Sign Language Recognition and Translation

Charles Yu yrtcha@umich.edu
Al Stahl stahlal@umich.edu
Tushar Jayaram tusharj@umich.edu
Sai Chengalvala schengal@umich.edu

1. Introduction

Sign language translation has been a challenging problem in our society. Across all age groups, approximately 600,000 people in the United States are deaf, [5] and many of them could not interact with the real world due to a shortage of human ASL (American Sign Language) translators. Researchers have found that many deaf people suffer from disability, and the unemployment rate for the deaf community has increased in the last decade. The demand for sign language translation is high, yet there is a shortage of those with the required skill-set to service the deaf community. Although ASL transcription is not the same problem as translating spoken language into sign language, this is a necessary prerequisite for that. If we can build a successful classifier that translates sign language into spoken language, video synthesis techniques can be leveraged to perform the reverse translation.

To ensure communications across all communities, our group aims to create a program that can transcribe sign-language from a video-file in real-time. In particular, we sought to build a deep-learning model to achieve a test-set accuracy well above that of a random guess. The main objective of this model is to classify different hand gestures into written English through machine learning techniques. To construct our translation program, we built off our work from previous homework on convolution neural networks to develop a similar architecture.

Techniques in computer vision could be leveraged as powerful tools in sign language interpretation. ASL is a language designed to communicate and convey ideas entirely through the use of visual stimuli in the form of body, hand and/or arm movements. As humans, we process this data optically and are able to translate this visual input into the idea, word or phrase that the signer meant to communicate. In comparison, Convolutional Neural Networks have been found to be extremely robust in interpreting visual input. With this understanding in mind, a 3D CNN architecture should provide accurate transcriptions when trained using well organized and compiled data.

Several projects have sought to solve this same problem of ASL transcription with scattered successes across architecturally distinct models. Most of the work done in this area has been concentrated in the compilation of efficient and complete datasets. Some of the most notable projects that provide researchers with useful datasets are Word-Level American Sign Language and ChicagoFSWild (short for Chicago Finger-Spelling in the Wild, as it pulls its data from videos of real signers teaching ASL for others to learn and study). For our project, we chose to employ ChicagoFSWild's dataset [1].

Our model employs a multi-layered, deep-learning model which utilizes 3D convolution as its basic foundation. By computing cross entropy loss and performing back-propogation, we sought to teach our model sign language through well focused training and revision.

2. Approach

When we first approached the idea of transcripting sign language videos, we discussed two possible methods. The first surrounded a nearest neighbor approach where our program aimed to identify hand signals from a video frame and identify the closest local neighbors for this frame, while the second attempted to extract specific video features from 3D Convolutional Neural Networks. After about a day of deliberation, we decided on the latter, mostly due to the power of PyTorch's [3] CNN feature.

The bulk of our approach was split up into two major parts. The first surrounded loading in and preprocessing many gigabytes of video data taken from the ChicagoFSWild fingerspelling dataset, and the second involved writing the actual architecture of our model, as well as the training and tuning of our parameters to minimize our loss and maximize our success rate.

2.1. Data Loading

Finding an efficient way to load in the ChicagoFSWild dataset served to be the first roadblock in our approach. After some intermediate trial and error, we settled on the following loading structure. First, we loaded in ChicafoFSWild.csv into a pandas dataframe [6] to store our mapping from video frames to their corresponding labels.

