پروژه نهایی درس سیستم های چندرسانه‌ای

اعضای گروه :

مریم مقتدری ۹۹۲۳۰۷۳

زهرا لطیفی ۹۹۲۳۰۶۹

املین غازاریان ۹۹۲۳۰۵۶

آیدا احمدی پارسا ۹۹۲۳۰۰۳

فهرست عناوین

مقدمه :

هدف از تشخیص گفتار که در متون علمی بیشتر با نام بازشناسی گفتار شناخته شده‌است، طراحی و پیاده‌سازی سیستمی است که اطلاعات گفتاری را دریافت و متن و فرمان گوینده را استخراج می‌کند. فناوری **بازشناسی گفتار** به رایانه‌ای که توانایی دریافت صدا را دارد (برای مثال به یک میکروفن مجهز است) این قابلیت را می‌دهد که گفتار کاربر را متوجه شود.

فناوری تبدیل گفتار به متن ممکن است به عنوان جایگزینی برای صفحه کلید یا ماوس برای وارد کردن دستورها مورد استفاده قرار گیرد. سیستم‌های تشخیص دهنده گفتار انواع مختلفی دارند، بعضی قادرند گفتار پیوسته را شناسایی نمایند، بعضی دیگر فقط می‌توانند گفتار گسسته (که بین کلمات سکوت وجود دارد) را شناسایی کنند. همچنین سیستم‌ها قادرند واژگان گفته شده توسط افراد مختلف یا فقط توسط یک گوینده را تشخیص دهند. به هر حال ایده‌آل‌ترین سیستم آن است که بتواند گفتار پیوسته غیر وابسته به گوینده را در محیط نویزی شناسایی نماید.

این سیستم‌ها با به‌کارگیری روش‌های مختلف طبقه بندی و شناسایی الگو قادر به تشخیص واژگان هستند که البته برای افزایش دقت در شناسایی از یک فرهنگ لغت نیز در انتهای سیستم استفاده می‌شود. روشهایی مانند Hidden Markov Modelیا Neural Network در بسیاری از سیستم‌های تشخیص گفتار مورد استفاده قرار می‌گیرند و در بخش‌های انتهایی سیستم از هوش مصنوعی کمک گرفته می‌شود.

یک سیستم بازشناسی گفتار خودکار (Automatic Speech Recognition) که به اختصار ASR نامیده می‌شود با چالشهای فراوانی روبروست. از جمله مهمترین این چالش‌ها می‌توان به وجود نویز، انتخاب مجموعه ویژگی‌های مناسب، انتخاب مدل آکوستیکی مناسب، تنوع زبان، تنوع جنسیت و مشکل لهجه در بازشناسی گفتار اشاره نمود. در مورد زبان‌های رایج مانند انگلیسی کارهای زیادی در جهت مقابله با این چالش‌ها انجام شده‌است اما در مورد زبان فارسی هنوز راه زیادی در پیش است.

الگوریتم پیاده‌سازی بر مبنای CNN :

ابتدا یک dataset شامل مجموعه‌ای از فایل‌های wave که همگی تقریبا ۳ ثانیه هستند تشکیل شد . هر فایل صوتی یک بردار با فرکانس نمونه‌برداری 16000 هرتز می‌باشد. برای اعمال CNN بر روی این داده‌ها ، spectrogram آنها را استخراج کرده و شبکه CNN را بر روی آن اعمال کنیم .

در این قسمت این پرسش مطرح می‌شود که چرا نمی‌توان از داده خام استفاده کرد؛ پاسخ به شرح زیر است :

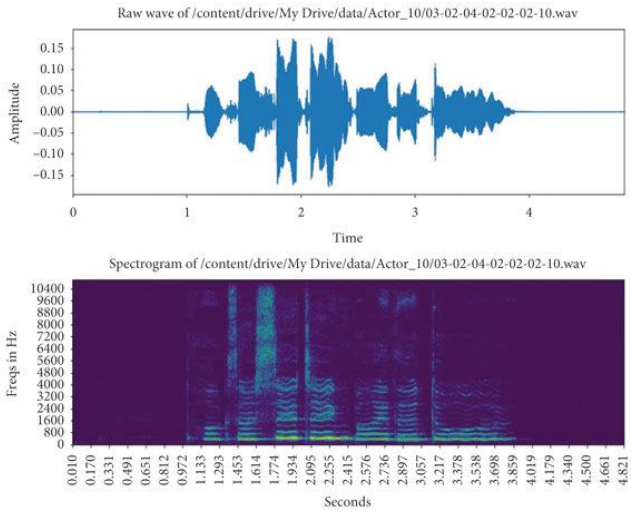
شبکه های CNN برای تصاویر دو بعدی به کار می‌روند . از طرفی spectrogram نیز یک تصویر دوبعدی از طیف فرکانسی صوت بر حسب زمان است . درنتیجه می‌توان از این روش بهره برد.

اما تفاوت هایی میان تصویر و spectrogram وجود دارد که شبکه CNN را در پردازش صوت دچار چالش خواهد کرد.

برای مثال ما اگر یک پیکسل از یک تصویر را داشه باشیم ، بر اساس رنگ و شکل تا حدی قابل حدس است که این پیکسل مرتبط با کدام تصویر می‌باشد ، اما اگر یک نمونه فرکانسی در یک لحظه مشخص از کل spectrogram را داشته باشیم ، قطعا قابل تشخیص نخواهد بود که این داده مربوط به چه صوتی می‌تواند باشد.

از طرفی معنای ابعاد تصویر و طیف کاملا متفاوت است. در تصویر ابعاد معرف مختصات x,y هستند اما محور افقی spectrogram معرف زمان بوده و محور عمودی آن معرف شدت فرکانس است . درنتیجه اگر نمونه را به صورت عمودی جابه‌جا کنیم داده ها به صورت کامل تغییر خواهند کرد.

برای نمونه در شکل زیر تفاوت دیتای صوتی خام و spectrogram مشاهده می‌شود :



دیتاهای ۳ ثانیه ای هر کدام شامل یک کلمه از مجموعه (up, down, right, left) در دیتاست قرار دارند که طی مراحلی ، فرکانس آنها را به 16000 هرتز تغییر داده و همچنین mono-channel شده‌اند. این مجموعه داده‌ها به صورت دستی و توسط ۱۳ صدای متفاوت ضبط شده‌اند.

بخش اول : آماده سازی دیتاست و نمونه‌ها برای Training

مرحله اول :

در ابتدا کتابخانه های مربوطه را import می‌کنیم :

- ✓ TensorFlow : برای ساختن و train کردن مدل CNN
- ✓ SeaBorn : کتابخانه‌ای برمبنای matplotlib برای data visualization
- ✓ Pyaudio : برای جمع آوری دیتای RealTime ضبط شده توسط میکروفون
- ✓ Librosa : پکیجی برای آنالیز موسیقی و صوت

```
✓ #Import Libraries and packages
✓ import os
✓ import pathlib
✓ import matplotlib.pyplot as plt
✓ import numpy as np
✓ import seaborn as sns
✓ import tensorflow as tf
✓ from tensorflow.keras import layers
✓ from tensorflow.keras import models
✓ from IPython import display
✓ import librosa
✓ import pyaudio
✓ import sys
✓ import keras
```

مرحله دوم :

به منظور reproductive بودن ، باید random number generator تعریف شود .

Random seed نقطه شروعی برای دسته‌ای از اعداد رندوم است. اولین عدد توسط seed مشخص شده و اعداد دیگر براساس عدد قبلی تولید می‌شوند. الگوریتم های متفاوتی برای تولید این اعداد سریال وجود دارد که البته به علت الگوریتمیک بودن ، درواقع اعداد شبه رندوم هستند.

لزوم وجود Seed به این صورت است که در صورت عدم وجود آن ، هر بار که کد ران می‌شود عدد متفاوتی برای نقطه شروع درنظر گرفته می‌شود و این اتفاق باعث می‌شود که قابلیت بازتولید نتایج قبلی و یا مقایسه مدل های متفاوت از بین برود.

```
seed = 40
tf.random.set_seed(seed)
np.random.seed(seed)
```

مرحله سوم :

دیتاست به صورت فایل زیپ در یک لینک ذخیره شده است ، ما باید این فایل را دانلود کرده و سپس فایل unzip شده آن را در مسیر صحیح ذخیره کنیم.

❖ نکته قابل توجه این است که این فرایند فقط یک بار انجام می‌شود و در مراحل بعدی صرفا مسیری که فایل‌ها در آن ذخیره شده‌اند را فراخوانی می‌کنیم.

```
❖ #Download .zip file and unzip it in data directory
❖ DATASET_PATH = 'datasetdir2/dataset'
❖ data_dir = pathlib.Path(DATASET_PATH)
❖ if not data_dir.exists():
❖   tf.keras.utils.get_file(
❖       'dataset.zip',
❖       origin="https://s8.uupload.ir/filelink/fEWlM65qSHQh_dabdf91cd9/datas
  et_kij6.zip",
❖       extract=True,
❖       cache_dir='.', cache_subdir='datasetdir2')
```

مرحله چهارم :

نمونه های هر کلاس در یک پوشه جداگانه داخل فایل زیپ ذخیره شده‌اند. می‌توانیم از نام آن پوشه ها برای ساختن آرایه دستورات استفاده کنیم.

```
#Extracting command, using the names of folders in our data directory
file_path = 'C:\\Users\\Berooz Stock\\datasetdir2'
commands = np.array(tf.io.gfile.listdir(str(file_path)))

#Making sure the .zip file name is not counted as a command
commands = commands[(commands != 'dataset.zip')]

#printing command array to be checked
print('Commands:', commands)
```

ابتدا مسیر فایل مشخص می‌شود. سپس با دستور np.array یک آرایه از نام پوشه های موجود در این مسیر می‌سازیم و درنهایت چک می‌کنیم که نام خود پوشه اصلی جز این دستورات نباشند و در مرحله آخر این آرایه را پرینت می‌کنیم تا از صحت آن مطمئن شویم.

