Speech Recognition On Spark

Project Report

Endrit **Halili**

Constantin **Knapp**

Daniel **Putzer**

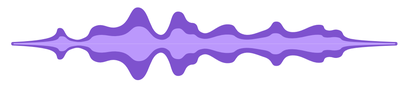
Data Processing II

Winter Semester 2019/2020

WU WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

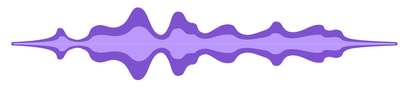
INTRODUCTION

Table Of Contents



INTRODUCTION

The Blessings Of Ignorance

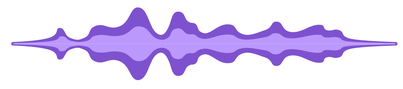


As it often does with these matters, it all started with the blessings of ignorance. Ignorance towards how difficult a thing can really be, when it comes to building something within the constraints of the real world. Twitter was the go-to project that offered itself right from the start. Much of the code had already been there from previous students and the lectures. But we wanted to do something else. University projects are a rare opportunity of creating something tangible with clearly defined time and resource constraints, but without any of the consequences that failure usually brings in a real life setting. We wanted to take advantage of that and risk a little. So we started looking. Big data is everywhere. What could we do with it? First logistics posed some interesting potential projects. Maritime surveillance through satellite images was our first serious contender. Unfortunately, it turned out access to good enough satellite imagery is quite expensive, nothing students could afford. So the search went on. A couple of semi-serious ideas later we settled for Speech Recognition. The goal was to build a system that translates audio to speech in real time. A lofty proposal indeed. But as the motivational quotes on the internet say: “Shoot for the stars and you’ll land on the moon”. We ended up landing somewhere on earth. After coming to the realization that training a model that could deal with all sorts of shapes and formats of audio input is even complexer than we thought we decided to switch to a different dataset and simplify our task to something more achievable. It was a turbulent journey from not knowing where to start over an incredible amount of Google searches without x to finally running our Tensorflow model on Spark Considering the fact that we were completely new to Spark and had only sporadically used Tensorflow before, we dare to say that the result is quite decent, but will leave the final judgement up to the reader.

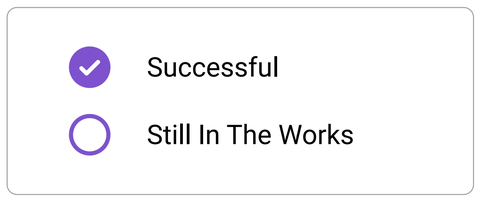
We welcome any kind of feedback or suggestions for future improvements. For this purpose our contact details and GitHub accounts can be found at the end of this report.

ROADMAP

The Road To Overfitting



Following our professor's recommendation, we divided the project into several milestones. These milestones represent smaller, more achievable goals that we worked off one by one. Seeing that the task at hand is anything but easy, especially with our limited experience in the field, this strategy helped to keep motivation high and pressure low. The roadmap shown below illustrates how we laid out the project and shows how far we have come on our journey to creating a not at all overfitting Tensorflow model on Spark.



LOCAL SPARK & TENSORFLOW SETUP

We decided to do most of our work on our own machines for increased flexibility. This required setting up Tensorflow and Spark locally. We ran into some issues (as it is almost always the case with these things), but were able to get both Spark and Tensorflow running, the later with GPU support.

AUDIO PROCESSING

Since we didn't know anything about audio processing in general, we had to first figure out what the best practices for handling audio were in Python. We tried several different approaches and libraries along the way and learned about Fourier transformation, spectrograms and more.

AUDIO PROCESSING ON SPARK

After having figured out how to process audio with Python, the whole approach had to be transferred to Spark. Fortunately, this proved to be less difficult than expected. Spark's Lambda function based structure made applying the previously written audio processing functions to an RDD quite straight forward.

TENSORFLOW MODEL

Using Keras, a high level API built into Tensorflow right out of the box, we set up a simple neural network to get input and output up and running without errors. After successfully doing so, we moved on to experiment with different network structures to improve performance.

TENSORFLOW MODEL ON SPARK

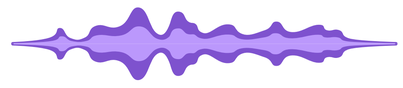
While setting up a model locally was quite doable, porting it over to Spark was challenging. We used TensorflowOnSpark, a Tensorflow distribution built to work on Spark maintained by Yahoo!. Currently the model is training successfully on Spark, but there are still some minor hiccups to be dealt with.

CLOUD DEPLOYMENT

To truly take advantage of horizontal scaling and use Spark to the fullest, we attempted to deploy the model on Google Cloud Platform's Data Proc service. This turned out to be much more complex than expected, as we will see later in this report.

