Autoencoders for Kuzushiji-MNIST Digit Classification

Matthew Ruppert, Zachary Wilkerson

I. Introduction

Autoencoders are a type of unsupervised neural network where our input data is projected into a latent space representation (encoded) before being used to reconstruct the input data as the output of the model (decoded). Auto encoders have many applications including, but not limited to, classification and data generation. Typically, when autoencoders are used for classification problems, only the trained encoder is used such that another model such as a support vector machine (SVM) or multilayer perceptron (MLP) takes the latent representation of the data as input; however, it is also possible to use the latent space itself for classification or data generation by comparing the latent projection of the data to a prior distribution.

A common method of training autoencoders is using mean squared error (MSE) as the loss function. MSE is an easy metric to compute and does generally does a good job of minimizing the error between reconstructed input and the input itself. Depending on the application; however, other loss functions may be more appropriate to ensure salient features of the input data are preserved.

Information Theoretic Learning (ITL) tells us that autoencoder accuracy can further be refined by comparing the latent output to a prior distribution and using the loss of that comparison as a regularizer. This regularization method increases the discrimination in the latent space [1].

In this project, we compare the accuracy of an autoencoder with and without an ITL regularizer alongside an MLP to a convolutional neural network (CNN) on the task of character classification on the Kuzushiji-MNIST dataset.

II. Description

A. Kuzushiji-MNIST Dataset

In this project, the Kuzushiji-MNIST dataset is used. This dataset, a relative to the MNIST dataset, consisting of 70,000 images in 28x28 grayscale format corresponding to 10 categories of Japanese handwritten characters. A few example images from the Kuzushiji-MNIST dataset are shown in Figure #1.

Figure 1: Sampled images from the Kuzushiji-MNIST dataset.

We set aside approximately 10% of the training data, with equal samples from each of the 10 categories, to be used as our validation set.

B. Designing an Autoencoder Network

Our goal is to design an autoencoder network that we can use to classify our dataset. We will begin with building our stacked autoencoder (SAE). For this project, we use a 5 hidden layer network following an 800-400-XXX-400-800 layout, where XXX represents the number of units in the bottleneck layer. All these hidden layers have ReLu activation functions. The output layer has dimension 784 (28*28), equal to the input image dimensions, with a sigmoid activation function. The topology of our encoder can be seen in Figure 2. We will train our model end-to-end using backpropagation. We determine the number of units in the bottleneck layer, as well as the other hyperparameters of our model, using a strategic iterative process. We choose a minimum and maximum value for each parameter of the model, calculate the midpoint value, and then evaluate our model at all three points. We then inspect the MSE loss curves for the training and validation test sets for each of the three points and choose the region which provides a curve that appears to not be overfitting or underfitting. This process allows us to speed up the search of our parameters, compared to a random search procedure, as we do not need to test every value in the range. Once we reach a point where visual inspection of the graph to differentiate performance becomes difficult, we then check all remaining values in the range to find the value that results in the minimum validation MSE. We will use early stopping based on the validation MSE to determine the best performing number of epochs.

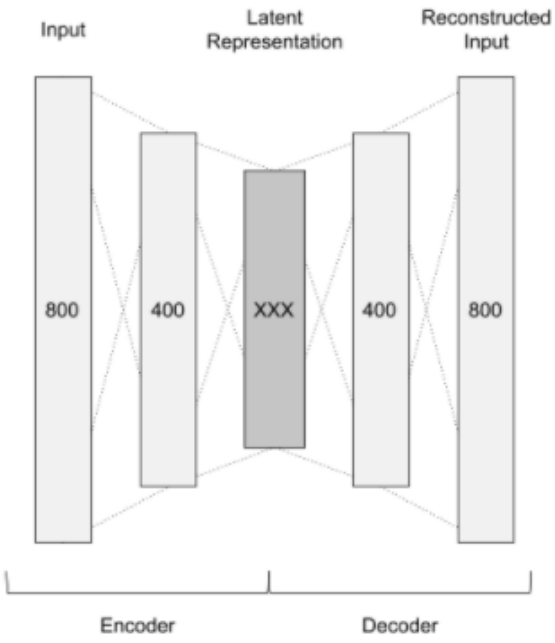

Figure 2: Topology of autoencoder.

We decided to determine the best performing topology – number of units in the bottleneck layer – first, as we believe the topology of the network is the biggest determiner of the model's success. We know that having a large number of units in the bottleneck layer could cause the model to learn features of the data that are redundant or non-essential, whereas having too little units in the

bottleneck layer could cause the model to not learn enough features needed to represent the data. We then move on to tune the learning rate and batch size simultaneously, for they are associated with one another. Finally, we determine the best performing number of epochs to train with using early stopping to avoid overtraining. For each step, we will set the remaining hyperparameters that haven't been fixed by the previous steps. The Adam optimizer was used for this autoencoder model for it provides an adaptive method of trans versing the gradient of the loss towards a local minimum.

We then move on to building our MLP for the task of classifying the bottleneck layer outputs. We decided to use a single hidden layer MLP for this task, as we believe that the latent space is sufficiently distilled for classification and does not require additional hidden layers to summarize input features. The input layer will take in the 225-dimensional outputs from our autoencoder bottleneck layer, send this through a dense layer with ReLu activation functions, and finally send this through our output layer of dimension 10 with a softmax activation function. The number of units in the hidden layer, the learning rate, and the batch size of the MLP will be learned using the same process as used for the autoencoder, with the modification that we will choose models that maximize validation accuracy instead of minimizing the validation loss as we did in the autoencoder case. The Adam optimizer was used for this MLP model, alongside a categorical cross-entropy loss function.

