Being able to communicate verbally is something that most people take for granted. For the mute and deaf in America, however, communication can only be done more arduously using methods such as pen and paper or the American Sign Language, or ASL. While the ASL can be an effective communication method, most people do not know it, which would defeat its purpose. Therefore, using machine learning techniques to classify ASL's different letters of the alphabet can be useful to both those who know it and those who do not. It would make communication easier between two parties if only one party knows sign language.

I will be using the ASL Alphabet dataset from Kaggle, which includes a training set of 87,000 images that are 200x200 pixels. Each image can be one of the 26 letters, a space, delete, or nothing. The test set only includes 29 images, but I plan to upload some of my own images for further testing. As the training set includes the proper classification, this will be a supervised machine learning problem.

Looking through each sign, it seems that a few letters have a similar sign. These letters are 'm' and 'n', 'g' and 'h','i' and 'j', and 's' and 't'. While the rest differ very slightly by only one finger, 'i' and 'j' are the exact same sign, with 'j' moving. Considering how the model will only be looking at still images, this could be problematic, especially with the 'i' and 'j'. I would expect some misclassifications between these letters. Furthermore, the 'j' and 'z' are motion signs, and could also provide some problems, as the model will only be looking at still images. Luckily, the 'z' sign is fairly distinct as none of the other letters have a similar sign, so the model would still be able to classify it accurately.

I will approach this problem with a deep learning framework. I plan to use heavily use the keras package in python to classify the images, and my deliverables will include a model to classify images of the ASL alphabet.

To obtain the pictures and their label from the file, I first had to get the label names from the folders. To do so, I used os.listdir, which allowed me to get the folder names as a list from the specified path. These folder names double as the classification label. With these labels, I was able to plot my initial images where I had one image of each label.

After the initial plots, I proceeded to get my training dataset by saving all 3000 images for each class as a 64 x 64 x 3 vector. The 64 x 64 x 3 vector means that each image is saved as 64 by 64 pixels with each pixel having 3 channels for red, blue, and green, with each pixel ranging from 0 to 255. I used the functions imread and resize from the cv2 package to do so. As this vector represents only one image, I end up with a training dataset with the shape of 87000 (representing all 87000 images), 64, 64, 3. Afterwards, I normalized the data by dividing this list by 255, so each pixel has a value between 0 and 1. I saved the labels as a separate list with a shape of 29000. I was able to get my testing dataset (which comprised of 28 images for 28 of the 29 classes) in a similar fashion. This testing dataset is extremely small, but I will use it as it was included in the kaggle data set, and only as the final testing set.

Afterwards, I proceeded by converting the labels of both the training and the final testing set into numbered labels, as before it was labeled by its class as a string. To do so, I first created a dictionary matching each class to its label, and then I used that dictionary to convert the classes into its numbered label. Afterwards, I used keras' to_categorical function to encode the labels. The results were a 29000x29 array for the training labels and a 28x29 array for the testing labels.

As the training data and labels are in order of the folders, shuffling will be required for the model to train properly. Using sklearn's shuffle function, I randomly shuffled the arrays for the training set and the encoded training labels. Then, I used sklearn's train_test_split to split the training data into a training and testing set, with the test set being 30% of the data. This testing set will be used to judge the model's accuracy along with the final testing set, and I ende up with a training set of 60900 images and a testing set of 26100 images

I then proceeded with some data augmentation techniques to not only increase the dataset, but also to generalize the model better for real life situations. The ASL can be signed with both the right hand or the left hand and, despite the fact that a right handed sign mirrors (or flips horizontally) the left handed sign, they represent the same letter. The result is that different images would have the same meaning. Therefore, my first data augmentation task was to create duplicates of all 60900 training images and flip them. I did not augment the testing images as those images represent the images I want my model to predict, and I do not want these images to be changed.

To do so, I created my own function that uses tensorflow's flip_left_right function, runs its own tensorflow session to use the function and then save the images in a tensorflow object. I use np.array at the end of the function to convert this object back into an array.

Afterwards, I use the same steps to create another set of duplicates, but this time with a randomly adjusted hue, contrast, and brightness. These changes can represent different lightings, and will help the model generalize in different lighting situations.

I end up with 3 arrays with 60900 images. The first is the unchanged images, the second is the flipped images, and the third is the color-adjusted images. I proceed by using numpy's

concatenate function to put all these arrays together. The result is a 182700x64x64x3 array for the training images and a 182700x29 array for the training labels. With these arrays, along with the two testing sets, I can begin training my model.