Group Project Report
Andy Christian, Rajkumar Conjeevaram Mohan, Daqian Dang
Deep Learning
DATS 6303
April 24, 2023

Deep Learning-Based ASL Fingerspelling Recognition System for Enhanced Communication and Accessibility

I. Introduction

American Sign Language (ASL) serves as the primary means of communication for deaf and hard of hearing individuals in the United States. Fingerspelling, a significant aspect of ASL, utilizes 26 unique hand configurations to denote the letters of the alphabet. In an effort to bridge the communication divide between hearing and deaf or hard of hearing communities, this report emphasizes the development of a sophisticated deep learning-based ASL fingerspelling recognition system. This state-of-the-art system aims to accurately recognize hand gestures and convert them into characters, fostering greater accessibility and inclusivity across diverse communities throughout the nation.

By concentrating on the accurate interpretation of ASL fingerspelling, we aim to establish a robust foundation upon which more sophisticated applications can be built in the future. By refining our deep learning model and focusing on the nuances of hand gestures that represent the alphabet, we will be able to develop a system that provides a wider range of ASL users, regardless of their signing shape and style.

In addition, our current focus on ASL fingerspelling recognition serves as a stepping stone, paving the way for future innovations that will enable seamless integration of communication through live ASL sign reading and transcription into text or conversion to speech. Expanding on this foundation, we envision the development of advanced tools and applications, such as real-time ASL interpretation for video calls, smart wearable devices for in-person interactions, and educational resources that help bridge the gap between the deaf or hard of hearing and hearing communities.

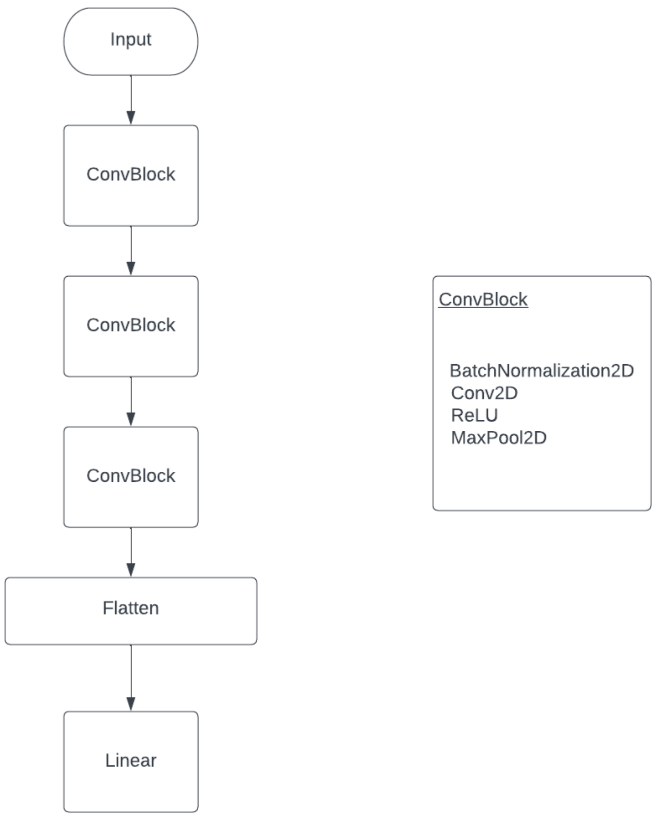
By enhancing the accessibility and inclusivity of communication technologies, we aspire to create a more interconnected world where individuals with varying communication abilities can engage with one another without barriers. This vision will not only benefit the deaf or hard of hearing community, but it will also enrich society as a whole by fostering understanding, empathy, and collaboration across different communities