C. Designing a Penalty Function that Enhances Discrimination in the Latent Space

To enhance discrimination in the latent space we used two different prior distributions to better organize the latent representation of the data. A two-dimensional swiss roll and a uniform random distribution of labels (0-10) that has been one hot encoded. A two-dimensional swiss roll was chosen for it was easier to visualize the resultant prior. The distribution for the one hot prior was uniform for all classes were equally distributed in the training set.

Three different cost penalties were experimented with to create the most organized latent representation. The first was the use of the Cauchy-Schwartz divergence (CSD) multiplied by a regularizer (lambda) to encourage the latent space to prior distribution (Figure 3). The second and third approaches used a binary discriminator model of the same architecture as the decoder portion of the autoencoder apart from using a 1-dimensional output layer instead of the original 784 (Figure 4). Using this discriminator model, we were able to create a data driven cost metric instead of purely statistical one. We used the sigmoid cross entropy with logits cost function as the loss function for the discriminator which was comparing latent representation and prior distribution. We also experimented with further penalizing the latent output with the CSD. In all cases these organizational penalties were only applied to the encoder portion of the autoencoder through variable scoping and the use of multiple optimization functions. The underlying structure of the autoencoder remained the same from part one, with the exception of changing the dimension of the bottleneck layer to match the respective prior distribution, two for swiss roll and ten for one hot.

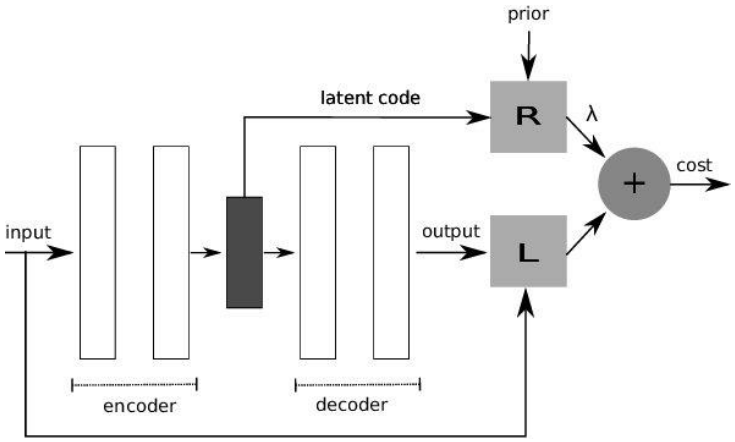

Figure 3: Topology of autoencoder with ITL regularizer [1].

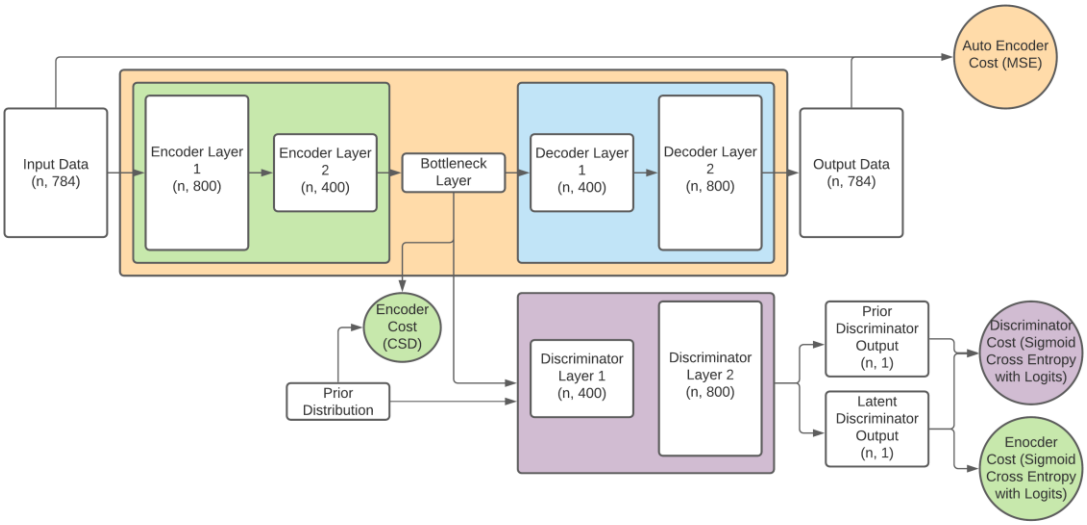

Figure 4. Topology of autoencoder with ITL and data driven regularization.

III. Evaluation

A. Optimizing the bottleneck layer of an autoencoder network

We began the search using the values [1,200,400] as the bottleneck layer dimensions. Using the selection process described above we arrived at an optimal value of 225 units for the bottleneck layer. We trained out models with an Adam optimizer with a set learning rate of 0.01, for 50 epochs with batch sizes of 128 images. A few examples of loss curves for select values for our hidden units are shown in Figure 5.

