


Finger Spelling Recognition Via Knuckle Placement Using Grid Search of Ensemble ML Techniques and Autocorrect

University of Michigan
William Dodson


Isaac Taylor

EECS 442: Intro to Computer Vision

wlldodsn@umich.edu


isaactay@umich.edu

1. Introduction

Sign languages make up the primary language tens of millions of deaf people. 1 in 8 people in the U.S. above the age of 11 have hearing loss in both of their ears. This is a significant part of the population and it is important that the world and the future are designed in a manner accessible to them and respectful to their culture. Sign language is not just a way to communicate, but an integral part of deaf culture and a force for inclusion and relationship in society. Sign language does not translate to spoken language in a simple way, so technology can be extremely helpful in helping someone learn sign language. Additionally studies have show that deaf children can often learn to understand and sign finger-spelled words before they learn to read or write. This means that there is significant demand for ML models that can convert signs to written language and vice versa. [6]

Our problem question: "Is it possible to enable deaf people to communicate with those who do not know sign language via technology?"

In this project, our team worked on creating an ASL Finger Spelling Recognition program using various Machine Learning techniques that we have learned over the semester. We looked to create an accessible, low-cost, and easy to use, program and model that allows sign-language users to transcribe their language for easier and more efficient communication between themselves and those who may not know sign language. Additionally, we added a multitude of features and options to allow a user to effectively apply our project to their specific situation.

We created our own dataset for this project, using a python script to take over 30,000 pictures of our own hands. We processed the dataset with a library that recognizes the knuckles of human hands and their orientation to one another. We then used these features to train random forest models, support vector machines, and neural networks,

utilizing a random grid search to select hyper parameters. Those models allow users to then put both of their hands in a live camera feed and see the letters that they sign spelled out in front of them along with an autocorrected version. Additionally, we attempted to create a generative model to generate ASL signs from text, enabling the two-way translation described earlier.

This project is an ensemble of various components and features. The ideas for some of the features came from researching ASL detectors and hand detection online, brainstorming together as a group, and one of the team member's independent ML research project.

2. Background

Current research in hand recognition revolves heavily around motion sensors, radio wave input, ultrasonic imaging, or simple camera input. [2] Each of these methods have their benefits and drawbacks. Motion sensors are commonly found computing sensors that are often placed on gloves in specific locations to measure triaxiality of hand and finger placement rendering data that once trained and passed through a CNN can classify hand placement, they often can be costly for the average individual and can require inaccessible technology. Both radio wave input and ultrasonic imaging use a combination of sensors and receivers to measure infrared or sound feedback from the hand, these unfortunately can also require costly equipment and may not be accessible to the majority of the populace. Vision-based recognition tends to be the most accessible of other methods of hand recognition solely requiring a camera and trained model for recognition.

Current datasets available for hand recognition research are relatively limited, specifically sign language datasets, which poses a significant challenge for training robust models. While there are some publicly available datasets, such as the ChicagoFS Wild+ Dataset [1], they often lack diver-

sity in terms of hand shapes, skin tones, organization, and gestures. This limitation hampers the generality of models trained on such datasets, especially when deployed in real-world scenarios where the variability of hand appearances is vast.

Additionally, privacy concerns surrounding the collection and usage of hand-related data further restrict the availability of comprehensive datasets for research purposes. As a result, researchers often resort to collecting their own data, which can be time-consuming and resource-intensive. Addressing the scarcity of high-quality, diverse datasets is crucial for advancing the field of hand recognition and enabling the development of more inclusive and accurate systems.

One difficult decision associated with detection models is the choice of a model architecture. For this project, we used random forests, support vector machines, and neural networks. In a random forest, many classification decision trees are created for a dataset, with a random subset of the features being used for the decision at each node of each tree and decisions being made based on impurity. Each tree in the random forest votes for the final overall classification of test data. The basic idea of a support vector machine is that it performs an operation that casts data into a higher dimensional space that is easier to split up and classify. Finally, a neural network is a machine learning model that uses artificial neurons to simulate the functionality of the human brain. Each neuron receives inputs from other neurons and has a different weight for each input. After weighting and summing its inputs, the neuron adds a constant scalar bias term and passes the result through some nonlinear activation function before sending the output to other neurons.

For full understanding of our work, it is useful for a reader to have a basic understanding of Python, machine learning, and signs used for finger spelling in American Sign Language.

3. Methodology

There are several stages in the technical flow of our project, as can be seen in Figure 1. First, data is collected with a python script that takes rapid pictures of a user’s webcam. Second, features pertaining to the locations of the knuckles on a hand are extracted using the `mp.solutions.hands` library. Next, the user is given the option to train either a random forest, a support vector machine, or a neural network on those features. After saving the model, a user can run our `inference_classifier` script to make signs on a video feed and see them classified in real time. As the signs are detected, the letters are displayed on the screen. Finally, our auto-correct feature displays an auto-corrected version of what the user is probably trying to communicate. Our following implementation was inspired by an existing project by `computervisioneng` published on Github [4]. The remainder of the section will

discuss how we implemented our technical flow to optimize for new features, training methods, and overall applicability. The code for our project can be found in Appendix A, additionally our Github repo [3].

3.1. Dataset

We generated our dataset by taking pictures of ourselves finger spelling the characters of the alphabet through our computer’s webcam. We accomplished this using OpenCV’s `cv2.VideoCapture` library and wrote a script to take any given pictures of us signing each character to capture how to spell a given character from different angles and distances. We noticed that the reference code to capture the images lacked any detail or direction in the process of capturing the multiple images. For instance, the original code captured the everything being seen by the video camera and did not tell the user which character in the alphabet it was capturing images for. We improved this process in two key ways. First, we changed the code to only capture what the webcam saw within a black box drawn on the screen. This ensured that the frame was restricted to just the hand signs, minimizing background noise and irrelevant features. Second, we printed on the screen which character the code was capturing images for at a given time. Additionally, we wanted to add support for detecting letters finger-spelled by both the right and left hand so we adapted the code to collect images of letters spelled by both hands. The code employed a timer-based mechanism to capture multiple images for each ASL sign, ensuring coverage of all letters with both right and left hands. The script was fine-tuned to maximize image clarity and maintain consistent lighting and depth, facilitating better feature extraction. Originally, we tried to use the ChicagoFS Wild+ Dataset, but this dataset does not contain images of single signs with single-letter ground truth labels. Rather, it contains bursts of images with several consecutive detected letters as a label. This presented a challenge: how would we separate these frames into individual letters? We tried two strategies, which are depicted in figure 3. In the first strategy, we split up each set of frames evenly according to how many letters were in the ground truth label, We then classified each frame according to its order. In the second strategy, we tried only taking the middle frame from each portion and discarding the rest. Unfortunately, neither of these strategies allowed us to classify individual frames with a reasonable level of certainty, so we opted to create our own dataset. Our home-made dataset contains over 30,000 images. To address the issue of diversity, we captured images of hands with various skin tones, and from different individuals, to ensure the model’s accuracy across different users. By training the model with this dataset, we aim to achieve a higher degree of generalization for real-world application.

