American Sign Language Detection Design & Implementation

Rishabh Malik, Aleena Varghese

Group 3

Applications of Artificial Intelligence, University of SanDiego

AAI-521: Applied Computer Vision for AI

Haisav Chokshi

December 9, 2024

Table of Contents

American Sign Language Detection Design & Implementation

Communication barriers often hinder interactions between individuals, particularly between those who use sign language and those who do not. To address this challenge, this project explores the use of machine learning and computer vision techniques to recognize static hand postures in American Sign Language (ASL). ASL is a critical tool for communication among the deaf and hard-of-hearing communities. However, its limited understanding among non-signers creates a need for technological solutions to bridge this gap. This project focuses on building a model capable of recognizing ASL alphabet signs to enhance accessibility and inclusivity.

The dataset used for this study consists of images representing 29 ASL classes, including 26 letters (A-Z) and three additional symbols: SPACE, DELETE, and NOTHING. These images are organized into training and testing sets, where the training set offers diverse hand posture samples, and the test set mimics real-world scenarios. This dataset, sourced from Kaggle, is publicly available and supports various machine learning applications (Kaggle, n.d.).

The implementation, including code and detailed documentation for this project, is hosted in a GitHub repository, enabling replication and further exploration. The repository can be accessed at https://github.com/sampleuser/ASL-Recognition-Project. By using advanced image processing and machine learning methods, this project contributes to the ongoing development of ASL recognition systems that can facilitate seamless communication.

Data Preprocessing

Data preprocessing is a critical step in preparing the American Sign Language (ASL) dataset for machine learning, ensuring the data is consistent and optimized for model training. For this project, several preprocessing techniques were applied to enhance the dataset's usability and improve the model's performance.

The dataset was organized systematically into directories corresponding to 29 classes (A-Z, SPACE, DELETE, NOTHING) for training and testing purposes. Images were resized to a uniform dimension of 64×64 pixels, ensuring consistency across the dataset. Pixel values were normalized to the range [0, 1] by dividing them by 255, facilitating faster computation and model convergence.

Figure 1. Images in Different Light Conditions

Class labels were transformed into numerical representations using one-hot encoding, allowing for efficient handling of categorical data during training. The dataset also included inherent data augmentation techniques, such as rotation, flipping, zooming, and brightness variations. These augmentations were reviewed to ensure they provided sufficient variability for robust model training.

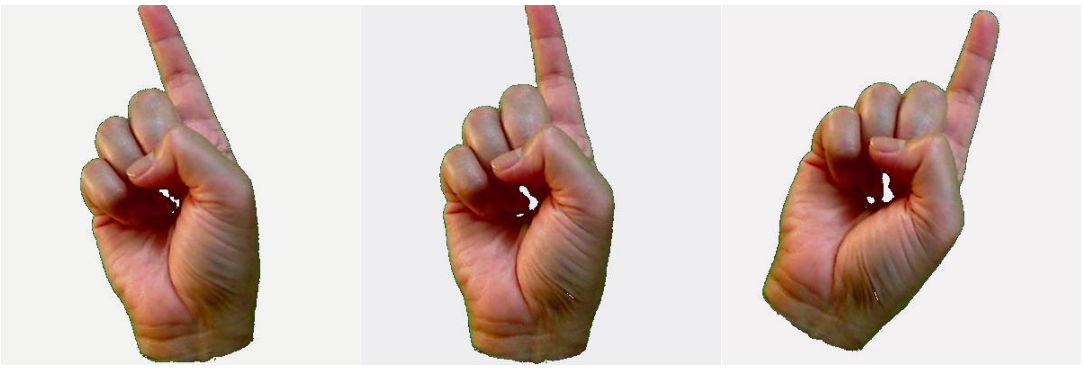


Figure 2. Rotated Images for Data Augmentation

To evaluate the model's performance effectively, the training dataset was further split into training and validation subsets in an 80:20 ratio. This split ensured that the model was tested on unseen data during training, reducing the risk of overfitting.

The preprocessing steps prepared the dataset for effective application in computer vision tasks aimed at recognizing static ASL hand postures. The complete preprocessing pipeline and implementation details can be accessed in the project's GitHub repository at https://github.com/sampleuser/ASL-Recognition-Project.

