ASSIGNMENT

Course Code: ECPE49

Course Name: Foundations of Artificial Intelligence

Submitted to: Dr Avik Hati

Submitted by:

1. **Subrahmanya Bharadwaaj B S - 108121126**
2. **J Isaac Samuel - 108121054**
3. **Sahaj Jain - 108121106**
4. **Pammy Thakur - 108121088**
5. **Nishmaie M. - 108121084**

Brief description of the work:

In this project, we worked with the Pascal VOC 2012 dataset to complete three tasks: denoising, classification, and segmentation. We added Gaussian noise to the dataset in two ways: with fixed mean and variance, and with varying variance. The goal was to apply deep learning methods to denoise the images while ensuring the denoised outputs matched the original image sizes. We calculated the Mean Squared Error (MSE) for the denoising task.

We conducted classification on all 20 classes, experimenting with different layers and applying regularization techniques to prevent overfitting. Specifically, we used weight decay and dropout to enhance model generalization. We also tried multi-label classification on the data to identify more than one object in the image.

Finally, we performed image segmentation, concentrating on 5 specific classes to accurately segment and differentiate various regions within the images. This involved training the model to identify and delineate distinct areas corresponding to the targeted classes, enabling more precise region-based classification within the images. The segmentation task helped enhance the model's ability to understand the spatial structure and composition of the visual data, making it capable of separating meaningful regions from the background.

Noising the dataset

Added gaussian noise using:

- mean = 0
- standard deviation = 1.

Noise is added and the noisy images are saved.

```python
def noise_addition(image, mean=0, sigma=1):
    noise = np.random.normal(mean, sigma, image.shape).astype("uint8")
    noise_image = cv2.add(image, noise)
    return noise_image
```

In addition, also implemented adding variable noise for every image.

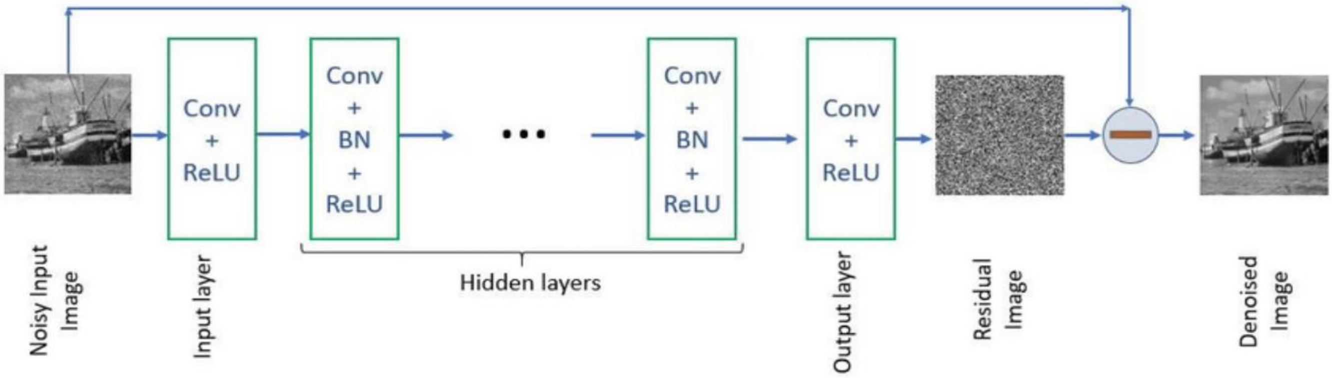
```python
def add_gaussian_noise(image, mean_range=(0, 50), sigma_range=(10, 50)):
    """Add Gaussian noise to an image with random mean and sigma."""
    mean = np.random.uniform(mean_range[0], mean_range[1])  # Random mean within the specified range
    sigma = np.random.uniform(sigma_range[0], sigma_range[1])  # Random sigma within the specified range

    gauss = np.random.normal(mean, sigma, image.shape).astype('uint8')  # Generate Gaussian noise
    noisy_image = cv2.add(image, gauss)  # Add noise to the image

    return noisy_image
```

Results:

Denoising the dataset

After introducing noise to the dataset, we apply DnCNN, a Denoising Convolutional Neural Network, to remove the noise. This model effectively cleans up the images, enhancing their clarity and making them look sharper.

DnCNN Architecture



Implementation of DnCNN Model

The below shown is the code that shows the implementation of the layers in the DnCNN Mode for Denoising the noisy images.

```
def _init_(self, depth=11, n_channels=64, image_channels=3, use_bnorm=True):
    super(DnCNN, self)._init_()
    layers = []

    # First layer: Convolution + ReLU
    layers.append(nn.Conv2d(in_channels=image_channels, out_channels=n_channels, kernel_size=3, padding=1, bias=True))
    layers.append(nn.ReLU(inplace=True))

    # Hidden layers: Convolution + BatchNorm + ReLU
    for _ in range(depth - 2):
        layers.append(nn.Conv2d(in_channels=n_channels, out_channels=n_channels, kernel_size=3, padding=1, bias=False))
        if use_bnorm:
            layers.append(nn.BatchNorm2d(n_channels))
        layers.append(nn.ReLU(inplace=True))

    # Last layer: Convolution (no activation)
    layers.append(nn.Conv2d(in_channels=n_channels, out_channels=image_channels, kernel_size=3, padding=1, bias=False))

    self.dncnn = nn.Sequential(*layers)
```

First Layer (Convolution + ReLU):

- Extracts low-level features (edges, textures, basic patterns).
- No Batch Normalization here because initial input statistics are important for denoising.
- ReLU ensures non-negative outputs, helping in learning meaningful features.

Hidden Layers (Conv + BN + ReLU):

- Multiple such layers form the backbone of feature extraction.
- Batch Normalization helps in stabilizing training and allowing higher learning rates.
- Each layer learns progressively more complex noise patterns.

Last Layer (Convolution only):

- Maps learned features back to image space
- Need to predict the residual (noise) directly

Output is subtracted from input to get a clean image.

```
def forward(self, x):
    noise = self.dncnn(x)
    return x - noise  # Subtract the noise from the input (residual learning)
```

Calculating the MSE:

```
# Training loop
num_epochs = 20
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for noisy_images, clean_images in train_loader:
        noisy_images, clean_images = noisy_images.to(device), clean_images.to(device)

        # Mixed precision training
        with autocast():
            outputs = model(noisy_images)
            loss = loss_fn(outputs, clean_images)

        # Backward pass and optimization
        optimizer.zero_grad()
        scaler.scale(loss).backward()  # Scale the loss
        scaler.step(optimizer)  # Update weights
        scaler.update()  # Update the scaler

        running_loss += loss.item()

    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/len(train_loader)}')
```

```
#loss function
loss_fn = nn.MSELoss()
```

Early stopping:

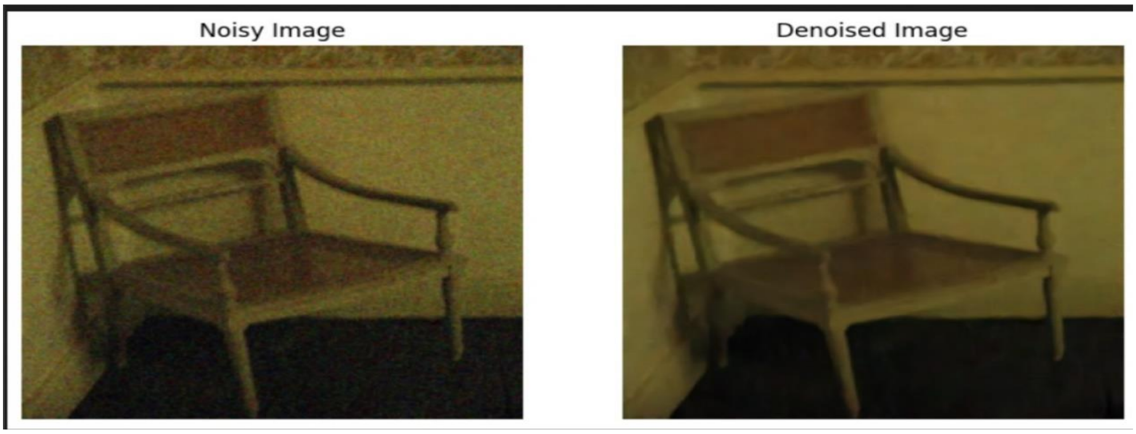
```python
# Early stopping check
if val_loss < best_val_loss:
    best_val_loss = val_loss
    epochs_without_improvement = 0
    torch.save(model.state_dict(), 'best_dncnn_denoising_model.pth')
    print("Validation loss improved, saving model...")
else:
    epochs_without_improvement += 1
    if epochs_without_improvement >= patience:
        print(f"Early stopping triggered after {epoch+1} epochs.")
        break
```

If the current validation loss is lower than the best-recorded loss, it updates the best validation loss, resets the counter for epochs without improvement to zero, saves the model's current state to 'best_dncnn_denoising_model.pth, and prints a message confirming the improvement. If there's no improvement, the counter for epochs without improvement is incremented. If this counter reaches a preset limit (patience), early stopping is triggered to prevent overfitting and avoid wasting time on further training.

