Deep Learning-Based American Sign Language Recognition: A Comprehensive Case Study on Digit Classification Using Convolutional Neural Networks

Bryan Tineo, Aiden Hock

Professor: Wenlu Zhang

California State University Long Beach

Github Repo: https://github.com/bryanmax9/SignLangClassifier

Abstract

This project explores the development and comparative evaluation of deep learning models for the classification of American Sign Language (ASL) hand gestures into corresponding alphabetical letters (A-Z) and numerical digits (0-9). The aim is to design and assess the performance of two distinct machine learning approaches: a scratch-built model and a fine-tuned pretrained model. The scratch-built model is developed from the ground up, using custom architectural design and trained exclusively on a curated dataset of 35 ASL gesture classes. In contrast, the fine-tuned model leverages transfer learning by adapting a pretrained model to the same ASL dataset, optimizing its performance for this specific task.

This study involves a detailed analysis of the models' design, training, and evaluation, with metrics such as accuracy, precision, recall, and F1-score used to measure their performance. Experimental results reveal the advantages and limitations of each approach, offering insights into the trade-offs between building a model from scratch and fine-tuning a pretrained architecture. The findings provide a comparative understanding of how different methodologies impact the efficiency and accuracy of ASL classification, contributing to the broader field of sign language recognition and its potential applications in assistive technologies.

Introduction

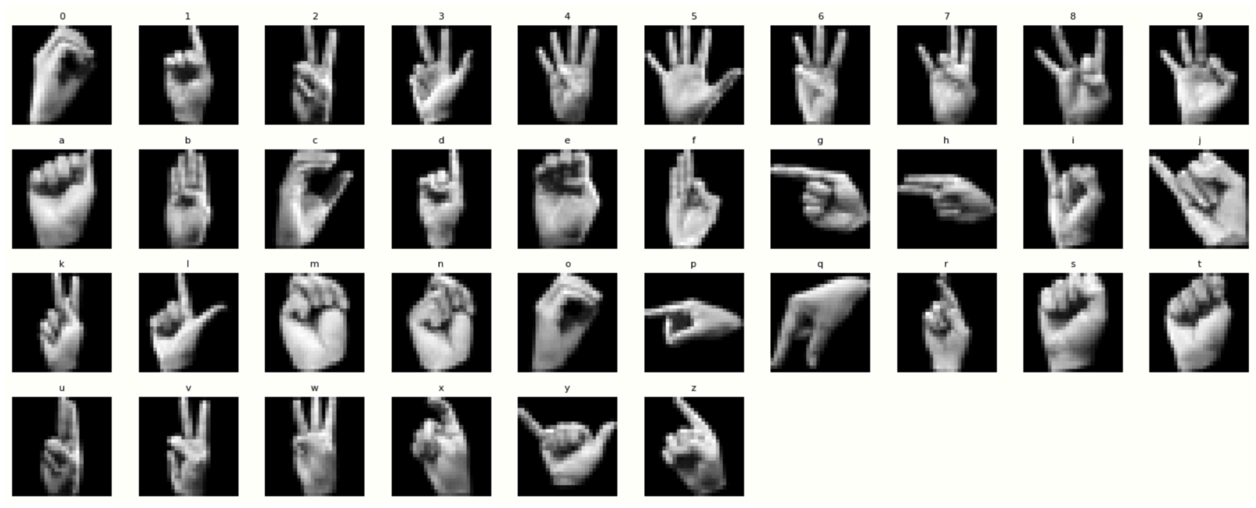
Sign language is a vital mode of communication for individuals who are deaf or hard of hearing, bridging the gap between spoken language and visual gestures. American Sign Language (ASL), widely used in North America, includes distinct hand gestures representing alphabetical letters and numerical digits. This project explores the design and performance analysis of machine learning models aimed at translating ASL gestures into corresponding alphanumeric characters, enabling improved accessibility and communication. The objective of this study is to develop and compare two distinct deep learning models for ASL gesture classification: a scratch-built model and a fine-tuned pretrained model. The scratch model is designed and trained exclusively on a custom dataset comprising 35 classes, representing the numbers 0-9 and the letters A-Z. This approach emphasizes the importance of carefully constructing a neural network architecture tailored to the task, employing layers such as convolution, pooling, and dropout, alongside optimization techniques like the Adam algorithm to enhance model performance. In contrast, the fine-tuned model leverages transfer learning, adapting a pretrained network to the same dataset to evaluate its performance relative to the scratch-built model.

