Systematic Disconnection in Densely Connected Convolutional Networks (DenseNet)

Ofir Azulay
ofirazu@post.bgu.ac.il

Moshiko Cohen
moshe8@post.bgu.ac.il

Abstract

Convolutional Neural Networks (CNNs) often encounter information flow issues where the input information can seemingly 'vanish' as it passes through multiple layers until it reaches the end of the network. To ensure maximum information flow between layers in the network Dense Convolutional Network (DenseNet) architecture was developed. DenseNet connects each layer to every other layer (with matching feature-map sizes) in a feed-forward fashion. 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. DenseNets present several notable advantages: they effectively address the vanishing-gradient problem, enhance feature propagation, promote feature reuse, and significantly decrease the number of parameters. However, the dense connections can introduce computational complexity and memory requirements, limiting their usability in real-time applications and resource-constrained environments. This research aims to analyze how the number of connections affects model performance. By comparing different models with varying connection numbers between layers to the standard dense architecture, we seek to identify patterns and trends. Each model examines a specific number of connections, which determines the number of preceding layers that contribute their output feature maps as inputs to a given layer. The evaluation of six different architectures, which differed in the number of layers within each block (L) and growth rate (k), revealed that connecting layers within a block enhances model accuracy. However, due to limited resources, a more practical approach could be to connect fewer previous layers. This approach still achieves commendable convergence to accuracy levels comparable to a fully dense network that requires substantial computing resources. Code is available at https://github.com/ofirazulay/Deep_Learning_Project.git

1. Introduction

Convolutional neural networks (CNNs) have emerged as the dominant approach in machine learning for visual object recognition. However, as CNNs get deeper, they encounter the "vanishing gradient" problem. This problem occurs when the information or gradient traversing multiple layers gradually diminishes or fades away, ultimately "washing out" by the time it reaches the network's end or beginning. In response to this challenge, researchers have proposed various methodologies [2,3,4] all sharing a common characteristic: the incorporation of shorter connections that link early layers to later layers. Recent works have demonstrated that the inclusion of shorter connections between layers close to the input and those close to the output in convolutional networks can lead to significant improvements in accuracy and training efficiency [5,6,7]. Based on this idea, DenseNet is a prominent architecture that effectively addresses the "vanishing gradient" challenge by facilitating an optimal flow of information between network layers [1]. In DenseNet, all layers are directly connected to all subsequent layers. Below is a figure illustrating this layout schematically:

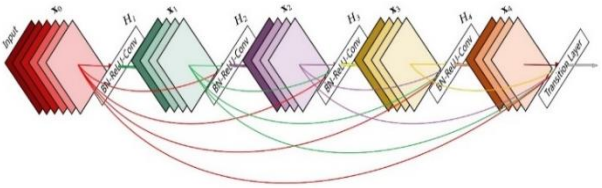

Figure 1 *- A 5-layer dense block. Each layer takes all preceding feature-maps as input [1].*

Consider a single image x_0 that is passed through a convolutional network. The network comprises L layers, each of which implements a non-linear transformation $H_l(\cdot)$ where l indexes the layer. We define $H_l(\cdot)$ as a composite function of three consecutive operations: batch normalization (BN), followed by ReLU function and convolution (Conv). We denote the output of the l^{th} layer as x_l.

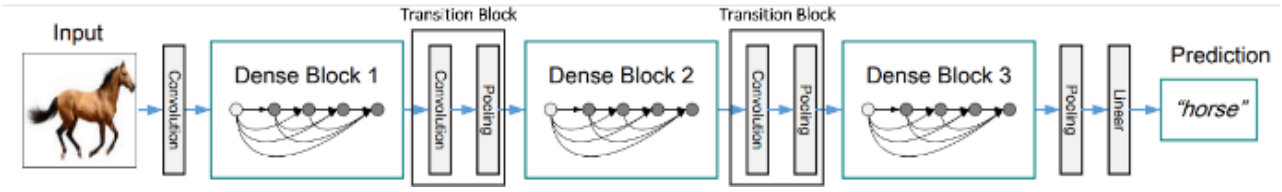

***Figure2** - A deep DenseNet with three dense blocks [1].*

To further improve the information flow between layers, the l^{th} layer receives the feature-maps of all preceding layers: $x_0, x_1, \ldots, x_{l-1}$ as input:

$$x_l = H_l([x_0, x_1, \ldots, x_{l-1}]) \qquad (1)$$

where $[x_0, x_1, \ldots, x_{l-1}]$ refers to the concatenation of the feature-maps produced in layers 0,1, ..., $l-1$. $H_l(\cdot)$ function produces k feature maps of its own that passes on to all L−l subsequent layers. Hence, the l^{th} layer has $k_0 + k \cdot (l-1)$ input feature-maps, where k_0 is the number of feature maps in the input layer. We refer to the hyperparameter k as the growth rate of the network - the number of feature maps that are added to each layer. The growth rate regulates how much new information each layer contributes to the network.

