Densely Connected Convolutional Networks

E4040.2024Fall.ANLM.report.jl6865.amn2225.lm3879
James Liu jl6865, Anirudh Natarajan amn2225, Lindsey (Lan Xin) Ma lm3879
Columbia University

Abstract

DenseNet, introduced by Huang et al., is a densely connected convolutional network architecture that emphasizes feature reuse to improve gradient flow, parameter efficiency, and overall performance. In this project, we reproduced the DenseNet-BC model using TensorFlow and Python, focusing on its application to CIFAR-10 and CIFAR-100 datasets. We systematically tested eight configurations of the architecture, with and without bottleneck layers (B) and compression (C), to evaluate their impact on model accuracy and efficiency. Configurations incorporating both bottleneck and compression achieved the best results, particularly on CIFAR-100, aligning with key insights from the original paper. However, due to computational constraints and reduced training epochs, our error rates were higher than those reported in the original study. Despite these challenges, our work highlights DenseNet's strengths in parameter efficiency and its dependence on architectural choices, providing a foundation for further optimization and scalability improvements.

1. Introduction

Deep learning has emerged as a transformative technology across diverse domains, with convolutional neural networks (CNNs) at the forefront of advancements in computer vision. While deeper neural networks generally improve performance, they often suffer from vanishing gradients, excessive parameter counts, and inefficient use of model capacity. To address these challenges, **DenseNet (Dense Convolutional Networks)** introduces a novel architecture that emphasizes feature reuse through dense connectivity. Proposed by Huang et al. in their seminal paper *"Densely Connected Convolutional Networks"* (2017), DenseNet connects each layer to every other layer in a feedforward manner, ensuring maximum information flow between layers. This unique design significantly reduces the number of parameters while enhancing feature propagation, which is crucial for improved gradient flow during training.

The primary goal of this project is to reproduce the results presented in the DenseNet paper using TensorFlow and Python. This includes reconstructing the model architecture, training the network on benchmark datasets such as CIFAR-100, and validating its performance against the metrics reported in the original work. We aim to closely replicate DenseNet's ability to achieve high accuracy with fewer parameters and conduct a detailed comparison of the reproduced results with those in the paper.

The technical challenges of this project lie in accurately implementing the densely connected architecture and ensuring consistency with the hyperparameters and training procedures described in the paper. Certain details, such as weight initialization, learning rate schedules, or data augmentation strategies, may be under-specified in the original text, requiring iterative experimentation and tuning. Additionally, reproducing DenseNet's performance metrics on high-dimensional datasets requires significant computational resources, as DenseNet's dense connections increase memory overhead and computational complexity.

To address these challenges, we will employ a structured approach:

1. **Model Reimplementation**: Building the DenseNet architecture with configurable bottleneck and compression options, following the principles outlined in the original paper.
2. **Training and Evaluation**: Testing the 8 combinations of DenseNet configurations (with and without bottleneck and compression) on CIFAR-10 and CIFAR-100, closely following the paper's training procedures.
3. **Performance Comparison**: Comparing the reproduced results with those reported in the original paper and analyzing the impact of different configurations on accuracy and parameter efficiency.
4. **Error Analysis and Tuning**: Identifying any discrepancies between our results and the paper's findings, analyzing the causes, and suggesting possible improvements.

This project not only demonstrates DenseNet's architectural innovations but also serves as an opportunity to critically evaluate its practical implementation. By reproducing the paper's results and reflecting on the challenges encountered, we aim to contribute to a deeper understanding of how densely connected networks influence the field of deep learning.

2. Summary of the Original Paper

2.1 Methodology of the Original Paper

Unlike traditional architectures, where each layer passes its output to only the next layer, DenseNet connects each layer to every subsequent layer in a feed-forward manner. This design promotes feature reuse, efficient gradient flow, and parameter efficiency, addressing common challenges like vanishing gradients and redundancy in deep neural networks. Each layer in a DenseNet receives the concatenated outputs of all previous layers as input, ensuring that earlier features are preserved throughout the network, while subsequent layers focus on higher-level representations.

DenseNet consists of key components, including *Dense Blocks*, which encapsulate the dense connectivity pattern, and *Transition Layers*, which reduce the spatial and channel dimensions to manage model complexity. The growth rate (k), defined as the number of feature maps added per layer, controls the compactness of the network. DenseNet also employs bottleneck layers, using 1x1 convolutions to enhance computational efficiency, and a compression factor in transition layers to further reduce parameters while maintaining performance. The network concludes with a global average pooling layer followed by a fully connected classification layer.

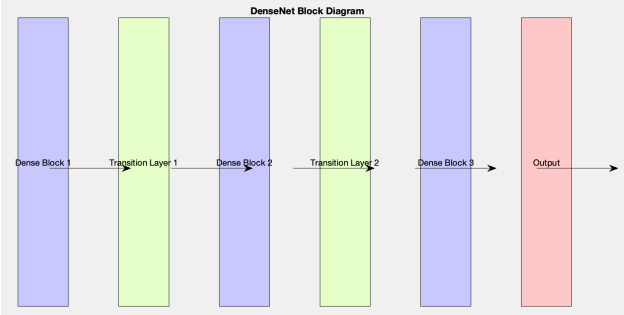


Figure 1. *This is a diagram illustrating the structural workflow of DenseNet's architecture, highlighting the dense blocks and transition layers. It effectively visualizes how layers are densely connected, promoting feature reuse and improving gradient flow.*

2.2 Key Results of the Original Paper

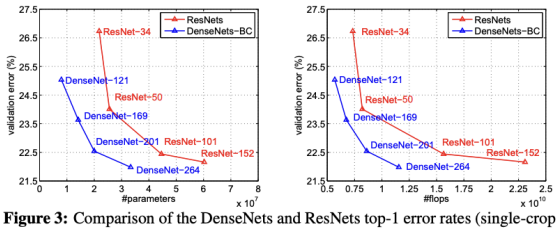


Figure 3: Comparison of the DenseNets and ResNets top-1 error rates (single-crop testing) on the ImageNet validation dataset as a function of learned parameters (*left*) and FLOPs during test-time (*right*).

Model	top-1	top-5
DenseNet-121	25.02 / 23.61	7.71 / 6.66
DenseNet-169	23.80 / 22.08	6.85 / 5.92
DenseNet-201	22.58 / 21.46	6.34 / 5.54
DenseNet-264	22.15 / 20.80	6.12 / 5.29

Table 3: The top-1 and top-5 error rates on the ImageNet validation set, with single-crop / 10-crop testing.

Figure 2*. Sets of Accuracy Graphs*
This is a collection of graphs depicting the relationship between parameter scale and validation error, along with a comparison of top-1 and top-5 error rates across models with varying depths.

The architecture was rigorously evaluated on multiple benchmark datasets, including CIFAR-10, CIFAR-100, SVHN, and ImageNet, demonstrating its effectiveness and efficiency. On CIFAR-10, DenseNet-BC with a depth of 100 and growth rate of 12 achieved a test error of 4.51%, outperforming ResNet-110, which had an error rate of 6.61%, while utilizing significantly fewer parameters (1.0M compared to ResNet's 1.7M). On ImageNet, DenseNet achieved a Top-1 Error of 22.27% and a Top-5 Error of 6.10% with a depth of 264 and growth rate of 32, showcasing competitive performance on large-scale datasets.

