Multi Scale DenseNet: Reproducibility and Applying GCN

Aviram Bar-Haim, aviramb@mail.tau.ac.il Iris Tal, iristal1@mail.tau.ac.il

January 31, 2018

1 Introduction

The popularity of using deep neural networks [DNN] for the task of image classification has increased substantially over the last years. While the accuracy of DNNs has improved in classification, the number of layers and parameters used has also increased - from 8 layers in Alexnet (2012) [1] to 152 layers in ResNet (2015) [2]. Training and using such complex models usually demand expensive computational resources with high power consumption. When developing an application for commonly used devices such as mobile phones, one must take into consideration constraints such as number of operations and efficient memory usage, when choosing his design principles. Our project is based on 2 recent papers in this field - Multi Scale Dense Network (MSDNet) [3] and Global Convolutional Network (GCN) [5], 2 network architectures addressing the problem in question. The former distingishes between "easy" and "hard" examples to classify and uses a DNN structure with multiple classifiers. An "easy" example can exit in an early stage and not waste unnecessary computations, while a hard example can be classified correctly on a deeper level of the network. The latter work focuses on an efficient implementation of convolutions requiring less parameters. This method can be utilized to use larger kernels in order to get a bigger receptive field and improve classification. The project focuses on the implementation of these projects and the results of their combination.

2 Papers Overview

The project described in this report is based on MSDNet and GCN, as mentioned previously. This section is a brief overview of the main ideas and principles explored in each paper. As MSDNet (Multi-Scale Dense Networks) is based on an architecture called DenseNet [4] it was added to the overview in order to provide one with a full background of the project.

2.1 DenseNet

The success of ResNet [2] has shown that networks can be substantially deeper, more accurate and efficient to train if they contain "short-cut connections". Those connections skip one or more layers and in ResNet's case simply perform identity mapping. Let the functionality (e.g a convolution, ReLU etc.) of a layer in a general network be denoted as $H_l(\cdot)$. The usual formula connecting 2 subsequent layers in a traditional network is $x_l = H_l(x_{l-1})$. The structure of ResNet contains skip-connections that bypass the non-linear transformations with an identity function:

$$x_l = H_l(x_{l-1}) + x_{l-1} \tag{1}$$

The skip layer enables gradients to flow directly through the identity function from later layers to earlier layers. This solves the problem of vanishing gradients by allowing direct information flow from non-subsequent layers.

The authors of DenseNet introduced a new approach based on this observation: for each layer, the feature-maps of all preceding layers are used as inputs, and its own feature-maps are used as inputs into all subsequent layers. The **dense** connectivity improves information flow between layers. Figure 1 illustrates a block (group of layers) with the suggested architecture. Mathematically, the formula describing the

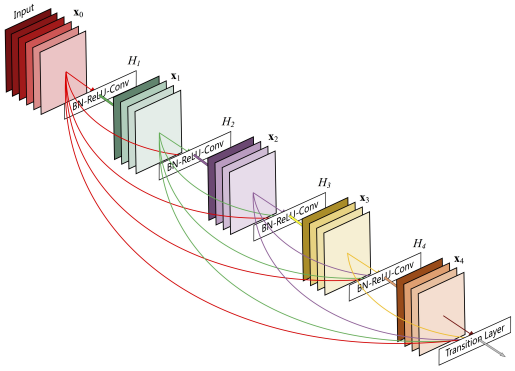


Figure 1: DenseNet basic building block with 5 layers and linear growth.

new connection between layer l input and output is:

$$x_l = H_l([x_0, x_1, ..., x_{l-1}]) \tag{2}$$

where $[x_0, x_1, ..., x_{l-1}]$ represents the concatenation of feature maps from all previous levels.

A few more important notations represented in the paper:

Transition Layers. Downsampling is an important operation in every network, and is performed by pooling layers. However, this operation collides with the concatenation introduced in eq. (2) because of the size difference between layers. The solution chosen by the authors is to divide the network into multiple densely connected dense blocks, and add between them transition layers. Transition layers consist of a batch normalization layer and a 1×1 convolutional layer followed by a 2×2 average pooling layer. An example of DenseNet with 3 blocks can be seen on figure 2.

