به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

پروژه شماره 2

نام و نام خانوادگی	محسن فیاض - محمدرضا عظیمی
شماره دانشجویی	810100521 - 810100524
تاریخ ارسال گزارش	سه‌شنبه، 10 خرداد 1401

فهرست گزارش سوالات

سوال 1 – Stock Market Prediction

الف) شبکه را با سلولهای LSTM، GRU و RNN طراحی کنید و عملکرد هر یک را با دیگری مقایسه و تحلیل نمایید. همچنین زمان آموزش برای تعدادی ایپاک مشخص برای هر یک از شبکه ها اندازه بگیرید و دلیل تفاوت را شرح دهید.

ابتدا داده را می‌خوانیم و به شکل مناسب برای مدل می‌کنیم.

```
df_goog = pd.read_csv("GOOG.csv", sep=" ")
df_aapl = pd.read_csv("AAPL.csv", sep=" ")
df_aapl
```

	Date	High	Low	Open	Close	Volume	Adj Close
0	2010-01-04	30.642857	30.340000	30.490000	30.572857	123432400.0	26.601469
1	2010-01-05	30.798571	30.464285	30.657143	30.625713	150476200.0	26.647457
2	2010-01-06	30.747143	30.107143	30.625713	30.138571	138040000.0	26.223597
3	2010-01-07	30.285715	29.864286	30.250000	30.082857	119282800.0	26.175119
4	2010-01-08	30.285715	29.865715	30.042856	30.282858	111902700.0	26.349140
...	...	...	...	...	...	...	...
2259	2018-12-24	151.550003	146.589996	148.149994	146.830002	37169200.0	144.656540
2260	2018-12-26	157.229996	146.720001	148.300003	157.169998	58582500.0	154.843475
2261	2018-12-27	156.770004	150.070007	155.839996	156.149994	53117100.0	153.838562
2262	2018-12-28	158.520004	154.550003	157.500000	156.229996	42291400.0	153.917389
2263	2018-12-31	159.360001	156.479996	158.529999	157.740005	35003500.0	155.405045

2264 rows × 7 columns

```
data = np.column_stack((np.array(df_goog["Close"]), np.array(df_aapl["Close"])))
SEQ_LENGTH = 30
inputs, outputs = [], []
for i in tqdm(range(0, len(data) - SEQ_LENGTH - 1, 1)):
    inputs.append(data[i: i + SEQ_LENGTH])
    outputs.append(data[i + SEQ_LENGTH + 1])
x_train, x_test, y_train, y_test = train_test_split(np.array(inputs), np.array(outputs), test_size=0.1, random_state=42)
x_train.shape, y_train.shape, x_test.shape, y_test.shape
```

```
100% 2233/2233 [00:00<00:00, 67270.10it/s]
((2009, 30, 2), (2009, 2), (224, 30, 2), (224, 2))
```

شکل 1 - بارگذاری و پیش‌پردازش داده

همانطور که مشخص است داده ورودی 30 در 2 است و خروجی 2 که یعنی زمان بعدی این 30 واحد زمانی است.

کد مدل و آموزش آن در زیر آمده است.

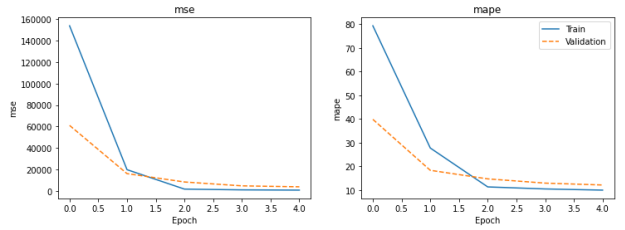
```
class RNN_Trainer:
    def __init__(self, rnn="LSTM", activation_function="relu",
                 optimizer="adam", loss_function="mse", units=512) -> None:
        self.optimizer = optimizer
        self.activation_function = activation_function
        self.loss_function = loss_function
        self.rnn = rnn
        self.model = self.build_model(units)
        self.history = None
        self.training_time = None

    def build_model(self, units):
        rnn_map = {
```

```
            "SimpleRNN": tf.keras.layers.SimpleRNN(units, dropout=0.1),
            "GRU": tf.keras.layers.GRU(units, dropout=0.1),
            "LSTM": tf.keras.layers.LSTM(units, dropout=0.1),
        }
        model = tf.keras.Sequential([
            tf.keras.layers.Input(shape=(30, 2)),
            rnn_map[self.rnn],
            tf.keras.layers.Dense(256, activation=self.activation_function),
            tf.keras.layers.Dropout(0.1),
            tf.keras.layers.Dense(128, activation=self.activation_function),
            tf.keras.layers.Dense(2)
        ])
        model.compile(
            optimizer=self.optimizer,
            loss=self.loss_function,
            metrics=['mse', 'mape']
        )
        return model

    def train(self, inputs, outputs, batch_size=8, epochs=5):
        early_stopping = tf.keras.callbacks.EarlyStopping(
            monitor='val_loss',
            verbose=1,
            patience=3,
            mode='min',
            restore_best_weights=True
        )
        self.history = self.model.fit(
            inputs, outputs,
            batch_size=batch_size,
            epochs=epochs,
            verbose=1,
            validation_split=0.1,
            callbacks=[early_stopping],
        )

    def plot_history(self):
        fig = plt.figure(figsize=(12, 4))
        metrics=['mse', 'mape']
        for n, metric in enumerate(metrics):
            plt.subplot(1, 2, n+1)
            plt.plot(self.history.epoch, self.history.history[metric], label='Train')
            plt.plot(self.history.epoch, self.history.history[f"val_{metric}"], linestyle="--", label='Validation')
            plt.xlabel('Epoch')
            plt.ylabel(metric)
            plt.title(metric)
        plt.legend()
        plt.show()

    def evaluate(self, x_test, y_test):
        [loss, mse, mape] = self.model.evaluate(x_test, y_test)
        print("Test MSE:", mse, "Test MAPE:", mape, "Test Loss:", loss)
```

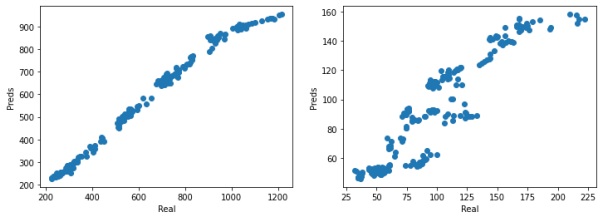
کد 1 - ساخت مدل، آموزش و ارزیابی آن

نتیجه مدل LSTM در زیر آمده است.

```
Epoch 1/5
226/226 [==============================] - 4s 12ms/step - loss: 153833.9219 - mse: 153833.9219 - mape: 79.2665 - val_loss: 60997.2422 - val_mse: 60997.2422 - val_mape: 39.8109
Epoch 2/5
226/226 [==============================] - 1s 6ms/step - loss: 19754.5000 - mse: 19754.5000 - mape: 27.7206 - val_loss: 16066.8594 - val_mse: 16066.8594 - val_mape: 18.3221
Epoch 3/5
226/226 [==============================] - 2s 7ms/step - loss: 1645.1012 - mse: 1645.1012 - mape: 11.3234 - val_loss: 8278.9902 - val_mse: 8278.9902 - val_mape: 14.7125
Epoch 4/5
226/226 [==============================] - 2s 7ms/step - loss: 933.6116 - mse: 933.6116 - mape: 10.4654 - val_loss: 4669.1133 - val_mse: 4669.1133 - val_mape: 12.9238
Epoch 5/5
226/226 [==============================] - 2s 7ms/step - loss: 689.1057 - mse: 689.1057 - mape: 9.9691 - val_loss: 3791.8113 - val_mse: 3791.8113 - val_mape: 12.1296
```




```
7/7 [==============================] - 0s 5ms/step - loss: 2762.4771 - mse: 2762.4771 - mape: 11.1052
Test MSE: 2762.47705078125 Test MAPE: 11.105165481567383 Test Loss: 2762.47705078125
```




```
trainer = RNN_Trainer(
    rnn='LSTM',
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4)
)
print(trainer.model.summary())
trainer.train(x_train, y_train, batch_size=8, epochs=5)
trainer.plot_history()
trainer.evaluate(x_test, y_test)
```

```
Model: "sequential_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_8 (LSTM)               (None, 512)               1054720

 dense_22 (Dense)            (None, 256)               131328

 dropout_7 (Dropout)         (None, 256)               0

 dense_23 (Dense)            (None, 128)               32896

 dense_24 (Dense)            (None, 2)                 258

=================================================================
Total params: 1,219,202
Trainable params: 1,219,202
Non-trainable params: 0
_________________________________________________________________
```

شکل 2 - نتیجه مدل LSTM

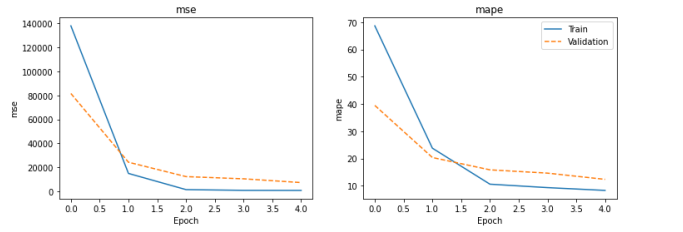
در نمودار مشخص است که لاس به خوبی کاهش یافته، و در انتها مقادیر پیش بینی شده نیز با مقادیر اصلی کوریلیشن مثبت خوبی دارند که نشان میدهد مدل به خوبی آموزش دیده است.

Test MSE: 4409.8623046875 Test MAPE: 12.683538436889648

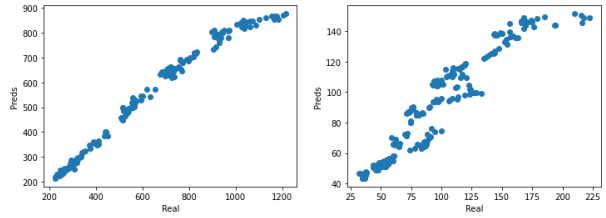
مقدار لاسها برای تست نیز در بالا آمده است. و زمان اجرا در مجموع 8 ثانیه است. تعداد پارامتر نیز 1,219,202 است.

در زیر نتیجه اجرای GRU آمده است.

```
Epoch 1/5
226/226 [==============================] - 3s 7ms/step - loss: 137751.1250 - mse: 137751.1250 - mape: 68.6611 - val_loss: 81492.5547 - val_mse: 81492.5547 - val_mape: 39.4753
Epoch 2/5
226/226 [==============================] - 1s 5ms/step - loss: 15020.2109 - mse: 15020.2109 - mape: 23.7982 - val_loss: 24339.6895 - val_mse: 24339.6895 - val_mape: 20.3805
Epoch 3/5
226/226 [==============================] - 1s 5ms/step - loss: 1557.0746 - mse: 1557.0746 - mape: 10.5930 - val_loss: 12378.5576 - val_mse: 12378.5576 - val_mape: 15.8539
Epoch 4/5
226/226 [==============================] - 1s 5ms/step - loss: 955.8207 - mse: 955.8207 - mape: 9.3321 - val_loss: 10543.4727 - val_mse: 10543.4727 - val_mape: 14.6478
Epoch 5/5
226/226 [==============================] - 1s 5ms/step - loss: 902.4995 - mse: 902.4995 - mape: 8.3118 - val_loss: 7413.6772 - val_mse: 7413.6772 - val_mape: 12.3634
```




```
7/7 [==============================] - 0s 5ms/step - loss: 5492.1455 - mse: 5492.1455 - mape: 11.3817
Test MSE: 5492.1455078125 Test MAPE: 11.381722450256348 Test Loss: 5492.1455078125
```




```
trainer = RNN_Trainer(
    rnn='GRU',
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4)
)
print(trainer.model.summary())
trainer.train(x_train, y_train, batch_size=8, epochs=5)
trainer.plot_history()
trainer.evaluate(x_test, y_test)
```

```
Model: "sequential_10"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 gru_10 (GRU)                (None, 512)               792576

 dense_30 (Dense)            (None, 256)               131328

 dropout_10 (Dropout)        (None, 256)               0

 dense_31 (Dense)            (None, 128)               32896

 dense_32 (Dense)            (None, 2)                 258

=================================================================
Total params: 957,058
Trainable params: 957,058
Non-trainable params: 0
_________________________________________________________________
```

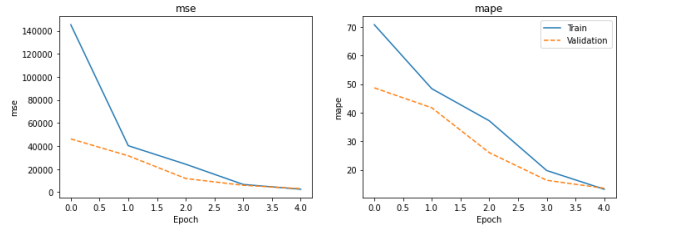
شکل 3 - نتیجه مدل GRU

Test MSE: 3080.09228515625 Test MAPE: 9.103705406188965

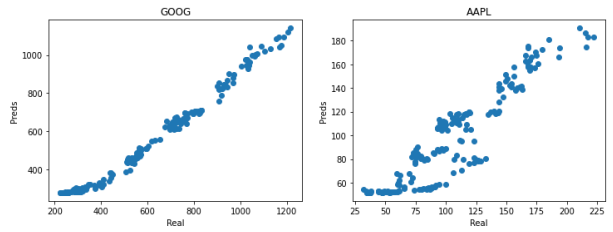
در این بخش هم مدل به خوبی آموزش دیده است و زمان اجرا در مجموع 7 ثانیه است. تعداد پارامتر نیز 957,058 است. در مورد تفاوتهای مدلها و علت این نتایج در آخر توضیح می‌دهیم.

نتیجه اجرا با RNN معمولی در زیر آمده است.

```
Epoch 1/5
226/226 [==============================] - 6s 21ms/step - loss: 145255.1875 - mse: 145255.1875 - mape: 70.7454 - val_loss: 46232.9023 - val_mse: 46232.9023 - val_mape: 48.6684
Epoch 2/5
226/226 [==============================] - 5s 20ms/step - loss: 40331.6992 - mse: 40331.6992 - mape: 48.3728 - val_loss: 31581.2559 - val_mse: 31581.2559 - val_mape: 41.7045
Epoch 3/5
226/226 [==============================] - 5s 21ms/step - loss: 24266.2148 - mse: 24266.2148 - mape: 37.1558 - val_loss: 11903.4922 - val_mse: 11903.4922 - val_mape: 26.0067
Epoch 4/5
226/226 [==============================] - 4s 20ms/step - loss: 6741.2280 - mse: 6741.2280 - mape: 19.7567 - val_loss: 6008.5605 - val_mse: 6008.5605 - val_mape: 16.3467
Epoch 5/5
226/226 [==============================] - 5s 20ms/step - loss: 2579.8728 - mse: 2579.8728 - mape: 13.2173 - val_loss: 3054.8389 - val_mse: 3054.8389 - val_mape: 13.5636
```




```
7/7 [==============================] - 0s 7ms/step - loss: 2604.2153 - mse: 2604.2153 - mape: 12.7016
Test MSE: 2604.21533203125 Test MAPE: 12.701550483703613 Test Loss: 2604.21533203125
```




```
trainer = RNN_Trainer(
    rnn='SimpleRNN',
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4)
)
print(trainer.model.summary())
trainer.train(x_train, y_train, batch_size=8, epochs=5)
trainer.plot_history()
trainer.evaluate(x_test, y_test)
```

```
Model: "sequential_12"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn_12 (SimpleRNN)   (None, 512)               263680

 dense_36 (Dense)            (None, 256)               131328

 dropout_12 (Dropout)        (None, 256)               0

 dense_37 (Dense)            (None, 128)               32896

 dense_38 (Dense)            (None, 2)                 258

=================================================================
Total params: 428,162
Trainable params: 428,162
Non-trainable params: 0
_________________________________________________________________
```

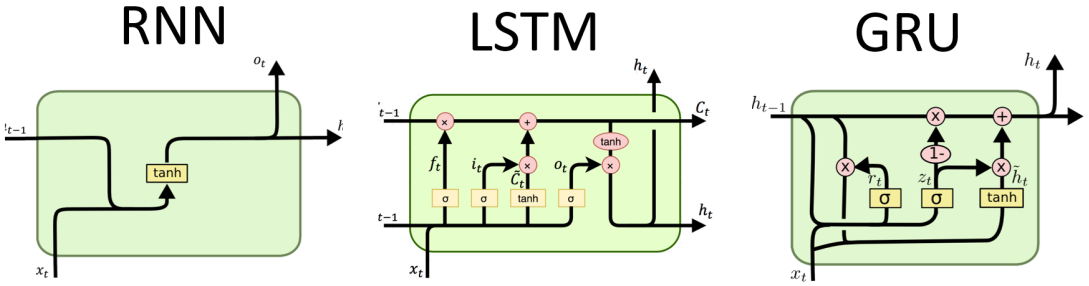
شکل 4 - نتیجه مدل SimpleRNN

Test MSE: 2009.743896484375 Test MAPE: 11.870356559753418

این بخش کمی نتایج از دو بخش قبل بدتر است و زمان اجرا در مجموع 25 ثانیه است که خیلی کندتر از بقیه است. تعداد پارامتر نیز 428,162 است.

حالا به توضیح تفاوتها و علتشان می‌پردازیم. ابتدا باید توضیح کلی در مورد این 3 مدل بدهیم.

در شکل زیر ساختار معماری RNN و GRU و LSTM آمده است.



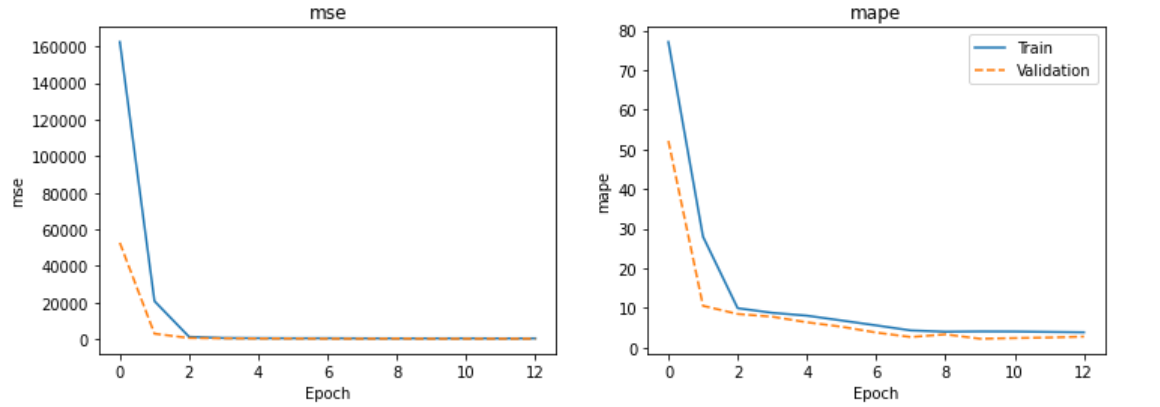
شکل 5 - ساختار مدلهای حافظه محور زمانی

معماری LSTM سه گیت دارد. گیت سمت چپ forget gate است که یک تابع سیگموید دارد و با ضرب خروجی آن که بین 0 و 1 است، مشخص می‌کند که محتویات cell state گذشته چقدر نگهداری شود. (توجه داشته باشید که این مقدار یک عدد نیست و به ازای هر بعد این عدد وجود دارد و می‌تواند بعضی ابعاد را فراموش و بعضی را نگه دارد.) گیت وسط input gate است که مشخص می‌کند چه مقدار از ورودی جدید وارد cell state شود که با ضرب سیگموید در مقدار ورودی که از tanh رد شده و در انتها جمع آن با cell state انجام می‌شود. در سمت راست output gate وجود دارد که ترکیب cell state و hidden state را مشخص می‌کند که در hidden state خروجی قرار گیرد که به زمان بعدی و همچنین به عنوان خروجی این زمان داده می‌شود.

