به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

مینی‌پروژه سری ۲

نام و نام خانوادگی	مهدی عبداله چالکی - مهدی مهدیخانی
شماره دانشجویی	۸۱۰۶۰۰۲۹۰-۸۱۰۶۰۰۲۹۷
تاریخ ارسال گزارش	۱۲ خرداد ۱۴۰۱

فهرست گزارش سوالات

سوال ۱ – Stock Market Prediction

پیش‌پردازش‌ها

در مرحله اول لازم است که نسبت به داده‌ها یک دید کلی پیدا کنیم. به همین منظور ستون Close برای goog و aapl در نمودارهای زیر رسم شده است:

شکل ۱ روند فیمت **close** برای **goog**

شکل ۲ روند فیمت **close** برای **aapl**

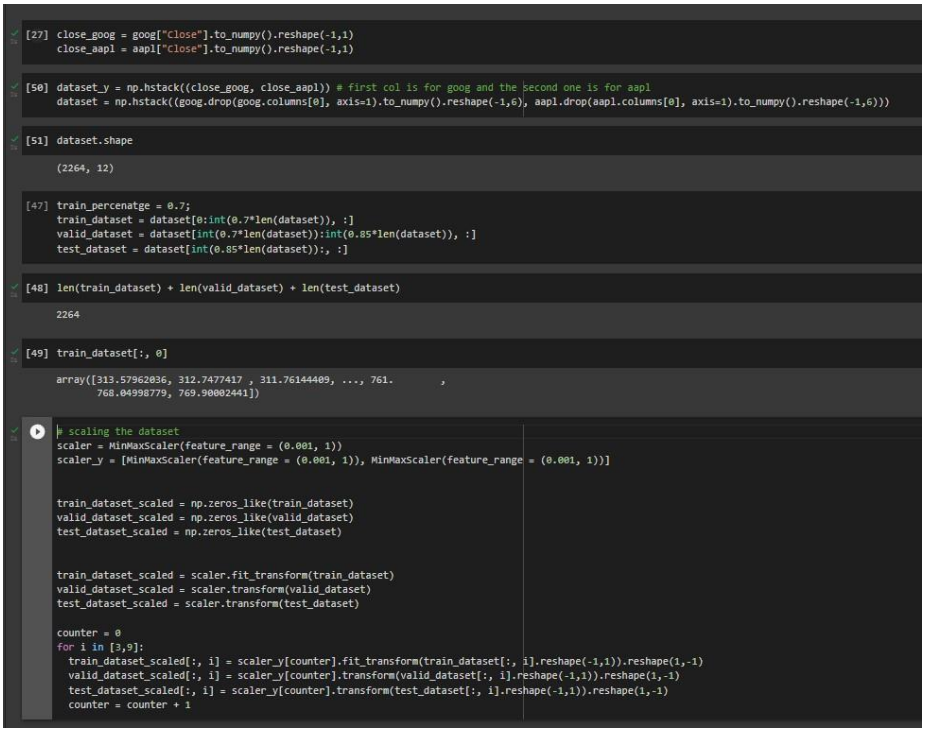
همانطور که دیده می‌شود یک روند کلی مشابه در هر دو نمودار دیده می‌شود. برای آموزش شبکه پیش‌بینی زمانی، ستون‌های date به شکل روز در آمدند (داده‌ها برای تمامی روزها ذکر شده است پس مشکلی از این بابت پیش نمی‌آید و همه روزها پشت سر هم هستند). همچنین ستون مورد نظر ، ستون Close است. از هر ۶ پارامتر دیگر نیز در این مدل استفاده شده است. درواقع ورودی شبکه یک ورودی ۱۲

ستونی است و خروجی شبکه یک خروجی ۲ ستونی است که پیش‌بینی قیمت برای دو کمپانی به صورت مجزا است.

از آنجا که خواسته شده است که شبکه به طور همزمان قیمت هر دو شرکت را پیش‌بینی کند، لازم است ورودی به فرم دو ستون باشد که به صورت موازی به شبکه داده می‌شود. همچنین از آنجایی که بازه زمانی ۳۰ روزه مد نظر است، پس هر داده ورودی دارای ۳۰ ردیف و ۱۲ ستون است که یک sequence را تشکیل می‌دهد. با کنار هم قرار دادن این بازه‌های زمانی به صورت batch، یادگیری صورت می‌گیرد.

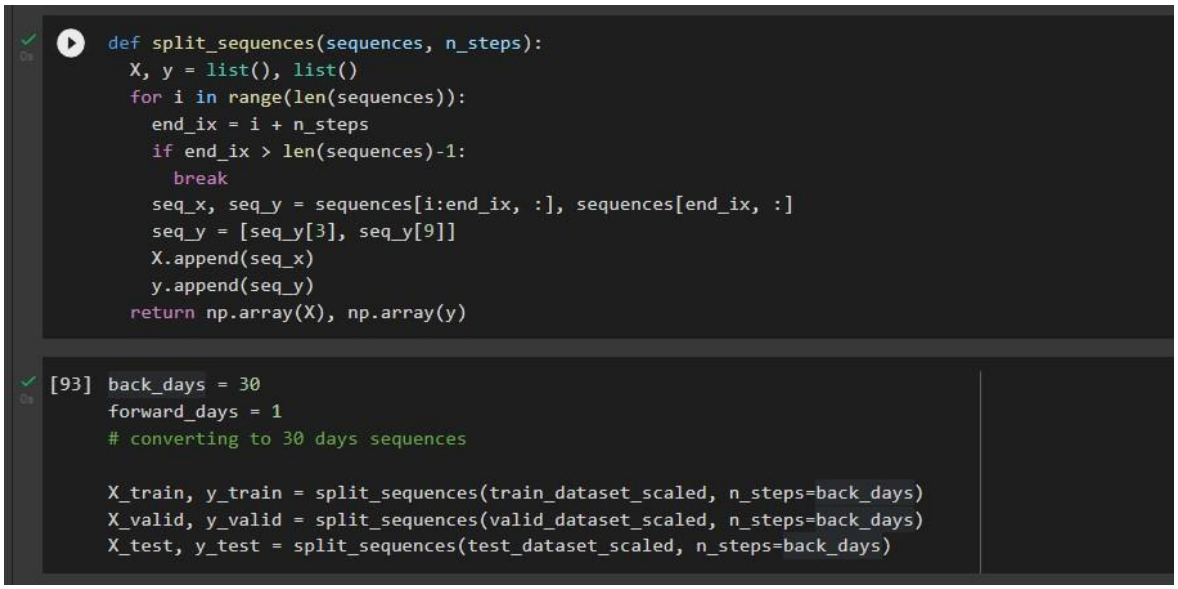
توصیه می‌شود که داده‌ها به بازه ۰ تا ۱ اسکیل شوند. اما ابتدا لازم است داده‌های یادگیری جدا شوند تا اسکیل تنها با استفاده از داده‌های یادگیری صورت بگیرد. در داده‌های سری زمانی نمی‌توان به صورت رندوم داده‌ها را جدا کرد و لازم است که از انتهای سری، داده‌های تست و ارزیابی جدا شوند تا عملکرد شبکه در پیش‌بینی زمانی به طرز صحیحی مشخص شود. به همین منظور ۷۰ درصد داده‌های ابتدایی را به داده‌های یادگیری، ۱۵ درصد بعدی به داده‌های ارزیابی و ۱۵ درصد انتهایی بازه زمانی (جدیدترین داده‌ها) به داده‌های تست تخصیص داده شد.

حال می‌توانیم اسکیل داده‌ها بر اساس داده‌های یادگیری را انجام دهیم. به همین منظور داده‌ها به بازه ۰.۰۰۱ تا ۱ مپ شدند. علت انتخاب ۰.۰۰۱ این است که قصد استفاده از MAPE به عنوان تابع خطا داریم. در این تابع خطا اگر عدد ۰ ظاهر شود، مقدارهای بزرگی ایجاد می‌شود که می‌تواند سیستم را منحرف کند و تاثیر زیادی بر روی تابع خطا بگذارد. تکه کد زیر برای جداسازی داده تست و یادگیری و اسکیل کردن داده‌ها مورد استفاده قرار گرفته است:



شکل ۳ جداسازی داده تست و یادگیری و اسکیل داده‌ها بر حسب داده یادگیری

همانطور که گفته شد لازم است داده‌ها به اندازه ۳۰ تایی جدا شوند و پنجره‌های ۳۰ تایی ایجاد شود که مقدار X را تشکیل می‌دهد و مقدار خروجی نیز برابر با یک تایم استپ بعدی است که قرار است تخمین زده شود و درواقع مقادیر y را تشکیل می‌دهد. برای داده‌های یادگیری، تست و ارزیابی این موضوع با استفاده از تکه کد زیر صورت گرفته است:



شکل ۴ تقسیم داده‌ها به پنجره‌های ۳۰ تایی

الف) طراحی سه شبکه

در این قسمت سه شبکه ساده RNN، LSTM و GRU ساخته و یادگیری شدند. این شبکه‌ها تنها از ۱ لایه ریکارنت مدنظر و یک لایه خروجی Dense تشکیل شده‌اند که خروجی دارای ابعاد ۲ است. می‌توان برای افزایش عمق به شبکه، چندین لایه ریکارنت را پشت یکدگیر قرار داد اما با آزمون و خطا دیده شد که یک لایه نیز کافی است و مشکل اورفیت یا یادگیری کم نیز رخ نمی‌دهد. تعداد یونیت‌ها هر لایه نیز برابر با ۱۰۰ انتخاب شد. برای انتخاب تعداد یونیت‌ها نیز آزمون و خطای زیادی انجام گرفت و مشاهده شد که همین ۱۰۰ یونیت تقریبا برای همه شبکه‌ها کافی است. تکه کد زیر برای تولید شبکه مدنظر مورد استفاده قرار گرفته است:

```python
def create_model(model_name, activation, optimizer, loss_function, dropout, recurrent_dropout, back_days, feature_size):
  model = Sequential()
  # input layers
  if model_name == "RNN":
    model.add(SimpleRNN(100, activation=activation, dropout=dropout, recurrent_dropout=recurrent_dropout, input_shape=(back_days, feature_size)))
  elif model_name == "LSTM":
    model.add(LSTM(100, activation=activation, dropout=dropout, recurrent_dropout=recurrent_dropout, input_shape=(back_days, feature_size)))
  elif model_name == "GRU":
    model.add(GRU(100, activation=activation, dropout=dropout, recurrent_dropout=recurrent_dropout, input_shape=(back_days, feature_size)))

  model.add(Dense(2))
  model.compile(optimizer=optimizer, loss=loss_function, metrics=['mse', "mape"])

  return model, model.summary()
```

شکل ۵ تابع ساخت مدل مدنظر

به عنوان مثال برای شبکه RNN، ساختاری به شکل زیر به دست آمد:

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn_2 (SimpleRNN)    (None, 100)               11300

