UNIVERSITÀ DI PISA

DEPARTMENT OF COMPUTER SCIENCE

MSc IN DATA SCIENCE AND BUSINESS INFORMATICS

Explaining Siamese Networks in Few-Shot Learning for Audio Data

TECHNICAL REPORT

Introduction

This technical report include excerpts from Chapter 5 and 6 of the MSc thesis which is publicly available[1]. Here we only report the technical details useful to understand how the networks were trained, evaluated and tested.

[1] https://etd.adm.unipi.it/t/etd-04062022-145234/

Contents

1 C-way one-shot Learning

Similarly to the process followed by Koch, Zemel, Salakhutdinov, et al. 2015 and Honka 2019, we approached *one-shot* learning in two main phases: first we ***train*** the Siamese Network to learn pairs similarity on *training classes*, while ***validating*** it by monitoring the mean accuracy of different random C-way one-shot tasks on *validating classes*. Then, we ***test*** the network performance in terms of mean accuracy of several random C-way one-shot tasks on *test classes*. The learning-validating process is repeated until model convergence is reached, or no accuracy improvements are observed for a certain amount of validating iterations. Training, validation and test sets are disjoint, so to measure the model ability to generalise on unseen class distributions. Before going into the details of each step, let us illustrate a complete overview of this approach with Algorithm 1.

Algorithm 1 Overview of the general framework

repeat
    X, y ← *GetTrainBatch()*
    m ← *TrainOnBatch(m, X, y)*
    if *current run involves evaluation* **then**
        accuracy ← *EvaluateOneShot(m, validation classes)*
        save model weights if accuracy has improved
until *model* convergence or *no validation accuracy improvement recorded*
EvaluateOneShot(m, test classes)

The training-validating step is included in a loop cycle (lines 1-7, Algo 1) and the training stops if the model converge or if no validation accuracy improvement has been recorded in a given number of such loop iteration runs. The very first step is the training phase, where the model is trained on learning semantic similarities and dissimilarities between input pairs (line 3, Algo.1). Line 3 has to be intended as a preliminary phase where the model gets fit to training instances, to then being queried on a query sample and a support set to predict the class scoring the highest similarity value based on the previous training phase. The f function is encapsulated in the *EvaluateOneShot* function (line 5 and 8, Algo.1), since it is the one responsible to evaluate the model performance predicting new class instances similarities in the context of *C-way one-shot* learning. Training pairs are generated from the *GetTrainBatch* function (line 3, Algo.1), which is detailed as follows.

Algorithm 2 GetTrainBatch(*pos*, *ratio*)

Input: ***pos*** - positive pairs to generate,
ratio - positive to negative ratio
Output: ***X*** - spectrogram pairs,
y - spectrogram pairs labels

```
1:  X, y ← Inizialize()
2:  neg ← compute number of negative airs to generate from pos and ratio
3:  for class in training classes do
4:      spect ← get a random sample from current class
5:      for n in pos do
6:          pos_spect ← get a different random sample from current class
7:          X ← add pair <spect, pos_spect>
8:          y ← add label 1
9:      different_classes ← get neg classes different from the current one
10:     for diff_class in different_classes do
11:         neg_spect ← get a random sample from diff_class
12:         X ← add pair <spect, neg_spect>
13:         y ← add label 0
14: return X, y.
```

The batches used to train the models are formed of *positive* and *negative pairs*, labelled as 1 and 0 respectively. This labelling system guides the model in understanding when two inputs are somehow similar or dissimilar. The total number of positive pairs is controlled via the ***pos*** parameter, while its combination with ***ratio*** defines the overall number of negative pairs to be generated (line 2, Algo. 2).

