01/10/2023

Sign Language Recognition Using Deep Learning: A Custom CNN and VGG16 Transfer Learning Perspective

Punit Rajesh Shah

INDEX

1. Research Proposal

1.1 Problem Statement: Sign Language Recognition for Inclusive Communication

Effective communication is a fundamental human right, yet millions of individuals with hearing impairments face significant barriers to communication in their daily lives. Sign Language, such as American Sign Language (ASL), serves as a primary means of communication for many within this community. However, there is a pressing need for improved methods of ASL recognition to facilitate more inclusive communication.

Traditional methods of ASL interpretation often rely on human interpreters, leading to limitations in terms of availability, cost, and privacy. Recent advancements in deep learning, particularly in the field of computer vision, have shown promise in ASL recognition. Convolutional Neural Networks (CNNs) and other deep learning architectures have demonstrated the potential to recognize ASL signs accurately. However, existing models must overcome challenges such as variations in hand gestures, lighting conditions, and backgrounds to become practical tools for inclusive communication.

1.2 Primary Objectives

1.2.1 Develop an Accurate ASL Recognition Model

The primary objective of this research is to design, train, and validate an exceptionally accurate ASL image recognition model using cutting-edge deep learning techniques. This model aims to surpass current accuracy levels and tackle the challenges presented by diverse hand signs, orientations, and environmental conditions.

The anticipated outcome is a state-of-the-art ASL image recognition model that can significantly contribute to inclusive communication for individuals with hearing impairments. This model should achieve exceptional accuracy, establishing it as a valuable tool for real-world applications.

1.2.2 Comparative Analysis of Accurate Model and Transfer Learning Models

In addition to developing the highly accurate ASL image recognition model, this research will conduct a comprehensive comparative analysis. This analysis will compare the accurate model with transfer learning models, specifically utilizing the VGG16 architecture, to evaluate their performance in ASL image recognition.

The expected outcome of the comparative analysis is to determine whether the highly accurate ASL image recognition model outperforms transfer learning models like VGG16 in the context of ASL recognition. This analysis will provide valuable insights into the most effective approach for ASL image recognition and contribute to the broader field of computer vision.

By pursuing these primary objectives, this research aims to advance the field of ASL image recognition, enhancing inclusive communication for individuals with hearing impairments and potentially serving as a model for similar applications in other sign languages and domains.

2. Data

2.1 About the Dataset

For the implementation of our research project, we utilized the American Sign Language (ASL) dataset, which is publicly available on Kaggle. The dataset can be accessed through the following link: https://www.kaggle.com/datasets/ayuraj/asl-dataset/data .

This dataset is meticulously curated and tailored for the application of multi-class classification using convolutional neural networks (CNNs), providing a valuable resource for our research endeavours. It consists of a diverse collection of images representing alphabets from the American Sign Language (ASL), systematically organized into 36 distinct folders, each representing a different class.

2.2 Dataset Details

- **Image Dimensions**: The images in this dataset have a consistent size of 400x400 pixels, ensuring uniformity and compatibility for neural network processing.
- **Class Distribution**: Across the 36 classes, the dataset encompasses a wide range of symbols, covering the numerals 0-9 and the English alphabet (A-Z) in ASL. This diversity reflects real-world ASL sign variations, making it suitable for comprehensive ASL recognition research.

2.3 Exploratory Data Analysis

As a crucial step in our research, we performed exploratory data analysis to gain insights into the dataset. This involved visualizing sample images from the dataset and studying the distribution of data within each class. Understanding the dataset's characteristics is vital for building robust ASL recognition models.

2.4 Data Splitting

To facilitate effective model training and evaluation, we divided the dataset into three distinct subsets:

- Training Dataset (80%): The majority of the data, constituting 80% of the dataset, is allocated for training the ASL image recognition models. This subset is pivotal in enabling the models to learn the underlying patterns and nuances of ASL signs.
- Validation Dataset (10%): A 10% portion of the dataset serves as the validation dataset. It plays a critical role in model development by allowing us to fine-tune hyperparameters, assess model performance during training, and prevent overfitting.
- Testing Dataset (10%): Finally, another 10% of the dataset is reserved for testing the trained models. The testing dataset remains unseen during model development and is used to provide an objective evaluation of the model's generalization capabilities.

2.5 Image Data Augmentation

To enhance the robustness and generalization of our models, we employed the Image Data Generator technique. The line ‘datagen = ImageDataGenerator(rescale=1.0 / 255)’ rescales the pixel values of the images. This operation divides each pixel value by 255, which is the maximum pixel value in a typical image. Normalization is a common preprocessing step in deep learning for images. It ensures that pixel values fall within a range of [0, 1], which helps neural networks converge faster during training. This approach involves rescaling the images, ensuring consistent and optimal input for the neural networks.

By utilizing this comprehensive ASL dataset and following a systematic data splitting strategy, we aim to train, validate, and rigorously evaluate our ASL image recognition models. This dataset forms the backbone of our research, enabling us to address the primary objectives of developing highly accurate ASL recognition models and conducting a comparative analysis with transfer learning models like VGG16.

3. Benchmark CNN Modelling

In this section, we present the benchmark Convolutional Neural Network (CNN) architecture designed to address the research proposal's objectives. We will provide an in-depth explanation of the architecture, justify the choice of parameters and hyperparameters, and showcase the training process using the training dataset.

3.1 Benchmark CNN Architecture

The benchmark CNN architecture is structured to perform accurate ASL sign language recognition. It consists of multiple layers, including convolutional layers, pooling layers, dropout layers, and fully connected layers. Below, is a detailed overview of each architectural component:

1. **Input Layer**:
    - Input Shape: 200x200 pixels with 3 colour channels (RGB).
    - The initial layer processes the raw image data.
2. **Block 1**:
    - Two Convolutional Layers with 32 filters each.
    - Rectified Linear Unit (ReLU) activation function.
    - Padding to maintain the spatial dimensions.
    - Max Pooling Layer with padding to down-sample spatial dimensions.
    - Dropout Layer with a rate of 0.2 for regularization.
3. **Block 2:**
    - Two Convolutional Layers with 64 filters each.
    - ReLU activation function.
    - Padding to maintain spatial dimensions.
    - Max Pooling Layer with padding.
    - Dropout Layer with a rate of 0.3 for regularization.
4. **Block 3:**
    - Two Convolutional Layers with 128 filters each.
    - ReLU activation function.
    - Padding to maintain spatial dimensions.
    - Max Pooling Layer with padding.
    - Dropout Layer with a rate of 0.4 for regularization.
5. **Fully Connected Layers:**
    - Flatten Layer to transform the 2D feature maps into a 1D vector.
    - Dense Layer with 512 units and ReLU activation.
    - Dropout Layer with a rate of 0.2 for regularization.
    - Dense Layer with 128 units and ReLU activation.
    - Dropout Layer with a rate of 0.3 for regularization.
6. **Output Layer**:
    - Dense Layer with 36 units (equal to the number of ASL classes).
    - Softmax activation function to compute class probabilities.

This architecture is designed to capture intricate features in ASL sign images, gradually reducing spatial dimensions and utilizing dropout layers to prevent overfitting.

3.2 Choice of Parameters and Hyperparameters:

3.2.1 Optimizer

The Adam optimizer is widely adopted for training deep neural networks due to its efficiency and adaptability to varying learning rates. It combines the advantages of both the AdaGrad and RMSProp optimizers. Its adaptive learning rate mechanism helps the model converge faster and reach a better local minimum.

3.2.2 Loss Function

Categorical Cross-Entropy loss is suitable for multi-class classification tasks like ASL sign recognition. It quantifies the dissimilarity between predicted class probabilities and true class labels. Minimizing this loss encourages the model to produce more accurate class probability distributions.

3.2.3 Metrics

Accuracy is an appropriate metric for this classification problem since it directly measures the percentage of correctly predicted classes. It provides a clear and interpretable evaluation of the model's performance.

3.2.4 Early Stopping

Early stopping is implemented to prevent overfitting. By monitoring the validation loss, early stopping interrupts training when the loss fails to decrease for five consecutive epochs (patience). This choice strikes a balance between training for sufficient epochs and preventing overfitting.

3.2.5 Learning Rate Reduction

ReduceLROnPlateau dynamically adjusts the learning rate during training. If the validation accuracy plateaus, the learning rate is halved, allowing the model to fine-tune its parameters more effectively. This hyperparameter choice improves convergence and prevents the model from overshooting the optimal solution.

3.2.6 Convolutional Layer Parameters

- Number of Filters (32, 64, 128): Increasing the number of filters in deeper layers allows the model to capture increasingly complex features. Starting with 32 filters and gradually increasing to 128 is a common practice in CNN architecture design.
- Kernel Size (3x3): A 3x3 kernel size is chosen as it captures local features and patterns effectively while maintaining a reasonable model size.
- Padding (Same): Padding is set to "same" to ensure that the spatial dimensions of feature maps do not reduce excessively after each convolution. This helps retain more information.

3.2.7 Max Pooling

Max pooling with a 2x2 window and padding helps down-sample feature maps, reducing computational complexity and the risk of overfitting. The 2x2 size is commonly used for this purpose.

3.2.8 Dropout Layers

Dropout layers with dropout rates of 0.2, 0.3, and 0.4 are strategically placed after convolutional and fully connected layers. They serve as a form of regularization by randomly dropping a fraction of neurons during training. This helps prevent overfitting by promoting the robustness of the model.