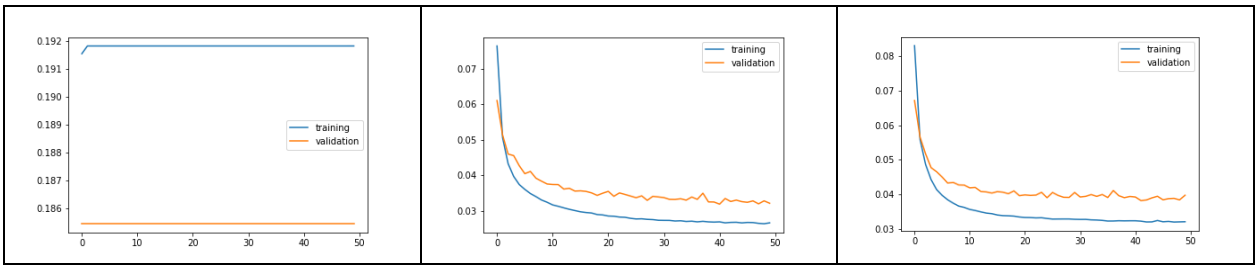

Figure 5: MSE loss curves for 50 hidden units (left), 175 hidden units (middle), and 250 hidden units (right) in the bottleneck layer of our autoencoder.

We see that if the number of hidden units is too low, such as 50 units, our model still has more to learn as indicated by the loss for both training and validation data being large. As we increase the number of units to too large a value, such as 250 units, we see that the error between the training and validation data becomes too large and therefore our model is overfitting. As we move into the middle range, such as 175 units, we see that the model is still overfitting but slightly less as the difference between the training and validation loss is smaller than for larger unit values. With our chosen value of 225 hidden units in the bottleneck layer, we still see this type of overfitting curve but we are able to achieve the smallest difference in loss. As we adjust the other parameters, such as learning rate, batch size, and number of epochs, we expect to reduce the overfitting occurring and thereby obtain learning curves that show a much better fit. With this selected topology – 800-400-225-400-800 – we get a model with 2,077,809 trainable parameters.

We then move on to selecting the best performing learning rate and batch size. We begin with [0.00001, 0.001, 0.1] as our initial learning rate values and [128,1024,2048] as our initial batch size values. A few examples of loss curves for the selected values for our learning rate and batch size are shown in Figures 6-8. We see in Figure 5 that using too small a learning rate such as 0.00001, no matter the batch size, our model still has the capability to learn further as the loss curves haven't stabilized. Using too large a learning rate such as 0.1, no matter the batch size, we see in Figure 6 that our model has not learned enough as indicated by the loss for both training and validation data being large. With such a large learning rate, we are rattling around and unable to get close to the absolute minimum because our steps are too large. Using a learning rate in the middle such as 0.001, we see in Figure 8 that our loss curve starts to stabilize indicating we are getting closer to a good fit. With that being said, we will test more mid-range learning rates between 0.001 and 0.00001. We see that using a smaller batch size, such as 128, gives us smaller loss values than a larger batch size, such as 1024 or 2048, and so we will test more batch sizes of small values.

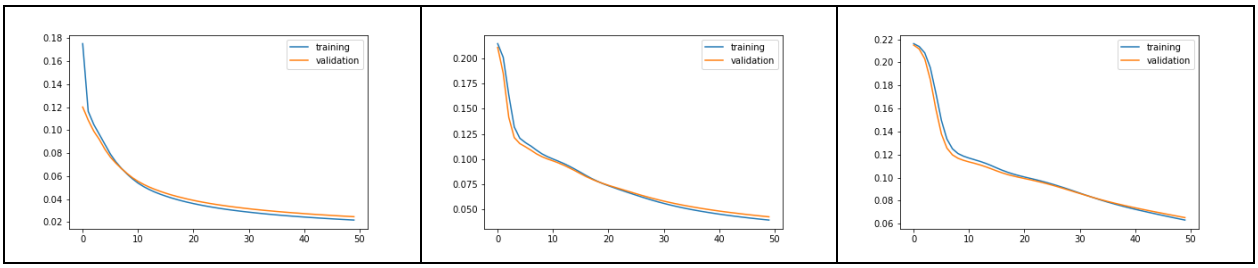

Figure 6: MSE loss curves for autoencoder with a learning rate of 0.00001 with batch sizes of 128 (left), 1024 (middle), and 2048 (right).

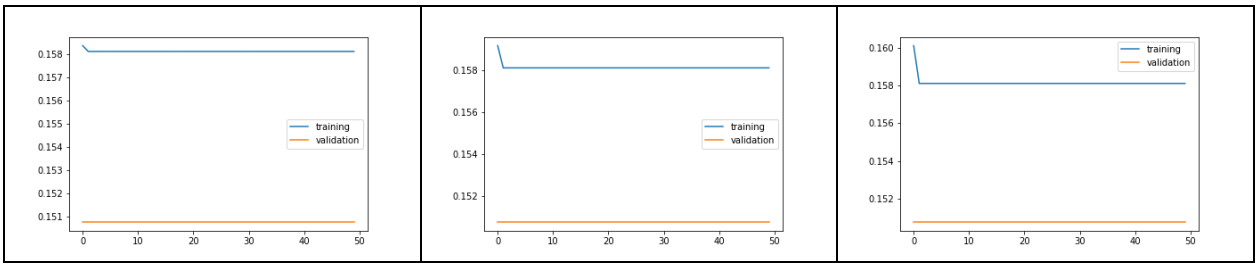

Figure 7: MSE loss curves for autoencoder with a learning rate of 0.1 with batch sizes of 128 (left), 1024 (middle), and 2048 (right).