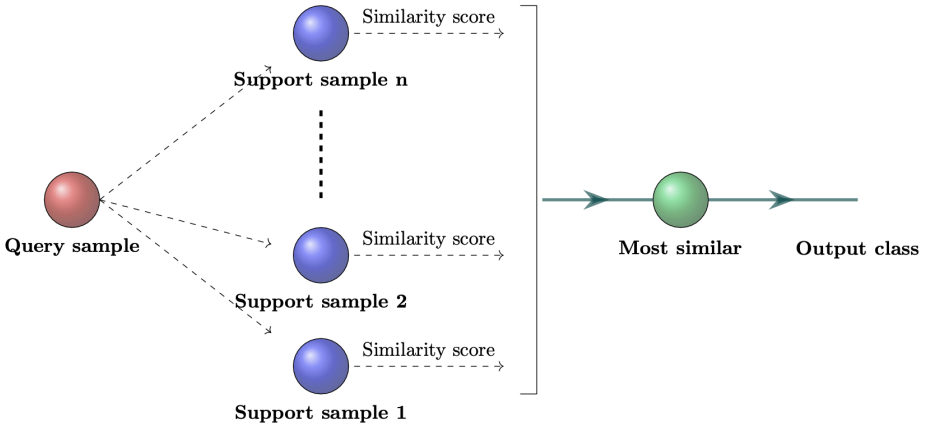


Figure 1.1 *One-shot classification process for C classes (Exhibited by Honka 2019).*

Let us say we want to generate 4 positive and 2 negative pairs per batch, then we would set $pos = 4$ and $ratio = 2 \div 1$. Such parameters setting enables to study how data balancedness influences the model performances. Each training batch results in $C \times neg$ negative and $C \times pos$ positive pairs, where C is the training set class cardinality. For each class a first sample is picked at random and it is paired with pos different spectrogram chosen in a randomise way (lines 4-8, Algo.2). A number of neg samples are then selected randomically from the remaining classes, and paired with spectrogram picked at the very beginning (line 12, Algo.2).

While training, we regularly evaluate the model performance by measuring the mean *C-way one-shot* accuracy on C validation classes. We do this by generating several random *C-way one-shot* classification tasks, where each consists of comparing a ***query sample*** from class $c_i \in C$ to each element $s_i \in S$ of the ***support set*** as shown in Figure 1.1. The support set is composed of C samples, each belonging to a different validation class. Only one element of the support set belongs to the same c_i class that the query sample belongs to. A pair-wise similarity score is computed between the query sample and each element of the support set. Finally, the classification outcome is determined according to the support set sample scoring the highest similarity prediction. This process is repeated for every $c_i \in C$, and the mean accuracy of a given class c_i is computed considering several different iteration where class c_i is fixed as *query class*. Let us discuss the evaluating process detailed

in Algorithm 3.

Algorithm 3 EvaluateOneShot(*model*, *runs*)

Input: ***model*** - model to evaluate,
runs - evaluation runs per class
Output: ***accuracy*** - normalized global mean accuracy

1: *accuracy* ← *0*
2: **for** *class in evaluating classes* **do**
3: **for** *run in runs* **do**
4: *X, y* ← *GetOneShotBatch(class, evaluating classes)*
5: *probs* ← *predict model probabilities on X*
6: **if** *argmax(probs) == argmax(y)* **then**
7: *accuracy* ← *accuracy + 1*
8: **return** *accuracy / (evaluating classes cardinality* × *runs).*

We are interested in monitoring the overall mean accuracy considering all C classes by looping on a specified ***runs*** number of evaluation runs per each of these classes (lines 2-3, Algo.3). In each run we generate a C-way one-shot classification task, and increment the global accuracy when the highest similarity value matches the only pair labelled as 1 (lines 4-7, Algo.3). Finally, we normalize the overall accuracy in the range of *[0,1]*. It is important to draw attention to the fact that in Algorithm 3 we use the term *evaluating classes*, which could indicate classes deriving from both *validating* or *test* sets separately. The *GetOneShotBatch* function (line 4, Algo.3) is responsible to generate one-shot pairs batch on which the model is evaluated on. For the sake of completeness, let us show its specifics in Algorithm 4.

Since the model expects two inputs, we have to generate a stack of C input pairs, where C is the overall number of evaluation classes. We do this by fixing the query sample class ***c*** and we pick at random the element which we refer to as ***query sample***. Then, another random sample from class c is selected, paired with the query sample and marked with label $y = 1$ to form a *positive pair* (line 4-5, Algo.4). Then, $C - 1$ *negative pairs* are formed: the query sample gets paired with $C - 1$ elements each belonging to a different class and labelled as 0.

