PeleeNet: An efficient CNN model architecture

E4040.2019Fall.PURJ.report
Ruturaj Nene (rn2494), Jeswanth Yadagani (jy3012), Ujwal Dinesha (ud2130)
Columbia University

Abstract

A growing need for running Convolutional Neural Network (CNN) models encourages studies on efficient model design. All sophisticated models are heavily dependent on depthwise separable convolution which lacks efficient implementation in most deep learning frameworks. In the project we studied, an efficient architecture named PeleeNet, which is built with conventional convolution instead and compared it with well-known counterparts.

1. Introduction

With the increase in applications of CNN in object recognition and tracking on resource-constrained devices many innovative architectures, such as MobileNets, ShuffleNet, NASNet-A have been proposed in recent years. However, all these architectures are heavily dependent on depthwise separable convolution which lacks efficient implementation.

As CNN tries to go deeper with stacked layers the information and gradient start vanishing this results in low representational ability in network. To solve this problem the authors have come up with a variant of popular DenseNet architecture called The PeleeNet for mobile devices. PeleeNet follows the connectivity pattern and some of the key design principles of DenseNet[3]. It is also designed to meet strict constraints on memory and computational budget. The key challenge is to build a model that meets the tight device constraints but is complex enough to learn representations efficiently.

This report is organized in the following way. A brief introduction to this paper has been given in Section-1. Section-2 summarizes the original paper of PeleeNet with some of its salient features. Section-3 describes the network architecture and discusses attaining PeleeNet from DenseNet-41 architecture. Section-4 provides implementation details including data used and technical challenges and limitations faced. Section-5 provides obtained results and does a certain comparative study with the original paper. Section-6 concludes the paper with the understanding that we obtained.

2. Summary of the Original Paper

2.1 Methodology

The authors described a network called PeleeNet and they then used a modified SSD to integrate it with PeleeNet to create a real-time object detection system called Pelee. The paper is divided into two parts the first explains the PeleeNet and its results and the second part explains the system Pelee. For training and baseline model comparison of PeleeNet, the dataset used is Stanford Dog dataset as a subset of ILSVRC 2012. Then the authors go on to train the Pelee system on different popular datasets such as ILSVRC 2012, VOC 2007 and COCO. The results are obtained by comparing the computational cost, accuracy, running time of the Pelee system with other frameworks such as YOLOv2, MobileNet+SSD.

2.2 Key Results

The results obtained in the original paper are based on either the PeleeNet(CNN model) or Pelee the real-time object detection technique. The results of both are summarized as follows[2]:

- **PeleeNet:**

The original paper used DenseNet-41 as a baseline model for comparison with PeleeNet. In the paper, authors have also obtained an increase in accuracy with the addition of each feature of PeleeNet into DenseNet-41 architecture. This result is summarized in the table shown in Figure 1. The dataset used for this experiment was Stanford Dog dataset obtained as a subset of ILSVRC 2012.

	From DenseNet-41 to PeleeNet						
Transition layer without compression		✓	✓	✓	✓	✓	✓
Post-activation			✓			✓	✓
Dynamic bottleneck channels				✓	✓	✓	✓
Stem Block					✓	✓	✓
Two-way dense layer						✓	✓
Go deeper (add 3 extra dense layers)							✓
Top 1 accuracy	**75.02**	76.1	75.2	75.8	76.8	78.8	**79.25**

Figure 1: From DenseNet-41 to PeleeNet

The PeleeNet was trained with a cosine learning rate annealing schedule. Cosine Learning Rate Annealing means that the learning rate decays with a cosine shape (the learning rate of epoch t (t <= 120) set to 0.5 ∗ (learning rate) ∗ (cos(π ∗ t/120) + 1). As can be seen from Figure 2, PeleeNet achieved higher accuracy than that of MobileNet and ShuffleNet at no more than 66% model size and the lower computational cost. The model size of PeleeNet was only 1/49 of VGG16.

