

LARGE LANGUAGE MODEL

Esistono diversi tipi di large language model, basati su uno o più punti del transformer:

- 1) ENCODER ONLY (BERT)
- 2) DECODER ONLY (GPT)
- 3) DECODER-ENCODER (T5)

Questi modelli sono anche detti foundation model.

BERT

La sigla sta per Bidirectional Encoder Representation from Transformers:



Abbiamo n parole in input e n word embeddings in output detti anche **contextualized word embeddings**, che possono essere interpretati come una rappresentazione del significato contestuale di ogni input token. Quindi lavora bene con il significato delle parole.

PRETRAINED

BERT viene preaddestrato come un masked LM, con una modifica, perché viene aggiunto il **next sentence prediction (NSP) task**:

$$\text{sentence} = \begin{cases} S_1 \\ S_2 \end{cases}$$

BERT ci deve dire se S_2 segue S_1 . Per farlo si aggiungono due token speciali: **[CLS]** e **[SEP]** che ci indicano l'inizio e la fine di una frase (con successivo inizio della frase che segue):

[CLS] S_1 [SEP] S_2 [SEP]

Per trasportare l'informazione delle due frasi, BERT usa semplici "position" embeddings per frasi: **sentence embeddings**

Se abbiamo due frasi, avremo due sentence embeddings da impostare: $W_s \in \mathbb{R}^{d \times d}$

$$S_i = \Sigma_i W_s \quad \text{con} \quad \Sigma_i = \begin{cases} [1, \emptyset] & \text{se } w_i \in S_1 \\ [\emptyset, 1] & \text{se } w_i \in S_2 \end{cases}$$

Dove quindi Σ_i è un vettore ziga che ci dice in quale frase sta la parola w_i .

Parametri di BERT:

	h	Layers	d	TOT
BERT-BASE	12	12	768	110M
BERT-LARGE	16	24	1024	340M

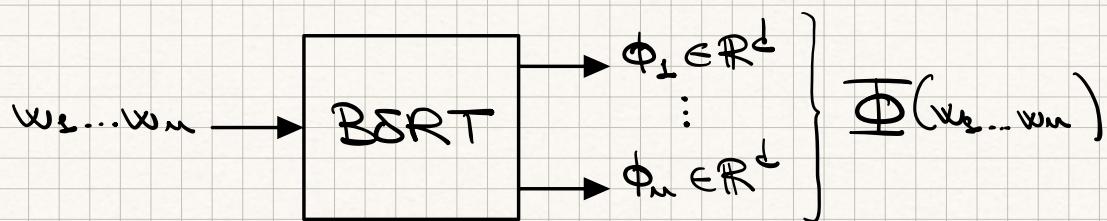
Entrambi i modelli accettano 512 tokens in ingresso.

Function

Possiamo modellare BERT come una funzione del tipo:

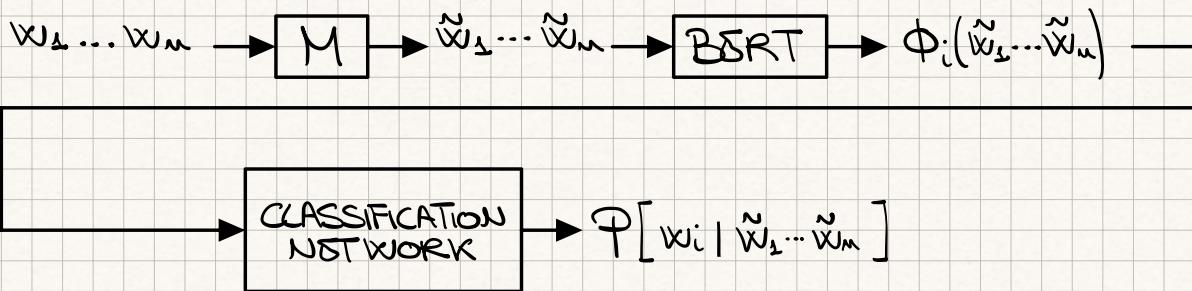
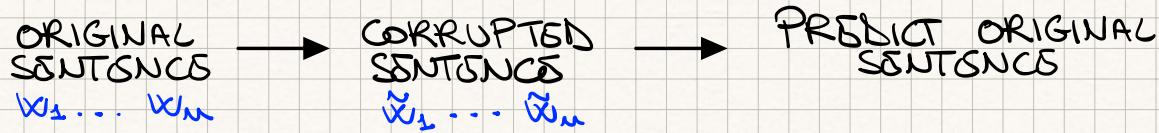
$$\Phi: V^m \rightarrow \mathbb{R}^{d \times m}$$

Dove per ogni ingresso w_i (singola parola) della sequenza è un embedding in \mathbb{R}^d .