Method	Depth	Params	C10	C10+	C100	C100+	SVHN
Network in Network [22]	-	-	10.41	8.81	35.68	-	2.35
All-CNN [32]	-	-	9.08	7.25	-	33.71	-
Deeply Supervised Net [20]	-	-	9.69	7.97	-	34.57	1.92
Highway Network [34]	-	-	-	7.72	-	32.39	-
FractalNet [17]	21	38.6M	10.18	5.22	35.34	23.30	2.01
with Dropout/Drop-path	21	38.6M	7.33	4.60	28.20	23.73	1.87
ResNet [11]	110	1.7M	-	6.61	-	-	-
ResNet (reported by [13])	110	1.7M	13.63	6.41	44.74	27.22	2.01
ResNet with Stochastic Depth [13]	110	1.7M	11.66	5.23	37.80	24.58	1.75
	1202	10.2M	-	4.91	-	-	-
Wide ResNet [42]	16	11.0M	-	4.81	-	22.07	-
	28	36.5M	-	4.17	-	20.50	-
with Dropout	16	2.7M	-	-	-	-	1.64
ResNet (pre-activation) [12]	164	1.7M	11.26*	5.46	35.58*	24.33	-
	1001	10.2M	10.56*	4.62	33.47*	22.71	-
DenseNet ($k = 12$)	40	1.0M	**7.00**	5.24	**27.55**	24.42	1.79
DenseNet ($k = 12$)	100	7.0M	**5.77**	**4.10**	**23.79**	**20.20**	1.67
DenseNet ($k = 24$)	100	27.2M	**5.83**	**3.74**	**23.42**	**19.25**	**1.59**
DenseNet-BC ($k = 12$)	100	0.8M	**5.92**	4.51	**24.15**	22.27	1.76
DenseNet-BC ($k = 24$)	250	15.3M	**5.19**	**3.62**	**19.64**	**17.60**	1.74
DenseNet-BC ($k = 40$)	190	25.6M	-	**3.46**	-	**17.18**	-

Table 2: Error rates (%) on CIFAR and SVHN datasets. k denotes network's growth rate. Results that surpass all competing methods are **bold** and the overall best results are **blue**. "+" indicates standard data augmentation (translation and/or mirroring). $*$ indicates results run by ourselves. All the results of DenseNets without data augmentation (C10, C100, SVHN) are obtained using Dropout. DenseNets achieve lower error rates while using fewer parameters than ResNet. Without data augmentation, DenseNet performs better by a large margin.

***Figure 3**. Original paper model performance*
This is a performance table with CIFAR + SVHN datasets.

DenseNet's parameter efficiency is a key strength, offering superior accuracy with fewer parameters than competing architectures like ResNet and Wide ResNet. The network's densely connected design improves gradient flow and reduces overfitting, making it particularly effective for deep networks. Additionally, the efficient reuse of features results in compact yet expressive models. These characteristics have established DenseNet as a benchmark in deep learning, inspiring further research in densely connected architectures and efficient model design.

3. Methodology (of the Students' Project)

Our goal was to reproduce the results of DenseNet on the CIFAR-10 and CIFAR-100 datasets and evaluate the impact of bottleneck layers and compression on performance metrics. We implemented the architecture in Python using TensorFlow and conducted experiments on Google Cloud Platform (GCP) with an NVIDIA T4 GPU.

Approach

Implementation Details

The implementation comprises three key files:

1. **`DenseConnectLayer.py`**: Defines the custom layer that implements the densely connected layers.
2. **`densenet.py`**: Builds the DenseNet architecture using dense blocks, transition layers, and a final classification layer.
3. **`load_data.py`**: Orchestrates dataset loading, normalization and augmentation

We made the implementation modular by parameterizing the dataset (`cifar-10` or `cifar-100`), the use of bottleneck layers, and compression. These three parameters result in eight combinations, which were systematically tested.

Architectural Design

The DenseNet architecture was implemented as follows:

1. **Dense Blocks**: Each block consists of multiple `DenseConnectLayer` instances. Each layer applies batch normalization, ReLU activation, and a 3×3 convolution. For bottleneck configurations, a 1×1 convolution reduces dimensionality before the 3×3 convolution.
2. **Growth Rate**: Set to 12 as in the original paper, controlling the number of output channels per layer.
3. **Transition Layers**: Positioned between dense blocks to reduce feature map size using a 1×1 convolution and average pooling. Compression reduces the number of channels by a factor of 0.5.
4. **Final Layers**: A global average pooling layer followed by a dense layer for classification.
5. **Normalization and Augmentation**: Input data was normalized using dataset-specific mean and standard deviation values. Training data was augmented with random crops and horizontal flips.

This section outlines the dataset (4.1), the DenseNet model and training process (4.2), and the software design, including algorithms and pseudo code (4.3).

3.1 Data

The CIFAR-10 and CIFAR-100 datasets are widely used benchmarks in the field of computer vision and deep learning. Both datasets consist of labeled 32x32 color images, with CIFAR-10 containing 60,000 images across 10 classes, such as airplanes, cars, and cats, while CIFAR-100 features 100 classes with 600 images per class, grouped into 20 superclasses. The datasets are divided into 50,000 training and 10,000 test images. While CIFAR-10 is simpler, focusing on general object classification, CIFAR-100 offers a more challenging classification task due to its finer-grained labels, making both datasets essential tools for evaluating image recognition models.

3.2 Deep Learning Network

The DenseNet architecture was implemented using TensorFlow and Keras. A custom dense connection layer was designed to manage feature concatenation between

layers effectively, ensuring parameter efficiency and gradient flow. The core DenseNet model incorporates dense blocks and transition layers, adhering to the DenseNet configuration described above and in the original paper.

Data loading and preprocessing were handled by the load_data.py script, which prepares datasets for training and validation, ensuring the inputs are correctly formatted and normalized. Training routines were managed by a custom training script, which included a training loop, validation procedures, and dynamic learning rate adjustments using step decay. The learning rate schedule adjusts based on the epoch count to ensure optimal convergence.

The training process utilized the Sparse Categorical Cross Entropy loss function and Stochastic Gradient Descent (SGD) optimizer with momentum. Metrics such as Top-1 and Top-5 accuracy were computed during training and validation. The main notebook orchestrates the execution flow, including model training, validation, and performance visualization, providing detailed insights into loss and accuracy trends across epochs.

This modular implementation allows flexibility in experimenting with different configurations while ensuring scalability for larger datasets.

3.3 Software Design

Here is the link to the github repository. Below is high level pseudo code for each file.

DenseConnectLayer.py

```
FUNCTION DenseConnectLayer(inputs, growth_rate, dropout_rate):
  FOR each input in inputs:
    Apply batch normalization and ReLU activation
     Apply a 3x3 convolutional layer with `growth_rate` filters
    Optionally, apply dropout with `dropout_rate`
  END FOR

    Concatenate outputs of all previous layers with the current layer's output
  RETURN concatenated_output
```

DenseNet.py

```
FUNCTION DenseNet(input_shape, num_classes):
  Initialize input layer

 Apply a 7x7 convolutional layer with stride 2
  Apply max pooling with a 3x3 kernel and stride 2
  FOR each dense block:
      Apply `DenseConnectLayer` for the number of layers in the block
    IF not the last block:
      Add a transition layer with batch normalization, ReLU, 1x1 convolution, and average pooling
    END IF
  END FOR

  Apply global average pooling
    Apply a fully connected layer with `num_classes` outputs
  RETURN model
```

load_data.py

```
FUNCTION load_data():
  Load dataset (e.g., CIFAR-10 or CIFAR-100)
  Normalize image pixel values to the range [0, 1]
  Split data into training and validation sets
  Convert labels to one-hot encoding (if required)
  RETURN train_dataset, val_dataset
```

main.ipynb

```
CALL load_data() to get train_dataset and val_dataset

CALL DenseNet(input_shape, num_classes) to create the model

DEFINE loss function (SparseCategoricalCrossentropy)
DEFINE optimizer (SGD with momentum and learning rate decay)
Initialize Trainer class with model, optimizer, loss, and hyperparameters

FOR each epoch in range(1, num_epochs + 1):
  CALL trainer.train(epoch, train_dataset)
  CALL trainer.validate(val_dataset)
  Record training and validation metrics
END FOR
```

4. Implementation

Experimental Configurations

- **Datasets**: CIFAR-10 and CIFAR-100 (32x32 images with 10 and 100 classes, respectively).
- **Bottleneck Layers**: Enabled or disabled.
- **Compression**: Applied (reduction=0.5) or not applied (reduction=1.0).
- **Hyperparameters**: Growth rate = 12 or 24, depth = 40 or 100, dropout rate = 0.2, batch size = 64, optimizer = SGD with momentum (0.9), and initial learning rate = 0.1 with cosine decay.
- **Hardware**: Experiments were conducted on GCP with an NVIDIA T4 GPU.

