Import Libraries

In [41]:
```
import os
import shutil
import kagglehub
import random
import os
from pathlib import Path
import torch
from torchvision import transforms, datasets
from torchvision.datasets import ImageFolder
from torch.utils.data import Dataset, DataLoader
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

Load dataset from kaggle

In [7]:
```
# Download latest version
path = kagglehub.dataset_download("sc0v1n0/animal-picture-set-cat-turtle")

print("Path to dataset file:", path)
```

```
Path to dataset file: /kaggle/input/animal-picture-set-cat-turtle
```

In [8]:
```
dataset_path = "/kaggle/input/animal-picture-set-cat-turtle"
destination_path = "/content/animal-picture-set-cat-turtle"

# Create the destination directory if it doesn't exist
os.makedirs(destination_path, exist_ok=True)

# Copy the contents of the dataset directory to the destination directory
shutil.copytree(dataset_path, destination_path, dirs_exist_ok=True)

print(f"Dataset copied to: {destination_path}")
```

```
Dataset copied to: /content/animal-picture-set-cat-turtle
```

Rearrange the dataset for correct structure

In [9]:
```
SOURCE_DIR ="/content/animal-picture-set-cat-turtle/cat_and_turtle"
DEST_DIR = "/content/Dataset"

#create unlabel and label data folders


SOURCE_DIR = Path(SOURCE_DIR)
```

```
DEST_DIR = Path(DEST_DIR)

(DEST_DIR /"unlabeled").mkdir(parents = True, exist_ok = True)
for split in ["train", "val", "test"]:
  for cls in ["cat", "turtle"]:
    (DEST_DIR/"labeled"/split/cls).mkdir(parents = True, exist_ok = True)

#collect all data
all_images = []
for subdir in ["animals_enormous", "animals_medium", "animals_small"]:
  for split in ["train", "valid"]:
    for cls in ["cats", "turtles"]:
      class_dir = SOURCE_DIR/subdir/split/cls
      if not class_dir.exists():
        continue
      label = "cat" if cls == "cats" else "turtle" # Modified this line
      for img_path in class_dir.glob("*.*"):
        all_images.append((img_path, label))

print(f"Found {len(all_images)} images")

random.shuffle(all_images)

num_total = len(all_images)
num_unlabeled = int(num_total*0.8)
num_labeled = num_total - num_unlabeled
num_train = int(num_labeled*0.6)
num_val = int(num_labeled*0.2)
num_test = num_labeled - num_train - num_val

print(f"Total: {num_total}, Unlabeled: {num_unlabeled}, Labeled: {num_labeled}")
print(f"Train: {num_train}, Val: {num_val}, Test: {num_test}")

for i, (img_path, label) in enumerate(all_images):
  img_name = f"{label}_{i}.jpg"
  if i <num_unlabeled:
    shutil.copy(img_path, DEST_DIR/"unlabeled"/img_name)
  else:
    if i < num_unlabeled + num_train:
      split = "train"
    elif i < num_unlabeled + num_train + num_val:
      split = "val"
    else:
      split = "test"
    shutil.copy(img_path, DEST_DIR/"labeled"/split/label/img_name)

print("done")
```

```
Found 1289 images
Total: 1289, Unlabeled: 1031, Labeled: 258
Train: 154, Val: 51, Test: 53
done
```

Add Gaussian and Salt and pepper noise

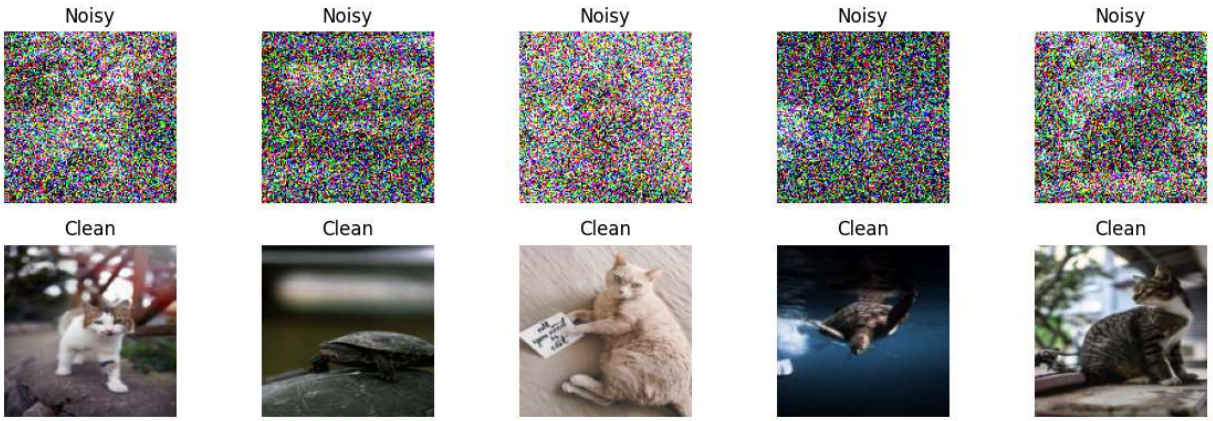
In [10]:
```
class AddNoiseDataset(Dataset):
  def __init__(self,root_dir, image_size = 128):
    self.image_paths = list(Path(root_dir).glob("*.*"))
    self.transform_clean = transforms.Compose([
        transforms.Resize((image_size, image_size)),
        transforms.ToTensor()
    ])

  def add_gaussian_noise(self,img, mean= 0 , std = 1):
    noise = torch.randn_like(img)*std + mean
    return torch.clamp(img+noise, 0, 1)

  def add_salt_pepper_noise(self, img, amount = 0.02):
    img_np = img.numpy()
    noisy = img_np.copy()
    num_salt = np.ceil(amount*img_np.size*0.5).astype(int)
    num_pepper = np.ceil(amount*img_np.size*0.5).astype(int)

    # Generate random coordinates for salt noise
    salt_coords = [np.random.randint(0, dim, num_salt) for dim in img_np.shape]
    noisy[tuple(salt_coords)] = 1

    # Generate random coordinates for pepper noise
    pepper_coords = [np.random.randint(0, dim, num_pepper) for dim in img_np.shape]
    noisy[tuple(pepper_coords)] = 0

    return torch.tensor(noisy).float()

  def __getitem__(self, idx):
    img = Image.open(self.image_paths[idx]).convert("RGB")
    clean = self.transform_clean(img)

    #Add both noises
    noisy = self.add_gaussian_noise(clean)
    noisy = self.add_salt_pepper_noise(noisy)
    return noisy, clean

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

unlabeled_dir = "/content/Dataset/unlabeled"
batch_size  = 32

dataset = AddNoiseDataset(unlabeled_dir)
dataloader = DataLoader(dataset, batch_size = batch_size, shuffle = True)
print("Dataset noising ready")
```

```
Dataset noising ready
```

Plot some noised data withclean data

In [11]:
```
def show_noisy_clean_batch(dataloader,n = 5):

  noisy_imgs, clean_imgs = next(iter(dataloader))
```

```
  # plot some data
  plt.figure(figsize = (12,4))
  for i in range(n):
    #noisy images
    plt.subplot(2, n, i+1)
    plt.imshow(noisy_imgs[i].permute(1,2,0))
    plt.title("Noisy")
    plt.axis("off")

    #clean images
    plt.subplot(2, n, i+1+n)
    plt.imshow(clean_imgs[i].permute(1,2,0))
    plt.title("Clean")
    plt.axis("off")

  plt.tight_layout()
  plt.show()

show_noisy_clean_batch(dataloader)
```



unet Feature extractor

In [12]:

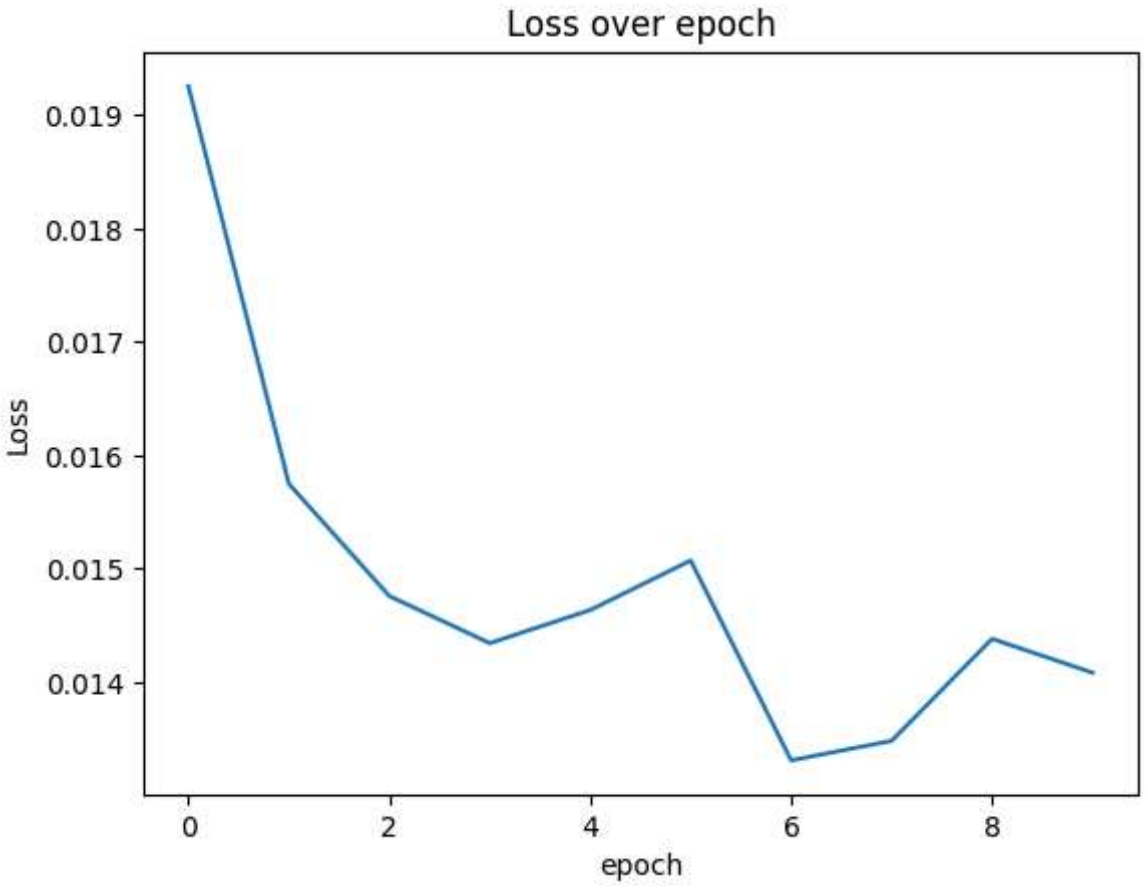
```
class DenoisingUnet(nn.Module):
  def __init__(self):
    super(DenoisingUnet, self).__init__()

    def conv_block(in_channels, out_channels):
      return nn.Sequential(
          nn.Conv2d(in_channels, out_channels, kernel_size = 3, padding = 1),
          nn.ReLU(inplace = True),
          nn.BatchNorm2d(out_channels),
          nn.Conv2d(out_channels ,out_channels, kernel_size = 3, padding = 1),
          nn.ReLU(inplace = True),
          nn.BatchNorm2d(out_channels)

