Autoencoders for Image Compression and Classification

Tara Gill
Dept. of Engineering
Boston University
Boston, MA 02215
taragill@bu.edu

Nathan Lau
Dept. of Engineering
Boston University
Boston, MA 02215
nathancl@bu.edu

Howell Xia
Dept. of Engineering
Boston University
Boston, MA 02215
howellx@bu.edu

Paper ID: *****

Abstract

Autoencoders are a non-linear method used to dimensionally reduce the feature vectors of a dataset. However, the greater the data is reduced, the greater the amount of data is lost. This poses an issue for machine learning applications of large datasets where efficient classification might result in a quick result but at the cost of poor accuracy. Thus, modifications to the autoencoders algorithm such as including label vectors in the input may be a method for improving the classification of the reduced characteristic vector compared to an unmodified "normal" autoencoder compression algorithm.

Categories and Subject Descriptors: I [Artificial Intelligence]: Learning, H.2.8 [Database Management]: Database Applications - Data Mining.
General Terms: Algorithms.
Keywords: Autoencoders, Correct Classification Rate (CCR), Means Squared Error (MSE).

1. Introduction

Redundancies in data will lead to a reduction in transfer rates and in the ability for hardware storage to efficiently manage data. Thus computing hardware bandwidth and capacity would need to advance in order to accommodate a greater degree in which data is used and stored. The volume of data that deep learning and artificial intelligence (AI) can be trained poses a computational burden on modern hardware. As a result, data compression can help save processing and storage burdens.

Especially with imaged based data, where the data space required increases exponentially with higher resolutions, there exists the necessity for the data compression of images. For example, the transfer of videos on the internet for a 1080p (1920 x 1080 pixels) video at 30 fps (frames per second) at 24 bit RGB color, the bandwidth needed founded by

$$30\text{fps} \bullet (1080 \times 1920\text{p}) \bullet 24 \text{ bits} = 1.49\text{Gbps} \quad (1)$$

Thus, a compression algorithm that is able to preserve the image details while reducing the data size is key towards faster and more efficient data transferring and processing [1].

Already, real world applications are used predominantly in the video space where video codecs including H.264, HVEC, VP9, and AV1 are commonly used for the purposes of compression for video transmittance. However, for data applications, reducing the number of classification features helps to address the Curse of Dimensionality which can often lead to overfitting or poor generalizations for high dimensional datasets. This is where algorithms such as autoencoders are able to compress non-linear data. The main negative of using autoencoders is the computational cost. This poses a challenge to efficiently reduce a large amount of data but not at the cost of the loss of information.

2. Literature Review

Autoencoders perform a non-linear dimensionality reduction on an input feature vector while retaining as much information of the original data as possible. By altering what the algorithm learns from the input training data, insight can be gained on efficient methods of reducing data for machine learning classification problems.

2.1. Linear Compression (PCA)

Machine learning compression algorithms can be classified into linear and non-linear methods. Principal Component Analysis (PCA) is an unsupervised linear method that is able to find the top principal directions that is the best k-dimensional subspace for projecting a centered training feature vector to minimize the mean squared error (MSE) during reconstruction. The principal directions are ordered so that the first direction correlates to a feature of the data with the greatest variance between the differing features. Also, because PCA operates by taking the linear combination of the original feature

vectors, it is a highly efficient algorithm and is useful for only linear datasets.

2.2. Huffman Coding Algorithm

Huffman coding is a lossless method of compression that aims to reduce the number or bits to represent an image pixel. This method assigns a specific code to the pixels of an image based on the frequency of the pixel occurrences. The pixels are organized by their length with shorter codes linked to more frequency pixels and longer codes linked with less frequent pixels. A binary tree is constructed from the codes that start from the leaves and move through the roots of the tree. Data can then be decoded through the binary tree to reconstruct an image. This compression algorithm is used in common formats such as MPEG-2 and JPEG.

A comparison between JPEG and other compression algorithms was done to compress colorless scaled images using the Huffman Coding algorithm in a study by the Swiss Federal Institute of Technology. The results showed that a compression ratio of up to 0.8456 was achievable using the Huffman Coding algorithm on the image [2].