Model	Computational Cost (FLOPs)	Model Size (Parameters)	Accuracy (%) Top-1	Top-5
VGG16	15,346 M	138 M	71.5	89.8
1.0 MobileNet	569 M	4.24 M	70.6	89.5
ShuffleNet 2x (g = 3)	524 M	5.2 M	70.9	-
NASNet-A	564 M	5.3 M	74.0	91.6
PeleeNet	**508 M**	**2.8 M**	**72.6**	**90.6**

Figure 2: PeleeNet compared to popular CNNs

- **Pelee System:**

The authors tried to optimize SSD to improve the speed with acceptable accuracy. In their proposed SSD, for each feature map used for detection, they build a residual block before conducting prediction. They also used small convolutional kernels to predict object categories and bounding box locations to reduce computational cost. There were 5 scales of feature maps used in their system for prediction: 19 x 19, 10 x 10, 5 x 5, 3 x 3, and 1 x 1. They did not use a 38 x 38 feature map layer to ensure a balance can be reached between speed and accuracy. With these design choices, they achieved the final system with 70.9% mAP on PASCAL VOC2007 and 22.4 mAP on the MS COCO dataset. The result on COCO outperformed YOLOv2 in consideration of higher precision, 13.6 times lower computational cost and 11.3 times smaller model size.

Results on VOC2007:

The authors summarized the effect of design choices on the accuracy and speed of the model. From Figure 3 It is evident that the residual prediction block effectively improved accuracy. The model with the residual prediction block achieved higher accuracy by 2.2% than the model without residual prediction block. The accuracy of the model using 1x1 kernels for prediction was almost the same as the one of the model using 3x3 kernels. However, 1x1 kernels reduce the computational cost by 21.5% and the model size by 33.9%.

38x38 Feature	ResBlock	Kernel Size for Prediction	FLOPs	Model Size (Parameters)	mAP (%)
✓	✗	3x3	1,670 M	5.69 M	69.3
✗	✗	3x3	1,340 M	5.63 M	68.6
✗	✓	3x3	1,470 M	7.27 M	70.8
✗	✓	1x1	1,210 M	5.43 M	**70.9**

Figure 3: Design choices and effect on accuracy

The comparison of the Pelee system with other frameworks showed the accuracy of Pelee being higher than that of TinyYOLOv2 by 13.8% and higher than that of SSD+MobileNet Huang et al. (2016b) by 2.9%. It was even higher than that of YOLOv2-288 at only 14.5% of the computational cost of YOLOv2-288. Pelee achieved 76.4% mAP.

The Pelee was tried on real devices to test the speed and evaluate it against other real-time object detection systems. The speed was calculated by the average time of 100 images processed by the benchmark tool. This time included the image preprocessing time, but it did not include the time of the post-processing part. Although the residual prediction block used in Pelee increased the computational cost, Pelee still ran faster than SSD+MobileNet on iPhone and on TX2 in FP32 mode. As can be seen from Figure 4, Pelee had a greater speed advantage compared to SSD+MobileNet and SSDLite+MobileNetV2 in FP16 mode.

Model	Input Dimension	FLOPs	Speed (FPS) iPhone 8	TX2 (FP16)	TX2 (FP32)
SSD+MobileNet	300x300	1,200 M	22.8	82	73
SSDLite+MobileNetV2	320x320	805 M	-	62	60
Pelee (ours)	304x304	1,290 M	**23.6**	**125**	**77**

Figure 4: Comparison of Pelee with other frameworks

Results on COCO dataset:

The Pelee was further validated on the COCO dataset. The batch size was set to 128. Pelee was not only more accurate than SSD+MobileNet but also more accurate than YOLOv2 in both mAP@[0.5:0.95] and mAP@0.75. Meanwhile, Pelee was 3.7 times faster in speed and 11.3 times smaller in model size than YOLOv2.

