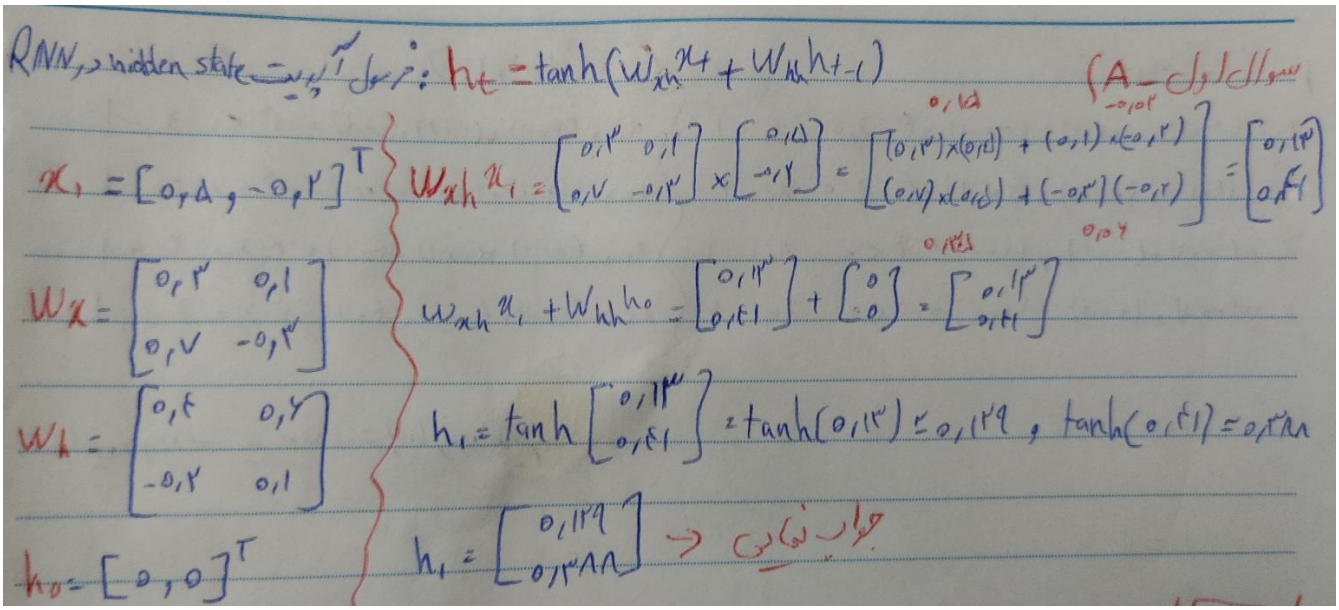
تمرین ششم شبکه عصبی

سوال اول

A)



B) شبکه عصبی RNN مانند یک سیستم حافظه عمل می کند که اطلاعات را به تدریج و با گذر زمان پردازش می کند. نحوه بروزرسانی hidden state در هر مرحله به شرح زیر است:

در هر مرحله زمانی، RNN به ورودی آن لحظه نگاه می کند. برای مثال در زمان t=1 ورودی x_1 برابر است با [0.5,−0.2] در ادامه این ورودی را با آنچه از گذشته به خاطر دارد ترکیب می کند که این حافظه "hidden state" نامیده می شود و در هر مرحله به روزرسانی می شود. در مرحله اول، هیچ حافظه ای از گذشته وجود ندارد، بنابراین با h_0 = [0,0] شروع می شود.

