CSE-411 Emerging Topics in Computer Science

Assignment 1

Comparative Study and Implementation of AlexNet and VGG Architectures on CIFAR-10 Dataset

Student Name: Sara Basheer Mohamed

Student ID: 120220035

Course Instructor: Dr. Ehab Elshazly

April 24, 2025

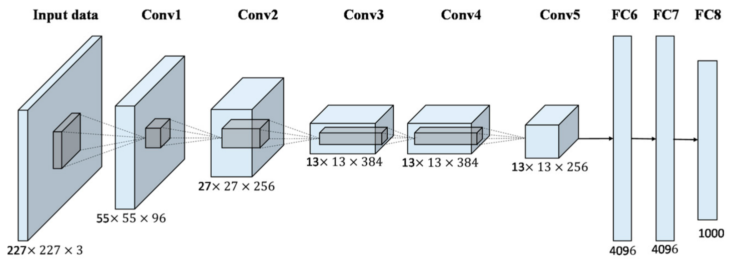
Contents

1 Architecture Designs and Components

1.1 AlexNet

AlexNet was first konwn from the paper "ImageNet Classification with Deep Convolutional Neural Networks" in 2012 and it was a breakthrough at that time.

- **Convolutional Layers**:
  - 5 convolutional layers
  - First layer: 96 filters of size 11×11 with stride 4
  - Second layer: 256 filters of size 5×5 with stride 1
  - Remaining layers: 384, 384, and 256 filters respectively, of size 3×3 with stride 1
- **Activation Functions**: ReLU (Rectified Linear Unit) after each convolutional(5 layers) and 2 of the fully connected layer. The activation function used for the last layer is softmax.
- **Pooling Strategy**:
  - Max pooling with 3×3 filters and stride 2
  - Applied after first, second, and fifth convolutional layers
- **Fully Connected Layers**:
  - 3 fully connected layers
  - First two with 4096 neurons each
  - Last layer with 1000 neurons (for ImageNet classification)
  - Dropout used in first two fully connected layers (rate=0.5)

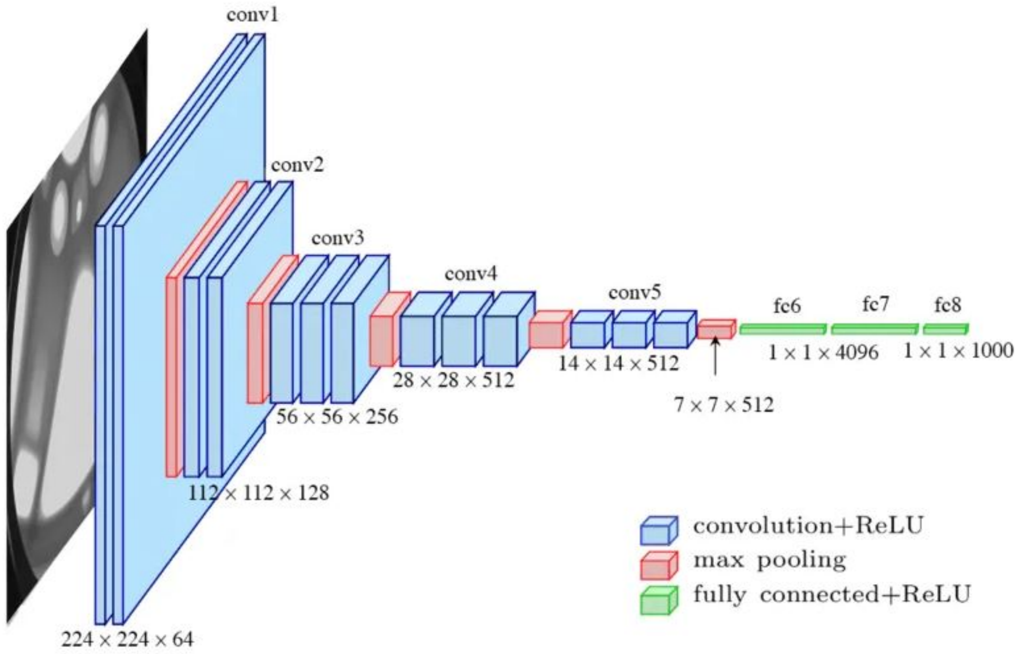


1.2 VGG-16

VGG-16 is characterized by its simplicity, using only 3×3 convolutional layers stacked in increasing depth.