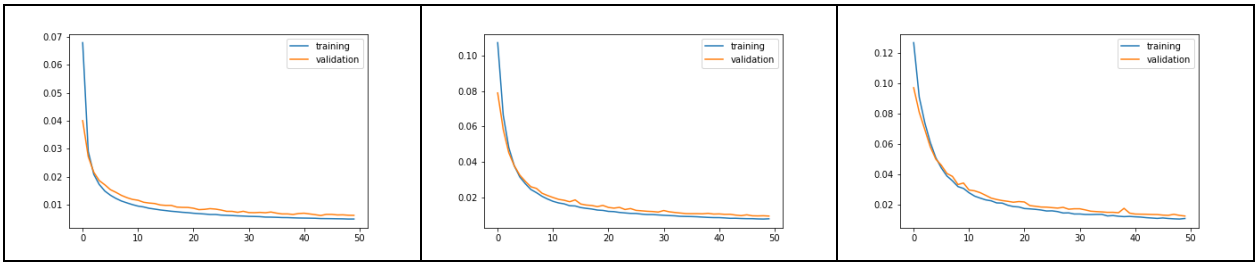

Figure 8: MSE loss curves for autoencoder with a learning rate of 0.001 with batch sizes of 128 (left), 1024 (middle), and 2048 (right).

Using our selection process, we determine our best learning rate to be 0.0008 and our best batch size to be 164.

Finally, we move on to selecting the best number of epochs to train our autoencoder for. With 225 units in the bottleneck layer, a learning rate of 0.0008, and a batch size of 164, we train our model using early stopping, where we stop training based on the validation loss not changing by more than 0.001 per epoch with a patience of 5 epochs. Using this method, we determine that the best number of epochs to train our model for is 25 epochs. Based on Figure 9, we see that our training and validation loss both begin to stabilize around this epoch value, so intuitively this makes sense. We also see that the difference between training and validation loss is very minimal, indicating that we have very little overfitting occurring and therefore are obtaining a model with relatively good fit.

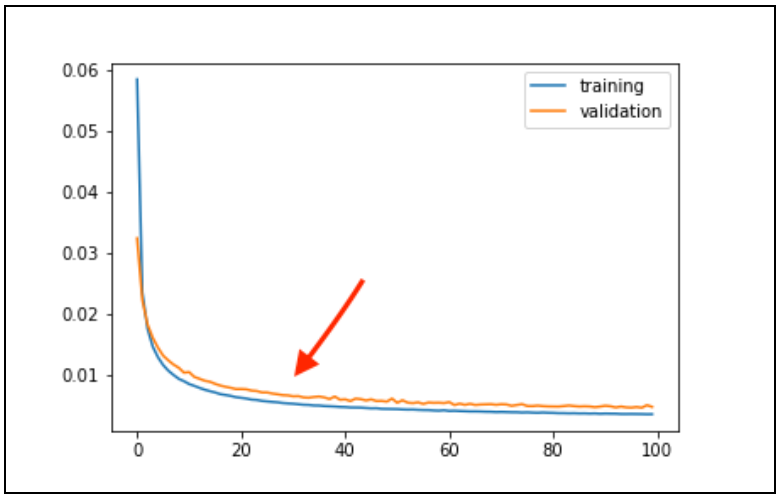

Figure 9: MSE loss curve for optimized autoencoder with arrow indicating epoch value to stop training based on early stopping criteria.

To visualize how well our autoencoder performs, we present test images examples before and after running through our model. The results are show in Figure 10. We see that there is no distinguishable difference between the original image and the reconstructed image to the naked eye.

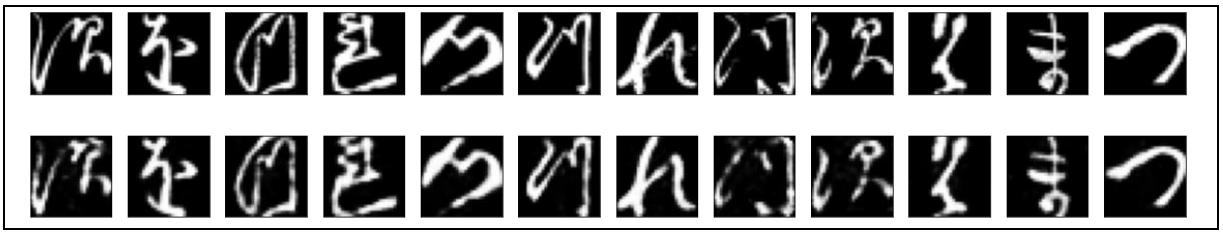

Figure 10: Sampled images (top) and their reconstructions (bottom) from the Kuzushiji-MNIST dataset.

Now that our autoencoder has been trained, we will build our MLP to take the latent representation from our autoencoder as input and classify our characters.

We began with [1,200,400] as our initial hidden unit values for the MLP, and using our selection process described earlier we decided to use 200 hidden units after training and testing our models with an Adam optimizer with a set learning rate of 0.01, for 50 epochs with batch sizes of 128. A few examples of loss curves for select values for our hidden units are shown in Figure 11.

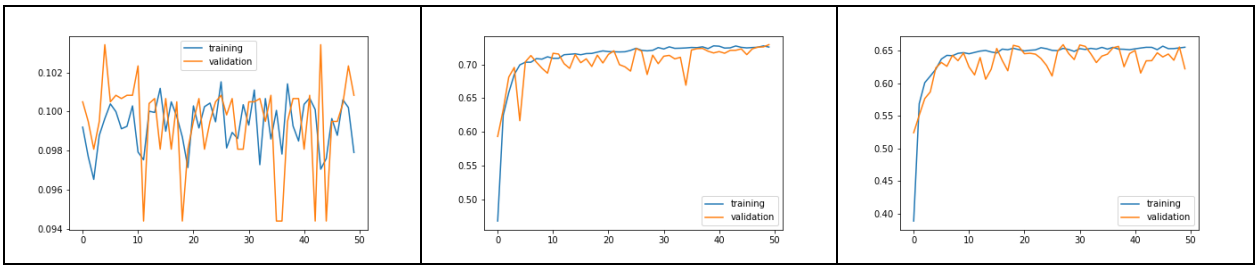

Figure 11: Accuracy curves for 50 hidden units (left), 150 hidden units (middle), and 275 hidden units (right) in the hidden layer of our MLP.