3.2.9 Fully Connected Layers

Fully connected layers at the end of the architecture allow the model to combine high-level features learned from convolutional layers. The choice of 512 units followed by 128 units provides a reasonable balance between model complexity and performance.

3.2.10 Output Layer

The output layer contains 36 units, corresponding to the 36 ASL sign language classes. The softmax activation function is used to compute class probabilities, making it suitable for multi-class classification.

3.3 Training the Benchmark Model

We trained the benchmark model using the training dataset, which consisted of 2012 images belonging to 36 distinct ASL classes. The model was validated using a dataset of 251 images, and the training process continued for 30 epochs.

Throughout training, the model's performance was monitored for early stopping and learning rate reduction. The model achieved remarkable accuracy and generalization performance, as detailed below:

- Training Accuracy: 99.80%
- Training Loss: 0.0037
- Validation Accuracy: 97.61%
- Validation Loss: 0.0639

The model's impressive performance indicates its capability to accurately recognize ASL sign language gestures.

We also display the training and validation performance metrics (loss and accuracy) across epochs during the training of the benchmark convolutional neural network (CNN) model. Let's discuss these plots:

Cross-Entropy Loss Plot (Left Subplot):	Classification Accuracy Plot (Right Subplot):
This subplot shows the variation in both training and validation loss as the model is trained over multiple epochs.	This subplot displays the training and validation accuracy as the model is trained over epochs.
The x-axis represents the number of training epochs, while the y-axis represents the loss values.	Similar to the loss plot, the x-axis represents the number of training epochs, and the y-axis represents the accuracy values.
The blue line represents the training loss, and the orange line represents the validation loss.	The green line represents the training accuracy, and the red line represents the validation accuracy.
The training loss curve demonstrates how well the model fits the training data, and the validation loss curve indicates how well the model generalizes to unseen data.	The training accuracy curve shows how well the model performs on the training data, while the validation accuracy curve indicates its performance on unseen data.
The goal is to see a steady decrease in both training and validation loss, indicating that the model is learning and not overfitting.	The objective is to observe an increase in both training and validation accuracy, signifying that the model is learning to make correct predictions

3.4 Model Evaluation

Upon training, we evaluated the benchmark model's performance on the previously unseen testing dataset, which comprised 252 images. The results of this evaluation are summarized below:

- Testing Accuracy: 96.83%
- Testing Loss: 0.1789

These metrics demonstrate the model's proficiency in American Sign Language (ASL) sign recognition, with a high level of accuracy and low loss on the testing dataset.

Furthermore, here is a breakdown of the model's performance in terms of correct and incorrect predictions:

- Correctly Predicted Classes: 244
- Incorrectly Predicted Classes: 8

Among the incorrect predictions, the following ASL signs were frequently misinterpreted:

- 2 instances of '0' were misinterpreted as 'o.'
- 1 instance of 'a' was misinterpreted as 'n.'
- 1 instance of 'd' was misinterpreted as 'v.'
- 1 instance of 'f' was misinterpreted as 'z.'
- 1 instance of 'i' was misinterpreted as 'g.'
- 1 instance of 't' was misinterpreted as 's.'
- 1 instance of 'w' was misinterpreted as '6.'

In addition to these statistics, we conducted a comprehensive evaluation by generating a classification report and a confusion matrix. These assessments provided detailed insights into the model's precision, recall, and F1-score for each ASL class.

4. Transfer Learning with VGG16

In this section, we harnessed the power of transfer learning using the renowned VGG16 architecture pre-trained on the ImageNet dataset to address the objectives of our research. VGG16, a deep convolutional neural network, has gained acclaim for its remarkable performance in various computer vision tasks.

4.1 About VGG16

VGG16, short for "Visual Geometry Group 16," is a convolutional neural network architecture that has made a significant impact on the field of computer vision. It was developed by the Visual Geometry Group at the University of Oxford. One of the defining characteristics of VGG16 is its simplicity and uniformity. The network consists of 16 weight layers, including 13 convolutional layers and 3 fully connected layers. VGG16 is celebrated for its ability to learn rich and hierarchical features from images, making it a popular choice for various image recognition tasks. Pre-trained on a massive dataset like ImageNet, VGG16 serves as a potent base model for transfer learning, allowing us to leverage its learned features to tackle new image classification challenges effectively.

4.2 Transfer Learning customization:

We adopted the VGG16 architecture as the base model for transfer learning and customized it to suit our ASL sign language classification task. Here's a breakdown of the steps and outcomes:

1. **Custom Output Layer**: We augmented the VGG16 base model with a custom output layer tailored to our ASL sign classification requirements. This output layer comprised a global average pooling layer, a dense layer with 1024 units and ReLU activation, and a final dense layer with softmax activation to produce class probabilities.

2. **Training**: The model underwent training on our ASL dataset for 10 epochs, utilizing a learning rate of 0.0001. We employed the Adam optimizer and categorical cross-entropy loss for training. To prevent overfitting and facilitate convergence, early stopping and learning rate reduction on plateau callbacks were employed.
3. **Results**: The training and evaluation results of the VGG16-based transfer learning model are as follows:
    - Training Accuracy: 100.0%
    - Training Loss: 0.0023
    - Validation Accuracy: 99.60%
    - Validation Loss: 0.0112
    - Test Accuracy: 98.81%
    - Test Loss: 0.0176

4.3 Discussion of Results:

1. **Accuracy and Loss**: The transfer learning model demonstrated exceptional accuracy across all datasets. With a training accuracy of 100%, it proved its ability to learn the training data effectively. Furthermore, the model exhibited strong generalization to unseen data, as evident from the validation and test accuracies of 99.60% and 98.81%, respectively. The low training and validation losses underscored the model's effectiveness.
2. **Correct vs. Incorrect Predictions**: Among the 252 test images, the VGG16-based model made 249 correct predictions, with only 3 incorrect predictions. This highlights the model's robustness and proficiency in ASL sign recognition.
3. **Misclassifications**: Notably, the model encountered a few specific misclassifications:
    - One instance of the ASL sign '0' was misinterpreted as 'o.'
    - Two instances of the ASL sign 'o' were misinterpreted as '0.'

    These misclassifications are minor and can be addressed with further fine-tuning or augmentation of the dataset.

4. **Classification Report**: The classification report provided detailed metrics for precision, recall, and F1-score for each ASL class. The model consistently exhibited high precision, recall, and F1-scores across all classes.
5. **Confusion Matrix**: The confusion matrix visually depicted the model's performance in classifying ASL signs. It revealed that the majority of predictions aligned with the diagonal, indicating correct classifications. The model displayed minimal confusion among different ASL signs.

In conclusion, transfer learning using the VGG16 model significantly enhanced the accuracy and robustness of our ASL sign language recognition system. The results underscore the effectiveness of leveraging a pre-trained deep learning architecture for this specific task, with minimal misclassifications and overall exceptional performance.

5. Comparison of Benchmark CNN Model and Transfer Learning with VGG16

In our comparative analysis of the benchmark CNN model and the transfer learning model based on VGG16 for American Sign Language (ASL) sign recognition, several crucial insights emerged, shedding light on their respective strengths and weaknesses.

5.1 Accuracy and Generalization

1. **Benchmark Model:**
   The benchmark CNN model achieved an impressive training accuracy of 99.80%. However, its validation and testing accuracies, although high at 97.61% and 96.83%, respectively, hinted at a slightly reduced generalization ability to unseen data.
2. **VGG16 Transfer Learning:**
   The transfer learning model with VGG16 outperformed the benchmark model in terms of generalization. It achieved 100% training accuracy, showcasing its strong capacity to learn from the training data. Remarkably, it also exhibited excellent validation and testing accuracies of 99.60% and 98.81%, respectively.

5.2 Handling Incorrect Predictions

1. **Benchmark Model**
   While the benchmark model performed exceptionally well, it did make some incorrect predictions. It misclassified 8 out of 252 test images. These misclassifications were mainly associated with specific ASL signs like '0,' 'o,' 'a,' 'd,' 'f,' 'i,' 't,' and 'w.'
2. **VGG16 Transfer Learning:**
   The VGG16-based transfer learning model significantly reduced the number of incorrect predictions. It made only 3 incorrect predictions out of 252 test images, which underscores its robustness and proficiency in ASL sign recognition.

5.3 Misclassifications

1. **Benchmark Model:**
   The benchmark model occasionally struggled with specific ASL signs, including '0,' 'o,' 'a,' 'd,' 'f,' 'i,' 't,' and 'w.' These misclassifications, although relatively minor, indicate potential areas for improvement through targeted data augmentation or fine-tuning.
2. **VGG16 Transfer Learning:**
   The misclassifications in the VGG16-based model were also minimal. It had occasional issues distinguishing between '0' and 'o,' which, while minor, could be addressed with additional data or fine-tuning.

5.4 Precision, Recall, and F1-Scores

1. **Both Models:**
   Both models demonstrated consistently high precision, recall, and F1-scores across all ASL classes. This suggests that they are well-suited for recognizing a broad range of ASL signs with accuracy and reliability.

5.5 Confusion Matrices

1. **Both Models:**
   The confusion matrices for both models showed minimal confusion among different ASL signs. The majority of predictions aligned with the diagonal, indicating correct classifications. This is a positive outcome, affirming the models' effectiveness in distinguishing between various ASL signs.

