```python
LightGlue/benchmark.py
# Benchmark script for LightGlue on real images
import argparse
import time
from collections import defaultdict
from pathlib import Path

import matplotlib.pyplot as plt
import numpy as np
import torch
import torch._dynamo

from lightglue import LightGlue, SuperPoint
from lightglue.utils import load_image

torch.set_grad_enabled(False)


def measure(matcher, data, device="cuda", r=100):
    timings = np.zeros((r, 1))
    if device.type == "cuda":
        starter = torch.cuda.Event(enable_timing=True)
        ender = torch.cuda.Event(enable_timing=True)
    # warmup
    for _ in range(10):
        _ = matcher(data)
    # measurements
    with torch.no_grad():
        for rep in range(r):
            if device.type == "cuda":
                starter.record()
                _ = matcher(data)
                ender.record()
                # sync gpu
                torch.cuda.synchronize()
                curr_time = starter.elapsed_time(ender)
            else:
                start = time.perf_counter()
                _ = matcher(data)
                curr_time = (time.perf_counter() - start) * 1e3
            timings[rep] = curr_time
    mean_syn = np.sum(timings) / r
    std_syn = np.std(timings)
    return {"mean": mean_syn, "std": std_syn}


def print_as_table(d, title, cnames):
    print()
    header = f"{title:30} " + " ".join([f"{x:>7}" for x in cnames])
    print(header)
    print("-" * len(header))
    for k, l in d.items():
        print(f"{k:30}", " ".join([f"{x:>7.1f}" for x in l]))


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Benchmark script for LightGlue")
    parser.add_argument(
        "--device",
        choices=["auto", "cuda", "cpu", "mps"],
        default="auto",
        help="device to benchmark on",
    )
    parser.add_argument("--compile", action="store_true", help="Compile LightGlue runs")
    parser.add_argument(
        "--no_flash", action="store_true", help="disable FlashAttention"
    )
    parser.add_argument(
        "--no_prune_thresholds",
        action="store_true",
        help="disable pruning thresholds (i.e. always do pruning)",
    )
    parser.add_argument(
        "--add_superglue",
        action="store_true",
        help="add SuperGlue to the benchmark (requires hloc)",
    )
    parser.add_argument(
        "--measure", default="time", choices=["time", "log-time", "throughput"]
    )
    parser.add_argument(
        "--repeat", "--r", type=int, default=100, help="repetitions of measurements"
    )
```

```python
    parser.add_argument(
        "--num_keypoints",
        nargs="+",
        type=int,
        default=[256, 512, 1024, 2048, 4096],
        help="number of keypoints (list separated by spaces)",
    )
    parser.add_argument(
        "--matmul_precision", default="highest", choices=["highest", "high", "medium"]
    )
    parser.add_argument(
        "--save", default=None, type=str, help="path where figure should be saved"
    )
    args = parser.parse_intermixed_args()

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    if args.device != "auto":
        device = torch.device(args.device)

    print("Running benchmark on device:", device)

    images = Path("assets")
    inputs = {
        "easy": (
            load_image(images / "DSC_0411.JPG"),
            load_image(images / "DSC_0410.JPG"),
        ),
        "difficult": (
            load_image(images / "sacre_coeur1.jpg"),
            load_image(images / "sacre_coeur2.jpg"),
        ),
    }

    configs = {
        "LightGlue-full": {
            "depth_confidence": -1,
            "width_confidence": -1,
        },
        # 'LG-prune': {
        #     'width_confidence': -1,
        # },
        # 'LG-depth': {
        #     'depth_confidence': -1,
        # },
        "LightGlue-adaptive": {},
    }

    if args.compile:
        configs = {**configs, **{k + "-compile": v for k, v in configs.items()}}

    sg_configs = {
        # 'SuperGlue': {},
        "SuperGlue-fast": {"sinkhorn_iterations": 5}
    }

    torch.set_float32_matmul_precision(args.matmul_precision)

    results = {k: defaultdict(list) for k, v in inputs.items()}

    extractor = SuperPoint(max_num_keypoints=None, detection_threshold=-1)
    extractor = extractor.eval().to(device)
    figsize = (len(inputs) * 4.5, 4.5)
    fig, axes = plt.subplots(1, len(inputs), sharey=True, figsize=figsize)
    axes = axes if len(inputs) > 1 else [axes]
    fig.canvas.manager.set_window_title(f"LightGlue benchmark ({device.type})")

    for title, ax in zip(inputs.keys(), axes):
        ax.set_xscale("log", base=2)
        bases = [2**x for x in range(7, 16)]
        ax.set_xticks(bases, bases)
        ax.grid(which="major")
        if args.measure == "log-time":
            ax.set_yscale("log")
            yticks = [10**x for x in range(6)]
            ax.set_yticks(yticks, yticks)
            mpos = [10**x * i for x in range(6) for i in range(2, 10)]
            mlabel = [
                10**x * i if i in [2, 5] else None
                for x in range(6)
                for i in range(2, 10)
            ]
            ax.set_yticks(mpos, mlabel, minor=True)
            ax.grid(which="minor", linewidth=0.2)
        ax.set_title(title)

