Università di Pisa

Master Of Science in Computer Engineering

intelligent systems - computational intelligence and deep learning

---

DeepSigns
ASL Fingerspelling Recognition System

Project members:
Massimo Valentino Caroti
Niccolò Mulè

Contents

1 Introduction

The problem we address concerns communication barriers between the hearing and deaf community, mainly in the context of American Sign Language (ASL) hand language. People who use ASL as their primary means of communication may face difficulties in communicating with hearing people who do not know or understand hand language. This situation can cause isolation, limitations in social participation, and a lack of mutual understanding.

Our solution is to develop a neural network that can recognize and interpret ASL hand signs, enabling immediate translation between hand language and text or audio understandable to the hearing community. This technology can be implemented in accessible applications and devices, enabling smoother and more inclusive communication between the two communities. The ultimate goal is to break down communication barriers, promote empathy, and foster greater mutual participation and understanding between the hearing and deaf community.

The work done and all pyhton notebooks are available in the following repository.

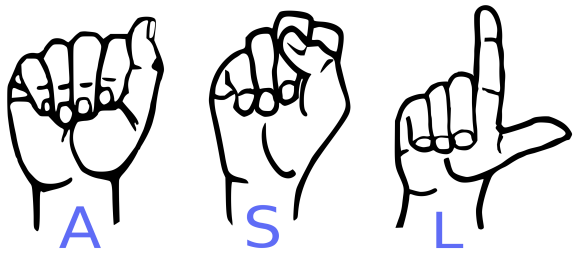


Figure 1: ASL spelling

2 American Sign Language

ASL orthography is a part of American Sign Language (ASL) used to represent and communicate the spelling of words. In ASL spelling, each letter of the alphabet is represented by a specific hand sign.
People use ASL speeling to communicate and understand specific words, proper names or technical terms that may not have a dedicated ASL sign.
ASL spelling consists of 26 distinct signs, corresponding to the letters of the English alphabet. Each sign represents a specific letter and is performed with the hands in different positions and movements. The correct representation of letters through ASL spelling is important to ensure accurate and clear communication.

2.1 Dataset

The dataset we decided to use is a collection of over 87.000 RGB images of ASL handshapes corresponding to the letters of the English Alphabet. More specifically, the dataset contains 29 classes, 26 of which are for the letters from A to Z and 3 classes for SPACE, DELETE, NOTHING.

Figure 2: ASL Dataset

For each ASL handshape there are exactly 3000 samples.

2.2 State of Art

Regarding state-of-the-art ASL handwriting recognition, there are several alternatives for the researchers and developers to consider, each with its own strengths and tradeoffs.
The most common approach involves the use of convolutional neural networks (CNNs), which have proven effective in extracting spatial features from video sequences or images of hand gestures. Depending on the specific requirements of the task, different CNN architectures can be used. For example, architectures such as GoogLeNet and SqueezeNet have been used in previous work because of their efficiency and performance.

2.2.1 Paper 1 : Stanford University

The paper entitled "Real-time American Sign Language Recognition with Convolutional Neural Networks" presents the development and implementation of an American Sign Language (ASL) fingerprint translator based on a convolutional neural network.
The authors use a pre-trained GoogLeNet architecture, trained on the ILSVRC2012 dataset and the Surrey University and Massey University ASL datasets, to apply transfer learning to this task.

The authors produced a robust model that correctly classifies letters in most cases. The authors also discuss the challenges of ASL recognition, including environmental issues, occlusion, sign boundary detection and co-articulation. The authors obtained a maximum validation accuracy of **0.9782**.

2.2.2 Paper 2 : VIT University

The paper titled "American Sign Language Alphabet Recognition using Deep Learning" proposes a model for recognizing the American Sign Language (ASL) alphabet from RGB images using deep learning.
The model was trained on a SqueezeNet architecture, a type of convolutional neural network (CNN) designed to be smaller and faster than other CNNs, making it suitable for use on mobile devices.

The authors obtained a maximum training accuracy of **0.8747**. The validation accuracy obtained was **0.8329**. The network learned ASL which enabled it to predict sign language in real time.

3 Neural Network from scratch

In this chapter, we deal with the process of building a convolutional neural network (CNN) from scratch. A CNN is a powerful deep learning architecture widely used for image classification tasks. We start with a simple network and address the challenges of underfitting and overfitting step by step, applying data pre-processing to improve model performance. We discuss the design of the architecture, correctly selecting layers and parameters to achieve accurate results. We also devote attention to optimising the model to ensure better generalisation.

3.1 Experiment 1 – Base model

We start by designing a basic CNN architecture that forms the foundation of our model. The architecture consists of several key components:

- A convolutional layer with 16 filters.
- Kernel size set to 3, to ensure that the convolutional operation effectively captures patterns.
- 'Same' padding to maintain the spatial dimensions of the feature maps.
- ReLU activation function, to introduce nonlinearity and improve the model's ability to capture complex relationships.
- Max Pooling layer with pooling window of size 2x2.
- Flatten layer to resize the output.
- Fully-connected layer with softmax activation for creating undistribution of probability over the different classes, allowing us to make predictions based on the highest probability.

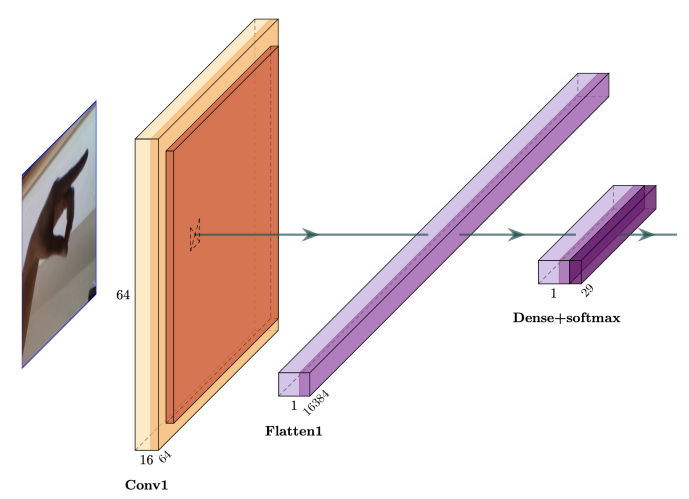


Figure 3: Base model archiecture

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 64, 64, 16)	448
max_pooling2d_1 (MaxPooling 2D)	(None, 32, 32, 16)	0
flatten_1 (Flatten)	(None, 16384)	0
dense_1 (Dense)	(None, 29)	475165

Total params: 475,613
Trainable params: 475,613
Non-trainable params: 0

Figure 4: Base model summary

After establishing the architecture, we proceed with the training of our neural network. Training consists of iteratively updating the model parameters in order to minimize the loss function and improve performance. We train the model for a total of 80 epochs. During each epoch, the model is exposed to the training data and the weights are adjusted according to the calculated loss. This iterative process helps the model learn and generalize from the training data.

