Vision HW2 Documentation

Fatemeh Mirzaei Kalani 400243075

CNN implementation to classify shoes

Our goal is to implement a CNN to classify images of shoes related to three classes (Adidas, Converse, and Nike).

For simplicity and flexibility, we define all the hyperparameters in a class:

```
@dataclass
class Config:
    num_classes = 3
    batch_size = 20
    learning_rate = 0.001
    num_epochs = 15
    num_workers = 2
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    data_dir = "/content/drive/MyDrive/Vision/shoes"
    input_size = (128, 128)
    weight_decay = 1e-5 # L2 regularization

config = Config()
```

For training data, data augmentation is done to prevent overfitting and improve generalization. For testing data, it only resizes the images and normalizes the pixel values to a specific range. In both cases, the images are converted to PyTorch tensors for use in the model.

```
def get_transform(train=True):
    if train:
        return transforms.Compose([
            transforms.RandomResizedCrop((128, 128)),  # Random crop and resize
            transforms.RandomHorizontalFlip(),         # Random horizontal flip
            transforms.RandomRotation(15),             # Random rotation within ±15 degrees
            transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),  # Adjust brightness, contrast
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
        ])
    else:
        return transforms.Compose([
            transforms.Resize((128, 128)),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
        ])
```

Based on batch size the data are loaded into dataloaders. Also, num_workers determines how many subprocesses are used to load the data. Pin_memory can

speed up data transfer by copying data to pinned (page-locked) memory (if you are using GPU).

```
train_dataloader = DataLoader(train_dataset, batch_size = config.batch_size,
                              shuffle = True, num_workers = config.num_workers,
                              pin_memory = True)
test_dataloader = DataLoader(test_dataset, batch_size = config.batch_size,
                             shuffle = True, num_workers = config.num_workers,
                             pin_memory = True)
```

Showing part of the data:

The main model consists of three conv2d. Batch normalization and dropout don't have an impact on the size of the input. Batch normalization layers stabilize and speed up training by normalizing the activations within a batch. Dropout randomly assigns o to some input unit to prevent overfitting. The last layer which is a fully connected layer, converts input to the number of our classes (three classes).

Dropout:

Training Phase :

$$\mathbf{y} = f((\mathbf{W} \circ \mathbf{M})\mathbf{x}), \quad M_{i,j} \sim Bernoulli(p)$$

Testing Phase :

$$\mathbf{y} = (\mathbf{W}\mathbf{x}) \circ \hat{\mathbf{m}}(\mathbf{Z})$$

Batch normalization:

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

```python
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        # First Block
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)   # Output: (N, 16, 128, 128)
        self.bn1 = nn.BatchNorm2d(16)                             # Output: (N, 16, 128, 128)
        self.pool1 = nn.MaxPool2d(2)                              # Output: (N, 16, 64, 64)

        # Second Block
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)  # Output: (N, 32, 64, 64)
        self.bn2 = nn.BatchNorm2d(32)                             # Output: (N, 32, 64, 64)
        self.pool2 = nn.MaxPool2d(2)                              # Output: (N, 32, 32, 32)
        self.dropout1 = nn.Dropout2d(0.1)

        # Third Block
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, padding=1)  # Output: (N, 64, 32, 32)
        self.bn3 = nn.BatchNorm2d(64)                             # Output: (N, 64, 32, 32)
        self.pool3 = nn.MaxPool2d(2)                               # Output: (N, 64, 16, 16)

        self.dropout2 = nn.Dropout(0.5)

        # Fully Connected Layer
        self.fc1 = nn.Linear(64 * 16 * 16, config.num_classes)
```

This function trains one epoch. model.train() is a training mode that enables certain operations like dropout, which are important for training but not for evaluation. In forward pass we pass the images into the model to get predictions and get the loss of that based on the real labels.

In backward propagation, we reset the gradients of parameters to zero. Then the gradients of loss with respect to weights and biases get calculated. With optimizer.step() the parameters get updated based on the optimizer algorithm. In the end, the loss of training is returned.

```python
def train_model(model, train_dataloader, criterion, optimizer):
    model.train()
    total_loss = 0.0
    for images, labels in train_dataloader:
        images, labels = images.to(config.device), labels.to(config.device)

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

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    return total_loss / len(train_dataloader)
```

model.eval() ensures the model is configured for inference (making predictions) rather than learning. **torch.no_gard()** temporarily disables gradient calculations, which are essential for training but unnecessary during evaluation. The main loop passes the images through the model to obtain predictions and save metrics.

Accuracy gives an overall sense of how well the model is performing across all classes.

Precision focuses on the quality of positive predictions.

Recall focuses on the completeness of positive predictions.

F1-score provides a balanced measure that considers both the quality and completeness of positive predictions.

$$\textbf{Precision} = \frac{\text{True Positive}}{\text{Actual Results}} \quad \text{or} \quad \frac{\text{True Positive}}{\text{True Positive + False Positive}}$$

$$\textbf{Recall} = \frac{\text{True Positive}}{\text{Predicted Results}} \quad \text{or} \quad \frac{\text{True Positive}}{\text{True Positive + False Negative}}$$

$$\textbf{Accuracy} = \frac{\text{True Positive + True Negative}}{\text{Total}}$$

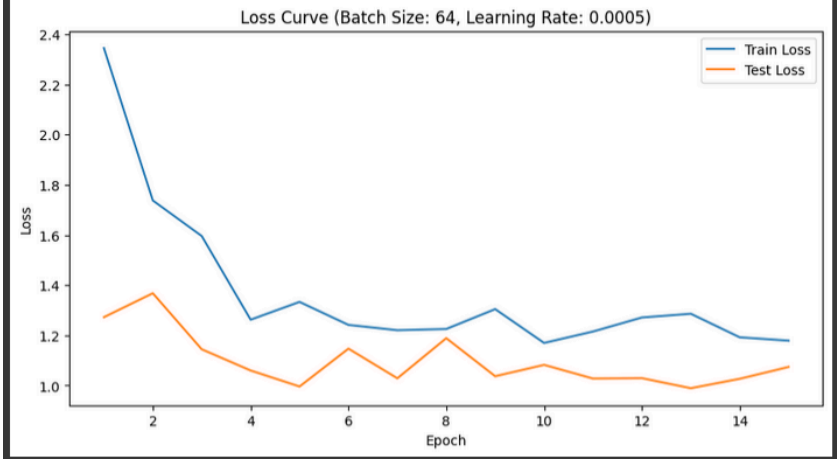
```python
def evaluate_model(model, test_dataloader, criterion):
    model.eval()
    total_loss = 0.0
    correct = 0
    total = 0
    all_labels = []
    all_predictions = []

    with torch.no_grad():
        for images, labels in test_dataloader:
            images, labels = images.to(config.device), labels.to(config.device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            total_loss += loss.item()

            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            all_labels.extend(labels.cpu().numpy())
            all_predictions.extend(predicted.cpu().numpy())

    accuracy = 100 * correct / total

    precision = precision_score(all_labels, all_predictions, average='weighted', zero_division=0)
    recall = recall_score(all_labels, all_predictions, average='weighted', zero_division=0)
    f1 = f1_score(all_labels, all_predictions, average='weighted', zero_division=0)

    return total_loss / len(test_dataloader), accuracy, precision, recall, f1
```

