Neural Network

Project Report

Dr. Monire Abdoos

writer:Mahdie Dolatabadi

Student Id:400443077

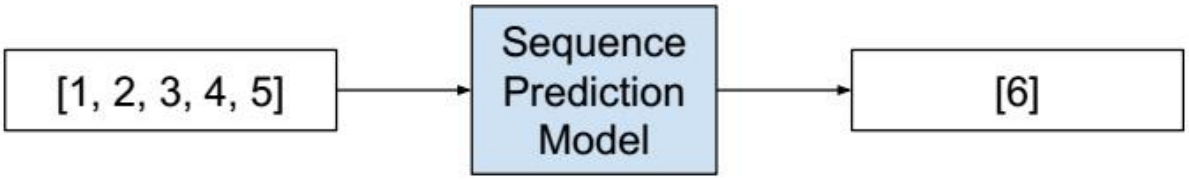
Faculty of Computer Engineering

مقدمه:

یکی از انواع مسائلی که ما در دنیای واقعی با آن‌ها برخورد می‌کنیم پیش‌بینی کردن داده‌های دارای توالی یا sequence predictions هستند. این دسته مسائل از جمله مسائلی هستند که در دسته supervised learning قرار می‌گیرند و در توالی داده‌های ورودی آن‌ها نظمی وجود دارد که باید مورد توجه شبکه یادگیری قرارگیرد و بهتر است این توالی هنگام آموزش مدل حفظ شود. به این دست از مسائل sequence prediction problem می‌گویند. این نوع مسائل معمولا در این چهار دسته قرار می‌گیرند:

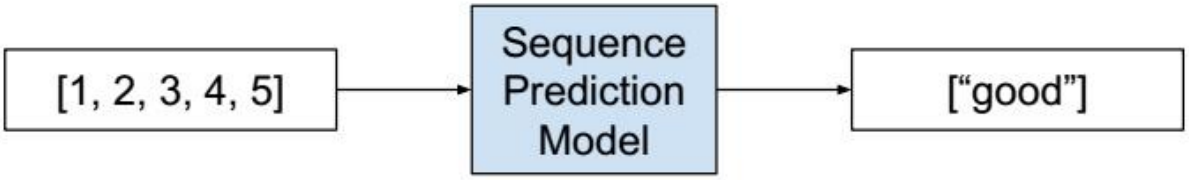
i. Sequence Prediction

مسائلی هستند که داده‌های ورودی و خروجی در آن یکی هستند. یعنی مدل ما یک توالی از داده را دریافت می‌کند و پیش‌بینی می‌کند که داده بعدی در آن توالی چیست. به طور مثال ورودی ما [1,2,3,4,5] به مدل داده می‌شود و شبکه تشخیص می‌دهد عدد بعدی 6 است.



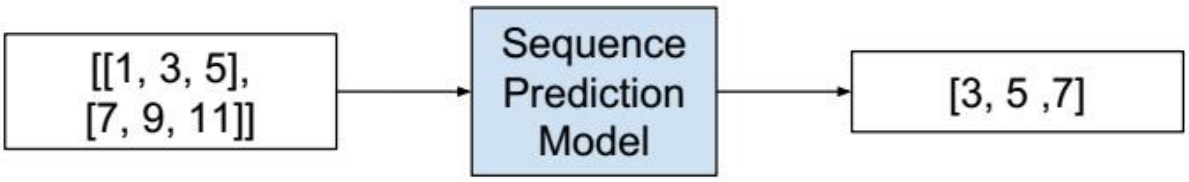
ii. Sequence Classification

در این دسته مدل یک توالی از داده‌ها را دریافت می‌کند و تشخیص می‌دهد که این توالی مربوط به یک کلاس خاص است. مثلا اگر ورودی شبکه [1,2,3,4,5] باشد خروجی 'good' است.



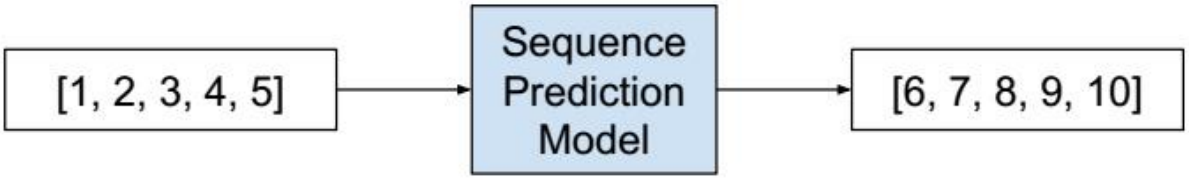
iii. Sequence Generation

این نوع مسائل داده ورودی را دریافت می‌کند و از بین آن‌ها یک توالی تولید می‌کند. مانند شکل زیر:



