ECE 60146 - HW5

Abin Mathew

3.1 Task 1: The Degradation Problem

Objective: Design a deeper network and observe that more layers do not automatically yield better performance

3.1.1 Designing Net3

We create Net3 with 8 convolutional layers by extending ExperimentsWithCIFAR through inheritance.

Code Snippet: hw5_AbinMathew/models.py

```
class Net3(DLStudio.ExperimentsWithCIFAR):
    """
    Input:  (B, 3, 32, 32)
    Output: (B, 10)
    """

    def __init__(self, num_classes=10):
        super(DLStudio.ExperimentsWithCIFAR, self).__init__()

        # Block 1: 32x32 -> 16x16
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(64)
        self.conv2 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)

        # Block 2: 16x16 -> 8x8
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.conv4 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.bn4 = nn.BatchNorm2d(128)

        # Block 3: 8x8 -> 4x4
        self.conv5 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.bn5 = nn.BatchNorm2d(256)
        self.conv6 = nn.Conv2d(256, 256, kernel_size=3, padding=1)
        self.bn6 = nn.BatchNorm2d(256)

        # Block 4: 4x4 -> 2x2
        self.conv7 = nn.Conv2d(256, 512, kernel_size=3, padding=1)
```

```
            self.bn7 = nn.BatchNorm2d(512)
            self.conv8 = nn.Conv2d(512, 512, kernel_size=3, padding=1)
            self.bn8 = nn.BatchNorm2d(512)

            self.pool = nn.MaxPool2d(2, 2)

            # Final feature map: 512 x 2 x 2
            self.fc1 = nn.Linear(512 * 2 * 2, 512)
            self.dropout = nn.Dropout(0.5)
            self.fc2 = nn.Linear(512, num_classes)

        def forward(self, x):
            # Block 1
            x = F.relu(self.bn1(self.conv1(x)))
            x = F.relu(self.bn2(self.conv2(x)))
            x = self.pool(x)

            # Block 2
            x = F.relu(self.bn3(self.conv3(x)))
            x = F.relu(self.bn4(self.conv4(x)))
            x = self.pool(x)

            # Block 3
            x = F.relu(self.bn5(self.conv5(x)))
            x = F.relu(self.bn6(self.conv6(x)))
            x = self.pool(x)

            # Block 4
            x = F.relu(self.bn7(self.conv7(x)))
            x = F.relu(self.bn8(self.conv8(x)))
            x = self.pool(x)

            x = x.view(x.shape[0], -1)
            x = self.dropout(F.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)

    # Calling code for Net3()
    if __name__ == "__main__":
        #We use same classes as the example.
        classes = (
            'plane', 'car', 'bird', 'cat', 'deer',
            'dog', 'frog', 'horse', 'ship', 'truck'
        )

        dls = DLStudio(
            dataroot="./data/CIFAR-10/",
            image_size=[32, 32],
```

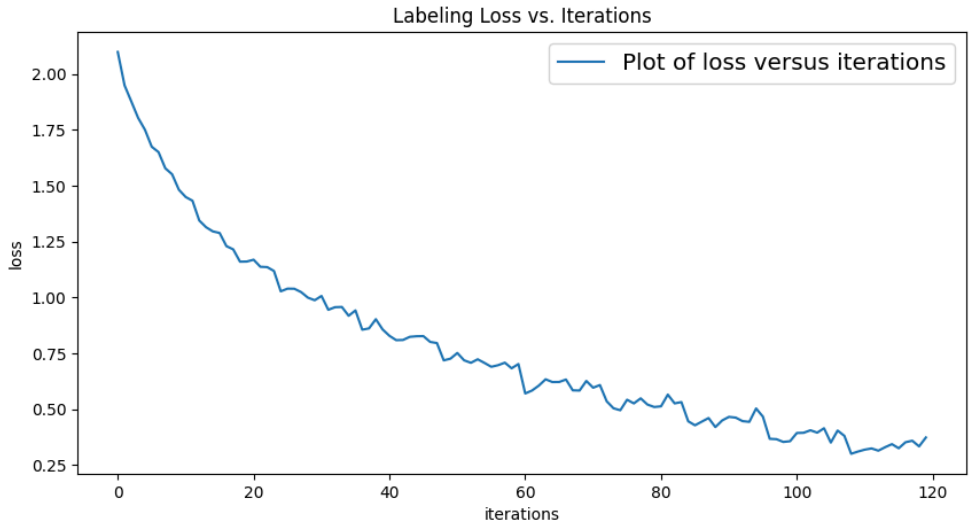
```
            path_saved_model="./saved_model_net3",
            momentum=0.9,
            learning_rate=1e-3,
            epochs=10,
            batch_size=4,
            classes=classes,
            use_gpu=True,
        )

        exp_cifar = DLStudio.ExperimentsWithCIFAR(dl_studio=dls)
        exp_cifar.load_cifar_10_dataset()

        net = Net3(num_classes=10)
        conv_count = sum(1 for m in net.modules() if isinstance(m,
nn.Conv2d))

        print(f"Network architecture:\n{net}")
        print(f"Number of nn.Conv2d layers: {conv_count}")
        print(f"Total trainable parameters:
{net.count_parameters():,}")

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

Results:

Number of `nn.Conv2d` layers: **8**

Total trainable parameters: **5,743,434**

Training loss curve for Net3:



Confusion Matrix (rows = true class, columns = predicted class)

True \ Pred	plane	car	bird	cat	deer	dog	frog	horse	ship	truck
plane	89.70	0.80	2.80	0.60	0.60	0.10	0.10	1.40	2.90	1.00
car	1.30	94.00	0.20	0.10	0.30	0.00	0.00	0.10	1.30	2.70
bird	5.50	0.10	82.00	2.40	3.30	3.30	1.40	1.10	0.70	0.20
cat	2.60	0.40	9.10	57.00	5.80	14.70	4.20	3.50	1.40	1.30
deer	1.90	0.10	6.40	2.50	83.20	1.50	1.30	2.60	0.50	0.00
dog	0.90	0.20	4.70	8.20	2.70	78.10	0.90	3.80	0.20	0.30
frog	0.50	0.10	6.20	3.20	1.30	0.80	86.40	0.40	0.80	0.30
horse	0.80	0.20	2.60	2.00	3.40	1.60	0.10	88.60	0.30	0.40
ship	4.80	0.90	0.90	0.10	0.00	0.40	0.00	0.10	91.80	1.00
truck	3.30	5.70	0.60	0.40	0.10	0.10	0.00	0.40	1.50	87.90

3.1.2 Comparative Analysis of Net, Net2, Net3, and BMEnet

We compare Net, Net2, Net3, and BMEnet in terms of overall accuracy and class-wise accuracy.

Table 1, Model Summary

Metric	Net	Net2	Net3	BMEnet
Conv Layers	2	3	8	299
Parameters	62006	1468036	5743434	69493890
Accuracy (%)	53	69	83	76

Table 2, Class-wise Accuracy (%)

Class	Net (%)	Net2 (%)	Net3 (%)	BMEnet (%)
plane	50	69	89	80
car	85	76	94	87
bird	22	57	82	62
cat	31	58	57	50
deer	53	68	83	74
dog	38	41	78	68

Class	Net (%)	Net2 (%)	Net3 (%)	BMEnet (%)
frog	55	82	86	83
horse	70	67	88	83
ship	77	92	91	85
truck	53	77	87	87

Discussion:

1. Accuracy improves from Net → Net2 → Net3, but the gain is not linear with parameter count.
2. `bird` remains one of the most affected classes across model changes.
3. BMEnet is much deeper and much larger in parameter count; skip connections help optimization, but higher complexity does not always guarantee higher accuracy than a well-tuned smaller model.

3.2 Task 2: Skip Connections on CIFAR-10

Objective: We learn how skip connections can help mitigate the degradation problem and improve performance of deeper networks by checking out BMEnet on CIFAR-10 dataset.

3.2.1 Studying the SkipBlock

Looking at the code for Skipblock here:

```
class SkipBlock(nn.Module):
            """
            Class Path:   DLStudio  ->  BMEnet  ->  SkipBlock
            """
            def __init__(self, in_ch, out_ch, downsample=False,
skip_connections=True):
                super(DLStudio.BMEnet.SkipBlock, self).__init__()
                self.downsample = downsample
                self.skip_connections = skip_connections
                self.in_ch = in_ch
                self.out_ch = out_ch
                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.bn1 = nn.BatchNorm2d(in_ch)
                self.bn2 = nn.BatchNorm2d(out_ch)
```

```
                self.in2out  =  nn.Conv2d(in_ch, out_ch, 1)
                if downsample:
                    ##  Setting stride to 2 and kernel_size to 1
amounts to retaining every
                    ##  other pixel in the image --- which halves
the size of the image:
                    self.downsampler1 = nn.Conv2d(in_ch, in_ch, 1,
stride=2)
                    self.downsampler2 = nn.Conv2d(out_ch, out_ch,
1, stride=2)

            def forward(self, x):
                identity = x
                out = self.convo1(x)
                out = self.bn1(out)
                out = nn.functional.relu(out)
                out = self.convo2(out)
                out = self.bn2(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
```

We can describe the architecture of SkipBlock having the following layers.

- `Conv2d(in_ch → in_ch, 3×3, stride=1, padding=1)` - Convolutional Layer
- `BatchNorm2d(in_ch)` - Batch Normalization
- `ReLU` - Activation
- `Conv2d(in_ch → out_ch, 3×3, stride=1, padding=1)` - Convolutional Layer
- `BatchNorm2d(out_ch)` - Batch Normalization
- `ReLU` - Activation
- `Conv2d(in_ch → out_ch, 1×1)` projection (`in2out`) for channel matching when needed
- If `downsample=True` , two extra `1×1, stride=2` convs (`downsampler1` , `downsampler2`) to reduce spatial size

Skip connections are controlled by the `skip_connections` flag.

```
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)
```

How it works:

- The input is stored as `identity = x`.
- The conv path computes a transformed output (`out`), i.e., the residual branch.
- The block returns `out + skip_path` after matching inpuut/output channels and downsamplinng.

Three forward() cases in this implementation:

1. **`in_ch == out_ch` and no downsampling**
   Direct add: `out = out + identity`.
2. **`in_ch != out_ch` and no downsampling**
   If the input and output channels dims differ, then we need to match them before adding.
   Project skip with `in2out(identity)` to match channels, then add.
3. **`in_ch != out_ch` and `downsample=True`**
   Similar case but both paths are downsampled spatially, and skip channels are expanded by concatenation (`torch.cat((identity, identity), dim=1)`), then added to `out`.

This also corresponds to residual learning:

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

where `F(x)` is the two-conv transform, and `S(x)` is the skip path. In the simplest case, `S(x)=x`, giving the residual learning formaultion:

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

3.2.2 Running BMEnet on CIFAR-10

We run playing_with_skip_connections.py to train BMEnet on CIFAR-10 dataset and observe the results when depth is set to 32.

