

CPSC 340 Assignment 6 (due Wednesday April 12 at 11:55pm)

Name(s) and Student ID(s):

1 Robust PCA for Background Subtraction

If you run `python main.py 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. 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

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?

1.2 Robust PCA Approximation and Gradient

If you run `python main.py 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.

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:

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

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:

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

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:

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

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

1.3 Robust PCA Implementation

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.

1.4 Reflection

Answer the following questions and briefly justify your answer.

1. Very briefly comment on the results from the previous section — does robust PCA seem to do better than regular PCA for this task?
2. How does the number of video frames and the size of each frame relate to n , d , and/or k ?
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)?

2 Neural Networks

2.1 Neural Networks by Hand

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 $x_i^T = [-4 \quad -2 \quad 4]$ what are the values in this network of z_i , $h(z_i)$, and \hat{y}_i ? (show your work)

2.2 Neural Networks vs. Linear Classifier

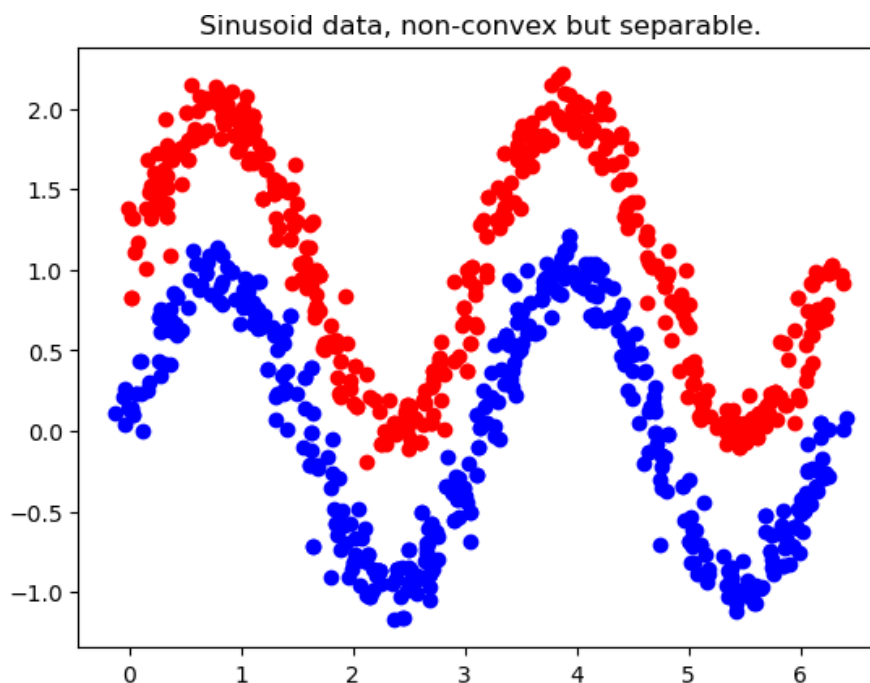
If you run `python main.py 2`, the program will train a multi-class logistic regression classifier on the MNIST handwritten digits data set using stochastic gradient descent with some pre-tuned hyperparameters. The scikit-learn implementation called by the code uses a minibatch size of 1 and a one-vs-rest 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 2.2`, 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%. [Submit your modified code and report \(1\) the training and validation errors that you obtain with your hyperparameters, and \(2\) the hyperparameters that you used.](#)