This report outlines the process of developing these models, starting with the setup and preparation of the dataset, followed by the architectural design of the scratch model and the fine-tuning of the pretrained model. The report delves into the experimental setup, including data preprocessing, model training, and evaluation metrics such as accuracy, precision, recall, and F1-score. By comparing the two approaches, this study aims to provide insights into the trade-offs between designing models from scratch and utilizing pretrained networks, contributing to the broader application of deep learning in sign language recognition and accessibility technologies.

Scratch Model Outline

The starting outline for our baseline model in this project focuses on utilizing foundational concepts in TensorFlow, which we learned during assignments under the guidance of Professor Wenlu Zhang. Our objective was to create a simple, yet effective, model to classify American Sign Language (ASL) hand gestures into corresponding numerical and alphabetical categories. To achieve this, we used grayscale image processing and focused on constructing a minimalistic pipeline that aligns with our coursework experience. This approach allowed us to apply the theoretical concepts from class to a practical, real-world problem in a straightforward manner.

The dataset preparation began with defining the path to the ASL image dataset stored in Google Drive. Using the `os` module, we ensured that the directory containing class folders (e.g., "0", "1", "a", "b", etc.) was correctly accessed. Each folder represented a specific class, and its contents were image files corresponding to that class. We looped through these folders systematically, verifying their existence and iterating over each file to extract images. This methodical traversal ensured no data was missed, and any issues with missing or corrupted files were logged for review.



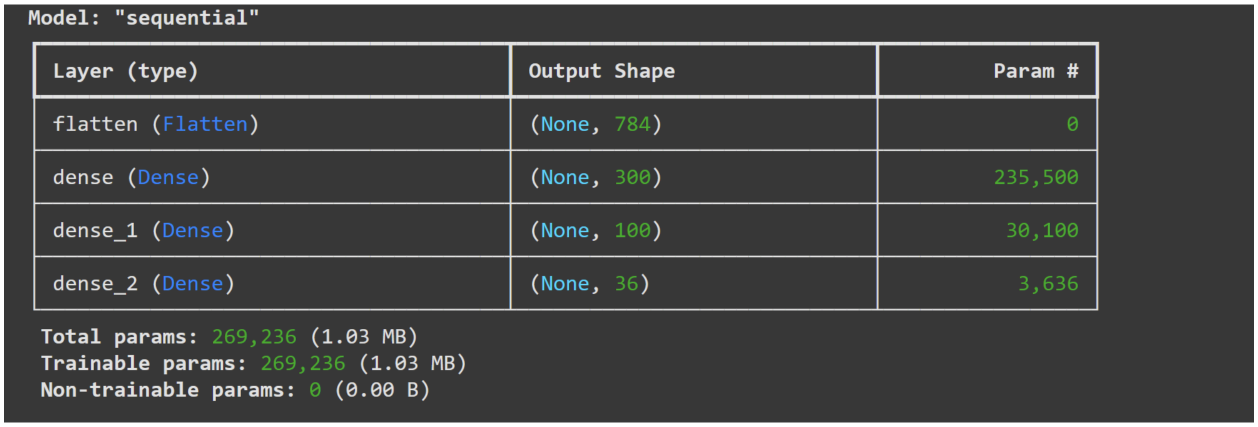
Once the images were accessed, we employed the Python Imaging Library (PIL) to preprocess them. Each image was converted to grayscale using the "L" mode in PIL, which effectively reduces the computational complexity by eliminating color channels while retaining essential structural information. The images were then resized to a uniform dimension of 28x28 pixels. This resizing step served two purposes: it standardized input dimensions for the neural network and reduced the memory footprint, making the model more efficient for training and inference.

