Golf Vision

WES-237B AI Project

Yi Song
Sebastian Nevarez

Table of Contents

Introduction 3
Method: Environment Set-up 5
Method: Code Execution 7
Results 11
Conclusion & Future Work 14
Appendix: References 16

Introduction

The game of golf is an endless pursuit of perfection. To that end, players are always looking for ways to improve their game through coaching and data analytics. The latest trends in teaching technology revolve around video analysis of a player's swing and providing them with swing path, clubhead speed, ball speed, distance, and a number of other data points to help them improve their game. Unfortunately, most players do not have access to 1:1 coaching that can provide them with the information they need to continue growing. This presents an opportunity for AI to offer players with 1:1 coaching so they can continue their pursuits of perfection.

The goals of this project are to create a video swing analysis using AI that will return a shot type (see Table 1 for the nine types) when a "down-the-line" swing input (see Image 1) is provided. For a "face-on" swing input (see Image 2), the system will return a swing type (see Table 2 for the three types) based on the player's hand position.

High-Draw	High-Straight	High-Fade
Medium-Draw	Medium-Straight	Medium-Fade
Low-Draw	Low-Straight	Low-Fade

Table 1: The nine shots

Hand Position	Swing Type
Hip-level	Chip
Between hip and shoulder-level	Pitch
Shoulder-level	Full-swing

Table 2: The three swings

Image 1: Down-the-line input type

Image 2: Face-on input type

To begin our project we will use the trt_pose project (see Reference 1 in the Appendix) as our starting point. Our plan is to build from this project following a similar pattern used to develop the trt_pose_hand project (see Reference 2 in the Appendix). We will reverse engineer the two projects to understand the underlying framework, and then apply transfer learning to our Golf Vision project to analyze swings and golf shots.

Method

Environment Set-up

To begin our environment set-up, the first thing we need to do is import the json package to describe the human pose task. The human pose is being divided into two parts: keypoints and skeleton. There are 18 keypoints consisting of facial features, shoulder, wrist, hip, knee and ankle. These keypoints are defined in a two-dimensional space as coordinates, whereas the skeleton outlines a human body using the connections between the keypoints.

Next, we import the torch package to import the PyTorch library into our code. PyTorch is an open-source deep learning framework primarily used for building and training neural networks. PyTorch provides us with a multi-dimensional array called a "tensor" that is similar to NumPy arrays but designed to work efficiently with GPUs. Tensors can be created for data storage and manipulation. PyTorch allows us to define and train neural networks with ease. We can create neural network layers, define custom architectures, and train models for tasks like image classification, natural language processing, and more.

Next, import the torch2trt package to convert PyTorch to TensorRT utilizing the TensorRT Python API. TensorRT (Tensor Runtime) is a high-performance deep learning inference library developed by NVIDIA. It is designed to optimize and accelerate the inference phase of deep neural networks, particularly those built using frameworks like TensorFlow and PyTorch. TensorRT is primarily focused on running inference on GPUs, making it well-suited for applications that require real-time or low-latency processing, such as autonomous vehicles, robotics, video analytics, and more. TensorRT uses various optimization techniques to accelerate inference. It performs layer fusion, precision calibration (such as reduced precision like FP16), and kernel auto-tuning to maximize the use of GPU resources and minimize latency.

Next, import cv2, torchvision.transforms, and PIL.Image to load and save images captured by onboard camera and perform preprocessing and augmentation on the images. OpenCV is a popular open-source computer vision library that provides a wide range of tools and functions for working with images and videos. It is widely used in computer vision and image processing applications. Torchvision is a package that provides access to popular datasets, model architectures, and image transformations for computer vision tasks. The *transforms* module, in particular, is used for performing various image preprocessing and augmentation operations on images. Pillow is a popular Python library for working with images. The Image module in Pillow provides classes and functions for creating, opening, manipulating, and saving images in various formats.

In our project we also use several packages from the trt_pose project. Let’s take a look at them one by one.

First, we import the trt_pose.coco package to help us detect common objects in context. COCO is a large-scale object detection, segmentation, and captioning dataset. The COCO

dataset has 330K images with more than 200k labels and 1.5 million object instances. Regarding human pose detection, it has 250,000 human poses with keypoints.

Second, we import the trt_pose.model package. This library contains predefined models and functions related to pose estimation using TensorRT. We initialize a pose estimation model using the specified architecture (resnet18_baseline_att) which will be configured based on the number of keypoints and links.

Third, we import the trt_pose.parse_objects and trt_pose.draw_objects to parse the objects from the neural network, as well as draw the parsed objects on an image.

Last, we import the jetcam.csi_camera package to utilize the CSI camera on the JETSON TX2 board. The package can easily read images as numpy arrays with image = camera.read() and set the camera to running to True to attach callbacks to new frames.