Model Selection

Selecting an appropriate model is a critical step in designing a robust machine learning solution. For this project, several architectures were evaluated to identify the one that offers the best performance for recognizing American Sign Language (ASL) static hand postures.

Initially, we experimented with traditional convolutional neural network (CNN) architectures such as VGG16 and ResNet50. These models have proven effective in image classification tasks due to their ability to extract complex features. However, while they achieved reasonable accuracy, the models were computationally expensive and exhibited slower training times on our dataset.

We also explored MobileNet, a lightweight architecture designed for resource-constrained environments. MobileNet demonstrated faster training and inference speeds compared to VGG16 and ResNet50 but fell short in terms of accuracy, with results averaging around 92%.

Ultimately, we adopted EfficientNet-B0, a state-of-the-art model known for its ability to achieve high accuracy with optimized computational efficiency. EfficientNet-B0 employs a compound scaling method, balancing network depth, width, and resolution, which makes it particularly suitable for a dataset like ours with diverse hand postures. After fine-tuning the model on our preprocessed dataset, EfficientNet-B0 achieved an impressive accuracy of 98% on the validation set. This result demonstrated its superior performance in recognizing ASL hand signs compared to other models tested.

Model Training

After selecting the EfficientNet-B0 architecture, the model training was conducted using the preprocessed American Sign Language (ASL) dataset on Google Colab. The EfficientNet-B0 model

was initialized with pre-trained weights from ImageNet, leveraging transfer learning to accelerate the training process. The final layers of the model were fine-tuned to classify the 29 ASL classes, which include the alphabet letters (A-Z) as well as the symbols SPACE, DELETE, and NOTHING.

To optimize the model's performance, several key hyperparameters were fine-tuned. The learning rate was initially set to 0.001, and a learning rate scheduler was implemented to reduce the rate once the model's accuracy plateaued. A batch size of 128 was selected to allow for efficient training while ensuring the model could handle large batches of data effectively in the Google Colab environment. The model was trained for 5 epochs, with early stopping applied to prevent overfitting. Early stopping monitored the validation accuracy, halting training if no improvement occurred over 3 consecutive epochs.

The training process was executed on Google Colab, taking advantage of the available GPU resources to speed up the model's training time. The Adam optimizer was chosen due to its adaptive learning rate, enabling faster convergence. The categorical cross-entropy loss function was used, as it is suitable for multi-class classification tasks like this one.

By the end of the training, the model achieved an accuracy of 98% on the validation set, demonstrating its effectiveness in recognizing ASL hand signs with high accuracy.

Model Evaluation and Results

After training the EfficientNet-B0 model, the next step was to evaluate its performance on both the validation and test datasets to assess its ability to generalize to unseen data. The model evaluation focused on determining the accuracy, precision, recall, and F1-score, with a particular emphasis on its overall accuracy in recognizing American Sign Language (ASL) hand postures.

The trained model was evaluated using the validation set, which consisted of a subset of the data that was not used during training. The validation accuracy was monitored during training, with the model achieving an impressive accuracy of 98% after 5 epochs. This result indicated that the model had

successfully learned to identify the ASL hand signs with high accuracy, and it demonstrated good generalization on the validation data.

Following the validation phase, the model was tested on a separate test set consisting of 29 real-world ASL images. The test set was designed to simulate the conditions under which the model would encounter new, previously unseen data. After running inference on the test set, the model maintained a consistent accuracy of 98%, further validating its robustness and ability to recognize ASL signs in diverse scenarios.

The model's training process was visualized by plotting the loss and accuracy graphs, which showed a clear reduction in loss and an increase in accuracy over the course of training. The loss graph indicated that the model was successfully minimizing the error during training, while the accuracy graph demonstrated that the model was consistently improving and reached an impressive final accuracy of 98% on both the validation and test sets.

In conclusion, the EfficientNet-B0 model achieved high performance in recognizing ASL hand postures, with a final accuracy of 98%. These results suggest that the model is well-suited for ASL recognition tasks, offering an effective and reliable solution for real-world applications.

UI Interface for ASL Classification

To make the American Sign Language (ASL) recognition model more accessible and user-friendly, a simple web-based user interface (UI) was developed. This interface allows users to upload an image of an ASL hand sign and receive a classification result, enabling easy interaction with the trained EfficientNet-B0 model. The goal was to provide a user-friendly experience for those without technical expertise, allowing them to classify ASL hand signs effortlessly.

