REPORT

1. Introduction

- About the internship program (organization/college, purpose).
- Scope of the internship (practical exposure, applying theoretical knowledge).
- Duration and objectives.

2. Background

- Importance of Python in modern computing (image processing, AI, visualization).
- Brief on each domain covered (image transformation, steganography, colorization, AI text-to-image).
- Tools and libraries (PIL, OpenCV, matplotlib, tkinter, PyTorch, Diffusion models).

3. Learning Objectives

- Apply Python programming for solving real-world problems.
- Gain hands-on experience in image processing, GUI development, and AI.
- Understand the workflow of model training/inference.
- Develop technical writing/documentation skills.

4. Activities and Tasks

Here we cover **Task1–Task4** in detail. Each task will have:

- **Objective**
- **Methodology**
- **Implementation (with code snippets + explanation)**
- **Results (expected outputs, screenshots placeholder)**
- **Analysis/Reflection**

4.1 Task 1 – Image Transformations with Python

4.2 Task 2 – Image Steganography with LSB

4.3 Task 3 – Conditional Image Colorization

4.4 Task 4 – AI-Generated Art using Diffusion Models

5. Skills and Competencies Developed

- Python programming
- Image processing
- GUI design with tkinter
- Deep learning inference (PyTorch)
- Prompt engineering for generative AI
- Documentation and project structuring

6. Feedback and Evidence

- Feedback from mentors/internship coordinators.
- Evidence: GitHub repo links, screenshots of results, working demos.

7. Challenges and Solutions

- Technical issues (dependencies, CUDA, model loading).
- Conceptual challenges (understanding U-Net, LSB algorithm, diffusion).
- Solutions (debugging, literature review, testing).

8. Outcomes and Impact

- Stronger Python foundation.
- Ability to design end-to-end mini-projects.
- Confidence in presenting and explaining projects.

9. Conclusion

- Summary of experience.
- Key takeaways.
- Future scope (extend colorization to video, improve steganography security, optimize diffusion prompts).

4.1 Task 1 – Experiment with Various Loss Functions

Objective

The objective of this task is to explore how different loss functions affect the performance of a U-Net based image colorization model. Two losses were tested:

1. Mean Squared Error (MSE): Pixel-wise error between predicted and ground truth images.
2. Perceptual Loss (LPIPS with VGG backbone): Measures similarity in a high-dimensional feature space, emphasizing perceived visual quality rather than strict pixel accuracy.

The experiment highlights trade-offs between numerical accuracy (MSE, PSNR, SSIM) and perceptual realism in image reconstruction.

Methodology

1. Dataset
   - CIFAR-10 dataset (32×32 RGB images).
   - Images were converted to grayscale inputs while original color images served as targets.
2. Model Architecture
   - U-Net with:
     - Encoder: Convolution, BatchNorm, ReLU, Dropout.
     - Decoder: Transposed convolution, skip connections.
   - Output layer uses tanh() to normalize RGB values between -1 and 1.
3. Loss Functions Compared
   - MSE Loss (nn.MSELoss)
     - Penalizes squared pixel errors.
     - Promotes accurate color matching but may yield dull images.
   - Perceptual Loss (LPIPS with VGG)
     - Uses pretrained VGG features to compute feature-space distance.
     - Captures texture and style similarity, producing more visually appealing colors.
4. Training
   - Optimizer: Adam (lr=0.001)

    - Epochs: 30
    - Batch size: 64

5. Evaluation Metrics
    - MSE – average reconstruction error.
    - LPIPS – perceptual distance.
    - PSNR – pixel-level fidelity (higher = better).
    - SSIM – structural similarity (closer to 1 = better).
    - Accuracy / Precision / Recall / F1 – evaluated using binning of color channels.
    - Confusion Matrices – visualize per-channel prediction quality.
6. Visualization
    - For each test sample:
        - Input grayscale image.
        - Ground truth color image.
        - Predicted output.
    - Saved inside models/ folder.

