IMPLEMENTATION OF CONVOLUTIONAL 
NEURAL NETWORKS ON FPGA 
FOR HUMAN ACTION RECOGNITION

A Thesis
Presented to the
Faculty of
California State Polytechnic University, Pomona

In Partial Fulfillment
Of the Requirements for the Degree
Master of Science
In
Electrical Engineering

By
Jonathan Hoong
2020
SIGNATURE PAGE

THESIS: IMPLEMENTATION OF CONVOLUTIONAL NEURAL NETWORKS ON FPGA FOR HUMAN ACTION RECOGNITION

AUTHOR: Jonathan Hoong

DATE SUBMITTED: Spring 2020

Department of Electrical Engineering

Dr. Mohamed E. Aly
Thesis Committee Chair
Electrical Engineering

Dr. Anas Salah Eddin
Electrical Engineering

Dr. Halima El Naga
Electrical Engineering
ACKNOWLEDGEMENTS

I would like to thank my parents who have believed in me and supported me every step of the way. I’d also like to express my gratitude to my advisor Dr. Mohamed El-Hadedy Aly for his support to achieve success with this thesis. I am grateful to Dr. Wen-Mei Hwu from University of Illinois, Urbana Champaign and Dr. Jinjun Xiong from IBM-C3SR for their guidance and feedback in this research. Finally, I’d like to thank my colleagues at Broadcom for their understanding and encouragement of my education. This research was made possible by Xilinx for supplying the PYNQ-Z1 boards.
ABSTRACT

Machine Learning is computer program in which has the capability to learn from experience acquired from performing some task with some performance measure. Deep learning, a subfield of ML has progressed to the stage of being able to classify objects and recognize human activity [2], [3]. Human action recognition (HAR) in videos is a difficult and challenging problem [8] that has been widely studied in computer vision [9]. Convolutional Neural Networks (CNNs), developed based on the brain’s visual cortex, is the ideal architecture for HAR due to its high performance on complex visual tasks.

Machine Learning research has predominately been focusing on learning algorithms in the past few years to invent ML models and applications where ML can be applied. In recent years, FPGAs have been used to deploy ML algorithms for training and inference. A PYNQ-Z1 board is a python-programmable FPGA-based System-on-Chip (SoC) that is used to deploy ML algorithms for video classification of HAR.
# TABLE OF CONTENTS

SIGNATURE PAGE .................................................................................................................. ii

ACKNOWLEDGEMENTS ....................................................................................................... iii

ABSTRACT ............................................................................................................................ iv

LIST OF TABLES .................................................................................................................. vii

LIST OF FIGURES ............................................................................................................... viii

1.0 INTRODUCTION ........................................................................................................ 1

1.1 Machine Learning .................................................................................................... 1

1.2 Deep Learning ......................................................................................................... 2

1.3 Convolutional Networks ......................................................................................... 3

2.0 RELATED WORKS .................................................................................................... 5

2.1 Human Action Recognition ..................................................................................... 5

2.2 FPGA - Hardware Acceleration ............................................................................... 5

3.0 TRAINING PROCESS ............................................................................................... 7

3.1 Tensorflow Training ............................................................................................... 7

3.2 Pytorch Training ..................................................................................................... 15

4.0 TENSORFLOW AND PYTORCH MODEL INFEERENCE USING GPU .......... 22

4.1 Inference using Tensorflow .................................................................................... 22

4.2 Inference using PyTorch ........................................................................................ 26

5.0 SOC ON PYNQ-Z1 ................................................................................................. 31
5.1 Tensorflow Implementation ................................................................. 31
5.2 PyTorch Implementation ................................................................. 31
5.3 Spark Cluster .................................................................................. 34
5.4 PYNQ Overlay ................................................................................ 36
6.0 RESULTS ....................................................................................... 38
7.0 FUTURE WORK ............................................................................. 40
8.0 CONCLUSION .............................................................................. 41
9.0 REFERENCES .............................................................................. 42
APPENDIX: CODE REPOSITORY .......................................................... 45
LIST OF TABLES

