Getting started on Deep Learning for Audio Data

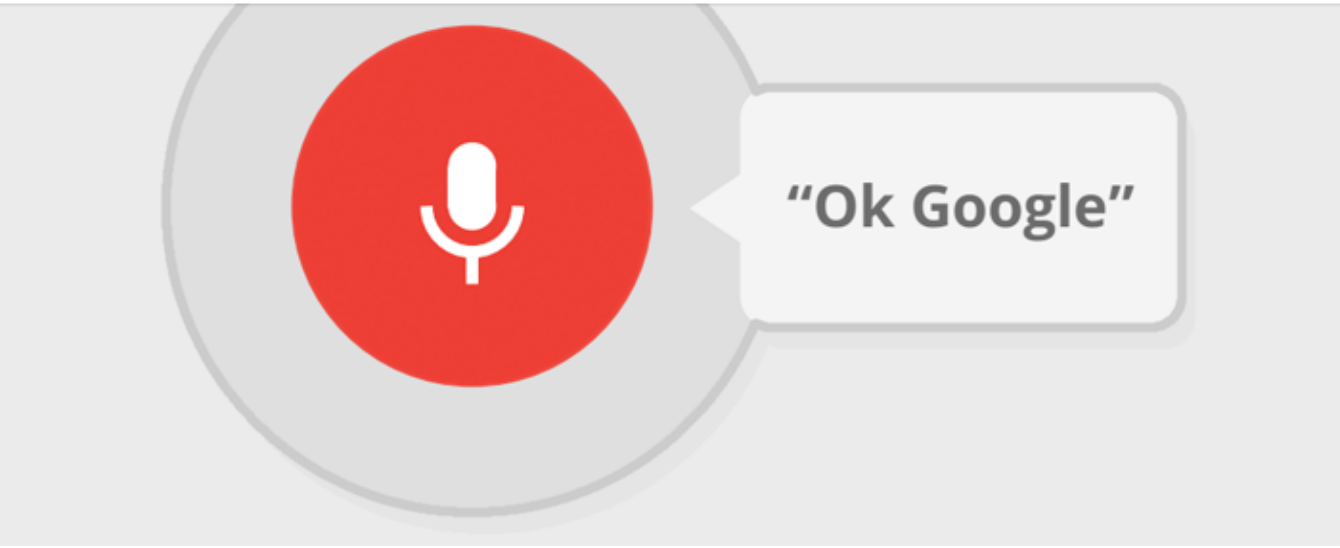
Tirmidzi Faizal Aflahi
Sep 25, 2019 · 9 min read ★

While traditional Machine Learning algorithm still triumphs on structural data with the insurgence of Gradient Boosting method, nothing beats Deep Learning on processing unstructured data.

Photo by Hitesh Choudhary on Unsplash

Let's get back to the title.

Audio Data, are you sure?



Ok, Google!

I can surely assume that most of us have tried this thing.

Have you ever think of how that feature built?

Deep Learning on Unstructured Data

Yes, unstructured data is by far the most suitable data for Deep Learning. While you can also use it on structured data, traditional Machine Learning can beat the performance easily on a moderate amount of data.

Things like Self-driving Cars, face recognition, or the notorious FaceApp that went viral several months ago, are the by-product of Deep Learning system on images data.

Image data = pixels of colors = unstructured data

Basically, unstructured data means a bunch of data points, without a mean to statistically analyze them. Well, you can't use any form of standard deviation on pixels, so images are unstructured.

So be texts and audios.

Audio data is a bunch of wave signals sequenced one after another. You can't possibly

In layman's term, the flow should be like this:

Your phone listens to surrounding words → someone says the keyword, "Ok, Google!" → the assistant app activates.

What!

My phone listened to me every time? Isn't that a violation of privacy?

Well, I don't want to talk about that problem with this article. Let's move on to step 2 and 3.

If your phone listens to the keyword, it opens the app.

So, in this particular tutorial, I want to share on how you can build an algorithm that able to distinguish the keyword from other voices.

Deep Learning tutorial on Audio Data

Let's take a look at Kaggle,

There is a competition on how to distinguished Turkey (the animal) sound from other voices. Why turkey? I don't know. At least, it fits our needs. **It doesn't have to be a human voice as the keyword, does it? Lol.**

Download the data, and you can see the training data (train.json)

train.json data

You will see a JSON list with 1195 items, which means, you have 1995 items as training data.

