FYS-STK4155 Project 3

Lars-Martin Gihle, Iris Bore & Jonas Båtnes
University of Oslo, Department of Physics
(Dated: December 14, 2024)

I. ABSTRACT

With the rise of artificial neural networks driven by improved computational resources, optimizing models for specific tasks has become crucial. Convolutional Neural Networks (CNNs) excel in image classification, proving valuable in applications such as autonomous driving and diagnostics. In this study, we conducted a systematic grid search to optimize digit classification on the MNIST dataset. We varied key hyperparameters, including the number of convolutional layers, kernel sizes, dropout rates, pooling, padding and activation functions, to assess their impact on performance. Our best CNN model achieved an accuracy of 98.9%, significantly surpassing the 92.0% accuracy of the baseline Logistic Regression model. These findings highlight the critical role of hyperparameter optimization in improving CNN performance, underscoring the importance of tailored model design for specific tasks. The significant improvement over traditional methods, such as Logistic Regression, demonstrates the potential of CNNs in achieving high accuracy for complex classification tasks.

II. INTRODUCTION

In recent years, convolutional neural networks (CNNs) have emerged as a cornerstone technology in computer vision and broader industrial applications, powering innovations in areas such as autonomous vehicles, facial recognition, and quality inspection [1, 2]. By leveraging convolutional layers to capture local spatial patterns, pooling operations to reduce dimensionality, and fully connected layers to integrate learned features, CNNs have demonstrated exceptional performance across a wide range of classification, detection, and segmentation tasks [3]. Their capacity to automatically learn hierarchical features from raw input data makes them particularly well-suited for complex image-based challenges.

In parallel, classical approaches such as Logistic Regression offer a useful baseline for understanding dataset complexity and benchmarking the performance gains afforded by more sophisticated neural architectures [4]. While Logistic Regression alone is not typically suited for image classification, comparing it against CNNs helps illuminate the advantages of nonlinear feature extraction and deep representation learning. To further explore the performance of Logistic Regression on classification tasks, we experiment with multiple hyperparameters. Specifically, we test five distinct learning rates, batch sizes, and numbers of epochs to assess how these variations influence the model's classification accuracy and convergence behavior.

In this study, we focus on the well-known MNIST dataset, a benchmark collection of handwritten digit images that serves as a standardized platform for evaluating classification models [5]. We employ a systematic approach to hyperparameter optimization by conducting a comprehensive grid search over CNN architectures and their key hyperparameters. By exploring various configurations, such as differing numbers of convolutional filters, kernel sizes, dropout rates, pooling, padding and network depths, we aim to identify the CNN architecture that yields the best classification accuracy on MNIST.

Our evaluation metrics center on classification accuracy, allowing us to directly compare the predictive power of our best-performing CNN against the best-performing Logistic Regression model. We also discuss the computational considerations associated with both

approaches, as modern industrial applications often require methods that are both accurate and efficient.

This article is structured as follows: We begin by introducing the MNIST dataset and detailing the preprocessing steps taken to ensure data quality and consistency. We then outline the architectural components of our CNN and the rationale behind their effectiveness in image classification tasks. We next define our baseline Logistic Regression model and the variations introduced in learning rates, batch sizes, and epochs to investigate their impact on performance. Subsequently we describe the K-fold cross validation and bootstrap algorithms which we use for model validation and evaluation. Following this, we describe the grid search procedure employed to explore the hyperparameter space of CNN architectures and linear configurations. Subsequently, we present and analyze the results, highlighting how different CNN architectures and hyperparameters compare against the various Logistic Regression configurations. Finally, we summarize our findings, discuss their implications for real-world industrial contexts, and suggest directions for future researchs.

III. METHODS

This study primarily focused on implementing and comparing the performance of various Convolutional Neural Networks (CNNs) with different layers, kernel sizes, numbers of filters, activation functions, and regularization techniques on the MNIST dataset. To illustrate the advantages of CNNs, we benchmarked our results against a Logistic Regression (LR) model tuned with five different learning rates, batch sizes, and numbers of epochs. To ensure the robustness and reliability of our findings, we employed several data validation and resampling techniques, including train-test splitting, cross-validation on the training set, and bootstrapping on the test set.

A. Data Loading and Preprocessing

1. Train-Test Split

We used PyTorch to load the MNIST dataset, which comprises 70,000 images in total. This dataset is pre-divided into a training set of 60,000 images and a test set of 10,000 images. During the optimization process, the training set was utilized to iteratively update the model parameters in order to minimize the loss function. Meanwhile, the test set remained completely untouched until the final evaluation phase. This separation allowed us to evaluate the model's ability to generalize to new, unseen data. By conducting model selection and hyperparameter tuning exclusively on the training data, and reserving final performance checks for the test data, we obtained a more realistic measure of the model's predictive power and robustness.

2. Feature Scaling

When loading our data set, we scaled the pixels to be within the range [0, 1]. This normalization ensured that the input features (pixels) lied within a more uniform and manageable range than the original pixel range of [0,255]. As a result, the optimization routines, Stochastic Gradient Descent for LR and Adam for CNN, could converge more quickly and reliably, reducing the chances of getting stuck in poor local minima or suffering from numerical instabilities. In essence, feature scaling fosters a more stable and efficient learning

environment for the model.

B. Classification Techniques

The following classification methods were implemented:

1. *Convolutional Neural Network*

Convolutional Neural Networks are a type of deep learning model specifically designed for processing structured grid data, such as images. CNNs exploit spatial hierarchies in data by applying convolutional operations to extract features. These features are then used to perform classification tasks.

A key component of a CNN is the convolution operation. For a 2D input, the convolution is defined as in equation 1:

$$(f * g)(x, y) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(m, n) \cdot g(x - m, y - n) \tag{1}$$

Here, $f(m, n)$ represents the input image, and $g(x - m, y - n)$ is the filter or kernel. The kernel slides over the image to compute feature maps.

To control the size of the output feature maps, CNNs often use padding and strides. Padding can increase the dimensionality of the output and as such avoid the shrinking that convolutional layers usually apply to the tensor when passing through them. The size of the output feature map, D_{out}, for an input of size D_{in}, a kernel (filter) size K, stride S, and padding P, is calculated as in equation 2:

$$D_{out} = \left\lfloor \frac{D_{in} - K + 2P}{S} \right\rfloor + 1 \tag{2}$$

The output tensor is then of dimensions $D_{out} \times D_{out} \times C$ where C is the number of channels. Padding is also useful to help capture more information from edge pixels, since it allows filters to capture edge pixels more times as it slides over the image. The final layers of a CNN are typically fully connected (dense) layers, which map the extracted features to class probabilities using the softmax function in equation 5:

$$p(y = c \mid \boldsymbol{x}) = \frac{\exp(z_c)}{\sum_{k=1}^{C} \exp(z_k)} \tag{3}$$

Here, z_c represents the output score for class c, and C is the total number of classes.

CNNs are trained using backpropagation. The loss function cross-entropy, which we use consistently in all of our experiments 4, guides the optimization of the weights:

$$L = -\sum_{i=1}^{n} \sum_{c=1}^{C} y_{i,c} \log \hat{y}_{i,c} \tag{4}$$

In this equation, $y_{i,c}$ is the true label, and $\hat{y}_{i,c}$ is the predicted probability for class c.

CNNs have demonstrated exceptional performance in image classification tasks by automatically learning hierarchical features from raw input data [6].

