CPSC 340 Assignment 6

Important: Submission Format [5 points]

Please make sure to follow the submission instructions posted on the course website. We will deduct marks if the submission format is incorrect, or if you're not using LaTeX and your submission is *at all* difficult to read – at least these 5 points, more for egregious issues. Compared to assignment 1, your name and student number are no longer necessary (though it's not a bad idea to include them just in case, especially if you're doing the assignment with a partner).

1 Robust PCA for Background Subtraction

If you run `python main -q 1`, the program will load a dataset X where each row contains the pixels from a single frame of a video of a highway. The demo applies PCA to this dataset and then uses this to reconstruct the original image. It then shows the following 3 images for each frame:

1. The original frame.
2. The reconstruction based on PCA.
3. A binary image showing locations where the reconstruction error is non-trivial.

Recently, latent-factor models have been proposed as a strategy for "background subtraction": trying to separate objects from their background. In this case, the background is the highway and the objects are the cars on the highway.

Why does this make any sense? Remember that PCA tries to find principal components that reconstruct the dataset well. Here the training examples are highway images. Now, we need to reconstruct these highway images out of a small number of PC images. Since we want to reconstruct them well, we're going to create PCs that capture the *common features across the entire dataset*. In other words, the PCs will try to reconstruct the highway and slight variations of the highway, because that is always there. If a car appears in the top-left of only a couple images, we might not want a PC for that case, since that PC is useless for the majority of the training set. Thus, when we try to reconstruct the images we are left to see only the parts of the image that can be made out of the PCs, or in other words that are common across all the images. Hence why the reconstructions look like the background. We can then subtract this reconstruction (background) from the original image to get objects of interest (foreground). Cool, right?

In this demo, we see that PCA does an OK job of identifying the cars on the highway in that it does tend to identify the locations of cars. However, the results aren't great as it identifies quite a few irrelevant parts of the image as objects. Who likes good news — everyone? Well good news, everyone! We can use robust PCA to do even better.

Robust PCA is a variation on PCA where we replace the L2-norm with the L1-norm,

$$f(Z,W) = \sum_{i=1}^{n}\sum_{j=1}^{d} |\langle w^j, z_i\rangle - x_{ij}|,$$

and it has recently been proposed as a more effective model for background subtraction.

1.1 Why Robust PCA [5 points]

In a few sentences, explain why using a robust loss might help PCA perform better for this background subtraction application. Some conversation-starters: what is an outlier here — a car or the highway? What does it mean to be robust to outliers — that we're good at reconstructing them or that we're back at reconstructing them?

Answer: The robust loss makes us robust to outliers in our data compared to a squared loss, specifically because the "penalty" of each reconstruction is **exactly proportional to the magnitude of its reconstruction loss**. Here, the outliers are the cars, as they are not a uniformly shared part of the dataset. Instead of prioritizing reducing the reconstruction error (making the reconstruction better) for areas where there are cars (outliers), the robust loss instead focuses on overall loss equally, and hence does better as a background subtraction tool.

1.2 Robust PCA Approximation and Gradient [5 points]

If you run `python main.py -q 1.3` , the program will repeat the same procedure as in the above section, but will attempt to use the robust PCA method, whose objective functions are yet to be implemented. You will need to implement it (yay!).

We'll use a gradient-based approach to PCA and a smooth approximation to the L1-norm. In this case the log-sum-exp approximation to the absolute value may be hard to get working due to numerical issues. Perhaps the Huber loss would work. We'll use the "multi-quadric" approximation:

$$|\alpha| \approx \sqrt{\alpha^2 + \epsilon},$$

where ϵ controls the accuracy of the approximation (a typical value of ϵ is 0.0001). Note that when $\epsilon = 0$ we revert back to the absolute value, but when $\epsilon > 0$ the function becomes smooth.

Our smoothed loss is:

$$f(Z,W) = \sum_{i=1}^{n}\sum_{j=1}^{d}\sqrt{(\langle w^j, z_i\rangle - x_{ij})^2 + \epsilon}$$

The partial derivatives of this loss with respect to the elements of W are (this derivation has been corrected since first posting, though the final answer was always right):

$$\begin{aligned}
\frac{\partial}{\partial w_{cj}} f(Z,W) &= \frac{\partial}{\partial w_{cj}} \sum_{i=1}^{n}\sum_{j'=1}^{d}\left((\langle w^{j'}, z_i\rangle - x_{ij'})^2 + \epsilon\right)^{\frac{1}{2}} \\
&= \sum_{i=1}^{n} \frac{\partial}{\partial w_{cj}} \left((\langle w^{j}, z_i\rangle - x_{ij})^2 + \epsilon\right)^{\frac{1}{2}} \qquad \text{(since the } j' \neq j \text{ terms have no } w_{cj} \text{ in them)} \\
&= \sum_{i=1}^{n} \frac{1}{2}\left((\langle w^{j}, z_i\rangle - x_{ij})^2 + \epsilon\right)^{-\frac{1}{2}} \frac{\partial}{\partial w_{cj}} \left((\langle w^{j}, z_i\rangle - x_{ij})^2 + \epsilon\right) \\
&= \sum_{i=1}^{n} \frac{1}{2}\left((\langle w^{j}, z_i\rangle - x_{ij})^2 + \epsilon\right)^{-\frac{1}{2}} \, 2\left(\langle w^{j}, z_i\rangle - x_{ij}\right) \frac{\partial}{\partial w_{cj}} \langle w^j, z_i\rangle \\
&= \sum_{i=1}^{n} \left((\langle w^{j}, z_i\rangle - x_{ij})^2 + \epsilon\right)^{-\frac{1}{2}} \left(\langle w^{j}, z_i\rangle - x_{ij}\right) z_{ic}
\end{aligned}$$

