Abstract

The Reverse Ising Problem is a mixed integer programming problem with non-convex, quadratic or cubic constraints. This is an NP-hard problem that is extremely expensive using traditional solvers. However, an appropriate choice of values for a set of variables known collectively as the auxiliary array converts the system into a linear system, which can be solved efficiently. Over the summer, we explore using discriminative models to classify given auxiliary arrays as feasible or infeasible with a very high accuracy. We also explore using generative models to produce feasible auxiliary arrays at a faster rate than current approaches. Additionally, we consider the distribution of these generated arrays.

Contents

1 Introduction 2
  1.1 Background and Current Theory: Introduction to the Ising Problem 2
    1.1.1 The Two Ising Problems 2
    1.1.2 Example: 1-bit AND gate 3
    1.1.3 The Need for Auxiliary Spins 4
    1.1.4 The Ising Model as a Nondeterministic Model of Computation 4
  1.2 Introduction to Machine Learning 5
  1.3 Machine Learning for the Reverse Ising Problem 6
  1.4 Project Goal 6

2 Discriminative Models 7

3 Generative Models 7
  3.1 Generative Adversarial Networks (GANs) 8
    3.1.1 Vanilla GAN 8
    3.1.2 Conditional Generative Adversarial Network (cGAN) 16
    3.1.3 Other GANs 22
  3.2 Variational Autoencoders (VAEs) 23
    3.2.1 Vanilla Variational Autoencoders 23
    3.2.2 Other VAEs 32
  3.3 Reinforcement Learning 33
  3.4 Generator-Only 38

∗University of North Texas, Denton, TX
†Tulane University, New Orleans, LA
‡University of Texas at San Antonio, San Antonio, TX
1 Introduction

1.1 Background and Current Theory: Introduction to the Ising Problem

The basis of the Ising model is a collection of spins, typically viewed as magnets. These spins are arranged in a graph. Each spin is in one of two possible states: “spin up” or “spin down” (corresponding to the poles of a magnet). Each spin is free to switch between states, as well as having some amount of influence over the states of other spins. Each spin has a local bias, which governs what state the spin will tend (“prefer”) to be in, and how strong that “preference” is. Each pair of spins can be coupled, which determines if the spins will tend (“prefer”) to be in the same state, and each coupled pair has a coupling strength which defines how strong this “preference” is.

The mathematics of this system is well defined. Each spin is represented with a binary variable $s_i$. The energy of a configuration of the system is given by the Hamiltonian:

$$ H = \sum_i h_i s_i + \sum_{\langle i,j \rangle} J_{i,j} s_i s_j, \quad s_i \in \{-1, 1\}, h_i, J_{i,j} \in \mathbb{R}, \quad (1) $$

where $\langle i,j \rangle$ denotes adjacent sites (each pair is only counted once), $h_i$ denotes the local bias on spin $s_i$, and $J_{i,j}$ denotes the coupling strength between spin $s_i$ and $s_j$.

The system naturally evolves to the configuration that has the lowest total energy. However the energy surface may have multiple local minima, so the system may end up in the configuration that does not have the minimum energy (depending on the starting configuration, the temperature of the system, and the temperature evolution of the system).
1.1.1 The Two Ising Problems

The Ising model is the basis for two distinct yet related problems, the following table summarizes the two problems.

<table>
<thead>
<tr>
<th>Problem Name</th>
<th>Fixed Variables</th>
<th>Unknown Variables</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward Ising Problem</td>
<td>Local bias ($h$'s) and coupling strengths ($J$'s)</td>
<td>Expected final configurations</td>
<td>Integer quadratic programming problem</td>
</tr>
<tr>
<td>Reverse Ising Problem</td>
<td>Expected final configurations</td>
<td>Local bias ($h$'s) and coupling strengths ($J$'s)</td>
<td>Mixed-integer quadratic or cubic programming problem</td>
</tr>
</tbody>
</table>

**Forward Ising Problem** This is the formulation of the Ising problem that is focused on in statistical mechanics. The local bias and coupling strengths are fixed. The goal is to study the expected equilibrium configuration; i.e., the configurations at which the system is most likely to stabilize. This is a quadratic integer programming problem. The forward Ising problem problem is not the focus of our research.

**Reverse Ising Problem** Realizing that we can treat spins as binary values (UP = 1, DOWN = -1), we can exploit the fact that an Ising system is very likely to settle in certain configurations in order to perform Ising-based logic. A key feature of the Ising model is that the local biases and coupling strengths govern which spin configurations are the low energy states of an Ising system. This feature facilitates a reprogrammable Ising system. For a given operation, designate a portion of the spins to correspond to each input; a disjoint portion of spins are designated to correspond to the output of the computation. The binary representation of the input values are encoded onto the designated “input spins” (which are then fixed). The system then evolves to an equilibrium configuration. For an appropriate choice of local biases and coupling strengths, the states of the designated “output spins” will correspond to the binary representation of the correct answer of the computation (with some probability).\(^1\)

1.1.2 Example: 1-bit AND gate

<table>
<thead>
<tr>
<th>Input spin 1 ($s_1$)</th>
<th>Input spin 2 ($s_2$)</th>
<th>Output spin ($s_3$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The truth table for a 1-bit AND gate can be mapped to an Ising model. Define a three spin Ising circuit, where two of the spins are designated as input spins ($s_1$ and $s_2$), and one spin is designated as the output spin ($s_3$). This circuit is depicted in figure\(^1\).

The 1-bit AND logic can be encoded on this Ising model. Values of zero in the truth table correspond to spin values of -1 (DOWN).

Now, in order for this circuit to be accurate we need to program the local biases and coupling strengths ($J_{1,3}, J_{2,3}, h_3$) such that the Hamiltonian (energy) of correct configurations are lower than the Hamiltonians of incorrect configurations\(^2\).

\(^1\)For a fixed set of input values, the probability of the equilibrium configuration corresponding to the correct answer is a function of the Hamiltonians of all the configurations of the system with input spins corresponding to those input values.

\(^2\)Note that the input spins don’t have local biases, and there is no coupling between the inputs. This is because the input spins are fixed during a computation.
The Hamiltonian of this system is equal to

\[ H = \sum_i h_i s_i + \sum_{(i,j)} J_{i,j} s_i s_j \]

\[ = h_3 s_3 + J_{1,3} s_1 s_3 + J_{2,3} s_2 s_3 . \]

The requirement that the Hamiltonian of a correct configuration is less than the Hamiltonian of an incorrect configuration imposes four constraints, corresponding to the four rows in the truth table. For the first row, this corresponds to

\[ H(-1, -1, -1) < H(-1, -1, +1) , \]

in words: the Hamiltonian of the configuration corresponding to "-1 AND -1 equaling -1" is less than the Hamiltonian of the configuration corresponding to "-1 AND -1 equaling +1".

Substituting values into the Hamiltonian:

\[ -h_3 + J_{1,3} + J_{2,3} < -h_3 - J_{1,3} - J_{2,3} . \]

Rearranging gives us the first constraint:

\[ -h_3 + J_{1,3} + J_{2,3} < 0 . \]

Repeating this process for the other rows of the truth table gives us the following system of inequalities.

\[ -h_3 + J_{1,3} + J_{2,3} < 0 \]
\[ -h_3 + J_{1,3} - J_{2,3} < 0 \]
\[ -h_3 - J_{1,3} + J_{2,3} < 0 \]
\[ h_3 + J_{1,3} + J_{2,3} < 0 . \]

Since this is a solvable system of inequalities, there are many solutions. One such solution is

\[ h_3 = 1, J_{1,3} = -1, J_{2,3} = -1 . \]

1.1.3 The Need for Auxiliary Spins

Repeating this process for a 1-bit XOR gates results in a system of inequalities that has no solution. In order to make the system solvable, additional degrees of freedom are required. These are obtained by adding auxiliary spins – extra spins to the system that have no role in the operation being performed. For every auxiliary bit added to the system, the number of configurations that correspond to each correct solution double.

The values of the auxiliary spins is irrelevant to the computation being performed. However, the Hamiltonian is now a function of the states of the auxiliary spins. Thus, in the process of calculating the local biases and coupling strengths, it is necessary to compute feasible sets of states of the auxiliary spins. Consequently, it is necessary to solve for the local biases, coupling strengths, and the states of the auxiliary spins simultaneously. This converts the problem from a linear problem into one that is mixed integer (since the auxiliaries are binary variables) and either a quadratic or a cubic programming problem. For further discussion of the parameters and scaling of the reverse Ising problem refer to Appendix C.

1.1.4 The Ising Model as a Nondeterministic Model of Computation

Ising Machines are physical devices that are realizations of an Ising-based model of computation. An Ising Machine is a nondeterministic model of computation. For each set of inputs, the output may be any valid configuration of the machine. The probability of an incorrect configurations occurring depends on the the energy of the correct and incorrect configurations, which depends on the local biases and coupling strengths.
Another factor influencing the convergence of the system is the temperature, which influences how likely the system is to get caught in a local minima – the probability of a correct result occurring can be improved by adding thermodynamic noise to the system.

Ising Machines have the promise of being more energy efficient than transistor based computers. A state transition on an Ising machine expends 6 zeptojoules ($10^{-21}$ Joules) of energy. Further, a system of coupled magnets evolves to equilibrium almost instantaneously, promising incredible efficiency for complicated calculations.

**Interesting Properties of Ising-Based Models of Computation**

**Compute-in-memory (CIM) architecture** In an Ising-based model of computation, the memory and processor are not necessarily separate hardware components – *Ising machines are examples of non Von-Neumann architectures*. This implies that Ising-based models of computation can be regarded as compute in memory (CIM) computers. This has potential benefits because it can experience reduced memory latency as compared to that experienced by transistor-based computers. What are the effects of this? How does one program such a device?

**Single-component multiplication circuits** Unlike transistor-based architectures which implement multiplication as a sequence of additions, Ising-based models of computation enable multiplication to be implemented as a single operation.

Is it possible to implement multiplication in an Ising model with a sequence of addition? Are there benefits to this?

**Reprogrammable circuitry** Suppose the local bias of a spin, as well as all the coupling strengths involving that spin, are set equal to zero. Then that spin will have no effects on any other spins, and neither will it be influenced by any other spins. Thus, given some set of $n$ spins, the system can be programmed as any circuit that requires no more than $n$ spins. (This “reprogramming” may also change the roles of the spins, i.e., a spin may correspond to the input of an operation in one circuit and an auxiliary spin in an operation in the reprogrammed circuit.) Of course, this reprogramming may be completely impossible in practice.

**Potential Limitation of Ising-Based Models of Computation**

Mapping the mathematical Ising system onto hardware has it’s own set of challenges due to hardware specifications.

**Limitations on local biases and coupling strengths** There may be physical limits on the dynamic range of the biases and coupling strengths that can be programmed onto the hardware. Further, there will likely be physical limits on the precision of the biases and coupling strengths, e.g., the values may need to be integers.

**Limitations on time to solution** Even thought the mathematical Ising system evolves to equilibrium configurations instantaneously, depending on the underlying hardware this is likely not be wholly correct on physical implementations of the Ising system. Thus, for certain problem instantiations, the Ising system may not reach a correct minimum energy state quickly enough or consistently enough to be useful.

### 1.2 Introduction to Machine Learning

*Machine learning*, a branch of artificial intelligence, has a number of different applications. These include classifying or generating images, fraud detection, and composing music, among other things. Machine learning operates by providing some feedback to a learning model in order to fine-tune it to be more effective for its designated purpose.

Below is a short breakdown of some important machine learning terms referenced in this paper.

- *Training data* - collection of samples that the machine learning model “learns” from
1.3 Machine Learning for the Reverse Ising Problem

The motivation for utilizing machine learning for our efforts this summer lies heavily in the scale of the attempted operation problems. Listed below are the two multiplication problems we worked on.

- **MUL 2x3x1**: 2-bit by 3-bit multiplication with 1 auxiliary spin
- **MUL 3x3x3**: 3-bit by 3-bit multiplication with 3 auxiliary spins

The MUL 2x3x1 problem’s auxiliary arrays are of length 32, and MUL 3x3x3’s are of length 192. By adding just one input spin and 2 auxiliary spins, the length of our arrays increases by a factor of 6. These length values were calculated with the following formula:

\[
\text{length} = k \cdot 2^{i+j}
\]  

where \(k\) is the number of auxiliary spins and \(i\) and \(j\) are the number of input spins.

As a consequence of larger auxiliary arrays the sample space also grows. For MUL 2x3x1, there are \(10^9\) possible auxiliary array configurations. For MUL 3x3x3, this number of possible configurations expands to \(10^{57}\).

With much larger auxiliary arrays and a much larger sample space, it becomes increasingly difficult to pick out patterns among auxiliary arrays that give clues to their feasibility or infeasibility. From a generative standpoint, with the current iteration of genIsingData.py it is possible to generate small clusters of feasible auxiliary arrays, but there is not a method in place for finding arrays that are sufficiently different from each other. With these considerations in mind, machine learning was an approach worth testing for both classification and generation.

1.4 Project Goal

The reverse Ising problem is an NP-hard, non-linear mixed integer programming problem. The existing solution technique start by searching for viable solutions to the integer variables, called auxiliary arrays. It then uses the auxiliary array to reduce the system to a linear system which it will optimize. Using traditional optimization methods, it typically requires about 47 minutes (see Appendix D) to generate a new feasible auxiliary array on our supercomputer Nemo, a 768 processor HPE Superdome Flex. Our goal this summer was to explore the use of machine learning techniques to classify and generate auxiliary arrays more efficiently than traditional methods.

Our first approach was data classification. We implemented various standard discriminative models to classify given auxiliary arrays as feasible or infeasible. The methods and results are discussed in section 2. Our second approach was data generation. We explored the applicability of several generative models to the reverse Ising problem. The models implemented and the results are discussed in section 3. Conclusions and possible future work are discussed in section 4. A description of the software tools used is found in Appendix A.

---

3See Appendices A and B for a description of scripts
2 Discriminative Models

For the Reverse Ising Problem, our aim was to find classifiers, which are discriminative models, that could quickly and accurately identify any auxiliary array for a particular problem as feasible or infeasible.

Evaluating Discriminative Models The measure of a classifier’s performance for this problem is its accuracy in distinguishing feasible auxiliary arrays from infeasible auxiliary arrays.

