Successes and Difficulties in Generating and Implementing Convolutional Neural Network Models to Detect American Sign Language Numbers

Justin Caringal
caringaljustin.a@gmail.com
Truman State University
Kirksville, Missouri, USA

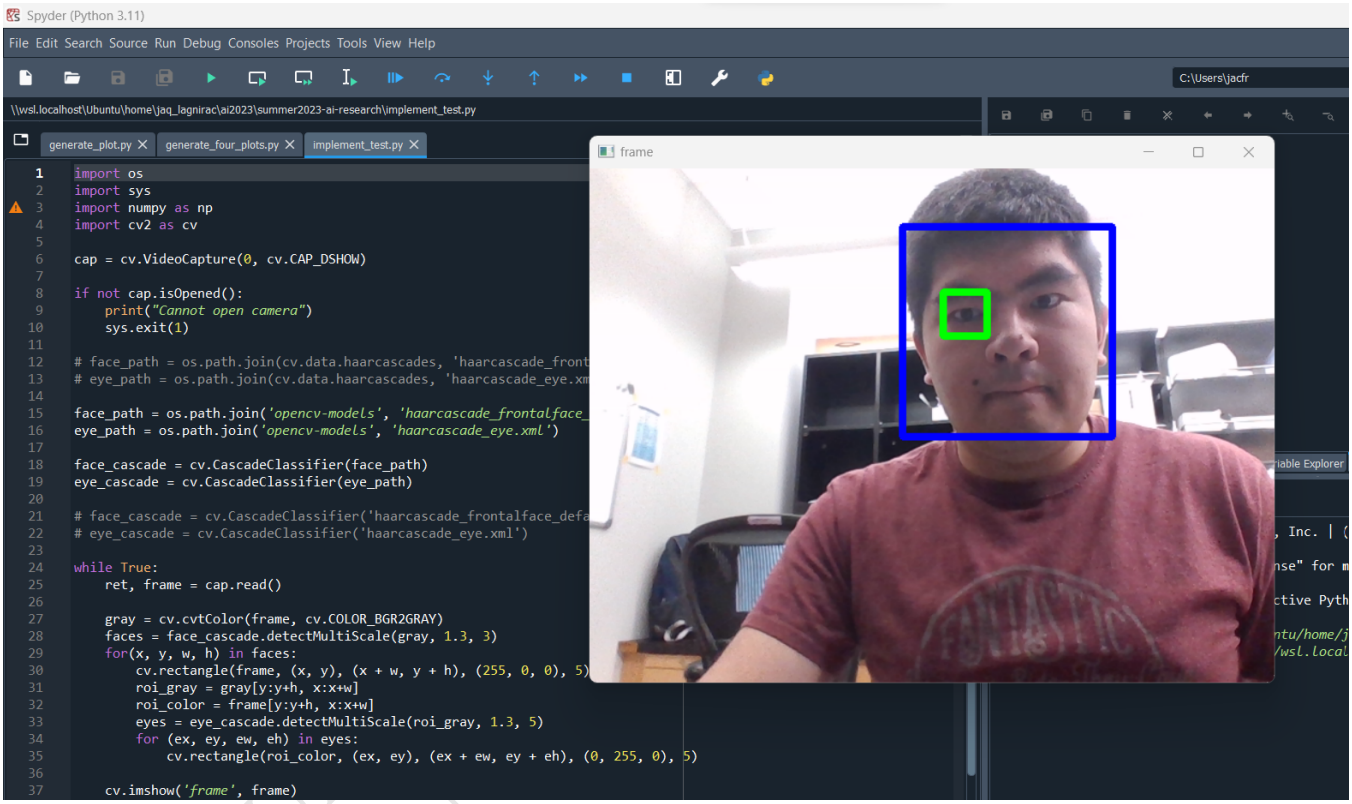

Figure 1: A test implementation of a Haar Cascade using OpenCV, executed through Spyder Anaconda on a Windows local environment, Jul. 20, 2023.

ABSTRACT

With the prevalence of artificial intelligence ever-growing, neural networks can be used to solve a variety of problems both large and small. One of these cases being the real-time translation of American Sign Language (ASL) into language that can be understood by a computer. This problem lacks much in-depth research and allows for the use of open-source packages, tools, and other resources to be utilized to their fullest extent. This paper discusses the research and generation of an ASL convolutional neural network model with a validation accuracy and loss of 96% and 21.2% and a testing accuracy and loss of 75.2% and 81.5%. While the ASL model is not currently able to be implemented due to difficulties presented, the project sheds a greater light on the development of contemporary large-scale models presently in the current popular zeitgeist and provides a launch-point for future projects and improvements; future directions are discussed.

Unpublished working draft. Not for distribution.

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

CCS CONCEPTS

• **Computing methodologies** → **Object identification**; **Machine learning**; **Cross-validation**; **Neural networks**.

