LEVERAGING AUTOMATED MACHINE LEARNING, EDGE COMPUTING FOR VIDEO UNDERSTANDING

A Thesis

by

ZAID PERVAIZ BHAT

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Xia “Ben” Hu
Co-Chair of Committee,	James Caverlee
Committee Member,	Xiaoning Qian
Head of Department,	Scott Schaefer

December 2022

Major Subject: Computer Science

ABSTRACT

Computer Vision is witnessing unprecedented growth over the past few years mainly because of the applications of deep learning methods to computer vision tasks like classification, action recognition, segmentation, and object detection. Video-based action recognition is an important task for video understanding with broad applications in security and behavior analysis. However, developing an effective action recognition solution often requires extensive engineering efforts in building and testing different combinations of the modules and optimizing for the best set of their hyperparameters. The recent advancements in computer vision has shown its vast applicability across several real-world problems. However, developing an optimal end-end machine learning pipeline requires considerable knowledge in computer vision and significant engineering efforts by the developers. To address these problems, in this paper, we present AutoVideo, an AutoML framework for automated video action recognition. AutoVideo aims to tackle these problems by 1) being a highly modular and extendable infrastructure following the standard pipeline language, 2) having an exhaustive list of primitives for pipeline construction, 3) including data-driven tuners to save the efforts of pipeline tuning, and 4) integrating an easy-to-use Graphical User Interface (GUI).

Another major problem with computer vision applications is the deployment of these machine learning models to edge devices for real world applications, especially because these usually require low latency, low power or data privacy. This requires significant research and engineering efforts due to the computational and memory limitations of edge devices. To tackle this problem, we also present BED, an object detection system for edge devices practiced on the MAX78000 DNN accelerator. To demonstrate real world applicability, we integrate on-device

DNN inference with a camera and a screen for image acquisition and output exhibition respectively.

AutoVideo is released at GitHub - AutoVideo-GitHub under MIT license with a demo video hosted at Demo Video-AutoVideo while BED is released at Github - BED_main-GitHub with a demo video at Demo Video-BED.

ACKNOWLEDGMENTS

There are several people I would like to acknowledge for this thesis work who not only opened the door of Artificial Intelligence to me but also made this 2-year Masters program at TAMU a memorable experience, one that I will always cherish.

I would first like to truly appreciate my advisor, Dr. Xia (Ben) Hu, for his support, guidance, and encouragement throughout the course of my Masters. He believed in me and gave me the opportunity to work on novel research at TAMU to build on ideas focussed on making a difference. His insightful discussions in the weekly and bi-weekly lab meetings have had a huge impact on the way I view research and development as the discussions motivate me to discover and tackle novel problems.

I would also like to thank my committee members, Dr. James Caverlee and Dr. Xiaoning Qian for their guidance and valuable comments on the research.

Furthermore, I would like to thank all members of the DATA lab, especially Daochen Zha, Guanchu Wang, Yi-Wei, Henry Lai, and Zhimeng Zhang Jiang. It was a wonderful experience and so much fun working with such a great group of people. The DATA Lab at TAMU maintained a healthy, inclusive and friendly environment. Finally, my deepest gratitude goes to my family for their support all through these years.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supervised by a thesis committee consisting of Dr. Xia Hu(advisor) and James Caverlee of the Department of Computer Science and Engineering and Professor Xiaoning Qian of the Department of Electrical and Computer Engineering. All work for the thesis was completed independently by the student.

Funding Sources

This work was, in part, supported by NSF IIS-1849085 and IIS-1900990 under the title of Research Assistant under the supervision of Dr. Xia (Ben) Hu. The views, opinions, and/or findings expressed are those of the author(s).

NOMENCLATURE

DNN	Deep Neural Network
CNN	Convolutional Neural Network
AutoML	Automated Machine Learning
BED	O**B**ject detection for **E**dge **D**evices

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

1. INTRODUCTION[1]

In recent years, Machine Learning has achieved a lot of success in a broad range of real-world applications ranging from Marketing, Finance, Healthcare[2], Security, Manufacturing among many other use-cases. Computer vision has gained a lot of attention in recent years and is driving groundbreaking research through video understanding[22] in areas such as Security, Healthcare, Transportation and Behavior Analysis.

Action recognition is one of the most important tasks in video understanding[22][10][3]. Given a video clip, it aims to identify the human actions performed in the video, such as brush hair, cartwheel, catch, chew, clap, climb, etc. It has broad applications, such as security, healthcare and behavior analysis[2].

Deep learning has achieved promising performance in action recognition. Although there are a plethora of libraries providing easy-to-use implementations of state-of-the-art machine learning algorithms and techniques.

Libraries such as Scikit-learn [1], torchvision [2], and frameworks like TensorFlow [4], PyTorch [5] are extensively used to develop deep learning solutions[29]. However, generating an optimal deep learning pipeline[27] which can handle the end-end system takes a lot more engineering effort than using these out-of-box library implementations of algorithms which requires significant amounts of skills and experience.

A typical machine learning workflow involves several repeated iterations of data modeling, feature selection, model development, benchmarking, hyper-parameter tuning, etc. Developing a deep learning solution therefore heavily relies on human efforts.

[1] Part of the data reported in this chapter is reprinted with permission from “Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence Demo Track”. Pages 5952-5955. IJCAI-ECAI 2022.

First, we often need a very complex training pipeline, including but not limited to data loading, frame extraction, video cropping/scaling, video augmentation, model training, etc., which requires huge engineering efforts.

Second, to achieve a good performance, a practitioner often needs extensive laborious trials on different combinations of the modules and their hyperparameters[54]. Moreover, for every change in data distribution/model requirements, this process needs to be reiterated which takes a lot of human engineering efforts.

These challenges led researchers to develop new frameworks based on automated machine learning (AutoML) systems[67][63][62][56]. These aim to address this repetitive and laborious nature of work and reduce the entry barriers to developers with minimal machine learning experience[55].

To bridge this gap, in this thesis, we present an automated video action recognition system named AutoVideo, which has several desirable features.

First, it is highly modular and extendable. It is designed using the standard pipeline language under D3M infrastructure, which defines primitives as basic building blocks and uses Directed Acyclic Graph~(DAG) to describe pipelines. The inputs, the outputs, and the hyperparameters of each module are well formatted so that we can easily develop and integrate new modules into the system with minimum engineering costs.

Second, it supports an exhaustive list of primitives for pipeline construction. Specifically, we have so far supported 193 primitives from video processing to recognition models. Users can create various training pipelines via combining these primitives in different ways.

Third, it provides data-driven tuners to save the efforts of pipeline tuning. We have implemented two commonly used AutoML tuners [63][61][56], including random search and

Hyperopt. They can automatically explore different primitive combinations and tune the hyperparameters to identify the best pipeline design.

Fourth, it provides a Graphical User Interface~(GUI). In the GUI, users can manually construct pipelines in a drag-and-drop fashion, launch the training or search, and monitor the training progress[55].

While there are some other open-sourced video understanding libraries out there, they only provide a model zoo for users. In contrast, we provide an exhaustive list of primitives to allow users to create various pipelines or search with AutoML tuners[67]. Unlike some other research-oriented AutoML studies [62][56] for videos, we target an easy-to-use and open-sourced package with unified interfaces for users. Thus, AutoVideo complements the existing efforts[59].

Another major task in video understanding is Object Detection. Be it keeping an eye on pedestrians and motorists in urban areas, people detection and tracking in video surveillance systems or robot vision, object detection finds applications everywhere. With the explosive growth of Internet of Things (IoT) technologies, deploying and directly running machine learning (ML) models on edge devices[45] have many important applications [Murshed et al., 2021].

In contrast to the traditional cloud-based services[36] that transmit the data to the cloud for computation, edge computing[28][26] brings computation closer to the source of the data. As such, it enjoys several benefits, such as low latency, low power, and data privacy. However, it is quite challenging to deploy ML models, especially deep neural networks (DNNs), to edge devices with limited computational power and memory.

First, ML models rely on massive computational resources [Devlinet al., 2018; Dosovitskiy et al., 2020; Redmon and Farhadi, 2018], which may not be available on the edge devices. Second, due to the limited capacity of edge devices[26], it is necessary to compress the model while maintaining the performance, which requires non-trivial research and engineering efforts.

In this thesis, we also provide an efficient setup to deploy ML models on edge devices for real-time object detection, which has broad applications, such as surveillance, human computer interaction, and robotics [Pathak et al., 2018]. In particular, we present oBject detection system for Edge Devices (BED)[58], an end-to-end system which integrates a DNN practiced on MAX78000 with I/O devices, as illustrated in Figure 1.

Specifically, the DNN model for the detection is deployed on MAX78000, an efficient and low-power DNN accelerator; the I/O devices include a camera and a screen for image acquisition and output exhibition, respectively. Experiment results demonstrate BED can provide accurate object detection with a 300KB tiny DNN model. In the live and interactive part of our demo, we will showcase BED for real-time object detection.

The overall goal of this framework is to provide easy to use solutions for video understanding for two types of users - the basic users and the machine learning experts. The basic users, who could use this system to get familiar with video understanding tasks and generate end-end well performing solutions for their specific data and use case. The machine learning experts, for whom this framework could provide help to automate and simplify parts of the workflow in their research and development life cycle.

Consider an example where a machine learning engineer has to develop an action recognition system to monitor elderly behavior in a hospital. It is hard to identify before-hand the

methods that would be most suitable for this task. The engineer would have to spend hours to build the end-end pipeline, benchmark results on the dataset and later tune for the best set of hyperparameters. Moreover, different datasets may have different data distributions and the results from one dataset might not scale to other datasets. This would require him to repeat the entire process for each of the datasets.

Our framework could help by providing an end-end system to establish a decent baseline to start the further development process. This initial baseline without the need for extensive efforts could give the engineer a decent starting point to build on the research and development process.

Additionally, since our framework supports end-end pipeline search along with hyperparameter tuning, this could really speed up the process of obtaining the optimal set of parameters. Therefore, the benefits of our framework are two fold.

Firstly, it will help him improve the predictive performance of the model and provide good starting points for speeding up the development.

Secondly, it will significantly reduce the workload required from model building to hyperparameter tuning which is all done in an automated fashion.

1.1 Objectives

The primary objective of this thesis is to provide a framework for automating the process of generating machine learning pipelines for video understanding tasks like video action recognition, object detection, etc. Our framework has the following advantages over other open-source video-understanding tools:

High Modularity and Extendability

AutoVideo in specific has been designed to be highly modular and extendable as our infrastructure follows standard pipeline language which makes it easier to modularise the package and extend the package to new, improved state-of-the-art techniques.

Good baseline performance

Since accurate prediction in video understanding tasks is very significant, specially in areas such as healthcare and security, obtaining benchmark results with even small improvements could make a huge difference.

Our framework supports benchmarking results on state-of-the-art datasets to demonstrate its effectiveness.

Exhaustive list of primitives supported

We support an exhaustive list of primitives for pipeline construction which includes 5 Data Processing primitives, 1 Video Processing primitive, 3 Video Transformation primitives, 175 Video Augmentation primitives and 9 Video Action Recognition primitives.

Automated pipeline search and hyperparameter tuning

Our framework supports data-driven tuners to help with automated pipeline search and hyperparameter tuning which aims to save the efforts of end-users and developers by several folds.

User friendly Design

AutoVideo and BED both support an easy-to-use Graphical User Interface(GUI) which is both simple and intuitive to use. The user does not need to spend a lot of time to learn our API to be able to use the GUI. The user can literally start building pipelines and run code with only a few lines of code or a few drag and drops and a few clicks.

Additionally, a user can extend their prior domain knowledge and experience to our framework without any significant overhead.

Support for Edge Devices

Through BED, we introduce a framework that lets us easily develop Deep learning solutions[27] for edge devices and deploy the same on AI powered microcontroller chips. For the purpose of this demonstration we have chosen the MAX78000 microcontroller chip.

Real-Time Detection

Through BED, we also support a framework for real-time detection of objects by integrating our framework with Camera and LCD screen for real time image acquisition, on-chip inference and display.

1.2 Thesis Outline

In this thesis, we plan to overcome the challenges mentioned above by developing an open-source framework for video understanding tasks that supports automated model selection and hyperparameter tuning without compromising much on accuracy. The thesis is organized in the following chapters:

Chapter 1 focuses on introducing the high level idea of the thesis and the overall organization of the thesis in various chapters.

Chapter 2 focusses on Literature review and background research for various state-of-the-art object detection systems, video action recognition systems, open-source packages providing support for end-end video understanding tasks.

Chapter 3 focuses on developing a framework for video action recognition which can predict and visualize results in a form of an easy to use GUI called AutoVideo.

Chapter 4 focuses on experimental results of the AutoVideo framework from Chapter 3.

Chapter 5 focuses on the live and interactive parts of the AutoVideo framework in the form of an easy-to-use GUI.

Chapter 6 focuses on the Conclusion for the AutoVideo framework.

Chapter 7 focuses on developing an object detection system called BED that can be deployed in highly constrained environments like edge devices where strict constraints are placed on memory, efficiency, inference speed[43], etc.

Chapter 8 focuses on Experiments for the BED framework.

Chapter 9 focuses on the quantization and synthesis techniques used before deploying our BED framework on the edge devices.

Chapter 10 focuses on the Evaluation and Demonstration of the BED framework on the edge device using on-board inference using live camera stream.

Chapter 11 focuses on the Live and interactive parts for the BED framework

Chapter 12 focuses on the Conclusions drawn for the BED framework

Chapter 13 offers discussion and future work.

2. RELATED WORK

Over the years, various attempts have been made to automate the complete video understanding pipeline or parts of it. Most of the previous works have focussed mainly on providing a model zoo for users. In the following sections, we would briefly explain the previous works related to automating machine learning systems in general and video understanding tasks in specific.

2.1. Model and Hyperparameter Optimization

The intent of automated model and hyperparameter search methods is to tune the model with the set of hyper-parameters to build the end-end machine learning pipeline in an automated fashion. Several optimization methods such as Bayesian Optimization, HyperOpt Search[18], Random Search, etc. are used for this purpose. Based on the WEKA [8] models, Auto-WEKA [16] is one of the first AutoML platforms[67] [63]. It uses SMAC [10] (a variant of Bayesian Optimization) for optimizing the hyper-parameters.