iv. Sequence Prediction

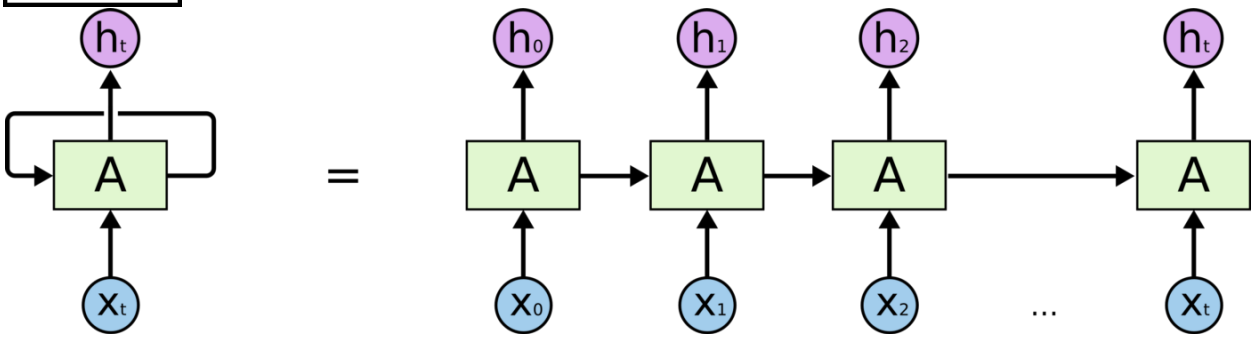
در این دسته مدل توالی را دریافت کرده و یک توالی پیش‌بینی می‌کند مانند زیر:



کاربرد MLPها برای پیش‌بینی توالی مستلزم آن است که توالی ورودی به دنباله‌های کوچک‌تر دارای همپوشانی تقسیم شود و به شبکه نشان داده می‌شود تا یک پیش‌بینی ایجاد شود. مراحل زمانی دنباله ورودی به ویژگی های ورودی شبکه تبدیل می شوند. دنباله‌های بعدی با هم تداخل دارند تا پنجره‌ای را شبیه‌سازی کنند که در امتداد دنباله لغزنده می‌شود تا خروجی مورد نیاز تولید شود. این می تواند روی برخی مشکلات به خوبی کار کند، اما چند محدودیت حیاتی دارد:

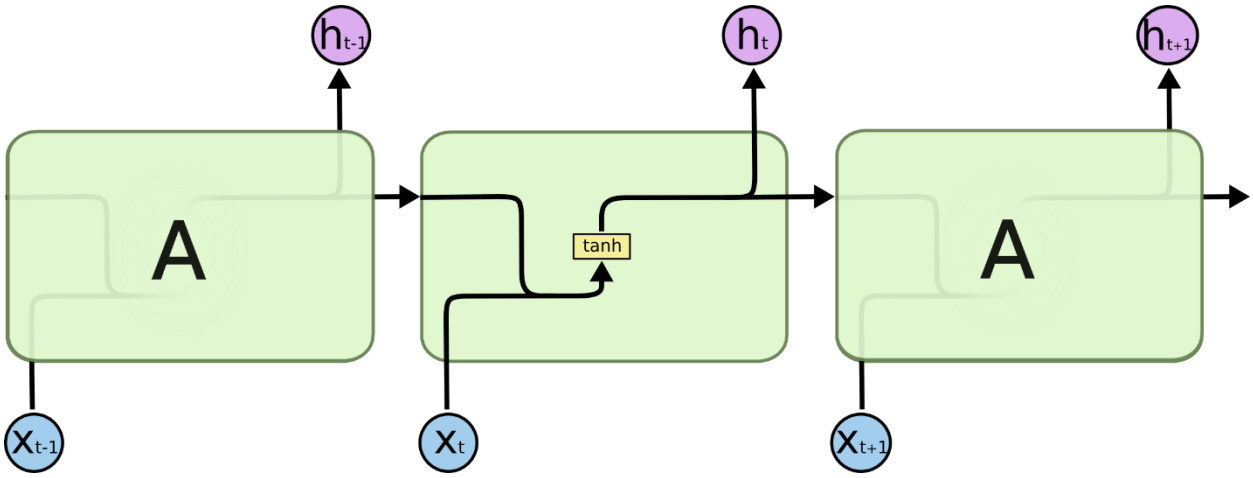
- ناآگاهی از ساختار زمانی : مراحل زمانی به عنوان ویژگی‌های ورودی مدل‌سازی می‌شوند، به این معنی که شبکه هیچ مدیریت یا درک صریحی از ساختار زمانی یا نظم بین مشاهدات ندارد.
- تقسیم‌بندی شلوغ و اصطلاحا کثیف : برای مسائلی که نیاز به مدلسازی چندین توالی در ورودی به طور موازی دارد تعداد ویژگی‌های ورودی به عنوان معیاری از اندازه پنجره کشویی به طور بی‌رویه‌ای افزایش می‌یابد.
- ورودی و خروجی ما اندازه ثابت دارد و قابل تغییر نیست.

به دلیل این محدودیت ها به سراغ شبکه‌های عصبی بازگشتی می‌رویم و از بین این شبکه‌ها مدلی به نام LSTM را اجرا می‌کنیم. شبکه‌های عصبی بازگشتی یا اصطلاحا RNN نوع خاصی از شبکه‌های عصبی هستند که مشکل توالی را حل میکنند. تفاوت این شبکه با MLP در این است که MLP بر اساس ورودی فعلی تصمیم می‌گیرد اما RNN براساس ورودی فعلی و ورودی‌های گذشته تصمیم می‌گیردکه چه خروجی بدهد.

