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 will include part of the training set as the test set. 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 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 1500 images for each class as a 64 x 64 x 3 vector. I opted to not use all 3000 images as it took too much CPU and would always cause my kernel to restart. 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 (43500 (representing 43500 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 (43500,) . 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, so I use part of the training set as the test set and add it to this 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 43500x29 array for the training labels and a 28x29 array for the testing labels.

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. This means that different images would have the same meaning. Therefore, my first data augmentation task was to create duplicates of all 43500 training images and flip them.

To do so, I created a 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 43500 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 130500x64x64x3 array for the training images and a 130500x29 array for the training labels.

To proceed, shuffling will be required for the model to train properly, ss the training data and labels are in order of the folders. 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.

Finally, I concatenate the split test set with the 28 images from the given test set. The resulting 4 arrays are as follows: a training image set of shape (91350, 64, 64, 3), a training label set of shape (91350, 29), a testing image set of shape (39178, 64, 64, 3) and a testing label set of shape (39178,29).

With these arrays, I am able to proceed with building my model. I chose to use a Convolutional Neural Network architecture, which are especially helpful with image recognition. The sequential model I built is made up 4 major layers: a 2D convolution layers (conv2d in keras) and 3 dense (or fully connected) layers, the first two with an output of 100 and the last one with an output of 29 (which represents the total number of classes). I needed a flatten layer between the convolution layer and the first dense layer to bridge the two layers and allow the convolution layer's output to work with the dense layer's input. I also included a dropout layer between the first and second dense layers to try to reduce overfitting.

My conv2d layer of 64 with a kernel_size of 3 means that this layer will have 64 receptive fields of 3x3 pixels. Each field goes through element wise multiplications and results in a single number to represent that particular field. This convolutional layer is particularly useful for images as it helps the model identify features of the image, such as straight edges or curves.

The first two dense layers perform linear operations that connect every input to every output. As they come after the conv2d layer, they look at the image features found in the conv2d layer and find their correlations to different classes. These two dense layers and the conv2d layer has a relu activation function, meaning that if the output is negative, it becomes 0. Otherwise, the output remains the same.

The dropout layer of .4 between the first two dense layers means that 40% of the input units are randomly dropped, or changed to 0, which would help prevent overfitting.

The final dense layer gives an output of 29, with each output corresponding to one of the 29 classes. Its softmax activation means that it gives the probabilities of what class the image belongs to.

With the model architecture built, I proceed by compiling it with an adam optimizer and a categorical cross-entropy loss function. An adam (adaptive moment estimation) optimizer, similarly to stochastic gradient descent, iteratively updates a set of parameters to minimize a loss function. Unlike stochastic gradient descent, however, the adam optimizer computes different learning rates for different parameters from the estimates of the first and second moments (or the mean and the uncentered variance) of the gradients.

Cross-entropy loss (or log loss) assumes 2 classes and calculates a loss score based on the truth and what the model predicts. Confident and incorrect predicted probabilities result in a high log loss score. Therefore, minimizing this log loss means that prediction accuracy will go up. Categorical cross-entropy loss is similar to cross-entropy loss, but not limited to only two classes, which is why I used this loss for the model.

After compiling the model, I train the model with my training set. I included an EarlyStopping patience of 3 meaning that, if after 3 epochs, the accuracy does not improve, then the model will stop training.

I end up with an accuracy of 86.10% and a validation accuracy of 92.05%. The testing accuracy 91.91% signifies that my model is not prone to overfitting and that it seems to be proficient at classifying the ASL signs.

My next step is to plot a confusion matrix to see which classes the model predicts the best and which ones the model struggles more with. To do so, I first revert the one-hot labels of the training set, so each label is a number from 1 to 29. Then, I use the predict method with my model to get the predicted probability distribution of the classes for xTrain. I then used numpy's argmax to find the class with the highest probability, which is the class that the model would predict the image to be. I inputted these two lists into scikit-learn's confusion_matrix function to get confusion matrix and, using a function from scikit-learn's website, plotted a visually pleasing confusion matrix.

Looking at the confusion matrix, I notice that that images classified as 'nothing' have the highest percentage of being correctly labelled. This makes sense as the 'nothing' label is the most distinct as it is the only class that doesn't have a hand in the picture. Also of note is that the 'V' label is also commonly misclassified as the 'W' label. These two signs are also very similar, so this common misclassification is not a surprise. Finally, the 'X' label seems to be the most misclassified.

Afterwards, I look at the changes in accuracy and loss for both the training and validation set. I was able to obtain all 4 in a list using keras' history method and subsequently plot the training and validation accuracy alongside the training and validation loss. A decrease in loss leads to the same increase in accuracy, so I will only mention the accuracy plot. It seems that the training accuracy has a much smoother increase compared to the validation accuracy, but it is also consistently lower than the validation accuracy.

Finally, I look at example images that were correctly and incorrectly classified. To do so, I first make an array for the correctly classified images and an array for the incorrectly classified

images. To make sure the title will show the class name rather than its numbered label, I made two new lists of class labels for the predicted images and training images. Finally, I plotted 9 random images from both the correctly and incorrectly labelled arrays titling each image with its predicted class and its actual class.