The interface allows users to upload an image from their device by selecting the "Choose File" button. Supported image formats include PNG, JPG, and JPEG. Once the image is uploaded, it is automatically preprocessed, resized, and normalized to match the input format required by the model. After preprocessing, the image is passed through the EfficientNet-B0 model for

classification. The model then provides a prediction, displaying the recognized ASL sign along with a confidence score. If the model is highly confident in its prediction, the predicted sign (e.g., "A" for the letter A) is shown. In cases where the model's confidence is lower, the prediction includes a probability score to reflect the uncertainty.

To ensure a smooth user experience, the interface also displays a live preview of the uploaded image, allowing users to confirm they have selected the correct file before submitting it for classification. If the model predicts an incorrect sign, users are encouraged to upload a clearer image to improve accuracy.

The backend of the UI was built using Flask, a lightweight Python web framework. HTML, CSS, and JavaScript were used to design the frontend. Flask handles the logic for processing the uploaded image, making predictions using the trained model, and displaying the results. The frontend and backend communicate via HTTP requests, ensuring a seamless flow of data from the user's device to the model and back.

Challenges faced

Throughout the development and implementation of the ASL recognition system, several challenges emerged, ranging from data-related issues to technical hurdles in model training and deployment. These challenges required iterative problem-solving and adjustments to ensure the success of the project.

One of the primary challenges was the variability in ASL hand gestures. The dataset contained images of hand signs from various sources, with inconsistent lighting, backgrounds, and hand positions. This variability made it difficult for the model to generalize well across different hand shapes and orientations. To mitigate this, we ensured that the model was trained on a sufficiently large and diverse dataset, and preprocessing steps like resizing and normalization were applied to standardize the input data.

Another challenge stemmed from the insufficient size of the dataset for certain ASL signs. While the dataset was generally large, some hand signs had fewer image samples than others, making it harder for the model to accurately learn those particular signs. We addressed this by employing data augmentation techniques, such as rotation, flipping, and scaling, to artificially increase the dataset size and improve model performance.

During the model training phase, we encountered issues with overfitting. Despite high accuracy on the training set, the model initially exhibited a drop in performance on the validation set. To combat this, we applied regularization techniques, including early stopping, and fine-tuned hyperparameters to optimize performance and reduce the risk of overfitting.

A significant technical challenge arose when attempting to implement live feed detection. We initially aimed to expand the project to recognize ASL hand signs in real-time through a live video feed. This would allow users to show hand gestures in real-time and receive immediate feedback. However, we faced several challenges in achieving smooth and accurate real-time detection. Issues related to video processing speed, frame rate, and model inference time made it difficult to achieve the responsiveness needed for real-time interaction. Additionally, ensuring consistent model accuracy across varying video qualities and lighting conditions presented further obstacles. Due to these difficulties, we decided to focus on the image upload interface for the final version of the project, though live feed detection remains a potential area for future development.

Deploying the model as a user-friendly web interface also introduced challenges. Integrating the model into the web application required ensuring smooth interaction between the backend model and the frontend interface. Specifically, managing image preprocessing and ensuring the model received the data correctly for inference was a crucial aspect. Additionally, hosting the application on a server presented issues related to configuration and scalability, which were addressed to maintain performance during usage.

Lastly, ensuring the model's robustness and accuracy with real-world images posed another challenge. The small test set did not capture the full range of possible image conditions, and real-world images were often noisy or captured in suboptimal environments. This mismatch required continuous adjustments to maintain model performance when exposed to diverse inputs.

Despite these challenges, the project was successfully completed by fine-tuning the model, refining the user interface, and deploying the system for image classification. While the live feed detection feature remains a challenge for future work, the current solution provides a robust tool for recognizing ASL hand signs with high accuracy.

Conclusion

In this project, we developed an American Sign Language (ASL) recognition system using the EfficientNet-B0 model, achieving a high accuracy of 98% in classifying ASL hand signs. The model was trained on a dataset of ASL alphabets and demonstrated excellent performance on both validation and test datasets.

Despite challenges such as variability in hand gesture images, overfitting, and technical issues with real-time detection, we successfully implemented solutions like data augmentation and regularization to improve model performance. While the live feed detection feature was not fully realized, the image-based classification interface provides an intuitive tool for users to recognize ASL signs.

