Handwritten Digit Recognition

Machine Learning Project

Dhruvs Sharma (s3812596)
Kyriakos Antoniou (s5715881)
Narjes Sharafi (s5697832)
Sumanth Seeram (s5652588)

Faculty of Science and Engineering
University of Groningen
The Netherlands

February 2024

Abstract

In this report, for "Handwritten Digit Recognition" three classifiers are presented and evaluated. These include linear regression, logistic regression, and a Convolutional Neural Network. Feature engineering and PCA are employed on the baseline linear regression model and the change in performance is analyzed. We find that the overall accuracy of the enhanced linear regression model improved significantly when compared to the baseline, being slightly above 52%, hence quite poor, but much better than the 20% of the baseline. The unsatisfying results lead us to employ other classification methods. First, we employ logistic regression, a method more suited out-of-the-box method for classification tasks, which performs well as expected (96% accuracy). Lastly, we also employ a baseline Convolutional Neural Network (CNN) to the task and compare the results with the enhanced version of it (with a more complex architecture, early stopping in place, and tuned hyperparameters. Here, we also tried L1 and L2 regularization, and found the enhanced CNN with L2 performs the best (96.18%)).

Contents

1 Introduction

The field of Computer Vision has seen much interest over the years, with one of the most fundamental tasks being digit classification. Handwritten digit recognition is a system that is trained to recognize and classify handwritten or printed digits from images. A typical pipeline involves pattern recognition and classification and can involve many avenues of machine learning to do so. When it comes to datasets employed for the digit classification task, the MNIST database has been widely used as a benchmark since 1998 [1]. It consists of 70,000 grayscale images in a 28x28 bounding box. A similar, easier-to-work-with toy dataset that we are going to use here is the "digits" benchmark dataset first devised by [2]. We use this dataset to train and evaluate two classical machine learning approaches, linear regression and logistic regression. Lastly, we dive into the world of Deep Learning to do the same classification task but now with an end-to-end Deep Learning pipeline involving a lightweight Convolutional Neural Network.

2 Data

The dataset we employed for this project, as mentioned earlier is the "digits" dataset devised by [2]. It contains 2000 grayscale images (size 16x15), of digits from 0-9, with 200 samples per digit. These grayscale images are arranged in $n = 240$-dimensional vectors, an example can be seen in Figure 1. The grayscale encoding here is done from 0 (black) to 6 (white) as this was the case in the provided `mfeat-pix.txt` file. We split the dataset into two equal parts: half the samples for training and validation (to tune hyperparameters, like max_iter for logistic regression) and the remaining samples for testing the best model we found.

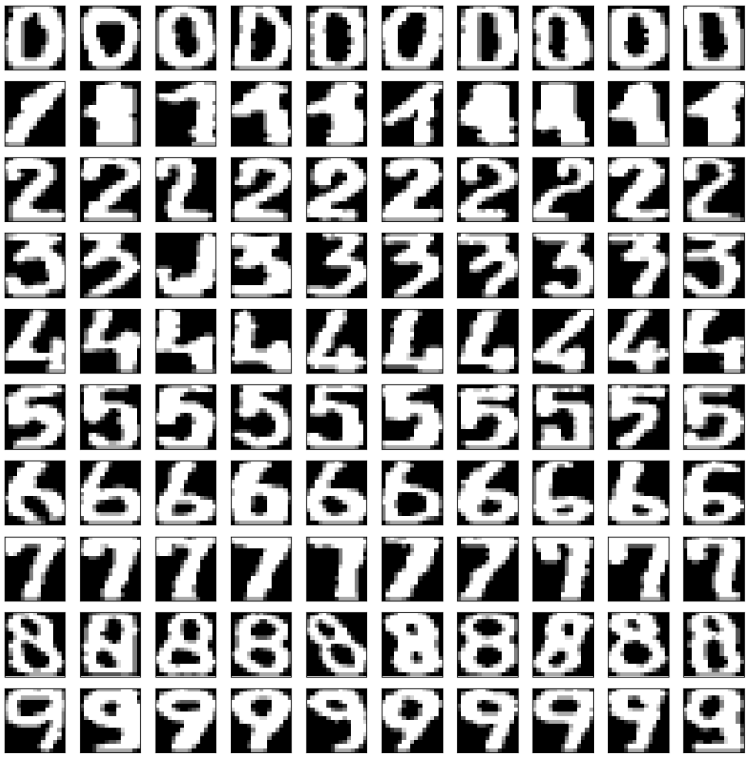


Figure 1: Some samples from the dataset from each digit class.

3 Methods and Experiments

In the report at hand, we first devised a baseline linear regressor. Next, we improved upon the performance of the baseline by incorporating feature engineering and PCA. Then, the performance of the two was compared.
On the other hand, a logistic regression model was also trained.
Lastly, a Convolutional Neural Network (CNN) was also trained with data augmentation, hyperparameter tuning, and regularisation techniques to prevent overfitting, and its performance was compared to the baseline CNN. In the subsections below, we shall dive into the methodological details of the techniques employed, including the implementation details.

3.1 Data Augmentation

For the training set at hand, 200 samples per class is quite a small number. This can of course lead to the risk of overfitting [3], which can be combated via the addition of samples for training. Hence, two data augmentation techniques were considered here, namely image rotations and scaling out. When it came to image rotations, we stuck to a random angle between -10 to +10 degrees per sample. The range was not set to a higher angle as it would lead to a more drastic change, leading to the rotated digit image looking quite different from the original, often involving artifacts. In Figure 2 such an example is shown with image rotation of 10 and 20 degrees. In terms of implementation, the well-known `skimage.transform` python library was used here to do image rotations, the `rotate()` method in particular.

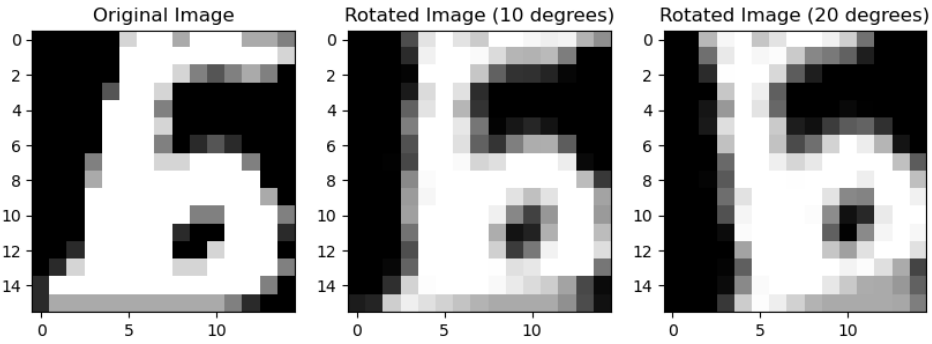


Figure 2: The digit "6" rotated by 10 and 20 degrees. Notice how the top part gets cut off and there are artifacts at the bottom, both of which increase with the angle of rotation.

On the other hand, for scaling out, we used the `rescale()` method (with a factor of 0.9) from `skimage.transform` python library, after which the scaled out image was padded back to the original 16x15 dimension. An example is shown in Figure 3. Notice how the edges get less sharp, and this happens more as the scaling factor is lowered, leading to a more loss of information. Hence, we stuck to a value of 0.9, with the idea of leaving some room for the image rotations, to strike a balance between information loss and room for rotation.

