In [144…

```
import os, math, time, json
from pathlib import Path
from dataclasses import dataclass
import cv2 as cv

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from typing import Tuple, List, Optional, Dict
import glob


import matplotlib.pyplot as plt

IMG_DIR = "coco_images"
TARGET_SIZE = (320, 240)
PATCH = 64
MAX_JITTER = 16
BINS = 21
DTYPE = np.float32
LR      = 1e-4
STEPS   = 50000
BATCH   = 4
EPOCH_SAMPLES = 5000
SEED    = 123
CKPT_DIR = "checkpoints"
CKPT_PATH_REG = os.path.join(CKPT_DIR, "homography_reg.pt")
CKPT_PATH_CLS = os.path.join(CKPT_DIR, "homography_cls.pt")
ALPHA = 0.5

torch.manual_seed(SEED)
np.random.seed(SEED)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Device:", device)
```

```
Device: cpu
```

In [ ]:

```
def _to_gray_uchar(img_bgr: np.ndarray) -> np.ndarray:
    if img_bgr.ndim == 3 and img_bgr.shape[2] == 3:
        g = cv.cvtColor(img_bgr, cv.COLOR_BGR2GRAY)
    elif img_bgr.ndim == 2:
        g = img_bgr
    else:
        raise ValueError("Unsupported image format")
    return g

def _load_and_preprocess(path: str, target_size: Tuple[int, int]) -> np.ndarra
    img = cv.imread(path, cv.IMREAD_COLOR)
    if img is None:
        raise IOError(f"Failed to read image: {path}")
    img = cv.resize(img, target_size, interpolation=cv.INTER_AREA)
```

```python
    g = _to_gray_uchar(img)
    return g

def _random_window(rng: np.random.Generator, W: int, H: int, patch: int, margi
    x = rng.integers(margin, W - patch - margin + 1)
    y = rng.integers(margin, H - patch - margin + 1)
    return int(x), int(y)

def _perturb_corners(rng: np.random.Generator, x: int, y: int, patch: int, max
    tl = np.array([x,         y        ], dtype=np.float32)
    tr = np.array([x + patch, y        ], dtype=np.float32)
    br = np.array([x + patch, y + patch], dtype=np.float32)
    bl = np.array([x,         y + patch], dtype=np.float32)
    pts_src = np.stack([tl, tr, br, bl], axis=0)

    jit = rng.integers(-max_jitter, max_jitter + 1, size=(4, 2)).astype(np.flo
    pts_dst = pts_src + jit
    offsets = (pts_dst - pts_src).reshape(-1)
    return pts_src, pts_dst, offsets


def _safe_margin_for_jitter(max_jitter: int) -> int:
    return max_jitter


def _crop(gray: np.ndarray, x: int, y: int, patch: int) -> np.ndarray:
    return gray[y:y+patch, x:x+patch]

def _stack_two_channel(a: np.ndarray, b: np.ndarray) -> np.ndarray:
    a = (a.astype(DTYPE) / 255.0)
    b = (b.astype(DTYPE) / 255.0)
    return np.stack([a, b], axis=0)

def _bin_centers(bins: int, max_jitter: int) -> np.ndarray:
    return np.linspace(-max_jitter, max_jitter, bins, dtype=DTYPE)

def quantize_offsets(offsets_8: np.ndarray, bins: int = BINS, max_jitter: int
    centers = _bin_centers(bins, max_jitter)
    idxs = []
    for v in offsets_8.astype(DTYPE):
        i = int(np.argmin(np.abs(centers - v)))
        idxs.append(i)
    return np.array(idxs, dtype=np.int64)

def one_hot_8xB(class_idxs_8: np.ndarray, bins: int = BINS) -> np.ndarray:
    y = np.zeros((8, bins), dtype=DTYPE)
    for i, c in enumerate(class_idxs_8):
        y[i, c] = 1.0
    return y
def generate_sample_from_image(gray_240x320: np.ndarray,
                               rng: Optional[np.random.Generator] = None,
                               patch: int = PATCH,
                               max_jitter: int = MAX_JITTER,
```

```python
                              return_cls: bool = True) -> Dict[str, np.ndarra
    if rng is None:
        rng = np.random.default_rng()

    H_img, W_img = gray_240x320.shape[:2]
    margin = _safe_margin_for_jitter(max_jitter)
    x, y = _random_window(rng, W_img, H_img, patch, margin)

    pts_src, pts_dst, offsets = _perturb_corners(rng, x, y, patch, max_jitter)
    H = cv.getPerspectiveTransform(pts_src.astype(np.float32), pts_dst.astype(
    H_inv = np.linalg.inv(H)
    warped = cv.warpPerspective(gray_240x320, H_inv, (W_img, H_img), flags=cv.

    crop_a = _crop(gray_240x320, x, y, patch)
    crop_b = _crop(warped,       x, y, patch)
    x2 = _stack_two_channel(crop_a, crop_b)

    out = {
        "x": x2,
        "y_reg": offsets.astype(DTYPE),
        "H": H.astype(DTYPE),
        "xy": np.array([x, y], dtype=np.int32),
    }
    if return_cls:
        idxs = quantize_offsets(offsets, bins=BINS, max_jitter=max_jitter)
        out["y_cls_idx"] = idxs
        out["y_cls_oh"] = one_hot_8xB(idxs, bins=BINS)
    return out

def image_paths_from_dir(img_dir: str, exts: Tuple[str, ...] = (".jpg", ".jpeg
    ps = []
    for e in exts:
        ps.extend(glob.glob(os.path.join(img_dir, f"*{e}")))
    return sorted(ps)

class HomographyDatasetOnTheFly:
    def __init__(self,
                 img_dir: str = IMG_DIR,
                 seed: Optional[int] = None,
                 patch: int = PATCH,
                 max_jitter: int = MAX_JITTER,
                 return_cls: bool = True):
        self.paths = image_paths_from_dir(img_dir)
        if not self.paths:
            raise RuntimeError(f"No images found in {img_dir}")
        self.rng = np.random.default_rng(seed)
        self.patch = patch
        self.max_jitter = max_jitter
        self.return_cls = return_cls

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

```
    def _rand_img(self) -> np.ndarray:
        p = self.paths[self.rng.integers(0, len(self.paths))]
        g = _load_and_preprocess(p, TARGET_SIZE)
        return g

    def sample(self) -> Dict[str, np.ndarray]:
        g = self._rand_img()
        return generate_sample_from_image(
            g, rng=self.rng, patch=self.patch, max_jitter=self.max_jitter, ret
        )

def load_model(head_type, ckpt_path, bins=BINS, device=device):
    model = HomographyNet(head_type=head_type, bins=bins).to(device)
    state = torch.load(ckpt_path, map_location=device)
    model.load_state_dict(state["model"])
    model.eval()
    print(f"Loaded {head_type} model from: {ckpt_path}")
    return model

def sample_with_full(ds, return_cls=True):
    gray = ds._rand_img()

    H_img, W_img = gray.shape[:2]
    patch = ds.patch
    max_jitter = ds.max_jitter

    margin = max_jitter
    rng = ds.rng
    x, y = _random_window(rng, W_img, H_img, patch, margin)
    pts_src, pts_dst, offsets = _perturb_corners(rng, x, y, patch, max_jitter)

    H = cv.getPerspectiveTransform(pts_src.astype(np.float32), pts_dst.astype(
    H_inv = np.linalg.inv(H)
    warped = cv.warpPerspective(gray, H_inv, (W_img, H_img), flags=cv.INTER_LI

    crop_a = _crop(gray,   x, y, patch)
    crop_b = _crop(warped, x, y, patch)
    x2 = _stack_two_channel(crop_a, crop_b)

    out = {
        "x": x2,
        "y_reg": offsets.astype(DTYPE),
        "H": H.astype(DTYPE),
        "xy": np.array([x, y], dtype=np.int32),
        "gray_full": gray,
        "warped_full": warped
    }
    if return_cls:
        idxs = quantize_offsets(offsets, bins=BINS, max_jitter=max_jitter)
        out["y_cls_idx"] = idxs
        out["y_cls_oh"]  = one_hot_8xB(idxs, bins=BINS)
    return out
```

```
def quad_from_xy_offsets(xy_tl, patch, offsets8):
    x, y = map(int, xy_tl)
    src = np.array([[x, y],[x+patch, y],[x+patch, y+patch],[x, y+patch]], dtyp
    dst = src + offsets8.reshape(4,2).astype(np.float32)
    return src, dst

def checkerboard_blend(a, b, tiles=8):
    h,w = a.shape
    tile_h, tile_w = h//tiles, w//tiles
    out = np.zeros_like(a)
    for i in range(tiles):
        for j in range(tiles):
            ys,ye = i*tile_h, (i+1)*tile_h if i<tiles-1 else h
            xs,xe = j*tile_w, (j+1)*tile_w if j<tiles-1 else w
            if (i+j)%2==0: out[ys:ye, xs:xe] = a[ys:ye, xs:xe]
            else:          out[ys:ye, xs:xe] = b[ys:ye, xs:xe]
    return out

def draw_homography_overlay(gray_full, x, y, patch, offsets_pred, offsets_gt=N
    Himg, Wimg = gray_full.shape[:2]
    img = cv.cvtColor(gray_full, cv.COLOR_GRAY2BGR)

    src = np.array([[x, y],
                    [x+patch, y],
                    [x+patch, y+patch],
                    [x, y+patch]], dtype=np.float32)

    dst_pred = src + offsets_pred.reshape(4,2).astype(np.float32)

    cv.polylines(img, [src.astype(np.int32)], isClosed=True, color=(255,255,25

    cv.polylines(img, [dst_pred.astype(np.int32)], isClosed=True, color=(0,0,2

    if offsets_gt is not None:
        dst_gt = src + offsets_gt.reshape(4,2).astype(np.float32)
        cv.polylines(img, [dst_gt.astype(np.int32)], isClosed=True, color=(0,2

        for i in range(4):
            p1 = tuple(dst_gt[i].astype(int))
            p2 = tuple(dst_pred[i].astype(int))
            cv.line(img, p1, p2, (0,200,200), 1, lineType=cv.LINE_AA)

    plt.figure(figsize=(6,4))
    plt.imshow(cv.cvtColor(img, cv.COLOR_BGR2RGB))
    plt.title(title or "Homography overlay (RED=pred, GREEN=GT, white=patch)")
    plt.axis("off")
    plt.show()

def warp_full_with_pred(gray_full, x, y, patch, offsets_pred):
    src = np.array([[x, y],
                    [x+patch, y],
                    [x+patch, y+patch],
                    [x, y+patch]], dtype=np.float32)
```

```
    dst = src + offsets_pred.reshape(4,2).astype(np.float32)

    H = cv.getPerspectiveTransform(src, dst)
    H_inv = np.linalg.inv(H)

    h, w = gray_full.shape[:2]
    aligned = cv.warpPerspective(gray_full, H_inv, (w, h), flags=cv.INTER_LINE
    return aligned

def _overlay_square_quad(gray_full, x, y, patch, offsets8,
                         color_square=(255,255,255), color_quad=(0,255,0)):
    img_bgr = cv.cvtColor(gray_full, cv.COLOR_GRAY2BGR)

    src = np.array([[x, y],
                    [x+patch, y],
                    [x+patch, y+patch],
                    [x, y+patch]], dtype=np.float32)

    dst = src + offsets8.reshape(4,2).astype(np.float32)

    cv.polylines(img_bgr, [src.astype(np.int32)], isClosed=True, color=color_s
    cv.polylines(img_bgr, [dst.astype(np.int32)], isClosed=True, color=color_q

    return cv.cvtColor(img_bgr, cv.COLOR_BGR2RGB)

def quick_view(ds):
    s = sample_with_full(ds, return_cls=True)
    x0, y0 = map(int, s["xy"])
    offsets = s["y_reg"]

    orig_full   = s["gray_full"]
    curved_full = s["warped_full"]
    overlay_rgb = _overlay_square_quad(orig_full, x0, y0, PATCH, offsets)

    plt.figure(figsize=(12, 4))
    plt.subplot(1,3,1); plt.imshow(orig_full, cmap="gray");   plt.title("Origi
    plt.subplot(1,3,2); plt.imshow(curved_full, cmap="gray"); plt.title("Curve
    plt.subplot(1,3,3); plt.imshow(overlay_rgb);              plt.title("Squar
    plt.tight_layout()
    plt.show()

def _classic_offsets_from_full(gray_full, warped_full, x, y, patch, nfeatures=
    orb = cv.ORB_create(nfeatures=nfeatures)
    kp1, des1 = orb.detectAndCompute(gray_full, None)
    kp2, des2 = orb.detectAndCompute(warped_full, None)
    if des1 is None or des2 is None or len(kp1) < 4 or len(kp2) < 4:
        return None
    bf = cv.BFMatcher(cv.NORM_HAMMING)
    matches = bf.knnMatch(des1, des2, k=2)
    good = [m for m, n in matches if m.distance < 0.75 * n.distance]
    if len(good) < 4:
        return None
    src_pts = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1,1,2)
```

```
    dst_pts = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1,1,2)
    H_g2w, _ = cv.findHomography(src_pts, dst_pts, cv.RANSAC, 3.0)
    if H_g2w is None:
        return None
    try:
        H_pred = np.linalg.inv(H_g2w).astype(np.float32)
    except np.linalg.LinAlgError:
        return None

    src = np.array([[x, y],
                    [x+patch, y],
                    [x+patch, y+patch],
                    [x, y+patch]], dtype=np.float32).reshape(-1,1,2)
    dst = cv.perspectiveTransform(src, H_pred).reshape(-1,2).astype(np.float32
    return (dst - src.reshape(-1,2)).reshape(-1)


def test_homography_head(head="cls",
                         ckpt_reg="checkpoints_google/homography_reg_best.pt",
                         ckpt_cls="checkpoints_google/homography_cls_best.pt",
                         ds=None,
                         classic=False):
    base = ds if ds is not None else base_eval
    ckpt = ckpt_reg if head == "reg" else ckpt_cls

    s_full = sample_with_full(base, return_cls=True)
    x_np      = s_full["x"]
    y_reg_np  = s_full["y_reg"]
    x0, y0    = map(int, s_full["xy"])
    gray_full = s_full["gray_full"]
    warped_gt = s_full["warped_full"]
    patch_sz  = base.patch

    # --- NN (unchanged) ---
    model = load_model(head, ckpt, bins=BINS, device=device)
    model.eval()

    with torch.no_grad():
        xt = torch.from_numpy(x_np).unsqueeze(0).float().to(device)
        if head == "reg":
            pred = model(xt)
            pred_offsets_nn = pred[0].cpu().numpy()
        else:
            logits = model(xt)
            probs  = torch.softmax(logits, dim=-1)
            centers = torch.linspace(-base.max_jitter, base.max_jitter, BINS,
            pred_offsets_nn = (probs * centers).sum(dim=-1)[0].cpu().numpy()

    rmse_nn = float(torch.sqrt(F.mse_loss(
        torch.from_numpy(pred_offsets_nn).float(),
        torch.from_numpy(y_reg_np).float()
    )))
    print("NN RMSE (pixels):  ", rmse_nn)
```

