CS230

SignNet: A Neural Network ASL Translator

Daniel Book and Grant Fisher
CS 230
Stanford University
`dbook@stanford.edu` and `gfisher7@stanford.edu`
https://github.com/dbook13/CS-230-Final-Project

Abstract

This work presents a supervised learning framework with convolutional neural networks and recurrent neural networks for American Sign Language recognition, where input data is a raw, head-on, video of a person performing a sign. Our overall architecture incorporates three distinct stages in translation from ASL to english. We first takes in an input video and divides it into many still-image frames. Following this, we feed the images into the OpenPose framework in exchange for (x, y) coordinates of key points on a person's body, face, and hands. Finally, we send these key points to an RNN, which outputs a prediction of what sign a person is performing. Our model achieves 93% accuracy on videos it has never seen before.

1 Introduction

For our project, we decided to address the problem of sign language translation. According to the Journal of Deaf Studies and Deaf Education, over 1 million people in the United States are functionally deaf. While American Sign Language (ASL) allows people with hearing loss to communicate in a robust way, most regular Americans don't know ASL and would have a difficult time understanding and responding. Our project looks to help bridge this gap by finding a simple translation solution. Because ASL is a dynamic language that incorporates hand motions, body language, and facial expressions, video is the only medium that can fully capture it.

Our approach to solving this problem involved converting videos of somebody signing a word and converting it to data about the location of a person's body, face, arms, and hands for each still frame in the video. The input to our algorithm is the location data for each image, which we pass into a recurrent neural network to output a prediction on if a word has been signed in the previous portions of the video.

To convert videos into location data, we used a framework called OpenPose. This is a convolutional neural network keypoint detector that takes in an image of a person and outputs a JSON file with (x, y) coordinates corresponding to the location key points on a person's body, hands, and face. This allows us significantly reduce variability in input relating to different people, clothing, and lighting, because OpenPose simply outputs (x ,y) coordinates of bodily features. This essentially regularizes our data that is then fed into an RNN for sign prediction.

2 Related work

Sign language translation is a task usually handled manually by translators, but within the past 20 years there has been a surge in the development of models to undertake this task automatically. Using machine learning to translate sign language is a well studied problem that has been approached through many different tactics, but there is not yet a robust and reliable solution. One of the first machine learning approaches, involved statistical representations of signs. In a 2004 paper by Bungeroth and Ney(1), a statistical model to translate sign language is presented, but even in the paper it is described as intensive and troublesome. As such, it was quickly replaced by neural network (NN) models.

Preliminary NN models for sign language translation mainly focused on translating still images which pretty much limited their scope to letters and numbers. A 2017 paper by Bheda and Radpour(2) describes a convolutional NN model to predict letters (excluding j and z which require movement) and simple numbers and a 2007 paper from Akmeliawati et al.(3) described a translation model using color segmentation and neural networks. A different approach that allowed for video translation (and thus full ASL translation) involved using a glove to track motion as described in the 2016 paper by Wu and Jafari(4).

Today the most state of the art models use feature extraction as well as NN to accomplish this task. A 2017 paper by Cui, Lui, and Zheng(5) details which achieves robust results using only video as an input. The model most similar to ours comes from a 2013 paper by Chai et al.(6) detailing an approach to sign language translation that uses Xbox's Kinect system to track hand movement and depth in video. However, there is not yet a model that extracts features from hands, body, and face to make translations.

3 Dataset and Features

When we first decided on sign language translation for our project, we thought it would be easy to find a large, pre-existing dataset of videos of different signs. However, as we began to look for useable data, we found that most datasets only contained one video per sign. In sign language, the same sign can differ minutely from one instance to another even when it is the same person making the sign; as a result, we decided to make our own dataset so that we would have multiple videos by two different people for each sign.

To make the dataset, we first chose 10 different ASL words and filmed ourselves signing each words 10 times using photobooth. Since we each did this, we ended up with 20 videos per sign and 200 videos total. To convert these videos into accurate and usable data, we needed to use OpenPose to extract information about the location of different body parts throughout each video. However, OpenPose was only able to accept images, so, to make these videos useable, we used ffmpeg to convert each video to a sequence of jpeg images. After passing in these images, OpenPose returned a JSON file for each image containing all the keypoint coordinates spotted in that image.

As a final preprocessing step, we wanted to read the JSON files into a numpy array of shape (200, 150, 390) where 200 is the number of videos, 150 is the maximum number of frames per video, and 390 is the amount of data points in an OpenPose JSON file for one image. To do this, we needed to iterate through each sequence of images one video at a time. By our naming scheme when creating the videos (Word-Name-Number), we were able to do this by sorting the JSON files, which were all in the same folder, alphabetically. However, our one oversight was that the alphabetical sort intermixed photos from videos numbered 1 and 10 so we had to write a script to rename all images that came from videos ended in 1. Then we initialized a numpy array of zeros to the desired shape (which also ensured any video with less than 150 frames would be zero padded to the desired length) and read the data sequentially from each video into the array. Finally we had input data that would be readable by a keras model.