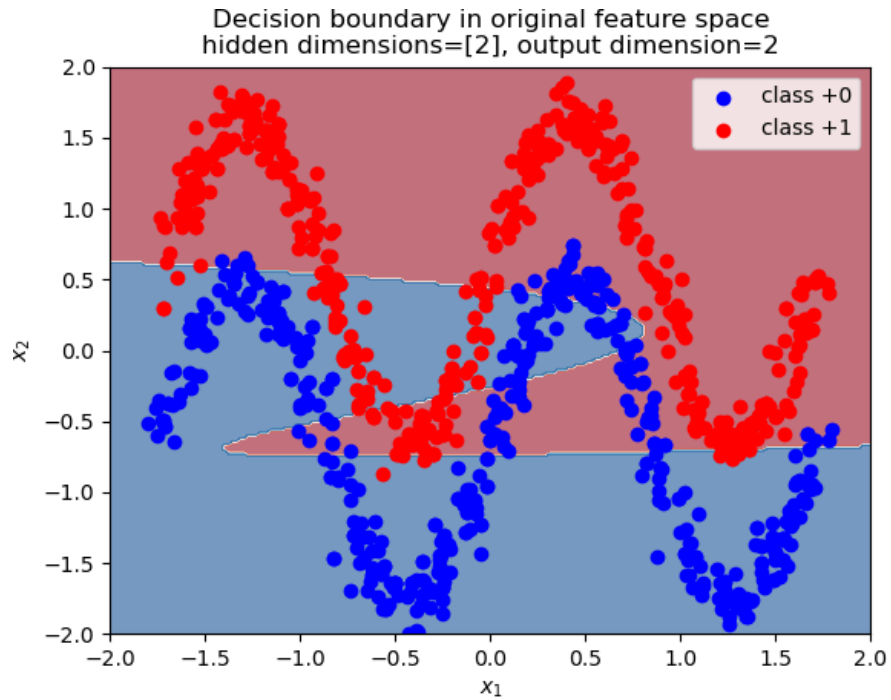
2.3 Neural Disentangling

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.

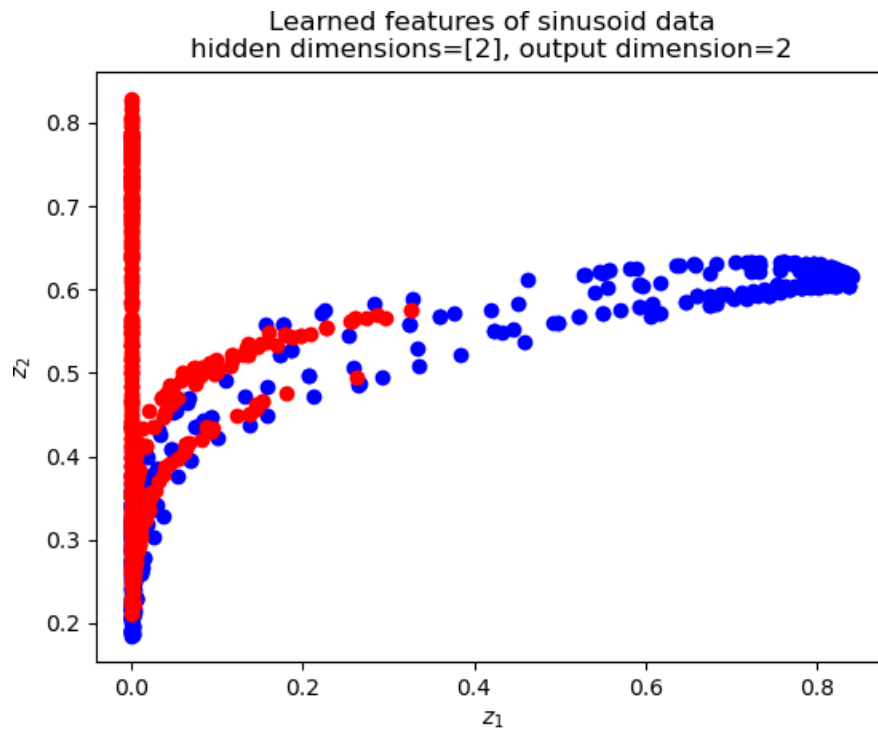
The following figures resulted from running `python main.py 2.3`. For this, a neural network model is used to encode the examples into a 2-dimensional feature space. Here's the original training data:



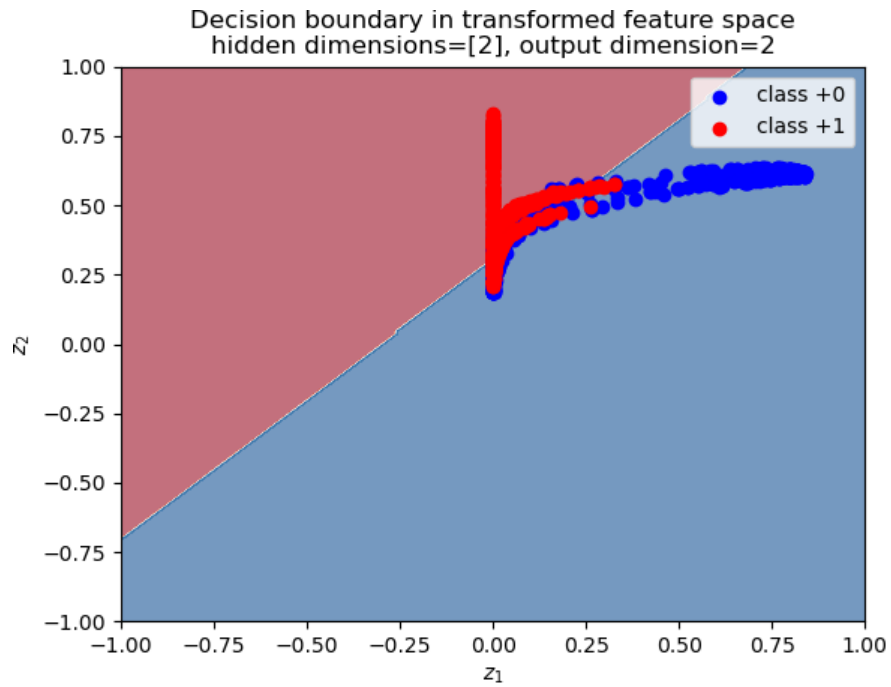
And here is the learned decision boundary of the neural network:



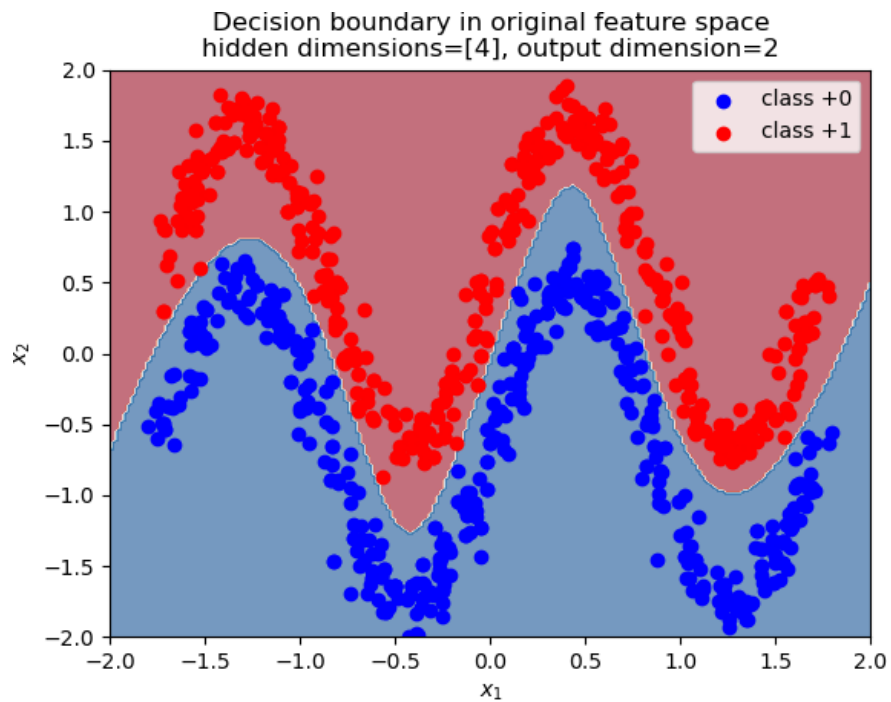
We can look inside and see what the encoder learned. Here is the training data shown in the 2D transformed feature space learned by the encoder (first layer):