```

```
        ax.set_xlabel("# keypoints")
        if args.measure == "throughput":
            ax.set_ylabel("Throughput [pairs/s]")
        else:
            ax.set_ylabel("Latency [ms]")

    for name, conf in configs.items():
        print("Run benchmark for:", name)
        torch.cuda.empty_cache()
        matcher = LightGlue(features="superpoint", flash=not args.no_flash, **conf)
        if args.no_prune_thresholds:
            matcher.pruning_keypoint_thresholds = {
                k: -1 for k in matcher.pruning_keypoint_thresholds
            }
        matcher = matcher.eval().to(device)
        if name.endswith("compile"):
            import torch._dynamo

            torch._dynamo.reset()  # avoid buffer overflow
            matcher.compile()
        for pair_name, ax in zip(inputs.keys(), axes):
            image0, image1 = [x.to(device) for x in inputs[pair_name]]
            runtimes = []
            for num_kpts in args.num_keypoints:
                extractor.conf.max_num_keypoints = num_kpts
                feats0 = extractor.extract(image0)
                feats1 = extractor.extract(image1)
                runtime = measure(
                    matcher,
                    {"image0": feats0, "image1": feats1},
                    device=device,
                    r=args.repeat,
                )["mean"]
                results[pair_name][name].append(
                    1000 / runtime if args.measure == "throughput" else runtime
                )
            ax.plot(
                args.num_keypoints, results[pair_name][name], label=name, marker="o"
            )
        del matcher, feats0, feats1

    if args.add_superglue:
        from hloc.matchers.superglue import SuperGlue

        for name, conf in sg_configs.items():
            print("Run benchmark for:", name)
            matcher = SuperGlue(conf)
            matcher = matcher.eval().to(device)
            for pair_name, ax in zip(inputs.keys(), axes):
                image0, image1 = [x.to(device) for x in inputs[pair_name]]
                runtimes = []
                for num_kpts in args.num_keypoints:
                    extractor.conf.max_num_keypoints = num_kpts
                    feats0 = extractor.extract(image0)
                    feats1 = extractor.extract(image1)
                    data = {
                        "image0": image0[None],
                        "image1": image1[None],
                        **{k + "0": v for k, v in feats0.items()},
                        **{k + "1": v for k, v in feats1.items()},
                    }
                    data["scores0"] = data["keypoint_scores0"]
                    data["scores1"] = data["keypoint_scores1"]
                    data["descriptors0"] = (
                        data["descriptors0"].transpose(-1, -2).contiguous()
                    )
                    data["descriptors1"] = (
                        data["descriptors1"].transpose(-1, -2).contiguous()
                    )
                    runtime = measure(matcher, data, device=device, r=args.repeat)[
                        "mean"
                    ]
                    results[pair_name][name].append(
                        1000 / runtime if args.measure == "throughput" else runtime
                    )
                ax.plot(
                    args.num_keypoints, results[pair_name][name], label=name, marker="o"
                )
            del matcher, data, image0, image1, feats0, feats1

    for name, runtimes in results.items():
        print_as_table(runtimes, name, args.num_keypoints)

    axes[0].legend()
    fig.tight_layout()
```

```
253     if args.save:
254         plt.savefig(args.save, dpi=fig.dpi)
255     plt.show()
```

```
0  LightGlue/demo.py
1  # If we are on colab: this clones the repo and installs the dependencies
2  from pathlib import Path
3
4  # if Path.cwd().name != "LightGlue":
5  #     !git clone --quiet https://github.com/cvg/LightGlue/
6  #     %cd LightGlue
7  #     !pip install --progress-bar off --quiet -e .
8
9  from lightglue import LightGlue, SuperPoint, DISK
10 from lightglue.utils import load_image, rbd
11 from lightglue import viz2d
12 import torch
13
14 torch.set_grad_enabled(False)
15 images = Path("assets")
16
17 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")  # 'mps', 'cpu'
18
19 extractor = SuperPoint(max_num_keypoints=2048).eval().to(device)  # load the extractor
20 matcher = LightGlue(features="superpoint").eval().to(device)
21
22 image0 = load_image(images / "DSC_0411.JPG")
23 image1 = load_image(images / "DSC_0410.JPG")
24
25 feats0 = extractor.extract(image0.to(device))
26 feats1 = extractor.extract(image1.to(device))
27 matches01 = matcher({"image0": feats0, "image1": feats1})
28 feats0, feats1, matches01 = [
29     rbd(x) for x in [feats0, feats1, matches01]
30 ]  # remove batch dimension
31
32 kpts0, kpts1, matches = feats0["keypoints"], feats1["keypoints"], matches01["matches"]
33 m_kpts0, m_kpts1 = kpts0[matches[..., 0]], kpts1[matches[..., 1]]
34
35 axes = viz2d.plot_images([image0, image1])
36 viz2d.plot_matches(m_kpts0, m_kpts1, color="lime", lw=0.2)
37 viz2d.add_text(0, f'Stop after {matches01["stop"]} layers', fs=20)
38
39 kpc0, kpc1 = viz2d.cm_prune(matches01["prune0"]), viz2d.cm_prune(matches01["prune1"])
40 viz2d.plot_images([image0, image1])
41 viz2d.plot_keypoints([kpts0, kpts1], colors=[kpc0, kpc1], ps=10)
```

```
LightGlue/lightglue/aliked.py
# BSD 3-Clause License

# Copyright (c) 2022, Zhao Xiaoming
# All rights reserved.

# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions are met:

# 1. Redistributions of source code must retain the above copyright notice, this
#    list of conditions and the following disclaimer.

# 2. Redistributions in binary form must reproduce the above copyright notice,
#    this list of conditions and the following disclaimer in the documentation
#    and/or other materials provided with the distribution.

# 3. Neither the name of the copyright holder nor the names of its
#    contributors may be used to endorse or promote products derived from
#    this software without specific prior written permission.

# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
# AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
# IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
# DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
# FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
# DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
# SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
# CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
# OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
# OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

# Authors:
# Xiaoming Zhao, Xingming Wu, Weihai Chen, Peter C.Y. Chen, Qingsong Xu, and Zhengguo Li
# Code from https://github.com/Shiaoming/ALIKED

from typing import Callable, Optional

import torch
import torch.nn.functional as F
import torchvision
from kornia.color import grayscale_to_rgb
from torch import nn
from torch.nn.modules.utils import _pair
from torchvision.models import resnet

from .utils import Extractor


def get_patches(
    tensor: torch.Tensor, required_corners: torch.Tensor, ps: int
) -> torch.Tensor:
    c, h, w = tensor.shape
    corner = (required_corners - ps / 2 + 1).long()
    corner[:, 0] = corner[:, 0].clamp(min=0, max=w - 1 - ps)
    corner[:, 1] = corner[:, 1].clamp(min=0, max=h - 1 - ps)
    offset = torch.arange(0, ps)

    kw = {"indexing": "ij"} if torch.__version__ >= "1.10" else {}
    x, y = torch.meshgrid(offset, offset, **kw)
    patches = torch.stack((x, y)).permute(2, 1, 0).unsqueeze(2)
    patches = patches.to(corner) + corner[None, None]
    pts = patches.reshape(-1, 2)
    sampled = tensor.permute(1, 2, 0)[tuple(pts.T)[::-1]]
    sampled = sampled.reshape(ps, ps, -1, c)
    assert sampled.shape[:3] == patches.shape[:3]
    return sampled.permute(2, 3, 0, 1)


def simple_nms(scores: torch.Tensor, nms_radius: int):
    """Fast Non-maximum suppression to remove nearby points"""

    zeros = torch.zeros_like(scores)
    max_mask = scores == torch.nn.functional.max_pool2d(
        scores, kernel_size=nms_radius * 2 + 1, stride=1, padding=nms_radius
    )

    for _ in range(2):
        supp_mask = (
            torch.nn.functional.max_pool2d(
                max_mask.float(),
                kernel_size=nms_radius * 2 + 1,
                stride=1,
                padding=nms_radius,
```

```
            )
            > 0
        )
        supp_scores = torch.where(supp_mask, zeros, scores)
        new_max_mask = supp_scores == torch.nn.functional.max_pool2d(
            supp_scores, kernel_size=nms_radius * 2 + 1, stride=1, padding=nms_radius
        )
        max_mask = max_mask | (new_max_mask & (~supp_mask))
    return torch.where(max_mask, scores, zeros)


class DKD(nn.Module):
    def __init__(
        self,
        radius: int = 2,
        top_k: int = 0,
        scores_th: float = 0.2,
        n_limit: int = 20000,
    ):
        """
        Args:
            radius: soft detection radius, kernel size is (2 * radius + 1)
            top_k: top_k > 0: return top k keypoints
            scores_th: top_k <= 0 threshold mode:
                scores_th > 0: return keypoints with scores>scores_th
                else: return keypoints with scores > scores.mean()
            n_limit: max number of keypoint in threshold mode
        """
        super().__init__()
        self.radius = radius
        self.top_k = top_k
        self.scores_th = scores_th
        self.n_limit = n_limit
        self.kernel_size = 2 * self.radius + 1
        self.temperature = 0.1  # tuned temperature
        self.unfold = nn.Unfold(kernel_size=self.kernel_size, padding=self.radius)
        # local xy grid
        x = torch.linspace(-self.radius, self.radius, self.kernel_size)
        # (kernel_size*kernel_size) x 2 : (w,h)
        kw = {"indexing": "ij"} if torch.__version__ >= "1.10" else {}
        self.hw_grid = (
            torch.stack(torch.meshgrid([x, x], **kw)).view(2, -1).t()[:, [1, 0]]
        )

    def forward(
        self,
        scores_map: torch.Tensor,
        sub_pixel: bool = True,
        image_size: Optional[torch.Tensor] = None,
    ):
        """
        :param scores_map: Bx1xHxW
        :param descriptor_map: BxCxHxW
        :param sub_pixel: whether to use sub-pixel keypoint detection
        :return: kpts: list[Nx2,...]; kptscores: list[N,....] normalised position: -1~1
        """
        b, c, h, w = scores_map.shape
        scores_nograd = scores_map.detach()
        nms_scores = simple_nms(scores_nograd, self.radius)

        # remove border
        nms_scores[:, :, : self.radius, :] = 0
        nms_scores[:, :, :, : self.radius] = 0
        if image_size is not None:
            for i in range(scores_map.shape[0]):
                w, h = image_size[i].long()
                nms_scores[i, :, h.item() - self.radius :, :] = 0
                nms_scores[i, :, :, w.item() - self.radius :] = 0
        else:
            nms_scores[:, :, -self.radius :, :] = 0
            nms_scores[:, :, :, -self.radius :] = 0

        # detect keypoints without grad
        if self.top_k > 0:
            topk = torch.topk(nms_scores.view(b, -1), self.top_k)
            indices_keypoints = [topk.indices[i] for i in range(b)]  # B x top_k
        else:
            if self.scores_th > 0:
                masks = nms_scores > self.scores_th
                if masks.sum() == 0:
                    th = scores_nograd.reshape(b, -1).mean(dim=1)  # th = self.scores_th
                    masks = nms_scores > th.reshape(b, 1, 1, 1)
            else:
                th = scores_nograd.reshape(b, -1).mean(dim=1)  # th = self.scores_th
                masks = nms_scores > th.reshape(b, 1, 1, 1)
```

```
            masks = masks.reshape(b, -1)

            indices_keypoints = []  # list, B x (any size)
            scores_view = scores_nograd.reshape(b, -1)
            for mask, scores in zip(masks, scores_view):
                indices = mask.nonzero()[:, 0]
                if len(indices) > self.n_limit:
                    kpts_sc = scores[indices]
                    sort_idx = kpts_sc.sort(descending=True)[1]
                    sel_idx = sort_idx[: self.n_limit]
                    indices = indices[sel_idx]
                indices_keypoints.append(indices)

        wh = torch.tensor([w - 1, h - 1], device=scores_nograd.device)

        keypoints = []
        scoredispersitys = []
        kptscores = []
        if sub_pixel:
            # detect soft keypoints with grad backpropagation
            patches = self.unfold(scores_map)  # B x (kernel**2) x (H*W)
            self.hw_grid = self.hw_grid.to(scores_map)  # to device
            for b_idx in range(b):
                patch = patches[b_idx].t()  # (H*W) x (kernel**2)
                indices_kpt = indices_keypoints[
                    b_idx
                ]  # one dimension vector, say its size is M
                patch_scores = patch[indices_kpt]  # M x (kernel**2)
                keypoints_xy_nms = torch.stack(
                    [indices_kpt % w, torch.div(indices_kpt, w, rounding_mode="trunc")],
                    dim=1,
                )  # Mx2

                # max is detached to prevent undesired backprop loops in the graph
                max_v = patch_scores.max(dim=1).values.detach()[:, None]
                x_exp = (
                    (patch_scores - max_v) / self.temperature
                ).exp()  # M * (kernel**2), in [0, 1]

                # \frac{ \sum{(i,j) \times \exp(x/T)} }{ \sum{\exp(x/T)} }
                xy_residual = (
                    x_exp @ self.hw_grid / x_exp.sum(dim=1)[:, None]
                )  # Soft-argmax, Mx2

                hw_grid_dist2 = (
                    torch.norm(
                        (self.hw_grid[None, :, :] - xy_residual[:, None, :])
                        / self.radius,
                        dim=-1,
                    )
                    ** 2
                )
                scoredispersity = (x_exp * hw_grid_dist2).sum(dim=1) / x_exp.sum(dim=1)

                # compute result keypoints
                keypoints_xy = keypoints_xy_nms + xy_residual
                keypoints_xy = keypoints_xy / wh * 2 - 1  # (w,h) -> (-1~1,-1~1)

                kptscore = torch.nn.functional.grid_sample(
                    scores_map[b_idx].unsqueeze(0),
                    keypoints_xy.view(1, 1, -1, 2),
                    mode="bilinear",
                    align_corners=True,
                )[
                    0, 0, 0, :
                ]  # CxN

                keypoints.append(keypoints_xy)
                scoredispersitys.append(scoredispersity)
                kptscores.append(kptscore)
        else:
            for b_idx in range(b):
                indices_kpt = indices_keypoints[
                    b_idx
                ]  # one dimension vector, say its size is M
                # To avoid warning: UserWarning: __floordiv__ is deprecated
                keypoints_xy_nms = torch.stack(
                    [indices_kpt % w, torch.div(indices_kpt, w, rounding_mode="trunc")],
                    dim=1,
                )  # Mx2
                keypoints_xy = keypoints_xy_nms / wh * 2 - 1  # (w,h) -> (-1~1,-1~1)
                kptscore = torch.nn.functional.grid_sample(
                    scores_map[b_idx].unsqueeze(0),
                    keypoints_xy.view(1, 1, -1, 2),
                    mode="bilinear",
```

```
                    align_corners=True,
                )[
                    0, 0, 0, :
                ]  # CxN
                keypoints.append(keypoints_xy)
                scoredispersitys.append(kptscore)  # for jit.script compatability
                kptscores.append(kptscore)

        return keypoints, scoredispersitys, kptscores


class InputPadder(object):
    """Pads images such that dimensions are divisible by 8"""

    def __init__(self, h: int, w: int, divis_by: int = 8):
        self.ht = h
        self.wd = w
        pad_ht = (((self.ht // divis_by) + 1) * divis_by - self.ht) % divis_by
        pad_wd = (((self.wd // divis_by) + 1) * divis_by - self.wd) % divis_by
        self._pad = [
            pad_wd // 2,
            pad_wd - pad_wd // 2,
            pad_ht // 2,
            pad_ht - pad_ht // 2,
        ]

    def pad(self, x: torch.Tensor):
        assert x.ndim == 4
        return F.pad(x, self._pad, mode="replicate")

    def unpad(self, x: torch.Tensor):
        assert x.ndim == 4
        ht = x.shape[-2]
        wd = x.shape[-1]
        c = [self._pad[2], ht - self._pad[3], self._pad[0], wd - self._pad[1]]
        return x[..., c[0] : c[1], c[2] : c[3]]


class DeformableConv2d(nn.Module):
    def __init__(
        self,
        in_channels,
        out_channels,
        kernel_size=3,
        stride=1,
        padding=1,
        bias=False,
        mask=False,
    ):
        super(DeformableConv2d, self).__init__()

        self.padding = padding
        self.mask = mask

        self.channel_num = (
            3 * kernel_size * kernel_size if mask else 2 * kernel_size * kernel_size
        )
        self.offset_conv = nn.Conv2d(
            in_channels,
            self.channel_num,
            kernel_size=kernel_size,
            stride=stride,
            padding=self.padding,
            bias=True,
        )

        self.regular_conv = nn.Conv2d(
            in_channels=in_channels,
            out_channels=out_channels,
            kernel_size=kernel_size,
            stride=stride,
            padding=self.padding,
            bias=bias,
        )

    def forward(self, x):
        h, w = x.shape[2:]
        max_offset = max(h, w) / 4.0

        out = self.offset_conv(x)
        if self.mask:
            o1, o2, mask = torch.chunk(out, 3, dim=1)
            offset = torch.cat((o1, o2), dim=1)
            mask = torch.sigmoid(mask)
        else:
```

```
            offset = out
            mask = None
        offset = offset.clamp(-max_offset, max_offset)
        x = torchvision.ops.deform_conv2d(
            input=x,
            offset=offset,
            weight=self.regular_conv.weight,
            bias=self.regular_conv.bias,
            padding=self.padding,
            mask=mask,
        )
        return x


def get_conv(
    inplanes,
    planes,
    kernel_size=3,
    stride=1,
    padding=1,
    bias=False,
    conv_type="conv",
    mask=False,
):
    if conv_type == "conv":
        conv = nn.Conv2d(
            inplanes,
            planes,
            kernel_size=kernel_size,
            stride=stride,
            padding=padding,
            bias=bias,
        )
    elif conv_type == "dcn":
        conv = DeformableConv2d(
            inplanes,
            planes,
            kernel_size=kernel_size,
            stride=stride,
            padding=_pair(padding),
            bias=bias,
            mask=mask,
        )
    else:
        raise TypeError
    return conv


class ConvBlock(nn.Module):
    def __init__(
        self,
        in_channels,
        out_channels,
        gate: Optional[Callable[..., nn.Module]] = None,
        norm_layer: Optional[Callable[..., nn.Module]] = None,
        conv_type: str = "conv",
        mask: bool = False,
    ):
        super().__init__()
        if gate is None:
            self.gate = nn.ReLU(inplace=True)
        else:
            self.gate = gate
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        self.conv1 = get_conv(
            in_channels, out_channels, kernel_size=3, conv_type=conv_type, mask=mask
        )
        self.bn1 = norm_layer(out_channels)
        self.conv2 = get_conv(
            out_channels, out_channels, kernel_size=3, conv_type=conv_type, mask=mask
        )
        self.bn2 = norm_layer(out_channels)

    def forward(self, x):
        x = self.gate(self.bn1(self.conv1(x)))  # B x in_channels x H x W
        x = self.gate(self.bn2(self.conv2(x)))  # B x out_channels x H x W
        return x


# modified based on torchvision\models\resnet.py#27->BasicBlock
class ResBlock(nn.Module):
    expansion: int = 1

    def __init__(
```

```
        self,
        inplanes: int,
        planes: int,
        stride: int = 1,
        downsample: Optional[nn.Module] = None,
        groups: int = 1,
        base_width: int = 64,
        dilation: int = 1,
        gate: Optional[Callable[..., nn.Module]] = None,
        norm_layer: Optional[Callable[..., nn.Module]] = None,
        conv_type: str = "conv",
        mask: bool = False,
    ) -> None:
        super(ResBlock, self).__init__()
