DenseNet Models for Tiny ImageNet Classification

E4040.2020Fall.ERP1.report
Phan Nguyen pn2363, Ryan Hsu reh2166, Edouard Sanson es3770
Columbia University

Abstract

In this paper, we present how we implemented two different Densely Connected Convolutional Networks classification models on the Tiny ImageNet Dataset. We implemented the two models using Tensorflow and Keras, and also used several Regularizers, Optimizers and Callbacks functions during our training process, such as L2Kernel Regularizers, Adam Optimizers, Reduce LR On Plateau and Triangular2 Cyclical Learning Rate. Despite computational constraints, the 2 models achieved an accuracy of respectively 48% and 61% on the validation dataset.

1. Introduction

Inspiration for the original paper stemmed from the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). The original authors attempted to build a model with architecture derived from that of DenseNet, with several modifications to optimize performance under an environment with more constrained resources [2]. These modifications are based on using innovative learning rate algorithms, as well as a series of both direct and indirect image augmentations.

In our report, we attempt to reconstruct the two networks described in the original paper, and introduced several modifications of our own in order to optimize the learning process given the timeline and resources provided for the final project.

As described later in the *Methodology* section, our project will have two main objectives, namely replication of original learning algorithms wherever needed, and reconstruction of similar validation accuracy and loss. In this case, successfully achieving the first objective was a necessary prerequisite for attaining the second. As we were limited to the *Tensorflow* and *Keras* libraries for construction of the neural network, we will provide explanations of implementation and derivation details for algorithms that required from-scratch implementation.

2. Summary of the Original Paper

2.1 Methodology of the Original Paper

The original paper described the concept of Densely Connected Convolutional Networks and presents two classification models based on this alongside specific data augmentation and regularization techniques to improve robustness of these models.

Densely Connected Convolutional Networks corresponds to a specific structure of connected neural network that relies on feature aggregation, which is more performant than feature summation and avoids vanishing gradient issues during training process.

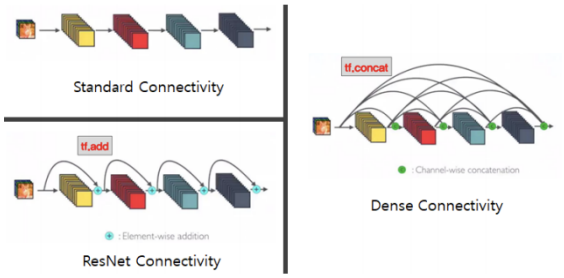


Figure 1. Standard vs ResNet vs DenseNet layers

Detailed structure of the 2 networks are presented in section 4.1.

During training, data augmentation is performed using both direct and indirect approaches. Direct image augmentation consists of performing scaling, rotation, flipping, translation, etc. operations on the image to artificially increase the size of the dataset. Indirect image augmentation corresponds to varying image size during training.

Finally, batch normalization and regularization techniques are used for both models during the training process to improve accuracy. In the first model, A call back function, ReduceLROnPlateau, reduces the learning of the Adam optimizer in case validation loss stagnates for 5 epochs. In the second model, the Cyclical Learning Rate method is used to adapt the learning rate of the Adam optimizer.

Network 1 was trained with 17.9 Million parameters for 235 epochs with an adaptatie batch size, depending on the images size.Network 2 was trained with 11.8 Million parameters for 108 epochs with a batch size of 128 images.

2.2 Key Results of the Original Paper

Network 1 reached overall a training accuracy of 67% and an accuracy of 59.5% on the validation dataset.
The accuracy on the validation dataset increases as follow during the training process:

- 48% after 45 epochs (indirect data augment.)
- 48-49% after 85 epochs (indirect data augment.)
- 59.5% after 235 epochs (direct data augment.)

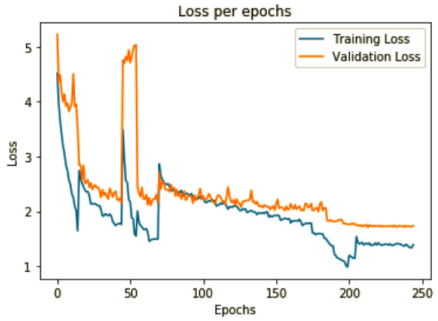


Figure 2. Loss curve of Network 1

Network 2 performed better than network 1, while being trained on less epochs and reached over a training accuracy of 68.11% and an accuracy of 62.73% on the validation dataset.

