Austin Prince

Data Mining

Homework 4

**Overview and Intro**

For this homework we are attempting to use an autoencoder that learns the binary representation for decimal numbers. In class we learned about the position encoding scheme which is simply a way to represent a binary number. The example we talked about in class applied to the decimal numbers 0 to 7 and here we are simply extending that same model to the numbers 0 to 15.

The position encoding scheme we discussed in class corresponds to specific architecture of this neural network. Given that we have 16 input nodes, the scheme we discussed is a column vector of 16 zeros with each place representing a value of 0 to 15. So if you have a one in the first row and zeros everywhere else this corresponds to a decimal value of 0, if you move this one down one place and have zeros in all other locations this corresponds to a value of 1 and so on.

The input data that I used was of the form



Or a 16x16 matrix with ones on the diagonal. For this project I used python so I simply stored these inputs in a 2d array.

An auto encoder is a specific type of neural network that takes in an input, in this case one column from the above matrix, and then encodes it into some compressed representation that is dictated by the architecture of the neural network. Then this compressed representation is decoded and passed to the output nodes where ideally it should mirror the input.

The autoencoder can be used for several purposes

Since we are using decimal numbers if we are encoding these decimal numbers, the most compressed form of notation would be a binary representation. Given that we have the numbers 0 to 15, the most compressed binary notation of this range of numbers would be a 4 digit representation.

**Methods**

I create an autoencoder using the keras library in python that provides a simple interface for creating an autoencoder. I had to create the encoding layer and the decoding layer separately but by using the following code I created a 3 layer NN that has a hidden layer of 5 perceptrons.

encoding\_dim = 5  
  
  
input\_data = Input(shape=(16,))  
  
encoded = Dense(encoding\_dim, activation='sigmoid',kernel\_initializer=initializers.RandomUniform(),  
 bias\_initializer='zero')(input\_data)  
  
  
decoded = Dense(16, activation='sigmoid',kernel\_initializer=initializers.RandomUniform(),  
 bias\_initializer='zero')(encoded)

rms = optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-08, decay=0.0)  
autoencoder.compile(optimizer=rms, loss='binary\_crossentropy')

autoencoder.fit(x\_train, x\_train,  
 epochs=200000,  
 verbose=2,  
 batch\_size=256,  
 shuffle=True,  
 validation\_data=(x\_train, x\_train))  
  
decoded\_data = decoder.predict(encoded\_data)

encoded\_data = encoder.predict(x\_train)

The above code ues the keras library to create a neural network that has 16 input nodes, a hidden layer of 5 perceptrons, activated by the sigmod function, and an output layer of 16 nodes as well.

This initializes an hidden layer of 5 perceptrons that are activated by the sigmod function, with random initial weights, and an ouput layer of 16 nodes with weights coming into these nodes that are random as well.

To build the deep neural network we constructed the architecture in a similar fashion, however, we made sure the feedforward the outputs from the previous layers to the appropriate next layer. The keras library allowed us to see the output from each of the hidden layers in our deep neural network, which is information that is included in the appendices.

**Question 1**

I ran this code 5 times using different randomly generated initial weights and observed the stable states of the hidden layer and the output layer. I did not use any sort of convergence metric for this I simply ran the autoencoder for 20000 epoches meaning 30000 instances of forward propogation followed by backpropogating the resulting error.

Because the initial values of the weights are different every time given that I am initializing the weights randomly, I do not expect the encoded representations across different trials to be the same. However, eventually I do expect that within one trial of learning the data, the encoded representations will eventually converge to some value, given that after running back propogation we eventually expect the weights to converge.

Using the loss function with a max epochs of 200000 and a learning rate of 0.001, after these 200,000 epochs I reached a relatively stable state for my outputs that is displayed below. At a certain point the computer is correcting the weights to deal with very, very small differences between the expected output and the output from the auto encoder, and this slows down the processing significantly.

