Foundations of Data Science and Machine Learning – *Homework 3*

Isaac Martin

Last compiled March 23, 2023

EXERCISE 1. Given labeled data $(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n) \in \mathbb{R}^d \times \{\pm 1\}$, consider the support vector machine with misclassification allowed but penalized by $\lambda > 0$:

$$\min_{\substack{\mathbf{w}\in\mathbb{R}^d \\ b\in\mathbb{R} \\ \mu\in\mathbb{R}^n}} \lambda\|\mathbf{w}\|_2^2 + \frac{1}{n}\sum_{j=1}^n \mu_j \quad \text{s.t.} \quad y_j(\langle \mathbf{w}, \mathbf{x}_j\rangle - b) \geq 1 - \mu_j,\ \mu_j \geq 0\ \forall j.$$

(a) Derive the dual problem by using the Lagrangian.

(b) Read about "Slater's condition" on Wikipedia. Does strong duality hold here?

Proof: For each j we have two constraints, one of the form $y_j(\langle \mathbf{w}, \mathbf{x}_j\rangle - b) - 1 + \mu_j \geq 0$ and another of the form $\mu_j \geq 0$. This means our Lagrangian is

$$\mathrm{cl}(\mathbf{w}, b, \mu, \alpha, \beta) = \lambda\|\mathbf{w}\|_2^2 + \frac{1}{n}\sum_{j=1}^n \mu_j \ - \ \sum_{j=1}^n \alpha_j(y_j(\langle \mathbf{w}, \mathbf{x}_j\rangle - b) - 1 + \mu_j) + \beta_j\mu_j.$$

The dual problem is then

$$\max_{\substack{\alpha\geq 0 \\ \beta\geq 0}} \min_{\mathbf{w},b,\mu} \lambda\|\mathbf{w}\|_2^2 + \frac{1}{n}\sum_{j=1}^n \mu_j \ - \ \sum_{j=1}^n \alpha_j(y_j(\langle \mathbf{w}, \mathbf{x}_j\rangle - b) - 1 + \mu_j) + \beta_j\mu_j$$

which becomes

$$\max_{\substack{\alpha\geq 0 \\ \beta\geq 0}} \|\alpha\|_1 + \min_{\mathbf{w},b,\mu} \lambda\|\mathbf{w}\|_2^2 + \frac{1}{n}\sum_{j=1}^n \mu_j \ - \ \sum_{j=1}^n \left(1/n - \alpha_j - \beta_j\right)\mu_j - \alpha_j y_j(\langle \mathbf{w}, \mathbf{x}_j\rangle - b)$$

after pulling out the -1 term and combining all μ_j terms. Note that technically these should be infinums and supremums since $\mathbf{w}$, b and μ are valued in the reals. If $1/n - \alpha_j - \beta_j \neq 0$, then the inside minima above will always be $-\infty$ as we can choose μ_j to be arbitrarily large or small. Similarly, unless $\sum_j \alpha_j y_j = 0$ we can choose b such that the minima is $-\infty$. If this is not the case, then we must have that $\beta_j = 1/n - \alpha_j$ and $\sum_j \alpha_j y_j = 0$. In this case we get the simpler optimization problem

$$\max_{\alpha} \min_{\mathbf{w}} \|\alpha\|_1 + \lambda\|\mathbf{w}\|_2^2 \ - \ \sum_{j=1}^n \alpha_j y_j \langle \mathbf{w}, \mathbf{x}_j\rangle$$

where $0 \leq \alpha_j \leq 1/n$ (so that $\beta_j \geq 0$ too) and $\sum_{j=1}^n \alpha_j y_j = 0$. Setting the gradient with respect to $\mathbf{w}$ to zero, we obtain

$$0 = \nabla\left[\|\alpha\|_1 + \lambda\|\mathbf{w}\|_2^2 \ - \ \sum_{j=1}^n \alpha_j y_j \langle \mathbf{w}, \mathbf{x}_j\rangle\right] = 2\lambda\mathbf{w} - \sum_{j=1}^n \alpha_j y_j x_j \implies \mathbf{w}* = \frac{1}{2\lambda}\sum_{j=1}^n \alpha_j y_j x_j.$$