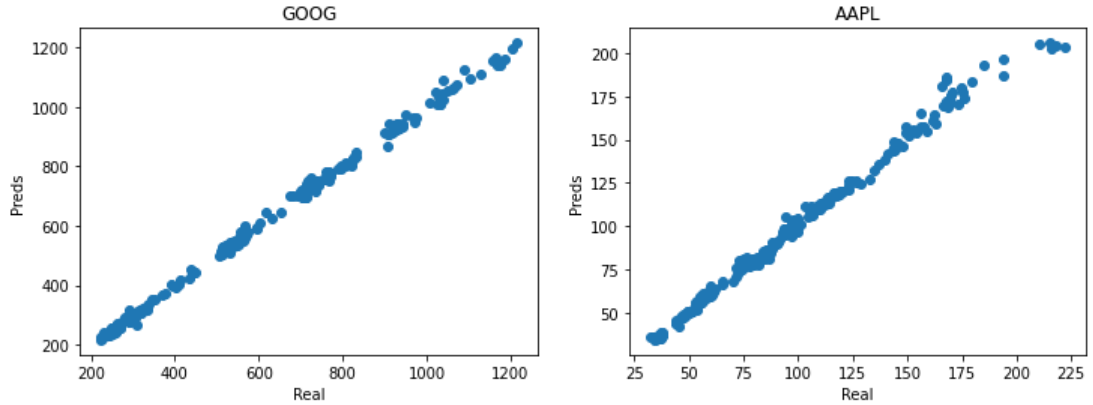
معماری GRU مهمترین تفاوتی که دارد این است که به جای دو خط cell state و hidden state فقط یک خط انتقال hidden state دارد. گیت سمت چپ reset gate است که با ضرب خروجی بعد از سیگموید در h مرحله قبلی تاثیر آن را در ترکیب با ورودی جدید مشخص می‌کند. سمت راست نیز update gate است که خروجی بعد از سیگموید را همزمان استفاده می‌کند تا میزان اضافه کردن مقدار جدید بر روی h را مشخص کند و همچنین با استفاده از

معکوس آن و ضربش در h قبلی تاثیر آن را کاهش می‌دهد. در مجموع GRU نسبت به LSTM محاسبات کمتری دارد و با سرعت بیشتری می‌تواند اجرا شود، و در مقابل آن، پیچیدگی های بیشتر LSTM پتانسیل یادگیری بیشتر و انعطاف بیشتر در انتقال اطلاعات را دارد.

در آزمایش‌ها ابتدا دیدیم که نتایج به ترتیب بهتر بودن از نظر لاس (کم بودن لاس) ابتدا مدل simpleRNN سپس GRU و درنهایت LSTM بود. اما باید گفت که این نتیجه برای تعداد ایپاک ثابت برای هر 3 مدل است. و چون ترتیب تعداد پارامتر هم از کم به زیاد همین ترتیب است، طبیعتا در تعداد ایپاک محدود، مدلی که کمتر پارامتر دارد می‌تواند نسبتا سریعتر آپدیت شده و نتیجه نسبتا کمی بهتر بگیرد. برای بررسی بهتر یک بار دیگر بدون محدودیت ایپاک اجرا کردیم که مقادیر لاس و ایپاکی که با early stopping متوقف شدیم در زیر آمده است. تصویر LSTM هم بعد از اجرای کامل برای نمونه آمده است که نشان می‌دهد خیلی بهتر می‌تواند پیش بینی کند نسبت به عکسهای قبلی که بعد 5 ایپاک بودند.




```
7/7 [==============================] - 0s 5ms/step - loss: 86.8505 - mse: 86.8505 - mape: 2.1786
Test MSE: 86.85050201416016 Test MAPE: 2.17856502532959 Test Loss: 86.85050201416016
```



شکل 6 - پیش بینی دقیق مدل LSTM بعد از آموزش به تعداد ایپاک کافی

LSTM

Epoch 13: early stopping

Test MSE: 86.85050201416016 Test MAPE: 2.17856502532959

GRU

Epoch 17: early stopping

Test MSE: 94.48188781738281 Test MAPE: 1.9911469221115112

SimpleRNN

Epoch 10: early stopping

Test MSE: 132.11981201171875 Test MAPE: 3.297455072402954

بر اساس مقدار لاس می‌بینیم که LSTM از نظر MSE بهترین بوده است که بر اساس پیچیدگی هایی که دارد و در بالا توضیح داده شد و تعداد پارامتر بیشترش منطقی است. سپس GRU است که نسبت به LSTM ساده تر است ولی باز مکانیزم فراموشی را دارد و توانسته در مقام دوم باشد. و در آخر مدل RNN ساده است که بخاطر همان سادگی و پارامتر کمتر طبیعتا نباید انتظار زیادی از آن داشت.

در مورد زمان اجرای 5 اپاک دیدیم که SimpleRNN از همه کندتر و سپس LSTM و سریعترین مدل GRU بود. بین LSTM و GRU مشخص است که LSTM توضیح داده شد پیچیده تر است و پارامتر بیشتر دارد و همین معادل کندتر شدن مدل است. اما مدل SimpleRNN با وجود اینکه پارامتر کمتری دارد اما بیشتر زمان می‌برد. طبیعتا بخاطر ساختار ساده تر و پارامتر کمتر باید این مدل سریعتر می‌بود. با کمی جستجو به نظر مشکل از پیاده سازی است همانطور که در لینک https://chowdera.com/2021/07/20210715130943615L.html توضیح داده شده است. و می‌دانیم موازی سازی کد و بهینه سازی آن تاثیر بسیاری دارد چون هر بهینه سازی ضربدر تکرارهای بسیار مدل می‌شود و همینکه مدل LSTM و GRU یک ورژن جلوتر از SimpleRNN هستند به نظر می‌رسد علت کند بودن SimpleRNN نسبت به آنها باشد.

ب) نحوه‌ی عملکرد شبکه برای دو تابع هزینه‌ی MSE و MAPE را بررسی کنید و نتایج بدست آمده و تفاوت این تابع را به صورت دقیق در گزارش خود ذکر کنید.

تابع هزینه MSE: این تابع مقدار loss را بر اساس میانگین فاصله‌ی مربعی مقادیر پیش بینی شده و مقادیر اصلی هدف محاسبه می کند. این تابع بر اساس فرمول زیر کار می‌کند:

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

شکل 7 - تابع MSE

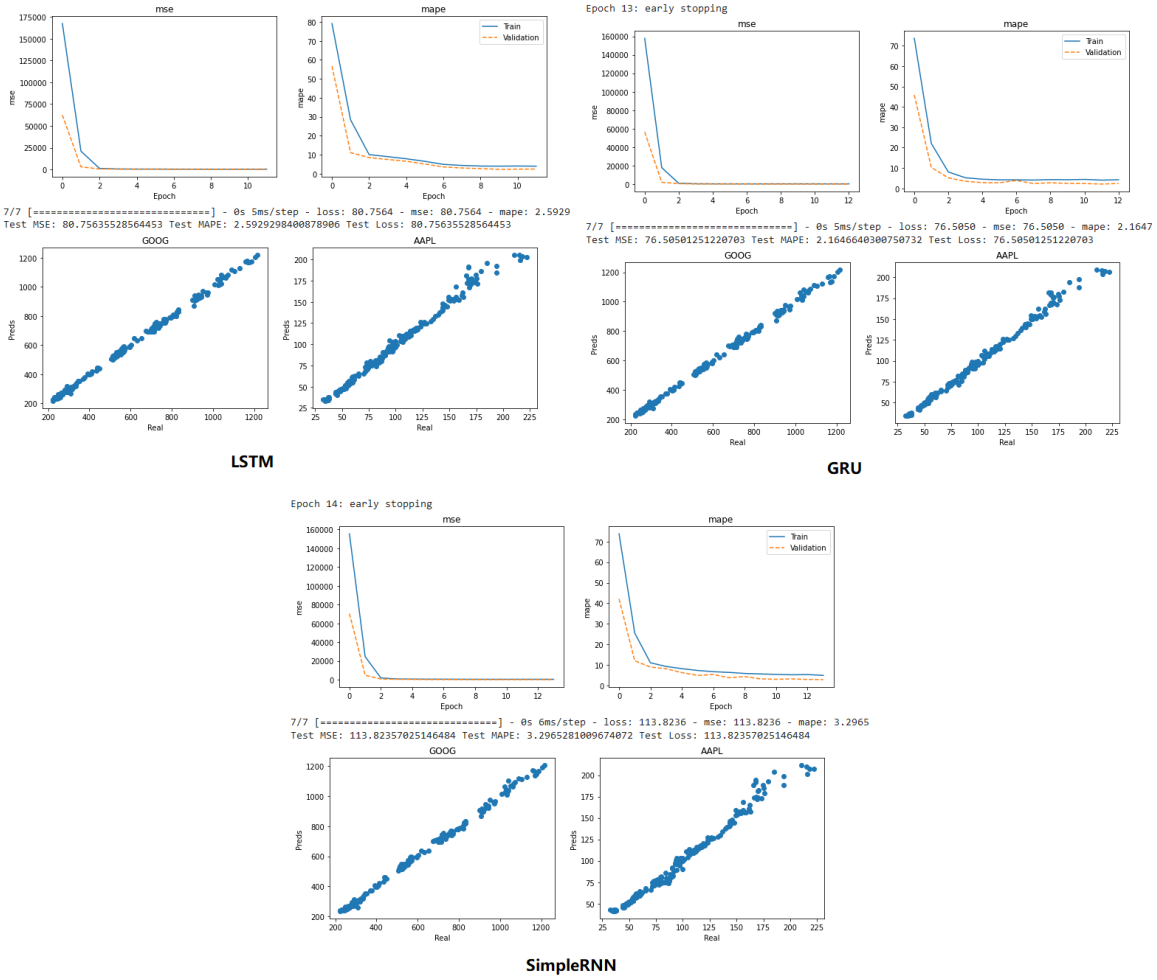
تابع هزینه MAPE: در ابن تابع برای محاسبه loss ابتدا اختلاف بین مقدار پیشبینی شده و هدف را محاسبه کرده و بر مقدار پیش بینی شده تقسیم میکنیم. سپس قدر مطلق این مقدار را محاسبه کرده و بین نمونه ها میانگین می گیریم.

این تابع بر اساس فرمول زیر کار می‌کند:

$$MAPE = \frac{1}{n}\sum_{t=1}^{n}\left|\frac{A_t - F_t}{A_t}\right|$$

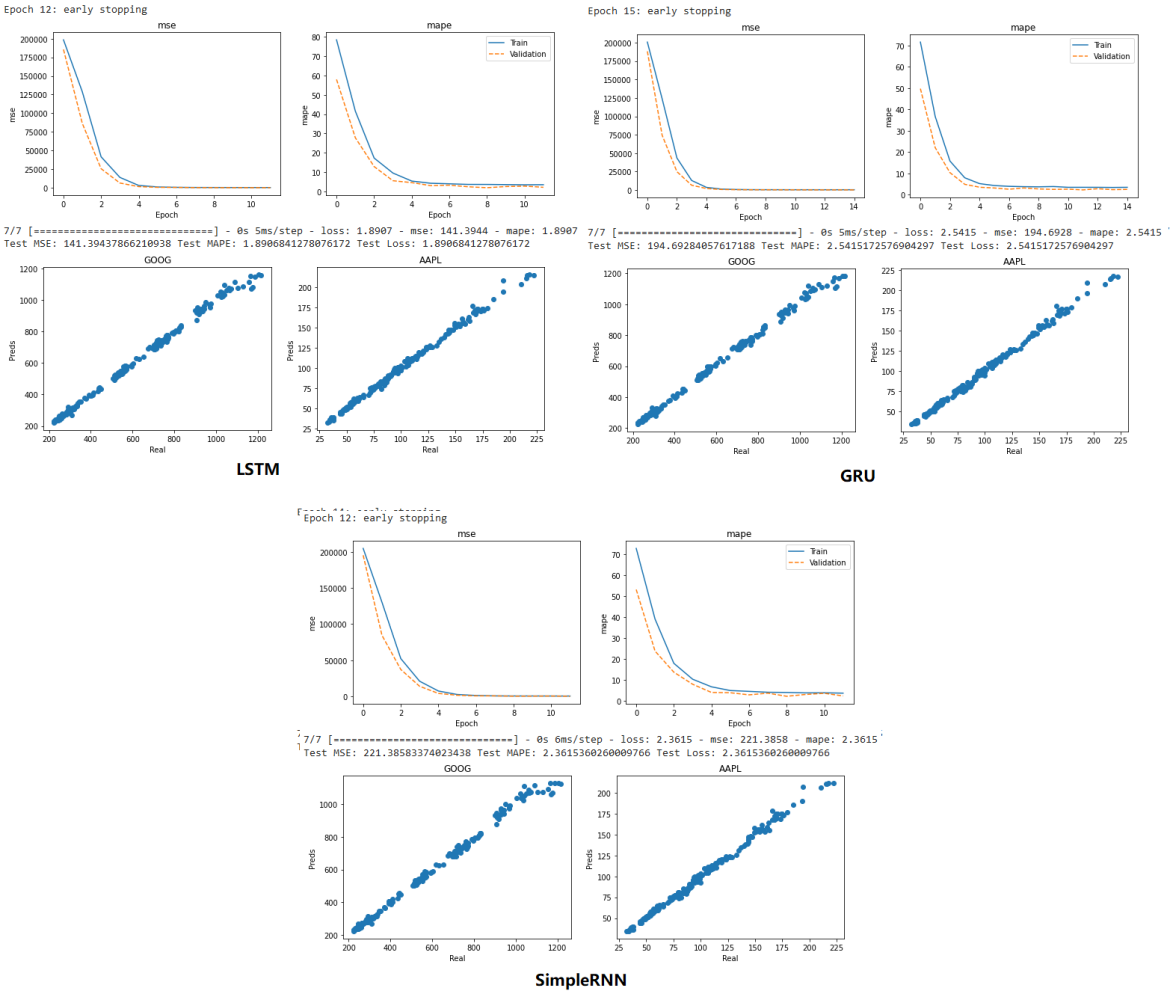
شکل 8 - تابع MAPE

ابتدا هر سه مدل را با استفاده از mse اجرا می‌کنیم.



شکل 9 - نتیجه اجرا با MAE

سپس هر سه مدل را با استفاده از MAPE اجرا کردیم.



شکل 10 - نتیجه اجرا با MAPE

همانطور که دیده می‌شود لاس روی هر کدام باشد، آن کمتر می‌شود و دیگری بیشتر (در کل روند نزولی است، منظور مینیمم است). تمام اطلاعات و مقادیر مورد نیاز در شکل ها آمده است. به عنوان نمونه با MAE مدل LSTM نتیجه

Test MSE: 80.75635528564453 Test MAPE: 2.5929298400878906

داشت و بعد با MAPE به نتیجه زیر رسید.

Test MSE: 141.39437866210938 Test MAPE: 1.8906841278076172

که دقیقا نشان می‌دهد در هر کدام، آن که در حال کاهشش بودیم روی داده تست نیز کمتر شده. البته همانطور که گفته شد هر دو لاس های مناسبی هستند و در هر دو حالت به نتیجه مطلوب رسیده‌ایم. توضیح تئوری لاس ها در بالاتر آمده است.

برای اطمینان از منظور سوال چون کمی گنگ است، یک متریک سوم هم در نظر میگیریم که کوریلیشن است و با scipy.stats.pearsonr آن را محاسبه می‌کنیم تا ببینیم بعد از آموزش با هر کدام از لاس ها کدام توانسته کوریلیشن بیشتری ایجاد کند.

این مقدار برای LSTM و MAE برابر زیر شد.

GOOG Correlation: (0.9990796326872471)

AAPL Correlation: (0.9962182328049571)

و وقتی با MAPE آموزش دادیم برابر مقدار زیر شد.

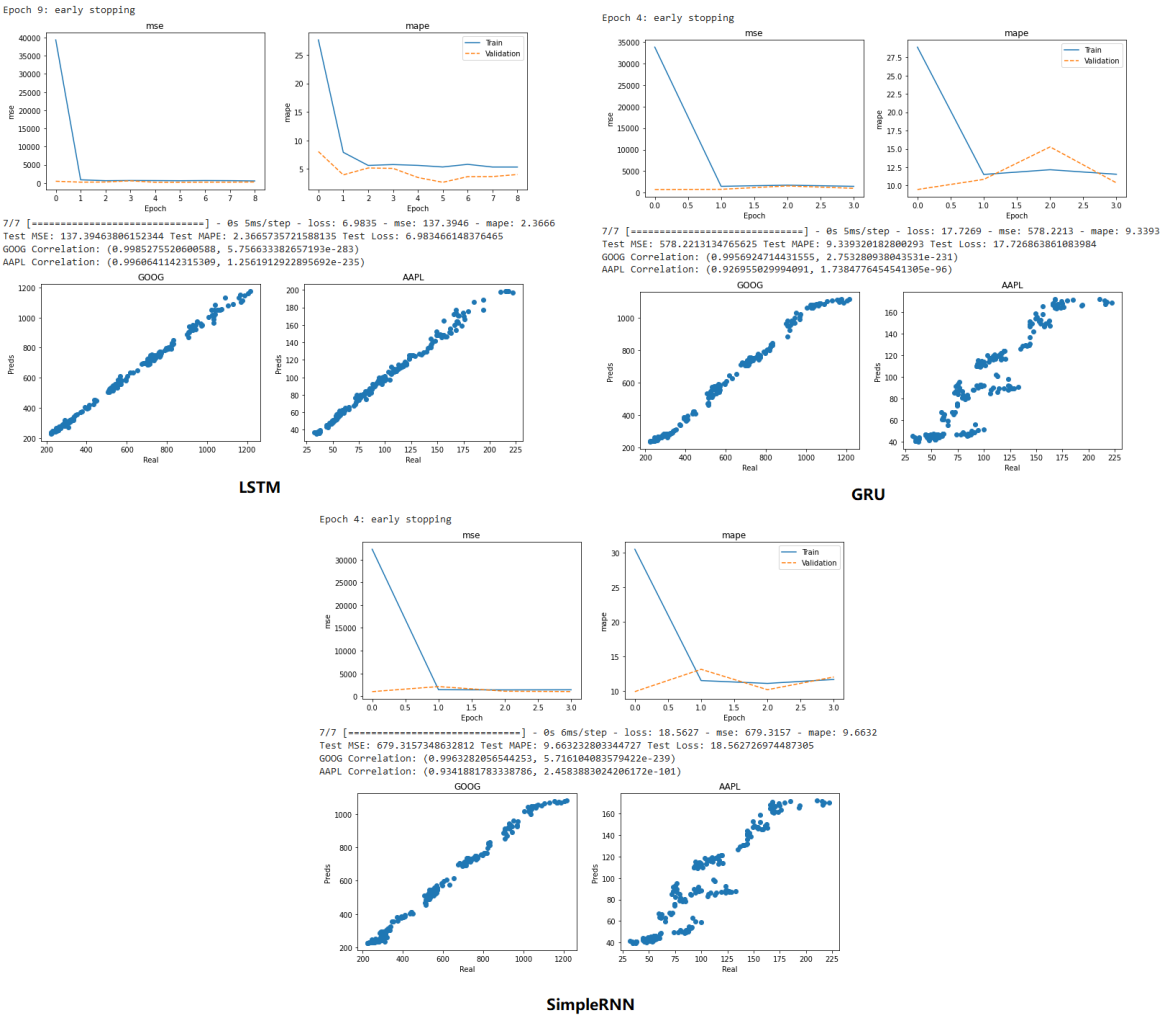
GOOG Correlation: (0.9979924821111592)

AAPL Correlation: (0.9982331305651372)

دیده می‌شود که این مقدار برای یک کلاس بهتر و برای دیگری کمی بدتر شده است. اما در مجموع، کوریلیشن انقدر بزرگ هست که نشان دهد هر دو لاس برای آموزش به خوبی عمل کرده‌اند.

