Neural Network Modules

```
import torch.nn as nn
import torch.nn.functional as F
import torch


class Flatten(nn.Module):
    def forward(self, x):
        return x.view(x.size(0), -1)

class ChannelGate(nn.Module):
    def __init__(self, gate_channels, reduction_ratio=16, pool_types=['avg', 'max']):
        super(ChannelGate, self).__init__()
        self.gate_channels = gate_channels
        self.mlp = nn.Sequential(
            Flatten(),
            nn.Linear(gate_channels, gate_channels // reduction_ratio),
            nn.ReLU(),
            nn.Linear(gate_channels // reduction_ratio, gate_channels)
            )
        self.pool_types = pool_types
    def forward(self, x):
        channel_att_sum = None
        for pool_type in self.pool_types:
            if pool_type=='avg':
                avg_pool = F.avg_pool2d( x, (x.size(2), x.size(3)), stride=(x.size(2), x.size(3)))
                channel_att_raw = self.mlp( avg_pool )
            elif pool_type=='max':
                max_pool = F.max_pool2d( x, (x.size(2), x.size(3)), stride=(x.size(2), x.size(3)))
                channel_att_raw = self.mlp( max_pool )
            elif pool_type=='lp':
                lp_pool = F.lp_pool2d( x, 2, (x.size(2), x.size(3)), stride=(x.size(2), x.size(3)))
                channel_att_raw = self.mlp( lp_pool )
            elif pool_type=='lse':
                # LSE pool only
                lse_pool = logsumexp_2d(x)
                channel_att_raw = self.mlp( lse_pool )

            if channel_att_sum is None:
                channel_att_sum = channel_att_raw
            else:
                channel_att_sum = channel_att_sum + channel_att_raw
        scale = torch.sigmoid( channel_att_sum ).unsqueeze(2).unsqueeze(3).expand_as(x)
        return x * scale
```

```
class BasicConv(nn.Module):
    def __init__(self, in_planes, out_planes, kernel_size, stride=1, padding=0, dilation=1, groups=1,
relu=True, bn=True, bias=False):
        super(BasicConv, self).__init__()
        self.out_channels = out_planes
        self.conv = nn.Conv2d(in_planes, out_planes, kernel_size=kernel_size, stride=stride,
padding=padding, dilation=dilation, groups=groups, bias=bias)
        self.bn = nn.BatchNorm2d(out_planes,eps=1e-5, momentum=0.01, affine=True) if bn else None
        self.relu = nn.ReLU() if relu else None

    def forward(self, x):
        x = self.conv(x)
        if self.bn is not None:
            x = self.bn(x)
        if self.relu is not None:
            x = self.relu(x)
        return x

class ChannelPool(nn.Module):
    def forward(self, x):
        return torch.cat( (torch.max(x,1)[0].unsqueeze(1), torch.mean(x,1).unsqueeze(1)), dim=1 )

class SpatialGate(nn.Module):
    def __init__(self):
        super(SpatialGate, self).__init__()
        kernel_size = 7
        self.compress = ChannelPool()
        self.spatial = BasicConv(2, 1, kernel_size, stride=1, padding=(kernel_size-1) // 2, relu=False)
    def forward(self, x):
        x_compress = self.compress(x)
        x_out = self.spatial(x_compress)
        scale = torch.sigmoid(x_out) # broadcasting
        return x * scale
```

explain all the functions in super simple words , what it is doing and give a simple example for what it is doing as well

1. Flatten

- **What it does**:
  - Flattens a multi-dimensional tensor into a 2D tensor where the first dimension represents batch size, and the second dimension is all the other dimensions combined into one.
- **Example**:
  - Input: A tensor of shape `[2, 3, 4]`.
  - Output: A tensor of shape `[2, 12]` (flattening the last two dimensions).

---

2. ChannelGate

- **What it does**:
  - Focuses on important channels (features) of the input tensor by assigning weights to them based on pooled statistics.
  - Uses techniques like average pooling, max pooling, etc., to gather information about each channel.
- **How it works**:
  1. Pools the input tensor (e.g., average pooling or max pooling) to create a channel descriptor.
  2. Passes the descriptor through a small neural network (MLP) to get attention scores.
  3. Combines attention scores from different pooling methods.
  4. Uses these scores to emphasize important channels.
- **Example**:
  - Input: Tensor with shape `` `[batch, channels, height, width]` ``.
  - Output: Tensor of the same shape, but some channels are amplified or reduced.

---

3. BasicConv

- **What it does**:
  - A basic block for 2D convolution that includes:
    - Convolution operation.
    - Batch normalization (optional).
    - Activation function (ReLU, optional).
- **How it works**:
  1. Applies a convolution to extract features.
  2. Normalizes the features (optional).
  3. Activates the features using ReLU (optional).
- **Example**:
  - Input: A tensor of shape `` `[batch, in_channels, height, width]` ``.
  - Output: A tensor of shape `` `[batch, out_channels, new_height, new_width]` ``.

---

4. ChannelPool

- **What it does**:
  - Combines the maximum and average values along the channel dimension into a single representation.
- **How it works**:
  1. Computes the maximum value for each pixel across channels.
  2. Computes the average value for each pixel across channels.
  3. Stacks these two results into a new tensor.
- **Example**:
  - Input: Tensor with shape `` `[batch, channels, height, width]` ``.
  - Output: Tensor with shape `` `[batch, 2, height, width]` `` (one channel for max and one for mean).

---

5. SpatialGate

- **What it does**:
  - Focuses on important spatial regions of the input tensor by assigning weights to spatial locations.
- **How it works**:

1. Compresses the input tensor along the channel dimension (using `ChannelPool`).
2. Passes the compressed tensor through a convolutional layer to generate spatial attention scores.
3. Uses these scores to emphasize important spatial regions.

- **Example**:
  - Input: Tensor with shape `[batch, channels, height, width]`.
  - Output: Tensor of the same shape, but certain spatial regions are emphasized or reduced.

Putting It Together

The **ChannelGate** focuses on the "importance" of channels (which features are critical), while the **SpatialGate** focuses on "where" in the spatial dimensions we should pay more attention. These gates can be combined to improve a network's ability to focus on relevant information.

For example:

- Input: An image of a cat.
- **ChannelGate**: Highlights channels that represent features like "edges" or "textures."
- **SpatialGate**: Focuses on spatial regions like the cat's face or paws, ignoring the background.

and even explain the codes of each function innsimple

Let's go through each function and break the code into **simple explanations** step by step.

---

1. `Flatten`

Code:

```python
class Flatten(nn.Module):
    def forward(self, x):
        return x.view(x.size(0), -1)
```

Explanation:

- `x.size(0)` gives the **batch size** (the number of examples in the input).
- `view(x.size(0), -1)` reshapes the input into 2 dimensions:
  - First dimension: The batch size.
  - Second dimension: All remaining values (flattened into one long row).
- This is useful for switching from convolution layers (multi-dimensional) to fully connected layers (2D).

Example:

- Input: A tensor of shape `[2, 3, 4]` (2 batches, 3 channels, 4 features).
- Output: `[2, 12]` (flattened per batch).

---

2. `ChannelGate`

Code:

```python
class ChannelGate(nn.Module):
    def __init__(self, gate_channels, reduction_ratio=16, pool_types=['avg', 'max']):
        super(ChannelGate, self).__init__()
        self.gate_channels = gate_channels
        self.mlp = nn.Sequential(
            Flatten(),
            nn.Linear(gate_channels, gate_channels // reduction_ratio),
            nn.ReLU(),
            nn.Linear(gate_channels // reduction_ratio, gate_channels)
        )
        self.pool_types = pool_types
```

Explanation:

- **Initialization**:
  - Takes the number of channels (`gate_channels`) and reduces it using a `reduction_ratio` (to make the model smaller and faster).
  - Defines a **multi-layer perceptron (MLP)**:
    - **Flatten**: Prepares data for fully connected layers.
    - **Linear Layers**: Reduce (`gate_channels // reduction_ratio`) and restore (`gate_channels`) the dimensionality.
    - **ReLU**: Adds non-linearity to learn better.
  - **Pooling types (`avg`, `max`)**: Methods to extract information from the input.

Forward Pass:

```python
    def forward(self, x):
        channel_att_sum = None
        for pool_type in self.pool_types:
            if pool_type == 'avg':
                avg_pool = F.avg_pool2d(x, (x.size(2), x.size(3)), stride=(x.size(2), x.size
(3)))
                channel_att_raw = self.mlp(avg_pool)
```

1. Loops through each pooling type (e.g., average pooling, max pooling).
2. `F.avg_pool2d`:
   - Shrinks the input along height (`x.size(2)`) and width (`x.size(3)`), leaving only channel information.
3. Passes the pooled result through the MLP to calculate attention scores for each channel.

Example Flow:

Input: `[batch, channels, height, width]`.

- Applies **average pooling** or **max pooling** to reduce dimensions to `[batch, channels, 1, 1]`.
- Passes it through the **MLP** to get scores for each channel.
- Final output: Adjusted tensor where important channels are emphasized.

---

3. `BasicConv`

Code:

```python
class BasicConv(nn.Module):
    def __init__(self, in_planes, out_planes, kernel_size, stride=1, padding=0, dilation=1,
groups=1, relu=True, bn=True, bias=False):
        super(BasicConv, self).__init__()
        self.conv = nn.Conv2d(in_planes, out_planes, kernel_size, stride, padding, dilation,
groups, bias)
        self.bn = nn.BatchNorm2d(out_planes, eps=1e-5, momentum=0.01, affine=True) if bn els
e None
        self.relu = nn.ReLU() if relu else None
```

Explanation:

- **Convolution** (`nn.Conv2d`): Extracts features from the input.
- **Batch Normalization** (`nn.BatchNorm2d`): Normalizes the features for faster training (optional).
- **ReLU** (`nn.ReLU`): Activates the features to make them non-linear (optional).

Forward Pass:

```python
    def forward(self, x):
        x = self.conv(x)
        if self.bn is not None:
            x = self.bn(x)
        if self.relu is not None:
            x = self.relu(x)
        return x
```

- Applies convolution, then batch normalization, then ReLU (if specified).

Example:

Input: Image of size `[batch, 3, 32, 32]`.

- Output: Processed image of size `[batch, out_channels, new_height, new_width]`.

---

4. `ChannelPool`

OpenAI

Code:

```python
class ChannelPool(nn.Module):
    def forward(self, x):
        return torch.cat((torch.max(x, 1)[0].unsqueeze(1), torch.mean(x, 1).unsqueeze(1)), dim=1)
```

Explanation:

- Combines **max pooling** and **mean pooling** along the channel dimension:
  - `torch.max(x, 1)[0]`: Finds the maximum value for each pixel across channels.
  - `torch.mean(x, 1)`: Computes the average value for each pixel across channels.
  - `unsqueeze(1)`: Adds a dimension to match tensor shapes.
- `torch.cat`: Combines the results into a new tensor with two channels.

Example:

Input: `[batch, 3, height, width]`.

- Output: `[batch, 2, height, width]` (max-pool and mean-pool channels).

---

5. `SpatialGate`

Code:

```python
class SpatialGate(nn.Module):
    def __init__(self):
        super(SpatialGate, self).__init__()
        kernel_size = 7
        self.compress = ChannelPool()
        self.spatial = BasicConv(2, 1, kernel_size, stride=1, padding=(kernel_size-1) // 2, relu=False)
```

Explanation:

- **ChannelPool**: Compresses the input along channels (using `max` and `mean`).
- **BasicConv**: A convolution that processes the compressed input to focus on important spatial regions.

Forward Pass:

```python
    def forward(self, x):
        x_compress = self.compress(x)
        x_out = self.spatial(x_compress)
```

```
        scale = torch.sigmoid(x_out)
        return x * scale
```

1. Compresses the input using max and mean pooling (output: `[batch, 2, height, width]`).
2. Passes the compressed tensor through a convolution to compute spatial attention scores.
3. Scales the original input using `torch.sigmoid` to emphasize important regions.

Summary of Each Class:

- **Flatten**: Turns multi-dimensional data into flat data.
- **ChannelGate**: Focuses on important features (channels).
- **BasicConv**: Performs convolution, normalization, and activation.
- **ChannelPool**: Combines `max` and `mean` pooling for channel compression.
- **SpatialGate**: Focuses on important regions in an image (spatial attention).

```
import glob
import os
import cv2
from torch.utils.data import Dataset


class CDRL_Dataset(Dataset):
    def __init__(self, root_path=None, dataset=None, train_val=None, transforms_A=None,
transforms_B=None):
        self.total_path = os.path.join(root_path, dataset, train_val)
        self.transforms_A = transforms_A
        self.transforms_B = transforms_B
        self.files = sorted(glob.glob(self.total_path + "/A/*.*")) +\
                     sorted(glob.glob(self.total_path + "/B/*.*"))
        self.A2BB2A_path = self.total_path.replace(dataset, dataset+'_A2B_B2A')

    def __getitem__(self, index):
        img_name = self.files[index % len(self.files)].split('/')[-1]

        img_A = cv2.imread(self.files[index % len(self.files)], cv2.IMREAD_COLOR)
        img_ori = img_A.copy()

        if '/A/' in self.files[index % len(self.files)]:
            img_B = cv2.imread(self.A2BB2A_path + '/A/'+img_name, cv2.IMREAD_COLOR)
        elif '/B/' in self.files[index % len(self.files)]:
            img_B = cv2.imread(self.A2BB2A_path + '/B/'+img_name, cv2.IMREAD_COLOR)

        transformed_A = self.transforms_A(image=img_A)
        transformed_B = self.transforms_B(image=img_B)

        img_A = transformed_A["image"]
        img_B = transformed_B["image"]
```

```
        return {"A":img_A , "B": img_B}

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




class CDRL_Dataset_test(Dataset):
    def __init__(self, root_path=None, dataset=None, transforms=None):
        self.total_path = os.path.join(root_path, dataset, 'val')
        self.transforms = transforms
        self.files = sorted(glob.glob(self.total_path + "/A/*.*"))

    def __getitem__(self, index):
        name = self.files[index % len(self.files)].split('/')[-1]

        img_A = cv2.imread(self.files[index % len(self.files)], cv2.IMREAD_COLOR)
        img_B = cv2.imread(self.files[index % len(self.files)].replace('/A/','/B/'), cv2.IMREAD_COLOR)

        transformed_A = self.transforms(image=img_A)
        transformed_B = self.transforms(image=img_B)

        img_A = transformed_A["image"]
        img_B = transformed_B["image"]

        return {"A": img_A, "B": img_B, 'NAME': name}

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

explain the codes of each class and its functions in simple words and what they do and a simple example for each

1. Class: `CDRL_Dataset`

This class is designed to load and preprocess training or validation datasets for a deep learning model. It works by reading image pairs from specific folder structures.

`__init__` Function: Initialization

```python
def __init__(self, root_path=None, dataset=None, train_val=None, transforms_A=None, transforms_B=None):
    self.total_path = os.path.join(root_path, dataset, train_val)
```