```
bme_net = dls.BMEnet(dls, skip_connections=True, depth=32)
```

Results:

The number of learnable parameters in the model: **69,493,890**
Overall accuracy of the network on the 10000 test images: 76 %

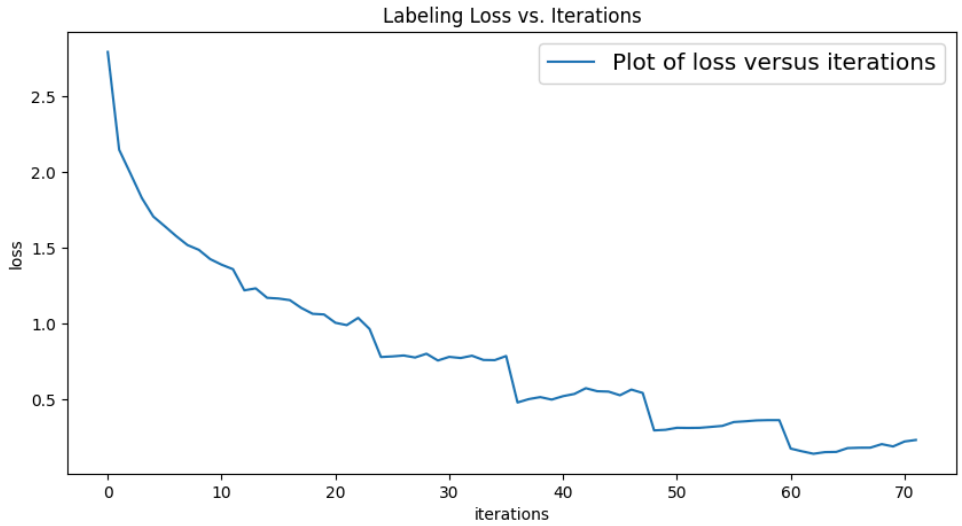
Confusion Matrix (rows = true class, columns = predicted class)

True \ Pred	plane	car	bird	cat	deer	dog	frog	horse	ship	truck
plane	80.60	1.60	2.90	2.20	2.60	0.70	0.60	0.70	4.60	3.50
car	1.30	87.30	0.20	0.50	0.40	0.40	0.70	0.10	1.70	7.40
bird	7.20	0.80	62.30	3.60	9.50	6.20	5.60	2.20	1.60	1.00
cat	2.10	1.10	5.60	50.80	8.80	18.20	7.20	3.90	1.20	1.10
deer	1.60	0.50	3.70	3.50	74.80	5.30	4.40	4.90	0.70	0.60
dog	0.80	0.40	3.80	11.60	5.80	68.40	2.70	4.80	1.00	0.70
frog	0.50	0.30	3.30	5.50	2.90	1.80	83.50	0.40	1.70	0.10
horse	1.00	0.30	2.60	1.70	4.30	4.90	0.70	83.20	0.30	1.00
ship	5.30	3.30	0.70	1.70	0.20	0.30	0.80	0.20	85.40	2.10
truck	1.90	5.30	0.50	0.90	0.70	0.50	0.50	0.80	1.80	87.10

Per-class Accuracy

Class	Accuracy (%)
plane	80
car	87
bird	62
cat	50
deer	74
dog	68
frog	83
horse	83
ship	85
truck	87

Training Loss curve:



3.2.3 BMEnet vs Deeper Networks Discussion

Table 1 and Table 2 shows the comparison of BMEnet with Net, Net2 and Net3 in terms of overall accuracy and class-wise accuracy.

Discussion:

1. The BMEnet with skip connections does not outperform Net3 in overall accuracy, despite net3 potentially suffering from degradation. This can be attributed to the fact that Net3 only has 8 conv layers, which is not deep enough to show significant degradation, while BMEnet has 299 conv layers.
2. The skip path gives a better direct route for the gradients to the earlier layers, which helps mitigate the vanishing gradient problem. Also as depth increases, plain nets often get higher training error but residual links reduce this degradation and layers can refine the features instead of learning it completely from scratch.
3. As explained before since the degradation problem is not severe in Net3, the skip connections in BMEnet do not provide a significant boost in accuracy. However, if we were to increase the depth of Net3 further (e.g., to 50 or 100 layers), we would likely see a more pronounced degradation without skip connections, thereby the CIFAR-10 classes do not show significant improvement with skip connections in BMEnet compared to Net3.

3.3 Task 3: Normalization Experiments

Objective: Implement and compare four normalization strategies (Batch Normalization, Instance Normalization, Layer Normalization, and Group Normalization) by modifying the SkipBlock and training on CIFAR-10.

3.3.1 Implementing NormSkipBlock

We implement the `NormSkipBlock` class to support different normalization strategies. The implementation allows us to switch between Batch Normalization, Instance Normalization, Layer Normalization, and Group Normalization.

Signature: NormSkipBlock(in_ch, out_ch, norm_type='batch', num_groups=8, downsample=False, skip_connections=True)

```
class NormSkipBlock(nn.Module):
    """A skip block with configurable normalization. Resuses the
implementation of SkipBlock from DLstudio"""
    def __init__(self, in_ch, out_ch,
norm_type='batch',num_groups=8,
downsample=False,skip_connections=True):
        self.downsample = downsample
        self.skip_connections = skip_connections
        self.in_ch = in_ch
        self.out_ch = out_ch
        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)
        self.norm2 = self._make_norm(norm_type, out_ch, num_groups)
        self.in2out  =  nn.Conv2d(in_ch, out_ch, 1)
        if downsample:
            ##  Setting stride to 2 and kernel_size to 1 amounts to
retaining every
            ##  other pixel in the image ---- which halves the size
of the image:
            self.downsampler1 = nn.Conv2d(in_ch, in_ch, 1,
stride=2)
            self.downsampler2 = nn.Conv2d(out_ch, out_ch, 1,
stride=2)

    """Factory method for creating normalization layers."""
    @staticmethod
    def _make_norm(norm_type, num_channels, num_groups=8):
        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
```

Explanations:

The major differences between BatchNorm, InstanceNorm, LayerNorm and GroupNorm are as follows:

- **Batch Normalization**: The ideas here is to normalize mean and std deviation across the batch dimension by computing mean and std deviation for each channel using the affine parameters(learnable parameters) to replace it.
- **Instance Normalization**: Normalizes each sample independently across the spatial dimensions. It computes mean and variance for each feature channel within each individual sample. Compared to BatchNorm, the eqns are the same but done not only per channel but also per instance(sample).
- **Layer Normalization**: Normalizes across all feature channels for each individual sample. It computes mean and variance across all the channels in contrast with BatchNorm and InstanceNorm. This is particularly useful for recurrent neural networks and transformers where batch sizes can be small or variable.
- **Group Normalization**: Divides the channels into groups and normalizes within each group. It computes mean and variance for each group of channels across the spatial dimensions. This can be more effective than BatchNorm when batch sizes are small.

The idea behind GroupNorm(1, C) for LayerNorm is that it treats all channels as a single group, effectively normalizing across the channel dimension for each sample, which is

the same as Layer Normalization.

3.3.2 Building NormBMEnet

We build the NormBMEnet by replacing the SkipBlock with NormSkipBlock in the BMEnet architecture. This allows us to experiment with different normalization strategies while keeping the overall architecture consistent.

```
class NormBMEnet(nn.Module):
    """ BMEnet that mirrors the architecture of BMEnet but uses
NormSkipBlock instead of SkipBlock. """
    def __init__(self, norm_type='batch', depth=8,
skip_connections=True, num_groups=8):
        super(NormBMEnet, self).__init__()

        self.depth = depth
        self.conv = nn.Conv2d(3, 64, 3, padding=1)

        self.skip64_arr = nn.ModuleList()
        for _ in range(self.depth):
            self.skip64_arr.append(
                NormSkipBlock(
                    64,
                    64,
                    norm_type=norm_type,
                    num_groups=num_groups,
                    downsample=False,
                    skip_connections=skip_connections,
                )
            )

        self.skip64to128ds = NormSkipBlock(
            64,
            128,
            norm_type=norm_type,
            num_groups=num_groups,
            downsample=True,
            skip_connections=skip_connections,
        )

        self.skip128_arr = nn.ModuleList()
        for _ in range(self.depth):
            self.skip128_arr.append(
                NormSkipBlock(
                    128,
                    128,
                    norm_type=norm_type,
                    num_groups=num_groups,
                    downsample=False,
                    skip_connections=skip_connections,
```

```
                )
            )

        self.skip128to256ds = NormSkipBlock(
            128,
            256,
            norm_type=norm_type,
            num_groups=num_groups,
            downsample=True,
            skip_connections=skip_connections,
        )

        self.skip256_arr = nn.ModuleList()
        for _ in range(self.depth):
            self.skip256_arr.append(
                NormSkipBlock(
                    256,
                    256,
                    norm_type=norm_type,
                    num_groups=num_groups,
                    downsample=False,
                    skip_connections=skip_connections,
                )
            )

        self.fc1 = nn.Linear(256 * 8 * 8, 1000)
        self.fc2 = nn.Linear(1000, 10)

    def forward(self, x):
        x = F.relu(self.conv(x))

        for skip64 in self.skip64_arr:
            x = skip64(x)

        x = self.skip64to128ds(x)

        for skip128 in self.skip128_arr:
            x = skip128(x)

        x = self.skip128to256ds(x)

        for skip256 in self.skip256_arr:
            x = skip256(x)

        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x


# Invoking code
def train_normbmenet_cifar10(
```

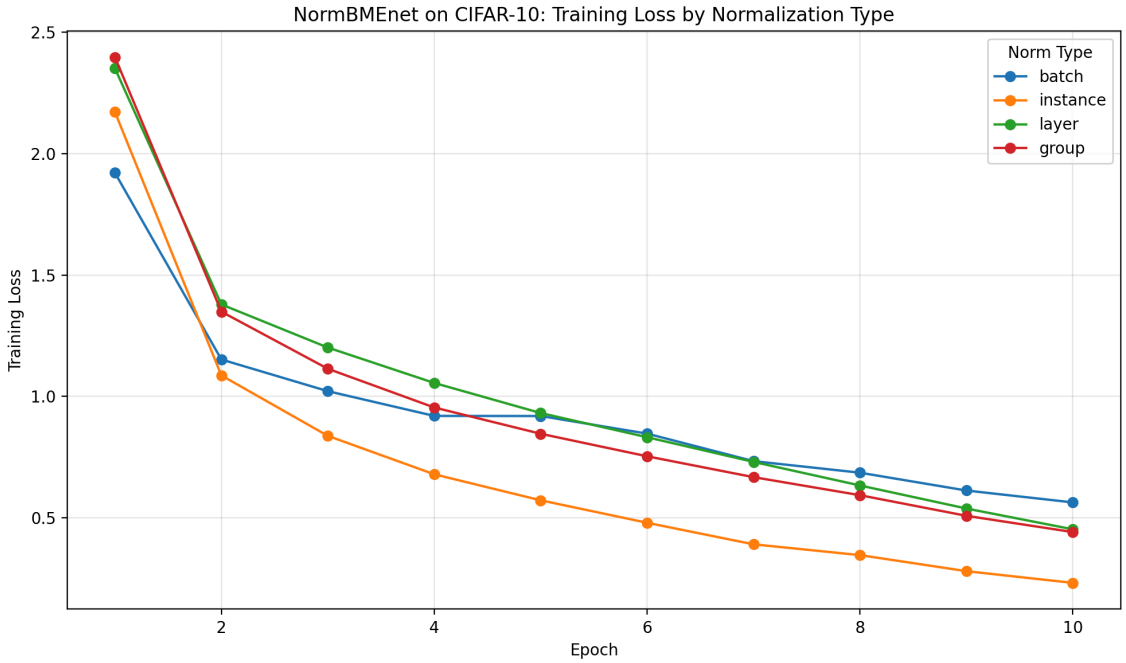
```
    norm_types=None,
    batch_size=64,
    epochs=10,
    lr=1e-3,
    depth=8,
    skip_connections=True,
    num_groups=8,
    plot_path="normbmenet_cifar10_loss_overlay.png",
):
    """
    Train NormBMEnet on CIFAR-10 for multiple normalization types
with identical hyperparameters.
    """


    if norm_types is None:
        norm_types = ["batch", "instance", "layer", "group"]

    device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
    print(f"Using device: {device}")

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

    train_dataset = datasets.CIFAR10(
        root="./data/CIFAR-10", train=True, download=True,
transform=transform
    )
    test_dataset = datasets.CIFAR10(
        root="./data/CIFAR-10", train=False, download=True,
transform=transform
    )

    train_loader = DataLoader(
        train_dataset, batch_size=batch_size, shuffle=True,
num_workers=2
    )
    test_loader = DataLoader(
        test_dataset, batch_size=batch_size, shuffle=False,
num_workers=2
    )

    all_losses = {}
    summary = {}

    for norm_type in norm_types:
        model = NormBMEnet(
```

```
                norm_type=norm_type,
                depth=depth,
                skip_connections=skip_connections,
                num_groups=num_groups,
            ).to(device)

