2023

EEL6814: Project 2

BY JOSEPH MADDEN AND ZAIN NASRULLAH

IEEE

EEL6814: Project 2

line 1: 1st Given Name Surname
line 2: *dept. name of organization (of Affiliation)*
line 3: *name of organization (of Affiliation)*
line 4: City, Country
line 5: email address or ORCID

Joseph Madden
Electrical and Computer Engineering
University of Florida
Gainesville, FL USA
josephmadden@ufl.edu

Zain Nasrullah
Electrical and Computer Engineering
University of Florida
Gainesville, FL USA
z.nasrullah@ufl.edu

line 1: 4th Given Name Surname
line 2: *dept. name of organization (of Affiliation)*
line 3: *name of organization (of Affiliation)*
line 4: City, Country
line 5: email address or ORCID

line 1: 5th Given Name Surname
line 2: *dept. name of organization (of Affiliation)*
line 3: *name of organization (of Affiliation)*
line 4: City, Country
line 5: email address or ORCID

line 1: 6th Given Name Surname
line 2: *dept. name of organization (of Affiliation)*
line 3: *name of organization (of Affiliation)*
line 4: City, Country
line 5: email address or ORCID

***Abstract*—This report presents the development and evaluation of a Stacked Autoencoder Network (SAE) integrated with a Multi-Layer Perceptron (MLP) classifier for application on the Kuzushiji-MNIST dataset. The project aims to explore feature extraction for image classification by experimenting with various SAE configurations, particularly focusing on the dimensions of the bottleneck layer and network hyperparameters. Additions to this model include adding impulsive noise in the input data and employing a correntropy cost function in place of the traditional Mean Squared Error (MSE). Furthermore, the study introduces a distinctive penalty function in the SAE's reconstruction cost, designed to create discriminative code targets in the latent space, thereby enhancing classification accuracy. Comparative analysis using confusion matrices and computation time evaluations assesses the performance of the SAE+Classifier model against a baseline MLP classifier. The outcomes underscore the model's enhanced noise resilience and superior classification efficiency, underscoring its potential in refining deep learning strategies for image classification tasks.**

Keywords—stacked autoencoder network, image classification

Part 1

I. Introduction

In the realm of machine learning, deep learning techniques have revolutionized the approach to complex tasks, particularly in image classification. Among these techniques, Stacked Autoencoder Networks (SAE) have emerged as a powerful tool for feature extraction, enabling the transformation of high-dimensional data into a more manageable and representative form. This report focuses on the design and evaluation of an SAE integrated with a Multi-Layer Perceptron (MLP) classifier, applied to the Kuzushiji-MNIST dataset.

The Kuzushiji-MNIST dataset, a variant of the traditional MNIST dataset, presents unique challenges due to its intricate patterns and historical significance in Japanese literature. This project aims to explore the potential of SAE in enhancing feature extraction capabilities for this complex dataset. Central to this exploration is the investigation of various configurations of the SAE, particularly the impact of the size of the bottleneck layer and the choice of hyperparameters on the overall classification performance.

Moreover, this study delves into the robustness of the SAE under conditions of impulsive noise. By substituting the traditional Mean Squared Error (MSE) with a correntropy cost function, the project evaluates the network's capacity to handle noise in input data, a crucial aspect for real-world applications. Additionally, the introduction of a novel penalty function in the SAE's reconstruction cost aims to foster discriminative feature learning, potentially enhancing classification accuracy.

The comparative analysis of the SAE+Classifier model against a standalone MLP classifier offers insights into the effectiveness of the proposed methods. This report documents the methodology, findings, and implications of these innovations, contributing to the ongoing research in the field of deep learning and image classification.

II. Methodology

This study develops and evaluates a SAE with a MLP classifier for the Kuzushiji-MNIST dataset. The approach involves optimizing the SAE, particularly its bottleneck layer and hyperparameters, and adapting it to handle impulsive noise using a correntropy-based cost function. Additionally, a specialized penalty function is implemented to enhance feature discrimination.

