Report Data Science Lab – Laboratory 7

The purpose of laboratory 7 is to analyze a dataset of 2000 recordings (.wav) of 4 different people saying numbers from 0 to 9 with english pronunciation and build a model capable of recognizing and classifying new audio files.
The dataset is divided into:

- 1500 labeled audio files used for training (**/dev** folder)
- 500 unlabeled audio files used for final evaluation (**/eval** folder)

The first step is **reading all the files**.
The first 1500 files are named with the id followed by the label (a number from 0 to 9).
The last 500 are named just with the id (used for submission).

```
dev_files = os.listdir(dev_path)
eval_files = os.listdir(eval_path)

for file in dev_files:
    dev_data["data"].append(wavfile.read(dev_path + "/" + file, True))
    name, label, _ = re.split('[_.]', file)  # split file name to id and label
    dev_data["label"].append(label)
    dev_data["id"].append(name)

for file in eval_files:
    eval_data["data"].append(wavfile.read(eval_path + "/" + file, True))
    name, _ = re.split('[.]', file)
    eval_data["id"].append(name)
```

The second step is **preprocessing**.
This is done by removing silences from files and applying a zero padding to match the length of the longest record.

```
def preprocessing(data, length=None):
    records = [np.array(x[1], dtype=float) for x in data]  # every record has also
                                                             the sample rate as first element

    print("Trimming") # removing silences
    trimmed = trim_data(records, TRIM_THRESHOLD)
    print("Padding")  # zero padding to uniform lengths
    padded = zero_padding(trimmed, length)
    return padded
```

The **trim** function is taken fom *libROSA* library, as additional argument is requested the decibel threshold (in my case is 10 dB), everything below the threshold will be cut resulting in shorter record.

Then the **feature extraction**.
For the features I used the **spectrogram**, that combines time and frequency informations by plotting the spectral density related to time and frequency as an heatmap.

The color represents the log10 of the spectral density.

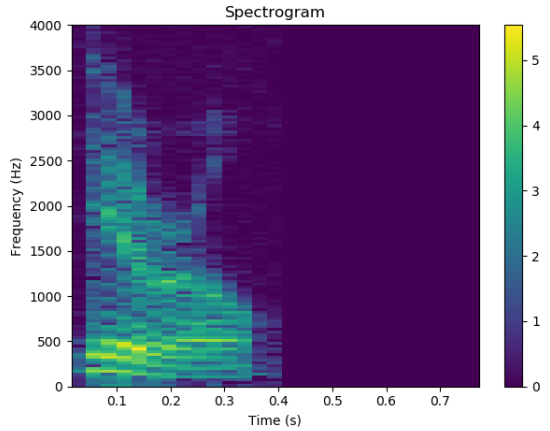


Figure 1 Spectrogram of a "zero"

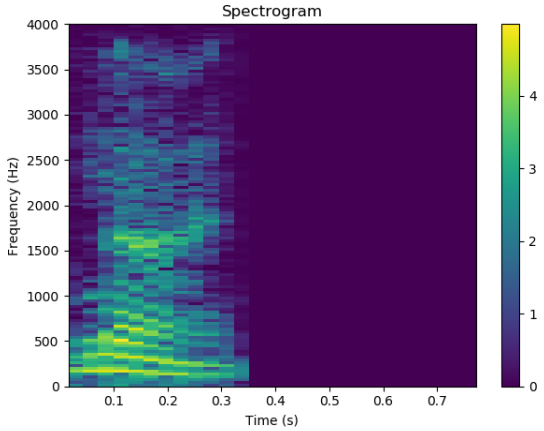


Figure 2 Spectrogram of a "nine"

The maximum frequency is 4kHz because we are sampling at 8kHz and for the Nyquist–Shannon sampling theorem, 4 kHz is the theoretical limit before aliasing.
The trailing part of the plot is often zero due to zero padding.

```
def get_spectrogram(signals):
    spec = []
    eps = 1
    for sig in signals:
        # compute spectrogram
        frequencies, times, s = signal.spectrogram(sig, SAMPLE_RATE, scaling='spectrum')
        spec.append(np.log10(s + eps))  # I need to sum an epsilon to avoid log10(0) = -Inf

        #plt.pcolormesh(times, frequencies, np.log10(s+eps))
        #plt.colorbar()
        #plt.xlabel("Time (s)")
        #plt.ylabel("Frequency (Hz)")
        #plt.title("Spectrogram")
        #plt.tight_layout()
        #plt.show()
    return spec
```

I needed to add an epsilon to the log10 computation to avoid negative infinite values. (even very small numbers work out but using 1, the min value is zero)

Now that we have the feature matrix we can feed it to a Convolutional Neural Network, useful when dealing with matrices.
(The detailed implementation is commented in the appendix)

Holdout with 0.8 ratio is used to split known records into train and test data.