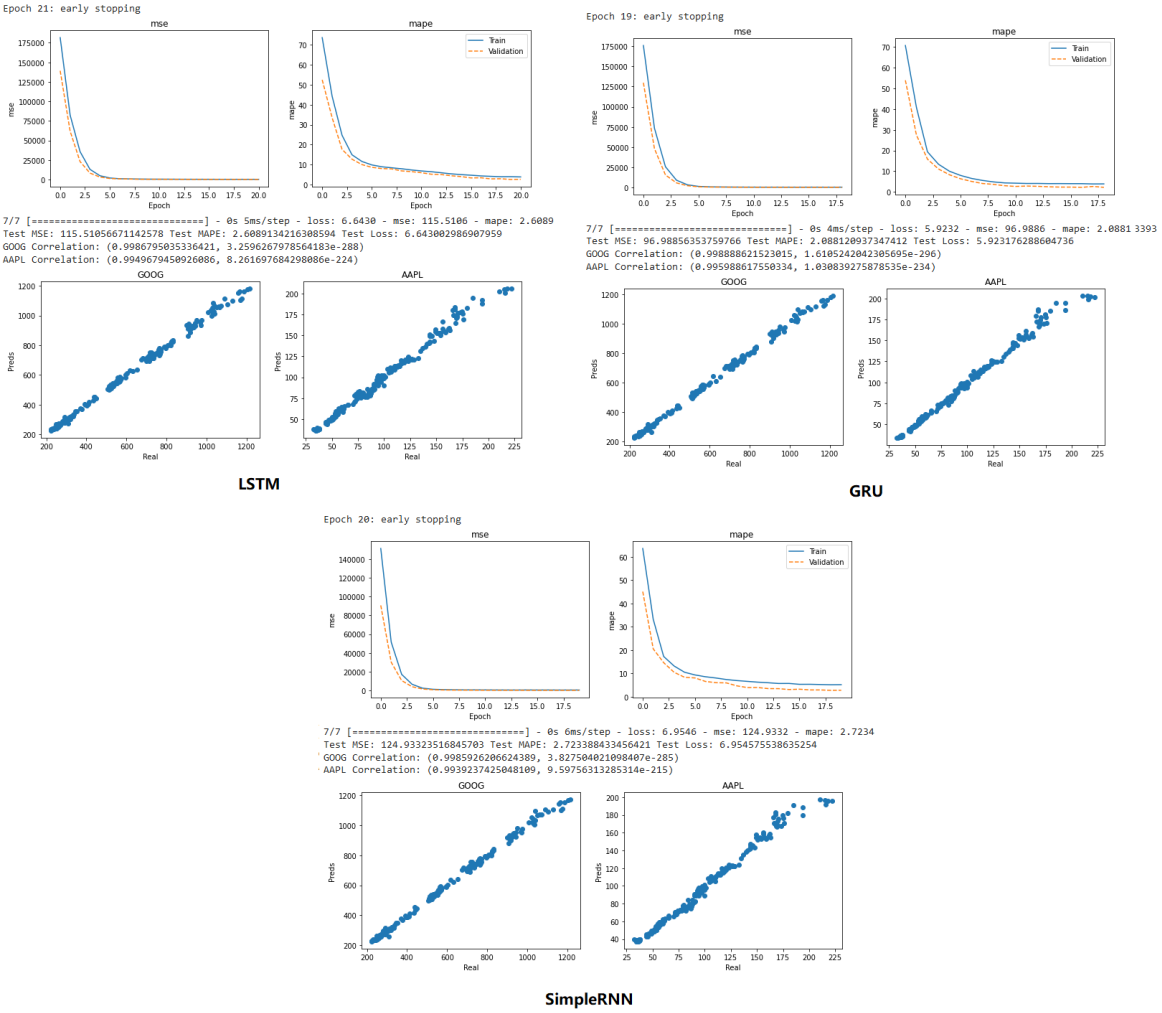
ج) نحوه ی عملکرد شبکه برای روشهای بهینه سازی متفاوت Adam، ADAgrad و RMSprop را بررسی کنید. نتایج بدست آمده و تفاوت این بهینه سازها را به صورت دقیق در گزارش خود ذکر کنید.

ابتدا نتایج اجرا با استفاده از Adam را می‌آوریم.



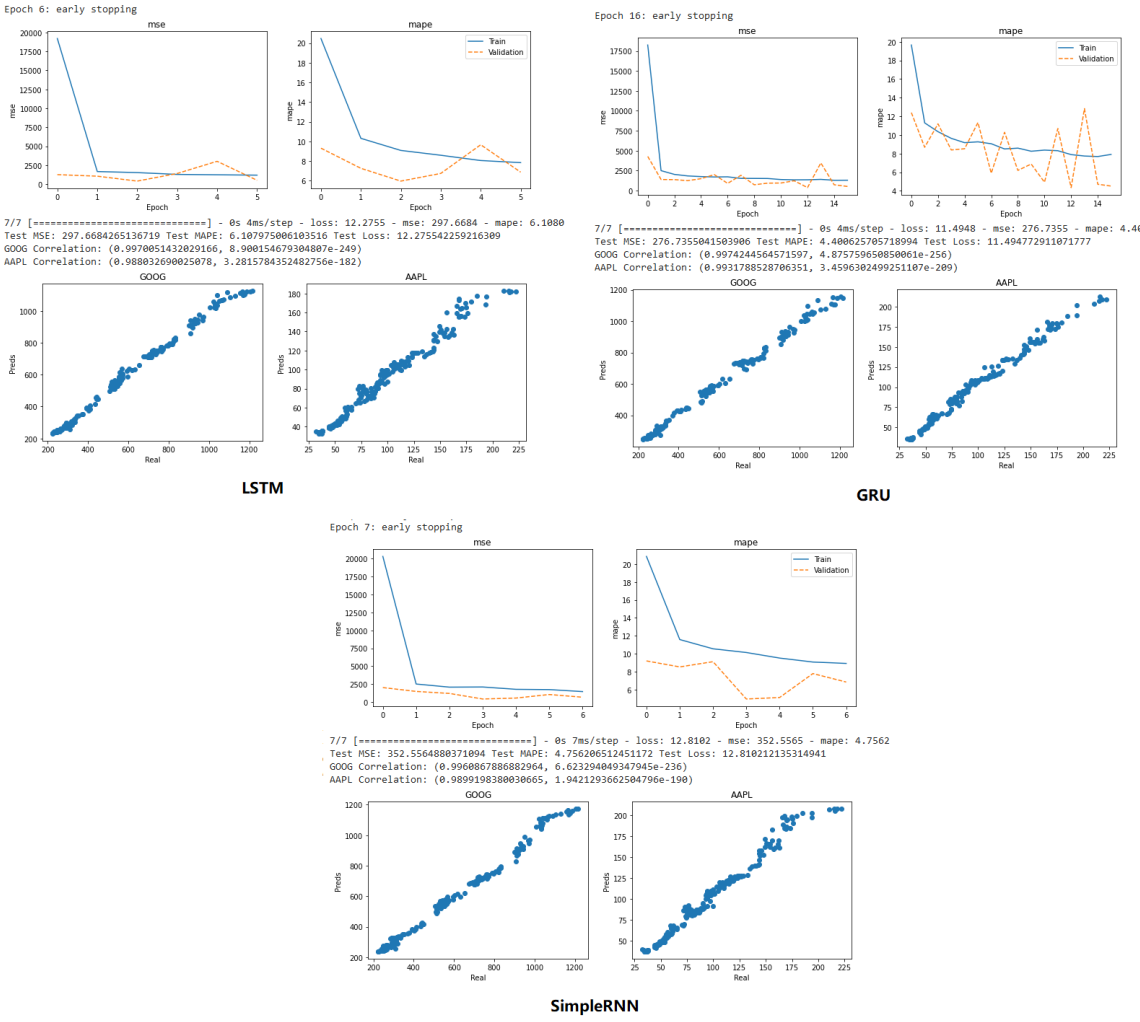
شکل 10 - نتیجه اجرا با ADAM

سپس نتیجه اجرا با ADAGRAD را می‌آوریم.



شکل 11 - نتیجه اجرا با ADAGRAD

و در نهایت نتیجه اجرا با بهینه ساز RMSprop را می‌آوریم.



شکل 12 - نتیجه اجرا با RMSprop

در مقایسه می بینیم که در این مثال خاص adagrad بهترین عملکرد، سپس adam و در آخر RMSProp است. بین adam و rmsprop می‌توان گفت چون روش adam معادل ترکیب کردن روش momentum و rmsprop است (رجوع شود به مقاله https://arxiv.org/abs/1412.6980) نشان داده شده از تک تک این دو بهتر می‌تواند عمل کند و اینجا هم دیدیم. برای تحلیل دقیق تر در زیر توضیح هر کدام را به طور کامل می‌دهیم. ضمنا باید اشاره کرد که این روش آزمایش برای مقایسه اصلا قابل اتکا نیست چون مخصوصا داریم به learning rate های دیفالت این روشها کفایت می‌کنیم درحالیکه همین هایپرپارامتر و خیلی هایپرپارامترهای دیگر می‌تواند تاثیرگذار باشد و بررسی به شکلی که بتوان از آن نتیجه قطعی گرفت خودش یک مقاله می‌شود.

ADAgrad: در روش ADAgrad یا Adaptive Gradient، بجای استفاده از مجموع گرادیان ها مثل momentum، مجموع مجذور آن ها را نگهداری می کند و از آن برای تطبیق گرادیان در جهات مختلف استفاده می کند:

$$sum\_of\_gradient\_squared = previous\_sum\_of\_gradient\_squared + gradient^2$$

$$delta = -learning\_rate * gradient / sqrt(sum\_of\_gradient\_squared)$$

$$theta += delta$$

RMSprop: مشکل AdaGrad این است که سرعت آن بسیار کند است. این به این دلیل است که مجموع مجذور گرادیان ها چون مثبت است فقط رشد می کند و هرگز کوچک نمی شود. بنابراین RMSProp یا Root Mean Square Propagation این مشکل را با افزودن یک عامل واپاشی یا همان decay برطرف می کند:

$$sum\_of\_gradient\_squared = previous\_sum\_of\_gradient\_squared * decay\_rate + gradient^2 * (1- decay\_rate)$$

$$delta = -learning\_rate * gradient / sqrt(sum\_of\_gradient\_squared)$$

$$theta += delta$$

Adam: روش Adam یا Adaptive Moment Estimation خوبی های هر دو روش قبلی را دارد و طبق تجربه عملکرد خوبی دارد. در نتیجه به انتخاب اول در حل مسائل یادگیری عمیق بدل شده است. این روش بصورت زیر عمل می‌کند:

$$sum\_of\_gradient = previous\_sum\_of\_gradient * beta1 + gradient * (1 - beta1) \text{ [Momentum]}$$

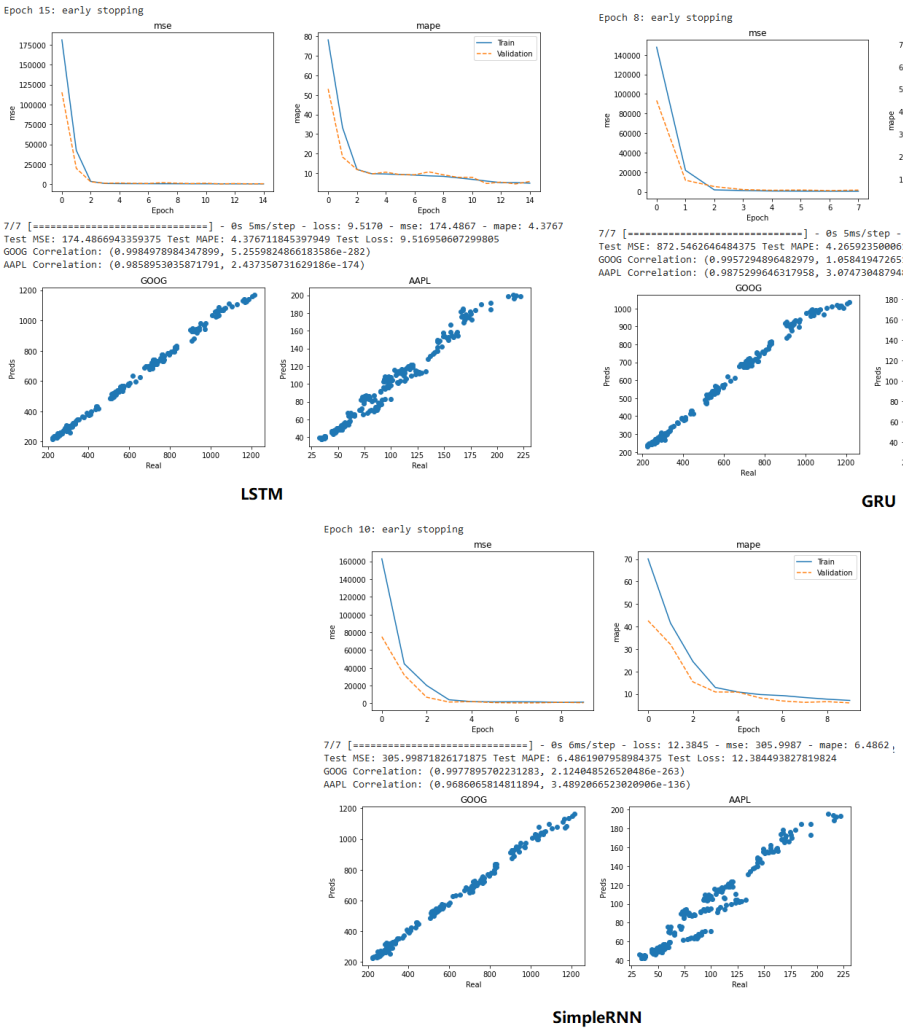
sum_of_gradient_squared = previous_sum_of_gradient_squared * beta2 + gradient^2 * (1- beta2) [RMSProp]

delta = -learning_rate * sum_of_gradient / sqrt(sum_of_gradient_squared)

theta += delta

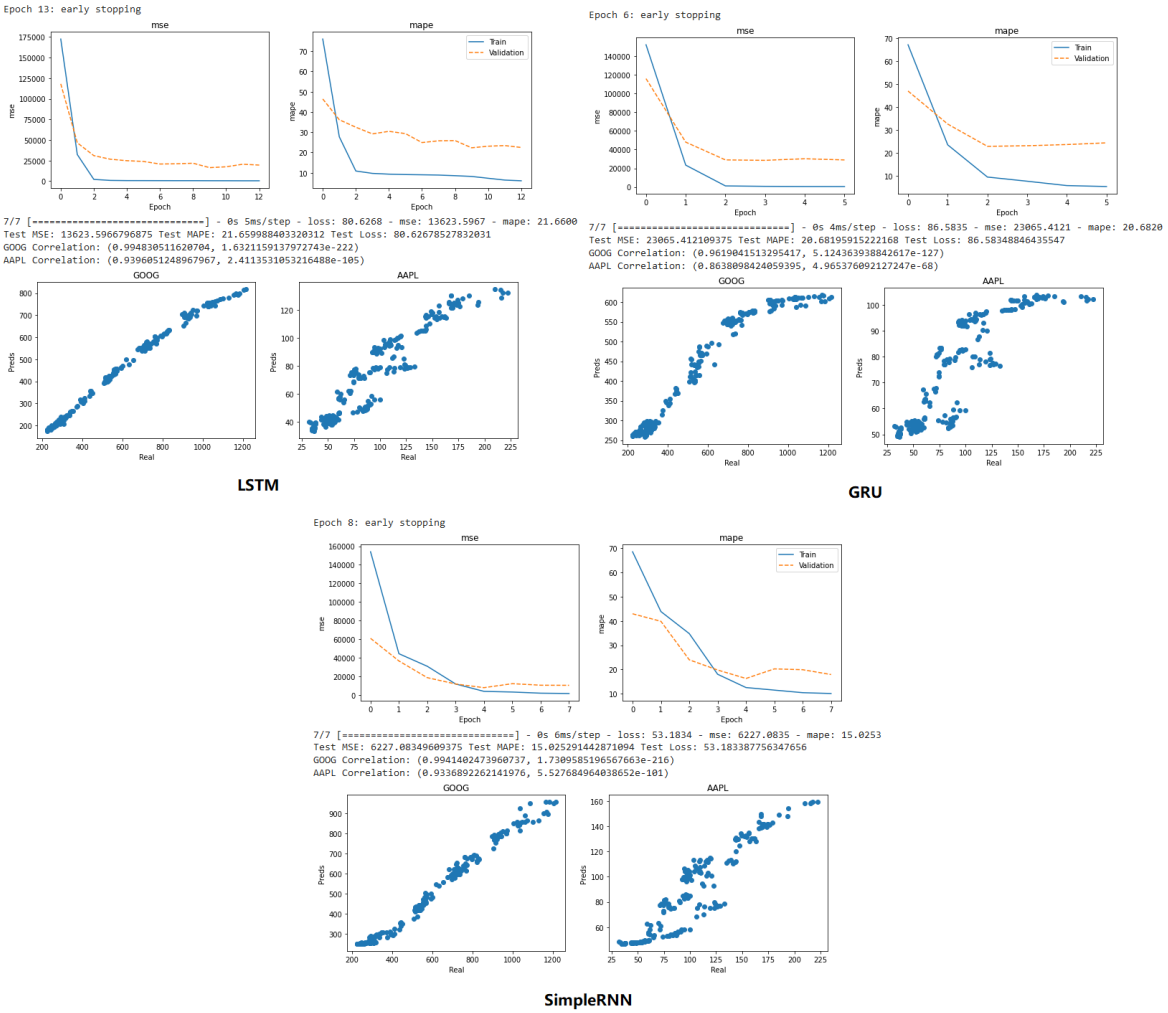
د) تاثیر dropout بر سلولهای بازگشتی را روی شبکه‌های طراحی شده بررسی کنید.