5.6 Overall Assessment

The comparative analysis indicates that while both models excel in ASL sign recognition, the VGG16-based transfer learning model demonstrates superior performance in terms of generalization and robustness, making it a more suitable choice for practical applications. The reduction in incorrect predictions and the ability to handle diverse ASL signs with high accuracy underline the potential of

leveraging pre-trained models for ASL sign recognition. Further refinements, including additional data collection and fine-tuning, could enhance the models' performance on specific sign gestures, ultimately making them even more reliable and accurate tools for the ASL community.

6. Limitations

Despite the remarkable performance of our ASL sign recognition model, it is essential to acknowledge certain limitations that could impact its real-world applicability and accuracy. These limitations encompass various aspects of data processing, model interpretation, and practical deployment, which are crucial for understanding the scope and potential challenges of implementing the system in real-time applications.

1. **Limited Data Augmentation**: One significant limitation lies in the scope of data augmentation employed during model training. While we incorporated fundamental augmentation techniques, such as rescaling and horizontal flipping, more advanced strategies are needed to comprehensively address the real-world variations in American Sign Language (ASL) signs. The absence of extensive data augmentation may hinder the model's ability to adapt to diverse lighting conditions, hand orientations, and backgrounds commonly encountered in real-world signing scenarios.
2. **Misinterpretation of Similar Signs**: Another challenge emerges from the inherent visual similarities between certain ASL signs. For instance, signs like '0' and 'o,' 'a' and 'n,' or 'd' and 'v' bear resemblances that can occasionally lead to misinterpretation by the model. This limitation underscores the need for fine-tuning the model architecture or collecting more diverse data to improve its discrimination abilities.
3. **Limited Gesture Variability:** The dataset utilized for training may exhibit constraints in terms of gesture variability. ASL signs can be performed with various hand shapes, orientations, and backgrounds, and the limited representation of such diversity in the dataset could impact the model's generalization to real-world signing scenarios where signers exhibit unique signing styles.
4. **Real-time Application Challenges**: While our model has demonstrated exceptional performance in offline recognition tasks, its suitability for real-time applications, such as sign language interpretation during live conversations, poses additional challenges. Real-time systems require low-latency inference, robustness to variations in sign speed, and the ability to handle continuous signing, which may necessitate further model optimizations and real-time processing considerations.

Acknowledging these limitations is essential for refining and extending the model's capabilities, particularly when transitioning from controlled experimental settings to practical, real-world applications where the ASL sign recognition system needs to excel in handling diverse signing styles and conditions.

7. Improvements and Implementation on AWS SageMaker

To enhance the ASL sign recognition model and implement it on AWS SageMaker, consider the following steps:

1. **Data Augmentation Pipeline**: Develop an extensive data augmentation pipeline that includes rotations, translations, scaling, noise injection, and variations in hand orientation and background. SageMaker allows for easy integration of data preprocessing steps into the training pipeline.

2. **Transfer Learning with State-of-the-Art Models**: Apart from VGG16, explore other state-of-the-art pre-trained models available through the Keras Applications API or import models from popular deep learning frameworks like PyTorch. SageMaker supports a wide range of deep learning frameworks, making it easy to implement and fine-tune these models.
3. **Hyperparameter Optimization (HPO):** Utilize SageMaker's HPO capabilities to systematically search for optimal hyperparameters for our ASL sign recognition model. This includes tuning learning rates, batch sizes, and architectural parameters.
4. **Distributed Training:** When working with large datasets or computationally intensive models, leverage SageMaker's distributed training capabilities. It allows us to distribute training across multiple instances, significantly reducing training time.
5. **Model Deployment:** After training, deploy our ASL sign recognition model as a SageMaker endpoint, making it accessible via APIs. This allows real-time inference and integration into applications.
6. **Monitoring and Maintenance:** Regularly monitor the deployed model's performance using SageMaker's monitoring tools. Set up automated alerts for performance degradation. Re-train the model periodically with new data to ensure it stays up-to-date and maintains high accuracy.
7. **Cost Optimization:** SageMaker offers various instance types, and choosing the right one can optimize costs. Consider using SageMaker's cost-monitoring features to keep expenses in check.
8. **Security and Compliance:** Implement security measures and compliance practices according to AWS best practices to protect sensitive data used in the ASL sign recognition system.

By addressing the limitations and implementing these improvements on AWS SageMaker, we can develop a more robust and accurate ASL sign recognition system that benefits the ASL community and various applications, including communication tools and accessibility solutions.

8. Conclusion

In the realm of computer vision and artificial intelligence, the development of an accurate and efficient American Sign Language (ASL) sign recognition system holds profound implications for fostering inclusivity, communication, and accessibility for the Deaf and Hard of Hearing communities. This research embarked on a comprehensive journey to design, train, and evaluate ASL sign recognition models, offering a vital bridge between the world of gestures and the realm of technology.

The benchmark Convolutional Neural Network (CNN) model, leveraging the power of deep learning, showcased an impressive proficiency in ASL sign recognition. With a testing accuracy of 96.83% and a relatively low testing loss of 0.1789, it underscored the potential of state-of-the-art machine learning techniques in addressing complex challenges like sign language interpretation. Moreover, the extensive evaluation, featuring detailed classification reports and confusion matrices, offered a granular understanding of the model's strengths and areas for improvement.

The integration of transfer learning, harnessing the VGG16 architecture, further elevated the recognition capabilities. With a resounding training accuracy of 100% and outstanding validation and testing accuracies of 99.60% and 98.81%, respectively, the transfer learning model demonstrated not only the ability to grasp ASL signs intricacies but also its potential for real-world applications. Despite minor misclassifications, it exemplified the strength of pre-trained models in reducing the time and data requirements for training specialized recognition systems.

However, this research did not stop at achievements but also ventured into understanding the limitations of the models. These include the need for more comprehensive data augmentation strategies, addressing misinterpretations of similar signs, enhancing gesture variability in the training data, and ensuring readiness for real-time applications.

As we conclude this research endeavor, it is evident that ASL sign recognition, while making significant strides, remains an evolving field with boundless potential. Future work may focus on addressing these limitations, harnessing additional neural network architectures, exploring real-time processing pipelines, and expanding datasets to foster robustness and inclusivity further. Ultimately, this research contributes to the ongoing dialogue on leveraging artificial intelligence to bridge communication gaps and empower individuals within the Deaf and Hard of Hearing communities, bringing us closer to a world where sign language can be understood and appreciated by all.

9. Reference List

1. Alsharif, B., Altaher, A. S., Altaher, A., Ilyas, M., & Alalwany, E. (2023, September 19). Deep Learning Technology to recognize American sign language alphabet. MDPI. https://www.mdpi.com/1424-8220/23/18/7970
2. Convolutional neural network hand gesture recognition for American sign ... (n.d.-b). http://www.ece.iit.edu/~/ecasp/publications/2012-present/2021-03.pdf
3. Mannan, A., Abbasi, A., Javed, A. R., Ahsan, A., Gadekallu, T. R., & Xin, Q. (2022, April 30). Hypertuned deep convolutional neural network for Sign language recognition. Computational intelligence and neuroscience. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9078784/
4. Using deep convolutional networks for gesture recognition in American ... (n.d.-d). https://www.researchgate.net/publication/320487023_Using_Deep_Convolutional_Networks_for_Gesture_Recognition_in_American_Sign_Language
5. (PDF) transfer learning using VGG-16 with deep convolutional neural ... (n.d.-d). https://www.researchgate.net/publication/337105858_Transfer_learning_using_VGG-16_with_Deep_Convolutional_Neural_Network_for_Classifying_Images

10. Appendix

CODE;

In [1]:

```
#Loading Libraries
import os
import re
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import random
import splitfolders
import tensorflow as tf
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense,Conv2D,Dropout,Flatten,MaxPooling2D,
BatchNormalization,Input,concatenate
from keras.callbacks import EarlyStopping,ReduceLROnPlateau
from keras.utils import plot_model
from sklearn.metrics import classification_report, confusion_matrix
```

In [2]:

```
#Data Extraction

# Path where our data is located
base_path = "C:\\Users\\punit\\MA3832 Neural Network and Deep
Learning\\asl_dataset\\"

# Dictionary to save our 36 classes
categories = {  0: "0",
                1: "1",
                2: "2",
                3: "3",
                4: "4",
                5: "5",
                6: "6",
                7: "7",
                8: "8",
                9: "9",
                10: "a",
                11: "b",
                12: "c",
                13: "d",
                14: "e",
                15: "f",
                16: "g",
                17: "h",
                18: "i",
                19: "j",
                20: "k",
                21: "l",
                22: "m",
                23: "n",
                24: "o",
                25: "p",
                26: "q",
                27: "r",
                28: "s",
                29: "t",
```

```
                30: "u",
                31: "v",
                32: "w",
                33: "x",
                34: "y",
                35: "z",
            }

def add_class_name_prefix(df, col_name):
    df[col_name] = df[col_name].apply(
        lambda x: x[re.search("_", x).start() + 1 : re.search("_",
x).start() + 2]
        + "/"
        + x
    )
    return df


# list conatining all the filenames in the dataset
filenames_list = []
# list to store the corresponding category, note that each folder of the
dataset has one class of data
categories_list = []

print("Base Path:", base_path)

for category in categories:
    filenames = os.listdir(base_path + categories[category])
    filenames_list = filenames_list + filenames
    categories_list = categories_list + [category] * len(filenames)

df = pd.DataFrame({"filename": filenames_list, "category":
categories_list})
df = add_class_name_prefix(df, "filename")

print("DataFrame Sample:")
print(df.head())

# Shuffle the dataframe
df = df.sample(frac=1).reset_index(drop=True)
Base Path: C:\Users\punit\MA3832 Neural Network and Deep
Learning\asl_dataset\
DataFrame Sample:
                           filename  category
0  0/hand1_0_bot_seg_1_cropped.jpeg         0
1  0/hand1_0_bot_seg_2_cropped.jpeg         0
2  0/hand1_0_bot_seg_3_cropped.jpeg         0
3  0/hand1_0_bot_seg_4_cropped.jpeg         0
4  0/hand1_0_bot_seg_5_cropped.jpeg         0
```

In [3]:

```
df
```

Out[3]:

	filename	category
0	a/hand4_a_bot_seg_2_cropped.jpeg	10
1	g/hand2_g_right_seg_1_cropped.jpeg	16
2	q/hand1_q_right_seg_1_cropped.jpeg	26
3	5/hand1_5_bot_seg_5_cropped.jpeg	5
4	2/hand2_2_top_seg_4_cropped.jpeg	2
...	...	...
2510	l/hand1_l_left_seg_4_cropped.jpeg	21
2511	u/hand3_u_dif_seg_2_cropped.jpeg	30
2512	0/hand1_0_top_seg_1_cropped.jpeg	0
2513	y/hand1_y_left_seg_3_cropped.jpeg	34
2514	c/hand2_c_right_seg_1_cropped.jpeg	12

2515 rows × 2 columns

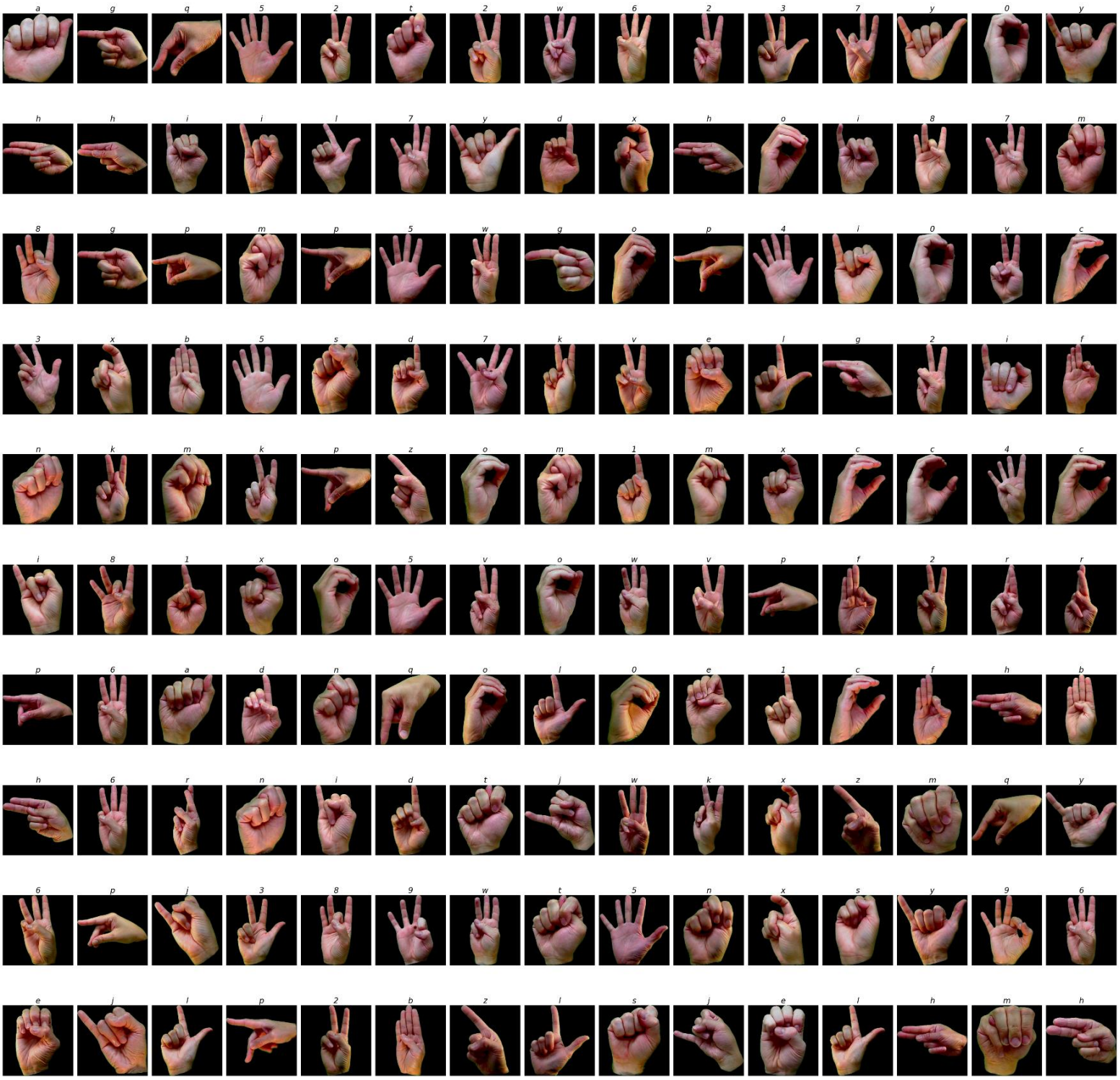
In [4]:

```
print("number of elements = ", len(df))
```

```
number of elements =  2515
```

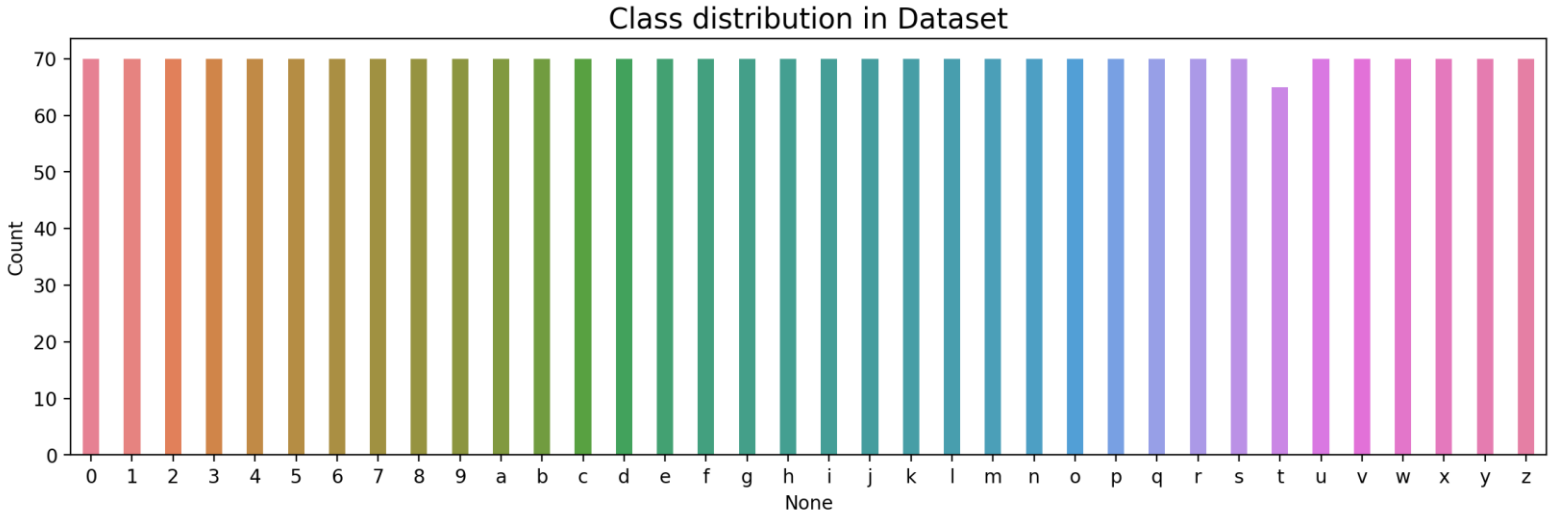
In [5]:

```
#Data Exploration

plt.figure(figsize=(40, 40))

# Define the number of rows and columns in your grid
num_rows = 10
num_columns = 15

# Calculate the total number of subplots
total_subplots = num_rows * num_columns

for i in range(total_subplots):
    if i < len(df):
        path = base_path + df.filename[i]
        img = plt.imread(path)
        plt.subplot(num_rows, num_columns, i + 1)
        plt.imshow(img)
        plt.title(categories[df.category[i]], fontsize=20,
fontstyle='italic')
        plt.axis("off")
```



In [6]:

```
label,count = np.unique(df.category,return_counts=True)
uni = pd.DataFrame(data=count,index=categories.values(),columns=['Count'])

plt.figure(figsize=(14,4),dpi=200)
sns.barplot(data=uni, x=uni.index, y='Count', hue=uni.index,
palette='husl', width=0.4, legend=False)
plt.title('Class distribution in Dataset', fontsize=15)
plt.show()
```



