BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

SPECIALIZATION COMPUTER SCIENCE

DIPLOMA THESIS

Machine Learning in Sound Classification

Supervisor: Lect. Dr. Radu-Lucian LUPŞA

Author: Abrudan Andrei

2019

Contents

Contents 2
Abstract 4
Introduction 4
Background 5
  Neural Networks 5
    Neurons 5
    Layers 6
  Batch size 7
  Epoch 7
  Overfitting 7
  Underfitting 7
  Accuracy 7
  Loss 8
  Gradient Descent 8
  Activation function 9
  Fuzzy Logic 9
TensorFlow 10
  Batch Normalization 11
  2D Convolution Layer 12
  Max Pooling 13
  Flatten 13
  Dense 14
  Dropout 14
How the program works 14
  Audio to Spectrogram 14
  Generating the Dataset 15
  Designing the Model 15
  Compiling the Model 17
    Optimizer: RMSprop 17
    Loss: categorical_crossentropy 17

Metric: accuracy 17
Deploying the Model 17
Obtained results 18
Conclusion 20
Bibliography 21

Abstract

Until recently, the task of examining and classifying sounds was done exclusively by humans. The reason is that automation could simply not cope with the amounts of data that make up an audio file. Furthermore, it is also meaningless to study sounds in vitro, so to speak, where every factor is crispy clear. For such a machine – able to classify music, speech or even background noise – to be reliable, it must be deployable in real life situations, where its receptors are often bombarded with a variety of sound waves.

Thankfully, with the introduction of neural networks (NN) and more specifically convolutional neural networks (CNN), this task is now at least approachable, if not solvable.

Working with raw audio data remains difficult, however. Because of this, there is a preliminary step involved: converting the sound to a spectrogram.

A spectrogram is a visual representation of the spectrum of frequencies of a signal as it varies with time [1]. It allows us to abstract from the audio a simplified image that is easy to work with.

Using the image, it becomes more manageable to process and classify the sound encoded and the neural network can take advantage of built-in mechanisms

Introduction

Machine learning has become an increasingly important field in computer science. Without the use of a neural network, a scientist would have resort to applying a variety of heuristics to a dataset in order to extract the relevant information within. Moreover, by distorting the data, relevant information may be lost resulting in false positives or other mistakes to be made. On top of this, heuristics themselves are by no means easy to invent, much less implement.

Because of this, the necessity of a system which could be fed raw data (or with as few transformations as possible) arose. Thankfully, neural networks came along and provided this exact advantage over traditional applications.

The topic of this paper will focus on how machine learning can be used to classify sounds.

The reason why this is such an interesting topic is because humans are really good at breaking down and classifying sounds (even in a conversation where there is a lot of background noise, we can distinguish words from everything else), but machines have a hard time doing the same.

The general overview of what we will achieve is:

- Extract a simplified version of the audio in what is known as a spectrogram
- Train a neural network to classify the spectrograms
- Use the classifier to label new sounds

Going forward, we will be taking a look at what a neural network is, what parameters we can change to improve both the performance and quality of the neural network, what are the building blocks of a neural network, and how the application was made with respect to all previous information.

Background

Neural Networks

A neural network (NN) is a set of algorithms that attempts to model the human brain in an effort to recognize patterns. These patterns are numerical and work with real numbers. Because of this, a weighting system is applied to determine the strength of a connection between two nodes resulting in fuzzy logic [2], much like a biological brain works. Neural networks come in two major types:

1. **Convolutional Neural Networks (CNN)** which are used to classify (or label) images
2. **Recurrent Neural Networks (RNN)** used in predicting data that follows a trend

though there are a variety of other ways they can be applied, such as in:

- **Multilayer Perceptrons** used to classify data that cannot be separated linearly
- **Modular Neural Networks** which feature different networks that function independently and do not interact with each other during the computation process
- **Feedforward Neural Network** mostly used in face recognition

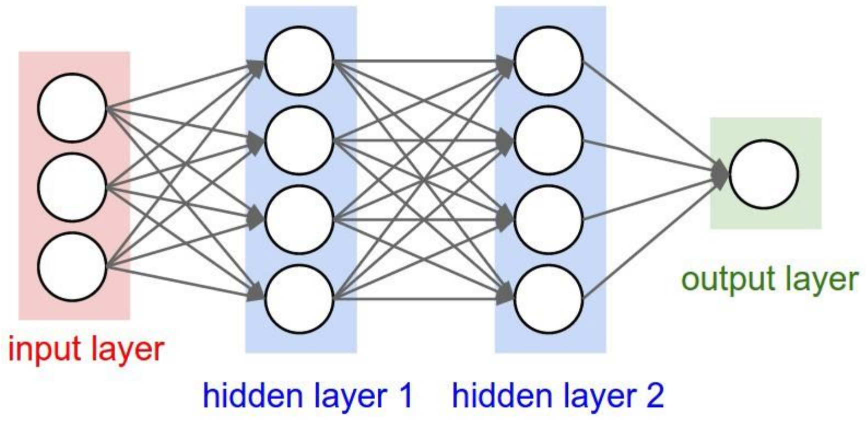


Figure 1 - Neural Network with two hidden layers [3]

Neurons

These networks are made up of neurons which take some input, apply a mathematical formula to it and produce an output. The formula that is applied is known as an *activation function*.

Layers

The neurons are stacked over each other in groups which are called *layers*. All neural networks have at least 3 layers, which are:

Input layer
There is exactly one input layer. Its role is to introduce the data into the neural network for further processing. The selection of this layer depends on the shape of the data. For example, if we want to process an image, we would have a 2D convolution layer as the input layer and the input shape would be a tuple that contains the number of images in a batch, the width of an individual image, its height and the number of channels each pixel contains (i.e. 1 if the image is greyscale, 3 if it is RGB etc.)