Results:

Visual Result:

1. **With fixed mean and standard variance for the whole dataset:**



2. With variable mean and standard variance for the whole dataset

Quantitative Result:

Noise with constant mean and variance

```
  scaler = GradScaler()
C:\Users\bhara\AppData\Local\Temp\ipykernel_1844\40
  with autocast():
Training on: cuda
Epoch [1/20], Loss: 0.001222935271063602
Validation Loss: 0.0006134171264928354
Validation loss improved, saving model...
Epoch [2/20], Loss: 0.0005348627789483072
Validation Loss: 0.0004712933541456952
Validation loss improved, saving model...
Epoch [3/20], Loss: 0.0004785154015463131
Validation Loss: 0.00045817418428013363
Validation loss improved, saving model...
Epoch [4/20], Loss: 0.0004561777671681799
Validation Loss: 0.000422690263550003
Validation loss improved, saving model...
Epoch [5/20], Loss: 0.0004358338442568185
Validation Loss: 0.00048386917444796107
Epoch [6/20], Loss: 0.00042205549739165245
Validation Loss: 0.0004032493449157364
Validation loss improved, saving model...
Epoch [7/20], Loss: 0.0003986938732054193
Validation Loss: 0.0003878436410707573
Validation loss improved, saving model...
Epoch [8/20], Loss: 0.0003898876624217556
Validation Loss: 0.0003770714057853099
Validation loss improved, saving model...
Epoch [9/20], Loss: 0.0003815196184093051
...
Epoch [11/20], Loss: 0.00037030193023808493
Validation Loss: 0.0004438837754288363
Early stopping triggered after 11 epochs.
Test Loss: 0.00044721723781210955
```

Noise with varying mean and variance

```
  with autocast():
Training on: cuda
Epoch [1/25], Loss: 0.03929414845170139
Validation Loss: 0.03244186410349663
Validation loss improved, saving model...
Epoch [2/25], Loss: 0.030744492521066973
Validation Loss: 0.028918517379212045
Validation loss improved, saving model...
Epoch [3/25], Loss: 0.027692487308950155
Validation Loss: 0.026651067820281905
Validation loss improved, saving model...
Epoch [4/25], Loss: 0.026080085952498205
Validation Loss: 0.025644966657926267
Validation loss improved, saving model...
Epoch [5/25], Loss: 0.024950043759344096
Validation Loss: 0.02613378043730404
Epoch [6/25], Loss: 0.024177204626702073
Validation Loss: 0.024390218309431434
Validation loss improved, saving model...
Epoch [7/25], Loss: 0.0230871524903173
Validation Loss: 0.022712643880605975
Validation loss improved, saving model...
Epoch [8/25], Loss: 0.022650276769063518
Validation Loss: 0.02380830265372713
Epoch [9/25], Loss: 0.022168334148893805
Validation Loss: 0.02382871952499742
Epoch [10/25], Loss: 0.02181605085289085
Validation Loss: 0.024353306166061732
Early stopping triggered after 10 epochs.
Test Loss: 0.023903859207449956
```

Noise parameters:

Varying mean and variance

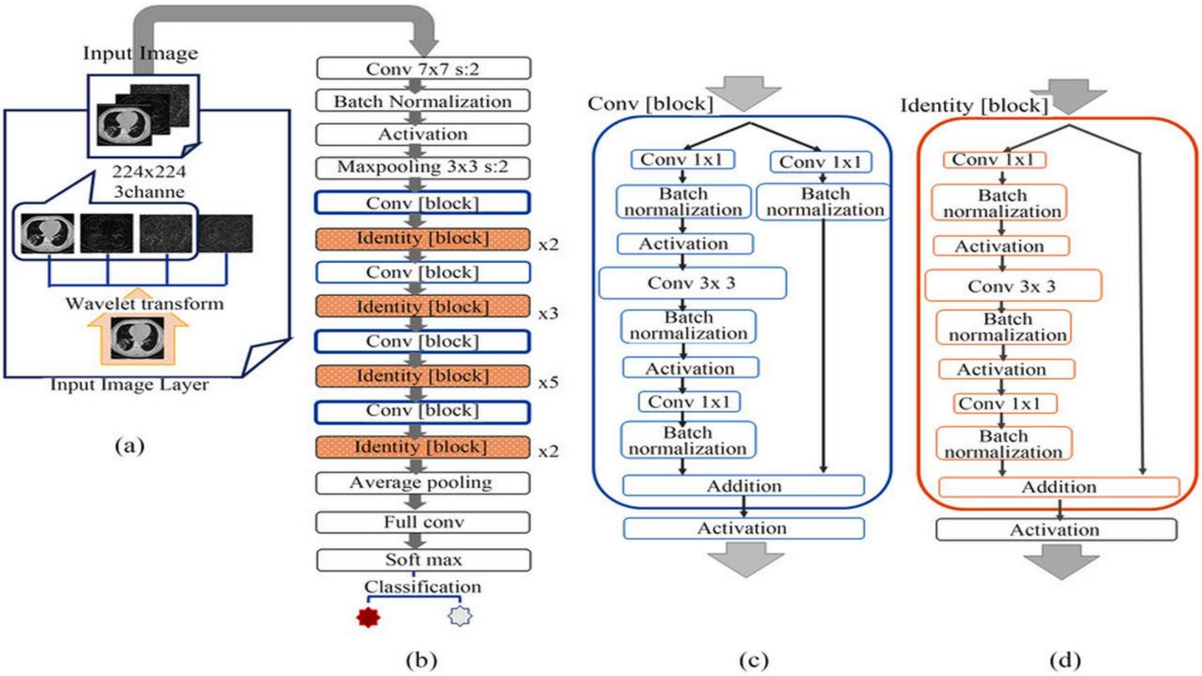
```
2007_000027.jpg | Noise Std: 0.0674
2007_000032.jpg | Noise Std: 0.0800
2007_000033.jpg | Noise Std: 0.0736
2007_000039.jpg | Noise Std: 0.1023
2007_000042.jpg | Noise Std: 0.0687
2007_000061.jpg | Noise Std: 0.0563
2007_000063.jpg | Noise Std: 0.0739
2007_000068.jpg | Noise Std: 0.0948
2007_000121.jpg | Noise Std: 0.0568
2007_000123.jpg | Noise Std: 0.0689
2007_000129.jpg | Noise Std: 0.1027
2007_000170.jpg | Noise Std: 0.0519
2007_000175.jpg | Noise Std: 0.0590
2007_000187.jpg | Noise Std: 0.0695
2007_000241.jpg | Noise Std: 0.0978
2007_000243.jpg | Noise Std: 0.0789
2007_000250.jpg | Noise Std: 0.0992
2007_000256.jpg | Noise Std: 0.0545
2007_000272.jpg | Noise Std: 0.0792
2007_000323.jpg | Noise Std: 0.0425
2007_000332.jpg | Noise Std: 0.1176
2007_000333.jpg | Noise Std: 0.0872
2007_000346.jpg | Noise Std: 0.1118
2007_000363.jpg | Noise Std: 0.0710
```

