Real-time Wildlife Detection on Embedded Systems

Ankit Mathur
Stanford University
ankit96@stanford.edu

Saelig Khattar
Stanford University
saelig@stanford.edu

Abstract

Modern machine learning algorithms are generally deployed on high performance GPUs, and require significant power and memory resources. However, such resources are not always available, especially when one wants to deploy such models on the "edge." In this project, we work with Jasper Ridge Biological Preserve in Stanford, CA, and develop an image classification system that can be deployed on the 18 camera traps they have setup around the preserve to detect wildlife. We experiment with and deploy various image classification models on a Raspberry Pi that can be connected to these camera traps. We show that our best model achieves a mean-per-class accuracy of 87.6% and can quickly run inference in real-time, on-device, using minimal power.

1. Introduction

Our project focuses on techniques to do real-time detection of wildlife from video feeds on a Raspberry Pi. In particular, we explore machine learning based approaches that have been shown to be successful on the dataset that we are exploring.

Jasper Ridge Biological Preserve is 15 minutes north of Stanford and home to a variety of flora and fauna that are protected from significant human interaction. Several kinds of research are conducted here to track their natural lifecycles and those of the surrounding ecosystem. Jasper Ridge has deployed a fleet of 18 cameras around the park for the purposes of capturing photos of animals naturally exploring the park.

In CS341 last year, a team worked on designing a neural network based solution to identify a variety of wildlife tracked within the park. They designed this to help reduce the load on the volunteers for the park who were helping classify the images.

This year, we need to take things a step further. Last years model was optimized for accuracy but not compute. Jasper Ridge is now working on designing cameras that have Raspberry Pi-like compute systems and long range radio signals. Our task is to design power-efficient models that can run on the Raspberry Pi and identify a significantly smaller number of classes. The goal is to be able to deploy this in real-time as well, so we seek to evaluate model architectures that are both fast and computationally efficient. Eventually, we may also work on being able to send a notification (containing a thumbnail of the image for verification) if we identify a human with a gun in order to identify poachers and potentially trigger a law enforcement action.

2. Data

The dataset consists of 3 classes - humans, mountain lions, and coyotes. This consists of only 33.9% of the original dataset, which contained up to 24 classes of wildlife. Moreover, the dataset is imbalanced, with 86% of the dataset corresponding to just the human class (See Figure 2). This is a classic problem with real world datasets, where the distributions of the things you see are not nearly similar.

The original dataset is composed of almost 200,000 JPEG images, originating from several different cameras at different angles in different places and collected over 10 years. For each camera array, there is an Excel sheet that corresponds to the correct labels. This Excel sheet also contains metadata like the light levels and the temperature. The subset that we work with in this project contains approximately 64,000 images.

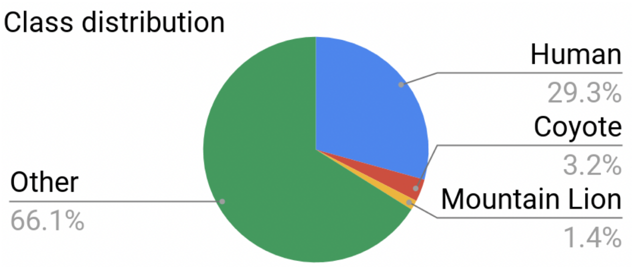

Figure 1. Jasper Ridge Dataset: We observe how this specific task uses only a subset of the data, and within that data, there is a significant class imbalance.

3. Related Work

Some of the first attempts at using machine learning to detect wildlife from camera trap images used hand-crafted features. For example, Yu et al. [7] used sparse coding spatial pyramid matching (ScSPM) to extract features and then used a linear SVM to make predictions on classifying different species. However, for complex tasks such as image detection and classification, deep learning methods which dont require hand-crafted features have proven to be more effective [3]. Villa et al. [6] shows that such methods are also effective for wildlife detection. They use very deep convolutional neural networks to identify the 26 most common species in the Snapshot Serengeti (SSe) dataset, achieving a. 5.4% Top-1 and 60.4% Top-5 accuracy. Others also attempt similar tasks [4, 5].