We see that if the number of hidden units is too low, such as 50 units, our model still has more to learn as indicated by the very small accuracy values for both training and validation data. As we increase the number of units to too large a value, such as 275 units, we see that our training and validation accuracy curves are following closely to one another, meaning we have a good fit, but the accuracy values themselves are too low to obtain good performance. As we move into the middle range, such as 150 units, we see that the model is still giving a good fit, as the training and validation curves move together, and the accuracy is higher than for the other number of units observed. With our chosen value of 200 units in the hidden layer, we were able to achieve the largest validation accuracy while still maintaining the closely moving training and validation curves as discussed previously. As we adjust the other parameters, such as learning rate, batch size, and number of epochs, we anticipate our model will be able to learn the data better and therefore increase our accuracy values. With this selected topology, we get a MLP with 47,210 trainable parameters.

We then move on to selecting the best performing learning rate and batch size. We begin with [0.00001, 0.001, 0.1] as our initial learning rate values and [128,1024,2048] as our initial batch size values. A few examples of loss curves for the selected values for our learning rate and batch size are shown in Figures 12-14. We see in Figure 12 that using too small a learning rate such as 0.00001, our

model still has the capability to learn further as the accuracy curves haven't stabilized. Using too large a learning rate such as 0.1, no matter the batch size, we see in Figure 13 that our model has not learned enough as indicated by the low, fluctuating, accuracy values for both training and validation data. With such a large learning rate, we are rattling around and unable to get close to the absolute minimum because our steps are too large. Using a learning rate in the middle such as 0.001, we see in Figure 14 that our loss curve starts to stabilize, indicating we are getting closer to a good fit. With that being said, we will test more mid-range learning rates between 0.001 and 0.00001. We see that using a smaller batch size, such as 128, gives us higher accuracy values than a larger batch size, such as 1024 or 2048, and so we will test more batch sizes of smaller value.

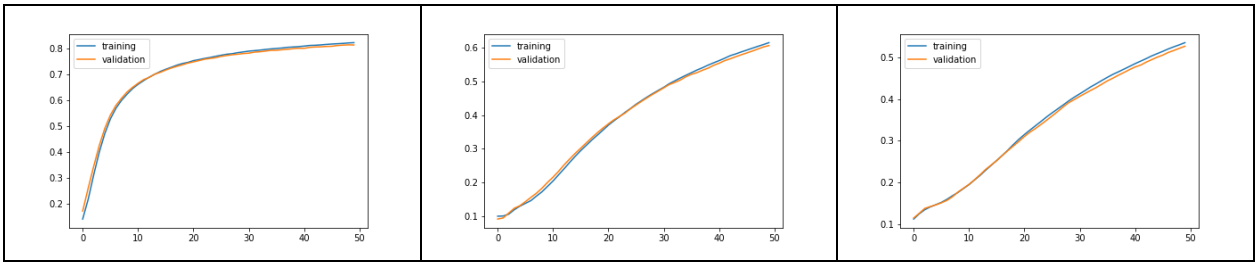

Figure 12: Accuracy curves for MLP with a learning rate of 0.00001 with batch sizes of 128 (left), 1024 (middle), and 2048 (right).

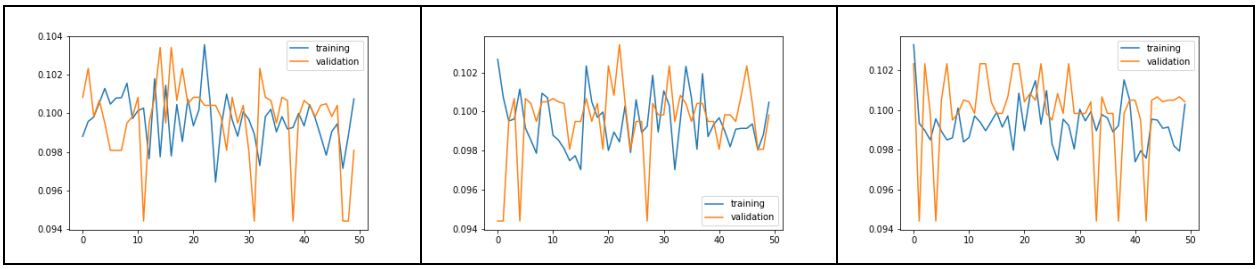

Figure 13: Accuracy curves for MLP with a learning rate of 0.1 with batch sizes of 128 (left), 1024 (middle), and 2048 (right).

