Functions

In [ ]:

```
from torch.cuda import device_of
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import math
import time

def set_general_params(n_0 = 1, p_s = 1, B = 4, K = 4, P = 4, N = 2, L = 2,
                       T = 5,epochs=15, J=10,X_is_variable = False,
                       W_is_phase_only = True, W_is_block_diag = True,
                       train_size = 1500, batch_size = 20, pga_iters=400,
                       optimizer_learning_rate=0.03):
    """
    This function sets the general parameters of the network.
     used in order to save space in the code when running multiple simulations.
    :param n_0: noise power
    :param p_s: signal power
    :param B: number of frequency bins
    :param K: number of users to be served
    :param P: number of panels in the base station
    :param N: number of antennas in each panel
    :param L: number of outputs in each panel
    :param T: number of inputs to the CPU
    :param epochs: number of epochs in the training
    :param J: number of iteration to be unfolded , usually 10.
    :param X_is_variable: if True, X is a variable in the network. usually False.
    :param W_is_phase_only: if True, W is a phase matrix. usually True.
    :param W_is_block_diag: if True, W is a block diagonal matrix. usually True.
    :param train_size: number of samples in the training set.
    :param batch_size: batch size in the training part.
    :param pga_iters: number of iterations in the PGA algorithm to be compared with. in o
rder of 300-400.
    :param optimizer_learning_rate: learning rate of the optimizer. denoted as eta in the
paper.
    :return: all the parameters.
    """
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    M = P * N  # Total number of antennas in the Base station
    snr = p_s / n_0
    T_opt = max(math.floor(M * (K - L) / K + 1), K)  # minimal number of inputs to the C
PU that allows loss-less processing. as described in "Trade-Offs in Decentralized Multi-A
ntenna Architectures: The WAX Decomposition".
    if T==5: # build the A matrix using the example given in "Trade-Offs in Decentralized
Multi-Antenna Architectures: The WAX Decomposition"
        A = torch.empty((L * P, T))
        A[:T, :] = torch.eye(T)
        A[T:, :3] = torch.eye(3)
        A[T:, 3:] = 1
        A = A.to(dtype=torch.cfloat).to(device)
    if T == 61: # build the A matrix using the example given in "Trade-Offs in Decentral
ized Multi-Antenna Architectures: The WAX Decomposition"
        #Make sure that L*P = 100
        A = torch.empty((100, 61))
        A[ :61, :] = torch.eye(61)
        A[61:, :39] = torch.eye(39)
        A[61:83, 39:] = torch.eye(22)
        A[83:, 39:56] = torch.eye(17)
        A[83:88, 56:] = torch.eye(5)
        A[88:93, 56:] = torch.eye(5)
        A[93:98, 56:] = torch.eye(5)
        A[98:, 56:58] = torch.eye(2)
        A[98:, 58:60] = torch.eye(2)
```

```
        A = A.to(dtype=torch.cfloat).to(device)
    Ik = torch.eye(T, dtype=torch.cfloat).to(device)
    Im = torch.eye(M, dtype=torch.cfloat).to(device)
    D = torch.zeros((M, L * P)).to(device)
    for p in range(P):
        p_n, p_l = p * N, p * L
        D[p_n:(p_n + N), p_l:(p_l + L)] = torch.ones((N, L))
    return device, n_0, p_s, B, K, P, N, M, L, T, snr, T_opt, A, Ik, Im, D,J, epochs, W
_is_block_diag, W_is_phase_only, X_is_variable, train_size, batch_size, pga_iters, optimi
zer_learning_rate


def gen_data(seed = 43, train_size = 1500, batch_size = 20, valid_size = 250):
    device = torch.device('cpu')  # generating all samples on the CPU for consistency
    torch.manual_seed(seed)
    H_train = torch.randn((train_size, B, M, K), dtype=torch.cfloat, device=device) # cr
eating H_train in an i.i.d. manner
    H_valid = torch.randn((valid_size, B, M, K), dtype=torch.cfloat, device=device) # cr
eating H_valid in an i.i.d. manner
    X = torch.eye(T, dtype=torch.cfloat)
    W = torch.randn((M, L*P), dtype=torch.cfloat)

    return H_train, H_valid, X, W


def project_onto_block_diagonal(w,D):
    """
    This function projects the matrix w onto the block diagonal matrix D.
    :param w: the matrix to be projected
    :param D: a predefined block diagonal matrix consisting of ones and zeros.
    :return: the projection of w onto a block diagonal matrix.
    """
    return w * D


def project_onto_phases(w):
    """
    This function projects the matrix w onto the phases of w. after projection, all entri
es of W has a unit magnitude.
    :param w: the matrix to be projected
    """
    W_phases = torch.exp(1j * torch.angle(w))
    return W_phases


class UnfoldedModel(nn.Module):
    """
    Building the unfolded model. in our code, this model is used as a skeleton for the Un
folded + Momentum algorithm.
    Significantly outperforms the PGA algorithm.
    Reaches almost the same performance as the Unfolded + Momentum algorithm.
    """

    def __init__(self, Mu_W, Mu_X, J):
        super().__init__()
        self.Mu_W = Mu_W
        self.Mu_X = Mu_X
        self.J = J
        self.D = D


    def forward(self, H, W, X, A, p_s, n_0):
        """
        This function describes the flow of the unfolded model. Here, the values of Mu_W
vary in each iteration.
        :param H: a matrix of size BxMxK containing the channel realizations for each fre
quency bin.
        :param W: the inital value of W. usually randomized. It is important not to take
a scaled identity matrix as the initial value of W, since it harms the convergence of the
algorithm.
        :param X: the inital value of X. set to be identity matrix.
        :param A: the initial value of A. predefined in the set_genral_params function.
```

```
        :return: the list of rates obtained in each iteration, the the final value of W a
nd X.
        """
        W_1 = W.detach()
        X_1 = X.detach()
        Rs = torch.zeros(self.J, dtype=torch.cfloat).to(device)
        # project W onto constraints:
        if W_is_phase_only:
            W_1 = project_onto_phases(W_1)
        if W_is_block_diag:
            W_1 = project_onto_block_diagonal(W_1, self.D)
        # starting the iterations:
        for j in range(self.J):
            dR_dW = self.calc_dR_dW(H, W_1, X_1, A, p_s, n_0)
            W_1 = W_1 + self.Mu_W[j] * dR_dW # making a gradient step
            if X_is_variable:
                dR_dX = self.calc_dR_dX(H, W_1, X_1, A, p_s, n_0)
                X_1 = X_1 + self.Mu_X[j] * dR_dX
            if W_is_phase_only: # projecting W onto constraints
                W_1 = project_onto_phases(W_1)
            if W_is_block_diag:
                W_1 = project_onto_block_diagonal(W_1,self.D)
            Rs[j] = self.calc_R(H, W_1, X_1, A, p_s, n_0)
        return Rs, W_1, X_1


    def calc_R(self, H, W, X, A, p_s, n_0):
        """gets a channel realization H and returns the achievable rate of it. H is of di
mension (B,M,K)"""
        R = 0
        B, M, K = H.shape
        G = W @ A @ X
        snr = p_s / n_0
        for b in range(B):
            h1 = H[b, :, :]
            g1 = G
            psudo_inv_g = torch.inverse(g1.transpose(0, 1).conj() @ g1)
            proj_mat = g1 @ psudo_inv_g @ g1.transpose(0, 1).conj()
            R += torch.log((Im + snr * proj_mat @ h1 @ h1.transpose(0, 1).conj()).det()
)
        return R/B


    def calc_max_r(self, H):
        """calclulates the maximal rate of the channel H, susbstituting G to be the match
ed filter. not used in the algorithm"""
        R = 0
        B, M, K = H.shape
        G = H # the matched filter is obtained when G = H^H
        snr = p_s / n_0
        for b in range(B): # iterating over each frequency bin
            h1 = H[b, :, :]
            g1 = G[b, :, :]
            psudo_inv_g = torch.inverse(g1.transpose(0, 1).conj() @ g1)
            proj_mat = g1 @ psudo_inv_g @ g1.transpose(0, 1).conj()
            R += torch.log((Im + snr * proj_mat @ h1 @ h1.transpose(0, 1).conj()).det()
)
        return R/B # averaging over the frequency bins


    def calc_dR_dW(self, H, W, X, A, p_s, n_0):
        """calculates the gradient of the rate with respect to W. the gradient has the sa
me dimension as W"""
        grad = 0
        for b in range(B): #iterating over each frequency bin
            h1 = H[b, :, :]
            snr = p_s / n_0
            X_con = X.conj()
            W_con = W.conj()
            H_con = h1.conj()
            G = W @ A @ X
            G_herm = G.transpose(0, 1).conj()
```

```
            h1_herm = h1.transpose(0, 1).conj()
            G_psu_inv = torch.inverse(G_herm @ G)
            Proj_mat = G @ G_psu_inv @ G_herm
            Z = snr * G_psu_inv @ G_herm @ h1 @ h1_herm @ G
            dR_dZ = torch.inverse(Ik + Z)
            grad += snr * (
                        (A @ X) @ dR_dZ @ (2 * G_psu_inv @ G_herm) @ h1 @ h1_herm @ (
Im - Proj_mat)).conj().transpose(0,
1)
        return grad / B # averaging over the frequency bins


    def calc_dR_dX(self, H, W, X, A, p_s, n_0):
        """calculates the gradient of the rate with respect to X. the gradient has the sa
me dimension as X"""
        """ not used in our code! """
        snr = p_s / n_0
        X_con = X.conj()
        W_con = W.conj()
        H_con = H.conj()
        G = W @ A @ X
        G_herm = G.transpose(0, 1).conj()
        H_herm = H.transpose(1, 2).conj()
        G_psu_inv = torch.inverse(G_herm @ G)
        Proj_mat = G @ G_psu_inv @ G_herm
        Z = snr * G_psu_inv @ G_herm @ H @ H_herm @ G
        dR_dZ = torch.inverse(Ik + Z)
        return snr * (dR_dZ @ (2 * G_psu_inv @ G_herm) @ H @ H_herm @ (Im - Proj_mat)
@ W @ A).conj().transpose(1, 2)


    def batch_rate(self, batch, W, X, A, p_s, n_0):
        """ Used in order to calculate the average rate of a batch. Calling the forward f
uncton for each sample in the batch"""
        batch_rate = 0
        for sample_num in range(batch.shape[0]):
            sample = batch[sample_num, :, :, :]
            Rs, W_1, X_1 = self.forward(sample, W, X, A, p_s, n_0)
            batch_rate += Rs
        return batch_rate / (batch.shape[0]), W_1, X_1


class PGAModel():
    def __init__(self, Mu, pga_iters):
        """
        The classical implementation of the PGA algorithm, a first order method the appro
ch a maximum in a convex problem.
