# Team 4 Project Proposal Overview:

MAE: Jake Honma & Harsh Savla | ECE: Andrew Dunker | Math/CS: Jingli Zhou

**Goals:**

1. Document the process to configure TensorFlow and DonkeyCar using a Raspberry Pi with an AI accelerator Hat.
2. Measure the performance of the Raspberry Pi 5 (with and without the AI Hat) relative to the Jetson Nano & Jetson Xavier NX.
3. Determine the feasibility of replacing the Jetson Nano with the Raspberry Pi 5 with AI Hat+ in ECE/MAE 148.

**Requirements:** Jetson Nano, Jetson NX, Raspberry with Hailo AI Hat

# Deliverables

Must Have :

1. Benchmarked performance for:
   a. Jetson Nano w Tensorflow/TensorRT
   b. Jetson NX w Tensorflow/TensorRT
   c. RPI w Tensorflow/HailoRT
2. Documentation for the Process.

Nice to Have :

1. Benchmarked performance at different resolutions
2. Fully integrate the Hailo Model with DonkeyCar

# Hardware Overview:

| | Jetson Nano | Jetson NX | Raspberry Pi 5 w Hailo Hat |
|---|---|---|---|
| CPU | Quad-core ARM Cortex-A57 | 6-core ARM v8.2 64-bit CPU (Carmel) | Quad-core ARM Cortex-A76 |
| GPU | 128-core Maxwell | 384-core Volta GPU with 48 Tensor Cores | 12-core VideoCore VII |
| NPU | N/A | N/A | Hailo-8 AI Accelerator |
| Memory | 4GB LPDDR4 | 8GB LPDDR4x | 4/8GB LPDDR4X |
| Performance | 0.472 TFLOPS (FP16)<br>~1.88 TOPS (TOPS) | 21 TOPS (INT8) | 26 TOPS (INT8) |
| Cost | $225 | $650 - 800 | $70 (+$10 for 8GB RAM)<br>(+$70 for AI Hat) |

# Software Overview:

| | Jetson Nano | Jetson Nano (w/ Docker) | Jetson NX | Raspberry Pi 5 |
|---|---|---|---|---|
| Donkeycar | 4.5.1 | 5.1.dev0 | 5.1.dev0 | 5.2.dev2 |
| Python | 3.6.9 | 3.8.10 | 3.8.10 | 3.11.2 |
| Tensorflow | 2.5.0 | 2.12.0 | 2.12.0 | 2.15.1 |
| Pytorch | 1.8.0 | 2.1.0a0+41361538.nv23.06 | 2.1.0a0+41361538.nv23.06 | N/A |
| TensorRT | 7.1.3.0-1+cuda10.2 | 8.5.2.2-1+cuda11.4 | 8.5.2.2-1+cuda11.4 | N/A |

# CPU & GPU & NPU

- CPU(Central Processing Unit): The hardware that is essentially the brain of the computer, and used for general computing. The Jetson uses the ARM architecture.

- GPU(Graphic Processing Unit): In a process called parallel computing, GPUs break down large tasks into smaller ones that can be ran in parallel. Softwares like Tensorflow, utilize libraries like CUDA to offload computation and inference to the GPU, accelerating the computational time.

- NPU(Neural Processing Unit): NPUs are specialized processors for neural networks. They are optimized for matrix operations, parallel processing, and low latency. This means computationally expensive task such as deep learning can be offloaded here.

CPU: Decision Making

GPU: Offloads Parallel Task.

Cuda: allows parallel processing
Tensor RT: specialized for NVIDIA GPUs

NPU:Offload Deep Learning Task.

# What We Did: Jetson/Tensor RT

- .h5 → .savedmodel → tensorrt directory (worked w/ DSE 190 Team)
  - Different processes for Python 3.6 and 3.8 (creates directories w/ different contents)
  - Python 3.6:

```
(donkey) jetson@ucsdrobocar-148-04:~/projects/d4/models/suareztrt$ ls
assets  saved_model.pb  variables
```

  - Python 3.8:

```
(donkey) jetson@ucsdrobocar-148-04:~/projects/d4/models/suarez_tensorrt$ ls
assets  fingerprint.pb  saved_model.pb  variables
```

- Problems: Outdated documentation (.uff) & GPU access in a docker

FOR PYTHON 3.6

```
python
>>from tensorflow.python.compiler.tensorrt import trt_convert as trt
>>import os
>>saved_model_path = "models/yourmodel_converted.savedmodel"
>>tensorrt_model_path = "models/yourmodel"

>>os.makedirs(tensorrt_model_path, exist_ok=True)

>>conversion_params = trt.ConversionParams(precision_mode="FP16")

>>converter = trt.TrtGraphConverterV2(
   input_saved_model_dir=saved_model_path,
   conversion_params=conversion_params
)

>>converter.convert()
>>converter.save(tensorrt_model_path)
```