در بخشهای قبل همیشه dropout مدل recurrent برابر 0 بود. برای تکمیل آزمایشها، این مقدار را در این بخش برابر 0.05 و 0.2 قرار می‌دهیم تا تاثیر آن را ببینیم.



شکل 13 - نتیجه اجرا با dropout برابر 0.05

و در زیر نتیجه اجرا با dropout برابر 0.2 آمده است.



شکل 13 - نتیجه اجرا با dropout برابر 0.2

dropout روشی است که در آن تعدادی از واحد های شبکه را بصورت تصادفی حذف می کنیم. استفاده از dropout باعث می‌شود تا از overfitting در فرایند آموزش جلوگیری شود و عملکرد و پرفورمنس مدل بهتر شود.

اما مقدار زیاد آن هم باعث بایاس بیش از حد و واریانس کم می‌شود که می‌تواند آن هم برای مدل بد باشد و باید حد متعادل را رعایت کرد. همانطور که دیده می‌شود مقدار 0.2 در آزمایشها بیش از حد بوده و نتیجه را بدتر کرده است. ضمنا مقدار 0.05 هم حتی زیاد بود و نسبت به مقادیری که در بخشهای قبل با dropout برابر 0 دیدیم بدتر شد مدل.

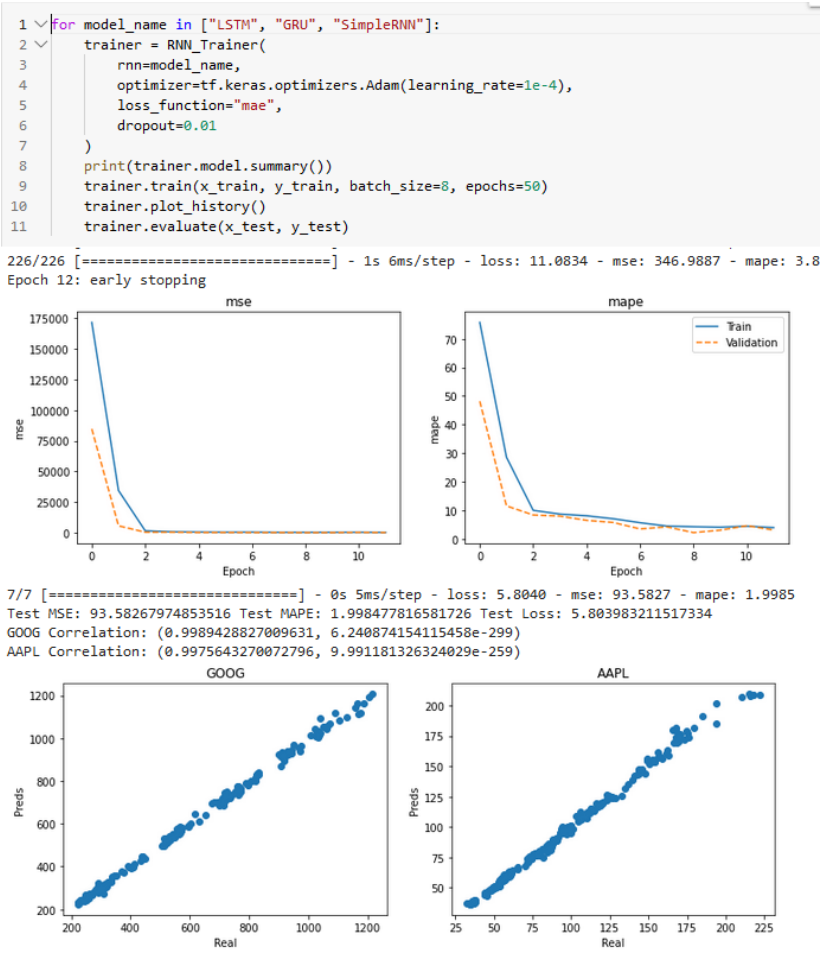
به همین دلیل این تست را روی LSTM دوباره با dropout برابر 0.01 تکرار کردیم و این مقدار، دقیقا مقدار مناسب است که توانست بهترین نتایج را بدهد و نتایج از وقتی dropout برابر بود و مقدار زیر را داشتند.

Test MSE: 109.67564392089844 Test MAPE: 3.9858860969543457

با dropout به مقدار 0.01 به نتایج زیر رسیدیم.

Test MSE: 93.58267974853516 Test MAPE: 1.998477816581726

رسیدیم که نشان می‌دهد از overfitting دور شدیم و generaliztion بهتری پیدا کردیم و توانستیم نتایج بهتری روی داده تست داشته باشیم.



شکل 13 - نتیجه اجرای LSTM با dropout برابر 0.01 که بهترین نتایج را داشت

سوال 2 – Text Generation

الف) مدلی مبتنی بر شبکه عصبی بازگشتی را طراحی کنید که بتواند 200 حرف متن را تولید نماید و نمودار خطا و صحتِ مدل را رسم کنید. همچنین ذکر کنید به چه دلیل پیش پردازش انتخاب شده میتواند باعث بهبود مدل شود (برای آموزش حداقل 3 ایپاک در نظر بگیرید. همچنین انتخاب مناسب پارامترهای مدل بر عهده دانشجو میباشد)

ابتدا مجموعه داده را بارگذاری می کنیم.

```
SEQ_LENGTH = 250

corpus = open("dataset.txt", 'r').read()
corpus = corpus.lower()
```

```
chars = sorted(list(set(corpus)))
char_to_id = {c: i for i, c in enumerate(chars)}
id_to_char = {i: c for i, c in enumerate(chars)}
vocab_size = len(chars)
tokenized_inputs = []
tokenized_outputs = []
for i in tqdm(range(0, len(corpus) - SEQ_LENGTH - 1, 251)):
    tokenized_inputs.append([char_to_id[c] for c in corpus[i: i + SEQ_LENGTH]])
    tokenized_outputs.append([char_to_id[c] for c in corpus[i + 1: i + SEQ_LENGTH + 1]])
tokenized_inputs = np.array(tokenized_inputs)
tokenized_outputs = np.array(to_categorical(tokenized_outputs, vocab_size))
x_train, x_test, y_train, y_test = train_test_split(tokenized_inputs,
                                                    tokenized_outputs,
                                                    test_size=0.1, random_state=42)
print("CHAR To ID:", char_to_id)
print("Inputs:", x_train.shape, x_test.shape, "Outputs:", y_train.shape, y_test.shape)
```

```
100% 4412/4412 [00:00<00:00, 26248.40it/s]

CHAR To ID: {'\t': 0, '\n': 1, ' ': 2, '!': 3, '"': 4, "'": 5, '(': 6, ')': 7, ',': 8, '-': 9, '.': 10, '/': 11, '0': 12,
'1': 13, '2': 14, '3': 15, '4': 16, '6': 17, '7': 18, '9': 19, ':': 20, ';': 21, '?': 22, '^': 23, '_': 24, 'a': 25, 'b':
26, 'c': 27, 'd': 28, 'e': 29, 'f': 30, 'g': 31, 'h': 32, 'i': 33, 'j': 34, 'k': 35, 'l': 36, 'm': 37, 'n': 38, 'o': 39, '
p': 40, 'q': 41, 'r': 42, 's': 43, 't': 44, 'u': 45, 'v': 46, 'w': 47, 'x': 48, 'y': 49, 'z': 50, '}': 51, '¢': 52, '¾': 5
3, 'â': 54, 'ã': 55, '€': 56}
Inputs: (3970, 250) (442, 250) Outputs: (3970, 250, 57) (442, 250, 57)
```

شکل 1 - بارگذاری و پیش‌پردازش داده

همانطور که خواسته شده مدلی که بر پایه کاراکتر باشد می‌خواهیم بسازیم بنابراین کاراکترها را به شماره ها نسبت می‌دهیم تا بتوان در مدل استفاده کرد. برای پیش‌پردازش نیز مهم است که ابتدا تمام حروف را lowercase کنیم زیرا اگر این کار را نکنیم تعداد کاراکترها حدودا دو برابر می‌شود. با این پیش پردازش باعث می‌شود 57 کاراکتر خاص داشته باشیم و مدل خیلی راحت تر می‌تواند آموزش ببیند و می‌توان با مدل کوچکتری متن های بهتری تولید کرد. این تعداد مستقیما در اندازه وان هات های ورودی مدل و تعداد لوجیت خروجی آن موثر است و پارامترها را تغییر می دهد و همچنین سختی مسئله را بیشتر می‌کند. در ادامه کار مهم است که داده مناسب برای آموزش مدل بسازیم. برای این کار، هر داده آموزشی یک متن به طول 250 در نظر می‌گیریم. و داده خروجی آن هم 250 کاراکتر است با این تفاوت که یکی به راست شیفت خورده است. به این ترتیب مدل باید با دیدن هر حرف ورودی، بتواند حرف بعدی را حدس بزند و این کار به ازای تمام 250 کاراکتر انجام می‌شود. بنابراین این پنجره 250 تایی را روی متن حرکت می‌دهیم و

داده را درست می‌کنیم. در آخر برای خروجی مدل باید هر کدام از خروجی ها به شکل onehot باشند و به همین دلیل با استفاده از to_categorical این کار را انجام می‌دهیم و همانطور که دیده می‌شود ابعاد خروجی به شکل سایز دیتا در 250 در 57 می‌شود. برای ورودی این اتفاق لازم تیست چون در مدل که در ادامه توضیح می‌دهیم از لایه Embedding استفاده می‌کنیم که خودش عدد می‌گیرد و تبدیل آن به وانهات به صورت ضمنی درونش انجام می‌شود و به همین دلیل ابعاد ورودی به شکل تعداد داده در 250 است.

همچنین پس از تمام این پردازش‌ها داده را به دو بخش آموزش و تست با نسبت 0.9 و 0.1 تقسیم می‌کنیم تا مدل های نهایی را روی بخش تست بسنجیم و گزارش کنیم. داده validation در ادامه توضیح داده می‌شود که چطور ساخته خواهد شد و نیاز نیست اینجا جدا شود.

در زیر کدی که برای ساخت و آموزش مدل، ارزیابی آن، نمایش نمودارها و پیش‌بینی نوشته شده است آمده است.

```
class RNN_Trainer:
    def __init__(self, vocab_size, seq_length, rnn="LSTM", activation_function="relu",
                 optimizer="adam", loss_function='categorical_crossentropy', units=1024) -> None:
        self.optimizer = optimizer
        self.activation_function = activation_function
        self.loss_function = loss_function
        self.rnn = rnn
        self.seq_length = seq_length
        self.model = self.build_model(vocab_size, seq_length, units)
        self.history = None
        self.training_time = None

    def build_model(self, vocab_size, seq_length, units):
        rnn_map = {
            "SimpleRNN": tf.keras.layers.SimpleRNN(units, return_sequences=True),
            "GRU": tf.keras.layers.GRU(units, return_sequences=True, dropout=0.0),
            "LSTM": tf.keras.layers.LSTM(units, return_sequences=True, dropout=0.0),
        }
        model = tf.keras.Sequential([
            tf.keras.layers.Embedding(input_dim=vocab_size, output_dim=64, input_length=seq_length),
            rnn_map[self.rnn],
            tf.keras.layers.Dense(256, activation=self.activation_function),
            tf.keras.layers.Dropout(0.1),
            tf.keras.layers.Dense(128, activation=self.activation_function),
            tf.keras.layers.Dense(vocab_size, activation='softmax')
        ])
        model.compile(
            optimizer=self.optimizer,
            loss=self.loss_function,
            metrics=['accuracy']
        )
        return model

    def train(self, tokenized_inputs, tokenized_outputs, batch_size=64, epochs=5):
        early_stopping = tf.keras.callbacks.EarlyStopping(
            monitor='val_loss',
            verbose=1,
            patience=3,
            mode='min',
            restore_best_weights=True
        )
```

```
        self.history = self.model.fit(
            tokenized_inputs, tokenized_outputs,
            batch_size=batch_size,
            epochs=epochs,
            verbose=1,
            validation_split=0.1,
            callbacks=[early_stopping],
        )

    def plot_history(self):
        fig = plt.figure(figsize=(12, 4))
        metrics = ['loss', 'accuracy']
        for n, metric in enumerate(metrics):
            plt.subplot(1, 2, n+1)
            plt.plot(self.history.epoch, self.history.history[metric], label='Train')
            plt.plot(self.history.epoch, self.history.history[f"val_{metric}"], linestyle="--",
label='Validation')
            plt.xlabel('Epoch')
            plt.ylabel(metric)
            plt.title(metric)
        plt.legend()
        plt.show()

    def evaluate(self, x_test, y_test):
        [test_loss,test_acc] = self.model.evaluate(x_test, y_test)
        print("Test Loss:", test_loss, "Test Accuracy:", test_acc)
        test_preds = np.argmax(self.model.predict(x_test), axis=-1)
        y_test = np.argmax(y_test, axis=-1)

    def predict(self, start_str="ma", argmax=True, temperature=0.5):
        tokenized_current = [char_to_id[c] for c in start_str]
        for i in tqdm(range(len(tokenized_current), self.seq_length)):
            mask = np.zeros(self.seq_length)
            mask[:i] = 1
            input_ids = np.zeros(self.seq_length)
            input_ids[:i] = tokenized_current
            predicted_logits = self.model(inputs=np.array([input_ids]), mask=mask)
            if argmax:
                next_id = np.argmax(predicted_logits, axis=-1)[0][i - 1]
            else:
                predicted_logits = predicted_logits / temperature
                next_id = tf.random.categorical(predicted_logits[:, i - 1, :], num_samples=1).numpy()[0][0]
            tokenized_current.append(next_id)
        return "".join([id_to_char[id] for id in tokenized_current])
```

کد 1 - کد آموزش مدل و ارزیابی و نمودارها و پیش‌بینی

همانطور که دیده می‌شود برای ساخت مدل از مدلهای حافظه recurrent استفاده می‌کنیم و کد به صورتی نوشته شده که به راحتی بتوان مدل آن را بین SimpleRNN یا GRU یا LSTM تغییر داد. (در بخشهای بعدی توضیحات بیشتری از این مدلها می‌دهیم.)

```
trainer = RNN_Trainer(vocab_size, SEQ_LENGTH,
#                       optimizer=tf.keras.optimizers.Adam(learning_rate=1e-2)
                      )
print(trainer.model.summary())
trainer.train(x_train, y_train, batch_size=8, epochs=10)
```

```
Model: "sequential_32"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_32 (Embedding)    (None, 250, 64)           3648

 lstm_32 (LSTM)              (None, 250, 1024)         4460544

 dense_88 (Dense)            (None, 250, 256)          262400

 dropout_32 (Dropout)        (None, 250, 256)          0

 dense_89 (Dense)            (None, 250, 128)          32896

 dense_90 (Dense)            (None, 250, 57)           7353

=================================================================
Total params: 4,766,841
Trainable params: 4,766,841
Non-trainable params: 0
_________________________________________________________________
None
Epoch 1/10
447/447 [==============================] - 37s 78ms/step - loss: 2.4602 - accuracy: 0.3049 - val_loss: 1.8985 - val_accuracy: 0.439
6
Epoch 2/10
447/447 [==============================] - 34s 77ms/step - loss: 1.6792 - accuracy: 0.4992 - val_loss: 1.4706 - val_accuracy: 0.555
9
Epoch 3/10
447/447 [==============================] - 35s 77ms/step - loss: 1.3928 - accuracy: 0.5770 - val_loss: 1.3119 - val_accuracy: 0.597
8
Epoch 4/10
447/447 [==============================] - 35s 77ms/step - loss: 1.2591 - accuracy: 0.6121 - val_loss: 1.2348 - val_accuracy: 0.618
4
Epoch 5/10
447/447 [==============================] - 34s 76ms/step - loss: 1.1768 - accuracy: 0.6342 - val_loss: 1.2001 - val_accuracy: 0.628
7
Epoch 6/10
447/447 [==============================] - 34s 76ms/step - loss: 1.1138 - accuracy: 0.6514 - val_loss: 1.1756 - val_accuracy: 0.635
5
Epoch 7/10
447/447 [==============================] - 35s 78ms/step - loss: 1.0594 - accuracy: 0.6656 - val_loss: 1.1727 - val_accuracy: 0.638
5
Epoch 8/10
447/447 [==============================] - 34s 76ms/step - loss: 1.0096 - accuracy: 0.6793 - val_loss: 1.1719 - val_accuracy: 0.640
3
Epoch 9/10
447/447 [==============================] - 34s 75ms/step - loss: 0.9614 - accuracy: 0.6922 - val_loss: 1.1905 - val_accuracy: 0.639
3
Epoch 10/10
447/447 [==============================] - 34s 76ms/step - loss: 0.9143 - accuracy: 0.7054 - val_loss: 1.2112 - val_accuracy: 0.639
4
```

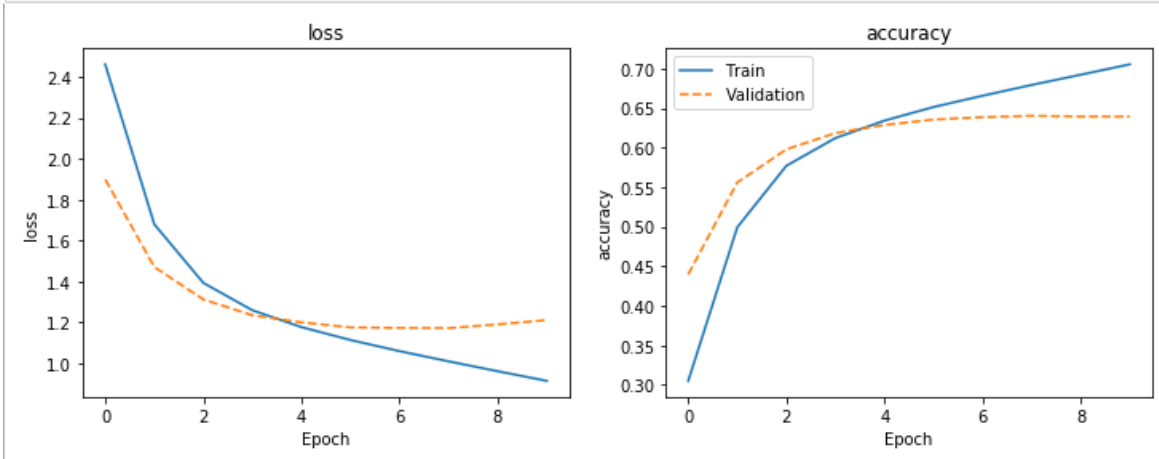
شکل 2 - ساختار مدل و آموزش آن و نتایج بخش validation

در بالا ساختار مدل و مراحل آموزش آن و نتایج لاس و دقت روی بخش validation در طول ایپاکها دیده می‌شود. برای ساخت مدل ابتدا یک لایه Embedding قرار داده ایم تا هر کاراکتر را به بردارهای dense به ابعاد 64 (دلخواه) تبدیل کند تا بازنمایی اولیه از هر کاراکتر داشته باشیم. سپس یک لایه LSTM قرار دادیم تا این کاراکترها را به صورت time series بگیرد و یک بازنمایی 1024 بعدی (دلخواه) از کاراکترهای زمان t و تمام زمانهای قبل t بدهد. ضمنا return_sequences برابر True قرار دادیم به این معنا که تمام بازنمایی ها در هر زمان t را بازگرداند تا بتوانیم به ازای هر زمان کاراکتر بعدی که قبلتر توضیح داده شد را به دست آوریم. سپس بر روی این خروجی‌ها یک شبکه چندلایه Feed Forward یا Dense قرار دادیم با activation function های ReLU که مرسوم است و در نهایت آخرین لایه که 57 نورون به

تعداد کاراکتر ها دارد و یک softmax روی آن زده می‌شود تا به شکل احتمالی در بیاید و هر چه کاراکتری که مدل فکر می‌کند بعدش می‌آید انتظار داریم احتمال بیشتری داشته باشد. برای آموزش مدل نیز از لاس cross entropy استفاده می‌کنیم که فاصله توزیع وان هاتی که ساخته بودیم را با همین خروجی سافتمکس مدل مقایسه می‌کند و هر چه فاصله بیشتر باشد عدد بزرگتر است و مدل سعی می‌کند تا با آپدیت کردن پارامترها این مقدار را کاهش دهد. ضمنا در میان لایه ها، لایه های Dropout هم گذاشتیم که با خاموش کردن رندوم درصدی از نورون ها باعث جلوگیری از overfitting و بهتر شدن generalization مدل می‌شود.در ضمن باید اشاره کرد که همانطور که قبلا گفته شد بخش validation از طریق ارگومان validation_split به تابع fit داده می‌شود و کراس داده آموزش را به دو بخش تقسیم و روی یکی ترین و روی دیگری validation را اجرا می‌کند در طول ایپاک ها. در اخر نیز تابع evaluate را نوشتیم که این کار را روی داده تست که قبلا جدا کرده ایم و ثابت است تست می‌کنیم.

