Contents

Introduction

Artificial neural networks (ANNs) draw inspiration from early models of brain sensory processing. These networks are created by simulating interconnected model neurons in a computer. *Can hand-written symbols or shapes be recognized using machine learning neural networks, specifically pertaining to Greek symbols commonly used in engineering-focused mathematics?* The significance of this research lies in its potential applications, such as enhancing educational tools and automating grading systems.

Understanding how various neural network parameters impact learning efficiency and classification accuracy is crucial. This report delves into the so-called 'black box' of a neural network, examining the parameters and their effects on performance. To validate these concepts, data will be collected from hand-written symbols, testing the model against real-world data. It is essential to contextualize this study within the broader field of neural network applications and highlight the significance of recognizing Greek symbols, which have practical applications in fields such as educational tools and automated grading systems. A detailed discussion of neural networks, their origins, and their theoretical underpinnings is provided in the Design and Theory chapter.

Once the neural network was understood, it became important to review how others have approached and solved the task, as well as the methods they used. Figure 1 illustrates various methods, both machine learning-based and traditional.

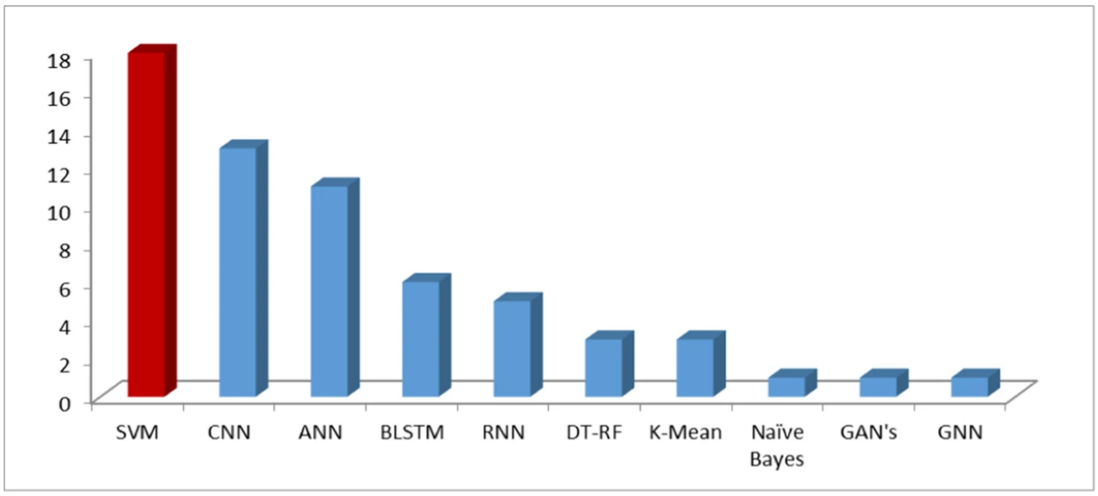


Figure 1: Machine learning and non-machine learning methods in mathematical recognition systems[1]

- Support Vector Machine (SVM)
- Artificial Neural Network (ANN)
- Convolutional Neural Network (CNN)
- Recurrent Neural Network (RNN)
- K-means (KNN - K nearest neighbours)
- Decision Tree and Random Forest (DT+RF)
- Generative Adversarial Networks (GANs)
- Graph Neural Network (GNN)
- Bidirectional Long Short-Term Memory (BLSTM)

Literature Review

Hand-drawn symbol and shape classification is an important area in computer vision and pattern recognition. The broad aim is to create a system capable of interpreting hand-drawn images and predicting the depicted symbol or shape. This task has applications in various fields such as document analysis, educational tools, and user interface design. This literature review focuses on the recent advancements and methodologies used in the classification of hand-drawn symbols and shapes.

Convolutional Neural Networks (CNNs) for Hand-drawn Symbol Recognition

Wang et al. propose a method for hand-drawn electronic component recognition using a Convolutional Neural Network (CNN) combined with a softmax classifier[2]. The CNN architecture includes a convolutional layer, an activation layer, and an average-pooling layer to extract features from hand-drawn images. The kernel function for the CNN is derived using a sparse auto-encoder method, achieving a recognition accuracy of 95% for electronic components such as capacitors, resistors, and diodes [2].

