Cornell Birdcall Identification

W251 Deep Learning in the Cloud and at the Edge – Summer 2020
Flying Toasters – Abhinav Sharma, James Byrne, Carlos Capellan, Isaac Law

1. Introduction

Cornell Birdcall Identification is a recent Kaggle competition. The competition challenges entrants to detect bird species in long soundscape recordings of 10 minutes in length. As birds can be found in nearly every environment and biome on Earth, the number of unique bird species in a given environment is an excellent indicator of habitat quality and environmental pollution. The main goals of the project are to build a deep learning model to identify a wide variety of bird vocalizations and deploy the model on the Nvidia Jetson TX2. This application of deep learning technology will help researchers better understand changes in habitat quality, pollution levels, and the effectiveness of restoration efforts.

2. Data

The dataset is a labelled dataset consisting of birdcall audio and the corresponding bird species as a 6 letter "alpha code" used by ornithologists as a standard shorthand. This dataset was collected by Cornell Lab of Ornithology's Center for Conservation Bioacoustics. There are 264 bird species in the data and 21,375 audio files.

We first scaled down the dataset to 20 bird species and performed extensive analysis on these 20 species, then we had a final training on all 264 bird species. Figure 1 shows the distribution of the 20 labels. Our dataset is a relatively balanced dataset, there are roughly 80 to 100 audio clips per each bird.

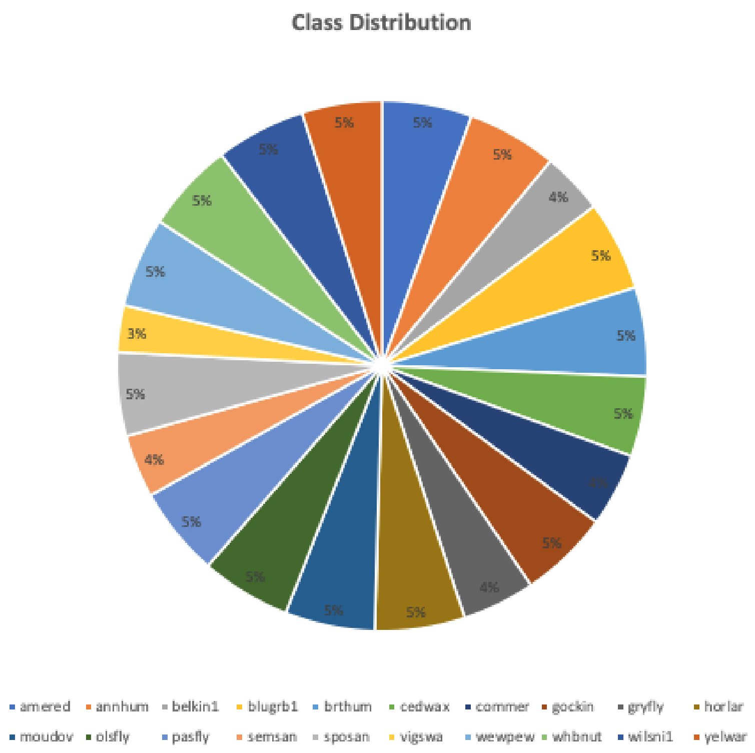


Figure 1 - Class Distribution of 20 Bird Species

3. Tools

IBM Cloud Object Storage

We have all 22 GB of raw audio data and 80 GB of Mel Spectrogram images saved in IBM Cloud object storage, allowing team members to easily retrieve data for processing and training via s3fs-fuse.

IBM Cloud Virtual Machines

We provisioned virtual machines with Nvidia P100 and V100 GPUs for model training. As our dataset was scaled up by more than 4 times after data augmentation, model training would not have been possible without the drastic performance improvements of P100 and V100 GPUs over regular CPUs.

Nvidia Jetson TX2

Our trained TensorFlow model is deployed using a pre-built Nvidia container on the Jetson TX2 for inference, our aim is to have a low-power edge device that is capable of running our model and able to be deployed in remote locations and/or harsh conditions. A standard Logitech USB webcam is attached to the Jetson, and the container running the inference code accesses the audio channel of the webcam to record live audio samples for inference.

4. Exploratory Data Analysis

The dataset consists of raw recordings of different lengths, the target birdcall can appear in any time interval of the recording, and there is often ambient noise in the sample. We first look at the distribution of the audio recording lengths.

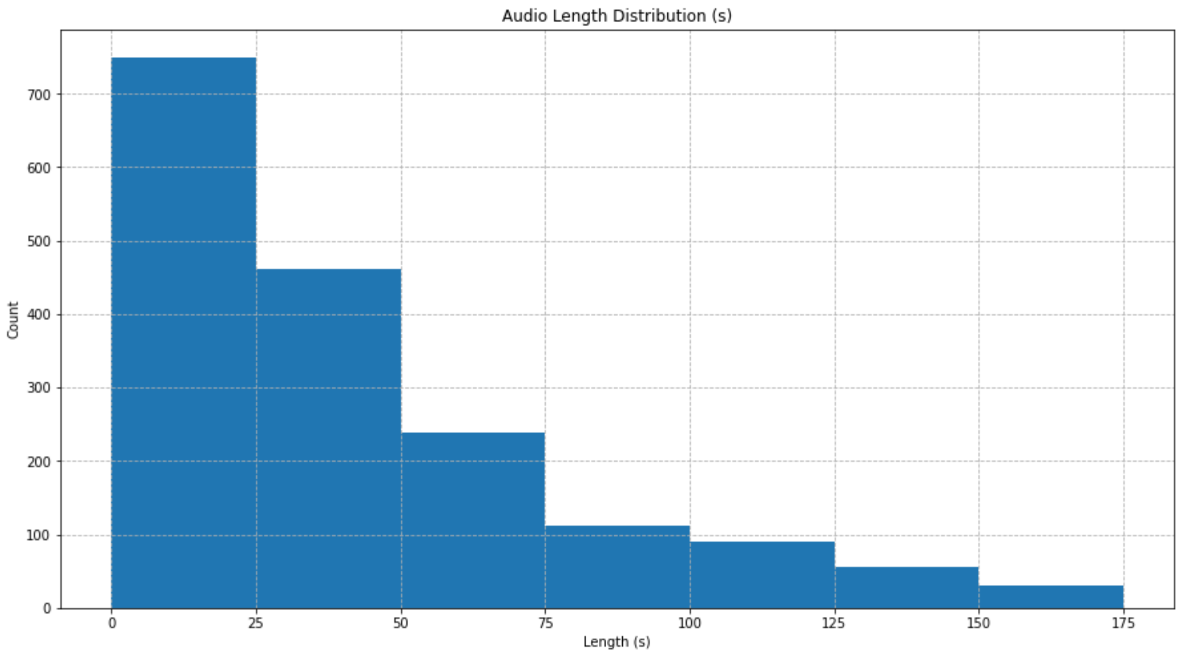


Figure 2 - Histogram of Audio Recording Lengths

Most of the audio samples are 25 seconds or less. By randomly listening to audio clips from different birds, we observed that the target birdcalls usually appear in the first 10 seconds of audio. Therefore, we started our model training by building models for classifying bird species based on the first 10 seconds of audio. Also, a 10 second audio interval is a reasonable choice from an inference standpoint. After some initial success building models for classifying the first 10 seconds, we applied a sliding 10 second window shifting 5 seconds each time to create a new set of training data, and we used this approach to build a more robust model for inference.

