Model Pipeline Explanations

Regina Arissaputri – 11 December 2024

This document highlights the file optimized_engine_pipeline.ipynb written by akudilczak, which is the main pipeline for generating our echo model.

Initialization

Libraries and Version

The required packages are listed below, as well as what we need them for:

Module	Details	Usage
Python	3.8.16	
warnings		To ignore warnings
os		To get environment details and paths
functools		To create function keys
diskcache		To use diskcache
hashlib		to create a hash of the key for shorter and consistent length
numpy		Basic python function to work with arrays
datetime		For making logs
random		Randomizing items
matplotlib		To create plots
tensorflow	2.10.1	For tensorflow
tensorflow_hub		To access the EfficientNetV2
tensorflow_addons		For tensorflow image augmentation
keras.utils		Dataset_utils for onehot encoding and create dataset from paths
librosa	0.9.2	Import audio and convert to melspectrogram
audiomentations	0.29.0	For audio augmentations

System Configuration

```
SC = {
    'AUDIO_DATA_DIRECTORY': "d:\\data\\b3", # set the location of bucket 3 from GCP
    'CACHE_DIRETORY': "d:\\pipeline_cache", # stores image samples so it does not regenerate images each epic

    'AUDIO_CLIP_DURATION': 5, # seconds
    'AUDIO_NFFT': 2048,
    'AUDIO_WINDOW': None,
    'AUDIO_STRIDE': 200,
    'AUDIO_SAMPLE_RATE': 48000,
    'AUDIO_MELS': 260,
    'AUDIO_FMIN': 20,
    'AUDIO_FMAX': 13000,
    'AUDIO_TOP_DB': 80,

    'MODEL_INPUT_IMAGE_WIDTH': 260,
    'MODEL_INPUT_IMAGE_HEIGHT': 260,
    'MODEL_INPUT_IMAGE_CHANNELS': 3,

    'USE_DISK_CACHE': True, #switch off disc cache here if preferred
    'SAMPLE_VARIANTS': 20,
    'CLASSIFIER_BATCH_SIZE': 16,
    'MAX_EPOCHS': 5000,
}
```

The system configurations shows the constants used in preprocessing the initial training data. Here is the explanation of each of the components:

- 'AUDIO_DATA_DIRECTORY': "d:\\data\\b3"
  This specifies the location of the audio dataset that is located in the local directory. We can access the training data from Google Cloud, and in this case we are using bucket 3.
- 'CACHE_DIRETORY': "d:\\pipeline_cache"
  The location of the preprocessed data cache to speed up training.
- 'AUDIO_CLIP_DURATION': 5
  Specifies the duration of the audio clips that will be used to convert to mel spectrograph in seconds. Clips shorter than this will be padded with silence, and longer clips are segmented.
- 'AUDIO_NFFT': 2048,
  Defines the number of points in FFT (Fast Fourier Transform) to generate the spectrogram. Higher value increases frequency resolution, but requires more computation.
- 'AUDIO_WINDOW': None,
  Window size for FFT, if none, then defaults to same value as AUDIO_NFFT
- 'AUDIO_STRIDE': 200,
  Specifies the step size for moving the FFT window across the audio signal. Smaller strides will increase time resolution but generate more data.
- 'AUDIO_SAMPLE_RATE': 48000,
  Define sample rate of audio data in Hz, if doesn't match, resampling will occur. 16000 matches telephone, 22050 Hz common in music, and 48000 is high quality audio.
- 'AUDIO_MELS': 260,
  The number of mel filter banks used to generate the mel spectrogram to determine the vertical resolution of the spectrogram
- 'AUDIO_FMIN': 20,
  Frequencies below this is ignored, to exclude irrelevant noise.
- 'AUDIO_FMAX': 13000,

Maximum frequency, above is ignored. However, the maximum frequency also depends on Nyquist frequency, which is at most half the sampling rate.

- 'AUDIO_TOP_DB': 80,
  Define the decibel threshold for spectrogram normalization
- 'MODEL_INPUT_IMAGE_WIDTH': 260,
  Width of inpuit image to match the model input
- 'MODEL_INPUT_IMAGE_HEIGHT': 260,
- Height of input image to match the model input
- 'MODEL_INPUT_IMAGE_CHANNELS': 3,
  Specify the number of color channels in input image. 3 indicates RGB
- 'USE_DISK_CACHE': True, #switch off disc cache here if preferred
  If true, enables caching to the specifies directory in CACHE_DIRECTORY
- 'SAMPLE_VARIANTS': 20,
  Define the number of augmented version of each audio sample, for data augmentation to improve model generalization
- 'CLASSIFIER_BATCH_SIZE': 16,
  Specifies batch size for model training
- 'MAX_EPOCHS': 5000,
  Sets the maximum number of epoch for training the model.

