Tutorial: Hierarchical Refinement for Large-Scale Optimal Transport

This tutorial presents the implementation of "Hierarchical Refinement" (HiRef), a method for solving large-scale optimal transport problems with linear space complexity. This approach is based on the paper "Hierarchical Refinement: Optimal Transport to Infinity and Beyond" by Halmos et al. (2025).

Introduction

Optimal transport (OT) is a mathematical framework for comparing probability distributions. While very powerful, it typically suffers from quadratic space complexity, limiting its application to datasets of moderate size. This implementation allows computing bijective correspondences between large-scale datasets, exceeding one million samples.

1. Setup and Imports

Let's start by importing the necessary libraries:

python

```python
from typing import List, Callable, Union, Dict, Any
import random
import operator
import functools
from functools import reduce

import numpy as np
import torch
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

import jax
import jax.numpy as jnp
from ott.geometry import costs, pointcloud
from ott.tools import sinkhorn_divergence, progot
from ott.problems.linear import linear_problem
from ott.solvers.linear import sinkhorn
import torchvision.transforms as transforms
from torch.utils.data import Dataset, DataLoader
import torch
import torchvision.models as models
import os
from PIL import Image
from tqdm import tqdm
import pickle

from ott.geometry import geometry
from ott.problems.linear import linear_problem
from ott.solvers.linear import sinkhorn
```

2. Basic Algorithms for Optimal Transport

2.1 Sinkhorn with epsilon-schedule

Entropic optimal transport with the Sinkhorn algorithm is the foundation on which we'll build. Here's an implementation of the log-stabilized Sinkhorn:

python

```python
def ott_log_sinkhorn(grad,
                     a,
                     b,
                     gamma_k,
                     max_iter = 50,
                     balanced = True,
                     unbalanced = False,
                     tau = None,
                     tau2 = None):
    """
    grad: cost matrix (n, m)
    a: source histogram (n,)
    b: target histogram (m,)
    gamma_k: inverse regularization (1/epsilon)
    """
    epsilon = 1.0 / gamma_k

    # Choose tau for margins
    if balanced and not unbalanced:
        tau_a, tau_b = 1.0, 1.0
    elif not balanced and unbalanced:
        tau_a = tau / (tau + epsilon)
        tau_b = tau2 / (tau2 + epsilon) if tau2 is not None else tau_a
    else:  # semi-relaxed
        tau_a, tau_b = 1.0, tau / (tau + epsilon)

    # Entropic geometry on the cost matrix
    geom = geometry.Geometry(cost_matrix=grad, epsilon=epsilon)

    # Construct the linear problem
    prob = linear_problem.LinearProblem(
        geom,
        a=a,
        b=b,
        tau_a=tau_a,
        tau_b=tau_b
    )

    # Sinkhorn solver
    solver = sinkhorn.Sinkhorn(max_iterations=max_iter)
    out = solver(prob)

    return out.matrix
```

