UNIVERSITÀ
DELLA CALABRIA

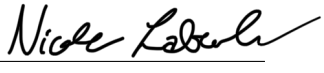
DIPARTIMENTO DI INGEGNERIA INFORMATICA,
MODELLISTICA, ELETTRONICA E SISTEMISTICA
(DIMES)

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA
INFORMATICA

Sviluppo di un approccio parameter-efficient per la distillazione di Large Language Models tramite tecniche di meta-learning

RELATORI
Dr. Riccardo Cantini
Ing. Alessio Orsino

CANDIDATO
Nicola Gabriele
242444



A.A. 2023-2024

Indice

Introduzione

I recenti sviluppi nel campo dell'*intelligenza artificiale (AI)* stanno avendo un impatto importante sulla società odierna in diversi ambiti. Nella sanità, ad esempio, l'AI offre prospettive volte ad una diagnostica più precisa, veloce e preventiva, nonché terapie personalizzate basate sui dati genetici dei pazienti. Anche il settore della finanza sta cambiando profondamente integrandosi con queste tecnologie; sempre più spesso strumenti basati su AI trovano impiego nelle analisi di mercato, nella gestione di portafogli d'investimento e nella rilevazione di attività fraudolente. Tra le innumerevoli tecniche racchiuse dal termine intelligenza artificiale, quelle che stanno riscuotendo maggiore successo negli ultimi anni sono quelle identificate come tecniche di *deep learning*, cioè l'insieme di tutte quelle tecniche di learning basate su *reti neurali*. Tali tecnologie ad oggi costituiscono lo stato dell'arte su molti task come la *classificazione*, la *machine translation* e il *language modeling*. A partire dal 2017, con la presentazione dell'architettura Transformer si è assistito ad un repentino aumento delle performance di questi modelli. Tuttavia, altrettanto repentino è stato l'aumento delle dimensioni, delle risorse computazionali e delle risorse energetiche richieste per il loro funzionamento, al punto che attualmente i più avanzati tra questi modelli sono conosciuti con l'appellativo di *Large Language Model*. Al tempo stesso, si fa sempre più marcata la necessità di integrare queste tecniche in dispositivi dalle limitate risorse computazionali, ad esempio nel campo dell'edge computing, ed altrettanto marcate sono le necessità di promuovere uno sviluppo ecosostenibile di queste tecnologie. Mossa da tali motivazioni, negli ultimi anni la letteratura scientifica in materia di deep learning si è arricchita di metodologie e proposte per la compres-

sione di questi modelli, con il fine di renderli più piccoli ed efficienti. Il presente elaborato ha l'obiettivo di presentare una nuova metodologia per la compressione di modelli di deep learning, proponendo un framework di *distillazione parameter-efficient* che integri tecniche di *meta-learning* nel processo. L'elaborato si sviluppa in quattro capitoli. Nel primo capitolo vengono presentate le principali tecniche di compressione presenti in letteratura, approfondendo le tecniche di *pruning*, *quantizzazione* e *distillazione*. A conclusione del primo capitolo vengono introdotte le tecniche di distillazione, presentandone una categorizzazione in tecniche *white-box* (pensate per qui contesti in cui si dispone dell'accesso agli output interni del modello da distillare) e tecniche *black-box* (pensate per quei contesti in cui si dispone solo dei token in output del modello da distillare). Nel secondo capitolo viene presentato lo stato dell'arte riguardo le tecniche di distillazione adatte ai Large Language Model. In particolare vengono presentate quattro importanti metodologie: *MiniLLM*, una metodologia di distillazione pensata appositamente per i modelli che generano testo in modo autoregressivo; *Patient Knowledge Distillation*, una metodologia di self-distillation che sfrutta gli stati interni del modello da distillare; *DiXtill*, una metodologia che integra tecniche di *eXplainable AI* nel processo di distillazione; e *MetaDistil* che introduce un framework basato sul meta-learning per il training congiunto del modello che trasferisce la conoscenza (affinchè si adatti a trasferirla meglio) e del modello che acquisisce conoscenza. Nel terzo capitolo viene introdotta la metodologia proposta. Per prima cosa vengono presentate le principali tecniche di *Parameter-Efficient Finetuning (PEFT)*, con particolare enfasi sulla tecnica *Low-Rank Adaptation (LoRA)* in quanto parte integrante della metodologia proposta. Successivamente vengono introdotte le architetture coinvolte nel processo di distillazione, e infine, viene presentata una nuova metodologia di distillazione parameter-efficient, che integra insieme PEFT e meta-learning in un framework di distillazione di tipo white-box. Il quarto ed ultimo capitolo è dedicato alla presentazione degli esperimenti condotti per la validazione della metodologia proposta. Nella prima parte del capitolo viene presentato il design degli esperimenti, descrivendo il dataset utilizzato, i principali iperparametri e le metriche di valutazione. Nella seconda parte del capitolo, in-

vece, vengono presentati i risultati sperimentali, il tuning degli iperparametri, e alcune conseguenze derivanti dall'uso del meta-learning nel processo di distillazione. Verrà quindi mostrato come l'utilizzo di questa metodologia permetta di ridurre significativamente il tempo di training mantenendo allo stesso tempo performance paragonabili a quelle ottenute attraverso l'applicazione di metodologie basate su full-finetuning.

Capitolo 1

Tecniche di compressione di Large Language Model

Negli ultimi anni, l'intelligenza artificiale, ha dimostrato di poter raggiungere (e in alcuni casi anche superare) performance umane in molti task del lavoro intellettuale. In particolare, a partire dal 2017, con la comparsa dell'architettura neurale *Transformer* (nota per l'introduzione del meccanismo della *self-attention* [36]), si è assistito ad un progressivo aumento nella dimensione dei modelli di deep learning che ha portato, ad oggi, a modelli che hanno un numero di parametri dell'ordine delle **migliaia di miliardi**. Un caso particolare è quello dei *Language Models*, che sono dei modelli basati su architettura Transformer e *di tipo decoder*, il cui task di training è appunto quello di modellare *il linguaggio*. Quando un modello del genere possiede un numero molto grande di parametri, rientra nella categoria dei **Large Language Model**. Diversi studi hanno dimostrato come i Large Language Model, imparando a modellare il linguaggio acquisiscano anche la capacità di svolgere task diversi da quelli per cui sono stati addestrati. Attualmente gli LLM, costituiscono la categoria di modelli di deep learning con il maggior numero di parametri: ad esempio, il modello *GPT-3.5*, sviluppato da *OpenAI* si stima abbia circa 175 miliardi di parametri. Questi modelli dunque, per quanto potenti essi siano, non permettono la loro esecuzione su dispositivi dalle ridotte capacità di calcolo (come nel caso dell'edge computing). Una soluzione

a questo problema risiede nelle *tecniche di compressione*. Nelle seguenti sezioni vengono approfondite le principali tecniche di compressione presenti in letteratura come il *pruning*, la *quantizzazione* e la *distillazione* (su cui si concentra il presente elaborato).

1.1 Pruning

Il pruning è una tecnica che punta a ridurre la dimensione dei modelli andando a rimuovere elementi della rete che non sono necessari. Questa tecnica può essere applicata in modo *strutturato* [39] e *non strutturato* [7].

1.1.1 Pruning non strutturato

Le tecniche di pruning non strutturato si basano sulla rimozione di pesi (il cui pruning viene fatto individualmente) in modo da costruire una struttura sparsa e irregolare. L'approccio generale per il pruning di un modello si basa sulla scomposizione del problema in un insieme di sotto problemi identici di pruning applicati a ciascun layer; per ciascun layer, si devono apprendere una matrice M_l detta *matrice di sparsità* (avente una certa densità target) e i pesi aggiornati $\hat{W}_l$ dimodoché questi ottimizzino la seguente funzione obiettivo:

$$argmin_{M_l,\hat{W}_l} \ ||W_l X_l - (M_l \odot \hat{W}_l) X_l||_2^2 \tag{1.1}$$

Questo problema, che richiede l'ottimizzazione congiunta della matrice M_l e dei pesi W_l, è NP-hard dunque non è ragionevole risolverlo in maniera esatta. La tecnica più comune per ovviare a questo problema consiste nel suddividere il problema in due fasi distinte: *mask selection* e *weight reconstruction*. Nella prima fase si costruisce la matrice di sparsità in accordo a dei criteri che devono essere stabiliti, mentre nella seconda si ottimizzano i pesi del layer mantenendo fissa la matrice di sparsità. Una volta fissata la matrice di sparsità, il valore ottimo di ciascuna riga della matrice dei pesi può essere calcolato risolvendo in maniera esatta

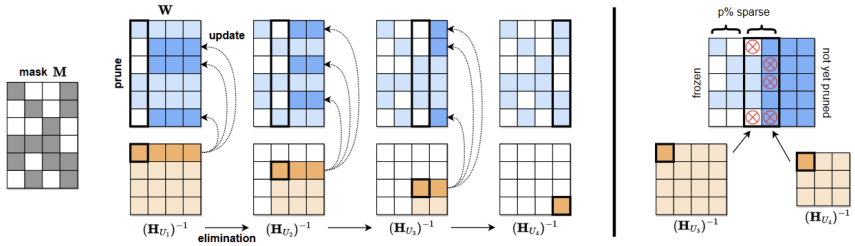


Figura 1.1: [A sinistra] Visualizzazione dell'algoritmo di ricostruzione SparseGPT. Data una maschera di pruning fissata M, viene fatto pruning dei pesi in ciascuna colonna della matrice W in modo incrementale usando una sequenza di Hessiane inverse $H_{U_j}^{-1}$, aggiornando i pesi rimanenti sulla riga e che si trovano a destra dell'elemento soggetto a pruning (quelli in bianco). [A destra] Illustrazione della selezione adattiva delle maschere.

il problema di weight recostruction sparsificato attraverso la seguente formula:

$$w_{M_i}^i = (X_{M_i} X_{M_i}^T)^{-1} X_{M_i} (w_{M_i} X_{M_i})^T \tag{1.2}$$

Dove X_{M_i} indica la matrice delle features di input i cui corrispondenti pesi non sono stati soggetti a pruning nella riga i. Risolvere questo problema richiede quindi di invertire la matrice Hessiana $(X_{M_i} X_{M_i}^T)^{-1}$ per ciascuna riga. Questa operazione costituisce una inefficienza che compromette la scalabilità della metodologia per modelli molto grandi. Per ovviare a questo problema, in [7] viene proposto un algoritmo chiamato *SparseGPT* che permette un pruning non strutturato efficiente. Per fare ciò, viene proposta una tecnica per la condivisione della stessa matrice Hessiana tra le righe che hanno maschere di pruning diverse. Tuttavia, le moderne architetture hardware basate su GPU faticano a sfruttare modelli sparsificati per rendere più efficiente l'inferenza. Di conseguenza, tecniche di pruning strutturato vengono spesso preferite nel contesto della compressione di Large Language Model. In figura 1.1 viene visualizzato graficamente l'algoritmo di ricostruzione SparseGPT.