```
        self.transforms_A = transforms_A
        self.transforms_B = transforms_B
        self.files = sorted(glob.glob(self.total_path + "/A/*.*")) + \
                     sorted(glob.glob(self.total_path + "/B/*.*"))
        self.A2BB2A_path = self.total_path.replace(dataset, dataset+'_A2B_B2A')
```

- **What it does**:
  - **Parameters**:
    - `root_path`: The base path of the dataset.
    - `dataset`: The name of the dataset folder.
    - `train_val`: Specifies if the dataset is for training or validation.
    - `transforms_A` and `transforms_B`: Data transformations for images in domains A and B.
  - **Key actions**:
    1. Constructs the full path to the dataset using `os.path.join`.
    2. Finds all images in `A` and `B` folders using `glob` and stores their paths in `self.files`.
    3. Prepares the path for paired data in the `*_A2B_B2A` folder (used later).
- **Example**:
  - Dataset folder structure:

```css
root_path/
  dataset_name/
    train/
      A/
        img1.jpg, img2.jpg
      B/
        img1.jpg, img2.jpg
```

  - `root_path="root_path"`, `dataset="dataset_name"`, `train_val="train"`.
  - `self.files` contains paths to all images in `A` and `B` folders.

`__getitem__` Function: Retrieve a Sample

```python
def __getitem__(self, index):
    img_name = self.files[index % len(self.files)].split('/')[-1]
    img_A = cv2.imread(self.files[index % len(self.files)], cv2.IMREAD_COLOR)
    img_ori = img_A.copy()

    if '/A/' in self.files[index % len(self.files)]:
        img_B = cv2.imread(self.A2BB2A_path + '/A/'+img_name, cv2.IMREAD_COLOR)
    elif '/B/' in self.files[index % len(self.files)]:
        img_B = cv2.imread(self.A2BB2A_path + '/B/'+img_name, cv2.IMREAD_COLOR)

    transformed_A = self.transforms_A(image=img_A)
    transformed_B = self.transforms_B(image=img_B)

    img_A = transformed_A["image"]
    img_B = transformed_B["image"]

    return {"A": img_A, "B": img_B}
```

- **What it does**:
  - Loads an image `img_A` and its paired image `img_B` based on the index.
  - Determines whether the image belongs to the `A` or `B` domain and fetches its pair from the corresponding folder.
  - Applies transformations (`transforms_A` and `transforms_B`) to preprocess the images.
  - Returns the processed pair as a dictionary: `{"A": img_A, "B": img_B}`.
- **Example**:
  - Input: `index = 0`.
  - Output:

    ```python
    {"A": <transformed_image_A>, "B": <transformed_image_B>}
    ```

`__len__` Function: Dataset Size

```python
def __len__(self):
    return len(self.files)
```

- **What it does**: Returns the number of image pairs in the dataset.
- **Example**:
  - If `self.files` contains 100 images, `__len__` returns `100`.

---

2. Class: `CDRL_Dataset_test`

This class is similar to `CDRL_Dataset`, but it's specifically designed for testing/validation. It assumes that each image in the `A` folder has a corresponding image in the `B` folder.

`__init__` Function: Initialization

```python
def __init__(self, root_path=None, dataset=None, transforms=None):
    self.total_path = os.path.join(root_path, dataset, 'val')
    self.transforms = transforms
    self.files = sorted(glob.glob(self.total_path + "/A/*.*"))
```

- **What it does**:
  - **Parameters**:
    - `root_path`: The base path of the dataset.
    - `dataset`: The name of the dataset folder.
    - `transforms`: Data transformations for both `A` and `B` images.
  - Constructs the path to the `val` folder (testing data).

- Finds all images in the `A` folder and stores their paths in `self.files`.
- **Example**:
  - Dataset folder structure:

```css
root_path/
  dataset_name/
    val/
      A/
        img1.jpg, img2.jpg
      B/
        img1.jpg, img2.jpg
```

  - `root_path="root_path"`, `dataset="dataset_name"`.
  - `self.files` contains paths to all images in the `A` folder.

`__getitem__` Function: Retrieve a Test Sample

```python
def __getitem__(self, index):
    name = self.files[index % len(self.files)].split('/')[-1]
    img_A = cv2.imread(self.files[index % len(self.files)], cv2.IMREAD_COLOR)
    img_B = cv2.imread(self.files[index % len(self.files)].replace('/A/', '/B/'), cv2.IMREAD
_COLOR)

    transformed_A = self.transforms(image=img_A)
    transformed_B = self.transforms(image=img_B)

    img_A = transformed_A["image"]
    img_B = transformed_B["image"]

    return {"A": img_A, "B": img_B, 'NAME': name}
```

- **What it does**:
  - Loads an image `img_A` and its paired image `img_B` (corresponding file in the `B` folder).
  - Applies the same transformation (`transforms`) to both images.
  - Returns the processed pair along with the image name as a dictionary: `{"A": img_A, "B": img_B, 'NAME': name}`.
- **Example**:
  - Input: `index = 0`.
  - Output:

```python
{"A": <transformed_image_A>, "B": <transformed_image_B>, 'NAME': "img1.jpg"}
```

`__len__` Function: Dataset Size

OpenAI

```python
def __len__(self):
    return len(self.files)
```

- **What it does**: Returns the number of test images in the `A` folder.
- **Example**:
  - If `self.files` contains 50 images, `__len__` returns `50`.

Comparison of the Two Classes

Feature	`CDRL_Dataset`	`CDRL_Dataset_test`
Purpose	Training/Validation Data	Testing Data
Folder Structure	Handles `A`, `B`, and paired folders like `*_A2B_B2A`.	Assumes corresponding images in `A` and `B`.
Transformations	Separate for `A` and `B`.	Same transformation for both `A` and `B`.
Output	Processed image pairs.	Processed image pairs with the image name.

```
import argparse
import os
import numpy as np
import math
import itertools
import time
import datetime
import sys

from torchvision.utils import save_image
from torch.utils.data import DataLoader
from torch.autograd import Variable

from models import *
from datasets import *

import torch.nn as nn
import torch

import albumentations as A
from albumentations.pytorch.transforms import ToTensorV2

def get_device():
    if torch.cuda.is_available():
        return torch.device("cuda")
    elif hasattr(torch.backends, 'mps') and torch.backends.mps.is_available():
```

```
        return torch.device("mps")
    else:
        return torch.device("cpu")

def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument("--epoch", type=int, default=0, help="epoch to start training from")
    parser.add_argument("--n_epochs", type=int, default=10, help="number of epochs of training")
    parser.add_argument("--root_path", type=str, default="/workspace/NAS_MOUNT/", help="root
path")
    parser.add_argument("--dataset_name", type=str, default="LEVIR-CD", help="name of the
dataset")
    parser.add_argument("--save_name", type=str, default="levir", help="name of the dataset")
    parser.add_argument("--batch_size", type=int, default=1, help="size of the batches")
    parser.add_argument("--lr", type=float, default=0.0002, help="adam: learning rate")
    parser.add_argument("--b1", type=float, default=0.5, help="adam: decay of first order momentum
of gradient")
    parser.add_argument("--b2", type=float, default=0.999, help="adam: decay of first order
momentum of gradient")
    parser.add_argument("--n_cpu", type=int, default=4, help="number of cpu threads to use during
batch generation")
    parser.add_argument("--img_height", type=int, default=256, help="size of image height")
    parser.add_argument("--img_width", type=int, default=256, help="size of image width")
    parser.add_argument("--sample_interval", type=int, default=2000, help="interval between
sampling of images from generators")
    return parser.parse_args()

def sample_images(val_dataloader, generator, opt, device, batches_done):
    """Sample images function"""
    imgs = next(iter(val_dataloader))
    img_A = imgs["A"].to(device)
    img_B = imgs["B"].to(device)
    img_B_fake = generator(img_A, img_B)
    img_A = img_A[:, [2,1,0],:,:]
    img_B_fake = img_B_fake[:, [2,1,0],:,:]
    img_B = img_B[:, [2,1,0],:,:]
    img_sample = torch.cat((img_A.data, img_B_fake.data, img_B.data), -2)
    save_image(img_sample, "images/%s/%d.png" % (opt.save_name, batches_done), nrow=5,
normalize=True)


def main():
    opt = parse_args()
    print(opt)

    # Create directories
    os.makedirs("images/%s" % opt.save_name, exist_ok=True)
    os.makedirs("saved_models/%s" % opt.save_name, exist_ok=True)

    # Set up device
    device = get_device()
```

```
print(f"Using device: {device}")

# Loss functions
criterion_GAN = torch.nn.MSELoss()
criterion_pixelwise = torch.nn.L1Loss()

lambda_pixel = 100
patch = (1, opt.img_height // 2 ** 4, opt.img_width // 2 ** 4)

# Initialize generator and discriminator
generator = GeneratorUNet_CBAM(in_channels=3).to(device)
discriminator = Discriminator().to(device)

# Move loss functions to device
criterion_GAN = criterion_GAN.to(device)
criterion_pixelwise = criterion_pixelwise.to(device)

# Initialize weights
generator.apply(weights_init_normal)
discriminator.apply(weights_init_normal)

# Optimizers
optimizer_G = torch.optim.Adam(generator.parameters(), lr=opt.lr, betas=(opt.b1, opt.b2))
optimizer_D = torch.optim.Adam(discriminator.parameters(), lr=opt.lr, betas=(opt.b1, opt.b2))

# Configure transforms
transforms_aug = A.Compose([
    A.Resize(opt.img_height, opt.img_width),
    A.Normalize(),
    ToTensorV2()
])

transforms_ori = A.Compose([
    A.Resize(opt.img_height, opt.img_width),
    A.Normalize(),
    ToTensorV2()
])

# Configure data loaders
dataloader = DataLoader(
    CDRL_Dataset(root_path=opt.root_path, dataset=opt.dataset_name, train_val='train',
                 transforms_A=transforms_aug, transforms_B=transforms_ori),
    batch_size=opt.batch_size,
    shuffle=True,
    num_workers=opt.n_cpu,
)

val_dataloader = DataLoader(
    CDRL_Dataset(root_path=opt.root_path, dataset=opt.dataset_name,  train_val='train',
                 transforms_A=transforms_ori, transforms_B=transforms_ori),
    batch_size=10,
```

```
        shuffle=False,
        num_workers=1,
    )

    # ----------
    #  Training
    # ----------
    prev_time = time.time()

    for epoch in range(opt.epoch, opt.n_epochs):
        for i, batch in enumerate(dataloader):
            # Configure input
            img_A = batch["A"].to(device)
            img_B = batch["B"].to(device)

            valid = torch.ones((img_A.size(0), *patch), device=device)
            fake = torch.zeros((img_A.size(0), *patch), device=device)

            # ------------------
            #  Train Generator
            # ------------------
            optimizer_G.zero_grad()

            gener_output = generator(img_A, img_B)
            gener_output_pred = discriminator(gener_output, img_A)

            loss_GAN = criterion_GAN(gener_output_pred, valid)
            loss_pixel = criterion_pixelwise(gener_output, img_A)

            loss_G = loss_GAN + lambda_pixel * loss_pixel
            loss_G.backward()

            optimizer_G.step()

            # ---------------------
            #  Train Discriminator
            # ---------------------
            optimizer_D.zero_grad()

            pred_real = discriminator(img_B, img_A)
            loss_real = criterion_GAN(pred_real, valid)

            B_pred_fake = discriminator(gener_output.detach(), img_A)
            loss_fake = criterion_GAN(B_pred_fake, fake)

            loss_D = 0.5 * (loss_real + loss_fake)
            loss_D.backward()
            optimizer_D.step()

            # --------------
            #  Log Progress
```

```
        # --------------
        batches_done = epoch * len(dataloader) + i
        batches_left = opt.n_epochs * len(dataloader) - batches_done
        time_left = datetime.timedelta(seconds=batches_left * (time.time() - prev_time))
        prev_time = time.time()

        sys.stdout.write(
            "\r[Epoch %d/%d] [Batch %d/%d] [D loss: %f] [G loss: %f, pixel: %f, adv: %f] ETA: %s"
            % (
                epoch,
                opt.n_epochs,
                i,
                len(dataloader),
                loss_D.item(),
                loss_G.item(),
                loss_pixel.item(),
                loss_GAN.item(),
                time_left,
            )
        )

        if batches_done % opt.sample_interval == 0:
            sample_images(val_dataloader, generator, opt, device, batches_done)

    # Save models at end of epoch
    torch.save(generator.state_dict(), "saved_models/%s/generator_%d.pth" % (opt.save_name,
epoch))
    torch.save(discriminator.state_dict(), "saved_models/%s/discriminator_%d.pth" %
(opt.save_name, epoch))


if __name__ == "__main__":
    main()
```

same like above explain all the functions in super simple words with codes and a simple example for each

1. `get_device` Function

```python
def get_device():
    if torch.cuda.is_available():
        return torch.device("cuda")
    elif hasattr(torch.backends, 'mps') and torch.backends.mps.is_available():
        return torch.device("mps")
    else:
        return torch.device("cpu")
```

- **Purpose**: This function determines whether a GPU (CUDA) or Apple's MPS is available for computation. If neither is available, it defaults to the CPU.
- **Example**:
  - If a CUDA-enabled GPU is available, it returns `"cuda"`.
  - If no GPU is available, it returns `"cpu"`.
  - Example usage:

    ```python
    device = get_device()
    print(device)  # Output: cuda or cpu
    ```

---

2. `parse_args` Function

```python
def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument("--epoch", type=int, default=0, help="epoch to start training from")
    parser.add_argument("--n_epochs", type=int, default=10, help="number of epochs of traini
ng")
    parser.add_argument("--root_path", type=str, default="/workspace/NAS_MOUNT/", help="root
path")
    parser.add_argument("--dataset_name", type=str, default="LEVIR-CD", help="name of the da
taset")
    parser.add_argument("--save_name", type=str, default="levir", help="name of the datase
t")
    parser.add_argument("--batch_size", type=int, default=1, help="size of the batches")
    parser.add_argument("--lr", type=float, default=0.0002, help="adam: learning rate")
    parser.add_argument("--b1", type=float, default=0.5, help="adam: decay of first order mo
mentum of gradient")
    parser.add_argument("--b2", type=float, default=0.999, help="adam: decay of first order
momentum of gradient")
    parser.add_argument("--n_cpu", type=int, default=4, help="number of cpu threads to use d
uring batch generation")
    parser.add_argument("--img_height", type=int, default=256, help="size of image height")
    parser.add_argument("--img_width", type=int, default=256, help="size of image width")
    parser.add_argument("--sample_interval", type=int, default=2000, help="interval between
sampling of images from generators")
    return parser.parse_args()
```

- **Purpose**: This function parses command-line arguments that configure the training and evaluation process.
  - Parameters include the dataset path, training epochs, batch size, learning rate, image size, and others.
- **Example**:
  - Running from the command line:

    ```bash
    python train.py --epoch 5 --batch_size 16
    ```

    This will overwrite the default values of `epoch` and `batch_size`.

---

3. `sample_images` Function

```python
def sample_images(val_dataloader, generator, opt, device, batches_done):
    imgs = next(iter(val_dataloader))
    img_A = imgs["A"].to(device)
    img_B = imgs["B"].to(device)
    img_B_fake = generator(img_A, img_B)
    img_A = img_A[:, [2,1,0],:,:]
    img_B_fake = img_B_fake[:, [2,1,0],:,:]
    img_B = img_B[:, [2,1,0],:,:]
    img_sample = torch.cat((img_A.data, img_B_fake.data, img_B.data), -2)
    save_image(img_sample, "images/%s/%d.png" % (opt.save_name, batches_done), nrow=5, norma
lize=True)
```

- **Purpose**: This function generates a sample of images during training and saves them to visualize progress.
  - `img_A`: Original image.
  - `img_B`: Target image.
  - `img_B_fake`: Image generated by the generator model.
  - Combines these images and saves them in the folder specified in the `opt.save_name`.
- **Example**:
  - After training for 1000 batches, it saves an image comparison:

    ```csharp
    [Original | Generated | Target]
    ```

---

4. `main` Function

