CV ASSIGNMENT - 1

SHOBHIT RAJ
2022482

Q1. THEORY

1. The original one-hot vector $y \in \{0,1\}^K$ is modified using label smoothing. The smoothed target distribution $\tilde{y}$ is as follows:

$$\tilde{y}_i = \begin{cases} 1 - \epsilon + \frac{\epsilon}{K} & \text{if } i = \text{correct class} \\ \frac{\epsilon}{K} & \text{if } i \neq \text{correct class} \end{cases}$$

($\tilde{y}_i$: probability assigned to class i)

a) As $H(p,q) = E_p[-\log q(x)] = -\sum_{i=1}^{K} p_i \log q_i$

$\therefore$ $H(\tilde{y}, q)$ → cross entropy with label smoothing

$$H(\tilde{y},q) = -\sum_{i=1}^{K} \tilde{y}_i \log q_i$$

$$= -\left[\left(1-\epsilon+\frac{\epsilon}{K}\right)\log q_{correct} + \sum_{i \neq correct} \frac{\epsilon}{K} \log q_i\right]$$

$$= -\left(1-\epsilon+\frac{\epsilon}{K}\right)\log q_{correct} - \frac{\epsilon}{K}\sum_{i \neq correct} \log q_i$$

$$= -(1-\epsilon)\log q_{correct} - \frac{\epsilon}{K}\sum_{i=1}^{K} \log q_i$$

(since $\sum_{i=1}^{K} \log q_i = \sum_{i \neq correct} \log q_i + \log q_{correct}$)

$\therefore$ $H(\tilde{y},q) = -(1-\epsilon)\log q_{correct} - \frac{\epsilon}{K}\sum_{i=1}^{K} \log q_i$

b) The above loss can be interpreted as:

$$H(\tilde{y},q) = (1-\epsilon)\underbrace{(-\log q_{correct})}_{\text{original loss (CE)}} + \epsilon\underbrace{\left(\frac{-1}{K}\sum_{i=1}^{K}\log q_i\right)}_{\text{Uniform loss}}$$

Here, the first term ensures the model focuses on the correct class. The second term acts as a regularizer by encouraging predictions to align with a uniform distribution which prevents the model from ignoring incorrect classes entirely.

P.T.O.

Label smoothing reduces the confidence of the model by preventing it from assigning a probability of 1 to a single class. This helps improve generalization & reduces overfitting as by softening the target distribution, label smoothing makes the model more robust to label noise, as it no longer strictly enforces a 0-1 target for incorrect classes.

2. $p(x) = \frac{1}{\sqrt{2\pi\sigma_p^2}} e^{-\frac{(x-\mu_p)^2}{2\sigma_p^2}}$; $q(x) = \frac{1}{\sqrt{2\pi\sigma_q^2}} e^{-\frac{(x-\mu_q)^2}{2\sigma_q^2}}$

a) $H(p,q) = E_{p(x)}[-\log q(x)]$

b) $-\log q(x) = -\log\left(\frac{1}{\sqrt{2\pi\sigma_q^2}} e^{-\frac{(x-\mu_q)^2}{2\sigma_q^2}}\right)$

$= -\log \frac{1}{\sqrt{2\pi\sigma_q^2}} + \frac{(x-\mu_q)^2}{2\sigma_q^2}$

$\therefore E_{p(x)}\left[-\log \frac{1}{\sqrt{2\pi\sigma_q^2}} + \frac{(x-\mu_q)^2}{2\sigma_q^2}\right] =$

$= -\log \frac{1}{\sqrt{2\pi\sigma_q^2}} + E_{p(x)}\left[\frac{(x-\mu_q)^2}{2\sigma_q^2}\right]$ (as $-\log \frac{1}{\sqrt{2\pi\sigma_q^2}}$ = const.)

$\rightsquigarrow E_{p(x)}[(x-\mu_q)^2] \approx E_{p(x)}[(x-\mu_p+\mu_p-\mu_q)^2]$

$\Rightarrow E_{p(x)}[(x-\mu_p)^2 + 2(x-\mu_p)(\mu_p-\mu_q) + (\mu_p-\mu_q)^2]$

$\Rightarrow \underbrace{E_{p(x)}[(x-\mu_p)^2]}_{\text{Var}(x)=\sigma_p^2} + 2(\mu_p-\mu_q)\underbrace{E_{p(x)}[x-\mu_p]}_{\int_{-\infty}^{\infty}(x-\mu_p)p(x)dx} + \underbrace{E_{p(x)}[(\mu_p-\mu_q)^2]}_{(\mu_p-\mu_q)^2 \text{ as constant}}$

$= \int_{-\infty}^{\infty} x\,p(x)\,dx - \mu_p \int_{-\infty}^{\infty} p(x)\,dx$

$= \mu_p - \mu_p \cdot 1 = 0.$

$\Rightarrow \sigma_p^2 + (\mu_p-\mu_q)^2$

$\therefore H(p,q) = -\log \frac{1}{\sqrt{2\pi\sigma_q^2}} + \frac{\sigma_p^2 + (\mu_p-\mu_q)^2}{2\sigma_q^2}$

P.T.O.

$$\Rightarrow H(p,q) = \frac{1}{2}\log(2\pi\sigma_q^2) + \frac{\sigma_p^2 + (\mu_p - \mu_q)^2}{2\sigma_q^2}$$

c) If $\sigma_p = \sigma_q = \sigma$

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

$$= \underbrace{\frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2}}_{\text{entropy of gaussian dist}^n} + \underbrace{\frac{(\mu_p - \mu_q)^2}{2\sigma^2}}_{\text{squared diff. b/t means}}$$

First term represents the inherent uncertainty in the true distribution $p(x)$.
Second term measures how well mean of $q(x)$ matches the mean of $p(x)$. It grows as μ_p & μ_q become more distant, meaning cross entropy increases when 2 distributions are farther apart. If $\mu_q = \mu_p$, this term vanishes & cross entropy reduces to the entropy of $p(x)$.

Both these show cross entropy punishes variance differences as well as mean mismatches.

3. a) For 1D convolution, the receptive field of single layer is

$$RF = K + (K-1)(r-1) = 1 + r(K-1)$$

K: kernel spans K elements
$(K-1)(r-1)$: dilation introduces gaps of size '$r-1$' b/t consecutive elements.

Case 1 - L stacked conv. layers each with dilation r.

Total receptive field $= (1 + (K-1)r_1) + (K-1)r_1 r_2 + \ldots + (K-1)r_1 r_2 \ldots r_L$

$$= 1 + (K-1)\sum_{l=1}^{L} r^l \quad (\text{as } r_1 = r_2 = \ldots = r_L = r)$$

$$= 1 + (K-1)(r + r^2 + r^3 + \ldots + r^L)$$

$$= 1 + (K-1)\, r \cdot \left(\frac{r^L - 1}{r - 1}\right) \leftarrow \text{gp sum for } r > 1$$

or

$1 + (K-1)\cdot L$ if $r = 1$.

P.T.O.

Case 2 - r increases layerwise by factor of i (eg. $i=2$ in ques.)

Total receptive field $= 1 + (K-1) r_1 + (K-1) r_1 r_1 i + (K-1) r_1 r_1 i r_i i^2 + \dots$

$$= 1 + (K-1) \sum_{l=1}^{L} r^l \, i^{\frac{l(l-1)}{2}}$$

this will be $\geq 1 + (K-1) r^L \, i^{\frac{L(L-1)}{2}}$

which shows total RF grows exponentially.

When $i=2$ as in question, RF $= 1 + (K-1) r^L \cdot 2^{L(L-1)/2}$.

b) For 2D convolutions, with KXK kernel, the receptive field in both height & width dimensions follows the 1D formula.

Assuming dilation factor r_l are applied in both dimension uniformly, RF becomes:-

$$RF_{2D} = \left[1 + (K-1) \sum_{l=1}^{L} r_l\right] \times \left[1 + (K-1) \sum_{l=1}^{L} r_l\right]$$

where r_l increases exponentially (i.e. $r^l = 2^{l-1}$)

Hence, RF grows exponentially in L (or r) for each dimension, resulting in square region of size $O((2^L)^2)$ i.e. grows quadratically in terms of area & exponentially in terms of no. of layers.

c) Let feature map size be HXW & C_{in} and C_{out} be # in channels & # out channels.

Standard Convolution - Each output pixel requires $K \times K \times C_{in}$ multiplications & total # output pixels in feature map $= H \times W \times C_{out}$

$\therefore$ Total multiply-add operations $= H \times W \times C_{out} \times$

$\rightsquigarrow$ Standard Conv. Complexity $= O(H \times W \times C_{in} \times C_{out} \times \underbrace{(K^2 \times C_{in} + K^2 \times C_{in} - 1)}_{\text{multiply} \quad \text{add}})$.

Dilated Convolution - Total # operations remain same as dilation doesn't change # Kernel parameters or computations, but just the spacing of input sampling.

$\rightsquigarrow$ Dilated Conv. Complexity $= O(H \times W \times C_{in} \times C_{out} \times K^2)$.

Both have identical computational complexity for fixed feature map size.

Question 2: Image Classification

Part 1. Dataset, Dataloaders & Visualization

This task involves handling the "Russian Wildlife Dataset" for image classification. The dataset consists of **10 classes** of wildlife images and has a total of **11,668 images**.

(a) A custom PyTorch Dataset class, `RussianWildlifeDataset`, was implemented to read images from class directories and apply transformations. Each image is assigned a label based on a predefined dictionary mapping class names to numerical values. All images were resized to `(224, 224, 3)` to maintain a consistent input shape for model training, inspired by AlexNet. To optimize loading efficiency, preprocessing and transformations were handled within `__init__` itself. An **80:20 stratified split** was performed using `train_test_split()` to ensure balanced distribution. The training set contains **9,334 images**, while the validation set has **2,334 images**, maintaining an 80:20 split. **WandB** was initialized to log experiment results and visualizations. A screenshot of the `RussianWildlifeDataset` class is shown below.

```python
class RussianWildlifeDataset(Dataset):
    """
    Custom Dataset for Russian Wildlife Image Classification
    """
    def __init__(self, img_dir, transform=None, target_transform=None):
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform
        self.data = []
        self.labels = []
        for folder in os.listdir(img_dir):
            folder_path = os.path.join(img_dir, folder)
            for file in os.listdir(folder_path):
                img = Image.open(os.path.join(folder_path, file))
                label = class_labels[folder]
                if self.transform:
                    img = self.transform(img)
                if self.target_transform:
                    label = self.target_transform(label)
                self.data.append(img)
                self.labels.append(label)
            # print(f"{folder}: {len(os.listdir(folder_path))}")
            # print(list(zip(self.data[-5:], self.labels[-5:])))

        # print(len(self.data))
        # print(len(self.labels))

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

    def __getitem__(self, idx):
        return self.data[idx], self.labels[idx]
```

Figure 1: RussianWildlifeDataset Custom Dataset

(b) The dataset was then wrapped into **DataLoader** objects to facilitate efficient batch loading.

```
# creating dataloaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
```

Figure 2: Dataloaders

(c) The class distributions for the full dataset, training set, and validation set were visualized using bar charts. The generated distribution plot was saved as `class_distribution.png` and logged using **Weights & Biases (WandB)** for tracking.

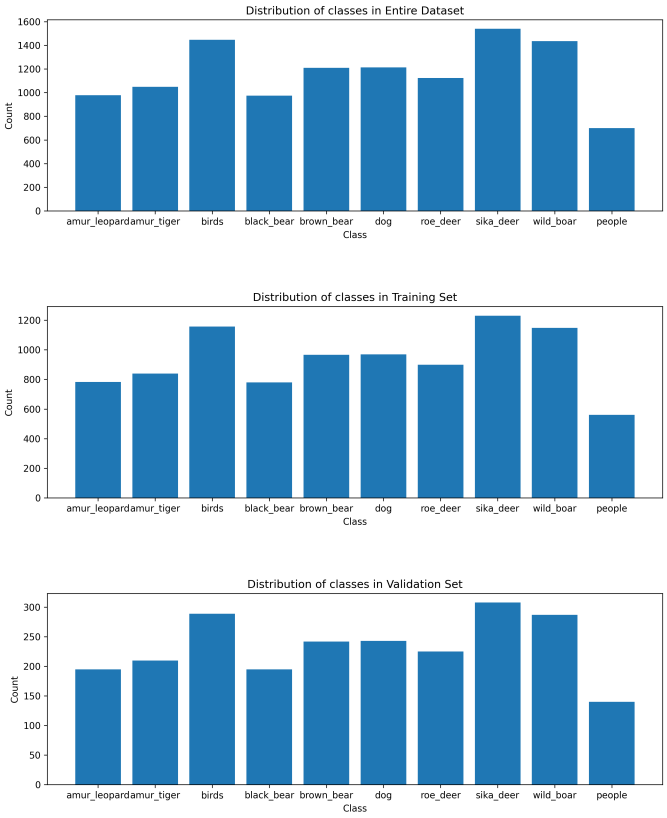


Figure 3: Distribution of Classes Visualization

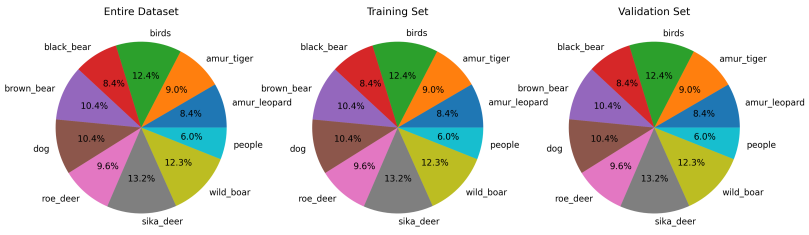


Figure 4: Distribution of Classes Visualization in Pie Chart

These visualization shows a balanced class distribution between both the training & validation sets, ensuring fairness in model training and evaluation.

Part 2. Training a CNN from Scratch for Russian Wildlife Dataset

In this task, I designed and trained a Convolutional Neural Network (CNN) for classifying images in the Russian Wildlife Dataset.