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

            metrics = {
                "train_loss": [],
                "train_accuracy": [],
                "val_loss": [],
                "val_accuracy": [],
            }

            for epoch in range(epochs):
                train_loss, train_acc = train_one_epoch(
                    model, train_loader, criterion, optimizer, device
                )
                val_loss, val_acc = validate(model, test_loader,
criterion, device)

                metrics["train_loss"].append(train_loss)
                metrics["train_accuracy"].append(train_acc)
                metrics["val_loss"].append(val_loss)
                metrics["val_accuracy"].append(val_acc)

                print(
                    f"[{norm_type}] Epoch {epoch + 1}/{epochs} - "
                    f"Train Loss: {train_loss:.4f}, Train Acc:
{train_acc:.2f}%, "
                    f"Test Loss: {val_loss:.4f}, Test Acc:
{val_acc:.2f}%"
                )

            cm = compute_confusion_matrix(model, test_loader,
num_classes=10, device=device)
            test_acc = 100.0 * np.trace(cm) / np.sum(cm)
            params = sum(p.numel() for p in model.parameters() if
p.requires_grad)

            all_losses[norm_type] = metrics["train_loss"]
            summary[norm_type] = {"params": params, "test_acc":
test_acc}

        plot_overlaid_training_losses(
            all_losses,
            title="NormBMEnet on CIFAR-10: Training Loss by
Normalization Type",
            save_path=plot_path,
        )
```

```
    print("\nSummary:")
    print(f"{'Norm Type':<16}{'Parameters':>14}{'Test Accuracy
(%)':>22}")
    print("-" * 52)
    for nt in norm_types:
        p = summary[nt]["params"]
        a = summary[nt]["test_acc"]
        pretty_name = {
            "batch": "Batch Norm",
            "instance": "Instance Norm",
            "layer": "Layer Norm",
            "group": "Group Norm (G=8)",
        }[nt]
        print(f"{pretty_name:<16}{p:>14,}{a:>21.2f}")

    return all_losses, summary
```

Results:

Training loss curves for different normalization strategies:



Summary of the accuracies:

Norm Type	Parameters	Test Accuracy (%)
Batch Norm	30,195,330	72.33
Instance Norm	30,195,330	77.63
Layer Norm	30,195,330	72.74
Group Norm (G=8)	30,195,330	72.35

Observations:

1. Instance Normalization outperforms the other normalization types by converging the fastest and achieving the highest test accuracy of 77.63%. This suggests that normalizing each sample independently across spatial dimensions may be more effective for this architecture and dataset.
2. Instance Normalization behaves differently from Batch Norm since it normalizes each sample independently per channel, in our training setup it performs much more smoothly than batch norm which slightly increases in training loss at epoch 5 before converging again, while instance norm shows a steady decrease in training loss throughout the epochs.
3. Generally groupnorm is a good comprimise between instance norm and layer norm since it lies between the two extremes(Layer norm is G=1 and Instance norm is roughly G=C), howeever in our case group norm with G=8 is closer to layer norm.

3.4 Task 4: Skip Connection Network for LVIS

Objective: Design a deep skip-connection-based CNN and train it on the LVIS multi-instance same-object dataset from HW4.

3.4.1 Network Architecture: LVISSkipNet

We use 24 skipblocks which would give a total of atleast 48 conv layers in the network. For the skipblocks, we are using the NormSkipBlock implementation from the previous task with instance normalization. The architecture consists of three stages of skip blocks with transition blocks in between that downsample the spatial dimensions and increase the channel dimensions.

Code Snippet:

```
class LVISSkipNet(nn.Module):
    """A skip-connection-based CNN for the LVIS multi-instance
same-object dataset. We use NormSkipBlock as the building block for
this architecture and we use 20 skipblocks(so to get a min of 40
learnable layers)"""
    def __init__(self, num_classes=5):
        super(LVISSkipNet, self).__init__()
        self.norm_type = 'instance'
        self.conv_initial = nn.Conv2d(3, 64, 3, padding=1)
        self.bn_initial = nn.BatchNorm2d(64)
        #Stage 1 SkipBlocks
        self.skip64_arr = nn.ModuleList()
        for _ in range(8):
```

```
            self.skip64_arr.append(
                NormSkipBlock(
                    64,
                    64,
                    norm_type=self.norm_type,
                    num_groups=8,
                    downsample=False,
                    skip_connections=True
                )
            )
        #Transition block 1: 64 -> 128 with downsampling
        self.skip64_to_128 = NormSkipBlock(
            64,
            128,
            norm_type=self.norm_type,
            num_groups=8,
            downsample=True,
            skip_connections=True
        )
        #Stage 2 SkipBlocks
        self.skip128_arr = nn.ModuleList()
        for _ in range(8):
            self.skip128_arr.append(
                NormSkipBlock(
                    128,
                    128,
                    norm_type=self.norm_type,
                    num_groups=8,
                    downsample=False,
                    skip_connections=True
                )
            )
        #Transition block 2: 128 -> 256 with downsampling
        self.skip128_to_256 = NormSkipBlock(
            128,
            256,
            norm_type=self.norm_type,
            num_groups=8,
            downsample=True,
            skip_connections=True
        )
        #Stage 3 SkipBlocks
        self.skip256_arr = nn.ModuleList()
        for _ in range(8):
            self.skip256_arr.append(
                NormSkipBlock(
                    256,
                    256,
                    norm_type=self.norm_type,
                    num_groups=8,
                    downsample=False,
                    skip_connections=True
```

```
                )
            )
        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)))
        for skip64 in self.skip64_arr:
            x = skip64(x)
        x = self.skip64_to_128(x)
        for skip128 in self.skip128_arr:
            x = skip128(x)
        x = self.skip128_to_256(x)
        for skip256 in self.skip256_arr:
            x = skip256(x)
        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        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) or isinstance(m, nn.Linear))

net = LVISSkipNet(num_classes=5)
print(f"LVISSkipNet architecture:\n{net}")
print(f"Total trainable parameters: {net.count_parameters():,}")
print(f"Total learnable layers (Conv2d + Linear):
{net.count_layers()}")
```

Results:

Architecture for LVISSkipNet and the number of learnable parameters and layers:

```
LVISSkipNet(
  (conv_initial): Conv2d(3, 64, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1))
  (bn_initial): BatchNorm2d(64, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
  (skip64_arr): ModuleList(
    (0-7): 8 x NormSkipBlock(
      (convo1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1,
1), padding=(1, 1))
      (convo2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1,
1), padding=(1, 1))
      (norm1): InstanceNorm2d(64, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=False)
```

```
      (norm2): InstanceNorm2d(64, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=False)
      (in2out): Conv2d(64, 64, kernel_size=(1, 1), stride=(1,
1))
    )
  )
  (skip64_to_128): NormSkipBlock(
    (convo1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1,
1), padding=(1, 1))
    (convo2): Conv2d(64, 128, kernel_size=(3, 3), stride=(1,
1), padding=(1, 1))
    (norm1): InstanceNorm2d(64, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=False)
    (norm2): InstanceNorm2d(128, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=False)
    (in2out): Conv2d(64, 128, kernel_size=(1, 1), stride=(1,
1))
    (downsampler1): Conv2d(64, 64, kernel_size=(1, 1),
stride=(2, 2))
    (downsampler2): Conv2d(128, 128, kernel_size=(1, 1),
stride=(2, 2))
  )
  (skip128_arr): ModuleList(
    (0-7): 8 x NormSkipBlock(
      (convo1): Conv2d(128, 128, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1))
      (convo2): Conv2d(128, 128, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1))
      (norm1): InstanceNorm2d(128, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=False)
      (norm2): InstanceNorm2d(128, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=False)
      (in2out): Conv2d(128, 128, kernel_size=(1, 1), stride=
(1, 1))
    )
  )
  (skip128_to_256): NormSkipBlock(
    (convo1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1,
1), padding=(1, 1))
    (convo2): Conv2d(128, 256, kernel_size=(3, 3), stride=(1,
1), padding=(1, 1))
    (norm1): InstanceNorm2d(128, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=False)
    (norm2): InstanceNorm2d(256, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=False)
    (in2out): Conv2d(128, 256, kernel_size=(1, 1), stride=(1,
1))
    (downsampler1): Conv2d(128, 128, kernel_size=(1, 1),
stride=(2, 2))
    (downsampler2): Conv2d(256, 256, kernel_size=(1, 1),
stride=(2, 2))
  )
```

```
  (skip256_arr): ModuleList(
    (0-7): 8 x NormSkipBlock(
      (convo1): Conv2d(256, 256, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1))
      (convo2): Conv2d(256, 256, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1))
      (norm1): InstanceNorm2d(256, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=False)
      (norm2): InstanceNorm2d(256, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=False)
      (in2out): Conv2d(256, 256, kernel_size=(1, 1), stride=
(1, 1))
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=256, out_features=5, bias=True)
)

Total trainable parameters: 13,801,733
Total learnable layers (Conv2d + Linear): 84
```

Channel progression: In LVISSkipNet, the feature channels follow a staged pattern: 3 → 64 via the initial conv. Stage 1: stays at 64 channels through 8 skip blocks. Transition 1: 64 → 128 using a downsampling skip block. Stage 2: stays at 128 channels through 8 skip blocks. Transition 2: 128 → 256 using a downsampling skip block. Stage 3: stays at 256 channels through 8 skip blocks.

Downsampling strategy: Spatial downsampling happens only in the two transition blocks (skip64_to_128 and skip128_to_256), not in the repeated same-channel blocks. Each downsampling block uses NormSkipBlock(with param downsample=True), where both residual and skip paths are reduced with convolutions before addition.

3.4.2 Training LVISSkipNet on LVIS Multi-instance Same-object Dataset.

We perform training of the LVISSkipNet on the LVIS multi-instance same-object dataset. The dataset consists of 5 classes with at least 300 images per class, and we use a standard train-test split of 80-20. We train the model for 10 epochs using a batch size of 32 and an Adam optimizer with a learning rate of 1e-3.

NOTE: Compared to Hw4, few categories have been changed to ensure that we have enough images for training and validation after the train-val split. The selected categories are: 'person', 'banana', 'broccoli', 'suitcase', 'sheep'. In the previous assignment the categories were:'person', 'book','kite','apple','chair'. Due to the change, the results may not be directly comparable to the previous assignment, but the training and evaluation process remains consistent.

Dataset verification:

```
def test_create_multi_same_dataset():
    lvis = LVIS(LVIS_TRAIN_JSON_PATH)
    train_category_info = get_category_info(lvis)
    dataset_type =  'multi_instance_same'
    all_datasets_info = {}
    min_train_images = 300
    min_val_images = 50
    max_train_ratio = 1.2
    #We delete the output directory if it already exists to avoid
confusion with old data.
    if os.path.exists(OUTPUT_DIR):
        import shutil
        shutil.rmtree(OUTPUT_DIR)
    output_dir = os.path.join(OUTPUT_DIR, dataset_type)
    all_datasets_info[dataset_type] = {}
    cat_images_data = {}
    raw_by_class = {}
    for cat_name in SELECTED_CATEGORIES:
        multi_same_train = create_multi_same_dataset(
            lvis,
            cat_name,
            train_category_info,
            min_instances=2,
            min_total_area_ratio=0.10
        )
        random.shuffle(multi_same_train)
        raw_by_class[cat_name] = multi_same_train

    train_pools = {}
    val_pools = {}
    for cat_name, raw_samples in raw_by_class.items():
        total_cat_images = len(raw_samples)
        val_count = max(min_val_images, int(0.2 *
total_cat_images))
        train_count = total_cat_images - val_count
        if train_count < min_train_images or val_count <
min_val_images:
            raise ValueError(
                f"Not enough original images for '{cat_name}': "
                f"train={train_count}, val={val_count}"
            )
        val_pools[cat_name] = raw_samples[:val_count]
        train_pools[cat_name] = raw_samples[val_count:]

    min_train_count = min(len(samples) for samples in
train_pools.values())
    max_allowed_train = int(min_train_count * max_train_ratio)

    for cat_name in SELECTED_CATEGORIES:
        train_pool = train_pools[cat_name]
```