Dense Block Implementation

The core dense block was implemented using the `DenseConnectLayer` class. Each layer concatenates its output with all previous layers in the block, ensuring maximum feature reuse. Bottleneck layers reduced the number of intermediate channels to using a 1×1 convolution before the main 3×3 convolution.

Transition Layers

Transition layers used 1×1 convolutions to reduce the number of channels, followed by average pooling. When compression was enabled, the number of channels was halved.

Stage	**Layer Type**	**Filter Size / Pool Size**	**Output Channels**	**Details**
Input	Input Layer	-	3	shape (size, size, 3)
Initial Convolution	Convolution (Conv2D)	7x7 (ImageNet) or 3x3	2 * growth_rate	Stride: 2 (ImageNet) or 1, no bias
Dense Block 1	DenseConnect Layers	3x3 (conv)	Growth Rate x N	N = (depth - 4) / num_blocks
Transition 1	Batch Norm + ReLU + Conv2D + AvgPool	1x1 (conv), 2x2 (pool)	Reduction x Channels	Reduction = 0.5 (if compression enabled)
Dense Block 2	DenseConnect Layers	3x3 (conv)	Growth Rate x N	Similar to Dense Block 1
Transition 2	Batch Norm + ReLU + Conv2D + AvgPool	1x1 (conv), 2x2 (pool)	Reduction x Channels	Similar to Transition 1
Dense Block 3	DenseConnect Layers	3x3 (conv)	Growth Rate x N	Similar to Dense Block 1
Transition 3	Batch Norm + Global Average Pool	Dynamic Pool Size	-	Global Average Pool for feature map
Output	Dense Layer (FC)	-	num_classes	softmax activation

Figure 4. *Flowchart of DenseNet Architecture*
This is a flowchart presenting the top-level workflow and detailed processes within the DenseNet implementation. Illustrated steps range from data input, passing through dense blocks, transition layers, and ending at the classification layer.

Training and Evaluation

We trained the model for 20 epochs for each configuration. Extending the training to 300 epochs resulted in deliberate overfitting, given our choice of "cosine" as the learning rate adjustment parameter, as evidenced in our checkpoint file overfitted sample -cifar100-nobc-k24-L100.ipynb. To strike a balance between an effective learning rate and practical training duration, we performed a few preliminary experiments and finalized 20 epochs as the optimal training duration.

The training dataset was augmented, and the test dataset remained unmodified. Performance was measured using top-1 accuracy on the test set. Each experiment was repeated for the following configurations:

Trial set 1:

CIFAR-100: Parameter trials

(which focus on recreating the result of original paper)

Model	GrowthRate&Depth
Densenet-BC(k=12)	40
Densenet-BC(k=12)	100
Densenet(k=12)	40
Densenet(k=24)	100

Trial set 2:

CIFAR-10:

(which focus on our own trials)

BottleNeck	Compression	GrowthRate&Depth
Yes	Yes	12 & 100
Yes	No	12 & 100
No	Yes	12 & 100
No	No	12 & 100

Trial set 3:

CIFAR-100:

(which focus on our own trials)

BottleNeck	Compression	GrowthRate&Depth
Yes	Yes	12 & 100
Yes	No	12 & 100
No	Yes	12 & 100
No	No	12 & 100

Figure 5. *Comparison Charts: Original Paper vs. Student Implementation*

4.1. Objectives and Technical Challenges

The objective of this project is to reproduce and analyze the DenseNet architecture as proposed in the original DenseNet-BC paper. DenseNet introduces densely connected convolutional layers to improve information flow and reduce parameter count compared to traditional architectures. The primary goal is to validate the architecture's performance on CIFAR-10 and CIFAR-100 datasets using TensorFlow while replicating the key elements of the DenseNet-BC configuration.

Key Objectives:

1. Implement the DenseNet architecture, including dense blocks and transition layers, based on the original paper.
2. Validate performance on CIFAR-10 and CIFAR-100 datasets using metrics such as training loss, accuracy, and test error.
3. Evaluate the effects of bottleneck layers, compression, and growth rate on the architecture's performance.
4. Compare the reproduced results with those reported in the original paper to identify any deviations and analyze their causes.

Technical Challenges

1. **Reproducing the DenseNet Architecture:** Translating the DenseNet-BC model from its original Lua-based Torch implementation into TensorFlow/Keras required careful consideration of layer operations, initialization, and dropout handling.
2. **Computational Constraints:** Training deep networks, such as DenseNet with 100 layers, demands significant computational resources. We employed an NVIDIA T4 GPU on GCP to balance model complexity and runtime feasibility.
3. **Data Preprocessing and Augmentation:** Normalizing the datasets to match the paper's preprocessing and implementing augmentation techniques such as random cropping and flipping posed challenges in aligning results.
4. **Limited Scope:** Unlike the original study, which included CIFAR-10, CIFAR-100, SVHN, and ImageNet datasets, our reproduction focused on CIFAR-10 and CIFAR-100 due to time and resource constraints.

4.2.Problem Formulation and Design Description

DenseNet addresses the inefficiency of conventional convolutional networks, where feature maps are often discarded between layers. By introducing dense connections, every layer has direct access to all preceding layers' feature maps, which enhances gradient flow and reduces the number of parameters. This is mathematically represented as:

Where:

- : Output of the -th layer
- : Composite function (Batch Normalization, ReLU, Convolution)
- : Concatenation of feature maps from all preceding layers

The network is divided into dense blocks interspersed with transition layers that reduce spatial dimensions and channel counts via convolution and pooling operations.