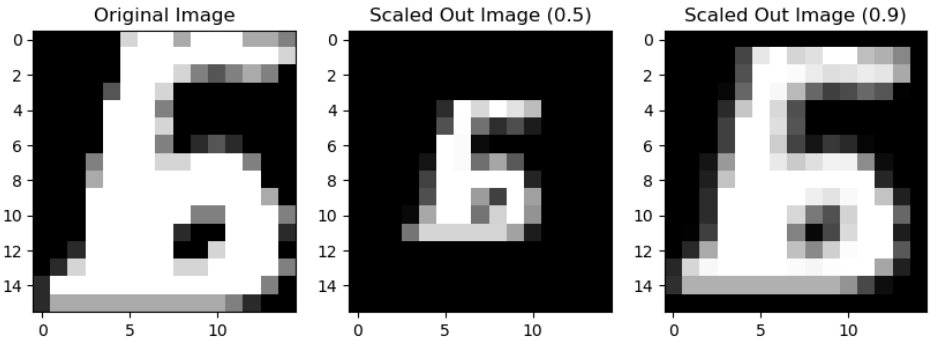


Figure 3: The digit "6" but now scaled out by a factor of 0.5 and 0.9. This means that the size of the scaled-out image is 0.5 and 0.9 times the size of the original, respectively .

When combined, both techniques produce a more robust result, with fewer artifacts, as seen in Figure 4. Using the combination of both data augmentation techniques, we get an image that resembles the original more but is also rotated at the same time. Knowing this combination, each sample in the training set was scaled out by a factor of 0.9 and randomly rotated between -10 to +10 degrees, effectively doubling the number of samples available for training. We applied data augmentation to the CNN pipeline. Hence 200 samples per class to train on, given we split the original 2000 sample dataset into two random equal halves first, and then double the 1000 training samples to 2000 via the addition of one augmented image per sample, while the test set still has 1000 samples from the original dataset.

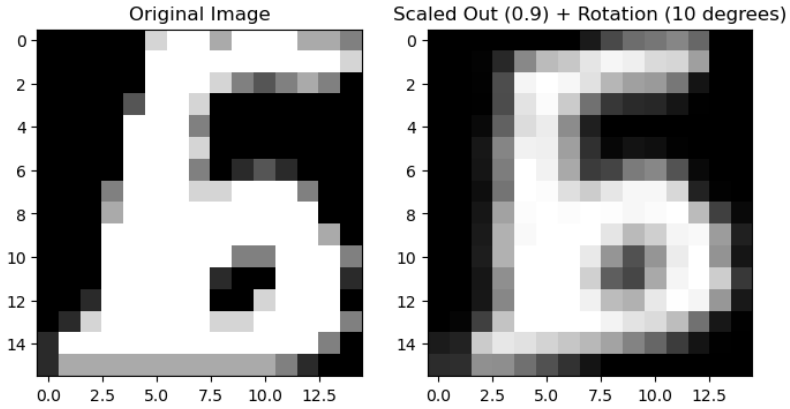


Figure 4: The digit "6" but now scaled out by a factor of 0.9 as well as rotated by 10 degrees. Notice how this is a much cleaner image compared to Figure 2.

3.2 Feature Engineering

Feature engineering refers to the process of creating new features from existing features. This process is selected based on the dataset and data types. for this dataset, we employ three feature engineering algorithms: Polynomial Features, Histograms of Oriented Gradients (HOG), and a Mixture of Gaussian (MoG). Feature engineering aims to add broader context to the available observation. In order to capture the non-linear characteristics in hand-digit data we perform the following methods.

3.2.1 Polynomial Features

Polynomial features are a type of feature engineering. Polynomial features are additional features created by raising the existing features to various powers. This process can be repeated for each input

variable in the dataset, creating a transformed version of each. The number of features added is based on the "degree" of the polynomial and typically small degrees are used such as 2 or 3 [4], as degrees higher than that lead to feature explosion. This results in a change in the probability distribution and separating the small and large values. This may help in prediction for some machine learning algorithms, typically for regression predictive modeling tasks. Therefore, this feature engineering method is done to improve the linear regression [5]. They can be used to capture nonlinear relationships between the predictor variables and the response variable.

The features created are:

- the bias (the value of 1.0)
- Values raised to power for each degree
- Interactions between all pairs of the features

Implementation details: To apply polynomial features, we use `PolynomialFeatures` function from `sklearn.preprocessing`. In order to prevent complexity and due to computational complexity we set `degree=2`.

3.2.2 Histogram of Oriented Gradients

Histogram of Oriented Gradients (HOG) is a feature descriptor technique commonly used in image classification tasks. The technique counts occurrences of gradient orientation in the localized portion of an image. HOG works by capturing the local distribution of gradients in an image to represent the structure of the image [6]. Therefore, this method is helpful for digit recognition with varying writing styles.

The process of HOG is:

1. The image gradients (G_x and G_y) using filters like Sobel or Scharr is computed.
2. The magnitude M and the orientation θ of gradients for each pixel is calculated.
3. The image is divided into cells and histograms of gradient orientations are accumulated within each cell.
4. Histograms of neighboring cells are concatenated to create block-level feature vectors and then they are normalized to enhance robustness.
5. Normalized block-level vectors are concatenated to obtain the feature vector for the image

Implementation details: To apply a histogram of Oriented Graients, we use `hog` function from `skimage.feature`.

3.2.3 Mixture of Gaussians

A Mixture of Gaussian (MoG) is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. The mixture of Gaussian implements the expectation maximization (EM) algorithm for fitting the model. [7]

The steps of a mixture of Gaussian process are:

1. Probability Density Function (PDF) of MoG with K components is calculated by:

$$P(x) = \pi \sum_{i=1}^{K} \cdot N(X|\mu_i, \sum_i) \tag{1}$$

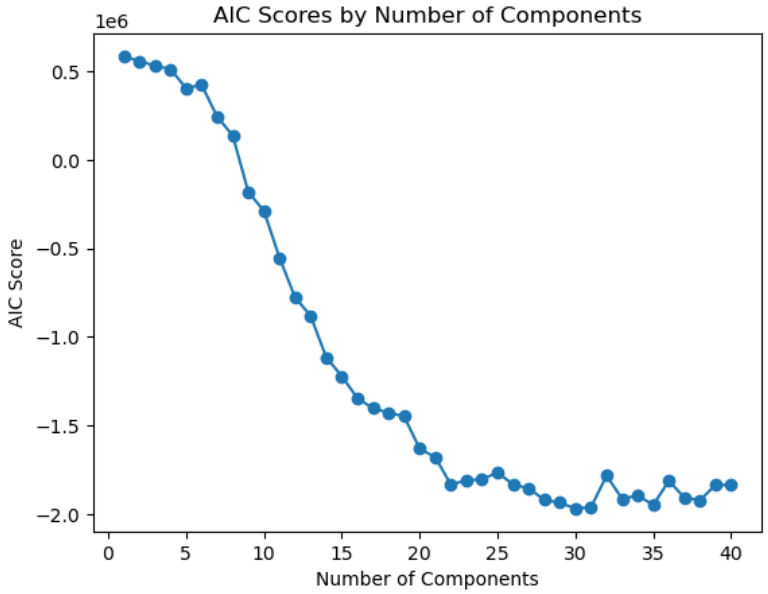


Figure 5: AIC estimation for different mixture components

where,

- x is the input
- π_i is the weight of the i-th component that represents the proportion of data belonging to that component.
- $N(X|\mu_i, \sum_i)$ is the Gaussian distribution with mean μ_i and covariance matrix $\sum_i$

2. Expectation-Maximization (EM) is calculated to update the parameters $\pi_i, \mu_i, \sum_i$ iteratively maximizing the likelihood of the observed data:

   - Expectation Step:
   $$w_i j = \frac{\pi_i \cdot N(X|\mu_i, \sum_i)}{\sum_{k=1}^{K} \pi_i \cdot N(X|\mu_i, \sum_i)} \tag{2}$$
   where, $w_i j$ is the probability that x_i belongs to the j-th component
   - Maximization Step: pi_i, μ_i, and $\sum_i$ is updated based on the weighted data.