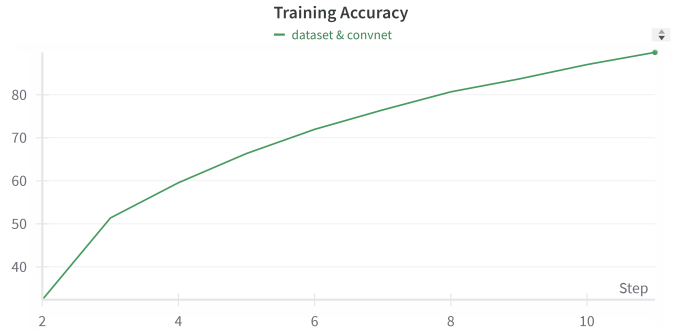
(a) The CNN consists of three convolutional layers with increasing feature maps (**32, 64, 128**) and max-pooling layers for downsampling. The final feature map is flattened and passed through a fully connected classification head. A screenshot of the `ConvNet` model class is shown below.

```python
class ConvNet(nn.Module):
    """
    Custom Convolutional Neural Network for image classification
    """
    def __init__(self):
        super(ConvNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=4, stride=4)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc = nn.Linear(128 * 14 * 14, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.relu2(x)
        x = self.pool2(x)
        x = self.conv3(x)
        x = self.relu3(x)
        x = self.pool3(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

Figure 5: ConvNet Custom CNN

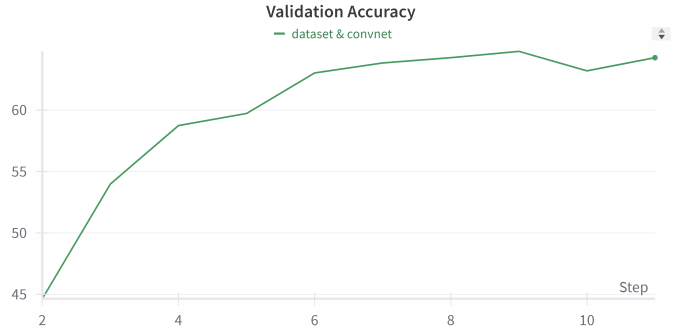
(b) I trained the model using the **Cross-Entropy Loss** with the **Adam optimizer** for 10 epochs. **Weights & Biases (wandb)** was used to log training and validation losses and accuracies. The trained model's state dictionary was saved as `convnet.pth`. The plots are shown below.

(a) Training Loss



(b) Training Accuracy

(c) Validation Loss



(d) Validation Accuracy

Figure 6: Training and Validation Losses and Accuracies logged in WandB

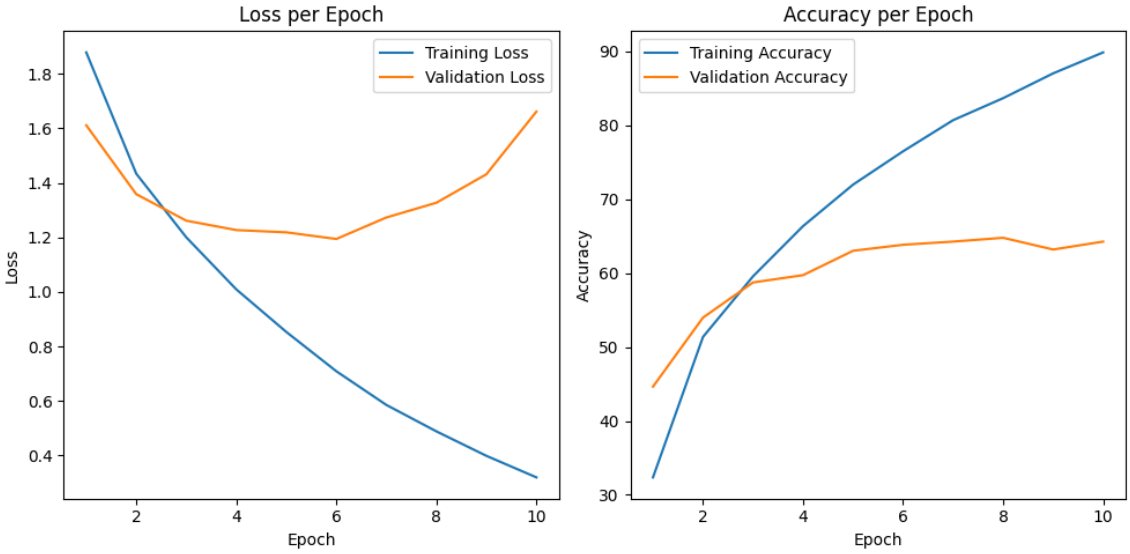
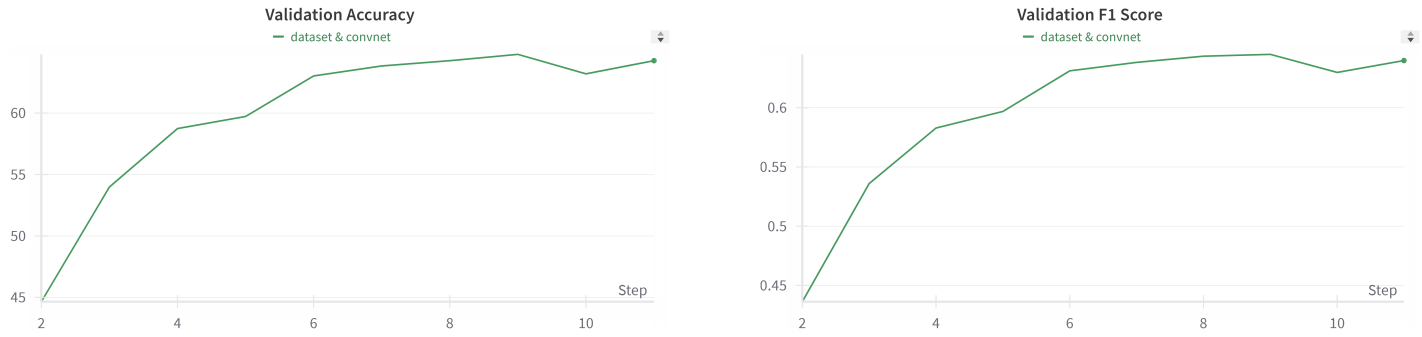


Figure 7: Training and Validation Losses and Accuracies plotted in Python

(c) It can be observed in Figure 6 (c) and Figure 7, the model is **overfitting** as the training loss continues to decrease while the validation loss increases.

(d) The trained model achieved an **Accuracy of 64.27%** and an **F1-Score of 0.6398** on the validation set. The **confusion matrix** was logged using **wandb**. The plots and confusion matrix are shown below.



(a) Validation Accuracy

(b) Validation F1-Score

Figure 8: Validation Metrics: Accuracy and F1-Score

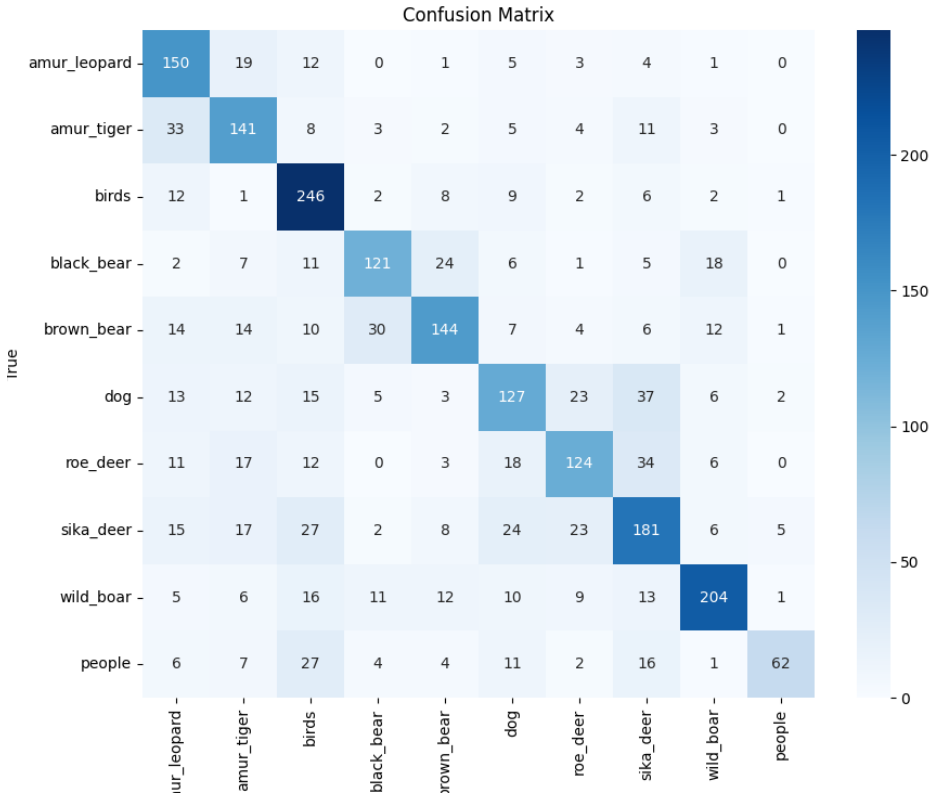


Figure 9: Confusion Matrix

(e) 3 misclassified samples per class along with their predicted class label are shown below.

Figure 10: Misclassified Images

The model fails due to several factors:

- **Similar visual features:** For example, the **Amur leopard** and **Amur tiger** are confused due to similar fur patterns.
- **Poor image quality:** Dark or blurry images, like birds mistaken for **sika deer**, hinder accurate classification.
- **Limited training data diversity:** Insufficient varied angles and postures, such as dogs misclassified as **sika deer**.

To improve performance:

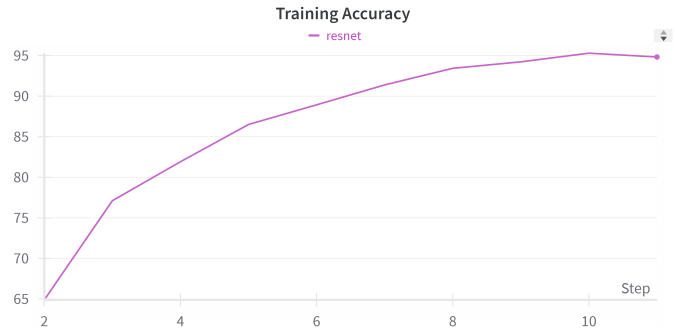
- **Enhance image preprocessing:** Improve brightness and sharpness for low-quality images.
- **Increase dataset diversity:** Include varied angles, lighting, and postures in training.
- **Use advanced feature extraction:** Focus on unique distinguishing characteristics.
- **Fine-tune with domain-specific knowledge:** Emphasize specific features of each animal for better accuracy.

Part 3. Fine-tuning Resnet-18 model for Russian Wildlife Dataset

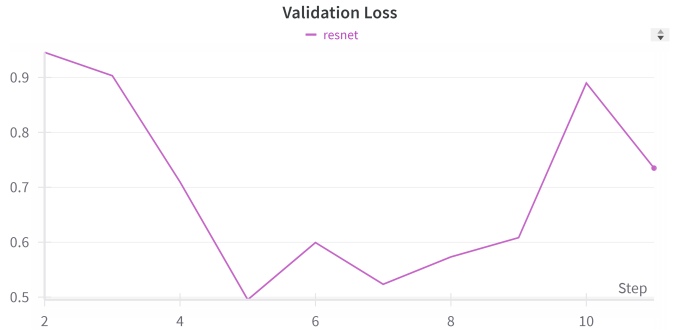
In this task, I fine-tuned ResNet-18 for image classification and feature visualization.

(a) I used a **pretrained ResNet-18 (trained on ImageNet)** and replaced the final fully connected (FC) layer for classifying images from the Russian Wildlife Dataset. Using the same training strategy as ConvNet model, I monitored performance metrics such as loss and accuracy using **wandb**. The trained model's state dictionary was saved as `resnet.pth`. The plots are shown below.

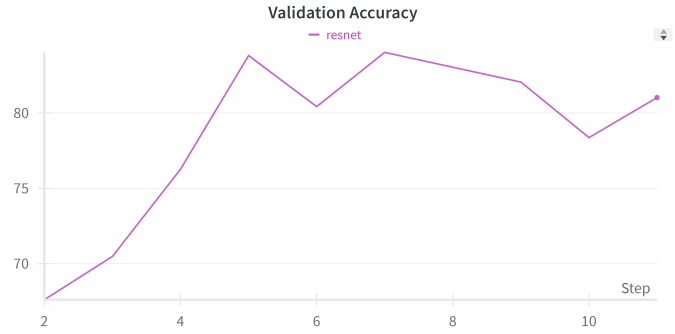
(a) Training Loss



(b) Training Accuracy



(c) Validation Loss



(d) Validation Accuracy

Figure 11: Training and Validation Losses and Accuracies logged in WandB

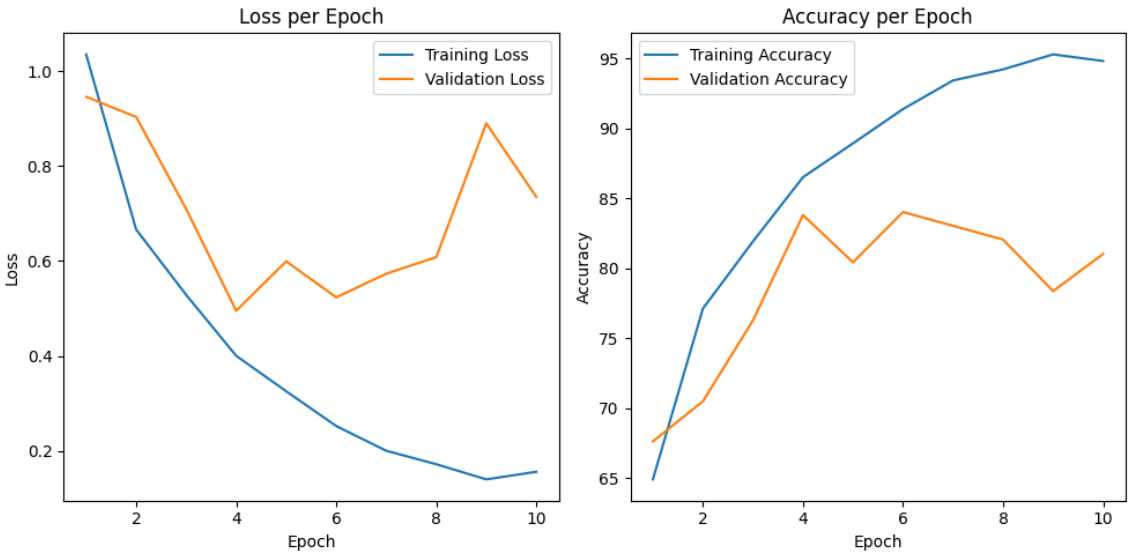
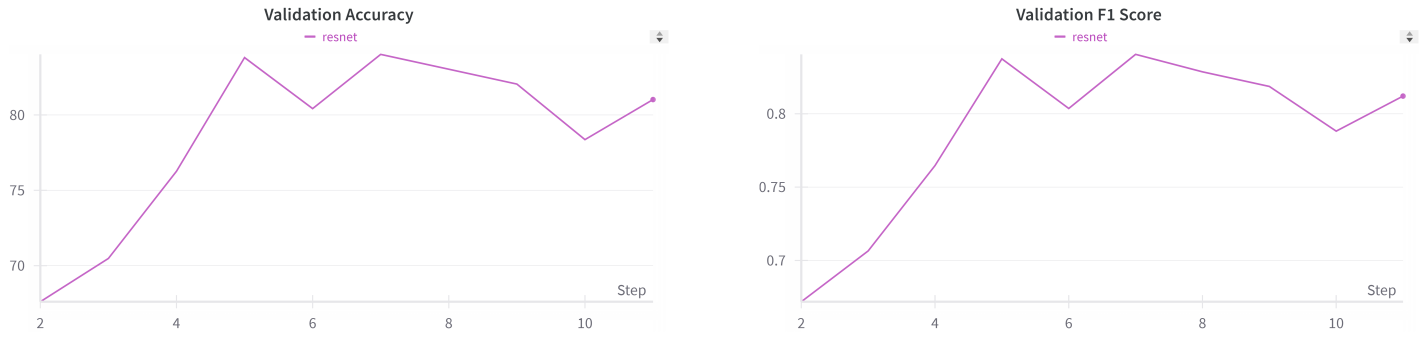


Figure 12: Training and Validation Losses and Accuracies plotted in Python

(b) It can be observed in Figure 11 (c) and Figure 12, the model is **overfitting** as the training loss continues to decrease while the validation loss increases and the gap between them widens.

(c) The trained model achieved an **Accuracy of 81.02%** and an **F1-Score of 0.8121** on the validation set. The **confusion matrix** was logged using **wandb**. The plots and confusion matrix are shown below.



(a) Validation Accuracy

(b) Validation F1-Score

Figure 13: Validation Metrics: Accuracy and F1-Score

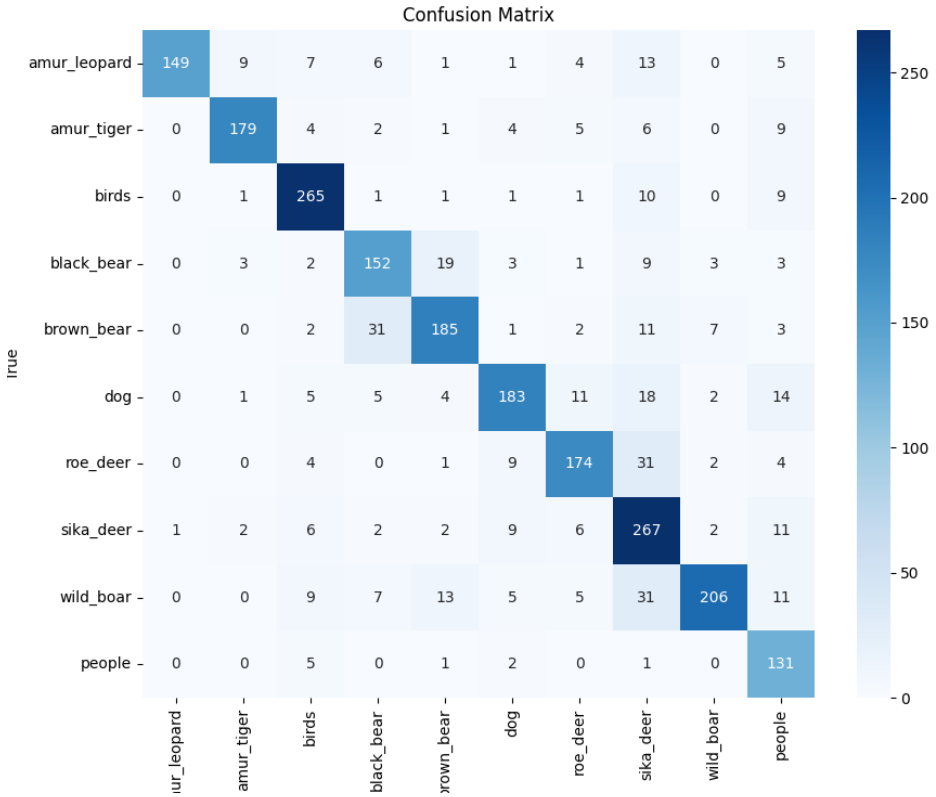
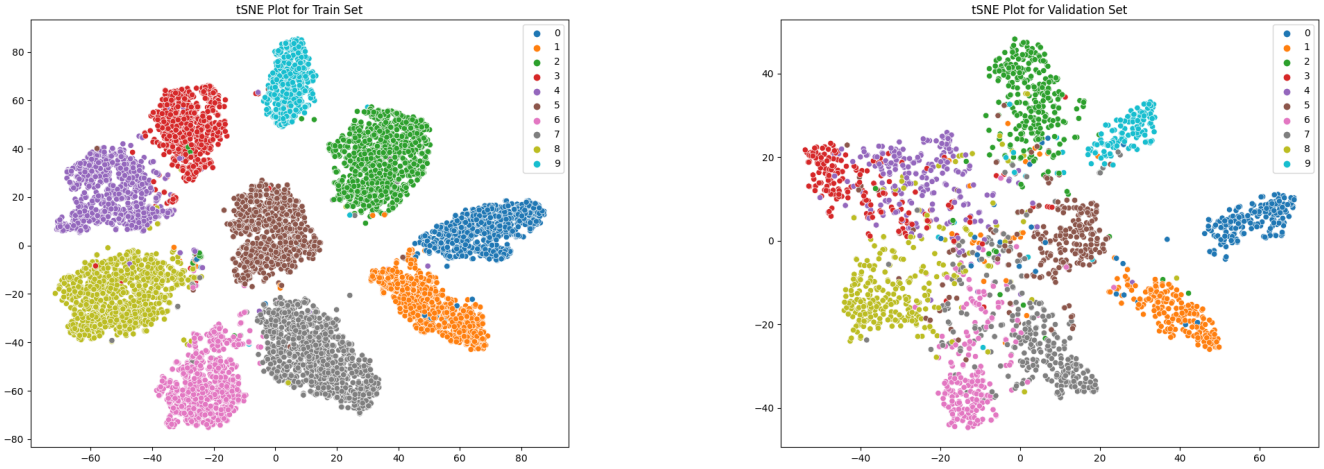


Figure 14: Confusion Matrix

(d) The feature vectors were extracted from the backbone of **ResNet-18**. **t-SNE** was used to visualize the feature space in 2D and 3D, showing how data samples cluster based on class labels. The plots are shown below.



(a) Training Set tSNE plot in 2D space

(b) Validation Set tSNE plot in 2D space

Figure 15: tSNE Visualizations in 2D space

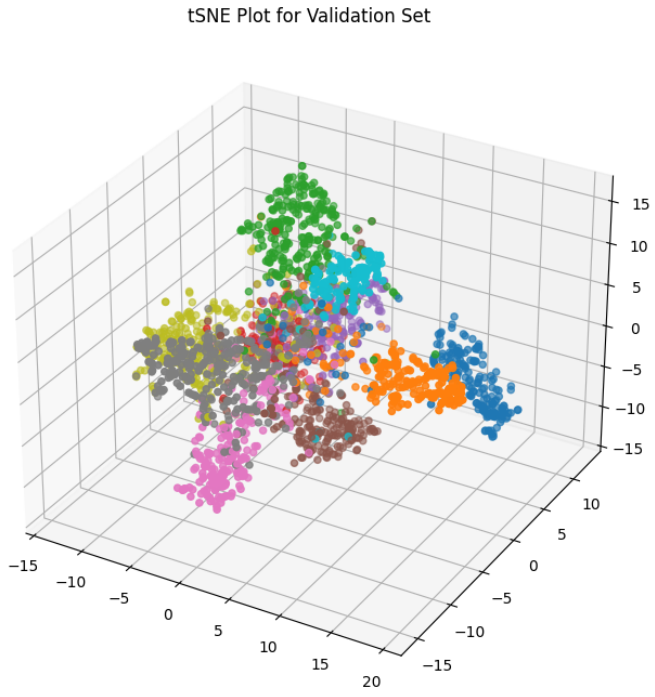


Figure 16: Validation Set tSNE plot in 3D space

Part 4. Data Augmentation techniques

In this task, I augmented dataset with more training data and trained the model on that dataset.

(a) To improve model generalization and address misclassification issues in Russian wildlife image classification, I applied data augmentation techniques to **expand the training dataset from 9,334 to 37,336 images** (original + 3 augmentations per image). The augmentations were applied **on the fly during training** using `transforms.Compose` in PyTorch to avoid excessive RAM usage that would occur if all augmented images were pre-stored. The chosen augmentations specifically targeted observed model weaknesses:

- **RandomAffine** – Applied small translations and scaling to introduce slight positional variations, addressing the issue of misclassification due to similar visual features, such as Amur leopards and tigers.
- **ColorJitter** – Adjusted brightness and contrast to account for variations in image quality, mitigating errors caused by poor lighting conditions (e.g., dark or blurry images leading to bird-deer misclassification).
- **RandomAdjustSharpness** – Enhanced edge clarity, which is crucial for distinguishing fine-grained differences between species.
- **RandomPerspective** – Introduced minor distortions to simulate different viewing angles and postures, helping differentiate visually similar species, such as dogs and sika deer.

The augmentations applied & augmented images are shown below.

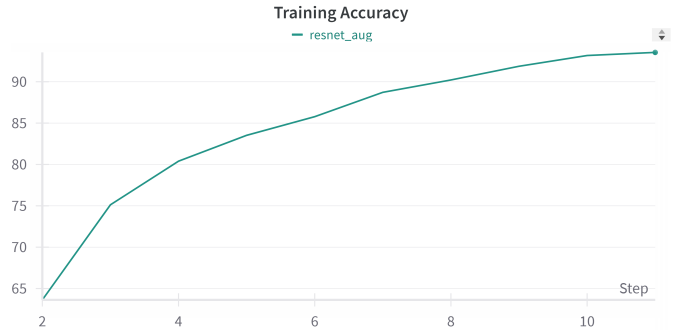
```
augmentations = transforms.Compose([
    transforms.RandomAffine(degrees=0, translate=(0.1, 0.1), scale=(0.9, 1.1)),
    transforms.ColorJitter(brightness=0.3, contrast=0.2),
    transforms.RandomAdjustSharpness(sharpness_factor=2, p=0.5),
    transforms.RandomPerspective(distortion_scale=0.2, p=0.5),
])
```

Figure 17: Augmentations applied

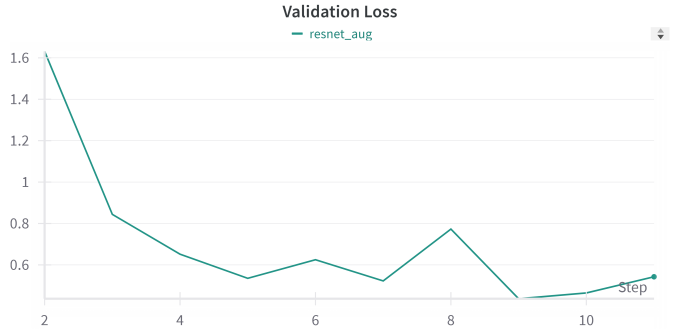
Figure 18: Visualization of augmented images

(b) Using the same training strategy as before, I monitored performance metrics of **ResNet-18 model** on the augmented training dataset such as loss and accuracy using **wandb**. The trained model's state dictionary was saved as `resnet_aug.pth`. The plots are shown below.

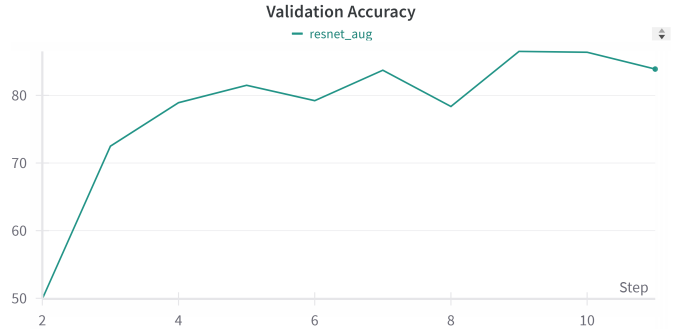
(a) Training Loss



(b) Training Accuracy



(c) Validation Loss



(d) Validation Accuracy

Figure 19: Training and Validation Losses and Accuracies logged in WandB

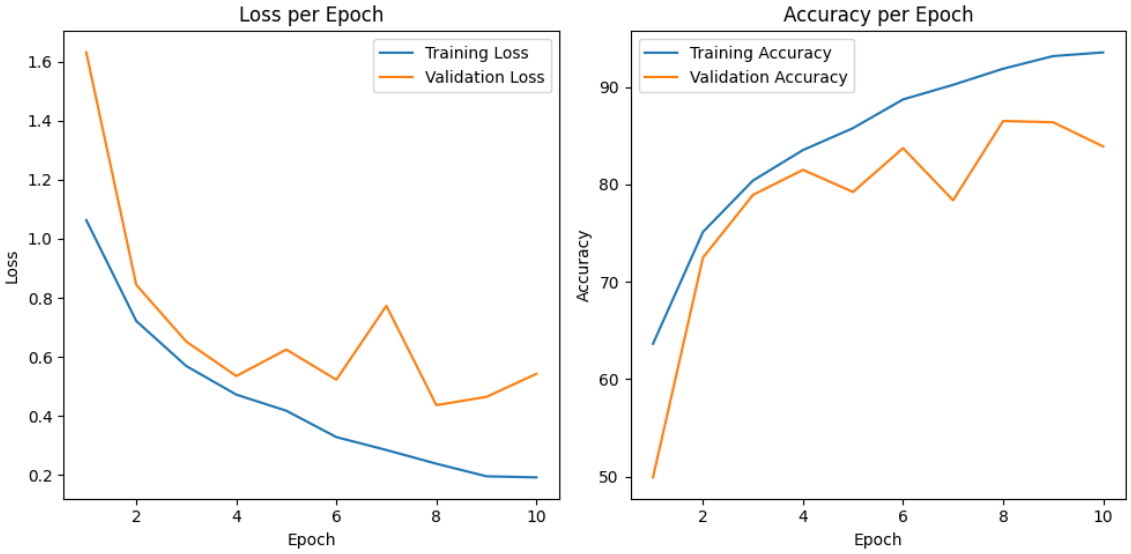
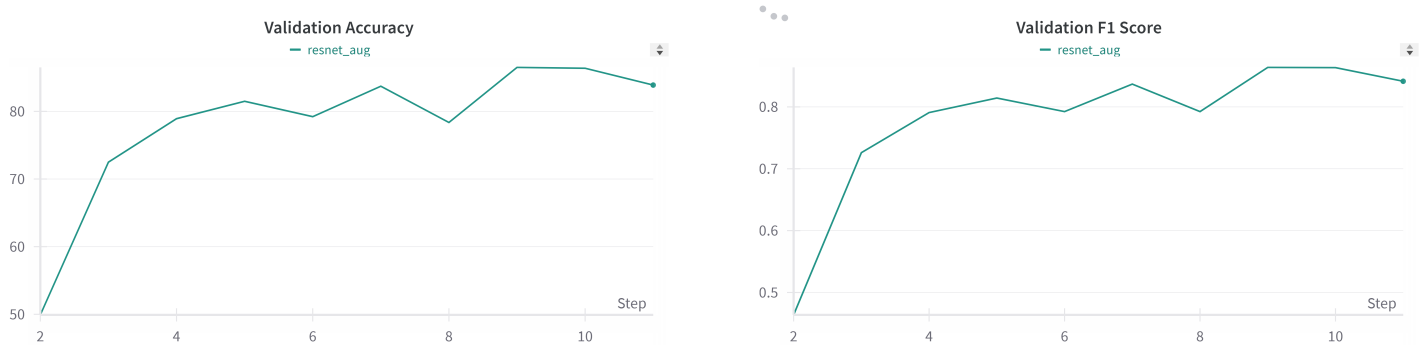


Figure 20: Training and Validation Losses and Accuracies plotted in Python

(c) It can be observed in Figure 19 (c) and Figure 20, augmentation helped in resolving overfitting, as the validation loss and accuracy are more aligned with lower gaps between training and validation metrics compared to the ConvNet and ResNet without augmentation, which show a larger gap between training and validation metrics, indicating **better generalization**.

(d) The trained model achieved an **Accuracy of 83.89%** and an **F1-Score of 0.8415** on the validation set. The **confusion matrix** was logged using **wandb**. The plots and confusion matrix are shown below.



(a) Validation Accuracy

(b) Validation F1-Score

Figure 21: Validation Metrics: Accuracy and F1-Score

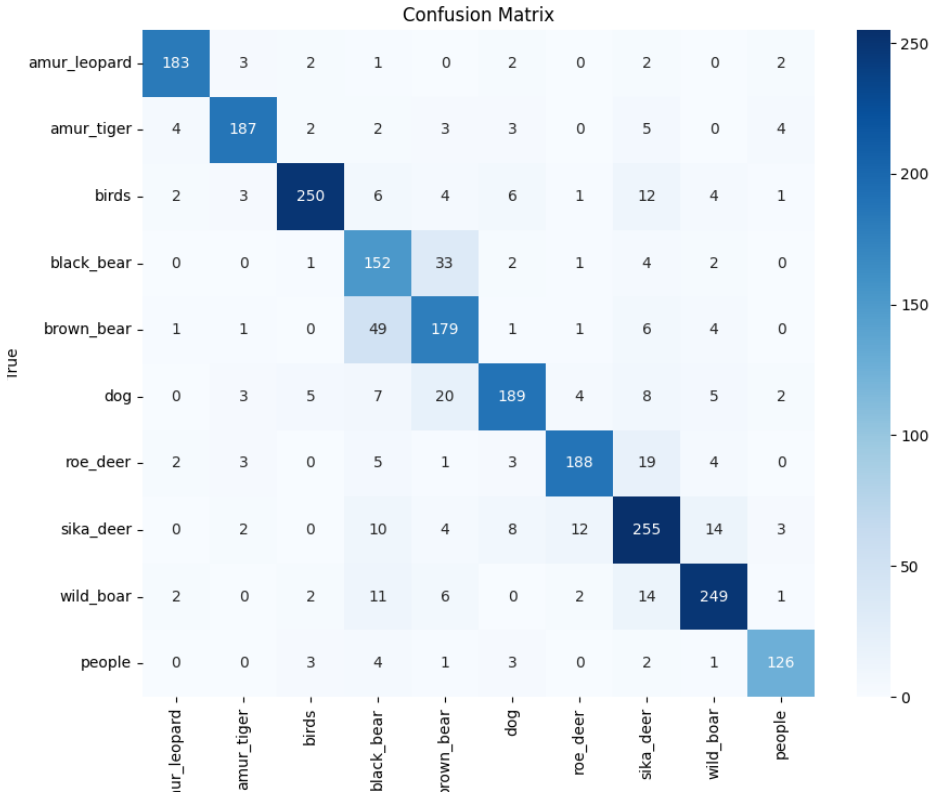


Figure 22: Confusion Matrix

Part 5. Comparison between Models

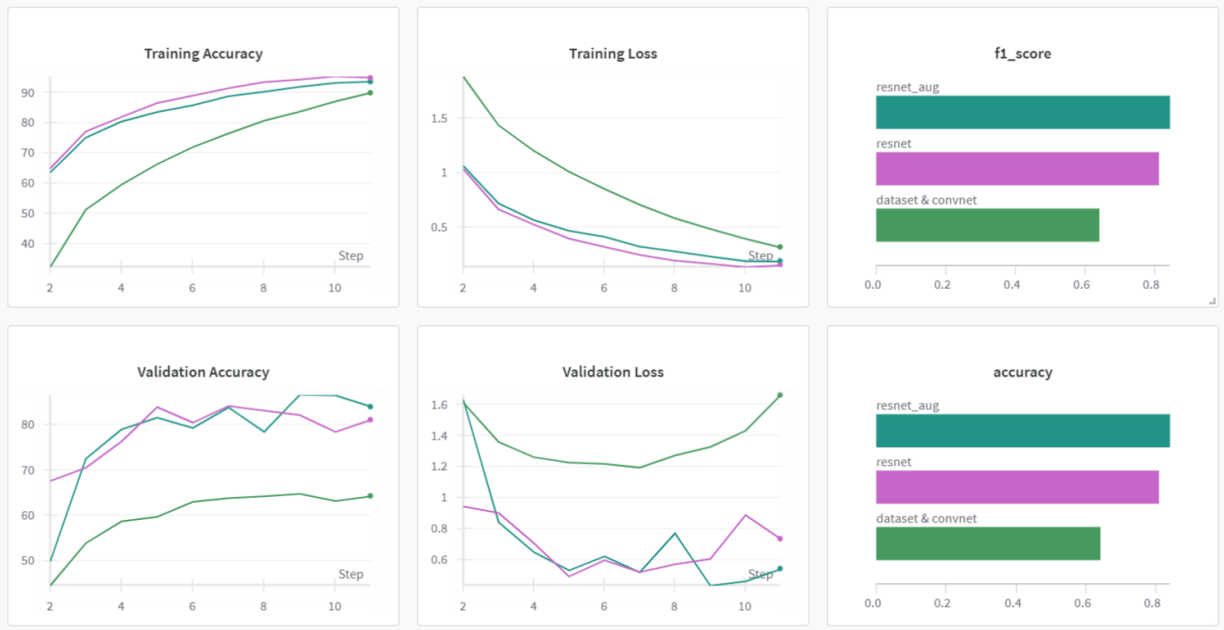


Figure 23: Comparison between Models' Plots

As observed in the WandB plots, the ResNet with augmentation demonstrates superior performance, achieving the highest accuracy and F1 score while maintaining a closer alignment between training and validation metrics, indicating better generalization. The standard ResNet shows good performance but with a slightly larger gap between training and validation, suggesting some overfitting. The ConvNet exhibits the lowest performance with significant overfitting, as indicated by the large disparity between training and validation metrics.

References

- For creating a custom dataset and dataloader with a stratified train-validation split, I referred to my own implementation from `ML Assignment 4, Section C, Part 1`, where a custom dataset and dataloader were prepared for the CIFAR-10 dataset. [GitHub Repository]
- For the training loop, I referred to my own implementation from `DL Assignment 1, Question 2(c)`, where training loops for MLP and CNN models were designed for the FashionMNIST dataset. [GitHub Repository]
- **Datasets and DataLoaders Tutorial:** [PyTorch Documentation]
- **Weights and Biases (W&B) QuickStart:** [W&B QuickStart]
- **Pretrained ResNet-18 Weights:** [PyTorch ResNet-18]
- **t-SNE for Feature Space Visualization:** [scikit-learn Documentation]
- **Transforming and Augmenting Images:** [PyTorch Transforms]

Question 3: Image Segmentation

Part 1. Dataset, Dataloaders & Visualization

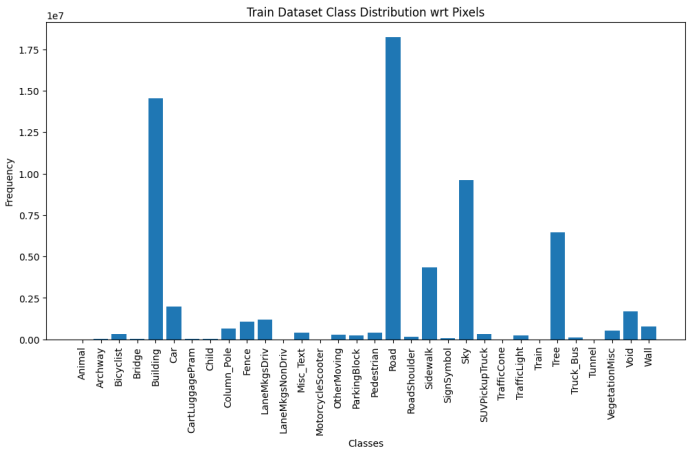
In this task, I use the "CAMVid Dataset " for image segmentation, where each image has a corresponding label image with color-coded classes. The dataset consists of **32 classes** and has a total of **369 images** in train set and **232 images** in test set.

(a) I implemented a custom dataset class, `CamVidDataset`, using PyTorch's Dataset module. The input images are then normalized using the **standard ImageNet mean [0.485, 0.456, 0.406] and standard deviation [0.229, 0.224, 0.225]**. The label images, which are color-segmented representations of different classes, are **converted into numerical labels** using a predefined color_to_label mapping. A screenshot of the `CamVidDataset` class is shown below.

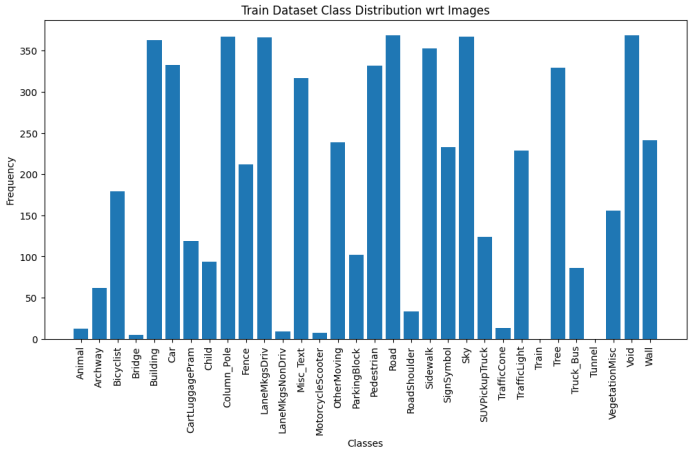
```python
class CamVidDataset(Dataset):
    """
    Custom Dataset for CamVid Image Segmentation
    """
    def __init__(self, img_dir, labels_dir, transform=None, label_transform=None):
        self.img_dir = img_dir
        self.labels_dir = labels_dir
        self.transform = transform
        self.label_transform = label_transform
        self.img_names = os.listdir(img_dir)
        self.images = []
        self.labels = []
        self.targets = []

