Conservatorio di Musica Alfredo Casella

Istituto Superiore di Studi Musicali

BIENNIO DI II LIVELLO IN NUOVE TECNOLOGIE E LINGUAGGI MUSICALI
MUSICA ELETTRONICA

INDIRIZZO REGIA E TECNOLOGIA DEL SUONO

Vowel phonemes Analysis & Classification by means of *OCON rectifiers* Deep Learning Architectures

Appendix A

Python code

RELATORE
Prof. MARCO GIORDANO

CANDIDATO
STEFANO GIACOMELLI

Matr. n°
812/II

ANNO ACCADEMICO 2022/2023

P.le Francesco Savini s.n.c. - 67100 L'Aquila - Tel.: 0862.22122 - Fax: 0862.62325 - e-mail:protocollo@consaq.it

Codice Fiscale 80007670666

REPOSITORIES

Google Drive (*temporary repository*)
Vowel phonemes Analysis & Classification by means of OCON Deep Learning Architectures
https://drive.google.com/drive/folders/1QNY3n0IT1dNtwUUJ0We-vCC64Gm5kXuN?usp=drive_link

GitHub (*permanent repository*)
Vowel phonemes Analysis & Classification by means of OCON Deep Learning Architectures
https://github.com/StefanoGiacomelli/Vowel_phonemes_Analysis_and_Classification_by_means_of_OCON_rectifiers_Deep_Learning_Architectures

GitLab
ASAP repository
https://gitlab.com/stefano.giacomelli/asap/

GOOGLE COLAB *NOTEBOOKS* LIST

Vowel phonemes Analysis & Classification by means of OCON Deep Learning Architectures

Preliminary Analysis

- *HGCW_Dataset_Analysis*.ipynb
- *One-Class_Sub-Network_Analysis*.ipynb

Phoneme Recognition

- *1_OCON_Model_Analysis_(3_features_all_speakers)*.ipynb
- *2_OCON_Model_Analysis_(4_features_all_speakers)*.ipynb
- *3_OCON_Model_Analysis_(3_features_no-children)*.ipynb
- *4_OCON_Model_Analysis_(12_features_all_speakers)*.ipynb

Speaker Recognition

- *5_OCON_Model_Analysis_(13_features_all_speakers)_Speaker_Recognition*.ipynb

Dataset Analysis & Pre-Processing

Author: S. Giacomelli

Year: 2023

Affiliation: A.Casella Conservatory (student)

Master Degree Thesis: "*Vowel phonemes Analysis & Classification by means of OCON rectifiers Deep Learning Architectures*"

Description: Python scripts for HGCW Dataset analysis, features extraction and pre-pocessing

```
# Numerical computations packages/modules
import numpy as np

# Graphic visualization modules
import matplotlib.pyplot as plt
import matplotlib_inline
matplotlib_inline.backend_inline.set_matplotlib_formats('svg')

# Common Seed initialization
SEED = 42  # ... the answer to the ultimate question of Life, the Universe, and Everything... (cit.)
```

HGCW (*Hillenbrand-Getty-Clark-Wheeler*) Dataset

Download link: "Vowel Data" - Western Michigan University (*...no longer maintained*)

References

> 1. L.A. Getty, 1990 - *Acoustic Characteristics of Vowels Produced by Men, Women and and Children*, Master Degree Thesis, Western Michigan University
> 2. J. Hillenbrand, R.T. Gayvert, 1993 - *Vowel Classification Based on Fundamental Frequency and Formant Frequencies*, in Journal of Speech and Hearing Research, vol. 36, pp. 694 - 700
> 3. J. Hillenbrand, L.A. Getty, M.J. Clark, K. Wheeler, 1995 - *Acoustic characteristics of American English vowels*, in The Journal of the Acoustical Society of America, 97, pp. 3099 - 3111

Filenames Structure

1	2-3	4-5	Example
m = man	nn = speaker n° (50 each)	xx = vocal label	m10ae
b = boy	nn = speaker n° (29 each)	xx = vocal label	b11ei
w = woman	nn = speaker n° (50 each)	xx = vocal label	w49ih
g = girl	nn = speaker n° (21 each)	xx = vocal label	g20oo

Audio files features:

- Duration: 1 sec.
- Sample Rate: 16 KHz
- Resolution depth: 16 bit
- File extension: .wav (*wave audio*)

Analysis File Structure

```
**"Formant_Fine_Sampling.csv" Columns**
0) filename
1) duration (in ms)
2) "f0" (fundamental frequency) at steady state
3) "F1" (1st formant frequency) at steady state
4) "F2" (2nd formant frequency) at steady state
5) "F3" (3rd formant frequency) at steady state
6) "F1" at 10% of vowel utterance duration
7) "F2" at 10% of vowel utterance duration
8) "F3" at 10% of vowel utterance duration
18) "F1" at 50% of vowel utterance duration
19) "F2" at 50% of vowel utterance duration
20) "F3" at 50% of vowel utterance duration
27) "F1" at 80% of vowel utterance duration
28) "F2" at 80% of vowel utterance duration
29) "F3" at 80% of vowel utterance duration
```

IMP.: *0s feature values = formant analysis errors*

...for more information (steady state times etc.) see "*Time_Measurements.dat*" and "*Descriptive_Statistics.dat*".

```
# Database (.DAT file) Features Reading (converted to NumPy array)
formant_analysis_data = np.loadtxt("./HGCW_LPC_formants_fine.dat", usecols=(2, 3, 4, 5))
formant_analysis_filenames = np.loadtxt("./HGCW_LPC_formants_fine.dat", usecols=0, dtype=str)

# Useful Parameters
vowels = ['ae', 'ah', 'aw', 'eh', 'er', 'ei', 'ih', 'iy', 'oa', 'oo', 'uh', 'uw']  # Vowels list
colors = ['red', 'saddlebrown', 'darkorange', 'darkgoldenrod', 'gold', 'darkkhaki', 'olive', 'darkgreen', 'steelblue', 'fuchsi
speakers = ['m', 'b', 'w', 'g']  # Speakers list
print(f"Dataset: {formant_analysis_data.shape[0]} samples (for {len(vowels)} labels) & {formant_analysis_data.shape[1]} featur
```

Filtering Functions

- Speaker-based dataset filtering
- Vowel-based dataset filtering
- Null elements dataset filtering

```
# Speaker filter
def speaker_filter(data_array, filenames_array, speaker: str = 'm'):
    """
    Return a list of indices and a filtered data array for a defined speaker string
    """
    assert len(data_array) == len(filenames_array)
    indices = []

    for i in range(len(filenames_array)):  # For each filename...
        if filenames_array[i].lower()[0] == speaker:
            indices.append(i)  # If filename contains speaker sub-string, append actual index to indices array

    return data_array[indices], indices
# ------------------------------------------------------------------------------------------------------------

# Vowels filter
def vowel_filter(data_array, filenames_array, vowel: str = 'ae'):
    """
    Return a list of indices and a filtered data array for a defined vowel string
    """
    assert len(data_array) == len(filenames_array)
    indices = []

    for i in range(len(filenames_array)):  # For each filename...
        if vowel in filenames_array[i].lower()[3: ]:
            indices.append(i)  # If filename contains vowel sub-string, append actual index to indices array

    return data_array[indices], indices
# ------------------------------------------------------------------------------------------------------------

# Null elements filter
def null_filter(data_array, filenames_array):
    """
    Return a list of "null-elements" indices and a filtered data and labels array, without null elements
    """
    assert len(data_array) == len(filenames_array)
    null_indices = []

    for i in range(len(filenames_array)):  # For each filename...
        for j in range(data_array.shape[1]):  # For each feature column...
            if (data_array[i, j] == 0):  # If any formant frequency is null...
                null_indices.append(i)  # Append actual index to indices array

    filtered_filenames = np.delete(filenames_array, null_indices, axis=0)  # Create output deleting null indices from filename
    filtered_data = np.delete(data_array, null_indices, axis=0)  # Create output deleting null indices from data array

    return filtered_data, filtered_filenames, null_indices
```

```
# Remove Null elements
nonnull_data, nonnull_filenames, _ = null_filter(formant_analysis_data, formant_analysis_filenames)
print(f"NON NULL Dataset: {nonnull_data.shape[0]} samples (for {len(vowels)} labels) & {nonnull_data.shape[1]} features each")
print('-------------------------------------------------------------------')
print()

# Outputs initialization
x_data_raw_np = np.zeros((len(nonnull_data), 4), dtype=float)  # Same Database n° of elements, 4 float features (columns)
y_labels_raw_np = np.zeros((len(nonnull_data), 1), dtype=int)  # Same Database n° of elements, integer label single column arr
```

```
# Subgroups extraction & analysis
end_idx = [0]  # Indices list initialization (0 and size values comprised)
vow_size = []  # Vowel groups size list initialization

for vowel_idx, vowel in enumerate(vowels):
    vow_data, _ = vowel_filter(nonnull_data, nonnull_filenames, vowel=vowel)  # Vowel sub-set extraction
    end_idx.append(end_idx[vowel_idx] + len(vow_data))  # Actual sub-group End-Index appending
    vow_size.append(len(vow_data))  # Actual sub-group length appending
    print(f'Vowel "{vowel}" sub-set : {len(vow_data)} samples')

    start_idx = end_idx[vowel_idx]  # Previous sub-set end-index
    print('1st  element Idx   :', start_idx)
    stop_idx = end_idx[vowel_idx] + len(vow_data)  # Actual stop index = previous End + actual Size
    print('Last element Idx   :', stop_idx - 1)
    x_data_raw_np[start_idx: stop_idx, :] = vow_data[:, :]   # Output data sub-set ordered writing (Fundamental, 1st, 2nd & 3r

    vow_labels = np.full((len(vow_data), 1), vowel_idx, dtype=int)  # Actual integer labels array creation
    print(f'Vowel LABEL        : {vowel} - {vowel_idx}')
    y_labels_raw_np[start_idx: stop_idx, :] = vow_labels  # Output labels sub-set ordered writing

    print('--------------------------------')

# Different labels counter
diff_labels = len(np.unique(y_labels_raw_np))

print()
print(f'--> RAW DATASET shape: {x_data_raw_np.shape}, w. {diff_labels} Labels')


# Raw Dataset Plot
plt.figure(figsize=(12, 15))
plt.suptitle('Dataset "2-Features" separation')

for index, vowel in enumerate(vowels):

    first_coords = x_data_raw_np[end_idx[index]: end_idx[index + 1], 1]
    second_coords = x_data_raw_np[end_idx[index]: end_idx[index + 1], 2]
    third_coords = x_data_raw_np[end_idx[index]: end_idx[index + 1], 3]

    plt.subplot(3, 1, 1)
    plt.title('$1_{st}$ VS $2_{nd}$')
    plt.scatter(first_coords, second_coords, marker='o', color=colors[index], label=f'Vowel "{vowel}"')
    plt.xlabel('$1_{st}$ Formant Frequency (in Hz))')
    plt.ylabel('$2_{nd}$ Formant Frequency (in Hz)')
    plt.legend(loc='best')
    plt.grid(True)

    plt.subplot(3, 1, 2)
    plt.title('$1_{st}$ VS $3_{rd}$')
    plt.scatter(first_coords, third_coords, marker='o', color=colors[index], label=f'Vowel "{vowel}"')
    plt.xlabel('$1_{st}$ Formant Frequency (in Hz)')
    plt.ylabel('$3_{rd}$ Formant Frequency (in Hz)')
    plt.legend(loc='best')
    plt.grid(True)

    plt.subplot(3, 1, 3)
    plt.title('$2_{nd}$ VS $3_{rd}$')
    plt.scatter(second_coords, third_coords, marker='o', color=colors[index], label=f'Vowel "{vowel}"')
    plt.xlabel('$2_{nd}$ Formant Frequency (in Hz)')
    plt.ylabel('$3_{rd}$ Formant Frequency (in Hz)')
    plt.legend(loc='best')
    plt.grid(True)

plt.tight_layout()
plt.savefig("raw_dataset_plot")
plt.show()


# Class Occurences Plot (Sample Balancing Analysis)
plt.figure(figsize=(12, 5))
plt.suptitle("Dataset Samples Balance")

for i in range(len(colors)):
    plt.bar(i, vow_size[i], color=colors[i])
    plt.xlabel('Vowels')
    plt.ylabel('Samples')

plt.xticks(ticks=[n for n in range(12)], labels=vowels)

plt.axhline(np.min(vow_size), color='grey', linestyle='--', label=f'Min: {np.min(vow_size)} samples')
plt.axhline(np.max(vow_size), color='red', linestyle='--', label=f'Max: {np.max(vow_size)} samples')
plt.legend(loc='best')
plt.grid()
```

```
plt.savefig('dataset_class_occurences')
plt.show()
```

"Formant to Fundamental" Normalization

Formant frequency ratio

$$formant_{ratio(i)} = \frac{freq_{formant_{(i)}}}{freq_{fund}}$$

```
# Fundamental Frequency (ratio) Normalization
x_data_fund_norm = np.zeros(x_data_raw_np.shape)  # Output initialization

for i in range(x_data_raw_np.shape[1]):  # For each feature...
    if i >= 1: # For each formant column...
        x_data_fund_norm[:, i] = x_data_raw_np[:, i] / x_data_raw_np[:, 0]  # i-Formant value / i-Fundamental value
    else:  # Exception for Fundamental freq column
        x_data_fund_norm[:, i] = x_data_raw_np[:, i]

print(f"'Fundamental Normalized' Dataset: {x_data_fund_norm.shape[0]} elements (w. {diff_labels} labels) & {x_data_fund_norm.s


# Fundamental Normalized dataset Plot
plt.figure(figsize=(12, 15))
plt.suptitle('Raw VS Normalized Datasets\n')

for index, vowel in enumerate(vowels):

    first_coords = x_data_fund_norm[end_idx[index]: end_idx[index + 1], 1]
    second_coords = x_data_fund_norm[end_idx[index]: end_idx[index + 1], 2]
    third_coords = x_data_fund_norm[end_idx[index]: end_idx[index + 1], 3]

    plt.subplot(3, 2, 2)
    plt.title('Fund. Normalized $1_{st}$ VS $2_{nd}$')
    plt.scatter(first_coords, second_coords, marker='o', color=colors[index], label=f'Vowel "{vowel}"')
    plt.xlabel('$1_{st}$ Formant Ratio')
    plt.ylabel('$2_{nd}$ Formant Ratio')
    plt.legend(loc='best')
    plt.grid(True)

    plt.subplot(3, 2, 4)
    plt.title('Fund. Normalized $1_{st}$ VS $3_{rd}$')
    plt.scatter(first_coords, third_coords, marker='o', color=colors[index], label=f'Vowel "{vowel}"')
    plt.xlabel('$1_{st}$ Formant Ratio')
    plt.ylabel('$3_{rd}$ Formant Ratio')
    plt.legend(loc='best')
    plt.grid(True)

    plt.subplot(3, 2, 6)
    plt.title('Fund. Normalized $2_{nd}$ VS $3_{rd}$')
    plt.scatter(second_coords, third_coords, marker='o', color=colors[index], label=f'Vowel "{vowel}"')
    plt.xlabel('$2_{nd}$ Formant Ratio')
    plt.ylabel('$3_{rd}$ FFormant Ratio')
    plt.legend(loc='best')
    plt.grid(True)

for index, vowel in enumerate(vowels):

    first_coords = x_data_raw_np[end_idx[index]: end_idx[index + 1], 1]
    second_coords = x_data_raw_np[end_idx[index]: end_idx[index + 1], 2]
    third_coords = x_data_raw_np[end_idx[index]: end_idx[index + 1], 3]

    plt.subplot(3, 2, 1)
    plt.title('Raw $1_{st}$ VS $2_{nd}$')
    plt.scatter(first_coords, second_coords, marker='o', color=colors[index], label=f'Vowel "{vowel}"')
    plt.xlabel('$1_{st}$ Formant Frequency (in Hz))')
    plt.ylabel('$2_{nd}$ Formant Frequency (in Hz)')
    plt.legend(loc='best')
    plt.grid(True)

    plt.subplot(3, 2, 3)
    plt.title('Raw $1_{st}$ VS $3_{rd}$')
    plt.scatter(first_coords, third_coords, marker='o', color=colors[index], label=f'Vowel "{vowel}"')
    plt.xlabel('$1_{st}$ Formant Frequency (in Hz)')
    plt.ylabel('$3_{rd}$ Formant Frequency (in Hz)')
    plt.legend(loc='best')
    plt.grid(True)

    plt.subplot(3, 2, 5)
    plt.title('Raw $2_{nd}$ VS $3_{rd}$')
```

```
    plt.scatter(second_coords, third_coords, marker='o', color=colors[index], label=f'Vowel "{vowel}"')
    plt.xlabel('$2_{nd}$ Formant Frequency (in Hz)')
    plt.ylabel('$3_{rd}$ Formant Frequency (in Hz)')
    plt.legend(loc='best')
    plt.grid(True)

plt.tight_layout()
plt.savefig("raw_vs_fund_norm_datasets_plot")
plt.show()
```

Min-Max Scaling

$$\hat{x} = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

and eventually

$$x^* = a + \hat{x}(b - a)$$

with a & b respectively, lower and upper bounds of the destination range.

IMP.: in this case Min-Max scaling is commonly applied to entire features space *(are measured on the same axis and represents samples of same spectral domain properties)*

```
a = 0.  # Lower bound
b = 1.  # Upper bound
x_data_minmax = np.zeros((x_data_fund_norm.shape))
print(f'Fundamental Ratios  STATS:   Min. = {x_data_fund_norm[:, 1:].min()}    Max. = {x_data_fund_norm[:, 1:].max()}')

x_data_minmax[:, 1:] = a + ((x_data_fund_norm[:, 1:] - x_data_fund_norm[:, 1:].min()) / (x_data_fund_norm[:, 1:].max() - x_dat
x_data_minmax[:, 0] = x_data_fund_norm[:, 0]  # Fundamental column exception

print(f'Min-Max Ratios      STATS:   Min. = {x_data_minmax[:, 1:].min()}                   Max. = {x_data_minmax[:, 1:].max()}
print('-------------------------------------------------------------------')
print(f"'Fundamental & Min-Max Normalized' Dataset: {x_data_minmax.shape[0]} elements (w. {diff_labels} labels) & {x_data_minm
```

Statistical Analysis

- Features **P**robability **M**ass **D**istribution (on the entire Dataset)
- Feaatures **P**robability **M**ass **D**istribution (for each class sub-set)

```
# Dataset Plot
dataset = x_data_minmax # x_data_fund_norm

plt.figure(figsize=(12, 15))
plt.suptitle('Normalized Dataset "2-Features" separation')

for index, vowel in enumerate(vowels):

    first_coords = dataset[end_idx[index]: end_idx[index + 1], 1]
    second_coords = dataset[end_idx[index]: end_idx[index + 1], 2]
    third_coords = dataset[end_idx[index]: end_idx[index + 1], 3]

    plt.subplot(3, 1, 1)
    plt.title('$1_{st}$ VS $2_{nd}$')
    plt.scatter(first_coords, second_coords, marker='o', color=colors[index], label=f'Vowel "{vowel}"')
    plt.xlabel('$1_{st}$ Formant Frequency ratio')
    plt.ylabel('$2_{nd}$ Formant Frequency ratio')
    plt.legend(loc='best')
    plt.grid(True)

    plt.subplot(3, 1, 2)
    plt.title('$1_{st}$ VS $3_{rd}$')
    plt.scatter(first_coords, third_coords, marker='o', color=colors[index], label=f'Vowel "{vowel}"')
    plt.xlabel('$1_{st}$ Formant Frequency ratio')
    plt.ylabel('$3_{rd}$ Formant Frequency ratio')
    plt.legend(loc='best')
    plt.grid(True)

    plt.subplot(3, 1, 3)
    plt.title('$2_{nd}$ VS $3_{rd}$')
    plt.scatter(second_coords, third_coords, marker='o', color=colors[index], label=f'Vowel "{vowel}"')
    plt.xlabel('$2_{nd}$ Formant Frequency ratio')
    plt.ylabel('$3_{rd}$ Formant Frequency ratio')
    plt.legend(loc='best')
    plt.grid(True)

plt.tight_layout()
plt.savefig("normalized_dataset_plot")
```

```
plt.show()


# Formants Probability Distribution plot
plt.figure(figsize=(12, 5))
plt.suptitle('Features Probability Mass Distribution\n(entire Dataset)')

for i in range(dataset.shape[1] - 1):
    plt.subplot(1, 3, i + 1)
    plt.hist(dataset[:, i + 1], bins=30, rwidth=0.9)
    plt.title(f'PMD $Formant_{i + 1}$ ratio')
    plt.xlabel('Normalized Formant Ratio')
    plt.ylabel('Frequency (occurrences)')
    plt.ylim([0, 350])
    plt.grid()

plt.tight_layout()
plt.savefig("normalized_dataset_stats")
plt.show()
```

Examinating features data distribution (per class), we evaluate *Z-Scoring* (standardization) usefulness/availability.

```
# Probability distribution (for each feature, in each class)
plt.figure(figsize=(12, 50))

for i in range(len(vowels)):
    vow_group = dataset[end_idx[i]: end_idx[i + 1], :]  # Vowel group Extraction
    for n in range(vow_group.shape[1] - 1):
        plt.subplot(12, 3, (n + 1) + i * 3)
        plt.hist(vow_group[:, n + 1], bins=30, rwidth=0.9, color=colors[i])
        plt.title(f'"{vowels[i]}" Group, $Formant_{n + 1}$ ratio')
        plt.xlabel('Normalized Formant Ratio')
        plt.ylabel('Frequency')
        plt.ylim([0, 50])
        plt.grid()

plt.tight_layout()
plt.savefig('normalized_dataset_stats_(per_class)')
plt.show()
```

OUTPUT Datasets

Reference: NPZ - NumPy Binary File Compression

```
# HGCW Dataset: 3 Formants (steady state) + Fundamental Normalization + MinMax Scaling
classes_size = np.array(vow_size)  # Phoneme classes sizes array
classes_indices = np.array(end_idx)  # Phoneme classes indices (start/end included)
np.savez_compressed(file='./HGCW_dataset_utils',
                    HGCW_raw = x_data_raw_np,
                    HGCW_fund_norm = x_data_fund_norm,
                    HGCW_minmax = x_data_minmax,
                    HGCW_labels = y_labels_raw_np,
                    classes_size = classes_size,
                    classes_idx = classes_indices)


# HGCW MEN Dataset: 3 Formants (steady state) + Fundamental Normalization + MinMax Scaling
import numpy as np

# Database (.DAT file) Features Reading (converted to NumPy array)
formant_analysis_data = np.loadtxt("./HGCW_LPC_formants_fine.dat", usecols=(2, 3, 4, 5))
formant_analysis_filenames = np.loadtxt("./HGCW_LPC_formants_fine.dat", usecols=0, dtype=str)

# Useful Parameters
vowels = ['ae', 'ah', 'aw', 'eh', 'er', 'ei', 'ih', 'iy', 'oa', 'oo', 'uh', 'uw']  # Vowels list
speakers = ['m', 'b', 'w', 'g']  # Speakers list
print(f"Dataset: {formant_analysis_data.shape[0]} samples (for {len(vowels)} labels) & {formant_analysis_data.shape[1]} featur
# ------------------------------------------------------------------------------------------------------------

# Gender Filtering (Male)
male_data, male_indices = speaker_filter(formant_analysis_data, formant_analysis_filenames, speaker = 'm')
male_filenames = formant_analysis_filenames[male_indices]

# Remove Null elements
nonnull_data, nonnull_filenames, _ = null_filter(male_data, male_filenames)
print(f"NON NULL Dataset: {nonnull_data.shape[0]} samples (for {len(vowels)} labels) & {nonnull_data.shape[1]} features each")
print('--------------------------------------------------------------------')
print()

# Outputs initialization
```

```
x_data_raw_np = np.zeros((len(nonnull_data), 4), dtype=float)  # Same Database n° of elements, 4 float features (columns)
y_labels_raw_np = np.zeros((len(nonnull_data), 1), dtype=int)  # Same Database n° of elements, integer label single column arr

# Subgroups extraction & analysis
end_idx = [0]  # Indices list initialization (0 and size values comprised)
vow_size = []  # Vowel groups size list initialization

for vowel_idx, vowel in enumerate(vowels):
    vow_data, _ = vowel_filter(nonnull_data, nonnull_filenames, vowel=vowel)  # Vowel sub-set extraction
    end_idx.append(end_idx[vowel_idx] + len(vow_data))  # Actual sub-group End-Index appending
    vow_size.append(len(vow_data))  # Actual sub-group length appending
    print(f'Vowel "{vowel}" sub-set : {len(vow_data)} samples')

    start_idx = end_idx[vowel_idx]  # Previous sub-set end-index
    print('1st  element Idx   :', start_idx)
    stop_idx = end_idx[vowel_idx] + len(vow_data)  # Actual stop index = previous End + actual Size
    print('Last element Idx   :', stop_idx - 1)
    x_data_raw_np[start_idx: stop_idx, :] = vow_data[:, :]   # Output data sub-set ordered writing (Fundamental, 1st, 2nd & 3r

    vow_labels = np.full((len(vow_data), 1), vowel_idx, dtype=int)  # Actual integer labels array creation
    print(f'Vowel LABEL        : {vowel} - {vowel_idx}')
    y_labels_raw_np[start_idx: stop_idx, :] = vow_labels  # Output labels sub-set ordered writing

    print('-------------------------------')

# Different labels counter
diff_labels = len(np.unique(y_labels_raw_np))
print()
print(f'--> RAW DATASET shape: {x_data_raw_np.shape}, w. {diff_labels} Labels')
print('---------------------------------------------------------------')
# -----------------------------------------------------------------------------------------------------------

# Fundamental Frequency (ratio) Normalization
x_data_fund_norm = np.zeros(x_data_raw_np.shape)  # Output initialization

for i in range(x_data_raw_np.shape[1]):  # For each feature...
    if i >= 1: # For each formant column...
        x_data_fund_norm[:, i] = x_data_raw_np[:, i] / x_data_raw_np[:, 0]  # i-Formant value / i-Fundamental value
    else:  # Exception for Fundamental freq column
        x_data_fund_norm[:, i] = x_data_raw_np[:, i]

print(f"'Fundamental Normalized' Dataset: {x_data_fund_norm.shape[0]} elements (w. {diff_labels} labels) & {x_data_fund_norm.s
print('---------------------------------------------------------------')
# -----------------------------------------------------------------------------------------------------------

a = 0.  # Lower bound
b = 1.  # Upper bound
x_data_minmax = np.zeros((x_data_fund_norm.shape))
print(f'Fundamental Ratios  STATS:   Min. = {x_data_fund_norm[:, 1:].min()}    Max. = {x_data_fund_norm[:, 1:].max()}')

x_data_minmax[:, 1:] = a + ((x_data_fund_norm[:, 1:] - x_data_fund_norm[:, 1:].min()) / (x_data_fund_norm[:, 1:].max() - x_dat
x_data_minmax[:, 0] = x_data_fund_norm[:, 0]  # Fundamental column exception

print(f'Min-Max Ratios      STATS:   Min. = {x_data_minmax[:, 1:].min()}                   Max. = {x_data_minmax[:, 1:].max()}
print('-------------------------------------------------------------------')
print(f"'Fundamental & Min-Max Normalized' Dataset: {x_data_minmax.shape[0]} elements (w. {diff_labels} labels) & {x_data_minm
# -----------------------------------------------------------------------------------------------------------

# Output Store
classes_size = np.array(vow_size)  # Phoneme classes sizes array
classes_indices = np.array(end_idx)  # Phoneme classes indices (start/end included)
np.savez_compressed(file='./HGCW_dataset_utils',
                    HGCW_raw = x_data_raw_np,
                    HGCW_fund_norm = x_data_fund_norm,
                    HGCW_minmax = x_data_minmax,
                    HGCW_labels = y_labels_raw_np,
                    classes_size = classes_size,
                    classes_idx = classes_indices)


# HGCW WOMEN Dataset: 3 Formants (steady state) + Fundamental Normalization + MinMax Scaling
import numpy as np

# Database (.DAT file) Features Reading (converted to NumPy array)
formant_analysis_data = np.loadtxt("./HGCW_LPC_formants_fine.dat", usecols=(2, 3, 4, 5))
formant_analysis_filenames = np.loadtxt("./HGCW_LPC_formants_fine.dat", usecols=0, dtype=str)

# Useful Parameters
vowels = ['ae', 'ah', 'aw', 'eh', 'er', 'ei', 'ih', 'iy', 'oa', 'oo', 'uh', 'uw']  # Vowels list
speakers = ['m', 'b', 'w', 'g']  # Speakers list
print(f"Dataset: {formant_analysis_data.shape[0]} samples (for {len(vowels)} labels) & {formant_analysis_data.shape[1]} featur
# -----------------------------------------------------------------------------------------------------------
```

```
# Gender Filtering (Male)
male_data, male_indices = speaker_filter(formant_analysis_data, formant_analysis_filenames, speaker = 'w')
male_filenames = formant_analysis_filenames[male_indices]

# Remove Null elements
nonnull_data, nonnull_filenames, _ = null_filter(male_data, male_filenames)
print(f"NON NULL Dataset: {nonnull_data.shape[0]} samples (for {len(vowels)} labels) & {nonnull_data.shape[1]} features each")
print('------------------------------------------------------------------')
print()

# Outputs initialization
x_data_raw_np = np.zeros((len(nonnull_data), 4), dtype=float)  # Same Database n° of elements, 4 float features (columns)
y_labels_raw_np = np.zeros((len(nonnull_data), 1), dtype=int)  # Same Database n° of elements, integer label single column arr

# Subgroups extraction & analysis
end_idx = [0]  # Indices list initialization (0 and size values comprised)
vow_size = []  # Vowel groups size list initialization

for vowel_idx, vowel in enumerate(vowels):
    vow_data, _ = vowel_filter(nonnull_data, nonnull_filenames, vowel=vowel)  # Vowel sub-set extraction
    end_idx.append(end_idx[vowel_idx] + len(vow_data))  # Actual sub-group End-Index appending
    vow_size.append(len(vow_data))  # Actual sub-group length appending
    print(f'Vowel "{vowel}" sub-set : {len(vow_data)} samples')

    start_idx = end_idx[vowel_idx]  # Previous sub-set end-index
    print('1st  element Idx   :', start_idx)
    stop_idx = end_idx[vowel_idx] + len(vow_data)  # Actual stop index = previous End + actual Size
    print('Last element Idx   :', stop_idx - 1)
    x_data_raw_np[start_idx: stop_idx, :] = vow_data[:, :]   # Output data sub-set ordered writing (Fundamental, 1st, 2nd & 3r

    vow_labels = np.full((len(vow_data), 1), vowel_idx, dtype=int)  # Actual integer labels array creation
    print(f'Vowel LABEL        : {vowel} - {vowel_idx}')
    y_labels_raw_np[start_idx: stop_idx, :] = vow_labels  # Output labels sub-set ordered writing

    print('--------------------------------')

# Different labels counter
diff_labels = len(np.unique(y_labels_raw_np))
print()
print(f'--> RAW DATASET shape: {x_data_raw_np.shape}, w. {diff_labels} Labels')
print('--------------------------------------------------------------')
# ------------------------------------------------------------------------------------------------------

# Fundamental Frequency (ratio) Normalization
x_data_fund_norm = np.zeros(x_data_raw_np.shape)  # Output initialization

for i in range(x_data_raw_np.shape[1]):  # For each feature...
    if i >= 1: # For each formant column...
        x_data_fund_norm[:, i] = x_data_raw_np[:, i] / x_data_raw_np[:, 0]  # i-Formant value / i-Fundamental value
    else:  # Exception for Fundamental freq column
        x_data_fund_norm[:, i] = x_data_raw_np[:, i]

print(f"'Fundamental Normalized' Dataset: {x_data_fund_norm.shape[0]} elements (w. {diff_labels} labels) & {x_data_fund_norm.s
print('--------------------------------------------------------------')
# ------------------------------------------------------------------------------------------------------

a = 0.  # Lower bound
b = 1.  # Upper bound
x_data_minmax = np.zeros((x_data_fund_norm.shape))
print(f'Fundamental Ratios  STATS:   Min. = {x_data_fund_norm[:, 1:].min()}    Max. = {x_data_fund_norm[:, 1:].max()}')

x_data_minmax[:, 1:] = a + ((x_data_fund_norm[:, 1:] - x_data_fund_norm[:, 1:].min()) / (x_data_fund_norm[:, 1:].max() - x_dat
x_data_minmax[:, 0] = x_data_fund_norm[:, 0]  # Fundamental column exception

print(f'Min-Max Ratios      STATS:   Min. = {x_data_minmax[:, 1:].min()}                   Max. = {x_data_minmax[:, 1:].max()}
print('------------------------------------------------------------------')
print(f"'Fundamental & Min-Max Normalized' Dataset: {x_data_minmax.shape[0]} elements (w. {diff_labels} labels) & {x_data_minm
# ------------------------------------------------------------------------------------------------------

# Output Store
classes_size = np.array(vow_size)  # Phoneme classes sizes array
classes_indices = np.array(end_idx)  # Phoneme classes indices (start/end included)
np.savez_compressed(file='./HGCW_dataset_utils',
                    HGCW_raw = x_data_raw_np,
                    HGCW_fund_norm = x_data_fund_norm,
                    HGCW_minmax = x_data_minmax,
                    HGCW_labels = y_labels_raw_np,
                    classes_size = classes_size,
                    classes_idx = classes_indices)


# HGCW Dataset (3 x 4 Formants: 3- steady state, 3 - 10%, 3 - 50%, 3 - 80%) + Transform
```

```
# Database (.DAT file) Features Reading (converted to NumPy array)
import numpy as np

formant_analysis_data = np.loadtxt("./HGCW_LPC_formants_fine.dat", usecols=(2, 3, 4, 5, 6, 7, 8, 18, 19, 20, 27, 28, 29))
formant_analysis_filenames = np.loadtxt("./HGCW_LPC_formants_fine.dat", usecols=0, dtype=str)
# Useful Parameters
vowels = ['ae', 'ah', 'aw', 'eh', 'er', 'ei', 'ih', 'iy', 'oa', 'oo', 'uh', 'uw']  # Vowels list
speakers = ['m', 'b', 'w', 'g']  # Speakers list
print(f"Dataset: {formant_analysis_data.shape[0]} samples (for {len(vowels)} labels) & {formant_analysis_data.shape[1]} featur
# -----------------------------------------------------------------------------------------------------------

# Remove Null elements
nonnull_data, nonnull_filenames, _ = null_filter(formant_analysis_data, formant_analysis_filenames)
print(f"NON NULL Dataset: {nonnull_data.shape[0]} samples (for {len(vowels)} labels) & {nonnull_data.shape[1]} features each")
print('-----------------------------------------------------------------------')
print()

# Outputs initialization
x_data_raw_np = np.zeros((len(nonnull_data), 13), dtype=float)  # Same Database n° of elements, fund + 12 formants features (c
y_labels_raw_np = np.zeros((len(nonnull_data), 1), dtype=int)  # Same Database n° of elements, integer label single column arr

# Subgroups extraction & analysis
end_idx = [0]  # Indices list initialization (0 and size values comprised)
vow_size = []  # Vowel groups size list initialization

for vowel_idx, vowel in enumerate(vowels):
    vow_data, _ = vowel_filter(nonnull_data, nonnull_filenames, vowel=vowel)  # Vowel sub-set extraction
    end_idx.append(end_idx[vowel_idx] + len(vow_data))  # Actual sub-group End-Index appending
    vow_size.append(len(vow_data))  # Actual sub-group length appending
    print(f'Vowel "{vowel}" sub-set : {len(vow_data)} samples')

    start_idx = end_idx[vowel_idx]  # Previous sub-set end-index
    print('1st  element Idx   :', start_idx)
    stop_idx = end_idx[vowel_idx] + len(vow_data)  # Actual stop index = previous End + actual Size
    print('Last element Idx   :', stop_idx - 1)
    x_data_raw_np[start_idx: stop_idx, :] = vow_data[:, :]   # Output data sub-set ordered writing (Fundamental, 1st, 2nd & 3r

    vow_labels = np.full((len(vow_data), 1), vowel_idx, dtype=int)  # Actual integer labels array creation
    print(f'Vowel LABEL        : {vowel} - {vowel_idx}')
    y_labels_raw_np[start_idx: stop_idx, :] = vow_labels  # Output labels sub-set ordered writing

    print('--------------------------------')

# Different labels counter
diff_labels = len(np.unique(y_labels_raw_np))
print()
print(f'--> RAW DATASET shape: {x_data_raw_np.shape}, w. {diff_labels} Labels')
print('-----------------------------------------------------------------')
# -----------------------------------------------------------------------------------------------------------

# Fundamental Frequency (ratio) Normalization
x_data_fund_norm = np.zeros(x_data_raw_np.shape)  # Output initialization

for i in range(x_data_raw_np.shape[1]):  # For each feature...
    if i >= 1: # For each formant column...
        x_data_fund_norm[:, i] = x_data_raw_np[:, i] / x_data_raw_np[:, 0]  # i-Formant value / i-Fundamental value
    else:  # Exception for Fundamental freq column
        x_data_fund_norm[:, i] = x_data_raw_np[:, i]

print(f"'Fundamental Normalized' Dataset: {x_data_fund_norm.shape[0]} elements (w. {diff_labels} labels) & {x_data_fund_norm.s
print('-----------------------------------------------------------------')
# -----------------------------------------------------------------------------------------------------------

a = 0.  # Lower bound
b = 1.  # Upper bound
x_data_minmax = np.zeros((x_data_fund_norm.shape))
print(f'Fundamental Ratios  STATS:   Min. = {x_data_fund_norm[:, 1:].min()}    Max. = {x_data_fund_norm[:, 1:].max()}')

x_data_minmax[:, 1:] = a + ((x_data_fund_norm[:, 1:] - x_data_fund_norm[:, 1:].min()) / (x_data_fund_norm[:, 1:].max() - x_dat
x_data_minmax[:, 0] = x_data_fund_norm[:, 0]  # Fundamental column exception

print(f'Min-Max Ratios      STATS:   Min. = {x_data_minmax[:, 1:].min()}                   Max. = {x_data_minmax[:, 1:].max()}
print('-----------------------------------------------------------------------')
print(f"'Fundamental & Min-Max Normalized' Dataset: {x_data_minmax.shape[0]} elements (w. {diff_labels} labels) & {x_data_minm
# -----------------------------------------------------------------------------------------------------------

# Output Store
classes_size = np.array(vow_size)  # Phoneme classes sizes array
classes_indices = np.array(end_idx)  # Phoneme classes indices (start/end included)
np.savez_compressed(file='./HGCW_dataset_utils',
                    HGCW_raw = x_data_raw_np,
                    HGCW_fund_norm = x_data_fund_norm,
```

```
                    HGCW_minmax = x_data_minmax,
                    HGCW_labels = y_labels_raw_np,
                    classes_size = classes_size,
                    classes_idx = classes_indices)


# HGCW Dataset (3 x 4 Formants: 3- steady state, 3 - 10%, 3 - 50%, 3 - 80%) + SPEAKER Label

# Database (.DAT file) Features Reading (converted to NumPy array)
import numpy as np

formant_analysis_data = np.loadtxt("./HGCW_LPC_formants_fine.dat", usecols=(2, 3, 4, 5, 6, 7, 8, 18, 19, 20, 27, 28, 29))
formant_analysis_filenames = np.loadtxt("./HGCW_LPC_formants_fine.dat", usecols=0, dtype=str)
# Useful Parameters
vowels = ['ae', 'ah', 'aw', 'eh', 'er', 'ei', 'ih', 'iy', 'oa', 'oo', 'uh', 'uw']  # Vowels list
speakers = ['b', 'g', 'm', 'w']  # Speakers list
print(f"Dataset: {formant_analysis_data.shape[0]} samples (for {len(vowels)} labels) & {formant_analysis_data.shape[1]} featur
# -------------------------------------------------------------------------------------------------------------

# Remove Null elements
nonnull_data, nonnull_filenames, _ = null_filter(formant_analysis_data, formant_analysis_filenames)
print(f"NON NULL Dataset: {nonnull_data.shape[0]} samples (for {len(vowels)} labels) & {nonnull_data.shape[1]} features each")
print('-------------------------------------------------------------------------')
print()

# Outputs initialization
x_data_raw_np = np.zeros((len(nonnull_data), 13), dtype=float)  # Same Database n° of elements, fund + 12 formants features (c
y_labels_raw_np = np.zeros((len(nonnull_data), 1), dtype=int)  # Same Database n° of elements, integer phoneme labels single c
z_labels_raw_np = np.zeros((len(nonnull_data), 1), dtype=int)  # Same Database n° of elements, integer speaker labels single c

# Subgroups extraction & analysis
end_idx = [0]  # Indices list initialization (0 value comprised)
vow_size = []  # Vowel groups size list initialization
spk_coords = []  # Will be a list of 12 lists: each sub-list will contain 4 speaker tuples (start_idx, speaker-phoneme size)


for vowel_idx, vowel in enumerate(vowels):
    vow_data, vow_indices = vowel_filter(nonnull_data, nonnull_filenames, vowel=vowel)  # Vowel sub-set extraction
    end_idx.append(end_idx[vowel_idx] + len(vow_data))  # Actual sub-group End-Index appending
    vow_size.append(len(vow_data))  # Actual sub-group length appending
    print(f'Vowel "{vowel}" sub-set : {len(vow_data)} samples')

    start_idx = end_idx[vowel_idx]  # Previous sub-set end-index
    print('1st  element Idx   :', start_idx)
    stop_idx = end_idx[vowel_idx] + len(vow_data)  # Actual stop index = previous End + actual Size
    print('Last element Idx   :', stop_idx - 1)
    x_data_raw_np[start_idx: stop_idx, :] = vow_data[:, :]   # Output data sub-set ordered writing (Fundamental, 1st, 2nd & 3r

    vow_labels = np.full((len(vow_data), 1), vowel_idx, dtype=int)  # Actual integer labels array creation
    print(f'Vowel LABEL        : {vowel} - {vowel_idx}')
    y_labels_raw_np[start_idx: stop_idx, :] = vow_labels  # Output labels sub-set ordered writing
    print()
    # ---------------------------------------------------------------------------------------------

    # Subset speaker snalysis
    vow_data_spk = np.zeros((len(vow_data), 1), dtype=int)
    vow_data_idx = []

    for speaker_idx, speaker in enumerate(speakers):
        _, spk_indices = speaker_filter(vow_data, nonnull_filenames[vow_indices], speaker=speaker)  # Extract n-speaker indice
        vow_data_idx.append((spk_indices[0], len(spk_indices)))
        vow_data_spk[spk_indices] = speaker_idx  # Set actual speaker label to actual vowel-speaker array
        print(f'"{speaker.upper()}"-speakers       : {len(spk_indices)} (w. label "{speaker_idx}")')

    z_labels_raw_np[start_idx: stop_idx, :] = vow_data_spk  # Append vowel-speaker to Output Speakers label
    spk_coords.append(vow_data_idx)
    print('--------------------------------')
    # ---------------------------------------------------------------------------------------------

# Different labels counter
diff_phoneme_labels = len(np.unique(y_labels_raw_np))
diff_speaker_labels = len(np.unique(z_labels_raw_np))
print()
print(f'--> RAW DATASET shape: {x_data_raw_np.shape}, w. {diff_phoneme_labels} PHONEME Labels & {diff_speaker_labels} SPEAKER
print('-------------------------------------------------------------------')
# -------------------------------------------------------------------------------------------------------------

# Fundamental Frequency (ratio) Normalization
x_data_fund_norm = np.zeros(x_data_raw_np.shape)  # Output initialization

for i in range(x_data_raw_np.shape[1]):  # For each feature...
    if i >= 1: # For each formant column...
        x_data_fund_norm[:, i] = x_data_raw_np[:, i] / x_data_raw_np[:, 0]  # i-Formant value / i-Fundamental value
```

```
    else:  # Exception for Fundamental freq column
        x_data_fund_norm[:, i] = x_data_raw_np[:, i]

print(f"'Fundamental Normalized' Dataset: {x_data_fund_norm.shape[0]} elements & {x_data_fund_norm.shape[1]} features each")
print('-----------------------------------------------------------------------')
# ---------------------------------------------------------------------------------------------------------

a = 0.  # Lower bound
b = 1.  # Upper bound
x_data_minmax = np.zeros((x_data_fund_norm.shape))
print(f'Fundamental Ratios  STATS:   Min. = {x_data_fund_norm[:, 1:].min()}    Max. = {x_data_fund_norm[:, 1:].max()}')

x_data_minmax[:, 1:] = a + ((x_data_fund_norm[:, 1:] - x_data_fund_norm[:, 1:].min()) / (x_data_fund_norm[:, 1:].max() - x_dat
x_data_minmax[:, 0] = x_data_fund_norm[:, 0]  # Fundamental column exception

print(f'Min-Max Ratios      STATS:   Min. = {x_data_minmax[:, 1:].min()}                   Max. = {x_data_minmax[:, 1:].max()}
print('---------------------------------------------------------------------------')
print(f"'Fundamental & Min-Max Normalized' Dataset: {x_data_minmax.shape[0]} elements & {x_data_minmax.shape[1]} features each
# ---------------------------------------------------------------------------------------------------------

# Output Store
phon_classes_size = np.array(vow_size)  # Phoneme classes sizes array
phon_classes_indices = np.array(end_idx)  # Phoneme classes indices (start/end included)
phoneme_speaker_coordinates = np.array(spk_coords) # Each couple is (vow_spk sub-group start idx, vow-spk sub-group size)
np.savez_compressed(file='./HGCW_dataset_utils',
                    HGCW_raw = x_data_raw_np,
                    HGCW_fund_norm = x_data_fund_norm,
                    HGCW_minmax = x_data_minmax,
                    HGCW_phon_labels = y_labels_raw_np,
                    HGCW_spk_labels = z_labels_raw_np,
                    phon_size = phon_classes_size,
                    phon_idx = phon_classes_indices,
                    phon_spk_coords = phoneme_speaker_coordinates)


# HGCW Dataset (3 x 4 Formants: 3- steady state, 3 - 10%, 3 - 50%, 3 - 80%) + SPEAKER Label

# Database (.DAT file) Features Reading (converted to NumPy array)
import numpy as np

formant_analysis_data = np.loadtxt("./HGCW_LPC_formants_fine.dat", usecols=(2, 3, 4, 5, 6, 7, 8, 18, 19, 20, 27, 28, 29))
formant_analysis_filenames = np.loadtxt("./HGCW_LPC_formants_fine.dat", usecols=0, dtype=str)
# Useful Parameters
vowels = ['ae', 'ah', 'aw', 'eh', 'er', 'ei', 'ih', 'iy', 'oa', 'oo', 'uh', 'uw']  # Vowels list
speakers = ['b', 'g', 'm', 'w']  # Speakers list
print(f"Dataset: {formant_analysis_data.shape[0]} samples (for {len(vowels)} labels) & {formant_analysis_data.shape[1]} featur
# ---------------------------------------------------------------------------------------------------------

# Remove Null elements
nonnull_data, nonnull_filenames, _ = null_filter(formant_analysis_data, formant_analysis_filenames)
print(f"NON NULL Dataset: {nonnull_data.shape[0]} samples (for {len(vowels)} labels) & {nonnull_data.shape[1]} features each")
print('---------------------------------------------------------------------------')
print()
```