A. Creating the Stacked Autoencoder Network

The architecture of the Stacked Autoencoder Network (SAE) was crafted using Python, employing the Keras-Tensorflow libraries. The network is structured with an input layer, five hidden layers, and an output layer. The input layer,

the first two hidden layers, and the bottleneck layer are all stored in one *Sequential* model and the remaining layers are stored in another. This split design is to make the later SAE+Classifier model easier to design.

The input layer is configured to handle arrays of 784 pixels, which correspond to 28x28 pixel grayscale images, with each pixel normalized within a 0 to 1 range. The output layer is the same, outputting the decoded image. The hidden layers are arranged in the sequence of 800-200-100-200-800 units. The central layer, hereafter known as the bottleneck layer, initially comprises 100 units and is designated as a hyperparameter for experimental variation. The activation function of the hidden layers is the Rectified Linear Unit (ReLU) function.

B. Tuning the SAE

The training process for the Stacked Autoencoder Network (SAE) is designed to optimize its performance using a substantial portion of the image data. Specifically, 70% of the image data is allocated for training, while the remaining 30% is set aside for validation purposes. To ensure a balanced representation, both the training and validation sets are stratified based on their class labels. The KMNIST dataset creators already provide the test set.

Key to the training process are the hyperparameters: the width of the bottleneck layer and the batch size for each training epoch. To efficiently tune these hyperparameters, KerasTuner from the Scikit-Learn library is utilized. The width/height of the bottleneck layer will vary in multiples of 5, ranging from 50 to 100 inclusive. For the batch sizes, the range considered is between powers of two from 32 to 256 inclusive. This approach results in a total of 44 distinct trials, allowing for a comprehensive exploration of the hyperparameter space.

Each model iteration is granted a maximum of 20 epochs for convergence, with the incorporation of callbacks to terminate training early. The specific callback criterion is set to halt training if there's no improvement of at least 0.05 in the loss over a span of 3 epochs. The chosen loss functions for this training are MSE and correntropy (tested separately), and the ADAM optimizer from the Keras-Tensorflow library is employed for learning. The labels for the training data will be the same as the samples for the training data.

C. Designing the SAE+Classifier

The encoder extracted from the Stacked Autoencoder Network (SAE) was integrated with a Multi-Layer Perceptron (MLP) classifier, forming a comprehensive classification model. This integration was also facilitated using the Keras-Tensorflow libraries.

The input layer of the MLP is aligned with the width of the bottleneck layer from the SAE, ensuring seamless integration and data flow between the encoder and the classifier. The output layer of the MLP classifier consists of 10 perceptrons, each corresponding to one of the classes in the dataset. The number and width of hidden layers is a hyperparameter and discussed in the following sub-section.

For the hidden layers, the Rectified Linear Unit (ReLU) function is chosen as the activation function. The output layer employs the softmax activation function, standard for classification problems. Additionally, the hidden layers of the MLP classifier are constructed using 'Dense' layers from the Keras-Tensorflow library.

D. Tuning the SAE+Classifier

The training process for the integrated SAE+Classifier model is meticulously designed to optimize its classification performance. The classifier component, a Multi-Layer Perceptron (MLP), is configured with several hyperparameters: the number of hidden layers, the number of perceptrons in each layer, and the batch size for training.

For the MLP structure, the number of hidden layers is varied among three options: 1, 2, or 3 layers. The perceptron count in each hidden layer can be one of the following: 5, 10, 15, 20, 100, 200, or 400, with uniformity across all hidden layers in any given trial. This uniformity is a constraint to reduce the search space. The choice in hidden layers is to make comparisons in network size and EEL6814: Project 1 possible. The batch sizes considered for training are 32, 64, 128, and 256.

Same as the SAE, the MLP has a maximum of 20 epochs to converge, with callbacks to end training early. The training will end early if there is not an improvement of at least 0.05 in the loss over a span of 3 epochs. ADAM is the chosen optimizer again.

