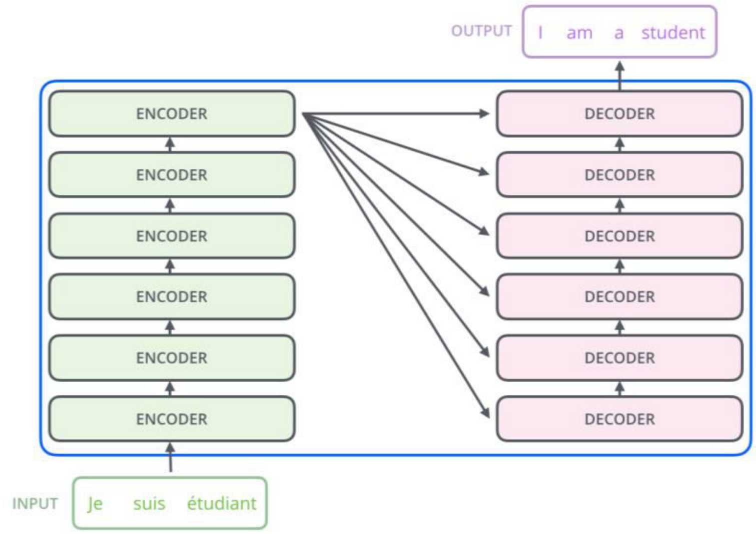
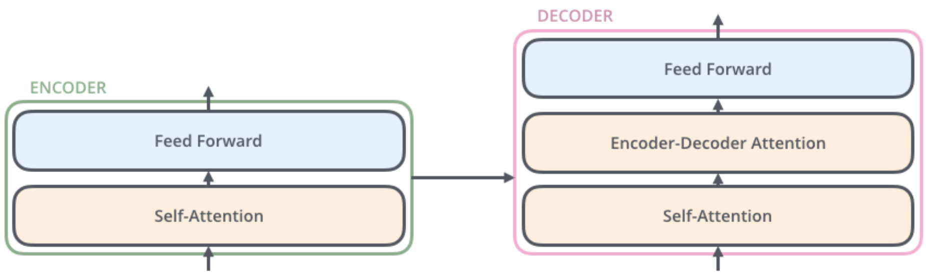
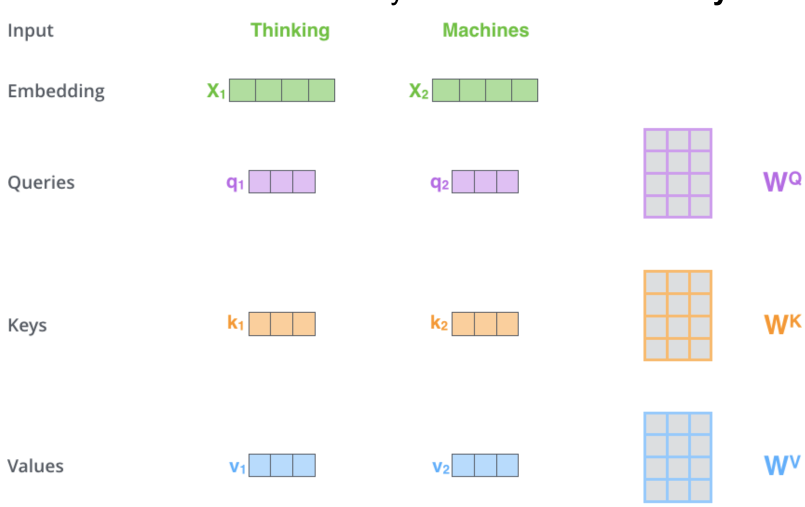




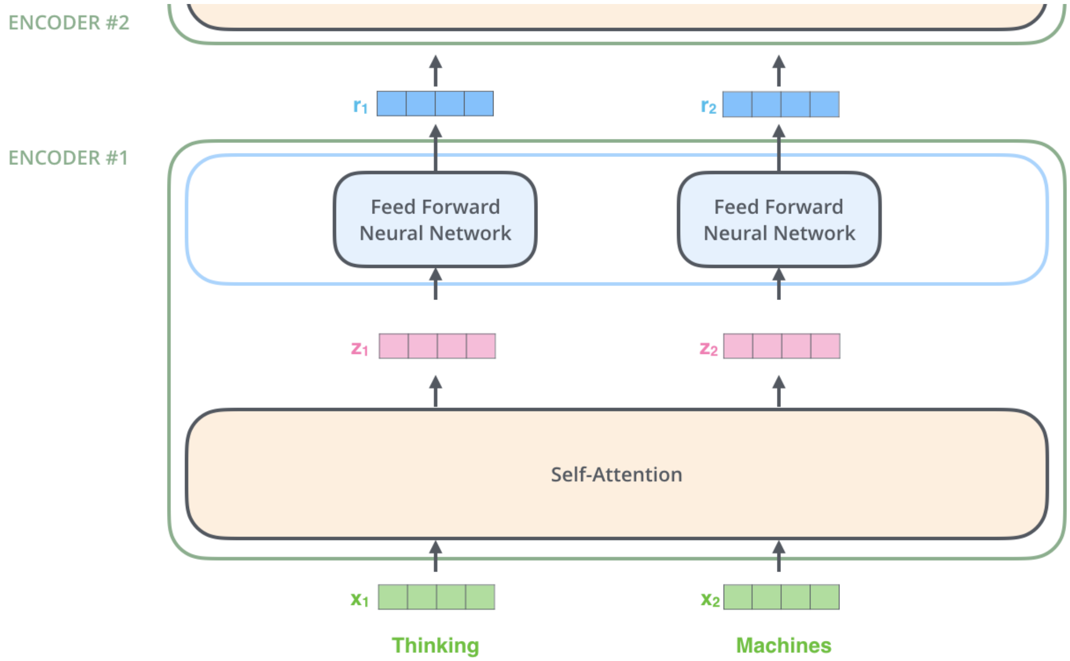
Encoder - Self-Attention Layer - helps encoder look at other words in the input sentence
Decoder - Encoder-Decoder Attention - a decoder layer that **helps the decoder focus on** relevant parts of **input**

Self-Attention - tells us which words in the input are related to which other words
- Self-attention requires us to compute **queries, keys,** and **values**
- This means we require to have **three weight matrices W^Q, W^K, W^V** that allow us to produce a separate **Q, K, V** for each embedding vector
  - **Q, K, V** tend to be of a smaller size than the embedding layer, typically **64**
- **Possible Issue -** Self-attention usually results in a word **mostly focused on itself**



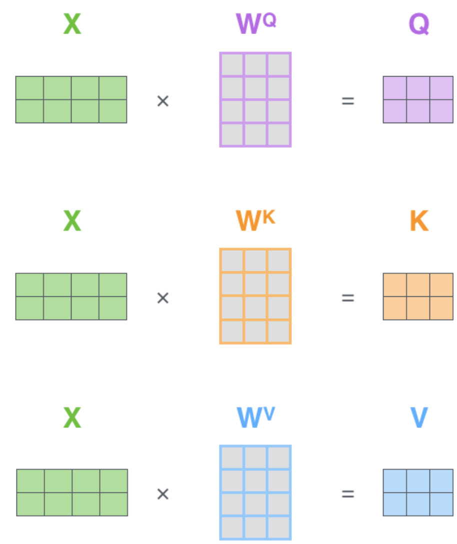
1. **One-hot vector encoding**
2. **Embedding -** Embedding translates a **ohv** into a a representation **of a smaller size** (512 / 768)
   - All **encoders** receive vectors of the same size
   - **Embedding size** is a **hyperparameter**
     - Usually the size of the **longest sentence in our corpus**
   - Each word is embedded separately
3. Each **embedding** (word) flows **through each encoder layer**

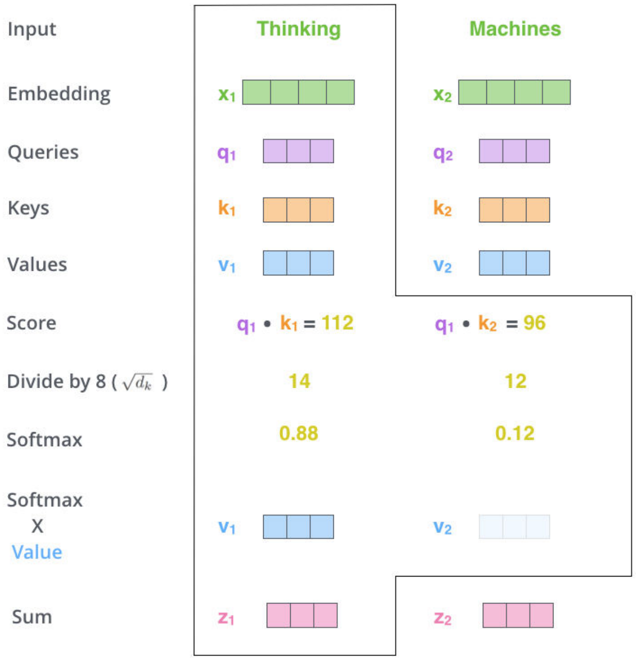
- There are **dependencies** between paths of different embeddings through the **self-attention layer**, but this is **not the case for the <u>feed forward layer</u>**, which can be hence done **in parallel**



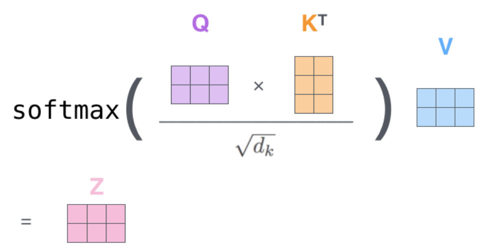
- *It's the same feed forward neural network for both embedding vectors*

4. For each word we create **Q, K, V** using 3 different **Weight matrices**



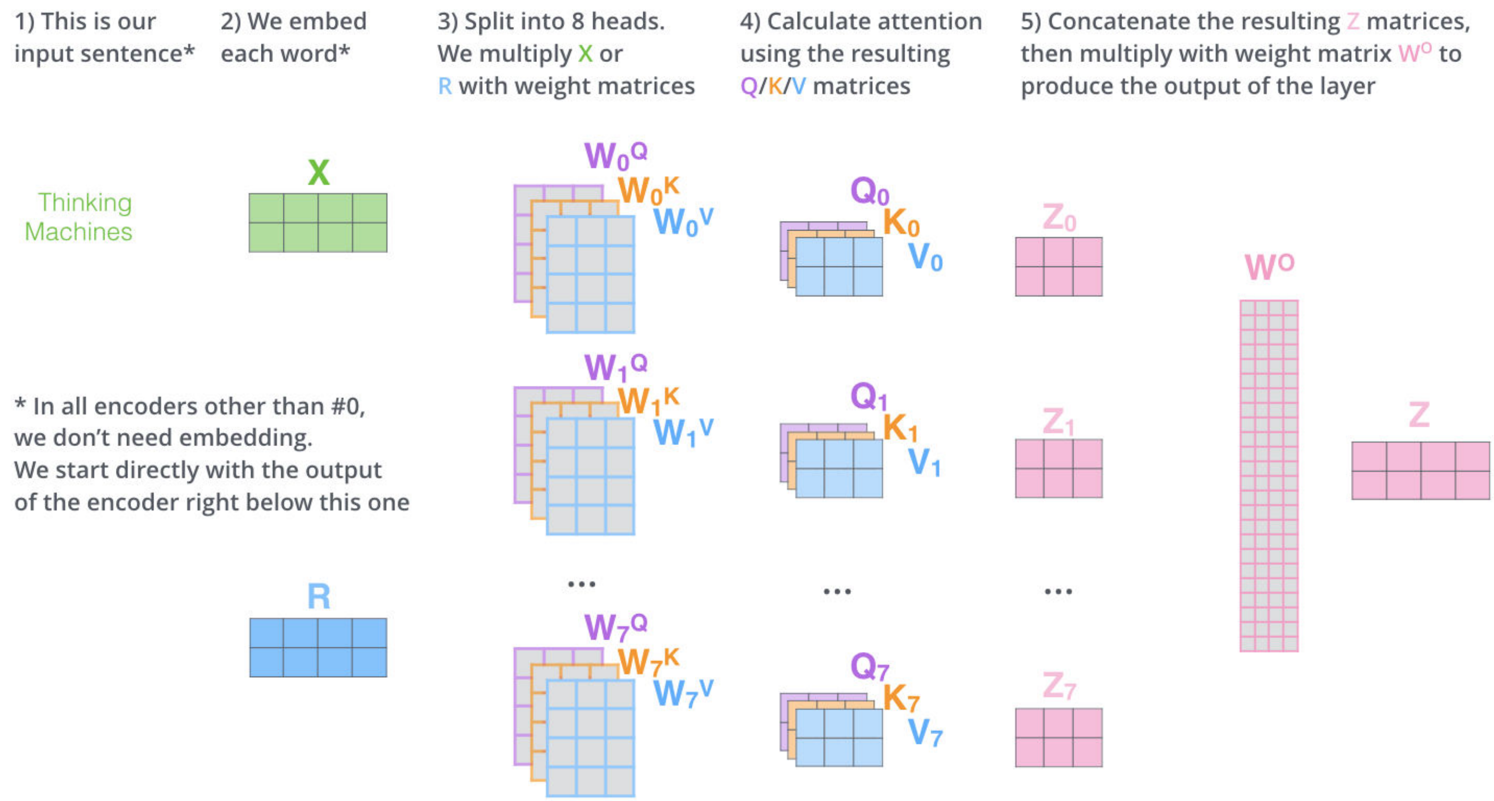


5. Divide the **score (q.k)** by **sqrt(len(key))**



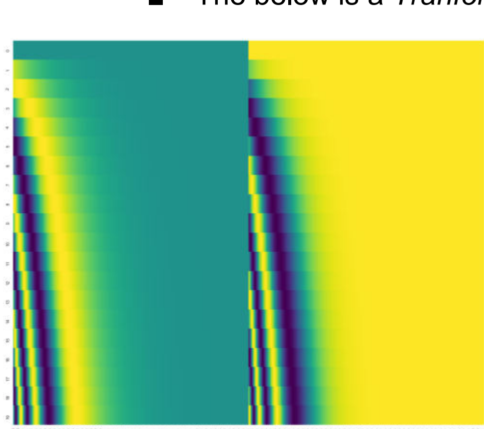
Multi-headed Attention

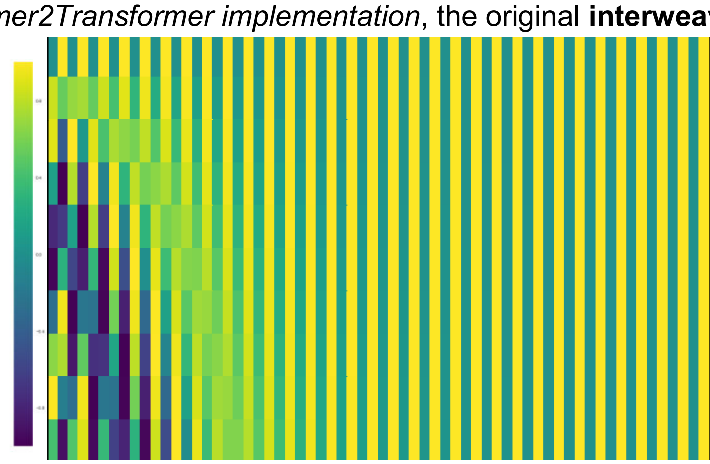
- Allows the word to focus **more on context** as opposed to itself
- Each multi-headed attention block has **numerous Q, K, V weight matrices**
    - Typically **8 different sets of weights** for each encoder/decoder
    - Allows for different **representation subspaces**
- This means that the output has **8** different **output matrices**
- As we want the output to be **same size as the input**, we **concatenate** and **multiply** by **final W^O**
    - Results in an output same dimension as the input



Positional Encoding

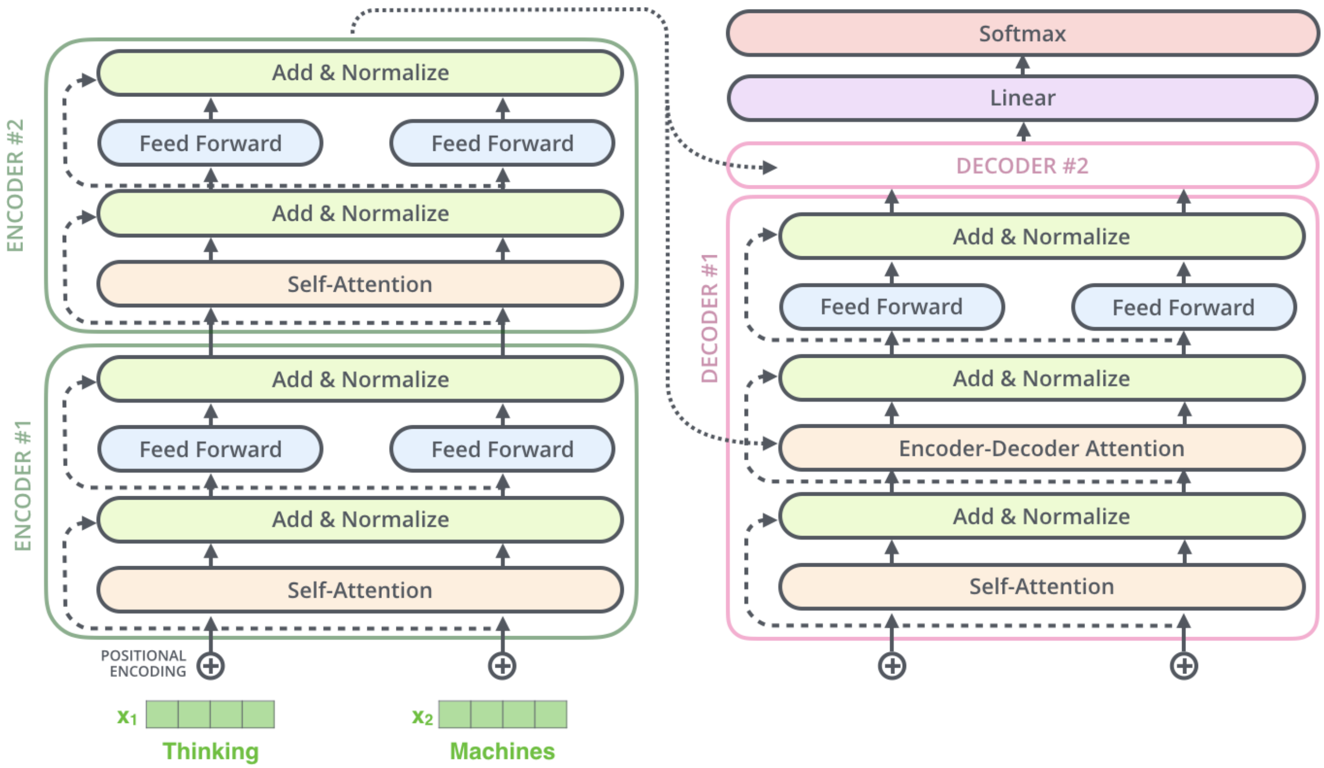
- **Positional Encoding** is used to hint the word order to the encoder
    - As all words are input at the same time (as if they were a set instead of a list), the ordering is lost
    - We **add positional vectors** to our **word embeddings**
    - Positional embeddings **follow a special pattern**, usually made out of **sinusoids**
    - *Each row below is a **512-size positional embedding vector** for each word in the input*
        - The below is a *Tranformer2Transformer implementation*, the original **interweaves** instead





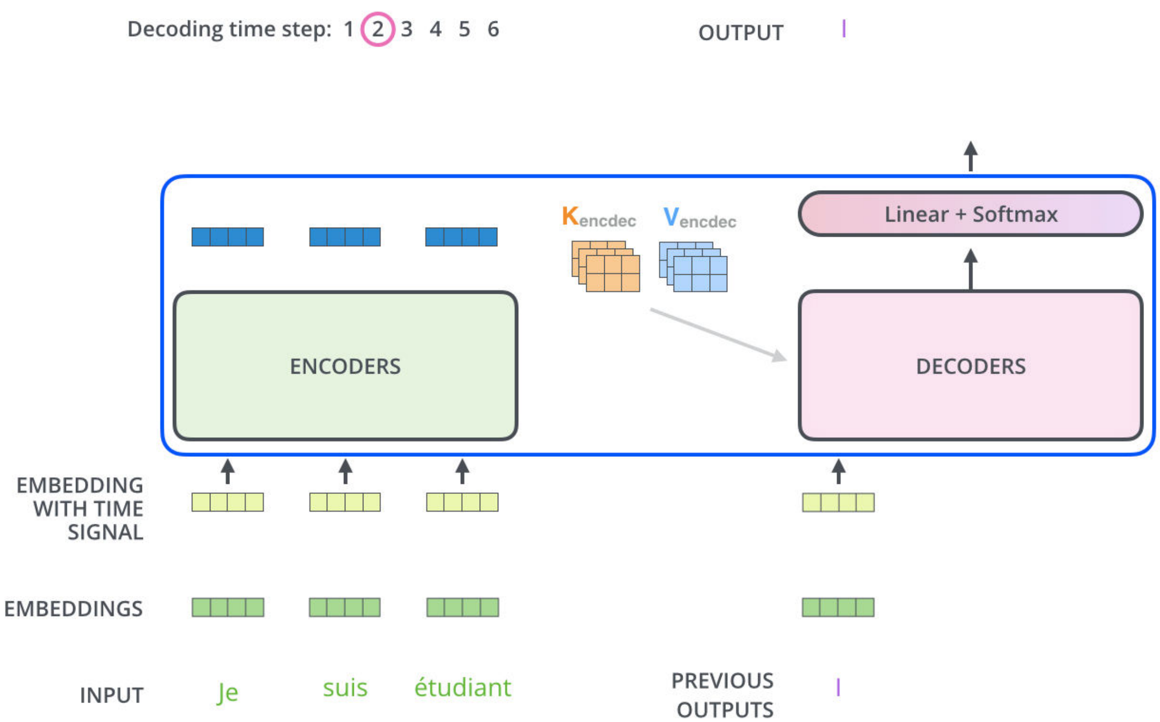
Residuals

- Each sublayer (**self-attention, fnn**) in an **encoder/decoder** has a **residual connection** around it, where the previous input vector is **added** to the output and then **normalized**



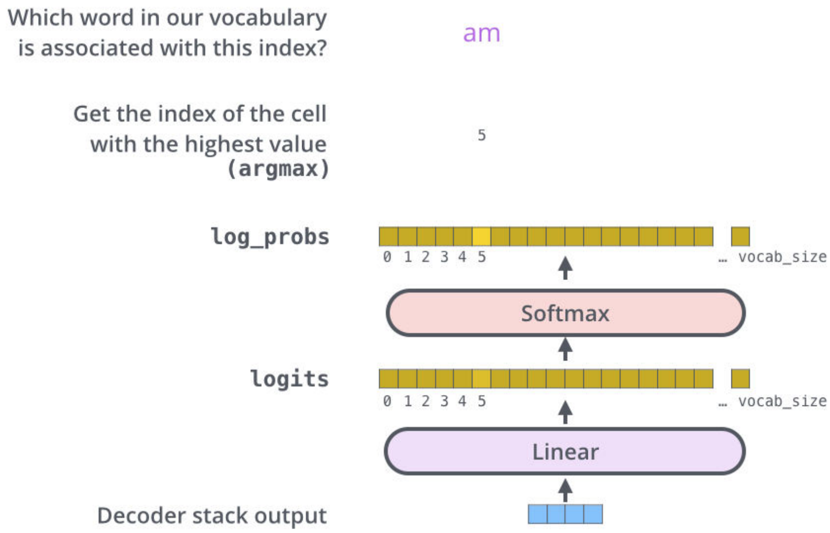
Decoders

- The final output of the **top encoder** is then used for **keys and values K, V** of **each decoder** in **EDA layer**
  - Allows the decoder to focus on the appropriate places of the input
- Each step of the decoding phase outputs **one output embedding**
- The output embedding is then used as the **input** in the **next** step
- **Decoder self-attention masks later** (future) **positions (with -inf)** before **softmax step**
- **EDA layer** works just like multiheaded self-attention
  - **Queries Q** generated from the layer below it, **Keys and Values K, V** from output of encoder



Final Linear and Softmax Layer

- Takes the output of the **last decoder** at **each time step**
- *Turns the output vector into a word*
- The **Final Linear** takes the **lower-dimensionality embedding** and converts it to a vector of size of **original ohv**
- It then uses **softmax** to get **probabilities** and then selects the word with the **highest value**



Training Recap

- We have a ohv label for each output word position
- **Loss function -** we use **cross-entropy** and **Kullback-Leibler divergence**
- We usually use a sentence as input and output
- We want to backpropagate over **every word**
- **Greedy encoding** - the model outputs **one (best) word at a time**, and **throws away the rest**
- **Beam Search -** We **branch** out **each time step**, by **keeping the top x words**
    - **Beam Size -** at all times, two partial hypotheses are kept in memory
    - **Top Beams -** how many translations do we return
    - *Hyperparameters we can optimise*