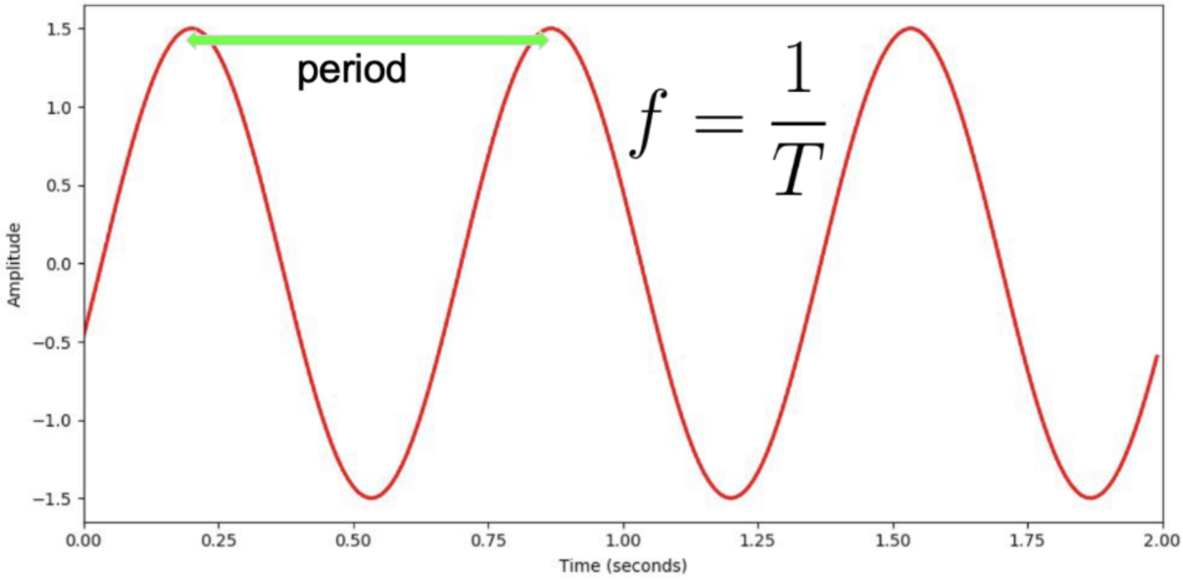
DLAP_3

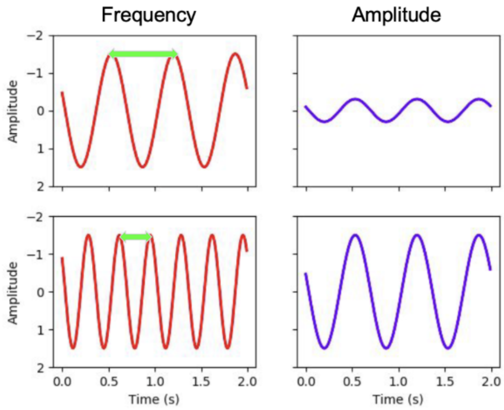
Simple neural network with Tensor

```
import tensorflow as tf
from sklearn.model_selection import train_test_split
import numpy as np
from random import random


def generate_dataset(num_samples, test_size=0.33):
    """Generates train/test data for sum operation
    :param num_samples (int): Num of total samples in dataset
    :param test_size (int): Ratio of num_samples used as test set
    :return x_train (ndarray): 2d array with input data for training
    :return x_test (ndarray): 2d array with input data for testing
    :return y_train (ndarray): 2d array with target data for training
    :return y_test (ndarray): 2d array with target data for testing
    """

    # build inputs/targets for sum operation: y[0][0] = x[0][0] + x[0][1]
    x = np.array([[random()/2 for _ in range(2)] for _ in range(num_samples)])
    y = np.array([[i[0] + i[1]] for i in x])

    # split dataset into test and training sets
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=test_size)
    return x_train, x_test, y_train, y_test


if __name__ == "__main__":

    # create a dataset with 2000 samples
    x_train, x_test, y_train, y_test = generate_dataset(5000, 0.3)

    # build model with 3 layers: 2 -> 5 -> 1
    model = tf.keras.models.Sequential([
      tf.keras.layers.Dense(5, input_dim=2, activation="sigmoid"),
      tf.keras.layers.Dense(1, activation="sigmoid")
    ])

    # choose optimiser
    optimizer = tf.keras.optimizers.SGD(learning_rate=0.1)

    # compile model
    model.compile(optimizer=optimizer, loss='mse')

    # train model
    model.fit(x_train, y_train, epochs=100)

    # evaluate model on test set
    print("\nEvaluation on the test set:")
    model.evaluate(x_test,  y_test, verbose=2)

    # get predictions
    data = np.array([[0.1, 0.2], [0.2, 0.2]])
    predictions = model.predict(data)

    # print predictions
    print("\nPredictions:")
    for d, p in zip(data, predictions):
        print("{} + {} = {}".format(d[0], d[1], p[0]))
```