The partial derivatives with respect to Z are similar:

$$\begin{aligned}\frac{\partial}{\partial z_{ic}} f(Z,W) &= \frac{\partial}{\partial z_{ic}} \sum_{i'=1}^{n}\sum_{j=1}^{d}\left((\langle w^j, z_{i'}\rangle - x_{i'j})^2+\epsilon\right)^{\frac{1}{2}} \\ &= \sum_{j=1}^{d}\left((\langle w^j, z_i\rangle - x_{ij})^2+\epsilon\right)^{-\frac{1}{2}} (\langle w^j, z_i\rangle - x_{ij})\, \frac{\partial}{\partial z_{ic}}\langle w^j, z_i\rangle \\ &= \sum_{j=1}^{d}\left((\langle w^j, z_i\rangle - x_{ij})^2+\epsilon\right)^{-\frac{1}{2}} (\langle w^j, z_i\rangle - x_{ij})\, w_{cj}\end{aligned}$$

If we put this into matrix(ish) notation, we get the following:

$$\nabla_W f(Z,W) = Z^T\left[R \oslash \left(R^{\circ 2}+\epsilon\right)^{\circ\frac{1}{2}}\right]$$

where $R \equiv ZW - X$, $A \oslash B$ denotes **element-wise** division of A and B, $A + s$ for a scalar s denotes element-wise adding s to each entry of A, and $A^{\circ p}$ denotes taking A to the **element-wise** power of p.

And, similarly, the gradient with respect to Z is given by:

$$\nabla_Z f(Z,W) = \left[R \oslash \left(R^{\circ 2}+\epsilon\right)^{\circ\frac{1}{2}}\right] W^T$$

Show that the two parts of the gradient above, $\nabla_W f(Z,W)$ and $\nabla_Z f(Z,W)$, have the expected dimensions.

Answer: To do this, we can examine the dimensions of the result of each operation incrementally. First, *a priori*, we know that if $X = n \times d$, then $Z = n \times k$ and $W = k \times d$. Then, $R = ZW - X$ is also $n \times d$.

Then, when we do element-wise operations, like $R' = R \oslash \left(R^{\circ 2}+\epsilon\right)^{\circ\frac{1}{2}}$, the dimensions of our result is not different than the initial input. Hence, the dimensions of R' are still $n \times d$.

Lastly, then, $\nabla_W f(Z,W) = Z^T \times R'$ will be $(k \times n) \times (n \times d) = (k \times d)$, which matches the dimensions of W; Similarly, $\nabla_Z f(Z,W) = R' \times W^T$ will be $(n \times d) \times (d \times k) = (n \times k)$, also the dimensions of Z.

1.3 Robust PCA Implementation [10 points]

In `fun_obj.py`, you will find classes named `RobustPCAFactorsLoss` and `RobustPCAFeaturesLoss` which should compute the gradients with respect to W and Z, respectively. Complete the `evaluate()` method for each using the smooth approximation and gradient given above. Submit (1) your code in `fun_obj.py`, and (2) one of the resulting figures.

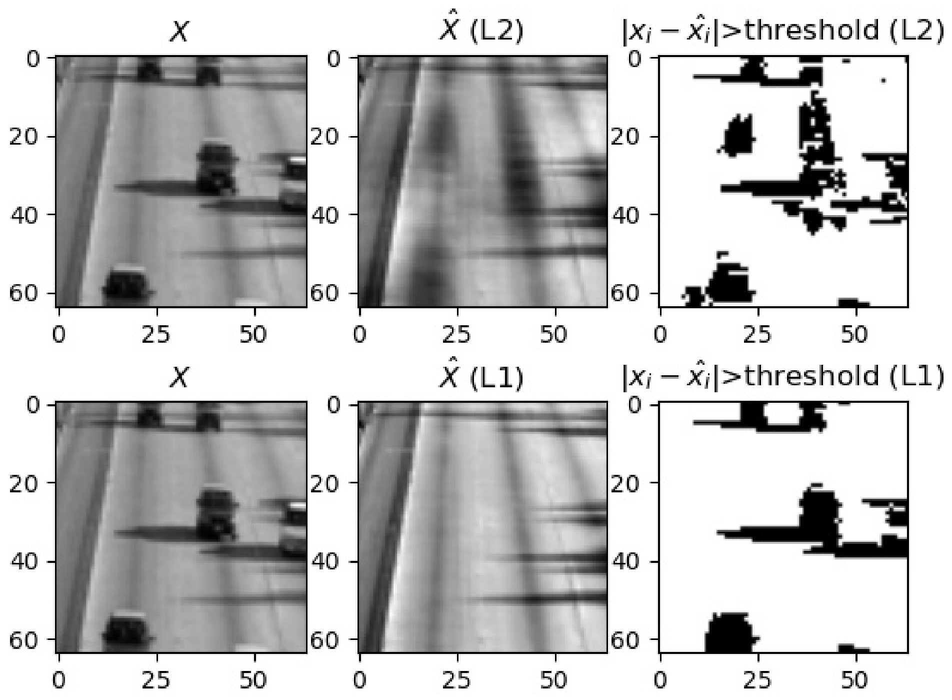
Hint: Your code will look similar to the already implemented `PCAFactorsLoss` and `PCAFeaturesLoss` classes. Note that the arguments for `evaluate()` are carefully ordered and shaped to be compatible with our optimizers.