Looking back at the general workflow described in Algorithm 1, we enter the final *test phase* once the model converges or stops training. This last testing phase is framed within the same framework used to evaluate the model while training, but

Algorithm 4 GetOneShotBatch(*c*, *evaluation_classes*)

Input: *c* - query sample class,
evaluation_classes - validation or test classes
Output: *X* - spectrogram pairs,
y - spectrogram pairs labels

X, y ← *Inizialize()*
query_spect ← *get a random sample from query class c*
pos_spect ← *get a different random sample from query class c*
X ← *add pair <query_spect, pos_spect>*
y ← *add label 1*
for every other class *diff_class* in *evaluation_classes* **do**
    neg_spect ← *get a random sample from diff_class*
    X ← *add pair <query_spect, neg_spect>*
    y ← *add label 0*
return *X, y.*

on a separate test set. The final model performance is therefore measured in terms of mean accuracy of different *C-way one-shot* task using the process we previously discussed (Algorithm 3). The only difference is that here we make use of the test set, instead of the validating one. This 2-tier ***train-validation*** and ***test*** framework allows to build the function f that we want to explain. Evaluating the system in a *C-way one-shot* learning context in two separate moments and on two separate sets of class, is the core aspect of the methodology we followed. At the end of this process (Algorithm 3) we possess a Siamese Network that, when presented with a query sample and a list of possible candidate - all belonging to unseen classes - it returns the one believed to be the most similar to the query input.

2 Experiments

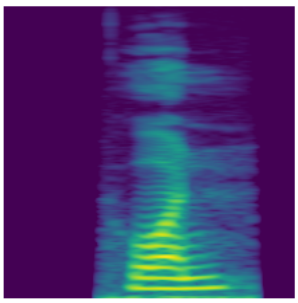
In this chapter we describe the different experiments we conducted to build and explain the architecture of interest. First of all, we introduce the datasets and detail how we pre-processed the data in Section 2.1, and we describe the network architectures in Section 2.2. We conclude detailing how we trained, evaluated and tested the models in Section 2.3.

2.1 Datasets

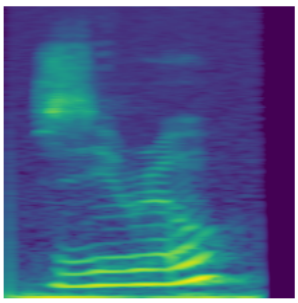
One of the core elements of our project is the audio nature of the input data. We utilize the *AudioMNIST*[1] and *ESC-50*[2] datasets, exploiting the *LibROSA* (McFee et al. 2015) package[3] to pre-process them.

The ***AudioMNIST*** dataset is composed of 30000 audio recordings of spoken digits (0-9) in the English language. Each digit is repeated 50 times for each of the 60 different speakers. These audios were recorded in quiet offices as mono channel signal at a sampling frequency of 48kHz using a RØDE NT-USB microphone. 12 of the speakers that recorded the clips were female, while 48 of them were male and their ages range between 22 and 61 years old. Considering the context of *C-way one-shot* learning we decided to use this dataset to pursue a *speaker recognition* task, due to the high number of speakers and digits repetitions. The high number of available speakers allows us to create three disjoint set: the training set is composed of *50* classes, while the remaining *10* are divided equally between validation and test set. The large number of available repetition for each of speakers, enables us to generate a reasonable amount of both training, validating and testing pairs. To transform the audio inputs, we decided to use their *logarithmic-mel spectrogram representation*, which is commonly used for audio classification tasks (Hershey et al.