AutoKeras [15][12] uses auto-sklearn [7] to tune the entire pipeline of preprocessors and models using conditional hyperparameter spaces using Scikit-learn models along with SMAC based optimization.

2.2. Neural Architecture Search

With the overwhelming demand for deep learning and its success in various domains, a lot of research efforts have focused on automating the search and design of the neural network structures. NAS [16], NASNet [17], ENAS [13], DARTS [11], AutoKeras [15] are a few examples of using AutoML[62] for deep learning[27].

Since video recognition tasks have huge computational cost and low interpretability due to the deep learning architectures used, we do not explore NAS based techniques for neural

architecture search. However, we focus on automating our framework using AutoML by supporting end-end automated pipeline search and hyperparameter tuning.

2.3. Video Action Recognition

With the unprecedented growth in video understanding and streaming, a number of challenges on how to perform video understanding on limited computation resources with high accuracy. This gave rise to hardware efficient video understanding research which was a breakthrough step towards real-world deployment of these video understanding systems, both on the edge and the cloud[36]. Thousands of hours of videos are being uploaded to YouTube everyday. Such industry applications require video understanding to be both accurate and efficient.

Deep Learning has achieved tremendous success in video understanding and has become the standard for most of the video understanding tasks [13]. One of the most important tasks in video understanding is to understand human actions. The main contribution for its popularity is its applications in the industry in areas like security, behavior analysis, video retrieval, gaming, and entertainment.

Human action understanding involves recognizing, localizing, and predicting human behaviors and this task is called video action recognition. Video-based action recognition has been an important area of research and has drawn a lot of attention from the academic community [1], [2], [3], [4], [5].

One of the earliest attempts to apply CNNs[44] to video understanding tasks was made in DeepVideo. Since video understanding is different from image understanding[37][35] as videos are temporal in nature and require some form of temporal modeling, the first major trend in video recognition was started by the paper on Two-Stream Network. This adds a second path to

model temporal information in a video which is achieved by training a CNN on the optical flow stream. This inspired a number of papers such as TDD, LRCN [37], Fusion [50], TSN [53], etc. Another major trend in video recognition was the use of 3D Convolutional kernels for temporal modeling and this also inspired a series of works such as I3D [6], R3D [4], S3D [239], Non-local, SlowFast [45], etc. Very recently, another trend emerged which docussed on computational efficiency so that the training could be scaled to larger datasets for real world adaptation. This includes work such as Hidden TSN [53][5], TSM[13], X3D [44], etc.

2.4. Object Detection

Though Object detection is a trivial task for humans and even a few months old child can start recognizing commonly seen objects, it is equally harder for a computer to perform object detection. The task involves both identifying and localizing all objects in an environment. The earliest approaches used a brute force technique to apply image classification[24] on sliding windows of variable sizes to find and localize objects. This was later replaced with a technique involving region proposals, where a separate algorithm only outputs certain region proposals and classifies these region proposals instead of applying classification on an exhaustive list of sliding window boxes. Deep learning has been getting a lot of attention and a lot of research is focussing on improving these deep learning techniques. This has led to a series of work focussing on improving object detection techniques.

Object detection algorithms based on deep learning are mainly classified into two categories. These include two-stage object detectors and one-stage object detectors. One-stage detection algorithms work by first generating several candidate regions which are then classified for their object categories. Works like YOLO [48][31] and SSD [27] work with feature pyramid networks (FPNs) [41] as a backbone to help the network detect and classify objects at different

scales in a single forward pass through the network.

On the other hand, two stage object detection algorithms involve segregating the task of object localization from object detection. They achieve this by having region proposal networks generate possible regions in which the likelihood of an object being present is maximum. After such region proposals have been identified, the network learns to classify these to their correct classes. Techniques like R-CNN, Fast R-CNN and Faster R-CNN have gained popularity in the object detection community.

3. AUTOVIDEO FRAMEWORK[2]

In this section, we present the AutoVideo framework for automated video action recognition to explain in detail and satisfy the objectives mentioned below: (1) High Modularity and Extendability (2) Good baseline performance (3) Exhaustive list of primitives supported (4) Automated pipeline search and hyperparameter tuning and (5) User friendly Design[59].

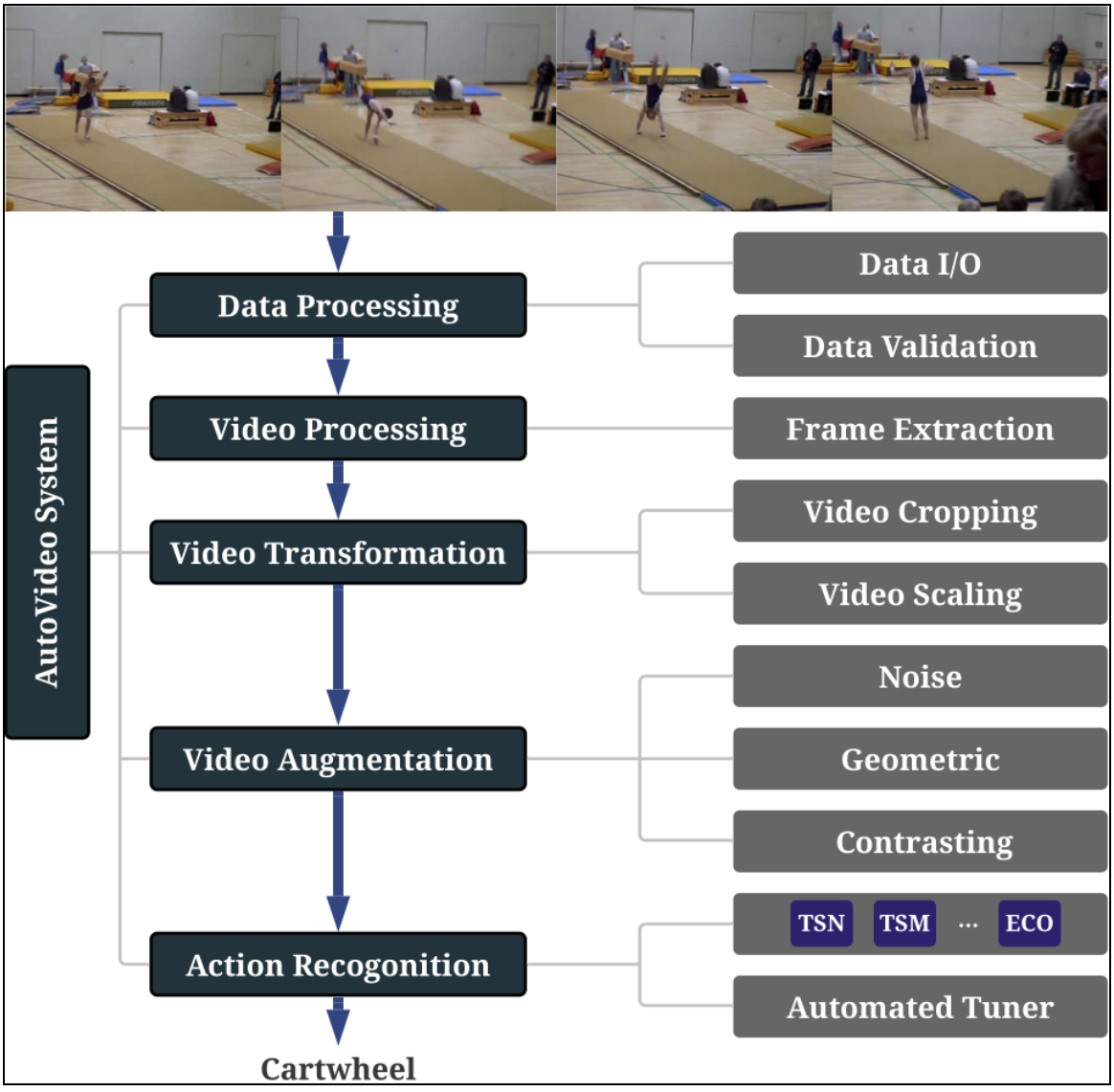

Figure 1. AutoVideo implements a complete training pipeline from data processing to action recognition, where each pipeline step can be instantiated with customized choices of primitives[59].

[2] Part of the data reported in this chapter is reprinted with permission from "Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence Demo Track". Pages 5952-5955. IJCAI-ECAI 2022.

3.1 AutoVideo System

Figure 1 shows an overview of the general architecture followed by our AutoVideo system. Similar to most of the video understanding pipelines, it implements a complete training pipeline consisting of data processing, video processing, video transformation, video augmentation, and action recognition, where each of the pipeline steps can be instantiated with customized choices of primitives, which leads to a large number of possible pipelines. First, during the data processing phase, the system takes in the dataset from the user and preprocesses the dataset so that the data can be ingested by the various downstream steps. Next, during the video processing, video transformation and video augmentation phase, the system reads the upstream data in the form of videos and performs the necessary transformations and augmentations at the video level. The final stage in the training process is using various action recognition algorithms supported by our package to train the network end-end. Then two automated tuners are provided to automatically discover the best pipelines based on the validation performance. Finally, during the test phase, the system uses the best pipeline searched by the tuners to run inference on the pre-trained weights stored by the training phase.

This section first introduces the standard pipeline language and then elaborates on our programming interface and GUI.

3.2 Primitives and Pipelines

We build AutoVideo on D3M infrastructure, which provides generic and extendable descriptions for modules and pipelines. The interface can accommodate various modules.

Here, we provide an overview of the basic concepts. More details of the pipeline language can be found in[59].

3.2.1. Primitive

It is the basic build block. It is an implementation of a function with some hyperparameters. A pipeline is a Directed Acyclic Graph (DAG) consisting of several primitive steps. Data types, such as DataFrame and NumPy Ndarrays, can be passed between steps in a pipeline. Each of the above concepts is associated with metadata to describe parameters, hyper-parameters, etc. Following this pipeline language, we wrap each component in AutoVideo (e.g., an action recognition algorithm or an augmentation module) as a primitive with some associated hyperparameters. A complete list of all the primitives supported by our package are summarized in Table 1 primitives

Module	Number	Examples
Data Processing	5	DatasetToDataFrame
Video Processing	1	FrameExtraction
Video Transformation	3	GroupScale, GroupCenterCrop
Video Augmentation	170	AdditiveGaussianNoise, Rotate
Action Recognition	9	TSN, TSM, C3D, ECO
Total	188	-

Table 1: The number of primitives implemented in each module of AutoVideo with some example primitives. Source: [59]

3.2.2. Data Processing

Data processing is designed for reading videos and the labels. We design a standard format to represent the datasets which gives users the flexibility to use our package on their custom datasets. This helps users to read the dataset consisting of videos and labels.

3.2.3. Video Processing

An important step in the video understanding pipeline is converting videos into frames that can later be used while training the models without the need to convert in each iteration. Video processing converts videos into frames to be used by the downstream modules.

3.2.4. Video Transformation

Since each video might have different resolution and scale but the network expects fixed sized input frames for video recognition, video transformation transforms the videos into the target size through cropping or scaling.

3.2.5. Video Augmentation

Since video data is very expensive to collect, it is very limited and is therefore very prone to overfitting. To enhance the training performance and prevent overfitting, we have so far supported 193 primitives for video augmentation.Video augmentation augments the videos during the training phase to supplement the training data and thus enhance performance.

3.2.6. Action Recognition

Finally, various action recognition algorithms are wrapped inside the Action recognition package to train deep learning models.

3.3. Pipeline Construction

Various pipelines can be constructed by combining these primitives in different ways. While our current scope focuses on action recognition, the genericity of the pipeline language allows AutoVideo to be easily extended to support other video-related tasks, which is our future work.

3.4. Optimization Techniques

Every AutoML system can be considered a form of optimization problem for finding the best performing setting for the pipeline consisting of the model, hyperparameters, various augmentations that can be applied, etc. In this section, we review and present the various optimization techniques that are used by our framework to support Automated Tuners. Although there are several optimization strategies such as Grid Search, Random Search, Bayesian Optimization, HyperOpt Search, etc, due to the time constraint on the experimentation, we select the following two options for our framework.

3.4.1. Random Search

The random search algorithm defines a search space by creating a bounded domain of hyperparameter values. It then randomly samples points in that domain without replacement and evaluates them on the cross validation set. Though the searching technique is fairly simple, it tends to perform reasonably well in most of the cases within limited time-constraints. To demonstrate the effectiveness of random search, let us consider n independent random samples. The probability of obtaining the top k percentile solution is given as:

$$P_{(top\ k\ solution)} >= 1 - (1-k)^n$$

Considering only 100 trials for an experiment, the probability of finding the top 10% of hyper-parameter settings is 0.9999 and this probability increases at an exponential rate with increase in the number of trials. Since we only need to store the results for the best performing trial after each trail, the space complexity is also *O(1)*.

3.5.2. HyperOpt Search

HyperOpt Search is a serial and parallel search optimization technique that supports searching over awkward search spaces. Since our search spaces can include parameters that are

real-values, discrete or conditional, HyperOpt provides a technique to handle such awkward search spaces. It uses a form of Bayesian Optimization for hyperparameter tuning which allows us to get the best set of parameters for a given model[18].

Bayesian optimization is one of the most popular techniques to find the optimal set of parameters for a given function. The popularity of Bayesian optimization is due to its ability to make smarter choices for searching through the search space based on the history of searched parameters and their performance. It uses a heuristic function to sample a set of hyperparameters based on the history.

In our package, we make use of HyperOpt search which is based on Bayesian Optimization. HyperOpt Search uses the Tree-structured Parzen Estimators algorithm which is a sequential model-based optimization (SMBO) approach.

3.5. Ray Tuner

Tune is an open source Python library that provides extensive tools for hyperparameter tuning at scale[20]. It supports most of the famous machine learning frameworks like PyTorch, XGBoost, TensorFlow and Keras and uses state of the art algorithms like BayesOptSearch, HyperBand/ASHA and integrates with other hyperparameter optimization tools like Optuna, HyperOpt, etc.

The reason for the popularity of Ray is the ability to provide out of the box asynchronous optimization and to scale easily from a single machine setup to multi-machine parallel optimization without having to modify the code. The primary reason for researchers and developers to adopt Ray in their machine learning pipelines are scale and flexibility[20].