```
    pred_offsets_cl = None
    if classic:
        pred_offsets_cl = _classic_offsets_from_full(gray_full, warped_gt, x0,
        if pred_offsets_cl is None:
            print("Classic: failed to estimate H — using identity (zeros).")
            pred_offsets_cl = np.zeros_like(y_reg_np)

    if not classic:
        draw_homography_overlay(
            gray_full, x0, y0, patch_sz,
            offsets_pred=pred_offsets_nn,
            offsets_gt=y_reg_np,
            title=f"Full-image homography — RED=NN pred, GREEN=GT (head={head}
        )
        aligned_pred = warp_full_with_pred(gray_full, x0, y0, patch_sz, pred_o
        fig, axs = plt.subplots(1, 3, figsize=(12, 4))
        axs[0].imshow(gray_full, cmap="gray"); axs[0].set_title("Original full
        axs[1].imshow(warped_gt, cmap="gray"); axs[1].set_title("GT warped (H^
        axs[2].imshow(aligned_pred, cmap="gray"); axs[2].set_title("NN warped
        plt.suptitle(f"Full-image alignment using NN ({head})")
        plt.show()
    else:
        aligned_nn = warp_full_with_pred(gray_full, x0, y0, patch_sz, pred_off
        aligned_cl = warp_full_with_pred(gray_full, x0, y0, patch_sz, pred_off

        img = cv.cvtColor(gray_full, cv.COLOR_GRAY2BGR)
        src = np.array([
            [x0, y0],
            [x0+patch_sz, y0],
            [x0+patch_sz, y0+patch_sz],
            [x0, y0+patch_sz]
        ], dtype=np.float32)

        cv.polylines(img, [src.astype(np.int32)], True, (255,255,255), 1, cv.L

        for off, col, thick in [
            (y_reg_np, (0,200,0), 1),
            (pred_offsets_nn, (0,0,255), 1),
            (pred_offsets_cl, (255,0,0), 1)
        ]:
            dst = (src + off.reshape(4,2).astype(np.float32)).astype(np.int32)
            cv.polylines(img, [dst], True, col, thick, cv.LINE_AA)

        overlay_rgb = cv.cvtColor(img, cv.COLOR_BGR2RGB)

        plt.figure(figsize=(6,4))
        plt.imshow(overlay_rgb)
        plt.title("Overlay — GT(green), NN(red), Classic(blue), Square(white)"
        plt.axis("off")
        plt.tight_layout()
        plt.show()
```

```python
    return {
        "gt_offsets": y_reg_np,
        "nn_offsets": pred_offsets_nn,
        "classic_offsets": pred_offsets_cl,
        "gray_full": gray_full,
        "warped_gt": warped_gt,
        "xy": (x0, y0),
        "patch": patch_sz,
        "rmse_nn": rmse_nn
    }
```

In [38]:
```python
class TorchHomographyDataset(Dataset):
    def __init__(self, base_ds, n_samples_per_epoch=5000, for_classification=T
        self.ds = base_ds
        self.n = int(n_samples_per_epoch)
        self.for_classification = for_classification

    def __len__(self):
        return self.n

    def __getitem__(self, idx):
        s = self.ds.sample()
        x = torch.from_numpy(s["x"]).float()
        y_reg = torch.from_numpy(s["y_reg"]).float()

        if self.for_classification:
            y_idx = torch.from_numpy(s["y_cls_idx"]).long()
            return {"x": x, "y_reg": y_reg, "y_cls_idx": y_idx}
        else:
            return {"x": x, "y_reg": y_reg}
```

In [80]:
```python
class BasicResBlock(nn.Module):
    def __init__(self, in_ch: int, out_ch: int):
        super().__init__()
        self.conv1 = nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1, bias=F
        self.bn1   = nn.BatchNorm2d(out_ch)
        self.conv2 = nn.Conv2d(out_ch, out_ch, kernel_size=3, padding=1, bias=
        self.bn2   = nn.BatchNorm2d(out_ch)

        self.proj = None
        if in_ch != out_ch:
            self.proj = nn.Conv2d(in_ch, out_ch, kernel_size=1, bias=False)
            self.proj_bn = nn.BatchNorm2d(out_ch)

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = F.relu(out, inplace=True)

        out = self.conv2(out)
        out = self.bn2(out)
```

```python
        if self.proj is not None:
            identity = self.proj_bn(self.proj(identity))

        out = out + identity
        out = F.relu(out, inplace=True)
        return out


class HomographyBackbone(nn.Module):
    def __init__(self, in_ch=2):
        super().__init__()
        self.stem = nn.Conv2d(in_ch, 64, kernel_size=3, padding=1, bias=False)

        self.b1a = BasicResBlock(64, 64)
        self.b1b = BasicResBlock(64, 64)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.b2a = BasicResBlock(64, 64)
        self.b2b = BasicResBlock(64, 64)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.b3a = BasicResBlock(64, 128)
        self.b3b = BasicResBlock(128, 128)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.b4a = BasicResBlock(128, 128)
        self.b4b = BasicResBlock(128, 128)

        self.flatten = nn.Flatten()
        self.fc = nn.Linear(128 * 8 * 8, 512)

    def forward(self, x):
        x = self.stem(x)

        x = self.b1a(x)
        x = self.b1b(x)
        x = self.pool1(x)

        x = self.b2a(x)
        x = self.b2b(x)
        x = self.pool2(x)

        x = self.b3a(x)
        x = self.b3b(x)
        x = self.pool3(x)

        x = self.b4a(x)
        x = self.b4b(x)

        x = self.flatten(x)
        x = self.fc(x)
        x = F.relu(x, inplace=True)
```

```python
        return x


class RegressionHead(nn.Module):
    def __init__(self, in_ch=512, max_jitter=16.0):
        super().__init__()
        self.fc = nn.Linear(in_ch, 8)
        self.max = max_jitter
    def forward(self, f):
        return torch.tanh(self.fc(f)) * self.max


class ClassificationHead(nn.Module):
    def __init__(self, in_ch=512, bins=21):
        super().__init__()
        self.bins = bins
        self.out = nn.Linear(in_ch, 8 * bins)

    def forward(self, f):
        logits = self.out(f)
        return logits.view(-1, 8, self.bins)

class HomographyNet(nn.Module):
    def __init__(self, head_type: str = "reg", bins: int = 21):
        super().__init__()
        assert head_type in ("reg", "cls")
        self.backbone = HomographyBackbone(in_ch=2)
        if head_type == "reg":
            self.head = RegressionHead(512)
        else:
            self.head = ClassificationHead(512, bins=bins)
        self.head_type = head_type

    def forward(self, x):
        f = self.backbone(x)
        return self.head(f)
```

In [81]:

```python
@dataclass
class TrainStepResult:
    loss: float
    rmse: float

def loss_regression(pred_8, y_reg, use_huber=True, delta=1.0):
    if use_huber:
        try:
            loss = F.smooth_l1_loss(pred_8, y_reg, beta=delta, reduction="mear
        except TypeError:
            loss = F.huber_loss(pred_8, y_reg, delta=delta, reduction="mean")
    else:
        loss = F.mse_loss(pred_8, y_reg, reduction="mean")

    rmse = torch.sqrt(F.mse_loss(pred_8, y_reg, reduction="mean"))
    return loss, rmse
```

```python
def loss_classification(logits_Bx8xK, y_idx_Bx8):
    B, eight, K = logits_Bx8xK.shape
    logits = logits_Bx8xK.reshape(B * eight, K)
    targets = y_idx_Bx8.reshape(B * eight)
    ce = F.cross_entropy(logits, targets, label_smoothing=0.05)
    with torch.no_grad():
        probs = F.softmax(logits_Bx8xK, dim=-1)
        idx_centers = torch.linspace(0, K - 1, K, device=logits_Bx8xK.device)
        pred_idx = (probs * idx_centers).sum(dim=-1)  # (B,8)
        pred_norm = 2 * (pred_idx / (K - 1)) - 1
        tgt_norm = 2 * (y_idx_Bx8.float() / (K - 1)) - 1
        rmse_proxy = torch.sqrt(F.mse_loss(pred_norm, tgt_norm, reduction="mea
    return ce, rmse_proxy
```

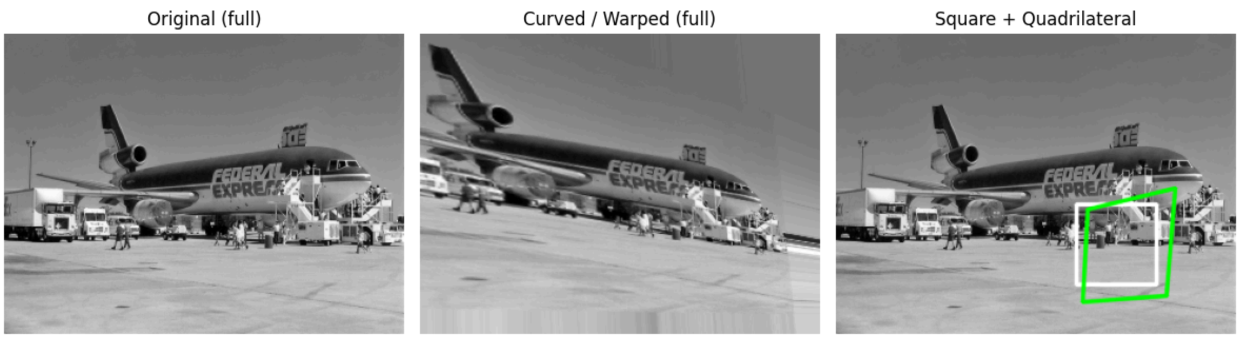
In [130…

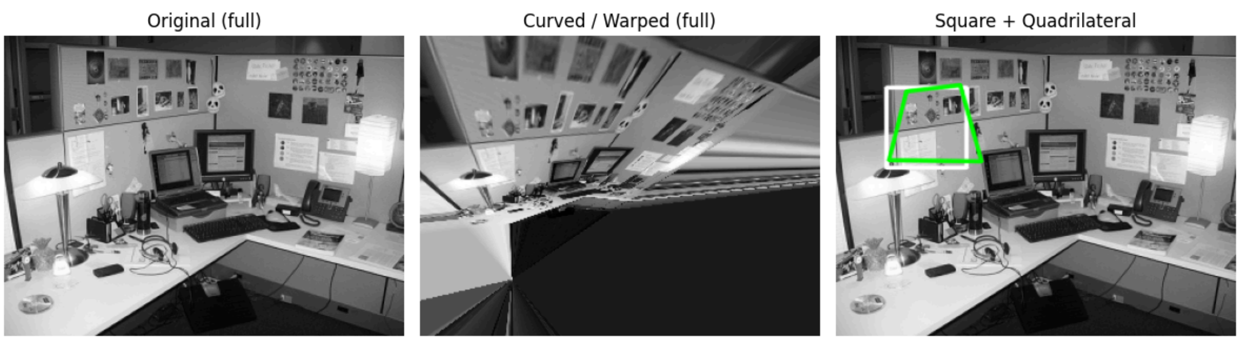
```python
def save_checkpoint(path, model, opt, step, best=False):
    os.makedirs(os.path.dirname(path) or ".", exist_ok=True)
    state = {"step": step, "model": model.state_dict(), "opt": opt.state_dict(
    torch.save(state, path)
    if best:
        torch.save(state, os.path.splitext(path)[0] + "_best.pt")

def train_loop(model, loader, head_type, optimizer, device, steps, log_every=2
    model.train()
    for m in model.modules():
        if isinstance(m, nn.BatchNorm2d):
            m.eval()
    step = 0
    best_loss = float("inf")
    t0 = time.time()

    log_steps, log_loss, log_rmse = [], [], []

    while step < steps:
        for batch in loader:
            x = batch["x"].to(device)
            y_reg = batch["y_reg"].to(device)
            if head_type == "cls":
                y_idx = batch["y_cls_idx"].to(device)

            optimizer.zero_grad(set_to_none=True)
            out = model(x)

            if head_type == "reg":
                loss, rmse = loss_regression(out, y_reg)
            else:
                loss, rmse = loss_classification(out, y_idx)

            loss.backward()

            optimizer.step()

            step += 1
```

```
            if loss.item() < best_loss:
                best_loss = loss.item()
                if ckpt_path:
                    save_checkpoint(ckpt_path, model, optimizer, step, best=Tr

            if step % log_every == 0:
                log_steps.append(step)
                log_loss.append(float(loss.item()))
                log_rmse.append(float(rmse.item()))

                elapsed = time.time() - t0
                print(f"[{step:6d}/{steps}] loss={loss.item():.6f} rmse={rmse.
                t0 = time.time()

            if step >= steps:
                break

    if ckpt_path:
        save_checkpoint(ckpt_path, model, optimizer, steps, best=False)

    return {"steps": log_steps, "loss": log_loss, "rmse": log_rmse}


@torch.no_grad()
def evaluate_rmse(model, base_ds, n_samples=1000, head_type="reg", bins=21, de
    model.eval()
    ds = TorchHomographyDataset(base_ds, n_samples_per_epoch=n_samples, for_cl
    loader = DataLoader(ds, batch_size=64, shuffle=False, num_workers=0)

    if head_type == "cls":
        max_jitter = base_ds.max_jitter
        centers = torch.linspace(-max_jitter, max_jitter, bins, device=device)

    se_sum = 0.0
    n = 0

    for batch in loader:
        x = batch["x"].to(device)
        y_reg = batch["y_reg"].to(device)

        pred = model(x)
        if head_type == "reg":
            pred_offsets = pred
        else:
            probs = F.softmax(pred, dim=-1)
            pred_offsets = torch.sum(probs * centers, dim=-1)  # (B,8)

        se = F.mse_loss(pred_offsets, y_reg, reduction="sum").item()
        se_sum += se
        n += y_reg.numel()

    rmse = math.sqrt(se_sum / n)
```

```
    return rmse
```

In [87]:
```
try:
    base_train = HomographyDatasetOnTheFly(img_dir=IMG_DIR, seed=SEED, return_
    base_eval  = HomographyDatasetOnTheFly(img_dir=IMG_DIR, seed=SEED+1, retur
except NameError as e:
    raise RuntimeError("Please paste or import your HomographyDatasetOnTheFly 

train_ds_reg = TorchHomographyDataset(base_train, n_samples_per_epoch=EPOCH_SA
train_loader_reg = DataLoader(train_ds_reg, batch_size=BATCH, shuffle=True, nu

train_ds_cls = TorchHomographyDataset(base_train, n_samples_per_epoch=EPOCH_SA
train_loader_cls = DataLoader(train_ds_cls, batch_size=BATCH, shuffle=True, nu

quick_view(base_eval)
quick_view(base_eval)
```





In [73]:
```
model_reg_display = HomographyNet(head_type="reg", bins=BINS)
model_cls_display = HomographyNet(head_type="cls", bins=BINS)
model_reg_display = model_reg_display.to(device)
model_cls_display = model_cls_display.to(device)

print("Model reg display:")
print(model_reg_display)
print("Model cls display:")
print(model_cls_display)
```

```
Model reg display:
HomographyNet(
  (backbone): HomographyBackbone(
    (stem): Conv2d(2, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bi
as=False)
    (b1a): BasicResBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
    )
    (b1b): BasicResBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
    )
    (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mod
e=False)
    (b2a): BasicResBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
    )
    (b2b): BasicResBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
    )
    (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mod
e=False)
    (b3a): BasicResBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
```

