Computer Vision

Assignment - 1 Report

Samridh Girdhar
2021282

Theory Questions

1. (a) Deriving the Cross-Entropy with Label Smoothing

Setup:

- We have K classes.
- A true (one-hot) label vector: $\mathbf{y}$, where exactly one entry is 1 and the rest are 0.
- A predicted probability distribution $\mathbf{q} = [q_1, q_2, \ldots, q_K]$ with $\sum_{i=1}^{K} q_i = 1$, $q_i \geq 0$.
- A label-smoothing parameter ε.
- The "smoothed" target distribution $\mathbf{p}$ is defined as:

$$p_i = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{K}, & \text{if } i = \text{correct class}, \\ \frac{\varepsilon}{K}, & \text{otherwise.} \end{cases}$$

Cross-Entropy Definition: The cross-entropy between a target distribution $\mathbf{p}$ and a predicted distribution $\mathbf{q}$ is:

$$H(\mathbf{p}, \mathbf{q}) = E_{X \sim \mathbf{p}}\big[-\log q(X) \big] = -\sum_{i=1}^{K} p_i \log q_i.$$

Substituting the Smoothed Targets: With label smoothing, the target p_i is not strictly $(0, 0, \ldots, 1, \ldots)$. Instead:

- For the correct class c:

$$p_c = 1 - \varepsilon + \frac{\varepsilon}{K}.$$

- For each incorrect class $j \neq c$:

$$p_j = \frac{\varepsilon}{K}.$$

Hence, the cross-entropy loss becomes:

$$\begin{aligned} H(\mathbf{p}, \mathbf{q}) &= -\sum_{i=1}^{K} p_i \log q_i \\ &= -\left(\big(1 - \varepsilon + \tfrac{\varepsilon}{K}\big) \log q_c \;\; + \;\; \sum_{j \neq c} \tfrac{\varepsilon}{K} \log q_j \right). \end{aligned}$$

This can be written more compactly as:

$$H(\mathbf{p}, \mathbf{q}) = -\big(1 - \varepsilon + \tfrac{\varepsilon}{K}\big) \log q_c \;-\; \tfrac{\varepsilon}{K} \sum_{j \neq c} \log q_j.$$

1. (b) Effect of Label Smoothing on the Loss and Its Interpretation

Reduced Overconfidence:

- Without label smoothing, the model tries to push the probability of the correct class very close to 1, making the predicted distribution extremely "peaky."
- With label smoothing, the target distribution never assigns the full probability mass (i.e., exactly 1) to a single class. Instead, a small fraction $\frac{\varepsilon}{K}$ is assigned to each of the other classes.
- As a result, the model is penalized for being overly confident, since it no longer benefits from pushing its predicted probability for the correct class arbitrarily close to 1.

Regularization and Generalization:

- Label smoothing acts as a form of regularization. By preventing the model from putting too much confidence in a single class, it helps reduce overfitting.
- This makes the model's gradients less extreme and can improve generalization performance—especially on classes with few training examples or when the model might otherwise memorize the training labels.

Interpretation:

- Conceptually, label smoothing encourages the model to consider alternate classes to some (small) degree, making its output distribution slightly "softer."
- Mathematically, this discourages the KL divergence between the predicted distribution $\mathbf{q}$ and the one-hot label $\mathbf{y}$ from being minimized too aggressively, thereby improving calibration (i.e., making the predicted probabilities better reflect actual correctness likelihood).

Key Takeaways:

1. **Cross-entropy formula** with smoothing replaces the one-hot target with a "mostly correct" plus a small uniform probability over the incorrect classes.
2. **Regularization benefit**: It prevents extreme confidence, can improve calibration, and helps avoid overfitting.

2. (a) Cross-Entropy as an Expectation

We have two univariate Gaussian (normal) distributions:

$$p(x) = \mathcal{N}(\mu_p, \sigma_p^2), \quad q(x) = \mathcal{N}(\mu_q, \sigma_q^2).$$

The probability density function (PDF) of a normal distribution $\mathcal{N}(\mu, \sigma^2)$ is given by:

$$\mathcal{N}(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\,\sigma^2}} \exp\Big(-\tfrac{(x-\mu)^2}{2\sigma^2}\Big).$$

The cross-entropy between $p(x)$ and $q(x)$ is defined as:

$$H(p, q) = E_{X\sim p}\big[-\ln q(X)\big] = -\int_{-\infty}^{\infty} p(x)\, \ln q(x)\, dx.$$

For univariate Gaussians:

$$H(p, q) = -\int_{-\infty}^{\infty} p(x)\, \ln\Big[\mathcal{N}(x \mid \mu_q, \sigma_q^2)\Big]\, dx. \,]$$

2. (b) Evaluating $H(p, q)$ in Closed Form

1. **Log of** $q(x)$: Since $q(x)$ is Gaussian, we expand:

$$\ln q(x) = \ln \frac{1}{\sqrt{2\pi\,\sigma_q^2}} - \frac{(x-\mu_q)^2}{2\,\sigma_q^2} = -\frac{1}{2}\ln\big(2\pi\,\sigma_q^2\big) - \frac{(x-\mu_q)^2}{2\,\sigma_q^2}.$$

2. **Substituting into the integral**:

$$H(p, q) = -\int p(x)\, \ln q(x)\, dx = -\int p(x)\Big[-\tfrac{1}{2}\ln(2\pi\,\sigma_q^2) - \tfrac{(x-\mu_q)^2}{2\,\sigma_q^2}\Big]\, dx.$$

$$= \int p(x)\Big[\tfrac{1}{2}\ln(2\pi\,\sigma_q^2) + \tfrac{(x-\mu_q)^2}{2\,\sigma_q^2}\Big]\, dx.$$

Since $\int p(x)\, dx = 1$, we separate terms:

$$H(p, q) = \tfrac{1}{2}\ln(2\pi\,\sigma_q^2) + \tfrac{1}{2\,\sigma_q^2} E_{X\sim p}[(X-\mu_q)^2].$$

3. **Compute** $E_p[(X-\mu_q)^2]$: If $X \sim \mathcal{N}(\mu_p, \sigma_p^2)$, then:

$$E_p[(X-\mu_q)^2] = \mathrm{Var}_p(X) + (E_p(X) - \mu_q)^2 = \sigma_p^2 + (\mu_p - \mu_q)^2.$$

Thus, we obtain:

$$H(p, q) = \tfrac{1}{2}\ln(2\pi\,\sigma_q^2) + \tfrac{1}{2\,\sigma_q^2}\Big[\sigma_p^2 + (\mu_p - \mu_q)^2\Big].$$

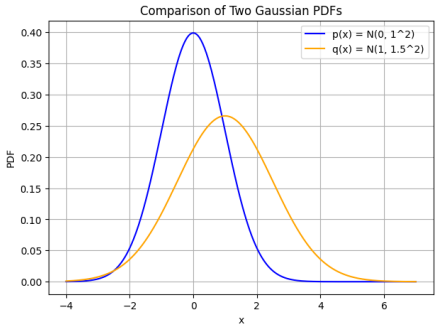


Figure 1:

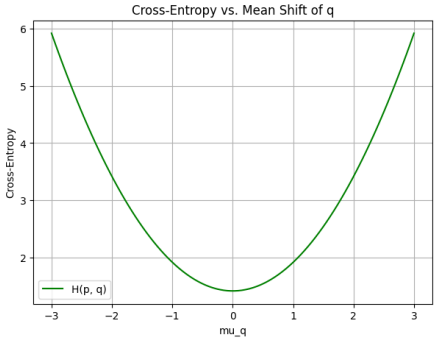


Figure 2:

2. (c) Special Case: $\sigma_p = \sigma_q = \sigma$

If $\sigma_p = \sigma_q = \sigma$, then:

$$H(p,q) = \frac{1}{2}\ln\big(2\pi\,\sigma^2\big) + \frac{1}{2\sigma^2}\Big[\sigma^2 + (\mu_p - \mu_q)^2\Big].$$

Distribute:

$$H(p,q) = \frac{1}{2}\ln(2\pi\,\sigma^2) + \frac{1}{2} + \frac{(\mu_p - \mu_q)^2}{2\,\sigma^2}.$$

Interpretation:

- The first term, $\frac{1}{2}\ln(2\pi\,\sigma^2)$, accounts for the normalizing factor of a Gaussian distribution.
- The second term, $\frac{1}{2}$, represents the contribution of the variance.
- The last term, $\frac{(\mu_p-\mu_q)^2}{2\sigma^2}$, penalizes the difference in means. If $\mu_p = \mu_q$, then this term is zero, meaning the cross-entropy depends only on variance.

Thus, when the variances match, the cross-entropy simplifies to a function of the difference in means, with larger differences leading to larger cross-entropy values.

3. (a) Cross-Entropy as an Expectation

We have two univariate Gaussian (normal) distributions:

$$p(x) = \mathcal{N}(\mu_p, \sigma_p^2), \quad q(x) = \mathcal{N}(\mu_q, \sigma_q^2).$$

The probability density function (PDF) of a normal distribution $\mathcal{N}(\mu, \sigma^2)$ is given by:

$$\mathcal{N}(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\,\sigma^2}}\exp\Big(-\tfrac{(x-\mu)^2}{2\sigma^2}\Big).$$

The cross-entropy between $p(x)$ and $q(x)$ is defined as:

$$H(p,q) = E_{X\sim p}\big[-\ln q(X)\big] = -\int_{-\infty}^{\infty} p(x)\,\ln q(x)\,dx.$$

For univariate Gaussians:

$$H(p,q) = -\int_{-\infty}^{\infty} p(x)\,\ln\Big[\mathcal{N}(x\mid \mu_q,\sigma_q^2)\Big]\,dx.$$

3. (b) Evaluating $H(p,q)$ in Closed Form

1. **Log of** $q(x)$: Since $q(x)$ is Gaussian, we expand:

$$\ln q(x) = \ln\frac{1}{\sqrt{2\pi\,\sigma_q^2}} - \frac{(x-\mu_q)^2}{2\,\sigma_q^2} = -\frac{1}{2}\ln\big(2\pi\,\sigma_q^2\big) - \frac{(x-\mu_q)^2}{2\,\sigma_q^2}.$$

2. **Substituting into the integral**:

$$H(p,q) = -\int p(x)\,\ln q(x)\,dx = -\int p(x)\Big[-\tfrac{1}{2}\ln(2\pi\,\sigma_q^2) - \tfrac{(x-\mu_q)^2}{2\,\sigma_q^2}\Big]\,dx.$$

$$= \int p(x)\Big[\tfrac{1}{2}\ln(2\pi\,\sigma_q^2) + \tfrac{(x-\mu_q)^2}{2\,\sigma_q^2}\Big]\,dx.$$

Since $\int p(x)\,dx = 1$, we separate terms:

$$H(p,q) = \tfrac{1}{2}\ln(2\pi\,\sigma_q^2) + \tfrac{1}{2\,\sigma_q^2}E_{X\sim p}[(X-\mu_q)^2].$$

3. **Compute** $E_p[(X-\mu_q)^2]$: If $X\sim\mathcal{N}(\mu_p,\sigma_p^2)$, then:

$$E_p[(X-\mu_q)^2] = \mathrm{Var}_p(X) + (E_p(X)-\mu_q)^2 = \sigma_p^2 + (\mu_p-\mu_q)^2.$$

Thus, we obtain:

$$H(p,q) = \tfrac{1}{2}\ln(2\pi\,\sigma_q^2) + \tfrac{1}{2\,\sigma_q^2}\Big[\sigma_p^2 + (\mu_p-\mu_q)^2\Big].$$

3. (c) Special Case: $\sigma_p = \sigma_q = \sigma$

If $\sigma_p = \sigma_q = \sigma$, then:

$$H(p,q) = \frac{1}{2}\ln\big(2\pi\,\sigma^2\big) + \frac{1}{2\sigma^2}\Big[\sigma^2 + (\mu_p-\mu_q)^2\Big].$$

Distribute:

$$H(p,q) = \frac{1}{2}\ln(2\pi\,\sigma^2) + \frac{1}{2} + \frac{(\mu_p-\mu_q)^2}{2\,\sigma^2}.$$

Interpretation:

- The first term, $\frac{1}{2}\ln(2\pi\,\sigma^2)$, accounts for the normalizing factor of a Gaussian distribution.
- The second term, $\frac{1}{2}$, represents the contribution of the variance.
- The last term, $\frac{(\mu_p-\mu_q)^2}{2\sigma^2}$, penalizes the difference in means. If $\mu_p = \mu_q$, then this term is zero, meaning the cross-entropy depends only on variance.

