

Intro to deep learning 2

Wednesday, August 20, 2025 12:46 PM

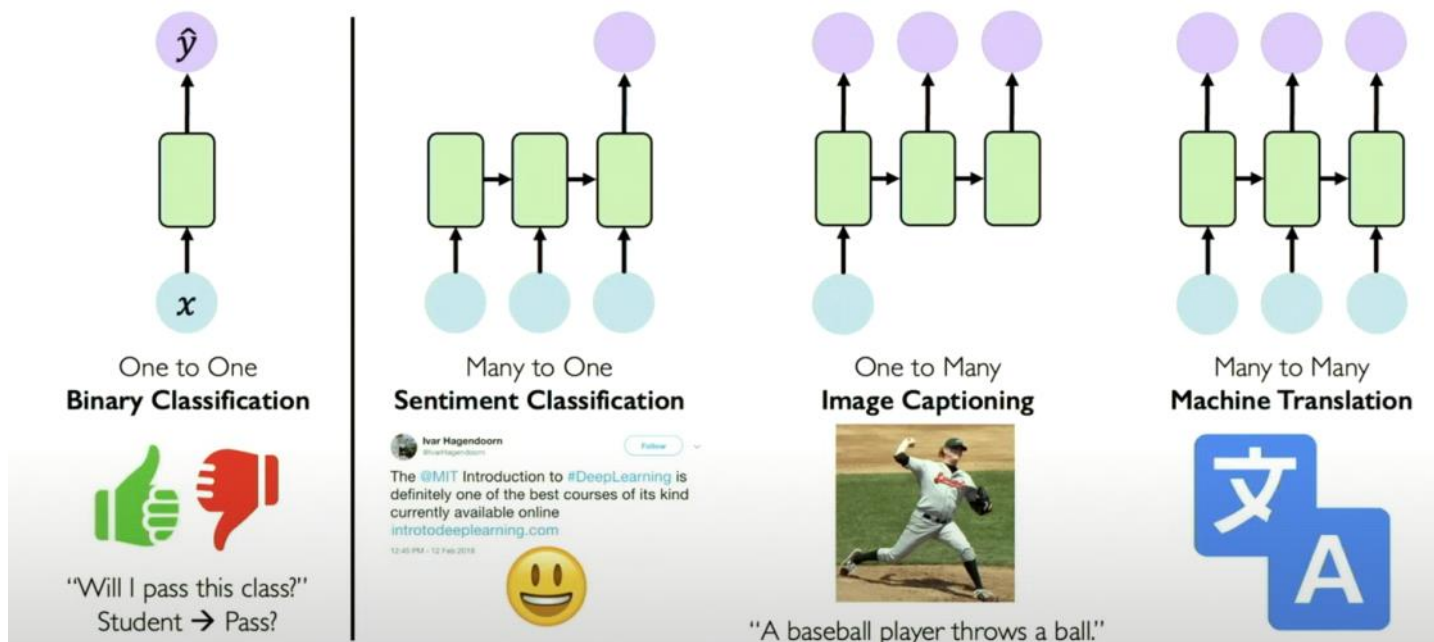
Lecture 2

Deep sequence modeling lecture

Sequential data and seq modeling sequential data is any form of data with a dependence of some degree over time up to now. Examples: DNA, market data, audio waves, texts....

Various types of modeling: one to one, many to one (sentiment analysis), one to many (video generation), many to many (translation of sequences of words). There are many other architectures and criteria used but the main ones are in this lecture

Sequence Modeling Applications

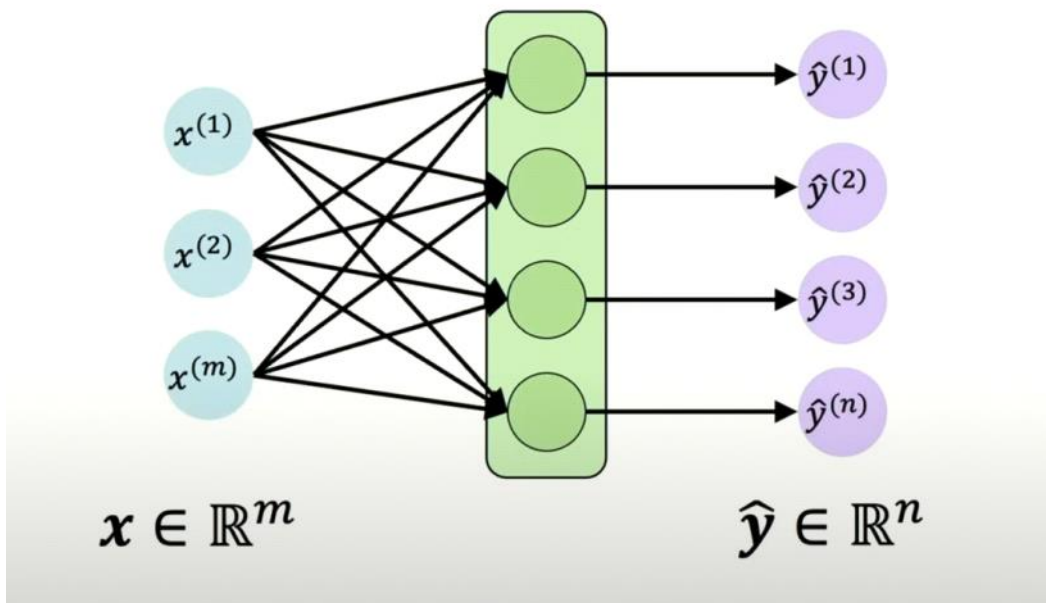


How is this done with NN?