RESOURCES

Libraries & Packages



When we first started out we tried a variety of different libraries, especially for the audio processing part, to find what works best for us. This resulted in a very long and messy list of packages, whose usage was spread out all over the project. After some time working we gained a better understanding of the task at hand and were, therefore, able to streamline our dependencies accordingly. The list below represents the libraries and packages currently in use in the final version of our code. The two most important ones, namely Tensorflow and PySpark, are described in more detail.

Spark

One of the two core libraries of our project. As learned in the lecture, Spark allows horizontal scaling of data processing and manipulation on small and large clusters. For our project we worked with PySpark, the Python wrapper for the underlying Apache Spark.

Tensorflow

Developed and maintained at Google, Tensorflow is one of the most used frameworks for building and deploying neural networks in the industry. With it we were able to build a convolutional neural network for audio classification.

TensorflowOnSpark

Since Tensorflow models cannot be run on a Spark cluster natively, we needed a special distribution of the framework to truly enable horizontal scaling. TensorflowOnSpark is being developed and maintained at Yahoo! and offers a fairly easy way of porting regular Tensorflow models to Spark.

Librosa

As the main driver of our audio processing steps, Librosa offers many useful functions to analyze and manipulate and audio. We used it specifically for Fourier transformation and producing spectrogram data, which was later fed to our Tensorflow model.

Libraries & Packages

Numpy

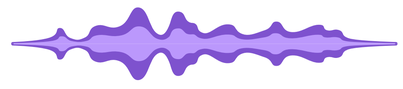
Almost any data science project involves NumPy in some way, ours was no different. It is a dependency of many of the other packages our project is built on and we use it specifically to reshape data to fit our Tensorflow model's input requirements.

PySoundFile

Since Librosa cannot read audio in binary format and Spark forced us to load our samples into memory as binary, we needed PySoundFile as a middle man between Spark and Librosa, taking the binary and returning an audio wave as a NumPy array that could be processed further.

RESOURCES

Data Sources



At first our goal was quite lofty. We wanted to do Speech To Text on all kinds of speech data. We quickly realized, that our experience with machine learning and neural networks wasn't sufficient to build a proper model to handle random snippets of speech and transfer them to written text. We, therefore, decided to look for a simpler application in the field of speech recognition. What we ended up with is a dataset of 30 spoken words that needed to be classified. We outlined both the speech-to-text and the speech classification dataset below.

Mozilla Common Voice

ORIGINAL DATASET

The Mozilla Common Voice dataset offers gigabytes of audio book lines read by volunteers and open for anybody to use. Anybody can go online and contribute to the dataset by recording snippets every day. This would be the perfect data source for our first idea, building a speech-to-text engine.

LICENSE

CC0

- No Copyright
- Commercially Usable
- Public Domain
- No Credit Necessary

LINK

Speech Commands Dataset v0.01

NEW DATASET

Originally part of a Kaggle challenge, this dataset contains audio recordings of 30 words with thousands of samples for each. While the audio processing steps stayed largely the same, using this dataset made the modeling process a bit more manageable. Very complex letter prediction turned into a classification problem.

LICENSE

CC BY

- Sharable (Can Copy & Redistribute)
- Adaptable (Can Remix, Transform & Build Upon)
- Commercially Usable
- Must Give Appropriate Credit

LINK

Legal & Ethical Issues

Every datasource comes with it’s issues. Is the data reliable? Is it accurate enough to yield reasonable performance on the task? Has it been obtained with people’s consent? All those are questions that should be answered as conscientious as possible. As illustrated on the page before, both datasets are distributed under a Creative Commons license, meaning they are free to use as long as the respective requirements are fulfilled. Distributed under a CC0 license, the Mozilla Common Voice dataset can be used however one desires. The Speech Commands Dataset, however, is distributed under a CC BY license and credit must, therefore, be given whenever the dataset is used for full compliance. Let’s handle that right away:

Warden P. Speech Commands

A public dataset for single-word speech recognition, 2017. Available from:

http://download.tensorflow.org/data/speech_commands_v0.01.tar.gz

Biases

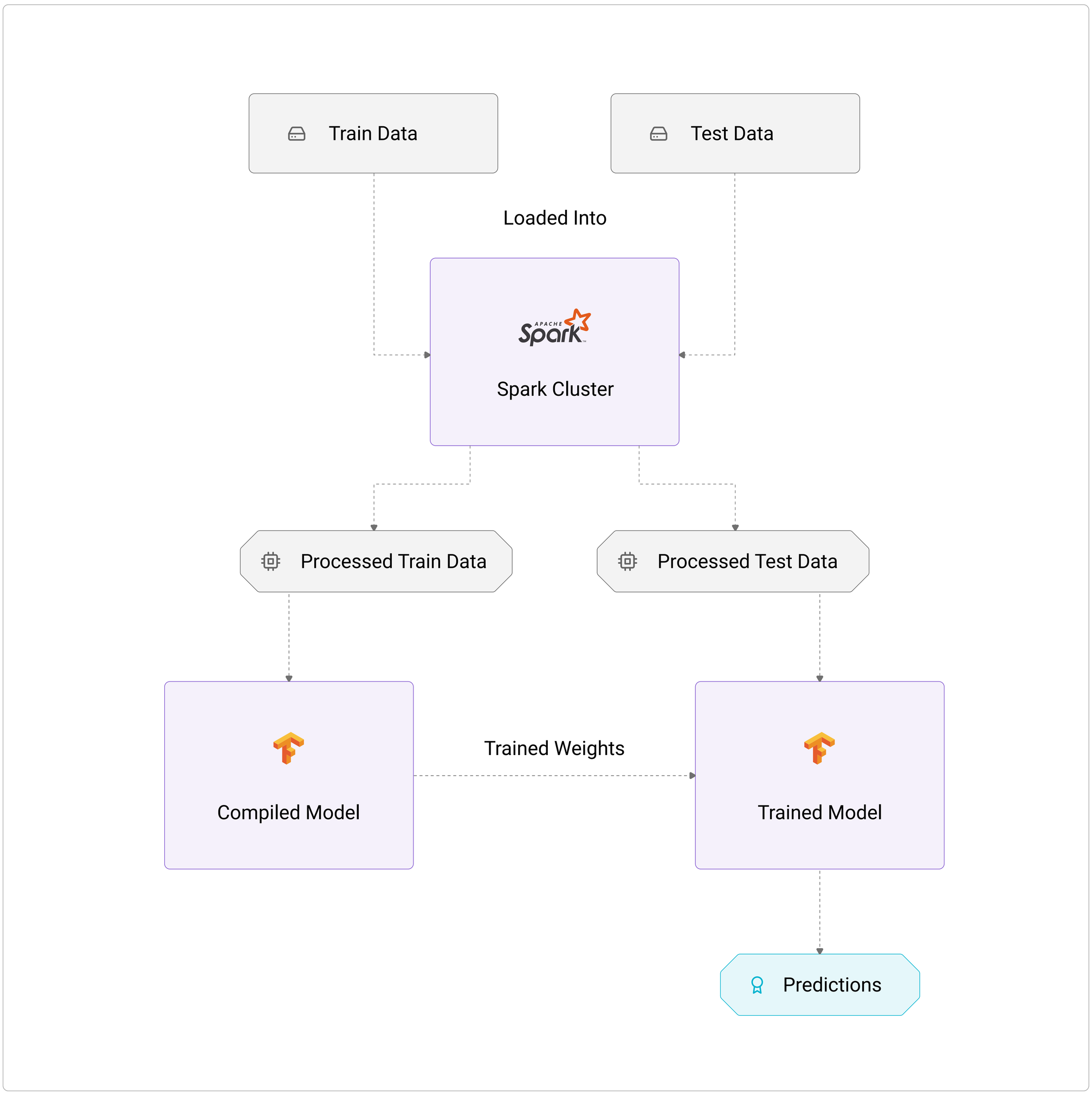
Working with speech data implies encountering a number of biases. One major source of bias could be gender. As the dataset is a collection of real world recordings of people talking, it is really hard to tell if there is any bias towards any specific gender without the proper labels.
Another characteristic of the data leading to bias could be accents. Although the dataset in use consists exclusively of English speech, different regions can have very different pronunciations or vernacularity i.e. slang.

Perhaps the biggest source of bias in our particular project is us. Our limited experience with building neural networks and choosing proper model parameters can influence the result in ways we can hardly anticipate. The main goal of this project is, therefore, to push the limits of our technical abilities and become more confident and competent to avoid bias in future projects.

IMPLEMENTATION

Some Architecture

After cutting all the unnecessary clutter, our architecture turned out to be quite simple and straight-forward. We first process the audio files on Spark, using Fourier transformation to compute a spectrogram for each sample. The spectrogram data is then passed on to a convolutional neural network (CNN) written in Tensorflow. The CNN, which we ported to run on a locally simulated Spark cluster trains on the data and is then ready for prediction. We also attempted to deploy our model on the Google Cloud Platform as a Spark job, but unfortunately ran into some issues.