Next, we explored time series plots of a 10 second randomly selected audio clip for each bird.

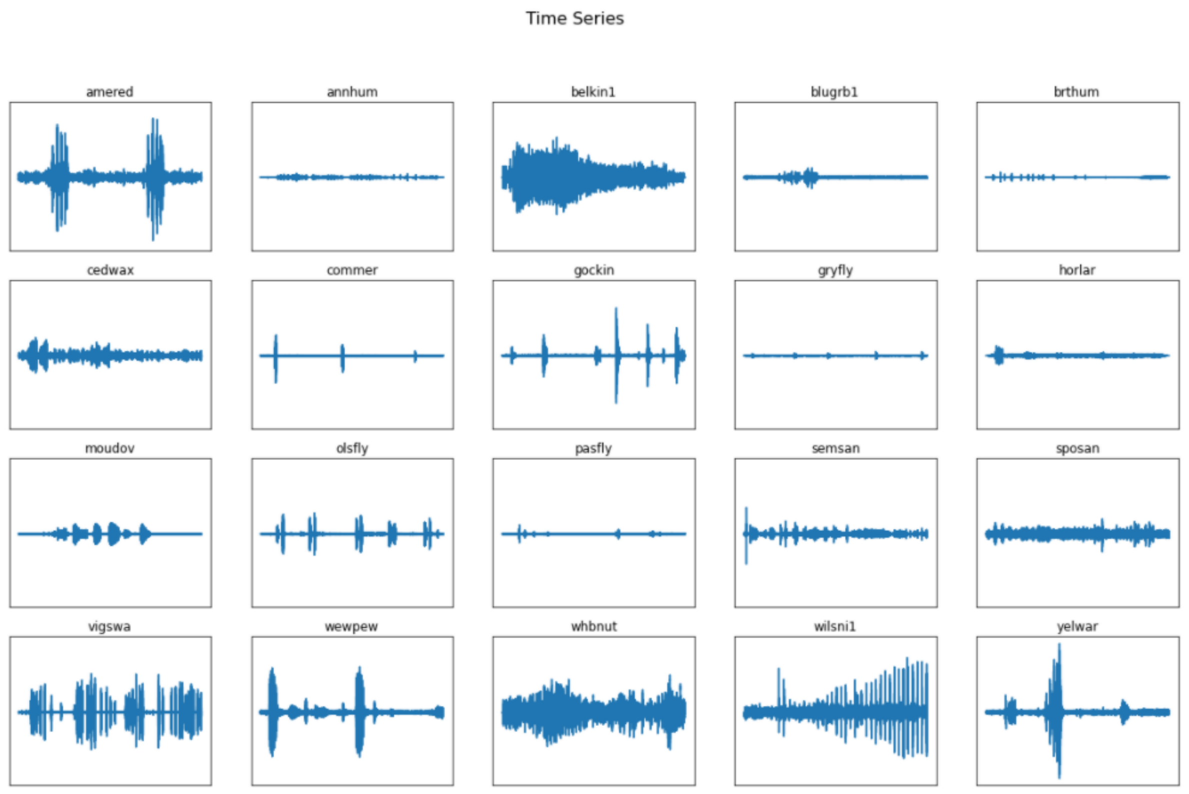


Figure 3 – 10 Second Spectrograms of a Randomly Selected Recording For Each Bird Species

The time series plots show the fluctuation of amplitude over time, we can see that there is quite a lot of information inside the first 10 seconds of audio, this reinforces our choice of a 10 second audio length for training.

However, there are 2 problems with this time series plot.

- Time series plots only show fluctuations of amplitude over time, these plots do not give us any information about the frequency spectrum of the birdcall.

- Time series plots are a sparse representation of an audio sample. Assuming a common sampling rate of 22,050 Hz (cycles per second), a 10 second audio sample is a one-dimensional vector with length 220,500. It would be difficult for a machine to learn and extract useful information with only this data.

To solve this problem, we use a common technique in audio analysis: the mel spectrogram transformation. An ordinary spectrogram is a visual representation of frequency (pitch) and amplitude (loudness), with the frequency on a log-scale Y axis and the amplitude illustrated by a color scale.

A mel spectrogram applies the Mel scale to the Y axis; it is scaled to how human perceive audio frequency differences. We are better at noticing differences at lower pitches than at higher pitches, pitch changes that we perceive as equidistant are in fact larger and larger pitch increases.

To create a mel spectrogram, we first perform a fast Fourier transform (FFT) on the audio data, this step transforms audio into a frequency representation. Next, we stack these frequency representations over time (in Mel scale) and use a color scale to represent power fluctuation. Figure 4 shows the mel spectrograms of our sample audio from 20 birds.