2.2 Utility Functions for Optimal Transport

python

```python
def utils__Delta(vark, varkm1, gamma_k):
    return (gamma_k**-2) * (jnp.linalg.norm(vark[0] - varkm1[0]) + jnp.linalg.norm(vark[1] - va

def utils__random_simplex_sample(key, N, dtype = jnp.float64):
    """
    Draws a random point from the (N-1)-simplex using normalized exponentiated Gaussian variate
    """
    # Sample N independent standard normals
    z = jax.random.normal(key, shape=(N,), dtype=dtype)
    # Exponentiate
    e = jnp.exp(z)
    # Normalize to sum to 1
    return e / jnp.sum(e)

def utils__initialize_couplings(a, b, gQ, gR, gamma, full_rank = True, key = jax.random.PRNGKey
    """
    Initialize coupling factors in JAX.
    """
    N1 = a.shape[0]
    N2 = b.shape[0]
    r = gQ.shape[0]
    r2 = gR.shape[0]

    one_N1 = jnp.ones((N1,), dtype=dtype)
    one_N2 = jnp.ones((N2,), dtype=dtype)

    if full_rank:
        # Full-rank initialization via log-Sinkhorn
        key, subkey = jax.random.split(key)
        C_random = jax.random.uniform(subkey, (N1, r), dtype=dtype)
        Q = ott_log_sinkhorn(C_random, a, gQ, gamma,
                             max_iter=max_iter,
                             balanced=True)

        key, subkey = jax.random.split(key)
        C_random = jax.random.uniform(subkey, (N2, r2), dtype=dtype)
        R = ott_log_sinkhorn(C_random, b, gR, gamma,
                             max_iter=max_iter,
                             balanced=True)

        # Compute updated inner marginals
        gR_new = R.T @ one_N2
        gQ_new = Q.T @ one_N1

        key, subkey = jax.random.split(key)
        C_random = jax.random.uniform(subkey, (r, r2), dtype=dtype)
```

```
        T = ott_log_sinkhorn(C_random, gQ_new, gR_new, gamma,
                             max_iter=max_iter,
                             balanced=True)

        # Inner inverse coupling
        if r == r2:
            Lambda = jnp.linalg.inv(T)
        else:
            Lambda = jnp.diag(1.0 / gQ_new) @ T @ jnp.diag(1.0 / gR_new)

    else:
        # Rank-2 initialization (Scetbon et al. 2021)
        if r != r2:
            raise ValueError("Rank-2 init requires equal inner ranks.")
        g = gQ
        lambd = jnp.minimum(jnp.min(a), jnp.min(b))
        lambd = jnp.minimum(lambd, jnp.min(g)) / 2.0

        # Sample or deterministic
        if rank2_random:
            key, *splits = random.split(key, 4)
            a1 = utils__random_simplex_sample(N1, splits[0], dtype)
            b1 = utils__random_simplex_sample(N2, splits[1], dtype)
            g1 = utils__random_simplex_sample(r, splits[2], dtype)
        else:
            g1 = jnp.arange(1, r + 1, dtype=dtype)
            g1 = g1 / jnp.sum(g1)
            a1 = jnp.arange(1, N1 + 1, dtype=dtype)
            a1 = a1 / jnp.sum(a1)
            b1 = jnp.arange(1, N2 + 1, dtype=dtype)
            b1 = b1 / jnp.sum(b1)

        a2 = (a - lambd * a1) / (1 - lambd)
        b2 = (b - lambd * b1) / (1 - lambd)
        g2 = (g - lambd * g1) / (1 - lambd)

        Q = lambd * jnp.outer(a1, g1) + (1 - lambd) * jnp.outer(a2, g2)
        R = lambd * jnp.outer(b1, g1) + (1 - lambd) * jnp.outer(b2, g2)

        gR_new = R.T @ one_N2
        gQ_new = Q.T @ one_N1

        T = (1 - lambd) * jnp.diag(g) + lambd * jnp.outer(gR_new, gQ_new)
        Lambda = jnp.linalg.inv(T)

    return Q, R, T, Lambda
```

3. Implementation of the FRLC Solver (Low-Rank Optimal Transport)

The paper relies on a low-rank optimal transport solver called FRLC (Factor Relaxation with Latent Coupling). Here's its implementation:

3.1 Gradient Computation for Wasserstein Problems

python

```python
def gd_Wasserstein_Grad(C_or_Cfactors, Q, R, Lambda, full_grad=True, low_rank=False):
    if low_rank:
        C1, C2 = C_or_Cfactors
        gradQ = C1 @ ((C2 @ R) @ Lambda.T)
    else:
        C = C_or_Cfactors
        gradQ = (C @ R) @ Lambda.T

    if full_grad:
        N1 = Q.shape[0]
        one_N1 = jnp.ones((N1,), dtype=Q.dtype)
        gQ = Q.T @ one_N1
        w1 = jnp.diag((gradQ.T @ Q) @ jnp.diag(1.0 / gQ))
        gradQ = gradQ - jnp.outer(one_N1, w1)

    if low_rank:
        gradR = C2.T @ ((C1.T @ Q) @ Lambda)
    else:
        gradR = (C.T @ Q) @ Lambda

    if full_grad:
        N2 = R.shape[0]
        one_N2 = jnp.ones((N2,), dtype=R.dtype)
        gR = R.T @ one_N2
        w2 = jnp.diag(jnp.diag(1.0 / gR) @ (R.T @ gradR))
        gradR = gradR - jnp.outer(one_N2, w2)

    return gradQ, gradR
```