        for img_name in self.img_names:
            img_path = os.path.join(self.img_dir, img_name)
            label_path = os.path.join(self.labels_dir, img_name.replace(".png", "_L.png"))
            image = Image.open(img_path)
            label = Image.open(label_path)
            if self.transform:
                image = self.transform(image)
            if self.label_transform:
                label = self.label_transform(label)

            self.images.append(image)
            self.labels.append(label)

            label_reshaped = label.permute(1, 2, 0)
            color_reshaped = (label_reshaped * 255).byte()
            h, w, _ = color_reshaped.shape
            color_flat = color_reshaped.view(-1, 3).tolist()
            mapped = [color_to_label.get(tuple(c), 30) for c in color_flat]
            target = torch.tensor(mapped, dtype=torch.int64).view(h, w)

            self.targets.append(target)

        print(len(self.images), len(self.labels), len(self.targets))

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

    def __getitem__(self, idx):
        return self.images[idx], self.labels[idx], self.targets[idx]
```

Figure 24: CamVidDataset Custom Dataset

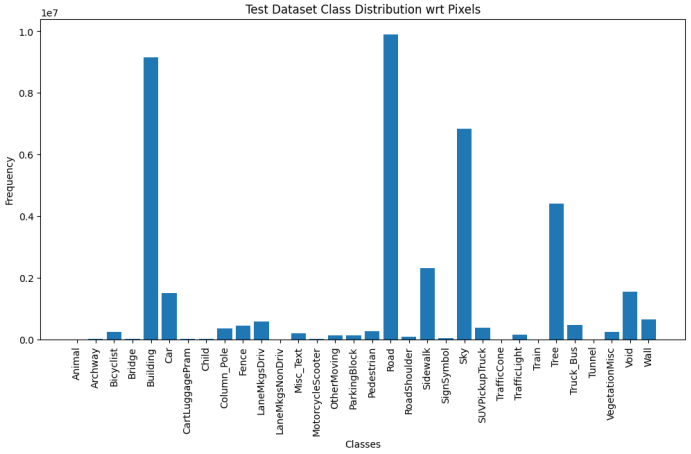
(b) To analyze the class distribution in the CAMVid dataset, I visualized the frequency of each class in both the training and test sets using bar charts, considering **both pixel count and image count**. The **Road class has the highest pixel count** in both datasets (18.2M in train, 9.9M in test), while **Train and Tunnel classes are absent**. In terms of image count, **Road and Void** appear in the all training images (369), while Column_Pole, Road, Sky, and Void appear in all the test images (232).The generated distribution plots were logged using **Weights & Biases (WandB)** and are shown below.



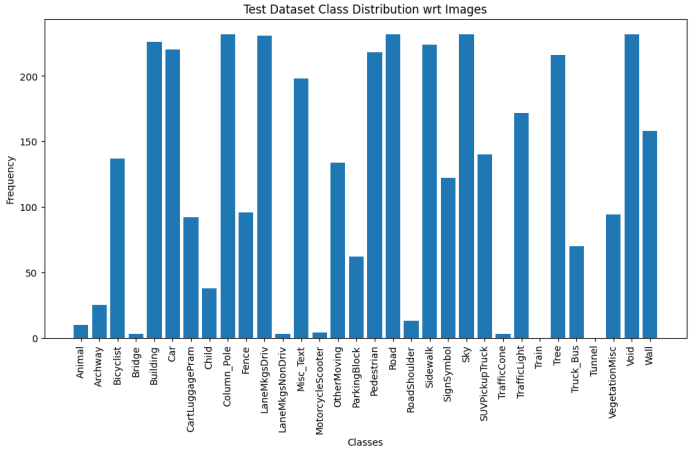
(a) Train Dataset Class Distribution wrt Pixels



(b) Train Dataset Class Distribution wrt Images



(c) Test Dataset Class Distribution wrt Pixels



(d) Test Dataset Class Distribution wrt Images

Figure 25: Training and Testing dataset Class Distribution

(c) I visualized two images for each class, highlighting only the respective class segment in its corresponding color while masking the rest of the image in black. The original images were denormalized for visualization purposes. **Train and Tunnel** classes were not visualized because they are absent in the dataset. The visualizations are shown below.

Class: Archway

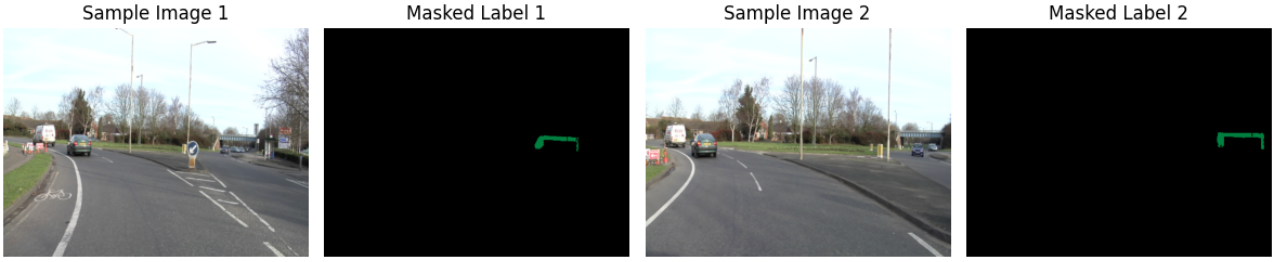
Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2

Class: Bicyclist

Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2

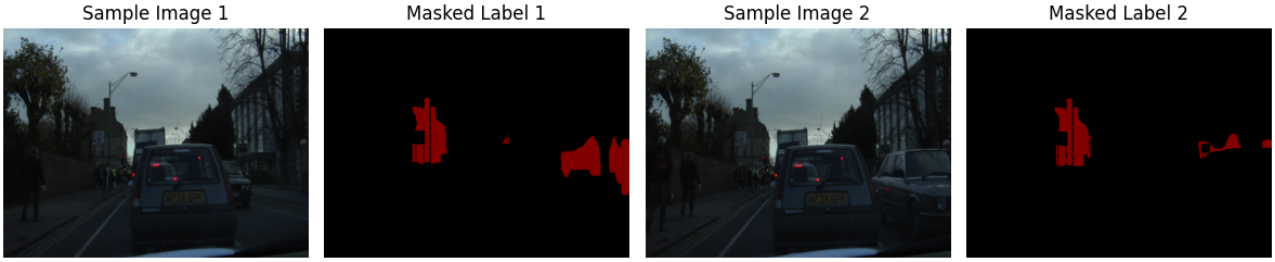
Class: Bridge

Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2



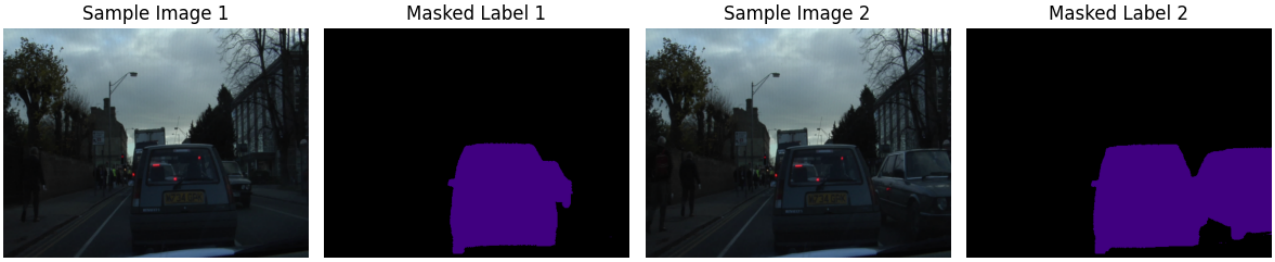
Class: Building

Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2



Class: Car

Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2



Class: CartLuggagePram

Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2

Class: Child

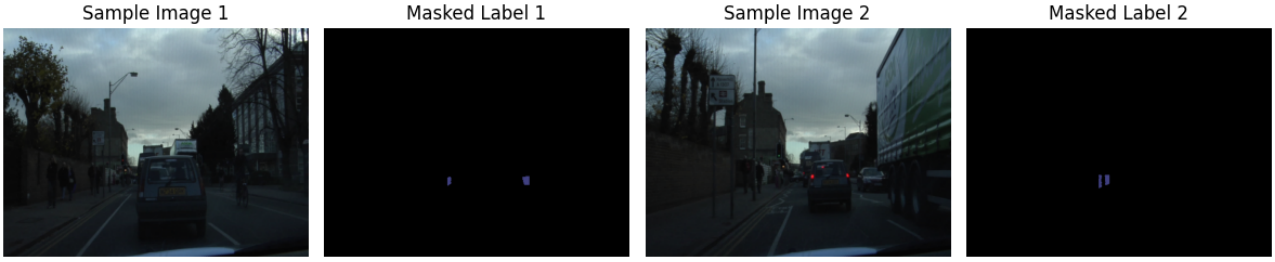
Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2

Class: Column_Pole

Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2

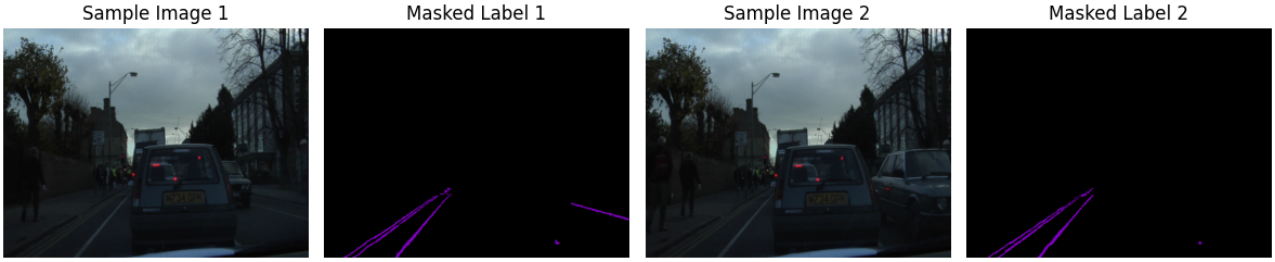
Class: Fence

Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2



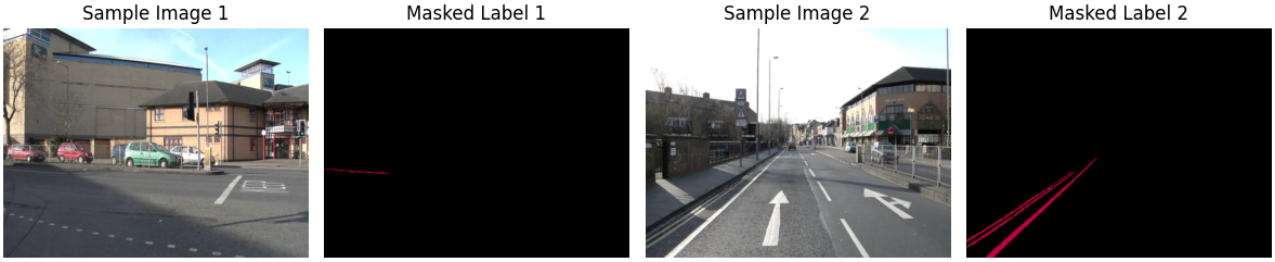
Class: LaneMkgsDriv

Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2



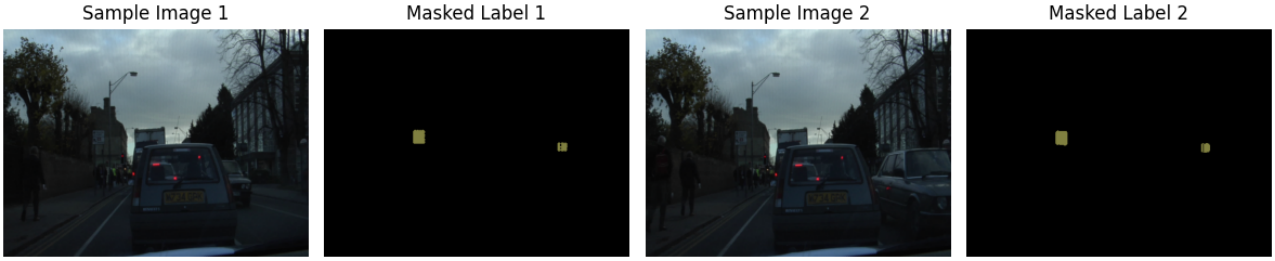
Class: LaneMkgsNonDriv

Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2



Class: Misc_Text

Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2

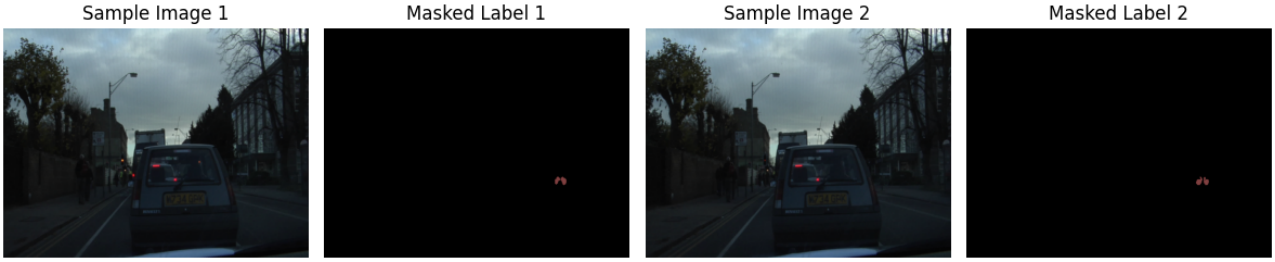


Class: MotorcycleScooter

Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2

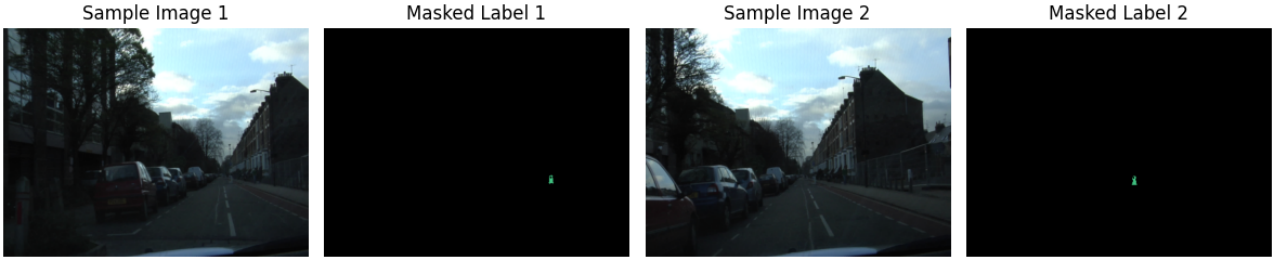
Class: OtherMoving

Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2



Class: ParkingBlock

Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2

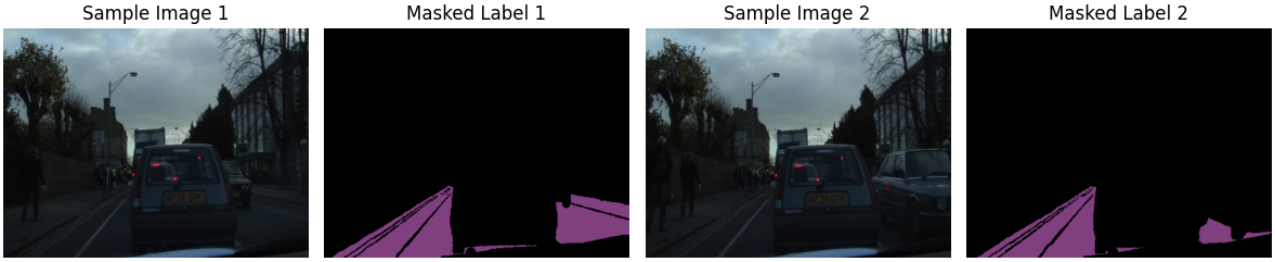


Class: Pedestrian

Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2

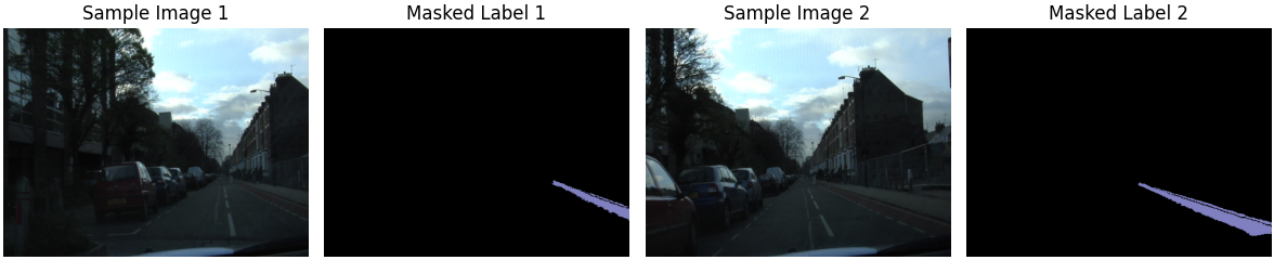
Class: Road

Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2



Class: RoadShoulder

Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2



Class: Sidewalk

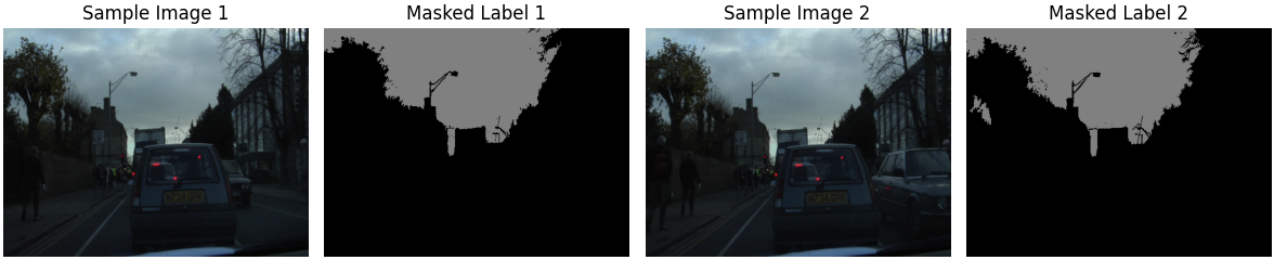
Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2

Class: SignSymbol

Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2

Class: Sky

Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2



Class: SUVPickupTruck

Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2

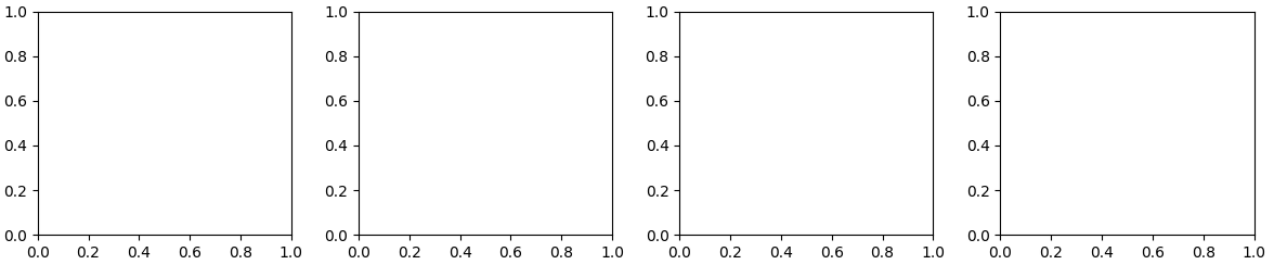
Class: TrafficCone

Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2

Class: TrafficLight

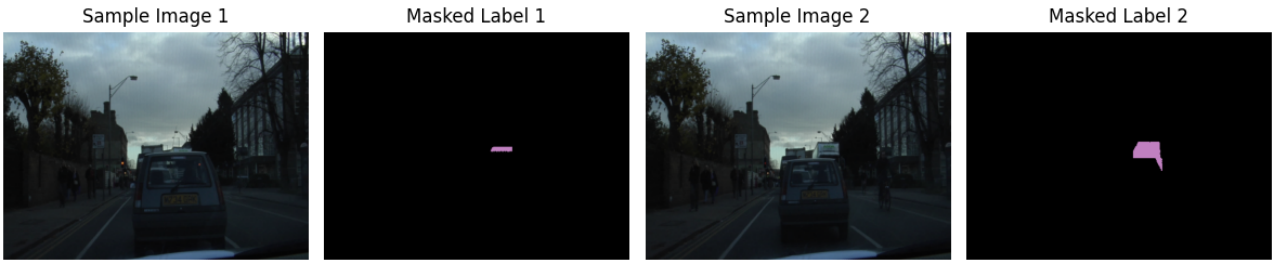
Sample Image 1 | Masked Label 1 | Sample Image 2 | Masked Label 2

Class: Train

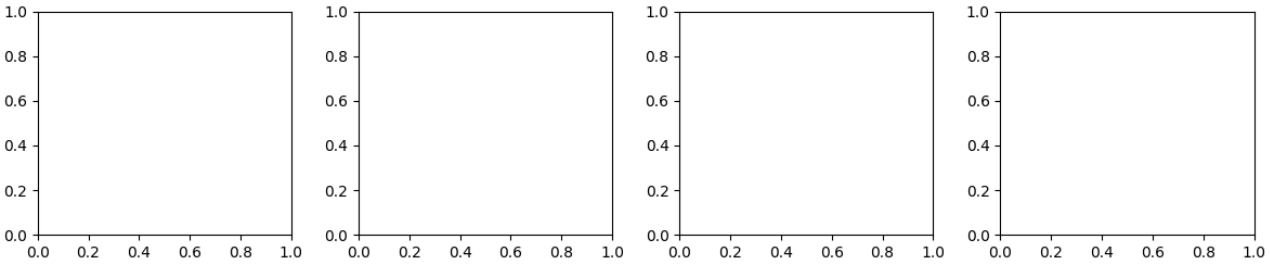


Class: Tree

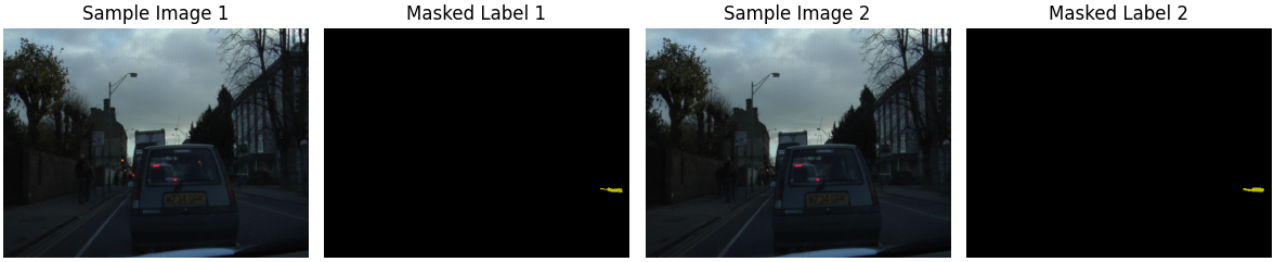
Class: Truck_Bus



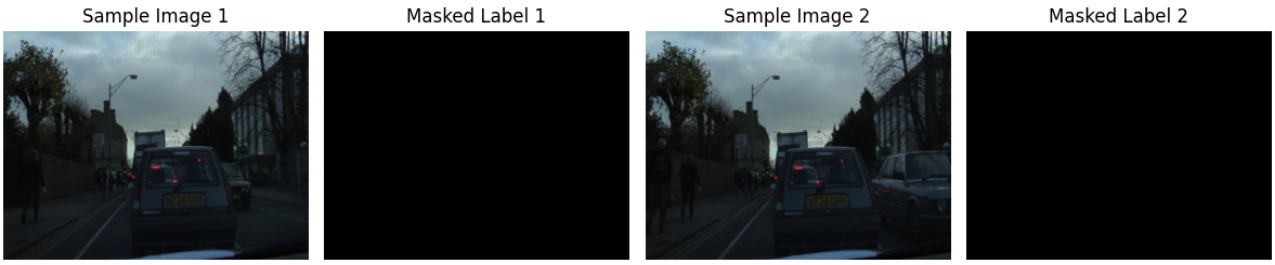
Class: Tunnel



Class: VegetationMisc



Class: Void



Class: Wall

Sample Image 1

Masked Label 1

Sample Image 2

Masked Label 2

Part 2. Training SegNet Decoder from scratch

In this task, the `SegNet_Decoder` is implemented from scratch and trained using a pre-trained `SegNet_Encoder` on the CamVid Dataset.

(a) The decoder was completed as per the provided skeleton and trained using cross-entropy loss, Batch Normalization momentum of 0.5, and the Adam optimizer. The training loss was logged using **Weights & Biases (wandb)** for monitoring. The loss plot is shown below.