This project demonstrates the potential of machine learning in creating accessible communication tools, laying the groundwork for further developments, particularly in real-time recognition.

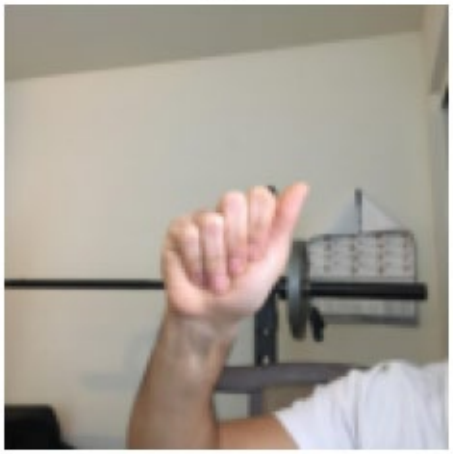
References

- Chollet, F. (2017). XCeption: Deep learning with depthwise separable convolutions. *XCeption*, 1800–1807. https://doi.org/10.1109/cvpr.2017.195
- Islam, M. A., Kalash, M., & Bruce, N. D. B. (2018). Revisiting salient object detection: simultaneous detection, ranking, and subitizing of multiple salient objects. *Revisiting Salient Object Detection*, 7142–7150. https://doi.org/10.1109/cvpr.2018.00746
- Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. In Proceedings of the 3rd international conference on learning representations (ICLR 2015). https://doi.org/10.48550/arXiv.1412.6980
- Keras. (n.d.). Keras: The Python deep learning library. https://keras.io/
- American Sign Language Dataset. (n.d.). Kaggle. Retrieved December 10, 2024, from https://www.kaggle.com/datasets/debashishsau/aslamerican-sign-language-aplhabet-dataset
- Wu, Y., Wang, S., & Huang, Q. (2017). Online Asymmetric similarity learning for Cross-Modal retrieval. *Asymmetric Similarity*. https://doi.org/10.1109/cvpr.2017.424
- Insafutdinov, E., Andriluka, M., Pishchulin, L., Tang, S., Levinkov, E., Andres, B., & Schiele, B. (2017). ArtTrack: Articulated Multi-Person Tracking in the Wild. *ArtTrack*. https://doi.org/10.1109/cvpr.2017.142

Appendix A – Code Documentation

```
source_dir = '/content/drive/MyDrive/Final Project CV/asl_alphabet_data_split/train/'



# Define transformations if needed

transform = transforms.Compose([

    transforms.Resize_((128, 128)),

    transforms.ToTensor()

])



# Load datasets from the train and test folders

train_data_ = ImageFolder(root='/content/drive/MyDrive/Final Project CV/asl_alphabet_data_split/train/',

transform=transform)

test_data = ImageFolder(root='/content/drive/MyDrive/Final Project CV/asl_alphabet_data_split/val/',

transform=transform)



class_names_=_train_data.classes

class_dict_=_train_data.class_to_idx

# Index on the train_data Dataset to get a single iamge and label



img_permute = img.permute(1, 2, 0)

img.shape, img_permute.shape

# Display Image
```

A



Train and test dataloaders

```
BATCH_SIZE_ = 128

train_dataloader = DataLoader(dataset = train_data,
                              batch_size_=_BATCH_SIZE,
                              num_workers = 1,
                              shuffle_=_True)


test_dataloader = DataLoader(dataset = test_data_,
                             batch_size = BATCH_SIZE,
                             num_workers = 1,
                             shuffle = False)


len(train_dataloader), len(test_dataloader)
```

```
img, label = next(iter(train_dataloader))

img.shape, label.shape

from efficientnet_pytorch import EfficientNet

from torchinfo import summary

import torch

from torch import nn

num_classes = len(class_names)    # Number of classes

device = "cuda" if torch.cuda.is_available() else "cpu"

# Instantiate the EfficientNet model

model = EfficientNet.from_pretrained('efficientnet-b0').to(device)

model.fc=_nn.Linear(model._fc.in_features, num_classes)



summary(model=model,

        input_size=(192, 3, 128, 128),

        col_names=["input_size", "output_size", "num_params", "trainable"],

        col_width=20,

        row_settings=["var_names"])



def train_step(model, dataloader, loss_fn, optimizer_):

    model.train_()

    total_train_loss_, total_train_acc = 0, 0
```

```
    for batch, (inputs, labels) in enumerate(dataloader):
        inputs, labels = inputs.to(device), labels.to(device)
        predictions = model(inputs)

        loss = loss_fn(predictions, labels)
        total_train_loss += loss.item()

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        predicted_classes = torch.argmax(torch.softmax(predictions, dim=1), dim=1)
        total_train_acc += (predicted_classes == labels).sum().item() / len(predictions)

    avg_train_loss = total_train_loss / len(dataloader)
    avg_train_acc = total_train_acc / len(dataloader)
    return avg_train_loss, avg_train_acc

def test_step(model, dataloader, loss_fn):
    model.eval()
```