Figure 3. Loss curve of Network 2

As explained in the paper, the 3 main reasons that could explain these classification errors are:

1) Low resolution of the images: some misclassified images have very low resolution and are hard to differentiate even for humans
2) Misunderstanding of the primary entity in the image: the network captured the wrong entity and classified the image accordingly
3) Confusion by similar items: some classes are nearly overlapping, thus these classes tend to performed worst than others

Class Name	Val. Accuracy
Plunger	25%
Umbrella	27.27%
Water Jug	30%
Bucket	30.43
Wooden Spoon	31.03%
Bannister	31.71%

Figure 4. Top worst classified classes

3. Methodology (of the Students' Project)

3.1. Objectives and Technical Challenges

The first primary objective of our project was to provide a best-efforts implementation of any algorithms or techniques used in the original reconstruction of the two networks that are not available within the base *Tensorflow 2.2* and *Keras* libraries. We identified several methods and algorithms that required re-implementation, either from scratch, or using existing Tensorflow scaffolding:

- *SpaceToDepth*
- *ReduceLROnPlateau*
- *Triangular2 Cyclical Learning Rate*

The first method described above is required for reconstruction of Network 1 of the original paper, while the latter two are implemented as callback functions passed into the *model.fit* method called upon training. We provide further details on each of these methods and algorithms in the *Implementation* section.

The next primary objective of our project was to achieve similar model performance as that of the original two networks presented. In the original paper, performance is measured via validation loss, as well as validation accuracy. As described in the original paper, learning rate modification methods and image augmentation were prerequisites required to achieve the performance. Hence, it was imperative that we achieve our first objective before proceeding to work on the second. While we strived to obtain a faithful reconstruction of the original author's results, we recognize that there are several technical limitations that would prevent us from achieving this goal.

The first obstacle that we faced was that the original authors had trained the different architectures for a combined total of more than 500 epochs. Using our Google Colab and GCP platforms, we found that training time for one epoch ranged anywhere from 7 to 15 minutes. This is equivalent to between 58 to 125 hours of training time. As we did not have enough remaining GCP credits, and google colab enforced a strict notebook runtime, we were forced to train the models for fewer total hours. Furthermore, due to periodic issues with GCP and Colab, we had to adopt a much more aggressive save-and-refresh approach as compared to the original authors. For network 1, we saved a copy of the model every 15 epochs, while in network 2, we saved a copy after every 6-12 epochs.

The second obstacle faced was that the original paper did not provide implementation details for all methods and algorithms. As an example, specific parameters used for each image augmentation mentioned in the original paper were not provided. In these cases, we resorted to personal heuristics to decide on parameter values, or used default method values (as in the case of *ReduceLROnPlateau*). We

describe in detail what manual decisions were made in the *Implementation* section. Due to time constraints, we did not have time to optimize for the best parameters (i.e: perform grid-search) in each ambiguous case. We describe differences in observed results in comparison to the original paper in the *Results* section and hypothesize on how differences in chosen parameter values may have impacted faithfulness of the reconstruction.

3.2 Problem Formulation and Design Description

Our project implementation follows the flow-chart below.

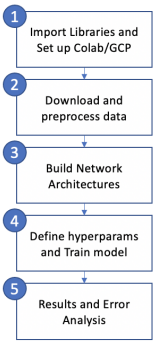


Figure 5. Project implementation flow-chart

The first step of our implementation requires setting up the coding environment. We set up Jupyter notebook environments for both GCP and Colab as some of us faced issues with running the project on GCP. Furthermore, Colab also provided a contiguous experience for debugging our models and collaboration. Instructions on how to run provided Jupyter notebooks and code are provided in the attached *README* file.