2. Logistic Regression

Logistic regression can be used for multiclass classification problems, predicting the probability that an input $\boldsymbol{x}_i$ belongs to one of several classes $y_i \in 0, 1, \ldots, K-1$. It models these probabilities using the softmax function in equation 5, which outputs values between 0 and 1 that sum to 1 across all classes:

$$p(y_i = k|\boldsymbol{x}_i, \boldsymbol{\beta}) = \frac{\exp\left(\boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{\beta} k\right)}{\sum j = 0^{K-1} \exp\left(\boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{\beta}_j\right)}, \quad \text{for } k = 0, 1, \ldots, K-1. \tag{5}$$

To estimate the parameters $\boldsymbol{\beta}$, Maximum Likelihood Estimation (MLE) is used. Assuming independent data points, the likelihood function for the dataset $\mathcal{D} = (\boldsymbol{x}_i, y_i)$ is described in equation 6:

$$L(\boldsymbol{\beta}) = \prod_{i=1}^{n} \prod_{k=0}^{K-1} [p(y_i = k|\boldsymbol{x}_i, \boldsymbol{\beta})]^{\mathbb{1}(y_i = k)}, \tag{6}$$

where $\mathbb{1}(y_i = k)$ is an indicator function that equals 1 if $y_i = k$ and 0 otherwise.

The log-likelihood function is then expressed as in equation 7:

$$\ell(\boldsymbol{\beta}) = \sum_{i=1}^{n} \sum_{k=0}^{K-1} \mathbb{1}(y_i = k) \log p(y_i = k|\boldsymbol{x}_i, \boldsymbol{\beta}). \tag{7}$$

The cost function to minimize in our implementation is the negative log-likelihood, which is equivalent to the cross-entropy loss as shown in equation 8:

$$C(\boldsymbol{\beta}) = -\ell(\boldsymbol{\beta}) = -\sum_{i=1}^{n}\sum_{k=0}^{K-1} \mathbb{1}(y_i = k) \log p(y_i = k|\boldsymbol{x}_i, \boldsymbol{\beta}). \tag{8}$$

The gradient of the cost function, used for optimization algorithms like gradient descent, is expressed in matrix form as in equation 9:

$$\nabla_{\boldsymbol{\beta}} C(\boldsymbol{\beta}) = -\mathbf{X}^{\mathrm{T}}(\mathbf{Y} - \mathbf{P}), \tag{9}$$

where $\mathbf{X}$ is the design matrix with rows $\boldsymbol{x}_i^{\mathrm{T}}$, $\mathbf{Y}$ is a one-hot encoded matrix of observed labels, $\mathbf{P}$ is a matrix of predicted probabilities $p(y_i = k|\boldsymbol{x}_i, \boldsymbol{\beta})$ for all k.

Parameters are estimated by minimizing $C(\boldsymbol{\beta})$ using iterative methods, such as gradient descent, since there is no closed-form solution for logistic regression coefficients [7].

C. Optimization Techniques

To explore the optimization the convolutional network performance, we used various techniques described in the following:

1. Kernel Size and Number of Filters

We conducted a grid search with kernel sizes $K = 3, 5, 7$ and filter counts $F = 8, 16, 32, 64$. Kernel size determines the spatial extent of the receptive field, controlling how much of the input image each convolution operation captures. Smaller kernel sizes, such as 3×3, allow for precise feature extraction and can be stacked to achieve a larger effective receptive field. In contrast, larger kernel sizes, such as 5×5 and 7×7, capture broader features but may lead to higher computational costs and increased risk of overfitting.

The number of filters F defines the depth of the output tensor, which determines the model's ability to learn diverse features. Increasing the number of filters enhances the network's capacity to capture complex patterns but also raises computational requirements and the potential for overfitting.

2. *Padding and Pooling:*

We used Max Pooling, which is used to reduce the dimensionality of an input tensor (usually after a convolutional layer). For a kernel size of n, scans $n \times n$ grids and selects the highest values respectively. This grid moves over a number of elements equal to the pooling layer's stride between each such scan, and in our implementation using torch.nn.MaxPool2d(), stride is always equal to kernel size such that these filters never overlap. Pooling modifies the dimensions of the tensor passed through it as shown in equation 10.

$$D_{out} = \left\lfloor \frac{D_{in} - K + 2P}{S} \right\rfloor \tag{10}$$

Here D represents the height and width of the input tensor (size $= D \times D \times C$ where C is channels). K, P, and S are the kernel size, padding size, and stride of the pooling layer respectively.

Padding (in convolutional layers) and Pooling are complimentary, increasing and decreasing tensor dimensions respectively. Setting padding (P) in a convolutional layer to $\frac{K-1}{2}$ where K is the kernel size means the output tensor has the same dimensionality (discounting number of channels) as the input tensor, assuming stride $= 1$ (2). We test $P = 0, \frac{K-1}{2}$, and Pooling Kernel Size $= 0, 2$, where 0 essentially means no pooling layer in a grid search. These values were chosen because they are common and pooling layers were placed after each convolutional layer.

3. *Dropout and Activation functions*

We performed a grid search with various dropout layers, and the activation functions ReLU and Leaky ReLU (used for all except the last output) as ReLU is the most commonly used activation function in convolutional neural networks, while leaky ReLU has also proven useful in convolutional networks [8]. Dropout is implemented in our code using torch.nn.Dropout(). Dropout sets the output of neurons of the preceding layer to zero with a set probability between zero and one. We tested the use of dropout after the first two (of three) linear layers, with probabilities {0.3, 0.4, 0.5} (same for both) in, as is similar to one of Raschka's implementations [6, p. 478].

D. Validation and Evaluation techniques

1. *Cross-Validation with K-Folds*

To assess the models' generalization capability and to avoid overfitting, we employed K-fold cross-validation with $K = 5$. In this method, the training data is partitioned into K subsets (folds). The model was trained on $k - 1$ folds and validated on the remaining fold. This process was repeated k times, each time with a different fold used for validation. The cross-validation procedure provides an average performance metric, which is more reliable than a single train-test split. The accuracy score were calculated for each fold and then averaged to obtain the final performance metrics. The algorithm is presented with pseudocode in Algorithm 1.

Algorithm 1 K-fold Cross Validation [9]

procedure K-FOLDCROSSVALIDATION($model, X, z, nfolds$)
    Divide data into K equal folds
    for $k \in \text{range}(0, K)$ **do**
        $V \leftarrow \text{Fold}_k$ in data
        $T \leftarrow \text{data} \setminus V$ ▷ Training on data except the validation data
        Train T
        $Acc_k \leftarrow$ evaluate V with trained model ▷ Accuracy evaluated for one fold
    $Acc \leftarrow \frac{1}{K} \sum_{k=1}^{K} Acc_k$ ▷ Total accuracy is evaluated

2. *Bootstrapping*

In addition to cross-validation, we used non-parametric bootstrap to estimate the stability of our model performance. Our bootstrapped implementation repeatedly resampled the test data with replacement to create multiple bootstrap samples B. For each bootstrap sample B, we evaluated model performance. The bootstrap algorithm is presented with pseudocode in Algorithm 2. This method allowed us to compute confidence intervals for the model performance. [4]

3. Confidence Intervals

After running the boostrap algorithm, we constructed a 95 % confidence interval of our model accuracy. A 95% confidence interval is a range of values that, with 95% probability, contains the true unknown value of a parameter[10]. The lower and upper limits of our confidence interval were calculated by taking the 2.5th and 97.5th percentiles of the bootstrapped accuracy estimates.