Table 1. Image Preprocessing Code ................................................................. 7
Table 2. Model instation and Unfreeze Layer .............................................. 8
Table 3. Train Model and Main Body ............................................................ 10
Table 4. 3D Convolutional Network ............................................................. 16
Table 5. Tensorflow Inference Code .............................................................. 22
Table 6. PyTorch Inference Code – Modified Version from Zhang .......... 26
Table 7. Worse Case Execution Time per ML Operation ............................... 36
Table 8. Performance of GPU and PYNQ-Z1 .............................................. 38
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.</td>
<td>Artificial Neural Network Architecture</td>
<td>2</td>
</tr>
<tr>
<td>Figure 2.</td>
<td>CNN Flow</td>
<td>3</td>
</tr>
<tr>
<td>Figure 3.</td>
<td>CNN layers with rectangular local perceptive fields</td>
<td>4</td>
</tr>
<tr>
<td>Figure 4.</td>
<td>Tensorflow/Keras Training: GPU Utilization</td>
<td>12</td>
</tr>
<tr>
<td>Figure 5.</td>
<td>Tensorflow/Keras Training: GPU Time Spent Accessing Memory</td>
<td>13</td>
</tr>
<tr>
<td>Figure 6.</td>
<td>Tensorflow/Keras Training: GPU Memory Usage</td>
<td>14</td>
</tr>
<tr>
<td>Figure 7.</td>
<td>Tensorflow/Keras Training: GPU Power Usage</td>
<td>15</td>
</tr>
<tr>
<td>Figure 8.</td>
<td>PyTorch Training: GPU Utilization</td>
<td>19</td>
</tr>
<tr>
<td>Figure 9.</td>
<td>PyTorch Training: GPU Time Spent Accessing Memory</td>
<td>19</td>
</tr>
<tr>
<td>Figure 10.</td>
<td>PyTorch Training: GPU Memory Allocation</td>
<td>20</td>
</tr>
<tr>
<td>Figure 11.</td>
<td>PyTorch Training: GPU Power Usage</td>
<td>21</td>
</tr>
<tr>
<td>Figure 12.</td>
<td>Tensorflow Inference: GPU Utilization</td>
<td>24</td>
</tr>
<tr>
<td>Figure 13.</td>
<td>Tensorflow Inference: GPU Time Spent Accessing Memory</td>
<td>24</td>
</tr>
<tr>
<td>Figure 14.</td>
<td>Tensorflow Inference: GPU Memory Allocated</td>
<td>25</td>
</tr>
<tr>
<td>Figure 15.</td>
<td>Tensorflow Inference: GPU Power Usage</td>
<td>25</td>
</tr>
<tr>
<td>Figure 16.</td>
<td>PyTorch Inference: GPU Utilization</td>
<td>28</td>
</tr>
<tr>
<td>Figure 17.</td>
<td>PyTorch Inference: GPU Time Spent Accessing Memory</td>
<td>29</td>
</tr>
<tr>
<td>Figure 18.</td>
<td>PyTorch Inference: GPU Memory Allocated</td>
<td>29</td>
</tr>
<tr>
<td>Figure 19.</td>
<td>PyTorch Inference: GPU Power Usage</td>
<td>30</td>
</tr>
<tr>
<td>Figure 20.</td>
<td>PyTorch Inference: CPU Utilization</td>
<td>32</td>
</tr>
<tr>
<td>Figure 21.</td>
<td>PyTorch Inference: Memory Utilization</td>
<td>33</td>
</tr>
</tbody>
</table>
1.0 INTRODUCTION

Research in Machine Learning (ML) is a hot topic that has developed rapidly in the past decade. Tensorflow, Keras, and PyTorch are some common developed APIs used in ML applications. Machine Learning is invaluable in industrial applications such as Internet of Things (IoT), autonomous self-driving vehicles, spam filter, etc. Research on ML algorithms in the past have primarily been deployed using graphics processing units (GPUs) and central processing unit (CPUs), however, field-programmable gate arrays (FPGAs) have gained popularity in ML applications.

1.1 Machine Learning

Machine Learning is computer program in which has the capability to learn from experience acquired from performing some task with some performance measure. It is great for problems in which solutions require a lot of hand tuning or long lists of rules, complex problems with no good solution using traditional approaches, fluctuating environments, or acquiring insights on complex problems and large amounts of data [1]. The main task in ML is to select a learning algorithm, and train it using a dataset. However, problems can arise if there are [1]:

- Insufficient quality of training data
- Nonrepresentative Training Data
- Poor-Quality Data
- Irrelevant Features
- Overfitting the Training Data
- Underfitting the Training Data
- Testing and Validating
1.2 Deep Learning

Deep learning, a subfield of ML has progressed to the stage of being able to classify objects and recognize human activity [2], [3]. Artificial Neural Networks (ANNs), modeled after the human brain, serve as the core of Deep Learning; they are versatile, power, and scalable making them ideal for highly complex ML tasks such as classification, speech recognition services, recommending videos on streaming services such as YouTube, or learning how to beat professional players in a game of Go [1]. Figure 1 depicts the basic architecture of ANNs, where the outputs of one layer act as the inputs of another layer [4]. Compared to the 1990s, ANNs have seen an increase in computing power partially due to Moore’s Law, but mainly because of the development of powerful GPUs.