Model	Input Dimension	Speed on TX2 (FPS)	Model Size (Parameters)	Avg. Precision (%), IoU: 0.5:0.95	0.5	0.75
Original SSD	300x300	-	34.30 M	25.1	43.1	25.8
YOLOv2	416x416	32.2	67.43 M	21.6	44.0	19.2
YOLOv3	320x320	21.5	62.3 M	-	51.5	-
YOLOv3-Tiny	416x416	105	12.3 M	-	33.1	-
SSD+MobileNet	300x300	80	6.80 M	18.8	-	-
SSDlite + MobileNet v2	320x320	61	4.3 M	22	-	-
Pelee (ours)	304x304	120	5.98 M	**22.4**	**38.3**	**22.9**

Figure 5: Pelee on COCO dataset against other frameworks

Conclusion:

The conclusions that were drawn by the authors are summarized henceforth. Depthwise separable convolution is not the only way to build an efficient model. Instead of using depthwise separable convolution, the author proposed PeleeNet and Pelee built with conventional convolution and have achieved compelling results on ILSVRC 2012, VOC 2007 and COCO. By combining efficient architecture design with mobile GPU and hardware-specified optimized runtime libraries, they were able to perform real-time prediction for image classification and object detection tasks on mobile devices. For example, Pelee, their proposed object detection system, could run 23.6 FPS on iPhone 8 and 125 FPS on NVIDIA TX2 with high accuracy

3. Methodology

We start by giving details of the objective of the project and the challenges faced during implementation. Then the architecture used as the baseline(DenseNet) and the actual PeleeNet architecture is discussed.

3.1. Objectives and Technical Challenges

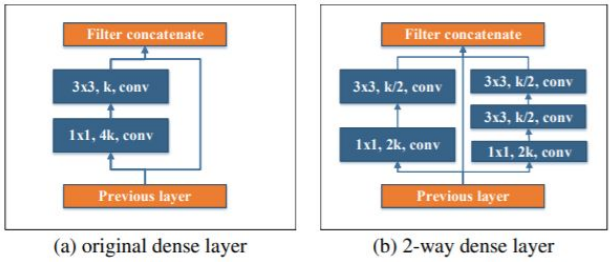
The objective of the project is to reproduce the results from the original Pelee paper and compare the results with the original paper. The paper introduces a cost-efficient CNN called PeleeNet which is inspired by the DenseNet architecture. We have tried to implement the same architecture. The technical challenges faced during this are as follows:

- **The dataset:** The authors have used the Imagenet ILSVRC 2012 and Stanford dog dataset for training purposes. The Imagenet dataset is large in size and also requires permission from the Imagenet organization which is very limited. This put restrictions on the data we can use to recreate the results.
- **Computation constraints:** The CNNs require a lot of computation power for training and validation purposes. The original paper utilizes two Nvidia TX2 GPUs to train its model by PyTorch with a batch size of 512 and is run for a total of 120 epochs with cosine learning rate annealing schedule. Instead, we have used Nvidia Tesla P100 with 104GB high memory, 16 vCPU on the google cloud platform to train our model. Thus we have limited our epochs to 20 and batch size to 64 which took more than 13 hours and 10 hours to finish training for ImageNet and CIFAR10 data respectively. And also learning rate has been set to a constant value of 1e-3.
- **Object detection system**: The authors used PeleeNet along with a modified version of SSD(Single shot multibox detector) for real-time object detection. This system was also trained and tested for standard datasets such as ILSVRC 2012, VOC 2007 and COCO. However, this system is difficult to implement and test on the aforementioned data. Hence, we did not implement the Pelee system in our project.

3.2. Problem Formulation and Design

The problem at hand is to design a CNN which is cost-efficient and complex enough to understand the representational features in the images and generalize well on unseen images. To achieve this goal we are trying to build the PeleeNet model. This model is inspired by the DenseNet but the key differences between the PeleeNet and DenseNet are as follows[2]:

Two-Way Dense Layer: the 2-way dense layer to get different scales of receptive fields. One way of the layer uses a 3x3 kernel size. The other way of the layer uses two stacked 3x3 convolution to learn visual patterns for large objects. The structure is shown in Figure 6.



(a) original dense layer (b) 2-way dense layer

Figure 6: The dense layer

Stem Block: The stem block can effectively improve the feature expression ability without adding computational cost too much - better than other more expensive methods, e.g., increasing channels of the first convolution layer or increasing growth rate. This stem block is shown in figure 7.

