8) In 'prediction.ipynb':
→ Load the ANN trained model, scaler, pickle, onehot
→ Take a sample i/p → convert it into suitable format i.e. OHE few columns & scale it.
→ Predict on the i/p.

9) Write 'app.py' → deploy on streamlit

10) Practice by predicting a value now i.e. practice regression prob. stmt. on this same dataset (Similar to prev.)

11) Determining the optimal number of hidden layers & neurons for an ANN:
↳ Guidelines:
→ Start Simple: Begin with a simple architecture & gradually increase complexity if needed.
→ Grid Search/Random Search: Use grid search / random search to try different architectures
→ Cross-Validation: Use cross-validation to evaluate the performance of different architectures.
→ Heuristics & Rules of thumb: → A common practice is to start with 1-2 layers.
↘ The no. of neurons in the hidden layer should be between the size of the i/p layer & the size of o/p layer
→ Refer 'hyperparametertuning.ipynb'.

// Refer files for above project.

NLP IN DEEP LEARNING

Simple RNN → LSTM/GRU RNN → Bidirectional RNN → Encoder Decoder
↓
Transformers ← Self Attention

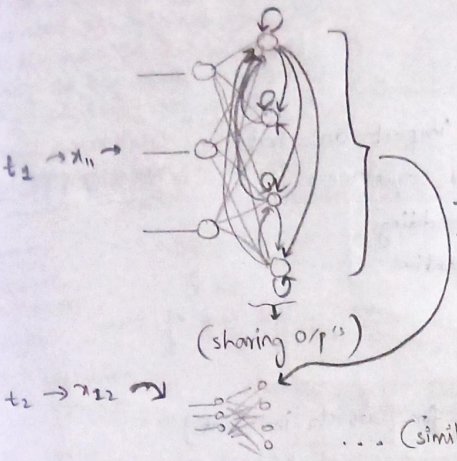
SIMPLE RNN INDEPTH INTUITION

- Limitations of ANN:
• ANN processes inputs simultaneously, ignoring word order.
• Inputting all words at once hinders context preservation.

- Intro. to RNN (Recurrent Neural Networks):
• Words are passed 1 at a time. (ex: "The" at t=1, "food" at t=2)
• Feedback loops: Hidden neurons share outputs with themselves & other neurons maintaining context.

- Each timestamp retains info. from previous words, preserving the sequential nature of text

- RNN Architecture:



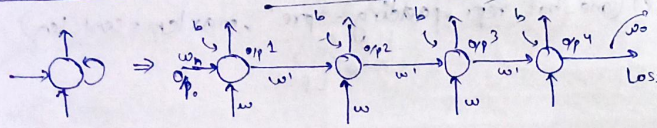
these hidden neurons will share their values at next time stamp.

... (similarly with $t_3, t_4 \ldots$)

eg: The food is good
The food is bad
The food is not good

- I/p prep.: → Preprocess the text by removing stop words (ex. "the food is good" → "food good") & tokenize it into words.
→ Create a vocab. of unique words & now convert it into no.'s (let → OHE) → : food = [1,0,0,0,0]
good = [0, 1, 0,0,0]

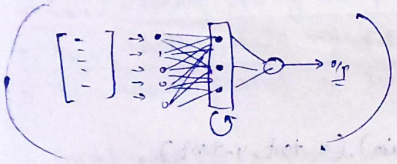
• FORWARD PROPAGATION:



(activation func.)

$o/p\,1 = f(x_{11} \cdot w + b_1)$
$o/p\,2 = f(x_{12} \cdot w + o_1 \cdot w' + b_1)$
$o/p\,3 = f(x_{13} \cdot w + o_2 \cdot w' + b_1)$
$\hat{y} = \sigma(o_3 * w_0)$

	x_{11}	x_{12}	x_{13}	x_{14}
t =	1	2	3	4
	(The)	(food)	(is)	(good)



• BACKWARD PROPAGATION:

(slope of grad. descent)

Update $[w_i, \ldots w_n, w_o]$; Weight updation formula $\Rightarrow w_{new} = w_{old} - \eta\left(\frac{\partial h}{\partial w_{old}}\right)$

1) Update w_o 2) Update w_h (Hidden Layer Weights) 3) Update Weight w_i (timestamps)

- Problems with RNN:

- Long term dependency cannot be captured by RNN (Vanishing Gradient Prob.)

ex: Sentence length = 50 words.

$$\therefore \frac{\partial h}{\partial w_{old}} = \left[\frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial s_{50}} \cdot \frac{\partial s_{50}}{\partial s_{11}} \cdots \frac{\partial o/p_2}{\partial w_{hold}}\right] + \left[\cdots\right] + \cdots + t_{50}\left[\frac{\partial h}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial o/p_{50}} \cdot \frac{\partial o/p_{50}}{\partial w_{hold}}\right]$$

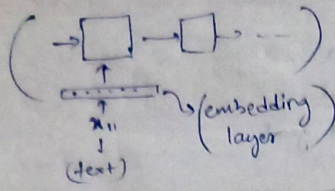
$(t=1) \rightsquigarrow \approx 0$; $t=2$

(∵ $t=1$ → word not participating in updating the weights)

- Embedding Layers = responsible for converting words into vectors.
↳ uses "Word Embeddings"

*Refer Word2Vec in "NLP FOR MACHINE LEARNING" CHAPTER.

E2E DEEP LEARNING PROJECT WITH SIMPLE RNN



① Embedding example

We need all sentences to be of same length to train our RNN model because all the sentences that will be going it will go for fixed no. of times bcoz of sentence length

① ⇒ from tensorflow.keras.preprocessing.text import one_hot
" " .utils import pad_sequences
" " .layers " Embedding
" " .models " Sequential

sent = ['...', '...', ...] → sentences ex: "the cup of tea"
voc_size = 1000 → vocab. size
one_hot_repr = [one_hot(words, voc_size) for words in sent]
↳ ex: [6186, 7872, 5337, 7113]
sent_len = 8
embedded_docs = pad_sequences(one_hot_repr, padding = "pre", maxlen = sent_len)
↳ print( )
2 methods { pre / post
dim = 10 → Feature representation
model = Sequential()
model.add(Embedding(voc_size, dim, input_length = sent_len))
model.compile('adam', 'mse')
model.summary()
model.predict(embedded_docs)

② Ex:
- importing
- loading IMDB dataset into (x_train, y_train), (x_test, y_test)
- mapping of words index back to words (for understanding)
- making sentences to equal length
- training Simple RNN

total 3 layers
↳ model = Sequential()
model.add(Embedding(max_features, 128, input_length = max_len))
1000 → (vocab. size)
(total features)
500
model.add(SimpleRNN(128, activation = 'relu'))
model.add(Dense(1, activation = 'sigmoid'))
model.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])

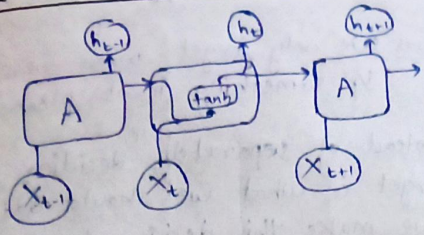
- create instance of Early stopping callback
- train the model with early stopping.
- save the model.

- Refer "app.py" → how to get movie review from saved model
  ↳ display it using streamlit.

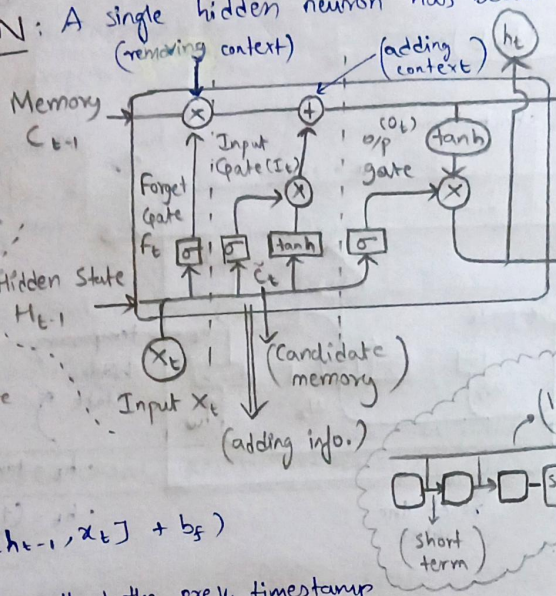
(Refer Notebook)

LSTM & GRU RNN INDEPTH INTUITION

- RNN:



- LSTM RNN: A single hidden neuron has below 3 imp. gates



$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$

removing or forgetting some info. — forgetting memory

- Neural network layer
- pointwise operation
- vector transfer
- Concatenate
- Copy

ex: v_1: [1 2 3]
v_2: [4 5 6]
⊗ ⇒ [4 10 18]
⊕ ⇒ [5 7 9]

- $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$
- C_{t-1} = memory cell of the prev. timestamp
- h_{t-1} = hidden state of previous timestamp
- x_t = word passed as i/p in the current timestamp

∴ Forget Gate (f_t) = based on the context, forget gate will let go some information or will not let go info. (forgetting)

ex: C_{t-1} = [6 8 9]

[0 0 0] [1 1 1] [0 1 0]

= [0 0 0] = [6 8 9] = [6 8 9]

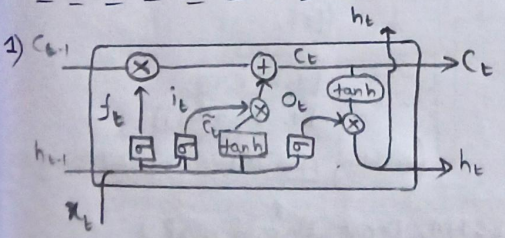
- $I_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$
- $\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$
- $O_t = \sigma(W_o [h_{t-1}, x_t] + b_o)$

$h_t = O_t * \tanh(C_t)$

- $[W_i, W_c, W_o]$ → updating ← back propagation

- Variants of LSTM RNN:

1)



$f_t = \sigma(W_f \cdot [C_{t-1}, h_{t-1}, x_t] + b_f)$

$i_t = \sigma(W_i \cdot [C_{t-1}, h_{t-1}, x_t] + b_i)$

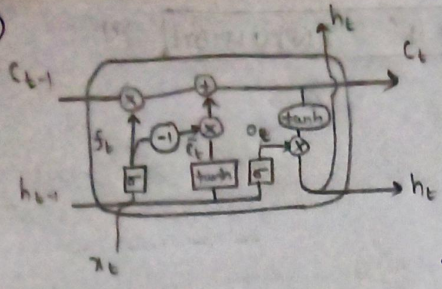
$O_t = \sigma(W_o \cdot [C_t, h_{t-1}, x_t] + b_o)$

Connections → from memory cell to (forget gate, ip gate, op gate)

Peephole connections

(we let the gate layers look at the cell state)

2)

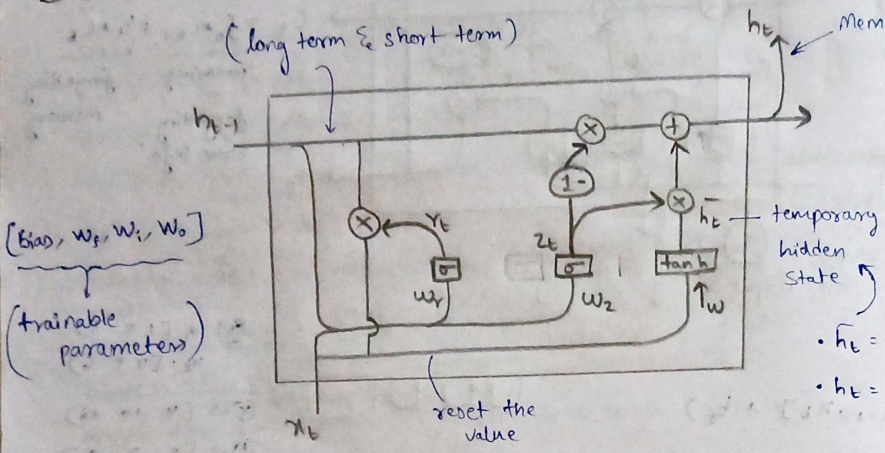


$C_t = f_t * C_{t-1} + (1-f_t) * \tilde{C}_t$

Goal: we only forget when we're going to i/p. something in its place.

{ Instead of separately deciding what to forget & what we should add new info., we make this decision together.

- GRU (Gated Recurrent Unit):



[bias, W_f, W_i, W_o]
(trainable parameters)

- $z_t = \sigma(W_z \cdot [h_{t-1}, x_t]$ (update gate)
- $r_t = \sigma(W_r \cdot [h_{t-1}, x_t]$ (reset gate)
- $\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t]$
- $h_t = (1-z_t) * h_{t-1} + z_t * \tilde{h}_t$

LSTM & GRU E2E DL PROJECT - PREDICTING NEXT WORD

Ex: (Refer Notebook)

```
# Data Collection
import nltk
data = gutenberg.raw('shakespeare-hamlet.txt')
# Import → Tokenizer, pad_sequences, train_test_split   (used to convert words into vectors)
tokenizer = Tokenizer()
tokenizer.fit_on_texts([text])   → assigns unique integer index to each word in a dataset
tokenizer.word_index
# Create i/p sequence → refer notebook
# Pad sequence
# Create predictors & labels
# Split into training & testing
# Define early stopping
# Define & train LSTM/GRU model
```

Hi Ankit you can.
1 687 4 45

↳ created in this way:

```
[[1],
 [1,687],
 [1,687,4],
 [1,687,4,45]]
```

↳ Import Sequential, Embedding, LSTM, Dense, Dropout, GRU

(disabling some neurons to avoid overfitting)

```
model = Sequential()
model.add(Embedding(total_words, 100, input_length = max_sequence_len-1))
model.add(LSTM(150, return_sequences = True))
model.add(Dropout(0.2))
model.add(LSTM(100))
model.add(Dense(total_words, activation = "softmax"))
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=['accuracy'])
```

```
model.summary()
history = model.fit(x_train, y_train, epochs=50, validation_data=(x_test, y_test),
                    verbose=1, callbacks=[early_stopping])
```

Replace 'LSTM' with 'GRU' to use 'GRU'

Func. to predict next word

Save the model & tokenizer

```
model.save('next_word_lstm.h5')
import pickle
pickle.open('tokenizer.pickle', 'wb') as handle:
    pickle.dump(tokenizer, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

(ensures that we use the most eff. serialization format available)

Custom text

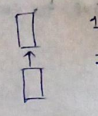
```
input_text =
max_sequence_len = model.input_shape[1]
next_word =
print(f"Next Word Prediction : {next_word}")
```

Deploy on streamlit

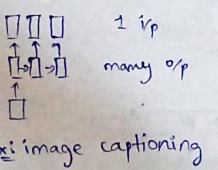
BIDIRECTIONAL RNN ARCHI. & INDEPTH INTUITION

Types of RNN:

1) One to One RNN:

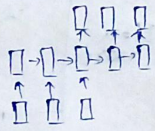


2) One to Many RNN:



ex: image captioning

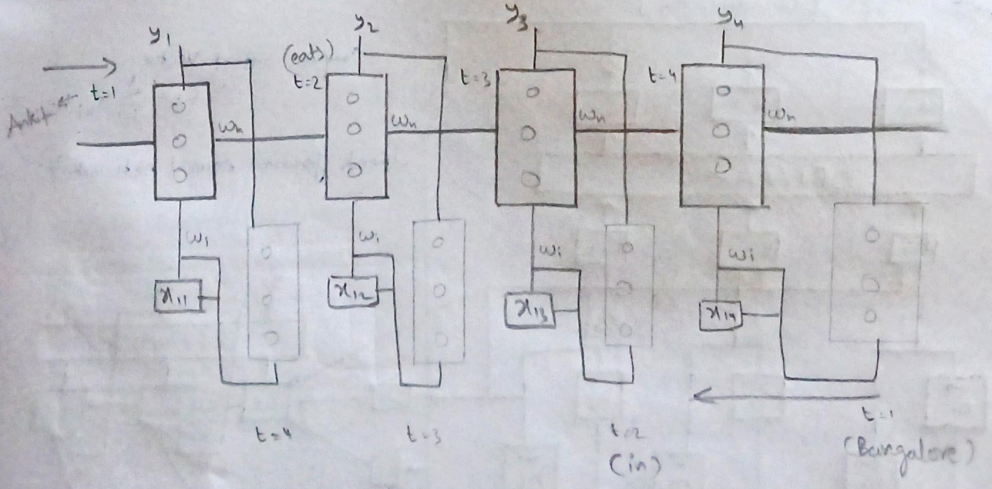
3) Many to Many RNN:



4) Many to One RNN:

Bidirectional RNN:

ex: Ankit eats ___ in Bangalore
x_{11} x_{12} x_{13} x_{14} x_{15}



Now only fw prop. couldn't predict 'x_{13}' accurately, so we run another RNN backwards, so now the model can accurately predict 'x_{13}' since it has context of its upcoming sentences as well.

ENCODER DECODER + ATTENTION MECHANISM - SEQUENCE 2 SEQUENCE

ex: One language to other!

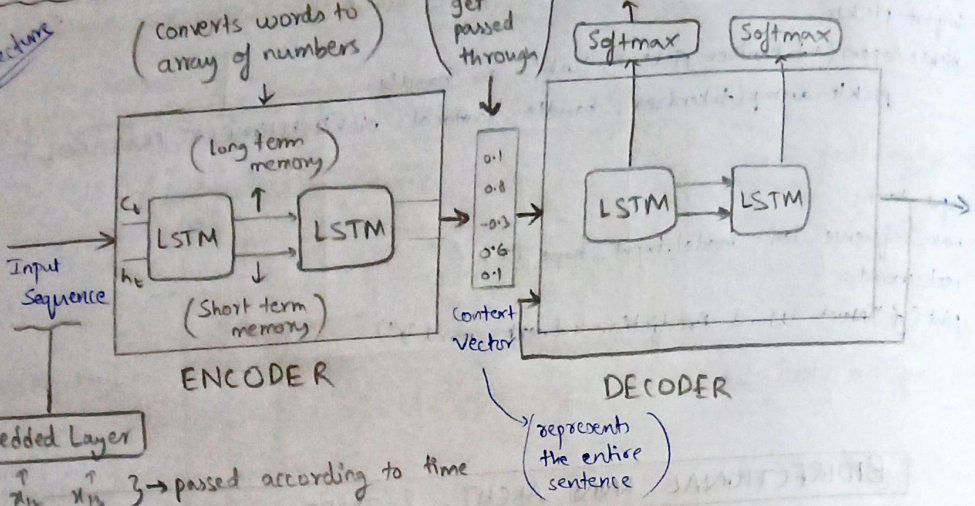
Encoder - Decoder:

Architecture

(converts words to array of numbers)

Hidden states get passed through

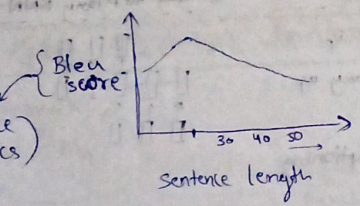
The decoder generates the o/p word-by-word & keeps feeding the prev. word into decoder again.



RNN → had vanishing gradient prob, so we have to move to encoder decoder

Drawbacks

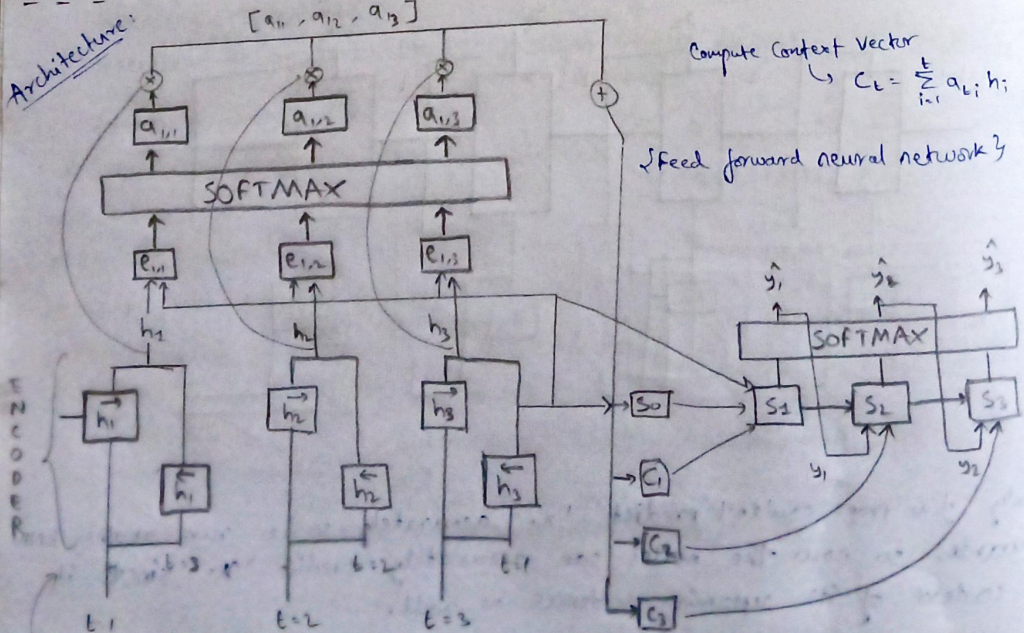
(performance metrics)



Attention Mechanism: → To resolve above drawback, we use this!

Refer video for detailed explanation

Architecture:



Compute Context Vector

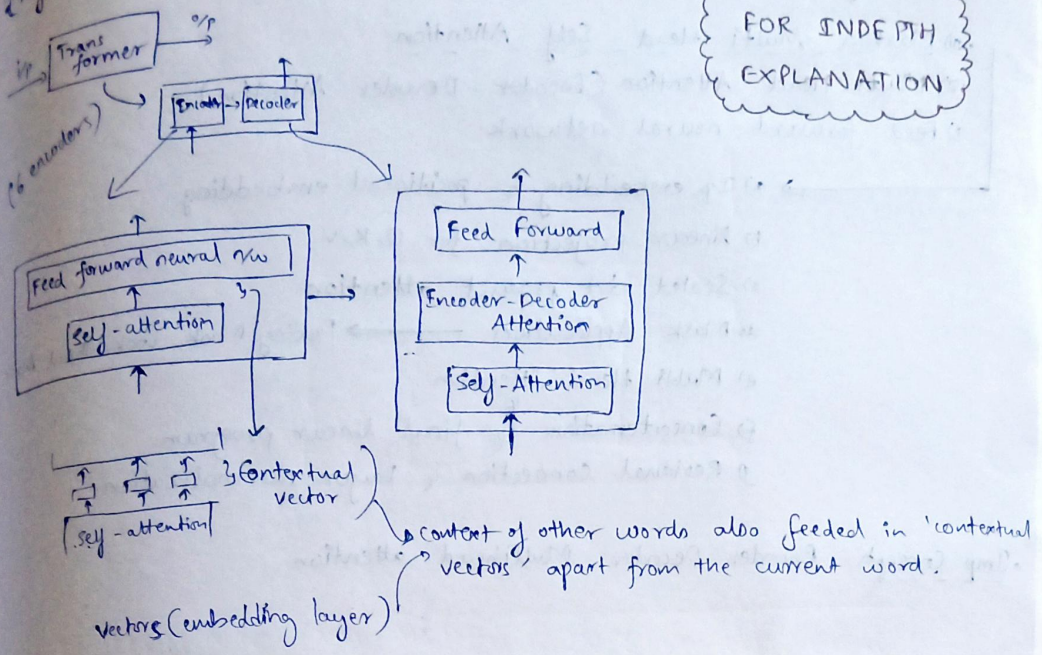
↳ $C_t = \sum_{i=1}^{t} a_{t,i} h_i$

{Feed forward neural network}

TRANSFORMERS

[Refer Architecture Diagram in Pdf]

REFER PDF FOR INDEPTH EXPLANATION



Self-attention: aka scaled dot-product attention, is a crucial mechanism in transformer archi. that allows the model to weigh the importance of different tokens in the i/p sequence relative to each other.

- **Inputs** = Queries, Keys, Values

→ **Query (Q) Vector:** represent the token for which we are calculating the attention

→ **Key (K) Vector:** represent all the tokens in the sequence & are used to compare with the query vectors to calculate attention scores.

→ **Value (V) Vector:** holds the actual info. that will be aggregated to form the o/p of the attention mechanism.

Steps:
1) Token Embedding
2) Linear Transformation
3) Compute Attention Scores
4) Scaling
5) Apply Softmax
6) Weight sum of values

Imp. Terms:
- Self Attention with multi heads
  - ↳ It expands model's ability to focus on different position of token
- Positional Encoding → (representing order of sequence)
  - Sinusoidal PE
  - Learned PE
- Layer Normalization & Batch Norm.
- Residual Connection

- The transformer decoder is responsible for generating the o/p sequence 1 token at a time, using the encoder's o/p & the previously generated tokens.

1) Masked_Multi_Head_Self_Attention
2) Multi Head Attention (Encoder - Decoder Attention)
3) Feed forward neural network

→ a) I/p embedding & positional embedding
b) linear projection for Q, K, V
c) Scaled dot product attention
d) Mask Application ⟶ Padding Mask, Look Ahead Mask
e) Multi Head Attention
f) Concatenation & final linear program
g) Residual Connection & Layer Normalization

- Imp Concepts: Encoder Decoder Multihead attention

E2E PROJ. (MLOPS) WITH ETL PIPELINES - BUILDING A NETWORK SECURITY SYSTEM

// Refer Obsidian !