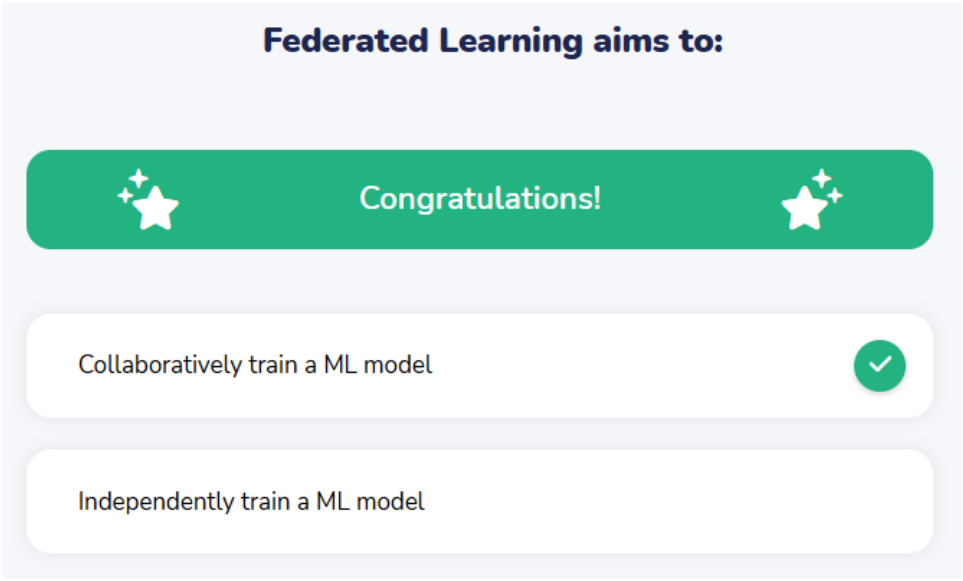
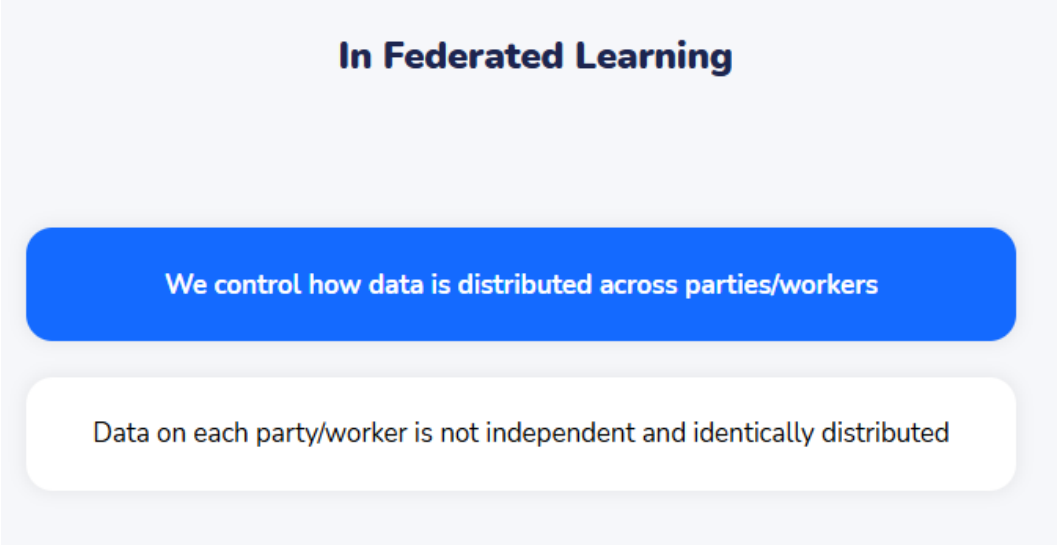
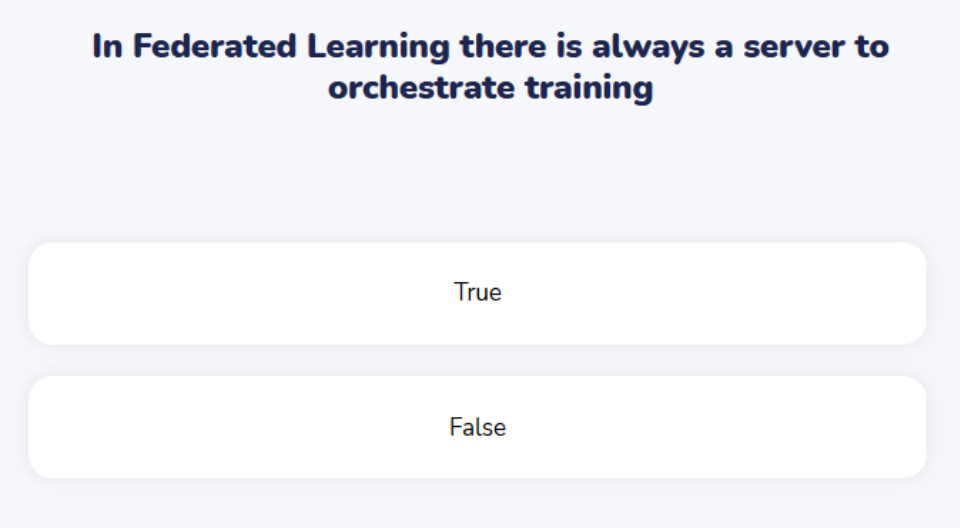
Lezione 13 - Transformers - 13/12/2024