Design Description:

The architecture was implemented in three components:

1. **DenseConnectLayer (denseconnectlayer.py):**
    - Implements a single layer of the dense block, supporting bottleneck and dropout configurations.
    - Pseudo-code:

```
class DenseConnectLayer:
    def __init__(n_channels, growth_rate, drop_rate,
bottleneck):
        if bottleneck:
            # Bottleneck: BN -> ReLU -> 1x1 Conv
            Apply BatchNorm -> ReLU -> 1x1 Conv -> Dropout
        # Main path: BN -> ReLU -> 3x3 Conv
        Apply BatchNorm -> ReLU -> 3x3 Conv -> Dropout

    def call(inputs):
        Process inputs through bottleneck and/or main path
        Concatenate outputs with inputs
```

2. **DenseNet Model (densenet.py):**
    - Constructs the full DenseNet-BC architecture by stacking dense blocks and transition layers. Each block consists of multiple DenseConnectLayers.
    - Pseudo-code:

```
def add_dense_block(x, num_layers, growth_rate, drop_rate,
bottleneck):
    for each layer in num_layers:
        Create DenseConnectLayer with growth_rate, drop_rate,
bottleneck
        Concatenate output with x
    return x

def add_transition(x, reduction, drop_rate, last):
    Apply BatchNorm -> ReLU -> 1x1 Conv -> Dropout (optional)
    Apply AveragePooling if not the last layer
    return x

def create_model(dataset_name, bottleneck, compression):
    Define input layer
    Apply initial convolution
    Add dense blocks and transition layers
    Apply global pooling and dense output layer
    return model
```

3. **Data Loader (load_data.py):**
    - Prepares CIFAR-10 and CIFAR-100 datasets with preprocessing and augmentation.
    - Pseudo-code:

```
def load_data(dataset):
    if dataset == 'cifar-10':
        Load CIFAR-10 dataset
    elif dataset == 'cifar-100':
        Load CIFAR-100 dataset

    Normalize images using dataset-specific mean and std
    Augment training data with random cropping and flipping
    return train_data, train_labels, test_data, test_labels
```

By structuring the design in this manner, we replicated the DenseNet-BC functionality while simplifying certain aspects (e.g., dropout omission and dataset scope) due to resource limitations.

Key Design Considerations:

- Balancing growth rate and depth for efficient training.
- Implementing transition layers to manage model complexity.
- Augmenting data to improve generalization on CIFAR-10 and CIFAR-100.

5. Results

5.1 Project Results

CIFAR-100: Parameter trials

(which focus on recreating the result of original paper)

Top - 1:

Model	Growth Rate&Depth	Params	Train Acc	Test Acc
Densenet-BC(k=12)	40	0.232 M	**61.96**	**38.17**
Densenet-BC(	100	0.926 M	**77.78**	**40.28**

k=12)				
Densenet(k=12)	40	0.232 M	**61.66**	**35.37**
Densenet(k=24)	100	3.334 M	**81.61**	**45.64**

Top - 5:

Model	Growth Rate&Depth	Params	Train Acc	Test Acc
Densenet-BC(k=12)	40	0.232 M	**87.90**	**65.77**
Densenet-BC(k=12)	100	0.926 M	**95.44**	**68.15**
Densenet(k=12)	40	0.232 M	**87.92**	**63.52**
Densenet(k=24)	100	3.334 M	**96.76**	**74.28**

CIFAR-10:

(which focus on our own trials)

Top - 1:

B/C	GrowthRate&Depth	Params	Train Acc (top-1)	Test Acc(top-1)
Yes/Yes	12 & 100	0.926 M	**91.56**	**77.98**
Yes/No	12 & 100	0.926 M	**91.07**	**77.71**
No/Yes	12 & 100	0.926 M	**91.90**	**79.00**
No/No	12 & 100	0.926 M	**91.03**	**76.56**

Top - 5:

B/C	GrowthRate&Depth	Params	Train Acc(top-5)	Test Acc(top-5)
Yes/Yes	12 & 100	0.926 M	**99.84**	**98.55**
Yes/No	12 & 100	0.926 M	**99.83**	**98.15**
No/Yes	12 & 100	0.926 M	**99.87**	**98.58**
No/No	12 & 100	0.926 M	**99.84**	**98.11**

CIFAR-100:

(which focus on our own trials)

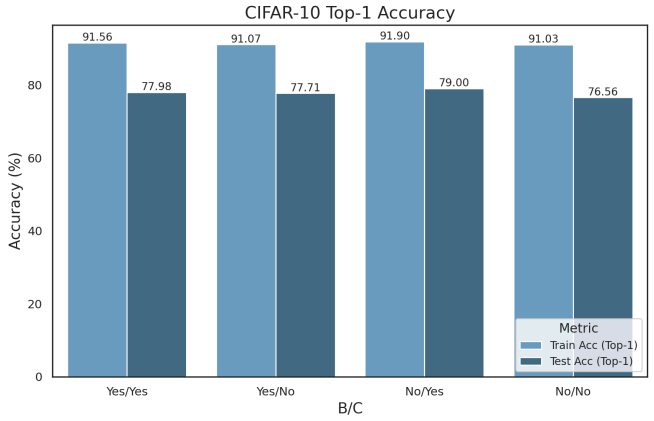
Top - 1:

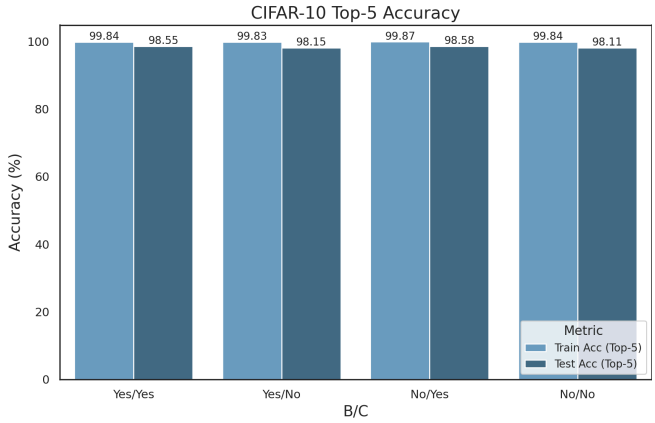
B/C	GrowthRate &Depth	Params	Train Acc	Test Acc
Yes/Yes	12 & 100	0.926 M	**77.78**	**40.28**
Yes/No	12 & 100	0.926 M	**77.20**	**41.45**
No/Yes	12 & 100	0.926 M	**77.64**	**40.61**
No/no	12 & 100	0.926 M	**77.70**	**36.42**

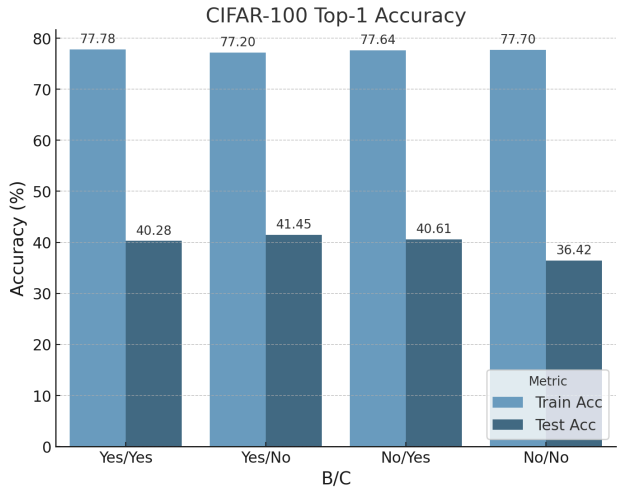
Top - 5:

B/C	GrowthRate &Depth	Params	Train Acc	Test Acc
Yes/Yes	12 & 100	0.926 M	**95.44**	**68.15**
Yes/No	12 & 100	0.926 M	**95.19**	**70.40**
No/Yes	12 & 100	0.926 M	**95.41**	**69.07**
No/No	12 & 100	0.926 M	**95.18**	**64.93**

