CPSC 340 Assignment 6 (due 2021-12-06 at 11:59pm)

Pedram Amani - 73993008

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 bad at reconstructing them?

Answer: Outliers are the cars (foreground) and a good loss should be bad at reconstructing them. The robust L1 loss encourages the residuals to reach exactly zero (same as with robust regression) and therefore achieve a sparse residual image. So that in the reconstructed image, the common features will remain unaltered and the outliers will be filtered.

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:

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

For the partial derivatives with respect to Z, all of the above steps are the same except for the last one:

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

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, 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: The dimensions are as follows:

- X has dimension $n \times d$
- Z has dimension $n \times k$
- W has dimension $k \times d$
- R has dimension $n \times d$
- $\nabla_W f(Z,W)$ has dimension $(k \times n) \cdot (n \times d) = k \times d$ which is same as the dimension of W as expected
- $\nabla_Z f(Z,W)$ has dimension $(n \times d) \cdot (d \times k) = n \times k$ which is same as the dimension of Z as expected

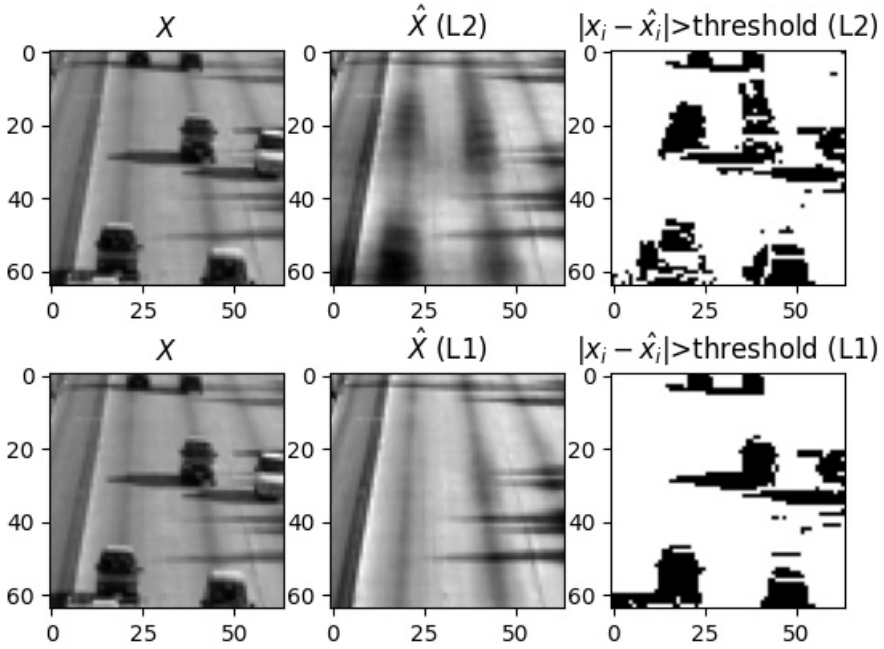
1.3 Robust PCA Implementation [10 points]

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.

```
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
        f = np.sum(np.sqrt(R ** 2 + self.epsilon))
        g = np.divide(R, np.sqrt(R ** 2 + self.epsilon)) @ 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
        f = np.sum(np.sqrt(R ** 2 + self.epsilon))
        g = Z.T @ np.divide(R, np.sqrt(R ** 2 + self.epsilon))
        return f, g.flatten()
```



First image generated by robust PCA

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. Across all the frames, the cars and their shadows are isolated by robust PCA much better than regular PCA.

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

   Answer:

   - n is the number of video frames
   - d is the total number of pixels in a frame
   - k is the number of PCs and determines model complexity

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: Increasing the threshold increases false negatives and decreasing the threshold increases false positives.

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 taken to be `np.nan`.

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

   Answer:

   - 80668 non-missing ratings in `Y_train`
   - 20168 non-missing ratings in `Y_valid`

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

   Answer: No. Obviously our training and validation sets should not and do not share any data.