Note: The robust PCA is somewhat sensitive to initialization, so multiple runs might be required to get a reasonable result.

Answer: Here is my code:

```
class RobustPCAFeaturesLoss(FunObj):
    def __init__(self, epsilon):
        self.epsilon = epsilon
    def evaluate(self, z, W, X):
        n, d = X.shape
        k, _ = W.shape
        Z = z.reshape(n,k)

        R = Z @ W - X
        R_squared_plus_e_sqrt = np.power(R**2 + self.epsilon, 0.5)
        f = np.sum(R_squared_plus_e_sqrt)

        g = (np.divide(R, R_squared_plus_e_sqrt)) @ W.T

        return f, g.flatten()

class RobustPCAFactorsLoss(FunObj):
    def __init__(self, epsilon):
        self.epsilon = epsilon

    def evaluate(self, w, Z, X):
        n, d = X.shape
        _, k = Z.shape
        W = w.reshape(k, d)

        R = Z @ W - X
        R_squared_plus_e_sqrt = np.power(R ** 2 + self.epsilon, 0.5)
        f = np.sum(R_squared_plus_e_sqrt)
        g = Z.T @ np.divide(R, R_squared_plus_e_sqrt)

        return f, g.flatten()
```

Here is one of my background-subtracted figures:



1.4 Reflection [6 points]

1. Very briefly comment on the results from the previous section — does robust PCA seem to do better than regular PCA for this task?

   Answer: Yes! Definitely. You can barely see the shadows of the cars.

2. How does the number of video frames and the size of each frame relate to n, d, and/or k?

   Answer: The number of video frames is n, and the size of each frame, if they are some width w by some height h, $w * h = d$.

3. What would the effect be of changing the threshold (see code) in terms of false positives (cars we identify that aren't really there) and false negatives (real cars that we fail to identify)?

   Answer: The threshold is our measure for how sensitive our car-detection is. The lower the threshold, the more sensitive we are, and the *more* things we would consider a car. So lower threshold means that we will get more false positives and less false negatives, while higher thresholds means we will get less false positives and more false negatives.

2 Movie Recommendations

If you run `python main.py -q 2`, the program will perform the following steps:

1. Loads the small educational version of the MovieLens dataset (`https://grouplens.org/datasets/movielens/`).
2. Prints out the first 5 rows of the ratings dataframe (which we'll use) and the movies dataframe (which we won't use, but is loaded FYI).
3. Transforms the ratings table into the Y matrix described in lecture.
4. Splits Y into train (80% of the ratings) and validation (20% of the ratings) matrices.

5. Prints out some stats about the dataset, including the average rating in the training set.

2.1 Understanding Y [6 points]

Answer the following questions:

1. In lecture, we used the "?" symbol to represent missing ratings. How are these missing entries of Y implemented in the code?

   Answer: They are represented by "NaN" values.

2. How many (non-missing) ratings are there in `Y_train` and `Y_valid`, respectively?

   Answer: In `Y_train`, there are 80668 non-missing ratings. In `Y_valid`, there are 20188 non-missing values. This sounds like a lot but it's not a large proportion of the full dataset.

3. Does the same user-movie rating ever appear in *both* `Y_train` and `Y_valid`? Is the result as expected?

   Answer: No, they will not ever appear in both. It is a sliced dataframe after randomizing. This **is expected** because we must separate training and testing/validating sets.

2.2 Implementing Collaborative Filtering [15 points]

If you run `python main.py -q 2.2`, the code will fit a model that doesn't actually do anything. It uses the same alternating minimization scheme as in the previous question on Robust PCA. Fill in the `evaluate` methods of the `CollaborativeFilteringWLoss` and `CollaborativeFilteringZLoss` classes in `fun_obj.py`. Submit your code.

Hint: Since we're minimizing the regularized PCA loss, your code should be quite similar to the `evaluate` methods in `PCAFeaturesLoss` and `PCAFactorsLoss`. I suggest you copy/paste those methods as a starting point. However, there are two changes you'll need to make: (1) modify the loss and gradient to account for the regularization, and (2) accounting for the fact that the loss should only sum over the available ratings, not the entire Y matrix. The easiest way to account for this is to modify the R matrix by setting all its NaN values to 0. Since R represents the residuals (or reconstruction errors), setting these to 0 says that these entries don't contribute to the loss, which is exactly what we want.