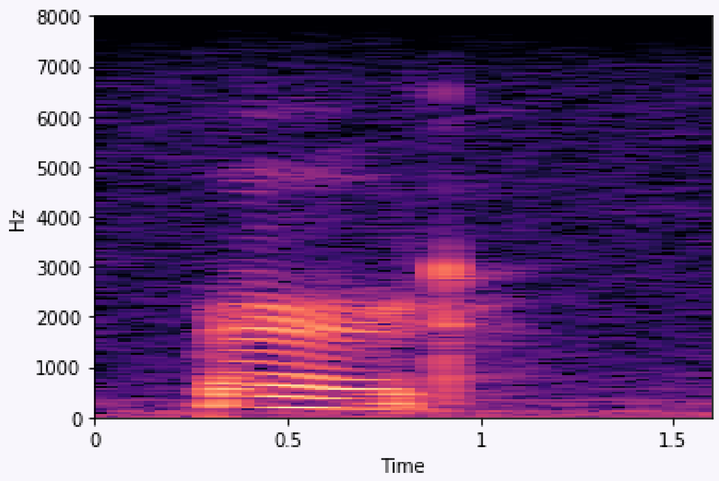
IMPLEMENTATION

How We Processed Audio

The audio processing part turned out to be quite a bit simpler than expected. Using a library called PySoundFile, we read our audio files into Python with a sample rate of 16000 Hz. That means for each one-second long audio clip we got an array of 16000 values in return. Since neural networks require each sample in the input data to be of the same shape, we first padded the samples that were shorter than one second (and therefore returned fewer than 16000 values) up and cut the ones longer than one second down to 16000 values. The next step in the process required further functionality in terms of audio manipulation. There are many packages readily available to manipulate audio of all shapes and formats. We went for the most common one, Librosa. A complex mathematical operation called Fourier transformation is what we needed to continue. Librosa offered just what we needed in it's **`stft`** function. Fourier transformation takes an audio sample and returns a complex value for the energy in each 20ms band (a value chosen by us to get 50 bands for each one-second sample). We turn those values into absolutes to get rid of the imaginary part of the complex value.

Then we convert them into energy distributions across frequencies using Librosa's **`amplitude_to_db`** function. The final result of these steps is a so-called spectrogram. An example can be found below.

The Spectrogram



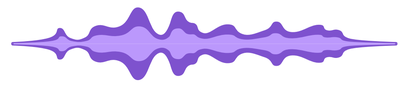
As mentioned before, spectrograms represent an audio wave's energy distribution across a frequency range over time. In this particular example the lower frequencies have visibily higher energy values, hinting at a lower, potentially male voice.

```
def processSample(path, sample):
    audio = sf.read(os.path.join(path, sample))[0]

    if len(audio) < 16000:
        audio = np.concatenate((audio, np.zeros(16000 - len(audio),)), axis=0)
    elif len(audio) > 16000:
        audio = audio[:16000]

    return librosa.amplitude_to_db(abs(librosa.stft(audio, hop_length=321)))
```

The code snippet shown above demonstrates how we handled the audio samples.

IMPLEMENTATION

Bringing Processing To Spark



Processing the audio data on Spark is nearly identical to processing it locally. The only thing that changes is the process of loading the data into memory. Since Spark does not natively support reading audio files, we were forced to treat the audio files like binary files. Fortunately, PySoundFile can handle the binary format and we did not have to change the rest of our pipeline. We use the same packages, namely Librosa and PySoundFile, to read the binary files from memory and apply Fourier transformation to get the spectrogram data. We decided to split the function seen in the previous section into two parts for convenience.

```
# Function to convert audio stored as binary in-memory to numerical data
def binaryToNumerical(x):
    return sf.read(io.BytesIO(x))[0]
```

Reading binary data from cluster memory

```
# Function to perform Fourier transformation on audio snippets to get energy in different
# frequency ranges
def fourierTransformation(x):
    audio = librosa.amplitude_to_db(abs(librosa.stft(x, hop_length=321)))

    # Padding audio up to one second length if it is shorter
    if audio.shape[1] < 50:
        filler = np.zeros((1025, 50 - audio.shape[1]))
        audio = np.concatenate((audio, filler), axis = 1)

    return audio
```

Applying Fourier transformation

```
# Loading data into memory
baseAudio = sc.binaryFiles(paths)

# Applying Transformations To Data
convertedAndLabeledAudio = baseAudio.map(lambda x:
    [carveClassName(x[0]), binaryToNumerical(x[1])]
)

transformedAudio = convertedAndLabeledAudio.map(lambda x:
    [x[0], fourierTransformation(x[1])]
)
```

Mapping our processing functions to the data

IMPLEMENTATION

Enter The Neural Net

To reach our goal, namely classify the recordings of the 30 different words in our audio data, we obviously needed a model. After some research we decided to try our luck with a neural network. We considered other approaches, such as Hidden Markov Models, but ultimately settled for a neural network, because we wanted to work specifically with Tensorflow. As the reader might or might not know, a neural network consists of several layers. These layers can be of different types and there are quite a few to choose from. In order to find a fitting neural network type and structure, some more research had to be done first. Given that after processing the audio, our data technically consists of spectrograms, i.e. graphical representations of the audio, it can be seen as image data. Convolutional neural networks (CNN) are the industry standard when it comes to image recognition. We decided to stick to what we read and implemented a CNN to classify our audio. But how does a convolutional layer work? What is "convolution"? Let's take a brief look.