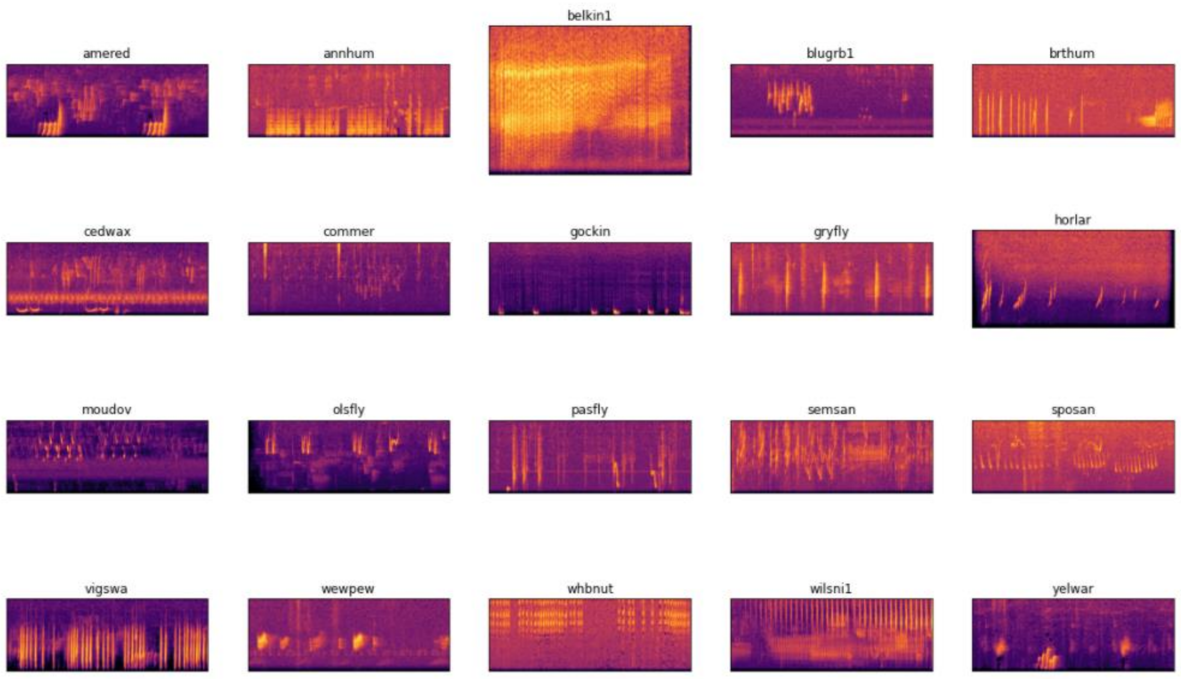


Figure 4 - Mel Spectrograms of Figure 3 Recordings

We can see this representation of data is more meaningful than a simple time series plot. Each plot shows the fluctuation of different frequency bands and power over time.

By converting audio data into mel spectrogram, it solves two challenges we had before:

- The mel spectrogram represents the change of frequency (in Mel scale) over time, as each bird has different changes in frequency in their calls by observing the patterns in mel spectrograms we should be able to classify birds.

- Mel spectrograms compress information from 1-D vector in 220,500-dimensional space into a 2-D representation of 224 pixels by 625 pixels. With a 2-D representation, we can perform standard deep learning image classification techniques, such as a 2D convolutional network, for training.

To better illustrate mel spectrograms, Figure 5 shows three representative samples from three bird species: the American Redstart (amered), the Olive-sided Flycatcher (olsfly), and the Violet-green Swallow (vigswa).

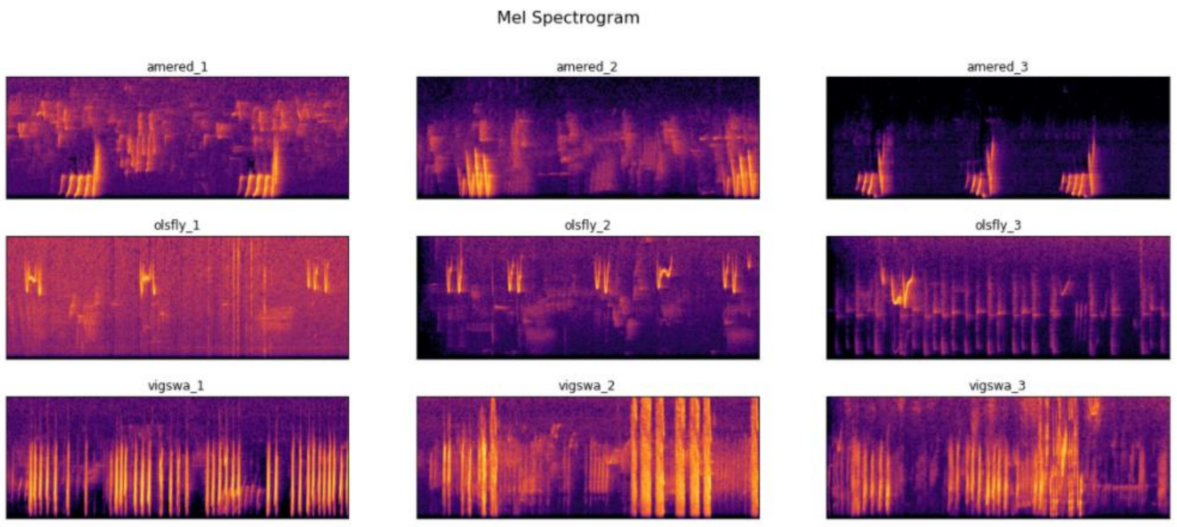


Figure 5 - Representative Mel Spectrograms for 3 Bird Species (amered, olsfly, vigswa)

We can see the American Redstart's sound has generally high pitch, and usually ends with a lower pitch fluctuation. The Olive-sided Flycatcher's sound usually in a very specific frequency range, while the Violet-green Swallow's sound has very wide range of frequencies. At this point, we can consider these distinctive patterns as "fingerprint" of bird, and this can be used for building classification models.

5. Data Preparation and Augmentation

The original dataset consists of raw recordings from different environments. As the recordings are made in the wild there is ambient noise, which in some cases emerges very prominently. In addition to the bird calls, recordings may have airplane noise, the bird calls of other species, or human speech. Given that bird calls are generally all higher frequency sounds, we applied a high pass filter at 1400 Hz as mean of noise filtering.

There were two paths for augmentation approaches, one was to transform the audio files themselves and the other was to transform the resulting spectrograms. We decided on the former, transforming the audio and then generating mel spectrograms from the transformed audio files.

The key reason for this decision was the nature of the problem we were trying to solve. Fundamentally, it is an audio recognition problem, even though we are using object detection and recognition techniques on the mel spectrograms of the audio. When doing object detection in a picture, a cat is a cat whether it is looking left, looking right, or even upside down. For audio, pitch and time is fundamental to what we are trying to detect. Image techniques such as rotation or flips applied to the spectrograms would inhibit identification as they could turn a rising series of calls into a falling one. Unlike the cat example, flips and rotations of the mel spectrogram would change the characteristics of the call we are attempting to learn and identify and additionally would never be encountered when performing inference.

We would expect the speed that multiple parts of the call are made and the pitch of the calls might vary somewhat within a species, but only by a relatively small amount, so we restricted transformations such as zooming (stretching out the audio), time shifting (moving the call earlier or later in the recording) and frequency shifting (making it a higher or lower tone) to limited random ranges.

In retrospect, there was a flaw in this initial reasoning. Although the pitch of a call (the vertical axis in the spectrogram) would have a narrow range, the time within the recording (the horizontal axis) when the call is heard could vary wildly, or even include multiple calls. Increasing the time-shifting brackets could have made the network more resilient to calls appearing at different points within a recording, as would implementing automatic extraction of each call instance from the longer recordings.

We somewhat addressed this in the inference step where we take overlapping slices of the example audio we capture, so the calls would appear in different portions of the audio clips that we processed.

Handling of Stereo Samples

The recordings in the data set are a mixture of stereo and mono recordings. To ensure consistency, stereo recordings were split into two separate mono recordings. This would result in two slightly different examples to train on. The worst case would be that a mono recording had been encoded as stereo with identical left and right channels, resulting in a duplicate.