In this code block, we change batch_size and learning_rate to show the change in the performance of the model. In the loop, cross-entropy loss is our loss function and Adam is the optimizer algorithm. Also, weight_decay is used to regularize parameters. The model is trained for each batch size and learning rate and its performance is evaluated.

```python
for batch_size in batch_sizes:
    for learning_rate in learning_rates:
        train_loader = DataLoader(dataset=train_dataset,
                                  batch_size=batch_size,
                                  shuffle=True,
                                  num_workers=config.num_workers,
                                  pin_memory=True)
        test_loader = DataLoader(dataset=test_dataset,
                                 batch_size=batch_size,
                                 shuffle=False,
                                 num_workers=config.num_workers,
                                 pin_memory=True)

        model = CNN().to(config.device)
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.Adam(model.parameters(), lr=learning_rate, weight_decay=Config.weight_decay)

        print(f"Training with batch_size={batch_size}, learning_rate={learning_rate}")

        train_losses = []
        test_losses = []

        for epoch in range(config.num_epochs):
            train_loss = train_model(model, train_loader, criterion, optimizer)
            test_loss, accuracy, precision, recall, f1 = evaluate_model(model, test_loader, criterion)

            train_losses.append(train_loss)
            test_losses.append(test_loss)

            print(f'Epoch [{epoch+1}/{config.num_epochs}], Train Loss: {train_loss:.4f}, Test Loss: {test_loss:.4f},

        plot_losses(train_losses, test_losses, batch_size, learning_rate)

        results.append((batch_size, learning_rate, accuracy, precision, recall, f1))
```

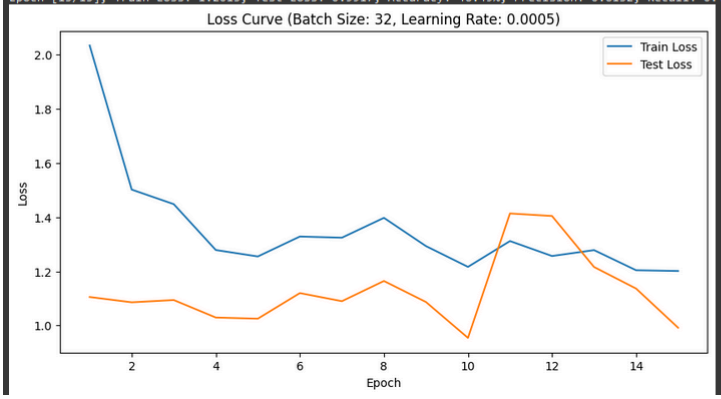
The result for batch_size=64 and learning_rate=0.0005:

```
Training with batch_size=64, learning_rate=0.0005
Epoch [1/15], Train Loss: 2.3452, Test Loss: 1.2725, Accuracy: 33.33%, Precision: 0.1111, Recall: 0.3333, F1 Score: 0.1667
Epoch [2/15], Train Loss: 1.7378, Test Loss: 1.3680, Accuracy: 42.11%, Precision: 0.3502, Recall: 0.4211, F1 Score: 0.3107
Epoch [3/15], Train Loss: 1.5961, Test Loss: 1.1446, Accuracy: 43.86%, Precision: 0.2919, Recall: 0.4386, F1 Score: 0.3496
Epoch [4/15], Train Loss: 1.2625, Test Loss: 1.0600, Accuracy: 45.61%, Precision: 0.5804, Recall: 0.4561, F1 Score: 0.4408
Epoch [5/15], Train Loss: 1.3335, Test Loss: 0.9961, Accuracy: 45.61%, Precision: 0.5015, Recall: 0.4561, F1 Score: 0.4281
Epoch [6/15], Train Loss: 1.2417, Test Loss: 1.1471, Accuracy: 42.98%, Precision: 0.4382, Recall: 0.4298, F1 Score: 0.4090
Epoch [7/15], Train Loss: 1.2207, Test Loss: 1.0289, Accuracy: 49.12%, Precision: 0.4925, Recall: 0.4912, F1 Score: 0.4911
Epoch [8/15], Train Loss: 1.2255, Test Loss: 1.1888, Accuracy: 42.98%, Precision: 0.4937, Recall: 0.4298, F1 Score: 0.3844
Epoch [9/15], Train Loss: 1.3047, Test Loss: 1.0371, Accuracy: 44.74%, Precision: 0.4861, Recall: 0.4474, F1 Score: 0.4236
Epoch [10/15], Train Loss: 1.1698, Test Loss: 1.0824, Accuracy: 49.12%, Precision: 0.5095, Recall: 0.4912, F1 Score: 0.4622
Epoch [11/15], Train Loss: 1.2154, Test Loss: 1.0278, Accuracy: 54.39%, Precision: 0.6416, Recall: 0.5439, F1 Score: 0.4950
Epoch [12/15], Train Loss: 1.2713, Test Loss: 1.0294, Accuracy: 46.49%, Precision: 0.6401, Recall: 0.4649, F1 Score: 0.3828
Epoch [13/15], Train Loss: 1.2862, Test Loss: 0.9893, Accuracy: 53.51%, Precision: 0.5563, Recall: 0.5351, F1 Score: 0.5314
Epoch [14/15], Train Loss: 1.1923, Test Loss: 1.0269, Accuracy: 48.25%, Precision: 0.5000, Recall: 0.4825, F1 Score: 0.4633
Epoch [15/15], Train Loss: 1.1789, Test Loss: 1.0742, Accuracy: 49.12%, Precision: 0.5972, Recall: 0.4912, F1 Score: 0.4538
```



Test loss is less than train loss so we prevented overfitting. The accuracy is around 50%.
As you can see with batch size 32 and a learning rate 0.0005 the metrics are different

```
Training with batch_size=32, learning_rate=0.0005
Epoch [1/15], Train Loss: 2.0342, Test Loss: 1.1052, Accuracy: 40.35%, Precision: 0.2935, Recall: 0.4035, F1 Score: 0.3017
Epoch [2/15], Train Loss: 1.5020, Test Loss: 1.0856, Accuracy: 43.86%, Precision: 0.4341, Recall: 0.4386, F1 Score: 0.4335
Epoch [3/15], Train Loss: 1.4482, Test Loss: 1.0940, Accuracy: 49.12%, Precision: 0.5013, Recall: 0.4912, F1 Score: 0.4624
Epoch [4/15], Train Loss: 1.2788, Test Loss: 1.0294, Accuracy: 44.74%, Precision: 0.4738, Recall: 0.4474, F1 Score: 0.4468
Epoch [5/15], Train Loss: 1.2550, Test Loss: 1.0251, Accuracy: 53.51%, Precision: 0.5453, Recall: 0.5351, F1 Score: 0.5266
Epoch [6/15], Train Loss: 1.3286, Test Loss: 1.1197, Accuracy: 44.74%, Precision: 0.4176, Recall: 0.4474, F1 Score: 0.3862
Epoch [7/15], Train Loss: 1.3243, Test Loss: 1.0897, Accuracy: 43.86%, Precision: 0.4298, Recall: 0.4386, F1 Score: 0.3847
Epoch [8/15], Train Loss: 1.3977, Test Loss: 1.1644, Accuracy: 43.86%, Precision: 0.2949, Recall: 0.4386, F1 Score: 0.3510
Epoch [9/15], Train Loss: 1.2933, Test Loss: 1.0868, Accuracy: 47.37%, Precision: 0.5530, Recall: 0.4737, F1 Score: 0.4263
Epoch [10/15], Train Loss: 1.2165, Test Loss: 0.9542, Accuracy: 54.39%, Precision: 0.5523, Recall: 0.5439, F1 Score: 0.5279
Epoch [11/15], Train Loss: 1.3119, Test Loss: 1.4138, Accuracy: 40.35%, Precision: 0.5310, Recall: 0.4035, F1 Score: 0.3088
Epoch [12/15], Train Loss: 1.2566, Test Loss: 1.4045, Accuracy: 47.37%, Precision: 0.5852, Recall: 0.4737, F1 Score: 0.4006
Epoch [13/15], Train Loss: 1.2784, Test Loss: 1.2161, Accuracy: 50.88%, Precision: 0.5369, Recall: 0.5088, F1 Score: 0.4414
Epoch [14/15], Train Loss: 1.2041, Test Loss: 1.1364, Accuracy: 43.86%, Precision: 0.5092, Recall: 0.4386, F1 Score: 0.4100
Epoch [15/15], Train Loss: 1.2013, Test Loss: 0.9917, Accuracy: 46.49%, Precision: 0.6132, Recall: 0.4649, F1 Score: 0.4441
```