Results of Classifiers Tested on MUL 2x3x1 We split a list of 10,000 auxiliary arrays into a training set for the training process and a test set for cross-validation. The cross-validation strategy provides a means of observing a tested model’s accuracy on previously unseen data. We also utilized GridSearchCV() of scikit-learn [2] for hyperparameter optimization. Table 3 summarizes the results of testing 11 different scikit-learn classifiers on the test set.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logistic Regression</td>
<td>0.4932</td>
</tr>
<tr>
<td>Ridge</td>
<td>0.4928</td>
</tr>
<tr>
<td>Bernoulli Naive Bayes</td>
<td>0.5016</td>
</tr>
<tr>
<td>Decision Trees</td>
<td>0.9888</td>
</tr>
<tr>
<td>Passive Aggressive Algorithm</td>
<td>0.4940</td>
</tr>
<tr>
<td>Support Vector Machines</td>
<td>0.9920</td>
</tr>
<tr>
<td>Random Forest</td>
<td>0.9996</td>
</tr>
<tr>
<td>AdaBoost</td>
<td>0.9990</td>
</tr>
<tr>
<td>Gradient Boosting</td>
<td>0.9776</td>
</tr>
<tr>
<td>K-Nearest Neighbors</td>
<td>0.9252</td>
</tr>
<tr>
<td>Radius Nearest Neighbors</td>
<td>0.4952</td>
</tr>
<tr>
<td>Stochastic Gradient Descent</td>
<td>0.4968</td>
</tr>
<tr>
<td>Perceptron</td>
<td>0.5084</td>
</tr>
<tr>
<td>Multi-Layer Perceptron</td>
<td>0.9988</td>
</tr>
</tbody>
</table>

Table 3: Accuracy of scikit-learn Classifiers on the MUL 2x3x1 Dataset

One metric that is used in machine learning is to measure the accuracy of the discriminator model against the accuracy of a “naive” predictor that always predicts the most common class. Our datasets are split evenly between feasible and infeasible samples. So this means that the naive predictor will have an accuracy of 50% (If it always predicts “feasible”, it will be correct 50% of the time). Thus, models with an accuracy of 50% (or less) are of no interest – they are no better than random guessing.

As far as we know, the sample space of auxiliary arrays for any given reverse Ising problem has no regularity or structure. Thus there is no particular reason to expect classification using machine learning to be effective. Seven of our classifiers scored above 90% accuracy in classifying. We consider these to be successful discriminative models. In fact, four of our discriminators were more than 99% accurate in classifying auxiliary arrays, this implies that there is significant structure in the sample space. This result was in itself significant, completely unexpected, and is still not thoroughly understood.

3 Generative Models

Soon after using machine learning for classification, we discovered generative machine learning. Generative models aim to probabilistically describe how a dataset is generated. While discriminative models aim to classify instances in a dataset, generative models attempt to generate new instances by learning the data distribution. Given X data instances with Y labels, generative models aim to learn the joint probability \( P(X, Y) \), or \( P(X) \) if labels do not exist.
Neural Networks  Neural networks are types of machine learning algorithms that attempt to mimic the structure of a brain. Data moves through perceptrons, which are parts of hidden layers. By means of mathematical calculations, usually matrix multiplication, the perceptrons transform the data into something that hopefully fits some target value. A lot of our approaches in this section employ neural networks to generate new feasible arrays.

Evaluating Generative Models  We determined 3 metrics to measure the performance of our generative models:

1. Quantity: For any approach, what proportion of our generated arrays are distinct and feasible?
2. Quality: Are the generated arrays “well distributed” across the domain of possible auxiliary arrays?
3. Speed: How fast can we generate new distinct feasible auxiliary arrays?

Metrics for Quantity, Quality, and Speed  Measuring the quantity and speed of our generative models was relatively straightforward. Quantity is evaluated by using genIsingData.py to count the number of feasible auxiliary values that were generated. These were then used to compute the proportion of feasible auxiliaries.

Speed is measured by recording the wall clock time required to generate a given number of feasible auxiliary arrays on a specified number of processors. For our best previous approaches, the median time to generate one feasible auxiliary arrays of MUL_3x3x3 is 47 minutes on 192 processors (see Appendix D).

Quality, a measure of the “spread” of the generated auxiliary arrays across the sample space, is less straightforward. We use Hamming distance, the number of indices at which two arrays differ, to quantify the distance between generated auxiliary arrays. Given a list of randomly generated arrays where each element is either $-1$ or $+1$, each of length $n$, the distribution of Hamming distances for each pair of arrays in the list corresponds to the binomial distribution $B\left(\frac{n}{2}, \frac{1}{2}\right)$. In general, we do not expect that the distribution of pairwise Hamming distances for feasible auxiliary arrays will be exactly binomial, but by comparing this distribution to the binomial distribution $B\left(\frac{n}{2}, \frac{1}{2}\right)$, we get a measure of how dissimilar the generated arrays are. We quantify this comparison using the Jensen-Shannon distance between (a) the distribution of pairwise Hamming distances of our generated feasible auxiliary arrays and (b) the binomial distribution. The Jensen-Shannon distance quantifies the difference between two discrete probability distributions. It is a value between zero, meaning the two distributions are exactly identical, and one, meaning the two distributions have no overlap. It has the desirable property that it is symmetric, meaning that its value does not change with the order in which the probability distributions are compared.

3.1 Generative Adversarial Networks (GANs)

One of the most common generative models is a Generative Adversarial Network, or GAN, which attempts to generate fake data that is hopefully indistinguishable from real data. A typical GAN architecture consists of two models, a Generator and a Discriminator. The Discriminator’s goal is to be able to correctly classify fake data and real data. The Generator’s goal, in turn, is to generate data that so closely resembles real data that the Discriminator cannot distinguish between real data and fake/generated data. The two models are trained in parallel, with the Generator trying to maximize the loss function of the Discriminator and the Discriminator trying to minimize its loss function. Most commonly, researchers use GANs to produce images. Since their advent in 2014, GAN-generated images have improved significantly. For example, one common use case is to train a GAN on thousands of images of real human faces and have it generate fake, yet seemingly real, human faces [10]. In 2014, fake human faces were clearly fake. Nowadays, the results are quite astounding, and it is nearly impossible for a human to distinguish a generated face from a real one.

3.1.1 Vanilla GAN

Adaptation to our Problem  Our problem is different than the typical GAN use case. For us, we have feasible auxiliary arrays and infeasible auxiliary arrays. Here, the notion of real vs. fake does not really make any sense. For each problem, there are a finite number of auxiliary arrays; the question is finding
them. Unlike human faces or dogs, whose state spaces are infinite and who both appear naturally on Earth, any auxiliary array must be created by some process or another; we are not looking to determine the process that created an array, rather we are attempting to determine the feasibility of the resulting array.

A GAN intended to generate images of dogs would be trained on images of dogs. It will not see any images of cats. We are not interested in generating any auxiliary arrays, but only feasible auxiliary arrays. Thus, feasible auxiliary arrays correspond to the category of, say, images of dogs, and the infeasible auxiliary arrays correspond to an entirely different category of images, such as images of cats. Accordingly, we train our Vanilla GAN on a dataset consisting of only feasible auxiliary arrays. The Generator, then, attempts to generate more feasible arrays. Our Discriminator is more similar to a typical Discriminator in that it still attempts to distinguish feasible arrays out of the original dataset from arrays generated by the Generator. The Generator will, of course, initially have a tendency to generate infeasible arrays since they are magnitudes more likely. As training progresses, though, in order to fool the Discriminator, the Generator will slowly gravitate towards producing arrays that more closely resemble the feasible arrays from the dataset.

**Implementation** We implement our Reverse Ising GAN using PyTorch, beginning with a basic GAN template given in [3]. Notably, we define two classes, `Generator` and `Discriminator`, that inherit from PyTorch’s `nn.Module` class and represent the Generator and Discriminator, respectively. In these classes, we define the neural networks with PyTorch’s `nn.Sequential()` method, allowing us to stitch together linear layers, activation functions, and other model components. We define `d_loss_function` and `gen_loss_function` as the loss functions of the Discriminator and Generator, respectively (e.g. `nn.BCELoss` for Binary Cross Entropy loss). We test many combinations of model architectures, loss functions, and hyperparameters on both `MUL 2x3x1` and `MUL 3x3x3`, but our training strategy remains consistent. Below, we give a brief summary of one epoch of our main training loop:

1. **Train the Discriminator**
   - (a) **Sample from latent space**: We use Python’s random module to generate a random list of size `batch_size x latent_space_size` (where each element is a −1 or 1) to mimic sampling from the `latent space`, which can be thought of here as the set of all possible auxiliary arrays.
     - i. We then convert this list to `batch_size` tensors, each of length `latent_space_size`
   - (b) **Sample from training data**: Select one batch of training data from the `dataLoader` instance containing all data
   - (c) **Pass latent space samples to generator**: Pass all latent space samples to an instance of the `Generator` class, which outputs `batch_size` generated samples (candidate auxiliary arrays). Note that the Generator’s raw output is real-numbered arrays, which we must convert to -1s and 1s after training when we attempt to generate feasible arrays.
   - (d) **Observe and collect Discriminator’s thoughts on generated and real data as labels**: Pass all generated samples and real samples to an instance of the `Discriminator` class, storing results in `output_discriminator_fake` and `output_discriminator_real`, respectively.
   - (e) **Calculate Discriminator loss**: Pass Discriminator’s labels and actual labels to `d_loss_function`.
   - (f) **Update Discriminator network and optimizer**: call the `backward` and `step` methods for the network and optimizer, respectively.