Ensure that the training data is nested in this format below:

`path-to-your-new-dataset/dataset/`
Name of Species 1/
audio_file1.mp3 (or '.ogg', '.mp3', '.wav', '.flac')
audio_file2.mp3
audio_file3.mp3
Name of Species 2/
audio_file4.mp3
audio_file5.mp3
...etc

Functions Explanations

Path Dataset

This code loads the file path and labels (converted to one hot labels), convert them to tensorflow datset, and split them into train_ds, val_ds, and test_ds

```
def paths_and_labels_to_dataset(image_paths, labels, num_classes):
    path_ds = tf.data.Dataset.from_tensor_slices(image_paths)
    label_ds = dataset_utils.labels_to_dataset(
        labels,
        'categorical',
        num_classes)
    zipped_path_ds = tf.data.Dataset.zip((path_ds, label_ds))
    return zipped_path_ds
def create_datasets(audio_files, train_split=0.7, val_split=0.2):
```

```
    file_paths, labels, class_names = dataset_utils.index_directory(
            audio_files,
            labels="inferred",
            formats=('.ogg','.mp3','.wav','.flac'),
            class_names=None,
            shuffle=True,
            seed=42,
            follow_links=False)

    dataset = paths_and_labels_to_dataset(
        image_paths=file_paths,
        labels=labels,
        num_classes=len(class_names))

    # Calculate the size of the dataset
    dataset_size = len(dataset)

    # Calculate the number of elements for each dataset split
    train_size = int(train_split * dataset_size)
    val_size = int(val_split * dataset_size)
    test_size = dataset_size - train_size - val_size

    # Split the dataset
    train_ds = dataset.take(train_size)
    val_ds = dataset.skip(train_size).take(val_size)
    test_ds = dataset.skip(train_size + val_size).take(test_size)

    return train_ds, val_ds, test_ds, class_names
```

The function `paths_and_labels_to_dataset()` returns a zipped file dataset that includes the file paths and labels. The function `create_datasets()` uses the dataset_utils function from keras to define the file paths, labels, and class_names from the bucket 3 folder that has the training data. Then it calls the function to get the combined path and label data and split the dataset according to the split specified to generate training, validation, and test dataset. Below is the usage of the function.

```
# Create the dataset
train_ds, val_ds, test_ds, class_names =
create_datasets(SC['AUDIO_DATA_DIRECTORY'],train_split=0.8, val_split=0.19)
print("Class names: ", class_names)
print(f"Training   dataset length: {len(train_ds)}")
print(f"Validation dataset length: {len(val_ds)}")
print(f"Test       dataset length: {len(test_ds)}")
```

Memory Limit

Below function can help limits the GPU memory that is being used, useful for small PC. In this code below, it will limit the memory usage to only 5GB.

```
def enforce_memory_limit(mem_mb):
  # enforce memory limit on GPU

  gpus = tf.config.experimental.list_physical_devices('GPU')
```

```
  if gpus:
    try:
      tf.config.experimental.set_virtual_device_configuration(
          gpus[0],[tf.config.experimental.VirtualDeviceConfiguration(memory_limit=mem_mb)]
)
      print(f"vram limit set to {mem_mb}MB")
    except RuntimeError as e:
      print(e)

# enforce max 5GB memory on GPU for this notebook if you have a small GPU
enforce_memory_limit(5120)
```

Cache Functions

Here is the functions that is related to disk caching. Caching will allow the results to be cached and utilized for the future, instead of calling the functions again, thus speeds up the data preprocessing. You will need a lot of space for caching (around 20GB) for large melspectrograms. The caching works by serializing a function call signature and hashing it into a key, which is used to store the result of the function.

```
if SC['USE_DISK_CACHE']:
    cache = dc.Cache(SC['CACHE_DIRETORY'], cull_limit=0, size_limit=10**9)
```

The function below generates a unique hash key for caching the function output based on its arguments. Function.partial is used for capturing the function arguments, and hashlib is for creating hash

```
#################################################################################################
# a helper function to create a hash key from a function signature and arguments
#################################################################################################
def create_function_key(func, *args, **kwargs):
    partial_func = functools.partial(func, *args, **kwargs)
    func_name = partial_func.func.__name__
    func_module = partial_func.func.__module__
    args_repr = repr(partial_func.args)
    kwargs_repr = repr(sorted(partial_func.keywords.items()))

    key = f"{func_module}.{func_name}:{args_repr}:{kwargs_repr}"
    # Use hashlib to create a hash of the key for shorter and consistent length
    key_hash = hashlib.sha256(key.encode()).hexdigest()

    return key, key_hash, partial_func
```

Mel-Spectrogram Generation Pipeline

Below is the functions explanation in the order of which it is executed.