The encoded representation that I got that converged using the 5 perceptron hidden layer was the following.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0.999996305 | 0.466294229 | 0.466294229 | 3.40E-06 | 0.99997592 |
| 0.054521699 | 4.34E-06 | 4.34E-06 | 0.288765848 | 0.000782793 |
| 0.150592148 | 1.58E-06 | 1.58E-06 | 0.999997377 | 0.331258267 |
| 0.999997973 | 0.999990821 | 0.999990821 | 0.999999642 | 0.999925971 |
| 0.236431062 | 3.04E-05 | 3.04E-05 | 0.497362643 | 0.943800807 |
| 0.999996543 | 0.373870581 | 0.373870581 | 2.24E-05 | 1.54E-07 |
| 6.11E-05 | 0.439897686 | 0.439897686 | 0.999999404 | 0.999999046 |
| 0.623939335 | 6.57E-06 | 6.57E-06 | 1.84E-06 | 0.405180007 |
| 6.22E-05 | 0.378803164 | 0.378803164 | 1.90E-05 | 0.800069034 |
| 0.00015365 | 0.485362977 | 0.485362977 | 0.999953985 | 3.64E-06 |
| 0.999996781 | 6.34E-05 | 6.34E-05 | 0.999967098 | 0.999993443 |
| 0.165358037 | 0.999939442 | 0.999939442 | 0.281707913 | 0.999998808 |
| 0.999987841 | 0.000249515 | 0.000249515 | 0.999999881 | 8.07E-07 |
| 0.999999523 | 0.892213106 | 0.892213106 | 0.651267946 | 7.80E-06 |
| 0.163125366 | 0.676158011 | 0.676158011 | 0.000894567 | 3.90E-06 |
| 0.999999762 | 0.99866128 | 0.99866128 | 1.06E-05 | 0.527402043 |
|  |  |  |  |  |

I have included the encoded representation of all my trial runs with different random initial weights in the appendix.

This table represent each of the numbers (inputted in position encoding scheme) as they have been encoded in the hidden layer of the auto encoder. Each row in the above table represents a number in position encoding scheme (0 to 15) and each column represents the value that a hidden node is using to represent this data.

**This data does not represent what we would expect as a binary encoding of the data. This is probably because the way that the neural network learned to represent this data, including the weights from the inputs to the hidden layer and the weights from the hidden layer to the output** does compactly does not exactly match up with what we would expect a binary representation to look like. However, after 200,000 runs through the loss value did converge to around 3.6021e-07, which suggests that this representation is encoding the data in an accurate fashion.

There are further reasons that this representation does not correspond to what we understand as a binary representation of a number. The weights and the encoded values at the hidden layer are continuous values which is different that binary numbers which are discrete (either 0 or 1). This means that our encoding has more representational power and can find representations that do not necessarily correspond to a binary representation.

If we look at the different runs of our auto encoder we can see that the encoded representations differ between runs, which makes sense given that, because our weights are randomized across each run they are converging to different local minima each time.

The decoded data that I gained from this auto encoder ended up converging to a loss value of around 1e-8, which is very small. When checking the decoded outputs, I got the following values, which closely mirror the input which is what I would expect given the low value of the loss function. (Included this data in appendix)

**Question 2**

Running the same code that has a hidden layer containing 4 perceptron’s, we would expect this auto encoder to be able to accurately encode the data, given that the numbers 0 to 15 can be accurately represented using 4 binary digits. I have included the encoded representations in the appendix. This trial displayed similar patterns as the auto encoder that I ran with 5 perceptron’s in the hidden layer.

* Differing encoded representations between trials (probably due to different local minima of the error function)
* Encoded representation does not correspond to what we would imagine as a binary representation of numbers from 0 to 15 (probably due to the continuous weights that are available)

**Question 3**

When approaching this problem, I initially did not expect the auto encoder to be able to find a representation that would compress the digits 0 to 15 down to 3 bits. I had this expectation because if we imagine a binary encoding as the “most compressed” version of integer representation, the numbers 0 to 15 need 4 bits to be represented in binary. However after running the auto encoder using a hidden layer containing 3 perceptrons I found that the auto encoder was able to find a compressed representation of the digits from 0 to 15.

I speculate that this is because of the continuous nature of the representation. A 3-bit binary number only has the representational power described by the equation

In this equation the 2 refers to the value that each bit can take, a discrete value of 0 or 1. However in our representation we have infinite values that each bit can take, given that we can encode our numbers as any float value between 0 and 1. This fact, that we are not limited by discrete bit values gives us far more representational power and means that we can in fact encode the numbers 0 to 15 using a 3-bit encoding. I have included the stable encoded states in the appendix, as well as the stable decoded states, please see there for specifics.