Hybrid CNN-RNN Networks for Handwriting Recognition

Dutta et al. present a modified CNN-RNN hybrid architecture for handwriting recognition[3]. Their approach includes efficient initialization using synthetic data for pre-training, image normalization for slant correction, and domain-specific data transformation and distortion techniques. This architecture has shown state-of-the-art results on popular datasets like IAM, RIMES, and GW [3].

Two-Stage CNNs for Symbol Recognition

A two-stage CNN approach for symbol recognition involves initial feature extraction followed by a classification stage. This method has been effective in various applications, including the recognition of handwritten mathematical symbols and other complex shapes. By employing a hierarchical structure, these networks can handle the variability in hand-drawn inputs and improve classification accuracy [4].

Recognition of Handwritten Mathematical Expressions

The recognition of handwritten mathematical expressions integrates several techniques, including CNNs for feature extraction and RNNs for sequence prediction. The challenge lies in accurately parsing the spatial arrangement of symbols to form valid mathematical expressions. Advanced models that combine these techniques have shown success in interpreting handwritten formulas with high precision [5].

Mathematical Formula Recognition

In addition to deep learning techniques, other methods have been explored for hand-drawn symbol and shape classification. Techniques such as minimum spanning trees and symbol dominance can be employed for symbol grouping and relationship identification, which is particularly useful in recognizing and classifying hand-drawn shapes and symbols[6]. Neural networks and graph-based approaches can also be used for symbol recognition and segmentation, allowing the system to handle the variability in hand-drawn inputs[6]. Incorporating contextual information and knowledge about the symbols and shapes being drawn can significantly improve the accuracy of the classification system. This can include rules and constraints based on the layout and structure of the symbols and shapes[6]. Additionally, machine learning and deep learning algorithms can be trained on datasets of hand-drawn images and their corresponding labels, enabling the system to learn patterns and features specific to hand-drawn symbols and shapes[6]. Pre-processing techniques such as image filtering, thresholding, and edge detection can be employed to enhance the quality of the input images and extract relevant features for classification. Furthermore, implementing mechanisms for error analysis and correction, such as confidence-based classification and post-processing steps, can help improve the overall accuracy and robustness of the system[6].

For further reading, [1] is highly recommended, as well as [7], which is an interesting and thorough review despite its relative compactness. Due to length constraints, this literature review does not cover ANNs in detail, as quality sources were difficult to find, especially pertaining to Greek symbols specifically (this research aims to fill this gap/niche). However, CNNs, being a specialized type of ANN, are ideal for this task although CNNs were not directly utilized in this report due to added complexity, while ANNs can still very effectively complete the task at hand.

Design and Theory

The brain's computations are performed by a complex network of neurons that communicate via electric pulses traveling through axons, synapses, and dendrites. In 1943, McCulloch and Pitts created a model of a neuron as a switch that receives inputs from other neurons and activates or remains inactive based on the total weighted input. The weight, reflecting the synaptic strength, determines the influence of an input and can be either positive (excitatory) or negative (inhibitory).[8]

Using algorithms that imitate neuronal processes, ANNs can learn to address various problems. Each model neuron, known as a threshold unit, receives inputs from other units or external sources, weights these inputs, and sums them. If the total input exceeds a certain threshold, the unit's output is one; otherwise, it is zero. This binary output results in the creation of a hyperplane, which in a two-dimensional space is a line and in a three-dimensional space is a plane. The hyperplane separates inputs into two classes, facilitating the solution of linearly separable classification problems.[8]

When tackling a separable classification problem, it's essential to set the weights and threshold so that the threshold unit correctly classifies the inputs. This is achieved iteratively through a process known as learning or training, where examples with known classifications are presented one by one. During training, the weights and threshold are adjusted incrementally to improve classification accuracy. [8]

Consider an example where an artificial neural network can classify cancer tumors based on gene expression by determining if they are responsive to treatment. While simple, linearly separable data can be handled by a single threshold unit, more complex, non-linear data requires multi-layer networks. These networks, called multi-layer perceptrons, use hidden layers of threshold units to handle intricate classification tasks. By employing functions like the sigmoid function (Although most modern networks use the ReLu function as it is considered to be more accurate), these networks can output probabilities for classification. Multi-layer networks are essential for problems like the XOR function, where each hidden layer performs partial classifications that are combined in the final layer to accurately classify complex data patterns. [8] The back-propagation algorithm trains feed-forward neural networks by adjusting weights to minimize error between predicted and actual outputs. Training starts with small random weights, and for each input example, the network's output is compared to the desired output. The total error is the sum of squared differences, which the algorithm reduces using gradient descent.

Weights and thresholds are updated iteratively to decrease the error. This process requires tuning parameters such as the learning rate and momentum. However, challenges include local minima and overfitting, where the network memorizes training data instead of generalizing [8].

Over-fitting occurs when too many parameters are learned from too few examples. For instance, a network with 10 hidden units and 20 input features has 221 parameters, excessive for 100 examples. Mitigation strategies include smaller networks, averaging results, regularization, and Bayesian methods[8].

Cross-validation helps estimate generalization performance. The dataset is split into subsets, with the network trained on some subsets and tested on others. Repeating this process ensures each subset is used for testing, providing an estimate of the network's ability to handle new data. Ensuring diverse examples in subsets is crucial for an unbiased estimate[8].

Final Design

The techniques mentioned above can be implemented in python using the tensor flow package. The following code shows the unique lines of code required to simply create a single hidden layer neural network that can take in a WidthxHeight pixel image, and output the desired amount of classes to fit and validate with the desired data.

```python
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Flatten(input_shape=(WIDTH,HEIGHT)))
model.add(tf.keras.layers.Dense(NEURONS, activation='relu'))
model.add(tf.keras.layers.Dense(CLASSES, activation='softmax'))
model.compile(optimizer='adam', loss= 'sparse_categorical_crossentropy', metrics=['accuracy'
    ↪ ])
model.fit(TRAINING_DATA, epochs=ITERATIONS, verbose=1)
model.evaluate(VALIDATION_DATA, batch_size=SIZE, verbose=1)
```

Methodology

Here the basic methodology and reasoning behind it is outlined to show the data collection and model development timeline for understanding of the experiment and repeatability.

1. **Proof of concept:** Before the model is to be created for interpretation of handwritten Greek symbols and real-world data collected. A model will be created on a massive standard database (in our case we used the MNIST database - but any large scale database can be used) to initially confirm the functionality of the code and concept of the theory involved.

2. **Control Model:** Once confirmed, a control will be created to test neural networks in identifying mathematical and Greek symbols. An online database from Kaggle will be used, that provides a bigger database than the one gathered. By creating a folder with the data separated into its respective classes, we can use the following code to import the data into TensorFlow and separate it into training and validation data.

```
con_train = tf.keras.preprocessing.image_dataset_from_directory('ControlSymbols',
↪     labels='inferred', label_mode = "int", color_mode='grayscale',
batch_size=bSizeC, image_size=(heightC, widthC), shuffle=True,    seed=SEED,
↪ validation_split=0.1, subset="training" )
```

   This will be used to confirm the functionality of classifying the data and can be used to compare it to the real-world data gathered. This allows us to draw accurate conclusions based on the models' performance and the data gathered. The parameters are to be experimented with in order to create a perfect control model and to decipher the most appropriate parameters to create the highest-performing data for the real-world model.

3. **Collection of real-world data:** Each of the 25 outstanding UCT students digitally transcribed a total of 100 pages (using canva), encompassing all five specified symbols - Pi, Theta, Alpha, Beta, and Infinity. This yielded a dataset comprising 2500 total symbols. It was decided that a bigger database of fewer classes would be more beneficial than vice versa.

(a) Control data: hand-written alpha

(b) Real-world data: hand-written alpha

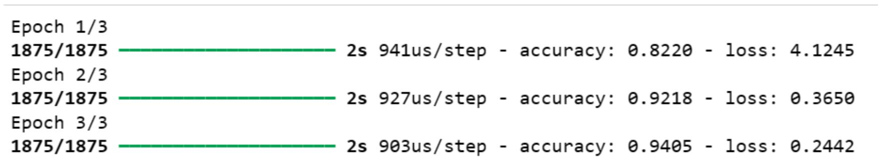
Figure 2: Example comparison of control data vs real-world data

4. **Real-world model:** ANN model will be created, adjusting the parameter in order to gain maximum performance. This model is to be compared against the control model and conclusions drawn.

Results and Discussion

Proof of Concept

This model uses the MNIST database, classifying a dataset size of 60 000 with 10 classes representing each digit.



```
Epoch 1/3
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 941us/step - accuracy: 0.8220 - loss: 4.1245
Epoch 2/3
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 927us/step - accuracy: 0.9218 - loss: 0.3650
Epoch 3/3
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 903us/step - accuracy: 0.9405 - loss: 0.2442
```

Figure 3: MNIST database model

The results proves the functionality of the code and theory and allow for the continuation of the experiment.

Control Model

An online dataset by Xai Nano of mathematical letters and symbols was used as a control[9]. The classes used were Pi, Theta, Alpha, Beta and Infinity, totalling 11 482 hand-drawn digital images. Initially, training models using the same techniques as the previous ones provided terrible results. However, due to the confirmed code functionality, it insinuates a dataset issue. It was found that imported data was getting sorted into very small mini-batches of 1 or 2. Investigating the mini-batches of the proof of concept which are of size 32, shows that this heavily affects the performance of the model. This parameter was looked into and the results are tabulated below in 1. A single epoch was used to test the following.

Model Validation Accuracy(%)						
Batch Size	Run1	Run2	Run3	Run4	Run5	Average
2	22.96	23.12	22.96	23.02	22.92	22.996
5	23.12	23.12	23.12	22.87	23.52	23.15
10	43.69	38.15	19.04	40.24	39	36.024
20	77.61	83.42	73.85	78.34	78.77	78.398
30	69.84	66.84	57.47	77.29	69.86	68.26
50	73.85	64.35	63.98	63.31	76.63	68.424
100	54.63	55.35	63.14	62.21	55.52	58.17

Table 1: Model validation accuracy for different training batch sizes; 1 epoch

All results were tested with a single epoch, with higher epochs of the lower batch sizes degrading the performance. The results show that the batch size used directly affects the performance of the model trained. Where smaller batch sizes could not ever train the model effectively, and higher batch sizes only after a single epoch drastically increase performance.

Further results were gathered on even higher batch sizes training using 7 epochs.

Model Validation Accuracy(%)						
Batch Size	Run1	Run2	Run3	Run4	Run5	Average
50	95.63	96.33	94.76	97.32	98.84	96.576
60	95.03	93.88	97.79	94.04	88.4	93.828
70	97.98	96.07	97.51	97.81	97.35	97.344
80	97.18	93.94	97.42	94.85	95.62	95.802
90	97.89	96.97	53.13	90.33	94.74	86.612
100	94.04	93.64	95.57	96.59	96.36	95.24

Table 2: Control Model validation accuracy for different training batch sizes; 7 epochs

Evident in the results, is that impressively performing models can be trained using high batch sizes with multiple epochs. This is due to the too small batch sizes over-fitting too much to a single few amount of data. With the small batch sizes, the back-propagation minimisation of the cost function makes much smaller and exact increments for very specific data. This causes the model to train slower and easily settle at a poor local minimum. Using bigger batch sizes with more iterations allows the gradient descent of the cost function to make bigger and more generalized minimisation allowing the model to find and settle to high-performing local minimums (that are closer to/are a global minimum) of the cost function more frequently, increasing the probability of training a good model for the data.

With the best-performing batch size and epoch parameters figured out, investigation into the performance associated with the hidden layer parameters of the neural network could take place.

Model Validation Accuracy(%)					
	Neurons				
Hidden Layers	64	128	256	384	512
1	96.6	98.93	93.19	99.68	98.52
2	96.47	98.81	98.26	95.24	91.96
3	92.3	96.38	88.6	76.32	50
4	93.04	97.8	97.28	99.9	22.18

Table 3: Control model validation accuracy for different numbers of hidden layer parameters

As can be seen the number of hidden layers had limited influence on the accuracy with some issues occurring at 512 neurons with both 3 and 4 neurons. This is a good example of an over-fit model. As can be seen in 4 the at a certain point the number of parameters were too many.

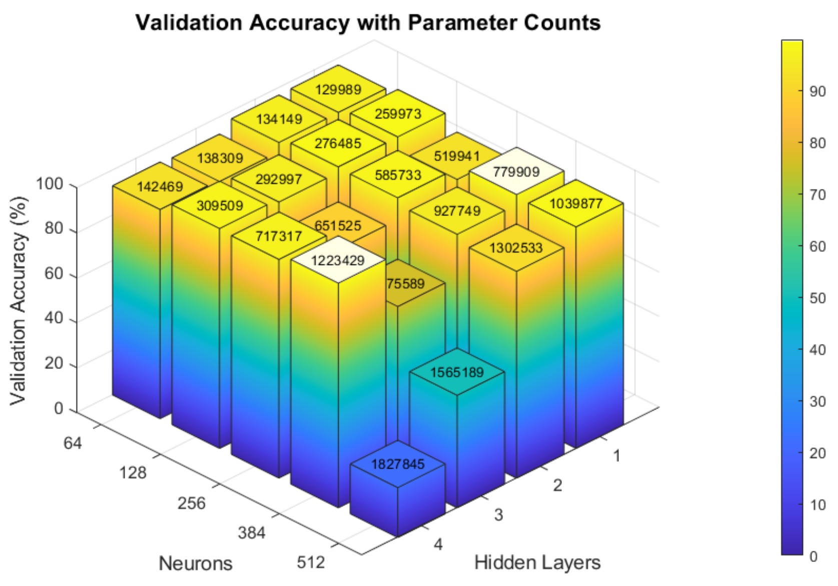


Figure 4: 3D Plot with accuracy as a gradient and number of parameters on each bar based off 3

Real World Model

Model Validation Accuracy(%)				
	Neurons			
Hidden Layers	64	128	256	384
1	83.39	90.61	90.46	90.43
2	94.01	91.24	92.1	88.41
3	87.93	85.33	87.96	91.52
4	83.05	90.65	86.2	80.6

Table 4: Real-world model validation accuracy for different numbers of hidden layer parameters

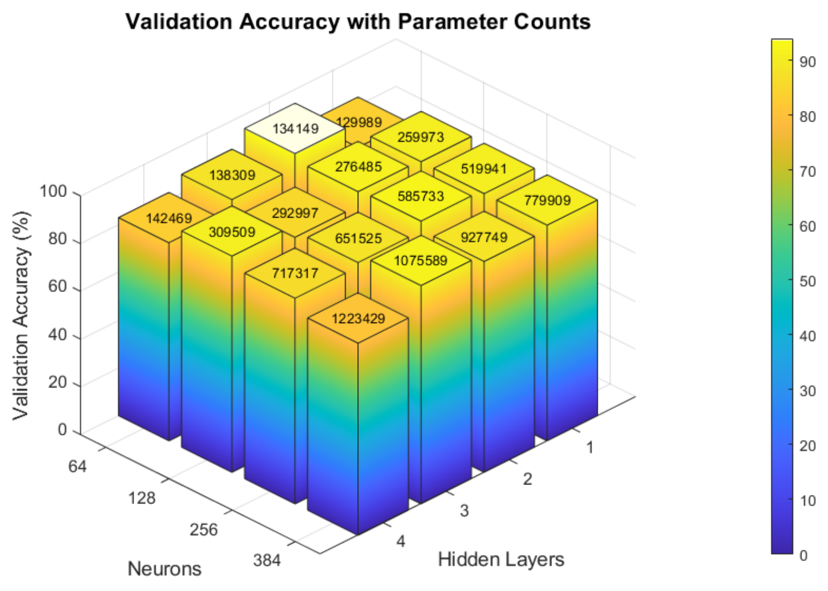


Figure 5: 3D Plot with accuracy as a gradient and number of parameters on each bar based off 4

The data used for the real world data is likely the reason worse results (more variance) are being seen in table 4 vs in 3. This could be due to a few reasons:

1. **Smaller data** - the dataset used was 2500 total examples whereas control data had 11 482 total examples. The smaller data would also lead to a smaller validation set.
2. **Noisier** - the data used for real world was first transcribed digitally in Canva on an IPad and then zipped and sent via one-drive to the laptop that was used to create the ML models. This process introduced noise as well as made the images slightly blurry, based upon the literature review it was thought to pre-process the data to remove the noise. However, when attempting this, proved to be tedious with large datasets, with poor pre-processing outputs and egregious results were produced by the model. Therefore pre-processing was not used, in future, this should be further explored.
3. **Thicker** - the real world data was drawn with a thicker pen size vs the control data. As can be seen in figure 2. This could potentially lead to irregular results when comparing the two. It was hypothesised that this would be more likely to make the model trained on the control to not translate to one trained on the real-world data and vice versa.

Model Performance across Datasets

The top-performing models were compared against the other respective datasets, with a model being trained on a combined dataset of the control and real-world sets. The results are shown in 5.

Model Validation Accuracy			
Model	Control Data	Real-world data	Combined Data
Control Model	99.78	50.42	89.66
Real-world Model	39.64	91.31	48.31
Combined data Model	96.68	91.61	98.01

Table 5: Top performing respective models accuracy against other databases

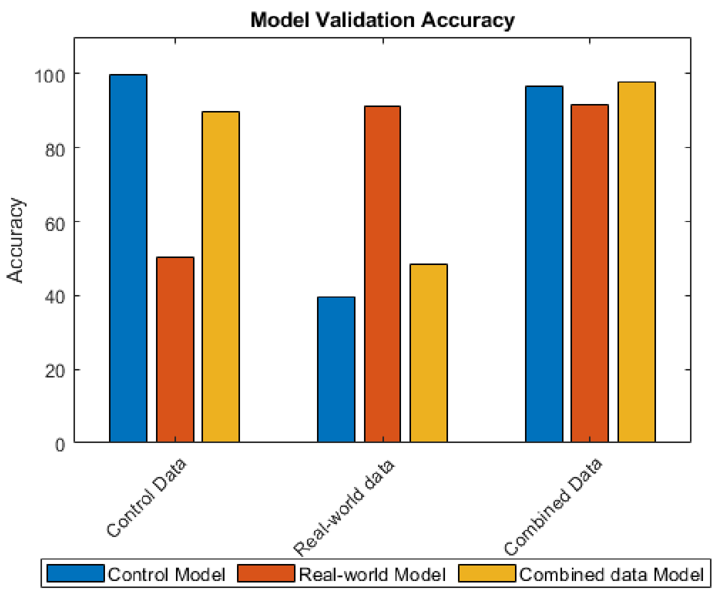


Figure 6: Top performing models visualized

The respective models performed poorly on opposing datasets as hypothesized. This was expected as the models had not been trained on similar data. As stated above, some big differences include line thickness, noise and blur between the two datasets. The combined model performs very well on both datasets, confirming our recent expectations and outlining the importance of the data used to train a model over changing their parameters. The parameters only require a certain threshold to be functional, and enough training attempts due to the randomness of the gradient descent a high-performing model can be created for a massive range of parameters. Whereas, in order for the model in supervised learning to be capable of determining any kind of handwritten symbol must be trained on all possibilities and the highest variance of styles the symbols can be written.

Conclusion

This report has demonstrated the effectiveness of artificial neural networks in accurately classifying hand-written Greek and mathematical symbols. Through comprehensive experimentation, it has been shown how variations in model architecture and training parameters significantly influence the effectiveness of training high-performing models. The importance of dataset size, variation, and quality in achieving accurate classification across diverse writing styles and line thicknesses has been emphasized. The highest-performing models were typically achieved with larger datasets that provided extensive coverage and appropriate parameters to minimize over-fitting and reduced performance on unseen data.

Future Work

While the results are promising, several avenues for future research remain open. One potential direction is to further optimize the model by exploring advanced architectures, such as convolutional neural networks (CNNs), hybrid models combining CNNs and recurrent neural networks (RNNs), and transformer-based models. Additionally, refining hyper-parameters and incorporating advanced training techniques, such as transfer learning and data augmentation, could further enhance performance. To ensure comprehensive validation, incorporate robustness checks like cross-validation. Enhance the evaluation by integrating precision, recall, and F1-score alongside accuracy. Due to space constraints in the report, these metrics were not included earlier. Expanding the dataset to include a wider variety of handwriting styles, pen thicknesses, and other symbols would improve the model's generalization capabilities. Implementing robust pre-processing techniques to handle noise, distortions, and variations in handwriting input could also enhance recognition accuracy. Investigating the integration of these neural networks into real-world applications, such as mobile apps or educational software, would be a valuable extension of this research.