Hidden layers
There can be as many hidden layers as we want. They are used to transform the data such that the neural network can make sense of it. They can be thought of as prisms through which the data changes and stimulates more and more only certain neurons that lead to a specific outcome

Output layer
Again, there is only one output layer. It will output a prediction based on the data it receives. The prediction is not a "black or white" outcome, but the neural network's certainty that the result is the correct one. For well trained networks, the certainty gets closer and closer to 100%

Although we have mentioned that there can theoretically be an infinite number of hidden layers, their number can drastically affect the performance (in terms of speed) of the model, and stacking as many layers on top of each other as the machine will allow, does not necessarily result in a better model either. It is crucial that the layers follow a specific idea of how we want the data to be interpreted and as such, the choice of layers in the model becomes the most important part of constructing a reliable network. Later on, we will take a look at the predefined layers at our disposal provided by TensorFlow and what each layer helps us achieve.

The number of neurons in a specific layer is proportional to the layer's place in the neural network. For good results and manageable consumption of resources, layers closer to the input layer should have a relatively small number of neurons. As they get closer to the output layer, this number increases. This culminates with the output layer, which generally has less neurons than the layer previous to it and can be determined by the type of neural network that is constructed:

- In a CNN, the number of neurons in the output layer is equal to the number of classes (labels) we have. For example, if we want to label a picture as either dog, cat or bird, the output layer will have exactly 3 neurons; even though the previous layer might have had as many as 1024
- In an RNN, the output layer always has exactly 1 neuron. This is because an RNN is expected to have only one output which is the data that best fits the trend.

As a rule of thumb, the number of neurons should be multiples 2.

Batch size

This is the number of samples that will be passed through the neural network at one time. Generally, the larger the batch size is, the quicker the model will complete an epoch during training. This is because a machine may be able to process more than one sample at a time. The tradeoff, however, is that even if our machine may be able to handle very large batches, the quality of the model may degrade as we set our batches to be larger.
This parameter should be tested and tuned based on how the model is performing during training. Not only that, the batch size will also have to be tuned in accordance with resource utilization as it affects both memory and processing power consumption.

Epoch

An epoch refers to the number of times we train the neural network with the same input data. Of course, the data is shuffled between each epoch, otherwise it would quickly lead to overfitting. It is desired to have more than one epoch such that our datasets do not have to contain millions of records. The drawback, however, is that setting a large number of epochs will also cause overfitting. At some point, the model will stop generalizing the features of each sample, but instead generalize the sample itself leading to very good training results and very poor predictions or testing results.

Overfitting

We call overfitting, the phenomenon where a model has stopped generalizing the features of the training data, and instead has started classifying the training data itself. This behavior becomes apparent when the model becomes very good at predicting the training data, resulting in accuracies close to 100%, but when faced with new examples, such as testing data, its performance becomes very poor.

Overfitting is a very common problem in the field of machine learning because it manifests in cases when there is little data to work with. Thankfully, this problem can be mitigated simply by adding more data to the training set or performing data augmentation, a process in which copies of examples that suffered slight modifications are added to the training set.

Underfitting

A model is said to be underfitting when it is not even able to classify the data it was trained on well; let alone data it has not seen before. We can tell that the model is underfitting when the metrics given for the training data are poor.

Accuracy

Accuracy is the main metric by which we measure the quality of our model. This value represents the certainty with which the model predicted what is the correct label for some input. Accuracy can only be applied to classification tasks.

Loss

The loss function is a metric similar to accuracy which the Stochastic Gradient Descent (SGD) tries to minimize by continuously updating the weights and the model during training. During the training process, at the end of each epoch, the loss will be calculated on the model's predictions. To obtain this value, the model calculates the error on each input by looking at what output it predicted for it and taking the difference between that value and its correct label.
The following are some commonly used loss functions:

- **Mean Squared Error (MSE)** […] Widely used in linear regression as the performance measure
- **Mean Squared Logarithmic Error (MSLE)** […] used to measure the difference between actual and predicted
- **Mean Absolute Error (MAE)** is a quantity used to measure how close forecasts or predictions are to the eventual outcomes
- **Mean Absolute Percentage Error (MAPE)** is a variant of MAE
- **Kullback Leibler (KL) Divergence** […] is a measure of how one probability distribution diverges from a second expected probability distribution
- **Poisson loss function** is a measure of how the predicted distribution diverges from the expected distribution
- **Cross Entropy** is commonly-used in binary classification (labels are assumed to take values 0 or 1) [4]
- **Categorical Cross Entropy** is used for single label categorization. This is when only one category is applicable for each data point
- **Sparse Categorical Cross Entropy** same as Categorical Cross Entropy, but labels are mutually exclusive

Gradient Descent

Gradient descent is the most commonly used optimization method deployed in machine learning and deep learning algorithms […] Through an iterative process, gradient descent refines a set of parameters through use of partial differential equations, or PDEs. It does this to minimize a given cost function to its local minimum [5].
There are three primary types of gradient descent used in modern machine learning:

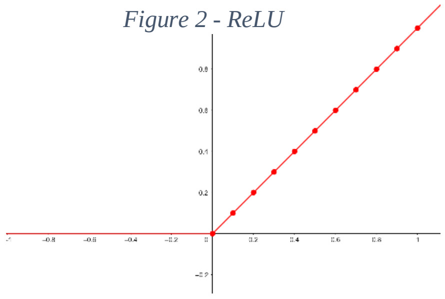
1. **Batch Gradient Descent** calculates the error for each example within the training batch. Its advantages are that it produces a stable error gradient and a stable convergence. The disadvantages are that the state of convergence might not be the best the model can achieve and it requires the entire training set to be loaded into memory to perform the computation.
2. **Stochastic Gradient Descent** updates the parameters according to the gradient of the error with respect to a single training example
3. **Mini Batch Gradient Descent**

We will be using Stochastic Gradient Descent as the optimizer for our model

Activation function

An activation function is simply a mathematical formula that takes a numerical input and outputs another.