1.1.2 Pruning strutturato

Le tecniche di pruning strutturato, differentemente dal caso precedente, si basano sulla rimozione di componenti a più ampia granularità come neuroni, layer e canali col fine di ridurre la dimensione del modello. Queste tecniche non soffrono della problematica hardware di cui è affetta la controparte non strutturata; tuttavia, esse non sono esenti da problematiche di altra natura. Molti degli approcci concettualmente più semplici al pruning strutturato, ad esempio, richiedono l'implementazione di operazioni algebriche ad hoc, e spesso raggiungono performance peggiori rispetto alla controparte non strutturata. Al fine di consentire un pruning strutturato, evitando le problematiche appena citate, in [39] viene proposta una metodologia chiamata *FLOP* (**F**actorized **Lo**w-rank **P**runing), che si basa sull'utilizzo di tecniche di fattorizzazione per ottenere un pruning che preservi la densità numerica delle matrici. In particolare, in fase di training, vengono apprese in modo adattivo le componenti a basso rango da eliminare per ottenere il miglior trade-off tra dimensioni del modello e performance. Un problema di pruning strutturato può essere formalizzato come un problema di learning *end-to-end* con regolarizzazione basata sulla norma l_0 [26]. Dato un modello con parametri θ, questi possono essere riscritti come: $\theta = \{\theta_j\}_{j=1}^{n}$ dove ciascun θ_j è un blocco di parametri. Detto ciò, una strategia di pruning può essere parametrizzata attraverso un vettore binario $\mathbf{z}$, mentre i parametri dopo il pruning saranno:

$$\tilde{\theta} = \theta \odot \mathbf{z} \tag{1.3}$$

La componente di regolarizzazione (norma di ordine zero), viene anch'essa calcolata sulla base del vettore $\mathbf{z}$ e sarà data dalla seguente formula:

$$||\tilde{\theta}||_0 = \sum_{j=1}^{n} z_j \tag{1.4}$$

Tale componente, che viene inserita all'interno della funzione obiettivo, rappresenta una misura della dimensione del modello dopo il pruning. In definitiva, fissata una funzione di loss per il training, la funzione obiettivo finale sarà data

da:

$$\mathbb{E}_{\mathbf{z}}\left[\frac{1}{D}\sum_{i=1}^{D}\mathcal{L}\left(x_i,y_i;\tilde{\theta}\right)+\lambda||\tilde{\theta}||_0\right] \tag{1.5}$$

Dove $\{x_i,y_i\}_{i=1}^{D}$ sono gli esempi di training, mentre λ è un iperparametro che regola quanto il modello finale sarà sparso. Nella pratica, data l'intrattabilità del problema così formulato, viene utilizzata una versione rilassata del problema in cui il vettore $\mathbf{z}$ non è un vettore a valori discreti ma a valori continui nell'intervallo [0,1]. Una caratteristica chiave della metodologia FLOP risiede nella scelta dei blocchi di parametri $\theta_1,...,\theta_n$ che sfrutta la tecnica della fattorizzazione. Data una matrice di parametri $W \in \mathbb{R}^{d'\times d}$, questa viene riparametrizzata e fattorizzata nel prodotto di 2 matrici più piccole, $W = PQ$ con $P \in \mathbb{R}^{d'\times r}$, $Q \in \mathbb{R}^{r\times d}$ e $r \leq min\{d,d'\}$. Dal momento che adesso W è risultante dalla somma di componenti a rango unitario p_k e q_k (*k-esima* riga della matrice q e *k-esima* colonna della matrice p), possiamo ottenere un pruning strutturato andando ad introdurre una variabile di pruning z_k per ciascuna componente:

$$W = PGQ = \sum_{k=1}^{r} z_k \times (\mathbf{p_k} \times \mathbf{q_k}) \tag{1.6}$$

Dove G è quindi una matrice diagonale, con valori $z_1,...,z_r$.

La metodologia appena discussa presenta ancora una problematica: la regolarizzazione con norma l_0 non permette un reale controllo sulla dimensione del modello dopo il pruning. Per ovviare a questo problema si può utilizzare una tecnica chiamata *Augmented Lagrangian Method* [39], che prevede l'utilizzo della seguente funzione obiettivo:

$$\max_{\lambda_1,\lambda_2}\min_{\theta,\alpha}\mathbb{E}_u\left[\frac{1}{D}\sum_{i=1}^{D}\mathcal{L}\left(x_i,y_i;\tilde{\theta}\right)\right]+g(\lambda,\alpha) \tag{1.7}$$

Dove $g(\lambda,\alpha)=\lambda_1\cdot(s(\alpha)-t)+\lambda_2\cdot(s(\alpha)-t)^2$ rappresenta una funzione di penalità in cui t è la dimensione target desiderata del modello dopo il pruning, $s(\alpha)$ è il valore atteso della dimensione del modello, mentre $\lambda_1,\lambda_2 \in \mathbb{R}$ sono moltiplicatori di Lagrange. Sostanzialmente, si introduce un vincolo sulla qualità della soluzio-

ne finale in modo da ottenere un modello che sia dimensionalmente quanto più possibile vicino alla dimensione desiderata.

1.1.3 Attention Head Pruning

Nel contesto delle architetture Transformer, di recente, si sta affermando una particolare tecnica di pruning, che non fa pruning dei pesi ma bensì delle attention head [29]. Al fine di rendere maggiormente chiaro quanto verrà discusso successivamente, di seguito viene riportato un breve paragrafo sulla multi-head attention.

Multi-headed Attention

Nella Multi-head attention, introdotta in [36], N_h layer di attention parametrizzati vengono calcolati parallelamente per produrre il risultato finale.

$$MHAtt(\mathbf{x}, q) = \sum_{h=1}^{N_h} Att_{W_k^h, W_q^h, W_v^h, W_o^h}(\mathbf{x}, q) \tag{1.8}$$

Dove $W_k^h, W_q^h, W_v^h \in \mathbb{R}^{d_h \times d}, W_o^h \in \mathbb{R}^{d \times d_h}$

In particolare, in [29], è stato mostrato che spesso una sola testa è sufficiente a garantire ottime performance, anche se il modello è stato addestrato con 12 o 16 teste. Anche se quanto detto non vale per tutti i layer, l'eliminazione delle teste superflue solo su un sottoinsieme di layer consente una buona riduzione delle dimensioni del modello senza degradare le prestazioni. Detto ciò, si pone il problema di scegliere quali teste rimuovere e quali mantenere; in [29] viene proposto un approccio basato su uno *score di importanza* sulla base del quale le teste vengono ordinate e poi rimosse una alla volta in un processo iterativo. Tale euristica si rende necessaria per evitare di interfacciarsi con un problema combinatorio impraticabile se si considera il numero di teste e il tempo richiesto per valutare ciascun modello.

Head importance score

Al fine di costruire uno score che sia un proxy per l'importanza delle teste, la prima cosa da fare è formulare una versione modificata della multi-head attention così definita:

$$MHAtt(\mathbf{x}, q) = \sum_{h=1}^{N_h} \xi_{\mathbf{h}} Att_{W_k^h, W_q^h, W_v^h, W_o^h}(\mathbf{x}, q) \tag{1.9}$$

Dove ξ_h è una *variabile maschera* con valori in $\{0,1\}$. Quando $\xi_h = 1 \ \forall h$ la formulazione equivale a quella vista nell'equazione 1.8, se invece per un certo h abbiamo $\xi_h = 0$ allora la testa corrispondente risulterà mascherata.

A questo punto, come mostrato in [29], possiamo definire un proxy per l'importanza delle teste guardando alla *sensitività* attesa del modello rispetto alla variabile ξ_h:

$$I_h = \mathbb{E}_{x \sim X} \left| \frac{\partial \mathcal{L}(x)}{\partial \xi_h} \right| \tag{1.10}$$

Dove X è la distribuzione dei dati mentre $\mathcal{L}(x)$ è il valore della funzione di loss per l'esempio x. Intuitivamente, se I_h ha valore alto allora è verosimile che cambiare il valore di ξ_h possa avere effetti importanti sul modello. È importante notare come sia strettamente necessario l'utilizzo del valore assoluto per evitare che contributi fortemente positivi e negativi si annullino a vicenda quando si sommano. Mettendo insieme le equazioni 1.8 e 1.10 e applicando la *chain rule* si ottiene la seguente:

$$I_h = \mathbb{E}_{x \sim X} \left| Att_h(x)^T \frac{\partial \mathcal{L}(x)}{\partial Att_h(x)} \right| \tag{1.11}$$

Dunque, la stima di I_h richiede di eseguire un passo di forward e uno di backward, il che lo rende non più lento del training stesso. Complessivamente, in [29], viene dimostrato che è possibile rimuovere rispettivamente il 20% delle teste dal modello WMT [36] e il 40% delle teste dal modello BERT [4] senza avere un impatto negativo notevole sulle performance.

1.2 Weight quantization

La quantizzazione è una tecnica di compressione dei modelli basata sulla riduzione della precisione dei parametri da *32-bit floating point* a *8-bit integers*. Allo stato dell'arte, i principali metodi di quantizzazione di modelli di deep learning sono: *Post-Training Quantizzation (PTQ)* che quantizza i parametri dopo il training senza modificare l'architettura del modello e *Quantization-Aware Training (QAT)* che integra la quantizzazione nel processo di training, consentendo un maggiore adattamento alla minore precisione con conseguente miglioramento delle performance rispetto a PTQ. Tuttavia, la metodologia QAT risulta essere poco scalabile all'aumentare delle dimensioni dei modelli (soprattutto nel contesto dei Large Language Model), e pertanto, la metodologia PTQ risulta essere attualmente più diffusa per la compressione di modelli di grandi dimensioni. Nel contesto delle tecniche PTQ, ricoprono particolare importanza *Activation-aware Weight Quantization* [25], *GPTQ* [8] e *dynamic quantization*.

1.2.1 Activation-aware Weigth Quantization

La quantizzazione mappa un numero a virgola mobile in uno intero rappresentabile con un minor numero di bit. Per sopperire alla consistente perdita di informazione che spesso affligge metodi di quantizzazione PTQ, in [25] è stata proposta una metodologia *Activation Aware*. Tale metodologia si basa sull'osservazione che in un LLM i pesi non sono tutti uguali riguardo all'impatto che questi hanno sulle performance del modello: infatti, c'è una piccola frazione (0.1%-1%) dei pesi che determina maggiormente le prestazioni. Dunque, evitando di quantizzare tali pesi che potremmo definire *"salienti"*, si può ottenere un modello compresso con un miglior trade-off tra dimensioni e prestazioni. Così facendo, però, si ottiene una matrice di pesi a *precisione mista* che non può essere utilizzata in modo efficiente a livello hardware. Per sopperire a questo, in [25] la quantizzazione viene eseguita *"per-channel"*. Se consideriamo un blocco di pesi w, avremo che dato un input x, $y = \mathbf{w}x$ e di conseguenza la controparte quantizzata sarà $y = Q(\mathbf{w})x$,

dove la funzione di quantizzazione è definita come segue:

$$Q(w) = \Delta \cdot Round(\frac{\mathbf{w}}{\Delta}), \;\; \Delta = \frac{max(|\mathbf{w}|)}{2^{N-1}} \tag{1.12}$$

Dove N è il numero di bit di quantizzazione mentre Δ è il fattore di scala. Se consideriamo ora $w \in \mathbf{w}$ e uno scalare $s > 1$ moltiplicando w per s e dividendo x per s otteniamo:

$$Q(s \cdot w) \cdot \frac{x}{s} = \Delta' \cdot Round(\frac{w \cdot s}{\Delta'}) \cdot x \cdot \frac{1}{s} \tag{1.13}$$

Dove Δ' è il fattore di scala dopo aver applicato s. In [25] viene dimostrato empiricamente che l'errore atteso della funzione $Round(\cdot)$ non varia (questo perchè la funzione *Round* mappa numeri a virgola mobile su numeri interi e l'errore è distribuito uniformemente nell'intervallo [0;0.5]). Inoltre, scalare un singolo elemento w di un valore s generalmente non varia il valore massimo di $\mathbf{w}$, di conseguenza, $\Delta \approx \Delta'$. A questo punto, al fine di trovare il valore ottimo di s (per ciascun canale di input), basta risolvere il seguente problema di ottimizzazione:

$$\begin{aligned} s^* &= arg\min_s \mathcal{L}(s) \\ \mathcal{L}(s) &= ||Q(\mathbf{W} \cdot diag(\mathbf{s}))(diag(\mathbf{s})^{-1} \cdot \mathbf{X}) - \mathbf{WX}|| \end{aligned} \tag{1.14}$$

Dove $\mathbf{Q}$ è la funzione di quantizzazione, $\mathbf{W}$ è la matrice originale dei pesi e $\mathbf{X}$ è la matrice delle feature di input. Tuttavia, questa funzione obiettivo non si presta bene ad essere ottimizzata dal momento che risulta essere *non differenziabile*. Pertanto, la precedente funzione obiettivo si sostituisce con la seguente:

$$\mathbf{s} = \mathbf{s_X}^{\alpha}, \;\; \alpha^* = arg\min_{\alpha} \mathcal{L}(\mathbf{s_X}^{\alpha}) \tag{1.15}$$

Dove $\mathbf{s_X}^{\alpha}$ è la magnitudine media della funzione di attivazione (per-channel) e α è un iperparametro utile a bilanciare quanto salvaguardare i pesi *"salienti"*. In figura 1.2 è riportata una rappresentazione grafica di quanto appena descritto.