Bottleneck Layers. A design in which a layer consisting of BN-ReLU-Conv(1×1) is inserted before a 3×3 convolution. This is helpful to reduce computational effort since it decreases the number of input layers.

Growth Rate. If each layer produces k feature maps, it follows from the DenseNet design that the l^{th} layer input is $k \times (l-1)$ feature maps. k is called the growth rate of the network.

2.2 MSDNet

Multi-Scale Dense Networks are a combination between the ideas of cascading classifiers (Viola and Jones [6]) and DNN. The authors observe that achievements from recent years in image classification go hand in hand with high computational demands. Simple networks can achieve a decent error. The need for complicated, expensive models originates in the challenge of classifying "hard" examples. A comparison between "easy" and "hard" examples is visualized in figure 3. The demand of integrating DNNs in mobile applications and edge devices is on the rise, meaning power saving becomes a constraint and should come to mind when designing a classification network.

MSDNet is a suggested solution to the problem presented above. The writers implemented a cascade of intermediate classifiers throughout the network. Instead of passing an image through an entire network, one can determine an early exit to each image by a thershold of computational budget.

2.2.1 Design Principles

The design principles that guided the authors are a result of 2 problems arising when trying to integrate classifiers in the bulk of a network.

Multi Scale. The first problem caused by inserting classfiers in a middle of a network is the lack of global information, caused by missing coarse level features. In a traditional network, the early layers learn fine-scale features based on a small receptive field, whereas later layers learn coarse-scale features based on a large receptive field. Inserting a classifier naively in a middle of a network won't achieve satisfying results since global information on the picture will be missing. The authors solved this by maintaining a feature representation at multiple scales throughout the network for each layer. The feature maps of a layer in a specified scale are computed by concatenating the result of the previous layer and same-scale features with the results of one or two convolutions: 1. the result of a regular convolution applied on the same-scale features from the previous layer (horizontal connections) and, if possible, 2. the result of a

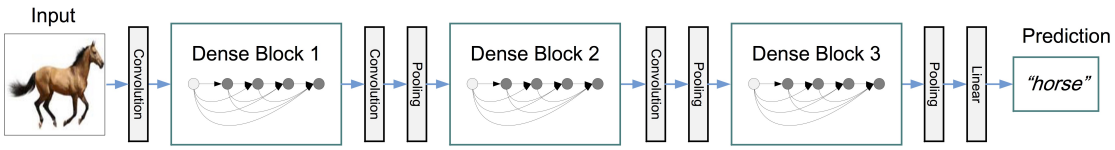


Figure 2: A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature-map sizes via convolution and pooling.

Figure 3: Two images of a horse: the left image is an "easy" example, whereas the right image requires a much more expensive network architecture.

strided convolution applied on the finer-scale feature map from the previous layer (diagonal connections). The horizontal connections can be seen as a traditional progress of a DNN from fine-scale details to coarse level. The vertical connections in each level are an adaptation in order to make classification plausibly possible in early levels of the network.

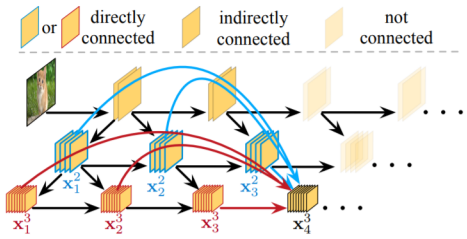
DenseNet. The second problem of introducing classifiers in the middle of the network is a degradation in accuracy - the authors state that ResNet loses 7% accuracy after inserting a single intermediate classifier. Using the dense connection blocks from DenseNet (introduced earlier) solves this issue: each layer is connected with all subsequent layers which enables features to bypass and reach the final classifier, maintaining its accuracy. In contrast to naive implementation, information from earlier stages cannot "collapse" or vanish after an intermediate classifier, since it is passed directly to the following layers.

