Transformers

Ivan Zelich

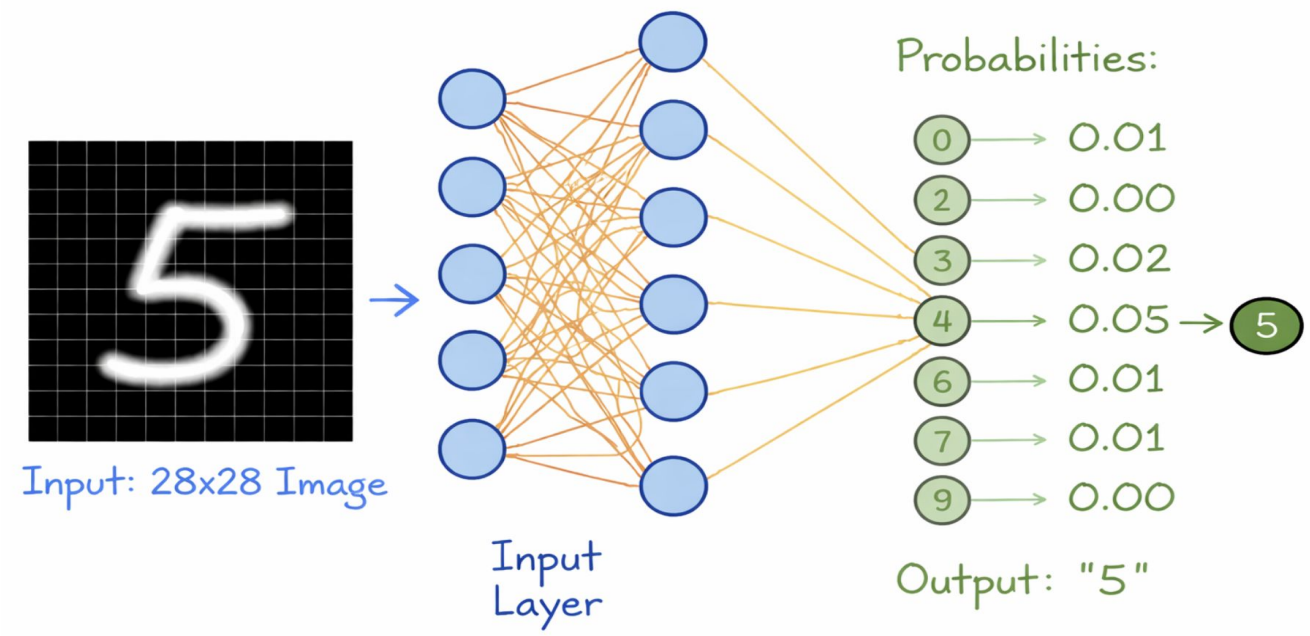
What is a neural network and how to make one?

The problem:

We have something that we want to predict (the output) given a huge amount of data the (input).

Neural networks are simply functions from the input to the output.

Example


Input: 28x28 Image
Input Layer
Probabilities:
0 → 0.01
2 → 0.00
3 → 0.02
4 → 0.05 → 5
6 → 0.01
7 → 0.01
9 → 0.00
Output: "5"

What is a neural network and how to make one?

The basic building blocks are three steps:

- **Forward pass**: Data is compressed into a smaller space, and a final function outputs our predictions
  - Typically a combination of linear transformations and non-linear activation functions
  - An activation function has the ability to ‘turn-on/off’ components in a vector
- **Loss function**: Used to measure how far our predictions are from ‘ground truths’
- **Backward pass**: Our function is then iteratively improved via an optimization protocol (typically some variant of gradient descent)

A brief history of LLMs

- ~1940s: Mccullough and Pitts conceived the idea of a neural network
- ~1950s: F. Rosenblatt introduced the perceptron
- ~1960s: S. Amari and H. Saito proposes the first deep learning neural network trained with gradient descent
- ~1980s: D. Rumelhart, G. Hinton and G. Williams popularized the use of back-propagation in neural networks
- ~2000s: GPUs become integrated with deep learning, and many different language learning algorithms became trainable (word2vec, glove, etc…)
- 2017: ‘Attention is all you need’ paper, detailed a highly scalable model, the Transformer, able to capture complex interrelationships of the input
- 2018-: Large Language Models as we know them today were developed.

The Transformer

Issue: Human language has strong temporal relationships.

How can we model this?

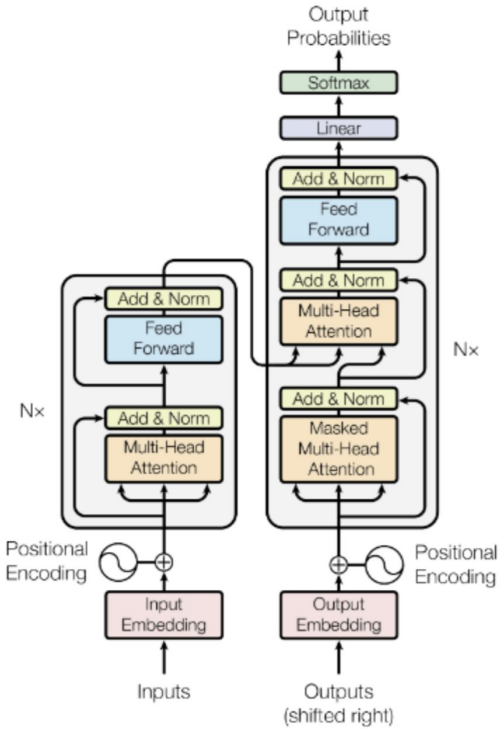
Key: Let the computer learn how to ‘attend’ to parts of sentences.

The modelling of language set-up.

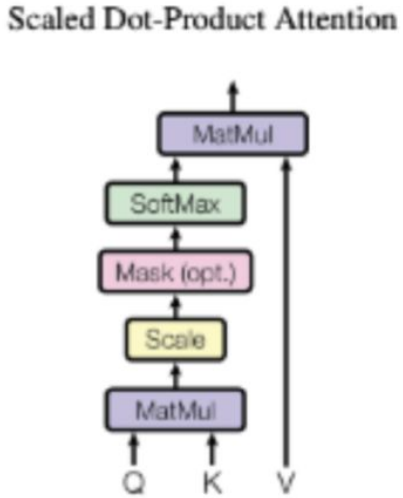
- Inputs: Vocabulary, embedded inside a large vector space $\mathbb{R}^V$
- Hidden layers: an $\mathbb{R}^d$ space, $d << V$
- Output: vector in $\mathbb{R}^V$ representing ‘probabilities’

In essence, a ‘matrix factorization’: $\mathbb{R}^V \to \mathbb{R}^d \to \mathbb{R}^V$

The Transformer



Attention



- Training data: Batches of sentences X
- Learnable parameters: $\{W_Q, W_K, W_V\}$
- $X \in \mathbb{R}^{d_v \times d_c}, W_Q \in \mathbb{R}^{d_Q \times d_v}, W_K \in \mathbb{R}^{d_K \times d_v}, W_V \in \mathbb{R}^{d_V \times d_v}$
- Here, we will just assume $d_K = d_K = d_V = d_h$
- Attention matrix (HUGE!): $A = W_K^T W_Q \in \mathbb{R}^{d_v \times d_v}$

Assume X is a sentence represented by $\{x_1, \ldots, x_{c-1}, x_c\}$
For every $t \in [c]$, we will compute keys, queries, values

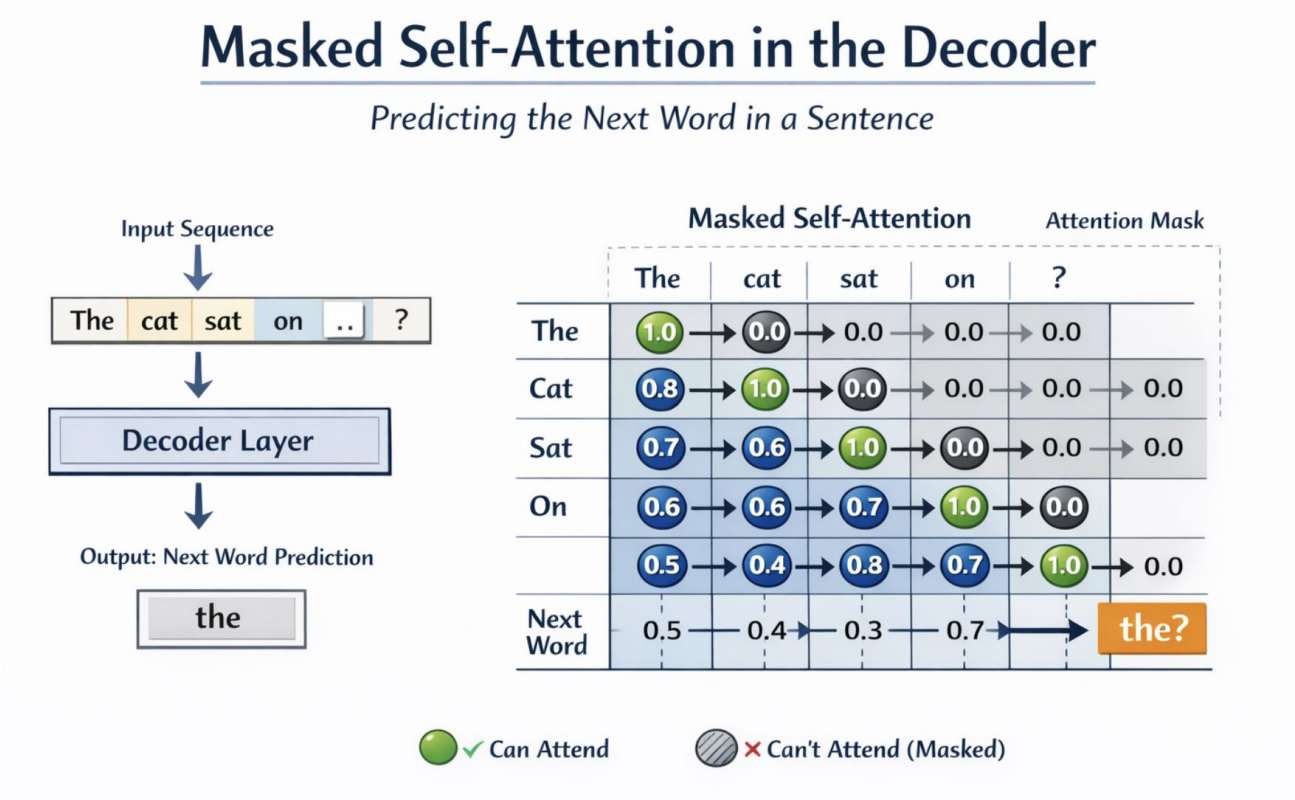
$$\{y_t^Q, y_t^K, y_t^V\}$$

For any two $t, s \in [c]$, the quantity

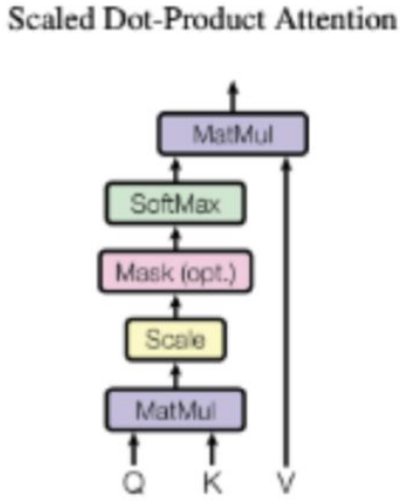
$$A_{st} = y_s^K \cdot y_t^Q$$

represents ‘how much the meaning of word t is explained by word s’.

Attention


Masked Self-Attention in the Decoder
Predicting the Next Word in a Sentence
Input Sequence
The cat sat on .. ?
Decoder Layer
Output: Next Word Prediction
the
Masked Self-Attention
Attention Mask
The cat sat on ?
The 1.0 0.0 0.0 0.0 0.0
Cat 0.8 1.0 0.0 0.0 0.0 0.0
Sat 0.7 0.6 1.0 0.0 0.0 0.0
On 0.6 0.6 0.7 1.0 0.0
0.5 0.4 0.8 0.7 1.0 0.0
Next Word 0.5 0.4 0.3 0.7 the?
Can Attend
Can't Attend (Masked)

Attention



After computing out attention matrix we will apply a scaling, then a mask (if we're trying to predict), and then a final softmax normalization.

Given our attention matrix $A_{st} = y_s^K \cdot y_t^Q$
fixing a t, the values $\{A_{st}\}_{s\in[d_h]}$ represent a set of attention weights for the word y_t.
There are h different values, so our normalization procedure is:

$$a_{st} = \mathrm{SoftMax}\left(\frac{(A_{st})_{s\in[d_h]}}{\sqrt{d_h}}\right)_s$$

our 'context-aware' representation of x_t is finally

$$x_t \mapsto \sum_{s\in[d_h]} a_{st} y_s^V$$

Attention: Key points

- Given a sentences consisting of (tokenized words), we learn the three matrices $\{W_Q, W_K, W_V\}$
- Our ‘context-aware’ embedding of each word is computed via the attention matrix $A = W_K^T W_Q$ which is computationally expensive.
- Our final output after attention and normalization is $(W_V X)\mathrm{SoftMax}\left(\frac{X^T A X}{\sqrt{d_h}}\right)$

A key feature of the Transformer is that it can be parallelized!

Indeed, notice there was no sequential operations (as opposed to an RNN, say). If we choose n heads, then multi-head attention has the learnable parameters

$$\{W_Q^i, W_K^i, W_V^i\}_{i\in[n]}$$

Our final output is then a concatenation of all these outputs, so $\dim = nd_h$

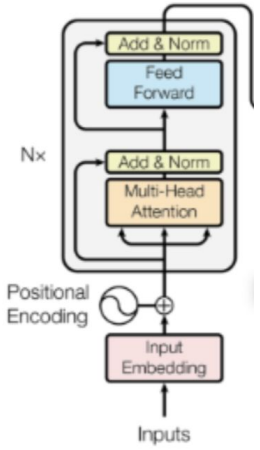
One small caveat

Right now, our mechanism is position-invariant.

- Rotated sentences {a,b,c,d} and {d,b,c,d} ultimately get sent to some rotation.
- We want to impose some positional information.
- We do essentially at pre-processing, we send $x_t \mapsto x_t + p_t$

Review of the procedure

- Step 1: Positional shift $X \mapsto X + P$
- Step 2: Do multi-head attention (with mask) $X \mapsto (W_V X)\mathrm{SoftMax}\left(\frac{X^T A X}{\sqrt{d_h}}\right)$
- Step 3: Apply a layer norm (to help with gradient flow): $X \mapsto \mathrm{LayerNorm}(X)$
- Step 4: Feed X into a FF Neural network $X \mapsto \mathrm{LayerNorm}(X + FF(X))$
  - Here, FF will consist of fully-connected layers and crucially non-linear activation functions (ReLU, leaky ReLU etc…)
- Repeat N times…



Outcomes

- Human language can be accurately understood by linear algebra and activation functions
- The compressed representations of language captured universal features that can be transferred to other tasks such as:
  - Use news data to predict stock price fluctuations
  - Spam prediction
  - Recommendation systems
  - Speech recognition
  - And more
- Fine-tuning: With low computation cost, we can optimize an LLM for new tasks
  - Chain of thought reasoning: We might feed in solutions to math problems with step-by-step proofs to teach an LLM to provide sequential reasoning

A word on the choice of optimizer

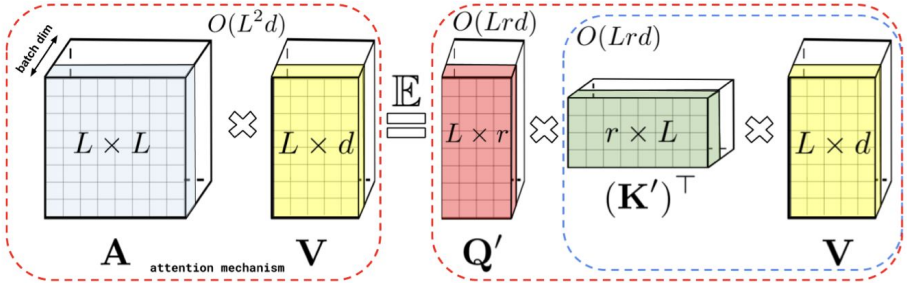
- The most widely used off-the-shelf optimizer seems to be ADAM
- Stands for Adaptive Moment Estimation
- Normal gradient descent has the form $\theta \mapsto \theta - \alpha \frac{\partial L}{\partial \theta}$
- ADAM has the form $\theta_{t+1} = \theta - \alpha \frac{m_t}{v_t + \epsilon}$
- Here, m_t keeps track of past gradients, weighted according to recency
- Then, v_t keeps track of past squares of gradients, according to recency
- In some sense, we're allowing for a dynamic modification of the learning rate

Computational Issues

Attention requires a O(V^2) computational complexity, and ideally we would like to speed this up. (See “Rethinking Attention with Performers”).

- Recall $A = W_K^T W_Q$
- Idea: Compute these dot-products in a smaller space, with some probabilistic guarantees; $y^t x \sim \mathbb{E}_w((w^t y)^t (w^t x))$
- Explicitly, the core-computational challenges is the output of the form

$$(W_K^T W_Q) W_V$$



Learnable Optimizers

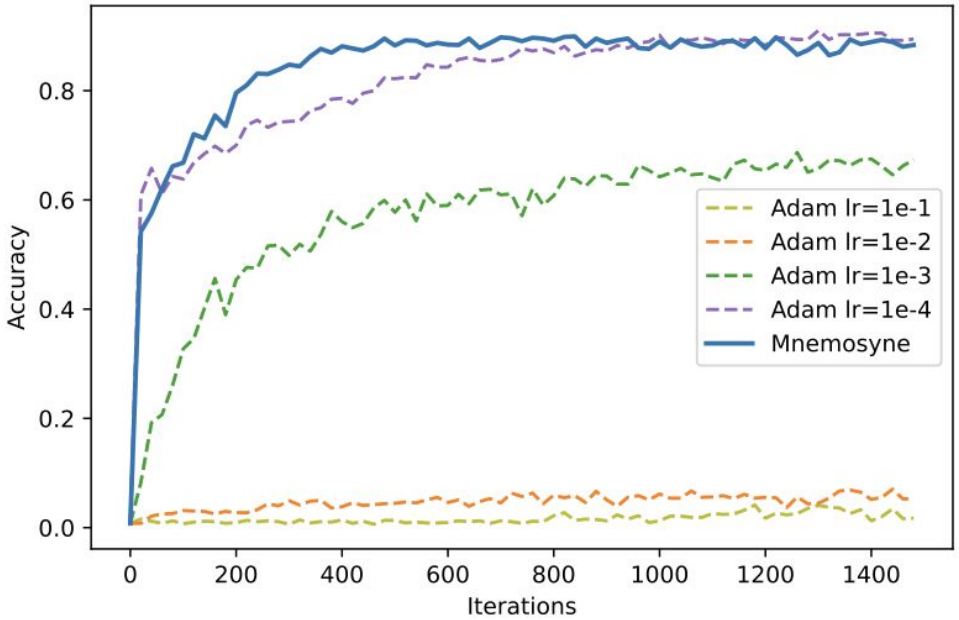
Question: Why ADAM, AdaBOOST, etc? Can we get a machine to learn how to optimize?

Answer: Yes! An active area of research right now.

Core idea: Recall ADAM incorporated a history of gradients. This is like modelling language, we let's use a transformer!

- Training data: A set of Tasks $\mathcal{T}$
- Implement a transformer $f_\theta(\text{gradients}_{t\in[T]}, \text{positions}_{t\in T})$ whose outputs at each time-step is an optimization protocol $\theta_{t+1} = \theta_t + f_\theta(\text{gradients}_{[t]}, \text{positions}_{[t]})$
- Loss function: Evaluate final output on the tasks at hand.
- Optimize the transformer (using ADAM !)

Example: Mnemosyne



Taken from “Learning to train Transformers with Transformers”

Application to mathematics

Given an ODE $x'(t) = f(x)$, we're interested to know if it is stable

This is hard. (See "Discovering Lyapunov functions with Transformers")

Lyapunov proved that if a certain function V(x) existed associated with the dynamical system, then the system is stable. Specifically, V has to be continuous, have continuous partial derivatives,is strictly positive outside of the origin and satisfies $-\nabla V \cdot f > 0$

Key: Given a Lyupanov function, can construct a f so that the ODE is stable.

Get data pairs (stable system, Lyupanov function).

Use a transformer to learn from a stable system the form of a Lyupanov function.

Fine-tuning

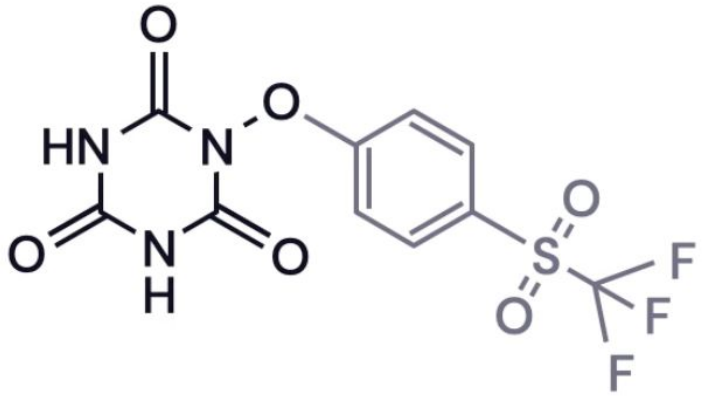
- OpenAI, Google, Facebook etc… did the hard yards for us.
- If we want to practically implement these GPT models for a task like spam prediction, we're already described one method
- Another method: LoRA
- Key idea: We want to perturb the attention matrices with low-rank, and optimize only on those matrices.
- Explicitly, $W \mapsto W + BA, A \in \mathbb{R}^{r \times d_v}, B \in \mathbb{R}^{d_h \times r}$
- The B's and A's in each layer are the only learnable parameters
- Upshot: Can optimize attention to the task at hand cheaply.

Graph Neural networks

In attention, every word's embedding depended on ALL the words before it (in the case of a decoder).

But what if our data came equipped with known temporal dependencies?

(Examples below taken from Petar Velickovic's talk, "Intro to graph neural networks")

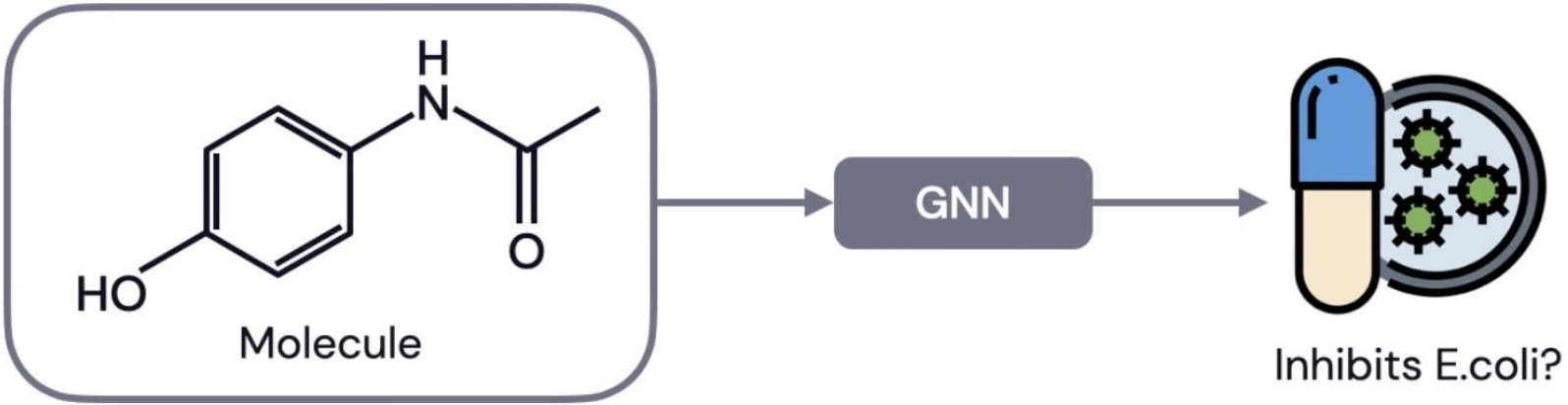


Graph Neural Networks

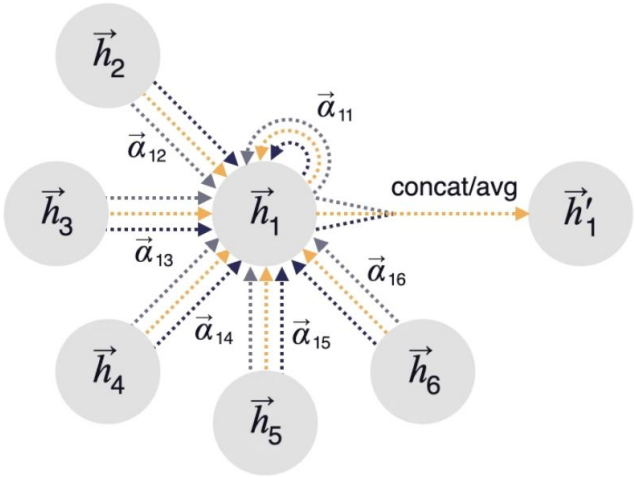
A very interesting application of Graph Neural networks has been to drug discovery.

We may be given a molecule, and want to determine whether it will lead to a potent drug.

(Examples below taken from Petar Velickovic's talk, "Intro to graph neural networks")



Graph Neural Networks



The attention coefficients are computed as follows:

$$a_{ij} = a(h_i, h_j, e_{ij})$$

$$\alpha_{ij} = \mathrm{Softmax}((a_{ij})_i)_j$$

$$h'_i \sim \sum_{j \in N_i} \alpha_{ij} W h_j$$

Key point: Transformers are in essence fully connected GNNs.

Comparison with dynamic programming