BME646 / ECE60146 — Homework 4 Report (Spring 2026)

Yupeng Zhou

Team:

1 DLStudio Exploration (Task 1)

1.1 Environment and Execution

DLStudio version: 2.5.5
Script executed: `playing_with_cifar10.py`

1.2 CIFAR-10 Results

Figure 1: Training loss curve produced by Net.

Table 1: CIFAR-10 test accuracy summary from the Net.

Class	**Accuracy**
plane	55%
car	73%
bird	40%
cat	35%
deer	28%
dog	29%
frog	68%
horse	65%
ship	69%
truck	73%

Table 2: Confusion matrix printed by the script (rows = ground truth, columns = predicted), in %.

	plane	car	bird	cat	deer	dog	frog	horse	ship	truck
plane	55.40	7.60	8.80	1.90	1.50	0.60	1.60	2.00	12.00	8.60
car	1.40	73.50	1.00	0.50	0.20	0.10	0.60	0.90	4.10	17.70
bird	6.30	3.10	40.10	9.80	8.60	3.90	15.10	6.90	2.70	3.50
cat	2.20	2.50	8.50	35.20	4.60	9.40	13.20	8.30	2.60	13.50
deer	2.60	3.80	12.80	5.90	28.90	1.70	22.20	16.20	2.10	3.80
dog	0.90	1.40	9.20	26.00	4.20	29.60	6.60	13.10	1.30	7.70
frog	0.60	2.10	4.90	8.50	2.20	0.90	68.20	3.20	0.90	8.50
horse	0.60	1.70	4.80	4.70	5.40	3.60	2.90	65.90	0.70	9.70
ship	6.90	9.70	2.50	1.80	0.60	0.40	0.60	1.20	69.30	7.00
truck	1.70	16.60	1.00	1.10	0.00	0.20	1.20	1.40	3.30	73.50

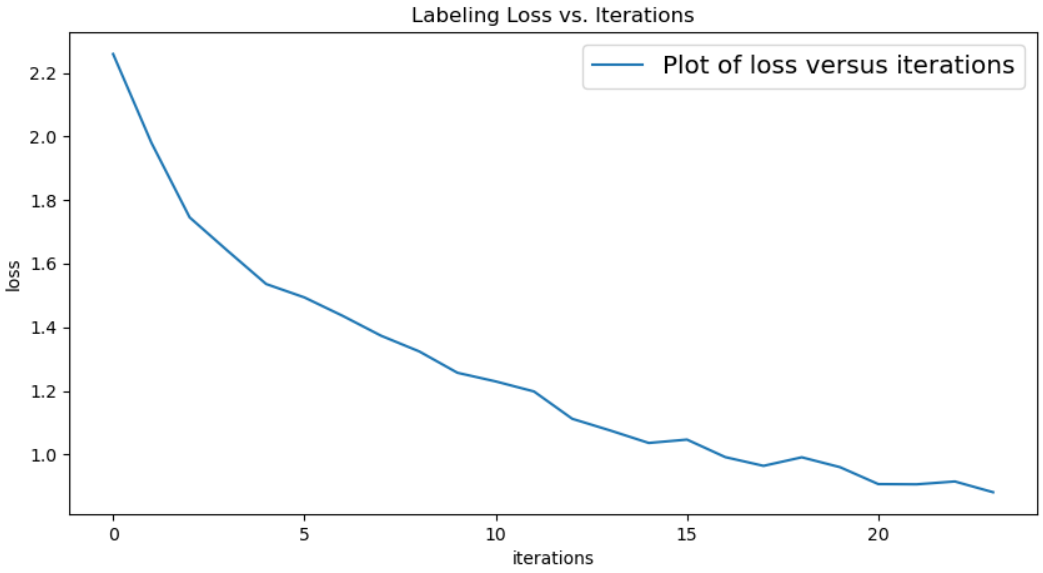

Figure 2: Training loss curve produced by Net2.

Table 3: CIFAR-10 test accuracy summary from the Net2.

Metric	**Value**
Overall test accuracy	68%

Table 4: Per-class test accuracy reported by Net2.

Class	**Accuracy**
plane	75%
car	76%
bird	70%
cat	45%
deer	67%
dog	39%
frog	73%
horse	62%
ship	87%
truck	82%

Table 5: Confusion matrix printed by the script (rows = ground truth, columns = predicted), in %.

	plane	car	bird	cat	deer	dog	frog	horse	ship	truck
plane	75.50	1.20	7.70	0.90	1.80	0.30	0.60	0.20	7.60	4.20
car	3.40	76.30	0.50	0.10	0.50	0.00	0.70	0.00	5.00	13.50
bird	7.10	0.70	70.60	3.80	7.00	1.70	4.60	0.50	2.20	1.80
cat	3.90	1.30	15.80	45.80	8.30	4.70	8.70	1.40	5.40	4.70
deer	3.80	0.40	14.40	4.40	67.30	0.60	4.00	2.20	2.40	0.50
dog	2.00	0.40	16.40	22.20	8.00	39.90	3.80	2.10	2.60	2.60
frog	1.20	0.70	12.20	3.70	4.20	0.20	73.00	0.20	2.30	2.30
horse	2.70	0.30	9.80	3.10	12.60	2.80	1.00	62.60	1.20	3.90
ship	5.80	1.90	1.70	0.30	0.40	0.00	0.40	0.10	87.30	2.10
truck	3.80	6.30	1.30	0.30	0.50	0.00	0.20	0.10	4.70	82.80

Quick observation: The largest confusions are between *car* ↔ *truck* (17.7% of cars predicted as trucks; 16.6% of trucks predicted as cars) and *plane* → *ship* (12.0%).

1.3 Network Architecture Analysis

The CIFAR-10 example used a CNN (DLStudio `Net2`) with three convolution blocks and three fully-connected layers. Input images are RGB 32×32.

Table 6: Net2 layer-by-layer tensor shapes (C, H, W).

