-chip memory for storing network weights. Consider a home monitoring system, for instance, which periodically downloads the DNN models from remote servers. Since light-weight models are small, they are faster to transmit and update on the client devices; making the over-the-air update process easy.

Inference: Many applications run on edge devices (e.g., AR/VR glasses and smart cameras) and demand online processing of data with low latency. Light-weight models have fewer operations; allowing them to process the input with low latency. Also, these networks are small, so they can easily fit onto on-chip memory and helps reduce the read/write latency. This further improves the inference speed.

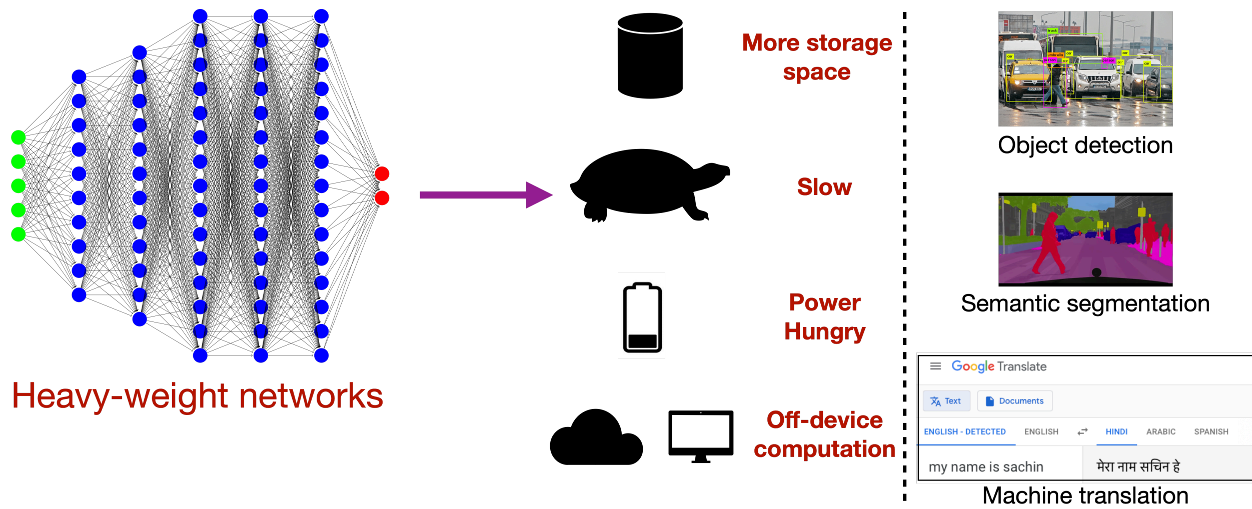
Power consumption: Efficient transformation functions reduce DNN operations in addition to network parameters. As a result, efficient DNNs have a lower battery discharge rate and consume less power in comparison to heavy-weight DNNs; allowing us to run applications for a longer duration on edge devices.

Carbon footprint: Efficient networks are fast and consume less power (or electricity). Since power is correlated with carbon emission, light-weight models have a lower carbon footprint.

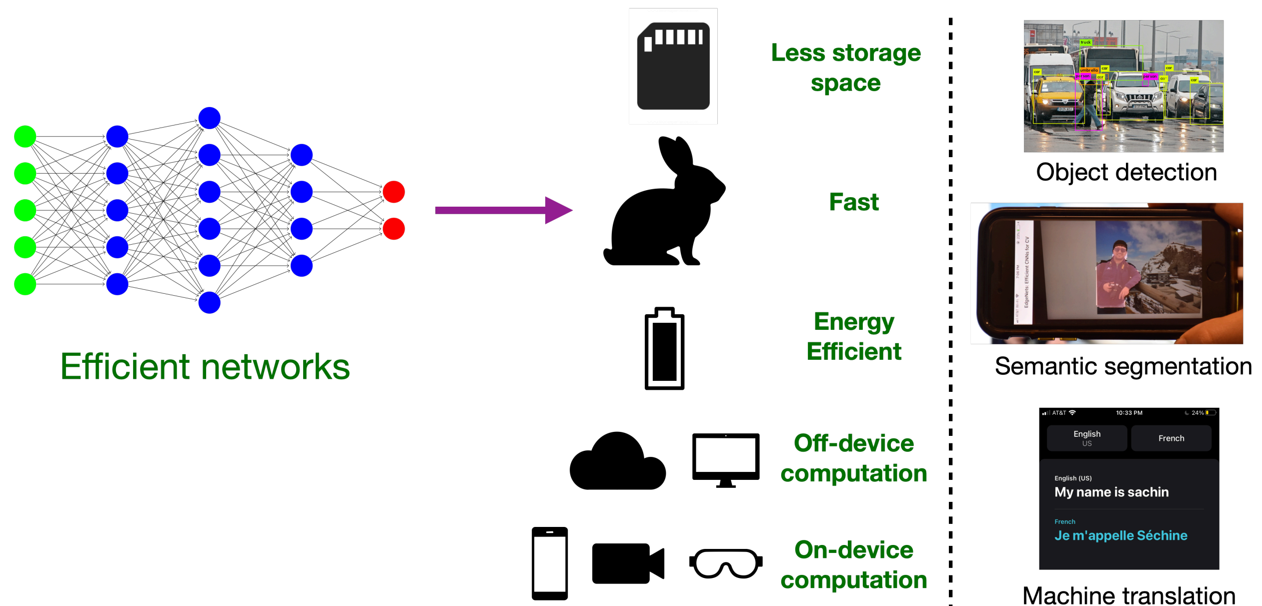
Scalability: Many vision and textual recognition applications powered by DNN, such as Google Translate, Google Vision API, and OpenAI API, run on the cloud. Running several instances of heavy-weight DNNs on the cloud to serve billions of users every day is very expensive. Since light-weight and efficient models are small, fast, and consume less power, many instances of such models can be run at a relatively lower cloud cost.

1.2 Contributions and Dissertation Outline

The basic building layer of DNNs for visual and textual data is convolutional layer (e.g., in convolutional neural networks [35]) and linear transformation function (e.g., in LSTMs [36] and Transformers [17]), respectively. Learning representations using these layers is computationally expensive. For example, when latent dimensions in LSTMs are scaled to increase the expressivity and capacity of a network, network parameters, operations, and model size also scale linearly with the latent dimension. Despite the performance improvements from



(a) Conventional approach



(b) Our approach

Figure 1.2: **Thesis contribution:** Efficient and generalizable (different tasks and datasets) neural architectures for visual (Chapter 2 & 3) and textual (Chapter 4 & 5) data.

such scaling, such heavy-weight networks cannot be used on edge devices because of limited computational resources. This leads us to following question:

How can neural architectures for visual and textual data that (1) are efficient, (2) deliver good performance, and (3) generalize to different tasks and datasets be designed?

In this dissertation, we answer this question. Algorithms introduced in this dissertation are centered around the development of efficient neural architectures (Figure 1.2).

Chapter 2 describes the *efficient spatial pyramid (ESP)* module for learning visual representations. ESP uses factorization to improve the efficiency of standard convolutions. Besides benchmarking the network built by stacking ESP modules, `ESPNet`, on the widely used ImageNet-1K object classification task, we also study its task-level generalizability on semantic segmentation and object detection tasks. The content of this chapter is based on [16, 37].

Chapter 3 introduces a novel convolutional layer, *dimension-wise convolution*, for modeling visual data. This layer allows us to aggregate information in a tensor efficiently and can be plugged easily into existing heavy-weight or light-weight DNNs to improve their efficiency and performance. The content of this chapter is based on [38].

Chapter 4 introduces a novel recurrent unit for sequence modeling tasks, the *pyramidal recurrent unit*. We replace the standard fully-connected layers (or linear transformations) in LSTMs with a novel structure and group linear transformations, allowing us to learn contextual representations efficiently. The content of this chapter is based on [39].

Chapter 5 introduces a light-weight and deep transformer model. We introduce the `DeLighT` layer that allows us to learn deeper and wider representations efficiently. We also introduce a *block-wise scaling* mechanism that allows us to allocate parameters inside and across blocks efficiently. The content of this chapter is based on [18, 40].

Chapter 6 summarizes the dissertation along with directions for future research.

Chapter 2

LEARNING VISUAL REPRESENTATIONS USING EFFICIENT SPATIAL PYRAMIDAL UNIT

2.1 Introduction

Semantic segmentation allows the extraction of rich contextual information about the objects present in a scene, thereby enabling complete visual scene understanding in real-world applications, including home monitoring systems, self-driving cars, robotics, and augmented reality. Many of these applications run on edge or embedded devices such as smart cameras, smartphones, and the NVIDIA Jetson TX2. These devices have limited energy overhead, restrictive memory constraints, and reduced computational capabilities in comparison to standard desktops, which have large amounts of dedicated resources for both CPU and GPU. To be effective, visual understanding tasks must allow for on-line processing with small memory footprints and low power consumption.

Existing convolutional neural network (CNN)-based visual recognition systems require large amounts of computational resources, including memory and power. While they achieve high performance on high-end GPU-based machines (e.g. with NVIDIA TitanX), they are often too expensive for resource-constrained edge devices such as cell phones and embedded compute platforms. For instance, ResNet-50 [41], one of the most well-known CNN architectures for image classification, has 26 million parameters (98 MB of memory) and requires 3.8 billion full-precision floating point operations (FLOPs) to classify an RGB image with spatial dimensions of 224×224. These numbers are even higher for deeper CNNs, e.g. ResNet-101. These models quickly overtax the limited resources, including compute capabilities, memory, and battery, available on edge devices. Therefore, CNNs for real-world applications running on edge devices should be light-weight and efficient while delivering high accuracy.

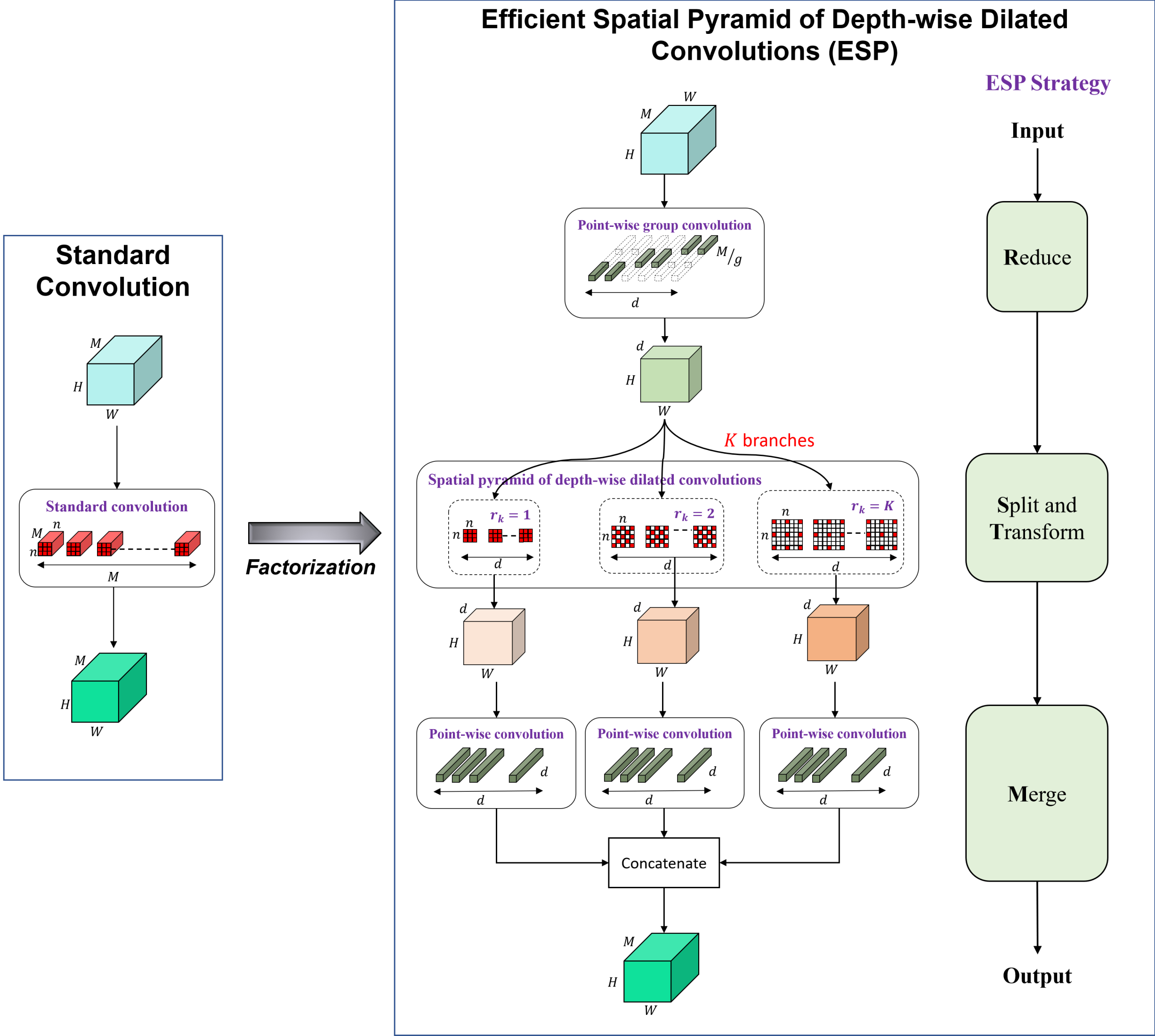


Figure 2.1: Overview of the efficient spatial pyramid (ESP) module. ESP decomposes the standard convolution using point-wise convolutions and spatial pyramid of depth-wise dilated convolutions to learn representations efficiently. See Section 2.3 for more details. Here, H, W, and M denote the height, width, and channels (depth) of the input tensor, respectively. $d = \frac{M}{K}$ denotes the number of channels after projection, where K is the number of dilated kernels in the ESP. Each dilated kernel has a size of $n \times n$ and dilation rate of r_k.

Convolution factorization has demonstrated its success in improving the efficiency of deep CNNs (e.g. GoogLeNet[42], ResNext [43], and Xception [44]). For example, with convolutional factorization, GoogLeNet [42] is able to deliver similar performance to VGG [45], yet with 10× fewer FLOPs (GoogLeNet vs. VGG: 2 vs. 20 GFLOPs for an RGB input of 224×224). Despite this, these networks are still beyond the computational budget of many edge devices. Another key challenge is the receptive field. Most of these networks uses 3×3 convolutional kernels in order to be computationally efficient. Using a larger convolutional kernel (e.g., 7×7 or 9×9) in these networks will further increase their computational cost, making them unsuitable for edge devices. To learn representations efficiently from a larger receptive field, we introduce an efficient convolutional module, ESP (efficient spatial pyramid). Based on these ESP modules, we introduce an efficient network structure, `ESPNet`, that can be easily deployed on resource-constrained edge devices. `ESPNet` is *fast*, *light-weight*, *low power*, and *low latency*, yet still preserves performance across several visual recognition tasks, including object detection and semantic segmentation.

Briefly, ESP decomposes a standard convolution into two steps: (1) ***point-wise convolutions*** and (2) a ***spatial pyramid of depth-wise dilated convolutions***, as shown in Figure 2.1. The point-wise convolutions help in reducing the computation, while the spatial pyramid of depth-wise dilated convolutions re-samples the feature maps to learn the representations from a large effective receptive field. We note that existing spatial pyramid methods, e.g., the dilated spatial pyramid module of Chen et al. [13], are computationally expensive and cannot be used at different spatial levels for learning the representations. In contrast to these methods, ESP is computationally efficient and can be used at different spatial levels of a CNN network.

Under the same constraints on memory and computation, `ESPNet` outperforms state-of-the-art networks, such as MobileNets [46, 47] and ShuffleNets [48, 49], on different visual recognition tasks while being more energy efficient. For example, our model outperforms MobileNetv2 [47] by 2% at a computational budget of 28 MFLOPs. Similarly, on the task of object detection (MS-COCO dataset), `ESPNet` outperforms YOLOv2 [50]

by 4.4% and has 6× fewer FLOPs. The source code of `ESPNet` along with pretrained models is publicly available at https://github.com/sacmehta/EdgeNets. Also, the source code of real-time semantic segmentation using `ESPNet` on iPhones is publicly available at: https://github.com/sacmehta/ESPNetv2-COREML/

2.2 Related Work

Efficient CNN architectures: Most state-of-the-art efficient networks [46, 47, 49] use depth-wise separable convolutions [44] which factorizes a standard convolution into two steps to reduce computational complexity: (1) a depth-wise convolution that performs light-weight filtering by applying a single convolutional kernel per input channel and (2) a point-wise convolution that usually expands the feature map along channels by learning linear combinations of the input channels. Another efficient form of convolution that has been used in efficient networks [48, 51] is group convolution [3], wherein input channels and convolutional kernels are factored into groups and each group is convolved independently. Despite these approaches are efficient, they learn representations from a limited receptive field (usually 3×3). This may hinder learning better representations from large structures (e.g., ducts in breast biopsy images [52, 53]). To allow the network to learn representations from a large effective receptive field, `ESPNet` uses depth-wise "dilated" separable convolutions instead of depth-wise separable convolutions.

Neural architecture search: Recently, neural architecture search-based methods have been proposed to automatically construct network architectures (e.g., [54–58]). These methods search over a large network space (e.g., MNASNet [55] searches over 8K different design choices) using a dictionary of pre-defined search space parameters, including different types of convolutional layers and kernel sizes, to find a heterogeneous network structure that satisfies optimization constraints, such as inference time. We believe that better neural architectures can be discovered by adding the ESP module in a neural search dictionary.

Quantization, compression, and distillation: Network quantization-based approaches [27–30] approximate 32-bit full precision convolution operations with fewer bits. This improves inference speed and reduces the amount of memory required for storing network weights. Network compression-based approaches [32–34, 59] improve the efficiency of a network by removing redundant weights and connections. Unlike network quantization and compression, distillation-based approaches [60–63] improve the accuracy of (usually shallow) networks by supervising the training with large pre-trained network(s). Though these approaches are effective, they treat CNNs as black boxes and optimizes only for hardware acceleration. We believe that the efficiency of `ESPNet` can be further improved using these methods.

2.3 `ESPNet`

This section elaborates on the `ESPNet` architecture in detail. We first describe *depth-wise dilated separable convolutions* that enable our network to learn representations from a large effective receptive field efficiently. We then describe the core unit of the `ESPNet` network, the ESP unit, which is built using group point-wise convolutions and depth-wise dilated separable convolutions.

2.3.1 Depth-wise dilated separable convolution

Convolution factorization is the key principle that has been used by many efficient architectures [46–49]. The basic idea is to replace the full convolutional operation with a factorized version such as depth-wise separable convolution [46] or group convolution [3]. In this section, we describe depth-wise dilated separable convolutions and compare with other similar efficient forms of convolution.

A standard convolution convolves an input $\mathbf{X} \in \mathbb{R}^{W \times H \times c}$ with convolutional kernel $\mathbf{K} \in \mathbb{R}^{n \times n \times c \times \hat{c}}$ to produce an output $\mathbf{Y} \in \mathbb{R}^{W \times H \times \hat{c}}$ by learning $n^2 c\hat{c}$ parameters from an effective receptive field of $n \times n$. Here, $n \times n$ represents the kernel size, H and W represents the height and width of tensors $\mathbf{X}$ and $\mathbf{Y}$ respectively, and c and $\hat{c}$ represents the number of

Convolution type	Parameters	Eff. receptive field
Standard	$n^2 c\hat{c}$	$n \times n$
Group	$\frac{n^2 c\hat{c}}{g}$	$n \times n$
Depth-wise separable	$n^2 c + c\hat{c}$	$n \times n$
Depth-wise dilated separable	$n^2 c + c\hat{c}$	$n_r \times n_r$

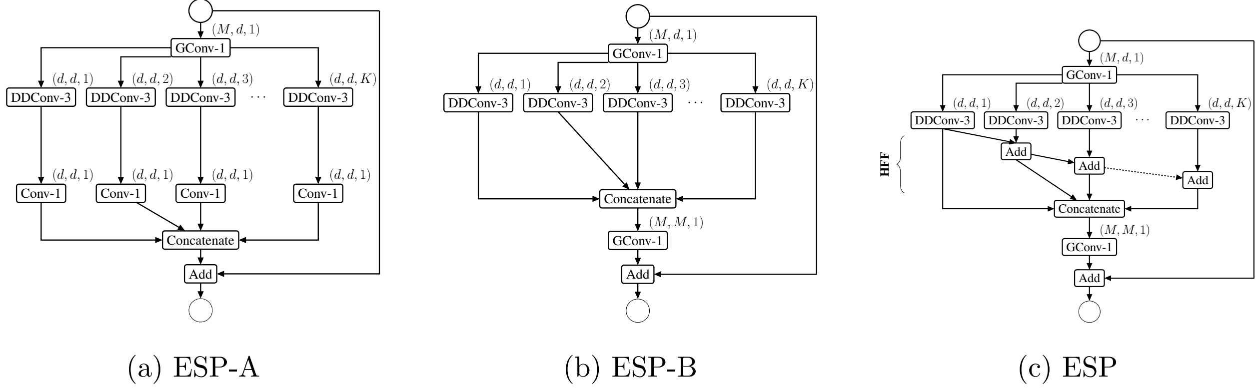
Table 2.1: Comparison between different type of convolutions. Here, $n \times n$ is the kernel size, $n_r = (n-1) \cdot r + 1$, r is the dilation rate, c and $\hat{c}$ are the input and output channels respectively, and g is the number of groups.

channels in $\mathbf{X}$ and $\mathbf{Y}$, respectively. In contrast to standard convolution, depth-wise dilated separable convolutions apply a light-weight filtering by factoring a standard convolution into two layers: 1) depth-wise dilated convolution per input channel with a dilation rate of r; enabling the convolution to learn representations from an effective receptive field of $n_r \times n_r$, where $n_r = (n-1) \cdot r + 1$ and 2) point-wise convolution to learn linear combinations of input. This factorization reduces the computational cost by a factor of $\frac{n^2 c\hat{c}}{n^2 c + c\hat{c}}$. A comparison between different types of convolutions is provided in Table 2.1. Depth-wise dilated separable convolutions are efficient and can learn representations from large effective receptive fields.

2.3.2 ESP unit

Taking advantage of depth-wise dilated separable and group point-wise convolutions, we introduce a new unit ESP, **E**fficient **S**patial **P**yramid of Depth-wise Dilated Separable Convolutions, which is specifically designed for edge devices. To factorize a standard convolution, ESP is based on a *reduce-split-transform-merge* strategy and sandwiches a spatial pyramid of depth-wise dilated convolutions between two point-wise group convolutions (Figure 2.1).

- *Reduce:* The first point-wise group convolution projects the high-dimensional feature maps (say M) into a low-dimensional space, say d. To do so, it divides the M-



(a) ESP-A (b) ESP-B (c) ESP

Figure 2.2: Block diagrams of the ESP module. ESP-A uses K point-wise convolutions to learn linear combinations of the output of depth-wise dilated convolutions (DDConv). ESP-B replaces K independent point-wise convolutions with a point-wise group convolutions (GConv) to make it more efficient. ESP adds a hierarchical feature fusion (HFF) to remove gridding artifacts in ESP-B. ESP units in (a-c) are equivalent in terms of computational complexity (parameters and FLOPs). Here, each convolutional layer (Conv-n: $n \times n$ standard convolution, GConv-n: $n \times n$ group convolution, DConv-n: $n \times n$ dilated convolution, DDConv-n: $n \times n$ depth-wise dilated convolution) is represented by (# input channels, # output channels, and dilation rate).

dimensional input into g groups and then applies a point-wise (1×1) convolution to each group independently. Each group produces a $\frac{d}{K}$-dimensional feature map; these are then concatenated to produce a d-dimensional output. Here, K is the number of dilated kernels in the ESP module.

- *Split & Transform:* The spatial pyramid of depth-wise dilated convolutions then resamples these low-dimensional feature maps using K $n \times n$ depth-wise dilated convolutional kernels simultaneously, each with a dilation rate of $r_k = k$, $k = \{1, \cdots, K\}$.

- *Merge:* The output of these branches are concatenated along the channel dimension

and then combined using the second point-wise group convolution with K groups to produce an N-dimensional output. We note that a single point-wise group convolution (Figure 2.2b) with K groups is equivalent to K point-wise convolutions that learns the linear combinations on each branch independently (Figure 2.2a). However, point-wise group convolution is more efficient in terms of implementation, because it launches one convolutional kernel rather than K point-wise convolutional kernels. Therefore, we use point-wise group convolution with K groups instead of K point-wise convolutions, as shown in Figure 2.2b. To improve the gradient flow inside the network, we set $N = M$ so that the input and output feature maps of the ESP module can be combined using a residual connection [41] .

This factorization, especially learning spatial representations in a low-dimensional space, allows the ESP unit to be efficient. The ESP module has $d(\frac{M}{g} + n^2K + N)$ parameters and its effective receptive field is $[(n-1)K+1]^2$, where $d = \frac{M}{g}$. Compared to the n^2NM parameters of the standard convolution, factorizing it using the ESP strategy reduces the total number of parameters by a factor of $\frac{n^2MN}{d(\frac{M}{g}+n^2K+N)}$, while increasing the effective receptive field by K^2. For example, an ESP module learns about 26 times fewer parameters with an effective receptive field of 9×9 than a standard convolutional kernel with an effective receptive field of 3×3 for $n = 3$, $N = M = 128$, and $K = 4$.

Hierarchical feature fusion (HFF) for de-gridding: While concatenating the outputs of dilated convolutions give the ESP module a large effective receptive field, it introduces unwanted checkerboard or gridding artifacts, as shown in Figure 2.3. To address the gridding artifact in ESP, the feature maps obtained using kernels of different dilation rates are hierarchically added before concatenating them. We call the resultant module with hierarchical feature fusion (HFF) ESP; it is shown in Figure 2.2c. HFF is simple and effective and does not increase the complexity of the ESP module, in contrast to existing methods that remove the gridding artifact by learning more parameters using dilated convolutional kernels with small dilation rates [64, 65].

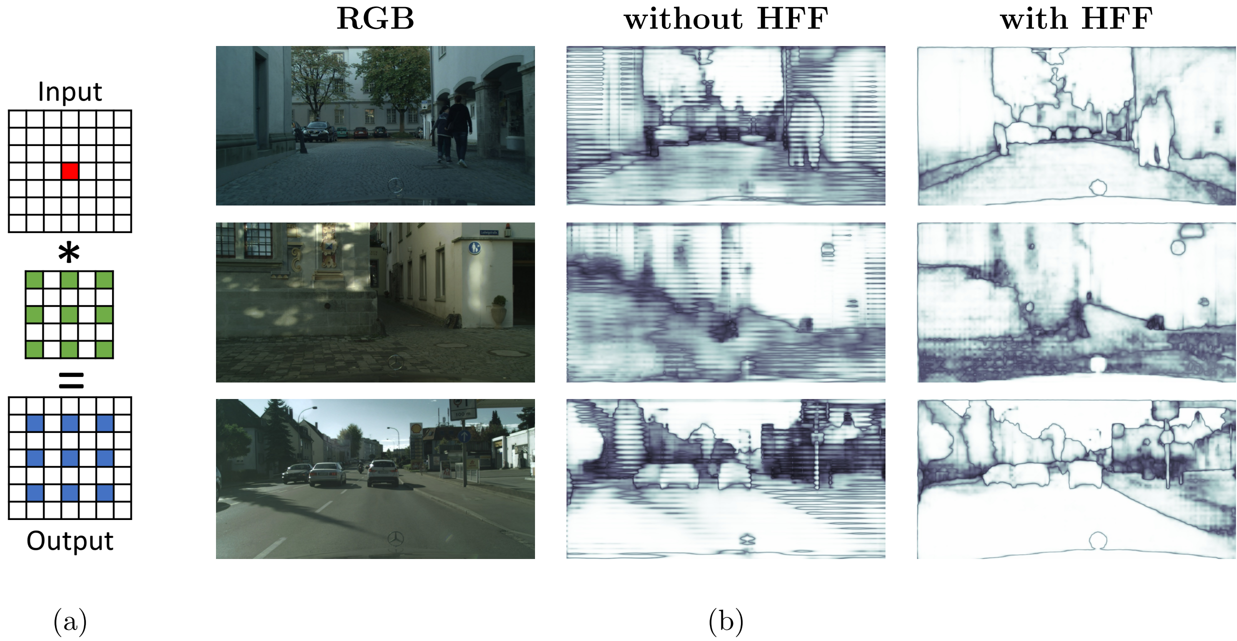


Figure 2.3: (a) An example illustrating a gridding artifact with a single active pixel (red) convolved with a 3×3 dilated convolutional kernel with dilation rate $r = 2$. (b) Visualization of feature maps of ESP modules with and without hierarchical feature fusion (HFF). HFF in ESP eliminates the gridding artifact.

Strided ESP with shortcut connection to an input image: To learn representations efficiently at multiple scales, we make the following changes to the ESP block in Figure 2.2c: 1) depth-wise dilated convolutions are replaced with their strided counterpart, 2) an average pooling operation is added instead of an identity connection, and 3) the element-wise addition operation is replaced with a concatenation operation, which helps in expanding the dimensions of feature maps efficiently [48].

