```
!ls
```

```
sample_data
```

```
# ============================================================
# CSET419 - Lab 5: Baseline CNN for Image-to-Image Translation
# Production-Level Implementation with Google Drive Integration
# ============================================================

# ╔══════════════════════════════════════════════════════════╗
# ║  CELL 1: Mount Google Drive & Setup Environment          ║
# ╚══════════════════════════════════════════════════════════╝

from google.colab import drive
drive.mount('/content/drive')

import os
import sys
import json
import time
import random
import numpy as np
from datetime import datetime
from typing import Dict, List, Tuple, Optional
import warnings
warnings.filterwarnings('ignore')

# Create project directory structure in Drive
PROJECT_NAME = "CSET419_Lab5_EncoderDecoder"
BASE_PATH = f"/content/drive/MyDrive/{PROJECT_NAME}"
CHECKPOINT_DIR = f"{BASE_PATH}/checkpoints"
RESULTS_DIR = f"{BASE_PATH}/results"
LOGS_DIR = f"{BASE_PATH}/logs"
CONFIG_PATH = f"{BASE_PATH}/config.json"

# Create directories
```

```
for dir_path in [BASE_PATH, CHECKPOINT_DIR, RESULTS_DIR, LOGS_DIR]:
    os.makedirs(dir_path, exist_ok=True)

print(f"✓ Project mounted at: {BASE_PATH}")
print(f"✓ Checkpoints: {CHECKPOINT_DIR}")
print(f"✓ Results: {RESULTS_DIR}")
print(f"✓ Logs: {LOGS_DIR}")

# Set random seeds for reproducibility
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
```

```
Mounted at /content/drive
✓ Project mounted at: /content/drive/MyDrive/CSET419_Lab5_EncoderDecoder
✓ Checkpoints: /content/drive/MyDrive/CSET419_Lab5_EncoderDecoder/checkpoints
✓ Results: /content/drive/MyDrive/CSET419_Lab5_EncoderDecoder/results
✓ Logs: /content/drive/MyDrive/CSET419_Lab5_EncoderDecoder/logs
```

```
#importing libraries

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
from torch.utils.tensorboard import SummaryWriter
import torchvision
import torchvision.transforms as transforms
from torchvision.utils import make_grid, save_image

# Verify GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"✓ Using device: {device}")
if torch.cuda.is_available():
    print(f"✓ GPU: {torch.cuda.get_device_name(0)}")
    print(f"✓ Memory: {torch.cuda.get_device_properties(0).total_memory / 1e9:.2f} GB")
```

```
# Set PyTorch seeds
torch.manual_seed(SEED)
if torch.cuda.is_available():
    torch.cuda.manual_seed(SEED)
    torch.cuda.manual_seed_all(SEED)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = True  # Enable for faster training
```

```
✓ Using device: cuda
✓ GPU: Tesla T4
✓ Memory: 15.64 GB
```

```
# ╔══════════════════════════════════════════════════════════╗
# ║  CELL 3: Hyperparameter Configuration (PRO LEVEL)        ║
# ╚══════════════════════════════════════════════════════════╝

class HyperParams:
    """Professional hyperparameter configuration with validation"""

    def __init__(self):
        # Data parameters
        self.dataset = 'CIFAR10'
        self.batch_size = 128
        self.num_workers = 4
        self.pin_memory = True

        # Image parameters
        self.image_size = 32
        self.channels = 3
        self.normalize_range = (-1, 1)  # Normalize to [-1, 1] as required

        # Model architecture
        self.encoder_blocks = [64, 128, 256, 512]  # Progressive feature extraction
        self.bottleneck_dim = 512
        self.use_batch_norm = True
        self.use_dropout = True
        self.dropout_rate = 0.2
```

```
        # Training parameters
        self.epochs = 100
        self.learning_rate = 2e-4
        self.weight_decay = 1e-5
        self.scheduler_type = 'cosine'  # 'step', 'cosine', 'plateau'
        self.warmup_epochs = 5

        # Loss configuration
        self.loss_type = 'combined'  # 'mse', 'l1', 'combined'
        self.mse_weight = 0.5
        self.l1_weight = 0.5
        self.ssim_weight = 0.0  # Optional: add SSIM loss for better quality

        # Optimization
        self.optimizer = 'adamw'  # 'adam', 'adamw', 'sgd'
        self.beta1 = 0.9
        self.beta2 = 0.999
        self.gradient_clip = 1.0

        # Checkpointing
        self.save_freq = 10  # Save every N epochs
        self.keep_best = 3   # Keep top N best checkpoints

        # Logging
        self.log_freq = 50   # Log every N batches
        self.sample_freq = 5  # Generate samples every N epochs

        # Augmentation (for robustness)
        self.use_augmentation = False  # Keep False for pure reconstruction

    def to_dict(self) -> Dict:
        return {k: v for k, v in self.__dict__.items()}

    def save(self, path: str):
        with open(path, 'w') as f:
            json.dump(self.to_dict(), f, indent=4)
```

```
    @classmethod
    def load(cls, path: str):
        with open(path, 'r') as f:
            config = json.load(f)
        hparams = cls()
        for k, v in config.items():
            setattr(hparams, k, v)
        return hparams

# Initialize hyperparameters
hparams = HyperParams()
hparams.save(CONFIG_PATH)
print("✓ Hyperparameters configured and saved")
print(json.dumps(hparams.to_dict(), indent=2))
```

```
✓ Hyperparameters configured and saved
{
  "dataset": "CIFAR10",
  "batch_size": 128,
  "num_workers": 4,
  "pin_memory": true,
  "image_size": 32,
  "channels": 3,
  "normalize_range": [
    -1,
    1
  ],
  "encoder_blocks": [
    64,
    128,
    256,
    512
  ],
  "bottleneck_dim": 512,
  "use_batch_norm": true,
  "use_dropout": true,
  "dropout_rate": 0.2,
  "epochs": 100,
  "learning_rate": 0.0002,
  "weight_decay": 1e-05,
```

```
  "scheduler_type": "cosine",
  "warmup_epochs": 5,
  "loss_type": "combined",
  "mse_weight": 0.5,
  "l1_weight": 0.5,
  "ssim_weight": 0.0,
  "optimizer": "adamw",
  "beta1": 0.9,
  "beta2": 0.999,
  "gradient_clip": 1.0,
  "save_freq": 10,
  "keep_best": 3,
  "log_freq": 50,
  "sample_freq": 5,
  "use_augmentation": false
}
```

```
# ╔══════════════════════════════════════════════════════════╗
# ║  CELL 4: Data Loading & Preprocessing (CIFAR10 Paired)   ║
# ╚══════════════════════════════════════════════════════════╝

class PairedCIFAR10Dataset(Dataset):
    """
    Creates paired images for image-to-image translation.