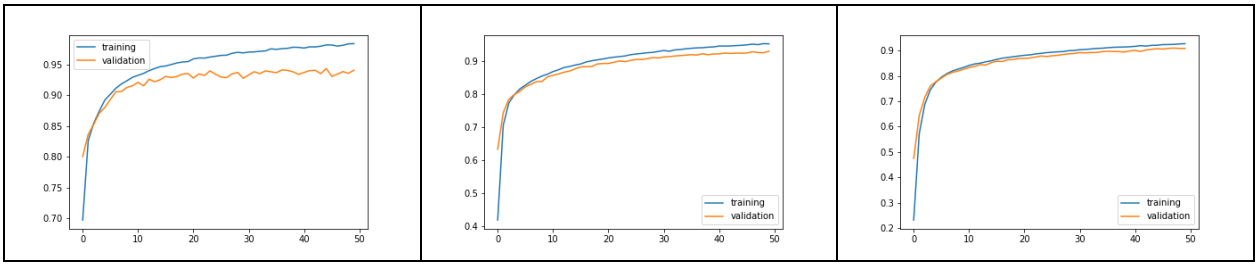

Figure 14: Accuracy curves for MLP with a learning rate of 0.001 with batch sizes of 128 (left), 1024 (middle), and 2048 (right).

Using our selection process, we determine our best learning rate to be 0.0006 and our best batch size to be 140.

Finally, we move on to selecting the best number of epochs to train our MLP for. With 200 units in the hidden layer, a learning rate of 0.0006, and a batch size of 140, we train our model using early stopping, where we stop training based on the validation accuracy not changing by more than 0.001 per epoch with a patience of 5 epochs. Using this method, we determine that the best number of epochs to

train our model for is 30 epochs. Based on Figure 15, we see that our validation accuracy begins to stabilize around this epoch value, so intuitively this makes sense. We evaluate our model on our test set and obtain an 88.0% accuracy.

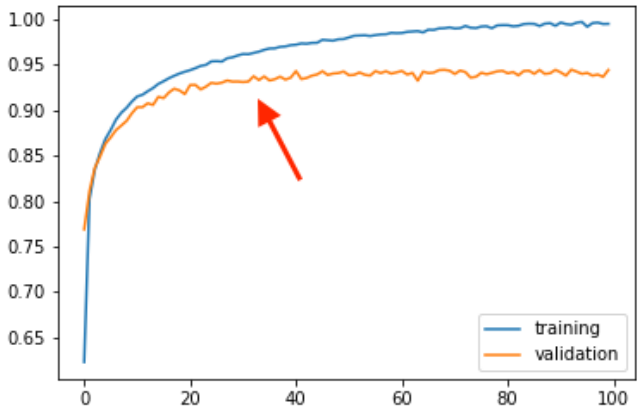

Figure 15: Accuracy curve for optimized MLP with arrow indicating epoch value to stop training based on early stopping criteria.

B. Designing a Penalty Function that Enhances Discrimination in the Latent Space

I. Version 1 (CSD only)

Unlike in part 1 of this project where the number of units in the bottle neck layer is variable, based on the prior distributions we have chosen, that number is fixed. Additionally, based on the recommendation to use a batch size of 1000 in order to preserve meaningfulness in the CSD comparison between latent representation and prior distribution, we were left with four hyper-parameters to optimize: regularization constant (λ̃), kernel size (σ), learning rate (lr), and number of epochs. We began the tuning process by optimizing λ̃ and σ for they are interrelated and will most significantly affect the loss ITL based loss function. We used a lambda ranging from [1 to 1000] and a sigma ranging from [1 to 20] using a learning rate of 0.001 and 20 epochs. We found the optimal value of sigma and lambda to be 5 and 32 respectively. These optimal values were chosen based on visual inspection of the resultant latent-swiss roll dataset created by the autoencoder (show in Figure 16). We then tuned the learning rate and number of epochs together for they are interrelated using learning rates between [0.1 and 0.0001] and epochs [10 to 100] using the same visual inspection method. The optimal learning rate and number of epochs was found to be 0.004 and 34 respectively. The resulting swiss roll using these parameters is shown in Figure 17.

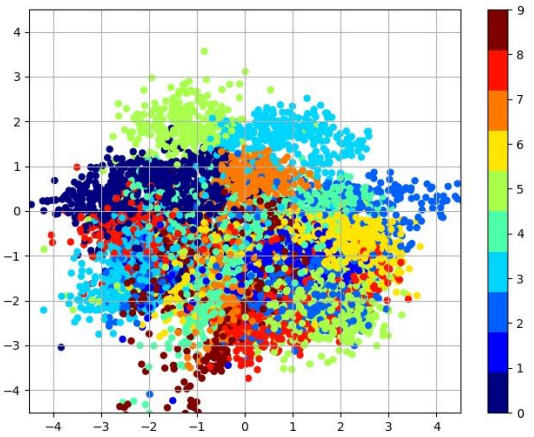

Figure 16: Swiss roll distribution given lr=0.001, lambda=32, and kernel size=5 when trained using 20 epochs.

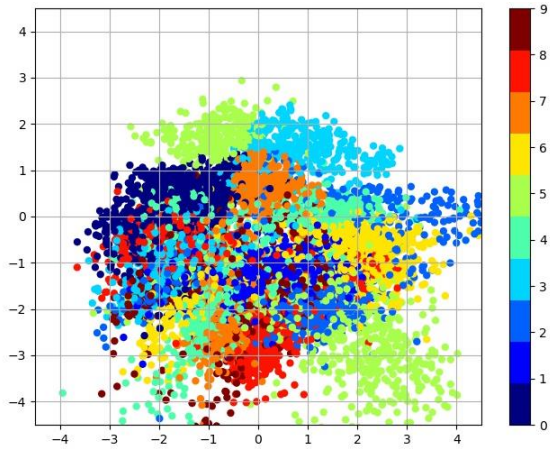

Figure 17: Swiss roll distribution given lr=0.004, lambda=32, and kernel size=5 when trained using 34 epochs.

