# MNIST Digit Recognizer Report

**Zane Lai**
11094794
The University of British Columbia
Department of Electrical and Computer Engineering
zaneyujunlai@outlook.com

## Abstract

This project explores handwritten digit recognition using both traditional machine learning and deep learning approaches. We first applied models such as Random Forest and Logistic Regression to the MNIST dataset, achieving over 90% accuracy on test set. However, these models performed poorly on real-world handwritten samples (below 50% accuracy). To address this, we implemented a Convolutional Neural Network (CNN), which also achieved over 90% accuracy on the MNIST test set and 84% on our custom-collected real-world handwritten dataset. The results demonstrate the superior ability of deep learning in handling diverse handwriting styles.

## 1 Introduction

The objective of this project is to develop a handwritten digit recognizer that can be used in the real world.

Handwritten digit recognition is a classic problem in machine learning with wide applications. The MNIST dataset, a benchmark dataset consisting of 28×28 grayscale images of handwritten digits (0–9), has long served as a standard dataset for evaluating and training classification models in this domain.

In this project, we used the MNIST dataset called **MNIST Digit Recognizer** on the Kaggle website (`https://www.kaggle.com/datasets/animatronbot/mnist-digit-recognizer`) [1]

We aimed to explore and compare two different ways to digit recognition problem: machine learning and deep learning techniques. Initially, we applied machine learning models such as Random Forest and Logistic Regression on the MNIST dataset. Although these models achieved high accuracy on the standard test set, their performance dropped significantly when tested on a custom handwritten digit dataset we collected by ourselves.

To overcome the limitations of machine learning models in handling real-world handwriting variability, we implemented a Convolutional Neural Network (CNN) deep learning model inspired by a VGG-style architecture [2]. CNNs are particularly effective for image-based tasks due to their ability to capture spatial features through convolutional operations.

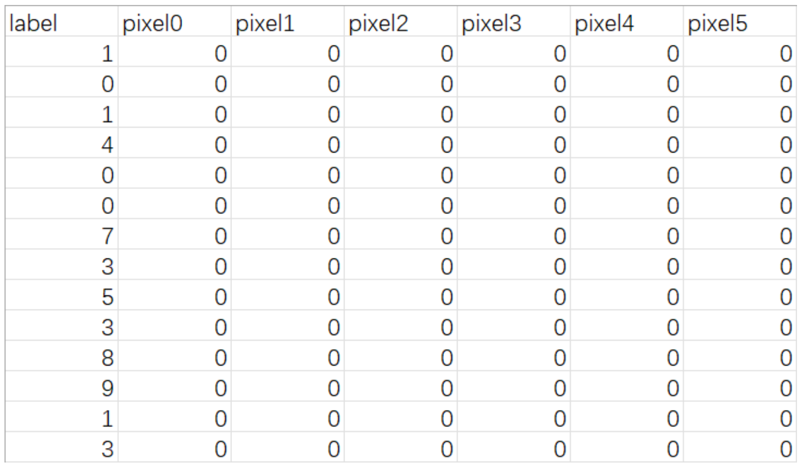
## 2 Data Preprocessing

In the MNIST training dataset, each row represents one image, and each column represents one pixel in that image.

Each image is $28 \times 28$ pixels, so there are 784 pixels in total, for example:

label, pixel1, pixel2, ..., pixel784

The pixel values are grayscale, ranging from 0 (white) to 255 (black).



| label | pixel0 | pixel1 | pixel2 | pixel3 | pixel4 | pixel5 |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 |

Figure 1: MNIST data in train.csv

However, if the range of input data values is too large (such as here, 0–255), training can become unstable, and the gradients may either vanish or explode. To address this, we apply normalization: dividing by 255 scales the pixel values from [0, 255] to [0, 1], which improves training stability, accelerates convergence, and helps the model learn more effectively.

$$X_{\text{norm}} = \frac{X}{255}$$

where $X$ is the original data range of 0-255, $X_{\text{norm}}$ is the data after normalization range of 0-1.

Then, we split the data into 80% training set and 20% testing set for the follow-up validation.