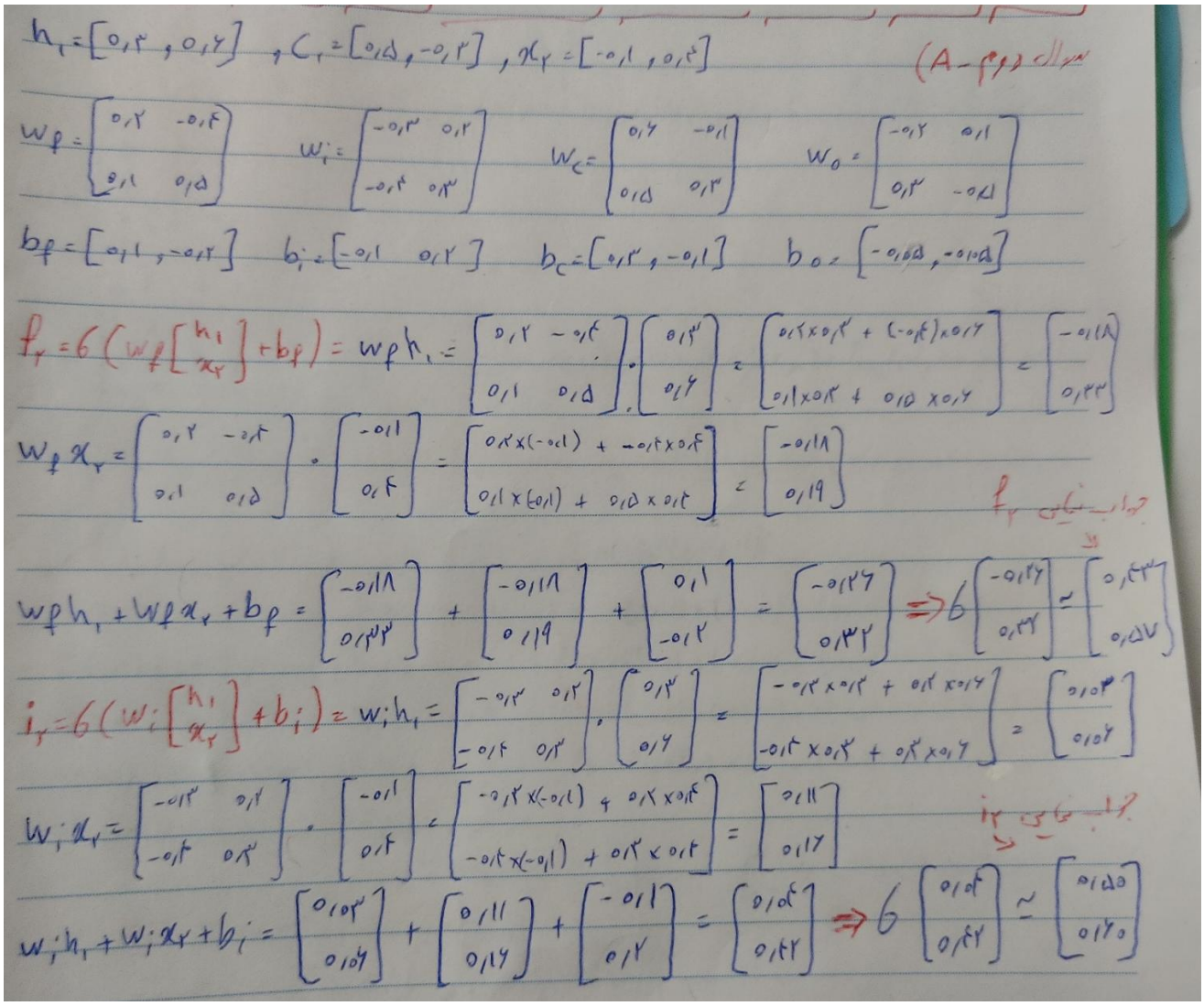
شبکه RNN از دو مجموعه وزن استفاده می کند:

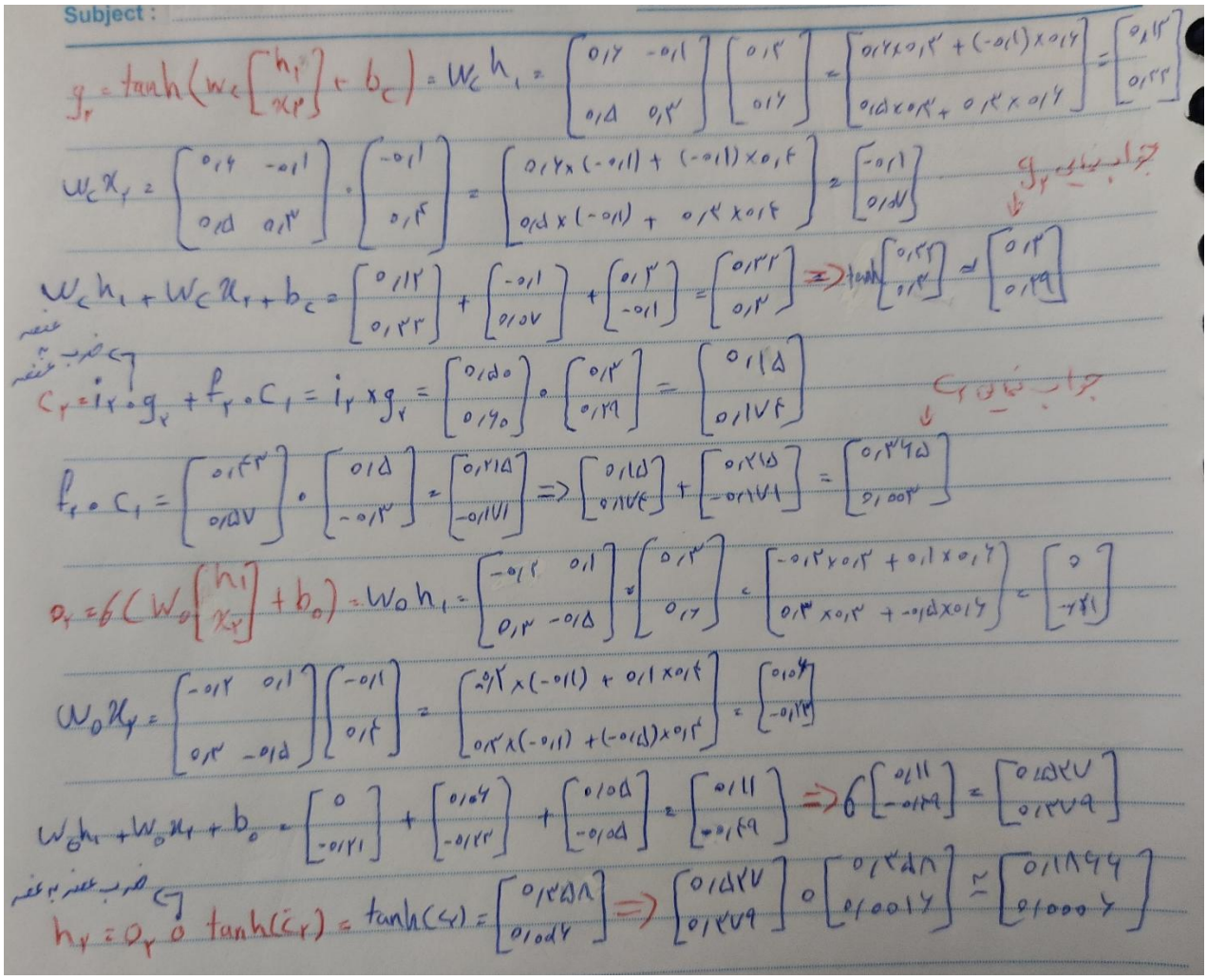
یکی W_x که تعیین می کند که ورودی فعلی چقدر اهمیت دارد و دیگری W_h که میزان اهمیت حافظه گذشته را تنظیم می کند. در ادامه این دو بخش (ورودی و حافظه) را با هم ترکیب می کند تا ترکیبی از آنچه اکنون در حال رخ دادن است و آنچه قبلا رخ داده است را ایجاد کند. سپس یک تابع فعالسازی غیر خطی (tanh) اعمال می کند که اطمینان حاصل می کند hidden state در محدوده 1-

تا 1+ باقی بماند. این کار کمک می کند تا از مقادیر افراطی جلوگیری شود و یادگیری پایدار باشد.
پس از این فرآیند hidden state RNN خود را به روز می کند که این hidden state جدید اکنون هم شامل اطلاعات ورودی فعلی و هم حافظه آنچه قبلا آمده است می باشد.
شبکه عصبی RNN این امکان را می دهد که دنباله ها را پردازش کنیم و با حفظ اطلاعات گذشته از طریق hidden state ها، الگوها و وابستگی ها را در داده های دنباله دار شناسایی کنیم که آن را برای وظایفی مانند مدل سازی زبان، پیش بینی سری های زمانی و غیره مناسب می سازد.

سوال دوم

A و B





C) LSTM (Long Short-Term Memory) نوع خاصی از معماری شبکه های RNN هستند که برای بهبود شناسایی وابستگی های بلندمدت و کاهش مشکل گرادیان در RNN های پایه طراحی شده اند. یک واحد LSTM شامل چندین جز کلیدی است، از جمله gate ها و یک سلول حافظه. LSTM مانند یک سیستم بایگانی پیشرفته در مغز ما است که این سیستم برای مدیریت اینکه چه چیزی را نگه داریم و یا چه چیزی را دور بیندازیم و چه چیزی را همین حالا به اشتراک بگذاریم طراحی شده است که آن را بسیار بهتر از یک سیستم بایگانی پایه مانند RNN سنتی می کند.در ادامه هر یک از این اجزا را شرح می دهیم:

سلول حافظه (C_t):

این بخش هسته LSTM است که اطلاعات را در طول زمان حفظ می کند. سلول حافظه یک ویژگی کلیدی است که به LSTM ها کمک می کند وابستگی های بلندمدت را به خاطر بسپارند.

بخش Hidden State (h_t):

نمایانگر خروجی واحد LSTM برای زمانی فعلی است و به عنوان ورودی برای لایه بعدی عمل می کند. این از سلول حافظه مشتق شده و توسط دروازه خروجی تنظیم می شود.

دروازه ها (Gates):

Gate ها مکانیزم هایی هستند که از تابع فعال سازی سیگموید و عملیات نقطه ای مثل ضرب عنصر به عنصر تشکیل شده اند و آن ها جریان اطلاعات را در داخل و خارج از سلول حافظه کنترل می کنند.