    For this lab, we use: Input = Slightly corrupted image, Target = Original
    This creates a meaningful translation task while using CIFAR10.
    """

    def __init__(self, root: str, train: bool = True, transform=None, noise_factor: float = 0.1):
        self.cifar10 = torchvision.datasets.CIFAR10(
            root=root,
            train=train,
            download=True,
            transform=None  # We'll handle transforms manually
        )
        self.transform = transform
        self.noise_factor = noise_factor
        self.train = train
```

```
    def __len__(self):
        return len(self.cifar10)

    def __getitem__(self, idx):
        img, label = self.cifar10[idx]

        # Convert to tensor [0, 1]
        if self.transform:
            img_tensor = self.transform(img)
        else:
            img_tensor = transforms.ToTensor()(img)

        # Create input by adding noise (denoising task)
        # This is a valid image-to-image translation task
        noise = torch.randn_like(img_tensor) * self.noise_factor
        input_img = torch.clamp(img_tensor + noise, 0, 1)

        # Normalize both to [-1, 1] as required
        normalize = transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
        input_img = normalize(input_img)
        target_img = normalize(img_tensor)

        return {
            'input': input_img,
            'target': target_img,
            'label': label
        }

# Transforms
train_transform = transforms.Compose([
    transforms.ToTensor(),  # Converts [0,255] to [0,1]
])

test_transform = transforms.Compose([
    transforms.ToTensor(),
])

# Create datasets
```

```
    train_dataset = PairedCIFAR10Dataset(
        root='./data',
        train=True,
        transform=train_transform,
        noise_factor=0.1
    )

    test_dataset = PairedCIFAR10Dataset(
        root='./data',
        train=False,
        transform=test_transform,
        noise_factor=0.1
    )

    # DataLoaders with optimized settings
    train_loader = DataLoader(
        train_dataset,
        batch_size=hparams.batch_size,
        shuffle=True,
        num_workers=hparams.num_workers,
        pin_memory=hparams.pin_memory,
        drop_last=True,
        persistent_workers=True if hparams.num_workers > 0 else False
    )

    test_loader = DataLoader(
        test_dataset,
        batch_size=hparams.batch_size,
        shuffle=False,
        num_workers=hparams.num_workers,
        pin_memory=hparams.pin_memory,
        persistent_workers=True if hparams.num_workers > 0 else False
    )

    print(f"✓ Train samples: {len(train_dataset)}")
    print(f"✓ Test samples: {len(test_dataset)}")
    print(f"✓ Batches per epoch: {len(train_loader)}")
```

```
# Visualize sample
sample = next(iter(train_loader))
print(f"✓ Input range: [{sample['input'].min():.2f}, {sample['input'].max():.2f}]")
print(f"✓ Target range: [{sample['target'].min():.2f}, {sample['target'].max():.2f}]")
```

```
100%|██████████| 170M/170M [00:04<00:00, 39.0MB/s]
✓ Train samples: 50000
✓ Test samples: 10000
✓ Batches per epoch: 390
✓ Input range: [-1.00, 1.00]
✓ Target range: [-1.00, 1.00]
```

```
# ╔══════════════════════════════════════════════════════════╗
# ║  CELL 5: Encoder-Decoder Architecture (Professional)     ║
# ╚══════════════════════════════════════════════════════════╝

class ConvBlock(nn.Module):
    """Professional Conv Block with BN, Activation, and Dropout"""

    def __init__(self, in_ch: int, out_ch: int, downsample: bool = True,
                 use_bn: bool = True, use_dropout: bool = False, dropout_rate: float = 0.2):
        super().__init__()

        layers = []

        # Convolution
        if downsample:
            layers.append(nn.Conv2d(in_ch, out_ch, 4, stride=2, padding=1, bias=False))
        else:
            layers.append(nn.ConvTranspose2d(in_ch, out_ch, 4, stride=2, padding=1, bias=False))

        # Batch Normalization
        if use_bn:
            layers.append(nn.BatchNorm2d(out_ch))

        # Activation
        layers.append(nn.LeakyReLU(0.2, inplace=True) if downsample else nn.ReLU(inplace=True))
```

```
            # Dropout
            if use_dropout and downsample:
                layers.append(nn.Dropout2d(dropout_rate))

            self.block = nn.Sequential(*layers)

            # Skip connection if dimensions match
            self.skip = (in_ch == out_ch) and not downsample

        def forward(self, x):