Answer: Here is my code:

```
class CollaborativeFilteringZLoss(FunObj):
    def __init__(self, lammyZ=1, lammyW=1):
        self.lammyZ = lammyZ
        self.lammyW = lammyW

    def evaluate(self, z, W, Y):

        n,d = Y.shape
        k,d= W.shape

        Z = z.reshape(n,k)

        R = Z @ W - Y
        R[np.isnan(R)] = 0
        g = (R @ W.T) + self.lammyZ * Z
        f = (1/2) * np.sum(R ** 2) + (self.lammyZ/2) * (np.linalg.norm(Z) ** 2) + (self.
    ↪                    lammyW/2) * (np.linalg.norm(W) ** 2)
```

```
        return f, g.flatten()

class CollaborativeFilteringWLoss(FunObj):
    def __init__(self, lammyZ=1, lammyW=1):
        self.lammyZ = lammyZ
        self.lammyW = lammyW

    def evaluate(self, w, Z, Y):

        n,d = Y.shape
        _,k = Z.shape

        W = w.reshape(k,d)

        R = Z @ W - Y

        R[np.isnan(R)] = 0
        g = (Z.T @ R) + self.lammyW * W
        f = (1/2) * np.sum(R ** 2) + (self.lammyZ/2) * (np.linalg.norm(Z) ** 2) \
            + (self.lammyW/2) * (np.linalg.norm(W) ** 2)

        return f, g.flatten()
```

2.3 Hyperparameter tuning [5 points]

If you run `python main.py -q 2.3`, the program will implement a baseline model that predicts the average rating (around 3.5 stars) for all missing ratings. As you'll see from the output, this baseline has a root mean squared error (RMSE) of slightly above 1 star. The hyperparameters we gave you in the code actually do worse than this (my validation RMSE is around 1.3 stars). Adjust the hyperparameters of your collaborative filtering model, namely k, λ_W, λ_Z, until you obtain a better validation error than the baseline. Submit the hyperparameters you found as well as your training and validation RMSE.

PS: there is nothing to stop you from just guessing random hyperparameters, but it would be cool if you think about the fact that the current model overfits and adjust each of the hyperparameters in the direction that reduces the complexity of the model to combat the overfitting.

Answer: I decided to leave k to be 50, as I felt like it did not seem too-sub-optimal, and I was also mainly curious to see the effect of λ on the losses, but if I were to tune k I would have tried to increase it first without having it negatively impact our validation RMSE. I ended up using both λ_W and $\lambda_Z = 10$, and got a training RMSE of 0.57 and validation RMSE of 0.9.

2.4 Regularization with PCA [5 points]

In lecture we discussed the fact that with regularized PCA, we need to regularize both W and Z for things to make sense. Test this out empirically by setting $\lambda_Z = 0$ and $\lambda_W > 0$. Modify the code so that it prints out the Frobenius norm of W and Z after each iteration of the alternating minimization (you don't have to submit this printing code though). Describe your observations and briefly discuss.

Answer: When I did so, the Z matrix norm just kept increasing with every iteration, while W (I assume to compensate for Z) kept decreasing smaller and smaller. This did not result in very good validation or training error, and this intuitively shows the impact of how the regularizer **keeps the magnitude of the parameters small**. This is very important for lowering test/validation error as well as being good for interpretability.

3 Neural Networks

3.1 Neural Networks by Hand [7 points]

Suppose that we train a neural network with sigmoid activations and one hidden layer and obtain the following parameters (assume that we don't use any bias variables):

$$W = \begin{bmatrix} -2 & 2 & -1 \\ 1 & -2 & 0 \end{bmatrix}, v = \begin{bmatrix} 3 \\ 1 \end{bmatrix}.$$

Assuming that we are doing regression, for a training example with features $x_i^T = \begin{bmatrix} -4 & -2 & 4 \end{bmatrix}$ what are the values in this network of z_i, $h(z_i)$, and $\hat{y}_i$?

Answer: First, our x_i goes through W to get z_i:

$$z_i = \begin{bmatrix} -2 & 2 & -1 \\ 1 & -2 & 0 \end{bmatrix} \begin{bmatrix} -4 \\ -2 \\ 4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Then, z_i goes through the sigmoid activation function to get $h(z_i)$:

$$h(z_i) = \begin{bmatrix} \frac{1}{1+\exp(-0)} \\ \frac{1}{1+\exp(-0)} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \end{bmatrix}$$

Lastly, we pass $h(z_i)$ through v to get our final $\hat{y}_i$:

$$\hat{y}_i = \begin{bmatrix} 3 & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \end{bmatrix} = 2$$

3.2 Neural Networks vs. Linear Classifier [7 points]

If you run `python main.py -q 3`, the program will train a multi-class logistic regression classifier on the MNIST handwritten digits data set using stochastic gradient descent with some pre-tuned hyperparameters. The scikit-learn implementation called by the code uses a minibatch size of 1 and a one-vs-rest strategy for multi-class logistic regression. It is set to train for 10 epochs. The performance of this model is already quite good, with around 8% training and validation errors. Your task is to use a neural networks classifier to outperform this baseline model.

If you run `python main.py -q 3.2`, the program will train a single-layer neural network (one-layer nonlinear encoder paired with a linear classifier) on the same dataset using some pre-tuned hyperparameters. Modify the code, play around with the hyperparameter values (configurations of encoder layers, batch size and learning rate of SGD, standardization, etc.) until you achieve validation error that is reliably below 5%. Report (1) the training and validation errors that you obtain with your hyperparameters, and (2) the hyperparameters that you used.