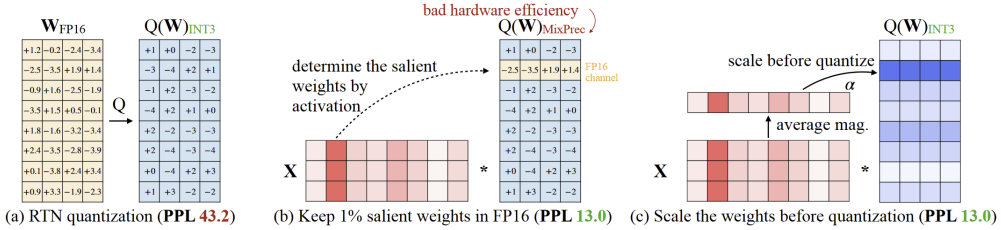


Figura 1.2: Osservando la distribuzione delle attivazioni è possibile determinare quale siano i pesi più importanti da non quantizzare (colonna colorata con una tonalità di rosso più marcata) [25].

1.2.2 GPTQ

GPTQ è una metodologia di quantizzazione di Large Language Model proposta in [8], che si colloca tra le tecniche di *Post-Training Quantization*. Tale metodo sfrutta informazioni del secondo ordine per rendere *più efficiente* il processo di quantizzazione e *più accurato* il modello quantizzato. L'algoritmo proposto si basa su tre step che vengono di seguito brevemente descritti.

Selezione arbitraria

Le tecniche tradizionali di quantizzazione, selezionano i pesi da quantizzare secondo una politica *Greedy*. Ad esempio, la tecnica *Optimal Brain Quantization* [9] seleziona ad ogni step il peso alla cui quantizzazione consegue il minore incremento dell'errore di quantizzazione. In [8], viene mostrato che selezionando i pesi secondo una certa politica, piuttosto che quantizzarli secondo un ordine randomico, la differenza in termini di qualità del risultato finale è minimale (e la differenza è ancor meno accentuata se ci collochiamo nel contesto di modelli molto grandi come gli LLM). L'approccio proposto punta a quantizzare i pesi di tutte le righe nello stesso ordine, e viene mostrato che questo porta a risultati simili alle tecniche tradizionali in termini di errore di quantizzazione. La conseguenza di ciò è che la matrice Hessiana (che come abbiamo già discusso in precedenza è fondamentale nel processo di quantizzazione) è uguale per tutte le righe, e quindi, non deve essere ricalcolata ad ogni passo (in quanto questa dipenderà solo dal-

Algorithm 1 Quantize $\mathbf{W}$ given inverse Hessian $\mathbf{H}^{-1} = (2\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}$ and blocksize B.

$\mathbf{Q} \leftarrow \mathbf{0}_{d_{\text{row}} \times d_{\text{col}}}$ *// quantized output*
$\mathbf{E} \leftarrow \mathbf{0}_{d_{\text{row}} \times B}$ *// block quantization errors*
$\mathbf{H}^{-1} \leftarrow \text{Cholesky}(\mathbf{H}^{-1})^\top$ *// Hessian inverse information*
for $i = 0, B, 2B, \ldots$ **do**
  for $j = i, \ldots, i + B - 1$ **do**
    $\mathbf{Q}_{:,j} \leftarrow \text{quant}(\mathbf{W}_{:,j})$ *// quantize column*
    $\mathbf{E}_{:,j-i} \leftarrow (\mathbf{W}_{:,j} - \mathbf{Q}_{:,j}) \,/\, [\mathbf{H}^{-1}]_{jj}$ *// quantization error*
    $\mathbf{W}_{:,j:(i+B)} \leftarrow \mathbf{W}_{:,j:(i+B)} - \mathbf{E}_{:,j-i} \cdot \mathbf{H}^{-1}_{j,j:(i+B)}$ *// update weights in block*
  end for
  $\mathbf{W}_{:,(i+B):} \leftarrow \mathbf{W}_{:,(i+B):} - \mathbf{E} \cdot \mathbf{H}^{-1}_{i:(i+B),(i+B):}$ *// update all remaining weights*
end for

Figura 1.3: Algoritmo GPTQ [8]

l'input e non dai pesi selezionati); ciò costituisce una ottimizzazione che rende la metodologia più adatta alla quantizzazione di modelli molto grandi.

Lazy-batch update

Quanto appena descritto, nella pratica non è direttamente implementabile in modo efficiente. Questo perchè la velocità con cui vengono trasportati i dati dalla memoria alle unità di calcolo è relativamente bassa, e di conseguenza, la bassa larghezza di banda dei dispositivi di memoria potrebbe costituire un collo di bottiglia per un utilizzo efficiente dei moderni dispositivi di calcolo GPU. Al fine di sopperire a questa problematica, in [8], viene proposto l'approccio descritto di seguito. Innanzitutto si può osservare che, per una certa colonna i, l'arrotondamento dipende solo dagli aggiornamenti che sono stati fatti sulla colonna stessa. Sfruttando tale osservazione, l'algoritmo può essere applicato ad un blocco di $B = 128$ colonne alla volta, mantenendo gli aggiornamenti contenuti in tali colonne e nel rispettivo blocco $B \times B$ dell'inversa della matrice Hessiana. Fatto ciò, quando l'intero blocco è stato processato, può essere applicato l'aggiornamento per intero alla matrice di pesi e all'inversa della matrice Hessiana.

Cholesky Reformulation

Un ulteriore problema tecnico a cui si deve far fronte sono le imprecisioni numeriche, che possono essere amplificate ulteriormente quando si effettuano gli

aggiornamenti a blocchi come visto nel paragrafo precedente. Potrebbe succedere, infatti, che l'inversa della matrice Hessiana diventi **indefinita**, con la conseguenza che la matrice quantizzata risultante sarà malformata e inaccurata. Tale problematica è tanto più marcata quanto più grande è il modello con cui si sta lavorando. Per risolvere questa problematica, in [8], viene proposta la seguente osservazione: l'unica informazione richiesta dalla matrice H_{Fq}^{-1} (H è la matrice Hessiana ed F_q denota i pesi non quantizzati quando si sta quantizzando il peso q) è la riga q, partendo dall'elemento sulla diagonale. Di conseguenza, queste righe possono essere *precalcolate* attraverso algoritmi numericamente stabili con un overhead minimo in termini di consumo di memoria. Possiamo notare che la rimozione di righe dalla matrice simmetrica H^{-1} attraverso la seguente:

$$H_{-q}^{-1} = \left(H^{-1} - \frac{1}{\left[H^{-1}\right]_{qq}} H_{:,q}^{-1} H_{q,:}^{-1}\right) \tag{1.16}$$

corrisponde ad una decomposizione *di Cholesky* di tale matrice, con la sola differenza che quest'ultima divide ulteriormente per un fattore $\left(\left[H_{Fq}^{-1}\right]_{qq}\right)^{\frac{1}{2}}$.

Decomposizione di Cholesky

Sia A una matrice *Hermitiana* e *definita positiva* su un campo $\mathbb{K}$, A può essere scomposta come:

$$A = LL^* \quad (A \in \mathbb{K}^{m\times m}) \tag{1.17}$$

con L *matrice triangolare inferiore* con elementi positivi sulla diagonale mentre L^* è la matrice *coniugata trasposta* di L. Inoltre, se A è **reale** e **simmetrica**, la coniugata trasposta coincide con la trasposta e la decomposizione si semplifica:

$$A = LL^T \quad (A \in \mathbb{R}^{n\times n}) \tag{1.18}$$

Di conseguenza, è possibile usare i *kernel* di Cholesky noti in letteratura per precalcolare tutte le informazioni necessarie dalla matrice H^{-1}. Inoltre, l'utilizzo

di questa tecnica fornisce un ulteriore speedup. L'algoritmo finale che raggruppa quanto appena descritto è riportato in figura 1.3.

1.3 Knowledge Distillation

La *Knowledge Distillation (KD)* è un processo che mira alla compressione e velocizzazione dei modelli, trasferendo la conoscenza da un modello molto grande e complesso in uno più piccolo ed efficiente. Sostanzialmente, la KD sfrutta la conoscenza acquisita da un modello molto complesso per *guidare* il training di uno più piccolo, in modo da ottenere un modello che sia efficiente nell'inferenza ma che abbia allo stesso tempo un degradamento delle di prestazioni minimo rispetto al modello più complesso [13]. Tale conoscenza acquisita dal modello più complesso, spesso risiede nelle distribuzioni di probabilità che produce in output, nelle rappresentazioni prodotte dai layer intermedi e nei valori della funzione di loss [40]. Durante il processo di distillazione, il modello più piccolo si addestra non solo sulle etichette degli esempi del dataset ma anche a *mimare* il comportamento del modello più grande. Il crescente interesse per i Large Language Model ha rafforzato l'interesse per le tecniche di distillazione e allo stesso tempo ha posto in esse sfide e problematiche nuove. Un esempio di tali problematiche risiede nella generalità degli LLM, che non sono progettati per un singolo task ma per una moltitudine di task diversi, soprattutto alla luce delle inaspettate capacità dimostrate da questi modelli [2]. Una prima categorizzazione delle tecniche di distillazione le suddivide in tre classi: *offline*, dove il modello teacher (quello che trasferisce la conoscenza) pre-addestrato guida l'addestramento del modello student (quello che deve apprendere come performare il task di downstream dal teacher); *online*, dove entrambi i modelli vengono aggiornati in un processo di training *end-to-end*; *self-distillation*, dove o lo stesso modello funge sia da student che da teacher (e trasferisce conoscenza dai layer più profondi a quelli più superficiali), oppure student e teacher sono due modelli separati ma condividono la stessa architettura. Tuttavia, al fine di distinguere le tecniche di distillazione in base al tipo e alla quantità di informazione coinvolta nel processo, di seguito,

distigueremo le tecniche in 2 categorie: *tecniche white-box*, che accedono alle informazioni interne del modello teacher (colui che trasferisce la conoscenza), e *tecniche black-box*, pensate per essere applicate laddove il modello teacher non è direttamente accessibile (ad esempio modelli accessibili solo tramite API) [40].

1.3.1 White-box distillation

I metodi white-box includono una vasta gamma di tecniche di distillazione che si applicano in tutti quei casi in cui si ha libero accesso alle informazioni interne del modello da distillare. In base a quanto la tecnica fa uso di tali informazioni interne durante il processo di distillazione, distinguiamo due possibili approcci.

Logit-based Introdotta in [13], tale metodologia convoglia la conoscenza attraverso i logit del modello teacher. A tale scopo si cerca di allineare i logit del modello student a quelli del modello teacher, cioè si vuole che il modello student apprenda una distribuzione di probabilità simile a quella del teacher. La funzione di loss base è la seguente:

$$\begin{gathered} \mathcal{L}_{logits} = KL(p^t||p^s) = \sum_{j=1}^{C} p_j^t log\left(\frac{p_j^t}{p_j^s}\right) \\ p_i^s = \frac{\exp(z_i^s/\tau)}{\sum_{j=1}^{C}\exp(z_j^s/\tau)}, \;\; p_i^t = \frac{\exp(z_i^t/\tau)}{\sum_{j=1}^{C}\exp(z_j^t/\tau)} \end{gathered} \tag{1.19}$$

Dove *KL* è la *divergenza di Kullback-Leibler* (una misura della distanza tra due distribuzioni di probabilità); $z^t, z^s \in \mathbb{R}^C$ sono rispettivamente i logit del teacher e i logit dello student, τ è il parametro temperatura che viene usato per scalare i logit e rendere la distribuzione meno sparsa e C è il numero di classi. Un approccio alternativo basato su logit è presente in [35], in cui viene proposta una metodologia di distillazione di BERT in una LSTM bidirezionale con un singolo layer. La loss utilizzata è la seguente:

$$\mathcal{L} = \alpha\mathcal{L}_{CE} + (1-\alpha)\mathcal{L}_{distill} = -\alpha\sum_i t_i log(y_i^S) + (1-\alpha)||z^{(B)} - z^{(S)}||_2^2 \tag{1.20}$$

Dove la prima componente è quella che permette l'addestramento sulle etichette reali del dataset, mentre la seconda è la componente che distilla la conoscenza contenuta nei logit del teacher nello student. Come già detto, questo tipo di tecniche, per funzionare bene necessitano di dataset molto grandi [35]; questo probabilmente è dovuto alla limitata quantità di conoscenza distillabile dai soli logit del modello teacher. Soprattutto in contesti di NLP, la data augmentation non è semplice da applicare, e quindi, ciò giustifica il fatto che le più avanzate tecniche white-box ad oggi siano hint-based.