Each item will have 5 attributes,

1. The video ID, which is the id of video from YouTube
2. Start Time of the clip
3. End Time of the clip (So, not all video is used, only 10 seconds of the video)
4. is_turkey, determine is that a turkey or not
5. The audio embedding, this is the 10 seconds embedding from the audio. They use the scripts in https://github.com/tensorflow/models/tree/master/research/audioset to change audio waves into numbers. We can just use the processed result.

For example, you can watch this awesome turkey video,

Awesome Turkey

Watch from the 30th seconds to 40th. You know that there are turkeys, even though the background music disturbs the peace.

The preparation

Wait, I am getting overwhelmed.

Or you can just go from what I have curated before.

So, now you are already familiar with the jargons and other kinds of stuff, you are ready to go.

Python and packages

Of course, Python. You need to install it first from the Python official website. You need to install version 3.6+ to keep up to date with the libraries.

And install other things from the terminal,

```
pip install numpy
pip install pandas
pip install seaborn
pip install matplotlib
pip install tensorflow
pip install keras
pip install jupyter
```

Let's type *jupyter notebook* from the terminal and we are ready to go!

```
import numpy as np
import pandas as pd
import os
import seaborn as sns
import matplotlib.pyplot as plt
import seaborn as sns
plt.style.use('fivethirtyeight')
from tqdm import tqdm
print(os.listdir("../input"))
```

Prepare the libraries, and you can see the output

```
['train.json', 'sample_submission.csv', 'test.json']
```

```
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential,Model
from keras.layers import LSTM, Dense, Bidirectional,
Input,Dropout,BatchNormalization,CuDNNLSTM, GRU, CuDNNGRU,
Embedding, GlobalMaxPooling1D, GlobalAveragePooling1D, Flatten
from keras import backend as K
from keras.engine.topology import Layer
from keras import initializers, regularizers, constraints
from sklearn.model_selection import KFold, cross_val_score,
train_test_split
```

And of course, prepare your data

```
train = pd.read_json('../input/train.json')
display(train.shape)
```

It will show

```
(1195, 5)
```

Yep, you got 1195 data to work on.

Let's take a look inside the data

```
train.head()
```

```
xval = [k for k in train_val['audio_embedding']]
yval = train_val['is_turkey'].values
```

train_test_split method by default splitting the data by 3:1 means 75% of the data goes into the training set, while the rest 25% to the validation set.

To create a standard on the data, let's pad all audio embedding with zero until all of them have the same length of 10 seconds.

```
# Pad the audio features so that all are "10 seconds" long
x_train = pad_sequences(xtrain, maxlen=10)
x_val = pad_sequences(xval, maxlen=10)

y_train = np.asarray(ytrain)
y_val = np.asarray(yval)
```

Let's create the model!

```
model = Sequential()
model.add(BatchNormalization(momentum=0.98,input_shape=(10, 128)))
model.add(Bidirectional(CuDNNGRU(128, return_sequences = True)))
model.add(Flatten())
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer =
optimizers.Nadam(lr=0.001), metrics=['accuracy'])
print(model.summary())
```

The detail would be like this:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
batch_normalization_7 (Batch (None, 10, 128)           512
```

```
Non-trainable params: 256
_________________________________________________________________
```

Take a look at this code

```
model.add(Bidirectional(CuDNNGRU(128, return_sequences = True)))
```

What the hell is that?

Photo by Sean O. on Unsplash

Let’s see the beautiful sunrise before continuing. And refresh your brain of course. Lol

Working with audio data

Both of these algorithms are really good at processing sequence of numbers. Basically, they can save what they have "read" and use that information when processing the next number. **That "short term memory" gives them an edge over other algorithms.**

And in this tutorial, I used GRU

Let's get back to the code

```
model = Sequential()
model.add(BatchNormalization(momentum=0.98,input_shape=(10, 128)))
model.add(Bidirectional(CuDNNGRU(128, return_sequences = True)))
model.add(Flatten())
model.add(Dense(1,activation='sigmoid'))
```

I create a simple model on Keras,

1. Add BatchNorm layer to standardize the input numbers (The audio inputs have not standardized yet)
2. Add the Bidirectional GRU to process the input
3. Flatten the result
4. Create a sigmoid Dense Layer for "True-False" or binary problems.

Training Time!