Figure 6. *Our Performance Tables*







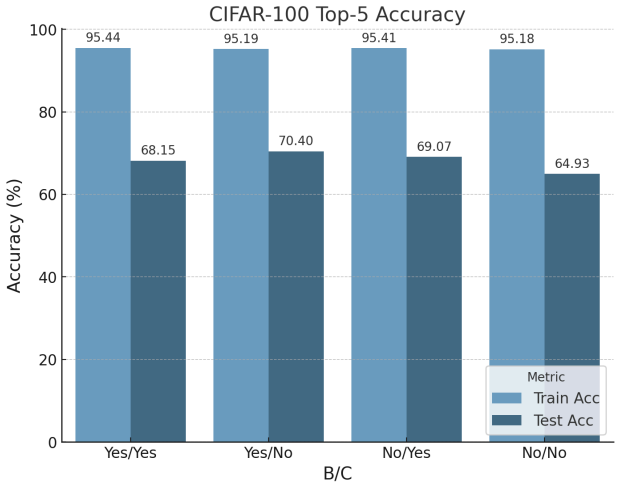
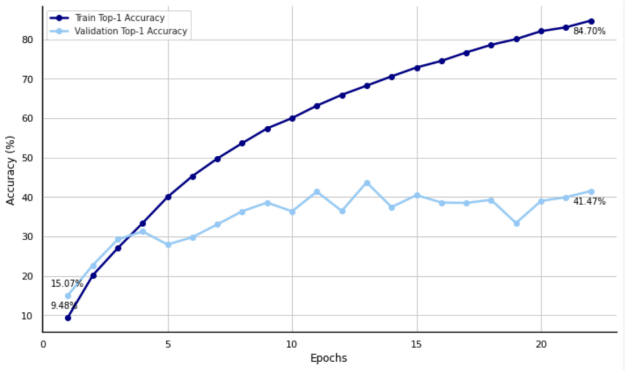


Figure 7. *Our Top-1 and Top-5 Accuracy Tables*



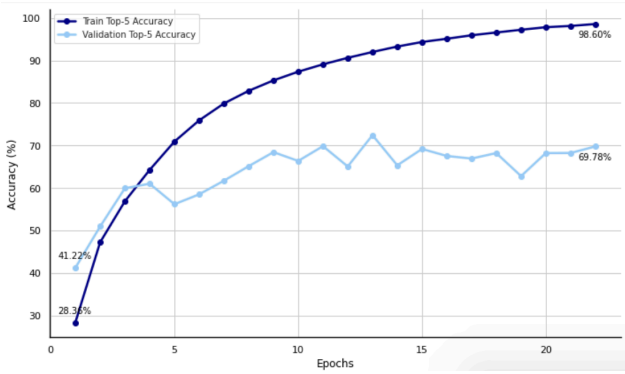


Figure 8.1 . *Our Train and Test loss Tables(initial epoch = 300 trial, lead to overfitting)*