3.6 Programming Interface

From the programming view, AutoVideo enables users to easily train any manually designed pipelines and also use automated tuners to help discover the best pipelines. A minimum example of fitting a pipeline is given below.

```
from autovideo import fit, build_pipeline

# Build pipeline based on config
pipeline = build_pipeline({
    "transformation": [("GroupScale", {"scale_size": 224})],
    "augmentation": [
        ("arithmetic_additive_gaussian_noise", {"scale": (0, 0.2*255)}),
        ("geometric_rotate", {"rotate": (-45, 45)}),
        ("contrast_log_contrast", {"gain": (0.6, 1.4)}),
    ],
    "recognition": ("tsn", {"learning_rate": 0.001}),
})

# Fit
fit(train_dataset, train_media_dir, target_index=target_index,
      ↪ pipeline=pipeline)
```

Example 1: Code snippet for fitting a model[59]

Here we can define each variable as below: train_dataset is the DataFrame describing the video file names and labels, train_media_dir contains the directory of the video files, and target_index specifies which column is label. Pipelines are described with a configuration dictionary specifying the primitives and hyperparameters in each pipeline step. Users can customize the configuration in build_pipeline to build pipelines with different combinations of the primitives and the hyperparameters. We can specify the configuration of the build_pipeline function by specifying the list of primitives we want to use for video transformation, video augmentation and finally video recognition primitives from the supported set of primitives in our package. The Fit function will return the fitted pipeline that can be saved for making predictions.

This fitted pipeline stores the trained model along with the trained weights that can later be used for inference to make predictions on the test set.

However, manually tuning the pipeline is laborious and tedious. Thus, we can alternatively use automated tuners. An example is presented below.

```
from ray import tune
from autovideo.searcher import RaySearcher

search_space = {
    "augmentation": {
        "aug_0": tune.choice[
            ("arithmetic_additive_gaussian_noise", {"scale": (0, 0.2*255)}),
            ("arithmetic_additive_laplace_noise", {"scale": (0, 0.2*255)}),
        ],
        "aug_1": tune.choice[
            ("geometric_rotate", {"rotate": (-45, 45)}),
            ("geometric_jigsaw", {"nb_rows": 10, "nb_cols": 10}),
        ],
    "recognition": {
        "algorithm": tune.choice(["tsn", "tsm"]),
        "learning_rate": tune.uniform(0.0001, 0.001),
    },
}

config = {
    "searching_algorithm": "hyperopt",
    "num_samples": 100,
}

searcher = RaySearcher(train_dataset, train_media_dir,
      ↪ valid_dataset=valid_dataset, valid_media_dir=valid_media_dir)

best_config = searcher.search(
    search_space=search_space,
    config=config
)
```

Example 1: Code snippet for automated tuning[59]

For automated tuning, we make use of ray for defining the search space, searching algorithm and searching the best pipeline. Here we define each section of the above code snippet:

The search_space is defined as a dictionary for each of the parts of the pipeline. We define the search_space for augmentation by listing various augmentation primitives we want in the optimal pipeline search along with their respective set of parameters. Similarly we define a list of different types of algorithms we want to choose from for video recognition. We also define a search space for the hyperparameters for these video recognition primitives like learning_rate, epochs, etc.

For searching the best set of hyperparameters, we define a config for the ray tuner which contains the searching algorithm we want to use and the details of the searching process like the number_of_samples we want to draw from the search space before choosing the best set of parameters.

3.7 GUI

In this section, we present our Graphical User Interface (GUI) which is an interactive drag-drop based interface that allows users to easily create pipelines on the go and train, evaluate or search automatically for the best pipeline using a simple interface. This allows users to use our framework without having to interact with the terminal or code[8]. Our GUI is based on the Orange framework[21] and uses PyQt widgets for creating an interactive interface.

3.7.1 Orange

Orange [60] framework allows for creating GUIs for interactive interfaces. It consists of an Orange Canvas which is the visual programming environment provided by Orange. Each Orange Widget is therefore a component in the Orange Canvas[21]. Widgets communicate with each other and pass objects through communication channels to interact with other widgets.

The widgets in Orange[60] each belong to a specific category and have an associated priority with that category. The users can simply choose these widget and drag these on the canvas in the

middle. These widgets can then simply be connected to create a working pipeline. These available widgets are displayed on the left in a toolbox as soon as the Orange Canvas is loaded which can be demonstrated in the below sections

3.7.2 GUI Interface

In this section, we present the GUI interface of the AutoVideo package. This GUI is based on the Orange framework [60] and enables users to develop pipelines on the go without interacting with low level code. Here we present the GUI as under:

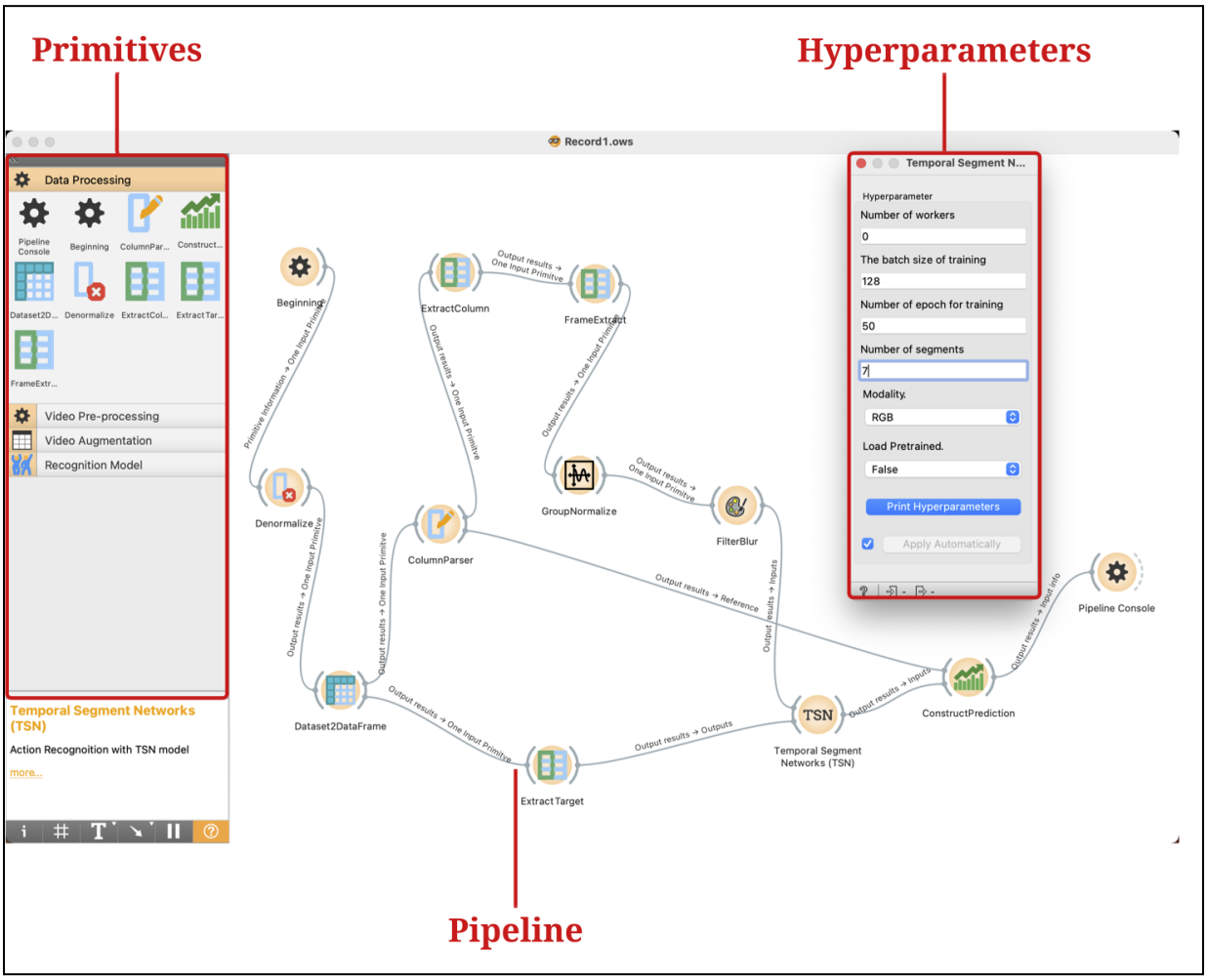

Figure 2: Presents our GUI to help users construct pipelines, fit the constructed pipelines, and make predictions. The GUI is implemented based on Orange[59].

Figure 2 illustrates the canvas for pipeline building. Users can construct a pipeline in a drag-and-drop fashion by dragging icons (primitives in the left) to the canvas and connecting them with lines. For each of the primitives, we can double-click the icon to modify the default hyperparameters of the primitive. Once a pipeline is built, the GUI will convert the constructed pipeline into the configuration dictionary as in Example 1 and the backend will instantiate the pipeline. In addition, users can save a pipeline and load it later just like editing a document in Microsoft Word.

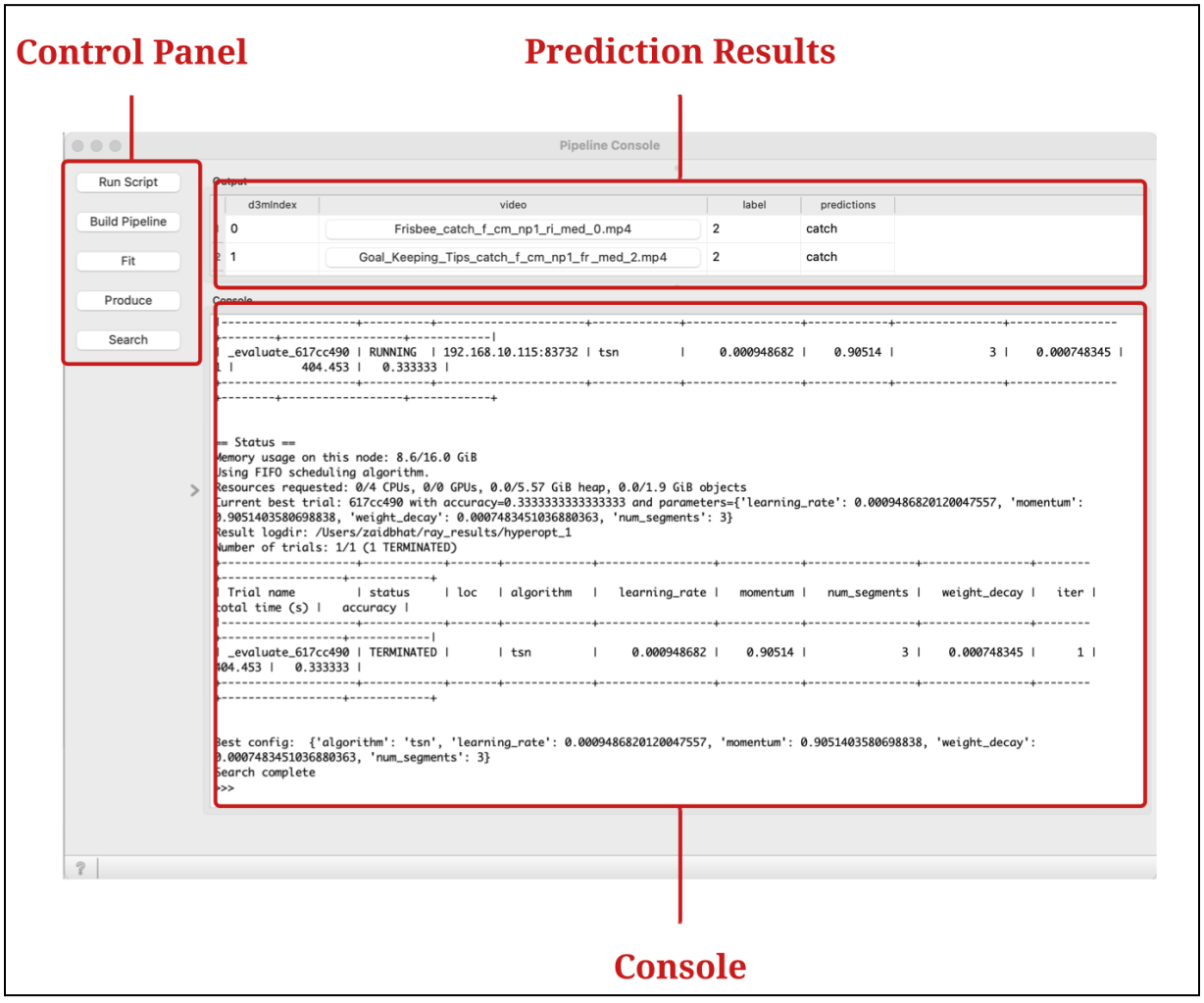

Figure 3: Presents the pipeline console which allows the user to fit a pipeline, save the trained pipeline, load the pipeline and run inference on it, visualize the predictions and also run automated searchers[59]

Figure 3 shows the pop-up windows for pipeline fitting and training progress monitoring, respectively. Specifically, the users can fit the pipeline, make predictions with a fitted pipeline, or perform automated searching by simply clicking a button in the control panel. For example, by clicking the ``fit'' button, the backend will train the model and visualize the training curve once the training is finished. By clicking the ``produce'' button, the backend will make predictions on the testing data. The ``search`` button will launch the automated tuners for pipeline search and save the best pipeline.

The prediction results will be displayed at the top of the window, where the users can watch each of the testing videos and check its predictions and labels. In addition, we also provide an interactive console, where the users can interact with the backend with command lines. This design provides flexibility to meet different needs of the users.