Implementation

1. U-Net Colorization

```
class UNetColorization(nn.Module):
    def __init__(self):
        super(UNetColorization, self).__init__()
        # Encoder
        self.enc1 = nn.Sequential(nn.Conv2d(1, 128, 3, padding=1), nn.BatchNorm2d(128), nn.ReLU(), nn.Dropout(0.4))
        self.enc2 = nn.Sequential(nn.Conv2d(128, 256, 3, stride=2, padding=1), nn.BatchNorm2d(256), nn.ReLU(),
nn.Dropout(0.4))
        self.enc3 = nn.Sequential(nn.Conv2d(256, 512, 3, stride=2, padding=1), nn.BatchNorm2d(512), nn.ReLU(),
nn.Dropout(0.4))
        # Decoder
        self.dec1 = nn.Sequential(nn.ConvTranspose2d(512, 256, 3, stride=2, padding=1, output_padding=1),
nn.BatchNorm2d(256), nn.ReLU())
```

```
        self.dec2 = nn.Sequential(nn.ConvTranspose2d(512, 128, 3, stride=2, padding=1, output_padding=1),
nn.BatchNorm2d(128), nn.ReLU())
        self.dec3 = nn.Conv2d(256, 3, 3, padding=1)
        self.tanh = nn.Tanh()

    def forward(self, x):
        e1 = self.enc1(x)
        e2 = self.enc2(e1)
        e3 = self.enc3(e2)
        d1 = self.dec1(e3)
        d2 = self.dec2(torch.cat([d1, e2], dim=1))
        d3 = self.dec3(torch.cat([d2, e1], dim=1))
        return self.tanh(d3)
```

2. Training with Different Losses

```
def train_model(model, train_loader, loss_type, epochs=30):
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    mse_loss = nn.MSELoss()
    loss_fn_vgg = lpips.LPIPS(net='vgg').to(device)

    for epoch in range(epochs):
        for images, _ in train_loader:
            grayscale = rgb_to_gray(images).to(device)
            outputs = model(grayscale)

            if loss_type == 'mse':
                loss = mse_loss(outputs, images)
            elif loss_type == 'perceptual':
                loss = loss_fn_vgg(outputs, images).mean()

            optimizer.zero_grad()
```

```
    loss.backward()

    optimizer.step()
```

3. Evaluation Metrics

```
mse = nn.MSELoss()(outputs, images).item()

perceptual = loss_fn_vgg(outputs, images).mean().item()

psnr_value = psnr(images_np, outputs_np, data_range=1.0)

ssim_value = ssim(images_np, outputs_np, data_range=1.0, channel_axis=-1)
```

Results

1. Quantitative Comparison
   - *MSE model:* Lower pixel error, higher PSNR, but duller images.
   - *Perceptual model:* Lower LPIPS (better perceptual similarity), slightly lower PSNR/SSIM.
2. Visualization (placeholders)
   - models/sample_0_mse.png – Grayscale → Ground truth → MSE prediction.
   - models/sample_0_perceptual.png – Same input but perceptual loss prediction.
3. Training Curves
   - Loss plots comparing MSE vs Perceptual (models/loss_comparison.png).
4. Confusion Matrices
   - Color-channel prediction quality for both models.

Analysis / Reflection

- MSE Loss
  - Pros: Stable training, good numerical accuracy, high PSNR/SSIM.
  - Cons: Tends to produce smooth but desaturated colors.
- Perceptual Loss
  - Pros: Captures textures, edges, and artistic quality better. Colors look more natural.
  - Cons: Pixel-level accuracy is weaker (higher MSE, lower PSNR).
- Key Takeaway

- MSE loss is better for objective evaluation, while perceptual loss improves subjective visual quality.
- A hybrid (MSE + Perceptual) could balance accuracy and realism

4.2 Task 2 – Artistic Style Transfer in Colorization

Objective

The purpose of this task is to extend image colorization by combining it with artistic style transfer. Instead of simply restoring realistic colors, the model allows users to apply a predetermined artistic style (e.g., Van Gogh, Monet, Ukiyo-e) to grayscale photographs. This task tests the ability to integrate deep learning models for both colorization (U-Net) and style transfer (VGG-based perceptual loss) within a user-friendly GUI.

