Sequential data is commonly found anywhere and everywhere

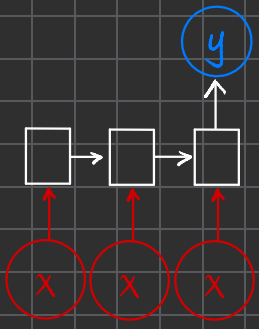
Sequencing Modeling Applications

One to One

Binary Classification

"Will a student pass this class"

Student → Pass

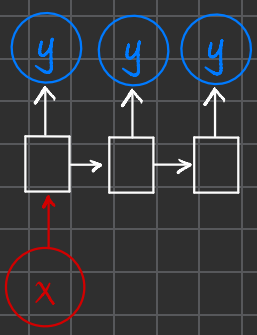


Many to One

Sentiment Classification

Text → Good

Tweet → Bad

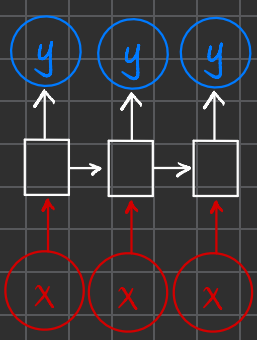


One to Many

Image captioning

Image →

"Baseball player throwing ball"



Many to Many

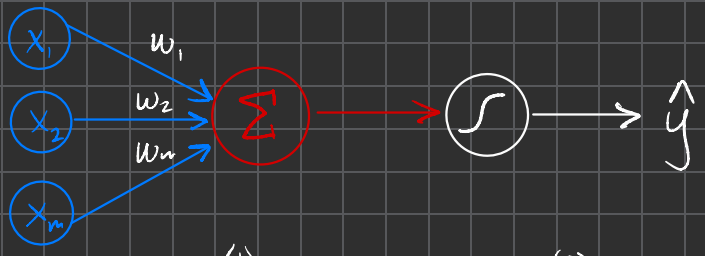
Machine Translation

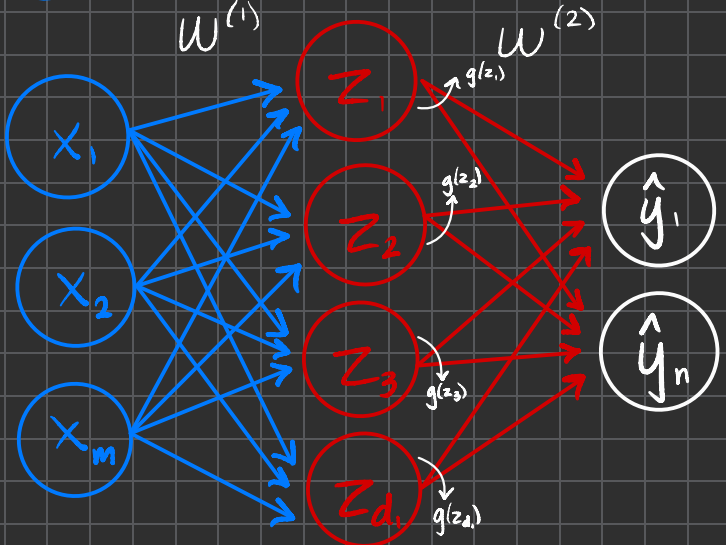
Language 1 →

Language 2

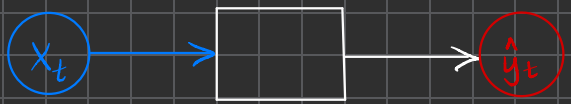
Neurons with Recurrence