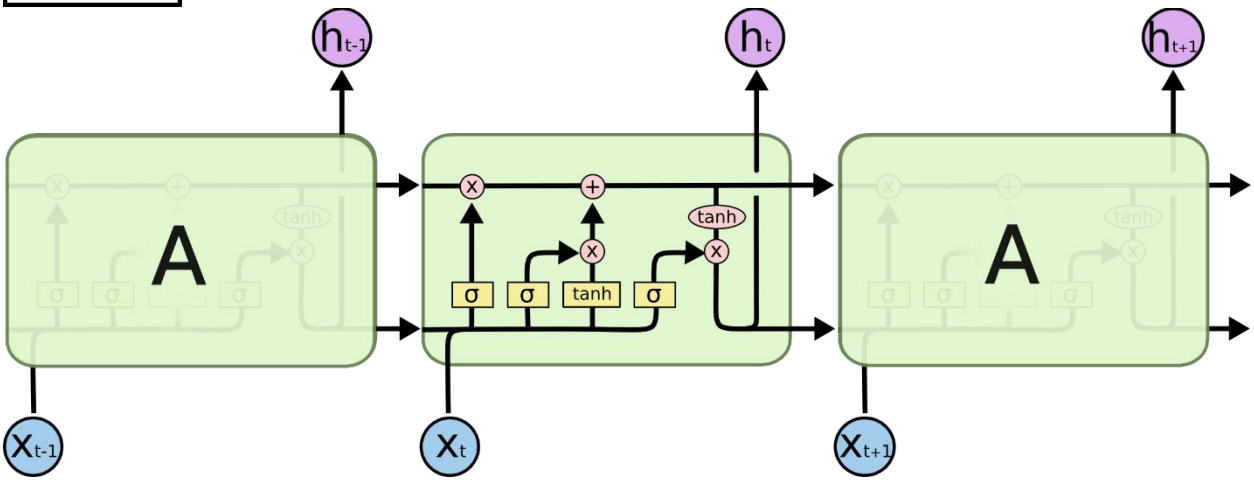


خوب توی برخی موارد ما نیاز داریم که ورودی‌های گذشته نزدیک رو یاد بگیریم که در اینجا خیلی به مشکل نمی‌خوریم. اما در مواردی که خروجی ما به ورودی در گذشته‌های دور نیازمند است ممکن است شبکه RNN معمولی دچار فراموشی بشوند یا در اونها گذشته‌های دور تاثیر کمی داشته‌باشند. LSTM این مشکل را حل می‌کند.

تمامی شبکه های عصبی بازگشتی به شکل زنجیره‌ای تکرارشونده از واحدهای شبکه عصبی هستند که در نوع استاندارد آنها این واحدها ساختار ساده‌ای دارند. مثلا شامل یکلایه تانژانت هایپربولیک هستند.



در LSTM واحد تکرارشونده ساختار متفاوتی دارد و به جای داشتن یک لایه، چهار لایه عصبی دارد.



عنصر اصلی این واحدها سلول حالت است که درواقع یک خط افقی است که بالای شکل قرار دارد.LSTM این توانایی را دارد که اطلاعات جدید را به سلول حالت اضافه یا کم کند. این کار توسط ساختارهایی به نام دروازه انجام می‌گیرد. دروازه‌ها یک لایه سیگموییدی هستند که خروجی آن‌ها بین صفر و یک است و تعیین می‌کنند چند درصد از ورودی در سلول حالت تاثیر می‌گذارند.

در تصویر بالا اولین دروازه از سمت چپ مشخص می‌کند که چقدر از گذشته را در سلول فعلی و سلول‌های آینده موثر باشد. دومین تصمیم میگیرد که چه اطلاعاتی به سلول حالت اضافه شود و چقدر در سلول حالت تاثیرگذار باشد. و نهایتا آخرین دروازه تصمیم می‌گیرد چه اطلاعاتی را به خروجی ببریم. بدین صورت می‌توانیم تصمیم بگیریم هر سلول چقدر در آموزش کل شبکه ما موثر باشد.

پروژه ما:

تشخیص فعالیت‌های انسانی از داده‌های حسگر محیطی پیوسته

داده‌ها:

این مجموعه داده‌های محیطی جمع‌آوری‌شده در خانه‌های دارای ساکنان داوطلب را نشان می‌دهد. داده ها به طور مداوم جمع آوری می شوند در حالی که ساکنان روال عادی خود را انجام می دهند.

سنسورهای حرکتی PIR محیطی، حسگرهای درب/دما، و سنسورهای سوئیچ نور در سرتاسر خانه داوطلب قرار می‌گیرند. حسگرها در مکان‌هایی در سراسر خانه قرار می‌گیرند که مربوط به فعالیت‌های خاصی از زندگی روزمره است که می‌خواهیم آن‌ها را ثبت کنیم.

وظیفه مدل ما پیش بینی فعالیتی هست که در خانه هوشمند رخ می‌دهد و توسط حسگرهای محیط مشاهده می‌شود.