```
#fit on a portion of the training data, and validate on the rest
from keras.callbacks import EarlyStopping, ModelCheckpoint,
ReduceLROnPlateau

reduce_lr = ReduceLROnPlateau(monitor='val_acc', factor=0.1,
patience=2, verbose=1, min_lr=1e-8)

early_stop = EarlyStopping(monitor='val_loss', verbose=1,
```

```
Epoch 16/16
 - 0s - loss: 0.1037 - acc: 0.9710 - val_loss: 0.1694 - val_acc:
0.9231
```

It has 92.31% Accuracy on validation test! Awesome!

Let's plot the training history

```
def eva_plot(History):
    plt.figure(figsize=(20,10))
    sns.lineplot(range(1, 16+1), History.history['acc'],
label='Train Accuracy')
    sns.lineplot(range(1, 16+1), History.history['val_acc'],
label='Test Accuracy')
    plt.legend(['train', 'validaiton'], loc='upper left')
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.show()
    plt.figure(figsize=(20,10))
    sns.lineplot(range(1, 16+1), History.history['loss'],
label='Train loss')
    sns.lineplot(range(1, 16+1), History.history['val_loss'],
label='Test loss')
    plt.legend(['train', 'validaiton'], loc='upper left')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.show()

eva_plot(history)
```

This simple model can actually do something great. We don't even do any optimization on it.

Well, if you want to optimize, you are welcome to do it.

Although, I want to do another form of optimization. I don't know about the result, so why not.

Attention Seekers!

No, I mean, attention mechanism. Take a look at the image below

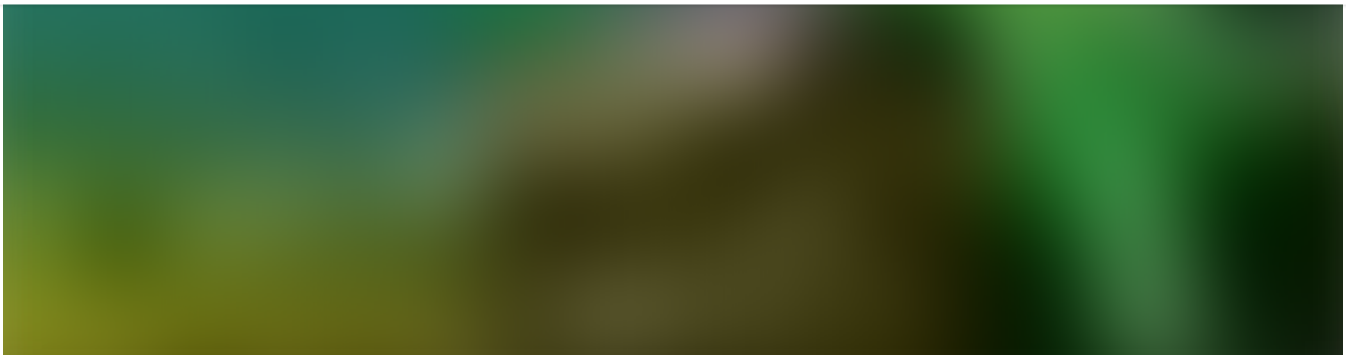


Photo by Zdeněk Macháček on Unsplash

What did you see?

Of course, **the bird**

Not the greenish background in the image.

That's how human perceive. And we want to simulate that mechanism to the machine.

Attention! Attention!

word.

This will make the machine keeping with the context.

The implementation

Thanks to our friend at kaggle, we can use this awesome class,

```
class Attention(Layer):
    def __init__(self, step_dim,
                 W_regularizer=None, b_regularizer=None,
                 W_constraint=None, b_constraint=None,
                 bias=True, **kwargs):
        self.supports_masking = True
        self.init = initializers.get('glorot_uniform')

self.W_regularizer = regularizers.get(W_regularizer)
        self.b_regularizer = regularizers.get(b_regularizer)

self.W_constraint = constraints.get(W_constraint)
        self.b_constraint = constraints.get(b_constraint)

self.bias = bias
        self.step_dim = step_dim
        self.features_dim = 0
        super(Attention, self).__init__(**kwargs)

    def build(self, input_shape):
        assert len(input_shape) == 3

        self.W = self.add_weight((input_shape[-1],),
                                 initializer=self.init,
                                 name='{}_W'.format(self.name),
                                 regularizer=self.W_regularizer,
                                 constraint=self.W_constraint)
        self.features_dim = input_shape[-1]

        if self.bias:
            self.b = self.add_weight((input_shape[1],),
                                     initializer='zero',
                                     name='{}_b'.format(self.name),
                                     regularizer=self.b_regularizer,
                                     constraint=self.b_constraint)
        else:
```