The encoder from the SAE is kept static—its weights are not trained further during this phase—and its outputs are fed as inputs into the MLP classifier. This approach ensures that the feature extraction capabilities developed in the SAE are not lost while training the MLP classifier.

Moreover, to accommodate the variations in the encoder training, separate tuning instances are established for encoders trained on Mean Squared Error (MSE) and those trained on correntropy. This distinction allows for a comparative analysis of the impact of different loss functions on the overall model performance.

E. Adding Noise

As part of the examination of the SAE and its classifier, noise will be added to the training images to observe the effect on the model.

1) Modifying the Training Data

The procedure begins by adding impulsive noise to the training images. This is executed with a predefined probability of 10%, targeting the alteration of pixels. The noise introduced is set at a value of 167, representing a mid-range gray, to provide a noticeable yet representative level of distortion.

2) Re-Tuning the SAE

Following the noise addition, the SAE undergoes a re-tuning process. This tuning adheres to the same methodology as outlined in the "Tuning the SAE" section. The objective

Identify applicable funding agency here. If none, delete this text box.

here is to assess and optimize the network's ability to process and extract features from the noised data, maintaining the integrity of the information despite the added complexity.

3) Encoding and MLP Model Training

The encoded values, now derived from the noise-adapted SAE, are subsequently fed into new MLP models. These models are tuned in accordance with the procedures described in the "Tuning the SAE+Classifier" section. The focus here is to evaluate how the noise-influenced encoded data impacts the classification accuracy and to fine-tune the MLP classifiers accordingly.

III. Results

A. Hyperparameter Tuning without Noise

1) SAE with MSE Loss Function

For the SAE trained on MSE, the best hyperparameters were a bottle neck width of 100 PEs and a batch size for training of 64.

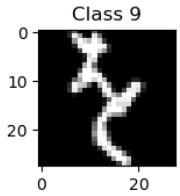

Fig.1. Image input to MSE SAE

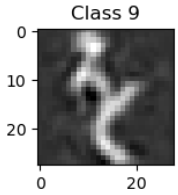

Fig.2. Image output from MSE SAE

2) SAE with Correntropy Loss

For the SAE trained on correntropy, the best hyperparameters were a bottle neck width of 100 PEs and a batch size for training of 64. The final validation score for this configuration was a bottle neck width of 100 PEs, a batch size of 32, and a sigma value of 1.0.

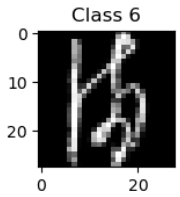

Fig.3. Image input from correntropy SAE

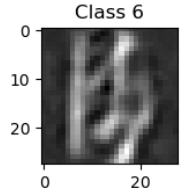

Fig.4. Image output from correntropy SAE

3) MLP Classifier with MSE Encoder

For the MLP classifier appended to the MSE encoder, the classifier had a width of 100 PEs, two hidden layers, a training batch size of 128, and a bottle neck width of 100 to connect to the encoder.

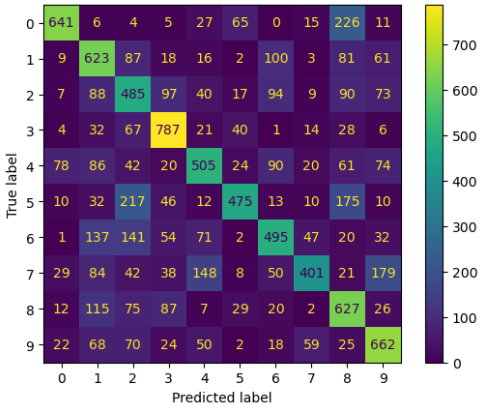

Fig.5. Confusion matrix for MSE SAE+Classifier (No Noise)

4) MLP Classifier with Correntropy Encoder

For the MLP classifier appended to the correntropy encoder, the classifier had a width of 100 PEs, one hidden layer, a batch size of 128, and a neck width of 100 to connect to the encoder.