Although 50% accuracy is not so satisfying for us. So the next plan is using Transfer learning with a pretrained densenet121 model.

Fine-tuning is a transfer learning technique. It is the process of taking a pre-trained model and adapting it to a specific task by continuing its training on a new dataset. Instead of training the model from scratch, which requires a large amount of data and computation, fine-tuning leverages the knowledge the model has already learned from its original training (e.g., on a large dataset like COCO). During fine-tuning, the model's parameters are updated to better fit the new task, often with a smaller learning rate to avoid overwriting the pre-trained weights.

```python
def get_transfer_learning_model():
    model = models.densenet121(weights='DEFAULT' )

    for param in model.parameters():
        param.requires_grad = False

    # unfreeze all BatchNormalization layers to make them use the new dataset's statistics instead of the original statistics
    for module in model.modules():
        if isinstance(module, nn.BatchNorm2d):
            for param in module.parameters():
                param.requires_grad = True

    # replace the final fully connected layer for our specific task
    classifier = nn.Sequential(nn.Linear(1024, 512),
                    nn.ReLU(),
                    nn.Linear(512, 256),
                    nn.ReLU(),
                    nn.Dropout(0.5),
                    nn.Linear(256, config.num_classes)
                    )

    model.classifier = classifier

    return model
```

To preserve knowledge gained from the dense121 model we freeze layers to prevent parameters from getting updated on the new dataset. However, we want the parameters of batch normalization to be updated during training.
We replace the original classification head (the last layer(s) of the pre-trained model responsible for making predictions) with this new classifier. This is necessary to adapt the model to the specific number of classes in the new dataset.

```
Epoch [1/15], Train Loss: 1.0748, Test Loss: 0.9663, Accuracy: 57.02%, Precision: 0.5820, Recall: 0.5702, F1 Score: 0.5582
Epoch [2/15], Train Loss: 0.8898, Test Loss: 0.7140, Accuracy: 70.18%, Precision: 0.7284, Recall: 0.7018, F1 Score: 0.7007
Epoch [3/15], Train Loss: 0.7343, Test Loss: 0.6182, Accuracy: 78.07%, Precision: 0.7984, Recall: 0.7807, F1 Score: 0.7843
Epoch [4/15], Train Loss: 0.7029, Test Loss: 0.5360, Accuracy: 76.32%, Precision: 0.7640, Recall: 0.7632, F1 Score: 0.7575
Epoch [5/15], Train Loss: 0.5892, Test Loss: 0.5049, Accuracy: 78.07%, Precision: 0.7889, Recall: 0.7807, F1 Score: 0.7808
Epoch [6/15], Train Loss: 0.5231, Test Loss: 0.4826, Accuracy: 84.21%, Precision: 0.8517, Recall: 0.8421, F1 Score: 0.8426
Epoch [7/15], Train Loss: 0.4626, Test Loss: 0.4735, Accuracy: 80.70%, Precision: 0.8087, Recall: 0.8070, F1 Score: 0.8049
Epoch [8/15], Train Loss: 0.4711, Test Loss: 0.4676, Accuracy: 87.72%, Precision: 0.8814, Recall: 0.8772, F1 Score: 0.8775
Epoch [9/15], Train Loss: 0.4738, Test Loss: 0.4919, Accuracy: 81.58%, Precision: 0.8164, Recall: 0.8158, F1 Score: 0.8153
Epoch [10/15], Train Loss: 0.4594, Test Loss: 0.4027, Accuracy: 87.72%, Precision: 0.8774, Recall: 0.8772, F1 Score: 0.8769
Epoch [11/15], Train Loss: 0.3817, Test Loss: 0.3621, Accuracy: 87.72%, Precision: 0.8779, Recall: 0.8772, F1 Score: 0.8768
Epoch [12/15], Train Loss: 0.4246, Test Loss: 0.3870, Accuracy: 83.33%, Precision: 0.8326, Recall: 0.8333, F1 Score: 0.8326
Epoch [13/15], Train Loss: 0.3699, Test Loss: 0.3993, Accuracy: 87.72%, Precision: 0.8798, Recall: 0.8772, F1 Score: 0.8781
Epoch [14/15], Train Loss: 0.4073, Test Loss: 0.3496, Accuracy: 88.60%, Precision: 0.8881, Recall: 0.8860, F1 Score: 0.8866
Epoch [15/15], Train Loss: 0.3426, Test Loss: 0.5242, Accuracy: 80.70%, Precision: 0.8100, Recall: 0.8070, F1 Score: 0.8036
Done!
```

Voila!!! We got 86% accuracy.

Denoising images with UNET

Our goal is to put salt and paper noise on images of a dataset and then denoise it using UNET model. Then we compare denoised images with clean images.

The data looks like this:

```
data = pd.read_csv(config.data_path)
data.head()
```

	emotion	pixels	Usage
0	0	70 80 82 72 58 58 60 63 54 58 60 48 89 115 121...	Training
1	0	151 150 147 155 148 133 111 140 170 174 182 15...	Training
2	2	231 212 156 164 174 138 161 173 182 200 106 38...	Training
3	4	24 32 36 30 32 23 19 20 30 41 21 22 32 34 21 1...	Training
4	6	4 0 0 0 0 0 0 0 0 0 0 0 3 15 23 28 48 50 58 84...	Training

The pixels should be converted to numpy array to be shown as images.