Constant mean and variance.

```
2008_000001.jpg | Noise Std: 0.0848
2008_000004.jpg | Noise Std: 0.0670
2008_000005.jpg | Noise Std: 0.0741
2008_000006.jpg | Noise Std: 0.0605
2008_000010.jpg | Noise Std: 0.0704
2008_000011.jpg | Noise Std: 0.0806
2008_000012.jpg | Noise Std: 0.0717
2008_000013.jpg | Noise Std: 0.1225
2008_000014.jpg | Noise Std: 0.0954
2008_000017.jpg | Noise Std: 0.0808
2008_000018.jpg | Noise Std: 0.0747
2008_000020.jpg | Noise Std: 0.0668
2008_000022.jpg | Noise Std: 0.0964
2008_000024.jpg | Noise Std: 0.0839
2008_000025.jpg | Noise Std: 0.0793
2008_000029.jpg | Noise Std: 0.0908
2008_000030.jpg | Noise Std: 0.0893
2008_000031.jpg | Noise Std: 0.0809
2008_000035.jpg | Noise Std: 0.0732
2008_000038.jpg | Noise Std: 0.0779
2008_000039.jpg | Noise Std: 0.0975
2008_000040.jpg | Noise Std: 0.0763
2008_000044.jpg | Noise Std: 0.0800
2008_000046.jpg | Noise Std: 0.0914
2008_000047.jpg | Noise Std: 0.0813
```

Image Classification

Architecture diagram



Implementation details

Libraries and Modules:

- **os, xml.etree.ElementTree (ET):** Used for file handling and parsing XML annotations.
- **PIL (Image):** To handle and manipulate images.
- **torch, torchvision, torch.nn, torch.optim:** Libraries for deep learning using PyTorch.
- **sklearn.metrics:** Used for computing confusion matrix and classification report.
- **matplotlib, seaborn:** For visualizing the confusion matrix.
- **numpy:** For array manipulation.

PascalVOC Dataset Class:

PascalVOCDataset is a custom dataset class inherits from PyTorch's 'Dataset' class and is used to load images and their corresponding labels from the Pascal VOC 2012 dataset.

- **__init__:** Initializes the dataset, defines class names, and loads the image paths and labels from the dataset.
- **load_data():** Loads image paths and annotations from the Pascal VOC dataset, particularly from the `train` and `val` sets.
- **parse_annotation():** Extracts the label (class) of the first object from an XML annotation file.
- **__len__:** Returns the length of the dataset (number of images).
- **__getitem__:** Retrieves an image and its label, applying transformations if specified.

Data Loading:

- **transform:** Defines the transformations (resize and convert to tensor) applied to the images.
- **train_loader** and **val_loader:** Data loader instances to load the training and validation datasets in batches.

Model Definition:

- **ResNet-n:** Pretrained ResNet-n model from `torchvision.models`. The final layer is replaced to adapt the model to the 20 classes in Pascal VOC. (We have used Resnet-34,50,101 in this project)
- **Loss and Optimizer:** Uses `CrossEntropyLoss` for classification and the Adam optimizer to train the model on single label classification.

Training Loop:

```
# Training loop
num_epochs = 20  # Set the number of epochs
for epoch in range(num_epochs):
    model.train()  # Set the model to training mode
    running_loss = 0.0
    correct = 0
    total = 0

    for images, targets in train_loader:
        images, targets = images.to(device), targets.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, targets)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        # Calculate training accuracy
        _, predicted = torch.max(outputs, 1)  # Get class with highest score
        correct += (predicted == targets).sum().item()
        total += targets.size(0)

    # Print epoch loss and accuracy
    print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {running_loss / len(train_loader):.4f}, Accuracy: {correct / total:.4f}')
```

Epochs: Runs for 20 epochs, and each epoch consists of training the model on all training images.

Accuracy Calculation: After each batch, the accuracy is calculated by comparing predicted labels with true labels.

Backpropagation: The model is updated using backpropagation and the optimizer after each forward pass.

Evaluation:

```
# Evaluation loop
all_preds = []
all_targets = []

model.eval()  # Set the model to evaluation mode
with torch.no_grad():
    for images, targets in val_loader:
        images, targets = images.to(device), targets.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)  # Get class with highest score

        all_preds.extend(predicted.cpu().numpy())  # Store predicted labels
        all_targets.extend(targets.cpu().numpy())  # Store true labels
```

Evaluation Loop: After training, the model is evaluated on the validation set. No gradients are calculated here.

Confusion Matrix: A confusion matrix is computed and visualized using seaborn to see how well the model predicts each class.

Classification Report: Displays precision, recall, and F1-score for each class using ‘classification_report()’.

Results

ResNet-50

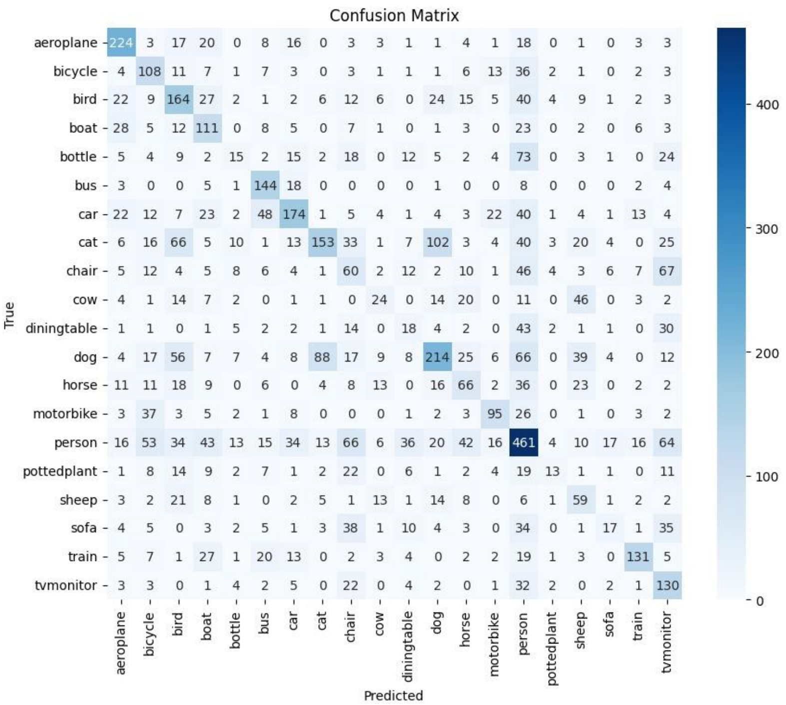
Without weight decay.

```
  warnings.warn(msg)
Epoch [1/20], Loss: 2.4983, Accuracy: 0.2302
Epoch [2/20], Loss: 2.1861, Accuracy: 0.3037
Epoch [3/20], Loss: 2.1073, Accuracy: 0.3227
Epoch [4/20], Loss: 1.9740, Accuracy: 0.3713
Epoch [5/20], Loss: 1.8539, Accuracy: 0.4042
Epoch [6/20], Loss: 1.7669, Accuracy: 0.4254
Epoch [7/20], Loss: 1.6589, Accuracy: 0.4581
Epoch [8/20], Loss: 1.5410, Accuracy: 0.5001
Epoch [9/20], Loss: 1.4403, Accuracy: 0.5312
Epoch [10/20], Loss: 1.3312, Accuracy: 0.5597
Epoch [11/20], Loss: 1.1706, Accuracy: 0.6096
Epoch [12/20], Loss: 1.0126, Accuracy: 0.6601
Epoch [13/20], Loss: 0.8498, Accuracy: 0.7252
Epoch [14/20], Loss: 0.6968, Accuracy: 0.7665
Epoch [15/20], Loss: 0.5312, Accuracy: 0.8239
Epoch [16/20], Loss: 0.4124, Accuracy: 0.8613
Epoch [17/20], Loss: 0.3540, Accuracy: 0.8874
Epoch [18/20], Loss: 0.2524, Accuracy: 0.9197
Epoch [19/20], Loss: 0.2275, Accuracy: 0.9236
Epoch [20/20], Loss: 0.2287, Accuracy: 0.9255
Validation Accuracy: 0.4089
```