Since we created our dataset, we also needed to label each video so our model would know what to predict. To do this, we scrolled through the sequence of images for each video and noted what image in the series corresponded to the end of the sign. We then initialized a numpy array of zeros to shape (200,150,11) where 11 was going to be a one hot vector corresponding to the alphabetical ordering of the words we had chosen (unless the value in the first position of the array was one which would mean no sign). Then, using the positions in the sequences of images we had previously noted, we set that position and the following 14 positions equal to the one hot representation for the correct words and set all other positions in the video equal to the one hot representation for no sign (inspired by trigger word detection). At last we had created functional version of both input and output data.

Because we had only 200 videos worth of data, we decided on an 80-20 split resulting in 160 videos in our training set and 40 videos in our test set.

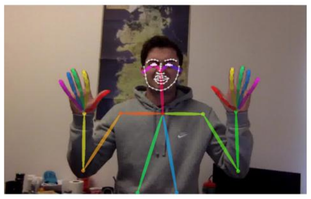

Figure 1, sample image from a sign video for “awful” with the OpenPose keypoints rendered on top of it

4 Methods

We fed the 9,360,000 (160 x 150 x 390) input array into the first layer of our RNN. We began with two LSTM layers with 128 units each. Following this, we feed into a LSTM layer with 64 output units. From there, we feed everything into a 64 unit LSTM layer. Finally, we send everything through a time distributed dense layer with a softmax activation, resulting in an 11 element vector.

We chose a softmax activation function at the final layer because we wanted to have our model output a probabilities rather than a strict prediction. This allowed us to go through an see where the model was making mistakes and how confident it was in its prediction. This information would have been lost with a strict prediction. A simple argmax can convert this vector into a strict prediction.

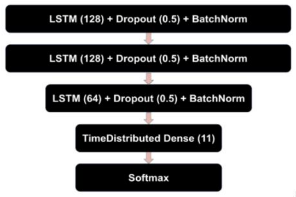
LSTM (128) + Dropout (0.5) + BatchNorm
LSTM (128) + Dropout (0.5) + BatchNorm
LSTM (64) + Dropout (0.5) + BatchNorm
TimeDistributed Dense (11)
Softmax

Figure 2, RNN architecture

The model incorporates 50% dropout after the four LSTM layers. This dropout rate was inspired by the paper, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting” by Srivastava et al. Along with this, we use Batch Normalization following the dropout. These two techniques help our model avoid overfitting, which were large concerns of ours because of the small size of our dataset. Dropout is a paramount aspect of our design because of the way we created the ground truth (y) vector. For each image in our implementation, we labeled (y) to predict that the person was not signing anything until the completion of the sign. At this point, we labeled (y) to reflect which word had just been signed, and gave this same (y) vector to the next 14 images (15 images in total). Thus, for most of a given set of images that make up one video, the ground truth vector (y) told the model that the person was not signing anything. A high dropout rate allowed us to keep our model from simply learning to predict every single time that the person was not signing anything. We hoped that the model would randomly decide to “drop-out” neurons relating to downstream predictions of “no sign.”

The model uses categorical cross-entropy as its loss function. Since our RNN architecture involves a 11 dimensional softmax output (y-hat), and our ground truth 11 dimensional vector (y) is one-hot, this loss function works well.

$$CCE = -\frac{1}{N}\sum_{i=0}^{N}\sum_{j=0}^{J} y_j * log(\hat{y_j}) + (1 - y_j) * log(1 - \hat{y_j})$$

Our optimization algorithm is Adam. Although we tweaked the parameters such as the learning rate and decay rate, we found that the default values suggested in the original paper by Diederik Kingma and Jimmy Ba to work best. These values were a learning rate of 0.001, a Beta 1 value of 0.9, a Beta 2 value of 0.99, and no decay.

5 Experiments/Results/Discussion

Our final method allowed us to achieve 94.9% categorical accuracy on our training data, and 93.0% categorical accuracy on data the model has never seen before. Along with this, we had an average accuracy score of 0.9904, an average recall score of 0.9475, and an average ROC AUC score of 0.971. We used these metrics to compare several prototype models against each other. More important than these metrics alone was the F score. In order to determine if our model was working correctly, we wanted a metric that took false positives and false negatives into account. This is especially important for our application because we have an uneven class distribution, and because a model could simply predict that there was never a sign and still get around a 85% raw accuracy. Our final model had an F score of 0.971.

To visualize this, we created a confusion matrix. Because we had 11 classes, our matrix is 11x11. We also normalized everything because most of our input images had a ground truth label of 'no sign' and thus our matrix would not be very informative because the row associated with 'no sign' would have much larger values than the other signs and would make it hard to compare. Figure 1 shows an early attempt in RNN architecture. This model had different learning rates and lacked some regularization techniques that are present in our final product. Even though this early model had 98% accuracy, its F score was only 0.79.

Much of our time was spent getting the model to predict actual signs rather than 'no sign' every single time. The confusion matrix from our early attempt, visualized in Figure 3, shows that the model was not learning what we had hoped.