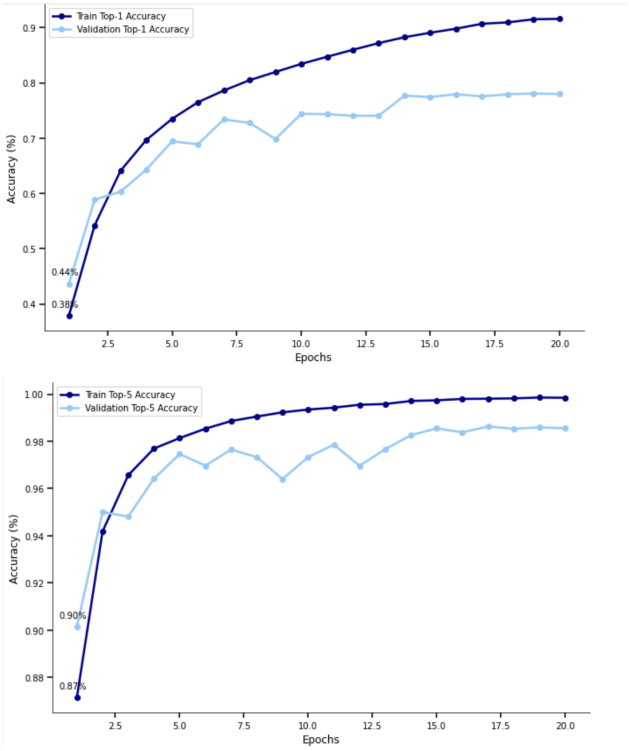


Figure 8.2 . *Our Train and Test loss Tables(CIFAR-10-B-C-k12-L1-0)*

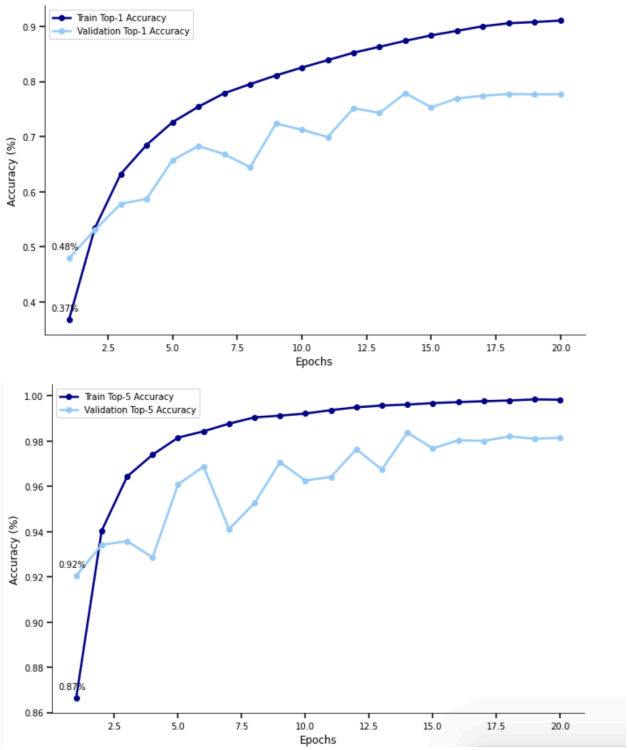


Figure 8.3 . *Our Train and Test loss Tables(CIFAR-10-B-C-k12-L100)*

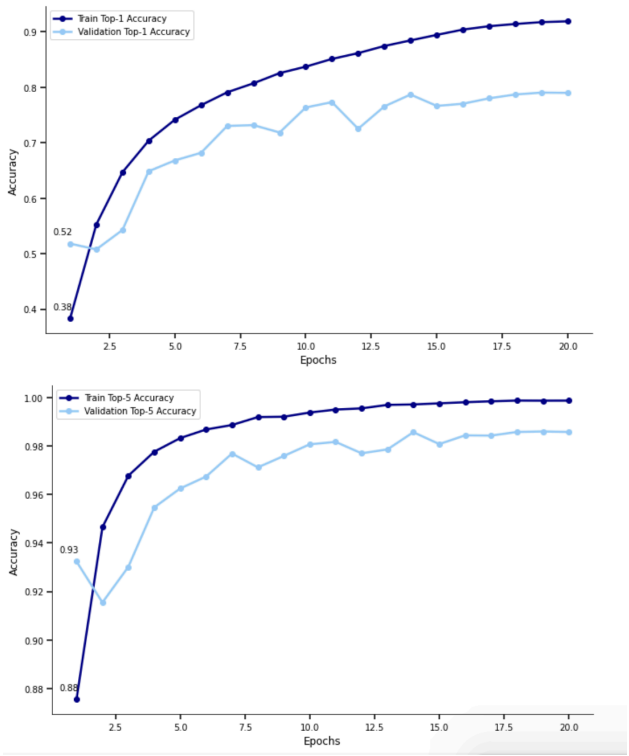


Figure 8.4 . *Our Train and Test loss Tables(CIFAR-10-noB-C-k12-L100)*

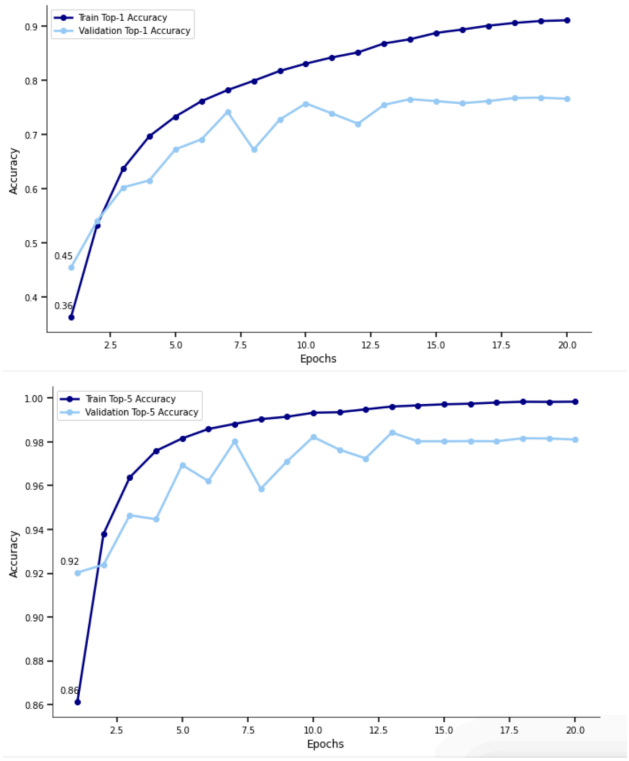


Figure 8.5 . *Our Train and Test loss Tables(CIFAR-10-noB-noC-k12-L100)*

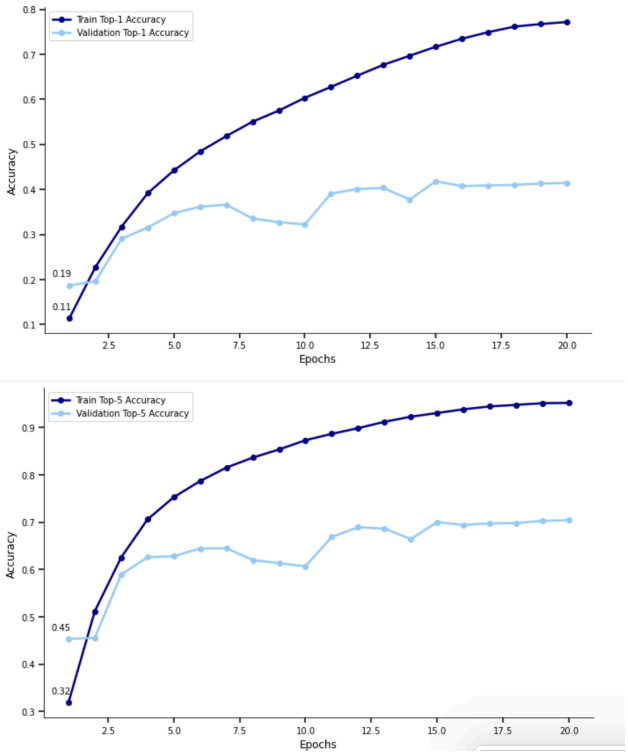


Figure 8.6 . *Our Train and Test loss Tables(CIFAR-100-B-noC-k12-L100)*

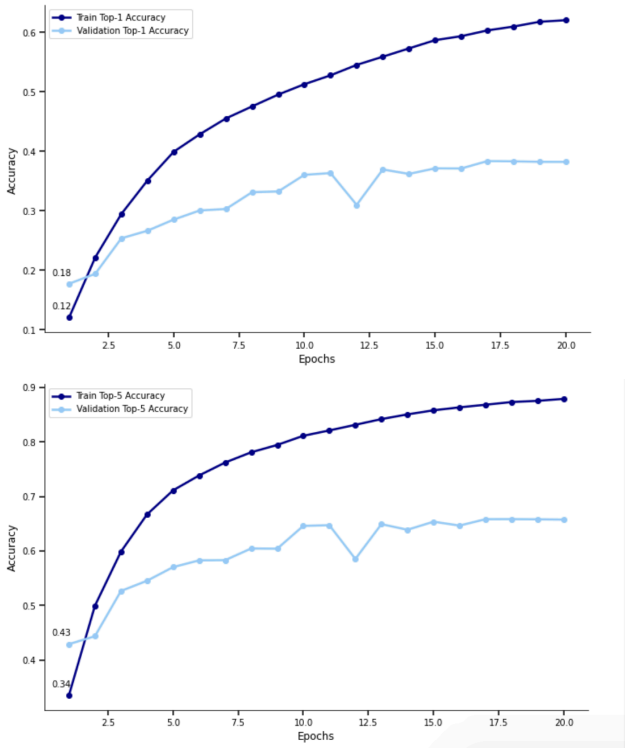


Figure 8.7 . *Our Train and Test loss Tables(CIFAR-100-B-C-k12-L40)*

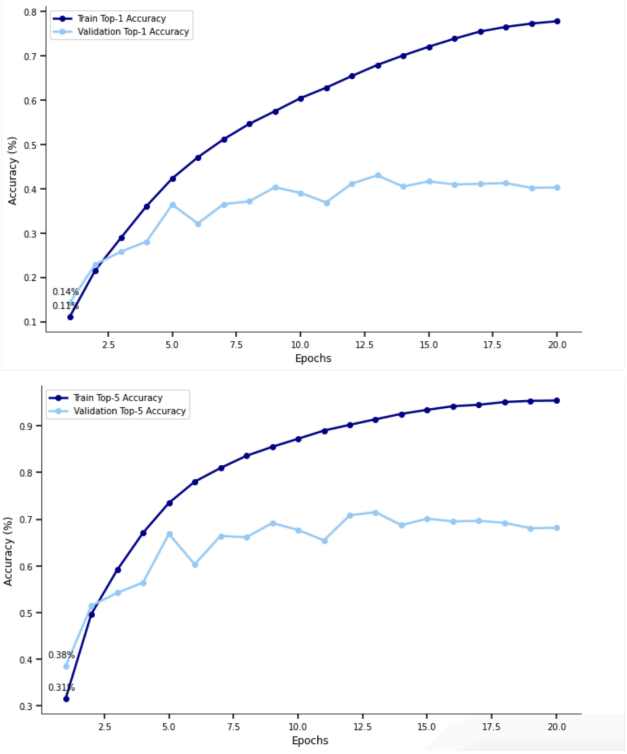


Figure 8.8 . *Our Train and Test loss Tables(CIFAR-100-B-C-k12-L100)*

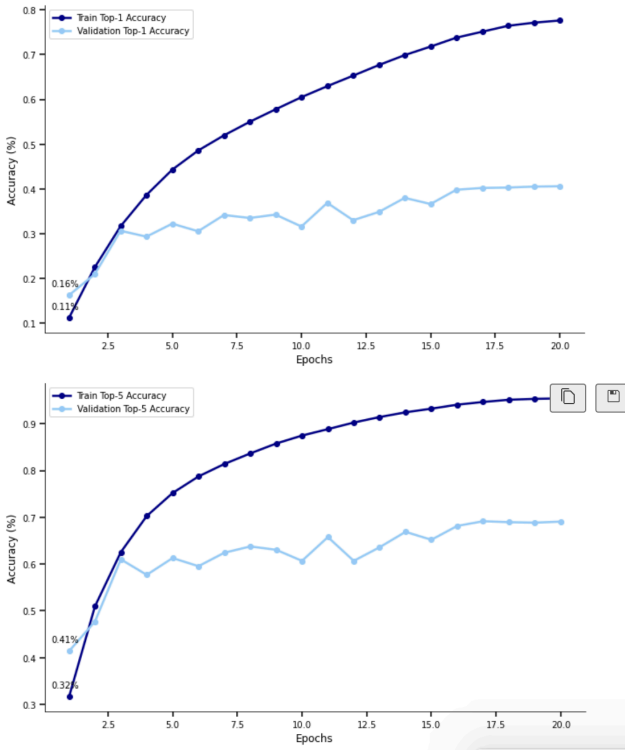


Figure 8.19 *. Our Train and Test loss Tables(CIFAR-100-noB-C-k12-L100)*

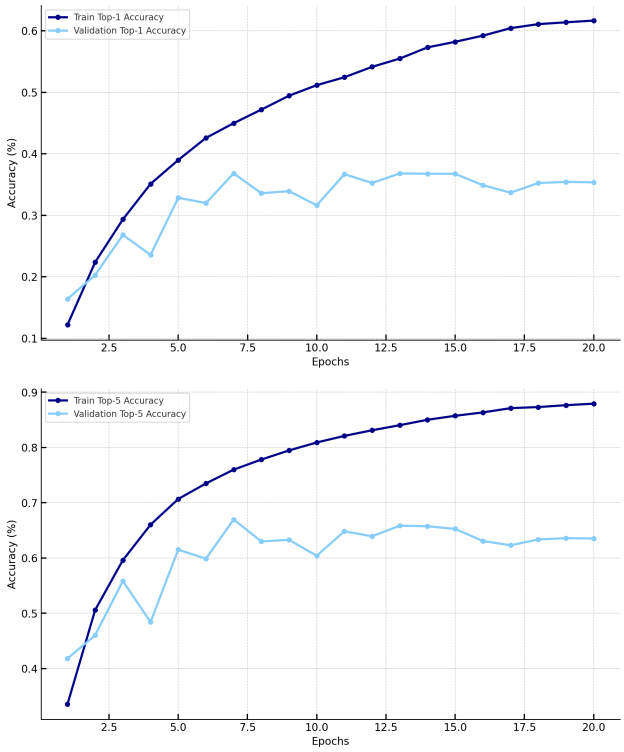


Figure 8.10 *. Our Train and Test loss Tables(CIFAR-100-noB-noC-k12-L40)*

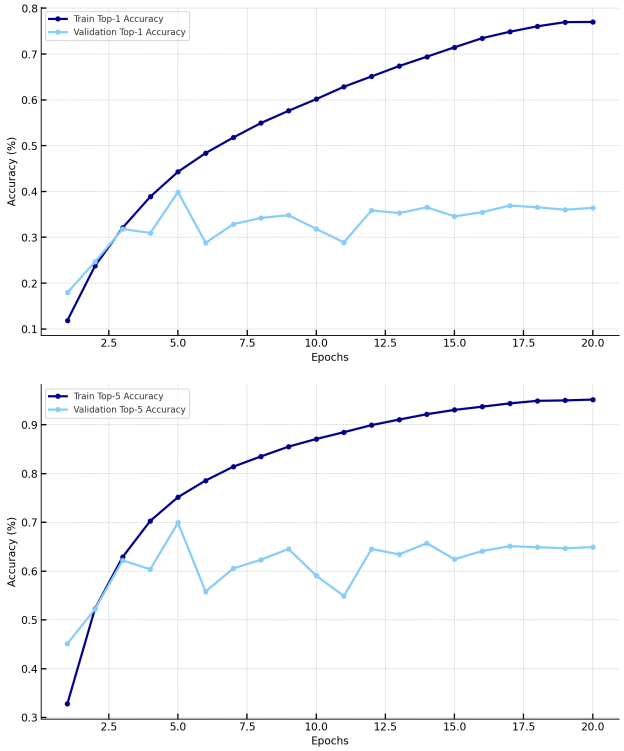


Figure 8.11 *. Our Train and Test loss Tables(CIFAR-100-noB-noC-k12-L100)*

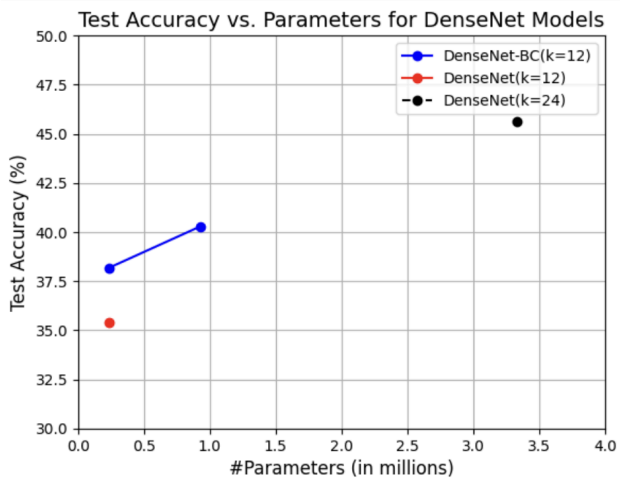


Figure 9. *Our Parameters and Accuracy Tables. High parameters scale lead to ideal performance.*

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

Our recreation of original paper:

	epochs	Error rate	details
Our results	20	1. Our error rate is approximately 5 times higher than reported in the original paper for CIFAR-10 and 3 times higher for CIFAR-100. 2. While we observed some efficiency improvements with Bottleneck and Compression for CIFAR-100, the impact was not as significant as expected overall.	Each takes about 1.5 hours to train with batch_size = 64 and initial lr = 0.1
Original paper	300	Their model demonstrates a clear positive correlation with the use of Bottleneck and Compression (BC), growth rate, and depth. However, due to computational limitations, we were unable to thoroughly test higher growth rates and depths, as the system failed when attempting configurations like $k = 24$ with $L = 100$ or more.	They are available to train models with more than 10M params, while our GCP broke down frequently when it comes to 10M+

