Survey of TinyML Methods for Humpback Whale Detection on Embedded Devices

Oleg Golev
ogolev@princeton.edu

Gerald Huang
gmhuang@princeton.edu

Vikash Modi
vikashm@princeton.edu

***Abstract*—In this study, we explore the tradeoffs between accuracy, model size, and training time for humpback whale vocalization detection models. We investigate the potential of using TensorFlow Lite (TFLite) and feature selection techniques to create smaller and more efficient models for low-power devices. We develop a custom Keras model using the pre-trained humpback whale model as a baseline, achieving 96.13% accuracy. We then train a TFLite model that achieves 76.25% accuracy but is only 5% of the size of the Keras model. Additionally, we experiment with different features, specifically mel-frequency cepstral coefficients (MFCCs), as inputs to a a VGG block-based CNN to showcase their higher performance (81.99% vs 68.14% accuracy), faster runtime, and smaller model size advantages over using the reference waveform features. Our findings highlight the potential of using TFLite as a baseline for creating efficient models while showing the importance of domain-specific knowledge in further optimizing model accuracy and size.**

***Index Terms*—TinyML, Humpback Whale Detection, Embedded Devices, Convolutional Neural Networks, TFLite**

Introduction

The rapid advancement of machine learning (ML) techniques, combined with the increasing availability of large-scale datasets, has enabled remarkable progress in a wide range of applications. In recent years, one such application has been the automated detection and analysis of marine mammal vocalizations, specifically the songs of humpback whales. The study "A Convolutional Neural Network for Automated Detection of Humpback Whale Song in a Diverse, Long-Term Passive Acoustic Dataset" [1] presents a convolutional neural network (CNN) that effectively identifies humpback whale songs in passive acoustic recordings.

The ability to automatically detect and analyze humpback whale vocalizations has significant ecological and conservation implications. As an endangered species, understanding humpback whales' distribution, abundance, and behavior is crucial for developing effective conservation strategies [2]. Passive acoustic monitoring (PAM) enables researchers to non-invasively study these animals over long periods and in remote locations, providing invaluable insights into their lives [3].

However, the scale and diversity of PAM data make manual analysis a time-consuming and resource-intensive task. The CNN presented in the original study [1] offers a solution to this challenge by automating the detection process and significantly reducing the time and effort required for analysis. Consequently, this enables researchers to monitor larger areas over extended periods, providing a more comprehensive understanding of humpback whale populations and their habitats.

While the original CNN model has

demonstrated impressive performance, its deployment on embedded devices, such as low-power, small-footprint underwater sensor nodes, remains challenging due to its computational and memory requirements. Shrinking the model size enables real-time, energy-efficient, and continuous monitoring in remote and inaccessible environments. This is where TinyML, the branch of machine learning focused on developing ML models for resource-constrained hardware, comes in [4].

In this paper, we propose an efficient TinyML-based humpback whale detection model that retains the performance of the original CNN while significantly reducing its computational and memory requirements. This optimization enables the model to be deployed on embedded devices, which offers two key benefits:

1) Real-time Cetecean Detection: The primary extension the original work discusses is utilizing the model for endpoint PAM feature extraction instead of historical analysis. By processing data on the device itself, the model can run inference on continuous streams of samples, enabling real-time detection and response to underwater audio events.
2) Scalability: With a smaller, more efficient model, it becomes feasible to deploy sensor nodes over vast ocean expanses, providing comprehensive spatial and temporal coverage of marine mammal activity and data collection.

For these reasons, this paper serves as a survey of methods and approaches that can enable on-device execution of ML-based humpback whale song detection systems, both being a medium for us to learn TinyML in an exciting context and providing an insight into how our work implicates further studies in automated marine research and conservation.

I. Related Work

A. Automated Marine Vocalization Detection

The increasing use of passive acoustic monitoring (PAM) in marine mammal research has led to the development of various automated detection and classification methods. A significant amount of research has been conducted on detecting the vocalizations of cetaceans, including humpback whales, using machine learning algorithms [3], [5]. The original study [1] we build upon presents a convolutional neural network (CNN) for the automated detection of humpback whale songs, demonstrating high performance in diverse and long-term passive acoustic datasets.

B. Deep Learning Model Compression Techniques

In order to deploy deep learning models on resource-constrained devices, several model compression techniques have been proposed. These techniques aim to reduce the size and computational requirements of the models while retaining their performance. Key methods include network pruning [6], weight quantization [7], and knowledge distillation [8]. In our work, we employ a combination of these techniques to shrink the various model we consider for deployment on embedded devices.

C. TinyML

TinyML, the branch of machine learning focused on developing ML models for resource-constrained hardware, has been gaining significant attention in recent years [4]. Researchers have developed various techniques and tools to facilitate the deployment of ML models on devices with limited memory, processing power, and energy supply, such as microcontrollers and IoT devices [9], [10]. By leveraging TinyML techniques, our work aims to develop an efficient humpback whale detection model that can be deployed on embedded devices for real-time, energy-efficient, and scalable monitoring.

Background

D. Network Pruning

Network pruning is a model compression technique that aims to reduce the size and computational requirements of a neural network by removing redundant or less important connections or neurons. The central idea behind network pruning is that not all connections contribute equally to the model's performance, and removing less important connections may not significantly impact the accuracy while providing substantial reductions in model size and computational complexity [6].

There are two main types of pruning: weight pruning and neuron pruning. In weight pruning, individual connections with small weights are removed, while in neuron pruning, entire neurons along with their connections are removed if their activations are consistently low throughout the training process.

A common method for pruning is to apply a threshold value to the weights or activations. Connections or neurons with values below the threshold are removed, and the remaining network is retrained to fine-tune the weights and recover any lost performance. The pruning and retraining process can be performed iteratively to achieve the desired level of compression.

E. Weight Quantization

Weight quantization is another model compression technique that aims to reduce the memory footprint and computational requirements of a neural network by reducing the precision of the weights and activations. This is achieved by representing the weights and activations using a smaller number of bits than the original representation, typically 16, 8, or even fewer bits [7].

Quantization can be performed uniformly, where the range of values is divided into equally spaced intervals, or non-uniformly, where the intervals are chosen adaptively based on the distribution of the values. Common types of quantization include linear quantization, logarithmic quantization, and binary or ternary quantization.

By using lower-precision representations, the memory required to store the weights and activations is reduced, leading to a smaller model size. Furthermore, lower-precision arithmetic can be computed more efficiently than high-precision arithmetic, resulting in faster computation and lower energy consumption.

F. Knowledge Distillation

Knowledge distillation is a model compression technique that transfers the knowledge from a large, complex teacher model to a smaller, more efficient student model [8]. The student model is trained to mimic the output distribution of the teacher model, rather than the ground truth labels, which allows it to learn a more generalized representation of the data.

The distillation process is typically performed by minimizing a loss function that combines the cross-entropy loss between the student's output and the ground truth labels and a distillation loss that measures the difference between the student's output and the teacher's output. The distillation loss is often computed using the Kullback-Leibler divergence between the two output distributions. The teacher's output is typically softened using a temperature parameter to produce a smoother distribution.

By learning from the teacher model, the student model can achieve higher performance than if trained directly on the ground truth labels. This enables the use of smaller and more efficient models without sacrificing significant performance.

Methodology

Our initial aims for this project were to shrink an already-existing model [11], create our own homebrew TFLite model, prune a

VGG block-based model, and compare which methods had the best outcome in terms of speed, size, and accuracy.

The Existing Model and Dataset

Published in 2021, the humpback_whale model [11] by NOAA and Google provides a way to analyze audio samples for the presence of humpback whale vocalizations. The model is a large CNN (ResNet-50) trained on over 187,000 hours of data collected across the Central and Western Pacific between 2005 and 2019 to identify humpback whale vocalizations [12]. The model does so with average precision of 97%, although it achieves this score at the cost of its size of 101.1 MB. The dataset is part of the Pacific Islands Passive Acoustic Network (PIPAN) dataset [13], totaling over ten terabytes in size. The dataset comes with an *annotations. csv* file, which details 38857 annotated audio segments, including specifying them as belonging to one of the following classes: (1) background or environmental noise, (2) recording device noise, (3) fish noise that is not that of a humpback whale, (4) a humpback whale vocalization, (5) vessel noise, (6) any other sounds not specified above. We wrote a script which finds, downloads, and processes each FLAC file that is listed in the CSV by converting it to a WAV format and cutting out all annotated clips out of the complete recordings.

Roadblocks and Setbacks

The attached dataset to the paper in [11] was an invaluable resource in our project's training and testing stages. However, the format in which the dataset was made available to us and the length of the audio made it infeasible for us to use right out of the box, even on the Adroit computing cluster. Downloading the entire dataset would have taken on the order of 10 terabytes, and the inference times would have been an order of magnitude larger than our inference times in this project. Therefore, substantial effort was expended on downloading and processing each audio file individually and trimming it concerning the critical features present in that file. This served as a significant roadblock, as concrete work with regard to training, testing, and comparing models could only be accomplished after this script was finished and could process the sample files. In addition, even with the more streamlined file, the entire dataset took multiple days to be processed, even on the Adroit cluster.

Throughout this final project, our endeavors to reduce the size of the existing model presented in [11] were met with limited success, primarily due to the SavedModel format employed by the paper's authors. The format the authors used to store and distribute their model were incompatible with the pruning methods that we attempted. As a result, we opted to incorporate their model as a layer in the development of our own custom baseline model. Much of the time was allocated to creating this new model and rendering it to be compatible with our software. Additionally, since this workaround to incorporate their pretrained model in ours was not discovered until much later, we were ultimately unable to implement the knowledge distillation portion of the project.

Additionally, we encountered numerous challenges when debugging the TFLite installation process. Following the TFLite tutorial code found at [14], we faced a series of errors when executing the command `pip install tflite-model-maker`, including the unintended installation of multiple versions of the `tf-nightly` package. This particular issue has been previously documented in the literature [15] and is actively being addressed by a team of TensorFlow engineers. Despite this obstacle, we eventually resolved the issue by reverting to an earlier Python environment available in

Google Colab. This Colab approach allowed us to successfully install `tflite-model-maker` package and commence further project work after rectifying additional environment setup errors.

Results

Keras Model

While attempting to get the pre-trained model provided in [11] working, we decided to train another model using their pre-trained model as the main layer (of type KerasLayer, since we obtained their model from TFHub) and a Keras dense layer to interpret the output of their pre-trained model. We trained this model for four epochs, using approximately 3000 labeled samples as training data. Our choice of the number of epochs and samples used was primarily motivated by the training speed—this model took about two hours per epoch to train on our laptops. In the end, we obtained a training accuracy of approximately 96.18% and a testing accuracy of approximately 96.13%. However, this model was quite large—as shown in Figure 1, the model had 23,538,758 total parameters and took up 306.6 megabytes of space. As we will see in later sections, we used this model as a baseline for the performance and space tradeoff using Tensorflow or TFLite.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 keras_layer (KerasLayer)    (None, 1)                 23538753

 prune_low_magnitude_dense (  (None, 1)                5
 PruneLowMagnitude)

=================================================================
Total params: 23,538,758
Trainable params: 23,477,443
Non-trainable params: 61,315
_________________________________________________________________
```

Fig. 1. TensorFlow summary of our sequential model.

TFLite

In contrast, we trained a TFLite model with the base model being YamNet for over 100 epochs and the same 3000 training samples. This resulted in a final model of size 14.4 megabytes, 2,050 total parameters and took about 4 hours to train. The summary of this model can be seen in Figure 2.

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 classification_head (Dense)  (None, 2)                 2050

=================================================================
Total params: 2,050
Trainable params: 2,050
Non-trainable params: 0
_________________________________________________________________
```

Fig. 2. TensorFlow summary of our TFLite model.

The model achieved a training accuracy of 78.52% on training data and 76.25% accuracy on the testing dataset as well, showing that using the TFLite library to create TinyML models could be a practical approach towards training small models to be used on devices with limited resources.

Additionally, we were able to use post-training quantization to shrink the size of our TFLite model instance down to 3.9 megabytes. However, we were unable to interpret the results of this model meaningfully. Our attempts to do so via extracting the maximum likelihood heuristic from the output were unsuccessful in achieving a result different from random chance. That is, we could not find an interpretation of the raw data output by the model to anything that achieved a $> 50\%$ accuracy on the testing data.

Finally, significant effort was put into conducting Quantization-Aware-Training for the Keras model. Our training infrastructure was similar to the above framework. However, we followed the quantized aware training example process highlighted in [16]. The primary issue with running the quantized aware training code was again that the Kaggle pre-trained model, even acting as an additional KerasLayer object, was almost immutable, failing to adapt the quantization aware penalties that this training applied. Thus while QAT could modify the

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_6 (Conv2D)            (None, 220, 220, 32)      320

conv2d_7 (Conv2D)            (None, 218, 218, 64)      18496

max_pooling2d_3 (MaxPooling  (None, 109, 109, 64)      0
2D)

dropout_6 (Dropout)          (None, 109, 109, 64)      0

flatten_3 (Flatten)          (None, 760384)            0

dense_6 (Dense)              (None, 128)               97329280

dropout_7 (Dropout)          (None, 128)               0

dense_7 (Dense)              (None, 2)                 258

=================================================================
Total params: 97,348,354
Trainable params: 97,348,354
Non-trainable params: 0
_________________________________________________________________

Input shape: (222, 222, 1)
```

Fig. 3. TensorFlow summary of our VGG block model.

behavior of the added Keras Dense layer, it served no use in modifying the underlying pre-trained model behavior.

VGG-based CNNs

Having never previously considered the model size implications in real-time low-power device scenarios, we wanted to assess the storage impact of classic deep-learning models. To better understand this, we designed a simple single VGG block convolution neural network shown in Figure 3 that follows VGG block conventions and takes in the same input as the original humpback whale detection model, re-dimensioned as a 2D numpy array. This model did not undergo any parameter optimizations or any acceleration methods, and so achieved a poor but better-than-guessing testing accuracy of 68.14%. However, the more critical observations come from the poor time performance and large size of the resulting model. One training epoch took on average 462.53 seconds, and the model took up 1.18 GB of storage space. Naturally, pruning or running quantization on this model could help runtime and size, yet the model itself is not performant enough for that to be worth it. This small experiment showed us that (as expected) constructing task-specific well-optimized models is difficult. Still, more importantly, it showed that even a seemingly simple model actually takes up way more space than would otherwise be available on a nimble low-powered device.

Smarter Feature Selection

The above experiment prompted another question: why did the authors use waveforms as their model input? In other machine learning applications, condensed audio features are often used as input for deep learning models. For instance, MFCCs (mel-frequency cepstral coefficients) is an audio power spectral density representation that is shown to be a powerful feature for sound classification in language models [17]. We proposed that better features like MFCCs would enable lighter and better deep-learning models, including the original model provided with our dataset. While we cannot retrain the original model given, we can hypothesize about feature selection impacts using the VGG block-based model presented in the previous section.

The first 28 MFCCs (28x28 float matrix) were extracted from each labeled sound segment and used as input to the CNN model. Appropriately, the input dimensions to the CNN were changed to (28, 28, 1). Using MFCCs, we achieved 81.99% testing accuracy, with the final model sized at 14.6 MB and a runtime of an average of 0.24 seconds per epoch. In other words, selecting MFCCs as the model input instead of the author-selected waveform enabled us to create a model that is $\sim$80x smaller, trains $\sim$2000x faster, and generally performs better.

Even if the model using waveform features could be optimized to achieve better accuracy than the MFCC-based model, running the waveform-based model on resource-constrained devices may still prove infeasible, prompting a switch to more condensed lightweight features like MFCCs. Thus, in addi-

tion to pruning and quantization, it is equally important to step back and assess feature selection as a significant factor in defining efficient TinyML models.

Future Work

Continuing this work, we wish to explore how much smaller we can make the TFLite and quantized models. Additionally, it would have been helpful to use and analyze other methods of creating smaller ML models, such as knowledge distillation, quantization-aware training, and pruning techniques, to create a complete picture of the tradeoffs between the different methods. Additionally, since the Keras model was able to achieve a stunningly high accuracy, it would be interesting to explore knowledge distillation with the Keras model as the teacher. Furthermore, it would be interesting to investigate if an intricate relationship exists between accuracy, model size, and training time, like the efficient frontier in investment theory.

Conclusion

In conclusion, we were able to discover that many different levels of accuracy, model size, and training time could be achieved. Of particular note is the TFLite instance, which was able to reach a model size that is $\sim 5\%$ of the size of the baseline Keras model that we trained, and an astonishing $\sim 1\%$ of the size of the VGG block model. While still maintaining such a compact size, the TFLite model was able to maintain an admirable accuracy of 76.25% during inference time, which is competitive with the accuracy of 96.13% and 81.99% experienced by the Keras and MFCC models, respectively.

We also explored the viability of using different feature selectors to bolster model accuracy while dramatically reducing model size. By using MFCCs, we were able to do so, shrinking the model down to almost the size of the TFLite instance. From this we conclude that the TFLite library is certainly a great general tool for creating models with little-to-no knowledge of the underlying data or applications, especially given its access to powerful pre-trained models, like YamNet. However, with more specific knowledge and the ability to restrict the problem domain, like we were able to accomplish in this project by using MFCCs, we can apply more powerful and domain-specific tools that can dramatically increase accuracy and decrease model size. We believe that using TFLite as a baseline model can be a powerful comparison tool for new analyses and methods proposed in the TinyML field.

Contributions

Oleg Golev

For this project, my contribution was two-fold: data management and working with the VGG block-based CNN. I wrote the script and data handling handle code responsible for the cleaning and preparation of the reference dataset [11], extraction of annotated clips, and processing of the data into WAV files that were shared to other group members for their model training and inference. I also designed, ran, and assessed the VGG block-based models on their performance, runtime, and size. I ran these models with both the original waveform features used by the reference model [11] authors and MFCCs features. Finally, I wrote major sections of the report, as did the others.

Gerald Huang

My contributions to this project were primarily in setting up the machine learning code (pruning, training, testing code), debugging Python, Tensorflow, TFLite, and other machine learning modules throughout this project. I was able to get the pre-trained model in [11] up and running, as well as figure out how to use the other packages and Tensorflow, none of which I had used before, to work with my local setup and Google Colab. I wrote scripts to organize the structure of the directory to

make our codebase compatible with the format expected by the Tensorflow library functions. A large portion of my work was delayed by various issues surrounding the Tensorflow code infrastructure, especially being unable to get the `tflite-model-maker` package installed until much later. Additionally, I trained the Keras, TFLite, and quantized TFLite models while also running inferences and analyzing each model's performance. I also wrote the corresponding sections in Results detailing the outcome of my efforts, as well as the Roadblocks, Future Work, and Conclusion sections.

Vikash Modi

My contributions to this project were as follows: Defining and shaping the project scope, motivation, background, and the exploration of related works. Integrating the original model with TensorFlow, and exploring quantization and knowledge distillation techniques for the pretrained model. Overcoming challenges associated with limited prior experience in the techniques used and addressing the complexities of the pretrained Kaggle model. Investigating the underlying structure of the original model, discovering that it incorporates a short-time Fourier transform (STFT) operation before feeding the data into the ResNet CNN architecture. Attempting to separate the STFT spectrogram creation module from the network, which could lead to more efficient pruning and quantization of the model in the future. This separation also has the potential to facilitate the incorporation of the STFT operation into embedded devices independently.

References

[1] G. Research, "A convolutional neural network for automated detection of humpback whale song in a diverse, long-term passive acoustic dataset," *NOAA*, 2021.

[2] W. C. André Karpištšenko, Eric Spaulding, "The marinexplore and cornell university whale detection challenge," 2013. [Online]. Available: https://kaggle.com/competitions/whale-detection-challenge

[3] M. F. Baumgartner and S. E. Mussoline, "Automated detection and measurement of humpback whale megaptera novaeangliae song in recordings from the gulf of maine," *Journal of the Acoustical Society of America*, vol. 124, no. 5, pp. 2958–2967, 2008.

[4] P. Warden and D. Situnayake, *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media, Inc., 2019.

[5] J. D. Kinder, D. P. Nowacek, A. M. Thode, J. A. Scherrer, and J. Tippmann, "Automated acoustic localization and call association for vocalizing humpback whales on the navy's pacific missile range facility," *Journal of the Acoustical Society of America*, vol. 133, no. 5, pp. 2879–2890, 2013.

[6] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," *Advances in Neural Information Processing Systems*, vol. 28, pp. 1135–1143, 2015.

[7] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," *Advances in Neural Information Processing Systems*, vol. 28, pp. 3123–3131, 2015.

[8] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[9] G. Developers, "Tensorflow lite for microcontrollers," *Online Documentation*, 2021. [Online]. Available: https://www.tensorflow.org/lite/microcontrollers

[10] J. Lin, W.-M. Zhao, J. Wang, Y. Bai, and S. Han, "Mcunet: Tiny deep learning on iot devices," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 10 831–10 842.

[11] "humpback_whale," Feb 2021. [Online]. Available: https://www.kaggle.com/models/google/humpback-whale

[12] A. N. Allen, M. Harvey, L. Harrell, A. Jansen, K. P. Merkens, C. C. Wall, J. Cattiau, and E. M. Oleson, "A convolutional neural network for automated detection of humpback whale song in a diverse, long-term passive acoustic dataset," *Frontiers in Marine Science*, vol. 8, 2021. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fmars.2021.607321

[13] N. P. I. F. S. Center, "Pacific islands passive acoustic network (pipan) 10khz data," 2021. [Online]. Available: NOAANationalCentersforEnvironmentalInformation

[14] "Transfer learning for the audio domain with tensorflow lite model maker." [Online]. Available: https://www.tensorflow.org/lite/models/modify/model_maker/audio_classification
[15] Tensorflow, "I failed to install tflite-model-maker · issue #53550 · tensorflow/tensorflow." [Online]. Available: https://github.com/tensorflow/tensorflow/issues/53550
[16] ——, "Quantization aware training." [Online]. Available: https://www.tensorflow.org/model_optimization/guide/quantization/training_example
[17] Z. K. Abdul and A. K. Al-Talabani, "Mel frequency cepstral coefficient and its applications: A review," *IEEE Access*, vol. 10, pp. 122 136–122 158, 2022.