            return self.block(x)

    class ResidualBlock(nn.Module):
        """Residual Block for better gradient flow"""

        def __init__(self, channels: int, use_bn: bool = True):
            super().__init__()
            self.conv1 = nn.Conv2d(channels, channels, 3, padding=1, bias=False)
            self.bn1 = nn.BatchNorm2d(channels) if use_bn else nn.Identity()
            self.conv2 = nn.Conv2d(channels, channels, 3, padding=1, bias=False)
            self.bn2 = nn.BatchNorm2d(channels) if use_bn else nn.Identity()
            self.relu = nn.ReLU(inplace=True)

        def forward(self, x):
            residual = x
            out = self.relu(self.bn1(self.conv1(x)))
            out = self.bn2(self.conv2(out))
            out += residual
            return self.relu(out)

    class EncoderDecoderCNN(nn.Module):
        """
        Professional Encoder-Decoder CNN for Image-to-Image Translation
        Architecture: Input -> Encoder -> Bottleneck -> Decoder -> Output
        """

        def __init__(self, hparams: HyperParams):
```

```
        super().__init__()

        self.hparams = hparams
        channels = hparams.channels
        blocks = hparams.encoder_blocks

        # ========== ENCODER ==========
        self.encoder = nn.ModuleList()
        in_ch = channels

        for i, out_ch in enumerate(blocks):
            self.encoder.append(
                ConvBlock(
                    in_ch, out_ch,
                    downsample=True,
                    use_bn=hparams.use_batch_norm,
                    use_dropout=hparams.use_dropout and i < 2,  # Dropout only in early layers
                    dropout_rate=hparams.dropout_rate
                )
            )
            in_ch = out_ch

        # Bottleneck with residual blocks for better representation
        self.bottleneck = nn.Sequential(
            ResidualBlock(blocks[-1], hparams.use_batch_norm),
            ResidualBlock(blocks[-1], hparams.use_batch_norm),
            ResidualBlock(blocks[-1], hparams.use_batch_norm),
        )

        # ========== DECODER ==========
        self.decoder = nn.ModuleList()
        reversed_blocks = list(reversed(blocks))

        for i in range(len(reversed_blocks) - 1):
            in_ch = reversed_blocks[i]
            out_ch = reversed_blocks[i + 1]

            self.decoder.append(
```

```
                ConvBlock(
                    in_ch, out_ch,
                    downsample=False,
                    use_bn=hparams.use_batch_norm,
                    use_dropout=False
                )
            )

        # Final output layer (no batch norm, tanh activation for [-1, 1] range)
        self.final = nn.Sequential(
            nn.ConvTranspose2d(reversed_blocks[-1], channels, 4, stride=2, padding=1),
            nn.Tanh()  # Output in [-1, 1]
        )

        self._initialize_weights()

    def _initialize_weights(self):
        """He initialization for better convergence"""
        for m in self.modules():
            if isinstance(m, (nn.Conv2d, nn.ConvTranspose2d)):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

    def forward(self, x):
        # Encoder
        skips = []
        for enc in self.encoder:
            x = enc(x)
            skips.append(x)

        # Bottleneck
        x = self.bottleneck(x)

        # Decoder with skip connections (U-Net style)
```

```
        for i, dec in enumerate(self.decoder):
            x = dec(x)
            # Add skip connection from encoder
            if i < len(skips) - 1:
                skip = skips[-(i+2)]
                if x.shape == skip.shape:
                    x = x + skip  # Residual connection

        # Final output
        x = self.final(x)
        return x

    def get_param_count(self):
        return sum(p.numel() for p in self.parameters() if p.requires_grad)
```

```
# Initialize model
model = EncoderDecoderCNN(hparams).to(device)
print(f"✓ Model initialized")
print(f"✓ Parameters: {model.get_param_count():,}")
print(model)

# ╔════════════════════════════════════════════════════════════╗
# ║  CELL 6: Loss Functions & Metrics (Professional)           ║
# ╚════════════════════════════════════════════════════════════╝

class CombinedLoss(nn.Module):
    """Combined MSE + L1 Loss with optional SSIM"""

    def __init__(self, hparams: HyperParams):
        super().__init__()
        self.hparams = hparams
        self.mse = nn.MSELoss()
        self.l1 = nn.L1Loss()

