ECEN-689: Project Final Report

Seyedeh Nahid Esmati, Arghamitra Talukder, and Ya Mei
Department of Electrical and Computer Engineering
Texas A&M University
nahid@tamu.edu, arghamitra.talukder@tamu.edu, yxm5152@tamu.edu

1 Introduction

Group testing (GT) is a well-known problem in statistics and combinatorial mathematics—that has found application in several real-world scenarios such as cybersecurity, bloom filters, public health, data science, and information theory (for a comprehensive survey on group testing, see [1]). In particular, GT has received a lot of attention during the Covid-19 pandemic, see, e.g., [2]. In the GT problem, the goal is to design tests for identifying the defective items among a much larger group of items using the minimum number of tests, while achieving a target level of accuracy. Most of the existing work on GT consider *binary tests* (with the test result being either *positive* or *negative*) on pools of items. In the noiseless setting of GT, each test is assumed to be error-free, and the noisy GT is based on the assumption that the binary tests are noisy. In many applications such as GT for COVID-19 detection, the latter assumption is expected to yield a more realistic model [2].

In non-adaptive (noiseless or noisy) GT, the pooling algorithm is specified by a binary testing matrix. In particular, each row of the testing matrix corresponds to one test, and the location of 1's in a row represent the indices of the items participating in the test corresponding to that row. In contrast to non-adaptive GT, there are adaptive GT schemes where testing is performed in multiple stages. A commonly used (non-adaptive) pooling algorithm for GT is based on randomly generated sparse binary testing matrices [1, 3]. A class of testing matrices that have been predominantly studied in the literature are *regular sparse testing matrices* [2]. A regular sparse matrix is designed by randomly generating a binary matrix with a fixed number of 1's per row and a fixed number of 1's per column.

In a noisy GT setting, given the observed noisy test result (also known as the noisy measurement vector), either (i) the goal is to correctly classify every item as defective or non-defective (*exact recovery*), or (ii) it suffices to identify each item as defective or non-defective such that the total number of false negatives and the total number of false positives are sufficiently small (*partial recovery*). Given a pooling algorithm (i.e., a testing matrix), in order to satisfy the requirement (i) or (ii), one needs to run a recovery algorithm on the noisy test results.

One of the recovery algorithms that has been successfully used for both noiseless and noisy GT is Belief Propagation (BP) [1, Section 3.3], which is a special case of iterative message-passing algorithms [4]. Consider the Tanner graph corresponding to a testing matrix, i.e., a bipartite graph whose left nodes (referred to as *variable nodes*) represent the components of the original signal (i.e., the status of all items), and right nodes (referred to as *check nodes*) represent the components in the noisy measurement vector. At each round of the recovery algorithm, some messages are sent from variable nodes to check nodes, and some messages are sent from check nodes back to variable nodes. This iterative message-passing process continues until the original signal is recovered or the process is halted. Conventionally, in the BP recovery algorithm, at each iteration the messages are propagated via a flooding approach (also known as simultaneous scheduling), that is, all variable nodes send their messages to their neighboring check nodes simultaneously, and all check nodes send their messages to their neighboring variables nodes simultaneously.

Recently, it was shown in [5] that the error performance and the computational complexity of the BP algorithm for decoding sparse graph codes such as Low-Density Parity-Check (LDPC) codes [4] can be improved by devising a single check-node scheduling policy, instead of flooding. The problem of designing an optimal scheduling policy can be modeled as a Reinforcement Learning (RL) problem.

In particular, the scheduling problem can be formulated as a Multi-Armed Bandits (MAB) problem which is a classical Markov Decision Process (MDP) problem. Motivated by the recent work of [6], our goal in this project is to design RL-based scheduling policies for the BP recovery algorithm that are superior to the flooding approach in the context of noisy GT.

2 Problem Formulation

Consider a population of n items (labeled $1, \ldots, n$), among which k ($\ll n$) are defective, and the rest are not defective. Let $\mathbf{x} = [x_1, \ldots, x_n]$ be a binary vector of length n that represents the status of the items in the population. Specifically, for any $v \in [n] = \{1, \ldots, n\}$, $x_v = 1$ if the item v is defective, and $x_v = 0$ otherwise. Let $H \in \{0,1\}^{m \times n}$ be a binary matrix of size $m \times n$ that represents the matrix used for non-adaptive group testing, i.e., the positions of the 1's in the cth row of H determine the indices of the items participating in the test c. Note that m represents the total number of tests performed on the population. The matrix H is referred to as the testing matrix in the GT literature. Motivated by the practical applications, in this work we focus on randomly generated sparse binary testing matrices. In particular, we consider randomly generated binary matrices that have a fixed number of 1's per row (denoted by d_c) and a fixed number of 1's per column (denoted by d_v). This implies that each test contains the same number of items, and each item participates in the same number of tests. Such random testing matrices are analogous to random linear codes, and are motivated by the success of random linear codes in the literature of channel coding.

In the noiseless setting of group testing, the result of each test is given by the Boolean inclusive OR of the status of the items participating in that test. That is, if $\mathcal{I}_c$ is the index set of the items in the test c, then the result of the test c, denoted by y_c, is given by $\vee_{v \in \mathcal{I}_c} x_v$, where the symbol "$\vee$" represents the Boolean inclusive OR operator. Let $\mathbf{y} = [y_1, \ldots, y_m]$ be the vector of the test results in the absence of any test errors. Note that, for each $c \in [m] = \{1, \ldots, m\}$, y_c is either 0 or 1. In particular, $y_c = 0$ if and only if none of the items participating in the test c is defective, and $y_c = 1$ otherwise. In the case that the tests are subject to error, we observe a noisy version of the results vector $\mathbf{y}$, denoted by $\hat{\mathbf{y}} = [\hat{y}_1, \ldots, \hat{y}_m]$, where $\hat{y}_c$ for each $c \in [m]$ is the output of a noisy channel for the input y_c. In this work, we model the noisy channel by a (memoryless) Binary Symmetric Channel (BSC) with crossover probability $0 < \delta < 1$. That is, for each test $c \in [m]$, independently of the other tests, $\hat{y}_c = y_c$ with probability $1 - \delta$, or $\hat{y}_c = 1 \oplus y_c$ with probability δ, where the symbol "$\oplus$" represents the Boolean exclusive OR operator (i.e., the addition in the binary field $\mathbb{F}_2$).