```
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (proj): Conv2d(64, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (proj_bn): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_r
unning_stats=True)
    )
    (b3b): BasicResBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    )
    (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mod
e=False)
    (b4a): BasicResBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    )
    (b4b): BasicResBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    )
    (flatten): Flatten(start_dim=1, end_dim=-1)
    (fc): Linear(in_features=8192, out_features=512, bias=True)
  )
  (head): RegressionHead(
    (fc): Linear(in_features=512, out_features=8, bias=True)
  )
)
Model cls display:
HomographyNet(
  (backbone): HomographyBackbone(
    (stem): Conv2d(2, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bi
as=False)
    (b1a): BasicResBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
```

```
1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
    )
    (b1b): BasicResBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
    )
    (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mod
e=False)
    (b2a): BasicResBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
    )
    (b2b): BasicResBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
    )
    (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mod
e=False)
    (b3a): BasicResBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (proj): Conv2d(64, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (proj_bn): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_r
unning_stats=True)
```

```
    )
    (b3b): BasicResBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    )
    (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mod
e=False)
    (b4a): BasicResBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    )
    (b4b): BasicResBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    )
    (flatten): Flatten(start_dim=1, end_dim=-1)
    (fc): Linear(in_features=8192, out_features=512, bias=True)
  )
  (head): ClassificationHead(
    (out): Linear(in_features=512, out_features=168, bias=True)
  )
)
```

In [ ]:

```
model_reg = HomographyNet(head_type="reg", bins=BINS).to(device)
opt_reg = torch.optim.Adam(model_reg.parameters(), lr=LR, weight_decay=LR)

hist_reg = train_loop(
    model_reg, train_loader_reg, "reg", opt_reg, device,
    steps=50000, log_every=200, ckpt_path=CKPT_PATH_REG
)

rmse_reg = evaluate_rmse(model_reg, base_eval, n_samples=1000, head_type="reg"
print(f"Regression head RMSE (offset units): {rmse_reg:.4f}")
```

```
[   200/50000] loss=7.845354 rmse=9.564452  (16.0s)
[   400/50000] loss=8.094589 rmse=9.508770  (15.5s)
[   600/50000] loss=9.448732 rmse=10.889608  (13.1s)
[   800/50000] loss=8.270405 rmse=9.953835  (12.5s)
[  1000/50000] loss=7.390285 rmse=9.500447  (12.3s)
[  1200/50000] loss=7.500259 rmse=9.531860  (12.4s)
[  1400/50000] loss=7.535744 rmse=9.245579  (12.5s)
[  1600/50000] loss=7.386853 rmse=9.096000  (14.1s)
[  1800/50000] loss=7.547968 rmse=8.973454  (12.6s)
[  2000/50000] loss=6.744994 rmse=8.704550  (13.5s)
[  2200/50000] loss=7.192206 rmse=9.057759  (12.7s)
[  2400/50000] loss=7.340994 rmse=9.039608  (12.5s)
[  2600/50000] loss=6.800165 rmse=9.127316  (13.1s)
[  2800/50000] loss=5.224873 rmse=7.022664  (12.5s)
[  3000/50000] loss=7.426796 rmse=9.378048  (12.6s)
[  3200/50000] loss=7.066835 rmse=9.584867  (13.2s)
[  3400/50000] loss=5.911444 rmse=7.561973  (12.4s)
[  3600/50000] loss=4.988547 rmse=7.040141  (12.9s)
[  3800/50000] loss=7.070647 rmse=9.365897  (12.2s)
[  4000/50000] loss=6.756580 rmse=8.525457  (12.1s)
[  4200/50000] loss=5.300085 rmse=7.634607  (13.5s)
[  4400/50000] loss=4.671831 rmse=6.552256  (12.0s)
[  4600/50000] loss=6.239097 rmse=8.076013  (12.3s)
[  4800/50000] loss=6.355613 rmse=8.586869  (12.2s)
[  5000/50000] loss=6.374621 rmse=9.209610  (13.6s)
[  5200/50000] loss=3.598414 rmse=5.436436  (13.1s)
[  5400/50000] loss=3.988561 rmse=5.634724  (12.4s)
[  5600/50000] loss=6.458029 rmse=8.884504  (13.2s)
[  5800/50000] loss=3.880324 rmse=5.586356  (12.4s)
[  6000/50000] loss=4.363981 rmse=5.450420  (12.4s)
[  6200/50000] loss=4.136287 rmse=5.772625  (13.5s)
[  6400/50000] loss=4.848718 rmse=7.125010  (12.4s)
[  6600/50000] loss=5.682841 rmse=7.457015  (12.5s)
[  6800/50000] loss=4.009364 rmse=5.993922  (12.4s)
[  7000/50000] loss=4.014655 rmse=5.827303  (12.3s)
[  7200/50000] loss=5.497490 rmse=9.009353  (12.4s)
[  7400/50000] loss=3.614478 rmse=5.631607  (12.3s)
[  7600/50000] loss=4.976005 rmse=7.144414  (12.9s)
[  7800/50000] loss=3.876025 rmse=5.650578  (12.0s)
[  8000/50000] loss=3.678087 rmse=5.934377  (12.1s)
[  8200/50000] loss=3.530886 rmse=5.962905  (13.1s)
[  8400/50000] loss=3.527323 rmse=5.319931  (13.0s)
[  8600/50000] loss=2.849411 rmse=4.529299  (12.2s)
[  8800/50000] loss=3.572257 rmse=5.771977  (12.2s)
[  9000/50000] loss=3.183847 rmse=4.476675  (12.5s)
[  9200/50000] loss=3.021691 rmse=4.539598  (12.3s)
[  9400/50000] loss=4.741549 rmse=7.165524  (12.3s)
[  9600/50000] loss=5.055687 rmse=7.390373  (12.3s)
[  9800/50000] loss=3.756292 rmse=5.604924  (12.3s)
[ 10000/50000] loss=4.536943 rmse=6.120351  (12.5s)
[ 10200/50000] loss=4.700075 rmse=6.429519  (12.2s)
[ 10400/50000] loss=5.133669 rmse=7.719763  (12.5s)
[ 10600/50000] loss=3.708106 rmse=5.633662  (12.1s)
[ 10800/50000] loss=3.498505 rmse=5.362708  (12.0s)
```

```
[ 11000/50000] loss=4.187168 rmse=5.888418  (12.8s)
[ 11200/50000] loss=2.710513 rmse=4.396047  (13.0s)
[ 11400/50000] loss=3.727095 rmse=5.115203  (12.2s)
[ 11600/50000] loss=3.081812 rmse=4.856049  (12.2s)
[ 11800/50000] loss=2.600600 rmse=3.901648  (12.3s)
[ 12000/50000] loss=6.690632 rmse=9.694099  (12.3s)
[ 12200/50000] loss=2.387419 rmse=3.406955  (12.2s)
[ 12400/50000] loss=4.111286 rmse=5.673726  (12.3s)
[ 12600/50000] loss=3.048019 rmse=4.579723  (12.8s)
[ 12800/50000] loss=2.599907 rmse=5.173110  (12.2s)
[ 13000/50000] loss=3.089477 rmse=4.831153  (13.0s)
[ 13200/50000] loss=3.536183 rmse=5.078465  (12.2s)
[ 13400/50000] loss=3.438670 rmse=5.364964  (12.3s)
[ 13600/50000] loss=2.679589 rmse=4.178216  (12.1s)
[ 13800/50000] loss=2.325797 rmse=4.375375  (12.0s)
[ 14000/50000] loss=3.273674 rmse=4.600495  (12.8s)
[ 14200/50000] loss=3.628319 rmse=5.077505  (12.0s)
[ 14400/50000] loss=4.041211 rmse=6.647391  (12.3s)
[ 14600/50000] loss=4.286749 rmse=6.262461  (12.4s)
[ 14800/50000] loss=4.408530 rmse=6.258455  (12.3s)
[ 15000/50000] loss=3.982361 rmse=6.295259  (12.2s)
[ 15200/50000] loss=2.895699 rmse=4.871090  (12.1s)
[ 15400/50000] loss=1.921746 rmse=2.839030  (12.3s)
[ 15600/50000] loss=2.271194 rmse=3.745638  (12.3s)
[ 15800/50000] loss=4.225043 rmse=6.135654  (12.2s)
[ 16000/50000] loss=3.295545 rmse=5.998500  (12.3s)
[ 16200/50000] loss=4.709785 rmse=6.672191  (12.3s)
[ 16400/50000] loss=3.446702 rmse=5.087717  (12.3s)
[ 16600/50000] loss=3.254494 rmse=5.493773  (12.0s)
[ 16800/50000] loss=3.589793 rmse=5.258380  (12.2s)
[ 17000/50000] loss=2.902499 rmse=4.638371  (13.0s)
[ 17200/50000] loss=3.694253 rmse=5.463843  (12.2s)
[ 17400/50000] loss=2.958549 rmse=4.486492  (12.2s)
[ 17600/50000] loss=2.229989 rmse=3.809993  (13.0s)
[ 17800/50000] loss=3.989594 rmse=6.885816  (12.3s)
[ 18000/50000] loss=3.590839 rmse=5.576548  (13.1s)
[ 18200/50000] loss=2.909623 rmse=4.521575  (12.3s)
[ 18400/50000] loss=2.364627 rmse=3.531402  (12.3s)
[ 18600/50000] loss=2.099844 rmse=3.379379  (12.3s)
[ 18800/50000] loss=3.342792 rmse=4.746519  (12.3s)
[ 19000/50000] loss=4.516835 rmse=7.106993  (12.3s)
[ 19200/50000] loss=3.894303 rmse=5.929589  (12.3s)
[ 19400/50000] loss=3.051088 rmse=4.450268  (12.3s)
[ 19600/50000] loss=2.463847 rmse=3.417320  (12.1s)
[ 19800/50000] loss=3.789097 rmse=6.271356  (12.1s)
[ 20000/50000] loss=2.437360 rmse=4.275567  (12.1s)
[ 20200/50000] loss=3.283207 rmse=5.131237  (12.4s)
[ 20400/50000] loss=3.863608 rmse=6.533658  (12.3s)
[ 20600/50000] loss=4.159543 rmse=6.133716  (12.4s)
[ 20800/50000] loss=2.875713 rmse=3.986456  (12.2s)
[ 21000/50000] loss=3.869926 rmse=5.682364  (12.2s)
[ 21200/50000] loss=2.864096 rmse=4.815007  (12.3s)
[ 21400/50000] loss=3.772692 rmse=6.507555  (12.4s)
[ 21600/50000] loss=4.038980 rmse=6.180099  (12.3s)
```

```
[ 21800/50000] loss=2.843810 rmse=4.537201  (12.3s)
[ 22000/50000] loss=2.789261 rmse=3.936451  (12.1s)
[ 22200/50000] loss=2.697875 rmse=4.483254  (12.0s)
[ 22400/50000] loss=2.525973 rmse=3.915117  (13.0s)
[ 22600/50000] loss=2.054804 rmse=3.399185  (12.3s)
[ 22800/50000] loss=3.160696 rmse=5.093309  (12.1s)
[ 23000/50000] loss=2.365504 rmse=3.633506  (12.3s)
[ 23200/50000] loss=2.461799 rmse=3.677299  (12.1s)
[ 23400/50000] loss=4.200178 rmse=6.687101  (13.1s)
[ 23600/50000] loss=3.650029 rmse=6.686574  (12.2s)
[ 23800/50000] loss=2.906072 rmse=4.122997  (12.2s)
[ 24000/50000] loss=1.792332 rmse=2.997540  (12.2s)
[ 24200/50000] loss=3.666263 rmse=5.181326  (12.3s)
[ 24400/50000] loss=2.105714 rmse=4.517451  (12.2s)
[ 24600/50000] loss=2.696733 rmse=4.258519  (12.2s)
[ 24800/50000] loss=2.539424 rmse=3.917817  (12.1s)
[ 25000/50000] loss=4.305904 rmse=6.823300  (12.0s)
[ 25200/50000] loss=2.389576 rmse=3.638485  (11.9s)
[ 25400/50000] loss=1.943551 rmse=3.321145  (12.2s)
[ 25600/50000] loss=3.802571 rmse=5.620234  (12.4s)
[ 25800/50000] loss=1.897460 rmse=2.953044  (12.1s)
[ 26000/50000] loss=3.385928 rmse=4.987241  (12.2s)
[ 26200/50000] loss=4.733193 rmse=7.329381  (12.2s)
[ 26400/50000] loss=2.088172 rmse=3.458334  (12.3s)
[ 26600/50000] loss=2.282268 rmse=3.897273  (12.3s)
[ 26800/50000] loss=3.230410 rmse=4.622880  (12.3s)
[ 27000/50000] loss=2.673938 rmse=4.168813  (12.3s)
[ 27200/50000] loss=2.651072 rmse=3.839484  (12.3s)
[ 27400/50000] loss=4.187821 rmse=6.020848  (12.1s)
[ 27600/50000] loss=3.091730 rmse=4.850762  (12.0s)
[ 27800/50000] loss=2.723626 rmse=4.301006  (12.5s)
[ 28000/50000] loss=4.077190 rmse=6.251694  (12.2s)
[ 28200/50000] loss=1.997131 rmse=3.262774  (12.3s)
[ 28400/50000] loss=4.222107 rmse=6.490458  (12.1s)
[ 28600/50000] loss=2.498706 rmse=4.211514  (12.3s)
[ 28800/50000] loss=1.901180 rmse=3.129218  (12.3s)
[ 29000/50000] loss=1.600649 rmse=2.486221  (12.3s)
[ 29200/50000] loss=2.498795 rmse=4.482595  (12.3s)
[ 29400/50000] loss=2.934659 rmse=4.491687  (13.0s)
[ 29600/50000] loss=3.149557 rmse=4.449492  (12.3s)
[ 29800/50000] loss=1.894315 rmse=2.837881  (12.1s)
[ 30000/50000] loss=2.342187 rmse=3.755680  (12.1s)
[ 30200/50000] loss=1.652010 rmse=2.892363  (12.1s)
[ 30400/50000] loss=2.532581 rmse=4.039885  (12.1s)
[ 30600/50000] loss=3.874401 rmse=7.148911  (12.2s)
[ 30800/50000] loss=2.522970 rmse=4.043844  (12.3s)
[ 31000/50000] loss=3.251106 rmse=5.633872  (12.3s)
[ 31200/50000] loss=2.767707 rmse=4.384394  (12.3s)
[ 31400/50000] loss=2.286766 rmse=3.708211  (13.4s)
[ 31600/50000] loss=1.670711 rmse=2.669962  (12.2s)
[ 31800/50000] loss=4.834499 rmse=7.504414  (12.2s)
[ 32000/50000] loss=2.513801 rmse=3.796278  (12.3s)
[ 32200/50000] loss=2.159378 rmse=3.458103  (12.4s)
[ 32400/50000] loss=2.263286 rmse=4.330888  (12.4s)
```