Hint-based È una metodologia che cerca di sopperire alla limitata quantità di conoscenza distillabile dai soli logit del modello teacher introducendo informazioni aggiuntive provenienti dallo spazio di embedding, layer trasformer e layer predittivi [15, 32]. Nella forma più generale, la funzione loss per questo tipo di distillazione è la seguente:

$$\mathcal{L}_{hint} = \mathcal{H}(F^s, F^t) = ||F^t - \phi(F^s)||^2 \tag{1.21}$$

Dove $F^s, F^t \in \mathbb{R}^{H \times W \times C}$ denotano le feature dei layer intermedi dei due modelli, la funzione ϕ assicura che le feature dello student siano dimensionalmente conformi a quelle del teacher e $\mathcal{H}$ è una metrica (ad esempio MSE). Un esempio di distillazione hint-based è TinyBERT [20], il cui processo di distillazione sfrutta a pieno la conoscenza del teacher (BERT) ad ogni livello di profondità. L'architettura di TinyBert è di tipo Transformer (per cui si colloca tra le tecniche di self-distillation). L'idea alla base è che se il modello student ha M layer mentre il teacher ne ha N, possiamo definire una funzione che mappi un sottoinsieme di cardinalità M degli N layer del teacher e questi layer verranno poi distillati nello student. La loss utilizzata è la seguente:

$$\mathcal{L}_{model} = \sum_{x \in \mathcal{X}} \sum_{m=0}^{M+1} \lambda_m \mathcal{L}_{layer}(f_m^S(x), f_m^T(x)) \tag{1.22}$$

Dove $f_m^S(x), f_m^T(x)$ sono rispettivamente l'output prodotto dallo student e dal teacher dal layer m su input x. Scendendo più nel dettaglio avremo:

$$\begin{aligned}
\mathcal{L}_{layer} &= \begin{cases} \mathcal{L}_{embed} & se \;\; m = 0 \\ \mathcal{L}_{hidn} + \mathcal{L}_{attn} & se \;\; M \geq m > 0 \\ \mathcal{L}_{pred} & se \;\; m = M+1 \end{cases} \\
\mathcal{L}_{embed} &= MSE(E^S W_e, E^T) \\
\mathcal{L}_{hidn} &= MSE(H^S W_h, H^T) \\
\mathcal{L}_{attn} &= \frac{1}{h} \sum_{i=1}^{h} MSE(A_i^S, A_i^T) \\
\mathcal{L}_{pred} &= CE(z^T/t, z^S/t)
\end{aligned} \tag{1.23}$$

Dove:

- l è la lunghezza del testo in input.
- $H^S \in \mathbb{R}^{l \times d'}$, $H^T \in \mathbb{R}^{l \times d}$ sono rispettivamente gli hidden state dello student e del teacher.
- $W_h \in \mathbb{R}^{d' \times d}$ è una matrice addestrabile che ha il compito di proiettare gli hidden state dello student nello stesso spazio dimensionale di quelli del teacher.
- E^S ed E^T sono rispettivamente gli embedding del teacher e dello student.
- W_e è una matrice addestrabile col medesimo ruolo di W_h.
- h è il numero di teste di attention, $A_i \in \mathbb{R}^{l \times l}$.
- MSE è la funzione errore quadratico medio.
- z^S ed z^T sono rispettivamente i vettori di logits predetti dai due modelli.
- CE è la cross entropy loss.

- t è il parametro temperatura (che serve a scalare la distribuzione prodotta dal teacher per rendere le sue predizioni più informative).

Dunque, il modello student apprenderà dalle predizioni finali del teacher (tramite il contributo dato da $\mathcal{L}_{pred}$), dai layer interni ($\mathcal{L}_{hidn} + \mathcal{L}_{attn}$) e dai layer più superficiali ($\mathcal{L}_{embed}$) rendendo il processo di distillazione nel suo complesso una tecnica white-box. Un ulteriore esempio che ha destato molto interesse in letteratura è DistilBERT [31]. La sua morfologia è in generale molto simile a quella del modello BERT, con la differenza che esso non ha il layer *token-type embeddings* e il layer *pooler*, mentre il numero di layer viene ridotto di un fattore pari a 2. Una caratteristica chiave della metodologia proposta in [31] risiede nell'inizializzazione dei parametri del modello student, i quali vengono inizializzati a partire da quelli del modello teacher prendendone i layer in maniera alternata (questo è possibile grazie al fatto che le dimensionalità latenti dei due modelli corrispondono). La loss complessiva del processo è composta da tre componenti: la prima è la loss di distillazione, con la quale si addestrano lo student sulle predizioni del teacher; la seconda è la loss sul task di pretraining che nel caso di DistilBERT è una loss di masked language modeling; l'ultima componente è la loss del coseno, utile ad allineare gli stati interni dello student rispetto a quelli del teacher. Sperimentalmente, DistilBERT ha dimostrato di riuscire preservare il **97%** delle performance di BERT.

1.3.2 Black-box distillation

Molti tra i più moderni e performanti LLM (ad esempio GPT-4 [30]) sono *closed source*, cioè non consentono l'accesso all'implementazione e alle informazioni interne del modello. Pertanto, le tecniche white-box non possono essere applicate a questi modelli. Tecniche di distillazione applicabili su questa tipologia di modelli possono basarsi solo sulle predizioni che questi forniscono, di conseguenza, l'intero modello teacher viene trattato come una black-box.

In-Context Learning Distillation Proposta in [19], è una tecnica che mira a distillare l'abilità di few shot learner del teacher nello student. Il processo permette

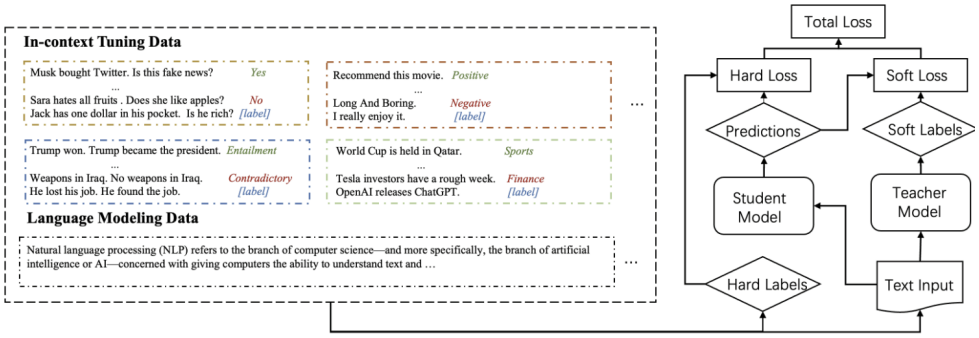


Figura 1.4: In-context distillation framework [19]

al modello più piccolo di apprendere dal teacher sia il few-shot learning che il language modeling. Per fare ciò, esso impara dalle predizioni fornite dal teacher. La funzione di loss $\mathcal{L}_{soft}$, che misura la discrepanza tra le predizioni dei due modelli, è composta da 2 contributi:

$$
\begin{aligned}
\mathcal{L}_{soft} &= \mathcal{L}^{ICL}_{soft} + \beta \mathcal{L}^{LM}_{soft} \\
\mathcal{L}^{ICL}_{soft} &= -\sum_{\mathcal{T} \in \mathcal{T}_{train}} \sum_{(x,y) \in D_{\mathcal{T}}} \sum_{c \in C_{\mathcal{T}}} P\big(y = c | x; S^x_k; \theta^t\big) log\left(P\big(y = c | x; S^x_k; \theta^s\big)\right) \\
\mathcal{L}^{LM}_{soft} &= \sum_{x \in \mathcal{D}_{LM}} P(x; \theta^t) log\big(P(x; \theta^t)\big)
\end{aligned}
\tag{1.24}
$$

Dove $\mathcal{L}^{ICL}_{soft}$ è la loss relativa all'in-context learning, $\mathcal{L}^{LM}_{soft}$ è il contributo del language modeling, β è un iper-parametro che media le due componenti, con θ^t, θ^s indichiamo rispettivamente i parametri del teacher e quelli dello student, $\mathcal{T}_{train}$ rappresenta l'insieme dei task di train, $\mathcal{D}_{LM}$ è l'insieme dei testi di carattere generale utilizzati per il language modeling e infine S^x_k è un insieme di k coppie (input, output) dimostrative che compongono gli esempi dimostrativi nel contesto.

Oltre a quanto descritto finora, lo student apprende ulteriormente dalle etichette reali di cui i dati sono corredati. Per fare ciò si introduce una seconda funzione

loss sulla falsariga di quanto fatto prima:

$$\begin{aligned}
\mathcal{L}_{hard} &= \mathcal{L}^{ICT}_{hard} + \mathcal{L}^{LM}_{hard} \\
\mathcal{L}^{ICT}_{hard} &= -\sum_{\mathcal{T}_{train}} \sum_{(x,y)\in\mathcal{D}_{\mathcal{T}}} log\Big(P\big(y|x;S^x_k;\theta^s\big)\Big) \\
\mathcal{L}^{LM}_{hard} &= -\sum_{x\in\mathcal{D}_{LM}} P\big(x|\theta^s\big)
\end{aligned} \tag{1.25}$$

Le due loss appena descritte $\mathcal{L}_{hard}$ e $\mathcal{L}_{soft}$ vengono poi combinate insieme nella loss di distillazione finale:

$$\mathcal{L}_{KD} = \alpha(t)\mathcal{L}_{hard} + (1-\alpha(t))\mathcal{L}_{soft} \tag{1.26}$$

Dove $\alpha(t)$ viene decrementato linearmente durate il training in modo da conferire progressivamente più peso alla componente soft e meno peso a quella hard. Una rappresentazione grafica del processo viene riportata in figura 1.4.

Chain-of-Thought Chain-of-Thought (CoT) rappresenta una strategia avanzata di prompting mirata al miglioramento delle capacità degli LLM di portare a termine task che richiedono ragionamento [40]. CoT integra all'interno del prompt, oltre alle coppie (input, output) fornite in contesti ICL, anche una serie di step di ragionamento intermedi che descrivono un processo attraverso il quale si giunge al risultato in output. Le principali tecniche di *CoT distillation* sfruttano le capacità degli LLM per costruire dataset arricchiti su task che richiedono ragionamento, i quali vengono poi utilizzati per il finetuning di modelli più piccoli. Ad esempio, in [23], vengono esplorati tre diversi metodi per la derivazione di interpretazioni usando gli LLM e per l'integrazione di essi in un framework di learning multi-task, in modo da ottenere modelli più piccoli con robuste capacità di reasoning. Inoltre, in [14], viene proposta una tecnica chiamata *finetuning CoT* utile a "catturare" le capacità di reasoning degli LLM per guidare modelli più piccoli nel risolvere task complessi. Tale tecnica prevede di utilizzare i modelli più grandi per generare soluzioni multiple a esempi campionati randomicamente nel dataset, per poi arricchire il dataset di queste nuove coppie. È stato dimostrato sperimen-

talmente che questa tecnica, su specifici task, permette a modelli con meno di 0.3 miliardi di parametri di performare meglio delle controparti più grandi.