      )

    self.pool = nn.MaxPool2d(kernel_size = 2, stride = 2)

    #Encoder
```

```
        self.c1 = conv_block(3,64)
        self.c2 = conv_block(64,128)
        self.c3 = conv_block(128,256)

        #Bottleneck
        self.c4 = conv_block(256,512)

        #Decoder
        self.u5 = nn.Upsample(scale_factor = 2, mode = "bilinear", align_corners = True
        self.c5 = conv_block(512+256,256)

        self.u6 = nn.Upsample(scale_factor = 2, mode = "bilinear", align_corners = True
        self.c6 = conv_block(256+128, 128)

        self.u7 = nn.Upsample(scale_factor = 2, mode = "bilinear", align_corners = True
        self.c7 = conv_block(128+64, 64)

        self.final = nn.Conv2d(64, 3, kernel_size = 1)

    def forward(self,x):
        #Encoder
        c1 = self.c1(x)
        p1 = self.pool(c1)

        c2 = self.c2(p1)
        p2 = self.pool(c2)

        c3 = self.c3(p2)
        p3 = self.pool(c3)


        #Bottleneck
        c4 = self.c4(p3)


        #Decoder
        u5 = self.u5(c4)
        u5 = torch.cat([u5,c3], dim = 1)
        c5 = self.c5(u5)

        u6 = self.u6(c5)
        u6 = torch.cat([u6, c2],dim=1)
        c6 = self.c6(u6)

        u7 = self.u7(c6)
        u7 = torch.cat([u7, c1], dim=1)
        c7 = self.c7(u7)

        out = self.final(c7)

        return torch.sigmoid(out)
```

Train to extract the features

In [ ]:
```
# Define training parameters
batch_size = 8
epochs = 10
learning_rate = 0.001

model = DenoisingUnet().cuda()
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr = learning_rate)

train_loader = DataLoader(dataset, batch_size = batch_size, shuffle=True)

def train(model, criterian, optimizer, train_loader, epochs):
  model.train()

  loss_history = []

  for epoch in range(epochs):
    epoch_loss = 0
    for batch_idx, (noisy,clean) in enumerate(train_loader):
      noisy, clean  = noisy.cuda(), clean.cuda()

      #forward pass
      optimizer.zero_grad()
      output = model(noisy)

      #Compute loss(MSE)
      loss = criterion(output, clean)

      #Backward pass
      loss.backward()
      optimizer.step()

      epoch_loss += loss.item()

    avg_epoch_loss = epoch_loss/len(train_loader)
    loss_history.append(avg_epoch_loss)

    print(f"Epoch [{epoch+1}/{epochs}], loss:{avg_epoch_loss:.4f}")

  return loss_history


loss_history = train(model, criterion, optimizer, train_loader, epochs)

torch.save(model.state_dict(),"denoising_unet.pth")
print("Model saved as 'denoisin_unet.pth'")
```

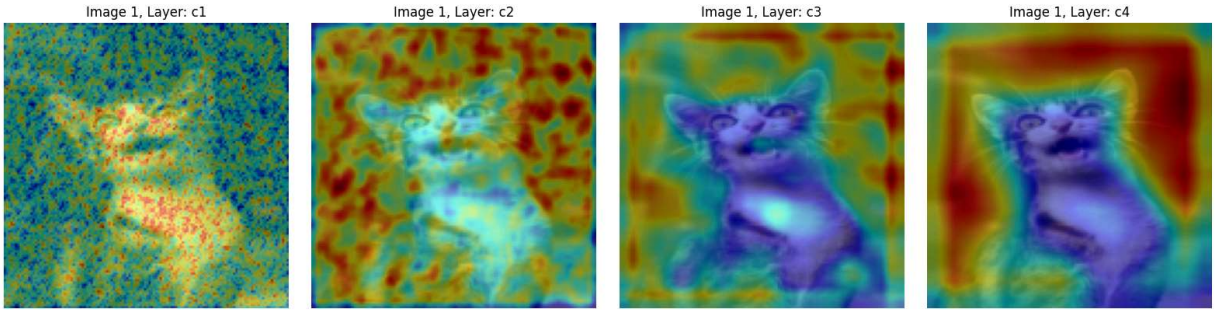
In [ ]:
```
#plot stats
plt.plot(range(epochs),loss_history)
plt.xlabel("epoch")
plt.ylabel("Loss")
plt.title("Loss over epoch")
plt.show()
```



Plot some features

In [67]:
```
import cv2

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

def overlay_heatmap(original_img, feature_map, alpha = 0.5):
  # Convert tensor to numpy
  img = original_img.permute(1,2,0).cpu().numpy()
  img = (img-img.min())/(img.max()-img.min())

  fmap = feature_map.cpu().numpy()
  fmap = (fmap- fmap.min())/(fmap.max()-fmap.min())

  fmap_resized = cv2.resize(fmap, (img.shape[1],img.shape[0]))

  heatmap = cv2.applyColorMap(np.uint8(255*fmap_resized),cv2.COLORMAP_JET)
  heatmap = cv2.cvtColor(heatmap, cv2.COLOR_BGR2RGB)/255.0

  overlay = (1-alpha) * img + alpha * heatmap
  return overlay

model = DenoisingUnet().to(device)
model.load_state_dict(torch.load("denoising_unet.pth", map_location=torch.device('c
model.eval()

# Get a batch of noisy and clean images from the dataloader
```

```
noisy_imgs, clean_imgs = next(iter(dataloader))

# Loop through the batch and plot heatmap overlays
for i in range(noisy_imgs.size(0)):
  noisy_img = noisy_imgs[i].unsqueeze(0).to(device)
  clean_img = clean_imgs[i]

  features = {}
  def get_activation(name):
    def hook(model, input, output):
      features[name] = output.detach()
    return hook

  # Register hooks
  hooks = [
      model.c1[0].register_forward_hook(get_activation("c1")),
      model.c2[0].register_forward_hook(get_activation("c2")),
      model.c3[0].register_forward_hook(get_activation("c3")),
      model.c4[0].register_forward_hook(get_activation("c4"))
  ]


  with torch.no_grad():
    output = model(noisy_img)

  # Remove hooks
  for hook in hooks:
    hook.remove()


  # plot in subplots
  fig, axes = plt.subplots(1, 4, figsize=(16, 4))
  layer_names = ["c1", "c2", "c3", "c4"]

  for j, layer_name in enumerate(layer_names):
      activation = features[layer_name].squeeze(0).cpu()
      mean_fmap = activation.mean(dim=0)
      overlay = overlay_heatmap(clean_img, mean_fmap)

      axes[j].imshow(overlay)
      axes[j].set_title(f"Image {i+1}, Layer: {layer_name}")
      axes[j].axis("off")