Waveform



- period가 짧을수록 주파수는 높아짐



- 높은 Frequency는 높은 pitch를 나타냄
- 높은 Amplitude는 큰 소리를 나타냄

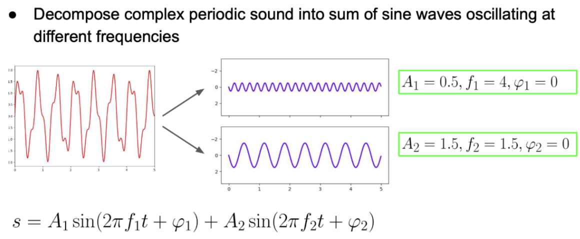
Sampling

- 연속 신호를 이산 신호로 변환할 때 1초에 몇 번 샘플하는지 나타내는 지표
    - 44100Hz = 1초를 44100개로 쪼갬

Quantization

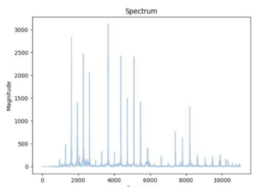
- 실수 범위의 이산 신호를 정수 범위의 이산 신호로 바꾸는것
    - 8비트 -128~127의 정수로 변환

Fourier transform



- time축을 frequency축으로 분해 및 표현이 가능함
    - 시간 정보가 없음

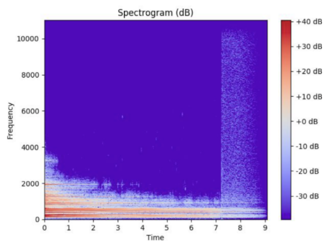
  → 푸리에 변환을 통해 나오는 그래프를 Spectrum이라함



STFT

- waveform을 특정한 길이 frame으로 잘라서 각 frame마다 푸리에 변환후 spectrum을 구함
    - 시간 정보 보존

  → STFT의 결과 - spectrogram (시간 변화에 따른 spectrum의 변화)



MFCCs

- Capture timbral/textural aspects of sound
- Frequency domain feature
- Approximate human auditory system
- 13 to 40 coefficients
- Calculated at each frame

Preprocessing audio data

```
import numpy as np
import librosa, librosa.display
import matplotlib.pyplot as plt

FIG_SIZE = (15,10)

file = "blues.00000.wav"
```

```
# load audio file with Librosa
signal, sample_rate = librosa.load(file, sr=22050)

# WAVEFORM
# display waveform
plt.figure(figsize=FIG_SIZE)
librosa.display.waveplot(signal, sample_rate, alpha=0.4)
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.title("Waveform")


# FFT -> power spectrum
# perform Fourier transform
fft = np.fft.fft(signal)

# calculate abs values on complex numbers to get magnitude
spectrum = np.abs(fft)
# 결과값이 실수부와 허수부로 나뉘어 나옴 -> 각 주파수의 크기를 알아내기 위해 절대값 취함

# create frequency variable
f = np.linspace(0, sample_rate, len(spectrum))

# take half of the spectrum and frequency
left_spectrum = spectrum[:int(len(spectrum)/2)]
left_f = f[:int(len(spectrum)/2)]
# 켤레 복소수로 대칭된 형태가 나오므로 반 나눠서 처음부분을 사용

# plot spectrum
plt.figure(figsize=FIG_SIZE)
plt.plot(left_f, left_spectrum, alpha=0.4)
plt.xlabel("Frequency")
plt.ylabel("Magnitude")
plt.title("Power spectrum")


# STFT -> spectrogram
hop_length = 512 # in num. of samples
n_fft = 2048 # window in num. of samples

# calculate duration hop length and window in seconds
hop_length_duration = float(hop_length)/sample_rate
n_fft_duration = float(n_fft)/sample_rate

print("STFT hop length duration is: {}s".format(hop_length_duration))
print("STFT window duration is: {}s".format(n_fft_duration))

# perform stft
stft = librosa.stft(signal, n_fft=n_fft, hop_length=hop_length)

# calculate abs values on complex numbers to get magnitude
spectrogram = np.abs(stft)

# display spectrogram
plt.figure(figsize=FIG_SIZE)
librosa.display.specshow(spectrogram, sr=sample_rate, hop_length=hop_length)
plt.xlabel("Time")
plt.ylabel("Frequency")
plt.colorbar()
plt.title("Spectrogram")

# apply logarithm to cast amplitude to Decibels
log_spectrogram = librosa.amplitude_to_db(spectrogram)

plt.figure(figsize=FIG_SIZE)
librosa.display.specshow(log_spectrogram, sr=sample_rate, hop_length=hop_length)
plt.xlabel("Time")
plt.ylabel("Frequency")
plt.colorbar(format="%+2.0f dB")
plt.title("Spectrogram (dB)")


# MFCCs
# extract 13 MFCCs
MFCCs = librosa.feature.mfcc(signal, sample_rate, n_fft=n_fft, hop_length=hop_length, n_mfcc=13)

# display MFCCs
plt.figure(figsize=FIG_SIZE)
librosa.display.specshow(MFCCs, sr=sample_rate, hop_length=hop_length)
plt.xlabel("Time")
plt.ylabel("MFCC coefficients")
plt.colorbar()
plt.title("MFCCs")

# show plots
plt.show()
```