Instruction Following L'instruction following è una tecnica che mira a migliorare le capacità dei language models di svolgere nuovi task senza affidarsi eccessivamente al limitato numero di esempi. Nella distillazione black-box, il trasferimento della conoscenza si affida solo al dataset, rendendo cruciale la disponibilità di dataset di grandi dimensioni. Di conseguenza, la maggior parte degli sforzi in questo ambito puntano alla creazione di dataset i cui esempi comprendono input, istruzioni e output. Ad esempio, in [38], viene introdotta la *self-instruction*, un processo semi-automatico che utilizza dei segnali indicatori prodotti dal modello stesso per raffinare le istruzioni. Il processo inizia con un insieme limitato di task creati manualmente per guidare il processo di generazione, il modello prompt utilizza questo insieme iniziale di istruzioni per generare un insieme di descrizioni più ampio, a questo punto, per ciascuna nuova istruzione viene generato un item (input, output). Alla fine del processo vengono usate delle euristiche per scartare le istruzioni di bassa qualità e/o duplicate, e le rimanenti vengono aggiunte al dataset. Il processo può essere ripetuto iterativamente fino al raggiungimento della dimensione desiderata. A questo punto si sarà ottenuto un dataset sufficientemente grande da consentire il training di un modello student.

Capitolo 2

Distillazione di Large Language Model: tecniche avanzate

Di recente, i Large Language Model hanno guadagnato una cospicua fama grazie alle loro spiccate capacità di comprensione e generazione del linguaggio. Tuttavia, essi spesso risultano inefficienti nell'utilizzo della memoria e dispendiosi in termini di risorse di calcolo necessarie al loro funzionamento; per esempio, BERT nella sua versione base possiede 110 *milioni* di parametri, mentre GPT-3 ne possiede 175 *miliardi*. Contemporaneamente alla nascita di tali modelli, si sta verificando uno sviluppo ed una diffusione sempre più massiccia di dispositivi a basse prestazioni, ad esempio, i dispositivi *Internet of Things (IoT)*. Nello scenario appena descritto, l'esigenza di avere delle tecniche di compressione sempre più accurate ed efficienti, che permettano l'integrazione di modelli di AI in tali dispositivi, diventa ancora più marcata. Spesso la distillazione costitusce la scelta più opportuna grazie alla sua grande flessibilità, in quanto permette di includere nel processo di compressione ogni tipo di informazione utile. Ad oggi, in letteratura sono svariate le tecniche di distillazione che hanno dimostrato la consistenza di questa tipologia di compressione. Ai fini di questo elaborato, risulta opportuno descrivere nel dettaglio alcune particolari tecniche di distillazione, ad esempio la distillazione con feedback, che saranno coinvolte successivamente nella metodologia proposta nel prossimo capitolo.

2.1 MiniLLM

Le tecniche tradizionali di distillazione perseguono un approccio generale, che non si cura del fatto che il task finale sia o meno open-ended (cioè che consista nella generazione autoregressiva di testo). Questo può portare le varie metodologie di distillazione ad essere più o meno accurate a seconda del task di downstream a cui vengono applicate. In [10] viene proposta una nuova metodologia, chiamata *MiniLLM*, per la distillazione di LLM in Language Model più piccoli. Tale tecnica si pone l'obiettivo di essere più adatta a modelli generativi, e per fare ciò, punta sulla sostituzione della classica KL loss con una che si adatta meglio a questo tipo di modelli. Questa tecnica si colloca tra le tecniche definite *white-box*, che come abbiamo già detto in precedenza, sono quelle in cui vengono sfruttati a pieno gli output del teacher, compresi gli stati interni dei layer intermedi. Consideriamo un task di generazione condizionata di testo, dove il modello produce una risposta $\mathbf{y} = \{y_t\}_{t=1}^{T}$ condizionatamente ad un prompt $\mathbf{x}$ campionato da una distribuzione $p_{\mathbf{x}}$. A questo punto, formuliamo il problema della distillazione come un problema di ottimizzazione, che minimizzi la differenza tra una distribuzione fissata $p(\mathbf{y}|\mathbf{x})$ (del teacher) e una distribuzione $q_\theta(\mathbf{y}|\mathbf{x})$ (dello student) parametrizzata da θ. Per fare ciò i metodi di distillazione tradizionali minimizzano la funzione di loss *forward Kullback-Leibler Divergence* (equazione 1.19). Anche se ampiamente usata, questa misura tende a sovrastimare le regioni vuote della distribuzione del teacher p nel task di text generation, soprattutto se lo student non è sufficientemente espressivo. Per ovviare a questa problematica in [10], viene utilizzata la *reverse Kullback-Leibler Divergence* tra le distribuzioni di probabilità di student e teacher:

$$\theta = \arg\min_{\theta} \mathcal{L}(\theta) = \arg\min_{\theta} \left[-\mathbb{E}_{\mathbf{x}\sim p_{\mathbf{x}}, \mathbf{y}\sim q_\theta} \log \left(\frac{p(\mathbf{y}|\mathbf{x})}{q_\theta(\mathbf{y}|\mathbf{x})} \right) \right] \tag{2.1}$$

La minimizzazione di questa funzione obiettivo consente al modello student di concentrarsi sulle regioni della distribuzione del teacher che sono più dense, e di trascurare quelle più sparse. Questo si traduce nella generazione di testo di maggiore qualità.

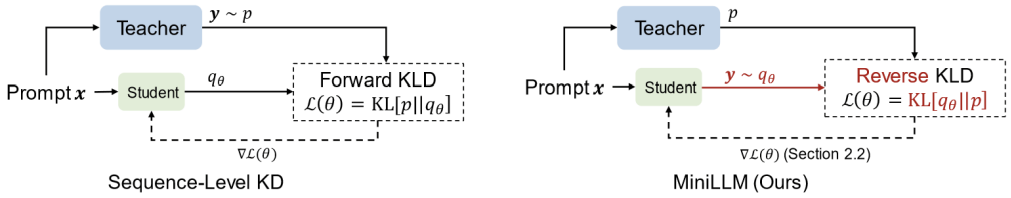


Figura 2.1: Confronto tra sequence-level KD (sinistra) e MiniLLM (destra); il primo forza lo student a memorizzare tutti gli esempi generati dal teacher mentre il secondo si serve del feedback del teacher per migliorare le capacità di text generetion dello student [10].

2.1.1 Ottimizzazione tramite Policy Gradient

Partiamo dall'osservazione che il gradiente della funzione obiettivo 2.1 può essere derivata usando il teorema *Policy Gradient*:

$$\nabla\mathcal{L}(\theta) = -\mathbb{E}_{\mathbf{x}\sim p_{\mathbf{x}},\mathbf{y}\sim q_\theta(\cdot|\mathbf{x})} \sum_{t=1}^{T}(R_t - 1)\nabla\log(q_\theta(y_t|\mathbf{y}_{<t},\mathbf{x})) \tag{2.2}$$

Dove $T = |\mathbf{y}|$, e $R_t = \sum_{t'=t}^{T}\log\left(\frac{p(y_{t'}|\mathbf{y}_{<t'},\mathbf{x})}{q_\theta(y_{t'}|\mathbf{y}_{<t'},\mathbf{x})}\right)$ corrisponde all'accumulo delle componenti $r_{t'} = \frac{p(y_{t'}|\mathbf{y}_{<t'},\mathbf{x})}{q_\theta(y_{t'}|\mathbf{y}_{<t'},\mathbf{x})}$ che misurano la qualità di ciascuno step di generazione. Intuitivamente, i testi generati dovrebbero avere alta probabilità sotto la distribuzione del teacher per effetto dell'incremento di $p(y_{t'}|\mathbf{y}_{<t'},\mathbf{x})$, rimanendo comunque diversificati per effetto della minimizzazione di $q_\theta(y_{t'}|\mathbf{y}_{<t'},\mathbf{x})$. Il valore atteso previsto dall'equazione 2.2 viene calcolato attraverso la *tecnica Monte Carlo*. Tuttavia, policy gradient soffre del problema dell'*alta varianza* e del *reward hacking*, e inoltre, R_t tende a favorire risposte corte, con la conseguenza che lo student tende a fornire risposte di lunghezza nulla (vuote). Per ovviare a questi problemi, in [10], utilizzano tre strategie che verranno di seguito illustrate.

Single-step Decomposition La qualità della generazione r_t al singolo step è critica, in quanto l'errore commesso sul primo token si accumula lungo tutta la sequenza, quindi occorre prestare maggiore attenzione a r_t. Per fare ciò, si può riscrivere $\nabla\mathcal{L}(\theta)$ in modo da decomporre r_t da R_t e calcolare direttamente il

gradiente di $\mathbb{E}_{y_t \sim q_\theta(t)}$.

$$\begin{aligned}\nabla \mathcal{L}(\theta) =& \mathbb{E}_{\mathbf{x}\sim p_\mathbf{x}, \mathbf{y}\sim q_\theta(\cdot|\mathbf{x})}\left[-\sum_{t=1}^{T} \nabla \mathbb{E}_{y_t \sim q_\theta(t)}[r_t]\right] + \\ &+ \mathbb{E}_{\mathbf{x}\sim p_\mathbf{x}, \mathbf{y}\sim q_\theta(\cdot|\mathbf{x})}\left[-\sum_{t=1}^{T} R_{t+1} \nabla \log(q_\theta(y_t|\mathbf{y}_{<t}, \mathbf{x}))\right] = \\ &= (\nabla \mathcal{L})_{Single} + (\nabla \mathcal{L})_{Long}\end{aligned} \tag{2.3}$$

Dove $q_\theta(t) = q_\theta(\cdot|\mathbf{y}_{<t}, \mathbf{x})$. Notiamo che $\mathbb{E}_{y_t \sim q_\theta(t)}[r_t]$ può essere calcolato direttamente sommando i contributi di ciascuna parola nel vocabolario, senza l'utilizzo della tecnica Monte Carlo, e inoltre, è derivabile rispetto a θ. Questa decomposizione fornisce una stima più accurata ed efficiente della qualità della generazione al singolo step, il che riduce la varianza durante il training e accelera la convergenza.

Teacher-Mixed Sampling Il training attraverso l'equazione 2.2 è affetto dal fenomeno del *reward hacking* poichè q_θ a volte produce frasi $\mathbf{y}$ degeneri (ad esempio frasi ripetute), che ottengono uno score alto dal teacher durante il campionamento. Per creare una distribuzione di campionamento migliore, in [10] la distribuzione del teacher e quella dello student vengono mescolate insieme.

$$\tilde{p}(y_t|\mathbf{y}_{<t}, \mathbf{x}) = \alpha \cdot p(y_t|\mathbf{y}_{<t}, \mathbf{x}) + (1-\alpha) \cdot q_\theta(y_t|\mathbf{y}_{<t}, \mathbf{x}) \tag{2.4}$$

Dove α controlla l'importanza della distribuzione del teacher nel mix. Campionando da $\tilde{p}$, la distribuzione del teacher aiuta a sopprimere la generazione di sentence di bassa qualità, e modera il problema del reward hacking. Possiamo includere questa modifica nell'equazione 2.3 come segue:

$$\begin{aligned}(\nabla \mathcal{L})_{Single} &= -\mathbb{E}_{\mathbf{x}\sim p_\mathbf{x}, \mathbf{y}\sim \tilde{p}(\cdot|\mathbf{x})}\left[\sum_{t=1}^{T} w_t \nabla \mathbb{E}_{y_t \sim q_\theta(t)}[r_t]\right] \\ (\nabla \mathcal{L})_{Long} &= -\mathbb{E}_{\mathbf{x}\sim p_\mathbf{x}, \mathbf{y}\sim \tilde{p}(\cdot|\mathbf{x})}\left[\sum_{t=1}^{T} w_t R_{t+1} \nabla \log(q_\theta(y_t|\mathbf{y}_{<t}, \mathbf{x}))\right]\end{aligned} \tag{2.5}$$

Dove $w_t = \prod_{t'=1}^{t} \frac{q_\theta(y_{t'}|\mathbf{y}_{<t'},\mathbf{x})}{\tilde{p}(y_{t'}|\mathbf{y}_{<t'},\mathbf{x})}$ è un peso che definisce l'importanza di ciascun termine. Tuttavia w_t, così calcolato, nella pratica causa alta varianza in quanto richiede di moltiplicare i pesi di importanza (per-token) su molti step temporali; e di conseguenza, la varianza ad ogni step si accumula. Perciò, tale peso può essere approssimato come $w_t \approx \frac{q_\theta(y_{t'}|\mathbf{y}_{<t'},\mathbf{x})}{\tilde{p}(y_{t'}|\mathbf{y}_{<t'},\mathbf{x})}$ per ridurre la varianza dello stimatore in 2.5.