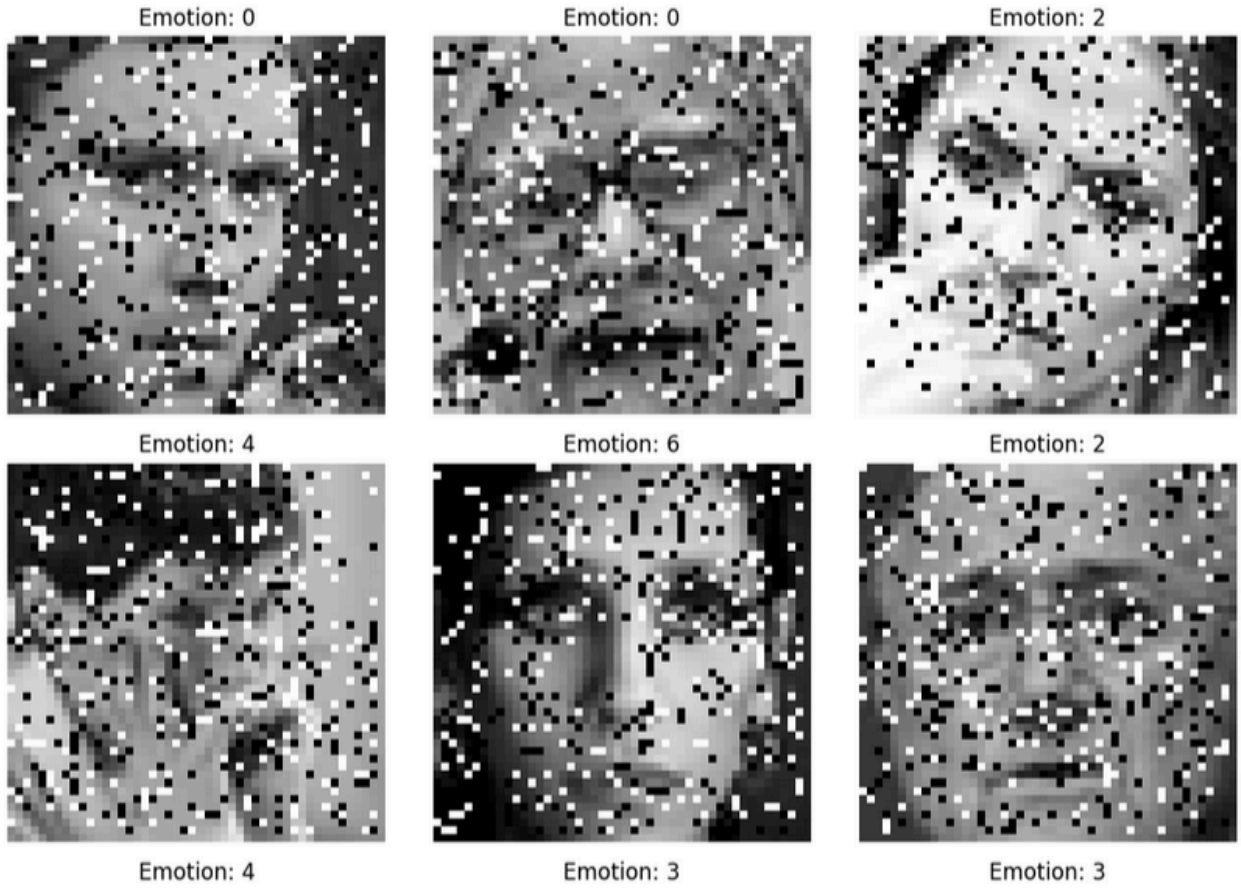
```
def process_pixels(pixel_string):
    return np.array(pixel_string.split(), dtype='float32').reshape(48, 48) # Convert the 'pixels' column into NumPy arrays

data['processed_pixels'] = data['pixels'].apply(process_pixels)
```

Then we construct the image:

Salt means 255 as the value of a pixel and paper is 0. Randomly, the value of some pixels gets changed to salt and paper.

```
def add_salt_and_pepper_noise(image, noise_factor=0.1):
    row, col = image.shape
    noisy_image = np.copy(image)

    # Salt noise (white pixels)
    num_salt = int(noise_factor * row * col)
    salt_coords = [np.random.randint(0, i-1, num_salt) for i in image.shape]
    noisy_image[salt_coords[0], salt_coords[1]] = 255

    # Pepper noise (black pixels)
    num_pepper = int(noise_factor * row * col)
    pepper_coords = [np.random.randint(0, i-1, num_pepper) for i in image.shape]
    noisy_image[pepper_coords[0], pepper_coords[1]] = 0

    return noisy_image
```

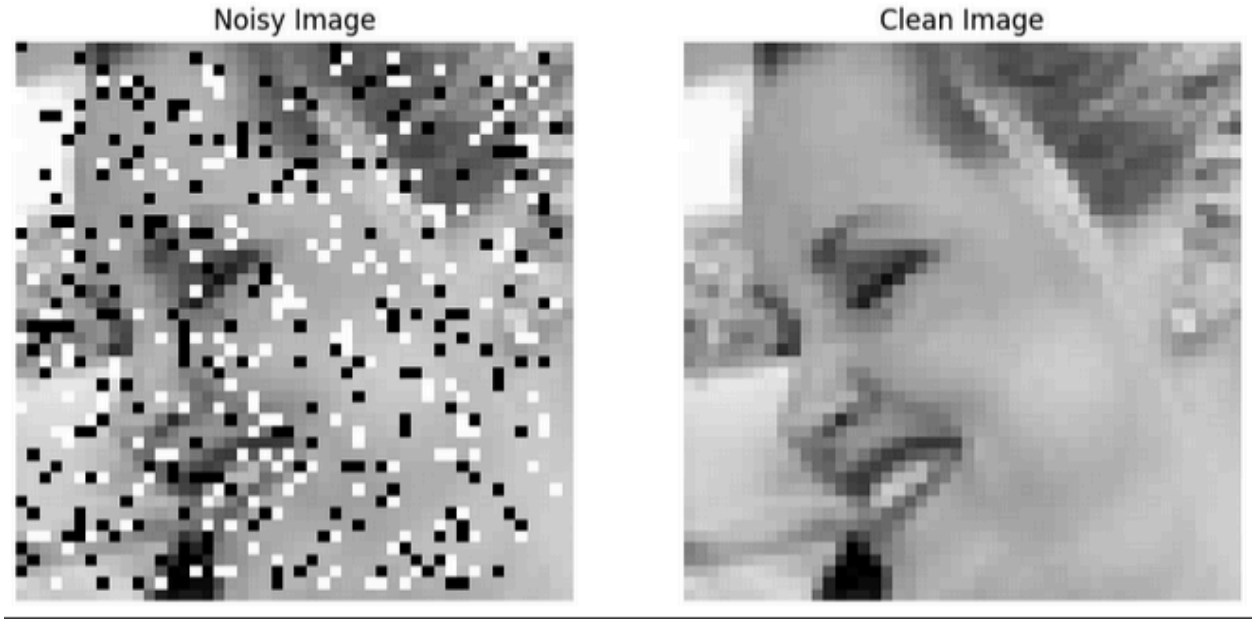
```
data['noisy_pixels'] = data['processed_pixels'].apply(lambda x: add_salt_and_pepper_noise(x, noise_factor=0.1))
```