The second step of the implementation consisted of data retrieval and preprocessing. Both networks utilized the same dataset as per the original paper. This is the *Tiny Imagenet* dataset, available for download here[3]. Both train and validation data folders were conveniently organized according to folder structures recognizable by *Keras' 'flow'* methods [4]. As such, we were able to rely on the *flow_from_directory* and *flow_from_dataframe* methods to create image data generators for both training and validation data [5]. In the case of Network 1, the use of the *flow* methods allowed us to specify an *image_size* parameter, which allowed us to perform the image resizing directly as described in the *Implementation* section. For Network 2, the original paper describes the application of a 'random sequence of 11 transformations to half of the dataset'. The *ImageDataGenerator* class does not provide functionality for application of these 11 augmentations in a probabilistic manner by default. As such, we needed to apply these augmentations via the preprocessing function. The *Implementation* section describes the construction of an Image Augmentation function that probabilistically applies the 11 augmentations.

The third step in our process was to build out the two network architectures. The DenseNet-derived architectures require concatenation of previous layer outputs at specific points in the network. Thus, we used the *Model* class to implement the architectures as the *Sequential* class would not allow us to access intermediate layer results.

For the training process, we trained our Network 1 for 70 epochs and Network 2 for 54 epochs. In the process of training the model, we found that Google Colab and GCP both provide relatively unstable environments for continuous training. As such, similar to the original paper, we periodically saved checkpoints of our model. This allowed us to reload the model and continue training should an issue arise and not lose progress. As we experienced periodic timeouts while training both networks, we chose to save a checkpoint of our models every 15 epochs for Network 1 and 6-12 epochs for Network 2. Saving every 12 epochs for network 2 allowed us to also re-initialize our *Triangular2 Cyclical Learn Rate* function with parameters provided below by the original paper:

Min LR	**Max LR**	**StepSize**	**Start Epoch**	**End Epoch**
1e-04	6e-04	6	0	12
1e-05	6e-05	6	12	24
1e-05	6e-05	4	24	36
1e-06	6e-06	2	36	42
1e-05	6e-05	2	42	48
1e-07	6e-07	2	48	54

Figure 6. Parameters for *Triangular2 Cyclical Learn Rate*

Lastly, we plotted graphs of training/validation loss over time as well as training/validation accuracy over time to evaluate our model performance and provide comparisons to the original implementations. We also explored several cases of misclassified examples to see if we observed similar cases of classification errors as the original paper.

4. Implementation

4.1. Deep Learning Network

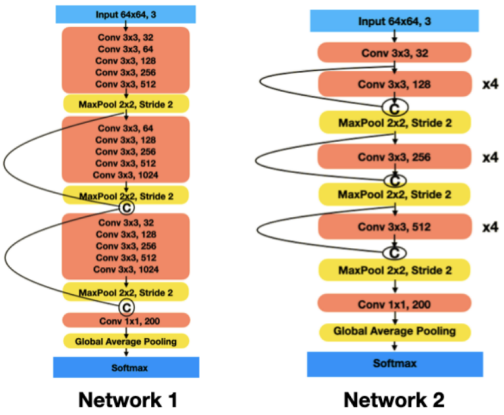


Figure 7. Model architectures

We implemented two different DenseNet architectures as in the original paper to try and achieve at least 40% accuracy on the Tiny ImageNet dataset. In Network 1, there are three blocks of convolutional layers followed by max pooling with pool size of (2,2). Within each of the first three blocks, there are five convolutional layers with filter sizes always doubling from the previous layer to the next. The first block starts with a layer of filter size 32 and ends with a layer of filter size of 512. The next two blocks start with a filter size of 64 and end with a layer of filter size 1024.

The fourth and final block contains one convolutional layer followed by global average pooling, before being passed to the softmax activation function to arrive at the output.

For Network 1 we used the popular Adam optimizer, which can be thought of as a combination of the AdaGrad and RMSProp optimizers. Since we are working with a multi-class classification problem, the model was trained using the categorical cross entropy loss function.

Batch normalization re-parametrizes the model to restore the zero mean and unit variance properties in the previous layer, meaning that the previous layer will tend to remain a unit Gaussian despite other updates to the lower layers. Here we used batch normalization after each convolutional layer. The output from each batch normalization step is then passed to a ReLU unit.