  plt.tight_layout()
  plt.show()
```



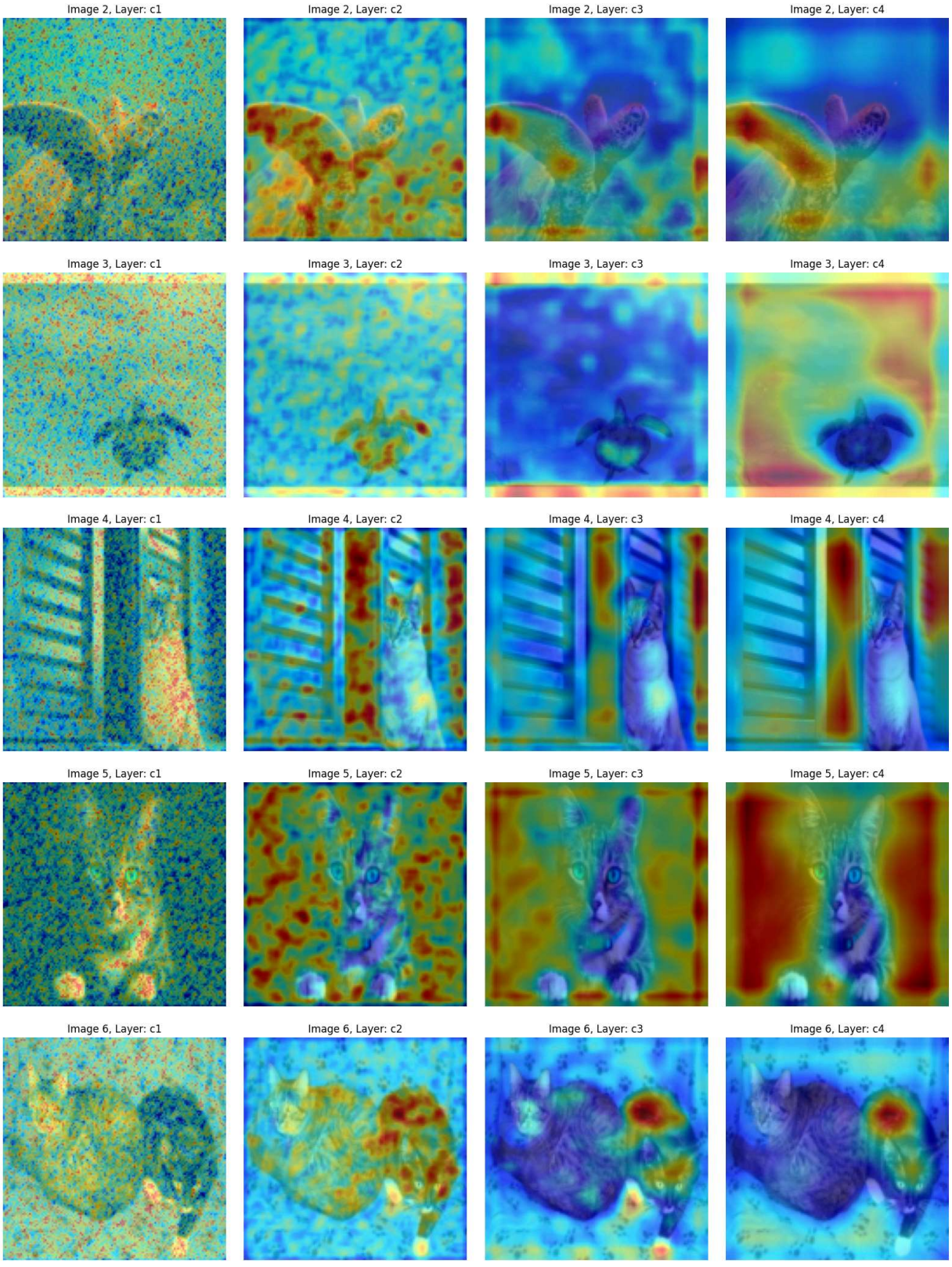


Image 2, Layer: c1
Image 2, Layer: c2
Image 2, Layer: c3
Image 2, Layer: c4
Image 3, Layer: c1
Image 3, Layer: c2
Image 3, Layer: c3
Image 3, Layer: c4
Image 4, Layer: c1
Image 4, Layer: c2
Image 4, Layer: c3
Image 4, Layer: c4
Image 5, Layer: c1
Image 5, Layer: c2
Image 5, Layer: c3
Image 5, Layer: c4
Image 6, Layer: c1
Image 6, Layer: c2
Image 6, Layer: c3
Image 6, Layer: c4

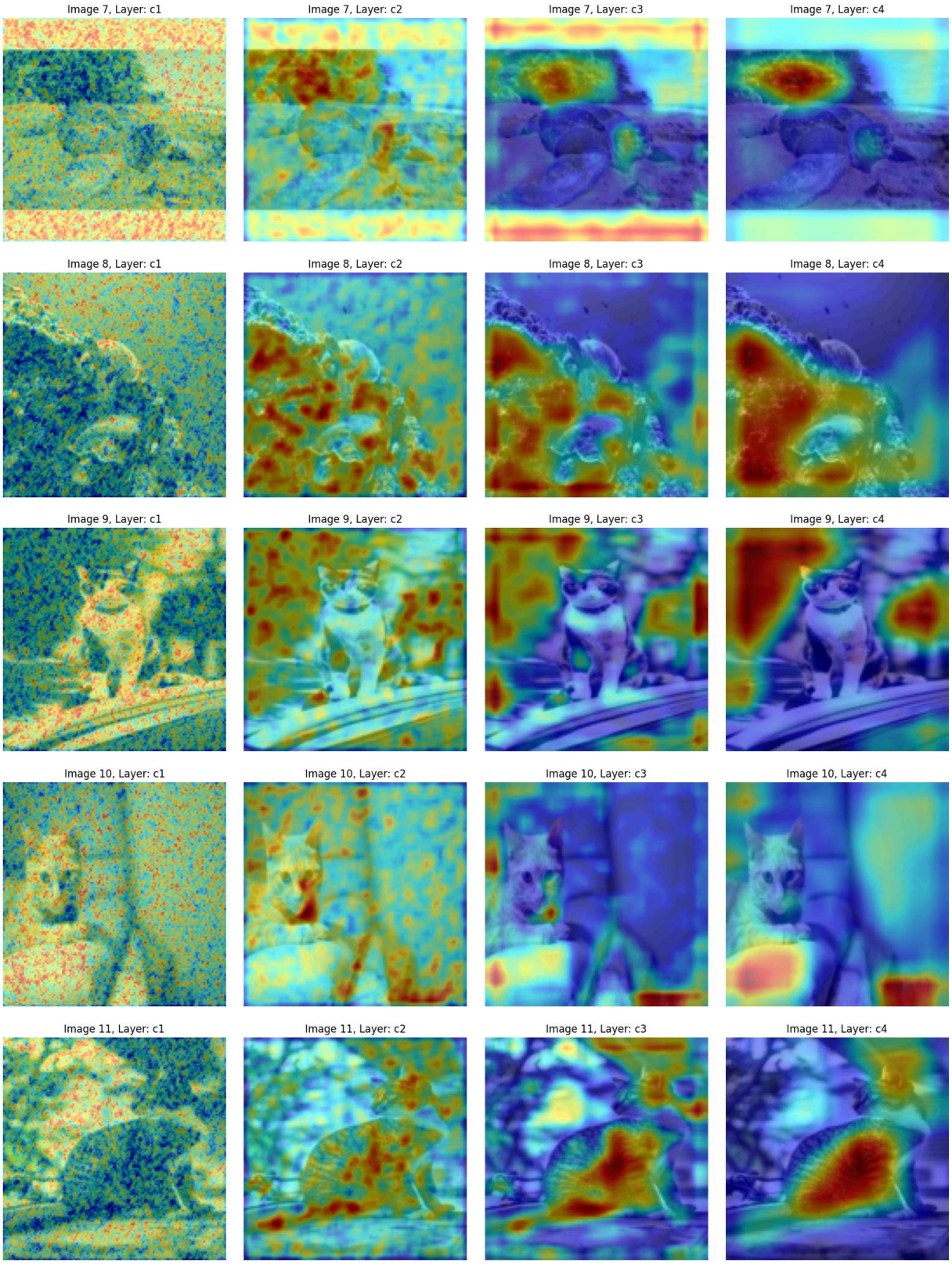


Image 7, Layer: c1
Image 7, Layer: c2
Image 7, Layer: c3
Image 7, Layer: c4
Image 8, Layer: c1
Image 8, Layer: c2
Image 8, Layer: c3
Image 8, Layer: c4
Image 9, Layer: c1
Image 9, Layer: c2
Image 9, Layer: c3
Image 9, Layer: c4
Image 10, Layer: c1
Image 10, Layer: c2
Image 10, Layer: c3
Image 10, Layer: c4
Image 11, Layer: c1
Image 11, Layer: c2
Image 11, Layer: c3
Image 11, Layer: c4

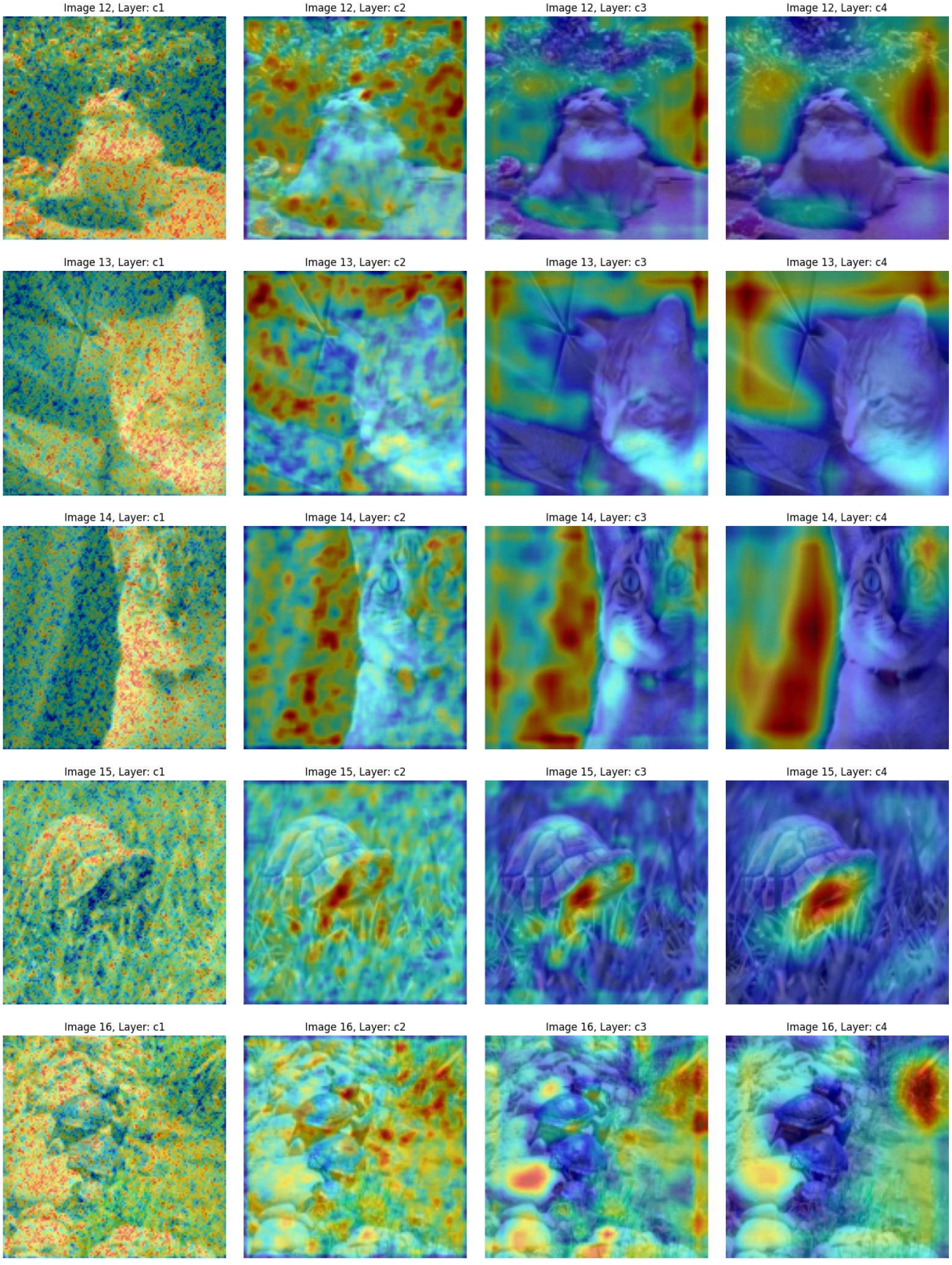


Image 12, Layer: c1
Image 12, Layer: c2