        :param Mu: The step size of the algorithm, a hyperparameter
        :param pga_iters: The duration of the algorithm, a hyperparameter
        """
        super().__init__()
        self.Mu = Mu
        self.pga_iters = pga_iters
        self.D = D


    def forward(self, H, W, X, A, p_s, n_0):
        """ based on the forward of the unfolded algorithm, but the value of Mu is fixed
"""
        W_1 = W.detach()
        X_1 = X.detach()
        Rs = torch.zeros(self.pga_iters, dtype=torch.cfloat).to(device)
        if W_is_phase_only:
            W_1 = project_onto_phases(W_1)
        if W_is_block_diag:
            W_1 = project_onto_block_diagonal(W_1, self.D)
        for j in range(self.pga_iters):
            dR_dW = self.calc_dR_dW(H, W_1, X_1, A, p_s, n_0)
            W_1 = W_1 + self.Mu * dR_dW
            if X_is_variable:
                dR_dX = self.calc_dR_dX(H, W_1, X_1, A, p_s, n_0)
                X_1 = X_1 + self.Mu * dR_dX
```

```
            if W_is_phase_only:
                W_1 = project_onto_phases(W_1)
            if W_is_block_diag:
                W_1 = project_onto_block_diagonal(W_1, self.D)
            Rs[j] = self.calc_R(H, W_1, X_1, A, p_s, n_0)
        return Rs, W_1, X_1


    def calc_R(self, H, W, X, A, p_s, n_0):
        """gets a channel realization and returns the achievable rate of it. H is of dime
nsion (B,M,K)"""
        R = 0
        B, M, K = H.shape
        G = W @ A @ X
        snr = p_s / n_0
        for b in range(B):
            h1 = H[b, :, :]
            g1 = G
            psudo_inv_g = torch.inverse(g1.transpose(0, 1).conj() @ g1)
            proj_mat = g1 @ psudo_inv_g @ g1.transpose(0, 1).conj()
            I_plus_Z = Im + snr * proj_mat @ h1 @ h1.transpose(0, 1).conj()
            det_I_plus_Z = torch.det(I_plus_Z.to(torch.complex128))
            R += torch.log(det_I_plus_Z)
        return R/B


    def calc_max_r(self, H):
        """calclulates the maximal rate of the channel H, susbstituting G to be the match
ed filter. not used in the algorithm"""
        R = 0
        B, M, K = H.shape
        G = H
        snr = p_s / n_0
        for b in range(B):
            h1 = H[b, :, :]
            g1 = G[b, :, :]
            psudo_inv_g = torch.inverse(g1.transpose(0, 1).conj() @ g1)
            proj_mat = g1 @ psudo_inv_g @ g1.transpose(0, 1).conj()
            I_plus_Z = Im + snr * proj_mat @ h1 @ h1.transpose(0, 1).conj()
            det_I_plus_Z = torch.det(I_plus_Z.to(torch.complex128))
            R += torch.log(det_I_plus_Z)
        return R/B


    def calc_dR_dW(self, H, W, X, A, p_s, n_0):
        """calculates the gradient of the rate with respect to W. the gradient has the sa
me dimension as W"""
        grad= 0
        for b in range(B):
            h1 = H[b, :, :]
            snr = p_s / n_0
            X_con = X.conj()
            W_con = W.conj()
            H_con = h1.conj()
            G = W @ A @ X
            G_herm = G.transpose(0, 1).conj()
            h1_herm = h1.transpose(0, 1).conj()
            G_psu_inv = torch.inverse(G_herm @ G)
            Proj_mat = G @ G_psu_inv @ G_herm
            Z = snr * G_psu_inv @ G_herm @ h1 @ h1_herm @ G
            dR_dZ = torch.inverse(Ik + Z)
            grad += snr * ((A @ X) @ dR_dZ @ (2 * G_psu_inv @ G_herm) @ h1 @ h1_herm @
(Im - Proj_mat)).conj().transpose(0, 1)
        return grad / B


    def calc_dR_dX(self, H, W, X, A, p_s, n_0):
        """calculates the gradient of the rate with respect to X. the gradient has the sa
me dimension as X"""
        """ not used in our code! """
        snr = p_s / n_0
        X_con = X.conj()
```

```
        W_con = W.conj()
        H_con = H.conj()
        G = W @ A @ X
        G_herm = G.transpose(0,1).conj()
        H_herm = H.transpose(1, 2).conj()
        G_psu_inv = torch.inverse(G_herm @ G)
        Proj_mat = G @ G_psu_inv @ G_herm
        Z = snr * G_psu_inv @ G_herm @ H @ H_herm @ G
        dR_dZ = torch.inverse(Ik + Z)
        return snr * (dR_dZ @ (2 * G_psu_inv @ G_herm) @ H @ H_herm @ (Im - Proj_mat)
@ W @ A).conj().transpose(1, 2)


    def batch_rate(self, batch, W, X, A, p_s, n_0):
        """ Used in order to calculate the average rate of a batch. Calling the forward f
uncton for each sample in the batch"""
        batch_rate = 0
        for sample_num in range(batch.shape[0]):
            sample = batch[sample_num, :, :, :]
            Rs, W_1, X_1 = self.forward(sample, W, X, A, p_s, n_0)
            batch_rate += Rs
        return batch_rate / (batch.shape[0]), W_1, X_1


class PGAModelMomentum():
    def __init__(self, Mu,beta, pga_iters):
        """ Works exactly the same like the PGAModel, but has a value of beta, which is t
he momentum parameter. """
        super().__init__()
        self.Mu = Mu  # step size parameter.
        self.pga_iters = pga_iters  # number of iterations of the PGA algorithm.
        self.D = D  # projction matrix
        self.beta = beta  # momentum parameter.


    def batch_rate(self, batch, W, X, A, p_s, n_0):
        """ Used in order to calculate the average rate of a batch. Calling the forward f
uncton for each sample in the batch"""
        batch_rate = 0
        for sample_num in range(batch.shape[0]):
            sample = batch[sample_num, :, :, :]
            Rs, W_1, X_1 = self.forward(sample, W, X, A, p_s, n_0)
            batch_rate += Rs
        return batch_rate / (batch.shape[0]), W_1, X_1


    def forward(self, H, W, X, A, p_s, n_0):
        """
        This function describes the flow of the PGA + Momentum model.
        Mu and beta are fixed scalars.
       :param H: a matrix of size BxMxK containing the channel realizations for each freq
uency bin.
       :param W: the inital value of W. usually randomized. It is important not to take o
nes or zeros matrix as the initial value of W, since it harms the convergence of the algo
rithm.
       :param X: the inital value of X. set to be identity matrix.
       :param A: the initial value of A. predefined in the set_genral_params function.
       :return: the list of rates obtained in each iteration, the the final value of W an
d X.
       """
        W_1 = W.detach()
        X_1 = X.detach()
        if W_is_phase_only:
            W_1 = project_onto_phases(W_1)
        if W_is_block_diag:
            W_1 = project_onto_block_diagonal(W_1, self.D)
        W_0 = torch.zeros_like(W_1) # initizlized as the zero matrix.
        X_0 = torch.zeros_like(X_1) # initizlized as the zero matrix.
        Rs = torch.zeros(self.pga_iters, dtype=torch.cfloat).to(device)
        for j in range(self.pga_iters):
            dR_dW = self.calc_dR_dW(H, W_1, X_1, A, p_s, n_0)
            W_2 = W_1 + self.Mu * dR_dW + self.beta * (W_1 - W_0)
```

```
            if X_is_variable:
                dR_dX = self.calc_dR_dX(H, W_2, X_1, A, p_s, n_0)
                X_2 = X_1 + self.Mu * dR_dX + self.beta * (X_1 - X_0)
            else:
                X_2 = X_1
            if W_is_phase_only:
                W_2 = project_onto_phases(W_2)
            if W_is_block_diag:
                W_2 = project_onto_block_diagonal(W_2, self.D)
            Rs[j] = self.calc_R(H, W_2, X_2, A, p_s, n_0)
            # updating the matrices:
            W_0 = W_1
            W_1 = W_2
            X_0 = X_1
            X_1 = X_2
        return Rs, W_2, X_2


    def calc_R(self, H, W, X, A, p_s, n_0):
        """gets a channel realization and returns the achievable rate of it. h is of dime
nsion (B,M,K)"""
        R = 0
        B, M, K = H.shape
        G = W @ A @ X
        snr = p_s / n_0
        for b in range(B):
            h1 = H[b, :, :]
            g1 = G
            psudo_inv_g = torch.inverse(g1.transpose(0, 1).conj() @ g1)
            proj_mat = g1 @ psudo_inv_g @ g1.transpose(0, 1).conj()
            I_plus_Z = Im + snr * proj_mat @ h1 @ h1.transpose(0, 1).conj()
            det_I_plus_Z = torch.det(I_plus_Z.to(torch.complex128))
            R += torch.log(det_I_plus_Z)
        return R/B


    def calc_max_r(self, H):
        """calclulates the maximal rate of the channel H, susbstituting G to be the match
ed filter. not used in the algorithm"""
        R = 0
        B, M, K = H.shape
        G = H
        snr = p_s / n_0
        for b in range(B):
            h1 = H[b, :, :]
            g1 = G[b, :, :]
            psudo_inv_g = torch.inverse(g1.transpose(0, 1).conj() @ g1)
            proj_mat = g1 @ psudo_inv_g @ g1.transpose(0, 1).conj()
            I_plus_Z = Im + snr * proj_mat @ h1 @ h1.transpose(0, 1).conj()
            det_I_plus_Z = torch.det(I_plus_Z.to(torch.complex128))
            R += torch.log(det_I_plus_Z)
        return R/B


    def calc_dR_dW(self, H, W, X, A, p_s, n_0):
        """calculates the gradient of the rate with respect to W. the gradient has the sa
me dimension as W"""
        grad = 0
        for b in range(B):
            h1 = H[b, :, :]
            snr = p_s / n_0
            X_con = X.conj()
            W_con = W.conj()
            H_con = h1.conj()
            G = W @ A @ X
            G_herm = G.transpose(0, 1).conj()
            h1_herm = h1.transpose(0, 1).conj()
            G_psu_inv = torch.inverse(G_herm @ G)
            Proj_mat = G @ G_psu_inv @ G_herm
            Z = snr * G_psu_inv @ G_herm @ h1 @ h1_herm @ G
            dR_dZ = torch.inverse(Ik + Z)
```

```
            grad += snr * ((A @ X) @ dR_dZ @ (2 * G_psu_inv @ G_herm) @ h1 @ h1_herm @
(Im - Proj_mat)).conj().transpose(0,1)
        return grad / B


    def calc_dR_dX(self, H, W, X, A, p_s, n_0):
        """calculates the gradient of the rate with respect to X. the gradient has the sa
me dimension as X"""
        """ not used in our code! """
        snr = p_s / n_0
        X_con = X.conj()
        W_con = W.conj()
        H_con = H.conj()
        # Im = torch.eye(M)
        G = W @ A @ X
        G_herm = G.transpose(0, 1).conj()
        H_herm = H.transpose(1, 2).conj()
        G_psu_inv = torch.inverse(G_herm @ G)
        Proj_mat = G @ G_psu_inv @ G_herm
        Z = snr * G_psu_inv @ G_herm @ H @ H_herm @ G
        dR_dZ = torch.inverse(Ik + Z)
        return snr * (dR_dZ @ (2 * G_psu_inv @ G_herm) @ H @ H_herm @ (Im - Proj_mat)
@ W @ A).conj().transpose(1, 2)


class MomentumModel(nn.Module):
    """ The model that outperforms all others. This is the innovative part in our work ""
"
    def __init__(self, Mu_W, Mu_X,betas, J):
        """ Works exactly the same like the Unfolded Model, but has a value of beta, whic
h is the momentum parameter. """
        super().__init__()
        self.Mu_W = Mu_W  # the step size for each entry of W. varies within every itera
tion of the algorhtim. Has the shape of W.
        self.Mu_X = Mu_X
        self.J = J
        self.D = D
        self.betas = betas  # the momentum parameter for every entry of W. varies within
every iteration of the algorhtim. Has the shape of W.


