Image Classification

Introduction

Image classification is the task of assigning a label or category to an image based on its visual content. Essentially, the model takes an image as input and predicts what object or scene is present in that image. In this Task, we work with the **Russian Wildlife Dataset**, a labeled dataset for classification tasks.

1.Russian Wildlife Dataset.

(a) Dataset Preparation

a.1 Dataset Description

The Russian Wildlife Dataset is a curated collection of images representing various animal species native to Russia. This dataset is designed to facilitate multi-class classification tasks, particularly in training and evaluating machine learning models for wildlife recognition.

Species Diversity: The dataset encompasses images of 10 distinct classes:

```
# Define class labels mapping
class_mapping = {
    '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
}
```

a.2 Dataset class

Applied image transformations following ResNet-18 standards:

- Resize the shorter side to 256 while maintaining aspect ratio.
- Center crop to 224x224.
- Convert image to tensor (C, H, W) in range [0, 1].
- Normalize using ImageNet mean and standard deviation: mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225].

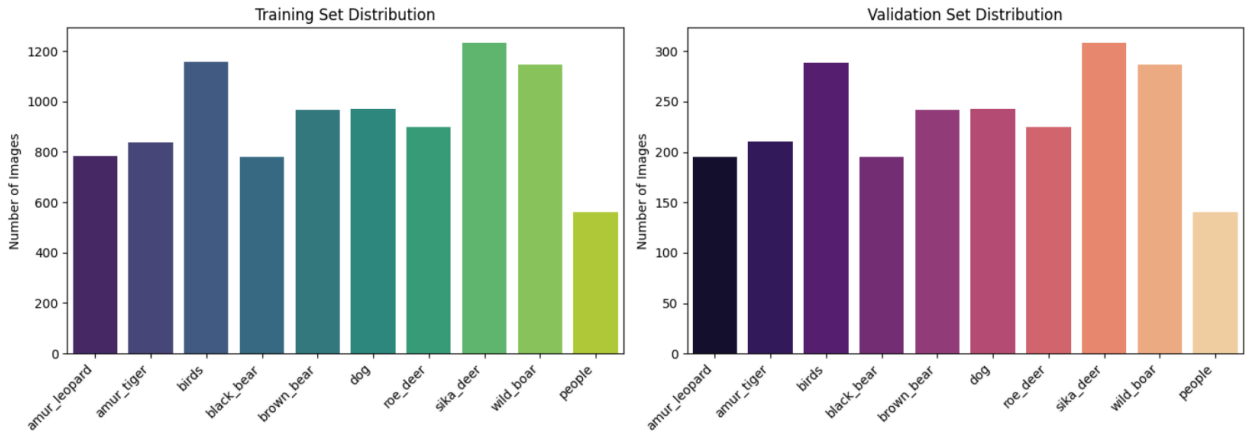
```
# Transformations Used by Resnet
resnet18_transform = transforms.Compose([
    transforms.Resize(256),  # Resize shorter side to 256 while maintaining aspect ratio
    transforms.CenterCrop(224),  # Crop to 224x224 at the center
    transforms.ToTensor(),  # Convert image to tensor (C, H, W) in range [0, 1]
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  # Normalize
])
```

```python
# Custom Dataset class
class RussianWildlifeDataset(Dataset):
    def __init__(self, image_paths, labels, transform=None, preload=True):
        self.image_paths = image_paths
        self.transform = transform
        self.preload = preload
        self.labels = torch.tensor(labels, dtype=torch.long)

        # Preload data into RAM if preload=True
        if self.preload:
            self.preloaded_data = [self.process_image(img_path) for img_path in self.image_paths]

    def process_image(self, img_path):
        image = Image.open(img_path).convert("RGB")
        if self.transform:
            image = self.transform(image)
        return image

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

    def __getitem__(self, idx):
        image = self.preloaded_data[idx] if self.preload else self.process_image(self.image_paths[idx])
        label = self.labels[idx]
        return image, label
```

(b) Create data loaders for all the splits (train and validation)

```python
# Create DataLoaders
batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)#,
```

(c) Data Distribution Visualization



2. Custom CNN from scratch

(a) ConvNet Architecture

This custom **ConvNet** is a convolutional neural network designed for image classification.

Input Shape:

- Input images are 3-channel (RGB) with a size of **224 x 224**.

🔹 **Layer 1: Convolution + Max Pooling**

- **Conv1:**
  - Input: 3 channels (RGB), Output: 32 feature maps.
  - Kernel size: 3x3, stride: 1, padding: 1 (same padding → output size remains 224x224).
  - Activation: ReLU.
- **MaxPool1:**
  - Kernel size: 4x4, stride: 4.
  - Reduces spatial dimensions by a factor of 4 → **224x224** → **56x56x32**.

🔹 **Layer 2: Convolution + Max Pooling**

- **Conv2:**
  - Input: 32 channels, Output: 64 feature maps.
  - Kernel size: 3x3, stride: 1, padding: 1 (same padding → output size remains 56x56).
  - Activation: ReLU.
- **MaxPool2:**
  - Kernel size: 2x2, stride: 2.
  - Reduces spatial dimensions by a factor of 2 → **56x56** → **28x28x64**.

🔹 **Layer 3: Convolution + Max Pooling**

- **Conv3:**
  - Input: 64 channels, Output: 128 feature maps.
  - Kernel size: 3x3, stride: 1, padding: 1 (same padding → output size remains 28x28).
  - Activation: ReLU.
- **MaxPool3:**
  - Kernel size: 2x2, stride: 2.
  - Reduces spatial dimensions by a factor of 2 → **28x28** → **14x14x128**.

🔹 **Flattening**

- **Flatten:**
  - Converts the 3D tensor into a 1D vector.
  - Size: **14 * 14 * 128 = 25088** features.

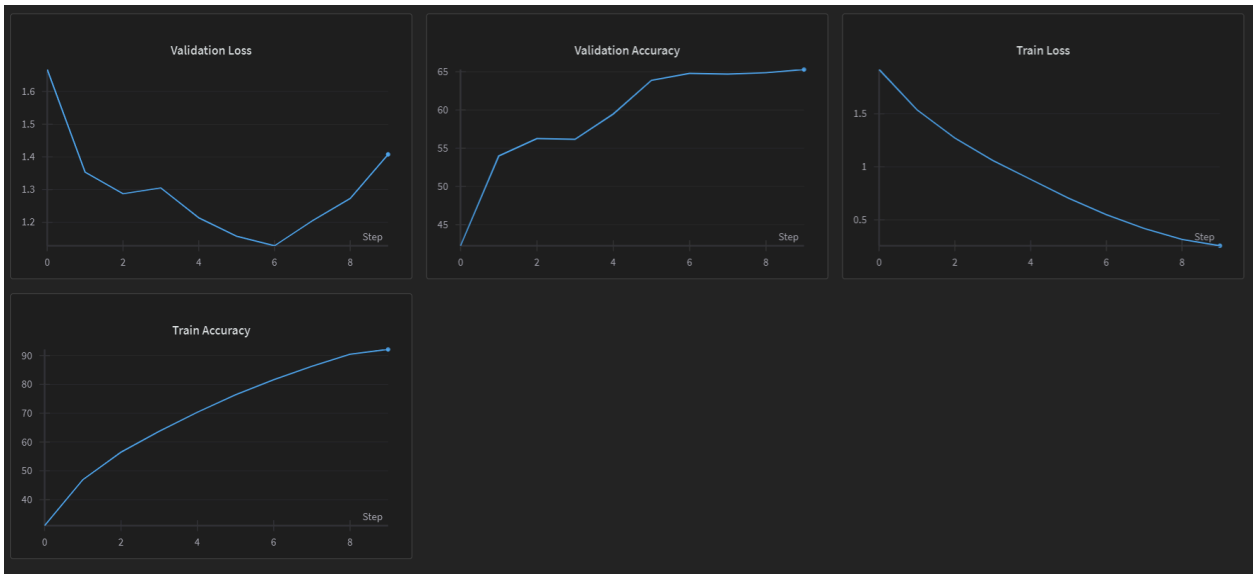
🔹 **Fully Connected Layers (Classification Head)**

- **FC1:**
  - Input: 25088, Output: 512.
  - Activation: ReLU.
- **FC2:**
  - Input: 512, Output: `num_classes` (10).
  - No activation (usually softmax applied later for classification).

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

        self.num_classes = num_classes  # Can be set dynamically before training

        # First Conv Layer: 3 -> 32 feature maps
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, stride=1, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=4, stride=4)

        # Second Conv Layer: 32 -> 64 feature maps
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Third Conv Layer: 64 -> 128 feature maps
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Input Size: 224x224, Flattened size: 14x14x128
        self.flatten = nn.Flatten()

        # Fully connected layers (classification head)
        self.fc1 = nn.Linear(128 * 14 * 14, 512)
        self.fc2 = nn.Linear(512, num_classes)

        # ReLU activation
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.pool1(x)

        x = self.relu(self.conv2(x))
        x = self.pool2(x)

        x = self.relu(self.conv3(x))
        x = self.pool3(x)

        x = self.flatten(x)
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

```

(b) Training (learning rate = 0.0003)