        if gate is None:
            self.gate = nn.ReLU(inplace=True)
        else:
            self.gate = gate
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        if groups != 1 or base_width != 64:
            raise ValueError("ResBlock only supports groups=1 and base_width=64")
        if dilation > 1:
            raise NotImplementedError("Dilation > 1 not supported in ResBlock")
        # Both self.conv1 and self.downsample layers
        # downsample the input when stride != 1
        self.conv1 = get_conv(
            inplanes, planes, kernel_size=3, conv_type=conv_type, mask=mask
        )
        self.bn1 = norm_layer(planes)
        self.conv2 = get_conv(
            planes, planes, kernel_size=3, conv_type=conv_type, mask=mask
        )
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.gate(out)

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

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.gate(out)

        return out


class SDDH(nn.Module):
    def __init__(
        self,
        dims: int,
        kernel_size: int = 3,
        n_pos: int = 8,
        gate=nn.ReLU(),
        conv2D=False,
        mask=False,
    ):
        super(SDDH, self).__init__()
        self.kernel_size = kernel_size
        self.n_pos = n_pos
        self.conv2D = conv2D
        self.mask = mask

        self.get_patches_func = get_patches

        # estimate offsets
        self.channel_num = 3 * n_pos if mask else 2 * n_pos
        self.offset_conv = nn.Sequential(
            nn.Conv2d(
                dims,
                self.channel_num,
                kernel_size=kernel_size,
                stride=1,
                padding=0,
                bias=True,
            ),
```

```
            gate,
            nn.Conv2d(
                self.channel_num,
                self.channel_num,
                kernel_size=1,
                stride=1,
                padding=0,
                bias=True,
            ),
        )

        # sampled feature conv
        self.sf_conv = nn.Conv2d(
            dims, dims, kernel_size=1, stride=1, padding=0, bias=False
        )

        # convM
        if not conv2D:
            # deformable desc weights
            agg_weights = torch.nn.Parameter(torch.rand(n_pos, dims, dims))
            self.register_parameter("agg_weights", agg_weights)
        else:
            self.convM = nn.Conv2d(
                dims * n_pos, dims, kernel_size=1, stride=1, padding=0, bias=False
            )

    def forward(self, x, keypoints):
        # x: [B,C,H,W]
        # keypoints: list, [[N_kpts,2], ...] (w,h)
        b, c, h, w = x.shape
        wh = torch.tensor([[w - 1, h - 1]], device=x.device)
        max_offset = max(h, w) / 4.0

        offsets = []
        descriptors = []
        # get offsets for each keypoint
        for ib in range(b):
            xi, kptsi = x[ib], keypoints[ib]
            kptsi_wh = (kptsi / 2 + 0.5) * wh
            N_kpts = len(kptsi)

            if self.kernel_size > 1:
                patch = self.get_patches_func(
                    xi, kptsi_wh.long(), self.kernel_size
                )  # [N_kpts, C, K, K]
            else:
                kptsi_wh_long = kptsi_wh.long()
                patch = (
                    xi[:, kptsi_wh_long[:, 1], kptsi_wh_long[:, 0]]
                    .permute(1, 0)
                    .reshape(N_kpts, c, 1, 1)
                )

            offset = self.offset_conv(patch).clamp(
                -max_offset, max_offset
            )  # [N_kpts, 2*n_pos, 1, 1]
            if self.mask:
                offset = (
                    offset[:, :, 0, 0].view(N_kpts, 3, self.n_pos).permute(0, 2, 1)
                )  # [N_kpts, n_pos, 3]
                offset = offset[:, :, :-1]  # [N_kpts, n_pos, 2]
                mask_weight = torch.sigmoid(offset[:, :, -1])  # [N_kpts, n_pos]
            else:
                offset = (
                    offset[:, :, 0, 0].view(N_kpts, 2, self.n_pos).permute(0, 2, 1)
                )  # [N_kpts, n_pos, 2]
            offsets.append(offset)  # for visualization

            # get sample positions
            pos = kptsi_wh.unsqueeze(1) + offset  # [N_kpts, n_pos, 2]
            pos = 2.0 * pos / wh[None] - 1
            pos = pos.reshape(1, N_kpts * self.n_pos, 1, 2)

            # sample features
            features = F.grid_sample(
                xi.unsqueeze(0), pos, mode="bilinear", align_corners=True
            )  # [1,C,(N_kpts*n_pos),1]
            features = features.reshape(c, N_kpts, self.n_pos, 1).permute(
                1, 0, 2, 3
            )  # [N_kpts, C, n_pos, 1]
            if self.mask:
                features = torch.einsum("ncpo,np->ncpo", features, mask_weight)

            features = torch.selu_(self.sf_conv(features)).squeeze(
                -1
```

```
            )  # [N_kpts, C, n_pos]
            # convM
            if not self.conv2D:
                descs = torch.einsum(
                    "ncp,pcd->nd", features, self.agg_weights
                )  # [N_kpts, C]
            else:
                features = features.reshape(N_kpts, -1)[
                    :, :, None, None
                ]  # [N_kpts, C*n_pos, 1, 1]
                descs = self.convM(features).squeeze()  # [N_kpts, C]

            # normalize
            descs = F.normalize(descs, p=2.0, dim=1)
            descriptors.append(descs)

        return descriptors, offsets


class ALIKED(Extractor):
    default_conf = {
        "model_name": "aliked-n16",
        "max_num_keypoints": -1,
        "detection_threshold": 0.2,
        "nms_radius": 2,
    }

    checkpoint_url = "https://github.com/Shiaoming/ALIKED/raw/main/models/{}.pth"

    n_limit_max = 20000

    # c1, c2, c3, c4, dim, K, M
    cfgs = {
        "aliked-t16": [8, 16, 32, 64, 64, 3, 16],
        "aliked-n16": [16, 32, 64, 128, 128, 3, 16],
        "aliked-n16rot": [16, 32, 64, 128, 128, 3, 16],
        "aliked-n32": [16, 32, 64, 128, 128, 3, 32],
    }
    preprocess_conf = {
        "resize": 1024,
    }

    required_data_keys = ["image"]

    def __init__(self, **conf):
        super().__init__(**conf)  # Update with default configuration.
        conf = self.conf
        c1, c2, c3, c4, dim, K, M = self.cfgs[conf.model_name]
        conv_types = ["conv", "conv", "dcn", "dcn"]
        conv2D = False
        mask = False

        # build model
        self.pool2 = nn.AvgPool2d(kernel_size=2, stride=2)
        self.pool4 = nn.AvgPool2d(kernel_size=4, stride=4)
        self.norm = nn.BatchNorm2d
        self.gate = nn.SELU(inplace=True)
        self.block1 = ConvBlock(3, c1, self.gate, self.norm, conv_type=conv_types[0])
        self.block2 = self.get_resblock(c1, c2, conv_types[1], mask)
        self.block3 = self.get_resblock(c2, c3, conv_types[2], mask)
        self.block4 = self.get_resblock(c3, c4, conv_types[3], mask)

        self.conv1 = resnet.conv1x1(c1, dim // 4)
        self.conv2 = resnet.conv1x1(c2, dim // 4)
        self.conv3 = resnet.conv1x1(c3, dim // 4)
        self.conv4 = resnet.conv1x1(dim, dim // 4)
        self.upsample2 = nn.Upsample(
            scale_factor=2, mode="bilinear", align_corners=True
        )
        self.upsample4 = nn.Upsample(
            scale_factor=4, mode="bilinear", align_corners=True
        )
        self.upsample8 = nn.Upsample(
            scale_factor=8, mode="bilinear", align_corners=True
        )
        self.upsample32 = nn.Upsample(
            scale_factor=32, mode="bilinear", align_corners=True
        )
        self.score_head = nn.Sequential(
            resnet.conv1x1(dim, 8),
            self.gate,
            resnet.conv3x3(8, 4),
            self.gate,
            resnet.conv3x3(4, 4),
            self.gate,
```

```
            resnet.conv3x3(4, 1),
        )
        self.desc_head = SDDH(dim, K, M, gate=self.gate, conv2D=conv2D, mask=mask)
        self.dkd = DKD(
            radius=conf.nms_radius,
            top_k=-1 if conf.detection_threshold > 0 else conf.max_num_keypoints,
            scores_th=conf.detection_threshold,
            n_limit=conf.max_num_keypoints
            if conf.max_num_keypoints > 0
            else self.n_limit_max,
        )

        state_dict = torch.hub.load_state_dict_from_url(
            self.checkpoint_url.format(conf.model_name), map_location="cpu"
        )
        self.load_state_dict(state_dict, strict=True)

    def get_resblock(self, c_in, c_out, conv_type, mask):
        return ResBlock(
            c_in,
            c_out,
            1,
            nn.Conv2d(c_in, c_out, 1),
            gate=self.gate,
            norm_layer=self.norm,
            conv_type=conv_type,
            mask=mask,
        )

    def extract_dense_map(self, image):
        # Pads images such that dimensions are divisible by
        div_by = 2**5
        padder = InputPadder(image.shape[-2], image.shape[-1], div_by)
        image = padder.pad(image)

        # ================================== feature encoder
        x1 = self.block1(image)  # B x c1 x H x W
        x2 = self.pool2(x1)
        x2 = self.block2(x2)  # B x c2 x H/2 x W/2
        x3 = self.pool4(x2)
        x3 = self.block3(x3)  # B x c3 x H/8 x W/8
        x4 = self.pool4(x3)
        x4 = self.block4(x4)  # B x dim x H/32 x W/32
        # ================================== feature aggregation
        x1 = self.gate(self.conv1(x1))  # B x dim//4 x H x W
        x2 = self.gate(self.conv2(x2))  # B x dim//4 x H//2 x W//2
        x3 = self.gate(self.conv3(x3))  # B x dim//4 x H//8 x W//8
        x4 = self.gate(self.conv4(x4))  # B x dim//4 x H//32 x W//32
        x2_up = self.upsample2(x2)  # B x dim//4 x H x W
        x3_up = self.upsample8(x3)  # B x dim//4 x H x W
        x4_up = self.upsample32(x4)  # B x dim//4 x H x W
        x1234 = torch.cat([x1, x2_up, x3_up, x4_up], dim=1)
        # ================================== score head
        score_map = torch.sigmoid(self.score_head(x1234))
        feature_map = torch.nn.functional.normalize(x1234, p=2, dim=1)

        # Unpads images
        feature_map = padder.unpad(feature_map)
        score_map = padder.unpad(score_map)

        return feature_map, score_map

    def forward(self, data: dict) -> dict:
        image = data["image"]
        if image.shape[1] == 1:
            image = grayscale_to_rgb(image)
        feature_map, score_map = self.extract_dense_map(image)
        keypoints, kptscores, scoredispersitys = self.dkd(
            score_map, image_size=data.get("image_size")
        )
        descriptors, offsets = self.desc_head(feature_map, keypoints)

        _, _, h, w = image.shape
        wh = torch.tensor([w - 1, h - 1], device=image.device)
        # no padding required
        # we can set detection_threshold=-1 and conf.max_num_keypoints > 0
        return {
            "keypoints": wh * (torch.stack(keypoints) + 1) / 2.0,  # B x N x 2
            "descriptors": torch.stack(descriptors),  # B x N x D
            "keypoint_scores": torch.stack(kptscores),  # B x N
        }
```

```python
LightGlue/lightglue/disk.py
import kornia
import torch

from .utils import Extractor


class DISK(Extractor):
    default_conf = {
        "weights": "depth",
        "max_num_keypoints": None,
        "desc_dim": 128,
        "nms_window_size": 5,
        "detection_threshold": 0.0,
        "pad_if_not_divisible": True,
    }

    preprocess_conf = {
        "resize": 1024,
        "grayscale": False,
    }

    required_data_keys = ["image"]

    def __init__(self, **conf) -> None:
        super().__init__(**conf)  # Update with default configuration.
        self.model = kornia.feature.DISK.from_pretrained(self.conf.weights)

    def forward(self, data: dict) -> dict:
        """Compute keypoints, scores, descriptors for image"""
        for key in self.required_data_keys:
            assert key in data, f"Missing key {key} in data"
        image = data["image"]
        if image.shape[1] == 1:
            image = kornia.color.grayscale_to_rgb(image)
        features = self.model(
            image,
            n=self.conf.max_num_keypoints,
            window_size=self.conf.nms_window_size,
            score_threshold=self.conf.detection_threshold,
            pad_if_not_divisible=self.conf.pad_if_not_divisible,
        )
        keypoints = [f.keypoints for f in features]
        scores = [f.detection_scores for f in features]
        descriptors = [f.descriptors for f in features]
        del features

        keypoints = torch.stack(keypoints, 0)
        scores = torch.stack(scores, 0)
        descriptors = torch.stack(descriptors, 0)

        return {
            "keypoints": keypoints.to(image).contiguous(),
            "keypoint_scores": scores.to(image).contiguous(),
            "descriptors": descriptors.to(image).contiguous(),
        }
```

```python
LightGlue/lightglue/dog_hardnet.py
import torch
from kornia.color import rgb_to_grayscale
from kornia.feature import HardNet, LAFDescriptor, laf_from_center_scale_ori

from .sift import SIFT


class DoGHardNet(SIFT):
    required_data_keys = ["image"]

    def __init__(self, **conf):
        super().__init__(**conf)
        self.laf_desc = LAFDescriptor(HardNet(True)).eval()

    def forward(self, data: dict) -> dict:
        image = data["image"]
        if image.shape[1] == 3:
            image = rgb_to_grayscale(image)
        device = image.device
        self.laf_desc = self.laf_desc.to(device)
        self.laf_desc.descriptor = self.laf_desc.descriptor.eval()
        pred = []
        if "image_size" in data.keys():
            im_size = data.get("image_size").long()
        else:
            im_size = None
        for k in range(len(image)):
            img = image[k]
            if im_size is not None:
                w, h = data["image_size"][k]
                img = img[:, : h.to(torch.int32), : w.to(torch.int32)]
            p = self.extract_single_image(img)
            lafs = laf_from_center_scale_ori(
                p["keypoints"].reshape(1, -1, 2),
                6.0 * p["scales"].reshape(1, -1, 1, 1),
                torch.rad2deg(p["oris"]).reshape(1, -1, 1),
            ).to(device)
            p["descriptors"] = self.laf_desc(img[None], lafs).reshape(-1, 128)
            pred.append(p)
        pred = {k: torch.stack([p[k] for p in pred], 0).to(device) for k in pred[0]}
        return pred
```

```python
LightGlue/lightglue/lightglue.py
import warnings
from pathlib import Path
from types import SimpleNamespace
from typing import Callable, List, Optional, Tuple

import numpy as np
import torch
import torch.nn.functional as F
from torch import nn

try:
    from flash_attn.modules.mha import FlashCrossAttention
except ModuleNotFoundError:
    FlashCrossAttention = None

if FlashCrossAttention or hasattr(F, "scaled_dot_product_attention"):
    FLASH_AVAILABLE = True
else:
    FLASH_AVAILABLE = False

torch.backends.cudnn.deterministic = True


@torch.cuda.amp.custom_fwd(cast_inputs=torch.float32)
def normalize_keypoints(
    kpts: torch.Tensor, size: Optional[torch.Tensor] = None
) -> torch.Tensor:
    if size is None:
        size = 1 + kpts.max(-2).values - kpts.min(-2).values
    elif not isinstance(size, torch.Tensor):
        size = torch.tensor(size, device=kpts.device, dtype=kpts.dtype)
    size = size.to(kpts)
    shift = size / 2
    scale = size.max(-1).values / 2
    kpts = (kpts - shift[..., None, :]) / scale[..., None, None]
    return kpts


def pad_to_length(x: torch.Tensor, length: int) -> Tuple[torch.Tensor]:
    if length <= x.shape[-2]:
        return x, torch.ones_like(x[..., :1], dtype=torch.bool)
    pad = torch.ones(
        *x.shape[:-2], length - x.shape[-2], x.shape[-1], device=x.device, dtype=x.dtype
    )
    y = torch.cat([x, pad], dim=-2)
    mask = torch.zeros(*y.shape[:-1], 1, dtype=torch.bool, device=x.device)
    mask[..., : x.shape[-2], :] = True
    return y, mask


def rotate_half(x: torch.Tensor) -> torch.Tensor:
    x = x.unflatten(-1, (-1, 2))
    x1, x2 = x.unbind(dim=-1)
    return torch.stack((-x2, x1), dim=-1).flatten(start_dim=-2)


def apply_cached_rotary_emb(freqs: torch.Tensor, t: torch.Tensor) -> torch.Tensor:
    return (t * freqs[0]) + (rotate_half(t) * freqs[1])


class LearnableFourierPositionalEncoding(nn.Module):
    def __init__(self, M: int, dim: int, F_dim: int = None, gamma: float = 1.0) -> None:
        super().__init__()
        F_dim = F_dim if F_dim is not None else dim
        self.gamma = gamma
        self.Wr = nn.Linear(M, F_dim // 2, bias=False)
        nn.init.normal_(self.Wr.weight.data, mean=0, std=self.gamma**-2)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """encode position vector"""
        projected = self.Wr(x)
        cosines, sines = torch.cos(projected), torch.sin(projected)
        emb = torch.stack([cosines, sines], 0).unsqueeze(-3)
        return emb.repeat_interleave(2, dim=-1)


class TokenConfidence(nn.Module):
    def __init__(self, dim: int) -> None:
        super().__init__()
        self.token = nn.Sequential(nn.Linear(dim, 1), nn.Sigmoid())

    def forward(self, desc0: torch.Tensor, desc1: torch.Tensor):
```

```
        """get confidence tokens"""
        return (
            self.token(desc0.detach()).squeeze(-1),
            self.token(desc1.detach()).squeeze(-1),
        )


class Attention(nn.Module):
    def __init__(self, allow_flash: bool) -> None:
        super().__init__()
        if allow_flash and not FLASH_AVAILABLE:
            warnings.warn(
                "FlashAttention is not available. For optimal speed, "
                "consider installing torch >= 2.0 or flash-attn.",
                stacklevel=2,
            )
        self.enable_flash = allow_flash and FLASH_AVAILABLE
        self.has_sdp = hasattr(F, "scaled_dot_product_attention")
        if allow_flash and FlashCrossAttention:
            self.flash_ = FlashCrossAttention()
        if self.has_sdp:
            torch.backends.cuda.enable_flash_sdp(allow_flash)

    def forward(self, q, k, v, mask: Optional[torch.Tensor] = None) -> torch.Tensor:
        if q.shape[-2] == 0 or k.shape[-2] == 0:
            return q.new_zeros((*q.shape[:-1], v.shape[-1]))
        if self.enable_flash and q.device.type == "cuda":
            # use torch 2.0 scaled_dot_product_attention with flash
            if self.has_sdp:
                args = [x.half().contiguous() for x in [q, k, v]]
                v = F.scaled_dot_product_attention(*args, attn_mask=mask).to(q.dtype)
                return v if mask is None else v.nan_to_num()
            else:
                assert mask is None
                q, k, v = [x.transpose(-2, -3).contiguous() for x in [q, k, v]]
                m = self.flash_(q.half(), torch.stack([k, v], 2).half())
                return m.transpose(-2, -3).to(q.dtype).clone()
        elif self.has_sdp:
            args = [x.contiguous() for x in [q, k, v]]
            v = F.scaled_dot_product_attention(*args, attn_mask=mask)
            return v if mask is None else v.nan_to_num()
        else:
            s = q.shape[-1] ** -0.5
            sim = torch.einsum("...id,...jd->...ij", q, k) * s
            if mask is not None:
                sim.masked_fill(~mask, -float("inf"))
            attn = F.softmax(sim, -1)
            return torch.einsum("...ij,...jd->...id", attn, v)


class SelfBlock(nn.Module):
    def __init__(
        self, embed_dim: int, num_heads: int, flash: bool = False, bias: bool = True
    ) -> None:
        super().__init__()
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        assert self.embed_dim % num_heads == 0
        self.head_dim = self.embed_dim // num_heads
        self.Wqkv = nn.Linear(embed_dim, 3 * embed_dim, bias=bias)
        self.inner_attn = Attention(flash)
        self.out_proj = nn.Linear(embed_dim, embed_dim, bias=bias)
        self.ffn = nn.Sequential(
            nn.Linear(2 * embed_dim, 2 * embed_dim),
            nn.LayerNorm(2 * embed_dim, elementwise_affine=True),
            nn.GELU(),
            nn.Linear(2 * embed_dim, embed_dim),
        )

    def forward(
        self,
        x: torch.Tensor,
        encoding: torch.Tensor,
        mask: Optional[torch.Tensor] = None,
    ) -> torch.Tensor:
        qkv = self.Wqkv(x)
        qkv = qkv.unflatten(-1, (self.num_heads, -1, 3)).transpose(1, 2)
        q, k, v = qkv[..., 0], qkv[..., 1], qkv[..., 2]
        q = apply_cached_rotary_emb(encoding, q)
        k = apply_cached_rotary_emb(encoding, k)
        context = self.inner_attn(q, k, v, mask=mask)
        message = self.out_proj(context.transpose(1, 2).flatten(start_dim=-2))
        return x + self.ffn(torch.cat([x, message], -1))