- **Convolutional Layers**:
  - 13 convolutional layers
  - All filters are 3×3 with stride 1 and padding 1
  - Depth increases through the network: 64, 128, 256, 512
  - Multiple consecutive conv layers at each depth (2 for 64/128, 3 for 256/512)
- **Activation Functions**: ReLU after each convolutional layer
- **Pooling Strategy**:
  - Max pooling with 2×2 filters and stride 2
  - Applied after each block of convolutional layers (5 total pooling layers)
- **Fully Connected Layers**:
  - 3 fully connected layers
  - First two with 4096 neurons each

- Last layer with 1000 neurons (for ImageNet classification)
- Dropout used in first two fully connected layers (rate=0.5)



1.3 VGG-19

VGG-19 is a deeper variant of VGG-16 with slightly better performance but higher computational cost.

- **Convolutional Layers**:
  - 16 convolutional layers
  - Same 3×3 filters with stride 1 and padding 1 as VGG-16
  - Additional conv layers in the 256 and 512 depth blocks (4 layers each instead of 3)
- **Activation Functions**: ReLU after each convolutional layer (same as VGG-16)
- **Pooling Strategy**:
  - Identical to VGG-16: 2×2 max pooling with stride 2 after each block
- **Fully Connected Layers**:
  - Same structure as VGG-16: 4096-4096-1000

2 Architectural Comparison: AlexNet vs. VGG

Below is a summary table comparing the key architectural details of AlexNet, VGG-16, and VGG-19, followed by a detailed comparison.

Table 1: Summary of Architectural Details

Feature	**AlexNet**	**VGG-16**	**VGG-19**
Convolutional Layers	5	13	16
Fully Connected Layers	3	3	3
Total Parameters (M)	∼60	138	144
Filter Sizes	11×11, 5×5, 3×3	3×3	3×3
Pooling Strategy	3×3, stride 2	2×2, stride 2	2×2, stride 2
Activation Function	ReLU	ReLU	ReLU
Dropout Rate (FC Layers)	0.5	0.5	0.5

Model Depth

AlexNet: 8-layer architecture (5 convolutional + 3 fully connected)
VGG: Deeper variants (VGG-16: 13 conv + 3 FC, VGG-19: 16 conv + 3 FC)

Parameter Count

AlexNet: ∼60 million parameters
VGG: VGG-16 (138M), VGG-19 (144M) parameters

Feature Extraction Strategy

AlexNet:

- Progressive downscaling (11×11 → 5×5 → 3×3 filters)
- Mixed receptive field sizes

VGG:

- Uniform 3×3 filters throughout
- Stacked convolutions for hierarchical features

Filter Design Philosophy

AlexNet:

- Decreasing filter sizes per layer
- Combines large and small receptive fields

VGG:

- "Small filters deep network" approach
- Multiple 3×3 filters simulate larger receptive fields

Regularization Techniques

AlexNet:

- Local Response Normalization (LRN)
- Dropout (p=0.5) in FC layers (dropout was described to be " a recently-developed regularization method" because AlexNet was one of the first networks in which dropout was used to reduce overfitting)

VGG:

- Only dropout in FC layers (no LRN)
- Implicit regularization through depth

Generalization Approach

AlexNet:

- PCA-based color augmentation
- Moderate depth prevents overfitting

VGG:

- Requires batch normalization in modern implementations

- Depth necessitates strong regularization

Table 2: Key Differences in Design Philosophy

Aspect	AlexNet	VGG
Filter Size Strategy	Mixed (11×11 to 3×3)	Uniform (3×3)
Depth	Shallow (8 layers)	Deep (16–19 layers)
Regularization	LRN + Dropout	Dropout Only
Parameter Efficiency	Moderate	Lower
Computational Cost	Lower	Higher

3 Analysis of the receptive field growth in each network

In the first cell of my last attempt, I included code snippet to visualize the growth of the receptive field in each network. It is obvious that the receptive field of vgg16 grows slower than that of alexnet.