The function `tensorflow_add_variant_and_cache(path, label)` is used for initializing the variant and cache_key and cache_found. The variant is a randomized integer that acts as a key for augmentation. It returns the sample, label, variant, cache_key, and cache_found.

```
def tensorflow_add_variant_and_cache(path, label):
    variant     = tf.random.uniform(shape=(), minval=0, maxval=SC['SAMPLE_VARIANTS'],
dtype=tf.int32)
    sample      = path
    cache_key   = b'no key'
    cache_found = np.int32(0)
    return sample, label, variant, cache_key, cache_found
```

The function `python_function_wrapper()` is used to integrate Python based functions (like function that uses NumPy or librosa and other libraries outside tensorflow into a TensorFlow workflow. Pipeline_fn is a python function that processes the sample, label, and class_name. out_types is the expected output data of the pipeline_fn to match TensorFlow tf.datatypes (like tf.float32 or tf.int32). The lambda function takes v1,v2,v3,v4,v5 (corresponding to sample, label, variant, cache_key, and cache_found) and call pipeline_fn. The pipeline_fn retuns outputs and the tf.numpy_function ensures the outputs are converted to tensorflow tensores as specified by out_types.

```
def python_function_wrapper(pipeline_fn, out_types, sample, label, variant, cache_key,
cache_found):

    # Use a lambda function to pass two arguments to the function
    sample, label, variant, cache_key, cache_found = tf.numpy_function(
        func=lambda v1,v2,v3,v4,v5: pipeline_fn(v1,v2,v3,v4,v5),
        inp=(sample, label, variant, cache_key, cache_found),
        Tout=out_types)

    return sample, label, variant, cache_key, cache_found
```

This function `python_disk_cache_start()` starts the diskcache. First the function check if the result of the function is already cached, if cache is found, it changes the cache_found to 1 if not, then it stays 0. It also calls the `create_function_key()` function to create unique hash key for caching based on its function and arguments.

```
def python_disk_cache_start(sample, label, variant, cache_key, cache_found):

    cache_key   = b'no key'
    cache_found = np.int32(0)

    if SC['USE_DISK_CACHE']:
        _,cache_key,_ = create_function_key(python_disk_cache_start, sample, label,
variant)
        if cache_key in cache:
            #print(f'found {cache_key} in cache')
            cache_found = np.int32(1)
        else:
            pass
            #print(f'{cache_key} not found in cache')

    return sample, label, variant, cache_key, cache_found
```

The function `python_load_and_decode_file()` loads and decodes the audio file (that was in a form of paths in the dataset before) and loads it using librosa, converts the audio from stereo to mono, and loads it into a NumPy array . But if cache is found (if the cache_found is 1, then it skips all these steps and just load the sample.

```
def python_load_and_decode_file(sample, label, variant, cache_key, cache_found):

    if cache_found == np.int32(0): # Simply load the final result if in cache

        tmp_audio_t = None

        with open(sample, 'rb') as file:

            # Load the audio data with librosa
            tmp_audio_t, _ = librosa.load(file, sr=SC['AUDIO_SAMPLE_RATE'])

            # cast and keep right channel only
            if tmp_audio_t.ndim == 2 and tmp_audio_t.shape[0] == 2:
                tmp_audio_t = tmp_audio_t[1, :]

            # cast and keep right channel only
            tmp_audio_t = tmp_audio_t.astype(np.float32)

            assert(tmp_audio_t is not None)
            assert(isinstance(tmp_audio_t, np.ndarray))

        sample = tmp_audio_t

    else:
        sample = cache[cache_key.decode('utf-8')]

    return sample, label, variant, cache_key, cache_found
```

The function `tensorflow_load_random_subsection()` loads and extract a random subsection, if the audiop duration is more than the system constant set before (AUDIO_CLIP_DURATION). It loads the 1D array of the audio sample and randomly subsect the audio sample if it's longer than the system constant AUDIO_CLIP_DURATION. If it is shorter than the set suration, it will add padding to the audio clip so that it matches the duration of AUDIO_CLIP_DURATION.

```
def tensorflow_load_random_subsection(sample, label, variant, cache_key, cache_found):

    if cache_found == np.int32(0):
        duration_secs = SC['AUDIO_CLIP_DURATION']

        # Determine the audio file's duration in seconds
        audio_duration_secs = tf.shape(sample)[0] / SC['AUDIO_SAMPLE_RATE']

        if audio_duration_secs>duration_secs:

            # Calculate the starting point of the 5-second subsection
            max_start = tf.cast(audio_duration_secs - duration_secs, tf.float32)
            start_time_secs = tf.random.uniform((), 0.0, max_start, dtype=tf.float32)
```

```
            start_index = tf.cast(start_time_secs * SC['AUDIO_SAMPLE_RATE'],
dtype=tf.int32)

            # Load the 5-second subsection
            end_index = tf.cast(start_index + tf.cast(duration_secs, tf.int32) *
SC['AUDIO_SAMPLE_RATE'], tf.int32)

            subsection = sample[start_index : end_index]

        else:
            # Pad the subsection with silence if it's shorter than 5 seconds
            padding_length = duration_secs * SC['AUDIO_SAMPLE_RATE'] - tf.shape(sample)[0]
            padding = tf.zeros([padding_length], dtype=sample.dtype)
            subsection = tf.concat([sample, padding], axis=0)

        sample = subsection

    return sample, label, variant, cache_key, cache_found
```