3.2 Gradient Computation for Wasserstein and Gromov-Wasserstein Problems

python

```python
def gd__compute_grad_A(C, Q, R, Lambda, gamma,
                       semiRelaxedLeft, semiRelaxedRight,
                       Wasserstein=True, FGW=False,
                       A=None, B=None,
                       alpha=0.0,
                       unbalanced=False,
                       full_grad=True,
                       low_rank=False,
                       C_factors=None, A_factors=None, B_factors=None):
    """
    JAX version of gradient computation for Wasserstein, GW and FGW.
    If `low_rank` is True, it uses low-rank factorized cost matrices.
    """

    r = Lambda.shape[0]
    one_r = jnp.ones((r,))
    One_rr = jnp.outer(one_r, one_r)

    if low_rank:
        # Assume: A_factors = (A1, A2), B_factors = (B1, B2), C_factors = (C1, C2)
        A1, A2 = A_factors
        B1, B2 = B_factors

        gradQ = -4 * (A1 @ (A2 @ (Q @ Lambda @ ((R.T @ B1) @ (B2 @ R)) @ Lambda.T)))
        gradR = -4 * (B1 @ (B2 @ (R @ (Lambda.T @ ((Q.T @ A1) @ (A2 @ Q)) @ Lambda))))

        if full_grad:
            N1, N2 = Q.shape[0], R.shape[0]
            one_N1 = jnp.ones((N1,))
            one_N2 = jnp.ones((N2,))
            gQ = Q.T @ one_N1
            gR = R.T @ one_N2

            MR = Lambda.T @ ((Q.T @ A1) @ (A2 @ Q)) @ Lambda @ ((R.T @ B1) @ (B2 @ R)) @ jnp.di
            MQ = Lambda @ ((R.T @ B1) @ (B2 @ R)) @ Lambda.T @ ((Q.T @ A1) @ (A2 @ Q)) @ jnp.di
            gradQ += 4 * jnp.outer(one_N1, jnp.diag(MQ))
            gradR += 4 * jnp.outer(one_N2, jnp.diag(MR))

        # Wasserstein gradients in low-rank form
        gradQW, gradRW = gd__Wasserstein_Grad(C_factors, Q, R, Lambda, full_grad=full_grad, low

        gradQ = (1 - alpha) * gradQW + (alpha / 2) * gradQ
        gradR = (1 - alpha) * gradRW + (alpha / 2) * gradR

    else:
        if Wasserstein:
```

```
            gradQ, gradR = gd_Wasserstein_Grad(C, Q, R, Lambda, full_grad=full_grad)
        elif A is not None and B is not None:
            if not semiRelaxedLeft and not semiRelaxedRight and not unbalanced:
                gradQ = -4 * (A @ Q) @ Lambda @ (R.T @ B @ R) @ Lambda.T
                gradR = -4 * (B @ R @ Lambda.T) @ (Q.T @ A @ Q) @ Lambda
            elif semiRelaxedRight:
                gradQ = -4 * (A @ Q) @ Lambda @ (R.T @ B @ R) @ Lambda.T
                gradR = 2 * (B @ B) @ R @ One_rr - 4 * (B @ R @ Lambda.T) @ (Q.T @ A @ Q) @ Lam
            elif semiRelaxedLeft:
                gradQ = 2 * (A @ A) @ Q @ One_rr - 4 * (A @ Q) @ Lambda @ (R.T @ B @ R) @ Lambd
                gradR = -4 * (B @ R @ Lambda.T) @ (Q.T @ A @ Q) @ Lambda
            elif unbalanced:
                gradQ = 2 * (A @ A) @ Q @ One_rr - 4 * (A @ Q) @ Lambda @ (R.T @ B @ R) @ Lambd
                gradR = 2 * (B @ B) @ R @ One_rr - 4 * (B @ R @ Lambda.T) @ (Q.T @ A @ Q) @ Lam

            if full_grad:
                N1, N2 = Q.shape[0], R.shape[0]
                one_N1 = jnp.ones((N1,))
                one_N2 = jnp.ones((N2,))
                gQ = Q.T @ one_N1
                gR = R.T @ one_N2
                F = Q @ Lambda @ R.T
                MR = Lambda.T @ Q.T @ A @ F @ B @ R @ jnp.diag(1. / gR)
                MQ = Lambda @ R.T @ B @ F.T @ A @ Q @ jnp.diag(1. / gQ)
                gradQ += 4 * jnp.outer(one_N1, jnp.diag(MQ))
                gradR += 4 * jnp.outer(one_N2, jnp.diag(MR))

            if FGW:
                gradQW, gradRW = gd_Wasserstein_Grad(C, Q, R, Lambda, full_grad=full_grad)
                gradQ = (1 - alpha) * gradQW + alpha * gradQ
                gradR = (1 - alpha) * gradRW + alpha * gradR
        else:
            raise ValueError("Provide either Wasserstein=True or distance matrices A and B for

    normalizer = jnp.max(jnp.array([jnp.max(jnp.abs(gradQ)), jnp.max(jnp.abs(gradR))]))
    gamma_k = gamma / normalizer

    return gradQ, gradR, gamma_k
```