Deep Neural Networks (DNNs) are ANNs with two or more hidden layers [1]. Convolutional Neural Networks (CNN/ConvNet) is a type of DNN that is widely used in computer vision computations.
1.3 Convolutional Networks

Convolutional Neural Networks, developed based on the brain’s visual cortex, have high performance on complex visual tasks such as image and video classification. Convolutional layers are the most important blocks of a CNN, where neurons are not connected every pixel in the image, but only pixels in their receptive fields [1]. Convolutional Neural Networks are great at performing tasks such as classification, object detection, segmentation [2]. The basic streamline flow of CNNs is represented in Figure 2.

Figure 2. CNN Flow [1]

Convolutional layers are the most important building blocks of a CNN: neurons in the first layer are connected to only pixels in their receptive fields, neurons in the second convolutional layer are connected to only neurons located within a small rectangle in the first layer, as shown in figure 3 [1]. The first hidden layer in a CNN architecture allows the network to extract low level features and assemble them into higher-level features in the next hidden layer making CNNs useful for image recognition [1] and visual feature extraction [5].
A filter, or convolution kernel, is a small image sized by the receptive field to represent a neuron’s weights with a typical size of 3x3, 5x5, or 7x7. A layer full of neurons using the same filters generates a feature map, which highlights the areas in an image that are most similar to the given filter. Within one feature map, all neurons share the same parameters (weights and bias), but different feature maps may have different parameters. A neuron’s receptive field extends across all previous layer’s feature maps, making it capable to detect multiple features anywhere in its inputs [1].

After a convolutional layer, pooling occurs to subsample the image so the computational load, memory usage, and number of parameters can be reduced. Similar to convolutional layers, the neurons in the pooling layer are connected to the outputs of a limited number of neurons in the previous layer. The typical receptive field typically has a size of 2x2. Pooling neurons are weightless, but aggregates the inputs using an aggregation function such as max or arithmetic average [6]. Machine Learning classification competitions, mainly ILSVRC, have developed many variants of the fundamental CNN architecture to improve classification accuracy. VGGNet and GoogLeNet (InceptionV1) are some architectures developed demonstrating the highest accuracy/lowest error rate [7].
2.0 RELATED WORKS

2.1 Human Action Recognition

Human action recognition (HAR) in videos is a difficult and challenging problem [8] that has been widely studied in computer vision [9]. Hand-crafted features and deep learning features for action recognition summarize the two categories in terms of feature learning based on previous studies. Hand-crafted features have been dominated in the past decade and describe local spatial-temporal variations in video. Deep learning features with CNNs have been widely applied in action recognition to extract translate-invariance feature in video frames, enabling good performance in action recognition.

The first CNN based method for action recognition was introduced by Karpathy’s group, whom also organized a large-scale sport video dataset for training CNNs [9]. In recent years, Zhou et al. [10] developed a two-streamed RNN/CNN for action recognition on 3D RGB videos. They propose an efficient two-stream RNN/CNN model that fused the capabilities of both RNN and CNN models with a 13% boost in accuracy over the RNN model. Similarly, in 2018, Wang et al. [9] adopts a two-stream 3-D convnet fusion for HAR with arbitrary size and lengths.

2.2 FPGA - Hardware Acceleration

GPUs have been the dominate machine learning hardware due to the optimization to compute intensive workloads and streaming memory modules [11]. GPUs also have higher memory bandwidth; Nvidia’s Tesla V100 claims to have 900 GB/s memory bandwidth whereas Intel’s Xeon E7 only has about 100 GB/s memory bandwidth.