Image 12, Layer: c3
Image 12, Layer: c4
Image 13, Layer: c1
Image 13, Layer: c2
Image 13, Layer: c3
Image 13, Layer: c4
Image 14, Layer: c1
Image 14, Layer: c2
Image 14, Layer: c3
Image 14, Layer: c4
Image 15, Layer: c1
Image 15, Layer: c2
Image 15, Layer: c3
Image 15, Layer: c4
Image 16, Layer: c1
Image 16, Layer: c2
Image 16, Layer: c3
Image 16, Layer: c4

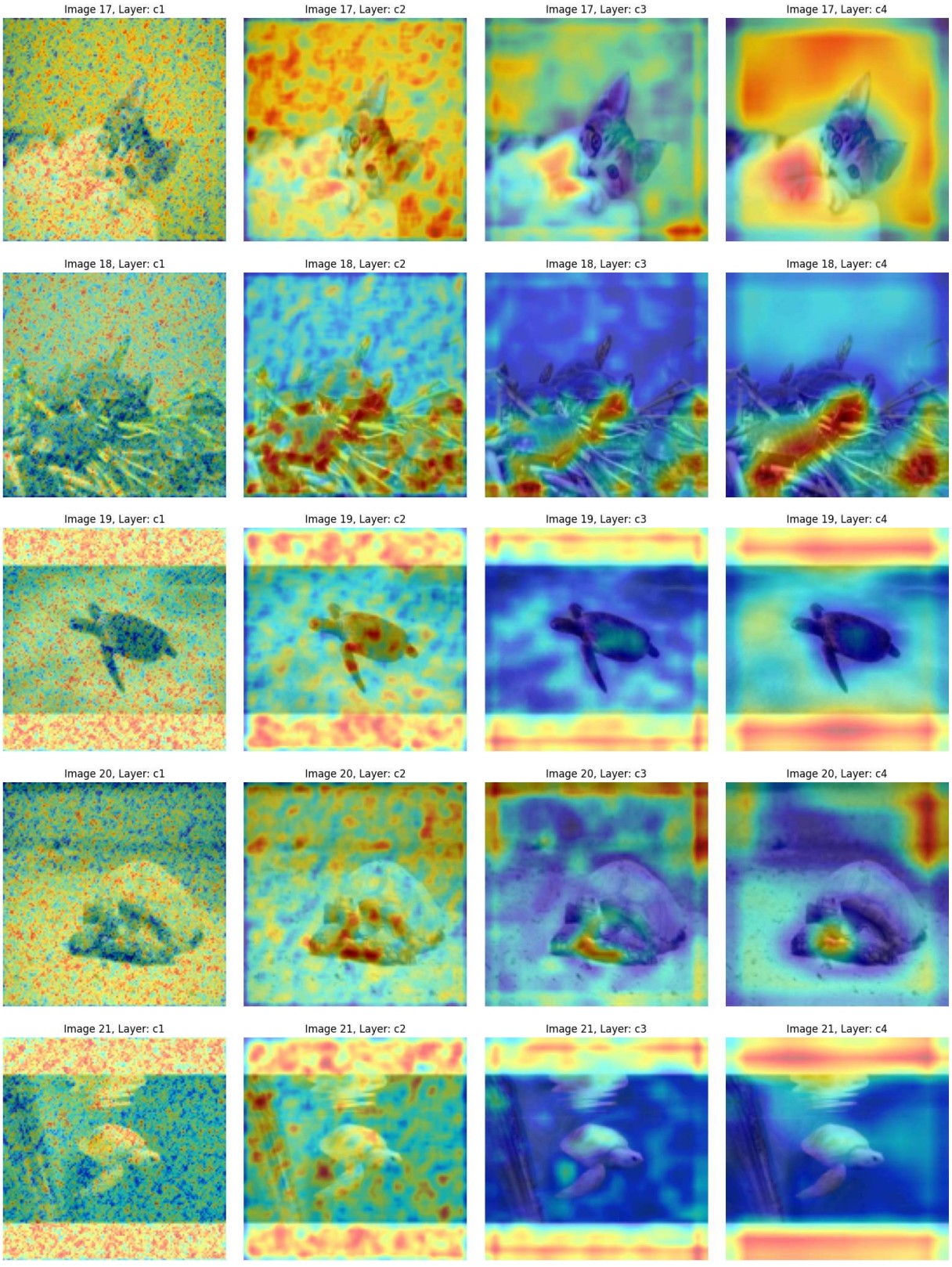


Image 17, Layer: c1
Image 17, Layer: c2
Image 17, Layer: c3
Image 17, Layer: c4
Image 18, Layer: c1
Image 18, Layer: c2
Image 18, Layer: c3
Image 18, Layer: c4
Image 19, Layer: c1
Image 19, Layer: c2
Image 19, Layer: c3
Image 19, Layer: c4
Image 20, Layer: c1
Image 20, Layer: c2
Image 20, Layer: c3
Image 20, Layer: c4
Image 21, Layer: c1
Image 21, Layer: c2
Image 21, Layer: c3
Image 21, Layer: c4

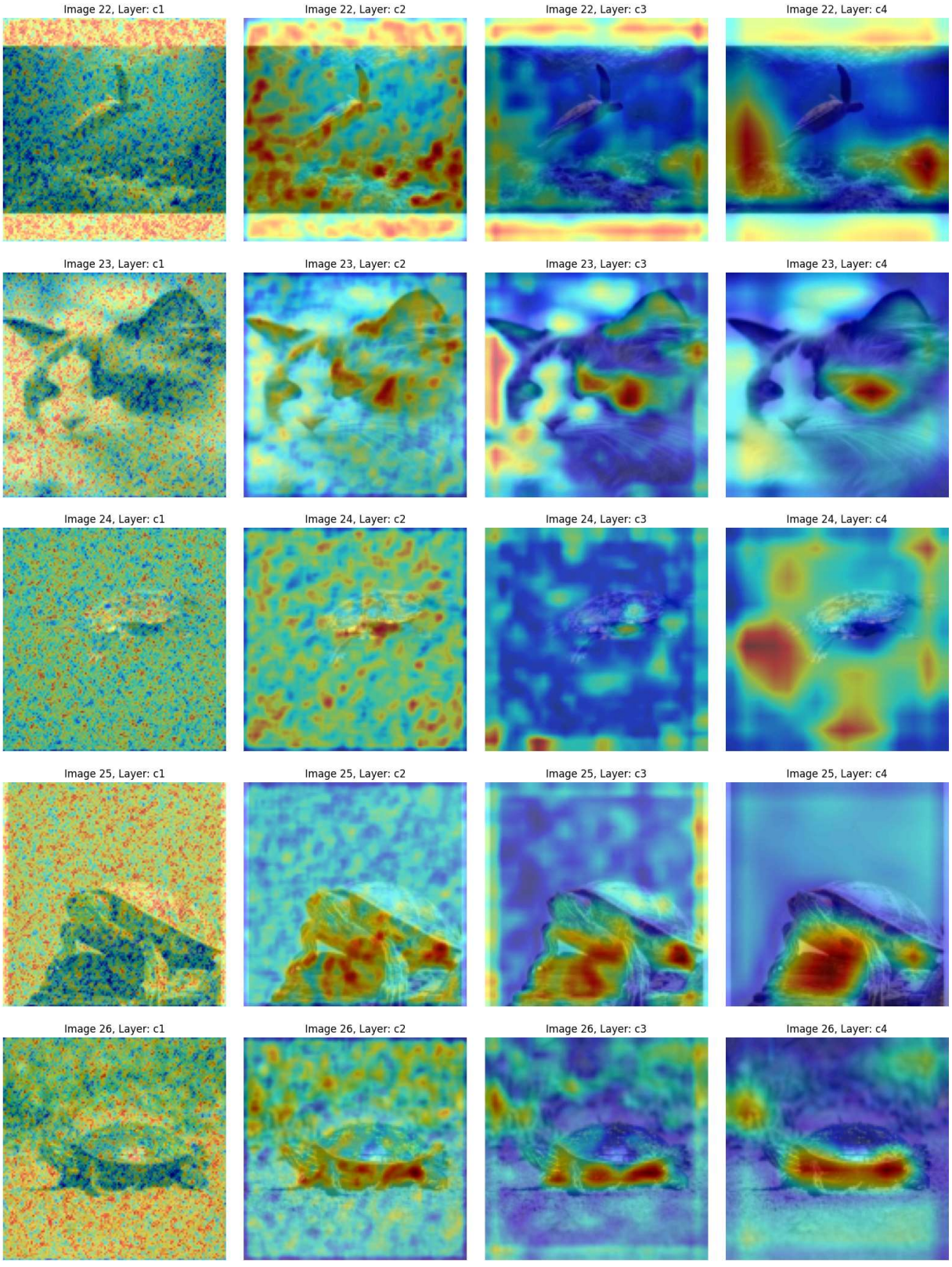


Image 22, Layer: c1
Image 22, Layer: c2
Image 22, Layer: c3
Image 22, Layer: c4
Image 23, Layer: c1
Image 23, Layer: c2
Image 23, Layer: c3