    def forward(self, pred, target):
        loss = 0
```

```
        if self.hparams.loss_type in ['mse', 'combined']:
            loss += self.hparams.mse_weight * self.mse(pred, target)

        if self.hparams.loss_type in ['l1', 'combined']:
            loss += self.hparams.l1_weight * self.l1(pred, target)

        return loss

class Metrics:
    """Track and compute metrics"""

    @staticmethod
    def psnr(pred, target, max_val=2.0):  # max_val=2 because range is [-1, 1]
        """Peak Signal-to-Noise Ratio"""
        mse = torch.mean((pred - target) ** 2)
        if mse == 0:
            return float('inf')
        return 20 * torch.log10(max_val / torch.sqrt(mse))

    @staticmethod
    def ssim(pred, target, window_size=11):
        """Structural Similarity Index (simplified)"""
        # Simplified SSIM calculation
        mu1 = torch.mean(pred)
        mu2 = torch.mean(target)
        sigma1 = torch.std(pred)
        sigma2 = torch.std(target)
        sigma12 = torch.mean((pred - mu1) * (target - mu2))

        c1 = 0.01 ** 2
        c2 = 0.03 ** 2

        ssim_val = ((2 * mu1 * mu2 + c1) * (2 * sigma12 + c2)) / \
                   ((mu1**2 + mu2**2 + c1) * (sigma1**2 + sigma2**2 + c2))
        return ssim_val

# Initialize loss
criterion = CombinedLoss(hparams).to(device)
```

```
    print(f"✓ Loss function: {hparams.loss_type}")
    print(f"✓ MSE weight: {hparams.mse_weight}, L1 weight: {hparams.l1_weight}")
```

```
          (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): LeakyReLU(negative_slope=0.2, inplace=True)
        )
      )
    )
    (bottleneck): Sequential(
      (0): ResidualBlock(
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)

          (0): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
```

```
        (0): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
      )
    )
    (2): ConvBlock(
      (block): Sequential(
        (0): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
      )
    )
  )
  (final): Sequential(
    (0): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): Tanh()
  )
)
✓ Loss function: combined
✓ MSE weight: 0.5, L1 weight: 0.5
```

```
# ╔══════════════════════════════════════════════════════════════╗
# ║  CELL 7: Optimizer & Scheduler (Professional)                ║
# ╚══════════════════════════════════════════════════════════════╝

def get_optimizer(model, hparams):
    """Configure optimizer based on hyperparameters"""

    if hparams.optimizer == 'adam':
        optimizer = optim.Adam(
            model.parameters(),
            lr=hparams.learning_rate,
            betas=(hparams.beta1, hparams.beta2),
            weight_decay=hparams.weight_decay
        )
    elif hparams.optimizer == 'adamw':
        optimizer = optim.AdamW(
            model.parameters(),
            lr=hparams.learning_rate,
            betas=(hparams.beta1, hparams.beta2),
```

```
                weight_decay=hparams.weight_decay
            )
        elif hparams.optimizer == 'sgd':
            optimizer = optim.SGD(
                model.parameters(),
                lr=hparams.learning_rate,
                momentum=0.9,
                weight_decay=hparams.weight_decay
            )
        else:
            raise ValueError(f"Unknown optimizer: {hparams.optimizer}")

        return optimizer

    def get_scheduler(optimizer, hparams):
        """Configure learning rate scheduler"""

        if hparams.scheduler_type == 'step':
            scheduler = optim.lr_scheduler.StepLR(
                optimizer, step_size=30, gamma=0.5
            )
        elif hparams.scheduler_type == 'cosine':
            scheduler = optim.lr_scheduler.CosineAnnealingWarmRestarts(
                optimizer, T_0=10, T_mult=2, eta_min=1e-6
            )
        elif hparams.scheduler_type == 'plateau':
            scheduler = optim.lr_scheduler.ReduceLROnPlateau(
                optimizer, mode='min', factor=0.5, patience=5
            )
        elif hparams.scheduler_type == 'onecycle':
            scheduler = optim.lr_scheduler.OneCycleLR(
                optimizer,
                max_lr=hparams.learning_rate,
                epochs=hparams.epochs,
                steps_per_epoch=len(train_loader),
                pct_start=0.3
            )
        else:
```

```
        scheduler = None

    return scheduler

optimizer = get_optimizer(model, hparams)
scheduler = get_scheduler(optimizer, hparams)

print(f"✓ Optimizer: {hparams.optimizer}")