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.

```
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, _ = W.shape
        Z = z.reshape(n, k)

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

```python
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
        f = np.sum(R ** 2) / 2 \
            + self.lammyZ * np.linalg.norm(Z) ** 2 / 2 \
            + self.lammyW * np.linalg.norm(W) ** 2 / 2
        g = Z.T @ R + self.lammyW * W
        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:

- $k = 3, \lambda_W = 10, \lambda_Z = 10$
- Training RMSE: 0.79
- Validation RMSE: 0.91

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: With hyper-parameters $k = 3, \lambda_W = 10, \lambda_Z = 0$, we observe the Frobenius norm of W decreasing and that of Z increasing after each iteration. Since the loss function contains ZW in the residual term, regularizing only W has almost no effect as the norm of Z can increase to compensate and keep the norm of ZW unchanged. We need to limit the norm of both matrices for regularization to be effective.

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:

$$z_i = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \; h(z_i) = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}, \; \hat{y}_i = 2$$

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:

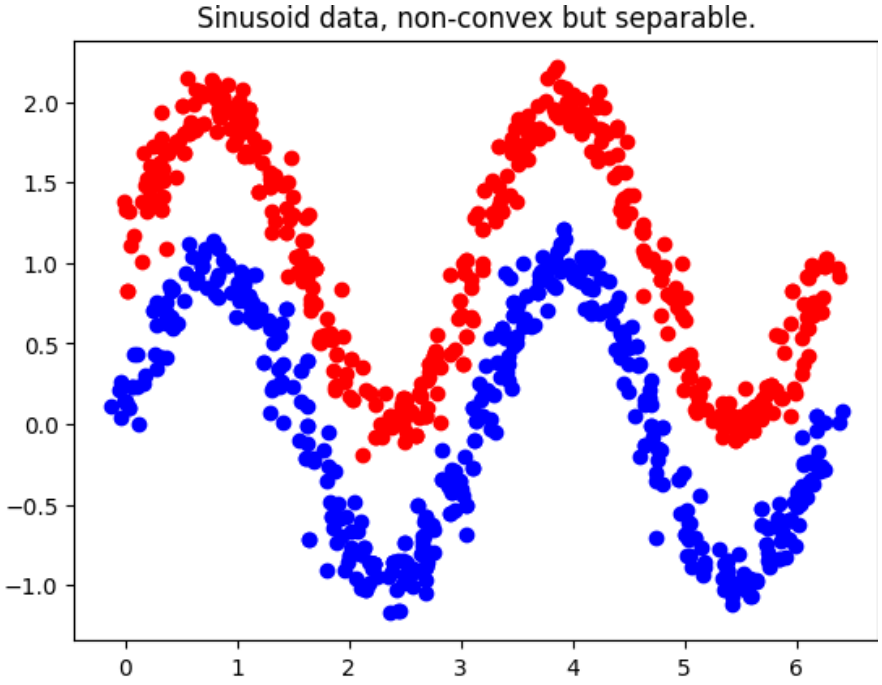
- Training error: 0.020
- Validation error: 0.028
- Hidden layer dimensions: [100]
- Output dimension: 10 (for each digit)
- Learning rate function: constant
- Learning rate: 0.005
- Regularization lambda: 0.1
- Batch size: 500
- Maximum evaluations: 20