Image 23, Layer: c4
Image 24, Layer: c1
Image 24, Layer: c2
Image 24, Layer: c3
Image 24, Layer: c4
Image 25, Layer: c1
Image 25, Layer: c2
Image 25, Layer: c3
Image 25, Layer: c4
Image 26, Layer: c1
Image 26, Layer: c2
Image 26, Layer: c3
Image 26, Layer: c4

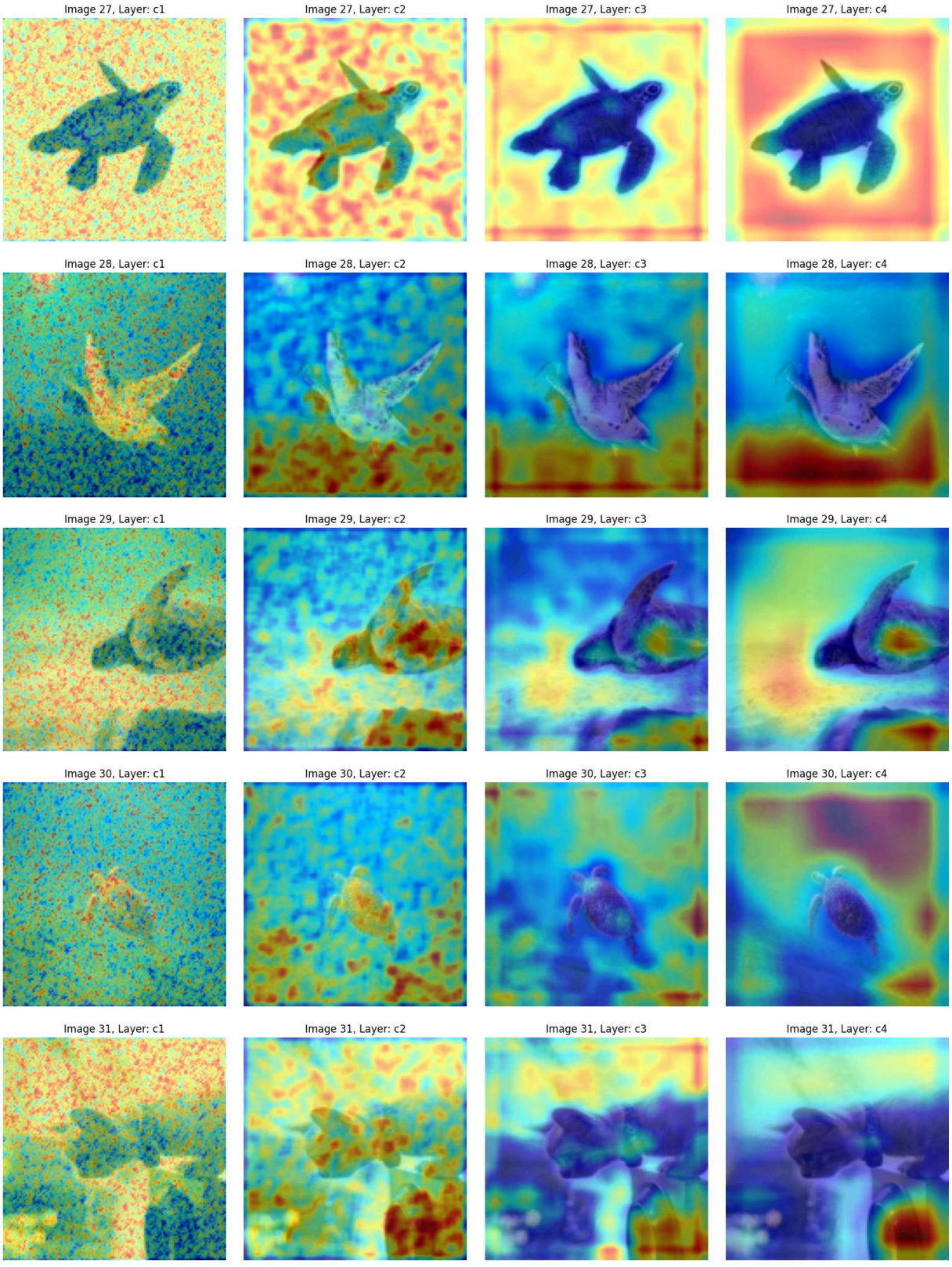
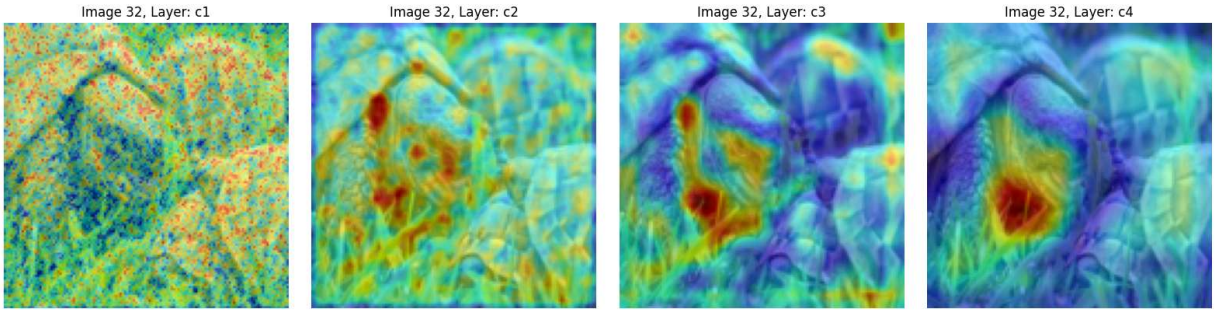
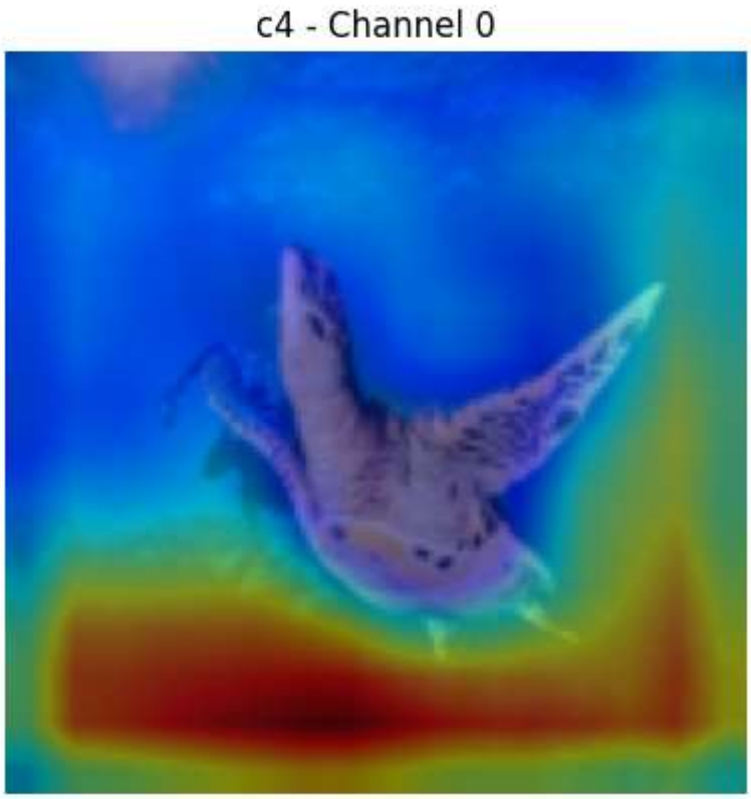


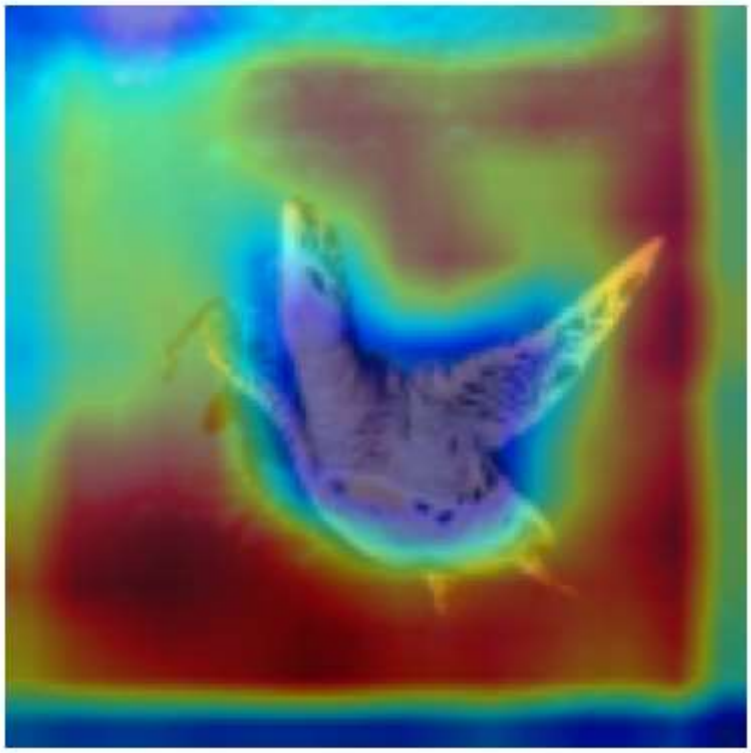
Image 27, Layer: c1
Image 27, Layer: c2
Image 27, Layer: c3
Image 27, Layer: c4
Image 28, Layer: c1
Image 28, Layer: c2
Image 28, Layer: c3
Image 28, Layer: c4
Image 29, Layer: c1
Image 29, Layer: c2
Image 29, Layer: c3
Image 29, Layer: c4
Image 30, Layer: c1
Image 30, Layer: c2
Image 30, Layer: c3
Image 30, Layer: c4
Image 31, Layer: c1
Image 31, Layer: c2
Image 31, Layer: c3
Image 31, Layer: c4



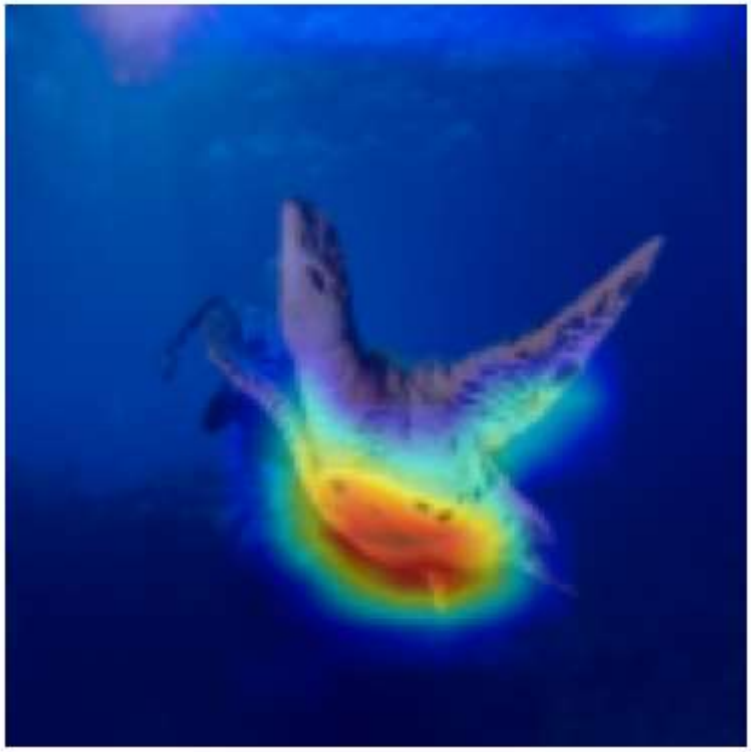
In [23]:
```
# Get feature map from encoder layer
act = features['c4'].squeeze(0).cpu()  # [C, H, W]

