Here are concrete, surgical fixes to your code to (a) cut frame-to-frame flicker, (b) make “similar in → similar out”, and (c) harden training/inference for VFX. No theory—just changes.

1) Replace BatchNorm + transposed convs (reduces flicker)

Small batches + BN → instability. Also ConvTranspose2d can checkerboard.

```
# --- replace in your model ---
class ResidualBlock(nn.Module):
    def __init__(self, channels: int, dilation: int = 1, groups: int = 32):
        super().__init__()
        g = min(groups, channels)  # safe for low channel counts
        self.conv1 = nn.Conv2d(channels, channels, 3, padding=dilation,
dilation=dilation)
        self.gn1   = nn.GroupNorm(g, channels)
        self.act   = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(channels, channels, 3, padding=dilation,
dilation=dilation)
        self.gn2   = nn.GroupNorm(g, channels)

    def forward(self, x):
        y = self.act(self.gn1(self.conv1(x)))
        y = self.gn2(self.conv2(y))
        return self.act(x + y)

class HDRNet(nn.Module):
    def __init__(self, in_channels=3, out_channels=3, base_channels=64,
num_resblocks=8):
        super().__init__()
        self.enc1 = nn.Conv2d(in_channels, base_channels, 3, stride=1, padding=1)
        self.enc2 = nn.Conv2d(base_channels, base_channels*2, 3, stride=2, padding=1)
        self.enc3 = nn.Conv2d(base_channels*2, base_channels*4, 3, stride=2, padding=1)

        self.bottleneck = nn.Sequential(*[
            ResidualBlock(base_channels*4, dilation=(2 if i % 2 == 0 else 1))
            for i in range(num_resblocks)
        ])

        # upsample + conv instead of ConvTranspose2d
        self.up3 = nn.Upsample(scale_factor=2, mode="bilinear", align_corners=False)
        self.conv3 = nn.Conv2d(base_channels*4, base_channels*2, 3, padding=1)
        self.up2 = nn.Upsample(scale_factor=2, mode="bilinear", align_corners=False)
        self.conv2 = nn.Conv2d(base_channels*2, base_channels, 3, padding=1)
        self.dec1 = nn.Conv2d(base_channels, out_channels, 3, padding=1)

        self.out_act = nn.Softplus(beta=1.0)

    def forward(self, x):
        e1 = F.relu(self.enc1(x), inplace=True)
        e2 = F.relu(self.enc2(e1), inplace=True)
        e3 = F.relu(self.enc3(e2), inplace=True)
        b  = self.bottleneck(e3)
        d3 = F.relu(self.conv3(self.up3(b)), inplace=True)
        d2 = F.relu(self.conv2(self.up2(d3)), inplace=True)
        out = self.dec1(d2)
```

```
        return self.out_act(out)
```

2) AdamW + weight decay (stabilizes, reduces drift)

```
# in train()
optimizer = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=1e-4)
Add flag:
ap.add_argument("--wd", type=float, default=1e-4)
# ...
optimizer = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=args.wd)
3) Exposure/scene conditioning = “similar in → similar out”
Append global luminance stats as extra channels so the net locks onto exposure context.
# in HDRDataset.__getitem__ after ldr/hdr tensors created
def _stats_map(img_chw: torch.Tensor):
    # img_chw in linear [0..1], shape [3,H,W]
    y = 0.2126*img_chw[0]+0.7152*img_chw[1]+0.0722*img_chw[2]
    mean = y.mean().clamp(1e-6).item()
    p50  = y.median().clamp(1e-6).item()
    p95  = torch.quantile(y.view(-1), 0.95).clamp(1e-6).item()
    v = torch.tensor([math.log(mean), math.log(p50), math.log(p95)],
dtype=torch.float32)
    H, W = y.shape
    return v.view(3,1,1).expand(3,H,W)

stats_chw = _stats_map(ldr_t)
ldr_t = torch.cat([ldr_t, stats_chw], dim=0)   # now in_channels=6
And update the model’s in_channels=6.
Add CLI to toggle:
ap.add_argument("--exposure-cond", action="store_true")
# in dataset construction
cond = args.exposure_cond
# only concat if cond; otherwise leave as-is
```

4) Temporal consistency loss (single-GPU, no heavy deps)