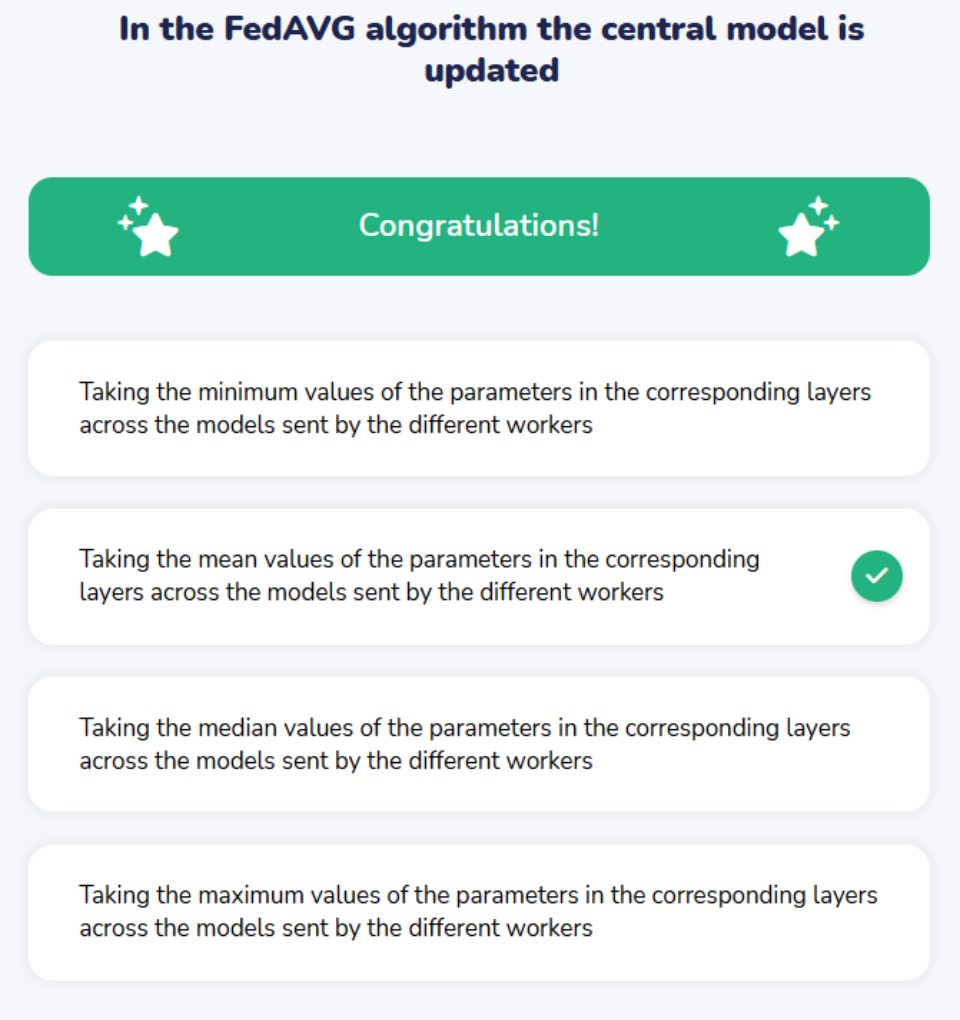




era l'altra quella giusta (e l'avevo fatta giusta, ma non ho fatto in tempo a fare lo screenshot)



False



Transformers

Abbiamo già introdotto le CNN:

1. Specializzate per elaborare dati che si trovano su una griglia regolare.
2. Particolarmente adatte all'elaborazione di immagini, che hanno un numero molto elevato di variabili di input (escludendo l'uso di reti completamente connesse).
3. Si comportano in modo simile in ogni posizione (condivisione dei parametri, ricordate?).

Poi abbiamo introdotto anche le recurrent NN per i dati sequenziali.

Oggi vediamo i **transformers**. Questa è un'architettura molto importante, è nata per il natural language processing, ma negli anni grazie al suo successo è stata estesa a molti tipi di dati, oggi sono lo standard anche per processare immagini e altri tipi di dati.

Questa transizione da linguaggio a immagini è stata possibile perchè i dataset condividono delle caratteristiche:

- la **grande quantità di variabili** in input
- Poi le **statistiche sono simili** ad ogni posizione, un cane è sempre un cane, che la parola sia all'inizio, in mezzo o alla fine della frase, esattamente come è sempre un cane in ogni posizione dell'immagine.

I **linguaggi** hanno una complicazione in più perché le **sequenze di input** hanno nativamente **lunghezze diverse**, e a differenza delle immagini non possiamo fare un resize in modo semplice come con le immagini

Consideriamo questo testo:

> *"The restaurant refused to serve me a ham sandwich because it only cooks vegetarian food. In the end, they just gave me two slices of bread. Their ambiance was just as good as the food and service."*

Per esempio vorremmo classificare questa review come positive negative, oppure l'utente potrebbe essere interessato a sapere se il ristorante serve carne.

L'**encoded input** può essere molto largo. Ogni parola è mappata in un **embedding**, estraendo delle **features rappresentative**. Questo spazio di embedding ha delle proprietà, tipo il fatto che topics simili sono vicini nei clusters mentre topics lontani sono lontani nei clusters.

Se usiamo un 1024-d embedding, il testo aveva 37 parole, 37×1024=37888 di input length, che è già molto per un testo di input corto. Si può immaginare come usare reti fully connected non sia un'opzione.

Una caratteristica dei problemi NLP è che gli input hanno **lunghezze diverse**. Questo suggerisce che la NN dovrebbe condividere parametri tra la stessa parola messa in diverse posizioni di input (un po' come le CNN condividono parametri tra diverse posizioni dell'immagine)

Inoltre, **il linguaggio è fondamentalmente ambiguo**. Per esempio dal testo precedente non siamo sicuri se si stia riferendo al ristorante o al panino.

> *"The restaurant refused to serve me a ham sandwich because **it** only cooks vegetarian food. In the end, they just gave me two slices of bread. Their ambiance was just as good as the food and service."*

Quindi non possiamo capire il contesto usando solamente la sintassi.

Abbiamo bisogno un **modello** per processare il testo, che:

- usa **parameter sharing** per risolvere il fatto che potremmo avere un testo molto lungo e di lunghezza diversa, di modo che gli stessi pesi siano applicati a tutti gli input.
- conterrà connessioni tra rappresentazioni delle parole che dipendono dalle parole stesse, ovvero la **rappresentazione** (o "embedding") di una parola in una frase deve **dipendere dal contesto** fornito dalle altre parole nella stessa frase..

I **transformers** acquisiscono entrambe queste proprietà usando il **dot-product self-attention**.

Dot-product self-attention

A standard NN layer $f[x]$ takes a $D\times1$ input x and applies a linear transformation followed by a non-linear activation function $a[\cdot]$:

$$f[x] = a[\beta + \Omega x]$$

where β contains the biases and Ω contains the weights.

A self-attention block $sa[\cdot]$ takes N inputs x_n, each of dimension $D\times1$, and returns N output vectors of the same size.

Quindi prende N input e ritorna un output che ha sempre la stessa dimensionalità.

Nel contesto di NLP ogni input x_n rappresenta una parola o un frammento di una parola.

Values

Innanzitutto, viene calcolato un set di **Values** per ciascun input.

$$v_n = \beta_v + \Omega_v x_n$$

Abbiamo una matrice di pesi Ω_v e una matrice di bias β_v. Otteniamo in output i valori v_n.

La **self attention** è la somma pesata moltiplicata dei valori v_n.

$$sa[x_n] = \sum_{m=1}^{N} a[x_m, x_n]\, v_m$$

In particolare, i pesi scalari $a[x_m, x_n]$, sono l'**attention che l'input x_n ha nei confronti dell'input x_m**. Questa è l'informazione di quanto sono collegati gli input. Se sono correlati avranno un valore alto, se sono completamente scorrelati avranno un peso basso.

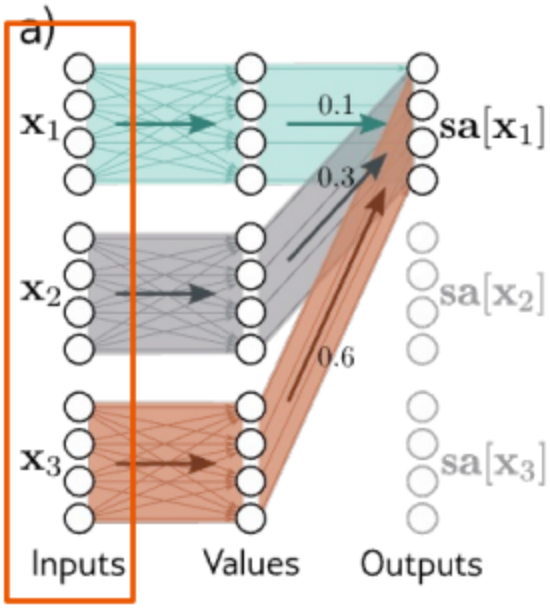
In particolare, abbiamo che **ciascuna colonna della matrice di pesi è non negativa e somma ad 1**, perchè ci chiediamo quanta attention stiamo dando a tutti gli altri input, che somma a 1, e l'attenzione non può essere negativa, quindi ci dice quanto ciascun input da attention agli altri input.