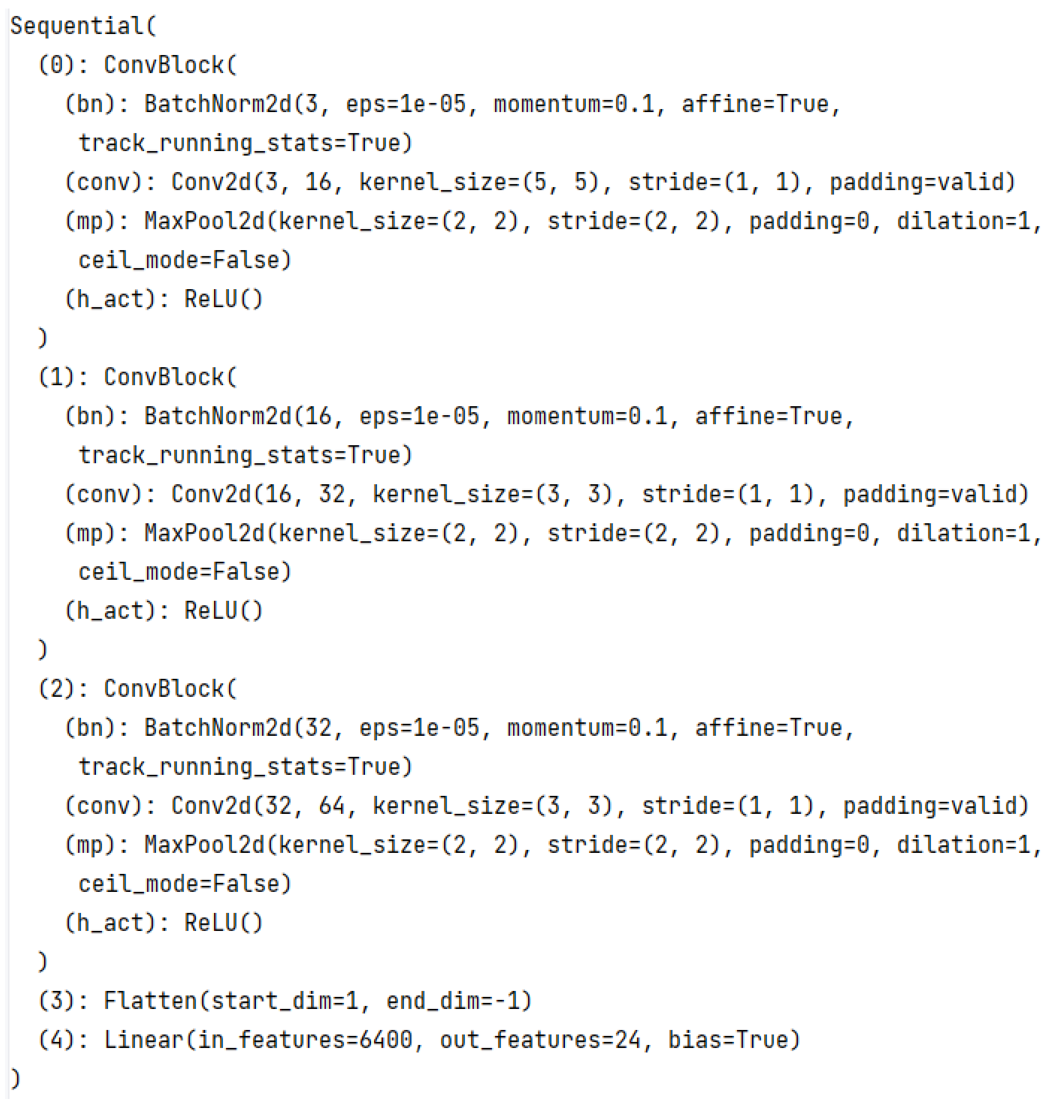
II. Description of Data Set

The dataset used in this group project was sourced from Kaggle's "ASL Fingerspelling Images (RGB & Depth)" repository. This comprehensive collection comprises color images that have been pre-cropped to focus solely on the signing hand. The dataset includes contributions from five different individuals, resulting in a total of 131,662 RGB images. Notably, the depth images available in the original dataset were not utilized in this project.

The dataset encompasses 24 distinct ASL classes derived from the original set of 26 alphabet letters. The letters J and Z have been excluded from the dataset, as their accurate representation in ASL fingerspelling necessitates the incorporation of singing motion, which cannot be captured in static images. The class distribution is fairly balanced, with a minimum of 5235 images and a maximum of 6220 images per class. This diverse and extensive dataset provides a solid basis for training our deep learning model to recognize a wider range of ASL fingerspelling variations, ultimately enhancing the model's performance and applicability in real-world scenarios.

