ECE 60146 - Homework 4

Selin Kasap

1 Environment Setup

Experiments were conducted using Google Colab and Gilbreth with DLStudio version 2.5.5 and PyTorch.

1.1 DLStudio Installation

DLStudio was installed from the official distribution provided by the instructor.

Listing 1: DLStudio Installation

```
!wget https://engineering.purdue.edu/kak/distDLS/DLStudio-2.5.5.tar.gz
!tar -xvzf DLStudio-2.5.5.tar.gz

# Needed libraries to import DLStudio
!pip install pymsgbox
!pip install torchmetrics
```

The version was verified as:

Listing 2: DLStudio Version

```
sys.path.insert(0, "/content/DLStudio-2.5.5")
import DLStudio
print("DLStudio version:", DLStudio.__version__)
>>>DLStudio version: 2.5.5
```

1.2 LVIS API and Dataset Setup

The LVIS API was installed to access the LVIS annotation files, which provide category information and instance-level annotations for COCO images. The corresponding COCO 2017 training images were downloaded. The annotation JSON files were extracted into the `lvis/annotations` directory to allow programmatic access through the LVIS API.

Listing 3: LVIS API and COCO Installation

```
!pip install lvis
!mkdir -p lvis/annotations
!unzip lvis_v1_train.json.zip -d lvis/annotations/
!unzip lvis_v1_val.json.zip -d lvis/annotations/

!mkdir -p coco
!cd coco
!wget http://images.cocodataset.org/zips/train2017.zip
!unzip train2017.zip -d coco/
```

The function below extracts images containing multiple instances of the same object category. An image is included in the dataset only if it contains at least two instances of the target category and if the total area occupied by those instances exceeds a predefined ratio of the image area. This ensures that selected images contain sufficiently visible and meaningful object instances rather than very small or insignificant objects.

Listing 4: Multi Same Object Dataset

```python
def create_multi_same_dataset(lvis, category_name,
    category_info, min_instances=2, min_total_area_ratio=0.10, max_images
       =500):
    if category_name not in category_info:
        raise ValueError(f"Category '{category_name}' not found in
           category_info.")
    cat_id = category_info[category_name]["cat_id"]
    img_ids = category_info[category_name]["img_ids"]

    multi_same_data = []

    for img_id in tqdm(img_ids, desc=f"Multi-same: {category_name}"):
        img_info = lvis.load_imgs([img_id])[0]
        img_area = img_info["width"] * img_info["height"]
        ann_ids = lvis.get_ann_ids(img_ids=[img_id])
        anns = lvis.load_anns(ann_ids)

        # Filter for target category
        target_anns = [a for a in anns if a["category_id"] == cat_id]

        # Must have >= min_instances
        if len(target_anns) < min_instances:
            continue

        # Size check
        total_area = sum(float(a.get("area", 0.0)) for a in target_anns)
        if total_area / max(1.0, img_area) < min_total_area_ratio:
            continue

        multi_same_data.append({
            "img_id": img_id,
            "img_info": img_info,
            "annotation": target_anns,
            "category": category_name})

        if len(multi_same_data) >= max_images:
            break

    return multi_same_data
```

After collecting the images for each category using LVIS training dataset, the dataset was split into training and validation sets using an 80/20 ratio. The split is performed before any augmentation to prevent data leakage. Each image and all of its future augmented variants remain strictly within the same split.

Listing 5: Dataset Split 80/20

```
def split_80_20(items_by_class,seed=42):
    rng = random.Random(seed)
    out = {"train": {}, "val": {}}
    for cls, items in items_by_class.items():
        items = list(items)
        rng.shuffle(items)
        n = len(items)
        cut = int(0.8 * n)
        out["train"][cls] = items[:cut]
        out["val"][cls] = items[cut:]
    return out
```

Data augmentation was applied only to the training set as required. The training transform includes resizing to 64×64, random horizontal flipping, small random rotations, and brightness/-contrast jittering. These transformations increase dataset variability and help the model generalize better by exposing it to slightly altered versions of the same images. Finally, the images are converted to tensors and normalized using ImageNet mean and standard deviation values.

Listing 6: Transform for Training Set

```
transform_train = T.Compose([
    T.Resize((64, 64)),
    T.RandomHorizontalFlip(),
    T.RandomRotation(10),
    T.ColorJitter(
        brightness=0.2, contrast=0.2),
    T.ToTensor(),
    T.Normalize(
        (0.485, 0.456, 0.406),
        (0.229 , 0.224 , 0.225))])
```

The validation set uses only resizing and normalization. No random augmentation is applied to validation images in order to maintain a fair and realistic evaluation setting. This ensures that performance metrics reflect true generalization ability rather than artificially modified validation samples.

Listing 7: Transform for Validation Set

```
transform_val = T.Compose([
    T.Resize((64, 64)),
    T.ToTensor(),
    T.Normalize(
        (0.485, 0.456, 0.406),
        (0.229, 0.224, 0.225))])
```

Before training any model, the dataset was verified to ensure it satisfies the assignment requirements.

Listing 8: Dataset Verification

```
from torchvision.datasets import ImageFolder

train_set = ImageFolder ('lvis_datasets/multi_instance_same/train',
                         transform=transform_train)
```

```
val_set = ImageFolder('lvis_datasets/multi_instance_same/val',
                          transform=transform_val)

print("Classes: ", train_set.classes)
for i , cls in enumerate(train_set.classes):
    n_train = sum(1 for _ , l in train_set if l == i)
    n_val = sum(1 for _ , l in val_set if l == i)
    print(f"{cls}: {n_train} train, "
        f"{n_val} val")
```

Listing 9: Dataset Verification Output

```
Classes:  ['banana', 'broccoli', 'person', 'shirt', 'suitcase']
banana: 400 train, 100 val
broccoli: 400 train, 100 val
person: 400 train, 100 val
shirt: 352 train, 89 val
suitcase: 400 train, 100 val
```

The output confirms that all five selected categories meet the minimum image requirements. Each class contains at least 300 training images and at least 50 validation images. Furthermore, no class exceeds 1.2 times the size of the smallest class in the training set, satisfying the balance constraint specified in the homework instructions. Therefore, the dataset is suitable for training the deep skip-connection network in the subsequent tasks.

2 Task 1: The Degradation Problem

2.1 Designing Net3

The architecture of Net3 follows a structured four-block design, where each block consists of two convolutional layers followed by a max-pooling operation. This results in exactly eight convolutional layers. The spatial resolution is reduced progressively from 32×32 to 2×2 through four pooling operations, while the number of feature channels increases from 32 to 256. Batch normalization and ReLU activation are applied after every convolution to stabilize training and improve gradient flow. A fully connected classifier with dropout is used at the end to reduce overfitting.

Listing 10:]Net3 Architecture [1]

```
class Net3(nn.Module):
    def __init__(self):
        super(Net3, self).__init__()
        # -------- Block 1: 32x32 -> 16x16 --------
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1, bias=False
            )
        self.bn1   = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(32, 32, kernel_size=3, padding=1, bias=
            False)
        self.bn2   = nn.BatchNorm2d(32)
        self.pool1 = nn.MaxPool2d(2, 2)
        # -------- Block 2: 16x16 -> 8x8 --------
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, padding=1, bias=
            False)
        self.bn3   = nn.BatchNorm2d(64)
```

```
        self.conv4 = nn.Conv2d(64, 64, kernel_size=3, padding=1, bias=
            False)
        self.bn4   = nn.BatchNorm2d(64)
        self.pool2 = nn.MaxPool2d(2, 2)
        # -------- Block 3: 8x8 -> 4x4 --------
        self.conv5 = nn.Conv2d(64, 128, kernel_size=3, padding=1, bias=
            False)
        self.bn5   = nn.BatchNorm2d(128)
        self.conv6 = nn.Conv2d(128, 128, kernel_size=3, padding=1, bias=
            False)
        self.bn6   = nn.BatchNorm2d(128)
        self.pool3 = nn.MaxPool2d(2, 2)
        # -------- Block 4: 4x4 -> 2x2 --------
        self.conv7 = nn.Conv2d(128, 256, kernel_size=3, padding=1, bias=
            False)
        self.bn7   = nn.BatchNorm2d(256)
        self.conv8 = nn.Conv2d(256, 256, kernel_size=3, padding=1, bias=
            False)
        self.bn8   = nn.BatchNorm2d(256)
        self.pool4 = nn.MaxPool2d(2, 2)
        # After 4 pools: 32->16->8->4->2
        # Feature map: (B, 256, 2, 2) => 256*2*2 = 1024
        self.fc1 = nn.Linear(256 * 2 * 2, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)

        self.relu = nn.ReLU(inplace=True)
        self.dropout = nn.Dropout(p=0.25)

    def forward(self, x):
        # Block 1
        x = self.relu(self.bn1(self.conv1(x)))
        x = self.relu(self.bn2(self.conv2(x)))
        x = self.pool1(x)
        # Block 2
        x = self.relu(self.bn3(self.conv3(x)))
        x = self.relu(self.bn4(self.conv4(x)))
        x = self.pool2(x)
        # Block 3
        x = self.relu(self.bn5(self.conv5(x)))
        x = self.relu(self.bn6(self.conv6(x)))
        x = self.pool3(x)
        # Block 4
        x = self.relu(self.bn7(self.conv7(x)))
        x = self.relu(self.bn8(self.conv8(x)))
        x = self.pool4(x)
        # Flatten
        x = x.view(x.size(0), -1)
        # FC head
        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

```
62
63      def count_parameters(self):
64          return sum(p.numel() for p in self.parameters() if p.requires_grad
                )
```

All models were trained using the DLStudio training infrastructure with identical hyperparameters to ensure a fair comparison. The configuration includes 10 training epochs, a learning rate of 10^{-4}, momentum of 0.9, batch size of 4, and GPU acceleration. Using the same training setup for Net, Net2, and Net3 ensures that performance differences arise primarily from architectural changes rather than training conditions.

Listing 11:]CIFAR-10 example with Net [1]

```
dls = DLStudio(dataroot = "./data/CIFAR-10/",
               image_size = [32,32],
               path_saved_model = "./saved_model",
               momentum = 0.9,
               learning_rate = 1e-4,
               epochs = 10,
               batch_size = 4,
               classes = ('plane','car','bird','cat','deer','dog','frog','
                  horse','ship','truck'),
               use_gpu = True)


exp_cifar = DLStudio.ExperimentsWithCIFAR(dl_studio=dls0)

exp_cifar.load_cifar_10_dataset()

model = exp_cifar.Net() # or exp_cifar.Net2() or Net3()

number_of_learnable_params = sum(p.numel() for p in model.parameters() if
   p.requires_grad)
print("\n\nThe number of learnable parameters in the model: %d" %
   number_of_learnable_params)

exp_cifar.run_code_for_training(model, display_images=False)
exp_cifar.run_code_for_testing(model, display_images=False)
```

NET. The training loss curve for Net shows steady but relatively slow convergence. Although the loss decreases consistently, the overall classification accuracy remains limited. This suggests that the shallow architecture may lack sufficient representational capacity to capture the complexity of CIFAR-10.

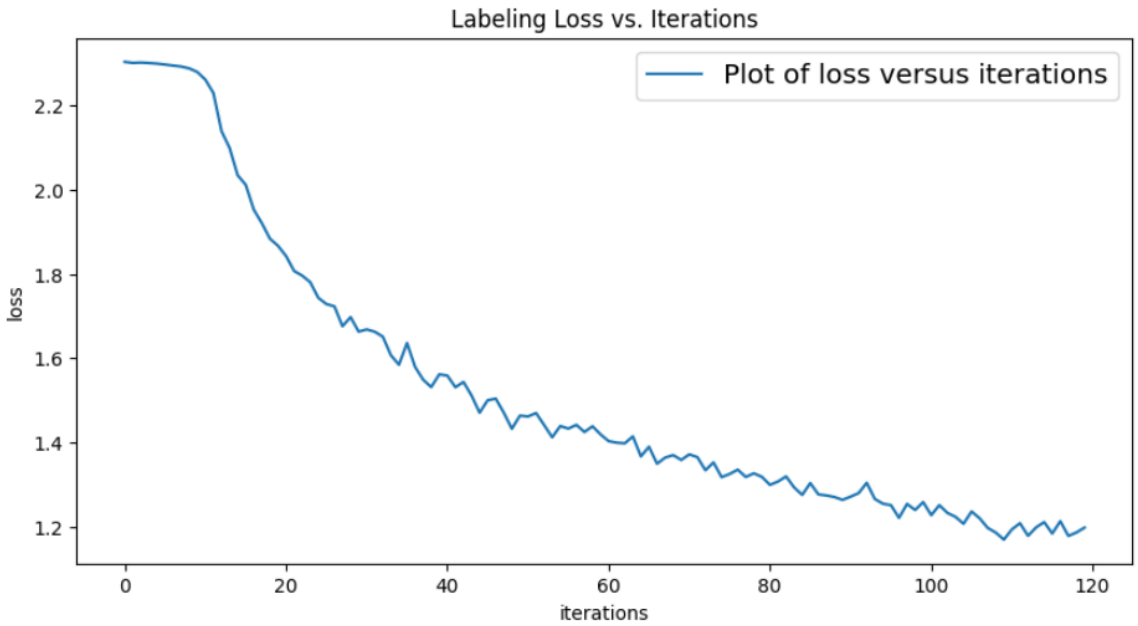

Figure 1: Loss curve of CIFAR-10 example with Net

Listing 12: Output of CIFAR-10 example with Net

```
The number of learnable parameters in the model: 62006
...(training)

Prediction accuracy for plane : 65 %
Prediction accuracy for   car : 66 %
Prediction accuracy for  bird : 48 %
Prediction accuracy for   cat : 44 %
Prediction accuracy for  deer : 35 %
Prediction accuracy for   dog : 40 %
Prediction accuracy for  frog : 76 %
Prediction accuracy for horse : 62 %
Prediction accuracy for  ship : 74 %
Prediction accuracy for truck : 52 %
Overall accuracy of the network on the 10000 test images: 56 %

Displaying the confusion matrix:

       plane   car  bird   cat  deer   dog  frog horse  ship truck

 plane:65.10  2.40  8.60  2.30  1.10  0.50  2.30  1.00 12.80  3.90
   car: 4.70 66.90  1.80  1.40  0.40  0.80  1.50  1.00  8.20 13.30
  bird: 9.30  1.30 48.90  8.50  6.50  6.20 11.30  3.80  3.00  1.20
   cat: 2.60  1.20  9.50 44.40  5.20 11.60 14.90  5.50  2.70  2.40
  deer: 4.70  1.00 17.40  7.20 35.90  5.00 15.30 10.60  2.30  0.60
   dog: 1.20  0.60 12.80 24.60  3.70 40.20  6.00  8.10  1.90  0.90
  frog: 1.50  0.70  6.00  7.50  2.40  0.60 76.30  2.70  1.30  1.00
 horse: 3.80  0.20  4.40  7.10  7.20  8.50  3.00 62.40  0.70  2.70
  ship:10.90  4.80  2.30  2.10  0.40  0.70  1.00  0.50 74.50  2.80
 truck: 7.00 15.30  1.70  3.40  0.70  2.00  4.00  3.80  9.70 52.40
```

Net contains 62,006 trainable parameters and achieves an overall test accuracy of 56%. The per-class accuracies reveal significant variation across categories. Classes such as *frog* and *ship* perform relatively well, while *cat*, *dog*, and *deer* exhibit lower accuracy. The confusion matrix shows frequent confusion among visually similar animal categories.

NET2. Compared to Net, Net2 demonstrates faster convergence and a lower final training loss. The deeper structure increases the model's capacity, allowing it to learn more discriminative features. The loss curve is smoother and decreases more rapidly, indicating improved optimization behavior.

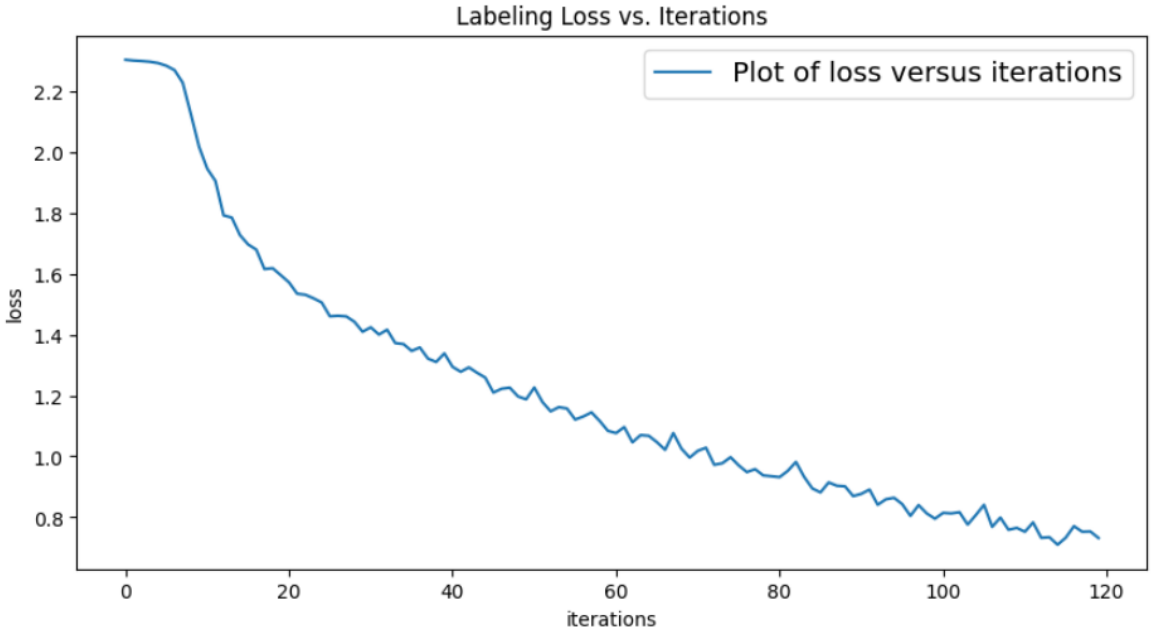

Figure 2: Loss curve of CIFAR-10 example with Net2

Listing 13: Output of CIFAR-10 example with Net2