[1] `https://github.com/soerenab/AudioMNIST`
[2] `https://github.com/karolpiczak/ESC-50`
[3] `https://librosa.org/`



(a) A man saying one



(b) A woman saying zero

Figure 2.1 *Examples of the AudioMNIST dataset spectrograms.*

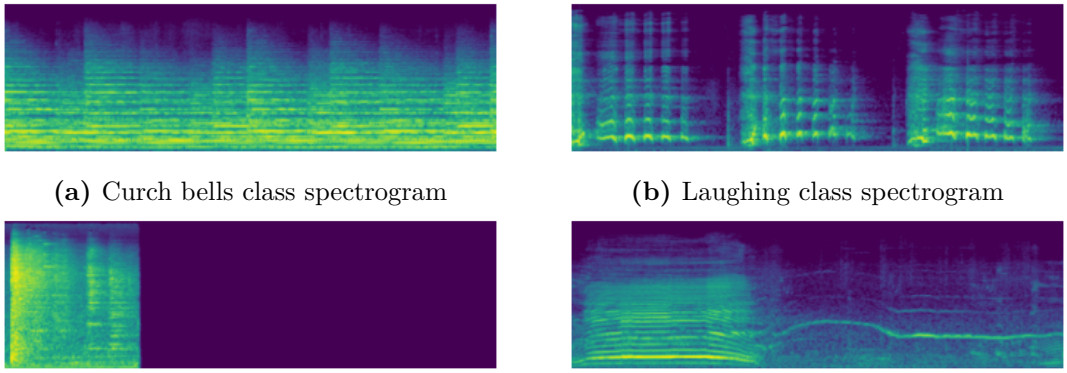
2016, Piczak 2015) and conveniently enough it allows to use architectures originally designed for image classification. Prior to the transformations, the original samples were down-sampled to 41kHz and zero-padded in order to generate input vectors equal in length. While zero-padding, we added silence to the audio (-80 dB) by placing the audio signal at random position within the vector. We followed this approach to add randomness to the data, preventing silence to be placed at the same position for all samples of our dataset. For each sound clip we computed its *log-mel spectrogram* representation using an FFT window size of 4096, hop length of 197 samples and 224 mel-bands[4]. Such process lead to spectrogram of sizes equal to 224×224, that would store enough information to allow an inverse transformation that helped us sonify the spectrograms in the later explaining phase. Figure 2.1 shows two examples of the resulting spectrograms for this dataset.

The **ESC-50** dataset is a collection of 2000 annotated 5-second audio clips divided into 50 different classes and 40 repetition per class. All dataset clips were recorded at a sampling rate of 44.1kHz. Similarly to the *AudioMNIST* dataset, the large category space of *ESC-50* is beneficial for Siamese Networks: a large amount of pairs can be created to train the model and then generalize on new unseen classes. In this case, we decided to pursue an *environmental audio classification* task. We split the dataset so to have *50* training class, *5* validating and *5* testing ones. The dataset recordings can be grouped into 5 macro-categories, each containing 10 classes. This division is shown in Table 2.1 to give the reader a general overview of the kind of classes we will deal with. Here again, we processed the audio clips to recover their

[4]Parameters specifics at `https://librosa.org/doc/main/generated/librosa.feature.melspectrogram.html`

Animals	*Natural soundscapes*	*Human, non-speech sounds*	*Domestic sounds*	*Urban Noises*
Dog	Rain	Crying baby	Door knock	Helicopter
Rooster	See waves	Sneezing	Mouse click	Chainsaw
Pig	Crackling fire	Clapping	Keyboard typing	Siren
Cow	Crickets	Breathing	Door, wood creaks	Car horn
Frog	Chirping birds	Coughing	Can opening	Engine
Cat	Water drops	Footsteps	Washing machine	Train
Hen	Wind	Laughing	Vacuum cleaner	Church bells
Insects (flying)	Pouring water	Brushing teeth	Clock alarm	Airplane
Sheep	Toilet flush	Snoring	Clock tick	Fireworks
Crow	Thunderstorm	Drinking, sipping	Glass breaking	Hand saw

Table 2.1 *ESC-50 classes*



(a) Curch bells class spectrogram

(b) Laughing class spectrogram

(c) Glass breaking class spectrogram

(d) Cow class spectrogram

Figure 2.2 *Examples of the ESC-50 dataset spectrograms.*

log-mel spectrogram representations. Each sound clip was converted using a FFT window size of 2048, hop length of 512 samples and 128 mel-bands. The dimensions of the each spectrogram converted on such parameters result to be of 128×431. In this occasion, the sampling rate was kept at 44.1kHz and no down-sampling was applied before conversion. Example of *ESC-50* spectrogrms resulting from this conversion process is showed in Figure 2.2.