نتیجه آموزش و دقت و لاس بخش تست در زیر آمده است.

```
trainer.plot_history()
trainer.evaluate(x_test, y_test)  # 6534 24s
```




```
14/14 [==============================] - 1s 50ms/step - loss: 1.2015 - accuracy: 0.6430
Test Loss: 1.2014552354812622 Test Accuracy: 0.6429864168167114
```

شکل 3 - نمودار خطا و صحت مدل و نتیجه تست

در شکل لاس و دقت تست نیز آمده است.

همانطور که دیده می‌شود به شکل مناسبی لاس رو به کاهش و دقت رو به افزایش است و در اواخر که کم کم داریم به overfitting می رسیم و لاس validation رو به افزایش بوده متوقف شدیم.

و حالا به توضیح بخش prediction مدل می‌پردازیم. برای این کار دو روش وجود دارد. یک روش این است که لایه های LSTM یا GRU می‌توانند state خود را خروجی دهند و می‌توان تک تک ورودی داد و خروجی و state آن ها را گرفت و در مرحله بعد ورودی زمان بعدی را به همراه state قبلی به آن داد. اما این کار کمی به خود مدل بستگی دارد چون مثلا LSTM دو خط انتقال دارد و GRU مثلا یکی. به همین دلیل از روش دوم استفاده می‌کنیم.

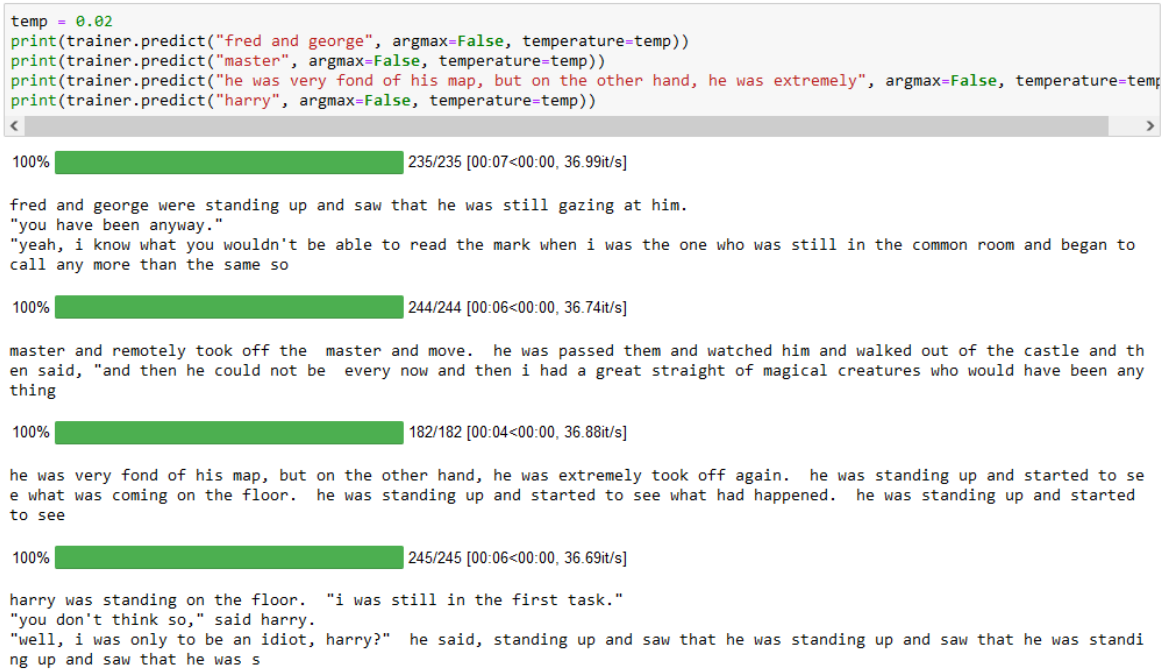
در روش دوم از mask استفاده می‌کنیم که در NLP بسیار پر استفاده است. در این مرحله مانند آموزش مدل 250 کاراکتر ورودی می‌دهیم و 250 تا logit هم خروجی می‌گیریم ولی در هر مرحله یک حرف می‌دهیم و باقی جمله را با mask می‌پوشانیم سپس حرف جدید پیش‌بینی شده را اضافه می‌کنیم و دوباره این دو حرف را می‌دهیم و همین کار را ادامه می‌دهیم. این روش در تابع predict که در بالاتر آمده پیاده شده است. ضنا برای این که هر بار argmax را بر نداریم و کمی هم بتوانیم خروجی ها متفاوت ایجاد کنیم از روش roulette wheel selection استفاده می‌کنیم به این شکل که احتمال برداشتن هر کدام از کاراکترها به نسبت احتمالشان در کل است. البته این روش به صورت خام خروجی های بیش از حد رندوم و بی معنا می‌داد که با بررسی مراجع، مثل https://www.tensorflow.org/text/tutorials/text_generation تصمیم گرفتیم از temperature نیز استفاده کنیم که معین می‌کند چه مقدار به سمت رندوم و چه مقدار به سمت بیشترین احتمال متمایل شویم و با کم کردن دما از آن رندومنس بیش از حد دور می‌شویم و نتایج بهتری می‌بینیم.

```
print(trainer.predict("fred and george", argmax=True))
print(trainer.predict("master", argmax=True))
print(trainer.predict("he was very fond of his map, but on the other hand, he was extremely", argmax=True))
print(trainer.predict("harry", argmax=True))
```

```
100% 235/235 [00:06<00:00, 37.06it/s]

fred and george were standing up and started to see what had happened to the statue the stars of the stands of the wands w
ere still gazing around the corridors an extraordinary invisibility cloak and then said, "and then he could have been a bi
t of a

100% 244/244 [00:06<00:00, 37.03it/s]

master and showing him and she was still staring at her.
"you have been a bit of a friend," said harry.  "i was still in the common room and forgetting the dark arts teacher will
be able to stop himself in my office on the way they were all right?"
"

100% 182/182 [00:04<00:00, 37.12it/s]

he was very fond of his map, but on the other hand, he was extremely took off against the darkness and the sounds of the w
and was still staring at her.  "i have a great strange of magical creatures, is it?"  said hermione sharply.  "i had a few
moden

100% 245/245 [00:06<00:00, 37.00it/s]

harry was standing up and saw that he was still gazing at him.  he was standing up and saw that he was still staring at he
r.
"you have been a bit of a friend," said harry.  "i was still in the common room and forgetting the dark arts teacher will
be
```

شکل 4 - نمونه هایی از خروجی های argmax مدل

در بالا نمونه هایی از حالت پیش بینی با استفاده از بیشترین احتمال دیده می‌شود. همانطور که دیده می‌شود، مدل نه تنها کلمات درست، بلکه جملات درستی نیز می‌دهد که نشان می‌دهد LSTM توانسته به خوبی ورودی ها در طول زمان را مدل کند و پیش‌بینی بعدی را به خوبی انجام دهد.



شکل 5 - نمونه هایی از خروجی های با رندوم دمایی از مدل

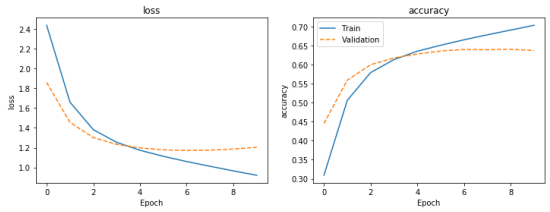
در بالا نمونه‌هایی از خروجی ها وقتی با روش رندومی که توضیح داده شد آمده است. همانطور که دیده می‌شود جمله های کاملا متفاوتی می‌توان با این روش تولید کرد. اگرچه چون همیشه بهترین گزینه مدل نبوده شاید اندکی کیفیت و ثبات argmax را نداشته باشد که البته با تنظیم دما می‌توان کاملا تنظیمش کرد.

ب) با استفاده از 2 تابع زیان دیگر خطای مدل را ارزیابی کنید.

از آنجا که این مسئله حالت classification دارد، توابع خطایی که بررسی می‌کنیم هم باید مختص همین مسئله باشد. به همین دلیل دو تابع زیان Kullback-Leibler divergence loss که مشابه همان cross entropy loss که در بخش قبل استفاده شده است با این تفاوت که خود آنتروپی را نیز در نظر دارد و به همین دلیل مقدار آن کف یا همان مینیمم دارد که برای تفسیرپذیری بهتر کمک می‌کند. این لاس مخصوصا در مسائل knowledge distillation مثل distilBERT استفاده می‌شود و زمان‌هایی مخصوصا که توزیع درست به شکل one-hot نیست و مثلا خودش توزیعی غیر 0 و 1 است. به همین دلیل تعمیم بیشتری دارد.ضمنا از تابع زیان BinaryFocalCrossentropy استفاده می‌کنیم.

```
trainer = RNN_Trainer(vocab_size, SEQ_LENGTH,
                      loss_function=tf.keras.losses.KLDivergence()
                      )
print(trainer.model.summary())
trainer.train(x_train, y_train, batch_size=8, epochs=10)
trainer.plot_history()
trainer.evaluate(x_test, y_test)
```

```
Model: "sequential_33"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_33 (Embedding)    (None, 250, 64)           3648

 lstm_33 (LSTM)              (None, 250, 1024)         4460544

 dense_91 (Dense)            (None, 250, 256)          262400

 dropout_33 (Dropout)        (None, 250, 256)          0

 dense_92 (Dense)            (None, 250, 128)          32896

 dense_93 (Dense)            (None, 250, 57)           7353

=================================================================
Total params: 4,766,841
Trainable params: 4,766,841
Non-trainable params: 0
_________________________________________________________________
None
Epoch 1/10
447/447 [==============================] - 39s 78ms/step - loss: 2.4370 - accuracy: 0.3092 - val_loss: 1.8596 - val_accuracy: 0.445
4
Epoch 2/10
447/447 [==============================] - 35s 78ms/step - loss: 1.6596 - accuracy: 0.5065 - val_loss: 1.4555 - val_accuracy: 0.559
6
Epoch 3/10
447/447 [==============================] - 35s 79ms/step - loss: 1.3813 - accuracy: 0.5795 - val_loss: 1.3014 - val_accuracy: 0.600
2
Epoch 4/10
447/447 [==============================] - 34s 77ms/step - loss: 1.2536 - accuracy: 0.6136 - val_loss: 1.2330 - val_accuracy: 0.618
0
Epoch 5/10
447/447 [==============================] - 34s 77ms/step - loss: 1.1733 - accuracy: 0.6358 - val_loss: 1.1972 - val_accuracy: 0.628
3
Epoch 6/10
447/447 [==============================] - 34s 77ms/step - loss: 1.1122 - accuracy: 0.6514 - val_loss: 1.1781 - val_accuracy: 0.636
2
Epoch 7/10
447/447 [==============================] - 35s 79ms/step - loss: 1.0592 - accuracy: 0.6658 - val_loss: 1.1703 - val_accuracy: 0.640
2
Epoch 8/10
447/447 [==============================] - 35s 79ms/step - loss: 1.0115 - accuracy: 0.6789 - val_loss: 1.1743 - val_accuracy: 0.639
8
Epoch 9/10
447/447 [==============================] - 35s 78ms/step - loss: 0.9648 - accuracy: 0.6914 - val_loss: 1.1856 - val_accuracy: 0.640
6
Epoch 10/10
447/447 [==============================] - ETA: 0s - loss: 0.9207 - accuracy: 0.7040Restoring model weights from the end of the bes
t epoch: 7.
447/447 [==============================] - 35s 78ms/step - loss: 0.9207 - accuracy: 0.7040 - val_loss: 1.2039 - val_accuracy: 0.637
9
Epoch 10: early stopping
```




```
14/14 [==============================] - 1s 50ms/step - loss: 1.1689 - accuracy: 0.6401
Test Loss: 1.1689121723175049 Test Accuracy: 0.6400905251502991
```

شکل 6 - استفاده از لاس KL Divergence

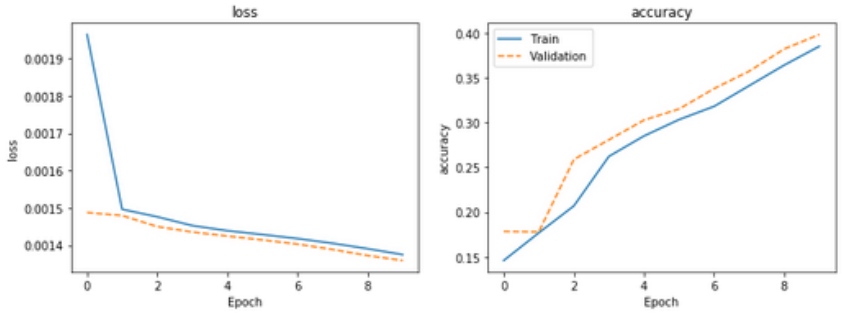
در شکل مراحل آموزش و در نهایت نمودار و دقت و لاس روی تست آمده است وقتی که از KL Divergence استفاده شود. لازم به ذکر است که $H(p,q) = H(p) + D_{KL}(p \parallel q),$ که یعنی cross entropy که در بخش اول استفاده شده بود برابر مجموع entropy و KL است. نکته جالب این است که وقتی p که توزیع صحیح است به شکل onehot که در مسائل classification و اینجا داریم، آنتروپی آن برابر 0 می‌شود و در اصل کراس انتروپی معادل KL Divergence می‌شود. البته باید اشاره کرد که محاسبه KL Divergence به علت داشتن همان

عبارت آنتروپی p، باعث می‌شود نسبت به محاسبه cross entropy کمی کندتر باشد که در زمانی که برای هر اپاک صرف شده است در این بخش و مقایسه آن با بخش قبل دیده می‌شود.

در زیر استفاده از تابع لاس BinaryFocalCrossentropy آمده است.

```
trainer = RNN_Trainer(vocab_size, SEQ_LENGTH,
                      loss_function=tf.keras.losses.BinaryFocalCrossentropy()
                     )
print(trainer.model.summary())
trainer.train(x_train, y_train, batch_size=8, epochs=10)
trainer.plot_history()
trainer.evaluate(x_test, y_test)
```

```
Model: "sequential_35"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_35 (Embedding)    (None, 250, 64)           3648

 lstm_35 (LSTM)              (None, 250, 1024)         4460544

 dense_97 (Dense)            (None, 250, 256)          262400

 dropout_35 (Dropout)        (None, 250, 256)          0

 dense_98 (Dense)            (None, 250, 128)          32896

 dense_99 (Dense)            (None, 250, 57)           7353

=================================================================
Total params: 4,766,841
Trainable params: 4,766,841
Non-trainable params: 0
_________________________________________________________________
None
Epoch 1/10
447/447 [==============================] - 38s 81ms/step - loss: 0.0020 - accuracy: 0.1458 - val_loss: 0.0015 - val_accuracy: 0.1784
Epoch 2/10
447/447 [==============================] - 35s 78ms/step - loss: 0.0015 - accuracy: 0.1767 - val_loss: 0.0015 - val_accuracy: 0.1779
Epoch 3/10
447/447 [==============================] - 35s 78ms/step - loss: 0.0015 - accuracy: 0.2069 - val_loss: 0.0015 - val_accuracy: 0.2589
Epoch 4/10
447/447 [==============================] - 35s 78ms/step - loss: 0.0015 - accuracy: 0.2623 - val_loss: 0.0014 - val_accuracy: 0.2808
Epoch 5/10
447/447 [==============================] - 35s 78ms/step - loss: 0.0014 - accuracy: 0.2851 - val_loss: 0.0014 - val_accuracy: 0.3028
Epoch 6/10
447/447 [==============================] - 35s 78ms/step - loss: 0.0014 - accuracy: 0.3034 - val_loss: 0.0014 - val_accuracy: 0.3152
Epoch 7/10
447/447 [==============================] - 34s 77ms/step - loss: 0.0014 - accuracy: 0.3182 - val_loss: 0.0014 - val_accuracy: 0.3381
Epoch 8/10
447/447 [==============================] - 35s 77ms/step - loss: 0.0014 - accuracy: 0.3413 - val_loss: 0.0014 - val_accuracy: 0.3575
Epoch 9/10
447/447 [==============================] - 35s 78ms/step - loss: 0.0014 - accuracy: 0.3643 - val_loss: 0.0014 - val_accuracy: 0.3824
Epoch 10/10
447/447 [==============================] - 35s 78ms/step - loss: 0.0014 - accuracy: 0.3853 - val_loss: 0.0014 - val_accuracy: 0.3985
```




```
14/14 [==============================] - 1s 52ms/step - loss: 0.0014 - accuracy: 0.3976
Test Loss: 0.0013581027742475271 Test Accuracy: 0.3975837230682373
```

شکل 7 - استفاده از لاس BinaryFocalCrossentropy

همانطور که دیده می‌شود، استفاده از این تابع باعث شد تا 10 اپاک برای آموزش کامل مدل کافی نباشد و نتواند به دقتی که توابع قبلی رسیدند برسد. باید اشاره کرد که این لاس و به طور

خاص بخش Focal اشاره به روشی دارد که در مقاله سال 2018 به آدرس https://arxiv.org/pdf/1708.02002.pdf آمده است که باعث می‌شود نمونه‌هایی که ساده تر هستند و مدل آنها را به خوبی یاد گرفته است (یعنی لوجیت ها خیلی شبیه وان هات درست شده اند) ارزش کمتری می‌یابند و تمرکز مدل بیشتر روی example هایی که کمتر یاد گرفته است می‌رود. همین موضوع می‌تواند از علل کندتر آموزش دیدن باشد چون بعد از کمی دیگر مواردی که ساده یاد گرفته شدند تاثیر کمتری در یادگیری پارامترها هم خواهند داشت و پارامترها به سرعت لاس های قبلی آموزش نمی‌یابند.