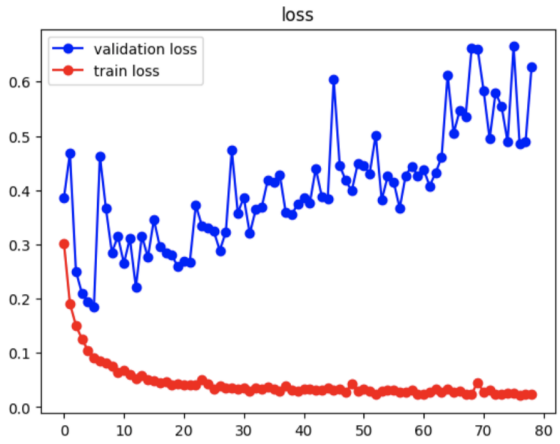


Figure 5: Base model loss

```
[ ] model.evaluate(X_test, y_test, return_dict = True)

    272/272 [==============================] - 1s 3ms/step - loss: 0.1356 - accuracy: 0.9939
    {'loss': 0.13563889265060425, 'accuracy': 0.9939080476760864}
```

The evaluation of the current network shows some notable signs of overfitting. The validation loss is gradually increasing despite the training loss approaching 0. This means that our network is losing the ability to generalize on unseen data. In the upcoming sections, we will first increase the network capacity in order to achieve a better accuracy level and secondly we will introduce some techniques to prevent overfitting.

3.2 Experiment 2 – Model B : Increasing the network capacity

To enhance the performance of our model, we are increasing its capacity by incorporating two additional convolutional layers. These layers will have a greater number of filters, enabling the model to capture more intricate features and patterns. Additionally, we have introduced an extra fully connected layer to further enhance the model's learning capabilities.
Although we anticipate improved accuracy with this expanded architecture, we are aware of the risk of overfitting on the training data. We will address this concern in subsequent steps to ensure that the model generalizes well to unseen examples. These are the improvements on the Base model:

- Adding two extra convolutional layers.
- Each additional layer will have an increased number of filters
- An extra fully connected layer is introduced

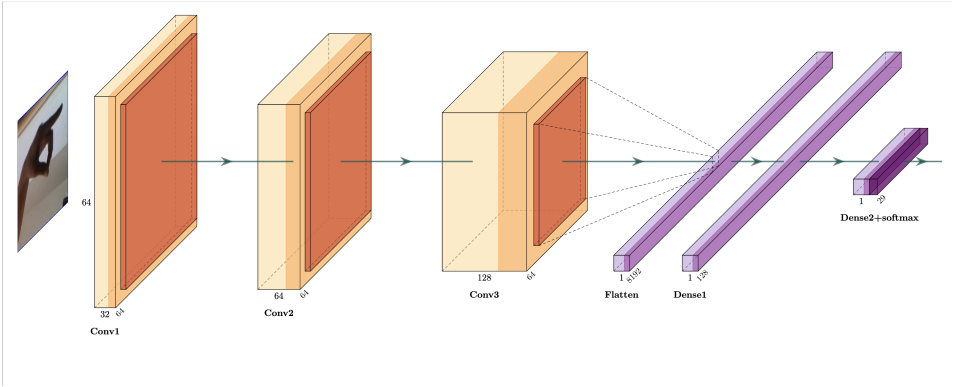


Figure 6: Model B archiecture

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 64, 64, 32)	896
max_pooling2d_2 (MaxPooling 2D)	(None, 32, 32, 32)	0
conv2d_3 (Conv2D)	(None, 32, 32, 64)	18496
max_pooling2d_3 (MaxPooling 2D)	(None, 16, 16, 64)	0
conv2d_4 (Conv2D)	(None, 16, 16, 128)	73856
max_pooling2d_4 (MaxPooling 2D)	(None, 8, 8, 128)	0
flatten_2 (Flatten)	(None, 8192)	0
dense_2 (Dense)	(None, 128)	1048704
dense_3 (Dense)	(None, 29)	3741

Total params: 1,145,693
Trainable params: 1,145,693
Non-trainable params: 0

After establishing the architecture, we proceed with the training of our neural network. We train the model for a total of 80 epochs.

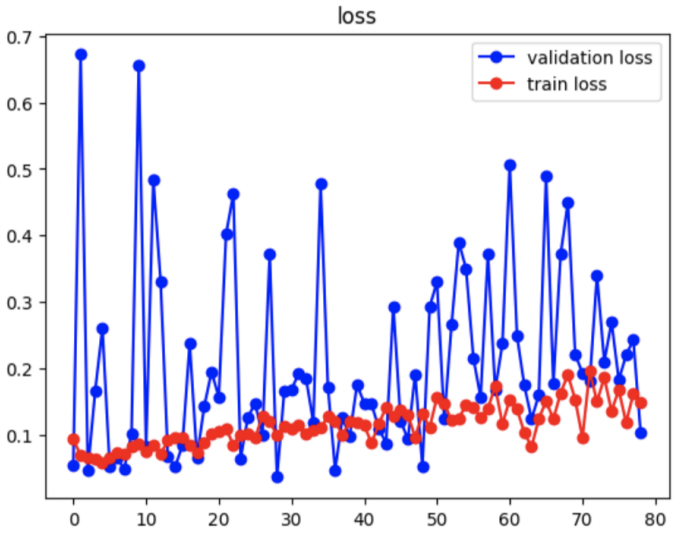


Figure 7: Model B loss

```
272/272 [==============================] - 1s 4ms/step - loss: 0.0518 - accuracy: 0.9992
{'loss': 0.05176781490445137, 'accuracy': 0.999195396900177}
```

As expected, the network exhibited significant improvements in accuracy, but it is still prone to overfitting. We have reached this conclusion based on two main observations:

1. The validation loss graph reveals occasional spikes, indicating that the model struggles to generalize effectively. These fluctuations imply that the model is not able to generalize properly

2. The training loss is increasing over time, suggesting that the model is learning patterns that are actually noise or random variations rather than useful features. This is clearly caused by the increase in capacity of the network

It is crucial to address this overfitting issue to ensure the model's generalization capability and enhance its performance on unseen examples.