The function `python_audio_augmentations()` adds audio augmentation using the audiomentation library to the audio sample with a fixed probability of p. The augmentations include adding noise, time stretching, pitch shift, and shifting the audio.

```
# Audio augmentation pipeline
# You can specifiy at what probability should it change i.e. p=0.2

def python_audio_augmentations(sample, label, variant, cache_key, cache_found):

    if cache_found == np.int32(0):
        # See https://github.com/iver56/audiomentations for more options
        augmentations = Compose([
            # Add Gaussian noise with a random amplitude to the audio
            # This can help the model generalize to real-world scenarios where noise is
present
            AddGaussianNoise(min_amplitude=0.001, max_amplitude=0.015, p=0.2),

            # Time-stretch the audio without changing its pitch
            # This can help the model become invariant to small changes in the speed of
the audio
            TimeStretch(min_rate=0.8, max_rate=1.25, p=0.2),

            # Shift the pitch of the audio within a range of semitones
            # This can help the model generalize to variations in pitch that may occur in
real-world scenarios
            PitchShift(min_semitones=-4, max_semitones=4, p=0.2),

            # Shift the audio in time by a random fraction
            # This can help the model become invariant to the position of important
features in the audio
            Shift(min_fraction=-0.5, max_fraction=0.5, p=0.2),
        ])
```

```
        # apply audio augmentation to the clip
        # note: this augmentation is NOT applied in the test and validation pipelines
        sample = augmentations(samples=sample, sample_rate=SC['AUDIO_SAMPLE_RATE'])

    return sample, label, variant, cache_key, cache_found
```

The process then moves on to the function `python_dataset_melspectro_pipeline()` to convert the audio data to melspectrogram using the librosa library. The AUDIO_TOP_DB is also used to normalize the audio to decibel scale/

```
def python_dataset_melspectro_pipeline(sample, label, variant, cache_key, cache_found):

    if cache_found == np.int32(0):
        # Compute the mel-spectrogram
        image = librosa.feature.melspectrogram(
            y=sample,
            sr=SC['AUDIO_SAMPLE_RATE'],
            n_fft=SC['AUDIO_NFFT'],
            hop_length=SC['AUDIO_STRIDE'],
            n_mels=SC['AUDIO_MELS'],
            fmin=SC['AUDIO_FMIN'],
            fmax=SC['AUDIO_FMAX'],
            win_length=SC['AUDIO_WINDOW'])

        # Optionally convert the mel-spectrogram to decibel scale
        image = librosa.power_to_db(
            image,
            top_db=SC['AUDIO_TOP_DB'],
            ref=1.0)

        # Calculate the expected number of samples in a clip
        expected_clip_samples = int(SC['AUDIO_CLIP_DURATION'] * SC['AUDIO_SAMPLE_RATE'] /
SC['AUDIO_STRIDE'])

        # swap axis and clip to expected samples to avoid rounding errors
        image = np.moveaxis(image, 1, 0)
        sample = image[0:expected_clip_samples,:]

    return sample, label, variant, cache_key, cache_found # the sample will be an image
```

The function `tensorflow_reshape_image_pipeline()` is used to reshape the image to match the input for training the model. First the function adds color dimension (by expand_dims()), converting the 2D mel spectrogram (height x width) to 3D tensor (height x width x 1) to add color dimension. Then it repeats the single channel across three channels, to create RGB like representation (height x width x 3), because our pre trained model will expect 3 channels input. Then it enforce the shape of the spectrogram as height = expected_clip_samples (time steps), width = AUDIO_MELS (frequency bins) and depth = MODEL_INPUT_IMAGE_CHANNELS (Color channels). Then it resizes the image to match the input size specified in SC[‘MODEL_INPUT_IMAGE_WIDTH] and