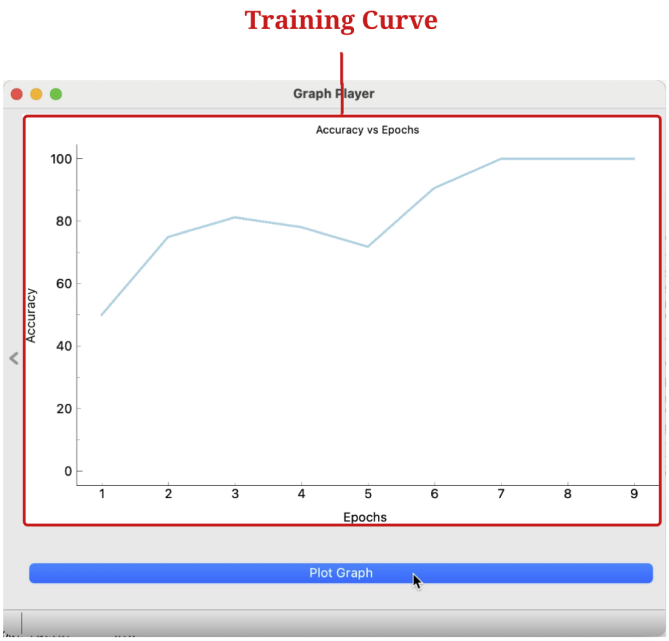

Figure 4: Illustration of training accuracy vs epoch visualized by the GUI[59]

Figure 4 illustrates the training curve to plot the Accuracy v/s Epochs in the training process which will be displayed at the end of the training process. Additionally, the interface also provides the ability to view the video along with the predicted label during the inference phase[43] which can be demonstrated as under:

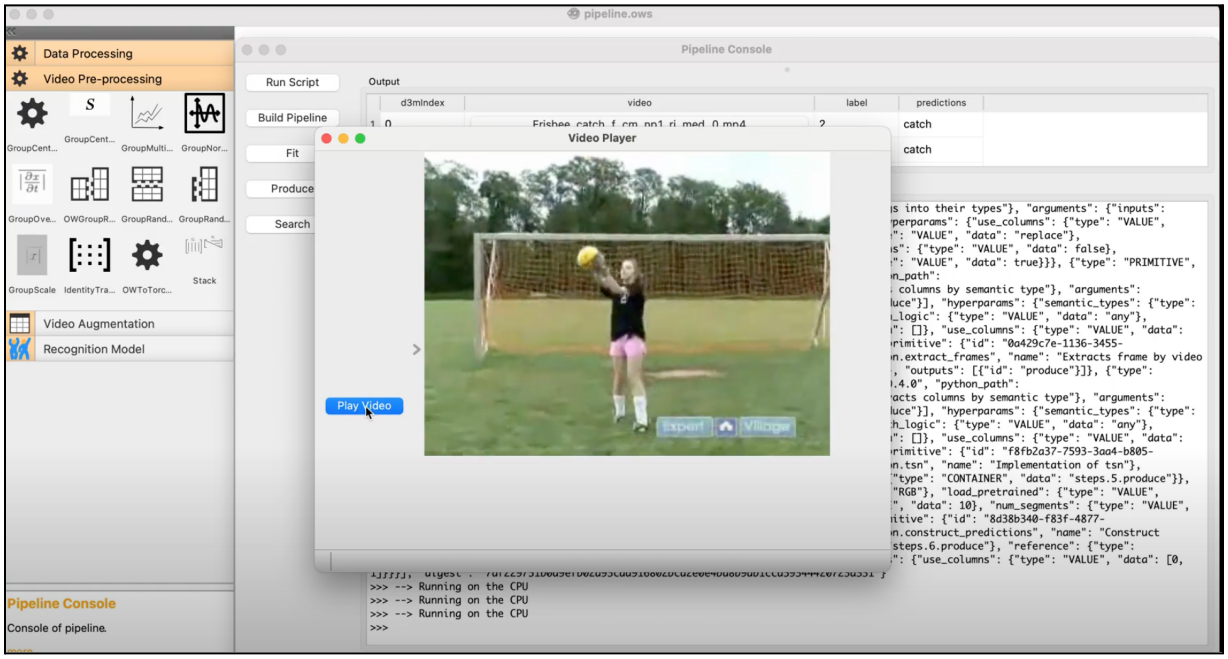

Figure 5: Figure demonstrates the visualization of the videos used in the inference process with predicted labels from the model[59]

4. EXPERIMENTS[3]

We conduct experiments on our framework to answer the following two questions. (1) Does the proposed AutoVideo framework achieve good performance on the state-of-the-art methods? (2) How effective are the automated searchers in finding the best set of hyperparameters?

In this section, we introduce the overall experimental setup including the datasets used for evaluation, algorithms and methods used and other implementation details.

4.1. Datasets

We have considered two benchmark datasets that are used for most of the Video Action Recognition research. These include the following:

4.1.1. HMDB-51

HMDB-51 is a large scale dataset for video understanding tasks and is collected from various sources like movies, public databases, etc. It is mostly collected from movies, and a small proportion from public databases such as the Prelinger archive, YouTube and Google videos. The dataset contains 6849 clips divided into 51 action categories, each containing a minimum of 101 clips.

4.1.2 UCF-101

UCF101 is another large scale action recognition data set of realistic action videos, collected from YouTube. It has 101 action categories and this data set is an extension of UCF50 dataset which has 50 action categories.

UCF101 gives the largest diversity in terms of action with 13320 videos from 101 action categories, UCF101 also presents a large variation in camera motion, object appearance and

[3] Part of the data reported in this chapter is reprinted with permission from "Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence Demo Track". Pages 5952-5955. IJCAI-ECAI 2022.

pose, object scale, viewpoint, cluttered background, illumination conditions, etc. Due to these reasons, it is one of the most challenging datasets to date.

4.2 Experimental Setup

For an AutoML system, the process of benchmarking performance has many challenges. Most of these challenges are also applicable to the general machine learning systems but due to the data driven nature of AutoML systems [61], these approaches are even more prominent.

First challenge to the performance of these systems stems from a significant amount of variance in the performance of the same AutoML system in different trial runs. Since most of the components of the machine learning pipeline such as the optimization techniques, models, automated tuning etc. have some amount of randomness involved, this greatly impacts the overall performance across several runs. To mitigate this problem, we fixed the random seed for all of the other modules used.

Second, since video action recognition systems take a lot of time to train for each experimental setup, this slows the overall development process by several magnitudes. To mitigate this, we developed a small subset of the HMDB51 dataset which we name HMDB6 which contains only 6 action recognition classes from the original HMDB51 inorder to speed up the time for each experimental run.

4.2.1. Evaluation Metrics

For evaluating the performance of the various components in our pipeline, we use the standard evaluation metric i.e, Accuracy which is used in all of the action recognition benchmarks.

4.3 Action Recognition

We use the following baseline models in our framework. We evaluate all models on the same datasets and same experimental settings.

- TSN[53]: Temporal Segment Network (TSN) is a novel framework for video-based action recognition and is based on the idea of long-range temporal structure modeling. It makes use of a sparse temporal sampling strategy combined with video-level supervision to enable efficient and effective learning using the whole action video.[52]
- TSM[13]: Temporal Shift Module (TSM) enjoys both high efficiency and high performance and achieves the performance of 3D CNNs but also maintains the complexity of 2D CNNs. TSM achieves this by shifting part of the channels along the temporal dimension thereby facilitating information exchanged among neighboring frames.
- R2P1D: R2P1D introduces spatiotemporal convolutions for video analysis. It empirically demonstrates the accuracy advantages of 3D CNNs over 2D CNNs within the framework of residual learning. R2P1D block further demonstrates that factorizing the 3D convolutional filters into separate spatial and temporal components yields significant gains in accuracy.
- R3D[4]: R3D proposes 3D CNN based on ResNets toward a better action representation for video tasks. It is based on the idea that convolutional neural networks with spatio-temporal 3D kernels (3D CNNs)[19] have an ability to directly extract spatio-temporal features from videos for action recognition.
- C3D[4]: C3D proposes a simple and effective approach for spatiotemporal feature learning using deep 3D ConvNets trained on a large scale supervised video dataset. C3D

is based on their experimental finding that include 1) 3D ConvNets are more suitable for spatiotemporal feature learning compared to 2D ConvNets; 2) A homogeneous architecture with 3 × 3 × 3 convolution kernels in all layers is among the best choices for best performing architectures for 3D ConvNets.

- ECO[7][14]: ECO introduces a network architecture that takes long-term content into account. It therefore enables fast per-video processing. The main idea behind the architecture is to merge long-term content already in the network rather than in a post-hoc fusion. It combines this with a sampling strategy, which exploits that neighboring frames are largely redundant, this yields high-quality action classification. It supports two variants ECO-Lite and ECO-Full[14].
- I3D[6]: I3D introduces a new Two-Stream Inflated 3D ConvNet (I3D). I3D is based on 2D ConvNet inflation where filters and pooling kernels of very deep image classification[24] ConvNets are expanded into 3D, making it possible to learn seamless spatio-temporal feature extractors from video while leveraging successful ImageNet architecture designs and even their parameters.
- STGCN: STGCN proposes a novel model of dynamic skeletons called Spatial- Temporal Graph Convolutional Networks (ST-GCN). This model moves beyond the limitations of previous methods. It automatically learns both the spatial and temporal patterns from data. This not only leads to greater expressive power but also stronger generalization capability.

4.4. Video Augmentation

We support around 175 image and video augmentation primitives that help to supplement

our training data to avoid overfitting. A few categories of augmentation techniques used can be found here:

4.4.1. Arithmetic

This type of augmentation allows modifying values sampled from distributions or combining several distributions with each other. This includes several techniques like SaltAndPepper augmentation, adding Gaussian noise, Inverting the frames, etc.

4.4.2. Blending/Overlaying Images

Since most of the augmentation techniques affect images in uniform ways per image, we might sometimes want to instead demand more localized effects (e.g. changing the color of only some of the image regions, while keeping the others unchanged).

We might also want to keep a fraction of the old image and combine it with a new image (e.g. blur the image and mix in a bit of the unblurred image). Blending/Overlapping augmenters are the desired augmenters for these use cases. They either mix two images using a constant alpha factor or using a pixel-wise mask.

4.4.3. Color/Contrast

Color and Contrast augmentation techniques are the most common techniques. These techniques include changing the brightness, Saturation, adding Hue to the image, using techniques such as CLAHE, Gamma Contrast, etc to achieve Color and Contrast augmentation.

4.4.4. Size and Scale

Resizing and scaling images to make the model robust to images is achieved through augmentation techniques like CentreCrop, Crop and Pad, Resize, etc.

4.4.5. Segmentation

These types of data augmentation techniques include transforming image, either

completely or partially, to their superpixel representation. Some examples are UniformVoronoi, Superpixels, etc.

4.5. Effectiveness of Pipeline Search

We conducted a preliminary experiment to showcase the effectiveness of pipeline search. We define a search space that allows choosing one of the 175 augmentation methods in the pipeline and tunes the hyperparameters of the TSN model with learning rate [0.001,0.0001], momentum [0.9,0.99], weight decay [5e-4,1e-3], number of segments {8,16,32}. We separate 5% of the training data for validation purposes. Then we apply the random search and Hyperopt tuners to search the pipeline configurations on the validation data. We run the search with 50 samples and evaluate the best configuration discovered in the search on the testing data.

	HMDB-6	HMDB-51
Default Pipeline	70.35%	46.8%
Random Search	90.01%	54.12%
Hyperopt	**92.90%**	**56.28%**

Table 2: Accuracy of tuners in AutoVideo

In Table 2, we compare the two tuners as well as the default pipeline configuration on HMDB-51, a task that aims to identify 51 human actions, and a subset of it with the first six actions named HMDB-6. Both the tuners outperform the default pipeline by a large margin, which verifies the effectiveness of pipeline search and hyperparameter tuning. We defer a more comprehensive evaluation to our future work since it is out of the scope of our demo presentation.

5. LIVE AND INTERACTIVE PARTS

As mentioned in the previous sections, we have created a very interactive drag-drop based user interface to make the whole pipeline interactive for the user. In the demo session, we will demonstrate our GUI to the audience. Here is a GIF demonstrating the performance of our model on a sample video:

Figure 6: Inference on sample test video [59]

As can be noticed, the model trained model obtained after running the automated pipeline search is able to perform well and detect the action with a very high confidence score.

6. CONCLUSION

We present AutoVideo[59], an automated video action recognition system. It provides a highly modular implementation of 195 primitives, on which users can flexibly create various training pipelines. It also supports automated tuners and an easy-to-use GUI to help the users quickly develop prototypes. We have open-sourced our implementations with the hope that they can benefit researchers and practitioners.

7. BED FRAMEWORK

7.1 Objectives

The primary objective of this section is to provide a framework for deploying AI models for specific use cases on highly constrained edge devices. We have chosen an AI powered microcontroller chip, MAX78000, as the deployment framework[57]. There are several challenges in deploying a video understanding task on edge devices[51][50][49]. For the purpose of this demonstration, we have chosen to deploy object detection. Our system, which we name BED, is an end to end system involving model training, Quantization[42][25] and deployment to the edge device can be explained in more detail in the following sections[58].

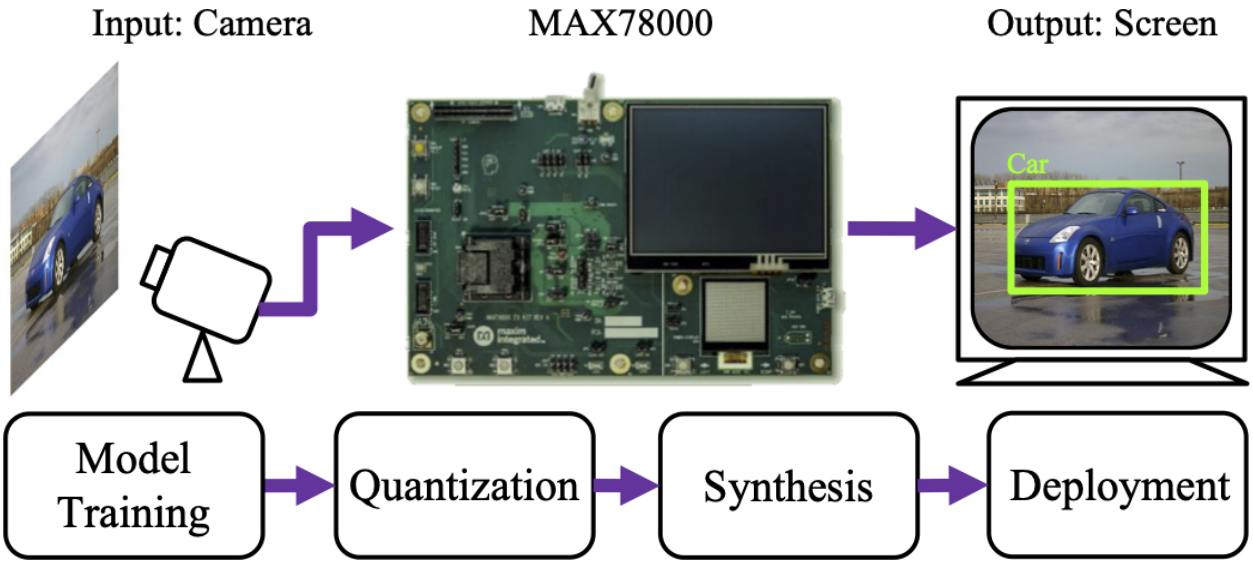

Figure 7: BED implements a real-time and end-to-end object detection system from the camera to the screen[58]

Figure 7 shows the BED pipeline, which includes four stages: (i) model training stage that employs Quantization Aware Training to train a model, (ii) quantization stage that performs an 8-bit quantization, (iii) synthesis stage that converts the model to executable C code, and (iv) deployment stage that compiles the C code and loads the executable model to the edge device. We will first provide a background of MAX78000, and then elaborate on each of the stages.