3. The trained MoG assigns each data point to the cluster with the highest probability.

The optimal number of Gaussian components is calculated by the Akaike Information Criterion (AIC) [8]. The AIC is calculated by:

$$AIC = 2k - 2ln(L) \tag{3}$$

where

- k is the number of parameters in the model which is influenced by the number of components.
- L is the maximum likelihood of likelihood function for the model

The lowest value for AIC demonstrates the model that best fits the data. In Figure 5 the optimal for this dataset is 30.

Implementation details: To apply a mixture of Gaussian, we use `mixture` function from `sklearn.GaussianMixture(n_component=30)`.

3.3 Dimensionality Reduction

In this subsection, we shall explore the world of dimensionality reduction, particularly PCA and t-SNE and how are useful in the context of our project.

3.3.1 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a fundamental statistical procedure for dimensionality reduction in the field of machine learning and data analysis, often used for datasets with multiple variables. Its usefulness lies in its capability to transform the data into a new coordinate system where the axes, known as principal components, are aligned with the maximum variance. This allows for the retention of significant structural aspects of the data while reducing its complexity.

PCA can be done through following the steps below:

1. Firstly, standardize the dataset if necessary.
2. Next, compute the covariance matrix of the data.
3. After which, obtain the eigenvalues and the corresponding eigenvectors of this covariance matrix.
4. Then, sort the eigenvectors by decreasing eigenvalues and choose a subset of eigenvectors as principal components.
5. And lastly, project the original dataset onto the space spanned by the selected principal components.

Mathematically, PCA finds a matrix $\mathbf{W}$ that linearly transforms the data $\mathbf{X}$ to a new space such that the variance of the projected data is maximized along orthogonal axes. The columns of $\mathbf{W}$, the principal components, are the eigenvectors of the covariance matrix $\mathbf{C}$ of $\mathbf{X}$, ordered by their corresponding eigenvalues in descending order. The transformation is given by:

$$\mathbf{Y} = \mathbf{W}^T\mathbf{X} \tag{4}$$

where $\mathbf{Y}$ represents the data in the new coordinate system. The principal components $\mathbf{W}$ are chosen to maximize the variance of $\mathbf{Y}$ under the constraint that $\mathbf{W}$ is orthogonal:

$$\mathbf{W} = \underset{\mathbf{W}}{\operatorname{argmax}} \ \mathrm{Var}(\mathbf{W}^T\mathbf{X}) \quad \text{subject to} \quad \mathbf{W}^T\mathbf{W} = \mathbf{I} \tag{5}$$

In this context, $\mathrm{Var}(\mathbf{W}^T\mathbf{X})$ refers to the total variance captured by the principal components, and $\mathbf{I}$ is the identity matrix, imposing the constraint of orthogonality.

For the digits dataset, applying PCA before classification models like linear or logistic regression can significantly reduce the number of features, thus simplifying the model and making it faster. PCA can also uncover the most influential features that contribute to the variance in the dataset, which might correspond to critical distinguishing features of the digits. We believe there is a chance of a numerical explosion in the features when adding Polynomial features of higher degrees (>2) to the dataset for training the regression models. PCA could be quite helpful in this situation.

A small note to keep track of here is that the choice of the number of principal components m retains a fraction of the total variance. This is determined by the eigenvalues, which represent the distribution of the total variance across the principal components.

Implementation details: To perform PCA, we used the `PCA()` method from `sklearn.decomposition` library. To reduce the dimensions after applying Polynomial features, we used the variance based criterion to retain the components which represents 0.99 proportion of variance.

3.3.2 t-Distributed Stochastic Neighbor Embedding (t-SNE)

t-Distributed Stochastic Neighbor Embedding (t-SNE) is a stochastic machine learning algorithm for data visualization that has gained popularity in the analysis of high-dimensional datasets, such as the MNIST digits dataset used for multi-class classification. Unlike PCA which is a linear technique, t-SNE is a non-linear technique particularly well-suited for embedding data into a two- or three-dimensional space for visualization [9].

The core idea behind t-SNE is to maintain the local structure of the data in the reduced dimensionality. It starts by converting the high-dimensional Euclidean distances into conditional probabilities that represent similarities. The optimization objective is to minimize the Kullback-Leibler (KL) divergence between the joint probabilities in the high-dimensional and low-dimensional spaces.

$$C = KL(P||Q) = \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}} \tag{6}$$

Here, p_{ij} are the probabilities in the high-dimensional space, and q_{ij} are the probabilities in the low-dimensional space.

Minimizing this divergence via gradient descent leads to a low-dimensional representation of the dataset where similar instances are modeled by nearby points, and dissimilar instances are modeled by distant points. Given the local structure preservation and the ability to capture non-linear relationships, t-SNE might be more beneficial than PCA when visualizing datasets with complex manifolds such as images of handwritten digits. But as with everything, t-SNE has a drawback, and that is it being computationally intensive and not suitable for use as an input to other machine learning models, simply because the reduced output by t-SNE lacks any meaningful representation in terms of the original features, not to mention that it is a non-parametric approach, and stochastic.

Implementation details: For t-SNE visualization, the `TSNE(n_components=3)` method was utilized from the `sklearn.manifold` library. Figure 6 shows the visualization of the dataset at hand.

3.4 Cross-Validation

In some instances throughout our journey, we felt the need for a robust way to search for model hyperparameters. This was greatly aided by **k-fold cross-validation**, which is a method used to evaluate the performance of a model by dividing the data into k subsets [10]. The procedure is summarized as follows:

1. Partition the dataset D into k equally sized folds.
2. For each fold D_i where $i = 1, 2, ..., k$:
   (a) Train the model on $D \setminus D_i$.
   (b) Validate the model on D_i.
   (c) Record the performance metric M_i.
3. Compute the average performance $\bar{M}$ across all k folds:

$$\bar{M} = \frac{1}{k} \sum_{i=1}^{k} M_i \tag{7}$$

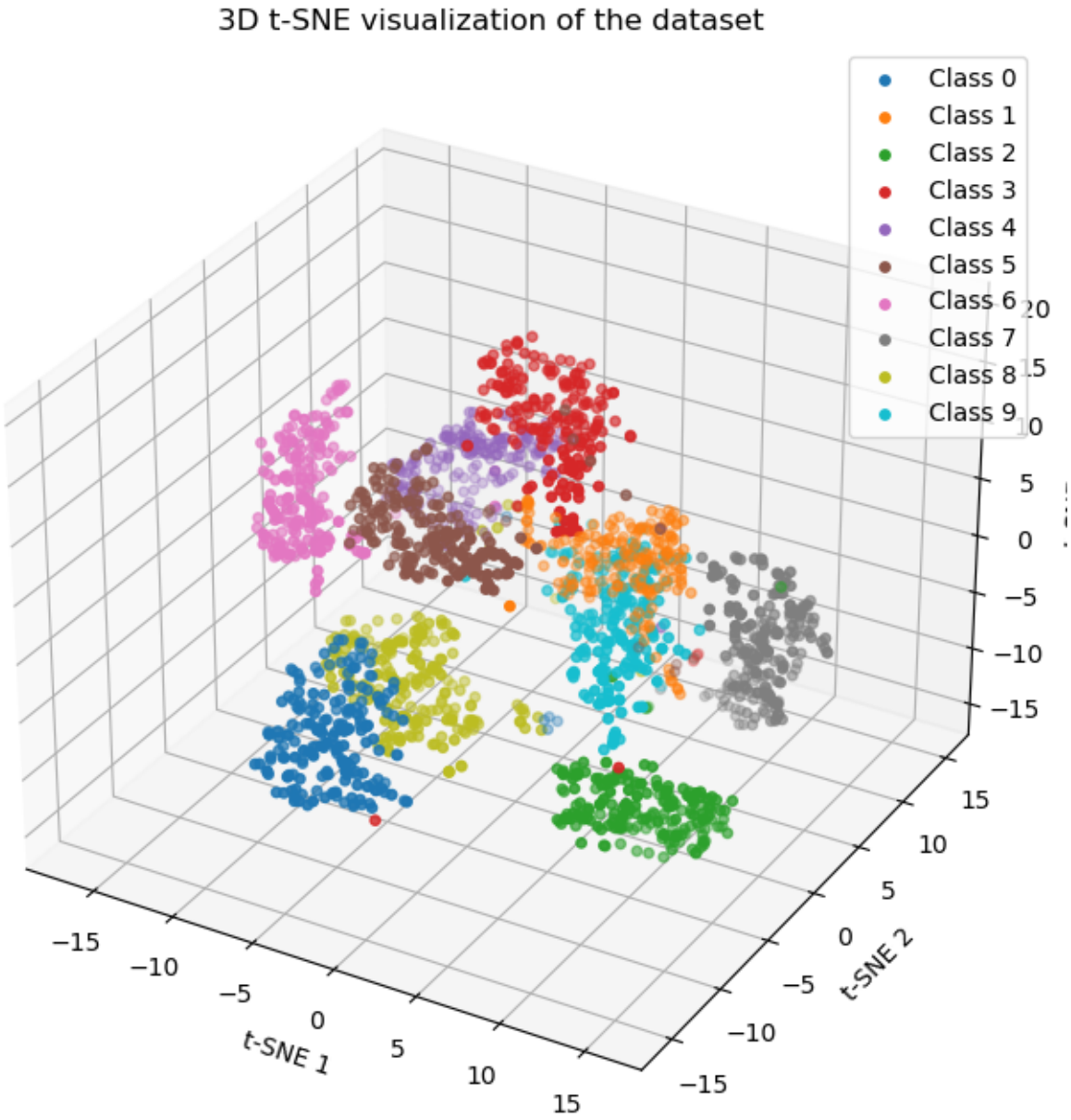


Figure 6: 3-D t-SNE plot for the 10-digit classes in the digits dataset. It displays the feature space segregation of the digits into ten distinct classes whereby each one has a unique color. The clusters that one can observe appear to be insightful in how the algorithm groups similar digit instances. One can see clear separations in some regions and overlaps in others. It is important to note, however, that some data points are visible within the bounds of dissimilar digit clusters, suggesting instances of feature overlap that could challenge the classifier's accuracy. The insights gained from

The idea here is a more robust estimation of the model's ability to generalize to unseen data compared to a single train-test split. The average of $\bar{M}$'s variation with the model hyperparameters can then be studied. Generally, when it comes to what value of k to choose, it links to the bias-variance dilemma, as a higher value of k would lead to less bias but more variance in the model's performance.

While k-fold cross-validation evaluates model performance, grid search is utilized to find the best set of hyperparameters for a model. It involves:

1. Defining a grid with a range of values for each hyperparameter.
2. Using k-fold cross-validation to evaluate every combination of hyperparameters in the grid.
3. Selecting the combination that yields the best average performance across folds.

Implementation Details: `GridSearchCV()` in `sklearn.model_selection` automates the above process to balance model complexity with predictive power. It was used to optimize the max_iter hyperparameter for logistic regression.

3.5 Linear Regression

As mentioned earlier, linear regression was employed to establish a baseline for comparing model errors by fitting a model to both pixel values of the training set and the feature vector of extracted features. The objective of linear regression is to find a linear relationship that maps input variables to an output variable [11]. Mathematically, this relationship can be expressed as:

$$(w, b) = \underset{w^*, b^*}{\operatorname{argmin}} \sum_{i=1}^{N} (w^* \cdot x_i + b^* - y_i)^2 \tag{8}$$

Here, w denotes the linear map, b is the bias term, and w^* and b^* are the estimated values for the regression weight vector and bias, respectively. The input vectors x_i and the output values y_i correspond to the features and target values. For this case, the input vectors can be either pixel values or feature vectors, and the output values are the classes the samples belong to.

Linear regression is anticipated to perform sub-optimally on raw pixel values due to the improbable linear relationship between pixel values and the class labels. However, by adding non-linearity to the input features (e.g. via Polynomial features, or extracting features like the average of pixel row values), the performance of linear regression is expected to improve.

Implementation details: To perform linear regression on the input features (both raw and transformed), we used the `LinearRegression()` method from `sklearn.linear_model` library. In the enhanced model pipeline, we first scale the features using `StandardScaler()` from `sklearn.preprocessing`, add polynomial features, and then perform PCA. After that, we fit the MoG model to the features and concatenate them with the HOG features.

3.6 Logistic Regression

We also applied logistic regression on the same task, to get a baseline. Unlike its binary classification version, logistic regression can be extended to multi-class problems by employing a one-vs-rest (OvR) or multinomial approach, which considers the probability that a given input corresponds to each of the possible categories [12].

The multi-class logistic regression model for the digits dataset, where the classes C represent the digits 0 through 9, can be formulated as:

$$P(Y = c) = \frac{\exp\left(\beta_{c0} + \beta_{c1} X_1 + \beta_{c2} X_2 + \ldots + \beta_{cn} X_n\right)}{\sum_{j=0}^{9} \exp\left(\beta_{j0} + \beta_{j1} X_1 + \beta_{j2} X_2 + \ldots + \beta_{jn} X_n\right)} \tag{9}$$

where:

- $P(Y = c)$ is the probability that the dependent variable Y corresponds to class c.
- exp(x) denotes e^x, which is the exponential function.
- $\beta_{c0}, \beta_{c1}, \ldots, \beta_{cn}$ are the coefficients for class c that define the relationship between the independent variables $X_1, X_2, \ldots, X_n$ and the log-odds of the dependent variable being in class c.

In this model, $X_1, X_2, \ldots, X_n$ represent the pixel values of the digit images after being flattened into a vector form. The pixel values serve as independent variables that contribute to the classification decision.

When it comes to the optimization of the coefficients, it is achieved via algorithms like gradient descent, aiming to maximize the likelihood of the observed training data, which equates to minimizing the cross-entropy loss in the multi-class setting.

Post-training, the model predicts the class of a new observation by assigning it to the class with the highest estimated probability. Unlike binary logistic regression, the multi-class version does not use a single threshold to assign classes but rather compares the probabilities of all possible outcomes. Within the context of the task at hand, we expect logistic regression to perform much better than linear regression.