مرحله پنجم :

در این مرحله یک دیتاست TensorFlow از فایل های صوتی ذخیره شده در مسیر مذکور می‌سازیم. ما از bach با سایز 32 استفاده کردیم و هر سری خروجی شامل 48000 bit است (هر دیتا ۳ ثانیه با فرکانس 16000 Hz). 80% از نمونه ها برای Training و مابقی برای Validation استفاده شده‌اند.

```
#Generate a TensorFlow dataset from audio files stored in the directory
train_ds, val_ds = tf.keras.utils.audio_dataset_from_directory(
    directory=file_path,
    batch_size=32,
    validation_split=0.2,
    seed=0,
    output_sequence_length=3*16000,
    subset='both')

#Extracting lable names based on our classes, and printing the lables array
label_names = np.array(train_ds.class_names)
print()
print("label names:", label_names)
```

نتایج این بخش شکل زیر است :

```
Found 420 files belonging to 4 classes.
Using 336 files for training.
Using 84 files for validation.
label names: ['down' 'left' 'right' 'up']
```

مرحله ششم :

یکی از اهداف ما جهت بهبود کارایی این سیستم ، این است که memory footprint را کاهش دهیم تا سرعت سیستم بالا برود. تابع squeeze همین وظیفه را به عهده دارد . به این صورت که آخرین بعد audio را که سایز ۱ داشته و برای پردازش استفاده نمی‌شود را حذف می‌کند.

نکته قابل تامل این است که تابع squeeze از map method استفاده می‌کند. یعنی به صورت موازی به هر درایه از دیتاست یک تابع map می‌کند و این روش برای دیتاست های بزرگ کارآمدتر می‌باشد.

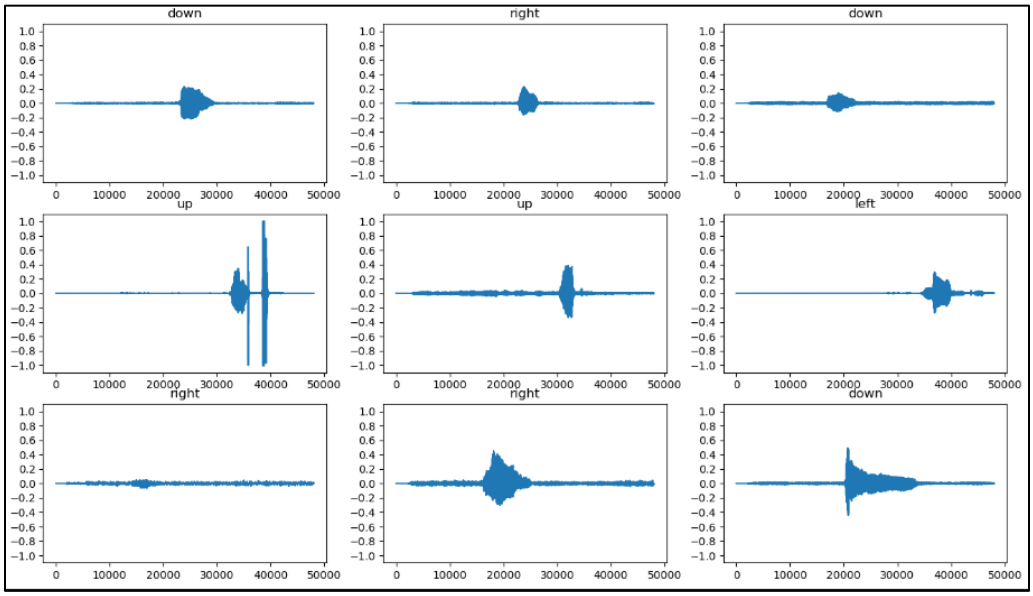
```
def squeeze(audio, labels):
  audio = tf.squeeze(audio, axis=-1)
  return audio, labels

#Apllying to our dataset
train_ds = train_ds.map(squeeze, tf.data.AUTOTUNE)
val_ds = val_ds.map(squeeze, tf.data.AUTOTUNE)


#Getting an example from our samples
for example_audio, example_labels in train_ds.take(1):
  print(example_audio.shape)
  print(example_labels.shape)
```

مرحله هفتم :

به منظور اطمینان از روند کار در این مرحله checkpoint انجام می‌دهیم. به این صورت که ۹ نمونه از دیتاست را plot می‌کنیم.

```
#Defining the number of columns and rows and making our subplot
rows = 3
cols = 3
n = rows * cols
fig, axes = plt.subplots(rows, cols, figsize=(16, 9))

#Plotting waveforms
for i in range(n):
  if i>=n:
    break
  r = i // cols
  c = i % cols
  ax = axes[r][c]
  ax.plot(example_audio[i].numpy())
  ax.set_yticks(np.arange(-1.2, 1.2, 0.2))
  label = label_names[example_labels[i]]
  ax.set_title(label)
  ax.set_ylim([-1.1,1.1])
plt.show()
```

نتایج این بخش به صورت زیر می‌باشد :



مرحله هشتم :

در این مرحله تابعی تعریف می‌کنیم که شکل موج را به spectrogram تبدیل کند.

```
def get_spectrogram(waveform):
  # Convert the waveform to a spectrogram via a STFT.
  spectrogram = tf.signal.stft(
      waveform, frame_length=512, frame_step=128)
  # Obtain the magnitude of the STFT.
  spectrogram = tf.abs(spectrogram)
  # Add a `channels` dimension, so that the spectrogram can be used as image-like
input data with convolution layers
  spectrogram = spectrogram[..., tf.newaxis]
  return spectrogram
```

Spectrogram توسط روشی به نام Short-Time-Fourier-Transform (STFT) ساخته می‌شود. این تابع محتوای فرکانسی یک سیگنال را در پنجره های کوتاه و دارای هم پوشانی ، محاسبه کرده و سپس محتوای فرکانسی بر حسب زمان رسم می‌شود تا spectrogram حاصل شود.

$$STFT(x(t),f,\tau)(\omega,\tau)=\int_{-\infty}^{\infty}x(t)w(t-\tau)e^{-i\omega t}dt$$

مرحله نهم :

مجددا در این مرحله برای اطمینان حاصل کردن از روند کار ، checkpoint انجام می‌دهیم.

```
for i in range(3):
  label = label_names[example_labels[i]]
  waveform = example_audio[i]
  spectrogram = get_spectrogram(waveform)

  print('Label:', label)
  print('Waveform shape:', waveform.shape)
  print('Spectrogram shape:', spectrogram.shape)
  print('Audio playback')
  display.display(display.Audio(waveform, rate=16000))
```

نتایج به شرح زیر است :

```
Label: left
Waveform shape: (48000,)
Spectrogram shape: (372, 257, 1)
Audio playback
Label: left
Waveform shape: (48000,)
Spectrogram shape: (372, 257, 1)
Audio playback
Label: down
Waveform shape: (48000,)
Spectrogram shape: (372, 257, 1)
Audio playback
```

مرحله دهم :

تابعی می‌نویسیم که spectrogramها را رسم کند.

```
def plot_spectrogram(spectrogram, ax):
  if len(spectrogram.shape) > 2:
    assert len(spectrogram.shape) == 3
    spectrogram = np.squeeze(spectrogram, axis=-1)
  # Convert the frequencies to log scale and transpose, so that the time is
represented on the x-axis (columns).
  # Add an epsilon to avoid taking a log of zero.
  log_spec = np.log(spectrogram.T + np.finfo(float).eps)
  height = log_spec.shape[0]
  width = log_spec.shape[1]
  X = np.linspace(0, np.size(spectrogram), num=width, dtype=int)
  Y = range(height)
  ax.pcolormesh(X, Y, log_spec)
```

نحوه کار هر بخش از این تابع به شرح زیر می‌باشد :

۱) تابع plot_spectrogram دو ورودی دارد . spectrogram که یک آرایه دوبعدی یا سه‌بعدی است. و ax که درواقع محوری است که Spectrogram روی آن نمایش داده می‌شود.

۲) اولین شرط if چک می‌کند که آیا ورودی اول تابع قبلی ، سه بعدی بود یا خیر. در این صورت باید تابع squeeze را اعمال کند.