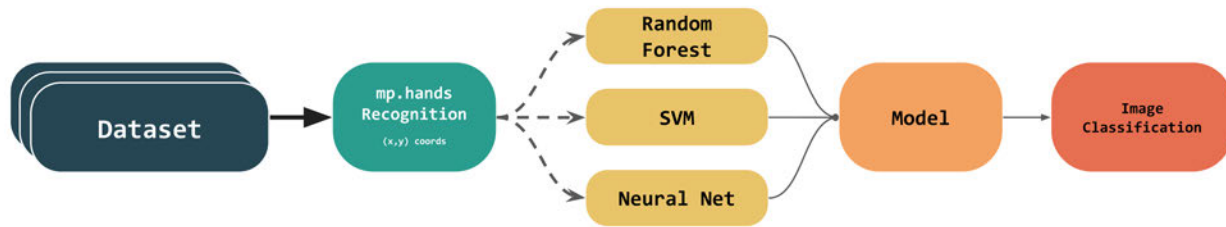


Figure 1. Overall technical flow of the project

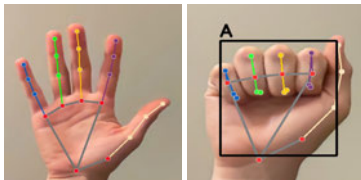


Figure 2. Hand identification through hands.mp: identifying knuckles and key points from hand image and an example of identification provided through our project

3.2. Data Pre-processing

After creating our dataset, we needed a way to draw features from the pictures of hands. Many similar projects have opted to use convolutional neural networks for such a task, since they preserve spatial information. However, we wanted to have the option to use several different type of machine learning models for classification. We needed to determine numerical features from the hand images that could be processed independently from the total spatial information of the images. To do this, we made use of the `hands` library from `mediapipe.solutions`. The `hands` is able to extract 42 unique landmark points in 3d space from a hand image. We used the `hands` library to extract this landmark information from the hand images for each ASL hand sign to use in our training stage. Figure 2 represents how `hands` detects the specific features for the hand images. The initial code was designed to only extract features for a single hand, so we had to adapt our code to be able to process both the right and left hand data, and store this information in pickle files.

3.3. Classifier Training

As mentioned previously, our training code allows the user the option to specify which type of machine learning model they would like to use. This is done via command line arguments. The choice of model could depend

on many factors, including the amount of data that the user has or how much they are worried about over-fitting to the training data. The available options for machine learning model types in our code are random forest, support vector machine, and neural network. For all of our model types, we begin by loading the data, separating the ground truth labels from the features, and arbitrarily splitting the data into training and testing data using `train_test_split()` from `sklearn`. Parameters for each model were chosen with a sort of random grid search, which will be explained more in another subsection below. After the model has been trained, it's accuracy on the test data is printed to the user.

3.3.1 Random Forest

Random forest is the default model type that we used for our project. It can also be specified with the command line option `-m random_forest`. If the random forest option is specified, we simply instantiate the model as `RandomForestClassifier()` from `sklearn`, fit the model to the training set, and predict the letter labels for the test set.

3.3.2 Support Vector Machine (SVM)

SVM can be selected as the model type in our project with the command line option `-m svm`. If SVM is chosen, the model is instantiated as `SVC()` from `sklearn`, it is fit to the training set, and labels are predicted for the test set.

3.3.3 Neural Network

A neural network can be selected as the model type in our project with the command line option `-m neural_net`. Since we did not use `sklearn` for this functionality, the process for this model type is a bit different from the others. If the neural net option is specified, a model is created using `tf.keras.models.Sequential`. The model has

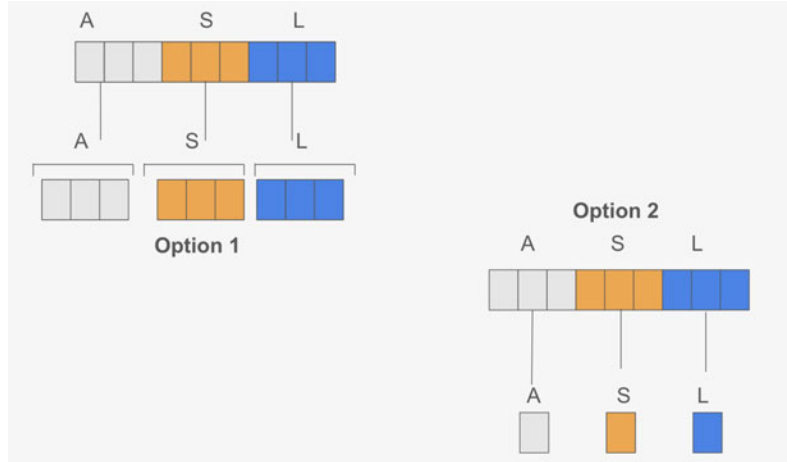


Figure 3. Initial strategies for using ChicagoFS Wild+

three fully connected layers with two dropout layers in the middle. The first layer has an input shape that matches the shape of the training data and an output shape of 128. The second layer has an output shape of 64, and the final layer has an output shape of 26 for the 26 letters in the alphabet. The output vector is decoded into a class prediction using `to_categorical` from `keras`. The model uses a categorical cross-entropy loss function. Before being trained, this `keras` neural network must be compiled with its classifier and loss function. Then it is trained and evaluated just like the other two model types.

3.3.4 Hyper Parameter Selection

For the support vector machines and the neural networks, there were many hyper parameters that needed to be chosen, and it was difficult to know which parameters would be best for performance as well as a specific user's application. Therefore, we opted to use a sort of random grid search for these parameters. Some potential parameter choices were hardcoded into two separate python dictionaries for SVM and neural net. For the SVM, possible values of 'C' (inversely correlated with strength of regularization) were 0.001, 0.01, 0.1, 1, 10, and 100, possible values of gamma were 0.01 and 0.1, and possible kernel types were radial basis function, linear, polynomial, and sigmoid function. For the neural network, possible optimizers were Adam, stochastic gradient descent, adaptive gradient algorithm, Adamax, and Nadam. Possible batch sizes were 8, 16, 32, 64, 128, 256, and 512, and possible numbers of epochs were 5, 10, 15, 20, 30, 40, 50, 75, and 100. For each model, these parameters were chosen using `random.choice`, and we were able to compare the accuracy of models with different hyper parameters.