Figure 2 - ReLU



In our model, we will be using **Rectified Linear Unit (ReLU)** activation. This is because a study has concluded that deep sparse rectifier networks are more beneficial to image classification tasks than logistic sigmoid neurons [6]. The function can be seen in Figure 2 which can be obtained by plotting $f(x) = \begin{cases} 0 \; for \; x \leq 0 \\ x \; for \; x > 0 \end{cases}$

Another activation function that we will be using is **Softmax**. In mathematics, the softmax function, also known as softargmax or normalized exponential function, is a function that takes as input a vector of K real numbers, and normalizes it into a probability distribution consisting of K probabilities proportional to the exponentials of the input numbers [7]. The formula for for the softmax function is $f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_i}}, for \; i = 1, \dots, J$

As the definition might suggest, we will be using softmax in the output layer to obtain a probability distribution which will represent the certainty with which the model thinks an input should have a certain lable

Fuzzy Logic

Fuzzy logic is an approach to computing based on "degrees of truth" rather than the usual "true or false" (1 or 0) Boolean logic on which the modern computer is based [2]. The reason why this type of logic is useful is because it is so similar to the way biologic neurons work: the stimulation of a nerve is not “digital” (i.e. it is not a case of stimulated vs not stimulated) but instead is an analog process (it can take an infinite amount of states, all in the space [no stimulation, maximum stimulation]).

TensorFlow

TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications. [8]

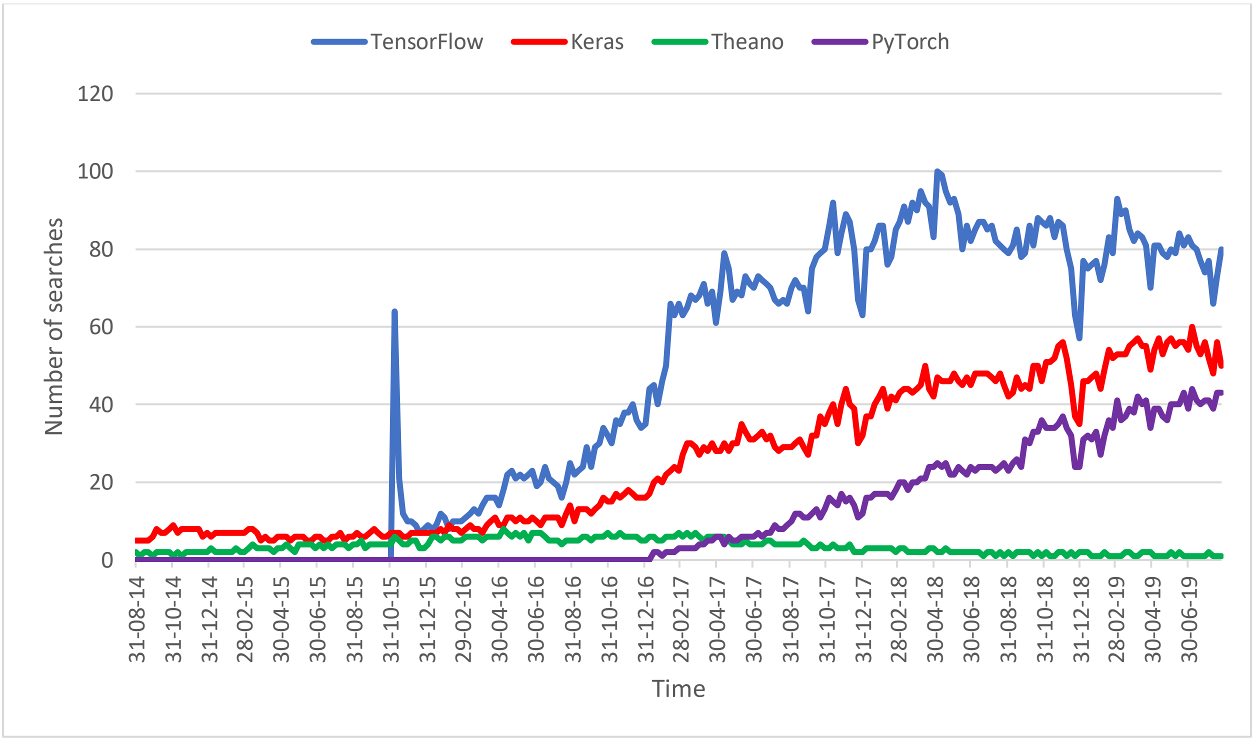


Figure 3 - Comparison between TensorFlow, Keras, Theano and PyTorch interest in the past 5 years

TensorFlow is one of the many machine learning APIs one might use to build a neural network. As can be seen in Figure 3**Error! Reference source not found.**, it quickly gained popularity since its launch in 2015 and still dominates the market to this day (though the graph might be biased since the data was gathered with Google Trends, and TensorFlow itself is a Google company).