Given the observed (noisy) vector $\hat{\mathbf{y}}$, the goal is to find an estimate $\hat{\mathbf{x}}$ of the status vector $\mathbf{x}$ such that the *success probability*, i.e., the probability of $\hat{\mathbf{x}} = \mathbf{x}$, is maximized. There are several recovery algorithms in the literature of noisy GT. One of the most commonly used algorithms is ***Belief Propagation (BP)***. The BP algorithm performs an approximate item-wise Maximum-A-Posteriori (MAP) estimation in order to identify the status of each item. That is, the BP algorithm attempts to solve $\hat{x}_v = \arg\max_{u_v \in \{0,1\}} \mathbb{P}(x_v = u_v | \hat{\mathbf{y}})$ for all $v \in [n]$. In the following, we explain how the BP algorithm works. Consider the Tanner graph corresponding to an $m \times n$ testing matrix H, i.e., a bipartite graph whose n left nodes (referred to as *variable nodes*) represent the status of all the n items, and m right nodes (referred to as *check nodes*) represent the m observed (noisy) test results. At each iteration of the BP algorithm, a subset of check nodes send some messages to their neighboring variable nodes, and a subset of variable nodes send some messages back to their neighboring check nodes.

Each message sent from a check node c to a variable node v is a vector with two components, where the first (or the second) component is the posterior probability of the variable node v to be non-defective (or defective) given the observed result of the test c (i.e., $\hat{y}_c$) and all the messages sent to the check node c in the previous iterations from all its neighboring variable nodes excluding the variable node v. Thus, at the iteration 0, the message vector $m^{(0)}_{c \to v}$ that each check node c sends to each of its neighboring variable nodes v is given by $[m^{(0)}_{c \to v}(0), m^{(0)}_{c \to v}(1)] = [1/2, 1/2]$. On the other hand, each message being sent form a variable node v to a check node c is a vector with two components, where the first (or the second) component is the posterior probability of the variable node v to be non-defective (or defective) given all the messages communicated to the variable node v in the previous iteration excluding the message received from the check node c. It is often assumed that each item is defective (or not) independently from other items. Under this assumption, the message vector $m^{(0)}_{v \to c}$ that each variable node v sends to each of its neighboring check nodes c is

given by $[m^{(0)}_{v\to c}(0), m^{(0)}_{v\to c}(1)] = [1-k/n, k/n]$, where k is the total number of defective items, and n is the total number of items in the population. Note that $q = k/n$ is the prior probability of each item to be defective, and $1-q$ is the prior probability of each item to be non-defective.

Let $\mathcal{N}(c)$ denote the set of labels of the variable nodes that are incident to the check node c (i.e., the index set of items participating in the test c), and let $\mathcal{N}(v)$ denote the set of labels of the check nodes that are incident to the variable node v (i.e., the index set of tests in which the item v participates). Using this notation, the message vector sent from a check node c to a variable node $v \in \mathcal{N}(c)$ at each iteration $l \geq 1$ is given by $m^{(l)}_{c\to v} = [m^{(l)}_{c\to v}(0), m^{(l)}_{c\to v}(1)]$, where

$$m^{(l)}_{c\to v}(u_v) \propto \sum_{\{u_{v'}\}_{v'\in\mathcal{N}(c)\setminus\{v\}}} \mathbb{P}(y_c|\{u_i\}_{i\in\mathcal{N}(c)}) \prod_{v'\in\mathcal{N}(c)\setminus\{v\}} m^{(l-1)}_{v'\to c}(u_{v'}), \quad u_v \in \{0,1\}, \tag{1}$$

and the message vector sent from a variable node v to a check node $c \in \mathcal{N}(v)$ at each iteration $l \geq 1$ is given by $m^{(l)}_{v\to c} = [m^{(l)}_{v\to c}(0), m^{(l)}_{v\to c}(1)]$, where

$$m^{(l)}_{v\to c}(u_v) \propto \big(q\mathbb{1}_{\{u_v=1\}} + (1-q)\mathbb{1}_{\{u_v=0\}}\big) \prod_{c'\in\mathcal{N}(v)\setminus\{c\}} m^{(l)}_{c'\to v}(u_v), \quad u_v \in \{0,1\}. \tag{2}$$

Note that the symbol "$\propto$" in (1) or (2) denotes equality up to a normalizing constant such that $m^{(l)}_{c\to v}(0) + m^{(l)}_{c\to v}(1) = 1$ or $m^{(l)}_{v\to c}(0) + m^{(l)}_{v\to c}(1) = 1$, respectively.

For the case of the BSC noise model with crossover probability δ, the update rule (1) for the messages $m^{(l)}_{c\to v}(u_v)$ can be further simplified as follows:

- If $\hat{y}_c = 0$, then

$$m^{(l)}_{c\to v}(u_v) \propto \begin{cases} \delta & u_v = 1 \\ \delta + (1-2\delta)\prod_{v'\in\mathcal{N}(c)\setminus\{v\}} m^{(l-1)}_{v'\to c}(0) & u_v = 0 \end{cases}, \tag{3}$$

- If $\hat{y}_c = 1$, then

$$m^{(l)}_{c\to v}(u_v) \propto \begin{cases} 1-\delta & u_v = 1 \\ 1-\delta-(1-2\delta)\prod_{v'\in\mathcal{N}(c)\setminus\{v\}} m^{(l-1)}_{v'\to c}(0) & u_v = 0 \end{cases}. \tag{4}$$

After running the BP algorithm for a pre-specified number of iterations (denoted by $l_{\max}$), the posterior log-likelihood ratio (LLR) computed by each variable node v is given by

$$L_v = \ln\frac{q}{1-q} + \sum_{c\in\mathcal{N}(v)} \ln\frac{m^{(l_{\max})}_{c\to v}(1)}{m^{(l_{\max})}_{c\to v}(0)}. \tag{5}$$

Note that $\ln\frac{q}{1-q} = \ln\frac{\mathbb{P}(x_v=1)}{\mathbb{P}(x_v=0)}$ is the (initial) posterior LLR computed by each variable node v at the beginning of the iteration 1 since no messages were communicated to the variable node v prior to the iteration 1. Depending on whether the total number of defective items in the population (k) is known *a priori* or not, there are two different decision rules for estimating the status of each item give the (final) posterior LLRs $L_1, \ldots, L_n$. If k is intially known, we sort $L_1, \ldots, L_n$ in descending order, and identify the items corresponding to the first k largest posterior LLRs as the defective items, and identify the rest of the $n-k$ items as non-defective. If k is not known in advance, each item v such that $L_v > 0$ (or $L_v < 0$) will be identified as defective (or non-defective), and each item v such that $L_v = 0$ will be identified as defective (or non-defective) with probability q (or $1-q$).