3.4. Detection

After training our models, we were able to detect ASL finger spelling in real time. Similar to the dataset generation we employed `OpenCV's cv2.VideoCapture` library to retrieve video cam video in real time. We constantly run a while loop and pass in the current frame into the `mediapipe.solutions.hands` library to extract relevant features from a hand detected through web cam. These features are then passed into the trained model to predict which letter of the alphabet is currently being spelled. Earlier in the project, our code had some significant limitations. First, it was only able to predict what a single hand was signing and was restricted to predicting only a right or only left hand. In fact the code would crash when an additional hand entered the frame. We improved upon this by adding support for multiple hand detection and prediction for which letter each hand was spelling. Second, the original code was designed to predict the finger spelling of just a right hand or just a left hand, depending on which hand was used to generate the finger spelling dataset. Since our improvements in the dataset generation allowed us to collect finger spelling for both hands, we could train right and left hand specific models (although we also trained single models on all of the data), and more accurately detect the finger spelling of both right and left hands. The `mediapipe.solutions.hands` library had great support in being able to determine whether a detected hand was a right or left hand, so after determining whether a hand was right or left, it was straight forward to call the right or left hand model for prediction. As letters are classified, our code prints them to the screen. One challenge that we faced in this portion of the project was the fact that the neural network and the other two frameworks have different output formats. We needed to add code to differentiate between when the output is simply an index and the case when it is

an encoded vector of size 26.

3.5. Auto-correction

Considering that we bootstrapped our own dataset, none of us knew ASL prior to this project, and we programmed this project in a week or two, we knew that our models being nearly perfect was highly unrealistic. That being said, we were still committed to the vision of this project and wanted to make sure that it could actually be used to translate sign language effectively. Our solution for the sometimes unreliable nature of our models was to add an "auto-correct" filter to our final output. The auto-correct is implemented very simply using the Python `autocorrect` library. It corrects the raw predictions from the model to actual English words. Additionally, we decided to implement a certain signing time-based protocol in our live classifier to ensure more reliable communication. A letter is only officially classified and printed if it is detected for three seconds straight. A space is printed if no letter is detected for ten seconds or more.

3.6. Generative Model

To try to tackle the challenge of two way communication between an adult who doesn't know any sign language and a deaf child who knows signs but not written language, we also tried to implement a generative model to generate signs from our dataset. More detail on this model is presented in the conclusion of this report.

3.7. Clarifications and Credit

For the code that operates our cameras to help us collect images and the code that uses `hands` to extract features, we drew heavily from a tutorial found on youtube in [4], which contributed high-level ideas and concepts, such as using pickle files for dataset condensing and `cv2` for video input. Although we made many tweaks for our purposes, much of the code for creating the dataset is not directly written by us. However, **the code for the training of the machine learning models and the code for classifying based on that model, which are the basis of our project, are entirely our own or heavily altered for our purposes.** We have have drawn some inspiration from various sources, but the code was written by us. The generative model was constructed based on the code from Homework 5 as well some code inspired by Open AI's Chat GPT 3.5. All of the code can be found in the appendices.

4. Results

The most concrete quantitative metric that we have for the success of this project is the accuracy of the models that we trained on the testing portion of the dataset. However, these results are meaningless if the live classification feature performs poorly. Unfortunately, the live classification

feature does not produce a numerical accuracy calculation. Therefore, we will discuss the results of the models on both static test data as well as the live feed.

4.1. Initial Model Accuracy

As explained in the methodology section, the parameters for the SVM and neural net were tuned randomly. At a basic level, parameter choice will drastically affect the performance of a model, so we will present a table for each model type with the results of some of the best parameter combinations shown.

4.1.1 Random forest results

Params	Best Accuracy	Notes
None	99.02%	See below

When using a smaller data set with a few thousand images, our accuracy was closer to 97%. As you can see, with a larger, more varied dataset, we were able to achieve over 99%. Random forest was our default model type for good reason. It consistently achieved results like this and seemed to be less overfit to the dataset than the other model types.

4.1.2 SVM results

Params	Best Accuracy	Notes
C		
0.001	37.99%	gamma 0.1, linear kernel
0.01	74.17%	gamma 0.01, linear kernel
0.1	88.44	gamma 0.01, linear kernel
1	96.03%	gamma 0.1, linear kernel
10	97.62%	gamma 0.1, linear kernel
100	98.93%	gamma 0.1, rbf kernel
gamma		
0.01	98.88%	C=100, linear kernel
0.1	98.93%	C=100, rbf kernel
kernel		
rbf	98.93%	C=100, gamma=0.1
linear	98.88%	C=100, gamma=0.01
polynomial	95.87%	C=100, gamma=0.1
sigmoid	96.08%	C=100, gamma=0.1

Unsurprisingly, models with larger values of C (and thus less regularization) had higher accuracy in the training stage. This is not a very meaningful finding, since these models most likely overfit to the dataset and are less likely to perform well for live classification. However, it was interesting to see that while our gamma value had an almost negligible effect on our accuracy, a radial basis function or linear function would clearly be the best kernel for our task. Our highest-performing SVM model had an accuracy

of 98.93%, a bit lower than that of the random forest model. A more complete documentation of our SVM random grid search results can be found in Appendix B.

4.1.3 Neural Network Results

Params	Best Accuracy	Notes
optimizer		
Adam	99.29%	batch 64, 75 epochs
SGD	94.45%	batch 32, 50 epochs
adaptive gradient	27.01%	batch 16, 5 epochs
Adamax	98.77%	batch 32, 75 epochs
Nadam	99.24%	batch 32, 75 epochs
batch size		
8	99.02%	nadam, 30 epochs
16	99.11%	nadam, 40 epochs
32	99.24%	nadam, 75 epochs
64	99.29%	adam, 75 epochs
128	96.6%	adamax, 15 epochs
256	98.51%	adam, 50 epochs
512	96.59%	adamax, 75 epochs
epochs		
5	95.84%	adam, batch 32
10	95.19%	adamax, batch 16
15	98.64%	nadam, batch 8
20	98.71%	nadam, batch 16
30	99.02%	nadam, batch 8
40	99.16%	adam, batch 32
50	98.51%	adam, batch 256
75	99.29%	adam, batch 64
100	98.11%	adamax, batch 256

As you can see, we achieved similar levels of accuracy with all of the optimizers except for the adaptive gradient optimizer. We believe this to be due to the fact that adagrad is designed to perform well on sparse datasets. Additionally, we found the best batch size to be around 32 or 64. Finally and unsurprisingly, we found that more epochs leads to more accuracy in the model training stage. Overall, our best accuracy was 99.29%. This is higher than that of both random forest and SVM.