```
    def call(self, x, mask=None):
        features_dim = self.features_dim
        step_dim = self.step_dim

        eij = K.reshape(K.dot(K.reshape(x, (-1, features_dim)),
                        K.reshape(self.W, (features_dim, 1))), (-1,
step_dim))

        if self.bias:
            eij += self.b

        eij = K.tanh(eij)

        a = K.exp(eij)

        if mask is not None:
            a *= K.cast(mask, K.floatx())

        a /= K.cast(K.sum(a, axis=1, keepdims=True) + K.epsilon(),
K.floatx())

        a = K.expand_dims(a)
        weighted_input = x * a
        return K.sum(weighted_input, axis=1)

    def compute_output_shape(self, input_shape):
        return input_shape[0],  self.features_dim
```

Don’t force yourself to understand the thing. Let’s just change our model

```
model = Sequential()
model.add(BatchNormalization(momentum=0.98,input_shape=(10, 128)))
model.add(Bidirectional(CuDNNGRU(128, return_sequences = True)))
model.add(Attention(10))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer =
optimizers.Nadam(lr=0.001), metrics=['accuracy'])
print(model.summary())
```

Change the flatten layer into attention. And you will see

```
attention_2 (Attention)      (None, 256)               266
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 257
=================================================================
Total params: 199,179
Trainable params: 198,923
Non-trainable params: 256
_________________________________________________________________
```

It reduces around 2,000 trainable params. Well, that's a good thing. Let's try the training

Training v2

```
early_stop = EarlyStopping(monitor='val_loss', verbose=1,
patience=20,  restore_best_weights=True)

model.fit(x_train, y_train,batch_size=512,
epochs=16,validation_data=[x_val, y_val],verbose = 2,callbacks=
[reduce_lr,early_stop])
```

Now you get:

```
Epoch 16/16
 - 0s - loss: 0.1037 - acc: 0.9710 - val_loss: 0.1680 - val_acc:
0.9331
```

Yeah! A 1% increase into **93.31%**

Not much, but definitely an increase!

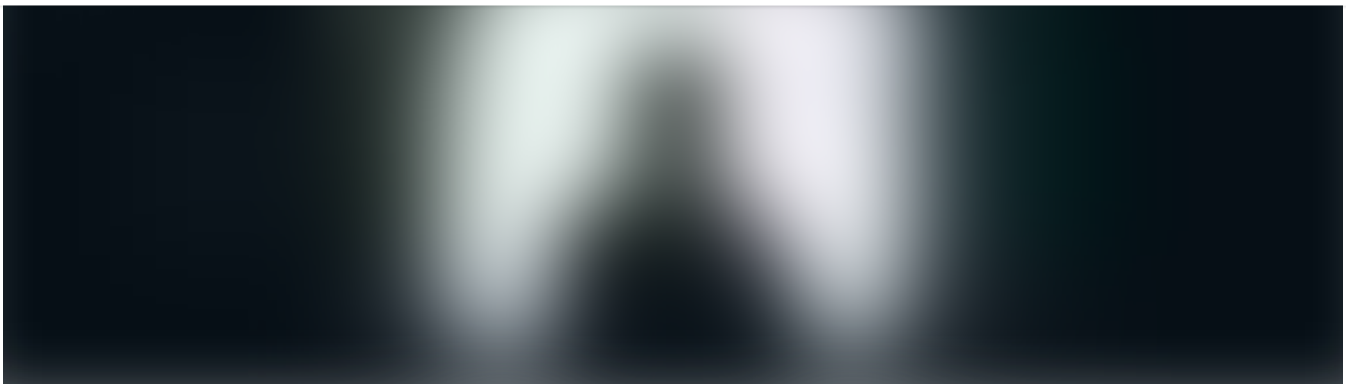


Photo by Warren Wong on Unsplash

Conclusion

It is amazingly easy to get started to create a Deep Learning system for audio data.

You just need to know-how. And get good to create the real “Ok, Google!”.

Go ahead and try. Of course, you can try the tutorial and change the parameters or the network however you want.

Good luck!

Machine Learning Deep Learning Artificial Intelligence Audio Towards Data Science