After preprocessing, the images were normalized by scaling their pixel values to the range [0, 1]. This normalization was crucial for ensuring consistent input data distribution, as neural networks tend to perform better when input features are normalized. The normalized images were stored in a NumPy array, which allowed for efficient manipulation and compatibility with TensorFlow. Alongside the image data, the corresponding class labels (derived from folder names) were also stored in an array, forming a complete dataset of inputs and outputs.

With the dataset prepared, we split it into training, validation, and testing subsets. We followed the standard practice of allocating 70% of the data for training, with the remaining 30% evenly divided between validation and testing. This splitting process was performed using train_test_split from the sklearn library, which ensured a randomized but reproducible division of the data. The resulting subsets were verified for shape and size, confirming that each contained images of dimension 28x28 with a single grayscale channel.

For visualization, we displayed one sample image from each class using matplotlib. This step was not merely for aesthetics but served as an important validation of the dataset integrity and preprocessing pipeline. It allowed us to confirm that the images were correctly resized, normalized, and labeled. The grid-like layout of images provided an intuitive overview of the dataset and reinforced our confidence in its readiness for training.

The neural network itself was designed as a sequential model in TensorFlow. Starting with a flattening layer, the 28x28 grayscale images were converted into a one-dimensional array of 784 pixels. This flattening step bridged the gap between the spatial representation of the images and the fully connected layers that followed. The first hidden layer consisted of 300 neurons with ReLU activation, which introduced non-linearity and enabled the network to learn complex features. A second hidden layer, with 100 neurons and ReLU activation, further refined these learned features.



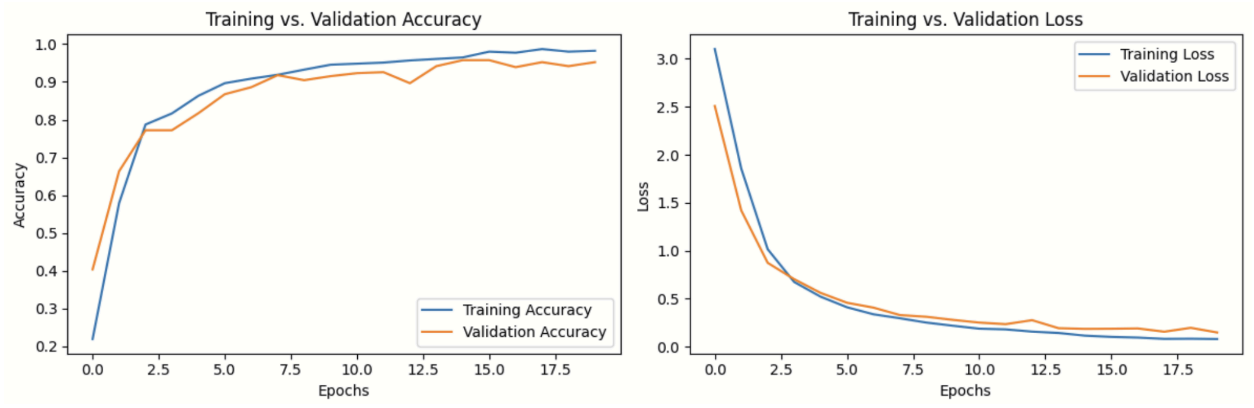
```
Model: "sequential"
┌──────────────────────────────────────┬─────────────────────────────┬─────────────────┐
│ Layer (type)                         │ Output Shape                │         Param # │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ flatten (Flatten)                    │ (None, 784)                 │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense (Dense)                        │ (None, 300)                 │         235,500 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_1 (Dense)                      │ (None, 100)                 │          30,100 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_2 (Dense)                      │ (None, 36)                  │           3,636 │
└──────────────────────────────────────┴─────────────────────────────┴─────────────────┘
 Total params: 269,236 (1.03 MB)
 Trainable params: 269,236 (1.03 MB)
 Non-trainable params: 0 (0.00 B)
```

The output layer was designed with 36 neurons (one for each class, including numbers 0-9 and letters a-z), using the softmax activation function. Softmax converted the raw outputs into probabilities, ensuring that the sum of probabilities across all classes equaled 1. This made it easy to interpret the network's predictions, as the class with the highest probability could be directly taken as the output.