2.2.2 Problem Setup

The article distinguishes between 2 computational constraints: Anytime prediction and Budgeted batch classification (ensemble).
Let us denote $\mathcal{D}_{test} = \{x_1, ..., x_M\}$, where x_i is the i^{th} test example.
Anytime Prediction. In this constellation [7], a budget $B > 0$ is assumed to be available for every example. The budget is nondeterministic and varies per test instance. Another assumption is that the budget is drawn from a general distribution $P(x, B)$. If we denote the loss function by $L(\cdot)$, we can define $L(f) = \mathbb{E}[L(f(x), B]$. This loss function is evaluated over an average of samples from $P(x, B)$ as commonly done in the empirical risk minimization framework.
Budgeted batch classification. In this setting, a budget $B > 0$ is assumed to be available for a set of examples $\mathcal{D}_{test}$. The loss $L(f(\mathcal{D}_{test}))$ needed to be minimized across all examples in $\mathcal{D}_{test}$ within a cumulative cost bounded by B.

2.2.3 Architecture

The network architecture is illustrated in figures 4, 5. In this project we chose to focus on CIFAR-10 and CIFAR-100 due to resource and time limits.
First Layer. Let us denote the output feature maps at layer l and scale s as x_l^s, and the input image as x_0^1. The first layer is an initialization of all levels of scale, and the only one containing direct vertical connections. Each vertical connection in the first layer is down-sampling the convolution output of the finer scale above, as seen in figure 4. h_1^s represents a sequence of 3×3 convolution, batch normalization [BN] and ReLU. $\tilde{h}_l^s$ is similar except the convolution is strided. The network implemented in the project



x_ℓ^s	$\ell=1$	$\ell=2$	$\ell=3$	$\ell=4$
$s=1$	$h_1^1(\mathbf{x}_0^1)$	$h_2^1\left(\mathbf{x}_1^1\right)$	$h_3^1\left(\left[\mathbf{x}_1^1,\mathbf{x}_2^1\right]\right)$	$h_4^1\left(\left[\mathbf{x}_1^1,\mathbf{x}_2^1,\mathbf{x}_3^1\right]\right)$
$s=2$	$\tilde{h}_1^2(\mathbf{x}_1^1)$	$\left[\begin{array}{c}\tilde{h}_2^2\left(\mathbf{x}_1^1\right),\\ h_2^2\left(\mathbf{x}_1^2\right)\end{array}\right]$	$\left[\begin{array}{c}\tilde{h}_3^2\left(\left[\mathbf{x}_1^1,\mathbf{x}_2^1\right]\right),\\ h_3^2\left(\left[\mathbf{x}_1^2,\mathbf{x}_2^2\right]\right)\end{array}\right]$	$\left[\begin{array}{c}\tilde{h}_4^2\left(\left[\mathbf{x}_1^1,\mathbf{x}_2^1,\mathbf{x}_3^1\right]\right),\\ h_4^2\left(\left[\mathbf{x}_1^2,\mathbf{x}_2^2,\mathbf{x}_3^2\right]\right)\end{array}\right]$
$s=3$	$\tilde{h}_1^3(\mathbf{x}_1^2)$	$\left[\begin{array}{c}\tilde{h}_2^3\left(\mathbf{x}_1^2\right),\\ h_2^3\left(\mathbf{x}_1^3\right)\end{array}\right]$	$\left[\begin{array}{c}\tilde{h}_3^3\left(\left[\mathbf{x}_1^2,\mathbf{x}_2^2\right]\right),\\ h_3^3\left(\left[\mathbf{x}_1^3,\mathbf{x}_2^3\right]\right)\end{array}\right]$	$\left[\begin{array}{c}\tilde{h}_4^3\left(\left[\mathbf{x}_1^2,\mathbf{x}_2^2,\mathbf{x}_3^2\right]\right),\\ h_4^3\left(\left[\mathbf{x}_1^3,\mathbf{x}_2^3,\mathbf{x}_3^3\right]\right)\end{array}\right]$

Figure 4: MSDNet first block architecture. The connections between layers are illustrated on the left, mathematical expressions for feature maps of each layer is organized in a table on the right. More details on h and $\widetilde{h}$ in section 2.2.3.

contains 3 scales, following the design described in the article for CIFAR-10 and CIFAR-100.