Tools Used for Non-Neural Augmentation

The library used was Audiomentations[1] which is a library of audio transformations, similar to the `imgaug` libraries for image transformation. It offers the following transformations:

1. Frequency Masking – eliminating bands of frequencies to enhance generalization
2. Time Masking – replacing segments of time with silence
3. Gaussian Noise – adds Gaussian noise within specified amplitudes

[1] Repository at https://github.com/iver56/audiomentations and is available via pypi: "pip install audiomentations"

4. Add Noise – adds white noise within specified amplitudes
5. Time Stretch – expand the length of the recording, stretching out the audio without altering the pitch
6. Pitch Shift – randomly shift the frequency of the audio up or down, limited to specified numbers of semitones.
7. Shift – shifts the recording time earlier or later – again, random times are applied within specified limits
8. Normalization – retransform the audio so the loudest sound corresponds to 0dbf, or +1.0 in a float array. Not used as the samples are normalized while generating the spectrograms.
9. Resampling – recreate the recording using a different sampling rate than the original.
10. Add Short Noises – overlay other sound files in short bursts to simulate real-world situations. Not a candidate for this exercise as the recordings already have background noise of various types included with the bird calls.

Table 1 shows the augmentation methods we selected. We did not evaluate combination augmentations.

Augmentation	**Details**
Time Shift	Randomized ± up to 500 milliseconds
Frequency Shift	Randomized ± up to 2 semitones
Time Stretch	Randomized up to 20% of the original
Gaussian Noise	Amplitude ranges from 0.005 to 0.24

Table 1 - Augmentation Methods Used

The most successful augmentation in terms of accuracy enhancement was Gaussian Noise, but only at moderate levels. Testing using the first example CNN model we tested (V1) resulted in the accuracy shown in Table 2.

Augmentation	**Accuracy from v1 model**
Original, unaugmented samples only	76.1%
Original plus time-shifted samples	82.3%
Original plus frequency shifted samples	82.3%
Original plus time-stretched samples	84.6%
Original plus gaussian noise (0.015) samples	88.3%

Table 2 - Accuracy of Selected Augmentations vs. Original Samples

We ran multiple tests at different levels of Gaussian noise amplitude between 0.005 and 0.24, and the best results were obtained using the 0.015 shown in the table above.

As can be seen from the noticeable improvement to accuracy regardless of the augmentation method, simply increasing the training data available had a noticeable impact on how well we could train the models.

Applying the selected augmentations provided a roughly 400% increase in the samples available. We were careful to ensure that the original and the transformed recordings derived from each only appeared in either the training or test datasets. Before this was done, we had some artificially high accuracies (>93%) as we were validating against what were only slightly altered versions of training examples.

Specific Size Mel Spectrograms

As noted in section 6 below, some of the models we tested expected certain image resolutions, generally as a square image. As the audio recordings were of differing lengths, we prepared multiple versions of the mel spectrograms from the augmented audio clips in 224x224, 299x299 and 331x331 versions. This was an alternate approach to sampling time-slices of the variable length recordings in the dataset, but also had the effect of compressing by differing amounts, as the variable length of each recording was mapped to a fixed-width in the spectrogram.

Convolutional Auto Encoders for Augmentation

Inspired by several articles suggesting that auto encoders and/or variational auto encoders could be useful mechanisms for data augmentation,[2] we attempted to apply the technique to the mel spectrogram images produced from the audio files.

Most examples available[3] operated on the MNIST digit dataset, and were somewhat successful, but it proved difficult to effectively scale up these models for use on the mel spectrograms. Firstly, the MNIST features are generally centered and take up much more of the available frame than the birdcalls do in the spectrograms, worse, the birdcalls could be at any point on the horizontal axis, or even occur multiple times. Secondly, there are significant differences in data size: even the smallest 224x224 spectrograms had 64 times the pixels of the MNIST digits. The mel spectrograms use colors and saturation to indicate the intensity at each frequency, so collapsing the RGB color channels and quantizing the data to black & white as done in the MNIST models to cut the data volume would eliminate a significant amount of the information in the recordings.

In the examples below, moving to grey scale and then quantizing to black and white removes a lot of detail. In the first example, losing all the information below the mid-range frequencies may be OK for feature recognition training, but this is a particularly loud and distinctive White-breasted Nuthatch call. The second example is of the less distinctive and more melodic call of the Horned Lark. Eliminating color and quantizing erases virtually all relevant data for the Lark's call.

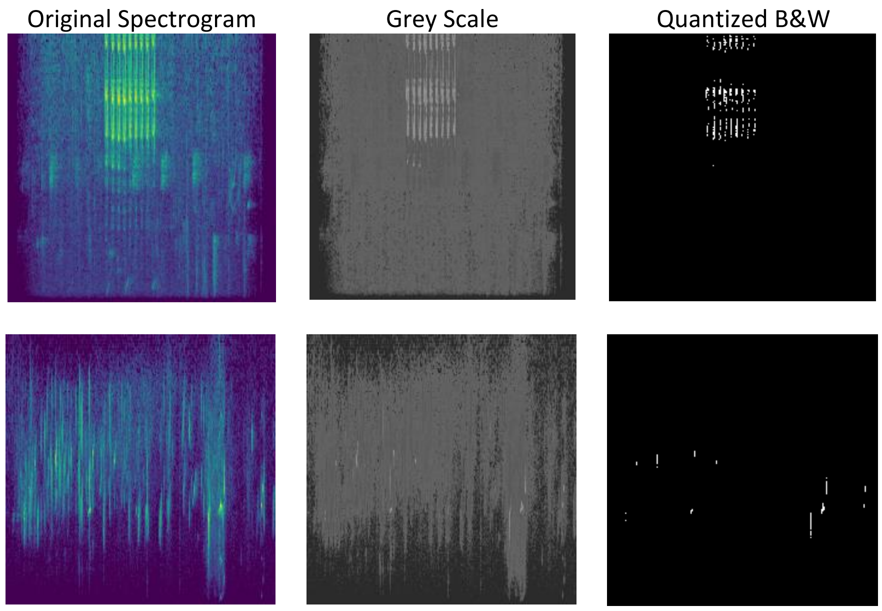


Figure 6 - Testing of Autoencoders with White-breasted Nuthatch and Horned Lark Examples

Examples we have previously used with MNIST could practically use a 28x28=784 node input layer. This became less practical for a 50,000+ node input layer, so this pointed towards a convolutional autoencoder approach, which we were able to build successfully. The decoder had two convolutional layers, outputting to a feature layer that did not have an activation function. The encoder reversed the process, finally reshaping back into the (224, 224, 4) image we wanted. Note that we have four channels ("RGBA") as the PNG files we used have an alpha channel as well as the standard RGB channels.

The results were not good – essentially creating an extreme blur transformation, and the resulting accuracy when using the generated images was significantly worse than the result from the smaller, unaugmented dataset. After many attempts to improve the quality of the generated images – including much larger, slower networks, we abandoned this approach.