در مجموع هر کدام از این لاس ها ویژگی های خاص خود را دارند و باید در کنار هایپرپارامترهای دیگر مثل تعداد ایپاک یا learning rate های مختص خود سنجیده شوند. اما همانطور که نشان داده شد از همگی می‌توان برای آموزش این مسئله classification بهره برد.

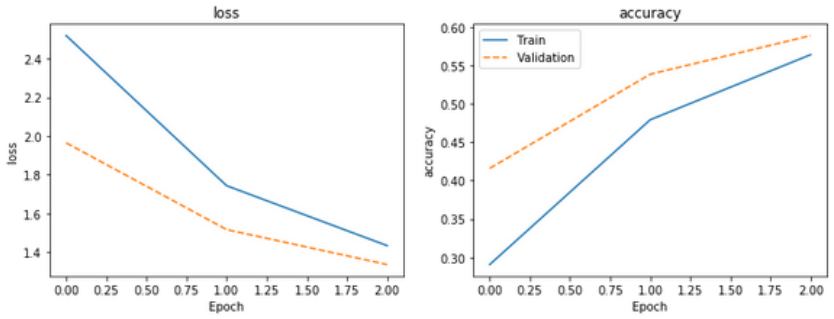
ج) با استفاده از 2 معیار متفاوت عملکرد مدل را بررسی کنید. (این معیارها میتواند شامل تعداد اپاک، بهینه ساز و ... باشد.)

در این بخش به تغییر تعداد اپاک و خود مدل (LSTM یا GRU) می‌پردازیم. برای اطمینان در آخر تغییر learning rate را نیز بررسی می‌کنیم.

ابتدا تعداد اپاک 3 و 5 قرار می‌دهیم و با 10 که قبل‌تر بود مقایسه می‌کنیم.

```
# EPOCH 3
trainer = RNN_Trainer(vocab_size, SEQ_LENGTH)
print(trainer.model.summary())
trainer.train(x_train, y_train, batch_size=8, epochs=3)
trainer.plot_history()
trainer.evaluate(x_test, y_test)
```

```
Model: "sequential_36"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_36 (Embedding)    (None, 250, 64)           3648

 lstm_36 (LSTM)              (None, 250, 1024)         4460544

 dense_100 (Dense)           (None, 250, 256)          262400

 dropout_36 (Dropout)        (None, 250, 256)          0

 dense_101 (Dense)           (None, 250, 128)          32896

 dense_102 (Dense)           (None, 250, 57)           7353

=================================================================
Total params: 4,766,841
Trainable params: 4,766,841
Non-trainable params: 0
_________________________________________________________________
None
Epoch 1/3
447/447 [==============================] - 37s 78ms/step - loss: 2.5196 - accuracy: 0.2906 - val_loss: 1.9649 - val_accuracy: 0.416
0
Epoch 2/3
447/447 [==============================] - 34s 77ms/step - loss: 1.7433 - accuracy: 0.4793 - val_loss: 1.5164 - val_accuracy: 0.538
6
Epoch 3/3
447/447 [==============================] - 34s 77ms/step - loss: 1.4335 - accuracy: 0.5641 - val_loss: 1.3358 - val_accuracy: 0.588
9
```



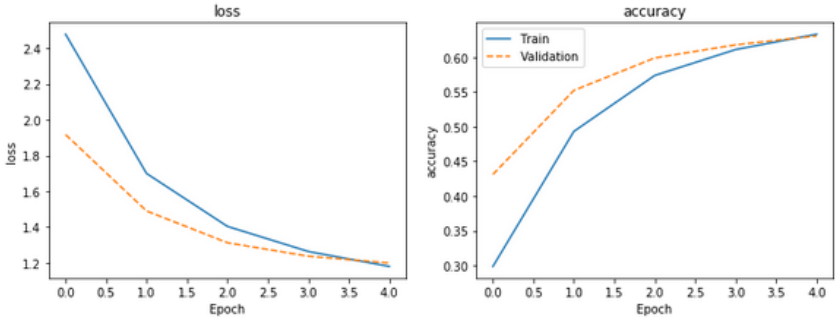

```
14/14 [==============================] - 1s 52ms/step - loss: 1.3318 - accuracy: 0.5896
Test Loss: 1.3317735195159912 Test Accuracy: 0.5896470546722412
```

شکل 8 - آموزش با 3 اپاک

همانطور که دیده می‌شود 3 اپاک هنوز کافی نبوده و هنوز جای برای بهبود دارد. ضمنا لاس و دقت به ترتیب برابر 1.33 و 0.5896 است که باید در مقابل اپاک 10 که در سوال های قبل دیدیم و به ترتیب برابر 1.20 و 0.6429 بود دید که مشخصا هنوز 3 اپاک کافی نیست.

```
# EPOCH 5
trainer = RNN_Trainer(vocab_size, SEQ_LENGTH)
print(trainer.model.summary())
trainer.train(x_train, y_train, batch_size=8, epochs=5)
trainer.plot_history()
trainer.evaluate(x_test, y_test)
```

```
Model: "sequential_37"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_37 (Embedding)    (None, 250, 64)           3648

 lstm_37 (LSTM)              (None, 250, 1024)         4460544

 dense_103 (Dense)           (None, 250, 256)          262400

 dropout_37 (Dropout)        (None, 250, 256)          0

 dense_104 (Dense)           (None, 250, 128)          32896

 dense_105 (Dense)           (None, 250, 57)           7353

=================================================================
Total params: 4,766,841
Trainable params: 4,766,841
Non-trainable params: 0
_________________________________________________________________
None
Epoch 1/5
447/447 [==============================] - 38s 80ms/step - loss: 2.4778 - accuracy: 0.2988 - val_loss: 1.9168 - val_accuracy: 0.431
1
Epoch 2/5
447/447 [==============================] - 34s 77ms/step - loss: 1.6999 - accuracy: 0.4931 - val_loss: 1.4903 - val_accuracy: 0.552
1
Epoch 3/5
447/447 [==============================] - 34s 77ms/step - loss: 1.4040 - accuracy: 0.5736 - val_loss: 1.3126 - val_accuracy: 0.598
8
Epoch 4/5
447/447 [==============================] - 34s 77ms/step - loss: 1.2637 - accuracy: 0.6109 - val_loss: 1.2375 - val_accuracy: 0.617
7
Epoch 5/5
447/447 [==============================] - 35s 78ms/step - loss: 1.1804 - accuracy: 0.6329 - val_loss: 1.1995 - val_accuracy: 0.630
6
```



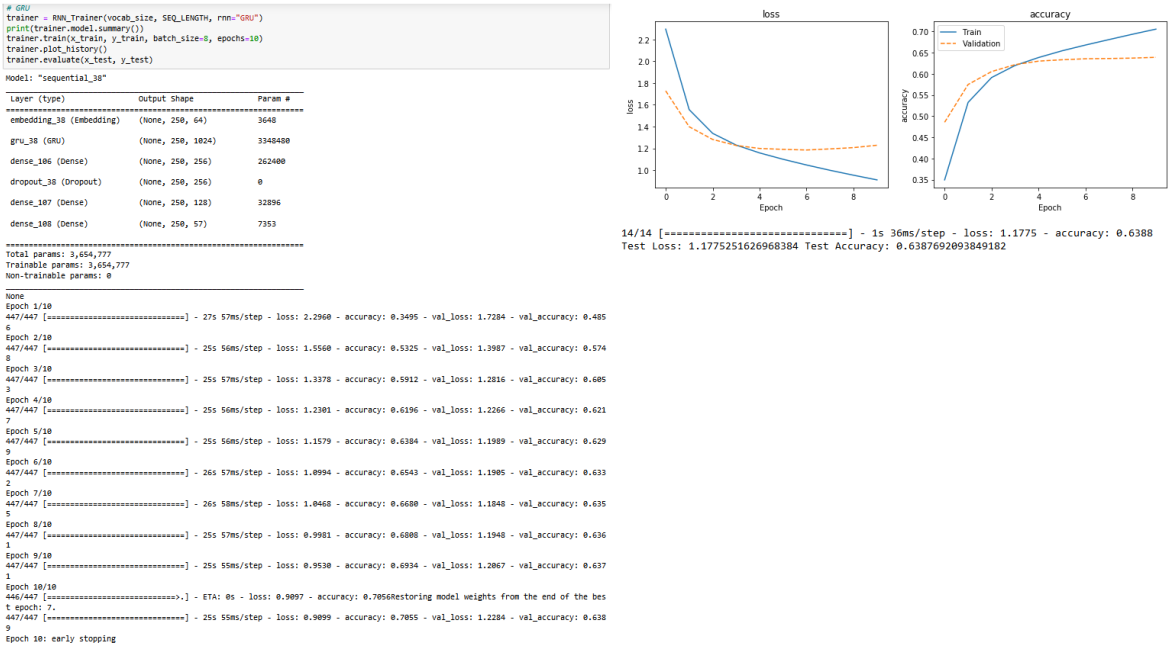

```
14/14 [==============================] - 1s 51ms/step - loss: 1.1919 - accuracy: 0.6322
Test Loss: 1.191850185394287 Test Accuracy: 0.6321991086006165
```

شکل 9 - آموزش با 5 ایپاک

با استفاده از 5 ایپاک می‌بینیم که لاس و دقت به ترتیب برابر 1.19 و 0.6321 شده است. در اینجا نکته جالبی را می‌بینیم که لاس کمتر شده است اما دقت نه. این می‌تواند به این دلیل باشد که مواردی که مدل درست می‌گوید با اطمینان بیشتری می‌گوید و لاس را کم می‌کند اما تعداد چیزهایی که درست تشخیص می‌دهد کمتر از اجرا با 10 ایپاک است. اما می‌توان در مجموع این 5 ایپاک را حالتی بین 3 و 10 دانست که هنوز کمی جا دارد تا بیشتر آموزش یابد.

و حالا مدل را به GRU تغییر می‌دهیم و تاثیر آن را بررسی می‌کنیم.

اولین نکته این است که GRU مدلی ساده تر نسبتا به LSTM دارد که در بخش بعدی بیشتر توضیح داده می‌شود. و چیزی که دیده می‌شود این است که با همان ابعاد سوال قبل، تعداد پارامتر از 4,766,841 در LSTM به 3,654,777 کاهش یافته و این مورد در زمان آموزش هم مشخص است که کاهش یافته است.



```
# GRU
trainer = RNN_Trainer(vocab_size, SEQ_LENGTH, rnn="GRU")
print(trainer.model.summary())
trainer.train(x_train, y_train, batch_size=8, epochs=10)
trainer.plot_history()
trainer.evaluate(x_test, y_test)
```

```
Model: "sequential_38"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_38 (Embedding)    (None, 250, 64)           3648

 gru_38 (GRU)                (None, 250, 1024)         3348480

 dense_106 (Dense)           (None, 250, 256)          262400

 dropout_38 (Dropout)        (None, 250, 256)          0

 dense_107 (Dense)           (None, 250, 128)          32896

 dense_108 (Dense)           (None, 250, 57)           7353

=================================================================
Total params: 3,654,777
Trainable params: 3,654,777
Non-trainable params: 0
_________________________________________________________________
None
Epoch 1/10
447/447 [==============================] - 27s 57ms/step - loss: 2.2960 - accuracy: 0.3495 - val_loss: 1.7284 - val_accuracy: 0.485
6
Epoch 2/10
447/447 [==============================] - 25s 56ms/step - loss: 1.5560 - accuracy: 0.5325 - val_loss: 1.3987 - val_accuracy: 0.574
8
Epoch 3/10
447/447 [==============================] - 25s 57ms/step - loss: 1.3378 - accuracy: 0.5912 - val_loss: 1.2816 - val_accuracy: 0.605
3
Epoch 4/10
447/447 [==============================] - 25s 56ms/step - loss: 1.2301 - accuracy: 0.6196 - val_loss: 1.2266 - val_accuracy: 0.621
7
Epoch 5/10
447/447 [==============================] - 25s 56ms/step - loss: 1.1579 - accuracy: 0.6384 - val_loss: 1.1989 - val_accuracy: 0.629
9
Epoch 6/10
447/447 [==============================] - 26s 57ms/step - loss: 1.0994 - accuracy: 0.6543 - val_loss: 1.1905 - val_accuracy: 0.633
2
Epoch 7/10
447/447 [==============================] - 26s 58ms/step - loss: 1.0468 - accuracy: 0.6680 - val_loss: 1.1848 - val_accuracy: 0.635
5
Epoch 8/10
447/447 [==============================] - 25s 57ms/step - loss: 0.9981 - accuracy: 0.6808 - val_loss: 1.1948 - val_accuracy: 0.636
1
Epoch 9/10
447/447 [==============================] - 25s 55ms/step - loss: 0.9530 - accuracy: 0.6934 - val_loss: 1.2067 - val_accuracy: 0.637
1
Epoch 10/10
446/447 [==============================>.] - ETA: 0s - loss: 0.9097 - accuracy: 0.7056Restoring model weights from the end of the bes
t epoch: 7.
447/447 [==============================] - 25s 55ms/step - loss: 0.9099 - accuracy: 0.7055 - val_loss: 1.2284 - val_accuracy: 0.638
9
Epoch 10: early stopping
```



```
14/14 [==============================] - 1s 36ms/step - loss: 1.1775 - accuracy: 0.6388
Test Loss: 1.1775251626968384 Test Accuracy: 0.6387692093849182
```

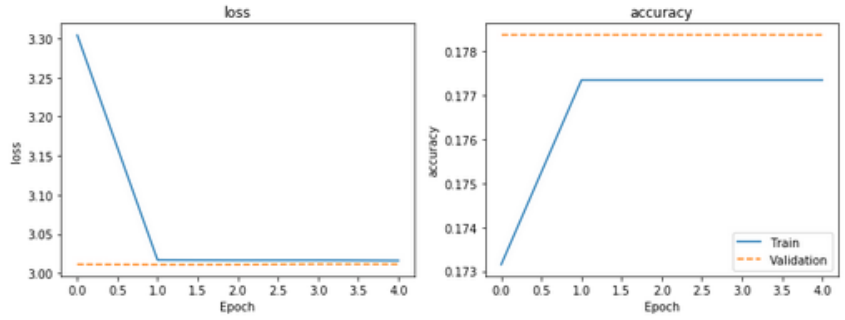
شکل 10 - آموزش با مدل GRU

در تصویر، آموزش مدل با GRU و نتیجه نهایی تست آن آمده است. دیده می‌شود که لاس و دقت تست به ترتیب برابر 1.17 و 0.6387 شده است. باز هم می‌بینیم که مدل از نظر دقت به مدل LSTM نتوانسته برسد که می‌توان آن را به علت تعداد پارامتر کمتر آن و ساده تر بودن عملیات داخلی آن دانست. اما خب از نظر سرعت می‌تواند گزینه مناسبی باشد.

در انتها برای اطمینان، مقدار lr روش adam را از حالت دیفالت 0.001 به 0.1 تغییر می‌دهیم تا تاثیر انتخاب lr مناسب را ببینیم.

```
# LR
trainer = RNN_Trainer(vocab_size, SEQ_LENGTH,
                      optimizer=tf.keras.optimizers.Adam(learning_rate=1e-1)
                     )
print(trainer.model.summary())
trainer.train(x_train, y_train, batch_size=8, epochs=10)
trainer.plot_history()
trainer.evaluate(x_test, y_test)
```

```
Model: "sequential_39"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_39 (Embedding)    (None, 250, 64)           3648

 lstm_39 (LSTM)              (None, 250, 1024)         4460544

 dense_109 (Dense)           (None, 250, 256)          262400

 dropout_39 (Dropout)        (None, 250, 256)          0

 dense_110 (Dense)           (None, 250, 128)          32896

 dense_111 (Dense)           (None, 250, 57)           7353

=================================================================
Total params: 4,766,841
Trainable params: 4,766,841
Non-trainable params: 0
_________________________________________________________________
None
Epoch 1/10
447/447 [==============================] - 38s 80ms/step - loss: 3.3041 - accuracy: 0.1731 - val_loss: 3.0113 - val_accuracy: 0.1784
Epoch 2/10
447/447 [==============================] - 34s 77ms/step - loss: 3.0169 - accuracy: 0.1773 - val_loss: 3.0110 - val_accuracy: 0.1784
Epoch 3/10
447/447 [==============================] - 34s 76ms/step - loss: 3.0165 - accuracy: 0.1773 - val_loss: 3.0111 - val_accuracy: 0.1784
Epoch 4/10
447/447 [==============================] - 35s 79ms/step - loss: 3.0165 - accuracy: 0.1773 - val_loss: 3.0117 - val_accuracy: 0.1784
Epoch 5/10
447/447 [==============================] - ETA: 0s - loss: 3.0161 - accuracy: 0.1773Restoring model weights from the end of the best epoc
h: 2.
447/447 [==============================] - 35s 79ms/step - loss: 3.0161 - accuracy: 0.1773 - val_loss: 3.0115 - val_accuracy: 0.1784
Epoch 5: early stopping
```




```
14/14 [==============================] - 1s 52ms/step - loss: 3.0160 - accuracy: 0.1773
Test Loss: 3.0159547328948975 Test Accuracy: 0.17733831511306763
```

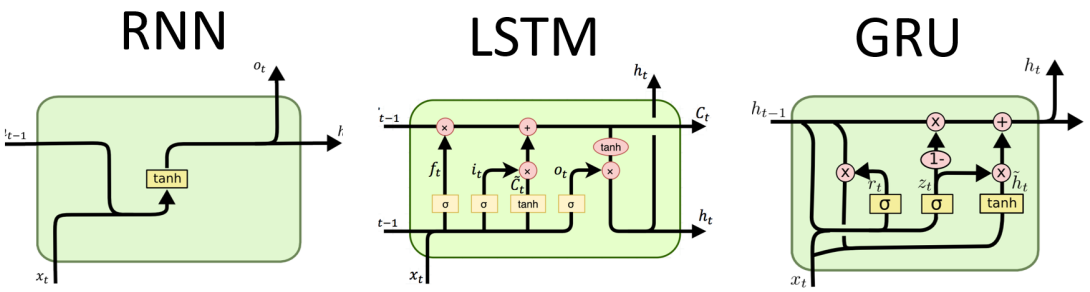
شکل 11 - آموزش با نرخ یادگیری 0.1

در شکل بالا، یادگیری با نرخ یادگیری 0.1 به جای 0.001 دیده می‌شود. همانطور که دیده می‌شود دیگر یادگیری به درستی انجام نمی‌شود و مدل نمی‌تواند اصلا به خوبی همگرا شود چون انقدر نرخ بزرگ است که به جای نزدیک شدن به مینیمم لاس، اصطلاحا overshoot داریم و اصلا نمی‌توان دیگر با ریزدانگی کافی یادگیری داشت. بنابراین انتخاب learning rate مناسب نیز از موارد مهمی است که در بخشهای قبلی به خوبی انجام شده بود.