The one hot encoded prior randomly sampled from the label space was only evaluated using CSD as the loss function. This decision was made because the one_hot prior was designed to be statistically similar to the latent representation as a whole, but would be very different from individual observations, thus making it easy for a discriminator model to tell the difference between the prior and the latent representation.

We first selected the optimal lambda and kernel size from the sample spaces [0.5 to 300] and [1 to 20] respectively and found ƛ=0.5 and σ=7 to be the best performing based on accuracy of using the latent space output as a classifier for the class of the input data. A log SoftMax function was applied to the latent representations before choosing the most probable class. The learning rate and epoch length were then selected from sample spaces [0.1 to 0.0001] and [10 to 50] respectively. The best performing model had epoch length of 20 and learning rate of 0.001. The confusion matrix shows that data points are most likely to get misclassified as class 1 (Figure 18).

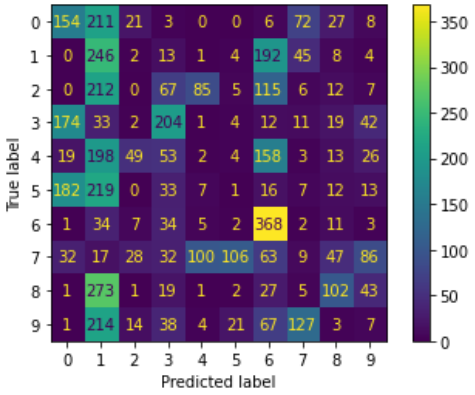

Figure 18: Confusion matrix of one hot encoded prior from label space.

II. Version 2 (Discriminator only)

With version 2, the only hyperparameters to optimize were learning rate and number of epochs, since the rest of the hyperparameters (hidden layer sizes, bottle neck size) were fixed. We then tuned the learning rate and number of epochs together for they are interrelated using learning rates between [0.1 and 0.0001] and epochs [10 to 100] using the same visual inspection method. We found that using a learning rate of 0.001 and 45 epochs gave us the best looking swiss roll, as shown in Figure 19.

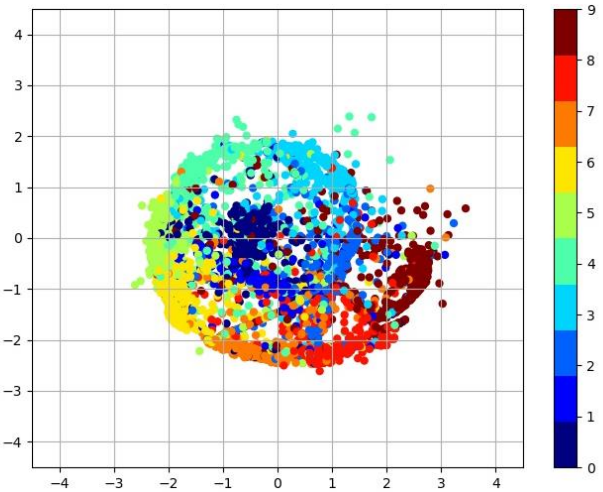

Figure 19: Swiss roll distribution given lr=0.001 when trained using 45 epochs.

III. Version 3 (CSD and Discriminator)

Version 3 has the same hyperparameters as version 1 and was tuned in an identical manner. The optimal values for lambda, kernel size, learning rate, and number of epochs were 25, 4, 0.006, and 40 respectively.

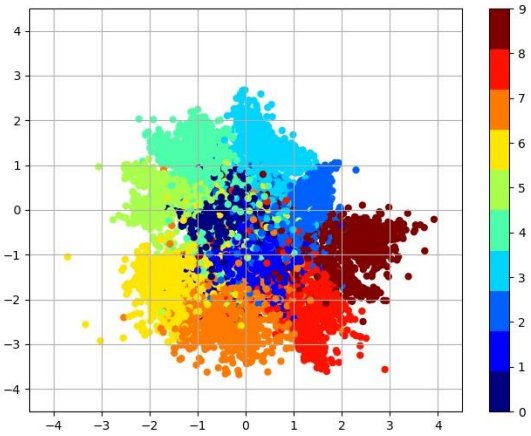

Figure 20: Swiss roll distribution given lr=0.006, lambda=25, and kernel size=4 when trained using 40 epochs.

IV. Classification

We then take our latent representation from one_hot prior autoencoder and run it through an MLP which was trained using the same procedure as used in Part A. We find that the best performing MLP, in terms of validation accuracy, had: 150 hidden units, a learning rate of 0.0009, a batch size of 220, and 80 epochs to be trained with. Figure 21 shows our training and validation accuracy curves. We evaluate our model on our test set and obtain an 78.1% accuracy.

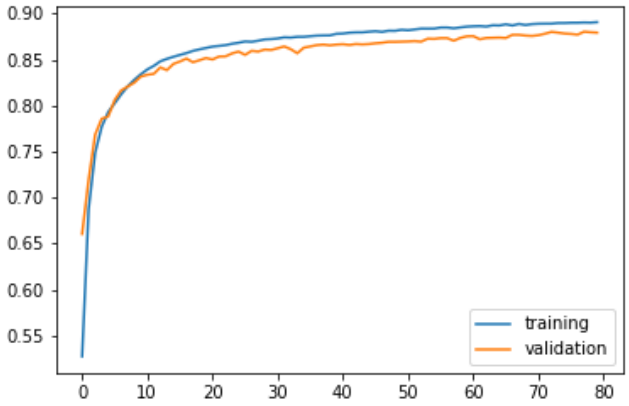

Figure 21: Accuracy curve for optimized MLP.