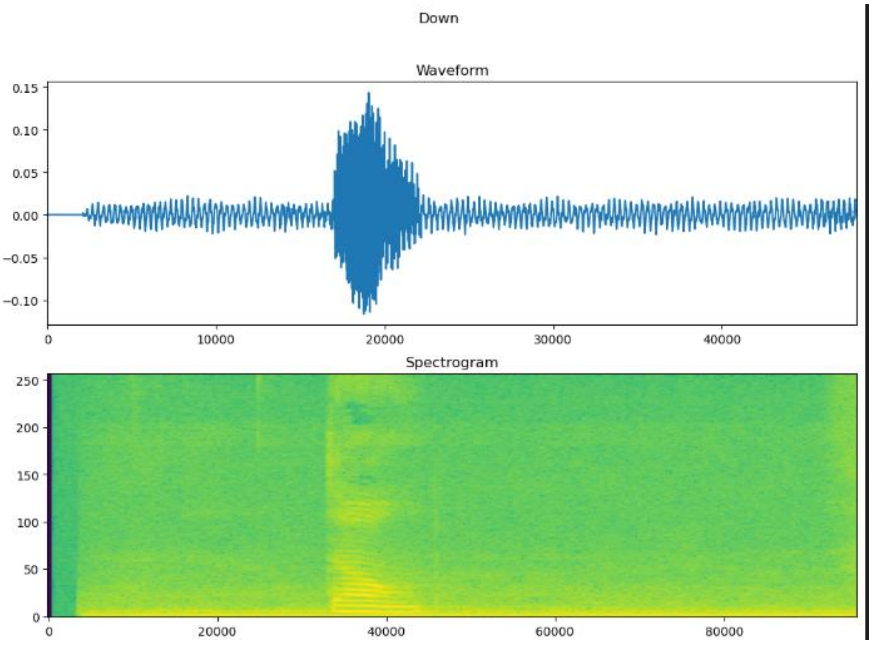
۳) خط بعدی دو تغییر روی spectrogram اعمال می‌کند. ابتدا با دستور log. از آن لگاریتم می‌گیرد و سپس با دستور T. آن را transpose می‌کند تا محور افقی مربوط به زمان و محور عمودی مربوط به طیف فرکانس شود. لازم به ذکر است که پیش از اعمال تابع لگاریتم یک مقدار اپسیلون به آن اضافه شده تا از وقوع لگاریتم صفر جلوگیری شود.

۴) سپس مقادیر روی محور xها و yها را با توابع موجود محاسبه می‌کنیم.

۵) در نهایت تابع pcolormesh یک pseudo-color از spectrogram می‌سازد که رنگ هر سلول آن معرف فرکانس در یک لحظه مشخص است.

مثالی از نحوه عملکرد این بخش از کد و مشاهده نتایج :

```
fig, axes = plt.subplots(2, figsize=(12, 8))
timescale = np.arange(waveform.shape[0])
axes[0].plot(timescale, waveform.numpy())
axes[0].set_title('Waveform')
axes[0].set_xlim([0, 48000])

plot_spectrogram(spectrogram.numpy(), axes[1])
axes[1].set_title('Spectrogram')
plt.suptitle(label.title())
plt.show()
```



مرحله یازدهم :

تابعی تعریف می‌کنیم که دیتاست را به عنوان ورودی دریافت کرده و روی هر یک از درایه های دیتاست یک Mapping Function اعمال می‌کند.

```
def make_spec_ds(ds):
  return ds.map(
      map_func=lambda audio,label: (get_spectrogram(audio), label),
      num_parallel_calls=tf.data.AUTOTUNE)
```

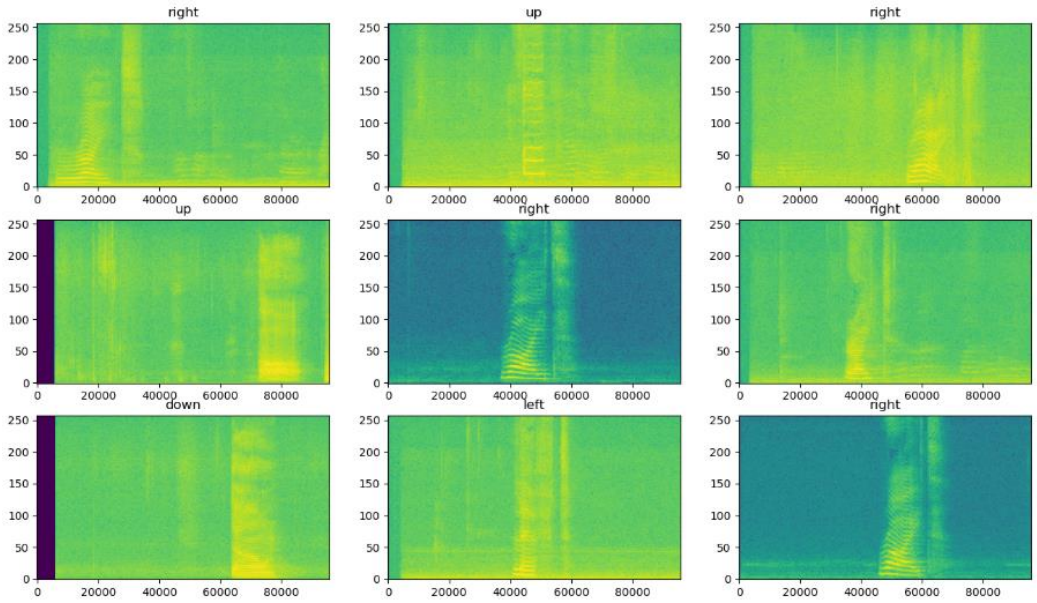
مرحله دوازدهم :

از تابع بالا برای ساختن دیتاست spectrogram به منظور Training و Validation استفاده می‌کنیم.

```
#creating new datasets with spectrograms
train_spectrogram_ds = make_spec_ds(train_ds)
val_spectrogram_ds = make_spec_ds(val_ds)

#Getting an example from our spectrograms
for example_spectrograms, example_spect_labels in train_spectrogram_ds.take(1):
  break
```

مرحله سیزدهم :

مجددا CheckPoint انجام می‌دهیم.



بخش دوم : Train کردن مدل CNN

مرحله اول :

در این مرحله باید سه عملیات روی spectrogram انجام شود :

۱) Caching : این تابع المان های دیتاست را از حافظه یا از disk ، Cache می‌کند . این عملیات برای دستیابی به سرعت بیشتر صورت می‌گیرد به این صورت که دیگر نیاز نیست که در هر epoch ، دیتا از روی disk لود شود و یا spectrogram آن محاسبه شود.

۲) Shuffle : این تابع ، به صورت رندوم المان های دیتاست را shuffle می‌کند. این عملیات با افزایش randomize کردن، از overfitting جلوگیری می‌کند.

۳) Prefetching : این تابع زمانی که مدل در حال Train شدن است ، در background دیتاست را prefetch می‌کند تا از زمان اضافه‌ای که باید صرف شود تا دیتا در دسترس قرار گیرد ، جلوگیری شود.

```
4) train_spectrogram_ds =
   train_spectrogram_ds.cache().shuffle(500).prefetch(tf.data.AUTOTUNE)
5) val_spectrogram_ds = val_spectrogram_ds.cache().prefetch(tf.data.AUTOTUNE)
6) #test_spectrogram_ds =
   test_spectrogram_ds.cache().prefetch(tf.data.AUTOTUNE)
```

مرحله دوم :

در این بخش با استفاده از Tensor Flow Keras API یک مدل sequential برای CNN می‌سازیم. با توجه به اینکه دیتاست کوچک است ، مدل های پیچیده به overfitting ختم می‌شوند. درنتیجه مدل را ساده طراحی کنیم و همچنین لایه Dropout برای کاهش احتمال overfitting به کار می‌رود.

```
#Getting the shape of the input spectrograms
input_shape = example_spectrograms.shape[1:]
print('Input shape:', input_shape)

#Getting the number of labels
num_labels = len(label_names)

# Instantiating the `tf.keras.layers.Normalization` layer.
norm_layer = layers.Normalization()

# Fitting the state of the layer to the spectrograms with `Normalization.adapt`.
norm_layer.adapt(data=train_spectrogram_ds.map(map_func=lambda spec, label:
spec))

#Adding the desired layers to our model
model = models.Sequential([
    layers.Input(shape=input_shape),
    # Normalize.
```

```
    #norm_layer,

    layers.Conv2D(24, 3, activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(32, 3, activation='relu'),
    layers.MaxPooling2D(),
    #layers.BatchNormalization(),
    layers.Dropout(0.2),
    layers.Conv2D(32, 3, activation='relu'),
    layers.MaxPooling2D(),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(num_labels),
])

# Printing out the summary of the created model
model.summary()
```

```
Input shape: (372, 257, 1)
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 370, 255, 24)      240

 max_pooling2d (MaxPooling2D  (None, 185, 127, 24)     0
 )

 conv2d_1 (Conv2D)           (None, 183, 125, 32)      6944

 max_pooling2d_1 (MaxPooling  (None, 91, 62, 32)       0
 2D)

 dropout (Dropout)           (None, 91, 62, 32)        0

 conv2d_2 (Conv2D)           (None, 89, 60, 32)        9248

 max_pooling2d_2 (MaxPooling  (None, 44, 30, 32)       0
 2D)

 flatten (Flatten)           (None, 42240)             0

 dense (Dense)               (None, 64)                2703424
...
Total params: 2,720,116
Trainable params: 2,720,116
Non-trainable params: 0
```

مرحله سوم :

با استفاده از Adam Optimizer مدل ساخته شده را کامپایل می‌کنیم.

```
#Compiling model
model.compile(
    optimizer=tf.keras.optimizers.Adam(),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=['accuracy'],
)
```

مرحله چهارم :