La **self-attention** può essere considerata come il **routing** dei valori in proporzioni diverse per creare ciascun output. Ovvero, la self- attention consente al modello di determinare quanto ciascun valore contribuisca all'output finale di una parola specifica $sa[x_n]$, basandosi sulle relazioni tra tutte le parole nella sequenza.

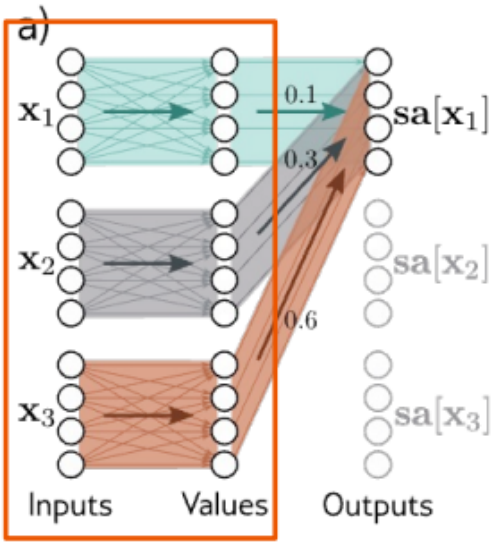
Vediamo visualmente cosa sta succedendo.

Abbiamo i nostri input x1 ×2 ×3 ciascuno di dimensione 4×1.

Calcoliamo i **Values**, che hanno la stessa grandezza dell'input, e sono lo stessa quantità dell'input. Sono collegati con un layer fully connected.

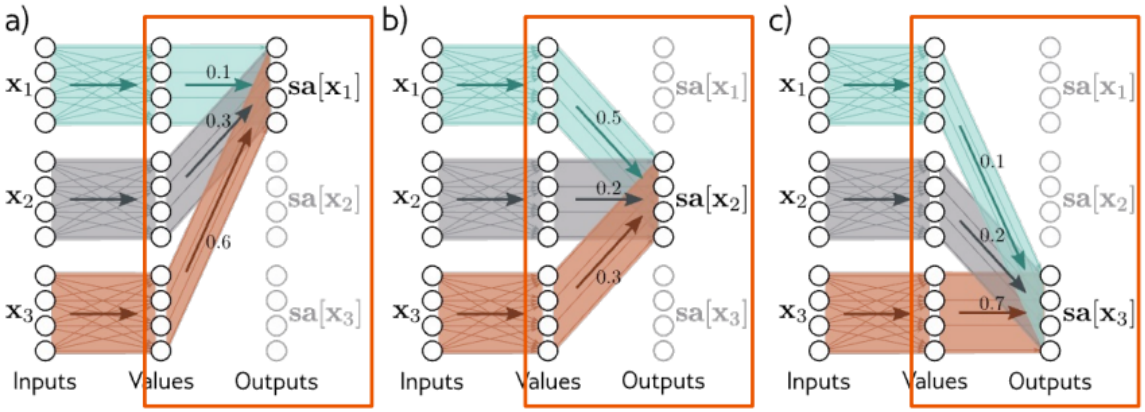


In particolare, **ciascuno è calcolato separatamente**, quindi non stiamo mixando le informazioni degli input per calcolarli.

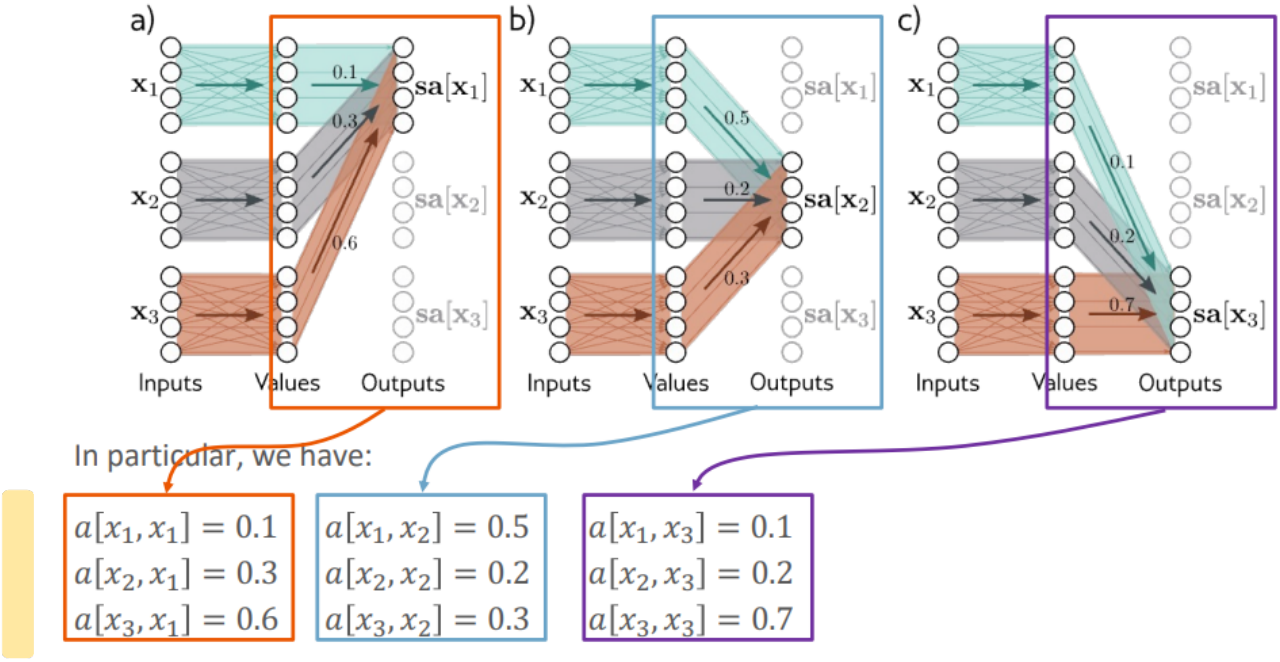


è invece quando stiamo calcolando la **self attention**, che mixiamo i diversi valori.

Per esempio l'output delle self attention per x1 è 0.1 rispetto a x1, 0.3 rispetto a x2, 0.6 rispetto a x3. Vediamo che sono positivi e sommano a 1.



Quindi la matrice self attention ha questi valori:



La prima è l'attenzione che l'input 1 da all'input 3, 0.6.

Calcolo e peso dei Values

Per calcolare i **Values**, gli **stessi pesi** Ω_v (con grandezza DxD) e gli **stessi bias** β_v (con grandezza Dx1) sono applicati a tutti gli input x_n (con grandezza Dx1) della nostra sequenza. Quindi ciascun Value ha la stessa grandezza dell'input.