```
[ 32600/50000] loss=1.944995 rmse=2.965471  (12.4s)
[ 32800/50000] loss=1.518929 rmse=2.806834  (12.2s)
[ 33000/50000] loss=2.352719 rmse=3.462545  (12.3s)
[ 33200/50000] loss=2.233601 rmse=3.398512  (12.0s)
[ 33400/50000] loss=1.150327 rmse=1.980441  (12.2s)
[ 33600/50000] loss=2.412039 rmse=4.915231  (12.3s)
[ 33800/50000] loss=3.140105 rmse=5.696006  (12.1s)
[ 34000/50000] loss=2.565599 rmse=4.269280  (12.2s)
[ 34200/50000] loss=1.861740 rmse=2.983701  (12.2s)
[ 34400/50000] loss=2.119403 rmse=3.252949  (12.3s)
[ 34600/50000] loss=2.853971 rmse=4.090674  (12.2s)
[ 34800/50000] loss=2.327495 rmse=3.706457  (12.4s)
[ 35000/50000] loss=3.663727 rmse=5.613760  (12.2s)
[ 35200/50000] loss=2.392201 rmse=4.071845  (12.3s)
[ 35400/50000] loss=3.115856 rmse=6.597031  (12.0s)
[ 35600/50000] loss=1.689198 rmse=2.929930  (12.1s)
[ 35800/50000] loss=2.327723 rmse=3.314020  (12.0s)
[ 36000/50000] loss=3.150800 rmse=4.961806  (12.2s)
[ 36200/50000] loss=2.031796 rmse=3.321368  (12.3s)
[ 36400/50000] loss=2.059221 rmse=3.054289  (12.2s)
[ 36600/50000] loss=2.769590 rmse=4.619416  (12.2s)
[ 36800/50000] loss=1.821187 rmse=3.143088  (12.9s)
[ 37000/50000] loss=2.876553 rmse=4.608281  (12.4s)
[ 37200/50000] loss=1.801593 rmse=3.176765  (12.2s)
[ 37400/50000] loss=1.989694 rmse=3.271806  (12.4s)
[ 37600/50000] loss=1.485093 rmse=2.531990  (12.2s)
[ 37800/50000] loss=2.899185 rmse=4.095365  (12.2s)
[ 38000/50000] loss=3.110429 rmse=4.592760  (12.8s)
[ 38200/50000] loss=2.366095 rmse=4.377188  (12.3s)
[ 38400/50000] loss=2.498755 rmse=4.464544  (12.1s)
[ 38600/50000] loss=3.780760 rmse=5.901096  (12.0s)
[ 38800/50000] loss=1.648259 rmse=2.821708  (12.3s)
[ 39000/50000] loss=1.569617 rmse=2.561773  (12.3s)
[ 39200/50000] loss=1.137962 rmse=2.049604  (12.3s)
[ 39400/50000] loss=2.958781 rmse=4.441221  (12.2s)
[ 39600/50000] loss=1.837527 rmse=3.058046  (12.2s)
[ 39800/50000] loss=2.737990 rmse=3.973726  (12.2s)
[ 40000/50000] loss=1.812775 rmse=3.062511  (12.3s)
[ 40200/50000] loss=3.993032 rmse=6.270740  (12.2s)
[ 40400/50000] loss=3.157679 rmse=5.140196  (12.3s)
[ 40600/50000] loss=2.393587 rmse=3.545604  (12.3s)
[ 40800/50000] loss=2.993763 rmse=5.461863  (12.3s)
[ 41000/50000] loss=2.058924 rmse=3.191133  (12.0s)
[ 41200/50000] loss=2.744513 rmse=5.139775  (12.2s)
[ 41400/50000] loss=1.989984 rmse=3.108439  (12.4s)
[ 41600/50000] loss=3.031622 rmse=5.401796  (12.2s)
[ 41800/50000] loss=1.955090 rmse=3.242553  (12.2s)
[ 42000/50000] loss=2.638767 rmse=3.956233  (12.3s)
[ 42200/50000] loss=1.751638 rmse=2.802602  (12.2s)
[ 42400/50000] loss=1.676919 rmse=3.059541  (12.3s)
[ 42600/50000] loss=1.476499 rmse=2.443615  (12.3s)
[ 42800/50000] loss=1.659507 rmse=2.769338  (12.5s)
[ 43000/50000] loss=1.709751 rmse=2.787214  (12.4s)
[ 43200/50000] loss=2.180532 rmse=3.589869  (12.5s)
```

```
[ 43400/50000] loss=1.319270 rmse=2.360157  (12.3s)
[ 43600/50000] loss=1.421857 rmse=2.652342  (12.1s)
[ 43800/50000] loss=1.644464 rmse=2.919933  (12.4s)
[ 44000/50000] loss=1.938987 rmse=3.463315  (12.3s)
[ 44200/50000] loss=1.859850 rmse=2.870679  (12.4s)
[ 44400/50000] loss=1.661625 rmse=2.975393  (12.4s)
[ 44600/50000] loss=1.886950 rmse=3.180976  (12.4s)
[ 44800/50000] loss=1.374073 rmse=2.403522  (12.5s)
[ 45000/50000] loss=3.198349 rmse=5.490693  (12.3s)
[ 45200/50000] loss=3.202837 rmse=5.415231  (12.9s)
[ 45400/50000] loss=1.842650 rmse=3.025287  (14.5s)
[ 45600/50000] loss=3.635389 rmse=5.788752  (12.9s)
[ 45800/50000] loss=2.675523 rmse=4.519527  (12.5s)
[ 46000/50000] loss=1.658316 rmse=2.890660  (12.3s)
[ 46200/50000] loss=1.653128 rmse=2.660655  (12.3s)
[ 46400/50000] loss=1.453971 rmse=2.368270  (12.3s)
[ 46600/50000] loss=2.456276 rmse=3.658919  (12.4s)
[ 46800/50000] loss=3.231087 rmse=5.342645  (12.1s)
[ 47000/50000] loss=2.250651 rmse=3.689837  (12.1s)
[ 47200/50000] loss=1.842889 rmse=2.930804  (11.9s)
[ 47400/50000] loss=2.450416 rmse=3.906028  (12.2s)
[ 47600/50000] loss=2.060431 rmse=3.183724  (12.2s)
[ 47800/50000] loss=1.783619 rmse=3.415666  (12.3s)
[ 48000/50000] loss=1.390919 rmse=2.401852  (12.3s)
[ 48200/50000] loss=3.389098 rmse=5.923814  (12.3s)
[ 48400/50000] loss=1.354129 rmse=2.190774  (12.3s)
[ 48600/50000] loss=1.965923 rmse=3.503609  (12.2s)
[ 48800/50000] loss=2.655124 rmse=3.921404  (12.1s)
[ 49000/50000] loss=2.127518 rmse=3.271364  (12.1s)
[ 49200/50000] loss=1.385918 rmse=2.782660  (12.1s)
[ 49400/50000] loss=2.937822 rmse=4.473844  (11.8s)
[ 49600/50000] loss=2.016658 rmse=3.335984  (11.9s)
[ 49800/50000] loss=1.909918 rmse=2.998585  (12.2s)
[ 50000/50000] loss=3.322631 rmse=4.779604  (12.2s)
Regression head RMSE (offset units): 3.6691
```

In [19]:

```python
model_cls = HomographyNet(head_type="cls", bins=BINS).to(device)
opt_cls = torch.optim.Adam(model_cls.parameters(), lr=LR)

hist_cls = train_loop(
    model_cls, train_loader_cls, "cls", opt_cls, device,
    steps=50000, log_every=200, ckpt_path=CKPT_PATH_CLS
)

rmse_cls = evaluate_rmse(model_cls, base_eval, n_samples=1000, head_type="cls"
print(f"Classification head RMSE (offset units): {rmse_cls:.4f}")
```

```
[   200/50000] loss=3.018384 rmse=0.550575  (17.1s)
[   400/50000] loss=3.018543 rmse=0.634577  (13.9s)
[   600/50000] loss=3.071156 rmse=0.577874  (16.8s)
[   800/50000] loss=3.032109 rmse=0.514353  (13.6s)
[  1000/50000] loss=3.035675 rmse=0.661504  (12.3s)
[  1200/50000] loss=3.005153 rmse=0.609619  (12.4s)
[  1400/50000] loss=3.069067 rmse=0.551665  (12.9s)
[  1600/50000] loss=2.940451 rmse=0.597881  (12.3s)
[  1800/50000] loss=3.008291 rmse=0.575162  (12.3s)
[  2000/50000] loss=2.870760 rmse=0.497712  (12.4s)
[  2200/50000] loss=3.012980 rmse=0.552904  (12.4s)
[  2400/50000] loss=2.946893 rmse=0.599340  (12.3s)
[  2600/50000] loss=3.053334 rmse=0.608538  (12.2s)
[  2800/50000] loss=2.948323 rmse=0.618105  (12.2s)
[  3000/50000] loss=2.971889 rmse=0.462782  (12.3s)
[  3200/50000] loss=2.915077 rmse=0.649617  (12.4s)
[  3400/50000] loss=2.993295 rmse=0.542502  (12.4s)
[  3600/50000] loss=2.983589 rmse=0.597109  (12.4s)
[  3800/50000] loss=3.015971 rmse=0.617729  (12.1s)
[  4000/50000] loss=2.942851 rmse=0.614315  (12.1s)
[  4200/50000] loss=2.921264 rmse=0.634726  (12.0s)
[  4400/50000] loss=2.991904 rmse=0.623247  (12.4s)
[  4600/50000] loss=2.983378 rmse=0.599693  (12.3s)
[  4800/50000] loss=2.946183 rmse=0.560697  (13.1s)
[  5000/50000] loss=2.965169 rmse=0.573789  (12.3s)
[  5200/50000] loss=3.027997 rmse=0.583293  (12.9s)
[  5400/50000] loss=2.990099 rmse=0.577292  (12.2s)
[  5600/50000] loss=3.116422 rmse=0.566661  (12.4s)
[  5800/50000] loss=2.957625 rmse=0.605933  (12.4s)
[  6000/50000] loss=2.991229 rmse=0.594683  (12.6s)
[  6200/50000] loss=2.948727 rmse=0.567270  (12.4s)
[  6400/50000] loss=3.001664 rmse=0.610436  (12.7s)
[  6600/50000] loss=2.968229 rmse=0.574516  (13.0s)
[  6800/50000] loss=3.053880 rmse=0.660689  (12.5s)
[  7000/50000] loss=2.991954 rmse=0.556084  (12.6s)
[  7200/50000] loss=2.941136 rmse=0.553297  (13.5s)
[  7400/50000] loss=2.920069 rmse=0.607544  (12.9s)
[  7600/50000] loss=2.882462 rmse=0.552237  (12.4s)
[  7800/50000] loss=2.948978 rmse=0.576106  (12.8s)
[  8000/50000] loss=3.019230 rmse=0.648518  (12.6s)
[  8200/50000] loss=2.907863 rmse=0.587743  (13.0s)
[  8400/50000] loss=2.862860 rmse=0.548896  (12.2s)
[  8600/50000] loss=2.596949 rmse=0.457435  (13.6s)
[  8800/50000] loss=2.875399 rmse=0.493881  (12.4s)
[  9000/50000] loss=2.798070 rmse=0.498474  (12.9s)
[  9200/50000] loss=2.833237 rmse=0.475920  (12.1s)
[  9400/50000] loss=2.803846 rmse=0.530141  (12.3s)
[  9600/50000] loss=2.871992 rmse=0.514984  (12.9s)
[  9800/50000] loss=2.733814 rmse=0.515918  (12.3s)
[ 10000/50000] loss=2.733140 rmse=0.441610  (13.8s)
[ 10200/50000] loss=2.790370 rmse=0.426100  (12.4s)
[ 10400/50000] loss=2.934055 rmse=0.448427  (12.3s)
[ 10600/50000] loss=2.750597 rmse=0.505922  (12.5s)
[ 10800/50000] loss=2.926224 rmse=0.612515  (12.6s)
```