سه gate اصلی:

- **gate ورودی (i_t):** این gate نسبت اطلاعات جدیدی را که باید در سلول حافظه "نوشته" شود را تعیین می کند.
- **gate فراموشی (f_t):** تعیین می کند که کدام بخش های محتوای فعلی سلول حافظه (C_{t-1}) برای حالت بعدی حفظ یا حذف خواهند شد. این بخش مانع می شود که مدل حافظه اش را با اطلاعات نامربوط شلوغ کند.
- **gate خروجی (o_t):** تعیین می کند که چه مقدار از محتوای سلول حافظه باید بر hidden state در مرحله زمانی فعلی تاثیر بگذارد.

چگونه LSTM به محدودیت های RNN کمک می کند؟

سلول حافظه (C_t) اجازه می دهد تا اطلاعات بدون مانع از طریق مراحل زمانی جریان یابد و از ناپدید شدن گرادیان ها که یک محدودیت در RNN بود جلوگیری کند. gate فراموشی (f_t) تصمیم می گیرد چه چیزی را کنار بگذارد، تا اطلاعات نامربوط گذشته حافظه را شلوغ نکند. gate ورودی (i_t) اطمینان حاصل می کند که تنها اطلاعات جدید مرتبط ذخیره شوند. gate خروجی (o_t) کنترل می کند که کدام بخش از حافظه داخلی به خروجی فعلی کمک می کند. با هماهنگ کردن این gate ها، LSTMها هم وابستگی های بلندمدت و هم دینامیک های زمانی را مدیریت می کنند و این امر آن ها را برای کارهایی مانند پیش بینی سری های زمانی، تولید دنباله و شناسایی گفتار قدرتمند می سازد.

سوال سوم

A) نوت بوک HW6_Q3 شامل پیاده سازی یک مدل پیش بینی آب و هوا با استفاده از LSTM است که ابتدا دیتاست لود میشود و بررسی می شود که مقادیر null و تکراری دارد یا خیر. در ادامه داده های موجود در ستون دوم را انتخاب و به یک آرایه NumPy تبدیل می کند.

```
def df_to_XY(df,window_size=10):
 X_train=[]
 y_train=[]

 for i in range(10,len(training_set)):
    X_train.append(training_set[i-10:i,0])
    y_train.append(training_set[i,0])

 X_train, y_train = np.array(X_train), np.array(y_train)
 return X_train, y_train
```

تابع df_to_XY داده ها را به صورت دسته های پشت سرهم تقسیم می کند. پارامتر window_size تعیین می کند که هر دسته چندتا داده را در بر بگیرد. X_train شامل دسته هایی از داده های گذشته به طول window_size و y_train مقدار بعدی (هدف) را برای هر دسته اضافه می کند. در انتها این دو لیست به آرایه های NumPy تبدیل و برگردانده می شوند.

تعداد کل نمونه های تولیدشده برای X و y به کمک تابع مربوطه تولید و سپس چاپ می شوند.

در ادامه این بخش داده های train و val و test جدا می شود.

```
WINDOW = 10
X,y = df_to_XY(df,WINDOW)
print(len(X),len(y))
X_train = X[:800]
y_train = y[:800]
X_val = X[800:1000]
y_val = y[800:1000]
X_test = X[1000:]
x_test = y[1000:]
```

```
X_train = np.reshape(X_train,(X_train.shape[0],X_train.shape[1],1))
X_val = np.reshape(X_val,(X_val.shape[0],X_val.shape[1],1))
X_test = np.reshape(X_test,(X_test.shape[0],X_test.shape[1],1))
```

در این بخش داده تغییر شکل می دهند تا برای مدل LSTM آماده شوند. داده ها به سه بعد (تعداد نمونه ها، طول دنباله، تعداد ویژگی ها) تبدیل می شوند.

معماری مدل:

```
regressor = Sequential()

regressor.add(LSTM(units=50, return_sequences = True, input_shape=(X_train.shape[1], 1)))
regressor.add(Dropout(0.2))

regressor.add(LSTM(units=50, return_sequences = True))
regressor.add(Dropout(0.2))

regressor.add(LSTM(units=50, return_sequences = True))
regressor.add(Dropout(0.2))

regressor.add(LSTM(units=50))
regressor.add(Dropout(0.2))

regressor.add(Dense(units=1))

regressor.compile(optimizer='adam',loss='mean_squared_error')
```

ابعاد ورودی این مدل برابر است با (10,1)

سپس یک لایه LSTM با 50 نورون و return_sequences=True است تا خروجی این لایه برای تمام گام های زمانی تولید شود و به لایه بعدی ارسال شود. در ادامه یک لایه dropout با نرخ 0.2 برای جلو گیری از overfitting قرار داده شده است.