    def batch_rate(self, batch, W, X, A, p_s, n_0):
        """ Used in order to calculate the average rate of a batch. Calling the forward f
uncton for each sample in the batch """
        batch_rate = 0
        for sample_num in range(batch.shape[0]):
            sample = batch[sample_num, :, :, :]
            Rs, W_1, X_1 = self.forward(sample, W, X, A, p_s, n_0)
            batch_rate += Rs
        return batch_rate / (batch.shape[0]), W_1, X_1


    def forward(self, H, W, X, A, p_s, n_0):
        """
        This function describes the flow of the Unfolded PGA + Momentum model.
        Here, the values of Mu_W and beta vary in each iteration.
        Mu_W and beta are matrices with the same size as W.
       :param H: a matrix of size BxMxK containing the channel realizations for each freq
uency bin.
       :param W: the inital value of W. usually randomized. It is important not to take o
nes or zeros matrix as the initial value of W, since it harms the convergence of the algo
rithm.
       :param X: the inital value of X. set to be identity matrix.
       :param A: the initial value of A. predefined in the set_genral_params function.
       :return: the list of rates obtained in each iteration, the the final value of W an
d X.
       """
        W_1 = W.detach()
        X_1 = X.detach()
        # projecting W on the constraints:
        if W_is_phase_only:
```

```
            W_1 = project_onto_phases(W_1)
        if W_is_block_diag:
            W_1 = project_onto_block_diagonal(W_1, self.D)
        W_0 = torch.zeros_like(W_1)
        Rs = torch.zeros(self.J, dtype=torch.cfloat).to(device)
        # Starting the forward path:
        for j in range(self.J):
            dR_dW = self.calc_dR_dW(H, W_1, X_1, A, p_s, n_0) # calculating the gradient
of the rate with respect to W
            W_2 = W_1 + self.Mu_W[j] * dR_dW + self.betas[j] * (W_1 - W_0) # updating W
            # projecting W on the constraints:
            if W_is_phase_only:
                W_2 = project_onto_phases(W_2)
            if W_is_block_diag:
                W_2 = project_onto_block_diagonal(W_2,self.D)
            Rs[j] = self.calc_R(H, W_2, X_1, A, p_s, n_0)  # documenting the rate
            # updating the matrices:
            W_0 = W_1
            W_1 = W_2
        return Rs, W_2, X_1


    def calc_R(self, H, W, X, A, p_s, n_0):
        """gets a channel realization and returns the achievable rate of it. h is of dime
nsion (B,M,K)"""
        R = 0
        B, M, K = H.shape
        G = W @ A @ X
        snr = p_s / n_0
        for b in range(B):
            h1 = H[b, :, :]
            g1 = G
            psudo_inv_g = torch.inverse(g1.transpose(0, 1).conj() @ g1)
            proj_mat = g1 @ psudo_inv_g @ g1.transpose(0, 1).conj()
            R += torch.log((Im + snr * proj_mat @ h1 @ h1.transpose(0, 1).conj()).det()
)
        return R/B


    def calc_max_r(self, H):
        """calclulates the maximal rate of the channel H, susbstituting G to be the match
ed filter. not used in the algorithm"""
        R = 0
        B, M, K = H.shape
        G = H
        snr = p_s / n_0
        for b in range(B):
            h1 = H[b, :, :]
            g1 = G[b, :, :]
            psudo_inv_g = torch.inverse(g1.transpose(0, 1).conj() @ g1)
            proj_mat = g1 @ psudo_inv_g @ g1.transpose(0, 1).conj()
            R += torch.log((Im + snr * proj_mat @ h1 @ h1.transpose(0, 1).conj()).det()
)
        return R/B


    def calc_dR_dW(self, H, W, X, A, p_s, n_0):
        """calculates the gradient of the rate with respect to W. the gradient has the sa
me dimension as W"""
        grad = 0
        for b in range(B):
            h1 = H[b, :, :]
            snr = p_s / n_0
            X_con = X.conj()
            W_con = W.conj()
            H_con = h1.conj()
            G = W @ A @ X
            G_herm = G.transpose(0, 1).conj()
            h1_herm = h1.transpose(0, 1).conj()
            G_psu_inv = torch.inverse(G_herm @ G)
            Proj_mat = G @ G_psu_inv @ G_herm
```

```python
            Z = snr * G_psu_inv @ G_herm @ h1 @ h1_herm @ G
            dR_dZ = torch.inverse(Ik + Z)
            grad += snr * ((A @ X) @ dR_dZ @ (2 * G_psu_inv @ G_herm) @ h1 @ h1_herm @
(Im - Proj_mat)).conj().transpose(0,1)
        return grad / B


    def calc_dR_dX(self, H, W, X, A, p_s, n_0):
        """calculates the gradient of the rate with respect to X. the gradient has the sa
me dimension as X"""
        """ not used in our code! """
        snr = p_s / n_0
        X_con = X.conj()
        W_con = W.conj()
        H_con = H.conj()
        G = W @ A @ X
        G_herm = G.transpose(1, 2).conj()
        H_herm = H.transpose(1, 2).conj()
        G_psu_inv = torch.inverse(G_herm @ G)
        Proj_mat = G @ G_psu_inv @ G_herm
        Z = snr * G_psu_inv @ G_herm @ H @ H_herm @ G
        dR_dZ = torch.inverse(Ik + Z)
        return snr * (dR_dZ @ (2 * G_psu_inv @ G_herm) @ H @ H_herm @ (Im - Proj_mat)
@ W @ A).conj().transpose(1, 2)


def train_unfolding(H_train, H_valid, X, W, A, p_s, n_0, J, Mu_X_init, Mu_W_init,
          optimizer_learning_rate, epochs, train_size, batch_size):
    """ Trains the unfolded model - not the momentum model."""
    """ Returns the trained model and the training and validation rates. Using ADAM optim
izer and maximizing the weighted sum rate function of each iteration. """
    # Setting the variables:
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    H_train, H_valid, X, W, A = H_train.to(device), H_valid.to(device), X.to(device), W
.to(device), A.to(device)
    Mu_W = torch.full((J, M, L * P), Mu_W_init, requires_grad=True, device = device)
    Mu_X = torch.full((J, T, T), Mu_X_init, requires_grad=X_is_variable, device=device)
    # crating the model and setting the optimizer
    unfolded_model = UnfoldedModel(Mu_W, Mu_X, J).to(device)
    optimizer = torch.optim.Adam([Mu_W], lr=optimizer_learning_rate)
    # projecting W on the constraints
    if W_is_phase_only:
        W = project_onto_phases(W)
    if W_is_block_diag:
        W = project_onto_block_diagonal(W,D)
    train_rates = [0]*(epochs*int(np.ceil(train_size/batch_size)))
    validation_rates = [0]*(epochs)
    i = 0
    # Start training:
    print("Unfolded starts training")
    for epoch in range(epochs):
        print("epoch: ", epoch + 1, " of ", epochs)
        initial_time = time.time()
        # Shuffling the training data:
        H_train = H_train[torch.randperm(H_train.shape[0])]
        for iter in range(0, train_size, batch_size):
            # dividing the data into batches:
            H = H_train[iter:iter + batch_size, :, :, :]
            Rs, W_1, X_1 = unfolded_model.batch_rate(H, W=W, X=X, A=A, p_s=p_s, n_0=n_0)
# Rs is the list of the calculated rates in each iteration.
            loss = -  torch.sum(Rs * torch.log(torch.arange(2, J + 2, device=device)))
# Negating the weighted loss since we want to maximize the rate.
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            train_rates[i] = Rs[-1]
            i = i+1
        # At the end of each epoch we calculate the validation rate:
        validation_rate, _, _ = unfolded_model.batch_rate(H_valid, W=W, X=X, A=A,p_s=p_s
,n_0=n_0)
        validation_rates[epoch] = validation_rate
```

```python
        print("epoch time: ", round(time.time() - initial_time, 2), " seconds")
    return unfolded_model, train_rates, validation_rates


def run_PGA_Momentum(PGA_Mu_init, beta, pga_iters, H_valid, W, X, A, p_s, n_0):
    """ Running the PGA + Momentum, with fixes Mu and Beta. Returns the list of calculated sum-rate for each iteration. """
    """ Used as a benchmark comparing the performance of the unfolded algorithm with the classical one. """
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    # projecting W on the constraints
    if W_is_phase_only:
        W = project_onto_phases(W)
    if W_is_block_diag:
        W = project_onto_block_diagonal(W, D)
    # moving the variables to "device":
    H_valid,W,X,A = H_valid.to(device), W.to(device),X.to(device),A.to(device)
    # setting the model and running the PGA:
    pga_model_Momentum = PGAModelMomentum(Mu=PGA_Mu_init,beta=beta, pga_iters=pga_iters)
    Rs_pga, W_2, X_2 = pga_model_Momentum.batch_rate(H_valid, W=W, X=X,A=A, p_s=p_s, n_0=n_0)
    return Rs_pga, W_2, X_2


def run_PGA(PGA_Mu_init, pga_iters, H_valid, W, X, A, p_s, n_0):
    """ Running the classical PGA , with fixes Mu and Beta. Returns the list of calculated sum-rate for each iteration. """
    """ Not used in our code. """
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    if W_is_phase_only:
        W = project_onto_phases(W)
    if W_is_block_diag:
        W = project_onto_block_diagonal(W, D)
    H_valid,W,X,A = H_valid.to(device), W.to(device),X.to(device),A.to(device)
    pga_model = PGAModel(Mu=PGA_Mu_init, pga_iters=pga_iters)
    Rs_pga, W_2, X_2 = pga_model.batch_rate(H_valid, W=W, X=X,A=A, p_s=p_s, n_0=n_0)
    return Rs_pga, W_2, X_2

def line_search(H_valid, X, W, A, p_s, n_0, grid, line_search_iters):
    """ Iterates over every sample in the validation set, and finds the best step size for each sample. """
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    # projecting W on the constraints:
    if W_is_phase_only:
        W = project_onto_phases(W)
    if W_is_block_diag:
        W = project_onto_block_diagonal(W, D)
    H_valid, W, X, A = H_valid.to(device), W.to(device), X.to(device), A.to(device)
    W_1 = W.detach()
    X_1 = X.detach()
    # Initializing the PGA + M object:
    pga_mom_model = PGAModelMomentum(Mu=0,beta=0.9, pga_iters=line_search_iters)
    best_steps = torch.zeros((H_valid.shape[0], line_search_iters), device=device)
    best_Rs = torch.zeros((H_valid.shape[0], line_search_iters), device=device)
    # Takes a single sample:
    for sample_num in range(H_valid.shape[0]):
        W_1 = W.detach()
        X_1 = X.detach()
        sample = H_valid[sample_num, :, :, :]
        # Start moving over all the iterations in "line_search_iters" (Usually 50).
        for iteration in range(line_search_iters):
            if W_is_phase_only:
                W_1 = project_onto_phases(W_1)
            if W_is_block_diag:
                W_1 = project_onto_block_diagonal(W_1, D)
            # Calculates the gradient:
            grad = pga_mom_model.calc_dR_dW(sample, W_1, X_1, A, p_s, n_0)
            best_step = 0
            best_R = -1
            # Looking for the best step size with the calculated gradient:
            for step in grid:
                W_new = W_1 + step * grad  # Updating W
```

```
                if W_is_phase_only:
                    W_new = project_onto_phases(W_new)
                if W_is_block_diag:
                    W_new = project_onto_block_diagonal(W_new, D)
                # Calculating the optional R.
                R = pga_mom_model.calc_R(sample, W_new, X, A, p_s, n_0)
                if abs(R) > abs(best_R): # If the calculated R is better than the best R so far, save the step size.
                    best_R = R
                    best_step = step
            # Before the next iteration, update W with the best step chosen.
            W_1 += best_step * grad
            if W_is_phase_only:
                W_1 = project_onto_phases(W_1)
            if W_is_block_diag:
                W_1 = project_onto_block_diagonal(W_1,D)
            best_steps[sample_num,iteration] = best_step # saving the best step and the best R.
            best_Rs[sample_num,iteration] = best_R
    average_best_Rs = torch.mean(best_Rs,axis=0)  # Calculate the mean R
    average_best_steps = torch.mean(best_steps,axis=0)  # Calculate the mean best step size, for comparison reasons.
    return average_best_Rs, average_best_steps


def train_momentum(H_train, H_valid, X, W, A,p_s,n_0, J, Mu_X_init, Mu_W_init,
          optimizer_learning_rate, epochs, train_size, batch_size, betas_init):
    """ Trains the unfolded model + Momentum. Returns the trained model and the training and validation rates.