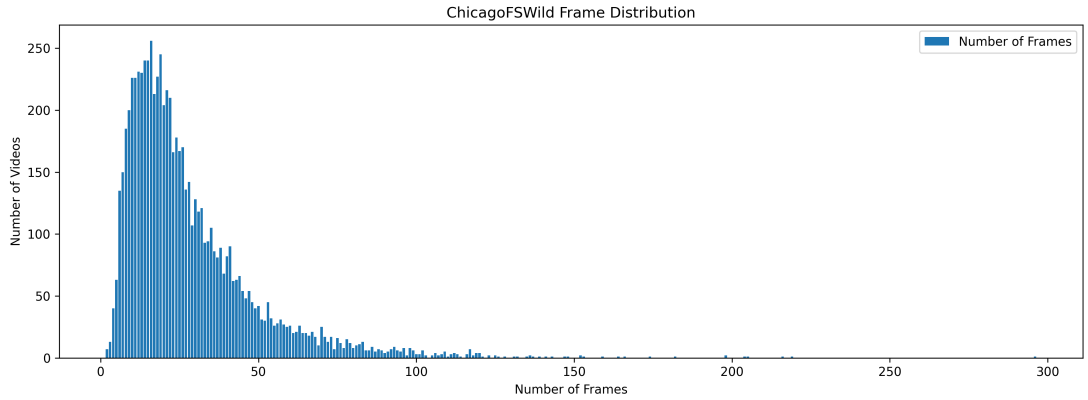


Figure 1. Distribution of $\frac{\#\ of\ Frames}{Video\ Sequence}$ Throughout ChicagoFSWild

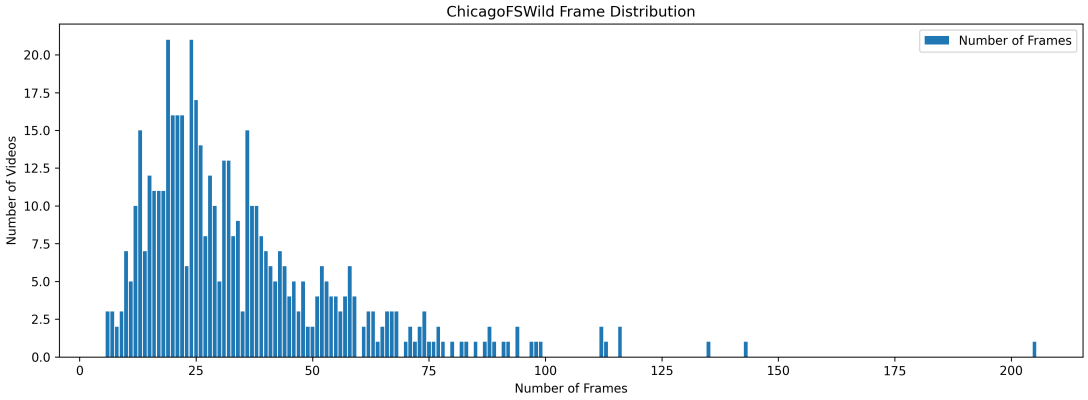


Figure 2. Distribution of $\frac{\#\ of\ Frames}{Video\ Sequence}$ Throughout Our 500 Video Subset of ChicagoFSWild

Next, we decided to build a list of 4d tensors of the following structure:

$$\left[\frac{\#\ of\ Channels}{Image} \quad \frac{\#\ of\ Frames}{Video\ Sequence} \quad Height \quad Width\right]$$

We set the number of channels per image as 1, since colored frames served no tangible purpose in identifying signs. However, the number of frames per video and frame size were not uniform across different videos in our dataset. As a result, constructing our 4D tensor proved impossible without applying some standard of normalization. We decided to standardize frame size to 150 by 100, and chose a number of frames to be 18, which corresponded to the 50th percentile of the dataset's distribution (**See Figure 1 and 2**). For any sequences with fewer frames than this 50th percentile, we repeated the last frame until we reached the appropriate size. For any sequences with more frames than this 50th percentile, we randomly chose frames to remove (maintaining time order). We stored each of these 4D tensors in a list, then split it into a training set and test set. Finally, we finished preparing the data by encoding our labels (strings) to integers using scikit-learn's $LabelEncoder()$. [4]

2.2. Model Creation

For our model (**See Figure 3**), we began with a 3d convolution layer with 1 input channel, 6 output channels, and a stride of 5. We then used a max 3d pool with a kernel size of 3 and a stride of 3. Our second 3d convolution layer transformed from 6 input channels to 32 output channels with a stride of 2. Our fully connected layers started with 21504 input features and outputted 512 features, and then took in these 512 inputs and had output features equal to our total number of labels.