KEYWORDS

datasets, neural networks, convolution, deep q-learning

ACM Reference Format:
Justin Caringal. 2018. Successes and Difficulties in Generating and Implementing Convolutional Neural Network Models to Detect American Sign Language Numbers. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

With the advent of tools such as ChatGPT and Dall-E becoming popular and common terms in the layperson's vocabulary, an age of research into artificial intelligence is upon the world. In order to capitalize on the increase in discussion surrounding AI, as well as the introduction, development, and release of primary and supplemental tools which aid in both the generation and implementation of artificially intelligent models, this paper showcases the development of a new model and tools to generate and implement a convolutional neural network model aimed at discerning between digits 0 through 9 in American Sign Language. This project hopes to get one step closer to a full implementation of a model, and does so somewhat successfully, although room for improvements is ever-present.

The model builds on previous research conducted by the author, while tackling a completely new problem in the space with new libraries and tools being used in the established workflow to improve the efficacy of the model as well as the actual research process. Not much research has been devoted to this specific problem, and this project aims to do so using tools, libraries, and software available to the everyday person. The groundwork of this project has been laid by Professor Donald Patterson of Westmont College[8] with minor modifications by Sam Myers, a student who has conducted similar research in the past.

Nothing revolutionary is coming out of this paper. The extraordinary thing about this project, however, is the cost of the program itself. Although a virtual machine was used in the later steps of the project to provide greater computing power (more on this in a later section), the rest of the project was conducted on a budget, with no new hardware or software being bought. The libraries and programs used were all publicly available, with no cost associated to the average consumer, with the only major investment being the time taken to understand the aforementioned resources. In doing this, the we hope to showcase that artificial intelligence, computer science, and technology is accessible to the masses.

2 RELATED WORK

As noted in the introduction, research in the field of computer vision has already been done. Pedestrian recognition was conducted by a team at the University of Pennsylvania School of Engineering and Applied Science[5]. A team at the Department of Electrical and Electronic Engineering at Imperial College London have also studied this problem and have created a solution for identifying pedestrians[6]. Finally, a paper published by Takahiro Sogawa et al. in the Public Library of Science on the topic of a different but adjacent focus (the paper worked on the identification myopic macular diseases) was also consulted on the efficacy of convolutional neural networks[3]. These works were not necessarily built off of, but rather looked to in order to prove that this line of research has a real-world basis, as well as wider applications beyond what was intended in the initial research process.

Figure 2: An image taken from the ASL number superset[4], used in training the model in this paper, via Kaggle (https://www.kaggle.com/datasets/lexset/synthetic-asl-numbers).

3 PROBLEM STATEMENT

The goal of the project undertaken in this research paper was to create programs to generate and implement a convolutional neural network model in Python to detect different ASL hand signs, as well as to create the necessary scripts and tools to aid in this main goal. As we needed a large dataset to increase the chances of success, multiple Kaggle datasets[7][9][4] were combined to create a superset of over 11,762 images of ASL numbers between 0 and 9. And as this problem dealt with the classification and identification of objects in images, the decision to use deep convolutional Q-learning was made. A convolutional neural network was created in a Python integrated development environment with the use of the Keras and TensorFlow libraries. Code was then developed, basing the initial framework off of a program from Professor Donald Patterson[8]. Modifications were made in regards to values in the variables, and model checkpoints and early stopping were introduced using Keras's callback libraries. The model generation architecture was also updated to reflect the newer Tensorflow v2 conventions that are slowly becoming the norm. We discuss our explanation of deep convolutional Q-learning in the next section.

Improvements were also mode in the quality-of-life and data-gathering processes. The API (application programming interface) for Discord (an online messaging service) was researched and implemented in order to provide updates to the researcher's phone, allowing the program to run in the background. The Python Time library was also implemented to provide timestamps and elapsed time functionality to the Discord bot. Other improvements and model implementation were researched but unfortunately were unable to be implemented in time for this paper's publication.

4 DEEP CONVOLUTIONAL Q-LEARNING

In this section, we provide an overview of the theory behind deep convolutional Q-learning as well as what exactly we used in order to solve the problem. Much of the research behind the project was gathered from Hadelin de Ponteves[2].

4.1 Q-Learning

Q-learning uses equations such as temporal difference and Bellman's equation to measure the "quality" of certain actions, then implements the said actions and observes the outcomes. The outcomes then get fed into the aforementioned equations, and the process repeats itself until the artificial intelligence program is able to discern the best outcome on new, never-before-seen datasets.

$$Q : (s \in S, a \in A) \mapsto Q(s, a) \in \mathbb{R}$$

The Q-value represents the aforementioned "quality" of a certain action during a certain state, and allows the program to choose what it deems the "best" option. This equation is feeds into the temporal difference and Bellman equations, allowing the artificial intelligence the freedom to learn.

$$TD_t(s_t, a_t) = R(s_t, a_t) + \gamma max(Q(s_{t+1}, a)) - Q(s_t, a_t)$$

Temporal difference serves as a "temporary intrinsic reward"[2] in the sense that it allows the program to seek out large Q-values during the beginning of the training. By searching and going towards these large Q-values, the program has a larger chance of succeeding during its current As the training goes on, the artificial intelligence will know all of the good and bad values already, and will diminish the impact that the Q-values and the temporal difference will have on the epochs.

$$Q_t(s_t, a_t) = Q_{t-1}(s_t, a_t) + \alpha TD_t(s_t, a_t)$$

The Bellman equation seen above is used to update the Q-values, allowing the program to increase the Q-values in locations where it finds high temporal differences from its previous iteration to its current iteration. In the equation, $\alpha \in \mathbb{R}$ is the learning rate of the program. The larger α is, the larger the Q-values will update and vice versa.