## 2.1 Machine Learning Data Preprocessing

For machine learning, we performed standardization to ensure different dimensions are in a "fair" state, and making it easier to converge.

$$X_{\text{scaled}} = \frac{X_{\text{norm}} - \mu}{\sigma}$$

where $X_{\text{scaled}}$ is the data after scaling, $\mu$ is the mean of the training data, $\sigma$ is the standard deviation of the training data.

However, we have a total of $28 \times 28 = 784$ dimensions, and such a high number of dimensions can easily lead to overfitting if used directly for training. Therefore, we applied **Principal component analysis** (PCA) for dimensionality reduction [3].

In our experiments, we selected the number of principal components using the **elbow method**. By plotting the cumulative explained variance against the number of components, we identified the "knee point" where the curve begins to stabilize. This point indicates that adding more components beyond this number yields minimal additional explained variance.

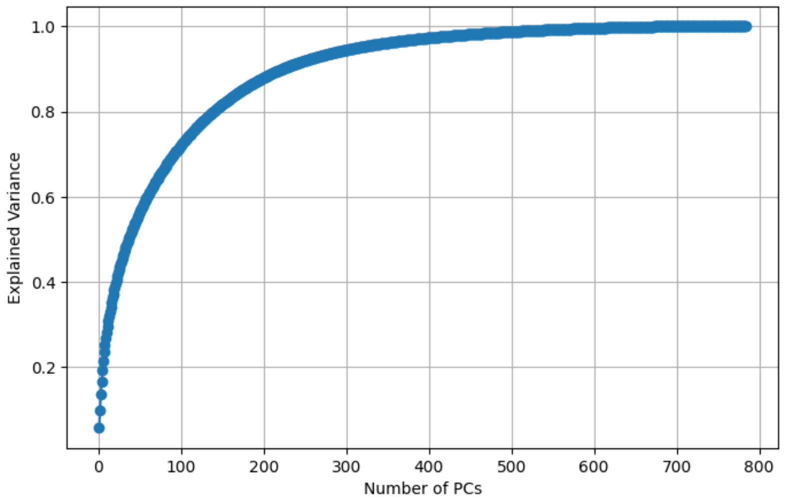


Figure 2: Explained variance vs. number of components

As shown in Figure 2, the curve flattens out beyond approximately 200 components, indicating that additional components contribute little new information. Conversely, when the number of components falls below 100, the explained variance drops rapidly, resulting in significant information loss. Therefore, selecting 180 components, as the knee point, offers a balanced trade-off between dimensionality and information reserve.

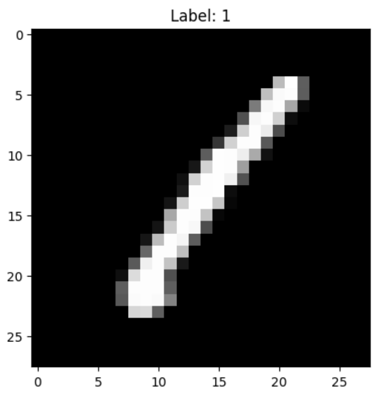


Figure 3: Example image

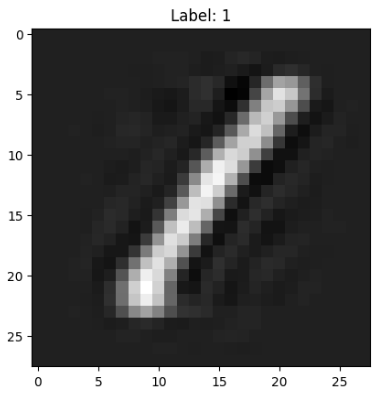


Figure 4: Reconstruction result after PCA

As shown in Figures 3 and 4, the PCA reconstruction retains most of the original features, indicating that the current preprocessed data are suitable for machine learning tasks.

## 2.2 Deep Learning (CNN) Data Preprocessing

For the convolutional neural network (CNN), the preprocessing is simpler compared to traditional machine learning models. Since CNNs are capable of learning spatial features directly from image, there is no need for feature engineering or dimensionality reduction.

After normalization, each image is reshaped from flattened vectors of shape [784] into a 2D image tensors of shape [1,28,28], where 1 indicates the number of channels (grayscale images). This format is required by PyTorch's convolutional layers, unlike the machine learning models, we do not apply standardization or PCA, as CNNs can learn appropriate feature representations directly through convolutional layers.