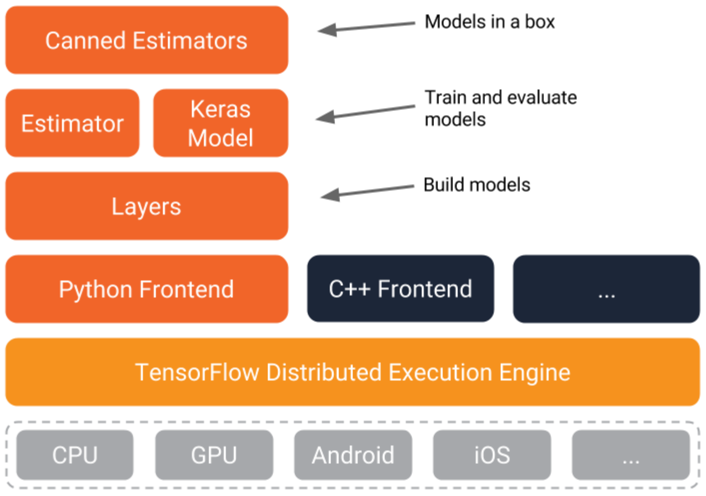


Figure 4 - TensorFlow architecture [17]

Along with TensorFlow, we will be using Keras. Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research. [9]

Figure 4 illustrates the many ways in which we can use TensorFlow. Everything above the TensorFlow Distributed Execution Engine provides a coding style that can take advantage of the complex algorithms which manage the neural network.

As we can see, the python frontend is by far the most developed and mature with Keras being a high-level API that allows us to write succinct code.

As discussed before, we will now have a look at the different layers Keras provides us with in order to build our models:

Batch Normalization

When training a neural network, we want to normalize or standardize our data in some way as part of the preprocessing step (this is a step where we get our data ready for training). The objective of normalization is to transform the data such that all the data points are on the same scale. A typical normalization process consists of scaling the numerical data down to be on a scale from zero to one.

The reason we need to normalize the data, is because we may have some numerical data points in our dataset that might be very high and others that might be very low, or we might have 2 features that are not on the same scale (for example a person’s age would vary on a scale from 1-100 vs their income which would be on a scale from $1.000-100.000). The large datapoints in an unnormalized dataset can cause instability in neural networks because the relatively large inputs can cascade down through the layers in the network and significantly decrease the training speed.

2D Convolution Layer

This layer allows the neural network to pick out or detect patterns and make sense of them. This pattern detection, makes these layers so useful for image analysis.

As the name might suggest, convolution layers are what distinguish other types of neural networks from CNNs.

With each convolution layer, we need to specify the number of filters the layer should have; these filters are what detect the patterns.

Patterns that filters can detect can be anything; some filters may be able to detect edges, others could detect corners, some may detect circles, others squares. These simple geometric shapes are what we would see at the start of the network. The deeper the network goes, the more sophisticated these filters become. So, in later layers, our filters may be able to detect specific objects like eyes, ears, hair, feathers etc. Towards the end, the filters might be able to detect even more complex objects like full dogs, cats and birds.

When the convolutional layer receives input, a filter of, let's say, size 3x3 will slide over each 3x3 set of pixels from the input until it has slid over every block in the image. This sliding is referred to as convolving, so it would be more accurate to say that the filter convolves over each 3x3 block of pixels from the image. The transformation applied is the dot product between the filter and the block over which it convolves.

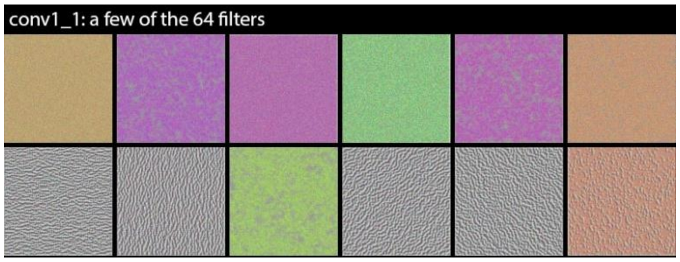


Figure 5 - 2D Conv after applying 64 filters [10]