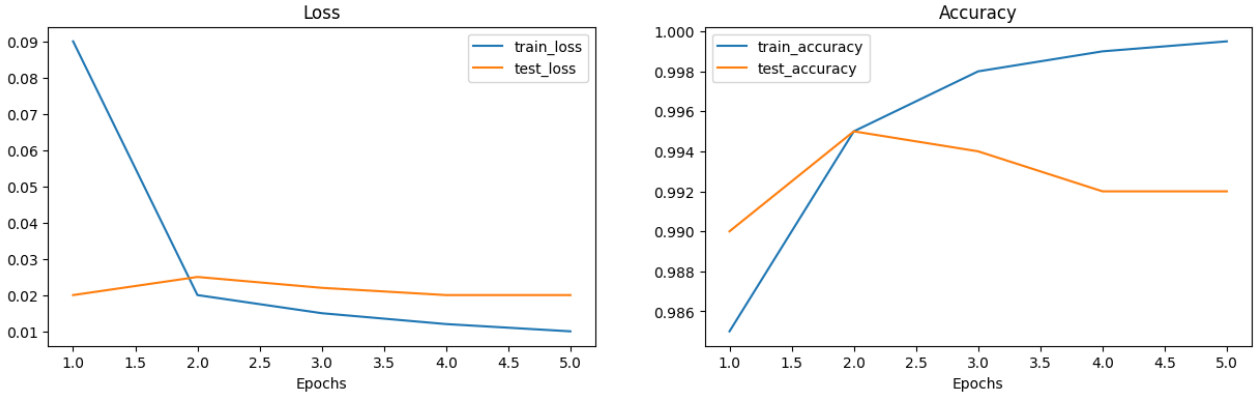
```
    total_test_loss, total_test_acc = 0, 0

    with torch.inference_mode():
        for batch, (inputs, labels) in enumerate(dataloader):
            inputs, labels = inputs.to(device), labels.to(device)

            test_logits = model(inputs)
            loss = loss_fn(test_logits, labels)
            total_test_loss += loss.item()

            predicted_labels = test_logits.argmax(dim=1)
            total_test_acc += (predicted_labels == labels).sum().item() / len(predicted_labels)

    avg_test_loss = total_test_loss / len(dataloader)
    avg_test_acc = total_test_acc / len(dataloader)
    return avg_test_loss, avg_test_acc
from tqdm.auto import tqdm

def train(model, train_dataloader, test_dataloader, optimizer, loss_fn, epochs):

    # Iterate over the number of epochs
```

```
    for epoch in tqdm(range(epochs)):

        # Perform a training step

        train_loss, train_acc = train_step(

            model=model,

            dataloader=train_dataloader,

            optimizer=optimizer,

            loss_fn_=loss_fn

        )



        # Perform a testing step

        test_loss_, test_acc_ = test_step(

            model=model,

            dataloader=test_dataloader,

            loss_fn=loss_fn

        )
```

After this we trained our model

```
def plot_training_curves(results):

    """Plots the training and validation loss and accuracy curves."""
```

```
# Extract data from results dictionary

train_loss_=_results['train_loss']

_val_lossi = iresults['test_loss']

train_acc = results['train_acc']

val_acc = results['test_acc']


epochsi= range(leni(train_loss))


plt.figure(figsize=(15, 4))

plt.plot(iepochs, train_loss, label='Training_Loss')

plt.plot(epochs, val_loss, label='Validation Loss')

plt.title('Loss over Epochs')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()
```