توضیح کامل ویژگی‌های دادگان:

- lastSensorEventHours
- lastSensorEventSeconds
- lastSensorDayOfWeek

اینها زمانی هستند که در طی روز سنسور ها ثبت وقایع کرده‌اند.

- windowDuration

مدت زمانی که این سنسور 30 رخداد را ثبت کند

- timeSinceLastSensorEvent

زمان برحسب ثانیه از آخرین بار که سنسوری ثبت کرده تا به حال

- prevDominantSensor1
- prevDominantSensor2

شناسه دو سنسوری که یک حرکت را ثبت کرده‌اند.

- lastSensorID

شناسه آخرین سنسوری که در پنجره رویدادی ثبت کرده‌است.

- lastSensorLocation

محل آخرین سنسوری که رخدادی را ثبت کرده‌است.

- lastMotionLocation

محل آخرین رخدادی که در پنجره ثبت شده آست.

- Complexity

پیچیدگی یا اندازه گیری آنتروپی در شمارش حسگرها.

- activityChange

تغییرات نوع فعالیت بین دو نیمه یک پنجره

- areaTransitions

تعداد انتقال بین مکان های اصلی حسگر در پنجره.

- numDistinctSensors

تعداد حسگرهای متمایز در پنجره، در حال حاضر روی همیشه 0 تنظیم شده است.

- sensorCount-Bathroom
- sensorCount-Bedroom
- sensorCount-Chair
- sensorCount-DiningRoom
- sensorCount-Hall
- sensorCount-Ignore
- sensorCount-Kitchen
- sensorCount-LivingRoom
- sensorCount-Office
- sensorCount-OutsideDoor
- sensorCount-WorkArea

تعداد وزنی حسگرها، بیشترین مقدار یک می‌گیرد

- sensorElTime-Bathroom
- sensorElTime-Bedroom
- sensorElTime-Chair
- sensorElTime-DiningRoom
- sensorElTime-Hall
- sensorElTime-Ignore
- sensorElTime-Kitchen
- sensorElTime-LivingRoom
- sensorElTime-Office
- sensorElTime-OutsideDoor
- sensorElTime-WorkArea

برای این دست داده‌ها تعداد ثانیه از آخرین باری که این سنسور در مکان خاص خود (مانند آشپزخانه و صندلی و ...) دیده شده است، حداکثر تا 86400 قابل ثبت است.

بخش اول پروژه:

ابتدا کتابخانه‌های لازم و ابتدایی را فراخوانی می‌کنیم:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import sys
#np.set_printoptions(threshold=sys.maxsize)
```

سپس دیتافریم مورد نظر را وارد سیستم می‌کنیم و نوع هرکدام از فیچر هارا بررسی می‌کنیم.

```
data =  pd.read_csv('/content/drive/MyDrive/csh111.ann.features.csv')
```

```
lastSensorEventHours        float64
lastSensorEventSeconds      float64
lastSensorDayOfWeek         float64
windowDuration              float64
timeSinceLastSensorEvent    float64
prevDominantSensor1         float64
prevDominantSensor2         float64
lastSensorID                float64
lastSensorLocation          float64
lastMotionLocation          float64
complexity                  float64
activityChange              float64
areaTransitions             float64
numDistinctSensors          float64
sensorCount-Bathroom        float64
sensorCount-Bedroom         float64
sensorCount-Chair           float64
sensorCount-DiningRoom      float64
sensorCount-Hall            float64
sensorCount-Ignore          float64
sensorCount-Kitchen         float64
sensorCount-LivingRoom      float64
sensorCount-Office          float64
sensorCount-OutsideDoor     float64
sensorCount-WorkArea        float64
sensorElTime-Bathroom       float64
sensorElTime-Bedroom        float64
sensorElTime-Chair          float64
sensorElTime-DiningRoom     float64
sensorElTime-Hall           float64
sensorElTime-Ignore         float64
sensorElTime-Kitchen        float64
sensorElTime-LivingRoom     float64
sensorElTime-Office         float64
sensorElTime-OutsideDoor    float64
sensorElTime-WorkArea       float64
activity                     object
dtype: object
```

می‌بنیم که تمامی داده‌ها از نوع float64هستند . پس کافیست آنهارا نرمال کنیم. همچنین خروجی به صورت کلاس است. بنابر این ابتدا هر کلاس را به یک عدد خاص تبدیل می‌کنیم و آن اعداد را تبدیل به کد های باینری (one hot) می‌کنیم. قبل از این تبدیلات ابتدا داده‌های ورودی و خروجی را جدا میکنیم. از ورودی سه ستون اول را به دلیل نامربوط بودن(زمان ثبت سنسورها) حذف کرده‌ایم.

```
data_in = data.iloc[:,3:36].values
data_out = data.iloc[:,36].values
```

```
data_out
```

```
array(['Sleep', 'Sleep', 'Sleep', ..., 'Sleep', 'Sleep', 'Other_Activity'],
      dtype=object)
```

این تابع برای تبدیل خروجی است که ابتدا labelEncoder و سپس oneHotEncoder را اعمال می‌کند.

