FEW-SHOT BIOACOUSTIC EVENT DETECTION VIA SEGMENTATION USING PROTOTYPICAL NETWORKS

Technical Report

Jens Johannsmeier, Sebastian Stober

Otto-von-Guericke-Universität
Faculty of Computer Science
Universitätsplatz 2, 39106 Magdeburg, Germany

ABSTRACT

This report describes our submission to task 5 of the 2021 DCASE challenge. We detail how we processed the data, the model structure as well as the training procedure. We may submit an extended version to the DCASE 2021 workshop.

1. OVERVIEW

For this technical report, we assume that the reader is familiar with the provided Deep Learning baseline[1]. We process the data as Mel spectrograms at a time resolution of roughly 6ms. On this data, we perform segmentation: Each frame is classified as either containing an event or not. Any switch from 0 to 1 is extracted as an event start; any switch from 1 to 0 as an event end. Classification is done using Prototypical networks [1], where a *prototype frame* is computed per class (positive/negative) and a given query frame is assigned whichever prototype is closest as measured by euclidean distance. Prototypes are computed using CNNs. Our implementation is in Python 3, using Tensorflow 2.3 [2], and can be found on GitHub[2]. In the rest of this report, we provide details on all these steps.

2. DATA

2.1. Data used

We only use the provided challenge data, consisting of 11 recordings with a total duration of 14 hours and 20 minutes for training and 8 recordings of 5 hours for validation. The former is used in its entirety for training; the latter is used for model selection exclusively.

2.2. Feature extraction

As a first step, we resample all data to 20,050 Hz. Then we compute Mel spectrograms with a window size of 1024 samples (about 46 ms) and a hop size of 128 samples (about 6 ms or 172 frames per second). We use 130 mel frequency bins. Afterwards we perform PCEN [3]; however only the first part (low-pass filtering) is performed at this point; the rest is part of the model (see subsection 3.2). We use a time constant of 200 ms. All these steps are performed using librosa [4].

[1]`https://github.com/c4dm/dcase-few-shot-bioacoustic/tree/main/baselines/deep_learning`

[2]`https://github.com/xdurch0/DCASE2021-Task5`

2.3. Event extraction

For the training data, we use the provided annotations (converting from seconds to frames) to extract the events for each class (species). Each event is extracted with a one-frame margin both at the beginning and the end to prevent events from being cut off. We also create a binary mask for each event which is 1 in the annotation window and 0 outside. We ignore events labeled UNK.

Furthermore, we extract the *entire* dataset as one large "negative" event with an all-zero mask. This is done to use more of the training data (since only a small fraction is actually covered by events) as well as to provide more examples of what should be classified as zero. Doing this will obviously include actual events in the negative data, but this is similar to how the model is used on the evaluation data (see subsection 3.4). While we presume this does not have a significant negative impact on training, we did not test this. Alternatively, it would be possible to use the provided annotations to exclude all labeled event data from the negative set.

In principle, our model can handle events of arbitrary length, as it is fully convolutional. Still, due to computational considerations we split events into segments of about 200 ms (34 frames) with a hop size of about 100 ms (17 frames) in-between. For events shorter than 34 frames, we simply include the audio after the event to get a 34-frame segment. Note that this differs from the "tiling" method used in the baseline.

For each recoding in the validation/evaluation set, we proceed in a similar manner: The five support events are extracted according to the annotations (potentially resulting in more segments for long events). For the negative class, we consider the full recording. For the query set, we consider the full recording starting at the end of the fifth annotated event.

3. MODEL

3.1. Framework

Our approach is based on Prototypical Networks [1]. Given k possible classes and a support set S_k of labeled examples per class, a *query sample* x is classified as follows:

1. Embed all support samples s_{ki} as well as x using a (learned) function f, such as a neural network.
2. For each class, compute the prototype as the average of all support embeddings for that class: $P_k = \frac{1}{n_k}\sum_i f(s_{ki})$.
3. Compute the distance $d(f(x), P_k)$ between the query embedding and all prototypes using some distance function d.

4. The predicted class is the one with the lowest distance to the respective prototype.

In our case:

1. f is a convolutional neural network that embeds an audio *segment*, however the embedding has the same temporal resolution as the input.
2. Prototypes are then computed using *masked average pooling*: The segment-level embedding is multiplied by the binary annotation mask, zeroing-out components corresponding to time steps that are not marked as an event of the relevant class. Then the embedding is summed over time and divided by the number of non-zero time steps. Thus, prototypes are averaged not only over multiple segments, but over frames as well, resulting in *prototype frames*.
3. For d, we use euclidean distance. This differs from the original paper on ProtoNets, which uses the *squared* euclidean distance and theoretically justifies this choice. However, we observed rather unstable training and got bad results using this distance function, which was alleviated by using euclidean distance.
4. Finally, *each frame* of the query embedding is classified independently, according to which prototype frame is closest.