```
        if len(train_pool) > max_allowed_train:
            train_dataset = random.sample(train_pool,
max_allowed_train)
        else:
            train_dataset = train_pool

        val_dataset = val_pools[cat_name]

        save_to_dir(train_dataset, LVIS_IMAGES_DIR,
os.path.join(output_dir, 'train', cat_name))
        save_to_dir(val_dataset, LVIS_IMAGES_DIR,
os.path.join(output_dir, 'val', cat_name))

        cat_images_data[cat_name] = {
            'train': len(train_dataset),
            'val': len(val_dataset),
            'augmented_images': 0
        }
    all_datasets_info[dataset_type] = cat_images_data
    print(f"\n--- Dataset: {dataset_type} ---")
    print(f"{'Category':20s} {'Train Images':15s} {'Val
Images':15s} {'Total Augmented Images Used':20s}")
    for cat_name, splits in cat_images_data.items():
        print(f"{cat_name:20s} {splits['train']:<15d}
{splits['val']:<15d} {splits['augmented_images']:<20d}")
```

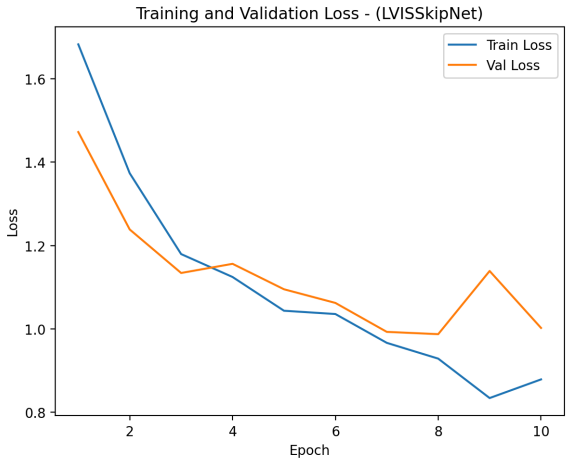
Results of dataset verification:

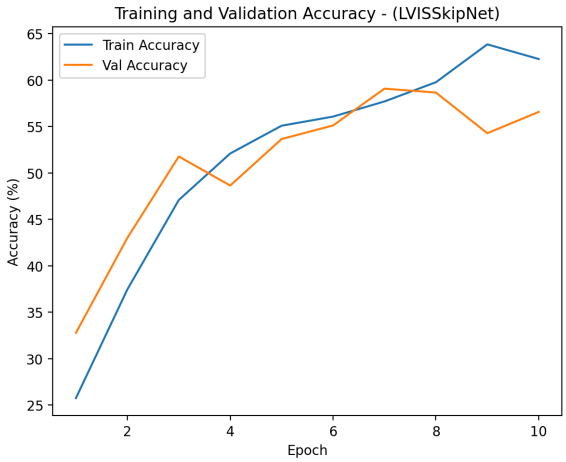
```
--- Dataset: multi_instance_same ---
Category             Train Images    Val Images      Total
Augmented Images Used
person               380             100             0
banana               380             100             0
broccoli             380             100             0
suitcase             380             100             0
sheep                317             79              0
```

We can observe that each category a min of 317 training images and 79 validation images, and the max is 1.2 times the min category as per the requirements. Also, no augmented images were used since we had enough original images for each category. Totally we have 1837 training images and 479 validation images across the 5 categories.

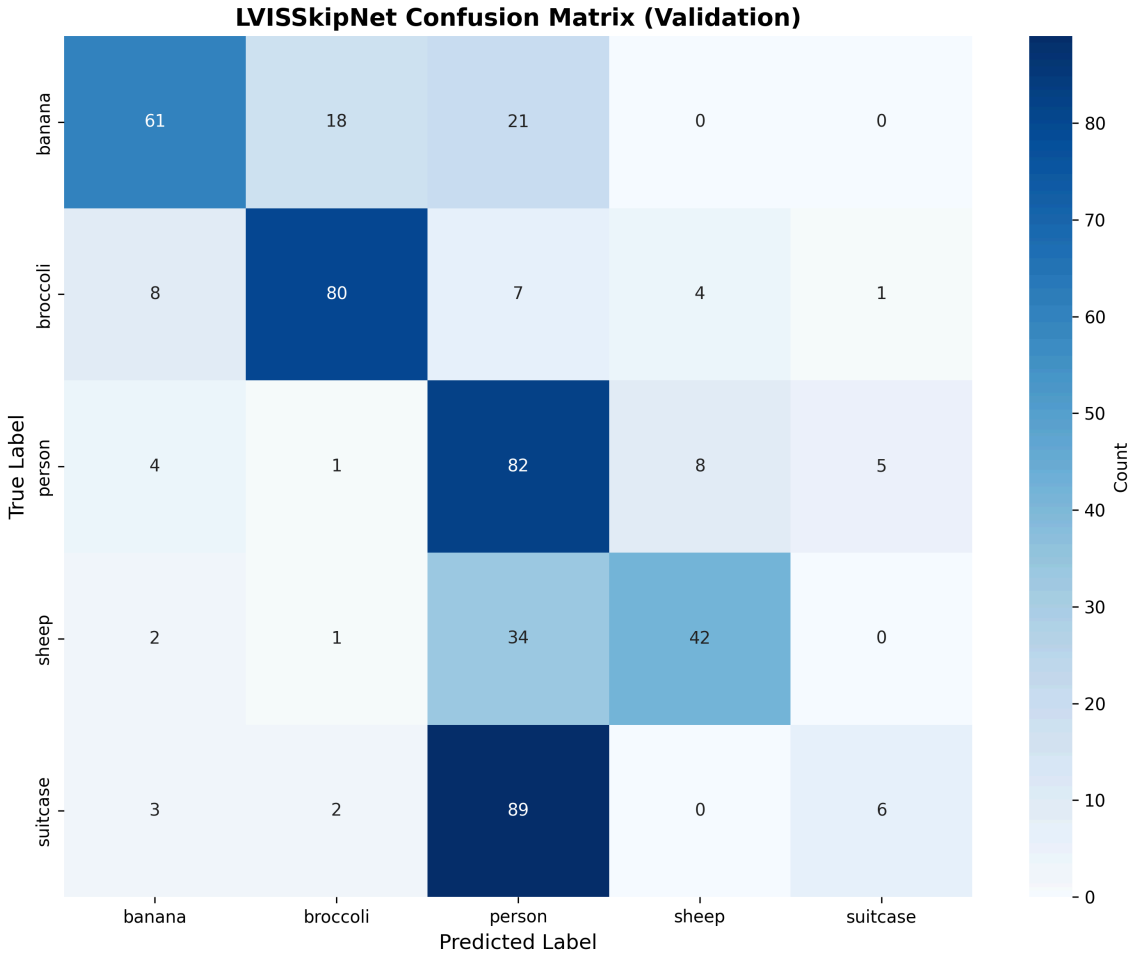
Training Results:

Training and Validation loss/accuracy curves for LVISSkipNet on the LVIS multi-instance same-object dataset:

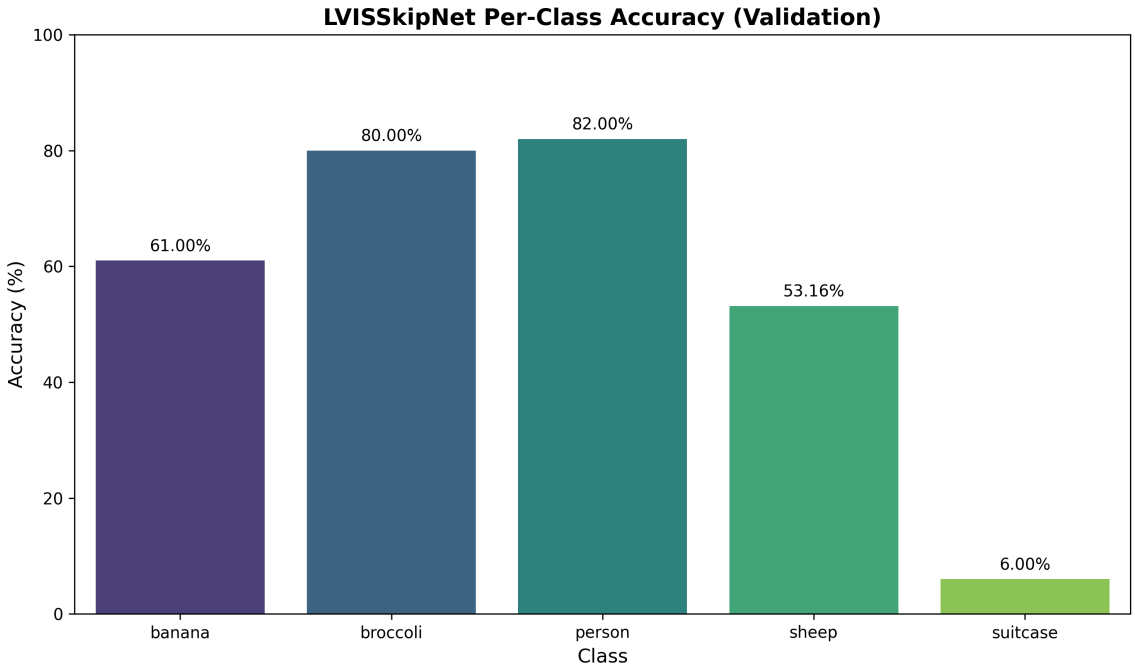




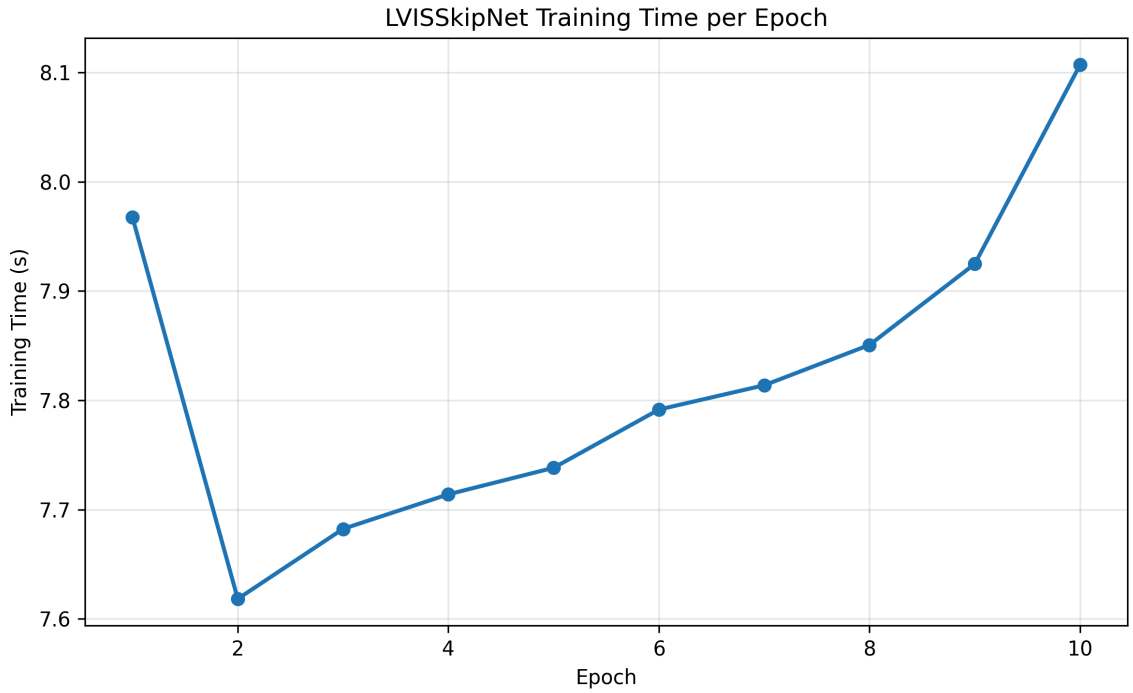
Confusion Matrix:



Per class accuracy:



Training time per epoch:



3.5 Task 5: Feature Map Visualization

Objective: We visualize the conv filters and intermediate feature maps for LVISNet vs LVISSkipNet to understand the impact of skip connections on feature learning and depth of the network.

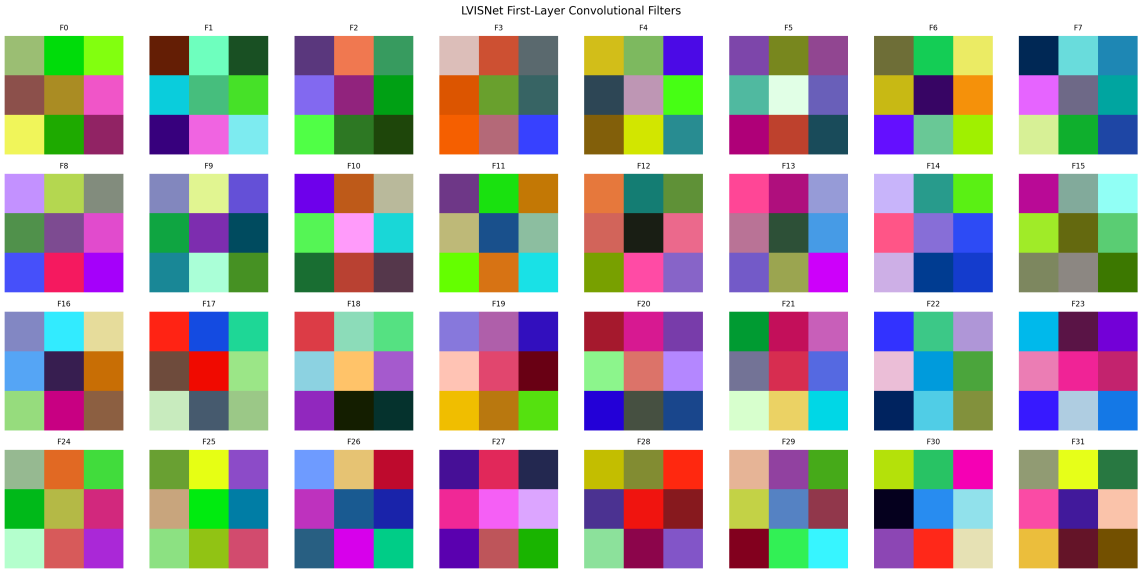
Code Snippet:

```
def
visualize_first_layer_filters(model,first_layer_name=None,max_filters=
        """
        Visualize first-layer convolution filters in a grid.

        For RGB filters, transpose to (H,W,3) and normalize each
filter to [0,1].
        """
        layer_name, filters = extract_first_layer_filters(model,
first_layer_name=first_layer_name)

        num_filters = min(max_filters, filters.shape[0])
        rows = int(math.ceil(num_filters / cols))

        if title is None:
                title = f"{model.__class__.__name__} first-layer
filters ({layer_name})"

        plt.figure(figsize=(2.2 * cols, 2.2 * rows))
        for idx in range(num_filters):
                ax = plt.subplot(rows, cols, idx + 1)
                kernel = filters[idx]

                if kernel.shape[0] >= 3:
                        rgb = kernel[:3].permute(1, 2, 0).numpy()
                        rgb = _normalize_channel(rgb)
                        ax.imshow(rgb)
                else:
                        gray = kernel[0].numpy()
                        gray = _normalize_channel(gray)
                        ax.imshow(gray, cmap="gray")

                ax.set_title(f"F{idx}", fontsize=8)
                ax.axis("off")

        plt.suptitle(title)
        plt.tight_layout()

        if save_path:
                save_dir = os.path.dirname(save_path)
                if save_dir:
                        os.makedirs(save_dir, exist_ok=True)
                plt.savefig(save_path, dpi=250,
bbox_inches="tight")
                print(f"First-layer filter visualization saved to
{save_path}")

        plt.show()
        return filters
```

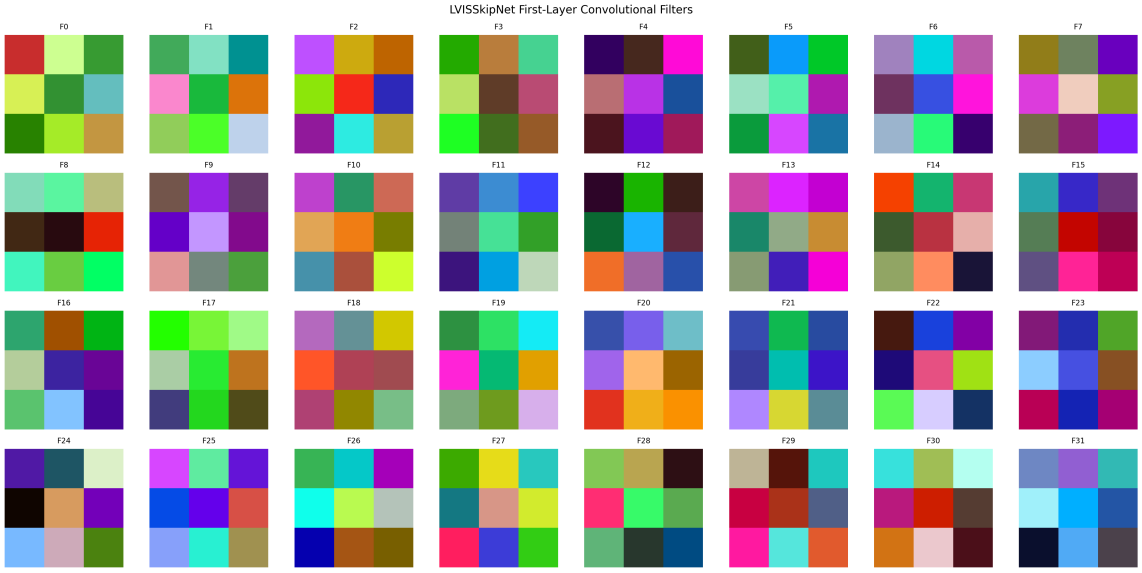
```
def visualize_lvis_first_layer_filters(lvisnet_model,
lvisskipnet_model, save_dir=None, max_filters=32):
        """
        Generate report-ready first-layer filter visualizations for
LVISNet and LVISSkipNet.
        """
        lvisnet_save = None
        lvisskipnet_save = None
        if save_dir:
                os.makedirs(save_dir, exist_ok=True)
                lvisnet_save = os.path.join(save_dir,
"LVISNet_first_layer_filters.png")
                lvisskipnet_save = os.path.join(save_dir,
"LVISSkipNet_first_layer_filters.png")

        visualize_first_layer_filters(
                model=lvisnet_model,
                first_layer_name="conv1",
                max_filters=max_filters,
                title="LVISNet First-Layer Convolutional Filters",
                save_path=lvisnet_save,
        )

        visualize_first_layer_filters(
                model=lvisskipnet_model,
                first_layer_name="conv_initial",
                max_filters=max_filters,
                title="LVISSkipNet First-Layer Convolutional
Filters",
                save_path=lvisskipnet_save,
        )

lvisnet_model = get_trained_lvis_model("lvisnet")
lvisskipnet_model = get_trained_lvis_model("lvis_skipnet")
report_dir = os.path.join("./lvis_datasets", "multi_instance_same",
"plots")
visualize_lvis_first_layer_filters(
    lvisnet_model=lvisnet_model,
    lvisskipnet_model=lvisskipnet_model,
    save_dir=report_dir,
    max_filters=32,
)
```

Results:

First layer convolutional filters for LVISNet:



First layer convolutional filters for LVISSkipNet:



The filters for both looks different when comparing between LVISNet and LVISSkipNet. Since its the first layer visualization, we can see that the filters are mostly edge detectors: Example-> F18(LVISNet) and F8(LVISSkipNet). We see some color filters too(F23, F31) for LVISSkipnet.

3.5.2 Visualizing Intermediate Feature Maps

For the visualization we extract three layers: the early conv layer, a middle skip block conv layer and a later skip block conv layer. The method will work for both LVISNet and LVISSkipNet since they share the same overall architecture. Code Snippet:

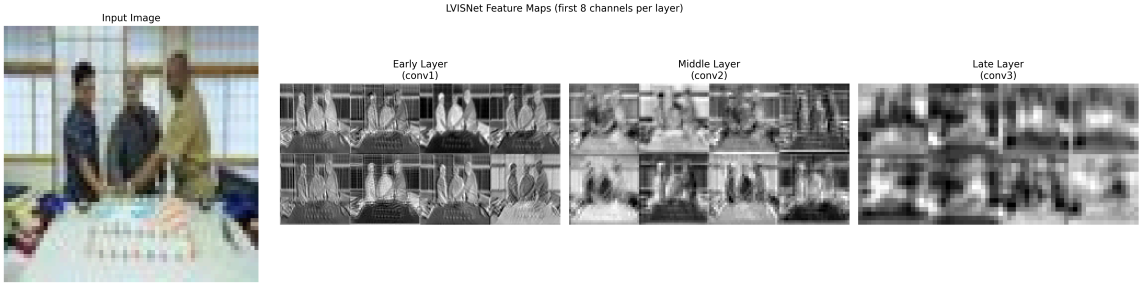
```
def
visualize_feature_depths(model,input_image,layer_names,layer_display_na
```

```
        """
        Show one input image and feature maps from
early/middle/late layers of any model.
        Each layer visualization is a grayscale grid of the first N
channels, with
        per-channel independent normalization.
        """
        if layer_display_names is None:
                layer_display_names = [
                        "Early Layer",
                        "Middle Layer",
                        "Late Layer",
                ]

        if len(layer_names) != 3:
                raise ValueError("layer_names must contain exactly
3 layers: early, middle, late.")

        feature_maps = get_feature_maps_with_hooks(
                model=model,
                input_image=input_image,
                layer_names=layer_names,
                device=device,
        )

        input_vis = _to_display_input(input_image)
        grids = []
        for name in layer_names:
                fmap = feature_maps[name]
                if fmap.dim() != 4:
                        raise ValueError(f"Expected 4D feature map
for layer '{name}', got shape {tuple(fmap.shape)}")
                grids.append(_feature_grid_image(fmap[0],
max_channels=max_channels, cols=cols))

        fig, axes = plt.subplots(1, 4, figsize=(20, 5))
        axes[0].imshow(input_vis)
        axes[0].set_title("Input Image")
        axes[0].axis("off")

        for idx in range(3):
                axes[idx + 1].imshow(grids[idx], cmap="gray")
                axes[idx + 1].set_title(f"
{layer_display_names[idx]}\n({layer_names[idx]})")
                axes[idx + 1].axis("off")

        model_name = model.__class__.__name__
        plt.suptitle(f"{model_name} Feature Maps (first
{max_channels} channels per layer)")
        plt.tight_layout()

        if save_dir:
```

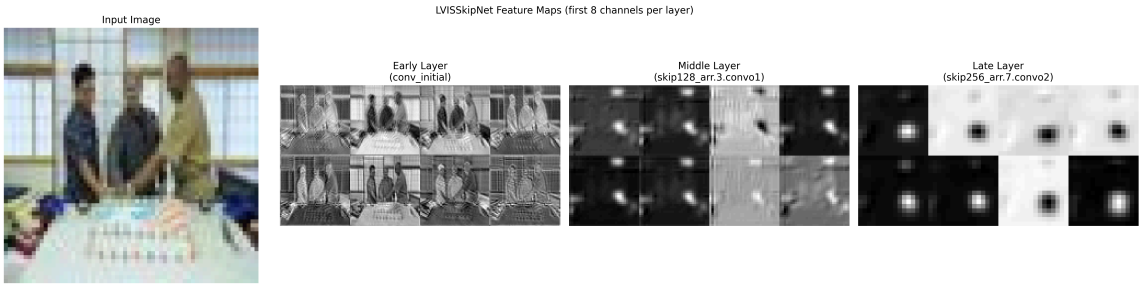
```
                save_path = os.path.join(save_dir, f"
{model.__class__.__name__}_feature_depths.png")
                os.makedirs(save_dir, exist_ok=True)
                plt.savefig(save_path, dpi=250,
bbox_inches="tight")
                print(f"Feature depth visualization saved to
{save_path}")