این دو سه بار دیگر هم تکرار شده با این تفاوت که لایه آخر آن return_sequences=False است و خروجی نورون آخر را به لایه بعدی می فرستد.

لایه بعدی یک لایه Fully Connected با یک نورون است که هدف این لایه تبدیل خروجی LSTM به یک مقدار نهایی (پیش بینی یک عدد) است.

در اینجا از بهینه ساز آدام و تابع خطای MSE استفاده شده است.

در ادامه مدل را با 100 ایپاک و batch_size=32 آموزش می دهیم که در ایپاک آخر به loss زیر رسیدیم:

```
Epoch 100/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - loss: 7.7149 - val_loss: 9.3394
```

B) در این بخش می خواهیم مدل LSTM و GRU و SimpleRNN را باهم مقایسه کنیم. مدل ها را با یک معماری مشابه ساختیم تا بتوانیم عملکرد هر یک را با هم مقایسه کنیم.

مدل LSTM:

Layer (type)	Output Shape	Param #
lstm_56 (LSTM)	(None, 10, 50)	10,400
dropout_100 (Dropout)	(None, 10, 50)	0
lstm_57 (LSTM)	(None, 10, 50)	20,200
dropout_101 (Dropout)	(None, 10, 50)	0
lstm_58 (LSTM)	(None, 10, 50)	20,200
dropout_102 (Dropout)	(None, 10, 50)	0
lstm_59 (LSTM)	(None, 50)	20,200
dropout_103 (Dropout)	(None, 50)	0
dense_25 (Dense)	(None, 1)	51

Total params: 71,051 (277.54 KB)
Trainable params: 71,051 (277.54 KB)
Non-trainable params: 0 (0.00 B)

مدل GRU:

Layer (type)	Output Shape	Param #
gru_32 (GRU)	(None, 10, 50)	7,950
dropout_104 (Dropout)	(None, 10, 50)	0
gru_33 (GRU)	(None, 10, 50)	15,300
dropout_105 (Dropout)	(None, 10, 50)	0
gru_34 (GRU)	(None, 10, 50)	15,300
dropout_106 (Dropout)	(None, 10, 50)	0
gru_35 (GRU)	(None, 50)	15,300
dropout_107 (Dropout)	(None, 50)	0
dense_26 (Dense)	(None, 1)	51

Total params: 53,901 (210.55 KB)
Trainable params: 53,901 (210.55 KB)
Non-trainable params: 0 (0.00 B)

مدل SimpleRNN:

Layer (type)	Output Shape	Param #
simple_rnn_12 (SimpleRNN)	(None, 10, 50)	2,600
dropout_108 (Dropout)	(None, 10, 50)	0
simple_rnn_13 (SimpleRNN)	(None, 10, 50)	5,050
dropout_109 (Dropout)	(None, 10, 50)	0
simple_rnn_14 (SimpleRNN)	(None, 10, 50)	5,050
dropout_110 (Dropout)	(None, 10, 50)	0
simple_rnn_15 (SimpleRNN)	(None, 50)	5,050
dropout_111 (Dropout)	(None, 50)	0
dense_27 (Dense)	(None, 1)	51

Total params: 17,801 (69.54 KB)
Trainable params: 17,801 (69.54 KB)
Non-trainable params: 0 (0.00 B)

از نظر تعداد پارامتر ها مدل SimpleRNN از همه کمتر است و بعد از آن مدل GRU و در نهایت مدل LSTM بیشترین پارامتر را دارد.

هر سه مدل را با هم آموزش دادیم و به نتایج زیر رسیدیم:

مدل LSTM:

```
Epoch 100/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - loss: 8.3375 - val_loss: 9.6318
```