```
------------------------------ TRAINING convnet ------------------------------
Epoch [1/10] -> Train Loss: 1.9181, Train Acc: 30.99% | Val Loss: 1.6674, Val Acc: 42.25%
Epoch [2/10] -> Train Loss: 1.5378, Train Acc: 46.93% | Val Loss: 1.3533, Val Acc: 53.98%
Epoch [3/10] -> Train Loss: 1.2716, Train Acc: 56.48% | Val Loss: 1.2874, Val Acc: 56.26%
Epoch [4/10] -> Train Loss: 1.0603, Train Acc: 63.71% | Val Loss: 1.3052, Val Acc: 56.17%
Epoch [5/10] -> Train Loss: 0.8823, Train Acc: 70.32% | Val Loss: 1.2138, Val Acc: 59.47%
Epoch [6/10] -> Train Loss: 0.7042, Train Acc: 76.40% | Val Loss: 1.1575, Val Acc: 63.88%
Epoch [7/10] -> Train Loss: 0.5488, Train Acc: 81.63% | Val Loss: 1.1283, Val Acc: 64.78%
Epoch [8/10] -> Train Loss: 0.4182, Train Acc: 86.28% | Val Loss: 1.2048, Val Acc: 64.70%
Epoch [9/10] -> Train Loss: 0.3152, Train Acc: 90.45% | Val Loss: 1.2731, Val Acc: 64.87%
Epoch [10/10] -> Train Loss: 0.2556, Train Acc: 92.15% | Val Loss: 1.4077, Val Acc: 65.30%
```



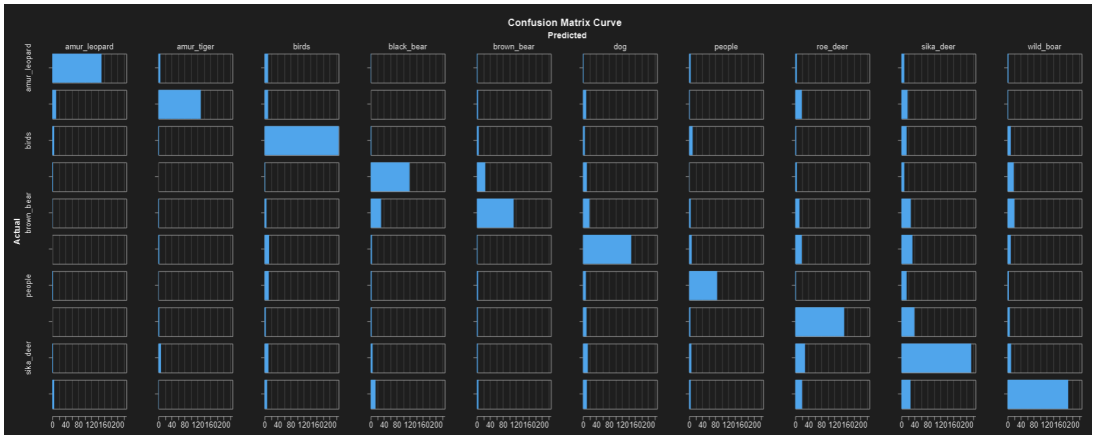
(c) Overfitting Analysis

- **Train-Validation Accuracy Gap:** The model achieves **92.15% train accuracy** but only **65.30% validation accuracy**, indicating it memorizes training data rather than generalizing.
- **Validation Loss Increasing After Epoch 6:** While training loss keeps decreasing, validation loss starts rising after **Epoch 6**, peaking at **1.4077** in the final epoch—showing the model is overfitting.

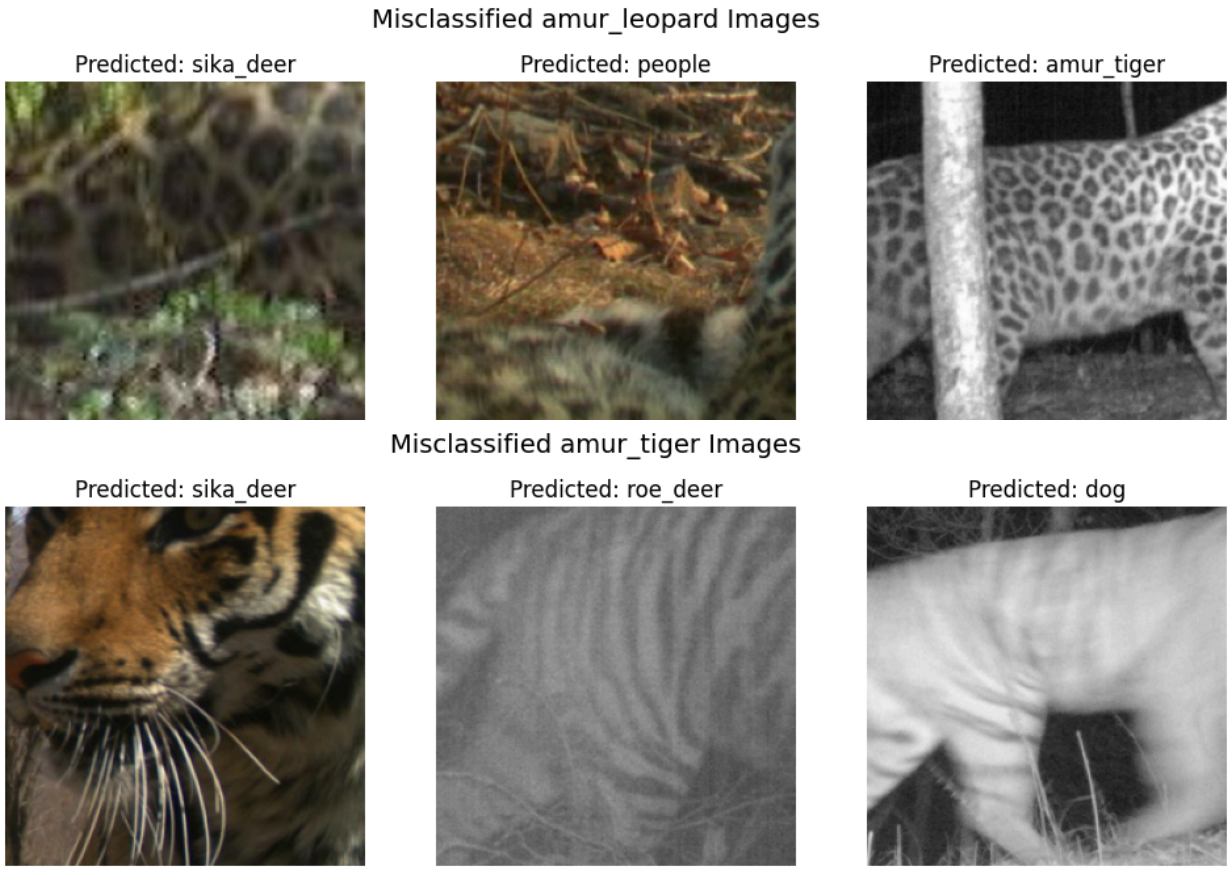
(d) d.1 Accuracy and F1-Score

```
Final Model Performance on Validation Set:

Accuracy: 65.2956%
F1 Score (Weighted): 0.6537
```

d.2 Confusion Matrix

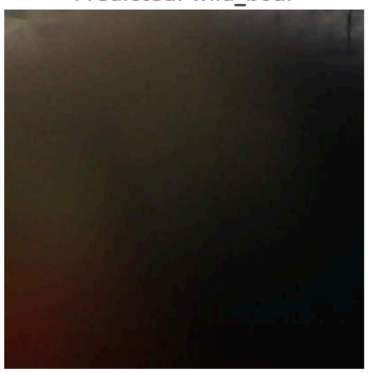


(e) Misclassification Analysis



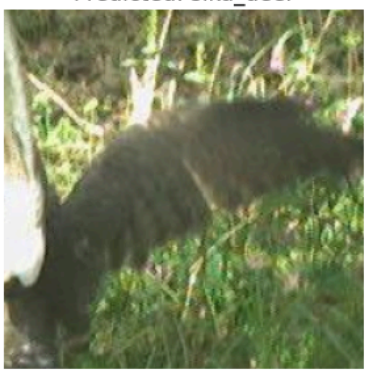
Misclassified birds Images

Predicted: wild_boar



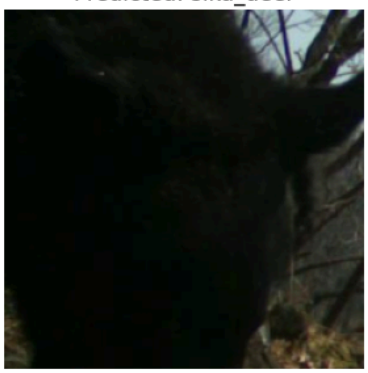
Predicted: sika_deer

Predicted: sika_deer



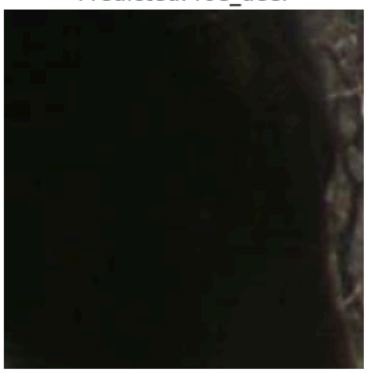
Misclassified black_bear Images

Predicted: sika_deer



Predicted: brown_bear

Predicted: roe_deer

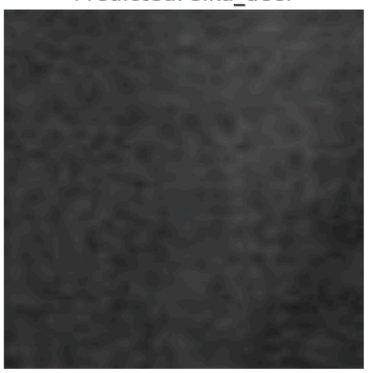


Misclassified roe_deer Images

Predicted: dog

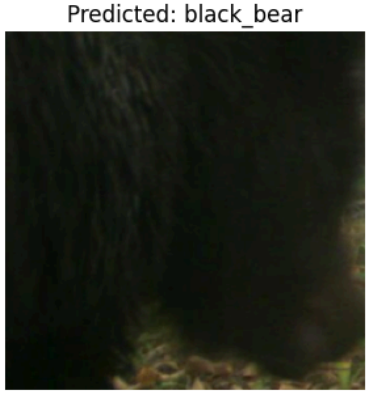
Predicted: dog

Predicted: sika_deer

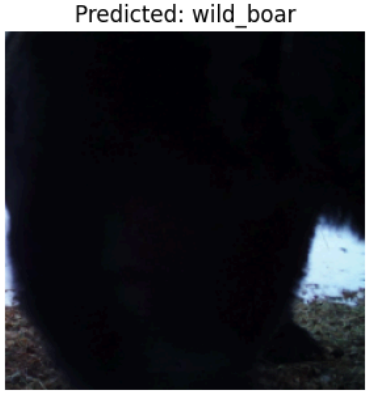


Misclassified brown_bear Images

Predicted: black_bear



Predicted: wild_boar

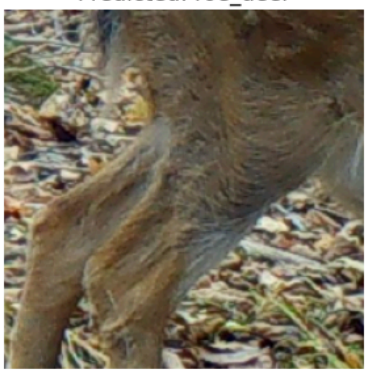


Predicted: black_bear

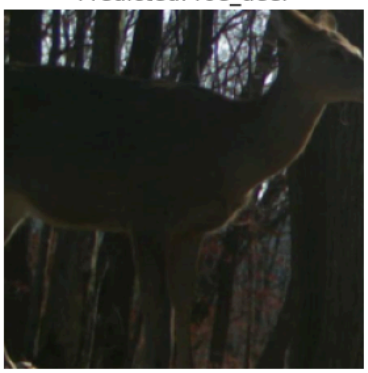
Misclassified sika_deer Images

Predicted: brown_bear

Predicted: roe_deer

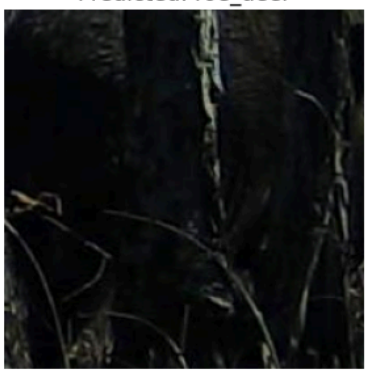


Predicted: roe_deer



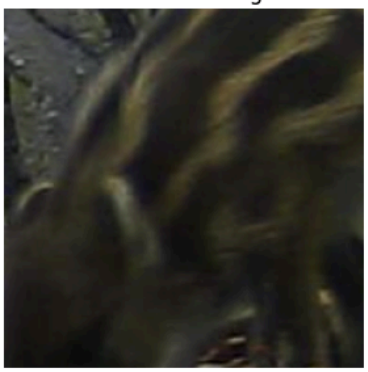
Misclassified wild_boar Images

Predicted: roe_deer



Predicted: roe_deer

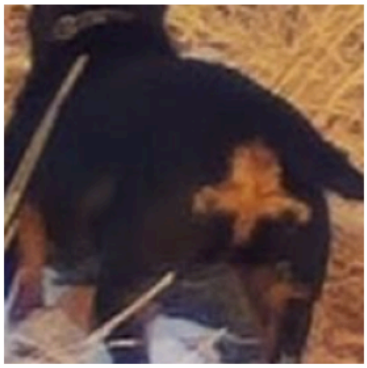
Predicted: dog



Misclassified people Images

Predicted: dog

Predicted: birds



Predicted: birds

Possible Reasons for Model Failure

1. **Image Does Not Contain the Ground Truth Class:**
    - Image might be mislabeled — a data quality issue.
    - Misleading labels can negatively affect model training and evaluation.
    - Ex - People Images have images of dog
2. **Image Looks More Similar to the Predicted Class:**
    - **Class Similarity:** Some classes share similar visual features (e.g.,black_bear vs brown_bear).
3. **Model Limitations:**
    - Insufficient discriminative features learned by the model.
    - Could be caused by:
        - Insufficient training data.
        - Lack of diversity in the dataset.
        - Inadequate model capacity.
4. **Challenging Samples:**
    - Some images are inherently difficult to classify due to:
        - **Occlusion:** Object is hidden or partially visible.
        - **Poor Lighting or Quality:** Low-resolution or poorly lit images.
        - **Unusual Poses or Angles:** Non-standard orientations confusing the model.
5. **Class Imbalance:**
    - Fewer samples for certain classes may cause the model to favor majority classes.
6. **Overfitting:**
    - Model may perform well on training data but fail to generalize to test/validation sets.

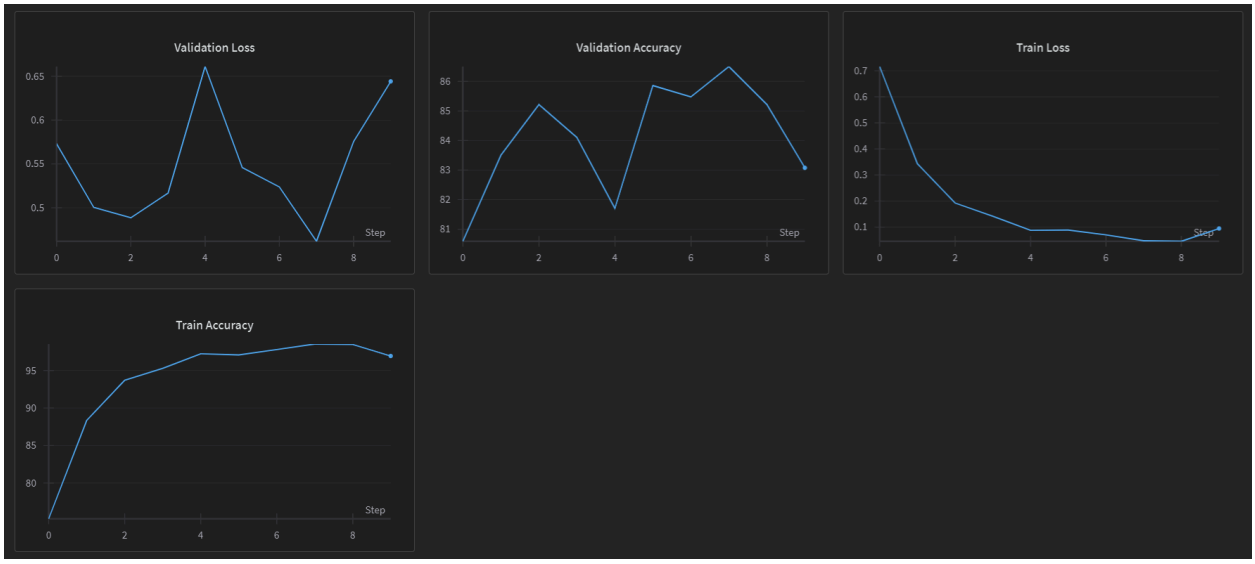
Workarounds for Such Samples

1. **Improve Data Quality:**
    - **Verify Labels:** Manually inspect and correct mislabeled samples.
    - **Remove Ambiguous Samples:** Consider removing unclear or highly ambiguous images.
2. **Data Augmentation:**
    - Use techniques like rotation, flipping, cropping, and brightness adjustments.
    - Helps the model generalize better to challenging and unseen variations.
3. **Class Balancing:**
    - **Oversampling:** Increase the number of samples for minority classes.
    - **Undersampling:** Reduce the number of samples from majority classes.
    - **Class-weighted Loss Functions:** Assign higher loss to minority classes to balance learning.

3. Fine-Tuning ResNet-18

(a) Training (learning rate = 0.0003)