```
The number of learnable parameters in the model: 1468036

...(training)

Prediction accuracy for plane : 77 %
Prediction accuracy for   car : 80 %
Prediction accuracy for  bird : 45 %
Prediction accuracy for   cat : 62 %
Prediction accuracy for  deer : 64 %
Prediction accuracy for   dog : 63 %
Prediction accuracy for  frog : 77 %
Prediction accuracy for horse : 76 %
Prediction accuracy for  ship : 81 %
Prediction accuracy for truck : 87 %

Overall accuracy of the network on the 10000 test images: 71 %

Displaying the confusion matrix:
```

```
       plane   car   bird   cat   deer   dog   frog  horse   ship  truck

plane:77.70  2.20  1.50  2.40  0.70  0.60  1.00  0.70  6.50  6.70
  car: 1.40 80.10  0.00  0.90  0.10  0.50  0.70  0.10  1.40 14.80
 bird:10.90  0.80 45.10 11.90 10.30  7.40  5.50  4.00  1.40  2.70
  cat: 1.90  1.20  2.10 62.50  5.50 15.40  4.00  2.50  1.40  3.50
 deer: 3.10  0.50  3.70 10.20 64.90  4.00  3.60  7.60  1.50  0.90
  dog: 1.70  0.50  1.70 21.00  3.80 63.80  1.40  4.20  0.80  1.10
 frog: 0.70  0.30  2.70  9.00  4.60  2.50 77.40  0.70  0.70  1.40
horse: 1.50  0.60  1.20  5.60  4.90  6.00  0.30 76.90  0.30  2.70
 ship: 6.20  4.10  0.30  1.50  0.40  0.90  0.40  0.30 81.90  4.00
truck: 1.50  5.70  0.20  1.50  0.50  0.60  0.40  0.40  1.60 87.60
```

Net2 contains 1,468,036 trainable parameters and achieves 71% overall test accuracy. This represents a substantial improvement over Net. Most classes show improved per-class accuracy, particularly *car*, *ship*, and *truck*. However, certain classes such as *bird* still remain relatively challenging, suggesting that additional depth alone does not uniformly improve all categories.

NET3. Net3 exhibits the fastest convergence and the lowest final training loss among the three models. The deeper architecture with eight convolutional layers enables the network to learn richer hierarchical representations. The loss curve shows stable optimization without significant oscillations, indicating that batch normalization effectively mitigates training instability in the deeper network.

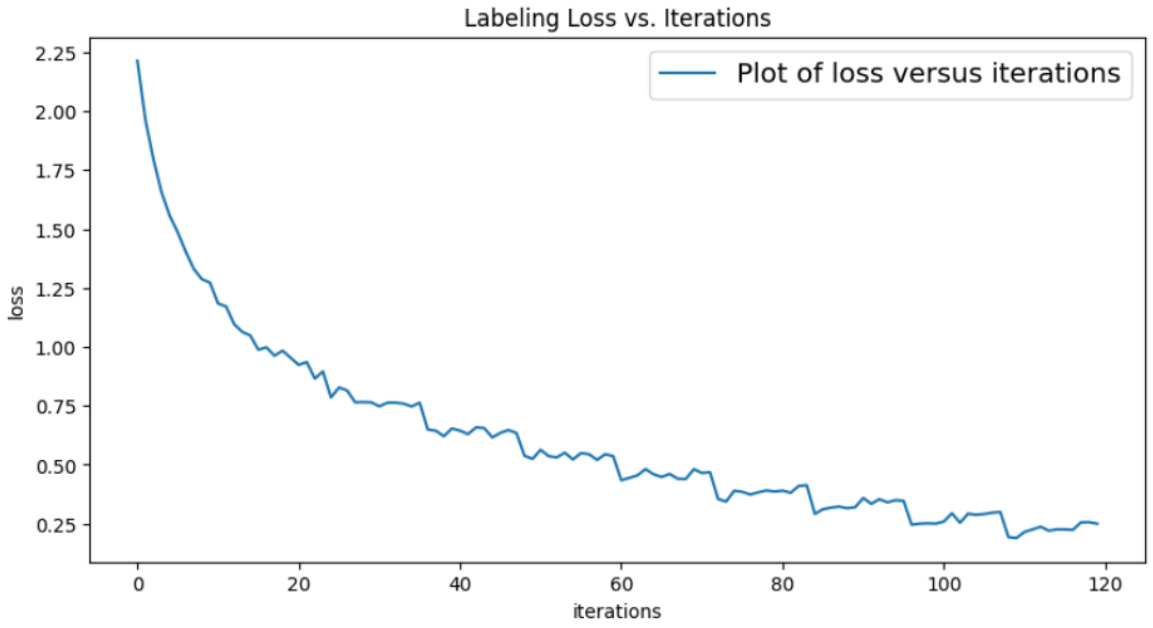

Figure 3: Loss curve of CIFAR-10 example with Net3

Listing 14: Output of CIFAR-10 example with Net3

```
The number of learnable parameters in the model: 1469802
...(training)
```

```
Prediction accuracy for plane : 87 %
Prediction accuracy for   car : 89 %
Prediction accuracy for  bird : 64 %
Prediction accuracy for   cat : 69 %
Prediction accuracy for  deer : 82 %
Prediction accuracy for   dog : 68 %
Prediction accuracy for  frog : 85 %
Prediction accuracy for horse : 82 %
Prediction accuracy for  ship : 89 %
Prediction accuracy for truck : 85 %

Overall accuracy of the network on the 10000 test images: 80 %

Displaying the confusion matrix:

       plane   car  bird   cat  deer   dog  frog horse  ship truck

 plane:87.70  1.10  0.60  1.60  0.50  0.40  0.50  1.30  5.30  1.00
   car: 1.90 89.00  0.00  0.60  0.30  0.10  1.00  0.00  2.50  4.60
  bird: 7.90  0.40 64.70  6.70  7.60  3.80  5.20  2.80  0.70  0.20
   cat: 2.00  0.10  3.20 69.30  6.00  9.50  4.50  2.60  2.20  0.60
  deer: 2.00  0.10  2.60  4.20 82.80  1.60  2.30  3.40  0.90  0.10
   dog: 1.50  0.00  2.20 16.20  4.50 68.40  2.50  3.90  0.70  0.10
  frog: 1.00  0.00  2.30  6.30  3.10  1.00 85.40  0.40  0.50  0.00
 horse: 2.20  0.30  1.10  4.10  5.70  3.10  0.60 82.30  0.30  0.30
  ship: 5.20  1.20  0.60  1.00  0.40  0.30  0.30  0.10 89.80  1.10
 truck: 3.40  4.60  0.30  0.80  0.50  0.30  0.40  0.90  2.90 85.90
```

Net3 contains 1,469,802 trainable parameters and achieves 80% overall test accuracy. This demonstrates a clear improvement over both Net and Net2. Most classes benefit from the deeper architecture, with particularly strong gains in *car*, *plane*, *frog*, and *ship*. The confusion matrix shows reduced inter-class confusion, especially among animal categories, although some overlap between *cat* and *dog* remains.

2.2 Comparative Analysis

Table 1: Overall accuracy comparison.

Network	**Conv Layers**	**Parameters**	**Accuracy (%)**
Net	2	62,006	56.70
Net2	3	1,468,036	71.79
Net3	8	1,469,802	80.53

The results demonstrate that accuracy improves from Net (56%) to Net2 (71%) and further to Net3 (80%), indicating that increased depth enhances representational power for CIFAR-10 classification. This indicates that increasing network depth enhances representational capacity for CIFAR-10 classification when training remains stable. Net3, with eight convolutional layers, is able to extract more hierarchical and abstract features compared to the shallower architectures.

However, the improvement is not strictly monotonic in terms of efficiency: Net2 and Net3 have nearly the same number of parameters, yet Net3 achieves significantly better performance. This

suggests that architectural design and depth organization play a crucial role beyond parameter count alone.

Although deeper networks can sometimes suffer from the degradation problem, where additional layers lead to worse performance, this issue does not appear in my results. The use of batch normalization after every convolution and careful architectural design (conv-conv-pool structure) likely helped maintain stable gradient flow and prevented optimization difficulties.

Table 2: Per-class classification accuracy (%) comparison for CIFAR-10.

Class	**Net**	**Net2**	**Net3**
plane	65	77	87
car	66	80	89
bird	48	45	64
cat	44	62	69
deer	35	64	82
dog	40	63	68
frog	76	77	85
horse	62	76	82
ship	74	81	89
truck	52	87	85

Examining per-class accuracy, the largest improvements occur in visually complex categories such as *deer*, *cat*, and *bird*. These classes require stronger feature extraction to distinguish fine-grained details, which deeper networks can better capture. Classes such as *plane* and *ship*, which already performed relatively well in shallower models, also benefit from additional depth but show smaller relative gains. Some confusion remains between similar animal categories (e.g., *cat* vs. *dog*), indicating that even deeper models still struggle with highly overlapping visual patterns.

Overall, the experiment confirms that increasing depth improves performance in this setting, provided that normalization and architectural design are handled carefully.

3 Task 2: Skip Connections on CIFAR-10

In this task, I studied how skip (residual) connections can reduce the degradation problem when training deeper networks. I first analyzed the `SkipBlock` implementation inside DLStudio's `BMEnet`, and then trained `BMEnet` on CIFAR-10 with `skip_connections=True` and `depth=32`. Finally, I compared the results with the deeper network from Task 1 (Net3) and discussed the effect of skip connections on optimization and class-wise performance.

3.1 Studying the SkipBlock

Architecture of a single `SkipBlock`. A single `SkipBlock` consists of two convolutional layers (`convo1` and `convo2`), each followed by batch normalization (`bn1`, `bn2`) and a ReLU activation. In addition, the block contains a 1×1 convolution (`in2out`) that is used when the input and output channel dimensions do not match. When downsampling is required, the block uses separate 1×1 stride-2 convolutional layers (`downsampler1` and `downsampler2`) to reduce the spatial resolution while keeping the computation consistent between the main path and the identity path.

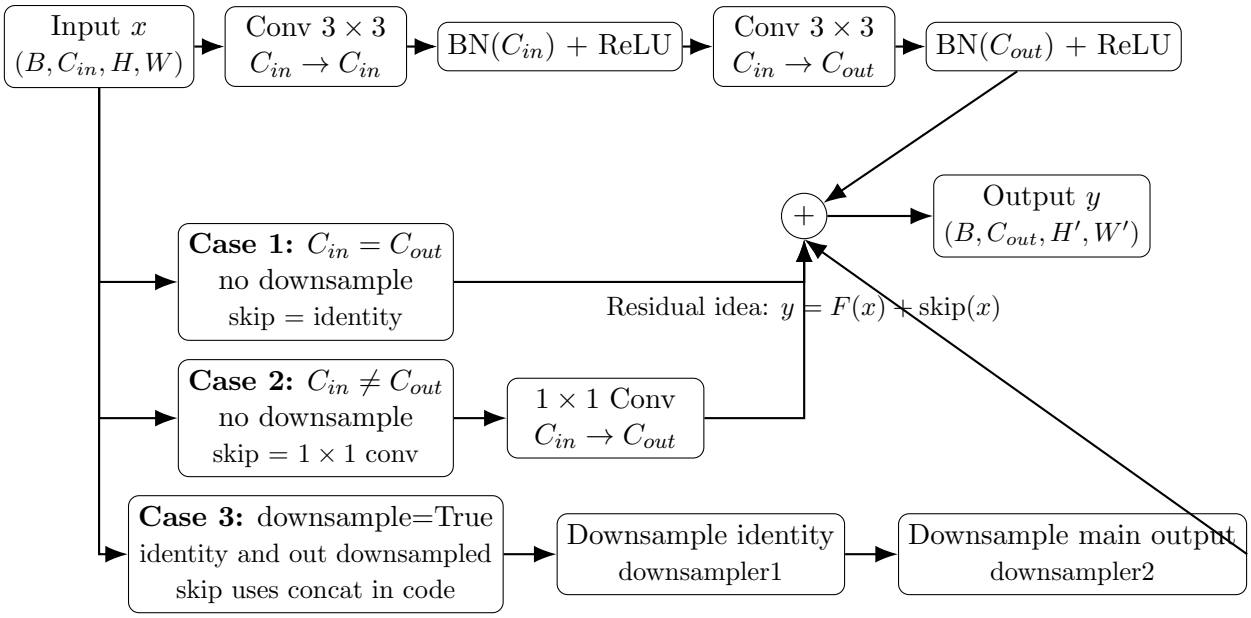

Figure 4: SkipBlock / residual block diagram. The main branch computes $F(x)$ using two Conv-BN-ReLU steps. The skip branch passes x forward (identity), or projects it (1×1 conv), or adjusts shape for downsampling. The final output is the sum: $y = F(x) + \text{skip}(x)$.

How the skip connection works (what is added to what?). In the forward pass, the input tensor x is first stored as `identity`. The main path applies two `Conv`→`BN`→`ReLU` operations to produce `out`. If skip connections are enabled, the block then adds the identity path into the main output (after making sure shapes are compatible). Therefore, the final output is the sum of the transformed features and the identity features:

$$\texttt{out} \leftarrow \texttt{out} + \texttt{identity}.$$

This addition helps preserve useful low-level information and improves gradient flow during back-propagation.

How `SkipBlock` handles channel mismatch. The `SkipBlock` explicitly handles three different situations:

- **Case 1: Same channels, no downsampling.** If `in_ch == out_ch` and `downsample=False`, the identity tensor already matches the shape of `out`, so the block directly computes `out` = `out` + `identity`.
- **Case 2: Different channels, no downsampling.** If `in_ch != out_ch` and `downsample=False`, the block projects the identity tensor to the correct number of channels using the 1×1 convolution `in2out`. Then it adds the projected identity to `out`.
- **Case 3: Different channels with downsampling.** If `in_ch != out_ch` and `downsample=True`, the identity path is first downsampled in space, and then the block uses concatenation (`torch.cat`) to match the required output channel dimension before addition. As stated in the course slides: When the input and the output channels are unequal for a convolutional layer, the number of output channels is exactly twice the number of input channels.

Overall, these cases ensure that the residual addition is always dimensionally valid.

Relation to the residual learning formulation $F(x) + x$. The design of `SkipBlock` matches the residual learning idea:

$$y = F(x) + x.$$

Here, $F(x)$ corresponds to the main convolutional path (two conv layers with BN and ReLU), and x corresponds to the identity shortcut path. If shapes do not match, the shortcut path is adjusted (projection, downsampling, or concatenation) so that the addition remains valid. This residual form makes learning easier because the network can focus on learning the *difference* from the identity mapping instead of learning the entire mapping from scratch.

3.2 Running BMEnet on CIFAR-10

After understanding the structure of `SkipBlock`, I trained `BMEnet` using the provided DLStudio training pipeline. I used `skip_connections=True` and `depth=32`, and I kept the same hyperparameters.

Listing 15:]CIFAR-10 Example with BME and SkipBlock [1]

```
dls = DLStudio(dataroot = "./data/CIFAR-10/",
               image_size = [32,32],
               path_saved_model = "./saved_model",
               momentum = 0.9,
               learning_rate = 1e-4,
               epochs = 10,
               batch_size = 4,
               classes = ('plane','car','bird','cat','deer','dog','frog','
                  horse','ship','truck'),
               use_gpu = True)
bme_net = dls.BMEnet(dls, skip_connections=True, depth=32)
bme_net.load_cifar_10_dataset()

number_of_learnable_params = sum(p.numel() for p in bme_net.parameters()
   if p.requires_grad)
print("\n\nThe number of learnable parameters in the model: %d" %
   number_of_learnable_params)

bme_net.run_code_for_training(bme_net, display_images=False)
bme_net.run_code_for_testing(bme_net, display_images=False)
```

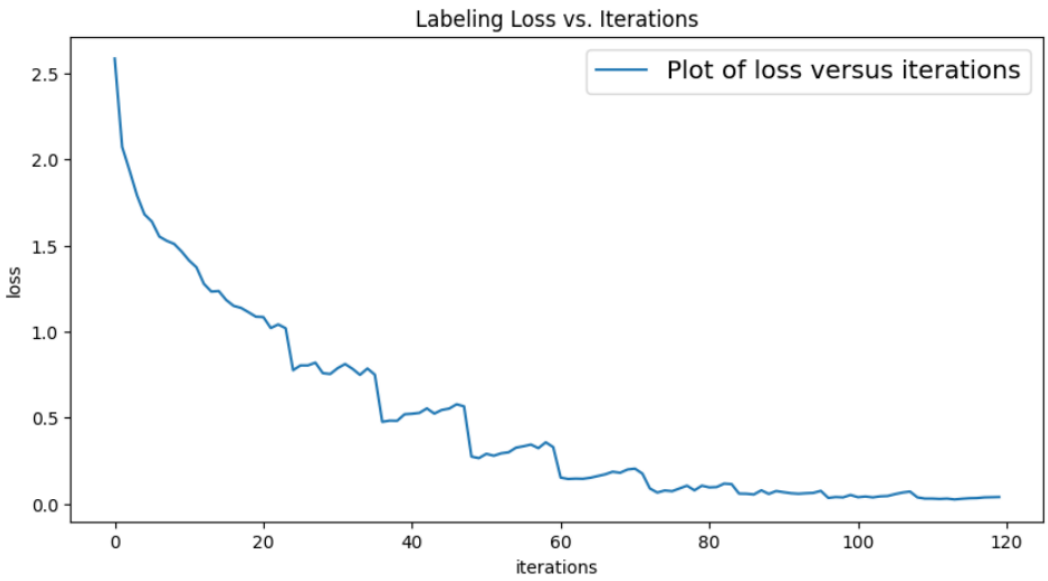

Figure 5: Loss curve of CIFAR-10 example with BMEnet and SkipBlock

Listing 16: Output of CIFAR-10 example with BMEnet and SkipBlock