MLM TRAINING

Si parte dalla frase originale, la si corrompe, e la si dà a BERT per avere una frase originale predetta.



Classification Network

Implementa una matrice di proiezione $\Sigma: \mathbb{R}^d \rightarrow \mathbb{R}^V$. La matrice $\Sigma \in \mathbb{R}^{d \times V}$. Per ogni elemento $\phi_i \in \Phi$:

$$P[x_i | \tilde{x}_1 \dots \tilde{x}_m] = \text{softmax}\left(\Sigma \cdot \phi_i(\tilde{x}_1 \dots \tilde{x}_m)\right)$$

Algoritmo di corruzione M

$$\tilde{x}_1 \dots \tilde{x}_m = M(x_1 \dots x_m)$$

Sia $I \subset \{1 \dots n\}$ - Sottinsieme casuale con almeno il 15% delle word position. Corrompiamo solo il 15% delle parole. Non possiamo corrompere tutto altrimenti BERT non si allenerà solo con le parole corrette.

```

for (i=1 ... n)
{
    if (i ∈ I)
    {
         $\tilde{w}_i = \begin{cases} [\text{MASK}] & \text{con } P=0.8 \\ w_i & \text{con } P=0.1 \\ \text{random } w & \text{con } P=0.1 \end{cases}$ 
    }
    else
    {
         $\tilde{w}_i = w_i$ 
    }
}
return  $\tilde{w}_1, \dots, \tilde{w}_n$ 

```

NEXT SENTENCE PREDICTION (NSP) TRAINING

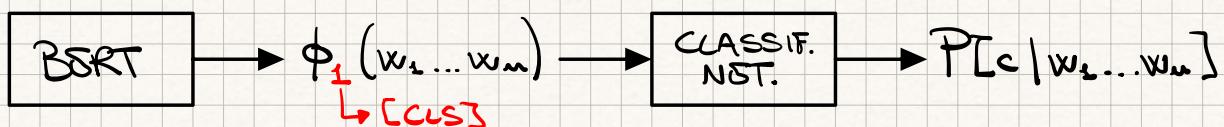
Abbiamo bisogno di due frasi s_1 e s_2 .

- Sia s_1 una frase del training corpus
- Con $P=0.5$, sia s_2 la frase che segue s_1 ($C=1$)
- Con $P=0.5$, s_2 è una frase casuale del training corpus ($C=2$)

$\text{classe}=1$

Abbiamo che l'input di BERT è:

$$w_1 \dots w_n = [\text{CLS}] s_1 [\text{SEP}] s_2 [\text{SEP}]$$



LOSS FUNCTION of BERT

Partiamo dal training corpus $\mathcal{D} = \{(x_1 \dots x_m, c)\}$:

$$\mathcal{L}_{BERT} = \sum_{(x_1 \dots x_m, c) \in \mathcal{D}} \left\{ \underbrace{\delta_{I, \tilde{x}_1 \dots \tilde{x}_m \sim M(x_1 \dots x_m)}}_{\text{Expectation per M randomico}} \cdot \left[\underbrace{\sum_{i \in I} -\log P[x_i | \tilde{x}_1 \dots \tilde{x}_m]}_{\text{Loss di ogni parola corretta}} + \right. \right. \\ \left. \left. -\log P[c | \tilde{x}_1 \dots \tilde{x}_m] \right] \right\}$$

Classificazione sulle frasi corrette

$$\mathcal{L}_{BERT} = \mathcal{L}_{MCM} + \mathcal{L}_{NSP}$$

GPT

La sigla sta per **Generative Pretraining Transformer**, e consiste nel solo blocco decoder di un Transformer.