```
-------------------------------- TRAINING resnet --------------------------------
Epoch [1/10] -> Train Loss: 0.7163, Train Acc: 75.23% | Val Loss: 0.5729, Val Acc: 80.59%
Epoch [2/10] -> Train Loss: 0.3433, Train Acc: 88.35% | Val Loss: 0.5005, Val Acc: 83.50%
Epoch [3/10] -> Train Loss: 0.1924, Train Acc: 93.70% | Val Loss: 0.4886, Val Acc: 85.22%
Epoch [4/10] -> Train Loss: 0.1418, Train Acc: 95.30% | Val Loss: 0.5167, Val Acc: 84.10%
Epoch [5/10] -> Train Loss: 0.0877, Train Acc: 97.24% | Val Loss: 0.6611, Val Acc: 81.71%
Epoch [6/10] -> Train Loss: 0.0887, Train Acc: 97.08% | Val Loss: 0.5458, Val Acc: 85.86%
Epoch [7/10] -> Train Loss: 0.0699, Train Acc: 97.79% | Val Loss: 0.5238, Val Acc: 85.48%
Epoch [8/10] -> Train Loss: 0.0475, Train Acc: 98.53% | Val Loss: 0.4618, Val Acc: 86.50%
Epoch [9/10] -> Train Loss: 0.0455, Train Acc: 98.48% | Val Loss: 0.5756, Val Acc: 85.22%
Epoch [10/10] -> Train Loss: 0.0946, Train Acc: 96.94% | Val Loss: 0.6441, Val Acc: 83.08%
```



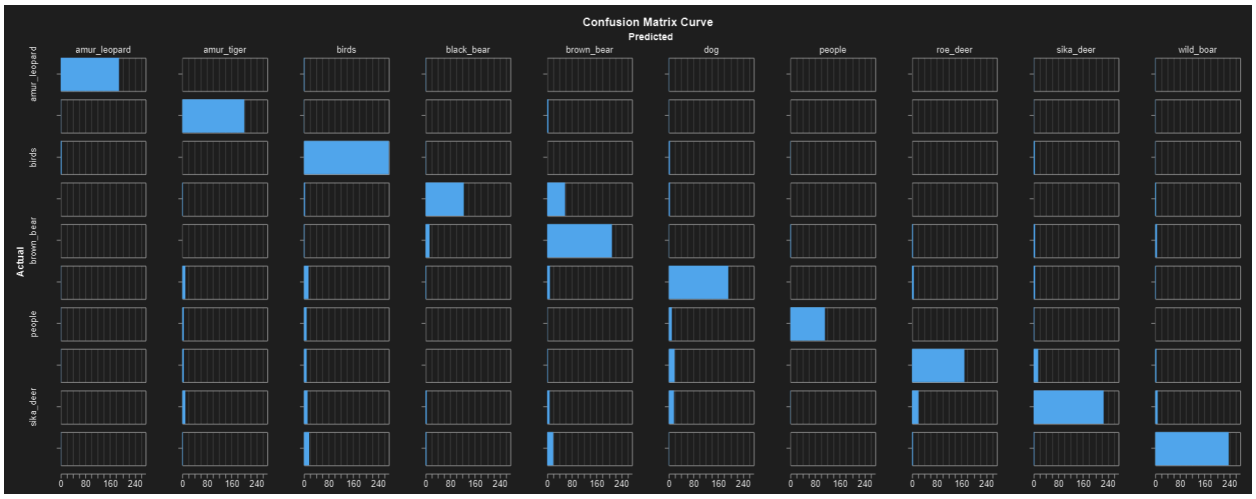
(b) Overfitting Analysis

- **Train-Validation Performance Gap:** The model reaches **98.53% train accuracy** but peaks at **86.50% validation accuracy**, indicating mild overfitting as the training accuracy is significantly higher.
- **Validation Loss Fluctuations:** Validation loss **decreases until Epoch 8 (0.4618)** but then increases again, reaching **0.6441 in the final epoch**, suggesting the model starts overfitting after Epoch 8.

(c) c.1 Accuracy and F1-Score

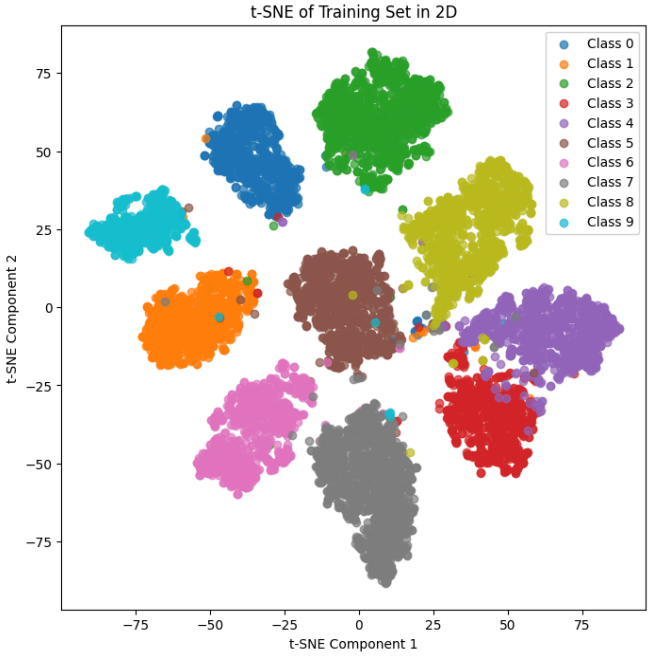
```
Final Model Performance on Validation Set:

Accuracy: 83.0763%
F1 Score (Weighted): 0.8299
```

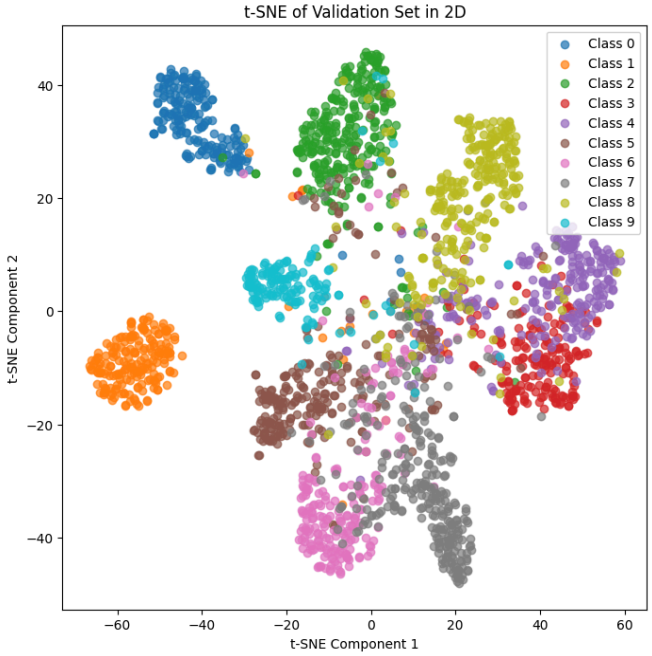
c.2 Confusion Matrix

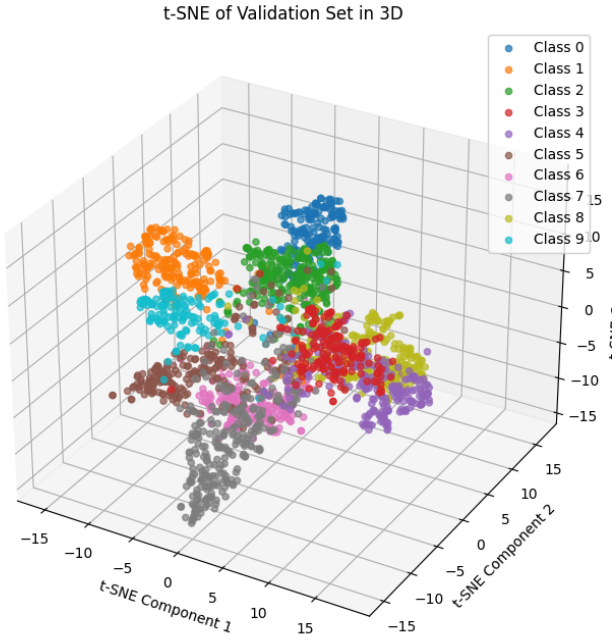


(d) Feature Space Visualization

- Extracted feature vectors from the ResNet-18 backbone.
- Visualized feature vectors using t-SNE (2D and 3D plots) for training and validation sets.







3. Fine-Tuning ResNet-18 with Augmented Data

(a) Data Augmentation

50% of Original Data was Augmented and concatenated

a.1 Augmentation Techniques

RandomHorizontalFlip(p=1.0):

- Flips the image horizontally with a probability of 100% (since `p=1.0`).
- This helps the model learn invariance to left-right orientations.

RandomRotation(10):

- Rotates the image randomly within a range of ±10 degrees.
- Adds slight rotational invariance, helping with varied object orientations.

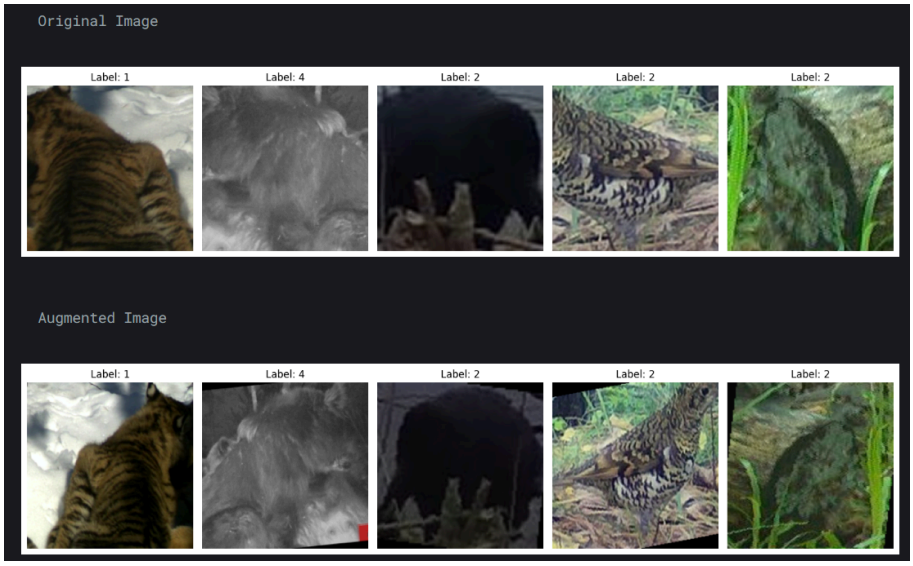
RandomResizedCrop(224, scale=(0.8, 1.0)):

- Crops a random portion of the image and resizes it to 224x224 pixels.
- The crop size is randomly selected within 80%-100% of the original image size, encouraging the model to learn from different parts of the image.

ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1):

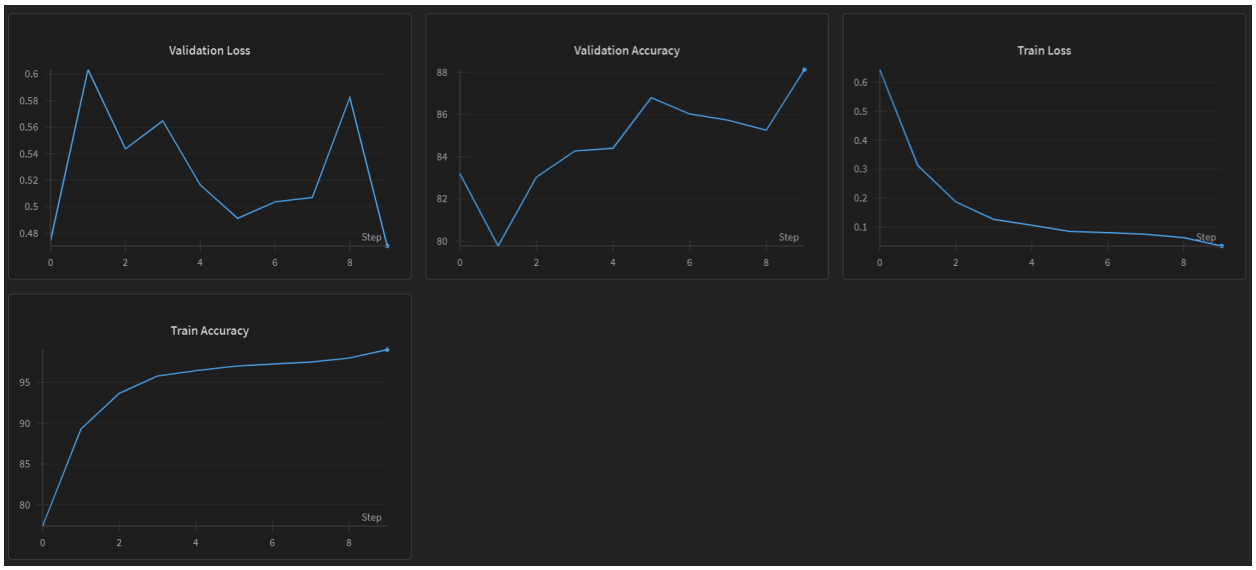
- Randomly changes the brightness, contrast, saturation, and hue of the image.
- This helps the model become robust to lighting variations and color differences.

a.2 Augmented Images Visualization



(b) Training (learning rate = 0.0003)

```
------------------------------ TRAINING resnet_aug ------------------------------
Epoch [1/10] -> Train Loss: 0.6443, Train Acc: 77.39% | Val Loss: 0.4747, Val Acc: 83.20%
Epoch [2/10] -> Train Loss: 0.3133, Train Acc: 89.29% | Val Loss: 0.6034, Val Acc: 79.78%
Epoch [3/10] -> Train Loss: 0.1872, Train Acc: 93.66% | Val Loss: 0.5437, Val Acc: 83.03%
Epoch [4/10] -> Train Loss: 0.1267, Train Acc: 95.79% | Val Loss: 0.5648, Val Acc: 84.28%
Epoch [5/10] -> Train Loss: 0.1064, Train Acc: 96.44% | Val Loss: 0.5165, Val Acc: 84.40%
Epoch [6/10] -> Train Loss: 0.0851, Train Acc: 96.99% | Val Loss: 0.4914, Val Acc: 86.80%
Epoch [7/10] -> Train Loss: 0.0809, Train Acc: 97.26% | Val Loss: 0.5037, Val Acc: 86.03%
Epoch [8/10] -> Train Loss: 0.0753, Train Acc: 97.49% | Val Loss: 0.5071, Val Acc: 85.73%
Epoch [9/10] -> Train Loss: 0.0635, Train Acc: 97.98% | Val Loss: 0.5823, Val Acc: 85.26%
Epoch [10/10] -> Train Loss: 0.0348, Train Acc: 99.01% | Val Loss: 0.4706, Val Acc: 88.13%
```



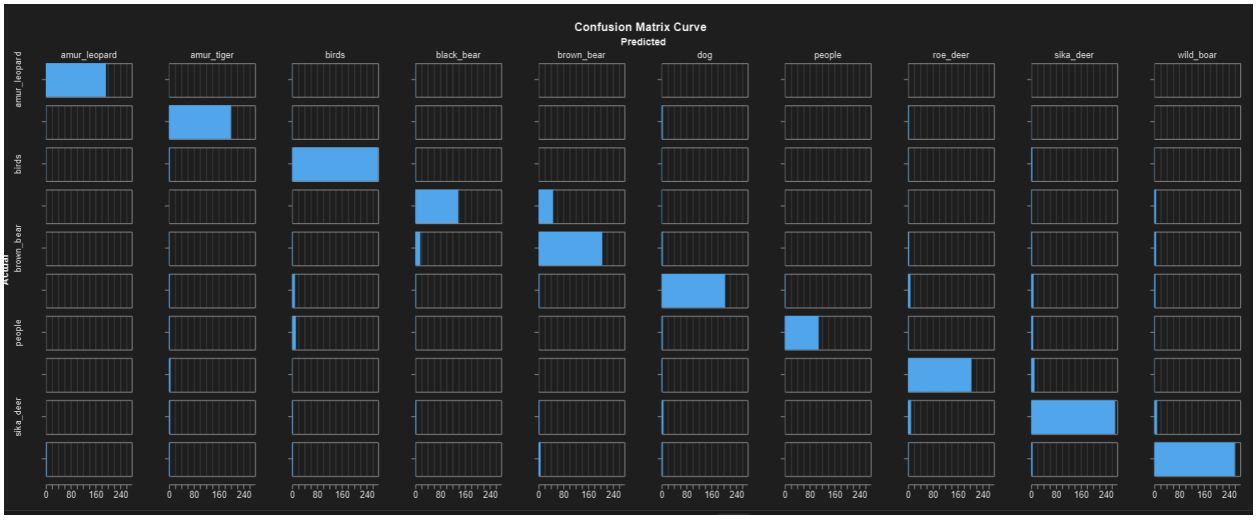
(c) Overfitting Analysis

- **Train-Validation Performance Gap:** The model achieves **99.01% train accuracy** but **88.13% validation accuracy**, showing a smaller gap than the previous ResNet model, indicating better generalization.
- **Validation Loss Stability:** Validation loss fluctuates but **ends lower (0.4706) than earlier epochs**, suggesting that data augmentation helped reduce overfitting and improved generalization.

(d) d.1 Accuracy and F1-Score

```
Final Model Performance on Validation Set:

Accuracy: 88.1320%
F1 Score (Weighted): 0.8804
```

d.2 Confusion Matrix



5. Model Comparison

Model	Train Acc (%)	Val Acc (%)	Train-Val Gap	Best Val Acc	Final Val Loss	Overfitting Severity
ConvNet	92.15	65.30	**26.85%**	65.30%	1.4077	High
ResNet	98.53	86.50	**12.03%**	86.50%	0.6441	Moderate
ResNet_Aug	99.01	88.13	**10.88%**	**88.13%**	0.4706	Low

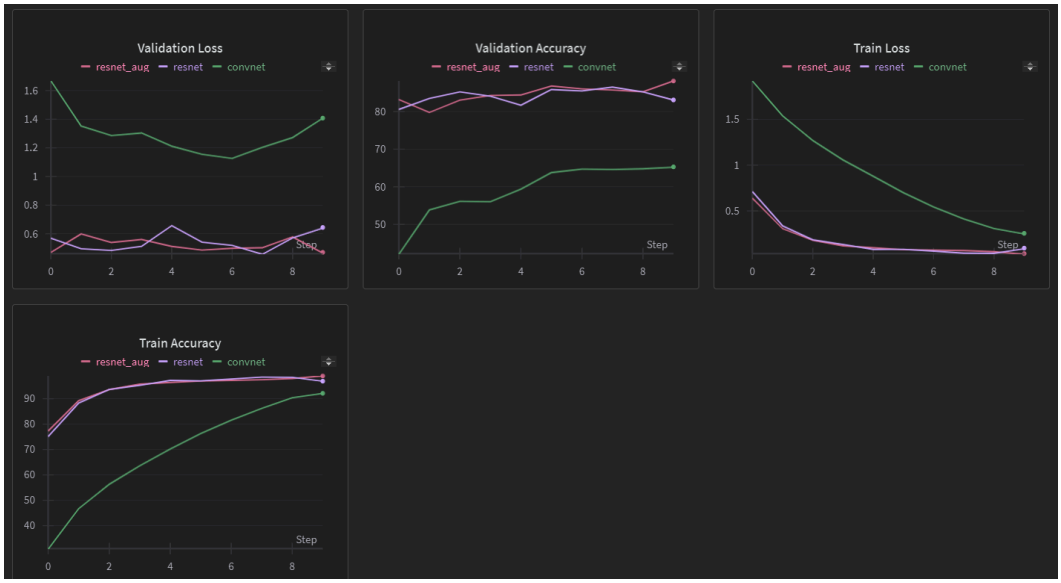
Observations and Comments:

1. **ConvNet is the weakest performer**
    - It has the **largest train-validation gap (26.85%)**, indicating severe overfitting.
    - Validation accuracy peaks at **only 65.30%**, making it significantly worse than the other models.
2. **ResNet improves performance**
    - **Higher validation accuracy (86.50%)** compared to ConvNet.
    - Overfitting is still present but much less severe than ConvNet.
3. **ResNet_Aug is the best model**
    - **Highest validation accuracy (88.13%)** with the lowest validation loss.

- **Smallest train-validation gap (10.88%)**, indicating the best generalization.
- Augmentation helped stabilize performance and reduce overfitting.

Final Verdict:

ResNet_Aug performs the best, offering the highest accuracy and best generalization. If you need a model for real-world deployment, this would be the best choice.



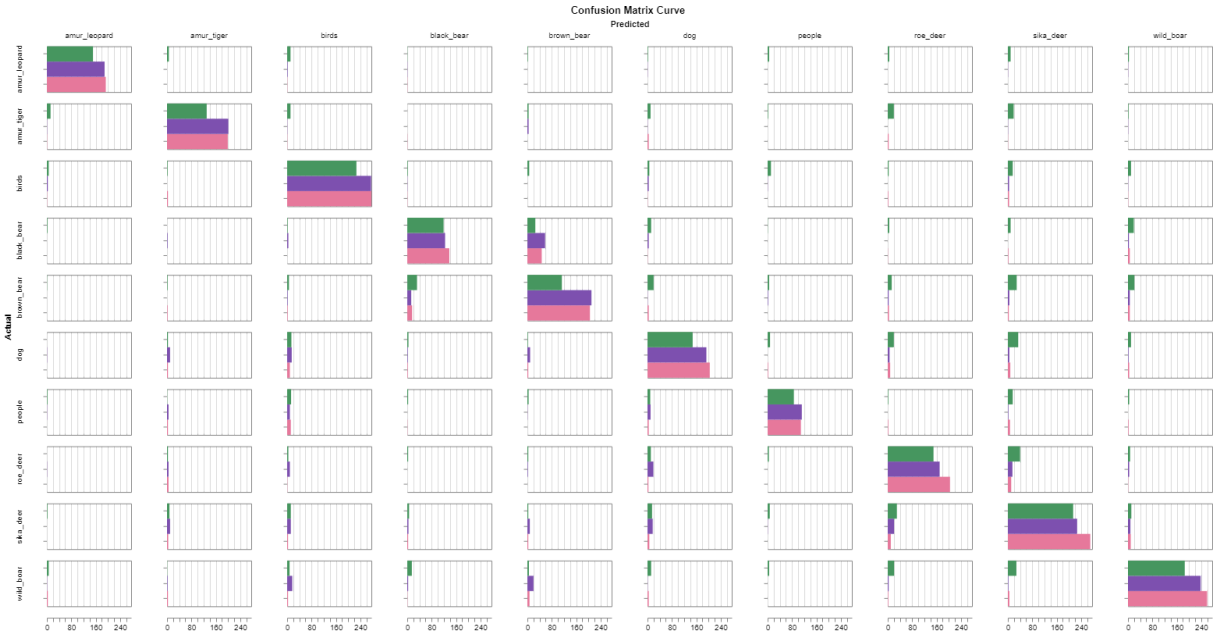


Image Segmentation

Introduction

Image segmentation is a fundamental task in computer vision that involves partitioning an image into meaningful regions, often corresponding to objects or specific structures. In this Task, we work with the **CAMVid dataset**, a labeled dataset for semantic segmentation tasks.

1. Download the CAMVid dataset.

(a) Dataset Preparation

a.1 Dataset Description

The **CAMVid dataset** consists of urban street scenes, with each pixel labeled according to its corresponding class. The dataset includes training, validation, and test images along with corresponding segmentation masks. The original image resolution is **(960 × 720)**.

a.2 Data Loading and Preprocessing

To ensure proper model input, we apply the following preprocessing steps:

- **Resizing** images to **(480 × 360)**.
- **Normalization** using the mean `[0.485, 0.456, 0.406]` and standard deviation `[0.229, 0.224, 0.225]`.

```
# Define transforms
transform = transforms.Compose([
    transforms.Resize((360, 480)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

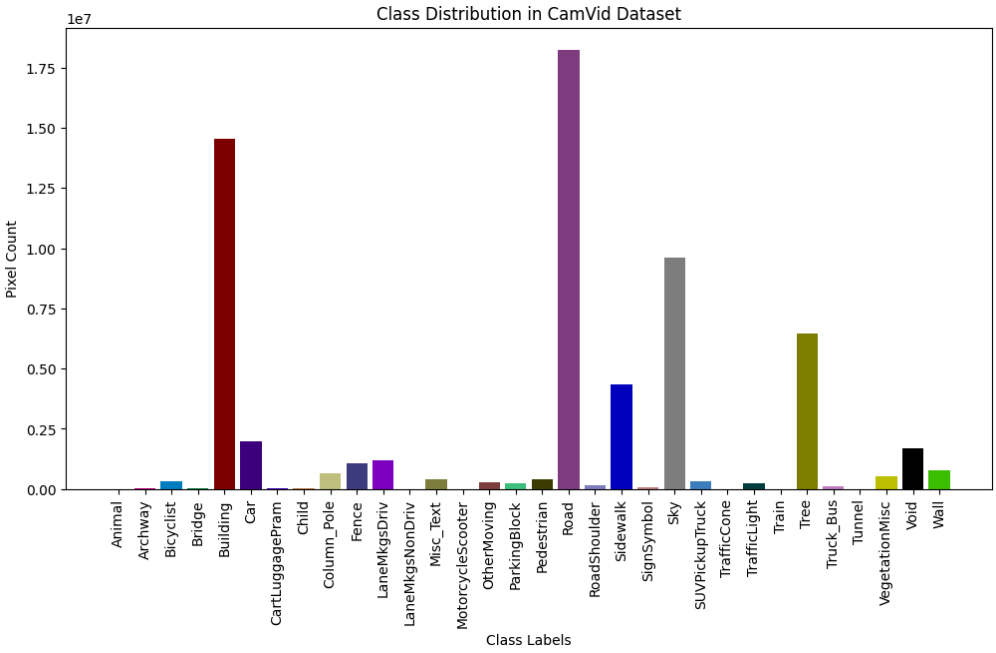
- **Encoding segmentation masks**, converting RGB masks into label indices.
- A custom PyTorch **Dataloader** was implemented to handle these transformations efficiently with **batch size: 10**

```
Train Dataset Size: 369
Test Dataset Size: 232
```

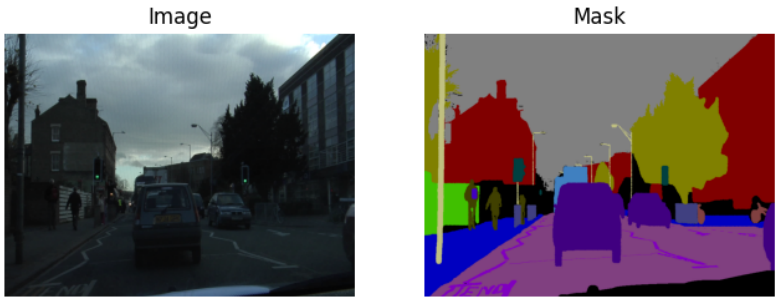
```python
class CamVidDataset(Dataset):
    def __init__(self, image_dir, mask_dir, class_dict_path, transform=None, preload=True):
        self.image_dir = image_dir
        self.mask_dir = mask_dir
        self.transform = transform
        self.preload = preload
        self.class_dict = pd.read_csv(class_dict_path)
        self.images = os.listdir(image_dir)

        # Load class mapping
        self.color_to_label = {
            tuple(self.class_dict.iloc[i, 1:4].astype(int)): i
            for i in range(len(self.class_dict))
        }

        self.mask_transform = transforms.Compose([
            transforms.Resize((360, 480), interpolation=Image.NEAREST)
        ])

        # Preload data into RAM if preload=True
        if self.preload:
            self.preloaded_data = []
            for img_name in self.images:
                self.preloaded_data.append(self.process_image_mask(img_name))

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

    def encode_mask(self, mask):
        mask = np.array(mask, dtype=np.uint8)
        label_mask = np.zeros(mask.shape[:2], dtype=np.int64)
        for color, label in self.color_to_label.items():
            label_mask[(mask == color).all(axis=-1)] = label
        return torch.tensor(label_mask, dtype=torch.long)

    def process_image_mask(self, img_name):
        img_path = os.path.join(self.image_dir, img_name)
        mask_path = os.path.join(self.mask_dir, img_name.replace('.png', '_L.png'))

        image = Image.open(img_path).convert("RGB")
        mask = Image.open(mask_path).convert("RGB")

        if self.transform:
            image = self.transform(image)

        mask = self.mask_transform(mask)
        mask = self.encode_mask(mask)

        return image, mask

    def __getitem__(self, idx):
        return self.preloaded_data[idx] if self.preload else self.process_image_mask(self.images[idx])

    def visualize_sample(self, idx):
        img_path = os.path.join(self.image_dir, self.images[idx])
        mask_path = os.path.join(self.mask_dir, self.images[idx].replace('.png', '_L.png'))

        image = Image.open(img_path).convert("RGB")
        mask = Image.open(mask_path).convert("RGB")

        fig, axes = plt.subplots(1, 2, figsize=(8, 5))
        axes[0].imshow(image)
        axes[0].set_title("Image")
        axes[0].axis("off")

        axes[1].imshow(mask)
        axes[1].set_title("Mask")
        axes[1].axis("off")

        plt.show()
```

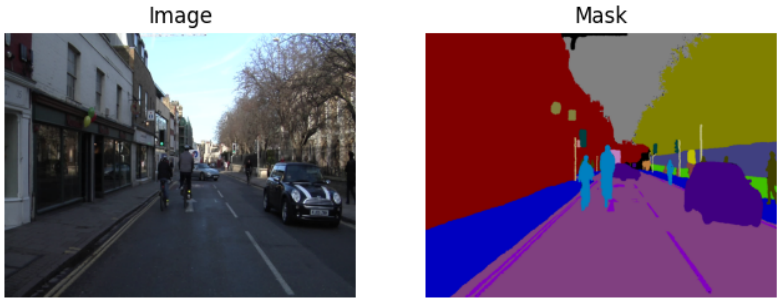
(b) Class Distribution Visualization

To analyze class frequency in the dataset, we computed the pixel-wise distribution of class labels across the training images.



(c) Image and Mask Visualization





2. SegNet Encoder-Decoder

(a) Implementation and Training

a.1 Architecture

The SegNet was implemented by following the architecture specified in the `model classes.py` file. The encoder consists of five stages, each corresponding to an encoding stage. The layers in the decoder mirror the encoder using transposed convolutions, batch normalization, and activation functions.

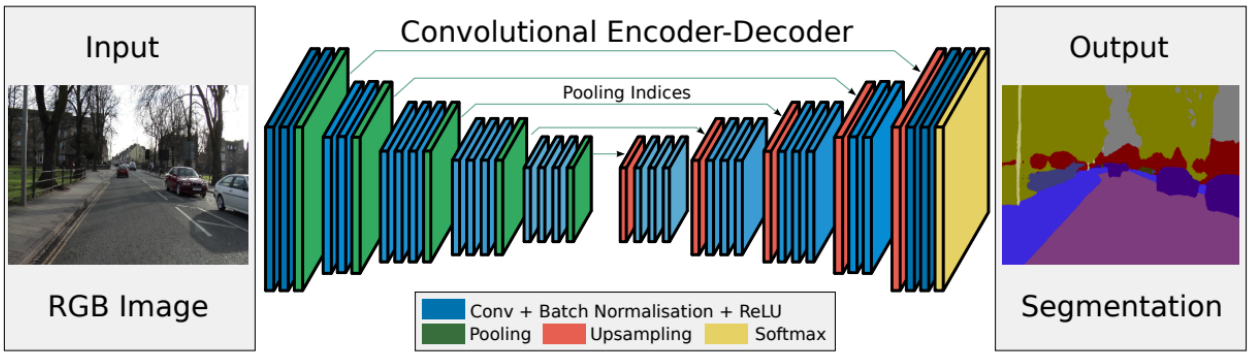
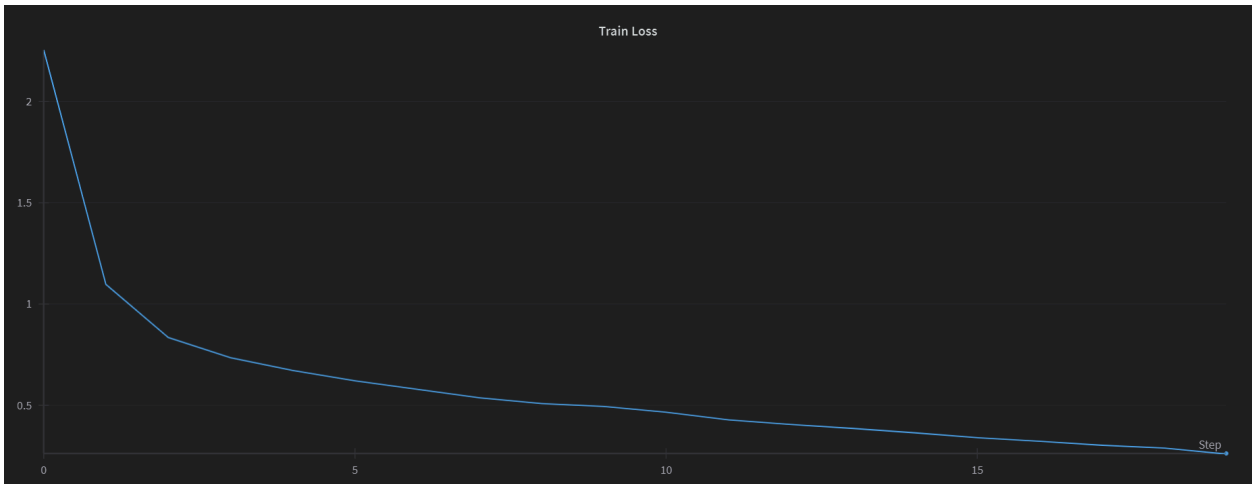


Fig. 2. An illustration of the SegNet architecture. There are no fully connected layers and hence it is only convolutional. A decoder upsamples its input using the transferred pool indices from its encoder to produce a sparse feature map(s). It then performs convolution with a trainable filter bank to densify the feature map. The final decoder output feature maps are fed to a soft-max classifier for pixel-wise classification.

a.2 Training Setup

- **Loss Function:** Cross-entropy loss
- **Optimizer:** Adam optimizer
- **Batch Normalization Momentum:** 0.5
- **Learning rate:** 0.0002
- **Epoch:** 20

a.3 Training logs




```
================================ TRAINING segnet ================================
Epoch [1/20] -> Train Loss: 2.2558
Epoch [2/20] -> Train Loss: 1.0993
Epoch [3/20] -> Train Loss: 0.8367
Epoch [4/20] -> Train Loss: 0.7373
Epoch [5/20] -> Train Loss: 0.6747
Epoch [6/20] -> Train Loss: 0.6237
Epoch [7/20] -> Train Loss: 0.5817
Epoch [8/20] -> Train Loss: 0.5400
Epoch [9/20] -> Train Loss: 0.5114
Epoch [10/20] -> Train Loss: 0.4970
Epoch [11/20] -> Train Loss: 0.4687
Epoch [12/20] -> Train Loss: 0.4307
Epoch [13/20] -> Train Loss: 0.4084
Epoch [14/20] -> Train Loss: 0.3887
Epoch [15/20] -> Train Loss: 0.3666
Epoch [16/20] -> Train Loss: 0.3428
Epoch [17/20] -> Train Loss: 0.3251
Epoch [18/20] -> Train Loss: 0.3056
Epoch [19/20] -> Train Loss: 0.2917
Epoch [20/20] -> Train Loss: 0.2621
================================ TRAINING COMPLETED ================================
```

a.4 Model Saving

The SegNet decoder was successfully trained using a pre-trained encoder

SegNet_Encoder is saved in ***encoder_model.pth***

SegNet_Decoder is saved in ***decoder.pth***

(b) Performance Evaluation

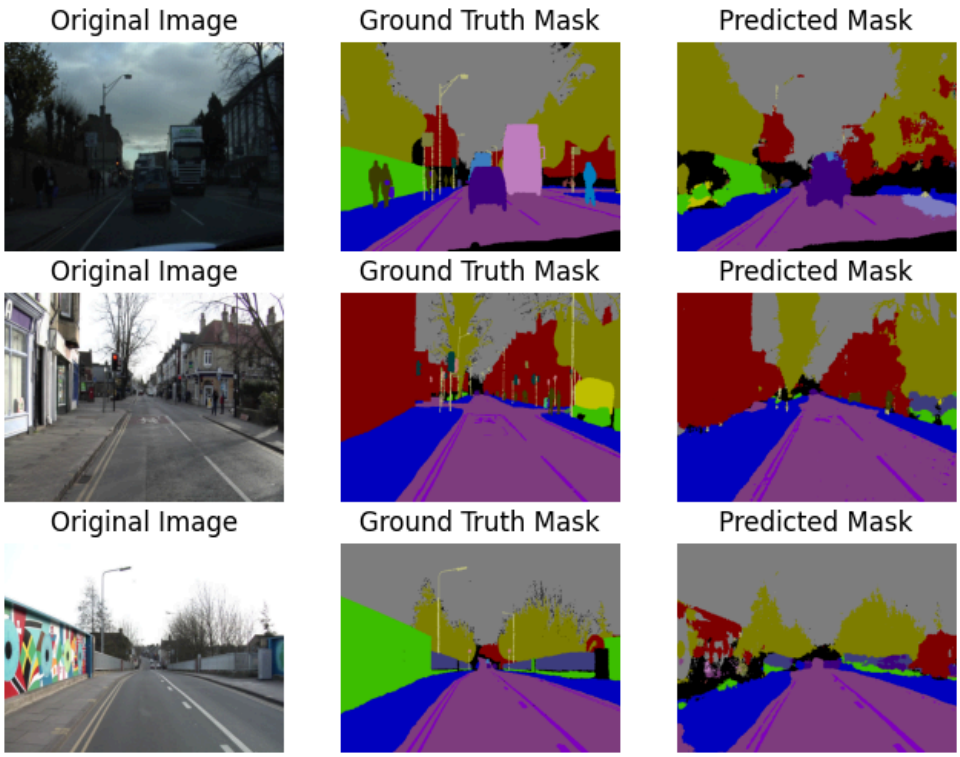
```
Pixel Accuracy: 0.8150
Mean IoU (mIoU): 0.2272

Class-wise Metrics:
Class 0: IoU=0.0000, Dice=0.0000, Precision=0.0000, Recall=0.0000
Class 1: IoU=0.0000, Dice=0.0000, Precision=0.0000, Recall=0.0000
Class 2: IoU=0.3223, Dice=0.4874, Precision=0.7264, Recall=0.3668
Class 3: IoU=0.0000, Dice=0.0000, Precision=0.0000, Recall=0.0000
Class 4: IoU=0.7550, Dice=0.8604, Precision=0.8510, Recall=0.8700
Class 5: IoU=0.6155, Dice=0.7620, Precision=0.7673, Recall=0.7568
Class 6: IoU=0.0000, Dice=0.0000, Precision=0.0000, Recall=0.0000
Class 7: IoU=0.0000, Dice=0.0000, Precision=0.0000, Recall=0.0000
Class 8: IoU=0.0523, Dice=0.0994, Precision=0.4449, Recall=0.0559
Class 9: IoU=0.2815, Dice=0.4393, Precision=0.4963, Recall=0.3941
Class 10: IoU=0.4408, Dice=0.6119, Precision=0.7384, Recall=0.5223
Class 11: IoU=0.0000, Dice=0.0000, Precision=0.0000, Recall=0.0000
Class 12: IoU=0.0317, Dice=0.0615, Precision=0.1046, Recall=0.0436
Class 13: IoU=0.0000, Dice=0.0000, Precision=0.0000, Recall=0.0000
Class 14: IoU=0.1922, Dice=0.3225, Precision=0.3902, Recall=0.2748
Class 15: IoU=0.1480, Dice=0.2579, Precision=0.2204, Recall=0.3106
Class 16: IoU=0.1188, Dice=0.2124, Precision=0.3126, Recall=0.1609
Class 17: IoU=0.8758, Dice=0.9338, Precision=0.9376, Recall=0.9300
Class 18: IoU=0.1430, Dice=0.2502, Precision=0.1621, Recall=0.5479
Class 19: IoU=0.6839, Dice=0.8123, Precision=0.8031, Recall=0.8217
Class 20: IoU=0.0052, Dice=0.0104, Precision=0.8651, Recall=0.0052
Class 21: IoU=0.8945, Dice=0.9443, Precision=0.9310, Recall=0.9580
Class 22: IoU=0.0585, Dice=0.1105, Precision=0.1739, Recall=0.0809
Class 23: IoU=0.0000, Dice=0.0000, Precision=0.0000, Recall=0.0000
Class 24: IoU=0.1687, Dice=0.2887, Precision=0.6612, Recall=0.1847
Class 25: IoU=0.0000, Dice=0.0000, Precision=0.0000, Recall=0.0000
Class 26: IoU=0.6588, Dice=0.7943, Precision=0.7279, Recall=0.8740
Class 27: IoU=0.0624, Dice=0.1175, Precision=0.5183, Recall=0.0663
Class 28: IoU=0.0000, Dice=0.0000, Precision=0.0000, Recall=0.0000
Class 29: IoU=0.1131, Dice=0.2033, Precision=0.3519, Recall=0.1429
Class 30: IoU=0.3189, Dice=0.4836, Precision=0.4197, Recall=0.5704
Class 31: IoU=0.3309, Dice=0.4972, Precision=0.5536, Recall=0.4513
```

(c) Visualization and Failure Analysis

c.1 Visualizing IoU ≤ 0.5 Cases

For each class, three images where the 0 < IoU ≤ 0.5 were selected but for some classes there are no examples which might be because class objects are not present in the test dataset. These images show the predicted masks compared to ground truth.

Example Figures:



c.2 Failure Analysis

The model struggles in the following cases:

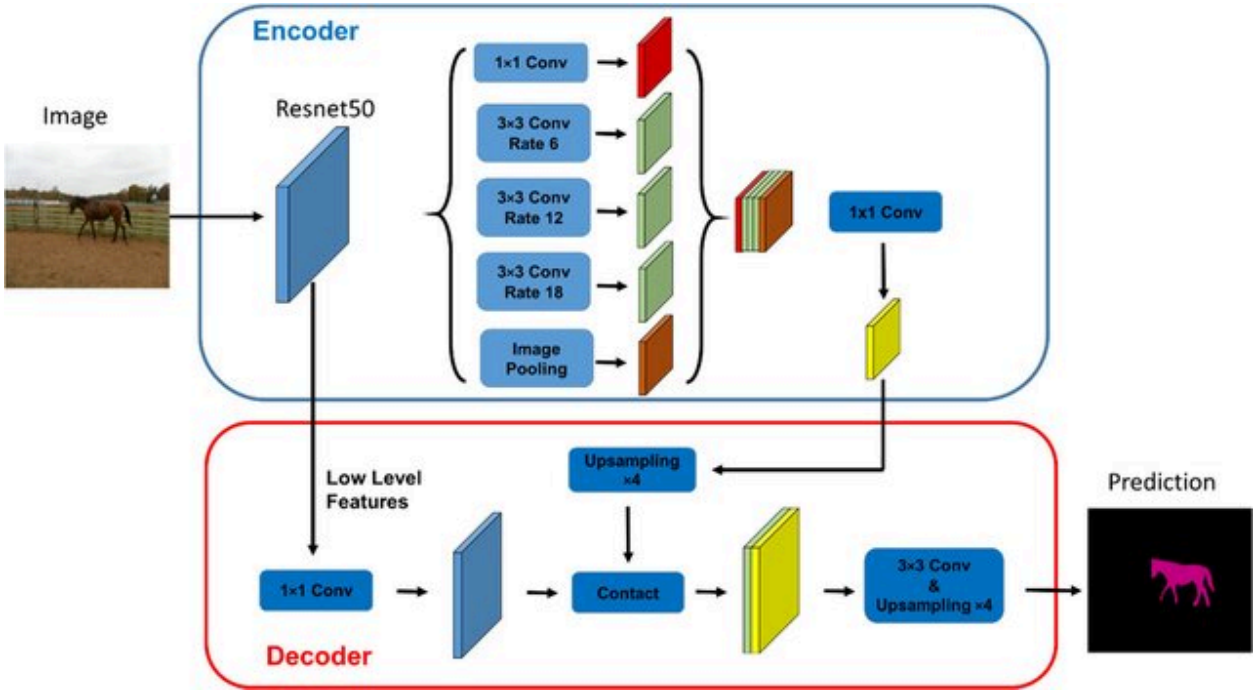
- **Occlusion:** Objects are partially visible.
- **Misclassification:** The model confuses similar-looking objects like bicyclist and pedestrians
- **Environmental Challenges:** Low lighting or complex backgrounds.
- **Small Object Detection:** CamVid contains small, thin objects like poles and signs, which are hard for convolutional layers to segment accurately.
- **Ambiguous Boundaries:** Blurred or unclear boundaries between objects (e.g., road and sidewalk) can confuse the model.
- **SegNet** lacks advanced boundary refinement techniques.

3. DeepLabv3 (ResNet50)

(a) Implementation and Training

a.1 Architecture

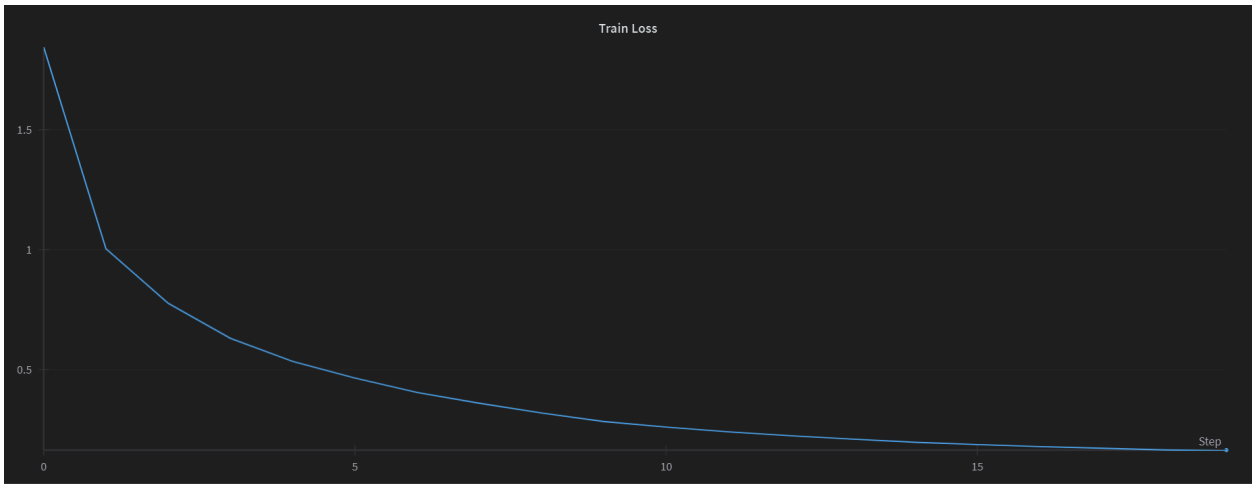
DeepLabV3, pre-trained on the Pascal VOC dataset, was fine-tuned for CamVid segmentation. The classifier was modified to predict 32 classes.



a.2 Training Setup

- **Loss Function:** Cross-entropy loss
- **Optimizer:** Adam optimizer
- **Batch Normalization Momentum:** 0.5
- **Learning rate:** 0.0002
- **Epoch:** 20

a.3 Training logs




```
================================ TRAINING deeplabv3 ================================
Epoch [1/20] -> Train Loss: 1.8444
Epoch [2/20] -> Train Loss: 1.0057
Epoch [3/20] -> Train Loss: 0.7784
Epoch [4/20] -> Train Loss: 0.6327
Epoch [5/20] -> Train Loss: 0.5366
Epoch [6/20] -> Train Loss: 0.4672
Epoch [7/20] -> Train Loss: 0.4074
Epoch [8/20] -> Train Loss: 0.3625
Epoch [9/20] -> Train Loss: 0.3218
Epoch [10/20] -> Train Loss: 0.2863
Epoch [11/20] -> Train Loss: 0.2632
Epoch [12/20] -> Train Loss: 0.2437
Epoch [13/20] -> Train Loss: 0.2273
Epoch [14/20] -> Train Loss: 0.2134
Epoch [15/20] -> Train Loss: 0.2003
Epoch [16/20] -> Train Loss: 0.1907
Epoch [17/20] -> Train Loss: 0.1823
Epoch [18/20] -> Train Loss: 0.1756
Epoch [19/20] -> Train Loss: 0.1684
Epoch [20/20] -> Train Loss: 0.1649
============================== TRAINING COMPLETED ==============================
```

a.4 Model Saving

The DeepLabv3 decoder was successfully fine tuned on CamVid

Model's state dict saved as ***deeplabv3.pth***

(b) Performance Evaluation

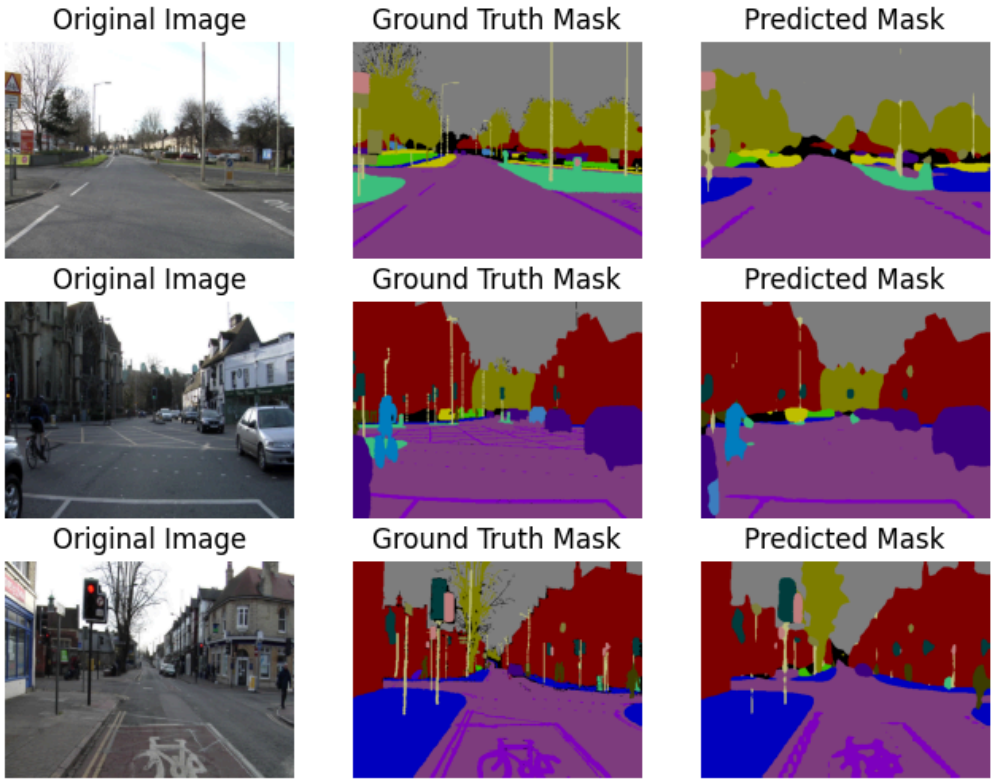
```
Pixel Accuracy: 0.8817
Mean IoU (mIoU): 0.3659

Class-wise Metrics:
Class 0: IoU=0.0000, Dice=0.0000, Precision=0.0000, Recall=0.0000
Class 1: IoU=0.0000, Dice=0.0000, Precision=0.0000, Recall=0.0000
Class 2: IoU=0.4604, Dice=0.6305, Precision=0.9345, Recall=0.4758
Class 3: IoU=0.2485, Dice=0.3981, Precision=0.9782, Recall=0.2499
Class 4: IoU=0.8431, Dice=0.9149, Precision=0.8773, Recall=0.9559
Class 5: IoU=0.8085, Dice=0.8941, Precision=0.8466, Recall=0.9473
Class 6: IoU=0.0000, Dice=0.0000, Precision=0.0000, Recall=0.0000
Class 7: IoU=0.0000, Dice=0.0000, Precision=0.0000, Recall=0.0000
Class 8: IoU=0.1434, Dice=0.2508, Precision=0.5185, Recall=0.1654
Class 9: IoU=0.5299, Dice=0.6928, Precision=0.6934, Recall=0.6921
Class 10: IoU=0.3887, Dice=0.5598, Precision=0.7211, Recall=0.4575
Class 11: IoU=0.0000, Dice=0.0000, Precision=0.0000, Recall=0.0000
Class 12: IoU=0.3540, Dice=0.5229, Precision=0.5947, Recall=0.4667
Class 13: IoU=0.0000, Dice=0.0000, Precision=0.0000, Recall=0.0000
Class 14: IoU=0.4276, Dice=0.5990, Precision=0.5526, Recall=0.6540
Class 15: IoU=0.3900, Dice=0.5611, Precision=0.8549, Recall=0.4176
Class 16: IoU=0.4044, Dice=0.5760, Precision=0.5887, Recall=0.5637
Class 17: IoU=0.9145, Dice=0.9553, Precision=0.9463, Recall=0.9646
Class 18: IoU=0.5162, Dice=0.6809, Precision=0.9262, Recall=0.5384
Class 19: IoU=0.7920, Dice=0.8839, Precision=0.8349, Recall=0.9390
Class 20: IoU=0.2989, Dice=0.4602, Precision=0.8748, Recall=0.3123
Class 21: IoU=0.9137, Dice=0.9549, Precision=0.9401, Recall=0.9702
Class 22: IoU=0.2391, Dice=0.3860, Precision=0.4827, Recall=0.3215
Class 23: IoU=0.0000, Dice=0.0000, Precision=0.0000, Recall=0.0000
Class 24: IoU=0.5217, Dice=0.6857, Precision=0.7886, Recall=0.6065
Class 25: IoU=0.0000, Dice=0.0000, Precision=0.0000, Recall=0.0000
Class 26: IoU=0.7705, Dice=0.8704, Precision=0.8720, Recall=0.8688
Class 27: IoU=0.1457, Dice=0.2544, Precision=0.7303, Recall=0.1540
Class 28: IoU=0.0000, Dice=0.0000, Precision=0.0000, Recall=0.0000
Class 29: IoU=0.5709, Dice=0.7268, Precision=0.7200, Recall=0.7338
Class 30: IoU=0.4720, Dice=0.6413, Precision=0.6936, Recall=0.5964
Class 31: IoU=0.5560, Dice=0.7147, Precision=0.8131, Recall=0.6375
```

(c) Visualization and Failure Analysis

c.1 Visualizing IoU ≤ 0.5 Cases

For each class, three images where the 0 < IoU ≤ 0.5 were selected but for some classes there are no examples which might be because class objects are not present in the test dataset. These images show the predicted masks compared to ground truth.