     Using ADAM optimizer and maximizing the weighted sum rate function of each iteration."""
    # setting the device to be GPU to accalerate calculations.
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    if W_is_phase_only:
        W = project_onto_phases(W)
    if W_is_block_diag:
        W = project_onto_block_diagonal(W,D)
    # saving variables in device:
    H_train, H_valid, X, W, A = H_train.to(device), H_valid.to(device), X.to(device), W.to(device), A.to(device)
    Mu_W = torch.full((J, M, L * P), Mu_W_init, requires_grad=True, device=device)
    Mu_X = torch.full((J, T, T), Mu_X_init, requires_grad=X_is_variable, device=device)
    betas = torch.full((J, M, L * P), betas_init, requires_grad=True, device = device)
    momentum_model = MomentumModel(Mu_W, Mu_X,betas, J).to(device)
    optimizer = torch.optim.Adam([Mu_W, betas], lr=optimizer_learning_rate)
    if W_is_phase_only:
        W = project_onto_phases(W)
    if W_is_block_diag:
        W = project_onto_block_diagonal(W,D)
    train_rates = [0]*(epochs*int(np.ceil(train_size/batch_size)))
    validation_rates = [0]*(epochs)
    i = 0
    # Starting training procedure:
    print("Momentum starts training")
    for epoch in range(epochs):
        print("epoch: ", epoch + 1, " of ", epochs)
        initial_time = time.time()
        H_train = H_train[torch.randperm(H_train.shape[0])]  # Shuffles the training data.
        for iter in range(0, train_size, batch_size):
            H = H_train[iter:iter + batch_size, :, :, :]  # dividing the data to batches.
            Rs, W_1, X_1 = momentum_model.batch_rate(H, W=W, X=X, A=A, p_s=p_s, n_0=n_0) # Rs is the list of the calculated rates in each iteration.
            loss = -  torch.sum(Rs * torch.log(torch.arange(2, J + 2, device=device))) # Negating the weighted loss since we want to maximize the rate.
            optimizer.zero_grad()
            loss.backward()  # Making the backward step.
            optimizer.step()
            train_rates[i] = Rs[-1]
            i = i+1
        # At the end of each epoch we calculate the validation rate:
```

```
        validation_rate, _, _ = momentum_model.batch_rate(H_valid, W=W, X=X, A=A,p_s=p_s
,n_0=n_0)
        validation_rates[epoch] = validation_rate
        print("epoch time: ", round(time.time() - initial_time, 2), " seconds")
    return momentum_model, train_rates, validation_rates


def constant_search(H_valid, W, X, A, p_s, n_0,constant_search_iters, grid):
    """ A function that finds the best Mu from a given grid, for a classical PGA algorith
m without momentum. Not used in our code. """
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    H_valid, W, X, A = H_valid.to(device), W.to(device), X.to(device), A.to(device)
    W_1 = W.detach()
    X_1 = X.detach()
    if W_is_phase_only:
        W_1 = project_onto_phases(W_1)
    if W_is_block_diag:
        W_1 = project_onto_block_diagonal(W_1, D)
    best_R = 0
    best_step = 0
    for step in grid:
        pga_model = PGAModel(Mu=step, pga_iters=constant_search_iters)
        R,_,_ = pga_model.batch_rate(H_valid, W_1, X_1, A, p_s, n_0)
        if abs(R[-1]) > abs(best_R):
            best_R = R[-1]
            best_step = step
    return best_step, best_R


def beta_search(H_valid, W, X, A, p_s, n_0,constant_search_iters, grid, best_Mu):
    """ A function that finds the best beta from a given grid, for a PGA + M , with a fi
xed Mu. Not used in our code. """
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    H_valid, W, X, A = H_valid.to(device), W.to(device), X.to(device), A.to(device)
    W_1 = W.detach()
    X_1 = X.detach()
    if W_is_phase_only:
        W_1 = project_onto_phases(W_1)
    if W_is_block_diag:
        W_1 = project_onto_block_diagonal(W_1, D)
    best_R = 0
    best_step = 0
    for step in grid:
        pga_model_mom = PGAModelMomentum(Mu=best_Mu,beta=step, pga_iters=constant_search
_iters)
        R,_,_ = pga_model_mom.batch_rate(H_valid, W_1, X_1, A, p_s, n_0)
        if abs(R[-1]) > abs(best_R):
            best_R = R[-1]
            best_step = step
    return best_step, best_R


def mu_and_beta_search(H_valid, W, X, A, p_s, n_0,constant_search_iters, grid_mu, grid_b
eta):
    """ A function that finds the best Mu and beta combination from some given grids, for
a classical PGA + M.
     Used in order to find the optimal values of PGA + M as benchmarks.
     The unfolded model is initiated with the values found here. """

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    H_valid, W, X, A = H_valid.to(device), W.to(device), X.to(device), A.to(device)
    W_1 = W.detach()
    X_1 = X.detach()
    if W_is_phase_only:
        W_1 = project_onto_phases(W_1)
    if W_is_block_diag:
        W_1 = project_onto_block_diagonal(W_1, D)
    best_R = 0
    best_mu = 0
    best_beta = 0
    pga_model_mom = PGAModelMomentum(Mu=0, beta=0, pga_iters=constant_search_iters)
    for mu in grid_mu:
```

```
        pga_model_mom.Mu = mu  # Changing the model hyperparameter
        for beta in grid_beta:
            pga_model_mom.beta = beta  # Changing the model hyperparameter
            print("mu:",round(mu.item(),2),"beta:",round(beta.item(),2))
            # for a given combination of Mu and Beta, we run the PGA + M algorithm to fin
d what combination brings the best R.
            R,_,_ = pga_model_mom.batch_rate(H_valid, W_1, X_1, A, p_s, n_0)
            if abs(R[-1]) > abs(best_R):  # Checking if the last R is better than the be
st R:
                best_R = R[-1]
                best_mu = mu
                best_beta = beta
    return best_mu,best_beta, best_R


def plot_learning_curve(train_rates, validation_rates):
    """ Plotting the improvment of the sum-rate after training the model for each epoch.
Helps us avoid overfitting, tune the learning rate, etc.
     Used for internal analysis. """
    iters_per_epoch = np.ceil(train_size / batch_size)
    plt.figure()
    y_t = [r.cpu().detach().numpy() for r in train_rates]
    x_t = np.array(list(range(len(train_rates)))) / iters_per_epoch
    y_v = [r[-1].cpu().detach().numpy() for r in validation_rates]
    x_v = np.array(list(range(len(validation_rates)))) + 1
    plt.plot(x_t, y_t, 'o', label='Train')
    plt.plot(x_v, y_v, '*', label='Valid')
    plt.grid()
    title = 'Channel Rate After ' + str(J) + ' Iterations Unfolded PGA, After training '
+ str(epochs) + ' Epochs \n ' + \
            W_is_block_diag * 'W is Block Diagonal' + 'W is not Block Diagonal' * (not W
_is_block_diag) + \
            ', W is Only Phases' * W_is_phase_only + ', W is not Only Phases' * (not W_i
s_phase_only) + \
            ', X is variable' * X_is_variable + ', X is fixed' * (not X_is_variable)
    plt.title(title)
    plt.xlabel('Epoch')
    plt.ylabel("Bits per channel use")
    plt.legend()
    plt.show()
    return


def plot_comparison(J, validation_rate, pga_iters, Rs_pga, labels = ['Unfolded PGA + M',
'PGA + M']):
    """ Plotting the calculted rate after each iteration, only for the first J iterations
(usually, J=10)."""
    plt.figure()
    plt.plot(list(range(1,J+1)) , validation_rate.detach().cpu().numpy(), label=labels[0
])
    plt.plot(list(range(1,pga_iters+1)) , Rs_pga.detach().cpu().numpy(), label=labels[1]
)
    plt.grid()
    plt.title('Channel Rate After ' + str(J) +
              ' Iterations Unfolded PGA, After training ' + str(epochs) + ' Epochs \n' +
              'with K = ' + str(K))
    plt.xlabel('Iteration')
    plt.ylabel("Bits per channel use per user")
    plt.axhline(y=max(abs(Rs_pga)).detach().cpu().numpy(), color='black', linestyle='--'
, label='Max rate after {} iters'.format(pga_iters))
    plt.legend()
    plt.xlim(1,J)
    # plt.show()
    return


def plot_comparison_full(J, validation_rate, pga_iters, Rs_pga, labels = ['Unfolded PGA
+ M','PGA + M']):
    """ Plotting the calculted rate after each iteration, For the whole range of iteratio
ns (usually, 400)."""
    plt.figure()
    plt.plot(list(range(1,J+1)) , validation_rate.detach().cpu().numpy(), label=labels[0
```

```
])
    plt.plot(list(range(1,pga_iters+1)) , Rs_pga.detach().cpu().numpy(), label=labels[1]
)
    plt.grid()
    plt.title('Channel Rate After ' + str(J) +
              ' Iterations Unfolded PGA, After training ' + str(epochs) + ' Epochs \n' +
              'with K = ' + str(K))
    plt.xlabel('Iteration')
    plt.ylabel("Bits per channel use per user")
    plt.axhline(y=max(abs(Rs_pga)).detach().cpu().numpy(), color='black', linestyle='--'
, label='Max rate after {} iters'.format(pga_iters))
    plt.legend()
    # plt.show()
    return


def plot_Mu_W(Mu_W, J):
    """ Plotting the Mu_W matrix, the step size in each iteration.