Future Works

- Extract an HGCW output with formant tracks only (No Steady States): test the minimum amount of time-points required for a correct evaluation;
- Repeat Analysis & feature extraction on:
  - PB Dataset
  - TIMIT Dataset
  - Bernard Dataset (Australian English)
  - VTRFormants Dataset
  - IRCAM VocalSet

One-Class Sub-Network Analysis

(Vowel Phonemes Binary Classifier)

Author: S. Giacomelli

Year: 2023

Affiliation: A.Casella Conservatory (student)

Master Degree Thesis: "*Vowel phonemes Analysis & Classification by means of OCON rectifiers Deep Learning Architectures*"

Description: Python scripts for "One-Class" neural network binary classifier analysis and optimization

```
# Numerical computations packages/modules
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

# Dataset processing modules
from torch.utils.data import DataLoader
from sklearn.model_selection import train_test_split

# Graphic visualization modules
import matplotlib.pyplot as plt
import matplotlib_inline
matplotlib_inline.backend_inline.set_matplotlib_formats('svg')

# Common Seed initialization
SEED = 42  # ... the answer to the ultimate question of Life, the Universe, and Everything... (cit.)
```

HGCW Dataset One-Hot Encoding

(class binarization)

```
# Load Dataset
HGCW_dataset_utils = np.load(file='./HGCW_dataset_utils.npz')
print('Raw features                       Data shape:', HGCW_dataset_utils['HGCW_raw'].shape)
print('Fundamental Normalized features Data shape:', HGCW_dataset_utils['HGCW_fund_norm'].shape)
print('MinMax features                    Data shape:', HGCW_dataset_utils['HGCW_minmax'].shape)
print('Labels                             Data shape:', HGCW_dataset_utils['HGCW_labels'].shape)
print('Classes size                       Data shape:', HGCW_dataset_utils['classes_size'].shape)
print('Classes indices                    Data shape:', HGCW_dataset_utils['classes_idx'].shape)

x_data_raw_np = HGCW_dataset_utils['HGCW_raw']
x_data_fund_norm = HGCW_dataset_utils['HGCW_fund_norm']
x_data_minmax = HGCW_dataset_utils['HGCW_minmax']
y_labels_raw_np = HGCW_dataset_utils['HGCW_labels']
vow_size = HGCW_dataset_utils['classes_size']
end_idx = HGCW_dataset_utils['classes_idx']

# Auxiliary lists
vowels = ['ae', 'ah', 'aw', 'eh', 'er', 'ei', 'ih', 'iy', 'oa', 'oo', 'uh', 'uw']  # Vowels list
colors = ['red', 'saddlebrown', 'darkorange', 'darkgoldenrod', 'gold', 'darkkhaki', 'olive', 'darkgreen', 'steelblue', 'fuchsi


# Class-specific One-hot encoding (Binarization)
def one_hot_encoder(sel_class_number: int = 3, dataset: np.ndarray = x_data_minmax, orig_labels: int = len(vowels), classes_si

    classes = [n for n in range(orig_labels)]  # Class Labels list initialization

    # Auxiliary Parameters Initialization
    if sel_class_number < len(classes):
        classes.remove(sel_class_number)  # REST Classes list
        if debug is True:
            print(f'Selected Class "{vowels[sel_class_number]}" : {classes_size[sel_class_number]} samples')
        sub_classes_size = classes_size[sel_class_number] // len(classes)
        if debug is True:
            print(f'Rest Classes size (...each): {sub_classes_size} samples')

        # 1-Subset processing
        sub_data = dataset[classes_idx[sel_class_number]: classes_idx[sel_class_number + 1], :]  # Selected Class feature slic
        sub_data_labels_bin = np.ones((classes_size[sel_class_number], 1), dtype='int')  # Selected Class labels (1) creation
        sub_data_labels = np.ones((classes_size[sel_class_number], 1), dtype='int') * sel_class_number

        # 0-Subset processing
        for i in classes:
```

```
            class_i_indices = np.random.choice(np.arange(classes_idx[i], classes_idx[i + 1], 1), size=sub_classes_size, replac
            sub_class_i_array = dataset[class_i_indices, :]
            sub_class_labels_bin_array = np.zeros((sub_class_i_array.shape[0], 1), dtype='int')  # Rest I-esimal Class labels
            sub_class_labels_array = np.ones((sub_class_i_array.shape[0], 1), dtype='int') * i

            # Outputs append
            sub_data = np.vstack((sub_data, sub_class_i_array))
            sub_data_labels_bin = np.vstack((sub_data_labels_bin, sub_class_labels_bin_array))
            sub_data_labels = np.vstack((sub_data_labels, sub_class_labels_array))
    else:
        raise ValueError(f'Invalid Class ID: "{sel_class_number}" --> It must be less than {len(classes)}!')

    return sub_data, sub_data_labels_bin, sub_data_labels

# Test Call
dataset = x_data_minmax
sel_class_number = 0
sub_data, sub_data_labels_bin, sub_data_labels = one_hot_encoder(sel_class_number=sel_class_number, dataset=dataset, debug=Tru
diff_labels_bin = len(np.unique(sub_data_labels_bin))
print('--------------------------------')
print(f"SUB'Min-Max' Normalized Dataset: {sub_data.shape[0]} elements (w. {diff_labels_bin} BINARIZED labels) & {sub_data.shap
print(f'Also AVAILABLE Standard Labels: {sub_data_labels.shape[0]} samples (w. {len(np.unique(sub_data_labels))} labels)')


# Sub-Dataset Plot (previous example)
classes = [n for n in range(len(vowels))]
sub_classes_size = vow_size[sel_class_number] // (len(classes) - 1)

plt.figure(figsize=(12, 15))
plt.suptitle(f'Sub-Dataset ({vowels[sel_class_number]} - example) One-Hot Encoding')

counter = 0
for index in classes:
    if index == sel_class_number:  # Selected Class exception (non increment counter variable)
        first_coords = sub_data[0: vow_size[sel_class_number], 1]
        second_coords = sub_data[0: vow_size[sel_class_number], 2]
        third_coords = sub_data[0: vow_size[sel_class_number], 3]
    else:
        start = vow_size[sel_class_number] + (counter * sub_classes_size)
        end = start + sub_classes_size

        first_coords = sub_data[start : end, 1]
        second_coords = sub_data[start : end, 2]
        third_coords = sub_data[start : end, 3]

        counter +=1

    plt.subplot(3, 2, 1)
    plt.title('$1_{st}$ VS $2_{nd}$')
    plt.scatter(first_coords, second_coords, marker='o', color=colors[index], label=f'Vowel "{vowels[index]}"')
    plt.xlabel('$1_{st}$ Formant Ratio')
    plt.ylabel('$2_{nd}$ Formant Ratio')
    plt.legend(loc='best')
    plt.grid(True)

    plt.subplot(3, 2, 3)
    plt.title('$1_{st}$ VS $3_{rd}$')
    plt.scatter(first_coords, third_coords, marker='o', color=colors[index], label=f'Vowel "{vowels[index]}"')
    plt.xlabel('$1_{st}$ Formant Ratio')
    plt.ylabel('$3_{rd}$ Formant Ratio')
    plt.legend(loc='best')
    plt.grid(True)

    plt.subplot(3, 2, 5)
    plt.title('$2_{nd}$ VS $3_{rd}$')
    plt.scatter(second_coords, third_coords, marker='o', color=colors[index], label=f'Vowel "{vowels[index]}"')
    plt.xlabel('$2_{nd}$ Formant Ratio')
    plt.ylabel('$3_{rd}$ Formant Ratio')
    plt.legend(loc='best')
    plt.grid(True)

plt.subplot(3, 2, 2)
plt.title('$1_{st}$ VS $2_{nd}$ Binarized')
plt.scatter(sub_data[0: vow_size[sel_class_number], 1], sub_data[0: vow_size[sel_class_number], 2], color=colors[sel_class_num
plt.scatter(sub_data[vow_size[sel_class_number]:, 1], sub_data[vow_size[sel_class_number]:, 2], color='grey', label=f'Rest')
plt.xlabel('$1_{st}$ Formant Ratio')
plt.ylabel('$2_{nd}$ Formant Ratio')
plt.legend(loc='best')
plt.grid(True)

plt.subplot(3, 2, 4)
plt.title('$1_{st}$ VS $3_{rd}$ Binarized')
plt.scatter(sub_data[0: vow_size[sel_class_number], 1], sub_data[0: vow_size[sel_class_number], 3], color=colors[sel_class_num
```

```
plt.scatter(sub_data[vow_size[sel_class_number]:, 1], sub_data[vow_size[sel_class_number]:, 3], color='grey', label=f'Rest')
plt.xlabel('$1_{st}$ Formant Ratio')
plt.ylabel('$3_{rd}$ Formant Ratio')
plt.legend(loc='best')
plt.grid(True)

plt.subplot(3, 2, 6)
plt.title('$2_{nd}$ VS $3_{rd}$ Binarized')
plt.scatter(sub_data[0: vow_size[sel_class_number], 2], sub_data[0: vow_size[sel_class_number], 3], color=colors[sel_class_num
plt.scatter(sub_data[vow_size[sel_class_number]:, 2], sub_data[vow_size[sel_class_number]:, 3], color='grey', label=f'Rest')
plt.xlabel('$2_{nd}$ Formant Ratio')
plt.ylabel('$3_{rd}$ Formant Ratio')
plt.legend(loc='best')
plt.grid(True)

plt.tight_layout()
plt.savefig(f'{vowels[sel_class_number]}_class_one_hot_encoding')
plt.show()


# Train/Test split (auxiliary function)
def train_test_split_aux(features_dataset, labels_dataset, test_perc, tolerance):
    """
    An auxiliary Train_Test_split function (based on Scikit Learn implementation) w. balance tolerance specification
    """
    test_size = int(test_perc / 100 * len(features_dataset))
    train_balance = 0  # Output Training set balance value initialization
    test_balance = 0  # Output Testing set balance value initialization

    min_tol = np.mean(labels_dataset) - tolerance
    max_tol = np.mean(labels_dataset) + tolerance
    print(f'Data Balancing  (TARGET = {np.mean(labels_dataset)} +- {tolerance}): ', end='')

    while (min_tol >= train_balance or train_balance >= max_tol) or (min_tol >= test_balance or test_balance >= max_tol):
        train_data, test_data, train_labels, test_labels = train_test_split(features_dataset, labels_dataset, test_size=test_s
        train_balance = np.mean(train_labels)
        test_balance = np.mean(test_labels)
        print('.', end='')
    else:
        print('OK')

    return train_data, test_data, train_labels, test_labels, train_balance, test_balance

# Train-Dev-Test split function
def train_dev_test_split(x_data, y_labels, split_list, tolerance=0.1, output='Loaders', debug=False):
    """
    Compute a Train, Development (Hold-Out) and a Test set split w. PyTorch Dataset conversion (and eventual Loaders initializ
    """
    if len(split_list) == 3:
        # Train - Dev+Test separation
        print('Training --- Devel/Test SPLIT')
        train_data, testTMP_data, train_labels, testTMP_labels, _, _ = train_test_split_aux(x_data, y_labels, (split_list[1] *
        print('---------------------------------')

        # Dev - Test separation
        print('Devel    ---     Test SPLIT')

        split = ((split_list[1] * 100) / np.sum(split_list[1:] * 100)) * 100  # Split in %
        dev_data, test_data, dev_labels, test_labels, _, _ = train_test_split_aux(testTMP_data, testTMP_labels, split, toleran
        print('---------------------------------')

        # Tensor Conversion
        train_data_tensor = torch.tensor(train_data).float()
        train_labels_tensor = torch.tensor(train_labels, dtype=torch.int64).squeeze()
        dev_data_tensor = torch.tensor(dev_data).float()
        dev_labels_tensor = torch.tensor(dev_labels, dtype=torch.int64).squeeze()
        test_data_tensor = torch.tensor(test_data).float()
        test_labels_tensor = torch.tensor(test_labels, dtype=torch.int64).squeeze()
        if debug is True:
            print(f'Training Data        Shape: {train_data.shape}')
            print(f'Development Data     Shape: {dev_data.shape}')
            print(f'Testing Data         Shape: {test_data.shape}')

            # Balance Evaluation
            print(f'Training Set       Balance: {np.mean(train_labels)}')
            print(f'Development Set    Balance: {np.mean(dev_labels)}')
            print(f'Testing Set        Balance: {np.mean(test_labels)}')

        if output != 'Loaders':
            return train_data_tensor, train_labels_tensor, dev_data_tensor, dev_labels_tensor, test_data_tensor, test_labels_t
        else:
            # PyTorch Dataset Conversion
            train_dataset = torch.utils.data.TensorDataset(torch.tensor(train_data).float(), torch.tensor(train_labels, dtype=
```

```
            dev_dataset = torch.utils.data.TensorDataset(torch.tensor(dev_data).float(), torch.tensor(dev_labels, dtype=torch.
            test_dataset = torch.utils.data.TensorDataset(torch.tensor(test_data).float(), torch.tensor(test_labels, dtype=tor

            # DataLoader (Batches) --> Drop-Last control to optimize training
            trainLoader = DataLoader(train_dataset, shuffle=False, batch_size = 32, drop_last=True)
            devLoader = DataLoader(dev_dataset, shuffle=False, batch_size = dev_dataset.tensors[0].shape[0])
            testLoader = DataLoader(test_dataset, shuffle=False, batch_size = test_dataset.tensors[0].shape[0])
            if debug is True:
                print(f'Training Set    Batch Size: {trainLoader.batch_size}')
                print(f'Development Set Batch Size: {devLoader.batch_size}')
                print(f'Testing Set     Batch Size: {testLoader.batch_size}')

            return trainLoader, devLoader, testLoader
    else:
        # Train - Test separation
        print('Training --- Test    SPLIT')
        train_data, test_data, train_labels, test_labels, _, _ = train_test_split_aux(x_data, y_labels, split_list[1] * 100, t
        print('--------------------------')

        # Tensor Conversion
        train_data_tensor = torch.tensor(train_data).float()
        train_labels_tensor = torch.tensor(train_labels, dtype=torch.int64).squeeze()
        test_data_tensor = torch.tensor(test_data).float()
        test_labels_tensor = torch.tensor(test_labels, dtype=torch.int64).squeeze()
        if debug is True:
            print(f'Training Data        Shape: {train_data.shape}')
            print(f'Testing Data         Shape: {test_data.shape}')

            # Balance Evaluation
            print(f'Training Set    Balance: {np.mean(train_labels)}')
            print(f'Testing Set     Balance: {np.mean(test_labels)}')

        if output != 'Loaders':
            return train_data_tensor, train_labels_tensor, test_data_tensor, test_labels_tensor
        else:
            # PyTorch Dataset Conversion
            train_dataset = torch.utils.data.TensorDataset(torch.tensor(train_data).float(), torch.tensor(train_labels, dtype=
            test_dataset = torch.utils.data.TensorDataset(torch.tensor(test_data).float(), torch.tensor(test_labels, dtype=tor

            # DataLoader (Batches) --> Drop-Last control to optimize training
            trainLoader = DataLoader(train_dataset, shuffle=False, batch_size = 32, drop_last=True)
            testLoader = DataLoader(test_dataset, shuffle=False, batch_size = test_dataset.tensors[0].shape[0])
            if debug is True:
                print(f'Training Set    Batch Size: {trainLoader.batch_size}')
                print(f'Testing Set     Batch Size: {testLoader.batch_size}')

            return trainLoader, testLoader
```

Multi-Layer Perceptron Binary Classifier

```
# Dynamic Multi-Layer Architecture Class (w. units and activation function specification)
class binaryClassifier(nn.Module):                                          # nn.Module: base class to inherit from
    def __init__(self, n_layers, n_units, act_fun):                         # self + attributes (architecture hyper-parame
        super().__init__()

        self.layers = nn.ModuleDict()                                       # Dictionary to store Model layers
        self.nLayers = n_layers                                             # Class instance parameter

        # Input Layer
        if n_layers == 1:
            self.layers['input'] = nn.Linear(3, n_units)                    # Key 'input' layer specification

        else:
            self.layers['input'] = nn.Linear(3, n_units[0])


        # Hidden Layers
        if n_layers == 1:
            self.layers[f'hidden0'] = nn.Linear(n_units, n_units)
        else:
            for i in range(n_layers):
                if i == (n_layers - 1):
                    self.layers[f'hidden{i}'] = nn.Linear(n_units[i], n_units[i])
                else:
                    self.layers[f'hidden{i}'] = nn.Linear(n_units[i], n_units[i + 1])

        # Output Layer
        if n_layers == 1:
            self.layers['output'] = nn.Linear(n_units, 1)                   # Key 'output' layer specification
        else:
```

```
            self.layers['output'] = nn.Linear(n_units[n_layers - 1], 1)

        # Activation Function
        self.actfun = act_fun                                                   # Function string-name attribute association

        # Weights initialization (Kaiming He - Normal Distributed)
        for layer in self.layers.keys():
            nn.init.kaiming_normal_(self.layers[layer].weight, mode='fan_in')

    # Forward Pass Method
    def forward(self, x):

        # Activation function object computation
        actfun = getattr(torch.nn, self.actfun)

        # Input Layer pass                                                      --> Weightening (Dot Product) "Linear transfor
        x = actfun()(self.layers['input'](x))

        # Hidden Layers sequential pass                                         --> Weightening (Dot Product) "Linear transfor
        for i in range(self.nLayers):
            x = actfun()(self.layers[f'hidden{i}'](x))

        # Output Layer pass                                                     --> Output Weightening (Dot Product) "Linear t
        x = self.layers['output'](x)
        x = nn.Sigmoid()(x)

        return x


# Train/Test function (w. variable Backpropagation Optimizer Algorithm definition)
def cross_val_train_test(model, optim: str, epochs: int, learning_rate, train_data: torch.Tensor, train_labels: torch.Tensor,
    """
    Train & Test an ANN Classifier w. Binary Cross Entropy Loss computation and the specified Backpropagation Optimizer algori
    """
    # Loss Function initialization
    loss_function = nn.BCELoss()

    # Optimizer Algorithm initialization
    optimizer_function = getattr(torch.optim, optim)  # Optimizer function retrieving
    optimizer = optimizer_function(model.parameters(), lr=learning_rate)  # Parameters application (rest are standard initiali

    # TRAINING Phase
    train_losses = []
    train_accuracies = []

    model.train()  # TRAINING Switch ON

    for i in range(epochs):
        train_predictions = model(train_data)
        train_loss = loss_function(train_predictions.squeeze(), train_labels.squeeze().to(torch.float))
        train_losses.append(train_loss.detach())

        # Backpropagation
        optimizer.zero_grad()
        train_loss.backward()
        optimizer.step()

        train_accuracy = 100 * torch.mean(((train_predictions.squeeze() > 0.5) == train_labels.squeeze()).float())
        train_accuracies.append(train_accuracy.detach())

        if debug is True:
            if i % 100 == 0:
                print(f'Epoch {i} --> Train Accuracy: {train_accuracy.detach()}%')
                print('-------------------------')

    # TESTING Phase
    model.eval()  # EVALUATION Switch ON (TRAINING Switch OFF)
    with torch.no_grad():  # Gradient (and Batch Normalization) deactivation
        test_predictions = model(test_data)
        test_accuracy = 100 * torch.mean(((test_predictions.squeeze() > 0.5) == test_labels.squeeze()).float())

        if debug is True:
            print(f'TEST ACCURACY: {test_accuracy.detach()} %')
            print('-------------------------')

    return test_predictions.detach(), test_accuracy.detach(), train_losses, train_accuracies

# Batch Training function
def mini_batch_train_test(model, optim: str, epochs: int, learning_rate, train_loader, dev_loader, test_loader, debug=False):
    """
    Train & Test an ANN Architecture via Mini-Batch Training (w. Train/Dev/Test PyTorch Loaders) and same params of cross_vali
    """
    # Loss Function initialization
```

```python
    loss_function = nn.BCELoss()

    # Optimizer Algorithm initialization
    optimizer_function = getattr(torch.optim, optim)
    optimizer = optimizer_function(model.parameters(), lr=learning_rate)

    # Output list initialization
    train_accuracies = []
    train_losses = []
    dev_accuracies = []

    # TRAINING Phase
    for epoch in range(epochs):
        model.train()  # TRAINING Switch ON

        batch_accuracies = []
        batch_losses = []

        # Training BATCHES Loop
        for data_batch, labels_batch in train_loader:
            train_predictions = model(data_batch)
            train_loss = loss_function(train_predictions.squeeze(), labels_batch.type(torch.int64).float())
            batch_losses.append(train_loss.detach())

            # Backpropagation
            optimizer.zero_grad()
            train_loss.backward()
            optimizer.step()

            # Accuracy
            train_accuracy = 100 * torch.mean(((train_predictions.squeeze() > 0.5) == labels_batch.type(torch.int64).squeeze()

            # Batch Stats appending
            batch_accuracies.append(train_accuracy.detach())
            batch_losses.append(train_loss.detach())

        # Training Stats appending
        train_accuracies.append(np.mean(batch_accuracies))  # Average of Batch Accuracies = Training step accuracy
        train_losses.append(np.mean(batch_losses))  # Average of Batch Losses = Training step Losses

        # EVALUATION (Dev) Phase
        model.eval()
        with torch.no_grad():
            dev_data_batch, dev_labels_batch = next(iter(dev_loader))
            dev_predictions = model(dev_data_batch)

            dev_accuracy = 100 * torch.mean(((dev_predictions.squeeze() > 0.5) == dev_labels_batch.type(torch.int64).squeeze()

            if debug is True:
                if epoch % 100 == 0:
                    print(f'Epoch {epoch} --> DEV ACCURACY: {dev_accuracy.detach():.3f} %')
                    print('--------------------------------')

            # Evaluation accuracy appending
            dev_accuracies.append(dev_accuracy.detach())

    # TEST Phase
    model.eval()
    with torch.no_grad():
        test_data_batch, test_labels_batch = next(iter(test_loader))
        test_predictions = model(test_data_batch)
        test_accuracy = 100 * torch.mean(((test_predictions.squeeze() > 0.5) == test_labels_batch.type(torch.int64).squeeze())

        if debug is True:
            print(f'TEST ACCURACY: {test_accuracy.detach():.2f} %')
            print('-------------------------------------------------------------------')

    return train_accuracies, train_losses, dev_accuracies, test_accuracy.detach()
```

Architecture optimal *hyper-parameters* estimate

Grid-Search (orders of magnitude)

- **Hidden Layers**: 1
- **Hidden Nodes**: $(10, 50, 100)$
- **Activation Function**: ReLU (*He standard distribution initialization*)

K. He, X. Zhang, S. Ren, J. Sun (2015) - [Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification]()

- **Learning Rates**: $(0.001, 0.0001, 0.00001)$
- **Optimizers**: Adam, RMSprop

Root Mean Square Propagation (RMSprop)

$$w_t \leftarrow w_{t-1} - \frac{\eta}{\sqrt{v_t + \epsilon}} \nabla L$$

with

$$v_t = (1-\beta)(\nabla L)^2 + \beta v_{t-1}$$

Similar to *Momentum* conditioning, but applied to Learning Rate coefficient (instead to Loss function) according to Gradient magnitudes. For this reason we speak about *Dynamic Learning Rate*, where:

- large gradients: implies small LR and smaller steps of minimization
- small gradients $(0 < x < 1)$: implies very large steps of minimization

ϵ is a standard positive coefficient added to denominator to avoid division by 0: usually 10^{-8}

Adaptive Momentum (Adam)

Probably nowadays best gradient optimizer:

$$w_t \leftarrow w_{t-1} - \frac{\eta}{\sqrt{s_t + \epsilon}} v_t$$

with

$$v_t = \frac{(1-\beta_1)\nabla L + \beta_1 v_{t-1}}{1-\beta_1^t}$$

$$s_t = \frac{(1-\beta_2)(\nabla L)^2 + \beta_2 s_{t-1}}{1-\beta_2^t}$$

A combined form of **Momentum** and **RMSprop** with a dampening normalization factor, learning epoch dependent.