        plt.show()
        return feature_maps

#Calling code

lvisnet_model = get_trained_lvis_model("lvisnet")
lvisskipnet_model = get_trained_lvis_model("lvis_skipnet")
report_dir = os.path.join("./lvis_datasets", "multi_instance_same",
"plots")
sample_image = get_sample_image_from_train_for_cat("person")
#We visualize feature maps for a sample image from training set for
both the models. We save the feature maps in the report directory
as well.
visualize_feature_depths(
    model=lvisnet_model,
    layer_names= ["conv1", "conv2", "conv3"],
    input_image=sample_image,
    save_dir=report_dir,
    device="cuda" if torch.cuda.is_available() else "cpu",
)
visualize_feature_depths(
    model=lvisskipnet_model,
    layer_names= ["conv_initial", "skip128_arr.3.convo1",
"skip256_arr.7.convo2"],
    input_image=sample_image,
    save_dir=report_dir,
    device="cuda" if torch.cuda.is_available() else "cpu",
)
```

Results

Feature maps for LVISNet:



Feature maps for LVISSkipNet:



Observations:

1. On comparing the feature maps for LVISNet and LVISSkipNet, the former shows smoother and scene like activations through the depths, whereas for the skipnet, it transitions to high contrast responses like a hotpot type, which means it is showing signs of selective abstraction.
2. Deeper feature maps are more discriminative and their late maps looks less like texture and more class specific activations. Whereas shallow nets are usually more diffuse.
3. In mulitchannel convolution, each filter computes weighted sums across the RGB channels so as to detect color opponent structures along with intensity edges. Early layers capture channel interactions while the deeper ones abstract object part cues.

3.6 Task 6: Final Comparative Analysis.

Objective: We synthesize all the results into a comparison across the architectures and norm strategies.

3.6.1 Confusion Matrices

1. CIFAR-10

Baseline Confusion matrix for Net(from HW4):

True \ Pred	plane	car	bird	cat	deer	dog	frog	horse	ship	truck
plane	50.50	7.10	1.80	3.20	3.80	0.90	0.80	2.00	25.30	4.60
car	1.40	85.50	0.30	0.60	0.30	0.20	0.40	1.00	3.60	6.70
bird	8.00	2.70	22.70	10.70	21.20	6.50	6.70	9.20	7.90	4.40
cat	2.20	6.30	3.00	31.10	10.90	13.40	8.00	9.70	7.60	7.80
deer	2.70	3.50	4.30	6.10	53.10	3.00	5.80	15.40	3.80	2.30
dog	0.90	2.40	2.70	19.00	8.80	38.10	2.50	17.40	3.70	4.50

True \ Pred	plane	car	bird	cat	deer	dog	frog	horse	ship	truck
frog	0.40	5.10	2.20	8.20	13.40	1.40	55.60	3.50	3.30	6.90
horse	1.50	1.30	1.40	4.00	4.60	4.60	0.90	70.50	2.40	8.80
ship	4.50	11.60	0.60	0.30	0.60	1.20	0.10	0.60	77.10	3.40
truck	1.30	31.80	0.50	1.10	0.70	0.40	0.50	2.10	8.00	53.60

Confusion matrix for Net3:

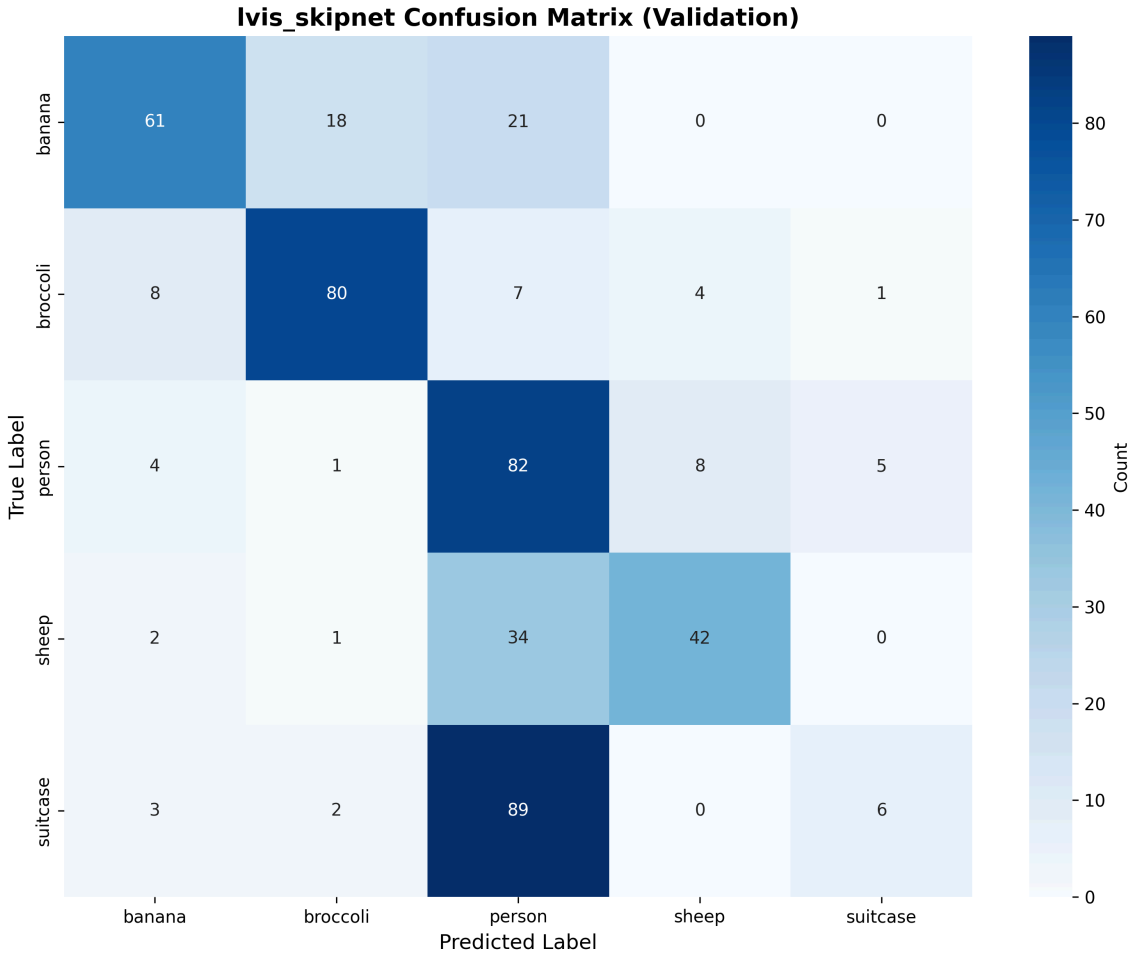
True \ Pred	plane	car	bird	cat	deer	dog	frog	horse	ship	truck
plane	89.70	0.80	2.80	0.60	0.60	0.10	0.10	1.40	2.90	1.00
car	1.30	94.00	0.20	0.10	0.30	0.00	0.00	0.10	1.30	2.70
bird	5.50	0.10	82.00	2.40	3.30	3.30	1.40	1.10	0.70	0.20
cat	2.60	0.40	9.10	57.00	5.80	14.70	4.20	3.50	1.40	1.30
deer	1.90	0.10	6.40	2.50	83.20	1.50	1.30	2.60	0.50	0.00
dog	0.90	0.20	4.70	8.20	2.70	78.10	0.90	3.80	0.20	0.30
frog	0.50	0.10	6.20	3.20	1.30	0.80	86.40	0.40	0.80	0.30
horse	0.80	0.20	2.60	2.00	3.40	1.60	0.10	88.60	0.30	0.40
ship	4.80	0.90	0.90	0.10	0.00	0.40	0.00	0.10	91.80	1.00
truck	3.30	5.70	0.60	0.40	0.10	0.10	0.00	0.40	1.50	87.90

Confusion Matrix for BMENet:

True \ Pred	plane	car	bird	cat	deer	dog	frog	horse	ship	truck
plane	80.60	1.60	2.90	2.20	2.60	0.70	0.60	0.70	4.60	3.50
car	1.30	87.30	0.20	0.50	0.40	0.40	0.70	0.10	1.70	7.40
bird	7.20	0.80	62.30	3.60	9.50	6.20	5.60	2.20	1.60	1.00
cat	2.10	1.10	5.60	50.80	8.80	18.20	7.20	3.90	1.20	1.10
deer	1.60	0.50	3.70	3.50	74.80	5.30	4.40	4.90	0.70	0.60
dog	0.80	0.40	3.80	11.60	5.80	68.40	2.70	4.80	1.00	0.70
frog	0.50	0.30	3.30	5.50	2.90	1.80	83.50	0.40	1.70	0.10
horse	1.00	0.30	2.60	1.70	4.30	4.90	0.70	83.20	0.30	1.00
ship	5.30	3.30	0.70	1.70	0.20	0.30	0.80	0.20	85.40	2.10
truck	1.90	5.30	0.50	0.90	0.70	0.50	0.50	0.80	1.80	87.10

2. LVIS:

LVISSkipNet Confusion Matrix:



3.6.2 LVISNet vs LVISSkipNet Performance Comparison

Network	Parameters	Val Accuracy (%)
LVISNet (HW4)	2,224,645	56.99
LVISSkipNet	13,801,733	56.58

Discussion

1. LVISSkipNet is much deeper and has substantially higher capacity than LVISNet (about 6 timesmore parameters), and it achieves higher training accuracy (62.28% vs 56.45%). However, this improvement does not transfer to validation accuracy in this run: LVISSkipNet obtains 56.58% while LVISNet reaches 56.99%. Therefore, for this dataset split and training setup, added depth with skip connections does not produce a generalization gain over the shallower baseline.

2. From the normalization experiments in Task 3, Instance Normalization gave the strongest and most stable CIFAR-10 performance. Since our LVISSkipNet was built

using the NormSkipBlock with instance normalization, this is still a reasonable normalization choice for deeper skip-based models, especially with moderate batch sizes. The current LVIS result suggests that normalization alone is not enough to guarantee higher validation accuracy when model capacity is increased; augmentation, and tuning are still needed.

3. The homework shows that depth and skip connections improve optimization (better training behavior), but better optimization does not automatically imply better validation accuracy. Model capacity must be matched to dataset size and diversity, and normalization helps training stability but is only one part of the final performance equation.

3.6.3 Master Summary Table

Summary Table

Network	Dataset	Layers	Parameters	Accuracy (%)
Net (HW4)	CIFAR-10	2 (Conv)	62,006	53.00
Net2 (HW4)	CIFAR-10	3 (Conv)	1,468,036	69.00
Net3	CIFAR-10	8 (Conv)	5,743,434	83.00
BMEnet	CIFAR-10	299 (Conv)	69,493,890	76.00
NormBMEnet (Batch Norm)	CIFAR-10	85 (Conv + Linear)	30,195,330	72.33
NormBMEnet (Instance Norm)	CIFAR-10	85 (Conv + Linear)	30,195,330	77.63
NormBMEnet (Layer Norm)	CIFAR-10	85 (Conv + Linear)	30,195,330	72.74
NormBMEnet (Group Norm, G=8)	CIFAR-10	85 (Conv + Linear)	30,195,330	72.35
LVISNet (HW4)	LVIS	6 (Conv + Linear)	2,224,645	56.99
LVISSkipNet	LVIS	84 (Conv + Linear)	13,801,733	56.58

Note: The accuracy is test accuracy for CIFAR-10 models and validation accuracy for LVIS models.

4. BONUS

4.1 Ablation Study: Normalization x Depth

We implement 3 variants of LVISSkipNet using NormSkipBlock with all four norm types. We use only 2 stages of skip blocks to reduce training time. rest of the parameters are kept the same as LVISSkipNet. We make a single class which can used for testing the different variants using parameters. Rest of the implementation is similar to LVISSkipNet with some additional helper methods to run the 12 experiments and make overlayed plots. Again most the plotting functions we make here are reused from previous tasks.

CODE for LVISSkipNet-10, LVISSkipNet-20, LVISSkipNet-30:

```
class LVISSkipNetAblation(nn.Module):
    """
    Two-stage LVIS skip network for normalization x depth
ablations.