3.3 Experiment 3 – model C : Handling overfitting

To address the issue of overfitting, we have implemented several techniques:

- Dropout layers: After each Convolutional-MaxPooling layer, we have introduced dropout layers. These layers randomly deactivate a certain percentage of neurons during each training iteration, allowing each neuron to learn independently without relying heavily on neighboring neurons. This technique helps prevent over-reliance on specific features and encourages the model to learn more robust representations.
- Early stopping: We have incorporated an early stopping mechanism by utilizing a callback during model training. This technique monitors the model's performance on a validation set and halts the training process if there is no significant improvement within a specified number of steps. By stopping the training early, we can prevent the model from overfitting and save computational resources.
- Learning rate reduction on plateau: To enhance convergence and avoid getting stuck in suboptimal solutions, we have implemented a learning rate reduction strategy. If there is a lack of significant improvement in the model's performance over a certain number of epochs, we dynamically decrease the learning rate. This adjustment allows the model to make smaller updates to the weights and potentially find better solutions.

By using these techniques, we try to prevent overfitting and improve the generalising ability of our model, ultimately improving its performance on unseen data.
Compared to the previous model, we have added :

- Three dropout layers at 0.3
- EarlyStopping with patience=10
- ReduceLROnPlateau with patience=5

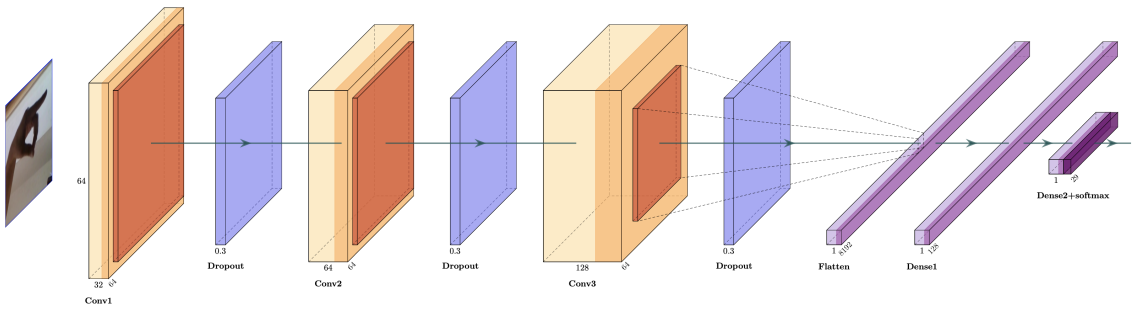


Figure 8: Model C archiecture

Layer (type)	Output Shape	Param #
conv2d_5 (Conv2D)	(None, 64, 64, 32)	896
max_pooling2d_5 (MaxPooling 2D)	(None, 32, 32, 32)	0
dropout (Dropout)	(None, 32, 32, 32)	0
conv2d_6 (Conv2D)	(None, 32, 32, 64)	18496
max_pooling2d_6 (MaxPooling 2D)	(None, 16, 16, 64)	0
dropout_1 (Dropout)	(None, 16, 16, 64)	0
conv2d_7 (Conv2D)	(None, 16, 16, 128)	73856
max_pooling2d_7 (MaxPooling 2D)	(None, 8, 8, 128)	0
dropout_2 (Dropout)	(None, 8, 8, 128)	0
flatten_3 (Flatten)	(None, 8192)	0
dense_4 (Dense)	(None, 128)	1048704
dense_5 (Dense)	(None, 29)	3741

Total params: 1,145,693
Trainable params: 1,145,693
Non-trainable params: 0

Figure 9: Model C summary

With the model's architecture established, we proceed to train the neural network., through 80 epochs of training,

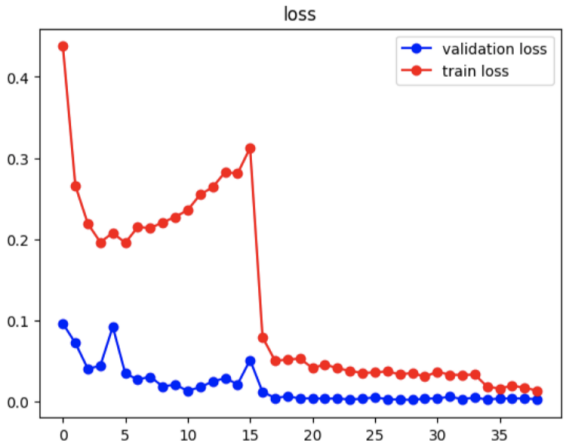


Figure 10: Model C loss

```
272/272 [==============================] - 1s 5ms/step - loss: 9.3140e-05 - accuracy: 1.0000
{'loss': 9.314011549577117e-05, 'accuracy': 1.0}
```

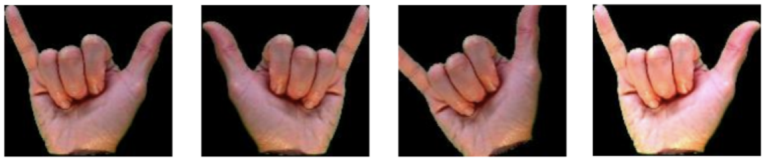
The techniques we applied allowed us to handle the overfitting. In the next section we try to increase the stability of the model by using preprocessing techniques on the dataset.

3.4 Experiment 4 – model D : Data augmentation and RGB normalization

To enhance the stability of our model's performance and improve its robustness in dealing with unseen data, we have adopted RGB Normalization and Data Augmentation strategies.
Data augmentation is the technique of generating synthetic data to increase the diversity of the training set.
We applied some transformations to the images during the training process. The specific transformations we used are described below:

- Rescale: We normalised the pixel values of the images by dividing each value by 255. This step helps to reduce the scale of pixel values in the range [0, 1], facilitating the training process of the neural network.
- Rotation Range: We applied a random rotation to the images in the range of 10 degrees.
- Zoom Range: We applied a random zoom transformation to the images in the range of 0.1.
- Brightness Range: We applied a random brightness transformation to images in the range of 0.8 to 1.2.