4.1.4 Initial Model Summary

Although our highest accuracy in training was achieved with a neural network, the differences between that figure and those of the random forest and SVM were essentially negligible. On top of this, when we began to experiment with these models on live feed classification, we found the random forest model to be more robust. We think that perhaps the neural networks and support vector machines were too complex and therefore were able to fit the training data

too strongly. More important than quantitative results is the ability to actually use the software to communicate, the mission of the project. Therefore, we used our random forest model for the following section.

4.2. Live Feed Accuracy

Live feed accuracy is a difficult thing to quantitatively measure in this environment due to the vast differing hand placements, hand sizes and shapes, as well as dialect differences in sign language. So without a uniform way to measure accuracy or an ASL signer to offer input, we opted to measure the accuracy of our final project using a common computer science testing phrase.

In practice, we recognize the shortcomings of this method of accuracy testing but also acknowledge it as an opportunity to present the visual results of our project.

In the created video [8] we finger spelled the phrase "Hello World" as a method of testing and deliverable output. Observing the hand classification viewers can see that most letters are often accurately and quickly predicted while a small amount of letters cause the classifier to "flutter" and may require some tweaking of hand placement. With the addition of the auto-correction feature, we provide a way for users to spell faster with less worry of accuracy halting communication.

Additionally, we recorded a second demo video [5] where we finger spell the phrase "A quick brown fox jumps over the lazy dog", known as a "pan-gram" as it is a sentence that contains all the letters of the alphabet. This time, we sign the sentence both right and left hands to demonstrate our systems ability to recognize finger spelling from both hands and ability to translate all letters of the English language with the help of auto correct.

Similar to the first demo, most letters are quickly predicted while letters that are similar to others cause the classifier to "flutter" between two or more guesses. The system fluttered frequently with the letter sets ("X" and "F"), ("U", "K", "V" and "R"), and ("S" and "N"). The classifier performed well to translate nearly every letter in the demo, except the letter "V" in the word "OVER", but the auto correct functionality helped the system to recover from the mistake and correctly spell the word "OVER".

These demos show definitively that our project can be easily used to translate signs into written language and allow pairs or groups of people to communicate who otherwise could not.

4.3. Results Summary

We experimented with many different machine learning models on our specific dataset, and we found that all of the model types were capable of achieving high prediction accuracy. However, we found that the accuracy of a model in training and validation was not the best indication of its

performance for classifying on the live feed. We were able to find certain models that worked well quantitatively and certain models that worked well qualitatively and fulfilled the purpose of our project.

5. Conclusion

We conclude our project report by summarizing our work, presenting opportunities for further development, and drawing meaningful conclusions.

5.1. Summary

In this project, we explored the concept of sign language detection with computer vision techniques. We sought to answer the question: "Is it possible to enable deaf people to communicate with those who do not know sign language via technology?" We found that a solution for this problem was not only very feasible, but that a fairly robust solution can be created even with very limited time and resources. We created our own datasets of ASL finger spelling letters, extracted features from these pictures, trained various machine learning models on these features, and classified live videos of finger spelling. We enabled a user to effectively and reliably spell out properly spelled written words by finger spelling in front of a camera.

5.2. Improvements and Future Work

There were several places where our project left much to be desired due to time constraints, and these present many opportunities for future work.

5.2.1 Dataset

Although our dataset was comprehensive, we believe could be expanded to include more diverse hand shapes, angles, and environmental conditions to improve the model's accuracy and robustness. We also fell short in capturing the dynamic nature of certain letters like "J" and "Z," which involve motion. The current dataset comprises still images and is unable to encapsulate the motion trajectory that helps identify these letters. To address this, future work could extend the dataset to include video snippets or a series of images that represent the hand movements over time. We could perhaps integrate Recurrent Neural Networks (RNNs) to accurately capture and learn from dynamics of the hand motion. Also given our time constraints we were unable to test on a larger database. This is something we would want to be able to improve on in the future to improve the overall performance of our program.

5.2.2 Time and computing power for random grid search

The hyperparameter tuning process is crucial for optimizing machine learning models. Due to time constraints, we were limited in our ability to exhaustively explore the hyperparameter space. In the future given more access to compute power, we would definitely take advantage for a more robust random grid search.

5.2.3 Generative model

One of our main ambitions for this project was to create a generative model that would work towards facilitating two-way communication between individuals who are unfamiliar with ASL and those proficient. The goal was to be able to generate sign language from text input. We were inspired by our work from HW5 on Image Generative Models [7]. We created a letter dictionary for labelling and created a custom dataset class, `CustomDataset`, for loading and transforming image for batch processing. We then defined a deep learning model with a series of convolutional layers which formed our `Generator Class`. We then initiated our training loop which iterated over the images, added noise, and then generated the new images. However, we ran into dimensional errors likely originating from how we were applying our noise scaling. In the future, given more time we would love to implement this as a key feature of our program.

5.3. Takeaways

While this project may be feasible with limited resources and time, it is important to note that with any translating project, there comes much weight and responsibility to offer communication tools in the most accessible and meaningful way. Collectively we say we did that within the scope of our project.

We acknowledge that this isn't a new idea and there have been people who have previously created sign translators. Considering that, this project gave us a holistic learning opportunity, as we are hearing non-signers, to learn about this unique unspoken language and contribute to the advancement of accessible technology in this community.

6. Appendices

Additional documentation for project materials and code. Providing any code that we wrote for the project, and additional accuracy data.