FOR PYTHON 3.8

```
python
>>from tensorflow.python.compiler.tensorrt import trt_convert as trt
>>import os
>>saved_model_path = "models/yourmodel_converted.savedmodel"
>>tensorrt_model_path = "models/yourmodel"

>>os.makedirs(tensorrt_model_path, exist_ok=True)

>>converter = trt.TrtGraphConverterV2(
        input_saved_model_dir=saved_model_path,
        precision_mode=trt.TrtPrecisionMode.FP16
)

>>converter.convert()
>>converter.save(tensorrt_model_path)
```

# What we did: RPI

- **To Benchmark Running DonkeyCar:**
  DonkeyCar has documentation to set up and run donkey on the raspberry pi. After installing DonkeyCar the process is the same as it is with the Jetson (without TensorRT).

- **To Benchmark with Hailo RT:**
  - Research how the Hailo NPU can be accessed through the RPI →
  - Research what models the Hailo NPU can run. →
  - Research, Implement and Debug Model conversion from .h5 to .hef
  - Research and Implement Benchmark .hef using the Hailo toolkit.
  - Integrate .hef into DonkeyCar

# Profile.py

- DonkeyCar’s benchmarking tool
- Loads a model
- Creates random image data
- Calculates FPS from the amount of time between run command and a response form model

```python
import os
from docopt import docopt
import donkeycar as dk
import numpy as np
from donkeycar.utils import FPSTimer # type: ignore


def profile(model_path, model_type):
    cfg = dk.load_config('config.py')
    model_path = os.path.expanduser(model_path)

    if model_path.endswith(".hef"):
        from hailo_runner import HailoModelRunner
        model = HailoModelRunner(model_path)
    else:
        model = dk.utils.get_model_by_type(model_type, cfg)
        model.load(model_path)

    h, w, ch = cfg.IMAGE_H, cfg.IMAGE_W, cfg.IMAGE_DEPTH

    # generate random array in the right shape in [0,1)
    img = np.random.randint(0, 255, size=(h, w, ch))

    # make a timer obj
    timer = FPSTimer()

    try:
        while True:
            # run inferencing
            model.run(img)
            # time
            timer.on_frame()

    except KeyboardInterrupt:
        if model_path.endswith(".hef"):
            model.shutdown()
        pass
```

# manage.py

```
def load_model_json(kl, json_fnm):
    start = time.time()
    print('loading model json', json_fnm)
    from tensorflow.python import keras
    try:
        with open(json_fnm, 'r') as handle:
            contents = handle.read()
            kl.model = keras.models.model_from_json(contents)
        print('finished loading json in %s sec.' % (str(time.time() - start)) )
    except Exception as e:
        print(e)
        print("ERR>> problems loading model json", json_fnm)

if model_path:
    # When we have a model, first create an appropriate Keras part
    kl = dk.utils.get_model_by_type(model_type, cfg)

    model_reload_cb = None
    #####
    if os.path.isdir(model_path):
        print("INFO: Detected TensorRT SavedModel directory. Proceeding to load it.")
        load_model(kl, model_path)
    #####

    elif '.h5' in model_path or '.trt' in model_path or '.tflite' in \
            model_path or '.savedmodel' in model_path or '.pth':
        # load the whole model with weigths, etc
        load_model(kl, model_path)

        def reload_model(filename):
            load_model(kl, filename)

        model_reload_cb = reload_model

    elif '.json' in model_path:
        # when we have a .json extension
        # load the model from there and look for a matching
        # .wts file with just weights
```

- The hub of using DonkeyCar
- For autopilot
    - Checks the model’s file extension to determine model type
    - Loads specified model type with model file
    - If your model is not static, it allows reloading for some model types
    - Model object is then handed off to