2. **Train the Generator**
   - (a) **Sample from latent space**: Refer to 1a above.
   - (b) **Pass latent space samples to Generator**: Refer to 1c above.
   - (c) **Observe and collect Discriminator’s thoughts on generated data as labels**: Pass all generated samples and real samples to an instance of the `Discriminator` class, storing results in `output_discriminator_generated`.
   - (d) **Calculate Generator loss**: Pass Discriminator’s labels (`output_discriminator_generated`) and a tensor of all 1s, to denote real data (what the generator wants to be generating), to `gen_loss_function`
**MUL 2x3x1 Results** We develop models that perform very well on the MUL 2x3x1 problem. We begin testing models with Generator and Discriminator neural networks that each have 3 hidden layers of sizes 256, 128, and 64. We follow relatively standard practices by using the Rectified Linear Unit function (nn.ReLU()) as our activation functions between the hidden layers in the two neural networks. For the networks’ output layers, we use no activation function for the Generator and the Sigmoid (nn.Sigmoid()) function for the Discriminator, since its output represents a probability—the probability that a sample is real. Following common practice, we introduce noise at each layer of the Discriminator with PyTorch’s nn.Dropout(p), which zeroes an element in the input tensor with probability p; here, we use $p = 0.3$. Listings 1 and 2 below show the Generator and Discriminator classes for this architecture, which we refer to as Architecture 1.

```
0  ... 
1  class Generator(nn.Module):
2  def __init__(self):
3      super().__init__()
4      self.model = nn.Sequential(
5          nn.Linear(32, 256),
6          nn.ReLU(),
7          nn.Linear(256, 128),
8          nn.ReLU(),
9          nn.Linear(128, 64),
10         nn.ReLU(),
11         nn.Linear(64, 32),
12         )
13  def forward(self, x):
14      output = self.model(x)
15      return output
Listing 1: Generator Architecture 1

With our Generator and Discriminator defined as above, we test several combinations of hyperparameters, namely the loss functions, learning rate, and batch size. In Table 4, we present some of these models along with their performance. We train all models for 500 epochs. For all of models, we then use the Generator to output 10k arrays; hence, the “Total Feasible” column is the total number of feasible arrays out of the 10k generated.

<table>
<thead>
<tr>
<th>Generator Loss Function</th>
<th>Learning Rate</th>
<th>Batch Size</th>
<th>Total Feasible</th>
<th>Distinct Feasible</th>
<th>Distinct Feasible /Total Feasible</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 BCELoss</td>
<td>0.001</td>
<td>5</td>
<td>8,367</td>
<td>71</td>
<td>0.0085</td>
</tr>
<tr>
<td>2 BCELoss</td>
<td>0.001</td>
<td>20</td>
<td>8,849</td>
<td>59</td>
<td>0.0067</td>
</tr>
<tr>
<td>3 BCELoss</td>
<td>0.001</td>
<td>40</td>
<td>4,766</td>
<td>78</td>
<td>0.0164</td>
</tr>
<tr>
<td>4 BCELoss</td>
<td>0.001</td>
<td>100</td>
<td>4,986</td>
<td>28</td>
<td>0.0056</td>
</tr>
<tr>
<td>5 BCELoss</td>
<td>0.0001</td>
<td>5</td>
<td>9,624</td>
<td>341</td>
<td>0.0354</td>
</tr>
<tr>
<td>6 BCELoss</td>
<td>0.0001</td>
<td>20</td>
<td>5,923</td>
<td>118</td>
<td>0.003</td>
</tr>
<tr>
<td>7 BCELoss</td>
<td>0.0001</td>
<td>100</td>
<td>2,532</td>
<td>89</td>
<td>0.0352</td>
</tr>
<tr>
<td>8 BCELoss</td>
<td>0.0001</td>
<td>123</td>
<td>114</td>
<td>116</td>
<td>0.94</td>
</tr>
</tbody>
</table>

Table 4: Results of Network Architecture 1 Using BCE Loss for Various Hyperparameter Values

1 BCE = Binary Cross Entropy

As seen in Table 4, we notice a relatively small proportion of the feasible arrays that our models generate are actually distinct. This is a phenomenon known as mode collapse, in which a generative model learns to output only a small number of samples (a few outputs are repeatedly generated). Mode collapse is common when the dataset follows a multimodal distributions because the Generator will learn that it can fool the
Discriminator with data from a single mode. The Discriminator will begin classifying everything from that mode as fake, at which point the Generator will switch to a different mode. This cycle will repeat until the training ends. A common way to defeat mode collapse is feature matching, which modifies the Generator’s loss function to account for and encourage diversity of the generated batches. Specifically, feature matching uses Mean Squared Error (MSE) as the Generator’s loss function. Thus, we train models nearly identical to the ones in Table 4, with the one exception being we use MSE as our loss function for the Generator.

<table>
<thead>
<tr>
<th>Generator Loss Function</th>
<th>Learning Rate</th>
<th>Batch Size</th>
<th>Total Feasible</th>
<th>Distinct Feasible (/10k)</th>
<th>Distinct Feasible /Total Feasible</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MSE Loss</td>
<td>0.001</td>
<td>5</td>
<td>252</td>
<td>35</td>
</tr>
<tr>
<td>2</td>
<td>MSE Loss</td>
<td>0.001</td>
<td>20</td>
<td>8,320</td>
<td>1,131</td>
</tr>
<tr>
<td>3</td>
<td>MSE Loss</td>
<td>0.001</td>
<td>40</td>
<td>8,586</td>
<td>2,822</td>
</tr>
<tr>
<td>4</td>
<td>MSE Loss</td>
<td>0.001</td>
<td>100</td>
<td>7,649</td>
<td>1,764</td>
</tr>
<tr>
<td>5</td>
<td>MSE Loss</td>
<td>0.0001</td>
<td>5</td>
<td>9,611</td>
<td>217</td>
</tr>
<tr>
<td>6</td>
<td>MSE Loss</td>
<td>0.0001</td>
<td>20</td>
<td>5,792</td>
<td>152</td>
</tr>
<tr>
<td>7</td>
<td>MSE Loss</td>
<td>0.0001</td>
<td>40</td>
<td>6,477</td>
<td>323</td>
</tr>
<tr>
<td>8</td>
<td>MSE Loss</td>
<td>0.0001</td>
<td>100</td>
<td>102</td>
<td>96</td>
</tr>
</tbody>
</table>

Table 5: Results of Network Architecture 1 Models Using MSE Loss for the Generator for Various Hyperparameter Values

As shown in Table 5, using the MSE loss function for the Generator awards us a much higher proportion of distinct feasible arrays. We further investigate the quality of the generated arrays by looking at the distribution of all their pairwise Hamming distances. Shown in Figure 2 are the distributions of the Hamming distances for Models 1-4 in Table 5. The distribution of Hamming distances is fairly decent. As batch size increases, the distribution becomes better—perhaps because the Generator sees a greater number of auxiliary arrays for each update to its parameters.
For all of our models, we record the losses of the Generator and Discriminator during training. In Figure 3 we show the losses for all models that use MSE Loss for the Generator and have a learning rate of .001 (Models 1-4 in Table 5). Loss is always nonnegative, lower values (closer to zero) are better. Of note is that we observe an increase in the loss of the Generator during training; thus, we would expect our models to be performing terribly, but they are not. This phenomena is partially a consequence of the adversarial design of the GAN. In particular, the performance of the Generator and the performance of the Discriminator are unbalanced – the Discriminator is performing significantly better than the Generator. The high accuracy of the Discriminator lead to a low Discriminator loss (it is achieving its goal of correctly predicting the output of the Generator as “fake”). This means that the Generator is unable to achieve its goal of “fooling” the Discriminator, and thus the loss of the Discriminator is higher. A balanced GAN would have both losses stabilizing around a value of .5 – which has the interpretation that the Generator is able to “fool” the Discriminator 50% of the time, which hopefully translates into the Generator producing output that “looks” real, or in our case, output that is indeed feasible.

In Figure 4 we see that our Discriminator is indeed making very accurate classifications. “D Accuracy Real” represents its accuracy on real samples. As mentioned in Step 1c. of Paragraph 3.1.1 the Generator outputs arrays of real numbers, and we have to convert these numbers to -1s and 1s to get our true candidate auxiliary arrays. During training, we update our networks based on the Discriminator’s predictions on the real-valued, or raw arrays; thus, “D Accuracy Fake” represents the Discriminator’s accuracy on generated, or fake, samples. For analysis purposes, we also convert these arrays to -1’s and 1’s and pass these converted...
samples to the Discriminator to view its accuracy on these. We do not update our networks based on these predictions, though. As indicated by “D Accuracy Converted” in Figure 4, the Discriminator is consistently much worse at classifying these converted arrays. In fact, its accuracy on these samples is much closer to the accuracy in most GANs (i.e., an accuracy of around .5). It would seem, then, that updating the networks (backpropogating) based on the predictions of the converted samples would yield a better Generator loss. While this may be the case, we have trained models that learn in this manner, and they have produced significantly worse results. The strong performance of our GANs (the original GANs trained on the raw arrays) is further supported by analyzing both the proportion of generated arrays that are feasible by epoch and the proportion of feasible arrays that are distinct (Figures 5 and 6, respectively). On this note, we accept our strangely increasing Generator loss. The exact reasons behind why the Generator still produces a high proportion of feasible, and distinct feasible, arrays, in spite of its high loss, is unknown and left to future investigation.

Figure 3: Generator and Discriminator Losses for GAN Models 1-4 in Table 5
Figure 4: Discriminator Accuracy for GAN Models 2, 3, 4, 6, 7, and 8 in Table 5

Figure 5: Proportion of Feasible Arrays (out of 10k Generated) by Epoch for all GAN Models in Tables 4 and 5
While it is very likely that different combinations of model architecture, loss functions, hyperparameters, and other architectural qualities could lead to even a higher proportion of distinct feasible arrays, we were pleased with the performance of the GAN at this point and decided to spend our time and energy on the much harder \textit{MUL 3x3x3} problem.

\textbf{MUL 3x3x3 Results} Confident that a more complex problem would likely require larger neural networks, we increased the size of our Generator and Discriminator networks so that they each have three hidden layers of sizes 1024, 512, and 256, respectively. With this new network architecture, we tried many combinations of learning rate, loss function, and number of epochs. We even tweaked the network sizes in some attempts. All of these approaches were \textit{simple}, in the sense that they were not necessarily employing any new techniques (when compared to our approaches on the \textit{MUL 2x3x1}). None of these simple approaches yielded any discovered feasible arrays. After consistent failures, we decided to create and train more advanced models. In all of these more advanced models, we mix and match a number of GAN training techniques. We summarize our employed GAN techniques in Table 6 and our models that employ some subset of these techniques in Table 7. Note that not all of the techniques listed in Table 6 are used in the models listed in Table 7. This is because we actually tested some of these techniques on the \textit{MUL 2x3x1} problem and saw no improvements; hence, we did not believe these techniques would show promise on \textit{MUL 3x3x3}. 

Figure 6: Proportion of Distinct Feasible Arrays (out of Total Feasible) by Epoch for all GAN Models in Tables 4 and 5
<table>
<thead>
<tr>
<th>Technique</th>
<th>Description</th>
<th>Intention</th>
</tr>
</thead>
<tbody>
<tr>
<td>D. Wasserstein Loss</td>
<td>D. acts as a critic that scores the realness of a sample. We use the Wasserstein Loss function for D.</td>
<td>Minimize the distance between observed training set data and observed generated data.</td>
</tr>
<tr>
<td>Label Smoothing (Real or Fake)</td>
<td>For real label smoothing, we use a label &lt;1 for for the real samples (typically .9). For fake label smoothing, we use a label &gt;0 for the fake samples.</td>
<td>Discourage overconfidence in the Discriminator by penalizing for predictions that are too low or too high.</td>
</tr>
<tr>
<td>D. Leaky ReLU Activation</td>
<td>Use Leaky ReLU as the activation function between each hidden layer in D.</td>
<td>Fixes the &quot;Dying ReLU&quot; problem by assigning a small negative value for inputs &lt;0, rather than 0.</td>
</tr>
<tr>
<td>G. Dropout</td>
<td>Use dropout in between layers of G.</td>
<td>Avoid overfitting by G.</td>
</tr>
<tr>
<td>G. Random Normal Input</td>
<td>Sample from a normal distribution with mean 0 and variance 1 to generate input arrays.</td>
<td>Enable better output diversity</td>
</tr>
<tr>
<td>Minibatch Discrimination</td>
<td>Input real data and generated data into D. separately, or in different batches, and calculate similarity within batches.</td>
<td>Avoid mode collapse</td>
</tr>
<tr>
<td>Tanh for G. Output Layer Activation</td>
<td>Use the Tanh activation function for the output layer of G. in order to scale all outputs between -1 and 1.</td>
<td>Output values closer to those in actual auxiliary arrays ({-1, 1})</td>
</tr>
</tbody>
</table>

Table 6: GAN training techniques

1 D. = Discriminator
2 G. = Generator
3 Leaky ReLU = Leaky rectified linear unit

We had little to no luck using a Vanilla GAN to generate feasible arrays for MUL 3x3x3, as shown in Table 7.

3.1.2 Conditional Generative Adversarial Network (cGAN)

Motivation GANs offer very little control over what kind of data the generator will generate. The conditional Generative Adversarial Network (cGAN) attempts to rectify this by providing labels to both the discriminator and generator, allowing us to specify what “class” or “category” of data the generator should produce.

For example, in a GAN that is trained to generate and classify images of cats and dogs, the generator only knows that if it produces an image that resembles a cat or a dog, it will fool the discriminator. However, neither neural network can recognize the difference between the two categories. As a result, we have no control over whether the generator is going to output an image of a cat or an image of a dog – if we wanted to generate an image of a cat, we would have to continuously generate images until we receive a picture of a cat. This is a lot of extraneous data and can take a lot of extra time.

Intuitively, this is not a huge problem for the GAN in our problem since the generator’s attempt to output data that mimics the training data should, ideally, mean that it is always trying to generate feasible arrays (since the training data contains only feasible auxiliary arrays). However, an additional potential benefit of the cGAN is that the discriminator is provided with a higher variety of data. By allowing it to see data from both categories - feasible and infeasible - we were hopeful that the discriminator would be able to better distinguish between real and generated data. The generator also inherits an additional metric with which to evaluate its performance: how well it can transform input noise into auxiliary arrays that fit the labels that are passed to it.

Figure 7 displays a typical training and testing flow for both the vanilla GAN and the conditional GAN. Note that the goals of the two neural networks is different in the cGAN than in the vanilla GAN. The
<table>
<thead>
<tr>
<th>Model</th>
<th>G. Network</th>
<th>D. Network</th>
<th>G. Loss Function</th>
<th>Advanced Techniques</th>
<th>Batch Size</th>
<th>Learning Rate</th>
<th>Epochs</th>
<th>Feasible (/10k)</th>
</tr>
</thead>
</table>
| 1     | 1024, 800, 512, 256 | 1024, 500 | BCELoss | - Real label smoothing = .9  
- D. batch norm  
- D. Leaky ReLU  
- G. dropout = .3  
- G. input size = 100 | 20 | .001 | 800 | 0 |
| 2     | 1536, 768, 384 | 1536, 768, 384 | BCELoss | - D. Wasserstein loss  
- D. Leaky ReLU | 20 | .001 | 800 | 0 |
| 3     | 1536, 768, 384 | 1536, 768, 384 | BCELoss | - D. Leaky ReLU | 20 | .001 | 800 | 0 |
| 4     | 384, 768, 1536 | 1536, 768, 384 | MSELoss | - D. Leaky ReLU | 20 | .001 | 800 | 1 |
| 5     | 1024, 800, 512, 256 | 1024, 500 | MSELoss | - Real label smoothing = .9  
- D. batch norm  
- D. Leaky ReLU  
- G. dropout = .3  
- G. input size = 100 | 20 | .001 | 800 | 0 |
| 6     | 1024, 800, 512, 256 | 1024, 500 | MSELoss | - Real label smoothing = .9  
- D. batch norm  
- D. Leaky ReLU  
- G. dropout = .3  
- G. input size = 100 | 20 | .00001 | 800 | 0 |
| 7     | 1024, 800, 512, 256 | 1024, 500 | MSELoss | - Real label smoothing = .9  
- D. batch norm  
- D. Leaky ReLU  
- G. input size = 100 | 100 | .00001 | 800 | 0 |
| 8     | 1024, 800, 512, 256 | 1024, 500 | MSELoss | - Real label smoothing = .9  
- D. batch norm  
- D. Leaky ReLU  
- G. input size = 100 | 250 | .00001 | 800 | 0 |
| 9     | 1024, 800, 512, 256 | 1024, 500 | MSELoss | - Real label smoothing = .7  
- D. batch norm  
- D. Leaky ReLU  
- G. dropout = .3  
- G. input size = 100 | 250 | .00001 | 800 | 0 |
| 10    | 384, 768, 1536 | 1536, 768, 384 | MSELoss | - D. Leaky ReLU | 20 | .0001 | 800 | 0 |

Table 7: **MUL 3x3x3 Vanilla GAN model results**

D. = Discriminator  
G. = Generator  
Leaky ReLU = Leaky rectified linear unit
generator’s goal is to generate data that both fools the discriminator and matches the specified label, while the discriminator’s goal is to correctly classify whether the data is received is both “real” (from the training dataset) and whether it matches the specified label.

![Figure 7: Training flow for the GAN vs. the cGAN for the Ising problem](image)

**Initial Promise**  We first trained and deployed a cGAN on the MUL 2x3x1 problem that shared its discriminator architecture with the best GAN model at the time, which consisted of:

- A discriminator with 5 hidden layers of size 256, 128, 100, 64, and 32, and
- A generator with 2 hidden layers of size 16 and 32.

Out of 10,000 generated auxiliary arrays, 1,308 were distinct and feasible, as confirmed by genIsingData.py. The proportion of distinct feasible auxiliary arrays was higher for the cGAN than for the GAN, although the exact factor is unknown.

The next goal was to improve the proportion of distinct feasible auxiliary arrays generated for MUL 2x3x1 before attempting a larger problem.

**Experimenting with the cGAN**  GANs and conditional GANS are notoriously difficult to stabilize. The structure and choice of hyperparameters are very important, yet difficult to pin down. If incorrectly built, losses for the battling neural networks can explode quickly or fail to converge to a low number, which is typically an indicator that the model is not learning aptly. For a lot of tests, one or both of these occurred with the cGAN generator, while the discriminator loss remained low.
The first attempt to improve the cGAN involved changing the hidden layer architectures for the discriminator and generator. The table below details the architectures we tested.

<table>
<thead>
<tr>
<th>Discriminator Architecture</th>
<th>Generator Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>256, 128, 100, 64, 32</td>
<td>256, 128, 100, 64, 32</td>
</tr>
<tr>
<td>256, 128, 100, 64, 32</td>
<td>16, 32, 64</td>
</tr>
<tr>
<td>256, 64, 32</td>
<td>32, 16</td>
</tr>
<tr>
<td>128, 64, 32</td>
<td>16, 32</td>
</tr>
<tr>
<td>64, 32</td>
<td>16, 32</td>
</tr>
<tr>
<td>256, 64</td>
<td>16, 32</td>
</tr>
<tr>
<td>256, 64, 32</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 8: Attempted cGAN MUL 2x3x1 Hidden Layer Architectures

Models were trained for between 300 and 700 epochs with a learning rate of 0.001 and using the Binary Cross-Entropy loss function. Each model yielded zero or very few distinct feasible auxiliary arrays.

Numerous other techniques were implemented, with no success. The following table summarizes these techniques.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Description</th>
<th>Intention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature Matching</td>
<td>Changing loss function from BCE to MSE. Successful for vanilla GAN.</td>
<td>Combat mode collapse and stabilize loss</td>
</tr>
<tr>
<td>Changing Activation Function</td>
<td>Switching activation function for discriminator and generator from ReLU to LeakyReLU</td>
<td>Fixes the “Dying ReLU” problem by assigning a small negative value for inputs &lt;0, rather than 0</td>
</tr>
<tr>
<td>Changing Range of Generator Output</td>
<td>Convert generator output from real numbers to integers</td>
<td>Prevent discriminator from easily classifying based on type difference</td>
</tr>
<tr>
<td>Modifying Batch Structure</td>
<td>Pass real and generated samples to the discriminator simultaneously</td>
<td>Avoid easy discriminator classification based on order</td>
</tr>
<tr>
<td>Decreasing Generator Learning Rate</td>
<td>Decrease generator learning rate from its initial value of 0.001</td>
<td>Prevent bouncing losses</td>
</tr>
<tr>
<td>Label Smoothing</td>
<td>Cap discriminator’s output at 0.9 instead of 1.0</td>
<td>Reduce discriminator “confidence” in its ability to classify</td>
</tr>
<tr>
<td>Training Generator in Sub-Loop</td>
<td>Update generator weights 3x as often as discriminator</td>
<td>Give generator an opportunity to catch up to discriminator</td>
</tr>