Figure 33: Training loss for training SegNet_Decoder

(b) In this task, I evaluated the performance of a **trained SegNet decoder** on the test set using various segmentation metrics, including **pixelwise accuracy, IoU, mIoU, Dice coefficient, precision, and recall**.

The test images and their corresponding ground truth masks were processed in batches using a PyTorch dataloader. The predictions were obtained from the model's logits using argmax, and **each metric was computed per class**.

Classes were **grouped into bins of size 0.1 based on their IoU scores**, and **average precision and recall were computed for each bin**. A bar chart was plotted to visualize these trends, to analyze model performance at different segmentation quality levels. The evaluation report on test set is shown below.

- **Test Loss:** 0.8569
- **Mean IoU (mIoU):** 0.2409

Pixelwise Accuracy per Class:
Animal: 0.0000; Archway: 0.0000; Bicyclist: 0.3824; Bridge: 0.0000; Building: 0.9062; Car: 0.9210; CartLuggagePram: 0.0000; Child: 0.0000; Column Pole: 0.1620; Fence: 0.3668; LaneMkgsDriv: 0.5427; LaneMkgsNonDriv: 0.0000; Misc Text: 0.0701; MotorcycleScooter: 0.0000; OtherMoving: 0.2019; ParkingBlock: 0.2126; Pedestrian: 0.1459; Road: 0.9508; RoadShoulder: 0.2322; Sidewalk: 0.7888; SignSymbol: 0.0185; Sky: 0.9732; SUVPickupTruck: 0.0340; TrafficCone: 0.0000; TrafficLight: 0.3602; Train: No pixels in dataset; Tree: 0.7542; Truck Bus: 0.0052; Tunnel: No pixels in dataset; VegetationMisc: 0.4395; Void: 0.5148; Wall: 0.2406.

IoU per Class:
Animal: 0.0000; Archway: 0.0000; Bicyclist: 0.2915; Bridge: 0.0000; Building: 0.7429; Car: 0.6070; CartLuggagePram: 0.0000; Child: 0.0000; Column Pole: 0.1187; Fence: 0.2733; LaneMkgsDriv: 0.4761; LaneMkgsNonDriv: 0.0000; Misc Text: 0.0433; MotorcycleScooter: 0.0000; OtherMoving: 0.1685; ParkingBlock: 0.1212; Pedestrian: 0.1058; Road: 0.8898; RoadShoulder: 0.1582; Sidewalk: 0.6798; SignSymbol: 0.0165; Sky: 0.8991; SUVPickupTruck: 0.0291; TrafficCone: 0.0000; TrafficLight: 0.1785; Train: No pixels in dataset; Tree: 0.6568; Truck Bus: 0.0052; Tunnel: No pixels in dataset; VegetationMisc: 0.2087; Void: 0.3255; Wall: 0.2307.

Dice Coefficient per Class:
Animal: 0.0000; Archway: 0.0000; Bicyclist: 0.4514; Bridge: 0.0000; Building: 0.8525; Car: 0.7554; CartLuggagePram: 0.0000; Child: 0.0000; Column Pole: 0.2122; Fence: 0.4293; LaneMkgsDriv: 0.6451; LaneMkgsNonDriv: 0.0000; Misc Text: 0.0830; MotorcycleScooter: 0.0000; OtherMoving: 0.2884; ParkingBlock: 0.2162; Pedestrian: 0.1913; Road: 0.9417; RoadShoulder: 0.2732; Sidewalk: 0.8094; SignSymbol: 0.0324; Sky: 0.9469; SUVPickupTruck: 0.0566; TrafficCone: 0.0000; TrafficLight: 0.3029; Train: No pixels in dataset; Tree: 0.7929; Truck Bus: 0.0103; Tunnel: No pixels in dataset; VegetationMisc: 0.3453; Void: 0.4912; Wall: 0.3749.

Precision per Class:
Animal: No pixels in dataset; Archway: No pixels in dataset; Bicyclist: 0.5506; Bridge: No pixels in dataset; Building: 0.8048; Car: 0.6403; CartLuggagePram: No pixels in dataset; Child: No pixels in dataset; Column Pole: 0.3075; Fence: 0.5173; LaneMkgsDriv: 0.7952; LaneMkgsNonDriv: No pixels in dataset; Misc Text: 0.1019; MotorcycleScooter: No pixels in dataset; OtherMoving: 0.5048; ParkingBlock: 0.2198; Pedestrian: 0.2777; Road: 0.9327; RoadShoulder: 0.3317; Sidewalk: 0.8310; SignSymbol: 0.1317; Sky: 0.9219; SUVPickupTruck: 0.1684; TrafficCone: No pixels in dataset; TrafficLight: 0.2614; Train: No pixels in dataset; Tree: 0.8357; Truck Bus: 0.3729; Tunnel: No pixels in dataset; VegetationMisc: 0.2844; Void: 0.4696; Wall: 0.8482.

Recall per Class:
Animal: 0.0000; Archway: 0.0000; Bicyclist: 0.3824; Bridge: 0.0000; Building: 0.9062; Car: 0.9210; CartLuggagePram: 0.0000; Child: 0.0000; Column Pole: 0.1620; Fence: 0.3668; LaneMkgsDriv: 0.5427; LaneMkgsNonDriv: 0.0000; Misc Text: 0.0701; MotorcycleScooter: 0.0000; OtherMoving: 0.2019; ParkingBlock: 0.2126; Pedestrian: 0.1459; Road: 0.9508; RoadShoulder: 0.2322; Sidewalk: 0.7888; SignSymbol: 0.0185; Sky: 0.9732; SUVPickupTruck: 0.0340; TrafficCone: 0.0000; TrafficLight: 0.3602; Train: No pixels in dataset; Tree: 0.7542; Truck Bus: 0.0052; Tunnel: No pixels in dataset; VegetationMisc: 0.4395; Void: 0.5148; Wall: 0.2406.

IoU Distribution

- **IoU Bin 0.0-0.1:** 12 classes
- **IoU Bin 0.1-0.2:** 6 classes
- **IoU Bin 0.2-0.3:** 4 classes
- **IoU Bin 0.3-0.4:** 1 class
- **IoU Bin 0.4-0.5:** 1 class
- **IoU Bin 0.5-0.6:** No classes
- **IoU Bin 0.6-0.7:** 3 classes
- **IoU Bin 0.7-0.8:** 1 class
- **IoU Bin 0.8-0.9:** 2 classes

Precision per IoU Bin:

- (0.0, 0.1): 0.1937
- (0.1, 0.2): 0.3172
- (0.2, 0.3): 0.5501
- (0.3, 0.4): 0.4696
- (0.4, 0.5): 0.7952
- (0.5, 0.6): No classes in this range
- (0.6, 0.7): 0.7690
- (0.7, 0.8): 0.8048
- (0.8, 0.9): 0.9273

Recall per IoU Bin:

- (0.0, 0.1): 0.0106
- (0.1, 0.2): 0.2191
- (0.2, 0.3): 0.3573
- (0.3, 0.4): 0.5148
- (0.4, 0.5): 0.5427
- (0.5, 0.6): No classes in this range
- (0.6, 0.7): 0.8213
- (0.7, 0.8): 0.9062
- (0.8, 0.9): 0.9620

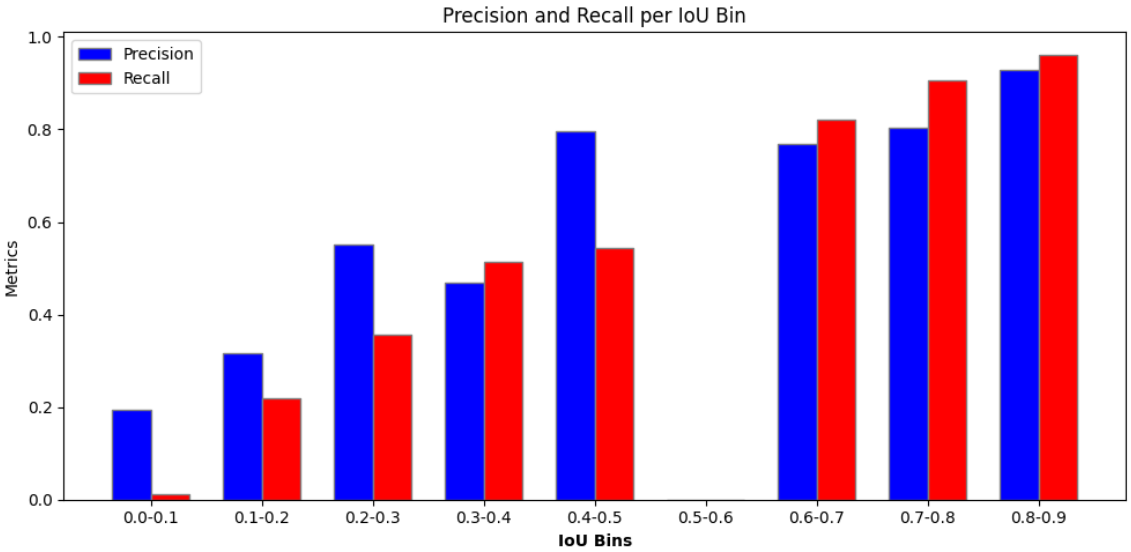
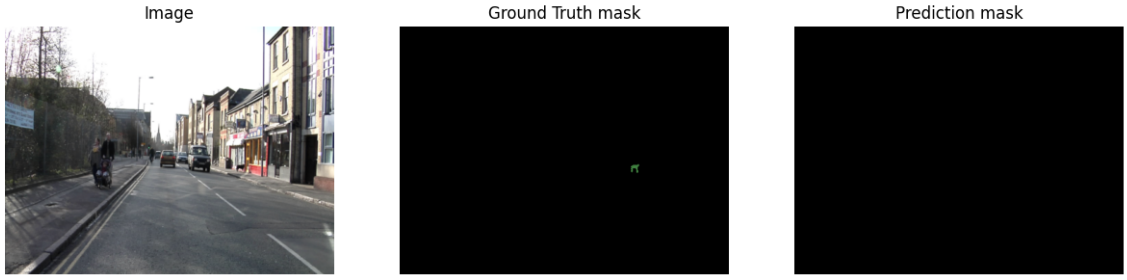


Figure 34: Precision & Recall per IoU Bin

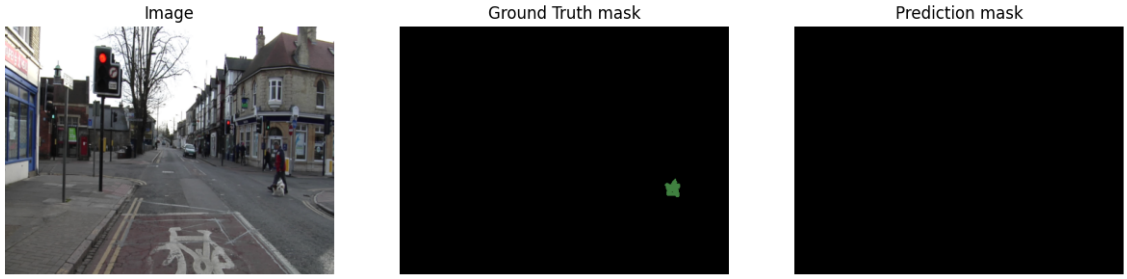
Precision and recall varied across IoU bins, with higher values in the 0.7–1.0 range, confirming that higher-quality segmentations yield better precision and recall.

(c) In this task, I analyzed three classes with **IoU $\leq$ 0.5** by visualizing their predicted and ground truth masks on three test images each. The visualizations highlighted only the target class in its respective color, while other regions were masked black for clarity.

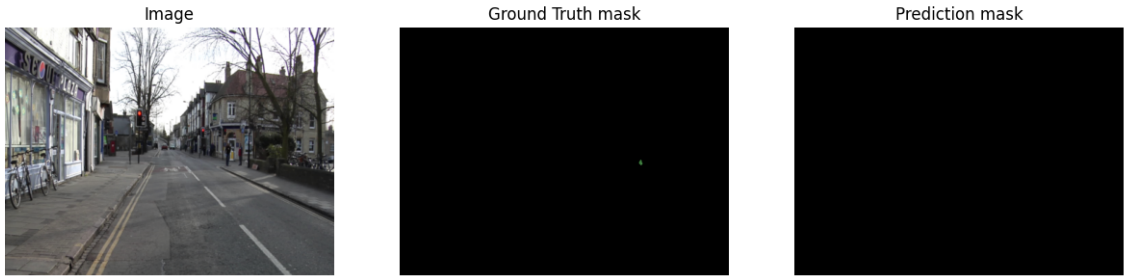