# hailo_runner.py

- The HailoModelRunner class is designed to be indistinguishable from other models used by DonkeyCar
    - RGB Image input -> run method -> Steering, Throttle tuple output
- In its current form, it is faster than most of its competition, but could be much faster

```python
import numpy as np
from donkeycar.utils import normalize_image, throttle as compute_throttle
from hailo_platform import (
    HEF,
    VDevice,
    HailoStreamInterface,
    InferVStreams,
    ConfigureParams,
    InputVStreamParams,
    OutputVStreamParams,
    FormatType
)

class HailoModelRunner:
    def __init__(self, hef_path):
        self.hef_path = hef_path

        self.device = VDevice()
        self.hef = HEF(hef_path)

        self.configure_params = ConfigureParams.create_from_hef(hef=self.hef, interface=HailoStreamInterface.PCIe)
        network_groups = self.device.configure(self.hef, self.configure_params)

        self.network_group = network_groups[0]
        self.network_group_params = self.network_group.create_params()

        self.input_vstreams_params = InputVStreamParams.make(self.network_group, format_type=FormatType.FLOAT32)
        self.output_vstreams_params = OutputVStreamParams.make(self.network_group, format_type=FormatType.FLOAT32)

        self.input_vstream_info = self.hef.get_input_vstream_infos()[0]
        self.output_vstream_info = self.hef.get_output_vstream_infos()[0]

        self.image_height, self.image_width, self.channels = self.input_vstream_info.shape
        self.input_batch = np.empty((1, self.image_height, self.image_width, self.channels), dtype=np.float32)

    def run(self, image):
        image = image.astype(np.uint8)

        image_normalized = normalize_image(image).astype(np.float32)
        self.input_batch[0] = image_normalized

        with InferVStreams(self.network_group, self.input_vstreams_params, self.output_vstreams_params) as infer_pipeline:
            input_data = {self.input_vstream_info.name: self.input_batch}
            with self.network_group.activate(self.network_group_params):
                infer_results = infer_pipeline.infer(input_data)

        outputs = infer_results[self.output_vstream_info.name]

        if outputs.shape[1] == 1:
            steering = outputs[0, 0]
            throttle = compute_throttle(steering)
        else:
            steering = outputs[0, 0]
            throttle = outputs[0, 1]
        return steering, throttle
```

# Final Metrics: Same Model at 120x160 res

| | Jetson Nano | Jetson Nano (w/ Donkey 5.1.dev0) | Jetson NX | Raspberry Pi 5 |
|---|---|---|---|---|
| Linear (.h5) | 46-47 fps | 26 fps | 41-42 fps | 61-66 fps |
| TFLite (.tflite) | 62 fps | 80 fps | 125-126fps | 313fps |
| TensorRT (directory) | 82fps | N/A | 250-260fps | N/A |
| Hailo (.hef) | N/A | N/A | N/A | 217**fps |

**With its own benchmarking tool the Hailo model can perform at over 13000 fps, and given is higher performance spec and better optimized model, it is likely that our method of integrating Hailo into DonkeyCar is not optimized.

# Training Higher Resolution Models

- Attempted to train:
  - 120x160 pixel
  - 360p
  - 720p
- 360p and above would not train on Datahub
- Attempted to train locally on a more powerful GPU instead
  - 720p maxed out VRAM usage and has very large RAM usage
- Required optimizing DonkeyCar's training process
  - Optimized Keras VRAM usage in DonkeyCar
  - Attempted to implement Mixed Precision Training
  - Dropped batch size to 8
- Was only able to successfully train for 4 epochs

# Final Recommendation:

**Use Case Based Recommendations:**

1. **If you own a Jetson (Nano/NX): Jetson**
   You can boost the performance of your Jetson using Tensor RT and given the limited support for the Hailo AI hat, the upgrade is not worthwhile.
2. **If you own a RPI 5: RPI 5 (w future Hailo Hat upgrade option)**
   Without acceleration the RPI 5 outperforms both Jetson models, and given its compatibility with newer software, larger support infrastructure, lower cost and the possibility of the integration of the AI Hat with DonkeyCar makes it a future proof purchase.
3. **If you a looking to purchase a new processor: RPI 5 (w future Hailo Hat upgrade option)**
   Considering the RPI has better compatibility with new software, a larger support infrastructure, lower cost, more developers, and likely integration with the AI hat, it is a better purchase.
4. **For this class:**
   To boost performance you can use TensorRT acceleration, especially since Hailo is not fully integrated. If you are looking to upgrade the current hardware, the RPI 5 is a better option.

# Future Prospects to build on this project

1. Optimize the integration of Hailo in DonkeyCar

   Though we were able to get a .hef file to run in DonkeyCar, based on the results and what we expected, the integration could be performed more efficiently.

2. Test Models at Different Resolutions

   Test and optimize models based on OAKD lite resolution and hardware (Jetson vs. Raspberry Pi 5) to find the highest performing resolutions for deep learning.

3. Continue to Optimize DonkeyCar Training

   Finish optimizing memory management to allow training of larger resolution models

# What did not work as expected

- Out of date documentation for TensorRT conversion (.uff)
- Creating .hef model
- Training new models due to GPU Cluster
    - Had to train locally which minimized how much time we had to test the performance of various models
- Full integration of Hailo Executable Files (.hef)
- Running manage.py due to Vesc problems at low speeds

# Linked Documentation

Jetson Benchmarking w/wo TensorRT Conversion

Raspberry Pi Setup & AI Hat Benchmarking