Reference: [PyTorch Reference - torch.optim Algorithms]

```
# Experiment Parameters
epochs = 1000  # For each "Batch-Set"
iterations = 3  # A total of 3000 Epochs of Training (3x Batch-Sub-Dataset shuffling)
min_tolerance = 0.1 # ...for sub-dataset balancing

# Architecture Hyper-Parameters
hidden_layers = 1
hidden_nodes = [10, 50, 100]
act_fun = 'ReLU'
learning_rates = [0.001, 0.0001, 0.00001]  # [10^-3, 10^-4, 10^-5]
optimizers = ['Adam', 'RMSprop']


# AVG. Time 2h
from time import perf_counter
debug=False
experiment_results = np.zeros((len(hidden_nodes), len(learning_rates), len(optimizers), 2))  # Output Matrix initialization (1

exp_counter = 0
for i in range(len(hidden_nodes)):
    for j in range(len(learning_rates)):
        for k in range(len(optimizers)):

            exp_counter += 1 # Aux variable increment
            print(f'Experiment {exp_counter}: Units (HL): {hidden_nodes[i]}, LR: {learning_rates[j]}, Optimizer: {optimizers[k

            test_accuracies = []  # List of Classes Test Accuracies (Re-initialized for each experiment)
            training_times = []  # List of Classes Training Times (Re-initialized for each experiment)

            # Experiment Routine
            for w in range(len(vowels)):

                # Reset Seed
                torch.manual_seed(SEED)

                # Create Classifier
                binary_classifier = binaryClassifier(1, hidden_nodes[i], act_fun)

                # Iterated (w. Batch-Sets shuffling) Training
                start_timer = perf_counter()
                for iteration in range(iterations):

                    # Dataset processing
                    sub_data, sub_data_labels_bin, _ = one_hot_encoder(sel_class_number=w, dataset=x_data_minmax, debug=debug)
                    print('----------------------------------')
                    trainLoader, devLoader, testLoader = train_dev_test_split(sub_data[:, 1:], sub_data_labels_bin, [0.7, 0.15
```

```
                    # Train/Test Architecture
                    _, _, _, test_accuracy = mini_batch_train_test(binary_classifier, optimizers[k], epochs, learning_rates[j]
                    print(f'Sub-Net "{vowels[w]}" Partial-{iteration + 1} TEST ACCURACY: {test_accuracy:.2f}%')

                stop_timer = perf_counter()

                # Class Outputs append
                test_accuracies.append(test_accuracy) # in %
                training_times.append(stop_timer - start_timer) # in sec.
                print('----------------------------------------------------------------------')

            print(f'Classes MEAN ACCURACY: {np.mean(test_accuracies)}%')
            print(f'Classes Mean Training Runtime: {np.mean(training_times)}sec.')

            experiment_results[i, j, k, 0] = np.mean(test_accuracies)  # Average of 12 classes Accuracies
            experiment_results[i, j, k, 1] = np.mean(training_times)  # Average of 12 classes Training Times

            print('----------------------------------------------------------------------------------------------------')

# Outputs Save
np.savez_compressed(file='./architecture_grid_search',
                    avg_test_accuracies=experiment_results[:, :, :, 0],
                    avg_training_times=experiment_results[:, :, :, 1])


# Architecture Grid-Search Experiment Plot
experiment_data = np.load(file='./architecture_grid_search.npz')

plt.figure(figsize=(12, 12))
plt.suptitle('Architecture Hyper-Parameters Experiment\n(average results across all classes)\n')

plt.subplot(2, 1, 1)
counter = 1
for i in range(experiment_data['avg_test_accuracies'].shape[0]):
    for j in range(experiment_data['avg_test_accuracies'].shape[1]):
        for k in range(experiment_data['avg_test_accuracies'].shape[2]):
            if k == 1:
                plt.bar(counter, experiment_data['avg_test_accuracies'][i, j, k], color='k', label=f'HN: {hidden_nodes[i]}, LR
            else:
                plt.bar(counter, experiment_data['avg_test_accuracies'][i, j, k], color='r', label=f'HN: {hidden_nodes[i]}, LR

            counter += 1

max_accuracy = np.max(experiment_data['avg_test_accuracies'])
plt.axhline(max_accuracy, color='grey', linestyle='--')
plt.title(f'Test Accuracies (RMSprop = Black, Adam = Red), Max.: {max_accuracy:.2f}%')
plt.xlabel('Experiment Run (index)')
plt.xticks([(n + 1) for n in range(18)], [(n + 1) for n in range(18)])
plt.ylabel('Accuracy (in %)')
plt.ylim([50, 100])
plt.grid()
plt.legend(loc='lower center', bbox_to_anchor=(0.5, -0.3), fancybox=True, shadow=True, ncol=6)

plt.subplot(2, 1, 2)
counter = 1
for i in range(experiment_data['avg_training_times'].shape[0]):
    for j in range(experiment_data['avg_training_times'].shape[1]):
        for k in range(experiment_data['avg_training_times'].shape[2]):
            if k == 1:
                plt.bar(counter, experiment_data['avg_training_times'][i, j, k], color='k', label=f'HN: {hidden_nodes[i]}, LR:

            else:
                plt.bar(counter, experiment_data['avg_training_times'][i, j, k], color='pink', label=f'HN: {hidden_nodes[i]},

            counter += 1

plt.title('Training Times (RMSprop = Black, Adam = Pink)')
plt.xlabel('Experiment Run (index)')
plt.xticks([(n + 1) for n in range(18)], [(n + 1) for n in range(18)])
plt.ylabel('Time (in sec.)')
plt.ylim([20, 40])
plt.grid()
plt.legend(loc='lower center', bbox_to_anchor=(0.5, -0.3), fancybox=True, shadow=True, ncol=6)

plt.tight_layout()
plt.savefig('architecture_grid_search')
plt.show()


# Best Test Accuracy inspection
best_run_acc = experiment_data['avg_test_accuracies'][2, 1, 0]
```

```
best_run_time = experiment_data['avg_training_times'][2, 1, 0]
print(f'Run 15 (Adam Optimizer): {best_run_acc:.2f}% in {best_run_time:.2f}sec. (per class)')
```

Architecture Optimization

```
Multi-Layer Perceptron
- Input Layer: 3 features [formant ratios, min-max normalized]
- Hidden Layer: 100 units
- Output Layer: 1 normalized probability
- Learning Rate: 0.0001 (10^-4)
- Optimizer : Adam (Adaptive Momentum)

- Mini-Batch Training:
    . Re-iterated Sub-Dataset Shuffling
    . Batch size = 32
```

- **Bias Initialization**: 0
- **Regularization**: DropOut, Batch-Normalization, L2 Loss Regularization

DropOut

Probabilistic method to "mute" (sparsing) learning inference of arbitrary nodes during each epoch. It aims to uniform learning patterns and avoid mnemonic recognition/association of data examples.

- N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov (2014) - [Dropout: A Simple Way to Prevent Neural Networks from Overfitting]

```
# Dynamic Multi-Layer Architecture Class (w. units, activation function and DropOut Rate specification)
class binaryClassifier_dropout(nn.Module):                                  # nn.Module: base class to inherit from
    def __init__(self, n_layers, n_units, act_fun, rate_in, rate_hidden):       # self + attributes (architecture hyper-parame
        super().__init__()

        self.layers = nn.ModuleDict()                                       # Dictionary to store Model layers
        self.nLayers = n_layers                                             # Class instance parameter

        # Input Layer
        self.layers['input'] = nn.Linear(3, n_units)                        # Key 'input' layer specification

        # Hidden Layers
        self.layers[f'hidden'] = nn.Linear(n_units, n_units)

        # Output Layer
        self.layers['output'] = nn.Linear(n_units, 1)                       # Key 'output' layer specification

        # Activation Function
        self.actfun = act_fun                                               # Function string-name attribute association

        # Dropout Parameter
        self.dr_in = rate_in
        self.dr_hidden = rate_hidden

        # Weights & Bias initialization
        for layer in self.layers.keys():
            nn.init.kaiming_normal_(self.layers[layer].weight, mode='fan_in')   # Kaiming He - Normal Distributed (ReLU specif
        for layer in self.layers.keys():
            self.layers[layer].bias.data.fill_(0.)                          # Bias initialization

    # Forward Pass Method
    def forward(self, x):

        # Activation function object computation
        actfun = getattr(torch.nn, self.actfun)

        # Input Layer pass                                                  --> Weightening (Dot Product) "Linear transfor
        x = actfun()(self.layers['input'](x))
        x = F.dropout(x, p=self.dr_in, training=self.training)              # Activate DropOut only when Model Training ==

        # Hidden Layers sequential pass                                     --> Weightening (Dot Product) "Linear transfor
        x = actfun()(self.layers[f'hidden'](x))
        x = F.dropout(x, p=self.dr_hidden, training=self.training)          # Same as "Input pass"

        # Output Layer pass                                                 --> Output Weightening (Dot Product) "Linear t
        x = self.layers['output'](x)
        x = nn.Sigmoid()(x)

        return x


# Experiment Parameters
epochs = 1000  # For each "Batch-Set"
iterations = 6  # A total of 6000 Epochs of Training (6x Batch-Sub-Dataset shuffling) --> w. Early Stopping
min_tolerance = 0.1 # ...for sub-dataset balancing

# Architecture Hyper-Parameters
hidden_layer = 1
hidden_nodes = 100
act_fun = 'ReLU'
learning_rate = 0.0001  # 10^-4
optimizer = 'Adam'

# DropOut Regularization Parameters
dropout_rates_in = [0.8, 0.9]
dropout_rates_hidden = (np.arange(5) / 10.) + 0.5


# AVG. Time 1h 30min
from time import perf_counter
debug=False
experiment_results = np.zeros((len(dropout_rates_in), len(dropout_rates_hidden), 2))

exp_counter = 0
for i in range(len(dropout_rates_in)):
    for j in range(len(dropout_rates_hidden)):
        exp_counter += 1
        print(f'Experiment {exp_counter}: DropOut Input: {dropout_rates_in[i]}, DropOut Hidden: {dropout_rates_hidden[j]}')

        test_accuracies = []  # List of Classes Test Accuracies (Re-initialized for each experiment)
        training_times = []  # List of Classes Training Times (Re-initialized for each experiment)

        # Experiment Routine
```

```
        for k in range(len(vowels)):

            # Reset Seed
            torch.manual_seed(SEED)

            # Create Classifier
            binary_classifier = binaryClassifier_dropout(1, hidden_nodes, act_fun, dropout_rates_in[i], dropout_rates_hidden[j

            # Iterated (w. Batch-Sets shuffling) Training
            iteration = 0

            start_timer = perf_counter()
            while iteration < iterations:

                # Dataset processing
                sub_data, sub_data_labels_bin, _ = one_hot_encoder(sel_class_number=k, dataset=x_data_minmax, debug=debug)
                print('-----------------------------------')
                trainLoader, devLoader, testLoader = train_dev_test_split(sub_data[:, 1:], sub_data_labels_bin, [0.7, 0.15, 0.

                # Train/Test Architecture
                _, _, _, test_accuracy = mini_batch_train_test(binary_classifier, optimizer, epochs, learning_rate, trainLoade
                print(f'Sub-Net "{vowels[k]}" Partial-{iteration + 1} TEST ACCURACY: {test_accuracy:.2f}%')

                if test_accuracy > 93.67:  # If specific class instance overshot class mean accuracy
                    iteration += 1
                    print(f'Training STOP {iteration}--------------------')
                    break  # Early stop

                iteration += 1  # Go to next Batch training iteration

            stop_timer = perf_counter()

            # Class Outputs append
            test_accuracies.append(test_accuracy) # in %
            training_times.append(stop_timer - start_timer) # in sec.
            print('-------------------------------------------------------------------')

        print(f'Classes MEAN ACCURACY: {np.mean(test_accuracies)}%')
        print(f'Classes Mean Training Runtime: {np.mean(training_times)}sec.')

        experiment_results[i, j, 0] = np.mean(test_accuracies)  # Average of 12 classes Accuracies
        experiment_results[i, j, 1] = np.mean(training_times)  # Average of 12 classes Training Times

        print('----------------------------------------------------------------------------------------------------')

# Outputs Save
np.savez_compressed(file='./dropout_grid_search',
                    avg_test_accuracies=experiment_results[:, :, 0],
                    avg_training_times=experiment_results[:, :, 1])


# DropOut Grid-Search Experiment Plot
experiment_data = np.load(file='./dropout_grid_search.npz')

plt.figure(figsize=(12, 8))
plt.suptitle('Dropout Regularization Experiment\n(average results across all classes)\n')

counter = 0
plt.subplot(2, 1, 1)
for i in range(experiment_data['avg_test_accuracies'].shape[0]):
    for j in range(experiment_data['avg_test_accuracies'].shape[1]):
        plt.bar(counter + 1, experiment_data['avg_test_accuracies'][i, j], label=f'DR_in: {dropout_rates_in[i]}, DR_hidden: {d
        counter += 1

max_accuracy = np.max(experiment_data['avg_test_accuracies'])
plt.axhline(max_accuracy, color='grey', linestyle='--')
plt.title(f'Test Accuracies, Max.: {max_accuracy:.2f}%')
plt.xlabel('Experiment Run (indices)')
plt.xticks([(n + 1) for n in range(10)], [(n + 1) for n in range(10)])
plt.ylabel('Accuracy (in %)')
plt.ylim([70, 100])
plt.grid()
plt.legend(loc='center right', bbox_to_anchor=(1.3, 0.5), fancybox=True, shadow=True, ncol=1)

counter = 0
plt.subplot(2, 1, 2)
for i in range(experiment_data['avg_training_times'].shape[0]):
    for j in range(experiment_data['avg_training_times'].shape[1]):
        plt.bar(counter + 1, experiment_data['avg_training_times'][i, j], label=f'DR_in: {dropout_rates_in[i]}, DR_hidden: {dr
        counter += 1

plt.title(f'Training Times')
plt.xlabel('Experiment Run (indices)')
```

```
plt.xticks([(n + 1) for n in range(10)], [(n + 1) for n in range(10)])
plt.ylabel('Time (in sec.)')
plt.ylim([20, 60])
plt.grid()
plt.legend(loc='center right', bbox_to_anchor=(1.3, 0.5), fancybox=True, shadow=True, ncol=1)

plt.tight_layout()
plt.savefig('dropout_grid_search')
plt.show()
```

```
# Best Test Accuracy inspection
best_run_acc = experiment_data['avg_test_accuracies'][0, 0]
best_run_time = experiment_data['avg_training_times'][0, 0]
print(f'Adam + DropOut: {best_run_acc:.2f}% in {best_run_time:.2f}sec. (per class)')
```

Batch Normalization

A form of regularization applied to layers inputs, in order to avoid covariance shift, vanishing or exploding gradients.

$$\hat{y} = \sigma(\tilde{x}^T w)$$

with

$$\tilde{x} = \gamma x + \beta$$

with γ and β respectively a scaling and shifting coefficient, learned by the model itself during training phase, while $\tilde{x}$ is a normalized "raw input" to the n-Layer.

- S. Ioffe, C. Szegedy (2015) - *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*

```
# Dynamic Multi-Layer Architecture Class (w. units, activation function, DropOut Rate specification)
class binaryClassifier_batchnorm(nn.Module):                                     # nn.Module: base class to inherit from
    def __init__(self, n_layers, n_units, act_fun, rate_in, rate_hidden):        # self + attributes (architecture hyper-para
        super().__init__()

        self.layers = nn.ModuleDict()                                            # Dictionary to store Model layers
        self.nLayers = n_layers                                                  # Class instance parameter

        # Input Layer
        self.layers['input'] = nn.Linear(3, n_units)                             # Key 'input' layer specification

        # Hidden Layers
        self.layers[f'hidden'] = nn.Linear(n_units, n_units)
        self.layers[f'batch_norm'] = nn.BatchNorm1d(n_units)

        # Output Layer
        self.layers['output'] = nn.Linear(n_units, 1)                            # Key 'output' layer specification

        # Activation Function
        self.actfun = act_fun                                                    # Function string-name attribute association

        # Dropout Parameter
        self.dr_in = rate_in
        self.dr_hidden = rate_hidden

        # Weights & Bias initialization
        for layer in self.layers.keys():
            try:
                nn.init.kaiming_normal_(self.layers[layer].weight, mode='fan_in') # Kaiming He - Normal Distributed (ReLU spec
            except:
                pass                                                             # Batch_norm Layer can't be initialized
        for layer in self.layers.keys():
            self.layers[layer].bias.data.fill_(0.)                               # Bias initialization (0.)

    # Forward Pass Method
    def forward(self, x):

        # Activation function object computation
        actfun = getattr(torch.nn, self.actfun)

        # Input Layer pass                                                       --> Weightening (Dot Product) "Linear transf
        x = actfun()(self.layers['input'](x))
        x = F.dropout(x, p=self.dr_in, training=self.training)                   # Activate DropOut only when Model Training

        # Hidden Layers sequential pass                                          --> Weightening (Dot Product) "Linear transf
        x = self.layers[f'batch_norm'](x)                                        # Apply batch normalization before hidden la
        x = actfun()(self.layers[f'hidden'](x))
        x = F.dropout(x, p=self.dr_hidden, training=self.training)               # Same as "Input pass"

        # Output Layer pass                                                      --> Output Weightening (Dot Product) "Linear
        x = self.layers['output'](x)
```

```
        x = nn.Sigmoid()(x)

        return x


# Experiment Parameters
epochs = 1000  # For each "Batch-Set"
iterations = 10  # A total of 10000 Epochs of Training (10x Batch-Sub-Dataset shuffling) --> w. Early Stopping
min_tolerance = 0.1 # ...for sub-dataset balancing

# Architecture Hyper-Parameters
hidden_layer = 1
hidden_nodes = 100
act_fun = 'ReLU'
learning_rates = [0.001, 0.0001, 0.00001]  # Try increasing and reducing actual LR
optimizer = 'Adam'

# Regularization Hyper-Parameters
dropout_rate_in = 0.8
dropout_rate_hidden = 0.5


# AVG. Time 40min.
from time import perf_counter
debug=False
experiment_results = np.zeros((len(learning_rates), 2))


for i in range(len(learning_rates)):
    print(f'Experiment {i + 1}: LR: {learning_rates[i]}')

    test_accuracies = []  # List of Classes Test Accuracies (Re-initialized for each experiment)
    training_times = []  # List of Classes Training Times (Re-initialized for each experiment)

    # Experiment Routine
    for k in range(len(vowels)):

        # Reset Seed
        torch.manual_seed(SEED)

        # Create Classifier
        binary_classifier = binaryClassifier_batchnorm(1, hidden_nodes, act_fun, dropout_rate_in, dropout_rate_hidden)

        # Iterated (w. Batch-Sets shuffling) Training
        iteration = 0

        start_timer = perf_counter()
        while iteration < iterations:

            # Dataset processing
            sub_data, sub_data_labels_bin, _ = one_hot_encoder(sel_class_number=k, dataset=x_data_minmax, debug=debug)
            print('----------------------------------')
            trainLoader, devLoader, testLoader = train_dev_test_split(sub_data[:, 1:], sub_data_labels_bin, [0.7, 0.15, 0.15],

            # Train/Test Architecture
            _, _, _, test_accuracy = mini_batch_train_test(binary_classifier, optimizer, epochs, learning_rates[i], trainLoade
            print(f'Sub-Net "{vowels[k]}" Partial-{iteration + 1} TEST ACCURACY: {test_accuracy:.2f}%')

            if test_accuracy > 93.86:  # If specific class instance overshot previous class mean accuracy
                iteration += 1
                print(f'Training STOP {iteration}--------------------')
                break  # Early stop

            iteration += 1  # Go to next Batch training iteration

        stop_timer = perf_counter()

        # Class Outputs append
        test_accuracies.append(test_accuracy) # in %
        training_times.append(stop_timer - start_timer) # in sec.
        print('---------------------------------------------------------------------')

    print(f'Classes MEAN ACCURACY: {np.mean(test_accuracies)}%')
    print(f'Classes Mean Training Runtime: {np.mean(training_times)}sec.')

    experiment_results[i, 0] = np.mean(test_accuracies)  # Average of 12 classes Accuracies
    experiment_results[i, 1] = np.mean(training_times)  # Average of 12 classes Training Times

    print('------------------------------------------------------------------------------------------')

# Outputs Save
np.savez_compressed(file='./batch_norm_lr',
```

```
                avg_test_accuracies=experiment_results[:, 0],
                avg_training_times=experiment_results[:, 1])


# Batch Normalization Experiment Plot
experiment_data = np.load(file='./batch_norm_lr.npz')

plt.figure(figsize=(12, 5))
plt.suptitle('Dropout + Batch-Norm Regularization Experiment\n(average results across all classes)\n')

counter = 0
plt.subplot(1, 2, 1)
for i in range(experiment_data['avg_test_accuracies'].shape[0]):
    plt.bar(i + 1, experiment_data['avg_test_accuracies'][i], label=f'LR: {learning_rates[i]}')
    counter += 1

max_accuracy = np.max(experiment_data['avg_test_accuracies'])
plt.axhline(max_accuracy, color='grey', linestyle='--')
plt.title(f'Test Accuracies, Max.: {max_accuracy:.2f}%')
plt.xlabel('Experiment Run (indices)')
plt.xticks([(n + 1) for n in range(3)], [(n + 1) for n in range(3)])
plt.ylabel('Accuracy (in %)')
plt.ylim([85, 97])
plt.grid()
plt.legend(loc='best')

plt.subplot(1, 2, 2)
for i in range(experiment_data['avg_training_times'].shape[0]):
    plt.bar(i + 1, experiment_data['avg_training_times'][i], label=f'LR: {learning_rates[i]}')

plt.title(f'Training Times')
plt.xlabel('Experiment Run (indices)')
plt.xticks([(n + 1) for n in range(3)], [(n + 1) for n in range(3)])
plt.ylabel('Time (in sec.)')
plt.ylim([40, 85])
plt.grid()
plt.legend(loc='best')

plt.tight_layout()
plt.savefig('batch_norm_lr')
plt.show()


# Best Test Accuracy inspection
best_run_acc = experiment_data['avg_test_accuracies'][1]
best_run_time = experiment_data['avg_training_times'][1]
print(f'Adam + DropOut + Batch-Norm: {best_run_acc:.2f}% in {best_run_time:.2f}sec. (per class)')
```

L2 (Ridge) Penalty

$L2$, also called "*Ridge regression*" or "*weight decay*" regularization it's expressed as:

$$J = \frac{1}{n}\sum_{i=1}^{n} L(\hat{y}_i, y_i) + \lambda ||w_i||_2^2$$

where: $||w_i||_2^2 = w^T w$

λ is a scalar coefficient, also called "regularization parameter/coefficient" and is usually expressed as:

$$\lambda = \frac{\alpha}{2m}$$

where m is the number of weights and $||w||$ represent the vector magnitude (norm) of weights.

Generally, we tend to prefer a relatively large value from the left term (the summation) and a relatively small value from the rgularization term in order to minimize cost function adding weights features itself.

[Wikipedia]

```
# Batch Training function (w. Adam Optimizer & L2 penalty term) Re-Definition
def mini_batch_train_test(model, weight_decay, epochs: int, learning_rate, train_loader, dev_loader, test_loader, debug=False)
    """
    Train & Test an ANN Architecture via Mini-Batch Training (w. Train/Dev/Test PyTorch Loaders) and Adam Backpropagation Opti
    """
    # Loss Function initialization
    loss_function = nn.BCELoss()

    # Optimizer Algorithm initialization
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=weight_decay)

    # Output list initialization
    train_accuracies = []
```

```
    train_losses = []
    dev_accuracies = []

    # TRAINING Phase
    for epoch in range(epochs):
        model.train()  # TRAINING Switch ON

        batch_accuracies = []
        batch_losses = []

        # Training BATCHES Loop
        for data_batch, labels_batch in train_loader:
            train_predictions = model(data_batch)
            train_loss = loss_function(train_predictions.squeeze(), labels_batch.type(torch.int64).float())
            batch_losses.append(train_loss.detach())

            # Backpropagation
            optimizer.zero_grad()
            train_loss.backward()
            optimizer.step()

            # Accuracy
            train_accuracy = 100 * torch.mean(((train_predictions.squeeze() > 0.5) == labels_batch.type(torch.int64).squeeze()

            # Batch Stats appending
            batch_accuracies.append(train_accuracy.detach())
            batch_losses.append(train_loss.detach())

        # Training Stats appending
        train_accuracies.append(np.mean(batch_accuracies))  # Average of Batch Accuracies = Training step accuracy
        train_losses.append(np.mean(batch_losses))  # Average of Batch Losses = Training step Losses

        # EVALUATION (Dev) Phase
        model.eval()
        with torch.no_grad():
            dev_data_batch, dev_labels_batch = next(iter(dev_loader))
            dev_predictions = model(dev_data_batch)

            dev_accuracy = 100 * torch.mean(((dev_predictions.squeeze() > 0.5) == dev_labels_batch.type(torch.int64).squeeze()

            if debug is True:
                if epoch % 100 == 0:
                    print(f'Epoch {epoch} --> DEV ACCURACY: {dev_accuracy.detach():.3f} %')
                    print('--------------------------------')

            # Evaluation accuracy appending
            dev_accuracies.append(dev_accuracy.detach())

    # TEST Phase
    model.eval()
    with torch.no_grad():
        test_data_batch, test_labels_batch = next(iter(test_loader))
        test_predictions = model(test_data_batch)
        test_accuracy = 100 * torch.mean(((test_predictions.squeeze() > 0.5) == test_labels_batch.type(torch.int64).squeeze())

        if debug is True:
            print(f'TEST ACCURACY: {test_accuracy.detach():.2f} %')
            print('------------------------------------------------------------------')

    return train_accuracies, train_losses, dev_accuracies, test_accuracy.detach()


# Experiment Parameters
epochs = 1000  # For each "Batch-Set"
iterations = 10  # A total of 10000 Epochs of Training (10x Batch-Sub-Dataset shuffling) --> w. Early Stopping
min_tolerance = 0.1 # ...for sub-dataset balancing

# Architecture Hyper-Parameters
hidden_layer = 1
hidden_nodes = 100
act_fun = 'ReLU'
learning_rate = 0.0001
optimizer = 'Adam'

# Regularization Hyper-Parameters
dropout_rate_in = 0.8
dropout_rate_hidden = 0.5
l2_lambda = np.logspace(-2, -4, num=3, base=10)  # [10^-2, 10^-3, 10^-4]


# AVG. Time 30min.
from time import perf_counter
debug=False
```

```python
experiment_results = np.zeros((len(l2_lambda), 2))


for i in range(len(l2_lambda)):
    print(f'Experiment {i + 1}: L2_Lambda (Weight Decay): {l2_lambda[i]}')

    test_accuracies = []  # List of Classes Test Accuracies (Re-initialized for each experiment)
    training_times = []  # List of Classes Training Times (Re-initialized for each experiment)

    # Experiment Routine
    for k in range(len(vowels)):

        # Reset Seed
        torch.manual_seed(SEED)

        # Create Classifier
        binary_classifier = binaryClassifier_batchnorm(1, hidden_nodes, act_fun, dropout_rate_in, dropout_rate_hidden)

        # Iterated (w. Batch-Sets shuffling) Training
        iteration = 0

        start_timer = perf_counter()
        while iteration < iterations:

            # Dataset processing
            sub_data, sub_data_labels_bin, _ = one_hot_encoder(sel_class_number=k, dataset=x_data_minmax, debug=debug)
            print('---------------------------------')
            trainLoader, devLoader, testLoader = train_dev_test_split(sub_data[:, 1:], sub_data_labels_bin, [0.7, 0.15, 0.15],

            # Train/Test Architecture
            _, _, _, test_accuracy = mini_batch_train_test(binary_classifier, l2_lambda[i], epochs, learning_rate, trainLoader
            print(f'Sub-Net "{vowels[k]}" Partial-{iteration + 1} TEST ACCURACY: {test_accuracy:.2f}%')

            if test_accuracy > 94.96:  # If specific class instance overshot previous class mean accuracy
                iteration += 1
                print(f'Training STOP {iteration}-------------------')
                break  # Early stop

            iteration += 1  # Go to next Batch training iteration

        stop_timer = perf_counter()

        # Class Outputs append
        test_accuracies.append(test_accuracy) # in %
        training_times.append(stop_timer - start_timer) # in sec.
        print('---------------------------------------------------------------------')

    print(f'Classes MEAN ACCURACY: {np.mean(test_accuracies)}%')
    print(f'Classes Mean Training Runtime: {np.mean(training_times)}sec.')

    experiment_results[i, 0] = np.mean(test_accuracies)  # Average of 12 classes Accuracies
    experiment_results[i, 1] = np.mean(training_times)  # Average of 12 classes Training Times

    print('----------------------------------------------------------------------------------------------------------')

# Outputs Save
np.savez_compressed(file='./L2_grid_search',
                avg_test_accuracies=experiment_results[:, 0],
                avg_training_times=experiment_results[:, 1])


# L2 Normalization Experiment Plot
experiment_data = np.load(file='./L2_grid_search.npz')

plt.figure(figsize=(12, 5))
plt.suptitle('Dropout + Batch-Norm + L2 Norm Experiment\n(average results across all classes)\n')

counter = 0
plt.subplot(1, 2, 1)
for i in range(experiment_data['avg_test_accuracies'].shape[0]):
    plt.bar(i + 1, experiment_data['avg_test_accuracies'][i], label=f'$\lambda$: {l2_lambda[i]}')
    counter += 1

max_accuracy = np.max(experiment_data['avg_test_accuracies'])
plt.axhline(max_accuracy, color='grey', linestyle='--')
plt.title(f'Test Accuracies, Max.: {max_accuracy:.2f}%')
plt.xlabel('Experiment Run (indices)')
plt.xticks([(n + 1) for n in range(3)], [(n + 1) for n in range(3)])
plt.ylabel('Accuracy (in %)')
plt.ylim([80, 100])
plt.grid()
plt.legend(loc='best')
```

```
plt.subplot(1, 2, 2)
for i in range(experiment_data['avg_training_times'].shape[0]):
    plt.bar(i + 1, experiment_data['avg_training_times'][i], label=f'$\lambda$: {l2_lambda[i]}')

plt.title(f'Training Times')
plt.xlabel('Experiment Run (indices)')
plt.xticks([(n + 1) for n in range(3)], [(n + 1) for n in range(3)])
plt.ylabel('Time (in sec.)')
plt.ylim([40, 70])
plt.grid()
plt.legend(loc='best')

plt.tight_layout()
plt.savefig('L2_grid_search')
plt.show()


# Best Test Accuracy inspection
best_run_acc = experiment_data['avg_test_accuracies'][2]
best_run_time = experiment_data['avg_training_times'][2]
print(f'Adam + DropOut + Batch-Norm + L2: {best_run_acc:.2f}% in {best_run_time:.2f}sec. (per class)')
```

OCON Model Analysis

(3-features all_speakers Dataset)

Author: S. Giacomelli

Year: 2023

Affiliation: A.Casella Conservatory (student)

Master Degree Thesis: "*Vowel phonemes Analysis & Classification by means of OCON rectifiers Deep Learning Architectures*"

Description: Python scripts for One-Class-One-Network (OCON) Model analysis and optimization

```
# Numerical computations packages/modules
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

# Dataset processing modules
from torch.utils.data import DataLoader
from sklearn.model_selection import train_test_split

# Graphic visualization modules
import matplotlib.pyplot as plt
import matplotlib_inline
matplotlib_inline.backend_inline.set_matplotlib_formats('svg')

# Common Seed initialization
SEED = 42  # ... the answer to the ultimate question of Life, the Universe, and Everything... (cit.)

# PyTorch Processing Units evaluation
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(f'AVILABLE Processing Unit: {device.upper()}')
!nvidia-smi
```

HGCW Dataset

- *Dataset_utils.npz* file read
- *One-Hot encoding* definition
- *Train/Dev/Test split* definition

```
# Load HGCW 4_features_all Dataset
HGCW_dataset_utils = np.load(file='./HGCW_dataset_utils.npz')
print('Raw features                    Data shape:', HGCW_dataset_utils['HGCW_raw'].shape)
print('Fundamental Normalized features Data shape:', HGCW_dataset_utils['HGCW_fund_norm'].shape)
print('MinMax features                 Data shape:', HGCW_dataset_utils['HGCW_minmax'].shape)
print('Labels                          Data shape:', HGCW_dataset_utils['HGCW_labels'].shape)
print('Classes size                    Data shape:', HGCW_dataset_utils['classes_size'].shape)
print('Classes indices                 Data shape:', HGCW_dataset_utils['classes_idx'].shape)

x_data_raw_np = HGCW_dataset_utils['HGCW_raw']
x_data_fund_norm = HGCW_dataset_utils['HGCW_fund_norm']
x_data_minmax = HGCW_dataset_utils['HGCW_minmax']
y_labels_raw_np = HGCW_dataset_utils['HGCW_labels']
vow_size = HGCW_dataset_utils['classes_size']
end_idx = HGCW_dataset_utils['classes_idx']

# Auxiliary lists
vowels = ['ae', 'ah', 'aw', 'eh', 'er', 'ei', 'ih', 'iy', 'oa', 'oo', 'uh', 'uw']  # Vowels list
colors = ['red', 'saddlebrown', 'darkorange', 'darkgoldenrod', 'gold', 'darkkhaki', 'olive', 'darkgreen', 'steelblue', 'fuchsi


# Class-specific One-hot encoding (Binarization)
def one_hot_encoder(sel_class_number: int = 3, dataset: np.ndarray = x_data_minmax, orig_labels: int = len(vowels), classes_si

    classes = [n for n in range(orig_labels)]  # Class Labels list initialization

    # Auxiliary Parameters Initialization
    if sel_class_number < len(classes):
        classes.remove(sel_class_number)  # REST Classes list
        if debug is True:
            print(f'Selected Class "{vowels[sel_class_number]}" : {classes_size[sel_class_number]} samples')
        sub_classes_size = classes_size[sel_class_number] // len(classes)
        if debug is True:
            print(f'Rest Classes size (...each): {sub_classes_size} samples')
```

```
        # 1-Subset processing
        sub_data = dataset[classes_idx[sel_class_number]: classes_idx[sel_class_number + 1], :]  # Selected Class feature slic
        sub_data_labels_bin = np.ones((classes_size[sel_class_number], 1), dtype='int')  # Selected Class labels (1) creation
        sub_data_labels = np.ones((classes_size[sel_class_number], 1), dtype='int') * sel_class_number

        # 0-Subset processing
        for i in classes:
            class_i_indices = np.random.choice(np.arange(classes_idx[i], classes_idx[i + 1], 1), size=sub_classes_size, replac
            sub_class_i_array = dataset[class_i_indices, :]
            sub_class_labels_bin_array = np.zeros((sub_class_i_array.shape[0], 1), dtype='int')  # Rest I-esimal Class labels
            sub_class_labels_array = np.ones((sub_class_i_array.shape[0], 1), dtype='int') * i

            # Outputs append
            sub_data = np.vstack((sub_data, sub_class_i_array))
            sub_data_labels_bin = np.vstack((sub_data_labels_bin, sub_class_labels_bin_array))
            sub_data_labels = np.vstack((sub_data_labels, sub_class_labels_array))
    else:
        raise ValueError(f'Invalid Class ID: "{sel_class_number}" --> It must be less than {len(classes)}!')

    return sub_data, sub_data_labels_bin, sub_data_labels


# Train/Test split (auxiliary function)
def train_test_split_aux(features_dataset, labels_dataset, test_perc, tolerance):
    """
    An auxiliary Train_Test_split function (based on Scikit Learn implementation) w. balance tolerance specification
    """
    test_size = int(test_perc / 100 * len(features_dataset))
    train_balance = 0  # Output Training set balance value initialization
    test_balance = 0  # Output Testing set balance value initialization

    min_tol = np.mean(labels_dataset) - tolerance
    max_tol = np.mean(labels_dataset) + tolerance
    print(f'Data Balancing  (TARGET = {np.mean(labels_dataset)} +- {tolerance}): ', end='')

    while (min_tol >= train_balance or train_balance >= max_tol) or (min_tol >= test_balance or test_balance >= max_tol):
        train_data, test_data, train_labels, test_labels = train_test_split(features_dataset, labels_dataset, test_size=test_s
        train_balance = np.mean(train_labels)
        test_balance = np.mean(test_labels)
        print('.', end='')
    else:
        print('OK')

    return train_data, test_data, train_labels, test_labels, train_balance, test_balance

# Train-Dev-Test split function
def train_dev_test_split(x_data, y_labels, split_list, tolerance=0.1, output='Loaders', debug=False):
    """
    Compute a Train, Development (Hold-Out) and a Test set split w. PyTorch Dataset conversion (and eventual Loaders initializ
    """
    if len(split_list) == 3:
        # Train - Dev+Test separation
        print('Training --- Devel/Test SPLIT')
        train_data, testTMP_data, train_labels, testTMP_labels, _, _ = train_test_split_aux(x_data, y_labels, (split_list[1] *
        print('----------------------------------')

        # Dev - Test separation
        print('Devel    ---     Test SPLIT')

        split = ((split_list[1] * 100) / np.sum(split_list[1:] * 100)) * 100  # Split in %
        dev_data, test_data, dev_labels, test_labels, _, _ = train_test_split_aux(testTMP_data, testTMP_labels, split, toleran
        print('----------------------------------')

        # Tensor Conversion
        train_data_tensor = torch.tensor(train_data).float()
        train_labels_tensor = torch.tensor(train_labels, dtype=torch.int64).squeeze()
        dev_data_tensor = torch.tensor(dev_data).float()
        dev_labels_tensor = torch.tensor(dev_labels, dtype=torch.int64).squeeze()
        test_data_tensor = torch.tensor(test_data).float()
        test_labels_tensor = torch.tensor(test_labels, dtype=torch.int64).squeeze()
        if debug is True:
            print(f'Training Data        Shape: {train_data.shape}')
            print(f'Development Data     Shape: {dev_data.shape}')
            print(f'Testing Data         Shape: {test_data.shape}')

            # Balance Evaluation
            print(f'Training Set       Balance: {np.mean(train_labels)}')
            print(f'Development Set    Balance: {np.mean(dev_labels)}')
            print(f'Testing Set        Balance: {np.mean(test_labels)}')

        if output != 'Loaders':
            return train_data_tensor, train_labels_tensor, dev_data_tensor, dev_labels_tensor, test_data_tensor, test_labels_t
```

```
        else:
            # PyTorch Dataset Conversion
            train_dataset = torch.utils.data.TensorDataset(torch.tensor(train_data).float(), torch.tensor(train_labels, dtype=
            dev_dataset = torch.utils.data.TensorDataset(torch.tensor(dev_data).float(), torch.tensor(dev_labels, dtype=torch.
            test_dataset = torch.utils.data.TensorDataset(torch.tensor(test_data).float(), torch.tensor(test_labels, dtype=tor

            # DataLoader (Batches) --> Drop-Last control to optimize training
            trainLoader = DataLoader(train_dataset, shuffle=False, batch_size = 32, drop_last=True)
            devLoader = DataLoader(dev_dataset, shuffle=False, batch_size = dev_dataset.tensors[0].shape[0])
            testLoader = DataLoader(test_dataset, shuffle=False, batch_size = test_dataset.tensors[0].shape[0])
            if debug is True:
                print(f'Training Set    Batch Size: {trainLoader.batch_size}')
                print(f'Development Set Batch Size: {devLoader.batch_size}')
                print(f'Testing Set     Batch Size: {testLoader.batch_size}')

            return trainLoader, devLoader, testLoader
    else:
        # Train - Test separation
        print('Training --- Test    SPLIT')
        train_data, test_data, train_labels, test_labels, _, _ = train_test_split_aux(x_data, y_labels, split_list[1] * 100, t
        print('--------------------------')

        # Tensor Conversion
        train_data_tensor = torch.tensor(train_data).float()
        train_labels_tensor = torch.tensor(train_labels, dtype=torch.int64).squeeze()
        test_data_tensor = torch.tensor(test_data).float()
        test_labels_tensor = torch.tensor(test_labels, dtype=torch.int64).squeeze()
        if debug is True:
            print(f'Training Data        Shape: {train_data.shape}')
            print(f'Testing Data         Shape: {test_data.shape}')

            # Balance Evaluation
            print(f'Training Set    Balance: {np.mean(train_labels)}')
            print(f'Testing Set     Balance: {np.mean(test_labels)}')

        if output != 'Loaders':
            return train_data_tensor, train_labels_tensor, test_data_tensor, test_labels_tensor
        else:
            # PyTorch Dataset Conversion
            train_dataset = torch.utils.data.TensorDataset(torch.tensor(train_data).float(), torch.tensor(train_labels, dtype=
            test_dataset = torch.utils.data.TensorDataset(torch.tensor(test_data).float(), torch.tensor(test_labels, dtype=tor

            # DataLoader (Batches) --> Drop-Last control to optimize training
            trainLoader = DataLoader(train_dataset, shuffle=False, batch_size = 32, drop_last=True)
            testLoader = DataLoader(test_dataset, shuffle=False, batch_size = test_dataset.tensors[0].shape[0])
            if debug is True:
                print(f'Training Set    Batch Size: {trainLoader.batch_size}')
                print(f'Testing Set     Batch Size: {testLoader.batch_size}')

            return trainLoader, testLoader
```