Figure 10. *Our Recreation of original paper result analysis*

Our trial on the combination of Bottleneck & Compression:

	Conclusion	Details
Our results	We did not observe significant improvement with either bottleneck or compression. **We believe it is due to our parameter scale.** Based on our experimental data, while bottleneck sometimes showed a slightly higher accuracy, we believe these differences are not statistically significant. As a result, we lack sufficient evidence to conclude that bottleneck or compression alone contributes to model improvement.	In our case, models with BC consistently outperformed those without BC, aligning with the findings in the original paper.
Original paper	As observed in the paper, the performance difference with BC becomes noticeable once the parameter count exceeds 2 million.	When combined, BC consistently outperforms configurations without BC in terms of test error rate.

Figure 11. *Our own trials result analysis*

While our implementation aligns closely with the original DenseNet-BC architecture, we focused on a subset of the experiments due to computational constraints. Notable differences include:

- **Dropout**: We use dropout rate at 0.2, whereas the original paper used it selectively.
- **Depth and Growth Rate**: We fixed the depth at 100 and growth rate at 12, while the original paper explored additional configurations.
- **Dataset-Specific Optimizations**: We normalized and augmented data as described in the paper but omitted additional dataset-specific optimizations due to time constraints.
- **Implementation and Datasets**: The original paper was implemented in Lua and tested on a broader set of datasets, including SVHN and ImageNet, in addition to CIFAR-10 and CIFAR-100.

5.3 Discussion / Insights Gained

In our experimental endeavors, we encountered several challenges that influenced our results, particularly when compared to the original paper's findings. These challenges encompassed learning rate and epoch configurations, overfitting tendencies, and hardware limitations.