```
[ 11000/50000] loss=2.760523 rmse=0.569143  (12.4s)
[ 11200/50000] loss=2.886260 rmse=0.511137  (12.6s)
[ 11400/50000] loss=2.788973 rmse=0.450788  (12.4s)
[ 11600/50000] loss=2.692179 rmse=0.411209  (12.3s)
[ 11800/50000] loss=2.805916 rmse=0.494823  (13.2s)
[ 12000/50000] loss=2.862520 rmse=0.419770  (12.4s)
[ 12200/50000] loss=2.652982 rmse=0.390306  (12.5s)
[ 12400/50000] loss=2.592195 rmse=0.335900  (12.8s)
[ 12600/50000] loss=2.639244 rmse=0.511720  (12.3s)
[ 12800/50000] loss=2.625078 rmse=0.416126  (12.2s)
[ 13000/50000] loss=2.715473 rmse=0.475693  (12.1s)
[ 13200/50000] loss=2.520436 rmse=0.449593  (13.0s)
[ 13400/50000] loss=2.575285 rmse=0.505331  (12.4s)
[ 13600/50000] loss=2.983329 rmse=0.438587  (12.6s)
[ 13800/50000] loss=2.684216 rmse=0.423548  (12.5s)
[ 14000/50000] loss=2.700834 rmse=0.465544  (12.5s)
[ 14200/50000] loss=2.426347 rmse=0.381267  (12.5s)
[ 14400/50000] loss=2.780886 rmse=0.476195  (13.2s)
[ 14600/50000] loss=2.595582 rmse=0.474191  (12.4s)
[ 14800/50000] loss=2.540014 rmse=0.365022  (12.5s)
[ 15000/50000] loss=2.836083 rmse=0.450511  (12.4s)
[ 15200/50000] loss=2.658104 rmse=0.398238  (12.6s)
[ 15400/50000] loss=2.656340 rmse=0.341709  (12.4s)
[ 15600/50000] loss=2.466552 rmse=0.438884  (12.7s)
[ 15800/50000] loss=2.447267 rmse=0.326320  (12.7s)
[ 16000/50000] loss=2.710407 rmse=0.356905  (12.5s)
[ 16200/50000] loss=2.605961 rmse=0.445062  (12.5s)
[ 16400/50000] loss=2.533343 rmse=0.372289  (12.3s)
[ 16600/50000] loss=2.595652 rmse=0.406437  (12.2s)
[ 16800/50000] loss=2.404014 rmse=0.314790  (13.3s)
[ 17000/50000] loss=2.538770 rmse=0.408793  (12.2s)
[ 17200/50000] loss=2.447212 rmse=0.325043  (12.3s)
[ 17400/50000] loss=2.497617 rmse=0.407396  (12.4s)
[ 17600/50000] loss=2.501730 rmse=0.293822  (13.6s)
[ 17800/50000] loss=2.565194 rmse=0.376791  (12.8s)
[ 18000/50000] loss=2.352953 rmse=0.401579  (12.5s)
[ 18200/50000] loss=2.418436 rmse=0.392968  (12.4s)
[ 18400/50000] loss=2.595705 rmse=0.459423  (13.3s)
[ 18600/50000] loss=2.515650 rmse=0.398798  (12.5s)
[ 18800/50000] loss=2.410021 rmse=0.373552  (12.6s)
[ 19000/50000] loss=2.588279 rmse=0.327478  (12.7s)
[ 19200/50000] loss=2.422560 rmse=0.213467  (13.3s)
[ 19400/50000] loss=2.379605 rmse=0.328984  (12.4s)
[ 19600/50000] loss=2.443720 rmse=0.386867  (12.5s)
[ 19800/50000] loss=2.399844 rmse=0.375533  (12.5s)
[ 20000/50000] loss=2.272465 rmse=0.218536  (13.4s)
[ 20200/50000] loss=2.705691 rmse=0.390255  (12.5s)
[ 20400/50000] loss=2.616607 rmse=0.414925  (12.4s)
[ 20600/50000] loss=2.280515 rmse=0.214745  (12.6s)
[ 20800/50000] loss=2.551784 rmse=0.331211  (12.7s)
[ 21000/50000] loss=2.433271 rmse=0.345237  (12.6s)
[ 21200/50000] loss=2.348276 rmse=0.266670  (12.6s)
[ 21400/50000] loss=2.348166 rmse=0.337505  (12.7s)
[ 21600/50000] loss=2.652042 rmse=0.426967  (13.4s)
```

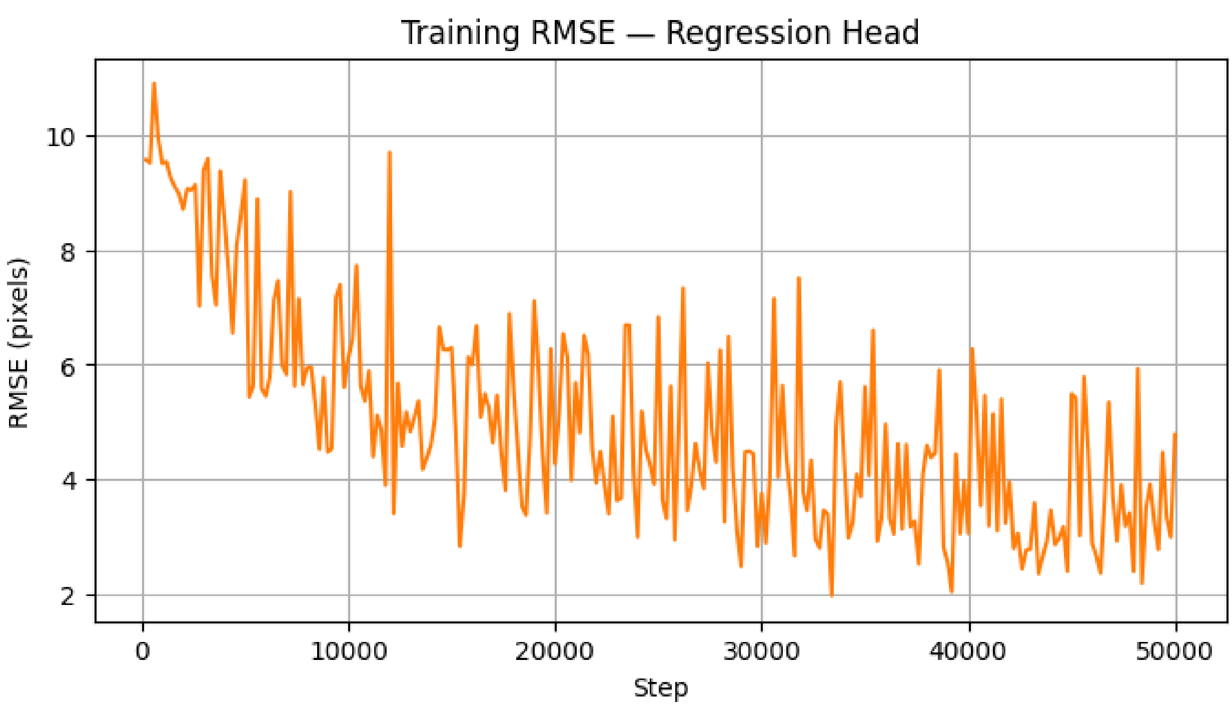
```
[ 21800/50000] loss=2.605084 rmse=0.372135  (12.7s)
[ 22000/50000] loss=2.430830 rmse=0.296838  (13.6s)
[ 22200/50000] loss=2.635053 rmse=0.382018  (12.7s)
[ 22400/50000] loss=2.632388 rmse=0.314765  (12.5s)
[ 22600/50000] loss=2.417004 rmse=0.257806  (13.1s)
[ 22800/50000] loss=2.384488 rmse=0.245031  (12.5s)
[ 23000/50000] loss=2.527646 rmse=0.343810  (12.5s)
[ 23200/50000] loss=2.288239 rmse=0.285426  (13.4s)
[ 23400/50000] loss=2.465264 rmse=0.274265  (13.4s)
[ 23600/50000] loss=2.439032 rmse=0.296920  (12.4s)
[ 23800/50000] loss=2.304975 rmse=0.209891  (12.6s)
[ 24000/50000] loss=2.408693 rmse=0.326227  (12.4s)
[ 24200/50000] loss=2.347372 rmse=0.266179  (12.5s)
[ 24400/50000] loss=2.272118 rmse=0.250106  (13.4s)
[ 24600/50000] loss=2.317859 rmse=0.307554  (12.5s)
[ 24800/50000] loss=2.334075 rmse=0.330270  (12.5s)
[ 25000/50000] loss=2.321965 rmse=0.259989  (12.4s)
[ 25200/50000] loss=2.353724 rmse=0.232927  (12.6s)
[ 25400/50000] loss=2.392181 rmse=0.269904  (12.5s)
[ 25600/50000] loss=2.075518 rmse=0.296529  (12.9s)
[ 25800/50000] loss=2.265875 rmse=0.237712  (12.5s)
[ 26000/50000] loss=2.319148 rmse=0.258803  (12.5s)
[ 26200/50000] loss=2.224672 rmse=0.219845  (12.6s)
[ 26400/50000] loss=2.150991 rmse=0.166104  (12.5s)
[ 26600/50000] loss=2.429613 rmse=0.449561  (12.5s)
[ 26800/50000] loss=2.321227 rmse=0.228432  (12.3s)
[ 27000/50000] loss=2.372042 rmse=0.304125  (12.3s)
[ 27200/50000] loss=2.434407 rmse=0.358507  (13.1s)
[ 27400/50000] loss=2.176497 rmse=0.375750  (12.1s)
[ 27600/50000] loss=2.232030 rmse=0.294572  (12.5s)
[ 27800/50000] loss=2.118291 rmse=0.141026  (13.5s)
[ 28000/50000] loss=2.178574 rmse=0.223432  (12.4s)
[ 28200/50000] loss=2.160125 rmse=0.183810  (12.5s)
[ 28400/50000] loss=2.214906 rmse=0.215893  (12.5s)
[ 28600/50000] loss=2.435222 rmse=0.339452  (12.6s)
[ 28800/50000] loss=2.370053 rmse=0.331594  (12.5s)
[ 29000/50000] loss=2.051243 rmse=0.235844  (12.6s)
[ 29200/50000] loss=2.268234 rmse=0.204110  (13.2s)
[ 29400/50000] loss=2.008056 rmse=0.167909  (12.6s)
[ 29600/50000] loss=2.113302 rmse=0.208352  (12.6s)
[ 29800/50000] loss=2.220459 rmse=0.230061  (12.5s)
[ 30000/50000] loss=2.076742 rmse=0.229862  (12.5s)
[ 30200/50000] loss=2.231645 rmse=0.309009  (13.3s)
[ 30400/50000] loss=2.527405 rmse=0.287265  (12.6s)
[ 30600/50000] loss=2.155145 rmse=0.215375  (14.0s)
[ 30800/50000] loss=2.191962 rmse=0.299396  (12.7s)
[ 31000/50000] loss=2.334623 rmse=0.294111  (12.6s)
[ 31200/50000] loss=2.128907 rmse=0.184481  (12.6s)
[ 31400/50000] loss=2.241035 rmse=0.231391  (12.8s)
[ 31600/50000] loss=2.450767 rmse=0.316757  (12.4s)
[ 31800/50000] loss=2.310082 rmse=0.210821  (12.5s)
[ 32000/50000] loss=2.275958 rmse=0.253460  (12.4s)
[ 32200/50000] loss=2.159954 rmse=0.173563  (13.9s)
[ 32400/50000] loss=2.037379 rmse=0.164196  (12.4s)
```

```
[ 32600/50000] loss=1.929613 rmse=0.168906  (12.3s)
[ 32800/50000] loss=2.114288 rmse=0.191209  (12.5s)
[ 33000/50000] loss=2.091089 rmse=0.194891  (12.5s)
[ 33200/50000] loss=1.940091 rmse=0.262309  (12.4s)
[ 33400/50000] loss=1.980493 rmse=0.154042  (12.6s)
[ 33600/50000] loss=2.100675 rmse=0.191849  (12.4s)
[ 33800/50000] loss=2.167689 rmse=0.269570  (12.4s)
[ 34000/50000] loss=2.270023 rmse=0.221726  (12.4s)
[ 34200/50000] loss=2.098538 rmse=0.163808  (12.3s)
[ 34400/50000] loss=2.182641 rmse=0.248377  (12.3s)
[ 34600/50000] loss=2.115400 rmse=0.233122  (12.4s)
[ 34800/50000] loss=2.011914 rmse=0.273695  (12.2s)
[ 35000/50000] loss=2.291761 rmse=0.286831  (12.3s)
[ 35200/50000] loss=2.095314 rmse=0.184057  (12.0s)
[ 35400/50000] loss=2.012444 rmse=0.206106  (12.2s)
[ 35600/50000] loss=1.903958 rmse=0.177742  (12.2s)
[ 35800/50000] loss=2.291021 rmse=0.280697  (12.4s)
[ 36000/50000] loss=2.050954 rmse=0.175403  (12.2s)
[ 36200/50000] loss=1.880675 rmse=0.171168  (12.2s)
[ 36400/50000] loss=2.284709 rmse=0.290813  (12.1s)
[ 36600/50000] loss=2.160909 rmse=0.206562  (12.2s)
[ 36800/50000] loss=2.092315 rmse=0.194452  (12.5s)
[ 37000/50000] loss=2.219581 rmse=0.232894  (12.3s)
[ 37200/50000] loss=2.246212 rmse=0.164538  (12.4s)
[ 37400/50000] loss=1.992575 rmse=0.148298  (12.4s)
[ 37600/50000] loss=2.112521 rmse=0.168088  (12.6s)
[ 37800/50000] loss=2.301836 rmse=0.239998  (13.5s)
[ 38000/50000] loss=2.045254 rmse=0.173004  (12.8s)
[ 38200/50000] loss=2.138680 rmse=0.128659  (12.7s)
[ 38400/50000] loss=2.317728 rmse=0.263513  (12.1s)
[ 38600/50000] loss=2.024173 rmse=0.268987  (12.3s)
[ 38800/50000] loss=1.804350 rmse=0.146435  (14.2s)
[ 39000/50000] loss=2.135746 rmse=0.235845  (12.1s)
[ 39200/50000] loss=2.099531 rmse=0.129459  (12.2s)
[ 39400/50000] loss=2.054991 rmse=0.185811  (12.4s)
[ 39600/50000] loss=2.285126 rmse=0.421821  (12.5s)
[ 39800/50000] loss=2.220817 rmse=0.329321  (12.7s)
[ 40000/50000] loss=2.198353 rmse=0.224226  (12.5s)
[ 40200/50000] loss=2.057725 rmse=0.254138  (12.9s)
[ 40400/50000] loss=2.250357 rmse=0.316122  (12.8s)
[ 40600/50000] loss=2.085669 rmse=0.185653  (12.5s)
[ 40800/50000] loss=1.844454 rmse=0.120059  (12.6s)
[ 41000/50000] loss=2.300375 rmse=0.356744  (12.5s)
[ 41200/50000] loss=2.107580 rmse=0.224047  (12.5s)
[ 41400/50000] loss=2.092994 rmse=0.198452  (12.5s)
[ 41600/50000] loss=1.972460 rmse=0.146956  (12.5s)
[ 41800/50000] loss=2.007123 rmse=0.224242  (12.4s)
[ 42000/50000] loss=1.985482 rmse=0.137983  (12.3s)
[ 42200/50000] loss=2.115071 rmse=0.219468  (12.4s)
[ 42400/50000] loss=2.125639 rmse=0.270769  (12.2s)
[ 42600/50000] loss=2.135283 rmse=0.164686  (12.3s)
[ 42800/50000] loss=2.190885 rmse=0.247934  (12.3s)
[ 43000/50000] loss=2.024522 rmse=0.162355  (12.5s)
[ 43200/50000] loss=1.863038 rmse=0.159813  (12.5s)
```

```
[ 43400/50000] loss=2.102222 rmse=0.192418  (12.3s)
[ 43600/50000] loss=1.784050 rmse=0.137274  (13.3s)
[ 43800/50000] loss=2.057804 rmse=0.265366  (12.2s)
[ 44000/50000] loss=1.897624 rmse=0.175116  (12.2s)
[ 44200/50000] loss=1.714592 rmse=0.107157  (12.3s)
[ 44400/50000] loss=2.063580 rmse=0.143829  (12.2s)
[ 44600/50000] loss=2.016559 rmse=0.144564  (12.3s)
[ 44800/50000] loss=1.809199 rmse=0.105118  (12.2s)
[ 45000/50000] loss=2.020009 rmse=0.193966  (12.2s)
[ 45200/50000] loss=2.259730 rmse=0.249695  (12.2s)
[ 45400/50000] loss=1.894788 rmse=0.153897  (12.3s)
[ 45600/50000] loss=2.128311 rmse=0.301822  (12.1s)
[ 45800/50000] loss=1.874139 rmse=0.168468  (12.0s)
[ 46000/50000] loss=1.957902 rmse=0.147631  (12.1s)
[ 46200/50000] loss=2.021766 rmse=0.212165  (12.2s)
[ 46400/50000] loss=1.872948 rmse=0.118733  (12.1s)
[ 46600/50000] loss=2.068835 rmse=0.147589  (12.2s)
[ 46800/50000] loss=2.143196 rmse=0.244582  (12.2s)
[ 47000/50000] loss=2.018422 rmse=0.201304  (12.2s)
[ 47200/50000] loss=2.221364 rmse=0.168774  (12.3s)
[ 47400/50000] loss=1.999424 rmse=0.240541  (12.3s)
[ 47600/50000] loss=1.750215 rmse=0.113771  (12.2s)
[ 47800/50000] loss=1.939888 rmse=0.150878  (12.2s)
[ 48000/50000] loss=1.906821 rmse=0.155171  (12.3s)
[ 48200/50000] loss=2.009037 rmse=0.159917  (12.0s)
[ 48400/50000] loss=1.895436 rmse=0.150405  (12.0s)
[ 48600/50000] loss=2.001608 rmse=0.172285  (12.1s)
[ 48800/50000] loss=1.942957 rmse=0.204255  (12.4s)
[ 49000/50000] loss=2.109408 rmse=0.254153  (12.1s)
[ 49200/50000] loss=2.007523 rmse=0.142701  (12.3s)
[ 49400/50000] loss=1.920663 rmse=0.175362  (12.3s)
[ 49600/50000] loss=1.892809 rmse=0.137227  (12.4s)
[ 49800/50000] loss=1.944520 rmse=0.162580  (12.2s)
[ 50000/50000] loss=1.784710 rmse=0.109347  (12.8s)
Classification head RMSE (offset units): 2.9731
```

In [ ]:

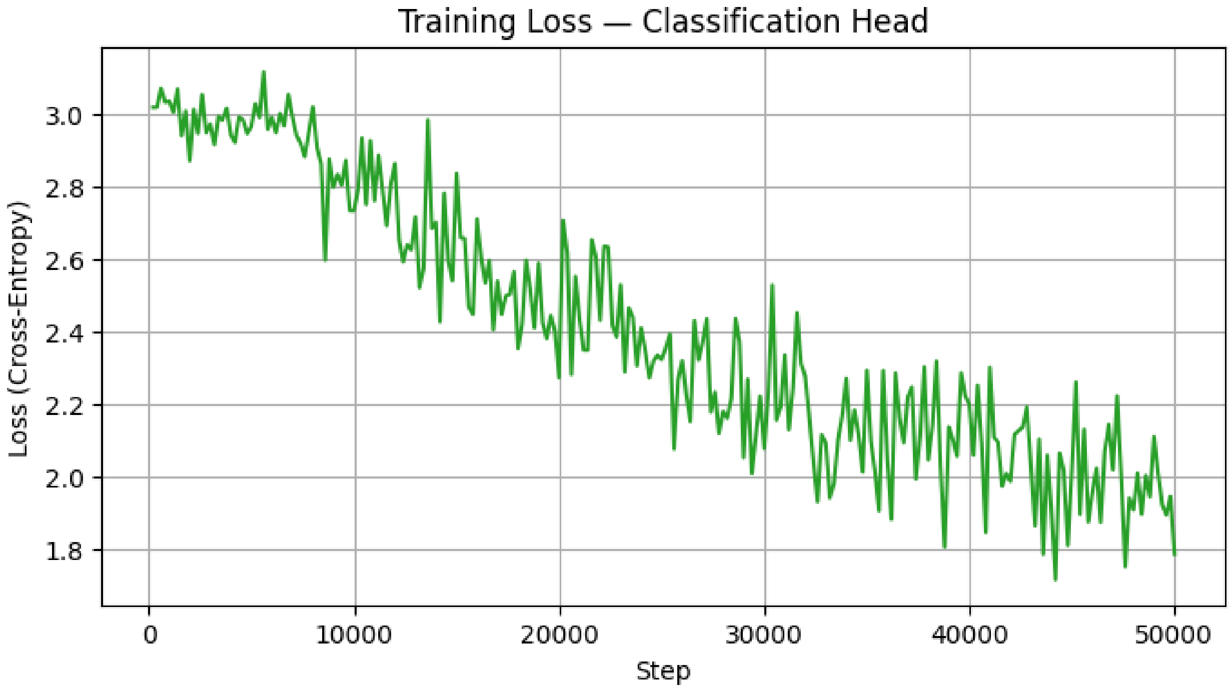
```python
# --- Regression model plots ---
plt.figure(figsize=(8,4))
plt.plot(hist_reg["steps"], hist_reg["loss"], color="tab:blue")
plt.xlabel("Step"); plt.ylabel("Loss (L2)")
plt.title("Training Loss — Regression Head")
plt.grid(True)
plt.show()