Class Animal Image 1



Class Animal Image 2



Class Animal Image 3



Class Archway Image 1

Image

Ground Truth mask

Prediction mask

Class Archway Image 2

Image

Ground Truth mask

Prediction mask

Class Archway Image 3

Image

Ground Truth mask

Prediction mask

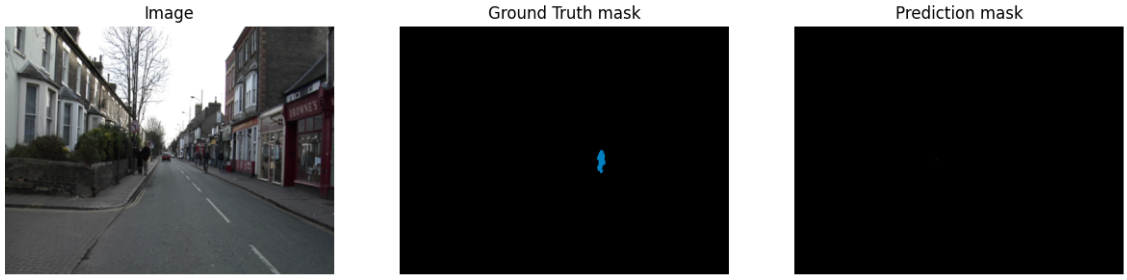
Class Bicyclist Image 1

Image

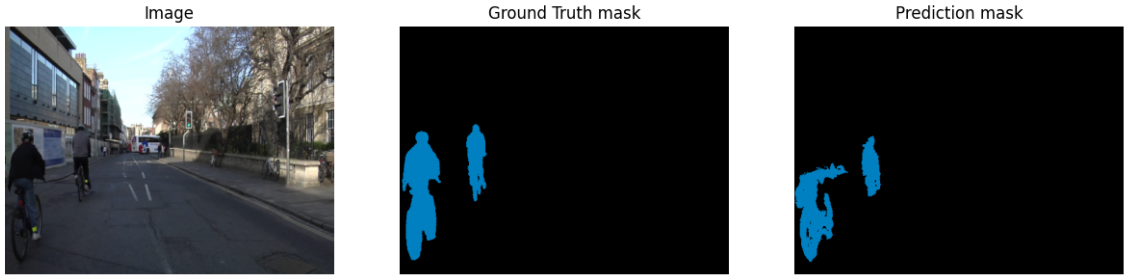
Ground Truth mask

Prediction mask

Class Bicyclist Image 2



Class Bicyclist Image 3



The model exhibits significant challenges in accurately segmenting objects from the “animal ”, “archway” and “bicyclist ” classes, as evidenced by IoU values ≤ 0.5 across all provided images. The model’s poor performance in segmenting animals can be attributed to their small size as observed in Ground Truth mask of animal images, which makes them challenging to detect and classify accurately. Archways are often missed or misclassified, likely because of their integration into complex urban environments or being partially occluded by surrounding structures or misclassified as other classes like buildings. Bicyclists are inconsistently identified, potentially due to partial occlusions or similarities with pedestrians.

In conclusion, the model performed well on major classes with distinct features but struggled with smaller or ambiguous objects.

Part 3. Fine-tuning DeepLabV3 on the CamVID Dataset

In this task, fine-tuned the `DeepLabV3` model pretrained on the Pascal VOC Dataset on the CamVid Dataset.

(a) The DeepLabV3 model was fine-tuned using cross-entropy loss and the Adam optimizer. The training loss was logged using **Weights & Biases (wandb)** for monitoring. The loss plot is shown below.

Figure 38: Training loss for training DeepLabV3

(b) In this task, I evaluated the performance of a **fine-tuned DeepLabV3** on the test set using various segmentation metrics, including **pixelwise accuracy, IoU, mIoU, Dice coefficient, precision, and recall**.

The test images and their corresponding ground truth masks were processed in batches using a PyTorch dataloader. The predictions were obtained from the model's logits using argmax, and **each metric was computed per class**.

Classes were **grouped into bins of size 0.1 based on their IoU scores**, and **average precision and recall were computed for each bin**. A bar chart was plotted to visualize these trends, to analyze model performance at different segmentation quality levels. The evaluation report on test set is shown below.

- **Test Loss:** 0.4136
- **Mean IoU (mIoU):** 0.4151

Pixelwise Accuracy per Class:
Animal: 0.0000; Archway: 0.0000; Bicyclist: 0.4812; Bridge: 0.4898; Building: 0.9306; Car: 0.9306; CartLuggagePram: 0.0000; Child: 0.2133; Column_Pole: 0.2495; Fence: 0.6161; LaneMkgsDriv: 0.4516; LaneMkgsNonDriv: 0.0000; Misc_Text: 0.3496; MotorcycleScooter: 0.0000; OtherMoving: 0.5127; ParkingBlock: 0.4564; Pedestrian: 0.6917; Road: 0.9712; RoadShoulder: 0.7823; Sidewalk: 0.9061; SignSymbol: 0.3910; Sky: 0.9710; SUVPickupTruck: 0.4171; TrafficCone: 0.0000; TrafficLight: 0.5235; Train: No pixels in dataset; Tree: 0.8510; Truck_Bus: 0.5666; Tunnel: No pixels in dataset; VegetationMisc: 0.6612; Void: 0.6360; Wall: 0.6845.

IoU per Class:
Animal: 0.0000; Archway: 0.0000; Bicyclist: 0.4562; Bridge: 0.4351; Building: 0.8402; Car: 0.7972; CartLuggagePram: 0.0000; Child: 0.1942; Column_Pole: 0.1928; Fence: 0.5328; LaneMkgsDriv: 0.3918; LaneMkgsNonDriv: 0.0000; Misc_Text: 0.3106; MotorcycleScooter: 0.0000; OtherMoving: 0.3939; ParkingBlock: 0.4172; Pedestrian: 0.3830; Road: 0.9145; RoadShoulder: 0.5690; Sidewalk: 0.8045; SignSymbol: 0.3386; Sky: 0.9128; SUVPickupTruck: 0.3404; TrafficCone: 0.0000; TrafficLight: 0.4858; Train: No pixels in dataset; Tree: 0.7711; Truck_Bus: 0.4200; Tunnel: No pixels in dataset; VegetationMisc: 0.5510; Void: 0.4478; Wall: 0.5519.

Dice Coefficient per Class:
Animal: 0.0000; Archway: 0.0000; Bicyclist: 0.6265; Bridge: 0.6064; Building: 0.9132; Car: 0.8872; CartLuggagePram: 0.0000; Child: 0.3252; Column_Pole: 0.3233; Fence: 0.6952; LaneMkgsDriv: 0.5631; LaneMkgsNonDriv: 0.0000; Misc_Text: 0.4740; MotorcycleScooter: 0.0000; OtherMoving: 0.5651; ParkingBlock: 0.5888; Pedestrian: 0.5539; Road: 0.9554; RoadShoulder: 0.7253; Sidewalk: 0.8917; SignSymbol: 0.5059; Sky: 0.9544; SUVPickupTruck: 0.5079; TrafficCone: 0.0000; TrafficLight: 0.6540; Train: No pixels in dataset; Tree: 0.8707; Truck_Bus: 0.5916; Tunnel: No pixels in dataset; VegetationMisc: 0.7105; Void: 0.6186; Wall: 0.7113.

Precision per Class:
Animal: No pixels in dataset; Archway: 0.0000; Bicyclist: 0.8975; Bridge: 0.7958; Building: 0.8964; Car: 0.8477; CartLuggagePram: 0.0000; Child: 0.6844; Column_Pole: 0.4589; Fence: 0.7977; LaneMkgsDriv: 0.7476; LaneMkgsNonDriv: No pixels in dataset; Misc_Text: 0.7357; MotorcycleScooter: No pixels in dataset; OtherMoving: 0.6296; ParkingBlock: 0.8291; Pedestrian: 0.4618; Road: 0.9400; RoadShoulder: 0.6761; Sidewalk: 0.8777; SignSymbol: 0.7162; Sky: 0.9384; SUVPickupTruck: 0.6492; TrafficCone: No pixels in dataset; TrafficLight: 0.8711; Train: No pixels in dataset; Tree: 0.8915; Truck_Bus: 0.6188; Tunnel: No pixels in dataset; VegetationMisc: 0.7677; Void: 0.6021; Wall: 0.7402.

Recall per Class:
Animal: 0.0000; Archway: 0.0000; Bicyclist: 0.4812; Bridge: 0.4898; Building: 0.9306; Car: 0.9306; CartLuggagePram: 0.0000; Child: 0.2133; Column_Pole: 0.2495; Fence: 0.6161; LaneMkgsDriv: 0.4516; LaneMkgsNonDriv: 0.0000; Misc_Text: 0.3496; MotorcycleScooter: 0.0000; OtherMoving: 0.5127; ParkingBlock: 0.4564; Pedestrian: 0.6917; Road: 0.9712; RoadShoulder: 0.7823; Sidewalk: 0.9061; SignSymbol: 0.3910; Sky: 0.9710; SUVPickupTruck: 0.4171; TrafficCone: 0.0000; TrafficLight: 0.5235; Train: No pixels in dataset; Tree: 0.8510; Truck_Bus: 0.5666; Tunnel: No pixels in dataset; VegetationMisc: 0.6612; Void: 0.6360; Wall: 0.6845.