Let us now discuss a final change we made to our data by quickly recalling that in our $x \times y$ spectrogram size definition, the x axes represents ***time*** while the y axes represents the ***frequency*** variable. Despite the fact that spectrograms might be intended as images due to their matrix structure, they still could not be programmatically used as such. An additional *channel dimension* was infact needed to use them as image-line input for common convolution-based architecture. The final spectrograms input dimension would therefore be $x \times y \times 1$.

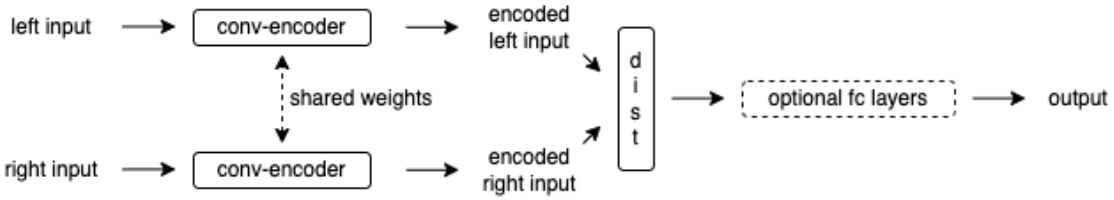


Figure 2.3 *Siamese Network overview*

2.2 Network architectures

In this project we built two distinct networks, one per each of the two dataset described. Before getting into their details, let us briefly present an overview of a typical Siamese Network that process spectrograms by means of Figure 2.3. The overall architecture is composed of two convolution-based encoders, followed by a distance (or merging) layer and a final output scoring layer. Some optional fully connected layers might be present right after the encoded inputs get merged together. The two encoders share the same architecture and weights, which are updated simultaneously during training so to result in identical embedding sub-net. Each conv-encoder is composed of several convolution blocks, usually followed by a flattening and a fully connected layer. The merging layer calculates element-wise metric distance between the embedded inputs. Finally, an output layer consisting of a single fully connected unit functions on such difference and generates a similarity output score between the two given inputs.

Spectrogram representation enables to use architecture originally designed for image-input like AlexNet (Krizhevsky, Sutskever, and Hinton 2012) and VGG Net (Simonyan and Zisserman 2014). This is why the encoders of our networks were implemented as convolution-based architecture. Starting from this well-known architectures and inspired by Becker et al. 2018, we conducted several experiments to adjust the inner encoder structure. In addition, we experimented on different parameters configuration to find the best architecture for each of the two dataset. The hyper-parameters we tested include arrangements for the inner-structure of the encoders as well as variables controlling the overall architecture learning process. Table 2.2 lists all the parameters we investigated together with their search space.

Hyper-parameter	**Search space**
Number of conv layers	*1, 2, 3, 4*
Number of filters per layer	*12, 32, 64*
Pooling layers type	*Average Pooling, Max Pooling*
Fully connected unit size	*1024, 2048, 4096*
Distance metric	*Absolute difference, Squared difference*
Loss function	*Binary crossentropy, MSE, Contrastive loss*
Dropout in conv layers	*0.0, 0.25, 0.5*
Dropout in fully connected layers	*0.0, 0.25, 0.5*
Number of positive pairs	*2, 4, 6*
Positive-to-negative pair ratio	$1 \div 1$, $1 \div 2$, $1 \div 3$, $2 \div 1$, $3 \div 1$

Table 2.2 *Hyper-parameter configurations*

The hyper-parameter tuning phase was conducted in the learning-evaluating framework described in Chapter 1 and the mean *5-way 1-shot* accuracy was marked as the value to maximize. After some preliminary trial runs, the number of maximum training epochs was set to 1500. Such number resulted to be a reasonable trade-off between the model performance and the run time. After 50 training epochs, the model would be evaluated on 100 random *5-way 1-shot* tasks, and the training would stop if no mean accuracy improvements was recorded after 10 of such evaluation runs. The *Adam* optimizer (Kingma and Ba 2014) was used for the loss function minimization process. This framework was applied separately on the two dataset.