```python
def main():
    opt = parse_args()
    print(opt)

    os.makedirs("images/%s" % opt.save_name, exist_ok=True)
    os.makedirs("saved_models/%s" % opt.save_name, exist_ok=True)

    device = get_device()
    print(f"Using device: {device}")

    criterion_GAN = torch.nn.MSELoss()
    criterion_pixelwise = torch.nn.L1Loss()

    lambda_pixel = 100
    patch = (1, opt.img_height // 2 ** 4, opt.img_width // 2 ** 4)

    generator = GeneratorUNet_CBAM(in_channels=3).to(device)
    discriminator = Discriminator().to(device)

    criterion_GAN = criterion_GAN.to(device)
    criterion_pixelwise = criterion_pixelwise.to(device)

    generator.apply(weights_init_normal)
    discriminator.apply(weights_init_normal)
```

```
    optimizer_G = torch.optim.Adam(generator.parameters(), lr=opt.lr, betas=(opt.b1, opt.b
2))
    optimizer_D = torch.optim.Adam(discriminator.parameters(), lr=opt.lr, betas=(opt.b1, op
t.b2))

    transforms_aug = A.Compose([
        A.Resize(opt.img_height, opt.img_width),
        A.Normalize(),
        ToTensorV2()
    ])

    transforms_ori = A.Compose([
        A.Resize(opt.img_height, opt.img_width),
        A.Normalize(),
        ToTensorV2()
    ])

    dataloader = DataLoader(
        CDRL_Dataset(root_path=opt.root_path, dataset=opt.dataset_name, train_val='train',
                     transforms_A=transforms_aug, transforms_B=transforms_ori),
        batch_size=opt.batch_size,
        shuffle=True,
        num_workers=opt.n_cpu,
    )

    val_dataloader = DataLoader(
        CDRL_Dataset(root_path=opt.root_path, dataset=opt.dataset_name,  train_val='train',
                     transforms_A=transforms_ori, transforms_B=transforms_ori),
        batch_size=10,
        shuffle=False,
        num_workers=1,
    )

    prev_time = time.time()

    for epoch in range(opt.epoch, opt.n_epochs):
        for i, batch in enumerate(dataloader):
            img_A = batch["A"].to(device)
            img_B = batch["B"].to(device)

            valid = torch.ones((img_A.size(0), *patch), device=device)
            fake = torch.zeros((img_A.size(0), *patch), device=device)

            optimizer_G.zero_grad()
            gener_output = generator(img_A, img_B)
            gener_output_pred = discriminator(gener_output, img_A)

            loss_GAN = criterion_GAN(gener_output_pred, valid)
            loss_pixel = criterion_pixelwise(gener_output, img_A)
            loss_G = loss_GAN + lambda_pixel * loss_pixel
            loss_G.backward()
            optimizer_G.step()

            optimizer_D.zero_grad()
            pred_real = discriminator(img_B, img_A)
            loss_real = criterion_GAN(pred_real, valid)

            B_pred_fake = discriminator(gener_output.detach(), img_A)
            loss_fake = criterion_GAN(B_pred_fake, fake)

            loss_D = 0.5 * (loss_real + loss_fake)
            loss_D.backward()
            optimizer_D.step()

            batches_done = epoch * len(dataloader) + i
            batches_left = opt.n_epochs * len(dataloader) - batches_done
            time_left = datetime.timedelta(seconds=batches_left * (time.time() - prev_time))
            prev_time = time.time()

            sys.stdout.write(
                "\r[Epoch %d/%d] [Batch %d/%d] [D loss: %f] [G loss: %f, pixel: %f, adv: %f]
ETA: %s"
                % (
                    epoch,
```

```
                        opt.n_epochs,
                        i,
                        len(dataloader),
                        loss_D.item(),
                        loss_G.item(),
                        loss_pixel.item(),
                        loss_GAN.item(),
                        time_left,
                    )
                )

                if batches_done % opt.sample_interval == 0:
                    sample_images(val_dataloader, generator, opt, device, batches_done)

        torch.save(generator.state_dict(), "saved_models/%s/generator_%d.pth" % (opt.save_na
me, epoch))
        torch.save(discriminator.state_dict(), "saved_models/%s/discriminator_%d.pth" % (op
t.save_name, epoch))
```

- **Purpose**: This is the main training loop for a GAN model:
  - **Generator**: Creates fake images.
  - **Discriminator**: Classifies real vs. fake.
  - Trains both models in a loop, logs progress, saves images and model weights.
- **Key Features**:
  - Uses **two loss functions**:
    - `MSELoss`: To guide the discriminator.
    - `L1Loss`: To guide the generator for pixel-wise accuracy.
  - Saves the generated images (`sample_images`) and model weights at regular intervals.
- **Example**:
  - Trains for `10 epochs` and saves:
    - Images in the `images/levir/` directory.
    - Model weights in the `saved_models/levir/` directory.

```
import numpy as np

###################      metrics     ###################
class AverageMeter(object):
    """Computes and stores the average and current value"""
    def __init__(self):
        self.initialized = False
        self.val = None
        self.avg = None
        self.sum = None
        self.count = None

    def initialize(self, val, weight):
        self.val = val
        self.avg = val
        self.sum = val * weight
        self.count = weight
        self.initialized = True
```

```
    def update(self, val, weight=1):
        if not self.initialized:
            self.initialize(val, weight)
        else:
            self.add(val, weight)

    def add(self, val, weight):
        self.val = val
        self.sum += val * weight
        self.count += weight
        self.avg = self.sum / self.count

    def value(self):
        return self.val

    def average(self):
        return self.avg

    def get_scores(self):
        scores_dict = cm2score(self.sum)
        return scores_dict

    def clear(self):
        self.initialized = False


###################    cm metrics    ###################
class ConfuseMatrixMeter(AverageMeter):
    """Computes and stores the average and current value"""
    def __init__(self, n_class):
        super(ConfuseMatrixMeter, self).__init__()
        self.n_class = n_class

    def update_cm(self, pr, gt, weight=1):
        """获得当前混淆矩阵，并计算当前F1得分，并更新混淆矩阵"""
        val = get_confuse_matrix(num_classes=self.n_class, label_gts=gt, label_preds=pr)
#        print('!1',self.n_class)
#        print('!2',weight)
#        print('!',gt.shape)
#        print('!!',pr.shape)
#        print(gt)

        self.update(val, weight)
        current_score = cm2F1(val)
        return current_score

    def get_scores(self):
        scores_dict = cm2score(self.sum)
        return scores_dict
```

```
def harmonic_mean(xs):
    harmonic_mean = len(xs) / sum((x+1e-6)**-1 for x in xs)
    return harmonic_mean


def cm2F1(confusion_matrix):
    hist = confusion_matrix
    n_class = hist.shape[0]
    tp = np.diag(hist)
    sum_a1 = hist.sum(axis=1)
    sum_a0 = hist.sum(axis=0)
    # ---------------------------------------------------------------------- #
    # 1. Accuracy & Class Accuracy
    # ---------------------------------------------------------------------- #
    acc = tp.sum() / (hist.sum() + np.finfo(np.float32).eps)

    # recall
    recall = tp / (sum_a1 + np.finfo(np.float32).eps)
    # acc_cls = np.nanmean(recall)

    # precision
    precision = tp / (sum_a0 + np.finfo(np.float32).eps)

    # F1 score
    F1 = 2 * recall * precision / (recall + precision + np.finfo(np.float32).eps)
    mean_F1 = np.nanmean(F1)
    return mean_F1


def cm2score(confusion_matrix):
#     print(confusion_matrix.shape)
    hist = confusion_matrix
    n_class = hist.shape[0]
    tp = np.diag(hist)
    sum_a1 = hist.sum(axis=1)
    sum_a0 = hist.sum(axis=0)
    # ---------------------------------------------------------------------- #
    # 1. Accuracy & Class Accuracy
    # ---------------------------------------------------------------------- #
    acc = tp.sum() / (hist.sum() + np.finfo(np.float32).eps)

    # recall
    recall = tp / (sum_a1 + np.finfo(np.float32).eps)
    # acc_cls = np.nanmean(recall)

    # precision
    precision = tp / (sum_a0 + np.finfo(np.float32).eps)

    # F1 score
    F1 = 2*recall * precision / (recall + precision + np.finfo(np.float32).eps)
```

```
    mean_F1 = np.nanmean(F1)
    # ------------------------------------------------------------------ #
    # 2. Frequency weighted Accuracy & Mean IoU
    # ------------------------------------------------------------------ #
    iu = tp / (sum_a1 + hist.sum(axis=0) - tp + np.finfo(np.float32).eps)
    mean_iu = np.nanmean(iu)

    freq = sum_a1 / (hist.sum() + np.finfo(np.float32).eps)
    fwavacc = (freq[freq > 0] * iu[freq > 0]).sum()

    #
    cls_iou = dict(zip(['iou_'+str(i) for i in range(n_class)], iu))

    cls_precision = dict(zip(['precision_'+str(i) for i in range(n_class)], precision))
    cls_recall = dict(zip(['recall_'+str(i) for i in range(n_class)], recall))
    cls_F1 = dict(zip(['F1_'+str(i) for i in range(n_class)], F1))

    score_dict = {'acc': acc, 'miou': mean_iu, 'mf1':mean_F1}
    score_dict.update(cls_iou)
    score_dict.update(cls_F1)
    score_dict.update(cls_precision)
    score_dict.update(cls_recall)
    return score_dict


def get_confuse_matrix(num_classes, label_gts, label_preds):
    """计算一组预测的混淆矩阵"""
    def __fast_hist(label_gt, label_pred):
        """
        Collect values for Confusion Matrix
        For reference, please see: https://en.wikipedia.org/wiki/Confusion_matrix
        :param label_gt: <np.array> ground-truth
        :param label_pred: <np.array> prediction
        :return: <np.ndarray> values for confusion matrix
        """
        mask = (label_gt >= 0) & (label_gt < num_classes)
#         print(np.unique(label_gt))
#         print(label_gt.shape)
        hist = np.bincount(num_classes * label_gt[mask].astype(int) + label_pred[mask],
                           minlength=num_classes**2).reshape(num_classes, num_classes)
        return hist
    confusion_matrix = np.zeros((num_classes, num_classes))
    for lt, lp in zip(label_gts, label_preds):
        confusion_matrix += __fast_hist(lt.flatten(), lp.flatten())
    return confusion_matrix


def get_mIoU(num_classes, label_gts, label_preds):
    confusion_matrix = get_confuse_matrix(num_classes, label_gts, label_preds)
    score_dict = cm2score(confusion_matrix)
```

return score_dict['miou']
same like above explain rhe codes in simple words and a simple example

This code provides tools to measure and evaluate model performance using metrics like **accuracy**, **mean Intersection over Union (mIoU)**, **F1 score**, and others based on a **confusion matrix**. Below is a simple explanation of each part of the code with examples.

---

1. AverageMeter Class

- **Purpose**: Keeps track of a metric's current value, average, sum, and count.
- **How it works**:
  - Initialize the class with a starting value (`val`) and weight.
  - Use `update()` to add new values and calculate running averages.

Example:

```python
# Initialize AverageMeter
meter = AverageMeter()

# Update with new values
meter.update(val=2, weight=3)  # Add value 2 with weight 3
meter.update(val=4, weight=1)  # Add value 4 with weight 1

# Get the average
print("Average:", meter.average())  # Output: Average: 2.5
```

---

2. ConfuseMatrixMeter Class

- **Purpose**: Extends `AverageMeter` to calculate and store confusion matrix metrics like **F1 score**.
- **How it works**:
  - Accepts predictions (`pr`) and ground truths (`gt`) to compute a confusion matrix.
  - Updates the running confusion matrix with new data using `update_cm()`.
  - Computes metrics like F1 score using the confusion matrix.

Example:

```python
# Create a ConfuseMatrixMeter for 3 classes
cm_meter = ConfuseMatrixMeter(n_class=3)

# Fake predictions and ground truths
predictions = [np.array([[0, 1], [1, 2]])]
ground_truths = [np.array([[0, 1], [1, 1]])]

# Update confusion matrix
f1_score = cm_meter.update_cm(pr=predictions, gt=ground_truths)
print("F1 Score:", f1_score)  # Outputs F1 score based on predictions
```

3. Helper Functions

`harmonic_mean(xs)`

- **Purpose**: Computes the harmonic mean of a list of numbers.
- **Example**:

```python
print(harmonic_mean([1, 2, 4]))  # Output: 1.714
```

`cm2F1(confusion_matrix)`

- **Purpose**: Calculates the mean F1 score for all classes using a confusion matrix.
- **How it works**:
  - Extracts **True Positives (TP)**, **Precision**, and **Recall**.
  - Uses the formula:

$$F1 = 2 \times \frac{\text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}}$$

  - Computes the average F1 score.

Example:

```python
# Fake confusion matrix for 2 classes
cm = np.array([[5, 2], [1, 7]])

print("Mean F1 Score:", cm2F1(cm))  # Output: Mean F1 Score: 0.827 (example)
```

`cm2score(confusion_matrix)`

- **Purpose**: Computes various metrics like accuracy, mIoU, and F1 score using a confusion matrix.
- **How it works**:
  - Extracts TP, FP, and FN from the confusion matrix.
  - Calculates:
    - **Accuracy**
    - **Mean IoU**
    - **Precision, Recall, F1 Score** for each class.
    - Returns a dictionary with all metrics.

Example:

```python
# Fake confusion matrix
cm = np.array([[5, 2], [1, 7]])

scores = cm2score(cm)
print(scores)
# Output: {'acc': ..., 'miou': ..., 'mf1': ..., 'iou_0': ..., 'iou_1': ...}
```

---

4. `get_confuse_matrix()`

- **Purpose**: Computes the confusion matrix for predictions and ground truths.
- **How it works**:
  - Counts occurrences of predictions and ground truths for each class.
  - Returns a matrix where:
    - Rows represent ground truths.
    - Columns represent predictions.

Example:

```python
# Predictions and ground truths for 2 classes
predictions = [np.array([[0, 1], [1, 1]])]
ground_truths = [np.array([[0, 1], [1, 0]])]

cm = get_confuse_matrix(num_classes=2, label_gts=ground_truths, label_preds=predictions)
print(cm)  # Output: [[1, 1], [1, 1]]
```

---

5. `get_mIoU()`

- **Purpose**: Computes the mean Intersection over Union (mIoU) for predictions and ground truths.
- **How it works**:
  - Calls `get_confuse_matrix()` to compute confusion matrix.
  - Extracts IoU for each class and computes the mean.

Example:

```python
# Predictions and ground truths
predictions = [np.array([[0, 1], [1, 1]])]
ground_truths = [np.array([[0, 1], [1, 0]])]

miou = get_mIoU(num_classes=2, label_gts=ground_truths, label_preds=predictions)
print("mIoU:", miou)  # Output: mIoU based on predictions
```

Summary

OpenAI

This code provides robust tools for tracking and evaluating model performance using metrics like F1 score, mIoU, and accuracy. Each component can be easily customized and integrated into deep learning pipelines.