 dense_2 (Dense)             (None, 2)                 202

=================================================================
Total params: 11,502
Trainable params: 11,502
Non-trainable params: 0
_________________________________________________________________
```

شکل ۶ ساختار شبکه **RNN**

مابقی شبکه‌ها نیز مشابه با همین ساختار هستند و تنها تفاوت در لایه اول است که به جای RNN، از LSTM و GRU استفاده شده است و از ذکر مجدد آن‌ها خودداری می‌کنیم.

در تمامی قسمت‌های این سوال از batch size برابر با ۱۴ انتخاب شد و تعداد EPOCH نیز برای همه موارد یادگیری برابر با ۳۰ قرار داده شد (به جز بخش اخر سوال که به دلیل وجود دراپ‌اوت، تعداد ایپاک بیشتری یادگیری صورت گرفت). همچنین در این قسمت تابع بهینه‌ساز ADAM انتخاب شد و تابع هزینه

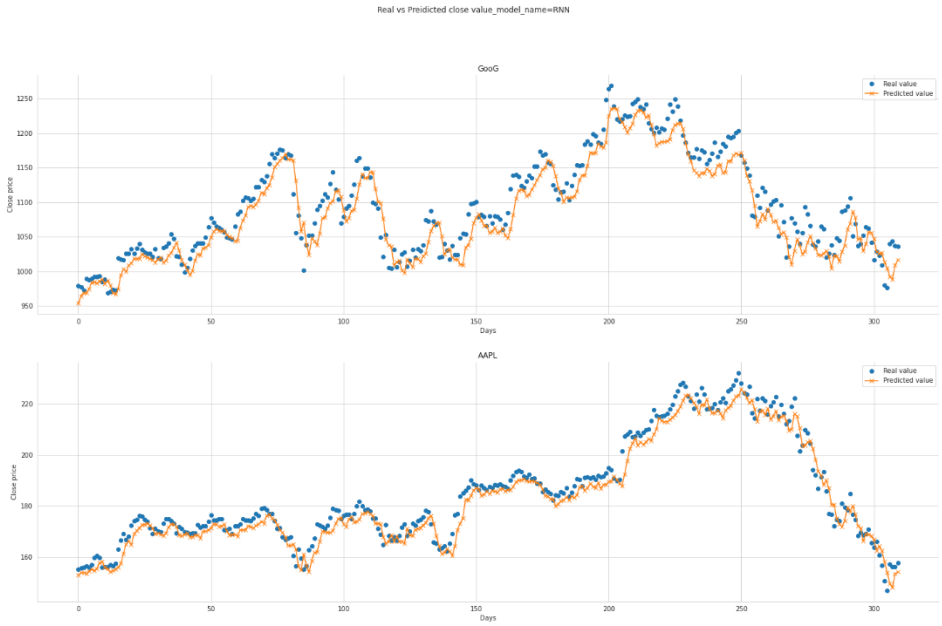
نیز MSE انتخاب شد. برای انتخاب تابع activation نیز توابع sigmoid، tanh وRELU مورد بررسی قرار گرفتند. دیده شد که RELU به صورت کلی بهترین عملکرد را با هردو تابع MSE و MAPE داشت. هرچند برای شبکه‌های LSTM و GRU توابع فعال‌ساز tanh و sigmoid بیشتر توصیه شدند و RELU به دلیل اینکه مقادیر منفی را صفر می‌کند و همچنین ممکن است که مقادیر بزرگی در خروجی داشته باشند، کمتر توصیه شده است. با این حال دیده شد که در این مسئله مشخص، RELU عملکرد بهتری را به همراه داشته است.

برای یادگیری با پارامترهای مختلف، یک تابع توسعه داده شد که در آن به صورت خودکار مدل شبکه با پارامترهای دلخواه تغییر پیدا می‌کند و در آخر اطلاعات و عکس‌ها ذخیره می‌شوند.

نمودار خطای شبکه RNN در حالت اول و با استفاده از تابع هزینه MSE در تصویر زیر قرار گرفته است.

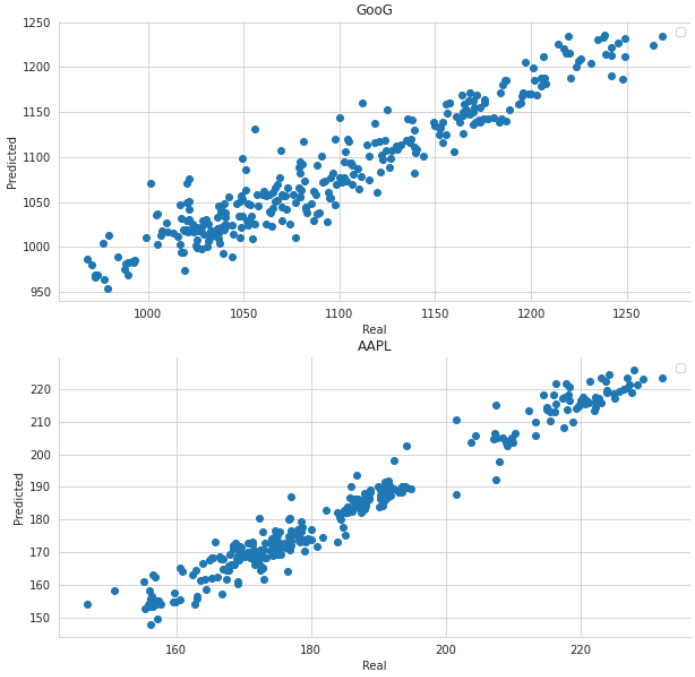
شکل ۷ خطای شبکه **RNN** برای داده‌های یادگیری و ارزیابی با تابع هزینه **MSE**

همچنین نمودار زمانی مقدار Close واقعی و مقدار به دست آمده برای داده‌های تست هر دو شرکت در تصویر زیر قرار گرفته است. تصویر بالا مربوط به GOOG و تصویر پایین مربوط به AAPL است (خط نارنجی مقدار پیش‌بینی شده و نقاط آبی مقادیر واقعی هستند).



شکل ۸ داده‌های واقعی و پیش‌بینی شده تست توسط شبکه **RNN** با تابع خطای **MSE** برای هردو شرکت

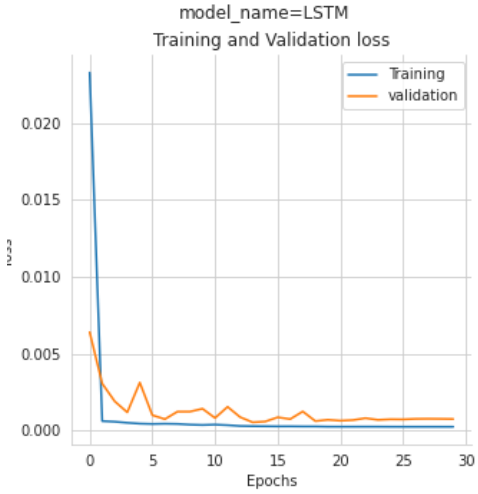
همچنین می‌توانیم نمودار مقدار واقعی در برابر مقدار تخمین‌زده شده را نیز رسم کنیم. اما به نظر می‌رسد نمودار بالا دید بهتری نسبت به دقت شبکه به ما می‌دهد بنابراین در قسمت‌های بعدی تنها از نمودارهای زمانی استفاده خواهیم کرد و نمودارهای مشابه با نمودار زیر در پیوست فایل‌ها قرار خواهد گرفت.



شکل ۹ مقدار پیش‌بینی شده در برابر مقدار واقعی برای دو شرکت با استفاده از شبکه **RNN**

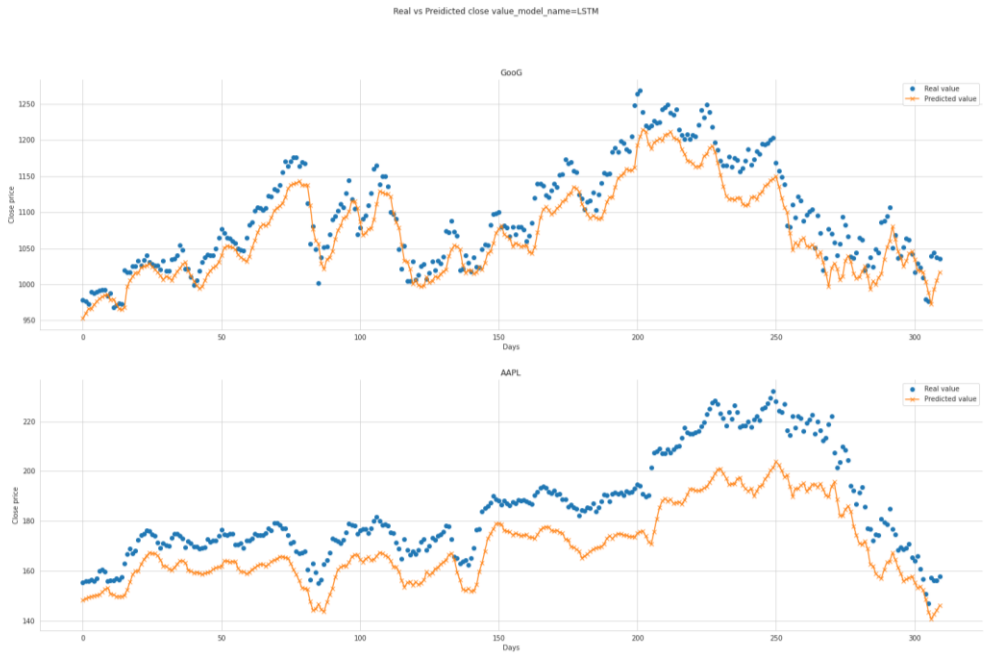
همانطور که دیده می‌شود نقاط نمودار نزدیک به خط y=x است که نشان می‌دهد خطای شبکه زیاد نیست.

با استفاده از لایه LSTM، نمودار زیر برای خطای شبکه به دست آمد. دقت شود که از یک callback به نام ReduceLROnPlateau در روند یادگیری استفاده شده است که اگر مقدار خطای ارزیابی افزایش شد، مقدار لرنینگ ریت کاهش پیدا کند تا احتمال اورفیت کاهش پیدا کند. همین موضوع سبب می‌شود یادگیری بهتری صورت بگیرد و از یه جایی به بعد شیب نمودار کمتر شود. نمودار خطای این شبکه در زیر قرار گرفته است.



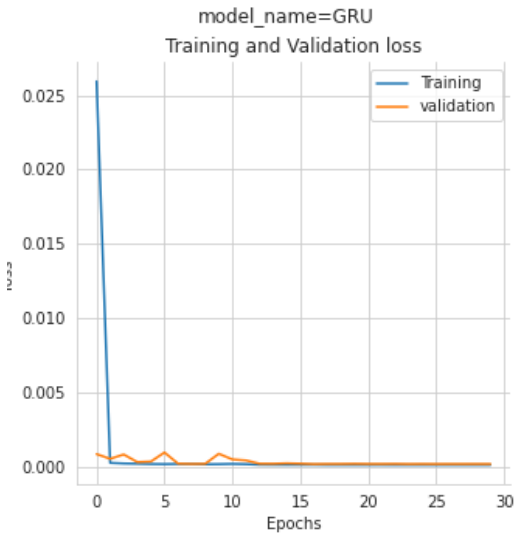
شکل ۱۰ خطای شبکه **LSTM** برای داده‌های یادگیری و ارزیابی با تابع هزینه **MSE**

نمودار زمانی مقدار واقعی و مقدار پیش‌بینی شده برای دو شرکت نیز در نمودار زیر قرار گرفته است.



شکل ۱۱ داده‌های واقعی و پیش‌بینی شده تست توسط شبکه **LSTM** با تابع خطای **MSE** برای هردو شرکت

در نهایت با استفاده از لایه GRU، نتایج زیر برای خطای شبکه به دست آمد.



شکل ۱۲ خطای شبکه **GRU** برای داده‌های یادگیری و ارزیابی با تابع هزینه **MSE**

و نمودار زمانی داده‌های پیش‌بینی و واقعی نیز در تصویر زیر قرار گرفته است.

شکل ۱۳ داده‌های واقعی و پیش‌بینی شده تست توسط شبکه **GRU** با تابع خطای **MSE** برای هردو شرکت

حال می‌توانیم شبکه‌های بالا را با یکدگیر مقایسه کنیم. به همین منظور خلاصه عملکرد این سه شبکه در جدول زیر گزارش شده است.

جدول**1** : مقایسه شبکه‌ها بر روی داده تست در استفاده از تابع هزینه **MSE**

Model name	epochs	Batch size	Test loss	Test mse	Test mape	training time (s)	activation	optimizer	Loss function
RNN	30	14	0.001955	0.001955	2.337703	51.04169	relu	adam	mse
LSTM	30	14	0.014455	0.014455	6.328667	82.72044	relu	adam	mse
GRU	30	14	0.001259	0.001259	1.769954	78.16926	relu	adam	mse

همانطور که دیده می‌شود کمترین خطا توسط شبکه GRU به دست آمده است. هرچند شبکه RNN نیز تقریبا نزدیک به همین شبکه است. از حیث زمان یادگیری نیز از آنجایی که بر روی GPU یادگیری صورت نگرفته است، تمام شبکه‌ها مقداری کند یادگیری شده‌اند. با این حال مقداری RNN سرعت بیشتری داشته است که می‌توان به ساختار ساده‌تر این شبکه و تنها وجود لاین‌های برگشتی در این شبکه اشاره کرد. شبکه GRU اندکی از شبکه LSTM سریع‌تر است. چرا که اندکی ساختار ساده‌تری دارد. در این شبکه به جای استفاده از یک یونیت مموری، به صورت مستقیم داده‌های کانال هیدن به بعدی منتقل می‌شود و

تنها دو گیت در یونیت‌های GRU وجود دارد. اما احتمال اینجا یکی از دلایل قابل استناد نبودن زمان‌ها، تغییر لرنینگ ریت در میانه یادگیری با استفاده از callback ذکر شده باشد که باعث می‌شود مسیر یادگیری برای هرشبکه متفاوت باشد و به عنوان مثال در RNN، چندین بار این مقدار لرنینگ ریت کاهش پیدا کرد تا اورفیت صورت نگیرد.

همچنین از حیث عملکرد ضعیف‌تر LSTM می‌توان گفت که این لایه توانایی بیشتری در یادآوری پترن‌های طولانی است و GRU و simpleRNN از این حیف ضعیف‌تر هستند. در این سوال بازه مد نظر چندان بزرگ نیست و تنها ۳۰ روز باید یادگیری شود و تنها یک روز آتی پیش‌بینی می‌شود. اما اگر میخواستیم که مثلا قیمت یک ماه آینده را پیش‌بینی کنیم، احتمالا LSTM می‌توانست عملکرد بهتری را از خود نشان دهد. در بین LSTM و RNN نیز شبکه GRU قرار می‌گیرد که مزیت هردوی این شبکه‌ها را به طور نسبی دارا هست. در این بخش نیز دیده شد که بهترین عملکرد را از خود نشان داد.

ب) تاثیر تابع هزینه

در بخش بالایی از تابع هزینه MSE استفاده شد. در این قسمت از تابع هزینه MAPE نیز استفاده خواهد شد و نتایج با یکدیگر مقایسه خواهد شد. در ابتدا اما لازم است توضیحاتی در رابطه با این دو تابع هزینه داده شود.

رابطه محاسبه MSE به شکل زیر است:

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

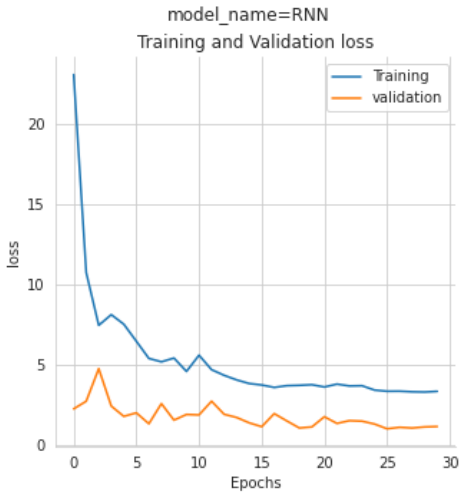
که به طور ساده می‌توان گفت که میانگین مربع خطا توسط این این معیار مشخص می‌شود. همچنین MAPE به شکل زیر محاسبه می‌شود:

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\frac{|A_i - F_i|}{A_i}$$

که با ضرب کردن این مقدرا در ۱۰۰ می‌توان آن را به فرم درصد نیز نمایش داد. در بحث پیش‌بینی زمانی، MAPE می‌تواند درک بهتری از خطا را به وجود آورد و برای کسانی که تخصص ندارند، قابل هضم‌تر است. درحالتی که MSE اینگونه نیست. همچنین MSE نسبت به اسکیل حساس است و اگر قرار باشد داده‌هایی با اسکیل‌های متفاوت استفاده شود، گزینه خوبی محسوب نمی‌شود. اما از آنجایی که داده‌ها در مخرج MAPE نیز قرار دارند، این تابع خطا نسبت به اسکیل‌های متفاوت کمتر حساس است.

اما مشکلی که MAPE دارد این است که نسبت به داده‌های منفی و داده‌هایی که نزدیک به ۰ هستند حساسیت بیشتری دارد و می‌تواند بسیار منحرف شود. همچنین این تابع قرینه نیست و بایاس دارد و به همین منظور برای جبران این مشکل از sMAPE استفاده می‌شود. درضمن MSE نسبت به داده‌های outlier حساسیت بیشتری از خود نشان می‌دهد.

نمودار خطای سیستم دراستفاده از MAPE به عنوان تابع هزینه در شبکه با لایه RNN در تصویر زیر قرار گرفته است.

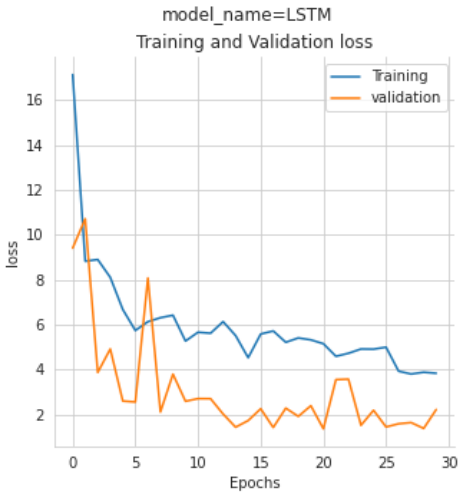


شکل ۱۴ خطای شبکه **RNN** برای داده‌های یادگیری و ارزیابی با تابع هزینه **MAPE**

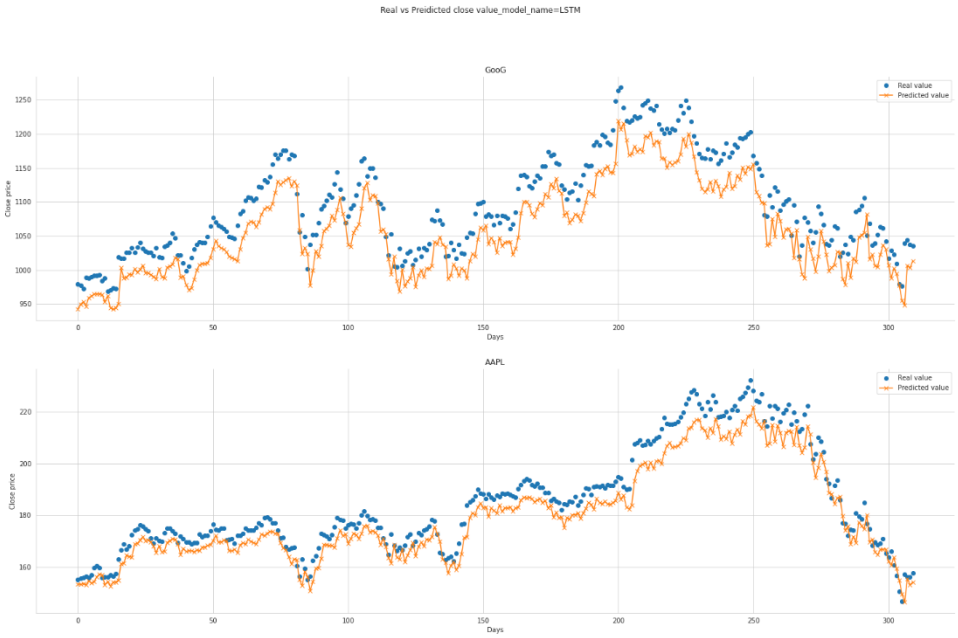
و نمودار زمانی داده‌های واقعی و پیش‌بینی شده نیز در تصویر زیر قرار گرفته است.

شکل ۱۵ داده‌های واقعی و پیش‌بینی شده تست توسط شبکه **RNN** با تابع خطای **MAPE** برای هردو شرکت

و نتایج استفاده از لایه LSTM در تصویر زیر قرار گرفته است.

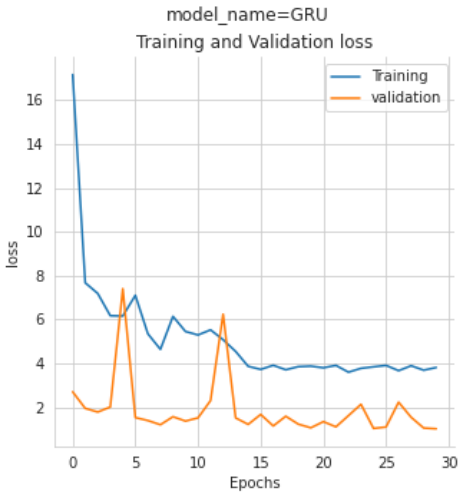


شکل ۱۶ خطای شبکه **LSTM** برای داده‌های یادگیری و ارزیابی با تابع هزینه **MAPE**



شکل ۱۷ داده‌های واقعی و پیش‌بینی شده تست توسط شبکه **LSTM** با تابع خطای **MAPE** برای هردو شرکت

و نتایج استفاده از لایه GRU در تصویر زیر دیده می‌شود.



شکل ۱۸ خطای شبکه **GRU** برای داده‌های یادگیری و ارزیابی با تابع هزینه **MAPE**

شکل ۱۹ داده‌های واقعی و پیش‌بینی شده تست توسط شبکه **GRU** با تابع خطای **MAPE** برای هردو شرکت

حال می‌توانیم با استفاده از مقادیر جدول زیر، سه شبکه را در دو حالت استفاده از دو تابع هزینه MSE و MAPE را با یکدگیر مقایسه کنیم.

جدول**2** : مقایسه شبکه‌ها بر روی داده تست در استفاده از توابع هزینه مختلف

Model name	epochs	Batch size	Test loss	Test mse	Test mape	training time (s)	activation	optimizer	Loss function
RNN	30	14	0.001955	0.001955	2.337703	51.04169	relu	adam	mse

LSTM	30	14	0.014455	0.014455	6.328667	82.72044	relu	adam	mse
GRU	30	14	0.001259	0.001259	1.769954	78.16926	relu	adam	mse
RNN	30	14	1.720694	0.001205	1.720694	50.43196	relu	adam	mape