C. Comparison

In terms of accuracy, we see the following: our CNN classifier from Project 1 obtained an accuracy of 95.3%, our MLP classifier taking our MSE autoencoder's latent space representation as input obtained an accuracy of 88.0%, and our MLP classifier taking our ITL-regularized autoencoder's latent space representation as input obtained an accuracy of 78.1%. We note that the bottlenecking of our autoencoder appears to not be able to capture the same informative features as the convolutional process of a CNN based on the lower accuracy that representation is able to obtain. However, making

use of ITL, we can guide the encoder to create a more organized latent representation which may be able to sufficiently represent the salient features of the data in a more condensed latent representation than one without ITL guidance. Figure 22 presents the confusion matrices for all three models.

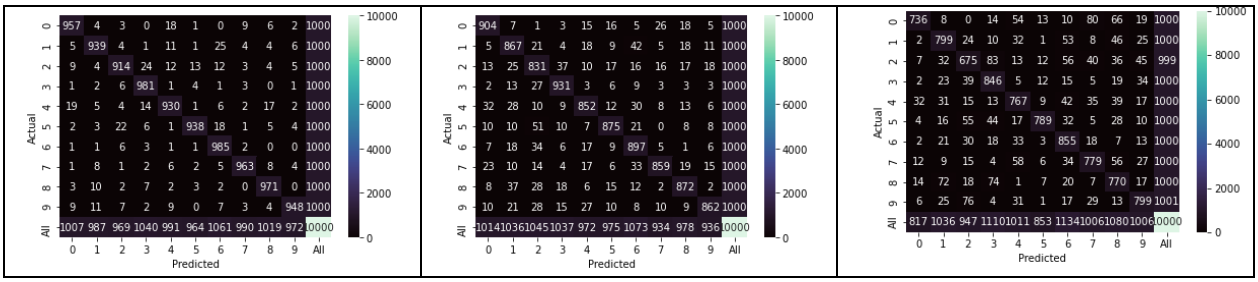

Figure 22: Confusion matrices for the CNN classifier (left), the MLP classifier taking MSE autoencoder's latent space representation as input (middle), and the MLP classifier taking ITL-regularized autoencoder's latent space representation as input (right).

In terms of computational time, we see the following: our CNN classifier from Project 1 trained in 886 seconds, our MLP classifier taking our MSE autoencoder's latent space representation as input trained in 105 seconds, and our MLP classifier taking our ITL-regularized autoencoder's latent space representation as input trained in 632 seconds. A single 2.6 GHz 6-Core Intel Core i7 processor was used for the entirety of this work, which may have impacted the training time.

In terms of number of parameters, we see the following: our CNN classifier from Project 1 required 557,322 parameters to be trained, our MLP classifier taking our MSE autoencoder's latent space representation as input required 2,125,019 parameters to be trained, and our MLP classifier taking our ITL-regularized autoencoder's latent space representation as input required 1,936,755 parameters to be trained.

D. Discussion

The choice of a two dimensional prior for the swiss roll worked reasonably well to separate the latent space by class while following the swiss roll pattern. A three-dimensional swiss roll may have resulted in better results than its two-dimensional counterpart after reconstruction for the additional latent dimension likely would have offered additional resolution in the latent space that could later be decoded into the output. Further experiments could be done to evaluate the advantages of such a change.

The ten-dimensional one hot prior constructed from a uniform random integer distribution matching that of the training labels did not work especially well as a prior distribution in order to classify the dataset in the latent space. Ideally, we would have like to use a one hot representation of the actual training labels, but we interpreted this to be not allowable by the assignment. The performance of the implemented one hot prior could likely have been improved by adding gaussian noise to the prior distribution.

In general, we expected a decrease in classification performance because of the relatively small latent dimension size allowable by the prior distributions chosen compared to the much larger latent dimension size selected in part A with the autoencoder. This shortcoming may have been overcome through experimentation with a deeper auto-encoder or manipulating the layer size.

Both parts A and B could likely have been improved through the use of synthetically augmented data to increase the number of training examples through rotation, shifting, and/or noise addition.

IV. Conclusion

In this project, we looked at the classification of the Kuzushiji-MNIST dataset using three different models – a convolutional neural network, an autoencoder, and an autoencoder with an additional ITL-regularization. We found that our CNN model was able to obtain the highest accuracy, due to convolutions ability to effectively capture feature information, and our ITL-regularized autoencoder to obtain the lowest accuracy, due to the limited number of latent space dimensions which were insufficient to adequately capture the underlying differences between classes. Unlike the MNIST dataset which has relatively simple symbols, the KMNIST dataset contained more complex patterns which were difficult to summarize in a small latent representation.

V. Division of Labor

Zachary and Matthew equally participated in the experimental design. Zach performed the coding for Part A and the MLP for Part B, while Matthew performed the coding for part A. Zachary performed the necessary experiments for Part A, along with Part 1 of Part B and the MLP for Part B. Matthew performed the experiments for the remainder of Part B.

VI. References

[1] Santana, Eder & Emigh, Matthew & Principe, Jose. (2016). Information Theoretic-Learning Auto-Encoder.
[2] https://github.com/EderSantana/seya/blob/master/seya/layers/regularization.py
[3] https://github.com/hwalsuklee/tensorflow-mnist-AAE