```
import torch.nn as nn
import torch.nn.functional as F
import torch
from cbam import *


def weights_init_normal(m):
    classname = m.__class__.__name__
    if classname.find("BasicConv") != -1:
        torch.nn.init.normal_(m.conv.weight.data, 0.0, 0.02)
        torch.nn.init.normal_(m.bn.weight.data, 1.0, 0.02)
        torch.nn.init.constant_(m.bn.bias.data, 0.0)
    elif classname.find("Conv") != -1:
        torch.nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find("BatchNorm2d") != -1:
        torch.nn.init.normal_(m.weight.data, 1.0, 0.02)
        torch.nn.init.constant_(m.bias.data, 0.0)


##############################
#      U-NET CBAM ver
##############################
class UNetDown(nn.Module):
    def __init__(self, in_size, out_size, normalize=True, dropout=0.0):
        super(UNetDown, self).__init__()
        layers = [nn.Conv2d(in_size, out_size, 4, 2, 1, bias=False)]
        if normalize:
            layers.append(nn.InstanceNorm2d(out_size))
        layers.append(nn.LeakyReLU(0.2))
        if dropout:
            layers.append(nn.Dropout(dropout))
        self.model = nn.Sequential(*layers)

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

class UNetUp_CBAM(nn.Module):
    def __init__(self, in_size, out_size, dropout=0.0):
        super(UNetUp_CBAM, self).__init__()
        layers = [
            nn.ConvTranspose2d(in_size, out_size, 4, 2, 1, bias=False),
            nn.InstanceNorm2d(out_size),
            nn.ReLU(inplace=True),
        ]
```

```
        if dropout:
            layers.append(nn.Dropout(dropout))

        self.model = nn.Sequential(*layers)
        self.ChannelGate = ChannelGate(out_size, 16, ['avg', 'max'])
        self.SpatialGate = SpatialGate()
    def forward(self, x, A, B):
        x = self.model(x)
        A = self.ChannelGate(A)
        B = self.SpatialGate(B)
        skip_input = A+B
        x = torch.cat((x, skip_input), 1)

        return x

class GeneratorUNet_CBAM(nn.Module):
    def __init__(self, in_channels=3, out_channels=3):
        super(GeneratorUNet_CBAM, self).__init__()

        self.down1 = UNetDown(in_channels, 64, normalize=False)
        self.down2 = UNetDown(64, 128)
        self.down3 = UNetDown(128, 256)
        self.down4 = UNetDown(256, 512, dropout=0.5)
        self.down5 = UNetDown(512, 512, dropout=0.5)
        self.down6 = UNetDown(512, 512, dropout=0.5)
        self.down7 = UNetDown(512, 512, dropout=0.5)
        self.down8 = UNetDown(512, 512, normalize=False, dropout=0.5)

        self.up1 = UNetUp_CBAM(512, 512, dropout=0.5)
        self.up2 = UNetUp_CBAM(1024, 512, dropout=0.5)
        self.up3 = UNetUp_CBAM(1024, 512, dropout=0.5)
        self.up4 = UNetUp_CBAM(1024, 512, dropout=0.5)
        self.up5 = UNetUp_CBAM(1024, 256)
        self.up6 = UNetUp_CBAM(512, 128)
        self.up7 = UNetUp_CBAM(256, 64)

        self.final = nn.Sequential(
            nn.Upsample(scale_factor=2),
            nn.ZeroPad2d((1, 0, 1, 0)),
            nn.Conv2d(128, out_channels, 4, padding=1),
            nn.Tanh(),
        )

    def forward(self, xt1, xt2):
        # Images Harmonization

        d1_xt1 = self.down1(xt1)
        d2_xt1 = self.down2(d1_xt1)
        d3_xt1 = self.down3(d2_xt1)
        d4_xt1 = self.down4(d3_xt1)
        d5_xt1 = self.down5(d4_xt1)
```

```
        d6_xt1 = self.down6(d5_xt1)
        d7_xt1 = self.down7(d6_xt1)
        d8_xt1 = self.down8(d7_xt1)

        d1_xt2 = self.down1(xt2)
        d2_xt2 = self.down2(d1_xt2)
        d3_xt2 = self.down3(d2_xt2)
        d4_xt2 = self.down4(d3_xt2)
        d5_xt2 = self.down5(d4_xt2)
        d6_xt2 = self.down6(d5_xt2)
        d7_xt2 = self.down7(d6_xt2)
        d8_xt2 = self.down8(d7_xt2)

        d8 = d8_xt1 + d8_xt2


        u1 = self.up1(d8, d7_xt1, d7_xt2)
        u2 = self.up2(u1, d6_xt1, d6_xt2)
        u3 = self.up3(u2, d5_xt1, d5_xt2)
        u4 = self.up4(u3, d4_xt1, d4_xt2)
        u5 = self.up5(u4, d3_xt1, d3_xt2)
        u6 = self.up6(u5, d2_xt1, d2_xt2)
        u7 = self.up7(u6, d1_xt1, d1_xt2)

        return self.final(u7)



##############################
#        Discriminator
##############################


class Discriminator(nn.Module):
    def __init__(self, in_channels=3):
        super(Discriminator, self).__init__()

        def discriminator_block(in_filters, out_filters, normalization=True):
            """Returns downsampling layers of each discriminator block"""
            layers = [nn.Conv2d(in_filters, out_filters, 4, stride=2, padding=1)]
            if normalization:
                layers.append(nn.InstanceNorm2d(out_filters))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            return layers

        self.model = nn.Sequential(
            *discriminator_block(in_channels * 2, 64, normalization=False),
            *discriminator_block(64, 128),
            *discriminator_block(128, 256),
            *discriminator_block(256, 512),
            nn.ZeroPad2d((1, 0, 1, 0)),
```

```
        nn.Conv2d(512, 1, 4, padding=1, bias=False)
    )

def forward(self, img_A, img_B):
    # Concatenate image and condition image by channels to produce input
    img_input = torch.cat((img_A, img_B), 1)
    return self.model(img_input)
```

same like above explain the function codes in simple words and a simpe example for each

This code implements a **U-Net-based generator** with attention (CBAM) and a **discriminator** for use in generative adversarial networks (GANs). Below is a breakdown of the classes and functions, explained in simple terms with examples.

---

1. `weights_init_normal()`

- **Purpose**: Initializes the weights of the model with a normal distribution.
- **How it works**:
  - For convolution layers (`Conv` or `BasicConv`), the weights are initialized with a mean of 0 and a standard deviation of 0.02.
  - For batch normalization layers, weights are initialized close to 1, and biases are set to 0.

Example:

```python
# Apply weights initialization to a model
model = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1)
weights_init_normal(model)
```

---

2. `UNetDown` Class

- **Purpose**: Creates a downsampling block for U-Net.
- **How it works**:
  - A convolution layer reduces the spatial size.
  - Instance normalization (optional) helps stabilize training.
  - Leaky ReLU is used as the activation function.
  - Dropout (optional) randomly deactivates some neurons to avoid overfitting.

Example:

```python
# Create a downsampling block
down_block = UNetDown(in_size=3, out_size=64)
```

```
    # Input tensor (e.g., image with 3 channels)
    input_tensor = torch.randn(1, 3, 128, 128)

    # Forward pass
    output = down_block(input_tensor)
    print(output.shape)  # Output shape will be reduced spatially
```

---

3. `UNetUp_CBAM` Class

- **Purpose**: Creates an upsampling block with CBAM (Channel and Spatial Attention).
- **How it works**:
  - A transposed convolution layer increases the spatial size.
  - Instance normalization and ReLU are applied.
  - **CBAM attention mechanism** adds channel and spatial attention to the skip connections.

Example:

```python
    # Create an upsampling block with CBAM
    up_block = UNetUp_CBAM(in_size=64, out_size=32)

    # Input tensor and skip connections
    x = torch.randn(1, 64, 16, 16)  # Input tensor
    A = torch.randn(1, 32, 16, 16)  # Channel attention input
    B = torch.randn(1, 32, 16, 16)  # Spatial attention input

    # Forward pass
    output = up_block(x, A, B)
    print(output.shape)  # Output shape increases spatially
```

---

4. `GeneratorUNet_CBAM` Class

- **Purpose**: Implements the full U-Net generator with CBAM.
- **How it works**:
  - Downsampling blocks compress input images (`down1` to `down8`).
  - Skip connections (from encoder to decoder) are used for details.
  - CBAM adds channel and spatial attention to skip connections.
  - Final layer upscales the output to match the input resolution and applies `tanh`.

Example:

```python
    # Create a U-Net generator
    generator = GeneratorUNet_CBAM(in_channels=3, out_channels=3)

    # Input tensors (e.g., two images)
    xt1 = torch.randn(1, 3, 256, 256)
    xt2 = torch.randn(1, 3, 256, 256)

    # Forward pass
```

```
output = generator(xt1, xt2)
print(output.shape)  # Output is the harmonized image
```

---

5. `Discriminator` Class

- **Purpose**: Implements a PatchGAN discriminator.
- **How it works**:
  - Takes two images (`img_A` and `img_B`) as input.
  - Concatenates them along the channel dimension.
  - Downsamples the input using convolutional layers to predict whether patches of the image are real or fake.

Example:

```python
# Create a discriminator
discriminator = Discriminator(in_channels=3)

# Input tensors (two images to compare)
img_A = torch.randn(1, 3, 256, 256)
img_B = torch.randn(1, 3, 256, 256)

# Forward pass
output = discriminator(img_A, img_B)
print(output.shape)  # Output is the patch-based prediction
```

---

6. How It All Fits Together

1. The **generator** takes two input images (`xt1` and `xt2`) and outputs a harmonized image.
2. The **discriminator** takes the output of the generator (harmonized image) and the original input (`xt1`) to classify patches as real or fake.

Key Notes:

- **CBAM (Convolutional Block Attention Module)**: Enhances feature maps by focusing on important regions (channels and spatial attention).
- **U-Net Structure**: A popular encoder-decoder architecture that combines high-level features and low-level details using skip connections.
- **PatchGAN Discriminator**: Operates at the patch level to classify small regions of the image as real or fake, making it suitable for image-to-image tasks like harmonization or super-resolution.

import argparse
import os
import numpy as np
import math
import itertools

```
import time
import datetime
import sys

import torch
import torch.nn as nn
from torchvision.utils import save_image
from torch.utils.data import DataLoader
from torch.autograd import Variable

from models import *
from datasets import *

import albumentations as A
from albumentations.pytorch.transforms import ToTensorV2
from torchvision import transforms
from torchvision.transforms.functional import to_pil_image
from video_processing import *

def get_device():
    if torch.cuda.is_available():
        return torch.device("cuda")
    elif hasattr(torch.backends, 'mps') and torch.backends.mps.is_available():
        return torch.device("mps")
    else:
        return torch.device("cpu")

def pixel_visual(gener_output_, A_ori_, name, save_name):
    # Move tensors to CPU for visualization processing
    gener_output = gener_output_.cpu().clone().detach().squeeze()
    A_ori = A_ori_.cpu().clone().detach().squeeze()

    pixel_loss = to_pil_image(torch.abs(gener_output - A_ori))
    trans = transforms.Compose([
        transforms.Grayscale(),
        transforms.ToTensor()])
    pixel_loss = trans(pixel_loss)

    thre_num = 0.7
    threshold = nn.Threshold(thre_num, 0.)
    pixel_loss = threshold(pixel_loss)
    save_image(pixel_loss, f'pixel_img/{save_name}/{str(name[0])}')
    save_image(gener_output.flip(-3), f'gener_img/{save_name}/{str(name[0])}', normalize=True)

def visualize_change_detection(img_A, img_B, gener_output, save_path):
    # Convert tensors to CPU and detach for visualization
    img_A = img_A.cpu().detach()
    img_B = img_B.cpu().detach()
    gener_output = gener_output.cpu().detach()

    # Create directory if it doesn't exist
```

```
    os.makedirs(save_path, exist_ok=True)

    # Save individual images
    save_image(img_A, f'{save_path}/input_t1.png', normalize=True)
    save_image(img_B, f'{save_path}/input_t2.png', normalize=True)
    save_image(gener_output, f'{save_path}/change_mask.png', normalize=True)

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--root_path", type=str, default="/workspace/NAS_MOUNT/", help="root
path")
    parser.add_argument("--dataset_name", type=str, default="LEVIR-CD", help="name of the
dataset")
    parser.add_argument("--save_name", type=str, default="levir", help="name of the dataset")
    parser.add_argument("--n_cpu", type=int, default=4, help="number of cpu threads to use during
batch generation")
    parser.add_argument("--img_height", type=int, default=256, help="size of image height")
    parser.add_argument("--img_width", type=int, default=256, help="size of image width")
    parser.add_argument('--save_visual', action='store_true', help='save pixel visualization map')
    parser.add_argument("--video_path_past", type=str, required=True, help="path to past video file")
    parser.add_argument("--video_path_present", type=str, required=True, help="path to present
video file")
    parser.add_argument("--frame_interval", type=int, default=1, help="extract every nth frame from
videos")
    opt = parser.parse_args()
    print(opt)

    # Print directory structure before processing
    print("\nChecking input paths...")
    print(f"Video path (past): {opt.video_path_past} (exists: {os.path.exists(opt.video_path_past)})")
    print(f"Video path (present): {opt.video_path_present} (exists:
{os.path.exists(opt.video_path_present)})")
    print(f"Root path: {opt.root_path} (exists: {os.path.exists(opt.root_path)})")

    # Create output directories if they don't exist
    os.makedirs(opt.root_path, exist_ok=True)
    os.makedirs(os.path.join(opt.root_path, 'A'), exist_ok=True)
    os.makedirs(os.path.join(opt.root_path, 'B'), exist_ok=True)

    # Check if frames already exist
    frames_a = glob(os.path.join(opt.root_path, 'A', '*.*'))
    frames_b = glob(os.path.join(opt.root_path, 'B', '*.*'))
    print(f"\nFound {len(frames_a)} frames in directory A")
    print(f"Found {len(frames_b)} frames in directory B")

    if len(frames_a) == 0 or len(frames_b) == 0:
        print("\nExtracting video frames...")
        try:
            num_frames = extract_frames_from_two_videos(
                opt.video_path_past,
                opt.video_path_present,
```

```
            opt.root_path,
            opt.frame_interval
        )
        print(f"Successfully extracted {num_frames} frame pairs")

        # Recount frames after extraction
        frames_a = glob(os.path.join(opt.root_path, 'A', '*.*'))
        frames_b = glob(os.path.join(opt.root_path, 'B', '*.*'))
        print(f"After extraction: Found {len(frames_a)} frames in directory A")
        print(f"After extraction: Found {len(frames_b)} frames in directory B")

        if len(frames_a) == 0 or len(frames_b) == 0:
            raise ValueError("No frames were extracted from the videos")

    except Exception as e:
        print(f"Error during frame extraction: {str(e)}")
        raise

# Set up device
device = get_device()
print(f"\nUsing device: {device}")

# Create output directories
os.makedirs(f'pixel_img/{opt.save_name}', exist_ok=True)
os.makedirs(f'gener_img/{opt.save_name}', exist_ok=True)
os.makedirs(f'results/{opt.save_name}', exist_ok=True)