```
from numpy import array
from numpy import argmax
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
def encod(data):
  values = array(data)
  # integer encode
  label_encoder = LabelEncoder()
  integer_encoded = label_encoder.fit_transform(values)
  # binary encode
  onehot_encoder = OneHotEncoder(sparse=False)
  integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
  onehot_encoded = onehot_encoder.fit_transform(integer_encoded)
  return onehot_encoded
```

و اینهم بر تمام ویژگی‌های ورودی (یعنی بر روی تمامی ستون‌ها جز خروجی اعمال می‌شود)

```
from pandas import Series
from sklearn.preprocessing import MinMaxScaler
def scale(data):
  # define contrived series
  #series = Series(data)
  # prepare data for normalization
  #values = data.values
  values = data.reshape((len(data), 1))
  # train the normalization
  scaler = MinMaxScaler(feature_range=(0, 1))
  scaler = scaler.fit(values)
  # normalize the dataset and print
  normalized = scaler.transform(values)
  return normalized
```

در زیر اعمال شده‌است.

```
out = encod(data_out)
```

```
for col in range(0,33):
  scaler = scale(data_in[:,col])
  #stan = standard(scaler.flatten())
  data_in[:,col:col+1] = scaler
```

```
data_in.shape
```

```
(351324, 33)
```

در اینجا ورودی نرمال‌شده را به صورت یک سری درآورده‌ایم.

```
X = []
y = []
```

```
for row in range(50,len(data_in[:,0])):
  X.append(data_in[row-50: row,:])
  y.append(out[row])
```

```
len(y)
```

```
351274
```

و سپس داده آموزش و تست را جدا کرده‌ایم. از آنجایی که داده‌ها به صورت لیست ذخیره‌شده‌اند آن‌ها را تبدیل آرایه می‌کنیم. شبکه آرایه را در ورودی می‌پذیرد.

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
```

```
type(x_train)
```

```
list
```

```
x_train, y_train = np.array(x_train), np.array(y_train)
```

```
type(y_train)
```

```
numpy.ndarray
```

سپس شکل داده‌ها را به صورتی تبدیل می‌کنیم که در ورودی شبکه قابل پذیرش باشد و کتابخانه‌های لازم را برای شبکه فراخوانی می‌کنیم.

```
x_train = np.reshape(x_train,(x_train.shape[0], x_train.shape[1], 33))
```

```
#y_train = np.reshape(y_train,(y_train.shape[0], 35))
```

```
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import Dropout
from tensorflow import keras
from tensorflow.keras import layers
```

شبکه دارای چهار لایه است. شکل ورودی را تنها لازم است در لایه اول ثبت شود. هر لایه 50 نورون دارد (منظور از نورون در شبکه LSTM در بالا توضیح داده‌شده‌است.) به جز لایه آخر که باید هم سایز تعداد کلاس‌های خروجی ما باشد. در لایه خروجی برای تبدیل کردن کل نورون ها به کلاسبندی‌های oneHot از تابع فعالسازی softmax استفاده‌شده‌است. از optimizer = ‘adam’ و loss=’categorical_crossentropy’ برای خروجی‌های چند کلاسه استفاده می‌کنیم. شبکه را بر روی داده با تعداد epoch 10 آموزش دادیم که به دقت موزد نظر رسید.

```
model = Sequential()
#First layer
model.add(LSTM(units=50,return_sequences=True, input_shape=( x_train.shape[1],33) ))

#LSTM Second layer
model.add(LSTM(units=50, return_sequences=True))

#LSTM Third layer
model.add(LSTM(units=50))

#model.add(layers.Activation('softmax'))
#Dense layer
model.add(Dense(units=35, activation='softmax'))
```

```
#compile
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
#fit
model.fit(x_train, y_train, epochs=10, batch_size=64)
```

```
Epoch 1/10
4391/4391 [==============================] - 166s 38ms/step - loss: 0.9153 - accuracy: 0.6988
Epoch 2/10
Epoch 1/10
4391/4391 [==============================] - 166s 38ms/step - loss: 0.9153 - accuracy: 0.6988
Epoch 2/10
4391/4391 [==============================] - 166s 38ms/step - loss: 0.8022 - accuracy: 0.7252
Epoch 3/10
4391/4391 [==============================] - 166s 38ms/step - loss: 0.7238 - accuracy: 0.7459
Epoch 4/10
4391/4391 [==============================] - 168s 38ms/step - loss: 0.6568 - accuracy: 0.7668
Epoch 5/10
4391/4391 [==============================] - 168s 38ms/step - loss: 0.5993 - accuracy: 0.7860
Epoch 6/10
4391/4391 [==============================] - 167s 38ms/step - loss: 0.5465 - accuracy: 0.8046
Epoch 7/10
4391/4391 [==============================] - 166s 38ms/step - loss: 0.5024 - accuracy: 0.8213
Epoch 8/10
4391/4391 [==============================] - 165s 38ms/step - loss: 0.4629 - accuracy: 0.8358
Epoch 9/10
4391/4391 [==============================] - 166s 38ms/step - loss: 0.4282 - accuracy: 0.8484
Epoch 10/10
4391/4391 [==============================] - 166s 38ms/step - loss: 0.4002 - accuracy: 0.8595
<keras.callbacks.History at 0x7f358b187d10>
```