This DenseNet architecture introduces $\frac{L(L+1)}{2}$ connections in an L-layer network, instead of just L, as in traditional architectures. The concatenation operation can be performed between layers with the same feature-map size. However, an essential part of convolutional networks is down-sampling layers that change feature-maps size. To facilitate down-sampling in DenseNet architecture we divide the network into multiple densely connected dense blocks, as shown in *Figure 2.* Each dense block consists of convolution layers. After a dense block, a transition layer is added to proceed to the next dense block. The transition layers change feature-map sizes via convolution and pooling.

DenseNets have several compelling advantages: they alleviate the vanishing-gradient problem, strengthen feature propagation, and encourage feature reuse because they create short paths from early layers to later layers. In addition, DenseNet substantially reduces the number of parameters by avoiding the need to relearn redundant feature maps. This architecture explicitly distinguishes between information that is added to the network and information that is preserved. Each DenseNet layer contributes only a small set of feature maps to the "collective knowledge" of the network while keeping the remaining feature maps unchanged.

Along with these advantages, the dense connections lead to an increased computational complexity. The need to store feature maps on a large-scale during network training, accompanied by the extensive number of convolutional operations, leads to significant memory requirements and increases computational complexity. These factors can impose practical limitations and extend training times. These implications are critical for real-time applications and resource-limited environments.

This research examines the influence of the number of connections in the DenseNet network on model performance. By changing the number of connections in the architecture, the research will evaluate the effect on various performance indicators such as model accuracy and computational efficiency expressed in training time and the number of parameters learned. By conducting a comparative analysis of different models in comparison to the DenseNet base architecture, we aim to identify patterns and trends. Each model is configured with a different value for the number of preceding layers whose output serves as input for a specific layer. In this way, we can achieve a balance between the number of connections and the quality and efficiency of the model. Consequently, it provides valuable insights for improved network planning, specifically regarding the number of connections and their specific placement based on specific requirements.

2. Proposal

In the field of deep learning, particularly in neural networks, the quest for efficient models becomes more significant in environments with limited computational resources. To enhance efficiency, we explore the effects of disconnecting connections in DenseNet networks. In this research, we propose a different connectivity pattern - by systematically and non-randomly disconnecting connections, we aim to reduce the computational complexity in resource-constrained environments while maintaining network performance.

Network connectivity structure

We define a new hyperparameter P as the number of layers preceding a particular layer in a block whose outputs (feature-maps) must be transferred as inputs to this layer as described in the following formula:

$$x_l = H_l([x_{l-p}, x_{l-p+1}, \ldots, x_{l-2}, x_{l-1}]) \quad (2)$$

For example, when $P = 2$ and each network block consists of 4 convolutional layers, then the 4^{th} layer will receive two inputs through concatenation: the output of the 3^{th} layer and the output of the 2^{th} layer. In the special case, when the value of the hyperparameter P is equal to the number of layers plus one $(P = L + 1)$ we obtain a complete DenseNet network with all possible connections from each layer to all the layers that follow it. By assigning varying values to the hyperparameter P, we can define multiple models, each having a unique connectivity structure within the network blocks. The following figure illustrates the block structure in each model, utilizing different P values:

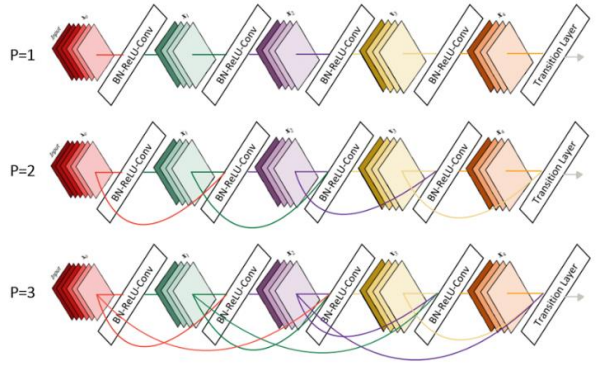

***Figure 3** - Connectivity structure within the DenseNet blocks by assigning varying values to the hyperparameter P.*

In *Figure 3*, different values of hyperparameter P are initialized for a DenseNet-like network block. For example, when $P = 1$, each convolutional layer in the block receives as input only the feature maps produced by the previous layer .However, when $P = 2$, each layer in the block receives feature maps not only from the previous layer but also from the layer two steps before it in the block. This means that the connectivity becomes more expansive, and each layer has access to information from a wider range of preceding layers. As the value of P increases, the connectivity pattern becomes even more extensive, with each layer incorporating information from a larger number of preceding layers. By exploring different values of P, various connectivity structures are defined, enabling the creation of multiple models tailored to specific computational resource constraints and performance requirements.

3. Methodology

Network Architectures

To explore how the number of connections to the previous layers impacts the performance of a DenseNet convolutional network, we will examine several network architectures. We train networks with different depths L (Number of layers in each block), and different growth rates k (the number of feature maps that are added to each layer). The exact networks configurations we used on our dataset are shown in *Table 1.*

Architecture	Configuration
Semi-DenseNet -1	{L=6 , k=6}
Semi-DenseNet -2	{L=6 , k=12}
Semi-DenseNet -3	{L=9 , k=6}
Semi-DenseNet -4	{L=9 , k=12}
Semi-DenseNet -5	{L=12 , k=6}
Semi-DenseNet -6	{L=12 , k=12}

Table 1 - Tested Architectures configurations

Each architecture used in our experiments has three dense blocks that each has an equal number of layers.

Before entering the first dense block, a 2D convolution with 24 filters, kernel size 3×3, and strides 1×1 is performed on the input images. This convolution operation is followed by Batch Normalization and then ReLU function. At the end of this initial stage, Max Pooling is performed (Max pooling is a pooling technique where the filter simply computes and picks the maximum pixel value in the receptive field of the feature map).

Each convolution layer in each block consists of the following consecutive operations: batch normalization (BN), followed by ReLU function and convolution operation with kernel size 3×3 and the number of filters depending on the growth rate. At the end of each convolution layer dropout operation with 0.2 rate is performed (the dropout operation randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting).

After each block, a transition block is applied. The transition block incorporates several operations to reduce the spatial dimensions of the feature maps. The transition layers between two contiguous dense

blocks consist of Batch Normalization, ReLU, 2D convolution operation with kernel size 1×1 and 2D Average Pooling with 2×2 pool size and strides 2×2. These steps collectively contribute to down-sampling the feature maps.

At the end of the last dense block, after Batch Normalization and ReLU, a 2D global average pooling is performed and then a SoftMax classifier is applied. The output of this layer represents the final output of the network, providing class probabilities for the given input.

During our experiments, we will systematically test all possible *P* values for each predefined architecture. The range of *P* will span from connecting only one previous layer to connecting all previous layers (Classic DenseNet structure). Each initialization of a specific *P* value will yield a distinct model with a unique connectivity structure.

Dataset

To evaluate the performance of the architecture with a different number of connections, we conducted experiments on an unbalanced binary classification problem. The dataset used for these experiments consisted of 17,500 images sourced from the Kaggle website [8]. Within this dataset, a subset of the images (75%) was labeled as cacti, and our objective was to accurately identify and classify these cacti using the developed models. We train models for 8 epochs with a batch size of 32.

Code Implementation

During the process of constructing the architectures and models, we relied on renowned libraries from the fields of machine learning and deep learning. We have implemented the network architectures in Keras - a deep learning API written in Python, running on top of the machine learning platform for building and training neural networks - TensorFlow. Additionally, we utilized sklearn, a machine learning library that help us to use the dataset for training.

In our implementation, we utilize the Adam optimizer as an optimization algorithm commonly used for deep learning tasks and is well-suited for training neural networks. The configuration of the Adam optimizer in our code includes parameters such as learning rate (lr=0.0001), beta values (beta_1=0.9 and beta_2=0.999), and epsilon (epsilon=1e-08). Additionally, we compile the model using the compile function from Keras, enabling the specification of the optimizer, loss function, and evaluation metrics for training. In our code, the model is compiled with the specified settings, utilizing the Adam optimizer with the provided parameters, the binary cross-entropy loss function, and the accuracy metric to evaluate the model's performance.