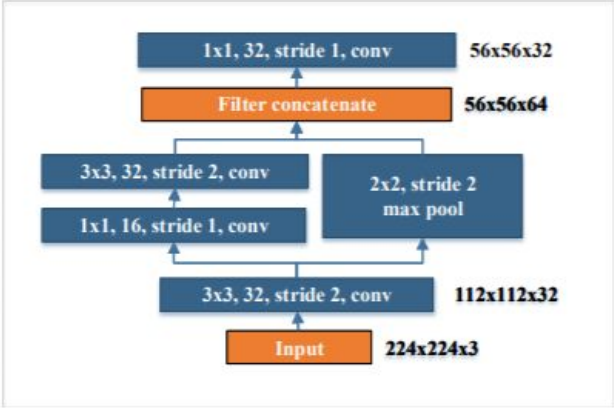


Figure 7: The stem block

Dynamic Number of Channels in Bottleneck Layer Another highlight is that the number of channels in the bottleneck layer varies according to the input shape instead of fixed 4 times of growth rate used in the original DenseNet.

Transition Layer without Compression: As the compression factor hurts feature expression it is not included in PeleeNet architecture.

Composite Function: To improve actual speed, the post-activation (Convolution - Batch Normalization - Relu) is used as a composite function instead of pre-activation used in DenseNet. For post-activation, all batch normalization layers can be merged with the

convolution layer at the inference stage, which can accelerate the speed greatly.

4. Implementation

In this section, we first discuss the methodology of data collection and processing. Then we introduce the architecture that was used to implement PeleeNet. After that, we discuss how different models' architectures were implemented in order to reproduce the results in the paper. Each model has a new feature of the PeleeNet added to the previous model in consideration. This is followed by a discussion of the algorithms used for the training of the PeleeNet model. We conclude the section by providing the software design process and file structure details used for the project.

4.1. Deep Learning Network

- **The PeleeNet adopted for the implementation:**

The PeleeNet model build has 4 stages. The first stage is the Stem block. It is followed by 3 stages consisting of a dense block and a transition layer. At each dense block, there are multiple dense 2-way dense layers. The transition layer between them consists of 1x1 convolution acting as a bottleneck channel and after that, there is 2x2 average pooling of stride 2 to decrease the feature map size. This is shown in Table 1

Stage		Layer	Output Shape
Input			224 x 224 x 3
Stage 0	Stem block		56x 56 x 32
Stage 1	Dense block	DenseLayer *3	28 x 28 x 128
	Transition Layer	1x1 convolution, stride 1	
		2x2 average pool, stride 2	
Stage 2	Dense block	DenseLayer *4	14 x 14 x 256
	Transition Layer	1x1 convolution, stride 1	
		2x2 average pool, stride 2	
Stage 3	Dense block	DenseLayer *8	7 x 7 x 512
	Transition Layer	1x1 convolution, stride 1	
		2x2 average pool, stride 2	
Stage 4	Dense block	DenseLayer *6	7 x 7 x 704
	Transition Layer	1x1 convolution, stride 1	
Classification layer		7x7 global average pool	1 x 1 x 704
		1000D fully connected, softmax layer	

Table 1: The PeleeNet architecture implemented

- **Architecture Spectrum:** As mentioned in the introduction of this section we have build models from baseline DenseNet-41[2] to the PeleeNet. But unlike paper, we have modularized the layers necessary for the model such that they can be called accordingly to add or remove a feature of the PeleeNet. Hence the new models which can be formed are represented as below in Table 2

Model Name	**Differential Description**
PeleeNet (m6)	Complete PeleeNet Architecture (figure-x)
Model 5 (m5)	Remove three dense layers from stage 4
Model 4 (m4)	Replace all two-way dense blocks with one-way dense blocks and use pre-activation instead of post
Model 3 (m3)	Remove stem block
Model 2 (m2)	Use static bottleneck channels and switch back to post-activation
Model 1 (m1)	Switch back to pre-activation
DenseNet-41 (m0)	Add compression to transition layers

Table 2: The differences in models

- **Training algorithm:** For training of the algorithm we used the concept of image augmentation[6]. Due to the image augmentation, the model becomes robust towards generalization on new data. Due to training time constraints, during the final model training, we removed the augmentation compromising a bit on accuracy. The model was trained using Adam optimizer. The optimizer choices which perform well on CNN task were RMSProp and Adam[7] but Adam outperformed the RMSProp and hence we finalized the Adam as our optimizer. The model was trained using mini-batch training rendering faster training time[8].

- **Data:** Original paper uses the ImageNet dataset which contains a total of 14,197,122 images with over 21,841 subcategories organized into 27 high-level categories for its final model training and class of dogs extracted from the same dataset for ablation study.

As mentioned in section 3.1, the dataset requires permissions and even after requesting for the same at ImageNet website[4], we couldn't get access. However, they provide URLs for each image according to which class that image belongs to. Using these URLs we have utilized work by Martins Frolovs[5] and built custom function according to our needs to download 198 classes with almost 500 images per class which occupied 14GB of disk space. Hereafter, ImageNet refers to this extracted subset. We have also trained our model on the CIFAR10 dataset which is readily available in Keras.

No.of Images	ImageNet	CIFAR10
Training	78117	50000
Validation	19643	10000

Table 3: Number of images in consideration

4.2. Software Design

The Readme file explains file structure, recommended system requirement and instructions about how to use the corresponding repository.

The main notebooks that need to be run in order to train the models are '**Run_this_cifar10**' and '**Run_this_imagenet**' to train on the CIFAR10 dataset and the ImageNet (198 classes) dataset respectively.

The folder 'imagenet_download' is a tool to download the 198 class subset of the ImageNet dataset. It will be utilized by the 'imagenet_utils.py' utility file.

The 'models' folder is just a location to save the model weights once they are trained.

Utils folder contains all utility functions and methods needed by the two main notebooks. 'imagenet_utils.py' contains functions that download, preprocess and return the ImageNet(198 classes) dataset. 'layer_utils.py' contains definitions of different architectural blocks (such as the stem block, two-way dense block, etc.) that form the building blocks of our models. 'model_utils.py' contains methods that implement the seven models under consideration.

The repository is organized into the following file structure[1]:

```
Repo:
|  README.md
|  requirements.txt
|  Run_this_cifar10.ipynb
|  Run_this_imagenet.ipynb
|
+---imagenet_download
|          classes_in_imagenet.csv
|          downloader.py
|          imagenet_class_info.json
|          prepare_stats.py
|          README.md
|          requirements.txt
|          words.txt
|
+---models
|  +---cifar
|  |       m0.h5
|  |       m1.h5
|  |       m3.h5
|  |       m4.h5
|  |       m5.h5
|  |       m6.h5
|  \---imagenet
|          m0.h5
|          m1.h5
|          m3.h5
|          m4.h5
|          m5.h5
|          m6.h5
|
\---utils
           imagenet_utils.py
           layer_utils.py
           model_utils.py
```

5. Results

5.1. Project Results

Models were built according to the architectural designs mentioned in Table 4 and trained on the ImageNet and CIFAR10 datasets. Number of model parameters and the time it took to train the models(for 20 epochs) are listed in the table below

Model	Imagenet		CIFAR10	
	Params	Time	Params	Time
Pelee-Net	2,251,430	1h 37min 42s	2,118,890	1h 3min 30s
m5	1,873,238	1h 31min 8s	1,740,698	1h 4min 16s
m4	1,238,514	49min 31s	1,105,974	41min
m3	1,224,342	3h 17min 21s	1,091,802	2h 12min 52s
m2	914,390	N/A	781,850	N/A
m1	1,049,398	2h 59min 21s	916,858	2h 18min 49s
Dense-Net-41	621,174	2h 32min 29s	554,810	2h 49min 41s

Table 4: Number of parameters, Running time, of models in ImageNet and CIFAR 10 data

OOM error explanation:- Model 2 threw OOM error due to insufficient RAM. However, all other models ran with the same amount of RAM (104GB). This could be a combined effect of two model components. Firstly, it doesn't contain Stem Block which causes input to be operated by dense block directly. This results in a 224*224*128 tensor as the first transition layer's convolution output which is comparatively larger than 56*56*128 which would have been the case with the presence of Stem Block. Secondly, for the initial convolution layer of Model 2, post-activation would make batch normalization happen on 128 features of shape 224*224, whereas pre-activation would reduce that to 3 features. These two reasons might have demanded larger memory and thus resulting in OOM error while training Model 2.