ونهایتا داده تست را دقت گرفتیم که باز هم دقت مورد نیاز سوال برآورده‌شد.

```
x_test, y_test = np.array(x_test), np.array(y_test)
```

```
x_test = np.reshape(x_test,(x_test.shape[0], x_test.shape[1], 33))
```

```
y_test.shape
```

```
(70255, 35)
```

```
x_test.shape
```

```
(70255, 50, 33)
```

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print('Test Loss: {}'.format(test_loss))
print('Test Accuracy: {}'.format(test_acc))
```

```
2196/2196 [==============================] - 36s 15ms/step - loss: 0.4035 - accuracy: 0.8604
Test Loss: 0.40352535247802734
Test Accuracy: 0.8603515625
```

یکبار دیگر برای اطمینان تعداد epochها را اضافه کردیم تا دقت لازم به طور کامل بدست آید.

```
#fit
model.fit(x_train, y_train, epochs=20, batch_size=128, validation_split=0.2)
```

```
Epoch 1/20
1757/1757 [==============================] - 31s 13ms/step - loss: 1.2530 - accuracy: 0.6281 - val_loss: 1.0012 - val_accuracy: 0.6793
Epoch 2/20
1757/1757 [==============================] - 21s 12ms/step - loss: 0.9567 - accuracy: 0.6894 - val_loss: 0.8986 - val_accuracy: 0.7025
Epoch 3/20
1757/1757 [==============================] - 21s 12ms/step - loss: 0.8730 - accuracy: 0.7091 - val_loss: 0.8746 - val_accuracy: 0.7088
Epoch 4/20
1757/1757 [==============================] - 23s 13ms/step - loss: 0.8139 - accuracy: 0.7237 - val_loss: 0.8050 - val_accuracy: 0.7278
Epoch 5/20
Epoch 15/20
1757/1757 [==============================] - 21s 12ms/step - loss: 0.4413 - accuracy: 0.8451 - val_loss: 0.4497 - val_accuracy: 0.8441
Epoch 16/20
1757/1757 [==============================] - 21s 12ms/step - loss: 0.4242 - accuracy: 0.8514 - val_loss: 0.4394 - val_accuracy: 0.8475
Epoch 17/20
1757/1757 [==============================] - 21s 12ms/step - loss: 0.4022 - accuracy: 0.8593 - val_loss: 0.4305 - val_accuracy: 0.8512
Epoch 18/20
1757/1757 [==============================] - 21s 12ms/step - loss: 0.3943 - accuracy: 0.8626 - val_loss: 0.4044 - val_accuracy: 0.8600
Epoch 19/20
1757/1757 [==============================] - 21s 12ms/step - loss: 0.3714 - accuracy: 0.8705 - val_loss: 0.4047 - val_accuracy: 0.8602
Epoch 20/20
1757/1757 [==============================] - 21s 12ms/step - loss: 0.3635 - accuracy: 0.8739 - val_loss: 0.3926 - val_accuracy: 0.8656
<keras.callbacks.History at 0x7fabce200cd0>
```

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print('Test Loss: {}'.format(test_loss))
print('Test Accuracy: {}'.format(test_acc))
```

```
2196/2196 [==============================] - 12s 5ms/step - loss: 0.3950 - accuracy: 0.8632
Test Loss: 0.39504268765449524
Test Accuracy: 0.8632410764694214
```

قسمت دوم پروژه:

در این قسمت تمامی مراحل قسمت گذشته است با این تفاوت که ابتدا دادگان را به یک شبکه CNN می‌دهیم تا از آن به خوبی ویژگی‌های لازم را استخراج کرده و بسازدو سپس خروجی را به یک شبکه LSTM می‌دهیم.شکل ورودی در شبکه CNN باید به صورت سه بعدی باشد بنابراین طول توالی را که 50 بوده‌است به 5*10 می‌شکنیم. سپس آن را وارد شبکه می‌کنیم.همچنین برای ساخت شبکه ابتدا کتابخانه‌های لازم را از keras فراخوانی می‌کنیم.

```
[18] from keras.models import Sequential
     from keras.layers import LSTM
     from keras.layers import Dense
     from keras.layers import Dropout
     from tensorflow import keras
     from tensorflow.keras import layers
     from keras.layers import Conv1D
     from keras.layers import MaxPooling1D
     from keras.layers import Flatten
     from keras.layers import TimeDistributed
```