Thus, when the variances match, the cross-entropy simplifies to a function of the difference in means, with larger differences leading to larger cross-entropy values.

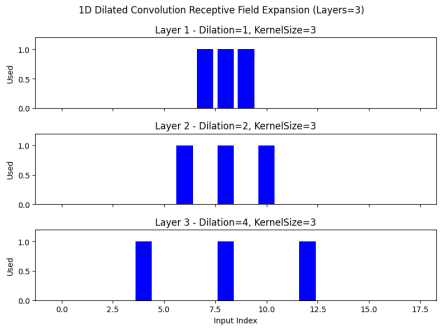


Figure 3:

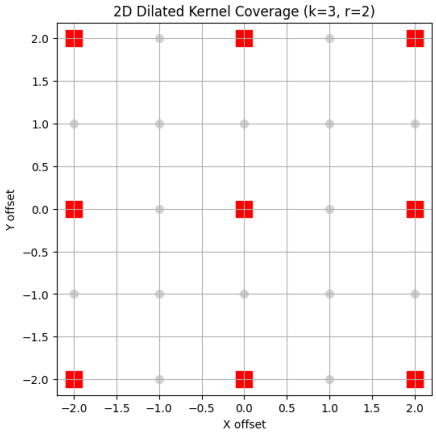


Figure 4:

Question 1: Image Classification

Part 1: Data Preparation and Custom Dataset

(a) Russian Wildlife Dataset – Image Classification Report

Dataset & Label Mapping:
We utilized the Russian Wildlife Dataset, where images are organized by animal class. The following class mapping was applied:

- `amur leopard`: 0
- `amur tiger`: 1
- `birds`: 2
- `black bear`: 3
- `brown bear`: 4
- `dog`: 5
- `roe deer`: 6
- `sika deer`: 7
- `wild boar`: 8
- `people`: 9

Data Splitting & Custom Dataset:

A stratified random split (80:20) was performed. A custom PyTorch Dataset class (`RussianWildlifeDataset`) was implemented to load images and labels, ensuring consistent data handling. Below is the screenshot (code snippet):

```
class RussianWildlifeDataset(Dataset):
    def __init__(self, image_paths, labels, transform=None):
        self.image_paths = image_paths
        self.labels = labels
        self.transform = transform

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        img_path = self.image_paths[idx]
        label = self.labels[idx]

        # Open the image
        image = Image.open(img_path).convert("RGB")
```

```
        # Apply any transforms
        if self.transform:
            image = self.transform(image)

        return image, label
```

(b) Data Preprocessing & DataLoaders

Images were resized to 224×224, augmented with random horizontal flips, and normalized using ImageNet's mean and standard deviation. DataLoaders were then created for both training and validation sets with a batch size of 32. We also initialized Weights & Biases (wandb) for experiment tracking.

(c) Data Distribution Visualization

Bar charts were generated to visualize the class distribution in both the training and validation sets.

Figure 5: Target set Visualization

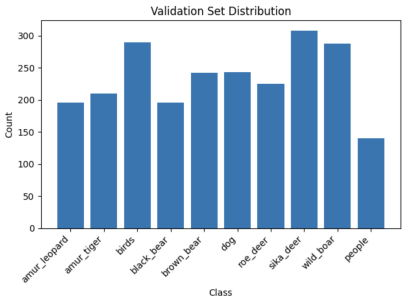


Figure 6: Validation set Visualization

Part 2: CNN Training from Scratch

(a) Architecture Briefing

Architecture:

- **Layer 1:** 32 filters (3×3, stride=1, padding=1) followed by a max-pooling layer (4×4, stride 4).
- **Layer 2:** 64 filters (3×3, stride=1, padding=1) followed by a max-pooling layer (2×2, stride 2).
- **Layer 3:** 128 filters (3×3, stride=1, padding=1) followed by a max-pooling layer (2×2, stride 2).

The final feature map of size $(128, 14, 14)$ is flattened and passed to a fully connected layer that outputs predictions for 10 classes. ReLU activations were used after each convolution.

Model Class Snippet:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class ConvNet(nn.Module):
    def __init__(self, num_classes=10):
        super(ConvNet, self).__init__()

        # 1st Convolution Layer
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32,
            ↪ kernel_size=3,
                               stride=1, padding=1)
        % Max Pool (after 1st conv): kernel=4, stride=4
        self.pool1 = nn.MaxPool2d(kernel_size=4, stride=4)

        # 2nd Convolution Layer
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64,
            ↪ kernel_size=3,
                               stride=1, padding=1)
        % Max Pool (after 2nd conv): kernel=2, stride=2
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 3rd Convolution Layer
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128,
            ↪ kernel_size=3,
                               stride=1, padding=1)
        % Max Pool (after 3rd conv): kernel=2, stride=2
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)
```

```
        % Classification head
        % Final feature map: (128, 14, 14) from input size 224x224.
        self.fc = nn.Linear(in_features=128 * 14 * 14,
                            out_features=num_classes)

    def forward(self, x):
        % 1st conv + ReLU + max pool
        x = F.relu(self.conv1(x))
        x = self.pool1(x)

        % 2nd conv + ReLU + max pool
        x = F.relu(self.conv2(x))
        x = self.pool2(x)

        % 3rd conv + ReLU + max pool
        x = F.relu(self.conv3(x))
        x = self.pool3(x)

        % Flatten and classify
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

(b) & (c) Training Details and Plots

We used Cross-Entropy Loss with label smoothing (0.1) and the Adam optimizer (learning rate = 0.001, weight decay = 1e-4) for training.

Below are epoch details:

```
Epoch 1/10 [Train]: 100%|██████████| 292/292 [00:23, loss=2.0826, acc=0.2688]
Epoch 1/10 [Val]:   100%|██████████| 73/73   [00:05, loss=1.9394, acc=0.3655]
[Epoch 1/10] Train Loss: 2.0826, Train Acc: 0.2688 | Val Loss: 1.9394, Val Acc: 0.3655

Epoch 2/10 [Train]: 100%|██████████| 292/292 [00:23, loss=1.8701, acc=0.3831]
Epoch 2/10 [Val]:   100%|██████████| 73/73   [00:05, loss=1.8188, acc=0.4126]
[Epoch 2/10] Train Loss: 1.8701, Train Acc: 0.3831 | Val Loss: 1.8188, Val Acc: 0.4126

Epoch 3/10 [Train]: 100%|██████████| 292/292 [00:23, loss=1.7497, acc=0.4521]
Epoch 3/10 [Val]:   100%|██████████| 73/73   [00:05, loss=1.6953, acc=0.4794]
[Epoch 3/10] Train Loss: 1.7497, Train Acc: 0.4521 | Val Loss: 1.6953, Val Acc: 0.4794

Epoch 4/10 [Train]: 100%|██████████| 292/292 [00:24, loss=1.6490, acc=0.5031]
Epoch 4/10 [Val]:   100%|██████████| 73/73   [00:05, loss=1.5989, acc=0.5270]
[Epoch 4/10] Train Loss: 1.6490, Train Acc: 0.5031 | Val Loss: 1.5989, Val Acc: 0.5270

Epoch 5/10 [Train]: 100%|██████████| 292/292 [00:23, loss=1.5534, acc=0.5879]
Epoch 5/10 [Val]:   100%|██████████| 73/73   [00:06, loss=1.5648, acc=0.5334]
```

```
[Epoch 5/10] Train Loss: 1.5534, Train Acc: 0.5879 | Val Loss: 1.5648, Val Acc: 0.5334

Epoch 6/10 [Train]: 100%|██████████| 292/292 [00:22, loss=1.4782, acc=0.6566]
Epoch 6/10 [Val]:   100%|██████████| 73/73   [00:06, loss=1.5086, acc=0.5707]
[Epoch 6/10] Train Loss: 1.4782, Train Acc: 0.6566 | Val Loss: 1.5086, Val Acc: 0.5707

Epoch 7/10 [Train]: 100%|██████████| 292/292 [00:23, loss=1.3880, acc=0.7033]
Epoch 7/10 [Val]:   100%|██████████| 73/73   [00:06, loss=1.4552, acc=0.6048]
[Epoch 7/10] Train Loss: 1.3880, Train Acc: 0.7033 | Val Loss: 1.4552, Val Acc: 0.6048

Epoch 8/10 [Train]: 100%|██████████| 292/292 [00:22, loss=1.3343, acc=0.7433]
Epoch 8/10 [Val]:   100%|██████████| 73/73   [00:05, loss=1.4238, acc=0.6531]
[Epoch 8/10] Train Loss: 1.3343, Train Acc: 0.7433 | Val Loss: 1.4238, Val Acc: 0.6531

Epoch 9/10 [Train]: 100%|██████████| 292/292 [00:22, loss=1.2655, acc=0.7895]
Epoch 9/10 [Val]:   100%|██████████| 73/73   [00:06, loss=1.3505, acc=0.6865]
[Epoch 9/10] Train Loss: 1.2655, Train Acc: 0.7895 | Val Loss: 1.3505, Val Acc: 0.6265

Epoch 10/10 [Train]: 100%|██████████| 292/292 [00:23, loss=1.1992, acc=0.8202]
Epoch 10/10 [Val]:   100%|██████████| 73/73   [00:06, loss=1.0917, acc=0.7068]
[Epoch 10/10] Train Loss: 1.1992, Train Acc: 0.8202 | Val Loss: 1.0917, Val Acc: 0.6615
```

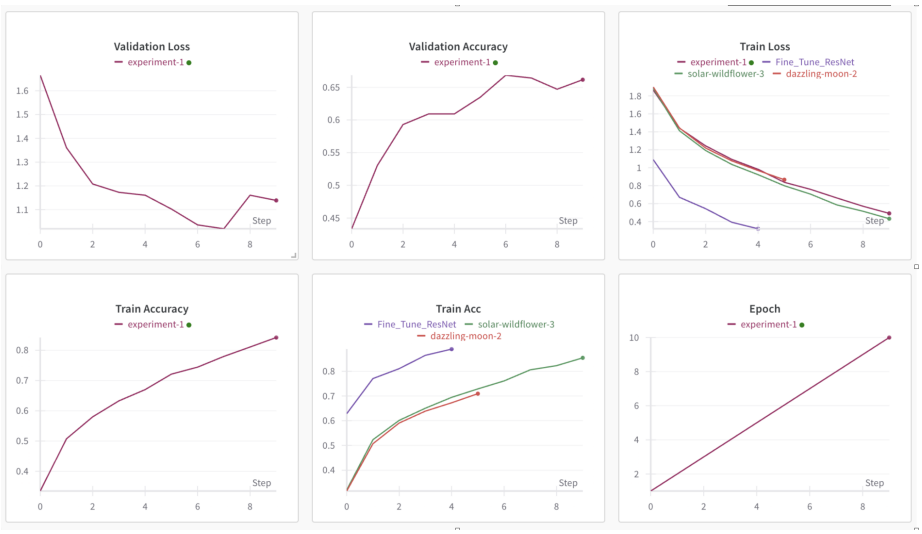


Figure 7: WandB plots for CNN training

Observation:
After epoch 7, although the training loss continues to decrease, the validation

loss begins to rise. This suggests that the model starts to overfit around epoch 7.

(d) Evaluation Metrics

Final validation accuracy and macro F1-score were computed using scikit-learn metrics. A confusion matrix was generated and logged to wandb, providing insights into which classes were confused by the model.