[2] Examples included https://quantdare.com/variational-autoencoder-as-a-method-of-data-augmentation, and https://tiao.io/post/tutorial-on-variational-autoencoders-with-a-concise-keras-implementation

[3] TensorFlow's own example using MNIST: https://www.tensorflow.org/tutorials/generative/cvae

6. Modelling and Result

We were originally inspired by the blog post “Sound-Based Bird Classification: How a group of Polish women used deep learning, acoustics and ornithology to classify birds”[4] that eventually informed our methodology for model training. The Polish team had transformed the bird classification problem into an image classifier problem by converting the audio files into mel spectrograms.

The model in the article focused on EfficientNetB3 as the model that was ultimately used in production. We wanted to explore other classifiers for benchmarking and validation purposes. Our methodology was to generalize the classification model implementation by providing a range of different “headless” classifiers for the data to train on. We wanted to leverage transfer learning by exploiting the feature extraction capability that had already been provided to us by these pre-trained models. We did recognize, however, that the pre-trained models had been trained on generalized data sets, such as “image net” or “inaturalist” - which may or may not translate well when performing transfer learning against the mel spectrograms. As expected, there were barely any pre-trained models that were trained exclusively on mel spectrograms that we could have used for our transfer learning task. Part of the discovery process was to figure out how “robust” the transfer learning process is, particularly when transferring context between datasets that may not have much in common.

The easiest way we found when implementing this was to use the different headless models provided by TensorFlow in their model repository. We generalized the classifier implementation by allowing the user to specify the headless model they wanted to train on. This would download the corresponding model from TensorFlow Hub and run the classifier against the pre-built model. We tried several different classifiers including ResNet, MobileNet, EfficientNet, InceptionNet and PNasNet. Each of these models had several versions available, however, we picked the latest iteration (number 4) within each model category. We did not have the compute resources or the time to try out all possible iterations of each model category.

This approach was chosen for several reasons:

- Each of these classifiers has already been pre-built and tuned appropriately for a wide range of image classification use cases. The idea was to just use these off-the-shelf models to see how well they performed.
- TensorFlow also allows for us to train *n-1* layers if need be (unfreezing *n* lower layers), such that we can get even more fine-tuned results when training our dataset.
- The models provided are of different classification architectures. This allows us to draw conclusions around which classification architecture works best for this particular classification task. For example, MobileNet uses the Xception architecture while the ResNet models use a residual architecture when defining the neural network. It would be interesting to see which models worked better in classifying mel spectrograms, and this is what we set out to do as part of this task.
- We can also quickly and efficiently benchmark our classification using this technique, as opposed to defining each of the architectures ourselves by hand.

There were several hyperparameters (and other parameters) we tuned as a part of modeling this problem. These are classified as performance parameters, model selection parameters and infrastructure parameters:

Performance Parameters

- Number of CPU Workers – this dictates the number of CPU cores that will be used to load the data during training time. This is used for parallelization of data processing. Since we have 8 CPU cores on our P100 or V100 virtual machine, we use 6 workers for faster loading.
- Steps per Epoch – this dictates the number of steps that are taken in a single epoch of model training. If there are a higher number of steps taken per epoch, the CPU must load data more times.
- Epochs – this dictates the number of times the model iterates through all the training data. If there are a higher number of passes through the full dataset, the more the model can train on the data and the higher the expected accuracy. However, too many epochs can lead to overfitting during training.
- Batch Size – dictates how many images are loaded in one batch of the training. Models that have a very high number of parameters (on the order of 10^6 or more) need to have a lower batch size, or else we

[4] https://towardsdatascience.com/sound-based-bird-classification-965d0ecacb2b

risk hitting memory exhaustion errors during training. Ideally, we want to have as high a batch size as possible so we can train our model quickly.

- Augmentations – these are the various augmentations that were performed on the original mel spectrograms – which include pitch shift, frequency shift, adding background noise to the spectrograms, and involving sliding window samples in the training process. Details of the augmentation techniques have been described in section 5.
- Optimizer – we tried RMS Prop and Adam in our model training. We decided to choose Adam as it was able to converge faster on most of our models.

Model Selection Parameters

- Headless Model – this determines which model is chosen to do the training. We chose a variety of models ranging from MobileNet, ResNet, InceptionNet and PNasNet.
- Trainable Layers – controls the number of layers that are allowed for tuning during the training process. For simplicity, we tried two settings – to freeze all layers except the classification layer and to unfreeze all layers in the entire neural network. The downside with unfreezing all layers is that the model takes longer to train and we reduce the effect of transfer learning. However, with this setting we do expect higher accuracy scores on the training process since we customize training on the mel spectrograms.
- Image Size – Classification models expect different image sizes when training images. For example, ResNet and MobileNet expect images to be of the form (224, 224), whereas InceptionNet expects image size (299, 299). Similarly, PNasNet expects images of size (331, 331). These are important criteria to keep in mind when providing image inputs. Padding images of size (224, 224) to fit (299, 299) is not optimal for training. The model can give inaccurate results if the correct input image size is not provided. Thus, we decided to perform augmentations on the images for each of the unique size combinations expected.

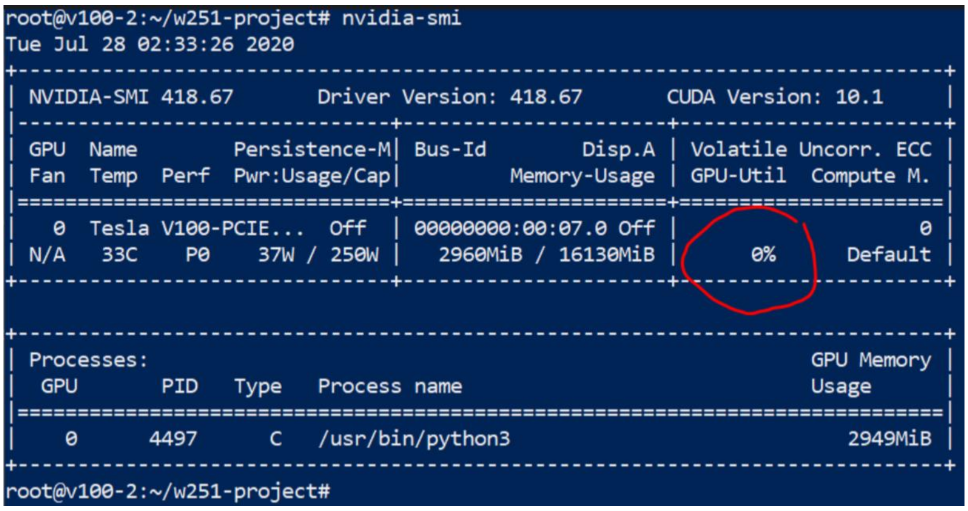
Infrastructure Parameters

- VM selection – we ran the models on both P100 and V100 virtual machines. These have different Nvidia GPU architectures (Volta vs Pascal) and we expected higher performance on V100 VMs. We noticed that performance improved slightly when training on the V100, but given the amount of training data involved (tens of thousands of images), an average epoch took around 7-11 minutes on the P100, which was not a large overhead when training on tens of epochs.
- CUDNN – we installed CUDNN and CUDA on our machines to ensure that the training process was using the full GPU capabilities.
- GPU customizations – we ensured that the computations were landing on the GPU and that TensorFlow had full visibility to the GPU on the machine. We also set the “allow_growth=True” parameter for memory so that the model was only reserving the memory it “needed” and not eagerly.