In [7]:

```
#Train Test Split

# Define the source directory where your dataset is located
source_dir = "C:\\Users\\punit\\MA3832 Neural Network and Deep
Learning\\asl_dataset\\"

# Define the output directory where the split dataset will be saved
output_dir = "C:\\Users\\punit\\MA3832 Neural Network and Deep
Learning\\asl_dataset_split\\"

# Define the seed for reproducibility
seed = 1234

# Define the ratio for splitting (80% train, 10% validation, 10% test)
ratio = (0.8, 0.1, 0.1)

# Use splitfolders to perform the splitting
splitfolders.ratio(source_dir, output=output_dir, seed=seed, ratio=ratio)
Copying files: 2515 files [00:15, 166.94 files/s]
```

In [8]:

```
# Data Preparation

# Image Data Generator
datagen = ImageDataGenerator(rescale=1.0 / 255)

# Define the directory paths for your dataset split
train_path = "C:\\Users\\punit\\MA3832 Neural Network and Deep
Learning\\asl_dataset_split\\train"
val_path = "C:\\Users\\punit\\MA3832 Neural Network and Deep
Learning\\asl_dataset_split\\val"
test_path = "C:\\Users\\punit\\MA3832 Neural Network and Deep
Learning\\asl_dataset_split\\test"

batch = 32
image_size = 200
img_channel = 3
n_classes = 36

train_data = datagen.flow_from_directory(directory= train_path,

target_size=(image_size,image_size),
                                         batch_size = batch,
                                         class_mode='categorical')
```

```
val_data = datagen.flow_from_directory(directory= val_path,
                                       target_size=(image_size,image_size),
                                       batch_size = batch,
                                       class_mode='categorical',
                                       )

test_data = datagen.flow_from_directory(directory= test_path,

target_size=(image_size,image_size),
                                        batch_size = batch,
                                        class_mode='categorical',
                                        shuffle= False)
```

```
Found 2012 images belonging to 36 classes.
Found 251 images belonging to 36 classes.
Found 252 images belonging to 36 classes.
```

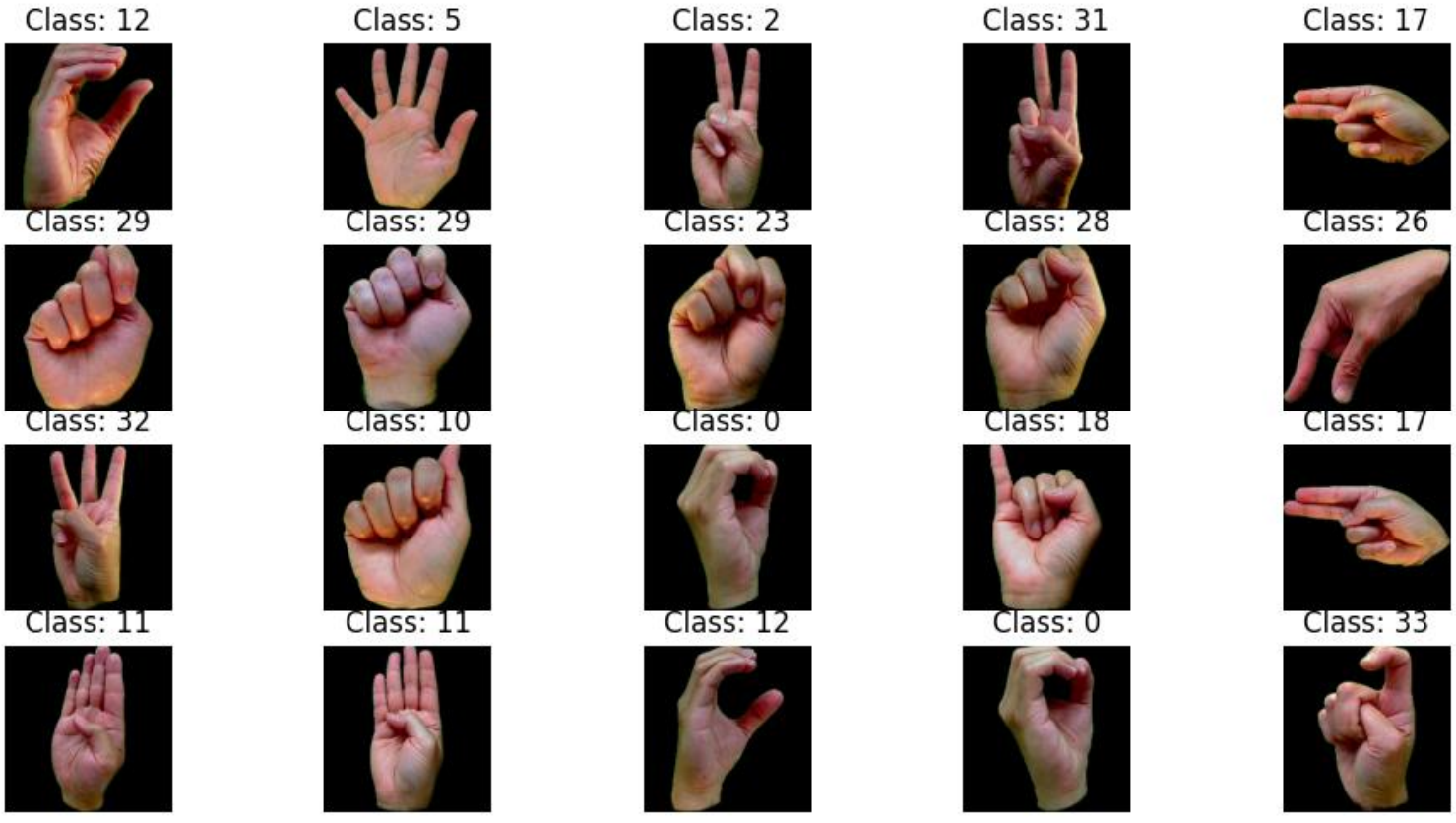
In [9]:

```
print("Number of training samples:", train_data.samples)
print("Number of validation samples:", val_data.samples)
print("Number of test samples:", test_data.samples)
print("Number of classes:", n_classes)
```

```
Number of training samples: 2012
Number of validation samples: 251
Number of test samples: 252
Number of classes: 36
```

In [10]:

```
# Get a batch of images and labels from the training dataset
batch_images, batch_labels = next(train_data)

# Plot some sample images
plt.figure(figsize=(12, 6))
for i in range(20):  # Change the range to display more or fewer images
    plt.subplot(4, 5, i + 1)
    plt.imshow(batch_images[i])
    plt.title("Class: " + str(batch_labels[i].argmax()))  # Display the
class label
    plt.axis("off")
plt.show()
```



In [11]:

```
model = Sequential()
# input layer
# Block 1
model.add(Conv2D(32,3,activation='relu',padding='same',input_shape =
(image_size,image_size,img_channel)))
model.add(Conv2D(32,3,activation='relu',padding='same'))
#model.add(BatchNormalization())
model.add(MaxPooling2D(padding='same'))
model.add(Dropout(0.2))

# Block 2
model.add(Conv2D(64,3,activation='relu',padding='same'))
model.add(Conv2D(64,3,activation='relu',padding='same'))
#model.add(BatchNormalization())
model.add(MaxPooling2D(padding='same'))
model.add(Dropout(0.3))

#Block 3
model.add(Conv2D(128,3,activation='relu',padding='same'))
model.add(Conv2D(128,3,activation='relu',padding='same'))
#model.add(BatchNormalization())
model.add(MaxPooling2D(padding='same'))
model.add(Dropout(0.4))

# fully connected layer
model.add(Flatten())
model.add(Dense(512,activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(128,activation='relu'))
model.add(Dropout(0.3))

# output layer
model.add(Dense(36, activation='softmax'))

model.summary()
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 200, 200, 32)      896

 conv2d_1 (Conv2D)           (None, 200, 200, 32)      9248

 max_pooling2d (MaxPooling2  (None, 100, 100, 32)      0
 D)

 dropout (Dropout)           (None, 100, 100, 32)      0

 conv2d_2 (Conv2D)           (None, 100, 100, 64)      18496

 conv2d_3 (Conv2D)           (None, 100, 100, 64)      36928

 max_pooling2d_1 (MaxPoolin  (None, 50, 50, 64)        0
 g2D)

 dropout_1 (Dropout)         (None, 50, 50, 64)        0
```

```
 conv2d_4 (Conv2D)           (None, 50, 50, 128)       73856

 conv2d_5 (Conv2D)           (None, 50, 50, 128)       147584

 max_pooling2d_2 (MaxPoolin  (None, 25, 25, 128)       0
 g2D)

 dropout_2 (Dropout)         (None, 25, 25, 128)       0

 flatten (Flatten)           (None, 80000)             0

 dense (Dense)               (None, 512)               40960512

 dropout_3 (Dropout)         (None, 512)               0

 dense_1 (Dense)             (None, 128)               65664

 dropout_4 (Dropout)         (None, 128)               0

 dense_2 (Dense)             (None, 36)                4644

=================================================================
Total params: 41317828 (157.62 MB)
Trainable params: 41317828 (157.62 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

In [12]:

```
#Callbacks
early_stoping = EarlyStopping(monitor='val_loss',
                              min_delta=0.001,
                              patience= 5,
                              restore_best_weights= True,
                              verbose = 0)

reduce_learning_rate = ReduceLROnPlateau(monitor='val_accuracy',
                                         patience = 2,
                                         factor=0.5 ,
                                         verbose = 1)
```

In [13]:

```
#Compile the model
model.compile(optimizer='adam', loss = 'categorical_crossentropy' ,
metrics=['accuracy'])
```

In [14]:

```
#Fit The model
asl_class = model.fit(train_data,
                      validation_data= val_data,
                      epochs=30,
                      callbacks=[early_stoping,reduce_learning_rate],
                      verbose = 1)
Epoch 1/30
63/63 [==============================] - 268s 4s/step - loss: 2.1558 -
accuracy: 0.4210 - val_loss: 0.5963 - val_accuracy: 0.8048 - lr: 0.0010
Epoch 2/30
63/63 [==============================] - 256s 4s/step - loss: 0.5794 -
accuracy: 0.8211 - val_loss: 0.2109 - val_accuracy: 0.9283 - lr: 0.0010
Epoch 3/30
63/63 [==============================] - 254s 4s/step - loss: 0.3259 -
accuracy: 0.8907 - val_loss: 0.1349 - val_accuracy: 0.9602 - lr: 0.0010
```

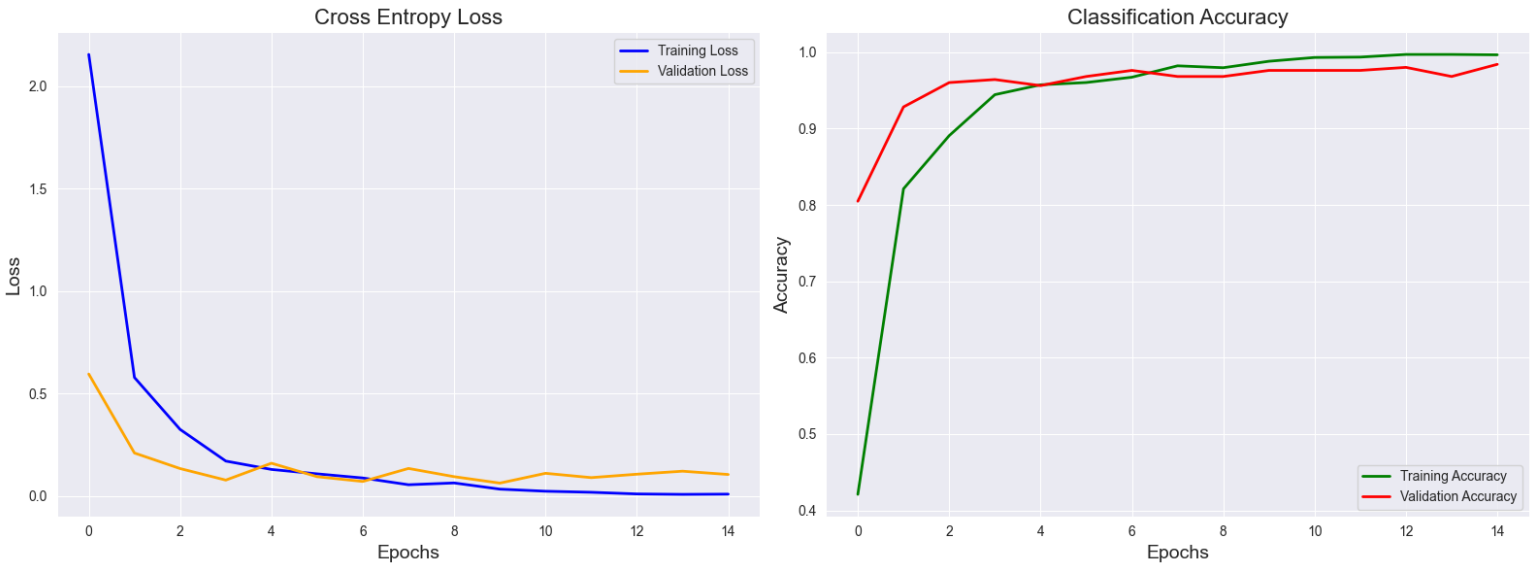
```
Epoch 4/30
63/63 [==============================] - 250s 4s/step - loss: 0.1715 -
accuracy: 0.9443 - val_loss: 0.0786 - val_accuracy: 0.9641 - lr: 0.0010
Epoch 5/30
63/63 [==============================] - 257s 4s/step - loss: 0.1306 -
accuracy: 0.9573 - val_loss: 0.1610 - val_accuracy: 0.9562 - lr: 0.0010
Epoch 6/30
63/63 [==============================] - 249s 4s/step - loss: 0.1091 -
accuracy: 0.9602 - val_loss: 0.0952 - val_accuracy: 0.9681 - lr: 0.0010
Epoch 7/30
63/63 [==============================] - 250s 4s/step - loss: 0.0889 -
accuracy: 0.9672 - val_loss: 0.0720 - val_accuracy: 0.9761 - lr: 0.0010
Epoch 8/30
63/63 [==============================] - 249s 4s/step - loss: 0.0560 -
accuracy: 0.9821 - val_loss: 0.1354 - val_accuracy: 0.9681 - lr: 0.0010
Epoch 9/30
63/63 [==============================] - ETA: 0s - loss: 0.0647 - accuracy:
0.9796
Epoch 9: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
63/63 [==============================] - 215s 3s/step - loss: 0.0647 -
accuracy: 0.9796 - val_loss: 0.0953 - val_accuracy: 0.9681 - lr: 0.0010
Epoch 10/30
63/63 [==============================] - 202s 3s/step - loss: 0.0345 -
accuracy: 0.9881 - val_loss: 0.0639 - val_accuracy: 0.9761 - lr: 5.0000e-04
Epoch 11/30
63/63 [==============================] - ETA: 0s - loss: 0.0243 - accuracy:
0.9930
Epoch 11: ReduceLROnPlateau reducing learning rate to
0.0002500000118743628.
63/63 [==============================] - 206s 3s/step - loss: 0.0243 -
accuracy: 0.9930 - val_loss: 0.1118 - val_accuracy: 0.9761 - lr: 5.0000e-04
Epoch 12/30
63/63 [==============================] - 206s 3s/step - loss: 0.0192 -
accuracy: 0.9935 - val_loss: 0.0907 - val_accuracy: 0.9761 - lr: 2.5000e-04
Epoch 13/30
63/63 [==============================] - 212s 3s/step - loss: 0.0111 -
accuracy: 0.9970 - val_loss: 0.1074 - val_accuracy: 0.9801 - lr: 2.5000e-04
Epoch 14/30
63/63 [==============================] - 216s 3s/step - loss: 0.0089 -
accuracy: 0.9970 - val_loss: 0.1221 - val_accuracy: 0.9681 - lr: 2.5000e-04
Epoch 15/30
63/63 [==============================] - 209s 3s/step - loss: 0.0101 -
accuracy: 0.9965 - val_loss: 0.1060 - val_accuracy: 0.9841 - lr: 2.5000e-04
```

In [15]:

```
# Evaluation
# Evaluate for train generator
loss,acc = model.evaluate(train_data , verbose = 0)

print('The accuracy of the model for training data is:',acc*100)
print('The Loss of the model for training data is:',loss)

# Evaluate for validation generator
loss,acc = model.evaluate(val_data, verbose = 0)

print('The accuracy of the model for validation data is:',acc*100)
print('The Loss of the model for validation data is:',loss)
The accuracy of the model for training data is: 99.80119466781616
The Loss of the model for training data is: 0.003659179899841547
The accuracy of the model for validation data is: 97.60956168174744
The Loss of the model for validation data is: 0.06394387036561966
```

In [20]:

```
# plots for accuracy and Loss with epochs

# Create a figure with subplots
plt.figure(figsize=(16, 6))

# Subplot 1: Cross Entropy Loss
plt.subplot(1, 2, 1)
plt.plot(error['loss'], label='Training Loss', color='blue', linewidth=2)
plt.plot(error['val_loss'], label='Validation Loss', color='orange',
linewidth=2)
plt.title('Cross Entropy Loss', fontsize=16)
plt.xlabel('Epochs', fontsize=14)
plt.ylabel('Loss', fontsize=14)
plt.legend()
plt.grid(True)

# Subplot 2: Classification Accuracy
plt.subplot(1, 2, 2)
plt.plot(error['accuracy'], label='Training Accuracy', color='green',
linewidth=2)
plt.plot(error['val_accuracy'], label='Validation Accuracy', color='red',
linewidth=2)
plt.title('Classification Accuracy', fontsize=16)
plt.xlabel('Epochs', fontsize=14)
plt.ylabel('Accuracy', fontsize=14)
plt.legend()
plt.grid(True)

# Adjust spacing between subplots
plt.tight_layout()

# Show the plot
plt.show()
```



In [26]:

```
# Perform predictions on the test dataset

# Predict using the trained model on the test data
result = model.predict(test_data, verbose=0)
```

```
# Extract the predicted class labels by selecting the class with the
highest probability
y_pred = np.argmax(result, axis=1)

# Get the true class labels from the test data generator
y_true = test_data.labels

# Evaluate the model on the test dataset

# Evaluate the model's performance on the test data and retrieve loss and
accuracy
loss, acc = model.evaluate(test_data, verbose=0)

# Print the accuracy and loss metrics for the testing data
print('The accuracy of the model for testing data is:', acc * 100)
print('The Loss of the model for testing data is:', loss)
```

```
The accuracy of the model for testing data is: 96.82539701461792
The Loss of the model for testing data is: 0.1789255142211914
```

In [27]:

```
# Calculate the number of correct and incorrect predictions

# Predicted class labels
p = y_pred
# True class labels
t = y_true

# Indices where predictions match true labels (correct predictions)
correct = np.nonzero(p == t)[0]

# Indices where predictions do not match true labels (incorrect
predictions)
incorrect = np.nonzero(p != t)[0]

# Print the number of correct and incorrect predictions
print("Correctly Predicted Classes:", correct.shape[0])
print("Incorrectly Predicted Classes:", incorrect.shape[0])
```

```
Correctly Predicted Classes: 244
Incorrectly Predicted Classes: 8
```

In [30]:

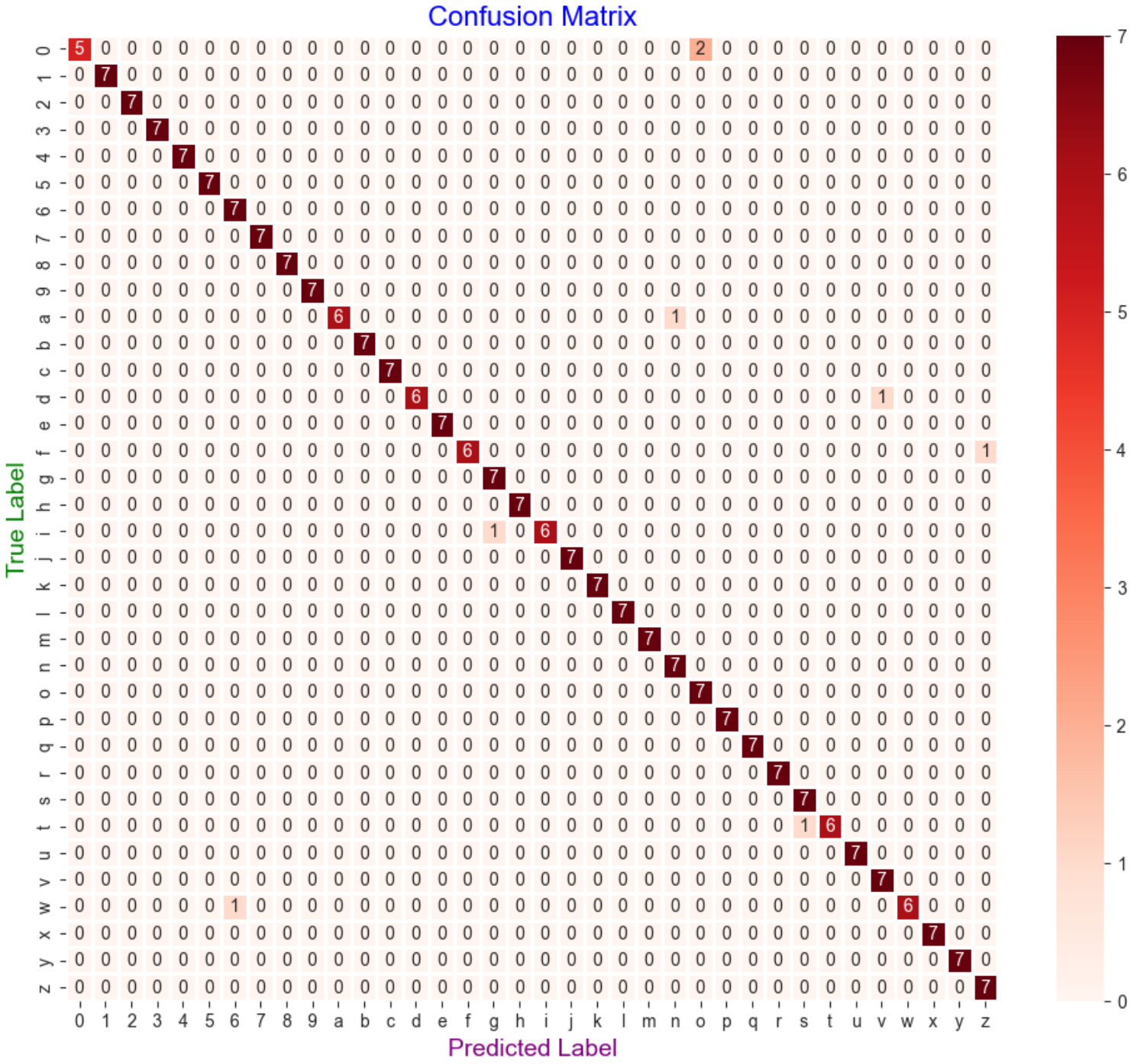
```
# Generate a classification report to assess model performance

# `y_true` represents the true class labels
# `y_pred` represents the predicted class labels
# `target_names` is used to specify the names of the classes/categories

print(classification_report(y_true, y_pred,
target_names=categories.values()))
```

```
              precision    recall  f1-score   support

           0       1.00      0.71      0.83         7
           1       1.00      1.00      1.00         7
           2       1.00      1.00      1.00         7
           3       1.00      1.00      1.00         7
           4       1.00      1.00      1.00         7
           5       1.00      1.00      1.00         7
           6       0.88      1.00      0.93         7
           7       1.00      1.00      1.00         7
           8       1.00      1.00      1.00         7
```

```
              9       1.00      1.00      1.00         7
              a       1.00      0.86      0.92         7
              b       1.00      1.00      1.00         7
              c       1.00      1.00      1.00         7
              d       1.00      0.86      0.92         7
              e       1.00      1.00      1.00         7
              f       1.00      0.86      0.92         7
              g       0.88      1.00      0.93         7
              h       1.00      1.00      1.00         7
              i       1.00      0.86      0.92         7
              j       1.00      1.00      1.00         7
              k       1.00      1.00      1.00         7
              l       1.00      1.00      1.00         7
              m       1.00      1.00      1.00         7
              n       0.88      1.00      0.93         7
              o       0.78      1.00      0.88         7
              p       1.00      1.00      1.00         7
              q       1.00      1.00      1.00         7
              r       1.00      1.00      1.00         7
              s       0.88      1.00      0.93         7
              t       1.00      0.86      0.92         7
              u       1.00      1.00      1.00         7
              v       0.88      1.00      0.93         7
              w       1.00      0.86      0.92         7
              x       1.00      1.00      1.00         7
              y       1.00      1.00      1.00         7
              z       0.88      1.00      0.93         7

    accuracy                              0.97       252
   macro avg       0.97      0.97      0.97       252
weighted avg       0.97      0.97      0.97       252
```

In [36]:

```
# Create a figure and axis for the heatmap
f, ax = plt.subplots(figsize=(12, 10))

# Create the heatmap using seaborn
sns.heatmap(confusion_mtx, annot=True,
            linewidths=1, cmap="Reds",
            fmt='.0f', ax=ax,
            cbar=True, xticklabels=categories.values(),
            yticklabels=categories.values())

# Set labels and title for the plot
plt.xlabel("Predicted Label", fontdict={'color': 'purple', 'size': 14})
plt.ylabel("True Label", fontdict={'color': 'green', 'size': 14})
plt.title("Confusion Matrix", fontdict={'color': 'blue', 'size': 16})

# Customize color bar
cbar = ax.collections[0].colorbar
cbar.set_label('Count', rotation=270, labelpad=20)

# Show the plot
plt.show()
```



In [40]:

```
from tensorflow.keras.applications import VGG16
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

# Load the pre-trained model with weights
base_model = VGG16(weights='imagenet', include_top=False,
input_shape=(image_size, image_size, img_channel))

# Add a custom output layer for your specific task
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu')(x)
predictions = Dense(n_classes, activation='softmax')(x)

# Create the transfer learning model
model = Model(inputs=base_model.input, outputs=predictions)
```

```
# Define the number of epochs
epochs = 10

# Define callbacks
early_stopping = EarlyStopping(monitor='val_loss', patience=5,
restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=2,
min_lr=1e-7)

# Compile the model
model.compile(optimizer=Adam(learning_rate=0.0001),
loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model on your ASL dataset
transfer_learning_history = model.fit(train_data, validation_data=val_data,
epochs=epochs, verbose=1, callbacks=[early_stopping, reduce_lr])

# Evaluate the model on validation and test datasets
val_loss, val_acc = model.evaluate(val_data)
test_loss, test_acc = model.evaluate(test_data)