     The functions is used for internal analysis. """
    fig1, axs1 = plt.subplots(1, J, figsize=(18, 4.5))
    # Plotting each slice of Mu_W with the same colorbar range :
    global_min = np.min(np.real(Mu_W[:, :, :].cpu().detach().numpy()))
    global_max = np.max(np.real(Mu_W[:, :, :].cpu().detach().numpy()))
    for j in range(J):
        im1 = axs1[j].imshow(np.real(Mu_W[j, :, :].cpu().detach().numpy()), vmin=global_
min, vmax=global_max)
        axs1[j].set_title('j=' + str(j), fontsize=14)
    # Addding a global colorbar for all the plots:
    fig1.colorbar(im1, ax=axs1[:], location='bottom')
    plt.suptitle('Mu_W', fontsize=28)
    plt.show()
    return


def plot_betas(betas, J):
    """ Plotting the Betas matrix, used for the momentum step.
     The function is used for internal analysis. """
    fig1, axs1 = plt.subplots(1, J, figsize=(18, 4.5))
    global_min = np.min(np.real(betas[:, :, :].cpu().detach().numpy()))
    global_max = np.max(np.real(betas[:, :, :].cpu().detach().numpy()))
    # Plot each slice of Mu_W with the same colorbar range
    for j in range(J):
        im1 = axs1[j].imshow(np.real(betas[j, :, :].cpu().detach().numpy()), vmin=global
_min, vmax=global_max)
        axs1[j].set_title('j=' + str(j), fontsize=14)
    # Add a global colorbar for all the plots
    fig1.colorbar(im1, ax=axs1[:], location='bottom')
    plt.suptitle('Momentum Coeffcients', fontsize=28)
    plt.show()
    return


def plot_all(train_rates, validation_rates, Rs_pga, Mu_W, J, pga_iters, to_plot_betas =
False, betas = None):
    """ Plotting all the relevant graphs for the unfolded PGA algorithm. """
    plot_learning_curve(train_rates, validation_rates)
    plot_comparison(J,validation_rates[-1], pga_iters, Rs_pga)
    plot_comparison_full(J,validation_rates[-1], pga_iters, Rs_pga)
    plot_Mu_W(Mu_W, J)
    if to_plot_betas:
        plot_betas(betas, J)
    return
```

Simulations

valid_size=100, pga_iters=100 - I'm here

In [ ]:

```
range_snr = range(0, 6) # SNR range to iterate over.
line_search_max = [0 for _ in range_snr]
pga_400_max = [0 for _ in range_snr]
pga_10_max = [0 for _ in range_snr]
unfolded_max = [0 for _ in range_snr]
momentum_max = [0 for _ in range_snr]
for SNR in range_snr:

    init_time = time.time()
    print("SNR =",SNR)

    # Setting parameters and variables:
    device, n_0, p_s, B, K, P, N, M, L, T, snr, T_opt, A, Ik, Im,D, J,\
     epochs, W_is_block_diag, W_is_phase_only, X_is_variable,\
        train_size, batch_size, pga_iters,\
         optimizer_learning_rate= set_general_params(J=10, epochs=5,
            batch_size=20,T=5,K=6,P=2,L=4,N=8,B=4,train_size = 2000,
              pga_iters = 400, optimizer_learning_rate=0.09, n_0 = 10**(-0.1*SNR))

    H_train,H_valid,X,W = gen_data(seed = 1, train_size = train_size, valid_size = 100)
    H_train,H_valid,X,W = H_train.to(device),H_valid.to(device),X.to(device),W.to(device
)
    grid_line = torch.cat((torch.linspace(0.1, 1, 3),torch.linspace(1.5, 9.5, 17))).to(d
evice)
    grid_mu =  torch.cat((torch.linspace(0.05, 1, 4),torch.linspace(1.5, 7.5, 4))).to(de
vice)
    grid_beta = torch.linspace(0.9, 0.99, 3).to(device)

    # Running benchmarks:
    # Linesearch:
    Rs_line_search, _ = line_search(H_valid, X, W, A, p_s, n_0, grid_line, line_search_i
ters = 50)

    # Classical PGA + M :
    best_mu, best_beta, best_R_mom = mu_and_beta_search(H_valid, W, X, A, p_s, n_0, cons
tant_search_iters=pga_iters,
                                                        grid_mu=grid_mu, grid_beta=grid
_beta)
    # result after 400 iterations:
    Rs_pga_400, _,_ = run_PGA_Momentum(PGA_Mu_init=best_mu, beta=best_beta, pga_iters=pg
a_iters,H_valid = H_valid,W=W,X=X,A=A, p_s= p_s, n_0=n_0)
    # result after 10 iterations:
    Rs_pga_10,_,_ = run_PGA_Momentum(PGA_Mu_init=best_mu,beta=best_beta, pga_iters=J,H_v
alid = H_valid,W=W,X=X,A=A, p_s= p_s, n_0=n_0)

    # Running and training the unfolded model:
    momentum_model, mom_train_rates,mom_validation_rates = train_momentum(H_train,H_vali
d, X,
                                                      W, A, p_s=p_s, n_0=n_0, J=J,Mu_X_
init=1.5, Mu_W_init=best_mu,
                                                      optimizer_learning_rate=optimizer
_learning_rate,
                                                      epochs=epochs,train_size = train_
size,
                                                      batch_size = batch_size, betas_in
it = best_beta)

    # printing the results when SNR = 0[dB] for sanity check.
    line_search_max[SNR] = Rs_line_search[-1]
    pga_400_max[SNR] = Rs_pga_400[-1]
    pga_10_max[SNR] = Rs_pga_10[-1]
    momentum_max[SNR] = mom_validation_rates[-1][-1]
    if SNR==0:
        plot_learning_curve(mom_train_rates, mom_validation_rates)
        plot_comparison(J, mom_validation_rates[-1], pga_iters, Rs_pga_400)
        plt.plot(range(1,51), Rs_line_search.cpu().detach().numpy(), label='Line Search'
)
        plt.xlim(1,J)
        plt.legend()
        plt.show()
        plot_comparison_full(J, mom_validation_rates[-1], pga_iters, Rs_pga_400)
        plt.plot(range(1,51), Rs_line_search.cpu().detach().numpy(), label='Line Search'
```

```
)
        plt.legend()
        plt.show()
        print("momentum:", mom_validation_rates[-1])
        print("line search:", Rs_line_search)
        print("pga 400:", Rs_pga_400)
    print("SNR time:", round(time.time()-init_time,2),"seconds")


# printing and plotting the other results:
plt.figure()
plt.plot(range_snr,torch.tensor(pga_10_max).cpu(), '--*',color="purple",label= 'PGA + M
- 10 Iters')
plt.plot(range_snr,torch.tensor(line_search_max).cpu(), 'g:x',label= 'Line Search - 50 It
ers', linewidth=2.5)
plt.plot(range_snr,torch.tensor(pga_400_max).cpu(),'--*' ,color="orange",label= 'PGA + M
- 400 Iters')
plt.plot(range_snr,torch.tensor(momentum_max).cpu(), 'b--s',label= 'Unfolded PGA + M - 10
Iters')
plt.grid()
plt.legend()
plt.xlabel('SNR [dB]')
plt.ylabel("Bits per channel use")
plt.show()
print("line search max: ", line_search_max)
print("pga 400 max: ", pga_400_max)
print("pga 10 max: ", pga_10_max)
print("momentum max: ", momentum_max)
pass
```

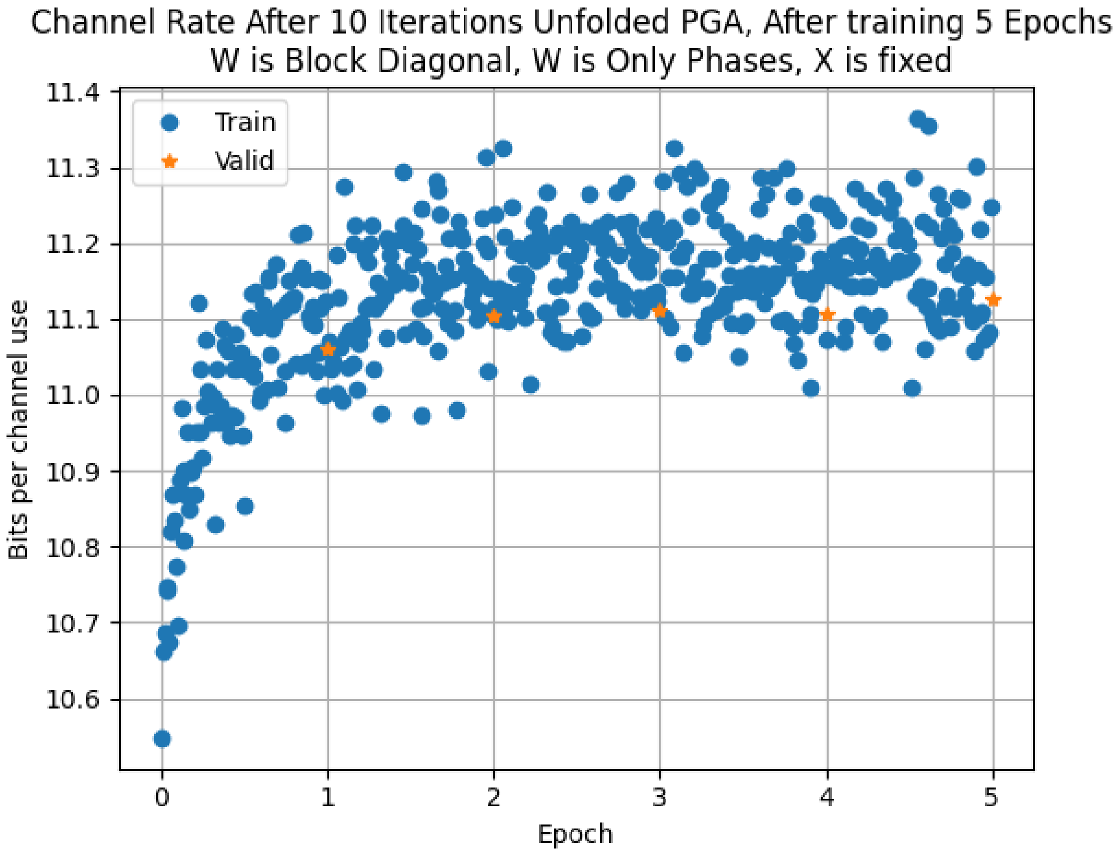
```
SNR = 0
```

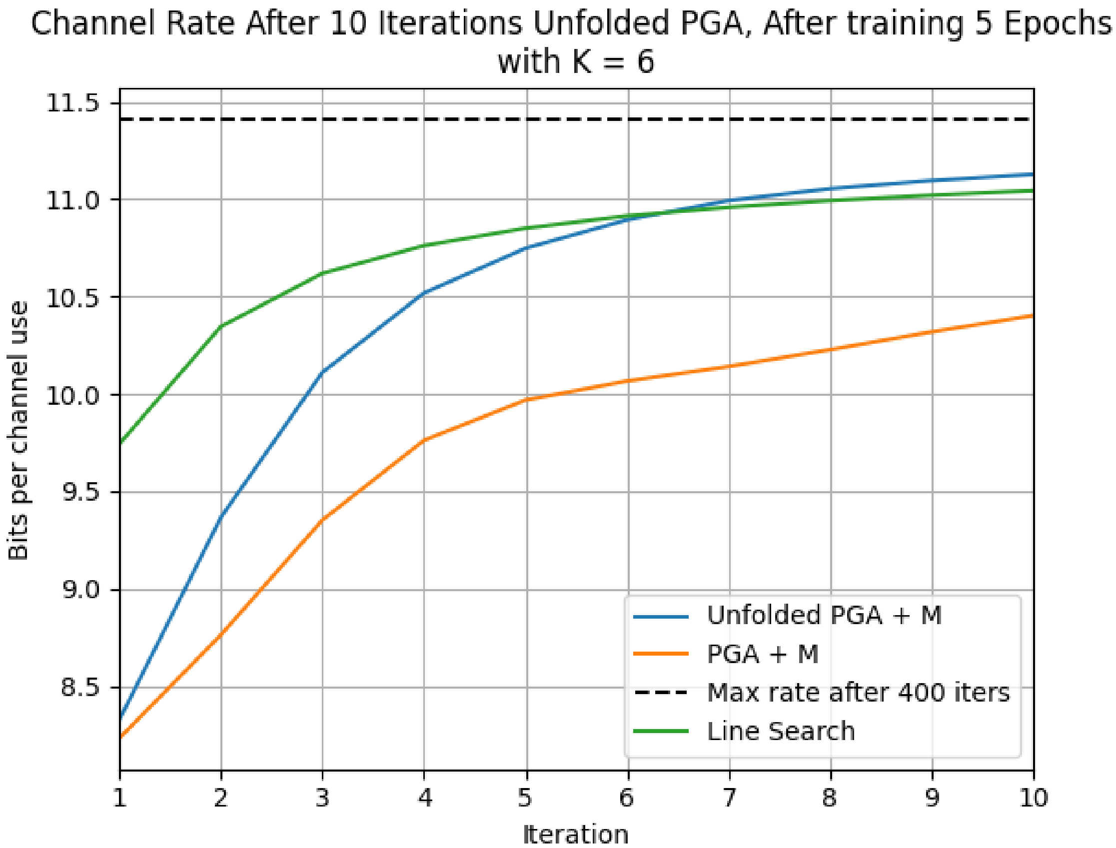
```
<ipython-input-1-bc89971480a9>:620: UserWarning: Casting complex values to real discards
the imaginary part (Triggered internally at ../aten/src/ATen/native/Copy.cpp:276.)
