Deep Accelerated Signal Processing (Week 4)

Erick Rodriguez
Advisor: Dr. HanQin Cai
Center for Research in Computer Vision REU

June 20, 2025

Problem

Hankel matrices appear often in signal processing problems

Problem

Hankel matrices appear often in signal processing problems

- Addresses limitations of the discrete Fourier transform

Problem

Hankel matrices appear often in signal processing problems

- Addresses limitations of the discrete Fourier transform
- Hankelized signals can be decomposed to learn underlying frequencies

Problem

Hankel matrices appear often in signal processing problems

- Addresses limitations of the discrete Fourier transform
- Hankelized signals can be decomposed to learn underlying frequencies

We aim to advance the state of the art in Hankel recovery algorithms

Problem

Hankel matrices appear often in signal processing problems

- Addresses limitations of the discrete Fourier transform
- Hankelized signals can be decomposed to learn underlying frequencies

We aim to advance the state of the art in Hankel recovery algorithms

- Learnability

Problem

Hankel matrices appear often in signal processing problems

- Addresses limitations of the discrete Fourier transform
- Hankelized signals can be decomposed to learn underlying frequencies

We aim to advance the state of the art in Hankel recovery algorithms

- Learnability
- Scalability

Progress Update

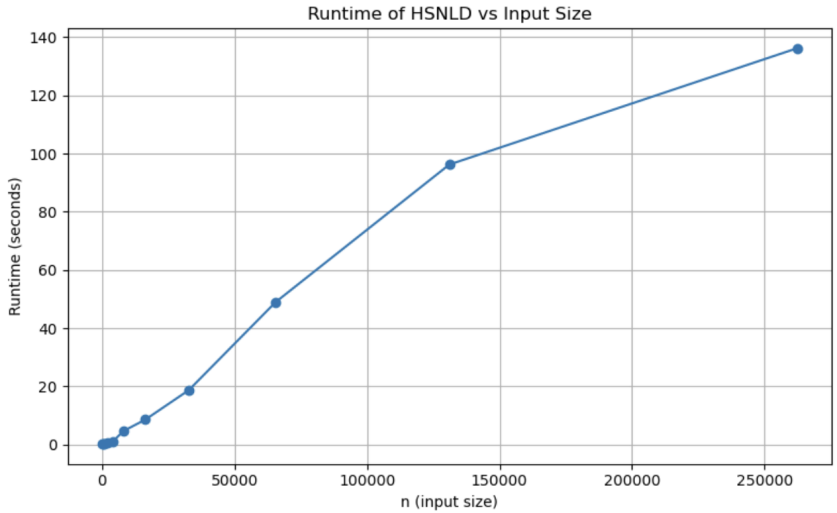


Figure: Runtime of HSNLD with PyTorch

Progress Update

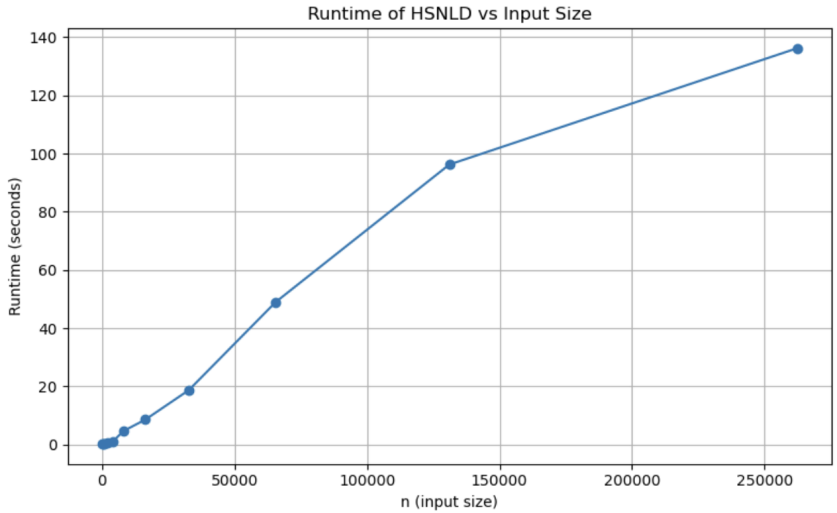


Figure: Runtime of HSNLD with PyTorch