For using dataloader we need a dataset object so we create a new customclass for our dataset. Also, each numpy array gets converted to a tensor and gets normalized between 0 and 1. Then train, val, and test dataloader get created. We include both noisy images and clean images in the dataset. The clean one acts like our label:

```
class CustomDataset(Dataset):
    def __init__(self, noisy_images, clean_images):
        self.noisy_images = noisy_images
        self.clean_images = clean_images

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

    def __getitem__(self, idx):
        noisy_image = torch.tensor(self.noisy_images[idx], dtype=torch.float32).unsqueeze(0) / 255.0  # [0...1]
        clean_image = torch.tensor(self.clean_images[idx], dtype=torch.float32).unsqueeze(0) / 255.0  # [0...1]
        return noisy_image, clean_image
```

```
def create_dataloader(data=train_data):
  X_train_noisy = np.array(data['noisy_pixels'].tolist())  # Assuming noisy images are added
  y_train_clean = np.array(data['processed_pixels'].tolist())  # Clean images

  dataset = CustomDataset(X_train_noisy, y_train_clean)
  dataloader = DataLoader(dataset, batch_size=config.batch_size, shuffle=True, num_workers=config.num_workers, pin_memory=True)
  return dataloader

train_dataloader, val_dataloader, test_dataloader = [create_dataloader(x) for x in [train_data, val_data, test_data]]
```



U-Net is a type of neural network that uses the Convolution structure in its architecture. the U-Net network uses an Encoder-Decoder structure, where in the Encoder part, the dimensions of the image are halved at each level of the network, and the number of filters is doubled. In the Decoder section, unlike the Encoder, the movement is from bottom to top, the dimensions are doubled, and the number of filters is halved.

the Encoder section performs the feature extraction process on the image and learns an abstract representation of the image. The Decoder receives the abstract representation created by the Encoder and creates a semantic segmentation mask.

Unet: Each block contains two convolutional layers with the LeakyReLU activation function and a Dropout layer. Also, stride is used instead of maxpooling. We use 5 Conv_block blocks in the downward stage to reach the number of input image channels to 1024 and reduce its dimensions. Then, in the upward stage, we again convert the number of channels to 1 and return the image dimensions to the original dimensions (48*48).

In upwarding the model should have a look at the feature maps of the downward stage in addition to the output of the previous stage.

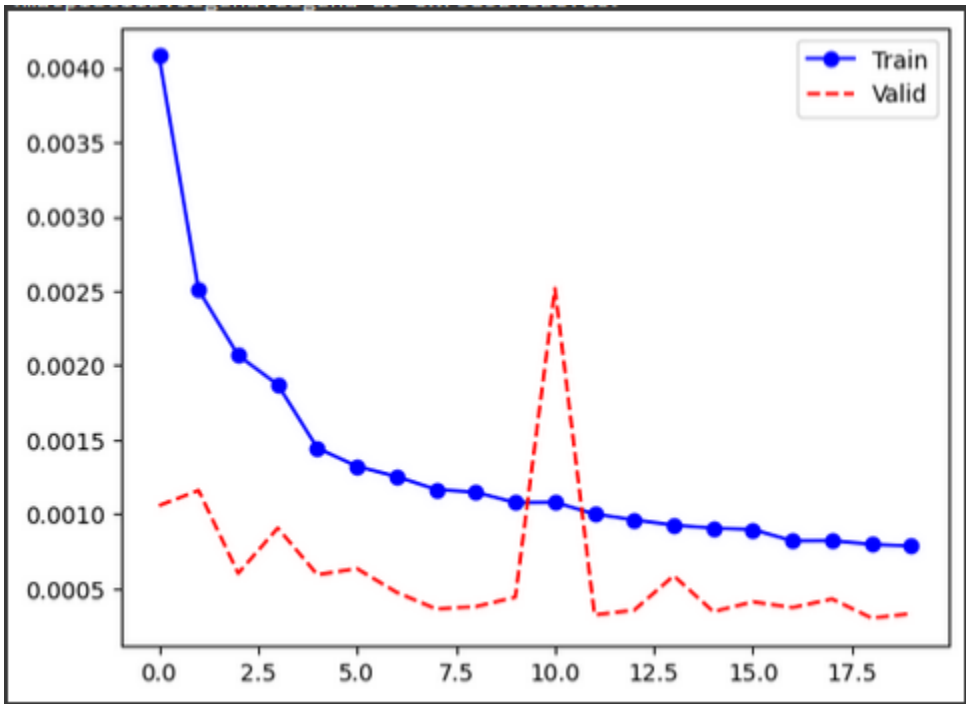
```
class Unet(nn.Module):
    # initializers
    def __init__(self, d=64, out_channels=1, dropout=0.2):
        super().__init__()

        # Unet encoder
        self.conv_blocks = nn.ModuleList([
            Conv_block(1, d, dropout=dropout),
            Conv_block(d, 2 * d, stride=2, dropout=dropout),
            Conv_block(2 * d, 4 * d, stride=2, dropout=dropout),
            Conv_block(4 * d, 8 * d, stride=2, dropout=dropout),
            Conv_block(8 * d, 16 * d, stride=2, dropout=dropout)
        ])

        # Unet decoder
        self.deConv_blocks = nn.ModuleList([
            DeConv_block(16 * d, 8 * d, dropout=dropout),
            DeConv_block(8 * d, 4 * d, dropout=dropout),
            DeConv_block(4 * d, 2 * d, dropout=dropout),
            DeConv_block(2 * d, d, dropout=dropout),
        ])

        self.output_conv = nn.Conv2d(d, out_channels, 3, padding=1)
        self.float()

    # forward method
    def forward(self, x):
        x1 = self.conv_blocks[0](x)
        x2 = self.conv_blocks[1](x1)
        x3 = self.conv_blocks[2](x2)
        x4 = self.conv_blocks[3](x3)
        x5 = self.conv_blocks[4](x4)

        x = self.deConv_blocks[0](x5, x4)
        x = self.deConv_blocks[1](x, x3)
        x = self.deConv_blocks[2](x, x2)
        x = self.deConv_blocks[3](x, x1)

        x = self.output_conv(x)

        return x
```

Mean squared error is used as a loss function. The model is trained in 20 epochs. The model gets evaluated based on the validation dataloader. Also, the best model with the least validation loss gets saved.

```
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=config.learning_rate)

valid_loss_min = np.Inf

model.to(config.device)
history = {'train': [], 'valid': []}
for i in range(config.num_epochs):
    print(f'Epoch {i+1} ...')

    model.train()
    sum_train_mse = 0
    training_steps = 0
    for (x, y) in tqdm(train_dataloader):
        optimizer.zero_grad()

        x = x.to(config.device)
        y = y.to(config.device)

        output = model(x)
        loss = criterion(output, y)

        sum_train_mse += loss.cpu().item()
        training_steps += 1

        loss.backward()
        optimizer.step()

    model.eval()
    sum_valid_mse = 0
    valid_steps = 0
    for (x, y) in tqdm(val_dataloader):

        x = x.to(config.device)
        y = y.to(config.device)

        with torch.no_grad():
            output = model(x)
        loss = criterion(output, y)

        valid_loss = loss.cpu().item()
        sum_valid_mse += loss.cpu().item()
        valid_steps += 1

    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}). The new model saved'.format(valid_loss_min, valid_loss))
        torch.save(model.state_dict(), 'UNET-1-0.07.pt')
        valid_loss_min = valid_loss

    history['train'].append(sum_train_mse / training_steps)
    history['valid'].append(sum_valid_mse / valid_steps)
```

```
Epoch 20, Average Train MSE: 0.000784984205702718, Average Validation MSE: 0.00033093698269201326
```

In most epochs, loss of validation is less than loss of Training so we avoided overfitting.