One-Class Architecture (Binary Classifier)

(see "*One-Class_Sub-Network_Analysis.ipynb*")

```
Multi-Layer Perceptron
- Input Layer: 3 features [formant ratios, min-max normalized]
- Hidden Layer: 100 units
- Output Layer: 1 normalized probability
- Learning Rate: 0.0001 (10^-4)
- Optimizer: Adam (Adaptive Momentum)

- Mini-Batch Training:
    . Re-iterated Sub-Dataset Shuffling
    . Early Stopping (Test Accuracy driven)
    . Batch size = 32

- Regularization:
    . Weight Decay (L2 Penalty): 0.0001 (10^-4)
    . DropOut:
        * Input Layer Drop Rate: 0.8
        * Hidden Layer Drop Rate: 0.5.
    . Batch Normalization
```

- *MLP Classifier Architecture* class definition
- *Mini-Batch Training* function definition

```
# Dynamic Multi-Layer Architecture Class (w. units, activation function, batch normalization and dropOut rate specification)
class binaryClassifier(nn.Module):                                                  # nn.Module: base class to inherit from
    def __init__(self, n_units, act_fun, rate_in, rate_hidden, model_name):                    # self + attributes (architectur
        super().__init__()

        self.layers = nn.ModuleDict()                                               # Dictionary to store Model layers
        self.name = model_name

        # Input Layer
        self.layers['input'] = nn.Linear(3, n_units)                                # Key 'input' layer specification

        # Hidden Layer
        self.layers[f'hidden'] = nn.Linear(n_units, n_units)
        self.layers[f'batch_norm'] = nn.BatchNorm1d(n_units)

        # Output Layer
        self.layers['output'] = nn.Linear(n_units, 1)                               # Key 'output' layer specification

        # Activation Function
        self.actfun = act_fun                                                       # Function string-name attribute association

        # Dropout Parameter
        self.dr_in = rate_in
        self.dr_hidden = rate_hidden

        # Weights & Bias initialization
        for layer in self.layers.keys():
            try:
                nn.init.kaiming_normal_(self.layers[layer].weight, mode='fan_in') # Kaiming He - Normal Distributed (ReLU spec
            except:
                pass                                                                # Batch_norm Layer can't be initialized
            self.layers[layer].bias.data.fill_(0.)                                  # Bias initialization (0.)

    # Forward Pass Method
    def forward(self, x):

        # Activation function object computation
        actfun = getattr(torch.nn, self.actfun)

        # Input Layer pass                                                          --> Weightening (Dot Product) "Linear transf
        x = actfun()(self.layers['input'](x))
        x = F.dropout(x, p=self.dr_in, training=self.training)                      # Activate DropOut only when Model Training

        # Single Hidden Layer pass                                                  --> Weightening (Dot Product) "Linear transf
        x = self.layers[f'batch_norm'](x)                                           # Apply batch normalization before hidden la
        x = actfun()(self.layers[f'hidden'](x))
        x = F.dropout(x, p=self.dr_hidden, training=self.training)                  # Same as "Input pass"

        # Output Layer pass                                                         --> Output Weightening (Dot Product) "Linear
        x = self.layers['output'](x)
        x = nn.Sigmoid()(x)

        return x


# Batch Training function (w. Adam Optimizer & L2 penalty term) Re-Definition
def mini_batch_train_test(model, weight_decay, epochs: int, learning_rate, train_loader, dev_loader, test_loader, debug=False)
    """
    Train & Test an ANN Architecture via Mini-Batch Training (w. Train/Dev/Test PyTorch Loaders) and Adam Backpropagation Opti
    """
    # Loss Function initialization
    loss_function = nn.BCELoss()

    # Optimizer Algorithm initialization
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=weight_decay)

    # Output list initialization
    train_accuracies = []
    train_losses = []
    dev_accuracies = []

    # TRAINING Phase
    for epoch in range(epochs):
        model.train()  # TRAINING Switch ON

        batch_accuracies = []
        batch_losses = []

        # Training BATCHES Loop
        for data_batch, labels_batch in train_loader:
            train_predictions = model(data_batch)
            train_loss = loss_function(train_predictions.squeeze(), labels_batch.type(torch.int64).float())
```

```
            batch_losses.append(train_loss.detach())

            # Backpropagation
            optimizer.zero_grad()
            train_loss.backward()
            optimizer.step()

            # Accuracy
            train_accuracy = 100 * torch.mean(((train_predictions.squeeze() > 0.5) == labels_batch.type(torch.int64).squeeze()

            # Batch Stats appending
            batch_accuracies.append(train_accuracy.detach())
            batch_losses.append(train_loss.detach())

        # Training Stats appending
        train_accuracies.append(np.mean(batch_accuracies))  # Average of Batch Accuracies = Training step accuracy
        train_losses.append(np.mean(batch_losses))  # Average of Batch Losses = Training step Losses

        # EVALUATION (Dev) Phase
        model.eval()
        with torch.no_grad():
            dev_data_batch, dev_labels_batch = next(iter(dev_loader))
            dev_predictions = model(dev_data_batch)

            dev_accuracy = 100 * torch.mean(((dev_predictions.squeeze() > 0.5) == dev_labels_batch.type(torch.int64).squeeze()

            if debug is True:
                if epoch % 100 == 0:
                    print(f'Epoch {epoch} --> DEV ACCURACY: {dev_accuracy.detach():.3f} %')
                    print('--------------------------------')

            # Evaluation accuracy appending
            dev_accuracies.append(dev_accuracy.detach())

    # TEST Phase
    model.eval()
    with torch.no_grad():
        test_data_batch, test_labels_batch = next(iter(test_loader))
        test_predictions = model(test_data_batch)
        test_accuracy = 100 * torch.mean(((test_predictions.squeeze() > 0.5) == test_labels_batch.type(torch.int64).squeeze())

        if debug is True:
            print(f'TEST ACCURACY: {test_accuracy.detach():.2f} %')
            print('--------------------------------------------------------------------')

    return train_accuracies, train_losses, dev_accuracies, test_accuracy.detach()
```

OCON (One-Class-One-Net) Model

Binary classifiers **Parallelization**

- *Classifiers-Bank* function definition
    - *Models Parameters* inspection
- Classifiers **Sequential** Training & Evaluation
- *Models Parameters State Save/Load* function definition

- MaxNet output algorithm
- Argmax output algorithm

```
def OCON_bank(one_class_function, hidden_units, act_fun, dr_in, dr_hidden, classes_list):
    """
    Create a One-Class-One-Network parallelization bank of an input Sub-Network definition
    """
    # Sub-Net names creation
    models_name_list = []
    for i in range(len(classes_list)):
        models_name_list.append("{}_{}".format(classes_list[i], "subnet"))  # Class name + _subnet

    # Sub-Networks instances creation
    sub_nets = []  # Sub Network list initialization

    for i in range(len(models_name_list)):

        torch.manual_seed(SEED)  # Seed re-initialization

        # Sub-Net instance creation
        locals()[models_name_list[i]] = one_class_function(hidden_units, act_fun, dr_in, dr_hidden, models_name_list[i])
        sub_nets.append(locals()[models_name_list[i]])
```

```
    return sub_nets


# Load Architecture Parameters State function
def load_model_state(model, state_dict_path):
    """
    Load an existent State Dictionary in a defined model
    """

    model.load_state_dict(torch.load(state_dict_path))
    print(f'Loaded Parameters (from "{state_dict_path}") into: {model.name}')

    return model


# Build The OCON Model
ocon_vowels = OCON_bank(binaryClassifier, 100, 'ReLU', 0.8, 0.5, vowels)  # Best MLP (see "One-Class_Binary_Classifier_Analysi

# Load Pre-Trained Architectures in a fresh Model instance:
#ocon_vowels = OCON_bank(binaryClassifier, 100, 'ReLU', 0.8, 0.5, vowels)  # Best MLP (see "One-Class_Binary_Classifier_Analys
#states_path = ["Trained_models_state/ae_subnet_Params.pth",
#               "Trained_models_state/ah_subnet_Params.pth",
#               "Trained_models_state/aw_subnet_Params.pth",
#               "Trained_models_state/eh_subnet_Params.pth",
#               "Trained_models_state/er_subnet_Params.pth",
#               "Trained_models_state/ei_subnet_Params.pth",
#               "Trained_models_state/ih_subnet_Params.pth",
#               "Trained_models_state/iy_subnet_Params.pth",
#               "Trained_models_state/oa_subnet_Params.pth",
#               "Trained_models_state/oo_subnet_Params.pth",
#               "Trained_models_state/uh_subnet_Params.pth",
#               "Trained_models_state/uw_subnet_Params.pth"]
#
#for i in range(len(ocon_vowels)):
#    load_model_state(ocon_vowels[i], states_path[i])


# OCON Evaluation function
def OCON_eval(ocon_models_bank, features_dataset: np.ndarray = x_data_minmax[:, 1:], labels: np.ndarray = y_labels_raw_np):
    """
    Evaluate OCON models-bank over an entire dataset
    """
    # Output lists initialization
    predictions = []
    dist_errors = []
    eval_accuracies = []
    g_truths = []  # For plotting purpouses

    # Evaluate each Sub-Network...
    for i in range(len(ocon_models_bank)):
        ocon_models_bank[i].eval()  # Put j-esimal Sub-Network in Evaluation Mode
        print(f'{ocon_models_bank[i].name.upper()} Evaluation -', end=' ')

        with torch.no_grad():

            # Make predictions
            features_data_tensor = torch.tensor(features_dataset).float()
            raw_eval_predictions = ocon_models_bank[i](features_data_tensor)

            # Create Ground Truths
            ground_truth = np.where(labels == i, 1, 0)
            ground_truth_tensor = torch.tensor(ground_truth, dtype=torch.int64).squeeze()

            # Compute Errors
            dist_error = ground_truth_tensor - raw_eval_predictions.detach().squeeze()  # Distances
            eval_accuracy = 100 * torch.mean(((raw_eval_predictions.detach().squeeze() > 0.5) == ground_truth_tensor).float())
            print(f'Accuracy: {eval_accuracy:.2f}%')

        # Outputs append
        predictions.append(raw_eval_predictions.detach())
        dist_errors.append(dist_error.detach())
        eval_accuracies.append(eval_accuracy.detach())

        g_truths.append(ground_truth)

    return predictions, dist_errors, eval_accuracies, g_truths

# For Pre-Trained Models
#ocon_predictions, ocon_dist_errors, ocon_eval_accuracies, ocon_g_truths = OCON_eval(ocon_vowels)


# Model Parameters State function
def model_desc(model):
```

```
    """
    Print a Console report of Neural Network Model parameters
    """
    # Parameters Description
    print('Params Description:')
    trainable_params = 0

    for parameter in model.named_parameters():
        print(f'Parameter Name      : {parameter[0]}')
        print(f'Parameter Weights   : {parameter[1][:]}')
        if parameter[1].requires_grad:
            print(f'...with {parameter[1].numel()} TRAINABLE parameters')
            trainable_params += parameter[1].numel()

        print('................................')

    print('----------------------------------------------------------------')

    # Nodes Count
    nodes = 0
    for param_name, param_tensor in model.named_parameters():
        if 'bias' in param_name:
            nodes += len(param_tensor)

    print(f'Total Nodes             : {nodes}')
    print('----------------------------------------------------------------')

# OCON-Model Description
for i in range(len(ocon_vowels)):
    print(f'OCON "{ocon_vowels[i].name}" Classifier STATE')
    model_desc(ocon_vowels[i])
    print()


# Training/Eval/Testing Parameters
epochs = 1000  # For each "Data Batch-Set"
loss_break = 0.20  # loss (for Early Stopping)
acc_break = 90.  # % accuracy (for Early Stopping)
min_tolerance = 0.01 # ...for sub-dataset balancing

# Outputs Initialization
loss_functions = [[] for _ in range(len(ocon_vowels))]
training_accuracies = [[] for _ in range(len(ocon_vowels))]
evaluation_accuracies = [[] for _ in range(len(ocon_vowels))]
test_accuracies = [[] for _ in range(len(ocon_vowels))]
training_times = []


# OCON Sub-Networks Training
from time import perf_counter
debug = False

for i, vowel in enumerate(vowels):
    print(f'Architecture "{ocon_vowels[i].name}" TRAINING PHASE')

    start_timer = perf_counter()
    # Iterated (w. Batch-Sets shuffling) Mini-Batch Training
    iteration = 0  # Batch Training iteration counter
    mean_loss = 1.
    test_accuracy = 0.

    while (mean_loss > loss_break) or (test_accuracy < acc_break):
        # Dataset processing
        sub_data, sub_data_labels_bin, _ = one_hot_encoder(sel_class_number=i, dataset=x_data_minmax, debug=debug)
        print('---------------------------------')
        trainLoader, devLoader, testLoader = train_dev_test_split(sub_data[:, 1:], sub_data_labels_bin, [0.5, 0.25, 0.25], tol

        # Train/Test Architecture
        train_accuracies, train_losses, dev_accuracies, test_accuracy = mini_batch_train_test(ocon_vowels[i], weight_decay=0.0
        print(f'Sub-Net "{vowel.upper()}" Epoch {(iteration + 1) * epochs} - TEST ACCURACY: {test_accuracy:.2f}%', end=' ')

        # Outputs append
        loss_functions[i].append(train_losses)
        training_accuracies[i].append(train_accuracies)
        evaluation_accuracies[i].append(dev_accuracies)
        test_accuracies[i].append(test_accuracy)

        # Repeating condition evaluation
        mean_loss = np.mean(train_losses[-50: ])  # Last 100 losses mean
        print(f'- MEAN LOSS: {mean_loss}')

        iteration += 1  # Go to next Batch training iteration
```

```
    print(f'Training STOPPED at iteration {iteration}')
    print('---------------------------------------------------------------------')
    stop_timer = perf_counter()

    print(f'"{ocon_vowels[i].name}" Training COMPLETED in {float(stop_timer - start_timer)}sec.')
    training_times.append(stop_timer - start_timer)
    print('---------------------------------------------------------------------')


# Graphical smoothing filter
def smooth(data, k=100):
    """
    A Convolution LP filter w. interval definition
    """
    return np.convolve(data, np.ones(k) / k, mode='same')


# Training Phase Plots
plt.figure(figsize=(12, 5 * 12))

# loss_functions, training_accuracies, evaluation_accuracies, test_accuracies, training_times
classes = len(ocon_vowels)

for i in range(classes):
    plt.subplot(classes, 2, (i * 2) + 1)
    flat_loss_function = [item for sublist in loss_functions[i] for item in sublist]

    plt.plot(smooth(flat_loss_function), 'k-')
    plt.axhline(loss_break, color='r', linestyle='--')
    plt.title(f'{ocon_vowels[i].name.upper()} Training Loss')
    plt.xlabel('Epochs')
    plt.xlim([100, len(flat_loss_function) - 100])
    plt.ylabel('GT - Predicted diff. (probability)')
    plt.grid()

    plt.subplot(classes, 2, (i * 2) + 2)
    flat_training_accuracy = [item for sublist in training_accuracies[i] for item in sublist]
    flat_dev_accuracy = [item for sublist in evaluation_accuracies[i] for item in sublist]
    flat_test_accuracy = test_accuracies[i]

    plt.plot(smooth(flat_training_accuracy), 'k-', label='Training')
    plt.plot(smooth(flat_dev_accuracy), color='grey', label='Development')
    if len(flat_test_accuracy) > 1:
        plt.plot([(n + 1) * epochs for n in range(len(flat_test_accuracy))], flat_test_accuracy, 'r-', label=f'Test')
    else:
        plt.axhline(test_accuracy, color='r', linestyle='-', label=f'Test')
    plt.title(f'{ocon_vowels[i].name.upper()} Accuracy (after {training_times[i]:.2f}sec.)')
    plt.xlabel('Epochs')
    plt.xlim([100, len(flat_training_accuracy) - 100])
    plt.ylabel('Accuracy (in %)')
    plt.ylim([40, 101])
    plt.grid()
    plt.legend(loc='best')

plt.tight_layout()
plt.savefig('OCON_training_phase')
plt.show()


# OCON Evaluation
ocon_predictions, ocon_dist_errors, ocon_eval_accuracies, ocon_g_truths = OCON_eval(ocon_vowels)


# Dataset Evaluation Analysis Plot
plt.figure(figsize=(22, 5 * len(ocon_vowels)))
plot_ticks = end_idx[:]
plot_ticks = np.delete(plot_ticks, -1)

for i in range(len(ocon_vowels)):
    plt.subplot(len(ocon_vowels), 3, (i * 3) + 1)
    plt.plot(ocon_predictions[i], 'k.', label='Raw Predictions')
    plt.plot(ocon_g_truths[i], 'rx', label='Ground Truths')
    plt.axhline(0.5, linestyle='--', color='grey')
    plt.title(f'{ocon_vowels[i].name.upper()} Predictions Accuracy: {ocon_eval_accuracies[i]:.2f}%')
    plt.xlabel('Data (Indices)')
    plt.xticks(ticks=plot_ticks, labels=vowels)
    plt.ylabel('Normalized Probability')
    plt.grid()
    plt.legend(loc='best')

    plt.subplot(len(ocon_vowels), 3, (i * 3) + 2)
    plt.plot(ocon_dist_errors[i], 'k')
    plt.title(f'Predicted to Measured Error')
    plt.xlabel('Data (Indices)')
```

```
    plt.xticks(ticks=plot_ticks, labels=vowels)
    plt.ylabel('Normalized Probability Error')
    plt.ylim([-1.1, 1.1])
    plt.grid()

    plt.subplot(len(ocon_vowels), 3, (i * 3) + 3)

    # Predictions list processing
    predictions_temp = ocon_predictions[i]
    class_predictions = [item for sublist in predictions_temp for item in sublist]  # Turn a list of lists in a single list (c
    for j in range(len(class_predictions)):  # Turn a list of tensors of one variable in a list of scalars (item() method)
        class_predictions[j] = class_predictions[j].item()

    # Positives & False-Positives extraction
    positives = []
    for w in range(len(vowels)):
        num = (np.array(class_predictions[end_idx[w]: end_idx[w + 1]]) > 0.5).sum()
        positives.append(num)

    plt.bar(np.arange(len(vowels)), positives, color='k')
    plt.title(f'"{vowels[i]}" Positive Probabilities Distribution')
    plt.xlabel('Normalized Probabilities')
    plt.ylabel('Occurences')
    plt.xticks([n for n in range(12)], vowels)
    plt.grid()

plt.tight_layout()
plt.savefig('OCON_bank_evaluation')
plt.show()


# Model Parameters Save/Load functions
from pathlib import Path

def save_model_state(model, folder_name: str = "Trained_models_state"):
    """
    Save Pre-Trained model parameters in a State Dictionary
    """

    MODEL_PATH = Path(folder_name)  # Placed in root
    MODEL_PATH.mkdir(parents=True, exist_ok=True)  # Pre-existing folder (w. same name) monitoring
    MODEL_NAME = '{}_{}'.format(model.name, "Params.pth")
    MODEL_SAVE_PATH = MODEL_PATH / MODEL_NAME

    print(f"Saving {model.name} Parameters in: {MODEL_SAVE_PATH}")
    torch.save(obj=model.state_dict(), f=MODEL_SAVE_PATH)

    return MODEL_SAVE_PATH

# Save Pre-Trained Models-bank
states_path = []  # Path for each model parameters state
for i in range(len(ocon_vowels)):
    state_path = save_model_state(ocon_vowels[i])
    states_path.append(state_path)
print()
```

Output **Maxnet Algorithm**

```
# OCON "MaxNet" Architecture (Weightening + Non Linearity apply)
class OCON_MaxNet(nn.Module):                                            # nn.Module: base class to inherit from
    def __init__(self, n_units, act_fun, eps):                                # self + attributes (architecture hyper-paramet
        super().__init__()

        self.layers = nn.ModuleDict()                                    # Dictionary to store Model layers
        self.eps_weight = eps

        # MaxNet Layer
        self.layers['MAXNET'] = nn.Linear(n_units, n_units)              # Key 'MaxNet' layer specification

        # Weights & Bias initialization
        self.layers['MAXNET'].weight.data.fill_(self.eps_weight)
        for i in range(n_units):
            self.layers['MAXNET'].weight[i][i].data.fill_(1.)  # Self Weight = 1

        self.layers['MAXNET'].bias.data.fill_(0.)

        # Activation Function
        self.actfun = act_fun  # Function string-name attribute association

    # Forward Pass Method
```

```
    def forward(self, x):

        # Activation function object computation
        actfun = getattr(torch.nn, self.actfun)

        # Maxnet Layer pass                                              --> Output Weightening (Dot Product) "Linear transfo
        x = actfun()(self.layers['MAXNET'](x.squeeze().float()))

        # Self

        return x


# Build OCON MaxNetwork Architecture
torch.manual_seed(SEED)
ocon_maxnet = OCON_MaxNet(n_units=12, act_fun='ReLU', eps=0.25)

# MaxNet & Sub-Networks Parameters
print('OCON MaxNet STATE')
model_desc(ocon_maxnet)


def maxnet_algo(maxnet_function, n_units, act_fun, eps, input_array):
    """
    MaxNet Re-iteration algorithm for Maximum Value retrieving from an input array
    """
    non_zero_outs = np.count_nonzero(input_array)  # Non Zero Values initialization
    maxnet_in = torch.from_numpy(input_array)  # MaxNet Input Tensor initialization

    results = []  # Results initialization

    counter = 0
    while non_zero_outs != 1:
        counter += 1

        # Create the MaxNet
        torch.manual_seed(SEED)  # Redundant
        maxnet = maxnet_function(n_units = n_units, act_fun = act_fun, eps = eps)

        # Compute Forward Pass
        results = maxnet(maxnet_in)

        # Non_zero outputs & Maxnet Input Update
        non_zero_outs = np.count_nonzero(results.detach().numpy())
        maxnet_in = results.detach()  # Save results for next iteration

    print(f'Maximum Value found in {counter} iterations')
    return np.argmax(results.detach().numpy())


# MaxNet on Sub-Networks predictions
ocon_predictions_prob = np.zeros((len(ocon_predictions), x_data_minmax.shape[0]))  # NumPy predictions matrix (12 * 1617)

# Convert from List of Tensors to 2D NumPy Array
for i in range(len(ocon_predictions)):
    ocon_predictions_prob[i, :] = ocon_predictions[i].detach().squeeze().numpy()

maxnet_class_predictions = []  # Classes Outputs list initialization
# MaxNet application
for i in range(x_data_minmax.shape[0]):
    print(f'Dataset Sample({i + 1}) Class Evaluation')

    samp_predictions = ocon_predictions_prob[:, i]  # Array of 12 predictions for each Dataset sample (OCON outputs)
    class_prediction = maxnet_algo(OCON_MaxNet, n_units=12, act_fun='ReLU', eps=-0.1, input_array=samp_predictions)  # MaxNet
    maxnet_class_predictions.append(class_prediction)  # Result appending
    print('----------------------------------------------')

maxnet_accuracy = 100 * np.mean((np.array(maxnet_class_predictions).reshape(1617, 1) == y_labels_raw_np))  # Accuracy computat
print(f'Maxnet Output --> Phoneme ACCURACY: {maxnet_accuracy}%')


# Argmax on Sub-Networks predictions (...for multiple 1s probabilities MaxNet infinite loops)
#ocon_predictions_prob = np.zeros((len(new_ocon_predictions), x_data_minmax.shape[0]))  # NumPy predictions matrix (12 * 1617)
#
# Convert from List of Tensors to 2D NumPy Array
#for i in range(len(new_ocon_predictions)):
#    ocon_predictions_prob[i, :] = new_ocon_predictions[i].detach().squeeze().numpy()
#
#maxnet_class_predictions = np.argmax(ocon_predictions_prob, axis=0)
#maxnet_accuracy = 100 * np.mean((np.array(maxnet_class_predictions).reshape(1617, 1) == y_labels_raw_np))  # Accuracy computa
#print(f'Maxnet Output ACCURACY: {maxnet_accuracy}%')
```

```
# Evaluation Analysis Plot
plt.figure(figsize=(30, 5))
plt.suptitle(f'OCON Bank + MaxNet Evaluation: {maxnet_accuracy:.0f}%')

plot_x_ticks = end_idx[:]
plot_x_ticks = np.delete(plot_x_ticks, len(end_idx) - 1)
plot_y_ticks = [n for n in range(len(vowels))]

plt.plot(y_labels_raw_np, 'rs', label='Ground Truths')
plt.plot(maxnet_class_predictions, 'k.', label='MAXNET Outputs')
plt.xlabel('Dataset samples')
plt.xticks(ticks=plot_x_ticks, labels=vowels)
plt.xlim([-10, len(y_labels_raw_np) + 10])
plt.ylabel('Labels')
plt.yticks(ticks=plot_y_ticks, labels=vowels)
plt.legend(loc='best')
plt.grid()

plt.tight_layout()
plt.savefig('OCON_model_evaluation')
plt.show()
```

Further improvements

> 1) Repeat Training Cycle in order to increase Loss minimization $(< 0.1, < 0.05)$
>
> 2) Other Datasets Evaluation
>
> 3) Ratios multi-resolution transform (Bark, ERB, Mel)
>
> 4) Dataset Augmentation via Pitch Shifting: mean fund, mean ratios + artifical randomness (variance)

> 1) Spectral features (HGCW dataset)

OCON Model Analysis

(4-features Complete Dataset)

Author: S. Giacomelli

Year: 2023

Affiliation: A.Casella Conservatory (student)

Master Degree Thesis: "*Vowel phonemes Analysis & Classification by means of OCON rectifiers Deep Learning Architectures*"

Description: Python scripts for One-Class-One-Network (OCON) Model analysis and optimization

```
# Numerical computations packages/modules
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

# Dataset processing modules
from torch.utils.data import DataLoader
from sklearn.model_selection import train_test_split

# Graphic visualization modules
import matplotlib.pyplot as plt
import matplotlib_inline
matplotlib_inline.backend_inline.set_matplotlib_formats('svg')

# Common Seed initialization
SEED = 42  # ... the answer to the ultimate question of Life, the Universe, and Everything... (cit.)

# PyTorch Processing Units evaluation
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(f'AVILABLE Processing Unit: {device.upper()}')
!nvidia-smi
```

HGCW Dataset

- *Dataset_utils.npz* file read
- *One-Hot encoding* definition
- *Train/Dev/Test split* definition

```
# Load HGCW 4_features_all Dataset
HGCW_dataset_utils = np.load(file='./HGCW_dataset_utils.npz')
print('Raw features                       Data shape:', HGCW_dataset_utils['HGCW_raw'].shape)
print('Fundamental Normalized features Data shape:', HGCW_dataset_utils['HGCW_fund_norm'].shape)
print('MinMax features                    Data shape:', HGCW_dataset_utils['HGCW_minmax'].shape)
print('Labels                             Data shape:', HGCW_dataset_utils['HGCW_labels'].shape)
print('Classes size                       Data shape:', HGCW_dataset_utils['classes_size'].shape)
print('Classes indices                    Data shape:', HGCW_dataset_utils['classes_idx'].shape)

x_data_raw_np = HGCW_dataset_utils['HGCW_raw']
x_data_fund_norm = HGCW_dataset_utils['HGCW_fund_norm']
x_data_minmax = HGCW_dataset_utils['HGCW_minmax']
y_labels_raw_np = HGCW_dataset_utils['HGCW_labels']
vow_size = HGCW_dataset_utils['classes_size']
end_idx = HGCW_dataset_utils['classes_idx']

# Auxiliary lists
vowels = ['ae', 'ah', 'aw', 'eh', 'er', 'ei', 'ih', 'iy', 'oa', 'oo', 'uh', 'uw']  # Vowels list
colors = ['red', 'saddlebrown', 'darkorange', 'darkgoldenrod', 'gold', 'darkkhaki', 'olive', 'darkgreen', 'steelblue', 'fuchsi


# Fundamental Frequencies Min-Max Normalization
a = 0
b = 1
x_data_minmax[:, 0] = a + ((x_data_minmax[:, 0] - x_data_minmax[:, 0].min()) / (x_data_minmax[:, 0].max() - x_data_minmax[:, 0
x_data_minmax[:, 1:] = x_data_minmax[:, 1:]  # Fundamental column exception


# Class-specific One-hot encoding (Binarization)
def one_hot_encoder(sel_class_number: int = 3, dataset: np.ndarray = x_data_minmax, orig_labels: int = len(vowels), classes_si

    classes = [n for n in range(orig_labels)]  # Class Labels list initialization

    # Auxiliary Parameters Initialization
    if sel_class_number < len(classes):
```

```
        classes.remove(sel_class_number)  # REST Classes list
        if debug is True:
            print(f'Selected Class "{vowels[sel_class_number]}" : {classes_size[sel_class_number]} samples')
        sub_classes_size = classes_size[sel_class_number] // len(classes)
        if debug is True:
            print(f'Rest Classes size (...each): {sub_classes_size} samples')

        # 1-Subset processing
        sub_data = dataset[classes_idx[sel_class_number]: classes_idx[sel_class_number + 1], :]  # Selected Class feature slic
        sub_data_labels_bin = np.ones((classes_size[sel_class_number], 1), dtype='int')  # Selected Class labels (1) creation
        sub_data_labels = np.ones((classes_size[sel_class_number], 1), dtype='int') * sel_class_number

        # 0-Subset processing
        for i in classes:
            class_i_indices = np.random.choice(np.arange(classes_idx[i], classes_idx[i + 1], 1), size=sub_classes_size, replac
            sub_class_i_array = dataset[class_i_indices, :]
            sub_class_labels_bin_array = np.zeros((sub_class_i_array.shape[0], 1), dtype='int')  # Rest I-esimal Class labels
            sub_class_labels_array = np.ones((sub_class_i_array.shape[0], 1), dtype='int') * i

            # Outputs append
            sub_data = np.vstack((sub_data, sub_class_i_array))
            sub_data_labels_bin = np.vstack((sub_data_labels_bin, sub_class_labels_bin_array))
            sub_data_labels = np.vstack((sub_data_labels, sub_class_labels_array))
    else:
        raise ValueError(f'Invalid Class ID: "{sel_class_number}" --> It must be less than {len(classes)}!')

    return sub_data, sub_data_labels_bin, sub_data_labels


# Train/Test split (auxiliary function)
def train_test_split_aux(features_dataset, labels_dataset, test_perc, tolerance):
    """
    An auxiliary Train_Test_split function (based on Scikit Learn implementation) w. balance tolerance specification
    """
    test_size = int(test_perc / 100 * len(features_dataset))
    train_balance = 0  # Output Training set balance value initialization
    test_balance = 0  # Output Testing set balance value initialization

    min_tol = np.mean(labels_dataset) - tolerance
    max_tol = np.mean(labels_dataset) + tolerance
    print(f'Data Balancing  (TARGET = {np.mean(labels_dataset)} +- {tolerance}): ', end='')

    while (min_tol >= train_balance or train_balance >= max_tol) or (min_tol >= test_balance or test_balance >= max_tol):
        train_data, test_data, train_labels, test_labels = train_test_split(features_dataset, labels_dataset, test_size=test_s
        train_balance = np.mean(train_labels)
        test_balance = np.mean(test_labels)
        print('.', end='')
    else:
        print('OK')

    return train_data, test_data, train_labels, test_labels, train_balance, test_balance

# Train-Dev-Test split function
def train_dev_test_split(x_data, y_labels, split_list, tolerance=0.1, output='Loaders', debug=False):
    """
    Compute a Train, Development (Hold-Out) and a Test set split w. PyTorch Dataset conversion (and eventual Loaders initializ
    """
    if len(split_list) == 3:
        # Train - Dev+Test separation
        print('Training --- Devel/Test SPLIT')
        train_data, testTMP_data, train_labels, testTMP_labels, _, _ = train_test_split_aux(x_data, y_labels, (split_list[1] *
        print('---------------------------------')

        # Dev - Test separation
        print('Devel    ---     Test SPLIT')

        split = ((split_list[1] * 100) / np.sum(split_list[1:] * 100)) * 100  # Split in %
        dev_data, test_data, dev_labels, test_labels, _, _ = train_test_split_aux(testTMP_data, testTMP_labels, split, toleran
        print('---------------------------------')

        # Tensor Conversion
        train_data_tensor = torch.tensor(train_data).float()
        train_labels_tensor = torch.tensor(train_labels, dtype=torch.int64).squeeze()
        dev_data_tensor = torch.tensor(dev_data).float()
        dev_labels_tensor = torch.tensor(dev_labels, dtype=torch.int64).squeeze()
        test_data_tensor = torch.tensor(test_data).float()
        test_labels_tensor = torch.tensor(test_labels, dtype=torch.int64).squeeze()
        if debug is True:
            print(f'Training Data        Shape: {train_data.shape}')
            print(f'Development Data     Shape: {dev_data.shape}')
            print(f'Testing Data         Shape: {test_data.shape}')

            # Balance Evaluation
```

```
            print(f'Training Set       Balance: {np.mean(train_labels)}')
            print(f'Development Set    Balance: {np.mean(dev_labels)}')
            print(f'Testing Set        Balance: {np.mean(test_labels)}')

        if output != 'Loaders':
            return train_data_tensor, train_labels_tensor, dev_data_tensor, dev_labels_tensor, test_data_tensor, test_labels_t
        else:
            # PyTorch Dataset Conversion
            train_dataset = torch.utils.data.TensorDataset(torch.tensor(train_data).float(), torch.tensor(train_labels, dtype=
            dev_dataset = torch.utils.data.TensorDataset(torch.tensor(dev_data).float(), torch.tensor(dev_labels, dtype=torch.
            test_dataset = torch.utils.data.TensorDataset(torch.tensor(test_data).float(), torch.tensor(test_labels, dtype=tor

            # DataLoader (Batches) --> Drop-Last control to optimize training
            trainLoader = DataLoader(train_dataset, shuffle=False, batch_size = 32, drop_last=True)
            devLoader = DataLoader(dev_dataset, shuffle=False, batch_size = dev_dataset.tensors[0].shape[0])
            testLoader = DataLoader(test_dataset, shuffle=False, batch_size = test_dataset.tensors[0].shape[0])
            if debug is True:
                print(f'Training Set    Batch Size: {trainLoader.batch_size}')
                print(f'Development Set Batch Size: {devLoader.batch_size}')
                print(f'Testing Set     Batch Size: {testLoader.batch_size}')

            return trainLoader, devLoader, testLoader
    else:
        # Train - Test separation
        print('Training --- Test    SPLIT')
        train_data, test_data, train_labels, test_labels, _, _ = train_test_split_aux(x_data, y_labels, split_list[1] * 100, t
        print('--------------------------')

        # Tensor Conversion
        train_data_tensor = torch.tensor(train_data).float()
        train_labels_tensor = torch.tensor(train_labels, dtype=torch.int64).squeeze()
        test_data_tensor = torch.tensor(test_data).float()
        test_labels_tensor = torch.tensor(test_labels, dtype=torch.int64).squeeze()
        if debug is True:
            print(f'Training Data        Shape: {train_data.shape}')
            print(f'Testing Data         Shape: {test_data.shape}')

            # Balance Evaluation
            print(f'Training Set    Balance: {np.mean(train_labels)}')
            print(f'Testing Set     Balance: {np.mean(test_labels)}')

        if output != 'Loaders':
            return train_data_tensor, train_labels_tensor, test_data_tensor, test_labels_tensor
        else:
            # PyTorch Dataset Conversion
            train_dataset = torch.utils.data.TensorDataset(torch.tensor(train_data).float(), torch.tensor(train_labels, dtype=
            test_dataset = torch.utils.data.TensorDataset(torch.tensor(test_data).float(), torch.tensor(test_labels, dtype=tor

            # DataLoader (Batches) --> Drop-Last control to optimize training
            trainLoader = DataLoader(train_dataset, shuffle=False, batch_size = 32, drop_last=True)
            testLoader = DataLoader(test_dataset, shuffle=False, batch_size = test_dataset.tensors[0].shape[0])
            if debug is True:
                print(f'Training Set    Batch Size: {trainLoader.batch_size}')
                print(f'Testing Set     Batch Size: {testLoader.batch_size}')

            return trainLoader, testLoader
```