برای توابع فعالسازی در CNN از relu استفاده می‌کنیم. پیش از اضافه کردن لایه Dropoutیک بار دیگر شبکه را آموزش دادیم و دچار overfit شد. بنابراین برای دولایه شبکه LSTM، Dropout با نرخ0.2 گذاشته‌ایم تا از این مشکل جلوگیری کند. باقی پارامترها در مدل قبی توضیح داده‌شده اند.

```
[19] x_train = np.reshape(x_train,( x_train.shape[0], 5, 10, 33))
```

```
model = Sequential()
# after having Conv2D...
model.add(TimeDistributed(Conv1D(64, (2), activation='relu'), input_shape=( 5, 10, 33)))


model.add(TimeDistributed(Conv1D(64, (2), activation='relu')))


model.add(TimeDistributed(MaxPooling1D(pool_size=(2))))
# We need to have only one dimension per output
# to insert them to the LSTM layer - Flatten or use Pooling
model.add(TimeDistributed(Flatten()))
# previous layer gives 5 outputs, Keras will make the job
# to configure LSTM inputs shape (5, ...)
model.add(LSTM(100, return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(50, return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(50, return_sequences=False))
# and then, common Dense layers... Dropout...
# up to you
model.add(Dense(35, activation='softmax'))

model.compile('adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
model.fit(x_train, y_train, batch_size=512, epochs=200, validation_split=0.2)
```

```
Epoch 1/200
440/440 [==============================] - 37s 47ms/step - loss: 1.6571 - accuracy: 0.5407 - val_loss: 1.2293 - val_accuracy: 0.6498
Epoch 2/200
440/440 [==============================] - 19s 42ms/step - loss: 1.1396 - accuracy: 0.6590 - val_loss: 1.0308 - val_accuracy: 0.6811
Epoch 3/200
440/440 [==============================] - 19s 42ms/step - loss: 1.0117 - accuracy: 0.6799 - val_loss: 0.9567 - val_accuracy: 0.6922
Epoch 4/200
440/440 [==============================] - 19s 43ms/step - loss: 0.9384 - accuracy: 0.6942 - val_loss: 0.8838 - val_accuracy: 0.7067
Epoch 5/200
440/440 [==============================] - 18s 42ms/step - loss: 0.8883 - accuracy: 0.7044 - val_loss: 0.8562 - val_accuracy: 0.7145
Epoch 6/200
440/440 [==============================] - 18s 42ms/step - loss: 0.8508 - accuracy: 0.7131 - val_loss: 0.8272 - val_accuracy: 0.7202
Epoch 7/200
440/440 [==============================] - 19s 42ms/step - loss: 0.8168 - accuracy: 0.7216 - val_loss: 0.7900 - val_accuracy: 0.7292
Epoch 8/200
440/440 [==============================] - 19s 42ms/step - loss: 0.7895 - accuracy: 0.7290 - val_loss: 0.7646 - val_accuracy: 0.7373
Epoch 9/200
440/440 [==============================] - 19s 43ms/step - loss: 0.7611 - accuracy: 0.7362 - val_loss: 0.7309 - val_accuracy: 0.7476
Epoch 10/200
440/440 [==============================] - 19s 42ms/step - loss: 0.7311 - accuracy: 0.7447 - val_loss: 0.7518 - val_accuracy: 0.7401
Epoch 11/200
440/440 [==============================] - 19s 42ms/step - loss: 0.7068 - accuracy: 0.7527 - val_loss: 0.6876 - val_accuracy: 0.7586
```

```
model.fit(x_train, y_train, batch_size=512, epochs=200, validation_split=0.2)
```

```
Epoch 38/200
440/440 [==============================] - 19s 42ms/step - loss: 0.3103 - accuracy: 0.8905 - val_loss: 0.2926 - val_accuracy: 0.8998
Epoch 39/200
440/440 [==============================] - 19s 42ms/step - loss: 0.3030 - accuracy: 0.8940 - val_loss: 0.2869 - val_accuracy: 0.9041
Epoch 40/200
440/440 [==============================] - 19s 42ms/step - loss: 0.2994 - accuracy: 0.8956 - val_loss: 0.2822 - val_accuracy: 0.9059
Epoch 41/200
440/440 [==============================] - 19s 42ms/step - loss: 0.2887 - accuracy: 0.8994 - val_loss: 0.2711 - val_accuracy: 0.9088
Epoch 42/200
440/440 [==============================] - 19s 42ms/step - loss: 0.2875 - accuracy: 0.8997 - val_loss: 0.2844 - val_accuracy: 0.9042
Epoch 43/200
440/440 [==============================] - 18s 42ms/step - loss: 0.2818 - accuracy: 0.9020 - val_loss: 0.2612 - val_accuracy: 0.9125
Epoch 44/200
440/440 [==============================] - 19s 42ms/step - loss: 0.2764 - accuracy: 0.9036 - val_loss: 0.2599 - val_accuracy: 0.9130
Epoch 45/200
440/440 [==============================] - 19s 42ms/step - loss: 0.2691 - accuracy: 0.9064 - val_loss: 0.2582 - val_accuracy: 0.9134
Epoch 46/200
440/440 [==============================] - 18s 42ms/step - loss: 0.2672 - accuracy: 0.9069 - val_loss: 0.2752 - val_accuracy: 0.9081
Epoch 47/200
440/440 [==============================] - 19s 42ms/step - loss: 0.2658 - accuracy: 0.9068 - val_loss: 0.2831 - val_accuracy: 0.9037
