CPSC 340 Assignment 6 (due Wednesday December 7 at 11:55pm)

Name(s) and Student ID(s):
David Perez Carrasco - 25984360
Lucas Amar - 37082823

1 Sparse Latent-Factor Models

If run the script *example_faces.jl* it will load a set of face images under different lighting conditions. It will then fit a PCA model (with $k = 16$) and creates 3 plots:

- Random faces taken from the dataset (*facesOriginal*).
- Reconstructed versions of these faces based on the PCA model (*facesCompressed*).
- The average face (*faceMu*).
- The principal compoenents (*eigenfaces*).

Since Plots.jl seems to only allow you to show one plot at at time, I think you can type 'facesOriginal" in the REPL to switch to that plot (for example). The reconstructions tend to be reasonable given that they compress each 1024-pixel face down to just 10 numbers, although they often lack specific details and are less accurate for faces that do not look like the average face. In this model, some of the principal components are interpretable (for example, they seem to reflect different lighting conditions), but many of them are not. You can change the value of k in this script to see the effect of including/excluding principal components, if you increase k it will do a better job of reconstructing faces that do not look like the average face.

1.1 Uniqueness of Principal Components

On different runs of the script, you may get different principal components, even though all that changes between runs is the order of the training examples. What is the specific difference between the principal components that are obtained between different runs of the algorithm?

Answer: Most of the times, the specific difference between the principal components is only the order of them, for instance, we notice a switch between PC1 and PC2 depending on the run of the algorithm. Other times, some principal components are replaced by others, but if we pay attention we can notice that this new principal components are quite similar to the older ones, some of them may just inverse the light intensity (changing the sign, for example) or decreasing/increasing it (multiplying the factors by some constant). Other times, the principal components are replaced but they show quite similar information, maybe because they are the rotated version of the previous ones, or one not orthogonal to that. All of this make sense, since the main problem of the uniqueness of PCA is that we can obtain equally optimized solutions by changing the sign, multiplying by a constant or rotating one principal component, or rotating two or more components between them.

1.2 Non-Negative Matrix Factorization

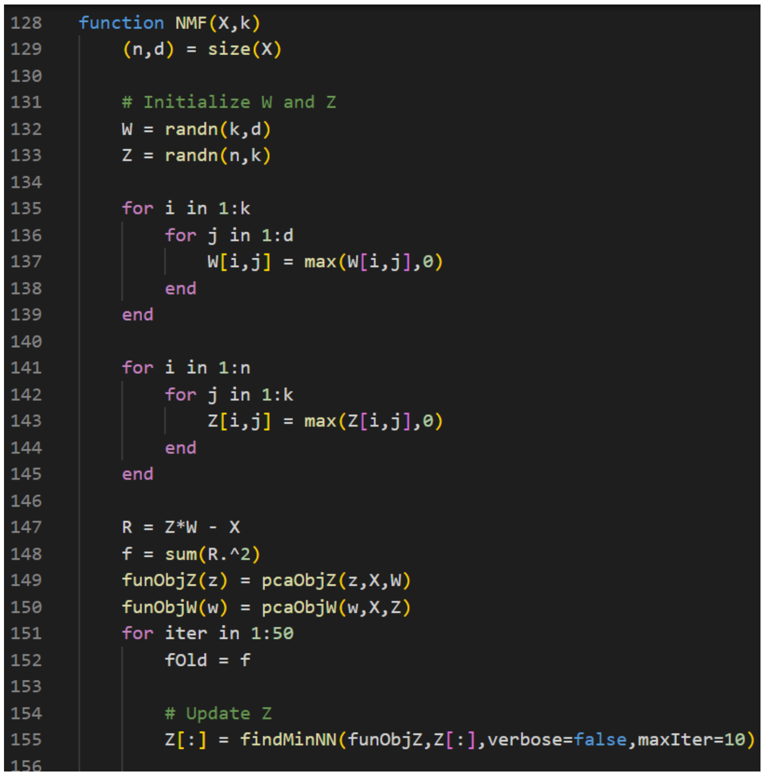
If you replace the function call to *PCA* with *PCA_gradient*, then instead of using the SVD to compute the principal components it will compute them numerically by using gradient descent steps (alternating between updating W and Z). Note that this returns different principal components because it does not enforce any constraints on W, but it will give an equivalent predictions (up to numerical accuracies).

