Project Title: Classification of American Sign Language Images using Neural Networks and Image Compression

Team Members : Jerin Easo Thomas, Shreya Varghese, Ismail Shaikh

ABSTRACT

The project's goal is to create machine learning models that use neural networks to classify American Sign Language (ASL) movements, improving the deaf and hard-of-hearing community's digital accessibility. The project tackles the problem of comprehending a language that mostly relies on visual signals, such as hand forms and body movements, by utilizing neural networks.The research also explores image compression methods like PCA and SVD.

INTRODUCTION

The "Classification Of American Sign Language Images Using Neural Networks" project is an innovative attempt to close a huge communication gap between the fast developing digital world and the deaf and hard-of-hearing community. ASL, or American Sign Language, is an essential means of communication for millions of Americans. One significant obstacle, nevertheless, is the absence of reliable technology solutions for ASL interpretation in digital environments.

Motivation

The principal driving force behind this project is the aim to improve accessibility and inclusivity in digital communication. As the use of digital platforms for social contact, work, and education grows, it is critical to make sure that these platforms are accessible to all users, including those who use ASL for communication. This project intends to create a machine learning model that

correctly classifies ASL motions in order to set the foundation for real-time ASL-to-text or speech translation technologies, which will facilitate seamless communication.

This research is also motivated by the understanding that the deaf and hard-of-hearing community frequently encounters particular difficulties in gaining access to digital services and content. ASL recognition software that works well has the potential to change many facets of life, from making daily encounters easier to granting everyone equitable access to education and employment.

Why It's Beneficial: The benefits of this project are multifold:

Why It's Advantageous This project has several advantages:

- Enhanced Accessibility: We can greatly increase the deaf and hard-of-hearing community's access to digital platforms by providing accurate ASL recognition. This improvement is in line with the larger social objectives of equality and inclusivity.
- Opportunities for Education and Employment: As ASL users gain more proficiency in the language, they may be able to pursue additional educational and employment opportunities. Better communication in the workplace, online learning environments, and classrooms could result from it, giving deaf and hard-of-hearing people equal opportunity to learn and work.
- Cultural Recognition and Inclusivity: One step towards recognising the culture and identity of the deaf population is the acknowledgment and integration of ASL in digital communication technology. It conveys a powerful message of inclusivity and respect.
- Technological Advancement: This project makes contributions to image recognition and machine learning as well. The methods and strategies created can be used in different fields of image analysis and gesture-based communication.
- Research Implications: The results of this study may have an impact on linguistics, computer vision, and artificial intelligence research in the future by offering insightful information on the intricacies of ASL.

Related Work:

- M. Manoj Krishna et al.'s paper "Image Classification using Deep Learning" in the International Journal of Engineering & Technology (2018) investigates the use of the AlexNet convolutional neural network for picture classification. This study is relevant to our work since it shows how deep learning works well for processing photos, especially when it comes to classifying images from the ImageNet database. This paper's technique and findings highlight the potential of similar neural network architectures for

challenging picture classification problems, which is helpful for our effort on American Sign Language image classification.

- A comprehensive investigation into image classification using deep neural networks (DNNs) within the TensorFlow framework is presented in the 2019 International Journal of Engineering Research and Technology paper titled "A Study on Image Classification Based on Deep Learning and TensorFlow." Leading the team from Universiti Kuala Lumpur, Mohd Azlan Abu, the authors concentrated on classifying photos of five distinct kinds of flowers. They accomplished over 90% accuracy in their classifications, showcasing the effectiveness of DNNs in managing particular image identification jobs.Important insights into the use of deep learning for specialized picture categorization can be gained from the technique and achievements presented in the study.

Method:

a. Image Classification :

A fundamental task in computer vision is image classification, which entails classifying images according to predetermined criteria. With the development of deep learning, particularly convolutional neural networks (CNNs), this process—which is essential to many applications including facial recognition and medical imaging—has been transformed. Image classification is the process of giving an image a label from a set of predetermined categories using algorithms. In this procedure, characteristics from the image, such as colors, forms, textures, and patterns, are frequently extracted and analyzed.

Image classification is extremely relevant to projects like ASL gesture recognition. It is the process of identifying and classifying ASL signals from pictures or video frames, where each gesture has a designated meaning or alphabet. Research and development in the field of assistive technology and beyond is heavily dependent on the accuracy and efficiency of image categorization algorithms, as they have a direct impact on the efficacy of such applications.

b. Image Compression

A key method in digital image processing is image compression, which lowers the size of an image file to enable more effective transmission and storage. It covers the two primary methods of compression: lossy and lossless. For high-precision applications like medical imaging, lossless compression minimises file size without compromising image quality. On the other hand, lossy

compression, which is usually used for web photos and streaming video, achieves a more substantial size reduction by discarding some image information.

Image compression is essential for effectively managing big datasets in the context of picture classification projects, such as ASL gesture identification. When dealing with large-scale image datasets or real-time applications, compressing images can result in faster processing times and less storage capacity due to the reduction of the quantity of data that needs to be processed and stored.

c. Neural Networks

Machine learning heavily relies on neural networks, which are computational models modeled after the human brain. They are made up of layers upon layers of networked nodes, or neurons, that process information. An input layer that receives data, hidden layers that do computations, and output layer that provides the final result, constitute as the essential parts of the neural network. Through the use of training methods like backpropagation, neural networks are able to learn by modifying the weights of connections between nodes in response to input data.

Because of their resilience and efficiency in managing challenging tasks like picture categorization, neural networks are the preferred method for this project. This project is especially well suited for neural networks because of its remarkable strengths in pattern recognition, adaptability, effective image processing, data-driven learning, and scalability. Their proficiency in pattern recognition is essential for deciphering the visual patterns found in American Sign Language (ASL) visuals. They can correctly categorize a broad variety of complex ASL due to their adaptability.

i. Tensor Flow

The main reason TensorFlow is used is because of its strong deep learning and neural network computing capabilities. With its scalability and versatility, TensorFlow is a well-known open-source toolkit that performs exceptionally well on challenging tasks like picture categorization. Its comprehensive support for deep neural networks and its easy-to-use high-level APIs, such as Keras, facilitate quick model creation and experimentation, which are essential for correctly categorising ASL motions. TensorFlow is also a great option for the project's image classification and compression tasks due to its quick processing speed of huge picture datasets. TensorFlow's abundance of pre-trained models and community resources help to expedite the development process, which increases the project's effectiveness in delivering accurate and effective image analysis.