We compiled the model using sparse categorical cross-entropy as the loss function, which is well-suited for multi-class classification problems. The Adam optimizer was chosen for its efficiency and adaptability during weight updates. To monitor the training process, we included accuracy as an evaluation metric. These choices were guided by our class assignments, where similar configurations yielded reliable results.

Finally, the model was trained on the dataset for 20 epochs with a batch size of 100. During training, validation data was used to monitor the model's performance and prevent overfitting. Post-training, the model was evaluated on the test dataset to assess its generalization capability.

The results, including training and validation accuracy and loss plots, were visualized to provide a comprehensive understanding of the model's learning dynamics.



In summary, this baseline model represents a fundamental approach to solving the ASL classification problem. By adhering to the concepts and techniques taught in class, we constructed a clear and logical pipeline, starting from data preprocessing and ending with model evaluation. This base model will essentially serve as a starting point for improvements, which will be covered in the next section.

Final Scratch Model

While the outline model's performance wasn't bad, we improved the model based on the recommendations of our professor Wenlu Zhang. The transition from initial design to final implementation showcased substantial advancements in dataset handling, architectural refinement, and training strategies, aligning the model with both industry best practices and Professor Zhang's recommendations.

Transitioning to PyTorch introduced a streamlined and efficient training workflow, capitalizing on its dynamic computational graph and modular architecture. This transition simplified the development process, allowing for seamless experimentation with various architectural configurations. A robust training loop was implemented, featuring real-time metrics computation to track progress and detect issues during training. Learning rate adjustments were automated using a step scheduler, optimizing convergence rates and minimizing loss oscillations.

Leveraging PyTorch's seamless GPU integration significantly accelerated the training process, enabling the efficient handling of large datasets and facilitating iterative experimentation. These optimizations streamlined the development cycle, allowing for rapid model iterations while minimizing resource constraints.

The dataset preparation process began with an incremental and dynamic class selection strategy, incorporating lessons from industry practices and academic recommendations. Initially, we focused on the numeric classes (0-9), gradually extending the scope to alphabetic characters (A-Z). This approach was strategically designed to examine how dataset complexity impacts model performance as the number of classes increases. Our flexible and dynamic codebase allowed for seamless specification of the number of classes during training, making it an essential tool for systematically conducting this analysis.

To improve accuracy, we incorporated Dr. Wenlu Zhang's recommendations to utilize RGB image datasets and integrate data augmentation. Data augmentation techniques such as random rotations, horizontal flips, and affine transformations introduced variability to the dataset, effectively mitigating underfitting. These transformations enhanced dataset diversity, mitigating the risk of underfitting and improving the model's ability to generalize. The switch to RGB images also provided richer feature sets, further boosting model performance.

We initiated the analysis by training the model on three classes, achieving a notable accuracy of 95%. As the class count increased to six, accuracy declined marginally to 93%. Scaling up to ten classes revealed a gradual reduction in accuracy, confirming our hypothesis that increasing class complexity impacts the model's generalization ability. When the class count reached 18, the drop in accuracy became marginal, indicating that the effect of additional classes diminishes beyond a certain threshold.