3.3 Gradient Computation with Respect to the Transport Plan T

python

```python
def gd__compute_grad_B(C=None, Q=None, R=None, Lambda=None, gQ=None, gR=None, gamma=1.0,
                       Wasserstein=True, FGW=False, A=None, B=None, alpha=0.0,
                       low_rank=False, C_factors=None, A_factors=None, B_factors=None):
    '''
    JAX version of the Wasserstein / GW / FGW gradient w.r.t. the transport plan T.
    Supports both full and low-rank computation.
    '''
    if low_rank:
        # Gradient using low-rank approximation
        C1, C2 = C_factors
        gradLambda = (1 - alpha) * ((Q.T @ C1) @ (C2 @ R))

        if A_factors is not None and B_factors is not None:
            A1, A2 = A_factors
            B1, B2 = B_factors
            grad_GW = -4 * ((Q.T @ A1) @ (A2 @ Q)) @ Lambda @ ((R.T @ B1) @ (B2 @ R))
            gradLambda += (alpha / 2.0) * grad_GW
    else:
        if Wasserstein:
            gradLambda = Q.T @ C @ R
        else:
            gradLambda = -4 * Q.T @ A @ Q @ Lambda @ R.T @ B @ R
            if FGW:
                gradLambda = (1 - alpha) * (Q.T @ C @ R) + alpha * gradLambda

    # Final gradient
    gradT = jnp.diag(1.0 / gQ) @ gradLambda @ jnp.diag(1.0 / gR)
    gamma_T = gamma / jnp.max(jnp.abs(gradT))
    return gradT, gamma_T
```