Spatial information is lost during down-sampling and convolution (filtering) operations. To better encode spatial relationships and learn representations efficiently, we add an efficient long-range shortcut connection between the input image and the current down-sampling unit (shown in red in Figure 2.4). This connection first down-samples the image to the same size as

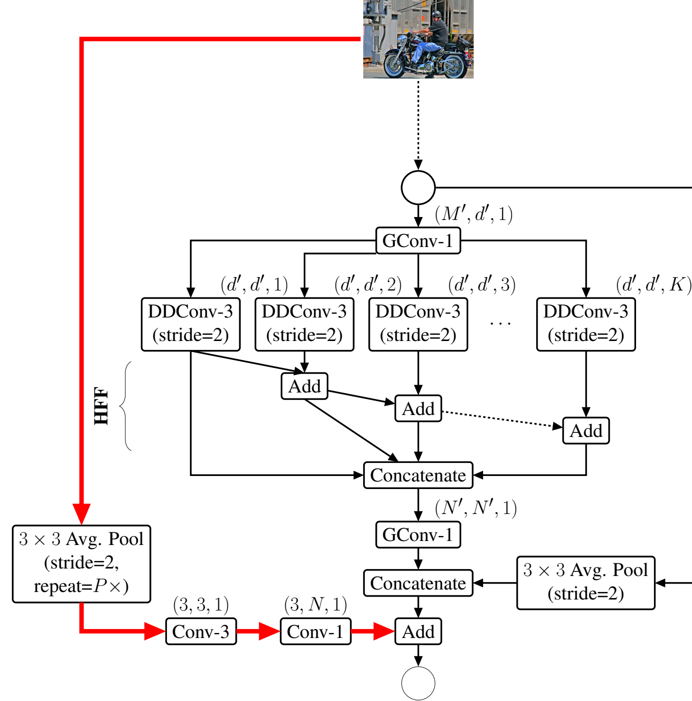


Figure 2.4: Strided ESP unit with shortcut connection to an input image (highlighted in red) for down-sampling. The average pooling operation is repeated $P\times$ to match the spatial dimensions of an input image and feature maps.

that of the feature map and then learns the representations using a stack of two convolutions. The first convolution is a standard 3×3 convolution that learns the spatial representations while the second convolution is a point-wise convolution that learns linear combinations between the inputs, and projects them to a high-dimensional space. The resultant ESP unit with long-range shortcut connection to the input is shown in Figure 2.4.

2.3.3 Network architecture

The `ESPNet` network is built using ESP units. At each spatial level, the `ESPNet` repeats the ESP units several times to increase the depth of the network. In the ESP unit (Figure 2.2c), we use batch normalization [66] and PReLU [1] after every convolutional layer, with the

Layer	Output Size	Kernel size / Stride	Repeat	Output channels for different ESPNet models					
Convolution	112 × 112	3 × 3 / 2	1	16	32	32	32	32	32
Strided ESP (Fig. 2.4)	56 × 56		1	32	64	80	96	112	128
Strided ESP (Fig. 2.4)	28 × 28		1	64	128	160	192	224	256
ESP (Fig. 2.2c)	28 × 28		3	64	128	160	192	224	256
Strided ESP (Fig. 2.4)	14 × 14		1	128	256	320	384	448	512
ESP (Fig. 2.2c)	14 × 14		7	128	256	320	384	448	512
Strided ESP (Fig. 2.4)	7 × 7		1	256	512	640	768	896	1024
ESP (Fig. 2.2c)	7 × 7		3	256	512	640	768	896	1024
Depth-wise convolution	7 × 7	3 × 3		256	512	640	768	896	1024
Group convolution	7 × 7	1 × 1		1024	1024	1024	1024	1280	1280
Global avg. pool	1 × 1	7 × 7							
Fully connected				1000	1000	1000	1000	1000	1000
Complexity				28 M	86 M	123 M	169 M	224 M	284 M
Parameters				1.24 M	1.67 M	1.97 M	2.31 M	3.03 M	3.49 M

Table 2.2: The ESPNet network at different computational complexities for classifying a 224×224 input into 1000 classes in the ImageNet dataset [5]. Network's complexity is evaluated in terms of total number of multiplication-addition operations (or FLOPs).

exception of the last group-wise convolutional layer where PReLU is applied after element-wise sum operation. To maintain the same computational complexity at each spatial-level, the feature maps are doubled after every down-sampling operation [41, 45].

In our experiments, we set the dilation rate r_k proportional to the number of branches in the ESP unit (K). The effective receptive field of the ESP unit grows linearly with K. Some of the kernels, especially at low spatial levels such as 7×7, might have a larger effective receptive field than the size of the feature map. Therefore, such kernels might not contribute to learning. In order to have meaningful kernels, we limit the effective receptive field at each spatial level l with spatial dimension $W^l \times H^l$ as: $\mathrm{n}_d^l(Z^l) = 5 + \frac{Z^l}{7}, Z^l \in \{W^l, H^l\}$ with the effective receptive field ($n_d \times n_d$) corresponding to the lowest spatial level (i.e. 7×7) as

5×5. Following [16], we set $K = 4$ in our experiments. Furthermore, in order to have a homogeneous architecture, we set the number of groups g in group point-wise convolutions equal to number of parallel branches K, i.e., $g = K$. The overall `ESPNet` architectures at different computational complexities are shown in Table 2.2.

2.4 Experiments

To showcase the power of the `ESPNet` network, we evaluate and compare the performance with state-of-the-art methods on three different tasks: (1) object classification, (2) semantic segmentation, and (3) object detection.

2.4.1 Image classification

Dataset: We evaluate the performance of the `ESPNet` on the ImageNet 1000-way classification dataset [5] that contains 1.28M images for training and 50K images for validation. We evaluate the performance of our network using the single crop top-1 classification accuracy, i.e. we compute the accuracy on the center cropped view of size 224×224.

Training: The `ESPNet` networks are trained using the PyTorch deep learning framework with CUDA 9.0 and cuDNN as the back-ends. For optimization, we use SGD [67] with *warm restarts*. At each epoch t, we compute the learning rate η_t as:

$$\eta_t = \eta_{max} - (t \mod T) \cdot \eta_{min} \tag{2.1}$$

where η_{max} and η_{min} are the ranges for the learning rate and T is the cycle length after which the learning rate will restart. Figure 2.5 visualizes the learning rate policy for three cycles. This learning rate scheme can be seen as a variant of the cosine learning policy [68], wherein the learning rate is decayed as a function of cosine before warm restart. In our experiment, we set $\eta_{min} = 0.1$, $\eta_{max} = 0.5$, and $T = 5$. We train our networks with a batch size of 512 for 300 epochs by optimizing the cross-entropy loss. For faster convergence, we decay the learning rate by a factor of two at the following epoch intervals: {50, 100, 130, 160, 190,

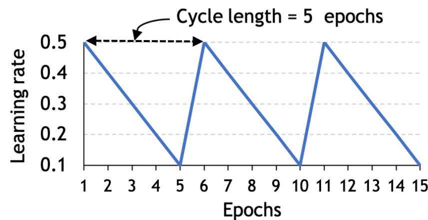
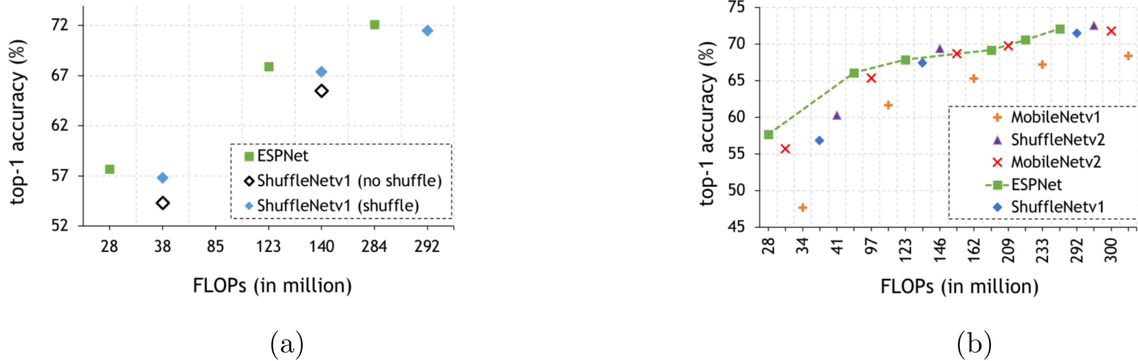


Figure 2.5: Cyclic learning rate policy with linear learning rate decay and warm restarts.

220, 250, 280}. We use a standard data augmentation strategy [41, 42] with an exception to color-based normalization. This is in contrast to recent efficient architectures that uses less scale augmentation to prevent under-fitting [48, 49]. The weights of our networks are initialized using the method described in [1].

Results: Figure 2.6 provides a performance comparison between `ESPNet` and state-of-the-art efficient networks. We observe that: (1) Like ShuffleNetv1 [48], `ESPNet` also uses group point-wise convolutions. However, `ESPNet` does not use any channel shuffle which was found to be very effective in ShuffleNetv1 (Figure 2.6a) and delivers better performance than ShuffleNetv1. (2) Compared to MobileNets, `ESPNet` delivers better performance especially under small computational budgets. With 28 million FLOPs, `ESPNet` outperforms MobileNetv1 [46] (34 million FLOPs) and MobileNetv2 [47] (30 million FLOPs) by 10% and 2% respectively. (3) `ESPNet` delivers comparable accuracy to ShuffleNetv2 [49] without any channel split, which enables ShuffleNetv2 to deliver better performance than ShuffleNetv1. We believe that such functionalities (channel split and channel shuffle) are orthogonal to `ESPNet` and can be used to further improve its efficiency and accuracy. (4) Compared to other efficient networks at a computational budget of about 300 million FLOPs, `ESPNet` delivered better performance (e.g. 1.1% more accurate than the CondenseNet [51]).

Multi-label classification: To evaluate the generalizability for transfer learning, we evaluate our model on the MS-COCO multi-object classification task [69]. The dataset consists



(a)

(b)

Figure 2.6: Performance comparison of different efficient networks on the ImageNet validation set. **(a)** `ESPNet` vs. ShuffleNetv1 [48]. **(b)** `ESPNet` vs. state-of-the-art efficient models at different network complexities (MobileNetv1 [46], MobileNetv2 [47], ShuffleNetv1 [48], and ShuffleNetv2 [49]). In (b), we count FLOPs for an input image of size 224×224.

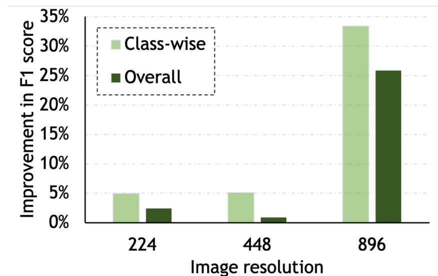
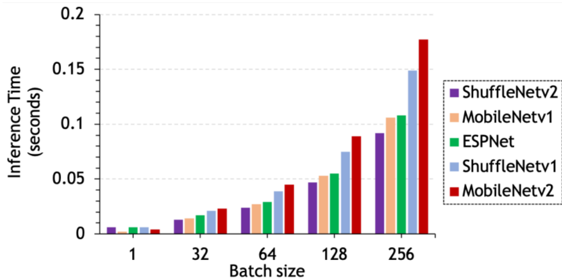
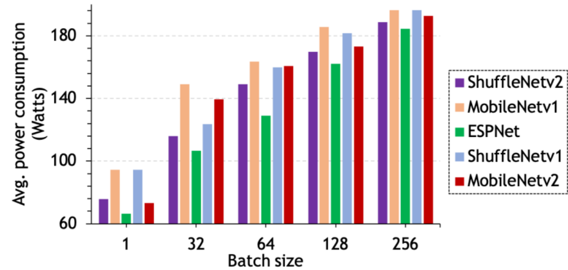


Figure 2.7: Performance improvement in F1-score of `ESPNet` over ShuffleNetv2 on MS-COCO multi-object classification task when *tested* at different image resolutions. Class-wise/overall F1-scores for `ESPNet` and ShuffleNetv2 for an input of 224×224 on the validation set are 63.41/69.23 and 60.42/67.58 respectively.

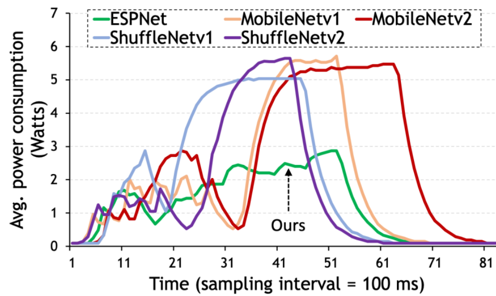
of 82,783 images, which are categorized into 80 classes with 2.9 object labels per image. Following [70], we evaluated our method on the validation set (40,504 images) using class-wise and overall F1 score. We finetune `ESPNet` (284 million FLOPs) and Shufflenetv2 [49] (299 million FLOPs) for 100 epochs using the same data augmentation and training settings as for



(a) Inference time vs. batch size (1080 Ti)



(b) Power vs. batch size (1080 Ti)



(c) Power consumption on TX2

Figure 2.8: Performance analysis of different efficient networks (computational budget is ≈ 300 MFLOPs). Inference time and power consumption are averaged over 100 iterations for a 224×224 input on a NVIDIA GTX 1080 Ti GPU and NVIDIA Jetson TX2. We do not report execution time on TX2 because there is not much substantial difference.

the ImageNet dataset, except η_{max}=0.005, η_{min}=0.001 and learning rate is decayed by two at the 50th and 80th epochs. We use binary cross entropy loss for optimization. Results are shown in Figure 2.7. `ESPNet` outperforms ShuffleNetv2 by a large margin, especially when tested at image resolution of 896×896; suggesting large effective receptive fields of the ESP unit help `ESPNet` learn better representations.

Performance analysis: An efficient network for edge devices should consume less power and have low latency with a high accuracy. We measure the efficiency of our network, `ESPNet`,

along with other state-of-the-art networks (MobileNets [46, 47] and ShuffleNets [48, 49]) on two devices: (1) a high-end graphics card (NVIDIA GTX 1080 Ti) and (2) an embedded device (NVIDIA Jetson TX2). For a fair comparison, we use PyTorch as a deep-learning framework. Figure 2.8 compares the inference time and power consumption, while networks complexity along with their accuracy are compared in Figure 2.6. The inference speed of `ESPNet` is slightly lower than the fastest network (ShuffleNetv2 [49]) on both devices, however, it is much more power efficient while delivering similar accuracy on the ImageNet dataset. This suggests that `ESPNet` network has a good trade-off between accuracy, power consumption, and latency; a much desirable property for any efficient network.

2.4.2 Semantic segmentation

Dataset: We evaluate the performance of the `ESPNet` on two datasets: (1) the Cityscapes dataset [8] and (2) the PASCAL VOC 2012 dataset [71]. The Cityscapes dataset consists of 5,000 finely annotated images (training/validation/test: 2,975/500/1,525). The task is to segment an image into 19 classes that belong to 7 categories. The PASCAL VOC 2012 dataset provide annotations for 20 foreground objects and has 1.4K training, 1.4K validation, and 1.4K test images. Following a standard convention [13, 14], we also use additional images from [69, 72] for training our networks.

Training: For segmentation, we use an encoder-decoder network [16]. We replace the encoder with the `ESPNet` and train the resultant network in two stages. In the first stage, we use a smaller image resolution for training (256×256 for the PASCAL VOC 2012 dataset and 512×256 for the Cityscapes dataset). We train `ESPNet` for 100 epochs using SGD with an initial learning rate of 0.007. In the second stage, we increase the image resolution to (384×384 for the PASCAL VOC 2012 and to 1024×512 for the Cityscapes dataset) and then finetune the `ESPNet` from first stage for 100 epochs using SGD with initial learning rate of 0.003. For both these stages, we use cyclic learning schedule discussed in Section 2.4.1. For the first 50 epochs we use a cycle length of 5, while for the remaining epochs we use

a cycle length of 50, i.e., for the last 50 epochs, we decay the learning rate linearly. We evaluate the accuracy in terms of mean Intersection over Union (mIOU) on the private test set using *online evaluation server*. For evaluation, we up-sample segmented masks to the same size as of the input image using nearest neighbour interpolation.

Results: Figure 2.9 compares the performance of ESPNet with state-of-the-methods on both the Cityscapes and the PASCAL VOC 2012 dataset. We can see that ESPNet delivers a competitive performance to existing methods while being very efficient. Under similar computational constraints, ESPNet outperforms existing methods like ENet [12] and RTSeg [75] by large margin. Notably, ESPNet is 2-3% less accurate than other efficient networks such as ICNet [74], ERFNet [15], and ContextNet [73], but has $9-12\times$ fewer FLOPs. Besides this, qualitative results in Figure 2.10 further demonstrate ESPNet's segmentation ability in

Network	FLOPs	mIOU
SegNet [10]	82 B	57.0
ContextNet [73]	33 B	66.1
ICNet [74]	31 B	69.5
ERFNet [15]	26 B	69.7
MobileNetv2** [47]	21 B	**70.7**
RTSeg w/ MobileNet [75]	13.8 B	61.5
RTSeg w/ ShuffleNet [75]	6.2 B	58.3
ENet [12]	3.8 B	58.3
ESPNet-val (Ours)	**2.7 B**	66.4
ESPNet-test (Ours)	**2.7 B**	66.2

(a) Cityscapes

Network	FLOPs	mIOU
FCN-8s [9]	181 B	62.2
DeepLabv3 [76]	81 B	**80.5**
SegNet [10]	31 B	59.1
MobileNetv1 [46]	14 B	75.3
MobileNetv2 [47]	5.8 B	75.7
ESPNet - val	0.76 B	67.0
ESPNet - test	**0.76 B**	68.0

(b) PASCAL VOC 2012

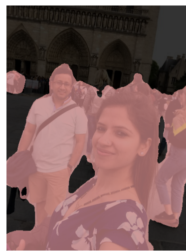
Figure 2.9: Semantic segmentation results on (a) the Cityscapes dataset and (b) the PASCAL VOC 2012 dataset. For a fair comparison, we report FLOPs at the same image resolution which is used for computing the accuracy. **MobileNetv2 [47] uses additional data from [69].

B-ground	Aero plane	Bicycle	Bird	Boat	Bottle	Bus
Car	Cat	Chair	Cow	Dining-Table	Dog	Horse
Motorbike	Person	Potted-Plant	Sheep	Sofa	Train	TV/Monitor

PASCAL VOC Colormap

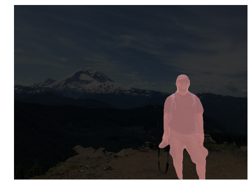
(a)

(b)



(c)

(d)



(e)

(f)

(g)

Figure 2.10: `ESPNet`'s qualitative performance on *unseen images* on the task of semantic segmentation. Each subfigure is organized as: on the **left**, we have an input image while on the **right**, we have segmentation mask overlayed on the input image.

diverse settings, including different backgrounds and image sizes.

2.4.3 Object detection

Dataset and training details: For object detection, we replace VGG [45] with `ESPNet` in single shot object detector [77]. We evaluate the performance on two datasets: (1) the PASCAL VOC 2007 and (2) the MS-COCO dataset. For the PASCAL VOC 2007 dataset, we also use additional images from the PASCAL VOC 2012 dataset. We evaluate the performance in terms of mean Average Precision (mAP). For the COCO dataset, we report mAP @ IoU of 0.50:0.95. For training, we use the same learning policy as in Section 2.4.2.

Results: Table 2.3 compares the performance of `ESPNet` with existing methods. `ESPNet` provides a good trade-off between accuracy and efficiency. Notably, `ESPNet` delivers the same performance as YOLOv2, but has $25\times$ fewer FLOPs. Compared to SSD, `ESPNet` delivers a very competitive performance while being very efficient.

2.5 Ablation Studies on the ImageNet Dataset

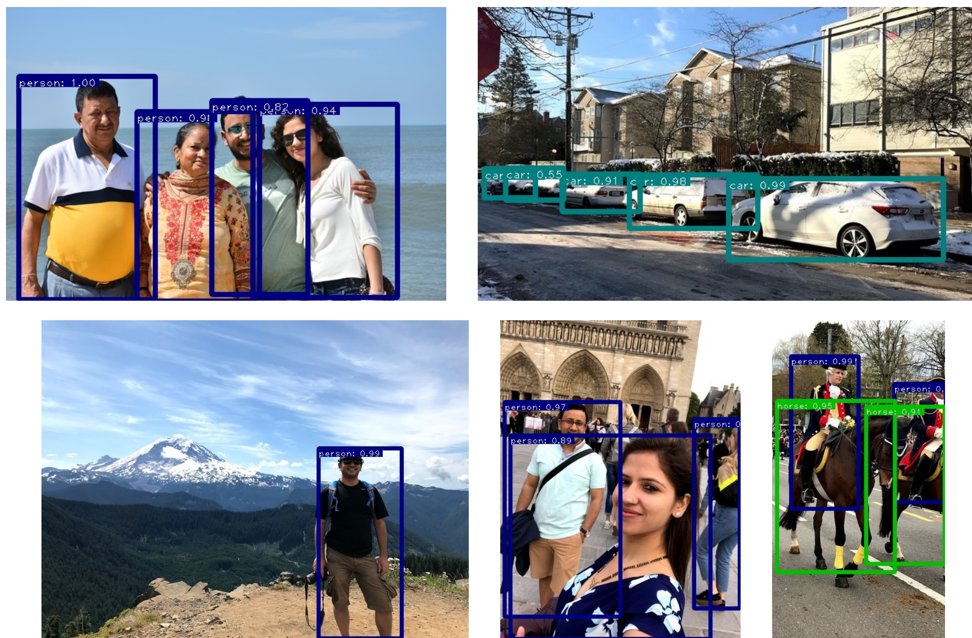
This section elaborate on various choices that helped make `ESPNet` efficient and accurate.

Impact of different convolutions: Table 2.4 summarizes the impact of different convolutions. Clearly, depth-wise dilated separable convolutions are more effective than dilated and depth-wise convolutions.

Impact of hierarchical feature fusion (HFF): `ESPNet`'s performance with and without HFF is shown in Table 2.5 (see R1 and R2). HFF improves classification performance by about 1.5% while having no impact on the network's complexity. This suggests that the role of HFF is dual purpose. First, it removes gridding artifacts caused by dilated convolutions (as noted by [16]). Second, it enables sharing of information between different branches of the ESP unit (see Figure 2.2c) that allows it to learn rich and strong representations.

Network	VOC07		MS-COCO	
	FLOPs	**mAP**	**FLOPs**	**mAP**
SSD-512 [77]	90.2 B	**74.9**	99.5 B	**26.8**
SSD-300 [77]	31.3 B	72.4	35.2 B	23.2
YOLOv2 [50]	6.8 B	69.0	17.5 B	21.6
MobileNetv1-320 [46]	–	–	1.3 B	22.2
MobileNetv2-320 [47]	–	–	0.8 B	22.1
`ESPNet`-512 (Ours)	2.5 B	68.2	2.8 B	26.0
`ESPNet`-384 (Ours)	1.4 B	65.6	1.6 B	23.2
`ESPNet`-256 (Ours)	**0.6 B**	63.8	**0.7 B**	21.9

(a) Comparison with existing methods on the PASCAL VOC 2007 and the MS-COCO dataset.



(b) Qualitative results of `ESPNet`-256 *in the wild.*

Table 2.3: `ESPNet`'s quantitative and qualitative performance on the task of object detection.

Convolution	FLOPs	Top-1
Dilated (standard)	478 M	69.2
Depth-wise	123 M	66.5
Depth-wise dilated	123 M	67.9

Table 2.4: Effect of different convolutions on the performance of `ESPNet`.

	Network properties		Learning schedule		Performance		
	HFF	LRSC	Fixed	Cyclic	# Params	FLOPs	Top-1
R1	✗	✗	✓	✗	1.66 M	84 M	58.94
R2	✓	✗	✓	✗	1.66 M	84 M	60.07
R3	✓	✓	✓	✗	1.67 M	86 M	61.20
R4	✓	✓	✗	✓	1.67 M	86 M	62.17
R5†	✓	✓	✗	✓	1.67 M	86 M	66.10

Table 2.5: Performance of `ESPNet` under different settings. Here, HFF represents hierarchical feature fusion and LRSC represents long-range shortcut connection with an input image. We train `ESPNet` for 90 epochs and decay the learning rate by 10 after every 30 epochs. For fixed learning rate schedule, we initialize learning rate with 0.1 while for cyclic, we set η_{min} and η_{max} to 0.1 and 0.5 in Eq. 2.1 respectively. Here, † represents that the learning rate schedule is the same as in Section 2.4.1.

Impact of long-range shortcut connections with the input: To see the influence of shortcut connections with the input image, we train the `ESPNet` network with and without shortcut connection. Results are shown in Table 2.5 (see R2 and R3). Clearly, these connections are effective and efficient, improving the performance by about 1% with a little (or negligible) impact on network's complexity.

Fixed vs cyclic learning schedule: A comparison between fixed and cyclic learning schedule is shown in Table 2.5 (R3 and R4). With a cyclic learning schedule, the `ESPNet` network achieves about 1% higher top-1 validation accuracy on the ImageNet dataset; suggesting that the cyclic learning schedule allows `ESPNet` find a better local minima than the fixed learning schedule. Further, when we trained `ESPNet` network for longer (300 epochs) using the learning schedule outlined in Section 2.4.1, performance improved by about 4% (see R4 and R5 in Table 2.5).

2.6 Summary

We introduce a light-weight and power efficient network, `ESPNet`, which better encode the spatial information in images by learning representations from a large effective receptive field. Our network is a general purpose network with good generalization abilities and can be used across a wide range of tasks and devices, including smartphones[1]. Our network delivered state-of-the-art performance across different tasks such as object classification, detection, and segmentation while being more power efficient than competing networks.

[1] `ESPNet` for real-time semantic segmentation on smartphones: https://github.com/sacmehta/ESPNetv2-COREML/

Chapter 3

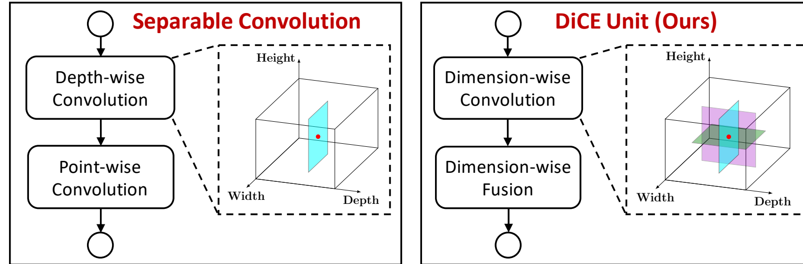
DIMENSION-WISE CONVOLUTIONS FOR EFFICIENT NETWORKS

3.1 Introduction

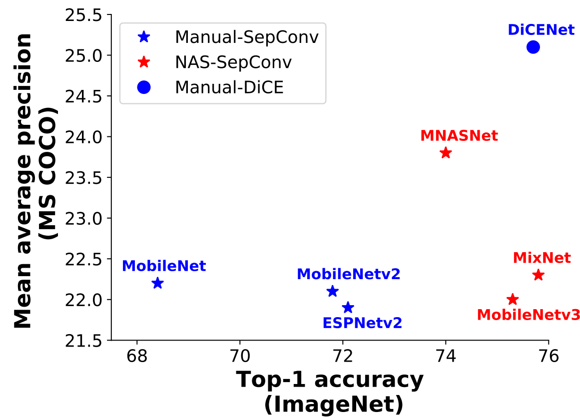
The basic building layer at the heart of convolutional neural networks (CNNs) is a convolutional layer that encodes spatial and channel-wise information simultaneously [3, 41, 45]. Learning representations using this layer is computationally expensive. Improving the efficiency of CNN architectures as well as convolutional layers is an active area of research. Most recent attempts have focused on improving the efficiency of CNN architectures using compression- and quantization-based methods (e.g. [33, 34]). Recently, several factorization-based methods have been proposed to improve the efficiency of standard convolutional layers (e.g. [16, 46]). In particular, depth-wise separable convolutions [44, 46] have gained a lot of attention (see Figure 3.1a). These convolutions have been used in several efficient state-of-the-art architectures, including neural search-based architectures [37, 47, 49, 55, 56][1].

Separable convolutions factorize the standard convolutional layer in two steps: (1) a light-weight convolutional filter is applied to each input channel using depth-wise convolutions [44] to learn spatial representations, and (2) a point-wise (1×1) convolution is applied to learn linear combinations between spatial representations. Though depth-wise convolutions are efficient, they do not encode channel-wise relationships. Therefore, separable convolutions rely on point-wise convolutions to encode channel-wise relationships. This puts a significant computational load on point-wise convolutions and makes them a computational bottleneck. For example, point-wise convolutions account for about 90% of total operations in ShuffleNetv2 [49] and MobileNetv2 [47].

[1]Related work is discussed in detail in Chapter 2.



(a) Block of separable convolutions [46] and the `DiCE` unit. Convolutional kernels are highlighted in color (**depth-**, **width-**, and **height-**wise).



(b) The network with the `DiCE` unit (`DiCENet`) has better **task-level generalization** properties than networks with separable convolutions (MobileNet [46], MobileNetv2 [47], MobileNetv3 [58], MixNet [78], MNASNet [55]). Here, Manual-SepConv and NAS-SepConv represent the models that use separable convolution without and with neural architecture search (NAS), respectively. Manual-DiCE represents `DiCENet` without NAS. On the ImageNet dataset, these networks have about 200-300 million floating point operations. See Section 3.5 for more details.

Figure 3.1: **Separable convolutions vs. the `DiCE` unit.**

In this chapter, we introduce `DiCENet`, **Di**mension-wise **C**onvolutions for **E**fficient **Net**works, which encodes spatial and channel-wise representations efficiently. Our main contribution is the novel and generic module, the `DiCE` unit (Figure 3.1a), that is built using `Dimension-`

wise Convolutions (DimConv) and Dimension-wise Fusion (DimFuse). The DimConv applies a light-weight convolutional filter across "each dimension" of the input tensor to learn local dimension-wise representations while DimFuse efficiently combines these dimension-wise representations to incorporate global information.

With DimConv and DimFuse, we build an efficient convolutional unit, the DiCE unit, that can be easily integrated into existing or new CNN architectures to improve their performance and efficiency. Figure 3.1b shows that the DiCE unit is effective in comparison to widely-used separable convolutions. Compared to state-of-the-art manually designed networks (e.g., MobileNet [46], MobileNetv2 [47], and ShuffleNetv2 [49]), DiCENet delivers significantly better performance. For example, for a network with about 300 million floating point operations (FLOPs), DiCENet is about 4% more accurate than MobileNetv2. Importantly, DiCENet, a manually designed network, delivers similar or better performance than neural architecture search (NAS) based methods. For example, DiCENet is 1.7% more accurate than MNASNet for a network with about 300 MFLOPs.