```

```python
class CrossBlock(nn.Module):
    def __init__(
        self, embed_dim: int, num_heads: int, flash: bool = False, bias: bool = True
    ) -> None:
        super().__init__()
        self.heads = num_heads
        dim_head = embed_dim // num_heads
        self.scale = dim_head**-0.5
        inner_dim = dim_head * num_heads
        self.to_qk = nn.Linear(embed_dim, inner_dim, bias=bias)
        self.to_v = nn.Linear(embed_dim, inner_dim, bias=bias)
        self.to_out = nn.Linear(inner_dim, embed_dim, bias=bias)
        self.ffn = nn.Sequential(
            nn.Linear(2 * embed_dim, 2 * embed_dim),
            nn.LayerNorm(2 * embed_dim, elementwise_affine=True),
            nn.GELU(),
            nn.Linear(2 * embed_dim, embed_dim),
        )
        if flash and FLASH_AVAILABLE:
            self.flash = Attention(True)
        else:
            self.flash = None

    def map_(self, func: Callable, x0: torch.Tensor, x1: torch.Tensor):
        return func(x0), func(x1)

    def forward(
        self, x0: torch.Tensor, x1: torch.Tensor, mask: Optional[torch.Tensor] = None
    ) -> List[torch.Tensor]:
        qk0, qk1 = self.map_(self.to_qk, x0, x1)
        v0, v1 = self.map_(self.to_v, x0, x1)
        qk0, qk1, v0, v1 = map(
            lambda t: t.unflatten(-1, (self.heads, -1)).transpose(1, 2),
            (qk0, qk1, v0, v1),
        )
        if self.flash is not None and qk0.device.type == "cuda":
            m0 = self.flash(qk0, qk1, v1, mask)
            m1 = self.flash(
                qk1, qk0, v0, mask.transpose(-1, -2) if mask is not None else None
            )
        else:
            qk0, qk1 = qk0 * self.scale**0.5, qk1 * self.scale**0.5
            sim = torch.einsum("bhid, bhjd -> bhij", qk0, qk1)
            if mask is not None:
                sim = sim.masked_fill(~mask, -float("inf"))
            attn01 = F.softmax(sim, dim=-1)
            attn10 = F.softmax(sim.transpose(-2, -1).contiguous(), dim=-1)
            m0 = torch.einsum("bhij, bhjd -> bhid", attn01, v1)
            m1 = torch.einsum("bhji, bhjd -> bhid", attn10.transpose(-2, -1), v0)
            if mask is not None:
                m0, m1 = m0.nan_to_num(), m1.nan_to_num()
        m0, m1 = self.map_(lambda t: t.transpose(1, 2).flatten(start_dim=-2), m0, m1)
        m0, m1 = self.map_(self.to_out, m0, m1)
        x0 = x0 + self.ffn(torch.cat([x0, m0], -1))
        x1 = x1 + self.ffn(torch.cat([x1, m1], -1))
        return x0, x1


class TransformerLayer(nn.Module):
    def __init__(self, *args, **kwargs):
        super().__init__()
        self.self_attn = SelfBlock(*args, **kwargs)
        self.cross_attn = CrossBlock(*args, **kwargs)

    def forward(
        self,
        desc0,
        desc1,
        encoding0,
        encoding1,
        mask0: Optional[torch.Tensor] = None,
        mask1: Optional[torch.Tensor] = None,
    ):
        if mask0 is not None and mask1 is not None:
            return self.masked_forward(desc0, desc1, encoding0, encoding1, mask0, mask1)
        else:
            desc0 = self.self_attn(desc0, encoding0)
            desc1 = self.self_attn(desc1, encoding1)
            return self.cross_attn(desc0, desc1)

    # This part is compiled and allows padding inputs
    def masked_forward(self, desc0, desc1, encoding0, encoding1, mask0, mask1):
        mask = mask0 & mask1.transpose(-1, -2)
        mask0 = mask0 & mask0.transpose(-1, -2)
        mask1 = mask1 & mask1.transpose(-1, -2)
```

```
        desc0 = self.self_attn(desc0, encoding0, mask0)
        desc1 = self.self_attn(desc1, encoding1, mask1)
        return self.cross_attn(desc0, desc1, mask)


def sigmoid_log_double_softmax(
    sim: torch.Tensor, z0: torch.Tensor, z1: torch.Tensor
) -> torch.Tensor:
    """create the log assignment matrix from logits and similarity"""
    b, m, n = sim.shape
    certainties = F.logsigmoid(z0) + F.logsigmoid(z1).transpose(1, 2)
    scores0 = F.log_softmax(sim, 2)
    scores1 = F.log_softmax(sim.transpose(-1, -2).contiguous(), 2).transpose(-1, -2)
    scores = sim.new_full((b, m + 1, n + 1), 0)
    scores[:, :m, :n] = scores0 + scores1 + certainties
    scores[:, :-1, -1] = F.logsigmoid(-z0.squeeze(-1))
    scores[:, -1, :-1] = F.logsigmoid(-z1.squeeze(-1))
    return scores


class MatchAssignment(nn.Module):
    def __init__(self, dim: int) -> None:
        super().__init__()
        self.dim = dim
        self.matchability = nn.Linear(dim, 1, bias=True)
        self.final_proj = nn.Linear(dim, dim, bias=True)

    def forward(self, desc0: torch.Tensor, desc1: torch.Tensor):
        """build assignment matrix from descriptors"""
        mdesc0, mdesc1 = self.final_proj(desc0), self.final_proj(desc1)
        _, _, d = mdesc0.shape
        mdesc0, mdesc1 = mdesc0 / d**0.25, mdesc1 / d**0.25
        sim = torch.einsum("bmd,bnd->bmn", mdesc0, mdesc1)
        z0 = self.matchability(desc0)
        z1 = self.matchability(desc1)
        scores = sigmoid_log_double_softmax(sim, z0, z1)
        return scores, sim

    def get_matchability(self, desc: torch.Tensor):
        return torch.sigmoid(self.matchability(desc)).squeeze(-1)


def filter_matches(scores: torch.Tensor, th: float):
    """obtain matches from a log assignment matrix [Bx M+1 x N+1]"""
    max0, max1 = scores[:, :-1, :-1].max(2), scores[:, :-1, :-1].max(1)
    m0, m1 = max0.indices, max1.indices
    indices0 = torch.arange(m0.shape[1], device=m0.device)[None]
    indices1 = torch.arange(m1.shape[1], device=m1.device)[None]
    mutual0 = indices0 == m1.gather(1, m0)
    mutual1 = indices1 == m0.gather(1, m1)
    max0_exp = max0.values.exp()
    zero = max0_exp.new_tensor(0)
    mscores0 = torch.where(mutual0, max0_exp, zero)
    mscores1 = torch.where(mutual1, mscores0.gather(1, m1), zero)
    valid0 = mutual0 & (mscores0 > th)
    valid1 = mutual1 & valid0.gather(1, m1)
    m0 = torch.where(valid0, m0, -1)
    m1 = torch.where(valid1, m1, -1)
    return m0, m1, mscores0, mscores1


class LightGlue(nn.Module):
    default_conf = {
        "name": "lightglue",  # just for interfacing
        "input_dim": 256,  # input descriptor dimension (autoselected from weights)
        "descriptor_dim": 256,
        "add_scale_ori": False,
        "n_layers": 9,
        "num_heads": 4,
        "flash": True,  # enable FlashAttention if available.
        "mp": False,  # enable mixed precision
        "depth_confidence": 0.95,  # early stopping, disable with -1
        "width_confidence": 0.99,  # point pruning, disable with -1
        "filter_threshold": 0.1,  # match threshold
        "weights": None,
    }

    # Point pruning involves an overhead (gather).
    # Therefore, we only activate it if there are enough keypoints.
    pruning_keypoint_thresholds = {
        "cpu": -1,
        "mps": -1,
        "cuda": 1024,
        "flash": 1536,
    }
```

```
338
339     required_data_keys = ["image0", "image1"]
340
341     version = "v0.1_arxiv"
342     url = "https://github.com/cvg/LightGlue/releases/download/{}/{}_lightglue.pth"
343
344     features = {
345         "superpoint": {
346             "weights": "superpoint_lightglue",
347             "input_dim": 256,
348         },
349         "disk": {
350             "weights": "disk_lightglue",
351             "input_dim": 128,
352         },
353         "aliked": {
354             "weights": "aliked_lightglue",
355             "input_dim": 128,
356         },
357         "sift": {
358             "weights": "sift_lightglue",
359             "input_dim": 128,
360             "add_scale_ori": True,
361         },
362         "doghardnet": {
363             "weights": "doghardnet_lightglue",
364             "input_dim": 128,
365             "add_scale_ori": True,
366         },
367     }
368
369     def __init__(self, features="superpoint", **conf) -> None:
370         super().__init__()
371         self.conf = conf = SimpleNamespace(**{**self.default_conf, **conf})
372         if features is not None:
373             if features not in self.features:
374                 raise ValueError(
375                     f"Unsupported features: {features} not in "
376                     f"{{{','.join(self.features)}}}"
377                 )
378             for k, v in self.features[features].items():
379                 setattr(conf, k, v)
380
381         if conf.input_dim != conf.descriptor_dim:
382             self.input_proj = nn.Linear(conf.input_dim, conf.descriptor_dim, bias=True)
383         else:
384             self.input_proj = nn.Identity()
385
386         head_dim = conf.descriptor_dim // conf.num_heads
387         self.posenc = LearnableFourierPositionalEncoding(
388             2 + 2 * self.conf.add_scale_ori, head_dim, head_dim
389         )
390
391         h, n, d = conf.num_heads, conf.n_layers, conf.descriptor_dim
392
393         self.transformers = nn.ModuleList(
394             [TransformerLayer(d, h, conf.flash) for _ in range(n)]
395         )
396
397         self.log_assignment = nn.ModuleList([MatchAssignment(d) for _ in range(n)])
398         self.token_confidence = nn.ModuleList(
399             [TokenConfidence(d) for _ in range(n - 1)]
400         )
401         self.register_buffer(
402             "confidence_thresholds",
403             torch.Tensor(
404                 [self.confidence_threshold(i) for i in range(self.conf.n_layers)]
405             ),
406         )
407
408         state_dict = None
409         if features is not None:
410             fname = f"{conf.weights}_{self.version.replace('.', '-')}.pth"
411             state_dict = torch.hub.load_state_dict_from_url(
412                 self.url.format(self.version, features), file_name=fname
413             )
414             self.load_state_dict(state_dict, strict=False)
415         elif conf.weights is not None:
416             path = Path(__file__).parent
417             path = path / "weights/{}.pth".format(self.conf.weights)
418             state_dict = torch.load(str(path), map_location="cpu")
419
420         if state_dict:
421             # rename old state dict entries
422             for i in range(self.conf.n_layers):
```

```
                pattern = f"self_attn.{i}", f"transformers.{i}.self_attn"
                state_dict = {k.replace(*pattern): v for k, v in state_dict.items()}
                pattern = f"cross_attn.{i}", f"transformers.{i}.cross_attn"
                state_dict = {k.replace(*pattern): v for k, v in state_dict.items()}
            self.load_state_dict(state_dict, strict=False)

        # static lengths LightGlue is compiled for (only used with torch.compile)
        self.static_lengths = None

    def compile(
        self, mode="reduce-overhead", static_lengths=[256, 512, 768, 1024, 1280, 1536]
    ):
        if self.conf.width_confidence != -1:
            warnings.warn(
                "Point pruning is partially disabled for compiled forward.",
                stacklevel=2,
            )

        torch._inductor.cudagraph_mark_step_begin()
        for i in range(self.conf.n_layers):
            self.transformers[i].masked_forward = torch.compile(
                self.transformers[i].masked_forward, mode=mode, fullgraph=True
            )

        self.static_lengths = static_lengths

    def forward(self, data: dict) -> dict:
        """
        Match keypoints and descriptors between two images

        Input (dict):
            image0: dict
                keypoints: [B x M x 2]
                descriptors: [B x M x D]
                image: [B x C x H x W] or image_size: [B x 2]
            image1: dict
                keypoints: [B x N x 2]
                descriptors: [B x N x D]
                image: [B x C x H x W] or image_size: [B x 2]
        Output (dict):
            matches0: [B x M]
            matching_scores0: [B x M]
            matches1: [B x N]
            matching_scores1: [B x N]
            matches: List[[Si x 2]]
            scores: List[[Si]]
            stop: int
            prune0: [B x M]
            prune1: [B x N]
        """
        with torch.autocast(enabled=self.conf.mp, device_type="cuda"):
            return self._forward(data)

    def _forward(self, data: dict) -> dict:
        for key in self.required_data_keys:
            assert key in data, f"Missing key {key} in data"
        data0, data1 = data["image0"], data["image1"]
        kpts0, kpts1 = data0["keypoints"], data1["keypoints"]
        b, m, _ = kpts0.shape
        b, n, _ = kpts1.shape
        device = kpts0.device
        size0, size1 = data0.get("image_size"), data1.get("image_size")
        kpts0 = normalize_keypoints(kpts0, size0).clone()
        kpts1 = normalize_keypoints(kpts1, size1).clone()

        if self.conf.add_scale_ori:
            kpts0 = torch.cat(
                [kpts0] + [data0[k].unsqueeze(-1) for k in ("scales", "oris")], -1
            )
            kpts1 = torch.cat(
                [kpts1] + [data1[k].unsqueeze(-1) for k in ("scales", "oris")], -1
            )
        desc0 = data0["descriptors"].detach().contiguous()
        desc1 = data1["descriptors"].detach().contiguous()

        assert desc0.shape[-1] == self.conf.input_dim
        assert desc1.shape[-1] == self.conf.input_dim

        if torch.is_autocast_enabled():
            desc0 = desc0.half()
            desc1 = desc1.half()

        mask0, mask1 = None, None
        c = max(m, n)
        do_compile = self.static_lengths and c <= max(self.static_lengths)
```

```
        if do_compile:
            kn = min([k for k in self.static_lengths if k >= c])
            desc0, mask0 = pad_to_length(desc0, kn)
            desc1, mask1 = pad_to_length(desc1, kn)
            kpts0, _ = pad_to_length(kpts0, kn)
            kpts1, _ = pad_to_length(kpts1, kn)
        desc0 = self.input_proj(desc0)
        desc1 = self.input_proj(desc1)
        # cache positional embeddings
        encoding0 = self.posenc(kpts0)
        encoding1 = self.posenc(kpts1)

        # GNN + final_proj + assignment
        do_early_stop = self.conf.depth_confidence > 0
        do_point_pruning = self.conf.width_confidence > 0 and not do_compile
        pruning_th = self.pruning_min_kpts(device)
        if do_point_pruning:
            ind0 = torch.arange(0, m, device=device)[None]
            ind1 = torch.arange(0, n, device=device)[None]
            # We store the index of the layer at which pruning is detected.
            prune0 = torch.ones_like(ind0)
            prune1 = torch.ones_like(ind1)
        token0, token1 = None, None
        for i in range(self.conf.n_layers):
            if desc0.shape[1] == 0 or desc1.shape[1] == 0:  # no keypoints
                break
            desc0, desc1 = self.transformers[i](
                desc0, desc1, encoding0, encoding1, mask0=mask0, mask1=mask1
            )
            if i == self.conf.n_layers - 1:
                continue  # no early stopping or adaptive width at last layer

            if do_early_stop:
                token0, token1 = self.token_confidence[i](desc0, desc1)
                if self.check_if_stop(token0[..., :m], token1[..., :n], i, m + n):
                    break
            if do_point_pruning and desc0.shape[-2] > pruning_th:
                scores0 = self.log_assignment[i].get_matchability(desc0)
                prunemask0 = self.get_pruning_mask(token0, scores0, i)
                keep0 = torch.where(prunemask0)[1]
                ind0 = ind0.index_select(1, keep0)
                desc0 = desc0.index_select(1, keep0)
                encoding0 = encoding0.index_select(-2, keep0)
                prune0[:, ind0] += 1
            if do_point_pruning and desc1.shape[-2] > pruning_th:
                scores1 = self.log_assignment[i].get_matchability(desc1)
                prunemask1 = self.get_pruning_mask(token1, scores1, i)
                keep1 = torch.where(prunemask1)[1]
                ind1 = ind1.index_select(1, keep1)
                desc1 = desc1.index_select(1, keep1)
                encoding1 = encoding1.index_select(-2, keep1)
                prune1[:, ind1] += 1

        if desc0.shape[1] == 0 or desc1.shape[1] == 0:  # no keypoints
            m0 = desc0.new_full((b, m), -1, dtype=torch.long)
            m1 = desc1.new_full((b, n), -1, dtype=torch.long)
            mscores0 = desc0.new_zeros((b, m))
            mscores1 = desc1.new_zeros((b, n))
            matches = desc0.new_empty((b, 0, 2), dtype=torch.long)
            mscores = desc0.new_empty((b, 0))
            if not do_point_pruning:
                prune0 = torch.ones_like(mscores0) * self.conf.n_layers
                prune1 = torch.ones_like(mscores1) * self.conf.n_layers
            return {
                "matches0": m0,
                "matches1": m1,
                "matching_scores0": mscores0,
                "matching_scores1": mscores1,
                "stop": i + 1,
                "matches": matches,
                "scores": mscores,
                "prune0": prune0,
                "prune1": prune1,
            }

        desc0, desc1 = desc0[..., :m, :], desc1[..., :n, :]  # remove padding
        scores, _ = self.log_assignment[i](desc0, desc1)
        m0, m1, mscores0, mscores1 = filter_matches(scores, self.conf.filter_threshold)
        matches, mscores = [], []
        for k in range(b):
            valid = m0[k] > -1
            m_indices_0 = torch.where(valid)[0]
            m_indices_1 = m0[k][valid]
            if do_point_pruning:
                m_indices_0 = ind0[k, m_indices_0]
```

```
                m_indices_1 = ind1[k, m_indices_1]
            matches.append(torch.stack([m_indices_0, m_indices_1], -1))
            mscores.append(mscores0[k][valid])

        # TODO: Remove when hloc switches to the compact format.
        if do_point_pruning:
            m0_ = torch.full((b, m), -1, device=m0.device, dtype=m0.dtype)
            m1_ = torch.full((b, n), -1, device=m1.device, dtype=m1.dtype)
            m0_[:, ind0] = torch.where(m0 == -1, -1, ind1.gather(1, m0.clamp(min=0)))
            m1_[:, ind1] = torch.where(m1 == -1, -1, ind0.gather(1, m1.clamp(min=0)))
            mscores0_ = torch.zeros((b, m), device=mscores0.device)
            mscores1_ = torch.zeros((b, n), device=mscores1.device)
            mscores0_[:, ind0] = mscores0
            mscores1_[:, ind1] = mscores1
            m0, m1, mscores0, mscores1 = m0_, m1_, mscores0_, mscores1_
        else:
            prune0 = torch.ones_like(mscores0) * self.conf.n_layers
            prune1 = torch.ones_like(mscores1) * self.conf.n_layers

        return {
            "matches0": m0,
            "matches1": m1,
            "matching_scores0": mscores0,
            "matching_scores1": mscores1,
            "stop": i + 1,
            "matches": matches,
            "scores": mscores,
            "prune0": prune0,
            "prune1": prune1,
        }

    def confidence_threshold(self, layer_index: int) -> float:
        """scaled confidence threshold"""
        threshold = 0.8 + 0.1 * np.exp(-4.0 * layer_index / self.conf.n_layers)
        return np.clip(threshold, 0, 1)

    def get_pruning_mask(
        self, confidences: torch.Tensor, scores: torch.Tensor, layer_index: int
    ) -> torch.Tensor:
        """mask points which should be removed"""
        keep = scores > (1 - self.conf.width_confidence)
        if confidences is not None:  # Low-confidence points are never pruned.
            keep |= confidences <= self.confidence_thresholds[layer_index]
        return keep

    def check_if_stop(
        self,
        confidences0: torch.Tensor,
        confidences1: torch.Tensor,
        layer_index: int,
        num_points: int,
    ) -> torch.Tensor:
        """evaluate stopping condition"""
        confidences = torch.cat([confidences0, confidences1], -1)
        threshold = self.confidence_thresholds[layer_index]
        ratio_confident = 1.0 - (confidences < threshold).float().sum() / num_points
        return ratio_confident > self.conf.depth_confidence

    def pruning_min_kpts(self, device: torch.device):
        if self.conf.flash and FLASH_AVAILABLE and device.type == "cuda":
            return self.pruning_keypoint_thresholds["flash"]
        else:
            return self.pruning_keypoint_thresholds[device.type]
```

```
LightGlue/lightglue/sift.py
import warnings

import cv2
import numpy as np
import torch
from kornia.color import rgb_to_grayscale
from packaging import version

try:
    import pycolmap
except ImportError:
    pycolmap = None

from .utils import Extractor


def filter_dog_point(points, scales, angles, image_shape, nms_radius, scores=None):
    h, w = image_shape
    ij = np.round(points - 0.5).astype(int).T[::-1]

    # Remove duplicate points (identical coordinates).
    # Pick highest scale or score
    s = scales if scores is None else scores
    buffer = np.zeros((h, w))
    np.maximum.at(buffer, tuple(ij), s)
    keep = np.where(buffer[tuple(ij)] == s)[0]

    # Pick lowest angle (arbitrary).
    ij = ij[:, keep]
    buffer[:] = np.inf
    o_abs = np.abs(angles[keep])
    np.minimum.at(buffer, tuple(ij), o_abs)
    mask = buffer[tuple(ij)] == o_abs
    ij = ij[:, mask]
    keep = keep[mask]

    if nms_radius > 0:
        # Apply NMS on the remaining points
        buffer[:] = 0
        buffer[tuple(ij)] = s[keep]  # scores or scale

        local_max = torch.nn.functional.max_pool2d(
            torch.from_numpy(buffer).unsqueeze(0),
            kernel_size=nms_radius * 2 + 1,
            stride=1,
            padding=nms_radius,
        ).squeeze(0)
        is_local_max = buffer == local_max.numpy()
        keep = keep[is_local_max[tuple(ij)]]
    return keep


def sift_to_rootsift(x: torch.Tensor, eps=1e-6) -> torch.Tensor:
    x = torch.nn.functional.normalize(x, p=1, dim=-1, eps=eps)
    x.clip_(min=eps).sqrt_()
    return torch.nn.functional.normalize(x, p=2, dim=-1, eps=eps)


def run_opencv_sift(features: cv2.Feature2D, image: np.ndarray) -> np.ndarray:
    """
    Detect keypoints using OpenCV Detector.
    Optionally, perform description.
    Args:
        features: OpenCV based keypoints detector and descriptor
        image: Grayscale image of uint8 data type
    Returns:
        keypoints: 1D array of detected cv2.KeyPoint
        scores: 1D array of responses
        descriptors: 1D array of descriptors
    """
    detections, descriptors = features.detectAndCompute(image, None)
    points = np.array([k.pt for k in detections], dtype=np.float32)
    scores = np.array([k.response for k in detections], dtype=np.float32)
    scales = np.array([k.size for k in detections], dtype=np.float32)
    angles = np.deg2rad(np.array([k.angle for k in detections], dtype=np.float32))
    return points, scores, scales, angles, descriptors


class SIFT(Extractor):
    default_conf = {
        "rootsift": True,
        "nms_radius": 0,  # None to disable filtering entirely.
```

```
        "max_num_keypoints": 4096,
        "backend": "opencv",  # in {opencv, pycolmap, pycolmap_cpu, pycolmap_cuda}
        "detection_threshold": 0.0066667,  # from COLMAP
        "edge_threshold": 10,
        "first_octave": -1,  # only used by pycolmap, the default of COLMAP
        "num_octaves": 4,
    }

    preprocess_conf = {
        "resize": 1024,
    }

    required_data_keys = ["image"]

    def __init__(self, **conf):
        super().__init__(**conf)  # Update with default configuration.
        backend = self.conf.backend
        if backend.startswith("pycolmap"):
            if pycolmap is None:
                raise ImportError(
                    "Cannot find module pycolmap: install it with pip"
                    "or use backend=opencv."
                )
            options = {
                "peak_threshold": self.conf.detection_threshold,
                "edge_threshold": self.conf.edge_threshold,
                "first_octave": self.conf.first_octave,
                "num_octaves": self.conf.num_octaves,
                "normalization": pycolmap.Normalization.L2,  # L1_ROOT is buggy.
            }
            device = (
                "auto" if backend == "pycolmap" else backend.replace("pycolmap_", "")
            )
            if (
                backend == "pycolmap_cpu" or not pycolmap.has_cuda
            ) and pycolmap.__version__ < "0.5.0":
                warnings.warn(
                    "The pycolmap CPU SIFT is buggy in version < 0.5.0, "
                    "consider upgrading pycolmap or use the CUDA version.",
                    stacklevel=1,
                )
            else:
                options["max_num_features"] = self.conf.max_num_keypoints
            self.sift = pycolmap.Sift(options=options, device=device)
        elif backend == "opencv":
            self.sift = cv2.SIFT_create(
                contrastThreshold=self.conf.detection_threshold,
                nfeatures=self.conf.max_num_keypoints,
                edgeThreshold=self.conf.edge_threshold,
                nOctaveLayers=self.conf.num_octaves,
            )
        else:
            backends = {"opencv", "pycolmap", "pycolmap_cpu", "pycolmap_cuda"}
            raise ValueError(
                f"Unknown backend: {backend} not in " f"{{{','.join(backends)}}}."
            )

    def extract_single_image(self, image: torch.Tensor):
        image_np = image.cpu().numpy().squeeze(0)

        if self.conf.backend.startswith("pycolmap"):
            if version.parse(pycolmap.__version__) >= version.parse("0.5.0"):
                detections, descriptors = self.sift.extract(image_np)
                scores = None  # Scores are not exposed by COLMAP anymore.
            else:
                detections, scores, descriptors = self.sift.extract(image_np)
            keypoints = detections[:, :2]  # Keep only (x, y).
            scales, angles = detections[:, -2:].T
            if scores is not None and (
                self.conf.backend == "pycolmap_cpu" or not pycolmap.has_cuda
            ):
                # Set the scores as a combination of abs. response and scale.
                scores = np.abs(scores) * scales
        elif self.conf.backend == "opencv":
            # TODO: Check if opencv keypoints are already in corner convention
            keypoints, scores, scales, angles, descriptors = run_opencv_sift(
                self.sift, (image_np * 255.0).astype(np.uint8)
            )
        pred = {
            "keypoints": keypoints,
            "scales": scales,
            "oris": angles,
            "descriptors": descriptors,
        }
        if scores is not None:
```

```
            pred["keypoint_scores"] = scores

        # sometimes pycolmap returns points outside the image. We remove them
        if self.conf.backend.startswith("pycolmap"):
            is_inside = (
                pred["keypoints"] + 0.5 < np.array([image_np.shape[-2:][::-1]])
            ).all(-1)
            pred = {k: v[is_inside] for k, v in pred.items()}

        if self.conf.nms_radius is not None:
            keep = filter_dog_point(
                pred["keypoints"],
                pred["scales"],
                pred["oris"],
                image_np.shape,
                self.conf.nms_radius,
                scores=pred.get("keypoint_scores"),
            )
            pred = {k: v[keep] for k, v in pred.items()}

        pred = {k: torch.from_numpy(v) for k, v in pred.items()}
        if scores is not None:
            # Keep the k keypoints with highest score
            num_points = self.conf.max_num_keypoints
            if num_points is not None and len(pred["keypoints"]) > num_points:
                indices = torch.topk(pred["keypoint_scores"], num_points).indices
                pred = {k: v[indices] for k, v in pred.items()}

        return pred

    def forward(self, data: dict) -> dict:
        image = data["image"]
        if image.shape[1] == 3:
            image = rgb_to_grayscale(image)
        device = image.device
        image = image.cpu()
        pred = []
        for k in range(len(image)):
            img = image[k]
            if "image_size" in data.keys():
                # avoid extracting points in padded areas
                w, h = data["image_size"][k]
                img = img[:, :h, :w]
            p = self.extract_single_image(img)
            pred.append(p)
        pred = {k: torch.stack([p[k] for p in pred], 0).to(device) for k in pred[0]}
        if self.conf.rootsift:
            pred["descriptors"] = sift_to_rootsift(pred["descriptors"])
        return pred
```

```
LightGlue/lightglue/superpoint.py
# %BANNER_BEGIN%
# ---------------------------------------------------------------------
# %COPYRIGHT_BEGIN%
#
#  Magic Leap, Inc. ("COMPANY") CONFIDENTIAL
#
#  Unpublished Copyright (c) 2020
#  Magic Leap, Inc., All Rights Reserved.
#
# NOTICE:  All information contained herein is, and remains the property
# of COMPANY. The intellectual and technical concepts contained herein
# are proprietary to COMPANY and may be covered by U.S. and Foreign
# Patents, patents in process, and are protected by trade secret or
# copyright law.  Dissemination of this information or reproduction of
# this material is strictly forbidden unless prior written permission is
# obtained from COMPANY.  Access to the source code contained herein is
# hereby forbidden to anyone except current COMPANY employees, managers
# or contractors who have executed Confidentiality and Non-disclosure
# agreements explicitly covering such access.
#
# The copyright notice above does not evidence any actual or intended
# publication or disclosure  of  this source code, which includes
# information that is confidential and/or proprietary, and is a trade
# secret, of  COMPANY.   ANY REPRODUCTION, MODIFICATION, DISTRIBUTION,
# PUBLIC  PERFORMANCE, OR PUBLIC DISPLAY OF OR THROUGH USE  OF THIS
# SOURCE CODE  WITHOUT THE EXPRESS WRITTEN CONSENT OF COMPANY IS
# STRICTLY PROHIBITED, AND IN VIOLATION OF APPLICABLE LAWS AND
# INTERNATIONAL TREATIES.  THE RECEIPT OR POSSESSION OF  THIS SOURCE
# CODE AND/OR RELATED INFORMATION DOES NOT CONVEY OR IMPLY ANY RIGHTS
# TO REPRODUCE, DISCLOSE OR DISTRIBUTE ITS CONTENTS, OR TO MANUFACTURE,
# USE, OR SELL ANYTHING THAT IT  MAY DESCRIBE, IN WHOLE OR IN PART.
#
# %COPYRIGHT_END%
# ----------------------------------------------------------------------
# %AUTHORS_BEGIN%
#
#  Originating Authors: Paul-Edouard Sarlin
#
# %AUTHORS_END%
# --------------------------------------------------------------------*/
# %BANNER_END%

# Adapted by Remi Pautrat, Philipp Lindenberger

import torch