LSTM	30	14	3.917286	0.004731	3.917286	73.0409	relu	adam	mape
GRU	30	14	1.592238	0.00106	1.592238	81.03649	relu	adam	mape

همانطور که دیده می‌شود برای هر سه شبکه، MAPE عملکرد بهتری را به ثبت رسانده است و مقدار MSE نیز از حالتی که از MSE به عنوان تابع هزینه استفاده می‌کردیم کمتر شده است. علت این موضوع این می‌تواند باشد که داده‌های ولیدیشن و تست اسکیل متفاوتی داشتند و بعد از اینکه با استفاده از مقادیر مربوط به داده‌های یادگیری اسکیل انجام شد، داده‌های تست و ولیدیشن تقریبا در بازه ۱ تا ۲ قرار گرفتند. دقت شود که بازه اسکیل داده‌های یادگیری برابر با ۰.۰۰۱ تا ۱ انتخاب شده بود. همچنین اختلاف ران تایم شبکه‌ها در استفاده از تابع هزینه MAPE بهتر دیده می‌شود و LSTM کندترین شبکه بوده است.

ج) تاثیر استفاده از بهینه‌سازهای مختلف

در اینجا از بهترین مدل بخش قبلی یعنی با تابع هزینه MAPE استفاده می‌کنیم. همچنین یکی از توابع بهینه (Adam) در بخش‌های قبلی استفاده شد و لازم است که دو بهینه‌ساز دیگر نیز بررسی شود. در اینجا برخلاف قسمت قبل، هر شبکه به صورت مجزا بررسی خواهد شد و برای حالت Adam نیز یکبار دیگر شبکه آموزش داده شده است. بنابراین ممکن است که نتایج کمی با جدول ۲ متفاوت به دست آید که به دلیل ذات رندوم بودن حل مسئله است. در ابتدا یک توضیح در مورد این توابع بهینه‌ساز می‌دهیم.

هر سه تابع بهینه‌ساز ADAgrad، Adam و RMSprop از جمله توابع بهینه‌ساز تطبیقی محسوب می‌شوند که در آن‌ها در طول زمان مقدار لرنینگ ریت به صورت تطبیقی تغییر پیدا می‌کند و همچنین برای پارامترهای مختلف نیز می‌توان از نرخ یادگیری متفاوتی استفاده کرد.

در روش ADAgrad، به جای استفاده از مجموع گرادیان‌ها، از مربع گرادیان استفاده می‌شود و با استفاده از همین پارامتر، نرخ یادگیری در مسیرهای مختلف انتخاب می‌شود و درواقع مقدار تغییر در نرخ یادگیری با استفاده از تاریخچه تغییرات مربع گرادیان صورت می‌گیرد. قانون بروزرسانی ADAgrad برای وزن‌ها و بایاس به شکل زیر است[۱].

[۱] https://bit.ly/3z7RgQE

$$v_t^w = v_{t-1}^w + (\nabla w_t)^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t^w + \epsilon}} * \nabla w_t$$
$$v_t^b = v_{t-1}^b + (\nabla b_t)^2$$
$$b_{t+1} = b_t - \frac{\eta}{\sqrt{v_t^b + \epsilon}} * \nabla b_t$$

درواقع ایده کلی در ADAgrad این اسکه هرچه یک پارامتر در گذشته بیشتر آپدیت شده باشد، در آینده کمتر آپدیت خواهد شد و به وزن‌های دیگر فرصت تغییر بیشتر داده می‌شود و معیار سنجش، مربع گرادیان است. همین موضوع سبب می‌شود که الگوریتم در نقاط Saddle point گیر نکند[1].

اما مشکلی که ADAgrad دارد، کند بودن آن در برخی مسائل است (این موضوع در برخی شبکه‌هایی که در ادامه به‌آن‌ها خواهیم پرداخت دیده شده است. اما مشکل این است که دسترسی به منابع Colab یکنواخت نیست و خیلی نمی‌توان به زمان‌های اجرای آن استناد کرد). علت آن هم این است که مربع خطا همیشه مقداری افزایش دارد و کاهشی نیست. همچنین در ADAgrad مقدار لرنینگ ریت با شیب زیادی گاه تغییر پیدا می‌کند. اما با افزایش مخرج، پارامترها تنها مقدار کمی آپدیت خواهند شد. به همین منظور در RMSprop یک نرخ decay اضافه می‌شود که رشد مربع گرادیان را کاهش دهد. این نرخ نشان می‌دهد که بیشتر تغییرات گرادیان اخیر تاثیر گذار هستند و تغییرات گرادیان قدیمی تاثیر کمتری بر بروزرسانی لرنینگ ریت دارند. همین کم‌تر نگه داشتن مجموع مربع گرادیان‌ها سبب می‌شود که تغییرات نرم‌تر صورت بگیرد و در انتها نیز همچنان تغییرات خوبی صورت بگیرد و درمجموع سرعت تابع بهینه‌ساز نیز افزایش پیدا می‌کند.

فرمول برزورسانی RMSprop به شرح زیر است:

$$v_t^w = \beta * v_{t-1}^w + (1-\beta)(\nabla w_t)^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t^w + \epsilon}} * \nabla w_t$$
$$v_t^b = \beta * v_{t-1}^b + (1-\beta)(\nabla b_t)^2$$
$$b_{t+1} = b_t - \frac{\eta}{\sqrt{v_t^b + \epsilon}} * \nabla b_t$$

در روش Adam سعی شده است که مزیت‌های RMSprop کمی بهبود پیدا کند. به همین منظور علاوه بر استفاده از مربع مجموع گرادیان‌ها، از مجموع گرادیان‌ها نیز استفاده می‌شود و همین موضوع سبب می‌شود بایاس کاهش پیدا کند. درواقع Adam شدت تغییرات را با استفاده از مجموع گرادیان‌ها مشخص می‌کند

[1] https://bit.ly/3NcUkPC

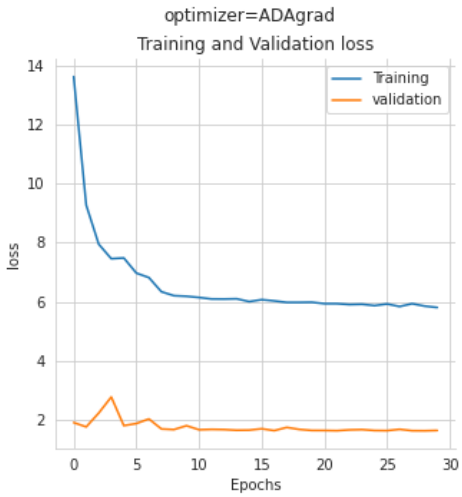
و توانایی در حرکت در جهت‌های مختلف را با استفاده از مجموع مربعات گرادیان‌ها به دست می‌آورد. فرمول بروزرسانی در این روش به شرح زیر است:

$$m_t = \beta_1 * m_{t-1} + \left(1 - \beta_1\right) * \nabla w_t$$
$$v_t = \beta_2 * v_{t-1} + \left(1 - \beta_2\right) * (\nabla w_t)^2$$
$$\widehat{m}_t = \frac{m_t}{1 - \beta_1^t} \widehat{v}_t = \frac{v_t}{1 - \beta_2^t}$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\widehat{v}_t + \epsilon}} * \widehat{m}_t$$

مزیت‌های Adam سبب شده است که در سال‌های اخیر محبوبیت زیادی داشته باشد.

از آنجایی که نمودارهای مربوط به استفاده از تابع هزینه MAPE و تابع بهینه‌ساز Adam در بخش ب قرار گرفته است، از ذکر مجدد آن‌ها اجتناب می‌کنیم. هرچند برای حالت Adam یکبار دیگر اجرا صورت گرفته است و نتایج کمی متفاوت هستند اما این تفاوت به شکلی نیست که چشمگیر باشد.

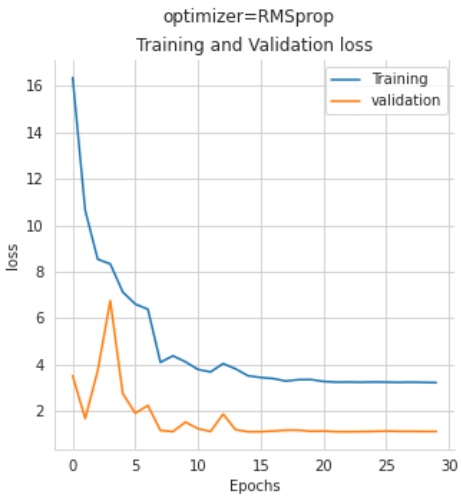
نتایج استفاده از ADAgrad با شبکه RNN و تابع هزینه MAPE در دو تصویر زیر قرار گرفته است.



شکل ۲۰ خطای شبکه **RNN** برای داده‌های یادگیری و ارزیابی با تابع بهینه‌ساز **ADAgrad**

شکل ۲۱ داده‌های واقعی و پیش‌بینی شده تست توسط شبکه **RNN** با تابع بهینه‌ساز **ADAgrad** برای هردو شرکت

نتایج استفاده از RMSprop با شبکه RNN نیز در زیر قرار گرفته است.



شکل ۲۲ خطای شبکه **RNN** برای داده‌های یادگیری و ارزیابی با تابع بهینه‌ساز **RMSprop**

شکل ۲۳ داده‌های واقعی و پیش‌بینی شده تست توسط شبکه **RNN** با تابع بهینه‌ساز **RMSprop** برای هردو شرکت

در جدول زیر می‌توان مقایسه این سه حالت را در شبکه RNN با یکدیگر مشاهده کرد. همانطور که گفته شد به دلیل متفاوت بودن نحوه اختصاص منابع در Colab، چندان نمی‌توان به تایم‌های اجرا استناد کرد. چرا که دیده می‌شود در ران دوم با استفاده از Adam، نتایج کمی متفاوت با جدول ۲ به دست آمدند. علت اجرای دوباره Adam نیز این است که زمان شروع یادگیری شبکه‌ها نزدیک به هم باشد و شاید اینگونه زمان یادگیری قابل استنادتر باشد.

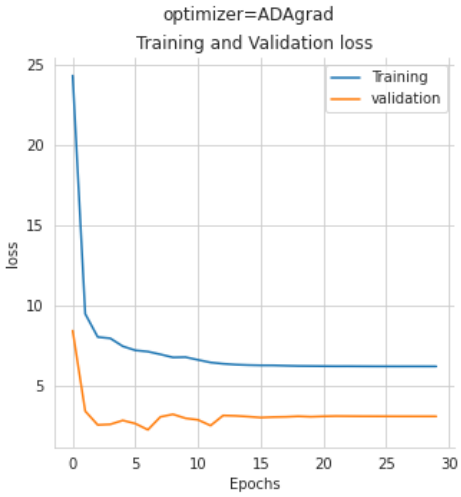
جدول**3** : نتایج یادگیری و اجرای شبکه **RNN** بر روی داده تست با توابع بهینه‌ساز مختلف

Model name	epochs	Batch size	Test loss	Test mse	Test mape	training time (s)	activation	optimizer	Loss function
RNN	30	14	1.656518	0.001196	1.656518	50.18811	relu	adam	mape
RNN	30	14	2.33085	0.002102	2.33085	82.53778	relu	ADAgrad	mape
RNN	30	14	1.641437	0.001154	1.641437	50.46576	relu	RMSprop	mape

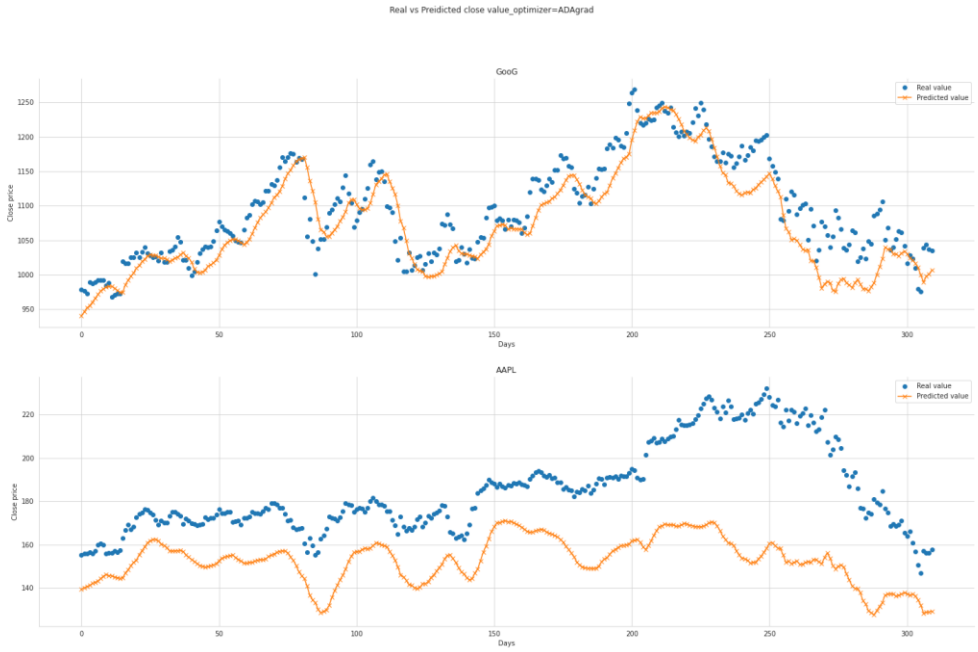
در انتها تفاوت‌ها بررسی خواهد شد.

به صورت مشابه این سه تابع بهینه‌ساز بر روی شبکه LSTM پیاده‌سازی شدند و نمودارهای مربوط به دو تابع جدید در ادامه ذکر خواهد شد.

در زیر نتایج استفاده از تابع بهینه‌ساز ADAgrad بر روی شبکه LSTM و با تابع هزینه MAPE دیده می‌شود.

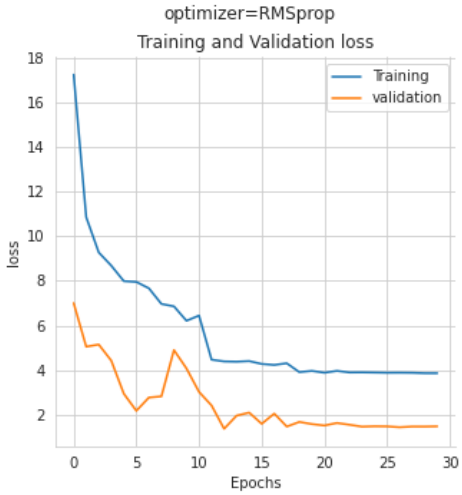


شکل ۲۴ خطای شبکه **LSTM** برای داده‌های یادگیری و ارزیابی با تابع بهینه‌ساز **ADAgrad**

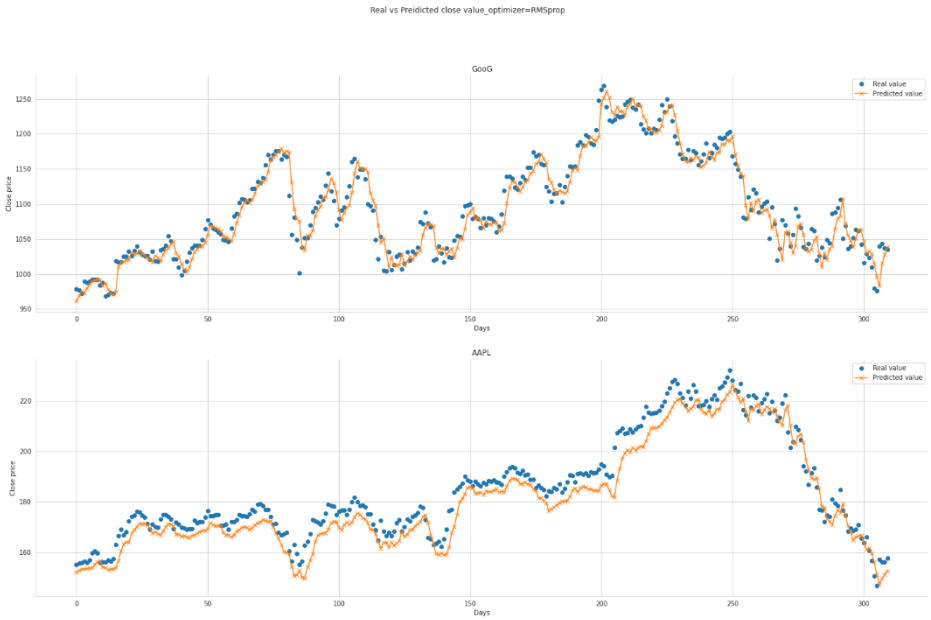


شکل ۲۵ داده‌های واقعی و پیش‌بینی شده تست توسط شبکه **LSTM** با تابع بهینه‌ساز **ADAgrad** برای هردو شرکت

و نتایج استفاده از RMSprop در دو تصویر زیر قرار گرفته است:



شکل ۲۶ خطای شبکه **LSTM** برای داده‌های یادگیری و ارزیابی با تابع بهینه‌ساز **RMSprop**



شکل ۲۷ داده‌های واقعی و پیش‌بینی شده تست توسط شبکه **LSTM** با تابع بهینه‌ساز **RMSprop** برای هردو شرکت

جدول زیر نیز نتایج پیش‌بینی شبکه بر روی داده تست و همچنین زمان یادگیری را برای این سه حالت به نمایش می‌گذارد.

جدول**4** : نتایج یادگیری و اجرای شبکه **LSTM** بر روی داده تست با توابع بهینه‌ساز مختلف

Model name	epochs	Batch size	Test loss	Test mse	Test mape	training time (s)	activation	optimizer	Loss function
LSTM	30	14	2.629166	0.00239	2.629166	70.15809	relu	adam	mape
LSTM	30	14	11.55437	0.06148	11.55437	82.69159	relu	ADAgrad	mape

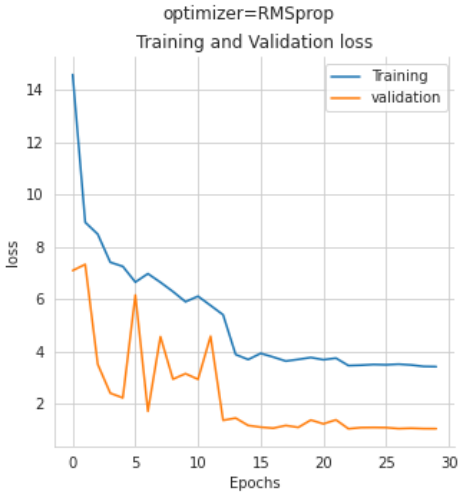
LSTM	30	14	2.496485	0.00226	2.496485	71.00884	relu	RMSprop	mape

و در نهایت لازم است که تاثیر استفاده از توابع بهینه‌ساز مختلف بر روی شبکه GRU بررسی شود. در دو تصویر زیر نتیجه استفاده از تابع بهینه‌ساز ADAgrad قرار گرفته است.

شکل ۲۸ خطای شبکه **GRU** برای داده‌های یادگیری و ارزیابی با تابع بهینه‌ساز **ADAgrad**

شکل ۲۹ داده‌های واقعی و پیش‌بینی شده تست توسط شبکه **GRU** با تابع بهینه‌ساز **ADAgrad** برای هردو شرکت

و نتایج استفاده از تابع بهینه‌ساز RMSprop برای شبکه GRU در دو تصویر زیر قرار گرفته است.



شکل ۳۰ خطای شبکه **GRU** برای داده‌های یادگیری و ارزیابی با تابع بهینه‌ساز **RMSprop**

شکل ۳۱ داده‌های واقعی و پیش‌بینی شده تست توسط شبکه **GRU** با تابع بهینه‌ساز **RMSprop** برای هردو شرکت

و جدول زیر نیز خلاصه عملکرد شبکه بر روی داده‌های تست و زمان یادگیری را با استفاده از سه تابع بهینه‌ساز مختلف نشان می‌دهد.

جدول**5** : نتایج یادگیری و اجرای شبکه **GRU** بر روی داده تست با توابع بهینه‌ساز مختلف