One-Class Architecture (Binary Classifier)

(see "*One-Class_Sub-Network_Analysis.ipynb*")

```
Multi-Layer Perceptron
- Input Layer: 3 features [formant ratios, min-max normalized]
- Hidden Layer: 100 units
- Output Layer: 1 normalized probability
- Learning Rate: 0.0001 (10^-4)
- Optimizer: Adam (Adaptive Momentum)

- Mini-Batch Training:
    . Re-iterated Sub-Dataset Shuffling
    . Early Stopping (Test Accuracy driven)
    . Batch size = 32

- Regularization:
    . Weight Decay (L2 Penalty): 0.0001 (10^-4)
    . DropOut:
        * Input Layer Drop Rate: 0.8
```

```
      * Hidden Layer Drop Rate: 0.5.
   . Batch Normalization
```

- *MLP Classifier Architecture* class definition
- *Mini-Batch Training* function definition

```
# Dynamic Multi-Layer Architecture Class (w. units, activation function, batch normalization and dropOut rate specification)
class binaryClassifier(nn.Module):                                                # nn.Module: base class to inherit from
    def __init__(self, n_units, act_fun, rate_in, rate_hidden, model_name):       # self + attributes (architecture hyper-para
        super().__init__()

        self.layers = nn.ModuleDict()                                             # Dictionary to store Model layers
        self.name = model_name

        # Input Layer
        self.layers['input'] = nn.Linear(4, n_units)                              # Key 'input' layer specification

        # Hidden Layer
        self.layers[f'hidden'] = nn.Linear(n_units, n_units)
        self.layers[f'batch_norm'] = nn.BatchNorm1d(n_units)

        # Output Layer
        self.layers['output'] = nn.Linear(n_units, 1)                             # Key 'output' layer specification

        # Activation Function
        self.actfun = act_fun                                                     # Function string-name attribute association

        # Dropout Parameter
        self.dr_in = rate_in
        self.dr_hidden = rate_hidden

        # Weights & Bias initialization
        for layer in self.layers.keys():
            try:
                nn.init.kaiming_normal_(self.layers[layer].weight, mode='fan_in') # Kaiming He - Normal Distributed (ReLU spec
            except:
                pass                                                              # Batch_norm Layer can't be initialized
            self.layers[layer].bias.data.fill_(0.)                                # Bias initialization (0.)

    # Forward Pass Method
    def forward(self, x):

        # Activation function object computation
        actfun = getattr(torch.nn, self.actfun)

        # Input Layer pass                                                        --> Weightening (Dot Product) "Linear transf
        x = actfun()(self.layers['input'](x))
        x = F.dropout(x, p=self.dr_in, training=self.training)                    # Activate DropOut only when Model Training

        # Single Hidden Layer pass                                                --> Weightening (Dot Product) "Linear transf
        x = self.layers[f'batch_norm'](x)                                         # Apply batch normalization before hidden la
        x = actfun()(self.layers[f'hidden'](x))
        x = F.dropout(x, p=self.dr_hidden, training=self.training)                # Same as "Input pass"

        # Output Layer pass                                                       --> Output Weightening (Dot Product) "Linear
        x = self.layers['output'](x)
        x = nn.Sigmoid()(x)

        return x

# Batch Training function (w. Adam Optimizer & L2 penalty term) Re-Definition
def mini_batch_train_test(model, weight_decay, epochs: int, learning_rate, train_loader, dev_loader, test_loader, debug=False)
    """
    Train & Test an ANN Architecture via Mini-Batch Training (w. Train/Dev/Test PyTorch Loaders) and Adam Backpropagation Opti
    """
    # Loss Function initialization
    loss_function = nn.BCELoss()

    # Optimizer Algorithm initialization
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=weight_decay)

    # Output list initialization
    train_accuracies = []
    train_losses = []
    dev_accuracies = []

    # TRAINING Phase
    for epoch in range(epochs):
        model.train()  # TRAINING Switch ON
```

```
        batch_accuracies = []
        batch_losses = []

        # Training BATCHES Loop
        for data_batch, labels_batch in train_loader:
            train_predictions = model(data_batch)
            train_loss = loss_function(train_predictions.squeeze(), labels_batch.type(torch.int64).float())
            batch_losses.append(train_loss.detach())

            # Backpropagation
            optimizer.zero_grad()
            train_loss.backward()
            optimizer.step()

            # Accuracy
            train_accuracy = 100 * torch.mean(((train_predictions.squeeze() > 0.5) == labels_batch.type(torch.int64).squeeze()

            # Batch Stats appending
            batch_accuracies.append(train_accuracy.detach())
            batch_losses.append(train_loss.detach())

        # Training Stats appending
        train_accuracies.append(np.mean(batch_accuracies))  # Average of Batch Accuracies = Training step accuracy
        train_losses.append(np.mean(batch_losses))  # Average of Batch Losses = Training step Losses

        # EVALUATION (Dev) Phase
        model.eval()
        with torch.no_grad():
            dev_data_batch, dev_labels_batch = next(iter(dev_loader))
            dev_predictions = model(dev_data_batch)

            dev_accuracy = 100 * torch.mean(((dev_predictions.squeeze() > 0.5) == dev_labels_batch.type(torch.int64).squeeze()

            if debug is True:
                if epoch % 100 == 0:
                    print(f'Epoch {epoch} --> DEV ACCURACY: {dev_accuracy.detach():.3f} %')
                    print('-------------------------------')

            # Evaluation accuracy appending
            dev_accuracies.append(dev_accuracy.detach())

    # TEST Phase
    model.eval()
    with torch.no_grad():
        test_data_batch, test_labels_batch = next(iter(test_loader))
        test_predictions = model(test_data_batch)
        test_accuracy = 100 * torch.mean(((test_predictions.squeeze() > 0.5) == test_labels_batch.type(torch.int64).squeeze())

        if debug is True:
            print(f'TEST ACCURACY: {test_accuracy.detach():.2f} %')
            print('---------------------------------------------------------------------')

    return train_accuracies, train_losses, dev_accuracies, test_accuracy.detach()
```

OCON (One-Class-One-Net) Model

Binary classifiers **Parallelization**

- *Classifiers-Bank* function definition
  - *Models Parameters* inspection
- Classifiers **Sequential** Training & Evaluation
- *Models Parameters State Save/Load* function definition

- MaxNet output algorithm
- Argmax output algorithm

```
def OCON_bank(one_class_function, hidden_units, act_fun, dr_in, dr_hidden, classes_list):
    """
    Create a One-Class-One-Network parallelization bank of an input Sub-Network definition
    """
    # Sub-Net names creation
    models_name_list = []
    for i in range(len(classes_list)):
        models_name_list.append("{}_{}".format(classes_list[i], "subnet"))  # Class name + _subnet

    # Sub-Networks instances creation
    sub_nets = []  # Sub Network list initialization
```

```
    for i in range(len(models_name_list)):

        torch.manual_seed(SEED)  # Seed re-initialization

        # Sub-Net instance creation
        locals()[models_name_list[i]] = one_class_function(hidden_units, act_fun, dr_in, dr_hidden, models_name_list[i])
        sub_nets.append(locals()[models_name_list[i]])

    return sub_nets


# Load Architecture Parameters State function
def load_model_state(model, state_dict_path):
    """
    Load an existent State Dictionary in a defined model
    """

    model.load_state_dict(torch.load(state_dict_path))
    print(f'Loaded Parameters (from "{state_dict_path}") into: {model.name}')

    return model


# Build The OCON Model
ocon_vowels = OCON_bank(binaryClassifier, 100, 'ReLU', 0.8, 0.5, vowels)  # Best MLP (see "One-Class_Binary_Classifier_Analysi

# Load Pre-Trained Architectures in a fresh Model instance:
#ocon_vowels = OCON_bank(binaryClassifier, 100, 'ReLU', 0.8, 0.5, vowels)  # Best MLP (see "One-Class_Binary_Classifier_Analys
#states_path = ["Trained_models_state/ae_subnet_Params.pth",
#               "Trained_models_state/ah_subnet_Params.pth",
#               "Trained_models_state/aw_subnet_Params.pth",
#               "Trained_models_state/eh_subnet_Params.pth",
#               "Trained_models_state/er_subnet_Params.pth",
#               "Trained_models_state/ei_subnet_Params.pth",
#               "Trained_models_state/ih_subnet_Params.pth",
#               "Trained_models_state/iy_subnet_Params.pth",
#               "Trained_models_state/oa_subnet_Params.pth",
#               "Trained_models_state/oo_subnet_Params.pth",
#               "Trained_models_state/uh_subnet_Params.pth",
#               "Trained_models_state/uw_subnet_Params.pth"]
#
#for i in range(len(ocon_vowels)):
#    load_model_state(ocon_vowels[i], states_path[i])


# OCON Evaluation function
def OCON_eval(ocon_models_bank, features_dataset: np.ndarray = x_data_minmax, labels: np.ndarray = y_labels_raw_np):
    """
    Evaluate OCON models-bank over an entire dataset
    """
    # Output lists initialization
    predictions = []
    dist_errors = []
    eval_accuracies = []
    g_truths = []  # For plotting purpouses

    # Evaluate each Sub-Network...
    for i in range(len(ocon_models_bank)):
        ocon_models_bank[i].eval()  # Put j-esimal Sub-Network in Evaluation Mode
        print(f'{ocon_models_bank[i].name.upper()} Evaluation -', end=' ')

        with torch.no_grad():

            # Make predictions
            features_data_tensor = torch.tensor(features_dataset).float()
            raw_eval_predictions = ocon_models_bank[i](features_data_tensor)

            # Create Ground Truths
            ground_truth = np.where(labels == i, 1, 0)
            ground_truth_tensor = torch.tensor(ground_truth, dtype=torch.int64).squeeze()

            # Compute Errors
            dist_error = ground_truth_tensor - raw_eval_predictions.detach().squeeze()  # Distances
            eval_accuracy = 100 * torch.mean(((raw_eval_predictions.detach().squeeze() > 0.5) == ground_truth_tensor).float())
            print(f'Accuracy: {eval_accuracy:.2f}%')

        # Outputs append
        predictions.append(raw_eval_predictions.detach())
        dist_errors.append(dist_error.detach())
        eval_accuracies.append(eval_accuracy.detach())

        g_truths.append(ground_truth)
```

```
    return predictions, dist_errors, eval_accuracies, g_truths

# For Pre-Trained Models
#ocon_predictions, ocon_dist_errors, ocon_eval_accuracies, ocon_g_truths = OCON_eval(ocon_vowels)


# Model Parameters State function
def model_desc(model):
    """
    Print a Console report of Neural Network Model parameters
    """
    # Parameters Description
    print('Params Description:')
    trainable_params = 0

    for parameter in model.named_parameters():
        print(f'Parameter Name      : {parameter[0]}')
        print(f'Parameter Weights   : {parameter[1][:]}')
        if parameter[1].requires_grad:
            print(f'...with {parameter[1].numel()} TRAINABLE parameters')
            trainable_params += parameter[1].numel()

        print('................................')

    print('----------------------------------------------------------------')

    # Nodes Count
    nodes = 0
    for param_name, param_tensor in model.named_parameters():
        if 'bias' in param_name:
            nodes += len(param_tensor)

    print(f'Total Nodes             : {nodes}')
    print('----------------------------------------------------------------')

# OCON-Model Description
for i in range(len(ocon_vowels)):
    print(f'OCON "{ocon_vowels[i].name}" Classifier STATE')
    model_desc(ocon_vowels[i])
    print()


# Training/Eval/Testing Parameters
epochs = 1000  # For each "Data Batch-Set"
loss_break = 0.20  # loss (for Early Stopping)
acc_break = 90.  # % accuracy (for Early Stopping)
min_tolerance = 0.01 # ...for sub-dataset balancing

# Outputs Initialization
loss_functions = [[] for _ in range(len(ocon_vowels))]
training_accuracies = [[] for _ in range(len(ocon_vowels))]
evaluation_accuracies = [[] for _ in range(len(ocon_vowels))]
test_accuracies = [[] for _ in range(len(ocon_vowels))]
training_times = []


# OCON Sub-Networks Training
from time import perf_counter
debug = False

for i, vowel in enumerate(vowels):
    print(f'Architecture "{ocon_vowels[i].name}" TRAINING PHASE')

    start_timer = perf_counter()
    # Iterated (w. Batch-Sets shuffling) Mini-Batch Training
    iteration = 0  # Batch Training iteration counter
    mean_loss = 1.
    test_accuracy = 0.

    while (mean_loss > loss_break) or (test_accuracy < acc_break):
        # Dataset processing
        sub_data, sub_data_labels_bin, _ = one_hot_encoder(sel_class_number=i, dataset=x_data_minmax, debug=debug)
        print('---------------------------------')
        trainLoader, devLoader, testLoader = train_dev_test_split(sub_data, sub_data_labels_bin, [0.5, 0.25, 0.25], tolerance=

        # Train/Test Architecture
        train_accuracies, train_losses, dev_accuracies, test_accuracy = mini_batch_train_test(ocon_vowels[i], weight_decay=0.0
        print(f'Sub-Net "{vowel.upper()}" Epoch {(iteration + 1) * epochs} - TEST ACCURACY: {test_accuracy:.2f}%', end=' ')

        # Outputs append
        loss_functions[i].append(train_losses)
        training_accuracies[i].append(train_accuracies)
        evaluation_accuracies[i].append(dev_accuracies)
```

```
        test_accuracies[i].append(test_accuracy)

        # Repeating condition evaluation
        mean_loss = np.mean(train_losses[-50: ])  # Last 50 losses mean
        print(f'- MEAN LOSS: {mean_loss}')

        iteration += 1  # Go to next Batch training iteration

    print(f'Training STOPPED at iteration {iteration}')
    print('-------------------------------------------------------------------')
    stop_timer = perf_counter()

    print(f'"{ocon_vowels[i].name}" Training COMPLETED in {float(stop_timer - start_timer)}sec.')
    training_times.append(stop_timer - start_timer)
    print('-------------------------------------------------------------------')


# Graphical smoothing filter
def smooth(data, k=100):
    """
    A Convolution LP filter w. interval definition
    """
    return np.convolve(data, np.ones(k) / k, mode='same')


# Training Phase Plots
plt.figure(figsize=(12, 5 * 12))

# loss_functions, training_accuracies, evaluation_accuracies, test_accuracies, training_times
classes = len(ocon_vowels)

for i in range(classes):
    plt.subplot(classes, 2, (i * 2) + 1)
    flat_loss_function = [item for sublist in loss_functions[i] for item in sublist]

    plt.plot(smooth(flat_loss_function), 'k-')
    plt.axhline(loss_break, color='r', linestyle='--')
    plt.title(f'{ocon_vowels[i].name.upper()} Training Loss')
    plt.xlabel('Epochs')
    plt.xlim([100, len(flat_loss_function) - 100])
    plt.ylabel('GT - Predicted diff. (probability)')
    plt.grid()

    plt.subplot(classes, 2, (i * 2) + 2)
    flat_training_accuracy = [item for sublist in training_accuracies[i] for item in sublist]
    flat_dev_accuracy = [item for sublist in evaluation_accuracies[i] for item in sublist]
    flat_test_accuracy = test_accuracies[i]

    plt.plot(smooth(flat_training_accuracy), 'k-', label='Training')
    plt.plot(smooth(flat_dev_accuracy), color='grey', label='Development')
    if len(flat_test_accuracy) > 1:
        plt.plot([(n + 1) * epochs for n in range(len(flat_test_accuracy))], flat_test_accuracy, 'r-', label=f'Test')
    else:
        plt.axhline(test_accuracy, color='r', linestyle='-', label=f'Test')
    plt.title(f'{ocon_vowels[i].name.upper()} Accuracy (after {training_times[i]:.2f}sec.)')
    plt.xlabel('Epochs')
    plt.xlim([100, len(flat_training_accuracy) - 100])
    plt.ylabel('Accuracy (in %)')
    plt.ylim([40, 101])
    plt.grid()
    plt.legend(loc='best')

plt.tight_layout()
plt.savefig('OCON_training_phase')
plt.show()


# OCON Evaluation
ocon_predictions, ocon_dist_errors, ocon_eval_accuracies, ocon_g_truths = OCON_eval(ocon_vowels)


# Dataset Evaluation Analysis Plot
plt.figure(figsize=(18, 5 * len(ocon_vowels)))
plot_ticks = end_idx[:]
plot_ticks = np.delete(plot_ticks, -1)

for i in range(len(ocon_vowels)):
    plt.subplot(len(ocon_vowels), 3, (i * 3) + 1)
    plt.plot(ocon_predictions[i], 'k.', label='Raw Predictions')
    plt.plot(ocon_g_truths[i], 'rx', label='Ground Truths')
    plt.axhline(0.5, linestyle='--', color='grey')
    plt.title(f'{ocon_vowels[i].name.upper()} Predictions Accuracy: {ocon_eval_accuracies[i]:.2f}%')
    plt.xlabel('Data (Indices)')
    plt.xticks(ticks=plot_ticks, labels=vowels)
```

```
    plt.ylabel('Normalized Probability')
    plt.grid()
    plt.legend(loc='best')

    plt.subplot(len(ocon_vowels), 3, (i * 3) + 2)
    plt.plot(ocon_dist_errors[i], 'k')
    plt.title(f'Predicted to Measured Error')
    plt.xlabel('Data (Indices)')
    plt.xticks(ticks=plot_ticks, labels=vowels)
    plt.ylabel('Normalized Probability Error')
    plt.ylim([-1.1, 1.1])
    plt.grid()

    plt.subplot(len(ocon_vowels), 3, (i * 3) + 3)

    # Predictions list processing
    predictions_temp = ocon_predictions[i]
    class_predictions = [item for sublist in predictions_temp for item in sublist]  # Turn a list of lists in a single list (c
    for j in range(len(class_predictions)):  # Turn a list of tensors of one variable in a list of scalars (item() method)
        class_predictions[j] = class_predictions[j].item()

    # Positives & False-Positives extraction
    positives = []
    for w in range(len(vowels)):
        num = (np.array(class_predictions[end_idx[w]: end_idx[w + 1]]) > 0.5).sum()
        positives.append(num)

    plt.bar(np.arange(len(vowels)), positives, color='k')
    plt.title(f'"{vowels[i]}" Positive Probabilities Distribution')
    plt.xlabel('Normalized Probabilities')
    plt.ylabel('Occurences')
    plt.xticks([n for n in range(12)], vowels)
    plt.grid()

plt.tight_layout()
plt.savefig('OCON_bank_evaluation')
plt.show()


# Model Parameters Save/Load functions
from pathlib import Path

def save_model_state(model, folder_name: str = "Trained_models_state"):
    """
    Save Pre-Trained model parameters in a State Dictionary
    """

    MODEL_PATH = Path(folder_name)  # Placed in root
    MODEL_PATH.mkdir(parents=True, exist_ok=True)  # Pre-existing folder (w. same name) monitoring
    MODEL_NAME = '{}_{}'.format(model.name, "Params.pth")
    MODEL_SAVE_PATH = MODEL_PATH / MODEL_NAME

    print(f"Saving {model.name} Parameters in: {MODEL_SAVE_PATH}")
    torch.save(obj=model.state_dict(), f=MODEL_SAVE_PATH)

    return MODEL_SAVE_PATH

# Save Pre-Trained Models-bank
states_path = []  # Path for each model parameters state
for i in range(len(ocon_vowels)):
    state_path = save_model_state(ocon_vowels[i])
    states_path.append(state_path)
print()
```

Output **Maxnet Algorithm**

```
# OCON "MaxNet" Architecture (Weightening + Non Linearity apply)
class OCON_MaxNet(nn.Module):                                           # nn.Module: base class to inherit from
    def __init__(self, n_units, act_fun, eps):                              # self + attributes (architecture hyper-paramet
        super().__init__()

        self.layers = nn.ModuleDict()                                   # Dictionary to store Model layers
        self.eps_weight = eps

        # MaxNet Layer
        self.layers['MAXNET'] = nn.Linear(n_units, n_units)             # Key 'MaxNet' layer specification

        # Weights & Bias initialization
        self.layers['MAXNET'].weight.data.fill_(self.eps_weight)
        for i in range(n_units):
```

```
                self.layers['MAXNET'].weight[i][i].data.fill_(1.)  # Self Weight = 1

            self.layers['MAXNET'].bias.data.fill_(0.)

            # Activation Function
            self.actfun = act_fun  # Function string-name attribute association

        # Forward Pass Method
        def forward(self, x):

            # Activation function object computation
            actfun = getattr(torch.nn, self.actfun)

            # Maxnet Layer pass                                              --> Output Weightening (Dot Product) "Linear transfc
            x = actfun()(self.layers['MAXNET'](x.squeeze().float()))

            # Self

            return x


# Build OCON MaxNetwork Architecture
torch.manual_seed(SEED)
ocon_maxnet = OCON_MaxNet(n_units=12, act_fun='ReLU', eps=0.25)

# MaxNet & Sub-Networks Parameters
print('OCON MaxNet STATE')
model_desc(ocon_maxnet)


def maxnet_algo(maxnet_function, n_units, act_fun, eps, input_array):
    """
    MaxNet Re-iteration algorithm for Maximum Value retrieving from an input array
    """
    non_zero_outs = np.count_nonzero(input_array)  # Non Zero Values initialization
    maxnet_in = torch.from_numpy(input_array)  # MaxNet Input Tensor initialization

    results = []  # Results initialization

    counter = 0
    while non_zero_outs != 1:
        counter += 1

        # Create the MaxNet
        torch.manual_seed(SEED)  # Redundant
        maxnet = maxnet_function(n_units = n_units, act_fun = act_fun, eps = eps)

        # Compute Forward Pass
        results = maxnet(maxnet_in)

        # Non_zero outputs & Maxnet Input Update
        non_zero_outs = np.count_nonzero(results.detach().numpy())
        maxnet_in = results.detach()  # Save results for next iteration

    print(f'Maximum Value found in {counter} iterations')
    return np.argmax(results.detach().numpy())


# MaxNet on Sub-Networks predictions
ocon_predictions_prob = np.zeros((len(ocon_predictions), x_data_minmax.shape[0]))  # NumPy predictions matrix (12 * 1617)

# Convert from List of Tensors to 2D NumPy Array
for i in range(len(ocon_predictions)):
    ocon_predictions_prob[i, :] = ocon_predictions[i].detach().squeeze().numpy()

maxnet_class_predictions = []  # Classes Outputs list initialization
# MaxNet application
for i in range(x_data_minmax.shape[0]):
    print(f'Dataset Sample({i + 1}) Class Evaluation')

    samp_predictions = ocon_predictions_prob[:, i]  # Array of 12 predictions for each Dataset sample (OCON outputs)
    class_prediction = maxnet_algo(OCON_MaxNet, n_units=12, act_fun='ReLU', eps=-0.1, input_array=samp_predictions)  # MaxNet
    maxnet_class_predictions.append(class_prediction)  # Result appending
    print('------------------------------------------------')

maxnet_accuracy = 100 * np.mean((np.array(maxnet_class_predictions).reshape(1617, 1) == y_labels_raw_np))  # Accuracy computat
print(f'Maxnet Output --> Phoneme ACCURACY: {maxnet_accuracy}%')


# Argmax on Sub-Networks predictions (...for multiple 1s probabilities MaxNet infinite loops)
#ocon_predictions_prob = np.zeros((len(new_ocon_predictions), x_data_minmax.shape[0]))  # NumPy predictions matrix (12 * 1617)
#
# Convert from List of Tensors to 2D NumPy Array
#for i in range(len(new_ocon_predictions)):
```

```
#    ocon_predictions_prob[i, :] = new_ocon_predictions[i].detach().squeeze().numpy()
#
#maxnet_class_predictions = np.argmax(ocon_predictions_prob, axis=0)
#maxnet_accuracy = 100 * np.mean((np.array(maxnet_class_predictions).reshape(1617, 1) == y_labels_raw_np))  # Accuracy computa
#print(f'Maxnet Output ACCURACY: {maxnet_accuracy}%')


# Evaluation Analysis Plot
plt.figure(figsize=(30, 5))
plt.suptitle(f'OCON Bank + MaxNet Evaluation: {maxnet_accuracy:.0f}%')

plot_x_ticks = end_idx[:]
plot_x_ticks = np.delete(plot_x_ticks, len(end_idx) - 1)
plot_y_ticks = [n for n in range(len(vowels))]

plt.plot(y_labels_raw_np, 'rs', label='Ground Truths')
plt.plot(maxnet_class_predictions, 'k.', label='MAXNET Outputs')
plt.xlabel('Dataset samples')
plt.xticks(ticks=plot_x_ticks, labels=vowels)
plt.xlim([-10, len(y_labels_raw_np) + 10])
plt.ylabel('Labels')
plt.yticks(ticks=plot_y_ticks, labels=vowels)
plt.legend(loc='best')
plt.grid()

plt.tight_layout()
plt.savefig('OCON_model_evaluation')
plt.show()
```

OCON Model Analysis

3-features NO-Children Dataset

Author: S. Giacomelli

Year: 2023

Affiliation: A.Casella Conservatory (student)

Master Degree Thesis: "*Vowels phonemes Analysis & Classification by means of OCON rectifiers Deep Learning Architectures*"

Description: Python scripts for One-Class-One-Network (OCON) Model analysis and optimization

```
# Numerical computations packages/modules
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

# Dataset processing modules
from torch.utils.data import DataLoader
from sklearn.model_selection import train_test_split

# Graphic visualization modules
import matplotlib.pyplot as plt
import matplotlib_inline
matplotlib_inline.backend_inline.set_matplotlib_formats('svg')

# Common Seed initialization
SEED = 42  # ... the answer to the ultimate question of Life, the Universe, and Everything... (cit.)

# PyTorch Processing Units evaluation
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(f'AVILABLE Processing Unit: {device.upper()}')
!nvidia-smi
```

HGCW Dataset

- *Dataset_utils.npz* file read
- *One-Hot encoding* definition
- *Train/Dev/Test split* definition

```
# Load HGCW 4_features_men Dataset
HGCW_dataset_utils_m = np.load(file='./HGCW_dataset_utils_men.npz')
print('MEN Sub-Dataset')
print('Raw features                     Data shape:', HGCW_dataset_utils_m['HGCW_raw'].shape)
print('Fundamental Normalized features Data shape:', HGCW_dataset_utils_m['HGCW_fund_norm'].shape)
print('MinMax features                  Data shape:', HGCW_dataset_utils_m['HGCW_minmax'].shape)
print('Labels                           Data shape:', HGCW_dataset_utils_m['HGCW_labels'].shape)
print('Classes size                     Data shape:', HGCW_dataset_utils_m['classes_size'].shape)
print('Classes indices                  Data shape:', HGCW_dataset_utils_m['classes_idx'].shape)

x_data_raw_np_m = HGCW_dataset_utils_m['HGCW_raw']
x_data_fund_norm_m = HGCW_dataset_utils_m['HGCW_fund_norm']
x_data_minmax_m = HGCW_dataset_utils_m['HGCW_minmax']
y_labels_raw_np_m = HGCW_dataset_utils_m['HGCW_labels']
vow_size_m = HGCW_dataset_utils_m['classes_size']
end_idx_m = HGCW_dataset_utils_m['classes_idx']
print()

# Load HGCW 4_features_women Dataset
HGCW_dataset_utils_w = np.load(file='./HGCW_dataset_utils_women.npz')
print('WOMEN Sub-Dataset')
print('Raw features                     Data shape:', HGCW_dataset_utils_w['HGCW_raw'].shape)
print('Fundamental Normalized features Data shape:', HGCW_dataset_utils_w['HGCW_fund_norm'].shape)
print('MinMax features                  Data shape:', HGCW_dataset_utils_w['HGCW_minmax'].shape)
print('Labels                           Data shape:', HGCW_dataset_utils_w['HGCW_labels'].shape)
print('Classes size                     Data shape:', HGCW_dataset_utils_w['classes_size'].shape)
print('Classes indices                  Data shape:', HGCW_dataset_utils_w['classes_idx'].shape)

x_data_raw_np_w = HGCW_dataset_utils_w['HGCW_raw']
x_data_fund_norm_w = HGCW_dataset_utils_w['HGCW_fund_norm']
x_data_minmax_w = HGCW_dataset_utils_w['HGCW_minmax']
y_labels_raw_np_w = HGCW_dataset_utils_w['HGCW_labels']
vow_size_w = HGCW_dataset_utils_w['classes_size']
end_idx_w = HGCW_dataset_utils_w['classes_idx']
```

```
# Auxiliary lists
vowels = ['ae', 'ah', 'aw', 'eh', 'er', 'ei', 'ih', 'iy', 'oa', 'oo', 'uh', 'uw']  # Vowels list
colors = ['red', 'saddlebrown', 'darkorange', 'darkgoldenrod', 'gold', 'darkkhaki', 'olive', 'darkgreen', 'steelblue', 'fuchsi


# MEN Sub-Dataset Plot
dataset = x_data_minmax_m # x_data_fund_norm

plt.figure(figsize=(12, 15))
plt.suptitle('MEN Normalized Dataset "2-Features" separation')

for index, vowel in enumerate(vowels):

    first_coords = dataset[end_idx_m[index]: end_idx_m[index + 1], 1]
    second_coords = dataset[end_idx_m[index]: end_idx_m[index + 1], 2]
    third_coords = dataset[end_idx_m[index]: end_idx_m[index + 1], 3]

    plt.subplot(3, 1, 1)
    plt.title('$1_{st}$ VS $2_{nd}$')
    plt.scatter(first_coords, second_coords, marker='o', color=colors[index], label=f'Vowel "{vowel}"')
    plt.xlabel('$1_{st}$ Formant Frequency ratio')
    plt.ylabel('$2_{nd}$ Formant Frequency ratio')
    plt.legend(loc='best')
    plt.grid(True)

    plt.subplot(3, 1, 2)
    plt.title('$1_{st}$ VS $3_{rd}$')
    plt.scatter(first_coords, third_coords, marker='o', color=colors[index], label=f'Vowel "{vowel}"')
    plt.xlabel('$1_{st}$ Formant Frequency ratio')
    plt.ylabel('$3_{rd}$ Formant Frequency ratio')
    plt.legend(loc='best')
    plt.grid(True)

    plt.subplot(3, 1, 3)
    plt.title('$2_{nd}$ VS $3_{rd}$')
    plt.scatter(second_coords, third_coords, marker='o', color=colors[index], label=f'Vowel "{vowel}"')
    plt.xlabel('$2_{nd}$ Formant Frequency ratio')
    plt.ylabel('$3_{rd}$ Formant Frequency ratio')
    plt.legend(loc='best')
    plt.grid(True)

plt.tight_layout()
plt.savefig("men_normalized_dataset_plot")
plt.show()


# WOMEN Sub-Dataset Plot
dataset = x_data_minmax_w # x_data_fund_norm

plt.figure(figsize=(12, 15))
plt.suptitle('WOMEN Normalized Dataset "2-Features" separation')

for index, vowel in enumerate(vowels):

    first_coords = dataset[end_idx_w[index]: end_idx_w[index + 1], 1]
    second_coords = dataset[end_idx_w[index]: end_idx_w[index + 1], 2]
    third_coords = dataset[end_idx_w[index]: end_idx_w[index + 1], 3]

    plt.subplot(3, 1, 1)
    plt.title('$1_{st}$ VS $2_{nd}$')
    plt.scatter(first_coords, second_coords, marker='o', color=colors[index], label=f'Vowel "{vowel}"')
    plt.xlabel('$1_{st}$ Formant Frequency ratio')
    plt.ylabel('$2_{nd}$ Formant Frequency ratio')
    plt.legend(loc='best')
    plt.grid(True)

    plt.subplot(3, 1, 2)
    plt.title('$1_{st}$ VS $3_{rd}$')
    plt.scatter(first_coords, third_coords, marker='o', color=colors[index], label=f'Vowel "{vowel}"')
    plt.xlabel('$1_{st}$ Formant Frequency ratio')
    plt.ylabel('$3_{rd}$ Formant Frequency ratio')
    plt.legend(loc='best')
    plt.grid(True)

    plt.subplot(3, 1, 3)
    plt.title('$2_{nd}$ VS $3_{rd}$')
    plt.scatter(second_coords, third_coords, marker='o', color=colors[index], label=f'Vowel "{vowel}"')
    plt.xlabel('$2_{nd}$ Formant Frequency ratio')
    plt.ylabel('$3_{rd}$ Formant Frequency ratio')
    plt.legend(loc='best')
    plt.grid(True)
```

```
plt.tight_layout()
plt.savefig("women_normalized_dataset_plot")
plt.show()


# Stacked Dataset Creation (Labels ordered, same as original files)
x_data_raw_np = np.zeros((x_data_raw_np_m.shape[0] + x_data_raw_np_w.shape[0], x_data_raw_np_w.shape[1]))
x_data_fund_norm = np.zeros((x_data_fund_norm_m.shape[0] + x_data_fund_norm_w.shape[0], x_data_fund_norm_w.shape[1]))
x_data_minmax = np.zeros((x_data_minmax_m.shape[0] + x_data_minmax_w.shape[0], x_data_minmax_w.shape[1]))
y_labels_raw_np = np.zeros((y_labels_raw_np_m.shape[0] + y_labels_raw_np_w.shape[0], y_labels_raw_np_w.shape[1]))

vow_size = []
end_idx = [0]

for i in range(len(vowels)):
    # Extract and Vertical Stack Class-specific Data from both Sub-Datasets
    class_data_raw_np = np.vstack((x_data_raw_np_m[end_idx_m[i]: end_idx_m[i + 1], :], x_data_raw_np_w[end_idx_w[i]: end_idx_w
    class_data_fund_norm = np.vstack((x_data_fund_norm_m[end_idx_m[i]: end_idx_m[i + 1], :], x_data_fund_norm_w[end_idx_w[i]:
    class_data_minmax = np.vstack((x_data_minmax_m[end_idx_m[i]: end_idx_m[i + 1], :], x_data_minmax_w[end_idx_w[i]: end_idx_w
    class_labels_raw_np = np.vstack((y_labels_raw_np_m[end_idx_m[i]: end_idx_m[i + 1], :], y_labels_raw_np_w[end_idx_w[i]: end

    vow_size.append(class_data_minmax.shape[0])
    end_idx.append(end_idx[i] + class_data_minmax.shape[0])

    # Append to Output Matrices
    x_data_raw_np[end_idx[i]: end_idx[i + 1], :] = class_data_raw_np
    x_data_fund_norm[end_idx[i]: end_idx[i + 1], :] = class_data_fund_norm
    x_data_minmax[end_idx[i]: end_idx[i + 1], :] = class_data_minmax
    y_labels_raw_np[end_idx[i]: end_idx[i + 1], :] = class_labels_raw_np

print('HGCW (NO-Children) Sub-Dataset')
print('Raw features                    Data shape:', x_data_raw_np.shape)
print('Fundamental Normalized features Data shape:', x_data_fund_norm.shape)
print('MinMax features                 Data shape:', x_data_minmax.shape)
print('Labels                          Data shape:', y_labels_raw_np.shape)
print('Classes size                    Data shape:', len(vow_size))
print('Classes indices                 Data shape:', len(end_idx))


# Dataset Plot
dataset = x_data_minmax # x_data_fund_norm

plt.figure(figsize=(12, 15))
plt.suptitle('NO-CHILDREN Normalized Dataset "2-Features" separation')

for index, vowel in enumerate(vowels):

    first_coords = dataset[end_idx[index]: end_idx[index + 1], 1]
    second_coords = dataset[end_idx[index]: end_idx[index + 1], 2]
    third_coords = dataset[end_idx[index]: end_idx[index + 1], 3]

    plt.subplot(3, 1, 1)
    plt.title('$1_{st}$ VS $2_{nd}$')
    plt.scatter(first_coords, second_coords, marker='o', color=colors[index], label=f'Vowel "{vowel}"')
    plt.xlabel('$1_{st}$ Formant Frequency ratio')
    plt.ylabel('$2_{nd}$ Formant Frequency ratio')
    plt.legend(loc='best')
    plt.grid(True)

    plt.subplot(3, 1, 2)
    plt.title('$1_{st}$ VS $3_{rd}$')
    plt.scatter(first_coords, third_coords, marker='o', color=colors[index], label=f'Vowel "{vowel}"')
    plt.xlabel('$1_{st}$ Formant Frequency ratio')
    plt.ylabel('$3_{rd}$ Formant Frequency ratio')
    plt.legend(loc='best')
    plt.grid(True)

    plt.subplot(3, 1, 3)
    plt.title('$2_{nd}$ VS $3_{rd}$')
    plt.scatter(second_coords, third_coords, marker='o', color=colors[index], label=f'Vowel "{vowel}"')
    plt.xlabel('$2_{nd}$ Formant Frequency ratio')
    plt.ylabel('$3_{rd}$ Formant Frequency ratio')
    plt.legend(loc='best')
    plt.grid(True)

plt.tight_layout()
plt.savefig("no-children_normalized_dataset_plot")
plt.show()


# Class-specific One-hot encoding (Binarization)
def one_hot_encoder(sel_class_number: int = 3, dataset: np.ndarray = x_data_minmax, orig_labels: int = len(vowels), classes_si
```

```
    classes = [n for n in range(orig_labels)]  # Class Labels list initialization

    # Auxiliary Parameters Initialization
    if sel_class_number < len(classes):
        classes.remove(sel_class_number)  # REST Classes list
        if debug is True:
            print(f'Selected Class "{vowels[sel_class_number]}" : {classes_size[sel_class_number]} samples')
        sub_classes_size = classes_size[sel_class_number] // len(classes)
        if debug is True:
            print(f'Rest Classes size (...each): {sub_classes_size} samples')

        # 1-Subset processing
        sub_data = dataset[classes_idx[sel_class_number]: classes_idx[sel_class_number + 1], :]  # Selected Class feature slic
        sub_data_labels_bin = np.ones((classes_size[sel_class_number], 1), dtype='int')  # Selected Class labels (1) creation
        sub_data_labels = np.ones((classes_size[sel_class_number], 1), dtype='int') * sel_class_number

        # 0-Subset processing
        for i in classes:
            class_i_indices = np.random.choice(np.arange(classes_idx[i], classes_idx[i + 1], 1), size=sub_classes_size, replac
            sub_class_i_array = dataset[class_i_indices, :]
            sub_class_labels_bin_array = np.zeros((sub_class_i_array.shape[0], 1), dtype='int')  # Rest I-esimal Class labels
            sub_class_labels_array = np.ones((sub_class_i_array.shape[0], 1), dtype='int') * i

            # Outputs append
            sub_data = np.vstack((sub_data, sub_class_i_array))
            sub_data_labels_bin = np.vstack((sub_data_labels_bin, sub_class_labels_bin_array))
            sub_data_labels = np.vstack((sub_data_labels, sub_class_labels_array))
    else:
        raise ValueError(f'Invalid Class ID: "{sel_class_number}" --> It must be less than {len(classes)}!')

    return sub_data, sub_data_labels_bin, sub_data_labels


# Train/Test split (auxiliary function)
def train_test_split_aux(features_dataset, labels_dataset, test_perc, tolerance):
    """
    An auxiliary Train_Test_split function (based on Scikit Learn implementation) w. balance tolerance specification
    """
    test_size = int(test_perc / 100 * len(features_dataset))
    train_balance = 0  # Output Training set balance value initialization
    test_balance = 0  # Output Testing set balance value initialization

    min_tol = np.mean(labels_dataset) - tolerance
    max_tol = np.mean(labels_dataset) + tolerance
    print(f'Data Balancing  (TARGET = {np.mean(labels_dataset)} +- {tolerance}): ', end='')

    while (min_tol >= train_balance or train_balance >= max_tol) or (min_tol >= test_balance or test_balance >= max_tol):
        train_data, test_data, train_labels, test_labels = train_test_split(features_dataset, labels_dataset, test_size=test_s
        train_balance = np.mean(train_labels)
        test_balance = np.mean(test_labels)
        print('.', end='')
    else:
        print('OK')

    return train_data, test_data, train_labels, test_labels, train_balance, test_balance

# Train-Dev-Test split function
def train_dev_test_split(x_data, y_labels, split_list, tolerance=0.1, output='Loaders', debug=False):
    """
    Compute a Train, Development (Hold-Out) and a Test set split w. PyTorch Dataset conversion (and eventual Loaders initializ
    """
    if len(split_list) == 3:
        # Train - Dev+Test separation
        print('Training --- Devel/Test SPLIT')
        train_data, testTMP_data, train_labels, testTMP_labels, _, _ = train_test_split_aux(x_data, y_labels, (split_list[1] *
        print('---------------------------------')

        # Dev - Test separation
        print('Devel    ---     Test SPLIT')

        split = ((split_list[1] * 100) / np.sum(split_list[1:] * 100)) * 100  # Split in %
        dev_data, test_data, dev_labels, test_labels, _, _ = train_test_split_aux(testTMP_data, testTMP_labels, split, toleran
        print('---------------------------------')

        # Tensor Conversion
        train_data_tensor = torch.tensor(train_data).float()
        train_labels_tensor = torch.tensor(train_labels, dtype=torch.int64).squeeze()
        dev_data_tensor = torch.tensor(dev_data).float()
        dev_labels_tensor = torch.tensor(dev_labels, dtype=torch.int64).squeeze()
        test_data_tensor = torch.tensor(test_data).float()
        test_labels_tensor = torch.tensor(test_labels, dtype=torch.int64).squeeze()
        if debug is True:
            print(f'Training Data        Shape: {train_data.shape}')
```

```
            print(f'Development Data     Shape: {dev_data.shape}')
            print(f'Testing Data         Shape: {test_data.shape}')

            # Balance Evaluation
            print(f'Training Set       Balance: {np.mean(train_labels)}')
            print(f'Development Set    Balance: {np.mean(dev_labels)}')
            print(f'Testing Set        Balance: {np.mean(test_labels)}')

        if output != 'Loaders':
            return train_data_tensor, train_labels_tensor, dev_data_tensor, dev_labels_tensor, test_data_tensor, test_labels_t
        else:
            # PyTorch Dataset Conversion
            train_dataset = torch.utils.data.TensorDataset(torch.tensor(train_data).float(), torch.tensor(train_labels, dtype=
            dev_dataset = torch.utils.data.TensorDataset(torch.tensor(dev_data).float(), torch.tensor(dev_labels, dtype=torch.
            test_dataset = torch.utils.data.TensorDataset(torch.tensor(test_data).float(), torch.tensor(test_labels, dtype=tor

            # DataLoader (Batches) --> Drop-Last control to optimize training
            trainLoader = DataLoader(train_dataset, shuffle=False, batch_size = 32, drop_last=True)
            devLoader = DataLoader(dev_dataset, shuffle=False, batch_size = dev_dataset.tensors[0].shape[0])
            testLoader = DataLoader(test_dataset, shuffle=False, batch_size = test_dataset.tensors[0].shape[0])
            if debug is True:
                print(f'Training Set    Batch Size: {trainLoader.batch_size}')
                print(f'Development Set Batch Size: {devLoader.batch_size}')
                print(f'Testing Set     Batch Size: {testLoader.batch_size}')

            return trainLoader, devLoader, testLoader
    else:
        # Train - Test separation
        print('Training --- Test    SPLIT')
        train_data, test_data, train_labels, test_labels, _, _ = train_test_split_aux(x_data, y_labels, split_list[1] * 100, t
        print('--------------------------')

        # Tensor Conversion
        train_data_tensor = torch.tensor(train_data).float()
        train_labels_tensor = torch.tensor(train_labels, dtype=torch.int64).squeeze()
        test_data_tensor = torch.tensor(test_data).float()
        test_labels_tensor = torch.tensor(test_labels, dtype=torch.int64).squeeze()
        if debug is True:
            print(f'Training Data        Shape: {train_data.shape}')
            print(f'Testing Data         Shape: {test_data.shape}')

            # Balance Evaluation
            print(f'Training Set    Balance: {np.mean(train_labels)}')
            print(f'Testing Set     Balance: {np.mean(test_labels)}')

        if output != 'Loaders':
            return train_data_tensor, train_labels_tensor, test_data_tensor, test_labels_tensor
        else:
            # PyTorch Dataset Conversion
            train_dataset = torch.utils.data.TensorDataset(torch.tensor(train_data).float(), torch.tensor(train_labels, dtype=
            test_dataset = torch.utils.data.TensorDataset(torch.tensor(test_data).float(), torch.tensor(test_labels, dtype=tor

            # DataLoader (Batches) --> Drop-Last control to optimize training
            trainLoader = DataLoader(train_dataset, shuffle=False, batch_size = 32, drop_last=True)
            testLoader = DataLoader(test_dataset, shuffle=False, batch_size = test_dataset.tensors[0].shape[0])
            if debug is True:
                print(f'Training Set    Batch Size: {trainLoader.batch_size}')
                print(f'Testing Set     Batch Size: {testLoader.batch_size}')

            return trainLoader, testLoader
```