This method gives good results but the prediction errors are often caused by overfitting of the network, that reaches 100% **f1_score** on training data after few epochs with worse results on test data.

This effect can be compensated by using a Random Forest of Neural Networks intead of just one. The implementation is similar to the one used in laboratory .

Firlsty I build each neural network, specifying the input shape (spectrogram shape) and output size (10 in this case)

```
def build(self):
    self.networks = [spectrogramCNN.ConvModel(input_shape=self.input_shape,
                     output_size=self.output_size) for _ in range(self.n_networks)]  # init models
    for network in self.networks:
        network.build()  # build the models
```

Then for the **fit** function we need to build the training set for each NN.
To do that we extract with replacement N records from the entire training set, in this case N coincides with the total amount of training records.
We repeat this process for each NN and then we call the fit function specifying also the validation set.

```
def fit(self, X_train, X_test, y_train, y_test):
    for i, model in enumerate(self.networks):  # for each model
        print(str(i) + " / " + str(len(self.networks)))

        # extraction with replacement for training data
        indexes = np.array(np.random.choice(range(len(X_train)), len(X_train)))
        x_tr = np.array(X_train)[indexes.astype(int)]  # fancy indexing *_*
        y_tr = np.array(y_train)[indexes.astype(int)]

        # fit the models with train data, test data is only used for validation
        model.fit(x_tr, X_test, y_tr, y_test, 100)
```

The **prediction** is done by applying the predict function of each neural network for each evaluation record and chosing the final prediction based on **majority**.

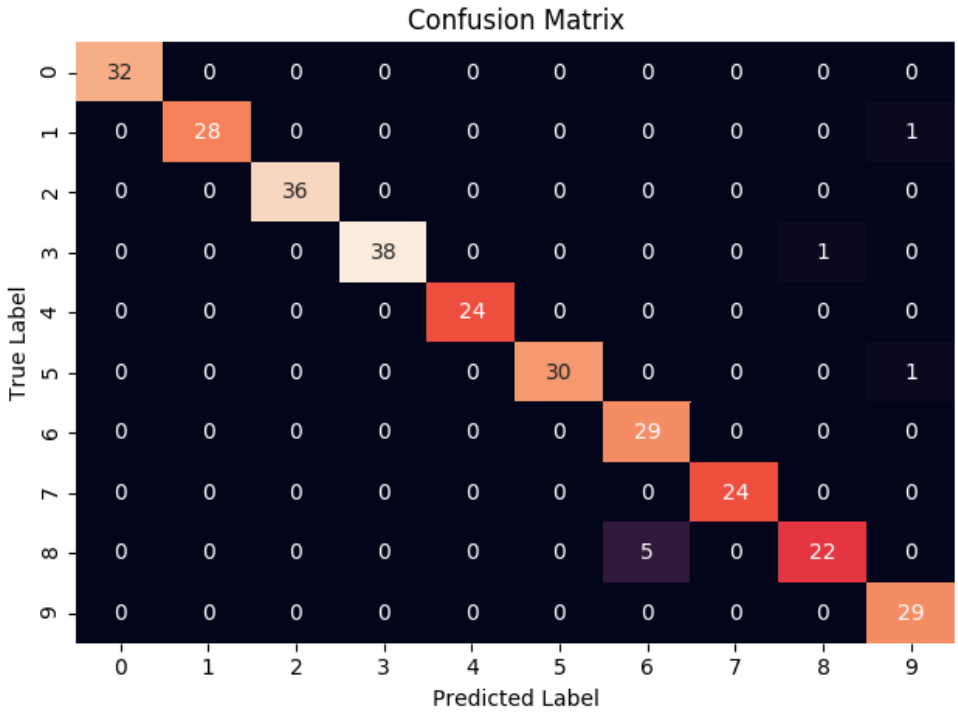
```
def predict(self, X):
    final_preds = []
    total_preds = []
    for model in self.networks:  # for each network of the forest
        labels = model.predict(X)  # predict the labels
        total_preds.append(labels)

    for i in range(len(total_preds[0])):  # for each record prediction
        count = [0] * self.output_size
        for pred in np.array(total_preds)[:, i]:  # choose the most frequent one
            count[int(pred)] += 1
        final_preds.append(np.argmax(count))
    return final_preds
```

In that way every model may overfit the own subset of training record but not the whole dataset.
We can see the results by plotting the **confusion matrix** on test data (20%):

```
confusion_matrix(np.array(y_test, dtype=int), model.predict(X_test))
sns.heatmap(m, annot=True, cbar=False)
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.title('Confusion Matrix')
plt.show()
```

This can be done with *matplotlib* and *seaborn* libraries.