به تعداد epoch 10 تعریف می‌کنیم تا مدل را با استفاده از دیتاست ساخته شده ، fit کنیم. پارامتری به عنوان patience تعریف می‌کنیم و مقدار آن را initialize 3 می‌کنیم به این منظور که اگر loss بعد از epoch 3 بهبود نیابد ، مدل متوقف شود.

```
#Defining the number of epochs
EPOCHS = 10

#Model fitting step
history = model.fit(
    train_spectrogram_ds,
    validation_data=val_spectrogram_ds,
    epochs=EPOCHS,
    callbacks=tf.keras.callbacks.EarlyStopping(verbose=1, patience=3),
)
```

```
Epoch 1/10
17/17 [==============================] - 38s 2s/step - loss: 1.5289 -
 accuracy: 0.3048 - val_loss: 1.3914 - val_accuracy: 0.2969
Epoch 2/10
17/17 [==============================] - 35s 2s/step - loss: 1.2623 -
 accuracy: 0.4210 - val_loss: 1.2682 - val_accuracy: 0.4219
Epoch 3/10
17/17 [==============================] - 36s 2s/step - loss: 1.0110 -
 accuracy: 0.5810 - val_loss: 1.0496 - val_accuracy: 0.6250
Epoch 4/10
17/17 [==============================] - 35s 2s/step - loss: 0.8360 -
 accuracy: 0.6857 - val_loss: 0.9711 - val_accuracy: 0.6562
Epoch 5/10
17/17 [==============================] - 35s 2s/step - loss: 0.6099 -
 accuracy: 0.7829 - val_loss: 0.8847 - val_accuracy: 0.6719
Epoch 6/10
17/17 [==============================] - 36s 2s/step - loss: 0.4355 -
 accuracy: 0.8210 - val_loss: 0.8696 - val_accuracy: 0.7031
Epoch 7/10
17/17 [==============================] - 36s 2s/step - loss: 0.3254 -
 accuracy: 0.8629 - val_loss: 0.8118 - val_accuracy: 0.8438
```

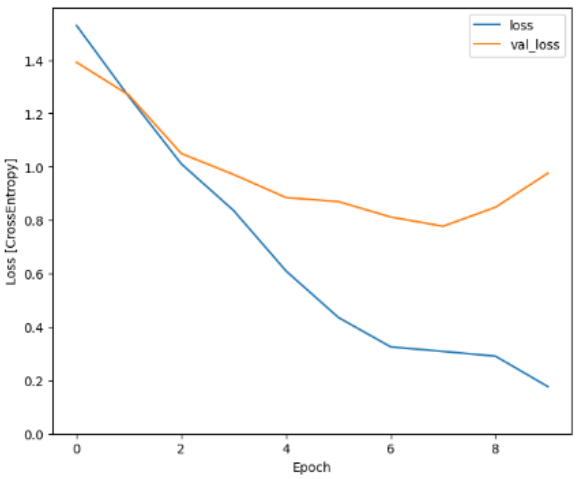
```
Epoch 8/10
17/17 [==============================] - 36s 2s/step - loss: 0.3086 -
 accuracy: 0.9105 - val_loss: 0.7774 - val_accuracy: 0.8125
Epoch 9/10
17/17 [==============================] - 36s 2s/step - loss: 0.2907 -
 accuracy: 0.9181 - val_loss: 0.8485 - val_accuracy: 0.7812
Epoch 10/10
17/17 [==============================] - 36s 2s/step - loss: 0.1766 -

 accuracy: 0.9543 - val_loss: 0.9760 - val_accuracy: 0.8281
```

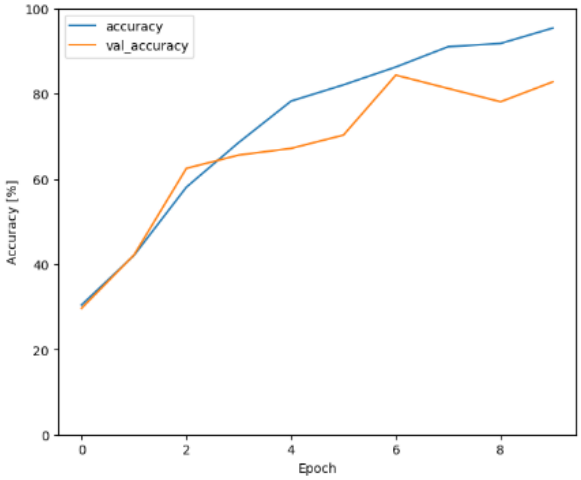
مرحله پنجم :

در این مرحله نتایج validation , training را برای Loss و Accuracy بر حسب تابعی از epoch رسم می‌کنیم. با افزایش تعداد epoch ها باید دقت افزایش یافته و loss کاهش یابد.

```
#Plotting Loss values for training and validation
metrics = history.history
plt.figure(figsize=(16,6))
plt.subplot(1,2,1)
plt.plot(history.epoch, metrics['loss'], metrics['val_loss'])
plt.legend(['loss', 'val_loss'])
plt.ylim([0, max(plt.ylim())])
plt.xlabel('Epoch')
plt.ylabel('Loss [CrossEntropy]')

#Plotting Accuracy values for training and validation
plt.subplot(1,2,2)
plt.plot(history.epoch, 100*np.array(metrics['accuracy']),
100*np.array(metrics['val_accuracy']))
plt.legend(['accuracy', 'val_accuracy'])
plt.ylim([0, 100])
plt.xlabel('Epoch')
plt.ylabel('Accuracy [%]')
```

نتایج به صورت زیر می‌باشد :



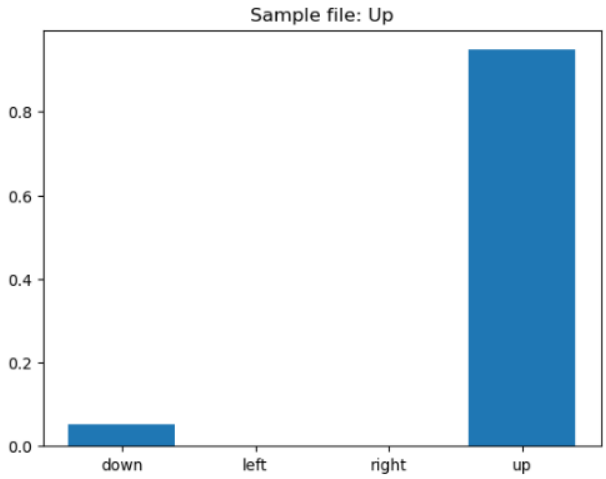


مرحله ششم :

در این مرحله CheckPoint انجام می‌دهیم. برای آزمایش یک نمونه که در دیتاست موجود نیست را به عنوان ورودی به مدل می‌دهیم.

```
#Reading the test file and decode it using the desired lengths and sample rate,
and number of channels
x = tf.io.read_file('Up6.wav')
x, sample_rate = tf.audio.decode_wav(x, desired_channels=1,
desired_samples=3*16000)
#Squeezing to remove the extra axis
x = tf.squeeze(x, axis=-1)
waveform = x
#Getting the spectrogram from waveform
x = get_spectrogram(x)
x = x[tf.newaxis,...]

#Predict the command, using our trained model
prediction = model(x)

#Plotting the results as a bar chart
x_labels = ['down', 'left', 'right', 'up']
plt.bar(x_labels, tf.nn.softmax(prediction[0]))
plt.title('up6')
plt.show()

#Displaying the original test sample, and the value of the predictions for each
command label
display.display(display.Audio(waveform, rate=16000))
print(prediction)
```

در ابتدا دیتا را خوانده و decode می‌کنیم و سپس عملیات ذکر شده در بخش اول را انجام می‌دهیم. پس از آن با استفاده از مدل ساخته شده ، چک می‌کند که سیستم چه کلمه‌ای را predict می‌کند.



مرحله هفتم :

در مرحله evaluation با استفاده از دیتای تست ، دقت را محاسبه می‌کنیم.

```
model.evaluate(test_spectrogram_ds, return_dict=True)
```

```
3/3 [==============================] - 2s 327ms/step - loss: 0.6643 - accuracy: 0.7313
{'loss': 0.664343535900116, 'accuracy': 0.7313432693481445}
```

مرحله هشتم :

در این مرحله مدل را ذخیره می‌کنیم.

```
model.save('saved_model3')
```

بخش سوم : تشخیص دستورات Real-Time

مرحله اول :

تابعی برای PreProcessing تعریف می‌کنیم که شامل ۵ مرحله است :

۱) شکل موج ورودی از 1- تا 1+ نرمالیزه می‌شوند. به این منظور دیتا را بر 32768 تقسیم می‌کنیم.
۲) شکل موج با دیتای float32 bit به tensor تبدیل می‌شود.
۳) تابع get_spectrogram را برای محاسبه Spectrogram ورودی استفاده می‌کنیم.
۴) یک بعد اضافی به ابتدای tensor spectrogram اضافه می‌شود تا shape دیتا به صورت زیر باشد

(1, height, width, channels)

۵) درنهایت spectrogram tensor به عنوان خروجی تابع نمایش داده می‌شود.

```
def preprocess_audiobuffer(waveform):
    """
    waveform: ndarray of size (48000, )

    output: Spectogram Tensor of size: (1, `height`, `width`, `channels`)
    """
    #  normalizing from [-32768, 32767] to [-1, 1]
    waveform =  waveform / 32768

    #Converting the raw waveform to a tensor
    waveform = tf.convert_to_tensor(waveform, dtype=tf.float32)

    #Getting the spectogram from the waveform
    spectogram = get_spectrogram(waveform)

    # adding one dimension
    spectogram = tf.expand_dims(spectogram, 0)

    return spectogram
```