While tuning, we noticed some similarities. First of all, both dataset architecture would stop training due to the evaluation threshold when the *MSE* and the *Contrastive loss* were employed. These loss function would not let the model train since its training accuracy score would only decrease progressively. Another common behaviour was found when dropout was in work: the mean one-shot accuracy would result equal to 1 (the maximum) after few evaluation runs and never unlock from this state. After investigating this condition, we noticed that the model would somehow explode and return a similarity score of 1 (the maximum) for every support-set pair. To summarize, dropout would led the model in classifying every pair as completely equal always and forever on both datasets. After noticing such similarities between the models behaviour on the two different datasets, we decided to test the best resulting architecture for *AudioMNIST* to *ESC-50*. Surprisingly enough, the

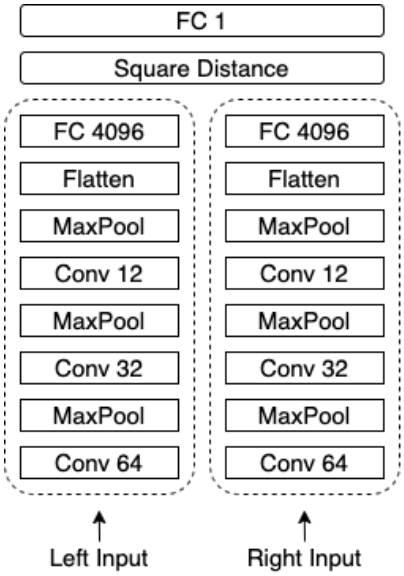
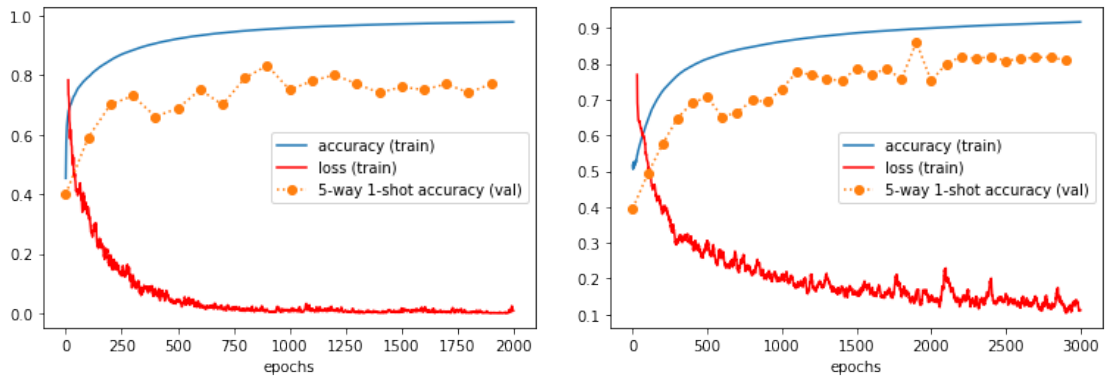


Figure 2.4 *Selected Siamese Network architecture.*

same architecture would achieve the best result on both the two dataset separately.

The final selected architecture, for both the two distinct dataset, is showed in Figure 2.4. The sub-encoders are composed of 3 convolution blocks, each containing a 2D-convolutional layer and a max-pooling layer. ***Conv 64*** is composed of 64 filters and a kernel window of size 5×5, ***Conv 32*** contains 32 filters and a 5×5 kernel size while the last ***Conv 12*** layer is composed of 12 filters with a 3×3 kernel window. Each convolution layer is activated with a rectified linear unit (ReLU) activation function. The ***max-pooling*** layers all share a 2×2 window pool size. The max-pooling layers down-sample the previous layer's output applying a max-filter to sub-regions of specific size, so to reduce the complexity of the input at different stages. Finally, a ***fully connected*** layer of 4096 units takes the input back to a 1-dimension form. The ***square distance*** operates as merging layer between the two encoded inputs, and a final fully connected layer composed of only 1 unit computes the similarity score by means of a sigmoid activation function. The ***positive-to-negative*** pair ratio was selected to be $1 \div 1$, since a slight tendency to overfit was noticed if it would tip to either one of the two pole. The only difference was in the number of ***positive (and negative) pairs*** to generate: *2* and *4* were selected for AudioMNIST and ESC-50 respectively. In both cases, a higher number would drastically slow the training procedure, without leading to better results.