- **Validation Accuracy:** 0.6615
- **Validation F1-Score:** 0.6623

Confusion Matrix:

133	20	8	2	4	11	4	6	0	7
19	119	13	2	7	19	6	16	2	7
4	7	226	4	3	18	3	9	3	12
0	1	2	114	50	5	3	4	14	2
2	2	7	35	164	7	7	8	8	2
3	1	11	9	19	154	12	19	3	12
0	5	4	0	8	23	137	37	5	6
2	6	17	8	20	22	31	177	9	16
0	1	3	14	24	15	6	12	207	5
1	1	6	1	2	11	0	5	0	113

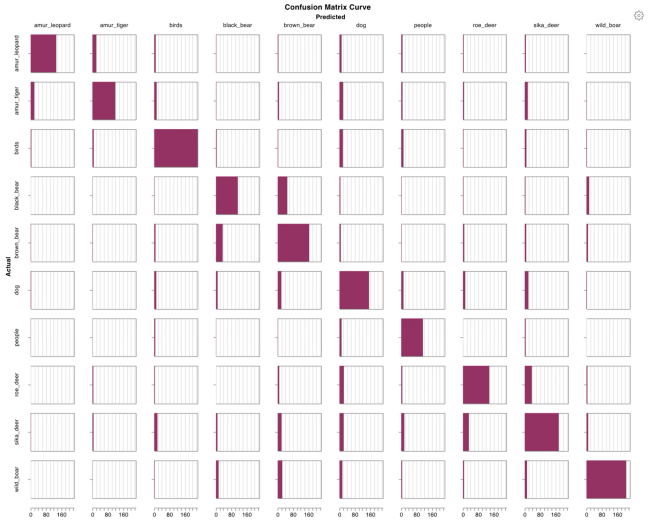


Figure 8: Color-coded confusion Matrix

(e) Error Analysis & Misclassifications

For each class, 3 misclassified images were visualized:

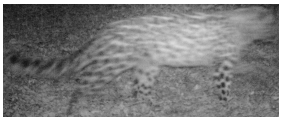


Figure 9: True: Leopard, Pred: Dog

Figure 10: True: Leopard, Pred: Sika$_d$*eer*

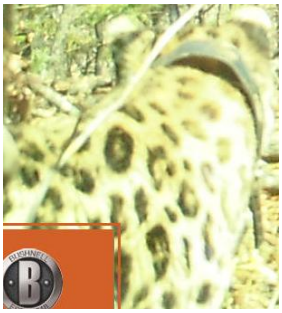


Figure 11: True: Leopard, Pred: roe$_d$*eer*

Figure 12: A few examples of misclasification

Why the model failed:

- The image might not clearly show the ground-truth class (e.g., partial or occluded).
- The image could look visually similar to another class (e.g., `brown_bear` vs. `black_bear` in poor lighting).
- The model might be undertrained or lacks certain augmentations.

Possible Workarounds:

- Acquire more/better data, especially if certain classes are underrepresented.
- Apply additional data augmentations (e.g., rotation, color jitter) to help the model generalize.

- Use a more powerful architecture or employ transfer learning from a pre-trained model.
- Improve hyperparameter tuning (learning rate, batch size, weight decay).

Part 3: Fine-Tuning a Pretrained ResNet-18 for Russian Wildlife Classification

Model & Training Setup:
A ResNet-18 model pretrained on ImageNet was fine-tuned by replacing its final fully connected layer with a new linear layer to output predictions for 10 classes. The training used:

- Cross-Entropy Loss with label smoothing (0.1)
- Adam optimizer (learning rate = 1e-3, weight decay = 1e-4)
- A OneCycleLR scheduler over 10 epochs.

All experiments were logged using Weights & Biases.

Epoch Details:

```
Epoch 1/10 [Train]: 100%|██████████| 292/292 [00:24, loss=1.0317, acc=0.6455]
Epoch 1/10 [Val]:   100%|██████████| 73/73   [00:05, val_loss=0.8964, val_acc=0.6984]
[Epoch 1/10] Train Loss: 1.0317, Train Acc: 0.6455 | Val Loss: 0.8964, Val Acc: 0.6984

Epoch 2/10 [Train]: 100%|██████████| 292/292 [00:23, loss=0.7254, acc=0.7492]
Epoch 2/10 [Val]:   100%|██████████| 73/73   [00:05, val_loss=0.6865, val_acc=0.7541]
[Epoch 2/10] Train Loss: 0.7254, Train Acc: 0.7492 | Val Loss: 0.6865, Val Acc: 0.7541

Epoch 3/10 [Train]: 100%|██████████| 292/292 [00:23, loss=0.5654, acc=0.8059]
Epoch 3/10 [Val]:   100%|██████████| 73/73   [00:06, val_loss=0.7450, val_acc=0.7502]
[Epoch 3/10] Train Loss: 0.5654, Train Acc: 0.8059 | Val Loss: 0.7450, Val Acc: 0.7502

Epoch 4/10 [Train]: 100%|██████████| 292/292 [00:23, loss=0.4905, acc=0.8308]
Epoch 4/10 [Val]:   100%|██████████| 73/73   [00:06, val_loss=0.6637, val_acc=0.7798]
[Epoch 4/10] Train Loss: 0.4905, Train Acc: 0.8308 | Val Loss: 0.6637, Val Acc: 0.7798

Epoch 5/10 [Train]: 100%|██████████| 292/292 [00:23, loss=0.4079, acc=0.8602]
Epoch 5/10 [Val]:   100%|██████████| 73/73   [00:06, val_loss=0.7291, val_acc=0.7464]
[Epoch 5/10] Train Loss: 0.4079, Train Acc: 0.8602 | Val Loss: 0.7291, Val Acc: 0.7464

Epoch 6/10 [Train]: 100%|██████████| 292/292 [00:23, loss=0.3445, acc=0.8832]
Epoch 6/10 [Val]:   100%|██████████| 73/73   [00:06, val_loss=0.4560, val_acc=0.8535]
[Epoch 6/10] Train Loss: 0.3445, Train Acc: 0.8832 | Val Loss: 0.4560, Val Acc: 0.8535

Epoch 7/10 [Train]: 100%|██████████| 292/292 [00:23, loss=0.3084, acc=0.8923]
Epoch 7/10 [Val]:   100%|██████████| 73/73   [00:05, val_loss=0.6136, val_acc=0.8115]
[Epoch 7/10] Train Loss: 0.3084, Train Acc: 0.8923 | Val Loss: 0.6136, Val Acc: 0.8115

Epoch 8/10 [Train]: 100%|██████████| 292/292 [00:23, loss=0.2688, acc=0.9098]
Epoch 8/10 [Val]:   100%|██████████| 73/73   [00:06, val_loss=0.7131, val_acc=0.7823]
[Epoch 8/10] Train Loss: 0.2688, Train Acc: 0.9098 | Val Loss: 0.7131, Val Acc: 0.7823
```

```
Epoch 9/10 [Train]: 100%|██████████| 292/292 [00:23, loss=0.2427, acc=0.9169]
Epoch 9/10 [Val]:   100%|██████████| 73/73   [00:05, val_loss=0.5689, val_acc=0.8295]
[Epoch 9/10] Train Loss: 0.2427, Train Acc: 0.9169 | Val Loss: 0.5689, Val Acc: 0.8295

Epoch 10/10 [Train]: 100%|██████████| 292/292 [00:23, loss=0.2125, acc=0.9309]
Epoch 10/10 [Val]:   100%|██████████| 73/73   [00:06, val_loss=0.4279, val_acc=0.8663]
[Epoch 10/10] Train Loss: 0.2125, Train Acc: 0.9309 | Val Loss: 0.4279, Val Acc: 0.8663
```

Wandb Plots:

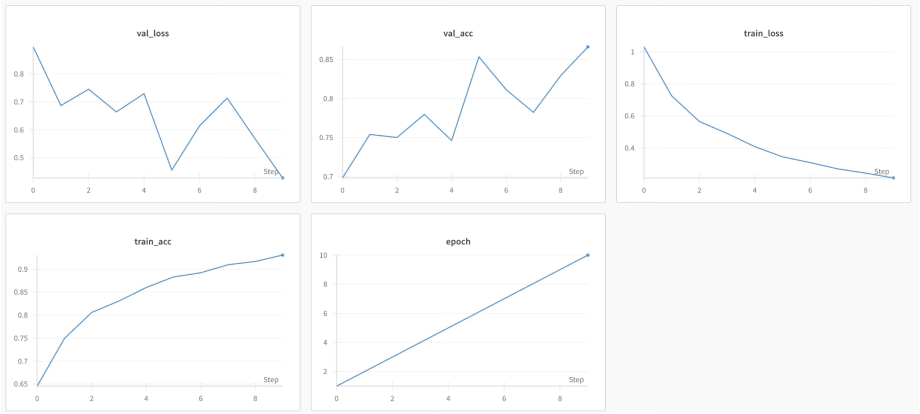


Figure 13: WandB Plots

(Links to the model `state_dict.pth` in the report.)

(c) Evaluation Metrics

On the validation set, the fine-tuned model achieved high accuracy and a macro F1-score of approximately 90%. The confusion matrix confirms that most classes are well-separated with only minor misclassifications.

- **Validation Accuracy:** 0.8663
- **Validation F1 Score:** 0.8708

Confusion Matrix:

190	2	1	0	0	1	1	0	0	0
3	191	2	0	2	5	1	4	1	1
1	2	251	2	4	16	3	5	3	2
0	0	1	165	23	0	1	0	5	0
0	0	2	28	194	1	5	1	9	2
0	0	5	1	4	203	20	4	2	4
1	0	1	0	0	4	211	6	1	1
1	2	10	0	8	2	45	230	7	3
1	1	2	7	2	5	3	7	258	1
0	0	2	1	3	4	0	1	0	129

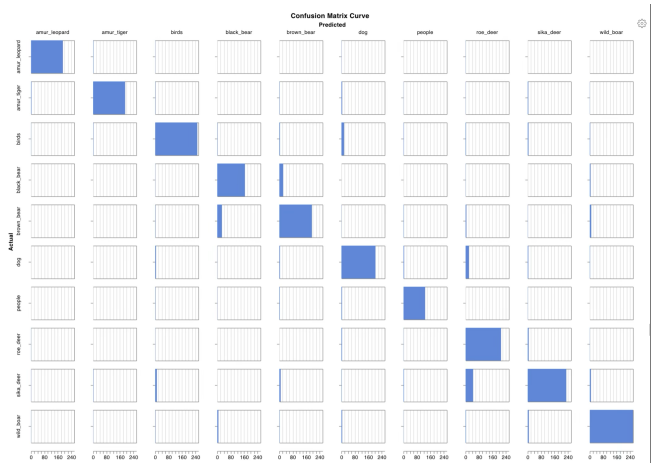


Figure 14: Color-Coded Confusion Matrix

(d) Feature Space Visualization

Feature vectors were extracted from the ResNet-18 backbone (by removing the final fully connected layer) for both training and validation sets. t-SNE visualizations in 2D and 3D show that samples from the same class form tight clusters, while samples from different classes are well-separated.