In the context of channel coding, the BP algorithm has been widely used for decoding of sparse graph codes such as LDPC codes. It was shown that the error performance and the computational complexity of the BP algorithm depends significantly on which check nodes and which variable nodes are scheduled for passing messages in each iteration. There are two main types of scheduling that have been considered previously for LDPC codes: *flooding* and *single check-node scheduling*.

In flooding, at each iteration, all check nodes send messages to their neighboring variable nodes, and all variable nodes send back messages to their neighboring check nodes. In contrast, in single check-node scheduling, at each iteration, one check node is selected according to a policy that is

determined *a priori*. Then, the selected check node sends messages to its neighboring variable nodes. Subsequently, the variable nodes that received some new messages in the current iteration send messages to their neighboring check nodes. For decoding LDPC codes, it has been recently shown that, when compared to flooding, scheduling a single check node according to some greedy policy can improve both the error performance and the computational complexity of the BP algorithm. An intuitive explanation for this improvement is that passing less number of messages in an iteration may yield less error propagation at each iteration. In addition, employing a single check-node scheduling policy obviously reduces the number of messages being communicated in each iteration, and this can substantially reduce the computational complexity of the BP algorithm. To the best of our knowledge, however, no such check-node scheduling was previously considered in the context of GT. This naturally raises a question whether one can design check-node scheduling policies for BP in GT that are superior to flooding. In this work, our goal is to exploit RL towards answering this question.

3 Modelling Check-Node Scheduling as a Markov Decision Process

As was recently shown in [6] for decoding of LDPC codes, the scheduling of a single check node at each iteration of the BP algorithm can be viewed as a Multi-Armed Bandit (MAB) process with dependent arms, and as a consequence, a Markov Decision Process (MDP) can formally model the environment of this problem as an RL problem. An optimal policy for scheduling an arm (i.e., a check node in the problem at hand) can be obtained by solving the corresponding MAB problem.

An MDP is represented by a tuple $(\mathcal{S}, \mathcal{A}, R, P, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the set of actions, $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the immediate reward function that depends on the current state and the current action, and $P(s'|s, a)$ is the probability that the action a in the state s leads to the next state s'. dThe parameter γ represents the discount factor indicating that a reward received in the future is worth less than an immediate reward. The goal is to find a policy for choosing an action in each state such that the (expected) accumulated reward is maximized.

A (finite-horizon) MAB problem with a total of T rounds is formulated as an MDP problem as follows. Suppose there are m slot machines (one-armed bandits), each with an unknown reward distribution. Suppose there is an agent who can pull one of these arms at each time step $t = 1, \ldots, T$. Let $a^{(t)} \in [m]$ be the index of the arm that is pulled (i.e., the action taken by the agent) at the time step t, and let $\mathbf{s}_i^{(t)}$ denote the state of the arm i at the time step t. Note that in a discrete state-space MAB, $s_i^{(t)}$ can take M possible real values for some integer M, whereas in a continuous state-space MAB, $\mathbf{s}_i^{(t)}$ can be any real number. At each time step t, the state of the environment is then defined by a vector $\mathbf{s}^{(t)} = [s_1^{(t)}, \ldots, s_m^{(t)}]$. In a discrete state-space MAB, since there are m arms and the state of each arm can take M possible values, the size of the state space is M^m. In contrast, in a continuous state-space MAB, the size of the state space is infinite. The immediate reward at the time step t, denoted by $R(\mathbf{s}^{(t)}, a^t)$, is defined as the reward that the agent receives (at the time step t) after taking the action $a^{(t)}$ given that the environment at the time step t was in the state $\mathbf{s}^{(t)}$.

Thinking of the m check nodes in the problem of check-node scheduling as the m one-armed bandits in the MAB problem, each iteration as a time step, and selecting a check node for sending messages to its neighboring variable nodes at an iteration as pulling an arm at a time step, it can be seen that the check-node scheduling problem is similar to the MAB problem, and hence can be modelled as an MDP problem. To complete the mapping between the two problems, it remains to properly define the state of the check nodes and the reward function. We have considered various definitions for the state and the reward when implementing different RL algorithms in the following sections.

4 Learning the Optimal Action-Value Function via Q-Learning

A Q-learning based approach was recently proposed in [6, 7] for designing a check-node scheduling policy for LDPC decoding. As closely related to this project, we have implemented this approach (for decoding LDPC codes). However, we were not able to reproduce the results presented in [6, 7]. We found several fundamental inconsistencies between the results presented in [6] and those in an earlier work [8]. In particular, our results were all consistent with the results in [8], but not those reported in [6]. We also implemented this approach for BP in GT. However, we did not observe any successful learning, and our results suggest that this approach may not yield an optimal (or

close to optimal) scheduling policy. As a benchmark, we also implemented the BP algorithm with flooding, and observed that the approach of [6]—directly applied to GT, does not outperform the flooding approach. In the following, we explain our implementation of the Q-learning based approach of [6]—adopted for BP in GT.