After some simplification, substitution into our dual problem yields

$$\max_{\alpha} \|\alpha\|_1 + \lambda \left\langle \frac{1}{2\lambda}\sum_{j=1}^{n} \alpha_j y_j x_j, \frac{1}{2\lambda}\sum_{j=1}^{n} \alpha_j y_j x_j \right\rangle - \sum_{j=1}^{n} \alpha_j y_j \left\langle \frac{1}{2\lambda}\sum_{k=1}^{n} \alpha_k y_k x_k, x_j \right\rangle$$
$$= \max_{\alpha} \|\alpha\|_1 - \frac{1}{4\lambda}\sum_{j,k=1}^{n} \alpha_j \alpha_k y_j y_k \langle x_j, x_k \rangle,$$

again subject to the constrains that $0 \leq \alpha_j \leq \frac{1}{n}$ and $\sum \alpha_j y_j = 0$. □

EXERCISE 2. The *weighted least squares problem* is a generalization of usual least squares:

$$\min_{\mathbf{x}\in\mathbb{R}^d} \|\mathbf{WAx} - \mathbf{Wb}\|_2^2,$$

where $\mathbf{W}$ is a fixed diagonal matrix of positive weights

$$\mathbf{W} = \begin{pmatrix} w_1 & 0 & \dots & 0 \\ 0 & w_2 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & w_n \end{pmatrix}.$$

Suppose $\mathbf{A}$ is an $n \times d$ full-rank matrix, and $n \geq d$. Suppose $\mathbf{b} = \mathbf{Ax}^* + \varepsilon$ for a fixed $\mathbf{x}^* \in \mathbb{R}^d$, and suppose the noise vector $\varepsilon = (\varepsilon_i)_{i=1}^n$ is a random vector whose components $\varepsilon_1, ..., \varepsilon_n$ are independent, mean-zero Gaussians with variances $\sigma_1^2, ..., \sigma_n^2$.

(a) Under this model, derive a natural choice of weights in the weighted least squares problem by considering the likelihood function.

(b) What is a closed-form expression for the solution to the weighted least squares problem? What if we add regularization to the problem and consider $\min_{x\in\mathbb{R}^d} \|\mathbf{WAx} - \mathbf{Wb}\|_2^2 + \lambda\|\mathbf{x}\|_2^2$ for $\lambda > 0$?

Proof:

(a) Set $y = \mathbf{Wb}$ and recall that the likelihood function in this case is proportional to

$$\mathcal{L}(\mathbf{w} \mid (\mathbf{x}_i, y_i)) \propto \prod_{i=1}^{n} \exp\left(-\frac{(\mathbf{w}^T\mathbf{x}_i - y_i)^2}{2\sigma_i^2}\right) = \exp\left(\frac{1}{2}\sum_{i=1}^{n} -\frac{(\mathbf{w}^T\mathbf{x}_i - y_i)^2}{\sigma_i^2}\right).$$

Maximizing likelihood over $\mathbf{w} \in \mathbb{R}^n$ is then equivalent to minimizing $\exp\left(-\frac{(\mathbf{w}^T\mathbf{x}_i - y_i)^2}{\sigma_i^2}\right)$.

Consider the components y_i of y. Letting $\mathbf{W}_i$ be the ith row of $\mathbf{W}$ we see that

$$y_i = \mathbf{W}_i\mathbf{b} = \mathbf{W}_i\mathbf{Ax}^* + \mathbf{W}_i\varepsilon = w_i(\mathbf{Ax}^*)_i + w_i\varepsilon_i$$

since $\mathbf{W}$ is diagonal. If we choose $w_i = \frac{1}{\sigma_i}$, then the random variable $w_i\varepsilon_i$ becomes normally distributed variance 1. This seems like a natural choice of w_i as it functionally shifts the components of the noise vector to be iid.