We empirically demonstrate in Section 3.5 and Section 3.6 that the DiCENet network, built by stacking DiCE units, achieves significant improvements on standard benchmarks across different tasks over existing networks. Compared to existing efficient networks, DiCeNet generalizes better to tasks (e.g., object detection) that are often used in resource-constrained devices. For instance, DiCENet achieves about 3% higher mean average precision than MobileNetv3 [58] and MixNet [78] on the MS-COCO object detection task with SSD [77] as a detection pipeline (Figure 3.1b). Our source code is available at https://github.com/sacmehta/EdgeNets/.

3.2 DiCENet

Standard convolutions encode spatial and channel-wise information *simultaneously*, but they are computationally expensive. To improve the efficiency of standard convolutions, separable (or depth-wise separable) convolutions are introduced [46], where spatial and channel-wise information is encoded *separately* using depth-wise and point-wise convolutions, respectively.

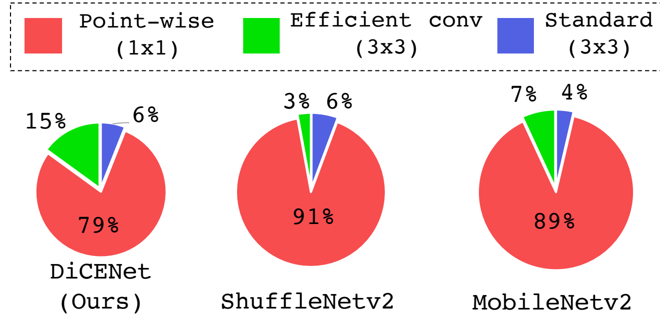


Figure 3.2: **Convolution-wise distribution of FLOPs** for different networks with similar accuracy. With dimension-wise convolutions in `DiCENet`, we are able to spend more compute on learning spatial representations as compared to other networks (MobileNetv2 and ShuffleNetv2). As a result, `DiCENet` achieves similar or better performance than existing methods with fewer operations. The size of pie charts is scaled with respect to MobileNetv2's FLOPs (300 million). In `DiCENet`, efficient convolutions correspond to dimension-wise convolutions, while in other networks they correspond to depth-wise convolutions.

Though this factorization is effective, it puts a significant computational load on point-wise convolutions and makes them a computational bottleneck, as shown in Figure 3.2.

To encode spatial and channel-wise information efficiently, we introduce the `DiCE` unit that is shown in Figure 3.3. The `DiCE` unit factorizes standard convolution using `Dim`ension-wise `Conv`olution (`DimConv`, Section 3.2.1) and `Dim`ension-wise `Fus`ion (`DimFuse`, Section 3.2.2). `DimConv` applies a light-weight filtering across each dimension of the input tensor to learn local dimension-wise representations. `DimFuse` efficiently combines these representations from different dimensions and incorporates global information. The ability to encode local spatial and channel-wise information from all dimensions using `DimConv` enables the `DiCE` unit to use `DimFuse` instead of computationally expensive point-wise convolutions.

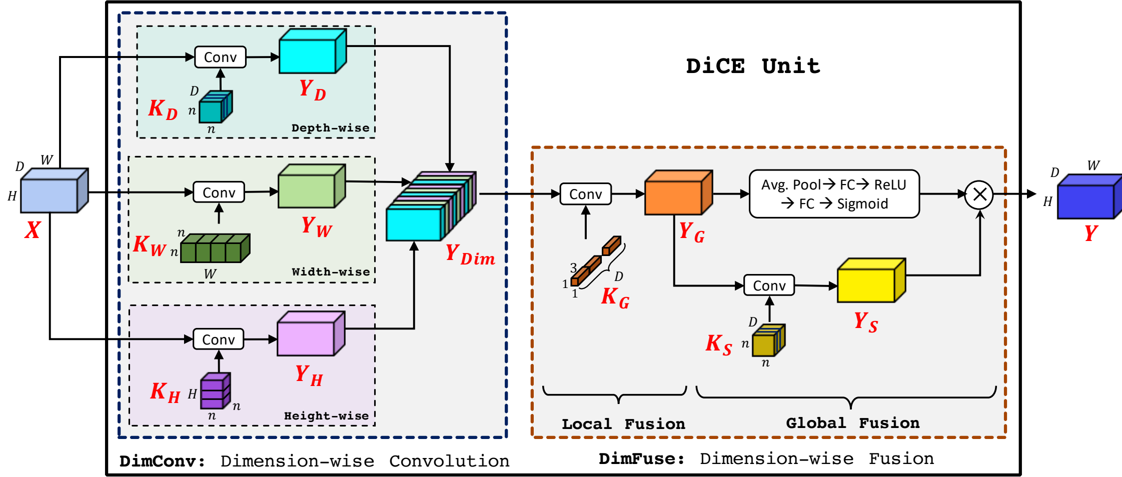


Figure 3.3: The DiCE unit efficiently encodes the spatial and channel-wise information in the input tensor $\mathbf{X}$ using dimension-wise convolutions (DimConv) and dimension-wise fusion (DimFuse) to produce an output tensor $\mathbf{Y}$. For simplicity, we show the kernels corresponding to each dimension independently. However, in practice, these three kernels are executed simultaneously, leading to faster run-time. See Sections 3.2.4 and 3.5 for more details.

3.2.1 Dimension-wise Convolution (DimConv)

We use dimension-wise convolutions (DimConv) to encode depth-, width-, and height-wise information *independently*. To achieve this, DimConv extends depth-wise convolutions to *all dimensions* of the input tensor $\mathbf{X} \in \mathbb{R}^{D \times H \times W}$, where W, H, and D correspond to width, height, and depth of $\mathbf{X}$. As illustrated in Figure 3.3, DimConv has three branches, one branch per dimension. These branches apply D depth-wise convolutional kernels $\mathbf{k}_D \in \mathbb{R}^{1 \times n \times n}$ along the depth, W width-wise convolutional kernels $\mathbf{k}_W \in \mathbb{R}^{n \times n \times 1}$ along the width, and H height-wise convolutional kernels $\mathbf{k}_H \in \mathbb{R}^{n \times 1 \times n}$ kernels along the height to produce outputs $\mathbf{Y_D}$, $\mathbf{Y_W}$, and $\mathbf{Y_H} \in \mathbb{R}^{D \times H \times W}$ that encode information from all dimensions of the input tensor. The outputs of these independent branches are concatenated along the depth dimension, such that the first spatial planes of tensors $\mathbf{Y_D}$, $\mathbf{Y_W}$, and $\mathbf{Y_H}$ are put together, the second planes are put together, and so on, to produce the output $\mathbf{Y_{Dim}} = \{\mathbf{Y_D}, \mathbf{Y_W}, \mathbf{Y_H}\} \in \mathbb{R}^{3D \times H \times W}$.

3.2.2 Dimension-wise Fusion (`DimFuse`)

The dimension-wise convolutions encode local information from different dimensions of the input tensor, but do not capture global information. A standard approach to combine features globally in CNNs is to use a point-wise convolution [41, 46]. A point-wise convolutional layer applies D point-wise kernels $\mathbf{k}_p \in \mathbb{R}^{3D\times 1\times 1}$ and performs $3D^2HW$ operations to combine dimension-wise representations of $\mathbf{Y_{Dim}} \in \mathbb{R}^{3D\times H\times W}$ and produce an output $\mathbf{Y} \in \mathbb{R}^{D\times H\times W}$. This is computationally expensive. Given the ability of `DimConv` to encode spatial and channel-wise information (though independently), we introduce a fusion module, Dimension-wise fusion (`DimFuse`), that allows us to combine representations of $\mathbf{Y_{Dim}}$ efficiently. As illustrated in Figure 3.3, `DimFuse` factorizes the point-wise convolution in two steps: (1) local fusion and (2) global fusion.

$\mathbf{Y_{Dim}} \in \mathbb{R}^{3D\times H\times W}$ concatenates spatial planes along depth dimension from $\mathbf{Y_D}$, $\mathbf{Y_W}$, and $\mathbf{Y_H}$ (see Figure 3.3). Therefore, $\mathbf{Y_{Dim}}$ can be viewed as a tensor with D groups, each group with three spatial planes (one from each dimension). `DimFuse` uses a group point-wise convolutional layer to combine dimension-wise information contained in $\mathbf{Y_{Dim}}$. In particular, this group convolutional layer applies D point-wise convolutional kernels $\mathbf{k}_G \in \mathbb{R}^{3\times 1\times 1}$ to $\mathbf{Y_{Dim}}$ and produces an output $\mathbf{Y_G} \in \mathbb{R}^{D\times H\times W}$. Since D kernels in $\mathbf{k}_G$ operate independently on D groups in $\mathbf{Y_{Dim}}$, we call this *local* fusion operation.

To efficiently encode the global information in $\mathbf{Y_G}$, `DimFuse` learns spatial and channel-wise representations independently and then propagates channel-wise encodings to spatial encodings using an element-wise multiplication. Specifically, `DimFuse` encodes spatial representations by applying D depth-wise convolutional kernels $\mathbf{k}_S \in \mathbb{R}^{1\times n\times n}$ to $\mathbf{Y_G}$ to produce an output $\mathbf{Y}_S$.[2] Motivated by the Squeeze-Excitation (SE) unit [79], we squeeze spatial dimensions of $\mathbf{Y_G}$ and encode channel-wise representations using two fully connected (FC) layers. The first FC layer reduces the input dimension from D to $\frac{D}{4}$ while the second FC layer expands dimensionality from $\frac{D}{4}$ to D. To allow these fully connected layers to learn non-linear

[2]When the depth of $\mathbf{Y_{Dim}}$ is different from $\mathbf{Y}$, then $\mathbf{k}_S$ is a group convolution, where number of groups is the greatest common divisor between the depth of $\mathbf{Y_{Dim}}$ and $\mathbf{Y}$.

representations, a ReLU activation is added in between these two layers. Similar to the SE unit, spatial representations $\mathbf{Y_G}$ are then scaled using these channel-wise representations to produce output $\mathbf{Y}$.

The computational cost of `DimFuse` is $HWD(3+n^2+D)$. Effectively, `DimFuse` reduces the computational cost of point-wise convolutions by a factor of $\frac{3D}{3+n^2+D}$. `DimFuse` uses $n = 3$, so the computational cost is approximately $3\times$ smaller than that of the point-wise convolution.

3.2.3 `DiCE` Unit for Arbitrary Sized Inputs

The `DiCE` unit stacks `DimConv` and `DimFuse` to encode spatial and channel-wise information in the input tensor efficiently. However, the two kernels (i.e., $\mathbf{k}_H$ and $\mathbf{k}_W$) in the `DimConv` unit correspond to spatial dimensions of the input tensor. This may pose a challenge when spatial dimensions of the input tensor are different from the ones with which the network is trained. To make `DiCE` units invariant to the spatial dimensions of the input tensor, we dynamically scale (either up-sample or down-sample) the height or width dimension of the input tensor to the height or width of the input tensor used in the pretrained network. The resultant tensors are then scaled (either down-sampled or up-sampled) back to their original size before being fed to `DimFuse`; this makes the `DiCE` unit invariant to input tensor size. Figure 3.4 sketches the `DiCE` unit with dynamic scaling. Results in Section 3.6, especially object detection and

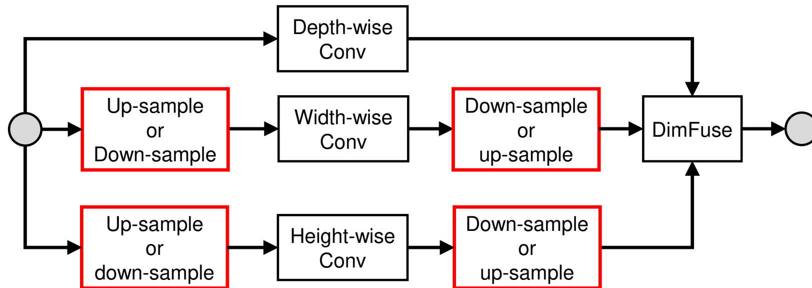


Figure 3.4: `DiCE` unit for arbitrary sized input.

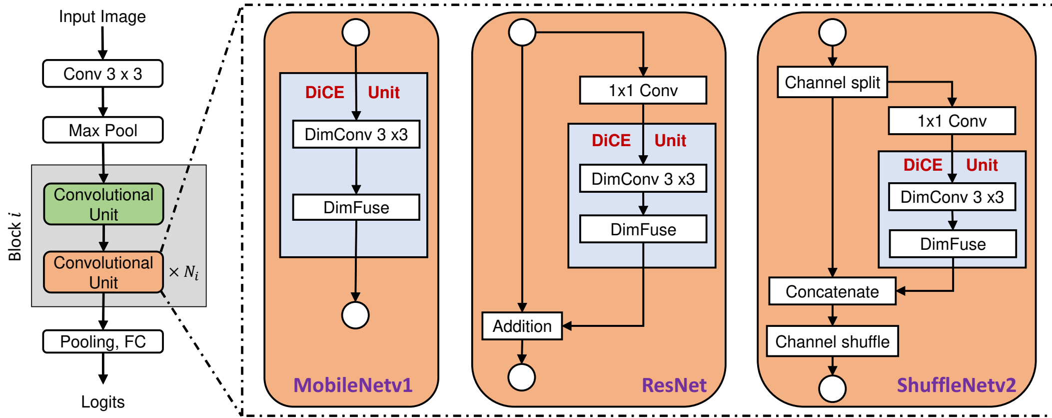


Figure 3.5: `DiCE` **unit in different architecture designs** for the task of image classification on the ImageNet dataset. Green and oranage boxes are with and without stride, respectively. Here, $N_i = \{3, 7, 3\}$ for $i = \{1, 2, 3\}$. See Appendix A.1 for detailed architecture specification.

semantic segmentation, show that the `DiCE` unit can handle arbitrary-sized inputs.

3.2.4 `DiCENet` Architecture

`DiCE` units are generic and can be easily integrated into any existing network. Figure 3.5 visualizes the `DiCE` unit with different architectures: (1) **MobileNet** [46] stacks separable convolutions (depth-wise convolution followed by point-wise convolution) to learn representations. (2) **ResNet** [41] introduces the bottleneck unit with residual connections to train very deep networks. The bottleneck unit is a stack of three convolutional layers: one 3×3 depth-wise convolutional layer[3] surrounded by two point-wise convolutions. This block can be viewed as a point-wise convolution followed by separable convolution. (3) **ShuffleNetv2** [49] is a state-of-the-art efficient network that outperforms other efficient networks, including MobileNetv2 [47]. ShuffleNetv2's unit stacks a point-wise convolution and separable convolution. It also uses channel split and shuffle to promote feature reuse.

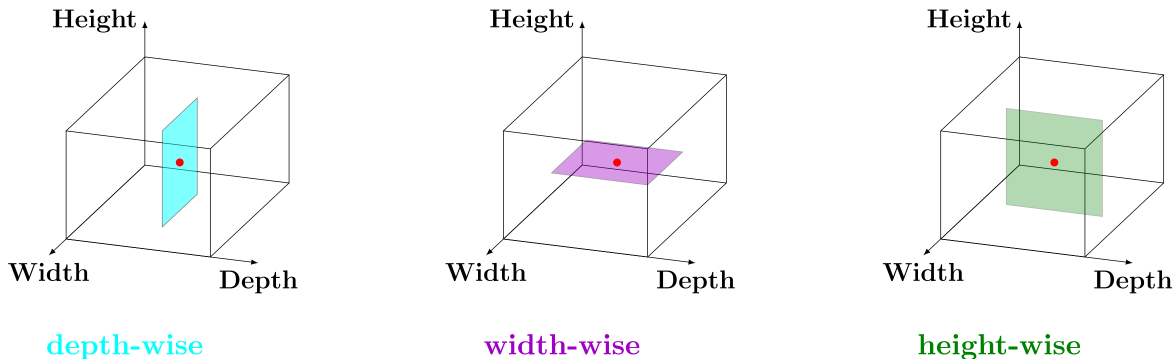
[3]Our focus is on efficient network. Therefore, we replace standard convolutional layer with depth-wise convolutional layer.

To illustrate the performance benefits and generic nature of the DiCE unit over separable convolutions, we replace separable convolutions with the DiCE unit in these architectures. Our empirical results in Section 3.4 shows that the DiCE unit with ShuffleNetv2's architecture delivers the best performance. Therefore, we choose the ShuffleNetv2 [49] architecture and call the resultant network DiCENet (ShuffleNetv2 with the DiCE unit).

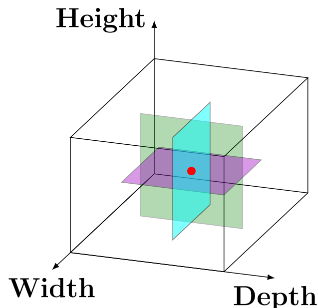
CUDA Implementation: DimConv applies D, W, and H depth-wise, width-wise and height-wise convolutional kernels to the input tensor $\mathbf{X}$ to aggregate information from different dimensions of the tensor, respectively. A standard solution would be to apply each kernel independently to the tensor and then concatenate their results, as shown in Figure 3.6a. Another solution would be to apply all kernels simultaneously, as shown in Figure 3.6b. Compared to three CUDA kernel calls in the former solution, the later one requires one kernel call, thus reducing the kernel launch time. Also, each thread in the CUDA kernel processes $3n^2$ elements compared to n^2 elements in the former solution for $n \times n$ convolutional kernels. This maximizes the work done per thread and improves speed. Our results in Section 3.5 shows that DiCENet is accurate and fast compared to state-of-the-art methods, including neural search-based methods.

3.3 Experimental Set-up

Following most architecture designs (e.g. [37, 41, 46]), we evaluate the generic nature of the DiCE unit on the ImageNet dataset [5] in Section 3.4. We integrate the DiCE unit in different image classification architectures (Figure 3.5) and study the impact on efficiency and accuracy. We also study the importance of the two main components of the DiCE unit, i.e. DimConv and DimFuse, and show that DiCE units are more effective than separable convolutions [46]. In Section 3.5, we evaluate the image classification performance of DiCENet on the ImageNet dataset and show that DiCENet delivers similar or better performance than state-of-the-art efficient networks, including neural search-based methods. In Section 3.6, we evaluate task-level generalization ability of DiCENet on three different visual recognition



(a) **Unoptimized:** Each kernel is applied to a pixel (represented by red dot) independently.



(b) **Optimized:** All kernels (**depth-**, **width-**, and **height-**wise) are applied to a pixel (represented by red dot) simultaneously, allowing us to aggregate the information from tensors efficiently.

Figure 3.6: **Implementation of dimension-wise convolution (`DimConv`)**

tasks, i.e. object detection, semantic segmentation, and multi-object classification, that are often used in resource-constrained devices. We demonstrate that `DiCENet` generalizes better than existing efficient networks that are built using separable convolutions.

Datasets: We use the following datasets in our experiments.

- ***Image classification:*** For *single label* image classification, we use ImageNet-1K classification dataset [5]. This dataset consists of 1.28M training and 50K validation images. All networks on this dataset are trained from scratch. For *multi-label* clas-

sification, we use the MS-COCO dataset [69] that has 2.9 labels (on an average) per image. We use the same training and validation splits as in [37].

- ***Object detection:*** We use the MS-COCO [69] and PASCAL VOC [71] datasets for evaluating on the task of object detection. Following a standard convention for training on the PASCAL VOC 2007 dataset, we augment it with the PASCAL VOC 2012 and the PASCAL VOC 2007 *trainval* sets for training and evaluate the performance on the PASCAL VOC 2007 *test* set.

- ***Semantic segmentation:*** We use the PASCAL VOC 2012 [71] dataset for this task. Following a standard convention, we use additional images for training from [72] and [69]. Similar to MobileNetv2 [47], we evaluate the performance on the validation set.

Efficiency metric: We measure efficiency in terms of the number of floating point operations (FLOPs) and inference time. We use PyTorch for training our networks.

3.4 Evaluating `DiCE` Unit on ImageNet

We first evaluate the two important properties of the `DiCE` unit, generic and efficiency, in Section 3.4.1. To evaluate this, we replace separable convolutions [46] in different architectures with the `DiCE` unit (Figure 3.5). We then study the importance of each component of the `DiCE` unit, `DimConv` and `DimFuse`, in Section 3.4.2. Recent studies (e.g., MobileNetv3 [58] and MNASNet [55]) use several different methods, such as exponential moving average (EMA) and large batch sizes, to improve the performance. In Section 3.4.3, we study the effect of these methods on the performance of `DiCENet`.

In these experiments, we use the ImageNet dataset [5]. In Section 3.4.1 and Section 3.4.2, we follow the experimental setup of ESPNet [37] (Chapter 2) and ShuffleNetv2 [49] (fewer training epochs with smaller batch size), while in Section 3.4.3 we follow experimental set-up similar to MobileNets [47, 58] (longer training with larger batch size).

FLOP Range (in millions)	Separable conv (SC) Top-1	FLOPs	DiCE unit (DU) Top-1	FLOPs	Absolute difference (DU - SC) Top-1	FLOPs
MobileNet [46]						
25-60	49.80	41 M	**52.55**	**29** M	**+2.75**	**-12 M**
120-170	65.30	**162** M	**69.05**	167 M	**+3.75**	+5 M
270-320	68.40	317 M	**70.83**	**277** M	**+2.43**	**-40 M**
ResNet [41]						
25-60	59.30	59 M	**61.35**	**52** M	**+2.05**	**-7 M**
120-170	67.80	142 M	**67.90**	**122** M	**+0.10**	**-20 M**
270-320	70.67	302 M	**71.80**	**300** M	**+1.13**	**-2 M**
ShuffleNetv2 [49]						
25-60	59.69	**41** M	**62.80**	46 M	**+3.11**	+5 M
120-170	68.14	146 M	**68.21**	**122** M	**+0.07**	**-24 M**
270-320	71.80	**292 M**	**72.90**	298 M	**+1.10**	+6 M

Table 3.1: Comparison between the DiCE unit and separable convolutions on the ImageNet dataset across different architectures. Models with the DiCE unit require fewer channels compared to models with separable convolution in order to obtain similar performance. Thus, models with the DiCE unit have fewer FLOPs compared to separable convolutions.

3.4.1 DiCE Unit vs. Separable Convolutions

Table 3.1 shows the performance of the DiCE unit with different architectures at different FLOP ranges. On replacing separable convolutions with the DiCE unit in the MobileNet architecture, we observe significant gains in performance both in terms of accuracy and efficiency. Because this architecture does not employ any advanced methods (e.g., residual connections) to improve performance, it allows us to understand the “true” gains of the DiCE unit over separable convolutions.

When we replace separable convolutions with the DiCE unit in ResNet and ShuffleNetv2, we observe significant improvements especially for small-(25-60 MFLOPs) and medium-sized

(120-170 MFLOPs) models. For instance, the `DiCE` unit improved the performance of ShuffleNetv2 by about 3% for small-sized model (about 40 MFLOPs). Similarly, ShuffleNetv2 with the `DiCE` unit requires 24 million fewer FLOPs to achieve the same accuracy as with separable convolutions for medium-sized models (120-170 MFLOPs). These results suggests that the `DiCE` unit is generic and learns better representations than separable convolutions.

3.4.2 Importance of `DimConv` and `DimFuse`

To understand the significance of each component of the `DiCE` unit, we replace `DimConv` with depth-wise convolution and `DimFuse` with different fusion methods, including point-wise convolutions and squeeze-excitation (SE) unit [79] and study their combinations for two architectures (ResNet and ShuffleNetv2)[4]. In these experiments, we study efficient models by restricting the computational budget to be between 120 and 150 MFLOPs.

Importance of `DimConv`: We replace depth-wise convolutional layers with `DimConv` in ResNet and ShuffleNetv2 architectures. Table 3.2a shows that these networks with `DimConv` require about 10-11 million fewer FLOPs to achieve similar accuracy as the depth-wise convolution. These results suggest that encoding spatial and channel-wise information independently in `DimConv` helps learning better representations compared to encoding only spatial information in depth-wise convolution.

Importance of `DimFuse`: To understand the effect of `DimFuse`, we replace `DimFuse` with two widely used fusion operations: point-wise convolution and SE unit. Table 3.2b summarizes the results. Compared to the widely used combination of depth-wise and point-wise convolutions (or separable convolution), the combination of `DimConv` and `DimFuse` (or the `DiCE` unit) is the most effective and improves the efficiency of networks by 15-20% with little or no impact on accuracy.

[4]MobileNet's performance is significantly lower than ResNet and ShuffleNetv2, therefore, we do not use MobileNet for these experiments.

Layer	ResNet		ShuffleNetv2	
	FLOPs	**Top-1**	**FLOPs**	**Top-1**
DWise + Point-wise	142 M	67.80	146 M	68.14
`DimConv` + Point-wise	**132 M**	**68.10**	**135 M**	**68.45**

(a) Importance of `DimConv`. DWise denotes depth-wise conv.

Layer	ResNet		ShuffleNetv2	
	FLOPs	**Top-1**	**FLOPs**	**Top-1**
DWise + Point-wise (Separable)	142 M	67.80	146 M	68.14
DWise + SE	137 M	63.90	140 M	64.70
DWise + Point-wise + SE	142 M	68.20	146 M	68.60
DWise + `DimFuse`	136 M	65.90	139 M	66.80
`DimConv` + Point-wise	132 M	**68.10**	135 M	**68.45**
`DimConv` + SE	134 M	64.80	138 M	65.40
`DimConv` + Point-wise + SE	132 M	67.90	135 M	68.30
`DimConv` + `DimFuse` (`DiCE` unit)	**122 M**	67.90	**122 M**	68.21

(b) Importance of `DimFuse`. DWise denotes depth-wise conv.

Table 3.2: Evaluating `DiCE` unit on the ImageNet dataset. Top-1 accuracy is reported on the validation set. Models with `DiCE` unit require fewer channels compared to models with separable convolution in order to obtain similar performance. Therefore, models with `DiCE` unit have fewer FLOPs compared to separable convolutions.

The combination of depth-wise convolutions and `DimFuse` is not as effective as `DimConv` and `DimFuse`. `DimConv` encodes local spatial and channel-wise information, which enables the `DiCE` unit to use a less complex fusion method (`DimFuse`) for encoding global information. Unlike `DimConv`, depth-wise convolutions only encode local spatial information and require computationally expensive point-wise convolutions to encode global information.

When point-wise convolutions are replaced with the SE unit, the performance of networks

Row		Network	ResNet		ShuffleNetv2	
#	Layer	Width	FLOPs	Top-1	FLOPs	Top-1
R1	Point-wise + DWise + `DimFuse`	1×	136 M	65.90	139 M	66.80
R2	`DimFuse` + DWise + `DimFuse`	1×	78 M	60.10	78 M	61.80
R3	`DimFuse` + DWise + `DimFuse`	4×	141 M	66.20	140 M	66.90
R4	Point-wise + `DimConv` + `DimFuse`	1×	122 M	67.90	122 M	68.21
R5	`DimFuse` + `DimConv` + `DimFuse`	1×	**72 M**	62.10	**72 M**	63.70
R6	`DimFuse` + `DimConv` + `DimFuse`	4×	129 M	**68.20**	132 M	**69.20**

Table 3.3: Impact of replacing all pointwise convolutions with `DimFuse`. Here, DWise denotes depth-wise convolution.

with depth-wise and `DimConv` convolutions dropped significantly. This is because the SE unit relies on an existing convolutional unit, such as ResNext [43], to encode global spatial and channel-wise information. When the SE unit is used as a *replacement* for point-wise convolutions, it fails to effectively encode this information, resulting in performance drop.

`DimFuse` replacing all point-wise convolutions: The first layer in ShuffleNetv2 and ResNet is a point-wise convolution (Figure 3.5; Section 3.2.4). Table 3.2b shows that `DimFuse` is an effective replacement for point-wise convolutions. A natural question arises if we can replace the first point-wise convolutional layer in these architectures with `DimFuse`. Table 3.3 shows the effect of replacing point-wise convolutions with `DimFuse`. For a fixed network width, the number of FLOPs are reduced by 45% when point-wise convolutions are replaced with `DimFuse`, however, the accuracy drops by about 5-7%. For a similar number of FLOPs, networks with `DimFuse` achieve higher accuracy. However, such networks (R3 and R6) are about 4× wider than the networks with point-wise convolutions (R1 and R4); this poses memory constraints for resource-constrained devices. Therefore, we only replace separable convolutions with the `DiCE` unit while keeping the remaining architecture intact.

3.4.3 Effect of MobileNet's training hyper-parameters

In previous experiments, we follow the experimental setup similar to ESPNet and ShuffleNetv2 i.e., each model is trained for 150 epochs using a batch size of 512 and minimizes cross-entropy loss using SGD. Recent efficient models (e.g., MobileNetv3 and MNASNet) are trained longer (600 epochs) with extremely large batch size (4096) using cross-entropy with label smoothing (CE-LS) and exponential moving average (EMA). We also trained `DiCENet` with CE-LS and EMA, with an exception to batch size (2048) and number of epochs (300). For training with larger batch size, we accumulated the gradients for 4 iterations. This resulted in an effective batch size of 2048 (128 images per NVIDIA GTX 1080

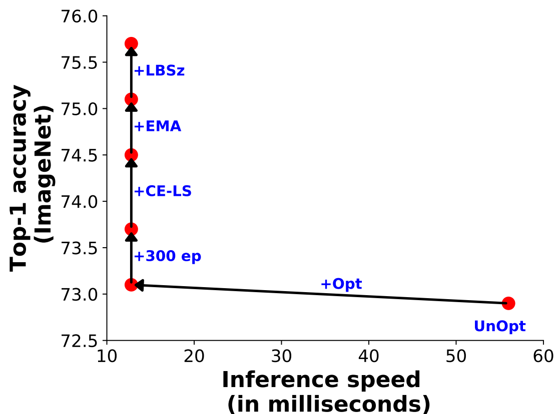


Figure 3.7: **Impact of different components in the training of `DiCENet`.** With +, we indicate that the component is added to the previous configuration. Here, **UnOpt** represents the un-optimized `DiCENet` trained for 150 epochs with a batch size of 512 with cross entropy and **Opt** represents `DiCENet` with custom CUDA kernel. **300 ep** denotes that model is trained for 300 epochs, **CE-LS** denotes that label-smooth cross-entropy is used, **EMA** denoted that exponential moving average is used, and **LBSz** denotes that large batch size (2048 images) is used for training. Here, inference time is measured on an NVIDIA GTX 1080 Ti GPU and is an average across 100 trials for a batch of 32 RGB images, each with a spatial dimension of 224×224.