3.4 FRLC Solver for Optimal Transport

python

```python
def FRLC_opt(C, A=None, B=None, C_factors=None, A_factors=None, B_factors=None,
             a=None, b=None, tau_in=50, tau_out=50, gamma=90, r=10, r2=None,
             max_iter=200, Wasserstein=True, returnFull=False, FGW=False, alpha=0.0,
             initialization='Full', init_args=None, full_grad=True,
             convergence_criterion=True, tol=1e-5, min_iter=25,
             max_inneriters_balanced=300, max_inneriters_relaxed=50,
             diagonalize_return=False, low_rank=False):
    """
    FRLC Optimal Transport solver. Supports dense and low-rank cost formulations.
    """
    if r2 is None:
        r2 = r

    n, m = (C.shape[0], C.shape[1]) if C is not None else (C_factors[0].shape[0], C_factors[1].

    if a is None:
        a = jnp.ones(n) / n
    if b is None:
        b = jnp.ones(m) / m

    # Initialize coupling decomposition Q, R, Lambda
    if initialization == 'Full':
        T0 = ott_log_sinkhorn(C, a, b, max_inneriters_balanced)
    elif initialization == 'Identity':
        T0 = jnp.outer(a, b)
    elif initialization == 'Rank-1':
        T0 = jnp.outer(a, b)
    elif initialization == 'Random':
        T0 = jax.random.uniform(jax.random.PRNGKey(0), shape=(n, m))
        T0 = T0 / T0.sum()
    elif initialization == 'Given' and init_args is not None:
        T0 = init_args
    else:
        raise ValueError("Unsupported initialization method")

    U, s, Vt = jnp.linalg.svd(T0, full_matrices=False)
    Q = U[:, :r]
    R = Vt[:r, :].T
    Lambda = jnp.diag(s[:r])

    err = jnp.inf
    it = 0
    converged = False

    while not converged and it < max_iter:
        # Gradient via factorized formulation or dense formulation
```

```
        gradQ, gradR, gamma_k = gd__compute_grad_A(
            C, Q, R, Lambda, gamma,
            False, False,
            Wasserstein=Wasserstein, FGW=FGW, A=A, B=B, alpha=alpha,
            full_grad=full_grad, low_rank=low_rank,
            C_factors=C_factors, A_factors=A_factors, B_factors=B_factors
        )

        # Update Q
        Q -= gamma_k * gradQ
        Q = Q / jnp.linalg.norm(Q, axis=0, keepdims=True)

        T_mid = Q @ Lambda @ R.T

        # Solve subproblem for R
        T_R = ott_log_sinkhorn(C if not low_rank else None, a, b, max_inneriters_relaxed, tau_i
        U, s, Vt = jnp.linalg.svd(T_R, full_matrices=False)
        R = Vt[:r2, :].T
        Lambda = jnp.diag(s[:r2])

        gQ = Q.T @ jnp.ones(n)
        gR = R.T @ jnp.ones(m)

        gradT, gamma_T = gd__compute_grad_B(
            C=C, A=A, B=B, Q=Q, R=R, Lambda=Lambda, gQ=gQ, gR=gR,
            gamma=gamma, Wasserstein=Wasserstein, FGW=FGW, alpha=alpha,
            low_rank=low_rank, C_factors=C_factors, A_factors=A_factors, B_factors=B_factors
        )

        # Update R
        R -= gamma_k * gradR
        R = R / jnp.linalg.norm(R, axis=0, keepdims=True)

        T_mid = Q @ Lambda @ R.T

        # Solve subproblem for Q
        T_Q = ott_log_sinkhorn(C if not low_rank else None, a, b, max_inneriters_relaxed, tau_i
        U, s, Vt = jnp.linalg.svd(T_Q, full_matrices=False)
        Q = U[:, :r]
        Lambda = jnp.diag(s[:r])

        # Convergence check
        if convergence_criterion:
            err = jnp.linalg.norm(T_Q - T_mid, ord='fro') / jnp.linalg.norm(T_Q, ord='fro')
            if it >= min_iter and err < tol:
                converged = True
```

```
        it += 1

    if returnFull:
        return Q @ Lambda @ R.T
    elif diagonalize_return:
        return Q, Lambda, R.T
    else:
        return Q, R, Lambda
```

3.5 Computing the Optimal OT Cost

python

```
def FRLC_compute_OT_cost(X, Y, C=None, Monge_clusters=None, sq_Euclidean=True):
    """
    Compute the optimal transport cost in linear space and time (without coupling), in JAX.
    Supports squared Euclidean cost via OTT cost object.
    """
    if Monge_clusters is None or len(Monge_clusters) == 0:
        return 0.0

    def compute_pair_cost(pair):
        idx1, idx2 = pair
        if C is not None:
            return C[idx1, idx2]
        else:
            diff = X[idx1] - Y[idx2]
            if sq_Euclidean:
                return jnp.sum(diff**2)
            else:
                return jnp.linalg.norm(diff)

    pair_costs = jax.vmap(compute_pair_cost)(jnp.array(Monge_clusters))
    total_cost = jnp.sum(pair_costs)
    return total_cost / len(Monge_clusters)
```

4. Implementation of Hierarchical Refinement

The core of the paper is the "Hierarchical Refinement" algorithm that uses low-rank decompositions recursively to build a full transport plan between two datasets. The key insight is that optimal factors of low-rank optimal transport co-cluster points with their image under the Monge map.