Neurons with recurrence, revisiting the neuron

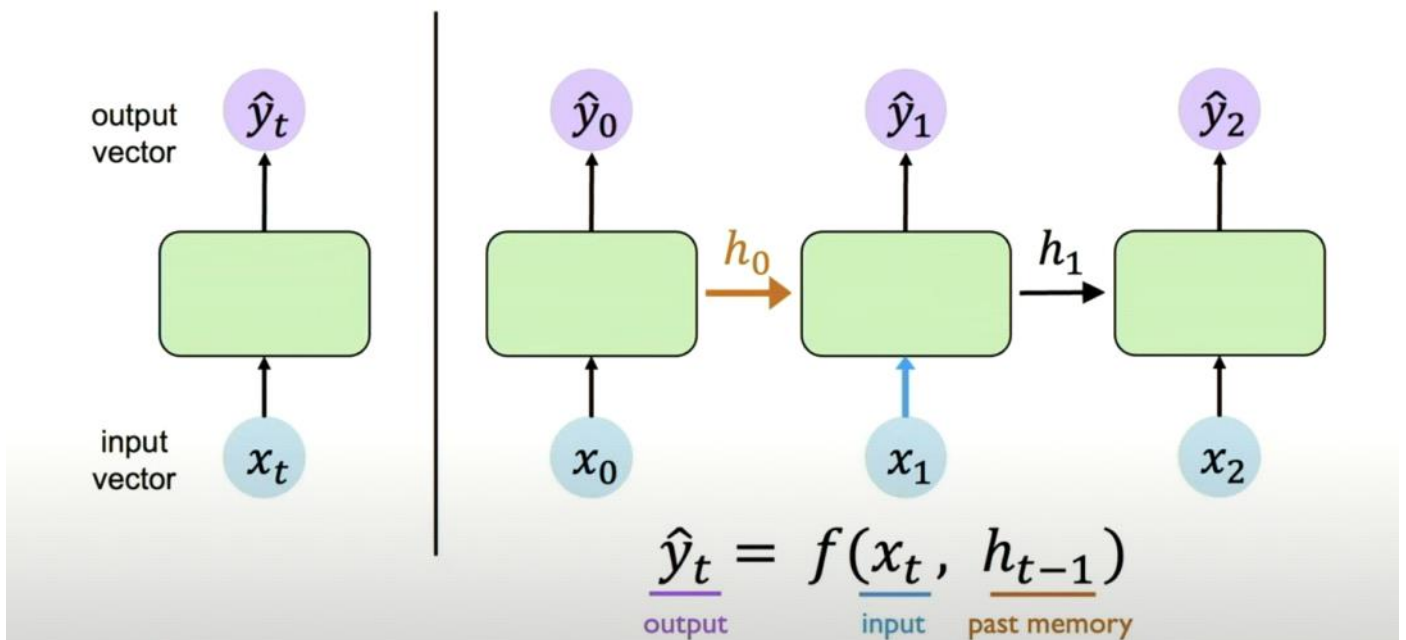
FFN as we know it. This does not give you any sequence, but it is a static input

Feed-Forward Networks Revisited

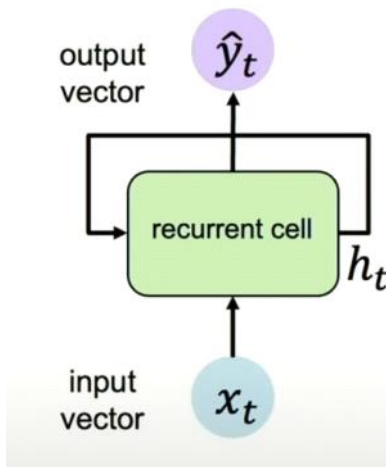


We build out the same FNN where there is a relationship between time steps using a variable h_t called state of the network. We are capturing what happens previously in time steps $t-1$ and is passed forward

Neurons with Recurrence



All wrapped up

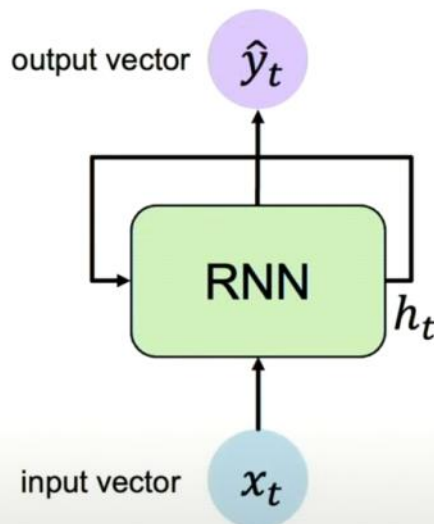


This is in few words a recurrent neural network RNN

RNNs

RNNs have states h_t updated at each time step. The relation is given below in the image. The set of weights is both function of h and the information x

Recurrent Neural Networks (RNNs)



Apply a **recurrence relation** at every time step to process a sequence:

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

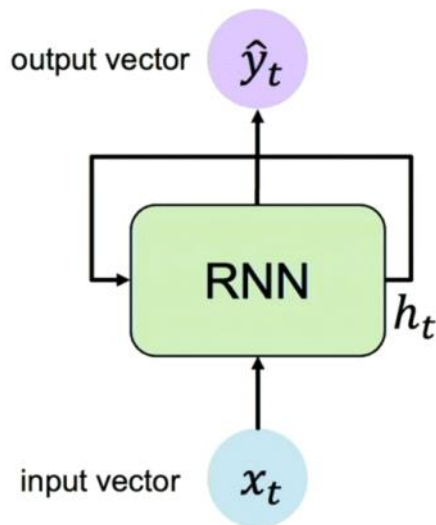
cell state
function with weights W
input
old state

Note: the same function and set of parameters are used at every time step

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

How the output is generated is similar to the other networks, but we have an extra equation for weight matrices to update h . We will also see later how are the various activation functions can be used

RNN State Update and Output



Output Vector

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

Update Hidden State

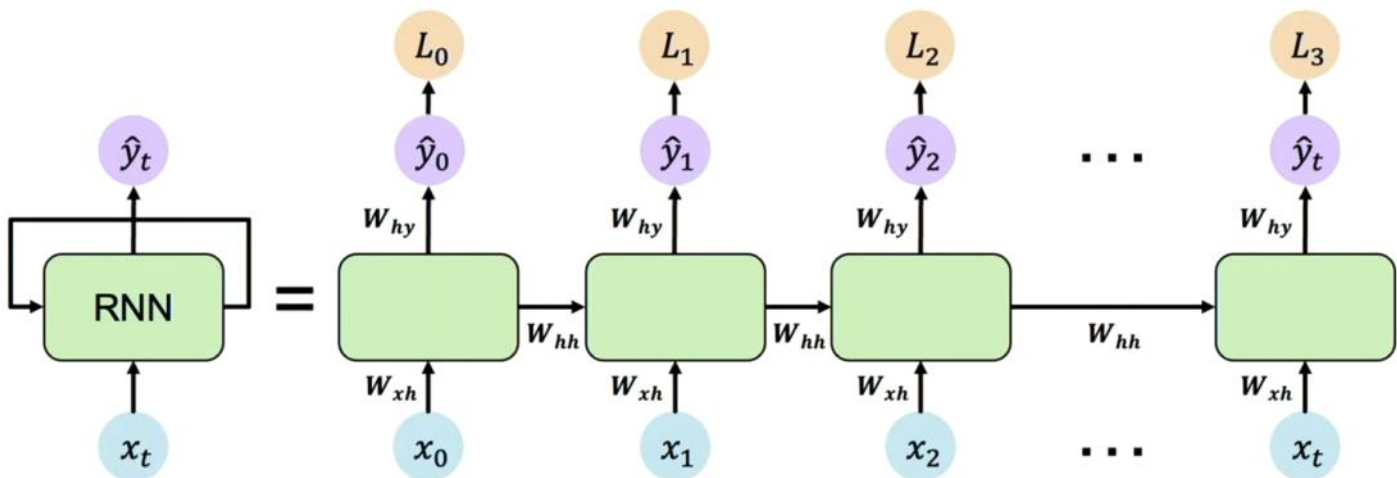
$$h_t = \tanh(W_{hh}^T h_{t-1} + W_{xh}^T x_t)$$

Input Vector

$$x_t$$

With this way of doing things over time steps, we have a loss at each step. Note all the weight matrices KEEP being updated over time steps. Below a computational graph

→ Forward pass



We can handle the losses by summing them together. The pseudocode looks like below

RNNs from Scratch



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

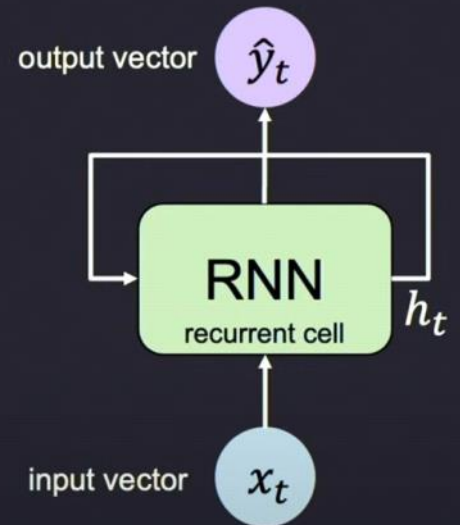
        # Initialize weight matrices
        self.W_xh = self.add_weight([rnn_units, input_dim])
        self.W_hh = self.add_weight([rnn_units, rnn_units])
        self.W_hy = self.add_weight([output_dim, rnn_units])

        # 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
```



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



Design criteria

What unique aspects we need the NN to capture to handle well the sequence data? Sequences have various lengths, long-term dependencies, specific order, and share parameters across the sequence. RNN cover those criteria with their architecture

Example problem: next word prediction

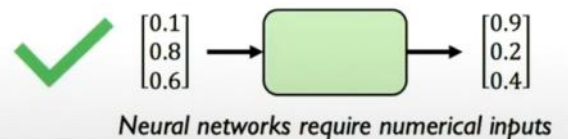
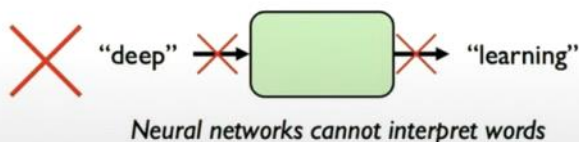
A neural network is a numerical operator. How do I feed this sequence data to the network? We transform the input in a numerical vector of fixed size

"This morning I took my cat for a walk."

given these words

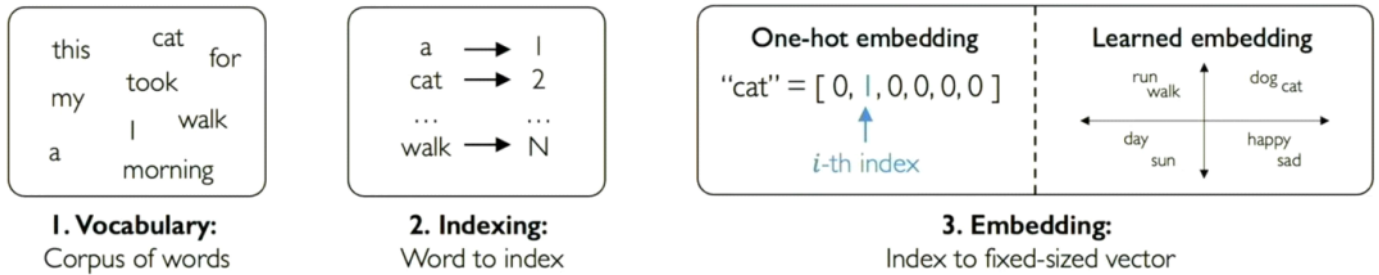
predict the
next word

Representing Language to a Neural Network



There are various ways to do this. A one-hot index with a huge vector encoding each word. Another way is to learn an embedding projecting words into a vector space (word2vec)

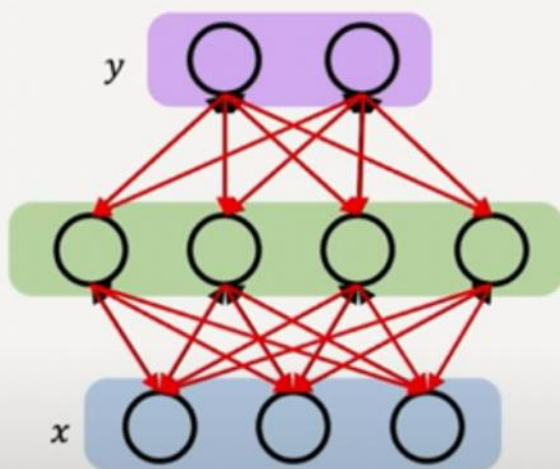
Embedding: transform indexes into a vector of fixed size.



Backpropagation through time BPTT

How the training is done this time! In FNN we propagate the gradient to adjust the parameters and minimize the loss.

Recall: Backpropagation in Feed Forward Models

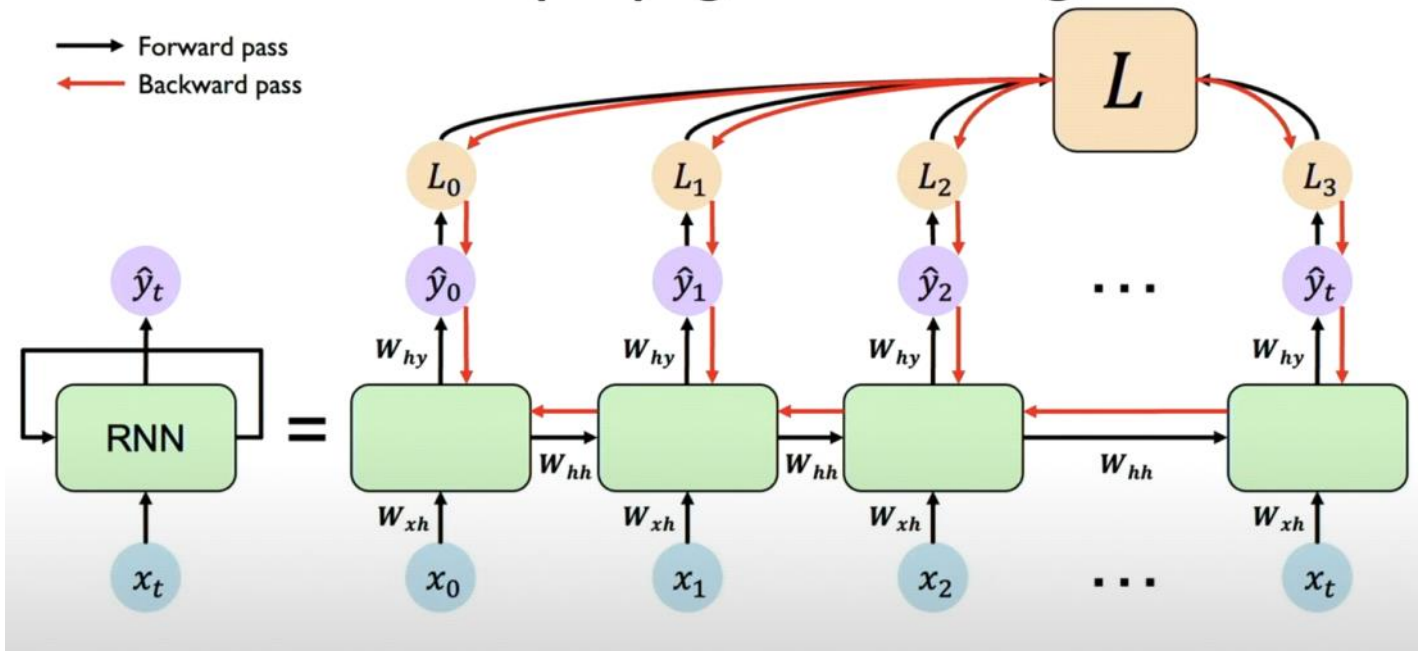


Backpropagation algorithm:

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

Now our loss is computed over time steps and added at the end. We need to backpropagate through each single loss the weight matrix for the output, and backpropagate through each time step of the networks. For each loss you thus continue through a different nr of time steps with the backpropagation

RNNs: Backpropagation Through Time



Gradients can become too big or small and there are techniques to solve the problem. Gradient clipping for large values, or network architecture, weight initialization and choice of activation fun for vanishing gradients.

Vanishing gradients are in reality diminishing the capacity of modeling of the network because with very long sequences you have a hard time keep track of all the time sequence

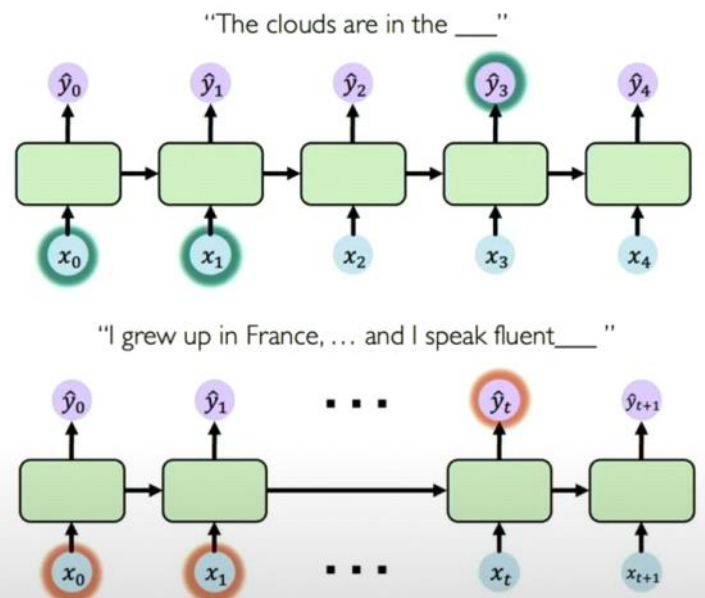
The Problem of Long-Term Dependencies

Why are vanishing gradients a problem?

Multiply many **small numbers** together

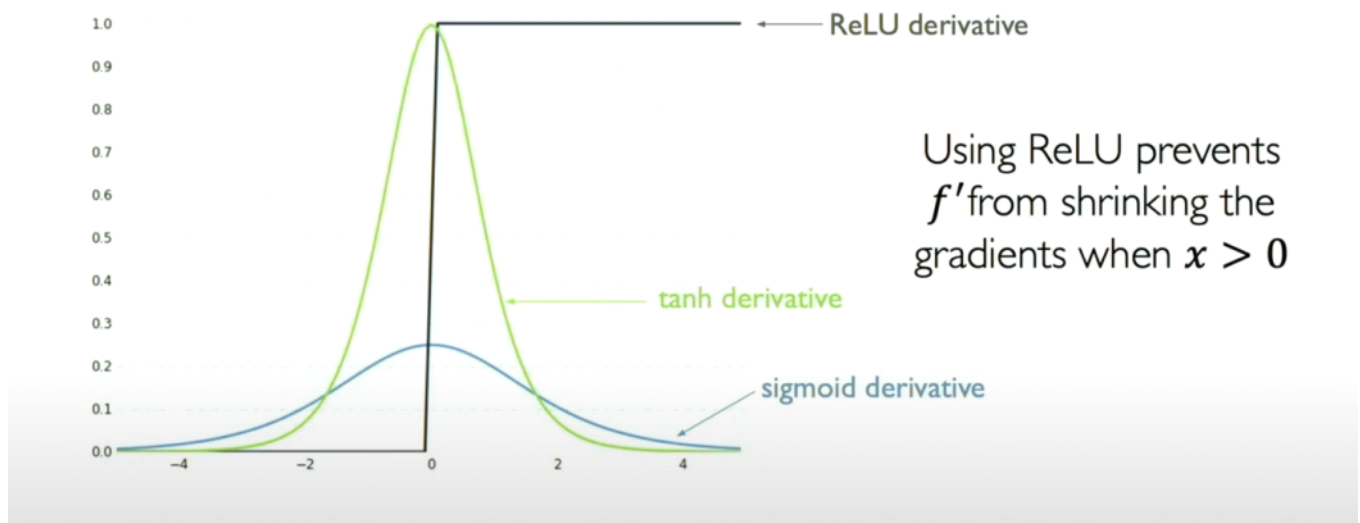
Errors due to further back time steps have smaller and smaller gradients

Bias parameters to capture short-term dependencies



RELU activation is also good for this problem because the derivative helps reducing the shrinking when x is positive

Trick #1: Activation Functions



Initialize weights which reduce shrinking

Trick #2: Parameter Initialization

Initialize **weights** to identity matrix

Initialize **biases** to zero

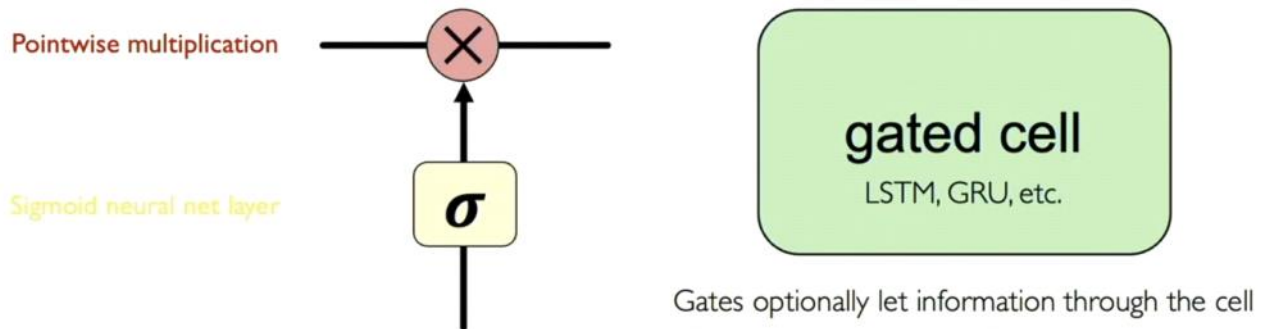
$$I_n = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}$$

This helps prevent the weights from shrinking to zero.

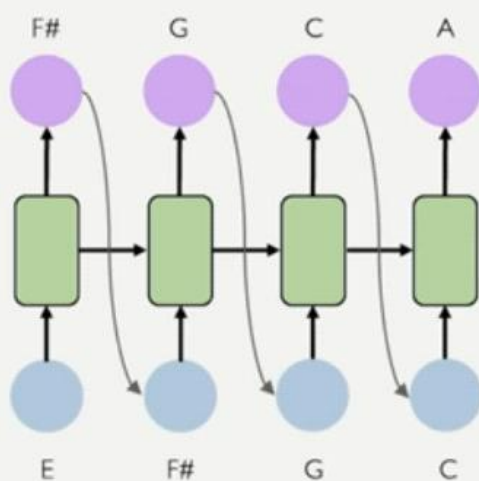
Keep or remove specific informations from the product with gating cells. Gating is for example employed in NN models such as LSTM (Long-Short term memory neural network). The update of cell state is such that an additional layer decides how to keep or ignore some of the information. The update to the cell state is then more intelligent and keeps necessary information. All this is part of the backpropagation of gradient.

Trick #3: Gated Cells

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



Example Task: Music Generation



Input: sheet music

Output: next character in sheet music

Listening to
3rd movement



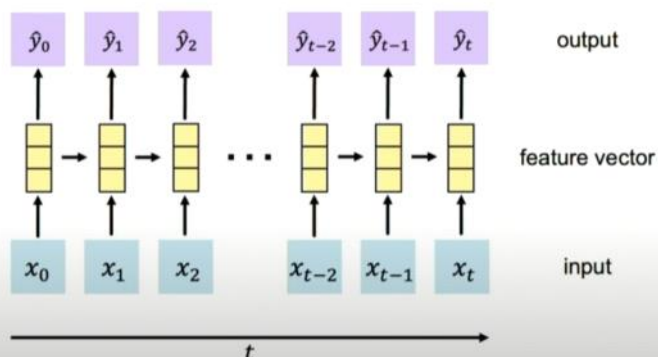
RNN are not always the answers because they have a bottleneck in the size of the hidden state. It is slow (not parallelized), it does not have real long term memory
We want models able to handle a continuous stream of information, a model efficient with large memory capacity

Desired Capabilities

Continuous stream

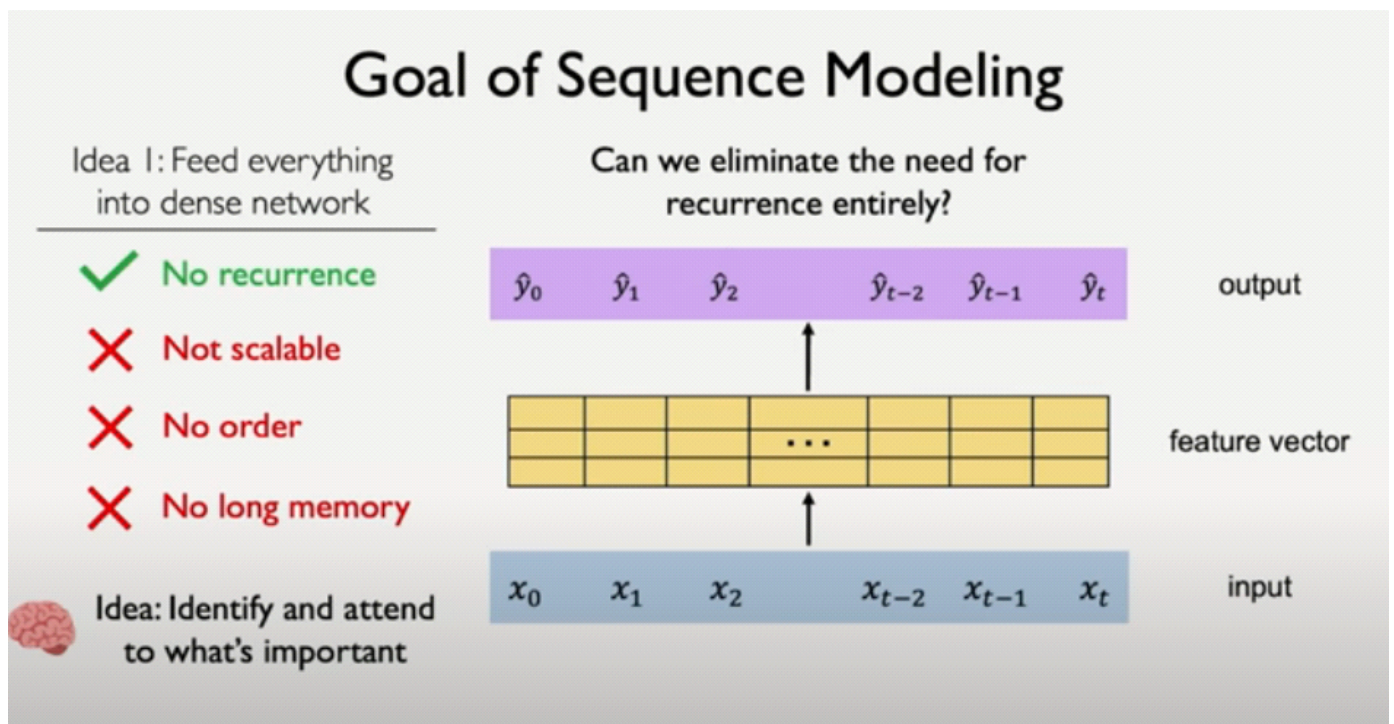
Parallelization

Long memory

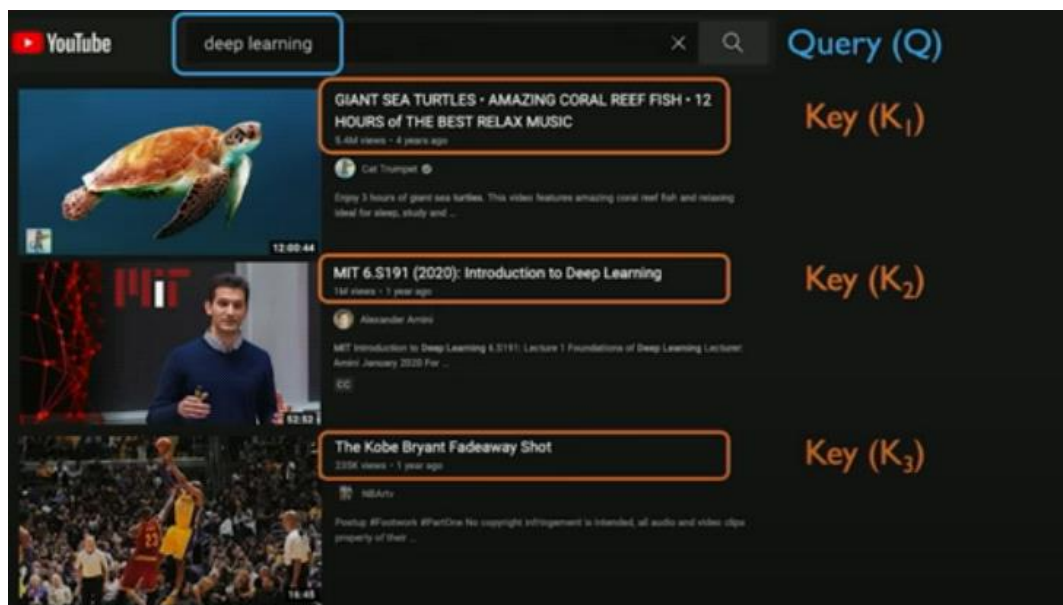


One could think about eliminating recurrence and concatenate all the data at once to generate output at once. This eliminates all notions of sequence and the scalability is not existing because we have a huge input. The memory capacity is also lost. Another idea is to learn a NN model which sees which parts of a sequence are important and convey info we should be capturing and learning. This is the core **idea of ATTENTION**. The **TRANSFORMER** is a type of NN where attention is at the foundation of its mechanism.

ATTENTION - SELF ATTENTION



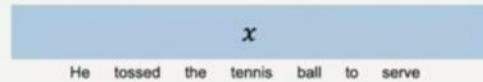
You need to find in the data the important parts. This is similar to a search problem. Let's look at how this work. For your query you want to find the most similar Set of features, a key. The most similar becomes your value. For example



In practice: Note we have gone past the recurrence need, we have instead all data at once. We want to extract the most important features.

Learning Self-Attention with Neural Networks

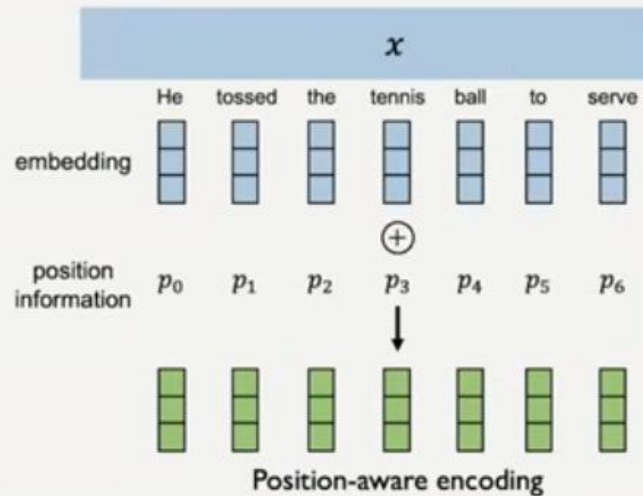
Goal: identify and attend to most important features in input.



First of all you need to encode the information about order and position. In self attention and transformer we use a position aware encoding with positional embeddings

Goal: identify and attend to most important features in input.

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



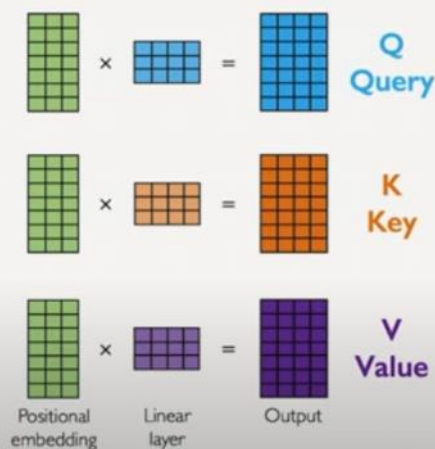
Data is fed all at once and also preserve positions now. Afterwards you need to extract the Query Q, Key K and Value V for search.

This is done learning three NN layers, which are three sets of matrices. Those are done with three NN layers on the same positional embedding.

Learning Self-Attention with Neural Networks

Goal: identify and attend to most important features in input.

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



The attention score is done with the pairwise similarity of Query and Key. Here you are effectively comparing which data features are important.

Similarity is computed with their dot product (cosine similarity). We apply a softmax function to the scaled similarity

so we have probabilities. The final step

is now to take the attention weighting and multiplying it by the Value.

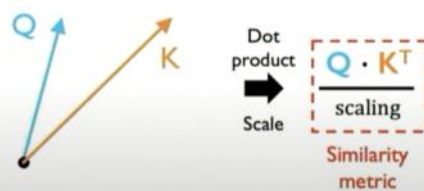
So we identify what is important and **extract** relevant features (self attendance step).

Goal: identify and attend to most important features in input.

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

Attention score: compute pairwise similarity between each **query** and **key**

How to compute similarity between two sets of features?



Also known as the "cosine similarity"

What does it all mean?

Learning Self-Attention with Neural Networks

Goal: identify and attend to most important features in input.

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

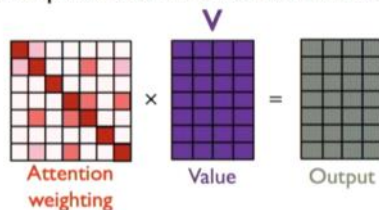
Attention weighting: where to attend to!
How similar is the key to the query?



Goal: identify and attend to most important features in input.

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

Last step: self-attend to extract features



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

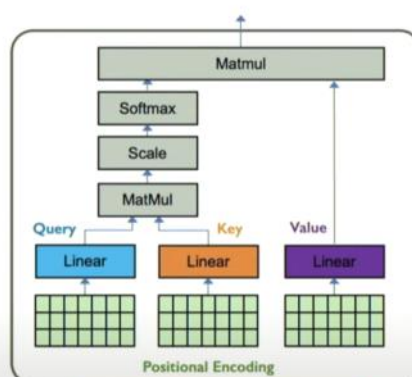
Wrapping up

Learning Self-Attention with Neural Networks

Goal: identify and attend to most important features in input.

1. Encode **position** information
2. Extract **query, key, 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.



$$\text{softmax} \left(\frac{Q \cdot K^T}{\text{scaling}} \right) \cdot V$$

You can have multiple attention heads which extract different sets of features that are relevant in the data

Applications examples

Language Processing



An armchair in the shape of an avocado

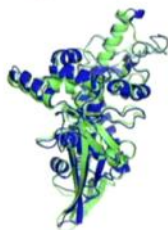
Transformers: BERT, GPT

Devlin et al., NAACL 2019
Brown et al., NeurIPS 2020



6.S191 Lab and Lecture!

Biological Sequences



Protein Structure Models

Jumper et al., Nature 2021
Lin et al., Science 2023

Computer Vision



Vision Transformers

Dosovitskiy et al., ICLR 2020

Deep Learning for Sequence Modeling: Summary

1. RNNs are well suited for **sequence modeling** tasks
2. Model sequences via a **recurrence 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 for many **large language models** – stay tuned!

Lab 1: Introduction to TensorFlow and Music Generation with RNNs

Link to download labs:
<http://introtodeeplearning.com/schedule>

1. Open the lab in Google Colab
2. Start executing code blocks and filling in the #TODOs
3. Need help? Find a TA/instructor!