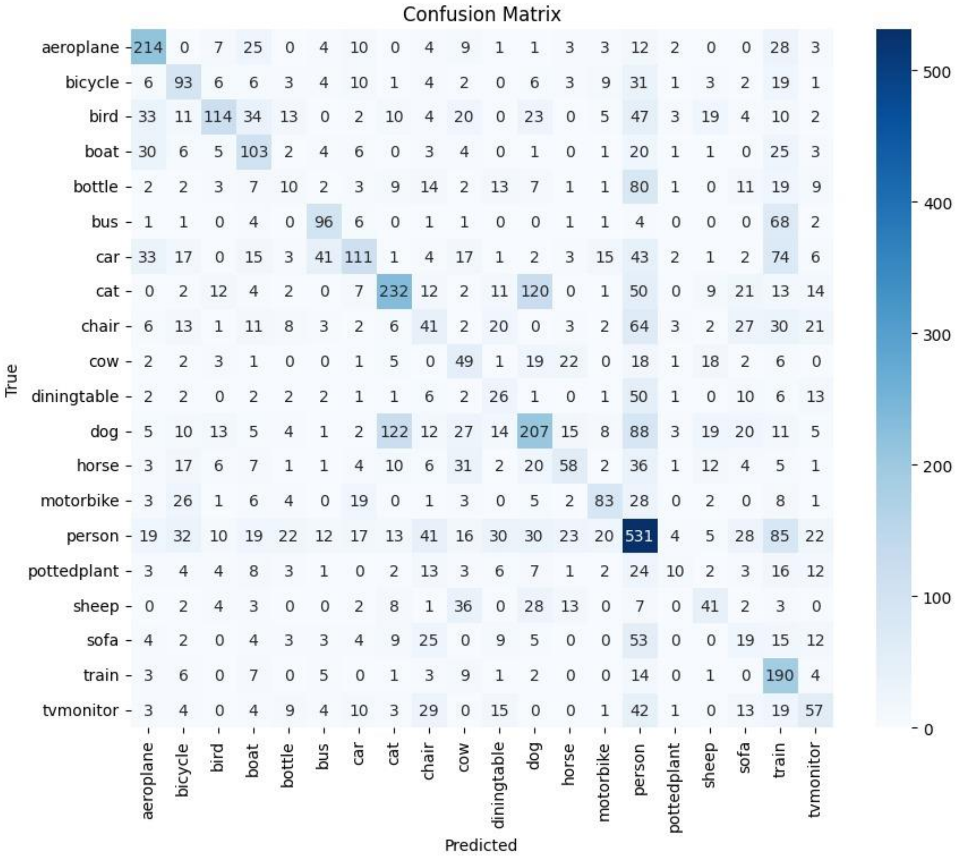
With weight decay.

```
  warnings.warn(msg)
Epoch [1/50], Loss: 2.5601, Accuracy: 0.2270
Epoch [2/50], Loss: 2.2503, Accuracy: 0.2874
Epoch [3/50], Loss: 2.1358, Accuracy: 0.3162
Epoch [4/50], Loss: 2.0544, Accuracy: 0.3409
Epoch [5/50], Loss: 1.9583, Accuracy: 0.3736
Epoch [6/50], Loss: 1.8773, Accuracy: 0.3974
Epoch [7/50], Loss: 1.8249, Accuracy: 0.4174
Epoch [8/50], Loss: 1.7586, Accuracy: 0.4308
Epoch [9/50], Loss: 1.6865, Accuracy: 0.4581
Epoch [10/50], Loss: 1.5962, Accuracy: 0.4812
Epoch [11/50], Loss: 1.5488, Accuracy: 0.4889
Epoch [12/50], Loss: 1.4604, Accuracy: 0.5284
Epoch [13/50], Loss: 1.3616, Accuracy: 0.5538
Epoch [14/50], Loss: 1.2503, Accuracy: 0.5917
Epoch [15/50], Loss: 1.1156, Accuracy: 0.6297
Epoch [16/50], Loss: 1.0073, Accuracy: 0.6682
Epoch [17/50], Loss: 0.8811, Accuracy: 0.7086
Epoch [18/50], Loss: 0.7366, Accuracy: 0.7541
Epoch [19/50], Loss: 0.6576, Accuracy: 0.7800
Epoch [20/50], Loss: 0.5736, Accuracy: 0.8057
Epoch [21/50], Loss: 0.4774, Accuracy: 0.8426
Epoch [22/50], Loss: 0.3890, Accuracy: 0.8711
Epoch [23/50], Loss: 0.3697, Accuracy: 0.8769
Epoch [24/50], Loss: 0.3300, Accuracy: 0.8923
Epoch [25/50], Loss: 0.2683, Accuracy: 0.9174
...
Epoch [48/50], Loss: 0.1556, Accuracy: 0.9540
Epoch [49/50], Loss: 0.1588, Accuracy: 0.9507
Epoch [50/50], Loss: 0.1823, Accuracy: 0.9411
Validation Accuracy: 0.3924
```

Confusion matrix (without weight decay)



Confusion matrix (with weight decay)



Classification report (without weight decay)

	precision	recall	f1-score	support
aeroplane	0.60	0.69	0.64	326
bicycle	0.34	0.51	0.41	210
bird	0.36	0.46	0.41	354
boat	0.34	0.52	0.41	215
bottle	0.19	0.08	0.11	196
bus	0.50	0.77	0.61	186
car	0.54	0.45	0.49	391
cat	0.55	0.30	0.39	512
chair	0.18	0.23	0.20	265
cow	0.28	0.16	0.20	150
diningtable	0.15	0.14	0.14	128
dog	0.50	0.36	0.42	591
horse	0.30	0.29	0.30	227
motorbike	0.54	0.49	0.52	192
person	0.43	0.47	0.45	979
pottedplant	0.35	0.10	0.16	124
sheep	0.26	0.39	0.31	150
sofa	0.30	0.10	0.15	167
train	0.68	0.53	0.60	246
tvmonitor	0.30	0.61	0.40	214
accuracy			0.41	5823
macro avg	0.38	0.38	0.37	5823
weighted avg	0.42	0.41	0.40	5823

Classification report (with weight)

	precision	recall	f1-score	support
aeroplane	0.58	0.66	0.61	326
bicycle	0.37	0.44	0.40	210
bird	0.60	0.32	0.42	354
boat	0.37	0.48	0.42	215
bottle	0.11	0.05	0.07	196
bus	0.52	0.52	0.52	186
car	0.51	0.28	0.37	391
cat	0.54	0.45	0.49	512
chair	0.18	0.15	0.17	265
cow	0.21	0.33	0.25	150
diningtable	0.17	0.20	0.19	128
dog	0.43	0.35	0.39	591
horse	0.39	0.26	0.31	227
motorbike	0.54	0.43	0.48	192
person	0.43	0.54	0.48	979
pottedplant	0.29	0.08	0.13	124
sheep	0.30	0.27	0.29	150
sofa	0.11	0.11	0.11	167
train	0.29	0.77	0.42	246
tvmonitor	0.30	0.27	0.28	214
accuracy			0.39	5823
macro avg	0.36	0.35	0.34	5823
weighted avg	0.40	0.39	0.38	5823