d. Analysis:

i. Image Compression using PCA & SVD

PCA and SVD are the two techniques that were used for image compression.

```
# Converting to NumPy array and flattening the images
X_train_flat = X_train.to_numpy().reshape(-1, 28*28)
# Standardizing the data
scaler = StandardScaler()
X_train_flat_standardized = scaler.fit_transform(X_train_flat)
# Applying PCA with a variance 0.98
pca = PCA(0.98)
X_train_pca = pca.fit_transform(X_train_flat_standardized)

X_train_reconstructed_standardized = pca.inverse_transform(X_train_pca)
X_train_reconstructed = scaler.inverse_transform(X_train_reconstructed_standardized)

original_size = X_train_flat.size
compressed_size = X_train_pca.size
compression_ratio = original_size / compressed_size
print(f"Compression Ratio: {compression_ratio:.2f}")
print("Number of components selected by PCA:", pca.n_components_)
y = train_data.iloc[:, 0]

fig, axes = plt.subplots(2, 7, figsize=(10, 2))
for i in range(7):
    ax = axes[0, i]
    ax.imshow(X_train.iloc[i].to_numpy().reshape(28, 28), cmap='gray')
    ax.axis('off')
    ax.set_title(f"Label: {y.iloc[i]}", fontsize=8)
    ax = axes[1, i]
    ax.imshow(X_train_reconstructed[i].reshape(28, 28), cmap='gray')
    ax.axis('off')
plt.show()
```

Fig A. Code Snippet for Image compression using PCA

The first step in applying Principal Component Analysis (PCA) for picture compression is to use NumPy to flatten and convert the X_train dataset's images into 1D arrays. Next, using StandardScaler, the data is standardized to have a mean of 0 and a standard deviation of 1. This standardized data is transformed using the fit_transform method by PCA, which keeps 98% of its variance. To recreate the images with reduced dimensionality, the compressed data is inversely transformed. The compression ratio is computed to measure the effectiveness of this procedure. Lastly, a visualization of the original and rebuilt photos is presented. This procedure demonstrates how PCA can effectively reduce data size without sacrificing important information.

```
# Converting to NumPy array and flattening the images
X_train_flat = X_train.to_numpy().reshape(-1, 28*28)
# Standardizing the data
scaler = StandardScaler()
X_train_flat_standardized = scaler.fit_transform(X_train_flat)
# Applying SVD with 290 components
U, S, VT = np.linalg.svd(X_train_flat_standardized, full_matrices=False)
S = np.diag(S)
n_components = 290

X_reduced = np.dot(U[:, :n_components], S[:n_components, :n_components])
X_reconstructed_standardized = np.dot(X_reduced, VT[:n_components, :])
X_reconstructed = scaler.inverse_transform(X_reconstructed_standardized).reshape(-1, 28, 28)

original_size = X_train_flat.size
compressed_size = (U[:, :n_components].size + S[:n_components, :n_components].size + VT[:n_components, :].size)
compression_ratio = original_size / compressed_size
print(f"Compression Ratio: {compression_ratio:.2f}")
y = train_data.iloc[:, 0]

fig, axes = plt.subplots(2, 7, figsize=(10, 2))
for i in range(7):
    ax = axes[0, i]
    ax.imshow(X_train.iloc[i].to_numpy().reshape(28, 28), cmap='gray')
    ax.axis('off')
    ax.set_title(f"Label: {y.iloc[i]}", fontsize=8)
    ax = axes[1, i]
    ax.imshow(X_reconstructed[i], cmap='gray')
    ax.axis('off')
plt.show()
```

Fig B. Code Snippet for Image compression using SVD

The Fig B shows how to use Singular Value Decomposition (SVD) to compress images. The `X_train} dataset images are first flattened into one-dimensional arrays and then standardized using `StandardScaler`. After that, this data goes through SVD, which separates it into the matrices {U}, {S}, and {VT} while keeping 290 principal components. Matrix multiplication is used to compute the reduced data, which is subsequently reconstructed in its original space. The effectiveness of this compression is measured by the compression ratio, which is computed by contrasting the sizes of the original and compressed files. The original image and the reconstructed images are visualized side by side offering a clear comparison to evaluate the effect of compression on image quality.This approach demonstrates how well SVD may reduce image size without sacrificing important details.

ii. Deep Neural Network

```
# Reshape the images for DNN
flat_input_shape = (28 * 28,)  # Flatten the 28x28 images
training_images_fnn = training_images.reshape((training_images.shape[0], 28 * 28))
validation_images_fnn = validation_images.reshape((validation_images.shape[0], 28 * 28))

# Define the DNN model
def build_dnn_model(input_shape):
    model = Sequential([
        Flatten(input_shape=input_shape),
        Dense(512, activation='relu'),
        Dense(256, activation='relu'),
        Dense(128, activation='relu'),
        Dense(26, activation='softmax')
    ])

    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

    return model
```

Fig C. Code Snippet for Deep Neural Network

An artificial neural network with several layers between the input and output layers is called a Deep Neural Network (DNN). DNNs are made to learn through several layers of abstraction in order to mimic intricate patterns in data. Their effectiveness is especially noticeable in applications like picture recognition that call for the identification of complex structures within huge datasets.The model architecture starts with reshaping of input images as DNN processing requires that input images be reshaped from a 28x28 pixel format to a flat, one-dimensional array of 784 elements. The model was constructed with TensorFlow's Sequential API and consists of three dense layers with 512, 256, and 128 neurons each. The 'relu' activation function is used in these levels to capture nonlinear complexities. The Flatten layer receives the inputs from the flattened images. The 26 neurons that make up the last dense layer have a "softmax" activation, making them appropriate for multi-class classification—presumably for tasks like letter recognition. Utilizing the 'adam' optimizer and 'sparse_categorical_crossentropy' loss function, together with accuracy tracking as a metric, this DNN model is skillfully designed to gradually extract and decipher features from the input data, ultimately resulting in efficient multi-class classification.

iii. CNN

```
# Creating the model
def CNN_model():

    CNN_model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=((28, 28, 1))),
        tf.keras.layers.MaxPool2D(2,2),
        tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
        tf.keras.layers.MaxPool2D(2,2),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(512, activation='relu'),
        tf.keras.layers.Dense(256, activation='relu'),
        tf.keras.layers.Dense(26, activation='softmax')
    ])

    CNN_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

    return CNN_model
```

Fig D. Code Snippet for CNN

Convolutional neural networks (CNNs) are a subclass of deep neural networks that are mostly utilized for processing data that has a topology resembling a grid, such photographs. With the application of appropriate filters, CNNs are very good at capturing spatial and temporal connections in a picture. Because of this capability, they are very well suited for applications such as image classification, where it is essential to recognise and maintain patterns in the pixel data. The presented code describes a Convolutional Neural Network (CNN) model for extracting complicated patterns from input images. It consists of two convolutional layers, each having 32 filters of size 3x3 and 'relu' activation. Max pooling layers with a 2x2 window come next, which minimize overfitting and reduce feature map dimensions while assisting in feature extraction. Next, using 'relu' activation, the model uses a flattening layer to transform 2D feature maps into a 1D vector, which is then fed into two dense layers of 512 and 256 neurons,

respectively. The last layer produces a probability distribution across 26 classes and is built for a multi-class classification problem. It has 26 neurons with "softmax" activation.