A disadvantage of PCA is that the principal components tend to be dense (they are all non-zero), which can make them more difficult to interpret. One of the first sparse variations on PCA is non-negative matrix factorization, where we use the PCA objective but add non-negative constraints on W. Using *PCA_gradient* as a template, write a function *NMF* that implements the non-negative matrix factorization (NMF) model.

Hint: you need to make several changes. First, you need to remove the step that centers the data matrix X (NMF cannot model the negative values that get introduced by centering). Second, you need to change the initialization so that negative values of W and Z are set to 0. Third, you need to change the updates of W and Z to use an optimization method that includes the non-negativity constraint. You can use the function *findMinNN* which minimizes a differentiable function subject to non-negative constraints. You will also need to update the *compress* function so that it does not center the data and it finds the non-negative Z minimizing the objective with W fixed, and the *expand* function so that it does not "un-center" the data.

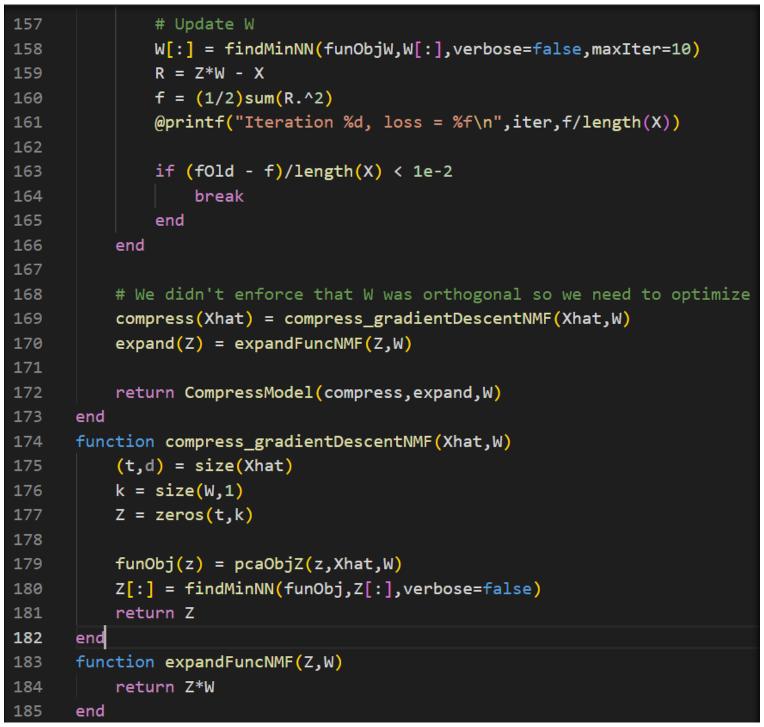
1. Hand in the 3 updated functions required to implement NMF instead of PCA.

   Answer:



```
function NMF(X,k)
    (n,d) = size(X)

    # Initialize W and Z
    W = randn(k,d)
    Z = randn(n,k)

    for i in 1:k
        for j in 1:d
            W[i,j] = max(W[i,j],0)
        end
    end

    for i in 1:n
        for j in 1:k
            Z[i,j] = max(Z[i,j],0)
        end
    end

    R = Z*W - X
    f = sum(R.^2)
    funObjZ(z) = pcaObjZ(z,X,W)
    funObjW(w) = pcaObjW(w,X,Z)
    for iter in 1:50
        fOld = f

        # Update Z
        Z[:] = findMinNN(funObjZ,Z[:],verbose=false,maxIter=10)
```

Figure 1: 1.2 - NMF update functions Part 1



```
        # Update W
        W[:] = findMinNN(funObjW,W[:],verbose=false,maxIter=10)
        R = Z*W - X
        f = (1/2)sum(R.^2)
        @printf("Iteration %d, loss = %f\n",iter,f/length(X))

        if (fOld - f)/length(X) < 1e-2
            break
        end
    end

    # We didn't enforce that W was orthogonal so we need to optimize
    compress(Xhat) = compress_gradientDescentNMF(Xhat,W)
    expand(Z) = expandFuncNMF(Z,W)

    return CompressModel(compress,expand,W)
end
function compress_gradientDescentNMF(Xhat,W)
    (t,d) = size(Xhat)
    k = size(W,1)
    Z = zeros(t,k)

    funObj(z) = pcaObjZ(z,Xhat,W)
    Z[:] = findMinNN(funObj,Z[:],verbose=false)
    return Z
end
function expandFuncNMF(Z,W)
    return Z*W
end
```

Figure 2: 1.2 - NMF update functions Part 2

2. What do you think the NMF factor represents if you set $k = 1$? (Hint: look at all the plots made by the original script.)