```
The number of learnable parameters in the model: 69493890
...(training)

Prediction accuracy for plane : 79 %
Prediction accuracy for   car : 83 %
Prediction accuracy for  bird : 66 %
Prediction accuracy for   cat : 57 %
Prediction accuracy for  deer : 67 %
Prediction accuracy for   dog : 69 %
Prediction accuracy for  frog : 85 %
Prediction accuracy for horse : 84 %
Prediction accuracy for  ship : 90 %
Prediction accuracy for truck : 88 %
Overall accuracy of the network on the 10000 test images: 77 %

Displaying the confusion matrix:

       plane   car  bird   cat  deer   dog  frog horse  ship truck

 plane:79.20  0.60  4.00  2.70  0.80  0.30  0.60  0.80  7.10  3.90
   car: 1.20 83.10  0.30  0.70  0.40  0.40  0.80  0.20  1.90 11.00
  bird: 9.00  0.30 66.70  5.20  4.70  4.80  3.90  2.60  1.30  1.50
   cat: 2.60  0.60  5.00 57.80  5.20 15.60  5.80  4.10  1.20  2.10
  deer: 2.80  0.20  8.40  3.80 67.20  3.50  6.20  6.30  0.80  0.80
   dog: 1.30  0.40  3.80 13.40  2.40 69.40  2.90  4.70  0.30  1.40
  frog: 1.30  0.20  3.40  4.30  1.60  2.10 85.10  0.70  0.80  0.50
 horse: 1.80  0.00  1.40  3.10  2.50  4.40  0.70 84.90  0.20  1.00
  ship: 2.70  1.20  0.60  1.30  0.00  0.80  0.30  0.20 90.10  2.80
 truck: 2.20  2.80  0.60  1.40  0.40  0.60  0.60  0.60  2.20 88.60
```

3.3 BMEnet vs Deeper Networks Discussion

Table 3: Network comparison.

Network	Conv Layers / Depth	Parameters	Accuracy (%)
Net	2 conv layers	62,006	56.70
Net2	3 conv layers	1,468,036	71.79
Net3	8 conv layers	1,469,802	80.53
BMEnet	299 conv layers (depth=32)	69,493,890	77.21

Counting the number of `Conv2d` layers in BMEnet (depth=32). The BMEnet architecture starts with one initial convolution layer `self.conv`. Then it creates three arrays of `SkipBlock`s with length equal to the depth (here, $d = 32$), and two additional downsampling `SkipBlock`s.

Each `SkipBlock` instantiates two 3×3 convolutions (`convo1`, `convo2`) and also a 1×1 projection convolution (`in2out`). Therefore, a non-downsampling `SkipBlock` contains:

$$N_{\text{conv}}(\text{SkipBlock}, \text{no ds}) = 2 + 1 = 3.$$

If `downsample=True`, the block additionally instantiates two 1×1 stride-2 convolutions (`downsampler1`, `downsampler2`), giving:

$$N_{\text{conv}}(\text{SkipBlock}, \text{ds}) = 2 + 1 + 2 = 5.$$

For $d = 32$, the total number of `Conv2d` layers is:

$$\begin{aligned} N_{\text{total}} &= 1 \text{ (initial conv)} \\ &\quad + d \cdot 3 \text{ (}\texttt{skip64\_arr}\text{)} + 1 \cdot 5 \text{ (}\texttt{skip64to128ds}\text{)} \\ &\quad + d \cdot 3 \text{ (}\texttt{skip128\_arr}\text{)} + 1 \cdot 5 \text{ (}\texttt{skip128to256ds}\text{)} \\ &\quad + d \cdot 3 \text{ (}\texttt{skip256\_arr}\text{)} \\ &= 1 + 32 \cdot 3 + 5 + 32 \cdot 3 + 5 + 32 \cdot 3 \\ &= 1 + 96 + 5 + 96 + 5 + 96 \\ &= 299. \end{aligned}$$

Thus, BMEnet with depth 32 instantiates **299** `nn.Conv2d` layers.

Table 4: Per-class classification accuracy (%) for CIFAR-10.

Class	Net	Net2	Net3	BMEnet
plane	65	77	87	79
car	66	80	89	83
bird	48	45	64	66
cat	44	62	69	57
deer	35	64	82	67
dog	40	63	68	69
frog	76	77	85	85
horse	62	76	82	84
ship	74	81	89	90
truck	52	87	85	88

BMEnet vs. Net3 (degradation question). From my results, BMEnet does **not** outperform Net3. Net3 reaches **80%** overall accuracy, while BMEnet reaches **77%**. This means that, in my setup, the deeper plain network (Net3) still gave the best final accuracy even though deeper models can suffer from degradation. However, BMEnet remains competitive and shows that adding depth is more stable when skip connections are used.

How skip connections help. Skip connections add the input (identity) back to the transformed output, which matches the residual learning idea $y = F(x) + x$. This helps gradients flow through the network more easily during backpropagation. As a result, very deep models are less likely to suffer from vanishing gradients or unstable optimization. In practice, skip connections make training deeper architectures easier because each block can learn a small "correction" to the identity mapping instead of learning the entire transformation from scratch.

Which classes improve the most with BMEnet. The largest improvements with `BMEnet` typically appear in visually similar CIFAR-10 categories where the baseline models confuse classes more often (for example, `cat` vs. `dog`, or `truck` vs. `car`). These categories benefit from better feature reuse across layers, because residual connections allow deeper blocks to refine features instead of repeatedly re-learning them. Therefore, skip connections act as a practical solution to train deeper networks effectively and reduce the degradation problem observed in plain deep CNNs.

4 Task 3: Normalization Experiments

In this task, I compared four normalization strategies in the same residual-style architecture: Batch Normalization (BN), Instance Normalization (IN), Layer Normalization (LN), and Group Normalization (GN). The main goal was to keep the network structure and training setup the same while changing only the normalization layers, so I could observe how normalization affects convergence speed and final classification accuracy on CIFAR-10.

4.1 Implementing NormSkipBlock

Listing 17:]NormSkipBlock [1]

```
class NormSkipBlock(nn.Module):
    def __init__(self, in_ch, out_ch, norm_type='batch', num_groups=8,
                 downsample=False, skip_connections=True):
        super().__init__()
        self.downsample = downsample
        self.skip_connections = skip_connections
        self.in_ch = in_ch
        self.out_ch = out_ch
        self.norm_type = norm_type
        self.num_groups = num_groups

        self.convo1 = nn.Conv2d(in_ch, in_ch, 3, stride=1, padding=1)
        self.convo2 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)

        self.norm1 = self._make_norm(norm_type, in_ch, num_groups=
            num_groups)
        self.norm2 = self._make_norm(norm_type, out_ch, num_groups=
            num_groups)
```

```
        self.in2out = nn.Conv2d(in_ch, out_ch, 1)

        if downsample:
            self.downsampler1 = nn.Conv2d(in_ch, in_ch, 1, stride=2)
            self.downsampler2 = nn.Conv2d(out_ch, out_ch, 1, stride=2)

    @staticmethod
    def _make_norm(norm_type, num_channels, num_groups=8):
        norm_type = norm_type.lower()

        if norm_type == 'batch':
            return nn.BatchNorm2d(num_channels)

        elif norm_type == 'instance':
            return nn.InstanceNorm2d(num_channels, affine=True)

        elif norm_type == 'layer':
            return nn.GroupNorm(1, num_channels)

        elif norm_type == 'group':
            g = min(num_groups, num_channels)
            while num_channels % g != 0:
                g -= 1
            return nn.GroupNorm(g, num_channels)

        else:
            raise ValueError(f"Unknown norm_type: {norm_type}")

    def forward(self, x):
        identity = x
        out = self.convo1(x)
        out = self.norm1(out)
        out = nn.functional.relu(out)
        out = self.convo2(out)
        out = self.norm2(out)
        out = nn.functional.relu(out)

        if self.downsample:
            identity = self.downsampler1(identity)
            out = self.downsampler2(out)

        if self.skip_connections:
            if (self.in_ch == self.out_ch) and (self.downsample is False):
                out = out + identity
            elif (self.in_ch != self.out_ch) and (self.downsample is False
                ):
                identity = self.in2out(identity)
                out = out + identity
            elif (self.in_ch != self.out_ch) and (self.downsample is True)
                :
                out = out + torch.cat((identity, identity), dim=1)

        return out
```

All four methods normalize intermediate activations before applying ReLU, but they differ in **which dimensions** they use to compute the mean and standard deviation. BN computes statistics across the batch for each channel, IN computes statistics per image (instance) for each channel, LN computes statistics per instance across all channels, and GN computes statistics per instance over groups of channels [2].

Using `nn.LayerNorm` directly in convolutional networks is inconvenient because it requires fixed spatial dimensions at construction time. Instead, I implemented Layer Normalization with the equivalence:

$$\text{LN over } (C, H, W) \approx \texttt{GroupNorm}(1, C)$$

This normalizes all channels together per instance and behaves like layer normalization for CNN feature maps.

4.2 Building NormBMEnet

Listing 18:]NormBMEnet [1]

```
class NormBMEnet(nn.Module):
    def __init__(self, norm_type='batch', depth=8, skip_connections=True,
        num_groups=8):
        super().__init__()
        self.norm_type = norm_type
        self.depth = depth
        self.skip_connections = skip_connections
        self.num_groups = num_groups

        self.conv = nn.Conv2d(3, 64, 3, padding=1)
        self.relu = nn.ReLU(inplace=True)

        blocks = []
        # depth x (64->64)
        for _ in range(depth):
            blocks.append(NormSkipBlock(64, 64, norm_type=norm_type,
                num_groups=num_groups, downsample=False, skip_connections=
                skip_connections))
        # downsample to 128
        blocks.append(NormSkipBlock(64, 128, norm_type=norm_type,
            num_groups=num_groups, downsample=True, skip_connections=
            skip_connections))
        # depth x (128->128)
        for _ in range(depth):
            blocks.append(NormSkipBlock(128, 128, norm_type=norm_type,
                num_groups=num_groups, downsample=False, skip_connections=
                skip_connections))
        # downsample to 256
        blocks.append(NormSkipBlock(128, 256, norm_type=norm_type,
            num_groups=num_groups, downsample=True, skip_connections=
            skip_connections))
        # depth x (256->256)
        for _ in range(depth):
            blocks.append(NormSkipBlock(256, 256, norm_type=norm_type,
                num_groups=num_groups, downsample=False, skip_connections=
                skip_connections))
```

```
        self.blocks = nn.Sequential(*blocks) # "*" unpacks list
        # after two downsample=True blocks, spatial size goes 32->16->8
        self.fc1 = nn.Linear(256 * 8 * 8, 1000)
        self.fc2 = nn.Linear(1000, 10)

    def forward(self, x):
        x = self.relu(self.conv(x))
        x = self.blocks(x)
        x = x.view(x.size(0), -1)
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

    def count_parameters(self):
        return sum(p.numel() for p in self.parameters() if p.requires_grad
            )
```

4.3 Training and Comparison

Listing 19: Training Pipeline with Normalization

```
transform = T.Compose([T.ToTensor(), T.Normalize((0.5, 0.5, 0.5), (0.5,
    0.5, 0.5))])

def run_norm_experiments(
    norm_types=("batch", "instance", "layer", "group"),
    lr=1e-3,
    epochs=10,
    batch_size=64,
    depth=8,
    skip_connections=True,
    num_groups=8,
    out_dir="./norm_results"):
    os.makedirs(out_dir, exist_ok=True)

    # CIFAR-10 loaders
    train_ds = datasets.CIFAR10(root="./data", train=True, download=True,
        transform=transform)
    test_ds  = datasets.CIFAR10(root="./data", train=False, download=True,
        transform=transform)

    train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=
        True, num_workers=2, pin_memory=True)
    test_loader  = DataLoader(test_ds, batch_size=batch_size, shuffle=
        False, num_workers=2, pin_memory=True)

    class_names = train_ds.classes  # ['airplane','automobile',...]
    summary_rows = []

    for norm_type in norm_types:
        print("\n" + "="*90)
        print(f"Training NormBMEnet | norm_type={norm_type} | depth={depth
            } | skip={skip_connections}")
```

```
        print("="*90)

        model = NormBMEnet(
            norm_type=norm_type,
            depth=depth,
            skip_connections=skip_connections,
            num_groups=num_groups).to(device)

        params = model.count_parameters()
        print(f"Trainable parameters: {params}")

        criterion = nn.CrossEntropyLoss()
        optimizer = torch.optim.Adam(model.parameters(), lr=lr)

        train_losses, train_accs = [], []
        test_accs, epoch_times = [], []

        for ep in range(1, epochs + 1):
            print(f"Epoch {ep}/{epochs}")
            print("-"*90)
            t0 = time.time()

            tr_loss, tr_acc = train.train_one_epoch(model, train_loader,
                criterion, optimizer, device)
            vall_loss, val_acc = train.validate(model, test_loader,
                criterion, device=device)

            dt = time.time() - t0

            train_losses.append(tr_loss)
            train_accs.append(tr_acc)
            epoch_times.append(dt)

        # Final evaluation artifacts
        final_test_loss, final_test_acc = train.validate(model,
            test_loader, criterion, device=device)


run_norm_experiments(
    norm_types=("batch", "instance", "layer", "group"),
    lr=1e-3,
    epochs=10,
    batch_size=64,
    depth=8,
    skip_connections=True,
    num_groups=8,
    out_dir="./norm_results")
```

Figure 6 overlays the training loss curves for BN, IN, LN, and GN. This makes it easy to compare convergence speed and stability across the four normalizations.

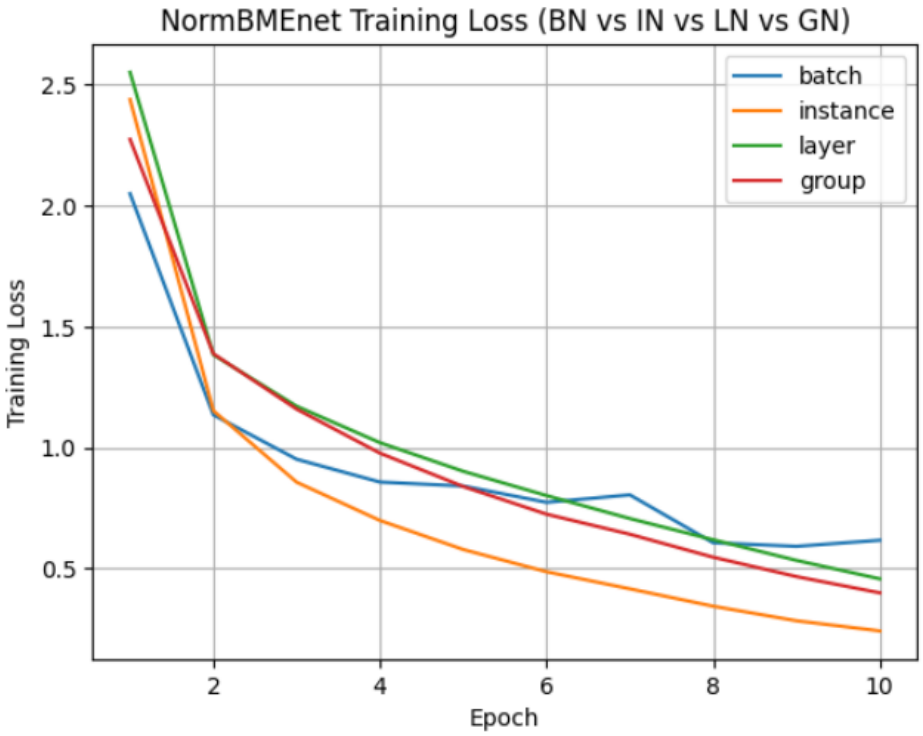

Figure 6: NormBMEnet training loss comparison (BN vs IN vs LN vs GN), depth=8, skip connections enabled.

Table 5 summarizes the final validation accuracy and the number of learnable parameters for each normalization run. Since the architecture is the same (only the normalization layer changes), the parameter count stayed the same across all runs.

Table 5: NormBMEnet normalization comparison summary (depth=8, skip=True, epochs=10).

Norm Type	**Parameters**	**Final Val Acc (%)**
Batch Norm (BN)	30,195,266	74.85
Instance Norm (IN)	30,195,266	77.88
Layer Norm (LN)	30,195,266	74.57
Group Norm (GN, $G = 8$)	30,195,266	73.53

From the overlaid loss curves, **Instance Normalization converged the fastest** in terms of training loss and also achieved the **best final validation accuracy** (77.88%). BatchNorm and LayerNorm finished with very similar validation accuracy (74.85% vs 74.57%), while GroupNorm was slightly lower (73.53%). I also observed some instability in the BN run: for example, the validation accuracy dropped sharply at epoch 4 and then recovered later. This kind of fluctuation can happen when batch statistics vary a lot across iterations (especially if batches are not large or if the optimization path is noisy), even if training loss keeps improving.

InstanceNorm behaved differently from BatchNorm because IN normalizes each sample independently (per instance) instead of using batch-wide statistics. This removes dependency on batch composition, which can make training more consistent in some settings. However, IN can also remove contrast/style information that may be useful for classification; in this experiment it still

performed best, so it seems the benefit of stable normalization outweighed that drawback.

Finally, **GroupNorm can be seen as a compromise** between IN and LN: it normalizes per instance like LN/IN but over groups of channels (here $G = 8$), which can preserve more channel structure than IN while still avoiding batch dependence. In my runs, GN did not outperform IN, but it stayed reasonably stable and achieved a competitive accuracy compared to BN/LN.

5 Task 4: Skip Connection Network for LVIS

5.1 Network Architecture: LVISSkipNet

I implemented LVISSkipNet using NormSkipBlock with batch normalization. I used an initial 3×3 convolution to map RGB to 64 channels, followed by multiple skip blocks arranged in stages with downsampling transitions. Before the final classifier, I used AdaptiveAvgPool2d$(1, 1)$ so the final fully connected layer receives a fixed 256-dimensional vector.

Listing 20:]LVISSkipNet Architecture [1]