3.2. Architecture

We split our architecture into three parts: Preprocessing, body, and distance function. For preprocessing, we apply the remaining PCEN steps of dynamic range compression. Each of the parameters α, δ, r and ϵ is learned, with a separate value for each of the 130 input channels, using the default librosa values as initialization. However, we should note at this point that we barely see these values change during training. It could be that the parameterization in log space (like in the librosa implementation) leads to bad gradients. After PCEN, we apply batch normalization [5] to the spectrogram "image". We also apply some light data augmentation: Inputs are randomly cropped from (34, 130) to (32, 128). Cropping in the frequency axis could be interpreted as pitch shifting the spectrogram. We also apply zoom on the time axis, zooming between 20% in and 20% out. This can be interpreted as time-stretching the signal. All these augmentations are very light, but changing the inputs too much could easily destroy the identity of the target species, making augmentation counter-productive.

For the support set, we apply augmentations in a slightly different fashion: *All* possible crops in the time axis are applied, resulting in the support set increasing three-fold in number of segments. It would be possible to do the same for cropping in the frequency axis, as well as taking several different zooms, to further increase the size of the support set. We did not do this simply because these augmentations were added later, and we did not have time to add this functionality. During inference, cropping is always performed to the center (except for cropping time in the support set, where once again all possibilities are enumerated) and no zooming is done.

For the body, we treat the input spectrograms as two-dimensional one-channel images and apply a stack of two-dimensional convolutions as our model. An overview of the architecture is given in Table 1. All convolutions use a filter size of 3×3. Each convolution is followed by batch normalization and a SiLU/Swish activation [6, 7, 8]. We also apply light dropout [9] (Spatial dropout with $p_{drop} = 0.05$) and L2 regularization (factor

Layer	# Filters	Time pooling	Frequency pooling
Encoder 1	32	2	2
Encoder 2	64	2	2
Decoder 1	128	0.5	1
Decoder 2	256	0.5	1

Table 1: Architecture summary. A pooling factor of less than 1 indicates upsampling. The partition into encoder and decoder layers is somewhat arbitrary and only based on whether a layer applies up- or downsampling.

1e-6) For upsampling, we use nearest-neighbor interpolation. There are no residual or skip connections; the model is a simple sequence of layers. Overall, the model has approximately 390,000 parameters.

The distance function is simply euclidean distance, as mentioned before. In principle, this part could easily be switched out for some other function, including a learned one (e.g. see [10]). However, we got the best and most stable results using euclidean distance. Note that this function is applied to the flattened embeddings, which thus have a dimensionality of $n_{freqs} \cdot n_{channels}$, i.e. the "height" of the final feature maps times the number of convolution channels of that layer. With our architecture, this means the embeddings are $32 \cdot 256 = 8192$-dimensional. This seems excessively large; however, our attempts to re-design the architecture to produce smaller embeddings resulted in worse performance.

3.3. Training

The model is trained using "episodes" (refer to [1] for details). For each episode, we randomly choose four out of the 19 classes. We also *always* draw examples from the negative class. Per class we use five support samples and a single query sample (all randomly chosen). Since the training data is highly imbalanced, we oversample by endlessly repeating the data for each class: This way, a class that has n samples would simply have all its data used twice in the time a class with $2n$ samples would be used once.

Within an episode, we use a standard cross-entropy loss. Targets are one-hot vectors for the respective class, whereas output probabilities are computed via softmax using the negative distances between queries and prototypes as logits. We actually only compute the loss based on the center half of the query frames, since the first and last quarter are discarded during inference anyway (but see subsection 3.4). Since there is strictly speaking no "center half" for 34 frames, we discard 9 frames in the beginning and 8 at the end (or 8 and 7 after random cropping).

For the optimizer, we use Adam with default parameters (including a learning rate of 0.001). The learning rate is divided by two whenever the average per-epoch loss does not decrease for three epochs, where we arbitrarily define an epoch as 100 training steps/episodes. Training is stopped if no improvement is seen for nine epochs (allowing for two reductions in learning rate before terminating), and the best-performing model is restored. In practice, models only train for around 50 epochs or so, corresponding to 5,000 training iterations, making training extremely fast (only a few minutes – 100 steps only take around five seconds). We also tried training for longer, forcing 50,000 steps with a cosine learning rate decay, but found that this gives worse results. We use a single NVIDIA GeForce 1080Ti for training,

3.4. Inference

Given a recording to annotate, we first embed the support segments and compute the prototype as before. To get a "negative" prototype, we randomly sample 650 segments (we simply left this number as in the baseline) from the recording. Assuming that the event density is low, not too many actual events should be included in the negative prototype, however this assumption may of course be violated. As such, this process deserves further thought.