Notably, a group in last years offering of CS341 has also done relevant work: Yu et al. implement deep convolutional neural networks to identify species in the Jasper Ridge dataset, which is the same dataset we are using for our project. They experimented with several data preprocessing techniques such as background subtraction and oversampling. They also experimented with different CNN based pipelines, and model ensembling methods. Their best performing ensemble achieved an accuracy of 93.7% and an average F1 score of 85.73. However, though we use the same dataset, our task is very different, since we need to our system to run in real-time on an embedded device with stringent memory and power constraints.

The need for doing species identification on the edge is very apparent in large natural parks and biological preserves, such as the Jasper Ridge Biological Preserve in Stanford, and many parks in Africa, such as the Garamba National Park in the Democratic Republic of Congo. These parks are very large and can be in very remote parts, making it very difficult to get access to wireless networks that have enough bandwidth to transfer high-resolution images to an offsite location with the compute power necessary to run these large deep learning networks. In [2], Elias et al. attempt this problem, and explore and implement methods that use resource rich systems to train convolutional neural networks, and use edge systems to perform the detection of bears, deer, and coyotes. They deploy their system at the UCSB Sedgwick Research Reserve, achieving an accuracy of 87% on these three classes.

4. Methodology

4.1. Preprocessing

In order to effectively train machine learning models, train-time augmentations are extremely important. We use 3 common forms of preprocessing to augment our data in this machine learning model:

Random flipping - flipping the image across the y axis can help teach the model different perspectives. This is a data augmentation technique that can help the model not overfit to a specific pose for a given class.

Random cropping - Cropping the image in different portions helps the model learns more high level salient features that generalize better to various zoom scales and different poses and subcrops of the same animal.

Per Image Standardization - several papers show that, empirically, models find it easier to learn when inputs are bias and variance normalized. This can be done across the entire dataset or per-image. In this case, we go with per image, since the dataset contains a wide variety of scenes with dramatically different lighting, meaning the variance is quite high across the entire dataset. It is not as high in a single given image.

4.2. Input Resolution

Because we are operating in the embedded systems domain, it's important that we consider the compute budget of the models. While we explore different model types, some with reduced computational intensity, another great way of controlling the amount of computation necessary is to adjust the input resolution of the image. Models that operate on thumbnails instead of full resolution images can often attain similar accuracy in many use cases. Intuitively, this is quite similar to being able to recognize the kind of animal something might be even if you're very far away.

Models trained on different input resolutions often have subtly different architecture parameters (including filter sizes and strides in convolutions, which control the receptive field of the convolutional layers). However, as we show, they can be very effective and useful considering the power vs. accuracy tradeoff.

4.3. Training

Training our model required us to keep several different aspects of Tensorflow in mind from an implementation standpoint.

4.3.1 Model Architecture and Learning Schedule

The model that we chose to use for this project was the ResNet model. ResNet is one among several extremely good architectures, like Inception, MobileNet, ShuffleNet, DenseNet, etc. which are popular for image classification tasks. ResNets' rely on the residual layers to train deeper networks in a more stable manner, along with being able to model more complex linearities. ResNets have shown themselves to be versatile, in applications from medical imaging to serving as the backbone for object detection and in general as a feature extractor. It is widely deployed in production, and, as such, most performance and compilation systems have put extra effort into compiling the operations used in ResNet into optimized kernels.

We use Tensorflow to train our model, implementing the model in both Tensorflow and Keras, using the official/models repository of Tensorflow. We train with a batch-size based learning rate, which decays every 30 epochs, following the learning schedule used to train on the more complex Imagenet-1000 task. Training required significant time for the ResNet-50, which needed almost 24 hours to train on a Tesla V100 GPU, using 8 cores.