The code, best-performing models, and all data used in this study can be accessed via the following GitHub repository: https://github.com/Lloydie010/RSSLO001_LNDACAI001_DSP_Project_Data_Models

Bibliography

[1] N. Sakshi and V. Kukreja, "Machine learning and non-machine learning methods in mathematical recognition systems: Two decades' systematic literature review," *Multimedia tools and applications*, vol. 83, no. 9, p. 27831–27900, Aug 2023. [Online]. Available: https://doi.org/10.1007/s11042-023-16356-z

[2] H. Wang, T. Pan, and M. Ahsan, "Hand-drawn electronic component recognition using deep learning algorithm," *International Journal of Computer Applications in Technology*, vol. 62, p. 13, 01 2020. [Online]. Available: 10.1504/IJCAT.2020.103905

[3] K. Dutta, P. Krishnan, M. Mathew, and C. Jawahar, "Improving cnn-rnn hybrid networks for handwriting recognition," in *2018 16th international conference on frontiers in handwriting recognition (ICFHR)*. IEEE, 2018, pp. 80–85. [Online]. Available: https://ieeexplore.ieee.org/document/8563230

[4] M. Dey, S. M. Mia, N. Sarkar, A. Bhattacharya, S. Roy, S. Malakar, and R. Sarkar, "A two-stage cnn-based hand-drawn electrical and electronic circuit component recognition system," *Neural Computing and Applications*, vol. 33, no. 20, p. 13367–13390, Apr 2021. [Online]. Available: https://doi.org/10.1007/s00521-021-05964-1

[5] N. E. Matsakis, "Recognition of handwritten mathematical expressions," Ph.D. dissertation, Massachusetts Institute of Technology, 1999. [Online]. Available: https://www.researchgate.net/publication/349747489_Online_Handwritten_Mathematical_Expression_Recognition_and_Applications_A_Survey

[6] T. Bluche, "Mathematical formula recognition using machine learning techniques," Ph.D. dissertation, University of Oxford, 2010. [Online]. Available: http://www.tbluche.com/files/dissertation.pdf

[7] A. K. Priya, H. KN, J. S. Chembeti *et al.*, "Survey on hand drawn symbol classification and recognition," in *Proceedings of the International Conference on Smart Data Intelligence (ICSMDI 2021)*, 2021. [Online]. Available: https://dx.doi.org/10.2139/ssrn.3852780

[8] A. Krogh, "What are artificial neural networks?" *Nature Biotechnology*, vol. 26, no. 2, p. 195–197, Feb 2008. [Online]. Available: https://www.nature.com/articles/nbt1386

[9] X. Nano, "Handwritten math symbols dataset," 2017. [Online]. Available: https://www.kaggle.com/datasets/xainano/handwrittenmathsymbols

Appendix

```
% Constants
widthC = 45;
heightC = 45;
neurons = [64, 128, 256, 384, 512];
hidden_layers = [1, 2, 3, 4];
output_neurons = 5;

% Initialize parameters array
parameters = zeros(length(hidden_layers), length(neurons));

% Calculate the number of parameters for each configuration
for i = 1:length(hidden_layers)
    for j = 1:length(neurons)
        input_to_hidden1 = (widthC * heightC * neurons(j)) + neurons(j);
        hidden_to_hidden = (neurons(j) * neurons(j) + neurons(j)) * (hidden_layers(i) - 1);
        hidden_to_output = (neurons(j) * output_neurons) + output_neurons;
        parameters(i, j) = input_to_hidden1 + hidden_to_hidden + hidden_to_output;
    end
end

% Accuracy data from the table
accuracy = [
    96.6, 98.93, 93.19, 99.68, 98.52;
    96.47, 98.81, 98.26, 95.24, 91.96;
    92.3, 96.38, 88.6, 76.32, 50;
    93.04, 97.8, 97.28, 99.9, 22.18
];

% Plot the 3D bar graph
figure;
h = bar3(accuracy);

% Create a custom colormap that transitions to a light color
custom_cmap = parula(256); % Start with parula colormap
custom_cmap(end, :) = [1, 1, 0.9]; % End with a light yellowish color

```

```
% Apply the custom colormap to the bars
colormap(custom_cmap);

% Customize the bar colors based on parameters
for k = 1:length(h)
    zdata = h(k).ZData;
    h(k).CData = zdata;
    h(k).FaceColor = 'interp';
end

% Label the axes
set(gca, 'XTickLabel', neurons, 'YTickLabel', hidden_layers, 'FontSize', 12);
xlabel('Neurons', 'FontSize', 14);
ylabel('Hidden Layers', 'FontSize', 14);
zlabel('Validation Accuracy (%)', 'FontSize', 14);
title('Validation Accuracy with Parameter Counts', 'FontSize', 16);

% Add text annotations for each bar (parameters only)
for i = 1:size(accuracy, 1)
    for j = 1:size(accuracy, 2)
        xpos = j; % x position for text
        ypos = i; % y position for text
        zpos = accuracy(i, j); % z position for text
        textString = sprintf('%d', parameters(i, j)); % text string with parameters
        text(xpos + 0.145, ypos + 0.145, zpos, textString, 'HorizontalAlignment', 'center',
    ↪ 'VerticalAlignment', 'bottom', ...
            'FontSize', 10, 'Color', 'k'); % Removed background color
    end
end

% Adjust view angle for better visibility
view(45, 30);

% Optional: Adjust figure size and layout
set(gcf, 'Position', [100, 100, 1000, 600]); % Adjust figure size
```