For Network 2, our implementation divides the architecture into 5 blocks each containing one or more convolutional layers. Similar to the original paper, we passed the outputs of all Conv2D layers through a BatchNormalization layer, followed by an Activation(ReLU) layer. Variance scaling, is a popular weight initialization strategy utilized in the original paper. This method allows the shape dimensions of the weight tensors to impact how weights are initialized. [6]. In *Tensorflow 2.2*, Variance Scaling can be enabled by passing *'VarianceScaling'* into the *kernel_initializer* field of a Conv2D layer initialization [7]. The original paper also uses an L2 kernel regularizer (lambda = 2e-4) to initialize all Conv2D layers. Similarly, we applied these two settings when initializing all of our Conv2D layers. The code snippet below shows sample parameters used to initialize a Conv2D layer.

```
block1 = Conv2D(filters=32, kernel_size=(3,3),
                strides=(1,1), padding='same',
                activation=None, kernel_initializer="VarianceScaling",
                kernel_regularizer=keras.regularizers.l2(2e-4))(inputs)
```

Figure 8. Code snippet for Conv2D initialization

The first block consists of a sole Conv2D layer with 32 filters, zero-padding enabled, kernel size of (3,3) as well as strides of (1,1).

The second block consists of 4 continuous Conv2D layers, each with 128 filters, zero-padding enabled, kernel size of (3,3) and strides of (1,1). Output from the first block is then retrieved that of block 2. At the end of this block, we applied a MaxPool2D layer, with pool size of (2,2) and stride of 2. As concatenation of previous blocks occurs prior to MaxPool2D layer application, we did not need to apply *SpaceToDepth* to any of our layers.

Implementation of the third and fourth block was largely similar to that of the second block with respect to layer structure and initialization parameters. However, in the third block, 256 filters were used for each Conv2D layer and in the fourth block, 512 filters. In both layers, concatenation of output from the previous block prior to MaxPool2D was also carried out.

The final layer consists firstly of a single Conv2D layer, with 200 filters, zero-padding enabled, kernel size (1,1), and strides of (1,1). Output from this layer was passed into a GlobalAveragePooling2D layer, and finally passed through a Softmax activation layer.

4.2. Software Design

Below we describe the implementation of the various algorithms needed for reconstruction of the networks described in the original paper. Link to the code and jupyter notebooks for the project may be obtained here.

Data augmentation was implemented using two different ways:

- Indirect augmentation used on network 1, which consists on training the model with different image size
- Direct augmentation, using ImageAug library[8]. This library offers a wide range of data

augmentations, among which we implemented: Horizontal Flip, Vertical Flip, Scale, Translate, Rotate, Gaussian Blur, Crop and Pad, Shear, Coarse Dropout, Multiply, Contrast Normalisation. The Sometimes() function from ImageAug library was used to randomly select and perform data augmentation on half of the Dataset.

```
def Image_augmentation(input_img):
  # Randomly apply 11 transformations to half of the dataset
  seq = iaa.Sometimes(0.5,iaa.Sequential([
  iaa.Fliplr(0.5),  # Horizontal flip
  iaa.Flipud(0.2), # Vertical flip
  iaa.GaussianBlur(sigma=(0, 0.5)),  # GaussianBlur
  iaa.CropAndPad(percent=(-0.25, 0.25), pad_mode=ia.ALL,pad_cval=(0, 255)),  # Crop images
  iaa.Affine( scale={"x": (0.8, 1.2), "y": (0.8, 1.2)},
              translate_percent={"x": (-0.15, 0.15),
                                 "y": (-0.15, 0.15)},rotate=(-30, 30), shear=(-15, 15)
                                 ,order=[0, 1],cval=(0, 255), mode=ia.ALL),  # Affine transformations: scale, translate, rotate,
  iaa.CoarseDropout((0.03, 0.15), size_percent=(0.02, 0.05),per_channel=0.2),  # Corse dropout
  iaa.Multiply((0.9, 1.1), per_channel=0.25),    # Multiply
  iaa.ContrastNormalization((0.75, 1.5), per_channel=0.5)], random_order = True))  #Contrast Normalization

  augm_img = seq.augment_image(input_img)
  return augm_img
```

Figure 9. Code snippet for ImageAugmentation

ReduceLROnPleateau allows for a dynamic learning rate and monitors a validation metric. If the metric, in our case validation accuracy, does not improve for a number of "patience" epochs, then the callback function will decrease the learning rate. The number of patience epochs and factor by which to decrease the learning rate were set to their default values, 10 and 0.1 respectively.