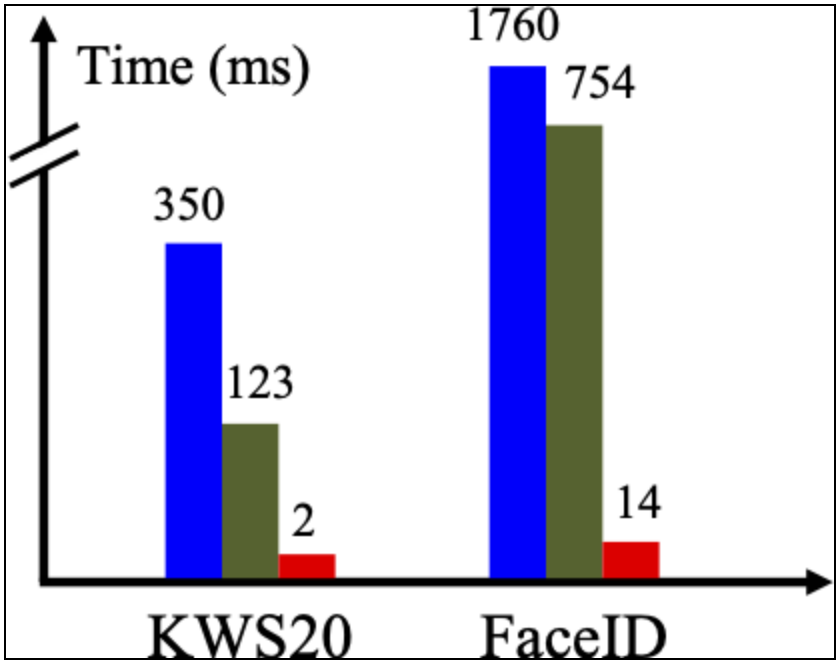

Figure 8: Comparison of inference time (left) and power consumption (right) of MAX78000 with two non-AI chips[58]

7.2. MAX78000 DNN Accelerator:

MAX78000 DNN Accelerator is a powerful AI microprocessor for efficient and low-power inference on edge devices[43]. Figure 6 ~\ref{fig:time_power_comparison} compares the inference time and the power consumption of MAX78000 with two non-AI microprocessors MAX32650 and STM32F7 on two representative DNN tasks KWS20 and FaceID. The reason for choosing MAX78000 to deploy our object detection systems include the following:

- MAX78000 enjoys a significant advantage in terms of inference time as is evident from the graph above.
- In addition to a significant advantage in inference time, MAX78000 also enjoys a huge advantage in terms of power consumption.

7.3. Challenges

Despite its clear advantages, MAX78000 has several hard constraints on the model, making it challenging to design DNNs. The challenges mainly include memory constraint, operator constraints, etc which are described below in detail:

7.3.1. Limited Operators Supported

First, to speed up the inference, it only supports very few operators: 3x3 convolution kernel, 1x1 convolution kernel, average pooling, maximum pooling, Relu activation function, etc. This severely limits the capability to leverage the various state-of-the-art object detection architectures whose operators are currently unsupported by the MAX78000 SDK. Techniques such as skip connections, 7x7 convolutional kernels, activation functions like LeakyReLU, etc are not supported. Other operators such as Dilated Convolution, group convolution, 3D Convolution, etc. are also unsupported. This limits the types of architectures that can be utilized for research and development.

7.3.2. Limited Flash Memory

Second, MAX78000 has only 432 KB flash for storing model parameters. This severely limits the capability to deploy a deep model on such a memory constrained environment[39]. With only 432 KB of memory it is impossible to fit even the smallest state-of-the-art object detection architecture.

7.3.3. Need for Quantization

Third, MAX78000 only supports INT8 inference while FP32 inference is not supported. This means that the models trained offline at FP32 cannot be directly deployed on the chip and require model Quantization before deployment. Quantization usually comes with reduced performance due to an inherent quantization loss associated with it[42].

It is, therefore, challenging to achieve a good performance under such operator and memory constraints. Hence, it is needed to adopt operator-level approximation for existing CNN backbones[44] such as the GoogleNet, MobileNet or EfficientNet.

7.4. Solutions

Solution to Physical constraints:

- Select GoogleNet as the backbone (without residual structure).
- Remove the unsupported operators (e.g. use Relu to replace Leaky Relu, use 3x3 conv2d to replace 7x7 conv2d, etc.).
- Remove the full-connection layers (use full-convolutional network).
- Clip input image to 3x224x224 (original 3x448x448)[37].
- Clip Channels of inter-media layers.
- Use streaming mode for large feature map forward.

Tricks involved in the training.

- Little learning rate and batch size.
- Reduce the difficulty-level to 5 class-objection detection.
- Pretrain the model on the small subset of training set (only 450 images).

Performance degradation of Post-training-quantization (PTQ).

- Adopt Quantization-aware Training (QAT).

7.5. Model Training

Model training involves the process of training the DNN architecture end-end as an offline training phase before the model can be deployed and exported to the board.

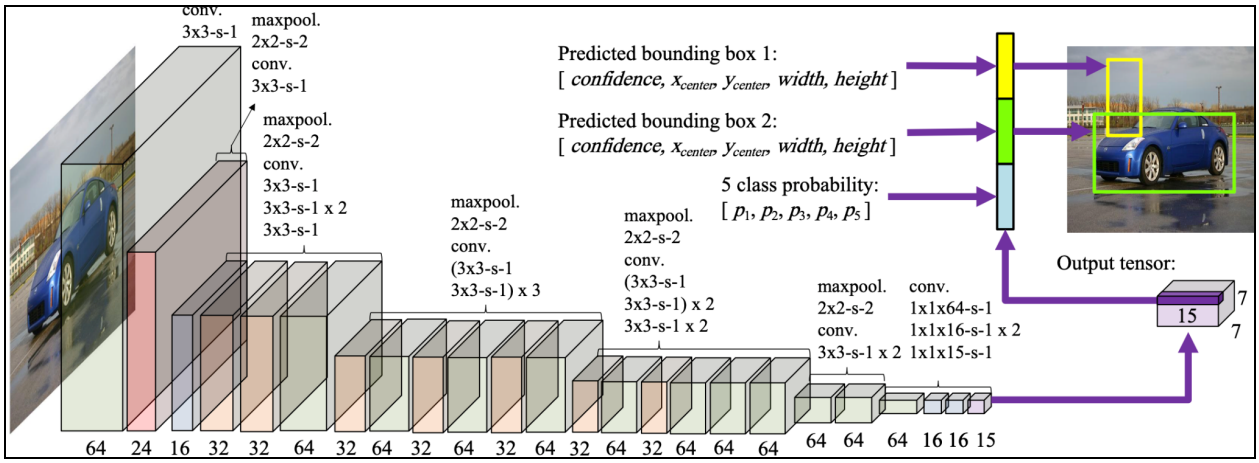

Figure 9: DNN model structure[58]

8. EXPERIMENTS

8.1. Datasets

VOC2007 Dataset[38]: For this experiment, we make sure of the highly competitive and state-of-the-art benchmark dataset called VOC-2007[38]. The training data consists of a huge set of images where each image has an annotation file. This file consists of a bounding box and an object class label for each object in one of the twenty classes present in the image. Multiple objects from multiple classes may be present in the same image which makes the task highly competitive.

8.2. Model Architecture

This subsection introduces the neural architecture of BED and the training details. We focus on standard object detection tasks, which aim to learn a DNN to detect the coordinate and the class of the existing objects from an image. We design and train the CNN model for the BED in this section[58].

Specifically, BED follows the general object detection to learn a CNN model to detect the coordinate and class of the existing objects from an input image. To maximally utilize the limited memory for model parameters, BED adopts a fully convolutional network for the detection.

Note that MAX78000 supports very limited operators as discussed in the above section. The DNN model is therefore constructed fully based on 3x3 convolutional layer, Relu activation, Batch normalization, 2x2 max-pooling without the use of other operators, as shown below in Figure 9.

In this way, BED spends only 300KB on the storage of model parameters to satisfy the harsh memory constraints on the MAX78000 device. An input image is in the RGB format with a size of 224x224x3, and is divided into a 7x7 grid, where each cell has a 32x32 area.

The model outputs a 7x7x15 tensor for each image, where each of the 7x7 cells corresponds to a 15 dimensional output vector consisting of class probabilities, two bounding boxes and their confidence scores. In such a manner, the model outputs the detection result which contains both the coordinates and the class[58].

8.3. Quantization

The model is trained on a subset of VOC2007 dataset which contains the images of five classes with their annotations. Post training, the model needs to be Quantized to INT8 precision which is currently supported on the MAX78000 device. However, directly applying quantization post training leads to a huge performance gap due to a Quantization error[25] associated with this process. To minimize the performance degradation after the post-quantization, we adopt Quantization Aware Training.

This involves training the model several epochs without Quantization and then applying Quantization in the training process. After applying Quantization, the model is again trained for several more epochs to ensure that the Quantization loss is minimized once post-training Quantization is applied later.

8.4. Network Pruning

How did we prune the model?

- Keeping the overall structure of GoogleNet intact.
- Prune the network in the channel dimension of each layer.
- Reduce the number of convolution operations in each block (MEM-limitation).
- Replace the unsupported ops by supported ops (ops, activation).

How did we know which layer to use and which layer to prune?

- Experiment results (convergence, testing result (FP))

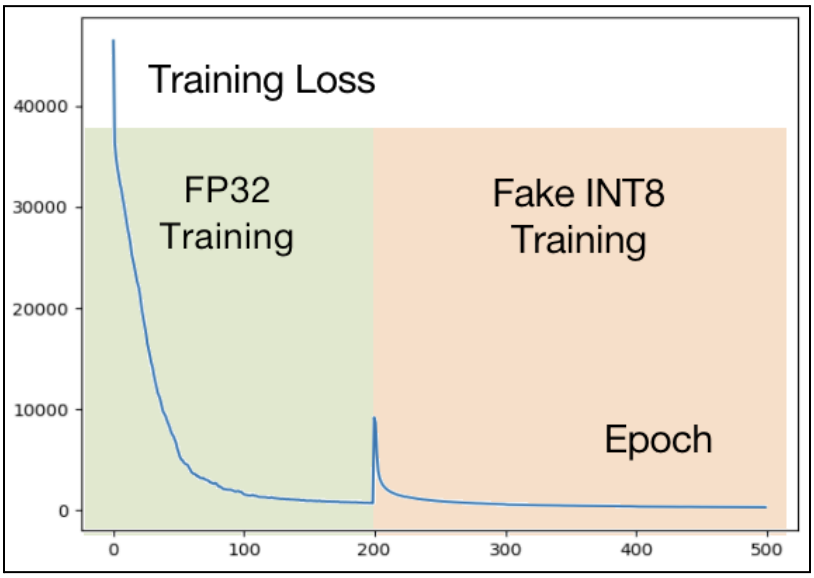

Figure 10: Training loss with QAT after 200 epochs

Model	**#Class**	**mAP**	**Mem-cost**	**Res. Layers**	**Bit width**	**Device**
Experiment 1	5	18.96	**299.5KB**	**No**	**8**	MAX78000
Experiment 2	5	3.04	**299.5KB**	**No**	**8**	MAX78000
Animals	5	2.34	**299.5KB**	**No**	**8**	MAX78000
Transport	5	11.08	**299.5KB**	**No**	**8**	MAX78000
Yolo-v3 tiny[69]	**80**	17.6	35.2 MB	Yes	32	GPU
Tiny yolo-v1	20	**23.7**	42.87 MB	Yes	32	GPU

Table 3: Comparing performance of various experiments of our BED with Yolo-v3 tiny

8.5. Common Quantization techniques

Quantization is a very important step when deploying DNN models on the edge. The two main types of Quantization techniques are given below:

8.5.1. Quantization Aware Training

We generally observe a significant accuracy drop as we move from floating point precision to a lower precision like integer precision. To help us minimize this loss, we use a technique called quant-aware training.

So basically, quant-aware training simulates low precision behavior in the forward pass, while the backward pass remains the same at floating point precision. This ensures that we are able to induce some quantization error which is accumulated in the total loss of the model. By this technique we can make sure that the optimizer tries to reduce this loss by adjusting the parameters accordingly. This makes our parameters more robust to quantization making our process almost lossless.

8.5.2. Post-Training Quantization

Oftentimes, a model that runs effectively on your system during training and inference might not be able to achieve the same performance on an edge device with hardware constraints. The main reason behind this is usually the hardware constraints of the target device. This is where we can leverage post-training quantization to help improve the optimization of the algorithms and models for the target device. Post-training quantization is a conversion technique which helps mitigate the performance degradation while also reducing model size and improving CPU and hardware accelerator latency.

8.6. Training Strategy

We follow the training strategy of Yolo-V1[48][31] to update the model. Specifically, we adopt cross entropy and mean square error loss functions for the classification and the bounding box coordinates, respectively.

Furthermore, we employ the SGD optimizer with $3x10^{-5}$ learning rate, and adopt the mini-batch updating with batch-size 16 to update the parameters of the model for 400 epochs. We back up and store the snapshot of the model at the end of each training epoch; and select the optimal model to maximize the mean average precision on the validating dataset.