## 3 Model Training

We divided our model training process into two main approaches: traditional machine learning and deep learning. In the machine learning track, we employed two models: Random Forest (RF) and Logistic Regression (LogReg). For the deep learning track, we implemented a Convolutional Neural Network (CNN) based on the VGG architecture.

### 3.1 Machine Learning Models Training

#### 3.1.1 Random Forest (RF) Training

For the Random Forest classifier, we explored the effect of different numbers of decision trees on model performance. Specifically, we trained the model with the number of trees set to 10, 100, 500, and 1000, respectively. For each setting, the model was trained on the training dataset and tested on the test dataset.

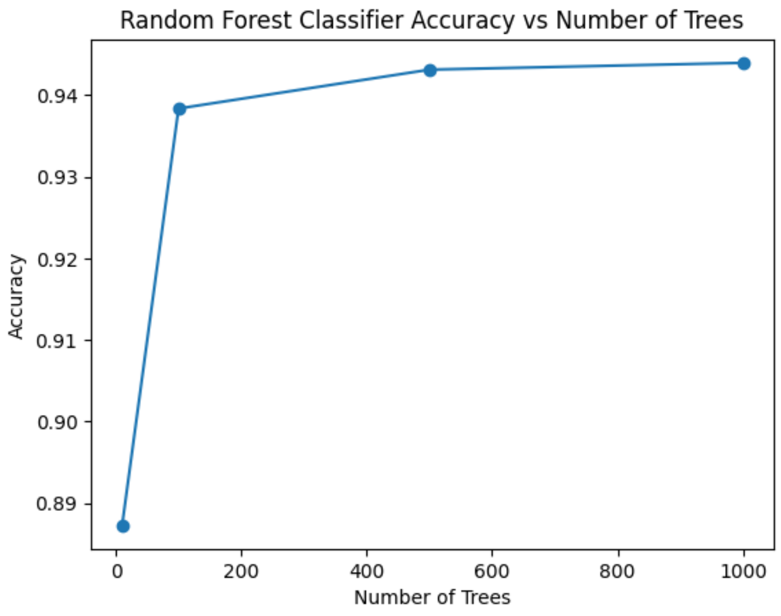


Figure 5: Random Forest Classifier Accuracy vs Number of Trees

As shown in the Figure 5, increasing the number of trees significantly improves accuracy up to around 0-100 trees. Beyond this point, the improvement becomes insignificant. Therefore, using 100 trees provides a good trade-off between accuracy and computational efficiency.

#### 3.1.2 Logistic Regression (LogReg) Training

For the Logistic Regression (LogReg) model, we followed a similar approach to that used in Random Forest training. We evaluated the effect of the inverse regularization strength parameter $C$. We tested four values: $C = [0.01, 0.1, 1, 10]$. For each setting, the model was trained on the training set and tested on the test dataset.

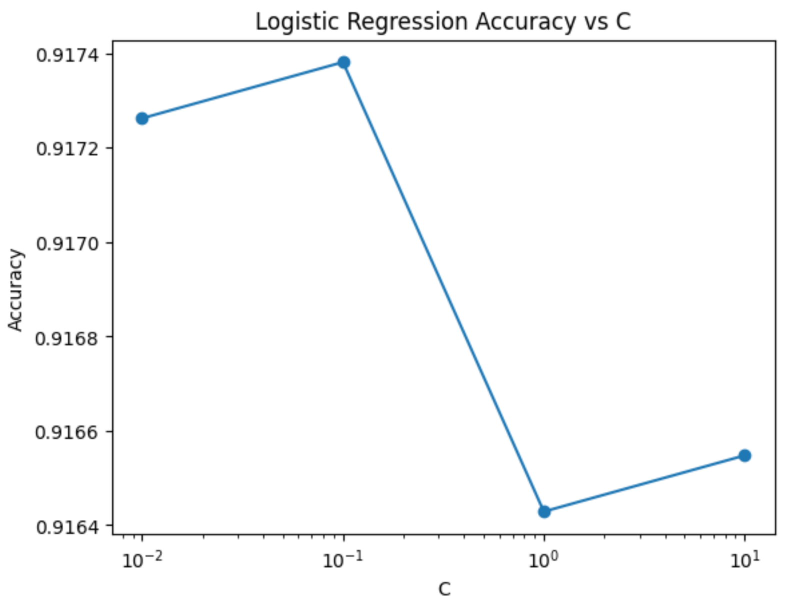


Figure 6: Logistic Regression Accuracy vs Cs

The Figure 6 shows that the best performance was achieved when $C = 0.1$. As $C$ increase (i.e., weaker regularization), the model tends to overfit, losing accuracy. On the other hand, smaller $C$ can lead to underfitting. Therefore, using $C = 0.1$ provides a good trade-off between bias and variance.