Epoch 48/200
440/440 [==============================] - 19s 42ms/step - loss: 0.2538 - accuracy: 0.9115 - val_loss: 0.2538 - val_accuracy: 0.9154
```

```
model.fit(x_train, y_train, batch_size=512, epochs=200, validation_split=0.2)
```

```
440/440 [==============================] - 19s 42ms/step - loss: 0.1647 - accuracy: 0.9424 - val_loss: 0.1965 - val_accuracy: 0.9369
Epoch 96/200
440/440 [==============================] - 19s 42ms/step - loss: 0.1644 - accuracy: 0.9423 - val_loss: 0.1936 - val_accuracy: 0.9366
Epoch 97/200
440/440 [==============================] - 19s 42ms/step - loss: 0.1635 - accuracy: 0.9428 - val_loss: 0.2029 - val_accuracy: 0.9338
Epoch 98/200
440/440 [==============================] - 19s 42ms/step - loss: 0.1594 - accuracy: 0.9438 - val_loss: 0.2026 - val_accuracy: 0.9340
Epoch 99/200
440/440 [==============================] - 19s 42ms/step - loss: 0.1624 - accuracy: 0.9426 - val_loss: 0.2000 - val_accuracy: 0.9358
Epoch 100/200
440/440 [==============================] - 19s 42ms/step - loss: 0.1593 - accuracy: 0.9443 - val_loss: 0.1905 - val_accuracy: 0.9398
Epoch 101/200
440/440 [==============================] - 19s 42ms/step - loss: 0.1581 - accuracy: 0.9446 - val_loss: 0.2210 - val_accuracy: 0.9299
Epoch 102/200
440/440 [==============================] - 19s 43ms/step - loss: 0.1603 - accuracy: 0.9439 - val_loss: 0.2032 - val_accuracy: 0.9347
Epoch 103/200
440/440 [==============================] - 19s 43ms/step - loss: 0.1603 - accuracy: 0.9437 - val_loss: 0.1993 - val_accuracy: 0.9369
Epoch 104/200
440/440 [==============================] - 19s 42ms/step - loss: 0.1547 - accuracy: 0.9455 - val_loss: 0.1920 - val_accuracy: 0.9378
Epoch 105/200
440/440 [==============================] - 19s 42ms/step - loss: 0.1543 - accuracy: 0.9459 - val_loss: 0.2020 - val_accuracy: 0.9351
Epoch 106/200
```

```
440/440 [==============================] - 19s 42ms/step - loss: 0.1102 - accuracy: 0.9609 - val_loss: 0.1984 - val_accuracy: 0.9425
Epoch 190/200
440/440 [==============================] - 19s 43ms/step - loss: 0.1141 - accuracy: 0.9599 - val_loss: 0.2018 - val_accuracy: 0.9414
Epoch 191/200
440/440 [==============================] - 19s 42ms/step - loss: 0.1131 - accuracy: 0.9599 - val_loss: 0.2100 - val_accuracy: 0.9389
Epoch 192/200
440/440 [==============================] - 19s 42ms/step - loss: 0.1165 - accuracy: 0.9591 - val_loss: 0.1997 - val_accuracy: 0.9417
Epoch 193/200
440/440 [==============================] - 19s 42ms/step - loss: 0.1116 - accuracy: 0.9603 - val_loss: 0.2018 - val_accuracy: 0.9403
Epoch 194/200
440/440 [==============================] - 19s 43ms/step - loss: 0.1113 - accuracy: 0.9609 - val_loss: 0.1967 - val_accuracy: 0.9424
Epoch 195/200
440/440 [==============================] - 19s 42ms/step - loss: 0.1112 - accuracy: 0.9604 - val_loss: 0.2014 - val_accuracy: 0.9408
Epoch 196/200
440/440 [==============================] - 19s 42ms/step - loss: 0.1129 - accuracy: 0.9600 - val_loss: 0.1977 - val_accuracy: 0.9423
Epoch 197/200
440/440 [==============================] - 19s 42ms/step - loss: 0.1125 - accuracy: 0.9603 - val_loss: 0.1943 - val_accuracy: 0.9433
Epoch 198/200
440/440 [==============================] - 19s 42ms/step - loss: 0.1087 - accuracy: 0.9613 - val_loss: 0.2073 - val_accuracy: 0.9390
Epoch 199/200
440/440 [==============================] - 19s 42ms/step - loss: 0.1142 - accuracy: 0.9595 - val_loss: 0.2057 - val_accuracy: 0.9389
Epoch 200/200
440/440 [==============================] - 19s 42ms/step - loss: 0.1154 - accuracy: 0.9591 - val_loss: 0.1927 - val_accuracy: 0.9437
```

```
[24] x_test, y_test = np.array(x_test), np.array(y_test)
```

```
[27] x_test = np.reshape(x_test,(x_test.shape[0], 5, 10, 33))
```

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print('Test Loss: {}'.format(test_loss))
print('Test Accuracy: {}'.format(test_acc))
```

```
2196/2196 [==============================] - 13s 6ms/step - loss: 0.2013 - accuracy: 0.9420
Test Loss: 0.20130935311317444
Test Accuracy: 0.9419969916343689
```

و اما در مورد قسمت سوم پروژه:

من پارامتر های تغییر اندازه کرنل،افزایش لایه‌ها، تغییر تابع فعالسازی، تغییر نوع optimizer، تغییر تعداد epoch و batchsize را انجام دادم. تقریبا فقط با افزایش لایه LSTM و افزایش batchsize و تعداد epoch دقت بالاتر رفته‌است. که نتیجه مهایی را در آخرین آموزشم نشان داده‌ام .