Use OpenCV DIS optical flow to warp previous prediction to current frame and penalize log-space differences. Minimal code:

```
# --- new util (top-level) ---
def _flow_warp(prev_img: torch.Tensor, curr_img: torch.Tensor, prev_pred: torch.Tensor):
    """
    prev_img, curr_img: [B,3,H,W] linear in [0..1] (LDR input, not HDR)
    prev_pred: [B,3,H,W] predicted HDR (normalized)
    returns prev_pred warped into curr frame using CPU DIS flow
    """
    import cv2
    B,_,H,W = prev_img.shape
    out = []
    for b in range(B):
        a = prev_img[b].mean(0).cpu().numpy().astype(np.float32)  # grayscale
        c = curr_img[b].mean(0).cpu().numpy().astype(np.float32)
        dis = cv2.DISOpticalFlow_create(cv2.DISOPTICAL_FLOW_PRESET_MEDIUM)
```

```
        flow = dis.calc(a, c, None)  # HxW x2
        # build grid for torch.grid_sample (N,H,W,2) in [-1,1]
        fx = flow[...,0]; fy = flow[...,1]
        gx = (np.arange(W)[None,:].repeat(H,0) + fx) / max(W-1,1) * 2 - 1
        gy = (np.arange(H)[:,None].repeat(W,1) + fy) / max(H-1,1) * 2 - 1
        grid =
torch.from_numpy(np.stack([gx,gy],axis=-1)).float().unsqueeze(0).to(prev_pred.device)
        warped = F.grid_sample(prev_pred[b:b+1], grid, mode="bilinear",
padding_mode="border", align_corners=True)
        out.append(warped)
    return torch.cat(out, dim=0)

def temporal_loss(pred_curr, pred_prev_warped, w=1.0, eps=1e-6):
    # log consistency in HDR space (normalized)
    return w * F.l1_loss(torch.log1p(pred_curr+eps), torch.log1p(pred_prev_warped+eps))
Wire into training: keep a previous batch from same sequence if available. Easiest: create
a sampler that yields (prev, curr) pairs; but minimally:
Add dataset option --sequences that returns (ldr_prev, ldr_curr, hdr_curr) when it can,
else falls back to (ldr, hdr).
# --- new dataset variant (minimal diff) ---
class HDRSeqDataset(HDRDataset):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # assume filenames have frame numbers; build prev mapping
        self.idx_by_key = {}
        for i,(l,h) in enumerate(self.pairs):
            k = Path(l).stem
            self.idx_by_key[k] = i
        self.keys = sorted(self.idx_by_key.keys())

    def _prev_of(self, key):
        # naive: decrement trailing number; adjust to your plate naming
        import re
        m = re.search(r'(\d+)$', key)
        if not m: return None
        num = int(m.group(1))
        cand = re.sub(r'(\d+)$', f"{num-1:0{len(m.group(1))}d}", key)
        return self.idx_by_key.get(cand, None)

    def __getitem__(self, idx):
        ldr_t, hdr_t = super().__getitem__(idx)
        key = Path(self.pairs[idx][0]).stem
        j = self._prev_of(key)
        if j is None:
            # duplicate current as prev to keep shapes
            ldr_prev = ldr_t.clone()
        else:
            ldr_prev, _ = super().__getitem__(j)
        return (ldr_prev, ldr_t, hdr_t)
```

Hook into CLI:

```
ap.add_argument("--sequences", action="store_true", help="enable temporal pairs")
ap.add_argument("--temporal-w", type=float, default=0.2, help="weight for temporal
loss")
```

Use it:

```
dataset_train = (HDRSeqDataset if args.sequences else HDRDataset)(...)
# loaders unchanged
```

In the train loop (_epoch_pass), when sequences enabled:

```
for bi, batch in enumerate(iterator):
    if args.sequences:
        ldr_prev, ldr, hdr = batch
    else:
        ldr, hdr = batch
        ldr_prev = ldr  # fallback

    ldr = ldr.to(device, non_blocking=pin_mem)
    hdr = hdr.to(device, non_blocking=pin_mem)
    ldr_prev = ldr_prev.to(device, non_blocking=pin_mem)

    if channels_last:
        ldr = ldr.to(memory_format=torch.channels_last)
        hdr = hdr.to(memory_format=torch.channels_last)
        ldr_prev = ldr_prev.to(memory_format=torch.channels_last)

    with autocast(device_type=("cuda" if is_cuda else "cpu"), enabled=amp_enabled):
        pred = model(ldr)
        loss_main = hdr_loss(pred, hdr, alpha=alpha_loss)
        if args.sequences:
            with torch.no_grad():
                pred_prev = model(ldr_prev)
                pred_prev = pred_prev.detach()
            pred_prev_warp = _flow_warp(ldr_prev, ldr, pred_prev)
            loss_temp = temporal_loss(pred, pred_prev_warp, w=args.temporal_w)
        else:
            loss_temp = 0.0
        loss = loss_main + (loss_temp if isinstance(loss_temp,float) else loss_temp)
```

5) Inference: motion-compensated temporal smoothing (no retrain)