Finally, there were a few pitfalls we encountered that gave us some insights as we iterated on our training. Some of these were obvious learnings, whereas others were more subtle in nature.

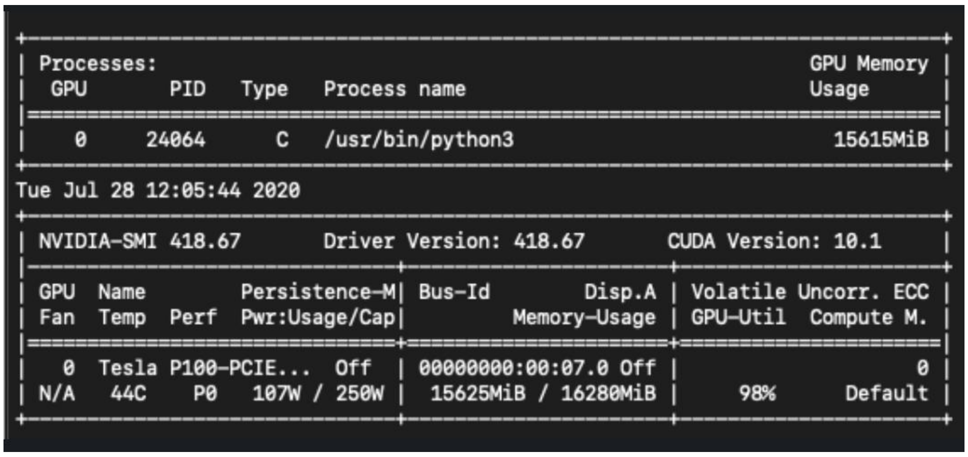
We ensured that we performed the augmentations ahead of the training process so that we were not augmenting images as we were training on them. This becomes ever important in the scenario where the data is not collocated with the compute, which is the case in our training. As we mounted the object storage on our virtual machine, performing augmentations during training would have been very inefficient as the CPU would be involved in augmenting the images, then syncing them over the network to object storage.

We also ensured that the compute was in the same datacenter as the storage. This may seem obvious in hindsight, but we experienced the consequences of failing to do this. Our object storage is in the Tokyo datacenter (JP-Tok), but we were initially provisioning our training VM in London. The result was that we were seeing extremely high training times per epoch (over 4 hours per epoch). We realized that the network was the bottleneck when we provisioned the same VM running the same model in the JP-Tok datacenter and reduced our training time per epoch from 4 hours to 3 minutes! The root cause here was that a VM provisioned in London was mostly busy trying to fetch the batch images over the public Internet during training and was hardly leveraging the GPU for deep learning training. We could observe this by seeing 0% utilization on the GPU persistently over long periods of time. Figures 7 and 8 shows examples of low and high GPU utilization between the London and Tokyo VMs, and network util for the VM in London. Figure 9 shows the dramatic difference in network utilization between the two VMs, over 26 times the traffic.



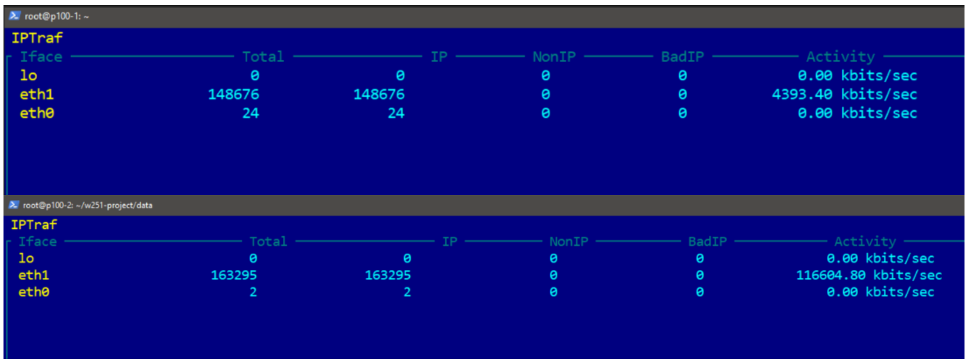
```
root@v100-2:~/w251-project# nvidia-smi
Tue Jul 28 02:33:26 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.67       Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-PCIE...  Off  | 00000000:00:07.0 Off |                    0 |
| N/A   33C    P0    37W / 250W |   2960MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      4497      C   /usr/bin/python3                            2949MiB |
+-----------------------------------------------------------------------------+
root@v100-2:~/w251-project#
```

Figure 7 - Low GPU Utilization for London VM



```
+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0     24064      C   /usr/bin/python3                           15615MiB |
+-----------------------------------------------------------------------------+
Tue Jul 28 12:05:44 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.67       Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  Off  | 00000000:00:07.0 Off |                    0 |
| N/A   44C    P0   107W / 250W |  15625MiB / 16280MiB |     98%      Default |
+-------------------------------+----------------------+----------------------+
```

Figure 8 - High GPU Utilization for Tokyo VM



```
root@p100-1: ~
IPTraf
 Iface ──────────────── Total ──────────────── IP ────────── NonIP ────────── BadIP ────────── Activity ──
 lo                        0                    0               0                0          0.00 kbits/sec
 eth1                 148676               148676               0                0       4393.40 kbits/sec
 eth0                     24                   24               0                0          0.00 kbits/sec

root@p100-2: ~/w251-project/data
IPTraf
 Iface ──────────────── Total ──────────────── IP ────────── NonIP ────────── BadIP ────────── Activity ──
 lo                        0                    0               0                0          0.00 kbits/sec
 eth1                 163295               163295               0                0     116604.80 kbits/sec
 eth0                      2                    2               0                0          0.00 kbits/sec
```

Figure 9 - Network Utilization of Tokyo (top) and London (bottom) VMs

Finally, we also optimized network latency by mounting our augmented data files against the private endpoint URL for object storage, as opposed to the public endpoint.

Public Endpoint mount command:

```
sudo s3fs audiodata /root/w251-project/data -o passwd_file=$HOME/.cos_creds -o sigv2 -o use_path_request_style -o url=https://s3.jp-tok.cloud-object-storage.appdomain.cloud
```

Private Endpoint mount command:

```
sudo s3fs audiodata /root/w251-project/data -o passwd_file=$HOME/.cos_creds -o sigv2 -o use_path_request_style -o url=https://s3.jp-tok.private.cloud-object-storage.appdomain.cloud
```

The latter command ensures that the traffic for communication between the VM and object storage does not leave the internal datacenter network and go over the Internet, thus saving round trip times for packet transfers. However, we did not get a chance to directly measure the improvement.