In summary, we have used packages of various functionalities ranging from image processing, feature description, deep learning and its optimization as well as some NVIDIA developed libraries of human pose estimation.

Code Execution

To begin the code execution, we first load the JSON file which describes the human pose task. This is in COCO format, it is the category descriptor pulled from the annotations file. We modify the COCO category slightly, to add a neck keypoint. We will use this task description JSON to create a topology tensor, which is an intermediate data structure that describes the part linkages, as well as which channels in the part affinity field each linkage corresponds to.

```
import json
import trt_pose.coco

with open('human_pose.json', 'r') as f:
    human_pose = json.load(f)

topology = trt_pose.coco.coco_category_to_topology(human_pose)
```

Next, we load our model. Each model takes at least two parameters, *cmap_channels* and *paf_channels* corresponding to the number of heatmap channels and part affinity field channels. The number of part affinity field channels is 2x the number of links, because each link has a channel corresponding to the x and y direction of the vector field for each link.

```
import trt_pose.models

num_parts = len(human_pose['keypoints'])
num_links = len(human_pose['skeleton'])
# print (num_parts)
# print (num_links)

model = trt_pose.models.resnet18_baseline_att(num_parts, 2 * num_links).cuda().eval()
```

Then, we load the model weights.

```
import torch

MODEL_WEIGHTS = 'resnet18_baseline_att_224x224_A_epoch_249.pth'

model.load_state_dict(torch.load(MODEL_WEIGHTS))
```

In order to optimize with TensorRT using the python library *torch2trt* we also need to create some example data. The dimensions of this data should match the dimensions that the network was trained with. Since we're using the resnet18 variant that was trained on an input resolution of 224x224, we set the width and height to these dimensions. We use torch2trt to

optimize the model. We also enable fp16_mode to allow optimizations to use reduced half precision.

```
WIDTH = 224
HEIGHT = 224

data = torch.zeros((1, 3, HEIGHT, WIDTH)).cuda()
```

```
import torch2trt

model_trt = torch2trt.torch2trt(model, [data], fp16_mode=True, max_workspace_size=1<<25)
```

The optimized model can be saved so we don't need to perform optimization again, we can just load the model. One thing to note is that TensorRT has device specific optimizations, so we can only use an optimized model on similar platforms. Also, we could then load the saved model using the TRTModule in *torch2trt* as follows.

```
OPTIMIZED_MODEL = 'resnet18_baseline_att_224x224_A_epoch_249_trt.pth'

torch.save(model_trt.state_dict(), OPTIMIZED_MODEL)
```

```
from torch2trt import TRTModule

model_trt = TRTModule()
model_trt.load_state_dict(torch.load(OPTIMIZED_MODEL))
```

We use the image processing packages to define a function that will preprocess the image, which is originally in BGR8 / HWC format, and prepare it for deep learning tasks. Inside the preprocess function, global device declares a global variable device, which is later set to the CUDA device. The input image is converted from the BGR color space (commonly used in computer vision) to the RGB color space. Then, it is converted from a NumPy array to a Pillow (PIL) Image object. Finally it converts the PIL Image object to a PyTorch tensor using the transforms.functional.to_tensor function. It also moves the resulting tensor to the CUDA device.

```
import cv2
import torchvision.transforms as transforms
import PIL.Image

mean = torch.Tensor([0.485, 0.456, 0.406]).cuda()
std = torch.Tensor([0.229, 0.224, 0.225]).cuda()
device = torch.device('cuda')

def preprocess(image):
    global device
    device = torch.device('cuda')
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    image = PIL.Image.fromarray(image)
    image = transforms.functional.to_tensor(image).to(device)
    image.sub_(mean[:, None, None]).div_(std[:, None, None])
    return image[None, ...]
```

Here we'll define two callable classes that will be used to parse the objects from the neural network, as well as draw the parsed objects on an image. The following jetcam code will initialize the camera, set its image dimension as well as frames per second. A widget is created which will be used to display the camera feed with visualizations.

```
from trt_pose.draw_objects import DrawObjects
from trt_pose.parse_objects import ParseObjects

parse_objects = ParseObjects(topology,cmap_threshold=0.12, link_threshold=0.15)
draw_objects = DrawObjects(topology)
```

```
# from jetcam.usb_camera import USBCamera
from jetcam.csi_camera import CSICamera
from jetcam.utils import bgr8_to_jpeg

# camera = USBCamera(width=WIDTH, height=HEIGHT, capture_fps=30)
camera = CSICamera(width=WIDTH, height=HEIGHT, capture_fps=30)

camera.running = True
```

```
import ipywidgets
from IPython.display import display

image_w = ipywidgets.Image(format='jpeg')

display(image_w)
```