Layers

Convolutional Layers

In a convolutional layer, one or more so-called kernels, i.e. a matrices of a certain size (3x3 in our case), are moved across the image from top left to bottom right. For each kernel-step we compute a value through a convolutional operation. The center of the kernel is considered the source pixel and convolution aims to recognize patterns in the surrounding pixels. To improve performance it is, therefore, recommended to use uneven numbers for the kernel size. This way the neural network learns features in the image and can recognize them in later examples.

Although it is generally considered a convolutional neural network, there is much more to it than convolutional layers. Let's take a look at all the other layer types that are part of our model in a little more detail on the next page.

What Is A Neural Network?

A neural network is a form of artificial intelligence inspired by the biology of brains. Just like in a brain, they consist of layers of nodes, also called neurons, that are connected through weights, the equivalent of synapses in the brain. To illustrate how the process works, let's first take the human brain as an example. When you open your eyes, your brain gets fed an image. Interconnected neurons in your brain then fire, (or don't) which leads to you being able to tell what you are looking at. Similarly, the data fed to a neural network is passed through an activation function at each node (neuron), which determines whether it fires or not. The strength of the connections between layers is determined by the previously mentioned weights, the digital sisters and brothers of our biological synapses. Some are stronger, some are weaker and as the neural network sees more data, they change and adjust to "learn" to identify patterns within the data. The illustration below shows a simple two-layer neural network.

Layers & Activation Functions

Although it is generally considered a convolutional neural network, there is much more to it than convolutional layers. Let's take a look at all the other layer types that are part of our model in a little more detail on the next page.

Pooling Layers
The Pooling layer's inner functionality is very similar to a convolutional layer, but instead of applying a convolutional operation to all the values in each kernel (see convolutional layer explanation for the definition of the term "kernel" in this context), it simply selects the highest value within each kernel and creates a new, lower-resolution representation of the sample at hand.

Dropout Layers
In order to avoid overfitting, that is learning the data by heart, a neural network can be subjected to regularization. Regularization is the process of penalizing i.e. lowering the impact of the weights of certain nodes (See 'What Is A Neural Network' box on the previous page for more information) to counter overfitting. Adding Dropout layers to your network structure is one of many ways to regularize a neural network. A dropout layer simply ignores (or drops out, so to say) a random set of nodes every iteration of training. This leads to the current layer being treated like it has a different number of nodes with different connections each iteration, giving the previous layer a different view of it.

Batch Normalization
As is often done with the data before feeding it to a model, scaling can improve performance and training speed. Batch normalization takes this approach to the weights within a neural network by scaling the activation values of each node. This increases performance and training speed.

Dense Layer
A dense layer is the most basic layer. It is often used to demonstrate or explain the basic functionality of neural networks. It simply connects each node in a layer to each node in the following layer. Dense layers are, therefore, often called Fully Connected layers.

As mentioned beforehand, some of the layers require a so called activation function. Let's explore what those activation functions are and how they work.

Activation Functions

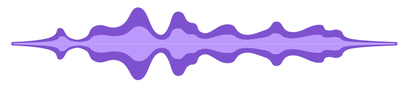
Activation functions are complex mathematical operations, just like all the other operations happening inside the black box-like neural network. Simply put, an activation function determines whether a neuron in our neural network should fire or not (See 'What Is A Neural Network' box). For our project we used two different activation functions.

Rectified Linear Unit or short ReLU simply returns the value passed to it if it is above zero or zero if it is below.

Softmax is a form of logistic regression that returns probabilities distributed among all values passed to it that sum up to one.

IMPLEMENTATION

The Network In Tensorflow



```
# Initiating model
model = Sequential()

# Building model structure
model.add(Input(shape=(1025, 50, 1)))

model.add(Conv2D(16, kernel_size=[3,3], activation='relu', data_format='channels_last'))
model.add(MaxPool2D(pool_size=[3,3], data_format='channels_last'))
model.add(Dropout(0.2))

model.add(Conv2D(32, kernel_size=[3,3], activation='relu', data_format='channels_last'))
model.add(MaxPool2D(pool_size=[3,3], data_format='channels_last'))
model.add(Dropout(0.2))

model.add(Flatten())
model.add(BatchNormalization())
model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(128, activation='relu'))

model.add(Dense(30, activation='softmax'))
```

Implementation of a convolutional neural network with Dropout and Batch Normalization in Keras

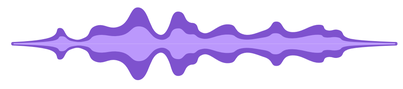
After all this information, what does such a neural network look like in practice? The snippet above shows our implementation of a convolutional neural network in Tensorflow, taking advantage of the high-level Keras API.

As can be seen, our model is of the sequential type. That means each layer's output is the following layer's input, the data gets passed from layer to layer sequentially. For reasons explained earlier, a kernel size of three by three was chosen rather arbitrarily. Inspired by a Kaggle kernel we decided to have two convolution and pooling pairs with dropout layers in between followed by two more dense layers. The two-dimensional output of the convolutional and pooling layers needs to be flattened i.e. reduced to one dimension, to be able to be passed on to the dense layers. While the convolutional layers learn the features and patterns found in the images, the dense layers are the ones doing the actual classification.