6.1 Appendix A

In this subsection, we provide our code

`collect_imgs.py`: *inspired heavily by Felipe [4]*

```
1 import os
2 import time
3 import cv2
4
5
6 DATA_DIR = './data'
7 if not os.path.exists(DATA_DIR):
8     os.makedirs(DATA_DIR)
9
10 number_of_classes = 26
11 dataset_size = 1100
12
13 cap = cv2.VideoCapture(0)
14 for j in range(number_of_classes):
15     if not os.path.exists(os.path.join(DATA_DIR, str(j))):
16         os.makedirs(os.path.join(DATA_DIR, str(j)))
17
18     print('Collecting data for class {}'.format(j))
19     done = False
20     while True:
21         ret, frame = cap.read()
22         H, W, _ = frame.shape
23
24         minY, maxY = H // 8, H - H // 8
25         minX, maxX = W // 3, W - W // 3
26
27         cv2.rectangle(frame, (minX,minY), (maxX,maxY), (0, 0, 0), 4)
28         cv2.putText(frame, 'Ready? Press "Q" ! :)', (100, 50), cv2.FONT_HERSHEY_SIMPLEX, 1.3, (0, 255, 0),
29         3,
30                     cv2.LINE_AA)
31         cv2.imshow('frame', frame)
32         if cv2.waitKey(25) == ord('q'):
33             break
34
35     counter = 0
36     while counter < dataset_size:
37         ret, frame = cap.read()
38         H, W, _ = frame.shape
39
40         minY, maxY = H // 8, H - H // 8
41         minX, maxX = W // 3, W - W // 3
42
43         cv2.rectangle(frame, (minX,minY), (maxX,maxY), (0, 0, 0), 4)
44         cv2.imshow('frame', frame)
45         cv2.waitKey(25)
46         cv2.imwrite(os.path.join(DATA_DIR, str(j), '{}_3.jpg'.format(counter)), frame[minY:maxY, minX:maxX
47         ])
48         time.sleep(0.05)
49         counter += 1
50
51 cap.release()
52 cv2.destroyAllWindows()
```

`create_dataset.py`: *inspired heavily by Felipe [4]*

```
1 import os
2 import pickle
3
4 import mediapipe as mp
5 import cv2
6 import matplotlib.pyplot as plt
7
8
9 mp_hands = mp.solutions.hands
10 mp_drawing = mp.solutions.drawing_utils
11 mp_drawing_styles = mp.solutions.drawing_styles
12
13 hands = mp_hands.Hands(static_image_mode=True, min_detection_confidence=0.3)
```



```

14
15 DATA_DIR = './data'
16 i = 1
17 data = []
18 labels = []
19 for dir_ in os.listdir(DATA_DIR):
20     print(dir_)
21     # Sometimes the lines below need commented in/out try each just in case
22     if os.path.join(DATA_DIR, dir_) == "./data/.DS_Store":
23         os.remove("./data/.DS_Store")
24     for img_path in os.listdir(os.path.join(DATA_DIR, dir_)):
25         data_aux = []
26
27         x_ = []
28         y_ = []
29
30         img = cv2.imread(os.path.join(DATA_DIR, dir_, img_path))
31         # print(os.path.join(DATA_DIR, dir_, img_path))
32         # print(img_path)
33         if img_path == ".DS_Store":
34             os.remove(os.path.join(DATA_DIR, dir_, img_path))
35         img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
36
37         results = hands.process(img_rgb)
38         if results.multi_hand_landmarks:
39
40             for hand_landmarks in results.multi_hand_landmarks:
41                 # print((hand_landmarks.landmark))
42                 for i in range(len(hand_landmarks.landmark)):
43                     x = hand_landmarks.landmark[i].x
44                     y = hand_landmarks.landmark[i].y
45
46                     x_.append(x)
47                     y_.append(y)
48
49                 for i in range(len(hand_landmarks.landmark)):
50                     x = hand_landmarks.landmark[i].x
51                     y = hand_landmarks.landmark[i].y
52                     data_aux.append(x - min(x_))
53                     data_aux.append(y - min(y_))
54
55                 if len(data_aux) == 42:
56                     data.append(data_aux)
57                     labels.append(int(dir_))
58
59
60 f = open('data.pickle', 'wb')
61 pickle.dump({'data': data, 'labels': labels}, f)
62 f.close()
63
64

```

train_classifier.py: *Our code*

```

1 import pickle
2 import argparse
3 from sklearn.ensemble import RandomForestClassifier
4 from sklearn.neighbors import KNeighborsRegressor
5 from sklearn.svm import SVC
6 from sklearn.model_selection import ParameterSampler
7 from sklearn.model_selection import train_test_split
8 from sklearn.metrics import accuracy_score
9 import numpy as np
10 import random
11 from keras.utils import to_categorical
12 from tensorflow.keras.models import Sequential
13 from tensorflow.keras.layers import Dense, Input
14 from tensorflow.keras.initializers import HeNormal
15 import tensorflow as tf
16
17
18 svm_params = {
19     #'C': [0.001, 0.01, 0.1, 1, 10, 100],
20     'C': [0.1, 1, 10, 100],
21     'gamma': [0.01, 0.1],

```

```

22     'kernel': ['rbf', 'linear', 'poly', 'sigmoid'] # Kernel types
23 }
24
25 nn_params = {
26     'optimizer': ['adam', 'SGD', 'adagrad', 'adamax', 'nadam'],
27     'batch_size': [8, 16, 32, 64, 128, 256, 512],
28     'epochs': [5, 10, 15, 20, 30, 40, 50, 75, 100],
29     'loss' : ['categorical_crossentropy']
30 }
31
32 def random_parameters(parameters_grid):
33     selected_parameters = {}
34     for param, values in parameters_grid.items():
35         selected_parameters[param] = random.choice(values)
36     return selected_parameters
37
38
39 def parse_command_line_arguments():
40     parser = argparse.ArgumentParser(description = "Description for my parser")
41     parser.add_argument("-m", "--model", help = "Example: random_forest, svm, neural_net", required =
42     False, default = "random_forest")
43     #parser.add_argument("-a", "--autocorrect", help = "Example: on", required = False, default = "on")
44     argument = parser.parse_args()
45
46     return argument.model
47
48 def train_hand_model(model_type):
49     f = None
50     data_dict = pickle.load(open('./data.pickle', 'rb'))
51     data = np.asarray(data_dict['data'])
52     # Print lengths of data elements
53     for i, item in enumerate(data):
54         if (len(item)) != 42:
55             print("Length of element {} in data: {}".format(i, len(item)))
56     labels = np.asarray(data_dict['labels'])
57
58     x_train, x_test, y_train, y_test = train_test_split(data, labels, test_size=0.2, shuffle=True,
59     stratify=labels)
60     model = None
61     if model_type=="random_forest":
62         model = RandomForestClassifier()
63         print("Running random forest classifier")
64         model.fit(x_train, y_train)
65         y_predict = model.predict(x_test)
66         score = accuracy_score(y_predict, y_test)
67
68     elif model_type=="svm":
69         params = random_parameters(svm_params)
70         print("Running support vector machine with parameters: ", params, "\n...\n")
71
72         model = SVC(probability=True, **params)
73         model.fit(x_train, y_train)
74         y_predict = model.predict(x_test)
75
76         score = accuracy_score(y_predict, y_test)
77
78     elif model_type=="neural_net":
79         selected_parameters = random_parameters(nn_params)
80
81         sel_optimizer = selected_parameters['optimizer']
82         sel_batchsize = selected_parameters['batch_size']
83         sel_epochs = selected_parameters['epochs']
84         sel_loss = selected_parameters['loss']
85         model = tf.keras.models.Sequential([
86             tf.keras.layers.Dense(128, activation='relu', input_shape=(x_train.shape[1],)),
87             tf.keras.layers.Dropout(0.2),
88             tf.keras.layers.Dense(64, activation='relu'),
89             tf.keras.layers.Dropout(0.2),
90             tf.keras.layers.Dense(26, activation='softmax')
91         ])
92
93         print("Running neural network with parameters: ", selected_parameters, "\n...\n")
94         model.compile(optimizer=sel_optimizer, loss=sel_loss, metrics=['accuracy'])
95         # Training the model