SC[‘MODEL_INPUT_IMAGE_HEIGHT]. Then it contibues to normalize the pixel values to the range of [0,1] and return the processed samples

```
def tensorflow_reshape_image_pipeline(sample, label, variant, cache_key, cache_found):

    if cache_found == np.int32(0):
        # reshape into standard 3 channels to add the color channel
        image = tf.expand_dims(sample, -1)

        # most pre-trained model classifer model expects 3 color channels
        image = tf.repeat(image, SC['MODEL_INPUT_IMAGE_CHANNELS'], axis=2)

        # Calculate the expected number of samples in a clip
        expected_clip_samples = int(SC['AUDIO_CLIP_DURATION'] * SC['AUDIO_SAMPLE_RATE'] /
SC['AUDIO_STRIDE'])

        # calculate the image shape and ensure it is correct
        image = tf.ensure_shape(image, [expected_clip_samples, SC['AUDIO_MELS'],
SC['MODEL_INPUT_IMAGE_CHANNELS']])

        # note here a high quality LANCZOS5 is applied to resize the image to match model
image input size
        image = tf.image.resize(image,
(SC['MODEL_INPUT_IMAGE_WIDTH'],SC['MODEL_INPUT_IMAGE_HEIGHT']),
                                method=tf.image.ResizeMethod.LANCZOS5)

        # rescale to range [0,1]
        image = image - tf.reduce_min(image)
        sample = image / (tf.reduce_max(image)+0.0000001)

    return sample, label, variant, cache_key, cache_found
```

After we have the images and they are reshaped to match the format for training input, the function `tensorflow_image_augmentations()` adds another augmentation to the image to randomly rotate the image between -2 degree and 2 degree.

```
# Image augmentation pipeline
def tensorflow_image_augmentations(sample, label, variant, cache_key, cache_found):

    if cache_found == np.int32(0):
        # random rotation -2 deg to 2 deg
        degrees = tf.random.uniform(shape=(1,), minval=-2, maxval=2)

        # convert the angle in degree to radians
        radians = degrees * 0.017453292519943295

        # rotate the image
        sample = tfa.image.rotate(sample, radians, interpolation='bilinear')

    return sample, label, variant, cache_key, cache_found
```

Lastly the function `python_disk_cache_end()` ends the disk cache by saving the result to the cache if it wasn't found initially. If there is cache found already, then it just returns the samples.

```
def python_disk_cache_end(sample, label, variant, cache_key, cache_found):
    cache_key = cache_key.decode('utf-8')
    if SC['USE_DISK_CACHE']:
        # if it was not found in the cache at the start, then populate with what we built
        # during the pipeline execution
        if cache_found == np.int32(0):
            #print(f'adding {cache_key} to cache')
            cache[cache_key] = sample
        #else:
        #    sample = cache[cache_key]

    return sample, label, variant, cache_key, cache_found
```

The last function `tensorflow_output_shape_setter()` is to set the shape of the tensors in a TensorFlow data pipeline. TensorFlow operations often require tensors with known fixed shapes. This function basically re enforce the shapes of the tensors to make sure it is in the correct shape.

```
def tensorflow_output_shape_setter(sample, label, variant, cache_key, cache_found):
    sample.set_shape([SC['MODEL_INPUT_IMAGE_WIDTH'], SC['MODEL_INPUT_IMAGE_HEIGHT'],
SC['MODEL_INPUT_IMAGE_CHANNELS']])
    label.set_shape([len(class_names),])
    return sample, label, variant, cache_key, cache_found
```

The last function `tensorflow_drop_variant_and_cache()` just drops the other variables so that the pipeline only returns the sample and the label

```
def tensorflow_drop_variant_and_cache(sample, label, variant, cache_key, cache_found):
    return sample, label
```

The pipeline of the data preprocessing is shown below:

```
###########################################################################################
# Create the datasets necessary for training a classification model
# Note: python and tensorflow functions are treated differently in the tensorflow
# pipeline.  Each python function needs to be wrapped.
# this is why each pipeline function starts with python_ or tensorflow_ to make it clear
###########################################################################################

# Get the length of the training dataset
len_train_ds = len(train_ds)
parallel_calls = tf.data.AUTOTUNE
cache_output_types = (tf.string,tf.float32,tf.int32,tf.string,tf.int32)
procs_output_types = (tf.float32,tf.float32,tf.int32,tf.string,tf.int32)

# Create the training dataset pipeline
train_dataset = (train_ds
                 # Shuffles all of the file names
                 .shuffle(len_train_ds)
                 # Adds variant and cache, loads file etc
```