Model Performance Evaluation

For each tested model, we assessed its accuracy on the validation set, examined the number of parameters the model learned, and measured the running time. These metrics were evaluated to measure the performance, model complexity, and computational efficiency of each tested model.

4. Results

Computational complexity

In each of the examined architectures, a clear correlation exists between the number of learned parameters and the *P* value, as shown in *Figure 4 - left graph*. As the value of P increases (indicating that a particular layer within the block receives input feature maps from more distant layers), the number of learned parameters also increases correspondingly. This behavior is due to the fact that when a layer receives input from additional preceding layers, it needs to learn new sets of weights and biases to process and combine these input feature maps effectively. Each connection between layers represents a set of parameters that the network adjusts during the training process to optimize its performance. learning a large number of parameters escalates the computational complexity, leading to extended running times, as depicted in *Figure 4 - Right graph*. This graph illustrates the trends in running time concerning the P value. As the P value increases, there is a corresponding increase in the running time.

Another aspect to consider is comparing architectures with different numbers of layers within a block (L) or different growth rate (k) values (*Figure 4 - Left graph*). When comparing architectures with an equal number of layers in each block (L), regardless of the value of the P, it is observed that higher growth rates (k) lead to an increased number of studied parameters. In addition, when comparing architectures with the same growth rate (k), it becomes clear that increasing the network depth (L) by incorporating additional layers in block results in a larger number of parameters that require learning.

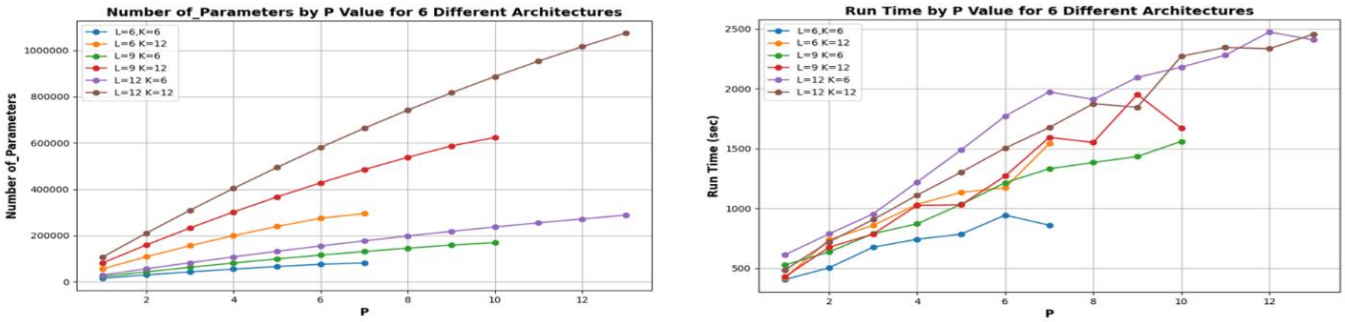

Figure 4 *- Computational complexity as a function of P for each architecture. (Left - number of parameters, Right - run time*

Accuracy

Figure 5 comprises 6 graphs, each representing a different architecture with different numbers of layers in the block (L) and different growth rates (K). In each graph, each trend line represents a distinct model initialized with a different P value, reflecting the number of previous connections, while maintaining the same configuration ({k, p}). In each graph, the accuracy values obtained by the models on the validation set at the end of each epoch are presented . The presented figure allows us to achieve valuable insights into the learning behavior of the models. It enables an examination of how different P values influence essential aspects such as the learning rate, convergence process, and achievable accuracies. In each of the examined architectures, when the hyperparameter P is set to one (each layer receives input just from the previous layer) the accuracy achieved on the validation set at the end of each epoch is noticeably lower compared to models initialized with larger P values (connections with more than one layer). The results indicate that there is a notable significance in receiving inputs from multiple previous layers compared to solely relying on the output of the last layer.