د) چگونه حافظه سلولهای عصبی استفاده شده در مدل شما در عملکرد مدل موثر است.

داده متن را می‌توان یک داده در طول زمان دانست و در این مسئله هر کاراکتر در یک زمان آمده است. اگر می‌خواستیم یک مدل ثابت برای کل متن داشته باشیم همیشه ورودی باید سایز ثابتی می‌شد و ضمنا مدل خیلی بزرگ می‌شد چون باید مثلا به ازای تمام طول زمان پارامتر می‌داشت. سلولهای حافظه باعث می‌شوند نیاز به مدل بزرگ نباشد و هر بار صرفا روی یک زمان اجرا می‌شوند و همچنین با ارسال hidden state به زمان بعدی باعث می‌شوند که مدل بتواند خلاصه‌ای از گذشته را نیز داشته باشد. به این ترتیب برای این مسئله بهترین روش استفاده از مدل‌هایی است که مخصوص ترتیب زمانی ساخته شده اند.

در شکل زیر ساختار معماری RNN و GRU و LSTM آمده است.



شکل 12 - ساختار مدلهای حافظه محور زمانی

معماری LSTM سه گیت دارد. گیت سمت چپ forget gate است که یک تابع سیگموید دارد و با ضرب خروجی آن که بین 0 و 1 است، مشخص می‌کند که محتویات cell state گذشته چقدر نگهداری شود. (توجه داشته باشید که این مقدار یک عدد نیست و به ازای هر بعد این عدد وجود دارد و می‌تواند بعضی ابعاد را فراموش و بعضی را نگه دارد.) گیت وسط input gate است که مشخص می‌کند چه مقدار از ورودی جدید وارد cell state شود که با ضرب سیگموید در مقدار ورودی که از tanh رد شده و در انتها جمع آن با cell state انجام می‌شود. در سمت راست output gate وجود دارد که ترکیب cell state و hidden state را مشخص می‌کند که در hidden state خروجی قرار گیرد که به زمان بعدی و همچنین به عنوان خروجی این زمان داده می‌شود.

معماری GRU مهمترین تفاوتی که دارد این است که به جای دو خط cell state و hidden state فقط یک خط انتقال hidden state دارد. گیت سمت چپ reset gate است که با

ضرب خروجی بعد از سیگموید در h مرحله قبلی تاثیر آن را در ترکیب با ورودی جدید مشخص می‌کند. سمت راست نیز update gate است که خروجی بعد از سیگموید را همزمان استفاده می‌کند تا میزان اضافه کردن مقدار جدید بر روی h را مشخص کند و همچنین با استفاده از معکوس آن و ضربش در h قبلی تاثیر آن را کاهش می‌دهد. در مجموع GRU نسبت به LSTM محاسبات کمتری دارد و با سرعت بیشتری می‌تواند اجرا شود، و در مقابل آن، پیچیدگی های بیشتر LSTM پتانسیل یادگیری بیشتر و انعطاف بیشتر در انتقال اطلاعات را دارد.

با این توضیحات الان روشن تر است که این مدل ها می‌توانند توالی کاراکترهای زمان 0 تا t-1 را در حافظه خود داشته باشند و بخشهای مهم را نگه دارند و یا بخشهایی که دیگر کاربرد ندارند را forget کنند و در نهایت به خوبی گزینه های مناسب را برای زمان t را خروجی دهند. در صورتی که اگر همچین گزینه ای نداشتیم مثلا به ngram با n خاصی محدود می‌شدیم و نمی‌توانستیم long dependency ها را به خوبی بازنمایی کنیم و در نهایت مثلا 3gram داشتیم که فقط 3 کاراکتر قبلی را داشت. ولی الان کل زمانهای قبلی را داریم. (البته باید اشاره کرد که از مشکلات این روشهای جدای از کند بودن به علت مشکل در اجرای موازی، به خوبی نگه نداشتن long dependency ها در متن‌های پیچیده تر است که مدلهای transformerای مانند BERT که از روش self attention استفاده می‌کنند هر دوی این مشکلات را برطرف می‌کنند.)

سوال 3 - Contextual Embedding + RNNs

1 -چه پیش پردازش هایی روی داده ها صورت گرفته است؟ نام ببرید و در خصوص هر یک توضیح مختصری ارائه دهید (اگر به نظرتان پیش پردازش های دیگری نیز برای دادگان نیاز است، آن ها را نام برده و اعمال کنید).

اولین پیش‌پردازش که دیده می‌شود تغییر منشن ها مثل @mayasolovely به <user> است. علت این کار این است که برای تشخیص توهین نباید خود اسم ها و ای دی ها مهم باشند، مثلا ممکن است یک یوزر همیشه توییت های توهین آمیز داشته باشد و مدل یاد می‌گیرد و هر چه در آن یوزر خاص باشد را به صورت بایاس شده توهین تشخیص می‌دهد. ضمنا وجود id ها یک ویژگی خاص و گذرا است. از این نظر که در این دیتاست یک سری ای دی خاص وجود دارد و اگر مدل با آن‌ها آموزش یابد بعدا برای استفاده از محیط production که دیگر این ای دی ها کمترند و ای دی های دیگری هستند می‌تواند مشکل ایجاد کند و توزیع آموزش با محیط تست متفاوت شود. (ضمنا استفاده از این ای دی ها مشکلات حریم شخصی هم ممکن است داشته باشد.)

پیش‌پردازش دیگر این است که ایموجی ها مثل 😂 مقدار <emotion> گذاشته شده است. این مورد هم چون اصل ایموجی ها توالی کاراکترهای خاصی هستند و احتمال زیاد مدل روی داده‌ای که در ایموجی بوده آموزش ندیده است باید انجام شود. وگرنه ایموجی یک سری اطلاعات گنگ برای مدل خواهد بود. البته اینکه هر ایموجی به یک مقدار برسد خیلی کار خوبی نیست و اگر به جای آن اسم خاص آن ایموجی مثل خنده گذاشته می شد می‌توانست بیشتر به مدل کمک کند.

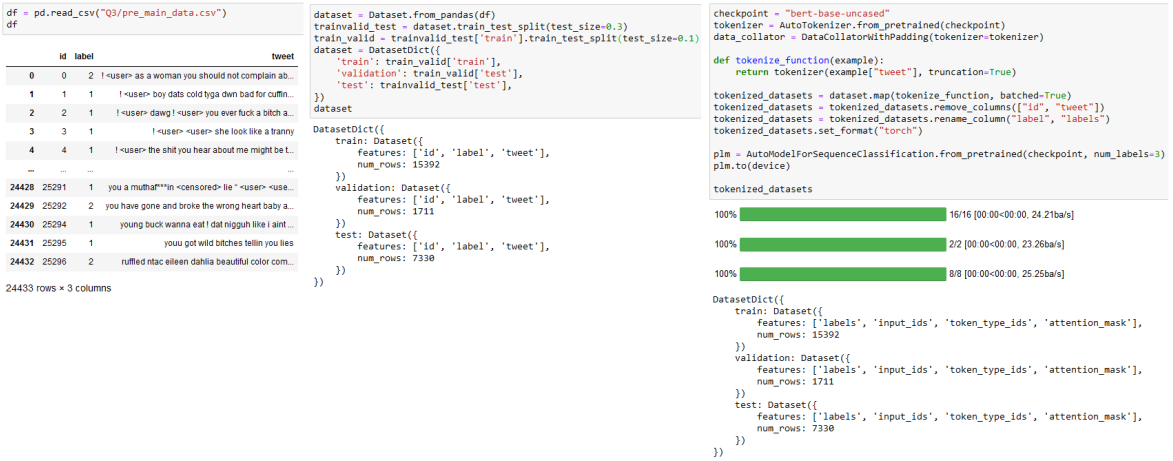
ضمنا لینکها نیز به <url> مپ شده اند و باز هم خوب است جون متنی که در یک url است می‌تواند برای مدل گمراه کننده باشد. (البته اگر مدل در پیش‌آموزش خود لینک هم دیده باشد شاید خوب باشد ولی احتمالا مدلهایی که داریم اینطور نیستند و اطلاع خاصی درباره لینک ها ندارند.)

همچنین کاراکترهای اضافه مثل " یا & یا # که ارزش معنایی برای این تسک ندارند حذف شده‌اند.

و البته تمام کاراکتر ها به حالت lowercase تغییر داده شده‌اند. که البته این انتخاب کمی قابل بحث است چون مدل‌های قوی‌تر ممکن است بتوانند از همان Case هم اطلاعاتی به دست آورند. مثل اینکه GOOD با good بار معنایی متفاوتی دارند. البته اگر مثلا از مدل bert-uncased استفاده شود طبیعتا تفاوتی نیست و در نهایت مدل اصلا case نداشته. ولی شاید مدلی که کیس داشته باشد در این تسک کمک کند ولی این مورد باید امتحان شود و نمی‌توان همینطوری به صورت قطعی گفت کیس در این مسئله کمک می‌کند یا ضرر دارد. (اما طبیعتا lowercase کردن ورودی ها را ساده تر می‌کند و مدلهایی که سبکترند راحت تر می‌توانند کار کنند و بحثهای قبلی مربوط به مدلی بود که قوی باشد و بتواند بهره ببرد از آن اطلاعات)

2 -به کمک bert و یکی از ماژول های حافظه (lstm، rnn یا gru) یک مدل طراحی کنید و روی داده ها آموزش دهید. نمودار دقت، نمودار loss و ماتریس آشفتگی را رسم کنید.

ابتدا خواندن داده را بررسی می‌کنیم.



```
df = pd.read_csv("Q3/pre_main_data.csv")
df
```

	id	label	tweet
0	0	2	! <user> as a woman you should not complain ab...
1	1	1	! <user> boy dats cold tyga dwn bad for cuffin...
2	2	1	! <user> dawg ! <user> you ever fuck a bitch a...
3	3	1	! <user> <user> she look like a tranny
4	4	1	! <user> the shit you hear about me might be t...
...	...	...	...
24428	25291	1	you a muthaf**in <censored> lie " <user> <use...
24429	25292	2	you have gone and broke the wrong heart baby a...
24430	25294	1	young buck wanna eat ! dat nigguh like i aint ...
24431	25295	1	youu got wild bitches tellin you lies
24432	25296	2	ruffled ntac eileen dahlia beautiful color com...

24433 rows × 3 columns

```
dataset = Dataset.from_pandas(df)
trainvalid_test = dataset.train_test_split(test_size=0.3)
train_valid = trainvalid_test['train'].train_test_split(test_size=0.1)
dataset = DatasetDict({
    'train': train_valid['train'],
    'validation': train_valid['test'],
    'test': trainvalid_test['test'],
})
dataset
```

```
DatasetDict({
    train: Dataset({
        features: ['id', 'label', 'tweet'],
        num_rows: 15392
    })
    validation: Dataset({
        features: ['id', 'label', 'tweet'],
        num_rows: 1711
    })
    test: Dataset({
        features: ['id', 'label', 'tweet'],
        num_rows: 7330
    })
})
```

```
checkpoint = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)

def tokenize_function(example):
    return tokenizer(example["tweet"], truncation=True)

tokenized_datasets = dataset.map(tokenize_function, batched=True)
tokenized_datasets = tokenized_datasets.remove_columns(["id", "tweet"])
tokenized_datasets = tokenized_datasets.rename_column("label", "labels")
tokenized_datasets.set_format("torch")

plm = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=3)
plm.to(device)

tokenized_datasets
```

```
100% 16/16 [00:00<00:00, 24.21ba/s]
100% 2/2 [00:00<00:00, 23.26ba/s]
100% 8/8 [00:00<00:00, 25.25ba/s]

DatasetDict({
    train: Dataset({
        features: ['labels', 'input_ids', 'token_type_ids', 'attention_mask'],
        num_rows: 15392
    })
    validation: Dataset({
        features: ['labels', 'input_ids', 'token_type_ids', 'attention_mask'],
        num_rows: 1711
    })
    test: Dataset({
        features: ['labels', 'input_ids', 'token_type_ids', 'attention_mask'],
        num_rows: 7330
    })
})
```

شکل 1 - بارگذاری داده و توکنایز کردن آن

در تصویر مشخص است که ابتدا داده ها بارگذاری می‌شوند و سپس با استفاده از train_test_split اول 30 درصد داده را برای تست کنار می‌گذاریم. (چون دیتاست نسبتا کوچک است درصد بیشتری از دیتا لازم است تا عملکرد مدل به خوبی سنجیده شود و دیتای

بخش تست کم نباشد.) و از 70 درصد باقیمانده 10 درصد برای validation و 90 درصدش را برای آموزش می‌گذاریم که تعداد دقیق هر بخش در تصویر هم آمده است.

سپس توکنایزر و مدل برت را بارگذاری کردیم و دیتاست را با آن توکنایز کردیم. این کار موارد مورد نیاز برای اجرا را می‌سازد مثل input_ids که عدد توکن های متن است. ضمنا بعد این کار ستون های بی استفاده در مدل مثل متن توییت را حذف می‌کنیم و در نهایت به حالت تنسور های تورچ داده را منتقل می‌کنیم تا با تورچ به آموزش بپردازیم.

حالا مدل طراحی شده را بررسی می‌کنیم.

```
class Head(torch.nn.Module):
        def __init__(self, input_size=768, num_classes=3):
        super().__init__()
        self.lstm = torch.nn.LSTM(input_size=input_size,
                                  hidden_size=384, num_layers=1,
                                  bidirectional=True, batch_first=True).to(device)
        net_list = [
        torch.nn.Linear(768, 512),
        torch.nn.Tanh(),
        torch.nn.LayerNorm(512),
        torch.nn.Dropout(0.1),
        torch.nn.Linear(512, num_classes)
        ]
        self.label_net = torch.nn.Sequential(*net_list).to(device)
        self.training_criterion = torch.nn.CrossEntropyLoss()
        self.optimizer = torch.optim.Adam(self.parameters(), lr=5e-4, weight_decay=0)
        self.to(device)

        def forward(self, plm_last_hidden_states):  # ~[8, 34, 768]
        x, (hn, cn) = self.lstm(plm_last_hidden_states)
        x = x[:, -1, :]  # Last LSTM
        x = self.label_net(x)
        return x
```

کد 1 - هد اضافه شده روی بازنمایی های برت

در کد بالا، هدی که روی بازنمایی های برت اجرا می‌کنیم و آموزش می‌یابد دیده می‌شود. همانطور که خواسته شده بود از یک مدل LSTM استفاده کردیم و سپس خروجی آن‌را به یک FFN یا MLP می‌دهیم و در انتها به تعداد کلاس ها که 3 است نورون داریم. برای اپتیمایزر از adam استفاده می‌کنیم و لاس مدل را کراس انتروپی انتخاب کردیم. همچنین از لایه های dropout نیز استفاده کردیم تا از overfitting با خاموش کردن رندوم بعضی از نورون ها در هر مرحله جلوگیری شود.

```
class Trainer:
        def __init__(self, plm, tokenized_datasets):
        self.plm = plm.to(device)
        self.tokenized_datasets = tokenized_datasets
        self.head = Head().to(device)
        self.dataloaders = {}
```

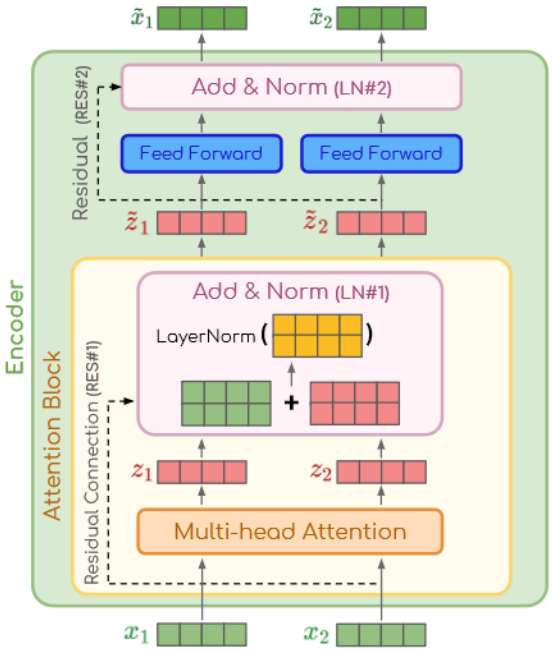
```
            self.dataloaders["train"] = DataLoader(
            tokenized_datasets["train"], shuffle=True, batch_size=8, collate_fn=data_collator
            )
            self.history = {"loss": {"train": [], "validation": [], "test": []},
                        "accuracy": {"train": [], "validation": [], "test": []}}
            print(self.head)


            def train(self, epochs=5):
            for epoch in tqdm(range(epochs)):
            plm.eval()
            self.head.train()
            running_loss = 0.0
            steps = 0
            for batch in tqdm(self.dataloaders["train"]):
                        batch = {k: v.to(device) for k, v in batch.items()}
                        outputs = self.plm(**batch, output_hidden_states=True, return_dict=True)
                        hidden_states = torch.stack([val.detach() for val in outputs.hidden_states])  # ~[13, 8, 34, 768]
                        last_hidden_states = hidden_states[-1].to(device)
                        output = self.head(last_hidden_states)
                        loss = self.head.training_criterion(output.to(device), torch.tensor(batch["labels"], dtype=torch.long))
                        loss.backward()
                        self.head.optimizer.step()
                        self.head.optimizer.zero_grad()

                        running_loss += loss.item()
                        steps += 1
            self.update_history(epoch)
            self.plot_history()

            def calc_loss(self, tokenized_dataset, print_metrics=False, desc=""):
            dataloader = DataLoader(
            tokenized_dataset, batch_size=8, collate_fn=data_collator
            )
            self.head.eval()
            with torch.no_grad():
            running_loss = 0
            steps = 0
            preds = None
            for batch in tqdm(dataloader, desc=desc):
                        batch = {k: v.to(device) for k, v in batch.items()}
                        # forward
                        outputs = self.plm(**batch, output_hidden_states=True, return_dict=True)
                        hidden_states = torch.stack([val.detach() for val in outputs.hidden_states])  # ~[13, 8, 34, 768]
                        last_hidden_states = hidden_states[-1].to(device)
                        output = self.head(last_hidden_states)
                        preds = output if preds == None else torch.cat((preds, output), 0)
                        loss = self.head.training_criterion(output.to(device),
                                                            torch.tensor(batch["labels"].clone().detach(), dtype=torch.long))
                        running_loss += loss.item()
                        steps += 1
            preds = preds.cpu().argmax(-1)
            y_true = np.array(tokenized_dataset["labels"])
            accuracy = sklearn.metrics.accuracy_score(y_true, preds)
            report = classification_report(y_true, preds)
            if print_metrics:
            print(report)
            self.plot_cm(y_true, preds)
            return running_loss / steps, accuracy

            def update_history(self, epoch, train_loss = None):
            for part in ["train", "validation", "test"]:
            loss, accuracy = self.calc_loss(self.tokenized_datasets[part], desc=f"{part} loss", print_metrics=part=="test")
            self.history["loss"][part].append(loss)
            self.history["accuracy"][part].append(accuracy)
            print(f"[Epoch {epoch + 1}] loss: {self.history['loss']['train'][-1]}, val_loss:
{self.history['loss']['validation'][-1]}")


            def plot_history(self):
            for metric in ["loss", "accuracy"]:
            loss_history = self.history[metric]
            plt.plot(loss_history["train"])
            plt.plot(loss_history["validation"])
            plt.title(f"{metric} history")
            plt.legend(['Train', 'Validation'])
            plt.show()
            print("TEST RESULTS:")
            print("Test Loss:     ", self.history["loss"]["test"][-1])
            print("Test Accuracy:", self.history["accuracy"]["test"][-1])

            def plot_cm(self, y_true, preds):
            cm = confusion_matrix(y_true, preds)
```

```
            plt.figure(figsize=(7, 5))
            ax = sns.heatmap(cm, annot=True, fmt="d")
            bottom, top = ax.get_ylim()
            ax.set_ylim(bottom + 0.5, top - 0.5)
            plt.title('Confusion Matrix')
            plt.ylabel('True Label')
            plt.xlabel('Predicted Label')
            plt.show()

trainer = Trainer(plm, tokenized_datasets)
trainer.train(epochs=3)
```