SpaceToDepth was a method used in the original paper to reshape the outputs of a Conv2D layer such that the *Concatenation* layer may be applied. As we can see from the architecture of Network 1, un-modified concatenation of output from the first block to that of the second is not possible due to the differences in output dimensions (BatchSize, 32, 32, 3) and (BatchSize, 16, 16, 3). The original paper circumvents this issue by transforming the original output into an array of shape (BatchSize, 16, 16, 12), effectively "moving" values from the *height* and *width* dimensions of an image into the *channels* dimension. Our implementation of the *SpaceToDepth* function that takes in an input array of size (BatchSize, N, N, 3) and returns an array of size (BatchSize, N/2, N/2, 12). We achieve this via the *tf.nn.space_to_depth* method provided by *Tensorflow* backend [9]. We called this function on an input array passing in *block_size* value of 2, indicating that we want to 'split' the height and width dimensions by 2.

Cyclical Learning Rate algorithms are a class of algorithms that allow for dynamic learning rates similar to *ReduceLROnPlateau*. These algorithms typically define a range of learning rates (minimum and maximum) that the optimizer can take and calculates a per-batch learning rate that cyclically increases and decreases based on a defined pattern [10]. The original paper used the *Triangular2* pattern to calculate per-batch learning rates. In this pattern, the initial learning rate starts off at the predefined minimum learning rate. This rate increases linearly per batch until a set number of batches have passed. This turning point is defined in terms of the number of epochs and is called *StepSize*. For example, a StepSize of 4 means that the learning rate will increase from the minimum defined rate to the maximum defined rate within 4 epochs. At the end of a full cycle (2**StepSize*), the difference between the maximum and minimum rate is divided by half, effectively creating smaller changes in learning rate.

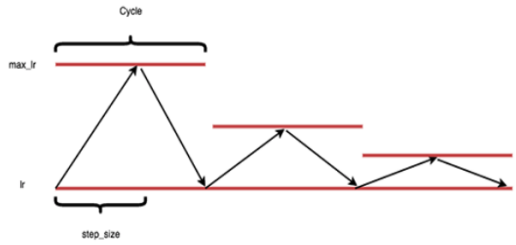


Figure 10. Visualization of *Triangular2* Cyclical Learning Rate [11]

As *Tensorflow 2.2* does not have a built-in implementation of *Triangular2 Cyclical Learning Rate,* we were required to implement the algorithm from scratch. Similar to *ReduceLROnPlateau*, we wanted to directly modify learning rates being used via the *callback* method that *model.fit* provides. Thus, we implemented a custom callback function to be passed in at training time. *Tensorflow* documentation provided us with a good resource to go about customizing our own callback function[12].

In the created custom callback class, *on_epoch_begin* and *on_batch_begin* provided access points during the training process where we could fetch and modify optimizer values. Within *on_epoch_begin*, at the start of each training cycle, we manually set the learning rate to the specified minimum learning rate. As our training processes required loading in models for later cycles, this method is a way to reset the initial learning rate without having to recompile the whole model.

Within *on_batch_begin*, we linearly increased the learning rate at the start of each batch as per the *Triangular2* algorithm. Here, we also checked if we have reached the *StepSize* threshold, whereby we will switch to decrement the learning rate (and vice-versa). A switch from decreasing learning rate to increasing learning rate signifies that we have completed a cycle. This is where we will reduce the difference between minimum, maximum learning rate by half as required by *Triangular2.* The figure below shows a sample learning rate turning point where learning rate switches from increasing to decreasing.

```
lLr for batch  775  is:  0.0005993324448354542
lLr for batch  776  is:  0.0005994391394779086
lLr for batch  777  is:  0.000599545834120363
lLr for batch  778  is:  0.0005996525287628174
lLr for batch  779  is:  0.0005997592234052718
lLr for batch  780  is:  0.0005998659180477262
': 0.4579 - val_loss: 3.5900 - val_accuracy: 0.31

56Lr for batch  1  is:  0.0005998659180477262
20Lr for batch  2  is:  0.0005997592234052718
22Lr for batch  3  is:  0.0005996525287628174
31Lr for batch  4  is:  0.000599545834120363
```