Using these transformations during model training increases the diversity of the training data and improves the generalisation capability of the model, allowing it to successfully cope with variations and noise in the test data.



Original image Flip image Rotation image Brightness image

By using the structure of the previous model we obtain these results:

Layer (type)	Output Shape	Param #
conv2d_7 (Conv2D)	(None, 64, 64, 32)	896
max_pooling2d_7 (MaxPooling 2D)	(None, 32, 32, 32)	0
dropout_6 (Dropout)	(None, 32, 32, 32)	0
conv2d_8 (Conv2D)	(None, 32, 32, 64)	18496
max_pooling2d_8 (MaxPooling 2D)	(None, 16, 16, 64)	0
dropout_7 (Dropout)	(None, 16, 16, 64)	0
conv2d_9 (Conv2D)	(None, 16, 16, 128)	73856
max_pooling2d_9 (MaxPooling 2D)	(None, 8, 8, 128)	0
dropout_8 (Dropout)	(None, 8, 8, 128)	0
flatten_3 (Flatten)	(None, 8192)	0
dense_5 (Dense)	(None, 128)	1048704
dense_6 (Dense)	(None, 29)	3741

Total params: 1,145,693
Trainable params: 1,145,693
Non-trainable params: 0

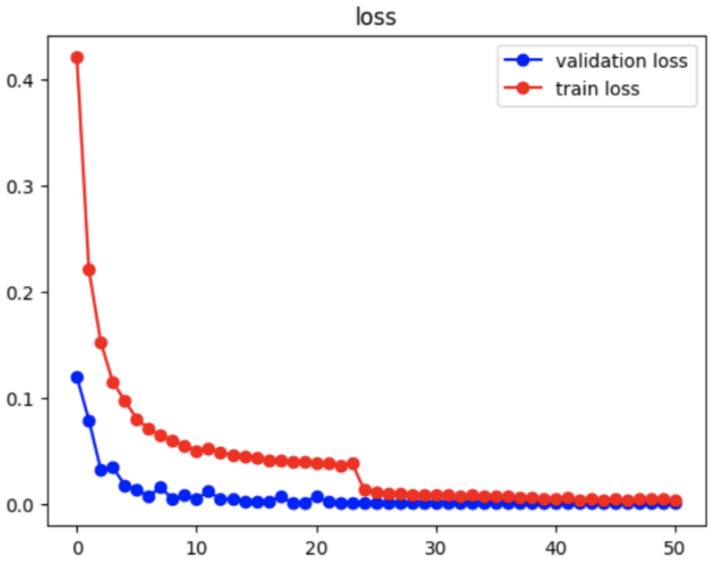


Figure 11: Model D loss

```
136/136 [==============================] - 1s 7ms/step - loss: 1.3415e-06 - accuracy: 1.0000
{'loss': 1.3415467492450261e-06, 'accuracy': 1.0}
```

Figure 12: Model D accuracy

As anticipated, we have achieved a more stable model and further reduced loss on unseen data by incorporating data augmentation. In the upcoming section, we will explore hyper-parameters tuning in depth, allowing us to find the best configuration for our network.

3.5 Hyper-parameters tuning

Our next phase of development involves fine-tuning the hyperparameters of our model to achieve optimal performance.
The hyperparameters under consideration include:

- Initial number of filters
- Number of convolutional layers
- Kernel size
- Initial learning rate
- Dropout rate.

To simplify the process, we performed the search for hyperparameters without implementing real-time data augmentation, which can be computationally intensive. This approach allows us to focus specifically on finding the best combination of hyperparameters for our model.
The hyperparameters we tested are:

- init-filters: The initial amount of filters, it can take the values 32 or 64.
- conv-layers: The number of convolution layers. Varies from 1 to 3 .
- kernel-size: The size of the kernel for the convolutions. Can take the values 3, 5 or 7.
- learning-rate: The initial learning rate for the optimizer. It can take the values 10^{-3} or 10^{-4}.
- dropout-i: The dropout rate for each convolution layer. It varies from 0.1 to 0.3 with a step size of 0.1.

These hyperparameters are selected via the `HyperParameters` object of `Keras Tuner`, which allowed us to specify values or ranges of possible values for each hyperparameter.
We use the tuner search method to perform the search for the optimal hyperparameters. During the search, we train the model for a total of 60 epochs using the training data (X-train and y-train) and evaluate its performance on the validation data (X-val and y-val).

```
Trial 30 Complete [00h 03m 15s]
val_accuracy: 0.900510847568512

Best val_accuracy So Far: 0.9984674453735352
Total elapsed time: 00h 47m 20s
The hyperparameter search is complete.
```

Figure 13: Hyperparameter search

After training the model using the best hyperparameters, we store the epoch in which we achieved the highest level of accuracy. In our case, the best epoch is 23. This gives us a benchmark to monitor the improvement of the model's performance during training.
Best epoch: 23

We retrain the model with optimal hyperparameters by limiting the training to the number of epochs that yielded the best results previously.

```
hypermodel.evaluate(X_test, y_test)
```

```
272/272 [==============================] - 1s 3ms/step - loss: 0.0030 - accuracy: 0.9992
[0.002996991155669093, 0.999195396900177]
```

Figure 14: Hyperparameter evaluate

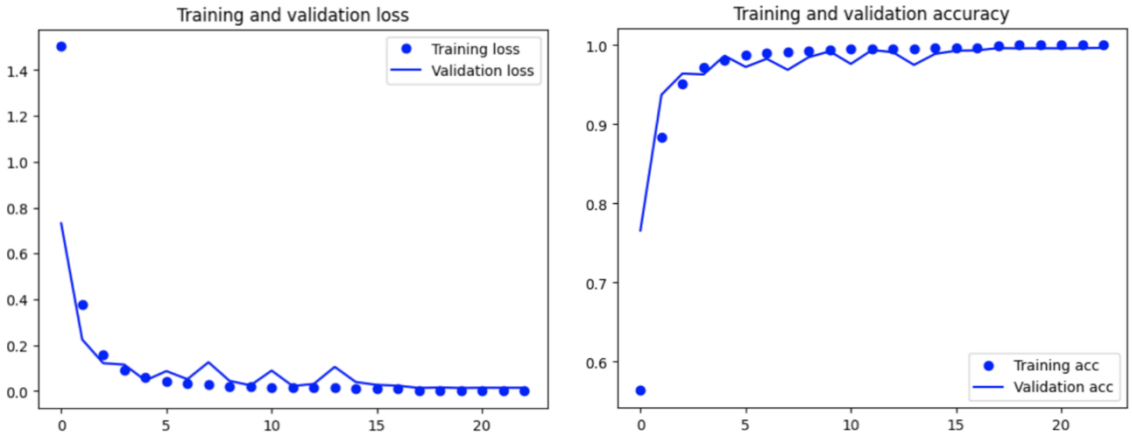


Figure 15: Hyperparameter accuracy and loss

These graphs provide a visual representation of model performance over epochs. The loss values decrease consistently, indicating that the model is effectively learning and reducing errors. This demonstrates the model's ability to generalize well over the dataset in question.
Overall, these results show the excellent performance of the model and its ability to learn effectively and make accurate predictions on unseen data.