Given an algorithmic complexity of $\mathcal{O}((r^2 n + rn \log n) \log(1/\varepsilon))$ tested with $r = 10$ and $\varepsilon = 1 \times 10^{-14}$, this graph scales nicely but shows a high runtime

Learning: Current Algorithm

Algorithm 1 **H**ankel **S**tructured **N**ewton-**L**ike **D**escent (HSNLD)

1: **Input:** $\Pi_\Omega \boldsymbol{f}$: partial observation on the corrupted Hankel matrix in reweighted vector form; r: the rank of underlying Hankel matrix; p: observation rate; α: outlier density; $\{\gamma_k\}$: parameters for sparsification operator; η: step size; K: maximum number of iterations.

2: **Initialization:** Let $\boldsymbol{L}_0, \boldsymbol{R}_0$ be produced by (6).

3: **for** $k = 0, 1, \ldots, K-1$ **do**

4: $\quad \boldsymbol{z}_{k+1} = \mathcal{G}^*(\boldsymbol{L}_k \boldsymbol{R}_k^*)$

5: $\quad \boldsymbol{s}_{k+1} = \Gamma_{\gamma_k \alpha p}(\Pi_\Omega(\boldsymbol{f} - \boldsymbol{z}_{k+1}))$

6: $$\begin{bmatrix} \boldsymbol{L}_{k+1} \\ \boldsymbol{R}_{k+1} \end{bmatrix} = \mathcal{P}_{\mathcal{C}} \begin{bmatrix} \boldsymbol{L}_k - \eta \nabla_{\boldsymbol{L}} \ell(\boldsymbol{L}_k, \boldsymbol{R}_k; \boldsymbol{s}_{k+1}) \left(\boldsymbol{R}_k^* \boldsymbol{R}_k\right)^{-1} \\ \boldsymbol{R}_k - \eta \nabla_{\boldsymbol{R}} \ell(\boldsymbol{L}_k, \boldsymbol{R}_k; \boldsymbol{s}_{k+1}) \left(\boldsymbol{L}_k^* \boldsymbol{L}_k\right)^{-1} \end{bmatrix}$$

7: **end for**

8: **Output:** $\boldsymbol{z}_{\text{output}} = \mathcal{G}^*(\boldsymbol{L}_K \boldsymbol{R}_K^*)$: recovered Hankel matrix in vector form.

Figure: HSNLD Algorithm (Cai et al., 2025)

Learning: Learned RPCA

Algorithm 1 Learned Robust PCA (LRPCA)