for i in range(min(8, act.shape[0])):  # Show first 8 channels
    fmap = act[i]
    overlay = overlay_heatmap(clean_img, fmap)
    plt.imshow(overlay)
    plt.title(f"{layer_name} - Channel {i}")
    plt.axis("off")
    plt.show()
```



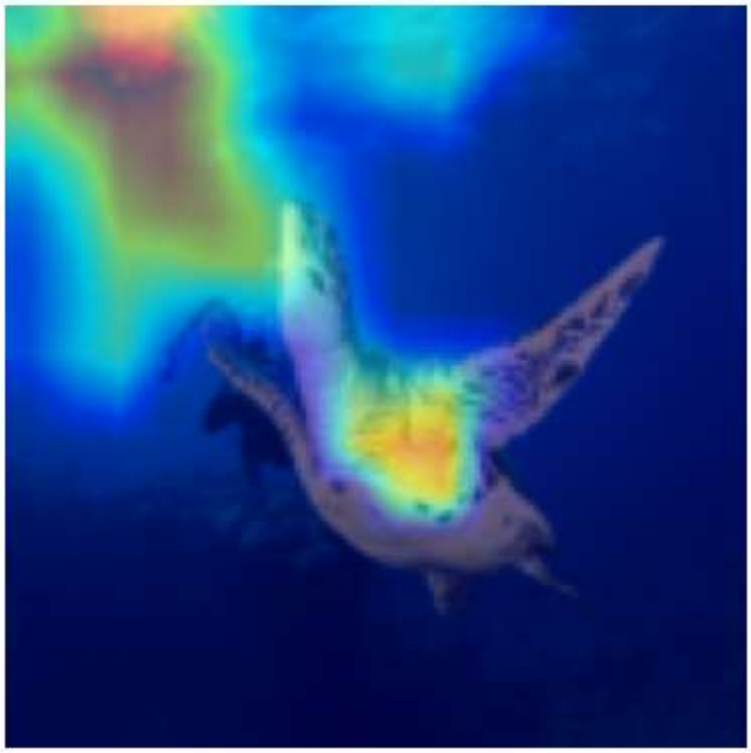
c4 - Channel 1



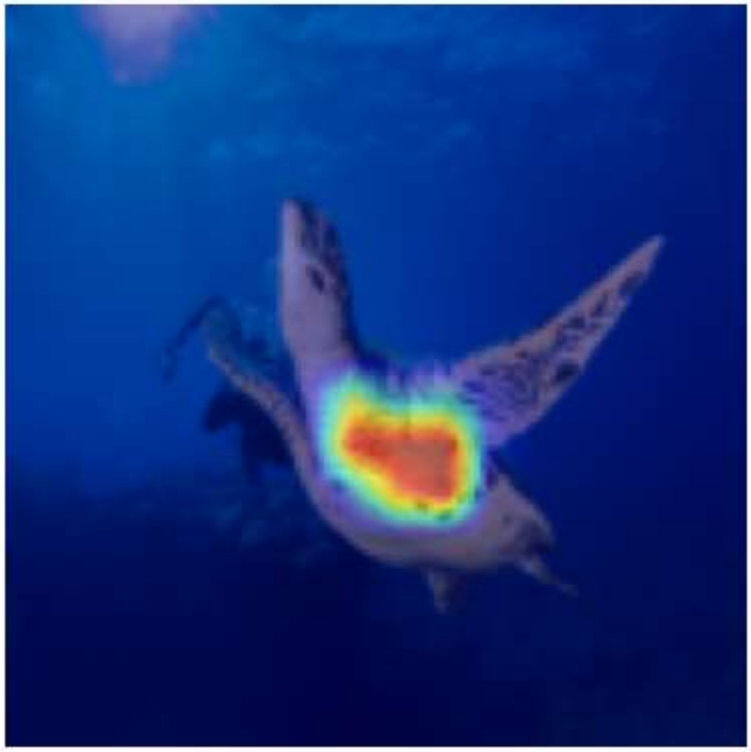
c4 - Channel 2



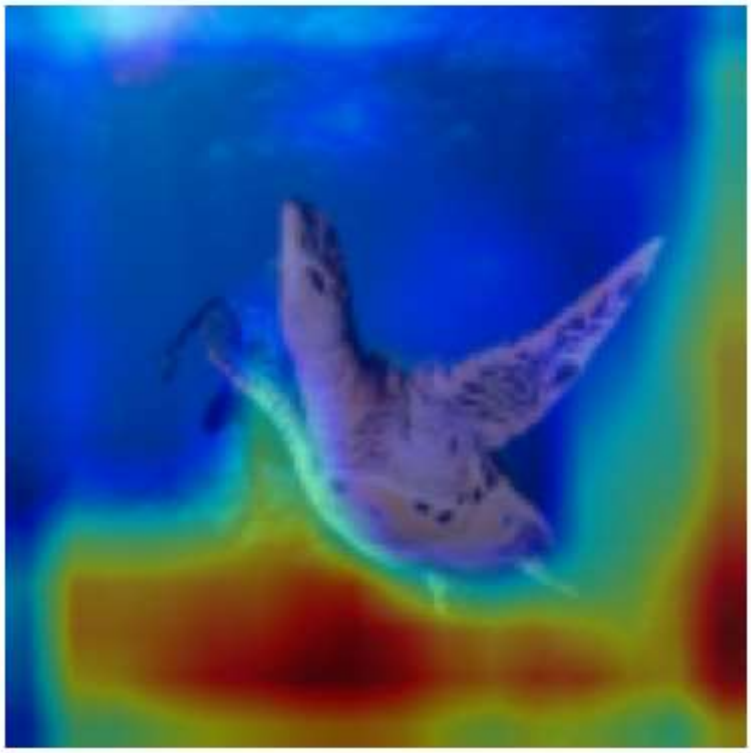
c4 - Channel 3



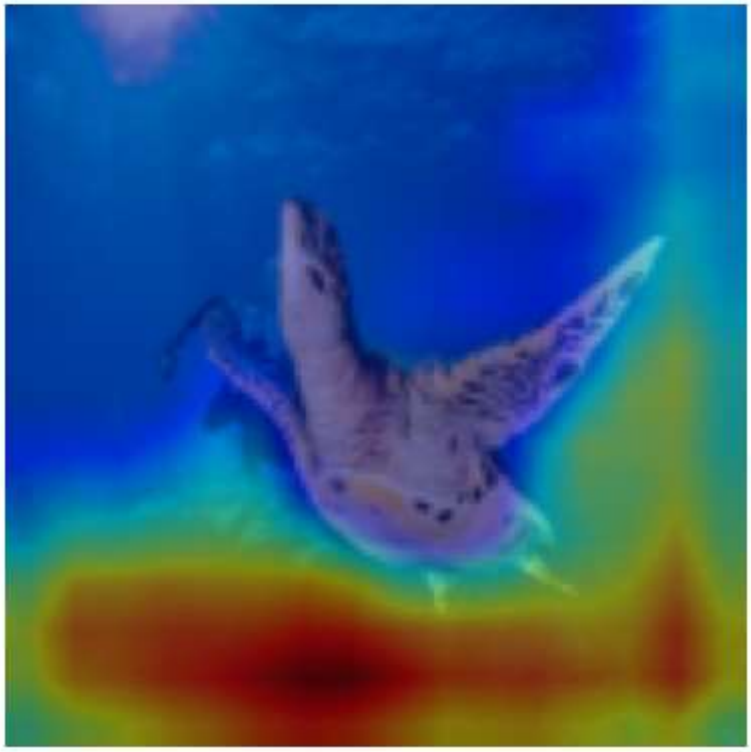
c4 - Channel 4

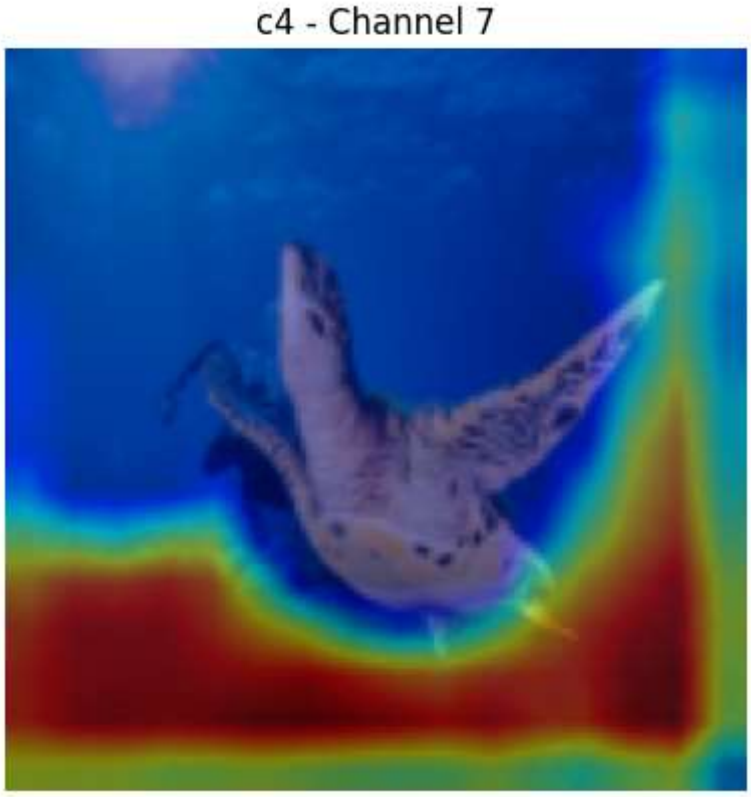


c4 - Channel 5



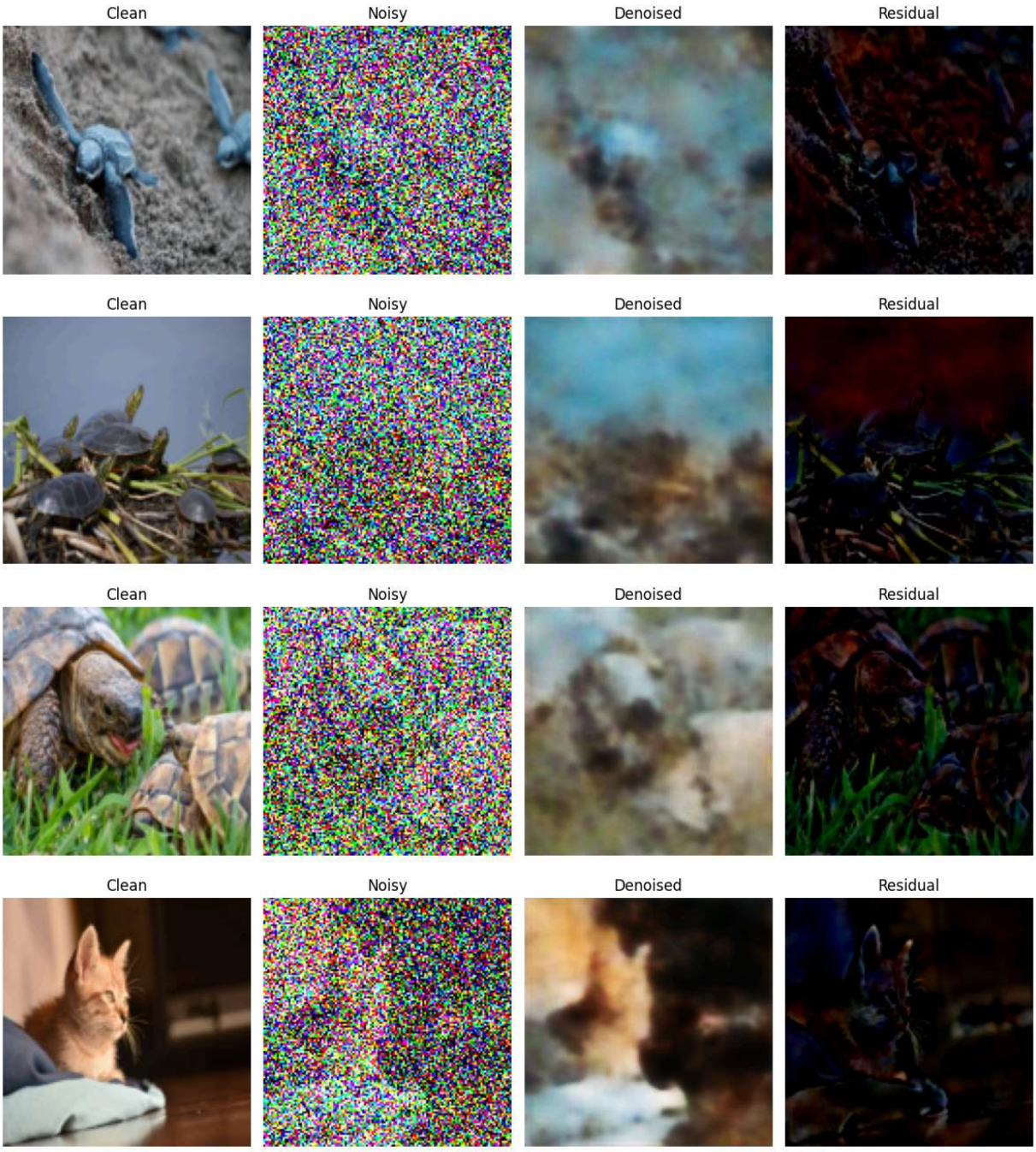
c4 - Channel 6

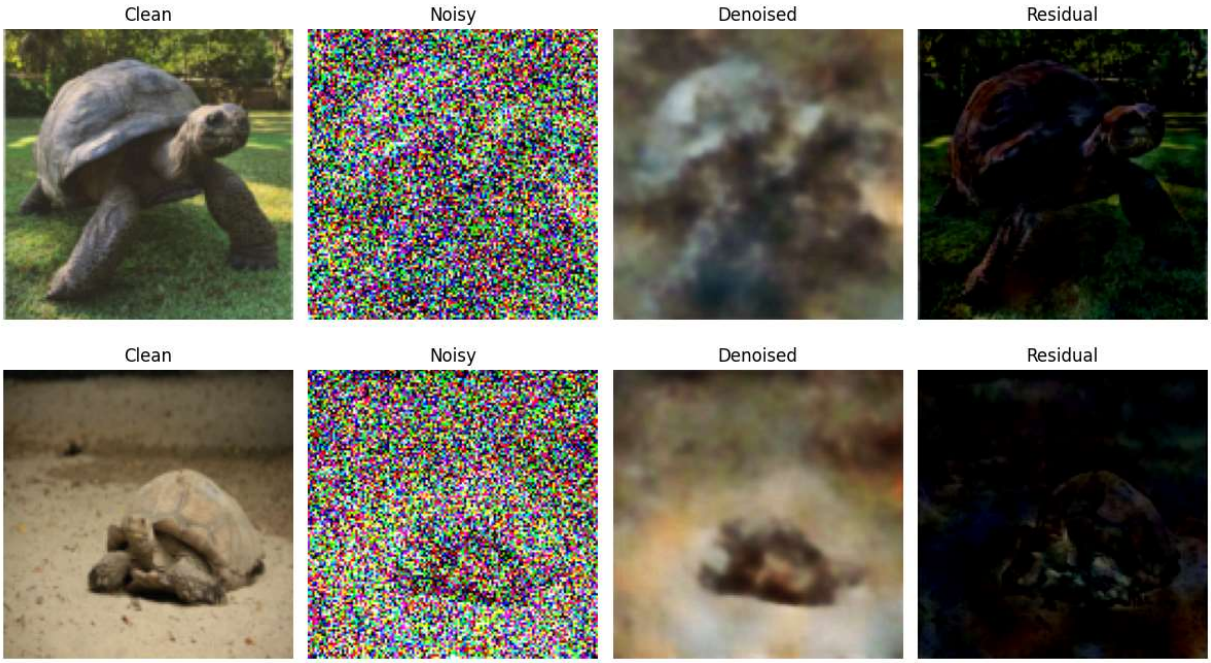




In [69]:
```python
import matplotlib.pyplot as plt
import torch
import torchvision.transforms as T