One-Class Architecture (Binary Classifier)

(see "*One-Class_Sub-Network_Analysis.ipynb*")

```
Multi-Layer Perceptron
- Input Layer: 3 features [formant ratios, min-max normalized]
- Hidden Layer: 100 units
- Output Layer: 1 normalized probability
- Learning Rate: 0.0001 (10^-4)
- Optimizer: Adam (Adaptive Momentum)

- Mini-Batch Training:
    . Re-iterated Sub-Dataset Shuffling
    . Early Stopping (Test Accuracy driven)
    . Batch size = 32

- Regularization:
```

```
. Weight Decay (L2 Penalty): 0.0001 (10^-4)
. DropOut:
    * Input Layer Drop Rate: 0.8
    * Hidden Layer Drop Rate: 0.5.
. Batch Normalization
```

- *MLP Classifier Architecture* class definition
- *Mini-Batch Training* function definition

```
# Dynamic Multi-Layer Architecture Class (w. units, activation function, batch normalization and dropOut rate specification)
class binaryClassifier(nn.Module):                                              # nn.Module: base class to inherit from
    def __init__(self, n_units, act_fun, rate_in, rate_hidden, model_name):                 # self + attributes (architectur
        super().__init__()

        self.layers = nn.ModuleDict()                                           # Dictionary to store Model layers
        self.name = model_name

        # Input Layer
        self.layers['input'] = nn.Linear(3, n_units)                           # Key 'input' layer specification

        # Hidden Layer
        self.layers[f'hidden'] = nn.Linear(n_units, n_units)
        self.layers[f'batch_norm'] = nn.BatchNorm1d(n_units)

        # Output Layer
        self.layers['output'] = nn.Linear(n_units, 1)                          # Key 'output' layer specification

        # Activation Function
        self.actfun = act_fun                                                   # Function string-name attribute association

        # Dropout Parameter
        self.dr_in = rate_in
        self.dr_hidden = rate_hidden

        # Weights & Bias initialization
        for layer in self.layers.keys():
            try:
                nn.init.kaiming_normal_(self.layers[layer].weight, mode='fan_in') # Kaiming He - Normal Distributed (ReLU spec
            except:
                pass                                                            # Batch_norm Layer can't be initialized
            self.layers[layer].bias.data.fill_(0.)                              # Bias initialization (0.)

    # Forward Pass Method
    def forward(self, x):

        # Activation function object computation
        actfun = getattr(torch.nn, self.actfun)

        # Input Layer pass                                                      --> Weightening (Dot Product) "Linear transf
        x = actfun()(self.layers['input'](x))
        x = F.dropout(x, p=self.dr_in, training=self.training)                  # Activate DropOut only when Model Training

        # Single Hidden Layer pass                                              --> Weightening (Dot Product) "Linear transf
        x = self.layers[f'batch_norm'](x)                                       # Apply batch normalization before hidden la
        x = actfun()(self.layers[f'hidden'](x))
        x = F.dropout(x, p=self.dr_hidden, training=self.training)              # Same as "Input pass"

        # Output Layer pass                                                     --> Output Weightening (Dot Product) "Linear
        x = self.layers['output'](x)
        x = nn.Sigmoid()(x)

        return x


# Batch Training function (w. Adam Optimizer & L2 penalty term) Re-Definition
def mini_batch_train_test(model, weight_decay, epochs: int, learning_rate, train_loader, dev_loader, test_loader, debug=False)
    """
    Train & Test an ANN Architecture via Mini-Batch Training (w. Train/Dev/Test PyTorch Loaders) and Adam Backpropagation Opti
    """
    # Loss Function initialization
    loss_function = nn.BCELoss()

    # Optimizer Algorithm initialization
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=weight_decay)

    # Output list initialization
    train_accuracies = []
    train_losses = []
    dev_accuracies = []
```

```
    # TRAINING Phase
    for epoch in range(epochs):
        model.train()  # TRAINING Switch ON

        batch_accuracies = []
        batch_losses = []

        # Training BATCHES Loop
        for data_batch, labels_batch in train_loader:
            train_predictions = model(data_batch)
            train_loss = loss_function(train_predictions.squeeze(), labels_batch.type(torch.int64).float())
            batch_losses.append(train_loss.detach())

            # Backpropagation
            optimizer.zero_grad()
            train_loss.backward()
            optimizer.step()

            # Accuracy
            train_accuracy = 100 * torch.mean(((train_predictions.squeeze() > 0.5) == labels_batch.type(torch.int64).squeeze()

            # Batch Stats appending
            batch_accuracies.append(train_accuracy.detach())
            batch_losses.append(train_loss.detach())

        # Training Stats appending
        train_accuracies.append(np.mean(batch_accuracies))  # Average of Batch Accuracies = Training step accuracy
        train_losses.append(np.mean(batch_losses))  # Average of Batch Losses = Training step Losses

        # EVALUATION (Dev) Phase
        model.eval()
        with torch.no_grad():
            dev_data_batch, dev_labels_batch = next(iter(dev_loader))
            dev_predictions = model(dev_data_batch)

            dev_accuracy = 100 * torch.mean(((dev_predictions.squeeze() > 0.5) == dev_labels_batch.type(torch.int64).squeeze()

            if debug is True:
                if epoch % 100 == 0:
                    print(f'Epoch {epoch} --> DEV ACCURACY: {dev_accuracy.detach():.3f} %')
                    print('-------------------------------')

            # Evaluation accuracy appending
            dev_accuracies.append(dev_accuracy.detach())

    # TEST Phase
    model.eval()
    with torch.no_grad():
        test_data_batch, test_labels_batch = next(iter(test_loader))
        test_predictions = model(test_data_batch)
        test_accuracy = 100 * torch.mean(((test_predictions.squeeze() > 0.5) == test_labels_batch.type(torch.int64).squeeze())

        if debug is True:
            print(f'TEST ACCURACY: {test_accuracy.detach():.2f} %')
            print('---------------------------------------------------------------------')

    return train_accuracies, train_losses, dev_accuracies, test_accuracy.detach()
```

OCON (One-Class-One-Net) Model

Binary classifiers **Parallelization**

- *Classifiers-Bank* function definition
  - *Models Parameters* inspection
- Classifiers **Sequential** Training & Evaluation
- *Models Parameters State Save/Load* function definition

- MaxNet output algorithm
- Argmax output algorithm

```
def OCON_bank(one_class_function, hidden_units, act_fun, dr_in, dr_hidden, classes_list):
    """
    Create a One-Class-One-Network parallelization bank of an input Sub-Network definition
    """
    # Sub-Net names creation
    models_name_list = []
    for i in range(len(classes_list)):
        models_name_list.append("{}_{}".format(classes_list[i], "subnet"))  # Class name + _subnet
```

```python
    # Sub-Networks instances creation
    sub_nets = []  # Sub Network list initialization

    for i in range(len(models_name_list)):

        torch.manual_seed(SEED)  # Seed re-initialization

        # Sub-Net instance creation
        locals()[models_name_list[i]] = one_class_function(hidden_units, act_fun, dr_in, dr_hidden, models_name_list[i])
        sub_nets.append(locals()[models_name_list[i]])

    return sub_nets


# Load Architecture Parameters State function
def load_model_state(model, state_dict_path):
    """
    Load an existent State Dictionary in a defined model
    """

    model.load_state_dict(torch.load(state_dict_path))
    print(f'Loaded Parameters (from "{state_dict_path}") into: {model.name}')

    return model


# Build The OCON Model
ocon_vowels = OCON_bank(binaryClassifier, 100, 'ReLU', 0.8, 0.5, vowels)  # Best MLP (see "One-Class_Binary_Classifier_Analysi

# Load Pre-Trained Architectures in a fresh Model instance:
#ocon_vowels = OCON_bank(binaryClassifier, 100, 'ReLU', 0.8, 0.5, vowels)  # Best MLP (see "One-Class_Binary_Classifier_Analys
#states_path = ["Trained_models_state/ae_subnet_Params.pth",
#               "Trained_models_state/ah_subnet_Params.pth",
#               "Trained_models_state/aw_subnet_Params.pth",
#               "Trained_models_state/eh_subnet_Params.pth",
#               "Trained_models_state/er_subnet_Params.pth",
#               "Trained_models_state/ei_subnet_Params.pth",
#               "Trained_models_state/ih_subnet_Params.pth",
#               "Trained_models_state/iy_subnet_Params.pth",
#               "Trained_models_state/oa_subnet_Params.pth",
#               "Trained_models_state/oo_subnet_Params.pth",
#               "Trained_models_state/uh_subnet_Params.pth",
#               "Trained_models_state/uw_subnet_Params.pth"]
#
#for i in range(len(ocon_vowels)):
#    load_model_state(ocon_vowels[i], states_path[i])


# OCON Evaluation function
def OCON_eval(ocon_models_bank, features_dataset: np.ndarray = x_data_minmax[:, 1:], labels: np.ndarray = y_labels_raw_np):
    """
    Evaluate OCON models-bank over an entire dataset
    """
    # Output lists initialization
    predictions = []
    dist_errors = []
    eval_accuracies = []
    g_truths = []  # For plotting purpouses

    # Evaluate each Sub-Network...
    for i in range(len(ocon_models_bank)):
        ocon_models_bank[i].eval()  # Put j-esimal Sub-Network in Evaluation Mode
        print(f'{ocon_models_bank[i].name.upper()} Evaluation -', end=' ')

        with torch.no_grad():

            # Make predictions
            features_data_tensor = torch.tensor(features_dataset).float()
            raw_eval_predictions = ocon_models_bank[i](features_data_tensor)

            # Create Ground Truths
            ground_truth = np.where(labels == i, 1, 0)
            ground_truth_tensor = torch.tensor(ground_truth, dtype=torch.int64).squeeze()

            # Compute Errors
            dist_error = ground_truth_tensor - raw_eval_predictions.detach().squeeze()  # Distances
            eval_accuracy = 100 * torch.mean(((raw_eval_predictions.detach().squeeze() > 0.5) == ground_truth_tensor).float())
            print(f'Accuracy: {eval_accuracy:.2f}%')

        # Outputs append
        predictions.append(raw_eval_predictions.detach())
        dist_errors.append(dist_error.detach())
```

```
        eval_accuracies.append(eval_accuracy.detach())

        g_truths.append(ground_truth)

    return predictions, dist_errors, eval_accuracies, g_truths

# For Pre-Trained Models
#ocon_predictions, ocon_dist_errors, ocon_eval_accuracies, ocon_g_truths = OCON_eval(ocon_vowels)


# Model Parameters State function
def model_desc(model):
    """
    Print a Console report of Neural Network Model parameters
    """
    # Parameters Description
    print('Params Description:')
    trainable_params = 0

    for parameter in model.named_parameters():
        print(f'Parameter Name      : {parameter[0]}')
        print(f'Parameter Weights   : {parameter[1][:]}')
        if parameter[1].requires_grad:
            print(f'...with {parameter[1].numel()} TRAINABLE parameters')
            trainable_params += parameter[1].numel()

        print('................................')

    print('-----------------------------------------------------------------')

    # Nodes Count
    nodes = 0
    for param_name, param_tensor in model.named_parameters():
        if 'bias' in param_name:
            nodes += len(param_tensor)

    print(f'Total Nodes             : {nodes}')
    print('-----------------------------------------------------------------')

# OCON-Model Description
for i in range(len(ocon_vowels)):
    print(f'OCON "{ocon_vowels[i].name}" Classifier STATE')
    model_desc(ocon_vowels[i])
    print()


# Training/Eval/Testing Parameters
epochs = 1000  # For each "Data Batch-Set"
loss_break = 0.15  # loss (for Early Stopping)
acc_break = 90.  # % accuracy (for Early Stopping)
min_tolerance = 0.01 # ...for sub-dataset balancing

# Outputs Initialization
loss_functions = [[] for _ in range(len(ocon_vowels))]
training_accuracies = [[] for _ in range(len(ocon_vowels))]
evaluation_accuracies = [[] for _ in range(len(ocon_vowels))]
test_accuracies = [[] for _ in range(len(ocon_vowels))]
training_times = []


# OCON Sub-Networks Training
from time import perf_counter
debug = False

for i, vowel in enumerate(vowels):
    print(f'Architecture "{ocon_vowels[i].name}" TRAINING PHASE')

    start_timer = perf_counter()
    # Iterated (w. Batch-Sets shuffling) Mini-Batch Training
    iteration = 0  # Batch Training iteration counter
    mean_loss = 1.
    test_accuracy = 0.

    while (mean_loss > loss_break) or (test_accuracy < acc_break):
        # Dataset processing
        sub_data, sub_data_labels_bin, _ = one_hot_encoder(sel_class_number=i, dataset=x_data_minmax, debug=debug)
        print('---------------------------------')
        trainLoader, devLoader, testLoader = train_dev_test_split(sub_data[:, 1:], sub_data_labels_bin, [0.4, 0.3, 0.3], toler

        # Train/Test Architecture
        train_accuracies, train_losses, dev_accuracies, test_accuracy = mini_batch_train_test(ocon_vowels[i], weight_decay=0.0
        print(f'Sub-Net "{vowel.upper()}" Epoch {(iteration + 1) * epochs} - TEST ACCURACY: {test_accuracy:.2f}%', end=' ')
```

```
        # Outputs append
        loss_functions[i].append(train_losses)
        training_accuracies[i].append(train_accuracies)
        evaluation_accuracies[i].append(dev_accuracies)
        test_accuracies[i].append(test_accuracy)

        # Repeating condition evaluation
        mean_loss = np.mean(train_losses[-50: ])  # Last 100 losses mean
        print(f'- MEAN LOSS: {mean_loss}')

        iteration += 1  # Go to next Batch training iteration

    print(f'Training STOPPED at iteration {iteration}')
    print('---------------------------------------------------------------------')
    stop_timer = perf_counter()

    print(f'"{ocon_vowels[i].name}" Training COMPLETED in {float(stop_timer - start_timer)}sec.')
    training_times.append(stop_timer - start_timer)
    print('---------------------------------------------------------------------')


# Graphical smoothing filter
def smooth(data, k=100):
    """
    A Convolution LP filter w. interval definition
    """
    return np.convolve(data, np.ones(k) / k, mode='same')


# Training Phase Plots
plt.figure(figsize=(12, 5 * 12))

# loss_functions, training_accuracies, evaluation_accuracies, test_accuracies, training_times
classes = len(ocon_vowels)

for i in range(classes):
    plt.subplot(classes, 2, (i * 2) + 1)
    flat_loss_function = [item for sublist in loss_functions[i] for item in sublist]

    plt.plot(smooth(flat_loss_function), 'k-')
    plt.axhline(loss_break, color='r', linestyle='--')
    plt.title(f'{ocon_vowels[i].name.upper()} Training Loss')
    plt.xlabel('Epochs')
    plt.xlim([100, len(flat_loss_function) - 100])
    plt.ylabel('GT - Predicted diff. (probability)')
    plt.grid()

    plt.subplot(classes, 2, (i * 2) + 2)
    flat_training_accuracy = [item for sublist in training_accuracies[i] for item in sublist]
    flat_dev_accuracy = [item for sublist in evaluation_accuracies[i] for item in sublist]
    flat_test_accuracy = test_accuracies[i]

    plt.plot(smooth(flat_training_accuracy), 'k-', label='Training')
    plt.plot(smooth(flat_dev_accuracy), color='grey', label='Development')
    if len(flat_test_accuracy) > 1:
        plt.plot([(n + 1) * epochs for n in range(len(flat_test_accuracy))], flat_test_accuracy, 'r-', label=f'Test')
    else:
        plt.axhline(test_accuracy, color='r', linestyle='-', label=f'Test')
    plt.title(f'{ocon_vowels[i].name.upper()} Accuracy (after {training_times[i]:.2f}sec.)')
    plt.xlabel('Epochs')
    plt.xlim([100, len(flat_training_accuracy) - 100])
    plt.ylabel('Accuracy (in %)')
    plt.ylim([40, 101])
    plt.grid()
    plt.legend(loc='best')

plt.tight_layout()
plt.savefig('OCON_training_phase')
plt.show()


# OCON Evaluation
ocon_predictions, ocon_dist_errors, ocon_eval_accuracies, ocon_g_truths = OCON_eval(ocon_vowels)


# Dataset Evaluation Analysis Plot
plt.figure(figsize=(18, 5 * len(ocon_vowels)))
plot_ticks = end_idx[:]
plot_ticks = np.delete(plot_ticks, -1)

for i in range(len(ocon_vowels)):
    plt.subplot(len(ocon_vowels), 3, (i * 3) + 1)
    plt.plot(ocon_predictions[i], 'k.', label='Raw Predictions')
    plt.plot(ocon_g_truths[i], 'rx', label='Ground Truths')
```

```python
    plt.axhline(0.5, linestyle='--', color='grey')
    plt.title(f'{ocon_vowels[i].name.upper()} Predictions Accuracy: {ocon_eval_accuracies[i]:.2f}%')
    plt.xlabel('Data (Indices)')
    plt.xticks(ticks=plot_ticks, labels=vowels)
    plt.ylabel('Normalized Probability')
    plt.grid()
    plt.legend(loc='best')

    plt.subplot(len(ocon_vowels), 3, (i * 3) + 2)
    plt.plot(ocon_dist_errors[i], 'k')
    plt.title(f'Predicted to Measured Error')
    plt.xlabel('Data (Indices)')
    plt.xticks(ticks=plot_ticks, labels=vowels)
    plt.ylabel('Normalized Probability Error')
    plt.ylim([-1.1, 1.1])
    plt.grid()

    plt.subplot(len(ocon_vowels), 3, (i * 3) + 3)

    # Predictions list processing
    predictions_temp = ocon_predictions[i]
    class_predictions = [item for sublist in predictions_temp for item in sublist]  # Turn a list of lists in a single list (c
    for j in range(len(class_predictions)):  # Turn a list of tensors of one variable in a list of scalars (item() method)
        class_predictions[j] = class_predictions[j].item()

    # Positives & False-Positives extraction
    positives = []
    for w in range(len(vowels)):
        num = (np.array(class_predictions[end_idx[w]: end_idx[w + 1]]) > 0.5).sum()
        positives.append(num)

    plt.bar(np.arange(len(vowels)), positives, color='k')
    plt.title(f'"{vowels[i]}" Positive Probabilities Distribution')
    plt.xlabel('Normalized Probabilities')
    plt.ylabel('Occurences')
    plt.xticks([n for n in range(12)], vowels)
    plt.grid()

plt.tight_layout()
plt.savefig('OCON_bank_evaluation')
plt.show()


# Model Parameters Save/Load functions
from pathlib import Path

def save_model_state(model, folder_name: str = "Trained_models_state"):
    """
    Save Pre-Trained model parameters in a State Dictionary
    """

    MODEL_PATH = Path(folder_name)  # Placed in root
    MODEL_PATH.mkdir(parents=True, exist_ok=True)  # Pre-existing folder (w. same name) monitoring
    MODEL_NAME = '{}_{}'.format(model.name, "Params.pth")
    MODEL_SAVE_PATH = MODEL_PATH / MODEL_NAME

    print(f"Saving {model.name} Parameters in: {MODEL_SAVE_PATH}")
    torch.save(obj=model.state_dict(), f=MODEL_SAVE_PATH)

    return MODEL_SAVE_PATH

# Save Pre-Trained Models-bank
states_path = []  # Path for each model parameters state
for i in range(len(ocon_vowels)):
    state_path = save_model_state(ocon_vowels[i])
    states_path.append(state_path)
print()
```

Output **Maxnet Algorithm**

```python
# OCON "MaxNet" Architecture (Weightening + Non Linearity apply)
class OCON_MaxNet(nn.Module):                                        # nn.Module: base class to inherit from
    def __init__(self, n_units, act_fun, eps):                             # self + attributes (architecture hyper-paramet
        super().__init__()

        self.layers = nn.ModuleDict()                                # Dictionary to store Model layers
        self.eps_weight = eps

        # MaxNet Layer
        self.layers['MAXNET'] = nn.Linear(n_units, n_units)          # Key 'MaxNet' layer specification
```

```python
        # Weights & Bias initialization
        self.layers['MAXNET'].weight.data.fill_(self.eps_weight)
        for i in range(n_units):
            self.layers['MAXNET'].weight[i][i].data.fill_(1.)  # Self Weight = 1

        self.layers['MAXNET'].bias.data.fill_(0.)

        # Activation Function
        self.actfun = act_fun  # Function string-name attribute association

    # Forward Pass Method
    def forward(self, x):

        # Activation function object computation
        actfun = getattr(torch.nn, self.actfun)

        # Maxnet Layer pass                                         --> Output Weightening (Dot Product) "Linear transfc
        x = actfun()(self.layers['MAXNET'](x.squeeze().float()))

        # Self

        return x


# Build OCON MaxNetwork Architecture
torch.manual_seed(SEED)
ocon_maxnet = OCON_MaxNet(n_units=12, act_fun='ReLU', eps=0.25)

# MaxNet & Sub-Networks Parameters
print('OCON MaxNet STATE')
model_desc(ocon_maxnet)


def maxnet_algo(maxnet_function, n_units, act_fun, eps, input_array):
    """
    MaxNet Re-iteration algorithm for Maximum Value retrieving from an input array
    """
    non_zero_outs = np.count_nonzero(input_array)  # Non Zero Values initialization
    maxnet_in = torch.from_numpy(input_array)  # MaxNet Input Tensor initialization

    results = []  # Results initialization

    counter = 0
    while non_zero_outs != 1:
        counter += 1

        # Create the MaxNet
        torch.manual_seed(SEED)  # Redundant
        maxnet = maxnet_function(n_units = n_units, act_fun = act_fun, eps = eps)

        # Compute Forward Pass
        results = maxnet(maxnet_in)

        # Non_zero outputs & Maxnet Input Update
        non_zero_outs = np.count_nonzero(results.detach().numpy())
        maxnet_in = results.detach()  # Save results for next iteration

    print(f'Maximum Value found in {counter} iterations')
    return np.argmax(results.detach().numpy())


# MaxNet on Sub-Networks predictions
ocon_predictions_prob = np.zeros((len(ocon_predictions), x_data_minmax.shape[0]))  # NumPy predictions matrix (12 * 1617)

# Convert from List of Tensors to 2D NumPy Array
for i in range(len(ocon_predictions)):
    ocon_predictions_prob[i, :] = ocon_predictions[i].detach().squeeze().numpy()

maxnet_class_predictions = []  # Classes Outputs list initialization
# MaxNet application
for i in range(x_data_minmax.shape[0]):
    print(f'Dataset Sample({i + 1}) Class Evaluation')

    samp_predictions = ocon_predictions_prob[:, i]  # Array of 12 predictions for each Dataset sample (OCON outputs)
    class_prediction = maxnet_algo(OCON_MaxNet, n_units=12, act_fun='ReLU', eps=-0.1, input_array=samp_predictions)  # MaxNet
    maxnet_class_predictions.append(class_prediction)  # Result appending
    print('-----------------------------------------------')

maxnet_accuracy = 100 * np.mean((np.array(maxnet_class_predictions).reshape(1090, 1) == y_labels_raw_np))  # Accuracy computat
print(f'Maxnet Output --> Phoneme ACCURACY: {maxnet_accuracy}%')
```

```
# Argmax on Sub-Networks predictions (...for multiple 1s probabilities MaxNet infinite loops)
#ocon_predictions_prob = np.zeros((len(new_ocon_predictions), x_data_minmax.shape[0]))  # NumPy predictions matrix (12 * 1617)
#
# Convert from List of Tensors to 2D NumPy Array
#for i in range(len(new_ocon_predictions)):
#    ocon_predictions_prob[i, :] = new_ocon_predictions[i].detach().squeeze().numpy()
#
#maxnet_class_predictions = np.argmax(ocon_predictions_prob, axis=0)
#maxnet_accuracy = 100 * np.mean((np.array(maxnet_class_predictions).reshape(1617, 1) == y_labels_raw_np))  # Accuracy computa
#print(f'Maxnet Output ACCURACY: {maxnet_accuracy}%')


# Evaluation Analysis Plot
plt.figure(figsize=(30, 5))
plt.suptitle(f'OCON Bank + MaxNet Evaluation: {maxnet_accuracy:.0f}%')

plot_x_ticks = end_idx[:]
plot_x_ticks = np.delete(plot_x_ticks, len(end_idx) - 1)
plot_y_ticks = [n for n in range(len(vowels))]

plt.plot(y_labels_raw_np, 'rs', label='Ground Truths')
plt.plot(maxnet_class_predictions, 'k.', label='MAXNET Outputs')
plt.xlabel('Dataset samples')
plt.xticks(ticks=plot_x_ticks, labels=vowels)
plt.xlim([-10, len(y_labels_raw_np) + 10])
plt.ylabel('Labels')
plt.yticks(ticks=plot_y_ticks, labels=vowels)
plt.legend(loc='best')
plt.grid()

plt.tight_layout()
plt.savefig('OCON_model_evaluation')
plt.show()
```

OCON Model Analysis

(12-features Complete Dataset)

Author: S. Giacomelli

Year: 2023

Affiliation: A.Casella Conservatory (student)

Master Degree Thesis: "*Vowel phonemes Analysis & Classification by means of OCON rectifiers Deep Learning Architectures*"

Description: Python scripts for One-Class-One-Network (OCON) Model analysis and optimization

```
# Numerical computations packages/modules
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

# Dataset processing modules
from torch.utils.data import DataLoader
from sklearn.model_selection import train_test_split

# Graphic visualization modules
import matplotlib.pyplot as plt
import matplotlib_inline
matplotlib_inline.backend_inline.set_matplotlib_formats('svg')

# Common Seed initialization
SEED = 42  # ... the answer to the ultimate question of Life, the Universe, and Everything... (cit.)

# PyTorch Processing Units evaluation
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(f'AVILABLE Processing Unit: {device.upper()}')
!nvidia-smi
```