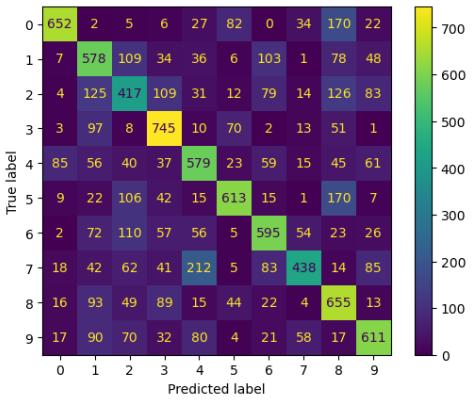

Fig.6. Confusion matrix for correntropy SAE+Classifier (No Noise)

B. Hyperparameter Tuning with Noise

1) SAE with MSE Loss Function

For the SAE trained on MSE, the best hyperparameters were a bottle neck width of 100 PEs and a batch size for training of 128.

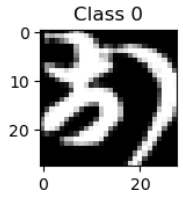

Fig.7. Noisy image input to MSE SAE

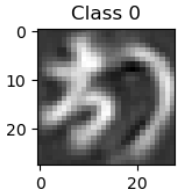

Fig.8. Noisy image output from MSE SAE

2) SAE with Correntropy Loss

For the SAE trained on correntropy, the best hyperparameters were a bottle neck width of 100 PEs and a batch size for training of 128. The final validation score for this configuration was a bottle neck width of 100 PEs, a batch size of 32, and a sigma value of 1.0.

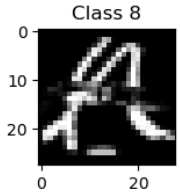

Fig.9. Noisy image input to correntropy SAE

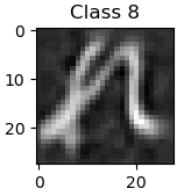

Fig.10. Noisy image output from correntropy SAE

3) MLP Classifier with MSE Encoder

For the MLP classifier appended to the MSE encoder, the classifier had a width of 100 PEs, three hidden layers, a training batch size of 64, and a bottle neck width of 100 to connect to the encoder.

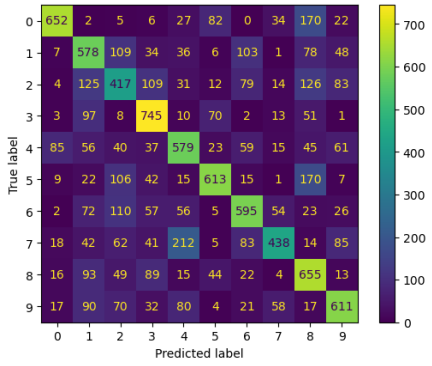

Fig.11. Confusion matrix for MSE SAE+Classifier (Noise)

4) MLP Classifier with Correntropy Encoder

For the MLP classifier appended to the correntropy encoder, the classifier had a width of 100 PEs, one hidden layer, a batch size of 64, and a neck width of 100 to connect to the encoder.

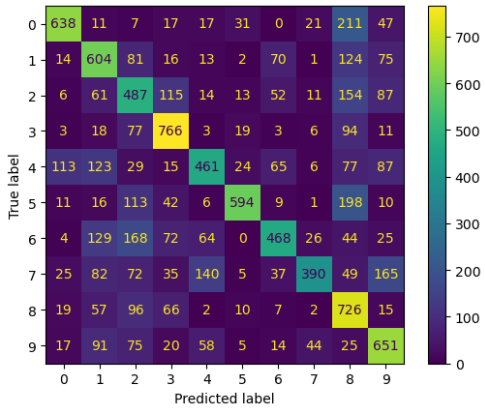

Fig.12. Confusion matrix for correntropy SAE+Classifier (Noise)

C. *Varying Bottleneck Size*

These results are from an MSE SAE+Classifier with all other hyperparameters held constant. The constant values were selected from the values chosen during hyperparameter tuning. A complete list of confusion matrices from 50-100 PEs can be found in the Jupyter Notebooks code.