(b) Let $\mathbf{M} = \mathbf{WA}$ and $\mathbf{d} = \mathbf{Wb}$. Then the problem becomes

$$\min_{\mathbf{x}\in\mathbb{R}^d} \|\mathbf{Mx} - \mathbf{d}\|_2^2.$$

We solved this in class, the optimal solution $\mathbf{x}^*$ is $(\mathbf{M}^\top\mathbf{M})^{-1}\mathbf{M}^\top\mathbf{d}$. In the ridge regression case, it is $\mathbf{x}^* = (\mathbf{M}^\top\mathbf{M} + \lambda I_n)^{-1}\mathbf{M}^\top\mathbf{d}$. This feels like a bit of a cop out, so I'll include that I'm referencing Lecture 15 slide 1.

(c) Yes, I believe strong duality does hold here. The book by Boyd and Lieven referenced in the Wikipedia article on Slater's condition proves that strong duality holds for quadratic optimization problems using Slater's condition.

□

EXERCISE 3. Consider a two-layer neural network model of the form

$$f(\mathbf{W}, \mathbf{a}, x) = \sum_{r=1}^{m} a_r \sigma(\langle \mathbf{w}_r, \mathbf{x}\rangle)$$

with ReLU activation function $\sigma(x) = \max(0, x)$. Fixing the second layer weights $\mathbf{a}$ and only training the first layer weights $\mathbf{W} = (\mathbf{w}_r)_{r=1}^m \in \mathbb{R}^{m\times d}$ via least squares loss, the optimization problem is

$$\min_{\mathbf{W}\in\mathbb{R}^{m\times d}} L(\mathbf{W}) = \min_{\mathbf{W}\in\mathbb{R}^{m\times d}} \frac{1}{n}\sum_{j=1}^{n} (f(\mathbf{W}, \mathbf{A}, \mathbf{x}_j) - y_j)^2.$$

(a) Derive an expression for the partial gradient $\frac{\partial L(\mathbf{W})}{\partial \mathbf{w}_r} \in \mathbb{R}^d$ where at $x = 0$ we set $\frac{d\sigma}{dx}\Big|_{x=0} = 1$.

(b) Suppose the data points are normalized so that $\|\mathbf{x}_j\|_2 = 1$. In neural network training, the weights $\mathbf{w}_r$ are initialized in gradient descent as independent spherical Gaussian vectors. Show that

$$\mathbb{E}\left[\mathbf{x}_i^\top \mathbf{x}_j \mathbf{I}\{\mathbf{w}^\top \mathbf{x}_i \geq 0,\ \mathbf{w}^\top \mathbf{x}_j \geq 0\}\right] = \mathbf{x}_i^\top \mathbf{x}_j \frac{\pi - \arccos(\mathbf{x}_i^\top \mathbf{x}_j)}{2\pi}.$$

Proof:

(a) Let's take some derivatives.

$$\frac{\partial L}{\partial w_{r,i}} = \frac{2}{n}\sum_{j=1}^{n} (f(\mathbf{W}, \mathbf{a}, \mathbf{x}_j) - y_j) \cdot \frac{\partial f}{\partial w_{r,i}}.$$

We then have that

$$\begin{aligned}\frac{\partial f}{\partial w_{r,i}} &= \sum_{\ell=1}^{m} a_r \frac{\partial}{\partial w_{r,i}} \sigma(\langle \mathbf{w}_r, \mathbf{x}_j\rangle) \\ &= \mathbf{a}_r \cdot \chi_{\mathbb{R}_{\geq 0}}(\langle \mathbf{w}_r, \mathbf{x}_j\rangle) \cdot \mathbf{x}_{r,i} = \begin{cases} \mathbf{a}_r \mathbf{x}_{r,i} & \langle \mathbf{w}_r, \mathbf{x}_i\rangle \geq 0 \\ 0 & \text{otherwise} \end{cases}\end{aligned}$$

where χ is the indicator function. Putting this together, we get that

$$\frac{\partial L}{\partial \mathbf{w}_{r,i}} = \frac{2}{n}\sum_{j=1}^{n}(f(\mathbf{W}, \mathbf{a}, \mathbf{x}_j) - y_j) \cdot \mathbf{a}_r \mathbf{x}_{r,i} \cdot \chi_{\mathbb{R}_{\geq 0}}(\langle \mathbf{w}_r, \mathbf{x}_j \rangle).$$

Fitting these together into a vector gives us the partial gradient $\nabla_{w_r} L = \left(\frac{\partial L}{\partial w_{r,1}}, ..., \frac{\partial L}{\partial w_{r,m}}\right)$.