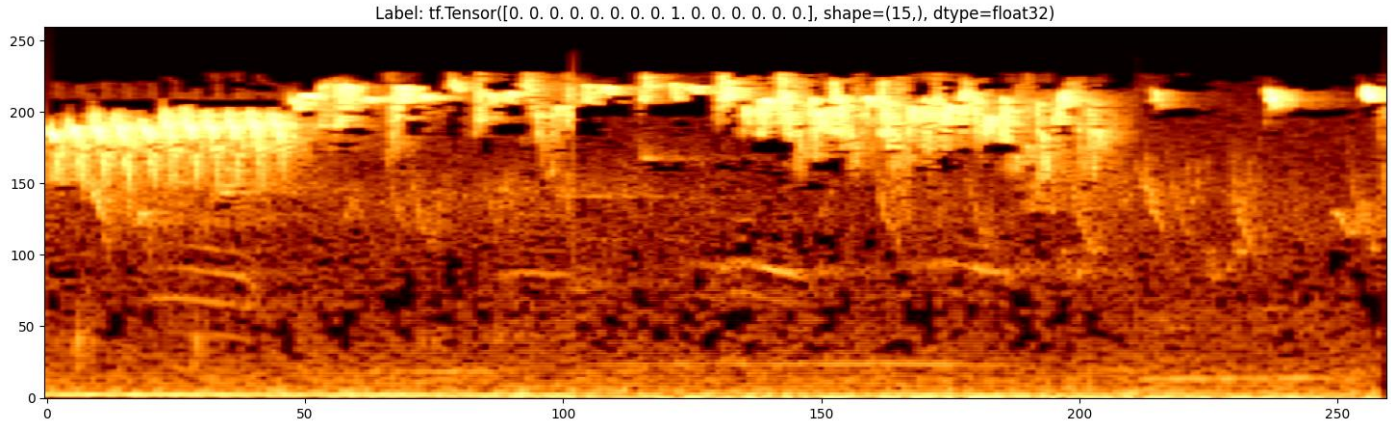
```
                .map(tensorflow_add_variant_and_cache, num_parallel_calls=parallel_calls)
                .map(functools.partial(python_function_wrapper, python_disk_cache_start,
cache_output_types), num_parallel_calls=parallel_calls)
                .map(functools.partial(python_function_wrapper,
python_load_and_decode_file, procs_output_types), num_parallel_calls=parallel_calls)
                # Completes random subsection
                .map(tensorflow_load_random_subsection,
num_parallel_calls=parallel_calls)
                # Completes audio augmentations
                .map(functools.partial(python_function_wrapper,
python_audio_augmentations, procs_output_types), num_parallel_calls=parallel_calls)
                # Converts to mel spectrogram
                .map(functools.partial(python_function_wrapper,
python_dataset_melspectro_pipeline, procs_output_types),
num_parallel_calls=parallel_calls)
                # Reshape so we can tell pipeline what shape the images are
                .map(tensorflow_reshape_image_pipeline,
num_parallel_calls=parallel_calls)
                # Complete image augmentation
                .map(tensorflow_image_augmentations, num_parallel_calls=parallel_calls)
                # Store in cache if new
                .map(functools.partial(python_function_wrapper, python_disk_cache_end,
procs_output_types), num_parallel_calls=parallel_calls)
                # What is the output shape
                .map(tensorflow_output_shape_setter, num_parallel_calls=parallel_calls)
                # Drop variant and cache information as it's not required anymore
                .map(tensorflow_drop_variant_and_cache,
num_parallel_calls=parallel_calls)
                # Ready to pass into the model
                .batch(SC['CLASSIFIER_BATCH_SIZE'])
                # Start preparing the next set of data
                .prefetch(parallel_calls)
                .repeat(count=1)
)
# Create the validation dataset pipeline
validation_dataset = (val_ds
                    .map(tensorflow_add_variant_and_cache,
num_parallel_calls=parallel_calls)
                    .map(functools.partial(python_function_wrapper,
python_disk_cache_start, cache_output_types), num_parallel_calls=parallel_calls)
                    .map(functools.partial(python_function_wrapper,
python_load_and_decode_file, procs_output_types), num_parallel_calls=parallel_calls)
                    .map(tensorflow_load_random_subsection,
num_parallel_calls=parallel_calls)
                    .map(functools.partial(python_function_wrapper,
python_dataset_melspectro_pipeline, procs_output_types),
num_parallel_calls=parallel_calls)
                    .map(tensorflow_reshape_image_pipeline,
num_parallel_calls=parallel_calls)
                    .map(tensorflow_output_shape_setter,
num_parallel_calls=parallel_calls)
                    .map(functools.partial(python_function_wrapper, python_disk_cache_end,
procs_output_types), num_parallel_calls=parallel_calls)
```

```
                    .map(tensorflow_output_shape_setter,
num_parallel_calls=parallel_calls)
                    .map(tensorflow_drop_variant_and_cache,
num_parallel_calls=parallel_calls)
                    .batch(SC['CLASSIFIER_BATCH_SIZE'])
                    .prefetch(parallel_calls)
                    .repeat(count=1)
)
# Create the test dataset pipeline
test_dataset = (test_ds
                .map(tensorflow_add_variant_and_cache, num_parallel_calls=parallel_calls)
                .map(functools.partial(python_function_wrapper, python_disk_cache_start,
cache_output_types), num_parallel_calls=parallel_calls)
                .map(functools.partial(python_function_wrapper,
python_load_and_decode_file, procs_output_types), num_parallel_calls=parallel_calls)
                .map(tensorflow_load_random_subsection,
num_parallel_calls=parallel_calls)
                .map(functools.partial(python_function_wrapper,
python_dataset_melspectro_pipeline, procs_output_types),
num_parallel_calls=parallel_calls)
                .map(tensorflow_reshape_image_pipeline,
num_parallel_calls=parallel_calls)
                .map(tensorflow_output_shape_setter, num_parallel_calls=parallel_calls)
                .map(functools.partial(python_function_wrapper, python_disk_cache_end,
procs_output_types), num_parallel_calls=parallel_calls)
                .map(tensorflow_output_shape_setter, num_parallel_calls=parallel_calls)
                .map(tensorflow_drop_variant_and_cache,
num_parallel_calls=parallel_calls)
                .batch(SC['CLASSIFIER_BATCH_SIZE'])
                .prefetch(parallel_calls)
                .repeat(count=1)
)
```