Model name	**epochs**	**Batch size**	**Test loss**	**Test mse**	**Test mape**	**training time (s)**	**activation**	**optimizer**	**Loss function**
GRU	30	14	2.302421	0.002161	2.302421	82.7164	relu	adam	mape

GRU	30	14	1.919756	0.001497	1.919756	82.49219	relu	ADAgrad	mape
GRU	30	14	1.632069	0.001156	1.632069	79.95543	relu	RMSprop	mape

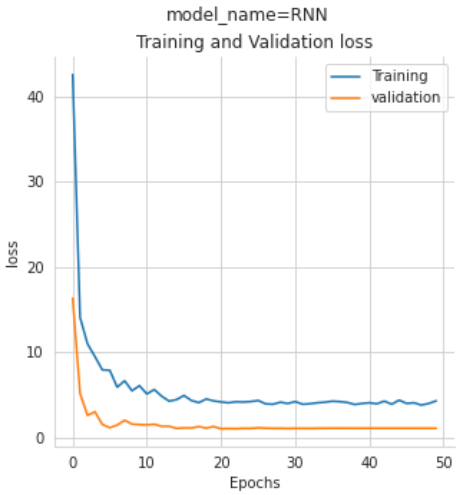
در ابتدای این قسمت توضحیاتی در رابطه با نحوه عملکرد این سه تابع داده شد و گفتیم که RMSprop و Adam شباهت زیادی به یکدگیر دارند و ADAgrad نیز نقاط ضعفی دارد که در این دو تابع بهینه‌ساز این نقاط ضعف تا حدودی برطرف شده‌اند. لذا انتظار داشتیم که بهترین عملکرد را از Adam و RMSprop شاهد باشیم که اجرای شبکه‌ها نیز این موضوع را تایید می‌کنند. در دو جدول از سه جدول ذکر شده در بالادیده شد که Adam بهترین عملکرد را از حیث کم بودن تابع هزینه بر روی داده تست را به همراه داشته است. البته در این ران GRU استثنائا ADAgrad نیز عملکرد خوبی داشت که می‌توانیم آن را به استیت رندوم نیز ربط دهیم. چرا که در برخی ران‌های دیگر اینگونه نبود. به خصوص در زمانی که تنها از فیچر Close استفاده کرده بودیم. اما در رانی که با ۱۲ فیچر گرفته شد، نتیجه به این شکل به دست آمد. به طور کلی می‌توانیم بگوییم که ADAgrad عملکرد خوبی را به همراه نداشت و نتوانست که به یادگیری خوبی دست پیدا کند. البته باید اشاره کنیم که ما با استفاده از ReduceLROnPlateau مقداری در عملکرد این توابع بهینه‌ساز دخالت می‌کنیم. در هر صورت عدم استفاده از این callback نتایج را تضعیف می‌کرد. همچنین در اجرای LSTM دیده شد که ADAgrad سرعت کمتری داشته است. اما همانطور که گفته شد، محدودیت منابع Colab و تغییرات آن باعث می‌شود که چندان نتوانیم به این داده‌های مربوط به زمان بتوانیم ارجاع کنیم.

د) تاثیر استفاده از Recurrent Dropout

در شبکه‌های بازگشتی می‌توانیم دو نوع Dropout اضافه کنیم که یکی از آن‌ها مربوط به dropout در ورودی و خروجی است و بین لایه‌های شبکه قرار می‌گیرد. اما می‌توانیم recurrent_dropout نیز استفاده کنیم که درواقع میان یونیت‌های لایه ریکارنت مورد استفاده قرار می‌گیرد. مقدار این پارامتر مشخص می‌کند که چه میزان از یونیت‌ها در هر فرآیند یادگیری غیرفعال می‌شوند. درواقع اتصال‌های x_t به h_t ها قطع می‌شود. نکته این است که در شبکه‌های پیش‌بینی زمانی، انجام این کار ممکن است سبب شود که بخشی از داده‌هایی که شبکه باید آن‌ها را یادآوری کند را از دست دهد و می‌تواند عملکرد شبکه را کاهش دهد. اما از طرفی ممکن است یادگیری غنی‌تری بر روی داده‌ها صورت بگیرد.

در ادامه مقدار recurrent_dropout برابر با ۰.۵ در نظر گرفته شد و هر سه شبکه با استفاده از این مقدار آموزش داده شدند.

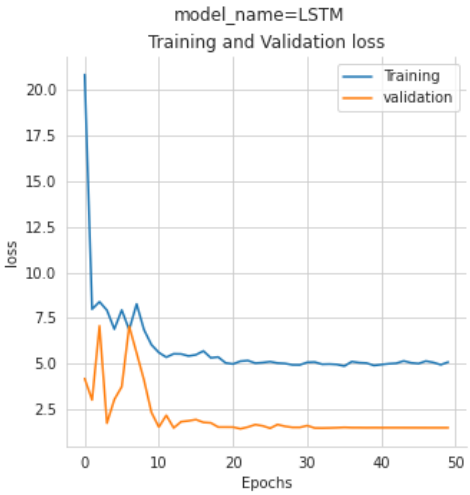
نتایج استفاده از Dropout بر روی سلول‌های بازگشتی شبکه RNN در دو تصویر زیر قرار گرفته است. تابع بهینه‌ساز برابر با Adam و تابع هزینه برابر با MAPE انتخاب شده‌اند. همچنین از آنجایی که تصور می‌شد به زمان یادگیری بیشتری لازم است، مقدار epoch برابر با ۵۰ انتخاب شد.



شکل ۳۲ خطای شبکه **RNN** برای داده‌های یادگیری و ارزیابی با **recurrent_dropout=0.5**

شکل ۳۳ داده‌های واقعی و پیش‌بینی شده تست توسط شبکه **RNN** با **recurrent_dropout=0.5** برای هردو شرکت

در زیر نیز می‌توان نتیجه استفاده از Dropout بر روی یونیت‌های لایه LSTM را مشاهده کرد.



شکل ۳۴ خطای شبکه **LSTM** برای داده‌های یادگیری و ارزیابی با **recurrent_dropout=0.5**

شکل ۳۵ داده‌های واقعی و پیش‌بینی شده تست توسط شبکه **LSTM** با **recurrent_dropout=0.5** برای هردو شرکت

و همین موضوع بر روی شبکه GRU نتایج زیر را رقم زده است.

شکل ۳۶ خطای شبکه **GRU** برای داده‌های یادگیری و ارزیابی با **recurrent_dropout=0.5**

شکل ۳۷ داده‌های واقعی و پیش‌بینی شده تست توسط شبکه **GRU** با **recurrent_dropout=0.5** برای هردو شرکت

برای اینکه بتوانیم دید بهتری از تاثیر dropout داشته باشیم، مقدار آن را به ۰.۲ کاهش دادیم. اما برای کوتاه نگه داشتن گزارش، از ذکر نمودارهای آن خودداری می‌کنیم و تنها مقادیر به دست آمده را در جدول زیر قرار می‌دهیم.

در جدول زیر می‌توان مقایسه‌ای میان حالات مختلف برای recurrent_dropout را مشاهده کرد.

جدول**6** : مقایسه شبکه‌ها بر روی داده تست در استفاده از مقادیر **recurrent_dropout** برابر با ۰، ۰.۲ و ۰.۵

Model name	epochs	Batch size	Test loss	Test mse	Test mape	activation	optimizer	Loss function	Recurrent dropout
RNN	30	14	1.553304	0.001062	1.553304	relu	adam	mape	0
LSTM	30	14	2.237981	0.00183	2.237981	relu	adam	mape	0
GRU	30	14	1.524223	0.001013	1.524223	relu	adam	mape	0
RNN	50	14	1.554018	0.001058	1.554018	relu	adam	mape	0.2
LSTM	50	14	2.731689	0.002494	2.731689	relu	adam	mape	0.2
GRU	50	14	1.579701	0.001078	1.579701	relu	adam	mape	0.2
RNN	50	14	1.690074	0.001222	1.690074	relu	adam	mape	0.5
LSTM	50	14	2.170603	0.001897	2.170603	relu	adam	mape	0.5
GRU	50	14	1.815238	0.001309	1.815238	relu	adam	mape	0.5

همانطور که دیده می‌شود عملکرد شبکه LSTM با افزایش recurrent_dropout به مقداری افزایش پیدا کرده است. علت این موضوع می‌تواند به یادگیری بهتر در چنین حالتی باشد. اما دو شبکه GRU و RNN با افزایش مقدار dropout، مقدار کمی عملکردشان افت پیدا کرده است. البته باید دقت داشت که در حالت اول یادگیری در ۳۰ ایپاک صورت گرفته است و در حالت بعدی، در ۵۰ ایپاک و این موضوع می‌تواند مقداری در نتیجه نهایی تاثیر گذار باشد (هرچند به نظر می‌رسد که در ۳۰ ایپاک تقریبا یادگیری به اتمام رسیده است). اما مشخص است که برای شبکه پیچیده‌تری مانند LSTM، افزودن Dropout می‌تواند سبب شود که مقداری یادگیری بهتر صورت بگیرد. اما در شبکه‌های ساده‌تر ممکن است اضافه کردن Dropout سبب شود که مقداری از داده‌های مهمی که نیاز به یادگیری داشتند فراموش شوند و این موضوع می‌تواند کمی بر عملکرد شبکه تاثیر گذار باشد. البته در نتایج بالا به صورت کلی تفاوت‌ها به شکلی نیست که بتوان گفت تفاوت‌ها چشمگیر است. همچنین تاثیر Dropout بین لایه‌های ورودی و خروجی نیز بررسی شد و دیده شد که اضافه کردن این لایه کمکی به یادگیری نمی‌کند و مقداری از عملکرد شبکه نیز می‌کاهد.

پس به صورت کلی می‌توانیم بگوییم در شبکه‌های بزرگ و پیچیده مقداری در نظر گرفتن recurrent_dropout می‌تواند به یادگیری بهتر شبکه منجر شود. اما در شبکه‌های ساده‌تر می‌تواند عملکرد شبکه را تضعیف کند.

سوال ۲ – Text Generation

در این سوال، قصد داریم به کمک شبکه‌های عصبی بازگشتی، حرف‌ها متنی از کتاب هری‌پاتر و جام آتش را تولید کنیم. به همراه سوال، بخشی از متن این کتاب در قالب فایل txt قرار داده شده که در ابتدای کد نوشته شده، بارگذاری می‌کنیم.

پیش‌پردازش داده‌ها

پس از خواندن فایل متنی، باید پیش‌پردازش‌هایی بر روی داده‌ها انجام گیرد. برای مثال، می‌توانیم حروف بزرگ را هم به حروف کوچک تبدیل کنیم تا تفاوتی از نظر پایتون نداشته باشند. با این کار، تعداد کاراکترها و در نتیجه تعداد لیبل‌ها کاسته شده و علاوه بر کاهش پارامترهای شبکه و حجم محاسبات، دقت خروجی نیز افزایش خواهد یافت. با همین استدلال، کاراکتر ’/n‘ را نیز حذف می‌کنیم؛ چرا که رفتن به خط بعد از ویژگی‌های متن محسوب نمی‌شود. همچنین ارقام را هم باید حذف کنیم تا تعداد کاراکترها کم شود. تعداد ارقام در کتابی مانند هری‌پاتر، درصد بسیار ناچیزی از متن اصلی است و می‌توان آن را نادیده گرفت. پس از انجام این حذفیات، تعداد کاراکترها در طول متن به 1,101,140 کاراکتر می‌رسد. (با تست کردن داده‌ها، تعداد ارقام در این یک میلیون کارامتر، ۲۵ عدد بود که نشان می‌دهد پیش‌تر فرض درستی کرده بودیم).

همچنین تعداد کاراکترهای یکتا در متن پس از انجام پیش‌پردازش، به ۴۵ عدد می‌رسد.

در گام بعدی، باید مشخص کنیم طول (تعداد کاراکتر) داده‌های ورودی به شبکه باید چقدر باشد. این عدد به دلخواه ۱۵۰ انتخاب شده‌است. هر چه طول بیشتری در نظر گرفته شود، خروجی شبکه به کلمات معنی‌دار نزدیک‌تر خواهد بود. همچنین طول گام (بدان معنا که پنجره‌ی ۱۵۰تایی باید به چه میزان حرکت کند تا داده‌ی بعدی مشخص شود) ۱۰ کاراکتر در نظر گرفته می‌شود. در نتیجه، هر نمونه به صورت یک ورودی شامل ۱۵۰ کاراکتر و یک خروجی شامل یک کاراکتر در می‌آید.

در نهایت، 110,099 ورودی داریم که با نسبت ۷۰ درصد داده‌های آموزش، ۱۵ درصد داده‌های ولیدیشن و ۱۵ درصد نیز داده‌های تست تقسیم می‌کنیم. لازم به ذکر است که ورودی و خروجی‌ها باید کدگذاری شوند تا بتوان از آن‌ها استفاده کرد. می‌توان برای هر حرف، یک رقم تعریف کرد و از همان استفاده کرد. اما کار بهتر آن است که برای هر کاراکتر یک بردار به طول تعداد کاراکترهای یکتا (در این سوال ۴۵ تا) در نظر بگیریم و به حالتی شبیه به One Hot عمل کنیم؛ یعنی شماره‌ی کاراکتر True قرار داده شده و بقیه درایه‌ها False در نظر گرفته می‌شوند. در نتیجه‌ی این عمل، برای مثال ابعاد داده‌های تست به صورت (93584, 150, 45) و ابعاد خروجی‌ها نیز به شکل (93584, 45) در می‌آید.

ایجاد شبکه‌ی عصبی و آموزش آن

پس از آماده‌سازی داده‌ها، نوبت به تعریف شبکه‌ی عصبی می‌رسد. در این تمرین به دلیل قابلیت‌های شبکه‌ی LSTM در به خاطر سپاری داده‌های پیشین و نیز عملکرد مناسب آن در زمینه‌ی پردازش زبان طبیعی، از این شبکه استفاده شده است. دو لایه‌ی LSTM ۲۵۶ تایی در نظر گرفته شده‌است که پس از هر کدام، یک لایه‌ی Dropout با آرگومان ورودی ۰.۲ قرار دارد و باعث می‌شود از اورفیت شدن شبکه بر روی داده‌هاس تست، تا حد امکان جلوگیری شود. معماری این مدل به شکل زیر است:

```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 lstm (LSTM)                  (None, 150, 256)          309248

 dropout (Dropout)            (None, 150, 256)          0

 lstm_1 (LSTM)                (None, 256)               525312

 dropout_1 (Dropout)          (None, 256)               0

 dense (Dense)                (None, 45)                11565

=================================================================
Total params: 846,125
Trainable params: 846,125
Non-trainable params: 0
```

شکل ۳۸: ساختار شبکه **LSTM** مورد استفاده برای پیشبینی کاراکترها

همچنین از آنجا که مساله از نوع طبقه‌بندی است، تابع فعال‌ساز لایه‌ی آخر را softmax قرار می‌دهیم و تعداد خروجی‌ها نیز قاعدتا برابر تعداد کاراکترهای یکتا یعنی ۴۵ خواهد بود. تابع بهینه‌ساز نیز Adam در نظر گرفته شده‌است تا همگرایی سریع‌تر و بهتری داشته باشیم. در این بخش، تابع خطا از نوع categorical_crossentropy قرار داده شده‌است.

از آنجا که احتمال دارد شبکه از تعدادی epoch به بعد دچار اورفیت شود، تعداد epochهای پیشفرض را ۱۵ عدد در نظر گرفته و در هر مرحله مدل را ذخیره می‌کنیم. بدین صورت می‌توانیم پس از رسم نمودارها و مسخص کردن بهترین تعداد epoch، مدل مربوطه را مجددا لود کرده و با استفاده از آن پیشبینی را انجام دهیم.

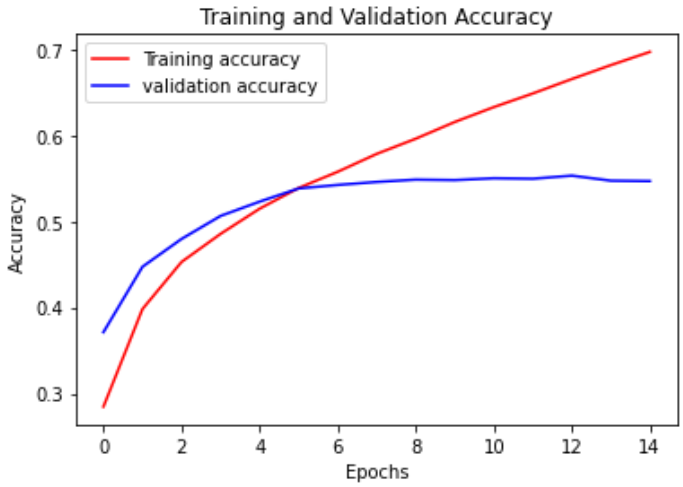
نتایج

پس از طی کردن مراحل پیشین، نوبت به تحلیل نتایج می‌رسد. در گام نهایی، نتایج بدین شرح است:

جدول ۷: نتایج دقت و خطای داده‌های آموزش و ارزیابی – مدل با تابع خطای **categorical_crossentropy**

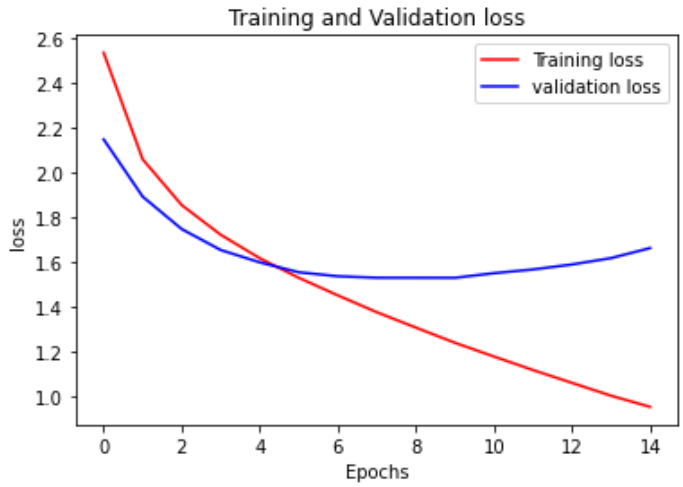
Training Accuracy	Training Loss	Validation Accuracy	Validation Loss
69.7 %	0.9511	54.68 %	1.66

نمودار دقت داده‌های آموزش و ارزیابی به شکل زیر است:



شکل ۳۹: نمودار دقت داده‌های آموزش و ارزیابی - مدل با تابع خطای **categorical_crossentropy**

نمودار خطای داده‌های آموزش و ارزیابی نیز به شکل زیر است:



شکل ۴۰: نمودار خطای داده‌های آموزش و ارزیابی - مدل با تابع خطای **categorical_crossentropy**

همان‌طور که از دو نمودار مشاهده می‌شود تقریبا از epoch هفتم بعد، دقت داده‌های ارزیابی بالاتر نمی‌رود اما مقدار تابع خطا بیشتر هم می‌شود. پس از مدلی که در این مرحله ذخیره کرده بودیم، استفاده

می‌کنیم تا کلمات را تولید کنیم. با استفاده از این مدل، دقت و مقددار تابع خطای داده‌های تست بدین صورت بدست می‌آید:

جدول ۸: نتایج دقت و خطای داده‌های تست – مدل با تابع خطای **categorical_crossentropy**

Test Loss	Test Accuracy
1.5420	53.88 %

که تقریبا برابر با دقت و خطای داده‌های ارزیابی است. با استفاده از این مدل، به تولید کاراکتر می‌پردازیم. بدین منظور، ابتدا یک ورودی ۱۵۰ کاراکتری به مدل داده و یک حرف را تولید می‌کنیم. حالا که ۱۵۱ کاراکتر داریم، از کاراکتر دوم تا ۱۵۱ را در نظر گرفته و به شبکه می‌دهیم و کاراکتر ۱۵۲ را تولید می‌کنیم. به همین ترتیب پیش رفته تا ۲۰۰ کاراکتر تولید شوند.

ورودی ۱۵۰ کاراکتری اولیه، بدین صورت است:

"a fine summer's morning when the riddle house had still been well kept and impressive, a maid had entered the drawing room to find all three riddles d"

عبارت تولید شده نیز به صورت زیر است:

darkly sturetting forby by smiting of you.... to grinnously."he's grettered, listing shritting the gryof started that still crumping, could me...."but ze tood hermione, she had go's reiding yol w

همان‌طور که مشاهده می‌شود، اکثر کلمات دارای معنا هستند، اما چون تمرکز در این تمرین بر روی تولید کاراکتر بوده و ما کلمات را tokenize نکردیم، نمی‌توان انتظار داشت که جملات تولید شده هم دارای معنی باشند.

ب) استفاده از دو تابع خطای دیگر

در این بخش، به بررسی فرایند آموزش بادو تابع خطای دیگر نیز می‌پردازیم و نتایج را با یکدیگر مقایسه می‌کنیم. از آنجا که مساله از نوع طبقه‌بندی است، باید به سراغ توابع خطای مناسب این نوع مسائل برویم.

ب – ۱) معیار واگرایی کولبک-لیبلر:

اولین تابع خطایی که بررسی می‌کنیم، تابع خطای Kullback-Leibler divergence است. در آمار ریاضی از Kullback–Leibler واگرایی به عنوان معیاری برای اندازه‌گیری واگرایی یک توزیع احتمال از یک توزیع احتمال ثانویه، یاد می‌شود. واگرایی Kullback-Leibler توزیع Q نسبت به P اغلب به صورت

DKL(P‖Q) نوشته می‌شود. برای توزیعهای احتمالاتی گسسته P و Q معیار واگرایی Kullback–Leibler واگرایی از Q به P، به صورت زیر تعریف می‌شود:

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_{i} P(i) \log \frac{P(i)}{Q(i)}$$

و در پایتون نیز بدین صورت پیاده‌سازی می‌شود:

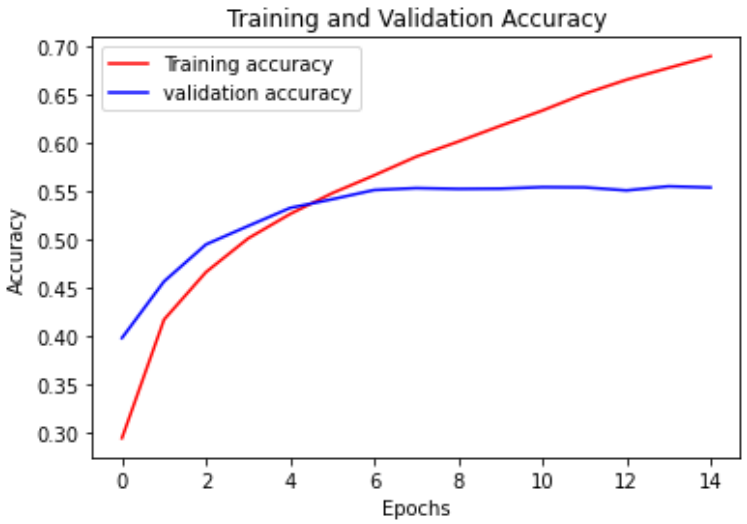
$$loss = y_{true} * log\,(\frac{y_{true}}{y_{pred}})$$

با استفاده از این تابع خطا، بار دیگر مدل را آموزش داده و نتایج را ارائه می‌دهیم. در گام نهایی، نتایج بدین شرح است:

جدول ۹: نتایج دقت و خطای داده‌های آموزش و ارزیابی – مدل با تابع خطای **Kullback-Leibler divergence**

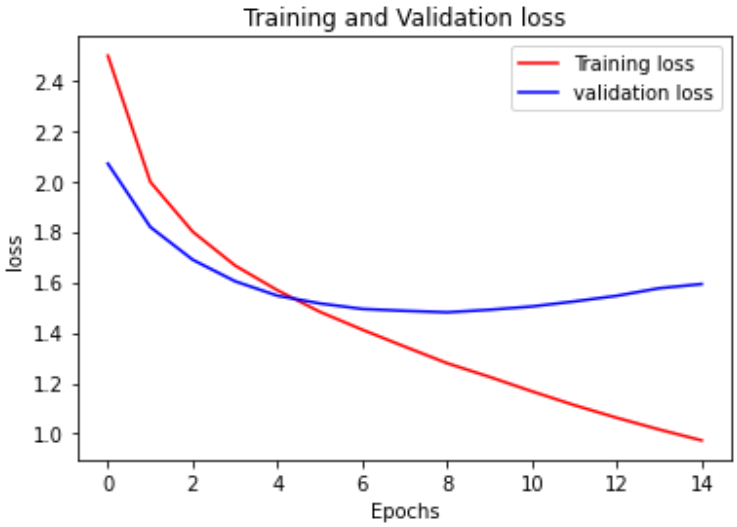
Training Accuracy	Training Loss	Validation Accuracy	Validation Loss
69.04 %	0.9734	55.43 %	1.5927

نمودار دقت داده‌های آموزش و ارزیابی به شکل زیر است:



شکل ۴۱: نمودار دقت داده‌های آموزش و ارزیابی - مدل با تابع خطای **Kullback-Leibler divergence**

نمودار خطای داده‌های آموزش و ارزیابی نیز به شکل زیر است:



شکل ۴۲: نمودار خطای داده‌های آموزش و ارزیابی - مدل با تابع خطای **Kullback-Leibler divergence**

در این بخش هم همانند قسمت قبل، تقریبا از epoch هفتم بعد دقت داده‌های ارزیابی بالاتر نمی‌رود اما مقدار تابع خطا بیشتر هم می‌شود. پس از مدلی که در این مرحله ذخیره کرده بودیم، استفاده می‌کنیم تا کلمات را تولید کنیم. با استفاده از این مدل، دقت و مقدار تابع خطای داده‌های تست بدین صورت بدست می‌آید:

جدول ۱۰: نتایج دقت و خطای داده‌های تست – مدل با تابع خطای **Kullback-Leibler divergence**

Test Loss	**Test Accuracy**
1.4971	55.22 %

مکانیزم تولید کاراکترها همانند بخش قبل است. ورودی ۱۵۰ کاراکتری اولیه، بدین صورت است:

"a fine summer's morning when the riddle house had still been well kept and impressive, a maid had entered the drawing room to find all three riddles d"

عبارت تولید شده نیز به صورت زیر است:

dinger, but harry who was at the words it of the dark walks, there whispered affer hir head."i odd we could heard nothing, but harry was that i know to tere what he was laughed into lest ofwer he was

در این حالت نسبت به حالت قبلی، تعداد کلمات بیشتری دارای معنا هستند. اما همچنان جملات دارای معنای خاصی نیستند.

ب – ۲) معیار BinaryCrossentropy

در این بخش، از معیار BinaryCrossentropy به عنوان تابع خطا استفاده می‌کنیم. آنتروپی متقاطع باینری، هر یک از احتمالات پیش‌بینی شده را با خروجی کلاس واقعی که می تواند ۰ یا ۱ باشد مقایسه می‌کند. سپس امتیازی را محاسبه می کند که احتمالات را بر اساس فاصله از مقدار مورد انتظار جریمه می‌کند. این بدان معناست که چقدر از مقدار واقعی نزدیک یا دور است. برای حالت چند کلاسه، فرمول محاسباتی آن بدین صورت است:

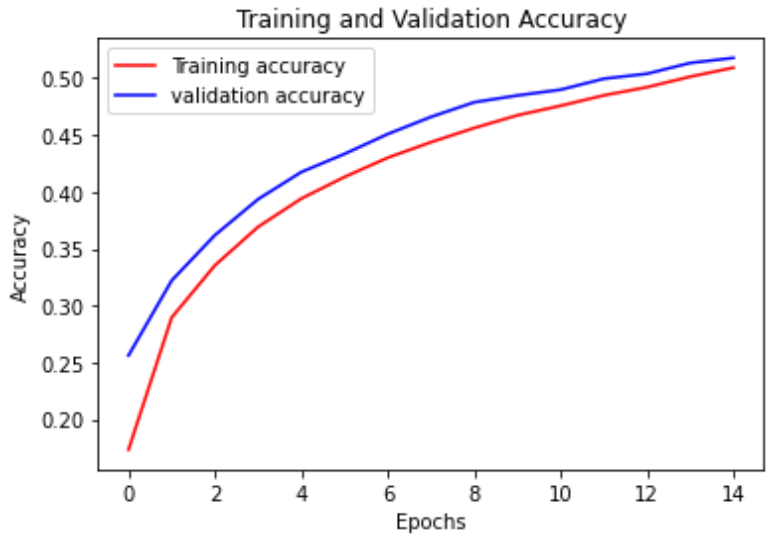
$$\text{logloss} = -\frac{1}{N}\sum_{i}^{N}\sum_{j}^{M} y_{ij}\log\left(p_{ij}\right)$$

که در آن N تعداد نمونه‌ها و M تعداد کلاس‌ها است. با استفاده از این تابع خطا، بار دیگر مدل را آموزش داده و نتایج را ارائه می‌دهیم. در گام نهایی، نتایج بدین شرح است:

جدول ۱۱: نتایج دقت و خطای داده‌های آموزش و ارزیابی – مدل با تابع خطای **BinaryCrossentropy**

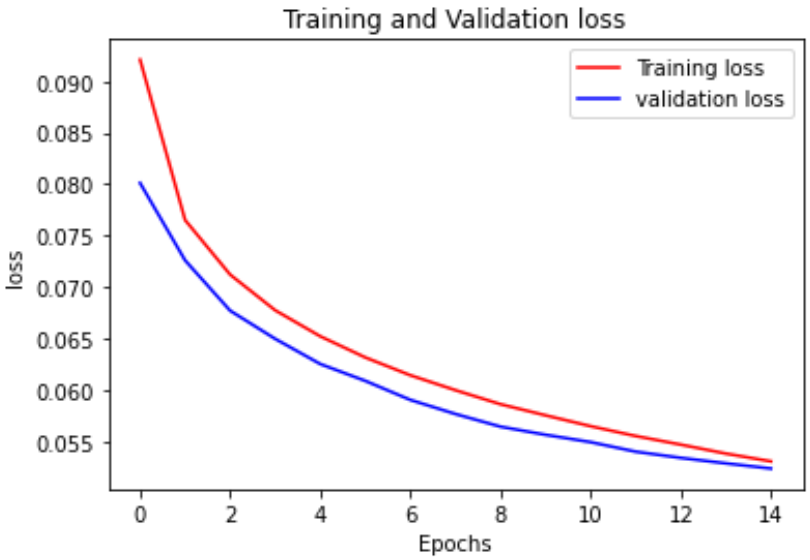
Training Accuracy	Training Loss	Validation Accuracy	Validation Loss
50.92 %	0.0531	51.78 %	0.0524

نمودار دقت داده‌های آموزش و ارزیابی به شکل زیر است:



شکل ۴۳: نمودار دقت داده‌های آموزش و ارزیابی - مدل با تابع خطای **BinaryCrossentropy**

نمودار خطای داده‌های آموزش و ارزیابی نیز به شکل زیر است:



شکل ۴۴: نمودار خطای داده‌های آموزش و ارزیابی - مدل با تابع خطای **BinaryCrossentropy**

در این بخش بر خلاف دو بخش قبلی، هیچ‌یک از دو نمودار همگرا نشده و همچنان در حال تغییر هستند. پس از آخرین مدلی که در این مرحله ذخیره کرده بودیم، استفاده می‌کنیم تا کلمات را تولید کنیم. با استفاده از این مدل، دقت و مقدار تابع خطای داده‌های تست بدین صورت بدست می‌آید:

جدول ۱۲: نتایج دقت و خطای داده‌های تست – مدل با تابع خطای **BinaryCrossentropy**

Test Loss	Test Accuracy
0.0524	51.29 %