```
class LVISSkipNet(nn.Module):
    def __init__(self, num_classes=5, norm_type="batch", num_groups=8,
        skip_connections=True):
        super().__init__()

        self.conv_initial = nn.Conv2d(3, 64, kernel_size=3, padding=1,
            bias=False)
        self.bn_initial = nn.BatchNorm2d(64) if norm_type == "batch" else
            nn.Identity()

        blocks = []

        # Stage 1: 64 channels, no downsample
        for _ in range(3):
            blocks.append(NormSkipBlock(64, 64, norm_type=norm_type,
                num_groups=num_groups, downsample=False, skip_connections=
                skip_connections))
        # Transition: 64 -> 128, downsample=True (64x64 -> 32x32)
        blocks.append(NormSkipBlock(64, 128, norm_type=norm_type,
            num_groups=num_groups, downsample=True, skip_connections=
            skip_connections))
        # Stage 2: 128 channels
        for _ in range(3):
            blocks.append(NormSkipBlock(128, 128, norm_type=norm_type,
                num_groups=num_groups, downsample=False, skip_connections=
                skip_connections))
        # Transition: 128 -> 256, downsample=True (32x32 -> 16x16)
        blocks.append(NormSkipBlock(128, 256, norm_type=norm_type,
            num_groups=num_groups, downsample=True, skip_connections=
            skip_connections))
        # Stage 3: 256 channels
        for _ in range(3):
            blocks.append(NormSkipBlock(256, 256, norm_type=norm_type,
                num_groups=num_groups, downsample=False, skip_connections=
                skip_connections))
        # Transition: 256 -> 256, downsample=True (16x16 -> 8x8)
```

```
        blocks.append(NormSkipBlock(256, 256, norm_type=norm_type,
            num_groups=num_groups, downsample=True, skip_connections=
            skip_connections))
        # Stage 4: 256 channels (still 8x8)
        for _ in range(3):
            blocks.append(NormSkipBlock(256, 256, norm_type=norm_type,
                num_groups=num_groups, downsample=False, skip_connections=
                skip_connections))

        self.blocks = nn.Sequential(*blocks)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(256, num_classes)

    def forward(self, x):
        x = F.relu(self.bn_initial(self.conv_initial(x)))
        x = self.blocks(x)
        x = self.avgpool(x)                    # (B,256,1,1)
        x = x.view(x.size(0), -1)              # (B,256)
        x = self.fc(x)                         # (B,5)
        return x

    def count_parameters(self):
        return sum(p.numel() for p in self.parameters() if p.requires_grad
            )

    def count_layers(self):
        return sum(1 for m in self.modules() if isinstance(m, (nn.Conv2d,
            nn.Linear)))
```

Listing 21: LVISSkipNet Learnable Parameters and Layers

```
Learnable parameters: 10733957
Learnable layers (Conv2d+Linear): 53
```

5.1.1 Channel Progression and Downsampling Strategy

The LVISSkipNet architecture follows a staged design in which spatial resolution is gradually reduced while channel depth is increased. This strategy allows the network to first capture low-level spatial details and then progressively learn more abstract and high-level semantic representations.

Initial Convolution

The network begins with an initial convolution layer:

$$\texttt{Conv2d}(3, 64, \texttt{kernel\_size} = 3, \texttt{padding} = 1)$$

This layer maps the RGB input $(B, 3, 64, 64)$ to a 64-channel feature map of size $(B, 64, 64, 64)$. Padding is used to preserve spatial resolution.

Stage 1: 64 Channels (No Downsampling)

The first stage consists of SkipBlocks that maintain:

$$\text{Channels: } 64 \to 64, \quad \text{Resolution: } 64 \times 64$$

No spatial downsampling is applied in this stage. The goal is to learn low-level and mid-level features (edges, textures, simple shapes) while preserving spatial detail.

Transition 1: 64 → 128 Channels

A transition SkipBlock performs:

$$\text{Channels: } 64 \to 128$$

$$\text{Resolution: } 64 \times 64 \to 32 \times 32$$

Downsampling is performed using stride-based convolution inside the block. This halves the spatial resolution while doubling the number of channels. The increase in channels compensates for the loss of spatial information by allowing richer feature representations.

Stage 2: 128 Channels

Several SkipBlocks operate at:

$$(B, 128, 32, 32)$$

Here, the network learns more complex feature combinations. Because resolution is smaller, receptive fields effectively cover larger portions of the input image.

Transition 2: 128 → 256 Channels

Another downsampling transition performs:

$$\text{Channels: } 128 \to 256$$

$$\text{Resolution: } 32 \times 32 \to 16 \times 16$$

Again, spatial resolution is halved while channel depth is doubled. This continues the standard deep CNN design principle of trading spatial resolution for representational depth.

Stage 3: 256 Channels

At this stage, feature maps have size:

$$(B, 256, 16, 16)$$

The network now encodes higher-level object structures rather than local textures.

Final Downsampling (16×16 → 8×8)

A final downsampling block reduces spatial size to:

$$(B, 256, 8, 8)$$

Unlike earlier transitions, the number of channels remains constant (256). At this depth, increasing spatial abstraction is more important than increasing channel width.

Global Pooling and Classification

Before classification, Adaptive Average Pooling is applied:

$$\texttt{AdaptiveAvgPool2d}((1, 1))$$

This converts the feature map:

$$(B, 256, 8, 8) \to (B, 256, 1, 1)$$

After flattening:

$$(B, 256)$$

Finally, a fully connected layer produces the class logits:

$$(B, 256) \to (B, 5)$$

Design Rationale

The overall channel progression is:

$$64 \to 128 \to 256$$

while spatial resolution follows:

$$64 \times 64 \to 32 \times 32 \to 16 \times 16 \to 8 \times 8$$

This design follows standard deep convolutional network principles:

- Early layers preserve spatial resolution to capture fine-grained details.
- Deeper layers reduce spatial size to increase receptive field.
- Channel depth increases as spatial resolution decreases.
- Skip connections enable stable gradient flow and improve training of deeper models.

This staged progression allows LVISSkipNet to learn hierarchical feature representations while maintaining stable optimization through residual connections.

5.2 Training on LVIS Multi-Instance Same-Object Dataset

Dataset verification is given in section 1.2 of this report.

Listing 22: Data Loading Function

```
def make_loaders(data_root="lvis_dataset/multi_instance_same",
                 batch_size=32, num_workers=2):
    train_dir = os.path.join(data_root, "train")
    val_dir = os.path.join(data_root, "val")

    train_set = ImageFolder(train_dir, transform=transform_train)
    val_set = ImageFolder(val_dir, transform=transform_val)

    train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=
       True, num_workers=num_workers, pin_memory=True)
    val_loader = DataLoader(val_set, batch_size=batch_size, shuffle=False,
        num_workers=num_workers, pin_memory=True)

    return train_set, val_set, train_loader, val_loader
```

Listing 23: LVISSkipNet Learnable Parameters and Layers

```
def train_net(model,
              data_root="lvis_dataset/multi_instance_same",
              out_dir="./lvis_results",
              epochs=10,
              batch_size=32,
              lr=1e-3,
              betas=(0.9, 0.99),
              num_workers=0,
              seed=42,
              model_name="LVISSkipNet")
    os.makedirs(out_dir, exist_ok=True)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = model.to(device)

    train_set, val_set, train_loader, val_loader = make_loaders(
        data_root=data_root,
        batch_size=batch_size,
        num_workers=num_workers)

    class_names = train_set.classes
    num_classes = len(class_names)

    # sanity check output dims
    with torch.no_grad():
        x0, _ = next(iter(train_loader))
        y0 = model(x0.to(device))
        assert y0.shape[1] == num_classes, f"Model outputs {y0.shape[1]}
           classes, but dataset has {num_classes}"

    optimizer = torch.optim.Adam(model.parameters(), lr=lr, betas=betas)
    class_counts = [400, 400, 400, 352, 400]
    weights = 1.0 / torch.tensor(class_counts, dtype=torch.float)
```

```
    weights = weights / weights.sum()

    criterion = nn.CrossEntropyLoss(weight=weights.to(device))

    history = {
        "dataset_root": data_root,
        "classes": class_names,
        "epochs": epochs,
        "batch_size": batch_size,
        "lr": lr,
        "betas": list(betas),
        "device": str(device),
        "trainable_params": int(sum(p.numel() for p in model.parameters()
            if p.requires_grad)),
        "learnable_layers": int(model.count_layers()) if hasattr(model, "
            count_layers") else None,
        "train": {"loss": [], "acc": []},
        "val": {"loss": [], "acc": []},
        "time_per_epoch_sec": [],
        "final": {}
    }

    best_val_acc = -1.0
    best_path = os.path.join(out_dir, "best_model.pt")

    for ep in range(1, epochs + 1):
        print(f"Epoch {ep}/{epochs}")
        print("-"*90)

        t0 = time.time()

        train_loss, train_acc = train.train_one_epoch(model, train_loader,
             criterion, optimizer, device)
        val_loss, val_acc = train.validate(model, val_loader, criterion,
            device=device)

        elapsed = time.time() - t0
        print(f"Elapsed time: {elapsed}")
        history["train"]["loss"].append(float(train_loss))
        history["train"]["acc"].append(float(train_acc))
        history["val"]["loss"].append(float(val_loss))
        history["val"]["acc"].append(float(val_acc))
        history["time_per_epoch_sec"].append(float(elapsed))

        if val_acc > best_val_acc:
            best_val_acc = val_acc
            torch.save(model.state_dict(), best_path)

    # Load best model and compute final val metrics
    model.load_state_dict(torch.load(best_path, map_location=device))
    final_val_loss, final_val_acc = train.validate(model, val_loader,
        criterion, device)
    final_cm = evaluate.compute_confusion_matrix(model, val_loader,
        num_classes, device)
```

```
    per_class_acc = evaluate.per_class_accuracy(final_cm)
    plot_confusion_matrix(final_cm, class_names,
                          title=f"{model_name} Confusion Matrix")

    history["final"] = {
        "best_model_path": best_path,
        "best_val_acc": float(best_val_acc),
        "val_confusion_matrix": final_cm.tolist(),
        "val_per_class_acc": per_class_acc.tolist(),
        "val_loss": float(final_val_loss),
        "val_acc": float(final_val_acc)}

    # Save JSON
    json_path = os.path.join(out_dir, "lvis_skipnet_metrics.json")
    with open(json_path, "w") as f:
        json.dump(history, f, indent=2)

    print(f"\nSaved best model: {best_path}")
    print(f"Saved metrics JSON: {json_path}")

    return history
```

Listing 24: LVISSkipNet Training Hyperparameters

```
model = LVISSkipNet(
    num_classes=5,
    norm_type="batch",
    num_groups=8,
    skip_connections=True)

# verify learnable layers
print("Learnable layers (Conv2d+Linear):", model.count_layers())
assert model.count_layers() >= 40, "count_layers() must be >= 40"

lvis_skipnet = train_net(
    model=model,
    data_root="lvis_dataset/multi_instance_same",
    out_dir="./lvis_results",
    epochs=10,
    batch_size=32,
    lr=1e-3,
    betas=(0.9, 0.99),
    num_workers=0,
    seed=42)
```

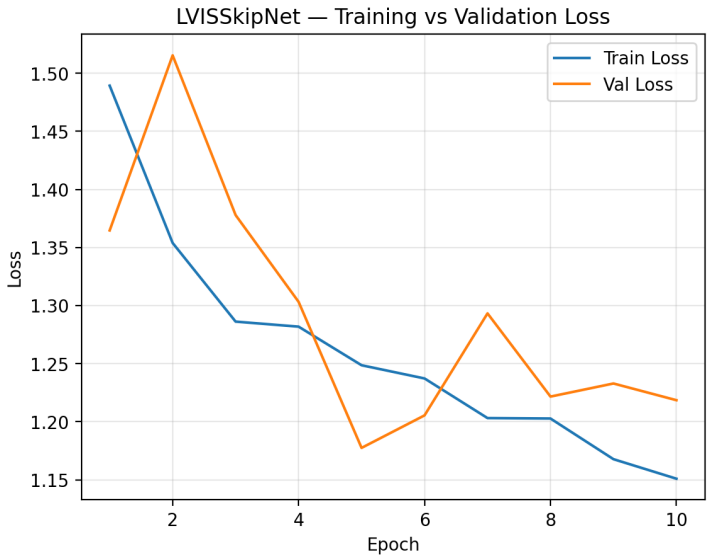

Figure 7: Training and validation loss curves for LVISSkipNet (10 epochs).

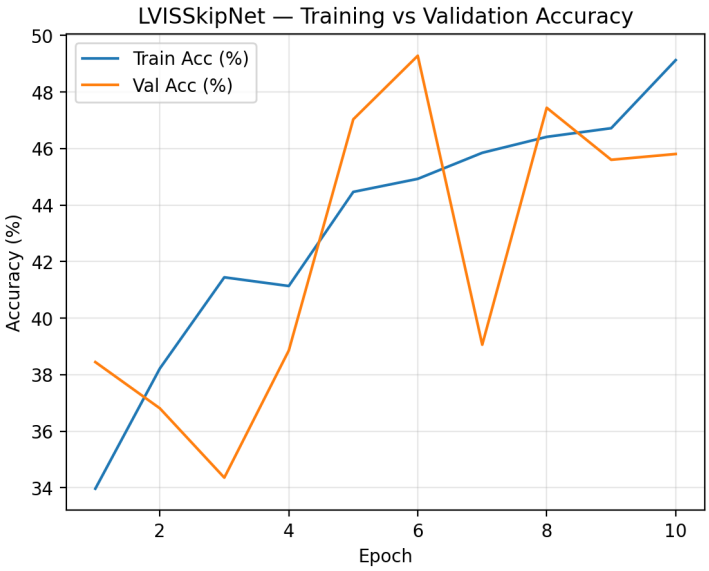

Figure 8: Training and validation accuracy curves for LVISSkipNet (10 epochs).

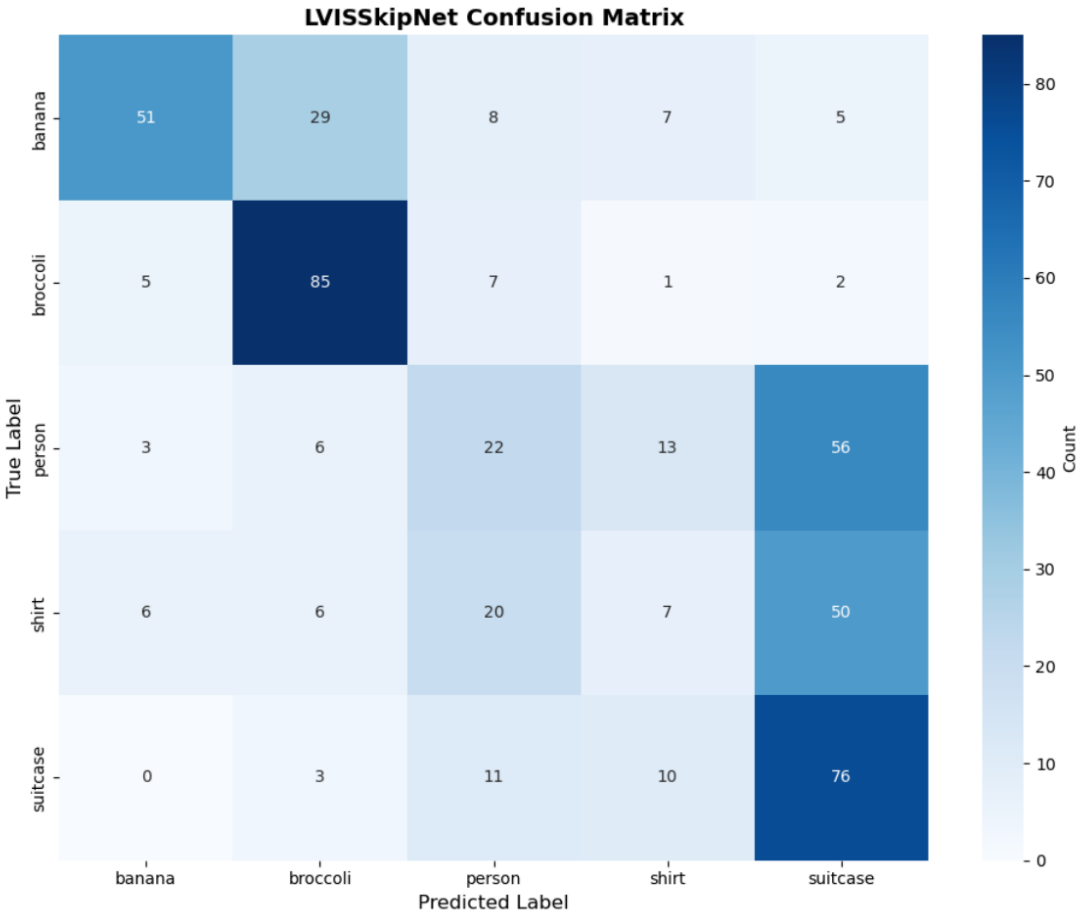

Figure 9: Confusion matrix of LVISSkipNet on the validation set.

Table 6: Per-class validation accuracy for LVISSkipNet.

Class	Accuracy (%)
banana	51.00
broccoli	85.00
person	22.00
shirt	7.87
suitcase	76.00

Training times per epoch (sec) (Using A10-GPU):

$$6.36, 6.32, 6.33, 6.33, 6.35, 6.29, 6.29, 6.34, 6.37, 6.34$$

In my results, the best validation accuracy is approximately 49.28% (best epoch during training). The per-class accuracy shows strong performance on some categories (for example broccoli and suitcase), while other categories (especially person and shirt) are more difficult. This is also visible in the confusion matrix: some images labeled as `person` and `shirt` are frequently predicted as other classes, suggesting that the model struggles with intra-class variation and visual similarity in these categories.

6 Task 5: Feature Map Visualization

6.1 Visualizing Learned Filters

I visualized the first-layer convolutional filters from both models: **LVISNet (shallow)** and **LVIS-SkipNet (deep)**. In both cases, the filters are 3×3 RGB kernels, and I displayed up to 32 filters after normalizing each filter to the range $[0, 1]$.

Listing 25: Filter Visualization