مرحله دوم :

این تابع برای ریکورد کردن صوت از طریق میکروفون و و تبدیل آن به بردار numpy می‌باشد.

```
#Defining the size of buffer, channels, sampling rate and format
FRAMES_PER_BUFFER = 3200
FORMAT = pyaudio.paInt16
CHANNELS = 1
RATE = 16000
p = pyaudio.PyAudio()

def record_audio():
    stream = p.open(
        format=FORMAT,
        channels=CHANNELS,
        rate=RATE,
        input=True,
        frames_per_buffer=FRAMES_PER_BUFFER
    )
```

```
    ##

    frames = []
    seconds = 3
    #Reading data from stream for 3 seconds
    for i in range(0, int(RATE / FRAMES_PER_BUFFER * seconds)):
        data = stream.read(FRAMES_PER_BUFFER)
        frames.append(data)

    ##

    #Stopping the stream and closing it
    stream.stop_stream()
    stream.close()

    return np.frombuffer(b''.join(frames), dtype=np.int16)


def terminate():
    p.terminate()
    frames = []
    seconds = 3
    #Reading data from stream for 3 seconds
    for i in range(0, int(RATE / FRAMES_PER_BUFFER * seconds)):
        data = stream.read(FRAMES_PER_BUFFER)
        frames.append(data)

    #print("recording stopped")

    #Stopping the stream and closing it
    stream.stop_stream()
    stream.close()

    return np.frombuffer(b''.join(frames), dtype=np.int16)


def terminate():
    p.terminate()
```

مرحله سوم :

در این مرحله دیتایی که توسط میکروفون ضبط شده است را به مدل train شده می‌دهیم. نکته قابل توجه این است که باید یک مقدار threshold مشخص کنیم که اگر max value از این مقدار کمتر باشد ، چیزی نمایش داده نشود.اما اگر max value از

threshold بیشتر بوده و قابل اعتماد باشد ، کلمه predict شده نمایش داده می‌شود. لازم به ذکر است که مقدار threshold با آژمون و خطا بدست آمده و به کیفیت میکروفون تحت آزمایش ، مرتبط است.

```
loaded_model = models.load_model("saved_model_final")

def predict_mic():
    #Getting Data from microphone, and preprocess it
    audio = record_audio()
    spec = preprocess_audiobuffer(audio)

    #Making the prediction, using our trained model
    prediction = loaded_model(spec)
    #print(prediction)

    #Checking if the prediction is reliable enough
    if np.max(prediction) > 18:
        label_pred = np.argmax(prediction, axis=1)
        #Printing the predicted command
        command = commands[label_pred[0]]
        print("Predicted label:", command)
    else:
        command = 0
    return command
```

مرحله چهارم :

تست نهایی در این مرحله صورت می‌گیرد .

```
if __name__ == "__main__":
    try:
        while True:
            command = predict_mic()
    except KeyboardInterrupt:
        print('Interrupted')
```

نتایج نهایی به صورت زیر می‌باشد :

```
Predicted label: down
Predicted label: right
Predicted label: right
Predicted label: up
Predicted label: right
Predicted label: left
Predicted label: down
Predicted label: left
Predicted label: up
Predicted label: down
Predicted label: down
Predicted label: down
Predicted label: down
Predicted label: right
Predicted label: down
Predicted label: down
Interrupted
```

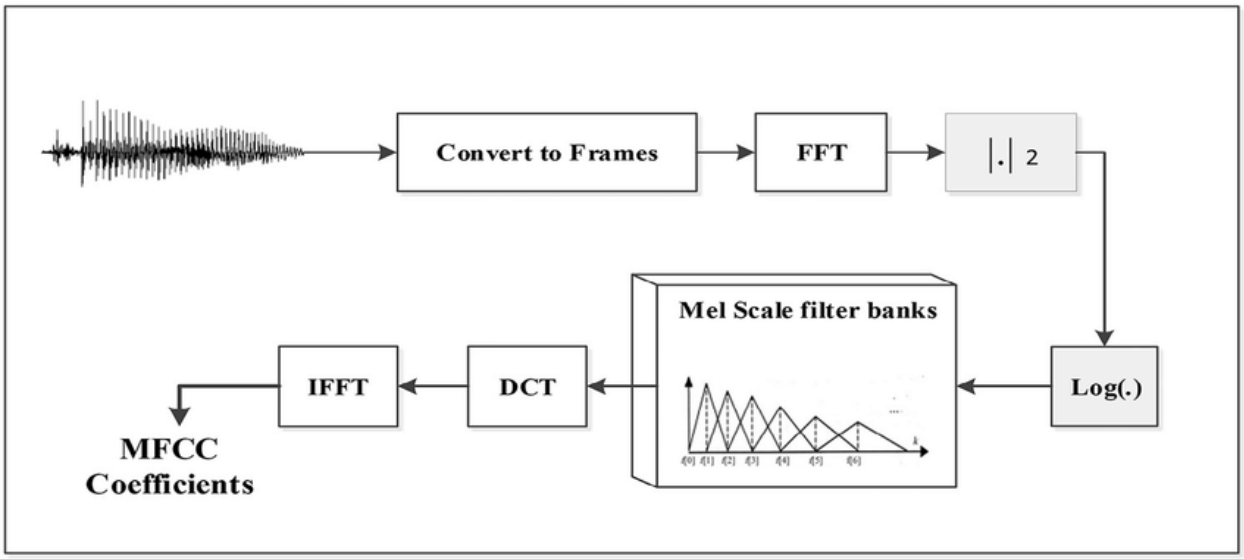
الگوریتم پیاده‌سازی با استفاده از روش MFCCs (Mel frequency cepstral coefficients) :

MFCC محبوب ترین و پرکاربردترین الگوریتم استخراج ویژگی(feature extraction) در زمینه سیستم فرمان صوتی(سیستم تشخیص گفتار) است . MFCC ها به عنوان یک ابزار پیشرفته برای استخراج اطلاعات از نمونه های صوتی شناخته می شود.

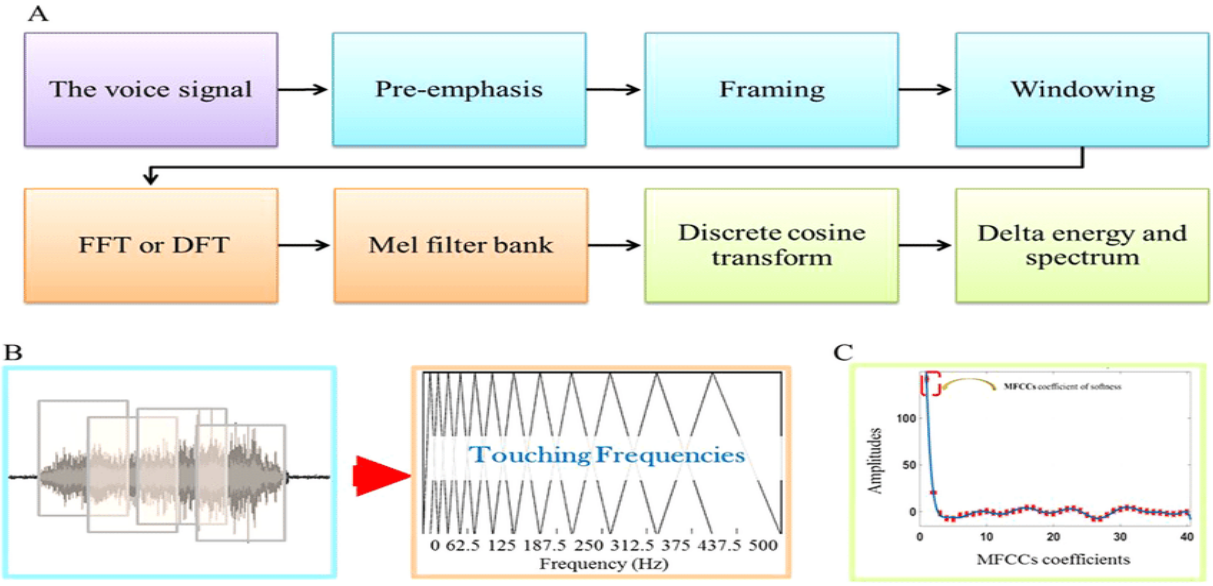
مراحل محاسبه MFCC برای یک نمونه صوتی داده شده:

- برش دادن سیگنال به فریم های کوتاه (از زمان)
- محاسبه طیف توان تخمین پریودوگرام برای هر فریم سیگنال
- اعمال Mel-space filter banks بر روی طیف توان و مجموع انرژی فیلترها.
- محاسبه لگاریتم انرژی filter banks و محاسبه DCT (تبدیل کسینوس گسسته) از لگاریتم ها

نمایش به صورت بلوک دیاگرام :



همانطور که در بلوک دیاگرام نشان داده شده است، هدف pre-emphasis تقویت فرکانس های بالاتر در ورودی است.



