Development of a 3D Convolutional Neural Network for ASL Gesture Recognition

Aidan Carter
Honors College
Indiana State University
Terre Haute, IN
arcarter129@gmail.com

Abstract* – *This paper presents the development process taken for constructing a 3D convolutional neural network for classifying American Sign Language gestures into English. The initial goal of the project was to implement a complex model based on the Inflated Inception V1 3D CNN architecture, however, was revised to a custom shallow 3D CNN due to hardware constraints. The model trained on a subset of 10 ASL labels derived from the MS-ASL dataset. Evaluation of the model indicated a validation accuracy of 71.429% and a testing accuracy of 72.840%, demonstrating the effectiveness of the model's ability to generalize to unseen samples. This project emphasized the importance of dataset preprocessing with the use of optical flow calculations to capture motion information. The results suggest that while the model can effectively classify gestures, model complexity needs to be increased to allow for an expanded vocabulary for practical use in real-world applications.

Keywords* – *3D Convolutional Neural Network, American Sign Language, Gesture Recognition, Optical Flow

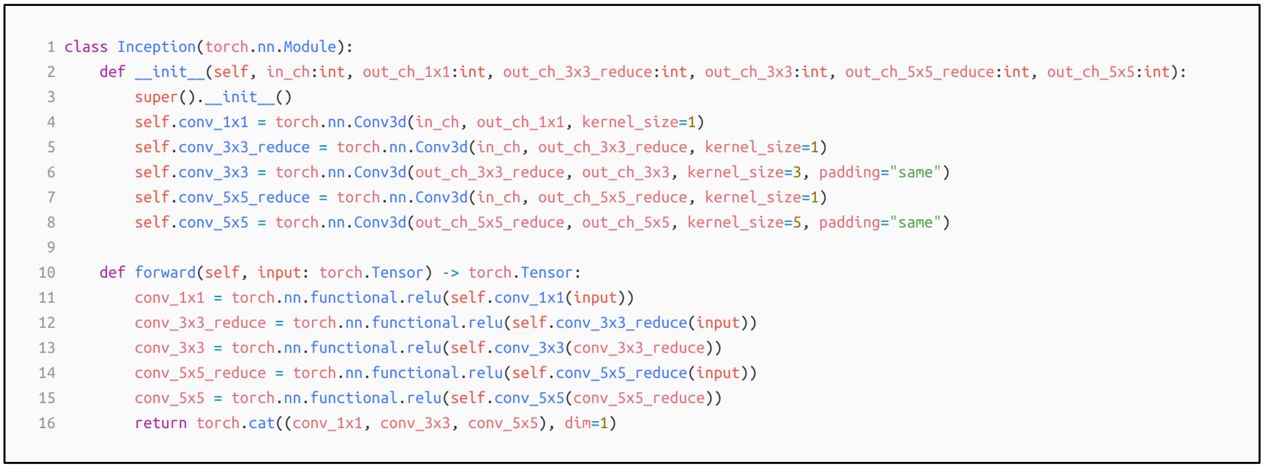
I. Introduction

While it has become quite easy to translate between two written/spoken languages using accessible software, translating a signed gesture to a word is far from trivial. In the hopes to address this problem, this project explores the development of a gesture recognition and classification model designed to translate American Sign Language (ASL) gestures into English words [1]. The approach utilizes a 3D Convolutional Neural Network which have proven effective in action recognition [2]. Building on this foundation, the model seeks to learn both spatial and temporal features necessary for accurate gesture interpretation.

While the initial goal was to implement an inception-based model such as an inflated version of GoogLeNet (Inception V1), a deep-learning model was unable to be trained efficiently on the limited hardware available [2][3]. Instead, the model was designed to be shallow and lightweight, significantly reducing the number of trainable parameters and thus reducing the memory requirements and training time of the model. Despite being compact, the model aims to extract meaningful spatial and temporal features and generalize to new, unseen data to classify gestures effectively.

The model was trained on a subset of 10 classes, down from the 1,000 classes available in the dataset. This was to reduce the training time significantly to make it feasible to train on limited hardware. The goal of this project is to demonstrate that meaningful gesture recognition can still be achieved in smaller networks and can then be scaled up to classify larger vocabularies. This can pave the way for future, real-time ASL gesture recognition and translation projects.

This paper outlines the process taken to develop and train such a model. Section II dives deeper into the initial I3D model design and the limitations discovered. Section III outlines the dataset and preprocessing steps. Section IV covers the data loading pipeline and training methodology, highlighting the various methods used in optimizing the model. Section V presents the simplified model adapted for use with the limited computational resources that were available, along with the revised goal of classifying a subset of 10 gestures. Section VI covers the evaluation of the model's performance, analyzing various training metrics and end results. Finally, Section VII concludes the paper with a discussion of limitations and future directions for expanding model capabilities.



```python
class Inception(torch.nn.Module):
    def __init__(self, in_ch:int, out_ch_1x1:int, out_ch_3x3_reduce:int, out_ch_3x3:int, out_ch_5x5_reduce:int, out_ch_5x5:int):
        super().__init__()
        self.conv_1x1 = torch.nn.Conv3d(in_ch, out_ch_1x1, kernel_size=1)
        self.conv_3x3_reduce = torch.nn.Conv3d(in_ch, out_ch_3x3_reduce, kernel_size=1)
        self.conv_3x3 = torch.nn.Conv3d(out_ch_3x3_reduce, out_ch_3x3, kernel_size=3, padding="same")
        self.conv_5x5_reduce = torch.nn.Conv3d(in_ch, out_ch_5x5_reduce, kernel_size=1)
        self.conv_5x5 = torch.nn.Conv3d(out_ch_5x5_reduce, out_ch_5x5, kernel_size=5, padding="same")

    def forward(self, input: torch.Tensor) -> torch.Tensor:
        conv_1x1 = torch.nn.functional.relu(self.conv_1x1(input))
        conv_3x3_reduce = torch.nn.functional.relu(self.conv_3x3_reduce(input))
        conv_3x3 = torch.nn.functional.relu(self.conv_3x3(conv_3x3_reduce))
        conv_5x5_reduce = torch.nn.functional.relu(self.conv_5x5_reduce(input))
        conv_5x5 = torch.nn.functional.relu(self.conv_5x5(conv_5x5_reduce))
        return torch.cat((conv_1x1, conv_3x3, conv_5x5), dim=1)
```

Figure 1. Inception module implementation showing both constructor and forward methods. [1]

II. Initial Model Design (I3D)

To tackle the initial goal of building a model capable of classifying 1,000 different ASL gestures, a model architecture based on the Inflated-3D Convolutional Neural Network was used. This model was originally designed for action recognition on the Kinetics dataset and had seen excellent performance in classifying videos containing gestures [2]. The I3D model itself is built upon the GoogLeNet (Inception V1) model and adapts its 2D convolutions into a third dimension in a process called *inflation* [2][3]. This allows the model to handle both spatial and temporal features for accurate action recognition.