Our final model, visualized by Figure 4, shows that our model was able to very accurately predict signs. One issue we had trouble with was getting the model to properly classify 'yellow' (noted by the 10th index in our confusion matrix). We believe this is due to the nature of the sign for 'yellow' and the ability for OpenPose to track signer's hand throughout. We chose this sign as one of our ten examples because it presents a challenge by incorporating quick motion and a side-view of one's hand. Other signs we selected, including 'house' and 'teacher' both involved almost the exact same hand motions, but differed in how they began. While the signer's thumbs and middle fingers started out touching in the sign for 'teacher', the signed began with an open hand for 'house'. Other than this difference for a fraction of a second, the signs were almost the exact same. Our model was able to effectively differentiate between these subtleties.

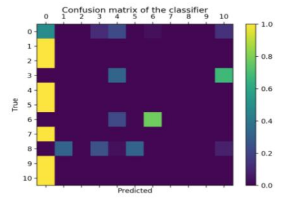

Figure 3, confusion matrix of an early model.

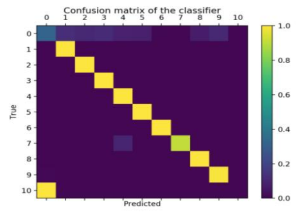

Figure 4, confusion matrix of final model.

6 Conclusion/Future Work

Our work is the first we have seen that can take in a raw video and run it through OpenPose to extract key features that we then feed into an RNN for a prediction. We were surprised to find that optimal RNN design came from a simple architecture, rather than a complicated one. Our approach may have been successful due to our feature detector selection. Originally, we had planned on running input video through our own CNN in order to extract key features like hand position and facial expressions; however, the OpenPose framework was much more effective and a much faster model. The beauty of selecting this framework for our feature extraction step was that it eliminated much variability in the data that would be fed into our RNN. Variability in data due to different people doing signs in varying lighting and clothing was attenuated before the final step in our pipeline.

At this point in our project, we do not have nearly enough data to train on, and our model is designed with this in mind. A more complicated RNN would likely overfit to our training data and would not perform well on data it has never seen before.

The tedious work of creating and labeling raw input videos would allow us to tweak our model's final RNN architecture to become more complicated and deep; allowing it to learn more intricate signs with greater accuracy. Beyond this, our model only translates individual signs at the moment. By appending another RNN at the current output of our model, we could translate ASL sentence structure back into traditional English sentence structure, thus creating a true end-to-end translator.

7 Contributions

Grant Fisher worked on building the RNN model and experimenting with different hyperparameters to increase categorical accuracy based on calculated metrics. He also handled the use of the OpenPose framework to get body keypoint data for each video.

Daniel Book worked on using FFmpeg to separate the videos into sequences of images. He also converted the JSON data returned from OpenPose into a valid input array and created the output array.

Grant and Daniel both filmed themselves signing the 10 words and located the end of the sign in each video.

References

1. Bungeroth, Jan, and Hermann Ney. "Statistical sign language translation." Workshop on representation and processing of sign languages, LREC. Vol. 4. 2004.
2. Bheda, Vivek, and Dianna Radpour. "Using Deep Convolutional Networks for Gesture Recognition in American Sign Language." arXiv preprint arXiv:1710.06836 (2017).
3. Akmeliawati, Rini, Melanie Po-Leen Ooi, and Ye Chow Kuang. "Real-time Malaysian sign language translation using colour segmentation and neural network." Instrumentation and Measurement Technology Conference Proceedings, 2007. IMTC 2007. IEEE. IEEE, 2007.
4. Wu, Jian, Lu Sun, and Roozbeh Jafari. "A wearable system for recognizing American sign language in real-time using IMU and surface EMG sensors." IEEE journal of biomedical and health informatics 20.5 (2016): 1281-1290.
5. Cui, Runpeng, Hu Liu, and Changshui Zhang. "Recurrent Convolutional Neural Networks for Continuous Sign Language Recognition by Staged Optimization." The IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2017.
6. Chai, Xiujuan, et al. "Sign language recognition and translation with kinect." IEEE Conf. on AFGR. 2013.
7. Z. Cao, T. Simon, S. Wei and Y. Sheikh, "Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields," 2017.
8. T. Simon, H. Joo, I. Matthews and Y. Sheikh, "Hand Keypoint Detection in Single Images using Multiview Bootstrapping," 2017.
9. S.Wei, V. Ramakrishna, T. Kanade and Y. Sheikh, "Convolutional pose machines," 2016
10. Mitchell, Ross E. "How many deaf people are there in the United States? Estimates from the Survey of Income and Program Participation." Journal of deaf studies and deaf education 11.1 (2005): 112-119.
11. Bellard, Fabrice et al. "FFmpeg." 2000, Github repository, https://github.com/FFmpeg/FFmpeg

12. Chollet, Francois et al. “Keras.” 2015, Github repository, https://github.com/keras-team/keras

13. Anon. (1996). NumPy: a numerical extension for the computer language Python. Version 1.7. The Regents of the University of California

14. Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

15. Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

16. Srivastava, Nitish, et al. "Dropout: A simple way to prevent neural networks from overfitting." The Journal of Machine Learning Research 15.1 (2014): 1929-1958.