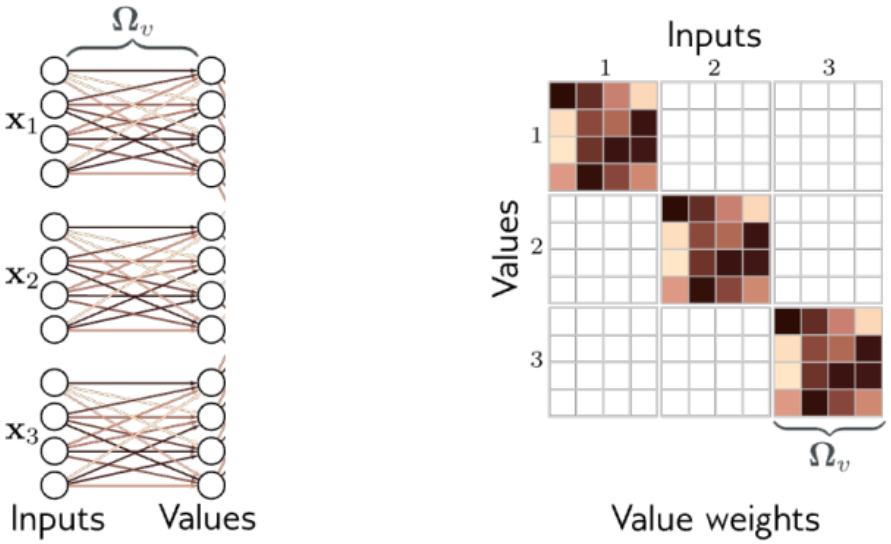
$$v_n = \beta_v + \Omega_v x_n$$

Questa computazione scala linearmente con la grandezza della sequenza di input N, quindi richiede **meno parametri** di una NN fully connected, che collega tutti gli input a tutti gli output.

Il calcolo dei valori può quindi essere visto come una moltiplicazione di matrici sparse con parametri condivisi.

Questo è quello che succede.

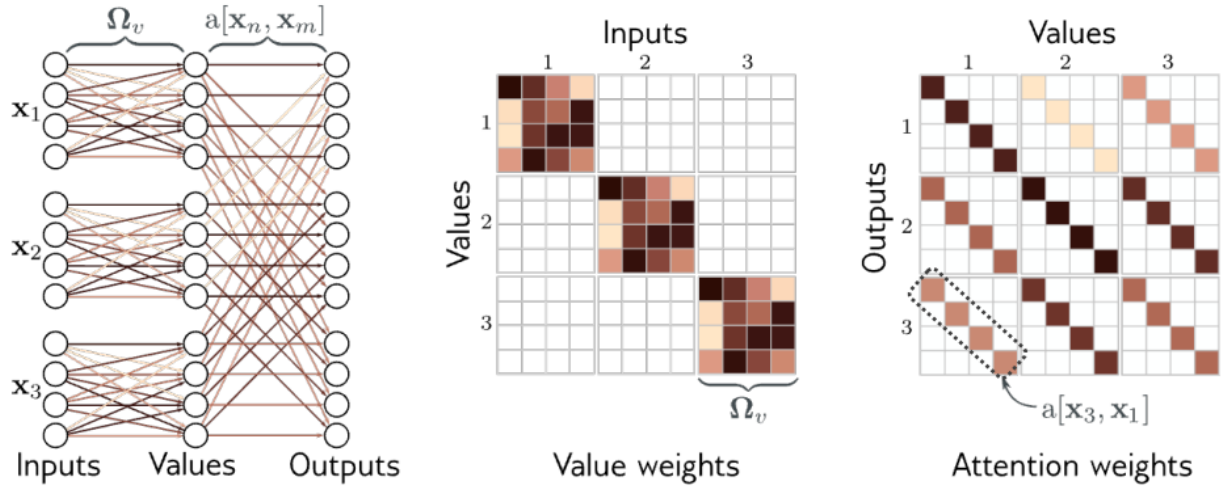
Gli stessi pesi sono usati poi anche per l'input 2 e per l'input 3 per ottenere i valori corrispondenti.



Quindi qui stiamo usando 4×4 pesi, rispetto ai 12×12 che useremo in una NN fully connected.

I pesi dell'**attention** $a[x_n, x_m]$ **combinano** i valori di diversi input. Sono anche sparsi, siccome c'è solo un peso per ciascuna coppia di input (x_m, x_n), a prescindere dalla grandezza degli input.

Questo è come possiamo implementare la computazione.



Il numero di pesi dell'attention ha una dipendenza quadratica rispetto alla lunghezza della sequenza N, ma è indipendente dalla lunghezza D di ciascun input.

Sostanzialmente, abbiamo visto che gli outputs sono il risultato di **2 trasformazioni lineari**:

- la prima, **dagli inputs ai Values**
- la seconda, combinazione lineare dei **values sui pesi dell'attention**

La self attention è però non lineare, per colpa di come gli attention weights sono creati (sono funzioni non lineari dell'input).

Quindi alla fine è un'operazione non lineare.

Questo è un caso di **hypernetwork**, dove c'è una parte della rete c'è la computazione dei parametri che un'altra parte della rete userà.

Calcolo degli attention weights

Per calcolare l'**attention**, applichiamo altre 2 trasformazioni lineari dell'input. Abbiamo 2 nuove matrici di pesi e 2 biases. C'è q e k perchè l'output è chiamato **queries** e **keys**.

$$q_n = \beta_q + \Omega_q x_n$$
$$k_n = \beta_k + \Omega_k x_n$$

Ciascuna entry della self attention è calcolata come il prodotto scalare tra le queries e le keys seguite da softmax:

$$a[x_m, x_n] = softmax_m[k_*^T q_n] = \frac{\exp[k_m^T q_n]}{\sum_{m'=1}^{N} \exp[k_{m'}^T q_n]}$$

Quindi alla fine ciascun 'a' è una trasformazione non lineare dell'input.

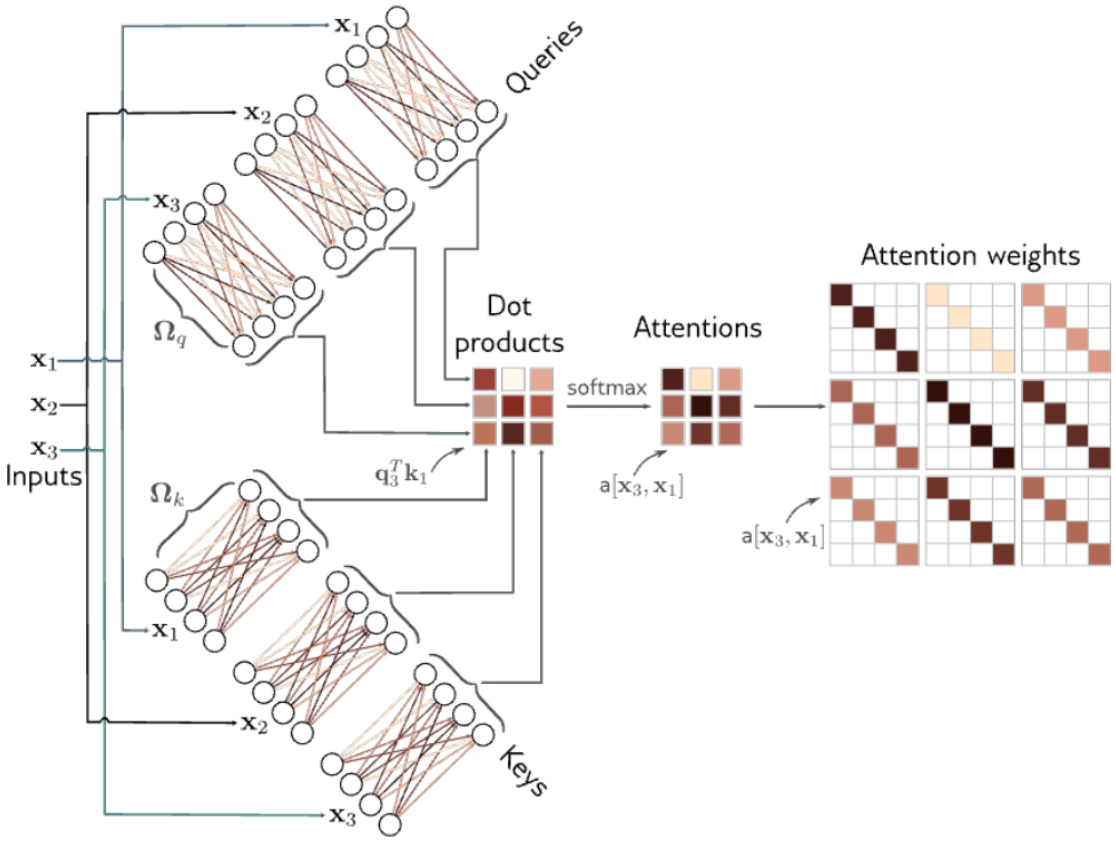
Quindi il fatto che gli 'a' sono positivi e sommano a 1 è dato dalla softmax.

Questa è chiamata la **dot-product self-attention**.

Vediamo come è computata.

Abbiamo gli input a sinistra, e poi abbiamo la matrice Ω_q che viene applicata a tutti gli input (la stessa matrice) per creare le queries, e poi abbiamo l'altra matrice Ω_k (quella delle keys) che è applicata indipendentemente a tutti gli input, generando le keys, che hanno la stessa grandezza e cardinalità degli input.

Poi combinando per esempio le queries 1 con le keys1 otteniamo il primo valore della self attention matrix. Su questo applichiamo la softmax e troviamo la matrice finale, che può essere vista nella full matrix representation dove ogni diagonale ha lo stesso peso.



I nomi **queries e keys** arrivano dall'information retrieval, e hanno la seguente **interpretazione**:

- Il prodotto scalare ritorna una misura di similarità tra gli inputs, quindi i pesi $a[x_*, x_n]$ dipendono nelle similatità relative tra ciascuna query e le chiavi.
- La funziona softmax significa che possiamo pensare ai vettori keys come "in competizione" l'un l'altro, per contribuire al risultato finale.

In particolare le queries e keys devono avere la stessa dimensione, altrimenti non possiamo fare il prodotto scalare.

Però, la **dimensione** di **queries** e **keys** può essere diversa da quelle dei **values**, che è di solito la stessa grandezza dell'input di modo che la rappresentazione non cambi dimensione. Vedremo tra poco l'ultima frase cosa significa.

Self-attention summary

> L'n-esmo output è la somma pesata della stessa trasformazione linare che è applicata a tutti gli input, con gli attention weights che sono positivi e sommano a

1.

I pesi dipendono da una misura di similarità tra gli input x_n e gli altri input.

Non c'è una funzione di attivazione, ma il meccanismo è comunque **non lineare** grazie al prodotto scalare e alla operazione softmax usata per calcolare gli **attention weights**.

Questo meccanismo soddisfa i requisiti iniziali:

- C'è un singolo set di parametri condivisi, che è indipendente dal numero di input N, quindi la rete può essere applicata a sequenze di lunghezza diversa.
- Ci sono connessioni tra gli input (words) e la forza di queste connessioni dipende dagli input stessi, attraverso gli attention weights.

Possiamo riscrivere tutto quello che abbiamo fatto fin ora in una forma più compatta.

I values, queries e keys possono essere calcolati come:

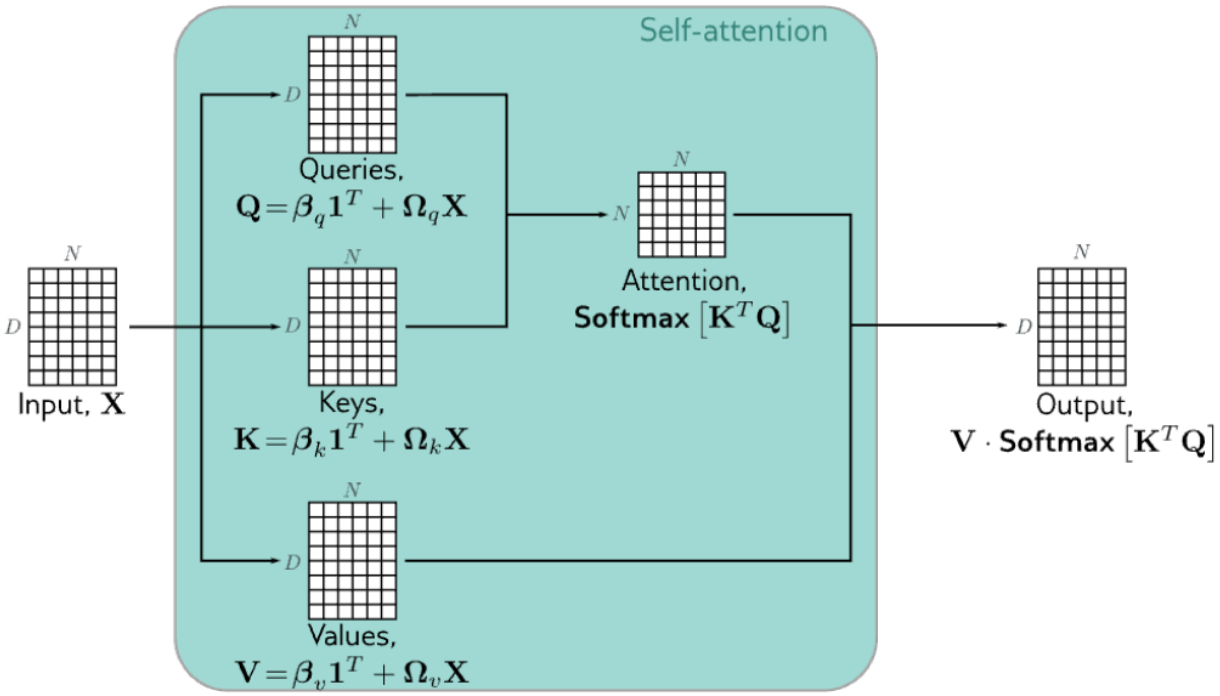
$$V[X] = \beta_v 1^T + \Omega_v X$$
$$Q[X] = \beta_q 1^T + \Omega_q X$$
$$K[X] = \beta_k 1^T + \Omega_k X$$

E la self-attention diventa:

$$Sa[X] = V[X] \cdot Softmax[K[X]^T Q[X]] = V \cdot Softmax[K^T Q]$$

dove softmax è applicata indipendentemente alle colonne dei suoi input.