plt.figure(figsize=(8,4))
plt.plot(hist_reg["steps"], hist_reg["rmse"], color="tab:orange")
plt.xlabel("Step"); plt.ylabel("RMSE (pixels)")
plt.title("Training RMSE — Regression Head")
plt.grid(True)
plt.show()
```

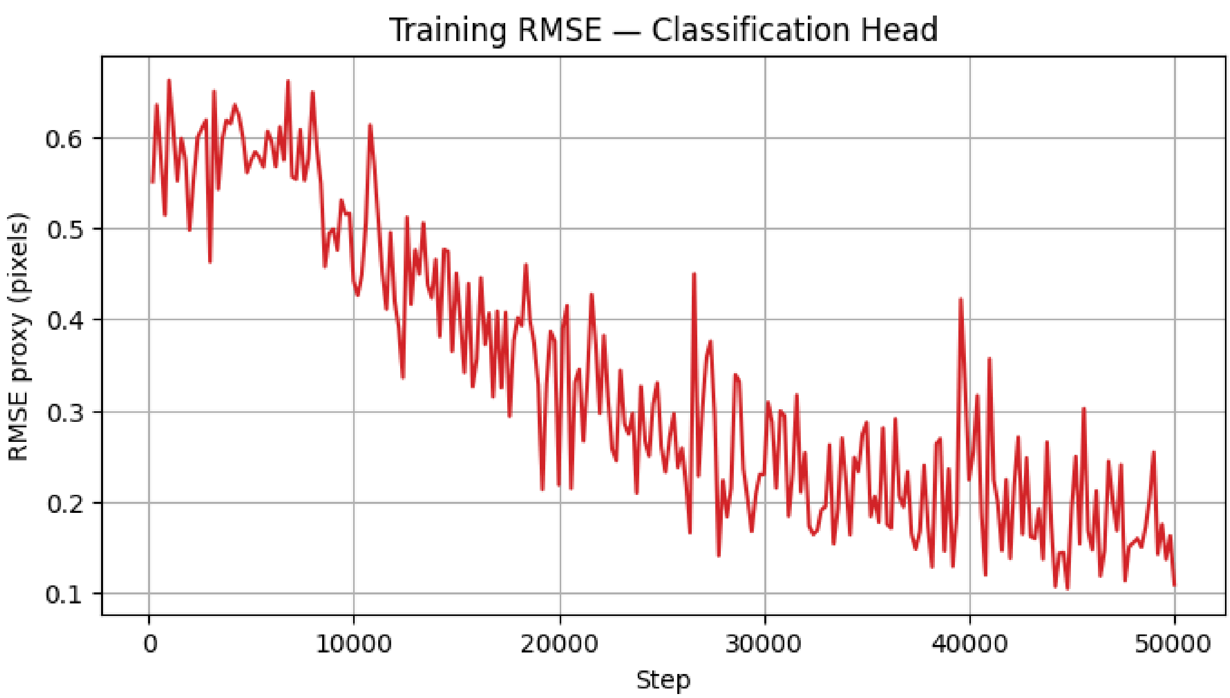


In [20]:

```
plt.figure(figsize=(8,4))
plt.plot(hist_cls["steps"], hist_cls["loss"], color="tab:green")
plt.xlabel("Step"); plt.ylabel("Loss (Cross-Entropy)")
plt.title("Training Loss — Classification Head")
plt.grid(True)
plt.show()

plt.figure(figsize=(8,4))
plt.plot(hist_cls["steps"], hist_cls["rmse"], color="tab:red")
plt.xlabel("Step"); plt.ylabel("RMSE proxy (pixels)")
```

```
plt.title("Training RMSE — Classification Head")
plt.grid(True)
plt.show()
```





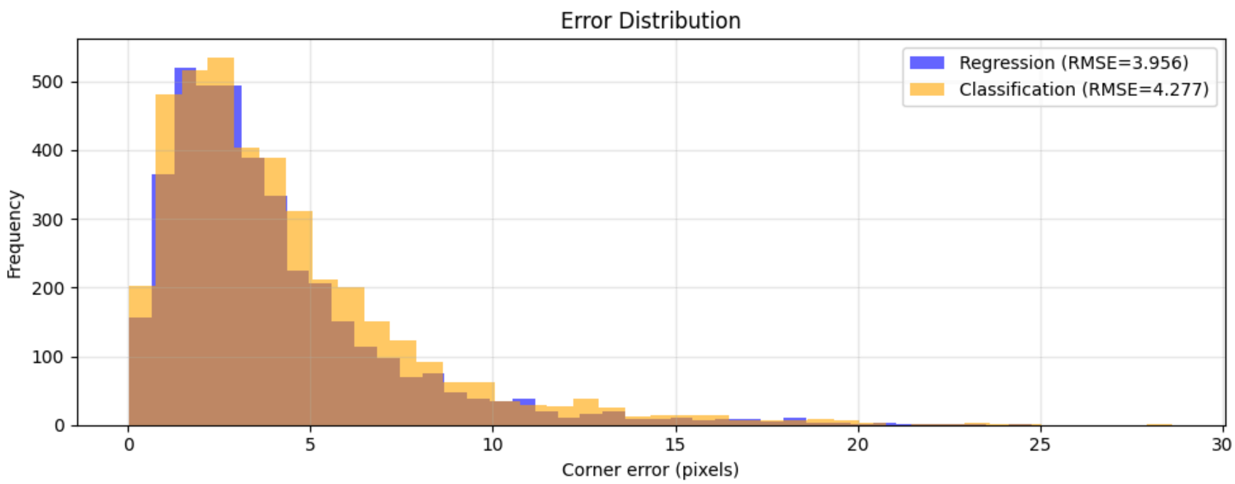
In [ ]:

```
model_reg = load_model("reg", "checkpoints_google/homography_reg_best.pt", bir
model_cls = load_model("cls", "checkpoints_google/homography_cls_best.pt", bir
def eval_nn_once(model_reg, model_cls, ds, n=1000, device=device, bins=BINS, c
    errors_reg, errors_cls = [], []
    for _ in range(n):
        s = ds.sample()
        x = torch.from_numpy(s["x"]).unsqueeze(0).float().to(device)
```

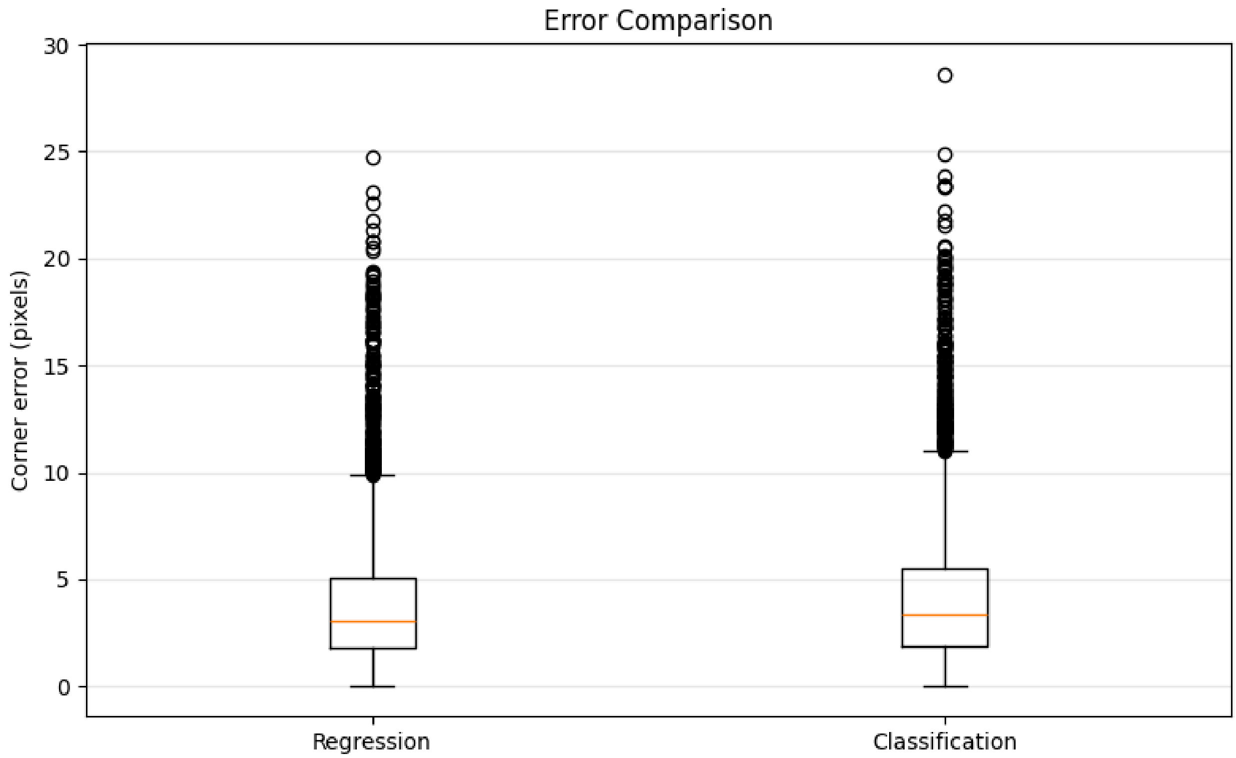
```
        y_true = s["y_reg"]

        with torch.no_grad():
            y_pred_reg = model_reg(x)[0].cpu().numpy()
            logits = model_cls(x)
            probs = torch.softmax(logits, dim=-1)
            centers = torch.linspace(-MAX_JITTER, MAX_JITTER, bins, device=dev
            y_pred_cls = (probs * centers).sum(dim=-1)[0].cpu().numpy()

        for c in range(4):
            dx = y_pred_reg[2*c]   - y_true[2*c]
            dy = y_pred_reg[2*c+1] - y_true[2*c+1]
            errors_reg.append(np.sqrt(dx*dx + dy*dy))

            dx = y_pred_cls[2*c]   - y_true[2*c]
            dy = y_pred_cls[2*c+1] - y_true[2*c+1]
            errors_cls.append(np.sqrt(dx*dx + dy*dy))

    errors_reg = np.array(errors_reg)
    errors_cls = np.array(errors_cls)
    mean_reg = float(np.mean(errors_reg))
    mean_cls = float(np.mean(errors_cls))

    if clip_hist_p is not None:
        cap = np.percentile(np.concatenate([errors_reg, errors_cls]), clip_his
        hist_range = (0, float(cap))
        title_suffix = f" (clipped ≤ P{clip_hist_p})"
    else:
        hist_range = None
        title_suffix = ""

    plt.figure(figsize=(10, 4))
    plt.hist(errors_reg, bins=40, range=hist_range, alpha=0.6,
             label=f"Regression (mean={mean_reg:.2f}px)")
    plt.hist(errors_cls, bins=40, range=hist_range, alpha=0.6,
             label=f"Classification (mean={mean_cls:.2f}px)")
    plt.axvline(mean_reg, linestyle='--', linewidth=1)
    plt.axvline(mean_cls, linestyle='--', linewidth=1)
    plt.xlabel("Corner error (pixels)")
    plt.ylabel("Frequency")
    plt.title("Error Distribution" + title_suffix)
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()

    plt.figure(figsize=(8, 5))
    plt.boxplot([errors_reg, errors_cls], labels=['Regression', 'Classificatio
    plt.ylabel('Corner error (pixels)')
    plt.title('Error Comparison')
    plt.grid(True, alpha=0.3, axis='y')
    plt.tight_layout()
    plt.show()
```

```
    return {
        "errors_reg": errors_reg,
        "errors_cls": errors_cls,
        "mean_reg": mean_reg,
        "mean_cls": mean_cls,
    }
res = eval_nn_once(model_reg, model_cls, base_eval, n=1000, clip_hist_p=99)
```

```
Loaded reg model from: checkpoints_google/homography_reg_best.pt
Loaded cls model from: checkpoints_google/homography_cls_best.pt
Evaluating 1000 samples...