Implementation details: Logistic regression was performed on the input features via the use of `LogisticRegression(max_iter=300)` method from `sklearn.linear_model` library.

3.7 Convolutional Neural Network

Lastly, two Convolutional Neural Networks were crafted, one set up as a baseline and the second one to show the best performance achieved using a more robust architecture and training regime. The power of CNNs lies in exploiting information from grid structures such as images by breaking down the image into feature maps. They can learn these hierarchical features from data where in the initial layers, they capture low-level features like edges and textures, and in the deeper layers, they capture more complex and abstract features [13, 14]. In addition, CNNs, like many other Deep Learning pipelines operate in an end-to-end fashion. In most cases, the raw input (for example, raw pixel values in our case) can be directly fed as input to the model, without the need for pre-processing steps like feature extraction in the case of a linear regression model.

The convolution operation, which is central to a CNN, can be mathematically described as follows. For an input image I and a filter (kernel) K, the feature map F at location (x, y) is computed as:

$$F(x, y) = (I * K)(x, y) = \sum_m \sum_n I(x + m, y + n)K(m, n) \tag{10}$$

where the operation $(I * K)$ denotes the 2D convolution of I with K (Typically performed by `Conv2D()` layer in `Keras`).
$I(x+m, y+n)$ accesses the pixel (or feature map value) that is m units away in the horizontal direction and n units away in the vertical direction from the point (x, y) in the input image. The above notation is used to describe how the filter is applied across the image: for each position (x, y) on the image, the convolution operation computes a weighted sum of the input image values around (x, y), where the weights are given by the filter (or kernel) K, and the specific input values are determined by the offsets m and n within the filter's dimensions. It is apparent from above that $K(m, n)$ is the value of the kernel at position (m, n). The feature map F captures the response of the filter at every spatial position of the input image. This helps in encoding localized features.

The subsequent layers (usually pooling layers) reduce the spatial dimensions of the feature maps and hence decrease the number of parameters and computational complexity. The above allows the

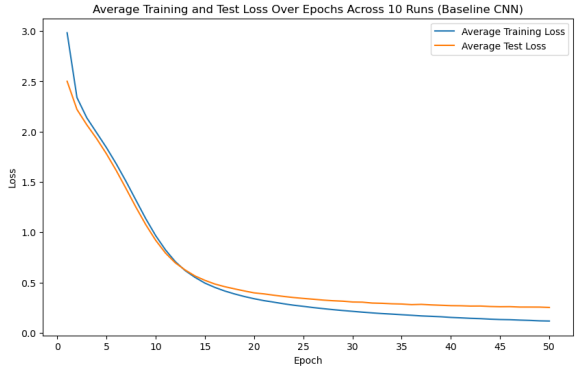
CNN to learn more global features of the input. For example, a max-pooling layer will downsample the feature map by taking the maximum value in a certain window, focusing on the most prominent features.

Lastly, fully connected layers integrate these learned local features to perform high-level reasoning (classification in our case). In these layers, neurons have full connectivity to all activations in the previous layer, as seen traditionally in neural networks. The combination of convolutional, pooling, and fully connected layers enables CNNs to transform raw pixel values into a complex hierarchy of features and classify them in an end-to-end fashion. [13, 14].

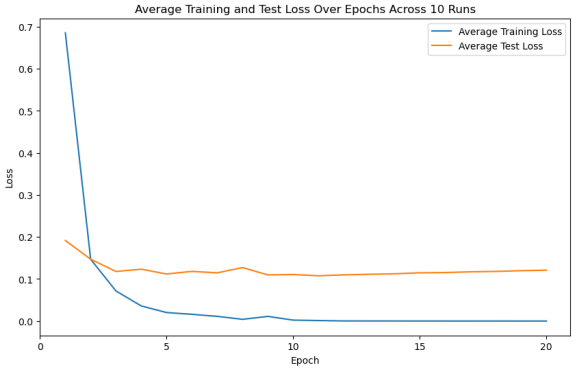
Implementation Details: `Tensorflow` was used here to build, train and evaluate our CNN.

3.7.1 Model Training

We begin by training each CNN model 10 times on the same data split (but dataset with augmented images for the enhanced CNN) and taking the average training and test errors over the 10 runs to compare. This step is mainly to show when the models start overfitting, and when early stopping can be put in place. For both the baseline and the improved CNN, the training loss and test loss were plotted in order to find the number of epochs to train the models for without overfitting them, which is where the test loss becomes flat or begins to increase. Figure 7a and 7b below show our training curves:



(a) Plot for the loss vs epoch for training loss and test loss on the baseline CNN. Both the curves start at a pretty high value, gradually going down as the epochs increase. The test loss seems to plateau at around 50 epochs but the training loss does not, suggesting the onset of overfitting as the model learns patterns specific to the training data that do not generalize to the validation data. The gap between the curves is also higher when compared to Figure 7b



(b) Plot for training loss and test loss on the improved CNN. Both training and validation loss decrease and stabilize, implying a good fit. The x-axis spans 20 epochs but it is apparent here as well that the model starts to overfit after epoch 5 as the test loss seems to stabilize, and hence early stopping was practiced here.

Figure 7: The two training curves.

3.7.2 Network Architecture

The architectures of the baseline CNN and the improved CNN are given in the Appendix in Figure 10. To keep it simple, we shall take a closer look at the architecture of only the enhanced neural network (Figure 10b), described in the following sequential manner:

1. `InputLayer`: The input layer (in our case, a part of `Conv2D()` as an argument) is designed to receive images of size 16×15. It does not perform any computation and serves as the entry point for the data.

2. `Conv2D(32, (3,3), activation='relu', padding='same', input_shape= (16, 5, 1))`: This is the first convolutional layer that applies 32 filters of size 2×2 to the input. The activation function used here is the ReLU (Rectified Linear Unit), defined as:

$$\text{ReLU}(x) = \max(0, x) \tag{11}$$

   The output of this layer is a feature map of dimensions $16 \times 15 \times 32$, indicating 32 feature maps for each filter applied.

3. `MaxPooling2D((2,2))`: The max pooling layer reduces the spatial dimensions of the feature map by applying a 2×2 pooling window, taking the maximum value within the window. The mathematical operation is:

$$S_{\text{maxpool}}(i, j) = \max_{m,n \in [2 \times 2]} F(m + i, n + j) \tag{12}$$

   This downsampling operation results in an output of size $8 \times 7 \times 32$.

4. `Conv2D(32, (3,3), activation='relu')`: The second convolutional layer applies 64 filters of size 2×2 using ReLU activation. It further processes the feature map to extract higher-level features and produces an output of dimensions $6 \times 5 \times 64$.

5. `MaxPooling2D((2,2))`: Another max pooling layer is used, further downsampling the feature maps to size $3 \times 2 \times 64$.

6. `Flatten()`: The flatten layer converts the 3D feature maps into a 1D feature vector, which is necessary before passing data to the fully connected layers. The output is a vector of length 384.

7. `Dense(64, activation='relu')`: The first dense (fully connected) layer has 64 neurons and uses ReLU activation. It integrates features from the flattened vector for higher-level reasoning.

8. `Dense(10, activation='softmax')`: The second dense layer is the output layer with 10 neurons corresponding to the 10 classes of our dataset. It uses the softmax activation function, which is defined for a vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ and for each element x_i as:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{13}$$

   The softmax function outputs a probability distribution over the 10 classes.