Figure 11. Triangular2 learning rates by batch

5. Results

5.1. Project Results

Model results for Network 1:

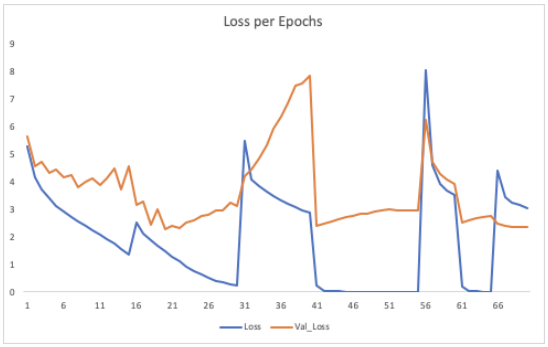


Figure 12. Loss curve of Network 1

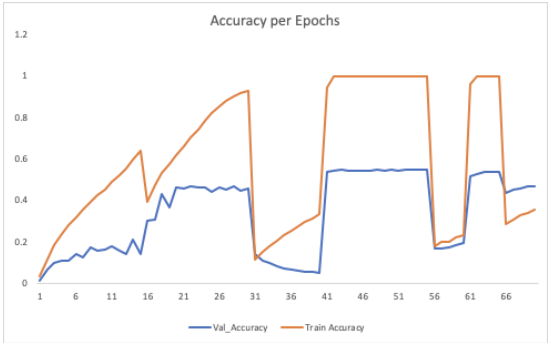


Figure 13. Accuracy curve of Network 1

Model results for Network 2:

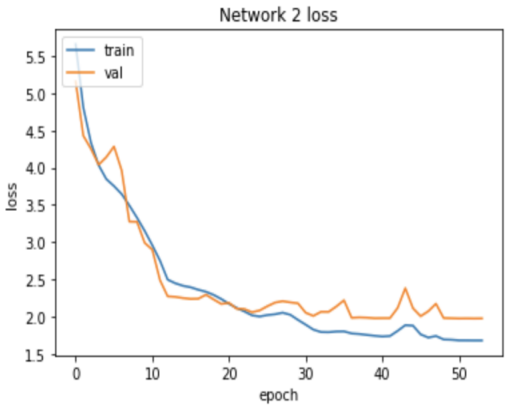
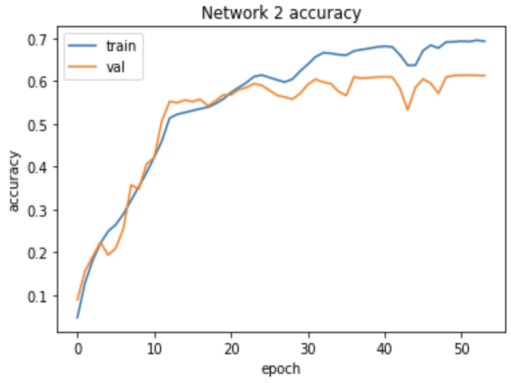


Figure 14. Training and Validation Loss for Network 2 Model




```
Maximum validation accuracy achieved was:  0.6138
```

Figure 15. Training and Validation Accuracy for Network 2 Model

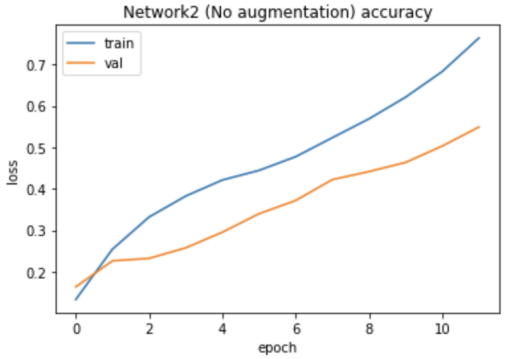


Figure 16. Training and Validation Accuracy for Network 2 Model without Image Augmentations