Regression RMSE: 3.9561 pixels
Classification RMSE: 4.2769 pixels
```




```
/var/folders/64/6wxk92sj3px37_tfkmrktpf40000gq/T/ipykernel_34169/3379553388.p
y:60: MatplotlibDeprecationWarning: The 'labels' parameter of boxplot() has bee
n renamed 'tick_labels' since Matplotlib 3.9; support for the old name will be
dropped in 3.11.
  plt.boxplot([errors_reg, errors_cls], labels=['Regression', 'Classificatio
n'])
```

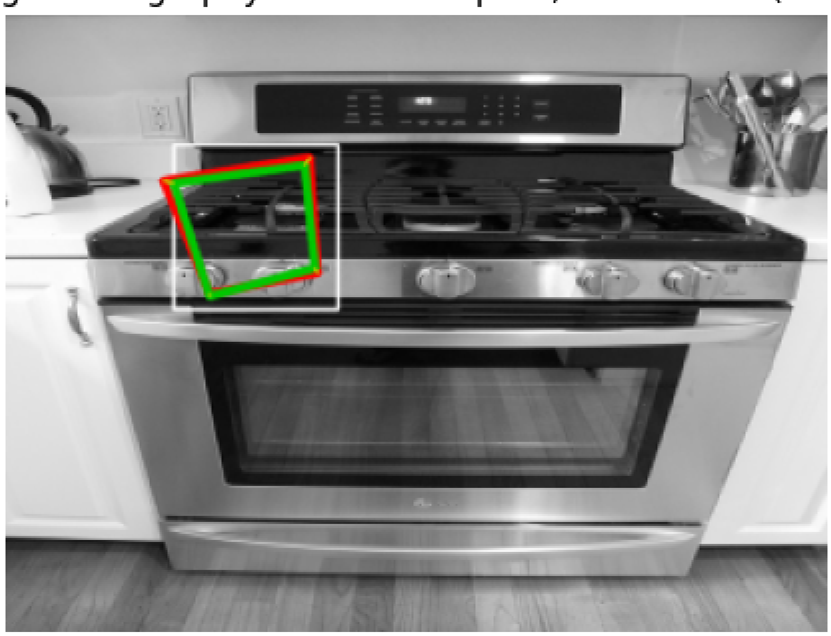


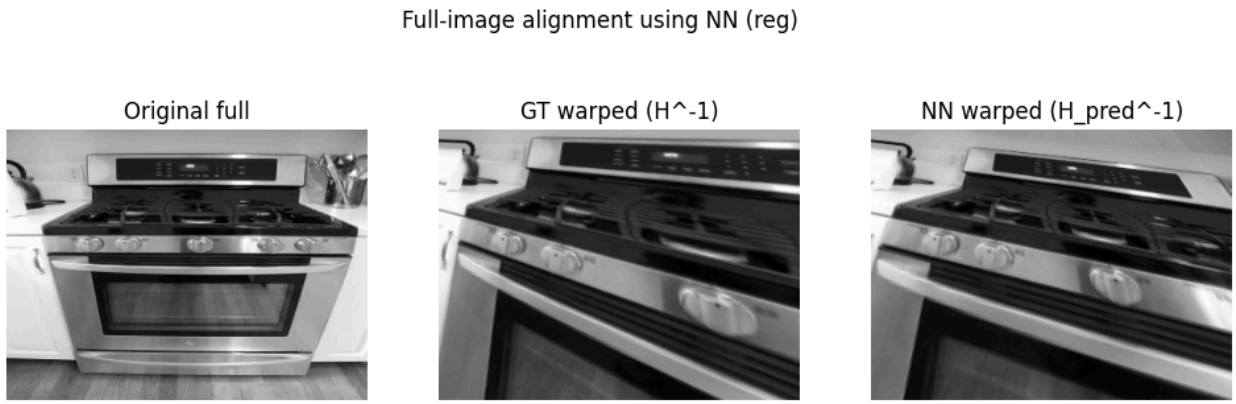
In [136…

```
result_reg = test_homography_head("reg")
result_reg = test_homography_head("reg")
```

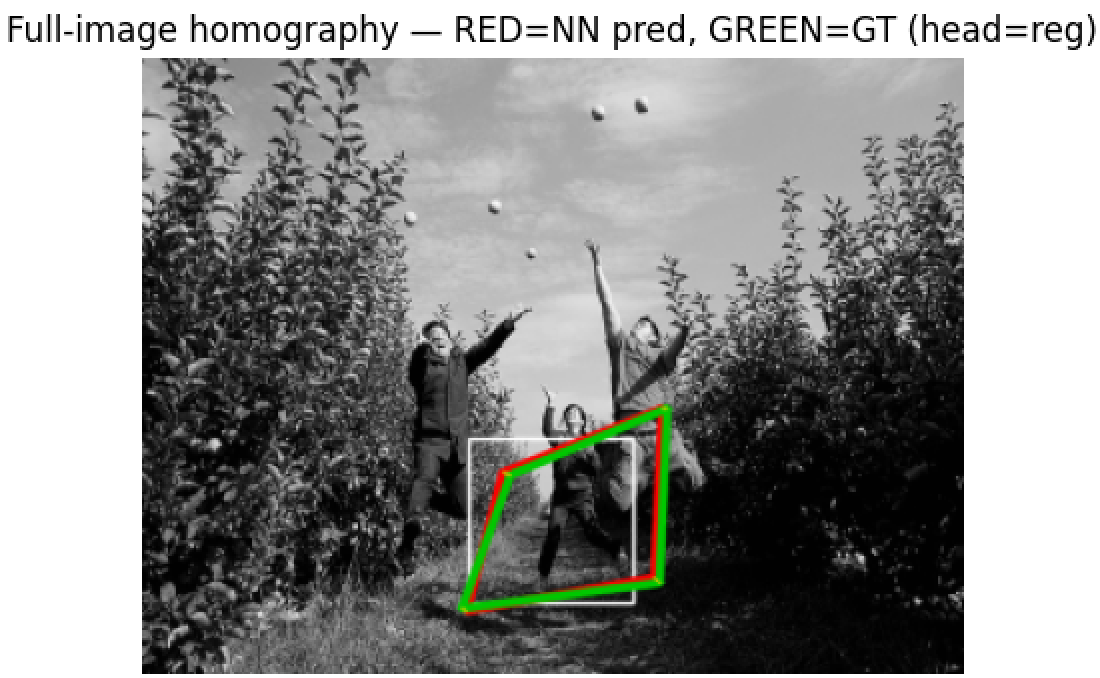
```
Loaded reg model from: checkpoints_google/homography_reg_best.pt
NN RMSE (pixels):   2.050689220428467
```

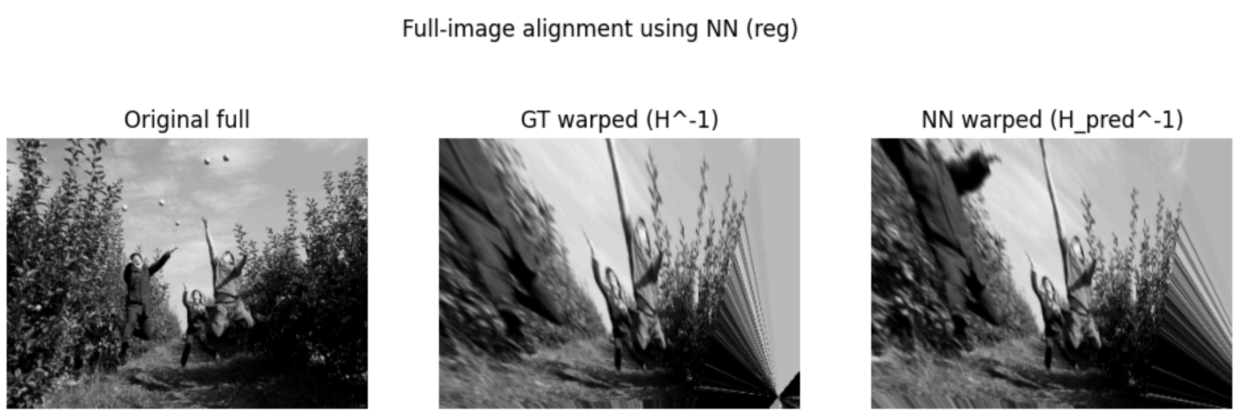
Full-image homography — RED=NN pred, GREEN=GT (head=reg)






```
Loaded reg model from: checkpoints_google/homography_reg_best.pt
NN RMSE (pixels):   1.7000116109848022
```





In [138…

```
result_reg = test_homography_head("cls")
result_reg = test_homography_head("cls")
```

```
Loaded cls model from: checkpoints_google/homography_cls_best.pt
NN RMSE (pixels):   0.605642557144165
```

Full-image homography — RED=NN pred, GREEN=GT (head=cls)

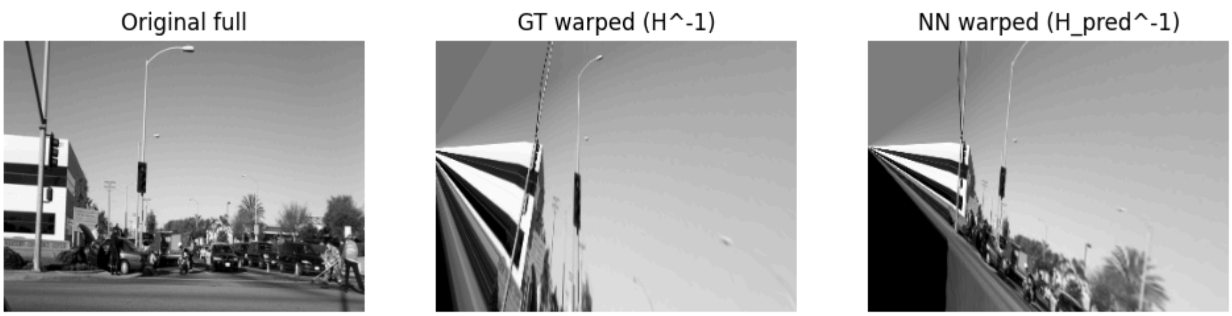
Full-image alignment using NN (cls)

Original full | GT warped (H^-1) | NN warped (H_pred^-1)

```
Loaded cls model from: checkpoints_google/homography_cls_best.pt
NN RMSE (pixels):   6.4123311042785645
```

Full-image homography — RED=NN pred, GREEN=GT (head=cls)

Full-image alignment using NN (cls)

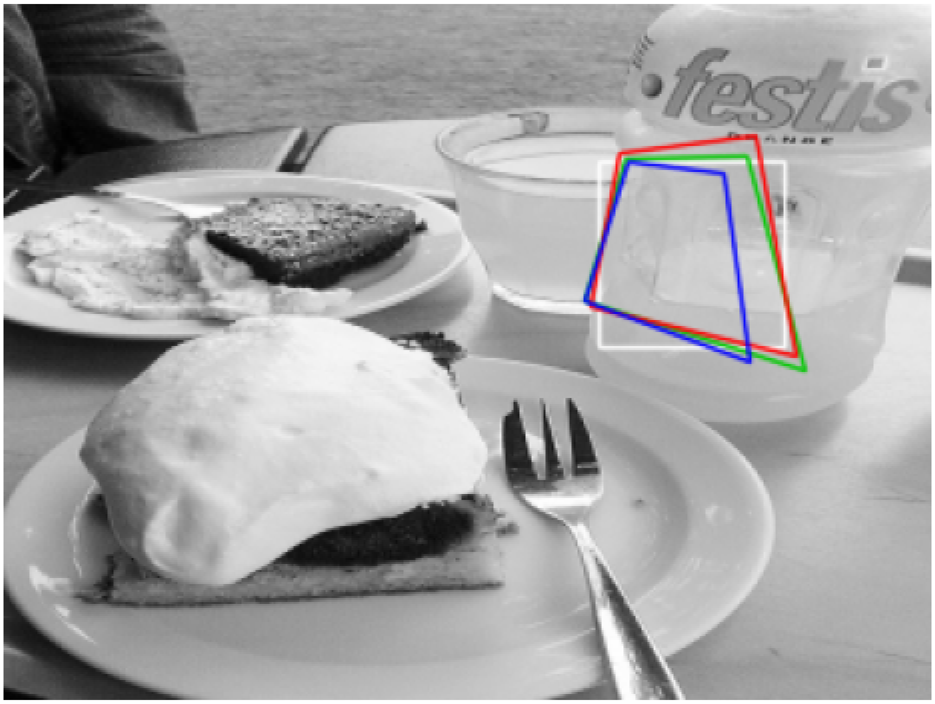


In [165…

```
out = test_homography_head(head="cls", classic=True)
```

```
Loaded cls model from: checkpoints_google/homography_cls_best.pt
NN RMSE (pixels):   3.456563949584961
```

Overlay — GT(green), NN(red), Classic(blue), Square(white)

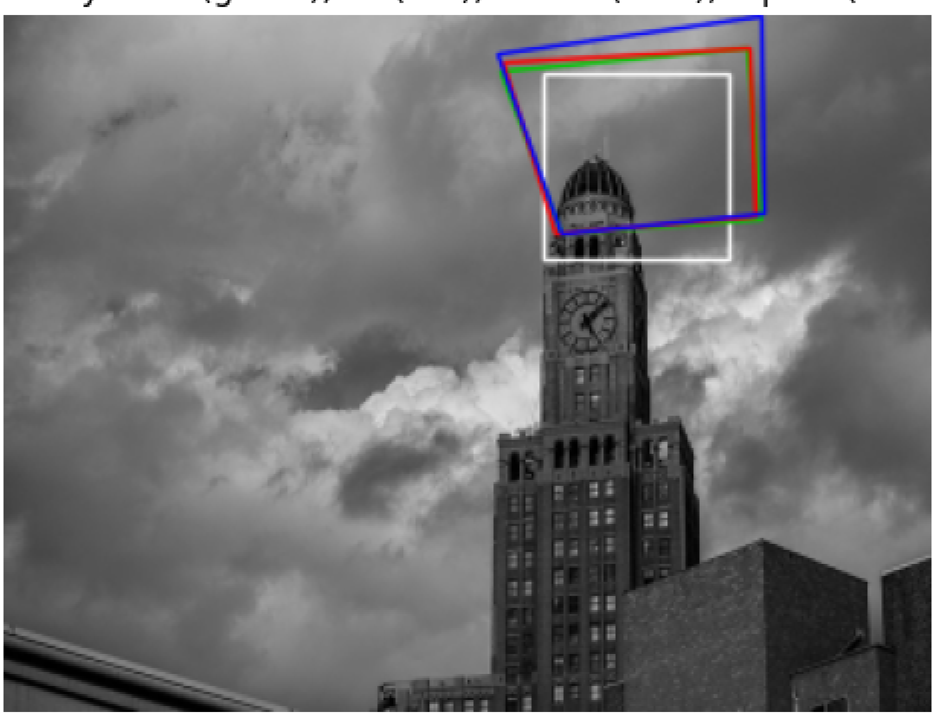


In [170…

```
out = test_homography_head(head="cls", classic=True)
```

```
Loaded cls model from: checkpoints_google/homography_cls_best.pt
NN RMSE (pixels):   1.5835593938827515
```

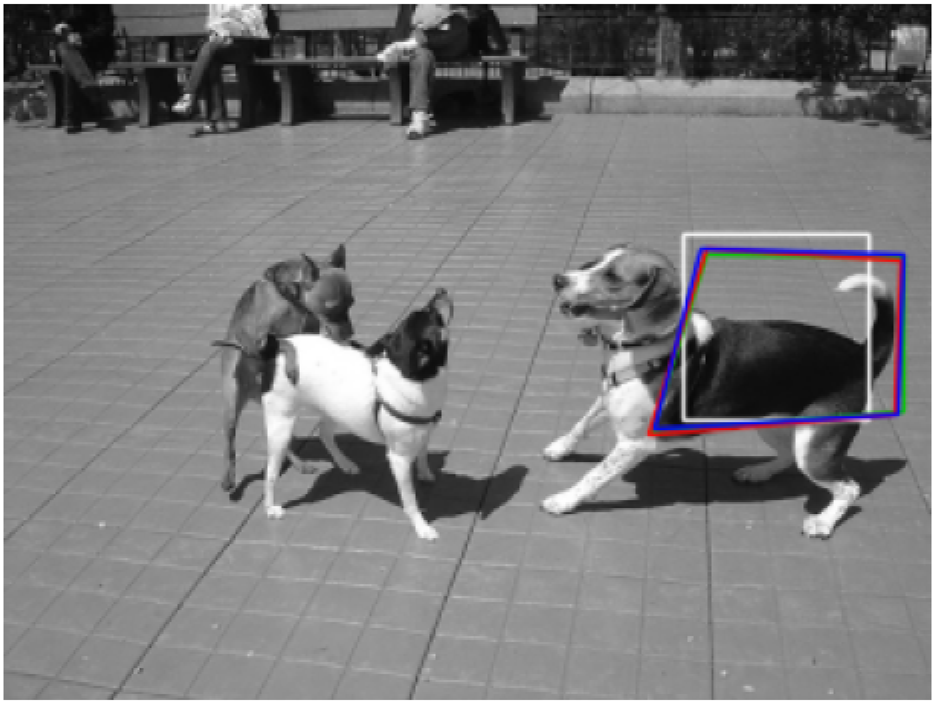
Overlay — GT(green), NN(red), Classic(blue), Square(white)



In [194…