```

```

96     y_train_encoded = to_categorical(y_train, num_classes=26)
97     y_test_encoded = to_categorical(y_test, num_classes=26)
98     model.fit(x_train, y_train_encoded, batch_size=batch_size, epochs=epochs, verbose=0)
99
100     scores = model.evaluate(x_test, y_test_encoded)
101     score = scores[1]
102
103
104
105     print('{}% of samples were classified correctly !'.format(score * 100))
106
107     f = open('model.p', 'wb')
108     pickle.dump({'model': model}, f)
109     f.close()
110
111 if __name__ == "__main__":
112     model_type = parse_command_line_arguments()
113     train_hand_model(model_type)
114
115
116
117

```

inference_classifier.py: *Our code*

```

1     import pickle
2     import time
3     import autocorrect
4
5     from textblob import TextBlob
6     import argparse
7
8     import cv2
9     import mediapipe as mp
10    import numpy as np
11
12    def parse_command_line_arguments():
13        parser = argparse.ArgumentParser(description = "Description for my parser")
14        parser.add_argument("-a", "--autocorrect", help = "Options: 1, 0", required = False, default = 1)
15        argument = parser.parse_args()
16        return int(argument.autocorrect)
17
18
19
20    def classify_user(auto_corr):
21
22        model_dict = pickle.load(open('./model.p', 'rb'))
23        model = model_dict['model']
24
25        cap = cv2.VideoCapture(0)
26
27        mp_hands = mp.solutions.hands
28        mp_drawing = mp.solutions.drawing_utils
29        mp_drawing_styles = mp.solutions.drawing_styles
30
31        hands = mp_hands.Hands(static_image_mode=True, min_detection_confidence=0.3)
32
33        labels_dict = {
34            0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F', 6: 'G', 7: 'H', 8: 'I',
35            9: 'J', 10: 'K', 11: 'L', 12: 'M', 13: 'N', 14: 'O', 15: 'P', 16: 'Q',
36            17: 'R', 18: 'S', 19: 'T', 20: 'U', 21: 'V', 22: 'W', 23: 'X', 24: 'Y', 25: 'Z'
37        }
38
39        last_change_time = time.time()
40        start_time = time.time()
41        current_char = None
42        word = ""
43        last_hand_time = time.time()
44        _, start_frame = cap.read()
45
46        while True:
47
48            # these should all be 2d arrays
49            data_aux = []
50            x_ = []

```

```

51     y_ = []
52
53     ret, frame = cap.read()
54
55     H, W, _ = frame.shape
56
57     frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
58
59     results = hands.process(frame_rgb)
60     if results.multi_hand_landmarks:
61         for hand_landmarks in results.multi_hand_landmarks:
62             mp_drawing.draw_landmarks(
63                 frame, hand_landmarks, mp_hands.HAND_CONNECTIONS,
64                 mp_drawing_styles.get_default_hand_landmarks_style(),
65                 mp_drawing_styles.get_default_hand_connections_style())
66
67         for hand_landmarks in results.multi_hand_landmarks:
68
69             sub_data_aux = []
70             sub_x = []
71             sub_y = []
72             for i in range(len(hand_landmarks.landmark)):
73                 x = hand_landmarks.landmark[i].x
74                 y = hand_landmarks.landmark[i].y
75                 sub_x.append(x)
76                 sub_y.append(y)
77
78             x_.append(sub_x)
79             y_.append(sub_y)
80
81             for i in range(len(hand_landmarks.landmark)):
82                 x = hand_landmarks.landmark[i].x
83                 y = hand_landmarks.landmark[i].y
84                 sub_data_aux.append(x - min(sub_x))
85                 sub_data_aux.append(y - min(sub_y))
86
87             data_aux.append(np.asarray(sub_data_aux))
88
89         for i in range(len(data_aux)):
90
91             x_coors = x_[i]
92             y_coors = y_[i]
93             data = data_aux[i]
94             x1 = int(min(x_coors) * W) - 10
95             y1 = int(min(y_coors) * H) - 10
96             x2 = int(max(x_coors) * W) - 10
97             y2 = int(max(y_coors) * H) - 10
98
99             data = np.asarray(data).reshape(1, -1)
100
101
102             prediction = model.predict(data)
103
104
105             if (type(prediction[0])!=np.int64):
106                 max_val = np.argmax(prediction[0])
107                 predicted_character = labels_dict[int(max_val)]
108
109             else:
110                 predicted_character = labels_dict[int(prediction[0])]
111
112
113
114
115             if predicted_character != current_char:
116                 current_char = predicted_character
117                 last_change_time = time.time() # Update last change time
118             else:
119                 if time.time() - last_change_time >= 3:
120                     word += current_char
121                     last_change_time = time.time()
122
123
124             # if word hasnt changed in 10 seconds add a space
125             cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 0, 0), 4)
126             cv2.putText(frame, predicted_character, (x1, y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 1.3, (0, 0,

```

```

0), 3,
127         cv2.LINE_AA)
128     else: # No hand detected
129         if time.time() - last_hand_time >= 10: # Check time since last hand detection
130             word += ' ' # Add a space
131             last_hand_time = time.time() # Update last time hand was detected
132
133
134
135     cv2.putText(frame, 'Output : ' + (word.replace(" ", "-")), (100, 100), cv2.FONT_HERSHEY_DUPLEX,
1.3, (100, 255, 100), 3,
136         cv2.LINE_AA)
137
138     cv2.putText(start_frame, "Press 'q' to quit", (int(W/2 - 250), int(H/2)), cv2.FONT_HERSHEY_SIMPLEX
, 1, (0, 0, 255), 2)
139     cv2.putText(start_frame, "Remove hands from view for 10 sec. to add a space", (int(W/2 - 250), int
(H/2)+50), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)
140     cv2.putText(start_frame, "Beginning in 10 seconds", (int(W/2 - 250), int(H/2)+100), cv2.
FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)
141
142
143     if time.time() - start_time > 10:
144         start_frame = frame
145
146     cv2.imshow('frame', start_frame)
147
148     if cv2.waitKey(1) & 0xFF == ord('q'):
149         break
150
151
152     cap.release()
153     cv2.destroyAllWindows()
154
155 if __name__ == "__main__":
156     auto_corr = parse_command_line_arguments()
157
158     classify_user(auto_corr)
159