  best_Rs[sample_num,iteration] = best_R
```

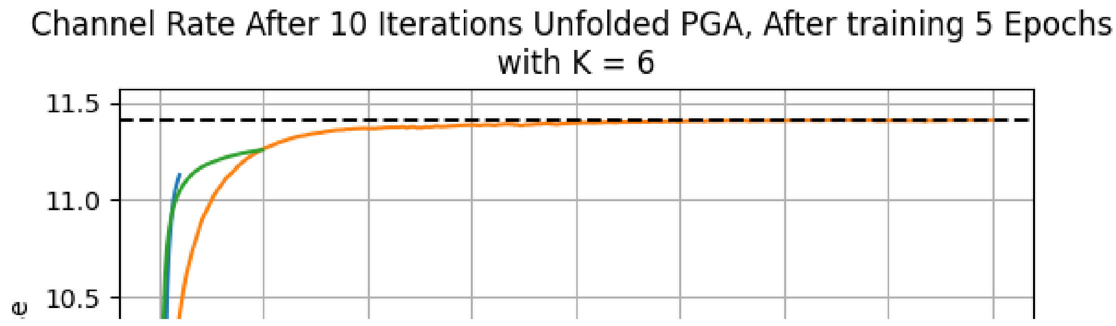
```
mu: 0.05 beta: 0.9
mu: 0.05 beta: 0.94
mu: 0.05 beta: 0.99
mu: 0.37 beta: 0.9
mu: 0.37 beta: 0.94
mu: 0.37 beta: 0.99
mu: 0.68 beta: 0.9
mu: 0.68 beta: 0.94
mu: 0.68 beta: 0.99
mu: 1.0 beta: 0.9
mu: 1.0 beta: 0.94
mu: 1.0 beta: 0.99
mu: 1.5 beta: 0.9
mu: 1.5 beta: 0.94
mu: 1.5 beta: 0.99
mu: 3.5 beta: 0.9
mu: 3.5 beta: 0.94
mu: 3.5 beta: 0.99
mu: 5.5 beta: 0.9
mu: 5.5 beta: 0.94
mu: 5.5 beta: 0.99
mu: 7.5 beta: 0.9
mu: 7.5 beta: 0.94
mu: 7.5 beta: 0.99
Momentum starts training
epoch:  1  of  5
epoch time:  144.08  seconds
epoch:  2  of  5
epoch time:  142.93  seconds
epoch:  3  of  5
epoch time:  142.47  seconds
epoch:  4  of  5
epoch time:  142.7  seconds
epoch:  5  of  5
epoch time:  142.4  seconds
```

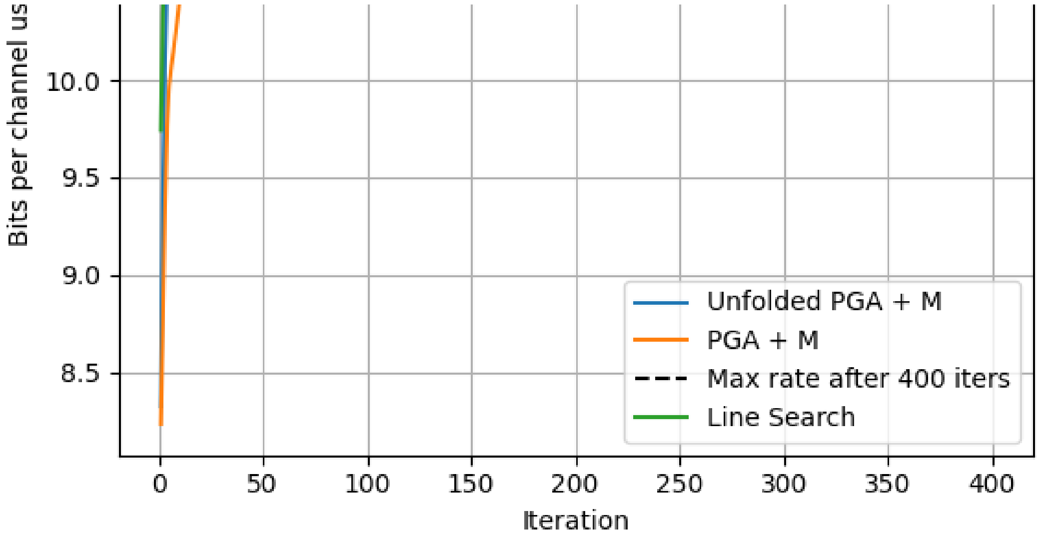
```
/usr/local/lib/python3.10/dist-packages/matplotlib/cbook/__init__.py:1335: ComplexWarning
```

```
: Casting complex values to real discards the imaginary part
  return np.asarray(x, float)
```










```
momentum: tensor([ 8.3266-2.6979e-08j,  9.3660-9.8024e-09j, 10.1106-2.0894e-08j,
        10.5180+2.5131e-08j, 10.7475-6.9064e-09j, 10.8934+2.4132e-08j,
        10.9923+1.0901e-08j, 11.0531+1.7860e-08j, 11.0950-4.3563e-09j,
        11.1271+5.4341e-10j], grad_fn=<DivBackward0>)
line search: tensor([ 9.7427, 10.3456, 10.6186, 10.7609, 10.8506, 10.9134, 10.9579, 10.99
20,
        11.0197, 11.0429, 11.0629, 11.0806, 11.0960, 11.1095, 11.1212, 11.1315,
        11.1409, 11.1493, 11.1568, 11.1638, 11.1701, 11.1758, 11.1809, 11.1855,
        11.1899, 11.1942, 11.1984, 11.2025, 11.2063, 11.2102, 11.2138, 11.2175,
        11.2206, 11.2237, 11.2265, 11.2292, 11.2318, 11.2344, 11.2368, 11.2390,
        11.2411, 11.2431, 11.2451, 11.2471, 11.2488, 11.2505, 11.2521, 11.2539,
        11.2555, 11.2570])
pga 400: tensor([ 8.2330+7.4808e-10j,  8.7630+4.0188e-09j,  9.3528+1.4483e-08j,
         9.7618-4.7346e-09j,  9.9686-1.9035e-08j, 10.0660+1.2387e-08j,
        10.1409+6.0179e-10j, 10.2270-4.6541e-09j, 10.3184+2.1371e-08j,
        10.4021-6.1937e-09j, 10.4765-1.7689e-08j, 10.5433+7.2204e-09j,
        10.6011+4.9058e-10j, 10.6501-1.6529e-08j, 10.6936+7.0838e-09j,
        10.7345+7.0859e-09j, 10.7682-7.9727e-09j, 10.7991-2.8507e-09j,
        10.8355-4.1661e-09j, 10.8740+9.7041e-09j, 10.9036-1.1254e-08j,
        10.9247+6.3165e-10j, 10.9448+3.2165e-09j, 10.9671+1.1326e-08j,
        10.9905+3.9685e-09j, 11.0124-2.4373e-08j, 11.0317-1.6450e-08j,
        11.0478-1.6634e-09j, 11.0608+8.5920e-09j, 11.0749-1.2055e-08j,
        11.0915-6.9043e-10j, 11.1071+5.1050e-09j, 11.1185-4.7759e-09j,
        11.1275+7.6407e-09j, 11.1369-1.9909e-09j, 11.1479+2.7014e-08j,
        11.1606+1.7413e-08j, 11.1717-1.3654e-08j, 11.1807+2.0679e-08j,
        11.1892-1.3956e-08j, 11.1992-1.7079e-09j, 11.2092+4.8455e-09j,
        11.2167-1.5931e-08j, 11.2240+4.7202e-08j, 11.2307+2.9465e-08j,
        11.2363-7.8776e-09j, 11.2432+4.5074e-09j, 11.2503+2.1511e-08j,
        11.2567+4.7867e-09j, 11.2619+2.7687e-08j, 11.2657-3.6164e-08j,
        11.2699+1.0609e-08j, 11.2740-2.7771e-08j, 11.2780+5.2599e-09j,
        11.2831-2.4946e-08j, 11.2882-3.2816e-09j, 11.2926+1.4908e-08j,
        11.2959+1.1196e-08j, 11.2992-1.2545e-08j, 11.3026+3.1824e-08j,
        11.3063-5.7723e-09j, 11.3101-1.0626e-08j, 11.3136+8.9568e-09j,
        11.3169+1.3967e-08j, 11.3196-2.6137e-08j, 11.3227-6.0956e-08j,
        11.3255+1.1221e-08j, 11.3273-4.5776e-08j, 11.3299+2.1442e-08j,
        11.3312+4.7249e-08j, 11.3322+1.3685e-08j, 11.3360-2.2586e-08j,
        11.3377+2.0637e-08j, 11.3400+1.7580e-08j, 11.3417-2.0773e-08j,
        11.3426-2.1732e-08j, 11.3434+3.5996e-08j, 11.3450+1.3509e-08j,
        11.3478-2.2874e-08j, 11.3502-2.4867e-08j, 11.3510-1.6803e-08j,
        11.3529+3.3044e-08j, 11.3547+1.1140e-08j, 11.3562-6.4781e-09j,
        11.3579+1.7201e-08j, 11.3584-3.6480e-09j, 11.3586-1.5291e-08j,
        11.3595+3.3181e-08j, 11.3597+2.1212e-08j, 11.3626-2.2705e-08j,
        11.3624-6.8769e-08j, 11.3621-2.5868e-08j, 11.3642+7.2826e-08j,
        11.3648-5.9478e-08j, 11.3647+2.7890e-08j, 11.3657-2.7643e-09j,
        11.3657-2.3270e-08j, 11.3666-8.9664e-08j, 11.3676-1.6122e-08j,
        11.3661+5.5981e-08j, 11.3646+6.6495e-08j, 11.3666+3.1922e-08j,
        11.3666-1.3108e-08j, 11.3651+3.5327e-08j, 11.3680+5.2306e-08j,
        11.3685-2.6299e-08j, 11.3689-1.2626e-08j, 11.3706-1.0673e-08j,
        11.3718+3.8841e-08j, 11.3714-2.6493e-08j, 11.3726+1.8138e-08j,
        11.3732-4.5045e-08j, 11.3721-2.8241e-08j, 11.3734+2.6611e-08j,
        11.3721-1.3176e-09j, 11.3742+5.2139e-08j, 11.3750+3.6254e-08j,
        11.3740+1.0653e-08j, 11.3690+5.9547e-08j, 11.3727+8.3786e-08j,
        11.3746-2.5214e-09j, 11.3739-6.4638e-10j, 11.3740-6.0140e-08j,
```

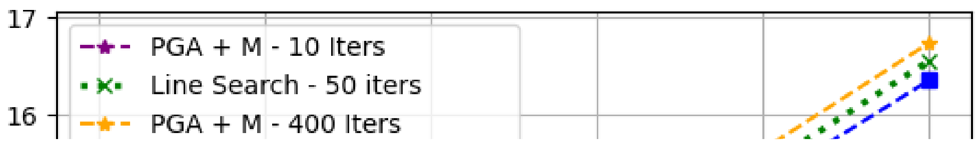
```
11.3691-5.3850e-08j, 11.3714+2.5295e-08j, 11.3696-5.1135e-09j,
11.3720+2.9419e-08j, 11.3748+7.7787e-09j, 11.3740-9.8411e-08j,
11.3732+8.8674e-09j, 11.3749-2.5194e-08j, 11.3755-4.6589e-08j,
11.3786-7.9169e-08j, 11.3783+5.9290e-08j, 11.3761-4.4705e-09j,
11.3759-3.9838e-09j, 11.3768+2.2216e-08j, 11.3786-2.5881e-08j,
11.3789-5.7299e-08j, 11.3797+9.6253e-09j, 11.3803+4.2548e-08j,
11.3801+2.2779e-09j, 11.3815+2.2702e-08j, 11.3829-1.4781e-08j,