(b) Notice first that since $\mathbf{x}_i$ and $\mathbf{x}_j$ are fixed, we can remove them from the expectation:

$$\mathbb{E}\left[\mathbf{x}_i^\top \mathbf{x}_j \mathbf{I}\{\mathbf{w}^\top \mathbf{x}_i \geq 0,\ \mathbf{w}^\top \mathbf{x}_j \geq 0\}\right] = \mathbf{x}_i^\top \mathbf{x}_j \mathbb{E}\left[\mathbf{I}\{\mathbf{w}^\top \mathbf{x}_i \geq 0,\ \mathbf{w}^\top \mathbf{x}_j \geq 0\}\right].$$

Thus we need only show that

$$\mathbb{E}\left[\mathbf{I}\{\mathbf{w}^\top \mathbf{x}_i \geq 0,\ \mathbf{w}^\top \mathbf{x}_j \geq 0\}\right] = \frac{\pi - \arccos(\mathbf{x}_i^\top \mathbf{x}_j)}{2\pi}.$$

Since the sign of $\mathbf{w}^\top \mathbf{x}_i$ depends only on the direction and not the magnitude of $\mathbf{w}$ and because the Gaussian is spherically symmetric, we may assume that $\mathbf{w}^\top$ lies on the unit sphere. The expectation of an indicator is simply a probability, so

$$\mathbb{E}\left[\mathbf{I}\{\mathbf{w}^\top \mathbf{x}_i \geq 0,\ \mathbf{w}^\top \mathbf{x}_j \geq 0\}\right] = \mathbb{P}[\mathbf{w}^\top \mathbf{x}_i \geq 0 \text{ and } \mathbf{w}^\top \mathbf{x}_j \geq 0].$$

The inner product $\mathbf{w}^\top \mathbf{x}_i$ is nonnegative if and only if the angle between $\mathbf{w}$ and $\mathbf{x}$ is less than or equal to $\frac{\pi}{2}$. Thus

$$\mathbb{P}[\mathbf{w}^\top \mathbf{x}_i \geq 0] = \frac{\text{Measure of } S^{d-1} \cap H_{\mathbf{x}_i}}{\text{Measure of } S^{d-1}} = \frac{1}{2}$$

where $H_{\mathbf{x}_i} = \{\mathbf{x} \in \mathbb{R}^d \mid \mathbf{x}^\top \mathbf{x}_i \geq 0\}$ is the closed halfspace determined by the hyperplane in $\mathbb{R}^d$ orthogonal to $\mathbf{x}_i$. Likewise,

$$\mathbb{P}[\mathbf{w}^\top \mathbf{x}_i \geq 0 \text{ and } \mathbf{w}^\top \mathbf{x}_j \geq 0] = \frac{\text{Measure of } S^{d-1} \cap H_{\mathbf{x}_i} \cap H_{\mathbf{x}_j}}{\text{Measure of } S^{d-1}}.$$

If $V = \text{span}_{\mathbb{R}}(x_i, x_j)$ is the plane spanned by $\mathbf{x}_i$ and $\mathbf{x}_j$ in $\mathbb{R}^d$, then $V \cap S^{d-1}$ is a copy of S^1. The set of viable $\mathbf{w} \in V \cap S^{d-1}$ is an arc of length $\frac{\pi - \theta}{2\pi}$, as illustrated in Figure 1