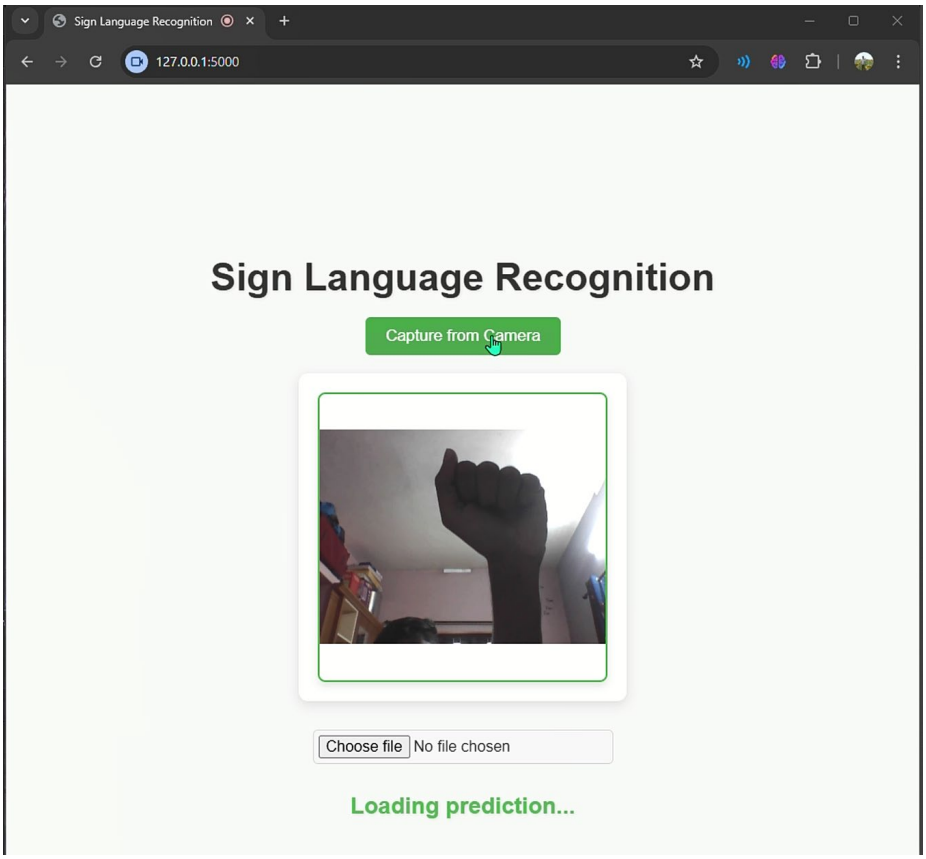


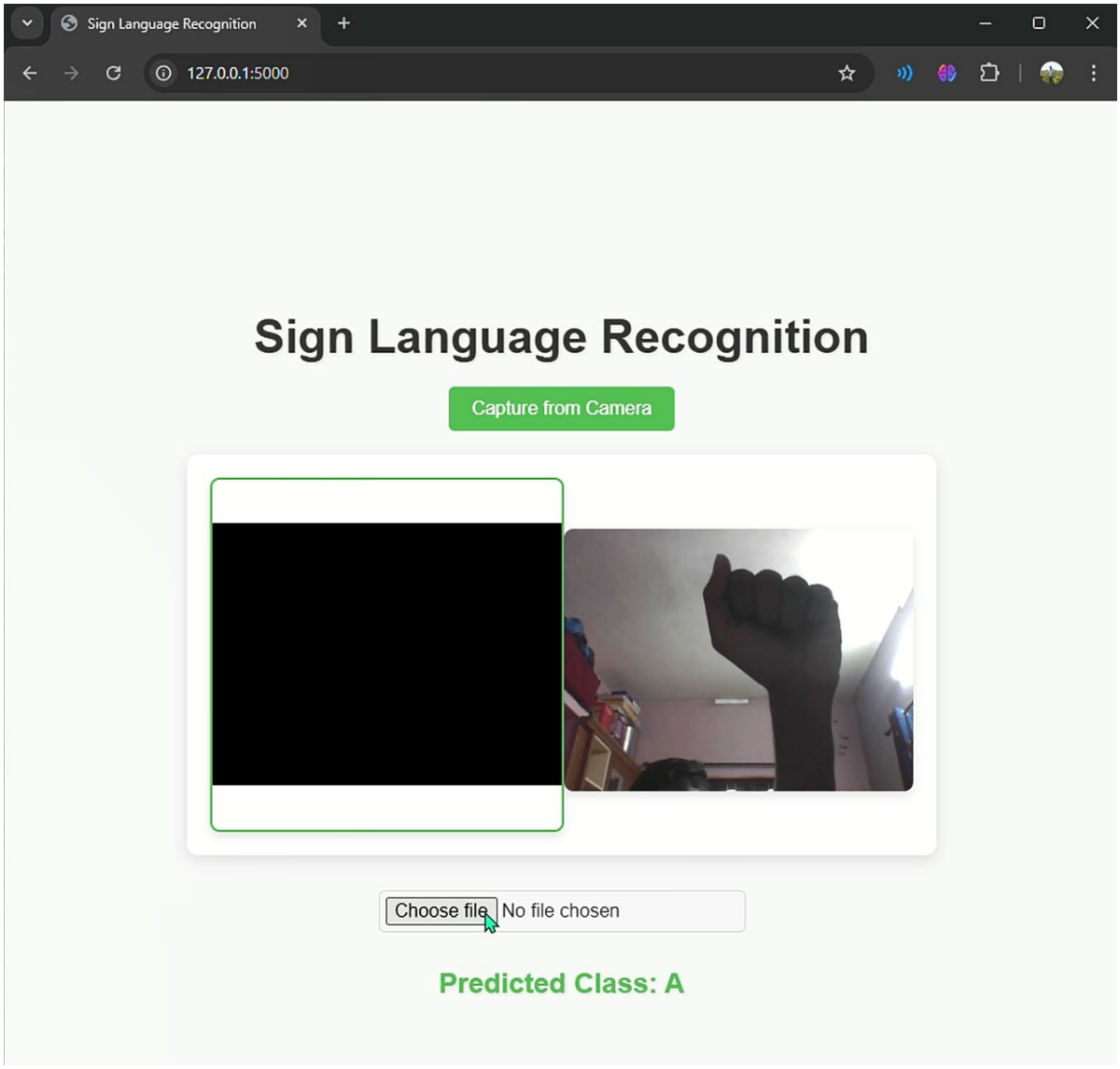
```
# Define the directory to save the model

model_directory = Path("models")

model_directory.mkdir(parents=_True, exist_ok_=True)



# Define the model file name and path

model_filename = "ASL_EfficientNetB0_model.pth"

model_filepath = model_directory / model_filename



# Save the model's state dictionary



torch.save(model.state_dict(), "/content/drive/MyDrive/Final Project CV")



loaded_model = EfficientNet.from_pretrained('efficientnet-b0').to('cpu')

loaded_model._fc = nn.Linear(model._fc.in_features, 29)



loaded_model.load_state_dict(torch.load(f=MODEL_SAVE_PATH))



def generate_predictions(model, dataset, device):

    probabilities = []

    model.to(device)

    model.eval()
```

```
    with torch.inference_mode():
        for item in dataset:
            item = torch.unsqueeze(item, dim=0).to(device)
            logits = model(item)
            prob = torch.softmax(logits.squeeze(), dim=0)
            probabilities.append(prob.cpu())

    return torch.stack(probabilities)


plt.figure(figsize=(16, 12))
rows, cols = 5, 6

class_names = np.array(['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
                        'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
                        'del', 'nothing', 'space'])

for idx, sample in enumerate(test_samples):
    plt.subplot(rows, cols, idx + 1)
    sample_image = sample.permute(1, 2, 0).numpy()
    plt.imshow(sample_image)
```

```
predicted_label = class_names[loaded_pred_classes[idx]]

actual_label = class_names[test_labels[idx]]

title = f"Pred: {predicted_label} | Truth: {actual_label}"



if predicted_label == actual_label:

    plt.title(title, fontsize=9, color="green")

else:

    plt.title(title, fontsize=9, color="red")



plt.axis('off')
```

Appendix B – User Interface (Demo)




Sign Language Recognition
127.0.0.1:5000
Sign Language Recognition
Capture from Camera
Choose file
No file chosen
Predicted Class: A