کد 2 - آموزش و ارزیابی

در کد بالا آموزش مدل، ارزیابی و کشیدن نمودارهای خواسته شده آمده است. برای آموزش مدل ابتدا Pretrained Language Model اجرا می‌شود و خروجی های hidden state آن که همان بازنمایی ها در تمام لایه ها است گرفته می‌شود. سپس خروجیهای لایه آخر را به هدی که بالاتر آورده شد می‌دهیم و در انتهایش سه نورون نهایی را می‌گیریم.

ساختار برت که برای بازنمایی اولیه استفاده کردیم را توضیح می‌دهیم. همانطور که می‌دانیم لایه 0 همان لایه امبدینگ است که از وان هات ووکب به بازنمایی 768 (در مدل های بیس) بعدی می‌برد. در ادامه اولین لایه از 12 لایه encoder را می‌بینیم که از بخشهای self attention و Add&Norm و فیدفوروارد و یک Add&Norm دیگر تشکیل شده است. در بخش اول با استفاده از روش attention که در مقاله transformer توضیح داده شده است به شکل استفاده از q, k, v توجه هر توکن به توکنهای دیگر را به دست می‌آوریم. در بخشهای بعدی دو residual connection داریم که ورودی را با خروجی جمع می‌کنند که نشان داده شده است به مدل کمک می‌کند و همچنین از مشکل vanishing gradients جلوگیری می‌کند. همچنین مدل dense هم داریم که اگر ترکیب خاصی لازم بود بتواند یاد بگیرد و اجرا کند که تفسیرپذیری سخت تری دارد.

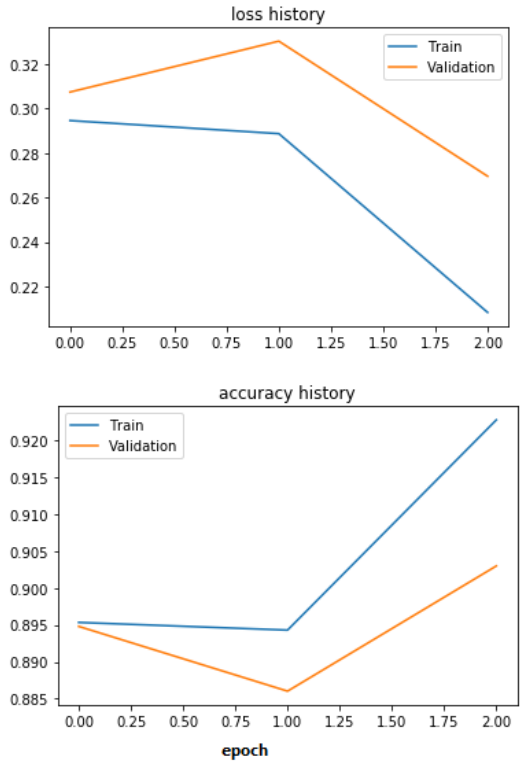


شکل 2 - بخش encoder ترنسفورمرها (برگرفته از https://arxiv.org/pdf/2205.03286.pdf)

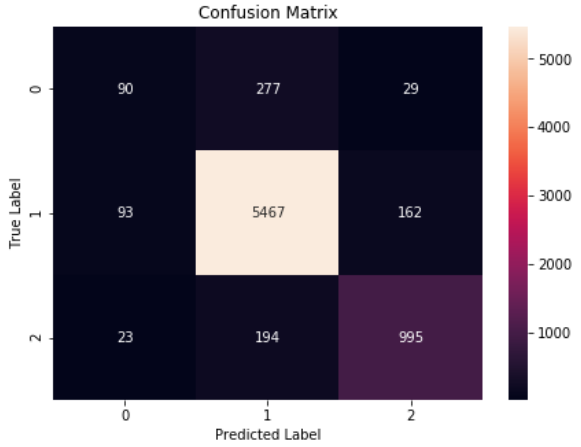
در شکل بالا بخشهایی که توضیح داده شد از شکل مقاله https://arxiv.org/pdf/2205.03286.pdf آمده است.

نتیجه اجرا در زیر آمده است.

```
[Epoch 3] loss: 0.20837957021928433, val_loss: 0.26956581089985077
```



	precision	recall	f1-score	support
0	0.44	0.23	0.30	396
1	0.92	0.96	0.94	5722
2	0.84	0.82	0.83	1212
accuracy			0.89	7330
macro avg	0.73	0.67	0.69	7330
weighted avg	0.88	0.89	0.89	7330



شکل 3 - نمودار دقت، لاس و ماتریس آشفتگی

در بالا نمودار خطا دیده می‌شود که به درستی رو به کاهش است. (به علت محدودیت زمان و منابع محاسباتی به 3 ایپاک اجرا کفایت کردیم.) همچنین دقت نیز به درستی رو به افزایش است. در ماتریس آشفتگی هم دیده می‌شود کلاس 1 و 2 بهتر تشخیص داده شده‌اند و کلاس 0 که خیلی ساپورت کمی داشته کمی بدتر عمل می‌کند. ضمنا مقادیر precision و recall و f1 و دقت نیز در شکل آمده است که نشان از آموزش مناسب مدل است.

Test Loss: 0.2809

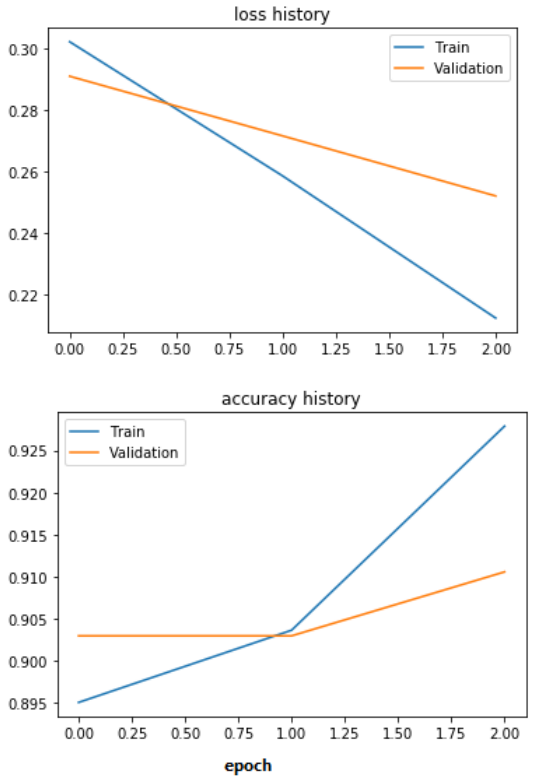
Test Accuracy: 0.8938

مقدار لاس و دقت تست در بالا نیز با اعشار دقیق تر آمده است.

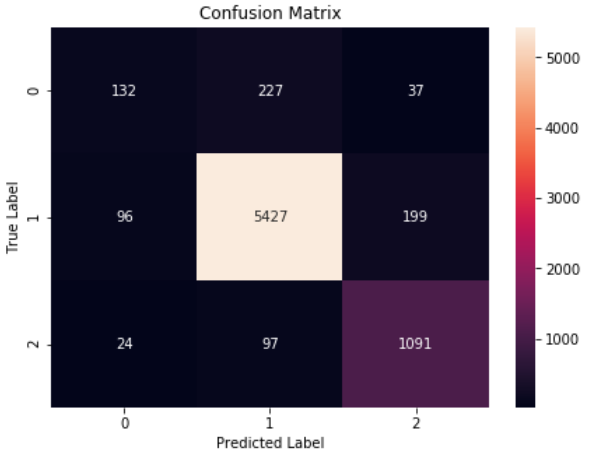
3 -در قسمت 2 ،به جای bert از hatebert استفاده کنید.

در این بخش با همان کد بخش قبل آزمایش را انجام می‌دهیم تنها با این تفاوت که مدل را از برت به GroNLP/hateBERT تغییر می‌دهیم.

```
[Epoch 3] loss: 0.21240579836378395, val_loss: 0.2520149628170461
```



	precision	recall	f1-score	support
0	0.52	0.33	0.41	396
1	0.94	0.95	0.95	5722
2	0.82	0.90	0.86	1212
accuracy			0.91	7330
macro avg	0.76	0.73	0.74	7330
weighted avg	0.90	0.91	0.90	7330



شکل 3 - نمودار دقت، لاس و ماتریس آشفتگی برای هیت برت

در شکل بالا نمودار لاس و دقت و ماتریس آشفتگی که خواسته شده بود آمده است. این بار دقتها همگی بهتر شدند و به دقت 91 درصد رسیدیم. مقدار دقیق در پایین آمده است.

Test Loss: 0.2576

Test Accuracy: 0.9072

توضیح علت این اتفاق در سوال بعد آمده است.

4 -نتایج قسمت 2 و 3 را با هم مقایسه کرده و علت طراحی شدن مدل هایی نظیر hatebert را توضیح دهید.

همانطور که دیده شد، دقتهای مدل hatebert بهتر از مدل برت عادی بود (91 درصد و 89 درصد). برای فهمیدن علت باید نحوه آموزش را در نظر گرفت. برت عادی روی دیتای کتاب و ویکیپدیا فقط آموزش دیده است که متن‌های نسبتا تمیز و مودبانه‌ای محسوب می‌شوند. بنابراین برای همان دیتا هم می‌توان به شکل خوبی عمل کند.

اما مدل hatebert که در مقاله https://arxiv.org/abs/2010.12472 آمده است روی دیتاستی آموزش دیده که از بخشهایی از Reddit بوده که به علت offensive, abusive hateful بودن بن شده‌اند. به همین دلیل به خوبی این مدل هم با ووکب این نوع صحبت، و همچنین جملات و مصداق‌های آن بسیار آشنا است. به همین دلیل همانطور که در مقاله نوشته شده و همچنین آزمایش ما هم تایید کرد، برای این مسئله که تشخیص تنفر است توانست بهتر از مدل عادی برت عمل کند. لازم به ذکر است که این موضوع بزرگتر از فقط تنفر است. مثلا در مقاله https://aclanthology.org/2020.acl-main.740/ آمده است که ادامه دادن pre-training روی همان موضوعات خاصی که می‌خواهیم مدل را آموزش دهیم مثل متن‌های پزشکی یا غیره می‌تواند بهبود ایجاد کند بنابراین در کل روی هر حوزه خاص اگر پیش‌آموزش مدل جامع تر باشد باعث بهبود دقت نهایی مدل می‌شود که از آشنایی بیشتر مدل با آن حوزه خاص ناشی می‌شود.

5) -امتیازی) به جای استفاده از bert در قسمت 2 ،از مدل T5 استفاده کنید. ویژگی منحصر به فرد این مدل نسبت به مدل های قبل چیست؟

مدل T5 جزء دسته مدل‌های sequence to sequence است. به این معنی که هم یک بخش encoder و هم بخش decoder دارد. مدل های بخش قبل که برت بودند یا مدلهای دیگر مشابه آن مثل ELectra, Roberta, albert, XLNet همگی فقط لایه‌های encoder دارند و ورودی را به فضای مخفی می‌برند. اما در مدلهای sequence to sequence مثل ترنسفورمر اولیه و اخیرا مدلهای t5 یا BART هم بخش encoder وجود دارد و سپس بخش decoder روی آن می‌آید تا بتواند متن هم تولید بکند. موارد کاربرد آن مانند ترجمه، خلاصه سازی متن و ... است. یک ویژگی خاص که مدل T5 دارد این است که تمام تسک های NLP را به صورت مسئله متن به متن می‌بیند. مثلا ورودی می‌تواند متن

.translate English to German: The house is wonderful

باشد به این صورت که توضیح تسک و سپس متن آمده است. روی دیتاستهای دیگری از GLUE هم این مدل آموزش دیده که مثلا می‌تواند تسک inference را انجام دهد اما همه به صورت متن به متن. این اتفاق نشان داده شده که می‌تواند مدلی نسبتا جامع بسازد.

برای اینکه کد ما از مدلهای sequence to sequence هم پشتیبانی کند قطعه کد زیر اضافه شد.

```
if self.seq_to_seq:
    outputs = self.plm(input_ids=batch["input_ids"], decoder_input_ids=batch["input_ids"], output_hidden_sta
    encoder_hidden_states = torch.stack([val.detach() for val in outputs.encoder_hidden_states])
    decoder_hidden_states = torch.stack([val.detach() for val in outputs.decoder_hidden_states])
    last_hidden_states = decoder_hidden_states[-1].to(device)
else:
    outputs = self.plm(**batch, output_hidden_states=True, return_dict=True)
    hidden_states = torch.stack([val.detach() for val in outputs.hidden_states])  # ~[13, 8, 34, 768]
    last_hidden_states = hidden_states[-1].to(device)
```

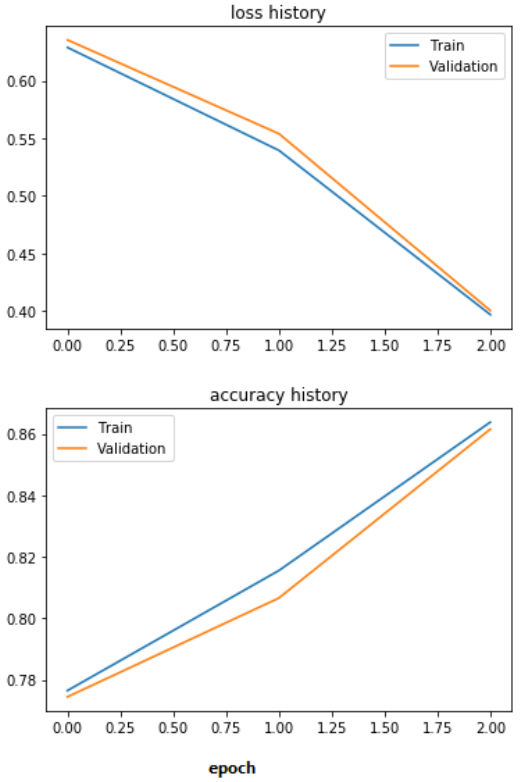
شکل 4 - کد گرفتن استیتها از مدل T5

همانطور که دیده می‌شود دیگر فقط یک بخش نیست و هم hidden state ها برای encoder هست و هم برای لایه های بعدی یعنی decoder. ضمنا ورودی هم دوتا است، یکی که به encoder داده می‌شود و دیگری که به decoder داده می‌شود. مثلا در زمان آموزش به صورت teacher forcing ورودی encoder می‌تواند متن باشد و ورودی decoder یکی شیفت خورده باشد. در این مورد در مقاله ترنسفورمر اولیه توضیح داده شده است.

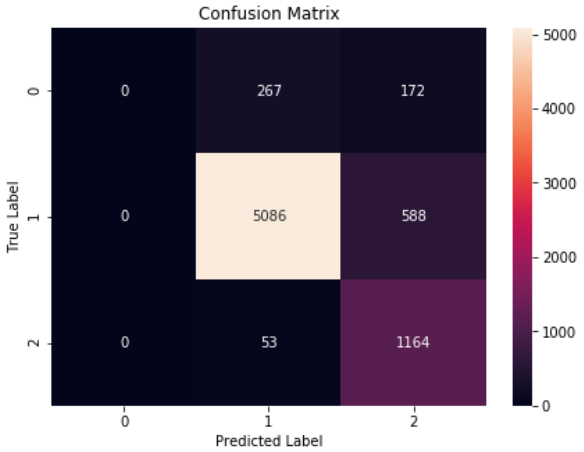
ضمنا به علت محدودیت ها، از مدل t5-small استفاده می‌کنیم که ابعادش 512 است. (مدل بیس روی گرافیک من جا نشد).

نتایج در زیر آمده است.

```
[Epoch 3] loss: 0.3970636044805114, val_loss: 0.4006013709738433
```



	precision	recall	f1-score	support
0	0.00	0.00	0.00	439
1	0.94	0.90	0.92	5674
2	0.60	0.96	0.74	1217
accuracy			0.85	7330
macro avg	0.52	0.62	0.55	7330
weighted avg	0.83	0.85	0.83	7330



شکل 5 - نتایج مدل t5

همانطور که دیده می‌شود در نمودارها، خطا به خوبی در حال کاهش و دقت به خوبی رو به افزایش است و تا ایپاک سوم به 85 درصد رسیده که باز هم بخاطر محدودیت زمان و منابع بیشتر آموزش ندادیم. اما باید دقت داشت که این مدل ابعاد بازنمایی 512 دارد در مقابل 768 مدلهای قبلی و همچنین وزن کل پارامترهایش حدودا نصف مدلهای قبلی است، بنابراین این دقت به دست آمده مناسب است.

امتیازی 91 درصد

در بخش hatebert دیدیم که به دقت 91 درصد رسیدیم.