Given the prototypes, we can extract event probabilities for the query set. This is once again done via softmax based on negative euclidean distance. Recall that we classify individual frames, but the model actually receives 200 ms segments to process at once, and consecutive segments have 50% overlap. Thus, many frames are actually classified twice, and we have to be careful when stitching segments together to get exactly one output for each frame of the query set. We achieve this by only keeping the center 100 ms of each segment and discarding the rest. Other methods are possible, such as averaging results for those frames where we have multiple outputs.

This entire process is repeated 10 times, sampling a separate negative set each time, and the resulting probabilities are averaged to reduce variance. Finally, we apply a Gaussian filter to the probabilities to reduce their "jaggedness". Here, we use 0.05 times the average length of the support events (for the given recording only) as the standard deviation.

Given the event probabilities, we choose a threshold in (0, 1) and binarize the probabilities accordingly. Given this binary representation, all changes $0 \rightarrow 1$ are regarded as a detected event start, and all changes $1 \rightarrow 0$ as an event end. We convert frame numbers back to seconds and store the results. Afterwards, we apply postprocessing similar to the baseline code, discarding all events that are shorter than 60% of the shortest support event. This massively improves the validation results – without postprocessing, our models only achieve around 15% F1-score.

To choose a threshold, we simply try values in the range $[0.1, 0.9]$ in increments of 0.01 and choose the one that results in the best F1-score on the validation set as returned by the provided evaluation script (i.e. macro-averaged over recordings). This is usually somewhere between 0.6 and 0.8 (our submissions all use thresholds very close to 0.7).

Three of our four submissions are the exact same procedure as described, repeated with different seeds. We do not select for good seeds; we simply train three models and use their results. The fourth model is an ensemble of the previous three, where we simply average their per-frame probabilities before thresholding. This tends to result in slightly better precision, although we did not find significant improvements over single models on the validation data in terms of overall performance. There are of course other ways to create ensembles; for example, one could first choose the optimal threshold for each model and then summarize their post-threshold decisions via majority vote.

Our models achieve approximately 60% macro-averaged F1-score on the validation set. For results on the evaluation set, please see the challenge website.

4. POSSIBLE IMPROVEMENTS

In this section, we would like to highlight parts of our procedure that we believe warrant further work. We did try several things ourselves, but usually found no effect or a decrease in performance.

First off, the model is very small and trains for comparatively few steps. This is if course not a bad thing per se, but given the trend towards ever larger models, it would be surprising if there were no more performance to be gained here by using much larger/deeper models. However, given the fact that we already observed overfitting in our models, this would probably need to be supported by larger datasets and/or significant data augmentation.

Second and perhaps somewhat related, the embeddings are almost comically large. It may be that the shallow network is unable to meaningfully compress the data into fewer dimensions. We believe that smaller embeddings are desirable since distance computations tend to be unreliable in high dimensions.

Regarding the topic of distance, we were puzzled by the fact that squared euclidean distance performed so much worse than euclidean distance. This was consistent over a wide selection of learning rates (we tested several orders of magnitude). It would be expected that the euclidean distance produces more problematic gradients due to the presence of the square root. Also, the ProtoNet paper presents a theoretical justification for using the squared distance, whereas we are not aware of any reason why euclidean distance should be a "good" choice. We also tried using cosine distance, but this lead to very bad results. Another option could be to use a learned distance function, similar to RelationNet [10]. We also tested this, but observed a drop in validation performance, presumably due to overfitting (since training accuracy usually increased).

Furthermore, the inference procedure is quite simplistic and could be improved. For example:

- A global threshold applied to all recordings is probably not ideal. Given that we are dealing with very different species of animals, different thresholds may be appropriate for each recording. Even within a recording, the optimal threshold may vary over time. We experimented with a leave-one-out procedure where we classified one support segment based on a prototype computed from all other segments and used the resulting (averaged) probabilities to scale the threshold for the respective recording, but did not see an improvement.
- We found that our model has a tendency to return a positive detection in case of just "any" event, e.g. noises that are simply loud. This could mean that the prototypes are not discriminative enough (again pointing to issues with the model). To improve the negative prototype, we proceeded as follows: Assuming the annotations are high-quality, we know that everything before the fifth support annotation is negative, as long as it does not fall in any of the previous support annotations. During inference, we can compute the model probabilities for these known-negative frames. Any frames that receive a high positive probability can be added to the negative prototype; this should presumably lead to a more negative output for this frame in the future. We iterate this process, updating the negative prototype each time, until there are no known-negative frames receiving a probability above a certain threshold. Unfortunately, we found that this does not terminate and there are always known-negative frames that receive high positive probability. As such, we did not test how this procedure influences performance. Still, we believe this idea (or something similar) deserves further attention.
- We hypothesize that having a single prototype per class is overly restrictive, particularly because we are dealing with single frames. We tried to cluster the embeddings using k-means, experimenting both with two and four clusters per class. Posi-