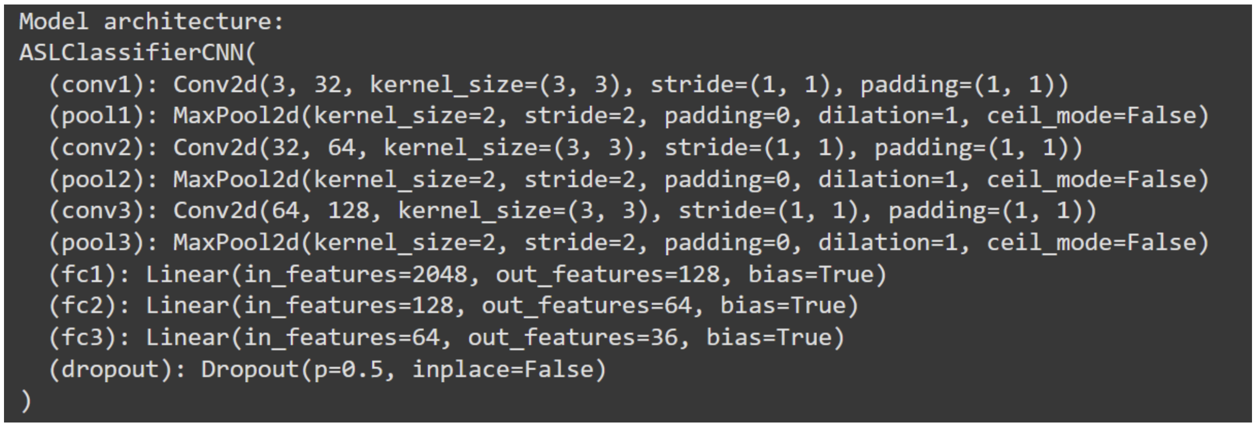
We also improved the model architecture to align with industry standards. Inspired by VGGNet, we added additional convolutional and pooling layers to deepen the network and enhance feature extraction capabilities. Dropout layers were introduced to reduce overfitting, and the fully connected layers were fine-tuned for optimal performance. Furthermore, the dataset splitting strategy was refined to ensure a balanced division between training, validation, and testing sets. This setup provided a robust framework for evaluating the model's generalization capabilities.

The findings underscored the value of incremental training and dynamic dataset preparation. The steady decline in accuracy with added classes, followed by stabilization beyond 18 classes, emphasized the importance of a structured class expansion strategy. Combining these insights with rigorous preprocessing, model regularization, and incremental validation cycles, we achieved a balance between bias and variance, yielding a model that generalizes effectively across diverse class distributions.

One of the standout features of our implementation was the introduction of a dynamic training pipeline, which allowed us to specify the number of classes for training. This flexibility proved instrumental in investigating the relationship between dataset complexity and model accuracy. By incrementally increasing the number of classes starting with numeric digits (0-9) and eventually incorporating alphabetic characters (A-Z) we observed a consistent and predictable pattern. As expected, accuracy decreased slightly with the addition of new classes due to the increasing complexity of the classification task. However, after surpassing 18 classes, the rate of accuracy decline diminished significantly, stabilizing around this threshold.

This incremental approach mirrors real-world scenarios where datasets expand over time, requiring models to adapt dynamically without compromising performance. The ability to train on subsets of classes and seamlessly integrate new ones ensured a flexible workflow.