#	Layer	Input	Output
1	Conv2d(3→128, 5×5, pad=2, stride=1)	(3, 32, 32)	(128, 32, 32)
2	ReLU	(128, 32, 32)	(128, 32, 32)
3	MaxPool2d(2×2, stride=2)	(128, 32, 32)	(128, 16, 16)
4	Conv2d(128→128, 3×3, pad=1, stride=1)	(128, 16, 16)	(128, 16, 16)
5	ReLU	(128, 16, 16)	(128, 16, 16)
6	MaxPool2d(2×2, stride=2)	(128, 16, 16)	(128, 8, 8)
7	Conv2d(128→128, 2×2, pad=1, stride=1)	(128, 8, 8)	(128, 9, 9)
8	ReLU	(128, 9, 9)	(128, 9, 9)
9	MaxPool2d(2×2, stride=1)	(128, 9, 9)	(128, 8, 8)
10	Flatten	(128, 8, 8)	(8192)
11	Linear(8192→150)	(8192)	(150)
12	ReLU	(150)	(150)
13	Linear(150→100)	(150)	(100)
14	ReLU	(100)	(100)
15	Linear(100→10)	(100)	(10)

Spatial trace (H×W): $32 \times 32 \to 16 \times 16 \to 8 \times 8 \to 9 \times 9 \to 8 \times 8$.

Note on the FC input size: The script computes `in_size_for_fc` $= 128 \times (32/(2 \cdot 2))^2 = 128 \times 8^2 = 8192$, which matches the final pooled feature map size.

1.4 torch.nn Concepts

- `nn.Conv2d`**:** Applies learnable 2D convolution filters to an input tensor shaped (batch, channels, height, width). The number of output channels equals the number of filters, while kernel size, stride, and padding control receptive field and spatial output size.
- `nn.MaxPool2d`**:** Downsamples spatial dimensions by taking the maximum value in each pooling window. This reduces computation, increases the effective receptive field, and adds some translation robustness.
- `view()` **/ flatten:** Convolution blocks output 4D tensors, but fully-connected layers expect 2D tensors (batch, features). Flattening reshapes each feature map stack into a single feature vector per image.
- `nn.CrossEntropyLoss`**:** Combines a softmax operation with negative log-likelihood loss. It expects raw logits (unnormalized scores) and integer class labels, making it standard for multi-class classification.

1.5 Trainable Parameter Count

For Net2, the total number of trainable parameters is:

$$\boxed{1{,}468{,}036}$$

Table 7: Trainable parameters per layer for Net2.

Layer	**# Params**
Conv1: 3→128, 5×5	9,728
Conv2: 128→128, 3×3	147,584
Conv3: 128→128, 2×2	65,664
FC1: 8192→150	1,228,950
FC2: 150→100	15,100
FC3: 100→10	1,010
Total	**1,468,036**

Example convolution calculation (Conv2):

$$\#\text{params} = C_{out} \cdot (C_{in} \cdot k^2 + 1) = 128 \cdot (128 \cdot 3^2 + 1) = 128 \cdot (1152 + 1) = 147{,}584$$

Example fully-connected calculation (FC1):

$$\#\text{params} = d_{out} \cdot (d_{in} + 1) = 150 \cdot (8192 + 1) = 150 \cdot 8193 = 1{,}228{,}950$$

2 LVIS Dataset Creation (Task 2)

2.1 Selected Categories

I used the same 5 LVIS categories across all three dataset variants. All selected categories belong to the **Frequent (f)** group, which provides a large pool of candidate images for constructing balanced train/validation splits.

Table 8: Selected LVIS categories used for all dataset variants.

Category Name	**Frequency Group**
blanket	Frequent (f)
elephant	Frequent (f)
giraffe	Frequent (f)
motorcycle	Frequent (f)
pizza	Frequent (f)

Justification: I selected these five categories because they are all in the *Frequent* group, ensuring a large pool of candidate images (over 1,900 instances per class). This makes it practical to meet the requirement of collecting **500 distinct images per dataset** while keeping the class distribution balanced.

In addition, these categories also satisfy the requirement for both datasets 1, 2, and 3.

2.2 Dataset 1: Single-Instance Images

Filtering criteria (as implemented):

- Image contains exactly one instance of the target category.
- Target instance area $\geq$ **5%** of image area (or your relaxed threshold).
- Images are de-duplicated across classes so the same image ID appears in only one class.

Train/Val split: Target is 400 train + 100 val per class when available; otherwise I used an 80/20 split and documented the final sizes.

2.3 Dataset 2: Multi-Instance Same Category

Filtering criteria:

- Image contains ≥ 2 instances of the target category.
- Combined target area $\geq$ **10%** of image area (or your chosen threshold).

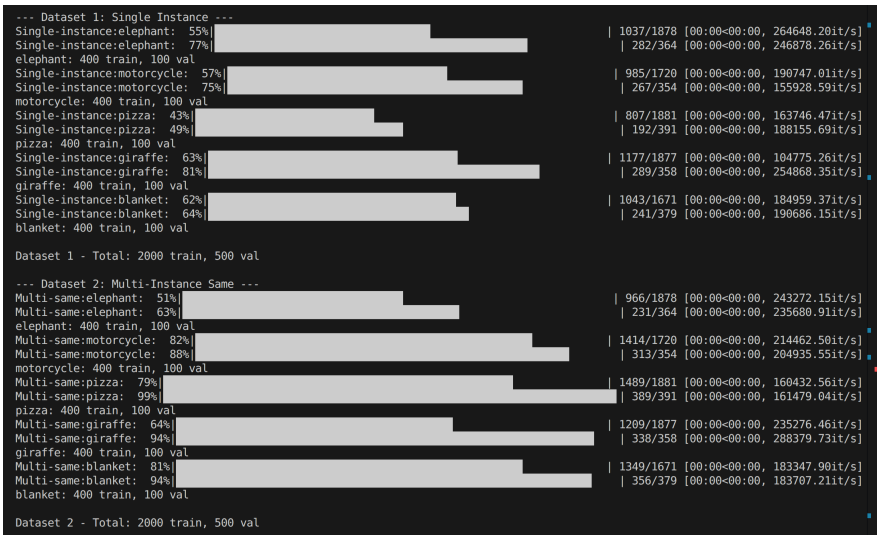

Figure 3: Created datasets 1 and 2.

2.4 Dataset 3: Multi-Instance Different Categories

Filtering criteria:

- Image contains the target category and at least one other category from the selected five.
- Primary class assignment uses **largest total instance area** among the five categories.
- Primary category area / image area $\geq$ **1%** (or your chosen threshold).