Methodology

1. Colorization Backbone
   - A U-Net model was trained on grayscale-to-color image mapping.
   - The model learns low-level features (edges, textures) in the encoder and reconstructs colored images in the decoder.
   - Output layer uses tanh() activation to normalize pixel values between -1 and 1.
2. Style Transfer Module
   - Pretrained VGG-19 network extracts content and style features.
   - Gram matrices represent texture/style information at multiple layers.
   - Optimization balances:
     - Content loss (retain structure of original image).
     - Style loss (transfer artistic patterns).
3. Color Enhancement
   - Post-processing exaggerates saturation and brightness using HSV transformations.
   - This makes styled outputs more vivid and visually appealing.
4. Graphical User Interface (GUI)
   - Built with tkinter for simplicity.
   - Users can:
     - Upload grayscale images.
     - Select an artistic style from a dropdown.
     - Click *"Colorize & Style"* to process the image.
     - Save styled output as PNG/JPG.

Implementation

U-Net Colorization

```
class UNetColorization(nn.Module):
    def __init__(self):
        super(UNetColorization, self).__init__()
        # Encoder
        self.enc1 = nn.Sequential(nn.Conv2d(1, 128, 3, padding=1), nn.BatchNorm2d(128), nn.ReLU(), nn.Dropout(0.4))
        self.enc2 = nn.Sequential(nn.Conv2d(128, 256, 3, stride=2, padding=1), nn.BatchNorm2d(256), nn.ReLU(),
nn.Dropout(0.4))
        self.enc3 = nn.Sequential(nn.Conv2d(256, 512, 3, stride=2, padding=1), nn.BatchNorm2d(512), nn.ReLU(),
nn.Dropout(0.4))
        # Decoder
        self.dec1 = nn.Sequential(nn.ConvTranspose2d(512, 256, 3, stride=2, padding=1, output_padding=1),
nn.BatchNorm2d(256), nn.ReLU())
        self.dec2 = nn.Sequential(nn.ConvTranspose2d(512, 128, 3, stride=2, padding=1, output_padding=1),
nn.BatchNorm2d(128), nn.ReLU())
        self.dec3 = nn.Conv2d(256, 3, 3, padding=1)
        self.tanh = nn.Tanh()


    def forward(self, x):
        e1 = self.enc1(x)
        e2 = self.enc2(e1)
        e3 = self.enc3(e2)
        d1 = self.dec1(e3)
        d2 = self.dec2(torch.cat([d1, e2], dim=1))
        d3 = self.dec3(torch.cat([d2, e1], dim=1))
        return self.tanh(d3)
```

Style Transfer Loss

```
class VGGStyleLoss(nn.Module):
```

```
    def __init__(self):
        super(VGGStyleLoss, self).__init__()
        vgg = torchvision.models.vgg19(weights=torchvision.models.VGG19_Weights.DEFAULT).features.to(device).eval()
        self.layers = {'0':'conv1_1', '5':'conv2_2', '10':'conv3_2', '19':'conv4_2', '28':'conv5_2'}
        self.model = nn.ModuleDict({name: nn.Sequential() for name in self.layers.values()})
        current_name = None
        for i, layer in enumerate(vgg.children()):
            for key, name in self.layers.items():
                if str(i) == key: current_name = name
            if current_name:
                if isinstance(layer, nn.ReLU):
                    self.model[current_name].add_module(str(len(self.model[current_name])), nn.ReLU(inplace=False))
                else:
                    self.model[current_name].add_module(str(len(self.model[current_name])), layer)
```

Style Application

```
def apply_style_transfer(colorized, style_image, vgg, content_weight=1e2, style_weight=1e7, steps=200):
    opt_img = colorized.detach().clone().requires_grad_(True)
    optimizer = optim.Adam([opt_img], lr=0.01)

    content_features = vgg(colorized)
    style_features = vgg(style_image)
    style_grams = {layer: gram_matrix(style_features[layer]) for layer in style_features}

    for _ in range(steps):
        optimizer.zero_grad()
        out_features = vgg(opt_img)
        content_loss = torch.mean((out_features['conv4_2'] - content_features['conv4_2']) ** 2)
        style_loss = sum(torch.mean((gram_matrix(out_features[l]) - style_grams[l]) ** 2) for l in style_grams) / 
len(style_grams)
        total_loss = content_weight * content_loss + style_weight * style_loss
```

```
        total_loss.backward(retain_graph=True)

        optimizer.step()


    return opt_img.detach()
```