Algorithm 2 Bootstrap Algorithm

procedure BOOTSTRAPPING(B, model, data)
  for $b = 1$ to B **do**
    $\mathcal{D}_{\text{test}}^{(b)} \leftarrow$ Sample n data points from $\mathcal{D}$ with replacement
    Evaluate the model on $\mathcal{D}_{\text{test}}^{(b)}$ and record the estimated accuracy
  Construct confidence intervals by taking the percentiles of the boostraped estimates.

E. Implementation

1. CNN

The CNN class was implemented using the `PyTorch` library [11], drawing inspiration from the "Training a Classifier" tutorial in the PyTorch documentation [12]. After studying this example as a proper way to code a CNN with PyTorch, we created a class capable of accommodating various network structures, enabling us to perform grid searches for the optimal configuration. For the initial parameters of our grid search experiments, we partially adopted the values from Raschka's example of a CNN for the MNIST dataset [6]. However, we opted to limit the number of epochs to ten since Raschka's validation curves indicated convergence after five epochs.

2. Logistic Regression

The logistic regression was also implemented using the `PyTorch` library. The original inspiration on how to set up Logistic Regression was found with help from Anthropic's

Claude 3.5 Sonnet AI model [13]. The code was however modified as we saw we could use the same structure as our CNN model. Based on initial experiments, we selected a learning rate in the range from 10^{-5} to 10^{0}, batch sizes in the range from 16 to 256, and epochs in the range from 5 to 25.

3. Selection Criterion

Due to our limited computational resources and the large size of our training dataset, it was not feasible to test every possible configuration of our model architecture and hyperparameters. Therefore, we opted to start with informed initial values for our hyperparameters based on sources like Raschka [6], rather than selecting them randomly. After each grid search, we iteratively replaced the initial parameters with the best-performing ones. Our criterion for selection was to choose the parameter that maximized cross-validated accuracy. In case of a tie between parameters, we always selected the value that represented a simpler model structure and incurred lower computational costs. It is worth noting that we only suggested values within a range that preliminary experiments indicated would align with our computational resources.

Raschka inspired the choice of initial values in our convolutional neural networks [6]. We used 10 or 20 epochs (as specified), a batch size of 64, a convolutional layer stride of 1, and a pooling layer (where applicable) stride of 1 and padding of 0. The following steps summarize the implementation process of our experiment, with the initial values for LogReg presented in Table I and for CNN presented in Table II:

TABLE I: Table with inital values for LogReg before hyper parameter tuning.

	Optimizer	**Learning Rate**	**Batch Size**	**Epochs**
Initial Values	SGD	Best of first grid search	64	10

TABLE II: Table with inital values for CNN before hyper parameter tuning, adapted from Raschka [6]

	Learning rate	**Optimizer**	**Kernel size**	**Filter Numbers**	**Padding** (conv. layer)	**Pooling kernel size** (After each conv. layer)
Inital values	0.001	Adam	5	(32, 64)	0	2

1. **Data Loading and Splitting:** Load the MNIST data set and split it into train and test sets using `torchvision.datasets`.

2. **Tune Baseline Model**: We use stochastic gradient decent as optimizer and negative log-likelihood for Logistic Regression.Tune our Logistic Regression model using cross-validation.

3. **Tune CNN**: Grid search the hyperparameters of our CNN in the following order, using cross-validation:

   (a) **Number of Linear and Convolutional Layers** Test one convolutional layer followed by two linear layers against two convolutional layer followed by three linear layers.

   (b) **Kernel Size and Number of Filters** We test kernel sizes: {3, 5, 7} and numbers of filters: {(8,16), (16, 32), (32,64)}.

   (c) **Pooling and Padding** We test convolutional layer padding values of $\{0, \frac{K-1}{2}\}$, where K is the convolutional layer kernel size, and Pooling Kernel sizes: {0,2}.

   (d) **Dropout rates and Activation Functions** We perform a grid search with the dropout rates {0.3, 0.5} and Activation Functions: {ReLU, Leaky ReLU}.

4. **Model Training:** Train CNNs and Logistic Regression on the training data with their respective validated hyper parameters.

5. **Testing:** Assess the final model performance on the test set.

F. Performance Metrics

To measure the performance of the classification problem, we use the accuracy score in equation 11:

$$Accuracy = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n}, \tag{11}$$

where the number of correctly guessed targets t_i is divided by the total number of targets.

G. Reproducibility

To ensure the reproducibility of our computational experiments, we have taken the following steps:

- Used a well known public data set with published benchmarked accuracies
- Presented all final model parameters, as well as values that were tested during tuning of hyper parameters
- Used config files for hyperparameters to ensure consistency and readability across different scripts.
- Seeded everything with the initial seed from a separate config file.
- Used cross-validation with varying seeds across separate runs to validate the results infurther.
- Used bootstrapped uncertainty measures for final model, where we vary the seeds across different runs.

H. Testing

We tested our implemention by comparing the performance of our model with the same initial values and structure as that of known benchmarked accuracies on the MNIST data set [6] and [1]. As a bonus test, we have made ChatGPT review our code, and ask it do describe what the scripts do. One could also argue that we have made our implementation more robust by having it reviewed by all members of the group, which in our experience increased the chance of catching errors.

I. Large Language Models

With inspiration from FYS-STK4155 project 1 [14] we summarize our use of large language models: We have been encouraged in the group sessions to use ChatGPT [15] in writing this report. We have done so in various ways. The abstract was created by first writing the whole paragraph, then sending it to ChatGPT with the prompt "Can you rewrite this with better and more concise language? Keep the references as they are and only keep the most important equations to explain the methods:". We have then read through the suggestion closely to make sure the values and content are the same as before. For the introduction and parts of the method we have asked ChatGPT to write the whole paragraph based on selected criteria and a template. After we have read through and done corrections to fit the overall report. We hope that this makes it easier for the reader to follow our discussion, especially when we discuss the figures. Screenshots from conversations with ChatGPT are uploaded in a folder on our GitHub. We have also used Github Copilot as an integrated tool [16]. Antropic's Cloude 3.5 Sonnet was used for inspiration for the implementation of the original Logistic Regression code, but it was later modified by us to be on the same format as the CNN model. [13].

J. Other Tools

We used the software Overleaf to write this report. To do basic mathematical operations we used the NumPy package [17]. Plotting our results was done with the Matplotlib package [18]. The code for the project can be found at: https://github.com/irisbore/FYS-STK4155_project3.git.

IV. RESULTS AND DISCUSSION

A. Results: Tuning Logistic Regression

For logistic regression, we observe that the learning rate, number of epochs, and batch size impact accuracy in different ways. As illustrated in Figure 1, model accuracy improves as the learning rate increases, from the smallest to the largest value. The highest accuracy achieved is approximately 92%, representing a total increase of about 70%. The best learning rate is 0.1. A limitation of this result is its sensitivity to the initial number of epochs, which for this run is 10. Smaller learning rates often require more epochs for the model to converge. Since the goal of this tuning is merely to establish a reasonable baseline model, we accepted this result and selected 0.1 as the learning rate.