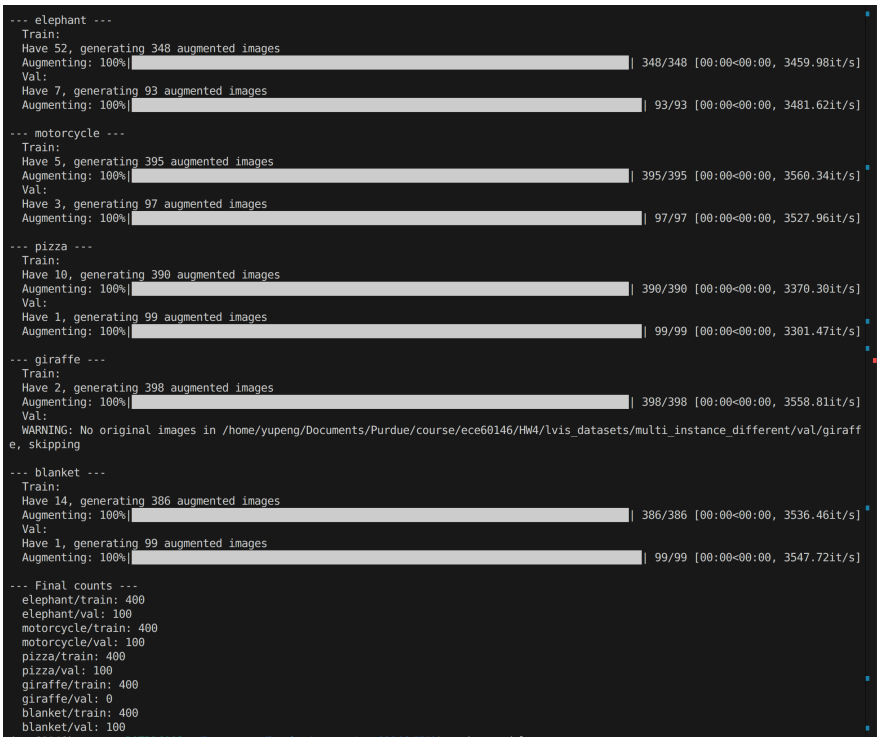

Figure 4: Created dataset 3.

3 Image Processing and Storage (Task 3)

3.1 Directory Layout

I stored datasets in the required directory structure:

```
lvis_datasets/
  single_instance/
    train/<category_name>/*.jpg
    val/<category_name>/*.jpg
  multi_instance_same/
    train/<category_name>/*.jpg
    val/<category_name>/*.jpg
  multi_instance_different/
    train/<category_name>/*.jpg
    val/<category_name>/*.jpg
```

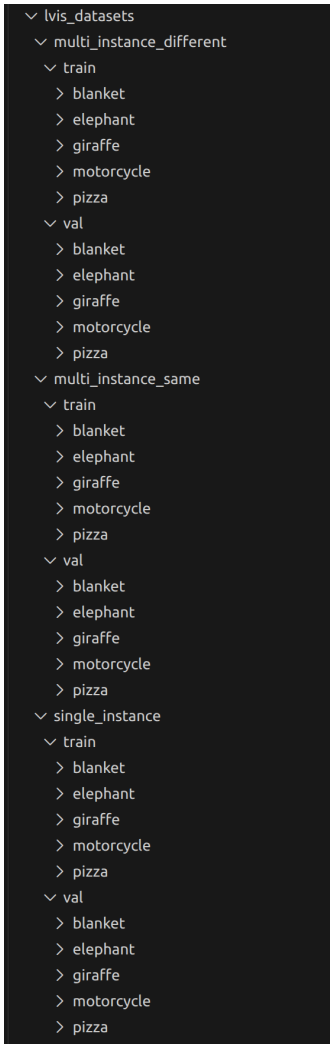

Figure 5: Created dataset structure.

3.2 Preprocessing and Augmentation

Preprocessing:

- Resized all images to **64×64** using bilinear/bicubic interpolation.
- Preserved RGB format (3 channels).

3.3 Data Augmentation for Class Balancing (Multi-Instance Different)

To ensure balanced class sizes in the `multi_instance_different` LVIS dataset, I used a script that augments the dataset across five categories: **elephant, motorcycle, pizza, giraffe,** and **blanket**. The goal was to pad each category to a fixed size of **400 training images** and **100 validation images**. When a class contained fewer images than the target, the script generated synthetic samples from existing images in that same class/split until the target count was reached.

Augmentation pipeline: Each augmented image was created by randomly applying a subset of the following transformations:

- **Horizontal flip** with 50% probability.
- **Rotation** up to ±30° with 70% probability.
- **Color jitter** (brightness, contrast, saturation), each with 50% probability using a factor range of 0.7–1.3.
- **Gaussian blur** with 30% probability.
- **Random crop and resize** back to 64×64 with 60% probability.

Augmentation procedure: For each split (`train` and `val`) and each category folder, the script: (i) counted the number of existing images, (ii) computed how many additional images were required to reach the target size, and (iii) repeatedly selected a random original image, applied the augmentation pipeline, and saved the output with a sequential filename. This process continued until every category reached the desired number of images in both splits.

3.4 Final Dataset Sizes

Table 9: Final dataset sizes per class (fill in with your counts).

Dataset	**Split**	Category 1	Category 2	Category 3	Category 4	Category 5
Single-instance	Train	400	400	400	400	400
Single-instance	Val	100	100	100	100	100
Multi-same	Train	400	400	400	400	400
Multi-same	Val	100	100	100	100	100
Multi-different	Train	400	400	400	400	400
Multi-different	Val	100	100	100	100	100

3.5 Sample Images

To stay under 20 pages, I combined each dataset type into a single 3×5 grid figure (15 samples).

Figure 6: Single-instance dataset: 3 samples per class (3×5 grid).

Figure 7: Multi-instance same-category dataset: 3 samples per class (3×5 grid).

Figure 8: Multi-instance different-category dataset: 3 samples per class (3×5 grid).

4 CNN Architecture Implementation (Task 4)

4.1 LVISNet Design

I implemented a CNN named `LVISNet` for **5-class** classification on 64×64 **RGB** images. The network contains three convolution blocks (Conv + BatchNorm + pooling), followed by three fully-connected layers. Dropout ($p = 0.5$) is applied after the first two fully-connected layers to reduce overfitting. ReLU activations are used between learnable layers in the forward pass.