4.3.2 Class Imbalance

It's worth noting that we had significantly more examples of humans than we had examples of coyotes and lions. As a result of this, if we trained optimizing for a standard metric or without regularization and specific learning rates, it was easy for our model to fall into the local optimum of predicting only the human class for all labels. As such, we take care to report and optimize for the mean per-class accuracy and track the accuracy on the individual classes.

We maintained the class imbalance in our validation and testing splits as well, since we thought it was important to be evaluating on a dataset as close to the real life distribution of objects as possible.

4.3.3 Tensorflow Lite on Raspberry Pi

Using Tensorflow models in Tensorflow Lite requires converting the models that are trained in Tensorflow into the SavedModel format. This requires using the freeze-graph utility. Then, we use the Tensorflow Lite converter to compile the model, setting flags that will auto-prune computation and other graph nodes that are unnecessary at inference time. This prunes our model sizes by nearly 50x. Moreover, they are compiled into model types that are optimized for execution on microarchitectures. Special kernels that organize data in convolutions in a different memory format typically have significant impact on the runtime and execution characteristics on low memory and low power architectures.

Furthermore, as part of the deliverable, we needed to be able to watch for new photos to be taken by the camera trap and run inference live on those images. As such, we need to be watching the file system for new files taken when the camera takes the photo. However, standard libraries in the Python runtime use a polling based interface that we found took nearly 15% CPU utilization on average on the Raspberry Pi 3. As part of the effort to focus on performance, we implemented a system which is event-based and takes only 2% of CPU while in sleep. This is important, since the system is in sleep the vast majority of the time, and this contribution reduces power by nearly 8x.

We implement the preprocessing operations in the Pillow image processing library, and we shipped a system that handles the full process of executing in both Tensorflow and Tensorflow Lite when configured to work with a given type of model file.

For the use case in which poachers want to be detected in Garamaba National Park, we need to be able to also be able to generate a thumbnail if the model is confident that the object in the image is a human being. This could be used to alert authorities to potentially suspicious activity. We set a flag that logs predictions and thumbnails to a subfolder of the folder being watched for photo generation by the camera trap.

4.4. Performance Benchmarking

In order to adequately benchmark on the Raspberry Pi, we ran a warmup stage to get the CPU and code loading warmed up. We then measured the average inference and preprocessing time for the model, combining those into a total inference execution time. We also had a power meter that we could probe with which we used to measure in Watts the amount of power that the Raspberry Pi was drawing throughout inference. This is an important step, since we hope to establish a power vs. accuracy tradeoff, and, while execution time serves as a great approximation of power used, the real measurement is also useful.

It's important to note that the power utilization of the system also includes the amount of power that the Raspbian operating system is using to continue running, which is about 3 Watts from our measurements.

5. Experiments

5.1. Setup

We split our dataset of 64,000 images into a training, validation, and testing set following a rough 80-10-10 split. To ensure our results can generalize to the actual distribution of the classes, we made sure the class balance among the different splits was the same. This procedure yielded approximately 50,000 images in the training set, 8000 in validation, and 6000 in testing.

We evaluate our models using top-1 accuracy. As we discuss briefly in section 4.3.2, due to the significant class imbalance, we also evaluate our models using mean-per-class accuracy, and the individual accuracy for each class.

Further, because we are ultimately interested in deploying these models on the edge on a low-power embedded system, we measure the maximum power drawn by these models (original model and quantized version), as well as the execution time for inference on a single image, as we discussed in section 4.4.

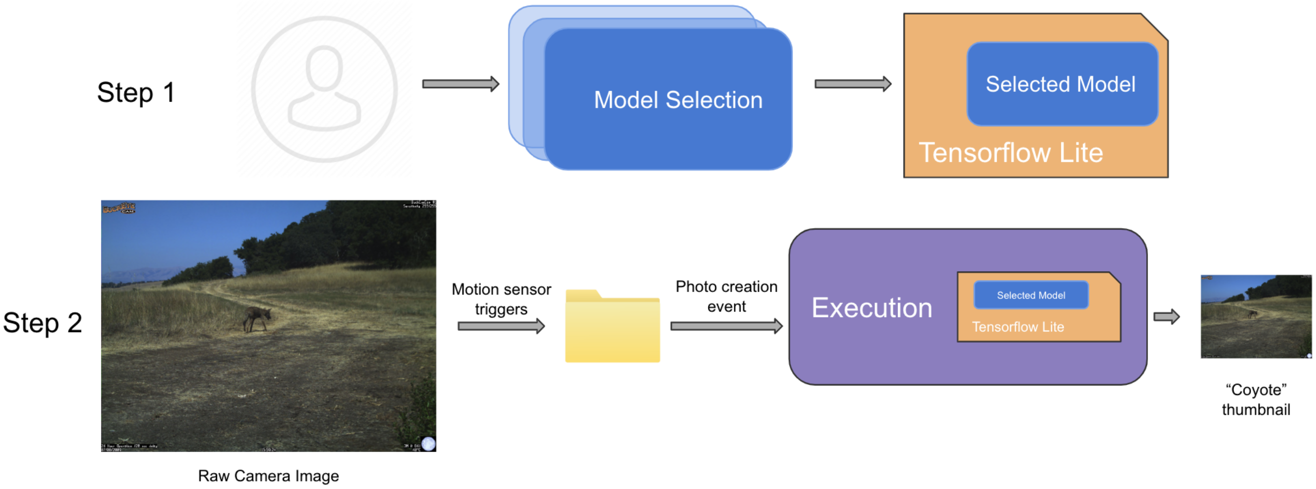

Figure 2. Full System Process. In Step (1) Users use our model selector, which chooses which model to use, given power/accuracy constraint. In Step (2), Users launch our script - it efficiently handles power consumption, preprocesses new images, and outputs predictions with thumbnails.

Table 1. Here, we see the various final top-1 accuracies on the test set (note that the plots shown over time are for the validation set. We see that the models generalize generalize as we expected them to and rank in the same order in terms of quality as we observed in the validation set.

MODEL	HUMAN ACCURACY	COYOTE ACCURACY	LION ACCURACY	MEAN PER-CLASS ACCURACY
RESNET-18 (64X64)	97.9	57.6	62.7	72.7
RESNET-18 (224X224)	99.3	71.5	81.7	84.2
RESNET-50 (64X64)	98.2	58.0	62.0	72.7
RESNET-50 (224X224)	99.2	80.1	83.7	87.6

5.2. Results

5.2.1 Model Performance

For each of our models, we tune hyperparameters to maximize performance the validation set, and then evaluate the models on the test set (See Appendix A for validation scores during training). These results are shown in in Table 1. Investigating the results, we immediately see that all four models were almost perfect at classifying humans. We note that the best performing model was the ResNet-50 with images at a 224x224 resolution, with a mean-per-class accuracy of 86.6%. This model gets $> 80\%$ for all classes. This is what one would expect – the larger model operating with a higher resolution does the best.

Interestingly, we see that image resolution makes a bigger difference than model size. Specifically, the ResNet-50 and ResNet-18 operating at 224x224 image resolution do the best, while there is very little difference between the two models operating at 64x64 image resolution (in fact, their mean-per-class accuracy is the same). Perhaps model size will have more significance at much higher resolutions — but models operating at these resolutions are unlikely to fit on an embedded device such as Raspberry Pi.

Further it is important to note that the lowest memory footprint model, the ResNet-18 at 64x64 doesn't do terrible as it gets a mean-per-class accuracy of 72.7% Such as model would be ideal for a situation where identifying humans was relatively more important, since it still achieves 97.9% accuracy on that task.

5.2.2 Model Power Benchmarking

In Table 2, we show the execution time and maximum power draw of each model as well as its quantized version compiled with TensorFlow Lite. The Rasberry Pi in an idle state (with LEDs and WiFi on) draws about 3 W of power — we report power increases over this baseline.

As we expected, the ResNet-50 at 224x224 took the longest time to make an inference, on average, and the Resnet-18 at 64x64 took the shortest amount of time, on average. Interestingly, we note large differences in execution time among the TesnorFlow (TF) and TensorFlow Lite

Table 2. Here, we can see the power times broken down by the framework the model is run on (Naive Tensorflow vs. Tensorflow Lite), along with seeing the different execution times (Exec Time) and maximum power draws (MPD) for the different models being run at different resolutions.

MODEL	TF LITE EXEC TIME (S)	TF EXEC TIME (S)	TF LITE MPD (W)	TF MPD (W)
RESNET-18 (64X64)	0.39	0.21	1	3
RESNET-18 (224X224)	4.03	1.09	2	4
RESNET-50 (64X64)	0.92	0.41	2	3
RESNET-50 (224X224)	10.58	2.54	2	4

(TF Lite) versions of the models operating at 224x224 image resolution. For example, the TF version of the ResNet-50 took about $1/4$ of the time to classify a single image than its TF Lite version at a 224x224 resolution. However, the TF version drew twice the amount of power. The other models follow a similar trend, though the relative differences between the two versions of the model may not be as large. These results introduce an interesting trade-off for the client: execution time vs power drawn. If there are constraints on the maximum current the embedded device can draw, than it may be advisable to use the TF Lite version of a given model. However, if energy consumption is the main concern, it may instead be advisable to the use the TF version of a model since though we draw more power, it is for a shorter amount of time. Of course, this trade-off varies model to model as one can see from Table 2.

5.2.3 Delivered System

Ultimately, we want to deploy a system that can operate on embedded systems directly connected to the cameras at the Jasper Ridge Biological Preserve, which run on solar power. As we discussed in prior sections, there are various tradeoffs in accuracy, execution time, and power among the different models. For this reason, we design a system that provides the following features: (1) Pretrained model selection, (2) an efficient execution framework, and (3) be able to run in real-time, on device, and use minimal power for inference. Our system efficiently handles power consumption with various optimizations discussed in Section 4.3.3, preprocesses new images as they come from the camera, and outputs predictions. If the prediction is particularly interesting depending on the use case (this is easily configurable), it will also output a thumbnail image. This would make it easy for a lower-power device to transmit this classified image over a limited-bandwith region. The full process is shown in Figure 2.

5.3. Challenges

One of the largest challenges we faced in this project was running the models using TensorFlow Lite (TF Lite) on the Raspberry Pi. While were were able to compile our models to TF Lite, we kept getting a memory allocation error when we tried to load the model on the Raspberry Pi, no matter which model or resolution we tried. Since this compilation process has been shown to work in literature [1], we were led to believe the TF Lite model file was corrupt. To work around this issues, we tried several things. First, we tried changing the SavedModel file that was exported by the training code by pruning operators using a describe-graph utility in Tensorflow. No matter what we changed and reworked, we would get the same error. Then, we tried converting a checkpoint from training, using a graph description file and converting that into a SavedModel and converting that into a TF Lite package. Unfortunately, the checkpoint to SavedModel step required several hurdles. Even still, once this worked, conversion to TFLite failed with a ProtoBuffer error which could not be resolved.

After were unable to get any of the above techniques to work, we decided to switch to Keras, a higher level framework which can use TensorFlow as a backend, and implemented the same models. With Keras, we were able to successfully convert our models to TF Lite and then load and run inference with them on the Raspberry Pi. However, even after extensive hyperparameter tuning, we were unable to achieve the same performance with the Keras versions of the models for whatever reason.

Because of this we decided to take another crack at converting TF models. After much more debugging, we realized that the saved graph that we exported from the TF model expected inputs of the same size as when training — this means it expected input in batches of 32! However, on the Raspberry Pi we want to run inference on a single image at a time, so we only need a batch size of 1. Because the saved graph expected input in batches of 32, however, TF Lite tried to allocate memory for such input when it loaded the graph, which was much more than it had on device, resulting in the memory allocation error. After we set the batch size to 1 in our exported graph, loading and running models in TF Lite worked perfectly.

6. Conclusion and Future Work

In conclusion, we developed a highly accurate model for wildlife detection, with a mean per-class accuracy of 87.6%, and a human classification accuracy of 99.2%. We also provided an analysis of power and execution time of the different models on a Raspberry Pi, which we hope Jasper Ridge can utilize to determine which models they should deploy based on their specific constraints and use cases. To assist with this, we have built a deployable system that enables them to easily select which model they would like to use.

In the future, we would like to explore further quantizing these models, such as by implementing weight pruning. This would likely decrease accuracy, but may decrease the power and memory footprint of these models.

References

[1] Alasdair Allan. Benchmarking tensorflow and tensorflow lite on the raspberry pi. https://blog.hackster.io/benchmarking-tensorflow-and-tensorflow-lite-on-the-raspberry-pi-43f51b796796. Accessed: 2019-05-15.

[2] Andy Rosales Elias, Nevena Golubovic, Chandra Krintz, and Rich Wolski. Where's the bear?-automating wildlife image processing using iot and edge cloud systems. In *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 247–258. IEEE, 2017.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[4] Slavomir Matuska, Robert Hudec, Miroslav Benco, Patrik Kamencay, and Martina Zachariasova. A novel system for automatic detection and classification of animal. In *2014 ELEKTRO*, pages 76–80. IEEE, 2014.

[5] Mohammad Sadegh Norouzzadeh, Anh Nguyen, Margaret Kosmala, Alexandra Swanson, Meredith S Palmer, Craig Packer, and Jeff Clune. Automatically identifying, counting, and describing wild animals in camera-trap images with deep learning. *Proceedings of the National Academy of Sciences*, 115(25):E5716–E5725, 2018.

[6] Alexander Gomez Villa, Augusto Salazar, and Francisco Vargas. Towards automatic wild animal monitoring: Identification of animal species in camera-trap images using very deep convolutional neural networks. *Ecological informatics*, 41:24–32, 2017.

[7] Xiaoyuan Yu, Jiangping Wang, Roland Kays, Patrick A Jansen, Tianjiang Wang, and Thomas Huang. Automated identification of animal species in camera trap images. *EURASIP Journal on Image and Video Processing*, 2013(1):52, 2013.

A. Training Curves for Each Model and Class

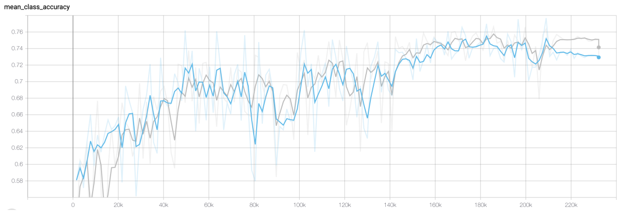

Figure 3. Mean per-class-accuracy over time for ResNet-18 and ResNet-50 trained at 64x64 on the whole dataset - ResNet-18 is in blue, ResNet-50 is in gray.

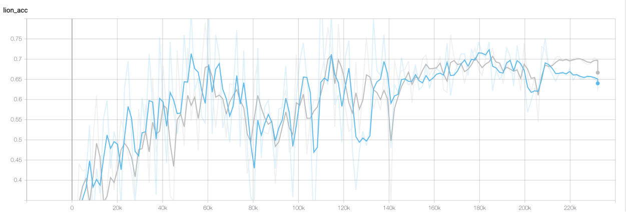

Figure 4. Lion Accuracy for ResNet-18 and ResNet-50 trained at 64x64 on the whole dataset - ResNet-18 is in blue, ResNet-50 is in gray.

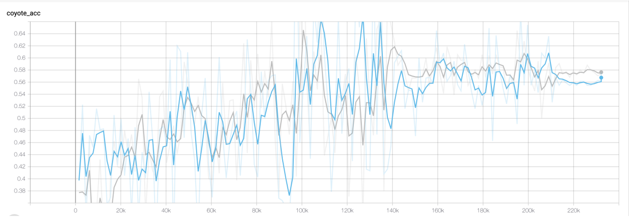

Figure 5. Coyote Accuracy for ResNet-18 and ResNet-50 trained at 64x64 on the whole dataset - ResNet-18 is in blue, ResNet-50 is in gray.

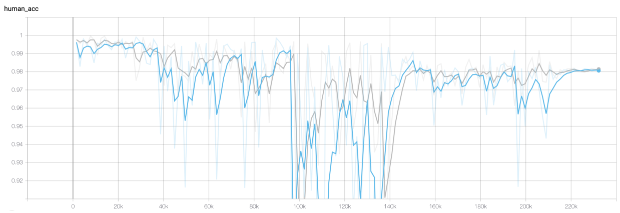

Figure 6. Human Accuracy for ResNet-18 and ResNet-50 trained at 64x64 on the whole dataset - ResNet-18 is in blue, ResNet-50 is in gray.

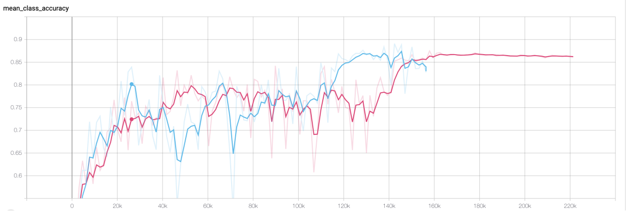

Figure 7. Mean per-class-accuracy for ResNet-18 and ResNet-50 trained at 224x224 on the whole dataset - ResNet-18 is in blue, ResNet-50 is in gray.

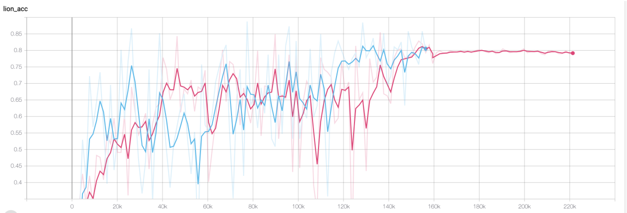

Figure 8. Lion Accuracy for ResNet-18 and ResNet-50 trained at 224x224 on the whole dataset - ResNet-18 is in blue, ResNet-50 is in red.

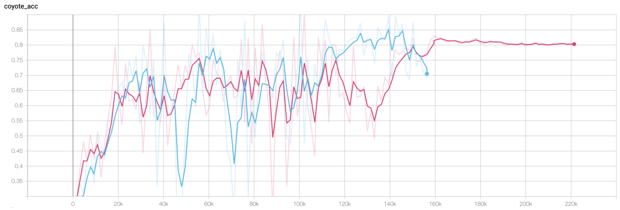

Figure 9. Coyote Accuracy for ResNet-18 and ResNet-50 trained at 224x224 on the whole dataset - ResNet-20 is in blue, ResNet-50 is in red.

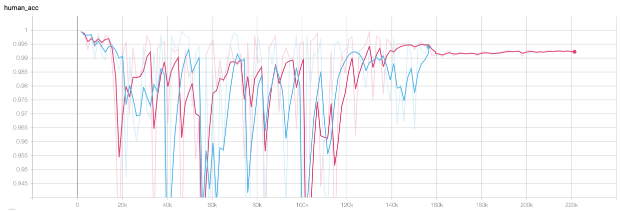

Figure 10. Human Accuracy for ResNet-18 and ResNet-50 trained at 224x224 on the whole dataset - ResNet-18 is in blue, ResNet-50 is in red.