tive and negative embeddings were clustered separately. Then, we averaged the per-cluster embeddings to receive the respective prototype. Hypothetically, this should allow for better representation of intra-class (or intra-event) variability. To return to a binary output, we tried two alternatives: First, we simply sum the probabilities for all positive and all negative prototypes, respectively. Or, second, we take the largest positive probability as well as the largest negative one and renormalize them to add to 1. Unfortunately, none of our experiments lead to improvements. Note that we did not train new models for this procedure – it may be better to *train* models with multiple prototypes per class, closer to Matching Networks [11].

Finally, we did not have the opportunity to properly treat the preprocessing part of the model. Mel spectrograms may not be optimal representations since they are tuned for human perception, while we are dealing with animal calls. Also, the PCEN was not tuned properly, with a single fixed time constant and learnable parameters that barely changed from their initial values. We experimented with using multiple different (fixed) time constants and concatenating the resulting PCEN representations into a multi-channel input, but this did not improve results. We would have liked to investigate different representations such as LEAF [12], but did not manage to do so in time.

5. CONCLUSION

We have presented a segmentation-based approach to few-shot bioacoustic event detection. While we expected our model to perform better, we believe it is superior to the "crude" approach that was provided as a baseline, where whole segments of audio were assigned a single label. This is fundamentally limited in its temporal resolution by the hop size between the segments. For example, at the baseline hop of 50 milliseconds, events can only be recognized at 100 milliseconds apart since there needs to be an "off" classification in-between to separate both events. This could be "fixed" by reducing the segment hop size, but this would in turn require the model to make different decisions for almost-identical segments (only shifted by a few ms or so), which is doomed to fail using methods like CNNs.

On the contrary, our approach is limited only by the resolution of the input features. This makes it more appropriate in scenarios where precise annotations are important. Still, it needs further work to reduce the noisiness of the outputs (essentially the opposite problem of the too-low resolution of the baseline) as well as improve the per-frame prototypes.

All in all, we are unsure whether it is reasonable to expect a single model to perform well both for short events such as bird calls and long ones such as hyenas or other mammals, since these have opposing requirements: Short, frequent events require precise localization, whereas longer events need robustness to noise and fluctuations. A better approach could be to design multiple systems for different time scales and use application-dependent prior knowledge to select the most appropriate one. However, this is not really possible for a challenge like this, where the data is unknown. Perhaps such a choice could be made heuristically based on the lengths (or other properties) of the support events. We are looking forward towards the creative solution of the other participants.

6. REFERENCES

[1] J. Snell, K. Swersky, and R. S. Zemel, "Prototypical networks for few-shot learning," *arXiv preprint arXiv:1703.05175*, 2017.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[3] Y. Wang, P. Getreuer, T. Hughes, R. F. Lyon, and R. A. Saurous, "Trainable frontend for robust and far-field keyword spotting," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2017, pp. 5670–5674.

[4] B. McFee, C. Raffel, D. Liang, D. P. Ellis, M. McVicar, E. Battenberg, and O. Nieto, "librosa: Audio and music signal analysis in python," in *Proceedings of the 14th python in science conference*, vol. 8. Citeseer, 2015, pp. 18–25.

[5] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*. PMLR, 2015, pp. 448–456.

[6] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *arXiv preprint arXiv:1710.05941*, 2017.

[7] S. Elfwing, E. Uchibe, and K. Doya, "Sigmoid-weighted linear units for neural network function approximation in reinforcement learning," *Neural Networks*, vol. 107, pp. 3–11, 2018.

[8] D. Hendrycks and K. Gimpel, "Gaussian error linear units (gelus)," *arXiv preprint arXiv:1606.08415*, 2016.

[9] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[10] F. Sung, Y. Yang, L. Zhang, T. Xiang, P. H. Torr, and T. M. Hospedales, "Learning to compare: Relation network for few-shot learning," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 1199–1208.

[11] O. Vinyals, C. Blundell, T. Lillicrap, K. Kavukcuoglu, and D. Wierstra, "Matching networks for one shot learning," *arXiv preprint arXiv:1606.04080*, 2016.

[12] N. Zeghidour, O. Teboul, F. d. C. Quitry, and M. Tagliasacchi, "Leaf: A learnable frontend for audio classification," *arXiv preprint arXiv:2101.08596*, 2021.