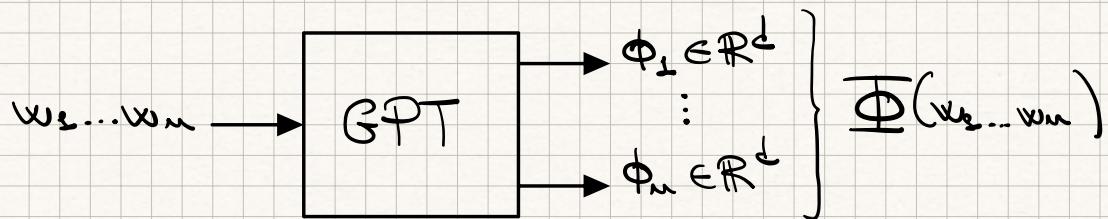
Si usa la **Causal self-attention**.

GPT guarda solo quello che viene prima nella sequenza in ingresso fino al token corrente.

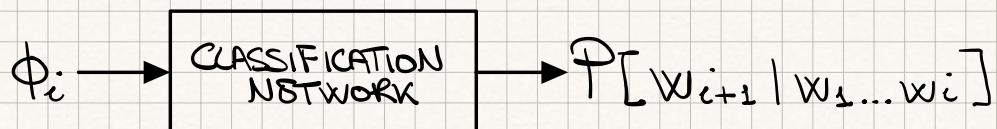
Parametri di GPT lingua inglese

	h	Layers	d	INPUT LENGTH	TOT	SIZE CORPUS
GPT	12	12	768	512	117M	4.5Gb
GPT-2	12	48	1600	1024	1.5B	40Gb
GPT-3	12	96	12288	2048	175B	570Gb

CAUSAL LM (CLM)



Come in BERT ogni ϕ_i è passata a una **rete di classificazione**:



$$P[w_{i+1} | w_1 \dots w_i] = \text{softmax}(\beta \cdot \phi_i(w_1 \dots w_i))$$

La matrice $B \in \mathbb{R}^{d \times V}$ e' la stessa vista in BERT.

LOSS FUNCTION of GPT

Sia \mathcal{D} , il training corpus, formato da un insieme di frasi. La loss function usata per GPT e' la ~~cross-entropy~~

$$\begin{aligned} L^{\text{GPT}} &= \sum_{w_1 \dots w_n \in \mathcal{D}} -\log P[w_1 \dots w_n] = \\ &= \sum_{w_1 \dots w_n \in \mathcal{D}} \sum_{i=1}^n -\log P[w_i | w_1 \dots w_{i-1}] \end{aligned}$$

T5

Sto per **Text-to-text Transfer Transformers**, ed è un modello encoder-decoder.

Parametri T5 per l'inglese

- 1) T5 small : 60 M di parametri
 - 2) T5 base : 220 M di parametri
 - 3) T5 large : 770 M di parametri
 - 4) T5 - 3B : 3B di parametri
 - 5) T5 - 11B : 11B di parametri
- } INPUT LENGTH = 512
CORPUS SIZE = 750Gb

TRAINING

Il modello T5, usa lo stesso loss di GPT. Per fare il training di task specializzati, si usa un token speciale messo come prefisso dell'ingresso.

Il task pre-allenato, è una variante del MLP, ed è detto **Span creation task (SCT)**.

Esempio

Prendiamo la frase

- 1) "One ring to bring all in the darkness
bind them"

Costruiamo la frase per darla in input al modello e quindi per allenarlo, usando L^{GPT}.

2) Input: "One ring to [x] in the [y] bind them"

3) Output: "[x] rule them [y] darkness"

2 e 3 sono date in input al modello in fase di training
mentre 1 è l'output da predire.

Sperimentalmente è provato che una coerenza del 15%. Funziona bene.

FINE-TUNING LLM

Un modo banale per specializzare un LLM è quello di specificare la descrizione del task, come sequenza di ingresso al modello:

- 1) "The cat couldn't fit into the hat, because it was too big.
Does it = the cat or the hat?"

Questo è detto **Task of coreference resolution**.

Se diamo la frase 1) per esempio a GPT, esso genera tre frasi. Fra le frasi possibili possiamo avere

- 2) "Because the cat was too big"
3) "Because the hat was too big"

Si sceglie la frase con probabilità maggiore.