As seen from the curves and runtimes in Figures 8 and 9, PeleeNet has comparatively lower runtime and performs better than DenseNet. As we move in the spectrum from the DenseNet model towards the PeleeNet model, performance progressively increases.

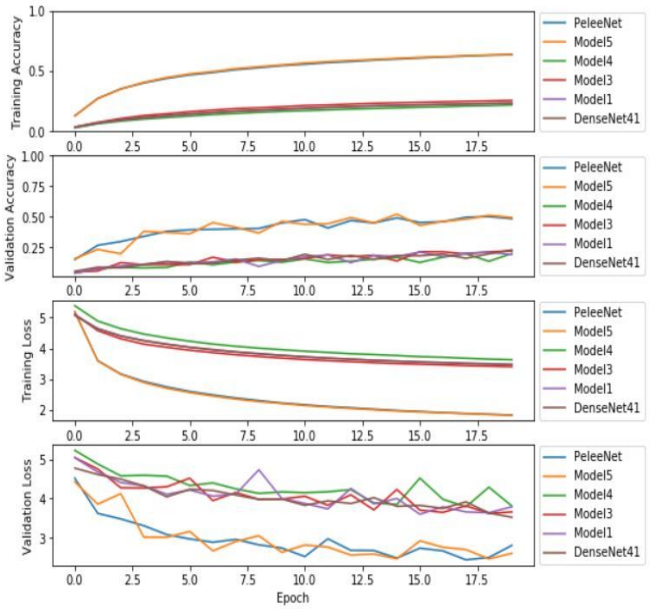


Figure 8: training accuracy and validation loss of different models on **Imagenet dataset**

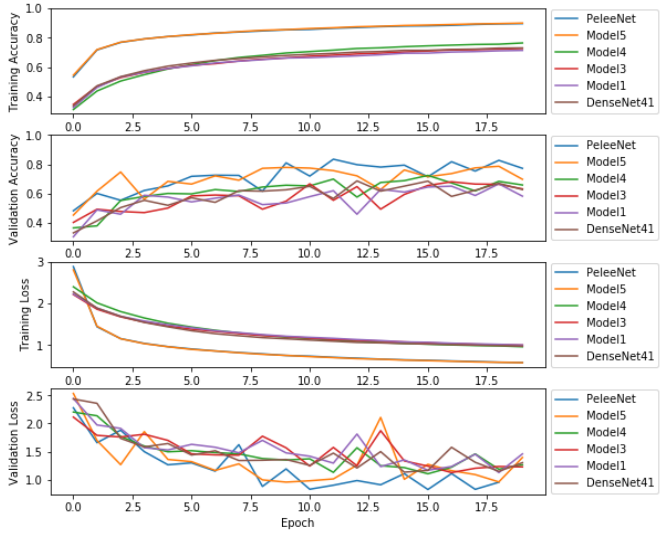


Figure 9: Training accuracy and validation loss of different models on **CIFAR10 dataset**

5.2. Comparison of Results

Accuracies obtained from our implementation and those claimed by the paper are listed in the table below.

Dataset	Accuracy (Top-1)
PeleeNet (Our implementation)	
ImageNet (198 classes)	48.25%
CIFAR10 (10 classes)	77.19%
PeleeNet (Original Paper)	
ImageNet (complete)	72.1%

Table 5: Comparison of our results with the original paper

The lower number of epochs due to time constraints and smaller datasets as a result of memory constraint accounted for reduced accuracy of our model when compared to that of the original paper.

Another result of the authors' that we tried to reproduce was the progressive improvements in model performance as the model slowly transitioned from DenseNet-41 to PeleeNet. We recreated the edge models (DenseNet-41 and PeleeNet) along with the five intermediate models. All these models were then trained on the CIFAR10 and the ImageNet(198 classes) dataset. Architectural features of each of these models along with the accuracies they produced are listed in the table below.