# Function to denormalize (if you normalized inputs)
def denormalize(tensor):
    return tensor.clamp(0, 1)

# Define device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Evaluate model on a few examples
model.eval()
num_samples = 6  # Number of images to plot

with torch.no_grad():
    for i in range(num_samples):
        noisy_img, clean_img = dataset[i+234]

        # Prepare batch format for model input
        input_tensor = noisy_img.unsqueeze(0).to(device)
        output = model(input_tensor)

        # Convert tensors to CPU and denormalize
        clean = denormalize(clean_img).permute(1, 2, 0).cpu().numpy()
        noisy = denormalize(noisy_img).permute(1, 2, 0).cpu().numpy()
        denoised = denormalize(output.squeeze(0)).permute(1, 2, 0).cpu().numpy()
        residual = (clean - denoised).clip(0, 1)

        # Plot
        fig, axs = plt.subplots(1, 4, figsize=(12, 4))
```

```
        axs[0].imshow(clean)
        axs[0].set_title("Clean")
        axs[1].imshow(noisy)
        axs[1].set_title("Noisy")
        axs[2].imshow(denoised)
        axs[2].set_title("Denoised")
        axs[3].imshow(residual)
        axs[3].set_title("Residual")

        for ax in axs:
            ax.axis("off")
        plt.tight_layout()
        plt.show()
```





Data Loader for Transfer learning

In [42]:
```
device  = torch.device("cuda" if torch.cuda.is_available() else "cpu")

IMG_SIZE = 128

train_transforms = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
])

val_transforms = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
])

train_dataset = datasets.ImageFolder(root = "/content/Dataset/labeled/train", trans
val_dataset = datasets.ImageFolder(root = "/content/Dataset/labeled/val", transform

train_loader = DataLoader(train_dataset, batch_size = 32, shuffle = True, num_worke
val_loader = DataLoader(val_dataset, batch_size = 32, shuffle = False, num_workers

print(f"Train dataset size: {len(train_dataset)}")
print(f"Validation dataset size: {len(val_dataset)}")
```

```
Train dataset size: 154
Validation dataset size: 51
```

Lets build classifier for final model

For that I used UNET encoder and add classification head. And freezed the encoder layers to prevent overwrite the features

In [48]:
```
class Classifier(nn.Module):
  def __init__(self, unet, num_class = 1, freeze_encoder = True):
    super().__init__()

    self.encoder = nn.Sequential(
        unet.c1,
        nn.MaxPool2d(2),
        unet.c2,
        nn.MaxPool2d(2),
        unet.c3,
        nn.MaxPool2d(2),
        unet.c4
    )

    if freeze_encoder:
      for param in self.encoder.parameters():
        param.requires_grad = False

    self.classifier = nn.Sequential(
        nn.Conv2d(512, 256, kernel_size=3, padding = 1),
        nn.ReLU(),
        nn.AdaptiveAvgPool2d((1,1)),
        nn.Flatten(),
        nn.Linear(256, num_class),
        nn.Sigmoid()
    )

  def forward(self,x):
    features = self.encoder(x)
    out = self.classifier(features)

    return out
```

In [49]:
```
unet = DenoisingUnet().to(device)
unet.load_state_dict(torch.load("denoising_unet.pth", map_location=torch.device('cp

model = Classifier(unet, num_class = 1, freeze_encoder=True)
model.to(device)
```

Out[49]:
```
Classifier(
  (encoder): Sequential(
    (0): Sequential(
      (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU(inplace=True)
      (2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_sta
ts=True)
      (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): ReLU(inplace=True)
      (5): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_sta
ts=True)
    )
    (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fals
e)
    (2): Sequential(
      (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU(inplace=True)
      (2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_st
ats=True)
      (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): ReLU(inplace=True)
      (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_st
ats=True)
    )
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fals
e)
    (4): Sequential(
      (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU(inplace=True)
      (2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_st
ats=True)
      (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): ReLU(inplace=True)
      (5): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_st
ats=True)
    )
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fals
e)
    (6): Sequential(
      (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU(inplace=True)
      (2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_st
ats=True)
      (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): ReLU(inplace=True)
      (5): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_st
ats=True)
    )
  )
  (classifier): Sequential(
    (0): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): AdaptiveAvgPool2d(output_size=(1, 1))
    (3): Flatten(start_dim=1, end_dim=-1)
    (4): Linear(in_features=256, out_features=1, bias=True)
    (5): Sigmoid()
```