The dense layer at the very end of our network structure returns an array of probabilities corresponding to each of the 30 words in our dataset.

IMPLEMENTATION

Training On Spark



Once we had TensorFlow and the neural network up and running, everything had to be ported to Spark. As always, here too we were standing on the shoulders of giants. TensorflowOnSpark is a Tensorflow distribution developed and maintained by Yahoo!. It enables the training of neural nets distributed on a cluster. On the official TensorflowOnSpark GitHub page they claim porting a model can be done with less than ten lines of code. This turned out to be true, but only after a few hours getting it wrong and trying to understand the process.

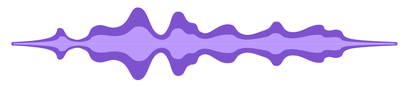
First we had to turn our Tensorflow model into a distributed Tensorflow model. To our surprise this functionality is built into Tensorflow natively and requires only minor additions to our codebase. This requires selecting a so-called distributed strategy. Following examples from the TensorflowOnSpark GitHub repository, we used the so-called MultiWorkerMirroredStrategy. To quote the Tensorflow documentation, “It implements synchronous distributed training across multiple workers”, meaning it makes separate copies of the neural network’s weights and parameters for each node in the cluster and then trains separately on each.

A distributed Tensorflow model, however, is not the same as a Tensorflow model on Spark. To get to that point, a few more changes were necessary. First of all we had to wrap our model into a function that could be called later. We then turned the data stored in-memory on the cluster into a so-called Tensorflow dataset, this simplifies feeding data to the model. Along the way we also split the data into features (the variables to be trained on) and labels (the variable to be predicted). This ended up being messy. Tensorflow can be very particular with the shape of the data you feed to it, but we didn’t know that at first. A few error messages later we finally figured out how to reshape the data and continue building. To distribute the model we had to create an instance of TensorflowOnSpark’s TFCluster class. We then passed the Spark context, the model wrapped in the function earlier and a few other parameters like cluster size. To be able to pass our data to the model later on in the fit method, we also had to set the input_method parameter. Since we are passing it a Spark RDD we chose input mode SPARK.

On the next few pages you can see the entire script porting and running the Tensorflow model on Spark.

IMPLEMENTATION

The Spark Job

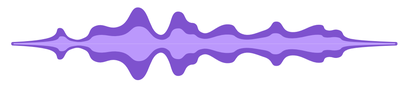


```
from __future__ import absolute_import, division, print_function, unicode_literals

def mainFun(args, ctx):
  import numpy as np
  import tensorflow as tf
  from tensorflow.keras.models import Sequential
  from tensorflow.keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPool2D, Input,
      BatchNormalization
  import tensorflow.keras as keras
  from tensorflowonspark import compat, TFNode

  # Setting distributed model strategy
  strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy()

  def buildAndCompileModel():

    # Initiating model
    model = Sequential()

    # Building model structure
    model.add(Input(shape=(1025, 50, 1)))

    # First convolution and pooling step
    model.add(Conv2D(16, kernel_size=[3,3], activation='relu', data_format='channels_last'))
    model.add(MaxPool2D(pool_size=[3,3], data_format='channels_last'))
    model.add(Dropout(0.2))

    # Second convolution and pooling step
    model.add(Conv2D(32, kernel_size=[3,3], activation='relu', data_format='channels_last'))
    model.add(MaxPool2D(pool_size=[3,3], data_format='channels_last'))
    model.add(Dropout(0.2))

    # Flattening output of convolution to pass on to Dense layers
    model.add(Flatten())
    model.add(BatchNormalization())
    model.add(Dense(128, activation='relu'))
    model.add(BatchNormalization())
    model.add(Dense(128, activation='relu'))

    # Output layer
    model.add(Dense(30, activation='softmax'))

    # Compiling model
    model.compile(
      loss='sparse_categorical_crossentropy',
      optimizer='adam',
      metrics=['accuracy']
    )

    return model
```