4.1 Computing the Rank Annealing Schedule

python

```python
def rank_annealing__factors(n):
    """
    Return list of all factors of an integer
    """
    n = int(n)  # Conversion for compatibility with jnp.arange
    candidates = jnp.arange(1, jnp.floor(jnp.sqrt(n)) + 1).astype(int)
    divisible = (n % candidates) == 0
    factors1 = candidates[divisible]
    factors2 = n // factors1
    all_factors = jnp.concatenate([factors1, factors2])
    unique_factors = jnp.unique(all_factors)
    return unique_factors

def rank_annealing__max_factor_lX(n, max_X):
    """
    Find max factor of n , such that max_factor ≤ max_X
    """
    factor_lst = rank_annealing__factors(n)
    factors_leq_max = factor_lst[factor_lst <= max_X]
    return jnp.max(factors_leq_max)

def rank_annealing__min_sum_partial_products_with_factors(n, k, C):
    """
    Dynamic program to compute the rank-schedule, subject to a constraint of intermediates beir

    Parameters
    ----------
    n: int
        The dataset size to be factored into a rank-scheduler. Assumed to be non-prime.
    k: int
        The depth of the hierarchy.
    C: int
        A constraint on the maximal intermediate rank across the hierarchy.
    """
    INF = 1e10  # Large constant instead of float('inf') for JAX compatibility

    dp = jnp.full((n+1, k+1), INF)
    choice = jnp.full((n+1, k+1), -1)

    def init_base_case(dp, choice):
        d = jnp.arange(1, n+1)
        mask = d <= C
        dp = dp.at[d[mask], 1].set(d[mask])
        choice = choice.at[d[mask], 1].set(d[mask])
        return dp, choice
```

```
    dp, choice = init_base_case(dp, choice)

    for t in range(2, k+1):
        for d in range(1, n+1):
            if dp[d, t-1] >= INF:
                continue
            for r in range(1, min(C, d)+1):
                if d % r == 0:
                    candidate = r + r * dp[d // r, t-1]
                    if candidate < dp[d, t]:
                        dp = dp.at[d, t].set(candidate)
                        choice = choice.at[d, t].set(r)

    if dp[n, k] >= INF:
        return None, []

    # Backtracking
    factors = []
    d_cur, t_cur = n, k
    while t_cur > 0:
        r_cur = int(choice[d_cur, t_cur])
        factors.append(r_cur)
        d_cur //= r_cur
        t_cur -= 1

    return dp[n, k], factors

def rank_annealing__optimal_rank_schedule(n, hierarchy_depth=6, max_Q=int(2**10), max_rank=16):
    """
    A function to compute the optimal rank-scheduler of refinement.

    Parameters
    ----------
    n: int
        Size of the input dataset -- cannot be a prime number
    hierarchy_depth: int
        Maximal permissible depth of the multi-scale hierarchy
    max_Q: int
        Maximal rank at terminal base case (before reducing the ≤ max_Q rank coupling to a 1-1
    max_rank: int
        Maximal rank at the intermediate steps of the rank-schedule
    """
    Q = int(rank_annealing__max_factor_lX(n, max_Q))
    ndivQ = int(n // Q)

    _, rank_schedule = rank_annealing__min_sum_partial_products_with_factors(ndivQ, hierarchy_d
    rank_schedule = sorted(rank_schedule)
```

```
    rank_schedule.append(Q)
    rank_schedule = [x for x in rank_schedule if x != 1]

    print(f'Optimized rank-annealing schedule: {rank_schedule}')

    assert functools.reduce(operator.mul, rank_schedule, 1) == n, "Error! Rank-schedule does nc

    return rank_schedule
```

4.2 Computing the Low-Rank Euclidean Cost Matrix

This function allows representing the squared Euclidean cost matrix in factorized form to avoid storing the full matrix, which is crucial for scaling the algorithm.

python

```
def compute_lr_sqeuclidean_matrix(X_s, X_t, rescale_cost: bool = False):
    """
    Adapted to JAX from the low-rank squared Euclidean cost decomposition,
    as in Scetbon, Cuturi & Peyré (2021), Section 3.5, Proposition 1.
    """

    ns, dim = X_s.shape
    nt, _ = X_t.shape

    # First low-rank term (source-side)
    sum_Xs_sq = jnp.sum(X_s ** 2, axis=1, keepdims=True)          # (ns, 1)
    ones_ns = jnp.ones((ns, 1), dtype=X_s.dtype)                  # (ns, 1)
```