HGCW Dataset

- *Dataset_utils.npz* file read
- *One-Hot encoding* definition
- *Train/Dev/Test split* definition

```
# Load HGCW 12_features_all Dataset
HGCW_dataset_utils = np.load(file='./HGCW_dataset_utils.npz')
print('Raw features                    Data shape:', HGCW_dataset_utils['HGCW_raw'].shape)
print('Fundamental Normalized features Data shape:', HGCW_dataset_utils['HGCW_fund_norm'].shape)
print('MinMax features                 Data shape:', HGCW_dataset_utils['HGCW_minmax'].shape)
print('Labels                          Data shape:', HGCW_dataset_utils['HGCW_labels'].shape)
print('Classes size                    Data shape:', HGCW_dataset_utils['classes_size'].shape)
print('Classes indices                 Data shape:', HGCW_dataset_utils['classes_idx'].shape)

x_data_raw_np = HGCW_dataset_utils['HGCW_raw']
x_data_fund_norm = HGCW_dataset_utils['HGCW_fund_norm']
x_data_minmax = HGCW_dataset_utils['HGCW_minmax']
y_labels_raw_np = HGCW_dataset_utils['HGCW_labels']
vow_size = HGCW_dataset_utils['classes_size']
end_idx = HGCW_dataset_utils['classes_idx']

# Auxiliary lists
vowels = ['ae', 'ah', 'aw', 'eh', 'er', 'ei', 'ih', 'iy', 'oa', 'oo', 'uh', 'uw']  # Vowels list
colors = ['red', 'saddlebrown', 'darkorange', 'darkgoldenrod', 'gold', 'darkkhaki', 'olive', 'darkgreen', 'steelblue', 'fuchsi


# Class-specific One-hot encoding (Binarization)
def one_hot_encoder(sel_class_number: int = 3, dataset: np.ndarray = x_data_minmax, orig_labels: int = len(vowels), classes_si

    classes = [n for n in range(orig_labels)]  # Class Labels list initialization

    # Auxiliary Parameters Initialization
    if sel_class_number < len(classes):
        classes.remove(sel_class_number)  # REST Classes list
        if debug is True:
            print(f'Selected Class "{vowels[sel_class_number]}" : {classes_size[sel_class_number]} samples')
        sub_classes_size = classes_size[sel_class_number] // len(classes)
        if debug is True:
            print(f'Rest Classes size (...each): {sub_classes_size} samples')
```

```python
        # 1-Subset processing
        sub_data = dataset[classes_idx[sel_class_number]: classes_idx[sel_class_number + 1], :]  # Selected Class feature slic
        sub_data_labels_bin = np.ones((classes_size[sel_class_number], 1), dtype='int')  # Selected Class labels (1) creation
        sub_data_labels = np.ones((classes_size[sel_class_number], 1), dtype='int') * sel_class_number

        # 0-Subset processing
        for i in classes:
            class_i_indices = np.random.choice(np.arange(classes_idx[i], classes_idx[i + 1], 1), size=sub_classes_size, replac
            sub_class_i_array = dataset[class_i_indices, :]
            sub_class_labels_bin_array = np.zeros((sub_class_i_array.shape[0], 1), dtype='int')  # Rest I-esimal Class labels
            sub_class_labels_array = np.ones((sub_class_i_array.shape[0], 1), dtype='int') * i

            # Outputs append
            sub_data = np.vstack((sub_data, sub_class_i_array))
            sub_data_labels_bin = np.vstack((sub_data_labels_bin, sub_class_labels_bin_array))
            sub_data_labels = np.vstack((sub_data_labels, sub_class_labels_array))
    else:
        raise ValueError(f'Invalid Class ID: "{sel_class_number}" --> It must be less than {len(classes)}!')

    return sub_data, sub_data_labels_bin, sub_data_labels


# Train/Test split (auxiliary function)
def train_test_split_aux(features_dataset, labels_dataset, test_perc, tolerance):
    """
    An auxiliary Train_Test_split function (based on Scikit Learn implementation) w. balance tolerance specification
    """
    test_size = int(test_perc / 100 * len(features_dataset))
    train_balance = 0  # Output Training set balance value initialization
    test_balance = 0  # Output Testing set balance value initialization

    min_tol = np.mean(labels_dataset) - tolerance
    max_tol = np.mean(labels_dataset) + tolerance
    print(f'Data Balancing  (TARGET = {np.mean(labels_dataset)} +- {tolerance}): ', end='')

    while (min_tol >= train_balance or train_balance >= max_tol) or (min_tol >= test_balance or test_balance >= max_tol):
        train_data, test_data, train_labels, test_labels = train_test_split(features_dataset, labels_dataset, test_size=test_s
        train_balance = np.mean(train_labels)
        test_balance = np.mean(test_labels)
        print('.', end='')
    else:
        print('OK')

    return train_data, test_data, train_labels, test_labels, train_balance, test_balance

# Train-Dev-Test split function
def train_dev_test_split(x_data, y_labels, split_list, tolerance=0.1, output='Loaders', debug=False):
    """
    Compute a Train, Development (Hold-Out) and a Test set split w. PyTorch Dataset conversion (and eventual Loaders initializ
    """
    if len(split_list) == 3:
        # Train - Dev+Test separation
        print('Training --- Devel/Test SPLIT')
        train_data, testTMP_data, train_labels, testTMP_labels, _, _ = train_test_split_aux(x_data, y_labels, (split_list[1] *
        print('----------------------------------')

        # Dev - Test separation
        print('Devel    ---     Test SPLIT')

        split = ((split_list[1] * 100) / np.sum(split_list[1:] * 100)) * 100  # Split in %
        dev_data, test_data, dev_labels, test_labels, _, _ = train_test_split_aux(testTMP_data, testTMP_labels, split, toleran
        print('----------------------------------')

        # Tensor Conversion
        train_data_tensor = torch.tensor(train_data).float()
        train_labels_tensor = torch.tensor(train_labels, dtype=torch.int64).squeeze()
        dev_data_tensor = torch.tensor(dev_data).float()
        dev_labels_tensor = torch.tensor(dev_labels, dtype=torch.int64).squeeze()
        test_data_tensor = torch.tensor(test_data).float()
        test_labels_tensor = torch.tensor(test_labels, dtype=torch.int64).squeeze()
        if debug is True:
            print(f'Training Data        Shape: {train_data.shape}')
            print(f'Development Data     Shape: {dev_data.shape}')
            print(f'Testing Data         Shape: {test_data.shape}')

            # Balance Evaluation
            print(f'Training Set       Balance: {np.mean(train_labels)}')
            print(f'Development Set    Balance: {np.mean(dev_labels)}')
            print(f'Testing Set        Balance: {np.mean(test_labels)}')

        if output != 'Loaders':
            return train_data_tensor, train_labels_tensor, dev_data_tensor, dev_labels_tensor, test_data_tensor, test_labels_t
```

```
        else:
            # PyTorch Dataset Conversion
            train_dataset = torch.utils.data.TensorDataset(torch.tensor(train_data).float(), torch.tensor(train_labels, dtype=
            dev_dataset = torch.utils.data.TensorDataset(torch.tensor(dev_data).float(), torch.tensor(dev_labels, dtype=torch.
            test_dataset = torch.utils.data.TensorDataset(torch.tensor(test_data).float(), torch.tensor(test_labels, dtype=tor

            # DataLoader (Batches) --> Drop-Last control to optimize training
            trainLoader = DataLoader(train_dataset, shuffle=False, batch_size = 32, drop_last=True)
            devLoader = DataLoader(dev_dataset, shuffle=False, batch_size = dev_dataset.tensors[0].shape[0])
            testLoader = DataLoader(test_dataset, shuffle=False, batch_size = test_dataset.tensors[0].shape[0])
            if debug is True:
                print(f'Training Set    Batch Size: {trainLoader.batch_size}')
                print(f'Development Set Batch Size: {devLoader.batch_size}')
                print(f'Testing Set     Batch Size: {testLoader.batch_size}')

            return trainLoader, devLoader, testLoader
    else:
        # Train - Test separation
        print('Training --- Test    SPLIT')
        train_data, test_data, train_labels, test_labels, _, _ = train_test_split_aux(x_data, y_labels, split_list[1] * 100, t
        print('-------------------------')

        # Tensor Conversion
        train_data_tensor = torch.tensor(train_data).float()
        train_labels_tensor = torch.tensor(train_labels, dtype=torch.int64).squeeze()
        test_data_tensor = torch.tensor(test_data).float()
        test_labels_tensor = torch.tensor(test_labels, dtype=torch.int64).squeeze()
        if debug is True:
            print(f'Training Data        Shape: {train_data.shape}')
            print(f'Testing Data         Shape: {test_data.shape}')

            # Balance Evaluation
            print(f'Training Set    Balance: {np.mean(train_labels)}')
            print(f'Testing Set     Balance: {np.mean(test_labels)}')

        if output != 'Loaders':
            return train_data_tensor, train_labels_tensor, test_data_tensor, test_labels_tensor
        else:
            # PyTorch Dataset Conversion
            train_dataset = torch.utils.data.TensorDataset(torch.tensor(train_data).float(), torch.tensor(train_labels, dtype=
            test_dataset = torch.utils.data.TensorDataset(torch.tensor(test_data).float(), torch.tensor(test_labels, dtype=tor

            # DataLoader (Batches) --> Drop-Last control to optimize training
            trainLoader = DataLoader(train_dataset, shuffle=False, batch_size = 32, drop_last=True)
            testLoader = DataLoader(test_dataset, shuffle=False, batch_size = test_dataset.tensors[0].shape[0])
            if debug is True:
                print(f'Training Set    Batch Size: {trainLoader.batch_size}')
                print(f'Testing Set     Batch Size: {testLoader.batch_size}')

            return trainLoader, testLoader
```

One-Class Architecture (Binary Classifier)

(see "*One-Class_Sub-Network_Analysis.ipynb*")

```
Multi-Layer Perceptron
- Input Layer: 3 features [formant ratios, min-max normalized]
- Hidden Layer: 100 units
- Output Layer: 1 normalized probability
- Learning Rate: 0.0001 (10^-4)
- Optimizer: Adam (Adaptive Momentum)

- Mini-Batch Training:
    . Re-iterated Sub-Dataset Shuffling
    . Early Stopping (Test Accuracy driven)
    . Batch size = 32

- Regularization:
    . Weight Decay (L2 Penalty): 0.0001 (10^-4)
    . DropOut:
        * Input Layer Drop Rate: 0.8
        * Hidden Layer Drop Rate: 0.5.
    . Batch Normalization
```

- *MLP Classifier Architecture* class definition
- *Mini-Batch Training* function definition

```
# Dynamic Multi-Layer Architecture Class (w. units, activation function, batch normalization and dropOut rate specification)
class binaryClassifier(nn.Module):                                                      # nn.Module: base class to inherit from
    def __init__(self, n_units, act_fun, rate_in, rate_hidden, model_name):                   # self + attributes (architectur
        super().__init__()

        self.layers = nn.ModuleDict()                                                   # Dictionary to store Model layers
        self.name = model_name

        # Input Layer
        self.layers['input'] = nn.Linear(12, n_units)                                    # Key 'input' layer specification

        # Hidden Layer
        self.layers[f'hidden'] = nn.Linear(n_units, n_units)
        self.layers[f'batch_norm'] = nn.BatchNorm1d(n_units)

        # Output Layer
        self.layers['output'] = nn.Linear(n_units, 1)                                   # Key 'output' layer specification

        # Activation Function
        self.actfun = act_fun                                                           # Function string-name attribute association

        # Dropout Parameter
        self.dr_in = rate_in
        self.dr_hidden = rate_hidden

        # Weights & Bias initialization
        for layer in self.layers.keys():
            try:
                nn.init.kaiming_normal_(self.layers[layer].weight, mode='fan_in') # Kaiming He - Normal Distributed (ReLU spec
            except:
                pass                                                                    # Batch_norm Layer can't be initialized
            self.layers[layer].bias.data.fill_(0.)                                      # Bias initialization (0.)

    # Forward Pass Method
    def forward(self, x):

        # Activation function object computation
        actfun = getattr(torch.nn, self.actfun)

        # Input Layer pass                                                              --> Weightening (Dot Product) "Linear transf
        x = actfun()(self.layers['input'](x))
        x = F.dropout(x, p=self.dr_in, training=self.training)                          # Activate DropOut only when Model Training

        # Single Hidden Layer pass                                                      --> Weightening (Dot Product) "Linear transf
        x = self.layers[f'batch_norm'](x)                                               # Apply batch normalization before hidden la
        x = actfun()(self.layers[f'hidden'](x))
        x = F.dropout(x, p=self.dr_hidden, training=self.training)                      # Same as "Input pass"

        # Output Layer pass                                                             --> Output Weightening (Dot Product) "Linear
        x = self.layers['output'](x)
        x = nn.Sigmoid()(x)

        return x


# Batch Training function (w. Adam Optimizer & L2 penalty term) Re-Definition
def mini_batch_train_test(model, weight_decay, epochs: int, learning_rate, train_loader, dev_loader, test_loader, debug=False)
    """
    Train & Test an ANN Architecture via Mini-Batch Training (w. Train/Dev/Test PyTorch Loaders) and Adam Backpropagation Opti
    """
    # Loss Function initialization
    loss_function = nn.BCELoss()

    # Optimizer Algorithm initialization
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=weight_decay)

    # Output list initialization
    train_accuracies = []
    train_losses = []
    dev_accuracies = []

    # TRAINING Phase
    for epoch in range(epochs):
        model.train()  # TRAINING Switch ON

        batch_accuracies = []
        batch_losses = []

        # Training BATCHES Loop
        for data_batch, labels_batch in train_loader:
            train_predictions = model(data_batch)
            train_loss = loss_function(train_predictions.squeeze(), labels_batch.type(torch.int64).float())
```

```
            batch_losses.append(train_loss.detach())

            # Backpropagation
            optimizer.zero_grad()
            train_loss.backward()
            optimizer.step()

            # Accuracy
            train_accuracy = 100 * torch.mean(((train_predictions.squeeze() > 0.5) == labels_batch.type(torch.int64).squeeze()

            # Batch Stats appending
            batch_accuracies.append(train_accuracy.detach())
            batch_losses.append(train_loss.detach())

        # Training Stats appending
        train_accuracies.append(np.mean(batch_accuracies))  # Average of Batch Accuracies = Training step accuracy
        train_losses.append(np.mean(batch_losses))  # Average of Batch Losses = Training step Losses

        # EVALUATION (Dev) Phase
        model.eval()
        with torch.no_grad():
            dev_data_batch, dev_labels_batch = next(iter(dev_loader))
            dev_predictions = model(dev_data_batch)

            dev_accuracy = 100 * torch.mean(((dev_predictions.squeeze() > 0.5) == dev_labels_batch.type(torch.int64).squeeze()

            if debug is True:
                if epoch % 100 == 0:
                    print(f'Epoch {epoch} --> DEV ACCURACY: {dev_accuracy.detach():.3f} %')
                    print('-------------------------------')

            # Evaluation accuracy appending
            dev_accuracies.append(dev_accuracy.detach())

    # TEST Phase
    model.eval()
    with torch.no_grad():
        test_data_batch, test_labels_batch = next(iter(test_loader))
        test_predictions = model(test_data_batch)
        test_accuracy = 100 * torch.mean(((test_predictions.squeeze() > 0.5) == test_labels_batch.type(torch.int64).squeeze())

        if debug is True:
            print(f'TEST ACCURACY: {test_accuracy.detach():.2f} %')
            print('--------------------------------------------------------------------')

    return train_accuracies, train_losses, dev_accuracies, test_accuracy.detach()
```

OCON (One-Class-One-Net) Model

Binary classifiers **Parallelization**

- *Classifiers-Bank* function definition
    - *Models Parameters* inspection
- Classifiers **Sequential** Training & Evaluation
- *Models Parameters State Save/Load* function definition

- MaxNet output algorithm
- Argmax output algorithm

```
def OCON_bank(one_class_function, hidden_units, act_fun, dr_in, dr_hidden, classes_list):
    """
    Create a One-Class-One-Network parallelization bank of an input Sub-Network definition
    """
    # Sub-Net names creation
    models_name_list = []
    for i in range(len(classes_list)):
        models_name_list.append("{}_{}".format(classes_list[i], "subnet"))  # Class name + _subnet

    # Sub-Networks instances creation
    sub_nets = []  # Sub Network list initialization

    for i in range(len(models_name_list)):

        torch.manual_seed(SEED)  # Seed re-initialization

        # Sub-Net instance creation
        locals()[models_name_list[i]] = one_class_function(hidden_units, act_fun, dr_in, dr_hidden, models_name_list[i])
        sub_nets.append(locals()[models_name_list[i]])
```

```
    return sub_nets


# Load Architecture Parameters State function
def load_model_state(model, state_dict_path):
    """
    Load an existent State Dictionary in a defined model
    """

    model.load_state_dict(torch.load(state_dict_path))
    print(f'Loaded Parameters (from "{state_dict_path}") into: {model.name}')

    return model


# Build The OCON Model
ocon_vowels = OCON_bank(binaryClassifier, 100, 'ReLU', 0.8, 0.5, vowels)  # Best MLP (see "One-Class_Binary_Classifier_Analysi

# Load Pre-Trained Architectures in a fresh Model instance:
#ocon_vowels = OCON_bank(binaryClassifier, 100, 'ReLU', 0.8, 0.5, vowels)  # Best MLP (see "One-Class_Binary_Classifier_Analys
#states_path = ["Trained_models_state/ae_subnet_Params.pth",
#               "Trained_models_state/ah_subnet_Params.pth",
#               "Trained_models_state/aw_subnet_Params.pth",
#               "Trained_models_state/eh_subnet_Params.pth",
#               "Trained_models_state/er_subnet_Params.pth",
#               "Trained_models_state/ei_subnet_Params.pth",
#               "Trained_models_state/ih_subnet_Params.pth",
#               "Trained_models_state/iy_subnet_Params.pth",
#               "Trained_models_state/oa_subnet_Params.pth",
#               "Trained_models_state/oo_subnet_Params.pth",
#               "Trained_models_state/uh_subnet_Params.pth",
#               "Trained_models_state/uw_subnet_Params.pth"]
#
#for i in range(len(ocon_vowels)):
#    load_model_state(ocon_vowels[i], states_path[i])


# OCON Evaluation function
def OCON_eval(ocon_models_bank, features_dataset: np.ndarray = x_data_minmax[:, 1:], labels: np.ndarray = y_labels_raw_np):
    """
    Evaluate OCON models-bank over an entire dataset
    """
    # Output lists initialization
    predictions = []
    dist_errors = []
    eval_accuracies = []
    g_truths = []  # For plotting purpouses

    # Evaluate each Sub-Network...
    for i in range(len(ocon_models_bank)):
        ocon_models_bank[i].eval()  # Put j-esimal Sub-Network in Evaluation Mode
        print(f'{ocon_models_bank[i].name.upper()} Evaluation -', end=' ')

        with torch.no_grad():

            # Make predictions
            features_data_tensor = torch.tensor(features_dataset).float()
            raw_eval_predictions = ocon_models_bank[i](features_data_tensor)

            # Create Ground Truths
            ground_truth = np.where(labels == i, 1, 0)
            ground_truth_tensor = torch.tensor(ground_truth, dtype=torch.int64).squeeze()

            # Compute Errors
            dist_error = ground_truth_tensor - raw_eval_predictions.detach().squeeze()  # Distances
            eval_accuracy = 100 * torch.mean(((raw_eval_predictions.detach().squeeze() > 0.5) == ground_truth_tensor).float())
            print(f'Accuracy: {eval_accuracy:.2f}%')

        # Outputs append
        predictions.append(raw_eval_predictions.detach())
        dist_errors.append(dist_error.detach())
        eval_accuracies.append(eval_accuracy.detach())

        g_truths.append(ground_truth)

    return predictions, dist_errors, eval_accuracies, g_truths

# For Pre-Trained Models
#ocon_predictions, ocon_dist_errors, ocon_eval_accuracies, ocon_g_truths = OCON_eval(ocon_vowels)


# Model Parameters State function
def model_desc(model):
```

```
    """
    Print a Console report of Neural Network Model parameters
    """
    # Parameters Description
    print('Params Description:')
    trainable_params = 0

    for parameter in model.named_parameters():
        print(f'Parameter Name      : {parameter[0]}')
        print(f'Parameter Weights   : {parameter[1][:]}')
        if parameter[1].requires_grad:
            print(f'...with {parameter[1].numel()} TRAINABLE parameters')
            trainable_params += parameter[1].numel()

        print('................................')

    print('----------------------------------------------------------------')

    # Nodes Count
    nodes = 0
    for param_name, param_tensor in model.named_parameters():
        if 'bias' in param_name:
            nodes += len(param_tensor)

    print(f'Total Nodes             : {nodes}')
    print('----------------------------------------------------------------')

# OCON-Model Description
for i in range(len(ocon_vowels)):
    print(f'OCON "{ocon_vowels[i].name}" Classifier STATE')
    model_desc(ocon_vowels[i])
    print()


# Training/Eval/Testing Parameters
epochs = 1000  # For each "Data Batch-Set"
loss_break = 0.15  # loss (for Early Stopping)
acc_break = 95.  # % accuracy (for Early Stopping)
min_tolerance = 0.01 # ...for sub-dataset balancing

# Outputs Initialization
loss_functions = [[] for _ in range(len(ocon_vowels))]
training_accuracies = [[] for _ in range(len(ocon_vowels))]
evaluation_accuracies = [[] for _ in range(len(ocon_vowels))]
test_accuracies = [[] for _ in range(len(ocon_vowels))]
training_times = []


# OCON Sub-Networks Training
from time import perf_counter
debug = False

for i, vowel in enumerate(vowels):
    print(f'Architecture "{ocon_vowels[i].name}" TRAINING PHASE')

    start_timer = perf_counter()
    # Iterated (w. Batch-Sets shuffling) Mini-Batch Training
    iteration = 0  # Batch Training iteration counter
    mean_loss = 1.
    test_accuracy = 0.

    while (mean_loss > loss_break) or (test_accuracy < acc_break):
        # Dataset processing
        sub_data, sub_data_labels_bin, _ = one_hot_encoder(sel_class_number=i, dataset=x_data_minmax, debug=debug)
        print('----------------------------------')
        trainLoader, devLoader, testLoader = train_dev_test_split(sub_data[:, 1:], sub_data_labels_bin, [0.5, 0.25, 0.25], tol

        # Train/Test Architecture
        train_accuracies, train_losses, dev_accuracies, test_accuracy = mini_batch_train_test(ocon_vowels[i], weight_decay=0.0
        print(f'Sub-Net "{vowel.upper()}" Epoch {(iteration + 1) * epochs} - TEST ACCURACY: {test_accuracy:.2f}%', end=' ')

        # Outputs append
        loss_functions[i].append(train_losses)
        training_accuracies[i].append(train_accuracies)
        evaluation_accuracies[i].append(dev_accuracies)
        test_accuracies[i].append(test_accuracy)

        # Repeating condition evaluation
        mean_loss = np.mean(train_losses[-50: ])  # Last 100 losses mean
        print(f'- MEAN LOSS: {mean_loss}')

        iteration += 1  # Go to next Batch training iteration
```

```python
    print(f'Training STOPPED at iteration {iteration}')
    print('--------------------------------------------------------------------')
    stop_timer = perf_counter()

    print(f'"{ocon_vowels[i].name}" Training COMPLETED in {float(stop_timer - start_timer)}sec.')
    training_times.append(stop_timer - start_timer)
    print('--------------------------------------------------------------------')


# Graphical smoothing filter
def smooth(data, k=100):
    """
    A Convolution LP filter w. interval definition
    """
    return np.convolve(data, np.ones(k) / k, mode='same')


# Training Phase Plots
plt.figure(figsize=(12, 5 * 12))

# loss_functions, training_accuracies, evaluation_accuracies, test_accuracies, training_times
classes = len(ocon_vowels)

for i in range(classes):
    plt.subplot(classes, 2, (i * 2) + 1)
    flat_loss_function = [item for sublist in loss_functions[i] for item in sublist]

    plt.plot(smooth(flat_loss_function), 'k-')
    plt.axhline(loss_break, color='r', linestyle='--')
    plt.title(f'{ocon_vowels[i].name.upper()} Training Loss')
    plt.xlabel('Epochs')
    plt.xlim([100, len(flat_loss_function) - 100])
    plt.ylabel('GT - Predicted diff. (probability)')
    plt.grid()

    plt.subplot(classes, 2, (i * 2) + 2)
    flat_training_accuracy = [item for sublist in training_accuracies[i] for item in sublist]
    flat_dev_accuracy = [item for sublist in evaluation_accuracies[i] for item in sublist]
    flat_test_accuracy = test_accuracies[i]

    plt.plot(smooth(flat_training_accuracy), 'k-', label='Training')
    plt.plot(smooth(flat_dev_accuracy), color='grey', label='Development')
    if len(flat_test_accuracy) > 1:
        plt.plot([(n + 1) * epochs for n in range(len(flat_test_accuracy))], flat_test_accuracy, 'r-', label=f'Test')
    else:
        plt.axhline(test_accuracy, color='r', linestyle='-', label=f'Test')
    plt.title(f'{ocon_vowels[i].name.upper()} Accuracy (after {training_times[i]:.2f}sec.)')
    plt.xlabel('Epochs')
    plt.xlim([100, len(flat_training_accuracy) - 100])
    plt.ylabel('Accuracy (in %)')
    plt.ylim([40, 101])
    plt.grid()
    plt.legend(loc='best')

plt.tight_layout()
plt.savefig('OCON_training_phase')
plt.show()


# OCON Evaluation
ocon_predictions, ocon_dist_errors, ocon_eval_accuracies, ocon_g_truths = OCON_eval(ocon_vowels)


# Dataset Evaluation Analysis Plot
plt.figure(figsize=(18, 5 * len(ocon_vowels)))
plot_ticks = end_idx[:]
plot_ticks = np.delete(plot_ticks, -1)

for i in range(len(ocon_vowels)):
    plt.subplot(len(ocon_vowels), 3, (i * 3) + 1)
    plt.plot(ocon_predictions[i], 'k.', label='Raw Predictions')
    plt.plot(ocon_g_truths[i], 'rx', label='Ground Truths')
    plt.axhline(0.5, linestyle='--', color='grey')
    plt.title(f'{ocon_vowels[i].name.upper()} Predictions Accuracy: {ocon_eval_accuracies[i]:.2f}%')
    plt.xlabel('Data (Indices)')
    plt.xticks(ticks=plot_ticks, labels=vowels)
    plt.ylabel('Normalized Probability')
    plt.grid()
    plt.legend(loc='best')

    plt.subplot(len(ocon_vowels), 3, (i * 3) + 2)
    plt.plot(ocon_dist_errors[i], 'k')
    plt.title(f'Predicted to Measured Error')
    plt.xlabel('Data (Indices)')
```

```
    plt.xticks(ticks=plot_ticks, labels=vowels)
    plt.ylabel('Normalized Probability Error')
    plt.ylim([-1.1, 1.1])
    plt.grid()

    plt.subplot(len(ocon_vowels), 3, (i * 3) + 3)

    # Predictions list processing
    predictions_temp = ocon_predictions[i]
    class_predictions = [item for sublist in predictions_temp for item in sublist]  # Turn a list of lists in a single list (c
    for j in range(len(class_predictions)):  # Turn a list of tensors of one variable in a list of scalars (item() method)
        class_predictions[j] = class_predictions[j].item()

    # Positives & False-Positives extraction
    positives = []
    for w in range(len(vowels)):
        num = (np.array(class_predictions[end_idx[w]: end_idx[w + 1]]) > 0.5).sum()
        positives.append(num)

    plt.bar(np.arange(len(vowels)), positives, color='k')
    plt.title(f'"{vowels[i]}" Positive Probabilities Distribution')
    plt.xlabel('Normalized Probabilities')
    plt.ylabel('Occurences')
    plt.xticks([n for n in range(12)], vowels)
    plt.grid()

plt.tight_layout()
plt.savefig('OCON_bank_evaluation')
plt.show()


# Model Parameters Save/Load functions
from pathlib import Path

def save_model_state(model, folder_name: str = "Trained_models_state"):
    """
    Save Pre-Trained model parameters in a State Dictionary
    """

    MODEL_PATH = Path(folder_name)  # Placed in root
    MODEL_PATH.mkdir(parents=True, exist_ok=True)  # Pre-existing folder (w. same name) monitoring
    MODEL_NAME = '{}_{}'.format(model.name, "Params.pth")
    MODEL_SAVE_PATH = MODEL_PATH / MODEL_NAME

    print(f"Saving {model.name} Parameters in: {MODEL_SAVE_PATH}")
    torch.save(obj=model.state_dict(), f=MODEL_SAVE_PATH)

    return MODEL_SAVE_PATH

# Save Pre-Trained Models-bank
states_path = []  # Path for each model parameters state
for i in range(len(ocon_vowels)):
    state_path = save_model_state(ocon_vowels[i])
    states_path.append(state_path)
print()
```

Output **Maxnet Algorithm**

```
# OCON "MaxNet" Architecture (Weightening + Non Linearity apply)
class OCON_MaxNet(nn.Module):                                              # nn.Module: base class to inherit from
    def __init__(self, n_units, act_fun, eps):                                     # self + attributes (architecture hyper-paramet
        super().__init__()

        self.layers = nn.ModuleDict()                                      # Dictionary to store Model layers
        self.eps_weight = eps

        # MaxNet Layer
        self.layers['MAXNET'] = nn.Linear(n_units, n_units)                # Key 'MaxNet' layer specification

        # Weights & Bias initialization
        self.layers['MAXNET'].weight.data.fill_(self.eps_weight)
        for i in range(n_units):
            self.layers['MAXNET'].weight[i][i].data.fill_(1.)  # Self Weight = 1

        self.layers['MAXNET'].bias.data.fill_(0.)

        # Activation Function
        self.actfun = act_fun  # Function string-name attribute association

    # Forward Pass Method
```

```
    def forward(self, x):

        # Activation function object computation
        actfun = getattr(torch.nn, self.actfun)

        # Maxnet Layer pass                                              --> Output Weightening (Dot Product) "Linear transfc
        x = actfun()(self.layers['MAXNET'](x.squeeze().float()))

        # Self

        return x


# Build OCON MaxNetwork Architecture
torch.manual_seed(SEED)
ocon_maxnet = OCON_MaxNet(n_units=12, act_fun='ReLU', eps=0.25)

# MaxNet & Sub-Networks Parameters
print('OCON MaxNet STATE')
model_desc(ocon_maxnet)


def maxnet_algo(maxnet_function, n_units, act_fun, eps, input_array):
    """
    MaxNet Re-iteration algorithm for Maximum Value retrieving from an input array
    """
    non_zero_outs = np.count_nonzero(input_array)  # Non Zero Values initialization
    maxnet_in = torch.from_numpy(input_array)  # MaxNet Input Tensor initialization

    results = []  # Results initialization

    counter = 0
    while non_zero_outs != 1:
        counter += 1

        # Create the MaxNet
        torch.manual_seed(SEED)  # Redundant
        maxnet = maxnet_function(n_units = n_units, act_fun = act_fun, eps = eps)

        # Compute Forward Pass
        results = maxnet(maxnet_in)

        # Non_zero outputs & Maxnet Input Update
        non_zero_outs = np.count_nonzero(results.detach().numpy())
        maxnet_in = results.detach()  # Save results for next iteration

    print(f'Maximum Value found in {counter} iterations')
    return np.argmax(results.detach().numpy())


# MaxNet on Sub-Networks predictions
ocon_predictions_prob = np.zeros((len(ocon_predictions), x_data_minmax.shape[0]))  # NumPy predictions matrix (12 * 1617)

# Convert from List of Tensors to 2D NumPy Array
for i in range(len(ocon_predictions)):
    ocon_predictions_prob[i, :] = ocon_predictions[i].detach().squeeze().numpy()

maxnet_class_predictions = []  # Classes Outputs list initialization
# MaxNet application
for i in range(x_data_minmax.shape[0]):
    print(f'Dataset Sample({i + 1}) Class Evaluation')

    samp_predictions = ocon_predictions_prob[:, i]  # Array of 12 predictions for each Dataset sample (OCON outputs)
    class_prediction = maxnet_algo(OCON_MaxNet, n_units=12, act_fun='ReLU', eps=-0.1, input_array=samp_predictions)  # MaxNet
    maxnet_class_predictions.append(class_prediction)  # Result appending
    print('-----------------------------------------------')

maxnet_accuracy = 100 * np.mean((np.array(maxnet_class_predictions).reshape(1597, 1) == y_labels_raw_np))  # Accuracy computat
print(f'Maxnet Output --> Phoneme ACCURACY: {maxnet_accuracy}%')


# Argmax on Sub-Networks predictions (...for multiple 1s probabilities MaxNet infinite loops)
#ocon_predictions_prob = np.zeros((len(new_ocon_predictions), x_data_minmax.shape[0]))  # NumPy predictions matrix (12 * 1617)
#
# Convert from List of Tensors to 2D NumPy Array
#for i in range(len(new_ocon_predictions)):
#    ocon_predictions_prob[i, :] = new_ocon_predictions[i].detach().squeeze().numpy()
#
#maxnet_class_predictions = np.argmax(ocon_predictions_prob, axis=0)
#maxnet_accuracy = 100 * np.mean((np.array(maxnet_class_predictions).reshape(1617, 1) == y_labels_raw_np))  # Accuracy computa
#print(f'Maxnet Output ACCURACY: {maxnet_accuracy}%')
```

```
# Evaluation Analysis Plot
plt.figure(figsize=(30, 5))
plt.suptitle(f'OCON Bank + MaxNet Evaluation: {maxnet_accuracy:.0f}%')

plot_x_ticks = end_idx[:]
plot_x_ticks = np.delete(plot_x_ticks, len(end_idx) - 1)
plot_y_ticks = [n for n in range(len(vowels))]

plt.plot(y_labels_raw_np, 'rs', label='Ground Truths')
plt.plot(maxnet_class_predictions, 'k.', label='MAXNET Outputs')
plt.xlabel('Dataset samples')
plt.xticks(ticks=plot_x_ticks, labels=vowels)
plt.xlim([-10, len(y_labels_raw_np) + 10])
plt.ylabel('Labels')
plt.yticks(ticks=plot_y_ticks, labels=vowels)
plt.legend(loc='best')
plt.grid()

plt.tight_layout()
plt.savefig('OCON_model_evaluation')
plt.show()
```

OCON Model Analysis

(13-features Complete Dataset - Speaker Recognition)

Author: S. Giacomelli

Year: 2023

Affiliation: A.Casella Conservatory (student)

Master Degree Thesis: "*Vowel phonemes Analysis & Classification by means of OCON rectifiers Deep Learning Architectures*"

Description: Python scripts for One-Class-One-Network (OCON) Model analysis and optimization

```
# Numerical computations packages/modules
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

# Dataset processing modules
from torch.utils.data import DataLoader
from sklearn.model_selection import train_test_split

# Graphic visualization modules
import matplotlib.pyplot as plt
import matplotlib_inline
matplotlib_inline.backend_inline.set_matplotlib_formats('svg')

# Common Seed initialization
SEED = 42  # ... the answer to the ultimate question of Life, the Universe, and Everything... (cit.)

# PyTorch Processing Units evaluation
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(f'AVILABLE Processing Unit: {device.upper()}')
!nvidia-smi
```