Visualmente questo è quello che fa:



Abbiamo l'input X, abbiamo la computazione dei Values (stessa grandezza degli input), Keys e Queries (in questo caso stessa grandezza dell'input ma non è obbligatorio).

Estensioni del dot-product self attention

Positional encoding

In quello visto fin ora, la self attention elimina delle informazioni importanti, dato che la computazione è la stessa a prescindere dell'ordine dell'input, che però è importante quando l'input corrisponde a parole in una frase.

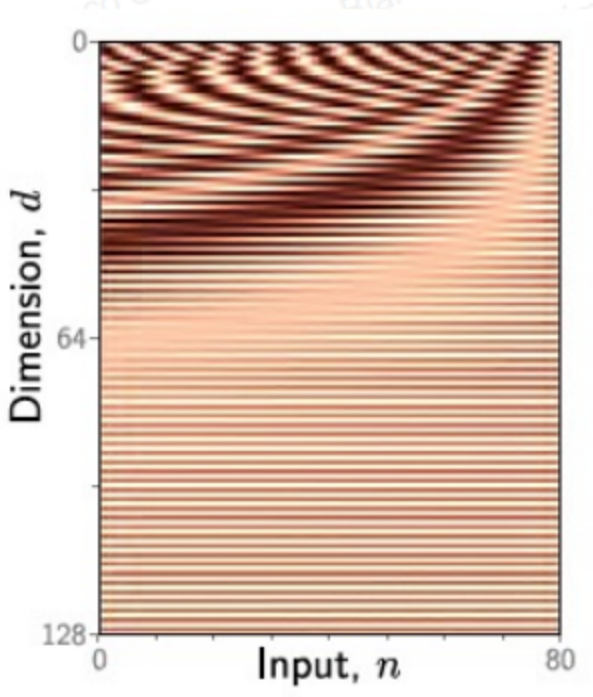
E.g., the sentence *The woman ate the raccoon* has a different meaning than *The raccoon ate the woman*.

Abbiamo due approcci per incorporare l'informazione posizionale:

- Absolute position embeddings
- Relative position embeddings

In **absolute position embedding** una matrice Π che codifica l'informazione posizionale è sommata agli input X. Ciascuna colonna è diversa e contiene informazioni relative alla posizione assoluta nella sequenza di input.

Questa matrice può essere scelta a mano (per esempio un pattern sinusoidale) o imparata. Ha la stessa dimensione dell'input. Può essere sommata agli input delle rete oppure ad ogni layer della rete. A volte è sommata a X nella computazione delle queries e keys, ma non per i values.



Nel **relative positional embeddings** invece, è utile dato che nei casi dove abbiamo solo una parte della frase, avere le posizioni assolute non ha senso, è sufficiente avere la posizione relativa.

L'input del meccanismo di self-attention può essere una frase, multiple frasi, o un fragmento di una frase. Qui la posizione assoluta di una parola è molto meno importante di quella relativa tra due input.

Ciascun elemento della matrice delle attention corrisponde ad un particolare offset tra la query in posizione a e la key in posizione b.

Relative position embeddings imparano un parametro $\pi_{a,b}$ per ciascun offset e lo usano per modificare l'attention matrix sommando per questi valori, moltiplicando

per questi valori, oppure usandoli per alterare la attention matrix in altri modi.

Scaled dot product self attention

I prodotti scalari nel calcolo dell'attenzione possono avere magnitudes elevate e spostare gli argomenti della funzione softmax in una regione in cui il valore più grande domina completamente. Di conseguenza piccole variazioni degli input alla funzione softmax hanno un effetto minimo sull'output (cioè, i gradienti diventano molto piccoli), rendendo difficile l'addestramento del modello.

Per prevenire questo, i prodotti scalari vengono **scalati** con la radice quadrata della dimensione D_q delle query e delle chiavi (cioè per esempio, il numero di righe in Ω_q e Ω_k, che devono essere uguali).

$$Sa[X] = V \cdot Softmax\left[\frac{K^T Q}{\sqrt{D_q}}\right]$$

Multiple heads

Di solito multipli meccanismi di self-attention sono applicati in parallelo, questo si chiama **multi-head self-attention**.

Quindi ora H diversi sets di values, keys e queries sono calcolati:

$$V_h[X] = \beta_{vh}1^T + \Omega_{vh}X$$
$$Q_h[X] = \beta_{qh}1^T + \Omega_{qh}X$$
$$K_h[X] = \beta_{kh}1^T + \Omega_{kh}X$$

L'h-esimo meccanismo di self-attention (o **head**) può essere scritto come:

$$Sa_h[X] = V_h \cdot Softmax\left[\frac{K_h{}^T Q_h}{\sqrt{D_q}}\right]$$

dove abbiamo diversi parametri per ciascun head.

Typically, if the dimension of the inputs is $x_m$is D and there are H heads, the values, queries, and keys will all be of size D/H as this allows for an efficient implementation.

Gli outputs di questi meccanismi di self attention sono concatenati verticalmente, e un'altra trasformazione lineare Ω_c è applicata per combinarli:

$$MhSa[X] = \Omega_c[Sa_1[X]^T, Sa_2[X]^T, \cdots, Sa_H[X]^T]$$

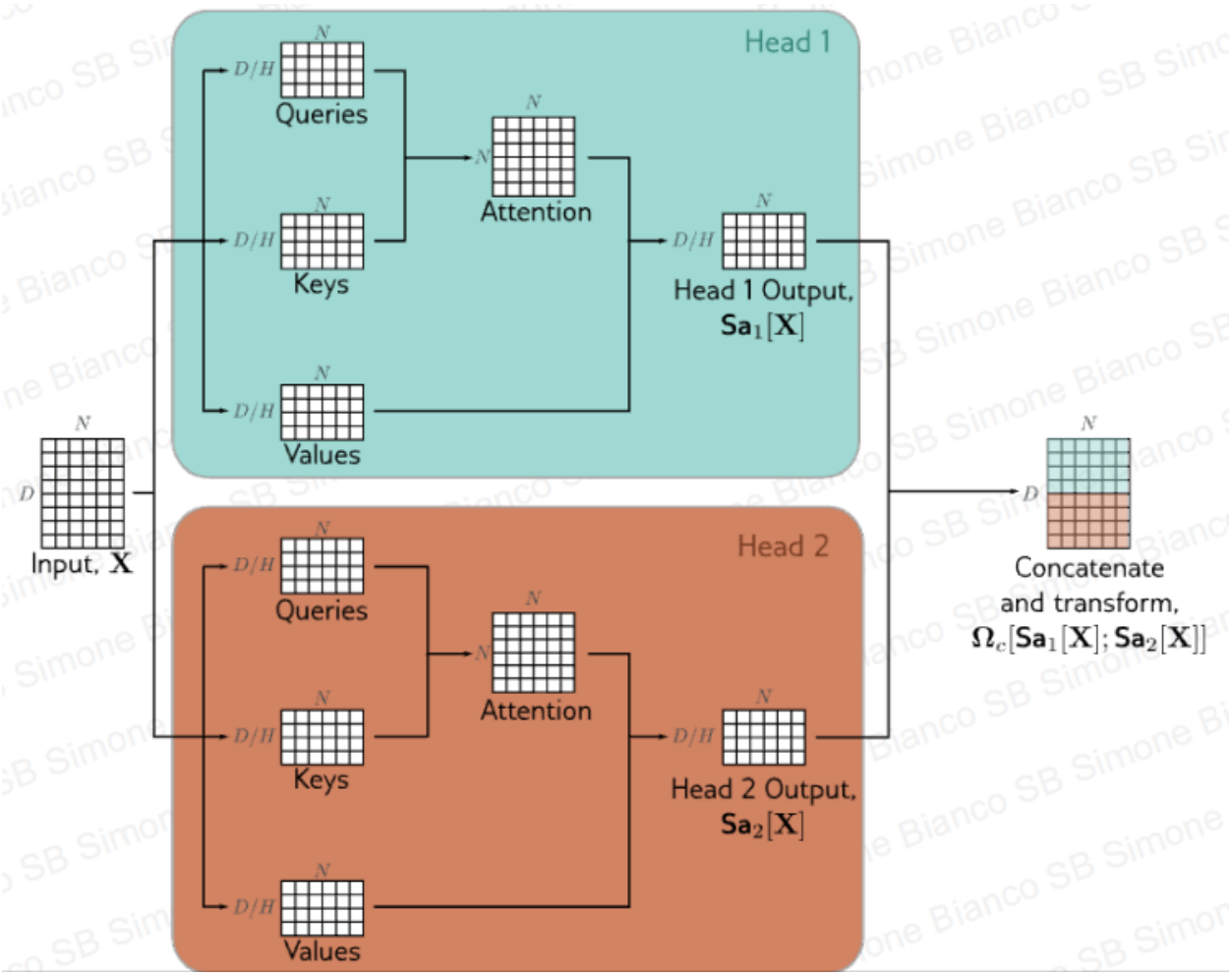
L'uso di multiple heads sembra sia necessario per fare funzionare bene il transformer. Si dice che rendano il meccanismo di self-attention della rete più robusto alle inizializzazioni brutte.

Vediamo cosa succede visualmente.

Abbiamo l'input X, N è il numero di input, quindi qua sono 6 words... non ho capito cos'è D

Usiamo 2 heads. Le queries keys e values, consideriamo per l'embedding la grandezza originale D diviso per...

L'output è quello della prima head combinato a quello della seconda head, su cui applichiamo on top una trasformazione Ω_c. Grazie alla trasformazione, se per esempio un head ha una brutta inizializzazione, la trasformazione può mettere quella parte a 0.



Poi può essere che ciascun output dia valori su range diversi. Quindi se la magnitude di uno è molto più grande, questo avrà un'importanza più grande. La trasformazione viene utilizzata per calibrare questa cosa, per portarli nello stesso range di valori.

Transformer layers

Fin ora abbiamo visto la self-attention, che è una sola parte di un transformer layer più grande.

In particolare, un **transformer layer** è composto di:

- un **multi-head transformation unit** (per capire le relazioni tra le parole)
- collegato ad un **fully connected network** (mlp) che opera su ciascuna parola separatamente

Tipicamente viene aggiunto anche un'operazione **LayerNorm** dopo sia il blocco self-attention che dopo quello fully connected (è simile a BatchNorm, ma usa statistiche tra i tokens di una sequenza di input per fare la normalizzazione).

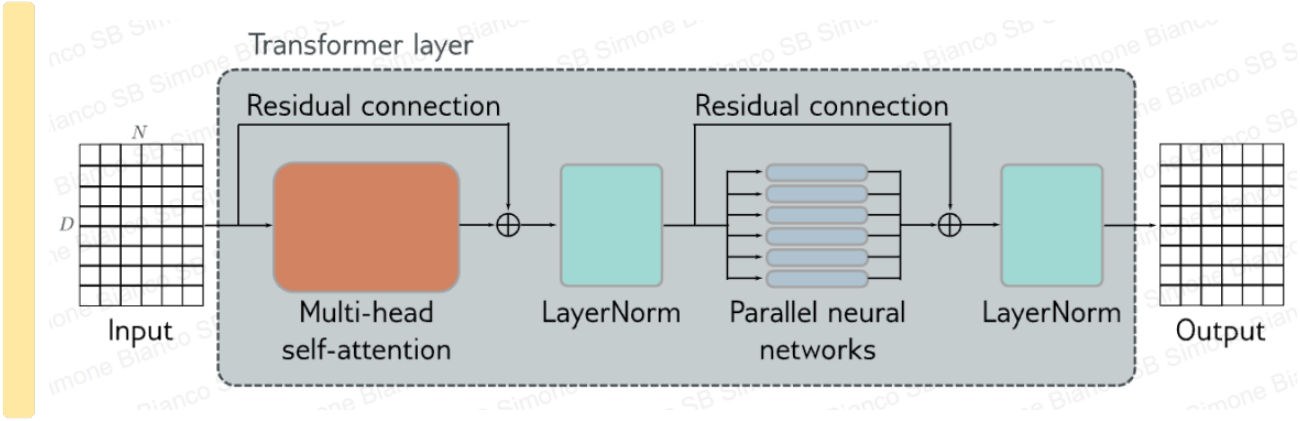
In una rete reale, i dati passano per una serie di questi layers.

Quindi abbiamo l'input, il multi-head self attention. In particolare abbiamo detto che l'input e l'ouput del multi-head self attention ha la stessa grandezza, e questo è importante perchè altrimenti non potremmo fare la residual connection.

Poi abbiamo il LayerNorm.

Poi abbiamo un layer fully connected che performa computazioni su ciascuna parola parallelamente.

Poi c'è un altro LayerNorm e otteniamo l'output.



Come possiamo permettere di avere un numero infinito di questi layers uno dopo l'altro? Se ci assicuriamo che l'input abbia la stessa grandezza dell'output.

Quindi questo è il transformer layer. Le **residual connection** permettono al gradiente di tornare indietro senza cambiamenti, il che aiuta per il learning.

Transformer per NLP

Fin ora, abbiamo descritto il **transformer block**. Ora vediamo come è usato nelle task di **natural language processing (NLP)**.