11.3831-3.5276e-08j, 11.3834+7.7041e-09j, 11.3839+7.3088e-09j,
11.3844+1.5742e-08j, 11.3846-2.7641e-08j, 11.3848+2.6546e-08j,
11.3853-4.5493e-08j, 11.3843+2.2065e-09j, 11.3833+5.0383e-09j,
11.3822+2.9004e-08j, 11.3841+3.2706e-08j, 11.3853-4.4510e-09j,
11.3857-1.5350e-08j, 11.3860-6.0973e-10j, 11.3857+6.1709e-08j,
11.3837-8.1828e-08j, 11.3838-2.0070e-08j, 11.3830+6.9352e-09j,
11.3870-5.1759e-08j, 11.3866+3.8803e-09j, 11.3878+7.1080e-08j,
11.3892+2.9953e-08j, 11.3888-7.2881e-08j, 11.3895-4.3692e-08j,
11.3879+5.0882e-08j, 11.3892-6.6517e-09j, 11.3844+1.7076e-08j,
11.3871+5.4023e-09j, 11.3831-6.1001e-09j, 11.3809+1.2994e-08j,
11.3830+1.1980e-08j, 11.3866+7.4059e-09j, 11.3855-8.7410e-10j,
11.3863+4.8426e-08j, 11.3861+2.9197e-08j, 11.3881-8.1085e-08j,
11.3887+5.1276e-08j, 11.3906-6.4794e-08j, 11.3887+4.6755e-08j,
11.3909+1.4886e-08j, 11.3918+3.9137e-08j, 11.3928+4.8634e-08j,
11.3921-6.6594e-08j, 11.3918-1.5230e-09j, 11.3910+2.7995e-08j,
11.3891+1.9625e-08j, 11.3869-5.9306e-08j, 11.3891+2.5135e-08j,
11.3899-9.5199e-08j, 11.3919+3.3017e-08j, 11.3915+5.2611e-09j,
11.3925+3.0243e-08j, 11.3930+7.3259e-09j, 11.3932+1.6518e-08j,
11.3936-6.7076e-08j, 11.3935-8.3171e-09j, 11.3939+2.2535e-08j,
11.3949-2.6800e-08j, 11.3965-1.1020e-08j, 11.3960+5.3503e-08j,
11.3959+4.0293e-08j, 11.3965-3.1495e-08j, 11.3970+4.4443e-08j,
11.3957-1.6395e-08j, 11.3955+2.2489e-08j, 11.3965-2.2001e-08j,
11.3972+2.8446e-09j, 11.3969-1.0111e-08j, 11.3976+3.6605e-08j,
11.3984-1.2733e-08j, 11.3989+1.7555e-08j, 11.3994+1.2603e-08j,
11.3987+1.1105e-07j, 11.3991-2.9546e-08j, 11.3972+1.8277e-08j,
11.3984+5.0799e-08j, 11.3997+1.7708e-08j, 11.4003+4.1388e-08j,
11.4001-5.6588e-08j, 11.3994-4.3681e-08j, 11.4001+4.7783e-08j,
11.4005+2.1284e-08j, 11.4007+3.3310e-08j, 11.4016-3.2996e-08j,
11.4012-1.0957e-09j, 11.4009+5.0347e-08j, 11.4003+7.0690e-10j,
11.4016+7.9654e-09j, 11.4019-5.1969e-08j, 11.4014+5.9489e-08j,
11.4008+7.7829e-09j, 11.4026-2.3076e-08j, 11.4031+5.7499e-09j,
11.4034+2.3465e-08j, 11.4029-3.5883e-08j, 11.4017-7.4931e-08j,
11.4021+2.3953e-08j, 11.4033-2.2583e-08j, 11.4031-8.6778e-08j,
11.4039-3.7639e-08j, 11.4029+1.4625e-08j, 11.4038+3.0180e-08j,
11.4048-4.8549e-08j, 11.4043-3.6638e-08j, 11.4024+2.1234e-08j,
11.4022+6.6338e-08j, 11.4042-4.1669e-08j, 11.4038-4.1848e-08j,
11.4049+2.3863e-08j, 11.4054+7.6533e-09j, 11.4051-2.0781e-08j,
11.4056-2.0645e-08j, 11.4061-4.1941e-08j, 11.4060-3.3144e-08j,
11.4055+2.6189e-08j, 11.4067-1.8086e-08j, 11.4062-1.3621e-08j,
11.4035-4.1822e-08j, 11.4064-6.1693e-09j, 11.4061+2.7255e-08j,
11.4072+3.6597e-08j, 11.4064-3.8838e-08j, 11.4064+4.5586e-08j,
11.4069-6.8744e-09j, 11.4075+6.0846e-08j, 11.4074-2.9806e-08j,
11.4072+1.7677e-08j, 11.4082+4.9825e-08j, 11.4081+8.8889e-10j,
11.4081-8.6930e-08j, 11.4072-9.5345e-09j, 11.4083+1.1326e-07j,
11.4072+2.2247e-08j, 11.4081-1.0554e-08j, 11.4086+2.9749e-09j,
11.4079+1.2478e-08j, 11.4083+4.7584e-08j, 11.4080-4.1809e-08j,
11.4065+2.6319e-08j, 11.4064+2.0861e-08j, 11.4075-2.2417e-08j,
11.4070-5.4901e-08j, 11.4069+3.6400e-08j, 11.4076+1.1108e-08j,
11.4074+1.6349e-08j, 11.4081-1.5115e-08j, 11.4077+2.3955e-08j,
11.4079+8.1178e-08j, 11.4085+2.8419e-08j, 11.4081+4.4581e-09j,
11.4079-6.7668e-08j, 11.4081+2.8868e-08j, 11.4087+5.6387e-08j,
11.4084-2.7065e-08j, 11.4079-6.9286e-08j, 11.4084-2.0818e-09j,
11.4088-1.6898e-08j, 11.4090-2.2973e-08j, 11.4088-7.1949e-09j,
11.4090+1.4625e-08j, 11.4086+6.5968e-08j, 11.4094+6.9505e-09j,
11.4088+3.0470e-08j, 11.4095-3.7439e-08j, 11.4095-5.7576e-08j,
11.4094+2.3832e-08j, 11.4096+4.8639e-08j, 11.4095-7.1273e-08j,
11.4093+8.5615e-08j, 11.4104+4.4906e-08j, 11.4089+3.5821e-08j,
11.4102-2.9974e-08j, 11.4086-1.2984e-08j, 11.4093-3.8409e-10j,
11.4092+1.9746e-08j, 11.4095-1.5644e-08j, 11.4102-5.6027e-08j,
11.4101+1.9828e-08j, 11.4092-3.2214e-08j, 11.4104+2.0609e-08j,
11.4101+5.0591e-08j, 11.4101+2.5383e-08j, 11.4113-8.9764e-09j,
11.4105-3.3611e-08j, 11.4095-2.9890e-10j, 11.4104-4.4181e-09j,
11.4102+2.6639e-08j, 11.4093-1.5462e-10j, 11.4103-2.4658e-08j,
11.4104-2.4307e-08j, 11.4105-7.9842e-08j, 11.4093+3.0933e-08j,
11.4097+7.6436e-09j, 11.4095+2.7539e-08j, 11.4092+5.8807e-08j,
```

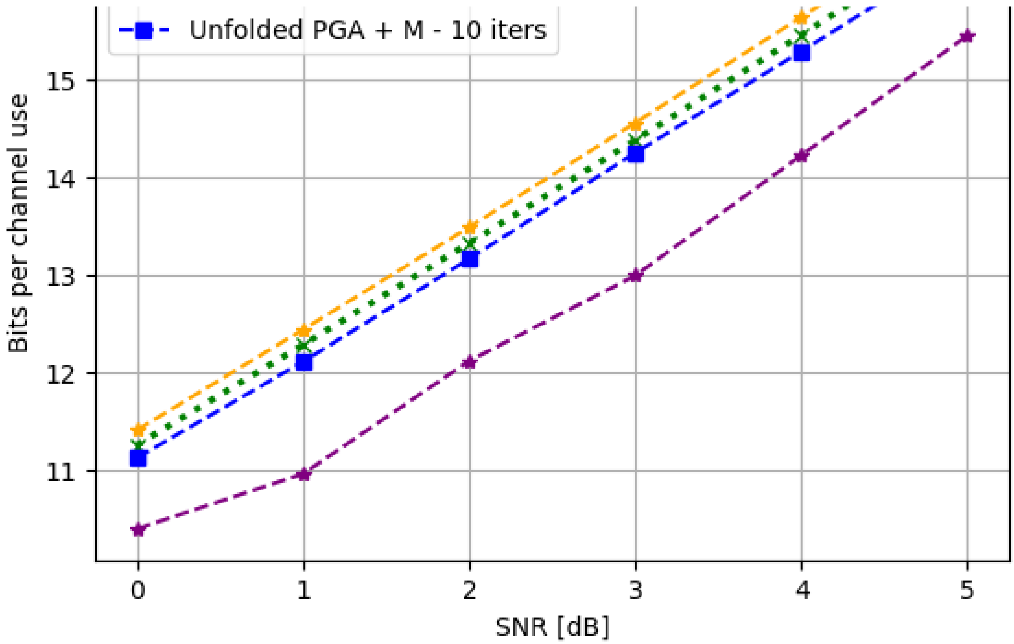
```
        11.4102-3.4362e-08j, 11.4099+7.3334e-08j, 11.4094-5.2128e-08j,
        11.4087+3.6199e-08j, 11.4094+1.6839e-08j, 11.4066+8.5635e-08j,
        11.4075+1.6142e-08j, 11.4080+5.3141e-09j, 11.4084-4.7823e-08j,
        11.4074+1.7334e-08j, 11.4085+7.1214e-09j, 11.4080+8.5211e-09j,
        11.4099-4.2191e-08j, 11.4086+3.1998e-08j, 11.4087-5.3131e-08j,
        11.4087-2.4623e-08j, 11.4086-7.6656e-08j, 11.4094-7.0683e-09j,
        11.4087+1.6439e-08j, 11.4075+3.3765e-08j, 11.4079-2.3578e-08j,
        11.4088-8.3967e-09j, 11.4080-3.8000e-08j, 11.4085-1.3877e-08j,
        11.4089+1.2072e-08j, 11.4086+5.3756e-08j, 11.4084-3.0379e-08j,
        11.4068+2.1034e-08j, 11.4049+3.8440e-08j, 11.4076+1.6485e-08j,
        11.4078+4.6072e-08j, 11.4080-5.5585e-08j, 11.4077+9.6651e-08j,
        11.4082-7.1403e-08j, 11.4083+3.3114e-08j, 11.4087+7.4515e-08j,
        11.4086-2.9989e-09j, 11.4093+1.0348e-08j, 11.4088-9.7283e-08j,
        11.4099-4.3203e-08j, 11.4098-9.6481e-08j, 11.4099+4.9497e-08j,
        11.4102+2.9487e-08j, 11.4099+6.6062e-08j, 11.4099+5.2721e-09j,
        11.4103+5.5867e-08j, 11.4091+1.1017e-07j, 11.4096-4.0474e-09j,