Machine Learning research has predominately been focusing on learning algorithms in the past few years to invent ML models and applications where ML can be applied. However, the growth moving forward is in inference because the models have
been deployed in Cloud or Edge. Machine Learning algorithms are currently ran using a
GPU or CPU on cloud or edge to train and run inference on models, with a small portion
of researchers utilizing Field Programmable Gate Arrays (FPGAs) instead. Researchers
and large corporations such as Intel and Xilinx are looking towards FPGAs as an alternate
source to GPUs and CPUs to reduce power consumption and improve latency [12].

In recent years, FPGAs have been used to deploy ML algorithms for training and
Network (LRCN) using a HLS-based design flow for FPGAs. In [13] an Alexnet CNN
was implemented using a 5-layer accelerator for MNIST digit recognition task in Vivado
HLS. An FPGA accelerator for LSTM-RNN is designed by a UCLA group in [14] which
optimized both computational performance and communication requirements on a Xilinx
VC707 board.
3.0 TRAINING PROCESS

3.1 Tensorflow Training

Tensorflow is an open source platform designed to help develop and train ML models. It contains a wide variety of comprehensive tools, libraries, and community resources that lets researchers push the state-of-the-art in ML and developers, new or experience, easily build and deploy ML powered applications. Tensorflow offers robust ML production, ease of model building via high-level APIs such as Keras, and a simple yet flexible architecture for powerful experimentation for research [15].

Keras is a high-level neural networks API developed with a focus on enabling fast experimentation for research. Keras has the capability to run on top of Tensorflow, supports convolutional networks, and has pretrained models trained on ImageNet which can be used for prediction, feature extraction, and fine-tuning [16]. From ILSVRC, Google’s Inception architecture demonstrates high accuracy/low error rate [7]. Keras has applications available for InceptionV3, an enhanced version of InceptionV1, which is fine-tuned using the UCF101 dataset [16].

The training process, based on Harvey’s Five Video Classification Methods [17] uses the pretrained weights from Keras’ InceptionV3 model. In order to train a model using images, pre-processing work must be done on the dataset using keras’ image processing module, shown in table 1 below.

Table 1. Image Preprocessing Code

```python
def get_generators():
    train_datagen = ImageDataGenerator(
        rescale=1./255,
        shear_range=0.2,
```
This process is done to prevent the model from seeing two of the exact same images and help prevent overfitting of the data. The next function required in to train a CNN model is to get the pretrained inception model and unfreeze the top layers for fine-tuning.

Table 2. Model Instatiation and Unfreeze Layer

```python
def get_model(weights='imagenet'):
    # create the base pre-trained model
```
base_model = InceptionV3(weights=weights, include_top=False)

# add a global spatial average pooling layer
x = base_model.output
x = GlobalAveragePooling2D()(x)

# let's add a fully-connected layer
x = Dense(1024, activation='relu')(x)

# and a logistic layer
predictions = Dense(len(data.classes), activation='softmax')(x)

# this is the model we will train
model = Model(inputs=base_model.input, outputs=predictions)
return model

def freeze_all_but_top(model):
    for layer in model.layers[:249]:
        layer.trainable = False
    for layer in model.layers[249:]:
        layer.trainable = True

The code shown in table 2 is based on Keras’ Applications documentation [16] for fine-tuning new classes using InceptionV3. In order to fine-tune the ImageNet model for action recognition, the bottom layers remain frozen and the remaining top layers are trained.
```python
def train_model(model, nb_epoch, generators, callbacks=[]):
    train_generator, validation_generator = generators
    model.fit_generator(
        train_generator,
        steps_per_epoch=100,
        validation_data=validation_generator,
        validation_steps=10,
        epochs=nb_epoch,
        callbacks=callbacks)
    return model

def main():
    model = get_model()
    generators = get_generators()

    model = freeze_all_but_top(model)
    model = train_model(model, 50, generators,
        [checkpointer, early_stopper, tensorboard])

def main(weights_file):
    # Helper: Save the model.
    checkpoint = ModelCheckpoint(
        filepath=os.path.join('data', 'checkpoints', 'inception.{epoch:03d}-{val_loss:.2f}.hdf5'),
        verbose=1,
        save_best_only=True)

    # Helper: Save the model.
    checkpoint = ModelCheckpoint(
        filepath=os.path.join('data', 'checkpoints', 'inception.{epoch:03d}-{val_loss:.2f}.hdf5'),
        verbose=1,
        save_best_only=True)

        filepath=os.path.join('data', 'checkpoints', 'inception.{epoch:03d}-{val_loss:.2f}.hdf5'),
        verbose=1,
        save_best_only=True)
```
When using Harvey’s algorithm to train a CNN model, the GPU’s VRAM bottlenecks the compute performance and increases the time required to train a model. The GPU isn’t fully utilized when training the module, however, the memory usage is consistently at or near 100% usage. Figures 4-7 illustrates the GPU resource usage required to train an InceptionV3 CNN model using Keras and Tensorflow.
Figure 4. Tensorflow/Keras Training: GPU Utilization
Figure 5. Tensorflow/Keras Training: GPU Time Spent Accessing Memory
Figure 6. Tensorflow/Keras Training: GPU Memory Usage
3.2 Pytorch Training