مکانیزم تولید کاراکترها همانند بخش قبل است. ورودی ۱۵۰ کاراکتری اولیه، بدین صورت است:

"a fine summer's morning when the riddle house had still been well kept and impressive, a maid had entered the drawing room to find all three riddles d"

عبارت تولید شده نیز به صورت زیر است:

down the found of the gracked excecture of his till, kister, from who thought all a tinny eyes down of rembys was compening the pangedare was bull seving. he was to the warked the mippur, started ron

در این حالت نسبت به حالت قبلی، تعداد کلمات کمتری دارای معنا هستند و همچنان جملات دارای معنای خاصی نیستند.

مقایسه نتایج:

با توجه به این نکته که محاسبه خطا در هر روش به شیوه خاص خود انجام می‌گیرد، تنها با استفاده از معیار دقت می‌توان عملکرد مدل آموزش دیده با سه تابع خطای مختلف را بررسی کرد. در جدول زیر، این مقایسه برای ۱۵ epoch صورت گرفته است:

جدول ۱۳: مقایسه‌ی دقت داده‌های تست برای سه تابع خطا

تابع خطا	دقت داده‌های تست (درصد)
Categorical Crossentropy	53.88
Kullback-Leibler divergence	55.22
Binary Crossentropy	51.29

با مقایسه مقادیر این جدول، می‌توان دید که دقت هر سه مدل در ۱۵ ایپاک تقریبا مشابه هم بوده است و معیار Kullback-Leibler divergence اندکی بهتر از دو حالت دیگر عمل کرده است. با مقایسه کاراکترهای تولید شده توسط این سه مدل هم می‌توان دید که تعداد کلمات درست در حالت دوم، اندکی بهتر از سایرین بوده است. البته این نکته را هم باید در نظر گرفت که در حالت Binary Crossentropy به حالت اشباع نرسیده بودیم و احتمالا با آموزش در ایپاک‌های بیشتر، می‌توانیم به دقت‌های بالاتری دست یابیم.

ج) بررسی عملکرد مدل با استفاده از دو معیار دیگر

در این بخش، ابتدا عملکرد مدل را با یک نرخ یادگیری (learning rate) متفاوت بررسی می‌کنیم، سپس تعداد ایپاک‌ها را مورد بررسی قرار می‌دهیم.

ج – ۱) اثر نرخ یادگیری: در این قسمت از مدل Binary Crossentropy استفاده می‌کنیم و به جای نرخ یادگیری پیش‌فرض بهینه‌ساز Adam که ۰.۰۰۱ تعریف شده است، از نرخ یادگیری ۰.۰۰۰۱ استفاده می‌کنیم. پس از طی کردن ۱۵ epoch، نتایج بدین صورت است:

در گام نهایی، نتایج بدین شرح است:

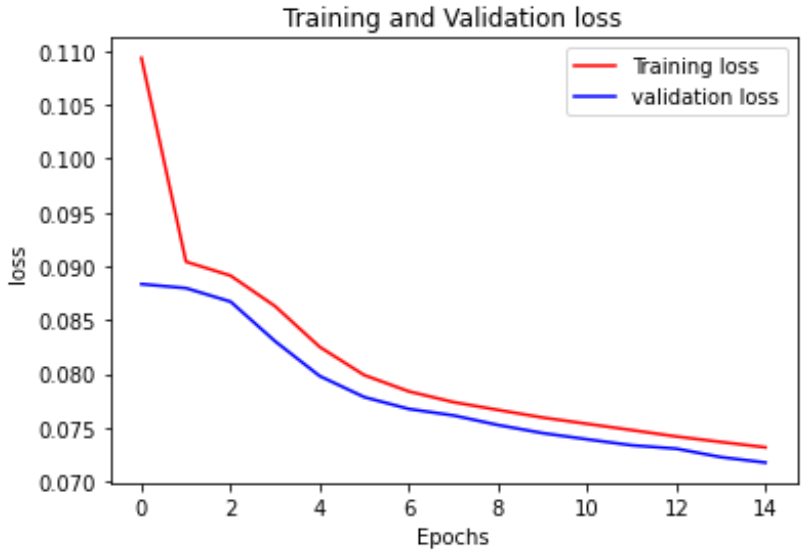
جدول ۱۴: نتایج دقت و خطای داده‌های آموزش و ارزیابی – مدل با تابع خطای **BinaryCrossentropy** و نرخ یادگیری ۰.۰۰۰۱

Training Accuracy	Training Loss	Validation Accuracy	Validation Loss
31.46 %	0.0732	32.94 %	0.0718

نمودار دقت داده‌های آموزش و ارزیابی به شکل زیر است:

شکل ۴۵: نمودار دقت داده‌های آموزش و ارزیابی - مدل با تابع خطای **BinaryCrossentropy** و نرخ یادگیری ۰.۰۰۰۱

نمودار خطای داده‌های آموزش و ارزیابی نیز به شکل زیر است:



شکل ۴۶: نمودار خطای داده‌های آموزش و ارزیابی - مدل با تابع خطای **BinaryCrossentropy** و نرخ یادگیری ۰.۰۰۰۱

مشاهده می‌کنیم هر دو نمودار در حال بهبود هستند. بنابراین از آخرین مدلی که در این مرحله ذخیره کرده بودیم، استفاده می‌کنیم تا کلمات را تولید کنیم. با استفاده از این مدل، دقت و مقدار تابع خطای داده‌های تست بدین صورت بدست می‌آید:

جدول ۱۵: نتایج دقت و خطای داده‌های تست – مدل با تابع خطای **BinaryCrossentropy** و نرخ یادگیری ۰.۰۰۰۱

Test Loss	**Test Accuracy**

0.0708	33.56 %

مکانیزم تولید کاراکترها همانند بخش قبل است. ورودی ۱۵۰ کاراکتری اولیه، بدین صورت است:

"a fine summer's morning when the riddle house had still been well kept and impressive, a maid had entered the drawing room to find all three riddles d"

عبارت تولید شده نیز به صورت زیر است:

ow cnawf lamlem sud theis to the weus eod the. lonsenase ga toudep ank racedy oveid bisine nare the ponmsedeye n of darel, to kon torear and anlederden mereo,, horr is the gad; -out o" kank yect and

می‌توان مشاهده کرد که اکثر کلمات معنای خاصی ندارند.

استفاده از نرخ یادگیری پایین‌تر، نیازمند تعداد ایپاک‌های بیشتری است و در نتیجه در تعداد ایپاک مشابه، نتوانسته‌است به عملکرد مطلوبی دست یابد. مقایسه این حالت با حالت اصلی بدین صورت است:

جدول ۱۶: مقایسه‌ی دقت داده‌های تست دو شبکه با تابع خطای **BinaryCrossentropy** و نرخ یادگیری ۰.۰۰۰۱ و ۰.۰۰۱

نرخ یادگیری	دقت (درصد)
0.001	51.29
0.0001	33.56

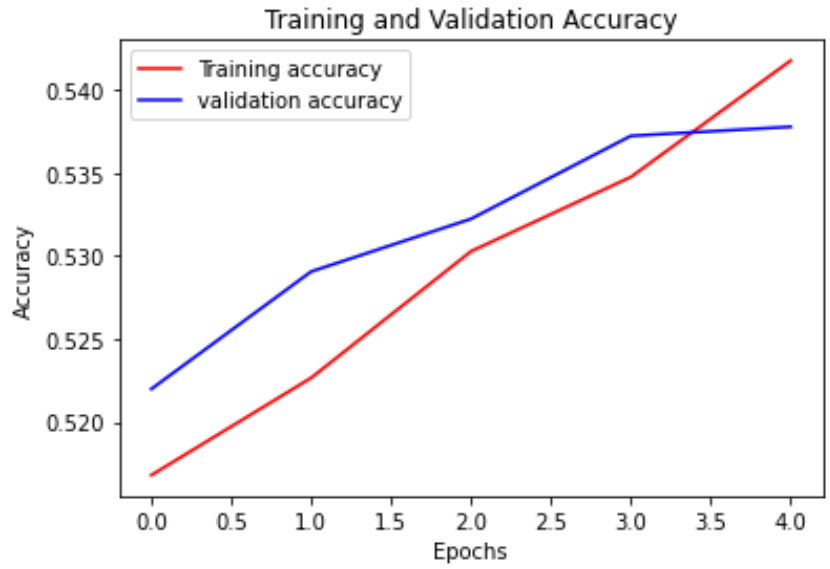
ج – ۲) تغییر تعداد epoch ها:

در این بخش، مدل قسمت ب-۲ را که در حال بهتر شدن بود، به تعداد ۵ ایپاک بیشتر آموزش می‌دهیم تا اثر تغییر تعداد ایپاک‌ها را بر روی آن بررسی کنیم. در گام نهایی، نتایج بدین شرح است:

جدول ۱۷: نتایج دقت و خطای داده‌های آموزش و ارزیابی – مدل با تابع خطای **BinaryCrossentropy** و تعداد ایپاک ۲۰

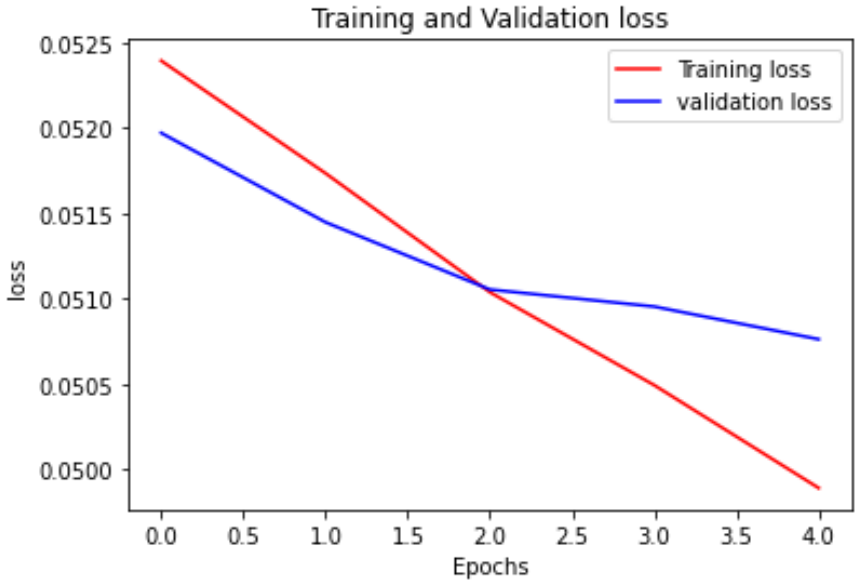
Training Accuracy	Training Loss	Validation Accuracy	Validation Loss
54.18 %	0.0499	53.78 %	0.0508

نمودار دقت داده‌های آموزش و ارزیابی برای ۵ ایپاک اضافه (یعنی ایپاک ۱۶ تا ۲۰) به شکل زیر است:



شکل ۴۷: نمودار دقت داده‌های آموزش و ارزیابی – ایپاک ۱۵ تا ۲۰ مدل با تابع خطای **BinaryCrossentropy**

نمودار خطای داده‌های آموزش و ارزیابی برای ۵ ایپاک اضافه (یعنی ایپاک ۱۶ تا ۲۰) نیز به شکل زیر است:



شکل ۴۸: نمودار خطای داده‌های آموزش و ارزیابی – ایپاک ۱۵ تا ۲۰ مدل با تابع خطای **BinaryCrossentropy**

مشاهده می‌کنیم هر دو نمودار همچنان در حال بهبود هستند. بنابراین از آخرین مدلی که در این مرحله ذخیره کرده بودیم، استفاده می‌کنیم تا کلمات را تولید کنیم. با استفاده از این مدل، دقت و مقدار تابع خطای داده‌های تست بدین صورت بدست می‌آید:

جدول ۱۸: نتایج دقت و خطای داده‌های تست – مدل با تابع خطای **BinaryCrossentropy** و تعداد ایپاک ۲۰

Test Loss	Test Accuracy
0.0511	52.89 %

مکانیزم تولید کاراکترها همانند بخش قبل است. ورودی ۱۵۰ کاراکتری اولیه، بدین صورت است:

"a fine summer's morning when the riddle house had still been well kept and impressive, a maid had entered the drawing room to find all three riddles d"

عبارت تولید شده نیز به صورت زیر است:

down to the back to be for might on the each of the roon melven, and and her hay hourd the rubure to down and the smeders. "the lad out her quidely harry of now was a tore and looked out the somether

می‌توان مشاهده کرد که کلمات بیشتری نسبت به حالت ۱۵ ایپاک، دارای معنی هستند.

مقایسه‌ی دو حالت این بخش بدین شکل است:

جدول ۱۹: مقایسه‌ی نتایج دو حالت با تعداد ایپاک ۱۵ و ۲۰

تعداد epoch	دقت (درصد)
15	51.29
20	52.89

که در این حالت، تعداد ایپاک بهتر نتایج بهتری را داده‌است. البته در بخش ب مشاهده کردیم که با توابع خطای دیگر، در ایپاک هفتم و هشتم به بهترین نتایج می‌رسیدیم و آموزش بیشتر منجر به اورفیت شدن می‌شد.

د) چگونگی اثر حافظه سلول های عصبی استفاده شده در مدل

در این سوال، هدف پیش‌بینی یک کاراکتر بر اساس کاراکترهای قبلی در متن یک کتاب بود. این مساله را می‌توان همانند یک مساله سری زمانی در نظر گرفت که با استفاده از داده‌های پیشین، قرار است داده‌ی آینده را پیشبینی کنیم. بنابراین مشابه مسائل سری زمانی، در اینجا نیز از شبکه‌ی عصبی از نوع LSTM بهره بردیم. شبکه‌های LSTM به طور خاص بدین منظور طراحی شده‌اند که بتوانند از پس وابستگی‌های طولانی‌مدت برآیند. این مشکلات در شبکه‌های عصبی عادی و بازگشتی دیگر به دلیل vanishing gradient بوجود می‌آید. شبکه‌های LSTM بر خلاف شبکه‌های فیدفوروارد، از فیدبک نیز استفاده می‌کنند که به

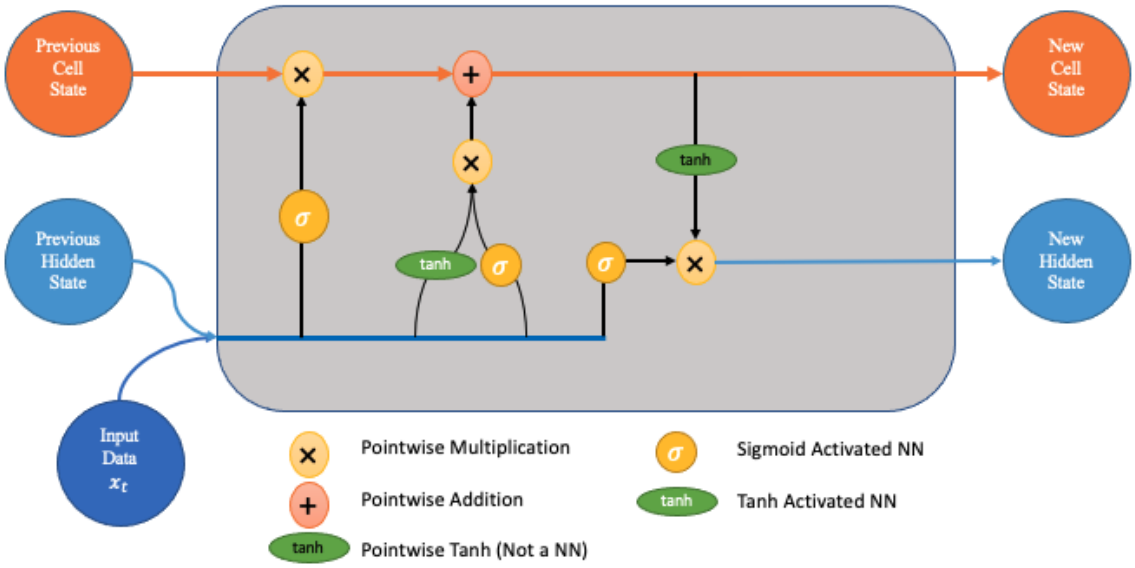
آن‌ها امکان می‌دهد بتوانند با محفوظ نگاه داشتن اطلاعات مفید زمانی‌های قبلی، تمام توالی داده‌ها را یه صورت یکجا بررسی کنند (و نه آنکه داده در هر زمان را به صورت مستقل بررسی کنند). بنابراین استفاده از این نوع شبکه‌ها برای پردازش داده‌هایی چون متن و گفتار و ...، مناسب است.

خروجی یک *LSTM* در یک نقطه خاص از زمان به سه چیز بستگی دارد:

- حافظه بلندمدت فعلی شبکه - که به عنوان وضعیت سلولی (cell state) شناخته می شود.
- خروجی در نقطه قبلی در زمان - به عنوان حالت پنهان قبلی (previous hidden state) شناخته می‌شود.
- داده‌های ورودی در مرحله زمانی فعلی

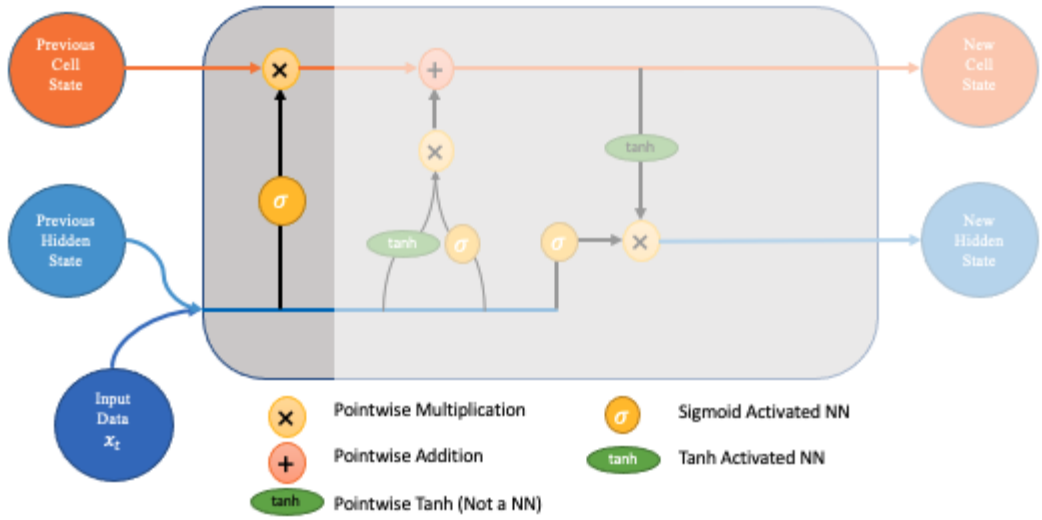
LSTM ها از یک سری gate استفاده می کنند که نحوه ورود، ذخیره و خروج اطلاعات یک توالی داده از شبکه را کنترل می‌کند. سه گیت در یک LSTM معمولی وجود دارد. گیت فراموشی، گیت ورودی و گیت خروجی. این گیت ها را می توان فیلتر در نظر گرفت و هر کدام خود یک شبکه عصبی هستند.

ساختار کلی یک سلول *LSTM* به شکل زیر است:



شکل ۴۹: نمودار شبکه **LSTM**

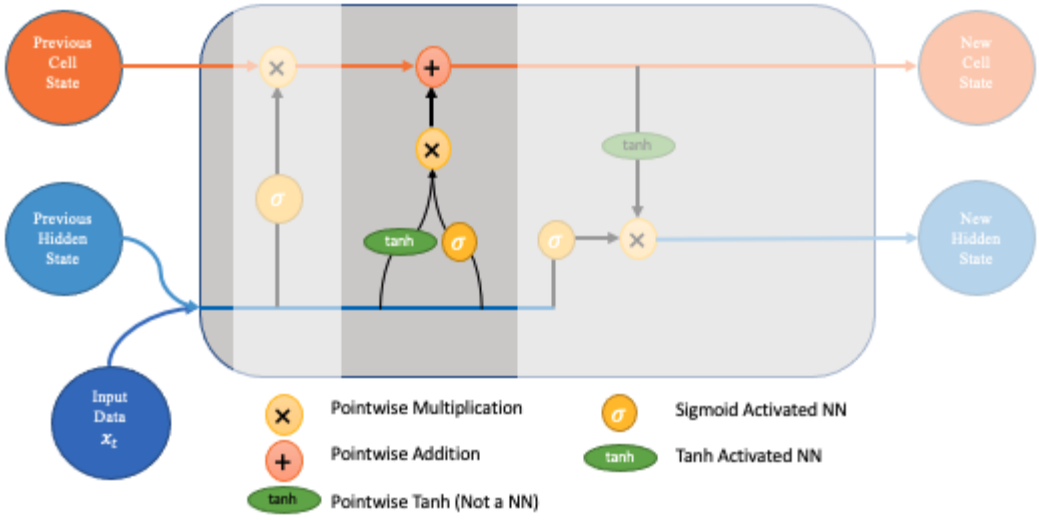