2.3. Autoencoder Comparison Between PCA and Huffman Coding

Anand Atreya and Daniel O'Shea from Stanford university compared a standard autoencoder algorithm for image compression against a PCA and JPEG (Huffman Coding) algorithm. It was found that deep autoencoder image compression offered a better reconstruction performance compared to PCA or JPEG methods. The autoencoder was able to find the regularities in the domain of the images used and learn the statistics of the class in a manner that JPEG could not. [3]

2.4. Non-Linear Compression (Autoencoders)

However, not all datasets are linear to which a differing method for reducing the data is needed. This is where autoencoders are able to learn more complex patterns from the feature vector. Through multiple hidden layers of the neural network, nodes and activation functions (such as ReLU or logarithmic) unique boundaries can be defined. It is able to adapt to both linear and non-linear datasets and is a more generalized method to PCA (which is linear) depending on the training data. As a result, if a linear set were to train both a PCA and an autoencoder algorithm, it is expected that both would reconstruct the dataset with a similar accuracy. The only difference is that PCA is more efficient for these applications.

Autoencoders take advantage of neural networks to learn. They are data specific meaning they are only capable of compressing a dataset that they have been trained on making the algorithm less versatile for daily uses in non machine learning applications. But in almost all compression algorithms, some amount of information of the input feature vector will be lost. The amount of data lost can be configured by the dimension on the reduced characterized vector with a (k $\times$ 1).

A study by Tribhuvan University took a large dataset of grayscale images and sought to find a suitable compression ratio while crossing a predetermined threshold performance to ensure an adequate reconstruction. It was found that a compression ratio of 4:1 was ideal between the input vector dimension and the compressed representational vector [4].

3. Problem Formulation and Solution Approaches:

3.1. Problem Formulation

The overall objective of this project is to train pre-existing unsupervised autoencoders to perform dimensionality reduction based to optimize classification. We will have access to training data feature vectors, training data labels, testing data feature vectors, and testing data labels only to be used for performance evaluation purposes. The feature vectors ideally should be high dimensional data so we can observe the effect of dimensionality reduction.

The optimal approach for training an autoencoder for classification involves adjusting the autoencoder's cost function to incorporate classification error. Unfortunately, we didn't have the capabilities to modify the internal workings of the autoencoder in order to modify the cost function. Consequently, we must treat the autoencoder as a black box for the purposes of this project, limiting our modifications to the basic parameters of the autoencoder and its inputs. Our goal is to still train the autoencoder to consider classification when it develops its encoding solution while being limited to those surface level modifications.

3.2. Solution Approach

Our approach involved the integration of labels into the training data employed for the autoencoder. This way, the autoencoder's encoding and decoding solutions incorporated MSE contributions from both the labels and feature vectors. To achieve this, we appended each feature vector with its corresponding label (± 1). The intent was to leverage a sufficiently large label vector, with the anticipation that the autoencoder would produce an encoding solution prioritizing the MSE associated with the label vector and improving the classification rate.

However, a notable challenge emerged when applying our solution to the testing data, as the labels for this

dataset were unavailable. To address this limitation, we appended an equivalently sized "label" vector to our testing feature vectors, populated solely with zeros. Consequently, our proposed solution is only suitable for binary classification problems, wherein one class is designated as positive, the other class as negative, and zero serves as the intermediary label value appended to the testing data.

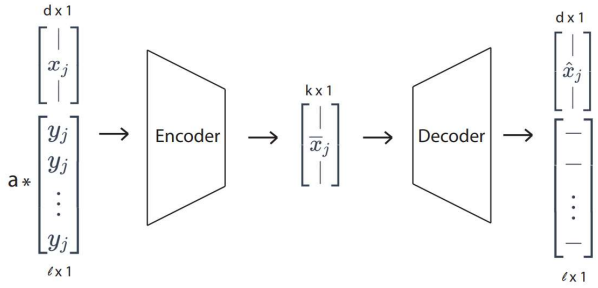

Figure 1. Modified Autoencoder Diagram

3.3. Parameters and Performance Evaluation

As seen in figure 1, the main parameters this project concerns itself with are the length of the appended label vector (l) and the scalar multiple of the label vector (a). We will also touch on how adding all zeros label vectors to our training and data/class selection affects our performance. Another significant parameter is the epochs that the autoencoder runs, as autoencoder performance heavily depends on epochs. However, due to computational complexity, we will not explore the effect of epochs on our algorithm as deeply and will use relatively low values for epochs.