To understand the quality of denoised images we can use PSNR and SSIM:

$$\mathrm{PSNR} = 10 \times \lg\left(\frac{255^2}{\mathrm{MSE}}\right)$$

Higher PSNR indicates less noise and higher quality. **30 ≤ PSNR < 40 dB** means good quality, minor differences may exist but not easily noticeable.

SSIM measures the similarity between two images based on luminance, contrast, and structural information. SSIM values range from **-1 to 1**, where:

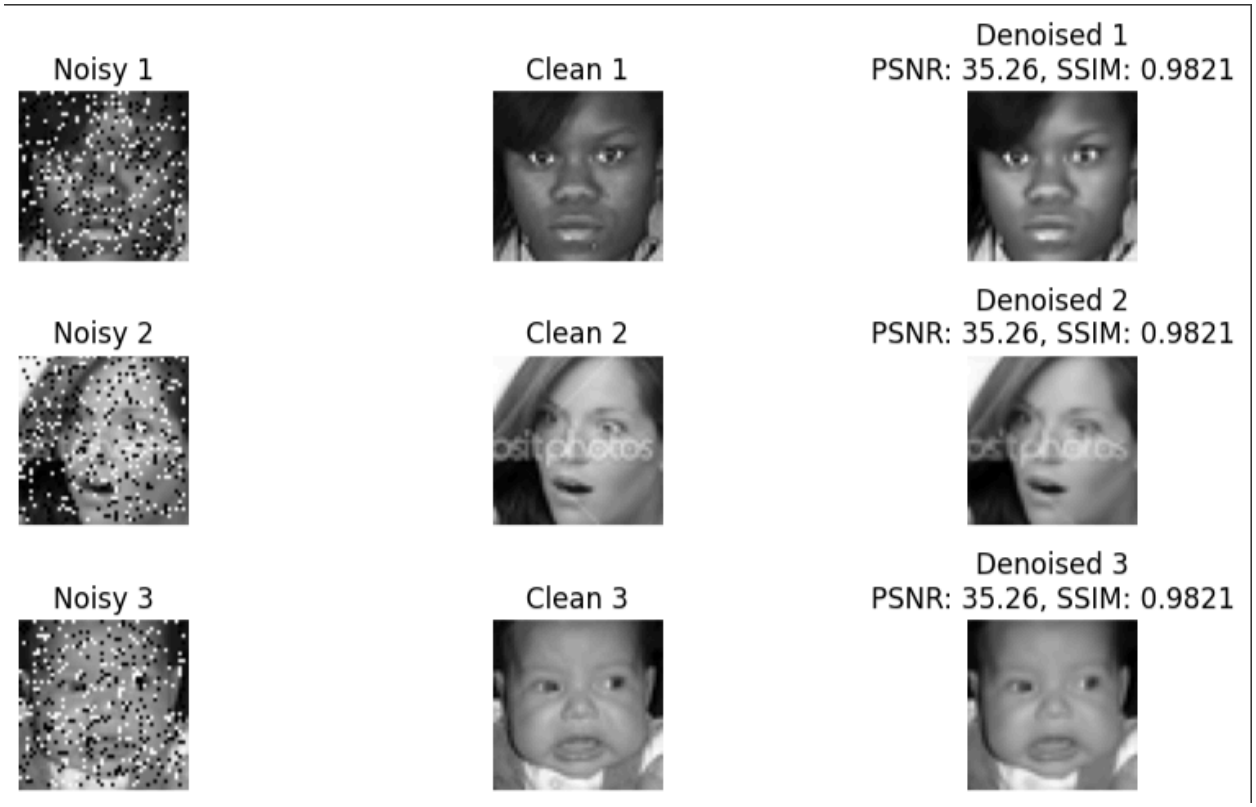
1: Perfect similarity (the two images are identical in content and quality).

0: No structural similarity.

SSIM emphasizes structural and perceptual similarity, often better aligning with human vision.

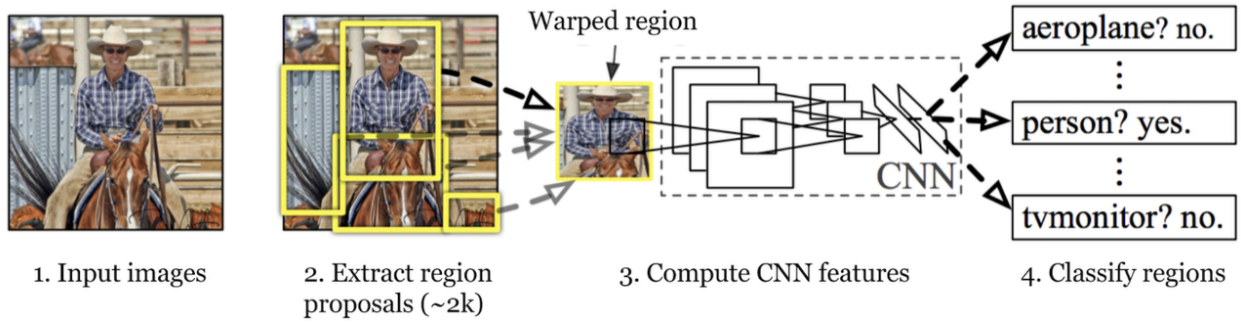
```python
from skimage.metrics import peak_signal_noise_ratio, structural_similarity

def calculate_metrics(denoised_imgs, clean_imgs):
    psnr_values = []
    ssim_values = []

    for denoised, clean in zip(denoised_imgs, clean_imgs):
        denoised = denoised.squeeze()
        clean = clean.squeeze()

        # Calculate PSNR
        psnr = peak_signal_noise_ratio(clean, denoised, data_range=1.0)  # Normalize to [0, 1] range
        psnr_values.append(psnr)

        # Calculate SSIM
        ssim = structural_similarity(clean, denoised, data_range=1.0)
        ssim_values.append(ssim)

    return np.mean(psnr_values), np.mean(ssim_values)
```

Based on PSNR and SSIM our model works pretty well!



Object detection for airplanes

RCNN is an object detection framework that combines convolutional neural networks with region proposal methods. It works by first generating potential object regions (proposals) using an external algorithm, such as Selective Search. These regions are then passed through a CNN to extract feature representations. A classifier, typically SVM, is used to categorize each region, and a bounding box regressor refines the coordinates of the predicted objects.



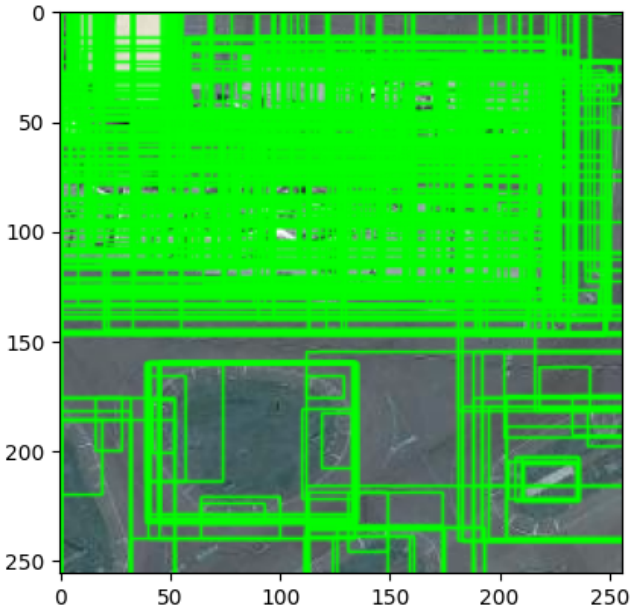
Selective search works by over-segmenting an image into small regions (superpixels) and then combining these regions in various ways to create larger regions that are likely to contain objects. This process uses a greedy algorithm based on **color** similarity, **texture**, and other visual cues to group the regions.