    total_skipblocks counts only the non-downsampling skip blocks.
    """

    def __init__(
        self,
        num_classes=5,
        total_skipblocks=20,
        norm_type="instance",
        num_groups=8,
        skip_connections=True,
        stage_channels=(64, 128),
    ):
        super(LVISSkipNetAblation, self).__init__()

        c1, c2 = stage_channels
        self.total_skipblocks = int(total_skipblocks)
        self.norm_type = norm_type

        stage1_blocks = self.total_skipblocks // 2
        stage2_blocks = self.total_skipblocks - stage1_blocks

        self.conv_initial = nn.Conv2d(3, c1, 3, padding=1)
        self.bn_initial = nn.BatchNorm2d(c1)

        self.skip_stage1_arr = nn.ModuleList()
        for _ in range(stage1_blocks):
            self.skip_stage1_arr.append(
                NormSkipBlock(
                    c1,
                    c1,
                    norm_type=self.norm_type,
                    num_groups=num_groups,
                    downsample=False,
                    skip_connections=skip_connections,
```

```
                    )
                )

                self.skip_stage1_to_stage2 = NormSkipBlock(
                    c1,
                    c2,
                    norm_type=self.norm_type,
                    num_groups=num_groups,
                    downsample=True,
                    skip_connections=skip_connections,
                )

                self.skip_stage2_arr = nn.ModuleList()
                for _ in range(stage2_blocks):
                    self.skip_stage2_arr.append(
                        NormSkipBlock(
                            c2,
                            c2,
                            norm_type=self.norm_type,
                            num_groups=num_groups,
                            downsample=False,
                            skip_connections=skip_connections,
                        )
                    )

                self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
                self.fc = nn.Linear(c2, num_classes)

            def forward(self, x):
                x = F.relu(self.bn_initial(self.conv_initial(x)))

                for skip_stage1 in self.skip_stage1_arr:
                    x = skip_stage1(x)

                x = self.skip_stage1_to_stage2(x)

                for skip_stage2 in self.skip_stage2_arr:
                    x = skip_stage2(x)

                x = self.avgpool(x)
                x = x.view(x.size(0), -1)
                x = self.fc(x)
                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) or isinstance(m, nn.Linear))
```

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

# Code for running the actual experiments in train.py

def run_lvis_skipnet_norm_depth_ablation(
    depths=(10, 20, 30),
    norm_types=("batch", "instance", "layer", "group"),
    batch_size=32,
    epochs=10,
    lr=1e-3,
    betas=(0.9, 0.99),
    num_workers=2,
    num_groups=8,
    skip_connections=True,
):
    """
    Run normalization x depth ablation for two-stage LVISSkipNet
variants.

    total_skipblocks is split across two stages.
    """
    from dataset_creation import OUTPUT_DIR, LVISDataLoader

    device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
    print(f"Using device: {device}")

    dataset_name = "multi_instance_same"
    train_dir = os.path.join(OUTPUT_DIR, dataset_name, "train")
    val_dir = os.path.join(OUTPUT_DIR, dataset_name, "val")

    ablation_dir = os.path.join(OUTPUT_DIR, dataset_name,
"ablation")
    plots_dir = os.path.join(ablation_dir, "plots")
    models_dir = os.path.join(ablation_dir, "models")
    os.makedirs(plots_dir, exist_ok=True)
    os.makedirs(models_dir, exist_ok=True)

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

```
    train_loader = LVISDataLoader(
        train_dir,
        batch_size=batch_size,
        shuffle=True,
        num_workers=num_workers,
        transform=transform_train,
    )
    val_loader = LVISDataLoader(
        val_dir,
        batch_size=batch_size,
        shuffle=False,
        num_workers=num_workers,
        transform=transform_val,
    )

    class_names = sorted(
        train_loader.dataset.label_to_index,
        key=train_loader.dataset.label_to_index.get,
    )
    num_classes = len(class_names)

    criterion = nn.CrossEntropyLoss()
    norm_display = {
        "batch": "Batch Norm",
        "instance": "Instance Norm",
        "layer": "Layer Norm",
        "group": "Group Norm",
    }

    results = {}
    accuracy_table = {}
    all_experiments = []

    for depth in depths:
        print(f"\n===== Ablation: LVISSkipNet-{depth} =====")
        depth_losses = {}
        results[depth] = {}
        accuracy_table[depth] = {}

        for norm_type in norm_types:
            label = f"LVISSkipNet-{depth}
({norm_display.get(norm_type, norm_type)})"
            safe_name = f"lvisskipnet_d{depth}_{norm_type}"
            print(f"\n--- Training {label} ---")

            model = LVISSkipNetAblation(
                num_classes=num_classes,
                total_skipblocks=depth,
                norm_type=norm_type,
                num_groups=num_groups,
                skip_connections=skip_connections,
```

```
            ).to(device)

            optimizer = torch.optim.Adam(model.parameters(), lr=lr,
betas=betas)
            metrics = train_and_validate(
                model,
                train_loader,
                val_loader,
                criterion,
                optimizer,
                device,
                epochs=epochs,
            )

            cm = compute_confusion_matrix(model, val_loader,
num_classes=num_classes, device=device)
            class_acc = per_class_accuracy(cm)
            val_acc = 100.0 * np.trace(cm) / np.sum(cm)

            params = sum(p.numel() for p in model.parameters() if
p.requires_grad)
            conv_plus_linear_layers = model.count_layers()
            conv_layers = model.count_conv_layers()

            model_save_path = os.path.join(models_dir, f"
{safe_name}_model.pth")
            torch.save(model.state_dict(), model_save_path)

            results[depth][norm_type] = {
                "label": label,
                "metrics": metrics,
                "val_acc": val_acc,
                "params": params,
                "layers_conv_plus_linear": conv_plus_linear_layers,
                "layers_conv_only": conv_layers,
                "confusion_matrix": cm,
                "per_class_accuracy": class_acc,
                "model_path": model_save_path,
            }
            accuracy_table[depth][norm_type] = val_acc
            depth_losses[norm_display.get(norm_type, norm_type)] =
metrics["train_loss"]

            all_experiments.append(
                {
                    "depth": depth,
                    "norm_type": norm_type,
                    "label": label,
                    "safe_name": safe_name,
                    "val_acc": val_acc,
                    "confusion_matrix": cm,
                    "per_class_accuracy": class_acc,
```

```
                    }
                )

            plot_overlaid_training_losses(
                depth_losses,
                title=f"LVISSkipNet-{depth}: Training Loss by
Normalization",
                save_path=os.path.join(plots_dir,
f"ablation_depth_{depth}_loss_overlay.png"),
            )

    ordered_norms = [nt for nt in ["batch", "instance", "layer",
"group"] if nt in norm_types]
    print("\n===== 3 x 4 Accuracy Table (Val %) =====")
    header = f"{'Depth':<10}" + "".join([f"{norm_display[n]:>16}"
for n in ordered_norms])
    print(header)
    print("-" * len(header))
    for depth in depths:
        row = f"{depth:<10}" + "".join([f"{accuracy_table[depth]
[n]:>16.2f}" for n in ordered_norms])
        print(row)

    csv_path = os.path.join(ablation_dir,
"lvis_skipnet_norm_depth_accuracy_table.csv")
    with open(csv_path, "w", newline="") as f:
        writer = csv.writer(f)
        writer.writerow(["depth"] + ordered_norms)
        for depth in depths:
            writer.writerow([depth] + [f"{accuracy_table[depth]
[n]:.2f}" for n in ordered_norms])
    print(f"Saved accuracy table to {csv_path}")

    selected_cases = []
    if all_experiments:
        ranked = sorted(all_experiments, key=lambda x:
x["val_acc"])
        selected_cases.append(ranked[0])
        selected_cases.append(ranked[-1])

        max_depth = max(depths)
        max_depth_experiments = [e for e in all_experiments if
e["depth"] == max_depth]
        if max_depth_experiments:
            selected_cases.append(max(max_depth_experiments,
key=lambda x: x["val_acc"]))

        dedup = {}
        for case in selected_cases:
            dedup[case["safe_name"]] = case
        selected_cases = list(dedup.values())
```

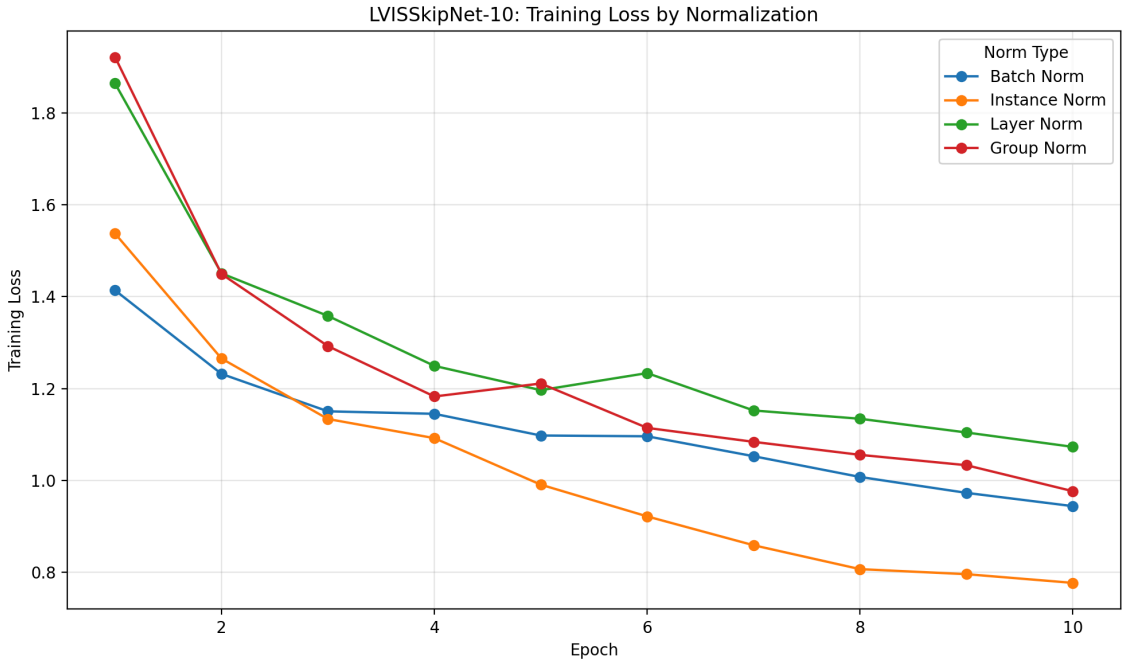
```
    for case in selected_cases:
        plot_confusion_matrix(
            case["confusion_matrix"],
            class_names,
            title=f"{case['label']} Confusion Matrix (Validation)",
            save_path=os.path.join(plots_dir, f"
{case['safe_name']}_confusion_matrix.png"),
        )
        display_per_class_accuracy(
            case["per_class_accuracy"],
            class_names,
            title=f"{case['label']} Per-Class Accuracy
(Validation)",
            save_path=os.path.join(plots_dir, f"
{case['safe_name']}_per_class_accuracy.png"),
        )

    return {
        "accuracy_table": accuracy_table,
        "results": results,
        "csv_path": csv_path,
        "selected_cases": [case["label"] for case in
selected_cases],
    }
```

Results:

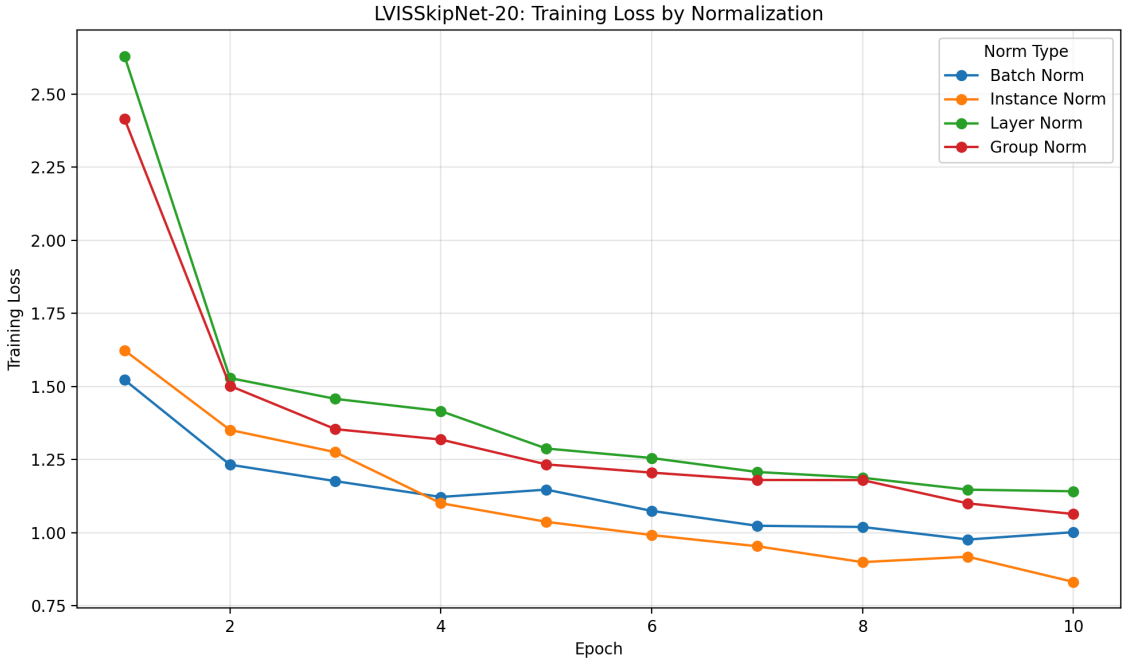
Accuracy Comparison:

```
===== 3 x 4 Accuracy Table (Val %) =====
Depth           Batch Norm   Instance Norm      Layer Norm
Group Norm
------------------------------------------------------------
-------------
10                   57.41          57.20           52.19
56.78
20                   55.32          59.71           48.02
49.90
30                   56.37          60.13           52.40
53.03
```

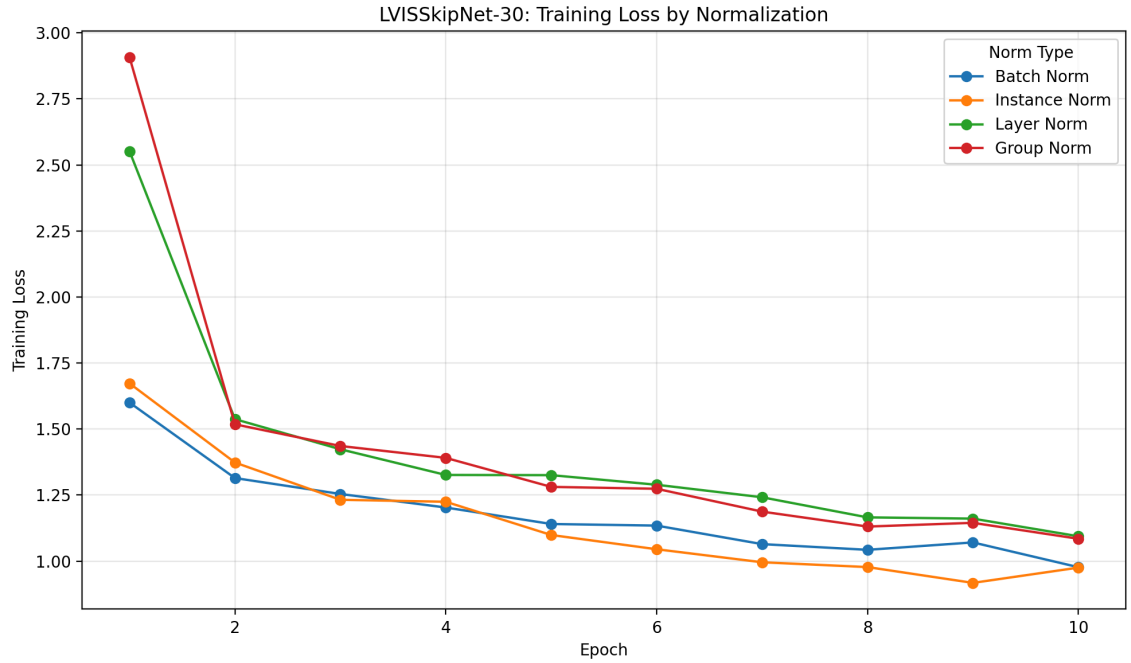
Training Loss for LVISSkipNet-10:



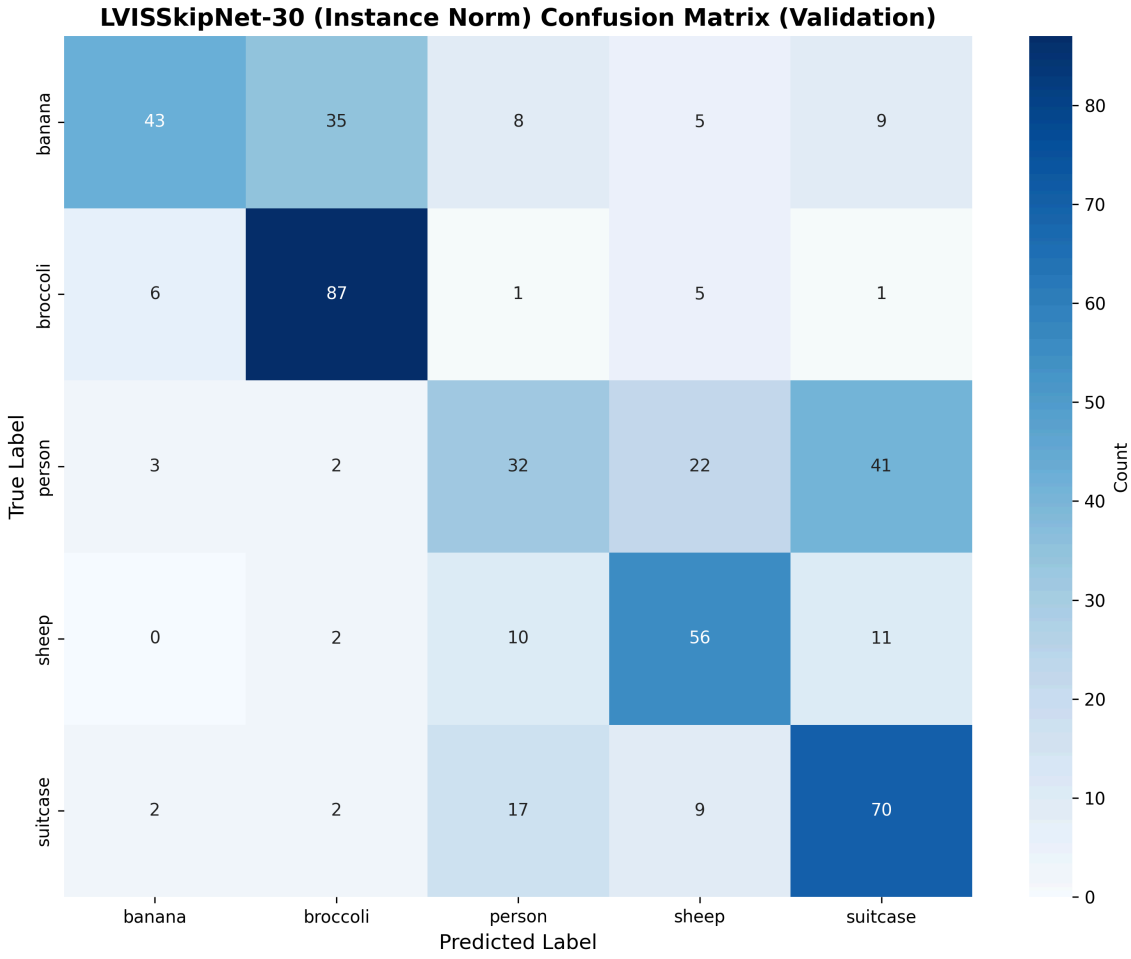
Training Loss for LVISSkipNet-20:



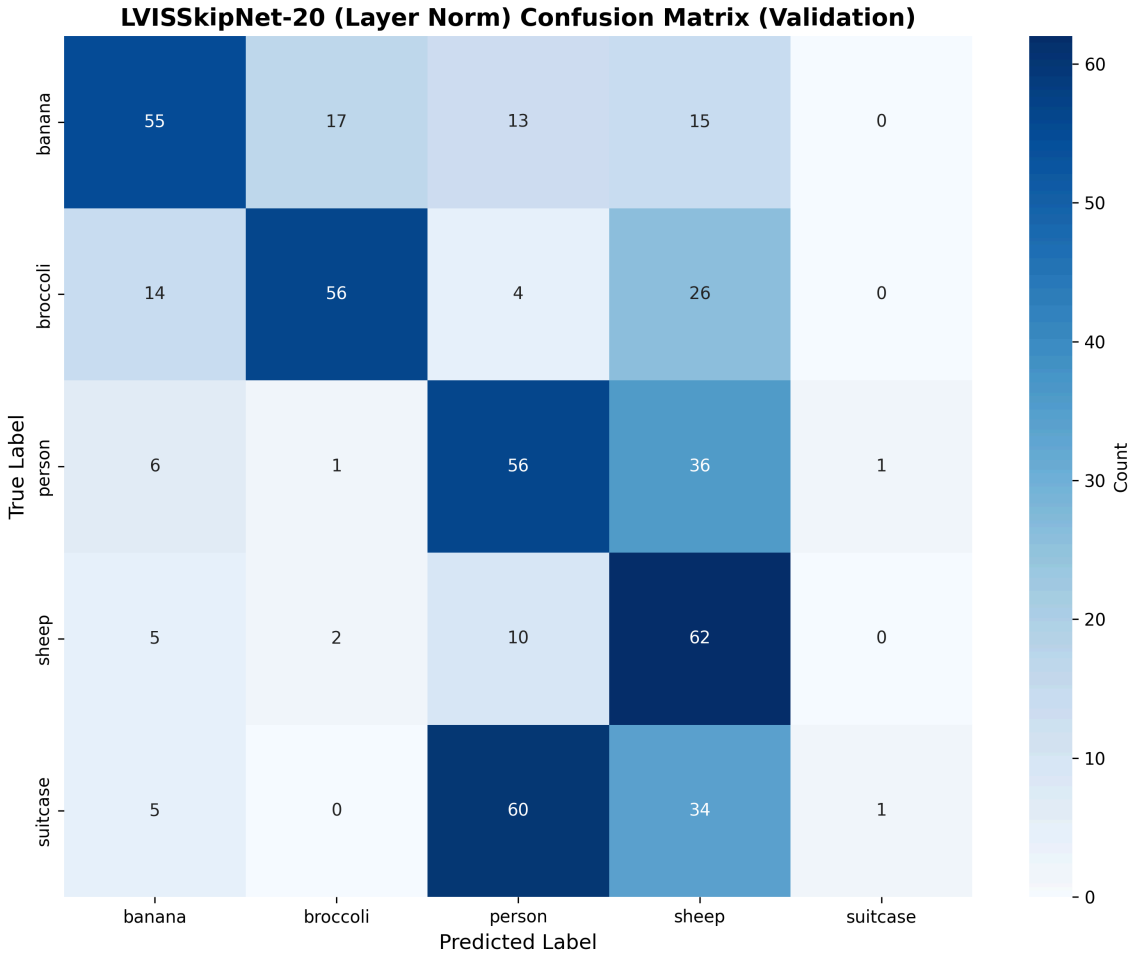
Training Loss for LVISSkipNet-30:



Confusion Matrix for LVISSkipNet-30 with Instance Norm (best case):



Confusion Matrix for LVISSkipNet-20 with Layer Norm (worst case):



Observations:

1. As the networks get deeper, one clear trend we see is that instance and batch norm consistently outperform layer and group norm. This can noticed in LVISSkipNet-20 and LVISSkipNet-30.
2. Diminishing returns actually shows up after 20ish skipblocks, best score improves only slightly from 59.71% to 60.13%.
3. In our testing setup, LN/GN do not become dominant at deeper depths and both stay below BN/IN.
4. We add two confusion matrices which represent the best and worst cases in terms of validation accuracy. "suitcase" category is one which shows a large drop in accuracy for the worst case. This could be due to the fact that in deeper layers, the model is able to learn more abstract and discriminative features which can help it to better distinguish between classes.