4 Transfer Learning

Transfer learning is a strategy of using pre-trained deep learning models on large datasets to solve new problems in similar domains. This technique is based on the concept that features learned from a model on a large dataset can also be useful for solving different problems.
By applying transfer learning, advantages such as higher training efficiency and better generalisation of models over smaller datasets can be achieved. Furthermore, transfer learning allows one to benefit from the experience and knowledge accumulated from pre-trained models, accelerating the process of developing new solutions and reducing the need to train models from scratch.
There are two main techniques of transfer learning: fine-tuning and feature extraction.

- In fine-tuning, a pre-trained model is taken and some of its upper layers are adapted for the new task. In this way, the model can learn specific features of the new domain, while retaining previously learned knowledge on larger datasets.
- In feature extraction, features learnt from the lower layers of the pre-trained model are extracted and used as input for a new classifier. This allows the feature extraction capabilities of the pre-existing model to be utilised, while only training the final classifier with a specific dataset.

4.1 Fine-tuning : VGG16

VGG16 is a powerful convolutional neural network developed at the University of Oxford in 2014 that is widely used for image recognition. This model was trained on the vast ImageNet dataset, consisting of millions of images belonging to different categories. Through such training, VGG16 learnt high-level features and developed a rich representation of images, making it one of the benchmark models in the field of transfer learning and image recognition.

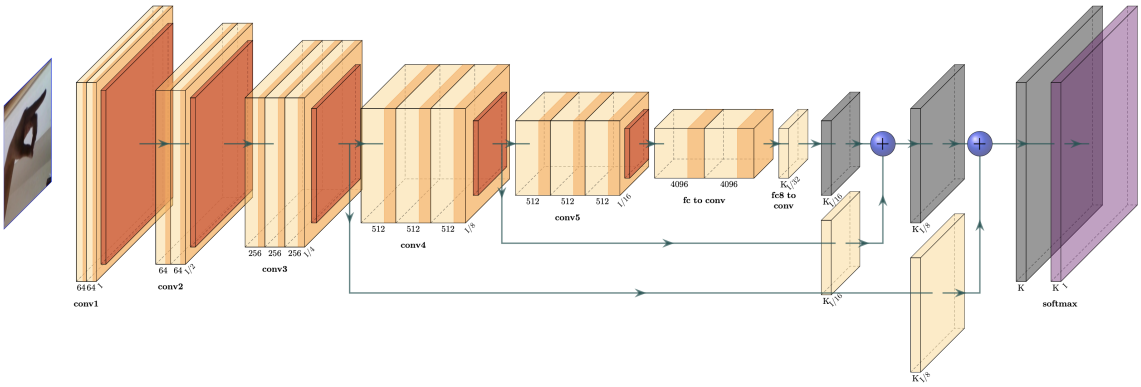


Figure 16: VGG16 archiecture

We are building a model using the VGG16 architecture. We set up the pre-trained VGG16 model as a base and added two new full-connected layers at the end of the network.

In the first phase we train only the final part of the model on our data, keeping the weights of the VGG16 network frozen to prevent them from being destroyed by backpropagation.

Layer (type)	Output Shape	Param #
vgg16 (Functional)	(None, 2, 2, 512)	14714688
flatten (Flatten)	(None, 2048)	0
dense (Dense)	(None, 128)	262272
dense_1 (Dense)	(None, 29)	3741

Total params: 14,980,701
Trainable params: 266,013
Non-trainable params: 14,714,688

Next, we freeze some weights of the base network and resume training on our dataset. In this case, the model will have more parameters since we are training both the new part and a portion of the VGG16 network. This approach allows the model to use the knowledge learned from VGG16 and adapt it to our specific problem.

Layer (type)	Output Shape	Param #
vgg16 (Functional)	(None, 2, 2, 512)	14714688
flatten_1 (Flatten)	(None, 2048)	0
dense_2 (Dense)	(None, 128)	262272
dense_3 (Dense)	(None, 29)	3741

Total params: 14,980,701
Trainable params: 7,345,437
Non-trainable params: 7,635,264

To evaluate our model's performance, we plot the accuracy and loss metrics.

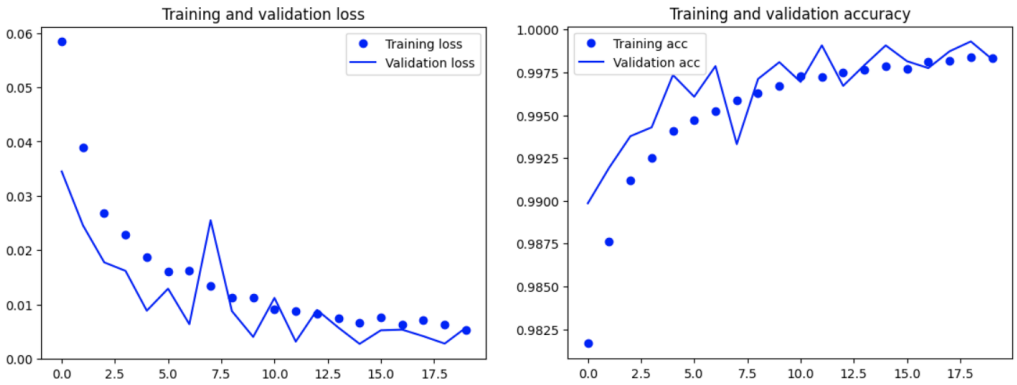


Figure 17: Fine-tuning accuracy and loss

By leveraging a pre-trained model, we managed to achieve very good results while mantaining a

```
136/136 [==============================] - 4s 30ms/step - loss: 0.0048 - accuracy: 0.9982
{'loss': 0.004787056241184473, 'accuracy': 0.9981608986854553}
```

Figure 18: Fine-tuning accuracy

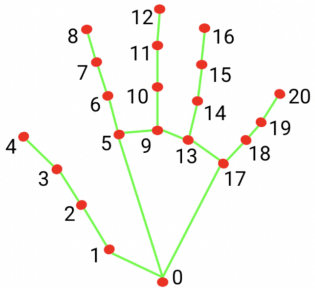
simplified code structure and putting less effort compared to designing the network from scratch. However, it's important to highlight the fact that although this approach is more convenient, this network is considerably bigger and slower than the ad-hoc solution we provided in the previous section.