فرکانس‌های بالا در مقایسه با فرکانس‌های پایین‌تر، اندازه کمتری دارند و این pre-emphasis باعث می شود magnitude فرکانس های بالا افزایش پیدا کند.که نمایش ریاضی آن به این صورت است :

$$y= x(t) - \propto x(t-1)$$

که در اینجا x سیگنال ورودی و y سیگنال خروجی است

$$\propto=(0.95 , 0.97)$$

pre-emphasis : به دلیل مشکلات مربوط به مقادیر سیگنال صوتی در فرآیند استفاده از FFT (تبدیل فوریه سریع) مفید است

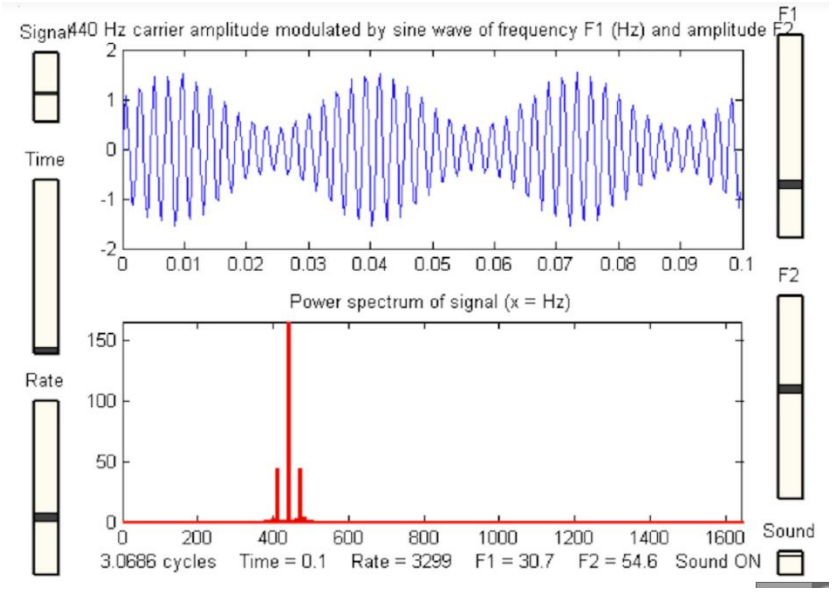
توضیح مراحل:

برش دادن سیگنال به فریم های کوتاه (از زمان) :

برش سیگنال صوتی به فریم های کوتاه از این جهت مفید است که به ما امکان می دهد صدای خود را در مراحل زمانی گسسته نمونه برداری کنیم. ما فرض می‌کنیم که در مقیاس‌های زمانی کوتاه، سیگنال صوتی تغییر نمی‌کند. مقادیر معمول برای مدت زمان فریم های کوتاه بین ۲۰ تا ۴۰ میلی ثانیه است. همپوشانی هر فریم ۱۰ تا ۱۵ میلی‌ثانیه نیز معمول است.

محاسبه طیف توان تخمین پریودوگرام برای هر فریم سیگنال :

هنگامی که فریم های خود را داریم، باید طیف توان هر فریم را محاسبه کنیم. طیف توان یک سری زمانی، توزیع توان را به اجزای فرکانس تشکیل دهنده آن سیگنال توصیف می کند. بر اساس تحلیل فوریه، هر سیگنال فیزیکی را می توان به تعدادی فرکانس گسسته، یا طیفی از فرکانس ها در یک محدوده پیوسته تجزیه کرد.



اعمال Mel-space filter banks بر روی طیف توان و مجموع انرژی فیلترها :

– حلزون گوش انسان به خوبی فرکانس های نزدیک را تشخیص نمی دهد و این اثر تنها با افزایش فرکانس ها بیشتر می شود. مقیاس mel ابزاری است که به ما امکان می دهد پاسخ سیستم شنوایی انسان را با دقت بیشتری نسبت به باندهای فرکانسی خطی تقریب کنیم.

– با افزایش فرکانس، فیلترهای mel گسترده تر می شوند، در فرکانس های پایین، جایی که تفاوت ها برای گوش انسان قابل تشخیص تر است و بنابراین در تجزیه و تحلیل ما اهمیت بیشتری دارد، فیلترها باریک هستند.

– اندازه های طیف‌های توان ما، که با اعمال تبدیل فوریه به داده‌های ورودی ما بدست امده اند ،با همبستگی(correlation) آنها با هر فیلتر Mel مثلثی binned* می‌شوند. این binning معمولاً به‌گونه‌ای اعمال می‌شود که هر ضریب در بهره فیلتر مربوطه ضرب شود، بنابراین هر فیلتر Mel جمع وزنی را نشان می‌دهد که نشان‌دهنده اندازه طیفی در آن کانال است.

Binned: data binning , also called data binning or data bucketing, is a data pre-processing technique used to reduce the effects of minor observation errors.

محاسبه لگاریتم انرژی filter banks :

– هنگامی که انرژی های bank filter خود را داریم، لگاریتم هر کدام را می گیریم. این مرحله دیگری است که به دلیل محدودیت های شنوایی انسان ایجاد می شود. انسان تغییرات volume را در مقیاس خطی درک نمی کند.

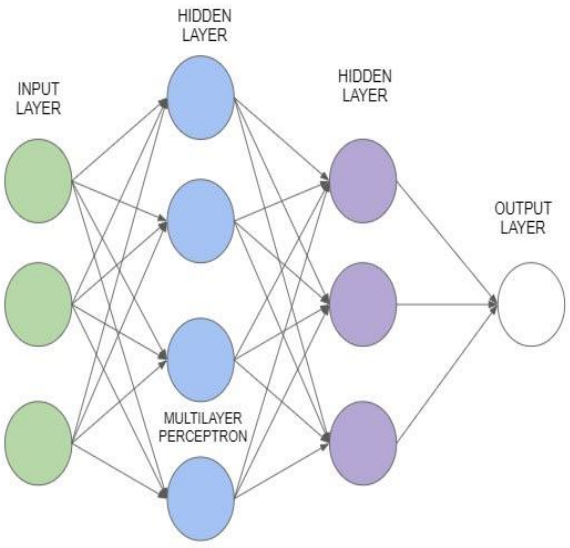
محاسبه DCT (تبدیل کسینوس گسسته) از لگاریتم ها :

تبدیل کسینوس گسسته (DCT) به خروجی mel-scale filter banks اعمال می شود زیرا خروجی آنها

بسیار correlated است که یک مشکل است برای classification و الگوریتم های ماشین لرنینگ . این تبدیل کسینوس گسسته خروجی را decorrelate می کند و نمایش فیلتر ها را فشرده می کند.

شبکه mpl : به طور خلاصه در این شبکه عصبی، نگاشت بین ورودی و خروجی غیر خطی است.

یک پرسپترون چند لایه دارای لایه های ورودی و خروجی و یک یا چند لایه پنهان با نورون های زیادی است که در کنار هم قرار گرفته اند. و در حالی که در پرسپترون، نورون باید یک تابع فعال سازی داشته باشد که آستانه ای را تحمیل می کند، مانند ReLU یا سیگموئید، نورون های یک پرسپترون چند لایه می توانند از هر تابع فعال سازی دلخواه استفاده کنند.انتخاب دقیق activation function تاثیر زیادی در عملکرد شبکه دارد



برسی توابع فعال سازی:

پرسپترون چندلایه در دسته الگوریتم‌های feedforward قرار می‌گیرد، زیرا ورودی‌ها با وزن‌های اولیه در مجموع وزنی ترکیب می‌شوند و دقیقاً مانند پرسپترون، تابع فعال‌سازی می‌شوند. اما تفاوت این است که هر ترکیب خطی به لایه بعدی انتشار می یابد.هر لایه در حال تغذیه لایه بعدی با نتیجه محاسبات خود است .حال اگر الگوریتم فقط یک تکرار را محاسبه کند یادگیری نخواهیم داشت پس در اینجا نیاز به backpropagation داریم که آن مکانیزم یادگیری است که به پرسپترون چندلایه اجازه می دهد تا وزن ها را در شبکه به طور مکرر تنظیم کند، با هدف به حداقل رساندن loss function.

توضیح کد :

- ❖ به علت مشابه بودن قسمت هایی از کد CNN و MLP ، در این بخش صرفا به توضیح بخش هایی از کد MLP می‌پردازیم که با CNN تفاوت دارد.

بخش اول :

- اولین تفاوت در عدد seed می‌باشد ، که الگوریتم MLP ، این عدد 42 انتخاب شده است.