III. Description of Deep Learning Network and Training Algorithm





```
Sequential(
  (0): ConvBlock(
    (bn): BatchNorm2d(3, eps=1e-05, momentum=0.1, affine=True,
     track_running_stats=True)
    (conv): Conv2d(3, 16, kernel_size=(5, 5), stride=(1, 1), padding=valid)
    (mp): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1,
     ceil_mode=False)
    (h_act): ReLU()
  )
  (1): ConvBlock(
    (bn): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
     track_running_stats=True)
    (conv): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=valid)
    (mp): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1,
     ceil_mode=False)
    (h_act): ReLU()
  )
  (2): ConvBlock(
    (bn): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
     track_running_stats=True)
    (conv): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=valid)
    (mp): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1,
     ceil_mode=False)
    (h_act): ReLU()
  )
  (3): Flatten(start_dim=1, end_dim=-1)
  (4): Linear(in_features=6400, out_features=24, bias=True)
)
```

A classic Convolutional Neural Network (CNN) was implemented for fitting the ASL data. It was our initial assumption that a more complex network would be required for this task; however, a simple model that we came up with performed the best relative to other versions of slightly complex CNN architectures we experimented with.

IV. Experimental Setup

To train the model, a convolution neural network (CNN) was used. Different versions of the model architecture included convolution layers, max pooling layers, dropout layers, and batch normalization layers. The data was divided into training, validation, and test sets in a stratified method. The training set was used to update the weights of each layer, the validation set to check how well the model was learning, and the test set was used to evaluate the model's performance on unseen data.

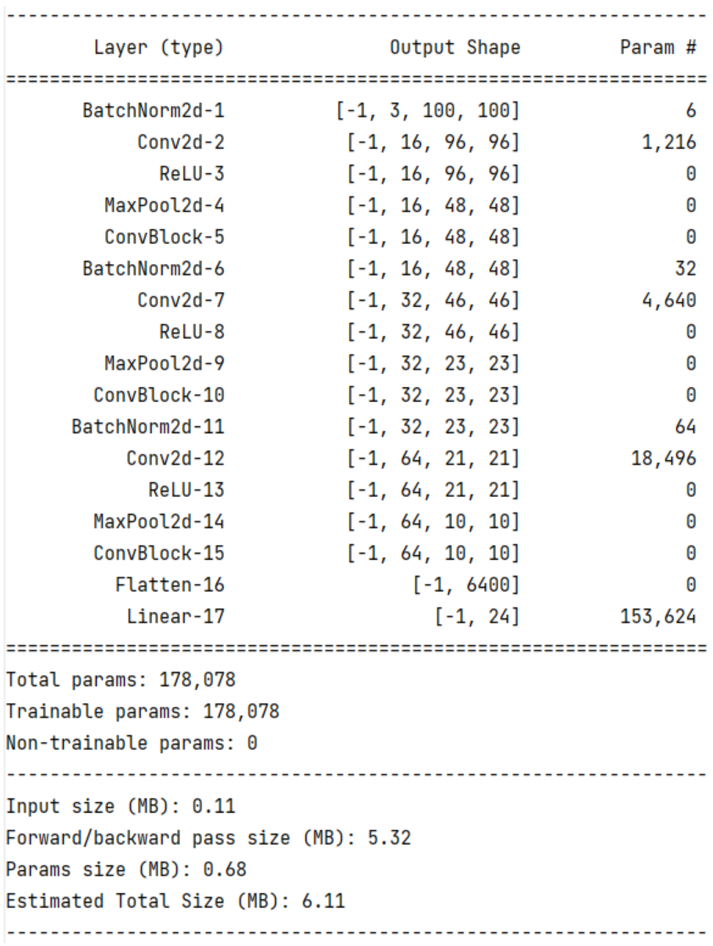
The model is composed of multiple ConvBlocks which contains BatchNormalizationn, Convolution2D, ReLU, and MaxPool2D and a Linear layer. An initial model with one linear layer produced an f1-score of 0.982. As part of the experiment process, an additional linear layer was included that produced a subtle improvement in the performance but not by a large margin. Cross entropy loss was used to verify the convergence and whether or not to continue training the model. Inside the Adam optimizer, L2 regularization was used to penalize incorrect predictions. Since there was good balance between all 24 character classes, accuracy on the validation set was used to determine whether to save the current version of the model after each training epoch. After the model was finished training, macro F1 was used to judge the model’s performance predicting on the test data set.