5.2. Comparison of the Results Between the Original Paper and Students' Project

For Network 1, there were 17.9 million parameters as in the original paper, however due to time constraints we trained the model for 70 epochs instead of the 235 epochs that the paper's authors used. The first 55 epochs were trained on the dataset without image augmentation, while the next 15 epochs were trained using augmentation. The first 15 epochs were trained on 32x32 resolution images and we observed an accuracy of 20%. As in the paper, we observed signs of overfitting as the training accuracy quickly increased to over 50%. Hence we then trained using 64x64 resolution images and trained for another 15 epochs which increased the validation accuracy to over 40%. It is interesting to note that overfitting at this step seemed a bit more severe in our model than the author's, as training accuracy reached 92% which signalled a relatively large gap between training and validation accuracy.

Afterwards, we trained the model for 10 epochs on 16x16 images in order to allow the model to learn to classify low resolution images. Here we noticed the same

spike in training and validation loss that the authors experienced while switching to the low resolution image size. Next, we reverted to using 64x64 resolution images and trained for another 15 epochs, which resulted in validation accuracy of about 55%. It was at this point that we implemented image augmentation. We noted that the validation accuracy we observed was relatively consistent with the author's at this point of the model training, however we observed our model seemed to suffer from overfitting more so than the author's, with a gap between training accuracy and validation accuracy of over 40% while the author did not see a gap larger than 20%.

After image augmentation, we saw the overfitting issue was mitigated with a gap between training and validation accuracy of under 20% again. Validation accuracy levelled out around 48%, which was acceptable to us and we did not train further.

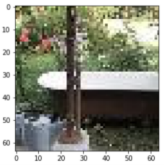
For Network 2, our model also had 11.8 million parameters as per the original paper, with batch size also set at 128. However, due to time constraints, we trained the model using half the number of epochs per cycle as the original paper. Our results show both similarities and differences with results from the original network.

While unable to exactly match the validation accuracy of 62.73% presented in the original paper, we were able to obtain a maximum validation accuracy of 61.38%. Furthermore, similar to the original paper, we observed a stark decrease in loss as well as increase in accuracy within the first 12 epochs of training.

Regarding differences, we noticed that while loss in the original implementation hovered around between 4.0 and 4.5, our model's loss continues to decrease over time. We hypothesize that this could potentially be due to image-augmentation related parameters used as the original paper did not specify exact manual augmentation parameters used.

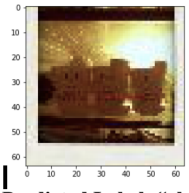
We present two illuminating examples of the errors made by our final model, which is shown in figure 17.

True Label: "bathtub, bathing tub, bath, tub"



Predicted Label: "Lawnmower, mower"

True Label: "refrigerator, icebox"



Predicted Label: "chest"

Figure 17. Examples of Misclassified Images

It seems that these errors can be explained by similarities to other images, for example the model likely labelled the top image as a lawn mower because it had learned that lawn mowers were typically surrounded by grass and greenery, while bathtubs were rarely surrounded by a green background. The low resolution of the images also played a large part in the misclassifications. This may have been exacerbated by the fact that we trained using 16x16 resolution images for 10 epochs in the mid-to-late stages of our training, but only trained for 25 epochs using higher resolution afterwards. We expect that additional training on 64x64 resolution images would bring the performance of our model to approach the validation accuracy achieved by the authors.

5.3. Discussion of Insights Gained

Our results showed that while we were not able to completely reconstruct the results produced in the original paper, we were able to reproduce several distinct trends within both network architectures.

After training the model for Network 1, It was clear that our model suffered from overfitting to a greater degree than in the original paper. Thus we note that image augmentation for this dataset is important in preventing model overfitting for the DenseNet model. Training images without occasionally changing the resolution size or augmenting the train set causes the gap between training accuracy and validation accuracy to quickly widen. In our model, we used default values for the ReduceLROnPleateau callback. We also set the steps per epoch to be equal to the number of samples the training set divided by the batch size, as opposed to the original paper which set the steps per epoch to be 200 throughout the entirety of training. We expect that additional hyperparameter tuning, alongside training the model for more epochs, would have allowed us to achieve validation accuracy closer to the authors.