Our performance evaluation will be based on the testing CCR obtained from Kernel SVM on the reduced, encoded testing feature vectors across different numbers of dimensions we reduce down to (k). While training our modified autoencoder, we will simultaneously train an unmodified autoencoder with the same training data without any label vectors. Once we have both encoding solutions, we will encode our testing data (appending the zeros vector to our input for the modified autoencoder) and compare the CCR obtained from Kernel SVM between these two encoders. The difference in the CCR between our modified autoencoder and unmodified autoencoder for the testing data will be our first performance parameter. In our case, we will do the performance of the modified minus the performance of the unmodified, meaning that a positive CCR difference is desirable where the modified is outperforming the unmodified and negative difference would show the opposite.

We will then reconstruct the feature vectors from the encoded vectors using our decoding solutions obtained from both autoencoders, and compare MSE of the original feature vectors. In the case of the modified decoder, it will reconstruct the testing vector with a “label” vector at the end which we will ignore for the purposes of MSE. The difference in the MSE between our modified autoencoder and unmodified autoencoder for the testing data will be our secondary performance parameter. In our case, we will do the performance of the modified minus the performance of the unmodified, meaning that a positive MSE difference is expected generally where the unmodified is outperforming the modified, and positive difference would show the opposite.

3.4. Time and Memory Complexity

Our program has extremely poor time complexity, since it retrains the autoencoder multiple times for the different k values. The runtime to train the modified autoencoder would theoretically be worse than an unmodified autoencoder due to the larger size of the training input, but we noticed no significant difference in runtime in our program. Another note is that high values of k and high values of epochs have considerably worse runtime. For a 10,000 sample training dataset being encoded to 500 dimensions from 784 dimensions, our autoencoder took approximately 10 minutes per 100 epochs using an AMD Ryzen 7 7730U at 100% utilization. Our program has poor time complexity as we would need to repeat this training for a range of k values, However, should this algorithm’s concept be implemented in a real-world environment where the desired k value is known, then the autoencoder would only need to be trained once, which would significantly improve the runtime.

Our modifications do affect the memory complexity as we are appending the label vectors and making our training input larger. Generally appended label vectors will be on the scale of d, the original number of dimensions for our data. The number of appended label vectors is on the scale of n, the number of samples of data. As a result, our memory complexity would go from $O(d*n)$ to $O(d^2 * n^2)$, which is significant.

4. Implementation

4.1. Data and Class Choice

For our data, we chose the MNIST numerical handwritten dataset. The images are 28 x 28px, which is 784px total per image. Different digits have different labels. For the binary classification, we chose digits ‘3’ and ‘8’ because they look somewhat similar, compared to ‘0’ and ‘1’ which we initially used for the binary classifier. With 0 and 1, we found the CCR reached 1 too quickly which made processing the data uninteresting. As

a result, we wanted to see if digits that look more similar would be harder to classify in order to better compare the performance between the original and modified autoencoders. With the time constraints on this project, we stuck with comparing two digits to make processing the data more manageable as these two classes include approximately 10,000 training and 2,000 testing data points.

We also used a smaller MNIST dataset with 16 x 16px images, with around 900 training points and 300 testing points for the two classes. This dataset was used mainly for developing the algorithm as the runtime was significantly better than the larger dataset. Our program however is generalized to work with either dataset, with just some slight differences in how the data is loaded into the program.

4.2. Code Implementation

The program first loads the *X_train* and *X_test* data into the program as matrices. Y_train is repeated *l* times, multiplied by *a*, then appended to *X_train* to create a new matrix, *X_train_appendedLabels*. A zero vector the same length of *Y_test* is repeated *l* times and appended to *X_test* to create a new matrix, *X_test_appendedZeros*.

Then the program loops the variable *k*, incrementing it by a specified step size until *k* is greater than the number of dimensions of our feature vectors. For each value of *k*, an unmodified autoencoder is trained to a hidden size of *k* using *X_train*. *X_train* is then encoded with the autoencoder and a binary classifier is trained with the encoded training data. Then *X_test* is encoded with the same autoencoder, and classified based on the previously trained binary classifier. We then use the true labels to get CCR and observe CCR behavior over *k*.

The same is repeated for *X_train_appendedLabels* instead of *X_train* and *X_test_appendedZeros* instead of *X_test*. We obtain an encoding solution based on an autoencoder trained with *X_train_appendedLabels*, which is the matrix *W* as seen in figure 2. *W* is first applied to the training data to encode it, and then a binary classifier is trained on the encoded training data. *W* is then applied to each column of *X_test_appendedZeros* to encode our testing data, and we classify our encoded testing data using the binary classifier. Then using *Y_test*, we compare the binary classifier labels with the true labels to obtain CCR and observe its behavior over different *k* values. The overall algorithm is summarized in the pseudo code in figure 10 in the appendix.

