NLP

May 2024

TP4 Report:

NLP with Sequence Models

- HACHOUD Mohammed

1. Load the **conll003-englishversion** dataset and perform pre-processing: data normalization.

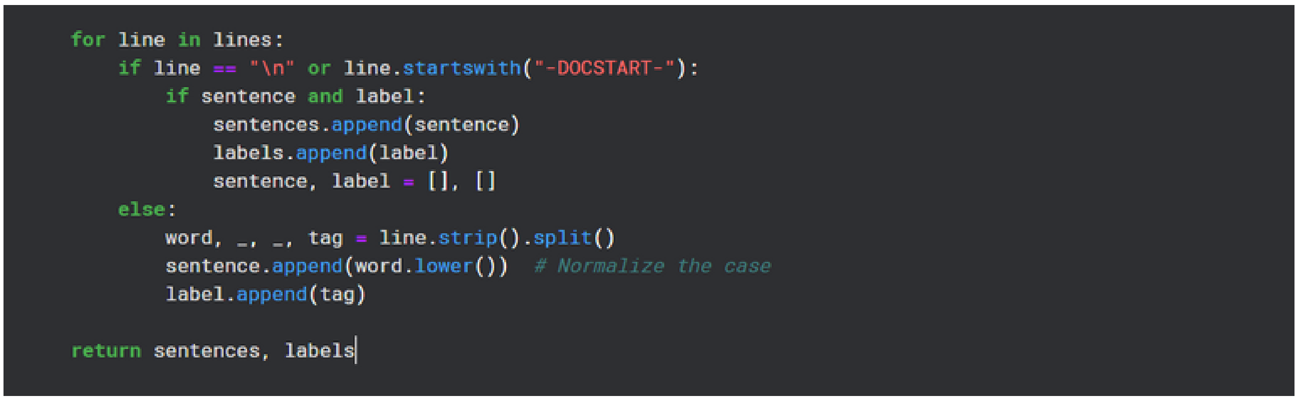
Conll003-englishversion dataset:

CoNLL-2003 is a dataset commonly used for named entity recognition (NER) and part-of-speech (POS) tagging tasks. It is derived from the Reuters corpus and consists of news articles annotated with named entities and part-of-speech tags. The dataset is divided into training, development, and test sets.
Each token in the dataset is tagged with its part-of-speech and labeled with its named entity type, such as PERSON, LOCATION, ORGANIZATION, etc. Researchers often use this dataset to train and evaluate NER and POS tagging models, as it provides a standardized benchmark for evaluating the performance of such systems.

Data Preprocessing:

Data Preparation & Case Normalization:

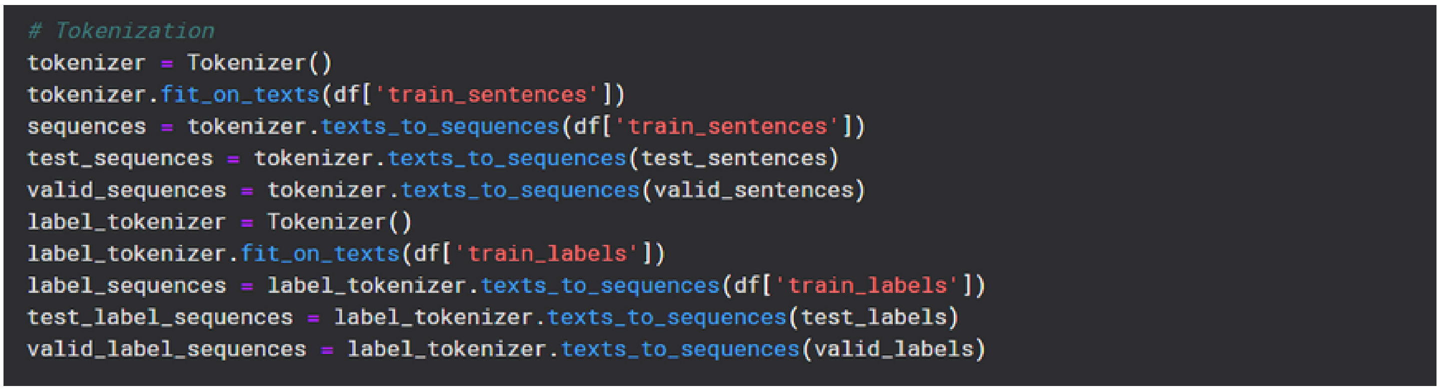
We will processes lines of text data, extracts words and corresponding tags (likely part-of-speech or named entity tags), and organizes them into sentences with their respective labels. then handles sentence boundaries and normalizes the case of words.



```
for line in lines:
    if line == "\n" or line.startswith("-DOCSTART-"):
        if sentence and label:
            sentences.append(sentence)
            labels.append(label)
            sentence, label = [], []
    else:
        word, _, _, tag = line.strip().split()
        sentence.append(word.lower())  # Normalize the case
        label.append(tag)

return sentences, labels
```

Tokenization & Vectorization:

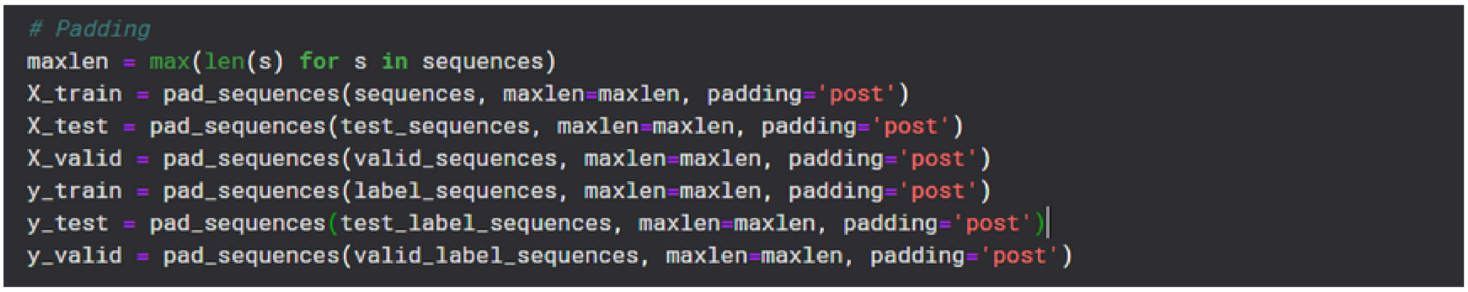
We'll break text into smaller units, Fit the Tokenizer from keras on the train Data then Converting sentences (and labels) from their original text format into sequences of integers, where each integer represents a token in the vocabulary learned by the tokenizers.



```
# Tokenization
tokenizer = Tokenizer()
tokenizer.fit_on_texts(df['train_sentences'])
sequences = tokenizer.texts_to_sequences(df['train_sentences'])
test_sequences = tokenizer.texts_to_sequences(test_sentences)
valid_sequences = tokenizer.texts_to_sequences(valid_sentences)
label_tokenizer = Tokenizer()
label_tokenizer.fit_on_texts(df['train_labels'])
label_sequences = label_tokenizer.texts_to_sequences(df['train_labels'])
test_label_sequences = label_tokenizer.texts_to_sequences(test_labels)
valid_label_sequences = label_tokenizer.texts_to_sequences(valid_labels)
```

Padding & Prepare data for Models:

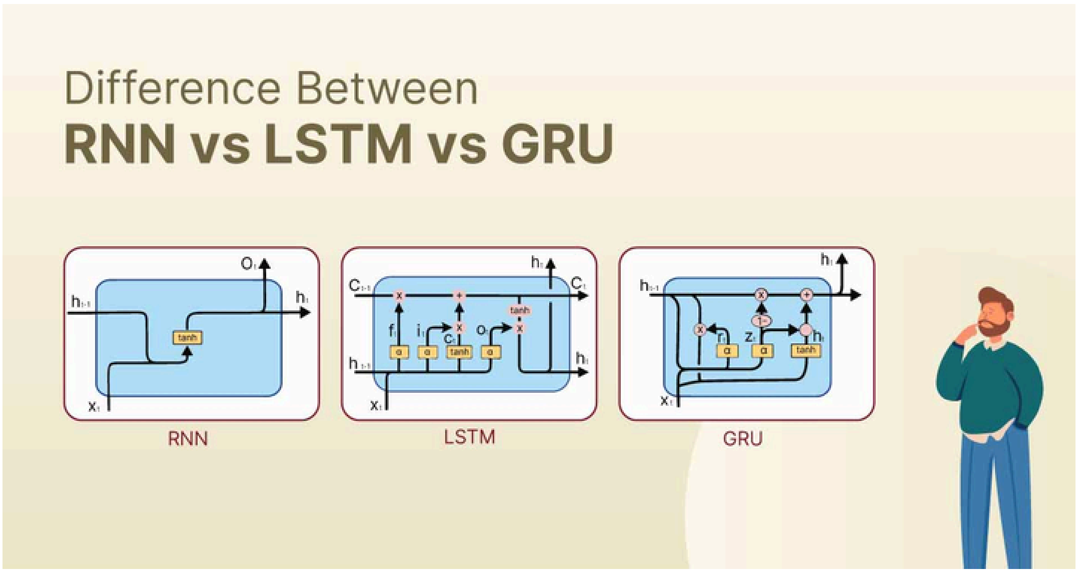
We will prepare the sequences of tokenized sentences and labels for model training and evaluation. and nsures that all sequences have the same length, which is necessary for feeding them into neural networks using **pad_sequences** function from Keras



```python
# Padding
maxlen = max(len(s) for s in sequences)
X_train = pad_sequences(sequences, maxlen=maxlen, padding='post')
X_test = pad_sequences(test_sequences, maxlen=maxlen, padding='post')
X_valid = pad_sequences(valid_sequences, maxlen=maxlen, padding='post')
y_train = pad_sequences(label_sequences, maxlen=maxlen, padding='post')
y_test = pad_sequences(test_label_sequences, maxlen=maxlen, padding='post')
y_valid = pad_sequences(valid_label_sequences, maxlen=maxlen, padding='post')
```

Models Training:

We will train 3 models of recurrent neural network (RNN) : **LSTM (Long Short-Term Memory), RNN (Vanilla Recurrent Neural Network), and GRU (Gated Recurrent Unit)**

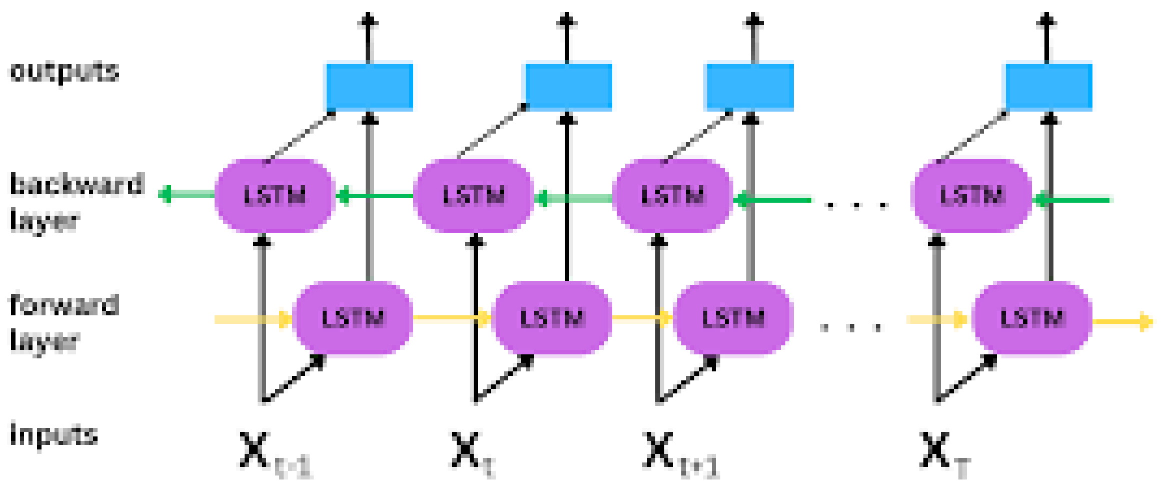


For the Three models we will use same architecutre where Our Input will pass through several stages:

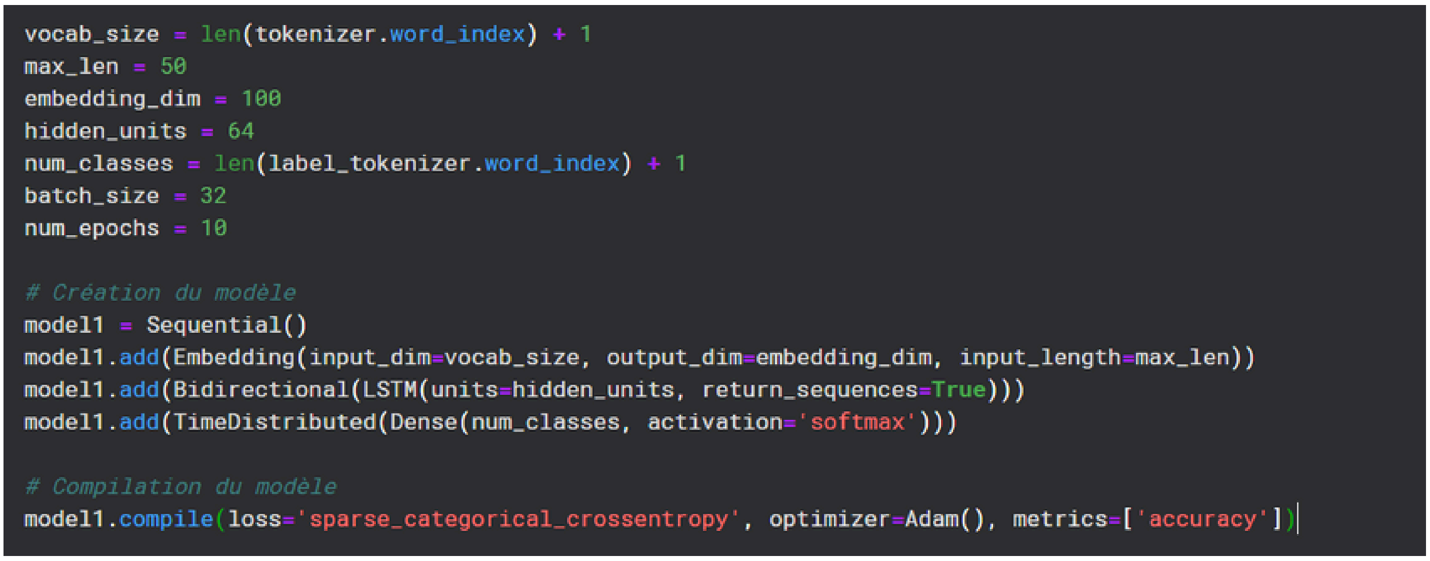
Embedding ➔ Model ➔ Dense layer ➔ Output

1.BiLSTM (Bidirectional Long Short-Term Memory):

A Bidirectional LSTM, or BiLSTM, is a type of LSTM (Long Short-Term Memory) that learns from input data from two directions instead of one, thus capturing past **(backward)** and future **(forward)** information simultaneously.

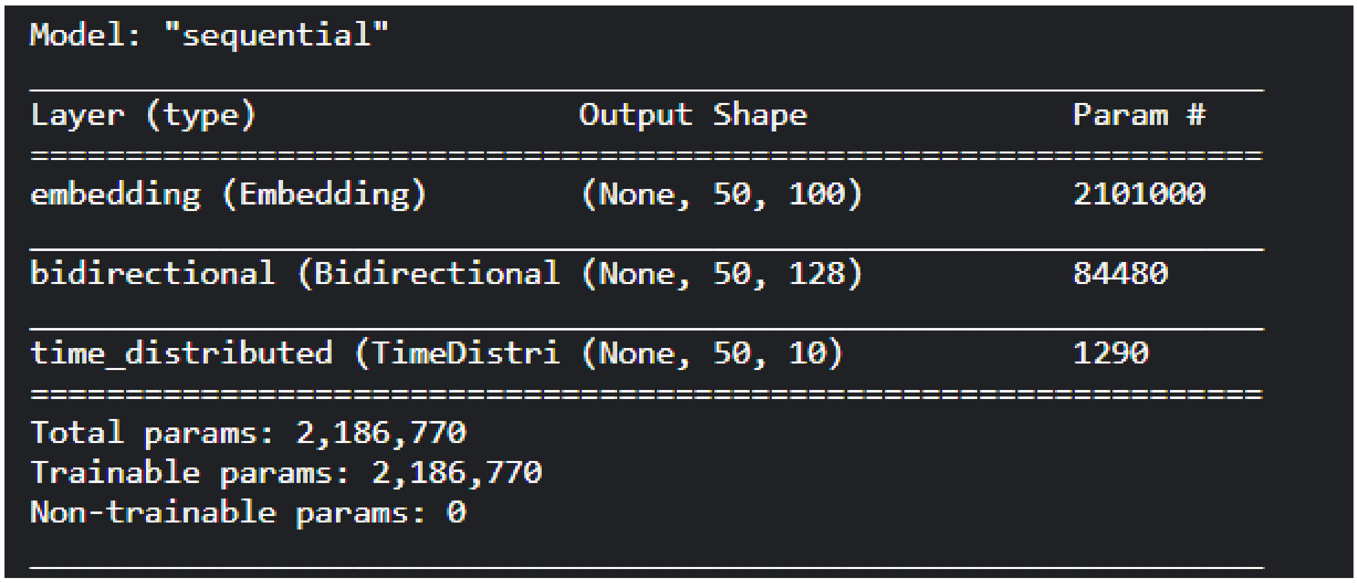


Build & Compile



```
vocab_size = len(tokenizer.word_index) + 1
max_len = 50
embedding_dim = 100
hidden_units = 64
num_classes = len(label_tokenizer.word_index) + 1
batch_size = 32
num_epochs = 10

# Création du modèle
model1 = Sequential()
model1.add(Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=max_len))
model1.add(Bidirectional(LSTM(units=hidden_units, return_sequences=True)))
model1.add(TimeDistributed(Dense(num_classes, activation='softmax')))

# Compilation du modèle
model1.compile(loss='sparse_categorical_crossentropy', optimizer=Adam(), metrics=['accuracy'])
```

Architecture

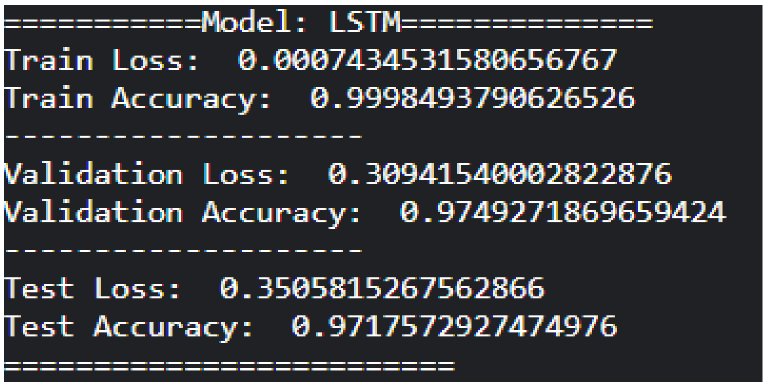


```
Model: "sequential"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
embedding (Embedding)       (None, 50, 100)           2101000

_________________________________________________________________
bidirectional (Bidirectional (None, 50, 128)          84480

_________________________________________________________________
time_distributed (TimeDistri (None, 50, 10)           1290
=================================================================
Total params: 2,186,770
Trainable params: 2,186,770
Non-trainable params: 0
_________________________________________________________________
```

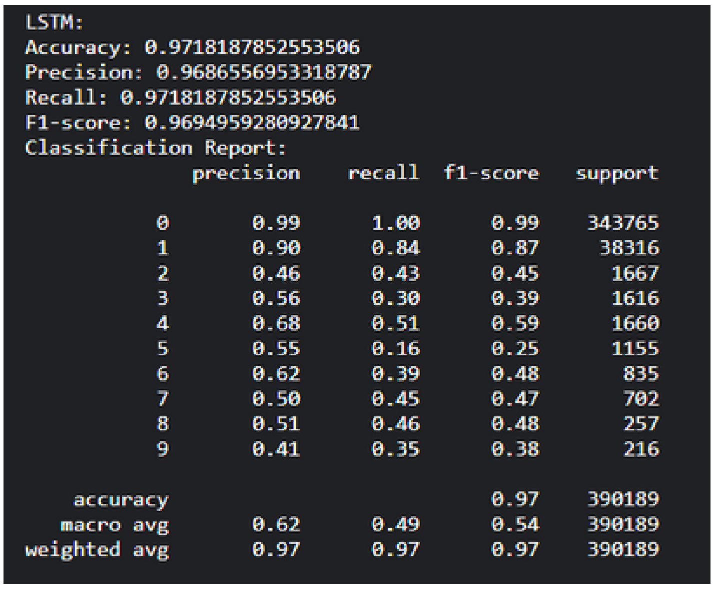
When you are dealing with sequence data, and you want to apply a Dense layer to each time step, you wrap the Dense layer in a **`TimeDistributed`** layer to apply the same Dense layer to each time step.

Training and Results

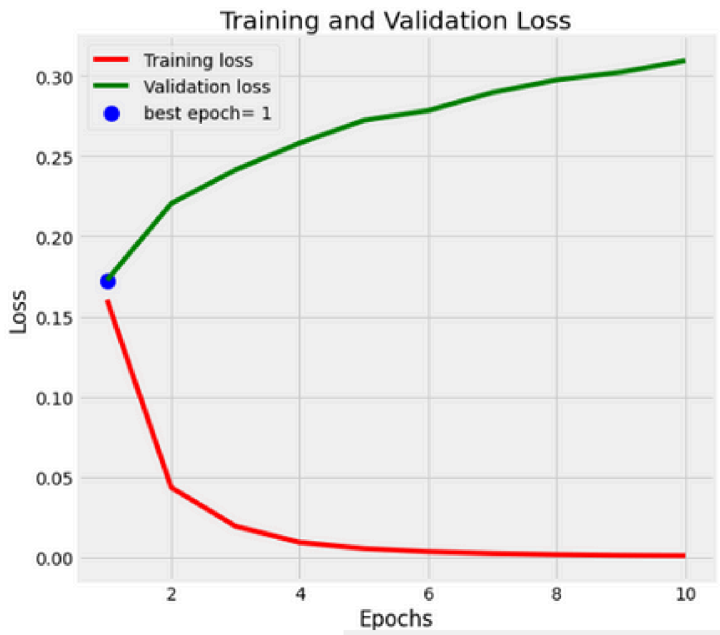
Our model got 2,186,770 trainable parameters. we trained our model for 10 epochs with a batch size of 32 , as an optimizer we used The ADAM optimizer

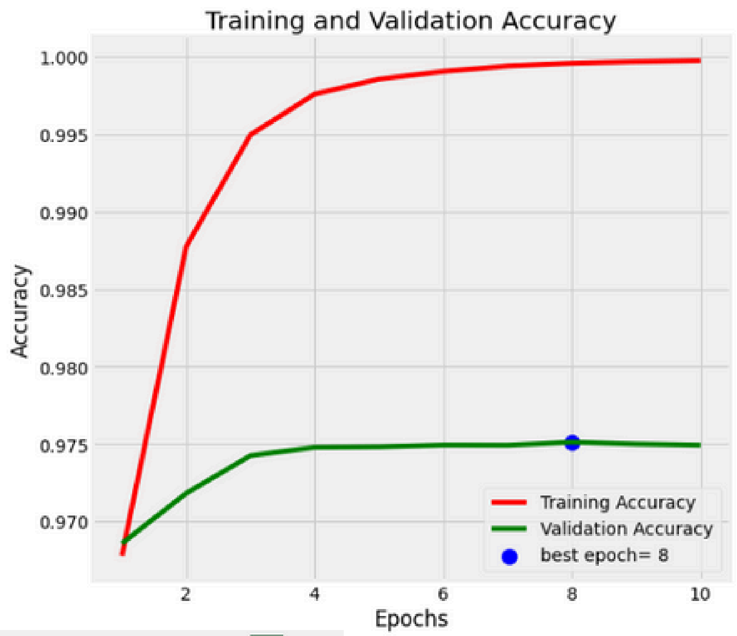


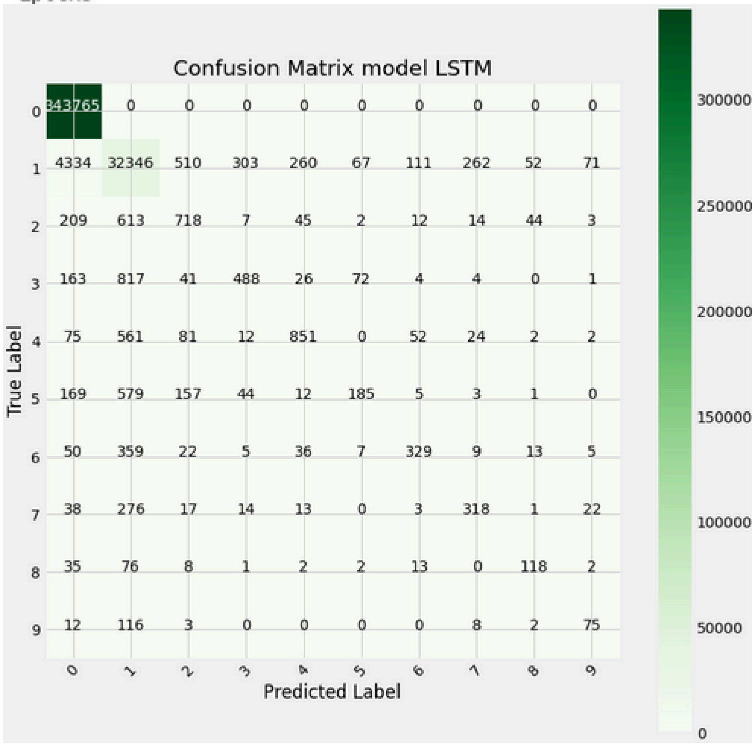
```
==========Model: LSTM=============
Train Loss:  0.0007434531580656767
Train Accuracy:  0.9998493790626526
--------------------
Validation Loss:  0.30941540002822876
Validation Accuracy:  0.9749271869659424
--------------------
Test Loss:  0.3505815267562866
Test Accuracy:  0.9717572927474976
=========================
```



```
LSTM:
Accuracy: 0.9718187852553506
Precision: 0.9686556953318787
Recall: 0.9718187852553506
F1-score: 0.9694959280927841
Classification Report:
              precision    recall  f1-score   support

           0       0.99      1.00      0.99    343765
           1       0.90      0.84      0.87     38316
           2       0.46      0.43      0.45      1667
           3       0.56      0.30      0.39      1616
           4       0.68      0.51      0.59      1660
           5       0.55      0.16      0.25      1155
           6       0.62      0.39      0.48       835
           7       0.50      0.45      0.47       702
           8       0.51      0.46      0.48       257
           9       0.41      0.35      0.38       216

    accuracy                           0.97    390189
   macro avg       0.62      0.49      0.54    390189
weighted avg       0.97      0.97      0.97    390189
```



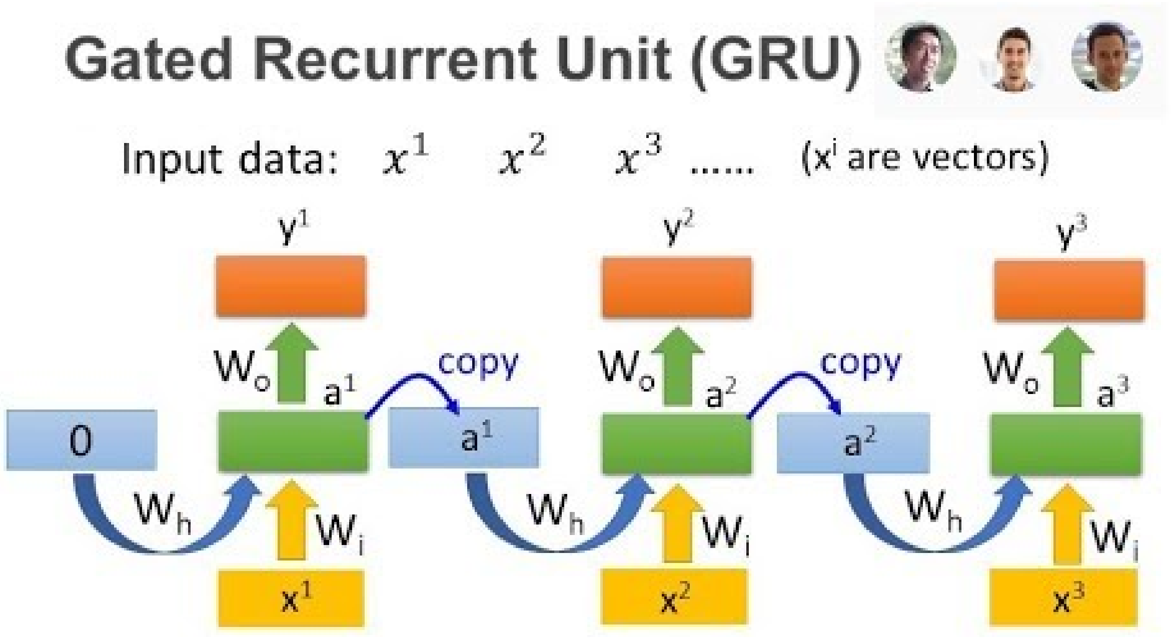




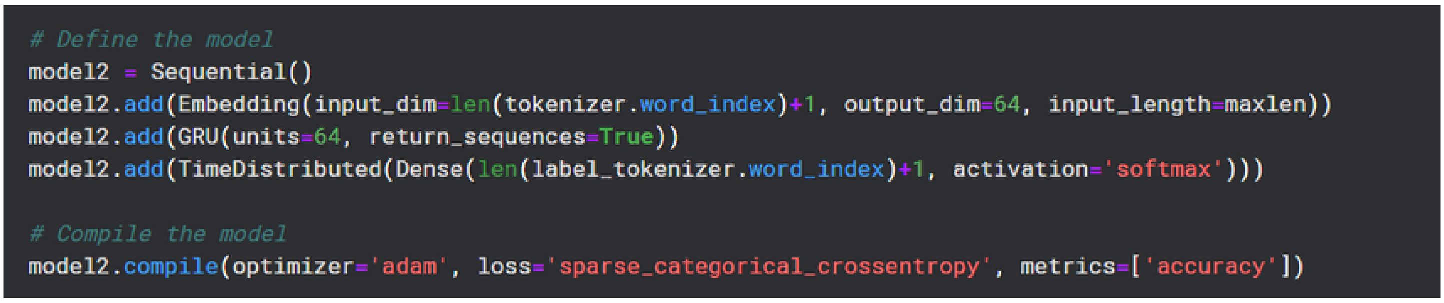
	0	1	2	3	4	5	6	7	8	9
0	343765	0	0	0	0	0	0	0	0	0
1	4334	32346	510	303	260	67	111	262	52	71
2	209	613	718	7	45	2	12	14	44	3
3	163	817	41	488	26	72	4	4	0	1
4	75	561	81	12	851	0	52	24	2	2
5	169	579	157	44	12	185	5	3	1	0
6	50	359	22	5	36	7	329	9	13	5
7	38	276	17	14	13	0	3	318	1	22
8	35	76	8	1	2	2	13	0	118	2
9	12	116	3	0	0	0	0	8	2	75

2.GRU (Gated Recurrent Unit):

GRU is a simplified version of the LSTM architecture, designed to be computationally efficient while still capturing long-range dependencies. It combines the forget and input gates into a single update gate, and merges the cell state and hidden state, resulting in fewer parameters compared to LSTM.

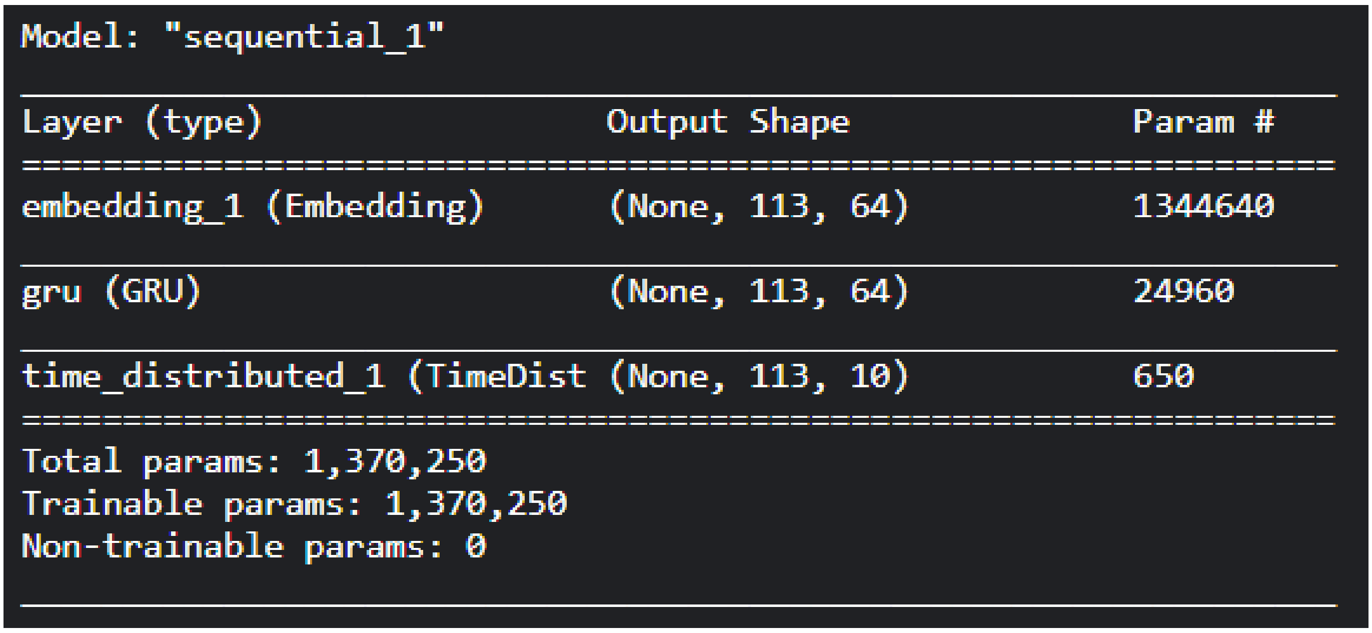

Gated Recurrent Unit (GRU)
Input data: x^1 x^2 x^3 ...... (x^i are vectors)
y^1
y^2
y^3
W_o
a^1
copy
a^2
a^3
0
W_h
W_i
x^1
x^2
x^3

Build & Compile



```
# Define the model
model2 = Sequential()
model2.add(Embedding(input_dim=len(tokenizer.word_index)+1, output_dim=64, input_length=maxlen))
model2.add(GRU(units=64, return_sequences=True))
model2.add(TimeDistributed(Dense(len(label_tokenizer.word_index)+1, activation='softmax')))

# Compile the model
model2.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

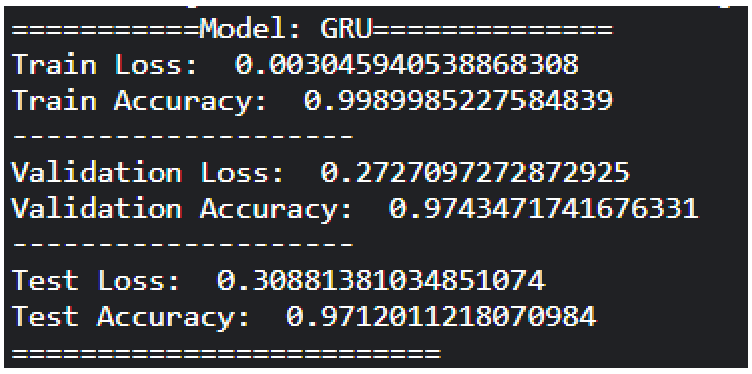
Architecture



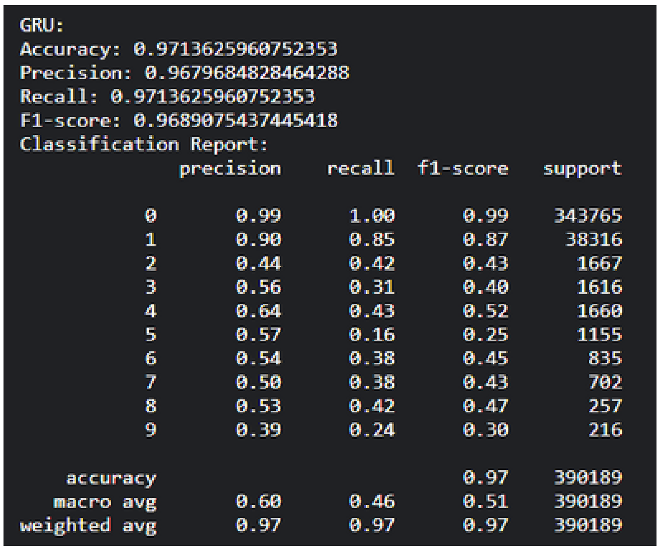
```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
embedding_1 (Embedding)     (None, 113, 64)           1344640

_________________________________________________________________
gru (GRU)                   (None, 113, 64)           24960

_________________________________________________________________
time_distributed_1 (TimeDist (None, 113, 10)          650
=================================================================
Total params: 1,370,250
Trainable params: 1,370,250
Non-trainable params: 0

_________________________________________________________________
```

Training and Results

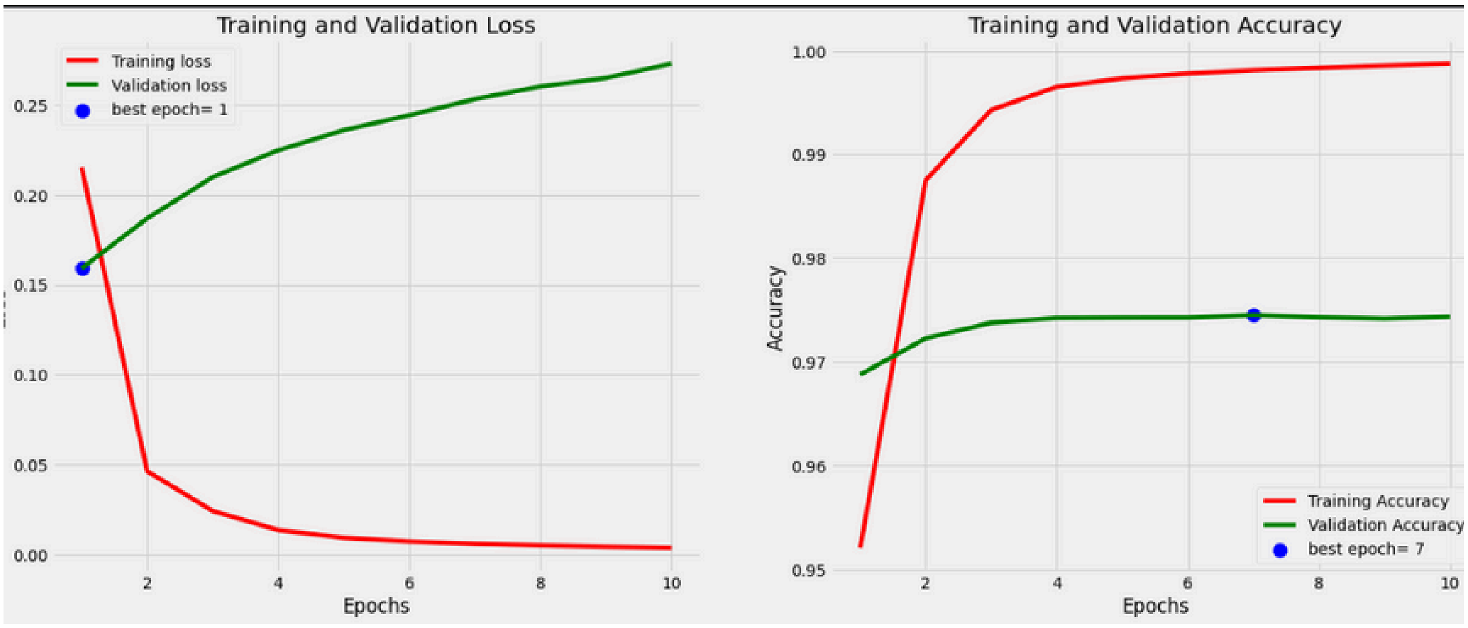
Our model got 1,370,250 trainable parameters. we trained our model for 10 epochs with a batch size of 32 , as an optimizer we used The ADAM optimizer

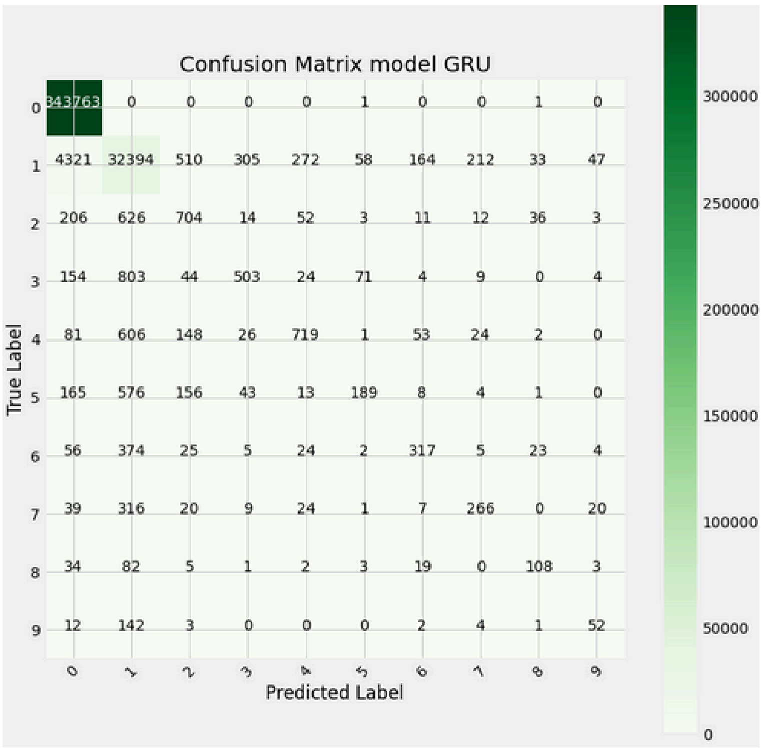


```
===========Model: GRU==============
Train Loss:  0.003045940538868308
Train Accuracy:  0.9989985227584839
--------------------
Validation Loss:  0.2727097272872925
Validation Accuracy:  0.9743471741676331
--------------------
Test Loss:  0.30881381034851074
Test Accuracy:  0.9712011218070984
=========================
```



```
GRU:
Accuracy: 0.9713625960752353
Precision: 0.9679684828464288
Recall: 0.9713625960752353
F1-score: 0.9689075437445418
Classification Report:
              precision    recall  f1-score   support

           0       0.99      1.00      0.99    343765
           1       0.90      0.85      0.87     38316
           2       0.44      0.42      0.43      1667
           3       0.56      0.31      0.40      1616
           4       0.64      0.43      0.52      1660
           5       0.57      0.16      0.25      1155
           6       0.54      0.38      0.45       835
           7       0.50      0.38      0.43       702
           8       0.53      0.42      0.47       257
           9       0.39      0.24      0.30       216

    accuracy                           0.97    390189
   macro avg       0.60      0.46      0.51    390189
weighted avg       0.97      0.97      0.97    390189
```

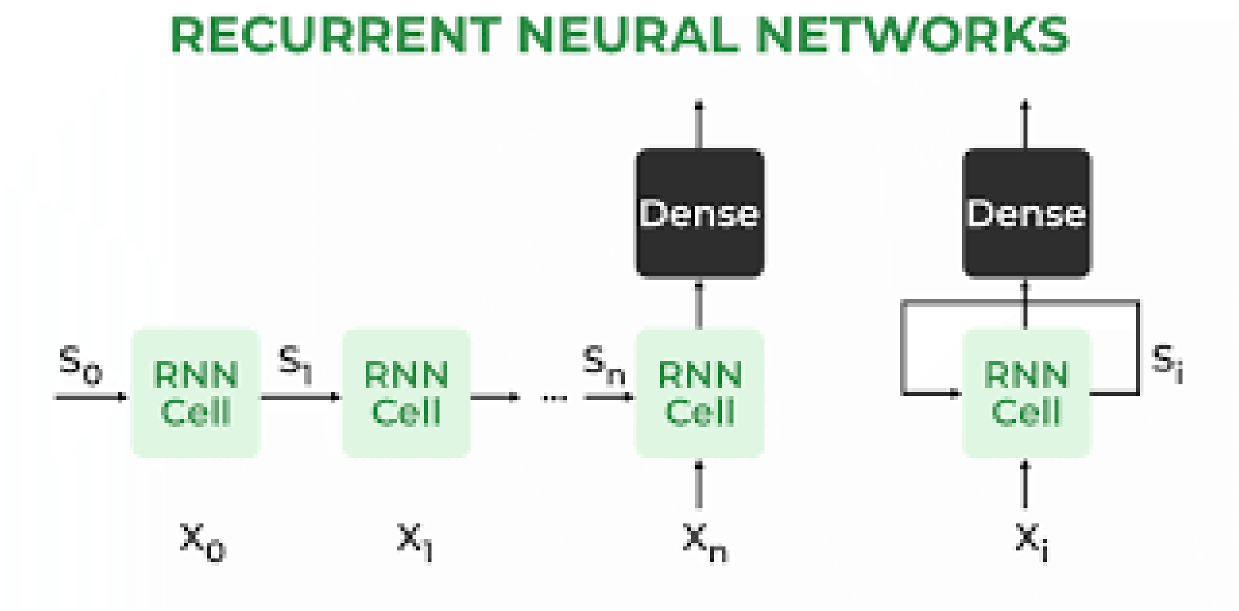




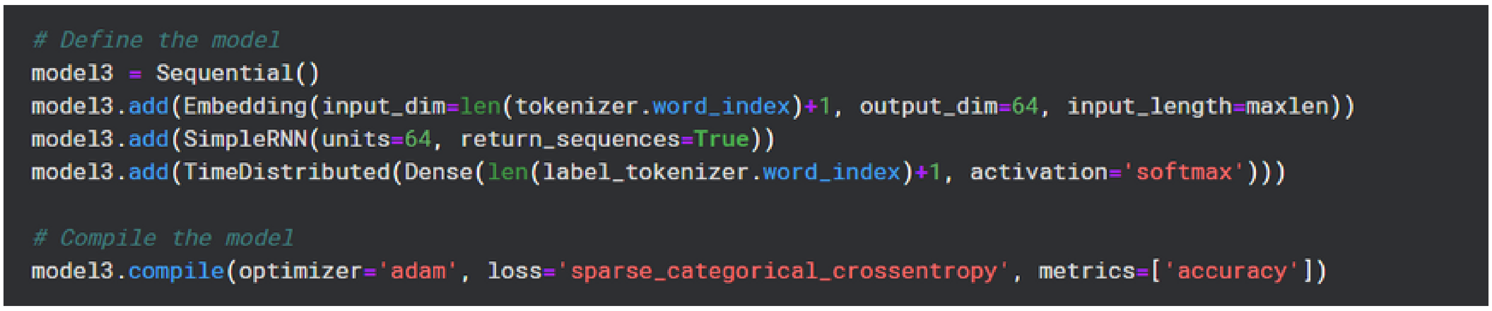
	0	1	2	3	4	5	6	7	8	9
0	343763	0	0	0	0	1	0	0	1	0
1	4321	32394	510	305	272	58	164	212	33	47
2	206	626	704	14	52	3	11	12	36	3
3	154	803	44	503	24	71	4	9	0	4
4	81	606	148	26	719	1	53	24	2	0
5	165	576	156	43	13	189	8	4	1	0
6	56	374	25	5	24	2	317	5	23	4
7	39	316	20	9	24	1	7	266	0	20
8	34	82	5	1	2	3	19	0	108	3
9	12	142	3	0	0	0	2	4	1	52

3. RNN (Vanilla Recurrent Neural Network):

RNN is the simplest form of recurrent neural network, where each neuron has a self-connected recurrent connection. It processes sequential data by iteratively updating its hidden state using the current input and previous hidden state.

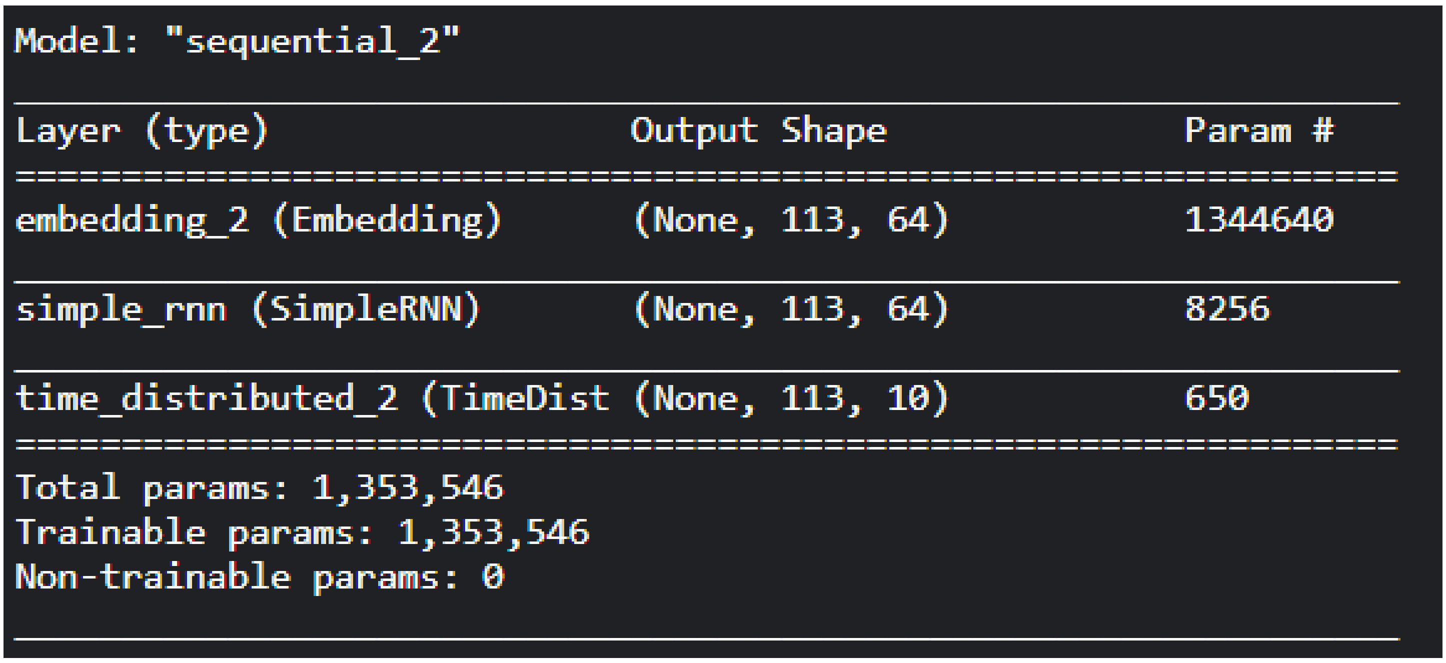


Build & Compile



```
# Define the model
model3 = Sequential()
model3.add(Embedding(input_dim=len(tokenizer.word_index)+1, output_dim=64, input_length=maxlen))
model3.add(SimpleRNN(units=64, return_sequences=True))
model3.add(TimeDistributed(Dense(len(label_tokenizer.word_index)+1, activation='softmax')))

# Compile the model
model3.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

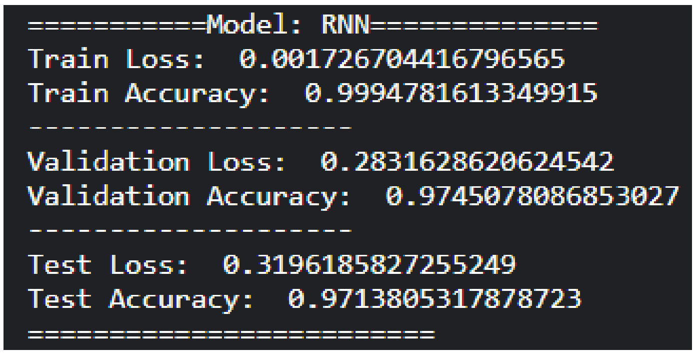
Architecture



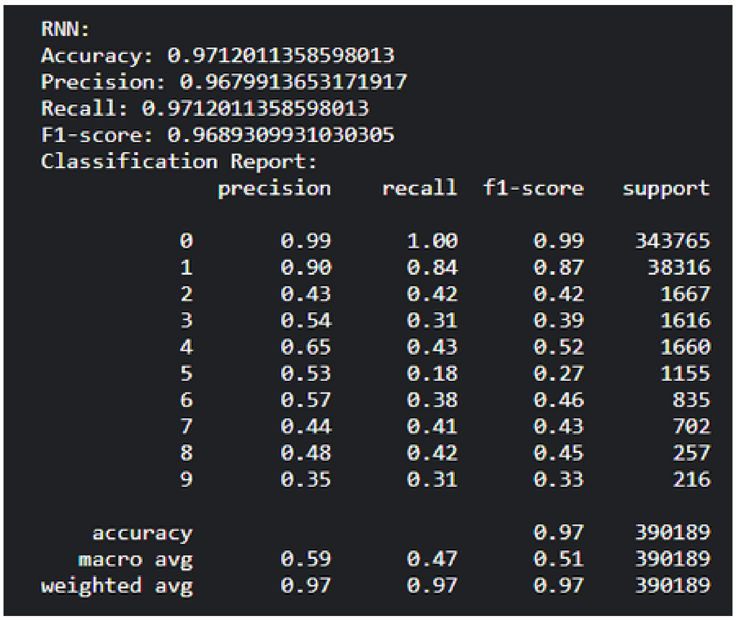
```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 113, 64)           1344640
_________________________________________________________________
simple_rnn (SimpleRNN)       (None, 113, 64)           8256
_________________________________________________________________
time_distributed_2 (TimeDist (None, 113, 10)           650
=================================================================
Total params: 1,353,546
Trainable params: 1,353,546
Non-trainable params: 0
_________________________________________________________________
```

Training and Results

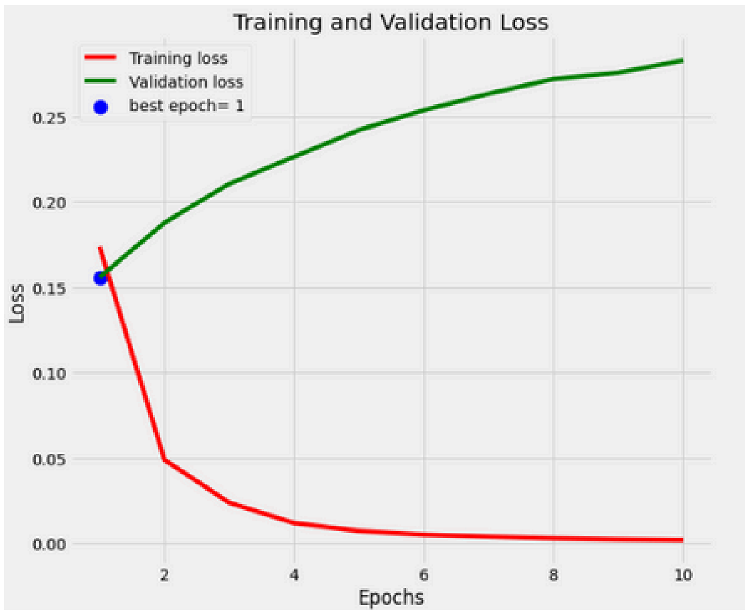
Our model got 1,353,546 trainable parameters. we trained our model for 10 epochs with a batch size of 32 , as an optimizer we used The ADAM optimizer

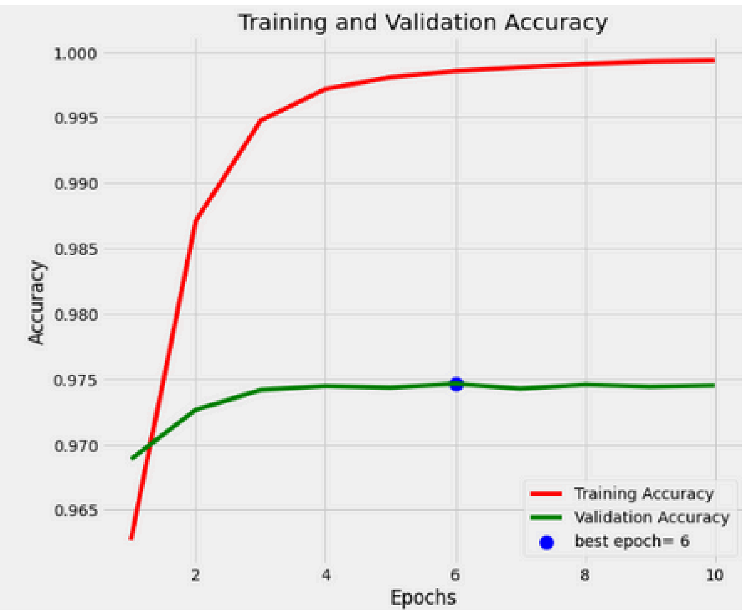


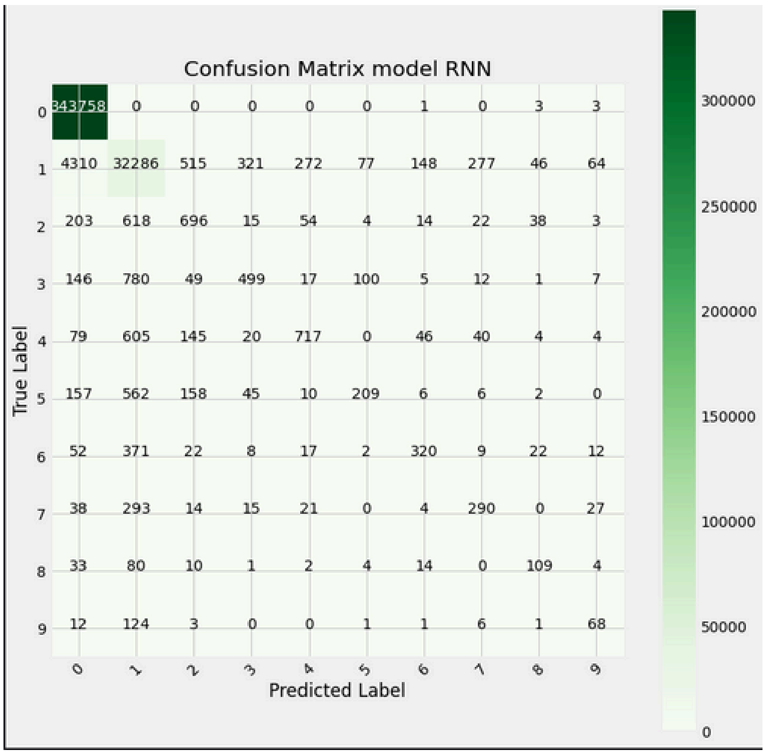
```
==========Model: RNN=============
Train Loss:  0.001726704416796565
Train Accuracy:  0.9994781613349915
--------------------
Validation Loss:  0.2831628620624542
Validation Accuracy:  0.9745078086853027
--------------------
Test Loss:  0.3196185827255249
Test Accuracy:  0.9713805317878723
========================
```



```
RNN:
Accuracy: 0.9712011358598013
Precision: 0.9679913653171917
Recall: 0.9712011358598013
F1-score: 0.9689309931030305
Classification Report:
              precision    recall  f1-score   support

           0       0.99      1.00      0.99    343765
           1       0.90      0.84      0.87     38316
           2       0.43      0.42      0.42      1667
           3       0.54      0.31      0.39      1616
           4       0.65      0.43      0.52      1660
           5       0.53      0.18      0.27      1155
           6       0.57      0.38      0.46       835
           7       0.44      0.41      0.43       702
           8       0.48      0.42      0.45       257
           9       0.35      0.31      0.33       216

    accuracy                           0.97    390189
   macro avg       0.59      0.47      0.51    390189
weighted avg       0.97      0.97      0.97    390189
```







True Label \ Predicted Label	0	1	2	3	4	5	6	7	8	9
0	343758	0	0	0	0	0	1	0	3	3
1	4310	32286	515	321	272	77	148	277	46	64
2	203	618	696	15	54	4	14	22	38	3
3	146	780	49	499	17	100	5	12	1	7
4	79	605	145	20	717	0	46	40	4	4
5	157	562	158	45	10	209	6	6	2	0
6	52	371	22	8	17	2	320	9	22	12
7	38	293	14	15	21	0	4	290	0	27
8	33	80	10	1	2	4	14	0	109	4
9	12	124	3	0	0	1	1	6	1	68

4. **Conclusion:**

- Each of these recurrent neural network architectures—LSTM, RNN, GRU, and BiLSTM—can achieve the desired results for sequential data tasks, but they may differ in terms of convergence speed and efficiency. as we can see LSTM converge at 8th epoch , GRU converge at 7th epoch and RNN at 6th epoch.

- while LSTM, RNN, GRU, and BiLSTM are all recurrent neural network architectures designed for sequential data tasks, they have distinct differences in terms of architecture, parameterization, and capability to capture long-term dependencies and bidirectional context. The choice of model depends on the specific requirements of the task at hand, computational resources available, and desired trade-offs between model complexity and performance.