```

genmodel.ipynb: Used some code from HW5 as well as AI generation

```

1 # -*- coding: utf-8 -*-
2 """genmodel.ipynb
3
4 Automatically generated by Colab.
5
6 Original file is located at
7     https://colab.research.google.com/drive/1_iEiXNTkVWKWWpFIfwk5kHwUNEeqgY5S4
8 """
9
10 from google.colab import drive
11 drive.mount('/content/drive')
12 GOOGLE_DRIVE_PATH_AFTER_MYDRIVE = 'data'
13
14 # Commented out IPython magic to ensure Python compatibility.
15 # %load_ext autoreload
16 # %autoreload 2
17
18 import os
19 import sys
20
21 GOOGLE_DRIVE_PATH = os.path.join('drive', 'MyDrive', GOOGLE_DRIVE_PATH_AFTER_MYDRIVE)
22 sys.path.append(GOOGLE_DRIVE_PATH)
23
24 print(GOOGLE_DRIVE_PATH)
25
26 # Commented out IPython magic to ensure Python compatibility.
27 # %cd /content/drive/MyDrive/data
28
29 letter_dict = {
30     'A': 0,
31     'a': 0,
32     'B': 1,
33     'b': 1,
34     'C': 2,

```



```

35     'c': 2,
36     'D': 3,
37     'd': 3,
38     'E': 4,
39     'e': 4,
40     'F': 5,
41     'f': 5,
42     'G': 6,
43     'g': 6,
44     'H': 7,
45     'h': 7,
46     'I': 8,
47     'i': 8,
48     'J': 9,
49     'j': 9,
50     'K': 10,
51     'k': 10,
52     'L': 11,
53     'l': 11,
54     'M': 12,
55     'm': 12,
56     'N': 13,
57     'n': 13,
58     'O': 14,
59     'o': 14,
60     'P': 15,
61     'p': 15,
62     'Q': 16,
63     'q': 16,
64     'R': 17,
65     'r': 17,
66     'S': 18,
67     's': 18,
68     'T': 19,
69     't': 19,
70     'U': 20,
71     'u': 20,
72     'V': 21,
73     'v': 21,
74     'W': 22,
75     'w': 22,
76     'X': 23,
77     'x': 23,
78     'Y': 24,
79     'y': 24,
80     'Z': 25,
81     'z': 25,
82 }
83
84 !pip install certifi>=2022.9.14
85 !pip install charset-normalizer>=2.1.1
86 !pip install contourpy>=1.0.5
87 !pip install cycler>=0.11.0
88 !pip install fonttools>=4.37.2
89 !pip install idna>=3.4
90 !pip install kiwisolver>=1.4.4
91 !pip install matplotlib>=3.6.0
92 !pip install numpy>=1.23.3
93 !pip install packaging>=21.3
94 !pip install Pillow>=9.2.0
95 !pip install pyparsing>=3.0.9
96 !pip install python-dateutil>=2.8.2
97 !pip install PyYAML>=6.0
98 !pip install requests>=2.28.1
99 !pip install scipy>=1.9.1
100 !pip install six>=1.16.0
101 !pip install tqdm>=4.64.1
102 !pip install typing-extensions>=4.3.0
103 !pip install urllib3>=1.26.12
104
105 !nvidia-smi
106
107 import os
108 import sys
109 from functools import partial
110 import os

```

```

111 import argparse
112 import yaml
113
114 import torch
115 import torchvision.transforms as transforms
116 import matplotlib.pyplot as plt
117 device = None
118 if torch.cuda.is_available():
119     print('gpu')
120     device = torch.device('cuda:0')
121 else:
122     print('cpu')
123     device = torch.device('cpu')
124
125 word_to_gen = input("What would you like to say? (one word at a time)")
126
127 """From ChatGPT:"""
128
129 import os
130
131 class CustomDataset(Dataset):
132     def __init__(self, data_folder, transform=None):
133         self.data_folder = data_folder
134         self.transform = transform
135         self.file_list = [f for f in os.listdir(data_folder) if f.endswith((''.jpg', '.jpeg', '.png'))]
136
137     def __len__(self):
138         return len(self.file_list)
139
140     def __getitem__(self, idx):
141         img_path = os.path.join(self.data_folder, self.file_list[idx])
142         img = Image.open(img_path)
143         if self.transform:
144             img = self.transform(img)
145         return img
146
147 import torch
148 import torch.nn as nn
149 import torch.optim as optim
150 from torch.utils.data import DataLoader
151 from torch.utils.data import Dataset
152 from torchvision import transforms
153 from PIL import Image
154 import numpy as np
155 import torch.nn.functional as F
156
157 class DDPMBlock(nn.Module):
158     def __init__(self, in_channels, out_channels, num_resblocks):
159         super(DDPMBlock, self).__init__()
160         self.resblocks = nn.ModuleList([
161             nn.Sequential(
162                 nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding=1),
163                 nn.ReLU(),
164                 nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1)
165             )
166             for _ in range(num_resblocks)
167         ])
168
169     def forward(self, x, noise, noise_scale):
170         for resblock, noise_scale_val in zip(self.resblocks, noise_scale):
171             noise_resized = F.interpolate(noise, size=x.size()[2:], mode='nearest')
172             print("Shape of x:", x.shape)
173             print("Shape of noise_resized:", noise_resized.shape)
174             print("Value of noise_scale:", noise_scale_val)
175             noise_scale_resized = noise_scale_val.view(-1, 1, noise_resized.size(2), noise_resized.size
176 (3))
177             print("Shape of noise_scale_resized:", noise_scale_resized.shape)
178             x += noise_scale_resized * noise_resized
179             x = resblock(x)
180         return x
181
182 class Generator(nn.Module):
183     def __init__(self, in_channels, out_channels, num_blocks):
184         super(Generator, self).__init__()