Prepareing the Dataset

```
import json
import os
import math
import librosa

DATASET_PATH = "path/to/marsyas/dataset"
JSON_PATH = "data_10.json"
SAMPLE_RATE = 22050
TRACK_DURATION = 30 # measured in seconds
SAMPLES_PER_TRACK = SAMPLE_RATE * TRACK_DURATION


def save_mfcc(dataset_path, json_path, num_mfcc=13, n_fft=2048, hop_length=512, num_segments=5):
    """Extracts MFCCs from music dataset and saves them into a json file along witgh genre labels.
        :param dataset_path (str): Path to dataset
        :param json_path (str): Path to json file used to save MFCCs
        :param num_mfcc (int): Number of coefficients to extract
        :param n_fft (int): Interval we consider to apply FFT. Measured in # of samples
        :param hop_length (int): Sliding window for FFT. Measured in # of samples
        :param: num_segments (int): Number of segments we want to divide sample tracks into
        :return:
        """

    # dictionary to store mapping, labels, and MFCCs
    data = {
        "mapping": [],
        "labels": [],
        "mfcc": []
    }

    samples_per_segment = int(SAMPLES_PER_TRACK / num_segments)
    num_mfcc_vectors_per_segment = math.ceil(samples_per_segment / hop_length)

    # loop through all genre sub-folder
    for i, (dirpath, dirnames, filenames) in enumerate(os.walk(dataset_path)):

        # ensure we're processing a genre sub-folder level
        if dirpath is not dataset_path:

            # save genre label (i.e., sub-folder name) in the mapping
            semantic_label = dirpath.split("/")[-1]
            data["mapping"].append(semantic_label)
            print("\nProcessing: {}".format(semantic_label))

            # process all audio files in genre sub-dir
            for f in filenames:

    # load audio file
                file_path = os.path.join(dirpath, f)
                signal, sample_rate = librosa.load(file_path, sr=SAMPLE_RATE)

                # process all segments of audio file
                for d in range(num_segments):

                    # calculate start and finish sample for current segment
                    start = samples_per_segment * d
                    finish = start + samples_per_segment

                    # extract mfcc
                    mfcc = librosa.feature.mfcc(signal[start:finish], sample_rate, n_mfcc=num_mfcc, n_fft=n_fft, hop_length=hop_length
                    mfcc = mfcc.T

                    # store only mfcc feature with expected number of vectors
                    if len(mfcc) == num_mfcc_vectors_per_segment:
                        data["mfcc"].append(mfcc.tolist())
                        data["labels"].append(i-1)
                        print("{}, segment:{}".format(file_path, d+1))

    # save MFCCs to json file
    with open(json_path, "w") as fp:
        json.dump(data, fp, indent=4)


if __name__ == "__main__":
    save_mfcc(DATASET_PATH, JSON_PATH, num_segments=10)
```