Ti GPU × 4 GPUs × accumulation frequency of 4). Figure 3.7 shows the effect of these changes. Similar to state-of-the-art models (e.g., MobileNetv2, MobileNetv3, MixNet, and MNASNet), `DiCENet` also benefits from these hyper-parameters and yields a top-1 accuracy of 75.7 on the ImageNet.

Importantly, the optimized CUDA kernel (Figure 3.6) improved the inference speed drastically over the unoptimized version. This is because the optimized kernel launches one kernel for these three branches compared to one per branch in the unoptimized one and also, maximizes work done per CUDA thread. This reduces latency. With optimized CUDA kernels, `DiCENet` models (10-300 MFLOPs) takes between one and three days for training on the ImageNet on 4 NVIDIA GTX 1080 Ti GPUs with an effective batch size of 2048.

3.5 *Evaluating* `DiCENet` *on the ImageNet*

In this section, we evaluate the performance of `DiCENet` on the ImageNet dataset and show that `DiCENet` delivers significantly better performance than state-of-the-art efficient networks. Recall that `DiCENet` is ShuffleNetv2 with the `DiCE` unit (Section 3.2.4).

Implementation details: We scale the number of output channels by a width scaling factor s to obtain `DiCENet` models at different complexity levels, ranging from 6 MFLOPs to 300 MFLOPs (see Appendix A.1 for details).

Evaluation metrics and baselines: We use 224 × 224 single-crop top-1 accuracy to evaluate the performance on the validation set. The performance of `DiCENet` is compared with state-of-the-art *manually* designed efficient networks (MobileNets [46, 47], ShuffleNetv2 [49], CondenseNet [51], and ESPNet [37]) and *automatically* designed networks (MNASNet [55], FBNet [56], MixNet [78], and MobileNetv3 [58]).

Results: Recent studies (e.g., MobileNetv3) have shown that longer training with extremely large batch sizes improves performance. To have fair comparisons with state-of-the-art methods, we report the performance of `DiCENet` in two settings. The first setting,

Network	Type	FLOP ranges (in millions)						
		< 10 M	10-20 M	21-60 M	61-90 M	91-130 M	131-170 M	200 -320 M
MobileNet [46]	Manual		41.5 (14)	56.3 (49)	59.1 (77)	61.7 (110)	65.3 (162)	68.4 (317)
MobileNetv2 [47]	Manual		45.5 (**11**)	61.0 (50)	63.9 (71)	66.4 (107)	68.7 (153)	71.8 (300)
ESPNet [37]	Manual				66.1 (86)	67.9 (124)		72.1 (284)
CondenseNet [51]	Manual							71.0 (274)
ShuffleNetv2 [49]	Manual	39.1 (8.0)		59.7 (41)			68.1 (142)	71.8 (292)
MNASNet [55]	NAS				62.4 (76)	67.3 (103)		74.0 (317)
FBNet [56]	NAS				65.3 (72)	67.0 (92)		74.1 (295)
MobileNetv3 [58]	NAS				67.4 (66)			75.2 (219)
MixNet [78]	NAS							**75.8** (256)
`DiCENet`**-E150-B512 (Ours)**	Manual	40.6 (6.5)	46.2 (14)	62.8 (46)	66.5 (70)	67.8 (98)	69.5 (139)	72.9 (298)
`DiCENet`**-E300-B2048 (Ours)**	Manual	**43.1** (6.5)	**48.2** (14)	**65.1** (46)	**68.5** (70)	**69.3** (98)	**72.0** (139)	75.7 (298)

Table 3.4: **Results on the ImageNet dataset.** `DiCENet` delivers similar or better performance than state-of-the-art methods, including neural architecture search (NAS)-based methods. Here, each entry is represented as top-1 accuracy and FLOPs are given within parentheses. `DiCENet`-E150-B512 models are trained for 150 epochs with a batch size of 512 (without EMA and label smoothing), while `DiCENet`-E300-B2048 models are trained for 300 epochs with an effective batch size of 2048 (with EMA and label smoothing).

`DiCENet`-E150-B512, is similar to networks like ESPNet, ShuffleNetv2, and CondenseNet where `DiCENet` is trained for fewer epochs (150) with a smaller batch size (512) without EMA and label smoothing. The second setting, `DiCENet`-E300-B2048, is similar to networks like MobileNets, MNASNet, and MixNet, where `DiCENet` is trained for 300 epochs with a batch size of 2048. Table 3.4 compares the performance of `DiCENet` with state-of-the-art efficient architectures at different FLOP ranges.

Compared to networks that are trained with smaller batch sizes and fewer epochs, e.g., ESPNet (epochs: 300; batch size: 512) and ShuffleNetv2 (epochs: 240 and batch size: 1024), `DiCENet` delivers better performance across different FLOP ranges. Similarly, when

Model	Network statistics			Device: GTX-960 M (Memory = 4GB)		Device: GTX-1080 Ti (Memory = 11 GB)		
	# Params	# FLOPs	Top-1	Batch size = 1	Batch size = 32	Batch size = 1	Batch size = 32	Batch size = 64
MobileNetv2	3.5 M	300 M	71.8	**5.6 ± 0.2 ms**	114 ± 0.1 ms	**5.9 ± 0.1 ms**	22.8 ± 0.1 ms	44.3 ± 0.8 ms
ShuffleNetv2	3.5 M	300 M	71.8	**5.8 ± 0.2 ms**	80.7 ± 0.6 ms	**5.8 ± 0.2 ms**	**12.9 ± 0.1 ms**	**24.1 ± 0.4 ms**
MobileNetv3	5.5 M	220 M	75.2	8.5 ± 0.2 ms	Out-of-memory	9.0 ± 0.3 ms	20.9 ± 0.1 ms	40.2 ± 0.2 ms
DiCENet (Ours)	5.1 M	297 M	**75.7**	**5.9 ± 0.1 ms**	**79 ± 0.1 ms**	**5.7 ± 0.1 ms**	**12.8 ± 0.1 ms**	**24.1 ± 0.6 ms**

Table 3.5: **Inference speed**. DiCENet and ShuffleNetv2 are comparatively faster than MobileNetv2 and MobileNetv3 on both devices (Mobile GPU: GTX-960M and Desktop GPU: GTX-1080 Ti). Inference results are an average over 100 trials for RGB input images of size 224×224. We used PyTorch with CUDA 10.2 for measuring speed. MobileNetv2 and ShuffleNetv2 implementations are taken from official PyTorch repository while MobileNetv3's implementation is taken from [80]

.

DiCENet is trained for longer with larger batch sizes, it delivers similar or better performance than state-of-the-art methods, including neural architecture search (NAS)-based methods. For about 300 MFLOPs, DiCENet is 4% more accurate than MobileNetv2. Specifically, we observe that DiCENet is very effective when model size is small (FLOPs $<$ 150 M). For example, DiCENet outperforms MNASNet, FBNet, and MobileNetv3 by 6.1%, 3.2%, and 1.1%, respectively, for network size of about 70 MFLOPs. Overall, these results shows that DiCE unit learns better representations than separable convolutions. We believe that incorporating the DiCE unit with NAS would yield a better network.

Inference speed: We measure the inference time on two GPUs: (1) embedded or mobile GPU (NVIDIA GTX 960M) and (2) desktop GPU (NVIDIA GTX 1080 Ti)[5]. DiCENet is as fast as ShuffleNetv2 but delivers better performance. Compared to MobileNetv2 and MobileNetv3, DiCENet has low latency while delivering similar or better performance. We

[5] We do not measure the inference speed on Smartphones because efficient implementations of DiCE unit are not yet available for such devices.

observe that MobileNetv2 and MobileNetv3 models are slow in comparison to ShuffleNetv2 and `DiCENet` when the batch size increases. This is because the number of channels in the depth-wise convolution in the inverted residual block of MobileNetv2 and MobileNetv3 are very large as compared to `DiCENet` and ShuffleNetv2. For example, the maximum number of channels in depth-wise convolution in MobileNetv2 (300 MFLOPs) are 960, while the maximum number of channels in `DimConv` and depth-wise convolution in `DiCENet` (298 MFLOPs) and ShuffleNetv2 (292 MFLOPs) are 576 and 352, respectively.

3.6 Task-level Generalization of `DiCENet`

Several previous works (e.g., [9, 81]) have shown that high accuracy on the Imagenet dataset does not necessarily correlates with high accuracy on visual scene understanding tasks (e.g., object detection and semantic segmentation). Since these tasks are widely used in real-world applications (e.g., autonomous wheel chairs and robots) and often run on resource-constrained devices (e.g., embedded devices), it is important that efficient networks generalize well on these tasks. Therefore, we evaluate the performance of `DiCENet` on three different tasks: (1) object detection (Section 3.6.1), (2) semantic segmentation (Section 3.6.2), and (3) multi-object classification (Section 3.6.3). Compared to existing efficient networks that are built using separable convolutions (e.g., MobileNets [46, 47, 58], MixNet [78], and ESPNet [37]), `DiCENet` delivers better performance.

3.6.1 Object Detection on VOC and MS-COCO

Implementation details: For object detection, we use a Single Shot object Detection (SSD) [77] pipeline. We use `DiCENet` (298 MFLOPs) pretrained on the ImageNet as a base feature extractor instead of VGG [45]. We fine-tune our network using SGD with smooth L1 and cross-entropy losses for object localization and classification, respectively.

Evaluation metrics and baselines: We evaluate the performance using mean Average Precision (mAP). For MS-COCO, we report mAP@IoU of 0.50:0.95. For SSD as a detection

SSD backbone	Image size	PASCAL VOC 2007		MS-COCO	
		FLOPs	mAP	FLOPs	mAP
VGG [45]	300x300	31.3 B	**72.4**	35.2 B	23.2
MobileNet [46]	320x320	–	–	1.3 B	22.2
MobileNetv2 [47]	320x320	–	–	**0.8 B**	22.1
ESPNet [37]	256x256	0.9 B	70.3	1.1 B	21.9
`DiCENet` (Ours)	300x300	**0.7 B**	71.9	0.9 B	**25.1**
MobileNetv3 [58] (NAS)	320x320	–	–	0.6 B	22.0
MixNet [78] (NAS)	320x320	–	–	0.9 B	22.3
MNASNet [55] (NAS)	320x320	–	–	0.8 B	23.0

(a) Quantitative results of SSD [77] with different backbones. On the MS-COCO dataset, total network parameters in SSD with different backbones are about 5 million, except VGG.

(b) Qualitative results of `DiCENet` under diverse background conditions

Table 3.6: **Object detection** using `DiCENet`.

pipeline, we compare DiCENet's performance with two types of base feature extractors: (1) *manual* (VGG [45], MobileNet [46], MobileNetv2 [47], and ESPNet [37]) and (2) *NAS-based* (MNASNet [55], MixNet [78], and MobileNetv3 [58]).

Results: Table 3.6 compares quantitative results of SSD with different backbone networks on the PASCAL VOC 2007 and the MS-COCO datasets. DiCENet significantly improves the performance of SSD-based object detection pipeline and delivers 1% to 4% higher mAP than other existing efficient variants of SSD, including NAS-based backbones such as MobileNetv3 (22.0 vs. **25.1**) and MixNet (22.3 vs. **25.1**). Compared to standard SSD with VGG as backbone, DiCENet achieves a higher mAP on MS-COCO while having **38**× fewer FLOPs.

3.6.2 Semantic Segmentation on PASCAL VOC

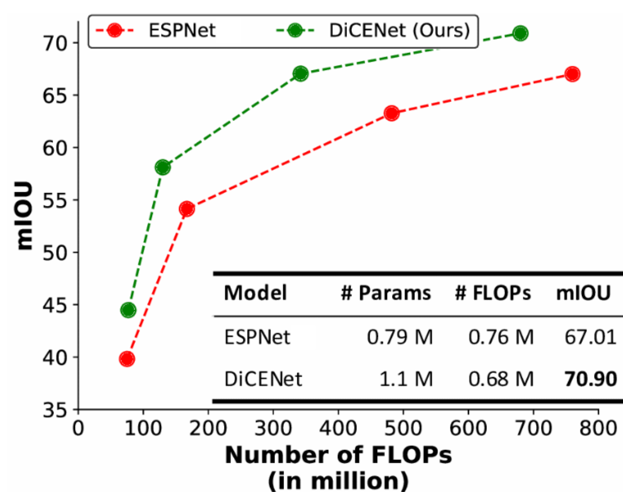
Implementation details: We adapt DiCENet to ESPNet's [37] encoder-decoder architecture. We choose this network because it delivers competitive performance to existing methods even with low resolution images (e.g. 256×256 vs. 512×512). We replace the encoder in ESPNet (pretrained on ImageNet) with the DiCENet and follow the same training procedure for fine-tuning as ESPNet. We do not change the decoder.

Evaluation metrics and baselines: The performance at different complexity levels (FLOPs) is evaluated using mean intersection over union (mIOU).

Results: Figure 3.8 compares the performance of DiCENet with ESPNet on the PASCAL VOC 2012 validation set. DiCENet significantly improves the segmentation performance i.e., for similar FLOPs, DiCENet achieves higher mIOU while for similar mIOU, DiCENet requires significantly fewer FLOPs.

3.6.3 Multi-object Classification on MS-COCO

Implementation details: DiCENet is fine-tuned using the binary cross-entropy loss.



(a) Qualitative results. Here, mIOU represents mean intersection over union.

(b) Quantitative results (**top row**: Input image, **middle row**: ground truth, **last row**: `DiCENet` predictions)

Figure 3.8: **Semantic segmentation** results on the PASCAL VOC 2012 validation set.

Network	# Params	# FLOPs	F1-score	
			Class-wise	Overall
ShuffleNetv2 [49]†	3.5 M	300 M	60.42	67.58
ESPNet [37]†	3.5 M	284 M	63.41	69.23
`DiCENet` (ours)	5.1 M	298 M	**66.92**	**73.41**

Table 3.7: **Multi-object classification** results on the MS-COCO dataset. Here, † indicates that results are from [37].

Evaluation metrics and baselines: Similar to [37], we evaluate the performance using overall and per-class F1 score and compare with two efficient architectures, i.e., ESPNet and ShuffleNetv2.

Results: Table 3.7 shows that `DiCENet` outperforms existing efficient networks by a significant margin (e.g., ESPNet and ShuffleNetv2 by 4.1% and 5.8%, respectively) on this task.

3.7 Summary

We introduce a novel and generic convolutional unit, the `DiCE` unit, that uses dimension-wise convolutions and dimension-wise fusion modules to learn spatial and channel-wise representations efficiently. Our empirical results suggest that the `DiCE` unit is more effective than separable convolutions. Moreover, when we stack `DiCE` units to build `DiCENet` model, we observe significant improvements across different computer vision tasks.

Chapter 4

EFFICIENT RECURRENT NEURAL NETWORK FOR LEARNING TEXTUAL REPRESENTATIONS

4.1 Introduction

Long short term memory (LSTM) units [36] are popular for many sequence modeling tasks and are used extensively in language modeling. A key to their success is their articulated gating structure, which allows for more control over the information passed along the recurrence. However, despite the sophistication of the gating mechanisms employed in LSTMs and similar recurrent units, the input and context vectors are treated with simple linear transformations prior to gating. Other local linear transformations such as convolutions [82] have been used, but these have not achieved the performance of well-regularized LSTMs for language modeling [83].

A natural way to improve the expressiveness of linear transformations is to increase the number of dimensions of the input and context vectors, but this comes with a significant increase in the number of parameters which may limit generalizability. An example is shown in Figure 4.1, where the LSTM's performance decreases with the increase in dimensions of the input and context vectors. Moreover, the semantics of the input and context vectors are different, suggesting that each may benefit from specialized treatment.

Guided by these insights, we introduce a new recurrent unit, the Pyramidal Recurrent Unit (PRU), which is based on the LSTM gating structure. Figure 4.2 provides an overview of the PRU. At the heart of the PRU is the *pyramidal transformation* (PT), which uses subsampling to effect multiple views of the input vector. The subsampled representations are combined in a pyramidal fusion structure, resulting in richer interactions between the individual dimensions of the input vector than is possible with a linear transformation.

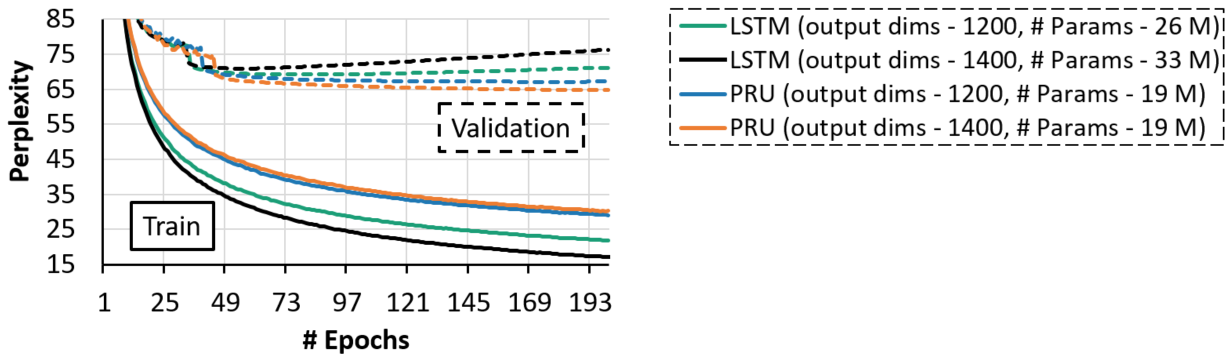


Figure 4.1: Comparison of training (solid lines) and validation (dashed lines) perplexities on the Penn Treebank with standard dropout for pyramidal recurrent units (PRU) and LSTM. PRUs learn latent representations in very high-dimensional space with good generalizability and fewer parameters. See Section 4.3 for more details about PRUs. Best viewed in color.

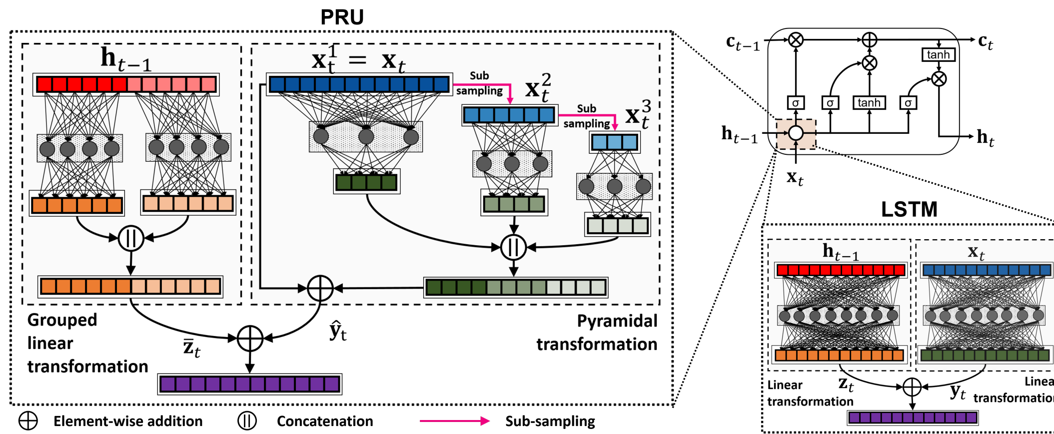


Figure 4.2: Block diagram visualizing the transformations in the pyramidal recurrent unit (left) and the LSTM (bottom right) along with the LSTM gating architecture (top right). **Blue**, **red**, **green** (or **orange**), and **purple** signify the current input $\mathbf{x}_t$, output of the previous cell $\mathbf{h}_{t-1}$, the output of transformations, and the fused output, respectively. The color intensity is used to represent sub-sampling and grouping operations.

Context vectors, which have already undergone this transformation in the previous cell, are modified with a *grouped linear transformation* (GLT) which allows the network to learn latent representations in high-dimensional space with fewer parameters and better generalizability (see Figure 4.1).

We show that PRUs can better model contextual information and demonstrate performance gains on the task of language modeling. The PRU improves the perplexity of the current state-of-the-art language model [84] by up to 1.3 points, reaching perplexities of 56.56 and 64.53 on the Penn Treebank and WikiText2 datasets, while learning 15-20% fewer parameters. Replacing an LSTM with a PRU results in improvements in perplexity across a variety of experimental settings. We provide detailed ablations which motivate the design of the PRU architecture, as well as a detailed analysis of the effect of the PRU on other components of the language model.

4.2 Related work

Multiple methods, including a variety of gating structures and transformations, have been proposed to improve the performance of recurrent neural networks (RNNs). We first describe these approaches and then provide an overview of recent work in language modeling.

Gating-based mechanisms: The performance of RNNs have been greatly improved by gating mechanisms such as LSTMs [36], GRUs [85], peep-hole connections [86], SRUs [87], and RANs [88]. This work extends the gating architecture of LSTMs [36], a widely used recurrent unit across different domains.

Transformations: Apart from the widely used linear transformation for modeling the temporal data, another transformation that has gained popularity is the convolution [35]. Convolution-based methods have gained attention in computer vision tasks [3] as well as some of the natural language processing tasks including machine translation [89]. Convolution-based methods for language modeling, such as CharCNN [82], have not yet achieved the

performance of well regularized LSTMs [83]. We inherit ideas from convolution-based approaches, such as sub-sampling, to learn richer representations [3, 90].

Regularization: Methods such as dropout [91], variational dropout [92], and weight dropout [84] have been proposed to regularize RNNs. These methods can be easily applied to PRUs.

Other efficient RNN networks: Recently, there has been an effort to improve the efficiency of RNNs. These approaches include quantization [93], skimming [94, 95], skipping [96], and query reduction [97]. These approaches extend standard RNNs and therefore, these approaches are complementary to our work.

Language modeling: Language modeling is a fundamental task for NLP and has garnered significant attention in recent years (see Table 4.1 for a comparison with state-of-the-art methods). Merity et al. [84] introduce regularization techniques such as weight dropping which, coupled with a non-monotonically triggered ASGD optimization, achieves strong performance improvements. Yang et al. [98] extend Merity et al. [84] with the mixture of the softmaxes (MoS) technique, which increases the rank of the matrix used to compute next-token probabilities. Further, Merity et al. [99] and Krause et al. [100] propose methods to improve inference by adapting models to recent sequence history. Our work is complementary to these recent softmax layer and inference procedure improvements.

In this chapter, we extend the state-of-the-art language model, AWD-LSTM [84], by replacing the LSTM with the PRU and study it on the task of language modeling. We show by experiments that the PRU improves the performance of [84] while learning fewer parameters.

4.3 Pyramidal Recurrent Units

We introduce Pyramidal Recurrent Units (PRUs), a new light-weight RNN architecture which improves modeling of context by allowing for higher-dimensional vector representations. Figure 4.2 provides an overview of PRU. We first elaborate on the details of the

pyramidal transformation (Section 4.3.1) and the grouped linear transformation (Section 4.3.2). We then describe our recurrent unit, PRU (Section 4.3.3).

4.3.1 Pyramidal transformation for input

The basic transformation in many recurrent units is a linear transformation $\mathcal{F}_L$ defined as:

$$\mathbf{y} = \mathcal{F}_L(\mathbf{x}) = \mathbf{W} \cdot \mathbf{x}, \tag{4.1}$$

where $\mathbf{W} \in \mathbb{R}^{N\times M}$ are learned weights that linearly map $\mathbf{x} \in \mathbb{R}^N$ to $\mathbf{y} \in \mathbb{R}^M$. To simplify notation, we omit the biases.

Motivated by successful applications of sub-sampling in computer vision (e.g., [3, 101, 102]), we subsample input vector $\mathbf{x}$ into K pyramidal levels to achieve representation of the input vector at multiple scales. This sub-sampling operation produces K vectors, represented as $\mathbf{x}^k \in \mathbb{R}^{\frac{N}{2^{k-1}}}$, where 2^{k-1} is the sampling rate and $k = \{1, \cdots, K\}$. We learn scale-specific transformations $\mathbf{W}^k \in \mathbb{R}^{\frac{N}{2^{k-1}} \times \frac{M}{K}}$ for each $k = \{1, \cdots K\}$. The transformed subsamples are concatenated to produce the pyramidal analog to $\mathbf{y}$, here denoted as $\bar{\mathbf{y}} \in \mathbb{R}^M$:

$$\bar{\mathbf{y}} = \mathcal{F}_P(\mathbf{x}) = \text{Concat}\left(\mathbf{W}^1 \cdot \mathbf{x}^1, \cdots, \mathbf{W}^K \cdot \mathbf{x}^K\right) \tag{4.2}$$

We note that the pyramidal transformation with $K = 1$ is the same as the linear transformation.

To improve gradient flow inside the recurrent unit, we combine the input and output using an element-wise sum (when the dimension matches) to produce the residual analog of the pyramidal transformation, as shown in Figure 4.2.

Sub-sampling: We sub-sample the input vector $\mathbf{x}$ into K pyramidal levels using the kernel-based approach [3, 35]. Let us assume that we have a kernel κ with $2e + 1$ elements. Then, the input vector $\mathbf{x}$ can be sub-sampled with a stride of s as:

$$\mathbf{x}^k = \sum_{i=1}^{N/s} \sum_{j=-e}^{e} \mathbf{x}^{k-1}[si]\kappa[j], \; k = \{2, \cdots, K\} \tag{4.3}$$

Reduction in parameters: The number of parameters learned by the linear transformation and the pyramidal transformation with K pyramidal levels to map $\mathbf{x} \in \mathbb{R}^N$ to $\bar{\mathbf{y}} \in \mathbb{R}^M$ are NM and $\frac{NM}{K} \sum_{k=1}^{K} 2^{(1-k)}$, respectively. Thus, a pyramidal transformation reduces the parameters of a linear transformation by a factor of $K / \left(\sum_{k=1}^{K} 2^{(1-k)}\right)$. As an example, the pyramidal transformation (with $K = 4$ and $N = M = 600$) learns 53% fewer parameters than the linear transformation.

4.3.2 Grouped linear transformation for context

Many RNN architectures apply linear transformations to both the input and context vector. However, this may not be ideal due to the differing semantics of each vector. In many NLP applications including language modeling, the input vector is a dense word embedding, which is shared across all contexts for a given word in a dataset. In contrast, the context vector is highly contextualized by the current sequence. The differences between the input and context vector motivate their separate treatment in the PRU architecture.

The weights learned using the linear transformation (Eq. 4.1) are reused over multiple time steps, which makes them prone to over-fitting [103]. To combat over-fitting, various methods, such as variational dropout [103] and weight dropout [84], have been proposed to regularize these recurrent connections. To further improve generalization abilities, while simultaneously enabling the recurrent unit to learn representations at very high dimensional space, we propose to use a grouped linear transformation (GLT) instead of a standard linear transformation for recurrent connections. While pyramidal and linear transformations can be applied to transform context vectors, our experimental results in Section 4.4.4 suggests that GLTs are more effective.

The linear transformation $\mathcal{F}_L : \mathbb{R}^N \rightarrow \mathbb{R}^M$ maps $\mathbf{h} \in \mathbb{R}^N$ linearly to $\mathbf{z} \in \mathbb{R}^M$. Grouped linear transformations break the linear interactions by factoring the linear transformation into two steps. First, a GLT splits the input vector $\mathbf{h} \in \mathbb{R}^N$ into g smaller groups such that $\mathbf{h} = \{\mathbf{h}^1, \cdots, \mathbf{h}^g\}, \forall\ \mathbf{h}^i \in \mathbb{R}^{\frac{N}{g}}$. Second, a linear transformation $\mathcal{F}_L : \mathbb{R}^{\frac{N}{g}} \rightarrow \mathbb{R}^{\frac{M}{g}}$ is applied to

map $\mathbf{h}^i$ linearly to $\mathbf{z}^i \in \mathbb{R}^{\frac{M}{g}}$, for each $i = \{1, \cdots, g\}$. The g resultant output vectors $\mathbf{z}^i$ are concatenated to produce the final output vector $\bar{\mathbf{z}} \in \mathbb{R}^M$.

$$\bar{\mathbf{z}} = \mathcal{F}_G(\mathbf{h}) = \left[\mathbf{W}^1 \cdot \mathbf{h}^1, \cdots, \mathbf{W}^g \cdot \mathbf{h}^g\right] \tag{4.4}$$

Group linear transformations splits the input into g groups and learn representations for each group independently. Therefore, a group linear transformation requires $g\times$ fewer parameters than the linear transformation. We note that a grouped linear transformation is the same as a linear transformation when $g = 1$.

4.3.3 Pyramidal Recurrent Unit

We extend the basic gating architecture of the LSTM with the pyramidal and grouped linear transformations outlined above to produce the Pyramidal Recurrent Unit (PRU), whose improved sequence modeling capacity is evidenced in Section 4.4.

At time t, the PRU combines the input vector $\mathbf{x}_t$ and the previous context vector (or previous hidden state vector) $\mathbf{h}_{t-1}$ using the following transformation function as:

$$\hat{\mathcal{G}}_v(\mathbf{x}_t, \mathbf{h}_{t-1}) = \hat{\mathcal{F}}_P(\mathbf{x}_t) + \mathcal{F}_G(\mathbf{h}_{t-1}), \tag{4.5}$$

where $v \in \{f, i, c, o\}$ indexes the various gates in the LSTM model, and $\hat{\mathcal{F}}_P(\cdot)$ and $\mathcal{F}_G(\cdot)$ represent the pyramidal and grouped linear transformations defined in Eqns. 4.2 and 4.4, respectively.