مرحله اول: اولین گام در این فرایند، گیت فراموشی است که بر اساس داده‌های ورودی جدید و *hidden state* قبلی، تصمیم می‌گیرد کدام یک از بیت‌های *cell state* مفید هستند.



شکل ۵۰: گیت فراموشی

با آموزش شبکه‌ی عشبی مربوط به این قسمت، ورودی‌هایی که غیرمفید تشخیص داده‌شوند در ۰ ضرب شده و در گام‌های بعدی، تاثیر زیادی ندارند. هر قدر اطلاعات مفیدتر باشند، در عددی نزدیکتر به ۱ ضرب می‌شوند تا در بخش‌های بعدی ایفای نقش کنند.

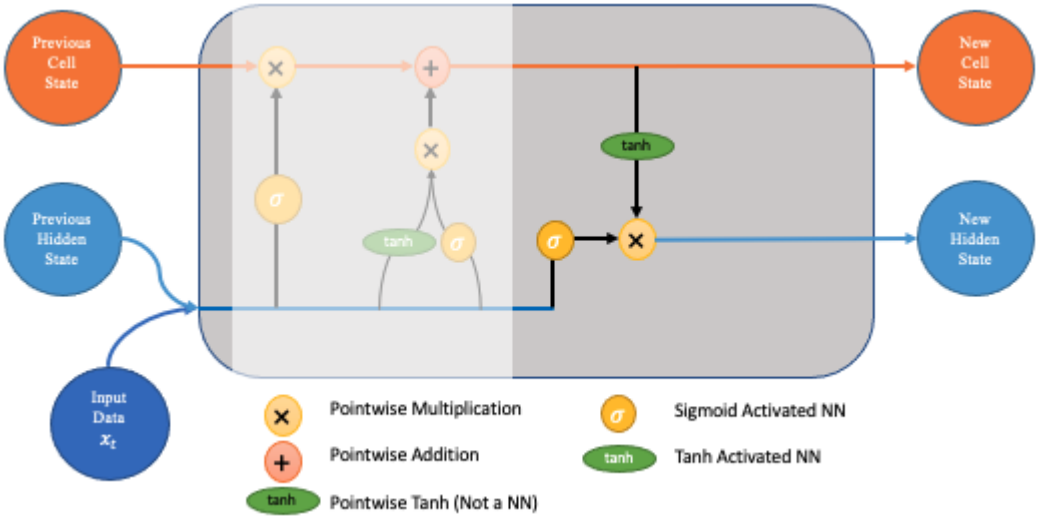
مرحله دوم: این مرحله شامل گیت ورودی و شبکه حافظه جدید است و بر اساس داده‌های جدید و hidden state قبلی، تصمیم می‌گیرد که چه اطلاعات جدیدی باید به cell state اضافه شوند.



شکل ۵۱: گیت ورودی

مرحله سوم: در این مرحله، بر اساس cell state آپدیت شده، hidden state مرحله‌ی قبلی و ورودی‌های جدید، hidden state مرحله‌ی بعدی تعریف می‌شود. این استیت به ما کمک می‌کند تا تنها

اطلاعات ضروری به عنوان خروجی داده شوند (و نه cell state، که هر چه شبکه از گذشته یاد گرفته‌است را شامل می‌شود).



شکل ۵۲: گیت خروجی

در نهایت، مراحل بالا به تعداد دفعات لازم (در این تمرین از آنجا که طول پنجره ورودی ۱۵۰ کاراکتر در نظر گرفته شده بود، ۱۵۰ دفعه) تکرار می‌شود و می‌تواند از این حافظه‌ی بلند مدت بهره گیرد تا خروجی بهتری برای خروجی خود داشته باشد.

سوال ۳ – Contextual Embedding + RNNs

در این سوال، قصد داریم تا با کمک شبکه‌های عصبی بازگشتی و استفاده از مدل‌های embedding نظیر BERT، مفهوم جملات داده‌شده را از نظر احساسات درونی آن (نرمال، تنفرانگیز یا توهین‌آمیز) پیدا کنیم.

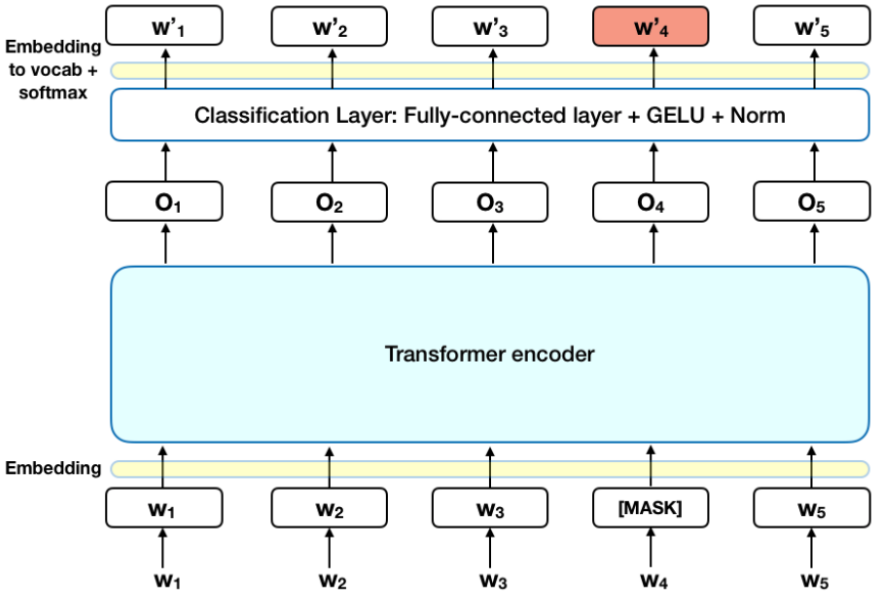
در ابتدا، توضیح مختصری در مورد مدل BERT می‌آوریم:

مدل BERT (Bidirectional Encoder Representations from Transformers) با ارائه نتایج پیشرفته در طیف گسترده ای از وظایف NLP، از جمله پاسخ به سؤال (SQuAD v1.1)، استنتاج زبان طبیعی (MNLI) و موارد دیگر، غوغایی در جامعه یادگیری ماشین ایجاد کرده است. همان‌طور که پیش‌تر در زمینه‌ی بینایی ماشین از transfer learning استفاده می‌شد، محققین نشان دادند که از این فرایند در عملیات مربوط به پردازش زبان نیز می‌توان استفاده کرد.

نوآوری فنی کلیدی BERT استفاده از آموزش دو جهته Transformer ها (که یک مدل attention محبوب هستند) در مدل‌سازی زبان است. این برخلاف تلاش‌های قبلی است که به دنباله‌ای از متن از چپ به راست یا ترکیبی از آموزش چپ به راست و راست به چپ نگاه می‌کردند. این مشخصه به مدل اجازه می‌دهد تا مفهوم یک کلمه را بر اساس تمام لغات اطراف آن (چه در سمت چپ و چه در سمت راست) یاد بگیرد. نتایج این مقاله نشان می‌دهد که یک مدل زبانی که به صورت دو جهته آموزش داده می‌شود، می‌تواند حس عمیق‌تری از بافت و جریان زبان نسبت به مدل‌های زبانی تک جهتی داشته باشد.

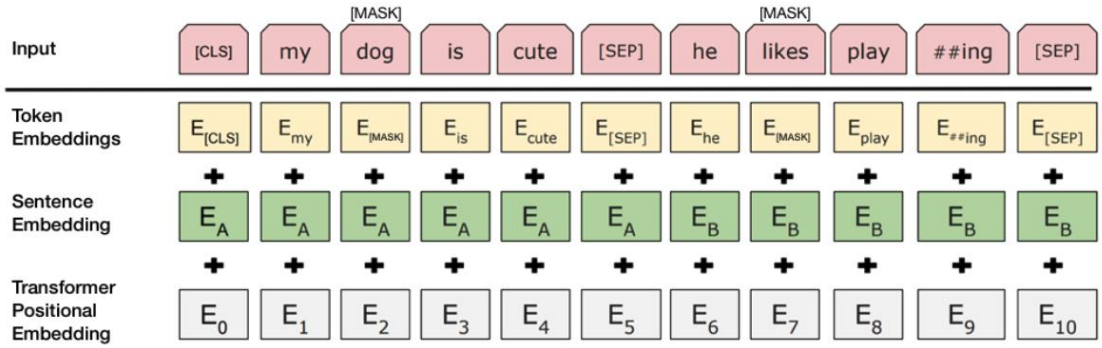
Transformerها شامل دو مکانیسم جداگانه هستند - یک انکودر که ورودی متن را می خواند و یک دیکودر که یک پیش‌بینی برای تسک مورد نظر ارائه می‌دهد. از آنجایی که هدف BERT تولید یک مدل زبان است، تنها مکانیزم انکودر ضروری است. با استفاده از این قابلیت دوطرفه، BERT روی دو هدف متفاوت اما مرتبط NLP از قبل آموزش دیده است: مدل‌سازی زبان ماسک‌شده (Masked Language Modeling) و پیش‌بینی جمله بعدی (Next Sentence Prediction).

در مورد اول(MLM) هدف آن است که در ابتدا کلماتی از متن پنهان شوند، و سپس برنامه بتواند بر اسا لغات اطراف آن‌ها، کلمات پنهان را حدس بزند. در تصویر زیر می‌توان مشاهده کرد که یک لایه‌ی طبقه‌بندی بر روی خروجی انکودر قرار گرفته‌شده و با ضرب در ماتریس embedding، خروجی‌ها از توکن به واژه تبدیل می‌شوند. در نهایت نیز توسط تابع سافت‌مکس، احتمال وجود هر لغت محاسبه می‌شود.



شکل ۵۳: استفاده از مدل **BERT** برای پیشبینی کلمات پنهان‌شده

در مورد دوم (NSP) نیز هدف این است که برنامه پیش‌بینی کند که آیا دو جمله داده شده یک ارتباط منطقی و ترتیبی دارند یا اینکه رابطه آنها تصادفی است. ۵۰ درصد جملات داده‌شده برای آموزش دارای ارتباط منطقی هستند و ۵۰ درصد بدون ارتباط‌اند. در تصویر زیر، مدل این فرایند آورده شده‌است:



شکل ۵۴: استفاده از مدل **BERT** برای پیشبینی ارتباط دو جمله ورودی

یک توکن CLS در ابتدای جمله اول و توکن‌های SEP در انتهای هر جمله می‌آیند، همچنین توکن‌های E_A و E_B به منظور مشخص کردن کلمات هر کلمه و در نهایت توکن‌هایی مربوط به ترتیب کلمات نیز اضافه می‌شوند. در نهایت همه‌ی این توالی ورودی به مدل ترنسفورمر رفته، خروجی توکن CLS به بردار با شکل ۲ در ۱ تبدیل می‌شود و احتمال اینکه آیا جمله بعدی مرتبط است یا خیر، توسط لایه سافت‌مکس محاسبه می‌شود.

مدل BERT در حالت large دارای ۳۴۵ میلیون پارامتر و در حالت base دارای ۱۱۰ میلیون پارامتر است. همچنین سرعت همگرایی این مدل از مدل‌های یک‌طرفه کمتر است، اما پس از چند ایپاک ابتدایی،

دقت بالاتری ارائه می‌دهد. این مدل قابلیت fine-tune کردن نیز دارد. برای مثال اهداف طبقه بندی مانند تجزیه و تحلیل احساسات مشابه طبقه بندی جمله بعدی، با افزودن یک لایه classification در بالای خروجی Transformer برای توکن [CLS] انجام می شود.

الف) پیش‌پردازش بر روی داده‌ها

با مقایسه‌ی دو فایل متنی داده‌های خام و داده‌های پیش‌پردازش‌شده، می‌توان دریافت که تغییراتی وجود دارند که در این بخش به بررسی هرکدام می‌پردازیم:

۱- اولین تغییر مشاهده‌شده، مربوط به تغییر یوزرهای کاربری به <user> است. این مورد از این جنبه اهمیت دارد که اسم هر یوزر معمولا حاوی اطلاعات خاصی نیست و اگر هم بر فرض شبکه بتواند میان توییت‌ها و اسامی اکانتهای کاربری ارتباطی بیابد، ممکن است سیاست‌های اعمالی بر روی کاربران دیگری با اسامی مشابه، همراه با بایاس منفی باشد. همچنین آیدی هر فرد می‌تواند در گذر زمان عوض شود و لزوما حتی برای اعمال سیاست علیه فرد خاطی هم شاید مفید نباشد.

۲- مورد دیگر مشاهده شده، حذف یک سری اطلاعات اضافه مانند سه نقطه (...) و یا چند علامت تعجب (!!!) است که تاثیری در محتوای حرفی ندارد و وجود یک علامت نیز کفایت می‌کند.

۳- لینک‌های موجود در توییت‌ها به صورت <url> نوشته شده‌اند. لینک‌ها معمولا حاوی رشته‌ای از کاراکترها هستند که معنا و مفهوم خاصی ندارند و پردازش آن‌ها اگر به شکل کلمات معنادار عادی باشد، باعث می‌شود فرایند آموزش به اشتباه صورت گیرد. می‌توان محتویات لینک‌ها را در مدلی جداگانه بررسی کرد و بر اساس آن‌ها قضاوت بهتری در مورد نوع توییت انجام داد.

۴- هشتگ‌ها نیز در این پیش‌پردازش حذف شده‌اند. البته احتمالا این مورد اثر منفی بر روی فرایند تشخیصی ما داشته باشد. چرا که هشتگ‌های بحث برانگیز در مورد موضعات خاص را می‌توان راحت‌تر بررسی کرد و احتمال وقوع توییت‌های نژادپرستانه یا توهین‌آمیز در آن‌ها بیشتر است.

۵- اموجی‌ها نیز در پردازش اولیه حذف شده‌اند و به جای آن‌ها، <emotion> قرار داده شده‌است. این اموجی‌ها که به صورت کد Decimal HTML Entity در فایل خام وجود دارند، توالی کاراکترها و ارقام است و نمی‌تواند مانند کلمات عادی بررسی شود. البته می‌توان نوع ایموجی را (مثلا خنده یا گریه) به صورت کلمه‌ای به عنوان یک ویژگی برای متن تعریف کرد.

۶- مورد بعدی، lowercase کردن تمامی حروف است. این کار باعث می‌شود میان کلماتی که با حروف بزرگ یا کوچک شروع می‌شوند، تفاوت خاصی وجود نداشته باشد و از طرفی احتمال خطای شبکه را کم می‌کند.

۷- از موارد دیگری که انجام دادن آن‌ها مفید است و در این پیش پردازش صورت نگرفته است، تبدیل بعضی مخففها به کلمات اصلی است. برای مثال تبدیل u به you، یا تبدیل I’ma به I’m going to و idk به I don’t know. در این صورت شبکه راحت‌تر می‌تواند منظور کلی توییت را درک کند و مدل معیار که بر روی دیتاست‌های علمی و با زبان رسمی مانند ویکیپدیا آموزش دیده‌است، دقت بالاتری خواهد داشت.

در مرحله اول پیاده‌سازی لازم است که این دیتاست خوانده شود و داده‌های آموزش و تست و ارزیابی از یکدگیر جدا شوند. به همین منظور از ۱۵ درصد از داده‌ها به عنوان داده تست و ۱۵ درصد به عنوان داده ارزیابی جدا شد.

پس از این مرحله لازم است داده‌ها را به فرمت Dataset پکیج Torch در آوریم تا دسترسی به داده‌ها ساده‌تر صورت بگیرد. به همین منظور یک کلاس از کلاس Dataset ارث برده شد. در انتها نیز یک متد به این کلاس اضافه کردیم که مربوط به tokenize کردن داده‌های متنی با استفاده از داده‌های متنی با استفاده از Bert_Tokenizer است که داده‌ها را به فرمت مد نظر این شبکه در می‌آورد. این دیتا ست در تصویر زیر قرار گرفته است. همچنین برای استفاده از tokenizer های دیگر لازم است که خط اول __init__ تغییر پیدا کند. (در اینجا از hateBERT دیده می‌شود که می‌توان از 'bert-base-uncased' نیز استفاده کرد که مدل Bert اصلی محسوب می‌شود)