GUI Workflow

- Dropdown menu for style selection:

```
self.style_var = tk.StringVar(value=list(style_images.keys())[0])

tk.OptionMenu(root, self.style_var, *style_images.keys()).pack()
```

- Buttons for upload, process, and save:

```
tk.Button(root, text="Upload", command=self.upload_image).pack()

tk.Button(root, text="Colorize & Style", command=self.process_image).pack()

tk.Button(root, text="Save Output", command=self.save_image).pack()
```

Results

1. Expected Outputs
    - A grayscale photo is uploaded.
    - The U-Net model produces a colorized base image.
    - Style transfer refines the result by overlaying textures and colors from the chosen style (Van Gogh → swirling brush strokes, Monet → pastel tones, Ukiyo-e → flat shading).
    - Output is enhanced with higher saturation and brightness.
2. Screenshots (placeholders)
    - Example: Black-and-white portrait → *Van Gogh* style → Output portrait with expressive blue/yellow tones.
3. User Workflow

1. Upload grayscale photo.
2. Choose artistic style from dropdown.
3. Click *"Colorize & Style"* to generate result.
4. Preview and save the output.

Analysis / Reflection

- Strengths

- Successfully merges two domains: colorization + style transfer.
- Provides multiple artistic choices via dropdown.
- GUI makes the tool accessible for non-programmers.

- Limitations
  - Style transfer optimization is computationally expensive (200 steps per image).
  - Requires pretrained weights and style reference images to be present.
  - High memory usage when working with large images.
- Possible Improvements
  - Replace iterative optimization with a feed-forward style transfer network for real-time results.
  - Allow users to upload custom style images instead of only predefined ones.
  - Add GPU/CPU usage indicator in the GUI to improve UX.

4.3 Task 3 – Conditional Image Colorization

Objective

The goal of this task is to build an **interactive application** that colorizes grayscale images based on **user-defined conditions**. Instead of the model automatically predicting all colors, users can manually assign colors to specific regions (e.g., blue for the sky, green for grass). This enables **greater control** and allows integration of **human guidance** into the colorization process.

Methodology

1. **Input and Preprocessing**
   - Users upload a **grayscale image** (.jpg, .png, etc.).
   - The image is resized to **512×512 pixels** (bicubic interpolation).
   - The grayscale image is converted to an RGB array to allow color overlays.
2. **Region Selection**
   - Users select regions by drawing bounding boxes on the image canvas.
   - Regions are stored as coordinates (x1, y1, x2, y2) along with a chosen RGB color.
3. **Color Application**
   - **Direct Coloring:** The selected color replaces pixel values in the region while preserving brightness.
   - **Blended Coloring:** Colors are smoothly blended with the grayscale intensity to produce natural tones.
4. **Graphical User Interface (GUI)**
   - Built with **tkinter** for usability.
   - Features:
     - Upload grayscale images.
     - Draw bounding boxes with the mouse.
     - Choose colors via a color picker.
     - Toggle between "Blend Mode" and "Direct Mode."
     - Apply colors and preview results side-by-side.
     - Save the output as PNG/JPG.