# Compare and discuss the results with benchmark models
Epoch 1/10
63/63 [==============================] - 757s 12s/step - loss: 1.9310 -
accuracy: 0.4657 - val_loss: 0.4942 - val_accuracy: 0.8486 - lr: 1.0000e-04
Epoch 2/10
63/63 [==============================] - 726s 12s/step - loss: 0.3570 -
accuracy: 0.8743 - val_loss: 0.1719 - val_accuracy: 0.9482 - lr: 1.0000e-04
Epoch 3/10
63/63 [==============================] - 738s 12s/step - loss: 0.1936 -
accuracy: 0.9279 - val_loss: 0.1226 - val_accuracy: 0.9442 - lr: 1.0000e-04
Epoch 4/10
63/63 [==============================] - 741s 12s/step - loss: 0.1483 -
accuracy: 0.9503 - val_loss: 0.1669 - val_accuracy: 0.9442 - lr: 1.0000e-04
Epoch 5/10
63/63 [==============================] - 744s 12s/step - loss: 0.1109 -
accuracy: 0.9642 - val_loss: 0.0757 - val_accuracy: 0.9681 - lr: 1.0000e-04
Epoch 6/10
63/63 [==============================] - 748s 12s/step - loss: 0.0923 -
accuracy: 0.9667 - val_loss: 0.0521 - val_accuracy: 0.9801 - lr: 1.0000e-04
Epoch 7/10
63/63 [==============================] - 740s 12s/step - loss: 0.0655 -
accuracy: 0.9776 - val_loss: 0.1117 - val_accuracy: 0.9641 - lr: 1.0000e-04
Epoch 8/10
63/63 [==============================] - 741s 12s/step - loss: 0.0607 -
accuracy: 0.9756 - val_loss: 0.1437 - val_accuracy: 0.9482 - lr: 1.0000e-04
Epoch 9/10
63/63 [==============================] - 741s 12s/step - loss: 0.0285 -
accuracy: 0.9896 - val_loss: 0.0122 - val_accuracy: 0.9880 - lr: 5.0000e-05
Epoch 10/10
63/63 [==============================] - 751s 12s/step - loss: 0.0045 -
accuracy: 0.9990 - val_loss: 0.0112 - val_accuracy: 0.9960 - lr: 5.0000e-05
8/8 [==============================] - 24s 3s/step - loss: 0.0112 -
accuracy: 0.9960
8/8 [==============================] - 23s 3s/step - loss: 0.0176 -
accuracy: 0.9881
```

In [43]:

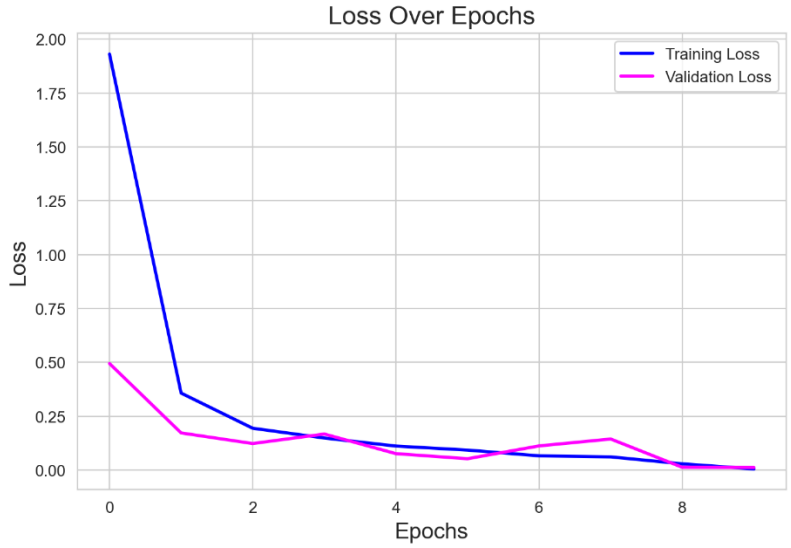
```
# Evaluation
# Evaluate for train generator
```

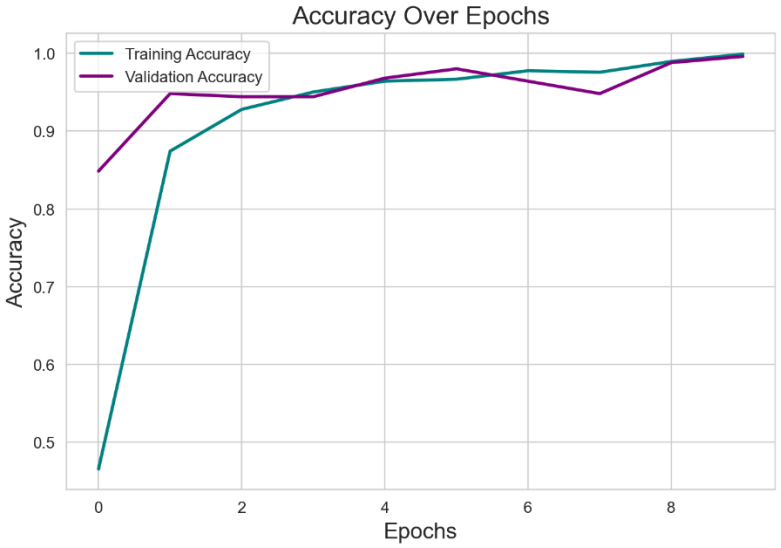
```
train_loss, train_acc = model.evaluate(train_data, verbose=0)
print('The accuracy of the model for training data is:', train_acc * 100)
print('The Loss of the model for training data is:', train_loss)

# Evaluate the model on validation dataset
print('The accuracy of the model for validation data is:', val_acc * 100)
print('The Loss of the model for validation data is:', val_loss)

# Evaluate the model on test dataset
print('The accuracy of the model for test data is:', test_acc * 100)
print('The Loss of the model for test data is:', test_loss)
```

```
The accuracy of the model for training data is: 100.0
The Loss of the model for training data is: 0.0022616726346313953
The accuracy of the model for validation data is: 99.60159659385681
The Loss of the model for validation data is: 0.01116271037608385
The accuracy of the model for test data is: 98.8095223903656
The Loss of the model for test data is: 0.01759868487715721
```

In [56]:

```
# Get training and validation loss and accuracy from history
train_loss = transfer_learning_history.history['loss']
val_loss = transfer_learning_history.history['val_loss']
train_acc = transfer_learning_history.history['accuracy']
val_acc = transfer_learning_history.history['val_accuracy']

# Create subplots for loss and accuracy
plt.figure(figsize=(14, 5), dpi=200)
plt.subplot(1, 2, 1)
plt.plot(train_loss, label='Training Loss', color='Blue', linewidth=2)
plt.plot(val_loss, label='Validation Loss',  color='Magenta', linewidth=2)
plt.title('Loss Over Epochs', fontsize=16)
plt.xlabel('Epochs', fontsize=14)
plt.ylabel('Loss', fontsize=14)
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(train_acc, label='Training Accuracy', color='Teal', linewidth=2)
plt.plot(val_acc, label='Validation Accuracy', color='Purple', linewidth=2)
plt.title('Accuracy Over Epochs', fontsize=16)
plt.xlabel('Epochs', fontsize=14)
plt.ylabel('Accuracy', fontsize=14)
plt.legend()

# Adjust spacing between subplots
plt.tight_layout()

plt.show()
```





In [48]:

```
# Predict labels for the test dataset using the VGG16 model
vgg16_predictions = model.predict(test_data, verbose=0)
vgg16_predicted_labels = np.argmax(vgg16_predictions, axis=1)

# Get the true labels for the test dataset
true_labels = test_data.classes

# Calculate the number of correct and incorrect predictions
correct_predictions = np.sum(vgg16_predicted_labels == true_labels)
incorrect_predictions = len(true_labels) - correct_predictions

print('Number of Correct Predictions:', correct_predictions)
print('Number of Incorrect Predictions:', incorrect_predictions)
```

```
Number of Correct Predictions: 249
Number of Incorrect Predictions: 3
```

In [49]:

```
# Generate a classification report
classification_rep = classification_report(true_labels,
vgg16_predicted_labels, target_names=categories.values())
print("Classification Report:\n", classification_rep)
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.75      0.86      0.80         7
           1       1.00      1.00      1.00         7
           2       1.00      1.00      1.00         7
           3       1.00      1.00      1.00         7
           4       1.00      1.00      1.00         7
           5       1.00      1.00      1.00         7
           6       1.00      1.00      1.00         7
           7       1.00      1.00      1.00         7
           8       1.00      1.00      1.00         7
           9       1.00      1.00      1.00         7
           a       1.00      1.00      1.00         7
           b       1.00      1.00      1.00         7
           c       1.00      1.00      1.00         7
           d       1.00      1.00      1.00         7
           e       1.00      1.00      1.00         7
           f       1.00      1.00      1.00         7
           g       1.00      1.00      1.00         7
           h       1.00      1.00      1.00         7
           i       1.00      1.00      1.00         7
           j       1.00      1.00      1.00         7
           k       1.00      1.00      1.00         7
           l       1.00      1.00      1.00         7
           m       1.00      1.00      1.00         7
           n       1.00      1.00      1.00         7
           o       0.83      0.71      0.77         7
           p       1.00      1.00      1.00         7
           q       1.00      1.00      1.00         7
           r       1.00      1.00      1.00         7
           s       1.00      1.00      1.00         7
           t       1.00      1.00      1.00         7
           u       1.00      1.00      1.00         7
           v       1.00      1.00      1.00         7
           w       1.00      1.00      1.00         7
           x       1.00      1.00      1.00         7
           y       1.00      1.00      1.00         7
```

```
           z       1.00      1.00      1.00         7

    accuracy                           0.99       252
   macro avg       0.99      0.99      0.99       252
weighted avg       0.99      0.99      0.99       252
```

In [61]:

```
# Generate a confusion matrix
confusion_mtx = confusion_matrix(true_labels, vgg16_predicted_labels)

# Create a heatmap for the confusion matrix
plt.figure(figsize=(12, 10), dpi=200)
sns.set_style('whitegrid')
sns.heatmap(confusion_mtx, annot=True, linewidths=1, fmt='d',
cmap='YlGnBu', xticklabels=categories.values(),
yticklabels=categories.values())
plt.xlabel('Predicted Labels', fontsize=14)
plt.ylabel('True Labels', fontsize=14)
plt.title('Confusion Matrix', fontsize=16)
plt.show()
```