from kornia.color import rgb_to_grayscale
from torch import nn

from .utils import Extractor


def simple_nms(scores, nms_radius: int):
    """Fast Non-maximum suppression to remove nearby points"""
    assert nms_radius >= 0

    def max_pool(x):
        return torch.nn.functional.max_pool2d(
            x, kernel_size=nms_radius * 2 + 1, stride=1, padding=nms_radius
        )

    zeros = torch.zeros_like(scores)
    max_mask = scores == max_pool(scores)
    for _ in range(2):
        supp_mask = max_pool(max_mask.float()) > 0
        supp_scores = torch.where(supp_mask, zeros, scores)
        new_max_mask = supp_scores == max_pool(supp_scores)
        max_mask = max_mask | (new_max_mask & (~supp_mask))
    return torch.where(max_mask, scores, zeros)


def top_k_keypoints(keypoints, scores, k):
    if k >= len(keypoints):
        return keypoints, scores
    scores, indices = torch.topk(scores, k, dim=0, sorted=True)
    return keypoints[indices], scores


def sample_descriptors(keypoints, descriptors, s: int = 8):
    """Interpolate descriptors at keypoint locations"""
    b, c, h, w = descriptors.shape
    keypoints = keypoints - s / 2 + 0.5
    keypoints /= torch.tensor(
```

```
        [(w * s - s / 2 - 0.5), (h * s - s / 2 - 0.5)],
    ).to(
        keypoints
    )[None]
    keypoints = keypoints * 2 - 1  # normalize to (-1, 1)
    args = {"align_corners": True} if torch.__version__ >= "1.3" else {}
    descriptors = torch.nn.functional.grid_sample(
        descriptors, keypoints.view(b, 1, -1, 2), mode="bilinear", **args
    )
    descriptors = torch.nn.functional.normalize(
        descriptors.reshape(b, c, -1), p=2, dim=1
    )
    return descriptors


class SuperPoint(Extractor):
    """SuperPoint Convolutional Detector and Descriptor

    SuperPoint: Self-Supervised Interest Point Detection and
    Description. Daniel DeTone, Tomasz Malisiewicz, and Andrew
    Rabinovich. In CVPRW, 2019. https://arxiv.org/abs/1712.07629

    """

    default_conf = {
        "descriptor_dim": 256,
        "nms_radius": 4,
        "max_num_keypoints": None,
        "detection_threshold": 0.0005,
        "remove_borders": 4,
    }

    preprocess_conf = {
        "resize": 1024,
    }

    required_data_keys = ["image"]

    def __init__(self, **conf):
        super().__init__(**conf)  # Update with default configuration.
        self.relu = nn.ReLU(inplace=True)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        c1, c2, c3, c4, c5 = 64, 64, 128, 128, 256

        self.conv1a = nn.Conv2d(1, c1, kernel_size=3, stride=1, padding=1)
        self.conv1b = nn.Conv2d(c1, c1, kernel_size=3, stride=1, padding=1)
        self.conv2a = nn.Conv2d(c1, c2, kernel_size=3, stride=1, padding=1)
        self.conv2b = nn.Conv2d(c2, c2, kernel_size=3, stride=1, padding=1)
        self.conv3a = nn.Conv2d(c2, c3, kernel_size=3, stride=1, padding=1)
        self.conv3b = nn.Conv2d(c3, c3, kernel_size=3, stride=1, padding=1)
        self.conv4a = nn.Conv2d(c3, c4, kernel_size=3, stride=1, padding=1)
        self.conv4b = nn.Conv2d(c4, c4, kernel_size=3, stride=1, padding=1)

        self.convPa = nn.Conv2d(c4, c5, kernel_size=3, stride=1, padding=1)
        self.convPb = nn.Conv2d(c5, 65, kernel_size=1, stride=1, padding=0)

        self.convDa = nn.Conv2d(c4, c5, kernel_size=3, stride=1, padding=1)
        self.convDb = nn.Conv2d(
            c5, self.conf.descriptor_dim, kernel_size=1, stride=1, padding=0
        )

        url = "https://github.com/cvg/LightGlue/releases/download/v0.1_arxiv/superpoint_v1.pth"  # noqa
        self.load_state_dict(torch.hub.load_state_dict_from_url(url))

        if self.conf.max_num_keypoints is not None and self.conf.max_num_keypoints <= 0:
            raise ValueError("max_num_keypoints must be positive or None")

    def forward(self, data: dict) -> dict:
        """Compute keypoints, scores, descriptors for image"""
        for key in self.required_data_keys:
            assert key in data, f"Missing key {key} in data"
        image = data["image"]
        if image.shape[1] == 3:
            image = rgb_to_grayscale(image)

        # Shared Encoder
        x = self.relu(self.conv1a(image))
        x = self.relu(self.conv1b(x))
        x = self.pool(x)
        x = self.relu(self.conv2a(x))
        x = self.relu(self.conv2b(x))
        x = self.pool(x)
        x = self.relu(self.conv3a(x))
        x = self.relu(self.conv3b(x))
        x = self.pool(x)
```

```
        x = self.relu(self.conv4a(x))
        x = self.relu(self.conv4b(x))

        # Compute the dense keypoint scores
        cPa = self.relu(self.convPa(x))
        scores = self.convPb(cPa)
        scores = torch.nn.functional.softmax(scores, 1)[:, :-1]
        b, _, h, w = scores.shape
        scores = scores.permute(0, 2, 3, 1).reshape(b, h, w, 8, 8)
        scores = scores.permute(0, 1, 3, 2, 4).reshape(b, h * 8, w * 8)
        scores = simple_nms(scores, self.conf.nms_radius)

        # Discard keypoints near the image borders
        if self.conf.remove_borders:
            pad = self.conf.remove_borders
            scores[:, :pad] = -1
            scores[:, :, :pad] = -1
            scores[:, -pad:] = -1
            scores[:, :, -pad:] = -1

        # Extract keypoints
        best_kp = torch.where(scores > self.conf.detection_threshold)
        scores = scores[best_kp]

        # Separate into batches
        keypoints = [
            torch.stack(best_kp[1:3], dim=-1)[best_kp[0] == i] for i in range(b)
        ]
        scores = [scores[best_kp[0] == i] for i in range(b)]

        # Keep the k keypoints with highest score
        if self.conf.max_num_keypoints is not None:
            keypoints, scores = list(
                zip(
                    *[
                        top_k_keypoints(k, s, self.conf.max_num_keypoints)
                        for k, s in zip(keypoints, scores)
                    ]
                )
            )

        # Convert (h, w) to (x, y)
        keypoints = [torch.flip(k, [1]).float() for k in keypoints]

        # Compute the dense descriptors
        cDa = self.relu(self.convDa(x))
        descriptors = self.convDb(cDa)
        descriptors = torch.nn.functional.normalize(descriptors, p=2, dim=1)

        # Extract descriptors
        descriptors = [
            sample_descriptors(k[None], d[None], 8)[0]
            for k, d in zip(keypoints, descriptors)
        ]

        return {
            "keypoints": torch.stack(keypoints, 0),
            "keypoint_scores": torch.stack(scores, 0),
            "descriptors": torch.stack(descriptors, 0).transpose(-1, -2).contiguous(),
        }
```

```python
LightGlue/lightglue/utils.py
import collections.abc as collections
from pathlib import Path
from types import SimpleNamespace
from typing import Callable, List, Optional, Tuple, Union

import cv2
import kornia
import numpy as np
import torch


class ImagePreprocessor:
    default_conf = {
        "resize": None,  # target edge length, None for no resizing
        "side": "long",
        "interpolation": "bilinear",
        "align_corners": None,
        "antialias": True,
    }

    def __init__(self, **conf) -> None:
        super().__init__()
        self.conf = {**self.default_conf, **conf}
        self.conf = SimpleNamespace(**self.conf)

    def __call__(self, img: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
        """Resize and preprocess an image, return image and resize scale"""
        h, w = img.shape[-2:]
        if self.conf.resize is not None:
            img = kornia.geometry.transform.resize(
                img,
                self.conf.resize,
                side=self.conf.side,
                antialias=self.conf.antialias,
                align_corners=self.conf.align_corners,
            )
        scale = torch.Tensor([img.shape[-1] / w, img.shape[-2] / h]).to(img)
        return img, scale


def map_tensor(input_, func: Callable):
    string_classes = (str, bytes)
    if isinstance(input_, string_classes):
        return input_
    elif isinstance(input_, collections.Mapping):
        return {k: map_tensor(sample, func) for k, sample in input_.items()}
    elif isinstance(input_, collections.Sequence):
        return [map_tensor(sample, func) for sample in input_]
    elif isinstance(input_, torch.Tensor):
        return func(input_)
    else:
        return input_


def batch_to_device(batch: dict, device: str = "cpu", non_blocking: bool = True):
    """Move batch (dict) to device"""

    def _func(tensor):
        return tensor.to(device=device, non_blocking=non_blocking).detach()

    return map_tensor(batch, _func)


def rbd(data: dict) -> dict:
    """Remove batch dimension from elements in data"""
    return {
        k: v[0] if isinstance(v, (torch.Tensor, np.ndarray, list)) else v
        for k, v in data.items()
    }


def read_image(path: Path, grayscale: bool = False) -> np.ndarray:
    """Read an image from path as RGB or grayscale"""
    if not Path(path).exists():
        raise FileNotFoundError(f"No image at path {path}.")
    mode = cv2.IMREAD_GRAYSCALE if grayscale else cv2.IMREAD_COLOR
    image = cv2.imread(str(path), mode)
    if image is None:
        raise IOError(f"Could not read image at {path}.")
    if not grayscale:
        image = image[..., ::-1]
    return image
```

```python


def numpy_image_to_torch(image: np.ndarray) -> torch.Tensor:
    """Normalize the image tensor and reorder the dimensions."""
    if image.ndim == 3:
        image = image.transpose((2, 0, 1))  # HxWxC to CxHxW
    elif image.ndim == 2:
        image = image[None]  # add channel axis
    else:
        raise ValueError(f"Not an image: {image.shape}")
    return torch.tensor(image / 255.0, dtype=torch.float)


def resize_image(
    image: np.ndarray,
    size: Union[List[int], int],
    fn: str = "max",
    interp: Optional[str] = "area",
) -> np.ndarray:
    """Resize an image to a fixed size, or according to max or min edge."""
    h, w = image.shape[:2]

    fn = {"max": max, "min": min}[fn]
    if isinstance(size, int):
        scale = size / fn(h, w)
        h_new, w_new = int(round(h * scale)), int(round(w * scale))
        scale = (w_new / w, h_new / h)
    elif isinstance(size, (tuple, list)):
        h_new, w_new = size
        scale = (w_new / w, h_new / h)
    else:
        raise ValueError(f"Incorrect new size: {size}")
    mode = {
        "linear": cv2.INTER_LINEAR,
        "cubic": cv2.INTER_CUBIC,
        "nearest": cv2.INTER_NEAREST,
        "area": cv2.INTER_AREA,
    }[interp]
    return cv2.resize(image, (w_new, h_new), interpolation=mode), scale


def load_image(path: Path, resize: int = None, **kwargs) -> torch.Tensor:
    image = read_image(path)
    if resize is not None:
        image, _ = resize_image(image, resize, **kwargs)
    return numpy_image_to_torch(image)


class Extractor(torch.nn.Module):
    def __init__(self, **conf):
        super().__init__()
        self.conf = SimpleNamespace(**{**self.default_conf, **conf})

    @torch.no_grad()
    def extract(self, img: torch.Tensor, **conf) -> dict:
        """Perform extraction with online resizing"""
        if img.dim() == 3:
            img = img[None]  # add batch dim
        assert img.dim() == 4 and img.shape[0] == 1
        shape = img.shape[-2:][::-1]
        img, scales = ImagePreprocessor(**{**self.preprocess_conf, **conf})(img)
        feats = self.forward({"image": img})
        feats["image_size"] = torch.tensor(shape)[None].to(img).float()
        feats["keypoints"] = (feats["keypoints"] + 0.5) / scales[None] - 0.5
        return feats


def match_pair(
    extractor,
    matcher,
    image0: torch.Tensor,
    image1: torch.Tensor,
    device: str = "cpu",
    **preprocess,
):
    """Match a pair of images (image0, image1) with an extractor and matcher"""
    feats0 = extractor.extract(image0, **preprocess)
    feats1 = extractor.extract(image1, **preprocess)
    matches01 = matcher({"image0": feats0, "image1": feats1})
    data = [feats0, feats1, matches01]
    # remove batch dim and move to target device
    feats0, feats1, matches01 = [batch_to_device(rbd(x), device) for x in data]
    return feats0, feats1, matches01
```

```python
LightGlue/lightglue/viz2d.py
"""
2D visualization primitives based on Matplotlib.
1) Plot images with `plot_images`.
2) Call `plot_keypoints` or `plot_matches` any number of times.
3) Optionally: save a .png or .pdf plot (nice in papers!) with `save_plot`.
"""

import matplotlib
import matplotlib.patheffects as path_effects
import matplotlib.pyplot as plt
import numpy as np
import torch


def cm_RdGn(x):
    """Custom colormap: red (0) -> yellow (0.5) -> green (1)."""
    x = np.clip(x, 0, 1)[..., None] * 2
    c = x * np.array([[0, 1.0, 0]]) + (2 - x) * np.array([[1.0, 0, 0]])
    return np.clip(c, 0, 1)


def cm_BlRdGn(x_):
    """Custom colormap: blue (-1) -> red (0.0) -> green (1)."""
    x = np.clip(x_, 0, 1)[..., None] * 2
    c = x * np.array([[0, 1.0, 0, 1.0]]) + (2 - x) * np.array([[1.0, 0, 0, 1.0]])

    xn = -np.clip(x_, -1, 0)[..., None] * 2
    cn = xn * np.array([[0, 0.1, 1, 1.0]]) + (2 - xn) * np.array([[1.0, 0, 0, 1.0]])
    out = np.clip(np.where(x_[..., None] < 0, cn, c), 0, 1)
    return out


def cm_prune(x_):
    """Custom colormap to visualize pruning"""
    if isinstance(x_, torch.Tensor):
        x_ = x_.cpu().numpy()
    max_i = max(x_)
    norm_x = np.where(x_ == max_i, -1, (x_ - 1) / 9)
    return cm_BlRdGn(norm_x)


def plot_images(imgs, titles=None, cmaps="gray", dpi=100, pad=0.5, adaptive=True):
    """Plot a set of images horizontally.
    Args:
        imgs: list of NumPy RGB (H, W, 3) or PyTorch RGB (3, H, W) or mono (H, W).
        titles: a list of strings, as titles for each image.
        cmaps: colormaps for monochrome images.
        adaptive: whether the figure size should fit the image aspect ratios.
    """
    # conversion to (H, W, 3) for torch.Tensor
    imgs = [
        img.permute(1, 2, 0).cpu().numpy()
        if (isinstance(img, torch.Tensor) and img.dim() == 3)
        else img
        for img in imgs
    ]

    n = len(imgs)
    if not isinstance(cmaps, (list, tuple)):
        cmaps = [cmaps] * n

    if adaptive:
        ratios = [i.shape[1] / i.shape[0] for i in imgs]  # W / H
    else:
        ratios = [4 / 3] * n
    figsize = [sum(ratios) * 4.5, 4.5]
    fig, ax = plt.subplots(
        1, n, figsize=figsize, dpi=dpi, gridspec_kw={"width_ratios": ratios}
    )
    if n == 1:
        ax = [ax]
    for i in range(n):
        ax[i].imshow(imgs[i], cmap=plt.get_cmap(cmaps[i]))
        ax[i].get_yaxis().set_ticks([])
        ax[i].get_xaxis().set_ticks([])
        ax[i].set_axis_off()
        for spine in ax[i].spines.values():  # remove frame
            spine.set_visible(False)
        if titles:
            ax[i].set_title(titles[i])
    fig.tight_layout(pad=pad)