(a) MNIST-Net **(b)** ESC-Net

Figure 2.5 *Training accuracy (blue) and loss (red) curves together with the 5-way 1-shot mean accuracy progress (orange). The validating mean accuracy was measured once every 100 epochs.*

2.3 5-way one-shot learning performance

Let us quickly summarize was we previously described. Two different Siamese Networks were built, one per dataset. They were trained to learn pair similarities on a training set while being validated in a *C-way one-shot* learning framework on a validation set, and then tested on a test set. The training/validation/test set split results in ***50/5/5*** and ***40/5/5*** classes for the *AudioMNIST* and *ESC-50* dataset respectively. Let us now give the details of the training, validation and test phase that the final selected architectures have gone through. In our discussion, we will refer to the two networks as *MNIST-Net* and *ESC-Net*.

Differently from the hyper-parameter tuning phase, the maximum number of training epochs was increased to 5000, with the model performance being evaluated still every 100 training runs but this time on 300 random *5-way 1-shot* tasks. While training, the model weights were updated retrieving the layers gradients with respect to the ***binary crossentropy*** loss value after every training batch. Figure 2.5 shows the *training-validating* process for the two distinct dataset. Both the training procedure stopped because no *5-way 1-shot* mean accuracy improvements was recorded in 10 evaluation runs. *MNIST-Net* reached its best mean 5-way one-shot accuracy after only 900 training epochs with a value of ***0.83 (83%)***, while the process for the *ESC-50* dataset needed 1900 training epochs with a pick value of ***0.86 (86%)***. Additional tests with higher value of the early-stopping threshold were explored, but

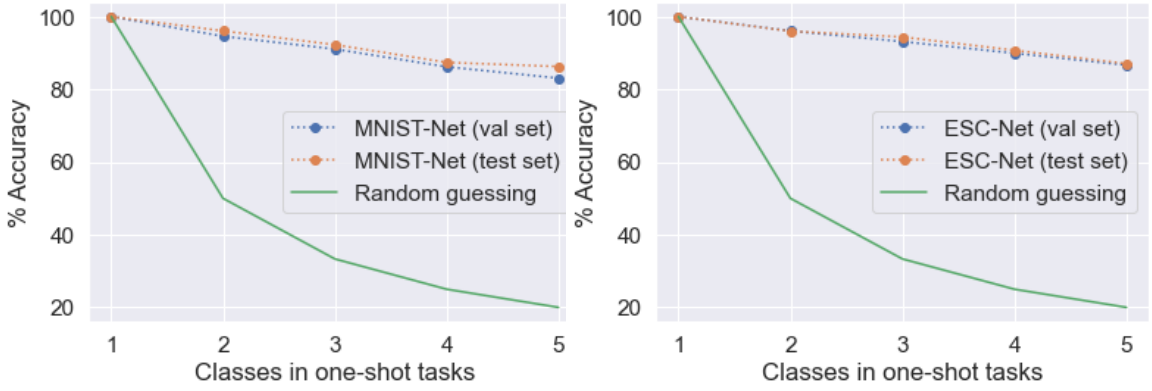
Class	Accuracy
04	93%
56	91%
55	88%
27	82%
46	78%
Mean total	86%

(a) *MNIST-Net*

Class	Accuracy
Glass breaking	99%
Church bells	93%
Frog	91%
Laughing	82%
Door wood creaks	71%
Mean total	87%