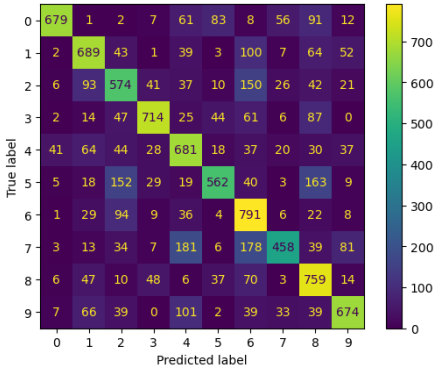

Fig.13. Confusion matrix for bottleneck_width=50

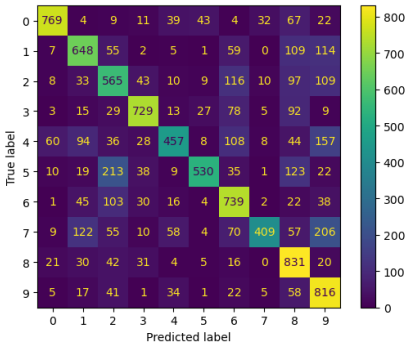

Fig.14. Confusion matrix for bottleneck_width=100

Part 2

I. Introduction

The point of part 2 of this project is to add a regularizer to the SAE reconstruction cost function, where part of what the regularizer does is calculate distances from the current codes produced by the encoder, to its constellation targets, where the

targets are in the space of the codes. This uses a modified cost function:

$$R = \frac{1}{k}\Sigma\, di$$

Where d_i is the distance between the current code and the current target. The overall cost function of the stacked auto encoder, SAE, is

J = L + lambda * R (Equation 1)

where lambda is the regularizer term.

II. Methodology

As seen above, this study implements a modified cost function that can retrain the SAE, for both a bottleneck width of 100, and a bottleneck width of 10.

A. Creating the Stacked Autoencoder Network from Part 1

A stacked auto encoder is used from part 1, to produce the codes that we can use as our ground truths. This encoder is build as a Sequential Model, where the layers go from input (784), 800, 200, neck_width, 200, 800, output (784). This SAE is inputted the training data, which is also the training labels (because we want the decoder to produce, from the codes, the same thing that the encoder has seen as its inputs to even produce those codes in the first place.

We can get the outputs of this encoder, that is trained on the training data. By taking the mean of these, we get "ground_truths_actual" in the code. This works as we assume the ground truth is Gaussian distributed, and by taking the mean of them, we can get the centers of these distributions where the codes are, and use these as the ground truths to help train future SAE's.

B. Creating the new Stacked Autoencoder Network that uses a Modified Cost Function

Now that we have our ground truths, we can make a new Stacked Autoencoder with a Modified Cost Function, which is *Equation 1* seen above. This model is a regular stacked auto encoder in structure, with some key differences- it now has 2 outputs- 1 output is the codes of the auto encoder, and the other output is the conventional output of an auto encoder, or the overall output of the network, from the output of the decoder part of the SAE. The output of the encoder, the codes, are subtracted by the ground truths, and the mean square of this is found to find R. This is then multiplied by lambda to the find the loss function of the encoder output.

The second output is the conventional output, or the decoder output, of an SAE. The Mean Squared Error reconstruction loss is used for this.

Building the model in tensorflow, we can see that the structure of the model looks like:

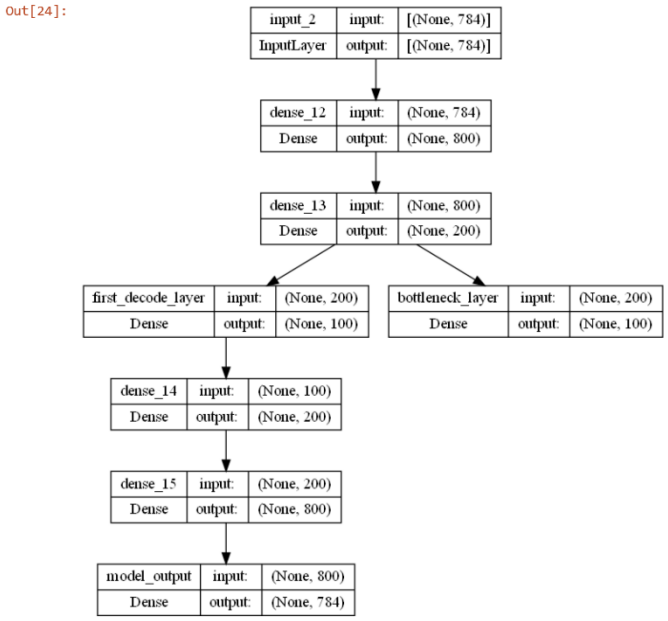

Fig 15: Structure of the new SAE model, with multiple outputs, as discussed above.

This is used first on a bottleneck layer of 100.

C. Using the Modified SAE with the new loss function using a bottleneck layer of 100

As seen in Fig 15, the 2 outputs of this model is the layer called "bottleneck_layer", and then the layer called "model_output".

We can see that the "model_output" layer's loss function implements L, where L is the MSE reconstruction cost. The layer "bottleneck_layer" implements the lambda * R part of the modified cost function. This is a mathematical representation of doing J = L + lambda * R, because overall loss function uses both of these two sub loss functions to calculate the overall loss.

The way this is implemented, is when this model is being compiled, where the model is "model_with_outputs" in the code, this model is compiled with a custom loss function that registers the "bottleneck_layer" to have its own custom loss function, and then the "model_output" layer to have its own custom loss function.

When "model_with_outputs" is fitted, the training data is the regular training data used before, but for the target labels, for the "bottleneck_layer", it is the ground truths calculated from before, and then the target labels for the "model_output" layer is the training data, as seen with a regular SAE. For the bottleneck layer, it'll output codes, which are automatically set to y_pred in the "loss_function_encoder" function, while the "ground_truths" is set to "y_pred" for the parameter of this function.

This same concept applies to the "loss_function_sae", where y_true turns out to be "data_train", and y_pred is the output of the decoder, or the attempted reconstruction of data_train from the codes that the encoder produces.

D. Building the Confusion Matrix from the new SAE architecture that uses the new Cost Function

In order to build the confusion matrix, we have the test labels, but we can't use the output of the SAE, as this is merely just images, nor can we use the output of the encoder, because the output is the codes, which is not in the same dimensional space as the test labels which we have to use for the confusion matrix.

To generate the confusion matrix, we first build an MLP, from Part 1 and the previous project. This consists of 2 hidden layers, with the first hidden layer having 100 processing elements and the second hidden layer having 64 processing elements.

The steps to build the confusion matrix are:

1. Predict the now trained SAE on the training data and the test data, to generate 2 new data vectors.
2. Train the MLP on the SAE's predicted output of the training data. We train the SAE using the training data, and we also train the MLP using the SAE output of the training data.
3. Now, we can predict the MLP on the SAE prediction of the test data. Take the argmax of this to generate a new data vector.
4. Finally, a confusion matrix can be generated from the output of step 3, in conjunction with the test labels.

The steps above outline how we can generate a confusion matrix once we have the SAE trained with the new cost function that we've been exploring.

E. Testing out the new SAE, but with the modified size of the bottleneck_layer being 10

The same steps in sections *C* and *D* are done, except now the encoder has a bottleneck size of 10. The results of this section will be talked about in the upcoming "Results" section.

F. Validating the selection of Lambda and R in the 3D space

To validate the selection of lambda and R, we first have to decide which variation of the new SAE to use- the one with a bottleneck layer of 100 or a bottleneck layer of 10- based off the confusion matrices, it is surprising to see that the SAE with a bottleneck layer of 10 performs better- there are more matches in the diagonals of the confusion matrices.

Once we have 10 to be the size of the bottleneck layer, we can run a hyperparameter search on different values of lambda- these different values of lambda will be passed into the cost function for the encoder, for loss function of the "bottleneck_layer".