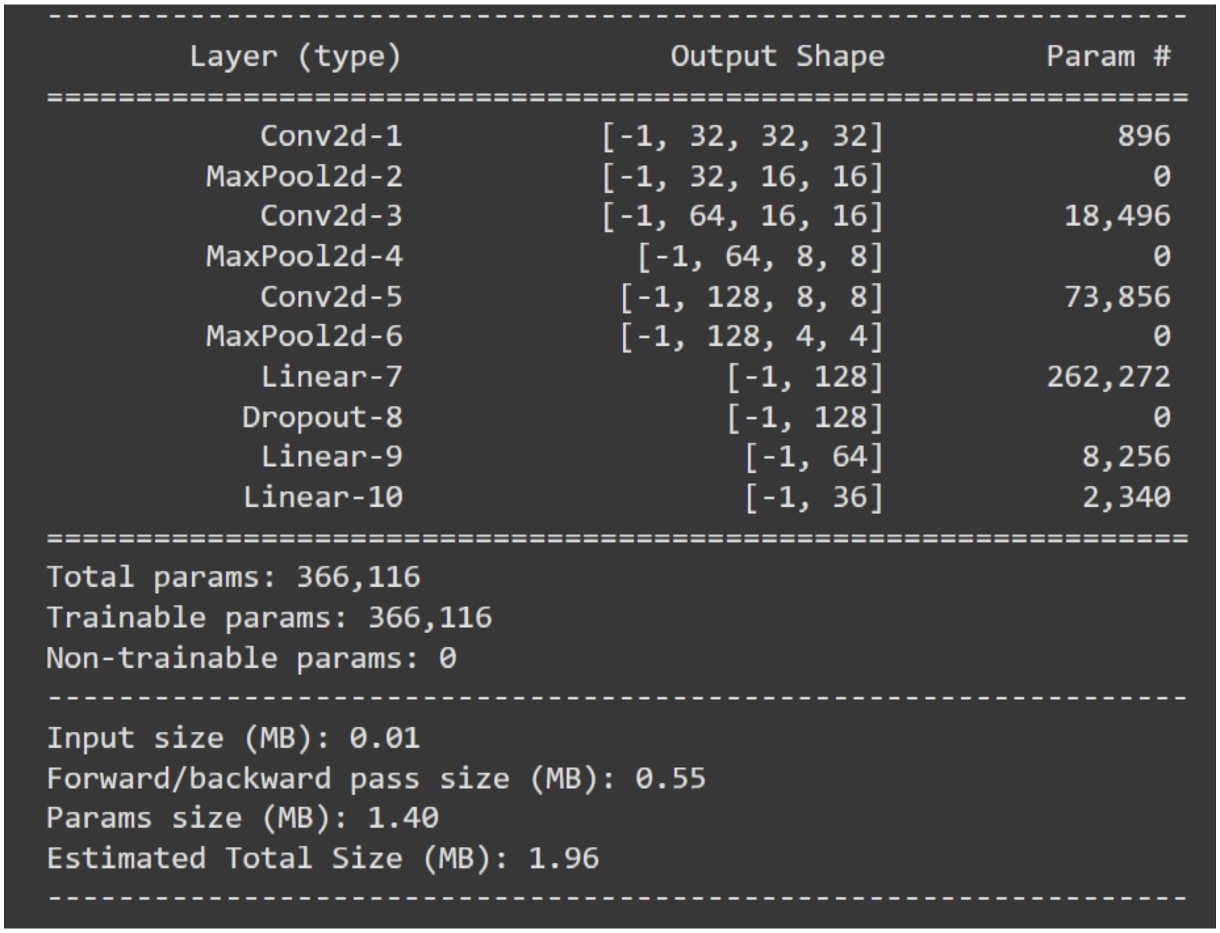
Additionally, this strategy provided key insights into the trade-offs between accuracy and scalability, equipping the model to handle diverse class distributions in future iterations.


```
Model architecture:
ASLClassifierCNN(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=2048, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=36, bias=True)
  (dropout): Dropout(p=0.5, inplace=False)
)
```

The refinement of the model architecture was another pivotal advancement in this project. Drawing inspiration from the VGGNet framework, we incorporated a series of convolutional and pooling layers to build a robust hierarchical feature extractor. The architecture was designed to progressively capture increasingly complex features, beginning with basic edges and textures and advancing to intricate patterns.

The model comprised three convolutional blocks, each with an increasing number of filters (32, 64, 128), to enhance feature extraction capacity. Each block was followed by a max-pooling layer, which reduced the spatial dimensions while preserving essential features, thereby lowering computational complexity without sacrificing accuracy. The use of ReLU activation functions in these layers introduced non-linearity, enabling the model to learn complex patterns effectively.

To prevent overfitting, we integrated dropout regularization after the first fully connected layer, randomly deactivating 50% of the neurons during training. This strategy forced the network to rely on a broader set of features, improving its generalization ability. Additionally, the fully connected layers were tailored to handle the variable number of classes dynamically, ensuring compatibility with the incremental training strategy.