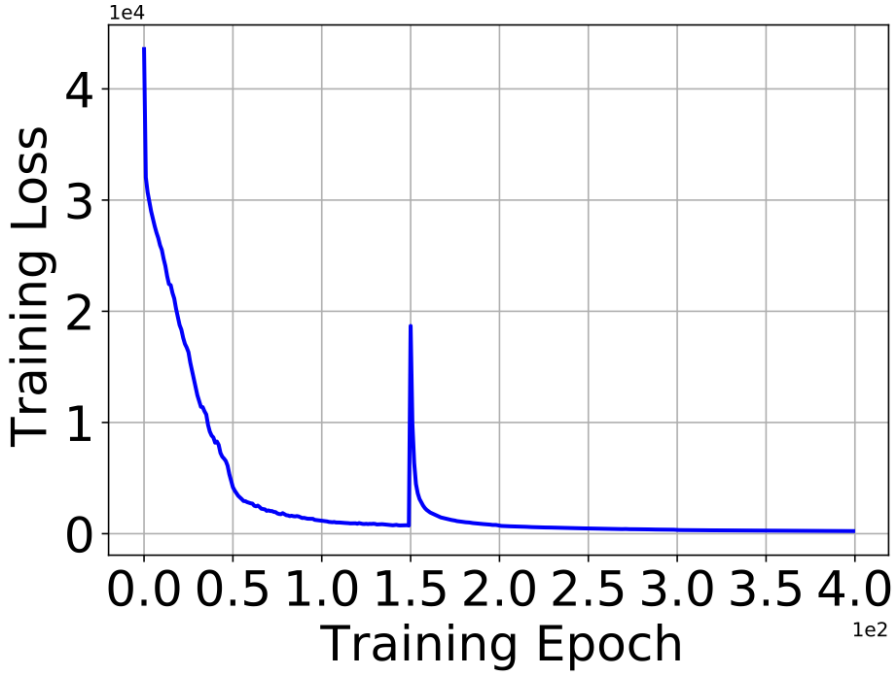

Figure 11: Loss vs epoch with QAT at 150 epoch

More training details are provided in our repository. The whole training process is based on the Maxim QAT training library. Each Batch-normal layer is merged to its front convolutional layer during the training for deploying the model after the training. During the training process,

each Batch-normal layer is merged to its front convolutional layer for the deployment to the hardware.

8.7. Preprocessing

The images from the VOC2007 dataset are first pre-processed to 224x224x3 before passing into the model architecture. Inspired by GoogleNet which pre-processes the images to 448x448x3, we choose 224x224x3 as the model memory constraints limit our capability to use larger image sizes. We also reduce the class number of the original architecture from 20 to 5 inorder to satisfy the memory constraints.

8.8. Model Training Workflow

This subsection introduces our training workflow including the process of finalizing on the model architectural details, quantization aware training, synthesis, etc. The entire workflow can be found in the Figure 12 below.

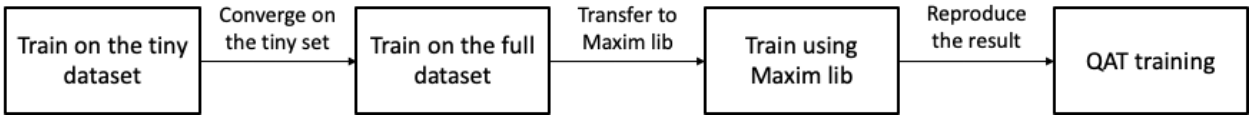

Figure 12: Training Workflow of BED

Train on the tiny dataset:

We first create a smaller subset of the original dataset for our architecture building phase. The small dataset ensures faster convergence. This in turn ensures that we are able to make faster iterations leading to exploration of different types of architectures.

Train on full dataset:

After converging the loss on the tiny dataset and choosing the best performing architecture, we then train this architecture on the full dataset. This ensures that we are quickly able to reach

higher performance on the actual dataset with smaller iterations on the tiny dataset.

Train using Maxim lib:

Finally, we transfer the code to the Maxim SDK codebase which includes operators, expressions that are supported by the MAX78000 microcontroller. This also includes APIs for other important tasks like Quantization, Synthesis, etc.

QAT training:

The last step in the training phase involves invoking quantization aware training after letting the model converge at FP32 precision. After several iterations, we start the QAT and ensure the model is able to converge at INT8 precision.

8.9. Object Detection

For model architecture exploration, our model architecture is inspired from several state-of-the-art architectures which include the following:

- MobileNet[68]: MobileNet presents a novel class of efficient models that can perform well for mobile and embedded vision applications. These architectures are based on a streamlined architecture that uses depth-wise separable convolutions and thereby is able to build light weight deep neural networks.
- Yolov3-tiny[69]: YOLO v3-Tiny is another version of YOLO v3[47][34][30] with the decreased depth of the convolutional layer[33]. It was proposed by Joseph Redmon [10]. It significantly increased the running speed (approximately 442% faster than the former variants of YOLO), but also led to a reduction in detection accuracy.
- Tiny Yolo-v1: This model can identify up to 20 classes and was trained using the Pasval VOC data set. The image is divided by the model into a 7x7 grid (49 grid cells total).

Two anchor boxes are present in each grid cell. These two anchor boxes each include 20 class probability values as well as an object score.

9. QUANTIZATION AND SYNTHESIS

This subsection describes the process of Quantization post-training in detail. It also describes the process converting the python code into corresponding C code which can later be deployed on the device.

9.1. Quantization

The trained model will be processed by quantization and synthesis such that it can be deployed on MAX78000.

The quantization stage reads a checkpoint file of the float pre-trained model and outputs the corresponding quantized model. During this process, the pre-trained model is processed by a 32-bit quantization for the last layer and an 8-bit quantization for the remaining layers, which involves the quantization of weight, bias and activation function for each layer of the model.

Our quantization code is available here.

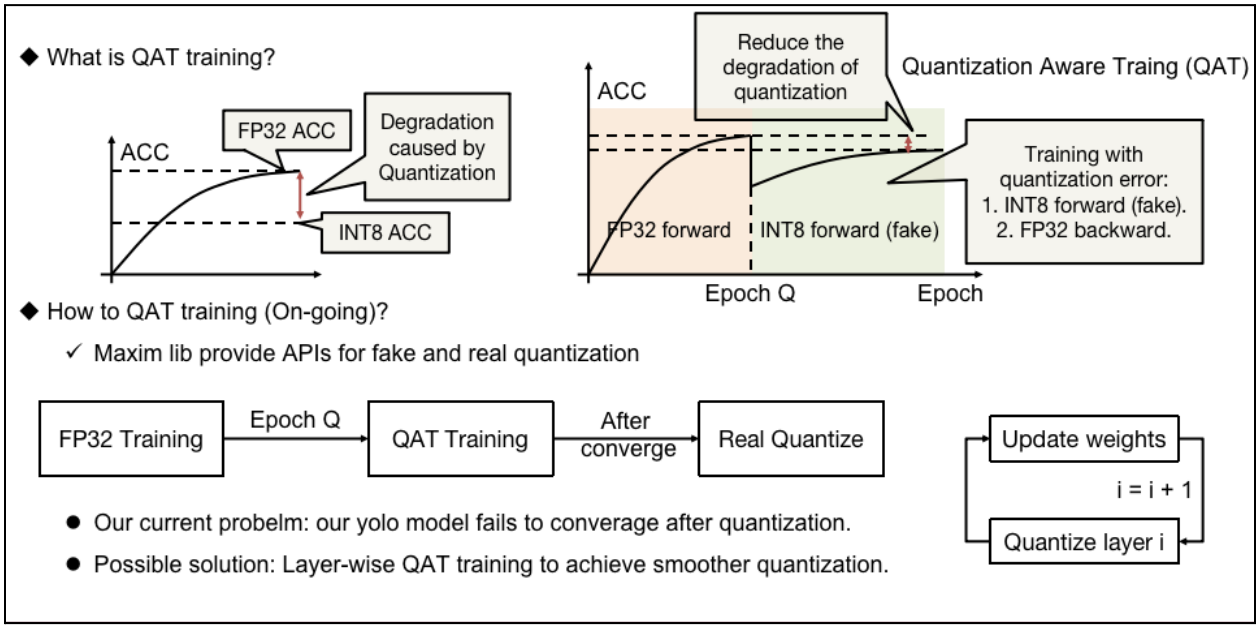

Figure 13: QAT technique used in our experiments

9.2. Synthesis

The synthesis stage converts the quantized pre-trained model into a C program. Specifically, it reads the checkpoint of the pretrained model after the quantization and automatically generates header files to store its weights, bias and hyper-parameters. It also wraps up other requirements including the configuration files for the deployment. Our synthesis code is available here.

BED adopts MAX78000 CNN accelerator for the deployment of the model following the pipeline in Figure 7.

9.3. Deployment

The subsection introduces the process of deploying the trained, synthesized code on the MAX78000 chip.

After the synthesis, the C program is compiled into executable code using the ARM embedding compiler. The executable model is loaded to MAX78000 through serial protocol.

The source code of the deployment is given here.

As an integrated system, BED adopts a camera to capture the images and an LCD screen to display the detection results. These can be showcased by Figure 11 which shows the MAX78000 board including the DNN accelerator, the camera and the LCD Screen.

The image is captured by the camera which is represented as a three-dimensional matrix with three channels of red, green and blue.

The images are loaded to MAX78000 block by block, where the blocks are temporarily stored to a flash memory with 896KB capacity inside MAX78000 until all of the blocks have been loaded.

After this, the model takes as input the image and outputs a 7x7x15 tensor to indicate the classification and bounding boxes in each grid of the input image.

The global result is selected from the grid-level 7x7x15 tensor via Non-maximum Suppression, where the global result is a 15-dimensional vector to indicate the detection result for the whole input image, as shown in Figure 7.

Finally, the result of classification and bounding box will be displayed on the LCD screen.

9.4. Latency of Real-time Detection

The latency of BED is measured by averaging 100 times of on-chip inference, which starts from an image loading to the output of detection results. The average inference time and energy of BED are given in Table 1, which are 91.9 ms and 1.845 mJ, respectively.

Moreover, BED merely requires 299.52 KB memory to store the network weights of the deep object detection model. Generally, the latency of BED satisfies the demands of real-world scenarios in terms of the constraints of latency, energy and memory.

Processing	Power	Inference time	Energy
Image loading and DNN inference	20.08 mW	91.9 ms	1.845 mJ

Table 4: Measurement of Power, Inference time and Energy consumption for the on-board processing[58]

10. EVALUATION AND DEMONSTRATION

This subsection involves evaluation of the model performance on metrics such as mAP. It also involves evaluation based on visualization of quality of bounding boxes obtained both from the offline inference on the server and online inference on the MAX78000 chip using the camera module.

10.1. Offline Evaluation

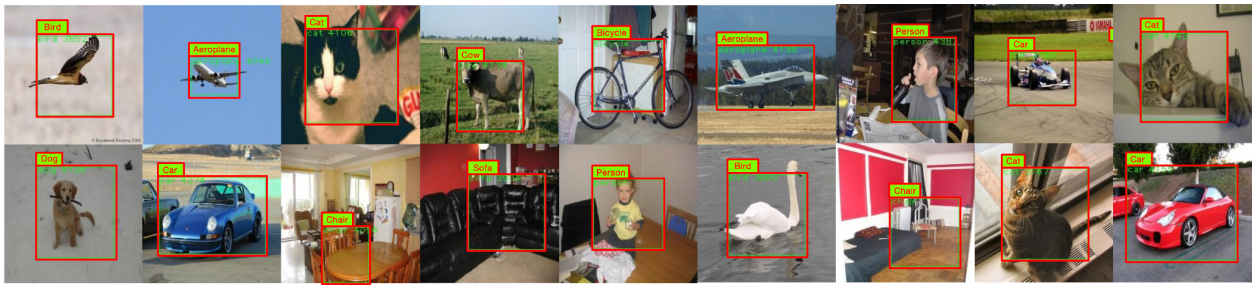

Figure 14: Offline detection results[58]

The offline experiment focuses on evaluating the performance of the pre-trained model (after the quantization) before loading it to the edge device. We visualize the detection results of some randomly chosen images from the testing set of VOC2007 in Figure 13.

It is observed that BED can accurately detect the objects in the input images. The results look highly promising considering the highly constrained requirements on the model architecture and the simple model architecture chosen for this task.

10.2 Offline Evaluation Metrics

Specifically, we measure the mean average precision (mAP) [32] of the pre-trained model on the VOC2007 testing set, and report the evaluation results in Table 5, where we compare our BED with Yolo-v3 [47][30] and Tiny yolo. It is observed that BED achieves competitive performance with more than 100X less memory-cost and much smaller bit-width.

Model	Classes	mAP	Mem	Backbone	Res. layer	Bit width	Device
Ours	5	18.96	**349KB**	**Ours**	**No**	**8**	MAX78000
Yolo-v3 tiny[2]	**80**	17.6	35.2MB	Darknet	Yes	32	GPU
Tiny yolo [3]	20	**23.7**	42.87MB	VGG19	No	32	GPU

Table 5: Comparison of performance with other state-of-the-art models

As can be noticed from the table above, our model utilizes far less memory compared to the smaller YOLO model available and is able to achieve a decent mAP[32]. This is not an apples-apples comparison as the number of classes, bit width, presence of residual blocks make the models inherently different. But it serves as an initial proof of concept that a tiny model under highly constrained environments can also work well on complicated tasks such as Object Detection.

10.3. Real-time Demonstration

We conduct two real-time experiments to demonstrate the on-device object detection. The first experiment focuses on on-device inference. Specifically, a personal computer will transmit images to MAX78000, which then sends the detection results back to the computer. On the computer-side, we implement a Graphical User Interface (GUI) to send the image, receive and show the detection results, as shown in Figure 14.

The second experiment eliminates the dependence on personal computers to send and receive images. It involves making use of real-time inference by making use of a camera module that captures videos in real-time, converts them into frames and applies real-time inference. The results are displayed real-time on the on-device LCD screen.

10.3.1 Graphical User Interface

We develop a simple GUI for on-device inference. The GUI consists of simple interface that lets us load images from the computer and transmit these images to the MAX78000. The device performs inference which is later transmitted back to the GUI to display the image along with the bounding box. The GUI also displays the top three classification results along with their corresponding probability scores. The GUI is shown in Figure 13.

For more results, please watch our demo video. More results of BED are given in our demonstration video.