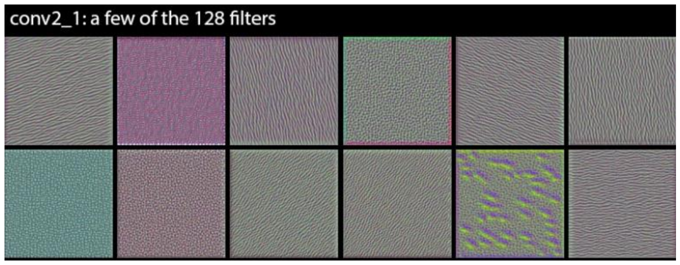


Figure 7 - 2D Conv after applying 128 filters

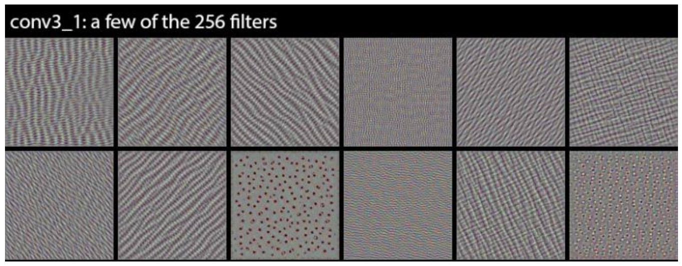


Figure 6 - 2D Conv after applying 256 filters

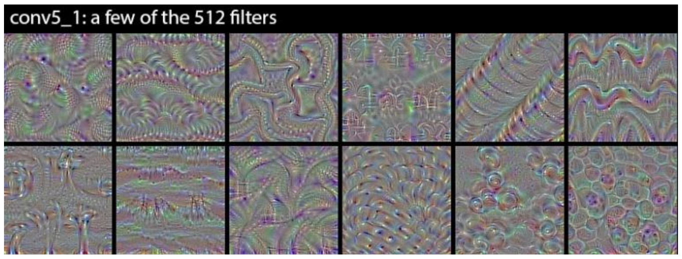


Figure 8 - 2D Conv after applying 512 filters

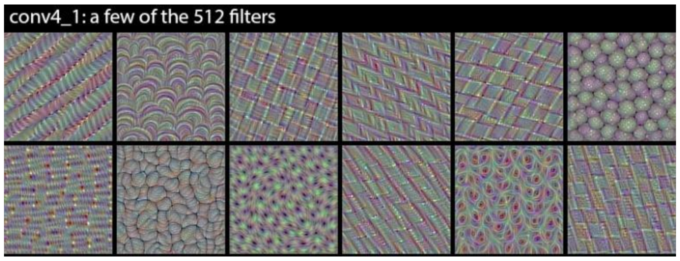


Figure 9 - 2D Conv after applying another 512 filters on top of the previous one

Max Pooling

Max pooling is an operation that is typically added to CNNs following individual convolutional layers. When added to a model, max pooling reduces the dimensionality of images by reducing the number of pixels from the output of the previous convolution layer.

Operation wise, max pooling takes an N x N region as the filter, for example 2x2. The other parameter for this operation is the stride, meaning by how many pixels we want the filter to move as it slides across the image; for the example we will use 2 as well.

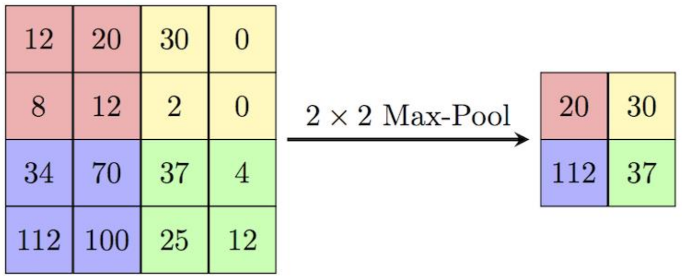
The transformation works as follows:

1. From the output of the convolution layer, we take a region that has the same size as the filter
2. We compute the maximum of that region and write it as the output
3. The region moves to the right by as many pixels as the stride specifies and repeats.
   If we reached the end of the row, the next row begins under the current row, offset by as many pixels as the stride specifies

As we can see in Figure 10 [11], max pooling reduces the amount of data the model has to work with, thus reducing complexity and improving performance.

However, because max pooling is a sort of lossy compression, it becomes dangerous to apply it to images with low resolution or in the case of very fine images where each detail is important.

Figure 10 - Illustration of how MaxPooling2D works



There are other alternatives the operation described above (which can be referred to as Max Pooling 2D as it works with 2 dimensional objects such as images). Those are Max Pooling 1D and 3D which work as the name might suggest.

Flatten

Flatten is part of the reshaping operations, which also include Reshape and Squeeze. These operations alter the shape of the data such that it becomes compatible with other layers we might want to use; for example, the output of a Convolution Layer cannot be used as input for a Dense layer, and so we need to reshape the data.

Flattening a tensor (i.e. the data) means to remove all of the axes, except for one. To reformulate, upon flattening any type of tensor, be it 2D, 3D or even larger, we obtain a vector that contains all the data in a sequential order.

Dense
A dense layer represents an operation in which every input neuron is connected to every output neuron by a weight that constantly gets updated. The name is reminiscent of the densely connected graphs, which strike a resemblance with the way one might imagine the connection between a previous layer and a dense one. We have already seen an example of how dense layers look like back in Figure 1.

Dense layers can be used to reduce the number of outputs of the previous layer. Usually, the output layer is a dense one in a CNN or RNN because of the necessity to restrict the outputs to the number of classes that we have.

Dropout
Dropout is a regularization technique patented by Google [12] for reducing overfitting in neural networks by preventing complex co-adaptations on training data. It is a very efficient way of performing model averaging with neural networks. The term "dropout" refers to dropping out units (both hidden and visible) in a neural network. [13]

How the program works

Audio to Spectrogram
As stated in the introduction, our model will be used to classify sounds, based on a training dataset which provides the audio recordings along with the classes that we will be limited to.

An intermediate step appears: we must first convert the recordings into spectrograms of manageable sizes. This is because of the complexity of the audio encodings. For example, a wave file generally encodes sounds at 44.100 Hz. That means there are 44 thousand samples (which are composed of a minimum and a maximum sample) being recorded across multiple channels (as audio usually comes in stereo format) and the size of each sample depends on the encoding used when creating the file (which can be either 16 or 32 bits).

As one might imagine, this amount of data coupled with the variety of formats can easily overwhelm any neural network. For this reason, we will take advantage of Sound eXchange [14]. SoX is a command line utility that can convert various formats of computer audio files in to other formats. It also provides us with some simple options to convert the audio files to spectrograms of any length.

The length of the audio is important as a larger file would probably contain more than one class. Therefore, it is desired to separate it into smaller chunks which can be individually analyzed by the neural network.

Generating the Dataset

Now that the data we want to process has been converted to images, we can start loading them into memory to create the dataset that will be used for training. It is important to have the data on hand, ready to be processed, as to not starve the CPU/GPU. A major improvement in this regard comes in the form of lazily loading the images, rather than all at once, though this task might prove challenging as once we pass the dataset to the training function, it is out of our hands, but doing multiple trainings will result in a poorer model.