<u>ResNet – 101</u>

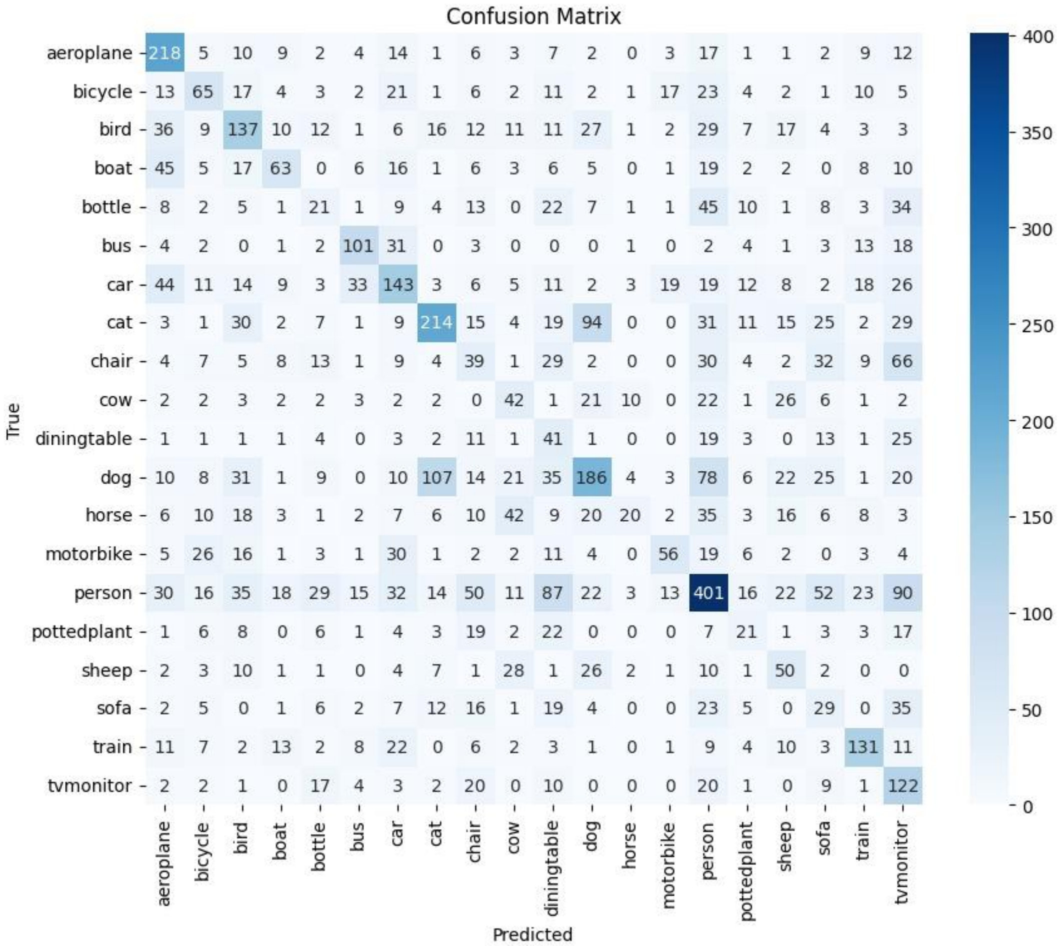
With weight decay

```
c:\Users\bhara\Desktop\AIAss\.venv\Lib\site-packages\t
  warnings.warn(
c:\Users\bhara\Desktop\AIAss\.venv\Lib\site-packages\t
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resn
100%|██████████| 171M/171M [13:45<00:00, 217kB/s]
Epoch [1/20], Loss: 2.6354, Accuracy: 0.2003
Epoch [2/20], Loss: 2.3670, Accuracy: 0.2489
Epoch [3/20], Loss: 2.2746, Accuracy: 0.2757
Epoch [4/20], Loss: 2.1553, Accuracy: 0.3075
Epoch [5/20], Loss: 2.0833, Accuracy: 0.3299
Epoch [6/20], Loss: 1.9859, Accuracy: 0.3675
Epoch [7/20], Loss: 1.9125, Accuracy: 0.3874
Epoch [8/20], Loss: 1.8319, Accuracy: 0.4116
Epoch [9/20], Loss: 1.7574, Accuracy: 0.4315
Epoch [10/20], Loss: 1.6790, Accuracy: 0.4460
Epoch [11/20], Loss: 1.5867, Accuracy: 0.4796
Epoch [12/20], Loss: 1.4714, Accuracy: 0.5108
Epoch [13/20], Loss: 1.3586, Accuracy: 0.5517
Epoch [14/20], Loss: 1.2347, Accuracy: 0.5882
Epoch [15/20], Loss: 1.0914, Accuracy: 0.6381
Epoch [16/20], Loss: 0.9140, Accuracy: 0.6911
Epoch [17/20], Loss: 0.7639, Accuracy: 0.7497
Epoch [18/20], Loss: 0.6083, Accuracy: 0.7941
Epoch [19/20], Loss: 0.4774, Accuracy: 0.8459
Epoch [20/20], Loss: 0.3906, Accuracy: 0.8704
```

Without weight decay

```
  warnings.warn(msg)
Epoch [1/20], Loss: 2.6286, Accuracy: 0.1922
Epoch [2/20], Loss: 2.3454, Accuracy: 0.2526
Epoch [3/20], Loss: 2.3126, Accuracy: 0.2566
Epoch [4/20], Loss: 2.1787, Accuracy: 0.2998
Epoch [5/20], Loss: 2.1064, Accuracy: 0.3248
Epoch [6/20], Loss: 2.0363, Accuracy: 0.3465
Epoch [7/20], Loss: 1.9785, Accuracy: 0.3689
Epoch [8/20], Loss: 1.9187, Accuracy: 0.3902
Epoch [9/20], Loss: 1.8735, Accuracy: 0.3988
Epoch [10/20], Loss: 1.8136, Accuracy: 0.4186
Epoch [11/20], Loss: 1.7629, Accuracy: 0.4266
Epoch [12/20], Loss: 1.7159, Accuracy: 0.4474
Epoch [13/20], Loss: 1.6559, Accuracy: 0.4564
Epoch [14/20], Loss: 1.5981, Accuracy: 0.4752
Epoch [15/20], Loss: 1.5064, Accuracy: 0.4985
Epoch [16/20], Loss: 1.4412, Accuracy: 0.5178
Epoch [17/20], Loss: 1.3247, Accuracy: 0.5559
Epoch [18/20], Loss: 1.2509, Accuracy: 0.5790
Epoch [19/20], Loss: 1.1092, Accuracy: 0.6297
Epoch [20/20], Loss: 0.9971, Accuracy: 0.6656
```

Confusion matrix for ResNet-101



Classification report

```
             precision    recall  f1-score   support

  aeroplane       0.49      0.67      0.56       326
    bicycle       0.34      0.31      0.32       210
       bird       0.38      0.39      0.38       354
       boat       0.43      0.29      0.35       215
     bottle       0.15      0.11      0.12       196
        bus       0.54      0.54      0.54       186
        car       0.37      0.37      0.37       391
        cat       0.54      0.42      0.47       512
      chair       0.15      0.15      0.15       265
        cow       0.23      0.28      0.25       150
diningtable       0.12      0.32      0.17       128
        dog       0.44      0.31      0.37       591
      horse       0.43      0.09      0.15       227
  motorbike       0.47      0.29      0.36       192
     person       0.47      0.41      0.44       979
pottedplant       0.17      0.17      0.17       124
      sheep       0.25      0.33      0.29       150
       sofa       0.13      0.17      0.15       167
      train       0.53      0.53      0.53       246
  tvmonitor       0.23      0.57      0.33       214

   accuracy                           0.36      5823
  macro avg       0.34      0.34      0.32      5823
weighted avg       0.39      0.36      0.36      5823
```