4.2 Layer-by-Layer Architecture and Tensor Shapes

The input tensor shape is `(batch, 3, 64, 64)` and the output logits have shape `(batch, 5)`.

Table 10: LVISNet architecture and tensor shapes (C, H, W).

#	Layer	Input	Output
1	Conv2d(3→32, 3×3, stride=1, pad=1)	(3, 64, 64)	(32, 64, 64)
2	BatchNorm2d(32)	(32, 64, 64)	(32, 64, 64)
3	MaxPool2d(2×2, stride=2)	(32, 64, 64)	(32, 32, 32)
4	Conv2d(32→64, 3×3, stride=1, pad=1)	(32, 32, 32)	(64, 32, 32)
5	BatchNorm2d(64)	(64, 32, 32)	(64, 32, 32)
6	MaxPool2d(2×2, stride=2)	(64, 32, 32)	(64, 16, 16)
7	Conv2d(64→128, 3×3, stride=1, pad=1)	(64, 16, 16)	(128, 16, 16)
8	BatchNorm2d(128)	(128, 16, 16)	(128, 16, 16)
9	MaxPool2d(2×2, stride=2)	(128, 16, 16)	(128, 8, 8)
10	Flatten	(128, 8, 8)	(8192)
11	Linear(8192→256)	(8192)	(256)
12	Dropout($p = 0.5$)	(256)	(256)
13	Linear(256→128)	(256)	(128)
14	Dropout($p = 0.5$)	(128)	(128)
15	Linear(128→5)	(128)	(5)

Spatial dimension trace (H×W): $64 \times 64 \rightarrow 32 \times 32 \rightarrow 16 \times 16 \rightarrow 8 \times 8$, with channels $3 \rightarrow 32 \rightarrow 64 \rightarrow 128$.

4.3 Trainable Parameter Count

The printed parameter total for `LVISNet` is:

2,224,645 trainable parameters

Table 11: Trainable parameters per LVISNet layer (including biases; BatchNorm counts only trainable γ, β).

Layer	Specification	# Params
conv1	$32 \cdot (3 \cdot 3 \cdot 3 + 1)$	896
bn1	$2 \cdot 32$	64
conv2	$64 \cdot (32 \cdot 3 \cdot 3 + 1)$	18,496
bn2	$2 \cdot 64$	128
conv3	$128 \cdot (64 \cdot 3 \cdot 3 + 1)$	73,856
bn3	$2 \cdot 128$	256
fc1	$256 \cdot (8192 + 1)$	2,097,408
fc2	$128 \cdot (256 + 1)$	32,896
fc3	$5 \cdot (128 + 1)$	645
Total		**2,224,645**

Sanity check: For a batch input of shape `[4, 3, 64, 64]`, the network produces output logits of shape `[4, 5]`, matching the 5-class classification requirement.

```
(ece60146) yupeng@POTRPC103:~/Documents/Purdue/course/ece60146/HW4$ python model.py
 Network : LVISNet(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=8192, out_features=256, bias=True)
  (dropout1): Dropout(p=0.5, inplace=False)
  (fc2): Linear(in_features=256, out_features=128, bias=True)
  (dropout2): Dropout(p=0.5, inplace=False)
  (fc3): Linear(in_features=128, out_features=5, bias=True)
)
 Total parameters :  2,224,645
 Input shape : torch.Size([4, 3, 64, 64])
 Output shape : torch.Size([4, 5])
```

Figure 9: Architecture implementation.

5 Custom Dataset and DataLoader (Task 5)

Implementation notes:

- Implemented a custom `torch.utils.data.Dataset` that reads images from the directory structure and returns (image, label).
- Applied transforms: resize to 64×64, conversion to tensor, and normalization.
- Verified class balance and label mapping (0–4).

```
================================================
Dataset Verification
================================================

Training samples: 2000
Validation samples: 500
Classes: ['blanket', 'elephant', 'giraffe', 'motorcycle', 'pizza']
Class-to-idx: {'blanket': 0, 'elephant': 1, 'giraffe': 2, 'motorcycle': 3, 'pizza': 4}

--- Images Per Class (Train) ---
  blanket: 400
  elephant: 400
  giraffe: 400
  motorcycle: 400
  pizza: 400

--- Images Per Class (Val) ---
  blanket: 100
  elephant: 100
  giraffe: 100
  motorcycle: 100
  pizza: 100

--- DataLoader Check ---
Batch image shape: torch.Size([32, 3, 64, 64])
Batch image dtype: torch.float32
Pixel value range: [0.0000, 1.0000]
Train batches: 63, Val batches: 16

================================================
All dataset checks passed!
================================================
```

Figure 10: Example batch samples after transforms (from training set).

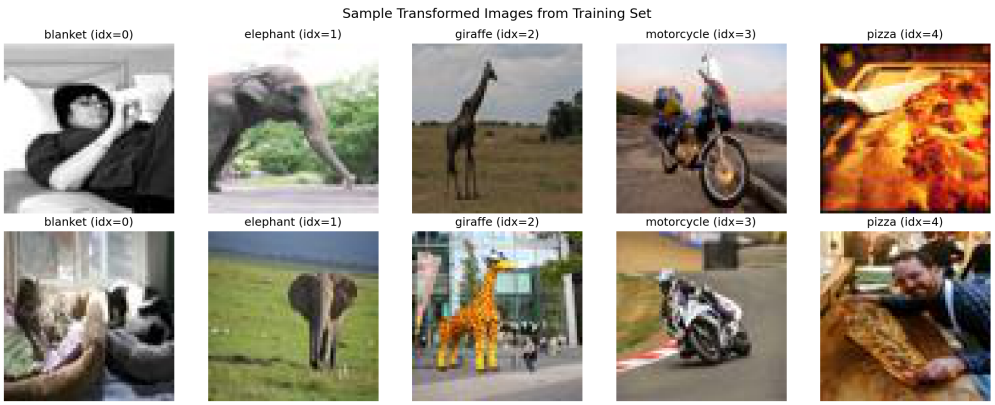

Figure 11: Sample images through Data loader.

6 Training Results (Task 6)

6.1 Training Configuration

- Optimizer: Adam, lr=1×10^{-3}, betas=(0.9, 0.99)
- Loss: CrossEntropyLoss
- Epochs: 7, Batch size: 32

- Device: CPU/GPU (state what you used)

6.2 Learning Curves

To save space, combine loss/accuracy curves into one figure per dataset (or a single multi-panel collage).