Using these three fundamental equations (Q-value, temporal difference, and Bellman's equation), an artificial intelligence program is able to learn and adapt to its surroundings, placing different weights on what it considered "good" by the researchers (higher reward) and avoiding what is considered "bad".

4.2 Deep Q-Learning

Deep Q-learning is the next step in the process. The major additions with this type of learning are artificial neural networks and experience replay memory.

The former emulates human neural networks (i.e. the brain) in the sense that they create "neurons" connected to different layers of other neurons. Information then gets inputted into the system, and after forward-propagating the input, the weights for each information are updated on each individual neuron. The process then repeats itself. Different neurons become activated at certain times, and this all depends on the type of activation function is used with that particular neuron.

$$\Phi(x) = 1 \text{ if } x \geq 0$$
$$\Phi(x) = 0 \text{ if } x < 0$$

The threshold activation function serves similarly to a binary switch or a Boolean function, turning on or off depending if its threshold is exceeded. It is a discontinuous signal that is often used as an output of an artificial neural network if the programmer wants a binary response.

$$\Phi(x) = {}^{1}/_{1+e^{-x}}$$

The sigmoid activation function looks like a soft curve, and is often used as an output to an artificial neural network that wants to return a value on a gradient. In other words, the network wants to return the probability that a dependent variable is 1. This function sends a continuous signal.

$$\Phi(x) = max(x, 0)$$

The rectifier activation function equals 0 when x equals 0, but begins increasing linearly when neuron is activated. This function is often used within the hidden layers of an artificial neural network, and is often known by the name "relu" (or Rectified Linear Unit). For our purposes, rectifier activation functions were used in the hidden layers while sigmoid functions were used as a part of the output layer.

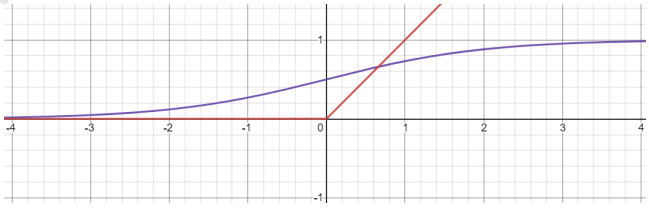

Figure 3: An example of the rectifier (or relu, in red) and sigmoid (in purple) activation functions, via Desmos (https://www.desmos.com/calculator).

Experience replay memory stores the results for an arbitrary amount of previous trials, allowing the neural network more information as it tweaks its weights. The neural network uses various forms of gradient descent to inch its way closer to the optimal results.

4.3 Deep Convolutional Q-Learning

The final step in coding the solution is to implement deep convolutional Q-learning, which adds a step before feeding the data into a neural network that converts the image into a multidimensional array. The array is then condensed further to allow the neural network faster processing time, then is flattened into a one-dimensional array to actually allow the neural network to read the information.

Four main steps exist in a convolutional neural network. The first step is convolution, where feature detectors are applied to an image in order to allow the program to "see". Two-dimensional arrays known as filters are applied to the image, counting the number of pixels which match the filter and inputting the number into a feature map. The filter is then slid along the input image and the process repeats until a full feature map is created. Multiple feature detectors are used to create multiple feature maps. These maps put together make up a convolution (or convolutional) layer.

The next step in the process is known as max pooling (or subsampling, depending on the source). Here, another filter is applied to the feature map that counts the highest value in a certain area. The program then records this value on a pooled feature map and jumps to the next area (allowing itself to run off the edge of the image), repeating this until the pooled feature map is created. This layer, known as the pooling layer, allows the program to lower the size for each map, thus abstracting the problem and keeping the neural network from receiving too many inputs. If the network received a full feature map, the computations would be too complex that the network wouldn't be able to learn properly. This step does not lose any important features, and makes the program partially resistant to translations and rotations in the inputted image.

The third step in the process is flattening. This step is fairly self-explanatory: all of the pooled maps are two-dimensional arrays, and this step converts and combines (or flattens) them into one-dimensional arrays. This is done because the artificial neural networks introduced in the last section only take one-dimensional arrays as inputs. The previous two steps were done in order to read an image into a two-dimensional array; this step bridges the gap between reading the array and inputting the information.