Learning Rate and Epochs:

Initially, we adopted a learning rate (LR) of 0.1 with a cosine adjustment over 300 epochs, aligning with standard practices for training DenseNet architectures on datasets like CIFAR-10 and CIFAR-100. However, this configuration led to overfitting after just 10 epochs, with the learning rate decreasing at a sluggish pace. Additionally, training for 22 epochs required approximately three hours, which was impractical given our computational constraints. This prolonged training time can be attributed to the limited memory capacity of our GPU (7.5 GB), which hampers efficient learning and convergence. To mitigate overfitting and reduce training duration, we adjusted the training regime to 20 epochs. This adjustment struck a balance, as evidenced by a continuous decrease in training loss and a generally declining, albeit occasionally fluctuating, test loss. Consequently, we determined that tailoring the learning rate and epoch count to our specific computational resources was essential, concluding that an initial LR of 0.1 and 20 epochs were optimal for our setup.

Overfitting:

We also explored the impact of dropout rates on overfitting. Implementing a dropout rate of 0.0 resulted in overfitting across both datasets, regardless of other hyperparameter settings. This observation underscores the necessity of incorporating appropriate regularization techniques to enhance model generalization, especially when computational limitations restrict the complexity and duration of training.

Hardware Limitations and Result Variability:

Our experiments revealed that different trials yielded varying outcomes, including fluctuations in overfitting tendencies and test error rates. This variability is inherent in training deep neural networks, where stochastic processes and initial conditions can lead to divergent results. Given our hardware constraints, particularly the limited GPU memory, conducting multiple trials and averaging the results would provide a more accurate assessment of the model's performance. However, due to resource limitations, we were unable to perform extensive repetitions. This limitation likely contributed to the discrepancies between our findings and those reported in the original paper.

6. Future Work

As we were constrained by our GCP compute resources, we focused on the core elements of the paper. Future work could include exploring the following:

1. **Architectural Enhancements**: Experimenting with variations of DenseNet, such as reducing the number of parameters while maintaining performance or integrating attention mechanisms to improve feature learning.
2. **Dataset Expansion**: Testing the model on larger, more complex datasets, or applying domain adaptation techniques to evaluate generalizability across diverse image datasets.
3. **Optimization Techniques**: Investigating advanced optimization strategies like adaptive learning rates, gradient clipping, or incorporating optimizers like AdamW to enhance convergence.
4. **Explainability and Interpretability**: Applying techniques such as Grad-CAM or SHAP to understand model predictions and feature importance.
5. **Transfer Learning**: Leveraging the trained DenseNet model for transfer learning on domain-specific tasks, such as medical imaging or satellite imagery classification.

7. Conclusion

This project successfully reproduced the core DenseNet-BC architecture and validated it on CIFAR-10 and CIFAR-100 datasets. While we observed trends consistent with the original paper, such as improved accuracy with bottleneck and compression layers, our error rates were higher—approximately five times higher for CIFAR-10 and three times higher for CIFAR-100—due to computational and resource constraints. Despite some efficiency improvements with Bottleneck and Compression for CIFAR-100, the impact was less significant than expected. Key lessons learned include the importance of parameter tuning, regularization techniques (such as dropout), and the need for adequate hardware resources to handle larger model configurations. Future work should extend to experimenting with deeper architectures, larger datasets, and more advanced optimization techniques, such as adaptive learning rates or gradient clipping, to better align with the original DenseNet benchmarks. Overall, the project reaffirmed DenseNet's strengths in feature reuse, parameter efficiency, and scalability while highlighting the challenges of reproducing large-scale deep learning models with limited computational resources.

6. Acknowledgement

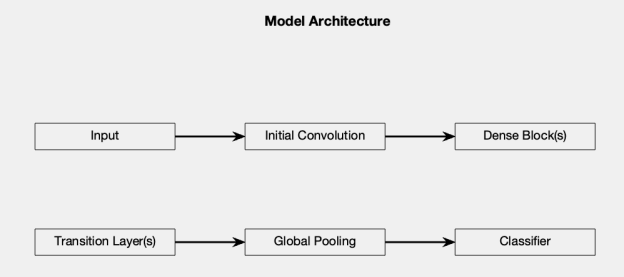
Used LLMs for research purposes. We would like to thank the course staff and the professor for a wonderful semester, and we learned a lot.

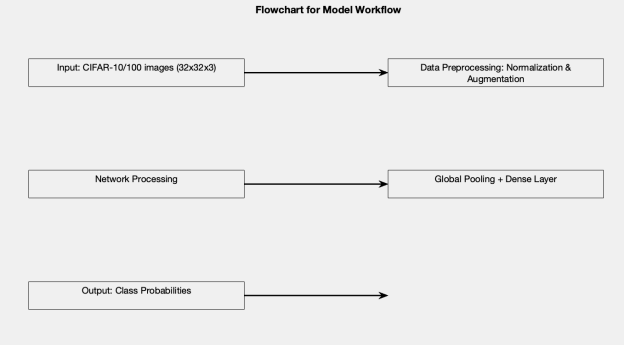
7. References

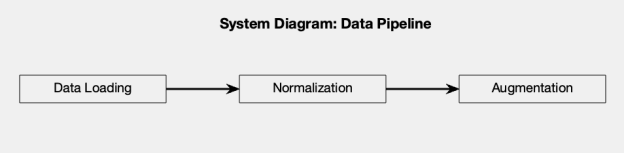
Include all references - papers, code in github, links to web pages, books. Use IEEE style of referencing (https://ieeeauthorcenter.ieee.org/wp-content/uploads/IEEE-Reference-Guide.pdf).
Examples:
[1] https://github.com/ecbme4040/e4040-2024fall-project-anlm
[2] G. Huang, Z. Liu, L. Van Der Maaten and K. Q. Weinberger, "Densely Connected Convolutional Networks," 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, USA, 2017, pp. 2261-2269, doi: 10.1109/CVPR.2017.243.
[3] H. Li, "Author Guidelines for CMPE 146/242 Project Report", *Lecture Notes of CMPE 146/242*, Computer Engineering Department, College of Engineering, San Jose State University, March 6, 2006, pp. 1.
[4] TensorFlow: Large-scale machine learning on heterogeneous systems,
2015. Software available from tensorflow.org.

8. Appendix

Figures: Additional illustrations to highlight overall architecture, workflow and pipeline







8.1 Individual Student Contributions in Fractions

	jl6865	amn2225	lm3879
Last Name	Liu	Natarajan	Ma
Fraction of (useful) total contribution	1/3	1/3	1/3
What I did 1	Code skeleton and first draft of DenseConnectLayer and densenet	Implemented load_data and created general structure for our project	Conducted training for all trials, adjusted all hyperparameters, and designed experiments along with comparison methods.
What I did 2	Wrote parts-	Edited and	Designed

	Introduction, Summary of Original Paper, Methodology, generation of corresponding figures and editing of overall report	solidified DenseNet, DenseLayer, and training loop and implemented everything together to get it working	and created those tables, charts, and graphs for the paper.
What I did 3	Testing of CIFAR-100 and initial checkpoints for individual DenseNet files	Wrote sections of the paper, specifically section 3/6	Composed the result analysis, data comparisons, and discussion & insights.

1. OVERALL CONTENT (~25%):
    a. Review of literature
    b. Description of the problem
    c. Description of student's implementation
    d. Flowcharts and block diagrams - student's implementation
    e. Figures
    f. Quality of writing

2. RESULTS (~25%):
    a. Accuracy
    b. Training times
    c. Effort in optimizing results
    d. Figures
    e. Tables

3. DISCUSSION (~25%):
    a. Discussion of results
    b. Comparison with other paper(s)
    c. Discussion of problems faced
    d. Discussion of insights gained

4. CODE (~25%):
    a. README content
    b. File organization
    c. Code comments
    d. Code works
    e. Code is original, not plagiarized