**Discussion about difference**

One thing that I did notice about the different trials of a 3 layer NN with different numbers of perceptron’s in the hidden layer is that the value of the loss function converged slower for the neural network with a 3 perceptron hidden layer. In addition, as can be seen in the table below the loss value ended up being lower for the auto encoder that had more perceptron’s in its hidden layer. This is probably because of the higher representational power encoded that exists in the auto encoder with more perceptron’s in the hidden layer.

|  |  |  |  |
| --- | --- | --- | --- |
| **num perceptrons in hidden layer** | 3 | 4 | 5 |
| **loss value after 20000 iterations** | 3.1520e-07 | 1.0765e-07 | 1.0120e-07 |

**Question 4**

**Method**

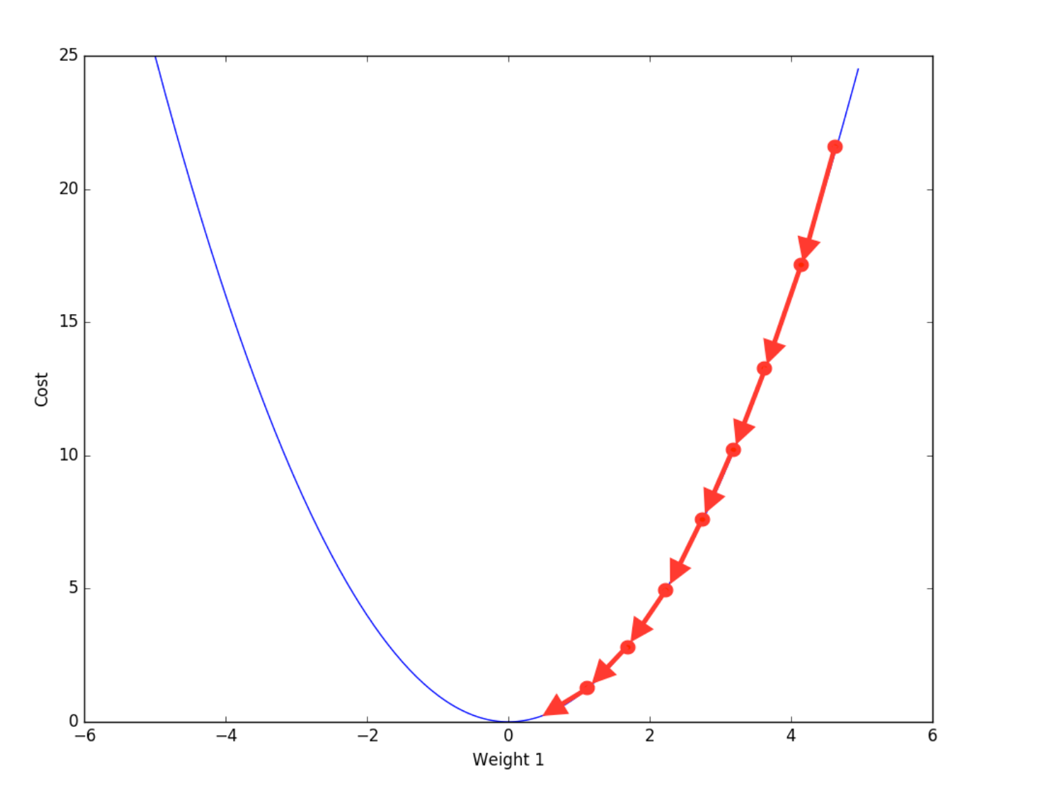
For this problem I constructed a 5 layer neural network. This network contains hidden layers that have 8, 4, and 8 perceptrons respectively.

**Comparison with first three architectures**

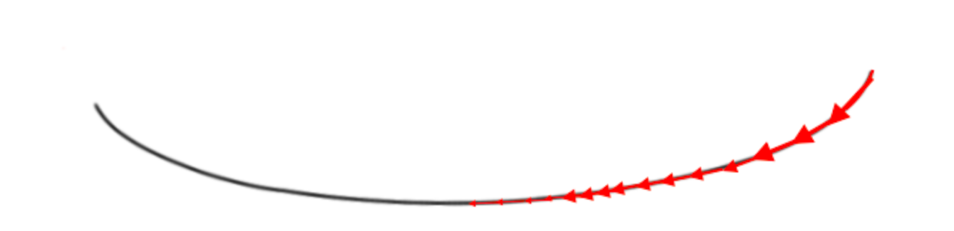
This is the first neural network architecture that we have built that can be considered a *deep* neural network. A deep neural network refers to any network that has more than 1 hidden layer. With deep neural networks we obtain some advantages and some drawbacks. With a deep neural network we gain the bonus of each layer being able to abstract a different feature or set of features, that can be indicative of the underlying structure of our data. One major drawback of a deep neural network is the problem of the vanishing gradient

The vanishing gradient problem is a serious problem in deep neural networks, and refers to the fact that as we backpropogate our error through our neural network we are “squishing” our values in the sigmoid function so they are between 0 and 1. When stack these values on top of each other through backpropogation this error term becomes small which means it will only change our weight a miniscule amount at the earlier layers (further into the backprop algorithm), which means that it will take a very long time for this function to converge.

A graphical representation of this can be seen in the “flattening” of the error surface as we back propogate further. Our optimal error surface would look like this



However as we pass our squished values back our error surface will quickly flatten and become something that appears more like this.



Having multiple layers in your neural network is useful for learning patterns in your data at different levels of abstraction. The separate layers in this neural network architecture will compress your data to different levels meaning that the encodings will represent different levels of compression with potentially different patterns/features being found in the data.

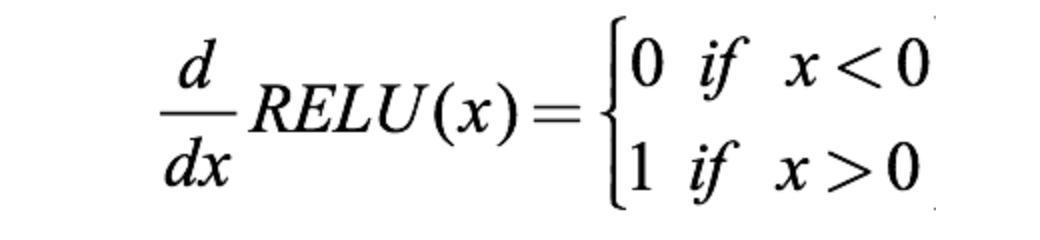
**Question 5**

The difference between question 4 and 5 is that in question 5 we are using the ReLu activation function which attempts to solve the vanishing gradient problem by taking using the following as the activation function.



The ReLu activation function attempts to solve the problem of the vanishing gradient by taking the max value between 0 and x.

The derivative of the ReLu function which is the value that will be back propogated through our system is the following.



This means that our value with be either 0 or 1, not a fractional value! So our gradient will not diminish as we back propogate our error through the system, thus resulting in a convergence that is quicker than using the sigmoid activation function.

**Results from Question 4 and 5**

I have included a table of the final loss value as well as the runtime of both the sigmoid and the relu activation function.

Sigmoid activation function

Iteration 1:

Loss value 0.6289

Runtime: 1457 seconds

Iteration 2:

Iteration 1:

Loss value: 0.348

Runtime: 1498 seconds

Iteration 3:

Loss value: 0.527

Runtime: 1250 seconds

Iteration 4:

Loss value: 0.475

Runtime: 1423 seconds

Iteration 5:

Loss value: 0.2860

Runtime: 1694 seconds

ReLu activation function

Iteration 1:

Loss value: 0.0224

Runtime: 1438 seconds

Iteration 2:

Iteration 1:

Loss value: 0.0131

Runtime: 1357 seconds

Iteration 3:

Loss value: 0.0269

Runtime: 1562 seconds

Iteration 4:

Loss value: 0.0140

Runtime: 1295 seconds

Iteration 5:

Loss value: 0.0293

Runtime: 1736 seconds

With both of these activation functions I found that my loss values converged much quicker than with a single layer neural network. They converged to a higher value, this is because of the vanishing gradient problem again. While the ReLu activation function does manage to speed up the convergence of our weights by assuring that the error surface remains convex as we propogate backwards, it still ends up converging to a higher value than our single layer neural network in the first three problems.

For this problem I did not implement any type of convergence criterion, this is because as we know from the vanishing gradient problem the values of the weights can change very slow. For this reason I was simply running the back propogation algorithm for a set number of times, and so the run times for sigmoid and ReLu were very similar as they were using the same architecture, optimization procedures and number of iterations with the only difference being the activation function.

Sources:

Obtained showing flattening gradient from:

<https://ayearofai.com/rohan-4-the-vanishing-gradient-problem-ec68f76ffb9b>