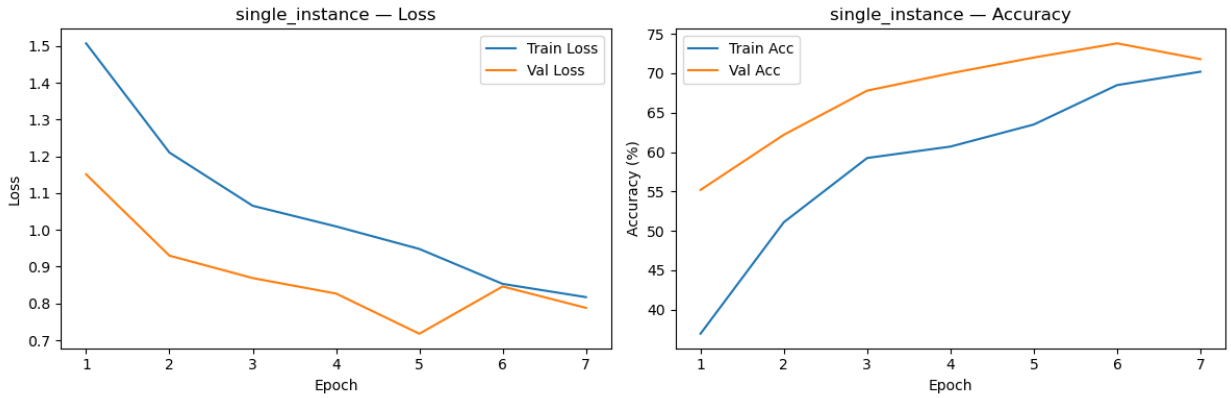

Figure 12: Training/validation curves for single-instance.

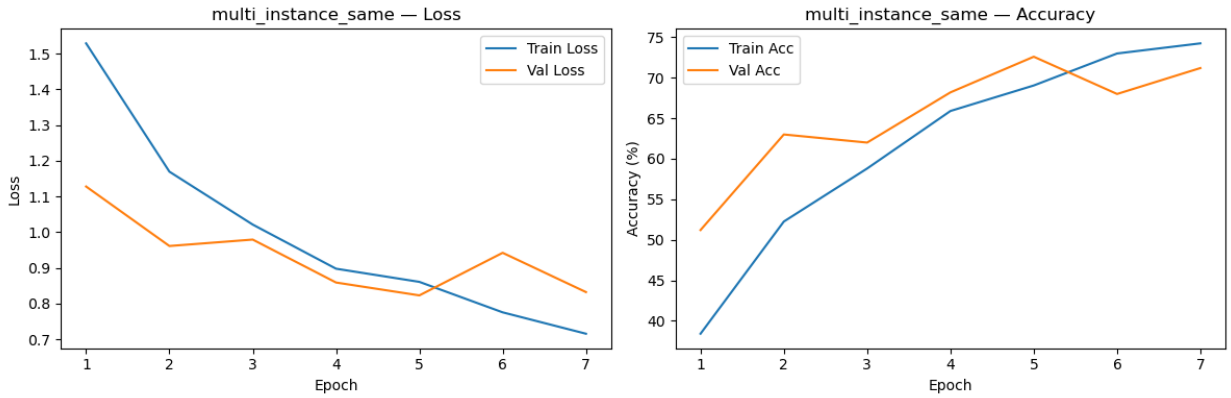

Figure 13: Training/validation curves for multi-same.

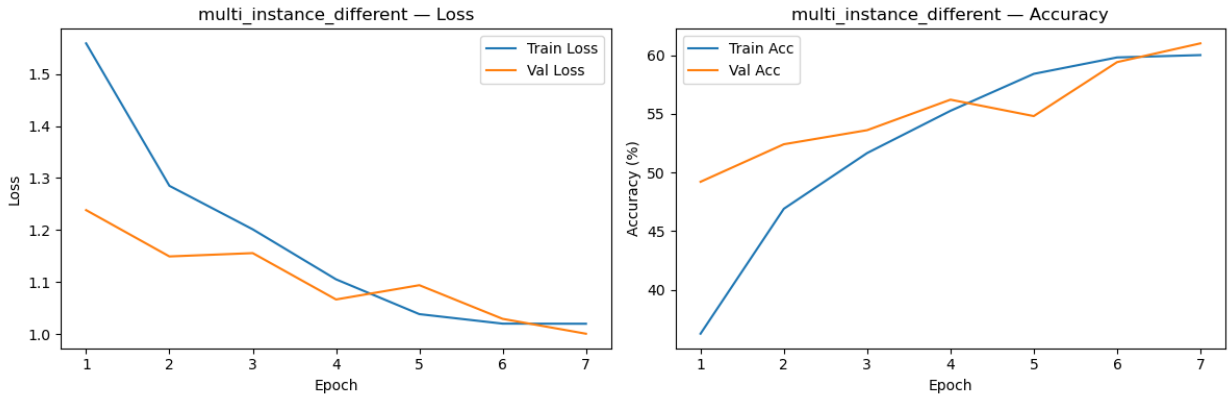

Figure 14: Training/validation curves for multi-different.

6.3 Results Summary

Table 12: Training time and final performance summary for LVISNet (7 epochs, batch size 32).

Dataset	**Train Loss**	**Train Acc (%)**	**Val Loss**	**Val Acc (%)**	**Train Time**
Single-instance	0.8172	70.20	0.7881	71.80	3.1s
Multi-instance (same)	0.7152	74.25	0.8319	71.20	3.1s
Multi-instance (different)	1.0198	60.00	1.0006	61.00	3.1s
Total					**9.3s**

7 Confusion Matrices (Task 7)

I computed confusion matrices *from scratch* (no sklearn for computation) and report both overall and per-class accuracies.