There are multiple images in the image folder and there are annotations for each image. They define the box around each airplane.

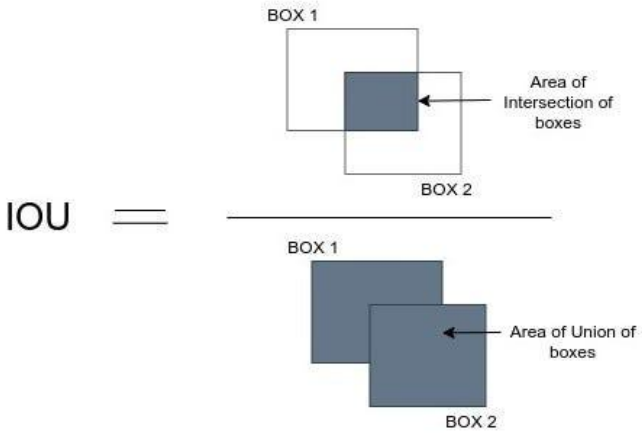
The annotation consists of x_min, y_min, x_max, y_max:

```
4
15 69 40 96
10 173 42 203
74 212 100 243
233 197 256 226
```

We use the fast mode of region proposals by using a selective search algorithm. In this mode, selective search generates region proposals more quickly, but with potentially fewer proposals than in the "accurate" mode.



IoU is a metric used to evaluate the accuracy of object detection models by measuring the overlap between the predicted and ground truth bounding boxes. A higher IoU indicates better performance, with 1 representing perfect overlap.



Gathering training dataset:

We apply the Selective Search algorithm to generate potential object regions from the image and calculate the IoU between each proposed region and the ground truth boxes. The code is designed to collect both positive and negative examples for training. Positive examples are regions with an IoU greater than 0.7, while negative examples are regions with an IoU less than 0.3, indicating they likely do not contain an airplane.

The process involves creating two lists, train_images and train_labels, to store the resized image patches and their associated labels (1 for positive examples, 0 for negative). The code limits the collection of positive examples to 30 and negative examples to 30 per image, ensuring balanced training data. After gathering enough samples, the script moves on to the next image.

For the transfer learning part, The **MobileNet** model is loaded with the ImageNet weights, including the top classification layers, and then all layers except the last two are frozen to prevent their weights from being updated during training. A custom Dense layer with 1 unit and a sigmoid activation function is added to the output of the MobileNet base model, making the model suitable for binary classification. The model is then compiled using the Adam optimizer and binary cross-entropy loss function.

```python
import tensorflow as tf

mobilenet = tf.keras.applications.MobileNet(
    include_top=True,  # Exclude the classification layer
    weights='imagenet',
    input_shape=None,  # Specify input shape
    pooling=None,  # Use global average pooling
    classes=1000
)

for layer in mobilenet.layers[:-2]:
    layer.trainable = False

last_output = mobilenet.output

# Add a custom Dense layer with sigmoid activation for binary classification
x = tf.keras.layers.Dense(1, activation='sigmoid')(last_output)

model = tf.keras.Model(mobilenet.input, x)

model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['acc']
)

model.summary()
```

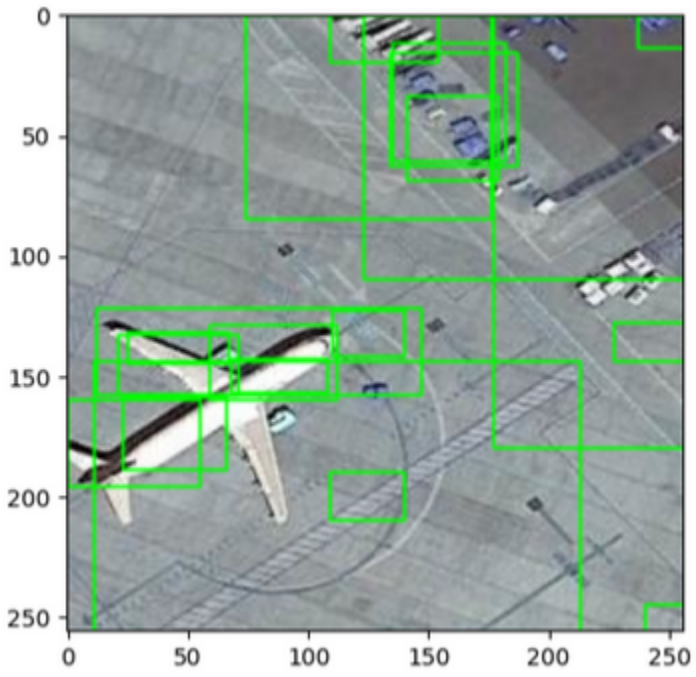
For the SVM dataset we considered all ground truth bounding boxes as positive examples and those which had an IOU less than 0.3 as false examples.

A custom Dense layer with 2 units and a softmax activation function is added to the MobileNet model's output to adapt it for a new classification task. The new Dense layer represents two classes in a classification problem, with the softmax activation ensuring that the outputs are probabilities. The model is then compiled with the Adam optimizer and hinge loss, which is commonly used for SVM tasks.

```
last_output = mobilenet.output

# Add a custom Dense layer with 2 units for your new task
x = tf.keras.layers.Dense(2, activation='softmax')(last_output)

new_model = tf.keras.Model(inputs=mobilenet.input, outputs=x)

new_model.compile(
    optimizer='adam',
    loss='hinge',
    metrics=['acc']
)
new_model.summary()
```

```
hist_final = new_model.fit(np.array(svm_image),np.array(svm_label),batch_size=32,epochs = 5,verbose = 1,shuffle = True,validation_split = 0.05)
```

Testing:



Using RetinaNet model to detect airplanes

In this class, we define a custom dataset that returns an image and its boxes for each image name. Also at the end, the transformation will be done on the image.

```
class AirplaneDataset(Dataset):
    def __init__(self, transforms=None):
        self.transforms = transforms
        self.image_files = [
            f for f in os.listdir(config.img_dir) if f.endswith(('.jpg', '.png'))
        ]

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

    def __getitem__(self, idx):
        img_name = self.image_files[idx]
        img_path = os.path.join(config.img_dir, img_name)
        image = Image.open(img_path).convert("RGB")

        ann_file = os.path.join(config.ann_dir, f"{os.path.splitext(img_name)[0]}.csv")
        annotations = pd.read_csv(ann_file, header=None)

        num_boxes = int(annotations.iloc[0, 0])  # First row is the number of boxes

        # Skip if no boxes are found
        if num_boxes == 0:
            boxes = torch.empty(0, 4, dtype=torch.float32)
            labels = torch.empty(0, dtype=torch.int64)
        else:
            boxes = annotations.iloc[1:1 + num_boxes].values
            boxes = [[float(coord) for coord in box_str.split()] for box_str in boxes.reshape(-1)]
            boxes = torch.tensor(boxes, dtype=torch.float32)

            if boxes.dim() == 1:  # If boxes is 1-dimensional, add a dimension
                boxes = boxes.unsqueeze(0)
```

```
        # Create target dictionary
        target = {
            "boxes": boxes[:, :4],  # [x_min, y_min, x_max, y_max]
            "labels": torch.ones((num_boxes,), dtype=torch.int64),  # Airplane label
        }

        if self.transforms:
            image = self.transforms(image)
        else:  # if no transforms are specified, apply ToTensor()
            image = T.ToTensor()(image)

        return image, target
```

The dataset is preprocessed with transformations including conversion to tensors and adding random color jitter to avoid overfitting. It is split into training (80%) and testing (20%) subsets using random_split. Data loaders are created for both subsets, with batch processing, shuffling for training, and a custom collate_fn to handle varying object annotations. The Dataloader efficiently loads data in parallel using multiple workers.