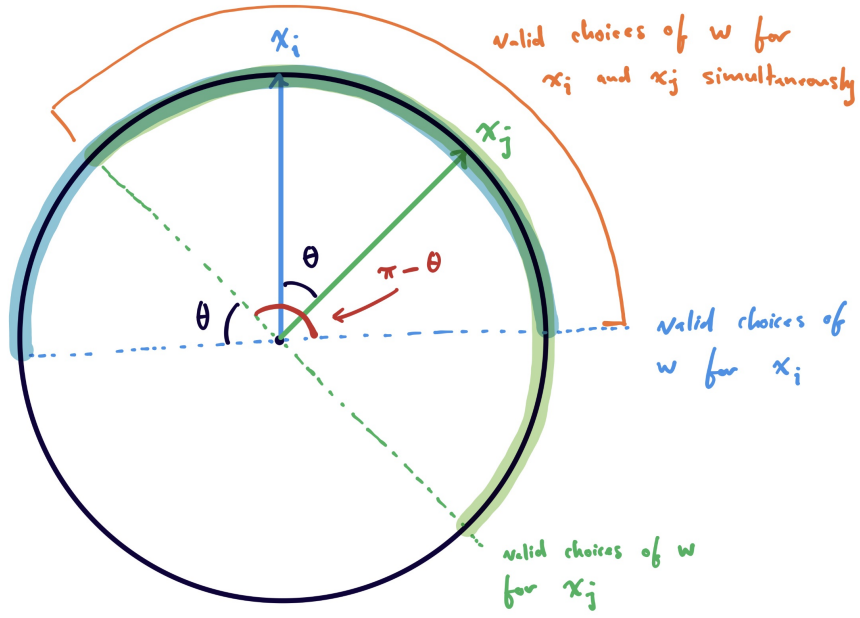

Figure 1: Feasible $\mathbf{w}$ in $V \cap S^{d-1}$

All other feasible choices of $\mathbf{w}$ lie on $\phi(V) \cap S^{d-1}$, where ϕ is translation by some vector in $V^{\perp}$: the case of S^2 is illustrated in Figure 2.

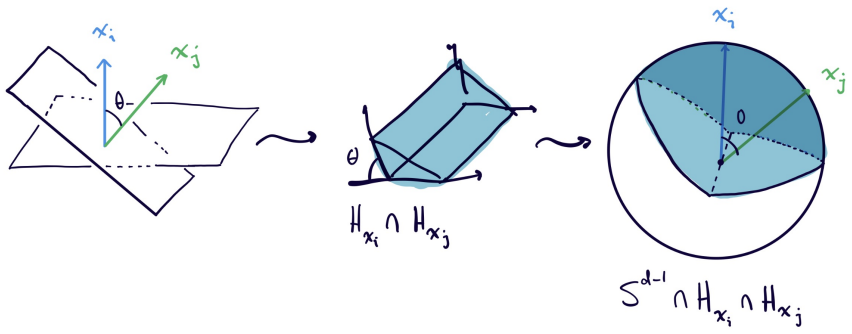

Figure 2: Feasible $\mathbf{w}$ in $V \cap S^{d-1}$

Hence

$$\mathbb{P}[\mathbf{w}^\top \mathbf{x}_i \geq 0 \text{ and } \mathbf{w}^\top \mathbf{x}_j \geq 0] = \frac{\text{Measure of } S^{d-1} \cap H_{\mathbf{x}_i} \cap H_{\mathbf{x}_j}}{\text{Measure of } S^{d-1}}$$
$$= \frac{A\frac{\pi-\theta}{2\pi}}{A} = \frac{\pi-\theta}{2\pi},$$

where θ is the angle between $\mathbf{x}_i$ and $\mathbf{x}_j$ and A is the surface area of S^{d-1} in $\mathbb{R}^d$. Since $\mathbf{x}_i$ and $\mathbf{x}_j$ are

normalized, $\theta = \arccos(\mathbf{x}_i^\top \mathbf{x}_j)$ and hence

$$\mathbb{E}\left[\mathbf{x}_i^\top \mathbf{x}_j \cdot \mathbf{I}\{\mathbf{w}^\top \mathbf{x}_i \geq 0,\ \mathbf{w}^\top \mathbf{x}_j \geq 0\}\right] = \mathbf{x}_i^\top \mathbf{x}_j \frac{\pi - \arccos(\mathbf{x}_i^\top \mathbf{x}_j)}{2\pi}$$

as desired.

□

EXERCISE 4.