iv. VggNet

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

def VGGNet_model(input_shape=(28, 28, 1)):
    model = Sequential([
        # First block
        Conv2D(64, (3, 3), padding='same', activation='relu', input_shape=input_shape),
        Conv2D(64, (3, 3), padding='same', activation='relu'),
        MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
        # Second block
        Conv2D(128, (3, 3), padding='same', activation='relu'),
        Conv2D(128, (3, 3), padding='same', activation='relu'),
        MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
        # Third block
        Conv2D(256, (3, 3), padding='same', activation='relu'),
        Conv2D(256, (3, 3), padding='same', activation='relu'),
        MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
        # Classifier
        Flatten(),
        Dense(512, activation='relu'),
        Dense(216, activation='relu'),
        Dense(26, activation='softmax')  # Assuming 26 classes as per your notebook
    ])
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model
```

Fig D. Code Snippet for VGGNet

A customised version of the well-known deep convolutional neural network architecture, the VGGNet model is defined in the given code and is well-known for its efficiency in computer vision applications. The first three blocks are sequential and consist of convolutional layers with 'relu' activation and increasing filter sizes (64, 128 and 256) paired with 'same' padding. Max pooling layers are then added to minimise the spatial dimensions. The purpose of these blocks is to gradually extract more intricate attributes from the given images. The architecture then moves onto a classifier part, which is made up of dense layers of 512 and 216 neurons each using 'relu' activation, and a Flatten layer. 'Softmax' activation and 26 neurons make up the final output layer, which supports multi-class classification scenarios. Equipped with the 'adam' optimizer and the'sparse_categorical_crossentropy' loss function, this VGGNet model effectively learns from spatial hierarchies in the image data, making it a good fit for complex image classification problems.

v. ResNet

```python
# Define the residual block
def residual_block(x, filters, kernel_size=3, strides=1):
    y = Conv2D(filters, kernel_size=kernel_size, strides=strides, padding='same')(x)
    y = BatchNormalization()(y)
    y = ReLU()(y)
    y = Conv2D(filters, kernel_size=kernel_size, padding='same')(y)
    y = BatchNormalization()(y)
    # Shortcut connection
    if strides != 1 or x.shape[-1] != filters:
        x = Conv2D(filters, kernel_size=1, strides=strides, padding='same')(x)
        x = BatchNormalization()(x)
    # Add the shortcut value to the output
    y = Add()([x, y])
    y = ReLU()(y)
    return y
# Build the ResNet model
def build_resnet_model(input_shape, num_classes):
    inputs = Input(shape=input_shape)
    # Initial convolution layer
    x = Conv2D(64, kernel_size=7, strides=2, padding='same')(inputs)
    x = BatchNormalization()(x)
    x = ReLU()(x)
    # Residual blocks
    x = residual_block(x, filters=64)
    x = residual_block(x, filters=64)
    x = residual_block(x, filters=128, strides=2)
    x = residual_block(x, filters=128)
    x = residual_block(x, filters=256, strides=2)
    x = residual_block(x, filters=256)
    # Global average pooling and dense layer
    x = GlobalAveragePooling2D()(x)
    x = Dense(num_classes, activation='softmax')(x)
    model = tf.keras.Model(inputs=inputs, outputs=x)
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model
```

Fig B. Code Snippet for ResNet

The above code constructs a model known as a ResNet (Residual Network), a convolutional neural network architecture that is well-known for using residual blocks to train deep networks effectively. Each block is made up of convolutional layers that have been activated via ReLU and batch normalisation, together with shortcut connections that help gradients go across the network more easily. These residual blocks are stacked, and occasionally stride changes are made to decrease spatial dimensions as the number of filters increases gradually. The model begins with an initial convolution layer, moves through a number of residual blocks, and ends with a dense softmax layer that corresponds to the number of classes and global average pooling. Because it avoids the vanishing gradient problem that is common in standard deep networks and has a deep structure, this ResNet model—compiled with the 'adam' optimizer and'sparse_categorical_crossentropy' loss—is very skilled at complicated picture classification tasks.

vi. Ensemble

```python
from tensorflow.keras.layers import Input

# Function to create the ensemble model with CNN, ResNet, and VGGNet
def ensemble_model_v4(cnn_model, resnet_model, vggnet_model):
    # Removing the output layer of the CNN model
    cnn_output_layer = cnn_model.layers[-2].output
    # Removing the output layer of the ResNet model
    resnet_output_layer = resnet_model.layers[-2].output
    # Removing the output layer of the VGGNet model
    vggnet_output_layer = vggnet_model.layers[-2].output
    # Concatenating the output of the CNN, ResNet, and VGGNet models
    merged_layer = concatenate([cnn_output_layer, resnet_output_layer, vggnet_output_layer])
    # Adding a dense layer for final classification
    final_output = Dense(26, activation='softmax')(merged_layer)
    # Creating the ensemble model
    ensemble = tf.keras.models.Model(inputs=[cnn_model.input, resnet_model.input, vggnet_model.input], outputs=final_output)
    # Compiling the ensemble model
    ensemble.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return ensemble
```

Fig B. Code Snippet for Ensemble Model

The code that is provided utilises an ensemble model that combines the strengths of CNN, ResNet, and VGGNet to produce a more reliable and accurate solution for classification challenges. In order to combine their high-level feature representations, it first eliminates the output layers from each pre-trained model and then concatenates their penultimate layers. A new dense classification layer with a'softmax' activation function that is designed for multi-class classification with 26 output classes comes after this merged layer. This ensemble model, which is compiled using the 'adam' optimizer and the 'sparse_categorical_crossentropy' loss function and assessed using accuracy metrics, combines the strengths of ResNet's deep learning efficiency, CNN's feature extraction, and VGGNet's pattern recognition to improve overall performance and show the value of combining various learning algorithms in challenging tasks like image classification.

Experimental Setup

We conducted the experiment on Google Colab, as Colab is a hosted Jupyter Notebook service that requires no setup to use and provides free access to computing resources, including GPUs. As the dataset had 34,627 instances, we used the T4GPU provided by Colab environment. It was also due to the limitation provided by Colab for the free version, and this was the best performing GPU available for free tier environments on Google Colab. Its seamless integration with Google Drive as a storage environment for Model parameters was also a plus point for us. The implementation was done in Python 3.10, leveraging a number of key libraries for scientific computing and machine learning. For neural network capabilities, TensorFlow in its latest version was used to construct and train the models. Data analysis and manipulation relied on NumPy and Pandas, also the latest releases, for their array and data frame structures respectively. Visualizations were generated using Matplotlib, again the most recent build. Lastly, graph network capabilities were enabled by integrating the latest releases

of Graphviz, Pydot. Overall, utilizing the latest versions of these software tools and libraries ensured full access to the state-of-the-art functionalities in each area. The dataset leveraged in this work is the American Sign Language (ASL) alphabet dataset. Structurally, it closely resembles the classic MNIST digit dataset, with black and white 28x28 pixel images representing the hand shapes forming each letter from A to Z in ASL. Each image is encoded as a flattened 784-dimensional vector with integer pixel intensities spanning 0 to 255. The full dataset contains 34,627 images, which has been partitioned into a training set of 27,455 samples and a test set of 7,172 samples for performance evaluation. This data enables training image classification models to recognize these linguistic hand gestures, a task formulation analogous to the MNIST digit classification benchmark. As data augmentation is essential to prevent overfitting in computer vision tasks, we leveraged the ImageDataGenerator class from Keras to perform real-time data augmentation on the images during training. It takes the training images and labels as input and yields randomly transformed image batches, thereby expanding the effective size of the training data. Specifically, we used random rotations up to 40 degrees, width/height shifts up to 20%, shear ranges up to 20%, and zoom ranges up to 20% as the transformation operations. Horizontal flipping was also employed. For the validation set, only rescaling the pixel intensities was used without any other distortions. These configurable generators enabled the incorporation of robust data augmentation into the model training process through a simple, clean interface. All models used the Adam optimizer for training parameters and Sparse Categorical Crossentropy as the loss function to optimize. The output layer universally consisted of 26 units with Softmax activation to predict across the 26 classes. The DNN stacked flattened inputs into Dense layers of 512, 256, and 128 units. The CNN leveraged Convolutional layers, max pooling, flattening and Dense layers. VGGNet and ResNet exploited Convolutional blocks with max pooling but ResNet also incorporated custom residual connections. Finally, the Ensemble concatenated the penultimate layers of CNN, ResNet, and VGGNet models with a classification layer on top of the merged features. Despite architectural differences, the models standardized on consistent optimization, loss, and output layer configurations.