IoU Distribution:

- **IoU Bin 0.0-0.1:** 6 classes
- **IoU Bin 0.1-0.2:** 2 classes
- **IoU Bin 0.2-0.3:** No classes
- **IoU Bin 0.3-0.4:** 6 classes
- **IoU Bin 0.4-0.5:** 6 classes
- **IoU Bin 0.5-0.6:** 4 classes
- **IoU Bin 0.6-0.7:** No classes
- **IoU Bin 0.7-0.8:** 2 classes
- **IoU Bin 0.8-0.9:** 2 classes
- **IoU Bin 0.9-1.0:** 2 classes

Precision per IoU Bin:

- (0.0, 0.1): 0.0000
- (0.1, 0.2): 0.5716
- (0.2, 0.3): No classes in this range
- (0.3, 0.4): 0.6567
- (0.4, 0.5): 0.7691
- (0.5, 0.6): 0.7454
- (0.6, 0.7): No classes in this range
- (0.7, 0.8): 0.8696
- (0.8, 0.9): 0.8870
- (0.9, 1.0): 0.9392

Recall per IoU Bin:

- (0.0, 0.1): 0.0000
- (0.1, 0.2): 0.2314
- (0.2, 0.3): No classes in this range
- (0.3, 0.4): 0.4689
- (0.4, 0.5): 0.5256
- (0.5, 0.6): 0.6860
- (0.6, 0.7): No classes in this range
- (0.7, 0.8): 0.8908
- (0.8, 0.9): 0.9183
- (0.9, 1.0): 0.9711

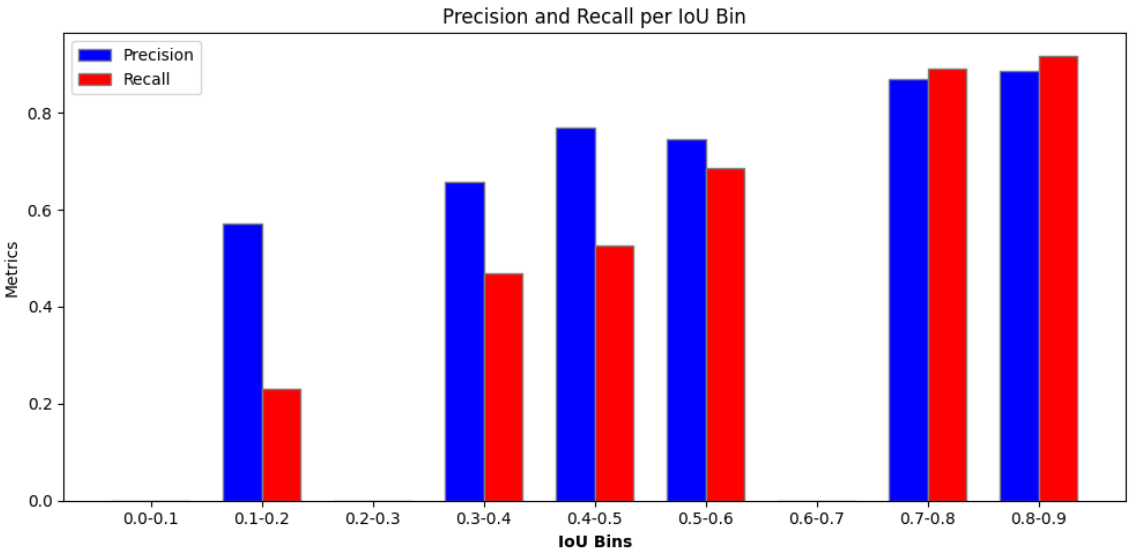
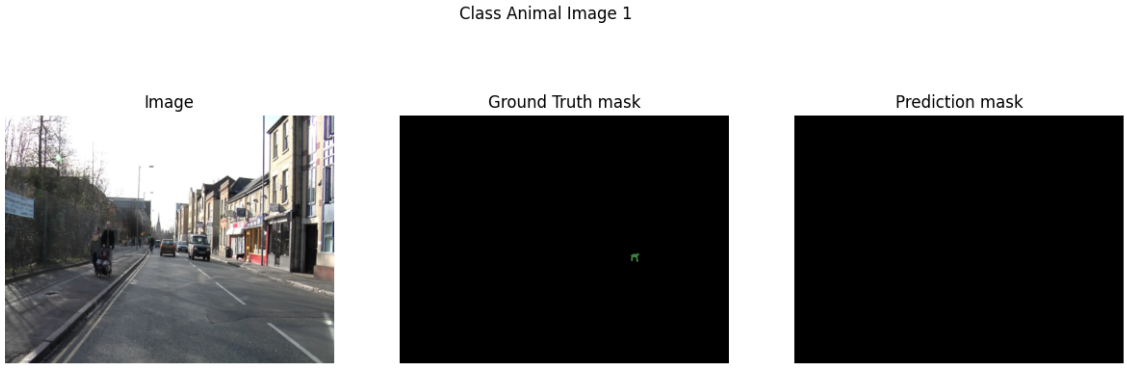
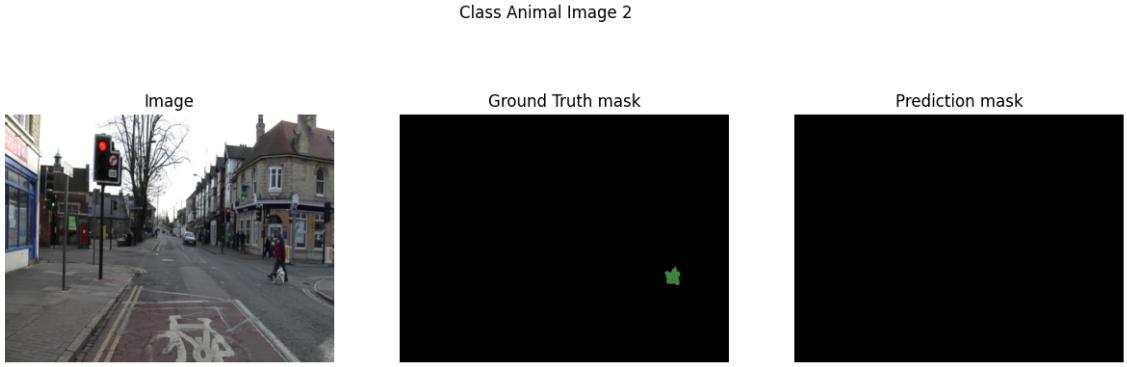


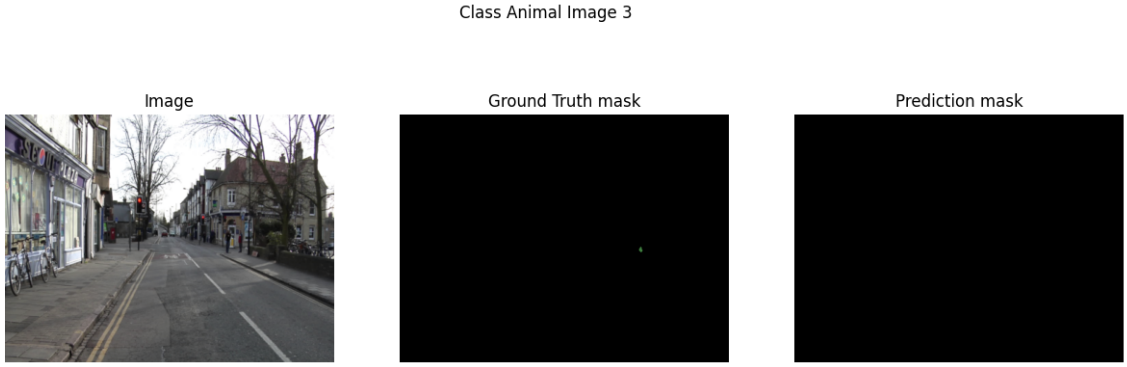
Figure 39: Precision & Recall per IoU Bin

The DeepLabV3 model trained on the CamVID dataset performs relatively well, as seen in the mIoU of 0.4151, compared to the SegNet model. While both models show high performance in some classes like "Road" (0.9145 IoU in DeepLabV3), DeepLabV3 consistently outperforms SegNet in several classes such as "Building", "Car", and "Sky". These are critical to accurate urban scene segmentation. However, the SegNet model outperforms in certain edge cases like smaller object classes. The key difference appears in the precision and recall for classes like "RoadShoulder " and "TrafficCone", where DeepLabV3 seems to struggle more with recall than SegNet. DeepLabV3 benefits from its atrous convolutions, allowing better feature extraction at multiple scales, leading to higher accuracy in complex scenes.

(c) In this task, I analyzed three classes with **IoU $\leq$ 0.5** by visualizing their predicted and ground truth masks on three test images each. The visualizations highlighted only the target class in its respective color, while other regions were masked black for clarity.







Class Archway Image 1

Image

Ground Truth mask

Prediction mask

Class Archway Image 2

Image

Ground Truth mask

Prediction mask

Class Archway Image 3

Image

Ground Truth mask

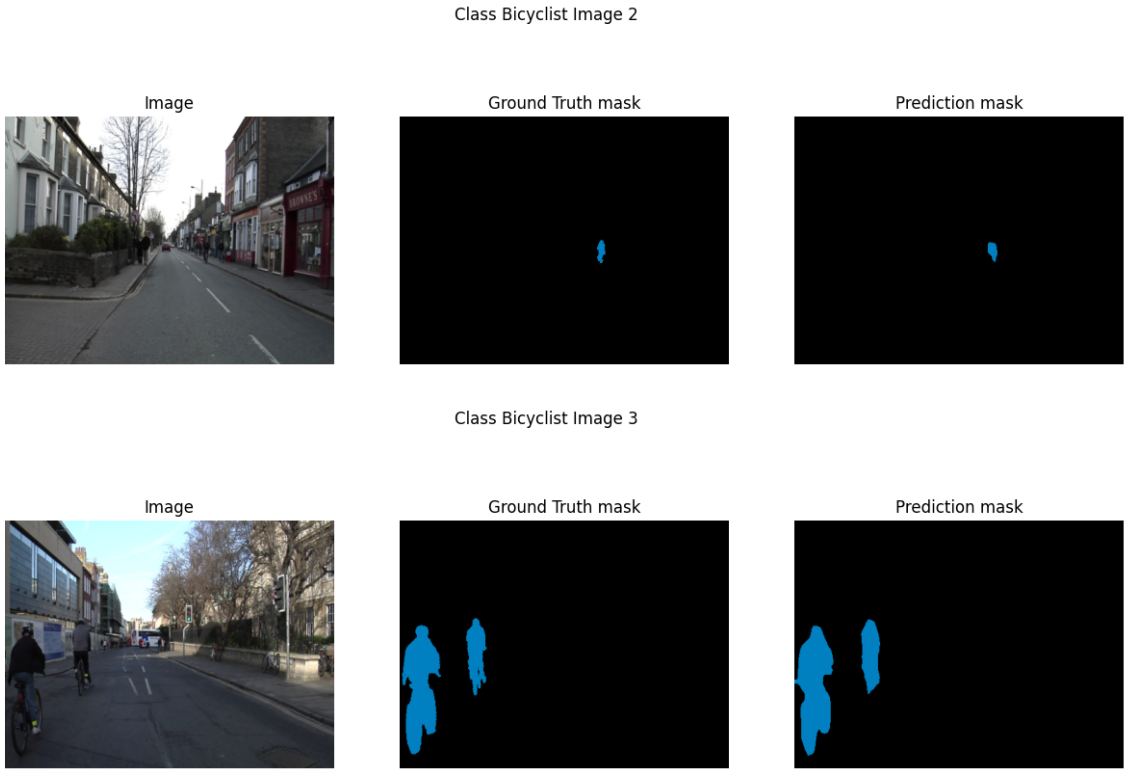
Prediction mask

Class Bicyclist Image 1

Image

Ground Truth mask

Prediction mask



The model's performance in segmenting the "animal ", "archway" and "bicyclist " classes, as evidenced by low IoU values and visualizations, reveals several common challenges. In all three cases, objects are often partially occluded or blend into complex urban backgrounds, making them difficult to detect accurately. The small size of animals and the archways' structural complexity and varying sizes, along with the varying positions may be lead to misclassification, as the model might struggle to distinguish them from other objects or background elements, and various postures of bicyclists, further complicates the segmentation task. The model may struggle with generalizing across different scenarios due to limited training data diversity.

In conclusion, the DeepLabV3 model achieved higher accuracy and better segmentation consistency than SegNet model, excelling in major classes but still facing challenges with small or occluded objects.

References

- For the training loop, I referred to my own implementation from `DL Assignment 1, Question 2(c)`, where training loops for MLP and CNN models were designed for the FashionMNIST dataset. [GitHub Repository]
- **Evaluating Image Segmentation Models**: link
- **Pixelwise Accuracy**: Section 12.6.1.2 (*Pixel Accuracy*) from the chapter *Semantic Scene Segmentation for Robotics* in the book *Deep Learning for Robot Perception and Cognition*: link
- **Dice Coefficient Metric**: link
- **Guide to Image Segmentation in Computer Vision**: link
- **DeepLabV3 Pretrained Weights**: Official PyTorch documentation for DeepLabV3 pretrained weights: link

Question 4: Object Detection and Multi-Object Tracking

Part 1. Object Detection

This task involves utilizing the "Ultralytics API" to detect objects in the "COCO 2017 validation set". The task involves downloading the dataset, running detection models, and doing in-depth error analysis of the predictions.

(a) I used the **Ultralytics API** to download the **COCO 2017 validation dataset**. The script fetches the necessary label files and validation images, which will be used for further object detection tasks.

(b) In this task, I used a **COCO-pretrained YOLOv8 model** to make predictions on the **COCO val2017 dataset**. The predictions were then saved in the standard COCO format in `coco_predictions.json`, where each entry contained the `image_id`, `category_id`, `bbox`, and `confidence score`. I correctly mapped category IDs using COCO's provided category annotations to ensure accurate label assignment in the predictions. The results were evaluated against the ground truth annotations using COCO's evaluation API, computing the Mean Average Precision (mAP).

The **mAP @ IoU=0.50:0.95** (main metric) achieved was **0.480**, while **mAP @ IoU=0.50** was **0.616**. A total of 34,600 predictions were generated, covering small, medium, and large object categories. The statistics of the evaluations are shown below.