Implementation

Preprocessing

```
def preprocess_image(image_path, size=(512, 512)):
    image = Image.open(image_path).convert('L')  # grayscale
    image = image.resize(size, Image.BICUBIC)
    return image
```

Color Application (Blended Mode Example)

```
def apply_color_with_blending(grayscale_image, regions, blend_strength=0.7):
    rgb_image = grayscale_image.convert('RGB')
    rgb_array = np.array(rgb_image, dtype=np.float32)
    result_array = rgb_array.copy()

    for region, color in regions:
        x1, y1, x2, y2 = region
        gray_values = rgb_array[y1:y2, x1:x2, 0]

        # Create colored region
        colored_region = np.zeros((y2-y1, x2-x1, 3))
        for c in range(3):
            colored_region[:, :, c] = gray_values * (color[c] / 255.0)

        # Blend original with colorized region
        original_region = rgb_array[y1:y2, x1:x2]
        result_array[y1:y2, x1:x2] = (
            original_region * (1 - blend_strength) +
            colored_region * blend_strength
        )
```

```python
    return Image.fromarray(result_array.astype(np.uint8))
```

GUI Highlights

- **Mouse Interaction:** Draw rectangles on canvas → maps to actual image coordinates.
- **Color Picker:** Uses tkinter.colorchooser for selecting RGB values.
- **Canvas Display:** Original (left) vs Colorized (right).
- **Export Options:** Save both comparison plots and full-resolution outputs.

```python
# Example GUI snippet
self.color_btn = tk.Button(control_frame, text="Choose Color",
                 command=self.choose_color, bg='lightgreen', font=('Arial', 11))
self.process_btn = tk.Button(control_frame, text="Apply Colors",
                 command=self.process_image, bg='lightcoral', font=('Arial', 12, 'bold'))
```

Results

1. **Expected Outputs**
   - Users can assign **custom colors** to objects of interest.
   - Two visualizations provided:
     - Side-by-side comparison (original grayscale vs. user-colored).
     - Saved high-resolution output image.
2. **Screenshots (placeholders)**
   - Example: Grayscale landscape → user selects *sky region* → chooses **blue** → output shows a blue sky with natural shading.
3. **User Workflow**

1. Upload grayscale image.
2. Draw bounding boxes over regions of interest.
3. Choose desired color (RGB).
4. Click **Apply Colors** to preview results.
5. Save the output.

Analysis / Reflection

- **Strengths**
  - Provides **human control** over AI-assisted colorization.
  - Interactive GUI makes it easy for non-technical users.
  - Blending ensures more **realistic** results compared to flat coloring.
- **Limitations**
  - Color application is **manual** (does not automatically segment sky, grass, etc.).
  - Rectangular bounding boxes may overlap unintended regions.
  - No pretrained deep learning inference here; it's a **rule-based approach**.
- **Possible Improvements**
  - Integrate with **segmentation models** (e.g., U^2-Net, Mask R-CNN) for automatic region selection.
  - Combine with pretrained **U-Net colorization** model to enhance realism.
  - Add **undo/redo** functionality for user editing.

4.4 Task 4 – Dataset Augmentation to Improve Colorization

Objective

The goal of this task was to **improve the performance of an image colorization model** by introducing **dataset augmentation** during training.
By artificially expanding the training dataset with transformations such as **rotation, flipping, affine transformations, and brightness adjustments**, the model learns more robust representations, generalizes better, and produces **higher-quality colorized outputs**.

Methodology

1. **Dataset**
   - CIFAR-10 dataset (32×32 color images).
   - Training images were augmented before being fed to the model.
   - Grayscale inputs were obtained from the **L channel of LAB color space**, while the **AB channels served as targets**.
2. **Data Augmentation Techniques**
   - **Random Rotation (±30°)** → increases rotational invariance.
   - **Random Horizontal Flip** → prevents bias toward left/right orientation.
   - **Random Affine Transformations** (translation, shear) → simulates object movement.
   - **Color Jitter** (brightness, contrast, saturation changes) → improves robustness to lighting conditions.
3. **Model Architecture**
   - **Encoder–Decoder Convolutional Network (ColorizationNet)**:
     - Encoder: multiple dilated convolutional layers with BatchNorm + ReLU.
     - Decoder: transposed convolutions and final **tanh activation** for AB color channels.
   - Lightweight yet effective for low-resolution datasets like CIFAR-10.
4. **Loss Function**
   - **Hybrid Loss = 0.7 × MSE + 0.3 × Perceptual Loss**
   - MSE ensures **numerical accuracy**, while **VGG16-based perceptual loss** encourages outputs that are **visually realistic**.
5. **Training Setup**
   - Optimizer: **Adam (lr = 0.001)**.