print(f"✓ Initial LR: {hparams.learning_rate}")
print(f"✓ Scheduler: {hparams.scheduler_type}")
print(f"✓ Weight decay: {hparams.weight_decay}")
```

```
✓ Optimizer: adamw
✓ Initial LR: 0.0002
✓ Scheduler: cosine
✓ Weight decay: 1e-05
```

```
# ╔════════════════════════════════════════════════════════════╗
# ║  CELL 8: Checkpoint Manager (Professional)                 ║
# ╚════════════════════════════════════════════════════════════╝

class CheckpointManager:
    """Professional checkpoint management with best model tracking"""

    def __init__(self, checkpoint_dir: str, keep_best: int = 3):
        self.checkpoint_dir = checkpoint_dir
        self.keep_best = keep_best
        self.best_losses = []  # List of (loss, path) tuples

    def save(self, model, optimizer, scheduler, epoch: int,
             train_loss: float, val_loss: float, is_best: bool = False):
        """Save checkpoint"""

        checkpoint = {
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
```

```
                'scheduler_state_dict': scheduler.state_dict() if scheduler else None,
                'train_loss': train_loss,
                'val_loss': val_loss,
                'hyperparameters': hparams.to_dict()
            }

            # Regular checkpoint
            regular_path = f"{self.checkpoint_dir}/checkpoint_epoch_{epoch:03d}.pt"
            torch.save(checkpoint, regular_path)
            print(f"✓ Saved checkpoint: {regular_path}")

            # Best checkpoint
            if is_best:
                best_path = f"{self.checkpoint_dir}/best_model.pt"
                torch.save(checkpoint, best_path)
                print(f"✓ Saved best model: {best_path}")

                # Track best losses
                self.best_losses.append((val_loss, best_path))
                self.best_losses.sort(key=lambda x: x[0])

                # Remove old best checkpoints if exceeding keep_best
                if len(self.best_losses) > self.keep_best:
                    # Keep only the best ones
                    self.best_losses = self.best_losses[:self.keep_best]

        def load_latest(self, model, optimizer, scheduler):
            """Load the most recent checkpoint"""
            checkpoints = [f for f in os.listdir(self.checkpoint_dir)
                          if f.startswith('checkpoint_epoch_')]

            if not checkpoints:
                return None, 0

            # Sort by epoch number
            checkpoints.sort()
            latest = checkpoints[-1]
            path = f"{self.checkpoint_dir}/{latest}"
```

```
            return self.load(path, model, optimizer, scheduler)

    def load_best(self, model, optimizer, scheduler):
        """Load the best checkpoint"""
        path = f"{self.checkpoint_dir}/best_model.pt"
        if os.path.exists(path):
            return self.load(path, model, optimizer, scheduler)
        return None, 0

    def load(self, path: str, model, optimizer, scheduler):
        """Load specific checkpoint"""
        checkpoint = torch.load(path, map_location=device)

        model.load_state_dict(checkpoint['model_state_dict'])
        optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
        if scheduler and checkpoint['scheduler_state_dict']:
            scheduler.load_state_dict(checkpoint['scheduler_state_dict'])

        epoch = checkpoint['epoch']
        print(f"✓ Loaded checkpoint from epoch {epoch}")
        return checkpoint, epoch

checkpoint_manager = CheckpointManager(CHECKPOINT_DIR, hparams.keep_best)
print(f"✓ Checkpoint manager initialized")
print(f"✓ Keeping top {hparams.keep_best} best models")
```

```
✓ Checkpoint manager initialized
✓ Keeping top 3 best models
```

```
# ╔══════════════════════════════════════════════════════════╗
# ║  CELL 9: Training Loop (Professional with All Features)  ║
# ╚══════════════════════════════════════════════════════════╝

class Trainer:
    """Professional training manager"""
```

```
    def __init__(self, model, criterion, optimizer, scheduler, hparams):
        self.model = model
        self.criterion = criterion
        self.optimizer = optimizer
        self.scheduler = scheduler
        self.hparams = hparams
        self.writer = SummaryWriter(LOGS_DIR)
        self.global_step = 0
        self.best_val_loss = float('inf')
        self.history = {
            'train_loss': [], 'val_loss': [],
            'train_psnr': [], 'val_psnr': [],