Listing 1: Code used for figure 4

```
% Constants
widthC = 45;
heightC = 45;
neurons = [64, 128, 256, 384];
hidden_layers = [1, 2, 3, 4];
```

```
output_neurons = 5;

% Initialize parameters array
parameters = zeros(length(hidden_layers), length(neurons));

% Calculate the number of parameters for each configuration
for i = 1:length(hidden_layers)
    for j = 1:length(neurons)
        input_to_hidden1 = (widthC * heightC * neurons(j)) + neurons(j);
        hidden_to_hidden = (neurons(j) * neurons(j) + neurons(j)) * (hidden_layers(i) - 1);
        hidden_to_output = (neurons(j) * output_neurons) + output_neurons;
        parameters(i, j) = input_to_hidden1 + hidden_to_hidden + hidden_to_output;
    end
end

% Accuracy data from the table
accuracy = [
    83.39, 90.61, 90.46, 90.43;
    94.01, 91.24, 92.1, 88.41;
    87.93, 85.33, 87.96, 91.52;
    83.05, 90.65, 86.2, 80.6
];

% Plot the 3D bar graph
figure;
h = bar3(accuracy);

% Create a custom colormap that transitions to a light color
custom_cmap = parula(256); % Start with parula colormap
custom_cmap(end, :) = [1, 1, 0.9]; % End with a light yellowish color

% Apply the custom colormap to the bars
colormap(custom_cmap);

% Customize the bar colors based on parameters
for k = 1:length(h)
    zdata = h(k).ZData;
    h(k).CData = zdata;
    h(k).FaceColor = 'interp';
end

% Label the axes
set(gca, 'XTickLabel', neurons, 'YTickLabel', hidden_layers, 'FontSize', 12);
```

```
xlabel('Neurons', 'FontSize', 14);
ylabel('Hidden Layers', 'FontSize', 14);
zlabel('Validation Accuracy (%)', 'FontSize', 14);
title('Validation Accuracy with Parameter Counts', 'FontSize', 16);

% Add text annotations for each bar (parameters only)
for i = 1:size(accuracy, 1)
    for j = 1:size(accuracy, 2)
        xpos = j; % x position for text
        ypos = i; % y position for text
        zpos = accuracy(i, j); % z position for text
        textString = sprintf('%d', parameters(i, j)); % text string with parameters
        text(xpos + 0.135, ypos + 0.135, zpos, textString, 'HorizontalAlignment', 'center',
    ↪ 'VerticalAlignment', 'bottom', ...
             'FontSize', 10, 'Color', 'k'); % Removed background color
    end
end

% Adjust view angle for better visibility
view(45, 30);

% Optional: Adjust figure size and layout
set(gcf, 'Position', [100, 100, 1000, 600]); % Adjust figure size
```

Listing 2: Code used for figure 5

```
% Define the data
models = {'Control Model', 'Real-world Model', 'Combined data Model'};
data_types = {'Control Data', 'Real-world data', 'Combined Data'};
accuracy = [99.78, 50.42, 89.66;
            39.64, 91.31, 48.31;
            96.68, 91.61, 98.01];

% Create a figure
figure('Color', 'w');

% Create a grouped bar chart
bar(accuracy, 'grouped');

% Set labels and title
xlabel('Model', 'FontSize', 12);
ylabel('Accuracy', 'FontSize', 12);
title('Model Validation Accuracy', 'FontSize', 16);
```

```

% Set x-tick labels
set(gca, 'XTick', 1:numel(data_types), 'XTickLabel', data_types, 'XTickLabelRotation', 45, '
    ↪ FontSize', 10);

% Add a legend
legend(models, 'Location', 'NorthWest', 'FontSize', 10);

% Adjust y-axis limits
ylim([0 110]);
```

Listing 3: Code used for figure 6