# Initialize models and move to device
print("\nInitializing models...")
generator = GeneratorUNet_CBAM(in_channels=3).to(device)
discriminator = Discriminator().to(device)

# Load model weights
print("Loading model weights...")
generator.load_state_dict(torch.load(
    f"saved_models/{opt.save_name}/generator_9.pth",
    map_location=device
))
discriminator.load_state_dict(torch.load(
    f"saved_models/{opt.save_name}/discriminator_9.pth",
    map_location=device
))

transforms_ = A.Compose([
    A.Resize(opt.img_height, opt.img_width),
    A.Normalize(),
    ToTensorV2()
])

print("\nInitializing dataset...")
val_dataset = VideoDataset(opt.root_path, transforms=transforms_)
```

```
print(f"Dataset size: {len(val_dataset)} samples")

if len(val_dataset) == 0:
    raise ValueError("Dataset is empty! No images found to process.")

val_dataloader = DataLoader(
    val_dataset,
    batch_size=1,
    shuffle=False,
    num_workers=opt.n_cpu,
)

# Rest of your processing code...
generator.eval()
discriminator.eval()

criterion_GAN = torch.nn.MSELoss().to(device)
criterion_pixelwise = torch.nn.L1Loss().to(device)
lambda_pixel = 100
patch = (1, opt.img_height // 2 ** 4, opt.img_width // 2 ** 4)

prev_time = time.time()
loss_G_total = 0
total_batches = 0

with torch.no_grad():
    for i, batch in enumerate(val_dataloader):
        img_A = batch["A"].to(device)
        img_B = batch["B"].to(device)
        name = batch["NAME"]

        valid = torch.ones((img_A.size(0), *patch), device=device)

        gener_output = generator(img_A, img_B)
        gener_output_pred = discriminator(gener_output, img_A)

        if opt.save_visual:
            visualize_change_detection(
                img_A,
                img_B,
                gener_output,
                f'results/{opt.save_name}/{str(name[0])}'
            )

        loss_GAN = criterion_GAN(gener_output_pred, valid)
        loss_pixel = criterion_pixelwise(gener_output, img_B)
        loss_G = loss_GAN + lambda_pixel * loss_pixel

        print('---------------------------------------------------------------------------')
        print('name : ', name[0])
        print('loss_G : ', round(loss_G.item(), 4))
```

```
        loss_G_total += loss_G
        total_batches += 1

    if total_batches > 0:
        print('---------------------------total-----------------------------')
        print('loss_G_total : ', round((loss_G_total/total_batches).item(), 4))
    else:
        print('No batches were processed!')

if __name__ == '__main__':
    main()
```

Explain the function codes in super simple words and give an simple example for each

This script performs **change detection** using two videos (past and present) by utilizing a trained **U-Net CBAM generator** and a **discriminator**. Below is a simplified explanation of the functions and their purposes, along with easy examples for each.

---

1. `get_device()`

- **Purpose**: Checks the available hardware (GPU, Apple MPS, or CPU) and selects the appropriate device for running the model.
- **How it works**:
  - Checks if a GPU is available using `torch.cuda.is_available()`.
  - Falls back to Apple's MPS if on macOS and supported.
  - Defaults to CPU if no GPU or MPS is available.

Example:

```python
device = get_device()
print(device)  # Output could be "cuda", "mps", or "cpu"
```

---

2. `pixel_visual()`

- **Purpose**: Creates and saves a pixel-wise difference map between the generator's output and the original image.
- **How it works**:
  - Converts the difference between two images into a grayscale map.
  - Applies a threshold to highlight significant differences.
  - Saves the pixel difference map and generator output.

Example:

```python
gener_output = torch.rand(1, 3, 256, 256)  # Fake output image
A_ori = torch.rand(1, 3, 256, 256)         # Original image
pixel_visual(gener_output, A_ori, "example", "test_run")
```

---

3. `visualize_change_detection()`

- **Purpose**: Saves the input images and the generated output (change mask) for visual inspection.
- **How it works**:
  - Converts tensors into images.
  - Saves images to a specified directory.

Example:

```python
img_A = torch.rand(1, 3, 256, 256)         # Past image
img_B = torch.rand(1, 3, 256, 256)         # Present image
gener_output = torch.rand(1, 3, 256, 256) # Predicted change mask
visualize_change_detection(img_A, img_B, gener_output, "results/test_visualization")
```

---

4. `main()`

- **Purpose**: The main script to handle the workflow, including:
  - Parsing arguments.
  - Extracting frames from videos.
  - Loading models and datasets.
  - Running the generator and discriminator for change detection.
  - Saving results and calculating losses.

Key Steps in `main()`:

1. **Argument Parsing**: Sets parameters for the script, like paths, dataset details, and model settings.
2. **Frame Extraction**: Extracts frames from input videos (if not already extracted).
3. **Device Selection**: Chooses GPU, MPS, or CPU for running the model.
4. **Model Initialization**: Loads pre-trained generator and discriminator models.
5. **Dataset Loading**: Prepares data for evaluation using `VideoDataset`.
6. **Change Detection**:
   - Runs the generator to predict changes.
   - Uses the discriminator to evaluate the generated changes.
   - Saves results and computes losses.

Example:

```bash
python script.py --video_path_past past_video.mp4 --video_path_present present_video.mp4 --s
ave_visual
```

---

5. `extract_frames_from_two_videos()`

- **Purpose**: Extracts frames from two videos (past and present) and saves them in separate folders.
- **How it works**:
  - Reads frames at regular intervals (`frame_interval`) from both videos.
  - Saves frames into directories `A` (for past) and `B` (for present).

Example:

If `frame_interval=5`, it extracts every 5th frame:

```python
extract_frames_from_two_videos("past.mp4", "present.mp4", "/output_path", 5)
```

---

6. Dataset and DataLoader

- `VideoDataset`: Prepares image pairs (past and present) for processing.
- `DataLoader`: Loads the dataset in batches for efficient processing.

Example:

```python
dataset = VideoDataset(root_path="/data/", transforms=None)
dataloader = DataLoader(dataset, batch_size=1, shuffle=False)

for batch in dataloader:
    img_A = batch["A"]  # Past frame
    img_B = batch["B"]  # Present frame
    print(img_A.shape, img_B.shape)
```

---

7. Model Initialization

- **Purpose**: Loads the generator and discriminator models and moves them to the selected device.

Example:

```python
generator = GeneratorUNet_CBAM(in_channels=3).to(device)
```

```
discriminator = Discriminator().to(device)
```

---

8. Loss Calculation

- `criterion_GAN`: Measures how well the generator fools the discriminator.
- `criterion_pixelwise`: Measures pixel-wise differences between the generated and real images.
- **Combined Loss**: `loss_GAN + lambda_pixel * loss_pixel`.

Example:

```python
criterion_GAN = nn.MSELoss().to(device)
criterion_pixelwise = nn.L1Loss().to(device)

loss_GAN = criterion_GAN(gener_output_pred, valid)
loss_pixel = criterion_pixelwise(gener_output, img_B)
loss_G = loss_GAN + 100 * loss_pixel
print(loss_G.item())
```

---

9. Saving Outputs

- Results (e.g., pixel maps, visualizations) are saved to folders like:
  - `pixel_img/`
  - `gener_img/`
  - `results/`

Summary:

This script performs **frame-level change detection** by:

1. Extracting frames from two videos.
2. Using a trained U-Net CBAM generator to predict changes.
3. Calculating losses and visualizing results.

The examples can be modified to test individual components (e.g., pixel visualizations, loss calculations, or dataset loading). Let me know if you'd like help with a specific part!

import cv2
import os
import numpy as np
from glob import glob
import os
from glob import glob
import cv2
import torch

```
from torch.utils.data import Dataset

def extract_frames_from_two_videos(video_path_past, video_path_present, output_dir,
frame_interval=1):
    """
    Extract frames from two videos and save them in A and B directories.

    Args:
        video_path_past: Path to the past video file
        video_path_present: Path to the present video file
        output_dir: Base directory to save the extracted frames
        frame_interval: Extract every nth frame (default=1)
    """
    os.makedirs(os.path.join(output_dir, 'A'), exist_ok=True)
    os.makedirs(os.path.join(output_dir, 'B'), exist_ok=True)

    # Open both video files
    cap_past = cv2.VideoCapture(video_path_past)
    cap_present = cv2.VideoCapture(video_path_present)

    if not cap_past.isOpened() or not cap_present.isOpened():
        raise ValueError("Error opening one or both video files")

    frame_count = 0

    while True:
        ret_past, frame_past = cap_past.read()
        ret_present, frame_present = cap_present.read()

        if not ret_past or not ret_present:
            break

        if frame_count % frame_interval == 0:
            # Save past frame to A directory
            cv2.imwrite(os.path.join(output_dir, 'A', f'frame_{frame_count:06d}.jpg'), frame_past)
            # Save present frame to B directory
            cv2.imwrite(os.path.join(output_dir, 'B', f'frame_{frame_count:06d}.jpg'), frame_present)

        frame_count += 1

    cap_past.release()
    cap_present.release()
    return frame_count

class VideoDataset(Dataset):
    def __init__(self, root_path, transforms=None):
        """
        Dataset for video frame change detection.

        Args:
            root_path: Path containing A (previous frames) and B (current frames) directories
```

```
        transforms: Albumentations transforms to apply to the images
        """
        self.root_path = root_path
        self.transforms = transforms
        self.files = sorted(glob(os.path.join(root_path, 'A', '*.*')))

    def __getitem__(self, index):
        name = os.path.basename(self.files[index])

        # Read previous (A) and current (B) frames
        img_A = cv2.imread(self.files[index], cv2.IMREAD_COLOR)
        img_B = cv2.imread(self.files[index].replace('/A/', '/B/'), cv2.IMREAD_COLOR)

        if self.transforms:
            transformed_A = self.transforms(image=img_A)
            transformed_B = self.transforms(image=img_B)
            img_A = transformed_A["image"]
            img_B = transformed_B["image"]

        return {"A": img_A, "B": img_B, "NAME": name}

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

Explain the class and functions codes in simple words and a simple example like how thry work

This code has two main parts:

1. **A function (`extract_frames_from_two_videos`)** to extract frames from two videos and save them as images.
2. **A class (`VideoDataset`)** to load and process the saved frames for change detection.

Let's break it down with simple explanations and examples.

---

1. Function: `extract_frames_from_two_videos`

This function extracts frames from two video files (representing past and present scenes) and saves them as images in two separate folders: `A` (past frames) and `B` (present frames).

How It Works:

1. **Directories**: Creates two folders, `A` and `B`, inside the given `output_dir`.
2. **Read Videos**: Opens the two videos using OpenCV (`cv2.VideoCapture`).
3. **Extract Frames**: Reads frames from both videos at the same time.
4. **Save Frames**: Every `frame_interval` frames, it saves:
   - The frame from the **past video** into the `A` folder.
   - The frame from the **present video** into the `B` folder.
5. **Return**: The total number of frames processed.

Example Usage:

```python
# Paths to past and present videos
past_video = "past_video.mp4"
present_video = "present_video.mp4"
output_dir = "./output_frames"
frame_interval = 5

# Extract frames
num_frames = extract_frames_from_two_videos(past_video, present_video, output_dir, frame_int
erval)
print(f"Extracted {num_frames} frames from the videos!")
```

What Happens:

- If the `frame_interval` is 5, it saves every 5th frame from both videos.
- The extracted frames are saved as:
  - `output_frames/A/frame_000000.jpg` (from past video)
  - `output_frames/B/frame_000000.jpg` (from present video)

---

2. Class: `VideoDataset`

This class is a **PyTorch Dataset** to load and preprocess the extracted frames for machine learning models. It helps to:

- Pair frames from the `A` and `B` folders.
- Apply transformations (e.g., resizing, normalizing) to make images ready for the model.
- Return images as a dictionary (`{"A": ..., "B": ..., "NAME": ...}`).

Key Components:

1. `__init__` **(Initialization)**:
   - Takes the `root_path` where `A` and `B` folders are stored.
   - Stores transformations (like resizing, normalizing) if provided.
2. `__getitem__`:
   - Reads one frame pair (past and present) from the directories.
   - Applies transformations if specified.
   - Returns the past frame (`A`), present frame (`B`), and the frame name.
3. `__len__`:
   - Returns the total number of frame pairs.

Example Usage:

```python
from torchvision.transforms import ToTensor
```

```
# Create dataset
dataset = VideoDataset(root_path="./output_frames", transforms=ToTensor())

# Get the first frame pair
sample = dataset[0]
print(sample["NAME"])  # Name of the frame, e.g., "frame_000000.jpg"
print(sample["A"].shape)  # Past frame as a tensor
print(sample["B"].shape)  # Present frame as a tensor

# Total number of samples
print(len(dataset))  # e.g., 100 if 100 frame pairs are found
```

What Happens:

- Suppose `output_frames/A/frame_000000.jpg` and `output_frames/B/frame_000000.jpg` exist:
  - It reads these two frames.
  - Applies transformations (e.g., converts them to tensors or resizes them).
  - Returns:

```python
{
    "A": Tensor of past frame,
    "B": Tensor of present frame,
    "NAME": "frame_000000.jpg"
}
```

- This allows batch loading and easy iteration for machine learning tasks.

Summary of Workflow:

1. **Extract Frames**:
   - Use `extract_frames_from_two_videos()` to get images from videos.
2. **Load Dataset**:
   - Use `VideoDataset` to load these images into a PyTorch-compatible format.
3. **Process for Model**:
   - Use the dataset to train or test a change detection model.