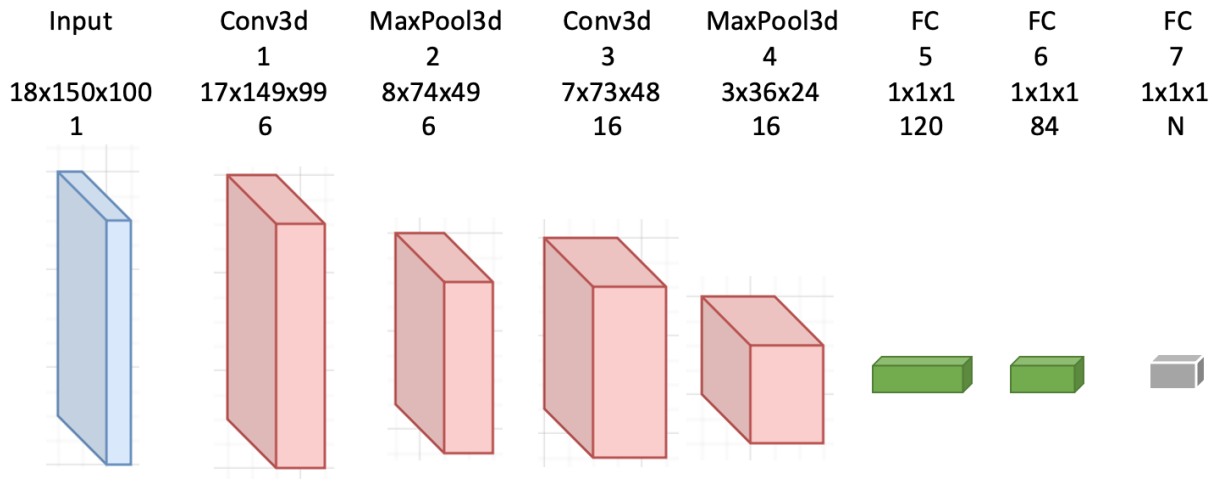


Figure 3. 3D CNN for ASL Transcription

We set our learning rate equal to $5 * 10^{-2}$, our $weight\_decay$ to $2.5 * 10^{-5}$, our $batch\_size$ to 8, and our num_epochs to 10. We trained and evaluated our data by iterating through each epoch, calculating the cross entropy loss for our model, backpropogating our gradients to all the tensors in our network, and updating the trainable weights. Finally, we ran our test set (15% of our data) through the same cross entropy loss function and printed out our testing accuracy.

2.3. Difficulties

2.3.1 RAM Crash

One of the most significant difficulties we faced in our building and training of our model was the regular RAM crashes we had to endure on Google Colab. Our most computationally intense portion of code involved loading in a list of 4d tensors of videos into a single variable, called our tensor list. For the full 500 videos, this meant that our tensor list variable had over 2 gigabytes of data loaded into it based on our initial computation of 3 channels by 300 pixels by 200 pixels. Needless to say, we ran out of RAM by the first training epoch. To combat this, we initially changed our image to just one channel by 300 pixels by 200 pixels, which seemed to temporarily fix the issue. However, we would still face occasional crashes. Finally, we recognized that changing the pixel size had almost no effect on the success of our model. We decided to half both the image height and image width to 150 by 100, and our RAM issues were fixed.

2.3.2 Managing Large Dataset

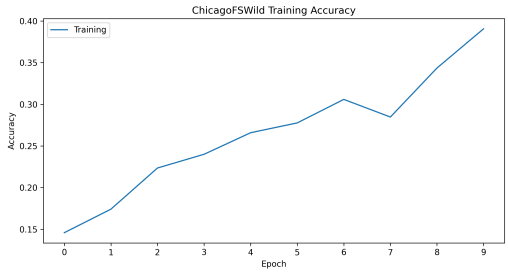
It was much more computationally intense to copy many files to colab than it was to copy a compressed file into colab and then decompress afterwards. Furthermore, once our data finally did load, we ran into countless issues actually training it that were nearly impossible to localize. To combat this, we slowly built up our program off of just 28 video files and gradually increased the number. These were issues that we didn't anticipate going into the project and had to learn as we went through the project.