Inspired by [6], first we modelled the check-node scheduling problem by a discrete state-space MAB, where the state of each check node is defined as a quantized version of the soft syndrome associated with that check node. Specifically, the state of each check node c at the end of each iteration l, denoted by $s_c^{(l)}$, is given by $s_c^{(l)} = g_M(S_c^{(l)})$, where $g_M : \mathbb{R} \to \mathcal{M}$ is an *M-level scalar quantization function* that maps any real value into one of the values in a pre-determined codebook $\mathcal{M}$ of size M (i.e., the set of M representation points specifying the quantization function), and $S_c^{(l)} = \sum_{v \in \mathcal{N}(c)} L_v^{(l)}$ is the *soft-syndrome* associated with the check node c, where

$$L_v^{(l)} = \begin{cases} \ln \frac{q}{1-q} & l = 0 \\ \ln \frac{q}{1-q} + \sum_{c \in \mathcal{N}(v)} \ln \frac{m_{c \to v}^{(l)}(1)}{m_{c \to v}^{(l)}(0)} & l \geq 1 \end{cases}. \tag{6}$$

Note that $L_v^{(l)}$ is the posterior LLR computed by the variable node v at the end of the iteration l. The state of the environment at the end of the iteration l is then given by the (quantized) soft-syndrome vector $\mathbf{s}^{(l)} = [s_1^{(l)}, \ldots, s_m^{(l)}]$.

Motivated by the work of [5] on BP with check-node scheduling for LDPC decoding, we also considered the immediate reward $R(\mathbf{s}, c) = r_c$ for scheduling the check node c when the state of the environment is $\mathbf{s}$, where r_c is given by

$$r_c = \max_{v \in \mathcal{N}(c)} |L'_{c \to v} - L_{c \to v}|, \tag{7}$$

where

$$L'_{c \to v} = \ln \frac{m'_{c \to v}(1)}{m'_{c \to v}(0)} \quad \text{and} \quad L_{c \to v} = \ln \frac{m_{c \to v}(1)}{m_{c \to v}(0)} \tag{8}$$

are the LLRs corresponding to the messages $m'_{c \to v} = [m'_{c \to v}(0), m'_{c \to v}(1)]$ and $m_{c \to v} = [m_{c \to v}(0), m_{c \to v}(1)]$, respectively. Here, $m'_{c \to v}$ is the message that the check node c would send to the variable node v in the current iteration if the check node c is the check node being scheduled in the current iteration, and $m_{c \to v}$ is the last message that the check node c has sent to the variable node v over the course of the previous iterations. The parameter r_c is referred to as the *maximum residual of the check node c*. The intuitive justification for defining the reward for scheduling each check node based on the maximum residual of that check node is that as the BP algorithm converges, the differences between the LLRs before and after an update diminish. Thus, a check node with the maximum residual is located in a part of the Tanner graph that has not yet converged. As a result, propagating the messages corresponding to such a check node at the current iteration can speed up the convergence of the BP algorithm [5].

Although the M-level scalar quantization technique helps in reducing the size of the original state space, the size of the resulting state space (M^m) is still exponential in the number of check nodes (m). In order to further reduce the size of the space state and improve the computational complexity of the learning process, we employed the clustering technique proposed in [6]. The idea is to partition the check nodes into m/z groups, each of size z. For the ease of notation, here we have assumed that z divides m. For each $i \in [m/z]$, suppose that the cluster i is formed by the check nodes $(i-1)z + 1, \ldots, iz$. Instead of defining a state for each check node, we can now define a state for each cluster. In particular, the state of each cluster i at the end of the iteration l is given by $\mathbf{s}_i^{(l)} = [s_{(i-1)z+1}^{(l)}, \ldots, s_{iz}^{(l)}]$, where $s_c^{(l)} = g_M(S_c^{(l)})$ is the quantized soft-syndrome of the check node c. Note that the size of the state space for each cluster is M^z, and as a consequence, the size of the state space of the environment is now reduced to $\frac{m}{z} \times M^z$ (instead of M^m). The action space of each cluster i is defined by $[z] = \{1, \ldots, z\}$, where the action $a_i \in [z]$ for the cluster i means that the a_ith check node in the cluster i is selected for message passing in the current iteration. In addition, the immediate reward obtained by scheduling the a_ith check node in the cluster i (i.e., the check node $(i-1)z + a_i$) when the state of the cluster i is $\mathbf{s}_i$ is defined as

$$R(\mathbf{s}_i, a_i) = r_{(i-1)z+a_i}, \tag{9}$$

where $r_{(i-1)z+a_i}$ is the maximum residual of the check node $(i-1)z + a_i$, as defined in (7).

In order to design an M-level scalar quantization function $g_M(\cdot)$, we first created a sufficiently large dataset of soft-syndrome values through a Monte Carlo simulation of the BP algorithm with the greedy check-node scheduling policy of [5] for 10 iterations and for 1000 randomly generated instances (i.e., 1000 randomly generated status vectors $\mathbf{x}$). Next, we used the collected soft-syndrome samples as the input of the Lloyd-Max algorithm (with a pre-specified quantization level M) that recursively optimizes the $M-1$ boundary values and the M representation values of the M-level scalar quantizer by minimizing the mean squared error (MSE) over the entire dataset. For instance, for a population of size $n = 100$ among which $k = 5$ items are defective, $m = 30$ tests specified by a randomly generated 30×100 regular binary matrix H with column weight $d_v = 3$ and row weight $d_c = 10$, the noise crossover probability $\delta = 0.05$, and the quantization level $M = 4$, the optimal codebook (i.e., the set of the representation points) we obtained was $\mathcal{M} = \{-63.71, -51.86, -37.05, -28.96\}$.

Next, we explain the steps of the clustered Q-learning algorithm we implemented for learning the action-value function $Q(\mathbf{s}_i, a_i)$ for all $i \in [m/z]$, $\mathbf{s}_i \in \mathcal{M}^z$, and $a_i \in [z]$.

The algorithm has two inputs: the testing matrix H and a collection $\hat{\mathcal{Y}}$ of T randomly generated noisy test results vectors $\hat{\mathbf{y}}_1, \ldots, \hat{\mathbf{y}}_T$, where $\hat{\mathbf{y}}_t$ is a noisy version of the test results vector (i.e., the output of a binary symmetric channel when the input of the channel is the true test results vector $\mathbf{y}_t$) for a randomly generated status vector $\mathbf{x}_t$ (with k randomly positioned 1's, and $n-k$ 0's). Given the inputs H and $\hat{\mathcal{Y}}$, the algorithm outputs an estimation of the action-value function $Q^{(l_{\max})}(\mathbf{s}_i, a_i)$ for all $i, \mathbf{s}_i, a_i$, where $l_{\max}$ is the total number of iterations of the BP algorithm for each input $\hat{\mathbf{y}}_t \in \hat{\mathcal{Y}}$.

The algorithm begins by initializing $Q^{(0)}(\mathbf{s}_i, a_i) = 0$ for all $i, \mathbf{s}_i, a_i$. For each input $\hat{\mathbf{y}}_t \in \hat{\mathcal{Y}}$, the algorithm performs the following procedure recursively. First, the algorithm computes the corresponding soft-syndrome $S_c = \sum_{v \in [n]} H_{c,v} L_v^{(l)}$ for each check node c, where $H_{c,v}$ represents the entry (c, v) of the matrix H, and the posterior LLR $L_v^{(l)}$ is computed according to (6). Then, the soft-syndromes of all check nodes will be quantized using the M-level scalar quantization function $g_M(\cdot)$, in order to obtain $s_c = g_M(S_c)$. The algorithm then computes the state of each cluster i, i.e., $\mathbf{s}_i = [s_{(i-1)z+1}, \ldots, s_{iz}]$. Next, for each iteration $0 \leq l < l_{\max}$, the algorithm proceeds as follows. In each iteration, the algorithm first performs check-node scheduling according to the following ϵ-greedy policy: with probability ϵ, a cluster i is selected at random and a check node c is then randomly selected from the cluster i, or with probability $1-\epsilon$, the check node $c = (i-1)z + a_i$ is selected where $a_i \in [z]$, $i \in [m/z]$ maximizes the function $Q^{(l)}(\mathbf{s}_i, a_i)$. Suppose that the scheduled check node c belongs to the cluster i, and $\mathbf{s}_i$ is the (current) state of the cluster i. The algorithm proceeds with the message propagation in the current iteration (l) via a double for-loop as follows. In the outer for-loop, the scheduled check node c sends the message $m_{c \to v}^{(l)}$ (given by (3) or (4)) to each of its neighboring variable nodes $v \in \mathcal{N}(c)$, and the posterior LLR of the variable node v, i.e., $L_v^{(l)}$, is updated according to (6). In the inner for-loop, each of the variable nodes $v \in \mathcal{N}(c)$ sends the message $m_{v \to c'}^{(l)}$ (given by (2)) to each of its neighboring check nodes $c' \in \mathcal{N}(v) \setminus \{c\}$, i.e., all the check nodes c' neighboring to the variable node v, excluding the scheduled check node c in the current iteration. Next, the algorithm computes the immediate reward $R(\mathbf{s}_i, a_i)$ according to (9), where i is the cluster to which the scheduled check node c belongs, $\mathbf{s}_i$ is the state of the cluster i in the current iteration, and a_i is the index of the scheduled check node in the cluster i. Given $\mathbf{s}_i$, a_i, $R(\mathbf{s}_i, a_i)$, the action-value function $Q^{(l+1)}(\mathbf{s}_i, a_i)$ us updated according to the following rule:

$$Q^{(l+1)}(\mathbf{s}_i, a_i) = (1-\alpha) Q^{(l)}(\mathbf{s}_i, a_i) + \alpha \left(R(\mathbf{s}_i, a_i) + \gamma \max_{j \in [m/z], a_j \in [z]} Q^{(l)}(f(\mathbf{s}_j, a_j), a_j) \right) \quad (10)$$

where α is the learning rate, γ is the discount factor, and $\mathbf{s}_j$ is the state of the cluster j in the current iteration, and $f(\mathbf{s}_j, a_j)$ is the next state of the cluster j if the a_jth check node in cluster j was scheduled in the current iteration. Lastly, the algorithm computes the updated soft-syndrome value of each check node c' that is a neighbor of the neighbors of the check node c (selected in the current iteration) according to $S_{c'} = \sum_{v' \in \mathcal{N}(c')} L_{v'}^{(l)}$, and computes the quantized value $s_{c'}$ given by $s_{c'} = g_M(S_{c'})$. Note that the updated values of $s_1, \ldots, s_m$ determine the next state of each cluster, and will be used at the beginning of the next iteration of the algorithm. Once the states are updated, the algorithm proceeds to the next iteration, and repeats the above procedure $l_{\max}$ times.

Once the algorithm is run for all the T input vectors $\hat{\mathbf{y}}_1, \ldots, \hat{\mathbf{y}}_T$, the most updated function $Q^{l_{\max}}(\cdot, \cdot)$ can be used as an estimate of the optimal action-value function.

Table 1: Success probability of BP algorithm with flooding and maximum-residual node scheduling

	Flooding	Maximum-Residual Node Scheduling
Success Probability (%) (Unknown k)	32.99	51.22
Success Probability (%) (Known k)	50.00	61.34

Instead of a direct implementation of the update rule (9) proposed in [6], which proved very computationally expensive in our simulations and not yielding any successful learning, we also tried modifying the update rule. In particular, instead of allowing the next state of the cluster j (i.e., $f(\mathbf{s}_j, a_j)$ in (9)) to depend on the action a_j, we considered the next state of each cluster j resulting from the action that is taken in the current iteration (i.e., a_i). Although this approach is more consistent with the update rule of the action-value function in the RL literature, it still did not yield a successful learning of the action-value function.

5 Learning an Optimal Node Scheduling Policy via Deep Q Network and Advantage Actor-Critic Policy Gradient Algorithms

In this section, we explain our implementation for the actor-critic policy gradient method as well as two variants of the Deep Q Network (DQN) algorithm that we have implemented, with the goal of finding an optimal node scheduling policy that maximizes the success probability of the BP recovery algorithm for noisy group testing, and we report our results for each algorithm.

For the sake of comparison as well as simulation complexity, we have focused on a case study in which the number of items in the population $n = 100$, the number of tests $m = 20$, the number of defective items in the population $k = 2$, and the crossover probability of the BSC (noisy) channel $\delta = 0.05$. In our simulations, the testing matrix H is a randomly generated 20×100 Bernoulli matrix, where each entry of this matrix is 1 with probability $\ln(2)/2 \approx 0.35$ (and 0 otherwise). The number of BP iterations for the flooding approach and the node scheduling approach was fixed to be 10 and 100, respectively. This is because for our example, the success probability of the flooding approach converges in less than 10 BP iterations, and running the BP algorithm with flooding for a larger number of iterations does not change the success probability. For the node scheduling approach, we allow enough number of iterations (i.e., 100 for this example) to make sure the success probability converges to a value. Throughout, we will compare the results of each of our RL-based scheduling policies with the results of the flooding approach [1] and the maximum-residual node scheduling of [8]. The results for these benchmark approaches are summarized in Table 1. These results are averaged over 1000 randomly generated problem instances. Depending on whether the number of defective items k is initially known or not, the decision rule at the end of the BP iterations will be different, and hence, the success probability of each algorithm will be different for the two cases.

In the implementation of the RL algorithms, we have used several different notions of *state* and *reward*. In particular, we have considered the following four definitions for the state:

(S1) The sum of the LLRs of the variable nodes connected to each check node,

$$\left[\sum_{v\in\mathcal{N}(1)} L_v^{(l)}, \sum_{v\in\mathcal{N}(2)} L_v^{(l)}, \ldots, \sum_{v\in\mathcal{N}(m)} L_v^{(l)}\right],$$

where $L_v^{(l)}$ is defined as in (6).

(S2) The LLRs of the variable nodes connected to each check node,

$$\left[\{L_v^{(l)}\}_{v\in\mathcal{N}(1)}, \{L_v^{(l)}\}_{v\in\mathcal{N}(2)}, \ldots, \{L_v^{(l)}\}_{v\in\mathcal{N}(m)}\right].$$

(S3) A quantized version of (i) or (ii).

(S4) The observed noisy test results $\hat{y}_1, \ldots, \hat{y}_m$ along with (S1), (S2), or (S3).

The four different definitions that we have considered for the reward are listed as follows:

(R1) The quantity r_{c^*} defined as the maximum change in the LLR of a variable node incident to the currently-scheduled check node c^*, i.e., $r_{c^*} = \max_{v \in \mathcal{N}(c^*)} |L'_{c^* \to v} - L_{c^* \to v}|$, where $L'_{c^* \to v}$ and $L_{c^* \to v}$ are defined in (8).

(R2) The ratio R defined as the maximum change in the LLR of a variable node incident to the currently-scheduled check node c^* to the maximum change in the LLR of any variable node when scheduling any check node c, i.e., $R = \frac{r_{c^*}}{\max_{c \in [m]} r_c}$, where r_c is defined in (7).

(R3) Minus the Hamming distance between the original signal $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ and the currently-estimated signal $\hat{\mathbf{x}}^{(l)}$, where $\hat{\mathbf{x}}^{(l)}$ is given by

$$\hat{\mathbf{x}}^{(l)} = \left[\mathbb{1}_{(L_1^{(l)} > 0)}, \mathbb{1}_{(L_2^{(l)} > 0)}, \ldots, \mathbb{1}_{(L_n^{(l)} > 0)} \right].$$

(R4) A linear (or nonlinear) function of (R1), (R2), or (R3) that encourages higher rewards and/or discourages lower rewards.

5.1 Actor-Critic Policy Gradient

In order to learn an optimal node scheduling policy, we implemented the policy gradient algorithm with the actor-critic method. In the following we focus on our implementation where we considered (S1) for the state, and (R4) for the reward. Specifically, for the reward, we used $10 \tanh(R - 0.5)$ as a nonlinear function of the ratio R, where R is defined in (R2). We tried various hyper-parameters. In particular, we considered the learning rate $\alpha \in \{0.01, 0.005, 0.001, 0.0001\}$, the discount factor $\gamma \in \{0.9, 0.95, 0.99\}$, the number of hidden layers $\{1, 2, 3, 4, 5\}$ and the number of nodes per hidden layer $\{20, 40, 60, 120, 240\}$ for both the actor and critic neural networks. In the following, we present our results for the case in which both (actor and critic) neural networks contain 2 hidden layers each with 60 nodes, and the discount factor γ and the learning rate α are 0.99 and 0.001, respectively.

Figure 1 depicts the episodic reward and the average episodic reward during the training phase, and Table 2 summarizes the success probability of the BP algorithm with the learned scheduling policy (for different number of training episodes) over 1000 randomly generated problem instances.

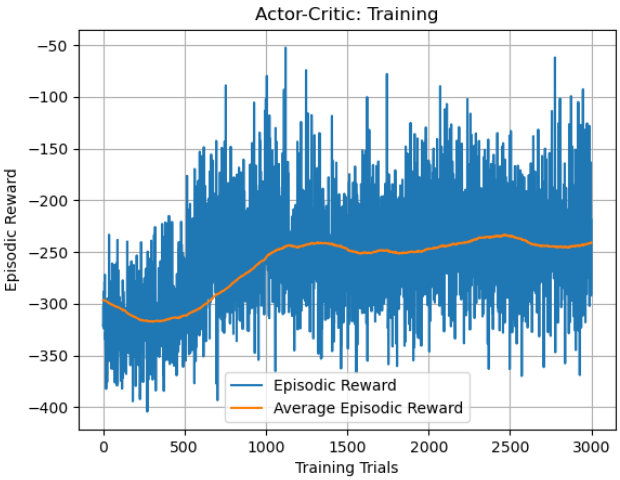

Figure 1: Episodic reward during the training (actor-critic method)

In the training process depicted in Figure 1, we observe that the average episodic reward converges to about -240 after approximately 1100 episodes. We observe that there is high variance associated with the implemented actor-critic policy gradient for our problem. Also, comparing the success probability of our RL-based policy (Table 2) with that of the maximum-residual policy (Table 1) shows that our PG algorithm converges to a sub-optimal policy. Leveraging more advanced PG algorithms such as Natural Policy Gradient (NPG) and Proximal Policy Optimization (PPO) may avoid this pitfall. NPG and PPO are ensuring that a new update of the policy does not change it too much from the previous policy. This can lead to less variance in training, resulting in a smoother training.