Questo comportamento è detto **zero-shot learning**, che è un esempio di **in-context learning (ICL)**.

La sequenza d'ingresso è detta **TEMPLATE o PROMPT**

Esempio:

IN: "Translate english to italian: cheese"

OUT: "formaggio"

Esempio di **machine translation**

Utilizziamo ora prompt più complessi

One-shot Learning:

IN: "Translate english to italian: sea otter → Cionta
chees → "

Few-shot Learning:

IN: "Translate english to italiano:

sea otter → Cionta di mare
peppermint → menta piperita
flesh giraffe → giraffa peluche
cheese →

PROMPT ENGINEERING

È il compito di disegnare prompt efficaci per i LLMs.

Un prompt T può essere descritto come una sequenza di input:

$$P_0 \dots P_i \textcolor{red}{x} P_{i+1} \dots P_m \textcolor{red}{y} P_{m+1} \dots$$

Dove x e y indicano parole che specificano il task, dette anche **task-specific words**.

Traduciamo questa sequenza con gli input-embeddings:

$$e(P_0) \dots e(P_i) e(x) e(P_{i+1}) \dots e(P_m) e(y) e(P_{m+1}) \dots$$

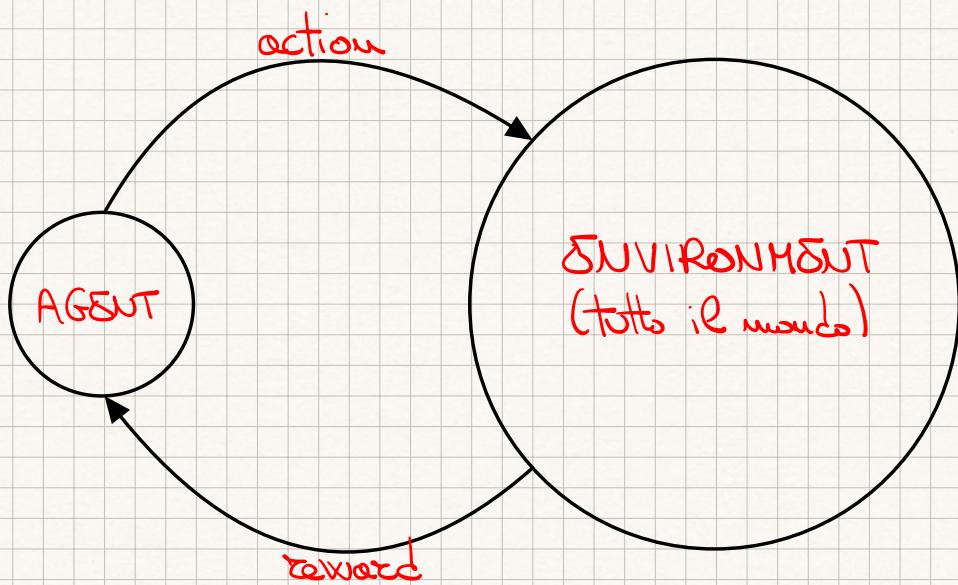
In questo caso T è disegnato a mano, e poi tradotto coi rispettivi embeddings, questo approccio è detto **hard prompting**

Invece di usare gli input-embeddings calcolati automaticamente dai LLMs, possiamo imparare nuovi input-embeddings $h(P_\theta)$... usando una semplice NN che viene specializzata (fine-tuned), e riempie gli input-embeddings del LLM. Questo secondo approccio è detto **soft prompting**

REINFORCEMENT LEARNING FROM HUMAN FEEDBACK (RLHF)

Problema dell' hallucination

I LLM sono letteralmente allenati con la morte che si trova nel web. Per questo, bisogna introdurre un meccanismo di validazione dell'output, per evitare che il modello risponda con affermazioni vere o false. Difatti si aggiunge un nuovo componente che classifica con "buono" o "non buono" un output, rispetto all'input dato al LLM.



Abbiamo un'entità chiamata agente, che evolve nel tempo, passando da uno stato all'altro. Mentre evolve il nostro agente interagisce con l'enivroment tramite una action. La risposta dell'enivroment è detta reward, e indica vari scenari possibili.