Hyperparameter tuning was used to optimize the model and find its best version. The parameters experimented with included batch size, learning rate, number of filter maps, size of filter map kernels, and size of the L2 penalty. Additional experimentations included adjusting the number of convolutional layers, and including dropout and batch normalization. The best hyperparameters, were found via structured and systematic testing. It is our understanding that having batch normalization in the model would enable having larger learning rate since each step towards the optimal set of parameters in the parameter space is equally proportional. Hence, this allowed us to crank up the learning rate for faster convergence.

Several techniques were used to avoid overfitting. Early stopping was implemented on the loss value of the validation set in order to stop the training if the model was no longer improving, thus avoiding training on noise in the training set. L2 regularization was used to penalize incorrect predictions. Additionally, while the model was allowed to continue training so long as the validation loss continued to decrease, the weights of the layers were only saved when the model’s accuracy on the validation set was better than previous epochs.

V. Results

Rajkumar’s Architecture:



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
       BatchNorm2d-1         [-1, 3, 100, 100]               6
            Conv2d-2          [-1, 16, 96, 96]           1,216
              ReLU-3          [-1, 16, 96, 96]               0
         MaxPool2d-4          [-1, 16, 48, 48]               0
         ConvBlock-5          [-1, 16, 48, 48]               0
       BatchNorm2d-6          [-1, 16, 48, 48]              32
            Conv2d-7          [-1, 32, 46, 46]           4,640
              ReLU-8          [-1, 32, 46, 46]               0
         MaxPool2d-9          [-1, 32, 23, 23]               0
        ConvBlock-10          [-1, 32, 23, 23]               0
      BatchNorm2d-11          [-1, 32, 23, 23]              64
           Conv2d-12          [-1, 64, 21, 21]          18,496
             ReLU-13          [-1, 64, 21, 21]               0
        MaxPool2d-14          [-1, 64, 10, 10]               0
        ConvBlock-15          [-1, 64, 10, 10]               0
          Flatten-16                [-1, 6400]               0
           Linear-17                  [-1, 24]         153,624
================================================================
Total params: 178,078
Trainable params: 178,078
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.11
Forward/backward pass size (MB): 5.32
Params size (MB): 0.68
Estimated Total Size (MB): 6.11
----------------------------------------------------------------
```

As a neat coding practice, I placed BatchNormalization, Convolution, and Maxpool layers inside a Convolution Block called “ConvBlock” to improve readability and debugging. It is understood that having a BatchNormalization would enable having a larger learning rate, which results in faster convergence because traversing in error surface from each dimension is equally proportional; in simple terms, batch normalization would ensure that each dimension is of same unit and hence allows the model to converge faster.

This arhictecture act as the baseline model that produced ~0.988 f1-score (macro) on test set. In order to make it even more efficient, different techniques such as Depthwise separable convolution was experimented, in order to reduce the overall number of parameters, and to improve the representational efficiency. However, the results did not outperform the current architecture; Hence Depthwise separable convolution was not opted.

This model consists a total of 166,558 parameters, and offers greater performance in terms of accuracy on unseen data.

Hyperparameter for Raj’s model	
Learning rate	0.001
minibatch size	500
prefetch_factor	30
Kernel size	5x5, 3x3, 3x3
N of layers	4
Epoch Number	30
Optimizer	ADAM

Results for Raj’s model	
Best N of Epoch	25
Train accuracy	1.0
Validation loss	0.09125
Validation accuracy	1.0
F1 score for test set	0.988856

Daqian’s Optimization:

Hyperparameter for Daqian’s model	
Learning rate	0.001
minibatch size	512
Kernel size	3 x 3
N of layers	3
Dropout implemented in each layer	0.3
Batch normalization implemented in each layer	16, 32, 64
Epoch Number	30
Optimizer	ADAM
L2 regularization (weight decay)	0.001

The model achieved a train accuracy of 0.950 and an even higher validation accuracy of 0.966, showing its potential for recognizing ASL fingerspelling on unseen data. The test F1 score of 0.965 further supports the model's effectiveness in this task. See result table below.

Results for Daqian’s model	
Best N of Epoch	29
Train loss	0.137