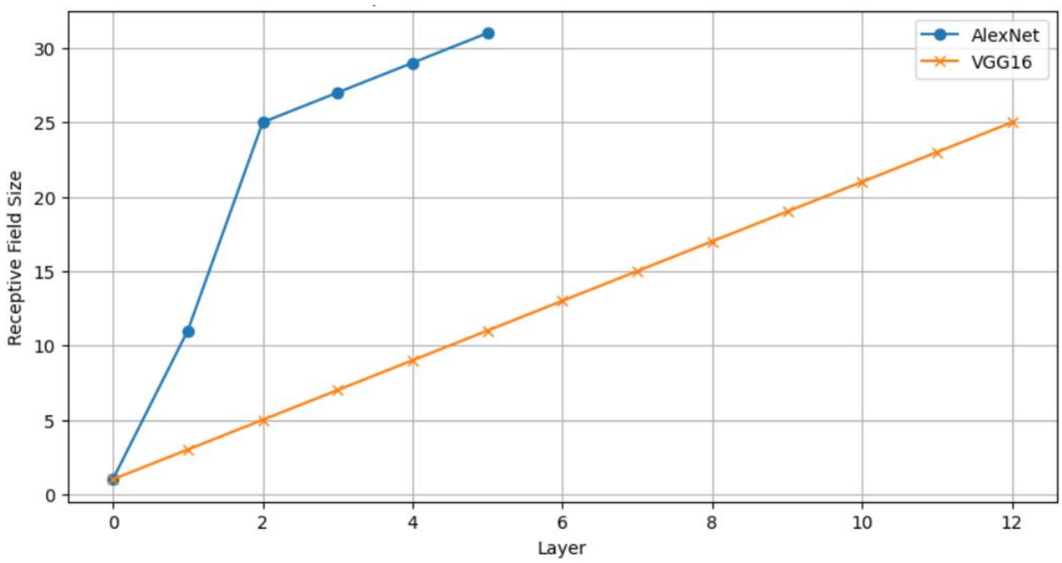
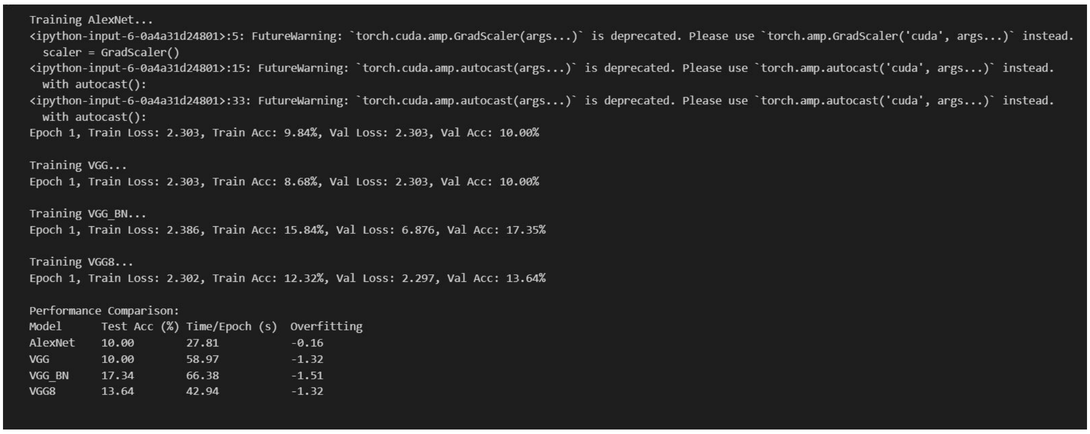