```
Accumulating evaluation results...
DONE (t=2.46s).
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.480
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.616
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.524
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.285
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.534
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.667
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.363
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.539
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.546
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.320
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.599
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.742
```

Figure 40: COCO Eval statistics (mAP)

(c) I used the **TIDE Toolbox** to analyze my model's predictions on COCO val2017.

Error Analysis:
The TIDE analysis of my model's predictions reveals that the model achieves a **bbox AP@50 of 61.61**, indicating decent overall performance. However, the error breakdown shows significant room for improvement. The **missed ground truth errors (Miss: 21.65 dAP)** dominate, meaning the model fails to detect many objects present in the ground truth, severely impacting recall. **Localization errors (Loc: 4.87 dAP)** are the second-largest issue, indicating inaccuracies in bounding box predictions. **Classification errors (Cls: 2.37 dAP)** and **background errors (Bkg: 1.59 dAP)** are relatively minor but still contribute to performance degradation. The **false negatives (FN: 29.27 dAP)** further highlight the recall problem, while **false positives (FP: 5.18 dAP)** suggest some overconfidence in predictions.

In summary, the model struggles most with **recall (missed objects)** and **localization accuracy** The Error Analysis reports & plots are shown below.

```
-- coco_predictions --

bbox AP @ 50: 61.61

                    Main Errors
============================================================
  Type      Cls      Loc      Both     Dupe      Bkg      Miss
------------------------------------------------------------
   dAP     2.37     4.87     0.57     0.13     1.59     21.65
============================================================

       Special Error
============================
  Type   FalsePos   FalseNeg
----------------------------
   dAP       5.18      29.27
============================
```

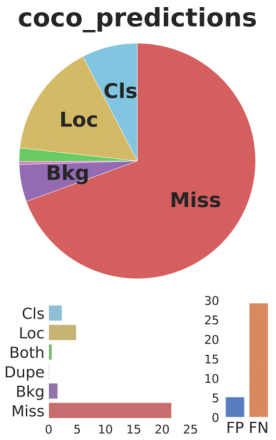


Figure 41: TIDE Analysis Metrics

(d) I computed the **Expected Calibration Error (ECE)** to assess the confidence calibration of my model's predictions on COCO val2017. Predictions were grouped by (image_id, category_id), sorted by confidence, and the top N predictions (where N is the number of ground truth objects for that category in the image) were marked as correct, while extra predictions were considered incorrect. Confidence values and correctness labels were binned into M = 10 intervals, and ECE was computed using the given formula in section 2 of the said paper.

Analysis: The obtained **ECE = 0.1598** suggests a moderate miscalibration, meaning the model's predicted confidence scores are not perfectly aligned with actual correctness. This indicates a slight overconfidence or underconfidence in predictions.

(e) In this task, I evaluated object detection performance across small, medium, and large object scales. Both the ground truth and predictions are filtered based on COCO's scale definitions, ensuring an accurate evaluation.

- **Expected Calibration Error (ECE) for Small Objects:** 0.2383
- **Expected Calibration Error (ECE) for Medium Objects:** 0.1006
- **Expected Calibration Error (ECE) for Large Objects:** 0.0447

Error Analysis:
The TIDE and ECE metrics reveal significant performance disparities across object scales. **Small objects** show the worst performance, with a low AP of 32.10, high missed GT error (36.31 dAP), and poor calibration (ECE: 0.2383), indicating severe recall and overconfidence issues. **Medium objects** perform better (AP: 58.26, Miss: 22.63 dAP, ECE: 0.1006), but still struggle with calibration. **Large objects** excel, achieving the highest AP (75.18), lowest missed GT (12.55 dAP), and best calibration (ECE: 0.0447).

Overall, the model is **size-sensitive**, performing well on large objects but struggling with smaller ones, particularly in recall and confidence calibration. The TIDE analysis reports & plots are shown below.

```
-- coco_predictions_small --

bbox AP @ 50: 32.10

                        Main Errors
============================================================
  Type      Cls      Loc      Both     Dupe      Bkg      Miss
------------------------------------------------------------
   dAP     2.40     2.14     0.19     0.05     5.28     36.31
============================================================

       Special Error
============================
  Type   FalsePos   FalseNeg
----------------------------
   dAP       7.13      43.91
============================
```

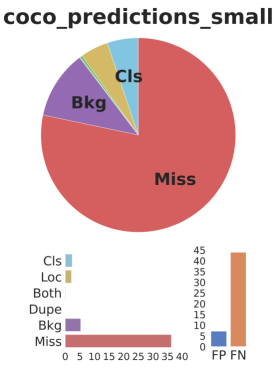


Figure 42: TIDE Analysis Metrics for Small Objects

```
-- coco_predictions_medium --

bbox AP @ 50: 58.26

                        Main Errors
============================================================
  Type      Cls      Loc      Both     Dupe      Bkg      Miss
------------------------------------------------------------
   dAP     1.97     2.80     0.50     0.04     5.39     22.63
============================================================

       Special Error
============================
  Type   FalsePos   FalseNeg
----------------------------
   dAP       8.07      27.80
============================
```

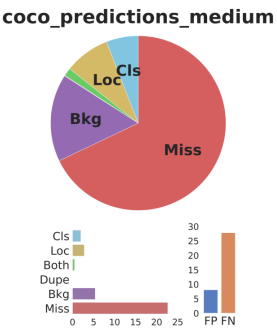


Figure 43: TIDE Analysis Metrics for Medium Objects

```
-- coco_predictions_large --

bbox AP @ 50: 75.18

                        Main Errors
============================================================
  Type      Cls      Loc      Both     Dupe      Bkg      Miss
------------------------------------------------------------
   dAP     2.84     3.55     0.78     0.11     1.45     12.55
============================================================

       Special Error
============================
  Type   FalsePos   FalseNeg
----------------------------
   dAP       4.65      18.46
============================
```

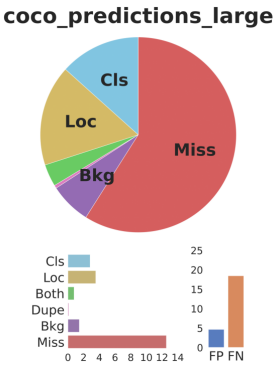


Figure 44: TIDE Analysis Metrics for Large Objects

(f) **Observations:**

- **Object size directly impacts performance**: Smaller objects have **lower AP** (32.10 for small vs. 75.18 for large) and **higher ECE** (0.2383 vs. 0.0447), indicating the model struggles with both detection and calibration for small objects.
- **Missed GT dominates errors**: Small objects have the highest missed GT error (**36.31 dAP**), suggesting severe recall issues for tiny objects.
- **Overconfidence in small objects**: High ECE for small objects (0.2383) implies the model's confidence scores are poorly calibrated (e.g., predicts 90% confidence but fails 24% more often).

Observations across scales:

- Small objects struggle due to low resolution and occlusion, leading to poor recall and calibration.
- Medium objects perform moderately well but still show overconfidence with an ECE above 0.1.
- Large objects are detected reliably with well-calibrated confidence scores.

Comparison with overall metrics:

- **Overall TIDE stats** (AP@50: 61.61, Miss: 21.65 dAP) indicate that small and medium objects significantly impact performance, while large objects improve the overall AP.
- **Overall ECE** (0.1234) lies between small (0.2383) and medium (0.1006), confirming that calibration worsens for smaller objects.

References

- **Ultralytics COCO Dataset Download:** Script to download the COCO validation split. [Ultralytics Documentation]
- **COCO Format for Predictions:** Guidelines for formatting predictions in the required style. [COCO Dataset]
- **TIDE Toolbox:** The TIDE (Toolbox for Identifying Detection Errors) framework was used to evaluate detection errors. [TIDE Toolbox Documentation]
- **TIDE Paper:** "TIDE: A General Toolbox for Identifying Object Detection Errors". [arXiv Paper]
- **Expected Calibration Error (ECE):** The formula for computing ECE is based on the paper "Calibration of Modern Neural Networks".[arXiv Paper]