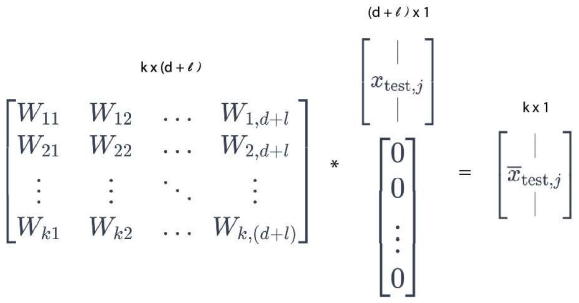

Figure 2. Encoding a Testing Feature Vector

5. Results

5.1. Binary Classifiers

When training several binary classifiers with the results from the original vs modified autoencoders, the results from the modified autoencoder performed consistently better. As seen by figures 3 and 4 below, the CCR for the modified autoencoder reaches 1, whereas the CCR for the original autoencoder reaches 0.9. The CCR plots for the original autoencoder are generally less smooth, and there is more fluctuation in CCR with different k values. This is expected because the modified autoencoder has the label as part of its input.

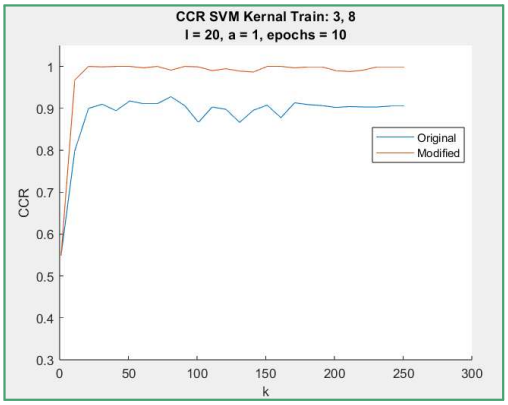

Figure 3. CCR of Kernel SVM (modified vs. original)

With the testing data, where we had to input all of the labels as 0, the modified autoencoder did worse than the original whereas the SVM linear and kernel performed similarly. With both of the SVM classifiers, neither outperformed the original significantly. There is lots of overlap in the CCR plots where both fluctuate around a CCR of 0.9. The kernel SVM did slightly better than the linear SVM, where the modified autoencoder slightly outperformed the original for the selected parameters. The trend with the kernel SVM is that the CCR is more "smoothed out", and there are fewer fluctuations in the short term CCR trends with respect to k with the modified

autoencoder. As a result, we decided to use the kernel SVM for the rest of our analysis because unsurprisingly, it performed the best across our tests.

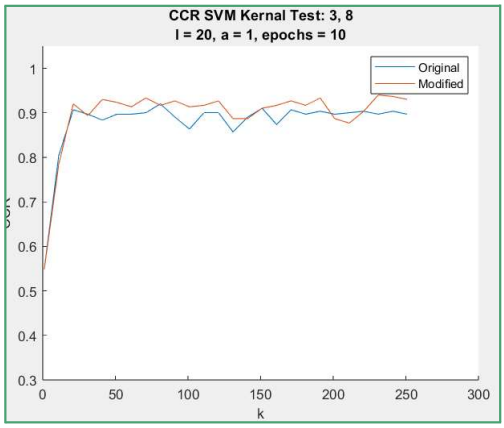

Figure 4. CCR Kernel SVM (modified vs. original)

5.2. Length

The two most impactful parameters were the length of the appended label vectors and the number of epochs that the autoencoder ran. The modified autoencoder did not perform significantly better than the unmodified autoencoder, generally having around the same performance under any parameter choices. There were some parameter choices we saw slight improvement, but in most cases, the modified performed the same or worse than the unmodified autoencoder.

With regards the length, generally relatively smaller lengths performed better. We expect the lower values of length to have a CCR difference of 0, since our modified feature vectors more closely resemble the unmodified feature vectors. The plots in figure 5 demonstrate the pattern that as length increases, the difference in CCR stays around 0, with the modified autoencoder improving in performance slightly, until the behavior starts to become extremely chaotic with longer lengths. With longer label vectors, we are appending more zeros to our testing feature vectors. Thus, when we encode the testing feature vectors with the large number of zeros appended, it perplexes our classifier as the encoded feature vectors contain a lot of reconstruction emphasis on zeros that aren't useful for classification.