Perceptron Revisited



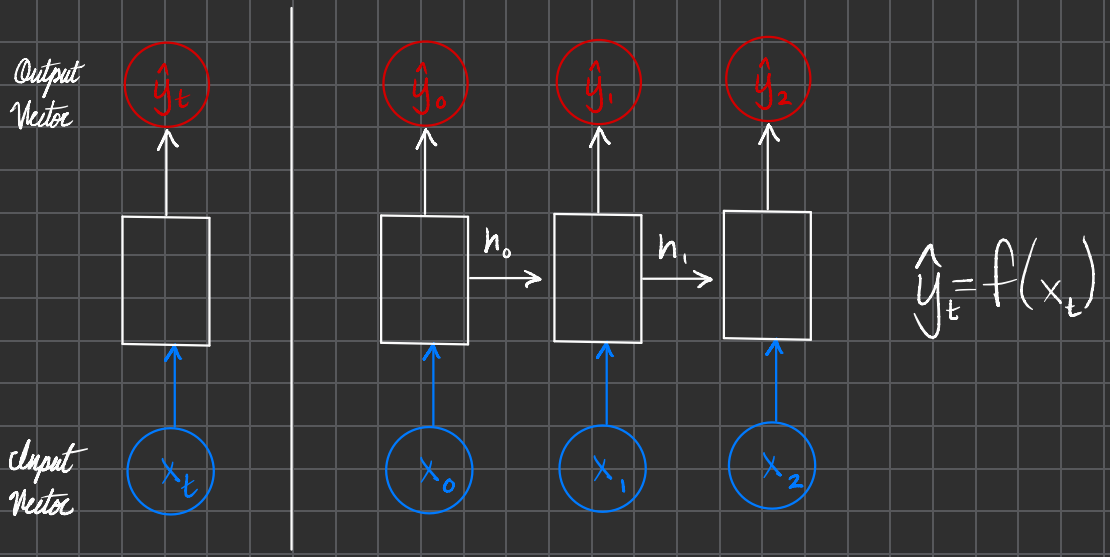


$$X \in \mathbb{R}^m \qquad \hat{y} \in \mathbb{R}^n$$



$x_t \in \mathbb{R}^m$ $\quad$ $\hat{y}_t \in \mathbb{R}^n$

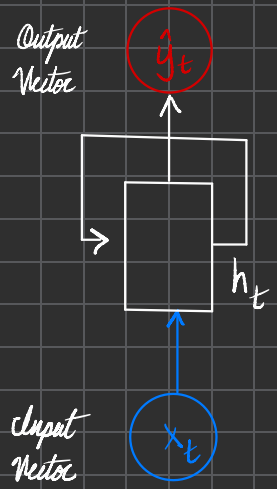
t represents time (for sequential data)



We can link the previous sequential results with h_n, which serves as memory or state.

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

These recurrent relations define Recurrent Neural Networks



functions with weights

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

RNNs have a state h that is updated at each time steps as a sequence is processed

initialize RNN

looping for recurrence

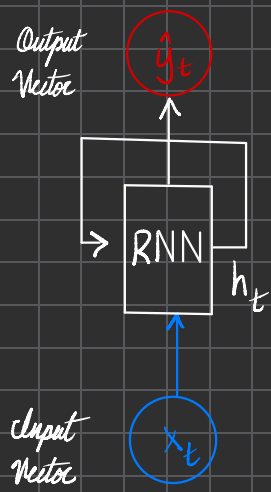
```
my_rnn = RNN()
hidden_state = [0, 0, 0, 0]

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

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

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

RNN state, output and output



Output Vector

$$\hat{y}_t = W_{hy}^T h_t$$

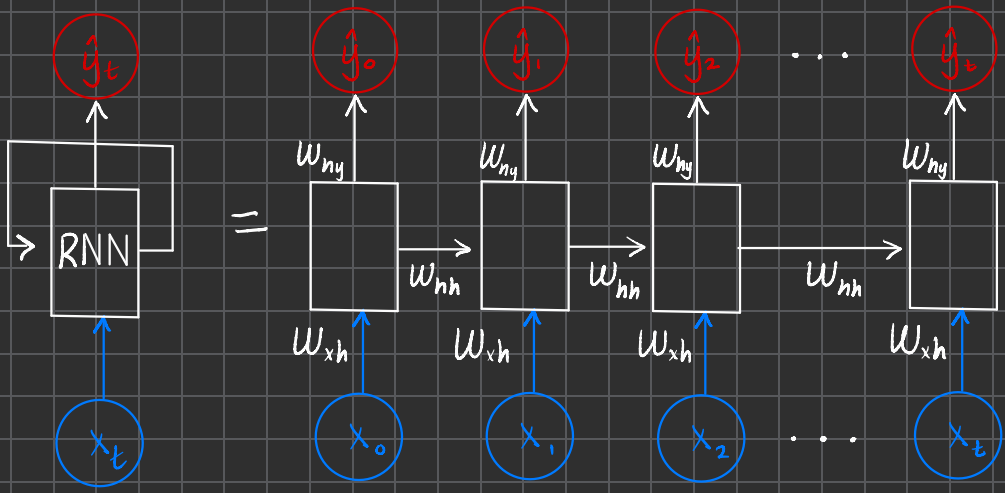
Update Hidden State

$$h_t = \tanh\left(W_{hh}^T h_{t-1} + W_{xh}^T x_t\right)$$

Input Vector

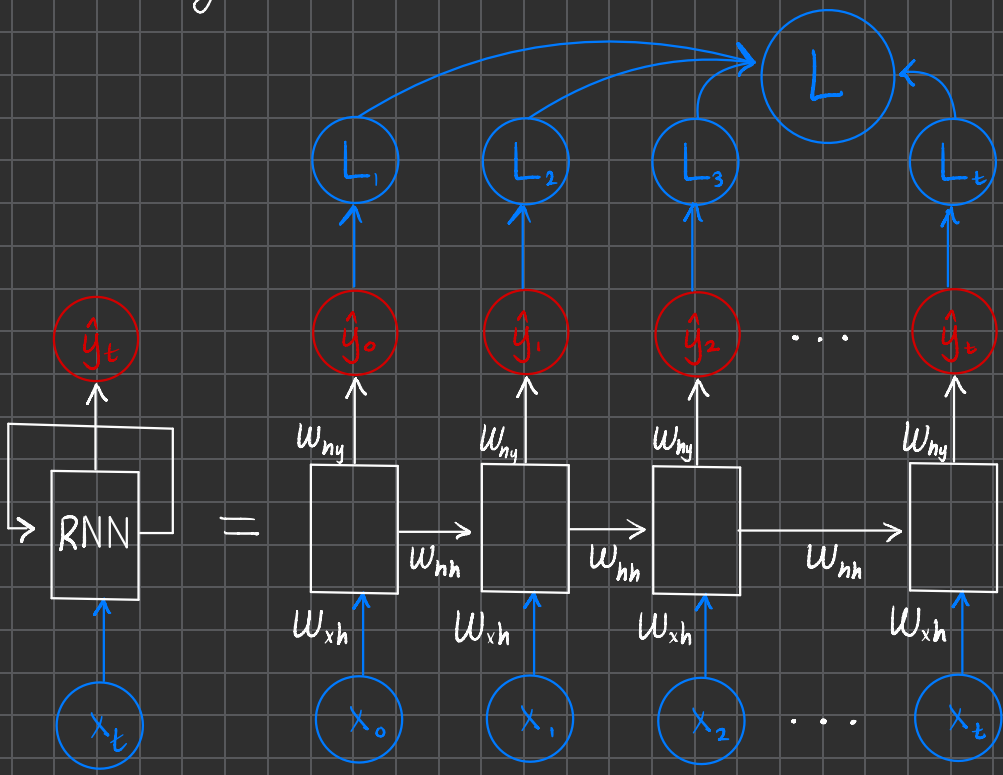
x_t

RNNs as Computational Graphs Across Time



Reuse the same weight matrices at every time step

How do you train these RNNs with loss?




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

    # Initialize weight matrices
    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
```

Abstracted to:

`tf.keras.layers.SimpleRNN(rnn_units)`

(Thanks Tensorflow)

To model sequences, we need to:

1.) Handle variable-length sequences

2.) Track long-term dependencies

3.) Maintain information on order

4.) Share parameters across the sequence

A sequence modeling problem: Predict the Next Word

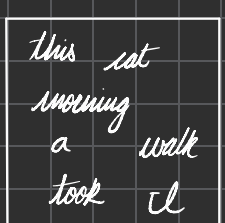
Step 1.) Representing language to a neural network

a.) Neural networks cannot interpret words

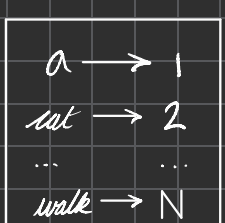
b.) Neural networks can interpret vectors

c.) Embeddings:

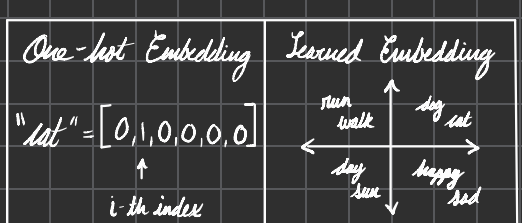
- Transforms indexes into a vector of fixed size



1.) Vocabulary



2.) Indexing



3.) Embedding

Step 2.) Handle variable length sequences

a.) "The food was ____" vs "I like ____"

Step 3.) Model Long-Term Dependencies

a.) We need information from the distant past to accurately predict the correct word

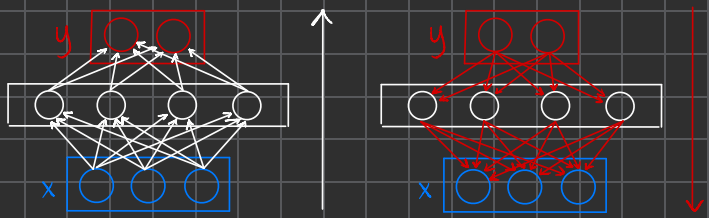
Step 4.) Captures Differences in Sequence Order

a.) "The food was good, not bad at all."

vs.

"The food was bad, not good at all."

Backpropogation Through Time (BPTT)

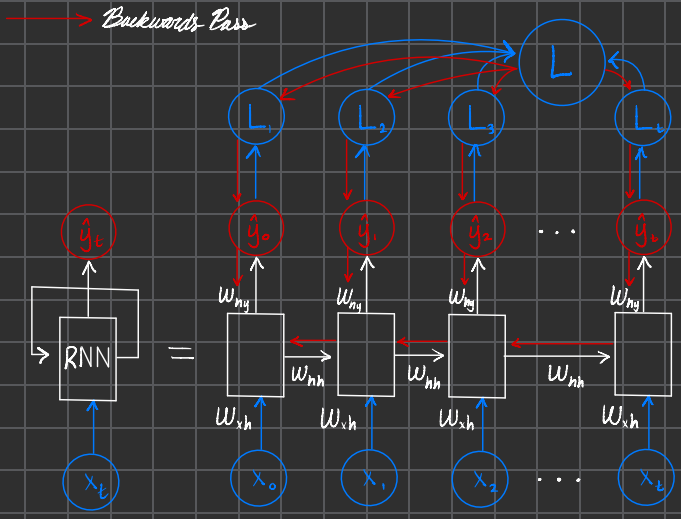


Backpropogation Algorithm:

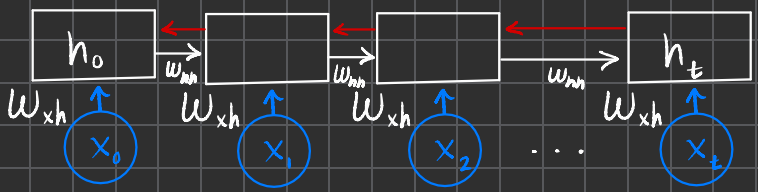
1.) Take the derivative (gradient) of the loss with respect to each parameter

2.) Shift parameters in order to minimize loss

In RNNs, you backpropogate loss through each network in time t. Then backpropagate across all time steps.



Standard RNN Gradient Flow



Computing the gradient wrt h_0 involves many factors of W_{hh} + repeated gradient computation

Many values > 1:	Many values < 1:
Exploding Gradients	Vanishing Gradients
Fix: Gradient Clipping	Fix: Activation, Network Architecture

Why are vanishing gradients a problem?

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

To fix vanishing gradients:

Trick #1: Activation Functions

$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & z \leq 0 \end{cases}$$

Using ReLU prevents g' from shrinking the gradients when $x > 0$

Sigmoid Derivative tanh derivative ReLU derivative

Trick #2: Parameter Initialization:

Initialize weights to identity matrix

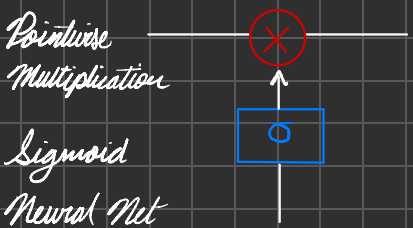
Initialize biases to zero

$$I_n = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

Helps prevent weights from shrinking to zero

Trick #3: Gated Cells

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

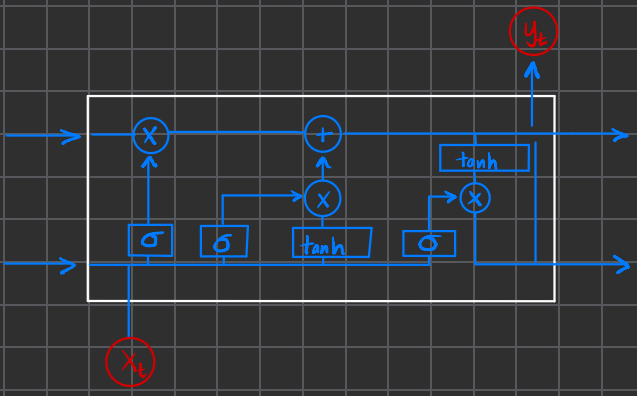


Gated Cell
LSTM, GRU, etc.

Long Short Term Memory (LSTM) networks rely on a gated cell to track information throughout many time steps

Gated LSTM cells control information flow:

1.) Forget 2.) Store 3.) Update 4.) Output



= tf.keras.layers.LSTM(num_units)

LSTM Key Concepts

1.) Maintain a cell state

2.) Use gates to control the flow of information

- Forget gate gets rid of irrelevant information
- Store relevant information from current output
- Selectively update cell state
- Output gate returns a filtered version of cell state

3.) Backpropagation through time with partially uninterupted gradient flow

RNN Applications and Limitations

Example Task: Music Generation

Input: Sheet music

Output: next character in sheet music

Example Task: Sentiment Classification

Input: Sequence of words

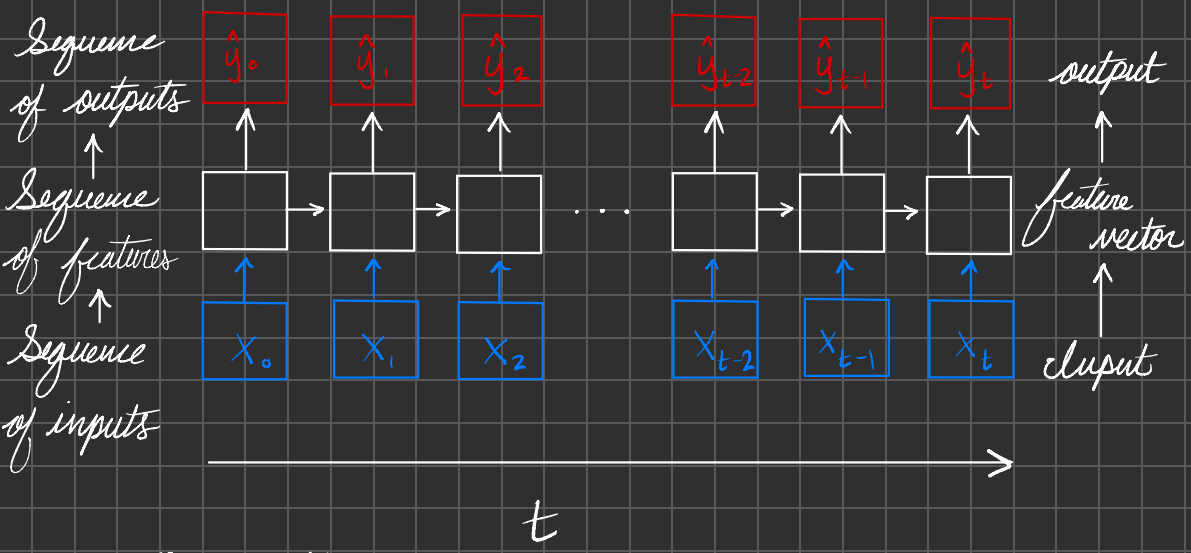
Output: probability of having positive sentiment

(loss = tf.nn.softmax_cross_entropy_with_logits(y, predicted))

Limitations of Recurrent Models

- Encoding Bottleneck
- Slow, no parallelization
- Not long memory

Goal of Sequence Modeling



Desired Capabilities:

- Continuous stream of data
- Parallelization of computation
- Long-memory capable

Approach #1: Squash data to eliminate recurrence

- Naive
- One vector input with data from all time points
- Still not scalable
- Lost our "in order" information

Approach #2: Attention

- Foundational mechanism of transformer architecture

Attention

Intuition Behind Self-Attention

- Attending to the most important part of an input
  - Identify which parts to attend to
  - Extract features with high attention

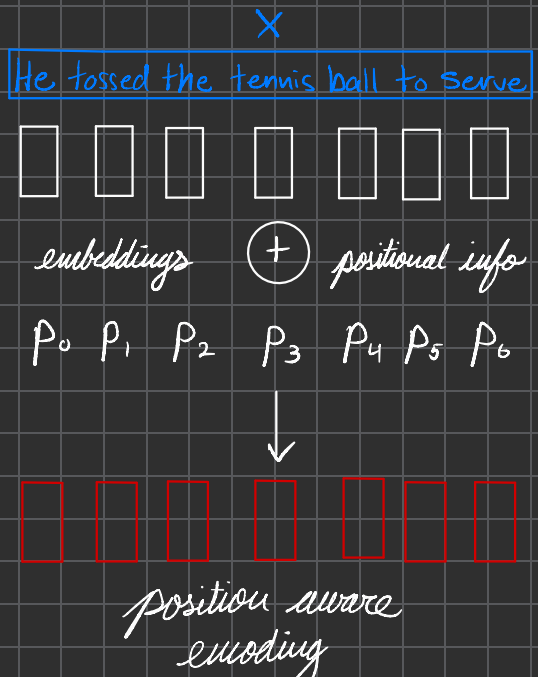
Similar to a search!

Self-Attention

Goal:
Identify and attend to most important features in input

1.) Encode position information

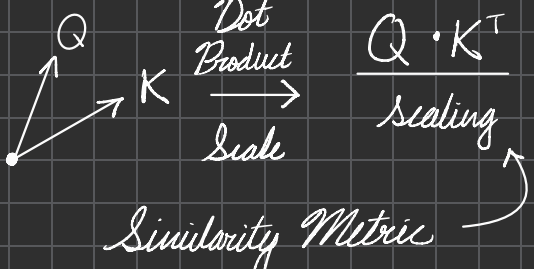
2.) Extract query, key, value for search (Use neural network layers one for each result)

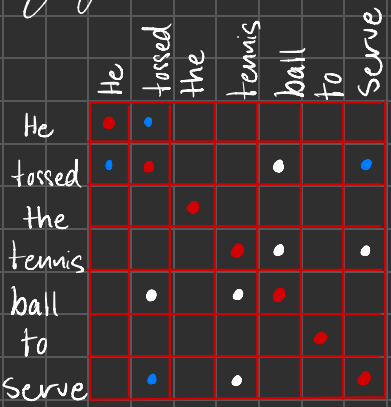


3.) Compute attention weighting

Attention Score: Compute pairwise similarity between each query and key

How do you compute similarity between two sets of features?