With regards to Network 2, it is entirely possible that we provided more lenient augmentations that more closely resembled the original images. This allowed the model to better learn the different representations. While this may have allowed us to achieve an almost equivalent validation accuracy within fewer epochs, this may have limited the effectiveness of Cyclical Learning. Based on

the results, validation loss started to plateau at around epoch 30. Application of a higher minimum/maximum learning rate at epoch 42 did not seem to 'break' the model out of the loss plateau as in the original paper. Validation accuracy also did not change and at certain epochs even decreased.

Furthermore, we see that the original paper did not directly quantify the benefits achieved from image augmentation with respect to Network 2. In our simple tests, we were able to achieve ~55% validation accuracy within 12 epochs (figure 16). Given more time, we are curious to see if further training will yield similar loss and accuracies as a model that undergoes augmentation.

6. Conclusion

We implemented two Densely Connected Convolutional Networks models on the Tiny ImageNet Dataset and added Regularizers, Optimizers and Callbacks functions to improve our models performances. Despite relatively low computational resources which prevented us from tuning the hyperparameters and training the model using as many as many epochs as performed in the original paper, we achieved a validation accuracy of respespectively 48% and 61% for model 1 and 2 respectively. With this project, we could learn how to build a deep learning based model from A to Z, using the latest training techniques and leveraging some concepts learnt this semester such as adaptive learning rate or data augmentation. The main area of improvement remains the tuning of hyperparameters, especially for data augmentation, to better reduce overfitting. Callbacks functions can also be improved to get a more robust training process.

7. Acknowledgements

We thank the teaching assistants and Professor Kostic of ECBM4040 at Columbia University for providing guidance related to the implementation of DenseNets in the Google Cloud and Google Colab environments.

8. References

[1] Link to github repo
[2] H. Abai, N. Rajmalwar, "DenseNet Models for Tiny ImageNet Classification", https://arxiv.org/ftp/arxiv/papers/1904/1904.10429.pdf
[3] ImageNet, "Tiny ImageNet Dataset", http://image-net.org/download-images
[4] J. Vijayabhaskar, "Tutorial on Keras flow_from_dataframe", https://vijayabhaskar96.medium.com/tutorial-on-keras-flow-from-dataframe-1fd4493d237c
[5] I. Michelin, "Data Generators and how to use them", https://towardsdatascience.com/keras-data-generators-and-how-to-use-them-b69129ed779c
[6] X. Glorot, Y. Bengio., "Delving deep into rectifiers:surpassing human-level performance on imagenet classification.", *IEEE International Conference on Computer Vision (ICCV)*, 2015.
[7] Keras, "Layer Weight Initializers", https://keras.io/api/layers/initializers/
[8] aleju/imgaug: Image augmentation for machine learning experiments.
[9] Tensorflow, "tf.nn.space_to_depth", https://www.tensorflow.org/api_docs/python/tf/nn/space_to_depth
[10] L. N. Smith, "Cyclical Learning Rates for Training Neural Networks" , *arXiv:1506.01186v6*, 2017
[11] A. Rosebrock, "Cyclical Learning Rates with Keras and Deep Learning", https://www.pyimagesearch.com/2019/07/29/cyclical-learning-rates-with-keras-and-deep-learning/
[12] Tensorflow, "Writing your own Callbacks", https://www.tensorflow.org/guide/keras/custom_callback

9. Appendix

8.1 Individual Student Contributions in Fractions

	pn2363	reh2166	es3770
Last Name	Nguyen	Hsu	Sanson
Fraction of (useful) total contribution	2/5	3/10	3/10
What I did 1	Network 2 Architecture and Notebook	Network 1 Notebook and Training	Network 1 Data Augmentation implementation
What I did 2	SpaceToDepth, Triangular2 CLR implementations	LearnLROnPleateau implementation	Network 2 Data Augmentation implementation
What I did 3	Report (Intro, Section 3, Section 4 (Network 2), Section 5 (Network 2)	Report (Section 4 (Network 1), Section 5, (Network 1)	Report (Extract, Section 2, Section 4 (Data Augmentation), Conclusion