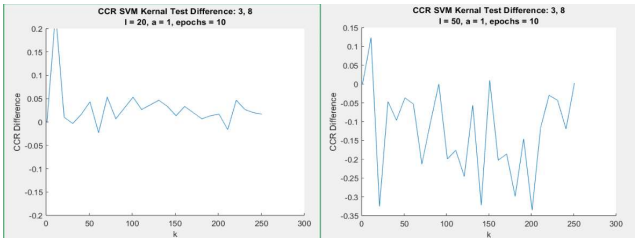

Figure 5. CCR Kernel SVM (l = 20 vs. 50)

Looking at the MSE difference in figure 6, we can see that it pretty fluctuates wildly around 0 as we increase length. Interestingly we see that the modified autoencoder does a better job reconstructing the original feature vector than the unmodified autoencoder, though both do a relatively poor job which can be seen in figure 9 in the appendix. Due to the low number of epochs used and the extreme amount of fluctuations for the modified autoencoder (the unmodified autoencoder has a more stable pattern), we infer that the modified autoencoder did not converge to an optimal solution. Thus, the patterns we observe are not conclusive.

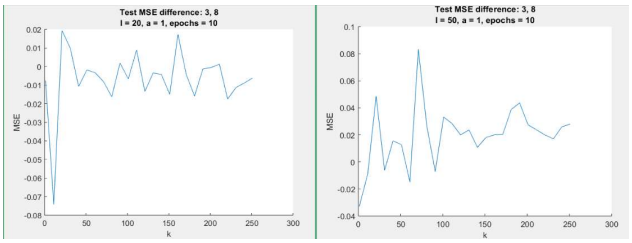

Figure 6. Testing MSE (l = 20 vs. 50)

5.3. Zeros and Scalar Multiplier

The other parameters we considered were the scalar multiple of the training label vectors (a), and including training points with an appended label vector of zeros. Figure 7 shows that the scalar multiple had negligible effect on both the CCR and the MSE differences, even comparing $a = 1$ with $a = 100$. It did make an impact on how quickly our training data with their appended labels converged, but since our testing data is appended with zero vectors, the scalar multiple modified autoencoder is trained with no effect.

Another consideration we had was to train the autoencoder with some training feature vectors that had zeros as a label vector in addition to the true label vectors, so that perhaps the autoencoder would be better equipped to handle the zero label vectors of the testing data. We duplicated the training feature vectors and used both to train our modified autoencoder. One would have its true label vector, and the other would have a zero vector. Figure 7 shows that this actually slightly worsened the performance of our modified autoencoder compared to only training with the true label, so we did not use it throughout the rest of the project.

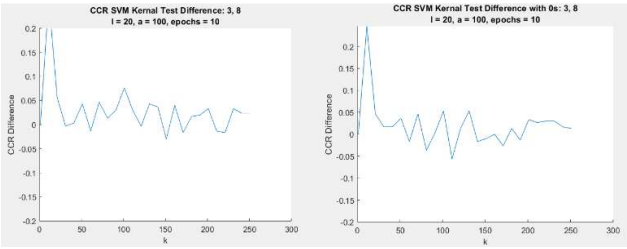

Figure 7. CCR Kernel SVM (with/without 0s input vector)

5.4. Epochs and Class Selection

As mentioned earlier, we presumed that our autoencoder did not converge to its optimal solution based on the fluctuating MSE graphs. If we increase the number of epochs, then we should see a more obvious MSE pattern, but due to the runtime, we weren't able to test it very thoroughly. We were able to do at least a preliminary run of the algorithm for epochs = 200 just to observe the behavior and got the results in figure 8. As we expected, MSE converges much more smoothly and reconstruction error improves significantly. Surprisingly, increasing the epoch number resulted in an inverse relation between CCR and *k*. We are not sure why this behavior occurs and weren't able to test it further, but it is a pretty clear relation. Much like with lower epochs, the modified autoencoder had around equivalent or worse performance to the unmodified autoencoder.