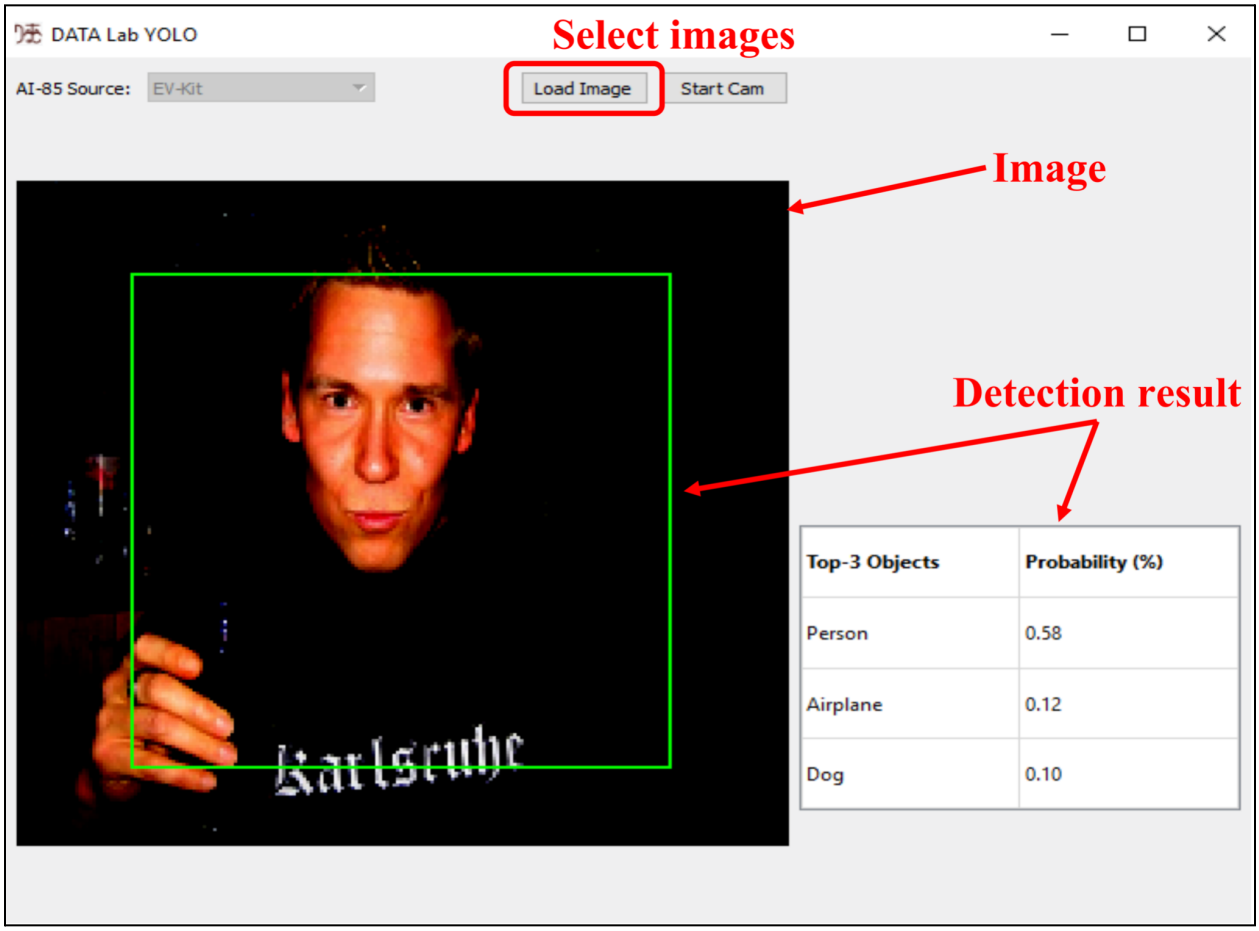

Figure 15: GUI

10.3.2. Real-time Demonstration

The second experiment demonstrates the whole pipeline of BED based on the testing bed in Figure 10, where an image is shown on a source screen; BED captures the image by the camera and shows the detection results on the screen.

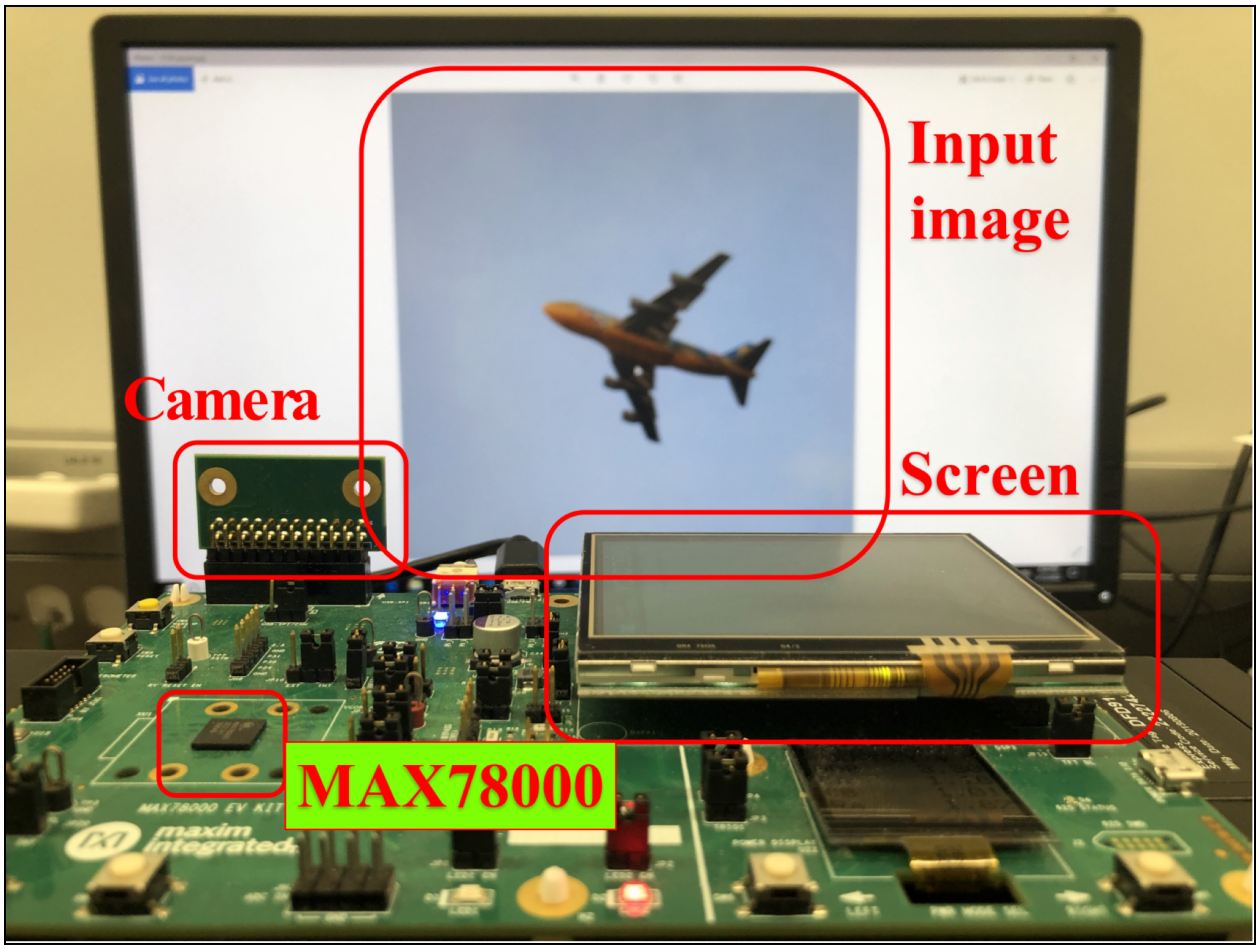

Figure 16: Testing bed[58]

We randomly select some images from the testing set of VOC2007, and give the real-time detection results in Figure 11. BED can correctly detect the object in the image captured by the camera and show the results on the screen.

Some of the results captured by BED on the LED screen can be displayed below in Figure 14 which indicates the effectiveness of the model on real-time detection.

Figure 17: Real-time detection results[58]

11. LIVE AND INTERACTIVE PARTS

In the thesis, we present a live demo of real-time object detection based on MAX78000 using our developed GUI in the appendix. Moreover, we will give a tutorial of BED, including platform setup and model deployment in the Appendix section with links to the GitHub consisting of tutorials, demo video and open-source code.

12. CONCLUSION

In this demo, we build BED, an integrated system for real-time object detection on edge devices. BED supports capturing images with a camera, displaying the detection results with an LCD screen, and on-chip inference of the DNN. BED shows promise in deploying complex DNNs to edge devices with very limited memory, bit-width and computational resources.

In the future, we will explore adopting neural architect search to optimize the network architecture under the constraints of memory, latency and power consumption.

13.1 Integrating AutoVideo and BED

The future work involved integrating both the AutoVideo project and BED in a single package. The idea is to have an end-end system which has the following stages in the Machine Learning Development lifecycle.

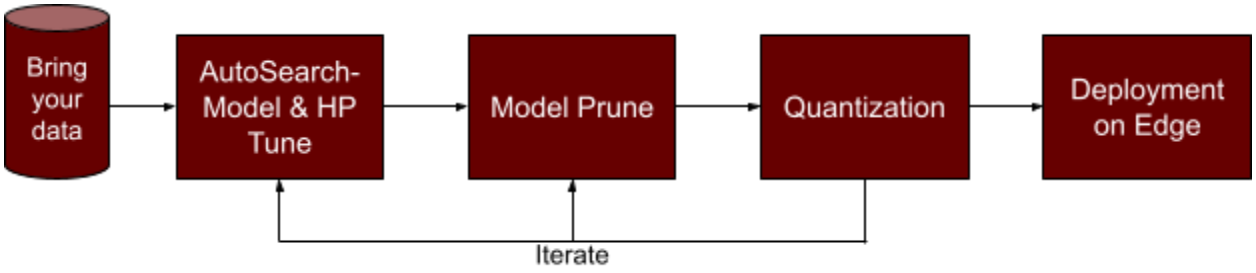

Figure 18: Proposed future workflow integration

This will help the developers to create accurate, customized, and production-ready AI models to power their computer vision AI research and development.

Currently we support all the steps in this lifecycle but we are yet to integrate both AutoVideo and BED as an end-end system. Additionally, for the current scope we performed manual pruning[40]. In the future we aim to support automated model pruning[40] and Quantisation in addition to supporting Deployment on popular edge devices. The key features which we support and plan to support include:

Augment Your Data

Improve your model performance with data augmentation features such as spatial and color transformations. We currently support 175 data augmentation techniques.

Optimize for Inference

Plan is to support optimization through model pruning which can achieve up to 4X speed up in inference with pruning and INT8 quantization while maintaining comparable accuracy.

Deploy on Edge

Though we currently support deploying object detection applications on the MAX78000 chip, we plan to have a generalized framework that can be deployed on few most commonly used edge devices like the NVIDIA Jetson Nano, MAX78000, etc

REFERENCES

[1] H. Meng, N. Pears and C. Bailey, "A Human Action Recognition System for Embedded Computer Vision Application," 2007 IEEE Conference on Computer Vision and Pattern Recognition, 2007, pp. 1-6, doi: 10.1109/CVPR.2007.383420.

[2] Y. Gao et al., "Human Action Monitoring for Healthcare Based on Deep Learning," in IEEE Access, vol. 6, pp. 52277-52285, 2018, doi: 10.1109/ACCESS.2018.2869790.

[3] Ronald Poppe, A survey on vision-based human action recognition, Image and Vision Computing, Volume 28, Issue 6, 2010, Pages 976-990, ISSN 0262-8856, https://doi.org/10.1016/j.imavis.2009.11.014.

[4] Tran, Du & Bourdev, Lubomir & Fergus, Rob & Torresani, Lorenzo & Paluri, Manohar. (2015). Learning Spatiotemporal Features with 3D Convolutional Networks. 4489-4497. 10.1109/ICCV.2015.510.

[5] L. Wang et al., "Temporal Segment Networks for Action Recognition in Videos," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 41, no. 11, pp. 2740-2755, 1 Nov. 2019, doi: 10.1109/TPAMI.2018.2868668.

[6] J. Carreira and A. Zisserman, "Quo Vadis, Action Recognition? A New Model and the Kinetics Dataset," 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 4724-4733, doi: 10.1109/CVPR.2017.502.

[7] Zolfaghari, Mohammadreza & Singh, Kamaljeet & Brox, Thomas. (2018). ECO: Efficient Convolutional Network for Online Video Understanding.

[8] [Yue Zhao, 2019] Dahua Lin Yue Zhao, Yuanjun Xiong. Mmaction. mmaction, 2019. Accessed: 2022-05-22

[9] [Team, 2021] PytorchVideo Team. Pytorchvideo. PyTorchVideo , 2021. Accessed: 2022-05-22

[10] [Ryoo et al., 2019] Michael S Ryoo, AJ Piergiovanni, Mingxing Tan, and Anelia ngelova. Assemblenet: Searching for multi-stream neural connectivity in video architectures. arXiv preprint arXiv:1905.13209, 2019.

[11] [Milutinovic et al., 2020] Mitar Milutinovic, Brandon Schoenfeld, Diego Martinez-Garcia, Saswati Ray, Sujen Shah, and David Yan. On evaluation of automl systems.In ICML Workshop, 2020

[12] [Lai et al., 2021a] Kwei-Herng Lai, Daochen Zha, Guanchu Wang, Junjie Xu, Yue Zhao, Devesh Kumar, Yile Chen, Purav Zumkhawaka, Minyang Wan, Diego Martinez, et al. Tods: An automated time series outlier detection system.In AAAI, volume 35, pages 16060–16062, 2021

[13] Lin, Ji et al. “Temporal Shift Module for Efficient Video Understanding.” ArXiv abs/1811.08383 (2018): n. pag.

[14] Hou, Rui & Chen, Chen & Sukthankar, Rahul & Shah, Mubarak. (2019). An Efficient 3D CNN for Action/Object Segmentation in Video.

[15] Haifeng Jin, Qingquan Song, and Xia Hu. 2019. Auto-Keras: An Efficient Neural Architecture Search System. In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '19). Association for Computing Machinery, New York, NY, USA, 1946–1956.