3.3 Neural Disentangling [10 points]

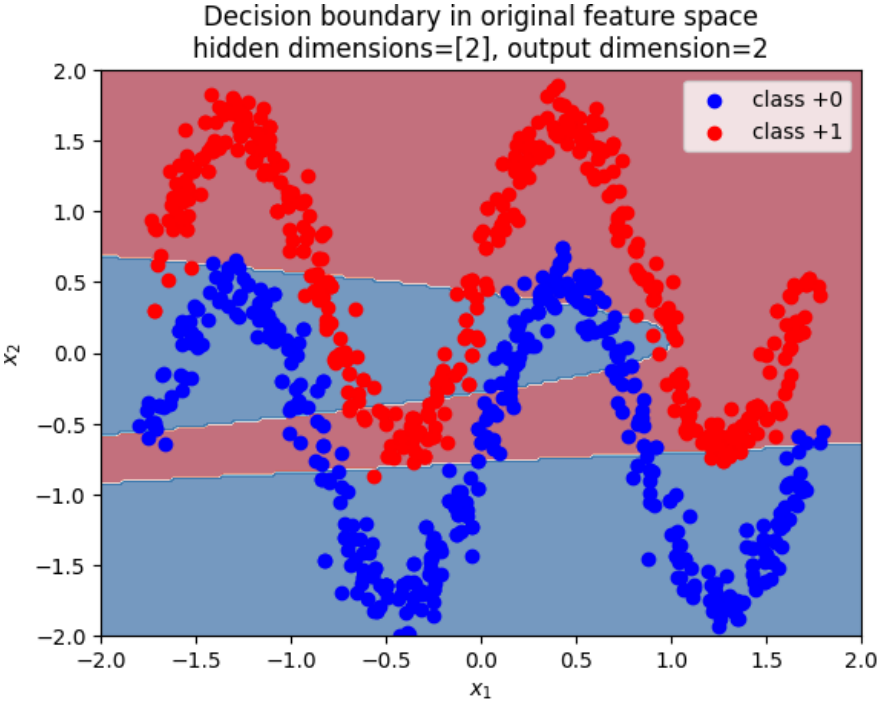
An intuitive way to think about a neural network is to think of it as a composition of an encoder (all the layers except the last one, including the final nonlinearity) and a linear predictor (the last layer). The

predictor signals the encoder to learn a better feature space, in such a way that the final linear predictions are meaningful.

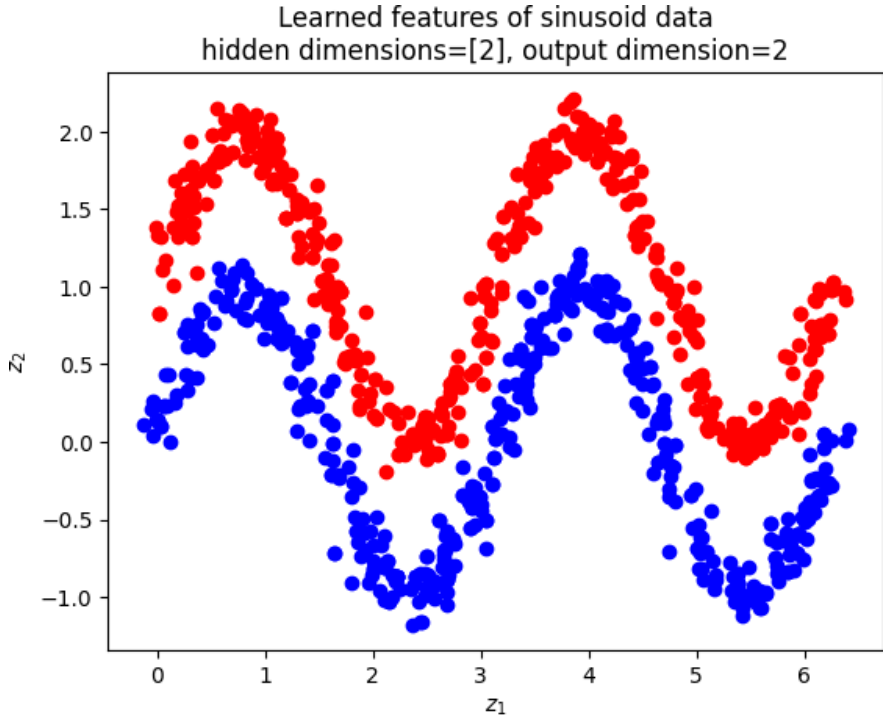
The following figures resulted from running `python main.py -q 3.3`. For this, a neural network model is used to encode the examples into a 2-dimensional feature space. Here's the original training data:



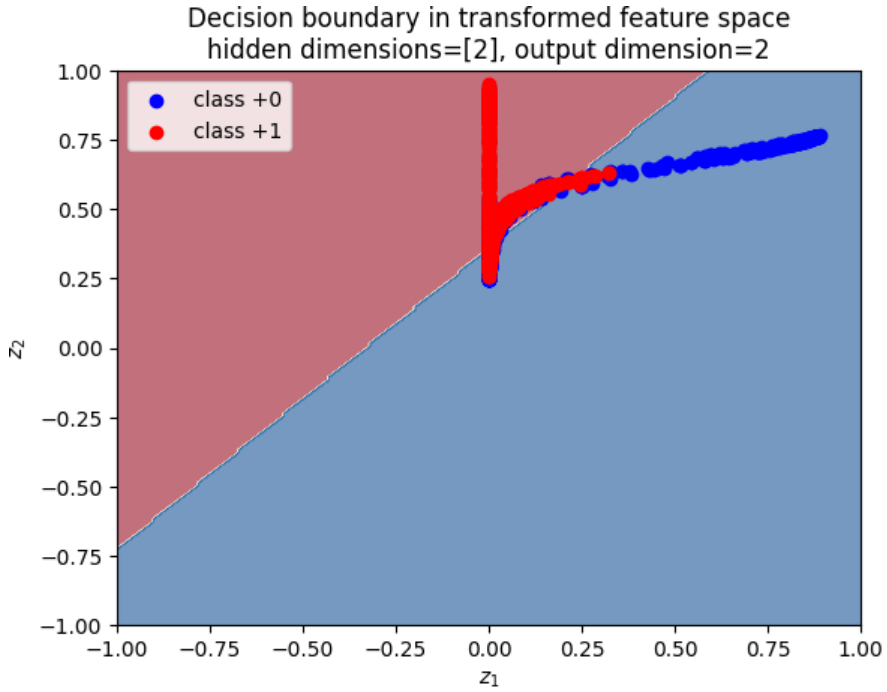
And here is the learned decision boundary of the neural network:



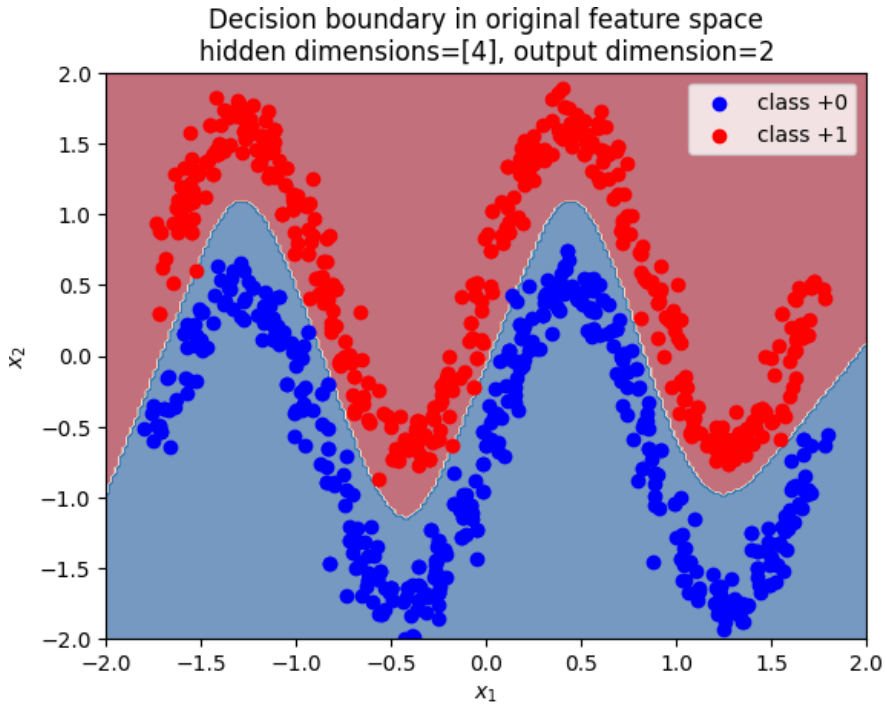
We can look inside and see what the encoder learned. Here is the training data shown in the 2D transformed feature space learned by the encoder (first layer):