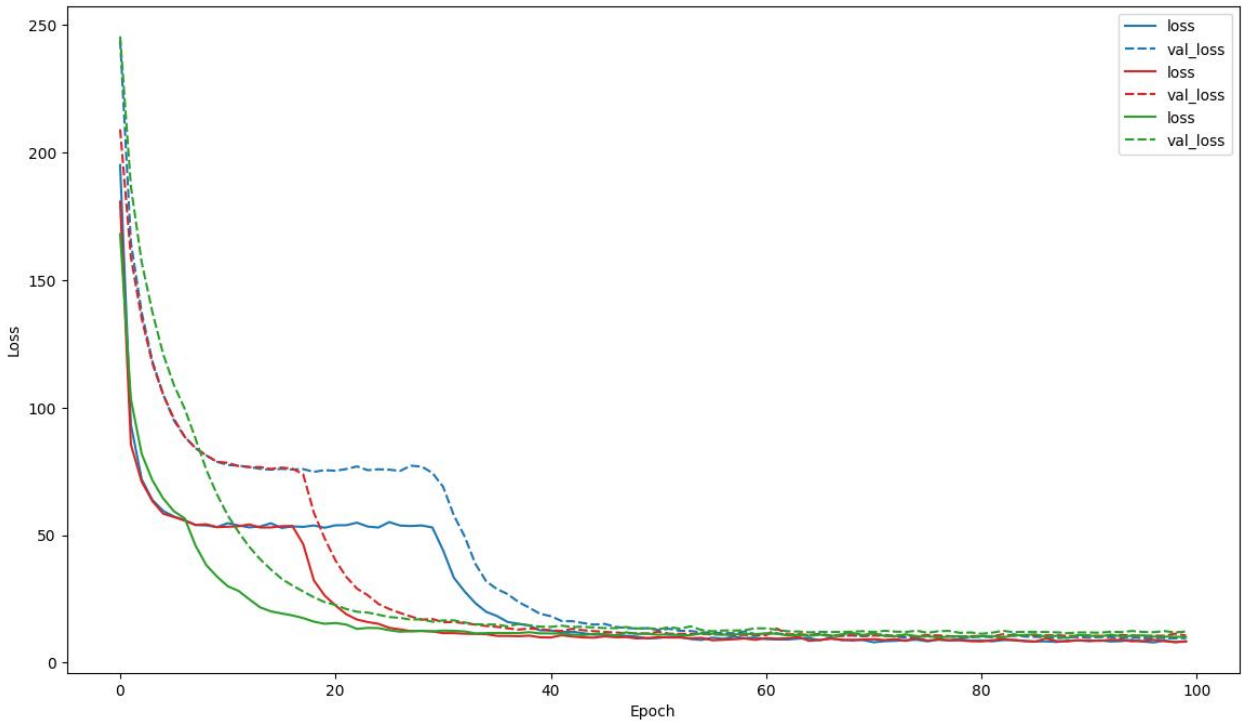
مدل GRU:

```
Epoch 100/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 36ms/step - loss: 9.1271 - val_loss: 10.6631
```

مدل SimpleRNN:

```
Epoch 100/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - loss: 9.2579 - val_loss: 12.1962
```

در اینجا مشخص می شود که loss در ایپاک آخر برای مدل LSTM از همه بهتر است. در مدل SimpleRNN مقدار loss روی داده های ولیدیشن بسیار بالاتر از داده های آموزشی است که به این معنی است که تعمیم خوبی ندارد.



این هم یک نمودار برای بررسی بیشتر نسبت loss به epoch در هر سه مدل.

آبی : LSMT

قرمز : GRU

سبز : SimpleRNN

سوال چهارم

```
X_train , X_test , y_train , y_test = train_test_split(X,y,test_size=0.2, random_state=42)
```

برای این سوال ابتدا داده ها را به دو بخش Train و Test تبدیل کردیم.

```
# Maximum Length in X_train_sequences
maxlen = max(len(seq) for seq in X_train_sequences)
```

در این بخش از بین تعداد کاراکتر برای هر sequence ماکسیمم می گیریم تا بتوانیم همه را با پدینگ یک اندازه کنیم.

```
model = Sequential()

model.add(Embedding(input_dim=input_size, output_dim=100, input_length=79))

model.add(Bidirectional(GRU(units=128, return_sequences=False)))

model.add(BatchNormalization())

model.add(Dropout(0.5))

model.add(Dense(units=64, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(units=6, activation='softmax'))
```

ابتدا یک مدل ایجاد می کنیم و اولین لایه را embedding قرار می دهیم که بردارهای ورودی را به بردارهای Dense با ابعاد ثابت تبدیل می کند.

لایه بعدی یک GRU دو طرفه است که 128 نورون دارد و خروجی آخرین لایه را بر می گرداند. در ادامه یک لایه Batch Normalization داریم که این لایه داده های خروجی از لایه قبلی را نرمال سازی می کند.

سپس یک لایه Dropout برای جلوگیری از Overfitting با نرخ 0.5 داریم. یک لایه Dense داریم که این لایه تمام نورون های لایه قبلی را به نورون های این لایه متصل می کند که 64 تا نورون دارد و از تابع فعالسازی ReLU استفاده می کند.

در ادامه یک لایه Dropout دیگر و یک لایه Dense با 6 نورون با تابع فعال ساز Softmax که مقادیر خروجی را به احتمال برای هر دسته تبدیل می کند.