<tr>
<td>Adding More Features</td>
<td>Add option to append pairwise, elementwise products of neighboring features</td>
<td>Allow discriminator to extract patterns more easily</td>
</tr>
<tr>
<td>Changing Range of Generator Input</td>
<td>Pass real numbers as noise to generator rather than -1s and 1s</td>
<td>Met with some success for Deep Convolutional GAN (DCGAN)</td>
</tr>
<tr>
<td>Adding Batch Normalization to Discriminator</td>
<td>Add BatchNorm after each hidden layer of discriminator</td>
<td>Standardize input, combat mode collapse</td>
</tr>
<tr>
<td>Adding Experience Replay</td>
<td>Allow discriminator to see old generated samples</td>
<td>Lessen effects of mode collapse</td>
</tr>
<tr>
<td>Testing Lower and Higher Batch Sizes</td>
<td>Increase and decrease batch size</td>
<td>Change how often discriminator and generator update weights</td>
</tr>
<tr>
<td>Changing Size of Generator Input</td>
<td>Pass smaller arrays of noise to generator</td>
<td>Fine-tune generator’s ability to pick out patterns</td>
</tr>
</tbody>
</table>

Table 9: cGAN training techniques

**Improved cGAN Models on MUL 2x3x1** After much experimentation, two strategies were found to improve the performance of original best cGAN model. The first strategy involved scaling down the archi-
architecture, specifically removing two of the central hidden layers from the discriminator. The cGAN performed the best with the following architecture:

- A discriminator with 3 hidden layers of size 256, 64, and 32
- A generator with 2 hidden layers of size 16 and 32

For 10,000 generated arrays, this model produced 2,329 distinct feasible auxiliary arrays - an improvement by almost a factor of 2.

The second strategy that improved the performance of the cGAN involves replacing the training dataset. All the feasible auxiliary arrays that have been generated by the cGAN over all previous experiments were collected together into a dataset. This dataset contains no arrays from the original training dataset. We trained the new better (smaller) architecture on this new training dataset. Recall that one of the motivations for looking at the cGAN was that it allowed the discriminator to see both feasible and infeasible auxiliary arrays in the training dataset. This removes that possibility: the training dataset contains only feasible auxiliary arrays.

We performed a few iterations where we trained and tested the cGAN (keeping the architecture constant), then we augmented the dataset with new feasible auxiliary arrays that were generated, and then we trained a new model on this augmented dataset. The resulting model generated more feasible, and more distinct feasible, auxiliary arrays - generating 50% more distinct feasible arrays! Testing the model on 10,000 arrays, 3,683 of the generated outputs were distinct and feasible.

Curiously, when the size of the replacement training dataset grew larger than 20,000 samples, the performance of the cGAN trained on the dataset only improved marginally – the proportion of distinct feasible arrays generated stabilized.

**Results of Improved cGAN Models on MUL 2x3x1** We conducted 3 trials for each model where the generator would produce 1,000, 10,000, and 20,000 arrays. The following tables summarize the progress made from the original best model to the current best model with the improvements mentioned in the above section.

<table>
<thead>
<tr>
<th>Trial No.</th>
<th>Arrays Generated</th>
<th>Num. Feasible</th>
<th>Num. Distinct Feasible</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,000</td>
<td>843</td>
<td>409</td>
</tr>
<tr>
<td>2</td>
<td>10,000</td>
<td>8,441</td>
<td>1,308</td>
</tr>
<tr>
<td>3</td>
<td>20,000</td>
<td>16,766</td>
<td>1,665</td>
</tr>
</tbody>
</table>

Table 10: Results of Original Best cGAN Model

D: 256, 128, 100, 64, 32, G: 16, 32

<table>
<thead>
<tr>
<th>Trial No.</th>
<th>Arrays Generated</th>
<th>Num. Feasible</th>
<th>Num. Distinct Feasible</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,000</td>
<td>891</td>
<td>597</td>
</tr>
<tr>
<td>2</td>
<td>10,000</td>
<td>8,939</td>
<td>2,329</td>
</tr>
<tr>
<td>3</td>
<td>20,000</td>
<td>17,778</td>
<td>3,084</td>
</tr>
</tbody>
</table>

Table 11: Results of Model with Improved Architecture on Original Training Data

D: 256, 64, 32, G: 16, 32
Table 12: Results of Model with Improved Architecture on new Feasible Training Dataset

<table>
<thead>
<tr>
<th>Trial No.</th>
<th>Arrays Generated</th>
<th>Num. Feasible</th>
<th>Num. Distinct Feasible</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,000</td>
<td>1,000</td>
<td>826</td>
</tr>
<tr>
<td>2</td>
<td>10,000</td>
<td>10,000</td>
<td>3,683</td>
</tr>
<tr>
<td>3</td>
<td>20,000</td>
<td>19,999</td>
<td>4,846</td>
</tr>
</tbody>
</table>

There is without a doubt significant improvements made as a result of all this experimentation and extensive hyperparameter tuning – the number of distinct feasible arrays more than doubled from our original best model to our final best model! Further, although a large proportion of the feasible auxiliary arrays are still duplicates, the model outputs very few, if any, infeasible auxiliary arrays. Also, observe that the number of distinct feasible auxiliary arrays does not scale linearly with the number of outputs generated.

It is worth mentioning that the final cGANs were trained on only feasible data. Despite our intentions, including infeasible arrays in the training data did not provide more information to the neural networks. Thus, the cGAN and the vanilla GAN have the same type of training data (feasible auxiliary arrays). The cGAN has an additional label that is passed into both the generator and the discriminator, but this label is always specified to be “feasible.” Thus, the two models seem to be basically identical. However, the cGAN performs better than the GAN, as measured by the number of distinct feasible auxiliary arrays generated out of 10,000 (the best for the cGAN is 3,683 whereas that number is 2,822 for the GAN). It is unclear if there is a theoretical explanation for this difference in performance (also note that the model architectures differ between the GAN and the cGAN, although similar ranges of architectures were implemented for both).

Data Quality  Training on the new dataset of distinct feasible auxiliary arrays not only improved the quantity of distinct feasible auxiliary arrays that were produced, it also improved the quality of that data, as shown in the following figures.

In the above figures, we can see the distribution of pairwise Hamming distances spreading toward larger values when using the new training dataset (containing the generated feasible auxiliary arrays as training data, but none of the original training data). The data generated by the model that was trained on the original training dataset did not have any outputs that had a pairwise hamming distance of 20 or larger, whereas more than 20% of the outputs generated by the model trained on the new training dataset have a pairwise hamming distance of 20 or larger.
Although the exact reasons for the improvements on the quality of the data that was generated as a result of replacing the training dataset are not clear, one possibility is that the original dataset did not contain enough “good quality” data to be able to generate a lot of distinct feasible arrays.

**Scaling Up**  With satisfactory results on MUL 2x3x1, we decided it was time to scale up to the larger MUL 3x3x3 problem. The following architectures yielded no distinct feasible auxiliary arrays:

<table>
<thead>
<tr>
<th>Discriminator Architecture</th>
<th>Generator Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>256, 64, 32</td>
<td>16, 32</td>
</tr>
<tr>
<td>256, 512, 1,024</td>
<td>64, 128</td>
</tr>
<tr>
<td>256, 64, 32</td>
<td>64, 128</td>
</tr>
<tr>
<td>128, 64, 32</td>
<td>256, 64, 32</td>
</tr>
<tr>
<td>64, 32</td>
<td>16, 32</td>
</tr>
</tbody>
</table>

Table 13: Attempted cGAN MUL 3x3x3 Hidden Layer Architectures

Each model took 3-7 days to complete the training and testing process on CPUs, so very few architectures were attempted due to time constraints. There is not currently a cGAN model that has produced any distinct feasible auxiliary arrays for MUL 3x3x3. This is an area for future investigation.

### 3.1.3 Other GANs

**Cycle GAN**  *CycleGANs* are a variation of GANs that convert data of one class to data of another class. Like Vanilla GANs, they are most frequently used for image to image translation. A few examples of CycleGANs are converting images of horses to images of zebras, converting real photographs to paintings by certain artists, and even converting faces to Ramen noodles. We adapted template PyTorch CycleGAN code for converting zebra images to horse images (and vice versa) to our problem, with the goal being to convert infeasible arrays to feasible arrays. We trained a CycleGAN model for 60 epochs, a learning rate of .0001, and a batch size of 4. We attempted to convert 3,750 testing infeasible MUL 2x3x1 auxiliary arrays into feasible arrays. Unfortunately, none of the resulting arrays were actually feasible.

**Unrolled GAN**  Unrolled GANs are yet another GAN variation that allow lookahead for the Generator. In an Unrolled GAN, the Generator *unrolls* $k$ steps forward to see how the Discriminator will perform. Unrolled GANs are a common technique for defeating mode collapse, as they reduce the likelihood of a the Generator overfitting one version of the Discriminator. As we do for many of our other GAN implementations, we adapt some template code to our own use case. We train an Unrolled GAN but noticed that our Generator and Discriminator losses went to 0 after less than 400 iterations. We were unable to fix this issue. As such, we were unable to use an Unrolled GAN to attempt to generate feasible arrays.

**Hamming Loss GAN**  With the intention of increasing the Hamming distances of generated arrays, we develop what we call a *Hamming Loss GAN*. Our Hamming Loss GAN adds an additional loss component to the Generator’s loss: 1 minus the average pair-wise Hamming distance of the generated auxiliary arrays. We train a Hamming Loss GAN that is nearly
identical to Model 3 in Table 5, with the only exception being that the total loss function for this model is equal to the sum of MSE and the Hamming Loss. Out of 10k generated samples, this model produces 7,309 feasible arrays, 3,621 of which are distinct—a much higher proportion of than its Model 3 counterpart. As shown in Figure 10, the quality of these arrays is also much better, with a higher mean Hamming distances and a lower Jensen-Shannon Distance.

**Deep Convolutional GAN (DCGAN)** The most successful GAN variation that we tried is the DCGAN. DCGANs are quite similar to Vanilla GANs, with the main difference being that they use deep convolutional layers for the Generator. Unlike our other GAN variations, our trained DCGAN models produced many distinct feasible arrays on the MUL 3x3x3 problem. However, the Hamming distance distributions were not what we would have liked to see, with the models generating clusters of arrays with small Hamming distances. This is an area for future investigation.

### 3.2 Variational Autoencoders (VAEs)

Variational Autoencoders, along with GANs (3.1), are among the most popular generative models. The variational autoencoder attempts to generate data that is similar, but not identical to, the data that it is trained on. As with GANs, VAEs are popular in image generation. This application seems logical: a VAE trained on images of cats will be attempting to output new images of cats that are similar, but not identical, to images of cats that it has already seen.

#### 3.2.1 Vanilla Variational Autoencoders

**Background: Autoencoders** An autoencoder (is a type of neural network based that is designed for unsupervised learning – finding patterns in data that is not labeled. The models encodes the (unlabeled) input data into a lower dimensional representation, and then attempts to decode this representation and reproduce the original data. The hope is that in this process of compressing and then decompressing the data, the model will learn the “essential characteristics” of the data and map these essential characteristics into the encoded representation. In other words, the model will “learn” an effective compression strategy without any user input. Figure 11 depicts the structure of an autoencoder.

![Figure 11: A Depiction of an Autoencoder](image.png)

Some terminology will be necessary for the rest of this section. The encoded representation of an input is known as a latent variable. If we denote the input to the autoencoder by \( X \) and then denote the encoded representation by \( Z \), then \( Z \) is a latent variable (that corresponds to the input \( X \)). The latent space refers to the portion of the model that contains the latent variables (the “space” between the encoder and the decoder). Note that just as the input can be multidimensional, so can the latent space (the latent variables are arrays).

Additionally, the vanilla variational autoencoder will be referred to as the variational autoencoder throughout this section.
Variational Autoencoders  Although the autoencoder is designed for unsupervised learning, it has been adapted in various ways to perform supervised learning – finding patterns in data that is labeled. This is fortunate because we have labeled data, and hence are doing supervised learning!

There are an assortment of popular autoencoders which are suitable for classification tasks, such as sparse autoencoders and denoising autoencoders. However, in this work we primarily focused on the use of variational autoencoders (VAEs), which are categorized as generative models and several smaller variants, like the convolutional VAE and conditional VAE. VAEs differ from the majority of autoencoders in that they are generative models. VAEs are also theoretically motivated by varitional Bayesian methods.

While the architecture of a VAE is much like a standard autoencoder, there is a key difference that is found in the design of the latent space. The autoencoder encodes the input into a latent representation and then decodes that representation. In contrast, the VAE encodes the input into a latent representation, injects noise into this representation, and then decodes that (“noisy”) representation. It is hoped that this will enable the model to produce an output that is similar, but not identical, to the input. This is intuitive in image applications: in a VAE trained on images of cats, we would want it to produce new images of cats that are similar, but not identical, to the images of cats that it has already seen. The intuition is less clear for out problem. We want outputs (auxiliary arrays) that are similar to other feasible auxiliary arrays in the sense that the output is also feasible, but we want new feasible auxiliary arrays that have not been seen before. We hope that the model is able to learn how to structure the latent space in such a way that feasible auxiliary arrays are similar to one another. (This idea of encouraging structure in the latent space is known in the literature as regularization.)

In order to inject noise into the representation, the latent variable in the standard autoencoder is converted to a probability distribution. In order to ensure that the model can shift this distribution as it learns, the encoder returns two arrays (instead of the one array in the standard autoencoder). The first array is interpreted as the center, or mean (denoted \( \mu \)), of the probability distributions, the second array is interpreted as the spread, or standard deviation (denote \( \sigma \)), of the probability distributions.\(^4\) The \( \mu \)-array corresponds to the “encoded representation” from the standard autoencoder; the \( \sigma \)-array is the mechanism through which the model adds noise into this representation. For computational simplicity, every dimension of the encoded representation is modeled using a normal distribution (with mean \( \mu_i \) and standard deviation \( \sigma_i \), where \( \mu_i \) and \( \sigma_i \) are the \( i \)’th elements of the \( \mu \)- and \( \sigma \)- arrays) Thus, the latent space in the VAE is a multidimensional normal distribution. A sample is drawn from this latent space distribution, and then passed through the decoder. This structure is depicted in Figure 12.

![Figure 12: A Depiction of a Variational Autoencoder](image)

However, there is a fundamental implementation issue with the VAE as just described. The goal of most neural networks is to minimize the loss function, this is the case with the VAE. To achieve this, backpropagation is used, where the gradient of the loss function is taken. (The loss function used for the VAE is discussed briefly below). However, currently the decoder is drawing a sample from the latent space probability distribution to obtain an input. We want the backprogration to reflect in influence of the \( \mu \) and \( \sigma \) values, not the influence of the values of the noise inherent in the latent space. The reparameterization

\(^4\)In the actual implementation, this second array is actually a function of the standard deviation because the encoder neural network does not know that variance is nonnegative.
trick is a technique that enables this.

A sample is drawn independently from the rest of the model from the standard normal distribution. Let \( \zeta \) denote this sample, then \( \zeta \sim N(0, I) \) (where \( I \) is the identity matrix). Some Gaussian properties can be exploited to obtain the following equation:

\[
z = \mu + \sigma \zeta.
\]

This variable \( z \) is identical to the latent variable obtained by sampling from the normal distribution defined using \( \mu \) and \( \sigma \) (as described above). However, the sampling operation has been removed from the flow of data through the model and there is no randomness in this equation. This enables backprogation that is a function of the desired parameters. This update to the algorithm is depicted in figure 13.

![Figure 13: A reparameterized VAE model](image)

For completeness, we describe briefly the loss function of the VAE. The loss function is the sum of two terms. The first term is the reconstruction loss, used in the autoencoder, that measures the difference between the output and the input of the VAE. This loss term is typically a binary cross entropy/log loss function. The second term is Kullback-Liebler (or KL) loss between the latent space distribution and a standard normal distribution (with \( \mu = 0, \sigma = 1 \)) – a measure of the distance between two probability distributions. This term counters the desire of the reconstruction loss to exactly reconstruct the input, as well as enabling the model to “regularize” the latent space (i.e., encourage hopefully beneficial structure on the latent space). The presence of the KL term also means that the loss of the VAE may not reach zero, although, as in the standard autoencoder, the goal is still to minimize the loss.

More complete explanations for reparameterization, KL loss and VAEs as a whole are out of scope for this report. For a more comprehensive background, *Tutorial on Variational Autoencoders* by Carl Doresch [4] and *Deep Learning* by Ian Goodfellow [7] are recommended references.

**Adaptation**  
VAEs are typically used in media applications, such as mimicking audio and reconstructing realistic images. This goal is very similar to what GANs attempt to achieve, with differences mostly lying in the approach. In the case of this project, the training dataset of the VAE consists of feasible auxiliary arrays. After the inputs pass through the latent space and are reparameterized, the goal is to create feasible auxiliary arrays that are similar, yet not identical to the inputs to the encoder. Much like with GANs, the biggest issue is that the sample space is huge, and distinct feasible auxiliary arrays become rarer as the dimensions increase. Can the VAE discover new distinct feasible auxiliary arrays?

The algorithm to train our model is described below.

1. Feed in **batch size** vectors of size \( n \) (where \( n \) is the problem-dependent vector width)
2. Inputs pass through the neural network layers of the encoder
3. Obtain a latent space parameterization of the data, which is comprised of two arrays for the $\mu$ and $\sigma$ values.

4. Inject noise into the latent space by sampling from a normal distribution: $\zeta \sim N(0, I)$, reparameterize our final latent variable as $a = \mu + \sigma \zeta$

5. Pass $z$ through the neural network of the decoder to obtain the output

6. Calculate the loss and gradients. Backpropogate and update the model parameters.

Adapted from PyTorch’s official example of a VAE [8], the VAE class below in Listing 1 contains four main functions: the encoder, decoder, reparameterization and forward. The “forward” function is from PyTorch’s neural network class `nn.Module`, which is called every time the model is called in a given program. Variables $fc41$ and $fc42$ refer to the $\mu$ and $\sigma$ that are passed from the encoder, while $fc5$ is taking the reparameterization and passing the data to the decoder.