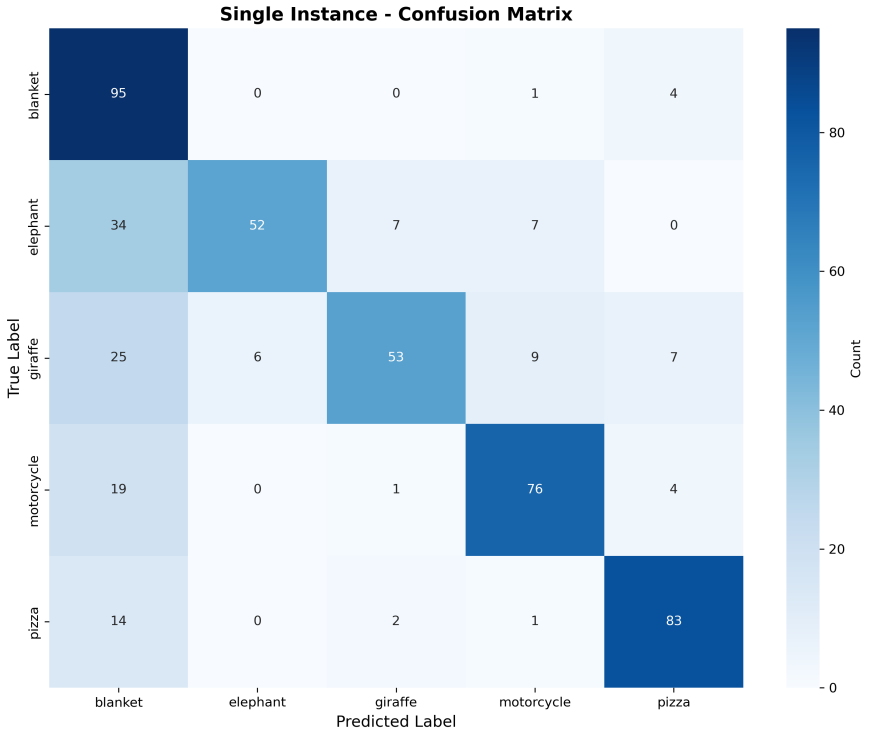

Figure 15: Training/validation curves for single-instance.

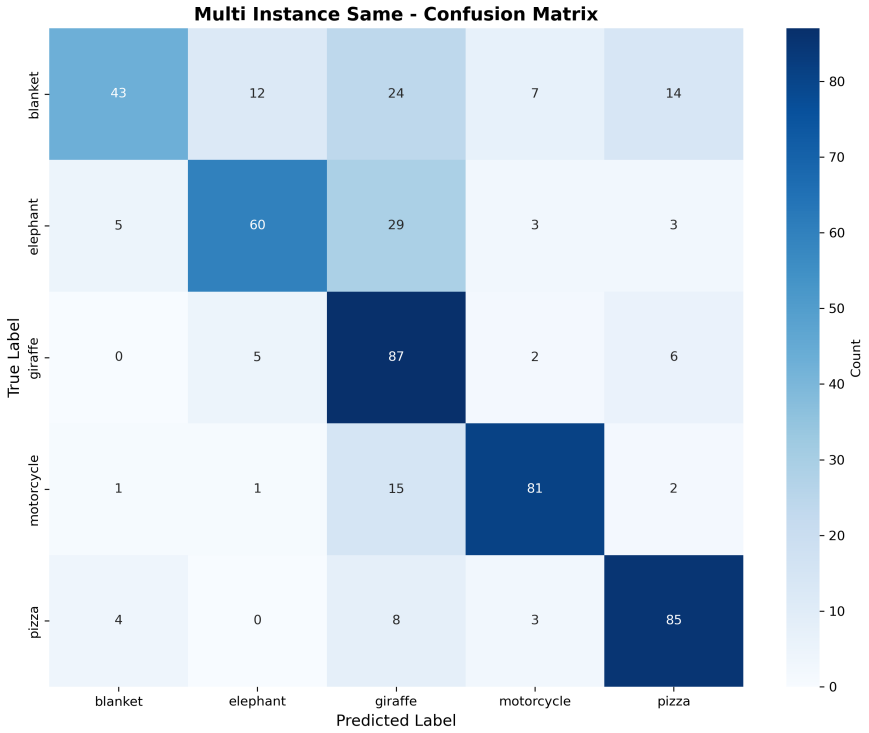

Figure 16: Training/validation curves for multi-same.

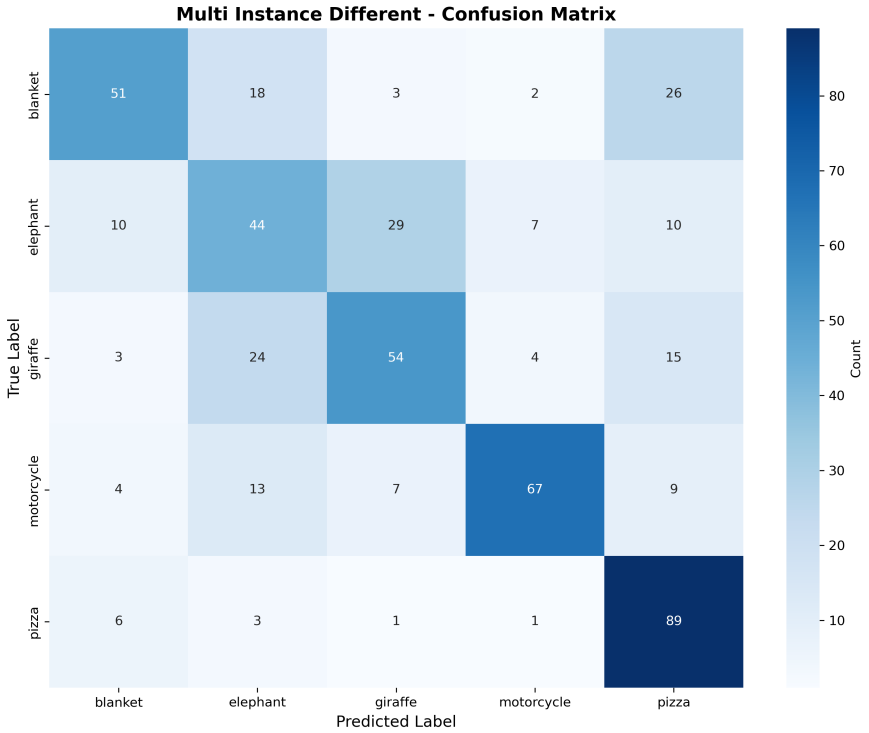

Figure 17: Training/validation curves for multi-different.

Table 13: Per-class validation accuracy for each dataset variant.