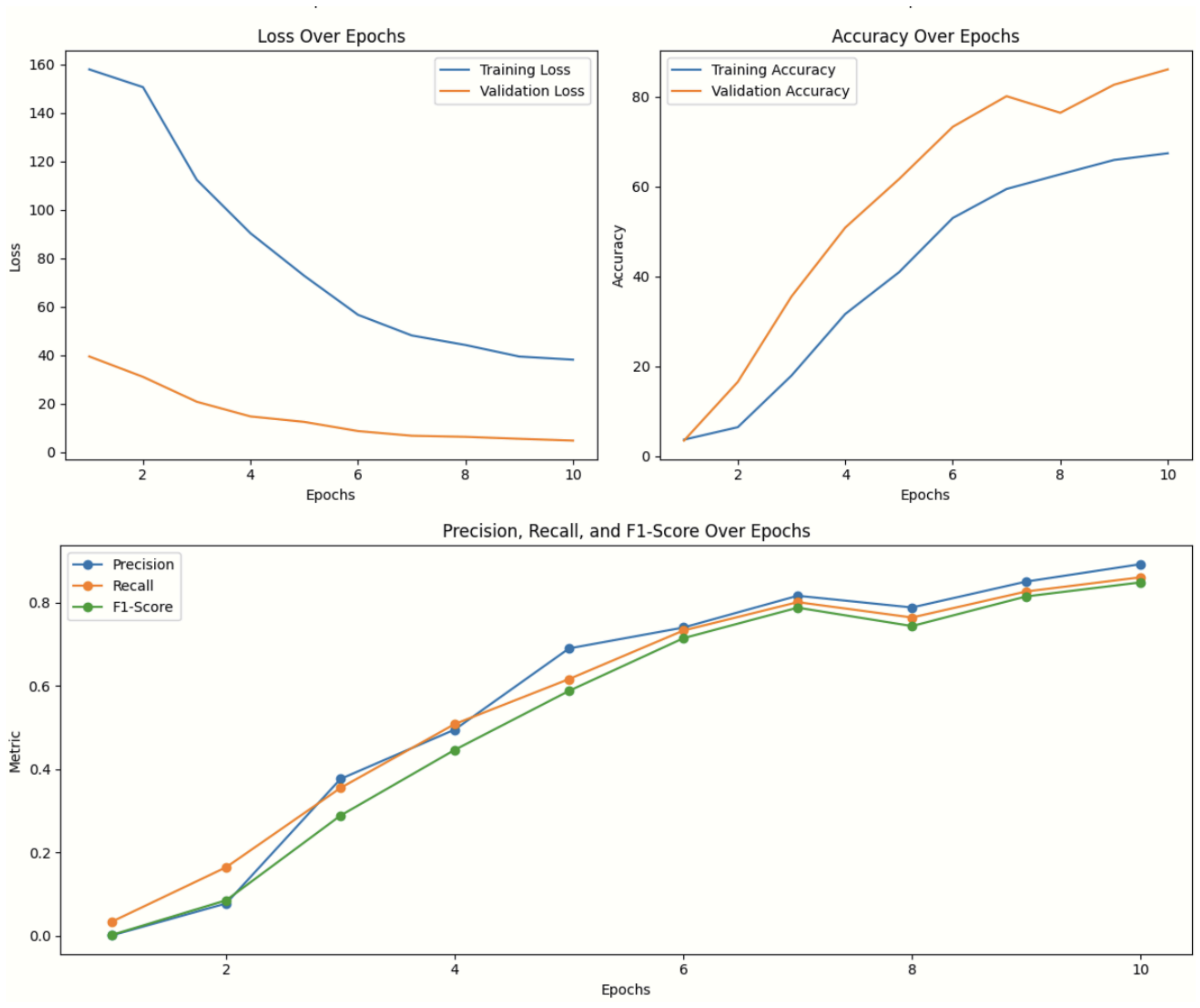


```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 32, 32]             896
         MaxPool2d-2           [-1, 32, 16, 16]               0
            Conv2d-3           [-1, 64, 16, 16]          18,496
         MaxPool2d-4             [-1, 64, 8, 8]               0
            Conv2d-5            [-1, 128, 8, 8]          73,856
         MaxPool2d-6            [-1, 128, 4, 4]               0
            Linear-7                  [-1, 128]         262,272
           Dropout-8                  [-1, 128]               0
            Linear-9                   [-1, 64]           8,256
           Linear-10                   [-1, 36]           2,340
================================================================
Total params: 366,116
Trainable params: 366,116
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 0.55
Params size (MB): 1.40
Estimated Total Size (MB): 1.96
----------------------------------------------------------------
```

The enhanced architecture not only aligned with industry standards but also demonstrated exceptional performance in capturing nuanced features and scaling efficiently with increasing dataset complexity. This iterative process of model refinement, paired with dynamic training, established a versatile and high-performing framework suitable for real-world applications.