As we can see, the model performs very well on test data.
The most common error is predicting 6 instead of 8 that is reflected on the recall score of label 8 and on the precision score of label 6.

Appendix

spectrogramCNN class

The used CNN is built with *Keras* library, that allows us to design neural networks by layers. In this case the CNN is built by pipelining convolution, pooling and dropout layers three times, followed by a dense layer with dropout and a dense softmax layer for final classification.

```
def build(self):

    self.model = Sequential()
    self.model.add( Conv2D(32, kernel_size=(2, 3), activation='relu', padding='same', strides=1,
                    input_shape=self.input_shape))
    self.model.add(MaxPooling2D((2, 2), strides=(2, 2), padding='same'))
    self.model.add(Dropout(0.2))

    self.model.add(Conv2D(32, kernel_size=(2, 3), activation='relu', padding='same', strides=1))
    self.model.add(MaxPooling2D((2, 2), strides=(2, 2), padding='same'))
    self.model.add(Dropout(0.2))

    self.model.add(Conv2D(32, kernel_size=(2, 3), activation='relu', padding='same', strides=1))
    self.model.add(MaxPooling2D((2, 2), strides=(2, 2), padding='same'))
    self.model.add(Dropout(0.2))

    self.model.add(Flatten())
    self.model.add(Dense(64, activation='relu', name='dense'))
    self.model.add(Dropout(0.2))

    self.model.add(Dense(self.output_size, activation='softmax'))
    self.model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=[f1])
    self.model.summary()
```

The Conv2D kernel size is not square due to the fact that the spectrogram had dimension (129, 28)
Dropout is used to prevent overfitting while **pooling** is applied on a 2x2 pool to avoid reducing dimensions too much.
The used metric is **f1_score**, the same used in evaluation.

The **fit function**

```
def fit(self, X_train, X_test, y_train, y_test, epochs):
    x_tr = np.array(X_train).reshape(np.array(X_train).shape + (1,))
    x_ts = np.array(X_test).reshape(np.array(X_test).shape + (1,))

    # i need to convert labels to one-hot labels (all zeros except for the label
    # for instance: 5 --> 0000010000
    y_tr = to_categorical(np.array(y_train))
    y_ts = to_categorical(np.array(y_test))

    self.model.fit(x_tr, y_tr, validation_data=(x_ts, y_ts), epochs=epochs, verbose=False)
```

Here we need to reshape the input data, to be suitable for the CNN fit function.
to_categorical is a *keras.utils* function used to encode labels to one-hot.

The **predict function**:

```
def predict(self, X):
    prediction = self.model.predict(np.array(X).reshape(np.array(X).shape + (1,)))
    return [np.argmax(x) for x in prediction]  # conversion from one-hot encoding to single label
```

argmax is used to decode one-hot in this case.

Others

Preprocessing functions, for removing silences and uniforming lengths

```
def trim_data(data, threshold):
    trimmed = []
    for record in data:
        trimmed.append((trim(record, top_db=threshold))[0])  # trim function from librosa
    return trimmed


def zero_padding(data, final_len=None):
    padded = []
    if final_len is None:
        final_len = int(np.max([len(x) for x in data]))

    for record in data:
        to_add = final_len - len(record)
        p = np.pad(np.array(record), (0, to_add), constant_values=0)  # append zeros to the end
        padded.append(p)

    return padded
```

Building and training a forest of 10 neural networks with 150k+ params would not have been possible in reasonable time without gpu acceleration, so for completeness I report the piece of code used to setup gpu computation.

```
# CONFIGURATION FOR GPU-ACCELERATION
config = tf.compat.v1.ConfigProto(device_count={'GPU': 1, 'CPU': 6})
config.gpu_options.allow_growth = True
sess = tf.compat.v1.Session(config=config)
tf.compat.v1.keras.backend.set_session(sess)
# _____________________________________
```