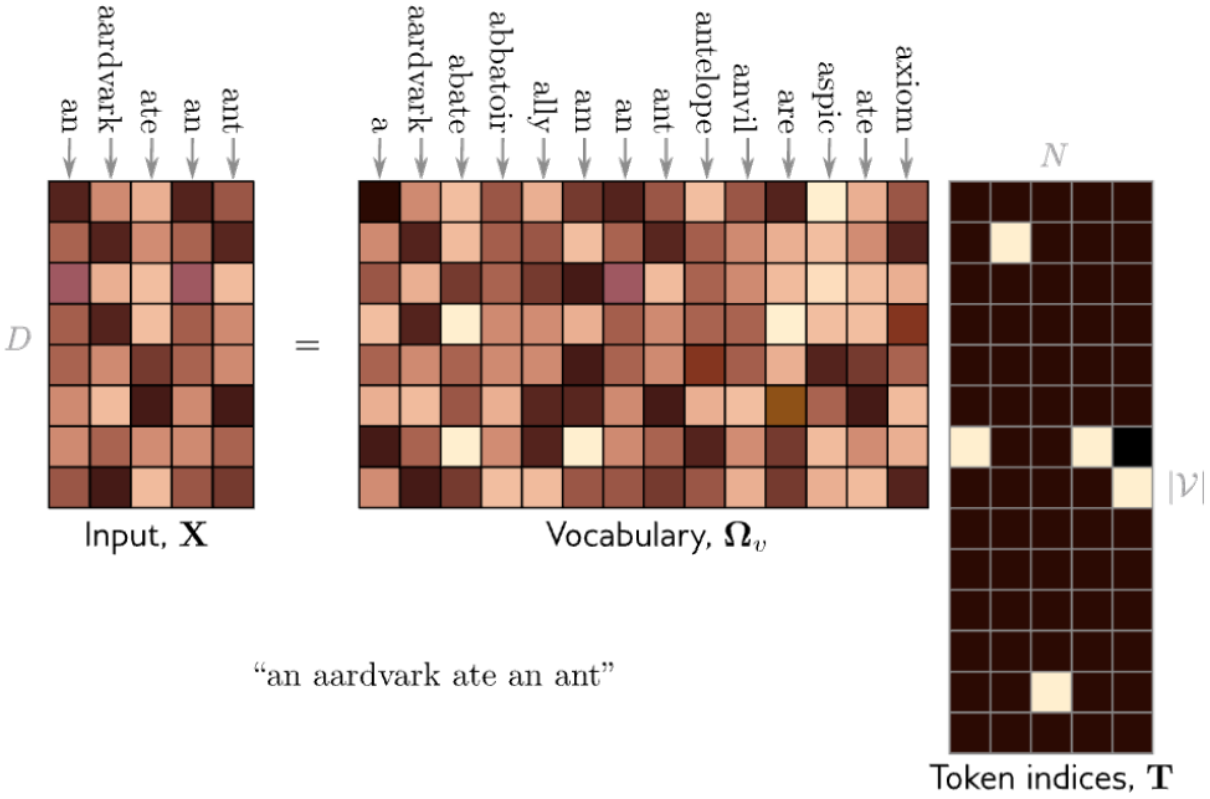
Una pipeline NLP tipica inizia con un **tokenizer** che divide il testo in words o word fragments.

Poi ciascuno di questi token è mappato ad un **embedding** imparato.

Questi embedding passano per una serie di **transformer layers**.

Ha skippato la tokenization e gli embeddings

Il punto è che abbiamo l'input, e poi abbiamo il vocabolario, prendiamo gli indici per capire cos'è il nostro input.



Quindi la **matrice di embedding** X che rappresenta il testo è fatta passare per una serie di transformer layers, chiamati **transformer model**.

Ci sono altri tipo di transformer models:

- Un **encoder** trasforma gli embeddings del testo in una rappresentazione che può supportare una varietà di tasks. Per esempio quello che abbiamo visto all'inizio per il ristorante, possiamo creare una rappresentazione utile per classificazione delle reviews.
- Un **decoder** genera un nuovo token che continua il testo di input.

- I **encoder-decoder** sono usati nelle task **sequence-to-sequence**, dove una stringa di testo è convertita in un'altra (per esempio machine translation)

Vediamo alcuni di questi esempi

Encoder model example: BERT

BERT is an encoder model that uses a vocabulary of 30,000 tokens.

Input tokens are converted to 1024-dimensional word embeddings and passed through 24 transformer layers, each containing a self-attention mechanism with 16 heads.

The queries, keys, and values for each head are of dimension 64 (i.e., the matrices $\Omega_{vh}, \Omega_{qh}, \Omega_{kh}$are 1024×64).

The dimension of the hidden layer in the neural network layer of the transformer is 4096.

The total number of parameters is $\sim$ 340 million.

When BERT was introduced, this was considered large, but it is now much smaller than state-of-the-art models.

Encoder models like BERT exploit transfer learning (remember?).

During pretraining, the parameters of the transformer architecture are learned using self-supervision from a large corpus of text.

The goal here is for the model to learn general information about the statistics of language.

The self-supervision task consists of predicting missing words from sentences from a large internet corpus.

In the fine-tuning stage, the resulting network is adapted to solve a particular task using a smaller body of supervised training data.

Decoder model example: GPT3

The basic architecture is extremely similar to the encoder model and comprises a series of transformer layers that operate on learned word embeddings.

However, the goal is different:

- The encoder aimed to build a representation of the text that could be finetuned to solve a variety of more specific NLP tasks.
- The decoder has one purpose: to generate the next token in a sequence. It can generate a coherent text passage by feeding the extended sequence back into the model (do you remember our practical session on RNNs?).

More formally GPT3 constructs an autoregressive language model.

Consider the sentence *It takes great courage to let yourself appear weak*. For simplicity, let's assume that the tokens are the full words.

The probability of the full sentence is:

$$\begin{aligned} Pr(\text{It takes great courage to let yourself appear weak}) \quad = \\ Pr(\text{It}) \times Pr(\text{takes}|\text{It}) \times Pr(\text{great}|\text{It takes}) \times Pr(\text{courage}|\text{It takes great}) \times \\ Pr(\text{to}|\text{It takes great courage}) \times Pr(\text{let}|\text{It takes great courage to}) \times \\ Pr(\text{yourself}|\text{It takes great courage to let}) \times \\ Pr(\text{appear}|\text{It takes great courage to let yourself}) \times \\ Pr(\text{weak}|\text{It takes great courage to let yourself appear}). \end{aligned}$$

More formally, an autoregressive model factors the joint probability $\Pr(t_1, t_2, \cdots, t_N)$ of the N observed tokens into an autoregressive sequence:

$$\Pr(t_1, t_2, \cdots, t_N) = \Pr(t_1) \prod_{n=2}^{N} \Pr(t_n | t_1, \cdots, t_{n-1})$$

The autoregressive language model is a *generative model*.

Since it defines a probability model over text sequences, it can be used to sample new examples of plausible text.

To generate from the model, we start with an input sequence of text (which might be just a special <start> token indicating the beginning of the sequence) and feed this into the network, which then outputs the probabilities over possible subsequent tokens.

We can then either pick the most likely token or sample from this probability distribution.

The new extended sequence can be fed back into the decoder network that outputs the probability distribution over the next token. By repeating this process, we can generate large bodies of text.

GPT3 applies these ideas on a massive scale: The sequence lengths are 2048 tokens long, and since multiple spans of 2048 tokens are processed at once, the batch size is 3.2 million tokens.

There are 96 transformer layers (some of which implement a sparse version of attention), each processing a word embedding of size 12288.

There are 96 heads in the self-attention layers, and the value, query, and key dimension is 128.

It is trained with 300 billion tokens and contains 175 billion parameters.