E come giocare a scacchi, **play and learn**, quando si perde si impara dove si è sbagliato. Migliorando per esperienza il comportamento dell'agent.

Matematicamente

Un sistema di questo tipo, detto **sistema RL**, può essere modellato come uno **MARKOV DECISION PROCESS (MDP)**,

$$\left. \begin{array}{l} 1) \text{ STATES} \\ 2) \text{ ACTIONS} \\ 3) \text{ REWARDS} \end{array} \right\} \text{A ogni tempo } t = \begin{cases} S_t \\ a_t \\ r_t \end{cases}$$

Intelligence of the agent

E' una **politica** che prende in input lo stato a tempo t S_t e produce come risultato una distribuzione di probabilità di tutte le azioni possibili per lo stato S_t :

$$\pi(a_t | s_t)$$

Evolution of the agent

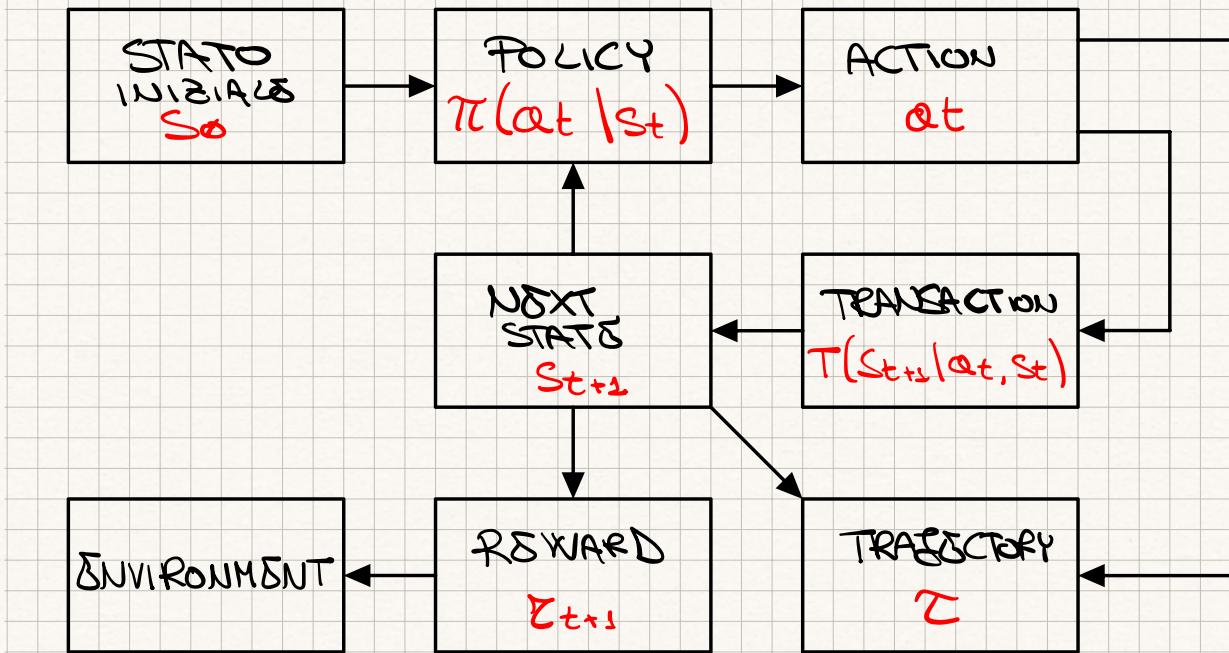
E' una **transizione** da uno stato che è soggetto a un'azione che lo fa passare al prossimo stato

$$T(s_{t+1} | s_t, a_t)$$

Traiettoria τ

Da qui il concetto di traiettoria, ovvero l'evoluzione dell'agente nel tempo. Assumiamo che la traiettoria è un insieme finito di transizioni \mathcal{T} fatto di stati e azioni che portano a nuovi stati:

$$\mathcal{T} = \left(s_0, a_0, s_1, a_1, \dots, s_{T-1}, a_{T-1}, s_T, \underbrace{a_T}_{\substack{\text{Azione nulla}}} \right)$$