3. Experiments

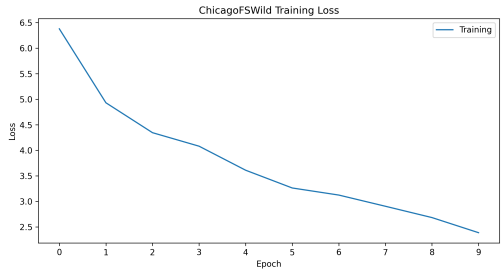
3.1. Data

As mentioned earlier, our dataset was the ChicagoFSWild ASL Finger-spelling Data Set, a group of videos collected from online sign language teachers. The data was split up into folders, with every folder corresponding to a certain video collected from online. Each of these folders had subfolders, and the individual subfolders matched up with a certain "label" indicated in the csv file, such as a combination of letters, like "dc", or a full word, like "aslized". Each subfolder contained a variable amount of jpeg images that our program analyzed.

This data makes sense for our purposes because it has huge amounts of video data (7,304 videos and 205,139 jpeg files), which is a much more difficult medium to transcribe than just images. Furthermore, none of the videos are necessarily standardized in terms of number of frames or frame size, adding another layer of complexity to our model. This helps extend our model outside just a very calculated set of data and aligns our model more to a real interpreter than just a program that needs a specific set of inputs to work. To facilitate the speed of our model and avoid running into RAM issues, we selected a subset of just 500 videos, leaving us with 17,502 jpegs to analyze.



(a) Training Accuracy Plot



(b) Training Loss Plot

Figure 4. Training Plots over 10 Epochs

3.2. Metrics of Success

We measured success mostly on the test set prediction success rate, as we found this to be the perfect medium between having enough data and not overfitting our training model. Some potential downsides that we came across when doing this was the variability between success rates: often times, back to back runs would result in very different success rates. Furthermore, as mentioned earlier, we needed to find the best way to avoid overfitting on our training data and allow our model to classify signs performed slightly differently than they were in our training data as their correct label.

3.3. Qualitative results

As shown in this section, **Figure 4a** reports the training accuracy across epochs. Training accuracy starts off at 15%, but as epochs continue, our training accuracy slowly begins to approach 40%. Our group considered this as a potential symptom of overfitting our training data, but saw that this model was our best model on our testing set as well.

Figure 4b similarly references the training loss (calculated by our cross entropy loss function) across epochs, and is gradually approaching 0 as the number of epochs increases. By the 10th epoch, our loss had reached about 2.5 from its initial value of approximately 6.5.

3.4. Quantitative Results

After fine tuning our model, we reached a test accuracy of 20 percent, meaning we accurately predicted 15 out of 75 possible labels. This was a very promising outcome, as we have found even the best video sign language transcriptors maximizing their test accuracy at around 30 percent.

4. Implementation

We used PyTorch to build our deep network. We primarily took advantage of CNN, NN, and pooling layers, as well as the library's built in cross entropy function. We also used Pandas to load in the ChicagoFSWild CSV file to store and extract our data. Scikit-learn was used to split our data into training and test splits and encode our labels with int values to allow for our CNN to process. We also used the numpy [2] library to store all of our arrays and tensors.

References

[1] J. Keane J. Michaux D. Brentari G. Shakhnarovich B. Shi A. Martinez Del Rio and K. Livescu. "American Sign Language fingerspelling recognition in the wild". In: *SLT* (2018).

[2] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[3] Adam Paszke et al. "Automatic differentiation in PyTorch". In: (2017).

[4] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[5] C. Reilly and S. Qi. "Snapshot Of Dear And Hard Of Hearing People, Postsecondary Attendance And Unemployment". In: *Deaf Studies Deaf Education* (2011).

[6] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.