PyTorch, developed primarily by Facebook, is another open source ML framework that enables fast, flexible experimentation and efficient production through a user-friendly front-end, distributed training, and ecosystem of tools and libraries [19].

Zhang’s PyTorch video recognition [20] repository was used to acquire the python code to train a CNN network. A 3D Convolutional Network (C3D) is used in Zhang’s repository for video human action recognition. C3D networks inherently apply convolutions and max pooling in 3D space, where the third dimension is time. Table 4 represents the typical algorithm for 3D convolutional networks.
def __init__(self, num_classes, pretrained=False):
    super(C3D, self).__init__()

    self.conv1 = nn.Conv3d(3, 64, kernel_size=(3, 3, 3), padding=(1, 1, 1))
    self.pool1 = nn.MaxPool3d(kernel_size=(1, 2, 2), stride=(1, 2, 2))

    self.conv2 = nn.Conv3d(64, 128, kernel_size=(3, 3, 3), padding=(1, 1, 1))
    self.pool2 = nn.MaxPool3d(kernel_size=(2, 2, 2), stride=(2, 2, 2))

    self.conv3a = nn.Conv3d(128, 256, kernel_size=(3, 3, 3), padding=(1, 1, 1))
    self.conv3b = nn.Conv3d(256, 256, kernel_size=(3, 3, 3), padding=(1, 1, 1))
    self.pool3 = nn.MaxPool3d(kernel_size=(2, 2, 2), stride=(2, 2, 2))

    self.conv4a = nn.Conv3d(256, 512, kernel_size=(3, 3, 3), padding=(1, 1, 1))
    self.conv4b = nn.Conv3d(512, 512, kernel_size=(3, 3, 3), padding=(1, 1, 1))
    self.pool4 = nn.MaxPool3d(kernel_size=(2, 2, 2), stride=(2, 2, 2))

    self.conv5a = nn.Conv3d(512, 512, kernel_size=(3, 3, 3), padding=(1, 1, 1))
    self.conv5b = nn.Conv3d(512, 512, kernel_size=(3, 3, 3), padding=(1, 1, 1))
    self.pool5 = nn.MaxPool3d(kernel_size=(2, 2, 2), stride=(2, 2, 2), padding=(0, 1, 1))

    self.fc6 = nn.Linear(8192, 4096)
    self.fc7 = nn.Linear(4096, 4096)
    self.fc8 = nn.Linear(4096, num_classes)
self.dropout = nn.Dropout(p=0.5)
self.relu = nn.ReLU()
self.__init_weight()

def forward(self, x):
    x = self.relu(self.conv1(x))
    x = self.pool1(x)

    x = self.relu(self.conv2(x))
    x = self.pool2(x)

    x = self.relu(self.conv3a(x))
    x = self.relu(self.conv3b(x))
    x = self.pool3(x)

    x = self.relu(self.conv4a(x))
    x = self.relu(self.conv4b(x))
    x = self.pool4(x)

    x = self.relu(self.conv5a(x))
    x = self.relu(self.conv5b(x))
    x = self.pool5(x)

    x = x.view(-1, 8192)
x = self.relu(self.fc6(x))

x = self.dropout(x)

x = self.relu(self.fc7(x))

x = self.dropout(x)

logits = self.fc8(x)

return logits

Running PyTorch using Zhang’s method on PyTorch removes the VRAM bottleneck seen when using Tensorflow to train a CNN model. Although the training time is almost 3 times longer than Harvey’s method to training a CNN, Zhang’s method has approximately 4 times more epochs due to the lack of an early stopper. Figures 8-11 illustrates the GPU resource usage required to train a C3D model in PyTorch.
GPU Utilization (%)

Figure 8. PyTorch Training: GPU Utilization

GPU Time Spent Accessing Memory (%)