TOTAL REWARD

Дата una traiettoria τ , la sua total reward $R(\tau)$:

$$R(\tau) = \sum_{t=0}^T r_t$$

RL NSI LLM

Applicare un sistema RL a un LLM fine-tuning è molto complicato per via delle troppe sequenze di input (prompts) e delle troppe sequenze di output (responses). In ogni caso abbiamo

- 1) LM \rightarrow Agent
- 2) Prompt $x \rightarrow$ state
- 3) Probabilità del LM $P(y|x)$ $\xrightarrow{\text{Policy}}$ Policy $P(a_t|s_t)$
Promo
output

Per ogni coppia \langle prompt, response \rangle , l'environment ritorna come reward:

$$r_t = r(s_t, a_t) = r(x, y)$$

La funzione di reward restituisce un numero reale che più alto è e meglio è.

La nostra policy $\pi(a_t|s_t)$ è implementata come un transformer dipendente dai parametri θ :

$$\pi_\theta(a_t|s_t)$$

Anche la traiettoria dipende da θ . Sia ζ_θ il set di tutte le traiettorie possibili (dipendenti da θ).

Objective Function $J(\theta)$

Abbiamo ora tutti gli elementi per definire la funzione obiettivo $J(\theta)$:

$$J(\theta) = \mathbb{E}_{z \sim \pi_\theta}[R(z)]$$

detto **expected total reward of the language model**.

Da qui il problema di trovare una policy ottima si reduce nel risolvere:

$$\theta^* = \operatorname{argmax}_\theta (J(\theta))$$

Risolvendo questo problema di ottimizzazione, tramite la scelta del gradiente: $\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta_t)$. Il valore di $\nabla_\theta J(\theta_t)$ è detto **policy gradient**, e lo calcoliamo:

$$\nabla_\theta J(\theta) = \nabla_\theta (\mathbb{E}_{z \sim \pi_\theta}[R(z)])$$

↓ Definizione di Expectation

$$= \nabla_\theta \left(\sum_{z \sim \pi_\theta} P_\theta(z) R(z) \right)$$

↓ Linearità dell'Expectation

$$= \sum_{z \sim \pi_\theta} \nabla_\theta (P_\theta(z)) R(z)$$

Dove $P_\theta(z)$ è la probabilità di generare la traiettoria z dato una certa azione che viene dalla policy π_θ .

Bisogna ora calcolare questa somma $\sum_{z \sim \pi_\theta} \nabla_\theta P_\theta(z) R(z)$.

Non possiamo approssimare con la **sample mean**:

$$\sum_{z \sim \pi_\theta} \nabla_\theta (P_\theta(z)) R(z) \not\approx \frac{1}{m} \sum_{i=1}^m \nabla_\theta (P_\theta(z_i)) R(z_i)$$

Perché $\nabla_\theta P_\theta(z)$ non è una distribuzione di probabilità, ma il suo gradiente.

In ogni caso da Analisi I sappiamo che:

$$\nabla \log(f(x)) = \frac{1}{f(x)} \cdot \nabla f(x)$$

$$\nabla f(x) = f(x) \nabla \log f(x)$$

Log-derivative trick

Riscriviamo:

$$\sum_{z \sim \pi_\theta} \nabla_\theta P_\theta(z) R(z) = \sum_{z \sim \pi_\theta} P_\theta(z) \cdot \nabla_\theta \log P_\theta(z) \cdot R(z)$$

Ora abbiamo la distribuzione di probabilità, per cui scriviamo:

$$\nabla_\theta J(\theta) \approx \mathbb{E}_z [\nabla_\theta \log P_\theta(z) \cdot R(z)]$$

Per definizione $P_\theta(\tau)$ è la probabilità di generare τ data l'azione che viene dalla politica $\pi_\theta(a_t | s_t)$:

$$P_\theta(\tau) = P(s_1) \prod_{t=1}^T T(s_{t+1} | s_t, \theta_t) \pi_\theta(a_t | s_t)$$

↓
Facciamo il log di tutto

$$= \log P(s_1) + \sum_{t=1}^T \left[\log T(s_{t+1} | s_t, \theta_t) + \log \pi_\theta(a_t | s_t) \right]$$

↓
Togliamo le componenti non dipendenti da θ , perché facciamo la derivata rispetto a θ .

$$\nabla_\theta \log P_\theta(\tau) = \sum_{t=1}^T \nabla_\theta \log \pi_\theta(a_t | s_t)$$

Alla fine abbiamo che:

$$\boxed{\nabla_\theta J(\theta)} \approx \mathbb{E}_\tau \left[\sum_{t=1}^T \nabla_\theta \log \pi_\theta(a_t | s_t) \cdot R(\tau) \right]$$

$$\approx \frac{1}{|D|} \sum_{\tau \in D} \sum_{t=1}^T \nabla_\theta \log \pi_\theta(a_t | s_t) \cdot R(\tau)$$

Con D set di traiettorie di training

Tutto questo meccanismo è detto **REINFORCEMENT algorithm**.

FUNZIONI DI REWARD

Bisogna capire cosa è $\mathbb{E}(s, a)$, ovvero la funzione di reward. La impariamo con una NN: $\mathcal{E}\psi(s, a)$.

Questa funzione va imparata usando dati che sappiamo essere buoni e verificati. Usiamo un dataset composto dalle seguenti tuple: (x, d^+, d^-) . Usiamo anche tuple negative perché deve riconoscere anche quando un risultato non è buono.

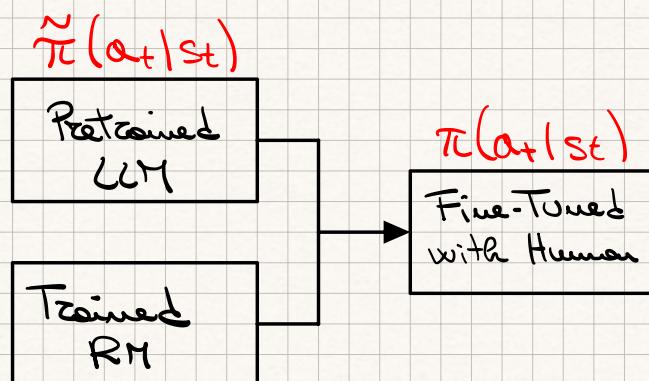
$$\left. \begin{array}{l} s^+ = \mathcal{E}\psi(x, d^+) \\ s^- = \mathcal{E}\psi(x, d^-) \end{array} \right\} \delta = s^+ - s^- \Rightarrow l(\delta) = -\log \sigma(\delta)$$

Delta score Sigmoid

Da qui ricaviamo un'ottima loss function per allenare ψ , perché vogliamo che s^- sia il più piccolo possibile:

LOSS FOR REWARD MODEL TRAINING

$$L^{RM} = \mathcal{L}(x, d^+, d^-) \sim \mathcal{N} \left[-\log (\sigma(\mathcal{E}\psi(x, d^+) - \mathcal{E}\psi(x, d^-))) \right]$$



Alla fine vorremo avere un LLM $\tilde{\pi}$ che non sia molto diverso dal LLM π di partenza (pre-trained). Per questo includiamo nella reward function un componente extra che penalizza la policy π se troppo diversa da $\tilde{\pi}$. Le due policy π e $\tilde{\pi}$ sono distribuzioni di probabilità, per cui possiamo confrontarle tramite:

$$D_{KL}(\pi \parallel \tilde{\pi})$$

$$\mathcal{L}(s_t | a_t) = \mathcal{L}_\psi(s_t | a_t) - \underbrace{\beta D_{KL}(\pi_\theta(a_t | s_t) \parallel \tilde{\pi}(a_t | s_t))}_{\text{penalty control parameter}}$$

Dove β è detto **penalty control parameter**: $\beta > 0$

SUMMARY

Ora abbiamo a nostra disposizione:

- 1) Un LLM pre-trainato (e/o fine-tuned)
- 2) Un reward model \mathcal{E}_θ e la reward function \mathcal{R}
- 3) Un algoritmo (**Policy gradient**) per ottimizzare 1) usando 2)

L'algoritmo è implementato come segue

- 1) Collezionare traiettorie τ partendo da π_θ^{old}
- 2) Per ogni τ calcolare $R(\tau)$
- 3) Calcolare $\nabla_\theta J(\theta)$
- 4) Aggiorniamo la policy $\pi_\theta^{\text{new}} = \pi_\theta^{\text{old}} + \alpha \nabla_\theta J(\theta)$
- 5) Ricomincia da 1.