(a) Create and train a simple deep learning network consisting of a convolution level with pooling, a fully connected level, and then softmax. Keep the network small. Use 20 channels for the convolution level and 100 gates for the fully connected level. For input data, use the MNIST data set with 28×28 images of digits.

(b) Create and train a second network with two fully connected levels, the first level with 200 gates and the second level with 100 gates. How does the accuracy of this network compare to the first?

Proof: In both parts of this problem we used a learning rate of 0.001, a batch size of 64 and a total of 5 epochs.

(a) This network consisted of a single convolution layer with 20 channels composed with leaky relu followed by max pooling and a single fully connected layer. After 5 epochs it correctly classified 58,083 out of 60,000 samples of the training data and 9650 out of 10,000 samples of the test data for accuracy ratings of 96.80 and 96.50 percent respectively. Note that we did not add a softmax at the end of our network as we used cross entropy loss for our loss function which, in pytorch, implements softmax itself. An additional use of softmax resulted in a vanishing gradient, hence we omitted it.

(b) This architecture consisted of two fully connected layers, both composed with leaky relu. This actually performed slightly better than the convolutional network, correctly classifying 59,478 out of 60,000 training data points and 9762 out of 10,000 test data points for accuracies of 99.13 percent and 97.62 percent respectively. I expected the convolutional neural net to perform better, and don't yet have a satisfactory explanation for these results. The code can be seen at the end of this pdf.

□

EXERCISE 5. (Apologies that this is longer than a single page, but the problem warrants a description.) An **Ising Graph** consists of a finite set $G = [1..g] \subset \mathbb{N}$ together with parameters $h \in \mathbb{R}^G$ called the local biases and $J \in \mathbb{R}^{G\times G}$ called the interaction terms, where $J_{i,j} = J_{j,i}$ and $J_{i,i} = 0$. It comes equipped with a Hamiltonian function

$$H : S^G \longrightarrow \mathbb{R}, \qquad H(s) = \sum_{i\in G} h_i s_i \; + \; \frac{1}{2} \sum_{i,j\in G\times G} J_{i,j} s_i s_j,$$

where $S = \{-1, 1\}$. The space S^G is the set of all functions $G \longrightarrow S$ and is called the *spin space* of G, whereas an individual component $s(i)$ is called the *spin* of s at vertex s. This collection of data is a simple graph in the sense that G is a vertex set and J an edge matrix which does not allow self connections.

In physics, one is typically given an Ising Graph and wishes to understand its dynamics. In particular, one is interested in finding the local minima of the Hamiltonian function. This is known as the *Ising problem.* In the *reverse Ising problem*, one is instead given a collection of spins $X \subset S^G$ and wishes to find h and J such that the spin states in X are local minima.

Example 1.1. Set $G = 1, 2, 3$, $N = 1, 2$, and $M = 3$. Decompose S^G As $S^N \times S^M$ and let $X = \{(1, 1, 1), (1, -1, -1), (-1, 1, -1), (-1, -1, -1)\}$. Then for $h_3 = 1$, $J_{1,3} = -1$ and $J_{2,3} = -1$ each spin in X is a "local minima", in the sense that

$$\begin{aligned} H(1,1,1) < H(1,1,-1), \qquad & H(1,-1,-1) < H(1,-1,1), \\ H(-1,1,-1) < H(-1,1,1), \qquad & H(-1,-1,-1) < H(-1,-1,1). \end{aligned}$$

Notice that in the previous example we have abused the notion of a "local minima" slightly; we say that a pair $(s, t) \in S^N \times S^M$ is a local minimum if $H(s, t) < H(s, t')$ for all $t \neq t' \in S^M$. To avoid confusion, we will instead say that (s, t) is the minimum of the level $L_s := \{s\} \times S^M \subseteq S^N \times S^M$.