Results

We observed that the best performing model category was EfficientNet compared to all other model classes. We tried to run the model against several other classes of models (ResNet, InceptionNet, MobileNet and PNasNet), however, it did seem that with all possible configurations, EfficientNet seemed to be performing the best. The poorest performing architectures were InceptionNet, ResNet and PNasNet. The latter two had single digit validation accuracy numbers, whereas InceptionNet also did not perform much better. It is interesting to note that these models underperformed severely compared to their peers, even though they were pre-trained on the same ImageNet dataset (just like the other model categories).

Model Name	**Best Validation Accuracy**	**Comments**
EfficientNetB0	0.5750	This is the most basic EfficientNet with 5.3M parameters. This simple model still outperformed most other architectures.
EfficientNetB1	0.7681	
EfficientNetB2	0.6221	
EfficientNetB3	0.7923	High performance coincides with the original model proposed by the Polish data science team
EfficientNetB4	0.6386	Trained quickly with 19M parameters.
EfficientNetB5	0.8047	Took much longer per epoch for training compared to B3 and B4 (1000 + sec compared to 500 sec). Has around 30M parameters for training.
EfficientNetB6	0.8063	Took much longer per epoch compared to B3 and B4. This is the highest performing model with 43M parameters.
EfficientNetB7	0.6102	Took much longer per epoch compared to B3 and B4. Has around 60M parameters and performed worse compared to its “less complex” counterparts with fewer parameters.
InceptionNet V3	0.2794	
MobileNet V2	0.4856	
MobileNet V2 140 Layers	0.4037	
ResNet V2 152 Layers	0.0569	Uses traditional skip connections between wider layers (residual block), which seems to be a less favored architecture than EfficientNets that use an inverted residual block (skip connection between narrower layers)
PNasNet	0.0640	

Table 3 - Accuracy of Pre-Trained Models Against Validation Dataset

Here are some observations we made around model performance from the table above:

1. There was something fundamentally different about the EfficientNet architectures when compared to the worse performing models (InceptionNet, ResNet, PNasNet) that led to such a drastic disparity of outcomes in prediction accuracy. This is something that we briefly explore in section 7.
2. Even within EfficientNet, we observe that the validation accuracy varies based on which architecture is chosen. We would have hypothesized that as model complexity and size increases (from EfficientNetB0 to EfficientNetB7), we would see a higher validation accuracy. However, we observe that this is not necessarily the case. For example, EfficientNetB7 performs worse in validation accuracy compared to EfficientNetB3 - B6. We note that EfficientB6 gives us the best validation accuracy, although it is very close to B3 and B5.

3. EfficientNet models use the "swish" activation function[5] during training, whereas other models use more traditional activations – ReLu, Tanh, etc. The swish activation has been associated with higher accuracy results on ImageNet. This could also have helped in increasing accuracy for EfficientNet based models on our dataset, although by a small amount.

According to several blog posts,[6] the outstanding performance of EfficientNets comes as no surprise. EfficientNet allows for principled parameter scaling (along depth, width and resolution) that allows for more compact models (much smaller than parameter heavy models like ResNet) and higher performance. Their compactness makes them desirable for quick inference tasks as well. For example, EfficientNets have already reached state of the art classification on ImageNet and are quickly becoming the standard for all computer vision tasks. The blog posts also mention that these models are good at transfer learning and are able to effectively translate to other datasets (such as CIFAR-100 and Flowers). Hence, we were confident that these class of models would be able to fit our bird sound classification task.

Further work in the future would involve a detailed study of the architectural differences between these models and understanding why one performed much better than the others. Another interesting analysis to carry out will be to study the difference in model performance between the different versions of EfficientNet, when looking at these models from the point of view of their different architectures.

7. Error Analysis and Model Comparison

We observed that certain models performed much worse than EfficientNet, our best performing model for this classification task, and here we briefly comment on the architectural differences between the underperformers compared to EfficientNet. A *Towards Data Science* blog post[7] gives an overview of the different architectures of the ResNet and InceptionNet networks we tested in our training.

Scaling the model along the depth (i.e., number of layers) or width (i.e., number of filters in a convolution layer) or resolution of image can lead to increased gains, but only to a certain point.[8] Accuracy gains start diminishing beyond a certain point (by only increasing any one of these 3 dimensions) when the models are already large. This implies that the scaling of network for increase in accuracy should be contributed in part by a combination of the three dimensions. EfficientNet allows for compound scaling by grid searching the right combination of depth, width and resolution in order to fit the computational constraints (FLOPS) of the machine, which the other models do not do; this is the key architectural difference in EfficientNet vs other models. Interestingly enough, this concept of compound scaling can be applied to other models – ResNet, InceptionNet etc. – however, in their original headless form, these architectures do not have compound scaling enabled on them. Part of future work will be to allow for compound scaling and observe performance improvement for these poor performers.

In the case of our worst performing models (ResNet 152 and PNasNet) – the models were already extremely large (80M parameters for ResNet and 82M for PNasNet compared to EfficientNets which are between 10-40M parameters). These poor performers have static architectures and do not allow for scaling of the 3 dimensions simultaneously in an appropriate way to allow for increased accuracy gains during training. This could be one factor that contributes to their subpar performance.

In the case of InceptionNet, which performed better than ResNet and PNasNet (27% vs single digit percentages), we observe that InceptionNet had around 22M parameters, which is far less than parameters in ResNet or PNasNet. Similarly, MobileNet V2, which performed much better than ResNet or PNasNet (40% vs single digit percentages), has only 2.5M parameters, far less than the poor performers with much larger parameter counts.

[5] https://medium.com/analytics-vidhya/image-classification-with-efficientnet-better-performance-with-computational-efficiency-f480fdb00ac6
[6] https://ai.googleblog.com/2019/05/efficientnet-improving-accuracy-and.html and https://towardsdatascience.com/efficientnet-scaling-of-convolutional-neural-networks-done-right-3fde32aef8ff
[7] https://towardsdatascience.com/illustrated-10-cnn-architectures-95d78ace614d
[8] https://towardsdatascience.com/efficientnet-scaling-of-convolutional-neural-networks-done-right-3fde32aef8ff

Of course, we know that EfficientNet models are also smaller than ResNet / PNasNet and outperformed them. Model size itself could be related model performance, in this case, a much larger model is underperforming in validation accuracy. One possible explanation that comes to mind is that the poor performers were overfitting the training data. This is entirely possible given that our training data was only in the tens of thousands of images, which is quite small when compared to these large model sizes (80M parameters). If we had perhaps millions of images, these poor performers could have done better.