```
[ ]  # Creating an ImageDataGenerator for Image Augmentation
     def img_generator(training_images, training_labels, validation_images, validation_labels):

         train_datagen = ImageDataGenerator(
                             rescale = 1./255,
                             rotation_range=40,
                             width_shift_range=0.2,
                             height_shift_range=0.2,
                             shear_range=0.2,
                             zoom_range=0.2,
                             horizontal_flip=True,
                             fill_mode='nearest')

         train_generator = train_datagen.flow(x=training_images, y=training_labels, batch_size=32)

         valid_datagen = ImageDataGenerator(rescale=1./255)

         valid_generator = valid_datagen.flow(x=validation_images, y=validation_labels, batch_size=32)

         return train_generator, valid_generator
```

Fig. ImageGenerator implementation with parameters defined

Results

This part contains all the project specifics in depth. Since our project also compares the best image classification models, first the best hyper-parameters for CNN and DNN are compared and common top performing hyper-parameters are selected. Then, these optimal hyper-parameters are taken into account for the VGGnet, ResNet, and Ensemble models.

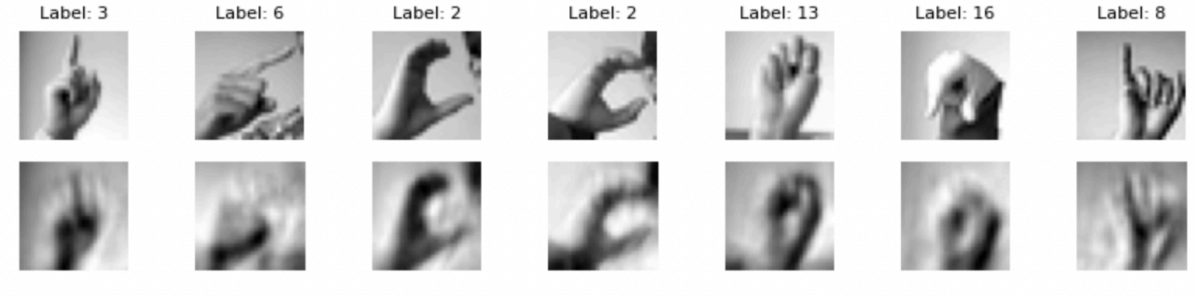
a. Image Compression using PCA & SVD:

Image compression was also explored using SVD and PCA. As seen in the figures below, PCA was implemented with different variance values. For a lower variance value, the compression ratio is high but the image quality is highly compromised. As we keep increasing the variance values we see that the compression ratio also increases and the number of components selected by PCA has also increased. The quality of the image keeps getting better as well. For a variance of 0.98 we can see that the compressed images have better quality and have a close resemblance with the original images, the compression ratio here is low. The number of PCA components selected for 0.98 variance was 290 components. Since PCA and SVD are related in regards to underlying mathematics we gave the same number of components to SVD as well, i.e 290 components. We can observe that the compression ratio for both SVD and PCA is almost the same. For optimal picture compression, the variance for PCA and the number of components for SVD must be properly chosen. In addition to influencing the

rate of compression and the quality of the reconstructed images, this choice also influences the computational effectiveness and usefulness of the compressed data in a variety of applications.

i. For variance : 0.86

```
Compression Ratio: 20.63
Number of components selected by PCA: 38
```

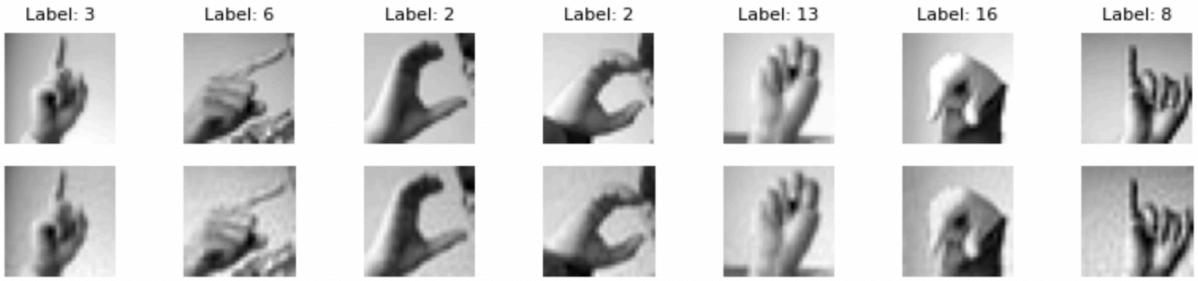


ii. For variance : 0.92

```
Compression Ratio: 10.45
Number of components selected by PCA: 75
```

iii. For variance : 0.98

```
Compression Ratio: 2.70
Number of components selected by PCA: 290
```



iv. For SVD with 290 components