This modular design makes it easy to switch between datasets or apply new preprocessing steps! Let me know if you'd like a deeper dive into any part.

```
import cv2
import os
import numpy as np
from glob import glob
from tqdm import tqdm

def create_comparison_video(results_dir, output_path, fps=0.5):
    """
    Create a video from frame folders containing input frames and change detection mask.
```

```
    Each frame will be shown for 2 seconds.

    Args:
        results_dir: Base directory containing frame_XXXXXX folders
        output_path: Path where the output video will be saved
        fps: Frames per second for output video (default=0.5 for 2 seconds per frame)
    """
    # Get all frame folders sorted numerically
    frame_folders = glob(os.path.join(results_dir, 'frame_*'))
    if not frame_folders:
        raise ValueError(f"No frame folders found in {results_dir}")

    # Sort folders numerically
    frame_folders = sorted(frame_folders,
                          key=lambda x: int(os.path.basename(x).split('_')[1].split('.')[0]))

    print(f"Found {len(frame_folders)} frame folders")
    print(f"First folder: {frame_folders[0]}")  # Debug print
    print(f"Each frame will be shown for {1/fps:.1f} seconds")

    # Read first frame to get dimensions
    first_frame_folder = frame_folders[0]
    first_t1 = cv2.imread(os.path.join(first_frame_folder, 'input_t1.png'))

    if first_t1 is None:
        print(f"Debug: Checking if file exists: {os.path.join(first_frame_folder, 'input_t1.png')}")
        print(f"Debug: Files in first folder: {os.listdir(first_frame_folder)}")
        raise ValueError(f"Could not read first frame from {os.path.join(first_frame_folder,
'input_t1.png')}")

    frame_height, frame_width = first_t1.shape[:2]

    # Calculate dimensions for the combined frame
    combined_width = frame_width * 3  # Three images side by side
    combined_height = frame_height

    # Create output directory if it doesn't exist
    os.makedirs(os.path.dirname(output_path), exist_ok=True)

    # Initialize video writer
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    out = cv2.VideoWriter(
        output_path,
        fourcc,
        fps,
        (combined_width, combined_height)
    )

    print("Creating video...")
    for folder in tqdm(frame_folders):
        # Define the paths for the three images
```

```
        t1_path = os.path.join(folder, 'input_t1.png')
        t2_path = os.path.join(folder, 'input_t2.png')
        mask_path = os.path.join(folder, 'change_mask.png')

        if not all(os.path.exists(p) for p in [t1_path, t2_path, mask_path]):
            print(f"Warning: Missing files in {folder}")
            print(f"Debug: Files in folder: {os.listdir(folder)}")
            continue

        t1_frame = cv2.imread(t1_path)
        t2_frame = cv2.imread(t2_path)
        mask_frame = cv2.imread(mask_path)

        if any(img is None for img in [t1_frame, t2_frame, mask_frame]):
            print(f"Warning: Could not read images from {folder}")
            continue

        # Create combined frame
        combined_frame = np.hstack([t1_frame, t2_frame, mask_frame])

        # Add text labels
        font = cv2.FONT_HERSHEY_SIMPLEX
        cv2.putText(combined_frame, 'Past Frame', (10, 30), font, 1, (255, 255, 255), 2)
        cv2.putText(combined_frame, 'Present Frame', (frame_width + 10, 30), font, 1, (255, 255, 255), 2)
        cv2.putText(combined_frame, 'Change Mask', (frame_width * 2 + 10, 30), font, 1, (255, 255, 255),
2)

        # Add frame number
        frame_num = os.path.basename(folder).split('_')[1].split('.')[0]
        cv2.putText(combined_frame, f'Frame: {int(frame_num)}', (10, combined_height - 20),
                    font, 0.8, (255, 255, 255), 2)

        # Write frame
        out.write(combined_frame)

    # Release video writer
    out.release()
    print(f"\nVideo saved to: {output_path}")
    print(f"Total video duration: {len(frame_folders)/fps:.1f} seconds")

if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser(description='Create video from frame results')
    parser.add_argument('--results_dir', type=str, required=True,
                        help='Directory containing frame_XXXXXX folders')
    parser.add_argument('--output_path', type=str, required=True,
                        help='Path where the output video will be saved')
    parser.add_argument('--fps', type=float, default=0.5,
                        help='Frames per second for output video (default=0.5 for 2 seconds per frame)')
```

args = parser.parse_args()

create_comparison_video(args.results_dir, args.output_path, args.fps)

explain each function codes in super simple words and a simple example for each

This script takes folders of images from past and present frames, along with a change detection mask, and creates a side-by-side comparison video.

---

1. `create_comparison_video` Function

This is the main function that:

1. Reads folders containing frames (`frame_XXXXXX`).
2. Combines three images side-by-side:
    - Past frame (T1).
    - Present frame (T2).
    - Change mask.
3. Adds labels to each image (e.g., "Past Frame", "Present Frame").
4. Saves the combined frames as a video.

Detailed Explanation:

1. **Finding Frame Folders**:
    - It looks for folders like `frame_000001`, `frame_000002`, etc., inside `results_dir`.
    - These folders should have:
        - `input_t1.png` (past frame).
        - `input_t2.png` (present frame).
        - `change_mask.png` (change detection result).
2. **Sorting and Validating**:
    - Folders are sorted to ensure frames are in the correct order.
    - It checks if all three images exist in each folder.
3. **Video Writer Initialization**:
    - Opens a video file (`output_path`) where the final video will be saved.
    - Defines video resolution based on the combined width of the three images.
4. **Combining Frames**:
    - Reads the three images.
    - Combines them horizontally using `np.hstack`.
    - Adds labels and frame numbers to each combined image.
5. **Saving the Video**:
    - Writes the combined images to the video file.
    - Saves the video when all frames are processed.
6. **FPS and Duration**:
    - `fps` determines how fast frames are shown (e.g., `0.5 fps` means 1 frame every 2 seconds).
    - Total video duration is calculated as `number of frames ÷ fps`.

Example Usage:

```bash
python create_comparison_video.py \
    --results_dir ./results/levir \
    --output_path ./comparison_video.mp4 \
    --fps 0.5
```

Example Workflow:

- **Input**:
  - Directory: `results/levir/`
  - Inside `results/levir`, folders like:

    ```bash
    results/levir/frame_000001/
        input_t1.png
        input_t2.png
        change_mask.png
    results/levir/frame_000002/
        input_t1.png
        input_t2.png
        change_mask.png
    ```

- **Output**:
  - Video file: `comparison_video.mp4`
  - Each frame in the video shows:
    - Side-by-side comparison of `input_t1.png`, `input_t2.png`, and `change_mask.png`.
    - Labels for each part and frame number.

What Happens in the Code:

1. **Combine Frames**: Each combined frame looks like this:

   ```markdown
   ---------------------------------------------
   | Past Frame    | Present Frame    | Change Mask |
   ---------------------------------------------
   ```

   For example, `frame_000001/` would create:

   ```
   input_t1.png + input_t2.png + change_mask.png
   ```

2. **Save as Video**: Writes all combined frames to a video with `fps=0.5`.

Debugging Tips:

- If any folder is missing files, it shows a warning with the folder name.
- If the first frame cannot be read, it lists available files for debugging.

---

2. Main Block (`if __name__ == "__main__":`)

This block allows the script to run as a standalone program when executed from the command line. It:

1. Parses arguments like:
    - `--results_dir` (where the input folders are).
    - `--output_path` (where the video will be saved).
    - `--fps` (frames per second).
2. Calls `create_comparison_video` with the provided arguments.

Simple Example (Pseudocode):

Here's a basic version of what this script does:

```python
folders = ["frame_000001", "frame_000002"]  # Example frame folders
for folder in folders:
    past = read_image(f"{folder}/input_t1.png")
    present = read_image(f"{folder}/input_t2.png")
    mask = read_image(f"{folder}/change_mask.png")

    combined = combine_images([past, present, mask])  # Side-by-side
    add_labels(combined)
    save_to_video(combined)
```

Let me know if you'd like more examples or a deeper dive into any part!

import streamlit as st
import torch
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import cv2
from PIL import Image
import os

import albumentations as A
from albumentations.pytorch.transforms import ToTensorV2

```
# Import custom modules
from models import GeneratorUNet_CBAM, Discriminator
from video_processing import VideoDataset
from test import main as test_main
from create_video import create_comparison_video

def get_device():
    if torch.cuda.is_available():
        return torch.device("cuda")
    elif hasattr(torch.backends, 'mps') and torch.backends.mps.is_available():
        return torch.device("mps")
    else:
        return torch.device("cpu")

def load_model(model_path, model_type='generator'):
    """
    Load the generator or discriminator model
    """
    device = get_device()

    if model_type == 'generator':
        model = GeneratorUNet_CBAM(in_channels=3).to(device)
    else:
        model = Discriminator().to(device)

    model.load_state_dict(torch.load(model_path, map_location=device))
    model.eval()
    return model

def analyze_change_detection_metrics(val_dataloader, generator, discriminator):
    """
    Compute detailed metrics for change detection
    """
    device = get_device()
    criterion_GAN = torch.nn.MSELoss().to(device)
    criterion_pixelwise = torch.nn.L1Loss().to(device)
    lambda_pixel = 100

    metrics = {
        'frame_names': [],
        'loss_G': [],
        'loss_pixel': [],
        'loss_GAN': [],
        'pixel_difference': []
    }

    with torch.no_grad():
        for batch in val_dataloader:
            img_A = batch["A"].to(device)
            img_B = batch["B"].to(device)
            name = batch["NAME"][0]
```

```
        valid = torch.ones((img_A.size(0), 1, 16, 16), device=device)

        gener_output = generator(img_A, img_B)
        gener_output_pred = discriminator(gener_output, img_A)

        loss_GAN = criterion_GAN(gener_output_pred, valid)
        loss_pixel = criterion_pixelwise(gener_output, img_A)
        loss_G = loss_GAN + lambda_pixel * loss_pixel

        pixel_diff = torch.mean(torch.abs(gener_output - img_A)).item()

        metrics['frame_names'].append(name)
        metrics['loss_G'].append(loss_G.item())
        metrics['loss_pixel'].append(loss_pixel.item())
        metrics['loss_GAN'].append(loss_GAN.item())
        metrics['pixel_difference'].append(pixel_diff)

    return pd.DataFrame(metrics)

def main():
    st.set_page_config(page_title="Change Detection Analysis", layout="wide")
    st.title("Change Detection Model Metrics & Visualization")

    # Sidebar Configuration
    st.sidebar.header("Model & Data Configuration")

    # Video Input
    past_video = st.sidebar.file_uploader("Upload Past Video", type=['mp4', 'avi'])
    present_video = st.sidebar.file_uploader("Upload Present Video", type=['mp4', 'avi'])

    # Model Configuration
    save_name = st.sidebar.selectbox("Dataset", ["levir", "other_datasets"])
    frame_interval = st.sidebar.slider("Frame Extraction Interval", 1, 10, 1)

    # Analysis Options
    st.sidebar.header("Analysis Options")
    show_metrics = st.sidebar.checkbox("Show Metrics Table")
    show_plots = st.sidebar.checkbox("Show Metrics Plots")
    # generate_video = st.sidebar.checkbox("Generate Comparison Video")

    if st.sidebar.button("Run Analysis"):
        if past_video and present_video:
            # Temporary file handling
            with open(os.path.join("/tmp", "past_video.mp4"), "wb") as f:
                f.write(past_video.getbuffer())
            with open(os.path.join("/tmp", "present_video.mp4"), "wb") as f:
                f.write(present_video.getbuffer())

            # Configure paths
            root_path = "/tmp/change_detection"
```

```
        os.makedirs(root_path, exist_ok=True)

        # Extract Frames
        st.write("Extracting Video Frames...")
        from video_processing import extract_frames_from_two_videos
        extract_frames_from_two_videos(
            "/tmp/past_video.mp4",
            "/tmp/present_video.mp4",
            root_path,
            frame_interval
        )

        # Prepare Transformations
        transforms_ = A.Compose([
            A.Resize(256, 256),
            A.Normalize(),
            ToTensorV2()
        ])

        # Create Dataset
        val_dataset = VideoDataset(root_path, transforms=transforms_)
        val_dataloader = torch.utils.data.DataLoader(
            val_dataset, batch_size=1, shuffle=False
        )

        # Load Models
        generator = load_model(f"saved_models/{save_name}/generator_9.pth")
        discriminator = load_model(f"saved_models/{save_name}/discriminator_9.pth",
'discriminator')

        # Analyze Metrics
        metrics_df = analyze_change_detection_metrics(val_dataloader, generator, discriminator)

        # Display Results
        if show_metrics:
            st.subheader("Metrics Table")
            st.dataframe(metrics_df)

        if show_plots:
            st.subheader("Metrics Visualization")
            col1, col2 = st.columns(2)

            with col1:
                fig, ax = plt.subplots()
                metrics_df.boxplot(column=['loss_G', 'loss_pixel', 'loss_GAN'])
                st.pyplot(fig)

            with col2:
                fig, ax = plt.subplots()
                sns.histplot(metrics_df['pixel_difference'], kde=True)
                st.pyplot(fig)
```

```
        # if generate_video:
        #     st.write("Generating Comparison Video...")
        #     video_output_path = os.path.join(root_path, "change_detection.mp4")
        #     create_comparison_video(root_path, video_output_path)

        #     with open(video_output_path, "rb") as video_file:
        #         st.video(video_file.read())

    else:
        st.error("Please upload both past and present videos")

if __name__ == "__main__":
    main()

same like above
```

Here's an explanation of the code in simpler terms with examples.

Purpose of the Code

This Streamlit application:

1. Takes two videos (past and present) as inputs.
2. Extracts frames from both videos for analysis.
3. Uses pre-trained models (Generator and Discriminator) to detect changes.
4. Displays detailed metrics and visualizations (e.g., loss values, pixel differences) based on the change detection.

---

1. `get_device` Function

Checks whether a GPU, MPS (Mac), or CPU is available for running the model and returns the appropriate device.

Example:

- If you have a GPU, it will return `cuda`.
- On a Mac with MPS, it will return `mps`.
- Otherwise, it defaults to `cpu`.

---

2. `load_model` Function

Loads a saved model (either a generator or discriminator) and prepares it for evaluation.

Parameters:

- `model_path`: Path to the saved model file.
- `model_type`: Type of model to load (`generator` or `discriminator`).

Example:

```python
generator = load_model("saved_models/levir/generator_9.pth")
discriminator = load_model("saved_models/levir/discriminator_9.pth", model_type='discriminat
or')
```

- Loads the models for the LEVIR dataset.

---

3. `analyze_change_detection_metrics` Function

Computes metrics (e.g., losses, pixel differences) for change detection using the generator and discriminator.

Steps:

1. Loops through the data loader containing paired frames (past and present).
2. Uses the generator to create predicted output based on the frames.
3. Computes:
    - **GAN Loss**: How well the generator fools the discriminator.
    - **Pixelwise Loss**: Difference between the generated and real images.
    - **Pixel Difference**: Average pixel difference between generated and real frames.
4. Collects these metrics in a DataFrame.

Example Output:

Frame Name	Loss_G	Loss_Pixel	Loss_GAN	Pixel Difference
frame_000001.jpg	0.210	0.140	0.070	0.120
frame_000002.jpg	0.250	0.180	0.070	0.150

---

4. Main Streamlit App (`main` function)

1. **Sidebar Configuration**:
    - **Upload Videos**: Lets the user upload past and present videos.
    - **Set Frame Interval**: Chooses how frequently to extract frames.
    - **Dataset Selection**: Chooses the dataset for pre-trained models (e.g., LEVIR).
    - **Toggle Metrics and Plots**: Options to display detailed metrics and visualizations.
2. **Run Analysis Button**: When clicked:
    - Extracts frames from the uploaded videos and stores them in folders (`/tmp/change_detection/A` and `/tmp/change_detection/B`).
    - Applies transformations to resize and normalize the frames.
    - Loads the generator and discriminator models.
    - Computes metrics and optionally generates visualizations.
3. **Metrics Table**: Displays a table of metrics like GAN loss, pixelwise loss, and pixel differences.
4. **Metrics Plots**:
    - **Box Plot**: Shows the distribution of losses.
    - **Histogram**: Displays the distribution of pixel differences.

How it Works - Example Flow

1. **Input**:
    - User uploads two videos: `` `past_video.mp4` `` and `` `present_video.mp4` ``.
    - Selects:
        - Frame extraction interval: e.g., every 2 frames.
        - Dataset: `` `levir` ``.
2. **Output**:
    - **Metrics Table**:

```lua
| Frame Name       | Loss_G | Loss_Pixel | Loss_GAN | Pixel Difference |
|------------------|--------|------------|----------|------------------|
| frame_000001.jpg | 0.210  | 0.140      | 0.070    | 0.120            |
| frame_000002.jpg | 0.250  | 0.180      | 0.070    | 0.150            |
```

    - **Plots**:
        - Box plot comparing Loss_G, Loss_Pixel, and Loss_GAN.
        - Histogram of pixel differences.
3. **Change Detection Video (Optional)**: Combines frames and masks into a video for visualization.