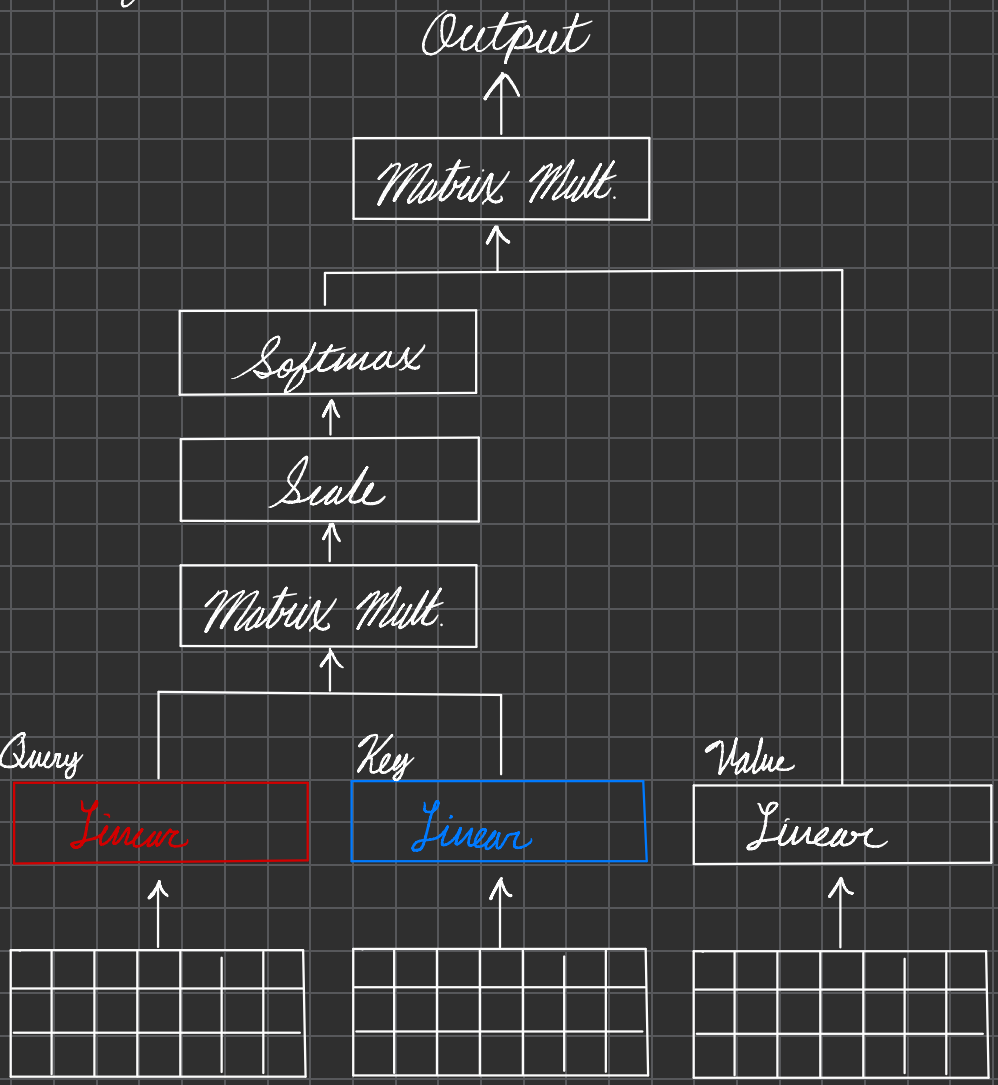


$$\text{Similarity Metric} = \text{softmax}\left(\frac{Q \cdot K^T}{\text{Scaling}}\right)$$

4.) Extract features with high attention

$$\text{softmax}\left(\frac{Q \cdot K^T}{\text{Scaling}}\right) \cdot V = A(Q, K, V)$$

One Self Attention Head:



(These operations form a self attention head that can plug into a larger network)

Self Attention powers:

- GPT 3
- BERT
- AlphaFold 2
- Vision Transformers