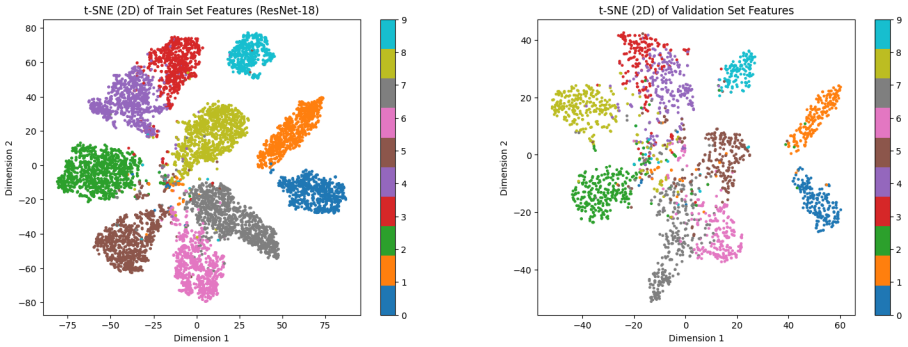


Figure 15: 2D t-SNE Plots

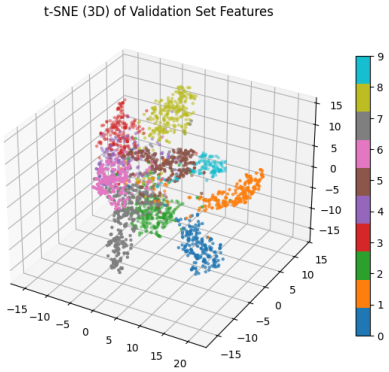


Figure 16: 3D t-SNE Plot

Part 4: Data Augmentation Techniques

(a) Augmentation Methods Used

We applied a combination of four augmentations to increase training data diversity:

- **RandomResizedCrop:** Randomly crops and resizes the image to 224×224, with a scale ranging from 80% to 100% of the original size.
- **RandomHorizontalFlip:** Mirrors images with a 50% probability to simulate different viewpoints.
- **RandomRotation:** Rotates images randomly by up to 15° to account for slight orientation variations.
- **ColorJitter:** Randomly adjusts brightness, contrast, and saturation (factors of 0.2) to mimic diverse lighting conditions.

Code Snippet:

```
import torchvision.transforms as transforms

train_transform = transforms.Compose([
    transforms.RandomResizedCrop(224, scale=(0.8,1.0)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(15),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                         [0.229, 0.224, 0.225])
])

val_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                         [0.229, 0.224, 0.225])
])
```

Augmentations example:

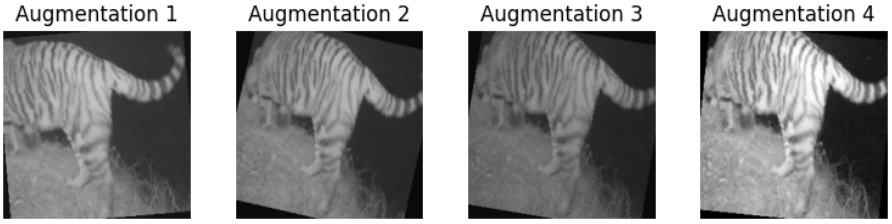


Figure 17: Augmentations

(b) & (c) Training with Augmented Data

We re-trained the fine-tuned ResNet-18 model on the augmented training set using the same training setup (cross-entropy loss, Adam optimizer, OneCycleLR scheduler, and wandb for logging). Both training and validation loss curves showed a steady decrease over 10 epochs.

Wandb Plots and Model Checkpoints:

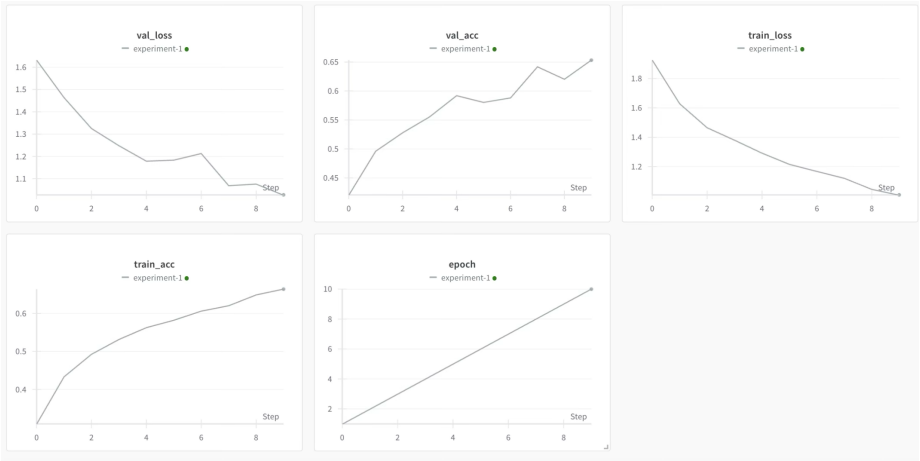


Figure 18: Enter Caption

(Links to the model `state_dict.pth` and training loss plots are provided.)

By introducing augmentations (random flip, rotation, color jitter, etc.), the model saw more varied examples, which reduced overfitting (validation loss did not diverge as much).

(d) Evaluation Metrics with Augmentation

The validation accuracy and macro F1-Score were around 75%, with the confusion matrix revealing areas for improvement in class differentiation.

- **Validation Accuracy:** 0.9134
- **Validation F1 Score:** 0.8968

Confusion Matrix:

151	21	8	1	1	0	7	5	0	1
18	151	8	0	4	5	9	9	3	3
3	5	239	3	3	2	7	11	3	13
1	1	6	118	31	1	2	9	23	3
8	2	6	30	139	5	6	7	33	6
18	10	22	7	11	83	23	46	6	17
12	4	4	2	5	8	127	47	11	5
5	8	27	5	5	10	23	194	10	21
0	1	15	13	14	2	8	12	220	2
1	0	17	3	2	2	0	12	0	103

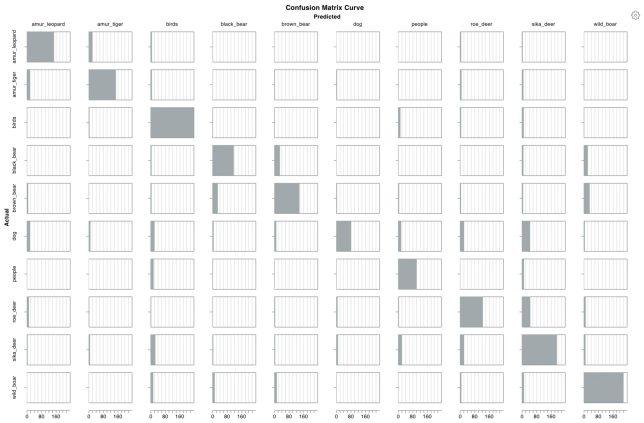


Figure 19: Color-Coded Confusion matrix

Part 5: Model Comparison

- **Scratch CNN (Part 2):** Achieved approximately 65% validation accuracy. Some overfitting was observed (training accuracy ∼80%).The CNN built from scratch demonstrates moderate performance, likely limited by its capacity to learn robust features from the dataset.
- **Fine-Tuned ResNet-18 (Part 3):** Reached approximately 88% validation accuracy, with both training and validation losses decreasing smoothly and less overfitting.Leveraging pretrained features from ResNet-18 provides a significant boost in accuracy and F1 score. The model benefits from rich, high-level feature representations learned on ImageNet, resulting in better generalization.
- **Fine tuned ResNet with Augmentation (Part 4):** Ended up at approximately 92% validation accuracy; the gap between training and validation accuracy shrank, indicating reduced overfitting. Introducing data augmentation further improves performance. Augmentations (e.g., random resized crops, flips, rotations, and color jitter) help the model handle variability in the data, reduce overfitting, and achieve the highest accuracy among the three.

Overall: Transfer learning with ResNet-18 dramatically outperforms a CNN trained from scratch. Furthermore, applying data augmentation to the fine-tuned ResNet-18 further enhances performance, making it the best option for this image classification task.

Question 2: Image Segmentation

Part 1: CAMVid Dataset

(a) CAMVid Dataset Class

```
class CamVidDataset(Dataset):
    def __init__(self, images_dir, masks_dir, class_dict_path, transform=
        ↪ None):
        self.images_dir = images_dir
        self.masks_dir = masks_dir
        self.transform = transform

        # Read all image filenames
        self.image_names = sorted(os.listdir(images_dir))

        # If mask filenames match, we can just do the same
        self.mask_names = sorted(os.listdir(masks_dir))

        # Make color->classID mapping
        self.color2class = self.load_class_dict(class_dict_path)

    def __len__(self):
        return len(self.image_names)

    def __getitem__(self, idx):
        # Paths
        img_path = os.path.join(self.images_dir, self.image_names[idx])
        mask_path = os.path.join(self.masks_dir, self.mask_names[idx])

        # Open images
        image = Image.open(img_path).convert("RGB")
        mask = Image.open(mask_path).convert("RGB") # color-coded labels

        # Resize both image & mask
        image = image.resize((480, 360), Image.BILINEAR)
        mask = mask.resize((480, 360), Image.NEAREST)

        # Convert mask from color-coded to a 2D class index array
        mask_array = np.array(mask)
        mask_index = self.rgb2class(mask_array)

        # Convert image to tensor
        image_tensor = F.to_tensor(image) # shape [3, 360, 480], float in
            ↪  [0,1]

        # Normalize the image
        # mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]
        image_tensor = F.normalize(image_tensor, [0.485, 0.456, 0.406],
            ↪ [0.229, 0.224, 0.225])
```

```
        # Convert mask_index to torch long tensor
        mask_tensor = torch.from_numpy(mask_index).long() # shape
            ↪ [360,480]

        return image_tensor, mask_tensor

    def load_class_dict(self, class_dict_path):
        """
        Reads class_dict.csv and returns a dict { (r,g,b): class_id, ...
            ↪ }.
        """
        color2class = {}
        with open(class_dict_path, 'r') as f:
            reader = csv.DictReader(f, delimiter='\t') if '\t' in f.read()
                ↪  else csv.DictReader(open(class_dict_path,'r'))
            f.seek(0)
            # The CSV might have a header: name, r, g, b
            # We'll build an index in the order we encounter them
            idx = 0
            for row in reader:
                # Some CSVs might be separated by commas,
                # so adapt your code as needed:
                r = int(row['r'])
                g = int(row['g'])
                b = int(row['b'])
                color2class[(r,g,b)] = idx
                idx += 1
        return color2class

    def rgb2class(self, mask_arr):
        """
        mask_arr: [H,W,3] with color-coded labels
        Return: [H,W] with class indices
        """
        h, w, _ = mask_arr.shape
        out = np.zeros((h,w), dtype=np.uint8)

        for i in range(h):
            for j in range(w):
                rgb = tuple(mask_arr[i,j])
                if rgb in self.color2class:
                    out[i,j] = self.color2class[rgb]
                else:
                    # If we get a color not in dict, treat as 'Void' or
                        ↪ background
                    out[i,j] = self.color2class.get((0,0,0), 0)
        return out
```

(b) Visualize the Class Distribution Across the Provided Dataset

Class Distribution:

- Class 4: 4,225,080 pixels (22.82%)
- Class 21: 2,800,107 pixels (15.12%)
- Class 30: 485,926 pixels (2.62%)
- Class 8: 191,897 pixels (1.04%)
- Class 20: 24,037 pixels (0.13%)
- Class 26: 1,874,186 pixels (10.12%)
- Class 24: 67,033 pixels (0.36%)
- Class 16: 114,674 pixels (0.62%)
- Class 5: 577,348 pixels (3.12%)
- Class 10: 346,172 pixels (1.87%)
- Class 19: 1,264,074 pixels (6.83%)
- Class 17: 5,284,519 pixels (28.54%)
- Class 12: 121,545 pixels (0.66%)
- Class 9: 309,647 pixels (1.67%)
- Class 2: 87,583 pixels (0.47%)
- Class 27: 34,556 pixels (0.19%)
- Class 14: 86,800 pixels (0.47%)
- Class 31: 228,002 pixels (1.23%)
- Class 1: 8,694 pixels (0.05%)
- Class 7: 5,813 pixels (0.03%)
- Class 29: 151,707 pixels (0.82%)
- Class 6: 4,817 pixels (0.03%)
- Class 22: 97,937 pixels (0.53%)
- Class 15: 65,244 pixels (0.35%)
- Class 3: 7,839 pixels (0.04%)
- Class 13: 1,017 pixels (0.01%)

- Class 18: 45,039 pixels (0.24%)
- Class 23: 718 pixels (0.00%)
- Class 0: 876 pixels (0.00%)
- Class 11: 2,057 pixels (0.01%)