Table 2: Success probability of our actor-critic based node scheduling policy

	RL-based Node Scheduling					
	500 eps	1000 eps	1500 eps	2000 eps	2500 eps	3000 eps
Success Probability (%) (Unknown k)	35.4	45.5	47.1	48.3	48.9	49.2
Success Probability (%) (Known k)	47.2	56.4	57.6	58.1	58.5	59.0

As we can see, the results of Table 2 show that our RL-based node scheduling policy learned through the actor-critic method outperforms the flooding approach in terms of success probability, and performs very closely to the maximum-residual node scheduling policy in terms of success probability. For instance, the BP algorithm using our RL-based policy (learned after 3000 training episodes) achieves a success probability of 49.2% for the case of unknown k, whereas BP with flooding (or maximum-residual policy) achieves a success probability of about 33% (or 51%). It should also be noted that our RL-based policy has lower computational complexity as compared to the maximum-residual policy. This is because, unlike our RL-based policy, the maximum-residual policy needs to perform an expensive computation at each BP iteration in order to compute the maximum residual (defined in (7)) for each check node.

5.2 DQN with a Single Neural Network

Another approach we took was trying to learn the action-value function $Q(\cdot, \cdot)$ by implementing a DQN with experience replay and target network, instead of directly updating the action-value function which proved computationally expensive even after quantizing the soft-syndrome of each check node as proposed in [6]. More specifically, we considered a continuous state space, and used a neural network as a non-linear function approximation of the optimal action-value function that yields a policy which maximizes the success probability of BP by scheduling one single check node at each iteration. In each training episode, we used the ϵ-greedy approach for choosing the action in each step (or BP iteration). In the following we focus on our implementation where we considered (S2) for the state, and (R3) for the reward. We tried various hyper-parameters. In particular, we considered the learning rate $\alpha \in \{0.01, 0.005, 0.001, 0.0001\}$, the discount factor $\gamma \in \{0.9, 0.95, 0.99\}$, the number of hidden layers $\{1, 2, 3, 4, 5\}$ and the number of nodes per hidden layer $\{1000, 500, 100\}$ for the agent's neural network. Also, we considered batch sizes $4, 16, 32$, target network's update frequencies $4, 8, 16$, and ϵ decay rates $0.99, 0.999, 0.9999, 0.99999$. In the following, we present our results for the case in which the agent's neural network was composed of 3 hidden layers with $1000, 500, 100$ nodes per layer, respectively, and the batch size 32, the update frequency 4, the discount factor $\gamma = 0.9$, and the learning rate $\alpha = 0.01$ were used.

Figure 2 depicts the episodic reward and the average episodic reward during the training phase. For this algorithm, learning an optimal or sub-optimal node scheduling policy was unsuccessful, and the success probability of the BP algorithm using the final policy was equal to 0.

In the training process depicted in Figure 2, we observe that the average episodic reward converges to about -200 after approximately 1400 episodes. We also observe that there is high variance associated with the implemented DQN algorithm for our problem. Also, we observe that the average episodic curve shows a decreasing trend starting from about -150 in the first episode and converges to -200.

5.3 DQN with Multiple Neural Networks

Another direction that we pursued was trying to learn separate action-value functions for each iteration instead of a universal action-value function. As for the state and reward, (S1) and (R1) were chosen. For each neural network (NN) or agent, two (hidden) layers of dimension 32 and 64 were used. To train each agent, the batch size of 64 and the buffer size of 100,000 were used for implementing the experience replay. The learning rate $\alpha = 0.0001$ and the discount factor $\gamma = 0.99$ were used. To choose the action, the ϵ-greedy policy was selected with an ϵ decay rate of 0.999. Additionally as we

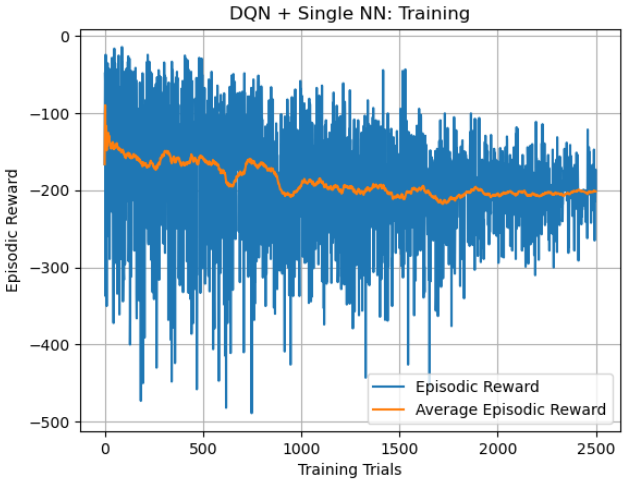

Figure 2: Episodic reward during the training (DQN with a single neural network)

tried to use multiple NNs, or one NN for each BP iteration (instead of one universal NN for all BP iterations), a distinct agent was declared and trained for each BP iteration.

We trained our DQN model using four different number of episodes $\{500, 1500, 2500, 3000\}$. Figure 3 shows the gradual change in reward throughout the training episode number 3000. Irrespective of training episodes the scores (episodic rewards) obtained were consistent which was around 55. Additionally as shown in Figure 3 converging rate also very similar; around episode 500, the scores show the converging pattern or stay almost same.

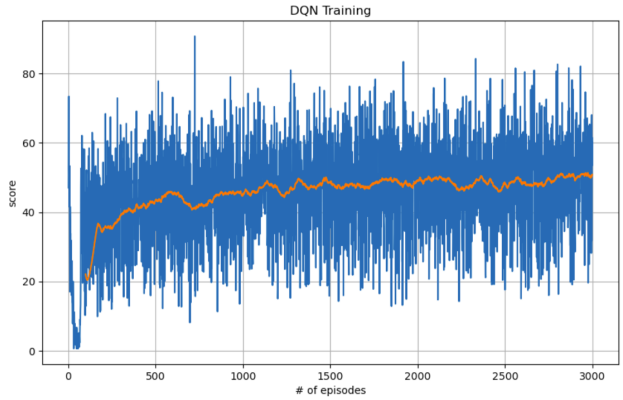

Figure 3: Episodic reward during the training (DQN with multiple neural networks)

To evaluate performance success rate of known k and success rate for unknown k have been used. Table 3 shows the success rates for different number of episodes. Though the score remains same irrespective of episode numbers, the success rate shows a gradual increase with the increased number of episodes. For 500 episodes the success rate of unknown k is 39.2%; for 1500 episodes it is 41.3% (5.36% more than episode 500), 2500 episodes is 46.9% (19.64% more than episode 500) and lastly for 3000 episodes it is 48.7% (24.23% more than episode 500). For known k, 500 episodes of training shows 50.2% success rate; for 1500 episodes of training, it is 53.5% (6.57% more than episode 500), 2500 episodes of training has 58.3% (16.53% more than episode 500) and lastly 3000 episodes of training shows 60.3% (20.12% more than episode 500) success rate.