We will now incorporate $\hat{\mathcal{G}}_v(\cdot, \cdot)$ into the LSTM gating architecture to produce PRU. At time t, a PRU cell takes $\mathbf{x}_t \in \mathbb{R}^N$, $\mathbf{h}_{t-1} \in \mathbb{R}^M$, and $\mathbf{c}_{t-1} \in \mathbb{R}^M$ as inputs to produce outputs of forget $\mathbf{f}_t$, input $\mathbf{i}_t$, output $\mathbf{o}_t$, and content $\hat{\mathbf{c}}_t$ gates. The inputs are combined with these gate signals to produce context vector $\mathbf{h}_t \in \mathbb{R}^M$ and cell state $\mathbf{c}_t \in \mathbb{R}^M$. Mathematically, the PRU with the LSTM gating architecture can be defined as:

$$\begin{aligned}
\mathbf{f}_t &= \sigma\left(\hat{\mathcal{G}}_f(\mathbf{x}_t, \mathbf{h}_{t-1})\right) && \text{\% Forget gate}\\
\mathbf{i}_t &= \sigma\left(\hat{\mathcal{G}}_i(\mathbf{x}_t, \mathbf{h}_{t-1})\right) && \text{\% Input gate}\\
\hat{\mathbf{c}}_t &= tanh\left(\hat{\mathcal{G}}_c(\mathbf{x}_t, \mathbf{h}_{t-1})\right) && \text{\% Content gate}\\
\mathbf{o}_t &= \sigma\left(\hat{\mathcal{G}}_o(\mathbf{x}_t, \mathbf{h}_{t-1})\right) && \text{\% Output gate}\\
\mathbf{c}_t &= \mathbf{f}_t \otimes \mathbf{c}_{t-1} + \mathbf{i}_t \otimes \hat{\mathbf{c}}_t && \text{\% Cell state}\\
\mathbf{h}_t &= \mathbf{o}_t \otimes tanh(\mathbf{c}_t) && \text{\% Contextualized output}
\end{aligned} \tag{4.6}$$

where $\otimes$ represents the element-wise multiplication operation, and σ and $tanh$ are the sigmoid and hyperbolic tangent activation functions. We note that LSTM is a special case of PRU when $g = K = 1$.

4.4 Experiments

To showcase the effectiveness of the PRU, we evaluate the performance on two standard datasets for word-level language modeling and compare with state-of-the-art methods. Additionally, we provide a detailed examination of the PRU and its behavior on the language modeling tasks.

4.4.1 Set-up

Dataset: Following recent works, we compare on two widely used datasets, the Penn Treebank (PTB) [104] as prepared by Mikolov et al. [105] and WikiText2 (WT-2) [99]. For both datasets, we follow the same training, validation, and test splits as in Merity et al. [84].

Language Model: We extend the language model, AWD-LSTM [84], by replacing the LSTM layers with the PRU. Our model uses 3-layers of PRU with an embedding size of 400. The number of parameters learned by state-of-the-art methods vary from 18M to 66M with the majority of the methods learning about 22M to 24M parameters on the PTB dataset. For a fair comparison with state-of-the-art methods, we fix the model size to 19M and vary the value of g and hidden layer sizes so that the total number of learned parameters is similar

across different configurations. We use 1000, 1200, and 1400 as hidden layer sizes for values of $g = 1,\ 2,$ and 4, respectively. We use the same settings for the WT-2 dataset. We set the number of pyramidal levels K to two in our experiments and use average pooling for sub-sampling. These values are selected based on our ablation experiments on the validation set in Section 4.4.4. We follow the same training strategy as in Merity et al. [84] and measure the performance of our models in terms of word-level perplexity. The perplexity measures how well the language model predicts the next word given previous words and for a sequence $S = (w_1, \cdots, w_T)$, it is defined as:

$$\text{Perplexity} = b^{\sum_{i=1}^{T} \log_b \ p(w_j | w_1, \cdots, w_{j-1})} \tag{4.7}$$

where b is the logarithm base. The lower value of perplexity means better performance.

To understand the effect of regularization methods on the performance of PRUs, we perform experiments under two different settings:

- *Standard dropout*: We use a standard dropout [91] with probability of 0.5 after embedding layer, the output between LSTM layers, and the output of final LSTM layer.
- *Advanced dropout:* We use the same dropout techniques with the same dropout values as AWD-LSTM [84]. We call this model AWD-PRU.

4.4.2 Results

Table 4.1 compares the performance of the PRU with state-of-the-art methods. We can see that the PRU achieves the best performance with fewer parameters.

Standard dropout: PRUs achieve either the same or better performance than LSTMs. In particular, the performance of PRUs improves with the increasing value of g. At $g = 4$, PRUs outperform LSTMs by about 4 points on the PTB dataset and by about 3 points on the WT-2 dataset. This is explained in part by the regularization effect of the grouped linear transformation (Figure 4.1). With grouped linear and pyramidal transformations, PRUs

Model	WT-2 Params	WT-2 Val	WT-2 Test	PTB Params	PTB Val	PTB Test
Quantized LSTM - 2 bit [93]	–	–	106.1	–	–	95.8
Quantized LSTM - Full precision [93]	–	–	100.1	–	–	89.8
CharCNN [82]	–	–	–	19 M	–	78.9
Pointer Sentinel-LSTM [99]	–	–	–	19 M	72.4	70.9
RHN [106]	–	–	–	23 M	67.9	65.4
NAS Cell [54]	–	–	–	25 M	–	64.0
Variational LSTM - [103, 107]	28 M	91.5	87	24 M	75.7	73.2
SRU - 6 layers [87]	–	–	–	24 M	63.4	60.3
QRNN [108]	–	–	–	**18 M**	82.1	78.3
RAN [88]	–	–	–	22 M	–	78.5
NAS Cell [54]	–	–	–	54 M	–	62.4
4-layer skip-connection LSTM [83]	–	–	–	24 M	60.9	58.3
AWD-LSTM - [84]	33 M	69.1	66	24 M	60.7	58.8
AWD-LSTM - [84]-finetuned	33 M	68.6	65.8	24 M	60	57.3
With standard dropout						
LSTM ($M = 1000$)	29 M	78.93	75.08	20 M	68.57	66.29
LSTM ($M = 1200$)	35 M	77.93	74.48	26 M	69.17	67.16
LSTM ($M = 1400$)	42 M	77.55	74.44	33 M	70.88	68.55
Ours -PRU ($g = 1$, $K = 2$, $M = 1000$)	28 M	79.15	76.59	19 M	69.8	67.78
Ours -PRU ($g = 2$, $K = 2$, $M = 1200$)	28 M	76.62	73.79	19 M	67.17	64.92
Ours -PRU ($g = 4$, $K = 2$, $M = 1400$)	28 M	75.46	72.77	19 M	64.76	62.42
With advanced dropouts						
Ours - AWD-PRU ($g = 1$, $K = 2$, $M = 1000$)	28 M	71.84	68.6	19 M	61.72	59.54
Ours - AWD-PRU ($g = 2$, $K = 2$, $M = 1200$)	28 M	68.57	65.7	19 M	60.81	58.65
Ours - AWD-PRU ($g = 4$, $K = 2$, $M = 1400$)	28 M	68.17	65.3	19 M	60.62	58.33
Ours - AWD-PRU ($g = 4$, $K = 2$, $M = 1400$)-finetuned	**28 M**	**67.19**	**64.53**	19 M	**58.46**	**56.56**

Table 4.1: Comparison of single model word-level perplexity of our model with the state-of-the-art on the validation and test sets of the Penn Treebank and Wikitext-2 dataset. For evaluation, we select the model with minimum validation loss. A lower perplexity value represents better performance.

learn rich representations in very high-dimensional space while learning fewer parameters. On the other hand, LSTMs overfit to the training data at such high dimensions and learn $1.4\times$ to $1.8\times$ more parameters than PRUs.

Advanced dropouts: With the advanced dropouts, the performance of PRUs improves by about 4 points on the PTB dataset and 7 points on the WT-2 dataset. This further improves with finetuning on the PTB (about 2 points) and WT-2 (about 1 point) datasets.

Comparison with state-of-the-art: For a similar number of parameters, the PRU with standard dropout outperforms most of the state-of-the-art methods by a large margin on the PTB dataset (e.g. RAN [88] by 16 points with 4M less parameters, QRNN [108] by 16 points with 1M more parameters, and NAS [54] by 1.58 points with 6M less parameters). With advanced dropouts, the PRU delivers the best performance. On both datasets, the PRU improves the perplexity by about 1 point while learning 15-20% fewer parameters.

Inference: The PRU is a drop-in replacement for the LSTM; therefore, it can improve language models with modern inference techniques such as dynamic evaluation [100]. When we evaluate PRU-based language models (only with standard dropout) with dynamic evaluation on the PTB test set, the perplexity of PRU ($g = 4, k = 2, M = 1400$) improves from 62.42 to 55.23, while the perplexity of an LSTM ($M = 1000$) with similar settings improves from 66.29 to 58.79; suggesting that modern inference techniques are equally applicable to PRU-based language models.

4.4.3 Analysis

It is shown above that the PRU can learn representations at higher dimensionality with more generalization power, resulting in performance gains for language modeling. A closer analysis of the impact of the PRU in a language modeling system reveals several factors that help explain how the PRU achieves these gains.

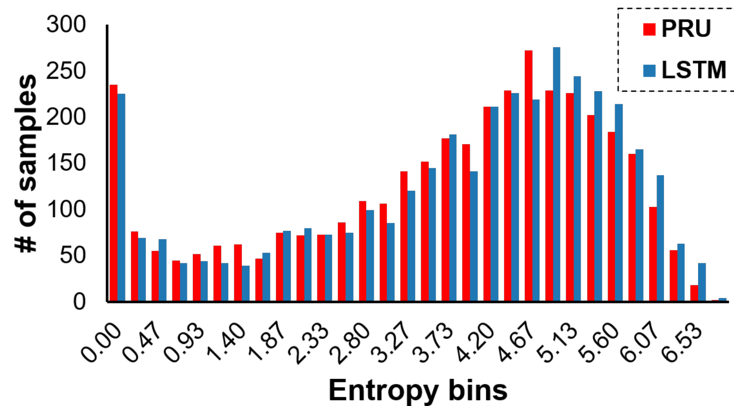


Figure 4.3: Histogram of the entropies of next-token distributions predicted by the PRU (mean 3.80) and the LSTM (mean 3.93) on the PTB validation set. Lower entropy values indicate higher confidence decisions, which is desirable if decisions are often correct.

Confidence: As exemplified in Figure 4.5a, the PRU tends toward more confident decisions, placing more of the probability mass on the top next-word prediction than the LSTM. To quantify this effect, we calculate the entropy of the next-token distribution for both the PRU and the LSTM using 3687 contexts from the PTB validation set. Figure 4.3 shows a histogram of the entropies of the distribution, where bins of size 0.23 are used to effect categories. We see that the PRU more often produces lower entropy distributions corresponding to higher confidences for next-token choices. This is evidenced by the mass of the red PRU curve lying in the lower entropy ranges compared to the blue LSTM's curve. The PRU can produce confident decisions in part because more information is encoded in the higher dimensional context vectors.

Variance in word embeddings: The PRU has the ability to model individual words at different resolutions through the pyramidal transform; this provides multiple paths for the gradient to the embedding layer (similar to multi-task learning) and improves the flow of information. When considering the embeddings by part of speech, we find that the pyramid level 1 embeddings exhibit higher variance than the LSTM across all POS categories (Fig-

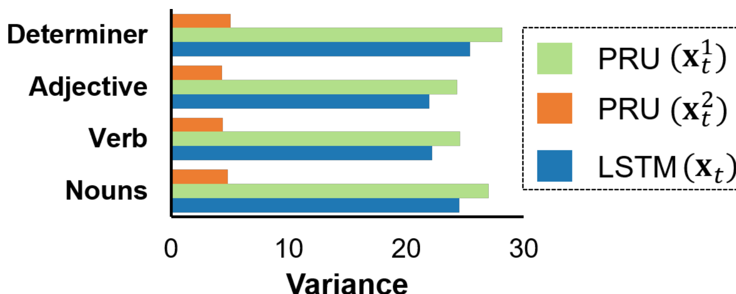
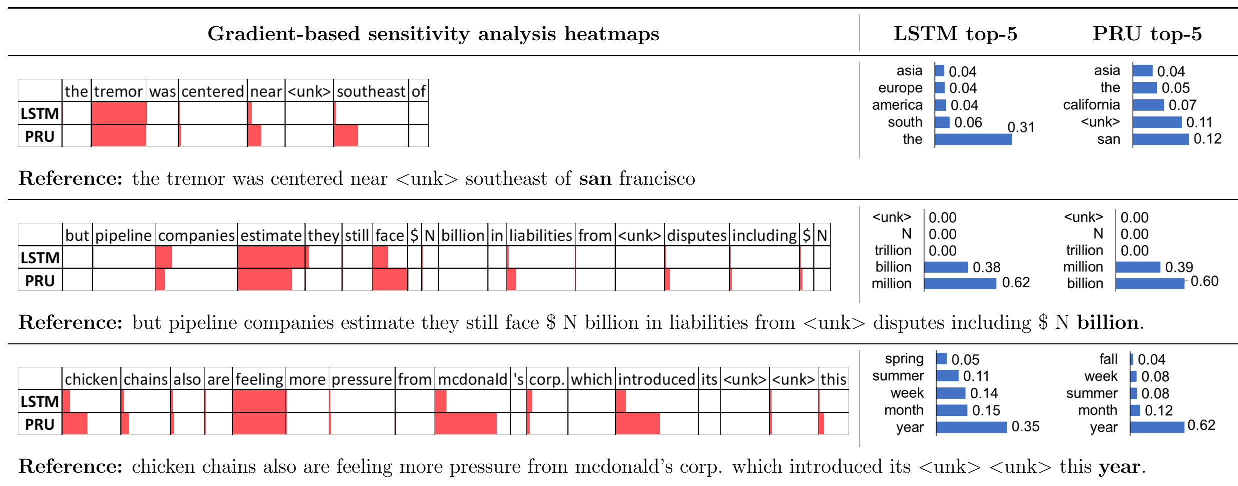


Figure 4.4: Variance of learned word embeddings for different categories of words on the PTB validation set. We compute the variance of a group of embeddings as the average squared euclidean distance to their mean. Higher variance may allow for better intra-category distinctions. The PRU with pyramid levels 1 and 2 is shown.

ure 4.4) and that pyramid level 2 embeddings show extremely low variance[1]. We hypothesize that the LSTM must encode both coarse group similarities and individual word differences into the same vector space, reducing the space between individual words of the same category. The PRU can rely on the subsampled embeddings to account for coarse-grained group similarities, allowing for finer individual word distinctions in the embedding layer. This hypothesis is strengthened by the entropy results described above: a model which can make finer distinctions between individual words can more confidently assign probability mass. A model that cannot make these distinctions, such as the LSTM, must spread its probability mass across a larger class of similar words.

Gradient-based analysis: Saliency analysis using gradients help identify relevant words in a test sequence that contribute to the prediction [109–111]. These approaches compute the relevance as the squared norm of the gradients obtained through back-propagation. Figure 4.5a visualizes the heatmaps for different sequences. PRUs, in general, give more relevance to contextual words than LSTMs, such as southeast (sample 1), face (sample 2),

[1]POS categories are computed using NLTK toolkit.



(a)

(b)

Figure 4.5: Qualitative comparison between the LSTM and the PRU. **(a)** Gradient-based saliency analysis along with top-5 predicted words. Salience score is proportional to cell coverage in **red**. **(b)** Gradients during back-propagation (**x-axis**: word vector dimensions, **y-axis**: test sequence). For computing the gradients for a given test sequence in (b), the top-1 predicted word was used as the *true* predicted word.

and introduced (sample 3), which help in making more confident decisions. Furthermore, when gradients during back-propagation are visualized (Figure 4.5b), we find that PRUs have better gradient coverage than LSTMs, suggesting PRUs use more features than LSTMs that contributes to the decision. This also suggests that PRUs update more parameters at each iteration which results in faster training. The language model in [84] takes 500 and 750 epochs to converge with the PRU and the LSTM as a recurrent unit, respectively.

4.4.4 *Ablation studies*

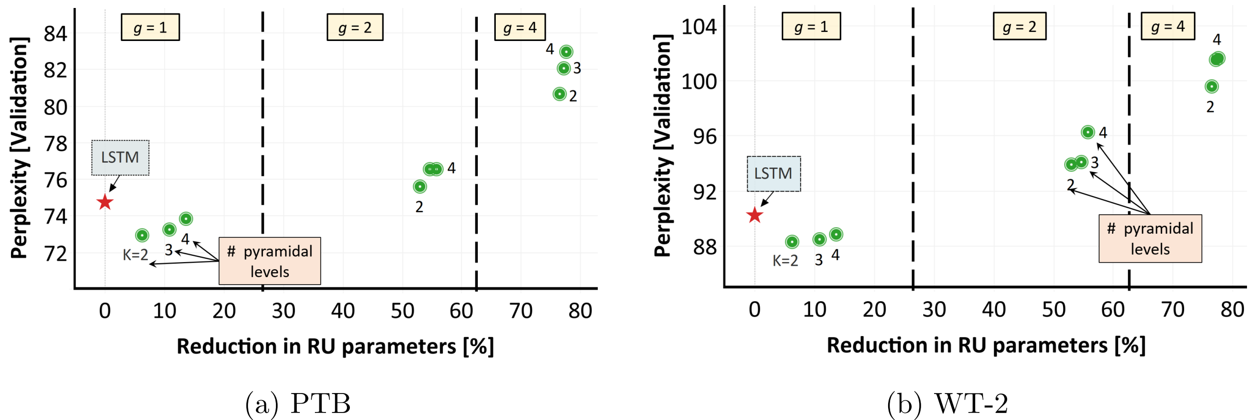
In this section, we provide a systematic analysis of our design choices. Our training methodology is the same as described in Section 4.4.1 with the standard dropouts. For a thorough understanding of our design choices, we use a language model with a single layer of PRU and fix the size of embedding and hidden layers to 600. The word-level perplexities are reported on the validation sets of the PTB and the WT-2 datasets.

Pyramidal levels K and groups g: The two hyper-parameters that control the trade-off between performance and number of parameters in PRUs are the number of pyramidal levels K and groups g. Figure 4.6 provides a trade-off between perplexity and recurrent unit (RU) parameters[2].

Variable K and fixed g: When we increase the number of pyramidal levels K at a fixed value of g, the performance of the PRU drops by about 1 to 4 points while reducing the total number of recurrent unit parameters by up to 15%. We note that the PRU with $K = 4$ at $g = 1$ delivers a similar performance to the LSTM while learning about 15% fewer recurrent unit parameters.

Fixed K and variable g: When we vary the value of g at a fixed number of pyramidal levels K, the total number of recurrent unit parameters decreases significantly with a minimal impact on the perplexity. For example, a PRU with $K = 2$ and $g = 4$ learns 77% fewer recurrent unit parameters, while its perplexity (lower is better) increases by about 12% in

[2]Total parameters = Embedding layer parameters + Recurrent unit (RU) parameters



(a) PTB

(b) WT-2

Figure 4.6: Impact of number of groups g and pyramidal levels K on the perplexity. Reduction in recurrent-unit (RU) parameters is computed with respect to LSTM. Lower perplexity value represents better performance.

comparison to LSTMs. Moreover, the decrease in number of parameters at a higher value of g enables PRUs to learn the representations in a high-dimensional space with better generalizability (Table 4.1).

Transformations: Table 4.2 shows the impact of different transformations of the input vector $\mathbf{x}_t$ and the context vector $\mathbf{h}_{t-1}$. We make the following observations: (1) Using the pyramidal transformation for the input vectors improves the perplexity by about 1 point on both the PTB and WT-2 datasets, while reducing the number of recurrent unit parameters by about 14% (see R1 and R4). We note that the performance of the PRU drops by up to 1 point when residual connections are not used (R4 and R6). (2) Using the grouped linear transformation for context vectors reduces the total number of recurrent unit parameters by about 75% while the performance drops by about 11% (see R3 and R4). When we use the pyramidal transformation instead of the linear transformation, the performance drops by up to 2% while there is no significant drop in the number of parameters (R4 and R5).

	Transformations		PTB		WT-2	
	Transformations		**PPL**	**# Params**	**PPL**	**# Params**
	Context	Input		(total/RU)		(total/RU)
R1	LT	LT	74.80	8.8/2.9	89.30	22.8/2.9
R2	GLT	GLT	84.38	**6.5/0.5**	104.13	**20.46/0.5**
R3	GLT	PT	82.67	6.6/0.64	99.57	20.6/0.64
R4	LT	PT	**74.18**	8.5/2.5	**88.31**	22.5/2.5
R5	PT	PT	75.80	8.1/2.1	90.56	22.1/2.1
R6	LT	PT†	75.61	8.5/2.5	89.27	22.5/2.5

Table 4.2: Impact of different transformations used for processing input and context vectors (LT - linear transformation, PT - pyramidal transformation, and GLT - grouped linear transformation). Here, † represents that PT was used without residual connection, PPL represents word-level perplexity (lower is better), and the number of parameters are in the millions. We used $K = g = 4$ in our experiments.

Dataset	**Sub-sampling method**			
	Skip	**Max pool**	**Avg. Pool**	**Convolution**
PTB	75.12	87.6	**73.86**	81.56
WT-2	89.24	107.63	**88.88**	93.16

Table 4.3: Impact of different sub-sampling methods on the word-level perplexity (lower is better). We used $g = 1$ and $K = 4$ in our experiments.

Subsampling: We set sub-sampling kernel κ (Eq. 4.3) with stride $s = 2$ and size of 3 ($e = 1$) in four different ways: (1) *Skip:* We skip every other element in the input vector. (2) *Convolution:* We initialize the elements of κ randomly from a normal distribution and learn them during training the model. We limit the output values to lie between -1 and 1 using a *tanh* activation function to make training stable. (3) *Avg. pool:* We initialize the elements of κ to $\frac{1}{3}$. (4) *Max pool:* We select the maximum value in the kernel window κ.

Table 4.3 compares the performance of the PRU with different sampling methods. Average pooling performs the best, while skipping give comparable performance. Both of these methods enable the network to learn richer word representations while representing the input vector in different forms, thus delivering higher performance. Surprisingly, a convolution-based sub-sampling method does not perform as well as the averaging method. The *tanh* function used after convolution limits the range of output values which are further limited by the LSTM gating structure, thereby impeding in the flow of information inside the cell. Max pooling forces the network to learn representations from high-magnitude elements, thus distinguishing features between elements vanishes, resulting in poor performance.

4.5 Summary

We introduce the Pyramidal Recurrent Unit, which better models contextual information by admitting higher-dimensional representations with good generalizability. When applied to the task of language modeling, PRUs improve perplexity across several settings, including recent state-of-the-art systems. Our analysis shows that the PRU improves the flow of gradient and expands the word embedding subspace, resulting in more confident decisions. Our source code is available at https://github.com/sacmehta/PRU.

Chapter 5

DEEP AND LIGHT-WEIGHT TRANSFORMER FOR SEQUENCE MODELING

5.1 Introduction

The attention-based transformer networks of Vaswani et al. [17] are gaining interest in sequence modeling tasks, including language modeling and machine translation. To improve performance, models are often scaled to be either wider, by increasing the dimension of hidden layers, or deeper, by stacking more transformer blocks. For example, T5 [7] uses a dimension of 65K, and GPT-3 [112] uses 96 transformer blocks. However, such scaling increases the number of network parameters significantly (e.g., T5 and GPT-3 have 11 billion and 175 billion parameters, respectively) and complicates learning, i.e., these models either require very large training corpora [2, 7, 112] or careful regularization [84, 113, 114]. In this chapter, we introduce a new parameter-efficient attention-based architecture that can be easily scaled to be both wide and deep.

Our `De`ep and `Ligh`t-weight `T`ransformer architecture, `DeLighT`, extends the transformer architecture of Vaswani et al. [17] and delivers similar or better performance with significantly fewer parameters and operations. At the heart of `DeLighT` is the `DeLighT` transformation that uses the group linear transformations (GLTs; Section 4.3.2) with an expand-reduce strategy for varying the width and depth of the `DeLighT` block efficiently. Since GLTs are local by nature, the `DeLighT` transformation uses feature shuffling, which is analogous to channel shuffling in convolutional networks [48], to share information between different groups. Such wide and deep representations facilitate replacing the multi-head attention and feed-forward layers in transformers with single headed attention and light-weight feed-forward layers, reducing total network parameters and operations. Importantly, unlike transformers,

the `DeLighT` transformation decouples the depth and width from the input size, allowing us to allocate parameters more efficiently across blocks by using shallower and narrower `DeLighT` blocks near the input and deeper and wider `DeLighT` blocks near the output.

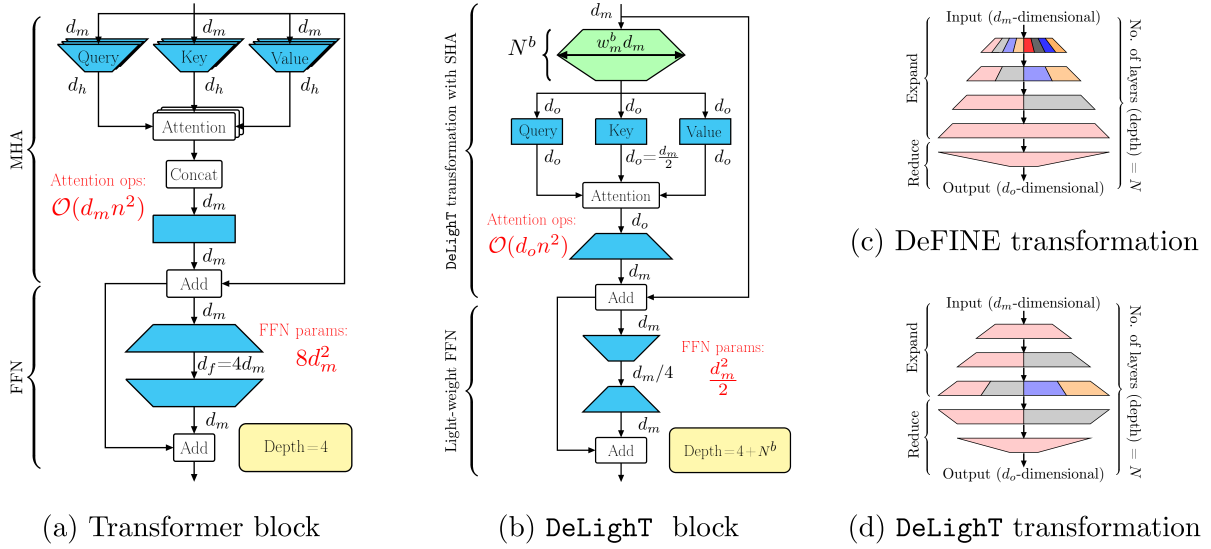
We demonstrate that `DeLighT` models achieve similar or better performance than transformer models with significantly fewer parameters and operations on two common sequence modeling tasks: (i) machine translation and (ii) language modeling. On the low resource WMT'16 En-Ro machine translation dataset, `DeLighT` attains transformer performance using 2.8× fewer parameters. On the high resource WMT'14 En-Fr dataset, `DeLighT` delivers better performance (+0.4 BLEU score) with 1.8× fewer parameters than baseline transformers. Similarly, on language modeling, `DeLighT` matches the performance of Transformer-XL [115] with 1.5× fewer parameters on the WikiText-103 dataset. Our source code is available at https://github.com/sacmehta/delight.

5.2 Related Work

Improving transformers: Several methods have been introduced to improve the transformer architecture. The first line of research addresses the challenge of computing self attention on long input sequences [116–118]. These methods can be combined with our architecture. The second line of research focuses on explaining multi-head attention [119, 120]. They show that increasing the number of transformer heads can lead to redundant representations [121, 122] and using fixed attention heads with predefined patterns [123] or synthetic attention matrices [124] improves performance. The third line of research focuses on improving transformers by learning better representations [125–127]. These works aim to improve the expressiveness of transformers using different transformations – for example, using convolutions [89, 125], gated linear units [128], or multi-branch feature extractors [126, 127]. Our work falls into this category. Unlike previous work, we show that it is possible to efficiently allocate parameters, both at the block-level using the `DeLighT` transformation and across blocks using block-wise scaling.

Model scaling: Model scaling is a standard method to improve the performance of sequence models [2, 7, 17, 57, 112, 129, 130]. Model dimensions are increased in width-wise scaling, [2, 17] while more blocks (e.g., Transformer blocks) are stacked in depth-wise scaling [112, 130, 131]. In both cases (and their combination), parameters inside each block of the network are the same, which may lead to a sub-optimal solution. To further improve the performance of sequence models, we introduce *block-wise scaling* that allows for variably-sized blocks and efficient allocation of parameters in the network. Our results show that (1) shallower and narrower `DeLighT` blocks near the input and deeper and wider `DeLighT` blocks near the output deliver the best performance, and (2) models with block-wise scaling coupled with model scaling achieve better performance compared to model scaling alone. We note that convolutional neural networks (CNNs) also learn shallower and narrower representations near the input and deeper and wider representations near the output. Unlike CNNs (e.g., ResNet [41]) that perform a fixed number of operations at each convolutional layer, the proposed block-wise scaling uses a variable number of operations in each layer and block.

Improving sequence models: There is also significant recent work on other related methods for improving sequence models, including (1) improving accuracy using better token-level representations – for example, using BPE [132], adaptive inputs [133] and outputs [134], and DeFINE [40], and (2) improving efficiency – for example, using compression [135, 136], pruning [32, 137], and distillation [60, 138]. The closest to this work is the DeFINE transformation that we introduced to learn deep token-level representations. The DeFINE transformation also learns representations using an expand-reduce strategy. The key difference between the DeFINE transformation (Figure 5.1c) and the `DeLighT` transformation (Figure 5.1d) is that the `DeLighT` transformation more efficiently allocates parameters within expansion and reduction layers. Unlike DeFINE, which uses fewer groups in group linear transformations to learn wider representations, the `DeLighT` transformation uses more groups to learn wider representations with fewer parameters. The `DeLighT` transformation achieves comparable performance to the DeFINE transformation but with significantly fewer parameters.