Length Normalization Le sentence molto lunghe tendono ad avere R_{t+1} piccolo, e pertanto, occorre normalizzare il termine R_{t+1} come segue:

$$R_{t+1}^{Norm} = \frac{1}{T-t-1} \sum_{t'=t+1}^{T} \log \frac{p(y_{t'}|\mathbf{y}_{<t'},\mathbf{x})}{q_\theta(y_{t'}|\mathbf{y}_{<t'},\mathbf{x})} \tag{2.6}$$

Integrando insieme le tre strategie otteniamo la seguente:

$$\nabla\mathcal{L}(\theta) = -\mathbb{E}_{\mathbf{x}\sim p_{\mathbf{x}},\mathbf{y}\sim\tilde{p}(\cdot|\mathbf{x})}\left[\sum_{t=1}^{T} w_t \left[\underbrace{\nabla \sum_{y'\in V} q_\theta(y'|\mathbf{y}_{<t},\mathbf{x}) \log\left(\frac{p(y'|\mathbf{y}_{<t},\mathbf{x})}{q_\theta(y'|\mathbf{y}_{<t},\mathbf{x})}\right)}_{(\nabla\mathcal{L})_{Single}\ part} + \underbrace{R_{t+1}^{Norm}\frac{\nabla q_\theta(y_t|\mathbf{y}_{<t},\mathbf{x})}{q_\theta(y_t|\mathbf{y}_{<t},\mathbf{x})}}_{(\nabla\mathcal{L})_{Long}^{Norm}\ part}\right]\right] \tag{2.7}$$

Algoritmo di training finale Si parte da un modello student pre-addestrato su un grande dataset di documenti $\mathcal{D}_{PT}$, dopodiché, si addestra lo student su un task di text generation su un dataset $\mathcal{D}$, con la supervisione di un modello teacher (ad esempio un LLM finetuned su $\mathcal{D}$). A questo punto viene preso il modello student con la loss più bassa come inizializzazione del seguente processo: si calcolano i gradienti $(\nabla\mathcal{L})_{Single}$ e $(\nabla\mathcal{L})_{Long}^{Norm}$ come visto in precedenza e si aggiunge una loss di language modeling $\mathcal{L}_{PT} = -\mathbb{E}_{\mathbf{d}\sim\mathcal{D}_{PT}} \log q_\theta(\mathbf{d})$, e il modello viene poi aggiornato secondo una combinazione dei gradienti, ad esempio per somma: $(\nabla\mathcal{L})_{Single} + (\nabla\mathcal{L})_{Long}^{Norm} + \nabla\mathcal{L}_{PT}$. L'algoritmo, nel complesso, è illustrato dalla figura 2.2

Input: Conditional generation dataset $\mathcal{D}$ consisting of prompts and ground-truth responses
Pre-training corpus $\mathcal{D}_{\text{PT}}$ consisting of long-document plain texts
A teacher model with output distribution p
An initial student model pre-trained on $\mathcal{D}_{\text{PT}}$, with the output distribution q_{θ_0}
Learning rate η; Batch size M; Clipping Threshold ϵ
Output: A student model with the output distribution q_θ
Fine-tune the student model from θ_0 on $\mathcal{D}$ supervised by the ground truth responses and choose θ with the lowest validation loss.
repeat
Sample a mini-batch of prompts from $\mathcal{D}$ and collect responses from $\widetilde{p}$ to get $\mathcal{S} = \{(\boldsymbol{x}^m, \boldsymbol{y}^m)\}_{m=1}^M$
Sample a mini-batch $\mathcal{D}'_{\text{PT}} = \{\boldsymbol{d}^m\}_{m=1}^M$ from $\mathcal{D}_{\text{PT}}$
Compute $(\nabla\mathcal{L})_{\text{Single}} = -\frac{1}{M}\sum_{\boldsymbol{x},\boldsymbol{y}\in\mathcal{S}}\sum_{t=1}^{T} w_t \nabla \sum_{y_t\in V} q_\theta(y_t|\boldsymbol{y}_{<t},\boldsymbol{x}) \log \frac{p(y_t|\boldsymbol{y}_{<t},\boldsymbol{x})}{q_\theta(y_t|\boldsymbol{y}_{<t},\boldsymbol{x})}$ ▷ Eq. 5
Compute $(\nabla\mathcal{L})^{\text{Norm}}_{\text{Long}} = -\frac{1}{|M|}\sum_{\boldsymbol{x},\boldsymbol{y}\in\mathcal{S}}\sum_{t=1}^{T} R^{\text{Norm}}_{t+1} \nabla \min[\rho_t(\theta), \text{clip}(\rho_t(\theta), 1-\epsilon, 1+\epsilon)]$,
where $\rho_t(\theta) = \frac{q_\theta(y_t|\boldsymbol{y}_{<t},\boldsymbol{x})}{\widetilde{p}(y_t|\boldsymbol{y}_{<t},\boldsymbol{x})}$ ▷ Eq. 5, Eq. 6
Compute the gradient of the language modeling loss: $\nabla\mathcal{L}_{\text{PT}} = -\frac{1}{M}\sum_{\boldsymbol{d}\in D'_{\text{PT}}} \nabla \log q_\theta(\boldsymbol{d})$
Update model parameters: $\theta \leftarrow \theta - \eta\left[(\nabla\mathcal{L})_{\text{Single}} + (\nabla\mathcal{L})^{\text{Norm}}_{\text{Long}} + \nabla\mathcal{L}_{\text{PT}}\right]$
until converge and **return** q_θ

Figura 2.2: Algoritmo di training MiniLLM [10]

2.2 Patient Knowledge Distillation

La Patient Knowledge Distillation [32], è una tecnica di distillazione che appartiene alla famiglia delle tecniche white-box. L'introduzione di tale tecnica, ha proposto per la prima volta in letteratura l'utilizzo di informazioni dai layer intermedi del modello teacher nel processo di distillazione. Dunque di seguito analizzeremo nel dettaglio tale tecnica.

Iniziamo col dire che, quando si adotta questa tecnica, è possibile seguire 2 differenti strategie: *(i) PKD-Last* in cui lo studente apprende solo dagli ultimi k layer del teacher, sotto l'assunzione che nei layer più superficiali del teacher sia contenuta la maggior parte della conoscenza; *(ii) PKD-Skip* in cui lo studente apprende da un layer ad intervalli di k, sotto l'assunzione che anche i layer più profondi del teacher contengono informazioni importanti da apprendere per lo student. La figura 2.3 illustra graficamente queste due strategie. Diamo dunque uno sguardo più da vicino al processo di distillazione.

Definizione del problema Il modello teacher è rappresentato secondo una funzione $f(x;\theta)$ dove x è l'input della rete mentre θ sono i parametri del modello. L'obiettivo del processo di distillazione è di apprendere un nuovo insieme di para-

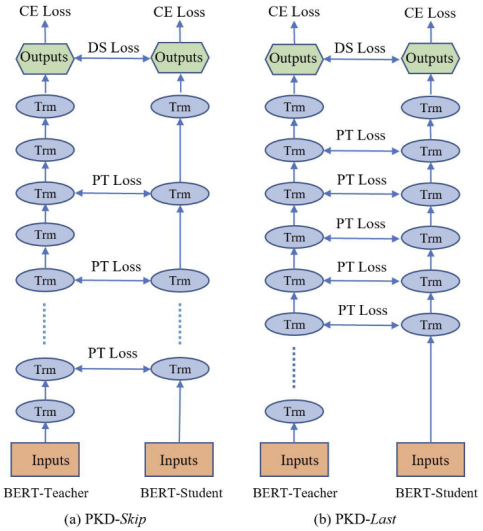


Figura 2.3: Rappresentazione delle due strategie di selezione dei layer da distillare: (Sinistra) *PKD-Skip*, lo student apprende dal teacher ogni due layer; (destra) *PKD-Last*, lo student apprende solo dagli ultimi sei layer [32]

metri θ' per il modello student $g(x;\theta')$ dimodoché questo raggiunga performance simili a quelle del modello da cui ha appreso.

Funzione Obiettivo Indichiamo con $\{x_i, y_i\}_{i=1}^{N}$ gli N esempi di training, dove ciascun x_i rappresenta l'input del modello mentre ciascuna y_i rappresenta l'etichetta ad esso associata. Le probabilità in output dato un input x_i possono essere formulate come:

$$\hat{y}_i = P^t(y_i|x_i) = softmax(\frac{\mathbf{Wh_i}}{T}) \tag{2.8}$$

Dove $P(\cdot|\cdot)$ rappresenta le probabilità in output dal teacher, $\hat{y}_i$ prende il nome di *soft-label* e T è un parametro di temperatura che regola quanto si fa affidamento sulle soft-prediction del teacher. Una temperatura più alta produce una distribuzione di probabilità più diversificata sulle varie classi. In modo analogo, se θ^s sono i parametri dello student, $P^s(\cdot|\cdot)$ saranno le probabilità prodotte in output dallo student. Di conseguenza, la distanza tra le predizioni del teacher e quelle

dello student può essere definita come:

$$L_{DS} = -\sum_{i\in[N]}\sum_{c\in C}\left[P^t(y_i = c|x_i;\hat{\theta}^t)\cdot log\big(P^s(y_i = c|x_i;\hat{\theta}^s)\big)\right] \tag{2.9}$$

Dove C è l'insieme delle etichette di classe. Oltre ad incoraggiare lo student ad emulare il comportamento del modello teacher, il modello student può anche essere soggetto a finetuning su un task di interesse inserendo una cross-entropy ad hoc:

$$L^s_{CE} = -\sum_{i\in[N]}\sum_{c\in C}\left[\mathbb{1}[y_i = c]\cdot log\big(P(y_i = c|x_i;\theta^s)\big)\right] \tag{2.10}$$

E quindi, mettendo insieme queste due componenti, otteniamo la funzione di loss complessiva a guida del processo di distillazione:

$$L_{KD} = (1-\alpha)L^s_{CE} + \alpha L_{DS} \tag{2.11}$$

In cui viene inserito l'iper-parametro α utile a bilanciare il contributo delle due componenti. Come abbiamo detto, questa tipologia di distillazione si distingue per il fatto che lo student apprende anche dai layer interni del teacher. Fare questo per tutti i token, però, sarebbe computazionalmente molto costoso e renderebbe la metodologia in pratica poco applicabile. Per sopperire a ciò, l'idea è di utilizzare solo gli embedding dei token [CLS] sui vari layer in modo che lo student possa raggiungere una capacità di generalizzazione paragonabile a quella del teacher senza incorrere in costi computazionali insostenibili. Per fare ciò innanzitutto definiamo un insieme I_{pt} come l'insieme degli indici di layer del teacher da distillare nello student, fatto ciò all'interno della loss cercheremo di minimizzare l'errore quadratico medio tra gli stati interni dei due modelli:

$$L_{PT} = \sum_{i=1}^{N}\sum_{j=1}^{M}\left|\left|\frac{\mathbf{h}^s_{i,j}}{||\mathbf{h}^s_{i,j}||_2} - \frac{\mathbf{h}^t_{i,I_{pt}(j)}}{||\mathbf{h}^t_{i,I_{pt}(j)}||_2}\right|\right|^2_2 \tag{2.12}$$

Dove M denota il numero di layer dello student, N denota il numero di esempi di training mentre $\mathbf{h}^s, \mathbf{h}^t$, denotano gli stati interni rispettivamente di student e

teacher. Aggiungendo questa componente alla 2.11 otteniamo la seguente loss finale:

$$L_{PKD} = (1-\alpha)L_{CE}^{s} + \alpha L_{DS} + \beta L_{PT} \tag{2.13}$$

Dove l'iper-parametro β permette di regolare l'importanza delle features dei layer intermedi. Questa metodologia risulta essere particolarmente valida, grazie ad essa infatti, è stato possibile distillare BERT base (12 layer) in una versione di BERT con soli 6 layer, ottenendo risultati paragonabili tra i due modelli sia sui benchmark GLUE [37] che sul test set RACE [22].