Figure 9. PyTorch Training: GPU Time Spent Accessing Memory
Figure 10. PyTorch Training: GPU Memory Allocation
Figure 11. PyTorch Training: GPU Power Usage
4.0 TENSORFLOW AND PYTORCH MODEL INFEERENCE USING GPU

Researchers are looking for ways to deploy Machine Learning models onto more power conservative devices. As algorithms have been fine-tuned over the past decade, ML growth is headed towards inference. Some of the challenges associated with inference are [12]:

- Rate of AI innovation
- Performance at low latency
- Lower power consumption
- Whole app acceleration

The resources used while running inference on both the Tensorflow/Keras model and PyTorch model on a Nvidia RTX 2070 GPU are significantly lower than the required resources to train a model.

4.1 Inference using Tensorflow

To run inference using the Tensorflow/Keras trained model, a webcam connection is required. Inference is ran using real-time live stream video from the webcam using a frame-by-frame approach. The input of the webcam is broken into frames and fed into the model. The model outputs the predicted action, which is written on top of the current frame along with the frames per second (FPS). Table 5 shows a modified version of Kumaar’s [18] inference script. From figures 12-15, it is apparent that there are no bottlenecks on the GPU when running inference on this model.

Table 5. Tensorflow Inference Code

```
logger.debug('cam read+')
cam = cv2.VideoCapture(args.camera)
```
ret_val, image = cam.read()

logger.info('cam image=%dx%d' % (image.shape[1], image.shape[0]))

# count = 0

while True:
    logger.debug('+image processing+')
    ret_val, image = cam.read()
    logger.debug('+classification+')

    # Classification
    scene_class = label_img_scene.classify(image)
    logger.debug('+displaying+)

    cv2.putText(image, "FPS: \(\frac{1}{t}\) % (1.0 / (time.time() - fps_time)), (20, 60),
        cv2.FONT_HERSHEY_SIMPLEX, 0.6,
        (0, 0, 255), 1)
    cv2.putText(image, "Predicted Scene: \(\text{\textit{str}}\) \(\text{\textit{str}}\) %scene_class),
        (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
        (0, 255, 0), 2)
    cv2.imshow('tf-pose-estimation result', image)
Figure 12. Tensorflow Inference: GPU Utilization

Figure 13. Tensorflow Inference: GPU Time Spent Accessing Memory
Figure 14. Tensorflow Inference: GPU Memory Allocated

Figure 15. Tensorflow Inference: GPU Power Usage
4.2 Inference using PyTorch

Inferencing on PyTorch follows the same concept as running inference on Tensorflow: a webcam feeds real-time live video and goes to the PyTorch model frame by frame. The key difference between PyTorch and Tensorflow models is the memory utilization required for PyTorch is almost half the memory required to run inference on the Tensorflow model. Also, the PyTorch model is utilizing the GPU more than the Tensorflow model, which can be attributed to the additional third dimension convolutions that need to be processed in the PyTorch model.

Table 6. PyTorch Inference Code – Modified Version from Zhang

```python
cap = cv2.VideoCapture(0)
retaining = True

clip = []

while retaining:
    retaining, frame = cap.read()

    if not retaining and frame is None:
        continue

    tmp_ = center_crop(cv2.resize(frame, (171, 128)))

    tmp = tmp_ - np.array([[90.0, 98.0, 102.0]])

    clip.append(tmp)

    if len(clip) == 16:
        inputs = np.array(clip).astype(np.float32)

        inputs = np.expand_dims(inputs, axis=0)

        inputs = np.transpose(inputs, (0, 4, 1, 2, 3))
```

26
inputs = torch.from_numpy(inputs)

inputs = torch.autograd.Variable(inputs, requires_grad=False).to(device)

with torch.no_grad():
    outputs = model.forward(inputs)

probs = torch.nn.Softmax(dim=1)(outputs)
label = torch.max(probs, 1)[1].detach().cpu().numpy()[0]

cv2.putText(frame, class_names[label].split(' ')[-1].strip(), (20, 20), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 255), 1)

cv2.putText(frame, "prob: %.4f" % probs[0][label], (20, 40), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 255), 1)

cv2.putText(frame, "FPS: %f" % (1.0 / (time.time() - fps_time)), (20, 60), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 255), 1)

fps_time = time.time()

cv2.imshow('result', frame)

cv2.imshow('result', frame)
Figure 16. PyTorch Inference: GPU Utilization
Figure 17. PyTorch Inference: GPU Time Spent Accessing Memory