Lastly, a less convincing reason for their underperformance could be the architectures of ResNet, InceptionNet and PNasNet. It is possible that the residual architecture in ResNet was optimized for addressing the vanishing gradient problem and for quicker backpropagation and not necessarily for better image classification. Similarly, it is possible that the Inception module for InceptionNet was designed for better computational performance (depth wise convolution, followed by point wise convolution for model parameter reduction) but not to address higher image classification accuracy. In other words, it is possible that there are genuinely better image classifiers compared to these poor performers, and the unique architectural improvements for these models are not actually contributing to image classification accuracy, but rather optimizing for something else. Another noteworthy point is that ResNet-based architectures use the traditional skip connection concept – where the residual skip connections are between larger, wider layers.[9] EfficientNet effectively reverses this, skip connections are between narrower blocks. This could also be contributing to a better classification in EfficientNet versus ResNet models, but this requires further testing and validation.

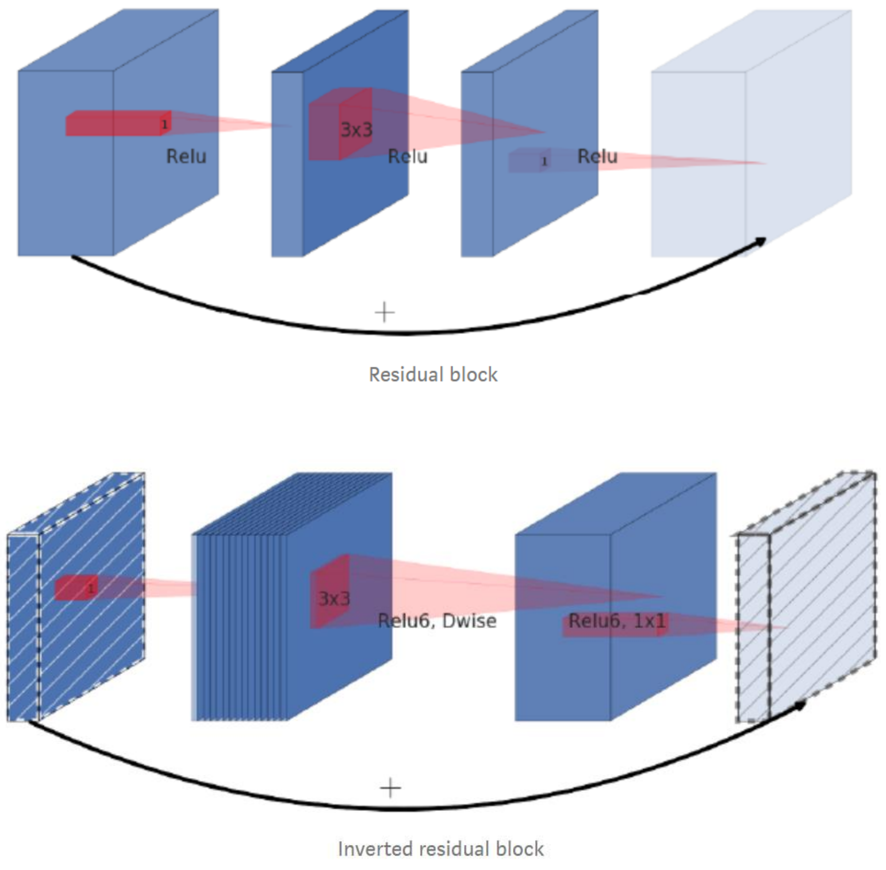


Figure 10 - Residual Block vs. Inverted Residual Block Architecture

It is also possible that transfer learning (due to being trained on the ImageNet dataset) for these architectures is less effective with mel spectrograms compared to transfer learning from (say) MobileNet. However, there is not strong evidence for this theory and more research is needed to prove or disprove.

[9] https://medium.com/analytics-vidhya/image-classification-with-efficientnet-better-performance-with-computational-efficiency-f480fdb00ac6

8. Edge Inference

Edge inference with the Jetson TX2 enables the trained model to be used in field conditions. A researcher could bring a Nvidia equipped device to the field and conduct live inference of bird species *in situ;* the scientific value of such a tool is undeniable. The obvious constraints of low or no bandwidth, battery life, and extreme weather conditions seem to warrant a low power edge device.

To implement inference, we load the saved model and run it against a live audio feed saved to a ring buffer. Several standard Python packages were used to implement the solution, including the `sounddevice` audio library to access the webcam microphone, the `librosa` and `kapre` packages to generate the mel spectrograms in an efficient manner.

We open an audio stream using `sounddevice` and it uses a callback function to write the sampled audio to a Python `deque` object, which is used to implement a ring buffer by setting a max size argument so that as additional data is enqueued and the queue grows past a certain size, the oldest data drops off. This object was also selected because of its good multithreading support, as the `sounddevice` callback function uses multithreading.

As the queue grows, we pass the full contents of the queue to be turned into a mel spectrogram, we then run inference on the generated image using a simple TensorFlow predict command. Attempting to implement a frozen graph and fully optimize for TensorRT resulted in several compatibility bugs with TensorFlow 2.2. However, we were using an Nvidia built TensorFlow container expressly for the Jetson, and therefore some TensorRT optimization do load when TensorFlow starts as part of the inference code.

9. Conclusion & Future Work

Some challenges that could be addressed in future work include calibrating inference and implementing multi-instance support. The training was naturally highly focused on optimizing against the training data set, however, the conditions necessary for inference may be quite different where perhaps longer samples might be needed. We tried to align basic aspects such as sample rate and audio length but more work is needed to further optimize.

Another challenge would be multi-instance support where more than one bird or species appear in a single sample, this shifts the problem from the simple image classification to what could be considered “object detection” (as we are still in the image domain for training/inference. Finally, the case of a “null species” where a sample has none of the classes needs to be added to the training data to better match real world conditions. With additional time, we would like to conduct deeper research into CVAEs for augmentation of data to confirm whether it would be a viable approach.

There may be significant benefit in automating the extraction of individual bird calls from each recording – similar to extracting faces or other objects from a full video frame before attempting identification. This would both increase the ratio of interesting information compared to silence and background noise in the training set, effectively centering the interesting information in the derived spectrograms and using recordings that included more than one instance of a call to increase the unaugmented data set. This technique could also contribute to both better calibration of inference, as we cut the audio lengths to just the calls themselves, and to multi-instance support – analogous to isolating two objects in an image before identifying them separately as a dog and a cat – by isolating the different calls from the full audio before identification is run.

It should be noted that there are already commercial applications of birdsong identification, for example Wildlife Acoustics (www.wildlifeacoustics.com) has been around for over 10 years, and can identify both bird and bat calls by species. Wildlife Acoustics’ approach however is to use low cost digital recording units in the field and conduct identification on desktop or on the cloud when the recordings are uploaded. Our solution attempts to identify real-time using inference at the edge, avoiding the delays of retrieving recording devices and uploading them.