[16] Thornton, Chris & Hutter, Frank & Hoos, Holger & Leyton-Brown, Kevin. (2012). Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. KDD. 10.1145/2487575.2487629.

[17] [Li et al., 2020] Yuening Li, Daochen Zha, Praveen Venugopal, Na Zou, and Xia Hu. Pyodds: An end-to-end outlier detection system with automated machine learning. In WWW, pages 153–157, 2020.

[18] [Bergstra et al., 2013] James Bergstra, Dan Yamins, David D Cox, et al. Hyperopt: A python library for Optimizing The Hyperparameters Of Machine Learning algorithms. In Proceedings of the 12th Python in science conference, volume 13, page 20. Citeseer, 2013

[19] Hara, Kensho et al. “Learning Spatio-Temporal Features with 3D Residual Networks for Action Recognition.” 2017 IEEE International Conference on Computer Vision Workshops (ICCVW)(2017): 3154-3160.

[20] [Moritz et al., 2018] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging{AI}Applications. In OSDI, pages 561–577, 2018

[21] [Demsaret al., 2004] Janez DemSar, Blaz Zupan, Gregor Leban, and Tomaz Curk. Orange: From experimental machine learning to interactive data mining. In PKDD, pages 537–539. Springer, 2004

[22] [Piergiovanni et al., 2019] AJ Piergiovanni, Anelia Angelova, Alexander Toshev, and Michael S Ryoo. Evolving space-time neural architectures for videos. In ICCV, pages 1793–1802, 2019

[23] [Devlin et al., 2018] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert:Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805, 2018.

[24] [Dosovitskiy et al. , 2020] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In International Conference on Learning Representations, 2020.

[25] [Krishnamoorthi, 2018] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. arXiv preprint arXiv:1806.08342,2018.

[26] [Kristopher and Robert, 2020] Ardis Kristopher and Muchsel Robert. Cutting the ai power cord:Technology to enable true Edge Inference. https:// cms.tinyml.org/ wp-content/ uploads/ talks2020/ tinyML Talks Kris Ardis and Robert Muchsel -201027.pdf, 2020.

[27] [Long et al., 2015] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 3431–3440, 2015.

[28] [Murshed et al., 2021] MG Sarwar Murshed, Christopher Murphy, Daqing Hou, Nazar Khan, Ganesh Anantha Narayanan, and Faraz Hussain. Machine learning at the network edge: A survey. ACM Computing Surveys (CSUR), 54(8):1–37, 2021.

[29] [Pathak et al., 2018] Ajeet Ram Pathak, Manjusha Pandey, and Siddharth Rautaray. Application of deep learning for object detection.Procedia computer science, 132:1706–1717, 2018.

[30] [Redmon and Farhadi, 2018] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. arXiv preprint arXiv:1804.02767, 2018.

[31] [Redmon et al., 2016] Joseph Redmon, Santosh Divvala,Ross Girshick, and Ali Farhadi.
You only look once: Unified, real-time object detection. In Proceedings of theIEEE conference on computer vision and pattern recognition, pages 779–788, 2016.

[32] Henderson P, Ferrari V. End-to-end training of object class detectors for mean average precision. InAsian Conference on Computer Vision 2016 Nov 20 (pp. 198-213). Springer, Cham.

[33] Redmon J, Farhadi A. Yolov3: An incremental improvement. arXiv preprint arXiv:1804.02767. 2018 Apr 8.

[34] Redmon, Joseph and Ali Farhadi. “YOLOv3: An Incremental Improvement.” ArXiv abs/1804.02767 (2018): n. pag.

[35] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding.
arXiv preprint arXiv:1810.04805, 2018.

[36] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud computing: issues and challenges. In 2010 24th IEEE international conference on advanced information networking and applications, pages 27–33. Ieee, 2010.

[37] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiao-hua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In International Conference on Learning Representations, 2020.

[38] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisser-man. The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results.
http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html.

[39] Yanjie Gao, Yu Liu, Hongyu Zhang, Zhengxian Li, Yonghao Zhu, Haoxiang Lin, and Mao Yang. Estimating gpu memory consumption of deep learning models. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 1342–1352, 2020.

[40] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv preprint arXiv:1510.00149, 2015.

[41] Najmul Hassan, Saira Gillani, Ejaz Ahmed, Ibrar Yaqoob, and Muhammad Imran. The role of edge computing in the internet of things. IEEE communications magazine, 56(11):110–115, 2018.

[42] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. arXiv preprint arXiv:1806.08342, 2018.

[43] Ardis Kristopher and Muchsel Robert. Cutting the ai power cord: Technology to enable true edge inference. https://cms.tinyml.org/wp-content/uploads/talks2020/tinyML_Talks_Kris_Ardis_and_Robert_Muchsel_-201027.pdf, 2020.

[44] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 3431–3440, 2015.

[45] MG Sarwar Murshed, Christopher Murphy, Daqing Hou, Nazar Khan, Ganesh Ananthanarayanan, and Faraz Hussain. Machine learning at the network edge: A survey. ACM Computing Surveys (CSUR), 54(8):1–37, 2021.

[46] Ajeet Ram Pathak, Manjusha Pandey, and Siddharth Rautaray. Application of deep learning for object detection. Procedia computer science , 132:1706–1717, 2018.

[47] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. arXiv preprint arXiv:1804.02767, 2018.

[48] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In Proceedings of the IEEE conference on computer vision and pattern recognition, pges 779–788, 2016.

[49] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. IEEE internet of things journal, 3(5):637–646, 2016.

[50] Science U.S. Department of Homeland Security and Technology Directorate. CCTV technology handbook.
https://www.dhs.gov/sites/default/files/publications/CCTV-Tech-HBK_0713-508.pdf.

[51] Blesson Varghese, Nan Wang, Sakil Barbhuiya, Peter Kilpatrick, and Dimitrios S Nikolopoulos. Challenges and opportunities in edge computing. In 2016 IEEE International Conference on Smart Cloud (SmartCloud), pages 20–26. IEEE, 2016.

[52] Wei Wei, Andrew T Yang, Weisong Shi, and Kewei Sha. Security in internet of things: Opportunities and challenges. In 2016 International Conference on Identification, Information and Knowledge in the Internet of Things (IIKI), pages 512–518.IEEE, 2016.

[53] Wang, L. et al. (2016). Temporal Segment Networks: Towards Good Practices for Deep Action Recognition. In: Leibe, B., Matas, J., Sebe, N., Welling, M. (eds) Computer Vision – ECCV 2016. ECCV 2016. Lecture Notes in Computer Science(), vol 9912. Springer, Cham. https://doi.org/10.1007/978-3-319-46484-8_2

[54] [Zha et al., 2021c] Daochen Zha, Jingru Xie, Wenye Ma, Sheng Zhang, Xiangru Lian, Xia Hu, and Ji Liu. Douzero: Mastering doudizhu with self-play deep reinforcement learning. In ICML, pages 12333–12344. PMLR, 2021

[55] [Zha et al., 2020] Daochen Zha, Kwei-Herng Lai, Mingyang Wan, and Xia Hu. Meta-aad: Active anomaly detection with deep reinforcement learning. In ICDM, pages 771–780. IEEE, 2020.

[56] [Zha et al., 2021a] Daochen Zha, Kwei-Herng Lai, Songyi Huang, Yuanpu Cao, Keerthana Reddy, Juan Vargas, Alex Nguyen, Ruzhe Wei, Junyu Guo, and Xia Hu. Rlcard: a platform for reinforcement learning in card games. In IJCAI, pages 5264–5266, 2021.

[57] [Zha et al., 2021b] Daochen Zha, Wenye Ma, Lei Yuan, Xia Hu, and Ji Liu. Rank the episodes: A simple approach for exploration in procedurally-generated environments. In ICLR, 2021

[58] Wang, Guanchu, Zaid Pervaiz Bhat, Zhimeng Jiang, Yi-Wei Chen, Daochen Zha, Alfredo Costilla Reyes, Afshin Niktash, Mehmet Gorkem Ulkar, Osman Erman Okman and Xia Hu. “BED: A Real-Time Object Detection System for Edge Devices.” ArXiv abs/2202.07503 (2022): n. Pag. In CIKM.

[59] Daochen Zha, Zaid Pervaiz Bhat, Yi-Wei Chen, Yicheng Wang, Sirui Ding, Anmoll Kumar Jain, Mohammad Qazim Bhat, Kwei-Herng Lai, Jiaben Chen, et al. 2022. AutoVideo: An Automated Video Action Recognition System. In IJCAI.

[60] Demsˇar, J.; Zupan, B.; Leban, G.; and Curk, T. 2004. Orange: From experimental machine learning to interactive data mining. In PKDD, 537–539. Springer

[61] Laptev, N.; Amizadeh, S.; and Flint, I. 2015. Generic and scalable framework for automated time-series anomaly detection. In KDD.

[62] Li, Y.; Chen, Z.; Zha, D.; Zhou, K.; Jin, H.; Chen, H.; and Hu, X. 2020a. AutoOD: Automated Outlier Detection via Curiosity-guided Search and Self-imitation Learning. arXiv preprint arXiv:2006.11321 .

[63] Li, Y.; Zha, D.; Venugopal, P.; Zou, N.; and Hu, X. 2020b. PyODDS: An End-to-end Outlier Detection System with Automated Machine Learning. In WWW.

[64] Milutinovic, M.; Schoenfeld, B.; Martinez-Garcia, D.; Ray, S.; Shah, S.; and Yan, D. 2020. On Evaluation of AutoML Systems. In ICML Workshop.

[65] Ren, H.; Xu, B.; Wang, Y.; Yi, C.; Huang, C.; Kou, X.; Xing, T.; Yang, M.; Tong, J.; and Zhang, Q. 2019. Time-Series Anomaly Detection Service at Microsoft. In KDD.

[66] Zha, D.; Lai, K.-H.; Wan, M.; and Hu, X. 2020. Meta-AAD: Active Anomaly Detection with Deep Reinforcement Learn- ing. arXiv preprint arXiv:2009.07415 .

[67] Zhao, Y.; Nasrullah, Z.; and Li, Z. 2019. PyOD: A python toolbox for scalable outlier detection. JMLR .

[68] Howard, Andrew & Zhu, Menglong & Chen, Bo & Kalenichenko, Dmitry & Wang, Weijun & Weyand, Tobias & Andreetto, Marco & Adam, Hartwig. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.

[69] P. Adarsh, P. Rathi and M. Kumar, "YOLO v3-Tiny: Object Detection and Recognition using one stage improved model," 2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS), 2020, pp. 687-694, doi: 10.1109/ICACCS48705.2020.9074315.

APPENDIX A

CODES

All the code for the thesis is open-sourced and hosted on GitHub on the following links:

- AutoVideo GitHub: https://github.com/datamllab/autovideo [59]
- BED GitHub: https://github.com/datamllab/BED_main [58]
- BED GUI GitHub: https://github.com/datamllab/BED_GUI
- BED Camera Deployment GitHub: https://github.com/datamllab/BED_camera

Code Snippets

Ease of using the AutoVideo interface can be demonstrated by the following steps.

Load training dataset:

```
train_table_path = os.path.join(args.data_dir, 'train.csv')
train_media_dir = os.path.join(args.data_dir, 'media')
target_index = 2

from autovideo import fit, build_pipeline, compute_accuracy_with_preds
# Read the CSV file
train_dataset = pd.read_csv(train_table_path)
```

Define Training Configuration/Build training pipeline:

```
# Build pipeline based on configs
# Here we can specify the hyperparameters defined in each primitive
# The default hyperparameters will be used if not specified
config = {
    "transformation":[
        ("RandomCrop", {"size": (128,128)}),
        ("Scale", {"size": (128,128)}),
    ],
    "augmentation": [
        ("meta_ChannelShuffle", {"p": 0.5} ),
        ("blur_GaussianBlur",),
        ("flip_Fliplr", ),
        ("imgcorruptlike_GaussianNoise", ),
    ],
    "multi_aug": "meta_Sometimes",
    "algorithm": args.alg,
    "load_pretrained": args.pretrained,
    "epochs": args.epochs,
}
pipeline = build_pipeline(config)
```

Train the model using the defined pipeline:

```
# Fit
_, fitted_pipeline = fit(train_dataset=train_dataset,
                         train_media_dir=train_media_dir,
                         target_index=target_index,
                         pipeline=pipeline)

# Save the fitted pipeline
import torch
torch.save(fitted_pipeline, args.save_path)
```

Make Predictions:

```
# Load fitted pipeline
import torch
if torch.cuda.is_available():
    fitted_pipeline = torch.load(args.load_path, map_location="cuda:0")
else:
    fitted_pipeline = torch.load(args.load_path, map_location="cpu")


# Produce
predictions = produce(test_dataset=test_dataset,
                      test_media_dir=test_media_dir,
                      target_index=target_index,
                      fitted_pipeline=fitted_pipeline)

# Get accuracy
test_acc = compute_accuracy_with_preds(predictions['label'], test_labels)
logger.info('Testing accuracy {:5.4f}'.format(test_acc))
```

Searcher Module:

```
searcher = RaySearcher(
    train_dataset=train_dataset,
    train_media_dir=train_media_dir,
    valid_dataset=valid_dataset,
    valid_media_dir=valid_media_dir
)
```

Search the best pipeline from the defined search space

```
# Tuning
config = {
    "searching_algorithm": args.alg,
    "num_samples": args.num_samples,
}

best_config = searcher.search(
    search_space=search_space,
    config=config
)
```

Define the search space:

```
#Search Space
search_space = {
    "augmentation": {
        "aug_0": tune.choice([
            ("arithmetic_AdditiveGaussianNoise",),
            ("arithmetic_AdditiveLaplaceNoise",),
        ]),
        "aug_1": tune.choice([
            ("geometric_Rotate",),
            ("geometric_Jigsaw",),
        ]),
    },
    "multi_aug": tune.choice([
        "meta_Sometimes",
        "meta_Sequential",
    ]),
    "algorithm": tune.choice(["tsn"]),
    "learning_rate": tune.uniform(0.0001, 0.001),
    "momentum": tune.uniform(0.9,0.99),
    "weight_decay": tune.uniform(5e-4,1e-3),
    "num_segments": tune.choice([8,16,32]),
}
```