```
class TrainDataset(Dataset):
    def __init__(self,X,y,text_len):
        self.embedder = Bert_Tokenizer('GroNLP/hateBERT')
        self.texts = X
        self.labels = y
        self.max_len = text_len

    def __len__(self):

        return len(self.texts)

    def __getitem__(self,idx):

        tokens,mask,tokens_len = self.mask_using_tokenizer(self.texts[idx],self.text_len)
        label = self.labels[idx]
        return [torch.tensor(tokens),torch.tensor(mask),torch.tensor(tokens_len)],label

    def mask_using_tokenizer(self,text,max_len):
        tokens = []
        mask = []
        text = self.bert_encode.encode(text)
        size = len(text)
        pads = self.bert_encode.encode(['PAD']*(max(0,max_len-size)))
        tokens[:max(max_len,size)] = text[:max(max_len,size)]
        tokens = tokens + pads[1:-1]
        mask = [1]*size+[0]*len(pads[1:-1])
        tokens_len = len(tokens)
        D
        return tokens,mask,tokens_len
```

شکل ۵۵ کلاس دیتاست مورد استفاده برای داده‌های آموزش و ولیدیشن

در مرحله tokenize کردن، یک ماکزیمم طول برای توییت‌ها در نظر گرفتیم. در صورتی که طول رشته متنی از این مقدار کمتر باشد، مابقی متن با پدینگ پر خواهد شد. پس از این مرحله نیز یک تابع به شکل زیر تعریف شد که داده‌ها را به سه دسته مد نظر تقسیم می‌کند و سپس دیتاست‌های مربوطه از آن‌ها تولید خواهد شد.

```python
def get_data_loaders():
    tokenizer = Bert_Tokenizer('GroNLP/hateBERT')
    X_train, X_test, y_train, y_test = train_test_split(dataset["tweet"].to_numpy(), dataset["label"].to_numpy().reshape(-1,1), test_size=0.15, random_state=51)
    X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.15, random_state=51)

    train_dataset = TrainDataset(X_train,y_train,text_len=120)
    train_loader = torch.utils.data.DataLoader(train_dataset,batch_size=1,shuffle=True)
    valid_dataset = TextDataset(X_val, y_val,text_len=120)
    valid_loader = torch.utils.data.DataLoader(valid_dataset,batch_size=1,shuffle=True)

    return train_loader , valid_loader
```

شکل ۵۶ نحوه ساخت **Dataloader** ها و تقسیم داده‌ها

حال می‌توانیم کلاس اصلی مربوط به ساختار شبکه و تابع forward را بسازیم. به همین منظور کلاس زیر توسعه داده شد که در لایه اول مدل Bert قرار می‌گیرد. این لایه دو خروجی pooler_output و last_hidden_state را تولید می‌کند. ما باید last_hidden_state را به عنوان ورودی شبکه LSTM در نظر بگیریم تا به این شبکه داده شود. بنابراین داده‌ها باید به فرمت سه بعدی مورد نظر LSTM در آورده شود. لازم است که در این مرحله خروجی‌ها به فرمت وان هات در آورده شوند و از یک softmax به عنوان لایه خروجی برای تشخیص کلاس‌ها استفاده می‌شود.

```python
class LSTMBertModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.bert = BertModel.from_pretrained('GroNLP/hateBERT')
        self.hidden_size = self.bert.config.hidden_size
        self.LSTM = nn.LSTM(self.hidden_size,self.hidden_size,bidirectional=True)
        self.clf = nn.Linear(self.hidden_size*2,1)

    def forward(self,inputs):
        out = self.bert(input_ids=inputs[0],attention_mask=inputs[1])
        last_hidden_state = out["last_hidden_state"]
        pooler_output = out["pooler_output"]
        last_hidden_state = last_hidden_state.permute(1, 0, 3)
        enc_hiddens, (last_hidden, last_cell) = self.LSTM(pack_padded_sequence(last_hidden_state, inputs[2].cpu()))
        output_hidden = torch.cat((last_hidden[0], last_hidden[1]), dim=1)
        output_hidden = F.dropout(output_hidden,0.2)
        output = self.clf(output_hidden)

        return F.sigmoid(output)
```

شکل ۵۷ شبکه طراحی شده برای مدل مورد استفاده

پس از این مرحله کد ما دچار باگ شد. با استفاده از تکه کد زیر که از یک کد اینترنتی پیدا شد، سعی کردیم که این شبکه آموزش داده شود. اما با این حال نتوانستیم که از کتابخانه Pytorch به نحو درستی استفاده کنیم و کد قابلیت یادگیری نداشت و متاسفانه به دلیل محدودیت زمانی، امکان اصلاح کد نیز وجود نداشت.

```python
def run(log_interval=100,epochs=2,lr=0.000006):
    train_loader ,valid_loader = get_data_loaders()
    model = LSTMBertModel()
    device = 'cuda:0' if torch.cuda.is_available() else 'cpu'

    criterion = torch.nn.CrossEntropyLoss()
    optimizer = AdamW(model.parameters(), lr=lr)
    lr_scheduler = ExponentialLR(optimizer, gamma=0.90)
    trainer = create_supervised_trainer1(model.to(device), optimizer, criterion, device=device)
    evaluator = create_supervised_evaluator1(model.to(device), metrics={'BCELoss': Loss(criterion)}, device=device)

    if log_interval is None:
        e = Events.ITERATION_COMPLETED
        log_interval = 1
    else:
        e = Events.ITERATION_COMPLETED(every=log_interval)

    desc = "loss: {:.4f} | lr: {:.4f}"
    pbar = tqdm(
        initial=0, leave=False, total=len(train_loader),
        desc=desc.format(0, lr)
    )


    try:
        trainer.run(train_loader, max_epochs=epochs)
    except Exception as e:
        import traceback
        print(traceback.format_exc())
        pass
    return model
```

شکل ۵۸ تکه کد مورد استفاده برای یادگیری شبکه

د) مقایسه نتایج قبلی و علت طراحی مدل HateBERT

مدل BERT اولین بار در سال ۲۰۱۸ معرفی شد. این مدل با استفاده از داده‌های بدون لیبل (۸۰۰ میلیون داده از BooksCorpus (شامل جملات و کلمات از فیلم‌ها) و ۲۵۰۰ میلیون لغت از ویکیپدیا) آموزش دیده است. این مدل عمومیت خوبی دارد و دارای دقت بالایی در پردازش‌های مربوط به NLP است. اما محققین در تلاش‌اند تا با استفاده از fine-tune کردن آن برای کاربری‌های متفاوت، دقت آن را بهبود بخشند. برای مثال مدل‌های مختلفی از جمله AlBERTo (Polignano et al., 2019) و یا TweetEval (Barbieri et al., 2020) برای توییتر، BioBERT (Lee et al., 2019) برای زبان انگلیسی بایومدیکال و FinBERT(Yang et al., 2020) برای زبان انگلیسی مورد استفاده در حوزه مالی، توسعه داده شده‌اند.

در حوزه‌ی متون توهین‌امیز نیز با بررسی مدل‌های ارائه شده بر روی دیتاست‌های خاص (برای مثال OffensEval 2019) مشاهده شده‌است که نقش پیش‌پردازش بسیار پررنگ‌تر از هایپرپارامترها و تعداد ایپاک‌ها بوده‌است. بنابراین تصمیم گرفته‌شد تا شبکه‌ای برای تشخیص این نوع متون، با re-train کردن مدل اصلی BERT انجام گیرد (مقاله در این لینک)

این آموزش مجدد، بر روی دیتاست RAL-E آموزش دیده شده‌است. این دیتاست شامل پست‌های جوامع کاربری Reddit است که به خاطر استفاده از متون توهین‌آمیز و تحقیرآمیز و یا کلمات ناسزا، از این وبسایت بن شده بودند. این مجموعه شامل 1,492,740 پیام بوده‌است که از ژانویه ۲۰۱۲ تا ژوئن ۲۰۱۵ جمع‌آوری شده‌است.

عملکرد این شبکه در مقایسه با مدل اصلی BERT بر روی دیتاست‌های مختلفی تست شده‌است که در جدول زیر خلاصه شده‌است:

جدول ۲۰: مقایسه عملکرد شبکه‌های **BERT** و **HateBERT** بر روی سه دیتاست مختلف

Dataset	Model	Macro F1	Pos. class - F1
OffensEval 2019	BERT	.803±.006	.715±.009
	HateBERT	**.809±.008**	**.723±.012**
	Best	.829	.599
AbusEval	BERT	.727±.008	.552±.012
	HateBERT	**.765±.006**	**.623±.010**
	Caselli et al. (2020)	.716±.034	.531
HatEval	BERT	.480±.008	.633±.002
	HateBERT	**.516±.007**	**.645±.001**
	Best	.651	–

همان‌طور که انتظار می‌رفت، عملکرد HateBERT بر روی این دیتاست‌ها که مخصوص کلمات توهین‌امیز یا تحقیرآمیز و abusive بوده‌اند، نتایج بهتری داشته است