```
  )
)
```

finetune the model

In [52]:

```
criterion = nn.BCELoss() #use binary cross entrophy loss
optimizer = optim.Adam(filter(lambda p: p.requires_grad, model.parameters()), lr =

epochs = 5
for epoch in range(epochs):
  model.train()
  running_loss = 0
  for imgs, labels in train_loader:
    imgs, labels = imgs.to(device), labels.float().to(device).unsqueeze(1)

    optimizer.zero_grad()
    outputs = model(imgs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()

    running_loss += loss.item()

    print(f"loss {loss}")

  print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader)}")
```

```
loss 0.6669149398803711
loss 0.644802451133728
loss 0.6502231359481812
loss 0.6004418730735779
loss 0.7057055234909058
Epoch 1/5, Loss: 0.6536175847053528
loss 0.5609515309333801
loss 0.548151433467865
loss 0.6030179858207703
loss 0.5822842121124268
loss 0.6978352665901184
Epoch 2/5, Loss: 0.5984480857849122
loss 0.5411409139633179
loss 0.6435787677764893
loss 0.5517104268074036
loss 0.6176486015319824
loss 0.5532307624816895
Epoch 3/5, Loss: 0.5814618945121766
loss 0.5860187411308289
loss 0.5617815852165222
loss 0.5187263488769531
loss 0.6062049865722656
loss 0.5526189208030701
Epoch 4/5, Loss: 0.565070116519928
loss 0.5440882444381714
loss 0.7163848876953125
loss 0.5676494836807251
loss 0.47416168451309204
loss 0.4919039309024811
Epoch 5/5, Loss: 0.5588376462459564
```

In [65]:
```python
model.eval()
correct, total = 0, 0
with torch.no_grad():
    for imgs, labels in val_loader:
        imgs, labels = imgs.to(device), labels.float().to(device).unsqueeze(1)
        outputs = model(imgs)
        predicted = (outputs > 0.5).float()
        correct += (predicted == labels).sum().item()
        total += labels.size(0)

print(f"Validation Accuracy: {100 * correct / total:.2f}%")
```

```
Validation Accuracy: 62.75%
```

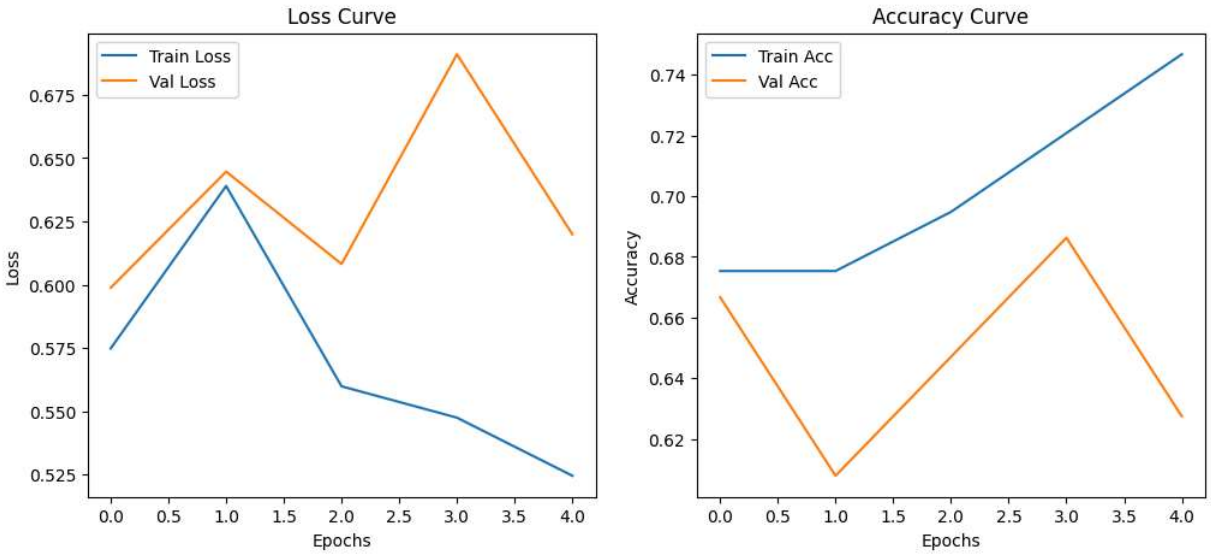
Modified version for plot Loss and accuracy of the model

In [61]:
```python
import matplotlib.pyplot as plt

criterion = nn.BCELoss()
optimizer = optim.Adam(filter(lambda p: p.requires_grad, model.parameters()), lr=1e

epochs = 5
train_losses = []
```

```
val_losses = []
train_accuracies = []
val_accuracies = []

for epoch in range(epochs):
    # ---------- Training ----------
    model.train()
    running_loss = 0
    correct = 0
    total = 0

    for imgs, labels in train_loader:
        imgs, labels = imgs.to(device), labels.float().to(device).unsqueeze(1)

        optimizer.zero_grad()
        outputs = model(imgs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        # Accuracy
        preds = (outputs > 0.5).float()
        correct += (preds == labels).sum().item()
        total += labels.size(0)

    train_loss = running_loss / len(train_loader)
    train_acc = correct / total
    train_losses.append(train_loss)
    train_accuracies.append(train_acc)

    # ---------- Validation ----------
    model.eval()
    val_running_loss = 0
    val_correct = 0
    val_total = 0

    with torch.no_grad():
        for imgs, labels in val_loader:
            imgs, labels = imgs.to(device), labels.float().to(device).unsqueeze(1)
            outputs = model(imgs)
            loss = criterion(outputs, labels)
            val_running_loss += loss.item()

            preds = (outputs > 0.5).float()
            val_correct += (preds == labels).sum().item()
            val_total += labels.size(0)

    val_loss = val_running_loss / len(val_loader)
    val_acc = val_correct / val_total
    val_losses.append(val_loss)
    val_accuracies.append(val_acc)

    print(f"Epoch {epoch+1}/{epochs} | "
          f"Train Loss: {train_loss:.4f} Acc: {train_acc:.4f} | "
```

```
          f"Val Loss: {val_loss:.4f} Acc: {val_acc:.4f}")

# ---------- Plot ----------
plt.figure(figsize=(12,5))

# Loss plot
plt.subplot(1,2,1)
plt.plot(train_losses, label='Train Loss')
plt.plot(val_losses, label='Val Loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.title("Loss Curve")

# Accuracy plot
plt.subplot(1,2,2)
plt.plot(train_accuracies, label='Train Acc')
plt.plot(val_accuracies, label='Val Acc')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.title("Accuracy Curve")

plt.show()
```

```
Epoch 1/5 | Train Loss: 0.5749 Acc: 0.6753 | Val Loss: 0.5988 Acc: 0.6667
Epoch 2/5 | Train Loss: 0.6390 Acc: 0.6753 | Val Loss: 0.6447 Acc: 0.6078
Epoch 3/5 | Train Loss: 0.5599 Acc: 0.6948 | Val Loss: 0.6082 Acc: 0.6471
Epoch 4/5 | Train Loss: 0.5475 Acc: 0.7208 | Val Loss: 0.6910 Acc: 0.6863
Epoch 5/5 | Train Loss: 0.5246 Acc: 0.7468 | Val Loss: 0.6199 Acc: 0.6275
```



Test with test dataset

In [58]:
```
# Test transforms (same as validation)
test_transforms = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
```

```
])

# Test dataset
test_dataset = datasets.ImageFolder(
    root="/content/Dataset/labeled/test",
    transform=test_transforms
)

# Test loader
test_loader = DataLoader(
    test_dataset,
    batch_size=32,
    shuffle=True,
    num_workers=2,
    pin_memory=True
)

print(f"Test dataset size: {len(test_dataset)}")
```

```
Test dataset size: 53
```

In [64]:

```
def denormalize(img):
    """Undo normalization for mean=0.5, std=0.5"""
    img = img * 0.5 + 0.5
    return torch.clamp(img, 0, 1)

def plot_test_predictions(model, test_loader, class_names=None, num_images=8):
    model.eval()
    device = next(model.parameters()).device
    images_shown = 0

    plt.figure(figsize=(12, 6))

    with torch.no_grad():
        for imgs, labels in test_loader:
            imgs, labels = imgs.to(device), labels.float().to(device)
            outputs = model(imgs)
            preds = (outputs > 0.5).float().cpu().numpy()

            for i in range(imgs.size(0)):
                if images_shown >= num_images:
                    plt.show()
                    return

                # ✅ Denormalize before plotting
                img_dn = denormalize(imgs[i].cpu())
                img_np = img_dn.permute(1, 2, 0).numpy()

                plt.subplot(2, num_images//2, images_shown + 1)
                plt.imshow((img_np * 255).astype("uint8"))
                plt.axis("off")

                pred_label = int(preds[i])
                true_label = int(labels[i].cpu().item())

                if class_names:
```

```
                    pred_label = class_names[pred_label]
                    true_label = class_names[true_label]

                plt.title(f"P: {pred_label}\nT: {true_label}", fontsize=10)
                images_shown += 1

class_names = ["Cat", "Turtle"]
plot_test_predictions(model, test_loader, class_names)
```

```
/tmp/ipython-input-2851037440.py:32: DeprecationWarning: Conversion of an array with
ndim > 0 to a scalar is deprecated, and will error in future. Ensure you extract a s
ingle element from your array before performing this operation. (Deprecated NumPy 1.
25.)
  pred_label = int(preds[i])
```