2.3 XAI-driven Knowledge Distillation

In [3], viene introdotta una tecnica di distillazione che include nel processo tecniche di *eXplainable AI (XAI)* (termine che racchiude l'insieme delle tecniche e delle metodologie che permettono di spiegare le predizioni dei modelli di intelligenza artificiale). In particolare, viene presentato *DiXtill*, un nuovo metodo di distillazione di Large Language Models. Tale metodologia punta a sfruttare spiegazioni locali fornite da metodi di XAI per guidare il processo di distillazione da un LLM in un modello molto più leggero. Prima di procedere, ed analizzare nel dettaglio tale metodologia, è opportuno introdurre preliminarmente l'explainable AI e la tecnica *Integrated Gradient*.

2.3.1 Integrated Gradient

Il problema di attribuire la predizione di una rete neurale ad un sottoinsieme delle features di input è un problema molto studiato in letteratura. In [33], viene proposta una metodologia chiamata *Integrated Gradient*, che ha i vantaggi di non richiedere modifiche alla rete neurale ed è molto semplice da implementare. Segue una descrizione dettagliata di questa metodologia. Per prima cosa occorre definire formalmente cosa si intende per attribuire la predizione alle feature in input.

Definizione supponiamo di avere una funzione $F : \mathbb{R}^n \to [0,1]$ che rappresenta una rete neurale, e un input $x = \{x_1, x_2, ..., x_n\} \in \mathbb{R}^n$. Una attribuzione della predizione all'input x rispetto ad un input base x' è un vettore che può essere scritto come: $A_F(x,x') = (a_1, a_2, ..., a_n) \in \mathbb{R}^n$, dove a_i è il contributo di x_i sulla predizione $F(x)$.

A questo punto possiamo introdurre due assiomi fondamentali su cui Integrated Gradient si basa.

Assioma di sensitività Un metodo di attribuzione soddisfa l'assioma di *Sensitività* se per ogni coppia (input, baseline) che differiscono per una sola feature, e per i quali la rete produce due predizioni differenti, allora a queste feature dovrebbe essere associata una attribuzione diversa da zero.

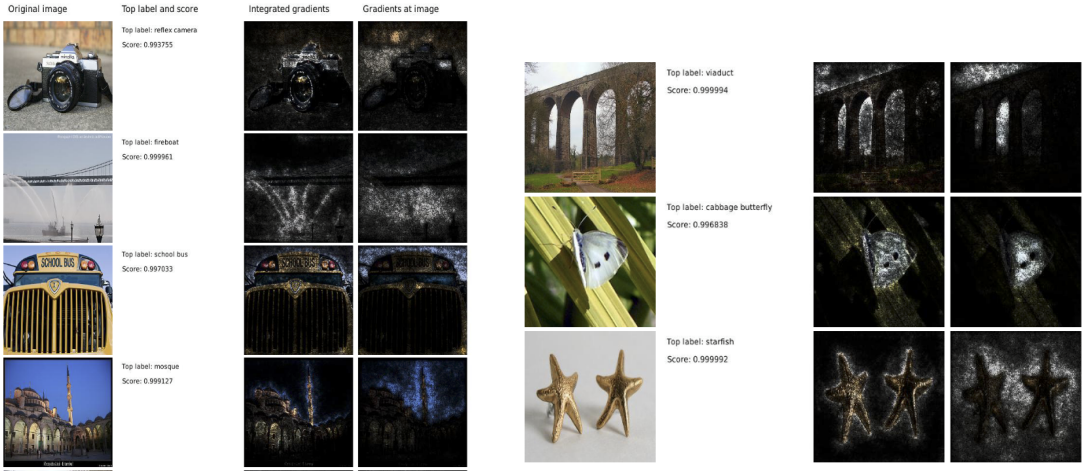


Figura 2.4: Confronto tra la visualizzazione di Integrated Gradient e i gradienti dell'immagine.

Assioma di implementazione invariante Due reti sono *funzionalmente equivalenti* se i loro output sono uguali per tutti gli input, nonostante abbiano implementazioni completamente diverse. Un metodo di attribuzione soddisfa l'assioma di *implementazione invariante* se le attribuzioni sono sempre identiche per due reti equivalenti.

Introdotti questi assiomi, possiamo procedere nel descrivere la metodologia Integrated Gradient. Sia $F : \mathbb{R}^n \to [0,1]$ una funzione che rappresenta una rete neurale, $x \in \mathbb{R}^n$ l'input della rete e $x' \in \mathbb{R}^n$ il baseline input (ad esempio per una rete che processa immagini potrebbe essere una immagine tutta nera oppure per una rete che processa testi potrebbe essere l'embedding nullo). Consideriamo la retta (in $\mathbb{R}^n$) che collega i due input x e x', e calcoliamo il gradiente per ciascun punto su questo path. Gli Integrated Gradient sono ottenuti andando ad accumulare questi gradienti, formalmente, gli Integrated Gradient sono definiti come l'integrale dei gradienti sul path rettilineo che collega x con x'.

L'Integrated Gradient lungo l'i^{th} dimensione di un input x (e di una baseline x') è definito come segue:

$$IntegratedGrads_i(x) = (x_i - x'_i) \times \int_{\alpha=0}^{1} \frac{\partial F(x' + \alpha \times (x - x'))}{\partial x_i} d\alpha \qquad (2.14)$$

Assioma di completezza Gli Integrated Gradients soddisfano l'assioma di *completezza* cioè le attribuzioni si sommano alla differenza tra x e x'. Inoltre vale la seguente:

Proposizione Se $F : \mathbb{R}^n \to \mathbb{R}$ è differenziabile quasi ovunque allora:

$$\sum_{i=1}^{N} IntegratedGrads_i(x) = F(x) - F(x') \qquad (2.15)$$

Per molte reti neurali è possibile trovare una baseline tale che $F(x') \approx 0$, in questi casi quindi avremmo una chiara interpretazione della predizione fornita dalla rete.

Applicare Integrated Gradient Al fine di una applicazione efficiente di questa metodolgia bisogna avere cura di due cose: primo, *scegliere accuratamente la baseline*, dimodoché questa abbia uno score di predizione molto vicino allo zero; e secondo, l'integrale dei gradienti può essere efficientemente approssimato per somma considerando un insieme di punti sufficientemente vicini sul path rettilineo

tra le input e baseline:

$$IntegratedGrads_i^{approx}(x) ::= (x_i - x_i') \times \sum_{k=1}^{m} \frac{\partial F(x' + \frac{k}{m} \times (x - x'))}{\partial x_i} \times \frac{1}{m} \quad (2.16)$$

Dove m è il numero di step nell'approssimazione di Riemann dell'integrale.

Consistenza di Integrated Gradient *Integrated Gradient soddisfa la proprietà di Sensitività* in quanto la completezza implica la Sensitività, inoltre, *le attribuzioni generate da Integrated Gradient soddisfano l'assioma di implementazione invariante* dal momento che queste sono basate solo sul gradiente della funzione rappresentata dalla rete. In figura 2.4, viene mostrato un esempio di visualizzazione di Integrated Gradient, la quale viene comparata alla visualizzazione dei gradienti delle immagini in input, si vede chiaramente come IG generi una visualizzazione migliore.

2.3.2 DiXtill

Introdotta la tecnica Integrated Gradient, possiamo procedere nell'analizzare nel dettaglio la metodologia proposta in DiXtill [3], che integra tale tecnica di XAI all'interno del processo di distillazione.

Modello student Il modello student è una rete *bidirectional Long Short Term Memory*. Tale rete viene inoltre dotata di masked attention per migliorarne le performance di classificazione e renderla facilmente interpretabile (assegnando degli score di attention maggiori a quelle parole che hanno un peso più rilevante nelle predizioni), e di un layer di embedding che fornisce una rappresentazione vettoriale dell'input. Dato un input x composto da una sequenza di k parole $w_1, ..., w_k$, il layer di embedding apprende una sequenza di k vettori $d-dimensionali$ $E = e_1, ..., e_k$ dove $e_i \in \mathbb{R}^d$ è l'embedding della parola w_i, quindi $E \in \mathbb{R}^{d \times k}$ è la matrice di embeddings. Tale matrice viene poi mandata in input alla rete LSTM, che apprende una sequenza di stati interni $h_1, ..., h_k$ (ciascuno ottenuto per concatenazione tra l'embedding calcolato in avanti $\overrightarrow{h_i}$ e quello calcolato all'indietro

$\overleftarrow{h_i}$ ciascuno dei quali è un vettore a u dimensioni). Quindi in output avremo una matrice di stati interni $H \in \mathbb{R}^{2u \times k}$, a questo punto, per ciascuno stato interno in H verrà prodotto uno *score di attention* dalla masked attention. In particolare, viene calcolato un vettore di score σ che determina l'importanza che ciascuna delle k parole ha avuto nel determinare la previsione sul task di classificazione. Tale vettore è calcolato attraverso un meccanismo di attention che si rifà a quello proposto in [1]:

$$\sigma = v^T tanh(U \cdot H) \tag{2.17}$$

Dove $U \in \mathbb{R}^{2u \times 2u}$ è una matrice addestrabile che effettua una proiezione lineare di H mentre $v \in \mathbb{T}^{2u}$ è un vettore addestrabile utilizzato per calcolare il vettore finale $\sigma \in \mathbb{R}^k$

Incorporare le spiegazioni nel processo di distillazione In DiXtill, viene usata una tecnica di spiegazione post-hoc (*Integrated Gradient*) per calcolare, per ogni istanza di training x, una spiegazione $\mathcal{E}(x)$ della predizione del teacher. Tale spiegazione deve essere una lista $(w_i : \sigma_i^T)$ in cui ciascun σ_i^T rappresenta una misura di quanto la parola w_i ha influenzato la predizione del teacher. Una volta che questa spiegazione è stata calcolata, può essere sfruttata per guidare il processo di distillazione. DiXtill propone una versione modificata della loss standard per la distillazione, introducendo un termine *XAI-based* $\mathcal{L}_{XAI}$, che pruomove l'allineamento tra le spiegazioni fornite dal teacher e quelle fornite dallo student usando la distanza del coseno. Siano $\mathcal{E}^T(x)$ e $\mathcal{E}^S(x)$ le spiegazioni rispettivamente di teacher e student su un certo input x, estratti i due vettori σ^S e σ^T possiamo formalizzare la componente XAI-based come segue:

$$\mathcal{L}_{XAI} = \frac{1}{2}\left(1 - \frac{\sigma^T \cdot \sigma^S}{||\sigma^T||||\sigma^S||}\right) \tag{2.18}$$

A questo punto la loss totale sarà data da 3 contributi:

$$\mathcal{L} = (1-\alpha)\mathcal{L}_{CE} + \alpha(\mathcal{L}_{KD} + \mathcal{L}_{XAI}) \tag{2.19}$$

- $\mathcal{L}_{CE}$ è la cross-entropy loss calcolata sulle etichette contenute nel dataset pesata secondo un fattore $(1-\alpha)$
- $\mathcal{L}_{KD}$ è la loss di distillazione standard, rappresentata da una versione normalizzata della distanza di Kullback–Leibler:

$$1-\exp\Big(-KL\big(p^T(\tau),p^S(\tau)\big)\Big) \tag{2.20}$$

  pesata secondo un fattore α
- $\mathcal{L}_{XAI}$ è il termine XAI-driven appena descritto.