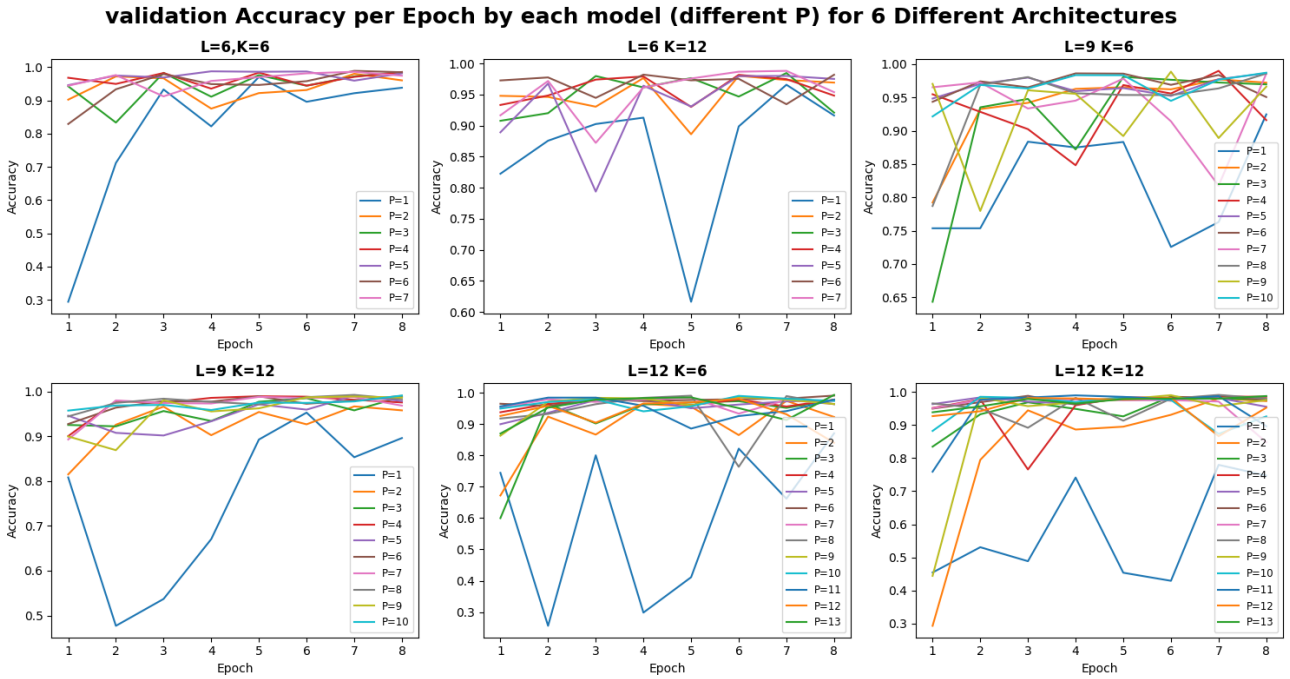

Figure5 *- Validation accuracy per epoch by each model (different P) for different examined Architectures.*

Figure 6 illustrates the Best Validation Accuracy achieved by each architecture, considering varying values of P. With analyzing each architecture individually, a notable enhancement in accuracy is observed as the P value increases from 1 to 2. However, the rate of improvement diminishes as P increases from 2 to 3 and beyond. These findings indicate that with an increase in the value of P, the rate of improvement in maximum accuracy diminishes. Specifically, augmenting the value of P leads to enhanced accuracy in the validation set until a certain point of convergence is reached, where the maximum accuracies of the models converge. Beyond this point, further increments in the value of P do not result in significant improvements in the maximum accuracy.