Our model from before wrapped in a function

IMPLEMENTATION

The Spark Job

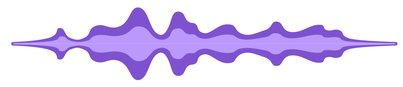


```
# Opening up datafeed to iterate over entries
tfFeed = TFNode.DataFeed(ctx.mgr, False)

# Function to split data into features and labels
def rddGenerator():
  while not tfFeed.should_stop():
    batch = tfFeed.next_batch(1)
    if len(batch) > 0:
      example = batch[0]

      # Splitting into X and y
      X = np.array(example[1]).astype(np.float32)
      y = np.array(example[0])

      # Encoding labels
      _, y = np.unique(y, return_inverse=True)
      y = y.astype(np.float32)

      # Adjusting data shape
      X = X.reshape(-1, 50, 1)

      # Returning features and labels as separate arrays
      yield (X, y)
    else:
      return

# Creating Tensorflow Dataset
ds = tf.data.Dataset.from_generator(
  rddGenerator,
  (tf.float32, tf.float32),
  (tf.TensorShape([1025, 50, 1]),
  tf.TensorShape([1]))
)
ds = ds.batch(1)

# Instantiating Model
with strategy.scope():
  multiWorkerModel = buildAndCompileModel()

  # Defining Training Parameters
  stepsPerEpoch = 600 / 1
  stepsPerWorker = stepsPerEpoch / 1
  maxStepsPerWorker = stepsPerWorker * 0.9

  # Fitting Model
  multiWorkerModel.fit(x = ds, epochs = 2, steps_per_epoch = stepsPerWorker)

# Exporting log files for Tensorboard
from tensorflow_estimator.python.estimator.export import export_lib
exportDir = export_lib.get_timestamped_export_dir(args.export_dir)
compat.export_saved_model(multiWorkerModel, exportDir, ctx.job_name == 'chief')

# Terminating feed tells spark to skip processing further partitions
tfFeed.terminate()
```

Preparing data for training and instantiating model

IMPLEMENTATION

The Spark Job

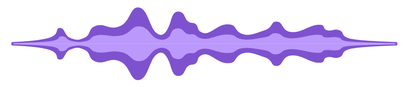


```
if __name__ == '__main__':
  # Initializing Spark
  import findspark
  findspark.init()

  # Importing Spark-related Things
  import argparse
  from pyspark import SparkContext
  from pyspark.sql import SparkSession

  # Importing TensorflowOnSpark
  from tensorflowonspark import TFCluster

  # Importing other packages
  import numpy as np
  import io
  import os
  import librosa
  import random
  import soundfile as sf

  # Creating Spark Session
  spark = SparkSession.builder \
  .master('local') \
  .appName('AudioAttempt') \
  .config('spark.executor.memory', '8gb') \
  .getOrCreate()

  # Creating Spark context
  sc = spark.sparkContext

  # Function to carve out class from file path
  def carveClassName(x):
    return x.split('/')[-2]

  # Function to convert audio stored as binary in-memory to numerical data
  def binaryToNumerical(x):
    return sf.read(io.BytesIO(x))[0]

  # Function for Fourier transformation on audio to get energy in different frequency ranges
  def fourierTransformation(x):
    audio = librosa.amplitude_to_db(abs(librosa.stft(x, hop_length=321)))

    # Padding audio up to one second length if it is shorter
    if audio.shape[1] < 50:
      filler = np.zeros((1025, 50 - audio.shape[1]))
      audio = np.concatenate((audio, filler), axis = 1)

  return audio
```

Spark setup and definition of audio processing functions

IMPLEMENTATION

The Spark Job



```
# Trial data paths
paths = 'data/local/bird,data/local/bed,data/local/cat'

# Loading data into memory
baseAudio = sc.binaryFiles(paths)

parser = argparse.ArgumentParser()
parser.add_argument("--batch_size", help="records per batch", type=int, default=1)
parser.add_argument("--cluster_size", help="nodes in the cluster", type=int, default=1)
parser.add_argument("--export_dir", help="path to export", default="speechModel")

args = parser.parse_args()

# Applying Transformations To Data
convertedAndLabeledAudio = baseAudio.map(lambda x:
  [carveClassName(x[0]), binaryToNumerical(x[1])]
)
transformedAudio = convertedAndLabeledAudio.map(lambda x:
  [x[0], fourierTransformation(x[1])]
)

# Defining Cluster
cluster = TFCluster.run(
  sc,
  mainFun,
  args,
  num_ps = 0,
  num_executors = 1,
  tensorboard = True,
  input_mode = TFCluster.InputMode.SPARK,
  master_node = 'chief'
)

# Training on cluster
cluster.train(transformedAudio, num_epochs=3)
# Shutting down cluster after training is complete
cluster.shutdown()
```

Applying functions to data and running distributed model

IMPLEMENTATION

A Tale Of Deployment

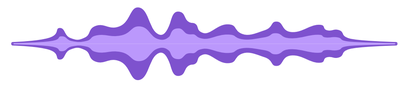
While setting up and distributing the model on our simulated local Spark cluster works, to truly train it a real cluster was necessary. We had some free Google Cloud Platform credits laying around so we thought might as well give it a shot. Google offers a service called Data Proc, specifically geared towards setting up Spark clusters to submit jobs to. The setup seemed quite easy at first and we had a cluster running within a few minutes. We quickly realized that a bare cluster won't do much, so we tried to install all the packages we needed and submit some test jobs. A few hours of trying yielded nothing more than loads of error messages, so for lack of time we decided to finish the rest of the project and leave cloud deployment for the future.