   Answer: The NMF factor seems to represent a face quite similar to the average face, which can work as the baseline for all the faces, and including more factors would include the representative aspects of each face, like if this first factor was the general face from which the rest of the factors add the details that enable to recognize all the faces. Also, it seems a bit brighter than the average face, like if the number of black pixels has increased, maybe becaus eof the non-negativeness constraint.

3. What do the NMF factors represent if you set $k = 2$?

   Answer: When k=2, the factors seem to represent the half of the average face. We always get a factor that represents a face with the left side of the face illuminated and another that represents a face with the right side of the face illuminated, what means these two factors are the opposite and represent the brighness (and maybe the most basic traits) of each side of the face.

4. Hand in a plot of the NMF factors with $k = 16$.

   Answer: Figure 3

5. Explain why the NMF model should have a higher or lower loss than the PCA model.

   Answer: If we talk about the more optimal PCA version, the one that applies basis, orthogonality and sequential fitting, then PCA is going to have lower loss than NMF, because the NMF components are constrained to be non-negative, not to be perpendicular, so the components of PCA are always going to get the optimal linear combination of k components so it gets the best k-rank approximation of the original matrix, while the NMF is not designed to obtain the best linear combination, hence it will have higher loss than PCA.

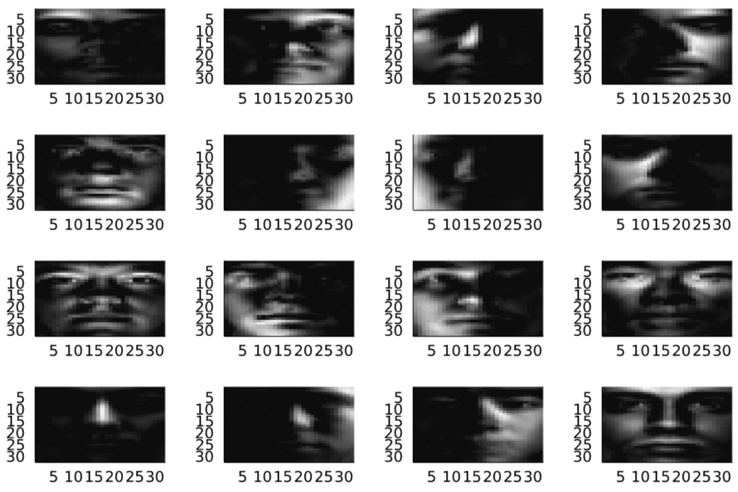


Figure 3: 1.2.4 - Plot of the NMF factors with k=16

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 $(x^i)^T = \begin{bmatrix} -3 & -2 & 2 \end{bmatrix}$ what are the values in this network of the hidden units z^i, activations $a^i = h(z^i)$, and prediction $\hat{y}^i$?

Answer: $z^i = Wx^i = \begin{bmatrix} -2 & 2 & -1 \\ 1 & -2 & 0 \end{bmatrix} * \begin{bmatrix} -3 \\ -2 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

Since $h(z^i) = \frac{1}{1+\exp{-z^i}}$ with sigmoid activations: $a^i = h(z^i) = \begin{bmatrix} 0.5 \\ 0.731 \end{bmatrix}$

$\hat{y}^i = \text{v}^t a^i = \begin{bmatrix} 3 & 1 \end{bmatrix} \begin{bmatrix} 0.5 \\ 0.731 \end{bmatrix} = 2.231$

2.2 Neural Network Tuning - Regression

The file `example_nnet.jl` runs a stochastic gradient method to train a neural network on the *basisData* dataset, and outputs a plot of the result (depending on your configuration, it may output these plots as the algorithm is running too so that can see how the learning proceeds). Unfortunately, in its current form the neural network does not fit the data well. Modify the training procedure and model to improve the performance of the neural network. Hand in your plot after changing the code to have better performance, and list the changes you made.

Hint: there are many possible strategies you could take to improve performance. Below are some suggestions, but note that the some will be more effective than others:

- Changing the network structure (*nHidden* is a vector giving the number of hidden units in each layer).
- Changing the training procedure (you can change the stochastic gradient step-size, use decreasing step sizes, use mini-batches, run it for more iterations, add momentum, switch to *findMin*, use Adam, and so on).
- Transform the data by standardizing the feautres, standardizing the targets, and so on.
- Add regularization (L2-regularization, L1-regularization, dropout, and so on).
- Change the initialization.
- Add bias variables.
- Change the loss function or the non-linearities (right now it uses squared error and tanh to introduce non-linearity).
- Use mini-batches of data, possibly with batch normalization.

Answer: Figure 4

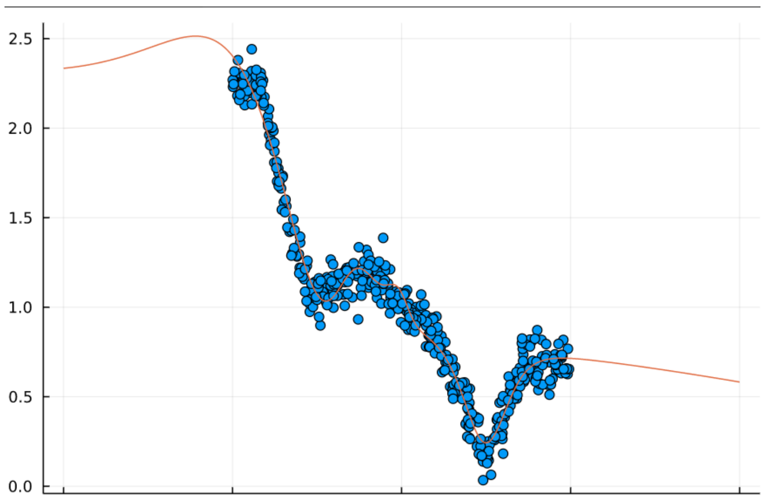


Figure 4: 2.2 - Plot of training neural network

We tried a lot of things in order to get the previous plot, almost all of the listed options. The changes made to the final version of the algorithm were:
-Add bias to the first layer, to all the rest of the layers and to the output (basically adding a column of 1s to X and Xhat and including a 1 in the first column of each layer of z, both in the backprop and in the predict functions).

- Change the neural network structure. We chose a neural network of 2 hidden layers with 10 units in each one (nHidden = [10 10]), although we got similar plots with 2 layers with 8 units or 3 layers with 5-10 units.
- Increase the number of iterations. We finally set it to 30000, although we got similar plots with 100000, 50000, 20000 and even with 10000 sometimes.
- Change the step size: We finally chose a step size of 1e-3. However, we also tried to update the step size at each iteration, for instance with 1e-3/sqrt(t), and it seemed to return similar plots, but since the constant step size already worked we decided for the constant one.

Other notes: We also tried to standardise the features and it did not seem to get better results, although standardising the output seemed pretty decent depending on the run. We also tried to change the loss function to the logistic function, but it did not seem to return better results neither.

2.3 Neural Network Tuning - Classification

The file `example_mnist35_logistic` runs a stochastic gradient method to train a neural network on the '3' and ''5' images from the MNIST dataset, which obtains a test error of 0.03. If you instead run `example_mnist35_nnet.jl`, it fits a neural network with stochastic gradient descent on the same data (using the squared error for training and classifies by taking the sign of the prediction) but obtains a worse error. Modify the neural network training procedure and model so that the neural network has a better error on the test set than the 0.03 achieved by logistic regression. List the changes you made and the best test performance that you were able to obtain.

Answer: The best test error that we could obtain was 0.02, and we did the following changes:
- Add a bias to X and Xhat and to every hidden layer. As we commented before, by adding a column of 1s to X and Xhat and setting 1 to z[layer][1] for each laye.
- Setting the stepsize to 1-e2
-Setting the maxIterations to 1100000
- Changing the network structure to 2 hidden layers with 10 units each one, although it also worked with 3 of 5-10.
- Changing the sigmoid function to the ReLu, or perceptron loss, basically by choosing the maximum between 0 and each value of the gradient.

3 Using a Deep Learning Package

The script *example_flux.jl* shows how the compute the loss function and gradient of a neural network with one hidden layer (with 3 units) using a variety of strategies, and how to update the parameters based on an SGD update using a variety of strategies. This includes the backpropagation code from the previous question, as well automatic differentiation and other functionality provided by the *Flux* package. The function does not produce any output, but you can verify that all methods return the same results up to numerical precision.

1. In the previous assignment we concatenated all parameters into one big vector of parameters w. What is an advantage of the other approaches in the script, where we have a matrix W and a vector v?

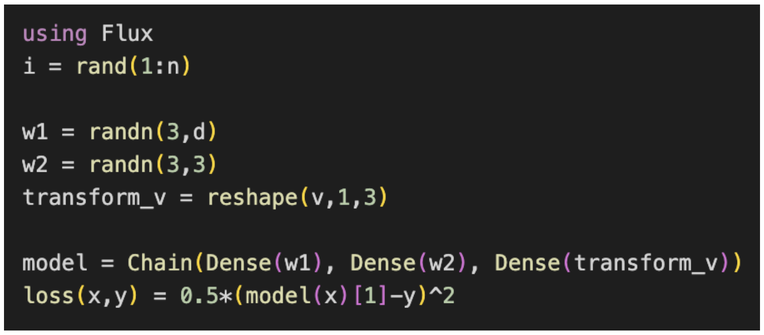
   Answer: Having a $k * d$ matrix W and a $k * 1$ vector v is beneficial because it allows for a linear transformation similar to PCA, given that it mixes the features in a way that we learn. You additionally avoid having to reshape

2. If we use use `Dense(W)` for the first layer in the network, how many parameters does the first layer have and what do the extra parameters represent? Hint: you can use `params(layer)` to get the parameters of a layer.

   Answer: 2355 parameters:
   The size of the hidden layer is 3, so we have an output that is a vector of size 3.
   784 features in the data-set, giving a $3 * 784 \rightarrow 2352$ variables and the extra 3 is the bias variables.

3. Give code for implementing a neural network with 2 hidden layers, using the `Dense` function within the `Chain` function.

   Answer:



```
using Flux
i = rand(1:n)

w1 = randn(3,d)
w2 = randn(3,3)
transform_v = reshape(v,1,3)

model = Chain(Dense(w1), Dense(w2), Dense(transform_v))
loss(x,y) = 0.5*(model(x)[1]-y)^2
```

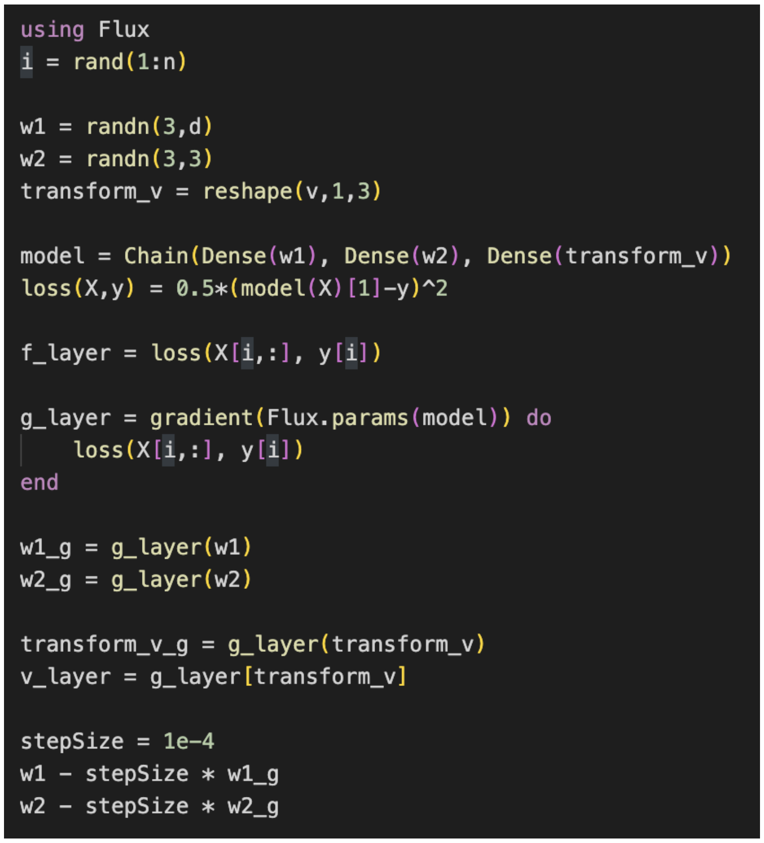
Figure 5: 3.3-Code

4. It we use *Conv((5,5),3=>6,relu)* for a layer, why do we get a layer with 456 parameters?

   Answer: Because $5 * 5 * 3 * 6 = 450 + 6$ bias variables $= 456$ parameters

5. Re-write the model and training procedure you developed for Question 2.3 to use one of these alternate ways to compute the gradient and the update of the parameters (so you should not be calling *NeuralNet.jl* anymore). Re-state the changes you made, hand in your modified code, and report whether you noticed any performance differences (in terms of runtime, memory, or accuracy).

   Answer:
   I have calculated the gradients in different ways as seen by the last 2 lines.
   I have implemented the f_layer and g_layer to calculate the loss.



```
using Flux
i = rand(1:n)

w1 = randn(3,d)
w2 = randn(3,3)
transform_v = reshape(v,1,3)

model = Chain(Dense(w1), Dense(w2), Dense(transform_v))
loss(X,y) = 0.5*(model(X)[1]-y)^2

f_layer = loss(X[i,:], y[i])

g_layer = gradient(Flux.params(model)) do
    loss(X[i,:], y[i])
end

w1_g = g_layer(w1)
w2_g = g_layer(w2)

transform_v_g = g_layer(transform_v)
v_layer = g_layer[transform_v]

stepSize = 1e-4
w1 - stepSize * w1_g
w2 - stepSize * w2_g
```

Figure 6: 3.5-Code

   I did not notice a significant performance difference.

4 Very-Short Answer Questions

1. 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?

   Answer: λ_Z acts as a regularizer, therefore, the greater λ_Z the more overfitting there will be. So:
   $\downarrow \lambda_Z \rightarrow \downarrow error_{train} \rightarrow \uparrow error_{test}$
   $\uparrow \lambda_Z \rightarrow \uparrow error_{train} \rightarrow \downarrow error_{test}$

   (b) What is the effect of k on the two parts?

   Answer: A larger k means that there are more dimensions, yielding a fall in training error and a rise in test error, and vice versa.

   (c) Would either of answers to the *previous two questions* change if $\lambda_W = 0$?

   Answer: λ_W is used for regularization, therefore if $\lambda_W = 0$, there is no more regularization, and therefore the training error will fall causing an increase in the test error as indicated by the fundamental trade-off.

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

   Answer: For new users, it is best to use content-based filtering, given that collaborative filtering does not work for them (users with no ratings). Collaborative filtering will learn a vector of features for each user, therefore it is more accurate for existing users, yet for new users it would not work.

3. Consider using a fully-connected neural network for 3-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: Bias column in x gives d= 11, so $100 * 11 + 3 * 100 + 3$ bias variables 1403 parameters.

4. 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).

   Answer: Non-parametric first one because it depends on n, parametric the second as it depends on d, not n, so, provided d does not change it will be parametric

5. Describe an experimental setup where you might see a double descent curve when fitting a logistic regression model with L2-regularization. You can assume you have a way to generate new relevant features. Caution: the global minimum is unique in this model so the double descent would not come from choosing among multiple global minima.

   Answer: If there is a situation with lots of parameters, a double descent curve might be a good approach.

6. Consder 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?

   Answer: Cost $= kd + ck + (L-1) * k^2$

7. What is the cost of backpropagation in the previous question?

   Answer: Cost is the same, given that we are using differentiation, so: $kd + ck + (L-1) * k^2$

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

   Answer: As more layers are added to the neural network, the gradient of the loss function falls, approaching 0, making it difficult to train the network.

9. What is one advantage of using automatic differentiation and one disadvantage?

   Answer:
   Advantage: You get the derivative for the same cost as the function, therefore you can have more features.
   Disadvantage: Stores all intermediate calculations, and therefore requires a lot of storage.

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

    Answer: Because with convolutional networks, you can have layers that apply several convolutions.