پس از مرحله تعریف تابع squeez در بخش اول ، توابعی مرتبط با mfcc تعریف می‌کنیم.

```
def my_MFCC(waveform):
  batch_size, num_samples, sample_rate = 32, 48000, 16000.0
# A Tensor of [batch_size, num_samples] mono PCM samples in the range [-1, 1].
  pcm = waveform #tf.random.normal([batch_size, num_samples], dtype=tf.float32)

# A 1024-point STFT with frames of 64 ms and 75% overlap.
  stfts = tf.signal.stft(pcm, frame_length=512, frame_step=256,
                         fft_length=512)
  spectrograms = tf.abs(stfts)

# Warp the linear scale spectrograms into the mel-scale.
  num_spectrogram_bins = stfts.shape[-1]
  lower_edge_hertz, upper_edge_hertz, num_mel_bins = 80.0, 7600.0, 80
  linear_to_mel_weight_matrix = tf.signal.linear_to_mel_weight_matrix(
      num_mel_bins, num_spectrogram_bins, sample_rate, lower_edge_hertz,
      upper_edge_hertz)
  mel_spectrograms = tf.tensordot(spectrograms, linear_to_mel_weight_matrix, 1)
  mel_spectrograms.set_shape(spectrograms.shape[:-1].concatenate(
      linear_to_mel_weight_matrix.shape[-1:]))

# Compute a stabilized log to get log-magnitude mel-scale spectrograms.
  log_mel_spectrograms = tf.math.log(mel_spectrograms + 1e-6)
```

```
# Compute MFCCs from log_mel_spectrograms and take the first 13.
  mfccs = tf.signal.mfccs_from_log_mel_spectrograms(
      log_mel_spectrograms)[..., :13]
  # tmp = mfccs.shape
  # out = tf.reshape(mfccs,[tmp[1]*tmp[2]])
  return(mfccs)
```

در مرحله بعد به منظور تست توابع تعریف شده ، checkpoint اعمال می‌کنیم.

```
for i in range(3):
  label = label_names[example_labels[i]]
  waveform = example_audio[i]
  MFCC = my_MFCC(waveform)

  print('Label:', label)
  print('Waveform shape:', waveform.shape)
  print('MFCC shape:', MFCC.shape)
  print('Audio playback')
  display.display(display.Audio(waveform, rate=16000))
```

نتایج به صورت زیر می‌باشد :

```
Label: right
Waveform shape: (48000,)
MFCC shape: (184, 13)
Audio playback
Label: right
Waveform shape: (48000,)
MFCC shape: (184, 13)
Audio playback
Label: up
Waveform shape: (48000,)
MFCC shape: (184, 13)
Audio playback
```

در روش قبلی ، پس از این مرحله باید شکل موج ها را به spectrogram تبدیل می‌کردیم اما در این روش باید تابعی تعریف کنیم که دیتاست را با عنوان ورودی دریافت کرده و یک mapping function برای هر المان اعمال می‌کنیم . mapping function درواقع شکل موج صوت و label آن (یکی از ۴ واژه up, down, left, right) را به عنوان ورودی دریافت کرده و یک tupple به عنوان خروجی تحویل می‌دهد.

```
def make_MFCC_ds(ds):
  return ds.map(
      map_func=lambda audio,label: (my_MFCC(audio), label),
      num_parallel_calls=tf.data.AUTOTUNE)
```

سپس از تابع فوق برای ساختن دیتاست MFCC جهت training و validation استفاده می‌کنیم.

```
#creating new datasets with MFCCs
train_features_ds = make_MFCC_ds(train_ds)
val_features_ds = make_MFCC_ds(val_ds)
test_features_ds = make_MFCC_ds(test_ds)
for example_MFCCs, example_spect_labels in train_features_ds.take(1):
 break
```

بخش دوم :

از جمله تفاوت های مهم و کلیدی این دو روش ، ساختن مدل MLP می‌باشد.

```
#Getting the shape of the input spectrograms
input_shape = example_MFCCs.shape[1:]
# input_shape = (186, 13)
print('Input shape:', input_shape)

#Getting the number of labels
num_labels = len(label_names)

# Instantiating the `tf.keras.layers.Normalization` layer.
norm_layer = layers.Normalization()

# Fitting the state of the layer to the spectrograms with `Normalization.adapt`.
norm_layer.adapt(data=train_features_ds.map(map_func=lambda spec, label: spec))

#Adding the desired layers to our model
model = models.Sequential([
    layers.Input(shape=input_shape),
```

```
#     norm_layer,
    layers.Flatten(),
    layers.Dense(1000, activation='relu'),
#     layers.Dense(100, activation='relu'),
#     layers.BatchNormalization(synchronized=True),
#     layers.Dropout(0.2),
    layers.Dense(500, activation='relu'),
    #layers.BatchNormalization(),
#     layers.Dropout(0.2),
#     layers.Dense(100, activation='relu'),
    layers.Dropout(0.4),
    layers.Dense(num_labels, activation = 'softmax'),
#     layers.Dense(num_labels),

])

# Printing out the summary of the created model
model.summary()
```

به این دلیل که دیتاست به صورت دستی توسط افراد گروه و با لوازم ضبط غیر حرفه‌ای ساخته شده است ، بعضی از دیتاها دارای نویز می‌باشند. و به همین علت برای دستیابی به بهترین نتیجه از نظر درصد accuracy و loss ، مدل های مختلف با لایه‌های مختلف تست شده‌اند و خط های کامنت شده بیانگر این موضوع می‌باشند.

خلاصه ای از خروجی مدل طراحی شده به شکل زیر می‌باشد :

```
Input shape: (184, 13)
Model: "sequential_20"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_20 (Flatten)        (None, 2392)              0

 dense_61 (Dense)            (None, 1000)              2393000

 dense_62 (Dense)            (None, 500)               500500

 dropout_25 (Dropout)        (None, 500)               0

 dense_63 (Dense)            (None, 4)                 2004

=================================================================
Total params: 2,895,504
Trainable params: 2,895,504
Non-trainable params: 0
```

سپس همانند روش قبلی ، مجددا با Adam Optimizer مدل طراحی شده را کامپایل می‌کنیم .

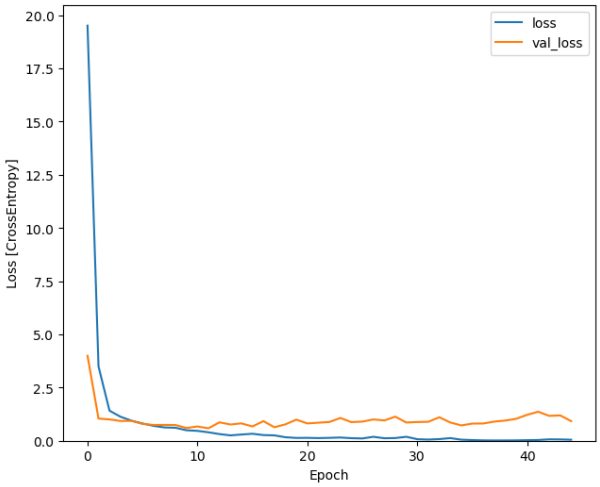
سپس به تعداد epoch 45 برای فیت شدن در مدل تعریف کرده و همچنان مقدار patience را 3 تعریف می‌کنیم تا اگر پس از ۳ ایپاک ، درصد loss کاهش نیافت، early-stopping رخ دهد.

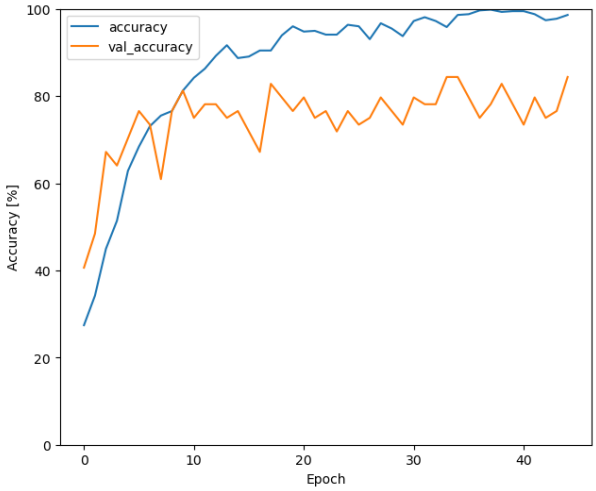
```
#Defining number of epochs to be worked on
EPOCHS = 45

#Model fitting step
history = model.fit(
    train_features_ds,
    validation_data=val_features_ds,
    epochs=EPOCHS
#     ,callbacks=tf.keras.callbacks.EarlyStopping(verbose=1, patience=5),
)
```