Train accuracy	0.950
Validation loss	0.145
Validation accuracy	0.966
F1 score for test set	0.965

Using ADAM optimization with L2 regularization also helped the model succeed, allowing it to perform well during the training process. The final performance metrics show that the chosen architecture and hyperparameters are suitable for ASL fingerspelling recognition.

When experimenting with a pretrained model, like EfficientNet, it was found that the computation time was much slower compared to the simpler model. This slower performance is due to the complexity of the EfficientNet architecture, which might not be necessary for the ASL fingerspelling recognition task. In fact, the results show that the simpler model was more effective for this specific application.

This finding highlights the importance of selecting the right model architecture for the problem. While complex models like EfficientNet are effective in many computer vision tasks, they may not always be the best choice, particularly for less complex tasks or when computational resources are limited. In such cases, simpler models with well-tuned hyperparameters can deliver better performance and faster training times, as demonstrated in this project.

Optimization Experiments and Results for Andy's Model:
Andy created four optimization versions of the model. He experimented with different learning rates, batch sizes, number of filters, kernel sizes of the filters, adding extra convolution layers, using dropout, including L2 regularization, and adjusting the patience of early stopping. The details of the model with the best results are in the following table:

Hyperparameter for Andy's model	
Learning rate	0.001
Minibatch size	64
Kernel size	5 x 5
N of convolution layers	3
Dropout implemented in each layer	0.3
Filter maps per layer	16,32,64
Epoch Number	108
Optimizer	ADAM
L2 regularization (weight decay)	0.001
Early Stopping patience	30

The model using these parameters was able to obtain an accuracy of 1.0 on the training and validation sets, and the test set achieved a macro F1 score of 0.9874. The loss values were, on average, between the 0.11 - 0.13 range. These loss values represented an average "confidence" of around 0.91-0.93 in predicting the most likely class for each observation. The results of the best model are below:

Results for Andy's model	
Best N of Epoch	108
Train loss	0.109
Train accuracy	1.0
Validation loss	0.13175
Validation accuracy	1.0
F1 score for test set	0.9874

Despite adjusting numerous parameters, it was observed that batch size and learning rate seemed to have the most meaningful impact on improving the results of the model. Larger batch sizes and smaller learning rates led to extremely low loss values (0.02-0.01), however, with such versions of the model the macro F1 of the test set only reached about 0.9, which was surprising for such low loss. It remains an open question as to why the model performed better in the version that had ~0.1 loss vs the model that had ~0.01 loss.

VI. Summary and Conclusion

The authors of the 2011 study from which the data was taken used a random forest model in their predictions. With such a model, the authors of the original study were able to achieve about 0.7 accuracy, on average, across letters. The introduction of CNNs since that time has greatly improved the ability to solve this type of image classification problems. With relatively simple CNN models, requiring little time and cost, nearly perfect results across all letters was achieved. Furthermore, simple model architecture proved the most effective in making predictions. For the most part "default" values, such as 3 convolution layers, using 16-32-64 filter maps, small kernel sizes, and typical batch sizes (64) had the best results.

This project was a good proof of concept that CNNs could be used effectively for ASL interpretation. Taking what was done here could be a launching point for more robust model creation, such as a model able to take in live feed of not only ASL fingerspelling but of full words, and then convert them directly into text or speech. An even more advanced version of such a tool could incorporate Natural Language

Processing (NLP) to evaluate whether the predicted word “made” sense in the context of the given sentence, thus adding in a further accuracy check and improving the quality of the predictions.

VII. References

Bowden, R., Pugeault, N. (2011). Spelling It Out: Real–Time ASL Fingerspelling Recognition. *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*. https://doi.org/10.1109/ICCVW.2011.6130290

Geislinger, V. (2020). ASL Fingerspelling Images (RGB & Depth) (Version 2)[Data Set]. Kaggle. https://www.kaggle.com/datasets/mrgeislinger/asl-rgb-depth-fingerspelling-spelling-it-out