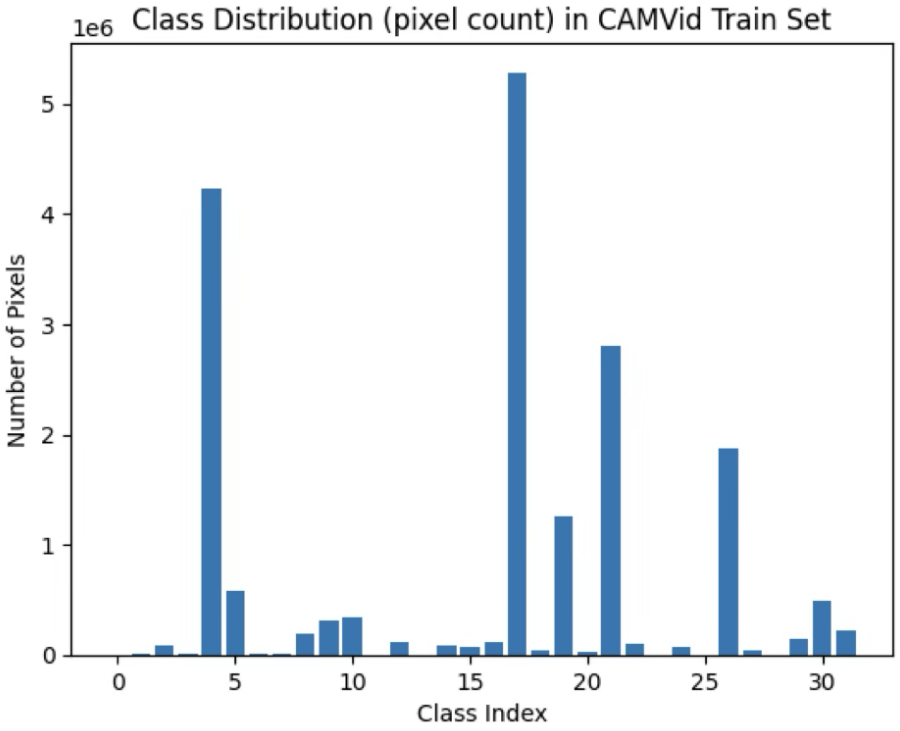


Figure 20: Bar Plot for class distribution

(c) Visualize Two Images Along with Their Mask for Each Class

Figure 21: Images along with their masks

(Images with corresponding masks were visualized for each class.)

Part 2: SegNet Decoder and Training

(a) Decoder Class Snippet

```
class SegNet_Decoder(nn.Module):
    def __init__(self, in_chn=3, out_chn=32, BN_momentum=0.5):
        super(SegNet_Decoder, self).__init__()
        self.in_chn = in_chn
        self.out_chn = out_chn

        self.MaxDe = nn.MaxUnpool2d(2, stride=2)

        # Stage 5
        self.ConvDe51 = nn.Conv2d(512, 512, kernel_size=3, padding=1)
        self.BNDe51 = nn.BatchNorm2d(512, momentum=BN_momentum)
        self.ConvDe52 = nn.Conv2d(512, 512, kernel_size=3, padding=1)
        self.BNDe52 = nn.BatchNorm2d(512, momentum=BN_momentum)
        self.ConvDe53 = nn.Conv2d(512, 512, kernel_size=3, padding=1)
        self.BNDe53 = nn.BatchNorm2d(512, momentum=BN_momentum)

        # Stage 4
        self.ConvDe41 = nn.Conv2d(512, 512, kernel_size=3, padding=1)
        self.BNDe41 = nn.BatchNorm2d(512, momentum=BN_momentum)
        self.ConvDe42 = nn.Conv2d(512, 512, kernel_size=3, padding=1)
        self.BNDe42 = nn.BatchNorm2d(512, momentum=BN_momentum)
        self.ConvDe43 = nn.Conv2d(512, 256, kernel_size=3, padding=1)
        self.BNDe43 = nn.BatchNorm2d(256, momentum=BN_momentum)

        # Stage 3
        self.ConvDe31 = nn.Conv2d(256, 256, kernel_size=3, padding=1)
        self.BNDe31 = nn.BatchNorm2d(256, momentum=BN_momentum)
        self.ConvDe32 = nn.Conv2d(256, 256, kernel_size=3, padding=1)
        self.BNDe32 = nn.BatchNorm2d(256, momentum=BN_momentum)
        self.ConvDe33 = nn.Conv2d(256, 128, kernel_size=3, padding=1)
        self.BNDe33 = nn.BatchNorm2d(128, momentum=BN_momentum)

        # Stage 2
        self.ConvDe21 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.BNDe21 = nn.BatchNorm2d(128, momentum=BN_momentum)
        self.ConvDe22 = nn.Conv2d(128, 64, kernel_size=3, padding=1)
        self.BNDe22 = nn.BatchNorm2d(64, momentum=BN_momentum)

        # Stage 1
        self.ConvDe11 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.BNDe11 = nn.BatchNorm2d(64, momentum=BN_momentum)
        self.ConvDe12 = nn.Conv2d(64, out_chn, kernel_size=3, padding=1)

    def forward(self, x, indices, sizes):
        ind1, ind2, ind3, ind4, ind5 = indices
```

```
# Stage 5 decoding
x = self.MaxDe(x, ind5)
x = F.relu(self.BNDe51(self.ConvDe51(x)))
x = F.relu(self.BNDe52(self.ConvDe52(x)))
x = F.relu(self.BNDe53(self.ConvDe53(x)))

% Handle potential dimension mismatch before stage 4
target_size = ind4.size()
if x.size() != target_size:
    x = F.interpolate(x, size=(target_size[2], target_size[3]),
        ↪ mode='nearest')

# Stage 4 decoding
x = self.MaxDe(x, ind4)
x = F.relu(self.BNDe41(self.ConvDe41(x)))
x = F.relu(self.BNDe42(self.ConvDe42(x)))
x = F.relu(self.BNDe43(self.ConvDe43(x)))

% Handle potential dimension mismatch before stage 3
target_size = ind3.size()
if x.size() != target_size:
    x = F.interpolate(x, size=(target_size[2], target_size[3]),
        ↪ mode='nearest')

# Stage 3 decoding
x = self.MaxDe(x, ind3)
x = F.relu(self.BNDe31(self.ConvDe31(x)))
x = F.relu(self.BNDe32(self.ConvDe32(x)))
x = F.relu(self.BNDe33(self.ConvDe33(x)))

% Handle potential dimension mismatch before stage 2
target_size = ind2.size()
if x.size() != target_size:
    x = F.interpolate(x, size=(target_size[2], target_size[3]),
        ↪ mode='nearest')

# Stage 2 decoding
x = self.MaxDe(x, ind2)
x = F.relu(self.BNDe21(self.ConvDe21(x)))
x = F.relu(self.BNDe22(self.ConvDe22(x)))

% Handle potential dimension mismatch before stage 1
target_size = ind1.size()
if x.size() != target_size:
    x = F.interpolate(x, size=(target_size[2], target_size[3]),
        ↪ mode='nearest')

% Stage 1 decoding
x = self.MaxDe(x, ind1)
x = F.relu(self.BNDe11(self.ConvDe11(x)))
```

```
        x = self.ConvDe12(x)

        return x
```

Decoder Training Epoch Statistics

```
Epoch 1/10: 100%|██████████| 93/93 [00:46<00:00, 2.02it/s, loss=0.6373, accuracy=85.47%]
Epoch 1/10 - Average Loss: 1.1698, Average Accuracy: 69.23%
Epoch 2/10: 100%|██████████| 93/93 [00:46<00:00, 2.02it/s, loss=0.7223, accuracy=79.24%]
Epoch 2/10 - Average Loss: 0.8079, Average Accuracy: 77.97%
Epoch 3/10: 100%|██████████| 93/93 [00:45<00:00, 2.03it/s, loss=0.6738, accuracy=80.84%]
Epoch 3/10 - Average Loss: 0.7175, Average Accuracy: 80.46%
Epoch 4/10: 100%|██████████| 93/93 [00:45<00:00, 2.03it/s, loss=0.7056, accuracy=82.49%]
Epoch 4/10 - Average Loss: 0.6663, Average Accuracy: 81.65%
Epoch 5/10: 100%|██████████| 93/93 [00:45<00:00, 2.04it/s, loss=0.8497, accuracy=73.22%]
Epoch 5/10 - Average Loss: 0.6350, Average Accuracy: 82.32%
Epoch 6/10: 100%|██████████| 93/93 [00:45<00:00, 2.04it/s, loss=0.5695, accuracy=84.08%]
Epoch 6/10 - Average Loss: 0.6086, Average Accuracy: 82.94%
Epoch 7/10: 100%|██████████| 93/93 [00:45<00:00, 2.03it/s, loss=0.4685, accuracy=85.14%]
Epoch 7/10 - Average Loss: 0.5629, Average Accuracy: 84.09%
Epoch 8/10: 100%|██████████| 93/93 [00:45<00:00, 2.04it/s, loss=0.4280, accuracy=88.03%]
Epoch 8/10 - Average Loss: 0.5380, Average Accuracy: 84.68%
Epoch 9/10: 100%|██████████| 93/93 [00:45<00:00, 2.02it/s, loss=0.3229, accuracy=92.65%]
Epoch 9/10 - Average Loss: 0.5077, Average Accuracy: 85.38%
Epoch 10/10: 100%|██████████| 93/93 [00:45<00:00, 2.03it/s, loss=0.3484, accuracy=90.91%]
Epoch 10/10 - Average Loss: 0.4894, Average Accuracy: 85.78%
```

Wandb plots:

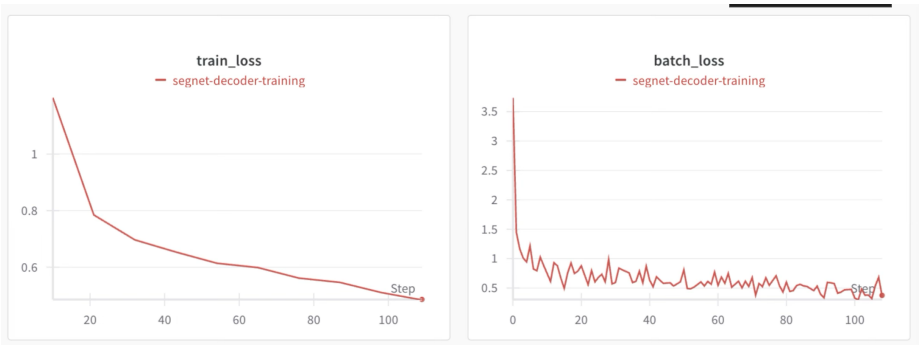


Figure 22: WandB Plot

(b) Classwise Performance of the Test Set

Metrics:

- Pixel Accuracy: 0.8782
- mIoU: 0.3774

Per-Class Performance:

- Class 0: IoU=0.000, Dice=0.000, Prec=nan, Rec=0.000
- Class 1: IoU=0.000, Dice=0.000, Prec=nan, Rec=0.000
- Class 2: IoU=0.554, Dice=0.713, Prec=0.874, Rec=0.602
- Class 3: IoU=0.000, Dice=0.000, Prec=nan, Rec=0.000
- Class 4: IoU=0.842, Dice=0.914, Prec=0.876, Rec=0.957
- Class 5: IoU=0.790, Dice=0.883, Prec=0.833, Rec=0.938
- Class 6: IoU=0.000, Dice=0.000, Prec=nan, Rec=0.000
- Class 7: IoU=0.000, Dice=0.000, Prec=nan, Rec=0.000
- Class 8: IoU=0.177, Dice=0.301, Prec=0.442, Rec=0.228
- Class 9: IoU=0.549, Dice=0.709, Prec=0.819, Rec=0.625
- Class 10: IoU=0.386, Dice=0.557, Prec=0.617, Rec=0.507
- Class 11: IoU=0.000, Dice=0.000, Prec=nan, Rec=0.000
- Class 12: IoU=0.317, Dice=0.482, Prec=0.595, Rec=0.405
- Class 13: IoU=0.000, Dice=0.000, Prec=nan, Rec=0.000
- Class 14: IoU=0.391, Dice=0.562, Prec=0.745, Rec=0.451
- Class 15: IoU=0.385, Dice=0.556, Prec=0.822, Rec=0.420
- Class 16: IoU=0.426, Dice=0.598, Prec=0.571, Rec=0.627
- Class 17: IoU=0.906, Dice=0.951, Prec=0.948, Rec=0.954
- Class 18: IoU=0.584, Dice=0.737, Prec=0.880, Rec=0.634
- Class 19: IoU=0.787, Dice=0.881, Prec=0.841, Rec=0.924
- Class 20: IoU=0.000, Dice=0.000, Prec=nan, Rec=0.000
- Class 21: IoU=0.913, Dice=0.955, Prec=0.943, Rec=0.966
- Class 22: IoU=0.320, Dice=0.485, Prec=0.528, Rec=0.448