Notice also that the set X is precisely the truth table for AND with -1 in the place of 0:

s_1	s_2	s_3
1	1	1
1	-1	-1
-1	1	-1
-1	-1	-1

We think of those spins in S^N as *input spins* and those in S^M as output spins. If we had some way to magically fix a spin $s \in S^N$ while allowing the Ising dynamics to affect the vertices in M, then we could build circuits out of this model. Indeed, if $f : S^N \longrightarrow S^M$ is some function and $X = \{(s, f(s)) \in S^N \times S^M \;\mid\; s \in S^N\}$ is the graph of f then finding an Ising circuit which models f is akin to solving the following optimization problem:

$$\text{find } \; h, J \quad \text{such that } \quad H(s, f(s)) < H(s, t) \quad \text{for all } s \in S^N \text{ and } t \neq f(s).$$

This is a linear programming problem with $2^{|N|\cdot(|M|-1)}$ constraints, but those constraints are often inconsistent and render the problem impossible. However, it becomes solvable if we add auxiliary spins which we ignore in the output.

Example 1.2. The XOR circuit defined by the function $f(s_1, s_2) = -s_1 \cdot s_2$ is infeasible. However, if we instead write $G = \{1, 2, 3, 4\}$, $N = 1, 2$, $M = \{3\}$ and $A = \{4\}$ and decompose the spin space $S^G = S^N \times S^M \times S^A$, then the function

$$\begin{aligned}(1, 1) &\mapsto (-1, 1)\\ (1, -1) &\mapsto (1, 1)\\ (-1, 1) &\mapsto (1, -1)\\ (-1, -1) &\mapsto (-1, 1)\end{aligned}$$

is both feasible and recovers the XOR circuit when the spin s_4 is ignored.

Thus, the reverse Ising problem can be solved by adding some number of auxiliary spins A to the circuit. Unfortunately, this both increases the number of constraints exponentially (there are now $2^{|N|(|M|+|A|-1)}$ many) and introduces nonlinearity due to the added challenge of choosing an appropriate auxiliary spin for each input.