Compared to the baseline CNN, one change not apparent from the figures is the number of neurons used in layers 2, 4, and 7 above is eight times the number of neurons in the baseline, where we used 4, 8, and 8 neurons respectively. Other key changes include an increased number of filters, along with the preservation of spatial dimensions through padding. The architecture change alongside hyperparameter tuning, regularization and early stopping contribute the most towards an increase in performance as we shall mention in the results later on.

3.7.3 Hyperparameter search

The models were optimized using the Adam optimizer. Optimizers in neural networks are used in order to help the (stochastic) gradient descent at finding a minimum. Adam is part of the family of algorithms that try to find the best learning rate for the model depending on the shape of the error landscape, it adjusts the learning rates for each parameter individually, computing an adaptive learning rate for each parameter based on the historical gradients [15].

Adam combines the advantages of two other extensions of stochastic gradient descent. Specifically:

- Adaptive Gradient Algorithm (AdaGrad) maintains a per-parameter learning rate that improves performance on problems with sparse gradients.
- Root Mean Square Propagation (RMSProp) also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight (which works well in on-line and non-stationary settings).

The ADAM update rule is given by the following equations:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{14}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{15}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{16}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{17}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{18}$$

where:

- m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name ADAM, which stands for adaptive moment estimation.
- g_t is the gradient of the objective function with respect to the parameter θ at time step t.
- β_1 and β_2 are exponential decay rates for these moment estimates.
- $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected versions of m_t and v_t.
- α is the step size or the learning rate.
- ϵ is a small scalar used to prevent division by zero, usually around 10^{-8}.

Furthermore, an ablation test was conducted to find the best batch size for the models, which was found to be x for the baseline CNN and x for the improved CNN. $\alpha = 0.001$

Parameter	**Baseline CNN**	**Enhanced CNN**
Learning Rate	0.001	0.001
Epochs Trained	50	5
Batch Size	64	8

Table 1: Comparison of Baseline and Enhanced CNN hyperparameters. In both cases, ADAM is used with the default hyperparameter settings, hence an α of 0.001. The enhanced model starts overfitting early, hence we only train it for 10 epochs. We also found the optimum batch size to be 8.

3.7.4 Regularization

Regularization techniques are widely used for improving the generalization of CNNs, quite crucial when dealing with a dataset like ours where the model complexity can lead to overfitting. Two commonly used methods are described below [13]:

- **L1 Regularization (Lasso):** A penalty equal to the absolute value of the magnitude of the coefficients is added to the loss. This can lead to sparse models with some weights reduced to zero:

$$\Omega(\theta) = \lambda \sum_{i=1}^{n} |\theta_i|$$

  where θ represents the weights of the network, n is the number of weights, and λ is the regularization hyperparameter.

- **L2 Regularization (Ridge):** Adds a penalty equal to the square of the magnitude of the coefficients. This discourages large weights through a penalty on their squared values but does not necessarily eliminate them:

$$\Omega(\theta) = \lambda \sum_{i=1}^{n} \theta_i^2$$

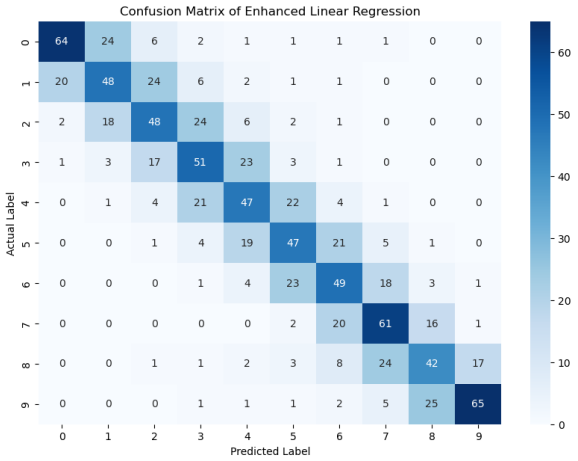
  L2 is less robust to outliers than L1 as it does not promote sparsity but is excellent for handling multicollinearity and model complexity.

Regularization terms are added to the loss function, and during training, they contribute to the overall loss by penalizing weight magnitudes. The hyperparameter λ controls the strength of the regularization; as higher values lead to more significant penalties on the weights. The choice between L1 and L2 regularization can be guided by the specific needs of the model and dataset.

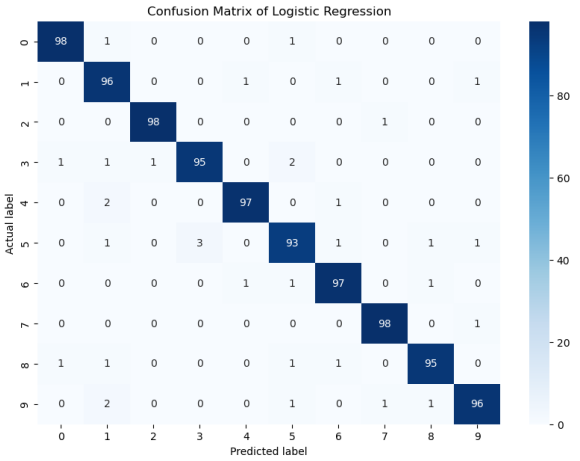
4 Results

In this section, the results of the performance of the three classifiers are presented. The confusion metrics in Figure 8 demonstrate the performance of enhanced linear regression and logistic regression. Table 4, Table 2, and Table 3 present the accuracy of these two classifiers in addition to the model trained by CNN. Table 2 presents the accuracy value for linear regression, an enhanced version of linear regression and logistic regression. All the results in this section are explained in the Discussion section.

4.1 Regression Models



(a) Confusion matrix of enhanced linear regression for digit classification, displaying true positive counts on the diagonal for each digit tested on 100 samples per class. The model shows high accuracy for digits "0", "9", and "7", as seen by the darker shades indicating higher true positive rates. In contrast, digit "8" shows the worst notable misclassifications, as seen by the lighter shade on its diagonal, implying the model's challenges with this particular digit.



(b) Confusion matrix of enhanced logistic regression (we test on 100 samples per class). The digits that got the lowest misclassification scores are "2", "0", and "7" with darker shades. However, the classifier performs poorly on digit "5", with lighter shades on the diagonal cell related to this digit.

Figure 8: The results of the two regression classifiers as confusion matrices. On the x-axis, we see the ground true label, while on the y-axis, the correct counts for each digit are displayed on the diagonals. The classification results are obtained by averaging the outputs from 50 different instances of the regression models (trained on a random train-test (50-50) split each time) and then converted to integer values.

Model	Accuracy (%) ± Std. Dev.
Baseline Linear Regression	20.45 ± 1.11
Enhanced Linear Regression	52.14 ± 4.42
Logistic Regression	96.2 ± 0.39

Table 2: Model Accuracies with Standard Deviations. Here too, the models were trained 50 times and their test results averaged over 50 times. Hence the standard deviation.

4.2 CNN

Model	**Accuracy** (%)
Enhanced CNN	94.70%
Baseline CNN	92.10%

Table 3: Model accuracies on the test set, similar to before (as in the regression models) averaged over outputs from 50 trained instances of the enhanced model with shuffled train-test splits. Here, no regularization is done.