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

# Display the model summary
model.summary()
```

سپس مدل را با بهینه ساز آدام و تابع ضرر sparse_categorical_crossentropy کامپایل میکنیم و خلاصه آن به شرح زیر است:

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 79, 100)	6,000,000
bidirectional (Bidirectional)	(None, 256)	176,640
batch_normalization (BatchNormalization)	(None, 256)	1,024
dropout (Dropout)	(None, 256)	0
dense (Dense)	(None, 64)	16,448
dropout_1 (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 6)	390

Total params: 6,194,502 (23.63 MB)
Trainable params: 6,193,990 (23.63 MB)
Non-trainable params: 512 (2.00 KB)

```python
# Model Train
history = model.fit(
    X_train_padded, y_train,
    validation_data=(X_test_padded, y_test),
    batch_size=32,
    epochs=5,
    verbose=1
)
```

در ادامه مدل را با 5 ایپاک آموزش می دهیم که نتایج آن برابر است با :

```
Epoch 1/5
10404/10404 ━━━━━━━━━━━━━━━━━━━━ 1170s 112ms/step - accuracy: 0.9055 - loss: 0.2403 - val_accuracy: 0.9370 - val_loss: 0.0990
Epoch 2/5
10404/10404 ━━━━━━━━━━━━━━━━━━━━ 1164s 112ms/step - accuracy: 0.9363 - loss: 0.1118 - val_accuracy: 0.9386 - val_loss: 0.0967
Epoch 3/5
10404/10404 ━━━━━━━━━━━━━━━━━━━━ 1221s 112ms/step - accuracy: 0.9386 - loss: 0.1037 - val_accuracy: 0.9393 - val_loss: 0.0980
Epoch 4/5
10404/10404 ━━━━━━━━━━━━━━━━━━━━ 1202s 110ms/step - accuracy: 0.9395 - loss: 0.0994 - val_accuracy: 0.9396 - val_loss: 0.0937
Epoch 5/5
10404/10404 ━━━━━━━━━━━━━━━━━━━━ 1195s 115ms/step - accuracy: 0.9410 - loss: 0.0961 - val_accuracy: 0.9407 - val_loss: 0.0949
```

در ادامه بهترین ایپاک را پیدا می کنیم و نمایش می دهیم :

```python
# Get the epoch with the highest validation accuracy
best_epoch = np.argmax(history.history['val_accuracy'])
print(f"The epoch with the highest validation accuracy is: {best_epoch + 1}")
```

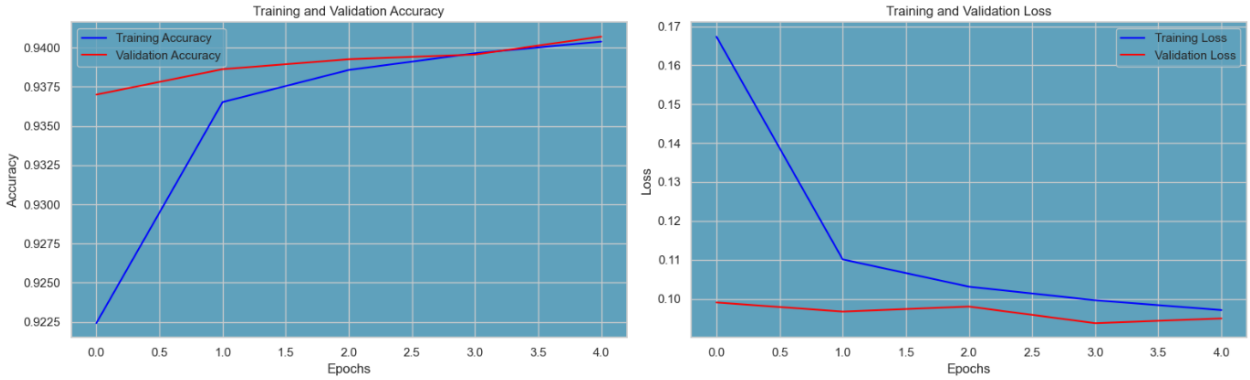
```
The epoch with the highest validation accuracy is: 5
```

سپس دو نمودار یکی به نسبت دقت داده های train و test و دیگری برای نمایش loss داده های train و test رسم کردیم.

```python
# Plot training and validation accuracy
axs[0].plot(history.history['accuracy'], label='Training Accuracy', color='blue')
axs[0].plot(history.history['val_accuracy'], label='Validation Accuracy', color='red')
axs[0].set_title('Training and Validation Accuracy')
axs[0].set_xlabel('Epochs')
axs[0].set_ylabel('Accuracy')
axs[0].legend()

# Plot training and validation loss
axs[1].plot(history.history['loss'], label='Training Loss', color='blue')
axs[1].plot(history.history['val_loss'], label='Validation Loss', color='red')
axs[1].set_title('Training and Validation Loss')
axs[1].set_xlabel('Epochs')
axs[1].set_ylabel('Loss')
axs[1].legend()

plt.tight_layout()
plt.show()
```




```
# Evaluate Test Data
test_loss, test_accuracy = model.evaluate(X_test_padded, y_test, verbose=1)
```