An important step after generating the dataset is establishing the number of classes the dataset has. These classes will be the names of the items that we try to classify (for example, dog, cat or bird), but a model cannot work with strings. Instead, we have to convert the data into a numerical form, which we can achieve through the use of a LabelEncoder.

A possible improvement would be to generate additional images that have slight modifications (such as being tilted, flipped, or even cropped) in order to discourage

Designing the Model

This part of the process will prove to be the most difficult and will require all the knowledge that has been laid down until now.

Figure 11 - The model used

```
model = tf.keras.Sequential([
    Conv2D(16, kernel_size=7, padding='same', activation='relu',
           input_shape=(SPEC_MAX_WIDTH, SPEC_MAX_HEIGHT, SPEC_NUM_CHANNELS)),
    BatchNormalization(),

    Conv2D(24, kernel_size=5, padding='same', activation='relu'),
    MaxPooling2D(),

    Conv2D(32, kernel_size=3, padding='same', activation='relu'),
    MaxPooling2D(),

    Dropout(0.25),
    Flatten(),

    Dense(64, activation='relu'),
    Dropout(0.4),
    Dense(output, activation='softmax')
])
```

Figure 11 shows the model used for this application.

We start with a Convolution Layer. This is because we are working with pictures and we are trying to find patterns that would tell us which class the corresponding sound belongs to. One parameter we have not talked about stands out: padding. Convolving the image does not only apply the filter over the image, but also has the unfortunate side effect of reducing the image size by a couple of pixels. The formula to calculate the dimensions of the output image is as follows: For an $M \times N$ input and a kernel size of $x \times y$, the output will have a size of

$$(M - x + 1) \times (N - y + 1)$$

So, for an image of 4×4 and a kernel of 3×3 the resulting image will have dimensions

$$(4 - 3 + 1) \times (4 - 3 + 1) = 2 \times 2$$

This is where padding comes into play and allows us not to lose any data by extending the image (padding it) with zeroes.

One could also notice that the kernel size is quite a large number: 7. This is because we are working with relatively large images (of resolutions greater than 128×128). At these sizes, it is recommended to have large values for the kernel for performance reasons. As we go deeper into the network, we observe that the kernel size decreases, but the number of filters we apply goes up.

The second layer normalizes the data (i.e. it translates the pixels from range $0 - 255$ to range $0 - 1$). This layer could have been the first just as well, but through experimentation, the results showed that the quality of the model was best in this configuration, slightly worse if normalization came after the last convolution, worse yet if it was the first layer and very poor if normalization was taken out entirely.

The next layers are two groups of Convolving and MaxPooling. Because convolution layers produce as many output images as there are filters (for example, if we apply 16 filters to a single image, we obtain 16 output images) this process is very taxing. This is why we reduce the image size by applying MaxPooling.
In theory, we could have many more such groups. For example, VGG16, a "Very Deep Convolutional Network for Large-Scale Image Recognition", features 13 convolution layers, but, as the article itself says: "It is painfully slow to train" [15].

After the convolution layers follows a Dropout. As explained before, its role is to discourage overfitting. The value of the dropout depends on how well the model behaves and responds to the dropout.

The next layer is Flatten, and it takes us from the 2D territory of images, to 1D as the following layers cannot continue to work with the former shape of data.

The Dense layer featured on line 15 finally allows us to start classifying the data. I would have liked to have more than one dense layer, but unfortunately, my machine could not handle it.

The dropout featured right before the output layer has the same role as the previous one

Finally, the output layer. It is worth noting that the number of neurons in this layer is not fixed, as in previous layers, but instead is a variable which tells the model how many classes the training dataset has.
The output of the model, will be an array of certainty which will tell us what the model thinks the result is.

Compiling the Model

```
model.compile(optimizer='rmsprop',
        loss='categorical_crossentropy',
        metrics=['accuracy'])
```

To compile the model (i.e. to get it ready for training, testing and predicting), we have to specify some parameters:

Optimizer: RMSprop

RMSprop is an unpublished optimization algorithm designed for neural networks, first proposed by Geoff Hinton in lecture 6 of the online course "Neural Networks for Machine Learning" [16]. Without getting into too many details, RMSprop is popular enough that, although it remains unpublished, most frameworks include its implementation out of the box

Loss: categorical_crossentropy

As mentioned before, loss is a metric that we try to minimize. Categorical Crossentropy is a loss function that is able to work with labels that are one-hot encoded.

Metric: accuracy

This is the metric we are interested in and as such, the output of the model will reflect this metric. We are allowed to specify other metrics, or even more than one, but needless to say, accuracy is the most important in classification problems

Deploying the Model

In order to test and train the model, a dataset is in order. To this end, the "Free Spoken Digit Dataset" which can be found on GitHub[1], appears to suit our needs precisely. It features recordings of people speaking the numbers 0 through 9 out loud.

Because generating the dataset also shuffles and reserve a percentage of the data for testing (through the use of function train_test_split, handily provided by scikit-learn, a tools library for machine learning), the next step will be simply to feed the data to the model and wait for it to be trained.

During the training process, we receive some feedback from TensorFlow; this tells us what epoch the model has reached, which batch is currently processed (and how many are left), the loss value and the accuracy of the model. Of course, the two metrics are subject to bias and will need to be compared to the testing results, but they give us an idea on how the model behaves.