Additional Features

- **Reusable Functions**:
    - The `` `load_model` `` and `` `analyze_change_detection_metrics` `` functions are modular and can be used in other scripts.
- **Error Handling**:
    - Validates if both videos are uploaded.
    - Checks if files exist in expected directories.

Would you like detailed explanations or simplified examples of specific parts?

```
import unittest
import torch
import os
import tempfile
import numpy as np
from models import GeneratorUNet_CBAM
from video_processing import extract_frames_from_two_videos, VideoDataset
import cv2
from torchmetrics import JaccardIndex, Precision, Recall, F1Score
from torch.utils.data import DataLoader
import albumentations as A
from albumentations.pytorch.transforms import ToTensorV2
import pytest
```

```
# Global dictionary to store metrics
METRICS = {}

@pytest.fixture(autouse=True)
def run_around_tests():
    yield
    if METRICS:  # Only print if metrics have been collected
        print("\n=== Final Metrics Summary ===")
        for metric_name, value in METRICS.items():
            print(f"{metric_name}: {value:.4f}")
        print("==========================")

class TestVideoChangeDetection(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        # Set up device
        try:
            if torch.cuda.is_available():
                cls.device = torch.device("cuda")
            else:
                cls.device = torch.device("cpu")
        except:
            cls.device = torch.device("cpu")

        print(f"\nUsing device: {cls.device}")

        # Initialize model
        cls.generator = GeneratorUNet_CBAM(in_channels=3).to(cls.device)

        # Set up transforms
        cls.transforms = A.Compose([
            A.Resize(256, 256),
            A.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
            ToTensorV2()
        ])

    def setUp(self):
        self.temp_dir = tempfile.mkdtemp()

    def tearDown(self):
        import shutil
        shutil.rmtree(self.temp_dir)

    def test_model_initialization(self):
        """Test if model initializes correctly"""
        try:
            self.assertIsInstance(self.generator, GeneratorUNet_CBAM)
            self.assertTrue(hasattr(self.generator, 'down1'))
            print("\nModel initialization test passed")
        except Exception as e:
```

```
            print(f"\nModel initialization test failed: {str(e)}")
            raise

    def test_model_output_shape(self):
        """Test if model produces correct output shape"""
        try:
            test_input1 = torch.randn(1, 3, 256, 256).to(self.device)
            test_input2 = torch.randn(1, 3, 256, 256).to(self.device)
            with torch.no_grad():
                output = self.generator(test_input1, test_input2)
            self.assertEqual(output.shape, (1, 3, 256, 256))
            METRICS['output_channels'] = float(output.shape[1])
            print("\nModel output shape test passed")
        except Exception as e:
            print(f"\nModel output shape test failed: {str(e)}")
            raise

    def test_frame_extraction(self):
        """Test frame extraction functionality"""
        try:
            video1_path = os.path.join(self.temp_dir, "video1.mp4")
            video2_path = os.path.join(self.temp_dir, "video2.mp4")

            self._create_dummy_video(video1_path)
            self._create_dummy_video(video2_path)

            frames_t1_dir = os.path.join(self.temp_dir, "frames_t1")
            frames_t2_dir = os.path.join(self.temp_dir, "frames_t2")

            os.makedirs(frames_t1_dir, exist_ok=True)
            os.makedirs(frames_t2_dir, exist_ok=True)

            num_frames = extract_frames_from_two_videos(
                video1_path,
                video2_path,
                self.temp_dir,
                frame_interval=1
            )

            self.assertTrue(num_frames > 0)
            self.assertTrue(os.path.exists(frames_t1_dir))
            self.assertTrue(os.path.exists(frames_t2_dir))
            METRICS['num_frames_extracted'] = float(num_frames)
            print(f"\nFrame extraction test passed (Extracted {num_frames} frames)")
        except Exception as e:
            print(f"\nFrame extraction test failed: {str(e)}")
            raise

    def test_dataset_loading(self):
        """Test if dataset loads correctly"""
        try:
```

```
        frames_a_dir = os.path.join(self.temp_dir, "A")
        frames_b_dir = os.path.join(self.temp_dir, "B")
        os.makedirs(frames_a_dir, exist_ok=True)
        os.makedirs(frames_b_dir, exist_ok=True)

        dummy_image = np.random.randint(0, 255, (100, 100, 3), dtype=np.uint8)
        cv2.imwrite(os.path.join(frames_a_dir, "frame_000000.png"), dummy_image)
        cv2.imwrite(os.path.join(frames_b_dir, "frame_000000.png"), dummy_image)

        dataset = VideoDataset(
            root_path=self.temp_dir,
            transforms=self.transforms
        )

        self.assertEqual(len(dataset), 1)
        sample = dataset[0]
        self.assertIn("A", sample)
        self.assertIn("B", sample)
        self.assertEqual(sample["A"].shape, (3, 256, 256))
        METRICS['dataset_size'] = float(len(dataset))
        print("\nDataset loading test passed")
    except Exception as e:
        print(f"\nDataset loading test failed: {str(e)}")
        raise

def test_model_metrics(self):
    """Test if model metrics can be computed correctly"""
    try:
        # Create dummy predictions and targets
        pred = torch.randint(0, 2, (1, 3, 256, 256), dtype=torch.float32).to(self.device)
        target = torch.randint(0, 2, (1, 3, 256, 256), dtype=torch.float32).to(self.device)

        # Initialize metrics
        jaccard = JaccardIndex(task='multiclass', num_classes=2).to(self.device)
        precision = Precision(task='multiclass', num_classes=2).to(self.device)
        recall = Recall(task='multiclass', num_classes=2).to(self.device)
        f1 = F1Score(task='multiclass', num_classes=2).to(self.device)

        # Compute metrics
        j_score = jaccard(pred, target)
        p_score = precision(pred, target)
        r_score = recall(pred, target)
        f1_score = f1(pred, target)

        # Store metrics
        METRICS['jaccard_score'] = j_score.item()
        METRICS['precision_score'] = p_score.item()
        METRICS['recall_score'] = r_score.item()
        METRICS['f1_score'] = f1_score.item()

        print("\nMetric Results:")
```

```
            print(f"Jaccard Score: {j_score.item():.4f}")
            print(f"Precision Score: {p_score.item():.4f}")
            print(f"Recall Score: {r_score.item():.4f}")
            print(f"F1 Score: {f1_score.item():.4f}")

            print("Model metrics test passed")
        except Exception as e:
            print(f"\nModel metrics test failed: {str(e)}")
            raise

    def _create_dummy_video(self, path, frames=10, size=(100, 100)):
        """Helper method to create dummy video for testing"""
        try:
            fourcc = cv2.VideoWriter_fourcc(*'mp4v')
            out = cv2.VideoWriter(path, fourcc, 1.0, size)

            for _ in range(frames):
                frame = np.random.randint(0, 255, (*size, 3), dtype=np.uint8)
                out.write(frame)
            out.release()
        except Exception as e:
            print(f"\nError creating dummy video: {str(e)}")
            raise

    def test_dataloader_creation(self):
        """Test if DataLoader works correctly with the dataset"""
        try:
            frames_a_dir = os.path.join(self.temp_dir, "A")
            frames_b_dir = os.path.join(self.temp_dir, "B")
            os.makedirs(frames_a_dir, exist_ok=True)
            os.makedirs(frames_b_dir, exist_ok=True)

            for i in range(3):
                dummy_image = np.random.randint(0, 255, (100, 100, 3), dtype=np.uint8)
                cv2.imwrite(os.path.join(frames_a_dir, f"frame_{i:06d}.png"), dummy_image)
                cv2.imwrite(os.path.join(frames_b_dir, f"frame_{i:06d}.png"), dummy_image)

            dataset = VideoDataset(
                root_path=self.temp_dir,
                transforms=self.transforms
            )

            dataloader = DataLoader(
                dataset,
                batch_size=2,
                shuffle=True,
                num_workers=0
            )

            batch = next(iter(dataloader))
            self.assertIn("A", batch)
```

```
        self.assertIn("B", batch)
        self.assertEqual(batch["A"].shape[0], 2)
        METRICS['dataloader_batch_size'] = float(batch["A"].shape[0])
        print("\nDataLoader creation test passed")
    except Exception as e:
        print(f"\nDataLoader creation test failed: {str(e)}")
        raise

if __name__ == '__main__':
    pytest.main([__file__, '-v'])
```

first exaplain pytest in simple words and then explain the function codes in super simple words

What is Pytest?

Pytest is a testing framework in Python that makes it easy to write and run tests for your code. It's used to ensure your code behaves as expected by checking whether the output of your functions and classes matches what you expect.

Key Features of Pytest:

- **Easy to write tests**: Pytest allows you to write simple test functions and even supports complex scenarios.
- **Automatic discovery**: It can automatically discover tests in your project by looking for files that start with `test_` or end with `_test.py`.
- **Easy assertions**: You can check if conditions are true using simple `assert` statements.
- **Test fixtures**: Pytest can set up and tear down tests, making it easy to initialize resources (like temporary files or database connections) that your tests need.

Pytest in This Code

In this code:

- The tests are written using the `unittest` framework, but Pytest is being used to run the tests (`pytest.main([__file__, '-v'])` at the end).
- Pytest will automatically look for test methods in the class `TestVideoChangeDetection` and run them.
- It also handles test fixture (`@pytest.fixture`) and provides a way to run tests and print out detailed results.

Explanation of Functions in the Code

1. `run_around_tests` **(Fixture)**
    - **Purpose**: This is a pytest fixture that runs before and after every test.
    - **How it works**: Before a test runs, it sets up some environment, and after the test runs, it prints out the final metrics summary, if available.
    - **Simplified**: It ensures that after the tests, the metrics are printed to give a final summary of the model's performance.

2. `setUpClass`
    - **Purpose**: This is a class method that runs before any test starts.
    - **How it works**: It sets up the device (GPU or CPU) and initializes the model (`GeneratorUNet_CBAM`). It also sets up the transformations for image processing (resizing and normalizing images).
    - **Simplified**: Before testing begins, it prepares the machine (CPU/GPU) and model, ensuring everything is ready.
3. `setUp`
    - **Purpose**: This method runs before each individual test.
    - **How it works**: It creates a temporary directory where files can be stored during tests.
    - **Simplified**: It creates a clean place to store files for each test, so tests don't affect each other.
4. `tearDown`
    - **Purpose**: This method runs after each test ends.
    - **How it works**: It cleans up by deleting the temporary directory and all its contents.
    - **Simplified**: After each test, it removes any temporary files to keep things tidy.
5. `test_model_initialization`
    - **Purpose**: This test checks if the model initializes correctly.
    - **How it works**: It checks that the `GeneratorUNet_CBAM` model is created properly and has specific parts (`down1`).
    - **Simplified**: This test just makes sure the model is created correctly and has the expected components.
6. `test_model_output_shape`
    - **Purpose**: This test checks if the model's output has the correct shape.
    - **How it works**: It passes random input through the model and verifies the output size is as expected (e.g., `(1, 3, 256, 256)`).
    - **Simplified**: It checks that the model gives back the right-sized output when it gets an input.
7. `test_frame_extraction`
    - **Purpose**: This test verifies that frames are correctly extracted from videos.
    - **How it works**: It creates dummy videos, extracts frames using `extract_frames_from_two_videos()`, and checks if frames are saved correctly in separate folders.
    - **Simplified**: This test checks if the system correctly extracts frames from uploaded videos.
8. `test_dataset_loading`
    - **Purpose**: This test checks if the dataset (containing video frames) can be loaded correctly.
    - **How it works**: It creates dummy images, saves them as frames, and checks if the `VideoDataset` class correctly loads the frames and applies transformations.
    - **Simplified**: This test ensures the dataset works as expected by loading the images and applying the necessary changes (like resizing).
9. `test_model_metrics`
    - **Purpose**: This test checks if metrics like Jaccard Index, Precision, Recall, and F1 Score can be calculated correctly.
    - **How it works**: It generates random predictions and targets, computes the metrics, and stores the results.
    - **Simplified**: It verifies that the model can compute and store performance metrics, like Jaccard and F1 score.
10. `_create_dummy_video`
    - **Purpose**: This helper function creates a dummy video file with random frames for testing.

- **How it works**: It generates a video with a specified number of frames (e.g., 10), and writes random pixel data as frames.
- **Simplified**: This helper just creates a random video file to use in the tests.

11. `test_dataloader_creation`
    - **Purpose**: This test checks if the `DataLoader` works properly.
    - **How it works**: It creates dummy image frames, loads them into the `VideoDataset`, and checks if the DataLoader can create batches of images as expected.
    - **Simplified**: This test ensures the data loading process works, i.e., it correctly groups frames into batches for training.

In Simple Words:

- **Pytest** helps run and check if the code works as expected.
- **Tests**: The test functions check if different parts of the system, like the model, frame extraction, dataset, and data loader, are working properly. It ensures that:
    - The model is set up right.
    - It produces the right output.
    - It can calculate useful metrics (like Precision or F1 Score).
    - It can handle frame extraction from videos.
    - It can load and process data correctly.

After the tests: Pytest shows a summary of any collected metrics (like accuracy scores) to evaluate the system's performance.

Explain unet architecture in super simple words with a simple example and then explain cbam architecture in super simple words with a simple example and explain in simple words how they have been taken n my poroject based on all the above files

Explaining U-Net Architecture in Simple Words

U-Net is a deep learning model architecture designed for image segmentation tasks. Segmentation means dividing an image into different parts, each representing a particular object or region of interest. U-Net is particularly popular in medical image segmentation, but it's also used in tasks like change detection (which is your use case).

Key Features of U-Net:

1. **Encoder-Decoder Structure**:
    - U-Net has two main parts:
        - **Encoder**: This part gradually reduces the size of the image, capturing the important features (such as edges, shapes, etc.). It uses convolutional layers to extract features.
        - **Decoder**: After compressing the image and capturing the features, the decoder gradually reconstructs the image back to its original size while keeping important details from the encoder. It "upsamples" the image to restore resolution.
2. **Skip Connections**:

  - To avoid losing details during the downsampling (encoder) and upsampling (decoder) processes, U-Net uses **skip connections**. These connections directly pass important features from the encoder to the corresponding decoder layer. It ensures the output is rich in both high-level features (from downsampling) and fine details (from the skip connections).

Simple Example:

Imagine you have an image of a house, and you want to segment the house from the background (like a binary mask). Here's how U-Net works:

- **Encoder**: It first reduces the image into smaller versions (like zooming out) and tries to recognize shapes (such as walls, windows, and doors).
- **Skip Connections**: The details (like edges of the walls and windows) are kept and sent directly to the decoder to make sure they don't get lost.
- **Decoder**: It uses this information to reconstruct the image back to its original size, while maintaining the important features (like the walls and windows) and ignoring the background.

U-Net in Your Project:

In your project, **GeneratorUNet_CBAM** is the U-Net model, which is used to generate a new image that represents the difference (or change) between the two input frames (past and present videos). The model tries to highlight the changes between the frames, and U-Net helps to preserve fine details like edges during this process, which is crucial for detecting changes accurately.