```

```

186     self.initial_conv = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding=1)
187     self.ddpm_blocks = nn.ModuleList([
188         DDPMBlock(out_channels, out_channels, num_resblocks=2)
189         for _ in range(num_blocks)
190     ])
191     self.final_conv = nn.Conv2d(out_channels, in_channels, kernel_size=3, stride=1, padding=1)
192
193     self.noise_scale = nn.Parameter(torch.tensor(0.01))
194     def forward(self, x, noise):
195         x = self.initial_conv(x)
196         print("After initial_conv shape:", x.shape)
197
198         for ddpm_block in self.ddpm_blocks:
199             x = ddpm_block(x, noise, self.noise_scale)
200             print("After ddpm_block shape:", x.shape)
201
202         x = self.final_conv(x)
203         print("Output shape:", x.shape)
204
205         return x
206
207 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
208
209 in_channels = 3
210 out_channels = 64
211 num_blocks = 4
212 num_resblocks = 4
213 num_epochs = 100
214 batch_size = 64
215 lr = 0.0002
216
217 for letter in word_to_gen:
218
219     dfold = '/content/drive/MyDrive/data/' + str(letter_dict[letter])
220
221     generator = Generator(in_channels, out_channels, num_blocks).to(device)
222
223     optimizer = optim.Adam(generator.parameters(), lr=lr)
224
225     transform = transforms.Compose([
226         transforms.Resize((64, 64)),
227         transforms.ToTensor()
228     ])
229
230     dataset = CustomDataset(data_folder=dfold, transform=transform)
231     dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
232
233     for epoch in range(num_epochs):
234         for i, real_images in enumerate(dataloader):
235             real_images = real_images.to(device)
236
237             noise = torch.randn_like(real_images).to(device)
238             print("Shape of real_images:", real_images.shape)
239             print("Shape of noise:", noise.shape)
240             fake_images = generator(real_images, noise)
241             print("Shape of real_images:", real_images.shape)
242             print("Shape of fake_images:", fake_images.shape)
243             loss = nn.MSELoss()(fake_images, real_images)
244
245             optimizer.zero_grad()
246             loss.backward()
247             optimizer.step()
248
249             if (i+1) % 100 == 0:
250                 print(f"Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{len(dataloader)}], Loss: {loss.item()
251                     :.4f}")
252
253 num_samples = 10
254 input_shape = (batch_size, in_channels, 64, 64)
255 with torch.no_grad():
256     noise = torch.randn(input_shape).to(device)
257     generated_images = generator(noise)
258

```

6.2 Appendix B

In this appendix, we provide the more exhaustive results of our random grid search.

SVM:

Accuracy	C	gamma	kernel
98.93	100	0.1	rbf
98.88	100	0.01	linear
98.66	100	0.1	linear
97.62	10	0.1	linear
97.55	10	0.1	rbf
97.47	10	0.1	rbf
98.76	100	0.01	linear
96.08	100	0.1	sigmoid
98.62	100	0.01	linear
96.03	1	0.1	linear
96.03	100	0.1	sigmoid
95.9	1	0.1	linear
97.59	10	0.01	linear
95.87	100	0.1	poly
95.39	10	0.1	sigmoid
92.26	1	0.1	rbf
92.1	1	0.1	rbf
88.9	1	0.1	sigmoid
82.79	10	0.1	poly
96.61	100	0.01	rbf
48.88	1	0.1	poly
38.83	0.01	0.1	sigmoid
91.27	10	0.01	rbf
88.44	0.1	0.01	linear
77.54	1	0.01	rbf
74.17	0.01	0.01	linear
37.99	0.001	0.1	linear
37.86	0.001	0.1	linear
73.87	1	0.01	sigmoid
37.09	0.001	0.1	rbf
73.07	1	0.01	sigmoid
41.1	0.1	0.01	rbf
36.79	0.001	0.1	sigmoid
14.46	0.1	0.1	poly
14.14	0.1	0.1	poly
40.59	0.1	0.01	rbf
15.37	100	0.01	poly
11.11	1	0.01	poly
10.29	0.01	0.1	poly
9.94	1	0.01	poly
9.72	0.01	0.1	poly
9.67	0.01	0.01	poly
9.32	0.001	0.1	poly
9.21	0.001	0.1	poly

Neural Network:

Accuracy	Optimizer	Batch Size	Epochs
99.29	adam	64	75
99.16	adam	32	40
99.24	nadam	32	75
98.64	nadam	8	15
99.11	nadam	16	40
98.77	adamax	32	75
99.02	nadam	8	30
98.99	nadam	8	40
98.71	nadam	16	20
93.01	nadam	128	5
98.51	adam	256	50
97.99	adamax	64	50
98.11	adamax	256	100
97.7	adam	64	15
98.76	adamax	8	75
97.62	nadam	64	15
96.24	adam	512	30
97.56	nadam	64	15
83.31	SGD	128	40
96.59	adamax	512	75
93.43	SGD	64	75
98.69	nadam	8	20
98.67	nadam	32	20
96.6	adamax	128	15
73.84	SGD	256	40
94.96	nadam	512	20
17.07	adagrad	128	5
95.84	adam	32	5
92.55	adamax	256	15
94.45	SGD	32	50
90.14	SGD	128	75
64.24	SGD	512	75
37.08	SGD	512	50
95.19	adamax	16	10
82.32	SGD	64	20
27.01	adagrad	16	5
22.37	SGD	256	15
9.42	adagrad	128	5
8.83	SGD	512	10
19.87	SGD	512	15

References

- [1] Chicago fingerspelling in the wild data sets. <https://home.ttic.edu/~klivescu/ChicagoFSWild.htm>. 1
- [2] Fahmid Al Farid, Noramiza Hashim, Junaidi Abdullah, Md Roman Bhuiyan, Wan Noor Shahida Mohd Isa, Jia Uddin, Mohammad Ahsanul Haque, and Mohd Nizam Husen. A structured and methodological review on vision-based hand gesture recognition system. *Journal of Imaging*, 8(6):153, May 2022. 1
- [3] William Dodson, Kushaal Marri, Aaryan Mukherjee, and Isaac Taylor. Github repo, Apr 2024. <https://github.com/aaryanmukherjeeumich/442FinalProject>. 2
- [4] Felipe. Sign language detection with python and scikit learn, 2023. <https://www.youtube.com/watch?v=MJCSjXepaAM>, <https://github.com/computervisioneng/sign-language-detector-python>. 2, 5, 8
- [5] Aaryan Mukherjee. Eecs 442 final proj demo 2, Apr 2024. <https://youtu.be/r8ckWfqQEhc>. 6
- [6] Carol Padden and Claire Ramsey. Reading ability in signing deaf children, 2019. 1
- [7] EECS442 Staff. Eecs 442 hw5, Apr 2024. <https://eecs442.github.io/hw5>. 7
- [8] Isaac Taylor. Eecs 442 final proj demo 1, Apr 2024. <https://youtu.be/x7zVmf5St7M>. 6