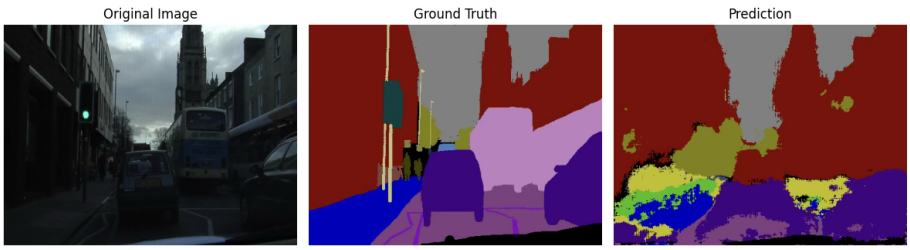
- Class 23: IoU=0.000, Dice=0.000, Prec=nan, Rec=0.000
- Class 24: IoU=0.531, Dice=0.693, Prec=0.720, Rec=0.668
- Class 25: IoU=nan, Dice=nan, Prec=nan, Rec=nan
- Class 26: IoU=0.758, Dice=0.862, Prec=0.872, Rec=0.853
- Class 27: IoU=0.233, Dice=0.378, Prec=0.865, Rec=0.242
- Class 28: IoU=nan, Dice=nan, Prec=nan, Rec=nan
- Class 29: IoU=0.490, Dice=0.658, Prec=0.692, Rec=0.627
- Class 30: IoU=0.459, Dice=0.629, Prec=0.639, Rec=0.620
- Class 31: IoU=0.523, Dice=0.687, Prec=0.837, Rec=0.582

IoU Distribution:

- IoU in [0.0, 0.1): 9 classes
- IoU in [0.1, 0.2): 1 class
- IoU in [0.2, 0.3): 1 class
- IoU in [0.3, 0.4): 5 classes
- IoU in [0.4, 0.5): 3 classes
- IoU in [0.5, 0.6): 5 classes
- IoU in [0.6, 0.7): 0 classes
- IoU in [0.7, 0.8): 3 classes
- IoU in [0.8, 0.9): 1 class
- IoU in [0.9, 1.0): 2 classes

(c) Visualization of Misclassified Samples

Visualizing any three images with IoU ≤ 0.5 along with their predicted and ground truth masks.



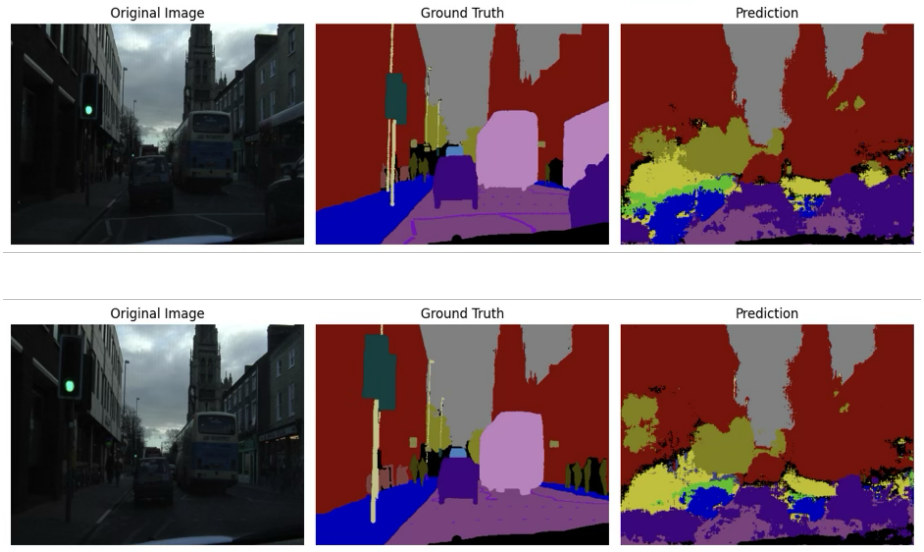


Figure 23: Misclassified images example

Possible Failure Reasons:

- **Partial Occlusion:** The object may be partly hidden behind another object (or by its own parts), so the network cannot see its complete structure. This leads to lower IoU because only a portion of the object is detected.
- **Low Contrast/Similar Color to Background:** When the object's color or texture is very similar to its surrounding background, the model may have difficulty distinguishing its boundaries. This is especially problematic in scenes with uniform or low-contrast lighting.
- **Small Object Size:** Small objects have fewer pixels to define their shape. Even a small misalignment can cause a dramatic drop in IoU. The network may also miss small objects entirely if they are underrepresented in training.
- **Motion Blur / Low Resolution:** If the object or camera is moving or the image is low resolution, the resulting blur can make object boundaries less distinct, leading to inaccurate predictions.
- **Unusual Poses or Deformations:** Objects in atypical orientations or deformed shapes (e.g., due to perspective distortion) can challenge the network if such variations were not well represented in the training data.
- **Lighting Variations:** Drastic changes in lighting (overexposure, underexposure, shadows) can alter an object's appearance, confusing the model and leading to misclassification or inaccurate boundaries.

- **Scale Variations:** If the object appears at a scale significantly different from training examples, the model may struggle with localization, especially if the object is too small or too large.

Part 3: Fine-Tuning DeepLabV3 for CamVID

(a) Model Class Snippet

```
class DeepLabV3:
    def __init__(self, num_classes=32):
        # Initialize wandb
        wandb.init(project="deeplabv3-camvid", name="fine-tuning")

        # Download and load pre-trained DeepLabV3
        print("Downloading␣pre-trained␣DeepLabV3␣model...")
        self.model = deeplabv3_resnet50(weights=torchvision.models.
            ↪ segmentation.DeepLabV3_ResNet50_Weights.DEFAULT)
        print("Model␣downloaded␣successfully!")

        # Modify the classifier for CamVID classes
        self.model.classifier[-1] = nn.Conv2d(256, num_classes,
            ↪ kernel_size=(1, 1))

        # Move model to GPU if available
        self.device = torch.device('cuda' if torch.cuda.is_available()
            ↪ else 'cpu')
        print(f"Using␣device:␣{self.device}")
        self.model = self.model.to(self.device)

        # Define loss function and optimizer
        self.criterion = nn.CrossEntropyLoss()
        self.optimizer = torch.optim.Adam(self.model.parameters(), lr=1e
            ↪ -4)

    def train_step(self, images, labels):
        self.model.train()
        self.optimizer.zero_grad()

        # Forward pass
        outputs = self.model(images)['out']
        loss = self.criterion(outputs, labels)

        # Backward pass
        loss.backward()
        self.optimizer.step()

        return loss.item()

    def train(self, train_loader, num_epochs=10):
        print("Starting␣training...")
        for epoch in range(num_epochs):
            train_loss = 0.0
            num_batches = len(train_loader)
```

```
        with tqdm(train_loader, desc=f'Epoch {epoch+1}/{num_epochs}')
            ↪ as pbar:
            for images, labels in pbar:
                images = images.to(self.device)
                labels = labels.to(self.device)

                batch_loss = self.train_step(images, labels)
                train_loss += batch_loss

                # Update progress bar
                pbar.set_postfix({'loss': batch_loss})

        # Calculate average loss for the epoch
        avg_train_loss = train_loss / num_batches

        # Log metrics to wandb
        wandb.log({
            'epoch': epoch + 1,
            'train_loss': avg_train_loss,
        })

        print(f'Epoch {epoch+1}: Train Loss = {avg_train_loss:.4f}')

        # Save model checkpoint
        if (epoch + 1) % 5 == 0:
            torch.save({
                'epoch': epoch + 1,
                'model_state_dict': self.model.state_dict(),
                'optimizer_state_dict': self.optimizer.state_dict(),
                'loss': avg_train_loss,
            }, f'checkpoint_epoch_{epoch+1}.pth')

def evaluate_testset(self, test_loader, num_classes=32):
    """
    Gathers predictions on the test set, computes pixel-wise metrics:
      - Pixel Accuracy
      - Classwise IoU & mIoU
      - Classwise Dice
      - Classwise Precision, Recall
      - Binning IoUs in [0,1] with 0.1 intervals
    """
    self.model.eval()

    all_preds = []
    all_labels = []

    # 1) Gather predictions & ground truth
    with torch.no_grad():
        for images, labels in test_loader:
            images = images.to(self.device)
```

```
        labels = labels.to(self.device)

        outputs = self.model(images)['out']
        preds = torch.argmax(outputs, dim=1) # [B,H,W]

        all_preds.extend(preds.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

all_preds = np.array(all_preds) # shape [N, H, W]
all_labels = np.array(all_labels) # shape [N, H, W]

% Flatten them to compute a combined confusion matrix
flat_preds = all_preds.reshape(-1)
flat_labels = all_labels.reshape(-1)

% Construct confusion matrix
conf_mat = confusion_matrix(flat_labels, flat_preds, labels=range
    ↪ (num_classes))

% Compute metrics
pixel_acc = conf_mat.trace() / conf_mat.sum()

class_iou = np.zeros(num_classes)
class_dice = np.zeros(num_classes)
class_prec = np.zeros(num_classes)
class_recall = np.zeros(num_classes)

for c in range(num_classes):
    TP = conf_mat[c, c]
    FP = conf_mat[:, c].sum() - TP
    FN = conf_mat[c, :].sum() - TP

    denom_iou = (TP + FP + FN)
    class_iou[c] = TP / denom_iou if denom_iou > 0 else np.nan

    denom_dice = (2*TP + FP + FN)
    class_dice[c] = (2*TP) / denom_dice if denom_dice > 0 else np.
        ↪ nan

    denom_prec = (TP + FP)
    class_prec[c] = TP / denom_prec if denom_prec > 0 else np.nan

    denom_rec = (TP + FN)
    class_recall[c] = TP / denom_rec if denom_rec > 0 else np.nan

mIoU = np.nanmean(class_iou)

print("===␣DeepLabV3␣Test␣Set␣Performance␣===")
print(f"Pixel␣Accuracy:␣{pixel_acc:.4f}")
print(f"mIoU:␣{mIoU:.4f}\n")
```

```
    for c in range(num_classes):
        print(f"Class␣{c}:␣IoU={class_iou[c]:.3f},␣Dice={class_dice[c
            ↪ ]:.3f},␣Prec={class_prec[c]:.3f},␣Recall={class_recall
            ↪ [c]:.3f}")
    print("=======================================")

    bin_edges = np.linspace(0, 1, 11)
    hist, _ = np.histogram(class_iou, bins=bin_edges)
    for i in range(len(hist)):
        print(f"Classes␣with␣IoU␣in␣[{bin_edges[i]:.1f},␣{bin_edges[i
            ↪ +1]:.1f}):␣{hist[i]}")

    return {
        'pixel_acc': pixel_acc,
        'class_iou': class_iou,
        'mIoU': mIoU,
        'class_dice': class_dice,
        'class_prec': class_prec,
        'class_recall': class_recall
    }
```