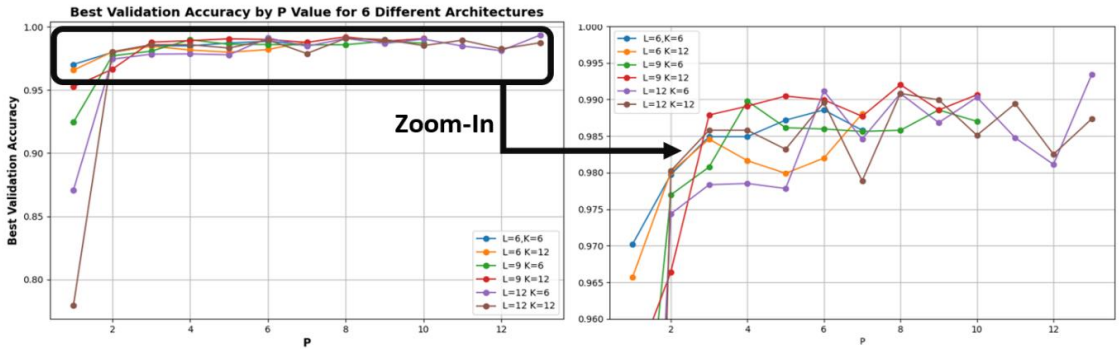

***Figure 6** - Best Validation Accuracy by P Value for 6 Different Architectures.*

Furthermore, a collective examination of all tested architectures reveals a noteworthy trend (*Figure 6*): as the value of parameter P increases, the differences between the maximum accuracy values attained by each architecture diminish. Notably, at P=1, there exists a substantial discrepancy in the maximum accuracy achieved by the tested architectures. However, as P increases, the distinctions between the architectures' maximum accuracies become less pronounced. This observation implies that higher values of P lead to a convergence of maximum accuracy across the different architectures, suggesting a certain level of robustness and consistency in model performance with respect to the parameter P.

learning Process

Figure 5 shows that during the learning process, when $P = 1$, there is significant volatility in the accuracy values achieved at the end of each epoch. Additionally, smaller P values such as $P = 2$ and $P = 3$ exhibit volatility compared to models with larger P values in most charts. To support the claim that smaller P values lead to higher volatility in the accuracy values achieved at the end of each epoch, let's examine *Figure 7*:

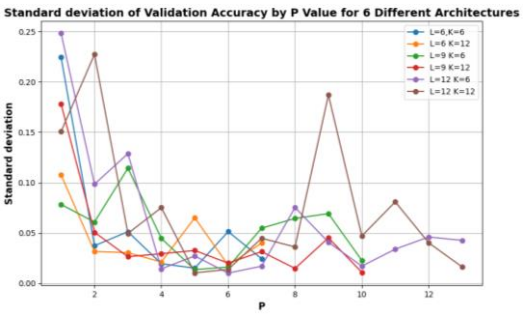

***Figure 7** - Standard deviation of Validation Accuracy by P Value for 6 Different Architectures.*

This chart illustrates the standard deviation of accuracy values obtained at the end of each epoch for each P value (X-axis) across our six different tested architectures (represented by different lines).

Figure 7 indeed displays a downward trend in the standard deviation, indicating that as P values increase, the standard deviation decreases. This implies that accuracy values achieved during the training process at the end of each epoch are more consistent in models initialized with larger P values than those initialized with smaller P values. From this observation, we can get valuable insights into the learning process of the models. Models with larger P values tend to achieve similar accuracies in most epochs, indicating quicker convergence and a more stable behavior compared to models with smaller P values.

5. Conclusions

The results discussed in the previous chapter provide crucial insights into the network's structural characteristics and its adaptability to technical and computational constraints. Increasing the number of connections ($P > 1$) positively impacts model accuracy. However, as the P value rises, the improvements become less significant, ultimately converging to a certain accuracy range. In specific requirements, connecting a subset of previous layers presents a viable alternative approach to a fully dense network. This approach yields satisfactory results while reducing computational demands and optimizing resource utilization.

Furthermore, it is feasible to opt for a model with a high number of connections (high P value), which enables the use of simpler architectures containing fewer layers in each block (L). Such models tend to attain comparable levels of accuracy to more complex models with a greater number of layers. These findings offer valuable insights for achieving high accuracy in resource-constrained environments, where processing power, memory, and runtime limitations are prevalent. By carefully selecting the architecture and optimizing layer connectivity, it is possible to improve accuracy while efficiently utilizing technical resources.

6. Bibliography

[1] Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely connected convolutional networks. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 4700-4708).

[2] G. Larsson, M. Maire, and G. Shakhnarovich. Fractalnet: Ultra-deep neural networks without residuals. arXiv preprint arXiv:1605.07648, 2016.

[3] B. M. Wilamowski and H. Yu. Neural network learning without backpropagation. IEEE Transactions on Neural Networks, 21(11):1793–1803, 2010

[4] Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. FeiFei. Imagenet: A large-scale hierarchical image database. In CVPR, 2009

[5] C.-Y. Lee, S. Xie, P. Gallagher, Z. Zhang, and Z. Tu. Deeplysupervised nets. In AISTATS, 2015

[6] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In CVPR, 2016.

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

[8] Dense Net Image Classification | Kaggle