Subsequent Layers. h_l^s stands for conv(1×1)-BN-ReLU-conv(3×3)-BN-ReLU. The number of output channels of the three scales is set to be 6, 12 and 24, respectively. The input to each layer is a concatenation of all previous layers' feature maps in the same scale and all previous layers' feature maps from an upper scale after strided convolution.

Block Division. The network is divided into blocks of multi scale DenseNet, at the end of each block there is a classifier. The first block of MSDNet is denoted as the **base**, containing a few layers.

Classifiers. Each classifier f_k consists of two downsampling convolutional layers with 128 dimensional 3×3 filters, followed by a 2×2 average pooling layer and a fully-connected layer (10 nodes for CIFAR-10 and 100 nodes for CIFAR-100).

Loss function. Cross-entropy is used as a loss function $L(f_k)$ for all classifiers and a weighted sum (with same weights) is minimized: $\frac{1}{|\mathcal{D}|}\sum_{(x,\,y)\in\mathcal{D}}\sum_k L(f_k)$.

Transition Layers. In order ro reduce computational efforts, the authors used the concept of transition layers previously explained in section 2.1. The size of the network is reduced by keeping only the coarsest $(S-floor(\frac{max(0,\,i-2)}{nLayers}\cdot S)$ scales in the i^{th} layer, where S is the initial number of scales. A transition layer is then added in order to coordinate the output and input of two adjacent blocks, and reduce number of features by half, with 1×1 convolution.

2.3 GCN

In the GCN paper [5] the authors address the problem of semantic image segmentation as a combination of 2 tasks: 1) classification, as every pixel in a segmentation map should be classified to the correct object category, and 2) localization, as every group of labeled pixels belonging to a certain category should be aligned to the appropriate coordinates in output score map. As our projects subject is classfication, we shall focus in this brief overview on the classification aspects introduced in the paper.

The foundational observation in [5] is that classifiers are densely connected to the entire feature map via fully connected layer or global pooling layer which makes features robust to locally disturbances and allows classifiers to handle different types of input transformations. Moreover, the receptive field also plays a role in classification - the intuition behind is that in order to classify an object correctly, the neuron must be able to 'see' the entire object. Although networks like ResNet have a theoretical large receptive field due to their deep structure, it has been shown by Zhou et al. [8] that only a portion of pixels in the theoretical receptive field actually influence the outcome of the network. Those pixels are called VRF (valid receptive field) in the paper.

Global convolutional network (GCN) is suggested as a building block intended for computing convolutions spreading over a large kernel with less parameters involved. This is done by combining $1\times k+k\times1$ and $k\times1+1\times k$ convolutions as seen in figure 6. This de-

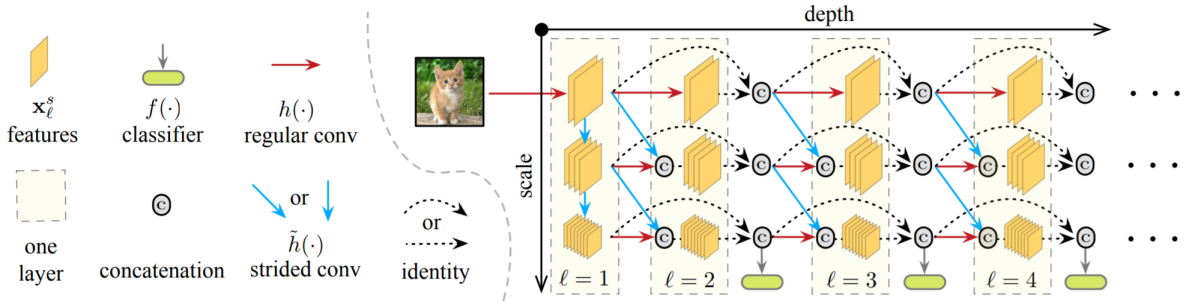


Figure 5: MSDNet architecture. The example illustrated here is of 3 blocks, each is followed by a classifier.

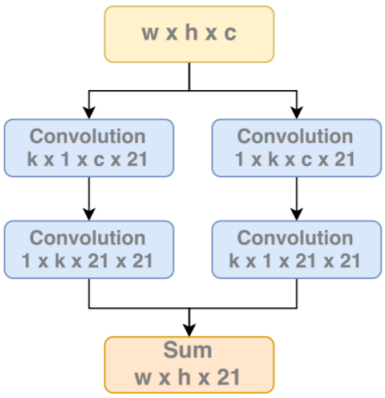


Figure 6: GCN basic block

sign enables keeping $2k$ parameters per filter, instead of k^2, saving memory and making large kernel sizes practical. A large kernel convolution can replace a stack of small kernel convolutions, e.g. two convolutions of 3×3 can be replaced by 5×5 convolution, etc.

3 Project Proposal

This project suggests a way to further decrease the computational demands of MSDNet, and specifically its number of parameters, by fusing between the architecture of MSDNet and the convolutional block of GCN. We divide the process into 3 stages: 1) implement MSDNet in pytorch framework and reproduce the results presented in the article, 2) replacing the spatial convolutions in the first layer of MSDNet with GCN blocks, thus needing less layers in the base, 3) replacing spatial convolutions in the whole network with GCN blocks. We anticipate stages 2 and 3 will result in improving the runtime of the network due to less layers and parameters needed to compute feature maps. Regarding accuracy we assume the network performance will keep similar results to the original MSDNet, based on experiments conducted in [5].

4 Implementation Details

In order to build, train and evaluate MSDNet models with and without GCN integration, we used two different code bases: 1) the original implementation in torch [https://github.com/gaohuang/MSDNet], provided by the original paper's writers, and 2) a pytorch implementation we wrote [https://github.com/avirambh/MSDNet-GCN], that includes the GCN blocks integration. This section describes the basic design decisions and principles that we used, in order to take advantage of the auto gradient mechanism in pytorch and keep the code simple and easy to extend with new variants.

Our first decision was to use pytorch framework. One of the reasons we chose to use pytorch is that it uses the same torch tensor mechanism like Lua, which is helpful in keeping our comparisons as fair as possible. An important design approach we chose is to

use pytorch ModuleList, in order to sign any of our new submodules to Autograd mechanism (instead of explicitly implementing the backward pass between different disconnected subgraphs). We implemented the MSDNet graph using 6 main modules: 1) MSDNet module: holds a ModuleList of blocks of layers and classifiers, where each block is a Sequential module with MSDNet and transition layers. The first half of the ModuleList holds the different sequential blocks of layers, and the second half holds the corresponding classifiers to the different blocks. 2) MSDNet first layer: this is the first layer of the first sequential block, holds a ModuleList with the different scales of convulotion sequentials, creating the initial feature maps for the network. 3) MSDNet layer: the most common component in MSDNet. It holds a ModuleList with different scales of DenseNet, involving 1 or more scales of feature maps from the previous layer and the Identity of the same scale feature map of the previous layer. 4) DynamicInputDenseBlock: a generic Module that is being used as a wrapper for the different scales in MSDNet layer. it performs the densnet concatenation operation between the Identity of the same scale in the previous layer, and different feature maps of convulotion operation of 1 or more scales in the previous layer. 5) Transition layer: Performs 1x1 convolution to decrease number of channels after reducing the spatial dimension. 6) Classifier: builds the last sequential block of each classifier, taking the output of the coarsest layer in the last layer of a block, and returning a prediction of 10 or 100 classes using small convnet, according to the CIFAR dataset we use.

For the GCN experiments, we implemented another module of global convolutional network, holding 4 different spatial convolution modules, used as a separable kernel. In the second part of our project, we plugged in the GCN modules with different kernel sizes, instead of the spatial convulotion modules, according to the relevant settings of the experiment (section 5). As described in the GCN paper and in the previous section, a stack of two 3x3 convulotions can be replaced by one 5x5 convolution, in order to get the same theoretical receptive field. This analysis led us to test the outcome of replacing the spatial 3x3 convulotion modules inside a block with $5\times1+1\times5$ GCN convolution, and with larger kernels as well.

As will be described in the next section, we focused on 3 main metrics in our tests: number of flops, number of parameters and runtime on a commonly used vcpu. In the original paper, there is an explanation of a method called 'lazy evaluation', aimed to measure only relevant scales for each classifier. There are different parts that affect the metrics calculation as well, such as transition layer policy, bottleneck policy etc. We tried to keep the measurements as 'apples to apples' within every specific comparison, although the flops counting details wasn't mentioned in the original paper nor provided, and was one of the biggest challenges we had to deal with. We used Google Cloud Platform for all of our experiments, using 1 vCPU, 7.5 GB RAM and Nvidia K80 GPU for a specific training.

5 Experiments

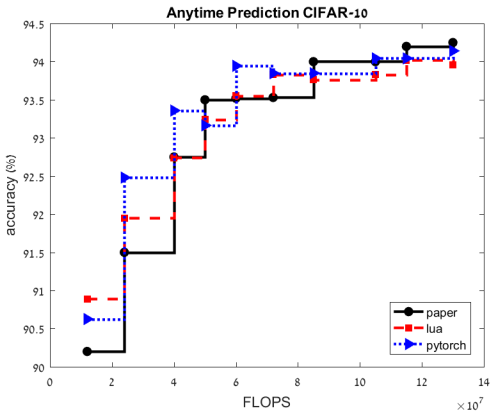
As mentioned before, we chose to use 2 datasets in our training: CIFAR-10 and CIFAR-100, due to the cost of also training on the ImageNet dataset like the original MSDNet paper [3] did. Moreover, the setting of anytime prediction was chosen for all the experiments, due to time and resource constraints.

General training details. All models are trained similarly to the original MSDNet settings - by using stochastic gradient descent (SGD) with mini-batch size 64, plus updating with Nesterov momentum with a momentum weight of 0.9 without dampening, and a weight decay of 10^{-4} . All models are trained for 300 epochs, with an initial learning rate of 0.1, which is divided by 10 after 150 and 225 epochs.

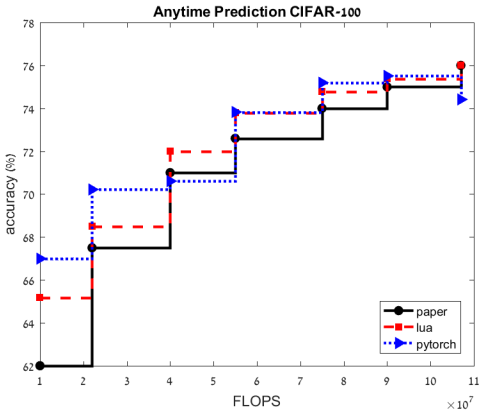
5.1 Reproduction

Before applying GCN-blocks on our pytorch implementation, we needed to validate our implementation performs similarly to the description in the paper. We compare ourselves both to the results brought in the article and results obtained by running the published code in lua. The comparison is presented in figure 7.

We can conclude from the graphs that our implementation is highly comparable with the original one. All results have mild differences, which is reasonable considering this is an optimization process applied on a non-convex problem, having more than one local minima. We shall denote from now on the pytorch implementation as 'baseline'.



(a)



(b)

Figure 7: Top-1 accuracy results for CIFAR-10 (a), and CIFAR-100 (b) obtained in pytorch implementation, compared to results presented in the article and in original lua code.