            'learning_rates': []
        }

    def train_epoch(self, dataloader, epoch):
        """Train for one epoch"""
        self.model.train()
        total_loss = 0
        total_psnr = 0
        num_batches = 0

        pbar = tqdm(dataloader, desc=f"Epoch {epoch}/{self.hparams.epochs} [Train]")

        for batch_idx, batch in enumerate(pbar):
            inputs = batch['input'].to(device)
            targets = batch['target'].to(device)

            # Forward pass
            outputs = self.model(inputs)
            loss = self.criterion(outputs, targets)

            # Backward pass
            self.optimizer.zero_grad()
            loss.backward()

            # Gradient clipping
            if self.hparams.gradient_clip > 0:
```

```
                torch.nn.utils.clip_grad_norm_(
                    self.model.parameters(),
                    self.hparams.gradient_clip
                )

            self.optimizer.step()

            # Metrics
            with torch.no_grad():
                psnr_val = Metrics.psnr(outputs, targets).item()

            total_loss += loss.item()
            total_psnr += psnr_val
            num_batches += 1

            # Logging
            if batch_idx % self.hparams.log_freq == 0:
                self.writer.add_scalar('Loss/train_batch', loss.item(), self.global_step)
                self.writer.add_scalar('PSNR/train_batch', psnr_val, self.global_step)

                # Update progress bar
                pbar.set_postfix({
                    'loss': f"{loss.item():.4f}",
                    'psnr': f"{psnr_val:.2f}",
                    'lr': f"{self.optimizer.param_groups[0]['lr']:.6f}"
                })

            self.global_step += 1

            # Update scheduler if OneCycle
            if isinstance(self.scheduler, optim.lr_scheduler.OneCycleLR):
                self.scheduler.step()

        avg_loss = total_loss / num_batches
        avg_psnr = total_psnr / num_batches

        return avg_loss, avg_psnr
```

```
    @torch.no_grad()
    def validate(self, dataloader, epoch):
        """Validation loop"""
        self.model.eval()
        total_loss = 0
        total_psnr = 0
        num_batches = 0

        all_inputs = []
        all_outputs = []
        all_targets = []

        pbar = tqdm(dataloader, desc=f"Epoch {epoch}/{self.hparams.epochs} [Val]")

        for batch in pbar:
            inputs = batch['input'].to(device)
            targets = batch['target'].to(device)

            outputs = self.model(inputs)
            loss = self.criterion(outputs, targets)

            psnr_val = Metrics.psnr(outputs, targets).item()

            total_loss += loss.item()
            total_psnr += psnr_val
            num_batches += 1

            # Store samples for visualization
            if len(all_inputs) < 64:  # Store up to 64 samples
                all_inputs.append(inputs.cpu())
                all_outputs.append(outputs.cpu())
                all_targets.append(targets.cpu())

            pbar.set_postfix({
                'loss': f"{loss.item():.4f}",
                'psnr': f"{psnr_val:.2f}"
            })
```

```
        # Concatenate samples
        all_inputs = torch.cat(all_inputs)[:64]
        all_outputs = torch.cat(all_outputs)[:64]
        all_targets = torch.cat(all_targets)[:64]

        avg_loss = total_loss / num_batches
        avg_psnr = total_psnr / num_batches

        return avg_loss, avg_psnr, (all_inputs, all_outputs, all_targets)

    def visualize_results(self, samples, epoch):
        """Save visualization of results"""
        inputs, outputs, targets = samples

        # Denormalize from [-1, 1] to [0, 1] for visualization
        def denorm(x):
            return (x + 1) / 2

        inputs = denorm(inputs)
        outputs = denorm(outputs)
        targets = denorm(targets)

        # Create grid: [Input | Output | Target]
        comparison = torch.cat([inputs, outputs, targets], dim=0)
        grid = make_grid(comparison, nrow=8, normalize=False, value_range=(0, 1))

        # Save image
        save_path = f"{RESULTS_DIR}/epoch_{epoch:03d}.png"
        save_image(grid, save_path)

        # Add to tensorboard
        self.writer.add_image('Results/Input_Output_Target', grid, epoch)

        return save_path

    def train(self, train_loader, val_loader, checkpoint_manager):
        """Full training loop"""
```

```
        print(f"\n{'='*60}")
        print(f"Starting Training: {self.hparams.epochs} epochs")
        print(f"{'='*60}\n")

        start_epoch = 1

        # Try to resume from checkpoint
        checkpoint, resumed_epoch = checkpoint_manager.load_latest(
            self.model, self.optimizer, self.scheduler
        )
        if checkpoint:
            start_epoch = resumed_epoch + 1
            self.best_val_loss = checkpoint.get('val_loss', float('inf'))
            print(f"Resuming from epoch {start_epoch}")

        for epoch in range(start_epoch, self.hparams.epochs + 1):
            epoch_start_time = time.time()

            # Train
            train_loss, train_psnr = self.train_epoch(train_loader, epoch)

            # Validate
            val_loss, val_psnr, samples = self.validate(val_loader, epoch)

            # Scheduler step (if not OneCycle)
            if self.scheduler and not isinstance(self.scheduler, optim.lr_scheduler.OneCycleLR):
                if isinstance(self.scheduler, optim.lr_scheduler.ReduceLROnPlateau):
                    self.scheduler.step(val_loss)
                else:
                    self.scheduler.step()

            current_lr = self.optimizer.param_groups[0]['lr']

            # Update history
            self.history['train_loss'].append(train_loss)
            self.history['val_loss'].append(val_loss)
            self.history['train_psnr'].append(train_psnr)
            self.history['val_psnr'].append(val_psnr)
```

```
                self.history['learning_rates'].append(current_lr)

                # TensorBoard logging
                self.writer.add_scalar('Loss/train', train_loss, epoch)
                self.writer.add_scalar('Loss/val', val_loss, epoch)
                self.writer.add_scalar('PSNR/train', train_psnr, epoch)
                self.writer.add_scalar('PSNR/val', val_psnr, epoch)
                self.writer.add_scalar('Learning_Rate', current_lr, epoch)

                # Visualization
                if epoch % self.hparams.sample_freq == 0 or epoch == 1:
                    vis_path = self.visualize_results(samples, epoch)

                # Checkpointing
                is_best = val_loss < self.best_val_loss
                if is_best:
                    self.best_val_loss = val_loss

                if epoch % self.hparams.save_freq == 0 or is_best or epoch == self.hparams.epochs:
                    checkpoint_manager.save(
                        self.model, self.optimizer, self.scheduler,
                        epoch, train_loss, val_loss, is_best
                    )

                # Print epoch summary
                epoch_time = time.time() - epoch_start_time
                print(f"\nEpoch {epoch}/{self.hparams.epochs} Summary:")
                print(f"  Time: {epoch_time:.2f}s | LR: {current_lr:.6f}")
                print(f"  Train Loss: {train_loss:.4f} | PSNR: {train_psnr:.2f}")
                print(f"  Val Loss: {val_loss:.4f} | PSNR: {val_psnr:.2f}")
                print(f"  Best Val Loss: {self.best_val_loss:.4f} {'⭐' if is_best else ''}")
                print("-" * 60)

            # Save final history
            history_path = f"{RESULTS_DIR}/training_history.json"
            with open(history_path, 'w') as f:
                json.dump(self.history, f, indent=4)
```

```
        self.writer.close()
        print(f"\n✓ Training completed! Best Val Loss: {self.best_val_loss:.4f}")
        print(f"✓ Results saved to: {RESULTS_DIR}")
        print(f"✓ Checkpoints saved to: {CHECKPOINT_DIR}")

        return self.history
```

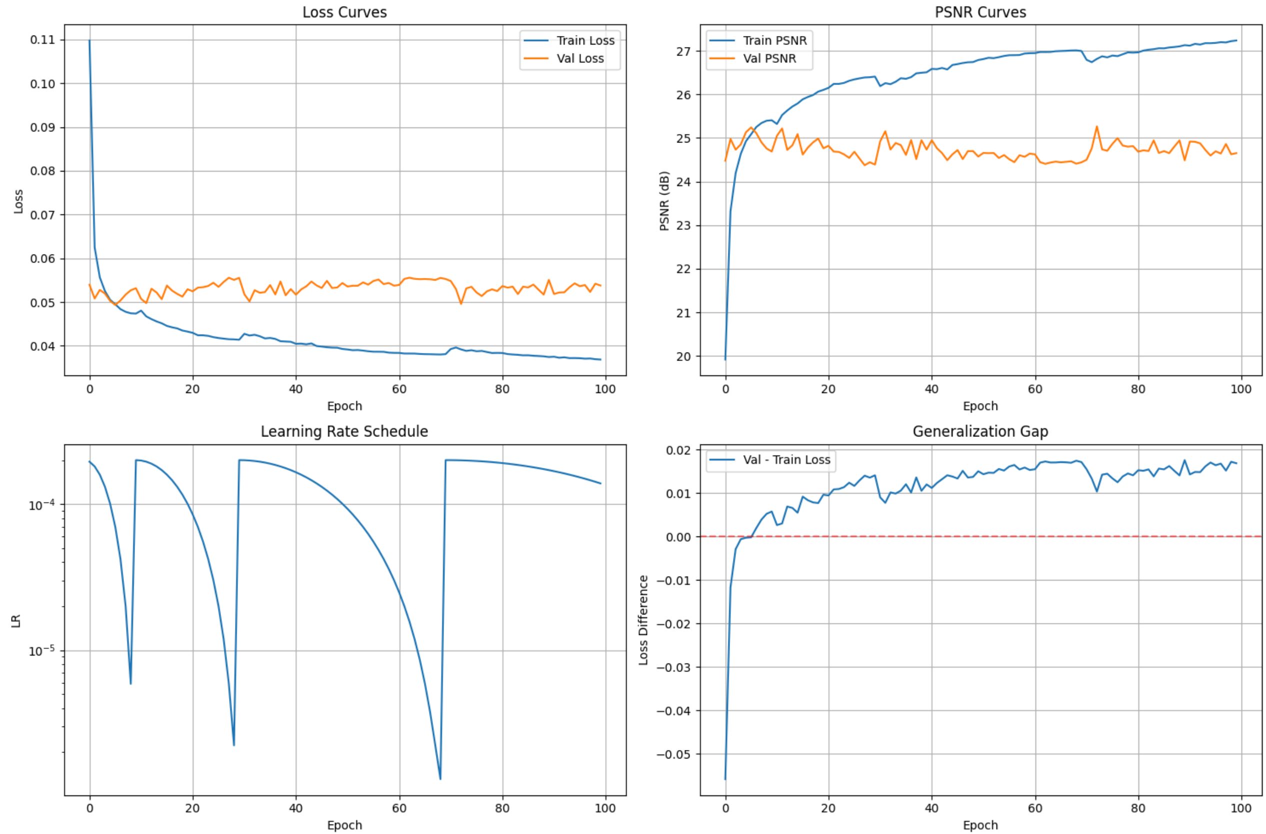
```
# ╔══════════════════════════════════════════════════════════╗
# ║  CELL 10: Initialize & Run Training                      ║
# ╚══════════════════════════════════════════════════════════╝

from tqdm import tqdm

# Initialize trainer
trainer = Trainer(model, criterion, optimizer, scheduler, hparams)

# Run training
history = trainer.train(train_loader, test_loader, checkpoint_manager)
```

```
Epoch 80/100 Summary:
```

```
Epoch 89/100 Summary:
  Time: 27.92s | LR: 0.000174
  Train Loss: 0.0376 | PSNR: 27.10
  Val Loss: 0.0517 | PSNR: 24.95
  Best Val Loss: 0.0493
------------------------------------------------------------
Epoch 90/100 [Train]: 100%|██████████| 390/390 [00:23<00:00, 16.71it/s, loss=0.0376, psnr=27.09, lr=0.000174]
Epoch 90/100 [Val]: 100%|██████████| 79/79 [00:04<00:00, 17.85it/s, loss=0.0570, psnr=24.29]