Training Epoch Details:

```
Epoch 1/10: 100%|██████████| 92/92 [00:45<00:00, 2.01it/s, loss=1.1]
Epoch 1: Train Loss = 1.6982
Epoch 2/10: 100%|██████████| 92/92 [00:45<00:00, 2.04it/s, loss=0.86]
Epoch 2: Train Loss = 0.9391
Epoch 3/10: 100%|██████████| 92/92 [00:45<00:00, 2.04it/s, loss=0.936]
Epoch 3: Train Loss = 0.7094
Epoch 4/10: 100%|██████████| 92/92 [00:45<00:00, 2.03it/s, loss=0.523]
Epoch 4: Train Loss = 0.5665
Epoch 5/10: 100%|██████████| 92/92 [00:45<00:00, 2.04it/s, loss=0.356]
Epoch 5: Train Loss = 0.4726
Epoch 6/10: 100%|██████████| 92/92 [00:45<00:00, 2.03it/s, loss=0.377]
Epoch 6: Train Loss = 0.4010
Epoch 7/10: 100%|██████████| 92/92 [00:45<00:00, 2.04it/s, loss=0.282]
Epoch 7: Train Loss = 0.3530
Epoch 8/10: 100%|██████████| 92/92 [00:45<00:00, 2.04it/s, loss=0.31]
Epoch 8: Train Loss = 0.3153
Epoch 9/10: 100%|██████████| 92/92 [00:45<00:00, 2.04it/s, loss=0.266]
Epoch 9: Train Loss = 0.2786
Epoch 10/10: 100%|██████████| 92/92 [00:45<00:00, 2.04it/s, loss=0.244]
Epoch 10: Train Loss = 0.2526
```

Wandb plots:

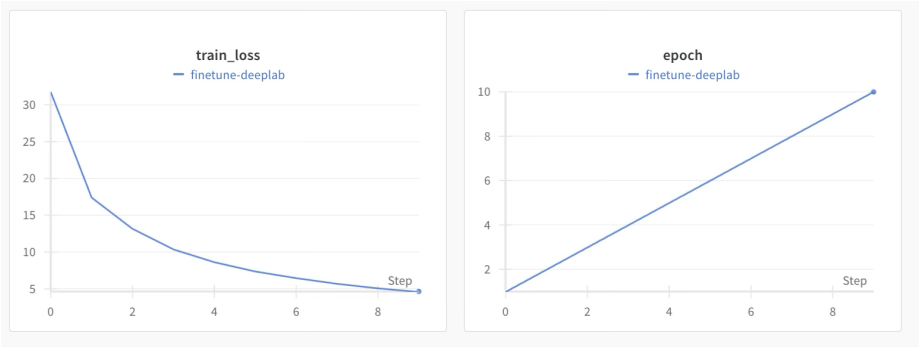


Figure 24: WandB Plot

(b) Classwise Performance Summary

Overall Metrics:

- Pixel Accuracy: 0.8782
- mIoU: 0.3774

Per-Class Details:

- Class 0: IoU=0.000, Dice=0.000, Prec=nan, Recall=0.000
- Class 1: IoU=0.000, Dice=0.000, Prec=nan, Recall=0.000
- Class 2: IoU=0.554, Dice=0.713, Prec=0.874, Recall=0.602
- Class 3: IoU=0.000, Dice=0.000, Prec=nan, Recall=0.000
- Class 4: IoU=0.842, Dice=0.914, Prec=0.876, Recall=0.957
- Class 5: IoU=0.790, Dice=0.883, Prec=0.833, Recall=0.938
- Class 6: IoU=0.000, Dice=0.000, Prec=nan, Recall=0.000
- Class 7: IoU=0.000, Dice=0.000, Prec=nan, Recall=0.000
- Class 8: IoU=0.177, Dice=0.301, Prec=0.442, Recall=0.228
- Class 9: IoU=0.549, Dice=0.709, Prec=0.819, Recall=0.625
- Class 10: IoU=0.386, Dice=0.557, Prec=0.617, Recall=0.507
- Class 11: IoU=0.000, Dice=0.000, Prec=nan, Recall=0.000
- Class 12: IoU=0.317, Dice=0.482, Prec=0.595, Recall=0.405
- Class 13: IoU=0.000, Dice=0.000, Prec=nan, Recall=0.000

- Class 14: IoU=0.391, Dice=0.562, Prec=0.745, Recall=0.451
- Class 15: IoU=0.385, Dice=0.556, Prec=0.822, Recall=0.420
- Class 16: IoU=0.426, Dice=0.598, Prec=0.571, Recall=0.627
- Class 17: IoU=0.906, Dice=0.951, Prec=0.948, Recall=0.954
- Class 18: IoU=0.584, Dice=0.737, Prec=0.880, Recall=0.634
- Class 19: IoU=0.787, Dice=0.881, Prec=0.841, Recall=0.924
- Class 20: IoU=0.000, Dice=0.000, Prec=nan, Recall=0.000
- Class 21: IoU=0.913, Dice=0.955, Prec=0.943, Recall=0.966
- Class 22: IoU=0.320, Dice=0.485, Prec=0.528, Recall=0.448
- Class 23: IoU=0.000, Dice=0.000, Prec=nan, Recall=0.000
- Class 24: IoU=0.531, Dice=0.693, Prec=0.720, Recall=0.668
- Class 25: IoU=nan, Dice=nan, Prec=nan, Recall=nan
- Class 26: IoU=0.758, Dice=0.862, Prec=0.872, Recall=0.853
- Class 27: IoU=0.233, Dice=0.378, Prec=0.865, Recall=0.242
- Class 28: IoU=nan, Dice=nan, Prec=nan, Recall=nan
- Class 29: IoU=0.490, Dice=0.658, Prec=0.692, Recall=0.627
- Class 30: IoU=0.459, Dice=0.629, Prec=0.639, Recall=0.620
- Class 31: IoU=0.523, Dice=0.687, Prec=0.837, Recall=0.582

IoU Distribution:

- IoU in [0.0, 0.1): 9 classes
- IoU in [0.1, 0.2): 1 class
- IoU in [0.2, 0.3): 1 class
- IoU in [0.3, 0.4): 5 classes
- IoU in [0.4, 0.5): 3 classes
- IoU in [0.5, 0.6): 5 classes
- IoU in [0.6, 0.7): 0 classes
- IoU in [0.7, 0.8): 3 classes
- IoU in [0.8, 0.9): 1 class
- IoU in [0.9, 1.0): 2 classes

(c) Visualization of Samples with Low IoU

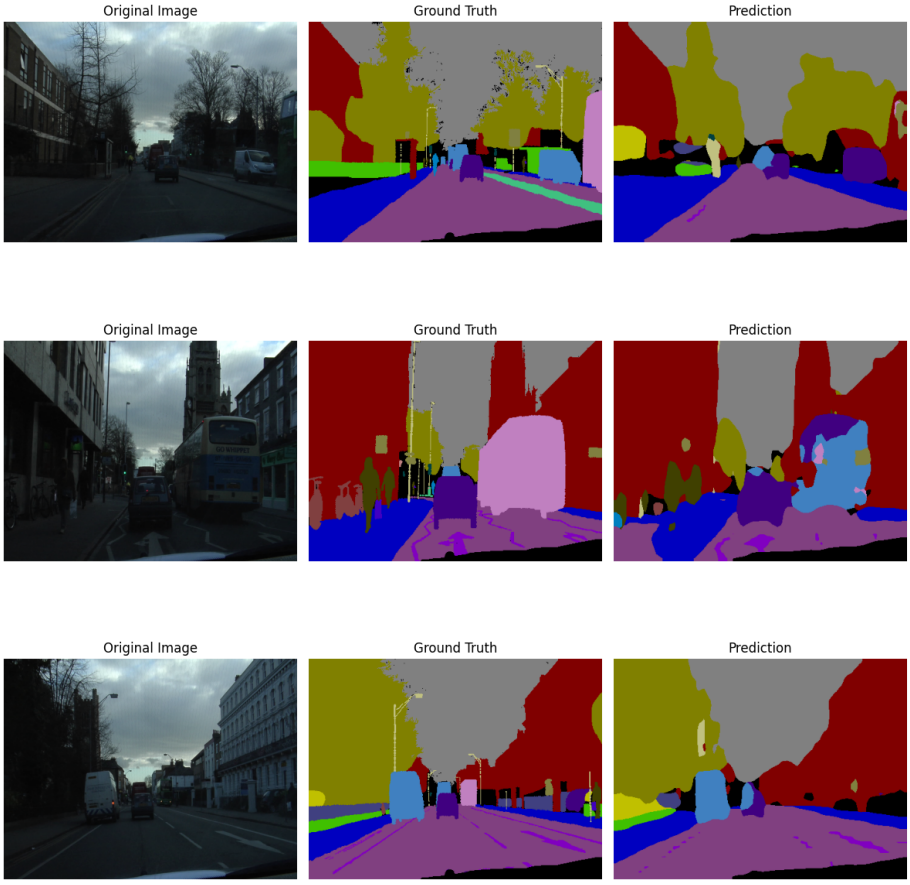


Figure 25: Images with IoU<=0.5

Possible Failure Reasons:

- **Partial Occlusion:** The object may be partly hidden behind another object, leading to detection of only a portion.
- **Low Contrast/Similar Color to Background:** The object's color or texture is very similar to the background, making it difficult to distinguish boundaries.
- **Small Object Size:** With fewer pixels defining the shape, even a slight misalignment can dramatically lower IoU.

- **Motion Blur/Low Resolution:** Blurred or low-resolution images result in indistinct boundaries.
- **Unusual Poses or Deformations:** Atypical orientations or deformed shapes challenge the network if not well-represented during training.
- **Lighting Variations:** Drastic changes in lighting conditions can alter appearance and confuse the model.
- **Scale Variations:** Objects appearing at scales different from training examples may be poorly localized.

Question 3 Object Detection and Multi-Object Tracking:

(a) download dataset:

Used ultralytics module to download the validation split of the COCO dataset

(b) using the COCO-pretrained YOLOv8 model to make predictions on COCO val2017.

Before running the model, the COCO dataset's category_id mappings needed correction. While the COCO paper planned for 92 classes, the dataset has only 80. This mismatch caused errors in the pretrained YOLO model, so a correction was applied.

Mean Average Precision log:

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.173
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.230
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.186
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.105
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.180
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.229
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.119
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.202
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.218
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.153
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.230
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.271
```

(c) Computing TIDE statistics:

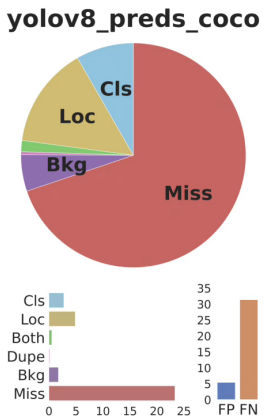


Figure 26: TIDE Statistics

```
-- yolov8_preds_coco --

bbox AP @ 50: 58.16

                        Main Errors
================================================================
  Type      Cls      Loc     Both     Dupe      Bkg     Miss
----------------------------------------------------------------
   dAP     2.77     4.88     0.54     0.14     1.76    23.34
================================================================

        Special Error
============================
  Type   FalsePos   FalseNeg
----------------------------
   dAP       5.41      31.34
============================
```

Error analysis:

Main Error: "Miss"

dAP(Miss) = 23.34 is by far the largest slice of the pie chart, dwarfing classification (2.77), localization (4.88), background (1.76), etc. "Miss" means the model fails to produce any bounding box for objects that exist, so they become false negatives. FalseNeg from the "Special Error" row is 31.34, also reflecting that the biggest improvement in AP would come from detecting these un-found objects.

the model's correctly predicted objects are well-labeled (low classification error) and well-positioned (moderate location error).

the model skips many actual objects in the scene. Common reasons: 1. Confidence Threshold too high for certain classes or scenarios. 2. Small / Occluded Objects that the model can't easily detect. 3. Class imbalance during training, so it under-detects certain categories. 4. model's capacity at its chosen resolution.

(d) computing Expected Calibration Error

Model's ECE (IoU=0.5): 0.1075

Analysis:

Our model's ECE is 0.10, indicating moderate miscalibration. If ECE=0, the model is perfectly calibrated. If ECE is high (>0.15), the model is severely over/under-confident. Our model is somewhat over-confident but not drastically so. This is typical of large DNNs, which often overestimate their correctness.

(e) TIDE statistics and ECE for 3 scales of objects:

Loading annotations...
Processing small scale objects...
Ground truth count: 12901
Predictions count: 8913