To implement the network, the PyTorch library was used. PyTorch is an open-source deep learning framework which is well-suited for designing and implementing custom neural network architectures [4]. Building off of inflating GoogLeNet, it was practical to match the input dimensions and follow the architecture for each layer, only replacing the 2D convolutions with their 3D counterpart. To start development the Inception module was constructed. It would contain several convolution layers stacked next to each other from the same input, and then the output would be concatenated. To reduce dimensionality and the number of computations necessary, 1x1x1 convolutions were placed before the 3x3x3 and 5x5x5 convolutional layers [3][2]. While the original version of the Inception module contained a max-pooling layer as well, this was left out for simplicity of matching dimensions without additional layers or padding.

The full Inflated Inception-V1 3D (I3D) network begins with two convolutional layers with max-pooling layers in-between to reduce the dimensionality of the data early to reduce computational cost through the network. The layers have stride 2 in the spatial dimensions to essentially half the amount of data while using stride 1 in temporal dimension to retain motion data [3][2]. After the convolutional and max-pooling layers, several layers of the Inception module are used to extract fine features from the input data. After a number of Inception modules, a pooling layer will be placed to further reduce the spatial and temporal dimensions as the model gains more feature maps. The final pooling layer consolidates the special and temporal dimension into 1x1x1 datapoints for each flattening. The flattened data is then input into a fully connected layer and returned from the model as logits to be used for input to a loss function or for classification based on the maximum value.

The full implementation uses the layers described and calls upon the previously defined Inception module to construct the full network (see Figure 2). The forward pass is then defined, connecting the layers together while adding activation functions between layers. The activation function used is ReLU (Rectified Linear Unit), a common function that adds non-linearity to the model by updating neurons to be the maximum value between 0 and the current value [4]. This helps the model better learn complex features from the data. The flatten function was used between the `pool4` and fully connected linear layer on dimension 1, which would retain the batches and flatten along the feature maps. No activation function was used before the output that way the model would return the raw logits to the loss function during training.

```python
class I3D(torch.nn.Module):
    def __init__(self, classes:int):
        super().__init__()
        self.conv0 = torch.nn.Conv3d(3, 64, kernel_size=7, stride=2)
        self.pool0 = torch.nn.MaxPool3d(kernel_size=(1,3,3), stride=(1,2,2), ceil_mode=True)
        self.conv1 = torch.nn.Conv3d(64, 192, kernel_size=3, stride=1)
        self.pool1 = torch.nn.MaxPool3d(kernel_size=(1,3,3), stride=(1,2,2), ceil_mode=True)
        self.inc0 = Inception(192, 64, 96, 128, 16, 32)
        self.inc1 = Inception(224, 128, 128, 192, 32, 96)
        self.pool2 = torch.nn.MaxPool3d(kernel_size=3, stride=2, ceil_mode=True)
        self.inc2 = Inception(416, 192, 96, 208, 16, 48)
        self.inc3 = Inception(448, 160, 112, 224, 24, 64)
        self.inc4 = Inception(448, 128, 128, 256, 24, 64)
        self.inc5 = Inception(448, 112, 144, 288, 32, 64)
        self.inc6 = Inception(464, 256, 160, 320, 32, 128)
        self.pool3 = torch.nn.MaxPool3d(kernel_size=3, stride=2, ceil_mode=True, padding=1)
        self.inc7 = Inception(704, 256, 160, 320, 32, 128)
        self.inc8 = Inception(704, 384, 192, 384, 48, 128)
        self.pool4 = torch.nn.AvgPool3d(kernel_size=7, stride=1)
        self.linear = torch.nn.Linear(896, classes)

    def forward(self, input:torch.Tensor) -> torch.Tensor:
        conv0 = torch.nn.functional.relu(self.conv0(input))
        pool0 = self.pool0(conv0)
        conv1 = torch.nn.functional.relu(self.conv1(pool0))
        pool1 = self.pool1(conv1)
        inc0 = self.inc0(pool1)
        inc1 = self.inc1(inc0)
        pool2 = self.pool2(inc1)
        inc2 = self.inc2(pool2)
        inc3 = self.inc3(inc2)
        inc4 = self.inc4(inc3)
        inc5 = self.inc5(inc4)
        inc6 = self.inc6(inc5)
        pool3 = self.pool3(inc6)
        inc7 = self.inc7(pool3)
        inc8 = self.inc8(inc7)
        pool4 = self.pool4(inc8)
        pool4_flattened = torch.flatten(pool4, start_dim=1)
        return self.linear(pool4_flattened)
```

Figure 2. Inflated Inception 3D Model implementation showing both constructor and forward methods. [1]

With the model complete, the next steps were to download and prepare the dataset.