Figure 18. PyTorch Inference: GPU Memory Allocated
Figure 19. PyTorch Inference: GPU Power Usage
5.0 SOC ON PYNQ-Z1

PYNQ-Z1 is a python-programmable FPGA-based System-on-Chip (SoC) containing a 32-bit ARM Cortex-A9 processor. It is part of the Artix-7 family and uses the Zynq-7000 SoC as a hardware platform for PYNQ open-source framework. The PYNQ framework runs on petalinux – an Embedded Linux System Development Kit based on Ubuntu and has the capability to utilize FPGA resources available on the board through the use of overlays in python, making it the ideal board of choice for this paper. Petalinux has many restrictions on what repositories/software can be downloaded. Docker, a tool used to build python modules from source, supports only 64-bit architectures causing issues to arise when attempting to build from ML APIs from source.

5.1 Tensorflow Implementation

Google’s Tensorflow, one of the largest ML APIs, fully supports 64-bit architectures. Although there is no official 32-bit support for Tensorflow, it is still possible to install Tensorflow on a PYNQ-Z1 board.

PYNQ-Z1’s 32-bit architecture causes large amount of issues to arise due to the incompatibility with many ML APIs. Tensorflow, one of the largest ML APIs, is currently supporting 64-bit architectures. It can be installed on the PYNQ-Z1 board using a pre-built wheel or building from source. However, the lack of 32-bit support from Tensorflow can cause instructional issues when working on complex ML algorithms.

5.2 PyTorch Implementation

Since Tensorflow isn’t able to fully operate on the PYNQ-Z1 board, PyTorch is the alternative API used to deploy ML models for inferencing. Vohra [21] has provided a PyTorch 1.2.0 source that is used to build and install PyTorch on PYNQ OS v2.4 or higher. Inferencing on only the PYNQ OS has extremely poor performance. Figures 20-22
illustrates the resource usage on the PYNQ board. The CPU utilization is constantly jumping from approximately 22% to 100% load, whereas the memory utilization ranges from 38% to 88%, suggesting the CPU may be the bottleneck in the system. Running ML algorithms on the ARM processor itself has worse performance than the GPU and CPU. Clusters and hardware acceleration via custom IP are two approaches that should improve inference performance on the PYNQ board.

Figure 20. PyTorch Inference: CPU Utilization
Figure 21. PyTorch Inference: Memory Utilization
5.3 Spark Cluster

A cluster of PYNQ boards are used in attempt to accelerate ML computations. Apache Spark was chosen to submit the code into the cluster, which is managed by Yarn. Hadoop Yarn is responsible for resource and node management. Figure 23 illustrates the Spark flow, where Hadoop Yarn is the cluster manager running on top of Spark.
While executing the code, the webcam has to be connected to the worker nodes. When attempting to leave the webcam connected to the master node, the workers are unable to access the webcam for data streaming. An alternative attempt to resolve this issue was to create a master python program to collect webcam data in 10 second intervals, save the clips as a temporary .avi file, and upload the data to the cluster. The issue with this process, however, is that the spark cluster has to be executed every time, causing all the files to have to be resubmitted to the cluster. Submitting jobs to the cluster is inefficient because all of the included files have to be re-uploaded to the cluster, including the model.

To continuously stream data from the master node for the worker nodes to use, Kafka could be used to replace Yarn. Kafka is a distributed streaming platform with the ability to build real-time streaming data pipelines to transmit data between the master and workers nodes [22]. It allows applications to react to the streams of data, which would be perfect for continuously re-uploading just the webcam data into the cluster. If more time permitted, I would have changed my cluster manager from Yarn to Kafka.
The cluster is executed using a linux OS as the master, and two PYNQ-Z1 boards as worker nodes. Running inference using a spark cluster shows no performance improvement. A possible reason for the lack of performance improvement could be attributed to the python code; the python script to run inference on the ML model is not optimized for clusters, negating the benefits of using a cluster of pynq boards to accelerate inference. The results of running inference on a cluster extremely similar to running inference on one, indicating the code is running on one PYNQ board instead of splitting the resources.