HGCW Dataset

- *Dataset_utils.npz* file read
- *One-Hot encoding* definition
- *Train/Dev/Test split* definition

```
# Load HGCW 13_features_complete_(w_speaker) Dataset
HGCW_dataset_utils = np.load(file='./HGCW_dataset_utils.npz')
print('Raw features                    Data shape:', HGCW_dataset_utils['HGCW_raw'].shape)
print('Fundamental Normalized features Data shape:', HGCW_dataset_utils['HGCW_fund_norm'].shape)
print('MinMax features                 Data shape:', HGCW_dataset_utils['HGCW_minmax'].shape)
print('Phoneme Labels                  Data shape:', HGCW_dataset_utils['HGCW_phon_labels'].shape)
print('Speaker Labels                  Data shape:', HGCW_dataset_utils['HGCW_spk_labels'].shape)
print('Phoneme Classes size            Data shape:', HGCW_dataset_utils['phon_size'].shape)
print('Phoneme Classes indices         Data shape:', HGCW_dataset_utils['phon_idx'].shape)
print('Phoneme-Speaker coordinates     Data shape:', HGCW_dataset_utils['phon_spk_coords'].shape)  # (Start_Idx, Vow-Spk group

x_data_raw_np = HGCW_dataset_utils['HGCW_raw']
x_data_fund_norm = HGCW_dataset_utils['HGCW_fund_norm']
x_data_minmax = HGCW_dataset_utils['HGCW_minmax']
y_labels_raw_np = HGCW_dataset_utils['HGCW_phon_labels']
z_labels_raw_np = HGCW_dataset_utils['HGCW_spk_labels']
vow_size = HGCW_dataset_utils['phon_size']
end_idx = HGCW_dataset_utils['phon_idx']
phon_spk_coords = HGCW_dataset_utils['phon_spk_coords']

# Auxiliary lists
vowels = ['ae', 'ah', 'aw', 'eh', 'er', 'ei', 'ih', 'iy', 'oa', 'oo', 'uh', 'uw']  # Vowels list (0 - 11)
speakers = ['b', 'g', 'm', 'w']  # Speakers list (0 - 3)
colors = ['green', 'blue', 'red']  # ['red', 'saddlebrown', 'darkorange', 'darkgoldenrod', 'gold', 'darkkhaki', 'olive', 'dark


# Fundamental Frequency - Min-Max Scaling
print(f'Previous Fundamental Frequency Min: {x_data_minmax[:, 0].min()}, Max: {x_data_minmax[:, 0].max()}')
x_data_minmax[:, 0] = (x_data_minmax[:, 0] - x_data_minmax[:, 0].min()) / (x_data_minmax[:, 0].max() - x_data_minmax[:, 0].mir
print(f'Actual Fundamental Frequency Min: {x_data_minmax[:, 0].min()}, Max: {x_data_minmax[:, 0].max()}')
```

```
# Children Labels encoding
z_labels_raw_np_alt = np.where(z_labels_raw_np <= 1, 0, z_labels_raw_np - 1)
speakers_alt = ['c', 'm', 'w']


def one_hot_encoder(sel_speaker_num=0, dataset=x_data_minmax, spk_labels=z_labels_raw_np_alt, speakers=speakers_alt, debug=Fal

    if sel_speaker_num < len(speakers):
        classes = [n for n in range(len(speakers))]  # Speaker indices
        sub_groups_size = []  # Same as vow_size list

        sub_data_one = dataset[np.where(spk_labels == sel_speaker_num)[0], :]  # Extract selected speaker sub-dataset
        sub_labels_one = np.ones((sub_data_one.shape[0], 1), dtype='int')  # Binarized 1-Label creation
        sub_labels_one_orig = np.ones((sub_data_one.shape[0], 1), dtype='int') * sel_speaker_num  # Create a copy of original
        sub_groups_size.append(sub_data_one.shape[0])
        if debug is True:
            print(f'Selected Class "{speakers[sel_speaker_num]}"-speaker : {sub_data_one.shape[0]} samples')

        sub_speakers_size = sub_data_one.shape[0] // 2  # Size for each other speaker sub-group (balancing)
        if debug is True:
            print(f'Rest Classes size (...each): {sub_speakers_size} samples')

        sub_data_zero = np.zeros((sub_speakers_size * 2, sub_data_one.shape[1]))  # Zero-label features dataset initialization
        sub_labels_zero_orig = np.zeros((sub_speakers_size * 2, 1), dtype='int')  # Original labels subset array initializatio

        classes.remove(sel_speaker_num)  # Remove selected speaker index
        counter = 0
        for i in classes:  # For other speakers...
            sub_data_zero_class =  dataset[np.where(spk_labels == i)[0], :]  # Extract speaker subgroup
            subset_indices = np.random.choice(np.arange(0, sub_data_zero_class.shape[0], 1), size=sub_speakers_size, replace=F

            sub_data_zero[counter * sub_speakers_size : (counter * sub_speakers_size) + sub_speakers_size, :] = sub_data_zero_
            sub_labels_zero_orig[counter * sub_speakers_size : (counter * sub_speakers_size) + sub_speakers_size, :] = np.ones
            sub_groups_size.append(sub_data_zero_class[subset_indices].shape[0])

            counter += 1

        sub_labels_zero = np.zeros((sub_data_zero.shape[0], 1), dtype='int')  # Binarized 0 Label creation

        # Output Matrices
        sub_data = np.vstack((sub_data_one, sub_data_zero))  # Vertical stacking 1s and 0s features array
        sub_data_labels_bin = np.vstack((sub_labels_one, sub_labels_zero))  # Vertical Stacking 1s and 0s labels array
        sub_data_labels_orig = np.vstack((sub_labels_one_orig, sub_labels_zero_orig))  # Vertical stacking original (1-class)
    else:
        raise ValueError(f'Invalid Class ID: "{sel_speaker_num}" --> It must be less than {len(speakers)}!')

    return sub_data, sub_data_labels_bin, sub_data_labels_orig, sub_groups_size

# Test Call
sel_speaker_num = 0
sub_data, sub_data_labels_bin, sub_data_labels_orig, sub_groups_size = one_hot_encoder(sel_speaker_num, dataset=x_data_minmax,
print(f'Output Array shapes: {sub_data.shape}, {sub_data_labels_bin.shape}, {sub_data_labels_orig.shape}, {len(sub_groups_size


# Sub-Dataset Plot (previous example)
classes = [n for n in range(len(speakers_alt))]

plt.figure(figsize=(12, 15))
plt.suptitle(f'Sub-Dataset ("{speakers_alt[sel_speaker_num].upper()}"-speaker - example) One-Hot Encoding')

counter = 0
for index in range(len(speakers_alt)):
    if index == sel_speaker_num:
        first_coords = sub_data[:sub_groups_size[0], 1]
        second_coords = sub_data[:sub_groups_size[0], 2]
        third_coords = sub_data[:sub_groups_size[0], 3]
    else:
        start_index = sub_groups_size[0] + counter * sub_groups_size[1]
        end_index = start_index + sub_groups_size[1]

        first_coords = sub_data[start_index: end_index, 1]
        second_coords = sub_data[start_index: end_index, 2]
        third_coords = sub_data[start_index: end_index, 3]

        counter += 1

    plt.subplot(3, 2, 1)
    plt.title('$1_{st}$ VS $2_{nd}$')
    plt.scatter(first_coords, second_coords, marker='o', color=colors[index], label=f'"{speakers_alt[index]}"-speaker')
    plt.xlabel('$1_{st}$ Formant Ratio')
    plt.ylabel('$2_{nd}$ Formant Ratio')
    plt.legend(loc='best')
    plt.grid(True)
```

```
    plt.subplot(3, 2, 3)
    plt.title('$1_{st}$ VS $3_{rd}$')
    plt.scatter(first_coords, third_coords, marker='o', color=colors[index], label=f'"{speakers_alt[index]}"-speaker')
    plt.xlabel('$1_{st}$ Formant Ratio')
    plt.ylabel('$3_{rd}$ Formant Ratio')
    plt.legend(loc='best')
    plt.grid(True)

    plt.subplot(3, 2, 5)
    plt.title('$2_{nd}$ VS $3_{rd}$')
    plt.scatter(second_coords, third_coords, marker='o', color=colors[index], label=f'"{speakers_alt[index]}"-speaker')
    plt.xlabel('$2_{nd}$ Formant Ratio')
    plt.ylabel('$3_{rd}$ Formant Ratio')
    plt.legend(loc='best')
    plt.grid(True)

plt.subplot(3, 2, 2)
plt.title('$1_{st}$ VS $2_{nd}$ Binarized')
plt.scatter(sub_data[0: sub_groups_size[0], 1], sub_data[0: sub_groups_size[0], 2], color=colors[sel_speaker_num], label=f'"{s
plt.scatter(sub_data[sub_groups_size[0]: , 1], sub_data[sub_groups_size[0]: , 2], color='grey', label=f'Rest')
plt.xlabel('$1_{st}$ Formant Ratio')
plt.ylabel('$2_{nd}$ Formant Ratio')
plt.legend(loc='best')
plt.grid(True)

plt.subplot(3, 2, 4)
plt.title('$1_{st}$ VS $3_{rd}$ Binarized')
plt.scatter(sub_data[0: sub_groups_size[0], 1], sub_data[0: sub_groups_size[0], 3], color=colors[sel_speaker_num], label=f'"{s
plt.scatter(sub_data[sub_groups_size[0]: , 1], sub_data[sub_groups_size[0]: , 3], color='grey', label=f'Rest')
plt.xlabel('$1_{st}$ Formant Ratio')
plt.ylabel('$3_{rd}$ Formant Ratio')
plt.legend(loc='best')
plt.grid(True)

plt.subplot(3, 2, 6)
plt.title('$2_{nd}$ VS $3_{rd}$ Binarized')
plt.scatter(sub_data[0: sub_groups_size[0], 2], sub_data[0: sub_groups_size[0], 3], color=colors[sel_speaker_num], label=f'"{s
plt.scatter(sub_data[sub_groups_size[0]: , 2], sub_data[sub_groups_size[0]: , 3], color='grey', label=f'Rest')
plt.xlabel('$2_{nd}$ Formant Ratio')
plt.ylabel('$3_{rd}$ Formant Ratio')
plt.legend(loc='best')
plt.grid(True)

plt.tight_layout()
plt.savefig(f'{speakers_alt[sel_speaker_num]}_speaker_one_hot_encoding')
plt.show()


# Train/Test split (auxiliary function)
def train_test_split_aux(features_dataset, labels_dataset, test_perc, tolerance):
    """
    An auxiliary Train_Test_split function (based on Scikit Learn implementation) w. balance tolerance specification
    """
    test_size = int(test_perc / 100 * len(features_dataset))
    train_balance = 0  # Output Training set balance value initialization
    test_balance = 0  # Output Testing set balance value initialization

    min_tol = np.mean(labels_dataset) - tolerance
    max_tol = np.mean(labels_dataset) + tolerance
    print(f'Data Balancing  (TARGET = {np.mean(labels_dataset)} +- {tolerance}): ', end='')

    while (min_tol >= train_balance or train_balance >= max_tol) or (min_tol >= test_balance or test_balance >= max_tol):
        train_data, test_data, train_labels, test_labels = train_test_split(features_dataset, labels_dataset, test_size=test_s
        train_balance = np.mean(train_labels)
        test_balance = np.mean(test_labels)
        print('.', end='')
    else:
        print('OK')

    return train_data, test_data, train_labels, test_labels, train_balance, test_balance

# Train-Dev-Test split function
def train_dev_test_split(x_data, y_labels, split_list, tolerance=0.1, output='Loaders', debug=False):
    """
    Compute a Train, Development (Hold-Out) and a Test set split w. PyTorch Dataset conversion (and eventual Loaders initializ
    """
    if len(split_list) == 3:
        # Train - Dev+Test separation
        print('Training --- Devel/Test SPLIT')
        train_data, testTMP_data, train_labels, testTMP_labels, _, _ = train_test_split_aux(x_data, y_labels, (split_list[1] *
        print('----------------------------------')
```

```
        # Dev - Test separation
        print('Devel    ---     Test SPLIT')

        split = ((split_list[1] * 100) / np.sum(split_list[1:] * 100)) * 100  # Split in %
        dev_data, test_data, dev_labels, test_labels, _, _ = train_test_split_aux(testTMP_data, testTMP_labels, split, toleran
        print('----------------------------------')

        # Tensor Conversion
        train_data_tensor = torch.tensor(train_data).float()
        train_labels_tensor = torch.tensor(train_labels, dtype=torch.int64).squeeze()
        dev_data_tensor = torch.tensor(dev_data).float()
        dev_labels_tensor = torch.tensor(dev_labels, dtype=torch.int64).squeeze()
        test_data_tensor = torch.tensor(test_data).float()
        test_labels_tensor = torch.tensor(test_labels, dtype=torch.int64).squeeze()
        if debug is True:
            print(f'Training Data        Shape: {train_data.shape}')
            print(f'Development Data     Shape: {dev_data.shape}')
            print(f'Testing Data         Shape: {test_data.shape}')

            # Balance Evaluation
            print(f'Training Set       Balance: {np.mean(train_labels)}')
            print(f'Development Set    Balance: {np.mean(dev_labels)}')
            print(f'Testing Set        Balance: {np.mean(test_labels)}')

        if output != 'Loaders':
            return train_data_tensor, train_labels_tensor, dev_data_tensor, dev_labels_tensor, test_data_tensor, test_labels_t
        else:
            # PyTorch Dataset Conversion
            train_dataset = torch.utils.data.TensorDataset(torch.tensor(train_data).float(), torch.tensor(train_labels, dtype=
            dev_dataset = torch.utils.data.TensorDataset(torch.tensor(dev_data).float(), torch.tensor(dev_labels, dtype=torch.
            test_dataset = torch.utils.data.TensorDataset(torch.tensor(test_data).float(), torch.tensor(test_labels, dtype=tor

            # DataLoader (Batches) --> Drop-Last control to optimize training
            trainLoader = DataLoader(train_dataset, shuffle=False, batch_size = 32, drop_last=True)
            devLoader = DataLoader(dev_dataset, shuffle=False, batch_size = dev_dataset.tensors[0].shape[0])
            testLoader = DataLoader(test_dataset, shuffle=False, batch_size = test_dataset.tensors[0].shape[0])
            if debug is True:
                print(f'Training Set    Batch Size: {trainLoader.batch_size}')
                print(f'Development Set Batch Size: {devLoader.batch_size}')
                print(f'Testing Set     Batch Size: {testLoader.batch_size}')

            return trainLoader, devLoader, testLoader
    else:
        # Train - Test separation
        print('Training --- Test    SPLIT')
        train_data, test_data, train_labels, test_labels, _, _ = train_test_split_aux(x_data, y_labels, split_list[1] * 100, t
        print('--------------------------')

        # Tensor Conversion
        train_data_tensor = torch.tensor(train_data).float()
        train_labels_tensor = torch.tensor(train_labels, dtype=torch.int64).squeeze()
        test_data_tensor = torch.tensor(test_data).float()
        test_labels_tensor = torch.tensor(test_labels, dtype=torch.int64).squeeze()
        if debug is True:
            print(f'Training Data        Shape: {train_data.shape}')
            print(f'Testing Data         Shape: {test_data.shape}')

            # Balance Evaluation
            print(f'Training Set    Balance: {np.mean(train_labels)}')
            print(f'Testing Set     Balance: {np.mean(test_labels)}')

        if output != 'Loaders':
            return train_data_tensor, train_labels_tensor, test_data_tensor, test_labels_tensor
        else:
            # PyTorch Dataset Conversion
            train_dataset = torch.utils.data.TensorDataset(torch.tensor(train_data).float(), torch.tensor(train_labels, dtype=
            test_dataset = torch.utils.data.TensorDataset(torch.tensor(test_data).float(), torch.tensor(test_labels, dtype=tor

            # DataLoader (Batches) --> Drop-Last control to optimize training
            trainLoader = DataLoader(train_dataset, shuffle=False, batch_size = 32, drop_last=True)
            testLoader = DataLoader(test_dataset, shuffle=False, batch_size = test_dataset.tensors[0].shape[0])
            if debug is True:
                print(f'Training Set    Batch Size: {trainLoader.batch_size}')
                print(f'Testing Set     Batch Size: {testLoader.batch_size}')

            return trainLoader, testLoader
```

One-Class Architecture (Binary Classifier)

(see "*One-Class_Sub-Network_Analysis.ipynb*")

```
Multi-Layer Perceptron
- Input Layer: 3 features [formant ratios, min-max normalized]
- Hidden Layer: 100 units
- Output Layer: 1 normalized probability
- Learning Rate: 0.0001 (10^-4)
- Optimizer: Adam (Adaptive Momentum)

- Mini-Batch Training:
    . Re-iterated Sub-Dataset Shuffling
    . Early Stopping (Test Accuracy driven)
    . Batch size = 32

- Regularization:
    . Weight Decay (L2 Penalty): 0.0001 (10^-4)
    . DropOut:
        * Input Layer Drop Rate: 0.8
        * Hidden Layer Drop Rate: 0.5.
    . Batch Normalization
```

- *MLP Classifier Architecture* class definition
- *Mini-Batch Training* function definition

```
# Dynamic Multi-Layer Architecture Class (w. units, activation function, batch normalization and dropOut rate specification)
class binaryClassifier(nn.Module):                                               # nn.Module: base class to inherit from
    def __init__(self, n_units, act_fun, rate_in, rate_hidden, model_name):      # self + attributes (architecture hyper-para
        super().__init__()

        self.layers = nn.ModuleDict()                                            # Dictionary to store Model layers
        self.name = model_name

        # Input Layer
        self.layers['input'] = nn.Linear(13, n_units)                             # Key 'input' layer specification

        # Hidden Layer
        self.layers[f'hidden'] = nn.Linear(n_units, n_units)
        self.layers[f'batch_norm'] = nn.BatchNorm1d(n_units)

        # Output Layer
        self.layers['output'] = nn.Linear(n_units, 1)                            # Key 'output' layer specification

        # Activation Function
        self.actfun = act_fun                                                    # Function string-name attribute association

        # Dropout Parameter
        self.dr_in = rate_in
        self.dr_hidden = rate_hidden

        # Weights & Bias initialization
        for layer in self.layers.keys():
            try:
                nn.init.kaiming_normal_(self.layers[layer].weight, mode='fan_in') # Kaiming He - Normal Distributed (ReLU spec
            except:
                pass                                                             # Batch_norm Layer can't be initialized
            self.layers[layer].bias.data.fill_(0.)                               # Bias initialization (0.)

    # Forward Pass Method
    def forward(self, x):

        # Activation function object computation
        actfun = getattr(torch.nn, self.actfun)

        # Input Layer pass                                                       --> Weightening (Dot Product) "Linear transf
        x = actfun()(self.layers['input'](x))
        x = F.dropout(x, p=self.dr_in, training=self.training)                   # Activate DropOut only when Model Training

        # Single Hidden Layer pass                                               --> Weightening (Dot Product) "Linear transf
        x = self.layers[f'batch_norm'](x)                                        # Apply batch normalization before hidden la
        x = actfun()(self.layers[f'hidden'](x))
        x = F.dropout(x, p=self.dr_hidden, training=self.training)               # Same as "Input pass"

        # Output Layer pass                                                      --> Output Weightening (Dot Product) "Linear
        x = self.layers['output'](x)
        x = nn.Sigmoid()(x)

        return x
```

```
# Batch Training function (w. Adam Optimizer & L2 penalty term) Re-Definition
def mini_batch_train_test(model, weight_decay, epochs: int, learning_rate, train_loader, dev_loader, test_loader, debug=False)
    """
    Train & Test an ANN Architecture via Mini-Batch Training (w. Train/Dev/Test PyTorch Loaders) and Adam Backpropagation Opti
    """
    # Loss Function initialization
    loss_function = nn.BCELoss()

    # Optimizer Algorithm initialization
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=weight_decay)

    # Output list initialization
    train_accuracies = []
    train_losses = []
    dev_accuracies = []

    # TRAINING Phase
    for epoch in range(epochs):
        model.train()  # TRAINING Switch ON

        batch_accuracies = []
        batch_losses = []

        # Training BATCHES Loop
        for data_batch, labels_batch in train_loader:
            train_predictions = model(data_batch)
            train_loss = loss_function(train_predictions.squeeze(), labels_batch.type(torch.int64).float())
            batch_losses.append(train_loss.detach())

            # Backpropagation
            optimizer.zero_grad()
            train_loss.backward()
            optimizer.step()

            # Accuracy
            train_accuracy = 100 * torch.mean(((train_predictions.squeeze() > 0.5) == labels_batch.type(torch.int64).squeeze()

            # Batch Stats appending
            batch_accuracies.append(train_accuracy.detach())
            batch_losses.append(train_loss.detach())

        # Training Stats appending
        train_accuracies.append(np.mean(batch_accuracies))  # Average of Batch Accuracies = Training step accuracy
        train_losses.append(np.mean(batch_losses))  # Average of Batch Losses = Training step Losses

        # EVALUATION (Dev) Phase
        model.eval()
        with torch.no_grad():
            dev_data_batch, dev_labels_batch = next(iter(dev_loader))
            dev_predictions = model(dev_data_batch)

            dev_accuracy = 100 * torch.mean(((dev_predictions.squeeze() > 0.5) == dev_labels_batch.type(torch.int64).squeeze()

            if debug is True:
                if epoch % 100 == 0:
                    print(f'Epoch {epoch} --> DEV ACCURACY: {dev_accuracy.detach():.3f} %')
                    print('--------------------------------')

            # Evaluation accuracy appending
            dev_accuracies.append(dev_accuracy.detach())

    # TEST Phase
    model.eval()
    with torch.no_grad():
        test_data_batch, test_labels_batch = next(iter(test_loader))
        test_predictions = model(test_data_batch)
        test_accuracy = 100 * torch.mean(((test_predictions.squeeze() > 0.5) == test_labels_batch.type(torch.int64).squeeze())

        if debug is True:
            print(f'TEST ACCURACY: {test_accuracy.detach():.2f} %')
            print('--------------------------------------------------------------------')

    return train_accuracies, train_losses, dev_accuracies, test_accuracy.detach()
```

OCON (One-Class-One-Net) Model

Binary classifiers **Parallelization**

- *Classifiers-Bank* function definition
    - *Models Parameters* inspection

- Classifiers **Sequential** Training & Evaluation
- *Models Parameters State Save/Load* function definition

- MaxNet output algorithm
- Argmax output algorithm

```
def OCON_bank(one_class_function, hidden_units, act_fun, dr_in, dr_hidden, classes_list):
    """
    Create a One-Class-One-Network parallelization bank of an input Sub-Network definition
    """
    # Sub-Net names creation
    models_name_list = []
    for i in range(len(classes_list)):
        models_name_list.append("{}_{}".format(classes_list[i], "subnet"))  # Class name + _subnet

    # Sub-Networks instances creation
    sub_nets = []  # Sub Network list initialization

    for i in range(len(models_name_list)):

        torch.manual_seed(SEED)  # Seed re-initialization

        # Sub-Net instance creation
        locals()[models_name_list[i]] = one_class_function(hidden_units, act_fun, dr_in, dr_hidden, models_name_list[i])
        sub_nets.append(locals()[models_name_list[i]])

    return sub_nets


# Load Architecture Parameters State function
def load_model_state(model, state_dict_path):
    """
    Load an existent State Dictionary in a defined model
    """

    model.load_state_dict(torch.load(state_dict_path))
    print(f'Loaded Parameters (from "{state_dict_path}") into: {model.name}')

    return model


# Build The OCON Model
ocon_speakers = OCON_bank(binaryClassifier, 100, 'ReLU', 0.8, 0.5, speakers_alt)  # Best MLP (see "One-Class_Binary_Classifier

# Load Pre-Trained Architectures in a fresh Model instance:
#ocon_vowels = OCON_bank(binaryClassifier, 100, 'ReLU', 0.8, 0.5, speaker)  # Best MLP (see "One-Class_Binary_Classifier_Analy
#states_path = ["Trained_models_state/b_subnet_Params.pth",
#               "Trained_models_state/g_subnet_Params.pth",
#               "Trained_models_state/m_subnet_Params.pth",
#               "Trained_models_state/w_subnet_Params.pth"]
#
#for i in range(len(ocon_speakers)):
#    load_model_state(ocon_speakers[i], states_path[i])


# OCON Evaluation function
def OCON_eval(ocon_models_bank, features_dataset: np.ndarray = x_data_minmax, labels: np.ndarray = z_labels_raw_np_alt):
    """
    Evaluate OCON models-bank over an entire dataset
    """
    # Output lists initialization
    predictions = []
    dist_errors = []
    eval_accuracies = []
    g_truths = []  # For plotting purpouses

    # Evaluate each Sub-Network...
    for i in range(len(ocon_models_bank)):
        ocon_models_bank[i].eval()  # Put j-esimal Sub-Network in Evaluation Mode
        print(f'{ocon_models_bank[i].name.upper()} Evaluation -', end=' ')

        with torch.no_grad():

            # Make predictions
            features_data_tensor = torch.tensor(features_dataset).float()
            raw_eval_predictions = ocon_models_bank[i](features_data_tensor)

            # Create Ground Truths
            ground_truth = np.where(labels == i, 1, 0)
            ground_truth_tensor = torch.tensor(ground_truth, dtype=torch.int64).squeeze()

            # Compute Errors
```

```
            dist_error = ground_truth_tensor - raw_eval_predictions.detach().squeeze()  # Distances
            eval_accuracy = 100 * torch.mean(((raw_eval_predictions.detach().squeeze() > 0.5) == ground_truth_tensor).float())
            print(f'Accuracy: {eval_accuracy:.2f}%')

        # Outputs append
        predictions.append(raw_eval_predictions.detach())
        dist_errors.append(dist_error.detach())
        eval_accuracies.append(eval_accuracy.detach())

        g_truths.append(ground_truth)

    return predictions, dist_errors, eval_accuracies, g_truths

# For Pre-Trained Models
#ocon_predictions, ocon_dist_errors, ocon_eval_accuracies, ocon_g_truths = OCON_eval(ocon_speakers)


# Model Parameters State function
def model_desc(model):
    """
    Print a Console report of Neural Network Model parameters
    """
    # Parameters Description
    print('Params Description:')
    trainable_params = 0

    for parameter in model.named_parameters():
        print(f'Parameter Name      : {parameter[0]}')
        print(f'Parameter Weights   : {parameter[1][:]}')
        if parameter[1].requires_grad:
            print(f'...with {parameter[1].numel()} TRAINABLE parameters')
            trainable_params += parameter[1].numel()

        print('................................')

    print('-----------------------------------------------------------------')

    # Nodes Count
    nodes = 0
    for param_name, param_tensor in model.named_parameters():
        if 'bias' in param_name:
            nodes += len(param_tensor)

    print(f'Total Nodes             : {nodes}')
    print('-----------------------------------------------------------------')

# OCON-Model Description
for i in range(len(ocon_speakers)):
    print(f'OCON "{ocon_speakers[i].name}" Classifier STATE')
    model_desc(ocon_speakers[i])
    print()


# Training/Eval/Testing Parameters
epochs = 1000  # For each "Data Batch-Set"
loss_breaks = [0.36, 0.08, 0.45]  # loss (for Early Stopping) --> class-specific (empyrical)
acc_breaks = [80., 97., 80.]  # % accuracy (for Early Stopping)  --> class-specific (empyrical)
min_tolerance = 0.01 # ...for sub-dataset balancing

# Outputs Initialization
loss_functions = [[] for _ in range(len(ocon_speakers))]
training_accuracies = [[] for _ in range(len(ocon_speakers))]
evaluation_accuracies = [[] for _ in range(len(ocon_speakers))]
test_accuracies = [[] for _ in range(len(ocon_speakers))]
training_times = []


# OCON Sub-Networks Training
from time import perf_counter
debug = False

for i, speaker in enumerate(speakers_alt):

    # Class-specific Early Stopping parameters
    loss_break = loss_breaks[i]
    acc_break = acc_breaks[i]

    print(f'Architecture "{ocon_speakers[i].name}" TRAINING PHASE')
    print(f'EARLY STOP THRESHOLD: Loss={loss_break}, Accuracy={acc_break}%')

    start_timer = perf_counter()
    # Iterated (w. Batch-Sets shuffling) Mini-Batch Training
    iteration = 0  # Batch Training iteration counter
```

```python
    mean_loss = 1.
    test_accuracy = 0.

    while (mean_loss > loss_break) or (test_accuracy < acc_break):
        # Dataset processing
        sub_data, sub_data_labels_bin, _, _ = one_hot_encoder(sel_speaker_num=i, dataset=x_data_minmax, debug=debug)
        print('---------------------------------')
        trainLoader, devLoader, testLoader = train_dev_test_split(sub_data, sub_data_labels_bin, [0.5, 0.25, 0.25], tolerance=

        # Train/Test Architecture
        train_accuracies, train_losses, dev_accuracies, test_accuracy = mini_batch_train_test(ocon_speakers[i], weight_decay=0
        print(f'Sub-Net "{speaker.upper()}" Epoch {(iteration + 1) * epochs} - TEST ACCURACY: {test_accuracy:.2f}%', end=' ')

        # Outputs append
        loss_functions[i].append(train_losses)
        training_accuracies[i].append(train_accuracies)
        evaluation_accuracies[i].append(dev_accuracies)
        test_accuracies[i].append(test_accuracy)

        # Repeating condition evaluation
        mean_loss = np.mean(train_losses[-50: ])  # Last 50 losses mean
        print(f'- MEAN LOSS: {mean_loss}')

        iteration += 1  # Go to next Batch training iteration

    print(f'Training STOPPED at iteration {iteration}')
    print('---------------------------------------------------------------------')
    stop_timer = perf_counter()

    print(f'"{ocon_speakers[i].name}" Training COMPLETED in {float(stop_timer - start_timer)}sec.')
    training_times.append(stop_timer - start_timer)
    print('---------------------------------------------------------------------')


# Graphical smoothing filter
def smooth(data, k=100):
    """
    A Convolution LP filter w. interval definition
    """
    return np.convolve(data, np.ones(k) / k, mode='same')


# Training Phase Plots
plt.figure(figsize=(12, 5 * 3))

# loss_functions, training_accuracies, evaluation_accuracies, test_accuracies, training_times
classes = len(ocon_speakers)

for i in range(classes):
    plt.subplot(classes, 2, (i * 2) + 1)
    flat_loss_function = [item for sublist in loss_functions[i] for item in sublist]

    plt.plot(smooth(flat_loss_function), 'k-')
    plt.axhline(loss_breaks[i], color='r', linestyle='--')
    plt.title(f'{ocon_speakers[i].name.upper()} Training Loss')
    plt.xlabel('Epochs')
    plt.xlim([100, len(flat_loss_function) - 100])
    plt.ylabel('GT - Predicted diff. (probability)')
    plt.grid()

    plt.subplot(classes, 2, (i * 2) + 2)
    flat_training_accuracy = [item for sublist in training_accuracies[i] for item in sublist]
    flat_dev_accuracy = [item for sublist in evaluation_accuracies[i] for item in sublist]
    flat_test_accuracy = test_accuracies[i]

    plt.plot(smooth(flat_training_accuracy), 'k-', label='Training')
    plt.plot(smooth(flat_dev_accuracy), color='grey', label='Development')
    if len(flat_test_accuracy) > 1:
        plt.plot([(n + 1) * epochs for n in range(len(flat_test_accuracy))], flat_test_accuracy, 'r-', label=f'Test')
    else:
        plt.axhline(test_accuracy, color='r', linestyle='-', label=f'Test')
    plt.title(f'{ocon_speakers[i].name.upper()} Accuracy (after {training_times[i]:.2f}sec.)')
    plt.xlabel('Epochs')
    plt.xlim([100, len(flat_training_accuracy) - 100])
    plt.ylabel('Accuracy (in %)')
    plt.ylim([40, 101])
    plt.grid()
    plt.legend(loc='best')

plt.tight_layout()
plt.savefig('OCON_training_phase')
plt.show()
```

```
sub_groups_size


# OCON Evaluation

# Dataset & Labels Ordering (Plot conveniences)
x_data_minmax_ordered = np.zeros((1, x_data_minmax.shape[1]))
z_labels_raw_np_alt_ordered = np.zeros((1, 1), dtype='int')
data_size = []

# Groups Ordering iteration
for i in range(len(ocon_speakers)):
    indices = np.where(z_labels_raw_np_alt == i)[0]
    data_size.append(len(indices))

    x_data_minmax_ordered = np.vstack((x_data_minmax_ordered, x_data_minmax[indices]))
    z_labels_raw_np_alt_ordered = np.vstack((z_labels_raw_np_alt_ordered, z_labels_raw_np_alt[indices]))

x_data_minmax_ordered = np.delete(x_data_minmax_ordered, 0, axis=0)  # Remove 1st initialization null row
z_labels_raw_np_alt_ordered = np.delete(z_labels_raw_np_alt_ordered, 0, axis=0)  # Remove 1st initialization null label
print(f'Features Dataset Shapes : original {x_data_minmax.shape} VS ordered {x_data_minmax_ordered.shape}')
print(f'Labels Dataset Shapes   : original {z_labels_raw_np_alt.shape} VS ordered {z_labels_raw_np_alt_ordered.shape}')

ocon_predictions, ocon_dist_errors, ocon_eval_accuracies, ocon_g_truths = OCON_eval(ocon_speakers, features_dataset=x_data_min


# Dataset Evaluation Analysis Plot
plt.figure(figsize=(18, 5 * len(ocon_speakers)))
plot_ticks = []
for n in range(len(data_size)):
    plot_ticks.append(np.sum(data_size[: n], dtype='int'))

iter_idx = plot_ticks + [len(x_data_minmax_ordered)]

for i in range(len(ocon_speakers)):
    plt.subplot(len(ocon_speakers), 3, (i * 3) + 1)
    plt.plot(ocon_predictions[i], 'k.', label='Raw Predictions')
    plt.plot(ocon_g_truths[i], 'rx', label='Ground Truths')
    plt.axhline(0.5, linestyle='--', color='grey')
    plt.title(f'{ocon_speakers[i].name.upper()} Predictions Accuracy: {ocon_eval_accuracies[i]:.2f}%')
    plt.xlabel('Data (Indices)')
    plt.xticks(plot_ticks, speakers_alt)
    plt.ylabel('Normalized Probability')
    plt.grid()
    plt.legend(loc='best')

    plt.subplot(len(ocon_speakers), 3, (i * 3) + 2)
    plt.plot(ocon_dist_errors[i], 'k')
    plt.title(f'Predicted to Measured Error')
    plt.xlabel('Data (Indices)')
    plt.xticks(plot_ticks, speakers_alt)
    plt.ylabel('Normalized Probability Error')
    plt.ylim([-1.1, 1.1])
    plt.grid()

    plt.subplot(len(ocon_speakers), 3, (i * 3) + 3)

    # Predictions list processing
    predictions_temp = ocon_predictions[i]
    class_predictions = [item for sublist in predictions_temp for item in sublist]  # Turn a list of lists in a single list (c
    for j in range(len(class_predictions)):  # Turn a list of tensors of one variable in a list of scalars (item() method)
        class_predictions[j] = class_predictions[j].item()

    # Positives & False-Positives extraction
    positives = []
    for w in range(len(speakers_alt)):
        num = (np.array(class_predictions[iter_idx[w]: iter_idx[w + 1]]) > 0.5).sum()
        positives.append(num)

    plt.bar(np.arange(len(speakers_alt)), positives, color='k')
    plt.title(f'"{speakers_alt[i]}" Positive Probabilities Distribution')
    plt.xlabel('Normalized Probabilities')
    plt.ylabel('Occurences')
    plt.xticks([n for n in range(3)], speakers_alt)
    plt.grid()

plt.tight_layout()
plt.savefig('OCON_bank_evaluation')
plt.show()


# Model Parameters Save/Load functions
from pathlib import Path
```

```python
def save_model_state(model, folder_name: str = "Trained_models_state"):
    """
    Save Pre-Trained model parameters in a State Dictionary
    """

    MODEL_PATH = Path(folder_name)  # Placed in root
    MODEL_PATH.mkdir(parents=True, exist_ok=True)  # Pre-existing folder (w. same name) monitoring
    MODEL_NAME = '{}_{}'.format(model.name, "Params.pth")
    MODEL_SAVE_PATH = MODEL_PATH / MODEL_NAME

    print(f"Saving {model.name} Parameters in: {MODEL_SAVE_PATH}")
    torch.save(obj=model.state_dict(), f=MODEL_SAVE_PATH)

    return MODEL_SAVE_PATH

# Save Pre-Trained Models-bank
states_path = []  # Path for each model parameters state
for i in range(len(ocon_speakers)):
    state_path = save_model_state(ocon_speakers[i])
    states_path.append(state_path)
print()
```

Output **Maxnet Algorithm**

```python
# OCON "MaxNet" Architecture (Weightening + Non Linearity apply)
class OCON_MaxNet(nn.Module):                                            # nn.Module: base class to inherit from
    def __init__(self, n_units, act_fun, eps):                                # self + attributes (architecture hyper-paramet
        super().__init__()

        self.layers = nn.ModuleDict()                                    # Dictionary to store Model layers
        self.eps_weight = eps

        # MaxNet Layer
        self.layers['MAXNET'] = nn.Linear(n_units, n_units)              # Key 'MaxNet' layer specification

        # Weights & Bias initialization
        self.layers['MAXNET'].weight.data.fill_(self.eps_weight)
        for i in range(n_units):
            self.layers['MAXNET'].weight[i][i].data.fill_(1.)  # Self Weight = 1

        self.layers['MAXNET'].bias.data.fill_(0.)

        # Activation Function
        self.actfun = act_fun  # Function string-name attribute association

    # Forward Pass Method
    def forward(self, x):

        # Activation function object computation
        actfun = getattr(torch.nn, self.actfun)

        # Maxnet Layer pass                                              --> Output Weightening (Dot Product) "Linear transfc
        x = actfun()(self.layers['MAXNET'](x.squeeze().float()))

        # Self

        return x


# Build OCON MaxNetwork Architecture
torch.manual_seed(SEED)
ocon_maxnet = OCON_MaxNet(n_units=3, act_fun='ReLU', eps=0.25)

# MaxNet & Sub-Networks Parameters
print('OCON MaxNet STATE')
model_desc(ocon_maxnet)


def maxnet_algo(maxnet_function, n_units, act_fun, eps, input_array):
    """
    MaxNet Re-iteration algorithm for Maximum Value retrieving from an input array
    """
    non_zero_outs = np.count_nonzero(input_array)  # Non Zero Values initialization
    maxnet_in = torch.from_numpy(input_array)  # MaxNet Input Tensor initialization

    results = []  # Results initialization

    counter = 0
    while non_zero_outs != 1:
        counter += 1
```

```
        # Create the MaxNet
        torch.manual_seed(SEED)  # Redundant
        maxnet = maxnet_function(n_units = n_units, act_fun = act_fun, eps = eps)

        # Compute Forward Pass
        results = maxnet(maxnet_in)

        # Non_zero outputs & Maxnet Input Update
        non_zero_outs = np.count_nonzero(results.detach().numpy())
        maxnet_in = results.detach()  # Save results for next iteration

    print(f'Maximum Value found in {counter} iterations')
    return np.argmax(results.detach().numpy())


# MaxNet on Sub-Networks predictions
ocon_predictions_prob = np.zeros((len(ocon_predictions), x_data_minmax_ordered.shape[0]))  # NumPy predictions matrix (12 * 16

# Convert from List of Tensors to 2D NumPy Array
for i in range(len(ocon_predictions)):
    ocon_predictions_prob[i, :] = ocon_predictions[i].detach().squeeze().numpy()

maxnet_class_predictions = []  # Classes Outputs list initialization
# MaxNet application
for i in range(x_data_minmax.shape[0]):
    print(f'Dataset Sample({i + 1}) Class Evaluation')

    samp_predictions = ocon_predictions_prob[:, i]  # Array of 12 predictions for each Dataset sample (OCON outputs)
    class_prediction = maxnet_algo(OCON_MaxNet, n_units=3, act_fun='ReLU', eps=-0.1, input_array=samp_predictions)  # MaxNet C
    maxnet_class_predictions.append(class_prediction)  # Result appending
    print('-----------------------------------------------')

maxnet_accuracy = 100 * np.mean((np.array(maxnet_class_predictions).reshape(1597, 1) == z_labels_raw_np_alt_ordered))  # Accur
print(f'Maxnet Output --> Speaker ACCURACY: {maxnet_accuracy}%')


# Argmax on Sub-Networks predictions (...for multiple 1s probabilities MaxNet infinite loops)
#ocon_predictions_prob = np.zeros((len(new_ocon_predictions), x_data_minmax.shape[0]))  # NumPy predictions matrix (12 * 1617)
#
# Convert from List of Tensors to 2D NumPy Array
#for i in range(len(new_ocon_predictions)):
#    ocon_predictions_prob[i, :] = new_ocon_predictions[i].detach().squeeze().numpy()
#
#maxnet_class_predictions = np.argmax(ocon_predictions_prob, axis=0)
#maxnet_accuracy = 100 * np.mean((np.array(maxnet_class_predictions).reshape(1617, 1) == y_labels_raw_np))  # Accuracy computa
#print(f'Maxnet Output ACCURACY: {maxnet_accuracy}%')


# Evaluation Analysis Plot
plt.figure(figsize=(30, 5))
plt.suptitle(f'OCON Bank + MaxNet Evaluation: {maxnet_accuracy:.0f}%')

plot_y_ticks = [n for n in range(len(speakers_alt))]

plt.plot(z_labels_raw_np_alt_ordered, 'rs', label='Ground Truths')
plt.plot(maxnet_class_predictions, 'k.', label='MAXNET Outputs')
plt.xlabel('Dataset samples')
plt.xticks(ticks=plot_ticks, labels=speakers_alt)
plt.xlim([-10, len(y_labels_raw_np) + 10])
plt.ylabel('Labels')
plt.yticks(ticks=plot_y_ticks, labels=speakers_alt)
plt.legend(loc='best')
plt.grid()

plt.tight_layout()
plt.savefig('OCON_model_evaluation')
plt.show()
```