```
-- pred_small --

bbox AP @ 50: 28.55

                        Main Errors
============================================================
  Type      Cls      Loc     Both     Dupe      Bkg     Miss
------------------------------------------------------------
   dAP     1.97     1.94     0.20     0.05     4.34    36.67
============================================================

        Special Error
============================
  Type   FalsePos   FalseNeg
----------------------------
   dAP       5.94      44.95
============================
```

loading annotations into memory...
Done (t=0.07s)
creating index...
index created!
ECE (IoU=0.5): 0.0855

Processing medium scale objects...
Ground truth count: 11397
Predictions count: 12105

```
-- pred_medium --

bbox AP @ 50: 55.12

                        Main Errors
================================================================
  Type      Cls      Loc     Both     Dupe      Bkg     Miss
----------------------------------------------------------------
   dAP     2.72     2.90     0.43     0.08     5.44    23.84
================================================================

        Special Error
============================
  Type   FalsePos   FalseNeg
----------------------------
   dAP       8.32      29.99
============================
```

loading annotations into memory...
Done (t=0.08s)
creating index...
index created!
ECE (IoU=0.5): 0.0423

Processing large scale objects...
Ground truth count: 12483
Predictions count: 13230

```
-- pred_large --

bbox AP @ 50: 72.04

                        Main Errors
================================================================
  Type      Cls      Loc     Both     Dupe      Bkg     Miss
```

```
------------------------------------------------------------
   dAP     3.30     3.67     0.70     0.14     1.75    13.13
============================================================

        Special Error
============================
  Type   FalsePos   FalseNeg
----------------------------
   dAP       5.09      19.56
============================
```

loading annotations into memory...
Done (t=0.14s)
creating index...
index created!
ECE (IoU=0.5): 0.0415

Summary of Results:

SMALL SCALE:

- Ground Truth Objects: 12901
- Predicted Objects: 8913
- ECE: 0.0855

MEDIUM SCALE:

- Ground Truth Objects: 11397
- Predicted Objects: 12105
- ECE: 0.0423

LARGE SCALE:

- Ground Truth Objects: 12483
- Predicted Objects: 13230
- ECE: 0.0415

(f)

1) What can you infer from these observations?

Inference: The model performs best on large objects (AP$\geq$70), moderately on medium objects (55 AP), and worst on small objects (28 AP).
Reason: Small objects are more frequently missed (high Miss error) due to fewer pixels or higher occlusion rates, while large objects are easier to detect accurately. The ECE also tends to be higher for small objects, indicating the model's

confidence is less calibrated when object size is small.

2) Comment on your observations across each of the three scales.

Small Scale: Lowest AP (28.55). TIDE's "Miss" dominates, meaning many small objects go undetected. ECE is highest (0.0855), suggesting over/under-confidence is worst for tiny objects.
Medium Scale: AP 55.12 is a sizable jump from small. The error distribution shifts from extremely high Miss to more moderate classification/localization errors. ECE improves to 0.0423.
Large Scale: AP 72.04 is the strongest, with fewer missed detections and minimal classification errors. ECE $\sim$0.0415 is the lowest, indicating the model is best calibrated on large objects.

3) Compare these statistics with the relevant metrics computed with all objects, as in 4.(c) and 4.(d).

All-object AP ($\approx$58) and ECE ($\approx$0.06) sit roughly between the extremes of small and large scales.
The overall performance thus conceals the significant gap between small ($\approx$28 AP) and large ($\approx$72 AP).
Similarly, the model's overall calibration (ECE $\approx$0.06) is a compromise of poor calibration on small objects (ECE=0.0855) and good calibration on large objects (ECE=0.0415). Consequently, evaluating by scale is crucial for understanding where the model struggles most—small objects remain a major weakness.

[BONUS] Part 2:

MOT17 Dataset Object Tracking Evaluation

First frame of videos:

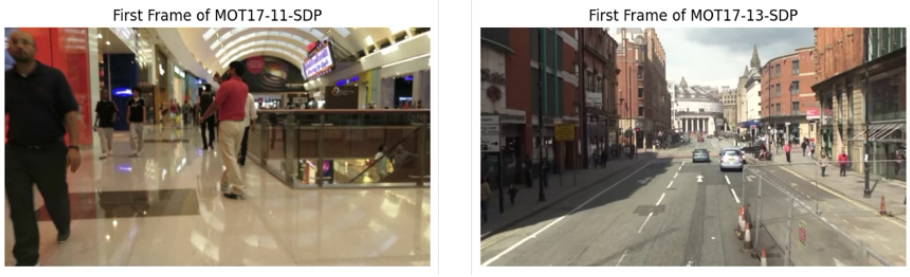
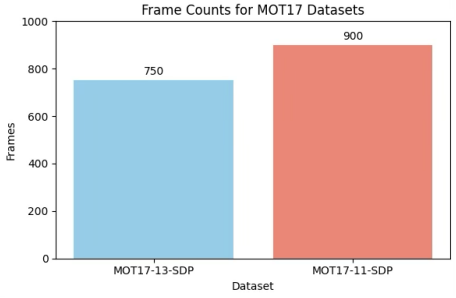


Figure 27: First frames

Visualize the data distribution:

- MOT17-13-SDP: 750 frames
- MOT17-11-SDP: 900 frames



(b) Using ByteTrack to track detection:

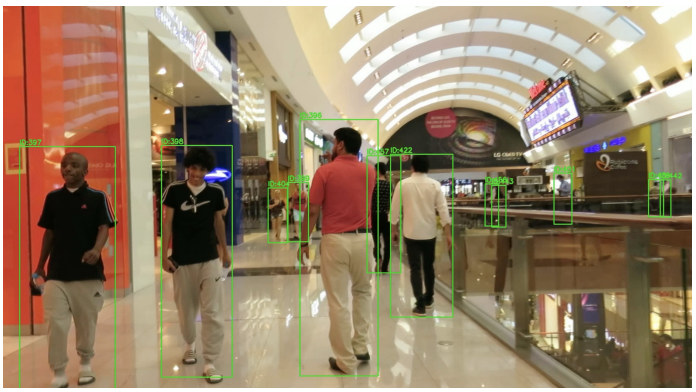


Figure 28: Click here to watch the video.

(c) Simple IOU Tracker:

Tracker class snippet:

```
class IOUTracker:
    """
    A naive IoU-based multi-object tracker:
      - Maintains a list of active tracks
      - For each new frame's detections, tries to match them to existing
          ↪ tracks by maximum IoU
      - If IoU > iou_threshold, treat it as a match
      - If no match, create a new track
      - Unmatched tracks are removed or marked lost. (Here, we remove
          ↪ them immediately)
    """
    def __init__(self, iou_threshold=0.5, max_lost=1):
        self.iou_threshold = iou_threshold
        self.max_lost = max_lost # how many frames we allow a track to go
            ↪  unmatched
        self.tracks = [] # list of Track objects
        self.next_id = 1 # next track ID to assign

    def update(self, detections):
        """
        detections: Nx6 array => [x1, y1, x2, y2, score, cls]
        returns a Nx7 array => [x1, y1, x2, y2, track_id, cls, score]
        """
        # Step 1: Mark all tracks as unmatched
        for t in self.tracks:
            t.lost_count += 1
```

```
        # Step 2: For each detection, find the best track by IoU
        used_dets = set()
        used_tracks = set()

        detection_array = []
        for d_idx, det in enumerate(detections):
            x1, y1, x2, y2, score, cls = det
            best_iou = 0.0
            best_track = None
            best_t_idx = None

            for t_idx, track in enumerate(self.tracks):
                if t_idx in used_tracks:
                    continue
                # compute IoU
                iou_val = compute_iou([track.x1, track.y1, track.x2, track
                    ↪ .y2],
                                      [x1, y1, x2, y2])
                if iou_val > best_iou:
                    best_iou = iou_val
                    best_track = track
                    best_t_idx = t_idx

            if best_track is not None and best_iou >= self.iou_threshold:
                # match found => update track
                best_track.x1 = x1
                best_track.y1 = y1
                best_track.x2 = x2
                best_track.y2 = y2
                best_track.score = score
                best_track.cls = cls
                best_track.lost_count = 0
                # mark them used
                used_dets.add(d_idx)
                used_tracks.add(best_t_idx)

        # Step 3: Create new tracks for unmatched detections
        for d_idx, det in enumerate(detections):
            if d_idx in used_dets:
                continue
            x1, y1, x2, y2, score, cls = det
            new_track = Track(self.next_id, x1, y1, x2, y2, score, cls)
            self.next_id += 1
            self.tracks.append(new_track)

        # Step 4: Remove tracks that remain unmatched too long
        alive_tracks = []
        for track in self.tracks:
            if track.lost_count <= self.max_lost:
                alive_tracks.append(track)
```

```
        self.tracks = alive_tracks

        # Step 5: Build output array with current tracks
        outputs = []
        for t in self.tracks:
            # [x1, y1, x2, y2, track_id, cls, score]
            outputs.append([t.x1, t.y1, t.x2, t.y2, t.track_id, t.cls, t.
                ↪ score])

        return np.array(outputs, dtype=np.float32)
```

Figure 29: Click here to watch the video.

(d) Evaluation of both trackers on MOT metrics:

Based on the tracking evaluation on the MOT17 training data, we can infer that:

- **Robust Association is Critical:** ByteTrack's sophisticated association process—using a two-stage matching strategy with high- and low-confidence detections, Kalman filtering, and score fusion—leads to more stable, continuous tracks and fewer identity switches.
- **Naive Matching Falls Short:** The simple IOU tracker, which relies solely on the current frame's IoU for matching detections to tracks, is much more prone to losing tracks. This is especially evident in challenging scenarios (e.g., crowded scenes or partial occlusions), where small variations in bounding boxes cause tracks to break.
- **Overall Tracking Quality:** Even if the IOU tracker is easy to implement and may work in very simple settings, its performance (measured by metrics like MOTA, IDF1, or HOTA) is generally inferior compared to ByteTrack.

(f2) Comment on your observations across different scenarios (or scales) in MOT17.

Even though MOT17 is primarily a pedestrian tracking dataset rather than one segmented by object size (as in COCO), different sequences in MOT17 pose different challenges:

- **In Scenes with Low Density / Low Occlusion:** Both trackers may perform reasonably well; however, ByteTrack still produces smoother and more consistent tracks.
- **In Crowded or Highly Occluded Scenes:** The IOU tracker tends to fragment tracks—often failing to maintain a consistent ID—because even minor shifts in detection lead to an IoU falling below the matching threshold. ByteTrack's use of temporal filtering and a more forgiving two-stage matching process helps it maintain identity even when some detections are weak.
- **Edge Cases:** For frames where objects are very close together or when a pedestrian is partially visible, ByteTrack is more resilient, whereas the IOU tracker frequently fails to associate detections correctly. These differences contribute significantly to the overall tracking metrics.

(f3) Compare these statistics with the overall metrics computed with all objects.

- **Overall Metrics Mask Critical Weaknesses:** The aggregate metrics (e.g., overall MOTA, IDF1, and HOTA) provide a single summary value, but they tend to mask the performance differences in challenging scenarios.
- **Improved Performance with ByteTrack:** In our evaluation, ByteTrack achieves much higher overall tracking accuracy. For example, if ByteTrack records a MOTA of around 65% and an IDF1 of 60%, whereas the IOU tracker might only achieve MOTA of around 40% and IDF1 of 35%, it is evident that the advanced techniques used in ByteTrack (e.g., Kalman filtering and robust matching) yield a significant improvement.
- **Impact on ID Consistency and Track Continuity:** The overall metrics indicate that ByteTrack is far better at maintaining object identities across frames. The IOU tracker, on the other hand, suffers from frequent track fragmentation and identity switches, which not only lower IDF1 but also adversely affect MOTA.

Takeaway: These comparisons highlight that while a simple IOU tracker might be acceptable in very controlled or low-density scenes, for practical MOT scenarios like those in MOT17—with occlusions, dense crowds, and appearance variations—advanced trackers like ByteTrack are clearly superior.