To inspect the current pipeline and show the mel spectrogram images of the first batch, we can run the code below

```
for melspectrogram,label in train_dataset.take(1):
    print(f' sample info: {melspectrogram.shape}, \n label info: {label.shape} \n
{label}')
    for example in range(melspectrogram.shape[0]):
        plt.figure(figsize=(18,5))
        plt.imshow(melspectrogram[example,:,:,0].numpy().T, cmap='afmhot', origin='lower',
aspect='auto')
        plt.title('Label: ' + str(label[example,:]))
        plt.show()
```

Here is the example result of the Mel spectrogram image:



Here the shape is 15 because I was only using a test data that consists of 15 animal species.

If the pipeline manages to produce similar mel spectrograph images, then we can continue the process to build the model.

The final shape of the dataset will be (16, 260, 260, 3) where:

- Batch size: 16, 16 samples are processed together in each batch
- Height: 260, second dimension for image height
- Width: 260, third dimension for image width
- Channels: 3, the fourth dimension indicating 3 channels, thus shows RGB color channel. Th emel spectrographs are replicated across three channels to match the input requirements of many pretrained models.

Classification Model

Thus once we have produced the Mel spectrograph images, we can continue to the CNN based image classification construction. The model leverages pre-trained model weights for the EfficientNetV2 feature model which generates a vector representation of 1000 floats for each input image. The output of the EfficientNetV2 model is then fed into 2 fully connected (dense) layers to perform the classification.

```
def build_model(trainable):
    # Build a classification model using a pre-trained EfficientNetV2
    model = tf.keras.Sequential(
        [
            # Input layer with specified image dimensions
            tf.keras.layers.InputLayer(input_shape=(SC['MODEL_INPUT_IMAGE_HEIGHT'],
                                                    SC['MODEL_INPUT_IMAGE_WIDTH'],
                                                    SC['MODEL_INPUT_IMAGE_CHANNELS'])),

            # Use the EfficientNetV2 model as a feature generator (needs 260x260x3 images)
            hub.KerasLayer("https://tfhub.dev/google/imagenet/efficientnet_v2_imagenet1k_b
0/classification/2", trainable),

            # Add the classification layers
            tf.keras.layers.Flatten(),
            tf.keras.layers.BatchNormalization(),
```

```
            # Fully connected layer with multiple of the number of classes
            tf.keras.layers.Dense(len(class_names) * 8,
                                  activation="relu"),
            tf.keras.layers.BatchNormalization(),

            # Another fully connected layer with multiple of the number of classes
            tf.keras.layers.Dense(len(class_names) * 4,
                                  activation="relu"),
            tf.keras.layers.BatchNormalization(),

            # Add dropout to reduce overfitting
            tf.keras.layers.Dropout(0.50),

            # Output layer with one node per class, without activation
            tf.keras.layers.Dense(len(class_names), activation=None),
        ]
    )
    # Set the input shape for the model
    model.build([None,
                 SC['MODEL_INPUT_IMAGE_HEIGHT'],
                 SC['MODEL_INPUT_IMAGE_WIDTH'],
                 SC['MODEL_INPUT_IMAGE_CHANNELS']])

    # Display the model summary
    model.summary()

    return model
```

The model uses EfficientNetV2 from TensorFlow Hub as a feature extractor. The model transforms the input image into a 1000 dimensional feature vector as output. The model was trained on ImageNet thus it can recognize various features from images. Because trainable=True, the weights of this layer can be fine-tuned.