Dataset	blanket	elephant	giraffe	motorcycle	pizza
Single-instance	95.00	52.00	53.00	76.00	83.00
Multi-same	43.00	60.00	87.00	81.00	85.00
Multi-different	51.00	44.00	54.00	67.00	89.00

7.1 Error Analysis

The most common errors occurred between visually similar or context-dependent classes, especially when the target object occupied a small portion of the image or was partially occluded by other objects in the scene. Across all three datasets, **elephant** and **giraffe** showed the lowest accuracies (roughly mid-40% to low-50% range), which is consistent with the fact that both are large animals with similar texture/shape cues and are often photographed in cluttered outdoor backgrounds. In contrast, **pizza** achieved the highest accuracy in every dataset (83–89%), likely because it has distinctive color/shape patterns and typically appears near the center of the image. The **blanket** class performed very well in the single-instance dataset (95%), but dropped substantially in multi-object settings, suggesting that blankets are often treated as background context or become ambiguous when other objects dominate the image. Overall, confusion increased in the *multi-instance different* dataset, where multiple categories co-occur and the primary label may not correspond to the most visually salient object, making classification more difficult.

8 Discussion and Conclusion

Key observations: (bullets or short paragraph)

- Single-instance vs multi-same vs multi-different difficulty comparison.
- Effects of area thresholds / augmentation on data quality and accuracy.
- Overfitting vs generalization gap (train vs val).

Challenges: Mention any implementation issues (LVIS naming, scarcity, speed, GPU memory) and how you resolved them.

Optional Bonus

8.1 Model Size (Parameter Count)

Table 14: Parameter count comparison.

Model	# Parameters
Baseline	2,224,645
Deep5	2,128,645
Deep10	7,625,797

8.2 Training Time per Epoch

Table 15: Training time per epoch (seconds).

Epoch	Baseline	Deep5	Deep10
1	0.6	0.5	0.7
2	0.4	0.5	0.6
3	0.4	0.4	0.6
4	0.4	0.4	0.6
5	0.4	0.4	0.6
6	0.4	0.4	0.6
7	0.4	0.5	0.6
Avg	0.4	0.5	0.6

8.3 Training Curves

Figure 18 shows the comparative training curves for Baseline vs. Deep5 vs. Deep10.

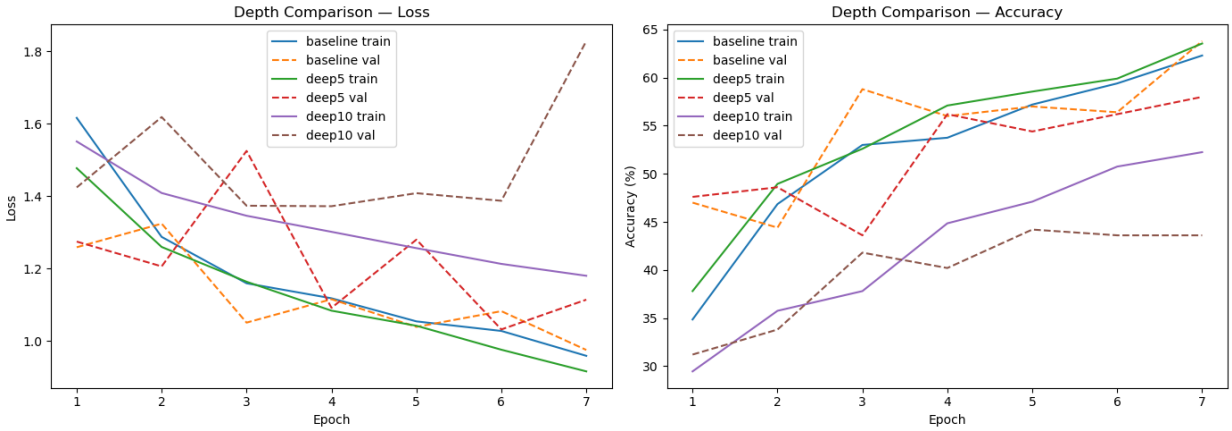

Figure 18: Training/validation curves comparison (Baseline vs. Deep5 vs. Deep10).

Table 16: Validation loss and overall accuracy.

Model	**Val Loss**	**Overall Acc (%)**
Baseline	0.9756	63.80
Deep5	1.1143	58.00
Deep10	1.8279	43.60

8.4 Validation Performance

8.5 Per-Class Accuracy

8.6 Confusion Matrices

We report confusion matrices for all three models in Figure **??**.

Table 17: Per-class accuracy on the validation set. (Each class has 100 validation samples.)

Class	**Baseline (%)**	**Deep5 (%)**	**Deep10 (%)**
blanket	56	82	15
elephant	56	62	19
giraffe	57	36	20
motorcycle	79	43	95
pizza	71	67	69

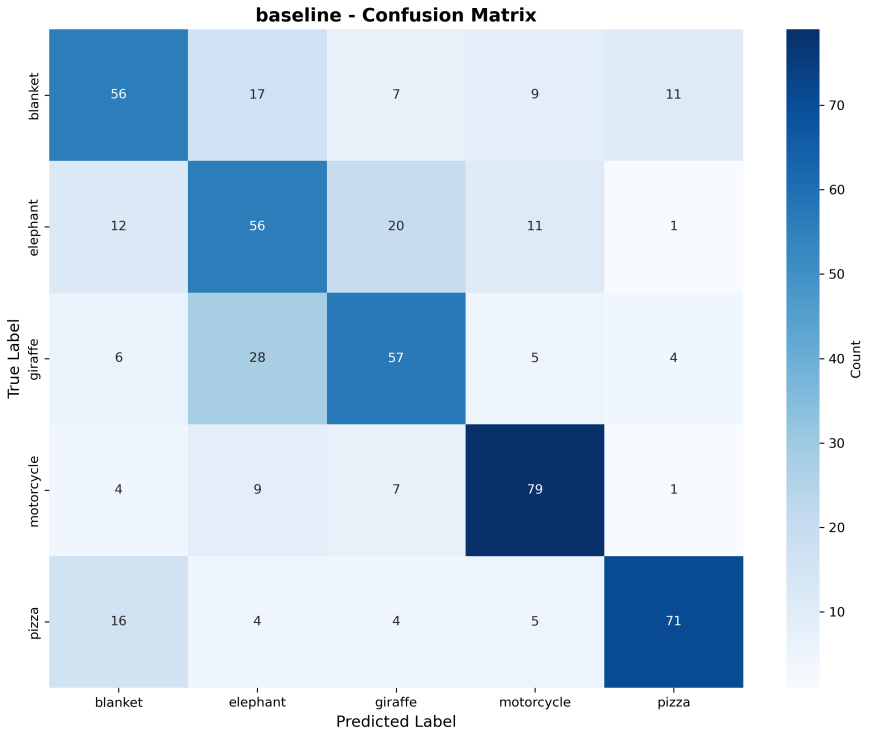

Figure 19: Baseline.