```
out = test_homography_head(head="cls", classic=True)
```

```
Loaded cls model from: checkpoints_google/homography_cls_best.pt
NN RMSE (pixels):   1.5842304229736328
```



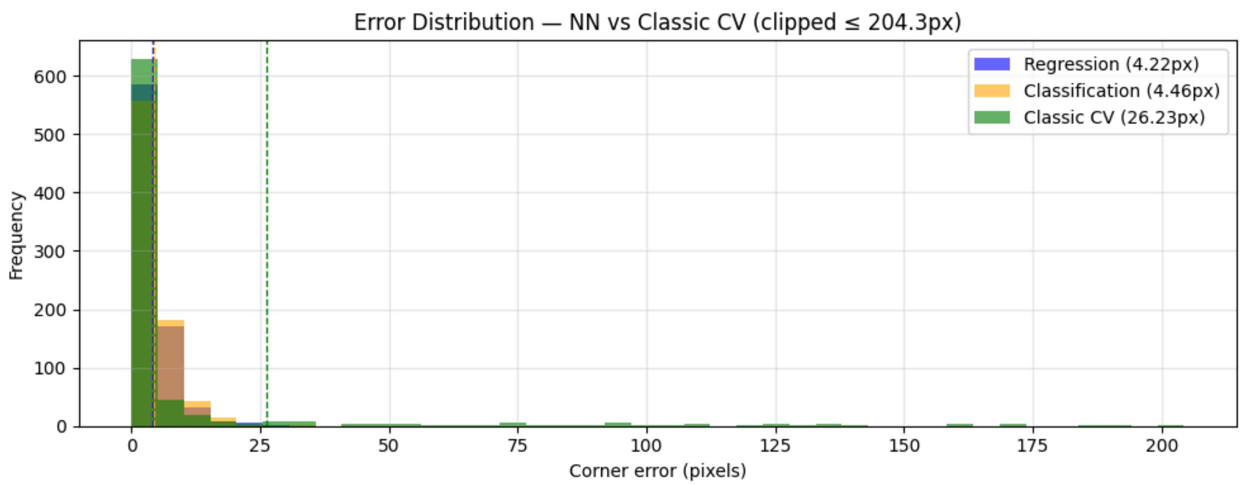
In [ ]:

```
model_reg = load_model("reg", "checkpoints_google/homography_reg_best.pt", bir
model_cls = load_model("cls", "checkpoints_google/homography_cls_best.pt", bir

print("Evaluating 200 samples (NN + Classic)...")
errors_reg, errors_cls, errors_cv = [], [], []

for i in range(200):
    s = sample_with_full(base_eval, return_cls=True)
    x = torch.from_numpy(s["x"]).unsqueeze(0).float().to(device)
    y_true = s["y_reg"]
    gray_full = s["gray_full"]
    warped_full = s["warped_full"]
    x0, y0 = map(int, s["xy"])

    with torch.no_grad():
        y_pred_reg = model_reg(x)[0].cpu().numpy()

    with torch.no_grad():
        logits = model_cls(x)
        probs = torch.softmax(logits, dim=-1)
        centers = torch.linspace(-MAX_JITTER, MAX_JITTER, BINS, device=device)
        y_pred_cls = (probs * centers).sum(dim=-1)[0].cpu().numpy()

    y_pred_cv = _classic_offsets_from_full(gray_full, warped_full, x0, y0, PAT
    if y_pred_cv is None:
        y_pred_cv = np.zeros_like(y_true)
```

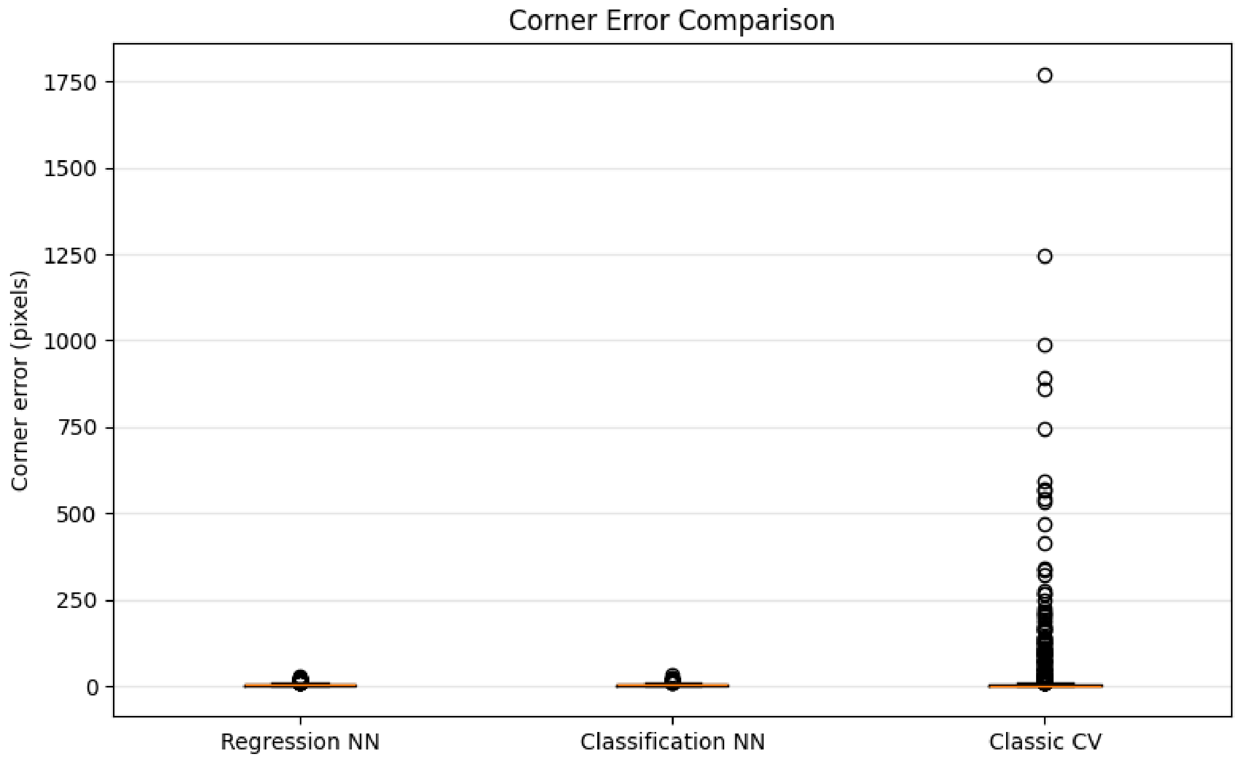
```
    for corner in range(4):
        dx = y_pred_reg[2*corner] - y_true[2*corner]
        dy = y_pred_reg[2*corner+1] - y_true[2*corner+1]
        errors_reg.append(np.sqrt(dx**2 + dy**2))

        dx = y_pred_cls[2*corner] - y_true[2*corner]
        dy = y_pred_cls[2*corner+1] - y_true[2*corner+1]
        errors_cls.append(np.sqrt(dx**2 + dy**2))

        dx = y_pred_cv[2*corner] - y_true[2*corner]
        dy = y_pred_cv[2*corner+1] - y_true[2*corner+1]
        errors_cv.append(np.sqrt(dx**2 + dy**2))

errors_reg = np.array(errors_reg)
errors_cls = np.array(errors_cls)
errors_cv = np.array(errors_cv)

print(f"\nRegression RMSE:      {np.mean(errors_reg):.4f} px")
print(f"Classification RMSE:  {np.mean(errors_cls):.4f} px")
print(f"Classic CV RMSE:      {np.mean(errors_cv):.4f} px")

p99 = np.percentile(np.concatenate([errors_reg, errors_cls, errors_cv]), 99)
print(f"Histogram clipped at P99 = {p99:.2f}px")

plt.figure(figsize=(10, 4))
plt.hist(errors_reg, bins=40, range=(0, p99), alpha=0.6,
         label=f'Regression ({np.mean(errors_reg):.2f}px)', color='blue')
plt.hist(errors_cls, bins=40, range=(0, p99), alpha=0.6,
         label=f'Classification ({np.mean(errors_cls):.2f}px)', color='orange'
plt.hist(errors_cv,  bins=40, range=(0, p99), alpha=0.6,
         label=f'Classic CV ({np.mean(errors_cv):.2f}px)', color='green')

plt.axvline(np.mean(errors_reg), color='blue', linestyle='--', linewidth=1)
plt.axvline(np.mean(errors_cls), color='orange', linestyle='--', linewidth=1)
plt.axvline(np.mean(errors_cv),  color='green', linestyle='--', linewidth=1)

plt.xlabel('Corner error (pixels)')
plt.ylabel('Frequency')
plt.title(f'Error Distribution — NN vs Classic CV (clipped ≤ {p99:.1f}px)')
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

plt.figure(figsize=(8, 5))
plt.boxplot([errors_reg, errors_cls, errors_cv],
            labels=['Regression NN', 'Classification NN', 'Classic CV'])
plt.ylabel('Corner error (pixels)')
plt.title('Corner Error Comparison')
plt.grid(True, alpha=0.3, axis='y')
plt.tight_layout()
plt.show()
```

```python
mean_reg = float(np.mean(errors_reg))
mean_cls = float(np.mean(errors_cls))
mean_cv  = float(np.mean(errors_cv))

plt.figure(figsize=(8, 5))
plt.boxplot([errors_reg, errors_cls, errors_cv],
            labels=['Regression NN', 'Classification NN', 'Classic CV'],
            showfliers=False)
plt.ylim(0, 30)
for i, m in enumerate([mean_reg, mean_cls, mean_cv], 1):
    if 0 <= m <= 30:
        plt.axhline(m, linestyle='--', linewidth=1, xmin=(i-1)/3, xmax=i/3, cc
        plt.text(i-0.05, m, f'{m:.2f}', va='bottom', ha='right', fontsize=9)
plt.ylabel('Corner error (pixels)')
plt.title('Corner Error Comparison (0–30 px, means)')
plt.grid(True, alpha=0.3, axis='y')
plt.tight_layout()
plt.show()
```

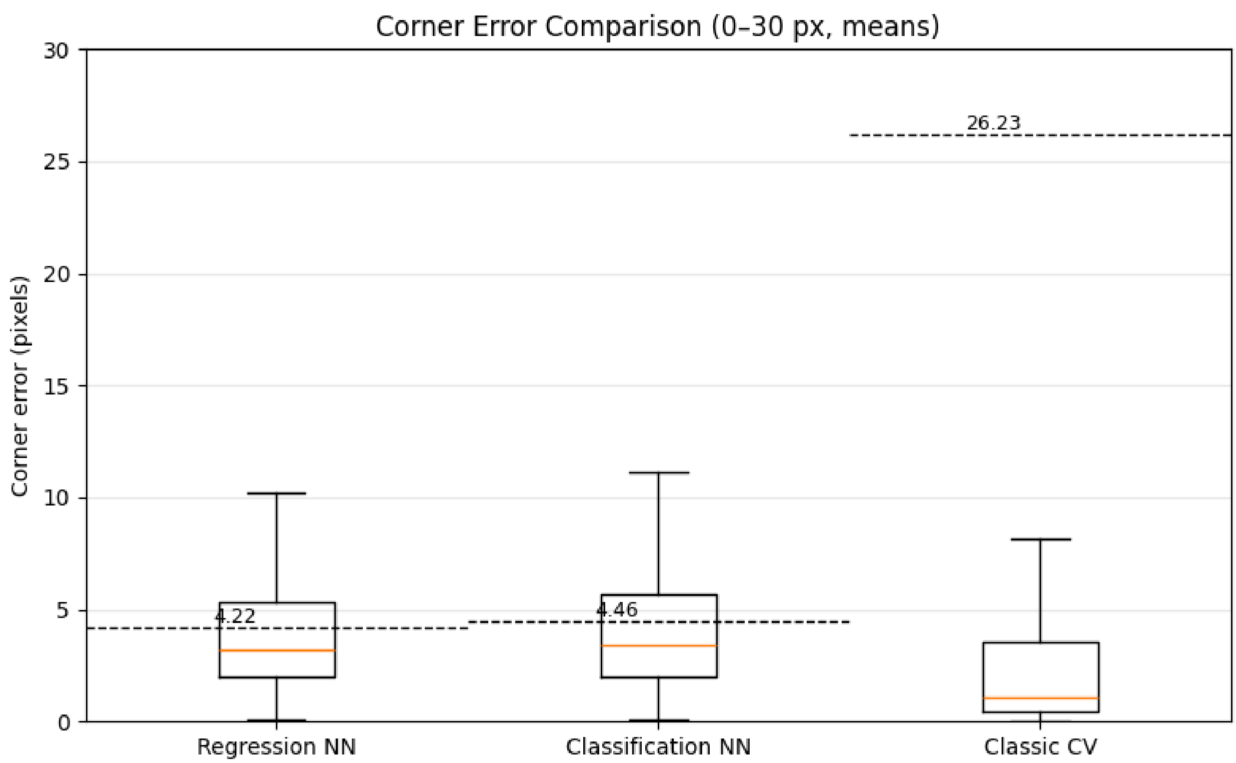
```
Loaded reg model from: checkpoints_google/homography_reg_best.pt
Loaded cls model from: checkpoints_google/homography_cls_best.pt
Evaluating 200 samples (NN + Classic)...

Regression RMSE:      4.2167 px
Classification RMSE:  4.4601 px
Classic CV RMSE:      26.2343 px
Histogram clipped at P99 = 204.34px
```




```
/var/folders/64/6wxk92sj3px37_tfkmrktpf40000gq/T/ipykernel_34169/952850076.py:8
1: MatplotlibDeprecationWarning: The 'labels' parameter of boxplot() has been r
enamed 'tick_labels' since Matplotlib 3.9; support for the old name will be dro
pped in 3.11.
  plt.boxplot([errors_reg, errors_cls, errors_cv],
```




```
/var/folders/64/6wxk92sj3px37_tfkmrktpf40000gq/T/ipykernel_34169/952850076.py:9
4: MatplotlibDeprecationWarning: The 'labels' parameter of boxplot() has been r
enamed 'tick_labels' since Matplotlib 3.9; support for the old name will be dro
pped in 3.11.
  plt.boxplot([errors_reg, errors_cls, errors_cv],
```



Rezultati kažejo, da nevronska mreža (NN) deluje bistveno bolje od klasične metode. NN ima povprečno napako 4.4 piksla, klasična metoda pa 26.2 piksla in veliko ekstremnih vrednosti, kar pomeni več napačnih poravnav. Kljub temu ima klasična metoda dober median (≈1.8 piksla), kar pomeni, da v nekaterih primerih deluje zelo natančno. Na splošno pa je NN natančnejša in stabilnejša rešitev za ocenjevanje homografije.