        11.4105-5.6858e-08j, 11.4094+4.3045e-08j, 11.4109-2.5757e-08j,
        11.4099+4.2673e-08j, 11.4093-5.1224e-08j, 11.4099+2.0127e-08j,
        11.4103-3.2794e-08j, 11.4106-7.5730e-09j, 11.4094+5.1362e-08j,
        11.4097+6.2265e-08j, 11.4100+3.1511e-08j, 11.4106-6.5711e-08j,
        11.4105+1.9959e-08j])
SNR time: 2904.77 seconds
SNR = 1
mu: 0.05 beta: 0.9
mu: 0.05 beta: 0.94
mu: 0.05 beta: 0.99
mu: 0.37 beta: 0.9
mu: 0.37 beta: 0.94
mu: 0.37 beta: 0.99
mu: 0.68 beta: 0.9
mu: 0.68 beta: 0.94
mu: 0.68 beta: 0.99
mu: 1.0 beta: 0.9
mu: 1.0 beta: 0.94
mu: 1.0 beta: 0.99
mu: 1.5 beta: 0.9
mu: 1.5 beta: 0.94
mu: 1.5 beta: 0.99
mu: 3.5 beta: 0.9
mu: 3.5 beta: 0.94
mu: 3.5 beta: 0.99
mu: 5.5 beta: 0.9
mu: 5.5 beta: 0.94
mu: 5.5 beta: 0.99
mu: 7.5 beta: 0.9
mu: 7.5 beta: 0.94
mu: 7.5 beta: 0.99
Momentum starts training
epoch:  1  of  5
epoch time:  142.5  seconds
epoch:  2  of  5
epoch time:  142.79  seconds
epoch:  3  of  5
epoch time:  143.03  seconds
epoch:  4  of  5
epoch time:  142.22  seconds
epoch:  5  of  5
epoch time:  142.53  seconds
SNR time: 2941.59 seconds
SNR = 2
mu: 0.05 beta: 0.9
mu: 0.05 beta: 0.94
mu: 0.05 beta: 0.99
mu: 0.37 beta: 0.9
mu: 0.37 beta: 0.94
mu: 0.37 beta: 0.99
mu: 0.68 beta: 0.9
mu: 0.68 beta: 0.94
mu: 0.68 beta: 0.99
mu: 1.0 beta: 0.9
mu: 1.0 beta: 0.94
mu: 1.0 beta: 0.99
mu: 1.5 beta: 0.9
```

```
mu: 1.5 beta: 0.9
mu: 1.5 beta: 0.94
mu: 1.5 beta: 0.99
mu: 3.5 beta: 0.9
mu: 3.5 beta: 0.94
mu: 3.5 beta: 0.99
mu: 5.5 beta: 0.9
mu: 5.5 beta: 0.94
mu: 5.5 beta: 0.99
mu: 7.5 beta: 0.9
mu: 7.5 beta: 0.94
mu: 7.5 beta: 0.99
Momentum starts training
epoch:  1  of  5
epoch time:  151.22  seconds
epoch:  2  of  5
epoch time:  154.49  seconds
epoch:  3  of  5
epoch time:  155.93  seconds
epoch:  4  of  5
epoch time:  154.68  seconds
epoch:  5  of  5
epoch time:  149.52  seconds
SNR time: 3309.13 seconds
SNR = 3
mu: 0.05 beta: 0.9
mu: 0.05 beta: 0.94
mu: 0.05 beta: 0.99
mu: 0.37 beta: 0.9
mu: 0.37 beta: 0.94
mu: 0.37 beta: 0.99
mu: 0.68 beta: 0.9
mu: 0.68 beta: 0.94
mu: 0.68 beta: 0.99
mu: 1.0 beta: 0.9
mu: 1.0 beta: 0.94
mu: 1.0 beta: 0.99
mu: 1.5 beta: 0.9
mu: 1.5 beta: 0.94
mu: 1.5 beta: 0.99
mu: 3.5 beta: 0.9
mu: 3.5 beta: 0.94
mu: 3.5 beta: 0.99
mu: 5.5 beta: 0.9
mu: 5.5 beta: 0.94
mu: 5.5 beta: 0.99
mu: 7.5 beta: 0.9
mu: 7.5 beta: 0.94
mu: 7.5 beta: 0.99
Momentum starts training
epoch:  1  of  5
epoch time:  144.51  seconds
epoch:  2  of  5
epoch time:  145.52  seconds
epoch:  3  of  5
epoch time:  150.85  seconds
epoch:  4  of  5
epoch time:  140.65  seconds
epoch:  5  of  5
epoch time:  143.36  seconds
SNR time: 3329.48 seconds
SNR = 4
mu: 0.05 beta: 0.9
mu: 0.05 beta: 0.94
mu: 0.05 beta: 0.99
mu: 0.37 beta: 0.9
mu: 0.37 beta: 0.94
mu: 0.37 beta: 0.99
mu: 0.68 beta: 0.9
mu: 0.68 beta: 0.94
mu: 0.68 beta: 0.99
mu: 1.0 beta: 0.9
mu: 1.0 beta: 0.94
```

```
mu: 1.0 beta: 0.94
mu: 1.0 beta: 0.99
mu: 1.5 beta: 0.9
mu: 1.5 beta: 0.94
mu: 1.5 beta: 0.99
mu: 3.5 beta: 0.9
mu: 3.5 beta: 0.94
mu: 3.5 beta: 0.99
mu: 5.5 beta: 0.9
mu: 5.5 beta: 0.94
mu: 5.5 beta: 0.99
mu: 7.5 beta: 0.9
mu: 7.5 beta: 0.94
mu: 7.5 beta: 0.99
Momentum starts training
epoch:  1  of  5
epoch time:  147.56  seconds
epoch:  2  of  5
epoch time:  146.74  seconds
epoch:  3  of  5
epoch time:  146.57  seconds
epoch:  4  of  5
epoch time:  147.4  seconds
epoch:  5  of  5
epoch time:  148.4  seconds
SNR time: 3367.88 seconds
SNR = 5
mu: 0.05 beta: 0.9
mu: 0.05 beta: 0.94
mu: 0.05 beta: 0.99
mu: 0.37 beta: 0.9
mu: 0.37 beta: 0.94
mu: 0.37 beta: 0.99
mu: 0.68 beta: 0.9
mu: 0.68 beta: 0.94
mu: 0.68 beta: 0.99
mu: 1.0 beta: 0.9
mu: 1.0 beta: 0.94
mu: 1.0 beta: 0.99
mu: 1.5 beta: 0.9
mu: 1.5 beta: 0.94
mu: 1.5 beta: 0.99
mu: 3.5 beta: 0.9
mu: 3.5 beta: 0.94
mu: 3.5 beta: 0.99
mu: 5.5 beta: 0.9
mu: 5.5 beta: 0.94
mu: 5.5 beta: 0.99
mu: 7.5 beta: 0.9
mu: 7.5 beta: 0.94
mu: 7.5 beta: 0.99
Momentum starts training
epoch:  1  of  5
epoch time:  151.66  seconds
epoch:  2  of  5
epoch time:  150.38  seconds
epoch:  3  of  5
epoch time:  150.11  seconds
epoch:  4  of  5
epoch time:  150.2  seconds
epoch:  5  of  5
epoch time:  149.66  seconds
SNR time: 3421.76 seconds
```

```
/usr/local/lib/python3.10/dist-packages/torch/_tensor.py:972: ComplexWarning: Casting com
plex values to real discards the imaginary part
  return self.numpy().astype(dtype, copy=False)
```






```
line search max:  [tensor(11.2570), tensor(12.2835), tensor(13.3222), tensor(14.3772), tensor(15.4529), tensor(16.5478)]
pga 400 max:  [tensor(11.4105+1.9959e-08j), tensor(12.4389+6.3842e-08j), tensor(13.4860+8.2025e-09j), tensor(14.5541+1.8625e-08j), tensor(15.6367+2.2638e-08j), tensor(16.7318-2.5085e-08j)]
pga 10 max:  [tensor(10.4021-6.1937e-09j), tensor(10.9638-3.2284e-08j), tensor(12.1190-2.5755e-08j), tensor(12.9912-2.1941e-08j), tensor(14.2235-1.6859e-08j), tensor(15.4480-4.7882e-09j)]
momentum max:  [tensor(11.1271+5.4341e-10j, grad_fn=<SelectBackward0>), tensor(12.1105+2.7457e-09j, grad_fn=<SelectBackward0>), tensor(13.1624+5.4161e-08j, grad_fn=<SelectBackward0>), tensor(14.2481-2.7585e-08j, grad_fn=<SelectBackward0>), tensor(15.2860-1.3050e-07j, grad_fn=<SelectBackward0>), tensor(16.3494-1.2638e-07j, grad_fn=<SelectBackward0>)]
```

In [ ]:

```
plot_Mu_W(momentum_model.Mu_W,J)
plot_betas(momentum_model.betas,J)
```