```
2601/2601 ━━━━━━━━━━━━━━━━━━━━ 56s 22ms/step - accuracy: 0.9413 - loss: 0.0935
```

در ادامه دقت مدل روی داده های تست را ارزیابی کردیم.

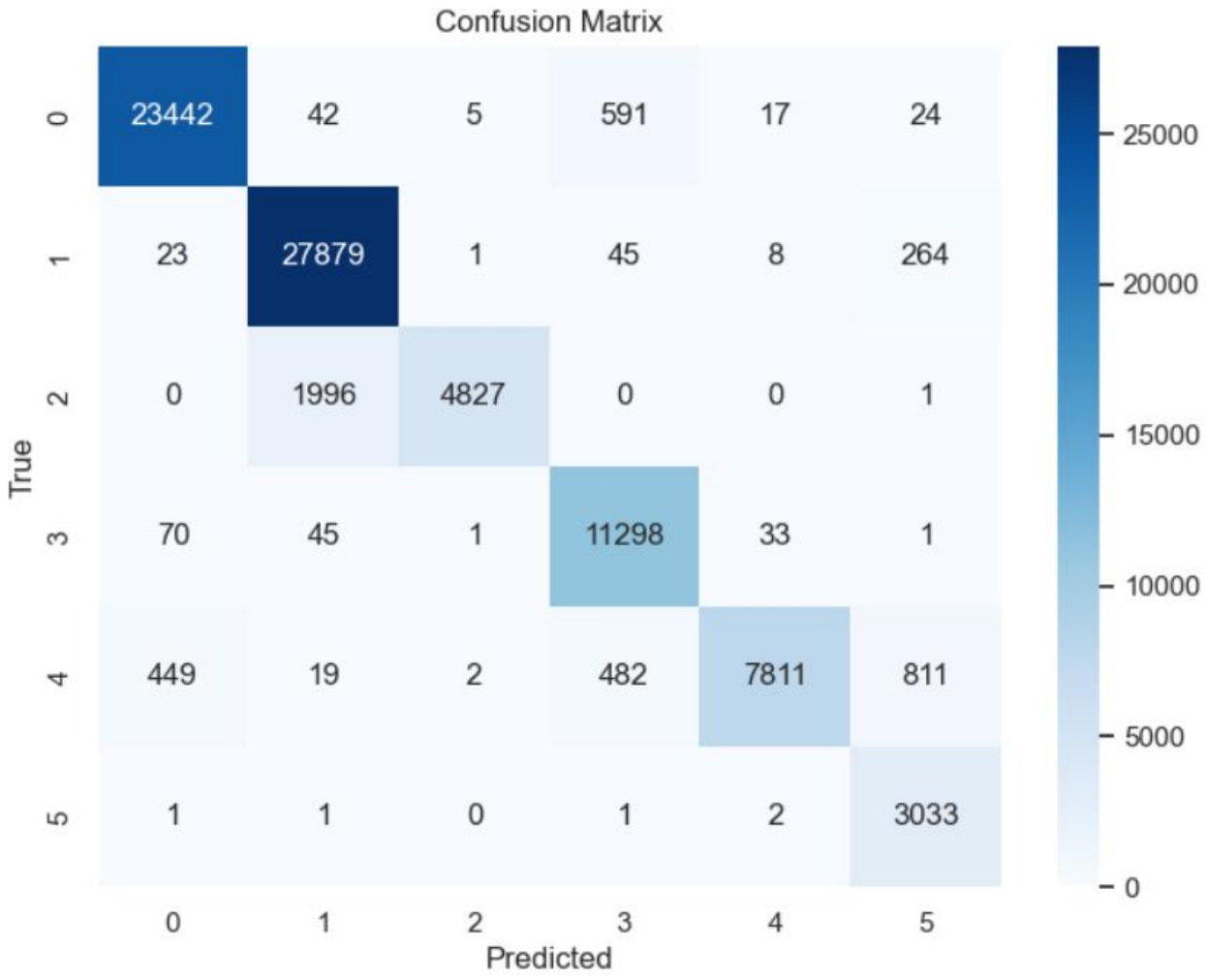
```
# Predictions On Test For Confustion Matrix
y_pred_probs = model.predict(X_test_padded)
y_pred = np.argmax(y_pred_probs, axis=1)
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

در نهایت Confusion Matrix را حساب کردیم و چاپ کردیم.

```
2601/2601 ━━━━━━━━━━━━━━━━━━━━ 61s 24ms/step
[[23442    42     5   591    17    24]
 [   23 27879     1    45     8   264]
 [    0  1996  4827     0     0     1]
 [   70    45     1 11298    33     1]
 [  449    19     2   482  7811   811]
 [    1     1     0     1     2  3033]]
```

در آخر آن را رسم کردیم:

```
# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap="Blues")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix")
plt.show()
```



محمد حقیقت - 403722042