- Scheduler: **ReduceLROnPlateau** (reduces LR if validation loss plateaus).
- Epochs: **10** (sufficient due to augmentation).
- Device: CUDA-enabled GPU if available, otherwise CPU.

6. **Evaluation**
   - Visual comparison of **original → grayscale → colorized** images.
   - Quantitative metrics (MSE, PSNR, SSIM, LPIPS) could be computed to compare **augmented vs non-augmented training**.

Implementation

1. Data Augmentation Setup

```
train_transform = transforms.Compose([
    transforms.RandomRotation(30),
    transforms.RandomHorizontalFlip(),
    transforms.RandomAffine(degrees=0, translate=(0.3, 0.3), shear=0.3),
    transforms.ColorJitter(brightness=0.3, contrast=0.3, saturation=0.3),
    transforms.ToTensor()
])
```

2. Colorization Network

```
class ColorizationNet(nn.Module):
    def __init__(self):
        super(ColorizationNet, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 64, 5, stride=1, padding=4, dilation=2),
            nn.BatchNorm2d(64), nn.ReLU(),
            nn.Conv2d(64, 128, 5, stride=1, padding=4, dilation=2),
            nn.BatchNorm2d(128), nn.ReLU(),
            nn.Conv2d(128, 256, 5, stride=1, padding=4, dilation=2),
            nn.BatchNorm2d(256), nn.ReLU(),
            nn.Dropout(0.3)
        )
```

```
    self.decoder = nn.Sequential(
      nn.ConvTranspose2d(256, 128, 5, stride=1, padding=4, dilation=2),
      nn.BatchNorm2d(128), nn.ReLU(),
      nn.ConvTranspose2d(128, 64, 5, stride=1, padding=4, dilation=2),
      nn.BatchNorm2d(64), nn.ReLU(),
      nn.Conv2d(64, 2, 5, stride=1, padding=4, dilation=2),
      nn.Tanh()
    )
```

3. Training with Hybrid Loss

```
loss_mse = criterion_mse(outputs, ab)
loss_perceptual = criterion_perceptual(outputs, ab)
loss = 0.7 * loss_mse + 0.3 * loss_perceptual
```

4. Visualization

```
def visualize_all_three(original_images, grayscale_images, colorized_images, n=5):
  fig = plt.figure(figsize=(3*n, 4))
  for i in range(n):
    plt.subplot(1, 3*n, 3*i+1)
    plt.imshow(original_images[i]); plt.title("Original"); plt.axis("off")
    plt.subplot(1, 3*n, 3*i+2)
    plt.imshow(grayscale_images[i], cmap='gray'); plt.title("Grayscale"); plt.axis("off")
    plt.subplot(1, 3*n, 3*i+3)
    plt.imshow(colorized_images[i]); plt.title("Colorized"); plt.axis("off")
  plt.show()
```

Results

1. **Qualitative Comparison**
   - Without augmentation:
     - Model tends to **overfit training data**, producing washed-out colors.
   - With augmentation:

        - Model generates **sharper, more diverse, and realistic colors**.

(Insert before-after comparison screenshots here:)

    - results/no_aug_sample.png
    - results/with_aug_sample.png

2. **Expected Output**
    - Grayscale → Colorized comparison after augmentation shows **better generalization**.

Example (placeholder):

Original Grayscale Colorized (Augmented)

3. **Quantitative Improvements** *(expected trends)*
    - Lower MSE & LPIPS.
    - Higher SSIM & PSNR.
    - More stable training curve (less overfitting).

Analysis / Reflection

- **Data augmentation significantly enhanced generalization** by exposing the model to diverse orientations, lighting, and distortions.
- Outputs appeared **more vivid and less prone to dull colors**, particularly for complex objects (cars, animals).
- **Limitation:**
    - Augmentation increases training time and may introduce unrealistic samples if transformations are too strong.
- **Future improvement:**
    - Combine augmentation with **GAN-based training** for even more realistic results.