	DenseNet-41	m1	m3	m4	m5	PeleeNet
Transition layer without compression		*	*	*	*	*
Post-Activation					*	*
Dynamic bottleneck			*	*	*	*
Stem Block				*	*	*
Two-way dense layer					*	*
Add 3 extra layers to dense layers						*
Accuracy – ImageNet (198 classes)	23.03	19.33	22.20	20.15	49.51	48.25
Accuracy – CIFAR10 (%)	62.67	58.13	63.13	65.78	69.69	77.19
Accuracy – Imagenet(Stanford dogs modified subset - original paper)	75.02	76.10	75.80	76.80	78.80	79.25

Table 6: The accuracy measurement of models on ImageNet and CIFAR10

As we can see, the results are consistent with that of the authors', in that the model performance progressively increases as the model slowly transitions from DenseNet-41 to PeleeNet.

5.3. Discussion of Insights Gained

After a detailed study of the paper under consideration and implementing the PeleeNet from scratch the insights gained are summarized as follows:

- Though the number of parameters is higher in PeleeNet, it gets trained faster because of efficient design. Stem Block plays a major role in decreasing the training time of a model. (Refer to section 5.1 OOM error explanation)
- Post and Pre activation can have an impact on memory required for a model to train. This can be inferred via an explanation of OOM error for Model 2 given in Section 5.1. (Note that Model 2 and Model 1 differs only by pre and post-activation and Model 1 gets trained with the available memory). We could not increase RAM beyond 104GB as it is the maximum possible memory on a google cloud platform with 16 vCPUs, above which GPU compatibility is lost.(We attempted to use Tesla k80, Tesla V100, Tesla P100, Tesla T4 in all possible regions)
- Though the original paper trained the model for 120 epochs, we couldn't go over 20 epochs as it took a total time of over 24 hours for 20 epochs itself.
- Not having access to the original ImageNet dataset forced us to look for alternatives and finally, we scraped through URLs provided on its official website to get the data. Later, processing this data demanded a certain amount of time and effort.
- The original paper utilized the whole ImageNet dataset to train. However, we could not do that because of the limited resources mentioned in section 3.1. We have used a subset of ImageNet (198 classes with around 500 images per class)

6. Conclusion

The PeleeNet implementation in the paper differs from ours on the ground of hyperparameters. The batch size used in the paper was larger and the model was trained for more epochs resulting in better accuracy than ours. This indicates the difference made by the hyperparameter tuning on model training. Also, our results differed from the paper as we trained and validated the model on two different datasets namely Imagenet(Subset) and CIFAR10 affirming the ability of the PeleeNet to learn complex representational structures with a limited number of parameters. The differential models from baseline DenseNet-41 to PeleeNet gave us an astute ability to understand how each block and feature affects the number of parameters, running time and

accuracy of the model. In addition to producing the results related to PeleeNet we also gathered results on the running time of each intermediate model on two different datasets. To conclude, we were able to build a cost-efficient CNN model that can outperform standard CNN models with almost half the number of parameters.

7. Acknowledgment

In order to understand DenseNet architecture, growth rate and bottleneck concepts in the original paper, a comprehensive explanation in [3] acted as our go-to resource. Martin Frolov's work on scraping through URLs to download ImageNet data enabled us to download subset of ImageNet data.

8. References

[1]https://github.com/cu-zk-courses-org/e4040-2019fall-project-purj-ud2310-rn2494-jy3012
[2] Robert J. Wang, Xiang Li & Charles X. Ling " Pelee: A Real-Time Object Detection System on Mobile Devices" (arXiv:1804.0682)
[3] Gao Huang, Zhuang Liu, Laurens van der Maaten, Kilian Q. Weinberger "Densely Connected Convolutional Networks"
[4]ImageNet website http://image-net.org/
[5]https://github.com/mf1024/ImageNet-Datasets-Downloader
[6]https://towardsdatascience.com/image-augmentation-examples-in-python-d552c26f2873
[7]https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/
[8]https://www.geeksforgeeks.org/ml-mini-batch-gradient-descent-with-python/

9. Appendix

9.1 Individual student contributions in fractions - table

	rn2494	jy3012	ud2130
Last Name	Nene	Yadagani	Dinesha
Fraction of (useful) total contribution	1/3	1/3	1/3
What I did 1	Building Layers in utils	Getting and processing data into a compatible format	Building Models
What I did 2	Image generative model training code	Comparative Study between different models including plots and OOM error	Training the models on a GCP instance
What I did 3	Review of literature and report writing	Adding docstrings and comments to code	Readme.md, file structure and parts of the report.