The last step, full connection, treads ground that this paper has already covered. Full connection takes the image, which has since been condensed and flattened into a one-dimensional array, and inputs it into a traditional artificial neural network. The exact specifications have been covered in greater depth elsewhere, as well as given a brief overview in the section above. These four steps, however, are what separate a normal artificial neural network created for deep Q-learning and make it part of deep convolutional Q-learning. The extra "convolution" part allows the program to take images as input data, rather than just raw numerical data like deep Q-learning can do.

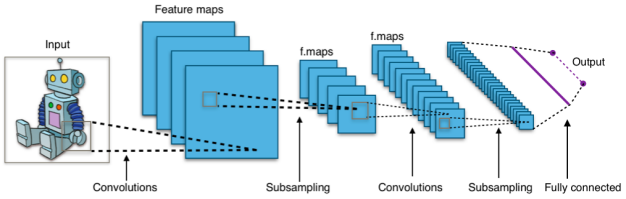

Figure 4: Layout of a typical convolutional neural network, [Public domain], via Wikimedia (https://upload.wikimedia.org/wikipedia/commons/6/63/Typical_cnn.png).

5 SOLUTION AND MODEL SPECIFICATIONS

As stated before, in this paper we based our model on a model created by Professor Donald Patterson of Westmont College[8] with modifications from Sam Myers. Myers's changes to the program mainly lied with reformatting the code in order to allow it to run on newer machines. His exposure to Patterson's videos were invaluable to the direction of the project.

The program was built using the Python programming language and was developed in Visual Studio Code on the Ubuntu distro of WSL2 (though was originally developed in the Spyder IDE using Anaconda package management, then developed on Visual Studio Code on Windows). The original program utilizes the Keras, TensorFlow, and Garbage Collector libraries in order to create the neural network and train and collect the data from it. A majority of the work was done with Keras, as its various sub-libraries cover image preprocessing, convolution layers, activation functions, early stopping, and other artificial intelligence-related functions. TensorFlow was used to clear the backend of any leftover memory in order to ensure previous builds would not interfere with a current program run. TensorFlow also was responsible for the backend calculations related to model generation. Garbage Collector functioned in a similar way, collecting any stray generations in the backend of the program, preventing any potential errors that may occur with leftover data still left in the system from previous program runs.

5.1 Original Model Generation

The first model generation program for this project came out of a previous project conducted under the supervision of Dr. Ruthie Halma. Loosely based on previous work done by Patterson and Myers, the initial program consisted of three convolutional layers consisting of 32, 64, and 32 neurons respectively, each with a 3 by 3 layer gathering the pixel data and outputting to the next layer based on a relu function. After each individual convolutional layer, there is a max pooling layer with a 2 by 2 filter. The output layer consisted of a sigmoid function. The network outputted a .json file and .h5 file (and eventually a Saved Model in Tensorflow's .pb Protobuf format) to the working directory that can be used at a later time.

Training and validation datasets were initially pulled from only one Kaggle dataset [7], being split between 1,962 and 100 respectively. Each dataset was broken down into subsets of 0 through 9 (according to American Sign Language) and was still checked to ensure no large biases were present that may affect model generation. Each input image was cropped and resized to 400 by 400 pixels, and the dataset was artificially inflated by rescaling, shearing, zooming, and flipping images in the training set. The batch sizes that the data is processed in is size 10, with the training sample size and the validation sample size each being set at 25. The maximum number of epochs that the program would run for was 250, and early stopping was added to decrease the validation loss (explained in later sections). Finally, the entire model generation program was automated so that if a minimum loss threshold was not reached (in this case 0.1, or a validation loss of 10 percent), the program would restart from the beginning.

5.2 Early Stopping

Early stopping, as defined by Brian Christian and Tom Griffiths, is a "regularization technique" in which models can "be kept from becoming overly complex simply by stopping the process short, before overfitting has had a chance to creep in"[1]. In order to apply this technique to the program, the Keras library was used to create an early stopping callback function. After researching the documentation for the callback function, early stopping was then implemented into the program.

After some initial testing tweaking the arguments of the function, the early stopping function was set up to measure the validation loss of the program and to wait 50 epochs beyond the best measured validation loss in order to see if the value would improve (in this case, decrease; loss will be described in the Data section of this paper). If 50 epochs passed and the validation loss did not drop below the best measured value, the program would terminate early and restore the neuron weights to what they were when the program achieved its best validation loss. In order to be considered an improvement in the eyes of the program, the measured validation loss for an epoch must decrease by a minimum delta of 0.01 or more. This was chosen due to the minute improvements that we were seeking in later stages of model generation: any little improvement was a step in the right direction for the program. In later implementations of the Early Stopping callback, the callback was only activated after a certain number of epochs in order to let the model generation "warm up", as it is assumed that the models generated near the beginning of the execution would not be good enough to keep, so were skipped over in the count.

```
# Added Early Stopping
my_callback = [EarlyStopping(
    monitor = 'val_loss',
    min_delta = 0.1,
    patience = 5,
    mode = 'auto',
    baseline = 1,
    restore_best_weights = True)]
```

Figure 5: An early version of the Early Stopping Keras callback implemented in the early stages of development, via Caringal.