```

```python

def plot_keypoints(kpts, colors="lime", ps=4, axes=None, a=1.0):
    """Plot keypoints for existing images.
    Args:
        kpts: list of ndarrays of size (N, 2).
        colors: string, or list of list of tuples (one for each keypoints).
        ps: size of the keypoints as float.
    """
    if not isinstance(colors, list):
        colors = [colors] * len(kpts)
    if not isinstance(a, list):
        a = [a] * len(kpts)
    if axes is None:
        axes = plt.gcf().axes
    for ax, k, c, alpha in zip(axes, kpts, colors, a):
        if isinstance(k, torch.Tensor):
            k = k.cpu().numpy()
        ax.scatter(k[:, 0], k[:, 1], c=c, s=ps, linewidths=0, alpha=alpha)


def plot_matches(kpts0, kpts1, color=None, lw=1.5, ps=4, a=1.0, labels=None, axes=None):
    """Plot matches for a pair of existing images.
    Args:
        kpts0, kpts1: corresponding keypoints of size (N, 2).
        color: color of each match, string or RGB tuple. Random if not given.
        lw: width of the lines.
        ps: size of the end points (no endpoint if ps=0)
        indices: indices of the images to draw the matches on.
        a: alpha opacity of the match lines.
    """
    fig = plt.gcf()
    if axes is None:
        ax = fig.axes
        ax0, ax1 = ax[0], ax[1]
    else:
        ax0, ax1 = axes
    if isinstance(kpts0, torch.Tensor):
        kpts0 = kpts0.cpu().numpy()
    if isinstance(kpts1, torch.Tensor):
        kpts1 = kpts1.cpu().numpy()
    assert len(kpts0) == len(kpts1)
    if color is None:
        color = matplotlib.cm.hsv(np.random.rand(len(kpts0))).tolist()
    elif len(color) > 0 and not isinstance(color[0], (tuple, list)):
        color = [color] * len(kpts0)

    if lw > 0:
        for i in range(len(kpts0)):
            line = matplotlib.patches.ConnectionPatch(
                xyA=(kpts0[i, 0], kpts0[i, 1]),
                xyB=(kpts1[i, 0], kpts1[i, 1]),
                coordsA=ax0.transData,
                coordsB=ax1.transData,
                axesA=ax0,
                axesB=ax1,
                zorder=1,
                color=color[i],
                linewidth=lw,
                clip_on=True,
                alpha=a,
                label=None if labels is None else labels[i],
                picker=5.0,
            )
            line.set_annotation_clip(True)
            fig.add_artist(line)

    # freeze the axes to prevent the transform to change
    ax0.autoscale(enable=False)
    ax1.autoscale(enable=False)

    if ps > 0:
        ax0.scatter(kpts0[:, 0], kpts0[:, 1], c=color, s=ps)
        ax1.scatter(kpts1[:, 0], kpts1[:, 1], c=color, s=ps)


def add_text(
    idx,
    text,
    pos=(0.01, 0.99),
    fs=15,
    color="w",
    lcolor="k",
    lwidth=2,
    ha="left",
    va="top",
```