Explaining CBAM Architecture in Simple Words

CBAM stands for **Convolutional Block Attention Module**, and it's an attention mechanism that can be added to a network like U-Net to help the model focus on the most important parts of an image. Attention mechanisms are like a "spotlight" that helps the model focus on relevant features and ignore irrelevant ones.

CBAM has two main types of attention:

1. **Channel Attention**:
   - This mechanism helps the model decide which features (channels) in the image are important. For example, in an image, certain features (like edges or textures) may be more important for the task at hand than others.
   - It uses a global average pooling operation and a fully connected layer to generate a weight for each channel, emphasizing the important channels and suppressing the less useful ones.
2. **Spatial Attention**:
   - This mechanism helps the model focus on **where** in the image the important features are located. For example, it might want to focus on the regions where changes are happening (like a door opening or a car moving) rather than the background.
   - It uses a convolutional operation to generate a spatial attention map, which assigns higher importance to certain spatial regions of the image.

Simple Example:

Imagine you're trying to find a moving car in a video frame. CBAM helps the model by:

- **Channel Attention**: It figures out which parts of the image (such as edges or colors) are more important for identifying a car and gives them more weight.
- **Spatial Attention**: It focuses on the area of the image where the car is, ignoring less important areas like the sky or the ground.

CBAM in Your Project:

In your project, CBAM is integrated with the U-Net architecture (hence `` `GeneratorUNet_CBAM` ``) to make the change detection model smarter. The **Channel Attention** helps the model focus on the most important features, such as edges of objects or regions where there's a significant change. The **Spatial Attention** helps it focus on the exact areas of the image where change has occurred (like areas where something has moved).

This combination allows your model to detect changes with more precision, because it's not only segmenting the image but also paying attention to the regions that matter most for change detection.

How U-Net and CBAM Are Used Together in Your Project

Your project uses the **GeneratorUNet_CBAM** model, which combines both **U-Net** and **CBAM**. Here's how they are used together:

1. **U-Net** (with encoder-decoder architecture) is responsible for extracting high-level features from the input frames (past and present video frames) and reconstructing the output, which highlights the changes between the two frames.
2. **CBAM** (Channel and Spatial Attention) is applied to enhance the model's focus on important features:
    - **Channel Attention** allows the model to focus on the key features that help in detecting changes.
    - **Spatial Attention** helps the model zoom in on the areas where changes have occurred, ensuring that small but important changes (like moving objects) are not missed.

This combination makes your model more efficient and accurate in detecting changes between frames from videos, as it enhances both feature extraction and the focus on relevant parts of the image.

In Summary:

- **U-Net** helps the model segment and reconstruct images, preserving key details.
- **CBAM** adds attention mechanisms to make the model focus on the most important features and regions.
- **Together in your project**, they allow the model to perform **change detection** by segmenting changes between two video frames and highlighting those areas, with a better focus on where and what exactly has changed.

based on all the above files given to you explain all the metrics available in the
def analyze_change_detection_metrics(val_dataloader, generator, discriminator):
    """
    Compute detailed metrics for change detection
    """
    device = get_device()

```
    criterion_GAN = torch.nn.MSELoss().to(device)
    criterion_pixelwise = torch.nn.L1Loss().to(device)
    lambda_pixel = 100

    metrics = {
        'frame_names': [],
        'loss_G': [],
        'loss_pixel': [],
        'loss_GAN': [],
        'pixel_difference': []
    }

    with torch.no_grad():
        for batch in val_dataloader:
            img_A = batch["A"].to(device)
            img_B = batch["B"].to(device)
            name = batch["NAME"][0]

            valid = torch.ones((img_A.size(0), 1, 16, 16), device=device)

            gener_output = generator(img_A, img_B)
            gener_output_pred = discriminator(gener_output, img_A)

            loss_GAN = criterion_GAN(gener_output_pred, valid)
            loss_pixel = criterion_pixelwise(gener_output, img_A)
            loss_G = loss_GAN + lambda_pixel * loss_pixel

            pixel_diff = torch.mean(torch.abs(gener_output - img_A)).item()

            metrics['frame_names'].append(name)
            metrics['loss_G'].append(loss_G.item())
            metrics['loss_pixel'].append(loss_pixel.item())
            metrics['loss_GAN'].append(loss_GAN.item())
            metrics['pixel_difference'].append(pixel_diff)

    return pd.DataFrame(metrics)
```

ie explain each metric in simple

In the function `analyze_change_detection_metrics`, several **metrics** are computed to assess the performance of the **change detection** model. Here's a breakdown of each metric in simple terms:

1. frame_names:

- **Purpose**: This is a list that stores the names of the frames (or the names associated with each input pair of frames). It's used to keep track of which frame pair the metrics correspond to.
- **Example**: If the input frames come from "frame_001" and "frame_002", this list will store these frame names for reference.

2. loss_G (Generator Loss):

- **Purpose**: This metric represents the total loss for the **Generator** model. It combines two types of losses:
  - **GAN Loss (loss_GAN)**: Measures how well the generator produces outputs that look realistic (fooling the discriminator).
  - **Pixel-wise Loss (loss_pixel)**: Measures how close the generated image is to the real image at the pixel level.
- **How it's calculated**:
  - `loss_G = loss_GAN + (lambda_pixel * loss_pixel)`
  - `lambda_pixel` is a weight used to balance the two types of losses. In this case, **100** is used to give more importance to the pixel-wise loss.
- **Why it's important**: This tells us how well the **Generator** is performing in generating outputs that are realistic (from a GAN perspective) and how close the generated images are to the ground truth (original image).

3. loss_pixel (Pixel-wise Loss):

- **Purpose**: This metric measures the **difference** between the generated output (from the generator) and the original frame (img_A) on a pixel-by-pixel basis.
- **How it's calculated**: It uses the **L1 loss**, which calculates the **absolute difference** between the pixels in the generated image and the ground truth image.
- **Why it's important**: It helps the model improve by minimizing the pixel-level difference between the generated image and the original, encouraging the model to generate more realistic outputs that closely match the original frames.

4. loss_GAN (GAN Loss):

- **Purpose**: This is the loss from the **Discriminator**'s perspective, indicating how well the **Generator** is at generating realistic outputs.
- **How it's calculated**: The **generator output** is passed through the **discriminator**, which decides if the output is "real" or "fake". The **MSE loss** (Mean Squared Error) is then calculated between the discriminator's prediction (how likely it thinks the output is real) and the target value, which is **1** (indicating "real" for a perfect discriminator).
- **Why it's important**: The GAN loss measures how well the **Generator** is fooling the **Discriminator**. Lower values of **loss_GAN** mean the generator is doing a better job at creating images that seem real to the discriminator.

5. pixel_difference:

- **Purpose**: This metric calculates the **average absolute pixel-wise difference** between the generated output and the original frame (img_A).
- **How it's calculated**: `pixel_diff = torch.mean(torch.abs(gener_output - img_A))`
  - The **absolute difference** between each pixel in the generated image and the original image is taken, and then the **mean** of these differences is computed.
- **Why it's important**: This metric provides an indication of the **overall difference** between the generated image and the original. A lower pixel difference means that the generated image is more similar to the original, which is important for tasks like change detection where accuracy in pixel matching is crucial.

Summary of Metrics in Simple Terms:

- **frame_names**: Tracks which frame pair each metric corresponds to.
- **loss_G**: A combined loss metric for the **Generator**, which includes how realistic the generated image is (from the discriminator) and how close it is to the original image (pixel-wise).
- **loss_pixel**: Measures the pixel-wise difference between the generated output and the original image, encouraging the generator to make more accurate predictions.
- **loss_GAN**: Measures how well the **Generator** is able to fool the **Discriminator**, i.e., how realistic the generated output looks.
- **pixel_difference**: Measures the average difference in pixel values between the generated output and the original image, indicating how similar the generated image is to the ground truth.

These metrics are essential in evaluating how well your **change detection model** is performing, specifically in terms of how realistic the generated changes are and how accurate the pixel-wise differences are between the generated image and the original.

Based on all the files given to you above explain my project in simple words and give me an basic workflow of my project like what is happening from the first step to last in super simple words

Project Overview:

Your project is focused on **change detection in videos** using **deep learning**. Specifically, you're using a **UNet-based generator model** and a **discriminator model** (as part of a **Generative Adversarial Network** or GAN) to identify changes between two videos. The goal is to automatically detect and highlight the differences (or changes) between the **past video** and the **present video** at the **frame level**.

You also have a user interface built with **Streamlit**, allowing users to upload two videos (a past and a present video), and then you analyze and visualize the changes between them using the models you trained.

Key Steps in Your Project:

1. **Video Upload**: The user uploads two videos — the **past video** and the **present video**. These videos are the input for the change detection model.
2. **Frame Extraction**: The videos are processed to extract frames from both the past and present videos. This is done at a configurable **frame interval**, allowing you to decide how often you want to extract frames (for example, every 1st, 2nd, 3rd frame, etc.).
3. **Data Preparation**: The extracted frames are then fed into a custom **Dataset class** (using PyTorch's `` `DataLoader` ``) where each pair of frames from the two videos are used as inputs for the model. The frames are also transformed (e.g., resized, normalized) to be ready for the model.
4. **Model Loading**: The **generator model** (a **UNet architecture with CBAM** attention modules) and the **discriminator model** are loaded. These models have been trained previously on similar data and are used to perform change detection.
5. **Change Detection**: The generator model takes the frames from the past and present videos as inputs and generates an output showing the predicted changes. The discriminator evaluates

whether these generated changes look "real" or "fake" (as part of the GAN structure).

6. **Metrics Calculation**: During the analysis, several **metrics** are calculated to evaluate the quality of the generated change detection:
   - **Generator Loss** (`loss_G`): Measures how well the generator is performing by combining GAN loss and pixel-wise loss.
   - **Pixel-wise Loss** (`loss_pixel`): Measures how close the generated output is to the original frame (at the pixel level).
   - **GAN Loss** (`loss_GAN`): Measures how realistic the generated output is from the discriminator's point of view.
   - **Pixel Difference**: Measures the average pixel-wise difference between the generated output and the original frame.

   These metrics give feedback on how well the change detection model is working.
7. **Results Visualization**: Once the analysis is done, the results are displayed to the user:
   - A **metrics table** shows the values of the loss functions for each frame.
   - **Plots** visualize the loss distributions and pixel differences.

   Optionally, you also have the ability to generate a **comparison video** showing the detected changes between the two videos (though this feature is currently commented out in the code).

Basic Workflow:

1. **User Uploads Videos**:
   - The user uploads a **past video** and a **present video** through the Streamlit app.
2. **Frames are Extracted**:
   - Frames are extracted from both videos at a user-defined interval (e.g., every 1st frame).
3. **Preprocessing**:
   - The frames are resized and normalized so they can be fed into the deep learning models.
4. **Model Inference**:
   - The pre-trained **generator model** takes pairs of frames and generates change detection outputs.
   - The **discriminator model** evaluates these outputs to determine if they look real or fake (in GAN terms).
5. **Metrics Calculation**:
   - **Loss values** are computed for each frame pair, including how realistic the changes are (GAN loss) and how similar the generated changes are to the original frames (pixel-wise loss).
   - **Pixel differences** are also calculated to indicate the magnitude of changes detected.
6. **Results Presentation**:
   - A table showing the metrics for each frame is displayed.
   - Plots of the loss values and pixel differences are generated for visual analysis.
7. **Optional Video Generation**:
   - A comparison video can be created to visually highlight the changes between the two videos, though this feature is currently not enabled.

Key Technologies and Models:

- **Generator (UNet + CBAM)**: The generator uses the UNet architecture, which is good for pixel-level predictions. It also integrates **CBAM (Convolutional Block Attention Module)**, which helps the

model focus on important areas of the image.
- **Discriminator (GAN)**: The discriminator evaluates the generator's output, helping to train the model in a **Generative Adversarial Network (GAN)** setup.
- **Change Detection**: The model detects changes between two video frames, generating a difference map.

Simple Example:

Imagine you have two videos:

1. **Past Video**: Shows a street with cars parked along the side.
2. **Present Video**: The same street, but now a car has moved.

Your system will:

- Extract frames from both videos.
- Use the **generator model** to highlight the changes (i.e., where the car was and where it is no longer).
- The **discriminator** will check if the generated output (the "difference map") is realistic.
- Finally, the **metrics** will tell you how well the model has identified the change and whether the generated output matches the real difference.

Summary:

- **Input**: Two videos (past and present).
- **Process**: Extract frames, run through deep learning models, calculate metrics.
- **Output**: A summary of the detected changes, including visual and numerical results.

This process automates the task of detecting and highlighting changes between two video clips, which can be useful in various applications like surveillance, environmental monitoring, and more.

like the entire process in simple words like ehat will happen in each step

Sure! Let me break down the entire process of your project in **simple steps**:

---

1. User Uploads Two Videos

- The user uploads two videos: one representing the **past video** (before changes) and one representing the **present video** (after changes).

---

2. Extracting Frames from the Videos

- **Frame Extraction**: The program takes frames (individual images) from both videos. For example, if a video is 10 seconds long and has 30 frames per second, the program might extract frames every 1 second (to get 10 frames).
- These frames are saved as image files so they can be processed by the model.

---

3. Preparing the Data

- **Resizing & Normalizing**: The extracted frames are resized to a smaller, consistent size (e.g., 256x256 pixels) so the model can handle them better. The pixel values are also **normalized** so the model can learn more efficiently.
- This helps the model "understand" the frames and look for changes later.

---

4. Model Processes the Frames

- **Generator Model**: The generator, based on **UNet** and **CBAM**, takes in pairs of frames (one from the past video and one from the present video) and tries to **find the difference** between them (i.e., where something has changed).
- **Discriminator Model**: The discriminator looks at the output from the generator and tries to figure out if the change detection output is **real** (correct) or **fake** (wrong).
- These models work together in a **Generative Adversarial Network (GAN)** setup, where the generator gets better by trying to fool the discriminator, and the discriminator gets better by trying to detect fake changes.

---

5. Calculating Metrics (Performance Measurement)

- The system calculates some **important metrics** to measure how well the models are performing:
  - **Loss Values**: These tell how well the generator and discriminator are doing.
  - **Pixel Difference**: This measures how much the generated output differs from the original frames at the pixel level.
  - These metrics help us know if the model is doing a good job at detecting changes.

---

6. Displaying Results

- **Metrics Table**: The system creates a table that shows the metrics for each frame pair, like how much the model "lost" in each calculation (i.e., how far the generated result was from the correct one).
- **Loss Plots**: Graphs showing the loss values over time are also shown, so you can see if the model is getting better with each frame.
- **Change Detection**: A **visual comparison** of the original and generated frames may be shown to highlight the detected changes.

---

7. Optional: Creating a Comparison Video

- If enabled, the system can take the frames with detected changes and create a **comparison video**, showing where the changes occurred between the two original videos.

Summary:

1. **Upload Videos**: User uploads two videos (past and present).

2. **Extract Frames**: Frames are taken from both videos.
3. **Data Prep**: Frames are resized and normalized for the model.
4. **Model Inference**: The generator detects changes between the frames, and the discriminator checks if they are correct.
5. **Metrics Calculation**: The system calculates how well the model did (using loss values and pixel differences).
6. **Show Results**: Results (metrics and comparison) are shown to the user.
7. **(Optional) Comparison Video**: A video highlighting the changes is created.

This entire process helps you automatically detect and show changes between two videos!