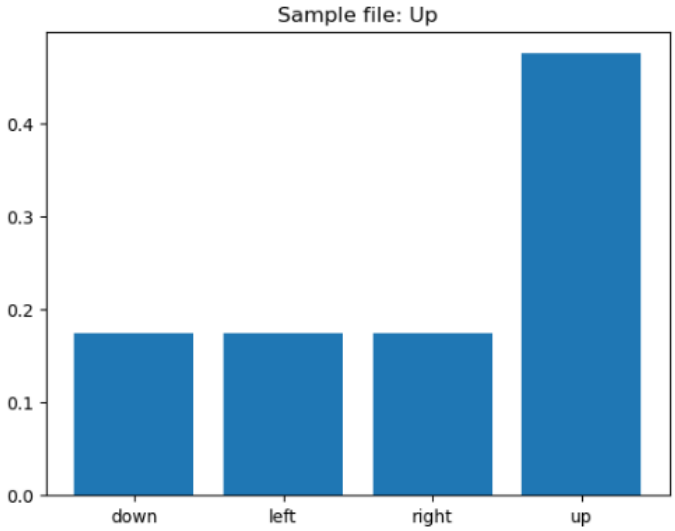
```
Epoch 1/45
18/18 [==============================] - 2s 81ms/step - loss: 19.5135 –
accuracy: 0.2743 - val_loss: 3.9938 - val_accuracy: 0.4062
Epoch 2/45
18/18 [==============================] - 0s 25ms/step - loss: 3.4907 –
 accuracy: 0.3420 - val_loss: 1.0422 - val_accuracy: 0.4844
Epoch 3/45
18/18 [==============================] - 0s 27ms/step - loss: 1.4133 –
 accuracy: 0.4497 - val_loss: 1.0026 - val_accuracy: 0.6719
Epoch 4/45
18/18 [==============================] - 0s 25ms/step - loss: 1.1277 –
 accuracy: 0.5139 - val_loss: 0.9295 - val_accuracy: 0.6406
Epoch 5/45
18/18 [==============================] - 0s 25ms/step - loss: 0.9432 –
 accuracy: 0.6285 - val_loss: 0.9347 - val_accuracy: 0.7031
Epoch 6/45
18/18 [==============================] - 0s 25ms/step - loss: 0.8125 –
 accuracy: 0.6840 - val_loss: 0.7922 - val_accuracy: 0.7656
Epoch 7/45
18/18 [==============================] - 0s 25ms/step - loss: 0.7004 –
 accuracy: 0.7309 - val_loss: 0.7421 - val_accuracy: 0.7344
Epoch 8/45
18/18 [==============================] - 0s 25ms/step - loss: 0.6253 –
 accuracy: 0.7552 - val_loss: 0.7445 - val_accuracy: 0.6094
Epoch 9/45
18/18 [==============================] - 0s 25ms/step - loss: 0.6091 –
 accuracy: 0.7656 - val_loss: 0.7380 - val_accuracy: 0.7656
Epoch 10/45
18/18 [==============================] - 0s 25ms/step - loss: 0.4927 –
 accuracy: 0.8125 - val_loss: 0.5997 - val_accuracy: 0.8125
Epoch 11/45
```

```
18/18 [==============================] - 0s 26ms/step - loss: 0.4643 -
 accuracy: 0.8420 - val_loss: 0.6699 - val_accuracy: 0.7500
Epoch 12/45
18/18 [==============================] - 0s 26ms/step - loss: 0.3964 -
 accuracy: 0.8628 - val_loss: 0.5852 - val_accuracy: 0.7812
Epoch 13/45
...
Epoch 44/45
18/18 [==============================] - 1s 30ms/step - loss: 0.0620 -
 accuracy: 0.9774 - val_loss: 1.1906 - val_accuracy: 0.7656
Epoch 45/45
18/18 [==============================] - 1s 28ms/step - loss: 0.0480 -

 accuracy: 0.9861 - val_loss: 0.9210 - val_accuracy: 0.8438
```

سپس نتایج loss و accuracy را رسم می‌کنیم :







مراحل بعدی این بخش کاملا مشابه با روش اول می‌باشد.

بخش سوم :

مراحل این بخش نیز کاملا مشابه با روش اول می‌باشد به این صورت که تابعی تعریف می‌کنیم تا صوت real-time از میکروفون دریافت کرده و مدل را روی آن اعمال کند.

نتایج نهایی به این صورت می‌باشد :

```
Predicted label: down
Predicted label: down
Predicted label: left
Predicted label: down
Predicted label: down
Predicted label: down
Predicted label: down
Predicted label: left
Predicted label: right
Predicted label: right
Predicted label: right
Predicted label: down
Interrupted
```

: Adam Optimization

الگوریتم Adam (مخفف Adaptive Moment Estimation) یکی از الگوریتم‌های بهینه‌سازی محبوب برای آموزش شبکه‌های عصبی است. این الگوریتم یک الگوریتم کاهش گرادیان تصادفی است که از میانگین متحرک گرادیان و میانگین متحرک گرادیان مربعی برای تنظیم نرخ یادگیری استفاده می‌کند.

در بهینه‌سازی Adam، نرخ یادگیری برای هر وزن شبکه (یا پارامتر) ثابت می‌ماند و به صورت جداگانه در حین بهینه‌سازی تنظیم می‌شود. این الگوریتم اولین و دومین لحظات گرادیان را محاسبه می‌کند و از آنها برای به‌روزرسانی وزن‌ها استفاده می‌کند.

مزایای اصلی بهینه‌سازی Adam شامل سرعت همگرایی سریع، مقاومت در برابر گرادیان‌های نویزی و قابلیت کنترل گرادیان‌های پراکنده است. به دلیل کارآیی و اثربخشی آن، این الگوریتم در بسیاری از کاربردهای یادگیری عمیق استفاده می‌شود.

الگوریتم Adam در بسیاری از چارچوب‌های یادگیری عمیق محبوب مانند TensorFlow، PyTorch و Keras پیاده‌سازی شده است.

: RELU VS. tanh

ReLU (Rectified Linear Unit) و Tanh (Hyperbolic Tangent) هر دو جز activation functionهای محبوب در شبکه‌های عصبی هستند.

اصلی‌ترین تفاوت بین این دو تابع، محدوده خروجی آنها است. ReLU یک تابع خطی قطعه‌ای است که برای مقادیر ورودی منفی خروجی ۰ و برای مقادیر ورودی مثبت ، خود ورودی را به عنوان خروجی می‌دهد. این به این معنی است که محدوده خروجی ReLU از [۰، بی‌نهایت) است. در مقابل، Tanh یک تابع غیرخطی است که مقادیر خروجی آن در محدوده [-۱، ۱] قرار دارند و در مقدار صفر نقطه انحنایی دارد.

تفاوت دیگری که وجود دارد، سادگی تابع ReLU نسبت به Tanh است که باعث افزایش کارایی محاسباتی آن می‌شود. همچنین، به دلیل کارایی بالای آن در بسیاری از کاربردهای یادگیری عمیق، ReLU مورد استفاده قرار می‌گیرد. با این حال، تابع ReLU ممکن است با مشکل "dying ReLU" مواجه شود که در آن برخی نورون‌ها ممکن است برای همه مقادیر ورودی ، خروجی صفر دهند که منجر به کاهش عملکرد کلی شبکه می‌شود.

از سوی دیگر، Tanh یک تابع صاف و مشتق‌پذیر است که به عنوان یک انتخاب خوب برای الگوریتم‌های بهینه‌سازی مبتنی بر گرادیان استفاده می‌شود. این تابع اغلب در شبکه‌های عصبی بازگشتی (RNN) و مدل‌های تولیدی مورد استفاده قرار می‌گیرد که محدوده خروجی [-۱، ۱] مطلوب است.

به طور خلاصه، در حالی که هر دو ReLU و Tanh توابع فعال‌سازی موثری هستند، تفاوت‌های آنها در محدوده خروجی و پیچیدگی محاسباتی آنها باعث می‌شود که هر یک برای نوع معماری و کاربردهای مختلف شبکه‌های عصبی مناسب باشند.

اما در این مدل‌های طراحی شده ، با اعمال tanh به هر لایه نتایج نامطلوبی نسبت به تابع Relu بدست آمد.

MLP vs. CNN :

Convolutional Neural Networks (CNN) و Multilayer Perceptrons (MLP) هر دو معماری شبکه‌های عصبی محبوبی هستند که در بسیاری از کاربردهای یادگیری ماشین، از جمله تشخیص دستورات صوتی، مورد استفاده قرار می‌گیرند.

CNN به خصوص برای پردازش تصویر و سیگنال مناسب است چراکه داده ورودی آنها ساختار شبکه‌ای مانند شبکه‌های پیکسل‌های تصویر یا اسپکتروگرام صوتی می‌باشند. این شبکه‌ها از لایه‌های Convolutional برای استخراج ویژگی‌ها از داده ورودی و از لایه‌های پولینگ برای کاهش ابعاد ویژگی‌های استخراج شده استفاده می‌کنند. این به آنها امکان می‌دهد تا نمایش سلسله مراتبی از داده ورودی را یاد بگیرند که برای تشخیص الگوها در تصاویر و سیگنال‌های صوتی موثر است.

در مقابل، Multilayer Perceptrons شبکه‌های عصبی کلی‌تر هستند که برای وظایف مختلف، از جمله تشخیص دستورات صوتی، مورد استفاده قرار می‌گیرند. Multilayer Perceptrons از چندین لایه از نورون‌های کاملاً متصل تشکیل شده‌اند و هر نورون با همه نورون‌های لایه قبلی و لایه بعدی متصل است. این شبکه‌ها معمولاً برای وظایفی استفاده می‌شوند که داده ورودی آنها ساختار برداری یا شبکه‌ای مانند داده‌های جدولی یا متنی دارند.

در مورد تشخیص دستورات صوتی، CNN در سال‌های اخیر عملکرد خوبی را از خود نشان داده‌اند. آنها می‌توانند به صورت مستقیم از سیگنال صوتی استفاده کنند یا ویژگی‌های موجود در اسپکتروگرام صوتی را استخراج کنند. با استفاده از Convolutional و پولینگ، می‌توانند الگوهای پیچیده را در سیگنال صوتی یاد بگیرند، حتی با ورودی های نویزی یا مشوش.

به عنوان مقابل، MLP می‌تواند برای تشخیص دستورات صوتی از طریق پردازش سیگنال صوتی به عنوان یک دنباله از ویژگی‌های استخراج شده از سیگنال صوتی، مانند Mel Frequency Cepstral Coefficients (MFCC) استفاده شوند. همچنین می‌توانند الگوهای موجود در دنباله ویژگی‌ها را یاد بگیرند که در تشخیص دستورات صوتی موثر هستند.

به طور خلاصه، هر دو شبکه می‌توانند برای تشخیص دستورات صوتی استفاده شوند، اما شبکه‌های CNN برای پردازش مستقیم سیگنال صوتی و استخراج ویژگی‌ها از اسپکتروگرام صوتی مناسب تر هستند، در حالی که MLP برای پردازش دنباله ویژگی‌های استخراج شده از سیگنال صوتی مناسب هستند. انتخاب معماری مناسب بستگی به نیازهای خاص وظیفه و ماهیت داده ورودی دارد.