Input: $\boldsymbol{Y} = \boldsymbol{X}_\star + \boldsymbol{S}_\star$: sparsely corrupted matrix; r: the rank of underlying low-rank matrix; $\{\zeta_k\}$: a set of *learned* thresholding values; $\{\eta_k\}$: a set of *learned* step sizes.
// Initialization:
$\boldsymbol{S}_0 = \mathcal{S}_{\zeta_0}(\boldsymbol{Y})$
$[\boldsymbol{U}_0, \boldsymbol{\Sigma}_0, \boldsymbol{V}_0] = \mathrm{SVD}_r(\boldsymbol{Y} - \boldsymbol{S}_0)$
$\boldsymbol{L}_0 = \boldsymbol{U}_0 \boldsymbol{\Sigma}_0^{1/2}, \qquad \boldsymbol{R}_0 = \boldsymbol{V}_0 \boldsymbol{\Sigma}_0^{1/2}$
// Iterative updates:
while Not(Stopping Condition) **do**
$\boldsymbol{S}_{k+1} = \mathcal{S}_{\zeta_{k+1}}(\boldsymbol{Y} - \boldsymbol{L}_k \boldsymbol{R}_k^\top)$
$\boldsymbol{L}_{k+1} = \boldsymbol{L}_k - \eta_{k+1}(\boldsymbol{L}_k \boldsymbol{R}_k^\top + \boldsymbol{S}_{k+1} - \boldsymbol{Y}) \boldsymbol{R}_k (\boldsymbol{R}_k^\top \boldsymbol{R}_k)^{-1}$
$\boldsymbol{R}_{k+1} = \boldsymbol{R}_k - \eta_{k+1}(\boldsymbol{L}_k \boldsymbol{R}_k^\top + \boldsymbol{S}_{k+1} - \boldsymbol{Y})^\top \boldsymbol{L}_k (\boldsymbol{L}_k^\top \boldsymbol{L}_k)^{-1}$
end while
Output: $\boldsymbol{X}_K = \boldsymbol{L}_K \boldsymbol{R}_K^\top$: the recovered low-rank matrix.

Figure: LRPCA Algorithm (Cai et al., 2021)

Next Steps & Contributions

```
def forward(self, Y0_t, r, U0_t, V0_t, num_l):

        ## Initialization
        S_t =  self.thre(Y0_t, self.ths_v[0])
        L, Sigma, R = torch.svd_lowrank(Y0_t-S_t, q = r, niter = 4)
        Sigsqrt = torch.diag(torch.sqrt(Sigma))
        U_t = torch.mm(L, Sigsqrt)
        V_t = torch.mm(R, Sigsqrt)

## Main Loop in LRPCA
        for t in range(1, num_l):
                YmUV = Y0_t - torch.mm(U_t, V_t.t())
                S_t =  self.thre(YmUV, self.ths_v[t])
                E_t = YmUV - S_t
                Vkernel = torch.inverse(V_t.t() @ V_t)
                Ukernel = torch.inverse(U_t.t() @ U_t)
                Unew = U_t + self.step[t] * (torch.mm(E_t,V_t) @ Vkernel)
                Vnew = V_t + self.step[t] * (torch.mm(U_t.t(),E_t).t() @ Ukernel)
                U_t = Unew
                V_t = Vnew

        ## loss function in training
        loss = (torch.mm(U_t, V_t.t()) - torch.mm(U0_t, V0_t.t())).norm()
        return loss
```

Figure: The forward pass in LRPCA's FRMNN model

Next Steps & Contributions

```
def forward(self, Y0_t, r, U0_t, V0_t, num_l):

        ## Initialization
        S_t =  self.thre(Y0_t, self.ths_v[0])
        L, Sigma, R = torch.svd_lowrank(Y0_t-S_t, q = r, niter = 4)
        Sigsqrt = torch.diag(torch.sqrt(Sigma))
        U_t = torch.mm(L, Sigsqrt)
        V_t = torch.mm(R, Sigsqrt)

## Main Loop in LRPCA
        for t in range(1, num_l):
                YmUV = Y0_t - torch.mm(U_t, V_t.t())
                S_t =  self.thre(YmUV, self.ths_v[t])
                E_t = YmUV - S_t
                Vkernel = torch.inverse(V_t.t() @ V_t)
                Ukernel = torch.inverse(U_t.t() @ U_t)
                Unew = U_t + self.step[t] * (torch.mm(E_t,V_t) @ Vkernel)
                Vnew = V_t + self.step[t] * (torch.mm(U_t.t(),E_t).t() @ Ukernel)
                U_t = Unew
                V_t = Vnew

        ## loss function in training
        loss = (torch.mm(U_t, V_t.t()) - torch.mm(U0_t, V0_t.t())).norm()
        return loss
```

Figure: The forward pass in LRPCA's FRMNN model

Next, I will reimplement the FRMNN model on HSNLD

Thank You!

Questions?