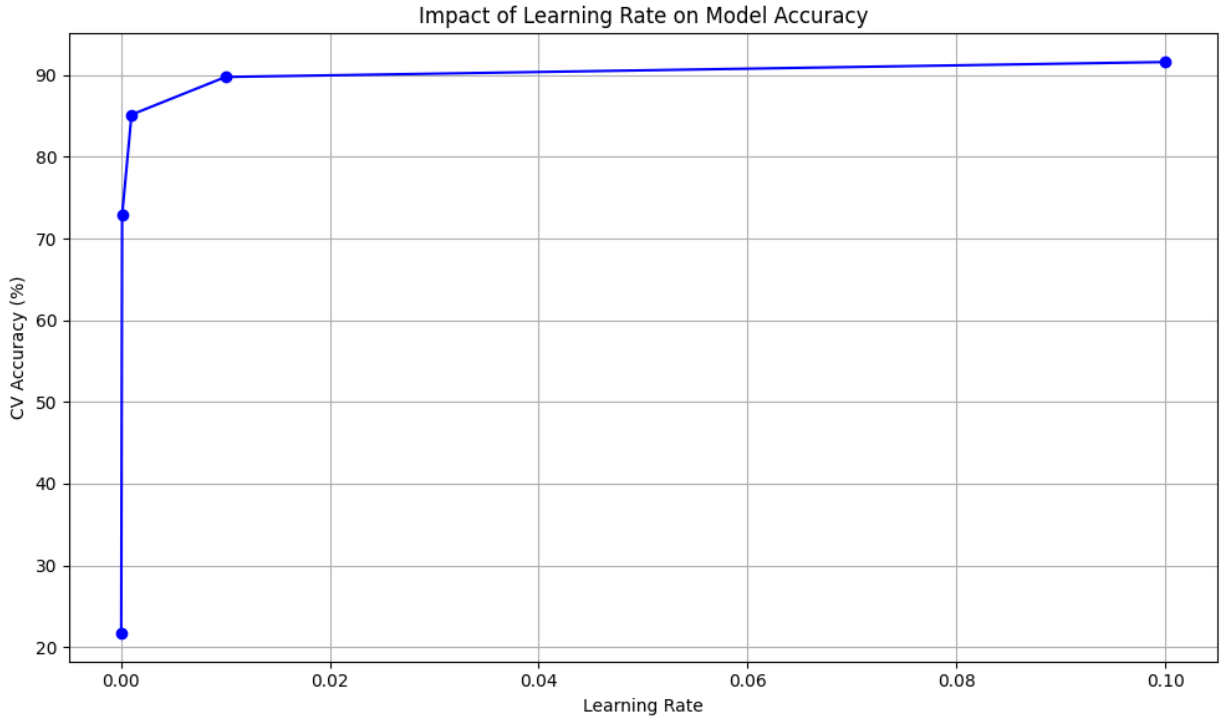

FIG. 1: Logistic regression classification with learningrates in the range from 10^{-5} to 10^{-1}. The highest accuracy of 91% was achieved with a learning rate of 0.1.

From Figure 2 we can see that accuracy increases with the number of epochs. A maximum accuracy of 92% was achieved with 25 epochs, which gives a difference of 0.8% from the lowest score obtained with 5 epochs. We continue with 25 epochs for our final model.

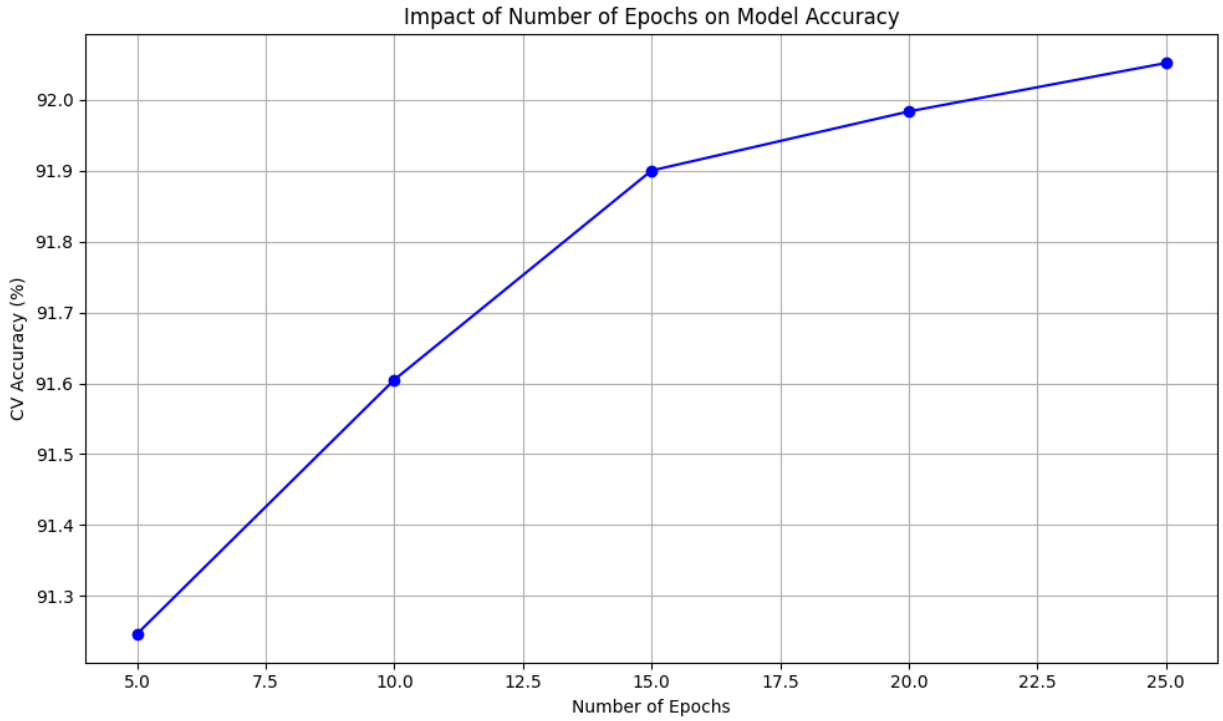

FIG. 2: Logistic regression classification with epochs in the range from 5 to 25. The best accuracy achieved is 92% with 25 epochs.

Referring to Figure 3, we observe that, unlike the number of epochs, an increase in batch size results in lower accuracy. The highest accuracy of 91.8% is achieved when a batch size 32 is used. In contrast a batch size of 256 gives the lowest accuracy of 90.8%. It is important to note that the absolute decrease in accuracy is approximately 1% for batch size and 0.8% for epochs, which is relatively small. This suggests that these two variables are less critical for optimizing accuracy compared to the learning rate, which showed a 70% difference between the lowest and highest values. However, we choose a batch size of 32 for our final model, in line with our selection criteria.