4.2 Feature Extractor : MediaPipe

In transfer learning, feature extraction involves using a pre-trained neural network to extract features from a set of input data. The extracted features are then used as input to a new model, which is trained to solve a different problem.

4.2.1 MediaPipe

The hand landmark model detects the location of key points of 21 hand knuckle coordinates within the detected regions. The model was trained on about 30,000 real-world images and several rendered synthetic hand models imposed on various backgrounds. The definition of the 21 landmarks is shown below:



0. WRIST
1. THUMB_CMC
2. THUMB_MCP
3. THUMB_IP
4. THUMB_TIP
5. INDEX_FINGER_MCP
6. INDEX_FINGER_PIP
7. INDEX_FINGER_DIP
8. INDEX_FINGER_TIP
9. MIDDLE_FINGER_MCP
10. MIDDLE_FINGER_PIP
11. MIDDLE_FINGER_DIP
12. MIDDLE_FINGER_TIP
13. RING_FINGER_MCP
14. RING_FINGER_PIP
15. RING_FINGER_DIP
16. RING_FINGER_TIP
17. PINKY_MCP
18. PINKY_PIP
19. PINKY_DIP
20. PINKY_TIP

Figure 19: MediaPipe hand landmark

Using this pre-trained model, we can extract information about the positions of key hand points from the input images, and pass them as input to the new neural network.

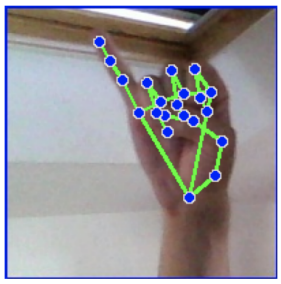


Figure 20: Landmark of letter I

Using the Mediapipe, we extracted the features corresponding to each image in the original dataset. As a result, we created a new dataset composed exclusively of the extracted features. However, it is important to note that the model was unable to correctly detect the hand in every single sample of the original dataset. This led to the creation of an **unbalanced dataset**.

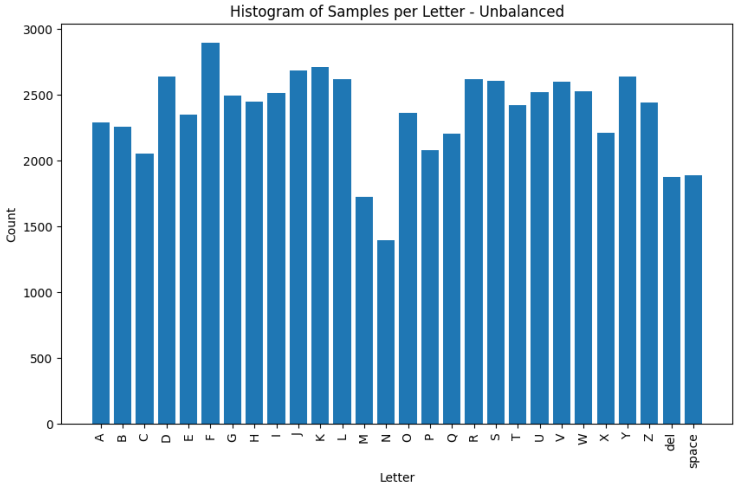


Figure 21: Visualizing dataset

4.2.2 Data Preprocessing

Before balancing the dataset, some transformations need to be performed. The feature extraction step with Mediapipe generates 21 landmarks for each recognized sign. From each landmark, coordinates in the three dimensions (x, y, z) are acquired. So, we modify the dataset to have each sign represented by one row, and each row contains 63 feature columns corresponding to the x, y, and z coordinates of each landmark.
In this way, we obtain a tabular representation of sign features ready for further processing and balancing of the dataset.
We created a neural network consisting of three layers of fully connected neurons with two dropout layers.

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 128)	8192
dropout (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 64)	8256
dropout_1 (Dropout)	(None, 64)	0
dense_3 (Dense)	(None, 29)	1885

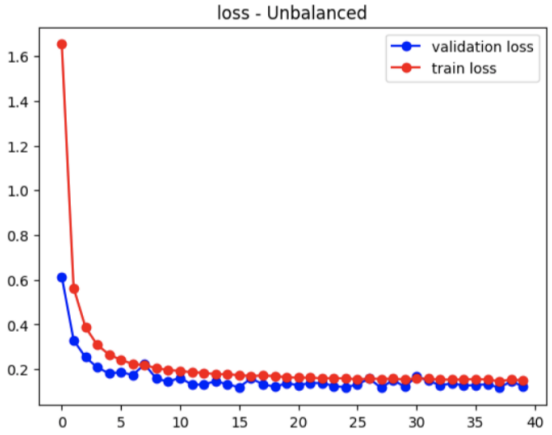
Total params: 18,333
Trainable params: 18,333
Non-trainable params: 0

This network is trained on the feature dataset and will return the corresponding sign. We used this network with three different datasets, as we decided to handle the imbalance using three different techniques:

- Unbalanced
- Random Undersampling
- Oversampling with SMOTE

4.2.2.1 Unbalanced

Leaving the dataset unbalanced and training the network for 40 epochs, we obtained the following results.