✓ Saved checkpoint: /content/drive/MyDrive/CSET419_Lab5_EncoderDecoder/checkpoints/checkpoint_epoch_090.pt

Epoch 90/100 Summary:
  Time: 28.85s | LR: 0.000171
  Train Loss: 0.0374 | PSNR: 27.13
  Val Loss: 0.0551 | PSNR: 24.49
  Best Val Loss: 0.0493
------------------------------------------------------------
Epoch 91/100 [Train]: 100%|██████████| 390/390 [00:24<00:00, 16.21it/s, loss=0.0377, psnr=27.03, lr=0.000171]
Epoch 91/100 [Val]: 100%|██████████| 79/79 [00:04<00:00, 17.55it/s, loss=0.0531, psnr=24.77]

Epoch 91/100 Summary:
  Time: 28.76s | LR: 0.000168
  Train Loss: 0.0375 | PSNR: 27.12
  Val Loss: 0.0518 | PSNR: 24.92
  Best Val Loss: 0.0493
------------------------------------------------------------
Epoch 92/100 [Train]: 100%|██████████| 390/390 [00:22<00:00, 17.19it/s, loss=0.0375, psnr=27.06, lr=0.000168]
Epoch 92/100 [Val]: 100%|██████████| 79/79 [00:05<00:00, 14.78it/s, loss=0.0544, psnr=24.64]

Epoch 92/100 Summary:
  Time: 28.23s | LR: 0.000165
  Train Loss: 0.0373 | PSNR: 27.16
  Val Loss: 0.0522 | PSNR: 24.91
  Best Val Loss: 0.0493
------------------------------------------------------------
Epoch 93/100 [Train]: 100%|██████████| 390/390 [00:22<00:00, 17.13it/s, loss=0.0360, psnr=27.43, lr=0.000165]
Epoch 93/100 [Val]: 100%|██████████| 79/79 [00:04<00:00, 16.09it/s, loss=0.0534, psnr=24.76]

Epoch 93/100 Summary:
  Time: 27.96s | LR: 0.000162
```

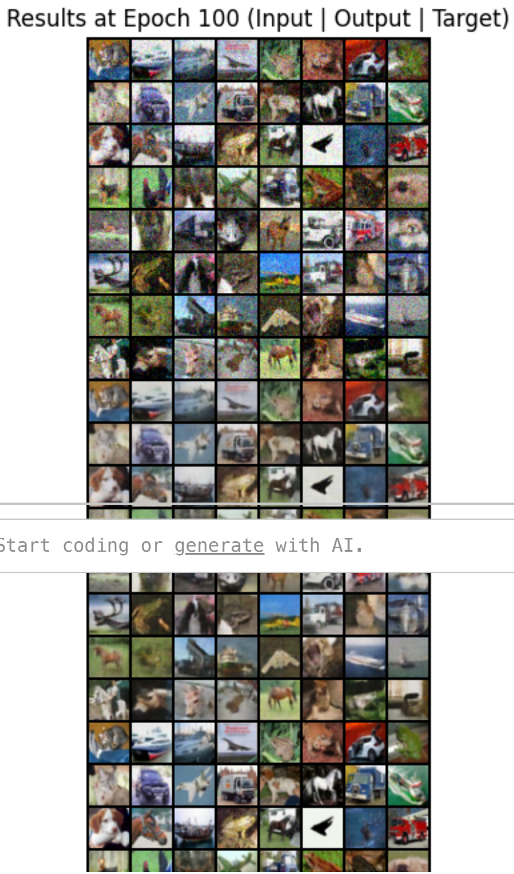
```
# ╔══════════════════════════════════════════════════════════════╗
# ║  CELL 11: Evaluation & Analysis                              ║
# ╚══════════════════════════════════════════════════════════════╝

import matplotlib.pyplot as plt
from PIL import Image

def plot_training_history(history):
    """Plot training curves"""
    fig, axes = plt.subplots(2, 2, figsize=(15, 10))

    # Loss curves
    axes[0, 0].plot(history['train_loss'], label='Train Loss')
    axes[0, 0].plot(history['val_loss'], label='Val Loss')
    axes[0, 0].set_title('Loss Curves')
    axes[0, 0].set_xlabel('Epoch')
    axes[0, 0].set_ylabel('Loss')
    axes[0, 0].legend()
    axes[0, 0].grid(True)

    # PSNR curves
    axes[0, 1].plot(history['train_psnr'], label='Train PSNR')
    axes[0, 1].plot(history['val_psnr'], label='Val PSNR')
    axes[0, 1].set_title('PSNR Curves')
    axes[0, 1].set_xlabel('Epoch')
    axes[0, 1].set_ylabel('PSNR (dB)')
    axes[0, 1].legend()
    axes[0, 1].grid(True)

    # Learning rate
    axes[1, 0].plot(history['learning_rates'])
    axes[1, 0].set_title('Learning Rate Schedule')
    axes[1, 0].set_xlabel('Epoch')
    axes[1, 0].set_ylabel('LR')
    axes[1, 0].set_yscale('log')
    axes[1, 0].grid(True)

    # Loss difference (overfitting indicator)
```

```
        diff = np.array(history['val_loss']) - np.array(history['train_loss'])
        axes[1, 1].plot(diff, label='Val - Train Loss')
        axes[1, 1].axhline(y=0, color='r', linestyle='--', alpha=0.5)
        axes[1, 1].set_title('Generalization Gap')
        axes[1, 1].set_xlabel('Epoch')
        axes[1, 1].set_ylabel('Loss Difference')
        axes[1, 1].legend()
        axes[1, 1].grid(True)

        plt.tight_layout()
        plt.savefig(f"{RESULTS_DIR}/training_analysis.png", dpi=300, bbox_inches='tight')
        plt.show()

    # Plot history
    plot_training_history(history)

    # Show final results
    print("\n" + "="*60)
    print("FINAL RESULTS SUMMARY")
    print("="*60)
    print(f"Best Validation Loss: {min(history['val_loss']):.6f}")
    print(f"Best Validation PSNR: {max(history['val_psnr']):.2f} dB")
    print(f"Final Training Loss: {history['train_loss'][-1]:.6f}")
    print(f"Final Validation Loss: {history['val_loss'][-1]:.6f}")
    print("="*60)

    # Display sample results
    from IPython.display import display
    latest_result = f"{RESULTS_DIR}/epoch_{hparams.epochs:03d}.png"
    if os.path.exists(latest_result):
        img = Image.open(latest_result)
        plt.figure(figsize=(20, 10))
        plt.imshow(img)
        plt.axis('off')
        plt.title(f"Results at Epoch {hparams.epochs} (Input | Output | Target)")
        plt.show()
```




```
============================================================
FINAL RESULTS SUMMARY
============================================================
Best Validation Loss: 0.049331
```

```
Best Validation PSNR: 25.27 dB
Final Training Loss: 0.036855
Final Validation Loss: 0.053773
============================================================
```



Start coding or generate with AI.