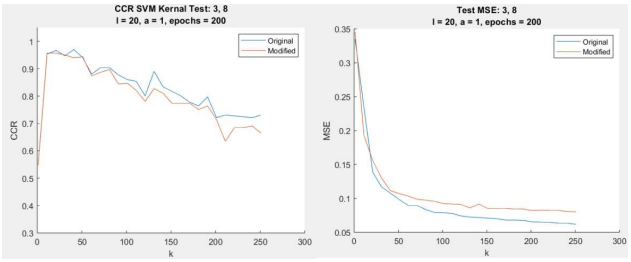

Figure 8. Testing CCR Kernel SVM (left), Testing MSE (right)

Ultimately, our results indicate that we don't get a significant performance benefit from appending label vectors to our data for autoencoders, and for most parameter choices, the CCR performance is equivalent or worse and the MSE ends up chaotic. We also did some quick test runs of our algorithm on other classes, such as having class -1 be digits 0-4 and class +1 be digits 5-9. Changing what data and classes we use obviously affects our results. While we got a slight performance improvement with the parameters l = 20 and epochs = 10 for digits 3 and 8, we did not get that same consistent performance improvement for 0-4 and 5-9. This suggests that for each class and dataset, the length and epoch number that our algorithm will do best for changes The optimal length and epoch number that we found for 3 and 8 can't be generalized for other classes and datasets, making our algorithm even less effective than it would originally appear.

6. Conclusion

Generally, our algorithm is not very useful. The algorithm is only somewhat productive for very specific length parameter values and epoch selections that are specific to the classes we chose. We took a lot of trial and error to find a good length and epoch selection which is why its implementation is not good for general use. In the future, we would expand on our project to find a quicker, more efficient way to find ideal epoch and length values which relies less on trial and error.

MSE values were very chaotic and inconsistent in short term trends with respect to k because of the low epochs we used for the sake of code runtime. While we conclude from the chaotic MSE graphs that we did not converge to an optimal solution, we also find that higher epochs give a better MSE but a generally worse CCR indicating a trade-off when selecting epoch values for our algorithm. To further develop our algorithm and make it more useful, we would take more time to explore behavior with a wider range of epoch values.

7. Description of Individual Efforts

7.1. Howell

Howell wrote section 3 (Problem Formulation and Solution Approaches), 4.2 (Code Implementation), and 5.2-5.4 (Results). Howell was also responsible for the algorithm development, implementation, and generating figures.

7.2. Nathan

Nathan was responsible for sections 1 (Introduction) and 2 (Literature Review) along with the abstract for the presentation as well as the report. Nathan was mainly researching compression algorithms like Huffman Coding and autoencoders.

7.3. Tara

Tara wrote section 4.1, 5.1 and section 6. Tara also presented section 5.3.

References

[1] Charles J. Turner and Larry L. Peterson. Image Transfer: An End-to-End Design. *University of Arizona*, Arizona, pages 258–262.

[2] D. Santa-Cruz, T. E. (2000). JPEG 2000 Still Image Coding Versus Other Sttandards. *Swiss Federal Institute of Technology*, Switzerland, 2000.

[3] O'Shea, A. A. (2009). Novel Lossy Compression Algorithms with Stacked Autoencoders. *Stanford University,* pages 2–4, 2009.

[4] Ashim Regmi. IMAGE COMPRESSION USING DEEP AUTOENCODER. *Tribhuvan University*, pages 12–23, 2016.

Appendix

Figure 9. Reconstruction of a Test Sample

```
Given X_train, X_test, Y_train, Y_test
X_train_appendedLabels = X_train with Y_train appended to the end
X_test_appendedZeros = X_train with zeros appended to the end


for k = 1:step_size:total_dimensions

    train autoencoderOriginal with X_train to hidden size of k
    encode X_train with autoencoderOriginal to X_train_encoded
    train binary classifier(s) with X_train_encoded

    encode X_test with autoencoderOriginal to X_test_encoded
    store CCR of the binary classifier(s) with X_train_encoded compared to Y_train
    store CCR of the binary classifier(s) with X_test_encoded compared to Y_test


    train autoencoderModified with X_train_appendedLabels to hidden size of k
    encode X_train_appendedLabels with autoencoderOriginal to X_train_appendedLabels_encoded
    train binary classifier(s) with X_train_appendedLabels_encoded and

    encode X_test_appendedZeros with autoencoderOriginal to X_test_appendedZeros_encoded
    store CCR of the binary classifier(s) with X_train_appendedLabels_encoded compared to Y_train
    store CCR of the binary classifier(s) with X_test_appendedZeros_encoded compared to Y_test

end for

plot CCR over k, comparing X_train_encoded vs X_train_appendedLabels_encoded
plot CCR over k, comparing X_test_encoded vs X_test_appendedZeros_encoded
```

Figure 10. Pseudo Code