In this project, we wish to investigate the $N_1 \times N_2$ Ising multiply circuits $\mathrm{MUL}_{N_1 \times N_2}$ for $N_1, N_2 \leq 8$. Preliminary theoretical results suggest that embedding S^G into the higher dimensional spin space S^V where $V = G \cup \{(i, j) \in S^{G \times G} \mid i < j\}$ and clustering with respect to Hamming distance might yield a way to make informed guesses of feasible auxiliary values. We will test our proposed technique against available data for $N_1, N_2 \in \{2, 3\}$ and if successful, proceed to investigate $\mathrm{MUL}_{N_1 \times N_2}$ for higher values of N_1 and N_2.

```
# Imports
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F  # relu, tanh
from torch.utils.data import DataLoader  # easier dataset management
import torchvision.datasets as datasets  #
import torchvision.transforms as transforms


# neural net for problem 4a
class NN4A(nn.Module):
    # this initializes all the neural net layers
    def __init__(self, num_classes):
        super(NN4A, self).__init__()
        # the convolution
        self.conv1 = nn.LazyConv2d(
            out_channels=20,
            kernel_size=(3, 3),
            stride=(3, 3),
            padding=(0, 0),
        )  # this is called a "same convolution"

        # this is the pooling
        self.pool = nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))

        # this turns a shape (64, 20, 4, 4) tensor into a shape (64, 320) tensor
        self.flatten = nn.Flatten(start_dim=1)
        # fully connected layer 1
        self.fc1 = nn.LazyLinear(100)
        # fully connected layer 2
        self.fc2 = nn.LazyLinear(num_classes)
        # this is just so we can see the shape of the vector through one pass of the network
        self._num = 0

        # NOTE
        # WE DON'T DO A SOFTMAX AT THE END SINCE IT'S INCLUDED IN OUR CHOICE OF LOSS FUNCTION

    # this runs a data point x through the neural net once
    def forward(self, x):
        # print one pass through the network
        if self._num == 0:
            self.shapeprint(x)
```

```
            self._num += 1

        x = F.leaky_relu(self.conv1(x))
        x = self.pool(x)
        x = self.flatten(x)
        x = F.leaky_relu(self.fc1(x))
        x = self.fc2(x)

        return x

    def shapeprint(self, x):
        x = F.leaky_relu(self.conv1(x))
        print("shape after conv:", x.shape)
        x = self.pool(x)
        print("shape after pool:", x.shape)
        x = self.flatten(x)
        print("shape after pool:", x.shape)
        x = F.leaky_relu(self.fc1(x))
        print("shape after fc1:", x.shape)
        x = self.fc2(x)
        print("shape after fc2:", x.shape)


# neural net for problem 4a
class NN4B(nn.Module):
    # this initializes all the neural net layers
    def __init__(self, input_size, num_classes):
        super(NN4B, self).__init__()
        self.flatten = nn.Flatten(start_dim=1)
        self.fc1 = nn.Linear(input_size, 200)
        self.fc2 = nn.LazyLinear(100)
        self.fc3 = nn.LazyLinear(num_classes)

        # NOTE
        # WE DON'T DO A SOFTMAX AT THE END SINCE IT'S INCLUDED IN OUR CHOICE OF LOSS FUNCTION

    # this runs a data point x through the neural net once
    def forward(self, x):
        x = self.flatten(x)
        x = F.leaky_relu(self.fc1(x))
        x = F.leaky_relu(self.fc2(x))
        x = self.fc3(x)
```

```
        return x


# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Hyperparameters
inchannels = 1
num_classes = 10
learning_rate = 0.001
batch_size = 64
num_epochs = 5

# Load Data

# all we use transform for is to convert data to tensor
train_dataset = datasets.MNIST(
    root="dataset/", train=True, transform=transforms.ToTensor(), download=True
)
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_dataset = datasets.MNIST(
    root="dataset/", train=False, transform=transforms.ToTensor(), download=True
)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=True)

# Initialize Network
model_4a = NN4A(num_classes=num_classes).to(device)
model_4b = NN4B(input_size=784, num_classes=num_classes).to(device)

# Loss and optimizer 4A
criterion_4a = nn.CrossEntropyLoss()
optimizer_4a = optim.Adam(model_4a.parameters(), lr=learning_rate)

# Loss and optimizer 4B
criterion_4b = nn.CrossEntropyLoss()
optimizer_4b = optim.Adam(model_4b.parameters(), lr=learning_rate)

# Train Network

print("Device used: {}".format(device))


def train4a():
```

```
    print("\nTRAINING 4A\n-------------\n")
    for epoch in range(num_epochs):
        print("4A EPOCH: ", epoch)
        for batch_idx, (data, targets) in enumerate(train_loader):
            data = data.to(device=device)
            targets = targets.to(device=device)

            # forward
            scores4a = model_4a(data)
            loss4a = criterion_4a(scores4a, targets)

            # backward
            optimizer_4a.zero_grad()
            loss4a.backward()

            # gradient descent or adam step
            optimizer_4a.step()  # update the weights depending on loss computed in loss.backward


# train 4B
def train4b():
    print("TRAINING 4B")
    for epoch in range(num_epochs):
        print("4B EPOCH: ", epoch)
        for batch_idx, (data, targets) in enumerate(train_loader):
            data = data.to(device=device)
            targets = targets.to(device=device)

            # forward
            scores4b = model_4b(data)
            loss4b = criterion_4b(scores4b, targets)

            # backward
            optimizer_4b.zero_grad()
            loss4b.backward()

            # gradient descent or adam step
            optimizer_4b.step()  # update the weights depending on loss computed in loss.backward


# Check accuracy on training and test our Network
def check_accuracy(loader, model):
    if loader.dataset.train:
```

```
        print("Checking accuracy on training data")
    else:
        print("Checking accuracy on test data")

    num_correct = 0
    num_samples = 0
    model.eval()

    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device)
            y = y.to(device=device)
            scores = model(x)

            # 64 x 10
            _, predictions = scores.max(1)
            num_correct += (predictions == y).sum()
            num_samples += predictions.size(0)

        print(
            f"Got {num_correct} / {num_samples} with accuracy {float(num_correct)/float(num_sampl
        )
    model.train()


train4a()
train4b()

print("############################\n## PROBLEM 4A\n############################")
check_accuracy(train_loader, model_4a)
check_accuracy(test_loader, model_4a)
print("\n\n############################\n## PROBLEM 4B\n############################")
check_accuracy(train_loader, model_4b)
check_accuracy(test_loader, model_4b)
```