```python
0 class VAE(nn.Module):
1    def __init__(self, size):
2        super(VAE, self).__init__()
3        self.fc1 = nn.Linear(self.size, 256)
4        self.fc2 = nn.Linear(256, 128)
5        self.fc3 = nn.Linear(128, 64)
6        self.fc41 = nn.Linear(64, 32)
7        self.fc42 = nn.Linear(64, 32)
8        self.fc5 = nn.Linear(32, 64)
9        self.fc6 = nn.Linear(64, 128)
10       self.fc7 = nn.Linear(128, 256)
11       self.fc8 = nn.Linear(256, self.size)
12
13    def encode(self, x):
14        h1 = F.relu(self.fc1(x))
15        h1 = F.relu(self.fc2(h1))
16        h1 = F.relu(self.fc3(h1))
17        return self.fc41(h1), self.fc42(h1)
18
19    def reparameterize(self, mu, logvar):
20        std = torch.exp(0.5*logvar)
21        eps = torch.randn_like(std)
22        return mu + eps*std
23
24    def decode(self, z):
25        h3 = F.relu(self.fc5(z))
26        h3 = F.relu(self.fc6(h3))
27        h3 = F.relu(self.fc7(h3))
28        return torch.sigmoid(self.fc8(h3))
29
30    def forward(self, x):
31        mu, logvar = self.encode(x)
32        z = self.reparameterize(mu, logvar)
33        return self.decode(z), mu, logvar
```

Listing 3: VAE Class in PyTorch

**Results**  Once the model has been trained, a set of trials were run to test how many distinct feasible auxiliary vectors the VAE could generate. The trained model is given a set of 10,000 auxiliary vectors, with the vast majority of those ($\approx 99.9\%$) being infeasible. Once the set of new vectors is passed through the model and normalized, we use `genIsingData.py` to verify if a given auxiliary array is feasible or infeasible. The figure below summarizes some of earlier attempts using the VAE. Entries (unless otherwise labeled) ran on 5,000 epochs, with a Kullback-Liebler Divergence (KLD) term modified by a factor of $1 \times 10^{-2}$ and a learning rate of $1 \times 10^{-4}$. 
<table>
<thead>
<tr>
<th>Encoder Architecture</th>
<th>Distinct Feasible</th>
<th>Distinct Infeasible</th>
<th>Total Infeasible</th>
<th>Total Feasible</th>
</tr>
</thead>
<tbody>
<tr>
<td>128/64/32/28</td>
<td>6926</td>
<td>7077</td>
<td>2923</td>
<td></td>
</tr>
<tr>
<td>256/128/64/32</td>
<td>362</td>
<td>3898</td>
<td>6102</td>
<td></td>
</tr>
<tr>
<td>256/128/64/32</td>
<td>554</td>
<td>5584</td>
<td>4416</td>
<td></td>
</tr>
<tr>
<td>256/128/64/32</td>
<td>8081</td>
<td>8086</td>
<td>1914</td>
<td></td>
</tr>
<tr>
<td>256/128/64/32</td>
<td>7867</td>
<td>7880</td>
<td>2120</td>
<td></td>
</tr>
<tr>
<td>256/128/64/32</td>
<td>5432</td>
<td>5622</td>
<td>4378</td>
<td></td>
</tr>
<tr>
<td>256/128/64/32</td>
<td>5314</td>
<td>5444</td>
<td>4556</td>
<td></td>
</tr>
<tr>
<td>512/256/128/64/32</td>
<td>6405</td>
<td>6453</td>
<td>3547</td>
<td></td>
</tr>
<tr>
<td>512/256/128/96/64</td>
<td>6488</td>
<td>6508</td>
<td>3492</td>
<td></td>
</tr>
<tr>
<td>1024/512/256/128/96/64/48</td>
<td>2997</td>
<td>3344</td>
<td>6666</td>
<td></td>
</tr>
</tbody>
</table>

Table 14: Results of training on several architectures

1 - These architectures have a learning rate of $1 \times 10^{-3}$ and an unmodified KL Divergence
2 - These models ran on 2.5k epochs instead of 5k epochs, still has an unmodified KLD term and LR = $1 \times 10^{-3}$
3 - Ran on 2.5k epochs, but the KLD term and LR were modified to $1 \times 10^{-2}$ and $1 \times 10^{-4}$ respectively

From Table 1, we see that the quantity of distinct feasible auxiliary arrays typically improves as we increase the size of the neural network architecture, as well as being affected by the learning rate and the KLD term. The maximum number of distinct feasible solutions was only roughly half of all the outputs from the trial. By augmenting the training dataset (which initially had only 5,000 samples), the quantity of feasible auxiliary arrays produced by the VAE increased.

**Visualizing the Latent Space** To investigate the effects of the latent space, particular in regards to its effect on the number of distinct feasible auxiliary arrays that the model outputs, we visualized the latent space in an attempt. Figure ?? depicts the latent space for a trained model. We expect the mu values to be close to zero, and the (normalized) sigma values to be around one. (This is a result of the KL loss function).

![Latent Space before normalization](image1.png) ![Latent Space after normalization](image2.png)

(a) Latent Space before normalization (b) Latent Space after normalization

Figure 14: Representations of the data within the latent space

It is not very intuitive to extract relevant information from merely looking at a series of samples from the latent space. To get a better idea of what information to analyze in order to improve the results of the VAE, it is appropriate to visualize our samples as probability distribution functions.

Upon viewing the probability distribution graphs, two of the three most common results were undesirable. If the graphs are too thin, as seen in Figure ??, then the noise injection is ignored by the model. These narrow probability distribution functions occur when $\sigma \approx 0$, making $z = \mu$ after the reparameterization.
The effect of this is that the output of the VAE will more likely be an identical reconstruction of the input.

The second undesirable result is if the probability distributions were too close together, as seen in Figure 15b. In this case, the intuition behind the VAE implies that the model is not learning the structure of the data well. Let $x_i$ ($i = 1, 2, ..., n$) denote a set of inputs to the VAE. Denote the corresponding encoded representations by $z_i$. Then,

$$z_1 \approx z_2 \approx z_3 \approx ... \approx z_n$$

This indicates that the VAE is unable to distinguish between different input values (for this dimension of the latent space). Each latent space is interpreted as an “essential characteristic” of the underlying data that the VAE has determined is important. This result indicates that either the VAE is not finding such features, or the dimension of the latent space has unnecessary dimensions. (This result is peculiar since we observe an improvement for VAEs with larger latent spaces.)

![Graph](image)

(a) The First Undesirable Latent Space

(b) The Second Undesirable Latent Space

Figure 15: Undesirable probability distributions after reparameterization

The third outcome, which is the most ideal, is when the resulting probability distributions have some level of overlap, as seen in Figure 15a. This matches the intuition behind VAEs – the inputs values have varying values of some “essential characteristic” that the VAE has found. Further, there is overlap between the various latent space representations, indicating that the VAE has constructed a latent space where there is some type of “similarity” between different inputs that the model is exploiting. When observing the outputs of the VAE model, typically the closer the loss converges to zero, the more desirable the probability distributions will appear. This hints that the loss is strongly related to the amount of distinct auxiliary arrays that are similar, but not identical to the inputs – specifically, that lower losses in the VAE result in higher quantities of distinct feasible auxiliary arrays.

**Improving the VAE** There are two other considerations to take into account in order to maximize the number of distinct feasible auxiliary arrays: the architectures of our encoder and decoder models and the training data provided to the VAE.

From Table 11, it appears that there are usually a higher amount of distinct feasible auxiliary arrays and total auxiliary arrays as the sizes of each layer (and layers added) increases. However, the cost of training models with a larger set of nodes is the amount of time required to compute large matrix multiplications on the CPU. To mitigate this, PyTorch has functionality to exploit GPUs on a given system. In Table 12 below, architectures that started with 2048 as the layer with largest amount of nodes, only took approximately an hour to train, giving roughly a 3 times speed-up from utilizing merely the CPU.
The second consideration is the lack of training data. The results shown in Table 11 are as a result of using a set of 5,000 solutions on MUL 2x3x1. This is especially important since there could potentially be data not within the training set that is of larger Hamming distance than the current set of solutions. Therefore, when the VAE attempts to map some reparameterized $z$ to an output $x'$, it may some features between a new set of feasible arrays (with a greater Hamming distance) and solutions that are already present leading to even more unseen auxiliary arrays.

To get a better idea of how the amount of training data would affect the VAE’s outputs, Table 12 shows some results of adding new data into training. In these tests, all training weights have been run for 2.5k epochs. Based on Table 12 the VAE was able to distinguish enough shared features to find more solutions than previously. In general, the proportion of distinct feasible solutions to all feasible solutions grows when the loss is much lower, there is more training data, and the architecture is bigger.

Data Quality Once a large quantity of auxiliary arrays had been generated relative to the amount of training data, the quality of the solutions was investigated. In addition to the histogramming the pairwise Hamming distances of the data, the amount of clustering of the data was visualized. Ideally there is not a lot of clustering and clusters are spread out.

For the MUL 2x3x1 problem, once 68k data points were obtained then 1k random auxiliary arrays were sampled and their distributions were measured. The Jensen-Shannon distance indicates a very close approximation to a binomial distribution and there is only some clustering of Hamming distance in solutions. The results demonstrate that the outputs of the VAE are of good quality, likely due to the statistical nature of the model.
Once MUL 2x3x1 was assessed for quality and quantity, the viability of the VAE was also investigated on the MUL 3x3x3 problem with some architectures that performed strongly on the former problem. The results are shown in Table 13. Given that MUL 3x3x3 is much more computationally difficult, with $10^{57}$ possible permutations of auxiliary arrays, a significantly fewer number of solutions were found. Despite this, the proportion of distinct feasible auxiliary arrays to the total number generated is slightly higher in the MUL 3x3x3 problem!

The quality of the solutions also follows a similar trend to the MUL 2x3x1 problem. However, due to the large sample space and the limited number of the solutions, it is much more difficult to obtain a shape closer to a binomial distribution on the distribution of hamming distance for the two different architectures that produced solutions.

The encoder architecture with a latent space of 96 has fewer solutions; however, the distribution of Hamming distances matches more closely to a binomial distribution in comparison to the encoder architecture with a latent space of 256. There is also less clustering on the former architecture than the latter. The quality of the solutions created by the VAE is promising based on the distribution of Hamming distances.

<table>
<thead>
<tr>
<th>Encoder Architecture</th>
<th>DF</th>
<th>TF</th>
<th>Feasibles in Training</th>
<th>Proportion of DF to TF</th>
<th>Eval Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2048/1024/512/256/128/96/96</td>
<td>120</td>
<td>128</td>
<td>$\approx 82k$</td>
<td>93.75%</td>
<td>36 mins</td>
</tr>
<tr>
<td>2048/1024/512/256/128/128/256</td>
<td>248</td>
<td>285</td>
<td>$\approx 82k$</td>
<td>87.02%</td>
<td>57 mins</td>
</tr>
</tbody>
</table>

1 - "Eval Time" is the approximate time it takes the trained model to evaluate the correctness of solutions produced by the VAE.
Figure 18: Hamming Distances of the Generated Auxiliary Arrays for MUL 3x3x3
**Speed**  The speed of the VAE in generating feasible auxiliary arrays for the MUL 3x3x3 problem surpasses the traditional approach using genIsingData.py, which takes a median time of 45 minutes to generate a single feasible auxiliary array for the MUL 3x3x3 problem (see Appendix D). As seen in Figure 19, the architecture with a latent space of 256 obtains 4.35 solutions per minute, while the architecture with the latent space of 96 obtains 3.33 solutions per minute. The convolutional VAE (discussed below) had roughly 9.09 solutions per minute. All three are significant increases from the original solving method of 0.022 solutions per minute.

![Solution Time for VAEs (MUL 3x3x3)](image)

Figure 19: Benchmark of solutions per minute with both architectures and the original solver (genIsingData.py)

### 3.2.2 Other VAEs

**Conditional Variational Autoencoders**  Conditional Variational Autoencoders, or cVAEs for short, are very similar to a traditional VAE. The only difference is cVAEs attach a label to the inputs of the data, as well as inputs to the latent space. The idea is to have the classification carry through both the inputs and latent space, such that the latent space can identify the type of input it was given (feasible on infeasible, for the reverse Ising problem). The cVAE was implemented. Results were similar to the vanilla VAE.

**Convolutional Variational Autoencoders**  Convolutional Variational Autoencoders, or CVAEs, function much like a traditional VAE. The main difference is that CVAEs feature the use of convolution in each of the layers of the encoder and decoder neural networks. Inputs to each of the layers are multiplied by segments of a related matrix, which in other words, is convolved.

The convolutional VAE (CVAE) was explored minimally. By testing an encoder architecture of 256, 128, 64 the on MUL 3x3x3, 1864 total feasible auxiliary arrays were found. However, only 591 of those were distinct feasible, meaning only approximately 31.7% of all feasible auxiliary arrays were distinct. There were also 7685 distinct infeasible auxiliary arrays and 8136 total infeasible auxiliary arrays. Although the quantity of solutions is high, the quality of the data is between the non-convolutional VAE with a latent space of 96
and the one with a latent space of 256. There is significantly more clustering in this set, but it does try to follow a normal distribution, despite some level of skew.

Figure 20: Hamming Distances of the Generated Auxiliary Arrays for the Convolutional VAE

Additionally, the convolutional VAE had roughly 9.09 solutions per minute, but a marginally small proportion of distinct feasibles to total feasible auxiliary arrays.

3.3 Reinforcement Learning

Motivated by the theory behind the Artificial-Incremental Algorithm (see Appendix D), the most successful feasible array generating algorithm prior to this summer, we develop a reinforcement learning approach to our problem. We begin this section with a background discussion of reinforcement learning; next, we provide a description of the methods we develop for our problem, followed by results on the MUL 2x3x1 and MUL 3x3x3 problems.

**Background**  
*Reinforcement Learning* is a machine learning paradigm in which an *agent* learns desired behavior through feedback. Specifically, an agent interacts with its *environment* by taking actions to move from one environment state to the next state. The agent’s ultimate goal is to arrive in a *terminal* state. The agent receives feedback in the form of a reward, meaning each state has an associated reward. Good, or desired, states have a high reward, and the agent, during training, learns how to arrive at a desired terminal state. We are interested in *model-free* reinforcement learning because our problem is quite complex, and all of the intricacies are not understood. Model-free reinforcement learning allows us to treat the environment of the problem as a black box of sorts. The basis of all RL algorithms is a variation of the *Bellman Equation*. The state-action Bellman Equation assigns a value to each (state, action) combination by observing that the value of (state, action), $Q(s, a)$, is equal to the sum of the immediate reward of taking action $a$ from state $s$ to arrive in next state $s'$ and the discounted value of $s'$, assuming the agent acts optimally. Specifically, the state-action Bellman Equation is: $Q(s_t, a_t) = E[R_t + \gamma Q(s_{t+1}, a_{t+1})]$, where $\gamma$ is a discount factor in $[0, 1]$, which represents how highly we value future rewards compared to immediate ones (1 means we value them just as highly, while 0 means we only value the most immediate reward). Several variations of model-free reinforcement algorithms exist, but in all, we train an agent for $n$ episodes, each with $t$ time steps; in each time step, we take some action $a$ from some state $s$ and update our (state, action) values using some update equation based on a Bellman Equation. For our approach, we use $Q$ Learning, which has the following update equation: $Q(s, a) = Q(s, a) + \alpha [R(s, a) + \gamma \max(Q'(s', a')) - Q(s, a)]$, where $\gamma$ is the discount factor and $\alpha$ is the learning rate. For step-by-step details of the Q-learning algorithm, see [5].
Adaptation to our Problem  Reinforcement learning is quite common—and successful—in games. Additionally, it is easy to conceptualize reinforcement learning in the context of a game because the environment, agent, states, and actions are rigidly defined. For example, a reinforcement learning paradigm in a Pacman game has the following structure: the environment is the PacMan grid, the agent is PacMan, a state is a snapshot of the grid (perhaps broken down into features, like the locations of PacMan, ghosts, and food), and legal actions are UP, DOWN, LEFT, and RIGHT. The art of reinforcement learning comes in the reward. How we assign rewards to states is arbitrary and dictates the performance of a reinforcement learning approach. As such, the performance of reinforcement learning on the Reverse Ising problem is dependent on a good reward function. Keeping the inherently game-like nature of reinforcement learning in mind, we gamify a specific Reverse Ising Problem as follows:

1. **States** are auxiliary arrays

2. the **Environment** is the set of all auxiliary arrays for the problem (e.g. $2^{32}$ for MUL 2x3x1)

3. **Actions** are flipping any one element in the array

4. a state’s **Reward** is equal to the size of the array minus the minimized sum of the artificial variables introduced to the array’s matrix inequality system. Thus far, we have only considered auxiliary array feasibility as a binary value (feasible or infeasible); now, we must treat feasibility as a scale—precisely what the minimized sum of the artificial variables does, with a lower value meaning an array is more feasible. 0 represents a completely feasible array, so for our reward we subtract the minimized sum from the array size so that better, or more feasible, arrays have a higher reward.

5. we can think of the **Agent** as an element-flipping robot that we are training to turn random auxiliary arrays into feasible ones in as few flips as possible.

Figure 21 presents a diagram demonstrating our RL strategy. The robot agent is in an initial state. It takes a certain action (by flipping an element) to arrive in another state.

The basic approach to Q Learning is tabular Q-learning, which stores and updates Q values for all (state, action) combinations. For small environments this is possible. For most practical problems, there are simply too many such combinations to store all of them. Chess, for example, has about $10^{45}$ states—even excluding the actions, this is too big a number to store. MUL 3x3x3 has $2^{192} = 10^{56}$ states and 192 actions per state, for a total of $192 * 10^{56}$—again, more states than is possible to store in a computer. Thus, for our implementation, we use deep Q-Learning, which uses neural networks to approximate the function $Q(s, a)$, rather than learn it for every possible input like tabular Q-learning.

**Implementation** We use template deep Q-Learning code written in PyTorch provided in [9]. To create our problem environment, we use Gym, a Python package for reinforcement learning that allows for the creation of custom environments. Specifically, we create a class `ReverseIsingEnv(gym.Env)`, the two most important attributes of which are `self.observation_space = spaces.MultiBinary(self.aux_array_size)`, which defines a state as a list of `self.aux_array_size` 0’s or 1’s (we convert the 0’s to -1’s when needed) and `self.action_space = spaces.Discrete(self.aux_array_size)`, which defines the agent’s valid actions as a number in $\{x | x \in [0, self.aux_array_size - 1], x \in \mathbb{Z}\}$. We also define methods `reset(self)`, which sets the agent in a randomly selected auxiliary array for the given problem, and `step(self, action)`, which
moves the agent to the next state by flipping the element at index action (we call this method at each time step during training).

**MUL 2x3x1 Results**  We trained a variety of models, varying hyperparameters, such as learning rate, \( \alpha \) and discount factor, \( \gamma \). We train all models for 60 episodes, and in all models both the target and local neural networks\(^5\) have three hidden layers of sizes 1024, 512, and 256, respectively. We include the results of training episodes, rather than actual deployment, because we found that our algorithm performs better in training than in testing—perhaps due to more exploration. In Table 17 below, we show our preliminary results.

The Hamming distance distributions are shown in Figure 22. While decent, they are not as high as we had hoped for. We figured that the nature of our RL approach—namely beginning at a randomly selected auxiliary array each episode and the lack of training data—would produce higher quality arrays. We figured that adding in more exploration during training might improve the quality. In Q-Learning picking the best move (i.e. the move with the highest Q-value) every time is not the wise approach. Such an approach would leave many paths of the environment completely unexplored, and the final state may largely be a result of the randomly-selected initial state. This is where the notion of exploitation versus exploration enters: exploitation refers to picking that highest Q-value action, while exploration refers to randomly selecting the best action. In all of the Table 17 models—as is commonplace in typical Q-Learning—for our exploit moves, we select the best actions. We hypothesized that selecting one of the top \( n \) moves for an exploit move might help improve the pair-wise distribution of our discovered arrays’ hamming distances. We ran the same algorithms presented in Table 17 only this time, we begin by selecting one of the top 8 moves for exploit moves; we decrement the number of top moves to select from by one every 12 epochs. Table 18 below shows the results of these models, while Figure 23 shows the hamming distribution graphs for all models (except Model 4 because it’s not good).

With the exception of Model 5, where we see a big improvement, the number of feasible arrays discovered remains relatively constant. However, as hypothesized we do see an increase in the Hamming distances, as well as a closer resemblance to the binomial distribution (with the exception of Model 3, which has a lower Jensen-Shannon score in the original algorithm).

**MUL 3x3x3 Results**  Using the same parameters as Model X in Table 17 we began training a model for the MUL 3x3x3 problem. We realized that training was taking place very slowly. Upon investigation, we

\(^5\)See [1] for an explanation of why we use two neural networks

<table>
<thead>
<tr>
<th>Model</th>
<th>Epsilon</th>
<th>Batch Size</th>
<th>Learning Rate</th>
<th>Gamma</th>
<th>Update Freq.</th>
<th>Distinct Feasible, Training (/60)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.3</td>
<td>250</td>
<td>.001</td>
<td>.95</td>
<td>5</td>
<td>57</td>
</tr>
<tr>
<td>2</td>
<td>.3</td>
<td>250</td>
<td>.001</td>
<td>.3</td>
<td>5</td>
<td>57</td>
</tr>
<tr>
<td>3</td>
<td>.3</td>
<td>250</td>
<td>.001</td>
<td>.85</td>
<td>20</td>
<td>39</td>
</tr>
<tr>
<td>4</td>
<td>.3</td>
<td>250</td>
<td>.2</td>
<td>.85</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>.3</td>
<td>250</td>
<td>.001</td>
<td>.3</td>
<td>5</td>
<td>57</td>
</tr>
</tbody>
</table>

*Table 17: Results of Preliminary RL Algorithms*

<table>
<thead>
<tr>
<th>Model</th>
<th>Epsilon</th>
<th>Batch Size</th>
<th>Learning Rate</th>
<th>Gamma</th>
<th>Update Freq.</th>
<th>Distinct Feasible, Training (/60)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.3</td>
<td>250</td>
<td>.001</td>
<td>.95</td>
<td>5</td>
<td>53</td>
</tr>
<tr>
<td>2</td>
<td>.3</td>
<td>250</td>
<td>.001</td>
<td>.3</td>
<td>5</td>
<td>56</td>
</tr>
<tr>
<td>3</td>
<td>.3</td>
<td>250</td>
<td>.001</td>
<td>.85</td>
<td>20</td>
<td>51</td>
</tr>
<tr>
<td>4</td>
<td>.3</td>
<td>250</td>
<td>.2</td>
<td>.85</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>.3</td>
<td>250</td>
<td>.001</td>
<td>.3</td>
<td>5</td>
<td>56</td>
</tr>
</tbody>
</table>

*Table 18: Results of Preliminary RL Algorithms*
Figure 22: Pairwise Hamming distance distributions for reinforcement learning models (Table 17)
Figure 23: Pairwise Hamming distance distributions for modified reinforcement learning models (Table 18)
discovered that the main bottleneck came in calculating our reward for each auxillary array state—running \texttt{glopSolve} in order to minimize the sum of the artificial variables. In its current state, \texttt{glopSolve} takes about 11 seconds on average to run on a MUL 3x3x3 auxillary array. In order for RL to be a viable solution on bigger Reverse Ising problems, the 11 second average would need to be reduced drastically. Even with the massive bottleneck of \texttt{glopSolve}, we trained our algorithm for about 20 episodes, at which point we ceased training because we had seen no improvement in the average reward of the states. This indicates that more modifications are needed for a successful RL approach on MUL 3x3x3. We hypothesize that both bigger target and local neural networks, in addition to more time steps per episode.\footnote{As we did for MUL 2x3x1, we use 2,000 time steps for MUL 3x3x3. It is very likely that many more time steps are needed for the agent to explore the massive state space adequately each episode—specifically to arrive at states farther down the path, or closer to terminal, auxillary arrays.}

\textbf{Comparison to our Generative Methods}  Perhaps the main benefit of this method compared to the generative ones discussed above is that \textit{it requires zero training data}. As discussed above, training data—especially well-distributed training data—is very hard to come by for the bigger problems, such as MUL 3x3x3. The likely culprit behind the low-quality data produced by GANs on MUL 3x3x3 is the fact that we train them on rather low quality data (in the cases where we have quality data, we do not have enough of it). Because the arrays generated by this RL approach do not depend on training data, we hypothesize that we can develop models that produce arrays of great quality (since every episode begins with a randomly-selected array). While we are optimistic about the quality of arrays future versions of these models could produce, even with major improvements in calculating an array’s reward, the RL approach will take much longer than our generative models. In our generative models, we have a one time cost to \textit{train} the model, after which, generating arrays is nearly instantaneous. On the other hand, there is a timely cost with finding each array using RL because the agent always begins at a random array and must take hundreds, or even thousands, of actions to find a feasible array, and at every single state, we must calculate the associated reward.

\subsection*{3.4 Generator-Only}

\textbf{Motivation}  Scaling up to larger problems introduced much longer training times. An idea to speed up this process is to remove the discriminator altogether. In this way, we can streamline the generation process by removing the middleman and transitioning from a more complex “real vs. fake” game to a simpler goal: “feasible vs. infeasible.”

Much like \textbf{reinforcement learning}, the generator-only approach would eliminate the need for training data, which, if successful, would make attempting larger problems much easier.

\textbf{Problems with Generator-Only}  In the initial GAN structure, the discriminator would output the probability that it believed a sample came from the “real” category. This provided a measure of how close the discriminator believed an array was to being feasible. However, once we transition from “real to fake” to “feasible vs. infeasible,” this measure becomes a binary value: the array is either feasible or infeasible, nowhere in between.

We can try to borrow from reinforcement learning’s approach by creating a “scale” of feasibility: for any one auxiliary array, how many constraint equations will it satisfy in our linear system? If it makes the entire system closer to a solvable state, then it is closer to being feasible. In its current state, the lone generator’s technique is no better than random guessing, but if we replace the current black-and-white input to the generator’s loss function with this measure of feasibility, it may be able to learn.

As a result of time constraints, this modification was not implemented.
4 Conclusions and Future Work

4.1 Conclusion

Below we list all the generative models that we explored.

- Generative Adversarial Network (GAN)
  * Vanilla Generative Adversarial Network (GAN)
  * Conditional GAN (cGAN)
  * Deep Convolutional GAN (DCGAN)
  * Unrolled GAN
  * CycleGAN
  * Hamming Loss GAN (HLCGAN)

- Variational Autoencoder (VAE)
  * Vanilla Variational Autoencoder (VAE)
  * Convolutional VAE (CVAE)
  * Conditional VAE (cVAE)

- Reinforcement Learning

- Generator-Only

- Generative Random Forest

The top performers were the GAN, cGAN, DCGAN, VAE, and CVAE. The various GANs tended to score very well with the quantity metrics. The VAEs did less well with quantity metrics, but better with quality – the data generated had larger pairwise Hamming distance compared to data generated by other models. Although we had less time to explore convolutions VAEs, the reinforcement learning approach, the Hamming loss GAN, and the generator-only approaches, these showed potential and merit further exploration. Another approach that seems promising is a combinations of the Hamming Loss GAN and the DCGAN into a HLCDCGAN.

4.2 Future Work

4.2.1 Continuing Technical Work

**cGAN** We were unable to find any cGAN architectures compatible with the MUL 3x3x3 problem. We would like to continue further testing of more complex architectures. If we find a high quantity of feasible auxiliary arrays, we would also like to investigate their quality. If this approach is to continue to be used, it would also be worth adapting for use on GPUs.

**Generator-Only** The Generator-Only approach did not reach its full potential as a result of time limitations. In the future, a course of action that interests us, discussed in Section 3.3, is to borrow reinforcement learning’s feasibility scale technique for training. If its performance is satisfactory in terms of quantity, quality, and speed, we would like to try this approach on larger problems, such as MUL 3x3x3.

**Reinforcement Learning** We would like to explore different, faster methods of calculating a state’s reward. For bigger problems, our current method is much too slow. As mentioned in Section 3.3, we also believe that modifications to the models, such as bigger network sizes, and more time steps would produce better models for bigger problems. We would like to explore these approaches.
**Hamming Loss GAN**  In the future we hope to attempt our Hamming Loss GAN on the MUL 3x3x3 problem. A successful model will likely require a bigger network, in addition to many of the GAN training tricks listed in Table 6.

**Convolutional VAE**  The exploration of the Convolutional VAE (CVAE) was only briefly explored. Although the data does look promising, some adjustments could be made to improve the quantity and/or quality of the data generated using this model. Some approaches could include the utilization of multiple GPUs to speed up training time, larger layers overall (notably a higher dimensional latent space seems to help improve the quality of the results), results from the DCGAN indicate that increasing the kernel size could lead to better quality results, or increasing the amount of strides. The effects of padding in a CVAE may also give different results in future iterations; the current version uses 'same' padding for each of the convolutional layers.

### 4.2.2 Applying Results to Ising Machines

A very interesting future research exploration concerns applying the data generated over the course of the summer to program existing Ising machines, i.e., Ising machines that are built in hardware. This research area comes with its own set of challenges. There may be physical limits on the dynamic range of the biases and coupling strengths. Further, there will likely be physical limits on the precision of the biases and coupling strengths, e.g., the values may need to be integers.

Further, although every feasible auxiliary array mathematically transforms the reverse Ising system of equations into one that is solvable, the probability of the correct result depends on the relative values of the Hamiltonians for every configuration in the system. Thus, the local biases and coupling strengths derived using a particular auxiliary array might not result in an Ising machine that performs with sufficient accuracy on the actual computations. This will be hardware dependent.

One research question of this work will be to use hardware specifications to formulate properties of the system Hamiltonian or of some function of the local biases and coupling strengths that lead to accurate computation. Once these are formulated, an optimization over the datasets generated over the summer can be run. This will hopefully lead to improved performance of the Ising machine.

Further research into the relationship between the Hamming distance between a set of auxiliary arrays and the relationships between the resulting values of the objective function decided upon is another interesting mathematical question.

---

7 for each fixed combination of input spins
Acknowledgements

We would like to express our appreciation to our mentors, John Daly and Jess Meyer. They provided support and guidance over the course of this research project.

References


Appendices

A Description of Software Tools

A.1 Tools for Dataset Preparation

genCoefMat is a C code used to create the coefficient matrices used by genIsingData. Below is an example of using it to generate the coefficient matrices required for the 3-bit by 3-bit multiply problem with three auxiliary spins (i.e., MUL_3x3x3).