In the main execution loop, we first get the new image from the camera stream and then preprocess the image. The preprocessed data is then fed into the neural network to get cmap

and paf, which are confidence maps and part affinity fields used in pose estimation tasks. The output is then fed into the parse_object function defined in trt_pose to get the number of objects detected in the image, object keypoints and normalized peaks. Finally the object information is visualized as keypoints and skeleton on the output jpeg image streaming on the widget. If we call the cell of the execute function, it will execute the function once on the current camera frame. Calling the cell with camera.observe() will attach the execution function to the camera's internal value. This will cause the execute function to be called whenever a new camera frame is received. Calling the cell with camera.unobserve_all() will unattach the camera frame callbacks.

```
def execute(change):
    image = change['new']
    data = preprocess(image)
#     print (data)
    cmap, paf = model_trt(data)
    cmap, paf = cmap.detach().cpu(), paf.detach().cpu()
    counts, objects, peaks = parse_objects(cmap, paf)
#     print (peaks)
#     joints = preprocessdata.joints_inference(image, counts, objects, peaks)
#     print((joints))
#     dist_bn_joints = preprocessdata.find_distance(joints)
#     print((dist_bn_joints))
    draw_objects(image, counts, objects, peaks)
    image_w.value = bgr8_to_jpeg(image[:, ::-1, :])
```

```
execute({'new': camera.value})
```

...

```
camera.observe(execute, names='value')
```

...

```
camera.unobserve_all()
```

```
camera.running = False
```

Results

After successfully running the above code, we can see that the code performs as we would expect. Taking camera streams, preprocessing the images, running the image in the neural network in real-time, and drawing the keypoints and skeletons on the person in frame. The results are demonstrated below through a series of screenshots of the camera stream. In the sample outputs of *trt_pose*, we see the green keypoints (18 total) displayed in the correct areas and the skeleton (21 total) drawn between the points showing the outline of a person in a swinging motion. In the sample outputs of *trt_pose_hand*, we see the green keypoints and skeleton outlining hand gestures. The models are pretrainied based on the model in trt_pose suitable for hand pose classification.

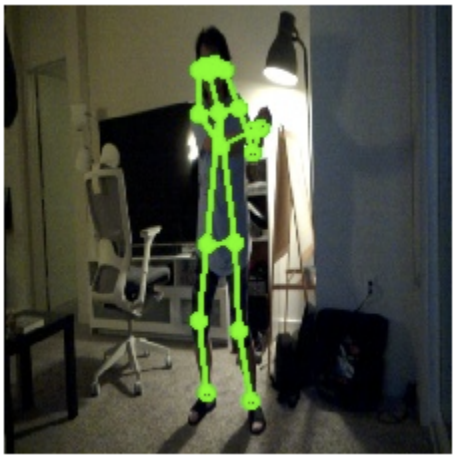

Image 3: Sample output 1 of trt_pose

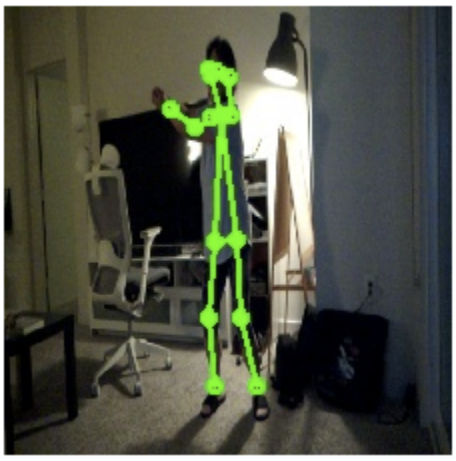

Image 4: Sample output 2 of trt_pose

The following images are screenshots of our reproduction of the trt_pose_hand project.

Image 5: “Stop” recognized

Image 6: “Fist” recognized

Image 7: “OK” recognized

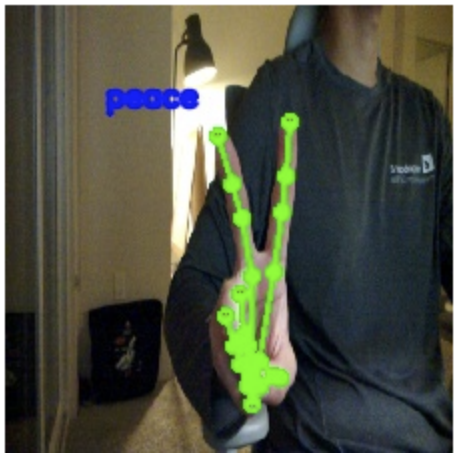

Image 8: “Peace” recognized

Image 9: “Pan” recognized

Conclusion & Future Work

We started this project with the goal of building an AI golf swing analysis tool to aid golfers in a 1:1 manner to develop their games. The system was envisioned to accept two forms of video input and return either a shot type or a swing type (return based on the input, see below).