(b) *ESC-Net*

Table 2.3 *5-way one-shot mean test accuracy scores.*



(a) MNIST-Net

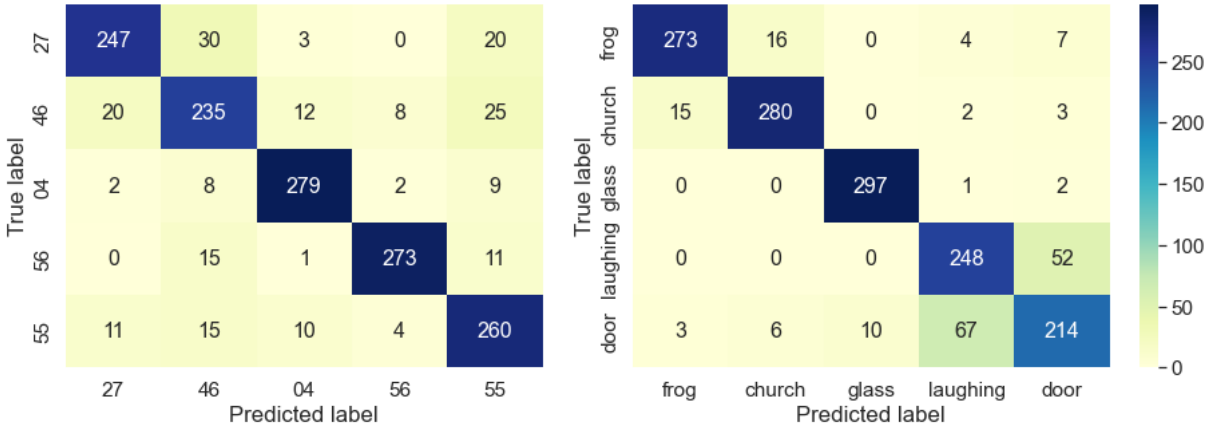
(b) ESC-Net

Figure 2.6 *C-way one-shot mean accuracy while varying C from 1 to 5.*

no overall improvement was found. The additional time needed by *ESC-Net* might be due to the fact that it is composed of a smaller number of training classes. Each training batch would therefore result in a lower overall number of pairs to learn from.

Both networks were tested on test sets composed of 5 unseen classes each[5]. To better comprehend the models behaviour, the mean *5-way one-shot* accuracy was also measured per each singular class. Test results for both networks are listed in Table 2.3. Results show that the network working on the *ESC-50* dataset reaches accuracy values higher than 90% on 3 out of 5 classes, but the *door wood creaks* class lowers the overall value scoring a mean accuracy of 71% singularly. Such value is smaller than the smallest *MNIST-Net* results in, which generally scores values

[5]Not even one sample of the test classes was ever seen during training, leading to a situation which is commonly referred as zero-shot learning. Despite this, in this thesis we use the term C-way one-shot learning to indicate that the additional support set is composed of a singular sample for each of the C classes. We are therefore asking the model to zero-shot the right classification of an unseen query class, by providing exactly one other sample of that same class in the support set.



(a) MNIST-Net **(b)** ESC-Net

Figure 2.7 *Confusion matrix classification results.*

that seems to be better distributed among the 5 classes. The final mean accuracy results to be of ***86%*** for *MNIST-Net* and ***87%*** for *ESC-Net*. A slight improvement of *1%* and *3%* has been recorded from the validation phase for the two networks respectively. This is not surprising, since we evaluated the model measuring the mean accuracy in a *5-way one-shot* framework. Figure 2.6 compares the model accuracy while varying C from 1 to 5 on both validation and test set. The mean accuracy decreases at increasing values of C, making it harder for the model to match the right support set element for a given query input. Interestingly enough, the mean accuracy decreasing behaviour is somewhat similar between the validating and test set on both networks. This might be a counter-proof to the fact that the model is interested in the samples semantic features, more than their class distribution. We can safely state that the training-while-validating framework has successfully trained the model to generalize on new classes.

Finally, let us now visualize the results achieved by the model in form of confusion matrix in Figure 2.7. Following on what the mean accuracy value results suggested, we can see that while the miss-classification are almost equally distributed in *MNIST-Net* (Fig. 2.7a), higher miss-classification values are found for *ESC-Net* (Fig. 2.7b) between the *laughing* and *door wood creaks* classes. This might be due to both *laughing* and *door wood creaks* consisting of sounds generally composed of repeated repetition distributed along a high range of the frequency spectrum.