(a) Transformer block (b) DeLighT block (c) DeFINE transformation (d) DeLighT transformation

Figure 5.1: **(a, b)** Block-wise comparison between the standard transformer block of Vaswani et al. [17] and the DeLighT block. In the DeLighT transformation, the number of operations for computing attention are reduced by half, while the number of parameters (and operations) in the FFN are reduced by 16×. Transformations with learnable parameters (Linear and DeLighT) are shown in color. The shapes of linear transformations indicate their operation (expansion, reduction, etc.). **(c, d)** compares the DeFINE transformation [40] with the DeLighT transformation. Compared to the DeFINE transformation, the DeLighT transformation uses group linear transformations (GLTs) with more groups to learn wider representations with fewer parameters. Different colors are used to show groups in GLTs. For simplicity, feature shuffling is not shown in (d). Here, FFN means feed-forward network, MHA means multi-headed attention, and SHA means single-head attention.

5.3 DeLighT: Deep and Light-weight Transformer

A standard transformer block (Figure 5.1a) consists of multi-head attention that uses a query-key-value decomposition to model relationships between sequence tokens, and a feed forward network (FFN) to learn wider representations. Multi-head attention obtains query

$\mathbf{Q}$, key $\mathbf{K}$, and value $\mathbf{V}$ by applying three projections to the input, each consisting of h linear layers (or heads) that map the d_m-dimensional input into a d_h-dimensional space, where $d_h = d_m/h$ is the head dimension. The FFN consists of two linear layers, where the first expands the dimensions from d_m to d_f and the second reduces the dimensions from d_f to d_m. The depth of a transformer block is 4, consisting of (1) three parallel branches for queries, keys, and values, (2) a fusion layer that combines the output of multiple heads, and (3) two sequential linear layers in the FFN. In general, transformer-based networks sequentially stack transformer blocks to increase network capacity and depth.

We extend the transformer architecture and introduce a deep and light-weight transformer, `DeLighT`. Our model uses a deep and light-weight expand-reduce transformation, the `DeLighT` transformation (Section 5.3.1), that enables learning wider representations efficiently. It also enables replacing multi-head attention and feed-forward network (FFN) layers with single-head attention and a light-weight FFN (Section 5.3.2). The `DeLighT` transformation decouples the attention dimensions from the depth and width, allowing us to learn representations efficiently using block-wise scaling instead of uniform stacking of transformer blocks (Section 5.3.3).

5.3.1 DeLighT Transformation

The `DeLighT` transformation maps a d_m dimensional input vector into a high dimensional space (expansion) and then reduces it down to a d_o dimensional output vector (reduction) using N layers of the group linear transformations (Section 4.3.2), as shown in Figure 5.1d. During these expansion and reduction phases, the `DeLighT` transformation uses group linear transformations (GLTs), because they learn local representations by deriving the output from a specific part of the input and are more efficient than linear transformations. To learn global representations, the `DeLighT` transformation shares information between different groups in the group linear transformation using feature shuffling, analogous to channel shuffling in convolutional networks [48].

A standard approach to increase the expressivity and capacity of transformers is to in-

crease the input dimensions, d_m. However, increasing d_m linearly also increases the number of operations in multi-head attention ($\mathcal{O}(n^2 d_m)$, where n is the sequence length) in a standard transformer block (Figure 5.1a). In contrast, to increase the expressivity and capacity of the `DeLighT` block, we increase the depth and width of its intermediate `DeLighT` transformations using expansion and reduction phases. This enables us to use smaller dimensions for computing attention, requiring fewer operations.

Formally, the `DeLighT` transformation is controlled by five configuration parameters: (1) number of GLT layers N, (2) width multiplier w_m, (3) input dimension d_m, (4) output dimension d_o, and (5) maximum groups g_{max} in a GLT. In the expansion phase, the `DeLighT` transformation projects the d_m-dimensional input to a high-dimensional space, $d_{max} = w_m d_m$, linearly using $\lceil \frac{N}{2} \rceil$ layers. In the reduction phase, the `DeLighT` transformation projects the d_{max}-dimensional vector to a d_o-dimensional space using the remaining $N - \lceil \frac{N}{2} \rceil$ GLT layers. Mathematically, we define the output $\mathbf{Y}$ at each GLT layer l as:

$$\mathbf{Y}^l = \begin{cases} \mathcal{F}\left(\mathbf{X}, \mathbf{W}^l, \mathbf{b}^l, g^l\right), & l = 1 \\ \mathcal{F}\left(\mathcal{H}\left(\mathbf{X}, \mathbf{Y}^{l-1}\right), \mathbf{W}^l, \mathbf{b}^l, g^l\right), & \text{Otherwise} \end{cases} \tag{5.1}$$

where $\mathbf{W}^l = \left\{\mathbf{W}_1^l, \cdots, \mathbf{W}_{g^l}^l\right\}$ and $\mathbf{b}^l = \left\{\mathbf{b}_1^l, \cdots, \mathbf{b}_{g^l}^l\right\}$ are the learnable weights and biases of group linear transformation $\mathcal{F}$ with g^l groups at the l-th layer. Briefly, the $\mathcal{F}$ function takes the input $\mathbf{X}$ $\left(\text{or } \mathcal{H}\left(\mathbf{X}, \mathbf{Y}^{l-1}\right)\right)$ and splits it into g^l non-overlapping groups such that $\mathbf{X} = \left\{\mathbf{X}_1, \cdots, \mathbf{X}_{g^l}\right\}$. The function $\mathcal{F}$ then linearly transforms each $\mathbf{X}_i$ with weights $\mathbf{W}_i^l$ and bias $\mathbf{b}_i^l$ to produce output $\mathbf{Y}_i^l = \mathbf{X}_i \mathbf{W}_i^l + \mathbf{b}_i^l$. The outputs of each group $\mathbf{Y}_i^l$ are then concatenated to produce the output $\mathbf{Y}^l$. The function $\mathcal{H}$ first shuffles the output of each group in $\mathbf{Y}^{l-1}$ and then combines it with the input $\mathbf{X}$ using the input mixer connection of Mehta et al. [40] to avoid vanishing gradient problems. Figure 5.2 visualizes the expansion phase in the `DeLighT` transformation with the group linear transformation, feature shuffling, and the input mixer connection.

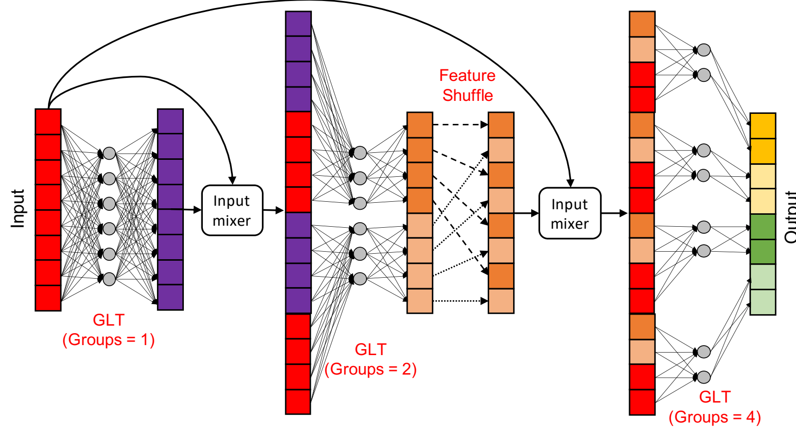


Figure 5.2: Illustration of the expansion phase in the `DeLighT` transformation that uses GLTs, feature shuffling, and an input mixer connection, to learn deeper and wider representations efficiently. For illustrative purposes, we have used the same input and output dimensions.

The number of groups at the l-th GLT in `DeLighT` transformation are computed as:

$$g^l = \begin{cases} \min(2^{l-1}, g_{max}), & 1 \leq l \leq \lceil N/2 \rceil \\ g^{N-l}, & \text{Otherwise} \end{cases} \tag{5.2}$$

In our experiments, we use $g_{max} = \lceil \frac{d_m}{32} \rceil$, so that each group has at least 32 input elements.

5.3.2 DeLighT block

Figure 5.1b shows how we integrate `DeLighT` transformation into the transformer block to improve its efficiency. The d_m-dimensional inputs are fed to the `DeLighT` transformation to produce d_o-dimensional outputs, where $d_o < d_m$. These d_o-dimensional outputs are then fed into a single head attention, followed by a light-weight FFN to model their relationships.

`DeLighT` layer and single head attention: Let us assume we have a sequence of n input tokens, each of dimensionality d_m. These n, d_m-dimensional inputs are first fed to the `DeLighT` transformation to produce n, d_o-dimensional outputs, where $d_o < d_m$. These n, d_o-dimensional outputs are then projected simultaneously using three linear layers to produce

d_o-dimensional queries $\mathbf{Q}$, keys $\mathbf{K}$, and values $\mathbf{V}$. We then model contextual relationships between these n tokens using scaled dot-product attention (Eq. 5.3). To enable the use of residual connections [41], the d_o-dimensional outputs of this attention operation are linearly projected into a d_m-dimensional space.

$$\text{Attention}(\mathbf{K}, \mathbf{Q}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_o}}\right)\mathbf{V} \tag{5.3}$$

We hypothesize that the ability of `DeLighT` to learn wider representations allows us to replace multi-head attention with single-head attention. The computational costs for computing attention in the standard transformer and the `DeLighT` block are $\mathcal{O}(d_m n^2)$ and $\mathcal{O}(d_o n^2)$ respectively, where $d_o < d_m$. Therefore, the `DeLighT` block reduces the cost for computing attention by a factor of d_m/d_o. In our experiments, we used $d_o = d_m/2$, thus requiring $2\times$ fewer multiplication-addition operations than transformers for attention.

Light-weight FFN: Similar to FFNs in transformers, this block also consists of two linear layers. Since the `DeLighT` block has already incorporated wider representations using the `DeLighT` transformation, it allows us to invert the functionality of FFN layers in the transformer. The first layer reduces the dimensionality of the input from d_m to d_m/r, while the second layer expands the dimensionality from d_m/r to d_m, where r is the reduction factor (see Figure 5.1b). Our light-weight FFN reduces the number of parameters and operations in the FFN by a factor of rd_f/d_m. In the standard transformer, the FFN dimensions are expanded by a factor of 4.[1] In our experiments, we use $r = 4$. Thus, the light-weight FFN reduces the number of parameters in the FFN by $16\times$.

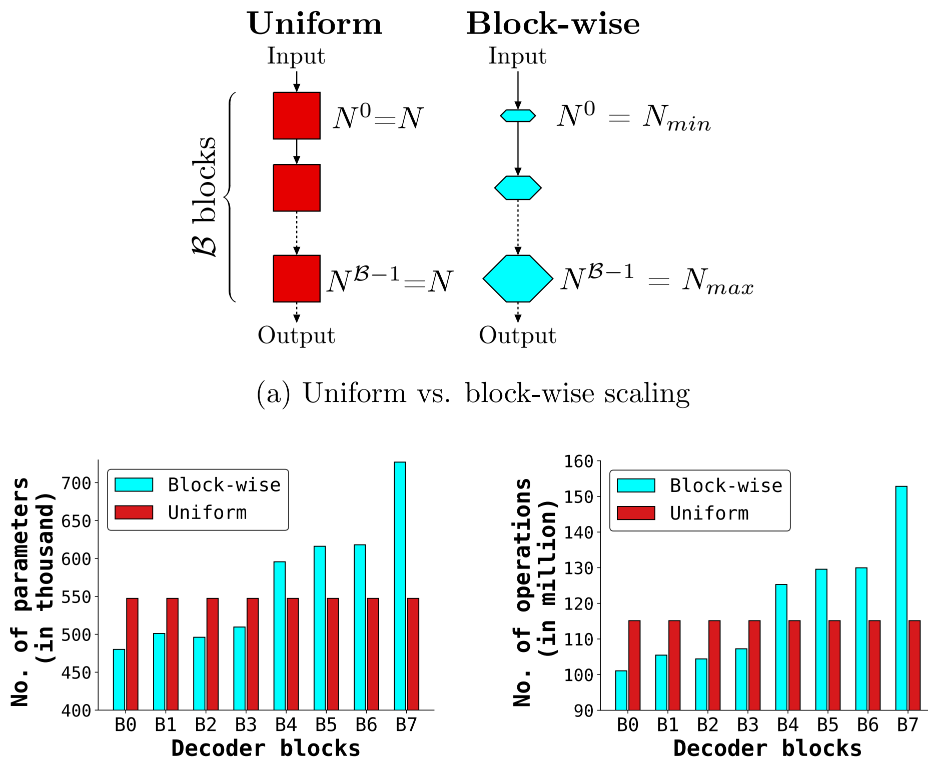
Block depth: The `DeLighT` block stacks (1) a `DeLighT` transformation with N GLTs, (2) three parallel linear layers for key, query, and value, (3) a projection layer, and (4) two linear layers of a light-weight FFN. Thus, the depth of the `DeLighT` block is $N + 4$. Compared to the standard transformer block (depth is 4), the `DeLighT` block is deeper.

---

[1] Transformer-base uses d_m=512 and d_f=2048 while Transformer-large uses d_m=1024 and d_f=4096.

5.3.3 *Block-wise scaling*

Standard methods for improving the performance of sequence models include increasing the model dimensions (width scaling), stacking more blocks (depth scaling), or both. However, such scaling is not very effective on small datasets. For example, when a Transformer-Base



(a) Uniform vs. block-wise scaling

(b) Distribution of parameters and operations within each block

Figure 5.3: **Block-wise scaling** in (a) allows us to efficiently allocate parameters and operations across blocks, leading to shallower and narrower `DeLighT` blocks near the input and deeper and wider `DeLighT` blocks near the output. In (b), `DeLighT` networks with both uniform (N=N_{min}=N_{max}=8) and block-wise (N_{min}=4, N_{max}=8) scaling have about 16.7 M parameters and perform 3.5 B operations (computed for a sequence length of $n = 30$); however, the `DeLighT` network with block-wise scaling delivered 2 points better perplexity.

($d_m = 512$) network is replaced with a Transformer-Large ($d_m = 1024$) network on the WMT'16 En-Ro corpus, the number of parameters increases by $\sim 4\times$, while the performance does not change appreciably (BLEU: 34.28 vs. 34.35). We hypothesize that this happens because scaling model width and depth allocates parameters uniformly across blocks, which may lead to learning redundant parameters. To create deep and wide networks, we extend model scaling to the block level. Figure 5.3 compares uniform scaling with block-wise scaling.

Scaling the DeLighT block: The `DeLighT` block learns deep and wide representations using the `DeLighT` transformation, whose depth and width are controlled by two configuration parameters: the number of GLT layers N and the width multiplier w_m, respectively (Figure 5.3a). These configuration parameters allow us to increase the number of learnable parameters inside the `DeLighT` block independent of the input d_m and output d_o dimensions. Such calibration is not possible with the standard transformer block, because their expressiveness and capacity are a function of the input (input dimension = number of heads $\times$ head dimension). Here, we introduce block-wise scaling that creates a network with variably-sized `DeLighT` blocks, allocating shallower and narrower `DeLighT` blocks near the input and deeper and wider `DeLighT` blocks near the output.

To do so, we introduce two network-wide configuration parameters: minimum N_{min} and maximum N_{max} number of GLTs in a `DeLighT` transformation. For the b-th `DeLighT` block, we compute the number of GLTs N^b and the width multiplier w_m^b in a `DeLighT` transformation using linear scaling (Eq. 5.4). With this scaling, each `DeLighT` block has a different depth and width (Figure 5.3a).

$$N^b = N_{min} + \frac{(N_{max} - N_{min})\ b}{\mathcal{B} - 1}, \quad w_m^b = w_m + \frac{(N_{max} - N_{min})\ b}{N_{min}(\mathcal{B} - 1)}, \quad 0 \leq b \leq \mathcal{B} - 1 \qquad (5.4)$$

Here, $\mathcal{B}$ denotes the number of `DeLighT` blocks in the network. We add superscript b to the number of GLT layers N and width multiplier w_m to indicate that these parameters are for the b-th block.

Network depth: The depth of a transformer block is fixed, i.e., 4. Therefore, previous works [7, 112, 131] have associated the depth of transformer-based networks with the number of transformer blocks. In `DeLighT`, we present a different perspective to learn deeper representations, wherein each block is variably-sized. To compute the network depth, we use the standard definition across different domains, including computer vision (e.g., ResNet of He et al. 41) and theoretical machine learning [139]. These works measure network depth as the number of sequential learnable layers (e.g., convolution, linear, or group linear). Similarly, the depth of `DeLighT` and transformer networks with $\mathcal{B}$ blocks is $\sum_{b=0}^{\mathcal{B}-1}(N^b + 4)$ and $4\mathcal{B}$, respectively.

5.4 *Experimental results*

We evaluate the performance of `DeLighT` on two standard sequence modeling tasks: (1) machine translation (Section 5.4.1) and (2) language modeling (Section 5.4.2).

5.4.1 Machine Translation

Datasets and evaluation: We benchmark `DeLighT` models on four datasets: (1) IWSLT'14 German-English (De-En), (2) WMT'16 English-Romanian (En-Ro), (3) WMT'14 English-German (WMT'14 En-De), and (4) WMT'14 English-French (WMT'14 En-Fr). For the IWSLT'14 De-En dataset, we replicate the setup of Wu et al. [125] and Edunov et al. [140], which use 160K/7K/7K sentence pairs for training, validation, and testing with a joint BPE vocabulary of about 10K tokens. For the WMT'14 English-German (En-De) dataset, we follow the setup of Vaswani et al. [17]. The dataset has 3.9M/39K/3K sentence pairs for training, validation, and testing respectively with a joint BPE vocabulary size of 44K.[2] For the WMT'14 English-French (En-Fr) dataset, we replicate the setup of Gehring et al. [89], which uses 36M/27K/3K sentence pairs for training, validation, and testing respectively with a joint BPE vocabulary size of 44K. The performance is evaluated in terms of *BLEU*

[2]We use training and validation data that is compatible with the Tensor2Tensor library [141] in order to have fair comparisons with recent works (e.g., Evolved Transformer).

[142] (higher is better) on the test set. We follow Wu et al. [125] for beam search related hyper-parameters.

Architecture: We follow the symmetric encoder-decoder architecture of Vaswani et al. [17] with sinusoidal positional encodings. Both the encoder and the decoder have $\mathcal{B}$ `DeLighT` blocks. Decoder blocks are identical to the encoder blocks (Figure 5.1b), except that they have an additional source-target single-head attention unit before the light-weight FFN. In the source-target single-head attention unit, keys and values are projections over the encoder output (full details in Appendix B.1). In our experiments, we use $w_m = 2$, $N_{min} = 4$, and $N_{max} = 8$ for WMT'16 En-Ro, WMT'14 En-De, and WMT'14 En-Fr; resulting in 222 layer deep `DeLighT` networks. For IWSLT'14 De-En, we used $w_m = 1$, $N_{min} = 3$, and $N_{max} = 9$ for IWSLT'14 De-En; resulting in a 289-layer deep network. For simplicity, we set $\mathcal{B} = N_{max}$. We use a learnable look-up table that maps every token in the vocabulary to a 128-dimensional vector. We implement our models using Fairseq [143] and use their provided scripts for data pre-processing, training, and evaluation.

Training: For IWSLT'14 De-En models, we follow the setup of [125] and train all our models for 50K iterations with a batch size of 4K tokens on a single NVIDIA GTX 1080 GPU. For WMT'16 En-Ro, we follow the training setup of [144] and train models for 100K iterations on 16 NVIDIA Tesla V100 GPUs with an effective batch size of 64K tokens. For WMT'14 En-De and WMT'14 En-Fr, we follow the training set-up of [125] and train our models on 16 V100 GPUs for 30K and 50K iterations, respectively. We use Adam [145] to minimize cross entropy loss with a label smoothing value of 0.1 during training. For a fair comparison, we trained baseline transformer models using the same training set-up.

Comparison with baseline transformers: The performance of `DeLighT` with the baseline transformers of Vaswani et al. [17] on different corpora is shown in Table 5.1. `DeLighT` delivers better performance with fewer parameters than transformers, across different cor-

pora. Specifically, on low-resource (WMT'16 En-Ro) and high resource (WMT'14 En-De and WMT'14 En-Fr) corpora, `DeLighT` delivers similar or better performance with 2.8× and 1.8× fewer parameters, respectively. When the number of parameters are increased, `DeLighT` outperforms transformers. For example, on WMT'14 En-Fr dataset, `DeLighT` is 3.7× deeper than transformers and improves its BLEU score by 1.3 points yet with 13 million fewer parameters and 3 billion fewer operations (see Table 5.2).

Particularly interesting are the performance comparisons of `DeLighT` with the baseline

	IWSLT'14 De-En				**WMT'16 En-Ro**			
Model	**# Params**	**Ratio**	**BLEU**	**Δ BLEU**	**# Params**	**Ratio**	**BLEU**	**Δ BLEU**
Transformer [17]	–	–	34.4†	–	62 M	–	34.3‡	–
Transformer (Our impl.)	42 M	1.0×	34.3	–	62 M	1.0×	34.3	–
`DeLighT`	14 M	0.3×	33.8	-0.5	22 M	0.35×	34.3	0.0
`DeLighT`	30 M	0.7×	**35.3**	**+1.0**	53 M	0.85×	**34.7**	**+0.4**

(a) Results on small corpora

	WMT'14 En-De				**WMT'14 En-Fr**			
Model	**# Params**	**Ratio**	**BLEU**	**Δ BLEU**	**# Params**	**Ratio**	**BLEU**	**Δ BLEU**
Transformer [17]	62 M	–	27.3	–	–	62 M	38.1	–
Transformer (Our impl.)	67 M	1.0×	27.7	–	67 M	1.0×	39.2	–
`DeLighT`	37 M	0.55×	27.6	-0.1	37 M	0.55×	39.6	+0.4
`DeLighT`	54 M	0.80×	**28.0**	**+0.3**	54 M	0.80×	**40.5**	**+1.3**

(b) Results on large corpora

Table 5.1: **Comparison with baseline transformers on machine translation corpora**. `DeLighT` models require significantly fewer parameters to achieve similar performance. Here, † and ‡ indicate the best reported transformer baselines from Wu et al. [125] and Ghazvininejad et al. [144], respectively.

transformers of Vaswani et al. [17] and its neural search variant, i.e., Evolved Transformer of So et al. [127], at two different parametric settings on WMT'14 En-De corpora in Figure 5.4. For small models (< 10 M parameters), `DeLighT` models delivers better performance and for attaining the same performance as these models, `DeLighT` models requires fewer parameters.

	Depth	# Params	# MACs	BLEU
Transformer	60	67 M	11.1 B	39.2
`DeLighT`	222	37 M	5.6 B	39.6
`DeLighT`	222	54 M	8.1 B	40.5

Table 5.2: **`DeLighT` networks are deep, light-weight and efficient** as compared to transformers. The BLEU score is reported on the WMT'14 En-Fr dataset. To compute network depth, we count the number of sequential layers in the network (Section 5.3.3). We used 20 source and 20 target tokens for computing multiplication-addition operations (MACs). See Appendix B.3 for details.

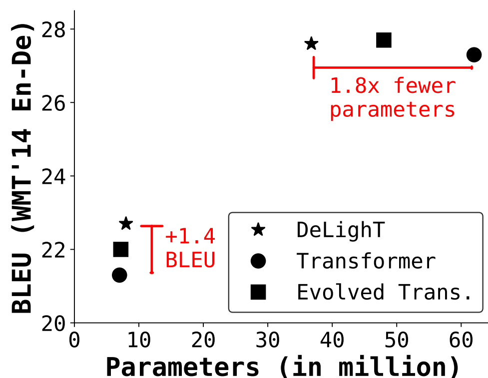


Figure 5.4: Comparison of `DeLighT` with Transformers and Evolved Transformers at two different settings, on the WMT'14 En-De corpus: (1) the number of parameters is the same and (2) the performance is the same.

Model	# Params	BLEU
Transformers [17]	42 M	34.3
Variational Attention [146]	–	33.1
Dynamic convolutions [17]	43 M	**35.2**
Lite Transformer‡ [126]	–	33.6
DeLighT (Ours)	**30 M**	**35.3**

(a) IWSLT'14 De-En

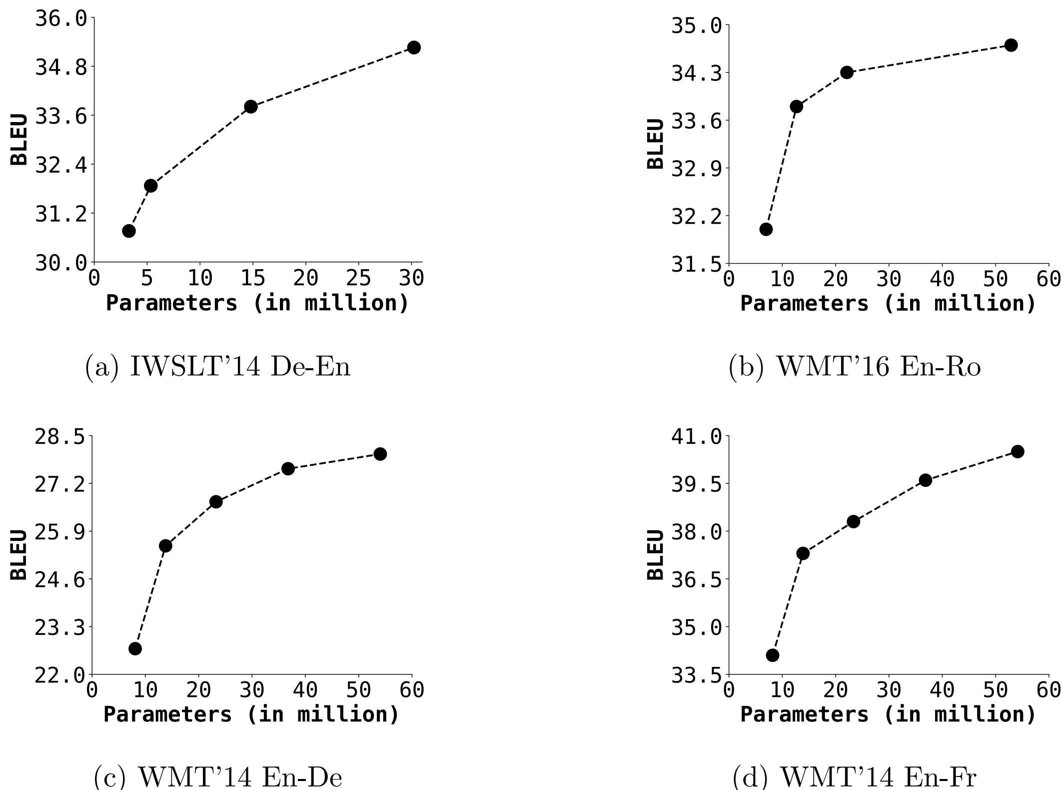
Model	# Params	BLEU
Transformer [17]	62 M	27.3
DLCL [131]	62 M	27.3
Evolved Transformer † [127]	46 M	**27.7**
Lite Transformer‡ [126]	–	26.5
DeLighT (Ours)	**37 M**	**27.6**

(b) WMT'14 En-De

Table 5.3: **Comparison with state-of-the-art methods on machine translation corpora**. DeLighT delivers similar or better performance than state-of-the-art models with fewer parameters. Here, † indicates that the network uses neural architecture search (NAS) and ‡ indicates that full network parameters are not reported.

Comparison with state-of-the-art methods: The performance of most state-of-the-art methods has been evaluated on WMT'14 En-De, while some have also been evaluated on IWSLT'14 De-En. Table 5.3 compares the performance of DeLighT with state-of-the-art methods on these two corpora. DeLighT delivers similar or better performance than existing methods. It is important to note that existing methods have improved baseline transformers with different design choices – for example, the asymmetric encoder-decoder structure [131] and neural architecture search [127]. We believe that DeLighT, in the future, would also benefit from such design choices.

Scaling up DeLighT models: Figure 5.5 shows the performance of DeLighT models improves with an increase in network parameters, suggesting their ability to learn representations across different corpora, including low-resource.



(a) IWSLT'14 De-En

(b) WMT'16 En-Ro

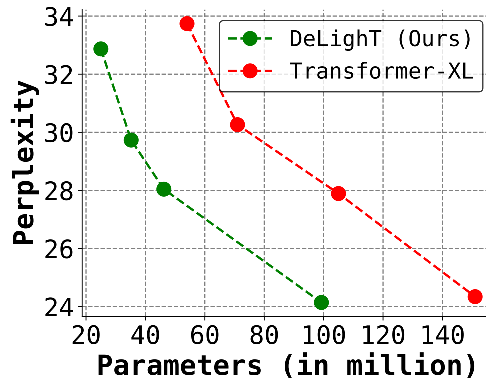
(c) WMT'14 En-De

(d) WMT'14 En-Fr

Figure 5.5: **Scaling up `DeLighT` models.** The performance of `DeLighT` improves with an increase in the number of network parameters, across different corpora, including low-resource (WMT'16 En-Ro).

5.4.2 Language Modeling

Datasets and evaluation: We evaluate on the WikiText-103 dataset [99] that has 103M/ 217K/245K tokens for training, validation, and testing. It has a word-level vocabulary of about 260K tokens. Following recent works [115, 133], we report performance in terms of *perplexity* on the test set (see Section 4.4.1 for the definition of perplexity). The lower value of perplexity means better performance.



(a) DeLighT vs. Transformer-XL

Method	Network Depth	Context Length	# Params (in million)	Perplexity (Test)
LSTM [147]	–	–	–	48.70
LSTM + Neural Cache [147]	–	–	–	40.80
QRNN [148]	–	–	151 M	33.00
Transformer-XL [115]	64	640	151 M	**24.03**
Transformer-XL (Our impl.)†	64	640	151 M	24.34
Transformer-XL (Our impl.)†	64	480	151 M	24.91
DeLighT (Ours)	158	**480**	**99 M**	**24.14**

(b) Comparison with existing methods

Table 5.4: **Results on the WikiText-103 dataset**. Compared to Transformer-XL, DeLighT delivers similar or better performance (lower perplexity) with fewer parameters. †For Transformer-XL, we reproduce results using the official source code. For evaluating Transformer-XL with a context length of 480, we set the mem_len hyper-parameter to 480 in the official evaluation scripts.

Architecture: We use the transformer-based decoder architecture of Baevski and Auli [133] with $\mathcal{B}$ `DeLighT` blocks. We use w_m=2, N_{min}=4, and N_{max}=12. We scale d_m using values $\{384, 512, 784, 1024\}$ for increasing network parameters. For simplicity, we set $\mathcal{B} = N_{max}$. Following standard practice, we use adaptive input [133] as a look-up table and adaptive output [134] as the classification layer with one head (head dimension is 128) and two tails (tail dimensions are 64 and 32). We also share weights between the input and the output layers.

Training: We follow the training setup of Baevski and Auli [133], except that we train our models on 8 NVIDIA Tesla V100 GPUs for 100K iterations with a context length of 512 and an effective batch size of 64K tokens. We use Adam during training and use a context length of 480 during test.

Results: Table 5.4b compares the performance of `DeLighT` with previous methods on WikiText-103. Table 5.4a plots the variation of perplexity with number of parameters for `DeLighT` and Transformer-XL [115] – which outperforms other transformer-based implementations (e.g., Baevski and Auli [133]). Both tables show that `DeLighT` delivers better performance than state-of-the-art methods (including Transformer-XL), and it does this using a smaller context length and significantly fewer parameters, suggesting that the `DeLighT` transformation helps learn strong contextual relationships.

5.5 Analysis and Discussions on Computational Efficiency

Training time and memory consumption: Table 5.5 compares the training time and memory consumption of `DeLighT` with baseline transformers. For an apples-to-apples comparisons, we implemented the Transformer unit without NVIDIA's dedicated CUDA kernel and trained both transformer and `DeLighT` full-precision networks for 30K iterations on 16 NVIDIA V100 GPUs. The transformer and `DeLighT` models took about 37 and 23 hours for training and consumed about 12.5 GB and 14.5 GB of GPU memory, respectively (R1

Row #	Model	# Params (in million)	BLEU (WMT'14 En-Fr)	Training time	Memory (in GB)
R1	Transformer (unoptimized)	67 M	39.2	37 hours	12.5 GB
R2	`DeLighT` (unoptimized)	54 M	40.5	23 hours	14.5 GB
R3	Transformer (w/ Apex optimized)	67 M	39.2	16 hours	11.9 GB
R4	`DeLighT` (w/ optimized grouping)	54 M	40.5	19 hours	11.5 GB

Table 5.5: Comparison with baseline transformers in terms of training speed and memory consumption. In R4, we implemented CUDA kernels for grouping and ungrouping functions only. We expect `DeLighT` to be more efficient with a single and dedicated CUDA kernel for grouping, transformation, feature shuffling, and ungrouping. Memory consumption is measured on a single NVIDIA GP100 GPU (16 GB memory) with a maximum of 4096 tokens per batch and without any gradient accumulation.

vs. R2). When we enabled the dedicated CUDA kernel provided by the APEX library[3] for multi-head attention in Transformers, the training time of the transformer model was reduced from 37 to 16 hours, while we did not observe any significant change in memory consumption. Motivated by this observation, we implemented dedicated CUDA kernels for grouping and ungrouping functions in GLTs. With these changes, training time and GPU memory consumption of `DeLighT` reduced by about 4 hours and 3 GB, respectively. We emphasize that grouping, linear transformation, feature shuffling, and ungrouping, can be implemented efficiently using a single CUDA kernel. In the future, we expect that a dedicated CUDA kernel for these operations would further reduce the memory consumption as well as training/inference time.

Regularization: Table 5.6 shows that `DeLighT` delivers similar performance to baseline transformers, but with fewer parameters and less regularization. This suggests that learning

[3]https://github.com/NVIDIA/apex

Model	Dropout	BLEU (WMT'14 En-De)
Transformer (62 M)	0.10	27.3
Transformer (62 M)	0.30	27.7
`DeLighT` (37 M)	0.05	27.6

Table 5.6: `DeLighT` models require less regularization as compared to baseline transformers.

representations with better transformation functions alleviates the need for dropout.

5.6 Ablations on the task of Language Modeling

Table 5.7 studies the impact of `DeLighT` block parameters on the WikiText-103 dataset, namely (1) minimum number of GLTs N_{min}, (2) maximum number of GLTs N_{max}, (3) width multiplier w_m, and (4) model dimension d_m (see Figure 5.1b). Figure 5.6, Figure 5.7, and Figure 5.8 show the impact of the `DeLighT` transformation, feature shuffling, and the light-weight FFN. Table 5.8 shows the effect of position of the `DeLighT` transformation in the `DeLighT` block, while Figure 5.10 shows the effect of scaling `DeLighT` networks. We choose the WikiText-103 dataset for ablations, because it has very large vocabulary compared to other datasets (267K vs. 30-40K), allowing us to test the ability under large vocabulary sizes. The performance is reported in terms of perplexity (lower is better) on the validation set. In our ablation studies, we used the same settings for training as in Section 5.4.2, except that we train only for 50K iterations.

DeLighT block: Overall, Table 5.7 shows that scaling depth and width using `DeLighT` transformation and block-wise scaling improves performance. We make the following observations:

a) Block-wise scaling (R4, R5) delivers better performance compared to uniform scaling (R1-R3). For instance, `DeLighT` with $N_{min} = 4$ and $N_{max} = 8$ (R4) is 1.25× shallower

Row #	N_{min}	N_{max}	w_m	d_m	Depth	Parameters	MACs	Perplexity
Uniform vs. block-wise scaling								
R1	4	4	2	256	43	14.1 M	2.96 B	56.19
R2	8	8	2	256	115	16.6 M	3.49 B	48.58
R3	8	8	4	256	115	22.1 M	4.64 B	45.10
R4	4	8	2	256	92	16.7 M	3.51 B	46.30
R5	4	12	2	256	158	21.0 M	4.41 B	41.18
Varying depth (N_{min} and N_{max} (Eq. 5.4)								
R6	4	8	2	256	92	16.7 M	3.51 B	46.30
R7	6	8	2	256	102	16.5 M	3.46 B	46.68
R8	4	12	2	256	158	21.0 M	4.41 B	41.18
R9	6	12	2	256	172	20.0 M	4.20 B	42.26
Varying `DeLighT` transformation's width w_m (Eq. 5.4)								
R10	4	12	2	256	158	21.0 M	4.41 B	41.18
R11	4	12	3	256	158	23.8 M	4.99 B	39.92
R12	4	12	4	256	158	27.1 M	5.69 B	39.10
Varying model width d_m								
R13	4	12	2	256	158	21.0 M	4.41 B	41.18
R14	4	12	2	384	158	29.9 M	6.28 B	35.14
R15	4	12	2	512	158	43.8 M	9.20 B	30.81
Deeper and wider near the Input								
R16	12	4	2	256	158	21.0 M	4.41 B	43.10

Table 5.7: **Ablations on different aspects of the `DeLighT` block**, including uniform vs. block-wise scaling, depth scaling, and width scaling. Rows partially highlighted in color have the same configuration (repeated for illustrating results). Our experimental setup is similar to Section 5.4, except that we train our models for 50K iterations. Multiplication and addition operations (MACs) are computed for 20 time steps.

than DeLighT with $N_{min} = 8$ and $N_{max} = 8$ (R2), but delivers better performance with a similar number of parameters and operations. Scaling w_m improves performance (R2 vs. R3); however, the improvement is significantly lower than for the model with block-wise scaling (R3 vs. R5). This suggests that non-uniform distribution of parameters across blocks allows the network to learn better representations.

b) Different ratios between N_{max} and N_{min} yield different results. We observe significant performance improvements when the ratio is greater than or equal to two. For example, when we scale $\frac{N_{max}}{N_{min}}$ from 2 to 3 (R6 vs. R8), the perplexity improves by about 5 points with only a moderate increase in network parameters. On the other hand, when the $\frac{N_{max}}{N_{min}}$ is close to 1 (R6 vs. R7), performance does not change appreciably. This is likely because the allocation of parameters across blocks is close to uniform (Eq. 5.4). This is consistent with our previous observation.

c) Learning shallower and narrower representations near the input and deeper and wider representations near the output achieves better performance. For example, when we scaled N_{max} from 8 to 12 for $N_{min} = 4$ (R6, R8), DeLighT delivered better performance with a similar number of parameters compared to a model with $N_{min} = 6$ (R7, R9). This is likely because the ratio of N_{max} and N_{min} is higher when $N_{min} = 4$, which helps allocate parameters per block more effectively.

d) Deeper and wider representations near the input and shallower and narrower representations near the output hurts performance (R13 vs. R16).

e) Scaling width using w_m and d_m improves performance (R10-R15), however, their impact is different. For example, when we scale w_m and d_m by two, the rate of increase in number of parameters and operations is more rapid with d_m compared to w_m. DeLighT's ability to learn wider representations in different ways may be useful in selecting application specific models.

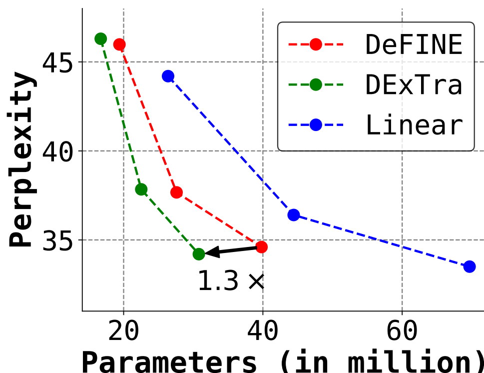


Figure 5.6: **Impact of different transformations.** `DeLighT` transformations are more parametric efficient than DeFINE and linear transformations. Lower perplexity value means better performance.

Impact of `DeLighT` transformation: We replace `DeLighT` transformation in the `DeLighT` block (Figure 5.1b) with (1) the DeFINE transformation and (2) a stack of linear layers. Figure 5.6 shows that `DeLighT` transformation delivers similar performance with significantly fewer parameters compared to the DeFINE unit and linear layers. In these experiments, the settings are the same as R13-R15 (Table 5.7), except, $N_{max} = 8$, because models with a stack of linear layers learn too many parameters.

Feature shuffling: Figure 5.7 shows that feature shuffling improves the performance of `DeLighT` by 1-2 perplexity points. Here, we use the same settings as in R13-R15 (Table 5.7).

Light-weight FFN: Figure 5.8 shows the impact of varying the reduction factor r in the light-weight FFN. We use the same settings as in R13 (Table 5.7). We did not observe any significant drop in performance until $r = 4$. Beyond $r = 4$, we see a drop in performance (perplexity increases by $\sim$2 points). In such cases, the inner dimensions of the light-weight FFN are very small and hurt performance. Notably, the light-weight FFN with $r = 2^2$ delivered the same performance as $r = 2^{-2}$, but with $1.28\times$ fewer network parameters. At

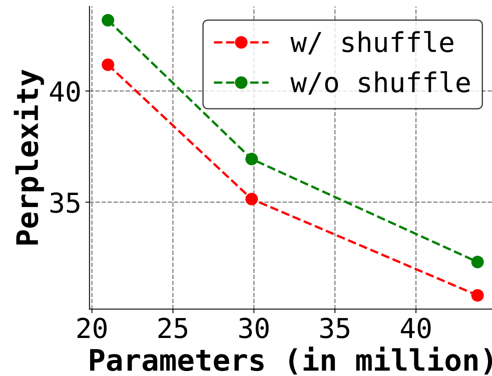


Figure 5.7: **Impact of feature shuffling.** Feature shuffling allows us to learn representations from global information and improves performance. Lower perplexity value means better performance.

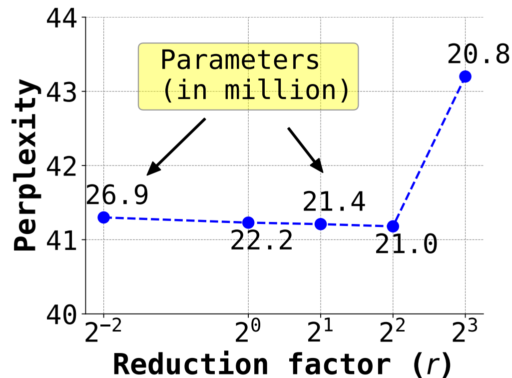


Figure 5.8: **Impact of reduction factor r in light-weight FFN.** The ability of `DeLighT` transformation to learn representations in high-dimensional spaces efficiently allows us to reduce the computational burden on the FFN. Lower perplexity value means better performance.

$r = 2^{-2}$, the light-weight FFN is the same as the FFN in [17]. This suggests that the ability of the `DeLighT` transformation to learn representations in high-dimensional spaces efficiently allows us to reduce the computational burden on the FFN.

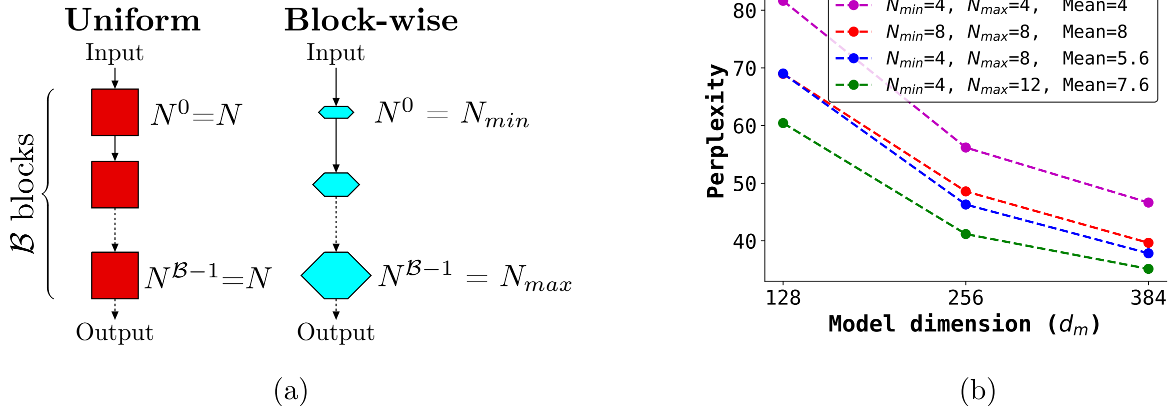


Figure 5.9: **Uniform vs. block-wise scaling.** (a) contrasts the uniform and block-wise scaling methods. (b) compares the results of DeLighT with uniform and block-wise scaling methods on the WikiText-103 dataset. DeLighT networks with block-wise scaling deliver better performance across different settings. Lower perplexity value means better performance.

We also tested removing the light-weight FFN, and while it reduced parameters by ∼0.5-1 M, performance dropped by about 2-3 perplexity points across different parametric settings.

Uniform vs. block-wise scaling: Figure 5.9 compares the performance of DeLighT with uniform and block-wise scaling. For a given model dimension d_m, DeLighT models with block-wise scaling deliver better performance.

Position of DeLighT transformation: We studied three configurations for the DeLighT transformation on the WikiText-103 validation set (Table 5.8): (1) DeLighT transformation followed by single-headed attention and light-weight FFN, (2) single-headed attention followed by DeLighT transformation, and (3) single-headed attention followed by DeLighT transformation and light-weight FFN. For similar number of parameters, we found that (2) and (3) drop the performance of (1) significantly across different parametric settings. This

Configuration	Parameters	Perplexity
DeLighT transformation + Single-head attention + Light-weight FFN	31 M	**34.20**
Single-head attention + DeLighT transformation	30 M	39.02
Single-head attention + DeLighT transformation + Light-weight FFN	31 M	39.43
DeLighT transformation + Single-head attention + Light-weight FFN	99 M	**23.16**
Single-head attention + DeLighT transformation	96 M	28.33
Single-head attention + DeLighT transformation + Light-weight FFN	99 M	27.94

Table 5.8: **Effect of the position of DeLighT transformation**. Lower value of perplexity means better performance.

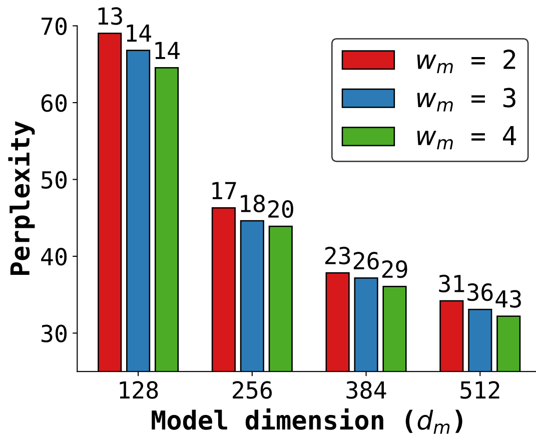
suggests that deeper and wider representations help learn better contextual representations, allowing us to replace multi-headed attention with single-headed attention.

Scaling up DeLighT: Figure 5.10 shows the results of DeLighT models obtained after varying configuration parameters of DeLighT transformations (N_{min}={4, 6}, N_{max}={8, 12}, w_m={2, 3, 4}, and d_m={256, 384, 512}). We can see that scaling one configuration parameter (e.g., d_m) while keeping other configuration parameters constant (e.g., N_{min}, N_{max}, and w_m) consistently improves performance.

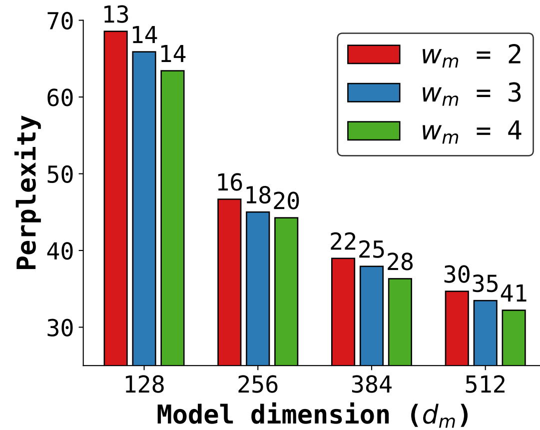
This work investigates relationships between N_{min}, N_{max}, w_m, and d_m, manually. We believe that a more principled approach, such as the compound scaling of Tan and Le [57] that establishes relationships between these parameters would produce more efficient and accurate models.

5.7 Summary

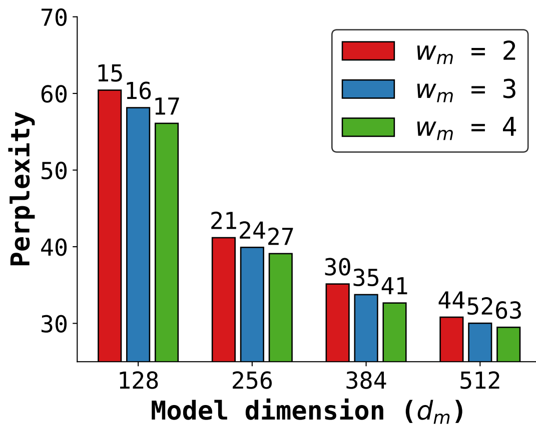
This chapter introduces a deep and light-weight transformer architecture, DeLighT, that efficiently allocates parameters both within the DeLighT block and across DeLighT blocks. Compared to state-of-the-art transformer models, DeLighT models are (1) deep and light-weight and (2) deliver similar or better performance.



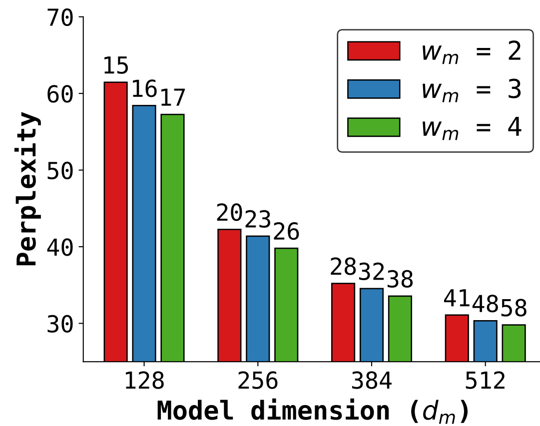
(a) N_{min}=4, N_{max}=8



(b) N_{min}=6, N_{max}=8



(c) N_{min}=4, N_{max}=12



(d) N_{min}=6, N_{max}=12

Figure 5.10: **Scaling up** `DeLighT`. Scaling one configuration parameter (e.g., d_m) while keeping other configuration parameters constant (e.g., N_{min}, N_{max}, and w_m) consistently improves performance. The numbers on top of each bar represent network parameters (in millions). Lower values of perplexity means better performance.

Chapter 6

CONCLUSION

Deep neural networks (DNNs) achieve jaw-dropping performances across different domains and are revolutionizing the world around us, from improving braille reading and writing skills to self-driving vehicles that sense and recognize obstacles. However, DNNs are resource-intensive. As a consequence, they are difficult to deploy on edge devices with limited computational capabilities. To run DNNs efficiently on such devices, we develop neural architectures that are light-weight, fast, and energy-efficient. This dissertation is geared towards improving the efficiency of DNNs by introducing efficient transformation functions, which allows networks to learn representations with fewer parameters and operations. Besides designing efficient architectures, we also demonstrated their generalization

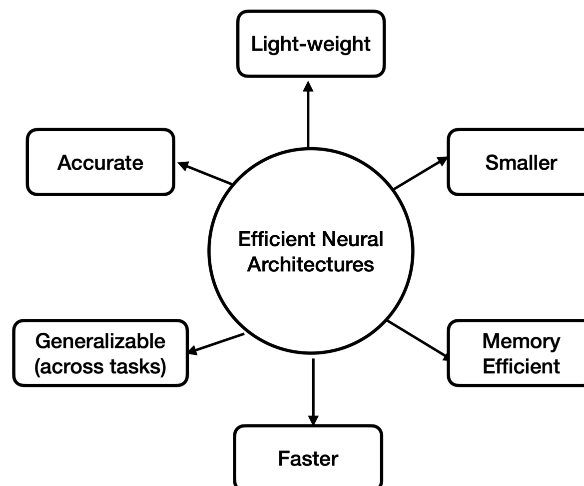


Figure 6.1: **Summary of the dissertation.** In this thesis, we develop efficient DNNs for learning representations from visual (Chapter 2 & 3) and textual (Chapter 4 & 5) data.

capabilities on several tasks and datasets. Figure 6.1 summarizes different aspects that we use to demonstrate the effectiveness of our networks.

Visual recognition tasks: To build efficient models, we introduce efficient spatial pyramidal (ESP) module in Chapter 2. The ESP module uses factorization to improve the computational efficiency of standard convolutions. The ESP module also allows learning representations from a larger effective receptive field, which helps it learn better representations from large structures (see Figure 2.7 and our work on segmenting and classifying breast biopsy whole slide images [52, 53, 149]). The `ESPNet` network, built using a stack of ESP modules, delivered similar or better performance than state-of-the-art methods while being much more power-efficient on a variety of tasks, including object detection and segmentation.

The basic building layer in the ESP module is depth-wise dilated convolution, which applies a single convolutional kernel per input channel. Though these convolutions have fewer parameters and operations compared to standard convolutions, they do not encode channel-wise relationships. To encode such relationships, we use point-wise (1×1) convolutions in the ESP module. However, point-wise convolutions account for a majority of operations (about 90%) in networks that are built using depth-wise convolutions, including `ESPNet`, MobileNets [46, 47], and ShuffleNets [48, 49]. In other words, these networks use less compute power to learn spatial representations. To better encode spatial relationships across different dimensions of the tensor, we introduce dimension-wise convolutions in Chapter 3, which applies a single convolutional kernel per dimension. The `DiCENet` network, built using dimension-wise convolutions, generalized better to different tasks and datasets, and outperformed existing manual as well as automatic efficient neural architectures by a significant margin across different computational budgets. This suggests that the efficiency and performance of neural architectures can be improved by learning better spatial representations.

Sequence modeling tasks: LSTMs [36] and Transformers [17] are two widely used sequence models. The basic building layer in both these models is a linear transformation

function. Because of the fully-connected nature of this function, they are computationally expensive and demand heavy regularization, especially on low-resource corpora [84].

Chapter 4 introduces an efficient recurrent unit, called the pyramidal recurrent unit (PRU). PRU replaces linear transformation functions in LSTMs with pyramidal and group linear transformations to model the input and context vectors, respectively. With these transformations, our networks can learn representations with fewer parameters and operations. Besides being efficient, PRUs require less regularization (or dropout) and deliver better performance as compared to LSTMs. Our analysis further revealed that PRUs have better gradient coverage as compared to LSTMs, suggesting that PRUs uses more features for decision making and suffers less from vanishing gradient problem.

Chapter 5 focuses on improving the efficiency of Transformers. We extend the group linear transformation in PRUs and introduce the `DeLighT` transformation that allows us to learn deeper and wider representations efficiently. Because of the ability of the `DeLighT` transformation to decouple the module's width and depth from the input and output dimensions, we can design variably-sized Transformer modules with `DeLighT`, i.e., each module has a different number of parameters and operators. Not only does `DeLighT` improve the efficiency of Transformers, but it also improves the performance with less regularization across different sequence modeling tasks and datasets, demonstrating good generalization capabilities.

In summary, this thesis takes important steps towards learning generalizable representations efficiently.

6.1 Future work

In this section, we present potential research directions for future work.

Neural architecture search: The efficient neural architectures introduced in this dissertation are *manually* designed, i.e., network hyper-parameters (e.g., kernel sizes, depth of the network, and latent dimensions) are manually chosen. Neural architecture search (e.g.,

[54, 55, 57]) allows us to design hardware-specific efficient neural architectures automatically. Many of the transformation functions that are introduced in this dissertation (e.g., the DiCE unit and the DeLighT layer) are novel and cannot be discovered using existing neural search-based methods. An important area of future research is to incorporate these efficient transformation functions in neural search dictionary and study the trade-off between performance and hardware efficiency.

Pretrained language models: Pretraining focuses on training a model on internet-scale data so that it can be used to provide contextualized embeddings (e.g., sentence-, paragraph-, or document-level) in down-stream tasks (e.g., question answering). With an exception to ELMO [150] that uses LSTMs, all pretrained models (e.g., BERT [2] and T5 [7]) uses the Transformer [17]. In Chapter 5, we introduce an efficient Transformer, called `DeLighT`. Therefore, pretraining a `DeLighT` model for edge devices could be an important research direction.

Domain-level generalization: This thesis focuses on computer vision and natural language processing tasks. However, the algorithms and architectures introduced in this dissertation are generic and can be used in different domains. An area of subsequent studies is to evaluate the generic nature of these architectures. For instance, PRUs can be used instead of LSTMs in modeling speech signals [151]. Similarly, `DeLighT` can be used instead of Transformers to learn visual representations for computer vision tasks, including computer-aided systems for diagnosing cancer using whole slide images [152, 153].

Compression and quantization: Efficient DNNs introduced in this dissertation uses full-precision floating-point operations. There is an opportunity to further improve the efficiency of our architectures using compression- and quantization-based approaches. Because computational requirements vary from device to device, we believe that device-aware compression and quantization methods for efficient DNNs would be an interesting area for future

research.

Learning multimodal representations efficiently: The efficient architectures introduced in this dissertation for learning visual and textual representations could be used for multimodal tasks (e.g., visual question answering and commonsense reasoning). Building efficient multimodal architectures could be an interesting future research direction.

We hope that the steps we take in this thesis towards designing efficient DNNs, along with all the future work it enables, will help spark future research in this and related areas.

BIBLIOGRAPHY

[1] 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*, pages 1026–1034, 2015.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*, 2019.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[4] Yuanqing Lin, Fengjun Lv, Shenghuo Zhu, Ming Yang, Timothee Cour, Kai Yu, Liangliang Cao, and Thomas Huang. Large-scale image classification: Fast feature extraction and svm training. In *CVPR 2011*, pages 1689–1696. IEEE, 2011.

[5] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115 (3):211–252, 2015.

[6] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[7] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning

with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21 (140):1–67, 2020.

[8] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3213–3223, 2016.

[9] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[10] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE transactions on pattern analysis and machine intelligence*, 39(12):2481–2495, 2017.

[11] Michael Treml, José Arjona-Medina, Thomas Unterthiner, Rupesh Durgesh, Felix Friedmann, Peter Schuberth, Andreas Mayr, Martin Heusel, Markus Hofmarcher, Michael Widrich, et al. Speeding up semantic segmentation for autonomous driving. In *MLITS, NIPS Workshop*, 2016.

[12] Adam Paszke, Abhishek Chaurasia, Sangpil Kim, and Eugenio Culurciello. Enet: A deep neural network architecture for real-time semantic segmentation. *arXiv preprint arXiv:1606.02147*, 2016.

[13] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE transactions on pattern analysis and machine intelligence*, 40(4):834–848, 2017.

[14] Hengshuang Zhao, Jianping Shi, Xiaojuan Qi, Xiaogang Wang, and Jiaya Jia. Pyramid

scene parsing network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2881–2890, 2017.

[15] Eduardo Romera, José M Alvarez, Luis M Bergasa, and Roberto Arroyo. Erfnet: Efficient residual factorized convnet for real-time semantic segmentation. *IEEE Transactions on Intelligent Transportation Systems*, 2018.

[16] Sachin Mehta, Mohammad Rastegari, Anat Caspi, Linda Shapiro, and Hannaneh Hajishirzi. Espnet: Efficient spatial pyramid of dilated convolutions for semantic segmentation. In *Proceedings of the european conference on computer vision (ECCV)*, pages 552–568, 2018.

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[18] Sachin Mehta, Marjan Ghazvininejad, Srinivasan Iyer, Luke Zettlemoyer, and Hannaneh Hajishirzi. DeLighT: Deep and light-weight transformer. In *International Conference on Learning Representations*, 2021.

[19] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[20] Blesson Varghese, Nan Wang, Sakil Barbhuiya, Peter Kilpatrick, and Dimitrios S Nikolopoulos. Challenges and opportunities in edge computing. In *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 20–26. IEEE, 2016.

[21] Clément Farabet, Cyril Poulet, Jefferson Y Han, and Yann LeCun. Cnp: An fpga-based processor for convolutional networks. In *2009 International Conference on Field Programmable Logic and Applications*, pages 32–37. IEEE, 2009.

[22] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In

Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays, pages 161–170, 2015.

[23] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 26–35, 2016.

[24] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 52(1):127–138, 2016.

[25] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *International Conference on Learning Representations*, 2018.

[26] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[27] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.

[28] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4820–4828, 2016.

[29] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. *Advances in neural information processing systems*, 29: 4107–4115, 2016.