Table 3: Success probability for different episodes

Success Probability	**500**	**1500**	**2500**	**3000**
% for unknown k	39.2	41.3	46.9	48.7
% for known k	50.2	53.5	58.3	60.3

6 Conclusion

6.1 Summary and Takeaway Messages

1. Our goal in this work was to find a node scheduling (NS) policy that maximizes the success probability of the BP recovery algorithm for noisy group testing.
2. Our results show that the learned RL-based NS policies based on the (advantage) actor-critic policy gradient algorithm and the DQN algorithm with multiple agents (neural networks) outperform the flooding approach and perform very closely to the maximum-residual NS policy in terms of success probability, and are less computationally complex.
3. The gap between the performance of our RL-based NS policies and that of the maximum-residual NS policy imply that the RL algorithms that we have implemented converge to sub-optimal policies.
4. Our attempt towards learning a universal action-value function (for all iterations of the BP algorithm) using the DQN algorithm with a single agent was not successful in learning the optimal policy.
5. Our implementation of the Q-learning algorithm (without function approximation) along with clustering and quantization did not yield any optimal (or close to optimal) scheduling policy.

6.2 Future Work

There are several other avenues that can be pursued in future. In the following we list a few of them.

1. One can apply averaged-DQN in order to improve the performance of DQN algorithm. The Averaged-DQN is a simple extension to the DQN algorithm, based on averaging previously learned Q-values estimates, which leads to a more stable training procedure and improved performance by reducing approximation error variance in the target values.
2. Improving policy gradient by applying a more directed exploration strategy that promotes exploration of under-appreciated rewards is another direction that can be taken as future work.
3. Another direction for future work is to implement more advanced policy gradient algorithms e.g., Natural Policy Gradient (NPG) and Proximal Policy Optimization (PPO) that ensure a new update of the policy does not yield a drastic change in the policy as compared to the policies learned in the past. This can lead to less variance in training, resulting in a smoother training, and may avoid converging to a sub-optimal policy.
4. Implementing DQN with multiple agents (neural networks) combined with target network is another approach that can be taken.
5. One can consider different ways of defining "actions" and/or "states". This will allow to investigate the trade-off between the rate of convergence and the computational complexity of the recovery algorithm. In particular, an action can be defined as scheduling more than one check node at each iteration, or as a tuple with two coordinates, where the first coordinate represents the index of the check node to be scheduled in the current iteration, and the second coordinate specifies whether this check node must flip its received value (from 0 to 1, or from 1 to 0). This, indeed, allows some room for exploration, and in particular, this may be helpful when the recovery algorithm halts prematurely. Similarly, one can consider several ways to define the state of a check node. For instance, the state of a check node can be defined based on the likelihood of that check node to correspond to an erroneous test or an error-free test, given the messages previously received by that check node.

6. Different metrics for defining the "immediate reward" can also be considered. In particular, one can define the reward as a function of the false negative rate and the false positive rate of the estimated status vector after scheduling a check node at each iteration. This modification can potentially speed up the convergence of the learning algorithm and yield a superior scheduling policy in terms of the error performance and/or the computational complexity.
7. The proposed RL-based BP recovery algorithms and the flooding (or maximum-residual) based BP recovery algorithm need to be compared for different range of problem parameters such as the population size (n), the number of defective items (k), the number of tests (m), the row/column weights or the sparsity parameter of the testing matrices, the noise model and the noise parameter(s), and the tunable hyper-parameters in the recovery algorithms under the consideration such as the discount factor (γ), the learning rate (α), the buffer size, the update frequency of the target network, and the number and the size of the hidden layers of the neural networks in DQN and actor-critic algorithms. In addition, instead of the exact recovery setting that we considered in this work, one can consider the *expected false negative rate* and the *expected false positive rate* in the partial recovery setting.

References

[1] Matthew Aldridge, Oliver Johnson, and Jonathan Scarlett. Group Testing: An Information Theory Perspective. *Foundations and Trends in Communications and Information Theory*, 15(3-4):196–392, 2019.

[2] Sabyasachi Ghosh, Rishi Agarwal, Mohammad Ali Rehan, Shreya Pathak, Pratyush Agrawal, Yash Gupta, Sarthak Consul, Nimay Gupta, Ritika Goyal, Ajit Rajwade, and Manoj Gopalkrishnan. A Compressed Sensing Approach to Group-testing for COVID-19 Detection. 2020.

[3] Matthew Aldridge. The Capacity of Bernoulli Nonadaptive Group Testing. *IEEE Transactions on Information Theory*, 63(11):7142–7148, Nov 2017.

[4] Tom Richardson and Rüdiger Urbanke. *Modern Coding Theory*. Cambridge University Press, 2008.

[5] A. I. V. Casado, M. Griot, and R. D. Wesel. LDPC Decoders with Informed Dynamic Scheduling. *IEEE Transactions on Communications*, 58(12):3470–3479, 2010.

[6] Salman Habib, Allison Beemer, and Joerg Kliewer. learning to decode: Reinforcement learning for decoding of sparse graph-based channel codes. In *Advances in Neural Information Processing Systems*, pages 22396–22406. Curran Associates, Inc.

[7] S. Habib, A. Beemer, and J. Kliewer. Learned Scheduling of LDPC Decoders Based on Multi-armed Bandits. In *2020 IEEE International Symposium on Information Theory (ISIT)*, pages 2789–2794, 2020.

[8] A. I. V. Casado, M. Griot, and R. D. Wesel. Ldpc decoders with informed dynamic scheduling. *IEEE Transactions on Communications*, 58(12):3470–3479, 2010.