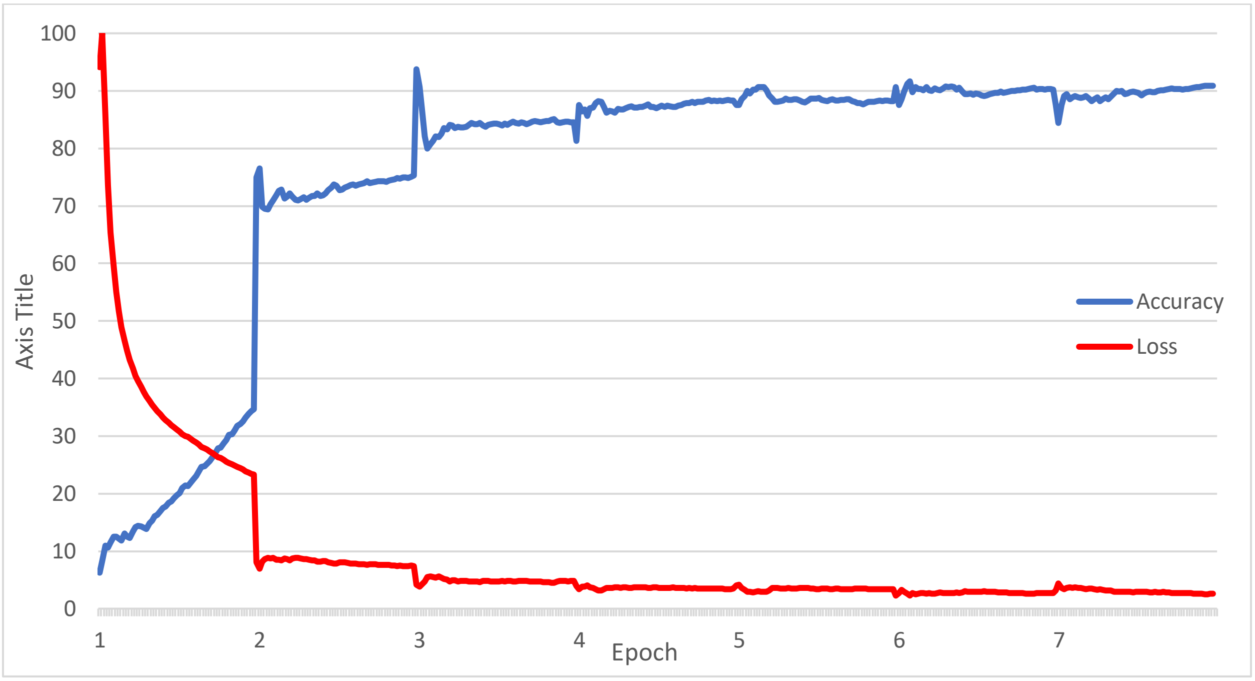
After training, the testing dataset will be fed to obtain a final evaluation, from which we can deduce if the model has been well trained or if it is subject to underfitting/overfitting.

After training, we can now finally use the model to predict on new data it has never seen before.

[1] https://github.com/Jakobovski/free-spoken-digit-dataset

Obtained results

Figure 12 - Quality over time of the model



Upon training the model using the “Free Spoken Digit Dataset”, we obtain the results featured in Figure 12. The dataset provides us with 2000 audio recordings. These have been split into training and testing at a ratio of 19:20, meaning that there will be 1900 training examples, and only 100 will be dedicated for testing. The reason this ratio was chosen, has to do with the size of the dataset. At two thousand examples, this dataset could be considered to be of medium size, therefore, we do not want to starve the model of its learning examples, in favor of a better accuracy measurement.

Figure 12 showcases the evolution of our model across 7 epochs. We can clearly see that the accuracy starts very close to 0% and the loss is maximum (note that the values for loss have been normalized to provide a better comparison to accuracy).

The model’s growth is most obvious during the fist epoch, where the accuracy goes from its minimum, up to 34.6% and the loss drops down to 21.4%.

At the start of the second epoch, we see a sudden spike that propels the accuracy from 34.6% straight to 75%, and a similar phenomenon happens for loss. The explanation for this is because between the epochs, the training algorithm decides to keep only the best performing weights, similar to how in evolution, only the most adaptive individuals tend to survive.

The same kind of spike happens between epochs 2 and 3, but this time around, the values seem to drop back down to match what is happening in the previous epoch. This is a small case of overfitting which is quickly corrected.

From here on out, the model stabilizes, gradually increasing from 80% accuracy up to 91% across epochs 3 through 7. The loss stabilizes much quicker in the generations due to the fact that it has reached a relative minimum, but continues to drop from 10% in generation 2 down to 2.5% in generation 7.

Unfortunately, using this model, the results are underwhelming. Predicting on recordings provided by a text to speech program reveals that the model is unable to work with other types of recordings (that may be cut in different ways or have different lengths). As such, the predictions using real life examples are incorrect.

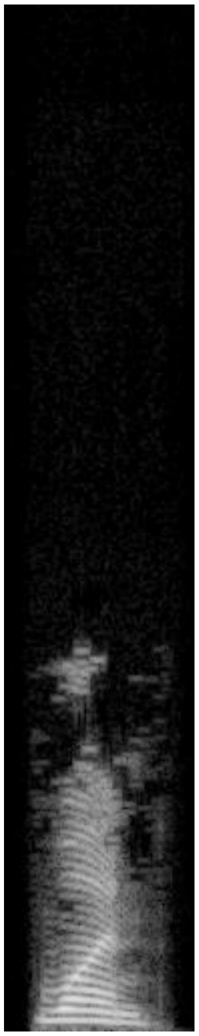


Figure 13 - Spectrogram of a recording of digit 1 taken from a text to speech program

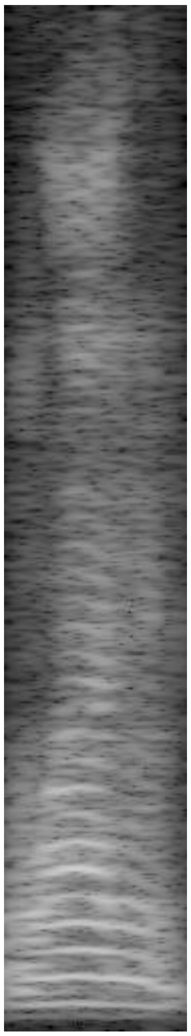


Figure 14 - Spectrogram of a recording of digit 1 spoken out loud from the "Free Spoken Digit Dataset"

A comparison between Figure 13 and Figure 14 reveals that the two spectrograms are indeed much different.