The best value of lambda was found to be 0.01. From the model that used the lambda of 0.01, we see that the "bottleneck_layer" layer had a final loss of 6.9538e-5, or the final loss of the encoder, which has a loss function of $J_{encoder}$ = lambda * R. Thus, we can divide the above value by R to find an R that is validated in 3D space.

We can now retrain the model the new cost- this is done using a lambda value of 0.01, a bottleneck of 10, using the model that was used in the hyperparameter search. However, while this model is training, it passes in lambda = 0.01 into the cost function for the encoder so the encoder can use this optimal lambda value.

Finally, the confusion matrix is generated for this model that uses the validated values of lambda and R.

The codes from this model predicting on the test set are issued as class assignments to generate the confusion matrix, and the results of this will be seen in the "Results" section below.

III. Results

A. Varying Bottleneck Size

When testing out the new SAE (with the modified loss function that has a separate output for the encoder's codes and a separate output for the overall decoder), we see initially that the confusion matrix for a bottleneck size of 100 doesn't turn out the best:

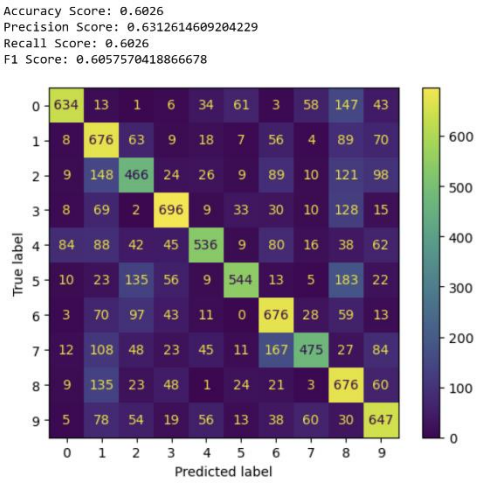

Fig 16: Confusion Matrix for a bottleneck size of 100.

As we can see, the accuracy is only 0.6026, with a lot of the samples being misclassified.

But when we try a bottleneck size of 10, we receive the following confusion matrix:

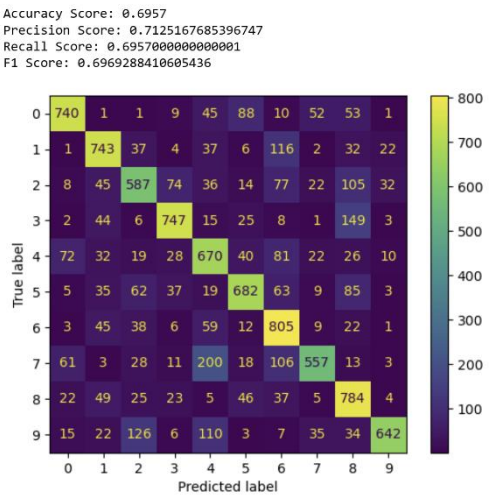

Fig 17: Confusion matrix for Bottleneck size of 10

With this, we receive a higher accuracy score of near 0.70. Thus, from here on out in the code, we decided to use a bottleneck size of 10. So projecting the images into a 10 dimensional space, and then producing the outputs from that seems to work best then projecting the images into a 100 dimensional space.

B. Hyperparameter tuning to find the best value of lambda and R

When choosing a bottleneck size of 10, a hyperparameter search for the best value of lambda was conducted. Different lambda values tried out were 0.0001, 0.005, 0.001, and 0.01. After training model with a bottleneck layer of 10 on all of these lambda values, a lambda value of 0.01 was found to give the best loss:

```
lambda value of 0.01 ...
1875/1875 [==============================] - 332s 177ms/step - loss: 0.0488
- bottleneck_layer_loss: 5.0437e-04 - model_output_loss: 0.0483
```

Fig 18: Output of loss for a lambda value of 0.01.

As we can see, the overall loss is 0.0483, which is lower than the other lambda values tried.