### 3.2 Deep Learning Model Training (CNN)

The CNN model is based on the first two blocks of the VGG architecture and consists of four convolutional layers with increasing feature map sizes ($32 \rightarrow 64 \rightarrow 128 \rightarrow 256$).

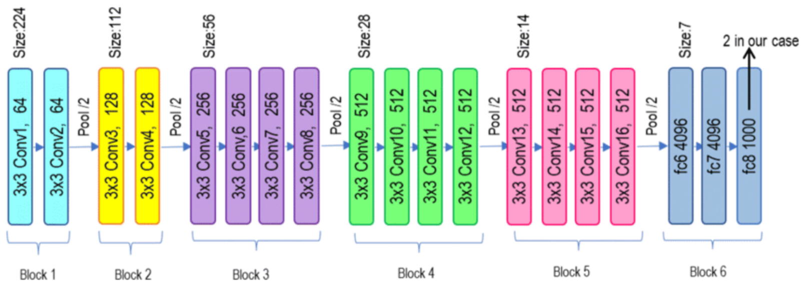


Figure 7: VGG-Net Architecture

Credit:`https://www.researchgate.net/figure/VGG-19-Architecture-39-VGG-19-has-16-convolution-layers` `fig5_359771670`

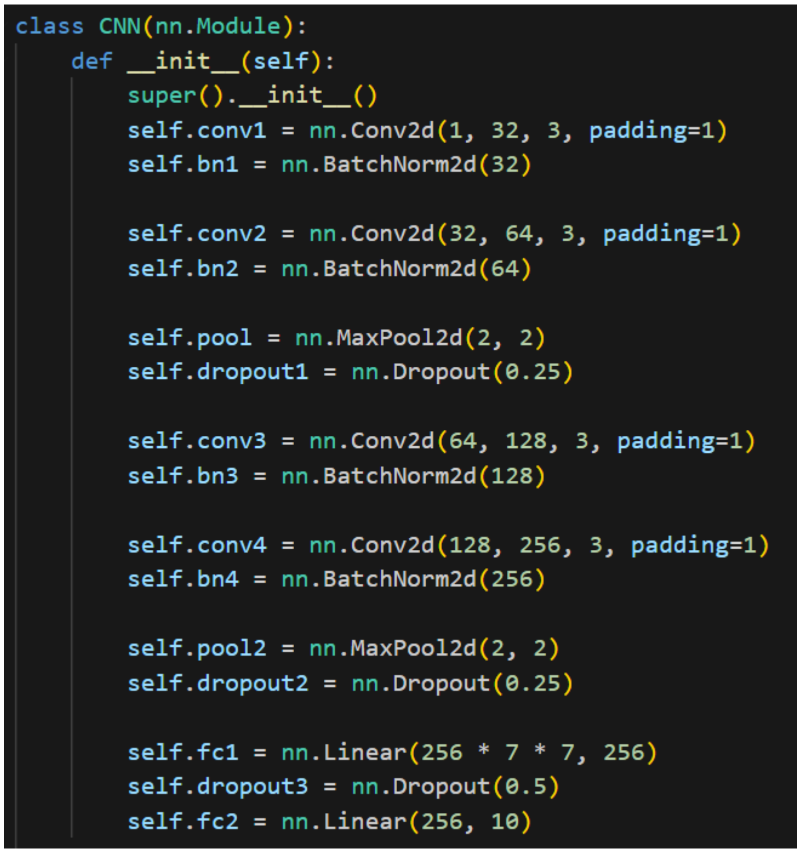
In the Figure 7, the traditional VGG-Net architecture, the input images are of size 224×224, and the model gradually reduces its size. This design is suited for large, complex images with high variability.