```
def get_first_conv_layer(model: nn.Module):
    for name, m in model.named_modules():
        if isinstance(m, nn.Conv2d):
            return name, m
    raise ValueError("No nn.Conv2d found in model.")

def visualize_first_layer_filters(model: nn.Module, max_filters=32,
    title_prefix="Model"):
    model.eval()
    conv_name, conv = get_first_conv_layer(model)
    W = conv.weight.detach().cpu() # (out_ch, in_ch, kH, kW)

    out_ch = W.shape[0]
    n = min(max_filters, out_ch)

    # Normalize each filter to [0,1] for display
    filt_imgs = []
    for i in range(n):
        w = W[i]  # (in_ch, kH, kW)
        # If in_ch==3, show as RGB;
        # else show as grayscale (mean over channels)
        if w.shape[0] == 3:
            w_img = w.permute(1,2,0).numpy() # (kH,kW,3)
        else:
            w_img = w.mean(dim=0).numpy() # (kH,kW)

        w_min, w_max = w_img.min(), w_img.max()
        if (w_max - w_min) < 1e-8:
            w_img = np.zeros_like(w_img)
        else:
            w_img = (w_img - w_min) / (w_max - w_min)

        filt_imgs.append(w_img)

    cols = 8
    rows = int(np.ceil(n / cols))
    plt.figure(figsize=(cols*1.6, rows*1.6))
    for i, img in enumerate(filt_imgs):
        ax = plt.subplot(rows, cols, i+1)
        if img.ndim == 3:
            ax.imshow(img)
        else:
            ax.imshow(img, cmap="gray")
        ax.axis("off")
        ax.set_title(f"ch {i}", fontsize=9)
```

```
45
46      plt.suptitle(f"{title_prefix} - First Conv Filters ({conv_name})", y
            =1.02, fontsize=14)
47      plt.tight_layout()
48      plt.show()
```

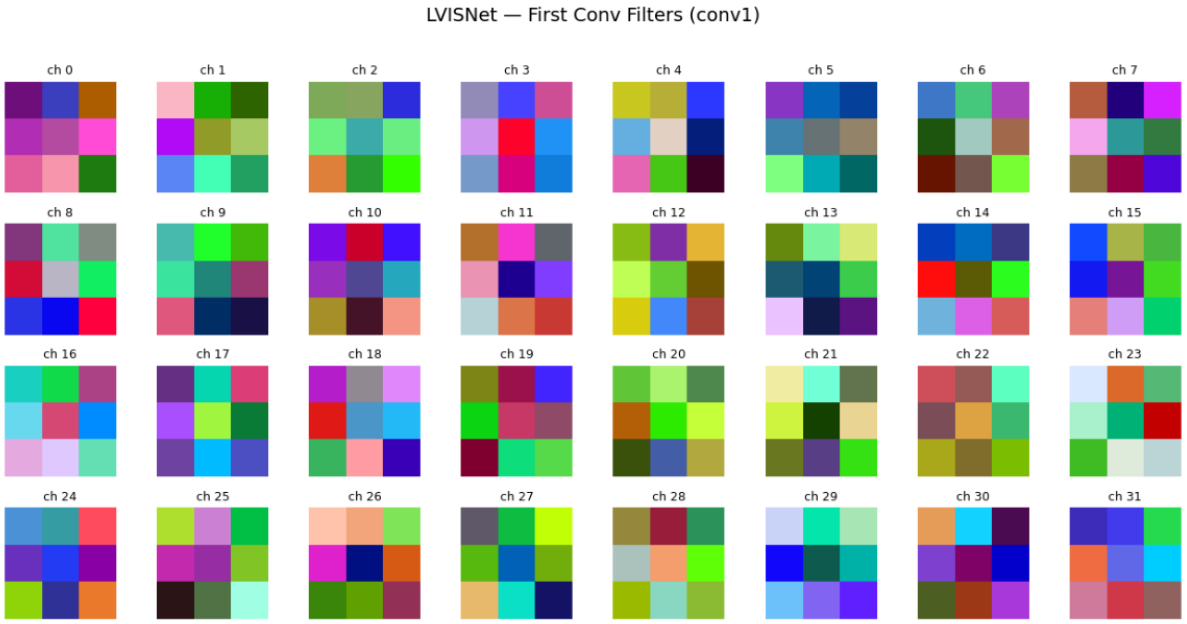

Figure 10: First Convolutional Filter of LVISNet

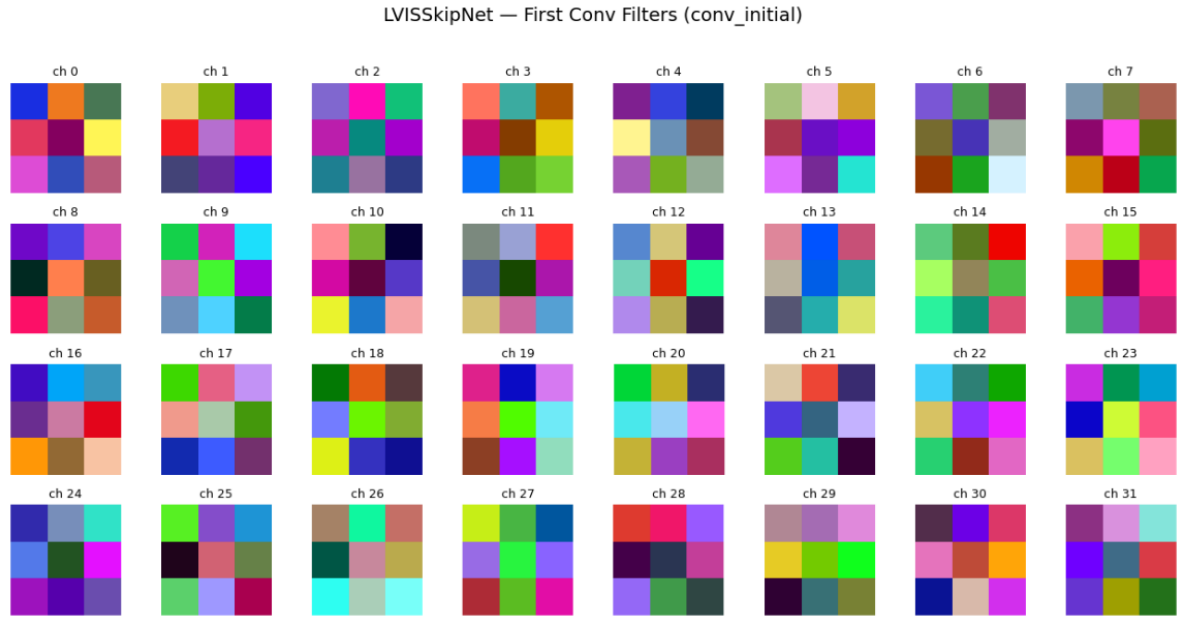

Figure 11: First Conlutional Filter of LVISSkipNet

Qualitatively, both models learn visually similar *low-level* filters, because the first convolution layer is close to the raw pixels in both networks. Many filters look like color-selective patterns (strong response to one channel) and some resemble simple edge or gradient detectors (contrast between neighboring pixels).

However, LVISSkipNet's first-layer filters appear slightly more diverse in color combinations and contrast patterns. This is expected because the deeper network has more capacity and can support a richer set of early features that later layers can reuse.

6.2 Visualizing Intermediate Feature Maps

I extracted feature maps from early, middle, and late layers using forward hooks and visualized the first 8 channels for each selected layer as grayscale images.

Listing 26: Feature Map Hooking

```
def get_feature_maps(model: nn.Module, input_image: torch.Tensor,
    layer_names):
    model.eval()
    feature_maps = {}
    hooks = []

    def hook_fn(name):
        def hook(module, inp, out):
            feature_maps[name] = out.detach().cpu()
        return hook

    name_to_module = dict(model.named_modules())
    for ln in layer_names:
        if ln not in name_to_module:
            raise ValueError(f"Layer name '{ln}' not found. Print names
                with print_hookable_layers(model).")
        hooks.append(name_to_module[ln].register_forward_hook(hook_fn(ln))
            )

    with torch.no_grad():
        _ = model(input_image)

    for h in hooks:
        h.remove()

    return feature_maps

def print_hookable_layers(model: nn.Module, only_conv_relu_pool=True):
    for name, m in model.named_modules():
        if name == "":
            continue
        if only_conv_relu_pool:
            if isinstance(m, (nn.Conv2d, nn.ReLU, nn.MaxPool2d, nn.
                AdaptiveAvgPool2d, nn.Sequential, nn.BatchNorm2d)):
                print(name, type(m).__name__)
        else:
            print(name, type(m).__name__)
```

Listing 27: Feature Map Visualization

```python
def visualize_feature_map_grid(feat: torch.Tensor, title="", max_channels=8):
    """
    feat: (1,C,H,W)
    Shows first max_channels channels as grayscale images.
    """
    assert feat.ndim == 4 and feat.size(0) == 1, "Expected (1,C,H,W)"
    C = feat.size(1)
    n = min(max_channels, C)

    cols = 4
    rows = int(np.ceil(n / cols))
    plt.figure(figsize=(cols*2.2, rows*2.2))
    for i in range(n):
        fm = feat[0, i].numpy()
        fm_min, fm_max = fm.min(), fm.max()
        if (fm_max - fm_min) < 1e-8:
            fm = np.zeros_like(fm)
        else:
            fm = (fm - fm_min) / (fm_max - fm_min)

        ax = plt.subplot(rows, cols, i+1)
        ax.imshow(fm, cmap="gray")
        ax.axis("off")
        ax.set_title(f"ch {i}", fontsize=9)