Figure 1: Performance comparison of all networks

4 Notes and takeaways from failed attempts

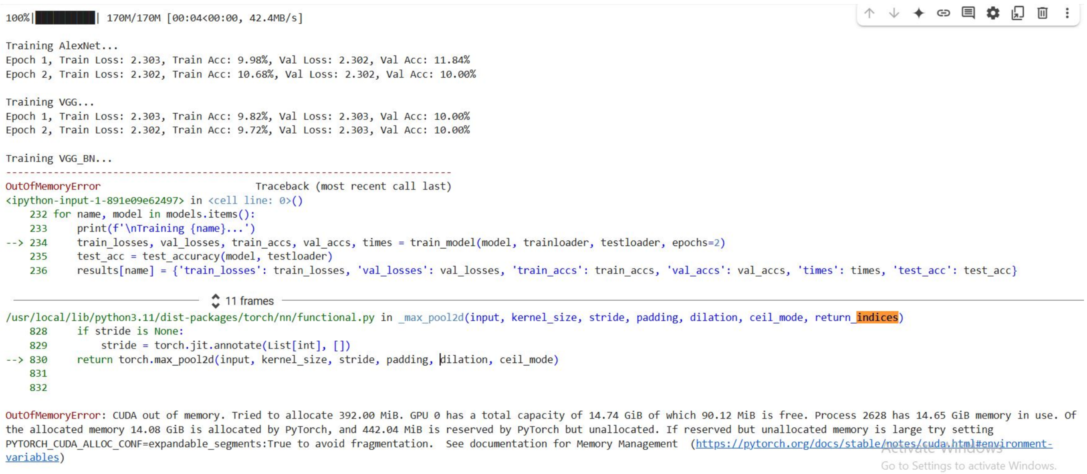
Attempt 1: I was just trying to get the classes built for the models. The problem is that the performance of the 2 models was pretty poor. Only 5% of the training dataset was used.i.e. 5000 images instead of 50,000 In addition, the no. of epochs was 1 for training. The accuracy in the test dataset was around 10%. Since the CIFAR dataset has only 10 classes, the test accuracy of 10% is barely not better than random guessing.



```
Training AlexNet...
<ipython-input-6-0a4a31d24801>:5: FutureWarning: `torch.cuda.amp.GradScaler(args...)` is deprecated. Please use `torch.amp.GradScaler('cuda', args...)` instead.
  scaler = GradScaler()
<ipython-input-6-0a4a31d24801>:15: FutureWarning: `torch.cuda.amp.autocast(args...)` is deprecated. Please use `torch.amp.autocast('cuda', args...)` instead.
  with autocast():
<ipython-input-6-0a4a31d24801>:33: FutureWarning: `torch.cuda.amp.autocast(args...)` is deprecated. Please use `torch.amp.autocast('cuda', args...)` instead.
  with autocast():
Epoch 1, Train Loss: 2.303, Train Acc: 9.84%, Val Loss: 2.303, Val Acc: 10.00%

Training VGG...
Epoch 1, Train Loss: 2.303, Train Acc: 8.68%, Val Loss: 2.303, Val Acc: 10.00%

Training VGG_BN...
Epoch 1, Train Loss: 2.386, Train Acc: 15.84%, Val Loss: 6.876, Val Acc: 17.35%

Training VGG8...
Epoch 1, Train Loss: 2.302, Train Acc: 12.32%, Val Loss: 2.297, Val Acc: 13.64%

Performance Comparison:
Model      Test Acc (%) Time/Epoch (s)  Overfitting
AlexNet    10.00        27.81           -0.16
VGG        10.00        58.97           -1.32
VGG_BN     17.34        66.38           -1.51
VGG8       13.64        42.94           -1.32
```