5.2 Applying GCN on first layer

The aim of this experiment is to test if replacing one regular convolutional block with GCN convolutional block does not cause substantial deterioration of performance. The replacement is performed in the first layer, as it is the seed of all feature maps in the network and has a wide impact on the results. When using 5×5 separable kernel we remove one layer from the base (first block), and when using 7×7 separabale kernel we remove 2 layers from the base. This is due to the logic explained earlier that a stack of two 3x3 convulotions can be replaced by one 5x5 convolution, and a stack of three 3x3 convulotions can be replaced by one 7x7 convolution. Prior assessment of the number parameters in each constellation led to the conclusion it is expected to grow. This is an intermediate stage in replacing all spatial convolutions with GCN convolutions, a step which is supposed to reduce drastically the number of parameters in the network.

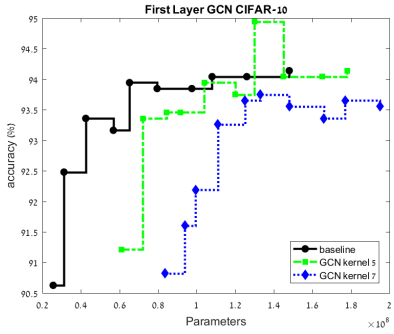
Results are depicted in figure 7. Parameters measured are of the entire network (feature maps+classifiers). The main conclusion from these graphs is that using GCN convolution with 5×5 kernel does not damage the accuracy of the network. This is deduced from comparing accuracy per classifier, and valid for both datasets. However, using GCN kerenel in the size of 7×7 causes a deterioration in performance. Due to this conclusion, we decided to focus on GCN kernel 5 in the next experiment.

5.3 Applying GCN on all layers

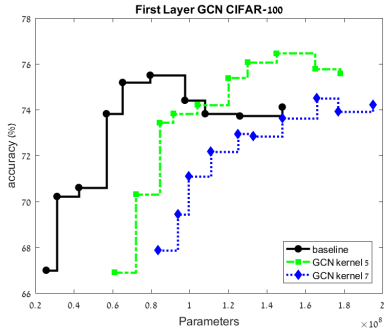
This experiment setting is a baseline MSDNet with **all** spatial convolutions in the network (excluding classifiers) replaced with GCN blocks. We chose 5×5 separable kernel size due to the results of previous experiment in 5.2, each convolution replacing 2 layers in the original model. Two variations were trained: the first containing 3 layers in the base block, and the second containing only 2 layers. Performance evaluation is presented in figure 9.

We measured only parameters associated with the parts we changed in the network. This means parameters of the classifiers, which are the same in both net-

works, were not included. Our main conclusion is we succeeded in our mission to reduce the number of parameters in the network - amount of parameters was decreased by 60%. As for accuracy, we managed to obtain the same accuracy. Classifiers in the begginning and end of the network do not show improvement, because the same accuracy could be achieved with a similar amount of parameters in both baseline and MSDNet-GCN. However, in the middle of the network both MSDNet-GCN variations achieve higher accuracy with same amount of parameters than the baseline.

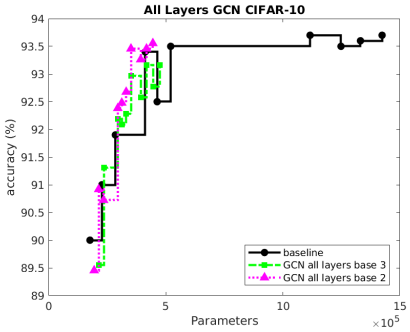


(a)

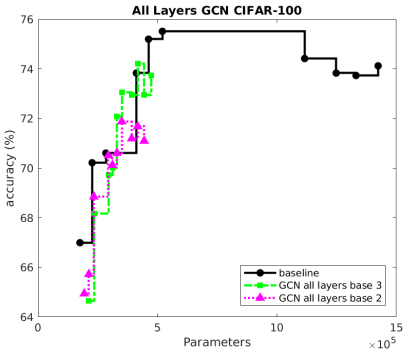


(b)

Figure 8: Top 1 accuracy results for CIFAR-10 (a), and CIFAR-100 (b), comparing baseline model to baseline model with GCN convolution on the first layer. For more information, please see section 5.2.



(a)



(b)

Figure 9: Top 1 accuracy results for CIFAR-10 (a), and CIFAR-100 (b), comparing all layer GCN to baseline model and to first layer GCN model. For more information, please see section 5.3.

5.4 Runtime

During the experiments performed for this project, a critical question arose: how can one compare between budgets of different architectures? Measuring FLOPs as a way to evaluate computational budget is not entirely accurate, as the run time of a program depends on more factors. Such factors are the ability to perform calculations in parallel, amount of time taking to retrieve data from memory, internal hardware limitations and more. This motivated us to conduct a mini experiment, comparing the runtime of a single image going through the entire network, to see if the reduction of parameters truely makes a difference in decreasing a budget of a network. This experiment was conducted on a vCPU with 4 GB RAM, in order to simulate an average computing environment. Results can be found in table 1.

	Runtime [sec/image]
baseline	0.023
MSDNet first layer (5.2)	0.023
MSDNet-GCN base 3	0.018
MSDNet-GCN base 2	0.017

Table 1: Runtime experiment for a single image on different MSDNet variations, performed on a vCPU with 4 GB RAM.

We can deduce from the table that new variations of MSDNet with GCN presented in this project have an improved runtime, up to 26%. This result is another justification for our approach that lowering a budget of a network can be done with parameter reduction.

6 Conclusions

We would like to divide our conclusions into 3 parts: the obstacles in implementing the system described in the paper, important points rising from the project and ideas for more research.

Difficulties in reproducing results. There are some missing clarifications about important features in the architecture of MSDNet. As far as we understand, the formula for calculating the position of transition layers is not accurate, and none of the figures depicts the entire architecture of the network. MSDNet has a bottleneck shape due to transition layers, thus figures 4 and 5 can be misleading. Moreover, the way FLOPs are measured is not elaborated. When calculating FLOPs by only summing the number of addition and multiplication operations performed, the number of FLOPs used in the baseline turned out to be larger than the paper. After manually trying other schemes to calculate FLOPs, our best guess is the authors only counted the FLOPs needed for creating feature maps, and excluded FLOPs caused by the classifiers.

Conclusions. Combining MSDNet and GCN can indeed improve the performance of MSDNet. We achieved both 1) 60% reduction in number of parameters of the feature maps, 2) improvement of accuracy per parameter in the middle of the network (section 5.3). We also demonstrated that reduction of parameters translates into faster runtime, which justifies the purpose of this project.

Further research. There are a few directions we would like to investigate in the future. First one is to repeat experiments with budgeted batch classification (section 2.2.2), which we could not complete due to time and resource constraints. Moreover, we would like to design a MSDNet-GCN with only GCN convolutions, both in convolutional layers and classifiers. Lastly, we would also like to apply the technical approach in this project on larger datasets (such as Imagenet) and on different research areas (such as semantic segmentation, where GCN was originated). It is interesting to try to design a more complex model that will fit larger datasets better but would not lengthen the runtime of the original one with GCN convolution.

References

[1] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. In NIPS, pp. 1106–1114, 2012.

[2] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In CVPR, 2016.

[3] G. Huang, D. Chen, T. Li, F. Wu, L. van der Maaten, and K. Q. Weinberger. Multi-scale dense convolutional networks for efficient prediction. arXiv preprint arXiv:1703.09844, 2017.

[4] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten. Densely connected convolutional networks. In CVPR, 2017.

[5] C. Peng, X. Zhang, G. Yu et al., "Large Kernel Matters--Improve Semantic Segmentation by Global Convolutional Network," arXiv preprint arXiv:1703.02719, (2017).

[6] P.A. Viola, M.J. Jones, Rapid object detection using a boosted cascade of simple features, in: CVPR, issue 1, 2001, pp. 511–518.

[7] Alexander Grubb and Drew Bagnell. Speedboost: Anytime prediction with uniform near-optimality. In AISTATS, volume 15, pp. 458–466, 2012.

[8] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba. Object detectors emerge in deep scene cnns. arXiv preprint arXiv:1412.6856, 2014.