[30] Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. Yodann: An architecture for ultralow power binary-weight cnn acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[31] Myle Ott, Sergey Edunov, David Grangier, and Michael Auli. Scaling neural machine translation. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, 2018.

[32] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[33] Chong Li and CJ Richard Shi. Constrained optimization based low-rank approximation of deep neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 732–747, 2018.

[34] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[35] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 1995.

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

[37] Sachin Mehta, Mohammad Rastegari, Linda Shapiro, and Hannaneh Hajishirzi. Espnetv2: A light-weight, power efficient, and general purpose convolutional neural network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9190–9200, 2019.

[38] Sachin Mehta, Hannaneh Hajishirzi, and Mohammad Rastegari. Dicenet: Dimension-wise convolutions for efficient networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[39] Sachin Mehta, Rik Koncel-Kedziorski, Mohammad Rastegari, and Hannaneh Hajishirzi. Pyramidal recurrent unit for language modeling. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Brussels, Belgium, 2018. Association for Computational Linguistics.

[40] Sachin Mehta, Rik Koncel-Kedziorski, Mohammad Rastegari, and Hannaneh Hajishirzi. DeFINE: Deep Factorized Input Token Embeddings for Neural Sequence Modeling. In *International Conference on Learning Representations*, 2020.

[41] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[42] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[43] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017.

[44] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[45] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-

scale image recognition. In *3rd International Conference on Learning Representations, ICLR*, 2015.

[46] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[47] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[48] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6848–6856, 2018.

[49] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European conference on computer vision (ECCV)*, pages 116–131, 2018.

[50] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[51] Gao Huang, Shichen Liu, Laurens Van der Maaten, and Kilian Q Weinberger. Condensenet: An efficient densenet using learned group convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2752–2761, 2018.

[52] Sachin Mehta, Ezgi Mercan, Jamen Bartlett, Donald Weaver, Joann Elmore, and Linda Shapiro. Learning to segment breast biopsy whole slide images. In *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 663–672. IEEE, 2018.

[53] Sachin Mehta, Ezgi Mercan, Jamen Bartlett, Donald Weaver, Joann G Elmore, and Linda Shapiro. Y-net: joint segmentation and classification for diagnosis of breast biopsy images. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 893–901. Springer, 2018.

[54] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations*, 2017.

[55] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[56] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10734–10742, 2019.

[57] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, 2019.

[58] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1314–1324, 2019.

[59] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured

sparsity in deep neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, 2016.

[60] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. In *NIPS Deep Learning and Representation Learning Workshop*, 2015.

[61] Saurabh Gupta, Judy Hoffman, and Jitendra Malik. Cross modal distillation for supervision transfer. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2827–2836, 2016.

[62] Junho Yim, Donggyu Joo, Jihoon Bae, and Junmo Kim. A gift from knowledge distillation: Fast optimization, network minimization and transfer learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4133–4141, 2017.

[63] Seyed Iman Mirzadeh, Mehrdad Farajtabar, Ang Li, Nir Levine, Akihiro Matsukawa, and Hassan Ghasemzadeh. Improved knowledge distillation via teacher assistant. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 5191–5198, 2020.

[64] Fisher Yu, Vladlen Koltun, and Thomas Funkhouser. Dilated residual networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 472–480, 2017.

[65] Panqu Wang, Pengfei Chen, Ye Yuan, Ding Liu, Zehua Huang, Xiaodi Hou, and Garrison Cottrell. Understanding convolution for semantic segmentation. In *2018 IEEE winter conference on applications of computer vision (WACV)*, pages 1451–1460. IEEE, 2018.

[66] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[67] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR, 2013.

[68] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. In *International Conference on Learning Representations (ICLR)*, 2017.

[69] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[70] Feng Zhu, Hongsheng Li, Wanli Ouyang, Nenghai Yu, and Xiaogang Wang. Learning spatial regularization with image-level supervisions for multi-label image classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5513–5522, 2017.

[71] Mark Everingham, SM Ali Eslami, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes challenge: A retrospective. *International journal of computer vision*, 111(1):98–136, 2015.

[72] Bharath Hariharan, Pablo Arbeláez, Lubomir Bourdev, Subhransu Maji, and Jitendra Malik. Semantic contours from inverse detectors. In *2011 International Conference on Computer Vision*, pages 991–998. IEEE, 2011.

[73] Rudra PK Poudel, Ujwal Bonde, Stephan Liwicki, and Christopher Zach. Contextnet: Exploring context and detail for semantic segmentation in real-time. In *Proceedings of the British Machine Vision Conference (BMVC)*, 2018.

[74] Hengshuang Zhao, Xiaojuan Qi, Xiaoyong Shen, Jianping Shi, and Jiaya Jia. Icnet for real-time semantic segmentation on high-resolution images. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 405–420, 2018.

[75] M. Siam, M. Gamal, M. Abdel-Razek, S. Yogamani, and M. Jagersand. RTSeg: real-time semantic segmentation comparative study. In *2018 25th IEEE International Conference on Image Processing (ICIP)*, 2018.

[76] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. *arXiv preprint arXiv:1706.05587*, 2017.

[77] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[78] Mingxing Tan and Quoc V. Le. Mixconv: Mixed depthwise convolutional kernels. In *30th British Machine Vision Conference*, 2019.

[79] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018.

[80] Ross Wightman. PyTorch Image Models, 2021. URL https://github.com/rwightman/pytorch-image-models.

[81] Zhong-Qiu Zhao, Peng Zheng, Shou-tao Xu, and Xindong Wu. Object detection with deep learning: A review. *IEEE transactions on neural networks and learning systems*, 30(11):3212–3232, 2019.

[82] Yoon Kim, Yacine Jernite, David Sontag, and Alexander Rush. Character-aware neural language models. In *Proceedings of the AAAI conference on artificial intelligence*, 2016.

[83] Gábor Melis, Chris Dyer, and Phil Blunsom. On the state of the art of evaluation in neural language models. In *International Conference for Learning Representations (ICLR)*, 2018.

[84] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing lstm language models. In *International Conference for Learning Representations (ICLR)*, 2018.

[85] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS 2014 Workshop on Deep Learning, December 2014*, 2014.

[86] Felix A Gers and Jürgen Schmidhuber. Recurrent nets that time and count. In *IEEE-INNS-ENNS International Joint Conference on Neural Networks (IJCNN)*, 2000.

[87] Tao Lei, Yu Zhang, and Yoav Artzi. Training rnns as fast as cnns. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

[88] Omer Levy, Kenton Lee, Nicholas FitzGerald, and Luke Zettlemoyer. Long short-term memory as a dynamically computed element-wise weighted sum. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Association for Computational Linguistics, 2018.

[89] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. In *International Conference on Machine Learning (ICML)*, 2017.

[90] Dongyoon Han, Jiwhan Kim, and Junmo Kim. Deep pyramidal residual networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5927–5935, 2017.

[91] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research (JMLR)*, 2014.

[92] Diederik P Kingma, Tim Salimans, and Max Welling. Variational dropout and the local reparameterization trick. In *Advances in Neural Information Processing Systems (NIPS)*, 2015.

[93] Chen Xu, Jianqiang Yao, Zhouchen Lin, Wenwu Ou, Yuanbin Cao, Zhirong Wang, and Hongbin Zha. Alternating multi-bit quantization for recurrent neural networks. In *International Conference for Learning Representations (ICLR)*, 2018.

[94] Minjoon Seo, Sewon Min, Ali Farhadi, and Hannaneh Hajishirzi. Neural speed reading via skim-rnn. In *International Conference on Learning Representations (ICLR)*, 2018.

[95] Adams Wei Yu, Hongrae Lee, and Quoc Le. Learning to skim text. In *Association for Computational Linguistics (ACL)*, 2017.

[96] Víctor Campos, Brendan Jou, Xavier Giró-i Nieto, Jordi Torres, and Shih-Fu Chang. Skip rnn: Learning to skip state updates in recurrent neural networks. In *International Conference for Learning Representations (ICLR)*, 2018.

[97] Minjoon Seo, Sewon Min, Ali Farhadi, and Hannaneh Hajishirzi. Query-reduction networks for question answering. In *International Conference on Learning Representations (ICLR)*, 2017.

[98] Zhilin Yang, Zihang Dai, Ruslan Salakhutdinov, and William W Cohen. Breaking the softmax bottleneck: a high-rank rnn language model. In *International Conference for Learning Representations (ICLR)*, 2018.

[99] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference for Learning Representations (ICLR)*, 2017.

[100] Ben Krause, Emmanuel Kahembwe, Iain Murray, and Steve Renals. Dynamic evaluation of neural sequence models. In *International Conference on Machine Learning (ICML)*, 2018.

[101] Peter J Burt and Edward H Adelson. The laplacian pyramid as a compact image code. In *Readings in Computer Vision*. Elsevier, 1987.

[102] David G Lowe. Object recognition from local scale-invariant features. In *IEEE international conference on Computer vision (ICCV)*, 1999.

[103] Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *Advances in neural information processing systems (NIPS)*, 2016.

[104] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 1993.

[105] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[106] Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutnık, and Jürgen Schmidhuber. Recurrent highway networks. In *International Conference on Machine Learning*, pages 4189–4198. PMLR, 2017.

[107] Hakan Inan, Khashayar Khosravi, and Richard Socher. Tying word vectors and word classifiers: A loss framework for language modeling. In *International Conference for Learning Representations (ICLR)*, 2017.

[108] James Bradbury, Stephen Merity, Caiming Xiong, and Richard Socher. Quasi-recurrent neural networks. In *International Conference for Learning Representations (ICLR)*, 2017.

[109] Muriel Gevrey, Ioannis Dimopoulos, and Sovan Lek. Review and comparison of methods to study the contribution of variables in artificial neural network models. *Ecological modelling*, 2003.

[110] Jiwei Li, Xinlei Chen, Eduard Hovy, and Dan Jurafsky. Visualizing and understanding neural models in nlp. In *North American Chapter of the Association for Computational Linguistics (NAACL)*, 2016.

[111] Leila Arras, Grégoire Montavon, Klaus-Robert Müller, and Wojciech Samek. Explaining recurrent neural network predictions in sentiment analysis. In *8th Workshop on Computational Approaches to Subjectivity, Sentiment and Social Media Analysis,*, 2017.

[112] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33 pre-proceedings (NeurIPS 2020)*, 2020.

[113] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[114] Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *International conference on machine learning*, pages 1058–1066, 2013.

[115] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *Association for Computational Linguistics*, 2019.

[116] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[117] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *International Conference on Learning Representations*, 2020.

[118] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv:2004.05150*, 2020.

[119] Alessandro Raganato and Jörg Tiedemann. An analysis of encoder representations in transformer-based machine translation. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, November 2018.

[120] Gino Brunner, Yang Liu, Damian Pascual, Oliver Richter, Massimiliano Ciaramita, and Roger Wattenhofer. On identifiability in transformers. In *International Conference on Learning Representations*, 2020.

[121] Elena Voita, Rico Sennrich, and Ivan Titov. The bottom-up evolution of representations in the transformer: A study with machine translation and language modeling objectives. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019.

[122] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? In *Advances in Neural Information Processing Systems*, pages 14014–14024, 2019.

[123] Alessandro Raganato, Yves Scherrer, and Jörg Tiedemann. Fixed encoder self-attention patterns in transformer-based machine translation. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, November 2020.

[124] Yi Tay, Dara Bahri, Donald Metzler, Da-Cheng Juan, Zhe Zhao, and Che Zheng. Synthesizer: Rethinking self-attention in transformer models. *arXiv preprint arXiv:2005.00743*, 2020.

[125] Felix Wu, Angela Fan, Alexei Baevski, Yann Dauphin, and Michael Auli. Pay less attention with lightweight and dynamic convolutions. In *International Conference on Learning Representations*, 2019.

[126] Zhanghao Wu, Zhijian Liu, Ji Lin, Yujun Lin, and Song Han. Lite transformer with long-short range attention. In *International Conference on Learning Representations*, 2020.

[127] David So, Quoc Le, and Chen Liang. The evolved transformer. In *Proceedings of the 36th International Conference on Machine Learning*, pages 5877–5886, 2019.

[128] Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 933–941. JMLR. org, 2017.

[129] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In *International Conference on Learning Representations*, 2020.

[130] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[131] Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F. Wong, and Lidia S. Chao. Learning deep transformer models for machine translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

[132] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, August 2016.

[133] Alexei Baevski and Michael Auli. Adaptive input representations for neural language modeling. In *International Conference on Learning Representations*, 2019.

[134] Édouard Grave, Armand Joulin, Moustapha Cissé, David Grangier, and Hervé Jégou. Efficient softmax approximation for GPUs. In *International Conference on Machine Learning*, 2017.

[135] Patrick Chen, Si Si, Yang Li, Ciprian Chelba, and Cho-Jui Hsieh. Groupreduce: Block-wise low-rank approximation for neural language model shrinking. In *Advances in Neural Information Processing Systems*, 2018.

[136] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. Mobilebert: a compact task-agnostic bert for resource-limited devices. In *Association for Computational Linguistics (ACL)*, 2020.

[137] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

[138] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. In *5th Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS*, 2019.

[139] Matus Telgarsky. Benefits of depth in neural networks. In *Conference on learning theory*, pages 1517–1539. PMLR, 2016.

[140] Sergey Edunov, Myle Ott, Michael Auli, David Grangier, and Marc'Aurelio Ranzato. Classical structured prediction losses for sequence to sequence learning. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, 2018.

[141] Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, Łukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit. Tensor2tensor for neural machine translation. *CoRR*, abs/1803.07416, 2018.

[142] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

[143] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. Fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

[144] Marjan Ghazvininejad, Omer Levy, Yinhan Liu, and Luke Zettlemoyer. Mask-predict: Parallel decoding of conditional masked language models. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 6114–6123, 2019.

[145] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

[146] Yuntian Deng, Yoon Kim, Justin Chiu, Demi Guo, and Alexander Rush. Latent alignment and variational attention. In *Advances in Neural Information Processing Systems*, pages 9712–9724, 2018.

[147] Edouard Grave, Armand Joulin, and Nicolas Usunier. Improving neural language models with a continuous cache. In *International Conference on Learning Representations*, 2017.

[148] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. An analysis of neural language modeling at multiple scales. *arXiv preprint arXiv:1803.08240*, 2018.

[149] Ezgi Mercan, Sachin Mehta, Jamen Bartlett, Linda G Shapiro, Donald L Weaver, and Joann G Elmore. Assessment of machine learning of breast pathology structures for

automated differentiation of breast cancer and high-risk proliferative lesions. *JAMA network open*, 2(8):e198777–e198777, 2019.

[150] Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, 2018.

[151] Shinji Watanabe, Takaaki Hori, Shigeki Karita, Tomoki Hayashi, Jiro Nishitoba, Yuya Unno, Nelson Enrique Yalta Soplin, Jahn Heymann, Matthew Wiesner, Nanxin Chen, Adithya Renduchintala, and Tsubasa Ochiai. ESPnet: End-to-end speech processing toolkit. In *Proceedings of Interspeech*, pages 2207–2211, 2018.

[152] Sachin Mehta, Ximing Lu, Donald Weaver, Joann G Elmore, Hannaneh Hajishirzi, and Linda Shapiro. Hatnet: An end-to-end holistic attention network for diagnosis of breast biopsy images. *arXiv preprint arXiv:2007.13007*, 2020.

[153] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021.

Appendix A

SUPPLEMENTARY MATERIAL FOR THE DICENET NETWORK

A.1 Architecture of the DiCENet *Network*

Table A.1 shows the overall architecture of DiCENet at different network complexities. The first layer is a standard 3×3 convolution with a stride of two while the second layer is a max pooling layer. All convolutional layers are followed by a batch normalization layer [66] and a PReLU non-linear activation layer [1], except for the last layer that feeds into a softmax for classification. We scale the number of output channels by a width scaling factor s to construct networks at different FLOPs. We initialize weights of our network using the same method as in [1].

Layer	Output size	Kernel size	Stride	Repeat	Output channels (network width scaling parameter s)			
					$s = 0.1$	$s = 0.2$	$s \in [0.5, 2.0]$	$s = 2.4$
Image	224×224				3	3	3	3
Conv1	112×112	3×3	2	1	8	16	24	24
Max Pool	56×56	3×3	2		8	16	24	24
Block 1	28×28		2	1	16	32	116 $\times s$	278
	28×28		1	3	16	32	116 $\times s$	278
Block 2	14×14		2	1	32	64	232 $\times s$	556
	14×14		1	7	32	64	232 $\times s$	556
Block 3	7×7		2	1	64	128	464 $\times s$	1112
	7×7		1	3	64	128	464 $\times s$	1112
Global Pool	1×1	7×7			512	1024	1024	1280
Grouped FC [39] (# groups=4)	1×1	1×1	1	1	512	1024	1024	1280
FC					1000	1000	1000	1000
FLOPs					6.5 M	12 M	24-240 M	298 M

Table A.1: Overall architecture of `DiCENet` (ShuffleNetv2 with `DiCE` unit) at different network complexities for the ImageNet classification. For other architecture designs in Figure 3.5 (MobileNetv1 and ResNet), we replace blocks 1, 2, and 3 with the corresponding blocks.

Appendix B

SUPPLEMENTARY MATERIAL FOR THE DeLighT

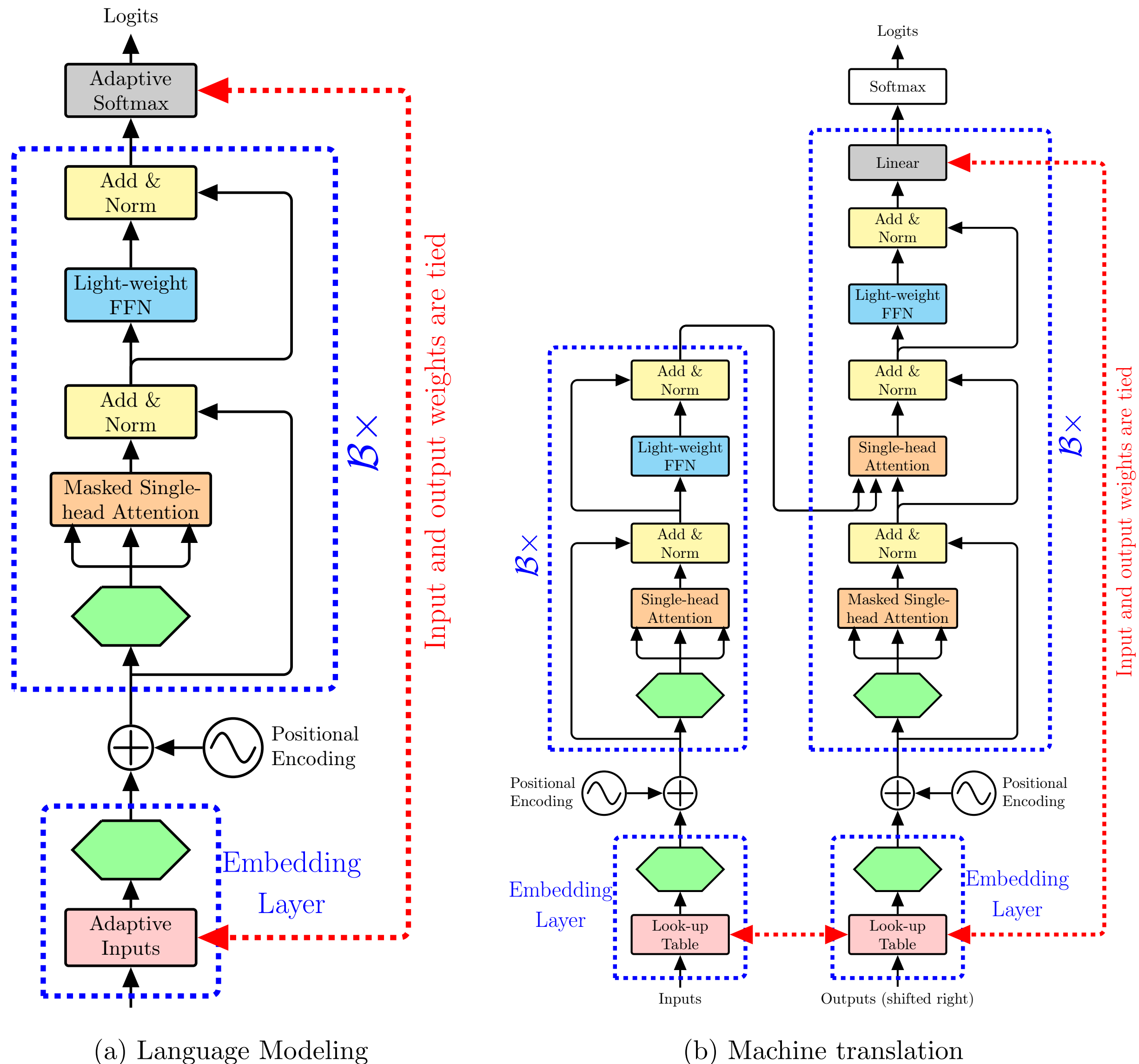
B.1 DeLighT *Architectures for Language Modeling and Machine Translation*

DeLighT architectures for language modeling and machine translation are shown in Figure B.1. For language modeling, we follow the architecture in Baevski and Auli [133] while for machine translation, we follow the architecture in Vaswani et al. [17].

Language modeling: Figure B.1a shows the architecture for language modeling. The architecture stacks $\mathcal{B}$ DeLighT blocks, the configuration of each block is determined using block-wise scaling. Each block has three sub-layers. The first layer is a DeLighT transformation that learns representations in high-dimensional space. The second layer is a single-head attention that encodes contextual relationships. The third layer is a position-wise light-weight feed-forward network. Similar to Vaswani et al. [17], we employ a residual connections [41]. Similar to previous works [115, 133], we use tied adaptive input [133] and adaptive softmax [134] to map tokens to vectors and vectors to tokens, respectively [107].

Machine translation: Figure B.1b shows the architecture for machine translation. The encoder stacks $\mathcal{B}$ DeLighT blocks, the configuration of each block is determined using block-wise scaling. Similar to language modeling, each encoder block has three sub-layers. The first layer is a DeLighT transformation that learns representations in high-dimensional space. The second layer is a single-head attention that encodes contextual relationships. The third layer is a position-wise light-weight feed-forward network. Similar to Vaswani et al. [17], we employ a residual connections [41]. We use learnable look-up table to map tokens to vectors. Similar to the encoder, the decoder also stacks $\mathcal{B}$ blocks. Decoder blocks are identical to encoder blocks, except that they have an additional source-target single-head attention unit before the light-weight FFN. Keys and values in source-target single-head attention unit are

projections over the encoder output. We use standard learnable look-up table to map tokens to vectors and linear classification layer to map vectors to tokens.



(a) Language Modeling

(b) Machine translation

Figure B.1: Sequence modeling with `DeLighT`. Here, green color hexagon represents the `DeLighT` transformation.

B.2 Group linear transformation with Input-mixer connection

Group linear transformation (GLT) $\mathcal{F}$ splits a d_m-dimensional input $\mathbf{X}$ into g non-overlapping groups such that $\mathbf{X} = \text{Concat}(\mathbf{X}_1, \cdots, \mathbf{X}_g)$, where $\mathbf{X}_i$ is a $\frac{d_m}{g}$-dimensional vector. $\mathbf{X}_i$'s are then simultaneously transformed using g linear transforms $\mathbf{W}_i \in \mathbf{R}^{\frac{d_m}{g} \times \frac{d_o}{g}}$ to produce g outputs $\mathbf{Y}_i = \mathbf{X}_i\mathbf{W}_i$. $\mathbf{Y}_i$'s are then concatenated to produce the final d_o-dimensional output $\mathbf{Y} = \text{Concat}(\mathbf{Y}_1, \cdots, \mathbf{Y}_g)$.

Figure B.2a shows an example of GLT in the expansion phase of `DeLighT` transformation. For illustrative purposes, we have used the same dimensions in this example. Recall that as we go deeper in the expansion phase, the number of groups increases. In this example, the first layer has one group, the second layer has two groups, and the third layer has four groups. GLTs learns group-specific representations and are local. To allow GLT to learn global representations, we use feature shuffle. An example of GLT with feature shuffle is shown in Figure B.2b. Furthermore, training deep neural networks by merely stacking linear or group linear (with or without feature shuffle) is challenging because of vanishing gradient problem. Residual connections introduced by He et al. [41] mitigates this problem and helps train deep neural networks. However, such connections cannot be employed when input and output dimensions are not the same (e.g., during the expansion and reduction phases in `DeLighT` transformation). To stabilize the training and learn deeper representations, we use input-mixer connection of Mehta et al. [40]. Figure B.2c shows an example of GLT with feature shuffle and input mixer connection.

B.3 Multiplication-Addition Operations in `DeLighT`

The `DeLighT` block is built using linear transformations, GLTs, and scaled dot-product attention. Total number of multiplication-addition operations (MACs) in a network is an accumulation of these individual operations.

Let n denotes the number of source tokens, m denotes the number of target tokens, d_m denotes the input dimension, d_o denotes the output dimension, and g denotes the number of

groups in GLT. The procedure for counting MACs for each of these operations is described below.

Group linear transformation (GLT): GLT $\mathcal{F}$ has g learnable matrices $\mathbf{W}_i \in \mathbf{R}^{\frac{d_m}{g} \times \frac{d_o}{g}}$. Therefore, GLT learns $\frac{d_m d_o}{g}$ parameters and performs $\frac{d_m d_o}{g}$ MACs to transform d_m-dimensional input to d_o-dimensional output. Following a standard practice, e.g., ResNet of He et al. [41], we count addition and multiplication as one operation instead of two because these operations can be fused in recent hardwares.

Importantly, when $g = 1$, the GLT is the same as linear transformation.

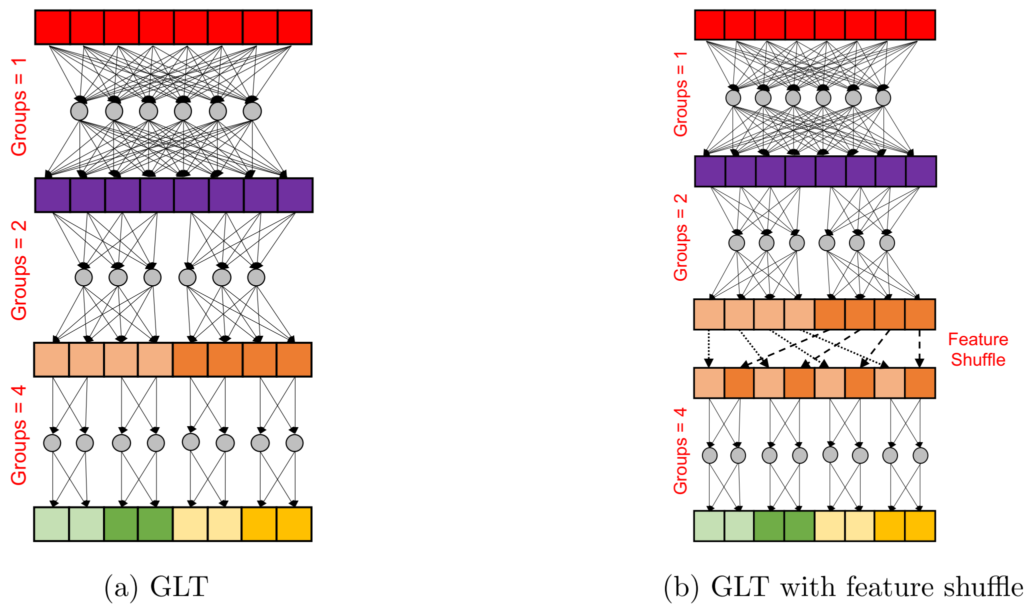
Self-attention in `DeLighT`: The scaled dot-product self-attention in `DeLighT` is defined as:

$$\text{Attention}(\mathbf{K}, \mathbf{Q}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_o}}\right)\mathbf{V} \tag{B.1}$$

where $\mathbf{Q} \in \mathbb{R}^{n \times d_o}$, $\mathbf{K} \in \mathbb{R}^{n \times d_o}$, $\mathbf{V} \in \mathbb{R}^{n \times d_o}$ denotes query, key, and value, respectively.

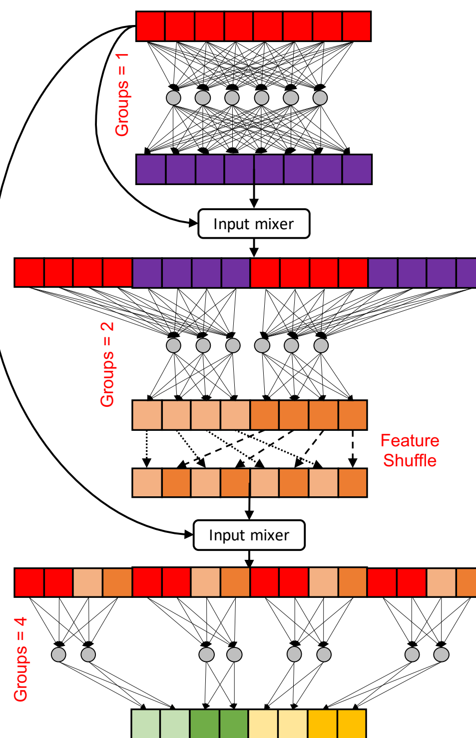
The attention operation involves two dot-products. The first dot product between $\mathbf{Q}$ and $\mathbf{K}$ while the second dot product is between the output of first dot product and $\mathbf{V}$. Both dot products require $d_o n^2$ MACs. Therefore, total number of MACs in computing scaled dot-product self-attention are $2 d_o n^2$.

In case of a source-target attention (as in machine translation), $\mathbf{K}$'s and $\mathbf{V}$'s are from the source (encoder) and $\mathbf{Q}$'s are incrementally decoded (one token at a time). Therefore, the number of MACs required to decode m target tokens given n source tokens are $\sum\limits_{k=1}^{m} 2knd_o$.



(a) GLT

(b) GLT with feature shuffle



(c) GLT with feature shuffle and input mixture connection

Figure B.2: This figure visualizes different variants of group linear transformations that are used in the `DeLighT` transformation.