For one, the recording taken from the training dataset features a lot more static noise, while the one produced by the text to speech program does not.

Secondly, if we look closely at Figure 13, we can see that before and after the recording there is a small period of silence. In Figure 14 however, the recording is perfectly cut such that no extra silence is present.

Finally, although not visible in the two adjacent images, the spectrograms have different sizes. The image taken from the dataset has dimensions 91×513, while the image from the text to speech is 103×513. Even if the difference is subtle, this can be enough to throw off a neural network that has trained exclusively on images of the first size.

With this in mind, it becomes apparent that the model is not at fault here, but instead it is the dataset which discourages the model from adapting to real life examples.

As a major improvement, I suggest applying some data augmentation over the dataset such that the model becomes able to cope with a wider range of examples.

Conclusion

Machine learning is an amazing field of research in computer science. Its immediate benefits are those to society: predicting diseases at an early stage with incredible accuracy, defeating humans in games with a high number of variables, providing virtual assistance to people worldwide and overall improving our lives. That is not the end of the story, however. With each incremental improvement to this field, we get closer and closer to understanding the inner workings of the human mind, the greatest mystery of all.

This paper has proven though, that the path is not a smooth one and machine learning is an incredibly complex topic that requires a lot of consideration. After all, we have not even discussed how to build our own neural network from scratch.

Data classification is a venerable niche in deep learning as it allows us to automize thousands of meticulous jobs and even creates new ones which would be impossible without it, such as voice assistants.

The findings in this paper show that even a complex problem such as audio classification can be solved using neural networks. The reason why sounds are so difficult to work with is because there is often more than one event going on (i.e. background noise) during a recording. Therefore, traditional algorithms for analyzing such data come short when presented with real life situations and must rely on complicated heuristics to simplify the data. In comparison to this, neural networks are able to work with raw data and provide an even better performance in terms of quality of their predictions.

TensorFlow is an amazing tool to help us build neural networks. It provides fast algorithms to manage the building, training and deployment of our models. Coupled with Keras, a high-level API, working with neural networks becomes an easy to learn, hard to master type of mechanism. Its other advantages are that it helps write code that is free of bugs and easy to understand.

The model built in this project does have shortcomings, however, the improvements mentioned in this paper would make a difference. Personally, I am pleased with the result as it has provided a valuable learning experience of machine learning and neural networks.

Bibliography

[1] Wikipedia contributors, "Spectrogram," 27 August 2019. [Online]. Available: wikipedia.org/w/index.php?title=Spectrogram&oldid=912782480.

[2] M. Rouse, "Definition Fuzzy Logic," August 2016. [Online]. Available: searchenterpriseai.techtarget.com/definition/fuzzy-logic.

[3] I. Dabbura, "Coding Neural Network — Forward Propagation and Backpropagtion," 1 April 2018. [Online]. Available: towardsdatascience.com/coding-neural-network-forward-propagation-and-backpropagtion-ccf8cf369f76.

[4] I. Changhau, "Loss Functions in Neural Networks," 7 June 2017. [Online]. Available: isaacchanghau.github.io/post/loss_functions/.

[5] ODSC - Open Data Science, "Understanding the 3 Primary Types of Gradient Descent," 26 December 2018. [Online]. Available: medium.com/@ODSC/987590b2c36.

[6] X. Glorot, A. Bordes and Y. Bengio, "Rectifier and softplus activation functions. The second one is a smooth version of the first.," AISTATS, 2011.

[7] Wikipedia contributors, "Softmax function," 1 August 2019. [Online]. Available: wikipedia.org/w/index.php?title=Softmax_function&oldid=908788637.

[8] TensorFlow, [Online]. Available: tensorflow.org.

[9] Keras, [Online]. Available: keras.io.

[10] F. Chollet, "How convolutional neural networks see the world," 30 January 2016. [Online]. Available: blog.keras.io/how-convolutional-neural-networks-see-the-world.html.

[11] Student created article, "Max-pooling," 27 February 2018. [Online]. Available: https://computersciencewiki.org/index.php/Max-pooling_/_Pooling.

[12] G. Hinton, A. Krizhevsky, I. Sutskever and N. Srivastva, "System and method for addressing overfitting in a neural network". United States Patent US9406017B2, 2013.

[13] Wikipedia contributors, "Dropout (neural networks)," 11 June 2019. [Online]. Available: en.wikipedia.org/w/index.php?title=Dropout_(neural_networks)&oldid=901427811.

[14] Sound eXchange, "Sound eXchange," [Online]. Available: sox.sourceforge.net.

[15] Popular networks, "VGG16 – Convolutional Network for Classification and Detection," 20 November 2018. [Online]. Available: neurohive.io/en/popular-networks/vgg16/.

[16] V. Bushaev, "Understanding RMSprop — faster neural network learning," 2 September 2018. [Online]. Available: towardsdatascience.com/62e116fcf29a.

[17] A. Unruh, "What is the TensorFlow machine intelligence platform?," 10 November 2017.