Then it flatten the 1000 dimensional feature vector into 1D array so that it is compatible with dense layers.

The dense layer is a fully connected layer with len(class_names)*8 neurons and ReLu activation function for non-linearity.

Then it goes through another dense layer with len(class_names)*4 neurons.

The last output layer is a dense layer with len(class_names) neurons (one for each class), it outputs a 1D array of size len(class_names) to represent logits for each class.

Model Training

In the model training has callbacks to track model training and performs the model fit. The callbacks also include checkpoints, to ensure the best model weights (lowest val_loss) is written to disk during training. The model stop training when there is no further improvement to the val_loss. Hopefully, the training will catch itself when it starts to overfit and stop further training.

```
if not os.path.exists('models/'):
```

```
    os.mkdir('models/')
if not os.path.exists('models/1'):
    os.mkdir('models/1')

# allow all the weights to be trained
model = build_model(True)

# the form_logits means the loss function has the 'softmax' buillt in.  This approach is
numerically more stable
# than including the softmax activation on the last layer of the classifier
model.compile(loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
              optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4),
              metrics=["accuracy"],
              )

# tensorboard for visualisation of results
log_dir = "tensorboard_logs/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir,
                                                      histogram_freq=1)

# reduce learning rate to avoid overshooting local minima
lr_reduce_plateau = tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss',
                                                      factor=0.75,
                                                      patience=8,
                                                      verbose=1,
                                                      mode='min',
                                                      cooldown=0,
                                                      min_lr=1e-7)

# end the training if no improvement for 16 epochs in a row, then restore best model
weights
early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor="val_loss",
    min_delta=0,
    patience=16,
    verbose=0,
    mode="min",
    baseline=None,
    restore_best_weights=True,
)

# save the best model as it trains..
mcp_save = tf.keras.callbacks.ModelCheckpoint('models/checkpoint_generic_model.hdf5',
                                           save_best_only=True,
                                           monitor='val_loss',
                                           mode='min')

# any changes to the source code will generally require the disk cache to be cleared.
# So to be safe, the cache is cleared before training the model.  If you are sure
# the cache is still valid then comment out this code
# the first few epochs of the model training will be slow as the cache is populated with
pipeline samples
# and will depend on the dataset size and the number of variants included
cache.clear()
```

```
# fit the model to the training set
# this may take 12-24 hours to run to full model convergence depending on your machine
model.fit(train_dataset,
          validation_data=validation_dataset,
          callbacks=[lr_reduce_plateau, early_stopping, tensorboard_callback, mcp_save],
          epochs=SC['MAX_EPOCHS'])
```

Model Prediction

Then we can use the model to make predictions such as the function below.

The function converts the raw outputs (logits) from the final layer of the model and returns a predicted species name as string and the predicted probability.

It does this by first squeezing (tf.squeeze) the predictions into dimension of size 1 from the tensor and also identify which index has the highest probability, and save it as predicted_index.

Then it continues to map the index to the class_name and fetch the species name, it saves the string as predicted_class.

It also computes the predicted_probability by applying softmax function to the logits to convert them into probabilities and change it into percentage format.

The function then returns the predicted_class and predicted_probability

```
# Function to predict class and probability given a prediction
def predict_class(predictions):
    # Get the index of the class with the highest predicted probability
    predicted_index = int(tf.argmax(tf.squeeze(predictions)).numpy())
    # Get the class name using the predicted index
    predicted_class = class_names[predicted_index]
    # Calculate the predicted probability for the selected class
    predicted_probability = 100.0 * tf.nn.softmax(predictions)[predicted_index].numpy()
    # Round the probability to 2 decimal places
    predicted_probability = str(round(predicted_probability, 2))
    return predicted_class, predicted_probability
```

Then the function is used below, where it iterate over each features and labels in test_dataset and predict it in the model using the outer most layer.

```
# Display class names and run prediction on test entries
print(f'Class names: {class_names}')
for features, labels in test_dataset:
    # Generate predictions for the given features
    predictions = model.predict(features, verbose=0)

    # Iterate over each item in the batch
    for batch_idx in range(predictions.shape[0]):
        # Get the index of the true class
        true_index = int(tf.argmax(tf.squeeze(labels[batch_idx])).numpy())
        # Get the true class name using the true index
        true_class = class_names[true_index]
```

```
    # Predict class and probability using the prediction function
    predicted_class, predicted_probability = predict_class(predictions[batch_idx])

    print(f'True class      : {true_class}')
    print(f'Predicted class : {predicted_class}')
    print(f'Predicted probability : {predicted_probability}')
```

Overview of Workflow

At a glance, here is everything that the pipeline in the notebook does: