BME646 and ECE60146: Homework 5

Spring 2026
Due Date: Tuesday, Feb 24, 2026, 11:59 pm ET
TA: Somosmita Mitra (mitra26@purdue.edu)

Turn in typed solutions via Gradescope. Post questions to Piazza. Additional instructions can be found at the end. **Late submissions will be accepted with penalty: -10 points per late day, up to 5 days.**

1 Introduction

In Homework 4, you built and trained `LVISNet` (a 3-block CNN) on three LVIS-based datasets of increasing complexity, and you ran DLStudio's `Net` and `Net2` on CIFAR-10 to understand how `torch.nn` is used to define, train, and evaluate CNNs. In this homework, you will build on that foundation to investigate three fundamental questions:

1. **Does adding more convolutional layers always improve performance?**
2. **How do skip connections address the challenges of training deeper networks?**
3. **How does the choice of normalization strategy (Batch, Instance, Layer, Group) affect training and performance?**

As discussed in the lecture on skip connections and normalization [2], simply stacking more layers does not guarantee better results. Skip connections provide shortcut paths for gradient flow, and normalization strategies stabilize the statistics of in-transit data flowing through the network. You will experiment with both.

You will also visualize what your networks actually learn by extracting and comparing feature maps at different depths, connecting the lecture material on convolution [3] to your experimental results.

Concretely, you will:

- Design `Net3`, an 8-conv-layer CNN, and observe the degradation problem on CIFAR-10
- Study DLStudio's `SkipBlock` and run `BMEnet` on CIFAR-10 to see how skip connections address degradation

- Implement and compare four normalization strategies (Batch, Instance, Layer, and Group normalization) in a skip-connection network on CIFAR-10
- Design `LVISSkipNet`, a deep ($\geq$40 learnable layers) skip-connection network
- Train `LVISSkipNet` on your LVIS multi-instance same-object dataset from HW4, comparing against your HW4 `LVISNet` results
- Visualize learned convolutional filters and feature maps, comparing shallow vs. deep networks

1.1 Why Depth Matters

One of the foundational insights in deep learning is that deeper networks can learn more complex representations. However, simply stacking more layers does not guarantee better performance. Two key challenges arise [2]:

- **Vanishing gradients:** As gradients are backpropagated through many layers, they can shrink exponentially, making early layers nearly impossible to train. As shown in the lecture, the variance of the backpropagated loss gradient with respect to the pre-activation values depends on $[n \cdot \text{Var}[w]]^{d-i}$, which becomes vanishingly small for early layers when $n \cdot \text{Var}[w] < 1$.
- **Degradation problem:** Deeper networks can actually perform *worse* than shallower ones, even on training data, not due to overfitting, but due to optimization difficulty.

Skip connections (also called residual connections) [5] address both issues by providing shortcut paths for gradient flow. Instead of learning a mapping $\mathcal{H}(x)$ directly, a residual block learns $\mathcal{F}(x) = \mathcal{H}(x) - x$, making it easier for the network to learn identity-like mappings when additional depth is not helpful.

1.2 Why Normalization Matters

As data flows through a deep network, the statistical properties of the activations can shift significantly from layer to layer, a phenomenon Ioffe and Szegedy called **Internal Covariate Shift** [2]. This shift can push activation values into saturating regions of nonlinearities, exacerbating vanishing

gradients. Normalization strategies counter this by normalizing in-transit data before each activation, but they differ in *what* they normalize over:

- **Batch Normalization (BN):** Normalizes per channel across all instances in a batch, Eqs. (1)–(3) in the lecture [2]
- **Instance Normalization (IN):** Normalizes per channel and per instance, Eqs. (6)–(8)
- **Layer Normalization (LN):** Normalizes across all channels for each instance, Eqs. (9)–(11)
- **Group Normalization (GN):** Normalizes across groups of channels per instance, a middle ground between IN and LN

Each strategy has different strengths depending on the task, batch size, and network architecture.

1.3 Quick Reference: Lecture Slide Numbers

Use this table to look up the relevant lecture material for each topic in this homework.

HW5 Topic	**Lecture**	**Slides**
Skip Connections and Normalization [2]		
Getting started with skip connections	SkipConsAndBN	9–12
SkipBlock definition and code	SkipConsAndBN	13–18
SkipBlock `forward()` (3 cases)	SkipConsAndBN	16–17
BMEnet architecture	SkipConsAndBN	19–23
Results with/without skip connections	SkipConsAndBN	24–29
Batch Normalization (BN) formulas	SkipConsAndBN	30–36
Instance Normalization (IN) formulas	SkipConsAndBN	37–40
Layer Normalization (LN) formulas	SkipConsAndBN	41–46
Vanishing gradients math	SkipConsAndBN	47–55
Why skip connections help (ensemble)	SkipConsAndBN	56–60
Loss landscape visualization	SkipConsAndBN	61–64
Demystifying the Convolutions in PyTorch [3]		
2D convolution definition	DemystifyConvo	5–8
`torch.nn.functional.conv2d()`	DemystifyConvo	12–27
`torch.nn.Conv2d` class	DemystifyConvo	28–35
Convolution vs. cross-correlation	DemystifyConvo	36–39
Multi-channel convolutions	DemystifyConvo	40–51
Tensor reshaping (`view`/`reshape`)	DemystifyConvo	52+

Which slides to read for each task:

Task	Key slides
Task 1: Degradation problem	SkipConsAndBN 24–29, 47–55
Task 2: SkipBlock and BMEnet	SkipConsAndBN 13–23
Task 3: Normalization experiments	SkipConsAndBN 30–46
Task 4: LVISSkipNet	SkipConsAndBN 13–23, 56–60
Task 5: Feature map visualization	DemystifyConvo 28–35, 40–51
Task 6: Comparative analysis	SkipConsAndBN 24–29, 61–64

2 Setting Up Your Environment

2.1 Prerequisites from HW4

This homework builds directly on your HW4 setup. Ensure you have:

- DLStudio version 2.5.5 installed (with the Examples directory)
- Your LVIS multi-instance same-object dataset from HW4
- Your trained `LVISNet` results from HW4 (confusion matrix, accuracies)
- Your `train.py` and `evaluate.py` from HW4 (you will reuse them)
- LVIS annotations (`lvis_v1_train.json`) and COCO 2017 training images

2.2 Verify DLStudio Installation

Verify that `playing_with_cifar10.py` and `playing_with_skip_connections.py` exist in the Examples directory, and that you can import the `DLStudio` module in Python.

2.3 Locating Key DLStudio Components

Open the main module file `DLStudio/DLStudio.py` in your text editor and familiarize yourself with:

1. The `SkipConnections` class and its `SkipBlock` building block
2. The `BMEnet` architecture built from `SkipBlock` components
3. The use of `nn.BatchNorm2d` within `SkipBlock`

Read the docstrings associated with each class definition. Understanding `SkipBlock` is essential: it will serve as the building block for the networks you create in Tasks 3 and 4.

2.4 LVIS Dataset Preparation Requirements

Many students in HW4 had issues with imbalanced classes, augmentation strategies, or had too few images. Follow the requirements below exactly. Failure to meet these requirements will result in lost points.

You will use your **multi-instance same-object** LVIS dataset from HW4 (images where multiple instances of the same category appear). If your HW4 dataset does not meet the requirements below, you must rebuild it before starting.

2.4.1 Minimum Image Counts

Your dataset must contain at least the following number of **original** (non-augmented) images:

Split	Min images per class	Min total (5 classes)
Training	300	1,500
Validation	50	250

If any class has fewer than 300 training images, you have two options: (1) choose a different LVIS category that has enough images, or (2) apply offline augmentation *only to the training set* to reach the minimum (see below).

2.4.2 Class Balance

All five classes must have approximately the same number of training images. Specifically, no class may have more than **1.2**$\times$ the number of images as the smallest class. For example, if your smallest class has 300 images, no class may have more than 360.

If your raw LVIS data is imbalanced, **undersample** the larger classes (randomly remove excess images) or **augment** the smaller classes to match. Report the final per-class image counts in your submission.

2.4.3 Train/Validation Split

Use an **80/20 split**: 80% of images for training, 20% for validation. The split must be done **before** any augmentation. That is, an image and its augmented variants must all belong to the same split. Never let an augmented version of a training image appear in the validation set.

2.4.4 Data Augmentation and Transforms

Augment only the training set. Do NOT augment the validation set.

Use exactly the following transforms:

Training transforms:

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

Validation transforms:

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

Why no augmentation on validation? The validation set should represent real-world test conditions. Augmenting it would artificially inflate your metrics and make results non-comparable across students.

2.4.5 Dataset Verification Checklist

Before training any model, verify and **report** the following in your submission:

1. Print the number of images per class in the training set
2. Print the number of images per class in the validation set
3. Confirm no class exceeds 1.2× the size of the smallest class (training set)
4. Confirm total training images $\geq$ 1,500 and total validation images $\geq$ 250

5. List your 5 LVIS category names

Include the output of this verification in your report. Use code like:

```
from torchvision.datasets import ImageFolder

train_set = ImageFolder('data/lvis/train',
                        transform=transform_train)
val_set   = ImageFolder('data/lvis/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")
```

3 Programming Tasks (100 Points Total)

3.1 Task 1: The Degradation Problem (5 points)

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

In HW4, you ran DLStudio's `Net` and `Net2` on CIFAR-10. You will now use those results as baselines and design a deeper `Net3` to demonstrate the degradation problem.

3.1.1 Designing Net3 (3 points)

Requirement: Create `Net3` with **exactly 8 convolutional layers** by extending `ExperimentsWithCIFAR` through inheritance (similar to how `Net` and `Net2` are defined in DLStudio).

Design guidelines:

- Input: $(B, 3, 32, 32)$ CIFAR-10 images. Output: $(B, 10)$ class logits.
- With 32×32 input, you can pool at most 5 times before spatial dimensions collapse. A good strategy is to group your 8 conv layers into 4 pairs with pooling between groups: `conv-conv-pool` (32→16), `conv-conv-pool` (16→8), `conv-conv-pool` (8→4), `conv-conv-pool` (4→2). Then flatten for FC layers.

- Include batch normalization and ReLU after each convolutional layer.
- The autograder will count `nn.Conv2d` instances in your model and verify there are exactly 8.

Train Net3 on CIFAR-10 using the DLStudio training infrastructure (10 epochs, same hyperparameters as the `playing_with_cifar10.py` script).

Deliverables:

- Training loss curve for Net3
- Confusion matrix (10×10)
- Overall and per-class classification accuracy
- Number of learnable parameters

3.1.2 Comparative Analysis (2 points)

Using your HW4 results for `Net` and `Net2` as baselines, create the following tables:

Table 1, Overall Accuracy:

Network	**Conv Layers**	**Parameters**	**Accuracy (%)**
Net (HW4)	?	?	?
Net2 (HW4)	?	?	?
Net3	8	?	?

Table 2, Per-Class Accuracy: A 10×3 table where each row is a CIFAR-10 class and each column is a network.

Discussion (1 paragraph):

- Does accuracy improve monotonically from Net $\rightarrow$ Net2 $\rightarrow$ Net3?
- If Net3 performs worse, explain why.
- Which classes are most affected?

Grading:

- Correct Net3 implementation with 8 conv layers and all deliverables (3 points)
- Comparative tables and discussion referencing HW4 results (2 points)

3.2 Task 2: Skip Connections on CIFAR-10 (15 points)

Objective: Learn how skip connections address the degradation problem by studying DLStudio's `SkipBlock` and running `BMEnet` on CIFAR-10.

3.2.1 Studying the SkipBlock (5 points)

Open `DLStudio/DLStudio.py` and search for `class SkipConnections`. Study the `SkipBlock` inner class implementation carefully. Refer to Slides 13–18 of the lecture [2] for a walkthrough of this code.

Report:

- Describe the architecture of a single `SkipBlock` (what layers does it contain?)
- Explain how the skip (residual) connection works: what is added to what?
- How does `SkipBlock` handle the case when input and output channel dimensions differ? Describe all three cases in the `forward()` method.
- Relate your explanation to the residual learning formulation: $\mathcal{F}(x) + x$

3.2.2 Running BMEnet on CIFAR-10 (5 points)

Run the skip connections example script `playing_with_skip_connections.py` from the DLStudio Examples directory. The script instantiates `BMEnet` with configurable `skip_connections` (True/False) and `depth` parameters. Run it with `skip_connections=True` and `depth=32`. Refer to Slides 19–28 of the lecture [2] for an explanation of the BMEnet architecture and expected results.

Deliverables:

- Training loss curve
- Confusion matrix (10×10)
- Per-class accuracy (10 rows $\times$ 1 column)
- Total number of learnable parameters

3.2.3 BMEnet vs. Deeper Networks Discussion (5 points)

Add BMEnet to your Tables 1 and 2 from Task 1. Write a discussion (1–2 paragraphs) addressing:

- Does BMEnet outperform Net3 despite Net3 potentially suffering from degradation?
- How do skip connections help with training deeper architectures?
- Which CIFAR-10 classes show the most improvement with BMEnet?

Grading:

- SkipBlock analysis and explanation (5 points)
- Successful BMEnet execution with all deliverables (5 points)
- Comparative discussion with updated tables (5 points)

3.3 Task 3: Normalization Experiments (15 points)

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.

As described in the lecture [2], all four strategies normalize in-transit data before the activation function, but they differ in *what dimensions* they compute the mean μ and standard deviation σ over:

Strategy	**Normalizes over**	**PyTorch class**
Batch Norm (BN)	(B, H, W) per channel	`nn.BatchNorm2d(C)`
Instance Norm (IN)	(H, W) per channel, per instance	`nn.InstanceNorm2d(C)`
Layer Norm (LN)	(C, H, W) per instance	`nn.GroupNorm(1, C)`
Group Norm (GN)	$(C/G, H, W)$ per instance, per group	`nn.GroupNorm(G, C)`

Note on Layer Normalization for convolutional layers: `nn.LayerNorm` requires the exact spatial dimensions (C, H, W) at construction time, which is inconvenient when H and W vary across layers. Instead, use `nn.GroupNorm(1, C)`, which normalizes over all channels per instance and is mathematically equivalent to Layer Normalization over (C, H, W).

Note on Instance Normalization: By default, `nn.InstanceNorm2d` does *not* have learnable affine parameters. Set `affine=True` so it learns γ and β, matching the behavior of the other normalizations.

3.3.1 Implementing NormSkipBlock (5 points)

Create a modified `SkipBlock` called `NormSkipBlock` that accepts a `norm_type` parameter (one of `'batch'`, `'instance'`, `'layer'`, `'group'`). The block should be otherwise identical to DLStudio's `SkipBlock`, but it replaces every `nn.BatchNorm2d` with the normalization layer corresponding to `norm_type`. Implement a factory method `_make_norm(norm_type, num_channels, num_groups)` that returns the appropriate normalization layer.

Constructor signature:

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

The autograder will test that `NormSkipBlock` works with all four normalization types and produces correct output shapes. For Group Normalization, use $G = 8$ groups. Ensure the number of groups divides the number of channels.

Factory Method Template:

```python
@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':
        # GroupNorm(1, C) is equivalent to
        # LayerNorm over (C, H, W)
        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}")
```

Use this factory in your `__init__` to create `self.norm1` and `self.norm2` in place of the `nn.BatchNorm2d` calls in the original `SkipBlock`.

Report:

- Include your complete `NormSkipBlock` code

- Explain the key difference between BN, IN, LN, and GN in your own words, referencing the lecture formulas [2]
- Explain the `GroupNorm(1, C)` trick for implementing Layer Normalization in convolutional layers

3.3.2 Building NormBMEnet (5 points)

Build a network called `NormBMEnet` that mirrors `BMEnet`'s architecture but uses your `NormSkipBlock` so you can swap the normalization strategy with a single constructor argument. The architecture should match BMEnet as described in Slides 19–23 of the lecture [2]:

- Initial `Conv2d(3, 64, 3, padding=1)` with ReLU
- `depth` number of `NormSkipBlock(64, 64)` instances
- One `NormSkipBlock(64, 128, downsample=True)`
- `depth` number of `NormSkipBlock(128, 128)` instances
- One `NormSkipBlock(128, 256, downsample=True)`
- `depth` number of `NormSkipBlock(256, 256)` instances
- Flatten, then `FC(256*8*8, 1000)` → ReLU → `FC(1000, 10)`

Constructor signature:

```
NormBMEnet(norm_type='batch', depth=8,
           skip_connections=True, num_groups=8)
```

Input: $(B, 3, 32, 32)$. Output: $(B, 10)$.
Report: Include your complete `NormBMEnet` code.

3.3.3 Training and Comparison (5 points)

Train `NormBMEnet` with each of the four normalization types (`'batch'`, `'instance'`, `'layer'`, `'group'`) on CIFAR-10. Use **identical hyperparameters** across all four runs: Adam optimizer with `lr=1e-3`, `CrossEntropyLoss`, 10 epochs, `batch_size=64`, `depth=8`, `skip_connections=True`.

Hint: If your training loss stays near 2.3 (random guessing for 10 classes) with Instance or Group Normalization, your network is not learning. InstanceNorm removes instance-level contrast information, which can slow classification training compared to BatchNorm. Try reducing the learning rate by

5–10x for those runs. It is acceptable to use a different learning rate for normalization types that fail to converge, as long as you document it and discuss why in your report.

Deliverables:

- Training loss curves for all four normalization types **overlaid on a single plot**
- Summary table:

Norm Type	**Parameters**	**Test Accuracy (%)**
Batch Norm	?	?
Instance Norm	?	?
Layer Norm	?	?
Group Norm ($G = 8$)	?	?

- Discussion (1–2 paragraphs):
  - Which normalization converges fastest? Which achieves the best final accuracy?
  - Does Instance Norm behave differently from Batch Norm? Why might this be? (Hint: think about what IN normalizes over vs. BN, as described in the lecture [2].)
  - Is Group Norm a good compromise between IN and LN?

Grading:

- Correct `NormSkipBlock` with factory method (5 points)
- Complete `NormBMEnet` implementation (5 points)
- All four experiments with overlaid loss curves, table, and discussion (5 points)

3.4 Task 4: Skip Connection Network for LVIS (30 points)

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

Important: Before starting this task, verify that your dataset meets the requirements in Section 2.4. Include the verification output in your report.

3.4.1 Network Architecture: LVISSkipNet (15 points)

Build a network called `LVISSkipNet` with the following requirements:

- **At least 40 learnable layers.** Since each `SkipBlock` uses two instances of `nn.Conv2d`, your network needs at least 20 `SkipBlock` instances. A learnable layer is any nn.Conv2d or nn.Linear module with learnable parameters.
- Your network must be composed of `SkipBlock` layers, **not** raw `nn.Conv2d` layers. Your network should consist of layers of `SkipBlock`, in the same manner as `BMEnet` is built from `SkipBlock` components.
- Input: $(B, 3, 64, 64)$ RGB images (matching your HW4 image size)
- Output: $(B, 5)$ class logits (your 5 LVIS categories from HW4)
- You may use an initial `Conv2d(3, 64, 3, padding=1)` to set the channel dimension from RGB. This is the one place you may use a raw `nn.Conv2d`.
- Use `AdaptiveAvgPool2d((1, 1))` before the final fully connected layer.

You may:

- Inherit the `SkipBlock` class from DLStudio, **OR**
- Copy `SkipBlock` into your code with appropriate citations to the source, **OR**
- Write your own skip block inspired by the DLStudio implementation with citations
- Use your `NormSkipBlock` from Task 3 if desired (document which normalization you chose)

Implement `count_parameters()` and `count_layers()` methods. Verify that `count_layers()` returns $\geq$40 before training.

Architecture Template:

```
class LVISSkipNet(nn.Module):
    def __init__(self, num_classes=5):
        super().__init__()
        self.conv_initial = nn.Conv2d(
            3, 64, 3, padding=1)
        self.bn_initial = nn.BatchNorm2d(64)
```

```
7
8         # === YOUR SKIPBLOCK STAGES HERE ===
9         # Example pattern (adapt as needed):
10        #   Stage 1: SkipBlock(64,64) x N
11        #   Transition: SkipBlock(64,128,ds=True)
12        #   Stage 2: SkipBlock(128,128) x N
13        #   Transition: SkipBlock(128,256,ds=True)
14        #   ...
15        # Total SkipBlocks >= 20 for >= 40 layers
16
17        self.avgpool = nn.AdaptiveAvgPool2d((1,1))
18        self.fc = nn.Linear(???, num_classes)
19
20    def forward(self, x):
21        x = F.relu(self.bn_initial(
22            self.conv_initial(x)))
23        # === YOUR SKIPBLOCK FORWARD HERE ===
24        x = self.avgpool(x)
25        x = x.view(x.size(0), -1)
26        x = self.fc(x)
27        return x
28
29    def count_parameters(self):
30        return sum(p.numel()
31            for p in self.parameters()
32            if p.requires_grad)
33
34    def count_layers(self):
35        return sum(1 for m in self.modules()
36               if isinstance(m, (nn.Conv2d, nn.Linear)))
```

Report:

- Include your complete `LVISSkipNet` code
- State the total number of learnable parameters and learnable layers
- Describe your channel progression and downsampling strategy

3.4.2 Training on LVIS Multi-Instance Same-Object Dataset (15 points)

Train `LVISSkipNet` on your multi-instance same-object dataset using the transforms from Section 2.4.

Training configuration:

- Optimizer: `Adam(lr=1e-3, betas=(0.9, 0.99))`

- Loss: `CrossEntropyLoss`
- Epochs: 10
- Batch size: 32
- Image size: 64×64
- Transforms: exactly as specified in Section 2.4

Deliverables:

- **Dataset verification output** (per-class counts, balance check) as described in Section 2.4
- Training and validation loss curves
- Training and validation accuracy curves
- Confusion matrix (5×5) on the validation set
- Per-class accuracy
- Training time per epoch

Grading:

- Correct `LVISSkipNet` architecture with $\geq$40 layers (15 points)
- Successful training with all deliverables including dataset verification (15 points)

3.5 Task 5: Feature Map Visualization (15 points)

Objective: Visualize what your networks learn by extracting convolutional filters and intermediate feature maps, comparing shallow (`LVISNet`) vs. deep (`LVISSkipNet`) networks on the multi-instance same-object dataset.

This connects the lecture material on how convolutions work [3] to what your trained networks actually compute.

3.5.1 Visualizing Learned Filters (5 points)

Extract and visualize the first-layer convolutional filters from both `LVISNet` (HW4) and `LVISSkipNet`. Access the filter weights via the `.weight.data` attribute of the first `Conv2d` layer. For 3-channel (RGB) filters, transpose to $(H, W, 3)$ and normalize to $[0, 1]$ for display. Show up to 32 filters in a grid layout.

Report:

- Show first-layer filter visualizations for both `LVISNet` and `LVISSkipNet`
- Do the filters look different? Do any resemble known edge detectors or color filters?
- Include the code you wrote for filter visualization

3.5.2 Visualizing Intermediate Feature Maps (5 points)

Extract and visualize feature maps at different depths for the same input image. Use PyTorch forward hooks (`register_forward_hook`) to capture the output of specific layers during a forward pass. Show the first 8 channels of each captured layer as a grid of grayscale images. Normalize each channel independently for display.

Hook Template:

```
def get_feature_maps(model, input_image,
                     layer_names):
    """Extract feature maps using hooks.
    Args:
        model: trained model
        input_image: tensor (1, 3, H, W)
        layer_names: list of module name strings
    Returns: dict {name: tensor (1, C, H, W)}
    """
    feature_maps = {}
    hooks = []

    def hook_fn(name):
        def hook(module, input, output):
            feature_maps[name] = \
                output.detach().cpu()
        return hook

    for name, module in model.named_modules():
        if name in layer_names:
            hooks.append(
```

```
                module.register_forward_hook(
                    hook_fn(name)))

    model.eval()
    with torch.no_grad():
        model(input_image)

    for h in hooks:
        h.remove()
    return feature_maps
```

To find hookable layer names, use:

```
for name, module in model.named_modules():
    print(name, type(module).__name__)
```

Report:

- For **one sample image**, show feature maps at early, middle, and late layers of `LVISSkipNet`
- Show the original input image alongside the feature maps
- Describe qualitatively how feature maps change with depth (early = edges/textures, late = more abstract)
- Include the code you wrote for feature map extraction and visualization

3.5.3 Comparative Analysis: Shallow vs. Deep (5 points)

Report:

- Compare feature maps from `LVISNet` (shallow, HW4) and `LVISSkipNet` (deep) for the same input image
- Do the deeper feature maps appear more discriminative?
- Does the deeper network's first layer learn similar or different filters compared to the shallow network?
- Relate your observations to the lecture material [3] on how multi-channel convolutions combine information across channels

Grading:

- First-layer filter visualization and analysis (5 points)

- Intermediate feature map visualization (5 points)
- Shallow vs. deep comparative analysis (5 points)

3.6 Task 6: Final Comparative Analysis (20 points)

Objective: Synthesize all results into a comparison across architectures and normalization strategies.

3.6.1 Confusion Matrices (8 points)

Generate and display confusion matrices for:

1. CIFAR-10: Net3, BMEnet (two 10×10 matrices, using your HW4 Net/Net2 results as baselines)
2. LVIS: `LVISSkipNet` on the multi-instance same-object dataset (one 5×5 matrix)

Reuse your confusion matrix implementation from HW4 (implemented from scratch, no sklearn). Visualization libraries (matplotlib, seaborn) are permitted for plotting the heatmaps.

3.6.2 LVISNet vs. LVISSkipNet (7 points)

This is the central comparison. Create the following summary table:

Network	Parameters	Val Accuracy (%)
LVISNet (HW4)	?	?
LVISSkipNet	?	?

Discussion (2–3 paragraphs):

- **Does depth help?** Compare the accuracy improvement from LVISNet to LVISSkipNet. Does the deeper network with skip connections outperform the shallow network?
- **Normalization insights:** Based on your Task 3 results, which normalization strategy would you recommend for the LVIS task? If you used your `NormSkipBlock` in `LVISSkipNet`, discuss the effect of the normalization choice.
- **Overall findings:** Summarize the key takeaways from this homework: what did you learn about depth, skip connections, normalization, and model capacity?

3.6.3 Master Summary Table (5 points)

Create a table covering all experiments:

Network	Dataset	Layers	Parameters	Accuracy (%)
Net (HW4)	CIFAR-10	?	?	?
Net2 (HW4)	CIFAR-10	?	?	?
Net3	CIFAR-10	8 conv	?	?
BMEnet	CIFAR-10	?	?	?
NormBMEnet (BN)	CIFAR-10	?	?	?
NormBMEnet (IN)	CIFAR-10	?	?	?
NormBMEnet (LN)	CIFAR-10	?	?	?
NormBMEnet (GN)	CIFAR-10	?	?	?
LVISNet (HW4)	LVIS multi-same	3 blocks	?	?
LVISSkipNet	LVIS multi-same	$\geq$40	?	?

Grading:

- All confusion matrices correctly generated (8 points)
- LVISNet vs. LVISSkipNet comparison with normalization discussion (7 points)
- Master summary table and overall discussion (5 points)

4 Bonus Task (20 Points)

4.1 Ablation Study: Normalization $\times$ Depth

Design and run a systematic ablation that crosses **normalization type** with **network depth** on the LVIS multi-instance same-object dataset:

1. Implement three variants of `LVISSkipNet` using `NormSkipBlock`:
   - **LVISSkipNet-10:** 10 SkipBlocks (20 conv layers)
   - **LVISSkipNet-20:** 20 SkipBlocks (40 conv layers)
   - **LVISSkipNet-30:** 30 SkipBlocks (60 conv layers)
2. Train each depth with **all four normalization types** (12 experiments total)
3. Report:

- A 3×4 accuracy table (rows = depth, columns = norm type)
- Training curves overlaid by normalization type for each depth
- Confusion matrices for the most interesting cases

4. **Analysis:**
   - Does the best normalization strategy change as the network gets deeper?
   - At what point do you see diminishing returns from more SkipBlocks?
   - Does BN remain the best choice at all depths, or does LN/GN become more important for very deep networks?

5 Submission Instructions

5.1 What to Submit in the Report

Read this section carefully. Incomplete submissions will lose points. Every item listed below must be present in your submission.

1. **Task 1: The Degradation Problem (5 points)**
   (a) **Code:** Your complete `Net3` class definition (the `__init__` and `forward` methods)
   (b) **Code:** The script you used to instantiate, train, and test Net3 via DLStudio
   (c) **Figure:** Training loss curve for Net3 (x-axis: epoch, y-axis: loss)
   (d) **Figure:** 10×10 confusion matrix for Net3 on CIFAR-10 test set
   (e) **Table 1:** Overall accuracy table with Net (HW4), Net2 (HW4), and Net3, showing conv layer count, parameter count, and accuracy for each
   (f) **Table 2:** Per-class accuracy table (10×3, rows = CIFAR-10 classes, columns = Net, Net2, Net3)
   (g) **Text:** One paragraph discussing the degradation problem, referencing your results

2. **Task 2: Skip Connections on CIFAR-10 (15 points)**
   (a) **Text:** Architecture description or diagram of a single `SkipBlock`

(b) **Text:** Explanation of how skip connections work (the three cases for channel mismatch)

(c) **Text:** Explanation relating `SkipBlock` to the $\mathcal{F}(x) + x$ formulation

(d) **Code:** The script you used to run BMEnet (showing the DLStudio constructor call and the `skip_connections=True, depth=32` arguments)

(e) **Figure:** Training loss curve for BMEnet

(f) **Figure:** 10×10 confusion matrix for BMEnet on CIFAR-10

(g) **Table:** Updated Tables 1 and 2 from Task 1 with BMEnet row added

(h) **Text:** 1–2 paragraph discussion comparing BMEnet vs. Net3 and the role of skip connections

3. **Task 3: Normalization Experiments (15 points)**

(a) **Code:** Complete `NormSkipBlock` class definition, including the `_make_norm` factory method

(b) **Code:** Complete `NormBMEnet` class definition

(c) **Code:** Training script showing the loop over all four normalization types with identical hyperparameters

(d) **Text:** Explanation of the differences between BN, IN, LN, and GN, referencing the lecture formulas

(e) **Text:** Explanation of the `GroupNorm(1, C)` trick for Layer Normalization

(f) **Figure:** Single plot with all four training loss curves overlaid (one curve per norm type, clearly labeled with a legend)

(g) **Table:** Normalization summary table showing norm type, parameter count, and test accuracy for each of the four experiments

(h) **Text:** 1–2 paragraph discussion of which normalization converges fastest, which achieves the best accuracy, and why

4. **Task 4: LVISSkipNet on LVIS (30 points)**

(a) **Code:** Complete `LVISSkipNet` class definition (including `count_parameters` and `count_layers` methods)

(b) **Text:** Description of your architecture: channel progression, number of SkipBlocks per stage, downsampling strategy

(c) **Text:** Total learnable parameters and total learnable layers (must be $\geq$40)

(d) **Output:** Dataset verification (per-class image counts, balance check, category names) as specified in Section 2.4

(e) **Code:** Training script showing the training loop, hyperparameters, and data loading

(f) **Figure:** Training and validation loss curves

(g) **Figure:** Training and validation accuracy curves

(h) **Figure:** 5×5 confusion matrix on validation set

(i) **Table:** Per-class accuracy

(j) **Text:** Training time per epoch

5. **Task 5: Feature Map Visualization (15 points)**

(a) **Code:** Your filter visualization function

(b) **Figure:** First-layer filter grid for `LVISNet` (HW4 model)

(c) **Figure:** First-layer filter grid for `LVISSkipNet`

(d) **Text:** Comparison of the two filter grids: do they look different? Do any resemble edge detectors or color filters?

(e) **Code:** Your feature map extraction function (using forward hooks)

(f) **Figure:** Feature maps at early, middle, and late layers of `LVISSkipNet` for one sample image, with the original input image shown alongside

(g) **Text:** Qualitative description of how feature maps change with depth

(h) **Text:** Shallow vs. deep comparison: compare feature maps from `LVISNet` and `LVISSkipNet` for the same image, discussing discriminative power and relating to the lecture material [3]

6. **Task 6: Final Comparative Analysis (20 points)**

(a) **Figures:** 10×10 confusion matrices for Net3 and BMEnet on CIFAR-10 (may reuse from Tasks 1–2)

(b) **Figure:** 5×5 confusion matrix for `LVISSkipNet` (may reuse from Task 4)

(c) **Code:** The confusion matrix computation and plotting code you used

(d) **Table:** LVISNet vs. LVISSkipNet comparison table

(e) **Table:** Master summary table covering all networks (10 rows)

(f) **Text:** 2–3 paragraph discussion addressing: does depth help, normalization insights from Task 3, and overall findings

7. **Bonus: Ablation Study (20 points, optional)**

   (a) **Code:** Ablation script

   (b) **Table:** 3×4 accuracy table (depth $\times$ norm type)

   (c) **Figures:** Training curves overlaid by norm type for each depth

   (d) **Figures:** Confusion matrices for the most interesting cases

   (e) **Text:** Analysis of how norm strategy interacts with depth

5.2 Code Files to Submit

Submit the following (see Section 5.3 for exact function signatures):

1. `models.py`: `Net3`, `NormSkipBlock`, `NormBMEnet`, `LVISSkipNet`
2. `train.py`: `train_one_epoch`, `validate` (reuse/extend from HW4)
3. `evaluate.py`: `compute_confusion_matrix`, `per_class_accuracy` (reuse from HW4)
4. `feature_visualization.py`: Filter and feature map visualization functions
5. `requirements.txt`: Dependencies
6. `README.md`: Instructions to run your code

5.3 Autograder Interface Requirements

The autograder will import your files and call specific functions. If your file names or function signatures do not match exactly, the autograder will fail and you will lose points.

5.3.1 Required File Names

Submit exactly these file names (case-sensitive):

- `models.py`
- `train.py`
- `evaluate.py`

5.3.2 `models.py`

Must contain the following classes with these exact signatures:

- `Net3(num_classes=10)`: 8-conv-layer CNN for CIFAR-10. Must have `forward(x)` accepting $(B, 3, 32, 32)$ and returning $(B, 10)$, and `count_parameters()` returning an `int`.
- `NormSkipBlock(in_ch, out_ch, norm_type='batch', num_groups=8, downsample=False, skip_connections=True)`: SkipBlock with configurable normalization. Must accept all four norm types and produce correct output shapes.
- `NormBMEnet(norm_type='batch', depth=8, skip_connections=True, num_groups=8)`: BMEnet with configurable normalization. `forward(x)` accepts $(B, 3, 32, 32)$ and returns $(B, 10)$.
- `LVISSkipNet(num_classes=5)`: Deep skip-connection network. `forward(x)` accepts $(B, 3, 64, 64)$ and returns $(B, 5)$. Must have `count_parameters()` and `count_layers()`, with `count_layers()` ≥ 40.

The autograder will run:

```
from models import Net3, NormSkipBlock, \
                   NormBMEnet, LVISSkipNet
import torch

# Test Net3
net3 = Net3(num_classes=10)
out3 = net3(torch.randn(4, 3, 32, 32))
assert out3.shape == (4, 10)
conv_count = sum(1 for m in net3.modules()
    if isinstance(m, torch.nn.Conv2d))
assert conv_count == 8

# Test NormSkipBlock with all 4 norm types
```

```
for nt in ['batch','instance','layer','group']:
    blk = NormSkipBlock(64, 64, norm_type=nt)
    out = blk(torch.randn(2, 64, 16, 16))
    assert out.shape == (2, 64, 16, 16)

# Test NormBMEnet with all 4 norm types
for nt in ['batch','instance','layer','group']:
    bme = NormBMEnet(norm_type=nt, depth=4)
    out = bme(torch.randn(2, 3, 32, 32))
    assert out.shape == (2, 10)

# Test LVISSkipNet
skipnet = LVISSkipNet(num_classes=5)
out_skip = skipnet(torch.randn(4, 3, 64, 64))
assert out_skip.shape == (4, 5)
assert skipnet.count_layers() >= 40
```

5.3.3 `train.py`

Same signatures as HW4:

- `train_one_epoch(model, dataloader, criterion, optimizer, device)`: returns `(avg_loss: float, accuracy: float)`
- `validate(model, dataloader, criterion, device)`: returns `(avg_loss: float, accuracy: float)`

5.3.4 `evaluate.py`

Same signatures as HW4:

- `compute_confusion_matrix(model, dataloader, num_classes, device)`: returns `np.ndarray` of shape `(num_classes, num_classes)`
- `per_class_accuracy(cm)`: returns `np.ndarray` of shape `(num_classes,)`

The autograder will test `per_class_accuracy` with a known confusion matrix and verify the output values.

5.3.5 Autograder Summary

File	Required Names	Autograder Tests
`models.py`	`Net3`, `NormSkipBlock`, `NormBMEnet`, `LVISSkipNet`	Forward pass shapes, conv count, layer count, all 4 norm types
`train.py`	`train_one_epoch`, `validate`	Return types (float, float)
`evaluate.py`	`compute_confusion_matrix`, `per_class_accuracy`	CM shape, known-input test

Important Notes:

- Do **not** rename these files or functions.
- Do **not** put executable code at the module level. Guard scripts with `if __name__ == '__main__':` blocks.
- Default argument values must match those shown above.
- You may reuse your HW4 implementations of `train.py` and `evaluate.py` if they match the required signatures.
- **Cite DLStudio source code** wherever you borrow or adapt `SkipBlock` or `BMEnet` code.

5.4 Code Quality Requirements

- Well-commented code explaining key steps
- Meaningful variable and function names
- Docstrings for all classes and functions
- Code should run without modification (given correct paths)
- Follow PEP 8 style guidelines

5.5 Report Formatting

- **Include code snippets in the report for every task.** For each task, show the code you wrote alongside the outputs it produced. If the code snippet is not present, you will lose points on that task.
- Use figures and tables with captions
- Number equations, figures, and tables
- Use proper citations when referencing papers or DLStudio

5.6 Additional Guidelines

- Convert Jupyter notebooks to .py files. **Do NOT submit .ipynb**
- If submitting late, submit only once
- **Do NOT submit dataset files**, only code
- Your code and report must be your own work

6 Frequently Asked Questions (FAQ)

Q: Can I reuse my HW4 code?

A: Yes. You are encouraged to reuse and extend your HW4 implementations. The `train.py` and `evaluate.py` signatures are identical. Reference your HW4 `LVISNet` results in your comparative analysis.

Q: Do I need to re-run Net and Net2 from HW4?

A: No. Use the results you already collected in HW4 as baselines. You only need to run Net3 and BMEnet in this homework.

Q: Should I use the same 5 LVIS categories from HW4?

A: Yes. Use identical categories to enable fair comparison between `LVISNet` (HW4) and `LVISSkipNet` (HW5). If you need to change a category because it does not meet the minimum image count, document it clearly.

Q: Can I use a learning rate scheduler?

A: Yes, but use it consistently across all experiments for fair comparison. Document it.

Q: I don't have a GPU. Can I still do this homework?

A: Yes. CIFAR-10 experiments with DLStudio are designed to run on CPU. For `LVISSkipNet`, CPU training will be slower (possibly 15–30 min per epoch) but feasible with 64×64 images. Use Google Colab's free GPU if needed.

Q: Can I work in a group?

A: No. This is individual work.

Q: `playing_with_skip_connections.py` crashes.

A: (1) Run from the Examples directory (paths are relative), (2) ensure CIFAR-10 data is downloaded, (3) reduce batch size if GPU memory is insufficient.

Q: Does Net3 need *exactly* 8 conv layers?

A: Yes, exactly 8 `nn.Conv2d` layers. The autograder will count them.

Q: My spatial dimensions shrink to zero with 8 conv layers.

A: Group convolutions and pool between groups. For example: `conv-conv-pool` (32→16), `conv-conv-pool` (16→8), `conv-conv-pool` (8→4), `conv-conv-pool` (4→2). That gives 8 conv layers with only 4 pooling operations.

Q: My Net3 performs *worse* than Net/Net2. Is that wrong?

A: That is the **degradation problem**, an expected result. Document it.

Q: My Net3 performs *better* than Net/Net2.

A: Possible with only 8 layers. The degradation problem is more pronounced with >20 layers. Report and discuss what you observe.

Q: What does "built from SkipBlock components" mean?

A: Use `SkipBlock` instances as main building blocks (like `BMEnet`). Do **not** use raw `nn.Conv2d` layers. Exception: initial convolution for channel dimension.

Q: Can I modify the SkipBlock from DLStudio?

A: Yes, but cite the original DLStudio source and document modifications.

Q: Dimension mismatch in SkipBlock when input channels ≠ output channels.

A: Study DLStudio's `SkipBlock`: it uses a 1×1 convolution to project the skip connection.

Q: How do I count "learnable layers" vs. "learnable parameters"?

A: Learnable layers = number of Conv2d and Linear modules. Your LVISSkipNet needs ≥40. Each SkipBlock contributes 2, so you need at least 20 SkipBlocks.

Q: `LVISSkipNet` runs out of GPU memory.

A: (1) Reduce batch size to 16 or 8, (2) use fewer channels in early blocks, (3) call `torch.cuda.empty_cache()`, (4) use Google Colab.

Q: Why use `GroupNorm(1, C)` for Layer Normalization instead of `nn.LayerNorm`?

A: `nn.LayerNorm` requires the exact spatial dimensions (C, H, W) at construction time, which is inconvenient for convolutional layers where H and W may vary. `GroupNorm` with 1 group normalizes over all channels per instance, which is mathematically equivalent to LayerNorm over (C, H, W) [2].

Q: `InstanceNorm2d` gives worse results than expected.

A: By default, `nn.InstanceNorm2d` does not have learnable affine parameters. Set `affine=True` so it learns γ and β, matching the BN behavior.

Q: How many groups should I use for Group Normalization?

A: Use $G = 8$ as specified in the task. Make sure the number of channels is divisible by G. If not, reduce G until it divides evenly.

Q: Can I use different normalization types in different parts of the same network?

A: For Task 3, use the *same* normalization type throughout the network in each experiment. In the bonus task, you may experiment with mixing.

Q: My LVIS dataset has imbalanced classes.

A: Follow the balancing requirements in Section 2.4. No class may exceed 1.2× the size of the smallest class. Undersample or augment as needed. Report final per-class counts.

Q: Can I augment the validation set to get more images?

A: **No.** Only augment the training set. The validation set uses resize and normalize only, no random transforms.

Q: A training image and its augmented copy ended up in different splits.

A: This is data leakage. Split **before** augmentation. An image and all its augmented variants must be in the same split (all in train or all in val).

Q: One of my LVIS categories has fewer than 300 images.

A: Either (1) choose a different LVIS category with enough images, or (2) apply offline augmentation to the training split only. Document whichever approach you use.

Q: My feature maps are all black or all white.

A: Normalize each channel independently for display: subtract the min, divide by $(\max - \min + \epsilon)$.

Q: Hooks are not capturing feature maps.

A: Make sure the layer names you pass match exactly. Use `model.named_modules()` to print all available names first.

Q: My feature map visualizations look random.

A: Use your *trained* models. Later layers produce abstract feature maps that may not be visually interpretable, but that is expected.

Q: `LVISSkipNet` accuracy is very low.

A: (1) Check if loss is decreasing, (2) verify normalization consistency, (3) ensure labels are 0–4, (4) try more epochs (10–15), (5) compare against HW4 baseline. **Focus on relative comparisons** between architectures.

Q: Loss becomes NaN.

A: (1) Lower learning rate to `lr=1e-4`, (2) add gradient clipping with `torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)`.

Q: Training is extremely slow with 40+ layers.

A: Verify GPU usage, set `num_workers=2` in DataLoader, use Google Colab.

References

[1] Avinash Kak, *DLStudio Platform*. 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/pdf-kak/DemystifyConvo.pdf

[4] Avinash Kak, *A First Introduction to Torch.nn for Designing Deep Networks and to DLStudio for Experimenting with Them*. Available at: https://engineering.purdue.edu/kak/pdf-kak/TorchnnAndDLStudio.pdf

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, *Deep Residual Learning for Image Recognition*, IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016. Available at: https://arxiv.org/abs/1512.03385

[6] Agrim Gupta, Piotr Dollár, and Ross Girshick, *LVIS: A Dataset for Large Vocabulary Instance Segmentation*, IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2019. Available at: https://arxiv.org/abs/1908.03195