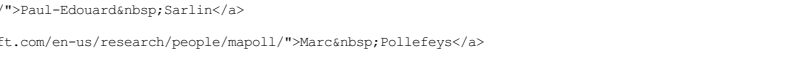
```
):
    ax = plt.gcf().axes[idx]
    t = ax.text(
        *pos, text, fontsize=fs, ha=ha, va=va, color=color, transform=ax.transAxes
    )
    if lcolor is not None:
        t.set_path_effects(
            [
                path_effects.Stroke(linewidth=lwidth, foreground=lcolor),
                path_effects.Normal(),
            ]
        )


def save_plot(path, **kw):
    """Save the current figure without any white margin."""
    plt.savefig(path, bbox_inches="tight", pad_inches=0, **kw)
```

LightGlue/README.md

  <h1 align="center"><ins>LightGlue</ins> ⚡️
Local Feature Matching at Light Speed</h1>

    Philipp Lindenberger
    ·
    Paul-Edouard Sarlin
    ·
    Marc Pollefeys

  <h2 align="center">
    ICCV 2023
    Paper |
    Colab |
    Poster |
    Train your own!
  </h2>

    LightGlue is a deep neural network that matches sparse local features across image pairs.
An adaptive mechanism makes it fast for easy pairs (top) and reduces t

##

This repository hosts the inference code of LightGlue, a lightweight feature matcher with high accuracy and blazing fast inference. It takes as input a set of keypoints ar

We release pretrained weights of LightGlue with [SuperPoint](https://arxiv.org/abs/1712.07629), [DISK](https://arxiv.org/abs/2006.13566), [ALIKED](https://arxiv.org/abs/23
The training and evaluation code can be found in our library [glue-factory](https://github.com/cvg/glue-factory/).

Installation and demo [![Open In Colab](https://colab.research.google.com/assets/colab-badge.svg)](https://colab.research.google.com/github/cvg/LightGlue/blob/main/demo

Install this repo using pip:

```bash
git clone https://github.com/cvg/LightGlue.git && cd LightGlue
python -m pip install -e .
```

We provide a [demo notebook](demo.ipynb) which shows how to perform feature extraction and matching on an image pair.

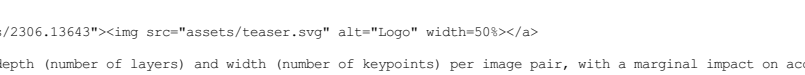
Here is a minimal script to match two images:

```python
from lightglue import LightGlue, SuperPoint, DISK, SIFT, ALIKED, DoGHardNet
from lightglue.utils import load_image, rbd

# SuperPoint+LightGlue
extractor = SuperPoint(max_num_keypoints=2048).eval().cuda()  # load the extractor
matcher = LightGlue(features='superpoint').eval().cuda()  # load the matcher

# or DISK+LightGlue, ALIKED+LightGlue or SIFT+LightGlue
extractor = DISK(max_num_keypoints=2048).eval().cuda()  # load the extractor
matcher = LightGlue(features='disk').eval().cuda()  # load the matcher

# load each image as a torch.Tensor on GPU with shape (3,H,W), normalized in [0,1]
image0 = load_image('path/to/image_0.jpg').cuda()
image1 = load_image('path/to/image_1.jpg').cuda()

# extract local features
feats0 = extractor.extract(image0)  # auto-resize the image, disable with resize=None
feats1 = extractor.extract(image1)

# match the features
matches01 = matcher({'image0': feats0, 'image1': feats1})
feats0, feats1, matches01 = [rbd(x) for x in [feats0, feats1, matches01]]  # remove batch dimension
matches = matches01['matches']  # indices with shape (K,2)
points0 = feats0['keypoints'][matches[..., 0]]  # coordinates in image #0, shape (K,2)
points1 = feats1['keypoints'][matches[..., 1]]  # coordinates in image #1, shape (K,2)
```

We also provide a convenience method to match a pair of images:

```python
from lightglue import match_pair
feats0, feats1, matches01 = match_pair(extractor, matcher, image0, image1)
```

##

  LightGlue can adjust its depth (number of layers) and width (number of keypoints) per image pair, with a marginal impact on accuracy.

Advanced configuration

<details>
<summary>[Detail of all parameters - click to expand]</summary>

- ```n_layers```: Number of stacked self+cross attention layers. Reduce this value for faster inference at the cost of accuracy (continuous red line in the plot above). De
- ```flash```: Enable FlashAttention. Significantly increases the speed and reduces the memory consumption without any impact on accuracy. Default: True (LightGlue automat
- ```mp```: Enable mixed precision inference. Default: False (off)
- ```depth_confidence```: Controls the early stopping. A lower values stops more often at earlier layers. Default: 0.95, disable with -1.
- ```width_confidence```: Controls the iterative point pruning. A lower value prunes more points earlier. Default: 0.99, disable with -1.
- ```filter_threshold```: Match confidence. Increase this value to obtain less, but stronger matches. Default: 0.1

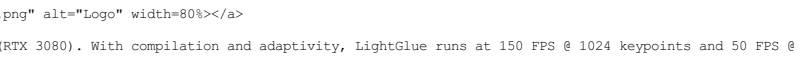
</details>

The default values give a good trade-off between speed and accuracy. To maximize the accuracy, use all keypoints and disable the adaptive mechanisms:
```python
extractor = SuperPoint(max_num_keypoints=None)
matcher = LightGlue(features='superpoint', depth_confidence=-1, width_confidence=-1)
```

To increase the speed with a small drop of accuracy, decrease the number of keypoints and lower the adaptive thresholds:
```python
extractor = SuperPoint(max_num_keypoints=1024)
matcher = LightGlue(features='superpoint', depth_confidence=0.9, width_confidence=0.95)
```

The maximum speed is obtained with a combination of:
- [FlashAttention](https://arxiv.org/abs/2205.14135): automatically used when ```torch >= 2.0``` or if [installed from source](https://github.com/HazyResearch/flash-attent
- PyTorch compilation, available when ```torch >= 2.0```:
```python
matcher = matcher.eval().cuda()
matcher.compile(mode='reduce-overhead')
```
For inputs with fewer than 1536 keypoints (determined experimentally), this compiles LightGlue but disables point pruning (large overhead). For larger input sizes, it autc

Benchmark

  <a> 

  Benchmark results on GPU (RTX 3080). With compilation and adaptivity, LightGlue runs at 150 FPS @ 1024 keypoints and 50 FPS @ 4096 keypoints per image. This is a 4-1

  <a>

```
  <br>
  <em>Benchmark results on CPU (Intel i7 10700K). LightGlue runs at 20 FPS @ 512 keypoints. </em>
</p>

Obtain the same plots for your setup using our [benchmark script](benchmark.py):
```
python benchmark.py [--device cuda] [--add_superglue] [--num_keypoints 512 1024 2048 4096] [--compile]
```

<details>
<summary>[Performance tip - click to expand]</summary>

Note: **Point pruning** introduces an overhead that sometimes outweighs its benefits.
Point pruning is thus enabled only when the there are more than N keypoints in an image, where N is hardware-dependent.
We provide defaults optimized for current hardware (RTX 30xx GPUs).
We suggest running the benchmark script and adjusting the thresholds for your hardware by updating `LightGlue.pruning_keypoint_thresholds['cuda']`.

</details>

## Training and evaluation

With [Glue Factory](https://github.com/cvg/glue-factory), you can train LightGlue with your own local features, on your own dataset!
You can also evaluate it and other baselines on standard benchmarks like HPatches and MegaDepth.

## Other links
- [hloc - the visual localization toolbox](https://github.com/cvg/Hierarchical-Localization/): run LightGlue for Structure-from-Motion and visual localization.
- [LightGlue-ONNX](https://github.com/fabio-sim/LightGlue-ONNX): export LightGlue to the Open Neural Network Exchange (ONNX) format with support for TensorRT and OpenVINO.
- [Image Matching WebUI](https://github.com/Vincentqyw/image-matching-webui): a web GUI to easily compare different matchers, including LightGlue.
- [kornia](https://kornia.readthedocs.io) now exposes LightGlue via the interfaces [`LightGlue`](https://kornia.readthedocs.io/en/latest/feature.html#kornia.feature.LightG

## BibTeX citation
If you use any ideas from the paper or code from this repo, please consider citing:

```txt
@inproceedings{lindenberger2023lightglue,
  author    = {Philipp Lindenberger and
               Paul-Edouard Sarlin and
               Marc Pollefeys},
  title     = {{LightGlue: Local Feature Matching at Light Speed}},
  booktitle = {ICCV},
  year      = {2023}
}
```


License
The pre-trained weights of LightGlue and the code provided in this repository are released under the [Apache-2.0 license](./LICENSE). [DISK](https://github.com/cvlab-epfl/
```

tree.txt

```
LightGlue/
├── assets/
│   ├── architecture.svg
│   ├── benchmark.png
│   ├── benchmark_cpu.png
│   ├── DSC_0410.JPG
│   ├── DSC_0411.JPG
│   ├── easy_hard.jpg
│   ├── sacre_coeur1.jpg
│   ├── sacre_coeur2.jpg
│   └── teaser.svg
├── benchmark.py
├── demo.ipynb
├── demo.py
├── images/
│   ├── desk_fix.jpg
│   ├── desk_fix2.jpg
│   └── desk_vedio.mp4
├── LICENSE
├── lightglue/
│   ├── aliked.py
│   ├── disk.py
│   ├── dog_hardnet.py
│   ├── lightglue.py
│   ├── sift.py
│   ├── superpoint.py
│   ├── utils.py
│   └── viz2d.py
├── lightglue.egg-info/
│   ├── dependency_links.txt
│   ├── PKG-INFO
│   ├── requires.txt
│   ├── SOURCES.txt
│   └── top_level.txt
├── pyproject.toml
├── README.md
└── requirements.txt
```