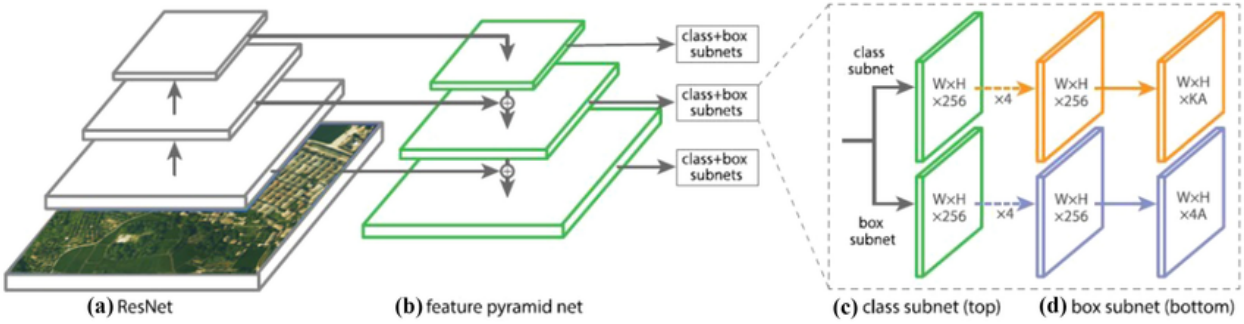
```
transform = T.Compose([
        T.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.2),  # Random color jitter
        T.ToTensor()  # Convert image to tensor
    ])

dataset = AirplaneDataset(transform)
train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = random_split(dataset, [train_size, test_size])

train_loader = DataLoader(train_dataset, batch_size=config.batch_size,
                          shuffle=True, collate_fn=lambda x: tuple(zip(*x)),
                          num_workers=config.num_workers)
test_loader = DataLoader(test_dataset, batch_size=config.batch_size,
                         shuffle=False, collate_fn=lambda x: tuple(zip(*x)),
                         num_workers=config.num_workers)
```

The training process involves transferring the model to the specified device (CPU/GPU) and iterating through a set number of epochs. During each epoch, the model processes batches of images, calculates losses, backpropagates gradients, and updates parameters using the optimizer. After each epoch, the model is evaluated on the test dataset.

```
def train_model(model, train_loader, test_loader, optimizer):
    model.to(config.device)

    for epoch in range(config.num_epochs):
        model.train()
        total_loss = 0
        for images, targets in train_loader:

            images = [img.to(config.device) for img in images]
            targets = [{k: v.to(config.device) for k, v in t.items()} for t in targets]

            optimizer.zero_grad()
            loss_dict = model(images, targets)
            losses = sum(loss for loss in loss_dict.values())
            losses.backward()
            optimizer.step()
            total_loss += losses.item()

        print(f"Epoch {epoch + 1}/{config.num_epochs}, Loss: {total_loss:.4f}")

        results = evaluate_model_multiple_iou_thresholds(model, test_loader)

        print("mAP for IoU thresholds:")
        iou_thresholds = [0.5, 0.75, 0.9]

        for iou, key in zip(iou_thresholds, ['map_50', 'map_75', 'map_90']):
            value = results.get(key, -1)
            print(f"IoU {iou}: {value:.4f}")
```

RetinNet replaces the traditional two-stage approach (like in Faster R-CNN) with a single unified network that predicts bounding boxes and class scores directly from input images in a single pass. The model is built on a feature pyramid network (FPN) and ResNet backbone, which extracts multi-scale features from images. RetinaNet is known for its ability to achieve state-of-the-art accuracy while maintaining high computational efficiency.

The standout feature of RetinaNet is its use of **Focal Loss**, which solves the problem of class imbalance by focusing training on hard-to-classify examples and down-weighting the contribution of easy negatives. Focal Loss applies a scaling factor to the loss for easy examples, effectively reducing their weight and allowing the model to concentrate on learning from harder examples.




```python
def get_retinanet_model():
    backbone = resnet_fpn_backbone(backbone_name='resnet18', weights='DEFAULT')
    model = RetinaNet(backbone=backbone, num_classes=config.num_classes)
    return model
```

We chose Resnet18 as the backbone of RetinaNet. It is lighter and faster than Resnet50.

IoU measures the overlap between predicted and ground truth bounding boxes, providing an average of how well the model predicts object locations. **mAP** calculates the average precision at multiple IoU thresholds (0.5 to 0.9). While IoU focuses on spatial accuracy, mAP considers both detection accuracy and precision-recall trade-offs.

```python
from torchmetrics.detection.mean_ap import MeanAveragePrecision

def evaluate_model_multiple_iou_thresholds(model, dataloader):
    model.eval()
    metric = MeanAveragePrecision(iou_type="bbox", iou_thresholds=[0.5, 0.75, 0.9])  # Add multiple IoU thresholds

    with torch.no_grad():
        for images, targets in dataloader:
            images = [img.to(config.device) for img in images]
            targets = [{k: v.to(config.device) for k, v in t.items()} for t in targets]
            outputs = model(images)

            metric.update(outputs, targets)

    results = metric.compute()
    return results  # Returns results with mAP for each IoU threshold
```

The evaluation of RetinaNet:

```
Epoch 5/10, Loss: 52.2071
mAP for IoU thresholds:
IoU 0.5: 0.7136
IoU 0.75: 0.2474
IoU 0.9: -1.0000
Epoch 6/10, Loss: 48.7602
mAP for IoU thresholds:
IoU 0.5: 0.7499
IoU 0.75: 0.3157
IoU 0.9: -1.0000
Epoch 7/10, Loss: 45.1293
mAP for IoU thresholds:
IoU 0.5: 0.7536
IoU 0.75: 0.3292
IoU 0.9: -1.0000
Epoch 8/10, Loss: 42.2332
mAP for IoU thresholds:
IoU 0.5: 0.7809
IoU 0.75: 0.3076
IoU 0.9: -1.0000
Epoch 9/10, Loss: 39.6986
mAP for IoU thresholds:
IoU 0.5: 0.7808
IoU 0.75: 0.3623
IoU 0.9: -1.0000
Epoch 10/10, Loss: 37.9494
mAP for IoU thresholds:
IoU 0.5: 0.7886
IoU 0.75: 0.3980
IoU 0.9: -1.0000
```

The model works better for images with an IoU of more than 0.5.