A Humble Suggestion

We had great fun working with Spark and Tensorflow and get our first hands-on experience with big data. However, our environments were limited to our local machines or the university servers. When trying to deploy on Google Cloud Platform, we realized that the actual big data workflow seems to be quite different from what we've seen in class or built in our projects. While we were and are able to work with the basic Spark API, accessing hosted data from a hosted cluster through a submitted Spark job is an entirely different beast. This is a very broad and difficult topic, but we do believe that it is worth exploring at some point in the course or at least the Data Science SBWL, as we also struggled with cloud deployment in the Data Analytics course. We could see the Data Processing II course taking up an entire semester instead of just a half-semester. This would enable a deeper look into Spark and how to work with it in a non-university setting, that means working with platforms like GCP, AWS or Azure. This, however, is just a humble suggestion from our side. We believe this is a very integral part of being a data scientist that is currently not content of the data science SBWL as far as we know and heard.

OUTLOOK

The Future & Our Takeaways



Future Improvements

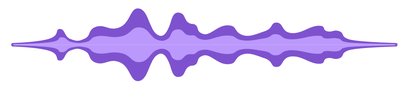
While we missed the mark on building a speech-to-text engine, from a learning perspective our project was a great success. Nonetheless, there are always improvements one can make.

As mentioned before, deploying our model on a cloud platform that offers the necessary power to truly train in a distributed way is one thing we definitely want to get right at some point. Online learning platforms like Udemy should offer the necessary auxiliary education to fill in this knowledge gap of ours. Furthermore, making good on our first promise of building the speech-to-text engine would also be interesting. While quite a few things we've already done could be adapted for this task, further research into neural network structures and speech processing would be necessary.

Takeaways

When we decided to break with the status quo and try something that no team has done before, we knew we were taking some risks. There was no guarantee that we would even come close to building any of the things we set out to build. Following our professor's advice we split the task into smaller goals that we could work off one by one (See the roadmap on page two for more detail). By doing this, even if we did not reach our final goal, we could at least celebrate the achievement of some smaller milestones along the way. As expected the task turned out to be quite difficult, but in the end we managed to do almost all the things we initially set out to do, although with a somewhat simpler data source. We learned a thing or two about how digital audio works and can be manipulated and processed. We learned how to build a simple convolutional neural network in Tensorflow and some basics about how to set parameters. We learned more about wrangling data on a Spark cluster and, finally, we learned how to port a simple Tensorflow model to Spark. Even though the deployment on the cloud did not work as expected, our main goals have been achieved and the only thing remaining is proper training of the model so improvements can be made to its performance. All in all, we are quite happy with our results and are looking forward to further refine our skills and apply our knowledge to future big data projects.

The Team



Endrit Halili

H11705087

Creation of spectrogram data, making data model ready

Special Packages

Librosa, PySpark, PySoundFile

GitHub

https://github.com/endrithalili

Constantin Knapp

H11703634

First exploration of data, preprocessing of audio data

Special Packages

Librosa, PySpark

GitHub

https://github.com/CoKn

Daniel Putzer

H11712865

Design and implementation of neural network

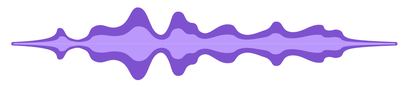
Special Packages

Tensorflow, TensorflowOnSpark, PySpark

GitHub

https://github.com/DanThePutzer

Setup



Packages

The following packages need to be downloaded and installed on the system in order to run our project files.

```
Tensorflow                  Numpy
TensorflowOnSpark           Matplotlib
PySpark                     TQDM
Findspark
Librosa
SoundFile
```

Data

The dataset we used can be downloaded through the following link:

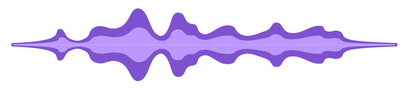
https://www.kaggle.com/c/tensorflow-speech-recognition-challenge/data

We recommend cloning our repository and following the steps listed in the README file.

The repository can be found at the following link:

https://github.com/DanThePutzer/transcribe

Sources



Basics Of Audio Processing And Spectrograms

https://medium.com/@ageitgey/machine-learning-is-fun-part-6-how-to-do-speech-recognition-with-deep-learning-28293c162f7a

TensorflowOnSpark Github

https://github.com/yahoo/TensorFlowOnSpark

TensorflowOnSpark Documentation

https://yahoo.github.io/TensorFlowOnSpark/index.html

Neural Network Structure Inspiration

https://www.kaggle.com/alphasis/light-weight-cnn-lb-0-74

Convolutional Neural Networks

https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac

More Convolutional Neural Networks

https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/

Softmax Function Explained

https://towardsdatascience.com/softmax-function-simplified-714068bf8156

Dropout In Neural Networks

https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/

Rectified Linear Unit Explained

https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/

Librosa Documentation

https://librosa.github.io/librosa/