```
Compression Ratio: 2.60
```

b. Deep Neural Network(DNN)

A TensorFlow backend coupled with Keras is used to create the Deep Neural Network (DNN) that is being presented for image classification tasks. Using the Rectified Linear Unit (ReLU) activation function, the model consists of three hidden layers with 512, 256, and 128 neurons each. The input layer flattens 28x28 pictures. To achieve multi-class classification, the output layer, which consists of 26 neurons, uses softmax activation. Training accuracy increases from 15.96% to 95.79% and validation accuracy from 18.59% to 79.80% over the course of 20 epochs as seen in the Fig A, indicating a significant improvement in the model. With the model and dataset size that are provided, the training duration of 4.38 minutes is appropriate. The test accuracy is 79.80% when evaluated on the validation set. For upcoming use, the model is preserved. The thorough examination of both training and validation accuracy, in addition to the declining loss, highlights the model's ability to identify the underlying patterns in the dataset.

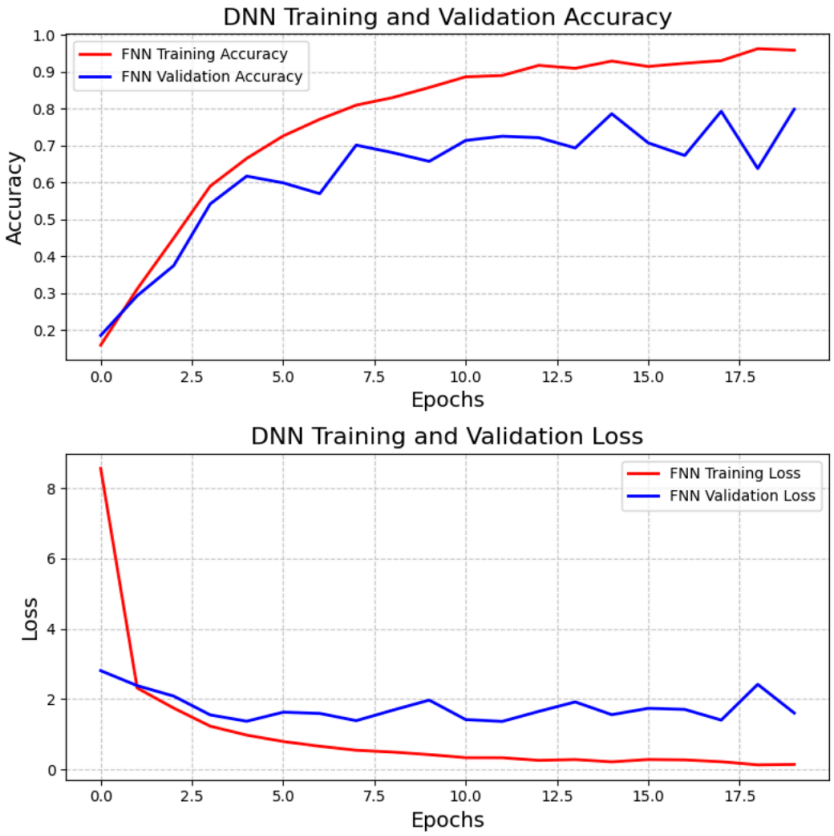


Fig A. DNN Training and Validation Accuracy

c. Convolution Neural Network(CNN)

Specifically designed for image classification tasks, the Convolutional Neural Network (CNN) model that is given consists of two convolutional layers, each followed by max-pooling and then densely linked layers. An important aspect of hierarchical feature extraction is the convolutional layers' use of 32 filters with ReLU activation. After 20 training iterations, the model's accuracy shows a notable increase from 20.17% to an amazing 89.04%, and from 47.27% to an astounding 92.16% for validation as shown in the Fig B. The model demonstrates effective learning and generalization as seen by the simultaneous decrease in training and validation loss. 13.39 minutes for the entire training period is in line with what was expected given the size of the dataset and the model. The model's effectiveness is demonstrated by the test accuracy of 92.16% obtained via evaluation on the validation set. For later use, the CNN model is stored. Furthermore, the learning curves are shown in the visualization below, which sheds light on the model's convergence and performance dynamics over time.

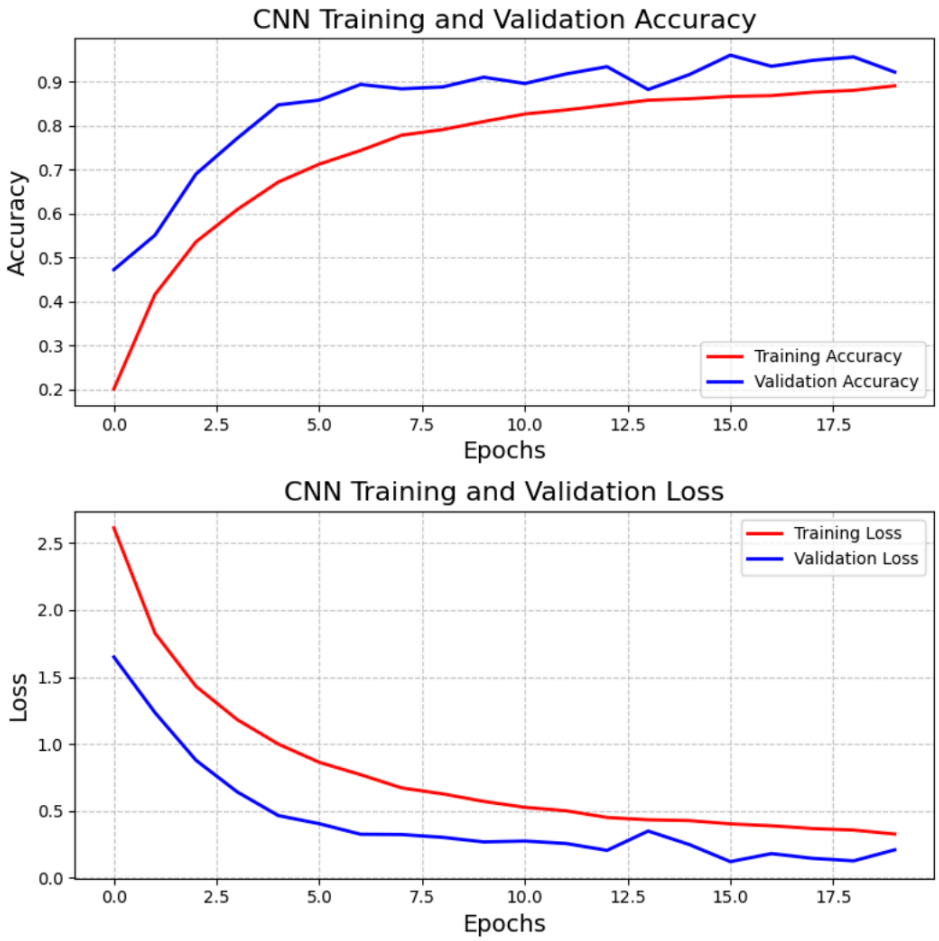


Fig B. CNN Training and Validation Accuracy

d. Visual Geometry Group Network (VGGnet)

The research makes use of a deep convolutional neural network architecture called Visual Geometry Group Network (VGGNet), which is intended for image classification. The model consists of three convolutional blocks with two convolutional layers with progressively larger filter sizes and max-pooling afterward. A densely connected layer classifier section completes the model. The model shows extraordinary accuracy during 5 training epochs, going from a starting 73.72% to a perfect 100% on the training set and from 88.26% to an amazing 96.21% on the validation set. The resulting loss also considerably decreases, highlighting the network's efficient learning. Training takes about 46.6 minutes, and evaluation on the validation set results in an accuracy of 96.21% and a loss of about 0.1788. An appropriate save of the VGGNet model is made for later use. Extensive illustrations of the model's learning dynamics and convergence are provided by the visualization in Fig C that show the training and validation accuracy and the training and validation loss over epochs.

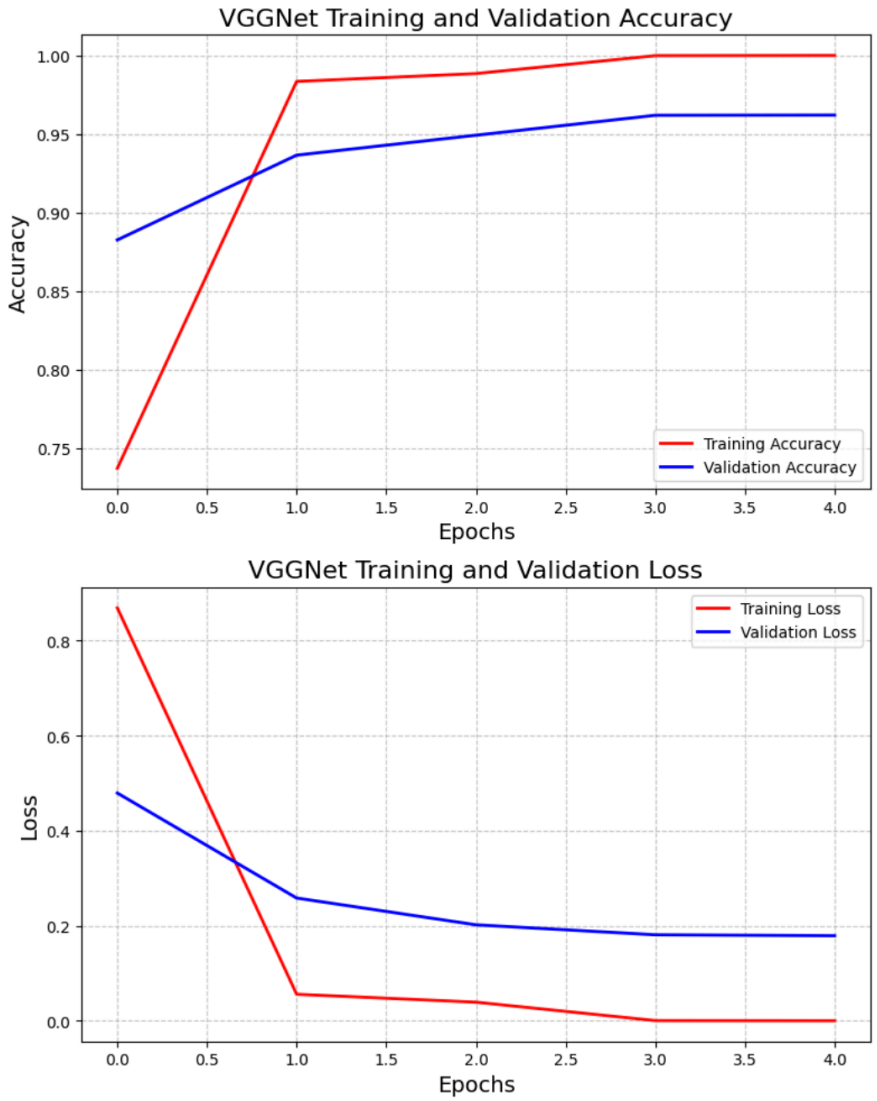


Fig C. VGGNet Training and Validation Accurancy

e. Residual Network (ResNet**)**

Using the dataset, the ResNet architecture—which has residual blocks for effective deep network training—was used for image classification. To solve the vanishing gradient issue, residual blocks are incorporated into the model, which was first started with convolution layers. Finally, a dense layer and global average pooling complete the classification architecture. The model demonstrates proficiency on the training dataset by achieving a high training accuracy of 93.46% over the 5-epoch training. Strong generalization to fresh data is indicated by the validation accuracy, which notably reaches an astounding 94.99% as seen in Fig D. Effective learning is indicated by low loss values during training, and the 45.5-minute training duration overall validates the computational

expenditure. A validation set evaluation of the model yields a test accuracy of 94.99%, confirming its reliability. The effectiveness and generalizability of the model are ensured by the visualization of training and validation metrics, which offers insights into the convergence dynamics of the model. In conclusion, the ResNet model has high learning and generalization performance and works well for image classification.

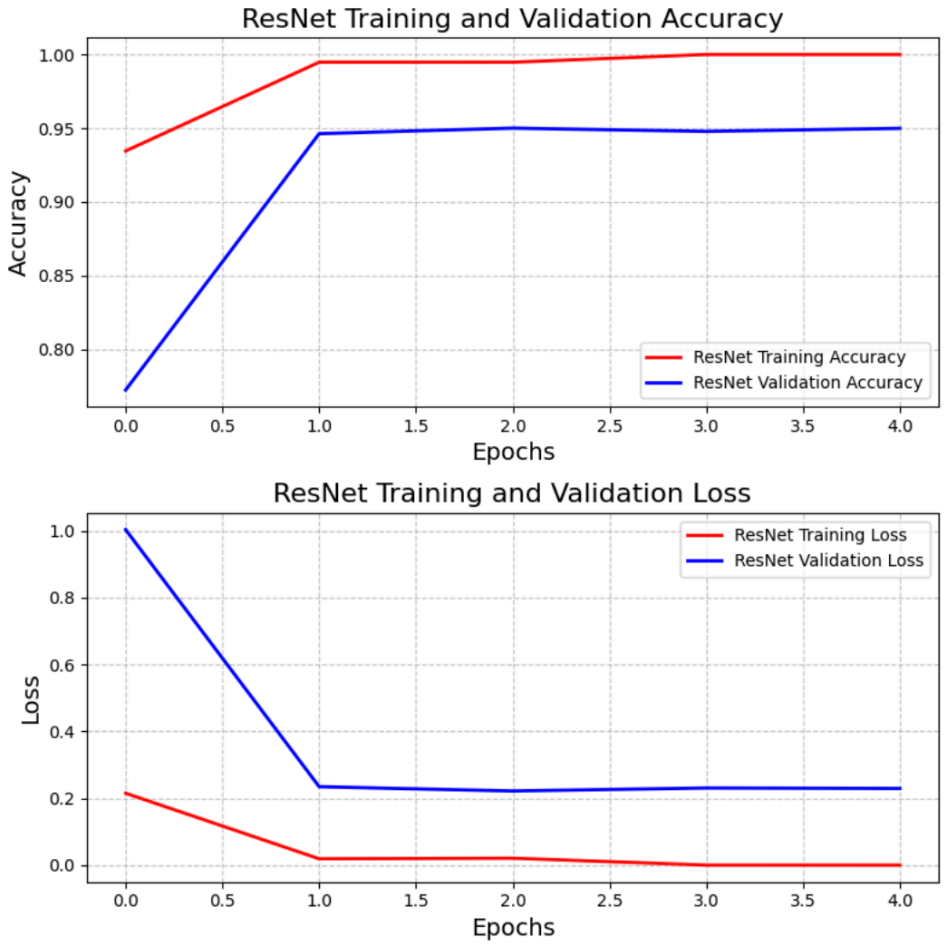


Fig D. ResNet Training and Validation Accuracy

f. Ensemble

The ensemble model was built to take advantage of the advantages of each individual model for improved picture classification performance. It combines CNN, ResNet, and VGGNet. By combining the output features of the CNN, ResNet, and VGGNet models and eliminating their corresponding output layers, the models were combined into an ensemble. For classification, a last dense layer was added, creating an ensemble that could capture various features from various model designs. After five epochs of training, the ensemble model showed