However, in our case, the input images are from the MNIST dataset, which consists of the images of digits with a much smaller size (28×28). Applying the full traditional VGG architecture directly would reduce the spatial features too much, leading to lose important information. Therefore, we made the following adjustments:

1. Simplified the architecture to include only the first two blocks of VGG.

2. Increased the feature map sizes gradually ($32 \rightarrow 64 \rightarrow 128 \rightarrow 256$).

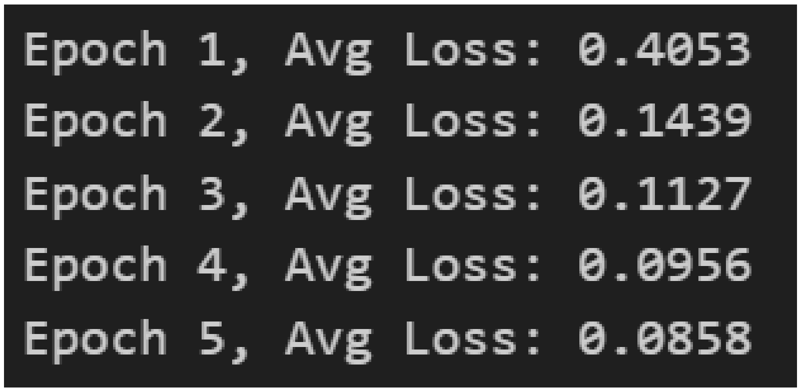
3. Padding is applied to preserve the spatial dimensions of the image, maintaining the original size of 28×28



```
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)

        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)

        self.pool = nn.MaxPool2d(2, 2)
        self.dropout1 = nn.Dropout(0.25)

        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)

        self.conv4 = nn.Conv2d(128, 256, 3, padding=1)
        self.bn4 = nn.BatchNorm2d(256)

        self.pool2 = nn.MaxPool2d(2, 2)
        self.dropout2 = nn.Dropout(0.25)

        self.fc1 = nn.Linear(256 * 7 * 7, 256)
        self.dropout3 = nn.Dropout(0.5)
        self.fc2 = nn.Linear(256, 10)
```

Figure 8: Adjusted VGG-Net Architecture

It was trained using the Adam optimizer with a learning rate of 0.001. The training process was conducted over 5 epochs. For each small batch, the gradients were first cleared using *optimizer.zero_grad()*, then the input was fed through the CNN to generate predictions. The negative log-likelihood loss (nll_loss) was computed, backpropagated, and the model parameters were updated accordingly using *optimizer.step()*. After each epoch, the average loss across all batches was calculated and printed to monitor the training progress [4].



```
Epoch 1, Avg Loss: 0.4053
Epoch 2, Avg Loss: 0.1439
Epoch 3, Avg Loss: 0.1127
Epoch 4, Avg Loss: 0.0956
Epoch 5, Avg Loss: 0.0858
```

Figure 9: Average negative log-likelihood loss for each epoch

## 4 Evaluating

### 4.1 Machine Learning Models Evaluating

We initially applied traditional machine learning models, Random Forest and Logistic Regression, to our MNIST test dataset. Then evaluated on the standard test set, both models achieved high accuracy: **93.83%** for Random Forest and **91.74%** for Logistic Regression. At first glance, these results indicate that traditional machine models are sufficient for digital recognition.

However, our project aimed to build a recognizer capable of handling real-world handwritten inputs. To evaluate model robustness, we developed a simple digit input app using Tkinter, allowing users to draw digits manually [5].

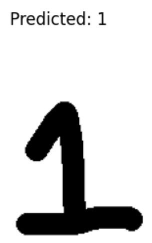


Figure 10: A correctly classified digit "1" drawn using the Tkinter input interface