- Input: “Down-the-line” swing video (Image 1) → Return: Shot type (Table 1)
- Input: “Face-on” swing video (Image 2) → Return: Swing type (Table 2)

Our strategy to complete this project was to begin by reverse engineering the trt_pose (Reference 1) and trt_pose_hand (Reference 2) NVIDIA GitHub projects. From there we would apply transfer learning to our swing analysis system to render skeletons on our video inputs. Through a series of post-processing operations, our AI would be able to return either the swing or shot type uploaded.

Our team was able to successfully reverse engineer both GitHub projects and reproduce the results, as detailed in the Results section of this paper. As we continue beyond WES-237B, our team has outlined the following plan to continue development of our AI system.

We will begin by splitting development into two phases. The first phase will focus on processing a face-on input. For this portion of the system we will be able to reuse some of the code that is provided by the trt_pose project as the face-on angle presents an entire view of the frontal skeleton so we can use the same human pose json to define the golf swing task.

A face-on input will return a swing type which can be determined by following the criteria outlined in Table 2. To determine where the hands are relative to hip or shoulder level we can determine a normalized hip and shoulder joint by finding the center point of the left and right joints. Since both hands will be holding the golf club and will be at similar positions throughout the swing, the two hand joints will be normalized into one hand joint. Once all joints have been properly defined, our AI will be able to follow the hands through the entire swing.

Next, we move to processing down-the-line inputs. For this phase, we will have to create a new json file that will be a combination of the human pose and hand pose json from trt_pose_hand. This type of input allows for only a limited view of the human skeleton so more emphasis will have to be placed on the joints of the hands for the AI to follow throughout the entire swing. We will retrain our updated model using the same model weights used in trt_pose and trt_pose_hand because the tasks defined in those projects are similar to the golf swing task we are defining.

In trt_pose_hand, our team identified the following code that calculates the distance between two joints.

```
def execute(change):
    image = change['new']
    data = preprocess(image)
    cmap, paf = model_trt(data)
    cmap, paf = cmap.detach().cpu(), paf.detach().cpu()
    counts, objects, peaks = parse_objects(cmap, paf)
    joints = preprocessdata.joints_inference(image, counts, objects, peaks)
    draw_joints(image, joints)
    #draw_objects(image, counts, objects, peaks)
    dist_bn_joints = preprocessdata.find_distance(joints)
    gesture = clf.predict([dist_bn_joints,[0]*num_parts*num_parts])
    gesture_joints = gesture[0]
    preprocessdata.prev_queue.append(gesture_joints)
    preprocessdata.prev_queue.pop(0)
    preprocessdata.print_label(image, preprocessdata.prev_queue, gesture_type)
    image_w.value = bgr8_to_jpeg(image)
```

We plan to update this code to measure the distance between the hip joint and the hands at any point in the swing. Using the same normalization strategy as in the face-on input, our AI will be able to follow the player's hands through the swing and tell what type of shot is being performed.

A shot type can be broken up into two parts - where the hands are before contact vs after contact. Before contact, we can tell whether the shot is a draw, straight, or fade based on this criteria:

- For a draw type shot, we would expect to see the distance between the hands and hips increase as the hands are moving towards hitting the golf ball due to the hands moving in an in-to-out trajectory
- For a straight type shot, we would expect to see the distance between the hands and hips remain steady due to the hands moving parallel to the hips along the trajectory line
- For a fade type shot, we would expect to see the distance between hands and hips decrease as the hands are moving towards hitting the golf ball due to the hands moving in an out-to-in trajectory

After contact, we can tell whether the shot is high, medium, or low based on this criteria:

- For a high type shot, we would expect to see a quick growth in the distance as a player would be exaggerating their swing motion upwards to finish high
- For a medium type shot, we would expect to see a linear growth in the distance as a player is following a swing motion outwards rather than up or down
- For a low type shot, we would expect to see a slow growth in the distance as the player would be exaggerating their swing motion downwards to stay a low as they can as long as they can in the swing

Appendix

References

Reference 1: https://github.com/NVIDIA-AI-IOT/trt_pose

Reference 2: https://github.com/NVIDIA-AI-IOT/trt_pose_hand

Reference 3: https://github.com/ZheC/Realtime_Multi-Person_Pose_Estimation

- *Cao, Zhe, et al. "Realtime multi-person 2d pose estimation using part affinity fields." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2017.*

Reference 4: https://github.com/Microsoft/human-pose-estimation.pytorch

- *Xiao, Bin, Haiping Wu, and Yichen Wei. "Simple baselines for human pose estimation and tracking." Proceedings of the European Conference on Computer Vision (ECCV). 2018.*