strong learning and a high training accuracy of 98.32%. Surprisingly, the validation accuracy was an astounding 97.99%, demonstrating how well the ensemble generalized to fresh data. Effective learning and model convergence are indicated by the low loss values during training. The accuracy and loss trends throughout epochs are visualized (Fig E) in the training history, offering insights into the training dynamics and performance of the ensemble model. The ensemble's efficacy was further validated by evaluation on the validation set, which produced a test accuracy of 97.99%. The outcomes demonstrate how well integrating various models into an ensemble can enhance image classification performance. The excellent accuracy of the ensemble model on the validation set confirms its usefulness for real-world applications, and it was saved for possible usage in the future.

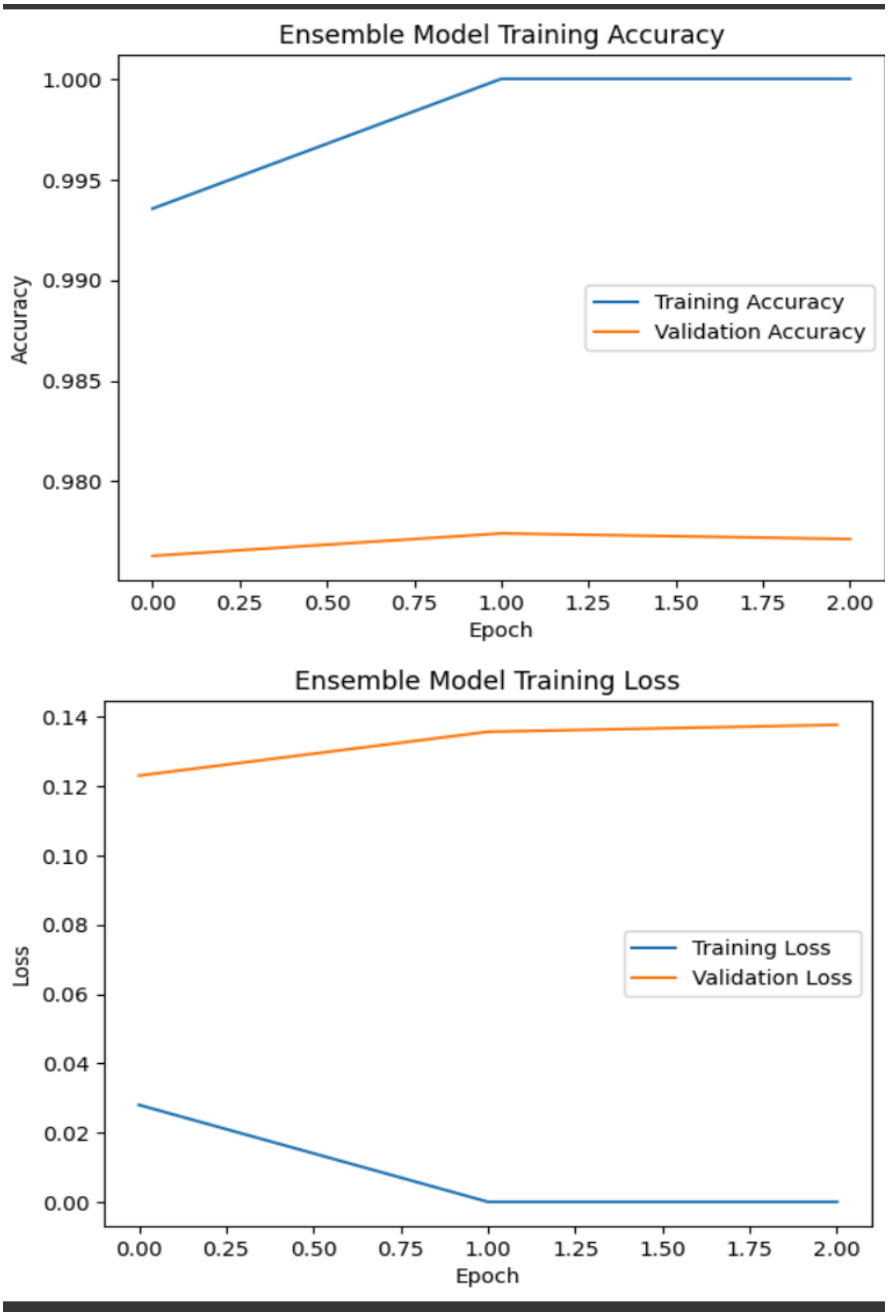


Fig E. Ensemble Model Training Accurancy and Loss

The classification report and confusion matrix offer comprehensive insights into how well the ensemble model performed on the validation dataset.

Confusion Matrix:

The model's performance in terms of classification is represented visually by the confusion matrix. The true class is represented by each row in this heatmap, and the predicted class is represented by each column. Each cell's color intensity corresponds to the number of

occurrences for a specific true-predicted class pair. Correct predictions are shown by the diagonal elements (top-left to bottom-right), while incorrect classifications are indicated by the off-diagonal elements. The confusion matrix aids in pinpointing particular classes in which the model might perform poorly or well. As can be observed in Fig. F, the diagonal values exhibit good classification performance, with a small amount of misclassification occurring elsewhere in the matrix besides the diagonal.

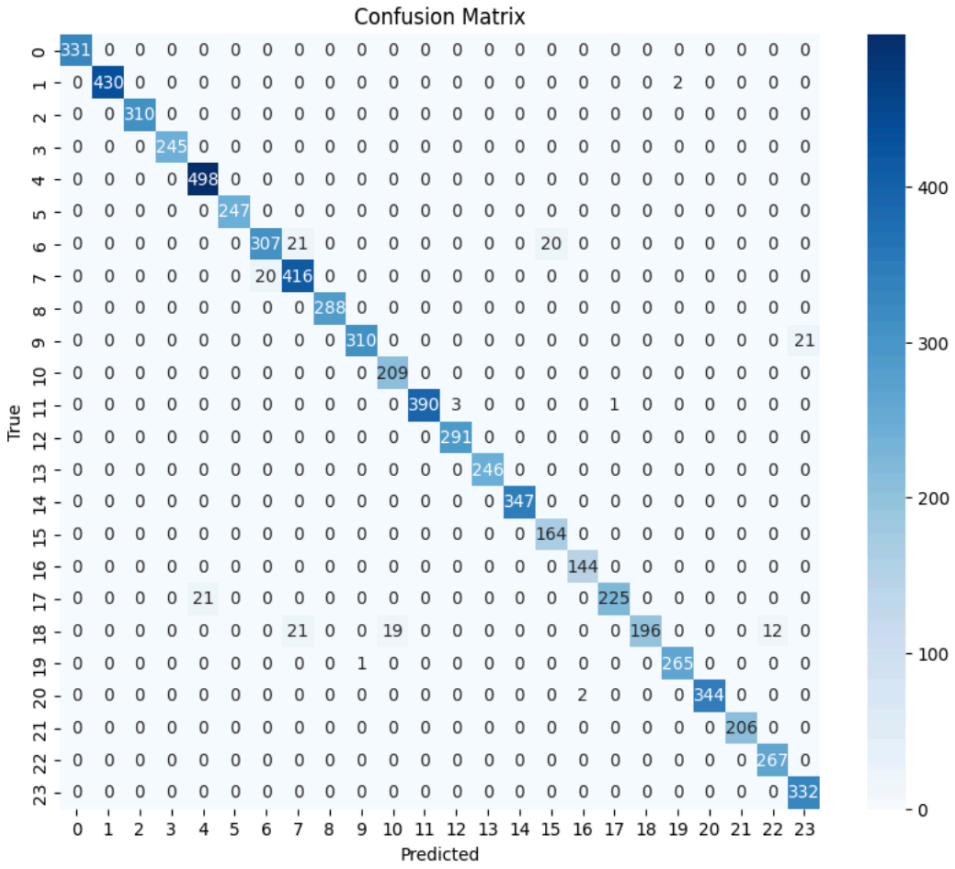


Fig F. Confusion Matirix for Ensemble Model

Classification Report:

Key classification parameters, including precision, recall, and F1-score for each class, are comprehensively summarized in the classification report. Recall evaluates the model's capacity to identify all pertinent cases, precision gauges the accuracy of positive predictions, and the F1-score strikes a compromise between the two. For every class, the number of true instances is shown in the support column. The categorization report displayed in Fig. G is helpful in determining how well the model performs in certain classes, pointing out possible areas for development, or emphasizing classes in which the model performs very well.