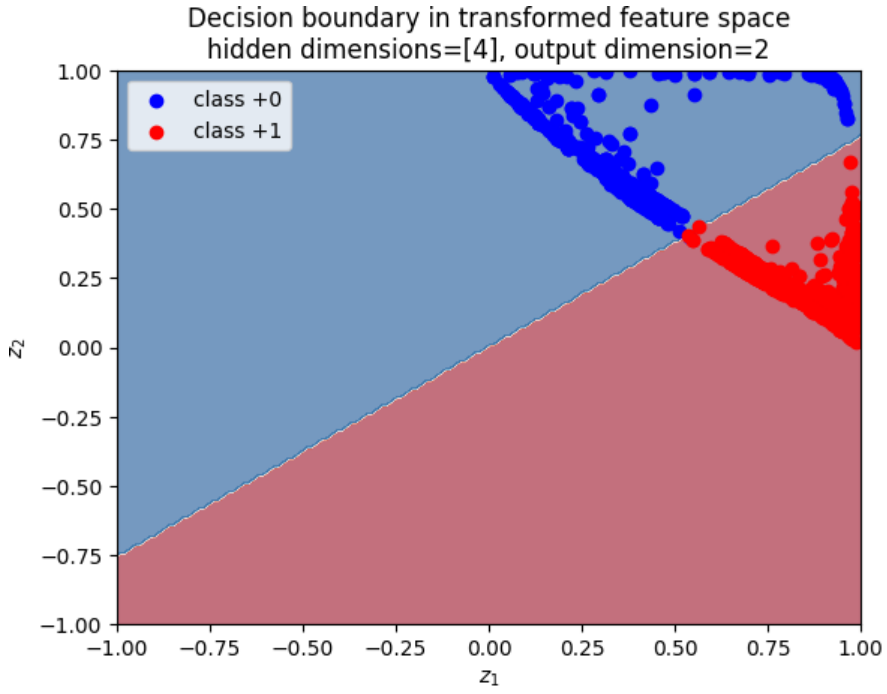


Here is the linear classifier learned by the predictor in this transformed feature space:



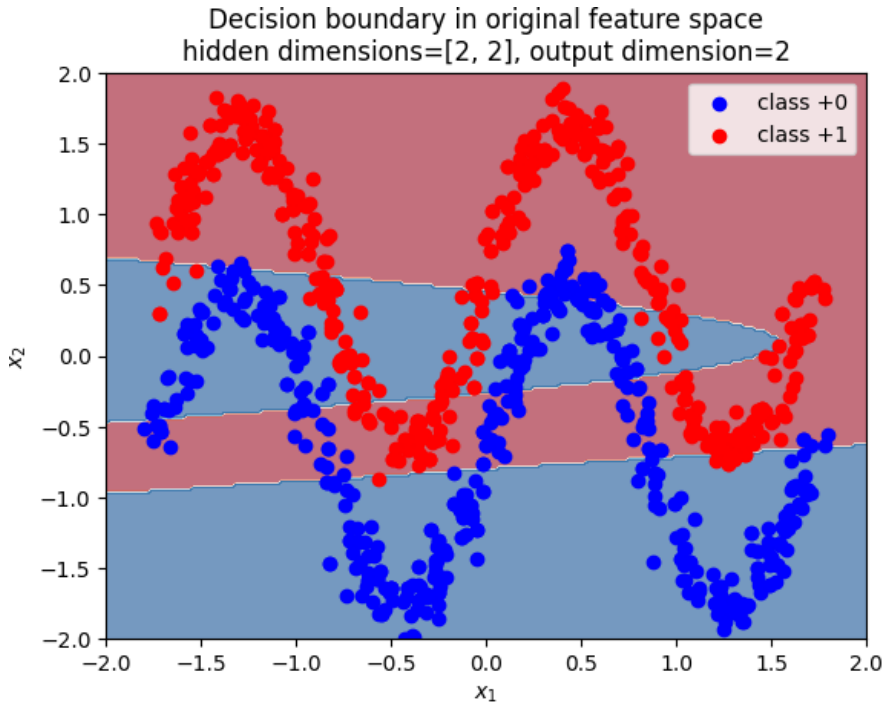
This particular neural network solution does not classify the given examples very well. However, if we use a 4-dimensional hidden features (with all else equal), we get a better-looking result:

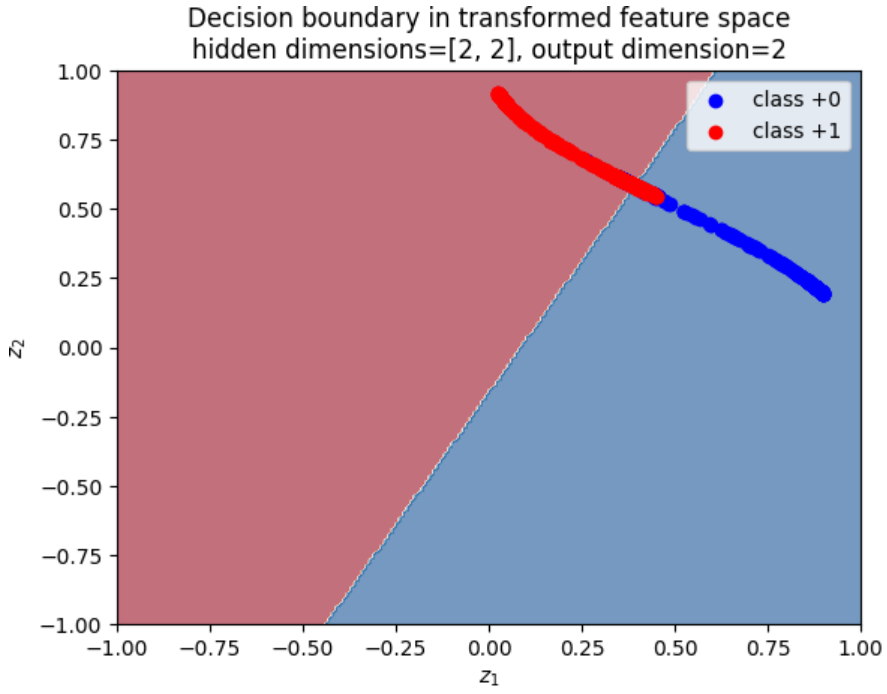




Note: in this case, since the hidden feature space has dimension 4, we are only able to plot 2 of the 4 dimensions. The actual linear classifiers is a 3D hyperplane dividing the 4D space. However, even with these two dimensions that we can plot, we can already see that the training data looks pretty 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: Since the sigmoid function has range $(0, 1)$, and z_1, z_2 are the result of the sigmoid applied on some value.

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

   Answer: No. A linear regression classifier would perform as well with less complication.

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

   Answer: A higher-dimensionality hidden feature space can model more complex aspects of the data.

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

   Answer: Since the activation functions introduce non-linearity, new layers further transform the data to a better (more separable) feature space by combining the outputs of preceding layers.

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: The method referred to as "continual warm-start" helps prevent getting stuck in local minima of the objective function by fitting the model several times in succession with reset optimizer parameters.

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: Sparse latent-factors are more intuitive (to us humans). They allow us to think of an example as a union of individual components.

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

   Answer: Content-based filtering, since it can recommend movies to them given some feature set of that new user. While collaborative-filtering would have no data (past ratings) for the new user to make a prediction.

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

   Answer: They are mainly used for supervised learning, since the outputs are provided in training. They are parametric since the network architecture is independent of the number of examples.

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

   Answer: A neutral network with more layers has a higher complexity and therefore is more likely to overfit in training. Regularization can be used to mitigate that.

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: There are $kd + 5k = 1500$ weights and $k + 5 = 105$ bias parameters, for a total of 1605 parameters.

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

Answer: The gradient of the sigmoid function approaches zero fast away from the origin. This is further exacerbated in deep networks by the sigmoids being chained. The near-zero gradients in the network result in high computational error.

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

   Answer: For an image with moderate resolution, the number of parameters in a multi-layer neural network quickly gets out of hand, requiring large amounts of storage and computation and easily overfitting the data. Yet another advantage of CNNs is they systematically consider the relation between neighboring pixels instead of considering them independently.