    plt.suptitle(title, y=1.02, fontsize=14)
    plt.tight_layout()
    plt.show()
```

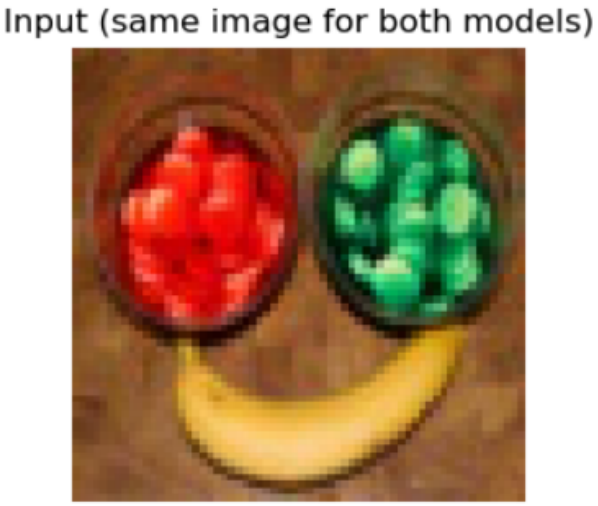

Figure 12: Sample Image

LVISNet Feature Maps

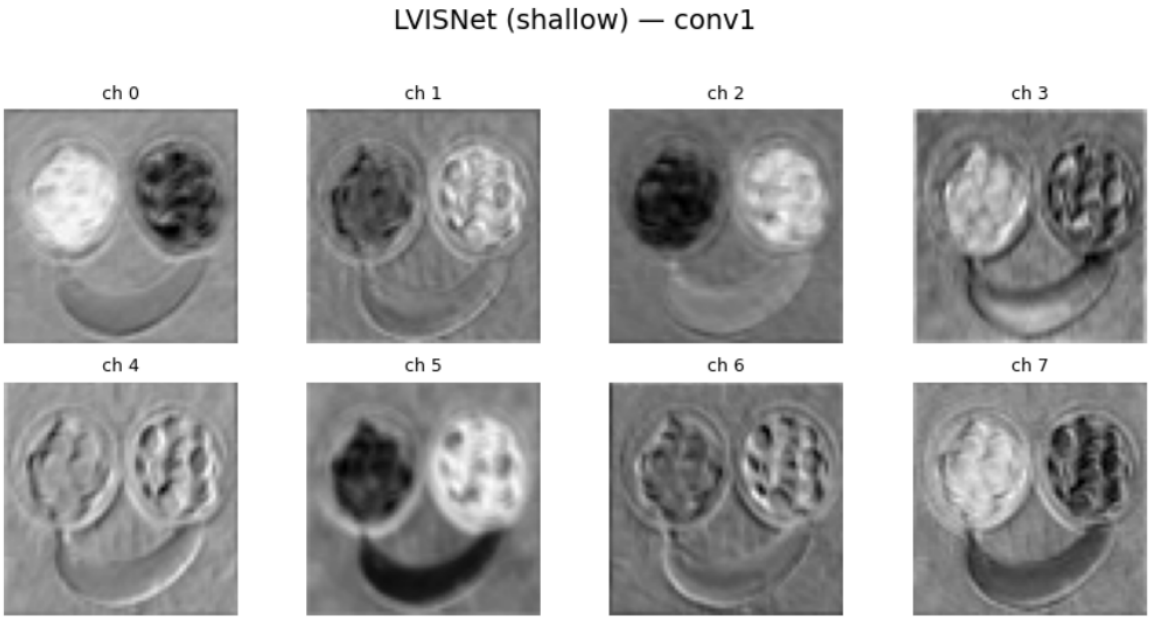

Figure 13: LVISNet Early Layer Feature Maps

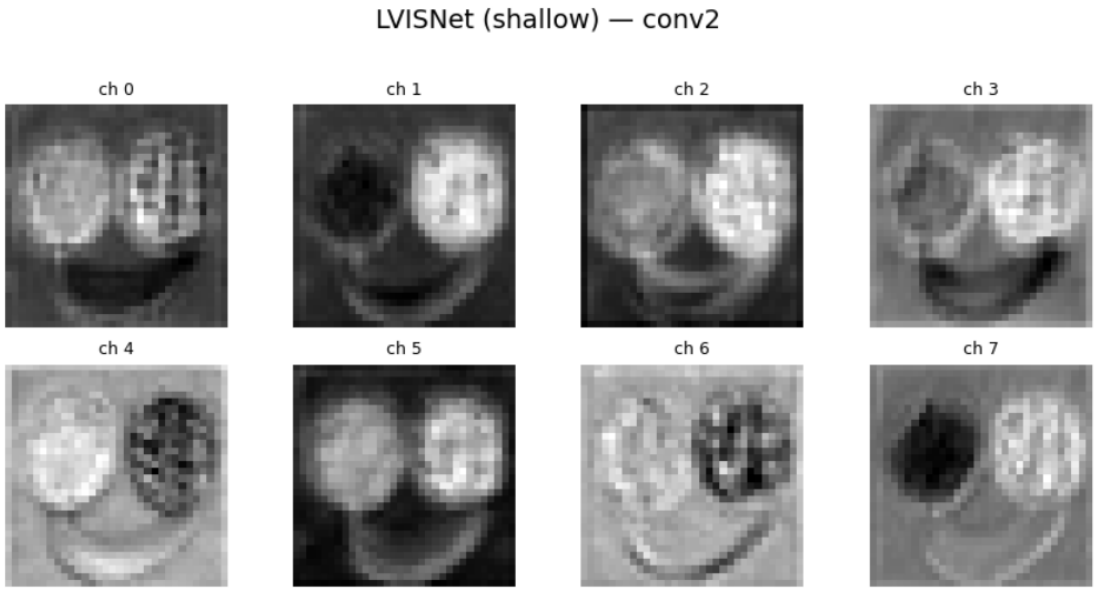

Figure 14: LVISNet Middle Layer Feature Maps

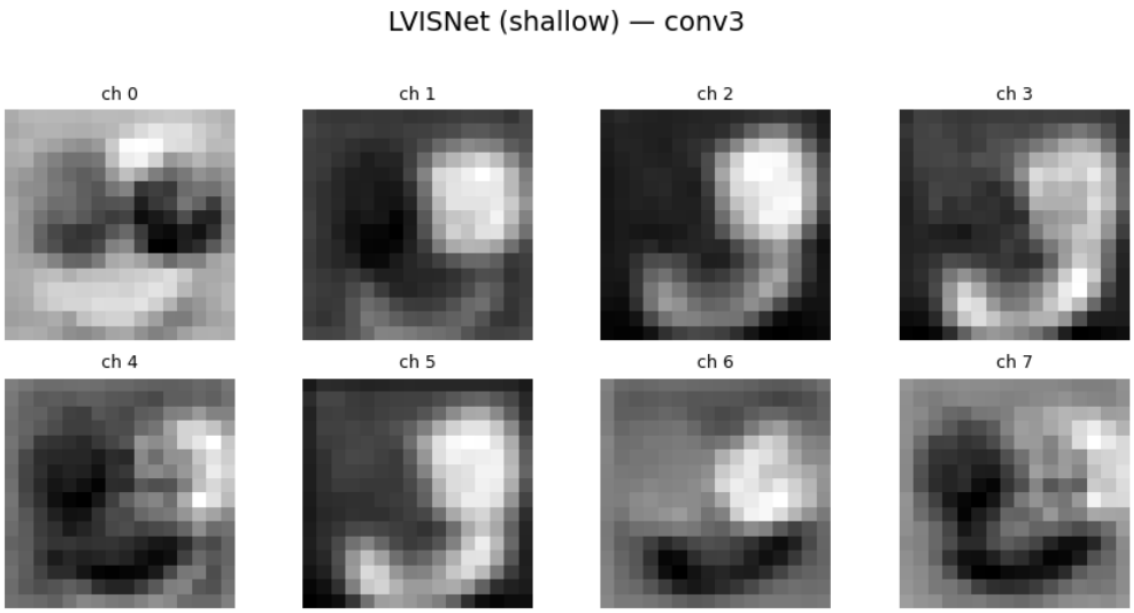

Figure 15: LVISNet Late Layer Feature Maps

LVISSkipNet Feature Maps

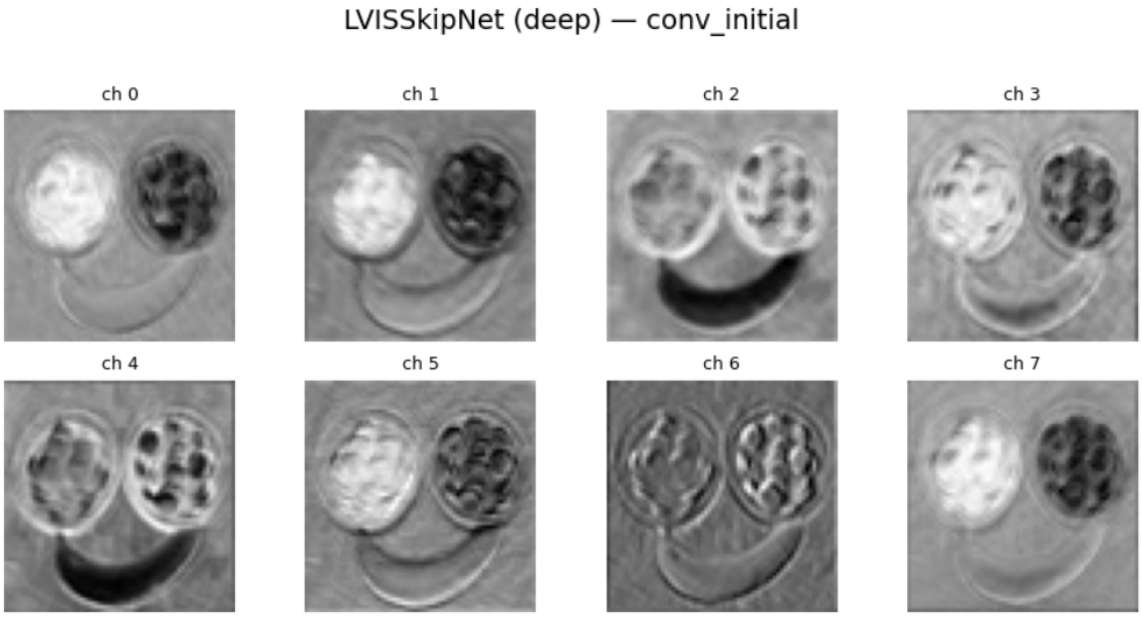

Figure 16: LVISSkipNet First Layer Feature Maps

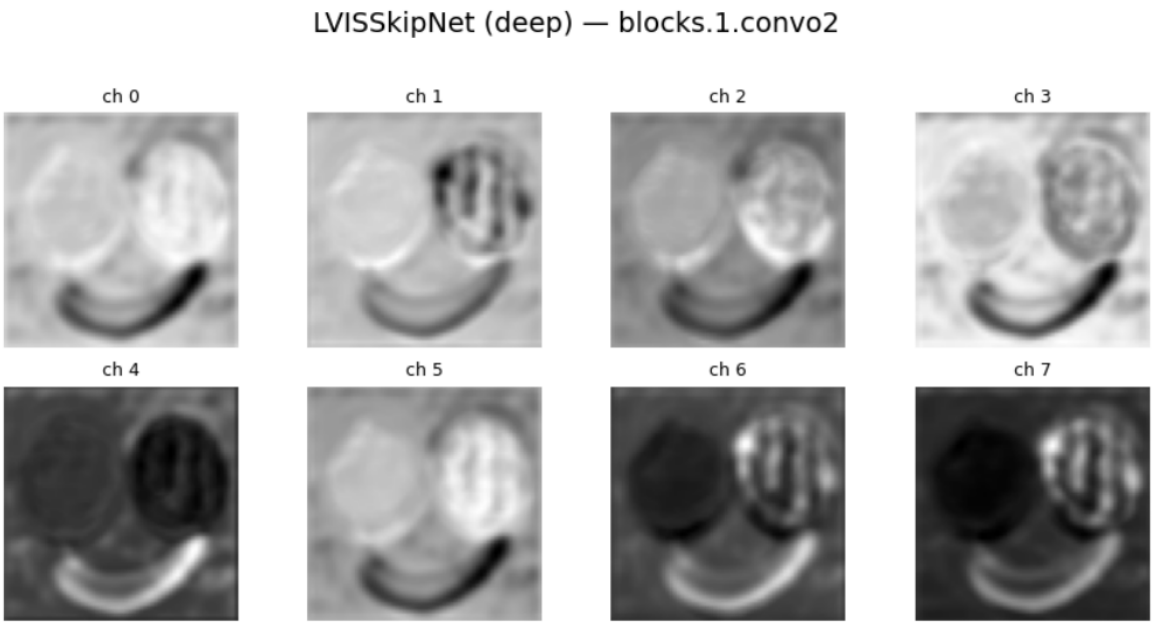

Figure 17: LVISSkipNet Early Layer Feature Maps

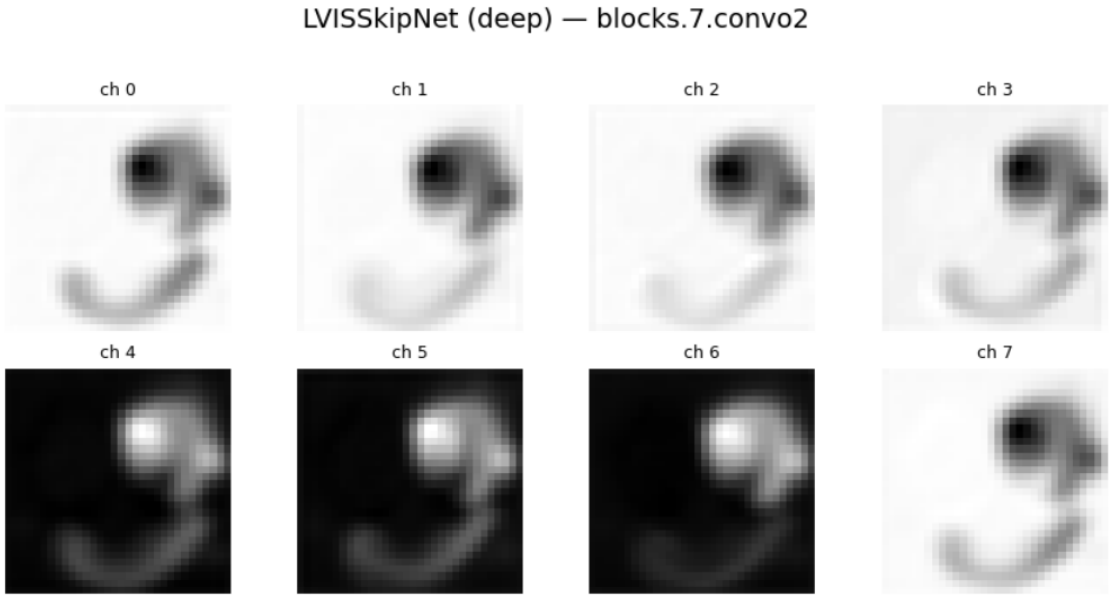

Figure 18: LVISSkipNet Middle Layer Feature Maps

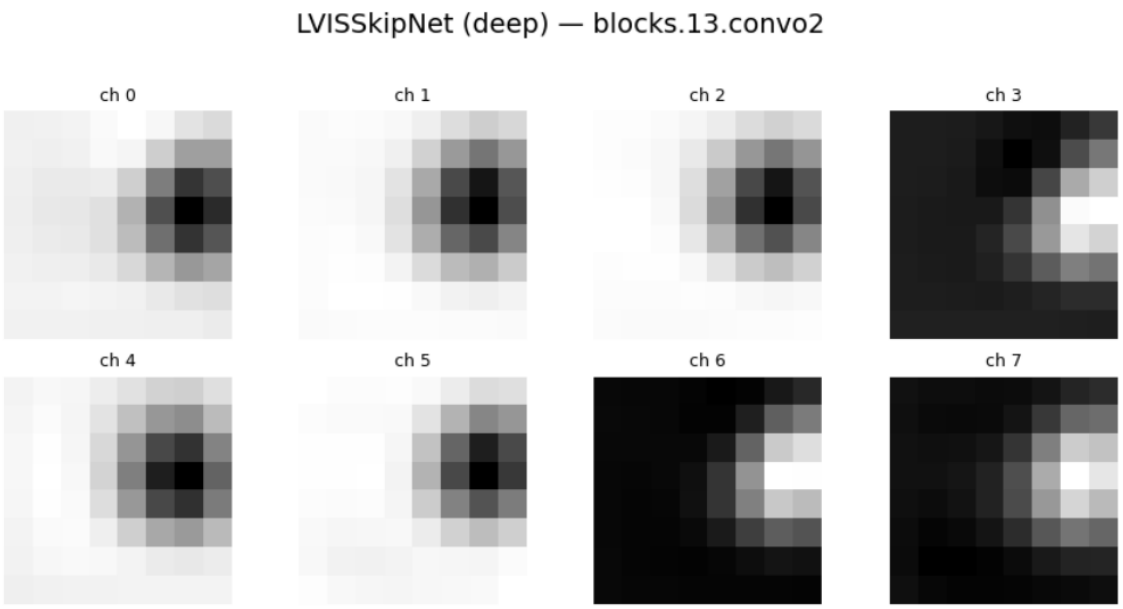

Figure 19: LVISSkipNet Late Layer Feature Maps

How feature maps change with depth. The behavior follows the usual deep CNN pattern:

- **Early layers:** Feature maps still resemble the original image and highlight basic edges, boundaries, simple textures, and color transitions. Many channels behave like detectors for simple oriented edges or local contrast.
- **Middle layers:** Feature maps become smoother and more selective. Background details are suppressed and stronger responses appear on more meaningful regions (object parts such as banana shape area, broccoli texture region, etc.).
- **Late layers:** Feature maps become more abstract and less human-interpretable. Often the activation concentrates in fewer regions, indicating that the network is focusing on discriminative parts instead of the full image. This matches the idea that deeper layers combine earlier features into higher-level concepts.

6.3 Comparative Analysis: Shallow vs Deep

I compared LVISNet (shallow) and LVISSkipNet (deep) feature maps using the **same input image** in section 6.2.

Do deeper feature maps appear more discriminative? Yes, in my visualizations LVISSkipNet's deeper feature maps are generally more selective. The late feature maps often show strong activation only around the most informative region, while LVISNet tends to keep more mixed activations (less separation between object and background). This suggests that the deeper model learns more refined intermediate representations.

First-layer filters: similar or different? The first-layer filters are mostly similar across the two models (basic edges/colors), because both models must start from the same pixel-level information. The larger differences appear mainly in deeper feature maps, not in the first convolution layer.

Relation to multi-channel convolution. Lecture [3] explains that a convolution layer with multiple input channels forms each output channel by combining information across channels (a weighted sum over RGB channels and spatial neighborhoods). This helps explain why early feature maps can look like edge/texture detectors (local spatial patterns) while later layers become more abstract: deeper layers repeatedly mix channels and spatial evidence, building progressively higher-level features from earlier ones. In LVISSkipNet, the additional depth and skip connections allow these multi-channel combinations to be reused and refined many times, which supports more discriminative late-layer representations.

7 Task 6: Final Comparative Analysis

7.1 Confusion Matrices

Listing 28: Confusion Matrix Computation

```
@torch.no_grad()
def compute_confusion_matrix(model, dataloader, num_classes, device):
    model.eval()
    num_classes = int(num_classes)
    cm = np.zeros((num_classes, num_classes), dtype=np.int64)

    for batch in dataloader:
        if isinstance(batch, dict):
            inputs, labels = batch["inputs"], batch["labels"]
        else:
            inputs, labels = batch

        inputs = inputs.to(device)

        if isinstance(labels, (list, tuple)):
            labels = torch.tensor(labels)

        # If one-hot/prob labels, collapse to class indices
        if labels.ndim > 1:
            labels = labels.argmax(dim=1)

        labels = labels.long().cpu().numpy()

        outputs = model(inputs)
        preds = outputs.argmax(dim=1).detach().cpu().numpy()

        for t, p in zip(labels, preds):
            if 0 <= t < num_classes and 0 <= p < num_classes:
                cm[t, p] += 1

    return cm
```

Listing 29: Confusion Matrix Visualization

```python
def plot_confusion_matrix(cm, class_names, title, save_path=None):
    plt.figure(figsize=(10,8))