```
> ../../code/genCoefMat MUL 3 3 3
op = MUL
i = 3, j = 3, k = 3
m = 6, n = 6, l = 15
M = 99, N = 32256
Progress: 100.00%
Elapsed time: 8.81 seconds
```

genIsingData is a Python script used to generate auxiliary arrays that serve as training data for machine learning. It is also used to evaluate the feasibility of the auxiliary arrays generated by machine learning. It can create its own random arrays to test, or it can use input values in from a file. Below is an example of using the script to generate all possible auxiliary combinations for a 1-bit by 1-bit XOR gate with one auxiliary spin (i.e., XOR_1x1x1).

```
> ./genIsingData.py --data ../matrices --op XOR 1 1 1
Version: Python 3.6.8
Command: ./genIsingData.py --data ../matrices --op XOR 1 1 1
Problem: XOR_1x1x1
m = 2, n = 1, l = 4
K = 4, M = 7, N = 8
Reading Cmat.mm
Reading Lmat_1.mm
Generating 16 auxiliary test vectors
[==================================================] 100%
Running glop solver on 1 of 1536 cpus
Testing 16 models for feasibility (True/False)
1 [[-1, -1, -1, -1]] False
2 [[-1, -1, -1, 1]] True
3 [[-1, -1, 1, -1]] True
4 [[-1, -1, 1, 1]] False
5 [[-1, 1, -1, -1]] True
6 [[-1, 1, -1, 1]] False
7 [[-1, 1, 1, -1]] False
8 [[-1, 1, 1, 1]] True
9 [[1, -1, -1, -1]] True
10 [[1, -1, -1, 1]] False
11 [[1, -1, 1, -1]] False
12 [[1, -1, 1, 1]] True
13 [[1, 1, -1, -1]] False
14 [[1, 1, -1, 1]] True
15 [[1, 1, 1, -1]] True
16 [[1, 1, 1, 1]] False
Found 8 (50%) feasible auxiliary combinations
Overhead peak memory usage: 36.98 Mbytes
Per processor memory usage: 39.16 Mbytes
Total read time (all files): 0:00:00.076404 (hr:min:sec)
Total build time (all cpus): 0:00:00.017155 (hr:min:sec)
Total solve time (all cpus): 0:00:00.045855 (hr:min:sec)
Application wall clock time: 0:00:00.189214 (hr:min:sec)
A.2 Tools for Dataset Post-Processing