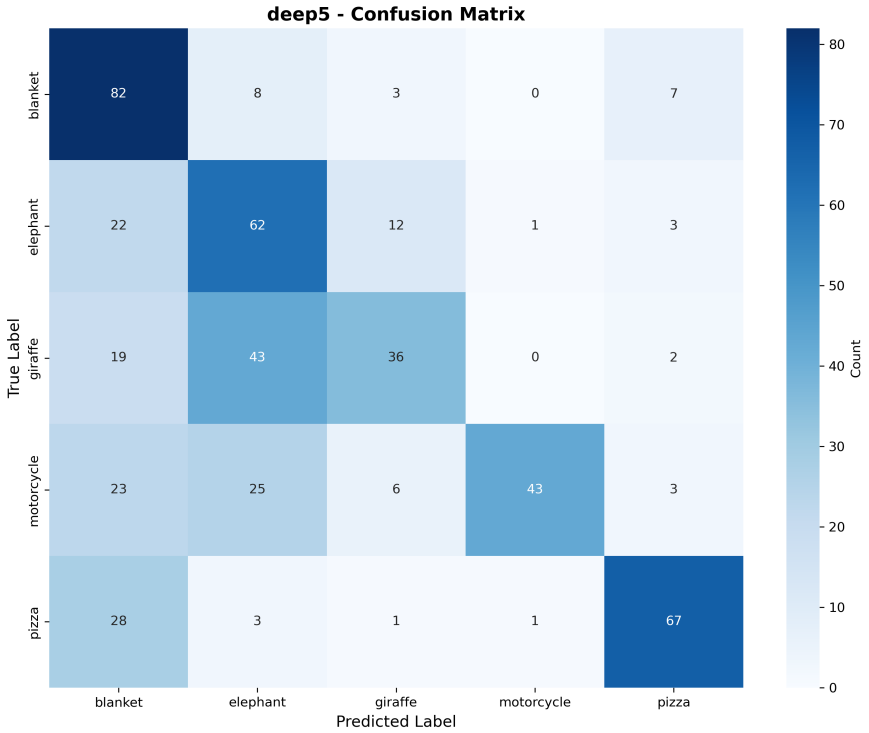

Figure 20: Deep5.

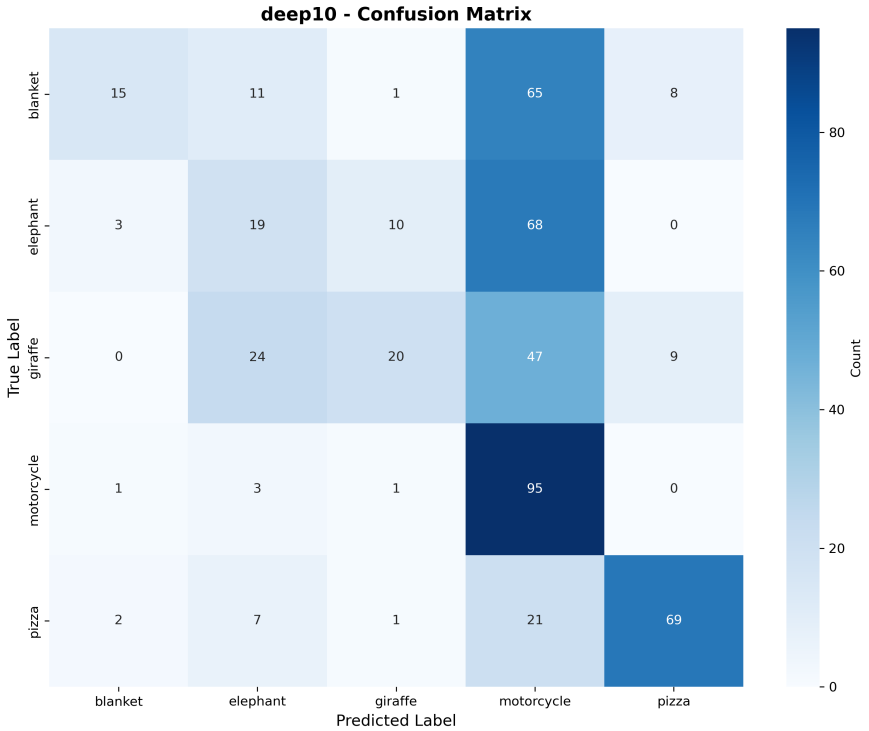

Figure 21: Deep10.

8.7 Analysis: Does Depth Help? What Challenges Arise?

Overall, increasing depth did **not** improve validation performance in this setting. The baseline achieved the best overall accuracy (63.8%) and lowest validation loss (0.9756), while Deep5 dropped to 58.0% and Deep10 further dropped to 43.6%. Although Deep10 is substantially larger (7.63M parameters), its generalization degraded and it exhibited more unstable validation behavior (highest validation loss at 1.8279).

A notable behavior is **class imbalance in learned performance across classes** for deeper models: Deep10 achieved very high accuracy on `motorcycle` (95%) and moderate performance on `pizza` (69%), but performed poorly on `blanket`, `elephant`, and `giraffe` (15–20%). This suggests that the deeper network may be over-specializing or converging to features that separate only a subset of classes well, while confusing others.

Key challenges with deeper networks under the *same training configuration* include:

- **Optimization difficulty**: deeper stacks can be harder to train without additional techniques (e.g., residual connections, normalization strategies, careful initialization, tuned learning rate schedules).
- **Overfitting risk / poorer generalization**: more parameters (Deep10) can memorize or latch onto spurious cues, especially with limited data and fixed regularization.
- **Compute cost**: training time per epoch increased with depth (avg. 0.4s $\rightarrow$ 0.5s $\rightarrow$ 0.6s for Baseline, Deep5, Deep10).