Figure 2: Performance comparison of all networks

Attempt 2: The percentage used from the original CIFAR dataset was increased to 10%; seeking better performance. However, I got my GPU running out of memory, and I was not able to see any results!



```
100%|██████████| 170M/170M [00:04<00:00, 42.4MB/s]

Training AlexNet...
Epoch 1, Train Loss: 2.303, Train Acc: 9.98%, Val Loss: 2.302, Val Acc: 11.84%
Epoch 2, Train Loss: 2.302, Train Acc: 10.68%, Val Loss: 2.302, Val Acc: 10.00%

Training VGG...
Epoch 1, Train Loss: 2.303, Train Acc: 9.82%, Val Loss: 2.303, Val Acc: 10.00%
Epoch 2, Train Loss: 2.302, Train Acc: 9.72%, Val Loss: 2.303, Val Acc: 10.00%

Training VGG_BN...
---------------------------------------------------------------------------
OutOfMemoryError                          Traceback (most recent call last)
<ipython-input-1-891e09e62497> in <cell line: 0>()
    232 for name, model in models.items():
    233     print(f'\nTraining {name}...')
--> 234     train_losses, val_losses, train_accs, val_accs, times = train_model(model, trainloader, testloader, epochs=2)
    235     test_acc = test_accuracy(model, testloader)
    236     results[name] = {'train_losses': train_losses, 'val_losses': val_losses, 'train_accs': train_accs, 'val_accs': val_accs, 'times': times, 'test_acc': test_acc}

11 frames
/usr/local/lib/python3.11/dist-packages/torch/nn/functional.py in _max_pool2d(input, kernel_size, stride, padding, dilation, ceil_mode, return_indices)
    828     if stride is None:
    829         stride = torch.jit.annotate(List[int], [])
--> 830     return torch.max_pool2d(input, kernel_size, stride, padding, dilation, ceil_mode)
    831
    832

OutOfMemoryError: CUDA out of memory. Tried to allocate 392.00 MiB. GPU 0 has a total capacity of 14.74 GiB of which 90.12 MiB is free. Process 2628 has 14.65 GiB memory in use. Of the allocated memory 14.08 GiB is allocated by PyTorch, and 442.04 MiB is reserved by PyTorch but unallocated. If reserved but unallocated memory is large try setting PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True to avoid fragmentation.  See documentation for Memory Management  (https://pytorch.org/docs/stable/notes/cuda.html#environment-variables)
```


Attempt 3: I went back to using only 5% of the training dataset for training. I added cells to visualize the filters, feature maps, and to compare between the performance of the networks built. The results were extremely poor and disappointing.

```
Performance Comparison:
Model      Test Acc (%) Time/Epoch (s)  Overfitting (%)
AlexNet    10.03        26.98           -0.11
VGG        10.00        62.62           0.20
VGG_BN     21.38        69.13           -6.29
VGG8       10.07        42.91           -0.11

Best Model: VGG_BN
Test Accuracy: 21.38%
```

Attempt 4: In this attempt, I printed the graphs of filters and feature maps inside the notebook rather than being saved as files in the same directory, just to make it more easy to track the results. In addition, I used only 5% of the dataset, and 10 epochs for training and testing. The results started to appear more reasonable. I know that the testing dataset's accuracy is expected to be around 80% to 90%. The accuracy in my notebook was too much below this range for the following reasons: 1- I only used 2,500 images for training (instead of 50,000 images) 2- I divided the dataset into 10 batches only (more epochs typically lead to higher test accuracy) 3- the dataset size was too small for such complex models to capture the patterns in the data and learn something out of it. The computational resources available for me limited my choices, so I just trained the models to get something working in the end. However, it was too difficult to train the models on the original dataset that has 50,000 images for taining.

```
Performance Comparison:
Model      Test Acc (%) Time/Epoch (s)  Overfitting (%)
AlexNet    28.94        26.69           -4.36
VGG        10.00        62.48           1.16
VGG_BN     34.65        69.76           -2.31
VGG8       42.66        43.39           0.19
```

Further attempts: some attempts failed to give better results. Due to to the fact that the small proportion of the dataset does not provide complete coverage, vgg_8 outperformed vgg_bn. However, the final version is reasonable and the results make sense.

References

[1] Siddhesh B. Vgg-net architecture explained. https://medium.com/@siddheshb008/vgg-net-architecture-explained-71179310050f, May 2023. Accessed: 2025-04-24.

[2] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016. Comparative study of computational requirements.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[4] Xiaobing Han, Yanfei Zhong, Liqin Cao, and Liangpei Zhang. Pre-trained alexnet architecture with pyramid pooling and supervision for high spatial resolution remote sensing image scene classification. *Remote Sensing*, 9(8):848, August 2017. This article belongs to the Special Issue Remote Sensing Big Data: Theory, Methods and Applications.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[6] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015. Explains why VGG benefits from BN in modern implementations.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[8] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[9] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013. Influential work on filter design philosophy.

[10] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11976–11986, 2022. Modern analysis of classic architectures including parameter counts.

[11] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[12] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[13] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014. Explains the effectiveness of small 3×3 filters in VGG.

[14] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.