**hammingDistance** is a Python script used to analyze the quality of the auxiliary array data. It produces a histogram of all the pairwise Hamming distances in an auxiliary array dataset and computes the Jensen-Shannon distance between the histogram of Hamming distances and a perfect binomial distribution. It also constructs a connectivity graph to aid in visualizing the clustering of auxiliary array values. Below is an example of using it to visualize 600 data points generated for the 3-bit by 3-bit multiply problem with three auxiliary spins (i.e., MUL_3x3x3).

```
> hammingDistance.py --cluster 16 data/MUL_3x3x3.dat
Read 600 auxiliaries from file "data/MUL_3x3x3.dat"
[==================================================] 100%
Computing Hamming distance statistics
[==================================================] 100%
Mean Hamming distance: 96.16
Median Hamming distance: 96.00
Jensen-Shannon distance: 0.23
```

**npy2txt** is a Python script used to reformat auxiliary arrays written in the standard binary format for NumPy to human readable text format. Below is an example of using it to reformat data generated for the 3-bit by 3-bit multiply problem with three auxiliary spins (i.e., MUL_3x3x3).

```
> npy2txt.py ../data/MUL_3x3x3.npy
Write 120 auxiliaries in text format to file "npy2txt.out"
[==================================================] 100%
```

B Tools for Training and Testing

In this section, when we refer to file or directory paths, we are referring to our internal GitLab repository, **ising_machine_learning**.

B.1 Directory **ising_machine_learning/code/btkahn/reinforcement_learning**

This directory contains all code and data for our reinforcement learning approach to the problem (Section 3.3).

**gym-env directory**

1. **Description**: Before developing RL algorithms for our problem, we had to develop supporting code that allows us to interact with the Reverse Ising Problem environment. **gym-env** is the directory that contains the Gym environment we have created for the problem, following the directory hierarchy here. We describe Gym and the important files below:

2. **Installation** In order to use this Gym environment in the other scripts in the **reinforcement_learning** directory, you must install Gym with `pip install gym` and install this directory itself with `pip install -e gym-env`.

3. **Gym Background** Gym is an API/Python library that makes reinforcement learning (RL) easy. One can import existing environments, or create their own. An environment is essentially just the space in which the RL agent is acting—it’s easy to think of it as a game (e.g. Gym has a LunarLander environment). Gym environments contain all necessary code for the environment/game, allowing the actual RL implementations to be abstracted away. A well-created Gym environment will contain methods to reset the environment, for the agent to step through its environment, in addition to the actual GUI of the game (sometimes).
Files

1. `/gym_env/envs/reverse_ising_gym_env.py`: this is the actual environment file. It defines a `ReverseIsingEnv(gym.Env)` class. One must create an instance of this class to interact with the Ising environment.

2. `/gym_env/envs/artificialising.py`: As mentioned in Section 3.3, we use the minimized sum of the artificial variables as our reward function for our states. Given a `psiList` and `rhoList` of an auxiliary array, the `glopSolve` function returns this value (in addition to a Boolean denoting whether or not the array is feasible). To calculate a next state’s reward in its `step` method, `reverse_ising_gym_env.py` calls `getObjStatus`, which calls `glopSolve`.

3. All files beginning with `deep_Q_reverse_ising`: These are the main deep Q learning scripts. Each one is the same, with the exception of certain training parameters and problem size. The code was adapted from the tutorial in [9] to work for our problem. If you want to try different RL approaches (e.g. change hyperparameters, such as `BATCH_SIZE`, `LEARNING_RATE`, etc.), copy one of these files and change the code as desired. Simply run `python3 [script_name].py` to run. As stated above, make sure you have installed the environment with `pip -e install gym-env`.

4. `RL_analysis.ipynb`: We use this notebook to analyze the performance of our RL models.

B.2 Directory `ising_machine_learning/code/btkahn/GAN`

This directory contains all code and data for our Vanilla GANs, CycleGANs, and Unrolled GANs.

1. `main_ising_gan_2x3x1.ipynb`: Main training and evaluation notebook for Vanilla GANs.

2. `main_ising_cycle_gan_2x3x1.ipynb`: Main training and evaluation notebook for our Cycle GAN implementation.

3. `main_ising_unrolled_gan_2x3x1.ipynb`: Main training and evaluation notebook for our Cycle GAN implementation.

4. `main_ising_hamming_loss_gan_2x3x1.ipynb`: Main training and evaluation notebook for our Cycle GAN implementation.

5. `ising_gan_results_analysis.ipynb`: we use this notebook to analyze our models’ performance. We look at metrics, such as loss, accuracy, and total feasible.

6. MUL_2.3.1 Directory:

   (a) `ising_gan_model.py`: a script that contains the training components of `main_ising_gan_2x3x1.ipynb`; for use on remote computers.

7. textbf{MUL_3.3.3} Directory:

   (a) All .py files beginning with `3x3x3`: GAN training scripts (adapted for the MUL 3x3x3 problem on remote machines.

B.3 Directory `ising_machine_learning/code/krclark/gan`

This directory contains all code and results directories for the conditional GAN (cGAN) and the Generator-Only approaches.
Files

1. **ising.cgan** is the script used to train and test cGAN models. It reads in a list of auxiliary arrays that have been passed through `genIsingData` for a particular problem. It splits this data, converts it to `torch` tensors, then passes it through customizable generator and discriminator architectures in a training loop. Post-training, it splits the results into 5 files under a directory for the specific problem (described in **Sub-Directories**):
   - Data (.csv): Saves the trial number, seed, and all generated arrays and their feasibility for each trial of testing
   - Results (.csv): Saves a results DataFrame that stores the trial number, seed, number of generated arrays, number of feasibles, number of distinct feasibles, generator and discriminator accuracy, and trial time for each trial of testing. This DataFrame is also printed to the terminal at the conclusion of testing.
   - Generated Feasibles (.csv): Saves all distinct generated feasibles, dropping duplicates
   - Discriminator Model (.pt): A PyTorch file that saves the trained discriminator model
   - Generator Model (.pt): A PyTorch file that saves the trained generator model

2. **cgan.functions** contains all of the functions utilized in **ising.cgan**.

Sub-Directories

1. **MUL_2_3_1**
   - `makeNewTrainData` collects all generated feasibles from a subdirectory and formats them for use with `genIsingData`.
   - `_INTERESTING_OBSERVATIONS_` has an informal description of the cGAN techniques described in Table 9.
   - The 231 feasibles list files are created via `makeNewTrainData`.
   - The `EVERYTHING_ELSE` and `MODELS_THAT_SEMI_WORK` directories hold sorted results files from **ising.cgan**.

2. **MUL_3_3_3** has its own `EVERYTHING_ELSE` and `WORKING_MODELS` directories. Currently, `WORKING_MODELS` is empty.

3. **G** houses files for the Generator-Only approach.

   - **ising_generator** is a watered-down iteration of **ising.cgan**. The primary change is the training loop, where the generator-generated samples are passed through `genIsingData` rather than the discriminator.
   - **generator.functions** contains all of the functions utilized in **ising_generator**. Many are almost identical to those in **cgan.functions**.