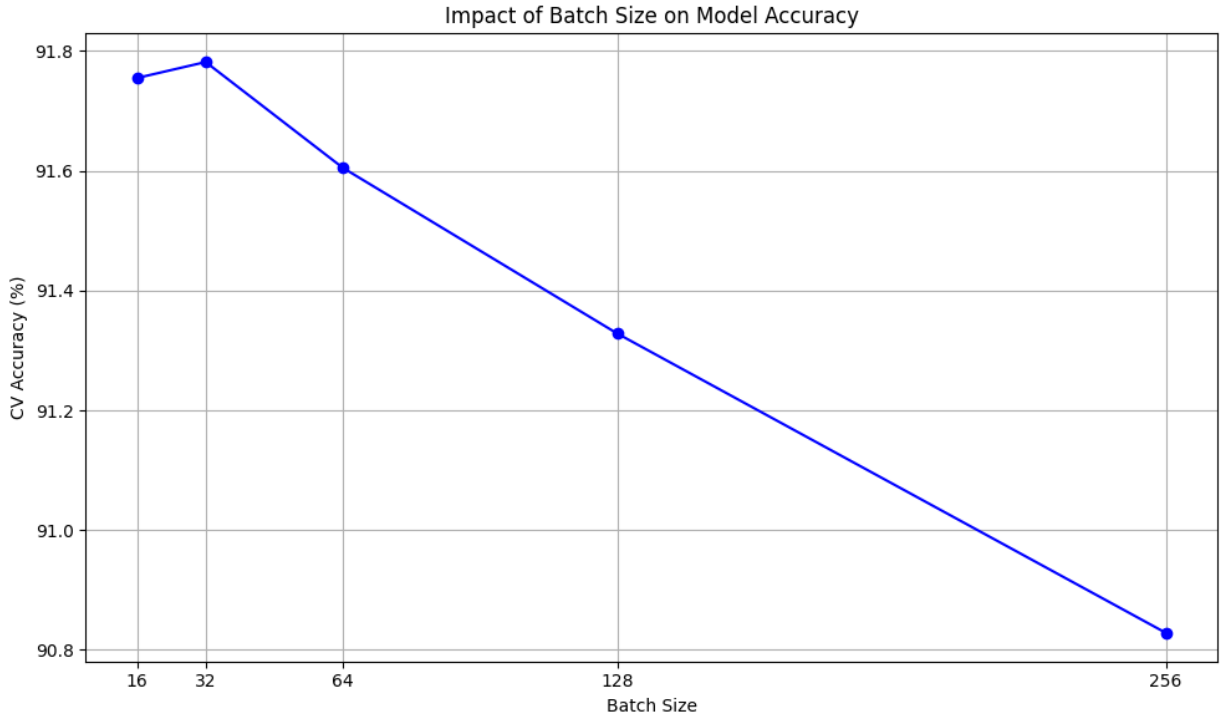

FIG. 3: Logistic regression classification with batch size in the range from 16 to 256. The highest accuracy achieved is 91.8% when batch size is 32.

B. Results: Grid search on CNN

1. *Number of Linear Layers and Convolutional Layers*

From Figure 4, we found that the addition of another convolutional and linear layer increases the accuracy by 0.2%, from 98.7% to 98.9% from the initial configuration. This suggests that, even with its relative simplicity compared to real-life datasets, the MNIST dataset does benefit from a somewhat deeper neural network structure for classification. We decided to go on with two convolutional and three linear layers.

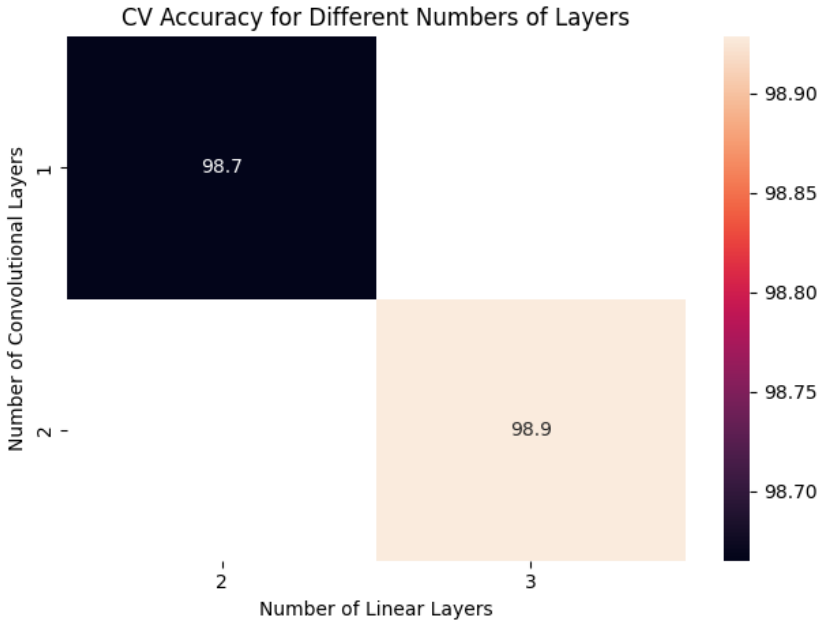

FIG. 4: Results of a grid search comparing a convolutional neural network (CNN) with one convolutional layer followed by two linear layers, against a CNN with two convolutional layers followed by three linear layers.

2. *Kernel Size and Number of Filters*

Our grid search testing different kernel sizes and numbers of filters in the two convolutional layers is illustrated in Figure 5. We found that our initial structure of (32, 64) led to the best performance, so we maintained that choice. Additionally, our initial kernel size of 5 performed better than 7 in all cases, and better than 3 for (32, 64) filters, so we continued with a kernel size of 5. The choice of a kernel size of 5 over 7 also benefits runtime because a lower kernel size reduces computational load. Overall the best cross-validated performance we achieved was 98.9%.

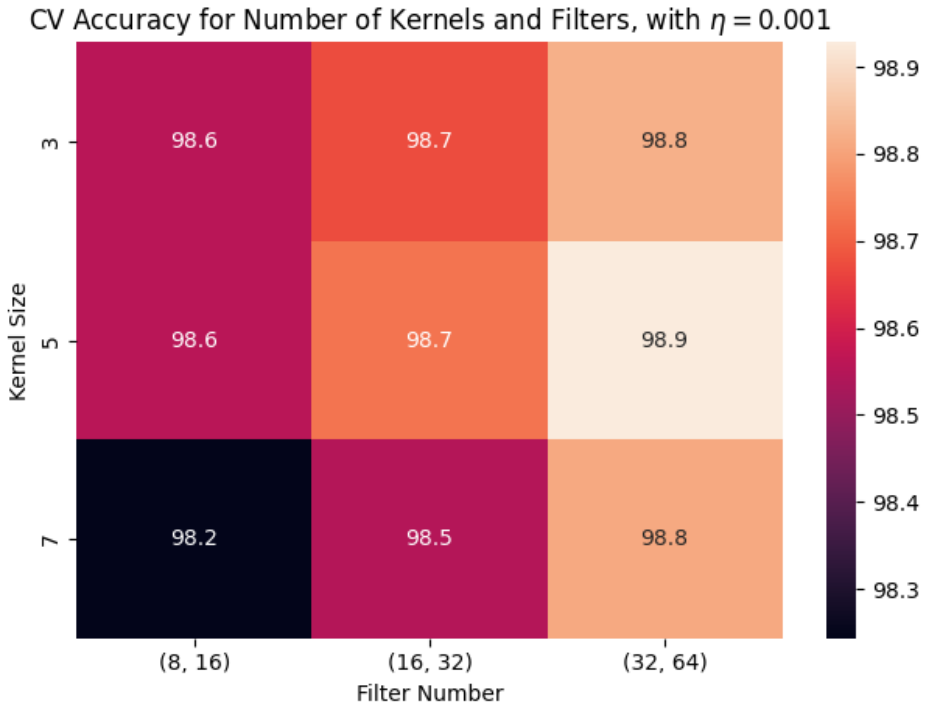

FIG. 5: Results of a a grid search comparing different kernel sizes and numbers of filters in each of two convolutional layers in an MNIST classifier. The learning rate is 0.001.

3. *Pooling and Padding*

Our grid search with pooling and convolutional layer padding is illustrated in Figure 7. It shows that adding padding had a minimal impact on the cross-validated accuracy of the MNIST classifier. This is logical, as the MNIST dataset is digitally centered, with the edges of each image conveying neglegible useful information about the digit pictured. This means that the main utility of padding, helping the model better capture information on the edge of inputs, may not have been impactful. Additionally, our initial configuration of adding pooling layers with a kernel size of 2 after each of the two convolutional layers has the highest accuracy of 98.9%.

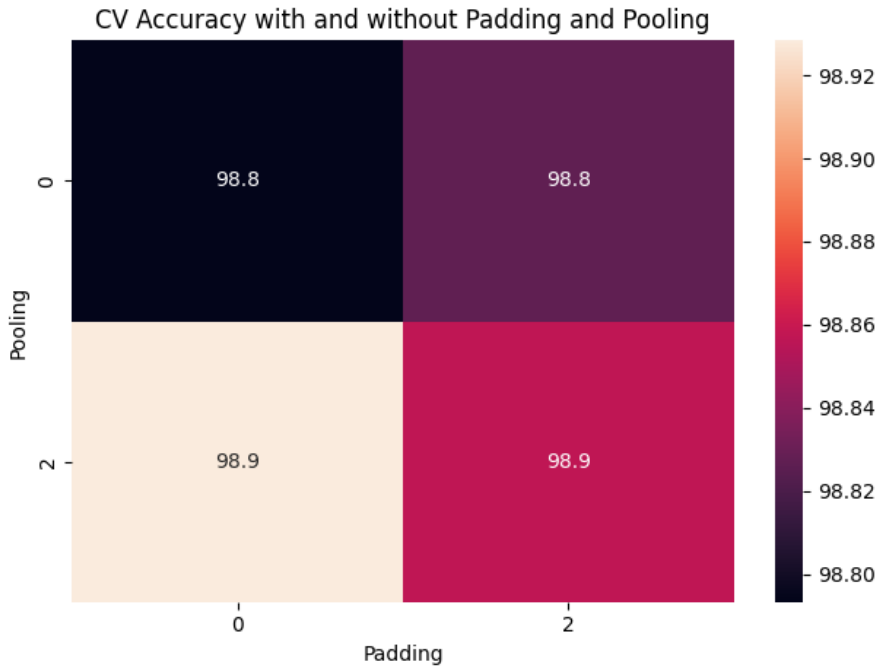

FIG. 6: Heatmap reasults of Pooling and Padding grid search with two convolutional layers followed by three linear layers. The Y-axis represents pooling kernel size (0 means no pooling), with pooling layers placed after each convolutional layer, and the X-axis represents the convolutional layers' padding.

In III we see that the addition of padding increases runtimes drastically, while pooling conversely decreases it significantly. Considering the positive and negligible effects on model accuracy of pooling and padding respectively, we identify no padding, and pooling with a kernel size of 2 after each convolutional layer as the ideal configuration.

TABLE III: Runtimes of cross-validation grid search of convolutional layer padding and subsequent pooling layers.

	No Padding	**Padding: 2**
No Pooling	28 minutes 58.43 seconds	55 minutes 3.76 seconds
Pooling: 2	7 minutes 50.80 seconds	19 minutes 17.78 seconds

4. *Dropout Rates and Activation Functions*

Figure 7 shows that the accuracy did not improve when using dropout or Leaky ReLU. With 10 epochs of training, we observed slightly lower accuracies compared to the previous configuration using ReLU and no dropout (98.9%, as shown in Figure 5). This suggests either that the model did not overfit significantly, or that these techniques were ineffective in reducing overfitting while maintaining model performance. The model without dropout or Leaky ReLU achieves 98.86% accuracy on the test set compared to 99.66% on the training set. This suggests that the overfitting may not be incredibly severe, which somewhat explains dropout and Leaky ReLU not significantly improving model performance. We suspect that the techniques' negative impact on accuracy may be due to the MNIST dataset being relatively simple and thus less prone to overfitting. Possibly, if the models were trained for more than 10 epochs, avoiding overfitting would become more important and the aforementioned techniques could become more beneficial. Based on our selection criteria, we decided to keep the original configuration with ReLU as the activation function and no dropout layers.

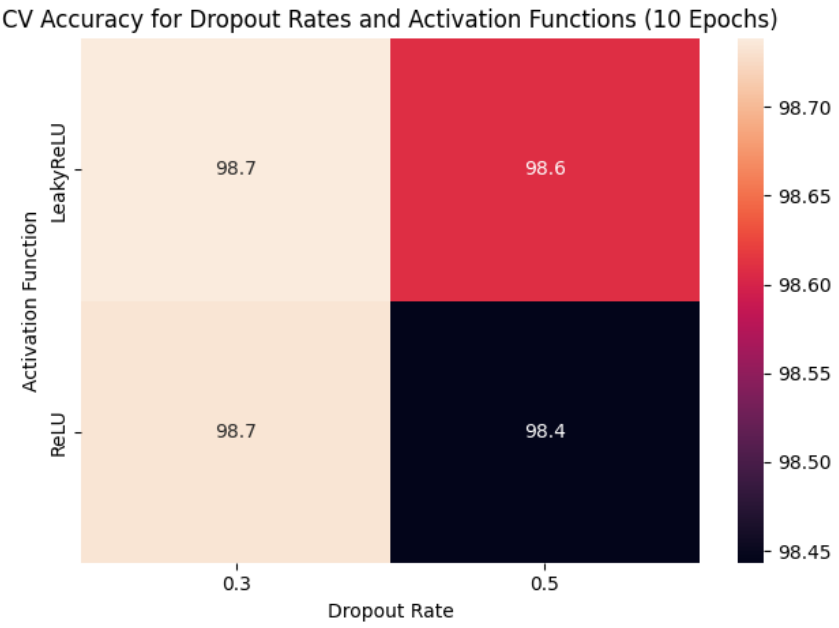

FIG. 7: Results of cross-validation grid search of MNIST classifier convolutional network with dropout rates and different activation functions.

C. Results: Final Assessment

Figure 8 illustrates that the performance of Logistic Regression varies depending on the digit class. The difference in accuracy between the lowest-performing class, class 8, and the highest-performing classes, class 0 and class 1, is about 10%. This variability is likely because logistic regression was originally designed for binary classification, and handling multiclass classification may be too complex for this model. The total accuracy on the test set without bootstrap is 91.99%, a small drop from the 92.72% accuracy on the training set. For the bootstrapped accuracy, Figure 11 in Appendix V shows a 95% confidence interval of [91.45, 92.54], reflecting that the performance of Logistic Regression varies between digit classes, and thus between the different bootstrapped samples.

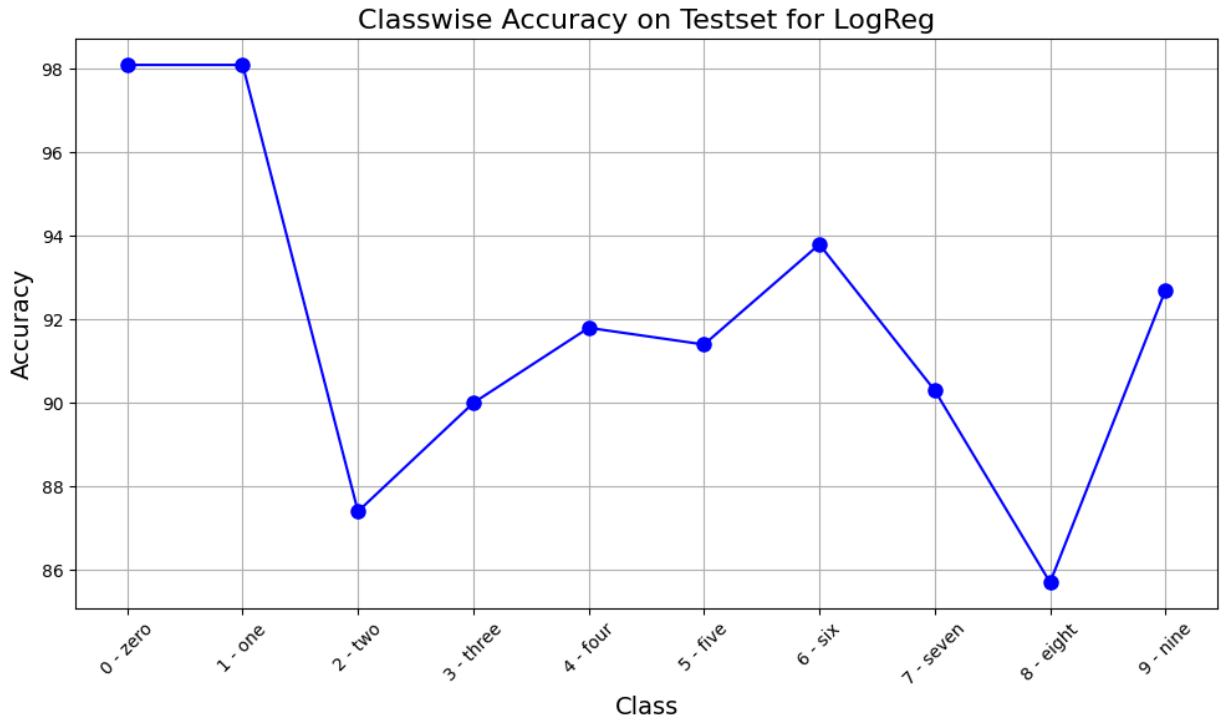

FIG. 8: Total accuracy of Logistic Regression on each class of number in the MNSIT dataset.

In contrast, Figure 9 shows that the performance of our best CNN on each digit class is constistently high. The classwise performance drop is only about 2%, ranging from the worst performance of 97.4% on class 3 to the best performance of 99.6% on class 1. One could suspect that the variation in performance is due to the model seeing more examples of the digit 1, as MNIST does not have a totally uniform distribution of classes, but in fact an

overrepresentation of the digit 1 [19]. Ones and zeros may also simply be easier to recognize for the model. The stable performance compared to Logistic Regression confirms CNNs' ability to effectively handle the classification of images. The accuracy on the whole test set is 98.86%, a less than 1% drop from the 99.66% accuracy on the training set.

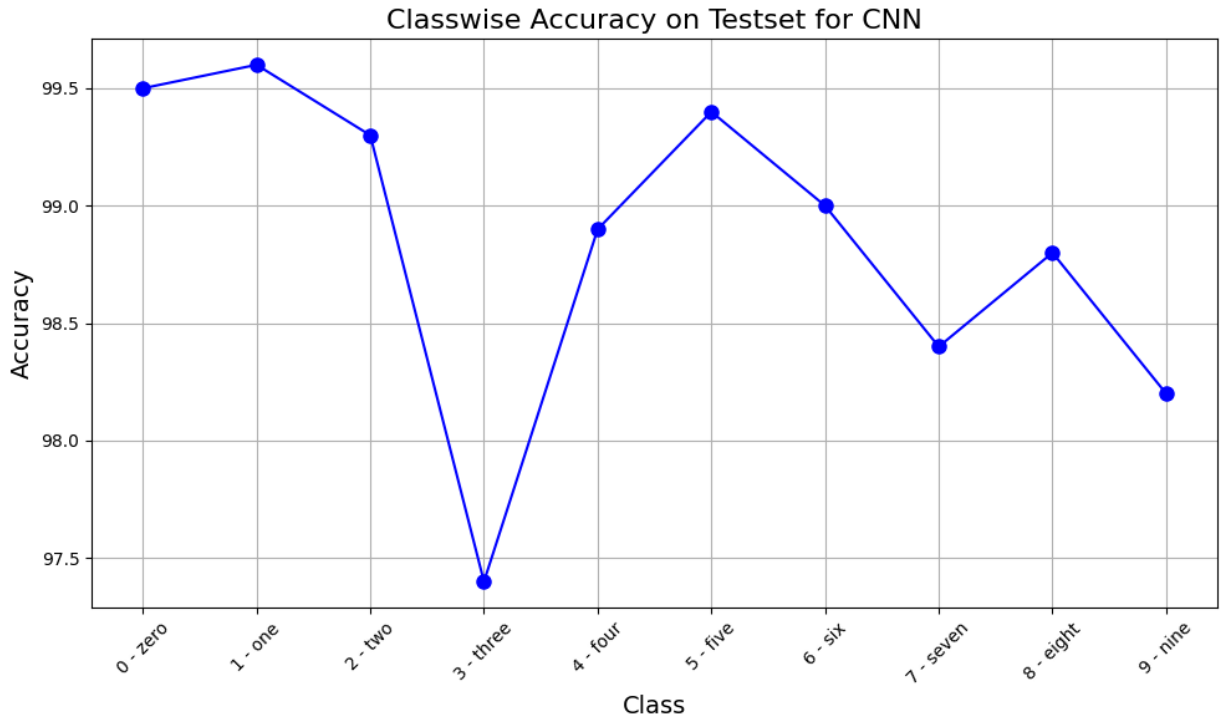

FIG. 9: Accuracy of CNN after grid search optimization, in classifying each digit in the MNIST dataset.

Our bootstrapped accuracy in Figure 10 shows that, for most of the bootstrapped test sets, the model performs close to the overall test set accuracy of 98.86%, with a 95% confidence interval of [98.66, 99.06]. This is significantly narrower than the confidence interval of the Logistic Regression model, reflecting that the CNN is more stable across the different digit classes. Our confidence interval indicates that even when the nature of our test set varies, the performance of our model does not change dramatically. This is a positive sign for deploying our model in a production setting with new data. Figure 10 also shows that although the distribution of the bootstrapped accuracy is not exactly skewed, it is still spiky even though we ran 1000 bootstraps. This unexpected pattern is more likely due to the nature of our dataset rather than a problem with the model itself, as we observe some of the same effects with our baseline model (see Figure 11 in Appendix V). Constructing a confidence interval by taking the percentiles of the boostrapped estimates assumes that the bootstrap distribution is symmetric and smooth, so there is a possibility that our confidence interval may be to optimistic [20].

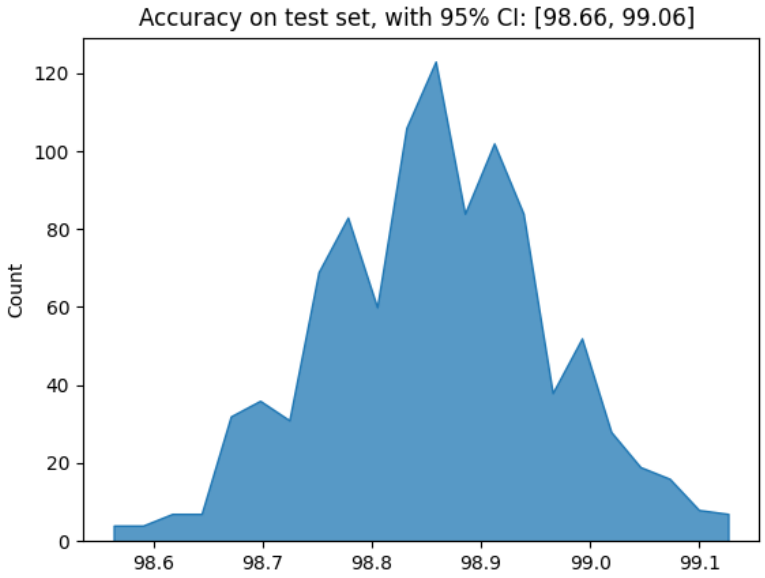

FIG. 10: Distribution optimized CNN MNIST classifier's performance on bootstrapped MNIST dataset, with 1000 bootstraps.

D. Discussion

As a benchmark, the best accuracy achieved on the MNIST dataset is 99.6% [21]. In contrast, we reached an accuracy of only 98.9% despite starting with good initial values. This demonstrates that extensive tuning is essential to develop highly accurate convolutional neural networks (CNNs). A key limitation in our approach was the high sensitivity of our results to the initial tuning values. In a neural network, all components are interrelated, which makes it challenging to isolate the effects of individual parameters.

Earlier in this course, we spent considerable time examining how factors such as learning rates, optimizers, and the number of epochs influence convergence and model performance. In this project, however, we focused on exploring hyperparameters specific to CNNs, such as filter numbers, kernel size, dropout, padding, and pooling. As a result, we devoted less attention to tuning basic parameters, which may have impacted our ability to achieve optimal performance.

Both the baseline logistic regression model and the CNN reached beyond 90% accuracy, without extensive tuning of the baseline model. This raises the question of whether using convolutional neural networks is worth the additional complexity, increased runtime, and reduced interpretability. However, in certain applications similar to classifying digits in MNIST, such as reading bank account numbers from handwritten digits, even tiny errors can be very costly. Therefore, it is crucial to minimize these errors as much as possible [6].

For tasks like digit recognition on the MNIST dataset, our results indicate that CNNs are worth the effort they require due to their superior ability to capture spatial hierarchies in image data. However, this is not to say that Logistic Regression is useless for tasks like this. In certain cases, Logistic Regression may still be a viable option, especially for simpler applications or when computational resources are limited.

V. CONCLUSION

In this study, we compared the performance of Convolutional Neural Networks (CNNs) and Logistic Regression for digit classification on the MNIST dataset. Through grid search and hyperparameter optimization, we found that CNNs outperformed Logistic Regression, achieving an average accuracy of 98.9%, compared to 92.0% for Logistic Regression. These results highlight the importance of nonlinear feature extraction in image classification tasks, with CNNs being essential for applications that demand near-perfect accuracy. Key insights include the usefulness of pooling in reducing computational costs, the lackluster impact of padding when inputs contain little edge information, as well as the minimal impact of dropout layers and activation functions designed to reduce overfitting in these specific experimental conditions. However, the complexity of CNNs and their reduced interpretability present challenges. The findings also underscore the trade-off between computational efficiency and model complexity. This study shows that achieving optimal performance when tuning networks with multiple parameters, while at the same ensuring that the process is reproducible, requires effective methods. While it is clear that CNNs outperform Logistic Regression on image recognition, future work could explore more efficient tuning methods, such as Bayesian optimization, to streamline the tuning process and further improve performance.

APPENDIX A

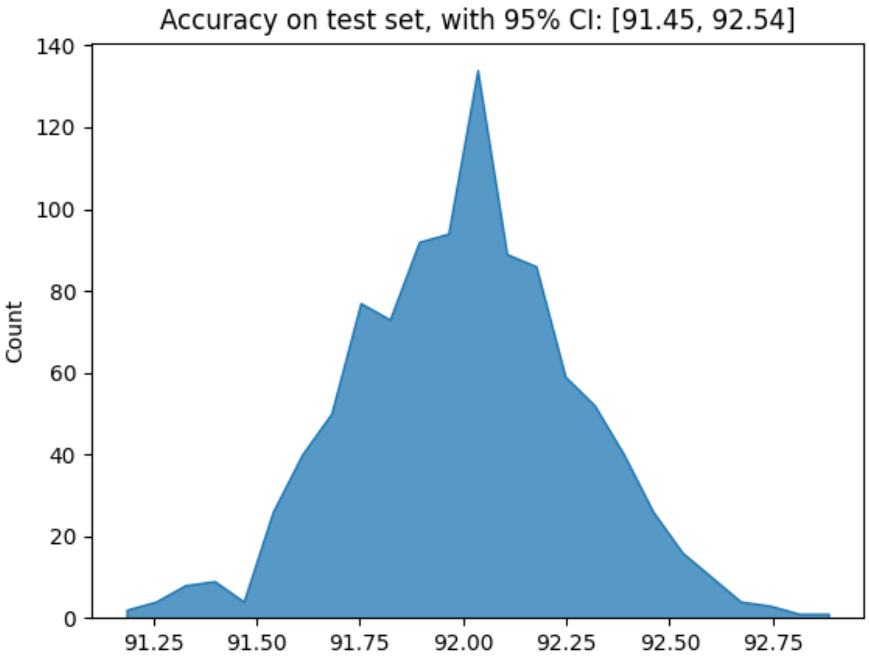

FIG. 11: Our best LogRegs performance on 1000 boostrapped test set.

REFERENCES

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. 60(6):84–90, 2017. Originally published at NIPS 2012.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. pages 770–778, 2016.

[4] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer, 2009.

[5] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[6] Sebastian Raschka, Yuxi (Hayden) Liu, and Vahid Mirjalili. *Machine Learning with PyTorch and Scikit-Learn: Develop machine learning and deep learning models with Python*. Packt Publishing, 2022.

[7] Morten Hjorth-Jensen. Machine learning lecture notes. https://github.com/CompPhysics/MachineLearning/tree/master/doc/pub, 2023. GitHub repository.

[8] Philipp Zimmermann. Most commonly used activation functions. https://hacking-and-security.cc/most-commonly-used-activation-functions/, 01 2024.

[9] Aji Gautama Putrada, Nur Alamsyah, Muhamad Fauzan, and Syafrial Fachri Pane. Ns-svm: Bolstering chicken egg harvesting prediction with normalization and standardization. *JUITA : Jurnal Informatika*, 11:11, 05 2023.

[10] Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani, Robert Tibshirani, Trevor Hastie, and Daniela Witten. *An Introduction to Statistical Learning: with Applications in R*, volume 103. Springer Nature, 1st edition, 2013.

[11] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style,

high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, pages 8024–8035, 2019.

[12] PyTorch Tutorials. Training a classifier. https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html. Accessed: 2024-12-07.

[13] Anthropic. Claude 3.5 sonnet. https://www.anthropic.com/news/claude-3-5-sonnet. Accessed: 2024-12-07.

[14] Iris Bore. FYS-STK4155 Project 1. https://github.com/irisbore/FYS-STK4155_project1, 2023.

[15] OpenAI. Chatgpt: Openai language model. https://openai.com/chatgpt, 2023. Accessed: 2024-09-22, 27-09-22.

[16] GitHub. GitHub Copilot. https://github.com/features/copilot, 2023. Accessed: October 2, 2024.

[17] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.

[18] John D Hunter. MatPlotLib: a 2D Graphics environment. *Computing in Science and Engineering*, 9(3):90–95, 1 2007.

[19] Richard Corrado. Mnist digit classification. https://richcorrado.github.io/MNIST_Digits-overview.html?

[20] A. C. Davison and D. V. Hinkley. *Bibliography*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1997.

[21] Patrice Y. Simard, David Steinkraus, and John C. Platt. Best practices for convolutional neural networks applied to visual document analysis. *Proceedings of the Seventh International Conference on Document Analysis and Recognition*, pages 958–963, 2003.