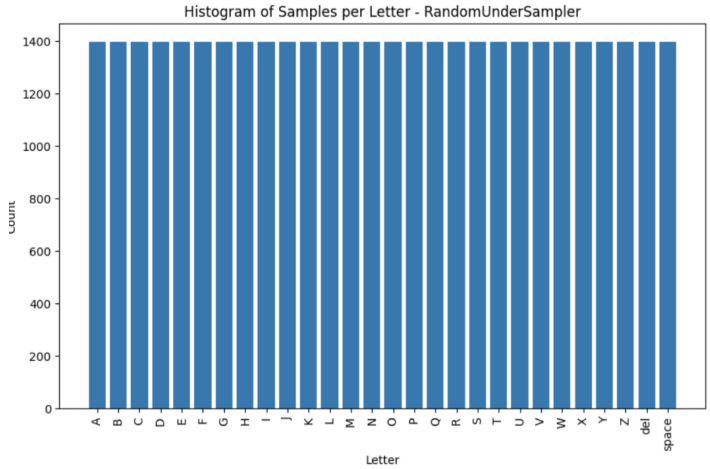


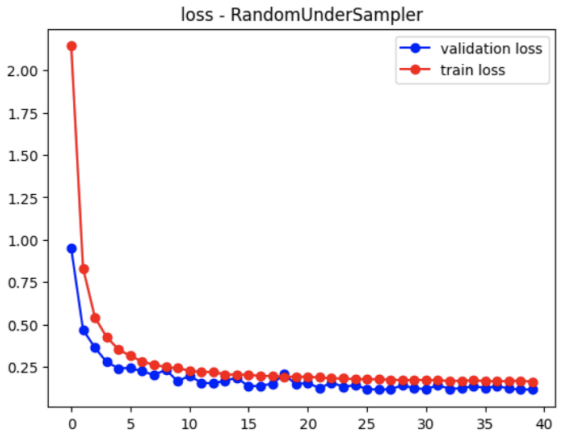

```
245/245 [==============================] - 0s 2ms/step - loss: 0.1226 - accuracy: 0.9710
Model performance - Unbalanced [0.12257082015275955, 0.9710052609443665]
```

Figure 22: Unbalanced accuracy

4.2.2.2 Under-sampling with RandomUnderSampler

Applying random subsampling on the dataset, we obtained about 1400 samples for each sign. By training the network for 40 epochs, we obtained the following results.






```
245/245 [==============================] - 0s 2ms/step - loss: 0.1294 - accuracy: 0.9728
Model performance - RandomUnderSampler [0.129404678940773, 0.972793459892273]
```

Figure 23: Undersampling accuracy

4.2.2.3 Over-sampling with Smote

Applying Smote to the dataset, we obtained about 2800 samples for each sign. By training the network for 40 epochs, we obtained the following results.

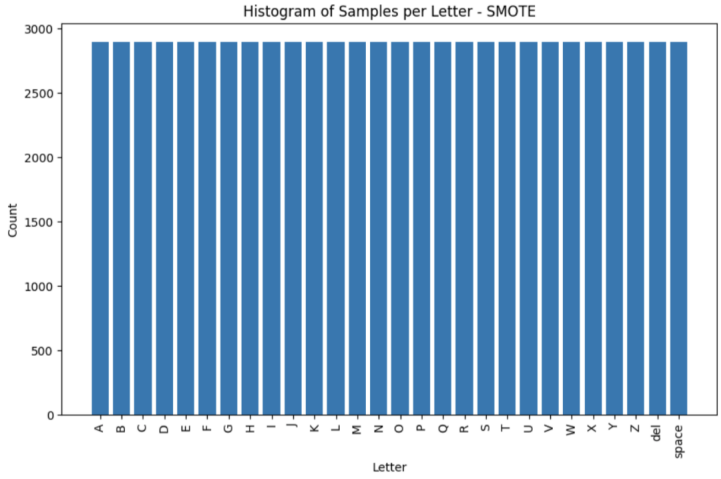
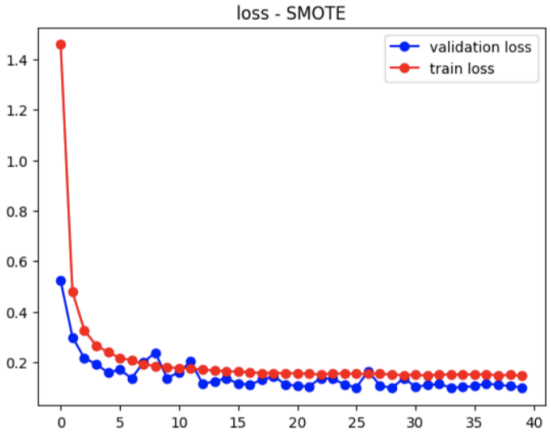


Figure 24: Oversampling dataset




```
245/245 [==============================] - 0s 2ms/step - loss: 0.1058 - accuracy: 0.9820
Model performance - SMOTE [0.10578987747430801, 0.9819900393486023]
```

Figure 25: Oversampling accuracy

As the results demonstrate, the accuracy and loss levels obtained in this phase are lower than those achieved in prior steps. However, this code structure is suitable for demo applications that use images of varied sizes, as feature extraction is conducted using a pre-trained model such as Mediapipe, which would otherwise require a more advanced model like a Region-based CNN.

5 Conclusion

In this paper, we explored the problem of ASL fingerspelling detection using an array of different methods. These included building our own CNN from scratch, and adapting pre-existing models for fine-tuning (VGG16) and feature extraction (MediaPipe).
Our most accurate results came from using the CNN we built from scratch. We also had good results with the VGG16 CNN, which, while easier to implement, was remarkably slower than our own model. We also developed a classifier using features extracted by MediaPipe. Although this didn't perform as well as the other models, it was much easier to integrate into demo applications, as we show in the next chapter.
However, it's important to note that our from-scratch CNN was only evaluated on our specific dataset, and not on real-world data. Because of this, we can't guarantee it will work well in different environments. On the other hand, the pre-trained models are more likely to handle real-world data well, as they have been trained on more vast and diverse datasets.
Each of the methods we explored has its own pros and cons. In order to make our own model more reliable, we could consider using more advanced and diverse data augmentation techniques, which we couldn't apply due to being computationally demanding when done on-the-fly. We could also consider training our model again on a custom dataset, including different backgrounds, hand sizes, skin colors in order to make our neural network more versatile.