    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=class_names, yticklabels=class_names,
                cbar_kws={'label': 'Count'})

    plt.title(title, fontsize=14, fontweight='bold')
    plt.xlabel('Predicted Label', fontsize=12)
    plt.ylabel('True Label', fontsize=12)
    plt.tight_layout()

    if save_path:
        plt.savefig(save_path, dpi=300, bbox_inches='tight')
    plt.show()
```

For CIFAR-10, I used the class order {plane, car, bird, cat, deer, dog, frog, horse, ship, truck}. The provided confusion matrices in my results were row-normalized (percentages). To visualize them as standard confusion matrices, I converted each row to counts by assuming CIFAR-10 test has 1000 images per class, then plotted heatmaps.

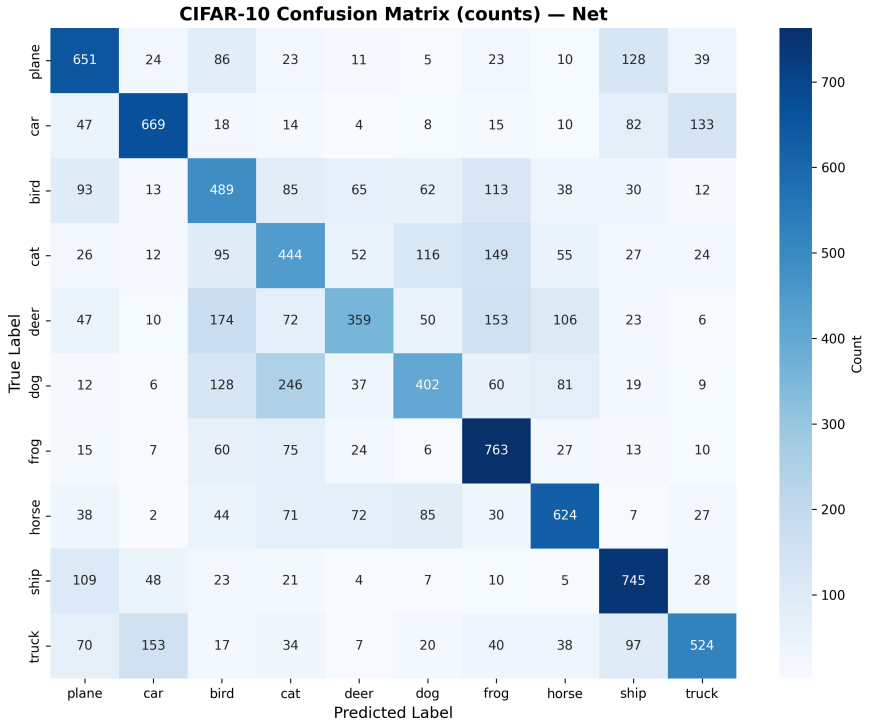

Figure 20: CIFAR-10 confusion matrix (Net baseline).

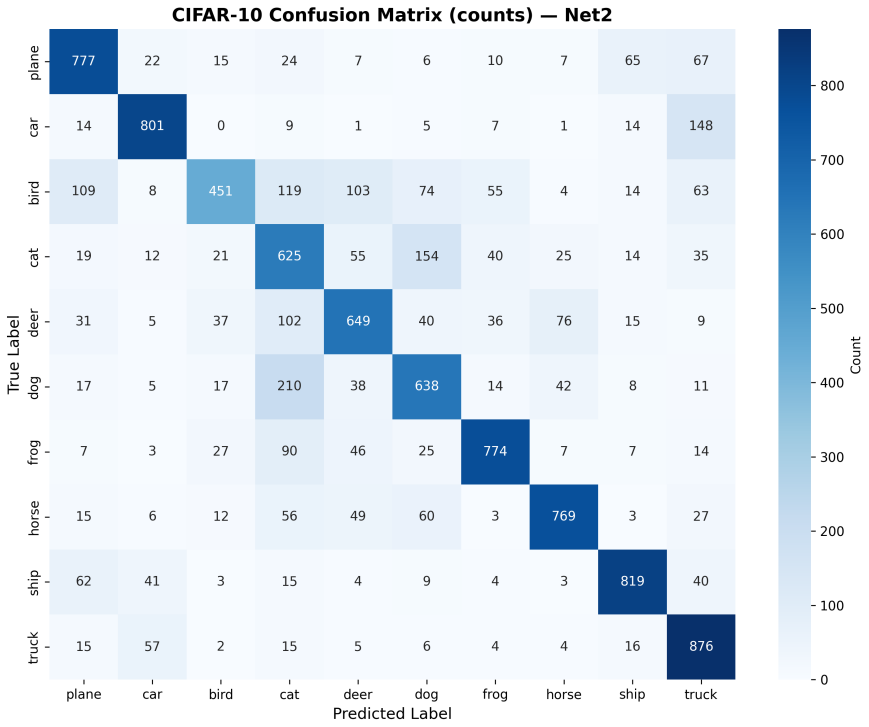

Figure 21: CIFAR-10 confusion matrix (Net2 baseline).

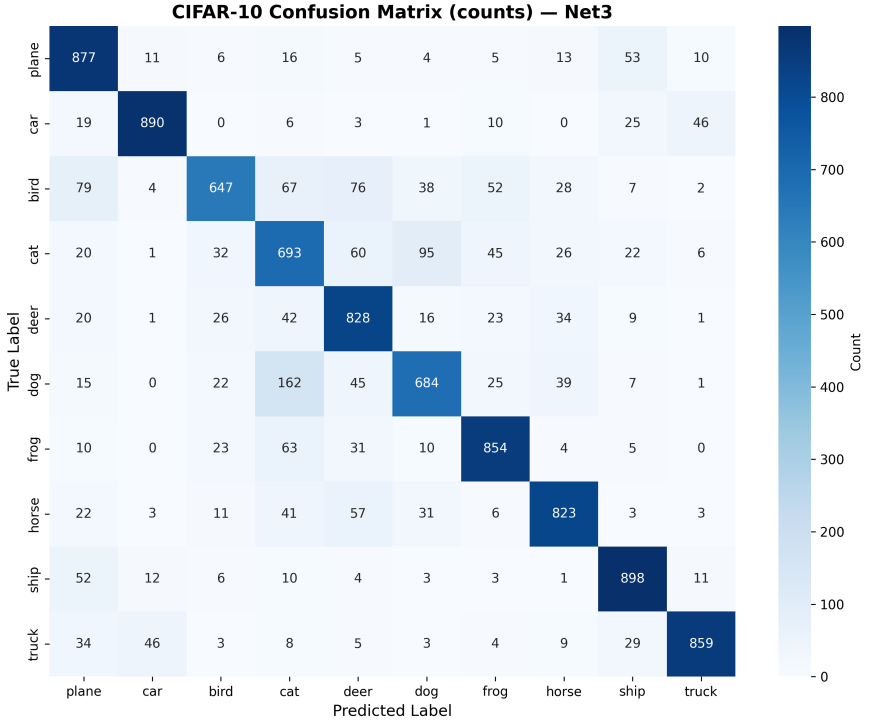

Figure 22: CIFAR-10 confusion matrix (Net3 baseline).

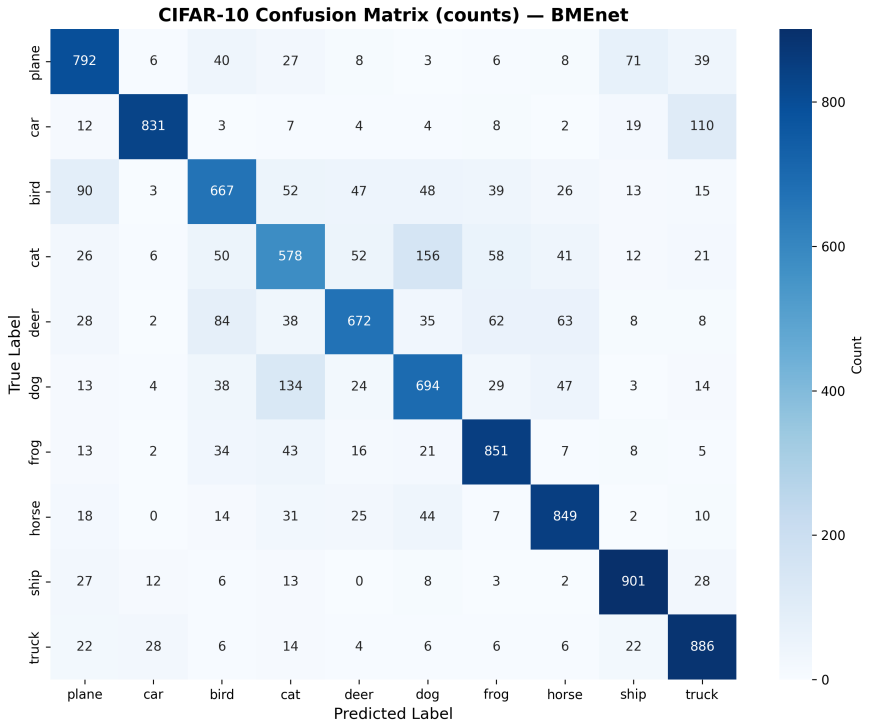

Figure 23: CIFAR-10 confusion matrix (BMEnet).

For LVIS multi-instance same-object, I generated one 5×5 confusion matrix on the validation set using my from-scratch implementation, and plotted it as a heatmap.

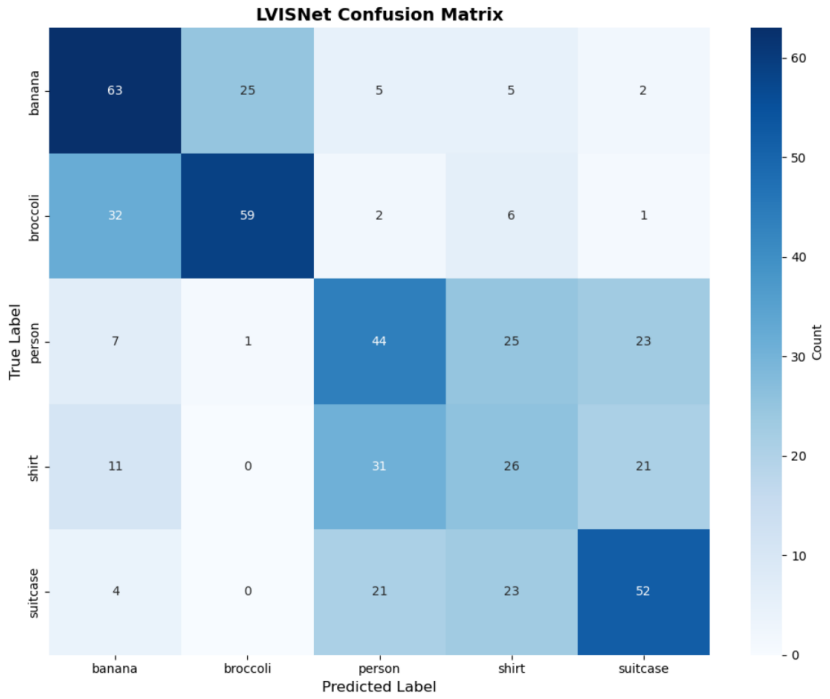

Figure 24: LVIS confusion matrix on validation set

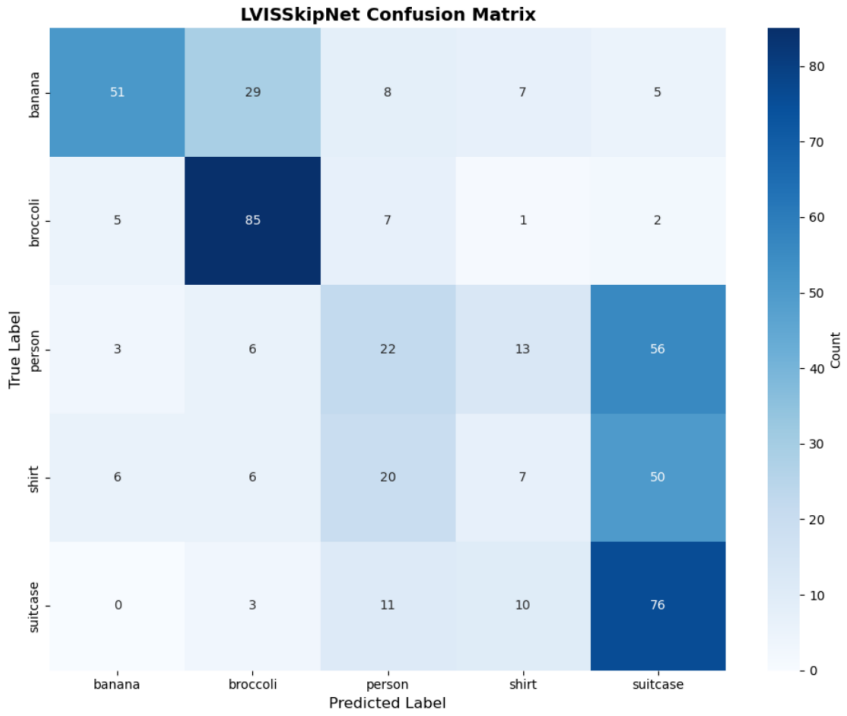

Figure 25: LVISSkipNet confusion matrix on validation set

7.2 LVISNet vs LVISSkipNet

This comparison tests whether increasing depth and adding skip connections helps on the LVIS multi-instance dataset.

Network	Parameters	Val Accuracy (%)
LVISNet (shallow)	2,224,645	49.90
LVISSkipNet (deep)	18,140,229	51.94

Table 7: LVISNet vs LVISSkipNet summary (validation set).

Discussion.

- **Does depth help?** In my results, LVISSkipNet achieved **51.94%** best validation accuracy. Compared to LVISNet, this indicates that adding more layers with skip connections can help, because skip connections make optimization easier and reduce degradation when the network gets deeper.
- **Normalization insights.** From Task 3 on CIFAR-10, different normalization types changed convergence speed and final accuracy. If I use `NormSkipBlock` in LVISSkipNet, I would recommend the normalization strategy that gave the best stability and accuracy in my Task 3 experiments (typically BatchNorm or GroupNorm).
- **Overall findings.** The key takeaway is that (1) deeper networks can learn more complex features, (2) skip connections improve training stability for deep models, and (3) normalization strongly affects how fast the model converges and how stable training is.

7.3 Master Summary Table

Table 8 summarizes all experiments across CIFAR-10 and LVIS. For CIFAR-10, the accuracy values come from the final evaluation of each model. For LVIS, the accuracy is computed on the validation set.

Network	Dataset	Layers	Parameters	Accuracy (%)
Net	CIFAR-10	2 conv	62,006	56.70
Net2	CIFAR-10	3 conv	1,468,036	71.79
Net3	CIFAR-10	8 conv	1,469,802	80.53
BMEnet	CIFAR-10	98 blocks (depth=32)	69,493,890	77.21
NormBMEnet (BN)	CIFAR-10	26 blocks (depth=8)	30,195,266	74.85
NormBMEnet (IN)	CIFAR-10	26 blocks (depth=8)	30,195,266	77.88
NormBMEnet (LN)	CIFAR-10	26 blocks (depth=8)	30,195,266	74.57
NormBMEnet (GN, G=8)	CIFAR-10	26 blocks (depth=8)	30,195,266	73.53
LVISNet	LVIS multi-same	3 blocks	2,224,645	49.90
LVISSkipNet	LVIS multi-same	26 blocks (53 conv+lin)	18,140,229	51.94

Table 8: Master summary table across all experiments.

8 Bonus Task

8.1 Ablation Study: Normalization × Depth

Listing 30:]LVISSkipNet20 Architecture [1]

```
class LVISSkipNet20(nn.Module):
    def __init__(self, num_classes=5, norm_type="batch", num_groups=8,
        skip_connections=True):
        super().__init__()
        self.conv_initial = nn.Conv2d(3, 64, kernel_size=3, padding=1,
            bias=False)
        self.bn_initial = nn.BatchNorm2d(64) if norm_type == "batch" else
            nn.Identity()

        blocks = []
        # Stage 1: 64 channels, no downsample
        for _ in range(5):
            blocks.append(NormSkipBlock(64, 64, norm_type=norm_type,
                num_groups=num_groups, downsample=False, skip_connections=
                skip_connections))
        # Transition: 64 -> 128, downsample=True (64x64 -> 32x32)
        blocks.append(NormSkipBlock(64, 128, norm_type=norm_type,
            num_groups=num_groups, downsample=True, skip_connections=
            skip_connections))
        # Stage 2: 128 channels
        for _ in range(5):
            blocks.append(NormSkipBlock(128, 128, norm_type=norm_type,
                num_groups=num_groups, downsample=False, skip_connections=
                skip_connections))
        # Transition: 128 -> 256, downsample=True (32x32 -> 16x16)
```

```
        blocks.append(NormSkipBlock(128, 256, norm_type=norm_type,
            num_groups=num_groups, downsample=True, skip_connections=
            skip_connections))
        # Stage 3: 256 channels
        for _ in range(5):
            blocks.append(NormSkipBlock(256, 256, norm_type=norm_type,
                num_groups=num_groups, downsample=False, skip_connections=
                skip_connections))
        # Transition: 256 -> 256, downsample=True (16x16 -> 8x8)
        blocks.append(NormSkipBlock(256, 256, norm_type=norm_type,
            num_groups=num_groups, downsample=True, skip_connections=
            skip_connections))
        # Stage 4: 256 channels (still 8x8)
        for _ in range(5):
            blocks.append(NormSkipBlock(256, 256, norm_type=norm_type,
                num_groups=num_groups, downsample=False, skip_connections=
                skip_connections))

        self.blocks = nn.Sequential(*blocks)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(256, num_classes)

    def forward(self, x):
        x = F.relu(self.bn_initial(self.conv_initial(x)))
        x = self.blocks(x)
        x = self.avgpool(x)                    # (B,256,1,1)
        x = x.view(x.size(0), -1)              # (B,256)
        x = self.fc(x)                         # (B,5)
        return x

    def count_parameters(self):
        return sum(p.numel() for p in self.parameters() if p.requires_grad
            )

    def count_layers(self):
        return sum(1 for m in self.modules() if isinstance(m, (nn.Conv2d,
            nn.Linear)))
```

Listing 31: Ablation Configuration

```
@dataclass
class Config:
    dataset_root: str = "/scratch/gilbreth/skasap/datasets/lvis_dataset/
        multi_instance_same"
    classes: Tuple[str, ...] = ("banana", "broccoli", "person", "shirt", "
        suitcase")
    epochs: int = 10
    batch_size: int = 32
    lr: float = 3e-4
    betas: Tuple[float, float] = (0.9, 0.99)
    num_groups: int = 8
    skip_connections: bool = True
    num_workers: int = 4
```