Regularizer	**Accuracy** (%)
L1	95.82%
L2	96.18%
L1L2	95.63%

Table 4: Accuracies on enhanced CNN tested with L1 and L2 regularization, obtained through averaging the outputs from 50 different instances of the enhanced CNN (each model instance trained on a random 50-50 train-test split). The regularisation amount for all three types was 0.001 and was added to the two Convolutional layers of the model

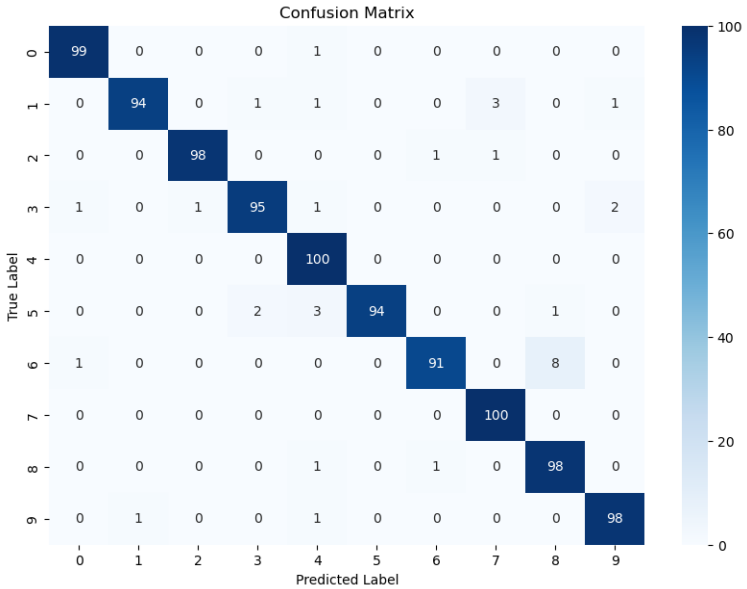


Figure 9: Confusion matrix for the enhanced CNN (the best performing CNN) with L2 regularization obtained through one forward pass of the model on the test set. True labels are on the x-axis and predicted labels are on the y-axis. The correct counts are on the diagonal cells, while misclassification counts are on off-diagonal cells. 100 samples are considered for the training set per group, and the same number for the testing set. The model performs best for digits "4" and "7" with no misclassifications. The worst performance is for classifying digit "6", as it is misclassified for digit "8".

5 Discussion

In this section, the results and the performance of each classifier are discussed and compared to each other.

5.1 Linear regression

The performance of the linear regression classifier on this digit dataset is presented in Table 2. The mean accuracy value over 50 different trained models of this classifier is 20.45%. In order to improve the performance, three different feature extraction methods are employed on linear regression. The performance of this enhanced linear regression model gets better to 52.14%, over 50 different trained models. The confusion matrix in Figure 8a presents the performance of the enhanced linear regression in classifying different digit classes. For each class, the number of samples in the training set is about 100 samples, varying in different iterations. For the testing set, the number of samples for each class is about 100 samples. Diagonal elements of the confusion matrix are the correct classification counts (true positives) for each digit, while the numbers of misclassifications are the numbers of off-diagonal elements. We notice that the distributions of this misclassification reveal the poor performance of linear regression, despite enhancing with feature extractions. This result was expected from linear regression due to the inherent non-linearity and the discrete nature of the dataset. One interesting thing to note here is how most misclassifications are happening in the top and bottom neighbors of the diagonal elements. We believe this could be a result of fitting the MoG model on the features.

5.2 Logistic Regression

Now let us discuss the results of the second classification method that is employed to classify this digit dataset, which is logistic regression. Table 2 shows the performance of this classifier is presented. One can observe that the mean value for accuracy over 50 various trained models for logistic regression is 96.2%. The confusion matrix presented in Figure 8b demonstrates the performance of logistic regression for classifying each digit. As mentioned in the caption, number of samples for the training set is about 100 samples per class, the same value for the testing set, varying in different training iterations. Large numbers of diagonal cells, which present the correct counts on classification, in contribution to the number of misclassifications on the off-diagonal cells, demonstrate the good performance of logistic regression. Digits "0" and "7" are classified the best with less than 3 misclassifications. However, the classifier performs worst for digit "5" with 6 misclassification counts, as it is misclassified as digit "3" more than other digits with 3 misclassification counts. The results reveal that logistic regression performs better on classifying handwritten digits as the output of classifying is discrete classes. Logistic regression models the probability of belonging to a certain class. This method with a nonlinear activation function like the sigmoid is better for capturing the non-linear relationships in the digit dataset.

5.3 CNN

Next, we discuss the CNN's performance. The baseline model performs quite well (92.10%) (Table 3) on the dataset out-of-the-box without any hyper-parameter tuning, data augmentation, regularization, or early stopping put in place. The model has a simpler architecture with a fewer number of neurons in the convolutional layers, on top of having one less convolutional layer to the enhanced model, which makes it converge slower. On the other hand, we see a step up in performance (94.7% average) with the enhanced model. The confusion matrix in Figure 9 supports this claim even better, where the classification results over one forward pass of the L2 regularized enhanced model are given. In this specific instance, digits "4" and "7" are the best classified by model with no misclassifications. On the other hand, digit "6" is misclassified the most, as it is misclassified as digit "8" for 8 counts. The

improvement in performance from the baseline here is supported by Table 4, where both ridge and lasso give a significant bump in model performance, via the penalization of the weights and making the model more sparse. One can observe that L2 gives the best average accuracy (96.18%), which aligns with what we expected since our model is quite complex compared to the dataset at hand and can easily overfit. Overall, we are pleased with the results but were expecting a more significant bump in performance from the baseline. Possibly, this bump seems to be smaller because we use shuffled train-test splits for each instance of the model trained and then test it on that specific test split (and then average the accuracies over each instance, so 50 in our case)

5.4 Future work and Conclusion

When it comes to the linear regression model, improvements could be made via the addition and careful integration of features from other feature extraction methods, like islands and Fourier transform to name a couple.
Next, additional ways to improve the CNN model would be to create an ensemble of CNN models with varying kernel sizes in order to allow the model to learn features common to different situations and so to generalize better. Other avenues that could be looked upon include adding uncertainty quantification to the CNN (MC-Dropout, MC-Dropconnect, Flipout, etc, see `https://github.com/mvaldenegro/keras-uncertainty` for more details). The data augmentation could also use some morphological operations, to make the scaled-out image look sharper for example. Lastly, we desperately felt the need for a more streamlined way of testing for accuracy, as the shuffled splits might not be the most optimal way to do so. We would like to end by concluding that the best classifier found was either one of the two models: Logistic regression or Enhanced CNN with L2 regularization, with very close accuracies output by both models.

References

[1] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[2] Richard Durbin et al. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 2000.

[3] Connon Shorten and Taghi M. Khoshgoftaar. "A survey on Image Data Augmentation for Deep Learning". In: *Journal of Big Data* 6.1 (July 2019), p. 60. ISSN: 2196-1115. `https://doi.org/10.1186/s40537-019-0197-0`. `https://doi.org/10.1186/s40537-019-0197-0`.

[4] areth James et al. *An introduction to statistical learning*. Springer, 2013.

[5] Kjell Johnson Max Kuhn. *Feature Engineering and Selection*. Chapman and Hall/CRC, 2019.

[6] Navneet Dalal and Bill Triggs. "Histograms of oriented gradients for human detection". In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Vol. 1. IEEE. 2005, 886–893 vol. 1. `https://doi.org/10.1109/CVPR.2005.177`. `https://ieeexplore.ieee.org/document/1467360`.

[7] Chris Stauffer and W. Eric L. Grimson. "Adaptive background mixture models for real-time tracking". In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (1999), pp. 246–252. `https://doi.org/10.1109/CVPR.1999.784637`. `https://ieeexplore.ieee.org/document/784637`.

[8] Hirotugu Akaike. "A new look at the statistical model identification". In: *IEEE Transactions on Automatic Control* 19.6 (1974), pp. 716–723.

[9] Farzana Anowar, Samira Sadaoui, and Bassant Selim. "Conceptual and empirical comparison of dimensionality reduction algorithms (PCA, KPCA, LDA, MDS, SVD, LLE, ISOMAP, LE, ICA, t-SNE)". In: *Computer Science Review* 40 (2021), p. 100378.

[10] Tadayoshi Fushiki. "Estimation of prediction error by using K-fold cross-validation". In: *Statistics and Computing* 21 (2011), pp. 137–146.

[11] Dastan Maulud and Adnan M. Abdulazeez. "A Review on Linear Regression Comprehensive in Machine Learning". In: *Journal of Applied Science and Technology Trends* 1.4 (2020), pp. 140–147.

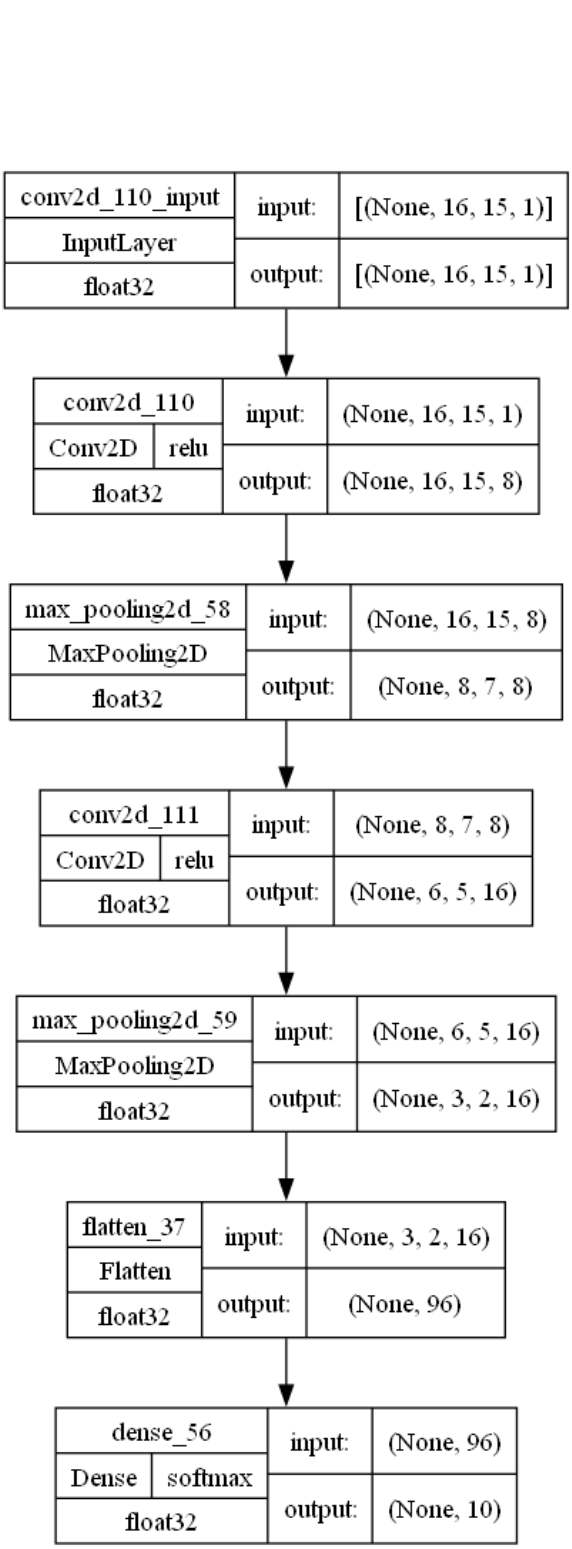
[12] David W. Hosmer, Stanley Lemeshow, and Rodney X. Sturdivant. "Applied Logistic Regression: Hosmer/Applied Logistic Regression". In: 2005. `https://api.semanticscholar.org/CorpusID:119351629`.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. ISBN: 978-0262035613. `http://www.deeplearningbook.org`.

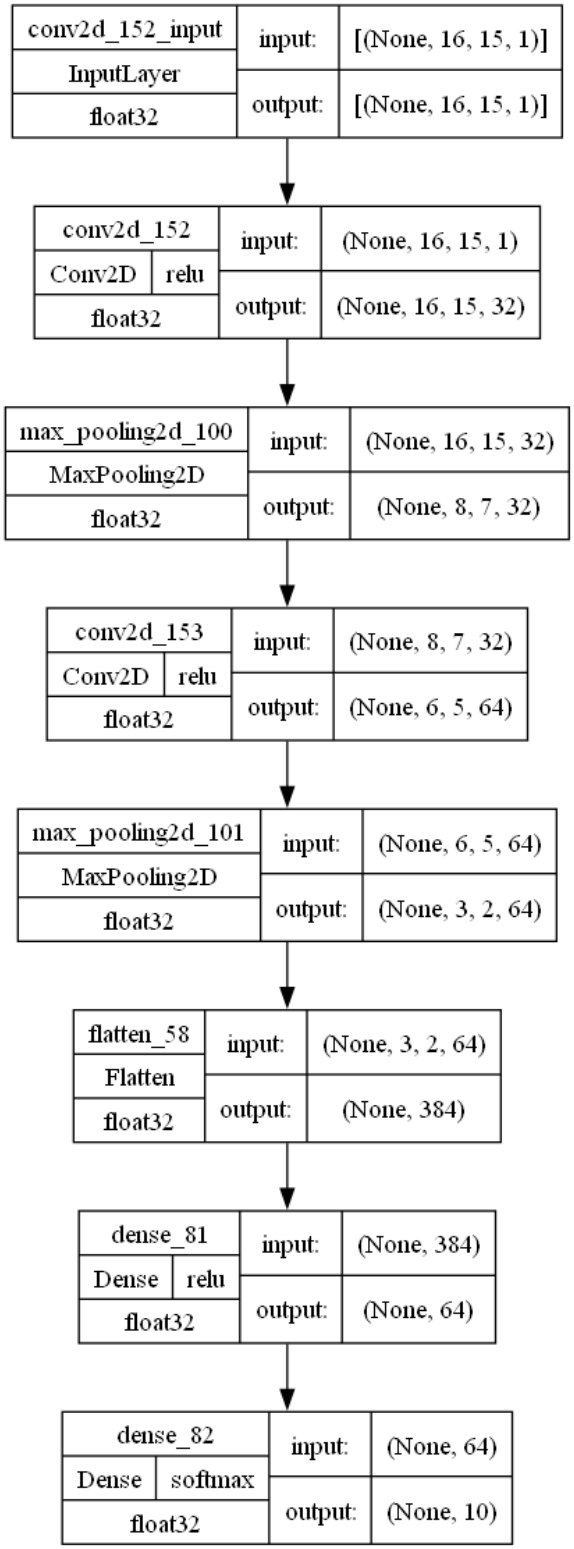
[14] Keiron O'Shea and Ryan Nash. "An Introduction to Convolutional Neural Networks". In: *arXiv preprint arXiv:1511.08458* (2015).

[15] Diederik P Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

6 Appendix



(a) Baseline CNN architecture



(b) Final CNN architecture

Figure 10: Our two versions of the CNN