Table 1: Model Performance

Model	Loss	Accuracy	ms/step
Base model	0.1356	0.9939	3
Model B	0.0518	0.9992	4
Model C	9.3140e-05	1.0000	5
Model D	1.3415e-06	1.0000	7
Hyper-parameters	0.0030	0.9992	3
VGG16 Fine-tuning	0.0048	0.9982	30
MediaPipe Feature Extractor - Unbalanced	0.1055	0.9729	-
MediaPipe Feature Extractor - Undersampling	0.1350	0.9563	-
MediaPipe Feature Extractor - Smote	0.0947	0.9770	-
Stanford University	-	0.9782	-
VIT University	-	0.8747	-

6 Demo

To demonstrate the effectiveness of our neural network for hand recognition in American Sign Language (ASL), we developed a simple application using the Python programming language. This application harnesses the power of MediaPipe for real-time hand detection and tracking using the device's webcam.
Using MediaPipe, we are able to capture hand landmarks, these are then passed to our trained neural network, which was previously described in the Transfer Learning chapter.

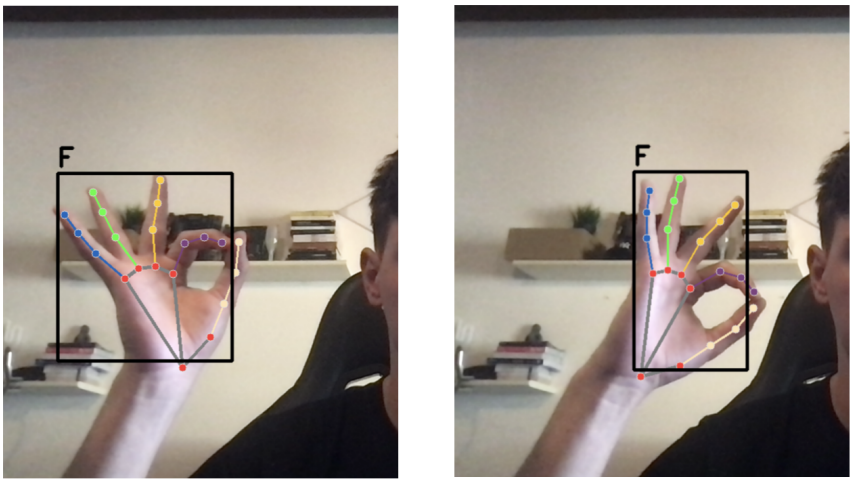


Figure 26: ASL of letter F

The application presents a simple user interface, where the real-time webcam video is displayed. When a hand is detected, landmarks are extracted and sent to the neural network for ASL classification. The classification result, representing the letter is displayed on the screen.
Our demo provides an interactive experience for users, allowing them to directly experience hand mark recognition using our neural network.

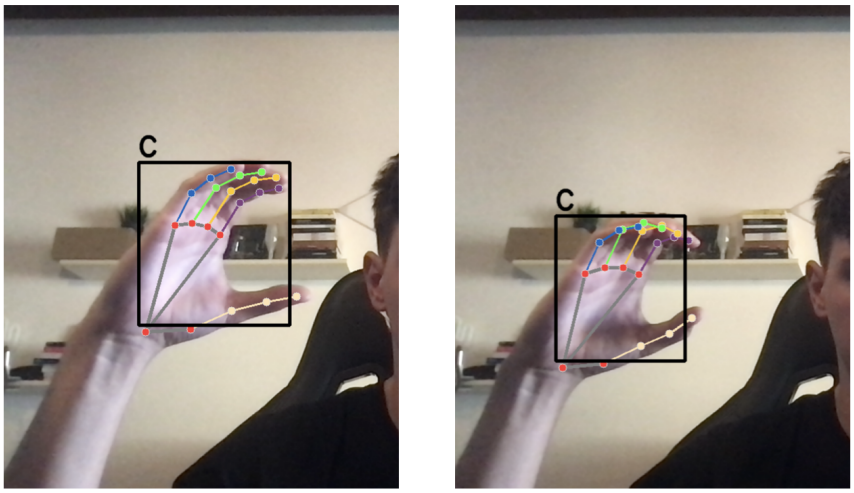


Figure 28: ASL of letter C

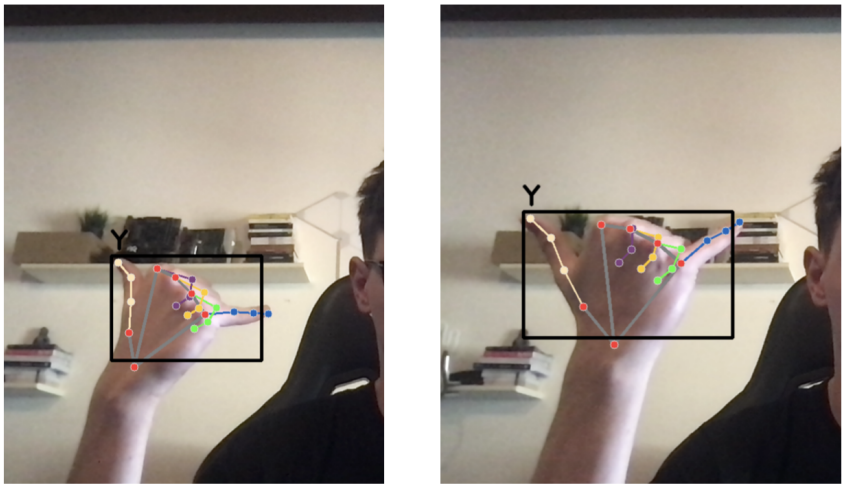


Figure 27: ASL of letter Y

The demo and all pyhton notebooks are available in the following repository .

7 References

References

[1] Website, Real-time American Sign Language Recognition with Convolutional Neural Networks, http://cs231n.stanford.edu/reports/2016/pdfs/214$_R$*eport.pdf*.

[2] Website, American Sign Language Alphabet Recognition using Deep Learning, https://arxiv.org/pdf/1905.05487.pdf.

[3] Website, Google MediaPipe Hand landmarks detection guide, https://developers.google.com/mediapipe/solutions/vision/hand$_l$*andmarker*.

[4] Website, ASL Alphabet Dataset, https://www.kaggle.com/datasets/grassknoted/asl-alphabet.

[5] Normalized RGB, Google MediaPipe Hand landmarks detection guide, https://aishack.in/tutorials/normalized-rgb/.

[6] Website, ASL Alphabet Recognition, https://www.kaggle.com/code/alanshmyga/asl-alphabet-recognition.

List of Figures

List of Tables