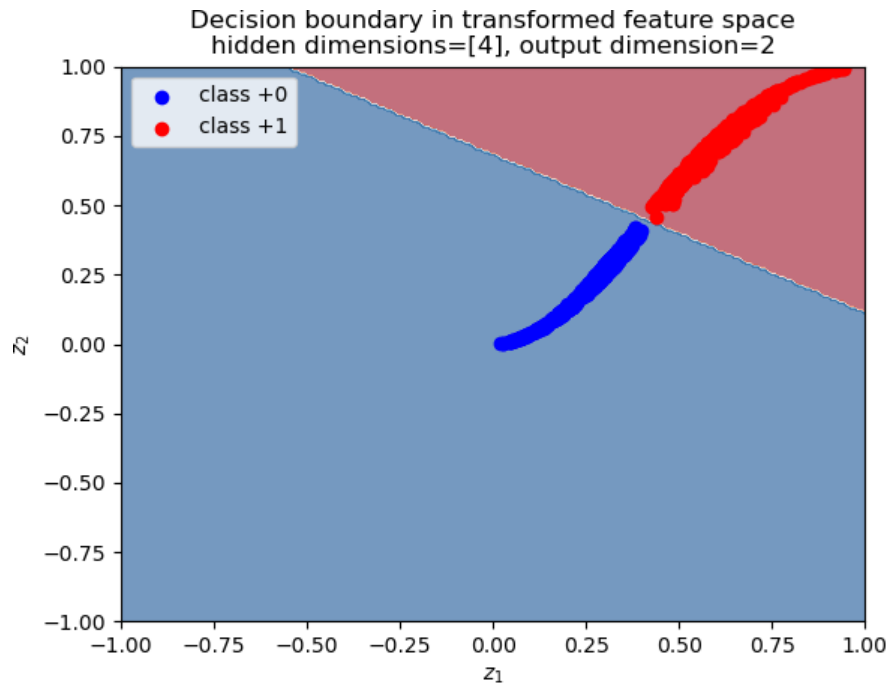


Here is the linear classifier learned by the predictor in this transformed feature space:



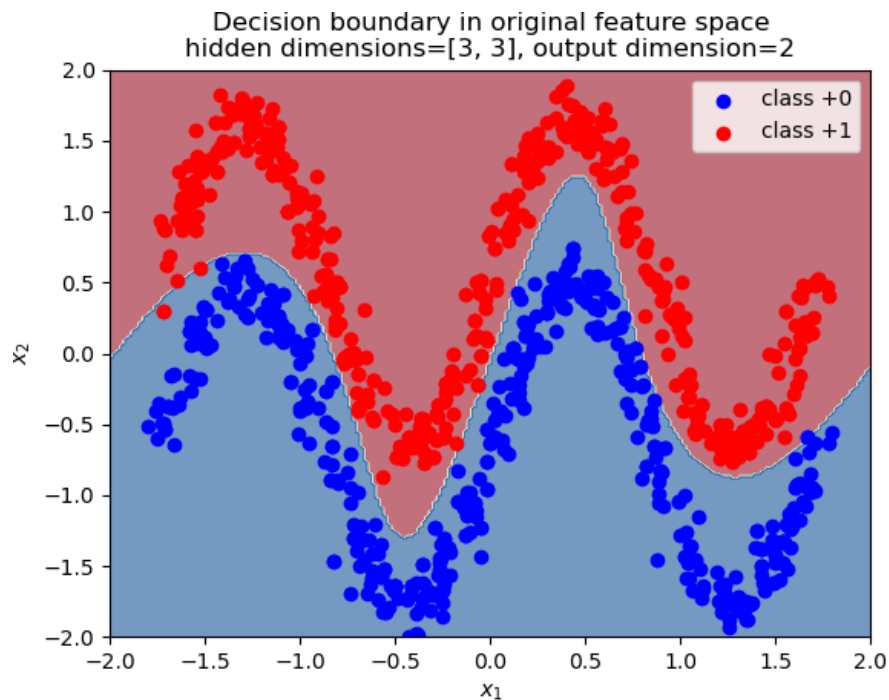
This particular neural network solution does not classify the given examples very well. However, if we use a 4-dimensional hidden features (with all else equal), we get a better-looking result:

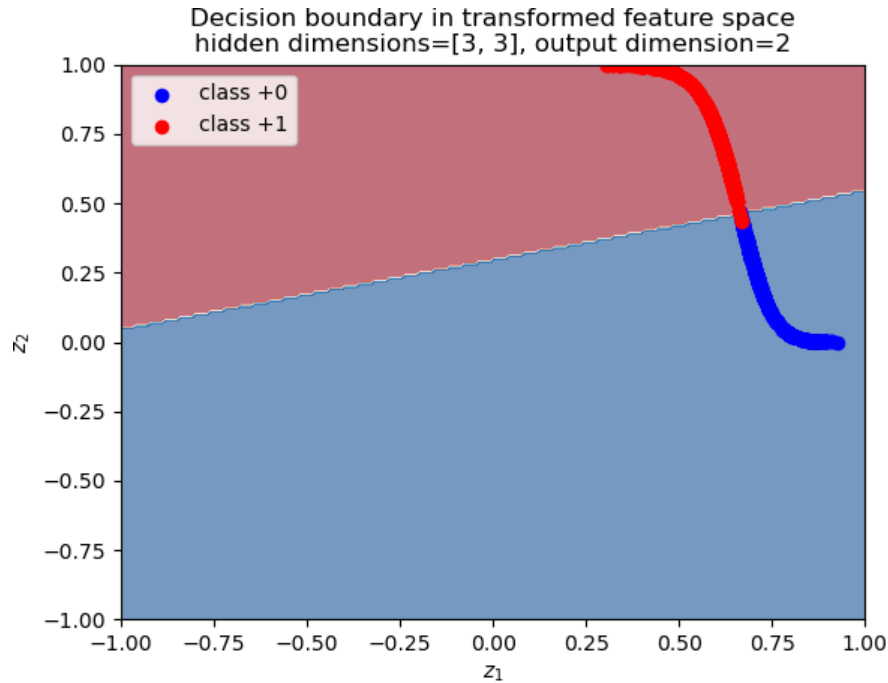




Note: in this case, since the hidden feature space has dimension 4, we are only able to plot 2 of the 4 dimensions. The actual linear classifier is a 3D hyperplane dividing the 4D space. However, even with these two dimensions that we can plot, we can already see that the training data looks pretty much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





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!
2. Does it make sense to use neural networks on datasets whose examples are linearly separable in the original feature space?
3. Why would increasing the dimensionality of the hidden features (which is a hyper-parameter) help improve the performance of our classifier?
4. Why would increasing the number of layers help improve the performance of our classifier?
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.)

3 Very-Short Answer Questions

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.
2. Are neural networks mainly used for supervised or unsupervised learning? Are they parametric or non-parametric?
3. Why might regularization become more important as we add layers to a neural network?
4. 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?

5. Consider fitting a linear latent-factor model, using L1-regularization of the w_c values and L2-regularization of the z_i values,

$$f(Z, W) = \frac{1}{2} \|ZW - X\|_F^2 + \lambda_W \sum_{c=1}^k [\|w_c\|_1] + \frac{\lambda_Z}{2} \sum_{i=1}^n [\|z_i\|^2],$$

- (a) What is the effect of λ_Z on the two parts of the fundamental trade-off in machine learning?
 - (b) What is the effect of k on the two parts?
 - (c) Would either of answers to the *previous two questions* change if $\lambda_W = 0$?
6. Which is better for recommending movies to a new user, collaborative filtering or content-based filtering? Briefly justify your answer.
7. Consider fitting a neural network with one hidden layer. Would we call this a parameteric or a non-parametric model if the hidden layer had n units (where n is the number of training examples)? Would it be parametric or non-parametric if it had d^2 units (where d is the number of input features).
8. Consider a neural network with L hidden layers, where each layer has at most k hidden units and we have c labels in the final layer (and the number of features is d). What is the cost of prediction with this network?
9. What is the cost of backpropagation in the previous question?
10. What is the “vanishing gradient” problem with neural networks based on sigmoid non-linearities?
11. Convolutional networks seem like a pain... why not just use regular (“fully-connected”) neural networks for image classification?