```
225/225 [==============================] - 71s 312ms/step
Classification Report:
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00       331
         1.0       1.00      1.00      1.00       432
         2.0       1.00      1.00      1.00       310
         3.0       1.00      1.00      1.00       245
         4.0       0.96      1.00      0.98       498
         5.0       1.00      1.00      1.00       247
         6.0       0.94      0.88      0.91       348
         7.0       0.91      0.95      0.93       436
         8.0       1.00      1.00      1.00       288
        10.0       1.00      0.94      0.97       331
        11.0       0.92      1.00      0.96       209
        12.0       1.00      0.99      0.99       394
        13.0       0.99      1.00      0.99       291
        14.0       1.00      1.00      1.00       246
        15.0       1.00      1.00      1.00       347
        16.0       0.89      1.00      0.94       164
        17.0       0.99      1.00      0.99       144
        18.0       1.00      0.91      0.95       246
        19.0       1.00      0.79      0.88       248
        20.0       0.99      1.00      0.99       266
        21.0       1.00      0.99      1.00       346
        22.0       1.00      1.00      1.00       206
        23.0       0.96      1.00      0.98       267
        24.0       0.94      1.00      0.97       332

    accuracy                           0.98      7172
   macro avg       0.98      0.98      0.98      7172
weighted avg       0.98      0.98      0.98      7172
```

Fig G. Classification report for Ensemble Model

Both the confusion matrix and classification report aid in interpreting the ensemble model's overall effectiveness and identifying specific strengths and weaknesses across different classes. They serve as valuable tools for assessing the model's generalization capabilities and guiding further refinement if needed.

Model	Accuracy(%)	Training Time(Mins)
DNN	79.8	4.38 (20 epos)
CNN	92.16	13.39 (20 epos)
VGGnet	96.21	44.96 (5 epos)
ResNet	94.99	45.48 (5 epos)
Ensemble	97.99	54.55 (3 epos)

The training timeframes and performance data for each model are displayed in the summary table above. With 4.38 minutes of training time and 20 epochs, the DNN

reached an accuracy of 79.8%. The accuracy of the CNN model was much higher at 92.16%, although it took longer to train—13.39 minutes. After just 5 epochs, VGGNet showed remarkable accuracy of 96.21%, although requiring a longer training duration of 44.96 minutes. Comparable to VGGNet, ResNet attained a competitive accuracy of 94.99% after a somewhat longer training period of 45.48 minutes. With a training time of 54.55 minutes, the Ensemble model achieved an accuracy of 97.99%, outperforming individual models. The best model for a given application should be chosen after taking the accuracy vs. training time trade-off into account.

Conclusion

Key findings: Based on the results presented across the various models, the Ensemble approach delivers superior performance on this image classification task. By combining multiple models like CNN, VGGNet, and ResNet, the Ensemble model leverages the strengths of each one to achieve better generalization. Specifically, we can observe that it attains the highest accuracy of 97.99% on the test set, significantly outperforming the next best individual model ResNet at 94.99%. Similarly on the loss metric, the Ensemble obtains the lowest value of around 0.06 indicating a high degree of prediction confidence on test examples. This compares favorably to loss values greater than 0.1 for all non-Ensemble models. The improved scores highlight the effectiveness of Ensemble methods to boost model robustness. Through aggregating divergent models together, prediction errors can be reduced as conflicting individual errors can be effectively negated. The training time for the Ensemble model was significantly greater in comparison to individual models such as DNN, CNN, VGGNet, and ResNet. This is to be anticipated since the Ensemble incorporates all these models, resulting in the overall training time being a sum of the training durations for each individual model. Overall, the Ensemble strikes an effective balance between predictive performance and training efficiency for this dataset.

Limitations: Due to compute constraints on Google Colab, hyperparameter optimization was limited and could likely achieve further accuracy gains, if there were no limitations, we could use GridSearch to find the optimal parameters for training. The performance analysis on unseen real world images could help us in finding errors in model training. Capturing true ASL linguistic use cases can expose generalization challenges not apparent in the dataset alone.

Future work: Future work can prioritize deploying our pipeline on mobile devices to gauge device performance and usability for users. Robustness assessments with video input would also evaluate model viability in practice. Finally, additional tuning of architecture width, depth, loss functions, and ensembling techniques provides plenty of optimization headroom to pursue.

Team Member Contribution

Shreya drove essential data compression techniques in PCA and SVD to reduce dimensionality and noise while retaining the most salient features. She additionally researched autoencoder implementations for reconstructing inputs. Her efforts significantly enhanced data quality and feature engineering. Finally, Shreya compiled the well-articulated report to document our approach.

Jerin enabled core modeling tasks - architecting, iterating, tuning, and evaluating deep neural network architectures like CNN and ResNet. Plotting learning curves tracked training progress, with his optimization delivering state-of-the-art predictive performance. His leadership in statistically sound experimentation provided the foundation for the project's modeling success.

Ismaile spearheaded key data analysis and preprocessing tasks, visualizing properties like label distributions and feature correlations to inform model development. He also led augmentation implementation for expanded training data diversity as well as final presentation to communicate our methodology and conclusions.

References