Add a stateful smoother you can call per frame:

```
class MCStabilizer:
    def __init__(self, alpha=0.6):  # blend current vs. warped prev
        self.prev_pred = None
        self.prev_ldr  = None
        self.alpha = alpha

    @torch.no_grad()
    def step(self, model, ldr_chw: torch.Tensor, device, scale_hdr=4.0):
        model.eval().to(device)
        x = ldr_chw.unsqueeze(0).to(device)
        with autocast("cuda", enabled=torch.cuda.is_available()):
            pred = model(x)[0]  # normalized HDR
```

```
        if self.prev_pred is not None:
            prev_warp = _flow_warp(self.prev_ldr.unsqueeze(0), x,
self.prev_pred.unsqueeze(0))[0]
            # blend in log space, safer for HDR
            p  = torch.log1p(pred).clamp_min(0)
            pw = torch.log1p(prev_warp).clamp_min(0)
            p_blend = self.alpha*p + (1-self.alpha)*pw
            pred = torch.expm1(p_blend).clamp_min(0)
        self.prev_pred = pred.detach()
        self.prev_ldr  = x.detach()
        return (pred * scale_hdr).float()  # [3,H,W]
```

Use it in a loop of frames; write EXR per frame as you already do.

6) Safer metric + validation preview (sequence-aware)

Evaluate temporal stability (quick scalar you can log):

```
@torch.no_grad()
def temporal_std(model, loader, device, max_batches=2):
    model.eval().to(device)
    vals = []
    prev = None
    for i,(ldr,hdr) in enumerate(loader):
        if i>=max_batches: break
        y = model(ldr.to(device)).float().cpu()
        if prev is not None:
            vals.append(float((torch.log1p(y)-torch.log1p(prev)).abs().mean()))
        prev = y
    return (sum(vals)/len(vals)) if vals else float("nan")
```

Log it next to PSNR/SSIM:

```
tstd = temporal_std(model, val_loader, device, max_batches=3)
metrics_logger.log(epoch, psnr_val, ssim_val)
if rc and rc.tb:
    rc.tb.add_scalar("metrics/temporal_std_log", tstd, epoch)
```

7) Minor but impactful

- **Clamp HDR targets** during training to a sane high percentile to avoid exploding losses from rare hot pixels (keep EXR scene-linear but clip at, say, P99.9 per-image, once, in dataset). Add --hdr-hi-pct 99.9.

```
ap.add_argument("--hdr-hi-pct", type=float, default=None)
# in _read_hdr end:
if self.scale_hdr and self.allow_non_exr is not None:
    pass
if getattr(self, "hdr_hi_pct", None):
    hi = np.percentile(arr, self.hdr_hi_pct)
```

```
    if np.isfinite(hi) and hi>0:
        arr = np.clip(arr, 0, hi)
```

Store self.hdr_hi_pct = kwargs.get("hdr_hi_pct", None) in constructor.

- **Determinism**: you already expose flags; for sequences, prefer --deterministic.

8) Optional “luminance stacking” as augmentation (single-image)

You can simulate bracketed inputs as extra channels (fast, local):

```
def _luminance_stack(ldr_chw):
    y = 0.2126*ldr_chw[0]+0.7152*ldr_chw[1]+0.0722*ldr_chw[2]
    # 3 synthetic exposures in linear domain
    exp = [0.5, 1.0, 2.0]
    stacks = [(y*e).clamp(0,1) for e in exp]
    return torch.stack(stacks, dim=0)  # [3,H,W]

# in dataset __getitem__:
if getattr(self, "lum_stack", False):
    y3 = _luminance_stack(ldr_t)
    ldr_t = torch.cat([ldr_t, y3], dim=0)  # +3 channels

# update model in_channels accordingly if used together with exposure-cond
```

Add --luminance-stack flag. This helps the net learn monotonic expansions and improves consistency in saturated regions.

TL;DR — What to change now

1. Swap **BatchNorm→GroupNorm** and **ConvTranspose→Upsample+Conv** (copy/paste block above).
2. Use **AdamW(weight_decay=1e-4)**.
3. Enable **exposure conditioning** (extra stat channels) so “similar in → similar out”.
4. Add **temporal loss** with light optical-flow warp (OpenCV DIS) and --temporal-w 0.2.
5. Use **motion-compensated stabilizer** at inference if you can’t retrain.
6. Log a **temporal stability scalar** (temporal_std).
7. (Optional) **Luminance stacking** as 3 extra channels.
8. (Optional) Clip HDR targets at **P99.9** during training to avoid loss spikes.

These edits are minimal, self-contained, and aimed squarely at your two pain points: temporal flicker and exposure-consistency—without changing your data model or EXR outputs.