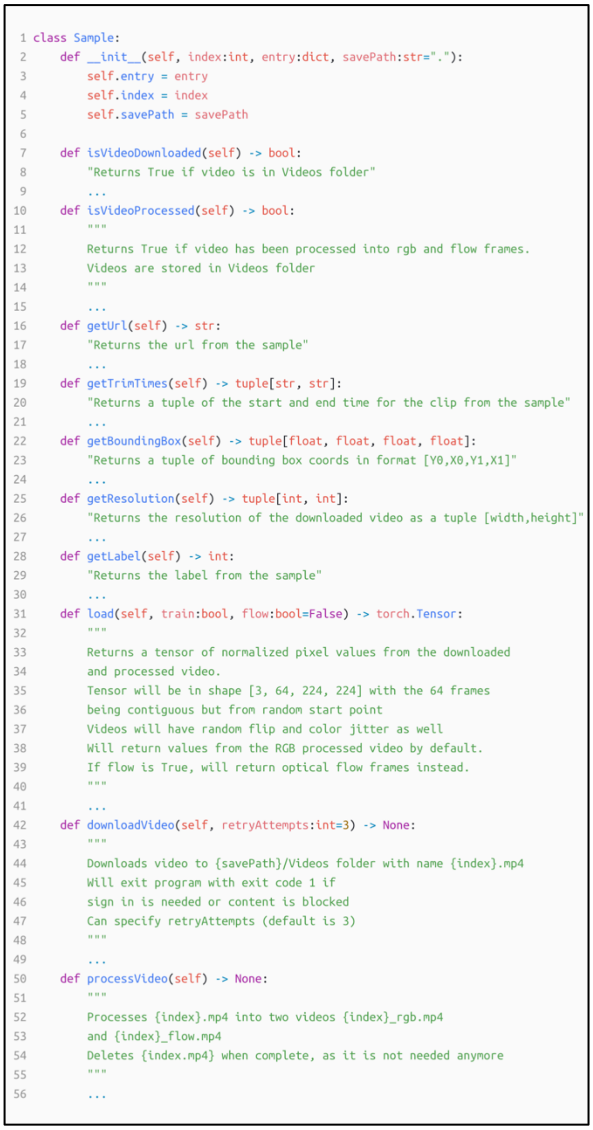
III. Dataset & Preprocessing

The dataset used in this project is the MS-ASL dataset from Microsoft. It consists of 25,513 samples split between training, validation, and testing set for 1,000 different labels [5]. The actual download for the dataset consisted of several JSON files that organized the data entries between the training, validation, and testing sets as well as a file detailing the individual labels. Each entry in the JSON files contained several fields such as label, URL to the video, height & width of video, normalized bounding box of the signer, FPS of video, start and end times for the gesture within the video, as well as identifying the signer and text in video [5]. For ease of use, the entries were abstracted into two Python classes, one describing each entry called `Sample` and one describing the entire dataset, called `Dataset`.

The `Sample` class contained three attributes. The `entry` attribute contained the Python dictionary representation of the JSON dataset entry. The `index` attribute contained the index of the entry within the JSON file, which was used to name the video once downloaded. The `savePath` attribute would store the path to where the video was downloaded. Several methods were introduced to quickly access data from the dataset entry. `getURL()`, `getTrimTimes()`, `getBoundingBox()`, `getResolution()`, and `getLabel()` would all return the corresponding information from the entry in Python datatypes. Two helper methods, `isVideoDownloaded()` and `isVideoProcessed()` were used to identify whether the videos were downloaded or processed. All of the previously identified methods were used to construct the `downloadVideo()` and `processVideo()` methods. These would respectively download the video using the yt-dlp Python library and then process each video into its RGB and Optical Flow counterparts.

Diving deeper into `downloadVideo()`, the method would attempt the video for the Sample instance a number of times based on the `retryAttempts` parameter. It utilized the yt-dlp library to do this, which would attempt to download a video based on the URL [6]. However, some of the videos in the dataset have either been deleted or privatized since the release of the dataset. This method would catch these errors, logging them for future reference. Certain errors, such as needing to be signed in to view the video, were the result of bot protections put in place by YouTube. To circumvent this, the method also provided an account's credentials to YouTube while also checking if the account was flagged as a bot [6]. If an account related error was caught, the program would end as any subsequent video would also be unable to download due to this error.

The `processVideo()` method processed the downloaded videos based on the entry. First, it would crop and trim the video based on the bounding box and timing fields in the entry. Since the bounding box information was normalized, it needed to be unnormalized based on the video's height and width. The actual crop and trim of the video was processed using the ffmpeg-python library [7]. The opencv-python library was then used for optical flow calculations on the frames [8]. Each frame began by being resized to the input of the model, 224x224 pixels. The RGB frames were written to the output file for the RGB video while simultaneously converted into grayscale for the optical flow calculation. To calculate the optical flow between the current and previous frames, the Farneback method provided by opencv-python was used, along with the code being adapted from the documentation surrounding this method [8]. The reasoning for these calculations is that gestures use motion, so extracting motion data from the videos is essential for good predictions. After the calculations were complete, the optical flow frames were also written to a file for use in the model. The method also removed unnecessary files from the disk, such as the full downloaded video, as they were no longer needed and space was limited on the hardware used.



```python
class Sample:
    def __init__(self, index:int, entry:dict, savePath:str="."):
        self.entry = entry
        self.index = index
        self.savePath = savePath

    def isVideoDownloaded(self) -> bool:
        "Returns True if video is in Videos folder"
        ...
    def isVideoProcessed(self) -> bool:
        """
        Returns True if video has been processed into rgb and flow frames.
        Videos are stored in Videos folder
        """

        ...
    def getUrl(self) -> str:
        "Returns the url from the sample"
        ...
    def getTrimTimes(self) -> tuple[str, str]:
        "Returns a tuple of the start and end time for the clip from the sample"
        ...
    def getBoundingBox(self) -> tuple[float, float, float, float]:
        "Returns a tuple of bounding box coords in format [Y0,X0,Y1,X1]"
        ...
    def getResolution(self) -> tuple[int, int]:
        "Returns the resolution of the downloaded video as a tuple [width,height]"
        ...
    def getLabel(self) -> int:
        "Returns the label from the sample"
        ...
    def load(self, train:bool, flow:bool=False) -> torch.Tensor:
        """
        Returns a tensor of normalized pixel values from the downloaded
        and processed video.
        Tensor will be in shape [3, 64, 224, 224] with the 64 frames
        being contiguous but from random start point
        Videos will have random flip and color jitter as well
        Will return values from the RGB processed video by default.
        If flow is True, will return optical flow frames instead.
        """

        ...
    def downloadVideo(self, retryAttempts:int=3) -> None:
        """
        Downloads video to {savePath}/Videos folder with name {index}.mp4
        Will exit program with exit code 1 if
        sign in is needed or content is blocked
        Can specify retryAttempts (default is 3)
        """

        ...
    def processVideo(self) -> None:
        """
        Processes {index}.mp4 into two videos {index}_rgb.mp4
        and {index}_flow.mp4
        Deletes {index.mp4} when complete, as it is not needed anymore
        """

        ...
```

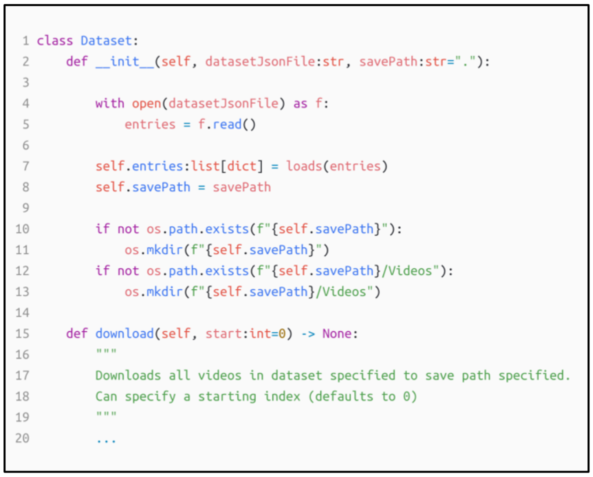
Figure 3. Sample class methods & attributes. [1]
Note: Method implementations are removed.

Further processing was done in the `load()` method. This method would load the video from the disk to a PyTorch tensor, which would be stored on the GPU memory if CUDA processing was available using the torchcodec library [9]. This would speed up load times as well as make sure the video was already in GPU memory when input into a model also placed on the GPU. The `train` parameter would tell whether augments to be made to the loaded video should be applied. This is useful during training to artificially inflate the size of the training dataset and allow for more variability. The augments by the torchvision library were a random horizontal flip 10% of the time to account for the occasional left-handedness, a random 0-5 degree rotation to account for different camera angles and hand movement, and some color jitter to account for slight variations in appearance (or in the case of optical flow where colors correlate to movement magnitude and angle, slight variations in speed/direction) [10]. All loaded videos regardless of training or optical flow/RGB would be extended or shortened to 64 frames to match the model's input dimensions. If longer, a random slice of 64 frames were used. Otherwise, the last frame was extended to meet the 64-frame requirement. The returned PyTorch tensors were also normalized so that each pixel value from the video would be between 0 and 1. This was done to improve model performance.

The `Dataset` class was used to further abstract the dataset to contain all of the samples. It would be initialized with the path to the JSON file containing the dataset entries, as well as a path to pass on to the samples for where videos should be saved. It contained a single method, used to download and process all the videos in the dataset. The `start` parameter was used to begin downloading from a set index, which was used in the cases of program exits and restarts. This method would also print to the screen progress information about the current download and preprocessing, as well as any errors that had occurred.

IV. Dataloader

The `Dataloader` class was constructed to abstract the entire dataset into a Python iterable object for easy access during training. It also added support for taking subsets of the dataset as well as receiving a set-size batch of videos at a time. To achieve this, several special attributes and dunder methods were used. There are several attributes contained in this class. The `dataset` attribute contains the `Dataset` object previously described which contains all the entries in the dataset. The `subset` attribute contains an integer of the subset of labels used. This would allow for the iterator object to only return videos that were within the subset. The `batchSize` attribute contains an integer of the number of samples to return when called as an iterator. The `flow` attribute dictates whether the returned video contains RGB or Optical Flow frames. The `currentIndex` attribute tracks the index currently returned from the iterator object. The `shuffledIndices` attribute contains all of the indices in the dataset in a random order, essentially shuffling the dataset so that the model would receive random samples which is useful during training. The `train` attribute contains whether the dataset is training data and is passed on to the `load()` method in the `Sample` class to enable augmentation.



```python
class Dataset:
    def __init__(self, datasetJsonFile:str, savePath:str="."):

        with open(datasetJsonFile) as f:
            entries = f.read()

        self.entries:list[dict] = loads(entries)
        self.savePath = savePath

        if not os.path.exists(f"{self.savePath}"):
            os.mkdir(f"{self.savePath}")
        if not os.path.exists(f"{self.savePath}/Videos"):
            os.mkdir(f"{self.savePath}/Videos")

    def download(self, start:int=0) -> None:
        """
        Downloads all videos in dataset specified to save path specified.
        Can specify a starting index (defaults to 0)
        """
        ...
```

Figure 4. Dataset class methods & attributes. [1]
Note: Method implementations are removed.

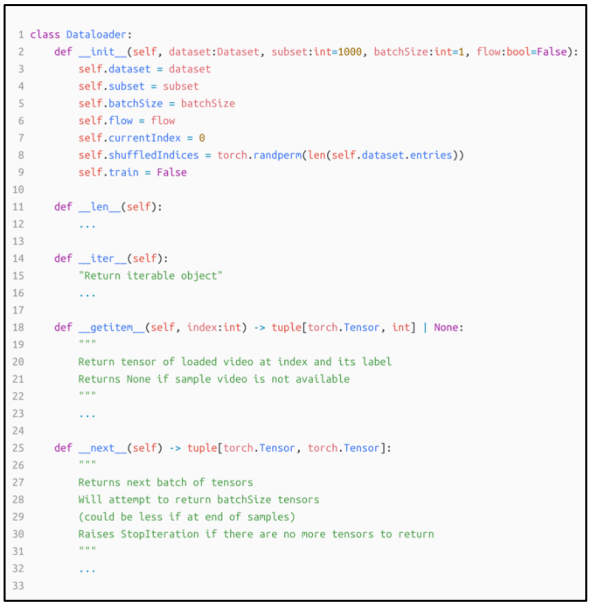
The dunder methods in the `Dataloader` class are used to make the class into a Python iterable object. The `__len__()` dunder method would return the length of the `Dataset` object based on the number of entries in the JSON file. The `__iter__()` dunder method returns an iterable object when an instance of the class is created. The `__getitem__()` dunder method returns a single loaded video and its corresponding label as a Python tuple. The method also has the possibility of returning a None type value when the video can be loaded (because it does not exist) or if the sample is not in the subset. That method was used when trying to load a video from a specific index, whereas the `__next__()` dunder method was used when calling class object as an iterable, such as in a Python for loop. This dunder method utilized the `__getitem__()` dunder method to return up to a `batchSize` of sample tuples (could return less if the end of the dataset was reached). This method would raise a `StopIteration` exception when called and there were no more samples in the dataset. The returned videos were shuffled by the `shuffleIndices` attribute to reduce overfitting on the model. These four dunder methods together, along with the `Sample` and `Dataset` classes, allowed for the entire training pipeline to be abstracted into a simple Python iterable object for use in the training logic.

V. Model Training

With the model created and the dataset downloaded and abstracted for easy use, the model could begin training. A class named `ASLModel` was constructed with two methods, `train()` and `test()`. Several attributes were introduced as well. Many of them are used as parameters passed to the `Dataloader` and `Dataset` classes, such as `savePath`, `batchSize`, `subset`, and `flow`. The reset are used to introduce key functions and hyperparameters to the training logic.

To begin, the `model` attribute contains the 3D-CNN model itself. The parameters of the model were loaded either to the CPU or GPU depending on whether CUDA processing is available on the machine. An optional `loadModelName` filename parameter could be defined in which the previous stated model would be loaded with parameter weights. This was useful for loading a pretrained model for testing. The `lossFunc` attribute contains the function used for calculating the loss in predicted vs actual values. For this model, it was trained using the `CrossEntropyLoss()` function provided by PyTorch [4]. This is an excellent function for using in classification problems with multiple classes and is the reason it was chosen for training on this model [4].

The next two attributes were used in conjuction to adjust the parameters of the model during training to minimize the loss computed from the loss function. This is the `optim` attribute and the `scheduler` attribute. The `optim` attribute contains the optimizer function which minimizes the loss. Originally, Stochastic Gradient Descent, or `SGD()`, was used as it is a simple and widely adopted method for optimization. However, the `AdamW()` optimizer was employed instead due to the fact that it converges faster and allows for weight decay for regularization [4]. The optimizer had an initial learning rate of 10^{-3} and weight decay of 10^{-2}. The `scheduler` attribute contains the function that adjusts the learning rate over time during training. The scheduler function used was `CosineAnnealingLR()`, which reduced the learning rate to 10^{-8} by the end of training. The learning rate follows a cosine curve, meaning most of the training the learning rate was kept high for larger improvements and then towards then end swung low to squeeze extra performance out of the model [4].

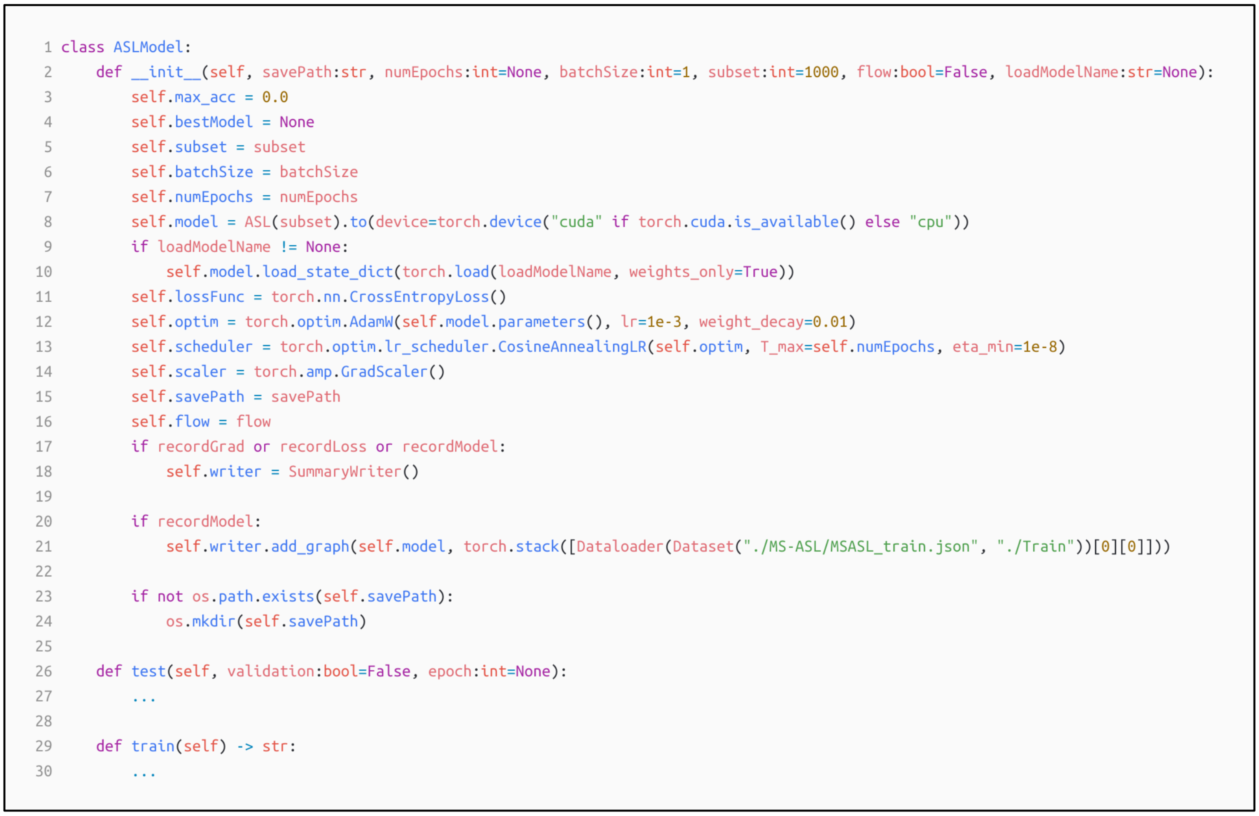


```python
class Dataloader:
    def __init__(self, dataset:Dataset, subset:int=1000, batchSize:int=1, flow:bool=False):
        self.dataset = dataset
        self.subset = subset
        self.batchSize = batchSize
        self.flow = flow
        self.currentIndex = 0
        self.shuffledIndices = torch.randperm(len(self.dataset.entries))
        self.train = False

    def __len__(self):
        ...

    def __iter__(self):
        "Return iterable object"
        ...

    def __getitem__(self, index:int) -> tuple[torch.Tensor, int] | None:
        """
        Return tensor of loaded video at index and its label
        Returns None if sample video is not available
        """
        ...

    def __next__(self) -> tuple[torch.Tensor, torch.Tensor]:
        """
        Returns next batch of tensors
        Will attempt to return batchSize tensors
        (could be less if at end of samples)
        Raises StopIteration if there are no more tensors to return
        """
        ...
```

Figure 5. Dataloader class methods & attributes. [1]
Note: Method implementations are removed.

To record the model during training and validation, the PyTorch version of the TensorBoard library was used [4][11]. This allowed for tracking the loss and accuracy during training on both the training and validation datasets. Gradient distributions calculated during training were also tracked. A visual map of the model could also be constructed using this library. Whether these values were recorded was controlled by arguments passed into the program on the command line.

The `train` method contains simple train/validate logic. First, the entire training dataset is run through with the model updating parameters using the optimizer. Batches of input run through the model with the logits returned being fed into the loss function. A backwards pass if performed on the model, and its parameters are updated by the optimizer. After each epoch, the `test` method is called with the parameter `validation` set to `True`. This runs through the model on the validation set in evaluation mode, so that no parameters are update and gradients are not calculated. The current epoch is also passed into the method for TensorBoard recording reasons. After training for `numEpochs`, the train method returns the best model which can then be passed into the class to load and test.



```python
class ASLModel:
    def __init__(self, savePath:str, numEpochs:int=None, batchSize:int=1, subset:int=1000, flow:bool=False, loadModelName:str=None):
        self.max_acc = 0.0
        self.bestModel = None
        self.subset = subset
        self.batchSize = batchSize
        self.numEpochs = numEpochs
        self.model = ASL(subset).to(device=torch.device("cuda" if torch.cuda.is_available() else "cpu"))
        if loadModelName != None:
            self.model.load_state_dict(torch.load(loadModelName, weights_only=True))
        self.lossFunc = torch.nn.CrossEntropyLoss()
        self.optim = torch.optim.AdamW(self.model.parameters(), lr=1e-3, weight_decay=0.01)
        self.scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(self.optim, T_max=self.numEpochs, eta_min=1e-8)
        self.scaler = torch.amp.GradScaler()
        self.savePath = savePath
        self.flow = flow
        if recordGrad or recordLoss or recordModel:
            self.writer = SummaryWriter()

        if recordModel:
            self.writer.add_graph(self.model, torch.stack([Dataloader(Dataset("./MS-ASL/MSASL_train.json", "./Train"))[0][0]]))

        if not os.path.exists(self.savePath):
            os.mkdir(self.savePath)

    def test(self, validation:bool=False, epoch:int=None):
        ...

    def train(self) -> str:
        ...
```

Figure 6. ASLModel class methods and attributes. [1]
Note: Method implementations are removed.

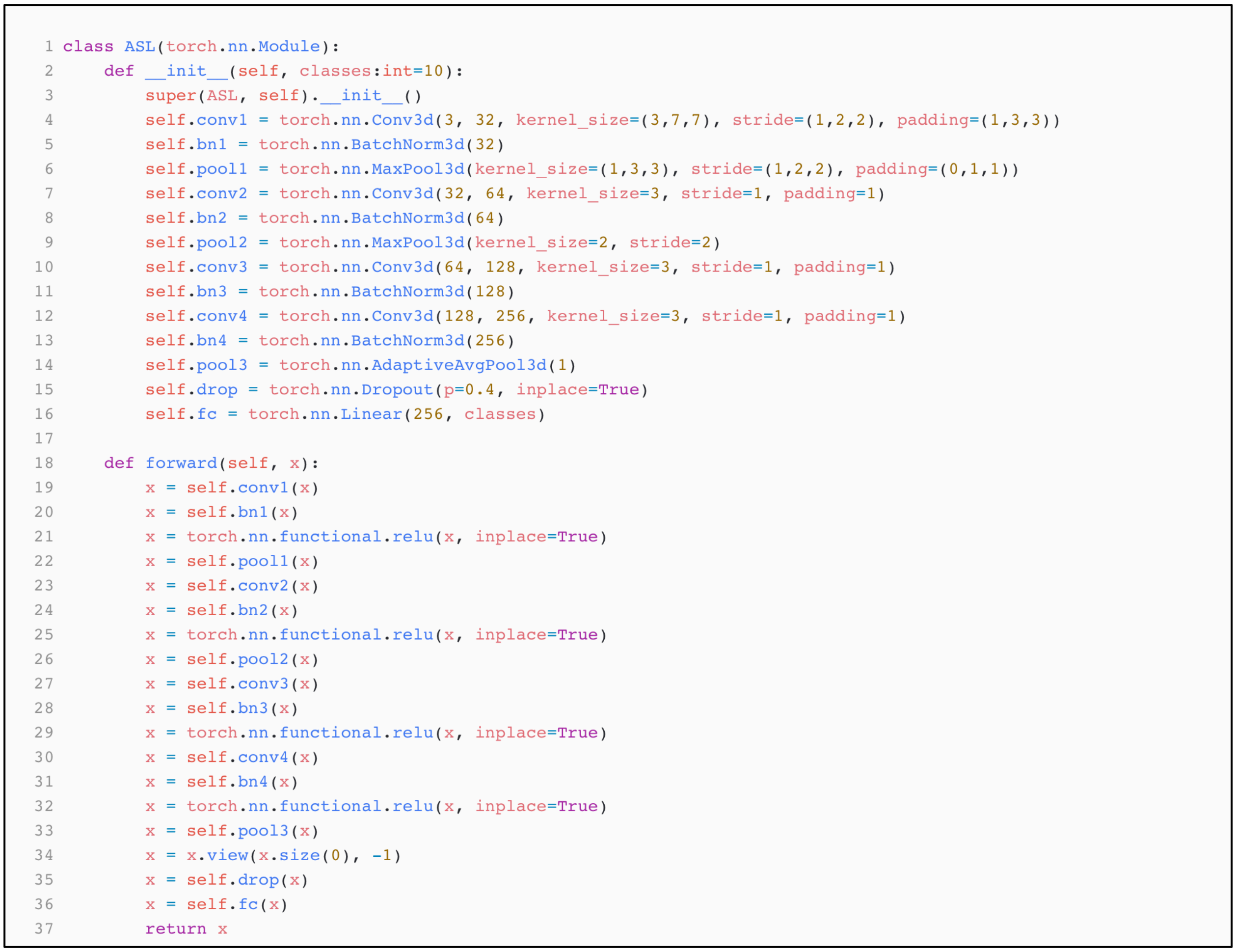
The hardware used for training was an i7-4770 CPU paired with a CUDA capable GTX 1070 GPU. The model began training on the GPU for 500 epochs, with all 1,000 classes being used. However, the hardware showed that it was not powerful enough to train such a large model with a large dataset in a timely matter. It was estimated to take weeks to train the model on the hardware, which did not fit the allotted time frame for this project. A new approach was needed to achieve a functional model in the time frame.

VI. Revised Goal & Shallow CNN

In order to meet the time frame of the project and have a functional model, a new design and training methodology was needed. While the initial model design used a deep convolutional neural network, a new approach of using a shallow, lightweight model seemed like a worthwhile goal. The initial model used 14,823,288 trainable parameters, the shallow model would cut this down to 1,433,768 parameters.

The design of the model followed the first few layers of the initial design in keeping information in the temporal dimension while cutting down on the spatial dimension. To achieve this, the first convolutional layer uses a kernel of 3x7x7 with stride 1,2,2 so that the spatial dimensions are cut in half to reduce computational cost early. It is then followed by a max-pooling layer with kernel 1x3x3 and stride 1,2,2, again halving the spatial dimension. Subsequent convolutional and pooling layer use stride 1,1,1 as the computational cost is already effectively reduced and not many computations are needed. The model only consists of four convolutional layers and 3 pooling layers, cutting back from the twenty convolutional layers and five pooling layers of the initial I3D model. Again, an average pooling layer reduces the dimensions down to a flattened tensor for input into a fully connected layer for classification. Convolutional layers are also followed by the ReLU activation function once again.

To promote regularization on such a small model, batch normalization layers between convolutions were introduced, as well as a dropout layer before the fully connected layer with a probability of 40%, which causes the network to not rely heavily on specific neurons [4]. This, along with reduced amounts of



```
class ASL(torch.nn.Module):
    def __init__(self, classes:int=10):
        super(ASL, self).__init__()
        self.conv1 = torch.nn.Conv3d(3, 32, kernel_size=(3,7,7), stride=(1,2,2), padding=(1,3,3))
        self.bn1 = torch.nn.BatchNorm3d(32)
        self.pool1 = torch.nn.MaxPool3d(kernel_size=(1,3,3), stride=(1,2,2), padding=(0,1,1))
        self.conv2 = torch.nn.Conv3d(32, 64, kernel_size=3, stride=1, padding=1)
        self.bn2 = torch.nn.BatchNorm3d(64)
        self.pool2 = torch.nn.MaxPool3d(kernel_size=2, stride=2)
        self.conv3 = torch.nn.Conv3d(64, 128, kernel_size=3, stride=1, padding=1)
        self.bn3 = torch.nn.BatchNorm3d(128)
        self.conv4 = torch.nn.Conv3d(128, 256, kernel_size=3, stride=1, padding=1)
        self.bn4 = torch.nn.BatchNorm3d(256)
        self.pool3 = torch.nn.AdaptiveAvgPool3d(1)
        self.drop = torch.nn.Dropout(p=0.4, inplace=True)
        self.fc = torch.nn.Linear(256, classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = torch.nn.functional.relu(x, inplace=True)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x = torch.nn.functional.relu(x, inplace=True)
        x = self.pool2(x)
        x = self.conv3(x)
        x = self.bn3(x)
        x = torch.nn.functional.relu(x, inplace=True)
        x = self.conv4(x)
        x = self.bn4(x)
        x = torch.nn.functional.relu(x, inplace=True)
        x = self.pool3(x)
        x = x.view(x.size(0), -1)
        x = self.drop(x)
        x = self.fc(x)
        return x
```

Figure 7. Revised 3D-CNN Model implementation showing both constructor and forward method. [1]

feature maps, significantly reduced computational cost and time using the model.

For training, mixed floating-point precision on the CUDA GPU was introduced. This allowed the model to use both `float32` and `float16` values to train the model, reducing memory requirements and training time without negatively affecting model performance [4]. To achieve this, a new attribute was added to the `ASLModel` class called `scaler`. This attribute contained the PyTorch mixed precision gradient scaler. The training logic also used the `torch.amp.autocast()` wrapper method to enable the model to run in mixed precision mode [4].

Finally, a subset of 10 gestures was used to train the model. This was to show that it was in fact possibly to train such a shallow model on the data and receive presentable results. The subset included the gestures for the following words: hello, nice, teacher, eat, no, happy, like, orange, want, and deaf. This subset of the dataset contained 328 training samples, 108 validation samples, and 84 testing samples. The model was trained for a total of 500 epochs, taking about 72 hours of training time. The model architecture map, the training loss/accuracy, and the validation loss/accuracy were recorded during this time using the TensorBoard library [11]. Best performing models on the validation set were saved to the disk for testing use. Training loss decreased to near 0, while validation loss converged on a value of 1.5 (compared to around 2.3 starting) (see Figure 8). Training accuracy increased to near 100%, while validation accuracy converged around 70% accuracy, with the best model achieving an accuracy of 71.429% (see Figure 9).

VII. Model Performance

To test the model, the same method used for validation was used in the `ASLModel` class with the `validation` parameter set to `False`. This modified the method to not record accuracy or loss through

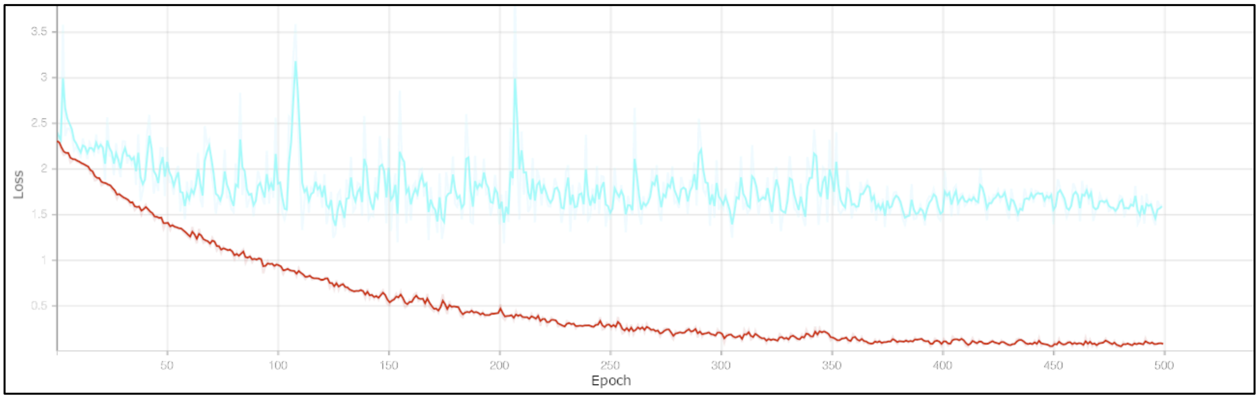


Figure 8. Loss per Epoch. Red: Training loss. Blue: Validation loss. [11].

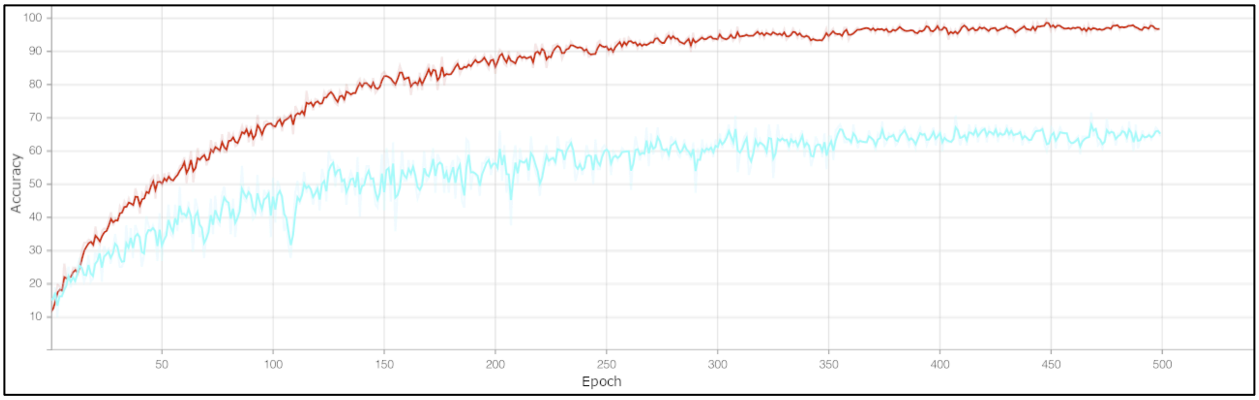


Figure 9. Accuracy per Epoch. Red: Training accuracy. Blue: Validation accuracy. [11].

TensorBoard, but instead use the MatplotLib Python library to construct a confusion matrix [12]. Accuracy and loss was tracked through the command-line itself, as only one final value needed to be recorded. The model tested was saved during training as `2025-04-12_11-06-26_10_71-429_flow.ASL`. The naming convention was chosen for easy telling of what the model is capable of. The first three numbers are the date it was produced, April 12th, 2025. The next three numbers are the 24-hour time of when the model was produced in Eastern Time, 11:06:26. The next number signifies the subset that the model is capable of classifying, which is 10 labels as previously mentioned. The next two numbers is the validation accuracy which should be an excellent measure of its performance on new, unseen data. Lastly, `flow` signifies the model expects optical flow input. All models were saved with the `.ASL` extension showing they are made for the `ASL` model class.

The model above was loaded using the `loadModelName` attribute in the `ASLModel` class. The model achieved excellent performance on the testing set, with an accuracy of 72.840% and an average loss of 1.570432. This shows that the model had in fact learned useful features from the training videos to be able to correctly classify new videos to their corresponding labels. The model correctly generalized to videos not in the training data with only some error in understanding the gestures.

To learn which labels the model would underperform on, the confusion matrix was reviewed. It displays the predicted labels from the model vs the actual label. Values on the diagonal of the graph show correct predictions, with a clear trend shown on the confusion matrix (see Figure 10). The labels *nice*, *teacher*, and *eat* showed excellent performance, correctly predicted those labels 100% of the time. Labels *no, happy,* and *like* also showed excellent performance with a few incorrect predicted. *Want*, *deaf,* and *hello* showed much more error, only receiving correct predictions around half of the time. This is likely due to confusion in gestures, limited awareness of fine motor features in the videos due to optical flow blurring motions together, or limited training information. The latter was especially true for the *hello* class, as it had only 19 training videos compared to an average of 30 for the other labels in the subset. The overall worst performer was the label *orange* with predictions being for *eat* 50% of the time. This is likely due to the fact that both *orange* and *eat* are gestured very similarly in ASL with more fine motions being used to differentiate. Due to the model lacking the capacity to capture fine motion details predictions were often incorrect.

VII. Conclusion

While the initial goal of this project aimed to provide a complex 3D convolutional neural network to improve accessibility for American Sign Language gesture recognition, the results of the revised goal proved valuable. The shallow and lightweight model proves that significant motion features can be extracted from the videos to correctly identify ASL gestures. Although this project faced challenges surrounding lack of computational resources for effective training, it can pave the way for more complex models, better feature extraction, and larger vocabularies to improve access to software that facilitates communication between spoken language and hand gestures.

During the process, significant emphasis was placed on the dataset preprocessing and preparation. Optical flow calculations were used to extract motion data from the videos for effective training. However, this did lead to fine motion details being lost. Along with limited samples in the dataset, the model had to learn features with these constraints. Future projects can focus on both better data preprocessing to extract more information from the samples as well as try and expand the dataset to enable more variation of samples. This, combined with using the full 1000 class dataset, will enable a model to achieve excellent accuracy on a large vocabulary of gestures.

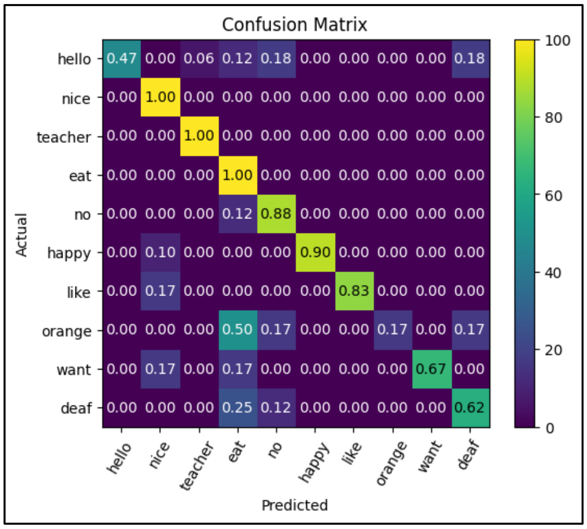


Figure 11. Confusion matrix of testing dataset. [12].

Although the shallow and lightweight model can extract enough information to make accurate predictions, increasing the complexity of the model will be necessary for finer motion details and increased label count. Advancements in computation, especially when it comes to neural network processors, will enable a model with many more parameters to still effectively train without taking an inordinate amount of time. Along with this, inference time on the model can be reduced on devices with limited computational power (such as mobile devices) to be used in real-time applications. Such applications would allow for practical, everyday use for accessibility.

References

[1] A. Carter, *ASL Translator: Development of a 3D-CNN for ASL Gesture Recognition.* GitHub repository, [Online]. Available: https://github.com/stubbornmarlin3/ASL_translator.

[2] J. Carreira and A. Zisserman, "Quo Vadis, Action Recognition? A New Model and the Kinetics Dataset," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Honolulu, HI, USA, 2017, pp. 4724–4733, doi: 10.1109/CVPR.2017.502.

[3] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Boston, MA, USA, 2015, pp. 1–9, doi: 10.1109/CVPR.2015.7298594.

[4] J. Ansel *et al.*, "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation," in *Proc. 29th ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems, Vol. 2 (ASPLOS '24)*, Apr. 2024. [Online]. Available: https://doi.org/10.1145/3620665.3640366

[5] H. R. Vaezi Joze and O. Koller, "MS-ASL: A large-scale data set and benchmark for understanding American Sign Language," in *Proc. British Machine Vision Conference (BMVC)*, Sept. 2019.

[6] yt-dlp contributors, "yt-dlp: A feature-rich command-line audio/video downloader," GitHub repository, [Online]. Available: https://github.com/yt-dlp/yt-dlp.

[7] K. Kroening, *ffmpeg-python: Python bindings for FFmpeg with complex filtering support*. GitHub. [Online]. Available: https://github.com/kkroening/ffmpeg-python.

[8] G. Bradski, "Optical Flow," *OpenCV Documentation*, Open Source Computer Vision Library, [Online]. Available: https://docs.opencv.org/3.4/d4/dee/tutorial_optical_flow.html.

[9] PyTorch Contributors, *TorchCodec: A Media Decoding Library for PyTorch*, GitHub repository, [Online]. Available: https://github.com/pytorch/torchcodec.

[10] TorchVision maintainers and contributors, *TorchVision: PyTorch's Computer Vision Library*. GitHub repository. [Online]. Available: https://github.com/pytorch/vision.

[11] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: https://www.tensorflow.org. [Accessed: Jan. 31, 2025].

[12] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 90–95, 2007. [Online]. Available: https://doi.org/10.1109/MCSE.2007.55