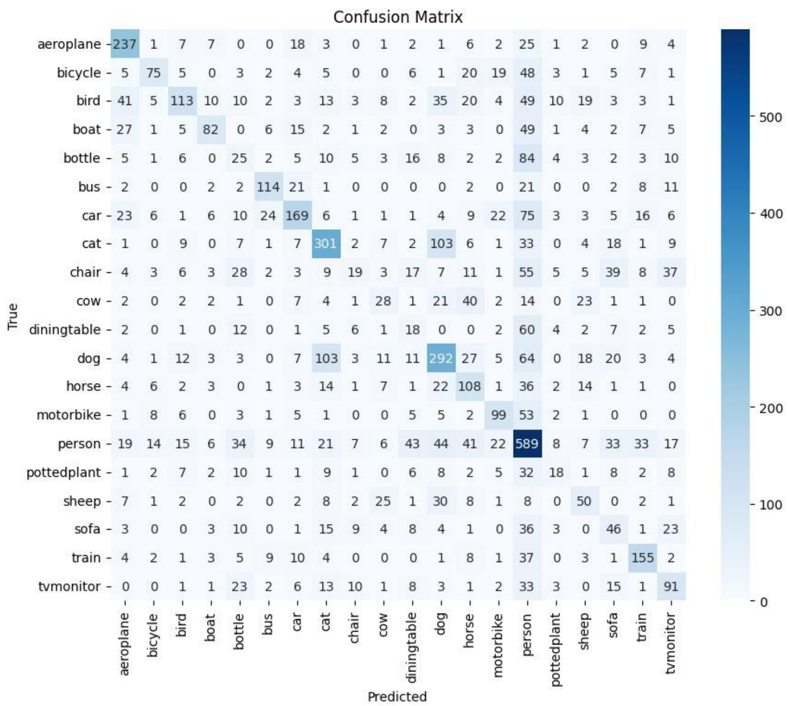
ResNet-34

Without weight decay

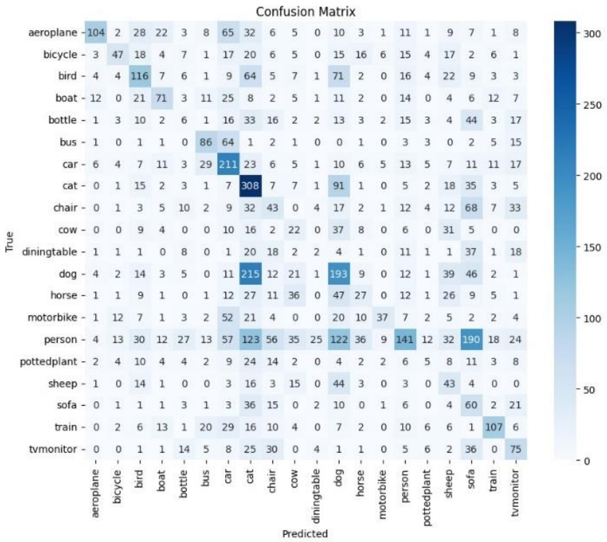
```
  warnings.warn(msg)
Epoch [1/30], Loss: 2.5914, Accuracy: 0.2204
Epoch [2/30], Loss: 2.1823, Accuracy: 0.3117
Epoch [3/30], Loss: 1.9641, Accuracy: 0.3768
Epoch [4/30], Loss: 1.8044, Accuracy: 0.4331
Epoch [5/30], Loss: 1.6084, Accuracy: 0.4903
Epoch [6/30], Loss: 1.4509, Accuracy: 0.5435
Epoch [7/30], Loss: 1.2817, Accuracy: 0.5823
Epoch [8/30], Loss: 1.0729, Accuracy: 0.6517
Epoch [9/30], Loss: 0.8549, Accuracy: 0.7243
Epoch [10/30], Loss: 0.6645, Accuracy: 0.7842
Epoch [11/30], Loss: 0.4837, Accuracy: 0.8393
Epoch [12/30], Loss: 0.3872, Accuracy: 0.8727
Epoch [13/30], Loss: 0.3160, Accuracy: 0.8940
Epoch [14/30], Loss: 0.2300, Accuracy: 0.9267
Epoch [15/30], Loss: 0.2165, Accuracy: 0.9300
Epoch [16/30], Loss: 0.1713, Accuracy: 0.9442
Epoch [17/30], Loss: 0.1684, Accuracy: 0.9444
Epoch [18/30], Loss: 0.1817, Accuracy: 0.9402
Epoch [19/30], Loss: 0.1348, Accuracy: 0.9554
Epoch [20/30], Loss: 0.1400, Accuracy: 0.9556
Epoch [21/30], Loss: 0.1299, Accuracy: 0.9559
Epoch [22/30], Loss: 0.1109, Accuracy: 0.9655
Epoch [23/30], Loss: 0.0841, Accuracy: 0.9717
Epoch [24/30], Loss: 0.1225, Accuracy: 0.9599
Epoch [25/30], Loss: 0.1021, Accuracy: 0.9668
...
Epoch [28/30], Loss: 0.0908, Accuracy: 0.9727
Epoch [29/30], Loss: 0.0639, Accuracy: 0.9811
Epoch [30/30], Loss: 0.0644, Accuracy: 0.9792
Validation Accuracy: 0.4515
```

```
Epoch [1/30], Loss: 2.6755, Accuracy: 0.2029
Epoch [2/30], Loss: 2.4544, Accuracy: 0.2384
Epoch [3/30], Loss: 2.3395, Accuracy: 0.2529
Epoch [4/30], Loss: 2.3112, Accuracy: 0.2653
Epoch [5/30], Loss: 2.2246, Accuracy: 0.2939
Epoch [6/30], Loss: 2.1800, Accuracy: 0.3077
Epoch [7/30], Loss: 2.1235, Accuracy: 0.3225
Epoch [8/30], Loss: 2.0910, Accuracy: 0.3330
Epoch [9/30], Loss: 2.0259, Accuracy: 0.3484
Epoch [10/30], Loss: 1.9850, Accuracy: 0.3588
Epoch [11/30], Loss: 1.9468, Accuracy: 0.3694
Epoch [12/30], Loss: 1.8995, Accuracy: 0.3878
Epoch [13/30], Loss: 1.8224, Accuracy: 0.4049
Epoch [14/30], Loss: 1.8052, Accuracy: 0.4275
Epoch [15/30], Loss: 1.7469, Accuracy: 0.4322
Epoch [16/30], Loss: 1.6456, Accuracy: 0.4586
Epoch [17/30], Loss: 1.6091, Accuracy: 0.4765
Epoch [18/30], Loss: 1.5295, Accuracy: 0.4957
Epoch [19/30], Loss: 1.4687, Accuracy: 0.5136
Epoch [20/30], Loss: 1.3396, Accuracy: 0.5681
Epoch [21/30], Loss: 1.2643, Accuracy: 0.5851
Epoch [22/30], Loss: 1.2044, Accuracy: 0.6015
Epoch [23/30], Loss: 1.0793, Accuracy: 0.6454
Epoch [24/30], Loss: 1.0371, Accuracy: 0.6468
Epoch [25/30], Loss: 0.9997, Accuracy: 0.6624
...
Epoch [28/30], Loss: 0.7808, Accuracy: 0.7411
Epoch [29/30], Loss: 0.7390, Accuracy: 0.7486
Epoch [30/30], Loss: 0.6604, Accuracy: 0.7803
Validation Accuracy: 0.2926
```

Confusion matrix (without weight decay)



Confusion matrix (with weight decay)



Classification report (without weight decay)

	precision	recall	f1-score	support
aeroplane	0.60	0.73	0.66	326
bicycle	0.60	0.36	0.45	210
bird	0.56	0.32	0.41	354
boat	0.62	0.38	0.47	215
bottle	0.13	0.13	0.13	196
bus	0.65	0.61	0.63	186
car	0.57	0.43	0.49	391
cat	0.55	0.59	0.57	512
chair	0.27	0.07	0.11	265
cow	0.26	0.19	0.22	150
diningtable	0.12	0.14	0.13	128
dog	0.49	0.49	0.49	591
horse	0.34	0.48	0.40	227
motorbike	0.52	0.52	0.52	192
person	0.42	0.60	0.49	979
pottedplant	0.27	0.15	0.19	124
sheep	0.31	0.33	0.32	150
sofa	0.22	0.28	0.25	167
train	0.59	0.63	0.61	246
tvmonitor	0.39	0.43	0.41	214
accuracy			0.45	5823
macro avg	0.42	0.39	0.40	5823
weighted avg	0.46	0.45	0.44	5823

Classification report (with weight decay)

	precision	recall	f1-score	support
aeroplane	0.72	0.32	0.44	326
bicycle	0.48	0.22	0.31	210
bird	0.36	0.33	0.34	354
boat	0.43	0.33	0.37	215
bottle	0.06	0.03	0.04	196
bus	0.47	0.46	0.46	186
car	0.34	0.54	0.42	391
cat	0.29	0.60	0.39	512
chair	0.16	0.16	0.16	265
cow	0.13	0.15	0.14	150
diningtable	0.05	0.02	0.02	128
dog	0.27	0.33	0.29	591
horse	0.20	0.12	0.15	227
motorbike	0.58	0.19	0.29	192
person	0.44	0.14	0.22	979
pottedplant	0.08	0.04	0.05	124
sheep	0.15	0.29	0.20	150
sofa	0.10	0.36	0.16	167
train	0.56	0.43	0.49	246
tvmonitor	0.28	0.35	0.31	214
accuracy			0.29	5823
macro avg	0.31	0.27	0.26	5823
weighted avg	0.34	0.29	0.28	5823

Examples:

```
C:\Users\bhara\AppData\Local\Temp\ipykernel_27416\1018871293.py:31: Fu
  model.load_state_dict(torch.load(model_path, map_location=device))
Predicted class: cow
```