5.4 PYNQ Overlay

The main feature of PYNQ ability to run python scripts on Petalinux and integrate FPGA resources through the use of overlays. Overlays are programmed either in HLS or Vivado to generate a .bit file which can be called in Python. Xilinx has a wide variety of IP integrations that can be used, however, there is currently no IP available for 3D convolution.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loading Model</td>
<td>~80</td>
</tr>
<tr>
<td>Convolution</td>
<td>~20</td>
</tr>
<tr>
<td>Pooling</td>
<td>~0.1</td>
</tr>
<tr>
<td>Fully Connected Layer</td>
<td>~20</td>
</tr>
</tbody>
</table>

From Table 7, the inference could be sped up if by accelerating the loading time of the model. The model could be loaded into a look up table (LUT) to improve the access time of the model objects. Hls4ml is a group of collaborators on github creating a package
for machine learning inference in FPGAs [23]. They support a variety of ML architectures/toolkits such as Keras/Tensorflow, PyTorch, and scikit-learn, however, there is no support for CNNs on PyTorch.

Another method to improving inference time is to create a custom IP on the convolutional layers. In the Convolutional 3D network, shown in table 4, there are eight convolution operations: reducing the convolution time has the most significant impact on improving inference performance. On github, Athanasiadis [24] has created a 3D convolutional algorithm on hls using Fourier transforms. Currently, the code isn’t synthesizable on hls to create a custom IP.
6.0 RESULTS

Two models were trained in this exercise due to the lack of API compatibility using Tensorflow on all PYNQ OS. Inferencing using Google’s Tensorflow API shows worse performance on the GPU relative to Facebook’s PyTorch API, seen in table 8, which may be attributed to the number of neurons in the Tensorflow model. Although the utilization is lower when running Tensorflow than PyTorch, the layer count in the Tensorflow model is significantly larger than the PyTorch model. InceptionV3 has over 200 layers, each with millions of neurons, whereas PyTorch only has 8 layers.

<table>
<thead>
<tr>
<th>Table 8. Performance of GPU and PYNQ-Z1</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
</tr>
<tr>
<td>Device</td>
</tr>
<tr>
<td>Power</td>
</tr>
<tr>
<td>FPS</td>
</tr>
<tr>
<td>Device Utilization</td>
</tr>
<tr>
<td>Memory Usage</td>
</tr>
<tr>
<td>Power Consumption</td>
</tr>
</tbody>
</table>

I used PYNQ OS v2.5 running on PyTorch v1.2.0 API to run inference using the PyTorch 3D Convolutional Network. While the performance of the PYNQ board is significantly worse than that of the GPU, approximately 1800 times worse, it consumes 24.5 times less power relative to a Nvidia RTX 2070. Performance improvements could also be made to improve the FPS if the FPGA resources available on the PYNQ-Z1 were fully utilized. Currently, running inference on the PYNQ-Z1 board is purely using the ARM processor onboard. Better performance could be acquired by acceleration the convolution operations
via hardware acceleration and utilizing BRAM in addition to DDR3 memory to reduce swap memory activity.
7.0 FUTURE WORK

The current framework for getting PyTorch to run on a PYNQ-Z1 board and running inference on it has been successful. However, the performance can be significantly improved by generating custom IPs in hls to create overlays. Currently, the python inference script and PyTorch module installed is not using any of the FPGA resources available on the board. The main advantage to a PYNQ-Z1 board is the ability to utilize FPGA resources while running python scripts in Petalinux.

In addition to generating a custom IP and creating overlays, the inference code can still be optimized for clusters, and different cluster managers can be used to improve the cluster flow. The current inference code does not have any optimization done to improve compatibility with clusters, and tools such as PySpark can be used to help with resource management in the python code. Kafka cluster on top of Apache Spark looks like an interesting approach to streaming data from the webcam into the cluster, removing the need to attach webcams to the worker nodes.
8.0 CONCLUSION

Machine Learning algorithm for human action recognition can successfully be deployed on a portable FPGA device. A PyTorch model can be deployed on a PYNQ-Z1 to run inference on real-time live video feed from a webcam connected to the USB port. Although inferencing can be deployed on a PYNQ-Z1, the current code is not feasible for deployment due to the poor performance. Inferencing on the PYNQ-Z1 board still needs to be further optimized to utilize free FPGA resources for single-node performance or improve cluster management for better performance in a cluster.
9.0 REFERENCES


APPENDIX: CODE REPOSITORY
The code repository can be found at https://github.com/Reconfigurable-Computing-
CalPoly-Pomona/human_action_recognition-