5.3 Modified Model Generation

The initial version of the model generation program provided a good basis for improvements in various areas, ranging from advancements in terms of the model itself (with the major measures of success being validation accuracy and validation loss) as well as increasing the ease-of-use and variety of statistics at the hands of the researchers. And as we increased the size of our dataset, combining two more datasets on top of the original, We needed access to greater amount of computing power using only the resources we already had on hand.

Dr. Don Bindner of Truman State University was consulted in order to provide access to the University's virtual machines and CPU cores, as well as to help set up the development environment that allowed us to iterate on the original model generation program. Migrating from a local version of Python on Ubuntu WSL2 to an IDLE/Debian setup proved somewhat challenging, though not impossible. Various libraries had to be loaded in and reworked in order to provide the same functionality that was originally present in the initial stages of development. For example, the Keras and Tensorflow libraries had to be uninstalled and reinstalled various times as the default pathways were not allowing the two libraries to communicate between one another. In the end, all of the libraries were successfully and correctly installed, and we were able to proceed with developing the model.

Various parameters and layers were attempted, tested, and recorded during the course of this project. The bulk of the early models were created using Keras's Conv2D and MaxPooling2D layers with 'relu' hidden Activation layers, which were eventually Flattened into Dense and Dropout layers with a final output 'sigmoid' Activation layer. For more specifics behind each model, please visit the connected Github repository for this program https://github.com/jaqlagnirac/summer2023-ai-research.

5.4 Discord Bot Integration

One of the early major improvements in the pipeline is the research, implementation, and integration of a Discord automated user (or "bot"). Discord is a free voice and instant messaging platform, popular with many age demographics but prolific with college-age students. For this reason, as well as a bot's other functionalities beyond this project (and by extension Discord itself), it was decided that Discord would be the main platform to get updates of the program while the researchers were physically away from the computer. Discord's API was researched through documentation on the Internet, and the program was reworked to allow for the Discord library to mesh with the existing model generation program. The bot ran the model generation program; gathered the best validation and testing accuracies and losses (testing statistics discussed later), and the model run elapsed time (discussed later); formatted and timestamped the metrics; and sent a text message to a Discord server set up for the express purpose of documenting this project. A subroutine bot would repeat this process until both the minimum loss threshold and maximum accuracy threshold were met, at which time the bot would announce that the model generation was done and output the total elapsed time, but this was eventually phased out in later model generations as we began fine-tuning the model's parameters. In the beginning, a new Discord channel was set up to document each program run, but automatic channel generation per program run was set up in order to turn each pipeline run into a single-button click on sand.

The benefit of Discord is that anyone with the invite link (https://discord.gg/D2JC36t7QE) can view the formatted data, allowing for the easier transmission of data and real-time updates to those wanting to stay in the loop.

Figure 6: A screenshot of the a program run through Discord. Note the network structure at the top of the image, as well as the various channels on the left, via Caringal and Discord (https://discord.gg/D2JC36t7QE).

5.5 Elapsed Time Measures

The second early major improvement to the program is the integration of the Time library to provide another statistic to be measured: time. Along with creating accurate models using little to no monetary investment, another objective was to see how quick it was to make a model with the highest accuracy and lowest loss. In order to achieve this, the Time library was combined with the Discord bot, allowing the program to post the formatted local timestamp during the program's start, during each iteration of the model run, as well as the local timestamp when the best model was achieved (a formatted example: Wed Jul 20 00:08:41 2022). The timestamp implementation also allowed for an automatic comparison of the start and end times by the program, allowing the bot to calculate the elapsed time for each model iteration as well as for the program run as a whole.

5.6 Introduction of Testing Statistics

Through further research, the move was made away from treating validation accuracy and loss as the end point-of-comparison between generated models due to the fact that the statistics outputted during validation were a measure of the model's ability to discern images contained within the validation set, and not "brand-new" and "never seen before" images. Therefore, the model may have become overfitted to both the validation and training sets, invalidating the model and masking the true performance of it.

A new program was added to the pipeline, outputting the test statistics on a testing set kept completely separate from both the training and the validation set, as well as outputted a summary of the model's layers and neurons. The new breakdown of the images (still totalling 11,762) was 8,712 training images, 2,905 validation images, and 145 testing images, roughly mirroring the 60%-30%-10% split seen in some literature online (and which was found to obtain the best results). The test statistic output was eventually folded into the model generation and directly added to the generation program as to increase the speed and decrease the time between iterations.

5.7 Other Keras Callbacks

In addition to the initial adoption of the Early Stopping callback, other callbacks were added on in the later stages of development in order to attempt to increase the testing accuracy and decrease the testing loss or to provide a better research experience and give researchers a greater breadth of tools to utilize.

One of the earliest callback additions was the CSVLogger, which outputted the a comma-separated values table file that contains five columns: epoch, accuracy, loss, validation accuracy, and validation loss. This table file provides a better visual overview between each epoch, and can be fed into ancillary programs (which have not been developed as of right now, but may contain packages such as Python's Matplotlib) to future researchers to better understand the model generation life cycle of the program.

Next, we added the LearningRateScheduler callback to test its effectiveness at improving the model's metrics. For background, there is a concept in machine learning in which a model learns from its training data at a certain rate (i.e. the learning rate). By reducing the rate that a model learns continuously in the later stages of model generation, we are able to slowly "freeze" the model and prevent overfitting. A scheduler was added and tested, to various degrees of success.

Finally, we added the TensorBoard callback to increase the turnaround time on model data analysis. TensorBoard is a visualization toolkit that comes with Tensorflow, and provides graphing tools and other tracking and visualizing implementations to better understand the main and accompanying metrics and statistics. This allowed us to make better-informed decisions when it came to tweaking the model's architecture, shape, and parameters.

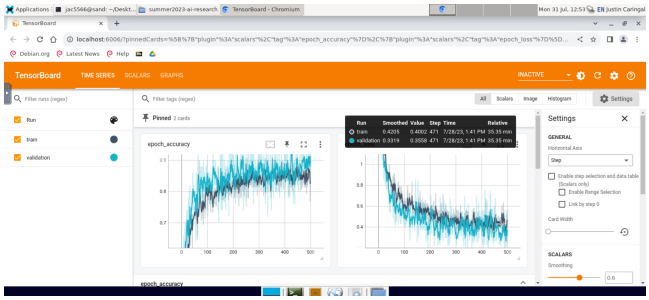

Figure 7: A screenshot of a TensorBoard output of epoch accuracy (left) and epoch loss (right) for a program execution, via Caringal.

5.8 Leveraging Tensorflow Model Zoo

As will be discussed later, the built-from-scratch models created in the earlier stages of development did not produce viable models for implementation, with testing accuracies around 60% and

testing losses being upwards of 400%. The decision was eventually made to leverage a pretrained model from Tensorflow's Model Zoo, a collection of similar models that comes standard with the Tensorflow package and can be downloaded from Tensorflow's website. MobileNetv2 was chosen due to its extensive training on ImageNet, a database of more than one million images. Through testing, varying amounts of MobileNetv2's 154 layers were "frozen" (i.e. made untrainable) to test the efficacy of the model and allow for the model to learn in various iterations of the generation process. In order to connect the existing model to the requisite outputs, a GlobalAveragePooling2D layer and a Dropout layer were added between a Dense and sigmoid Activation layer (for more, please see https://github.com/jaq-lagnirac/summer2023-ai-research). As will be discussed in the Data section, a noticeable improvement in both testing accuracy and testing loss was observed, leading us to focus much of our later efforts on improving the pretrained model rather than focus our efforts on the built-from-scratch model.

5.9 Beginning OpenCV Implementation

Research was also conducted into how to implement a generated model into a real-world application, in this case through the use of an integrated internal web-camera on a standard Windows laptop. Python's OpenCV package was chosen for this, being one of the premiere open-source and readily available computer vision libraries out there (with possible cross-compatibility with the C family of languages). A test implementation was created using the built-in Haar Cascade classifier, set up to detect faces and eyes through the internal webcam. Attempts were made at integrating the generated ASL model into this implementation, but due to the differences between Tensorflow and OpenCV (especially the migration of most resources from Tensorflow Version 1 to Tensorflow Version 2, and specifically the differences in eager execution and output types), we were not able to successfully integrate these two programs. As will be discussed in Future Directions, we hope to achieve a successful implementation and integration in the future.

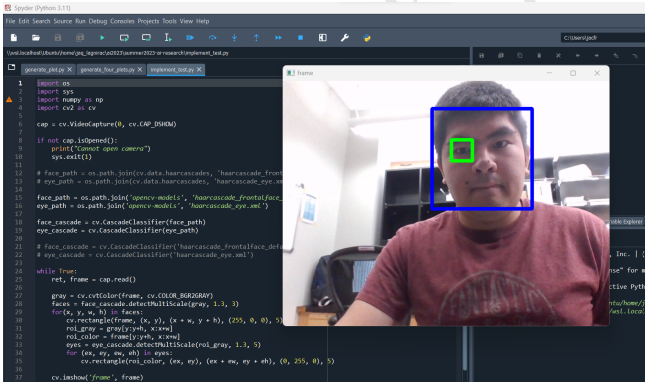

Figure 8: OpenCV's built-in Haar Cascade Classifier, set up to detect faces and eyes, Jul. 20, 2023.

6 DATA

Four main data points were collected throughout the training and validation of the various trials. These were testing accuracy, testing loss, validation accuracy, and validation loss. Accuracy is the a measure of how often the program is correct in identifying which image contains people and which image does not. Loss is a cost function which measures how bad the model was able to predict the image for a single example. Testing accuracy applies to the testing set, while validation accuracy is measured for the validation set. The same applies for testing loss and validation loss: the former is for the testing set and the latter for the validation set. Training accuracy and loss are also measured, but are not important compared to these other metrics as they are an intermediary statistic only applicable and relevant when training the model. The initial neural network was created with the explicit purpose to maximize the accuracy (read: training accuracy) of the model. Validation loss was used as the metric to determine when the program should halt early. The main goal of early stopping is to reduce overfitting to the training set, and validation loss is measured by the model's ability to fit to new unseen data such as the testing dataset. Thus, it makes sense for early stopping to use validation loss as its monitor value.

6.1 Cross-Validation

Cross-validation was applied to all versions of the program, allowing the program to produce 4 data points instead of 2. As described by Christian and Griffiths, cross-validation is a method which "[assesses] not only how well a model fits the data it's given, but how well it generalizes to data it hasn't seen"[1]. In addition to early stopping (described in an earlier section), cross-validation in a model serves to reduce the amount of overfitting, as if a model has a low loss but a high validation loss, it is more than likely that the model has become accustomed to the training dataset and will not perform well outside of it. In addition, if the model has both a low training and validation loss but a high testing loss, the model may have become overfitted to both the training and validation sets, an equally bad case.

6.2 Data Collected

As stated before, 4 measurements were taken throughout the tests. The most relevant metrics are testing accuracy and testing loss, as both apply to unseen data inputs (which is what will be inputted into the model in the real world).

Below contains a table of the some results, picked from the early and later stages of this project as to better illustrate the progress made during the course of this project; this means that not all of the program runs executed are shown. In addition, the exact architecture of each model cannot be easily and succinctly expressed. In both of these cases, as well as for more information, please visit the Discord server set up for this project: https://discord.gg/D2JC36t7QE.

The shorthand "C/BFS" describes models that were "custom/built-from-scratch" which predate the MobileNetv2 implementation (denoted by "MNv2"). "Code" denotes an automatically generated label tied to the local timestamp. Its full-length format is "YYYY-MM-DD-HHMMSS" ("Year-Month-Day-HourMinuteSecond"). Two separate time codes exist: one for the Discord channel and one for the output file directory. Time codes located on the table reference the former (the channel) and are shortened, as the bulk of the relevant model generation took place during the same calendar month and year

Table 1: Various Results Achieved by Programs

Model	Code	VA	VL	TA	TL	ET
C/BFS	20-131023	94.0	16.1	60.0	140.3	2.67
C/BFS	20-160248	94.0	16.2	48.3	161.3	1.77
C/BFS	20-182231	90.0	20.7	63.4	132.7	1.53
C/BFS	25-113027	100.0	3.9	62.1	128.9	1.55
C/BFS	26-192442	98.0	13.9	46.9	203.1	1.35
C/BFS	27-101838	96.0	8.8	63.4	142.5	1.19
MNv2	27-143535	94.0	32.0	71.7	83.5	0.25
MNv2	27-154003	94.0	32.9	71.7	93.0	0.17
MNv2	27-162115	98.0	19.0	71.0	80.0	0.25
MNv2	27-164541	94.0	20.1	72.4	75.6	0.32
MNv2	27-215322	100.0	1.4	74.5	81.2	0.36
MNv2	28-082337	96.0	21.2	75.2	81.5	0.38
MNv2	28-110102	96.0	20.7	73.1	72.6	0.35

VA - Validation Accuracy (%)
VL - Validation Loss (%)
TA - Testing Accuracy (%)
TL - Testing Loss (%)
ET - Elapsed Time (hours)
NOTE: Percents rounded to nearest tenth, hours rounded to nearest hundredth.

(July 2023), ergo those fields are irrelevant. If you are accessing the Github repository or the Discord server, please add "2023-07-" to the codes to properly find the correct channels and files.

The table contains percent values above 100%, particularly with early iterations of testing loss. This is due to the use of the built-in Tensorflow categorical cross-entropy loss function. Therefore, it is not uncommon or incorrect to have losses exceeding 100%.

7 PERFORMANCE EVALUATION

We were happily surprised with our outputted model. With the introduction of new metrics (namely testing accuracy and testing loss), more emphasis was placed on fine-tuning the model's architecture than had been in the past. And with over 11,000 input images used (as well as a jump to 10 separately identifiable classes), we are very happy to have produced a model with a relatively high testing accuracy and reasonable loss, which translates to an acceptable accuracy in the real world. Although we were unable to successfully integrate this model using OpenCV, steps have already been taken and research has already been done so that we might implement this model in the future. The initially daunting number of images present in the dataset proved a surmountable obstacle due to an increase in understanding over the course of two years, as well as an increase in computing power offered by Dr. Bindner through the use of Truman's Sand servers. As will be discussed down below, we hope that we are able to capitalize on the lessons learned and the research conducted in the future in order to produce an improved model and to implement it successfully.

8 CONCLUSION

By this project's conclusion, we hoped to expand our current knowledge and understanding in the computer vision field by generating

Figure 9: A QR code for the invite link to the Discord server used in this project, via Caringal and Discord (https://discord.gg/D2JC36t7QE).

an artificial intelligence model to detect and classify the differences between the American Sign Language numbers zero through nine. By implementing new metrics, tools, and functionalities into our existing Keras/Tensorflow pipeline, as well as beginning research and development into the successful implementation and integration of OpenCV, we were able to begin building on our wide knowledge base of Python and its associated libraries and packages, bringing us one step closer to a usable live computer vision model.

8.1 Future Directions

We are very proud with the output and knowledge gained from this project, but we still have a breadth of ideas and possible avenues of research to go down. The next major step is implementing the generated model into a preliminary use using widely available cameras, such as integrated webcams or smartphone cameras. Much research has been conducted in this area, and many different directions can be taken. One would be to use third-party tools (found on various forums and chats) to convert the generated Tensorflow v2 model into a usable format for OpenCV. However, due to various versions existing and with Tensorflow v2 only existing for four years, the literature is somewhat confusing and can be bogged down between version migrations. Another possible avenue would be to train models directly using OpenCV, or another package such as ImageAI or the PyTorch framework. Not much local research has been conducted down these roads, and would require a greater time investment on our part in order to get up and running.

Beyond these major paths, smaller minor ones can be performed in the future. A greater suite of tools can be developed using various packages such as Python's Matplotlib in order to gain a better

understanding of the fine-tuning of a model's architecture. Much of the advancements in this particular area during this project were developed in the later stages of research, so more time could be allowed for this in the future. In addition, a standardization of parameters would be beneficial to keeping the various programs used in line with one another. A .json configuration file was initially set up, but was not wholly integrated into the pipeline do to shifting focuses and priorities of researchers. This, however, will become of greater use as the focus shifts back form generation to implementation. Finally, the last minor improvement would be the automation of model tweaking and reporting. Currently, most adjustments to the model are manually performed, as are the reporting of the different layers of the model to Discord (the automation discussed previously executes the same model over and over, never making any tweaks to the architecture). The successful integration of both of these aspects to the Discord pipeline may prove invaluable in reducing the downtime between iterations and leading to a faster development of a high-quality model.

9 RELATED LINKS

Below is a list of notable linked referenced throughout this paper. For more information about the project or to reach us, please visit the links below:

Please visit https://github.com/jaq-lagnirac/summer2023-ai-research to view the full Github repository as well as the network graph to see the various commits and stages of development of the project.

Please visit https://discord.gg/D2JC36t7QE to gain access to the Discord server used in this project, which holds all of the data obtained throughout the course of this project.

ACKNOWLEDGMENTS

To Dr. Ruthie Halma of Truman State University, for providing us the opportunity to gain invaluable experience in the artificial intelligence and computer vision fields.

To Dr. Don Bindner of Truman State University, for providing us with the necessary tools and computing power that allowed us the ability to properly generate our models.

And to Sam Myers, for providing us with the right direction in creating the convolutional neural network model.

REFERENCES

[1] Brain Christian and Tom Griffiths. 2016. *Algorithms to Live By: The Computer Science of Human Decisions.* William Collins, London, Great Britain.

[2] Hadelin de Ponteves. 2019. *AI Crash Course: A fun and hands-on introduction to reinforcement learning, deep learning, and artificial intelligence with Python.* Packt Publishing, Birmingham, England.

[3] Takahiro Sogawa et al. 2020. Accuracy of a deep convolutional neural network in the detection of myopic macular diseases using swept-source optical coherence tomography. *Public Library of Science* (April 2020). https://journals.plos.org/plosone/article?id=10.1371%2Fjournal.pone.0227240

[4] Lexset. 2022. Synthetic ASL Numbers. https://www.kaggle.com/datasets/lexset/synthetic-asl-numbers

[5] Gang Song Liming Wang, Jianbo Shi and I fan Shang. 2020. Object detection combining recognition and segmentation. *University of Pennsylvania School of Engineering and Applied Science* (April 2020). https://www.seas.upenn.edu/~jshi/papers/obj_det_liming_accv07.pdf

[6] Tianrui Liu and Tania Stathaki. 2018. Faster R-CNN for robust pedestrian detection using semantic segmentation network. *UFrontiers in Neurorobotics* (Oct. 2018). https://www.frontiersin.org/articles/10.3389/fnbot.2018.00064/full

[7] Arda Mavi. 2017. Sign Language Digits Dataset. https://doi.org/10.34740/KAGGLE/DSV/11071

[8] Donald Patterson. 2020. Machine Learning Image Recognition Website Tutorial (python, flask, keras and tensorflow). *YouTube* (Jan. 2020). https://www.youtube.com/watch?v=9tLWj7iOiP8

[9] Ayush Thakur. 2019. American Sign Language Dataset. https://www.kaggle.com/datasets/ayuraj/asl-dataset