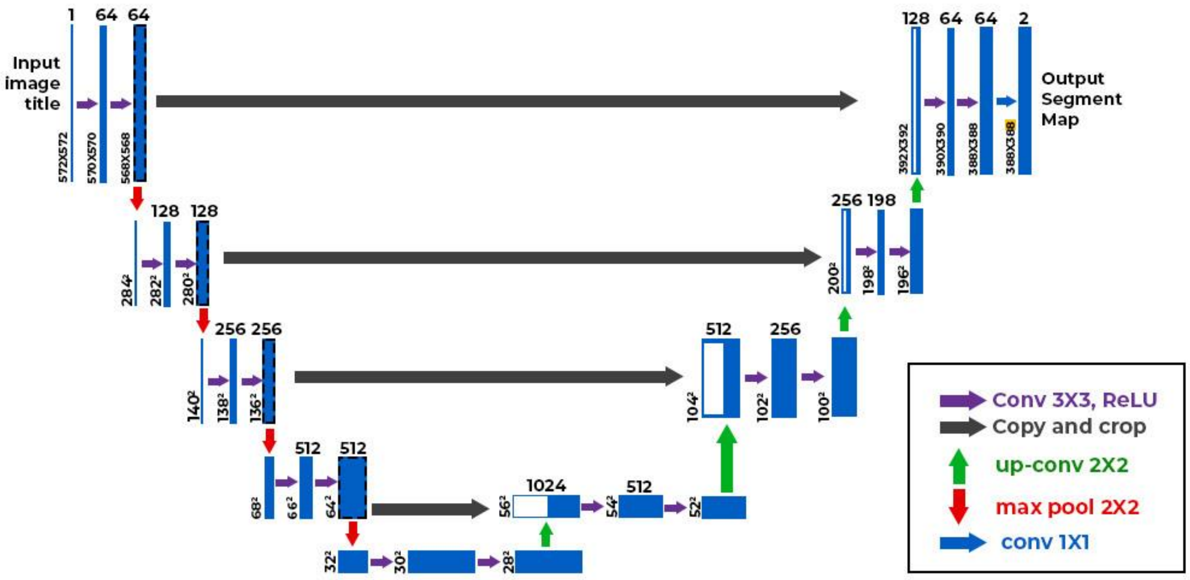
Predicted: cow

```
c:\Users\bhara\Desktop\AIAss\.venv\Lib\site-packages\torchvision
  warnings.warn(
c:\Users\bhara\Desktop\AIAss\.venv\Lib\site-packages\torchvision
  warnings.warn(msg)
C:\Users\bhara\AppData\Local\Temp\ipykernel_27416\1115911172.py:
  model.load_state_dict(torch.load(model_path, map_location=devi
Predicted class: train
```

Predicted: train

Image Segmentation

Architecture



Implementation details

This code is a complete pipeline for training a U-Net model for image segmentation using the Pascal VOC 2012 dataset.

Libraries and Layers

You import several necessary TensorFlow and Keras layers such as ‘Conv2D’, ‘Conv2DTranspose’, and ‘concatenate’. These are the building blocks for the U-Net architecture, where ‘Conv2D’ is used for down sampling, ‘Conv2DTranspose’ is used for up sampling, and ‘concatenate’ helps to merge feature maps between the encoder and decoder.

Data Loading and Preprocessing

- File Paths: You define paths to the images and their corresponding segmentation masks.
- Dataset Creation: The code creates a ‘tf.data.Dataset’ for loading images and masks using ‘tf.data.Dataset.list_files()’. The `dataset` is then shuffled using the ‘shuffle()’ method to randomize the dataset while maintaining the image-mask pair.
- Remapping Masks: The segmentation masks may have specific pixel values for each class. The ‘remap_mask’ function normalizes those pixel values (e.g., converting [0, 64, 128, 192, 224] to [0, 1, 2, 3, 4]). This step ensures the labels are mapped correctly for the model.
- Preprocessing: The ‘process_path’ function loads and decodes images, resizing them to a fixed size (96x128 in this case). The `preprocess` function further remaps the mask values for training.

Check Mask Values Function

The 'check_unique_mask_values' function checks the unique values in the processed masks. This is important for ensuring the masks have the correct number of class values before training.

Convolutional and Up sampling Blocks

- **Convolutional Block ('conv_block'):** This block is used to perform two consecutive convolutions with optional dropout and max pooling. This is the encoder part of U-Net, which captures spatial features at different scales.
- **Up sampling Block ('upsampling_block'):** This block uses `Conv2DTranspose` to upsample the feature maps, concatenates the corresponding downsampled feature maps, and applies further convolutions. This forms the decoder part of U-Net, which helps reconstruct the image at the original resolution.

U-Net Model Architecture

```python
def unet_model(input_size=(96, 128, 3), n_filters=32, n_classes=5):
    """
    Unet model
    """
    inputs = Input(input_size)

    cblock1 = conv_block(inputs, n_filters)

    cblock2 = conv_block(cblock1[0], 2*n_filters)
    cblock3 = conv_block(cblock2[0], 4*n_filters)
    cblock4 = conv_block(cblock3[0], 8*n_filters, dropout_prob=0.3)

    cblock5 = conv_block(cblock4[0], 16*n_filters, dropout_prob=0.3, max_pooling=False)

    ublock6 = upsampling_block(cblock5[0], cblock4[1],  n_filters*8)

    ublock7 = upsampling_block(ublock6, cblock3[1],  n_filters*4)
    ublock8 = upsampling_block(ublock7, cblock2[1],  n_filters*2)
    ublock9 = upsampling_block(ublock8, cblock1[1],  n_filters)


    conv9 = Conv2D(n_filters, 3, activation='relu', padding='same', kernel_initializer='he_normal')(ublock9)

    conv10 = Conv2D(n_classes, 1, padding='same')(conv9)

    outputs = tf.keras.layers.Softmax()(conv10)  # Add softmax activation
    model = tf.keras.Model(inputs=inputs, outputs=outputs)
    return model
```

- The 'unet_model' function defines the full U-Net architecture. It consists of:
- A contracting path (encoder) built using 'conv_block' layers with increasing filter sizes.
- A bottleneck layer (`cblock5`) that captures the most abstract features.
- An expansive path (decoder) built using `upsampling_block` layers that progressively reconstruct the feature map size.
- The final layer outputs a segmentation map with `n_classes` (5 in this case) using `Softmax` activation to classify each pixel into one of the classes.

Custom Mean IoU Metric

- The custom `MeanIoUWithArgmax` metric calculates the Mean Intersection over Union (IoU), which is commonly used for evaluating segmentation tasks. It includes an `update_state` method that converts the predicted softmax values into integer labels using `tf.argmax`.

Training the U-Net Model

- The U-Net model is compiled using the Adam optimizer and Sparse Categorical Crossentropy loss function, which is suitable for multi-class segmentation tasks where the labels are integers.
- Training: The model is trained on the dataset using `fit()` for 50 epochs.

Prediction and Visualization

show_predictions(): This function displays predictions made by the U-Net model. It uses `unet.predict()` to get the predicted mask for an input image and then visualizes the input image, true mask, and predicted mask using `matplotlib`.

Overall overview

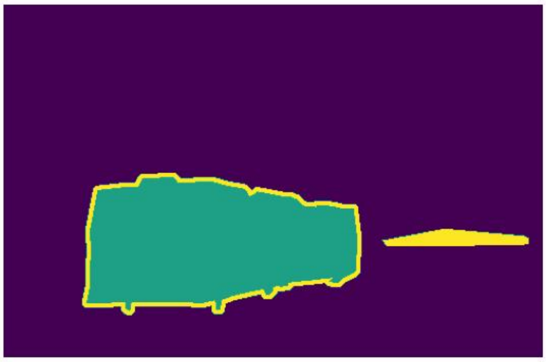
1. Data Loading and Preprocessing: Images and masks are loaded and paired, resized, and remapped to ensure correct labels.
2. Model Construction: U-Net architecture is constructed using convolutional and up sampling blocks.
3. Training: The U-Net is trained for multi-class segmentation, with IoU used as an evaluation metric.
4. Prediction and Visualization: Trained model is used to predict and display segmentation results.

Results

Given dataset:

Input Image

True Mask

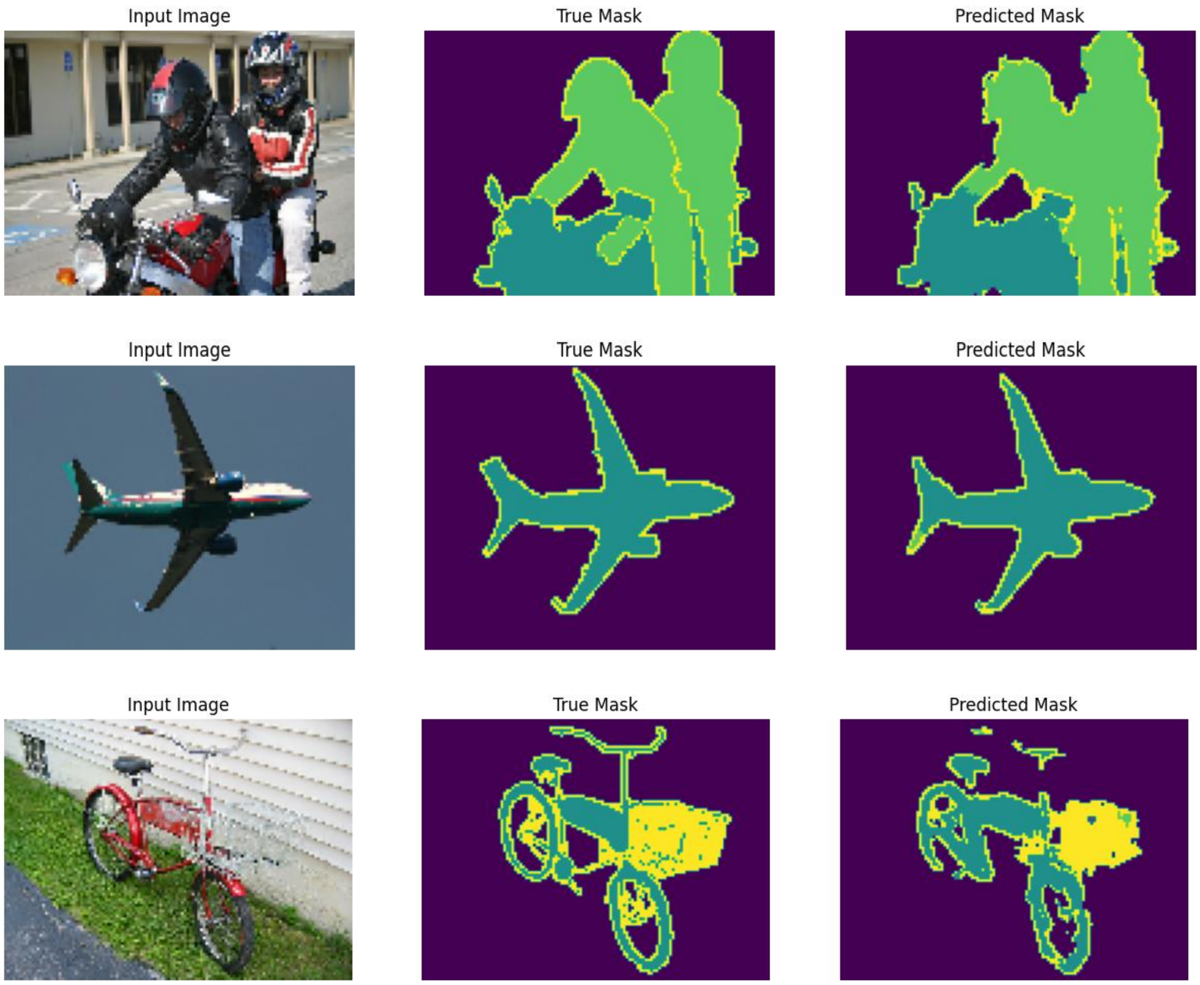


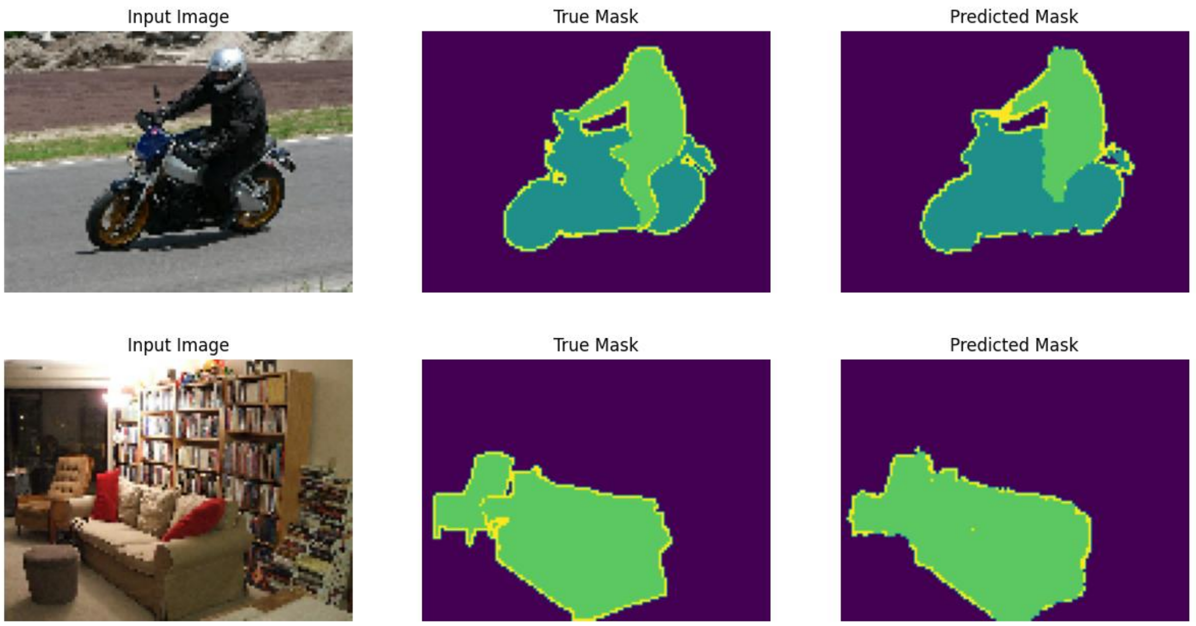
Training dataset output

```
Epoch 1/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 130s 2s/step - loss: 0.2338 - mean_io_u_with_argmax: 0.6990 - val_loss: 0.3574 - val_mean_io_u_with_argmax: 0.6459
Epoch 2/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 127s 2s/step - loss: 0.2284 - mean_io_u_with_argmax: 0.7039 - val_loss: 0.3258 - val_mean_io_u_with_argmax: 0.6250
Epoch 3/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 128s 2s/step - loss: 0.2429 - mean_io_u_with_argmax: 0.6866 - val_loss: 0.3988 - val_mean_io_u_with_argmax: 0.5961
Epoch 4/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 127s 2s/step - loss: 0.2811 - mean_io_u_with_argmax: 0.6514 - val_loss: 0.3893 - val_mean_io_u_with_argmax: 0.6112
Epoch 5/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 126s 2s/step - loss: 0.2438 - mean_io_u_with_argmax: 0.6863 - val_loss: 0.3635 - val_mean_io_u_with_argmax: 0.6439
Epoch 6/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 126s 2s/step - loss: 0.2070 - mean_io_u_with_argmax: 0.7179 - val_loss: 0.3049 - val_mean_io_u_with_argmax: 0.6651
Epoch 7/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 127s 2s/step - loss: 0.1901 - mean_io_u_with_argmax: 0.7385 - val_loss: 0.3392 - val_mean_io_u_with_argmax: 0.6571
Epoch 8/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 126s 2s/step - loss: 0.1952 - mean_io_u_with_argmax: 0.7359 - val_loss: 0.3183 - val_mean_io_u_with_argmax: 0.6616
Epoch 9/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 126s 2s/step - loss: 0.1866 - mean_io_u_with_argmax: 0.7420 - val_loss: 0.3430 - val_mean_io_u_with_argmax: 0.6488
Epoch 10/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 126s 2s/step - loss: 0.1843 - mean_io_u_with_argmax: 0.7458 - val_loss: 0.3538 - val_mean_io_u_with_argmax: 0.6680
Epoch 11/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 128s 2s/step - loss: 0.1750 - mean_io_u_with_argmax: 0.7531 - val_loss: 0.3627 - val_mean_io_u_with_argmax: 0.6528
Epoch 12/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 126s 2s/step - loss: 0.1784 - mean_io_u_with_argmax: 0.7478 - val_loss: 0.3313 - val_mean_io_u_with_argmax: 0.6533
Epoch 13/50
...
Epoch 49/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 131s 2s/step - loss: 0.1313 - mean_io_u_with_argmax: 0.7998 - val_loss: 0.3270 - val_mean_io_u_with_argmax: 0.7083
Epoch 50/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 127s 2s/step - loss: 0.1245 - mean_io_u_with_argmax: 0.8105 - val_loss: 0.3669 - val_mean_io_u_with_argmax: 0.6939
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

Validation dataset results




Input Image
True Mask
Predicted Mask
Input Image
True Mask
Predicted Mask