Example Figures:



c.2 Failure Analysis

Overall DeepLabv3 is performing better than segnet but it also struggles in some cases:

- **Occlusion:** Objects are partially visible.
- **Misclassification:** The model confuses similar-looking objects like bicyclist and pedestrians
- **Environmental Challenges:** Low lighting or complex backgrounds.
- **DeepLabV3** handles boundaries better but may still struggle with overlapping objects.

Object Detection

Introduction

Object detection is a computer vision task that involves identifying and locating objects within an image or video. It goes beyond simply recognizing that an object is present (as in image classification) — it determines *where* the objects are by drawing bounding boxes around them and assigning a class label to each detected object.

(a) Ultralytics API and COCO 2017 Validation Set

- Used the Ultralytics API to load the COCO 2017 validation set.
- Downloaded only the val2017 split.

(b) YOLOv8 Predictions on COCO val2017

- Used a COCO-pretrained YOLOv8 extra large model to make predictions on the COCO validation set.
- Saved predictions in coco_predictions.json in the standard COCO format.
- Reported mAP (Mean Average Precision) of the model.

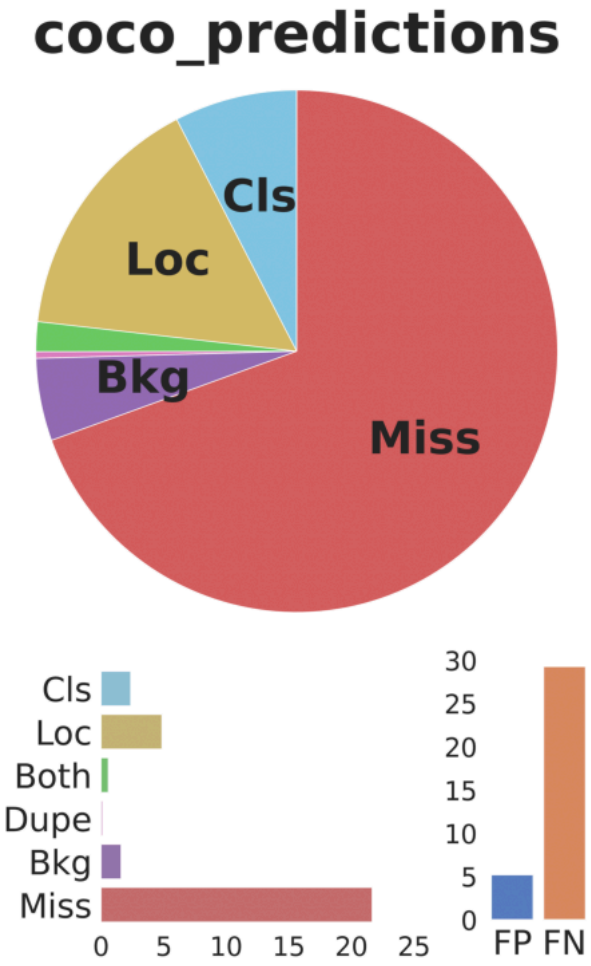
mAP Results:

```
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 ] = 0.525
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.284
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
```

(c) TIDE Toolbox Error Analysis

- Used the TIDE Toolbox to analyze prediction errors.
- Loaded the coco_predictions.json and ground truth annotations for the validation set

```
-- coco_predictions --

bbox AP @ 50: 61.62

                      Main Errors
============================================================
  Type      Cls      Loc     Both     Dupe      Bkg     Miss
------------------------------------------------------------
   dAP     2.37     4.86     0.57     0.13     1.59    21.65
============================================================

       Special Error
============================
  Type   FalsePos   FalseNeg
----------------------------
   dAP       5.19      29.26
============================
```



Analysis:
The most significant error came from missed detections, suggesting the model struggles to identify certain objects, potentially due to occlusion, small object sizes, or challenging backgrounds.

(d) Expected Calibration Error (ECE)

- Computed the Expected Calibration Error (ECE) using Equation 3 from the referenced paper.
- Compared predicted confidence scores with the correctness likelihood.

```
Expected Calibration Error (ECE), area = all : 0.073812
```

Analysis:
The ECE indicates that the model's confidence scores are reasonably calibrated but could be improved to better reflect true correctness likelihood.

(e) Object Scale-based Analysis

- Filtered the coco_predictions.json into three subsets based on object scale (small, medium, large).
- Calculated TIDE statistics and ECE for each scale.

TIDE Statistics by Object Scale:

Scale	Localization	Classification	Duplicates	False Positives	Missed Detections
Small	2.04	3.07	0.11	3.84	37.77
Medium	3.10	2.09	0.48	4.38	23.21
Large	3.56	2.83	0.79	1.33	12.56

ECE by Object Scale:

Scale	ECE
Small	0.080770
Medium	0.041552
Large	0.031818

Analysis:
The model performs significantly worse on small objects, with higher missed detection rates and calibration errors. This suggests that improving the model's ability to detect and properly score small objects could lead to substantial performance gains.

(f) Additional Observations and Comparisons

- **What can you infer from these observations?**

  The model's performance is strongly affected by object scale, with small objects posing the most significant challenge. This implies that either the feature extraction process or the model architecture may struggle with fine-grained details in small objects.

- **Comment on your observations across each of the three scales.**

  The small scale shows the highest missed detection and false negative rates, suggesting the model frequently overlooks small objects.

  Medium-sized objects perform better, with moderate error rates across all metrics.

  Large objects achieve the best performance, with low missed detection rates and the best calibration, indicating that larger, more distinguishable objects are much easier for the model to correctly detect and classify.

- **Compare these statistics with the relevant metrics computed with all objects.**

  Overall, the all-object statistics (mAP 61.62, ECE 0.073812) reflect a balance across scales, but breaking the data down reveals that small objects significantly drag down performance.

  The comparison shows that calibration error decreases with increasing object size, confirming that confidence estimation improves as the model handles larger and clearer objects.

Multi-Object Tracking

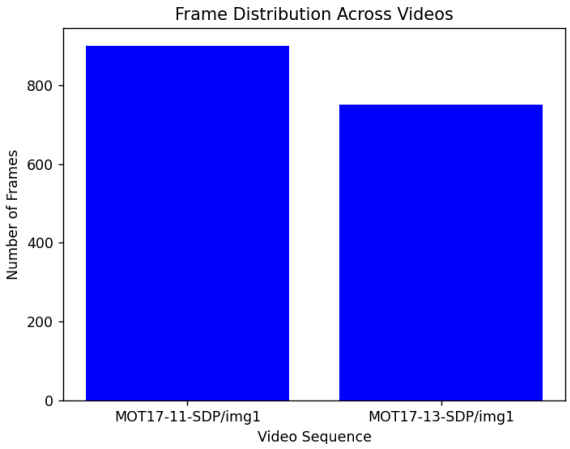
Introduction

Object tracking is a fundamental task in computer vision, particularly in surveillance, autonomous driving, and robotics. This report evaluates ByteTrack, a state-of-the-art multi-object tracker, using the MOT17 dataset. Additionally, a simple IoU-based tracker is implemented, and its performance is compared with ByteTrack.

(a) Dataset: MOT17

The MOT17 dataset is a widely used benchmark for multi-object tracking (MOT) tasks. It consists of multiple video sequences captured in real-world environments with pedestrian annotations. The dataset includes both training and test sets, with ground-truth bounding box annotations for evaluation.

Data Visualization



(b) ByteTrack Evaluation

ByteTrack is a high-performance multi-object tracker that efficiently associates detected objects over time. The algorithm leverages a two-stage association strategy using high-confidence and low-confidence detections.

```
PS F:\CV\Assignment 1\Tracking> python  "f:\CV\Assignment 1\Tracking\tracking.py"
Loaded 750 frames from video.
Detections generated: 7695
Byte Tracking results generated: 7064
Saved Byte tracking results to Byte.pkl
20202
11642
         IDF1   IDP   IDR  Rcll  Prcn  GT MT PT ML  FP   FN IDs   FM  MOTA  MOTP IDt IDa IDm
Overall 59.9% 79.2% 48.1% 55.6% 91.7% 110 43 20 47 586 5164 120  230 49.6% 0.203  55  57  20
Byte MOTA Score: 0.4958
PS F:\CV\Assignment 1\Tracking>
```

(c) Implementation of IoU Tracker and Evaluation

A simple Intersection over Union (IoU) tracker is implemented as a baseline comparison. The IoU tracker follows these steps:

1. Initializes new tracks when new detections appear.
2. Matches detections to existing tracks based on the highest IoU score.
3. Updates tracks frame by frame.
4. Handles occlusions by maintaining track states for a predefined number of frames before deletion.

```
PS F:\CV\Assignment 1\Tracking> python tracking.py
Loaded 750 frames from video.
Detections generated: 7695
IouTracker Tracking results generated: 7695
Saved IouTracker tracking results to IouTracker.pkl
20202
11642
        IDF1  IDP  IDR  Rcll  Prcn  GT MT PT ML  FP   FN  IDs   FM MOTA  MOTP IDt  IDa IDm
Overall 3.5% 4.5% 2.9% 57.6% 87.1% 110 47 25 38 993 4940 5161  375 4.7% 0.211  13 5148   0
IouTracker MOTA Score: 0.0471
PS F:\CV\Assignment 1\Tracking>
```

(d) Performance Comparison - MOT metrics

Metric	ByteTrack	IoU Tracker
IDF1	59.9%	3.5%
IDP	79.2%	4.5%
IDR	48.1%	2.9%
Recall	55.6%	57.6%
Precision	91.7%	87.1%
GT	110	110
MT	43	47
PT	20	25
ML	47	38
FP	586	993
FN	5164	4940
ID Swaps	120	5161
FM	230	375
MOTA	49.6%	4.7%
MOTP	0.203	0.211
IDt	55	13
IDa	57	5148
IDm	20	0

ByteTrack, as expected, demonstrates superior performance in accuracy and identity tracking due to its advanced association strategy. The IoU tracker, while being simpler, struggles with occlusions and ID switches.