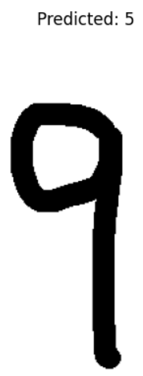
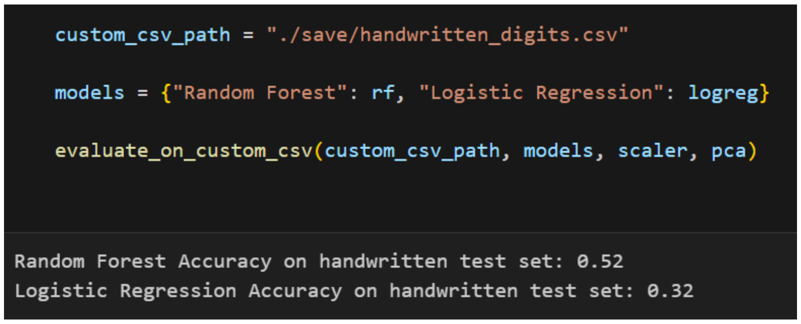


Figure 11: An example of model misclassification on a real handwritten digit (Tkinter input interface)

When tested on these real handwritten inputs, as shown in Figure 11, the performance of both models dropped significantly. To further validate this observation, I created a small custom test set *handwritten_digits.csv* using *LoopDigitCollector.py*, consisting of real handwriting samples. The results were unbelievable: the Random Forest model achieved only **52%** accuracy, while Logistic Regression dropped to **32%**.



```
custom_csv_path = "./save/handwritten_digits.csv"

models = {"Random Forest": rf, "Logistic Regression": logreg}

evaluate_on_custom_csv(custom_csv_path, models, scaler, pca)
```

```
Random Forest Accuracy on handwritten test set: 0.52
Logistic Regression Accuracy on handwritten test set: 0.32
```

Figure 12: Evaluate through my own handwritten digits

These findings show a fundamental limitation: although traditional machine learning models perform well on clean and standardized datasets like MNIST, they lack the robustness needed to handle real-world data, especially when affected by noise and variations in input quality.

### 4.2 Deep Learning Models Evaluating (CNN)

we also evaluated CNN's performance on both the standard test set and our custom handwritten test set. The model achieved an accuracy of **98.54%** on the standard MNIST test set, and **84.0%** on the handwritten test.

## 5 Explanation and Future Work

We explored the performance of both traditional machine learning models and CNNs on the task of handwritten digit recognition. Our findings show a clear performance gap between the two approaches: while Random Forest and Logistic Regression performed well on the clean, standardized MNIST dataset, their performance dropped significantly on real-world handwritten input.

In contrast, the CNN demonstrated strong generalization ability, achieving an accuracy of 84.0% on our custom handwritten test set, compared to 52% and 32% for Random Forest and Logistic Regression, respectively.

This result can be attributed to the fundamental differences in how the models interpret image data. Traditional machine learning models treat images as flat arrays, focusing on individual pixel values while ignoring the spatial relationships between them. Hence, these models are unable to capture the structure and layout of the image as a whole. However, in the real world, handwriting varies greatly in size, position, and style. Therefore, CNNs are better for such tasks, as they are designed to learn hierarchical spatial features directly from the image data.

For the future work, the traditional machine learning models can use techniques such as n-fold cross validation to better select hyperparameters and avoid overfitting or underfitting.

In the CNN model, the architecture used in this project was a lightweight version inspired by only the first two blocks of the VGG network as mentioned. Expanding the depth or width of the network by adding more convolutional layers could improve performance further.

## References

[1] "Kaggle: your machine learning and data science community." `https://www.kaggle.com/`

[2] S. Bangar, "VGG-Net Architecture Explained - Siddhesh Bangar - Medium," *Medium*, Jun. 29, 2022. [Online]. Available: `https://medium.com/@siddheshb008/vgg-net-architecture-explained-71179310050f`

[3] K. Parte, "Dimensionality reduction: principal component analysis," *Medium*, Feb. 21, 2024. [Online]. Available: `https://medium.com/analytics-vidhya/dimensionality-reduction-principal-component-analysis-d1402b58feb1`

[4] "NLLLoss — PyTorch 2.6 documentation." *Python Documentation*`https://pytorch.org/docs/stable/generated/torch.nn.NLLLoss.html`

[5] "tkinter — Python interface to Tcl/Tk," *Python Documentation*. `https://docs.python.org/3/library/tkinter.html`

[6] OpenAI, "ChatGPT," OpenAI, San Francisco, CA, USA. [Online]. Available: `https://chat.openai.com/`