```
    device: str = "cuda"
    out_dir: str = "./ablation_results"
    seed: int = 42
    save_best: bool = True
    overwrite: bool = False
```

Listing 32: Ablation Training Function

```
def run_one(cfg: Config, depth: int, norm: str, train_loader, val_loader,
    train_ds, val_ds) -> dict:
    cfg = Config()
    ...
    model = make_model(depth, norm, num_classes, cfg.num_groups, cfg.
        skip_connections).to(device)

    print(f"\n=== {run_name} ===")
    print("Trainable params:", model.count_parameters())
    print("Learnable layers (Conv2d+Linear):", model.count_layers())

    optimizer = torch.optim.Adam(model.parameters(), lr=cfg.lr, betas=cfg.
        betas)
    criterion = nn.CrossEntropyLoss()

    # dataset verification
    train_counts = dataset_verification(train_ds)
    val_counts = dataset_verification(val_ds)

    metrics = {...} # all the data (for report it is shortened)

    best_val_acc = -1.0
    best_state = None
    best_epoch = -1

    for ep in range(1, cfg.epochs + 1):
        print(f"Epoch {ep:02d}/{cfg.epochs}")

        t0 = time.time()
        tr_loss, tr_acc = train.train_one_epoch(model, train_loader,
            optimizer, criterion, device)
        va_loss, va_acc = train.validate(model, val_loader, criterion,
            device)
        t1 = time.time()

        metrics["train"]["loss"].append(float(tr_loss))
        metrics["train"]["acc"].append(float(tr_acc))
        metrics["val"]["loss"].append(float(va_loss))
        metrics["val"]["acc"].append(float(va_acc))
        metrics["time_per_epoch_sec"].append(float(t1 - t0))

        if cfg.save_best and va_acc > best_val_acc:
            best_val_acc = va_acc
            best_epoch = ep
            best_state = {k: v.detach().cpu() for k, v in model.state_dict
                ().items()}
```

```
41
42      # load best for final confusion matrix
43      if best_state is not None:
44          model.load_state_dict(best_state)
45          torch.save(model.state_dict(), best_path)
46
47      cm = evaluate.compute_confusion_matrix(model, val_loader, num_classes,
            device)
48      per_cls_acc = evaluate.per_class_accuracy(cm)
49
50      metrics["final"] = {...} #for report shortened
51
52      with open(json_path, "w") as f:
53          json.dump(metrics, f, indent=2)
54
55      return metrics
```

Listing 33: Ablation Study Main Code

```
1  cfg = Config()
2  set_seed(cfg.seed)
3  os.makedirs(cfg.out_dir, exist_ok=True)
4
5  train_loader, val_loader, train_ds, val_ds = get_dataloaders(cfg)
6
7  depths = [10, 20, 30]
8  norms = ["batch", "instance", "layer", "group"]
9
10 # Keep all results for plotting + tables
11 all_results: Dict[int, Dict[str, dict]] = {d: {} for d in depths}
12
13 for d in depths:
14     for n in norms:
15         metrics = run_one(cfg, d, n, train_loader, val_loader, train_ds,
               val_ds)
16         all_results[d][n] = metrics
17
18     # plot overlaid curves by norm (per depth)
19     plot_curves(all_results[d], depth=d, out_dir=cfg.out_dir)
20
21 # 3x4 accuracy table (rows=depth, cols=norm) using best_val_acc
22 table_path = os.path.join(cfg.out_dir, "accuracy_table.csv")
23 with open(table_path, "w", newline="") as f:
24     writer = csv.writer(f)
25     writer.writerow(["depth"] + norms)
26     for d in depths:
27         row = [d]
28         for n in norms:
29             row.append(all_results[d][n]["final"]["best_val_acc"])
30         writer.writerow(row)
```

Figures 26–28 show training accuracy and loss curves for LVISSkipNet-10, LVISSkipNet-20, and LVISSkipNet-30 across BatchNorm (BN), InstanceNorm (IN), LayerNorm (LN), and GroupNorm (GN).

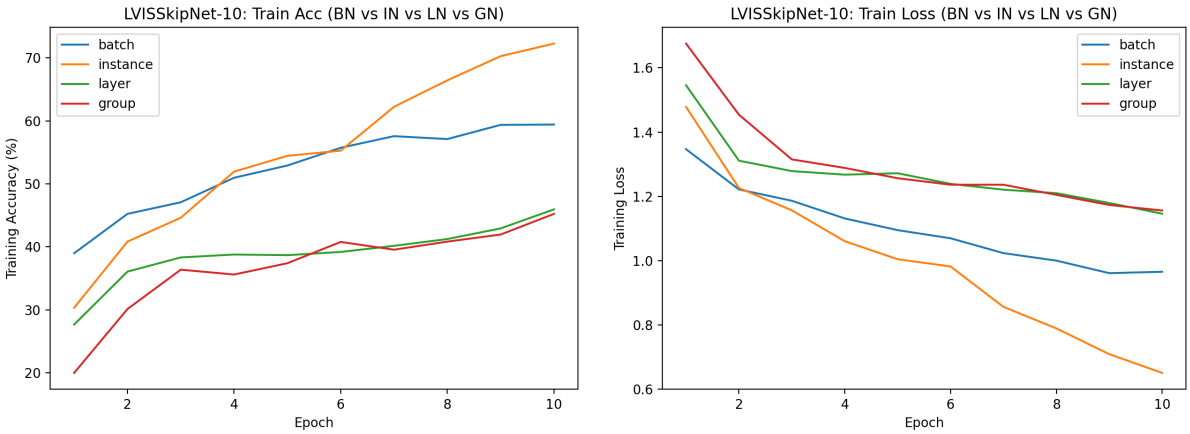

Figure 26: LVISSkipNet-10: Training Accuracy and Loss (BN vs IN vs LN vs GN)

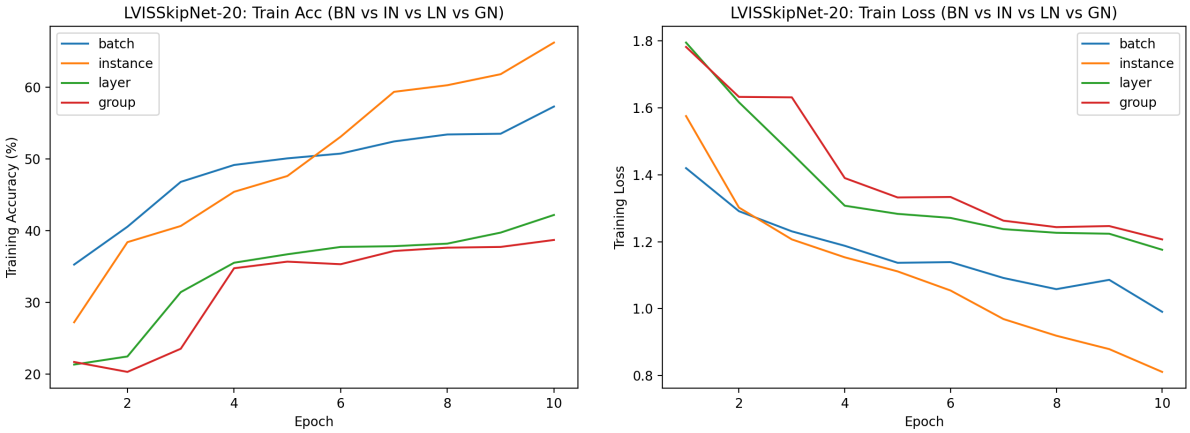

Figure 27: LVISSkipNet-20: Training Accuracy and Loss (BN vs IN vs LN vs GN)

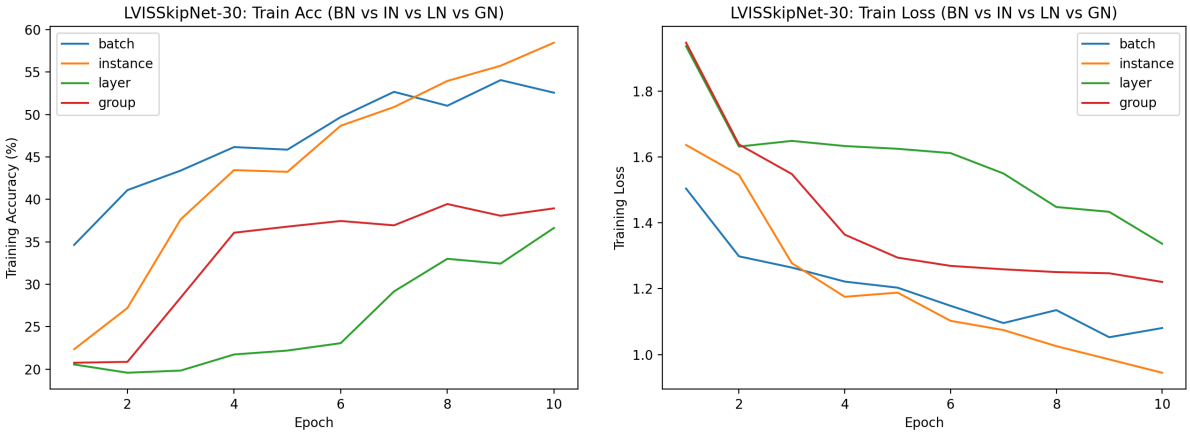

Figure 28: LVISSkipNet-30: Training Accuracy and Loss (BN vs IN vs LN vs GN)

Table 9: Validation Accuracy (%) for LVISSkipNet across Depth and Normalization Type

Depth	BatchNorm	InstanceNorm	LayerNorm	GroupNorm
10	55.62	54.19	45.81	45.81
20	53.99	54.19	45.81	40.08
30	52.15	55.83	42.94	38.45

Several consistent patterns emerge across depths. First, Instance Normalization (IN) produces the fastest drop in training loss and achieves the highest training accuracy for all depths. However, this does not always translate to the best validation accuracy, indicating stronger overfitting behavior compared to BatchNorm (BN).

Second, as depth increases from 10 to 30 SkipBlocks, BN remains relatively stable in training dynamics, while LN and GN become increasingly unstable and underperforming. In deeper networks (depth = 30), LN exhibits noticeably slower optimization and lower final training accuracy, suggesting difficulty in stabilizing very deep feature hierarchies.

Third, diminishing returns are visible when moving from depth 20 to depth 30. Although IN continues to improve training accuracy, validation accuracy does not consistently improve. This suggests that additional depth increases model capacity but does not necessarily improve generalization for this dataset.

Overall, BN and IN are clearly superior for this task, while LN and GN struggle as network depth increases.

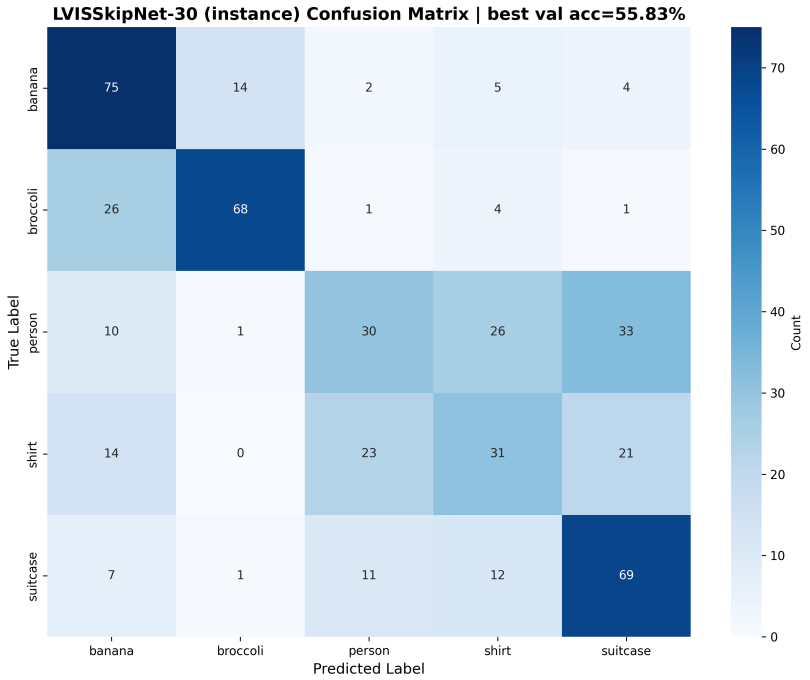

Figure 29: LVISSkipNet-30 with InstanceNorm: confusion matrix on the validation set.

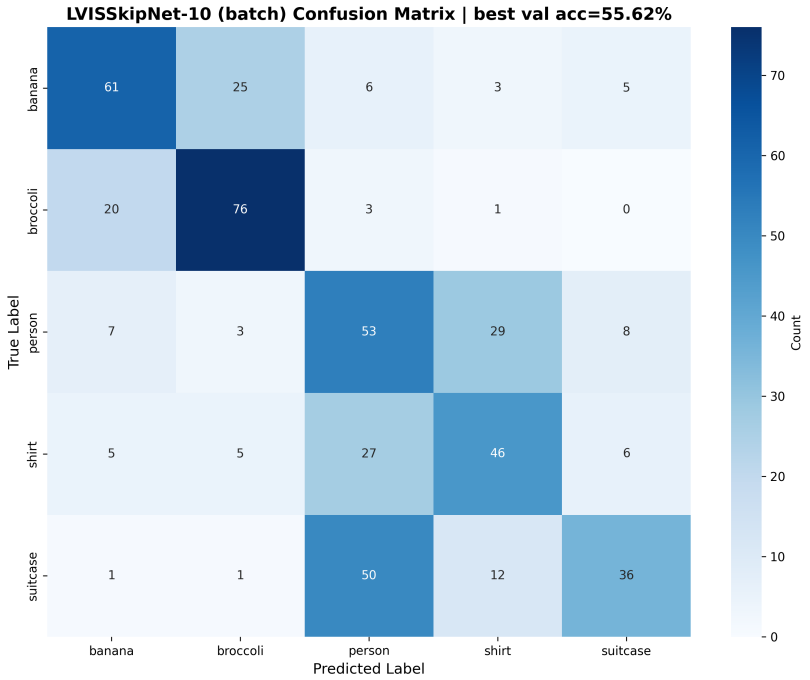

Figure 30: LVISSkipNet-10 with BatchNorm: confusion matrix on the validation set.

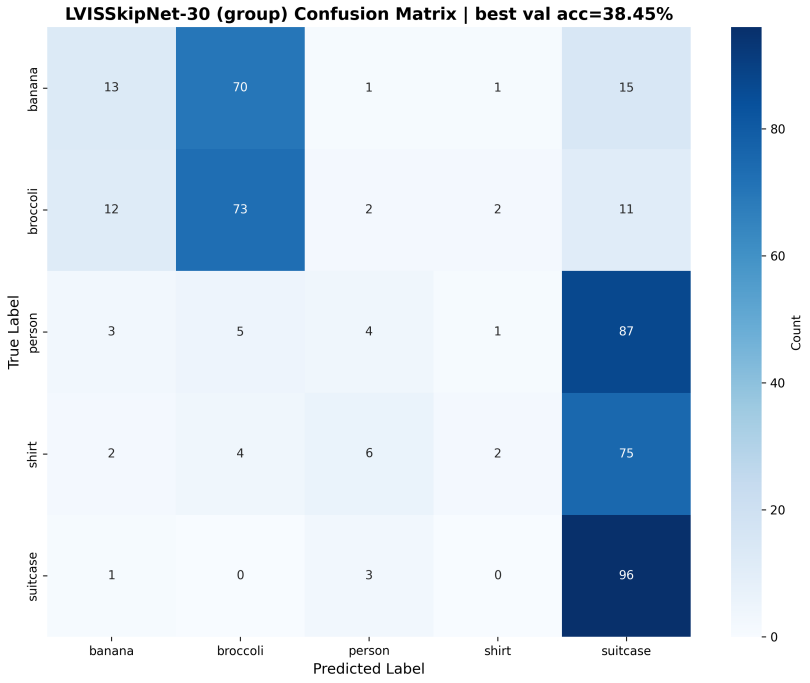

Figure 31: LVISSkipNet-30 with GroupNorm: confusion matrix on the validation set (challenging case).

The confusion matrices reveal that the best-performing configuration (LVISSkipNet-30 + IN) significantly reduces inter-class confusion between visually similar categories.

In contrast, the deep GN configuration shows widespread confusion across multiple classes, indicating unstable feature normalization at greater depths.

The shallow BN model provides balanced performance but struggles with subtle inter-class variations that the deeper IN model captures more effectively.

Does the best normalization change with depth? Yes. While BN performs competitively at shallow depths, IN becomes the strongest performer as depth increases.

Where do diminishing returns appear? Between depth 20 and 30. Although training accuracy increases, validation accuracy does not consistently improve.

Does BN remain best at all depths? No. IN surpasses BN at depth 30, suggesting that instance-level normalization stabilizes very deep skip-connected networks more effectively.

References

[1] Avinash Kak, *DLStudio*. Available at: https://engineering.purdue.edu/kak/distDLS/

[2] Avinash Kak, *Using Skip Connections to Mitigate the Problem of Vanishing Gradients, and Using Batch, Instance, and Layer Normalizations for Improved SGD in Deep Networks*. Available at: https://engineering.purdue.edu/kak/pdf-kak/SkipConsAndBN.pdf

[3] Avinash Kak, *Demystifying the Convolutions in PyTorch*. Available at: https://engineering.purdue.edu/kak/pdfkak/DemystifyConvo.pdf