B.4 Directory **ising_machine_learning/code/krclark/scikit**

Files

The files in this directory are likely almost identical to those in **btkahn/ and adsmith/**.

   - **pipeline** is the preprocessing script for discriminative machine learning. Its command line argument is a file that has been passed through `genIsingData`.
   - **machine_learning_pipeline** performs the actual training and testing of discriminative models. It has no command line arguments.
### B.5 Main Interface Code (found in ising_machine_learning/code)

We have created some starting functionality for our goal of creating a main Ising-generation interface. The ultimate goal of this program is for the user to be able to use any of our machine learning approaches on any of the Reverse Ising Problems to generate any number of auxiliary arrays. The program should save results appropriately and record statistics about the experiment/generation process. That is the goal. Below is a description of the components in their current states.

1. **mainInterface.py**: This is the main script of the main interface code. It must be run with `python3 mainInterface.py`, at which point the user must make a number of menu selections: 1) desired operation (e.g. “MUL” or “ADD”), 2) problem size (e.g. “2 3 1”), 3) desired model, and 4) desired action (“generate” or “train”). If a trained model exists for the desired operation, problem size, and model combination, “generate” is on number (4). If an untrained version of the desired model exists (i.e. the code for training is present, but it hasn’t been trained), “train” will be on (4). If the user selects “generate”, they then input how many arrays they want to (attempt to) generate. At that point, the model code is run. `mainInterface.py` is currently very implementation-specific, in that it only works for PyTorch GANs.

2. **trained_models directory**: Contains the model files, under `[model_type]/[problem_operation]/G_[problem_dimensions]` (e.g. the Vanilla GAN Generator model for MUL 2x3x1 is in trained_models/VanillaGAN/MUL/G_2x3x1).

3. **trained_models_pytorch_classes directory**: Contains the relevant code needed to load the model files in, under `[model_type]/[problem_operation]/dim1x_dim2x_dim3.py` (e.g. the Vanilla GAN Generator class code is in trained_models_pytorch_classes/VanillaGAN/MUL/2x3x1.py).

4. **untrained_models directory**: This directory has a similar naming convention to the above two. It must contain all code required to train the model. Functionality in `mainInterface.py` has yet to be developed in order to train these models.

5. **main_interface_data directory**: The results of any experiments will be outputted here. The subdirectory naming convention is the same as the above directories. For each experiment run, two CSV files are created: `YYYY-mm-dd hh:mm:ss_data.csv` which contains all generated samples of the experiment and `YYYY-mm-dd hh:mm:ss_results.csv`, which stores statistics about the experiments (number feasible, time, etc.)

### B.6 ising_machine_learning/code/btkahn/pipeline.ipynb

This is one of the notebooks we used to test common discriminative models on our data.

### B.7 ising_machine_learning/code/btkahn/pipeline_functions.py

This file contains many helper functions for all aspects of the ML pipeline (in both our discriminative and generative models), such as functions for pre-processing data, post-processing data, converting raw Generator samples to auxiliary arrays of -1s and 1s, running `genIsingData.py`, etc.

### B.8 ising_machine_learning/code/adsmith

The following directory includes the work done for (mostly) the VAE (and different types of VAEs explored). Below is a list of several directories and their brief descriptions

1. **Base Directory**: Leads to the other directories, but mostly contains any scripts necessary to train, test and evaluate VAE models for the MUL 2x3x1 problem.

2. **MUL_3x3x3**: This directory contains all the types of VAE models attempted for the MUL 3x3x3 problem, which follows a similar convention to the base directory and files associated for the MUL 2x3x1 problem.

3. **sklearn-tests**: A directory dedicated to running tests and models with scikit-learn and two classifiers associated with scikit-learn on both MUL 2x3x1 and MUL 3x3x3 problems.
B.8.1 Base Directory

Below is a list of several important scripts that are associated with the directory `ising_machine_learning/code/adsmith`.

1. **isingVAE.py**: This is the original script which uses the Pytorch library for the VAE. The script trains the model for a certain amount of epochs (which can be given by the user, or defaults to 10 epochs) and saves the weights in a nearby directory. The script automatically attempts to train on a GPU if there are any on the system, otherwise it defaults to utilizing the CPU. Results are saved to the `weights/` directory.

2. **isingVAE_test_gpu.py**: In order to evaluate the feasibility of the trained VAE model, `isingVAE_test_gpu.py` is the main script that performs this action. Borrowing functions from a script called `pipeline_functions.py` (which contains an assortment of functions to assist in evaluating auxiliary arrays before and after the model evaluates inputs), it displays several statistics about the evaluated model on a set of 10,000 auxiliary arrays passed into the VAE. If there are solutions that have not been seen before, the script will automatically append newly found solutions to the training data. Results are then appended to a CSV file, which is labeled `VAE_Results_MUL_2x3x1.csv`.

3. `pipeline_functions.py`: This script is a variation of the script `pipeline_functions.py` found in the `btkhan` directory. The major difference is that this has modified functions which navigate directories and evaluate models in order to display statistics of how a trained VAE model performs.

4. **isingCVAE.py**: This script uses PyTorch in order to train a cVAE model, rather than a VAE model. The label is fed in the latent space, as well as in the input. The final trained weights are saved in the `weights/` subdirectory, much like `isingVAE.py`.

5. **isingCVAE_test.py**: Much like `isingVAE_test_gpu.py`, it performs the exact same action in order to evaluate the cVAE model and obtain results.

6. **tf_convVAE.py**: This script uses Tensorflow in order to implement a CVAE, which was adapted from [6] to the reverse Ising Problem. Weights are saved to a `tf/` subdirectory. Once the model is finished training, the program will proceed to evaluate the model and output statistics, much like `isingVAE_test_gpu.py` and `isingCVAE_test.py`, such as distinct feasible auxiliary arrays.

7. **genIsingData_MUL_2x3x1_Soln_Only.npy** Dataset comprising of only solutions for the MUL 2x3x1 problem. Currently contains roughly 68k feasible auxiliary arrays.

B.8.2 MUL_3x3x3 Directory

1. **isingVAE_MUL_3x3x3.py**, **isingVAE_MUL_3x3x3_test.py**: Both scripts perform identically to `isingVAE.py` and `isingVAE_test_gpu.py` respectively, except the input for training and testing the VAE is expanded from 32 to 192. Results were also saved to a CSV, labeled `VAE_Results_MUL_3x3x3.csv`.

2. **tf_convVAE_MUL_3x3x3.py**: Behaves identically to `tf_convVAE.py`, but it is applied to the MUL 3x3x3 problem.

B.8.3 sklearn-tests Directory

1. **knnClassifierTest_MUL_2x3x1.py**, **knnClassifierTest_MUL_3x3x3.py**: Both scripts utilize the K-Nearest Neighbors classifier from the scikit-learn library. The former and the latter have this classifier applied to each of their respective problems. The program takes in the datasets for each of the problems, splits the dataset into a training and testing set and finally uses the non-linear classifier to attempt to classify auxiliary arrays as feasible or infeasible.
C Characterizing Scaling of the Reverse Ising Problem in Term of the Input Parameters

Reverse Ising problems are named using the convention \texttt{<op>_<i>x<j>x<k>}\, where \texttt{<op>} is one of AND, OR, XOR, ADD, MUL and \texttt{<i>}, \texttt{<j>}, and \texttt{<k>} are respectively the number of input spins for the first operand \((i)\), input spins for the second operand \((j)\), and auxiliary spins \((k)\). The number of input spins \((m)\), output spins \((n)\), and the total number of spins \((l)\) are then given as follows. Notice that this assumes neither the multiplicand or multiplier is 1. If either is 1 then logical MUL is equivalent to logical AND.

\[
m = i + j
\]

\[
n = i + j
\]

\[
l = m + n + k
\]

The number of local biases \((h_i)\), excluding inputs that are zero, is \(l - m\). The number of coupling strengths \((J_{ij})\), excluding those between inputs which do not interact, is given by \(\frac{1}{2}(n + k)(m + l - 1)\). Together, these are referred to as the potential variables. The total number of auxiliary variables \((K)\) is the number of auxiliary spins times the number of input configurations. The total number of potential variables \((M)\) is the sum of the number of local bias variables and the number of coupling strength variables. The total number of inequality constraints \((N)\) is the number of input configuration times the number of auxiliary configurations times the number of incorrect output configurations. These are summarized as follows.

\[
K = k^m \\
M = \frac{1}{2}(n + k)(m + l + 1) \\
N = 2^{m+k}(2^n - 1)
\]

Now the constraint satisfaction problem can be generalized in terms of an \(N \times M\) auxiliary constraint matrix \((A)\) and an \(M \times 1\) vector of potential variables \((\psi)\) as follows.

\[
A(\alpha_\xi)_{N \times M} \cdot \psi_{M \times 1} \geq 1_{N \times 1}
\]

The auxiliary value variables are represented as an array of vectors \(\alpha_1, \alpha_2, \ldots, \alpha_k\) where each \(2^m \times 1\) dimension auxiliary value vector is defined as follows.

\[
(\alpha_\xi)_{2^m \times 1} = \begin{bmatrix}
\alpha_{\xi,1} \\
\alpha_{\xi,2} \\
\vdots \\
\alpha_{\xi,2^m}
\end{bmatrix}
\]

Using the example 1-bit by 1-bit XOR with \((m, n, l) = (2, 2, 5)\) and \((K, M, N) = (4, 7, 8)\), the values of the constraint matrix and potential vector are as follows.

\[
A(\alpha_\xi)_{8 \times 7} = \begin{bmatrix}
1 & 0 & -1 & 0 & -1 & 0 & \alpha_{1,1} \\
1 & -\alpha_{1,3} & -1 & \alpha_{1,1} & -1 & \alpha_{1,1} & 0 \\
-1 & 0 & 1 & 0 & -1 & 0 & -\alpha_{1,2} \\
-1 & -\alpha_{1,2} & 1 & \alpha_{1,2} & -1 & -\alpha_{1,2} & 0 \\
-1 & 0 & -1 & 0 & 1 & 0 & -\alpha_{1,3} \\
-1 & -\alpha_{1,3} & -1 & -\alpha_{1,3} & 1 & \alpha_{1,3} & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & \alpha_{1,4} \\
1 & -\alpha_{1,4} & 1 & -\alpha_{1,4} & 1 & -\alpha_{1,4} & 0
\end{bmatrix}
\]

\[
\psi_{7 \times 1} = \begin{bmatrix}
h_3 \\
h_4 \\
J_{1,3} \\
J_{1,4} \\
J_{2,3} \\
J_{2,4} \\
J_{3,4}
\end{bmatrix}
\]

Figure 24 below demonstrates how the constraint satisfaction problem scales for different operations and input sizes. Notice that there will be \(2^K\) possible different configurations of the \(K\) auxiliary value variables since each can have a value of +1 or -1.
Figure 24: A table of select operations and spin combinations demonstrating the scaling of the associated problem parameters.

### D Baseline Algorithm for Finding Feasible Auxiliary Arrays

Solving for Feasible Auxiliary Values: The “Artificial-Incremental” Algorithm

\[
A(\alpha_{\xi})_{N \times M} \cdot \psi_{M \times 1} \geq 1_{N \times 1}
\]

The general constraint satisfaction problem was generalized above as a simple system of inequality constraints.

In order to satisfy this set of constraints it was necessary to choose an auxiliary value vector \((\alpha_{\xi})_{2^m \times 1}\) and then solve for a potential vector \(\psi_{M \times 1}\). When the constraints are satisfiable for a given choice of the auxiliary value vector it is said to be feasible. This section discusses the current best method of identifying feasible auxiliary vector for a given constraint problem using artificial variables. An artificial variable is a type of variable introduced into a constraint satisfaction problem to test for a basic feasible solution. It is called “artificial” because it does not correspond to any physical or real characteristics of the problem. Let \(\rho_{N \times 1}\) be a vector of \(N\) artificial variables added to the constraint satisfaction problem to form a new set of constraints.

\[
A(\alpha_{\xi})_{N \times M} \cdot \psi_{M \times 1} + \rho_{N \times 1} \geq 1_{N \times 1}
\]

Notice that for the new constraint satisfaction problem it is always possible to choose values of the artificial variables that satisfy the inequalities. Additionally, in the case that the \(\rho_{N \times 1}\) are all zero the
solutions $(\alpha_\xi)_{2^m \times 1}$ and $\psi_{M \times 1}$ to the new problem will also be solutions to the original problem. The sum of the artificial variables is therefore an objective that measures how far a given auxiliary value vector is from creating a satisfiable system of inequalities. When the objective is zero the original system will be satisfiable. The artificial-incremental algorithm minimizes the sum of the artificial variables over a sequence of incremental updates to the auxiliary value vector.

$$\min_{\rho \in \mathbb{R}^{N}} \sum_{i=1}^{N} \rho_i \quad \text{with} \quad A(\alpha_\xi)_{N \times M} \cdot \psi_{M \times 1} + \rho_{N \times 1} \geq 1_{N \times 1} \quad \text{and} \quad \rho_{N \times 1} > 0_{N \times 1}$$

The artificial-incremental algorithm proceeds as follows:

1. Start with a randomly generated initial auxiliary value vector $\alpha_\xi^{(0)}$ assuming a single auxiliary spin, i.e., $k = 1$.

2. Construct the $K - 1$ candidate auxiliary value vectors that each differ from $\alpha_\xi^{(0)}$ in exactly one of its $K$ dimension.

3. Solve the minimization problem for each candidate auxiliary value vectors using a simple linear programming solver.

4. Choose $\alpha_\xi^{(1)}$ to be the auxiliary value vector that minimizes the objective. If there is more than one minimum then choose at random from the available minima.

5. Repeat until the minimum objective stops decreasing.

   (a) If the minimum objective is greater than zero add a new auxiliary spin, i.e., $k = k + 1$, and restart the algorithm at step #2 above.

      i. Random — Initialize the new spin to a random value.

      ii. Previous — Initialize the new spin to the final values of the previous spin.

   (b) If the minimum objective is greater than zero and the number of auxiliary spins exceeds a user defined upper limit then restart the algorithm at step #1 above.

   (c) If the minimum objective is zero the auxiliary value vector is feasible.

Remember that a random search of the auxiliary value vector space scales exponentially and could require generating as many as candidate solutions. On the other hand, the artificial-incremental algorithm scales polynomially in making it significantly more efficient than random search. In practice, the artificial-incremental algorithm converges relatively quickly for up to 3-bit multiplication, but the solve time is highly dependent on the initial choice of $\alpha_\xi$. With a good initial value vector the solution will converge rapidly, but with a poor initial value vector the solution will converge slowly and may require multiple restarts. Over 100 runs of the algorithm to compute MUL_3x3x3, the minimum solve time was 4.6 minutes and the maximum solve time was 17.5 hours. The median solve time was 47 minutes.