Lec-2(Recurrent Neural Networks, Transformers, and Attention)

Sequence Modelling

There are many examples of sequence data like voice, DNA encoding, stock market, etc. Their applications are majorly of 4 types as below:

1. Many to one - like sentiment classification where we take in a sequence of words(sentence) and try to predict the sentiment
2. One to many - like image captioning where a single image is given and we try to generate a sequence of output(caption)
3. Many to many - like translate speech/text between 2 diff language where we take in a sequence and output a sequence

Neurons with Recurrence

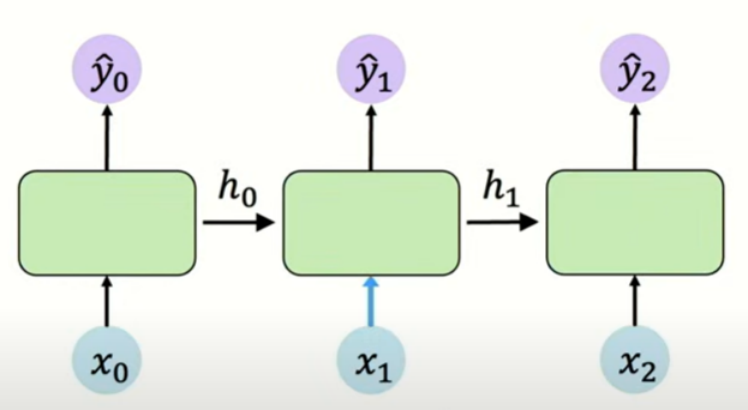
simple perceptron → layer of perceptrons

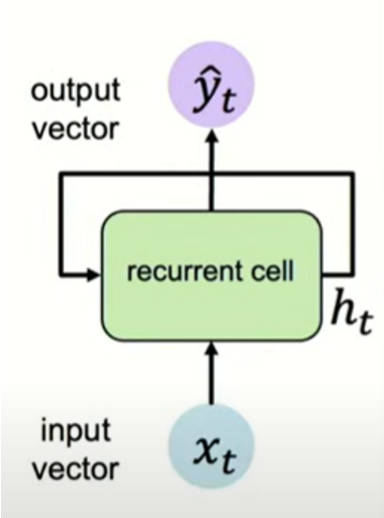
how does layer of neural network consider sequential data??

→ We can consider the NN tries to map the i/p and o/p at that instant of time (t)

$$\hat{y}_t = f(x_t,\ h_{t-1})$$

(y is output, x is input at that instant and h is the previous state from memory)





Recurrent Neural Networks

The notion of recurrence shown above is the core idea behind architecture of RNN.

RNNs are one of the foundational frameworks of **Sequence modelling** problems.

RNNs have a state h_t that is updated at each time step as a sequence is processed.

$$h_t = f_W(x_t,\ h_{t-1})$$

(h_t is cell state, f_W is function with weights W, x_t is input and h_{t-1} is old state)

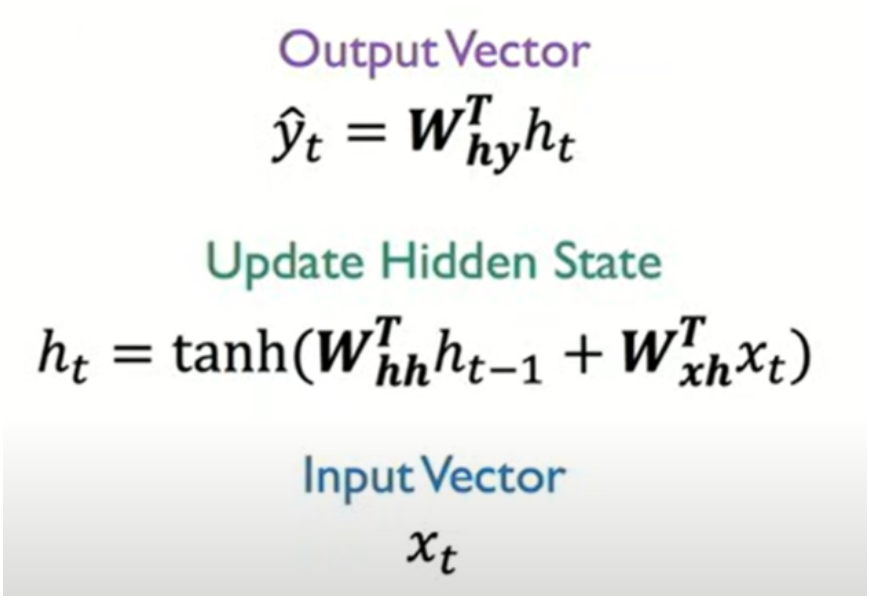
Above equation is called **"Recurrent Relation"**.

```
my_rnn = RNN()
hidden_state = [0,0,0,0]

sentence = ["I", "love", "recurrent", "neural"]

for word in sentence:
    prediction, hidden_state = my_rnn(word, hidden_state)
```

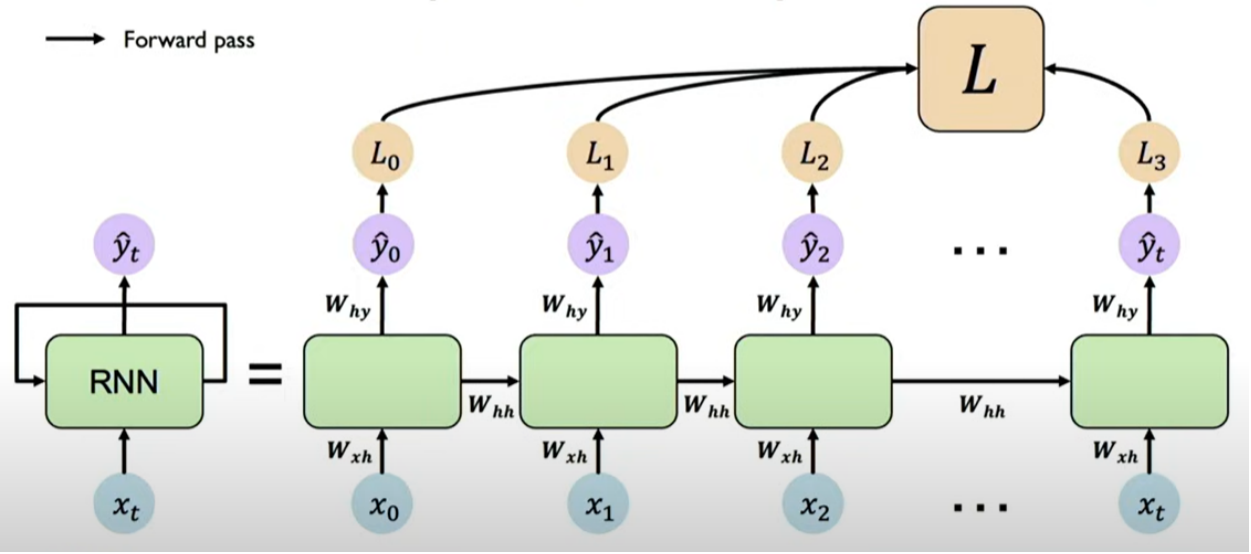
```
next_word_prediction = prediction
# >>> "networks!"
```



RNNs: Computational Graph Across Time

The same weight matrix is used from the input to the hidden layer transformation and from the hidden layer to output transformation that is effectively reused and re-updated.

To learn about the weights of RNN, we have to compute loss and learn the technique of back-propagation to understand how to adjust our weights based on how we've computed loss




```
class MyRNNCell(tf.keras.layers.Layer):
    def __init__(self, rnn_units, input_dim, output_dim):
        super(myRNNCell, self).__init__()

        #Initialize weight matrix
        self.W_xh = self.add_weight([rnn_units, input_dim])
        self.W_hh = self.add_weight([rnn_units, rnn_units])
        self.W_hy = self.add_weight([output_dim, rnn_units])

        #Initialize hidden state to zeros
        self.h = tf.zeros([rnn_units, 1])

    def call(self, x):
        #update the hidden state
        self.h = tf.math.tanh(self.W_hh * self.h + self.W_xh * x

        #compute the output
        output = self.W_hy * self.h

        #return the current output and hidden state
        return output, self.h
```

Sequence Modelling: Design Criteria

To model sequences, we need to

1. Handle **variable-length** sequences.
2. Track **long term** dependencies - we need to maintain a sense of memory
3. Maintain the **order** of information
4. **Share parameters** across the sequence

RNNs meet the design criteria of sequence modelling

Encoding Language for Neural Network

NN don't understand any other language apart from math. They require numerical inputs i.e. encode language in a way to make the NN to operate on it numerically.

Embedding - transform indexes into a vector of fixed size

One-hot encoding - only one binary representation which maps with that word. very simplistic

Learned embedding - more fancy, the words which are related to each other in language should numerically be similar and close to each other in space. Words which are unrelated should numerically be dissimilar and far away from each other.

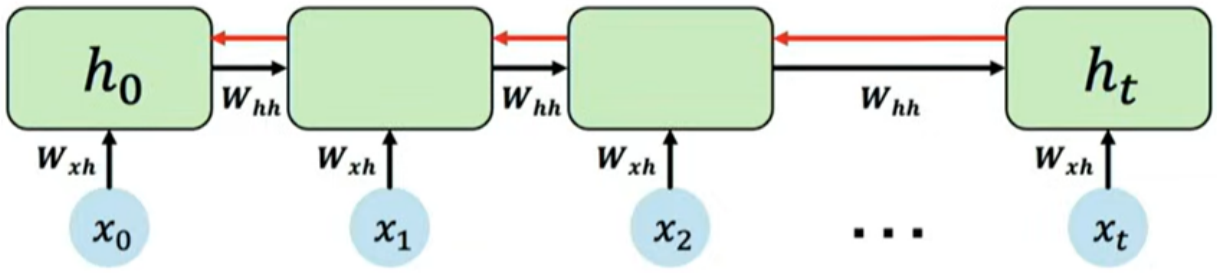
Backpropagation Through Time (BPTT)

BP Algorithm

💡 1. Take the derivative (gradient) of the loss with respect to each parameter.
2. Shift parameters in order to minimize the loss.

But for RNNs, we have to backpropagate per the time stamp and across all the time stamps from the end all the way to beginning of sequence. - BPTT

Computing gradients wrt. to h_0 involves **many factors of W_{hh} + repeated gradient computation**



⇒ **Standard RNN Gradient Flow: Exploding Gradients**

Many value > 1: exploding gradients

Gradient clipping to scale big gradients

⇒ **Standard RNN Gradient Flow: Vanishing Gradients**

Many values < 1: vanishing gradients

Strategies to mitigate -

a. Activation Function

b. Weight initialization

c. Network architecture - **Gated cells**

Problem of Long Term dependency

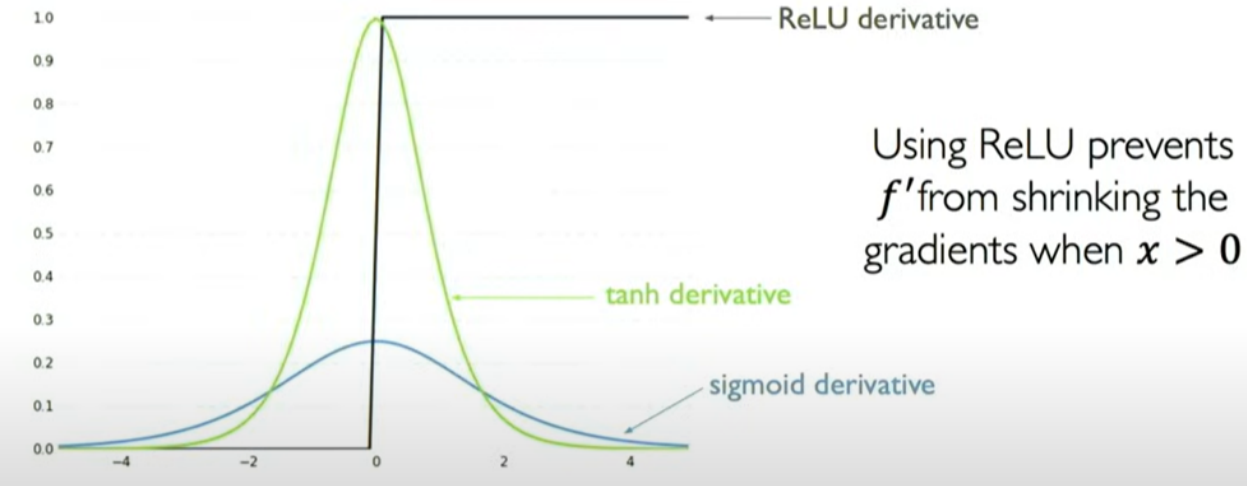
Vanishing gradients can be a problem in long term dependency, because as we move back on each time frame, the gradients become smaller and smaller thereby

reducing/destroying the network's capacity to model the dependency.

Multiply many smaller numbers together → Errors due to further back time steps have smaller and smaller gradients → Bias parameters to capture short-term dependencies

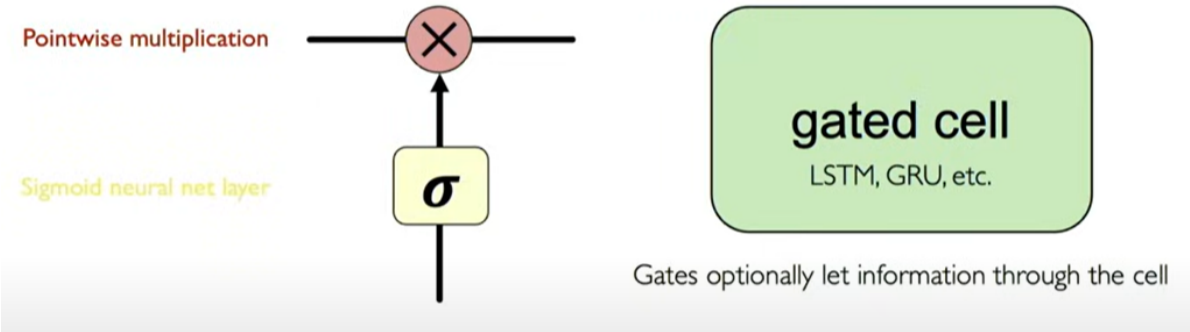
Tricks to mitigate above problem

1. Activation function -



2. Parameter Initialization - Initialize weights to identity matrix. Initialize biases to 0. This will help prevent weights from shrinking to zero.
3. Network Architecture - **Gated Cells**

   Idea: use **gates** to selectively **add or remove** information within each recurrent unit with



Long Short Term Memory (LSTM)

LSTM networks rely on gated cell to maintain information throughout many time steps.

Key Concepts:

1. Maintain a **cell state**
2. Use **gates** to control **flow of information**
    a. **Forget** gate gets rid of irrelevant information
    b. **Store** relevant information from the current input
    c. Selectively **update** cell state
    d. **Output** gate returns filtered version of cell state
3. Backpropagation through time with partially **uninterrupted gradient flow**

RNN Applications and Limitations

⇒ Music generation

⇒ Sentiment classification

Limitations -

a. Encoding bottleneck
b. Slow, no parallelization
c. Not long memory

Attention Fundamentals

What we exactly want for sequence modelling

a. Continuous streaming

b. Parallelization

c. Long memory

Intuition behind self-attention : **Attending to the most important parts of an input**

1. Identify which parts to attend to
2. Extract features with high attention

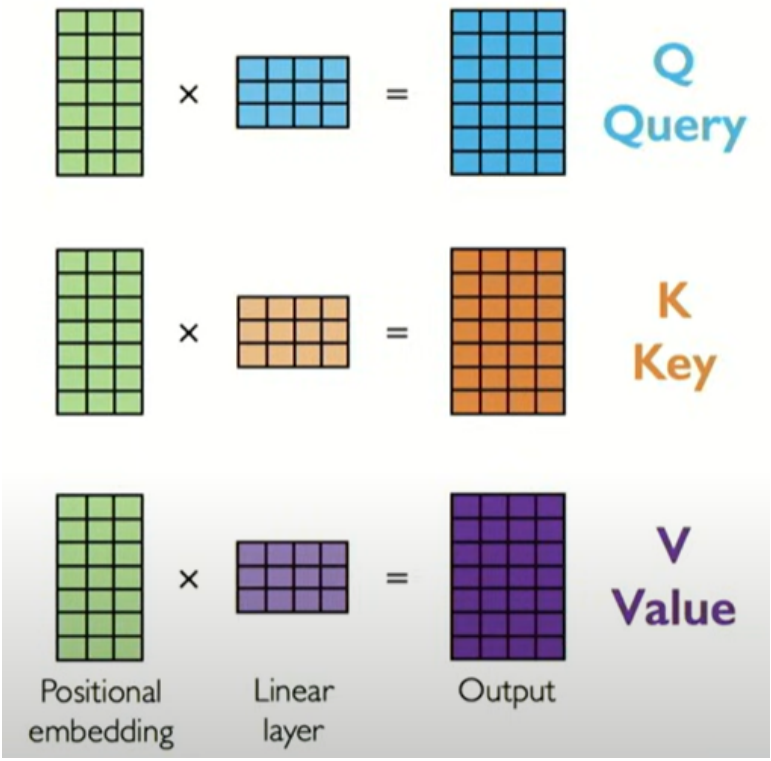
Learning Self-Attention with Neural Networks

Goal - identify and attend to most important features in input

1. Encode **position** information

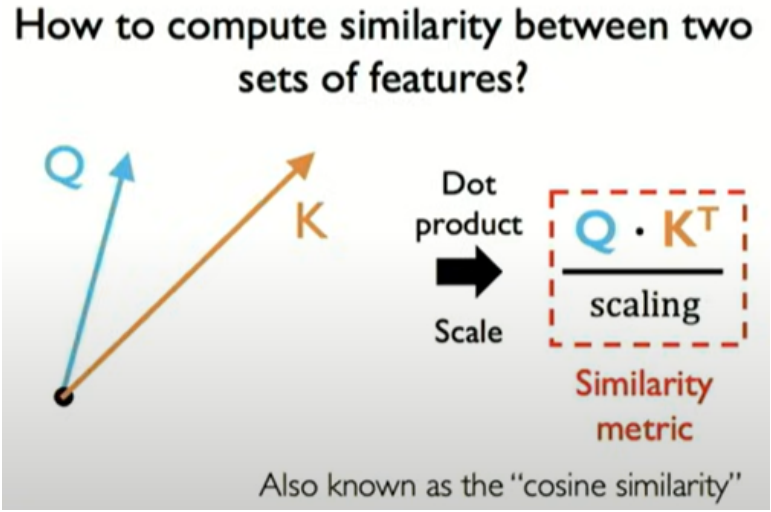
   Since data is fed all at once, we need to encode position information to understand the order. We take the embeddings and encode position info thereby giving us **Position-aware encoding**

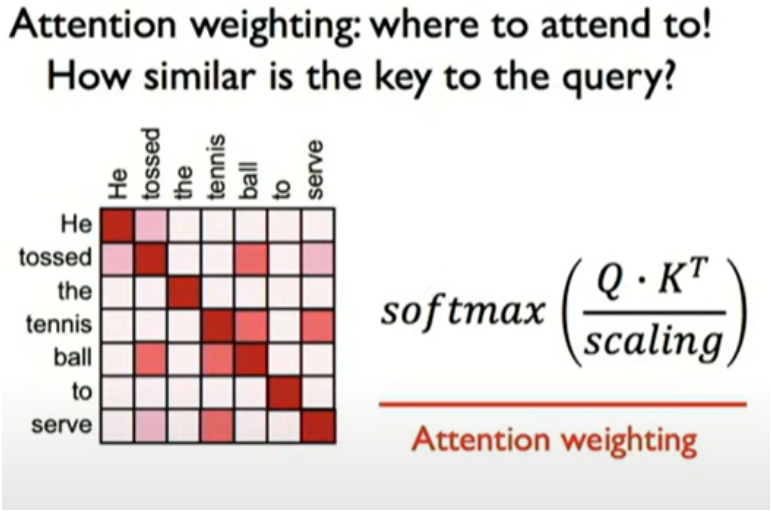
2. Extract **query, key and value** for search



3. Compute **attention weighting**

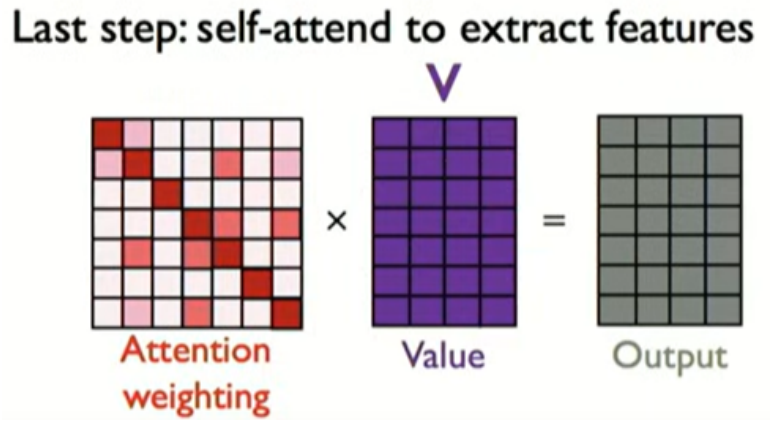
   Attention score - compute pairwise similarity between each query and key





4. Extract **features with high attention**

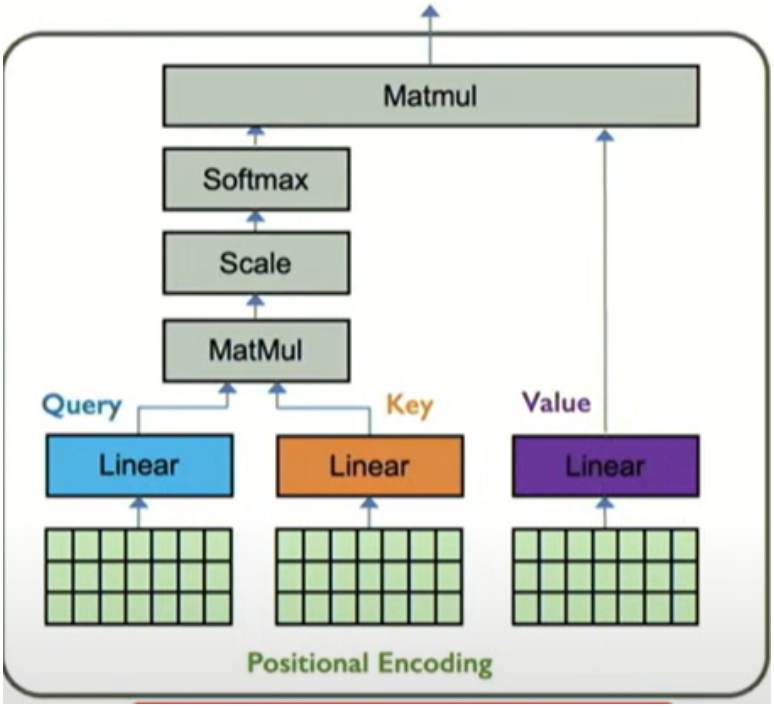
$$softmax(\frac{Q.K^T}{scaling}) \, . \, V = A(Q,\, K,\, V)$$



Conclusion of steps involved -

1. Encode position information
2. Extract query, key and value for search
3. Compute attention weighting
4. Extract features with high attention

These operations form a self-attention head that can plug into a larger network. Each head attends to a different part of input.



👉 **Attention is the foundational building block of Transformer architecture**

Self-Attention Applied

⇒ Language processing - Transformers like Bert and GPT

⇒ Biological sequences - Protein structure models

⇒ Computer vision - Vision transformers

Summary

1. RNNs are well suited for **Sequence Modelling** tasks
2. Model sequences via a **recurrent relation**
3. Training RNNs with **Backpropagation through time**
4. Models for music generation, classification, machine translation and more
5. **Self-attention** to model sequences **without recurrence**
6. Self-attention is the basis of many **LLMs**