The loss for "bottleneck_layer", which represents lambda*R from *Equation 1*, is 0.00050437, and thus solving for R (and making lambda = 0.01), we get R = 0.050437. We can now use the model for lambda = 0.01, and retrain the model that we used for hyperparameter tuning. This shows that both lambda and R have been validated in 3D space, and are used to retrain the SAE. This results in a confusion matrix looking like:

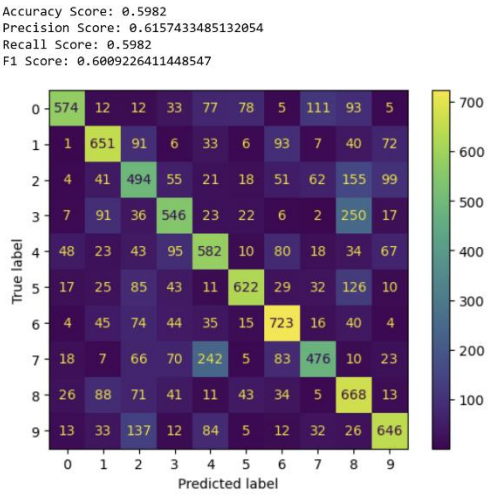

Fig 19: Confusion Matrix after Hyperparameter search

We see that the code ends up performing poorer, and this may be because the value of lambda was chosen at the end of the hyperparameter range we tried, so we just need to make the value of lambda even bigger. This concludes this section about Part 2.

IV. Discussion

When tuning the different SAE+Classifier models, there were some trends in the hyperparameters worth noting. The 100 PEs in the bottleneck layer was consistently the best choice for that hyperparameter. Worth noting is that other PE amounts yielded similar results (approximately >0.05 difference between them), yet there was a consistent downward trend in loss as more PEs were added. It would seem that 50 PEs in the bottleneck layer is sufficient to get highly predictive features with diminishing returns as the number increases.

This conclusion is further evidenced from the results of testing multiple bottleneck layer widths. The confusion matrices from each configuration were very similar and the accuracy metrics were close as well.

A 100 PEs in the classifier MLP for every model is another trend that makes sense. As referenced in the project 1 paper, a good choice for hidden layers is the sum of the number of inputs and half of the number of outputs [1]. This would be 105, which is closest to 100 in our hyperparameter tuning.

In regards to adding noise to the data, the results show that correntropy outperforms MSE for training the SAE for the SAE+Classifier. This is likely due to the nonlinearity of the correntropy loss function. The gaussian kernel assigns less weight to outliers whereas MSE assigns weight uniformly to outliers.

For Part 2, it seems that using an SAE with the modified cost function in general gave us better confusion matrices than in

Part 1, where we only used MSE or cross entropy. If we compare *Figure 5* and *Figure 6*, the confusion matrices of SAE's with the modified cost function, as in *Figure 16* and *Figure 17*, we see that the latter perform better, the confusion matrices show higher accuracy. This makes sense, as now we have a cost function that takes into account the output of the decoder, combining it with a cost function that takes into account the output of the encoder- this gives us more information and lets us better train the SAE.

V. Conclusion

SAEs are capable of automatic feature engineering, utilizing nonlinearities to find hierarchical features in the data. In our model, we found a 100 PEs in the bottleneck layer to be a consistently good choice for image classification.

In addition, the correntropy loss function is a great candidate for image classification. It filters noise well due to its nonlinearity and is an efficient measure of similarity between two datasets.

We can also observe that adding a modified cost function that takes into account 1) the MSE reconstruction cost of the decoder and 2) distances between current codes and current targets (in the space of the codes) for the encoder, we see that this trains SAE's to be more accurate than if the loss function were to only take into account 1), as seen in Part 1. This tells us that in the future, for stacked autoencoders, our loss functions should consider both the encoder AND the decoder when training these models for accuracy.

Acknowledgment

[1] Sheela, K. G., & Deepa, S. N. (2013, June 20). Review on methods to fix number of hidden neurons in neural networks. Mathematical Problems in Engineering. https://www.hindawi.com/journals/mpe/2013/425740/