Answer: In my answer, I used:

- Batch size = 1000
- Hidden layers: [60]
- Output Dimension of the layers = 30

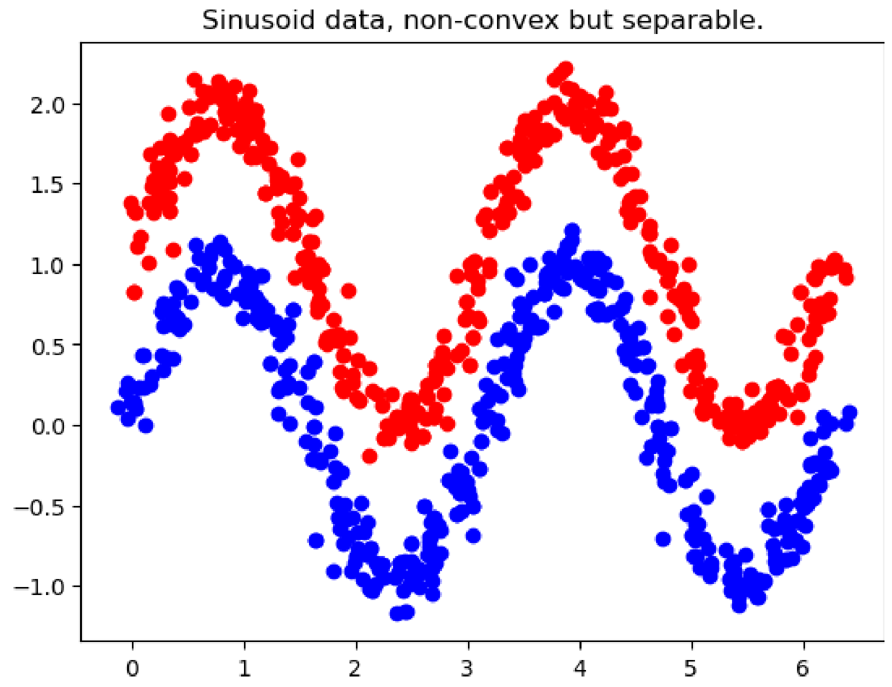
- The InvSqrt learning rate, with initial value of 0.0975

This got me a training RMSE of 0.039 and validation RMSE of 0.0477.

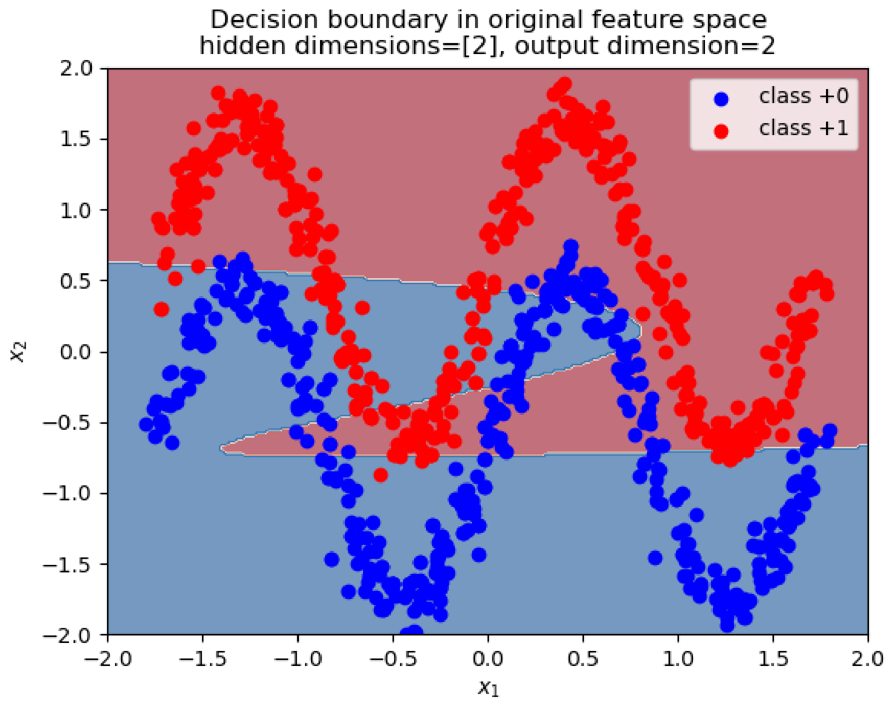
3.3 Neural Disentangling [10 points]

An intuitive way to think about a neural network is to think of it as a composition of an encoder (all the layers except the last one, including the final nonlinearity) and a linear predictor (the last layer). The predictor signals the encoder to learn a better feature space, in such a way that the final linear predictions are meaningful.

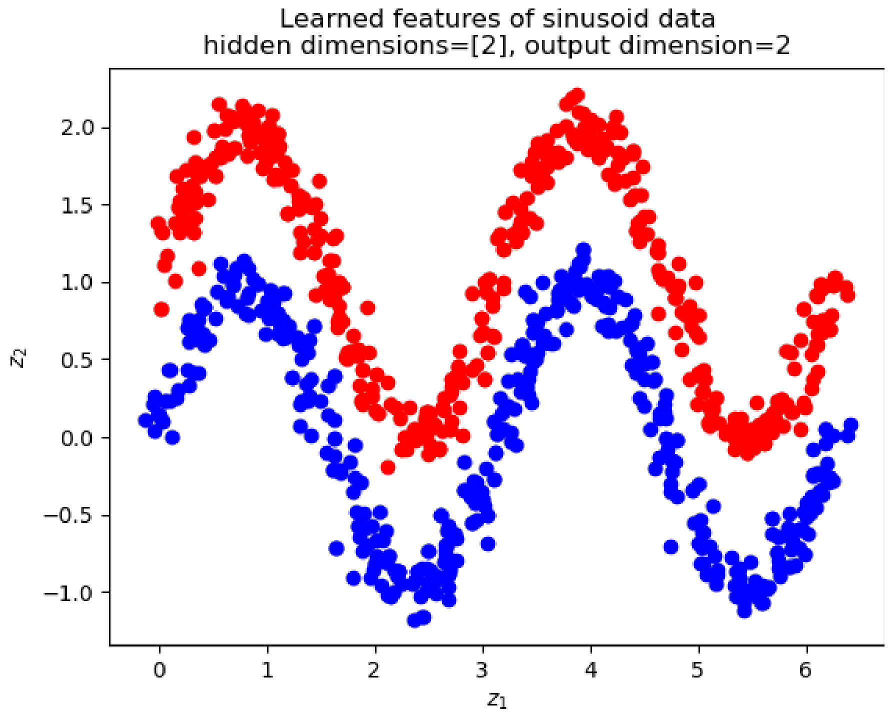
The following figures resulted from running `python main.py -q 3.3`. For this, a neural network model is used to encode the examples into a 2-dimensional feature space. Here's the original training data:



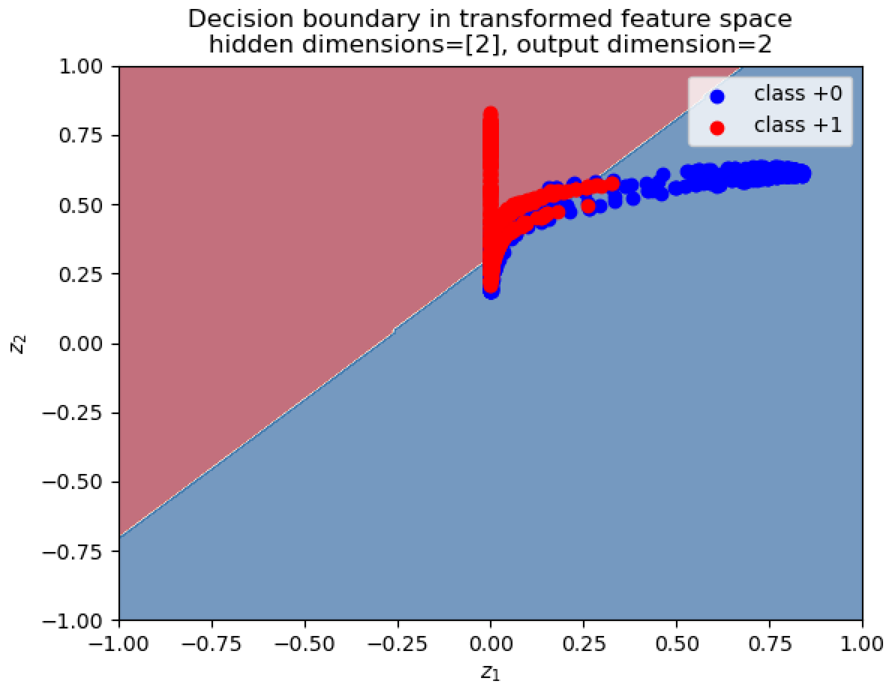
And here is the learned decision boundary of the neural network:



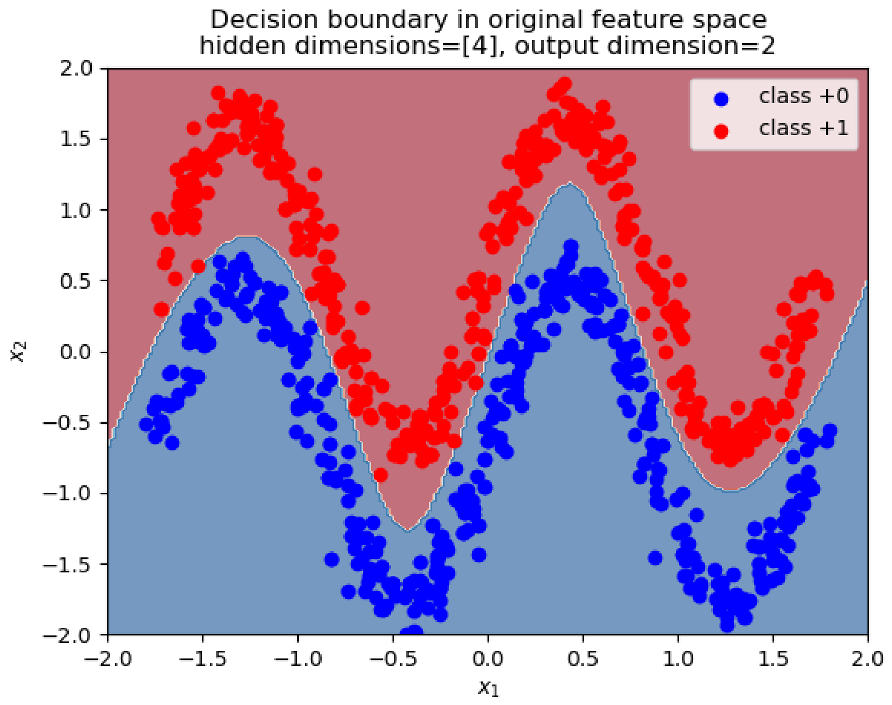
We can look inside and see what the encoder learned. Here is the training data shown in the 2D transformed feature space learned by the encoder (first layer):

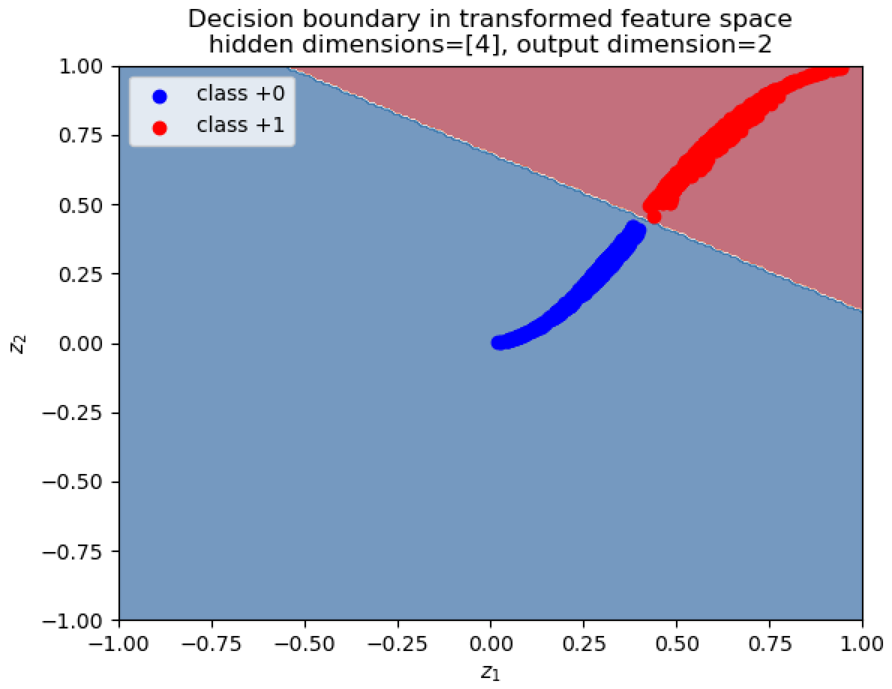


Here is the linear classifier learned by the predictor in this transformed feature space:



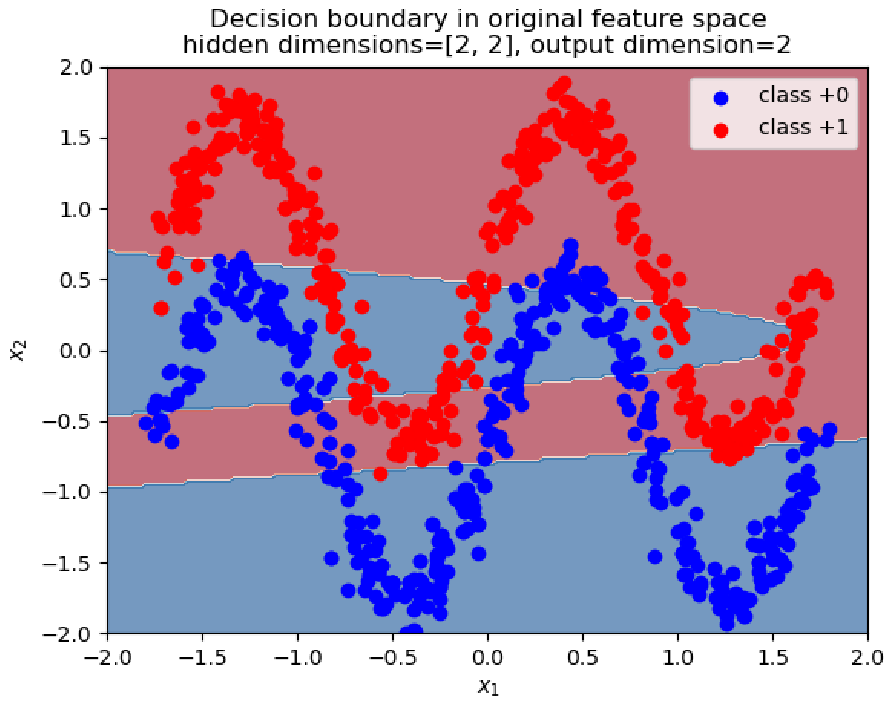
This particular neural network solution does not classify the given examples very well. However, if we use a 4-dimensional hidden features (with all else equal), we get a better-looking result:

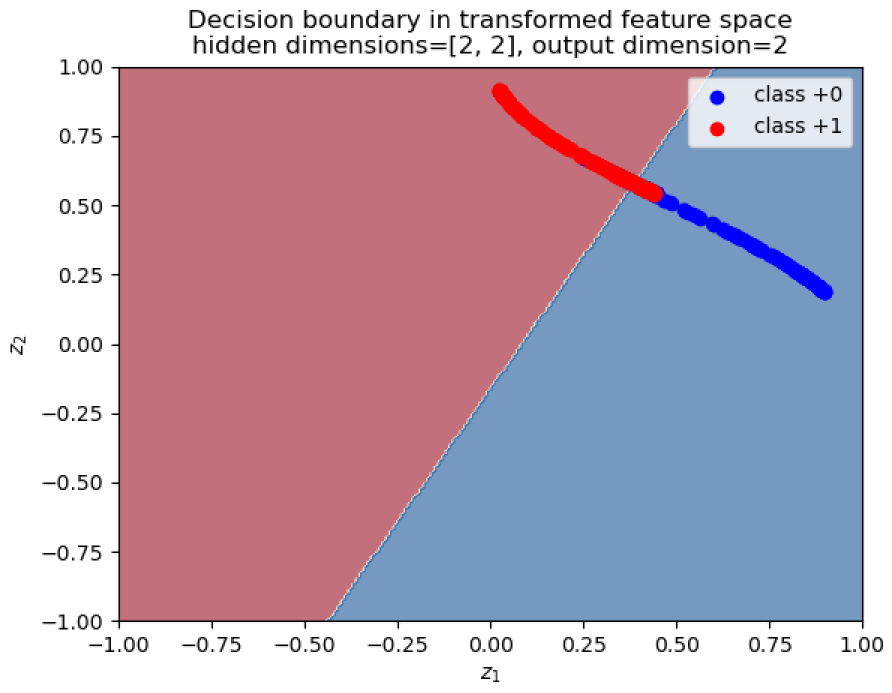




Note: in this case, since the hidden feature space has dimension 4, we are only able to plot 2 of the 4 dimensions. The actual linear classifiers is a 3D hyperplane dividing the 4D space. However, even with these two dimensions that we can plot, we can already see that the training data looks pretty much linearly separable, and we learn a good classifier.

Previously we changed the *hidden layer size* from 2 to 4. Another change we can make is to increase the *number of layers* from 1 to 2. Let's use 2 layers of hidden features of size 2:





Answer the following questions and provide a brief explanation for each:

1. Why are the axes (z_1 and z_2) of the learned feature space scatterplot constrained to $[0, 1]$? Note/hint: here, our definition of z is after the nonlinearity is applied, whereas in the lecture z is defined as before the nonlinearity is applied. Sorry about that!

   Answer: This is because the sigmoid function we used as our non-linearity, and the range of the sigmoid function is between 0 and 1.

2. Does it make sense to use neural networks on datasets whose examples are linearly separable in the original feature space?

   Answer: No, not really. We have much simpler methods that take less time and energy to train for those problems, and we can just use a linear hyperplane (of an appropriate dimension) to separate the data. Doing this simple work with a neural net may risk overfitting since it's clear what the best solution is already.

3. Why would increasing the dimensionality of the hidden features (which is a hyper-parameter) help improve the performance of our classifier?

   Answer: Qualitatively, additional dimensions may give us more "angles" to examine the data from, which can reveal more things like whether it is separable in a higher dimension space or additional features that we can use.

4. Why would increasing the number of layers help improve the performance of our classifier?

   Answer: The more layers our neural network has, the more complex the model can get, which allows us to create more complicated expressions, features, and inter-relationships than before.

5. Neural networks are known to suffer problems due to a highly non-convex objective function. What is one way we can address this problem and increase the chances of discovering a global optimum? (Hint: look at the code.)

   Answer: One thing we could do is occasionally jump out of a minimum and see if it lands us in another local minima. We would implement this by occasionally randomly making our step size larger than usual, and then continuing regular gradient descent after that. Another thing to do is use momentum, and store and utilize previous gradients to guide our next steps properly as well.

4 Very-Short Answer Questions [14 points]

Answer each of the following questions in a sentence or two.

1. Give one reason why you might want sparse solutions with a latent-factor model.

   Answer: The sparsity can lead to better interpretability, telling us exactly what latent factors contribute to what and how much they contribute. Since latent factors are often already hard to interpret (they are remnants of the complex interplays of the data), we would want to keep their solutions/weights as simple as possible to understand each their roles easier.

2. Which is better for recommending movies to a new user, collaborative filtering or content-based filtering? Briefly justify your answer.

   Answer: It's better to use content-based filtering. Collaborative filtering derives most of its understanding of the data (PCs) based on the **users in the training data** and their distributions. Collaborative filtering may find a tough time predicting preferences for a new user because it has to use the previous set of users to understand the new user.

   However, content-based methods are learning *features about movies/users*, which make them more able to apply these same features to a new user.

3. Are neural networks mainly used for supervised or unsupervised learning? Are they parametric or nonparametric?

   Answer: Neural networks (in the way that we have learnt about in our course) are usually used for supervised learning. I would consider them as parametric, we are still learning a certain number of features/weights for our model, and that number does not depend on the input data (though, in complicated neural networks, I would imagine the number of parameters and the data size could be on a similar order of magnitude.

4. Why might regularization become more important as we add layers to a neural network?

   Answer: As we add more layers, the neural network gets more complex, so the chance of overfitting increases. Regularization can help reduce overfitting by keeping our parameters small.

5. Consider using a fully-connected neural network for 5-class classification on a problem with $d = 10$. If the network has one hidden layer of size $k = 100$, how many parameters (including biases), does the network have?

   Answer: With one hidden layer of size $k = 100$, we need to transform our $d = 10$ original features into the k values. We do this by setting the weights W to be a $100 + 1 \times d$ matrix. Then, after multiplying, $z_i = Wx_i$ will then be multiplied by the V matrix which is $5 \times 100 + 1$, one for each of the z_i values. This results in a total of $101d$ total parameters in this network.

6. What is the "vanishing gradient" problem with neural networks based on sigmoid non-linearities?

   Answer: At very small or very large input values into the sigmoid function, the gradient is almost 0, which can make optimization very difficult. As layers get deeper, you may end up taking sigmoids of a sigmoids, which further "vanishes" the gradient.

7. Convolutional networks seem like a pain... why not just use regular ("fully connected") neural networks for image classification?

Answer: Convolutions allow us to get multiple “perspectives” of an image, and it provides us with extra features and not just what is physically there. Also, convolutions can be combined, learned by the network, and it overall provides a useful added complexity to the classification process.