To ensure robust and reliable training, we adhered to industry-standard data splitting practices by partitioning the dataset into three distinct subsets, 70% for training, 15% for validation, and 15% for testing. This approach guarantees that the model is trained, validated, and tested on mutually exclusive data, thereby preventing data leakage and ensuring the integrity of the evaluation process.

Furthermore, we employed a cross-validation strategy to enhance the robustness of the model. Cross-validation, achieved by dividing the dataset into multiple folds and iteratively training and validating on distinct subsets, offered a thorough evaluation of the model's generalization performance. This technique helped mitigate the risk of overfitting or underfitting to specific data subsets, ensuring that the model performs consistently across diverse data distributions. Furthermore, this methodical evaluation process was instrumental in fine-tuning hyperparameters and optimizing the overall model performance.



Visualization played a critical role, acting as a diagnostic tool to identify performance bottlenecks and guide architectural refinements. Metrics like precision, recall, F1-score, and confusion matrices offered detailed insights into the model's classification accuracy. Confusion matrices were especially useful in pinpointing misclassifications, informing targeted optimizations.

Training and validation losses and accuracies were plotted over epochs to monitor the model's learning trajectory and detect bias-variance issues. A balanced model achieves high performance on both training and validation datasets with minimal divergence between the two metrics. These visualizations allowed us to identify scenarios where the model was either overfitting or underfitting and make necessary adjustments, such as refining dropout rates or tweaking convolutional filters.

These insights drove iterative improvements in the architecture, ensuring that the final model not only achieved high accuracy but also exhibited balanced generalization capabilities, making it well-suited for real-world applications. By leveraging visual analytics, we effectively bridged the gap between model design and performance optimization.

A meticulous bias-variance analysis was conducted to fine-tune the model and achieve a balance between underfitting and overfitting. Instances of low training and validation accuracies highlighted underfitting, signaling that the model lacked complexity to capture underlying patterns in the data. Conversely, a pronounced gap between training and validation accuracies indicated overfitting, where the model performed well on the training data but struggled to generalize to unseen samples.

To address these issues, regularization techniques such as weight decay and dropout were systematically optimized. Weight decay penalized overly large weights, preventing the model from fitting noise in the data, while dropout randomly deactivated neurons during training, reducing the model's reliance on specific features. Graphical analysis of training and validation losses and accuracies revealed insights into the model's bias-variance tradeoff, enabling informed adjustments to hyperparameters for optimal performance.

Ultimately, the model achieved a near-optimal balance between bias and variance, with validation accuracy closely mirroring training accuracy. This equilibrium ensured the model’s ability to generalize effectively while maintaining high performance, making it robust and reliable for real-world applications.

A comprehensive bias-variance analysis was pivotal in refining the model. Underfitting was identified when both training and validation accuracies were consistently low, while overfitting became evident through a significant divergence between the two metrics. To address these issues, regularization techniques were optimized, including weight decay to penalize large weights and dropout to introduce stochasticity during training.

The use of graphical plots to monitor training and validation losses and accuracies provided actionable insights. These visual diagnostics ensured that the model achieved a balanced state, where validation accuracy closely followed training accuracy. This balance minimized the risk of both overfitting and underfitting, resulting in a robust model capable of generalizing effectively to unseen data.

While the project primarily focused on building a custom architecture from scratch, we explored the potential of transfer learning as a comparative approach, as recommended by Professor Zhang. Transfer learning involves leveraging pretrained models, such as VGGNet or ResNet, and fine-tuning them for specific tasks like ASL classification. This method promises faster training times and potentially higher baseline accuracies by transferring knowledge from general-purpose datasets. Though not fully implemented in this iteration, transfer learning remains a compelling avenue for future exploration to benchmark its performance against our custom architecture.

The final model reflects iterative refinements and adherence to industry standards. By tackling challenges like dataset variability, scalability, and architectural robustness, it is well-equipped for practical applications. Guided by Professor Zhang's recommendations, the project highlights the importance of incremental class handling, data augmentation, and bias-variance analysis. These principles ensure the model's adaptability and reliability in real-world scenarios, such as translating ASL gestures into textual or verbal outputs.

With its solid foundation, the model stands as both an academic milestone and a practical solution for ASL classification, paving the way for deployment in assistive technologies and communication tools.