Tale tecnica avanzata di distillazione ha dimostrato la sua validità migliorando in termini di performance rispetto ai metodi di distillazione classici e raggiungendo prestazioni comparabili a quelle del modello teacher sul task di classificazione. Uno dei vantaggi chiave di questa tecnica, e in generale delle tecniche di distillazione cross-architecture, è che distillando il modello teacher in uno student che ha una architettura diversa si possono utilizzare modelli student con archietture molto più leggere, pur mantenendo prestazioni paragonabili a quelle del teacher grazie a strategie come quella appena descritta. Di conseguenza, le tecniche cross-architecture generalmente permettono di avere fattori di compressione più grandi rispetto a tecniche di tipo diverso.

2.4 Knowledge Distillation tramite Meta-Learning

Le tecniche classiche di distillazione, comprese quelle appena descritte, hanno tutte una caratteristica in comune: il modello teacher non viene mai coinvolto nella propagazione del gradiente. Questa caratteristica, nonostante abbia il vantaggio di rendere il processo di distillazione molto veloce, tende a rendere questo processo meno accurato di quanto potrebbe potenzialmente essere. In [41], viene proposta una nuova metodologia di distillazione, che fonde insieme le classiche tecniche di distillazione e tecniche di *Meta-Learning* in un framework di training congiunto in cui il modello student impara dal teacher a performare il task di downstream,

e allo stesso tempo, il teacher impara dallo student come distillare meglio la conoscenza da esso posseduta. Prima di procedere nel vedere più nel dettaglio il funzionamento di questa metodologia, vale la pena di dare uno sguardo più da vicino al concetto di Meta-Learning, con lo scopo di comprendere meglio quanto discuteremo di seguito.

2.4.1 Model-Agnostic Meta-Learning

L'obiettivo del meta learning è di addestrare un modello su una varietà di task diversi di learning, dimodoché esso possa successivamente imparare a risolvere nuovi task utilizzando un numero limitato di esempi. In [6], viene proposto un algoritmo di meta learning che non dipende da un particolare tipo di modello, e quindi che può essere definito *model agnostic* nel senso che si può applicare a qualsiasi tipo di modello addestrato utilizzando la discesa del gradiente. L'idea sottostante alla metodologia proposta è quella di addestrare i parametri iniziali di un modello in modo che questo abbia performance massimali su un nuovo task dopo uno o più step di aggiornamenti tramite gradiente su un numero limitato di esempi. Diversamente dalle tradizionali tecniche di meta learning, questa non apprende funzioni di aggiornamento o regole di apprendimento, inoltre, non impone alcun vincolo sull'architettura del modello; ulteriore flessibilità è data dal fatto che questo algoritmo si può integrare con diversi tipi di funzioni di loss (ad esempio quelle differenziabili tipiche del learning supervisionato o quelle non differenziabili tipiche del reinforcement learning). Il processo di addestrare i parametri di un modello in modo che esso sia semplice da adattare a nuovi task, attraverso pochi (eventualmente anche solo uno) [2] aggiornamenti del gradiente, può essere visto come la costruzione di una rappresentazione interna dei parametri che sia ampia e adatta a vari task. Questo potrebbe tradursi in modelli per cui è sufficiente, ad esempio, sottoporre a fine tuning solo i layer più vicini all'output per ottenere buoni risultati.

Definizione di un problema di Meta-Learning Consideriamo un modello f, che mappa esempi in input x in output a. Durante il processo di meta-learning il

modello è addestrato perchè sia abile ad adattarsi ad un numero grande (potenzialmente infinito) di task. Formalmente, un generico task di learning può essere scritto come $\mathcal{T} = \{\mathcal{L}(x_1, a_1, ..., x_H, a_H), q(x_1), q(x_{t+1}|x_t, a_t), H\}$ cioè consiste di un loss function $\mathcal{L}$, una distribuzione sulle osservazioni iniziali $q(x_1)$, una distribuzione di transizione $q(x_{t+1}|x_t, a_t)$ e una lunghezza della sequenza H (in problemi di learning supervisionato con variabili identiche ed identicamente distribuite avremo $H = 1$). La loss $\mathcal{L}(x_1, a_1, ..., x_H, a_H) \rightarrow \mathbb{R}$, fornisce un feedback specifico per il task che potrebbe avere la forma di un errore di misclassificazione oppure di una funzione di costo in un processo Markoviano. In uno scenario di meta-learning, vorremmo che il modello sia capace di adattarsi ad una distribuzione sui task $p(\mathcal{T})$. In un contesto di K-shot learning, il modello è addestrato per apprendere un nuovo task $\mathcal{T}_i$, campionato da $p(\mathcal{T})$ su un numero K di esempi campionati secondo q_i usando la loss function $\mathcal{L}_{\mathcal{T}_i}$. Durante il processo di meta-training, viene campionato un task $\mathcal{T}$ da $p(\mathcal{T})$, il modello è addestrato su K esempi e sul feedback dato dalla loss $\mathcal{L}_{\mathcal{T}_i}$, successivamente viene testato su nuovi esempi per lo stesso task. Il modello f viene dunque migliorato andando a considerare come varia l'errore sui test su dati nuovi campionati da q_i rispetto ai parametri del modello. In effetti, l'errore sul test per un certo stask campionato $\mathcal{T}_i$, funge da training error per il processo di meta-learning. Alla fine del processo di meta-training, vengono campionati dei nuovi task da $p(\mathcal{T})$, e le performance del modello vengono valutate sulla base di quanto il modello riesce ad essere performate dopo essere stato addestrato con soli K esempi su questi task.

Algoritmo Model-Agnostic Meta-Learning L'idea alla base di questo algoritmo è che alcune rappresentazioni interne (in termini di parametri), risultano essere maggiormente trasferibili di altre. Ad esempio, una rete neurale potrebbe apprendere delle features che sono universalmente applicabili a tutti i task in $p(\mathcal{T})$, e non solo a task individuali. Quello che sappiamo sicuramente su come verrà fatto il finetuning del modello su un nuovo task, è che sarà utilizzata la regola di learning del gradiente; pertanto, l'algoritmo cerca di addestrare il modello in modo che questo sia predisposto per una veloce discesa del gradiente verso il valore ottimo

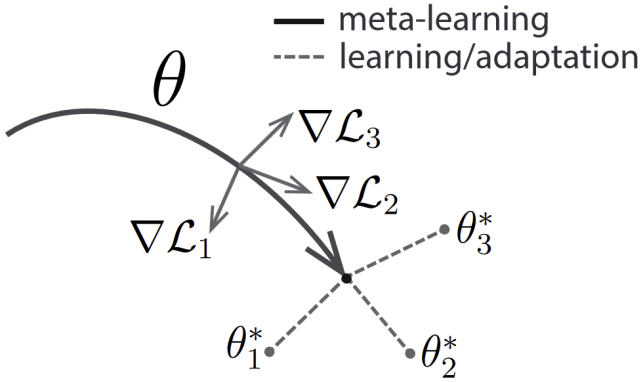


Figura 2.5: Diagramma dell'algoritmo di model agnostic meta learning (MAML) che ottimizza i parametri θ dimodoché essi siano facilmente adattabili a nuovi task [6].

dei parametri per un certo task campionato da $p(\mathcal{T})$. Questo si traduce nel trovare dei parametri che siano estremamente sensibili a cambiamenti nel task, in modo che piccole variazioni nei parametri possano provocare grossi migliormenti sulla funzione di loss. Nella figura 2.5 viene mostrato graficamente questo processo. Dunque, le uniche assunzioni che questa tecnica fa sono le seguenti: il modello è parametrizzato da un vettore di parametri θ, è possibile utilizzare la discesa del gradiente. Formalmente, indicando il modello con f_θ, quando adattiamo il modello per un certo task $\mathcal{T}_i$, i parametri θ diventano θ_i' per effetto degli aggiornamenti subiti. θ_i' viene calcolato attraverso uno o più step di aggiornamento del gradiente sul task $\mathcal{T}_i$, in formule avremo quanto segue:

$$\theta_i' = \theta - \alpha\nabla_\theta\mathcal{L}_{\mathcal{T}_i}(f_\theta) \tag{2.21}$$

Dove α è il learning rate, che può essere fissato come iperparametro oppure meta-appreso a sua volta. Facendo l'assunzione esemplificativa (ma senza perdita in generalità) di avere un solo aggiornamento del gradiente, possiamo scrivere la funzione obiettivo del processo di meta learning sarà la seguente:

$$\min_\theta \sum_{\mathcal{T}_i\sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'}) = \sum_{\mathcal{T}_i\sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}\big(f_\theta - \alpha\nabla_\theta\mathcal{L}_{\mathcal{T}_i}(f_\theta)\big) \tag{2.22}$$

Algorithm 1 Model-Agnostic Meta-Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ with respect to K examples
6: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
7: **end for**
8: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$
9: **end while**

Figura 2.6: Algoritmo MAML [6]

Quindi, i parametri del modello sono addestrati ottimizzando le performance di $f_{\theta'}$ su vari task campionati da $p(\mathcal{T})$. E' interessante notare come la meta-ottimizzazione viene effettuata sui parametri θ mentre la funzione obiettivo viene calcolata sui parametri specifici per il task θ_i', questo proprio perchè lo scopo è addestrare il modello in modo che esso sia predisposto al meglio per il finetuning. La meta-ottimizzazione sui task viene effettuata trami SGD (*Stochastic Gradient Descent*), in modo che θ venga aggiornato secondo la seguente:

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T} \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'}) \tag{2.23}$$

Dunque, l'aggiornamento dei parametri θ tramite una operazione che potremmo definire di *meta-gradient update*, richiede un secondo step di *backpropagation* su f per il calcolo di prodotti vettoriali che coinvolgono la matrice Hessiana. L'algoritmo completo viene illustrato in figura 2.6.

2.4.2 Learning to Teach

Una volta introdotto e discusso il concetto di meta learning, possiamo discutere come questo possa essere integrato nel processo di distillazione per renderlo

più accurato. In particolare, tratteremo nel dettaglio quanto proposto in [41] in quanto strettamente correlato a quanto proposto nel presente elaborato. Sicuramente, un aspetto chiave dell'integrazione del meta-learning all'interno di una tecnica di distillazione è quello di poter fornire al teacher un meccanismo di feedback, attraverso il quale questo possa correggersi e rendere la propria conoscenza maggiormente distillabile. In un contesto di meta-learning, il teacher prende il nome di *meta-learner* (poichè esso impara ad insegnare), mentre lo student prende il nome di *inner-learner*. Ovviamente, coinvolgere il teacher come meta-learner, introduce nel processo un overhead computazionale dovuto all'aggiornamento dei suoi parametri; per cui, vediamo brevemente quali sono le principali motivazioni che rendono accettabile questo costo. In primo luogo, possiamo osservare che il teacher potrebbe non essere *"consapevole"* delle capacità dello student, questa motivazione deriva da alcuni studi condotti in pedagogia, che hanno osservato come un processo di learning costruito intorno alle capacità dello student porta a risultati migliori. In aggiunta, il teacher non è ottimizzato per la distillazione, infatti, il processo di pretraining e di finetuning di un LLM si focalizza sul far apprendere al modello una conoscenza che gli permetta di performare bene sul task di downstream e non che gli permetta di trasferire tale conoscenza in modo agevole. Segue una descrizione dettagliata di MetaDistil [41], una metodologia che introduce tecniche di meta-learning nel processo di distillazione.

MetaDistil Per ciascuno step di training, viene creata una copia S' dello student S, e tale copia viene aggiornata secondo una comune loss di distillazione. In questo modo, si ottiene uno "student sperimentale" al quale viene sottoposto un "quiz", e a questo punto si calcola la loss di questo student sperimentale sul quiz set (un insieme di esempi che non sono né nel training set e né nel test set). Tale loss, viene usata come feedback per il calcolo delle derivate del secondo ordine e per il conseguente aggiornamento del teacher. Alla fine, lo studente sperimentale viene scartato e si utilizza il teacher aggiornato per distillare la conoscenza all'interno dello student reale, poi il processo ricomincia con il prossimo step di training. L'utilizzo del meta-learning consente al teacher di ricevere un feed-

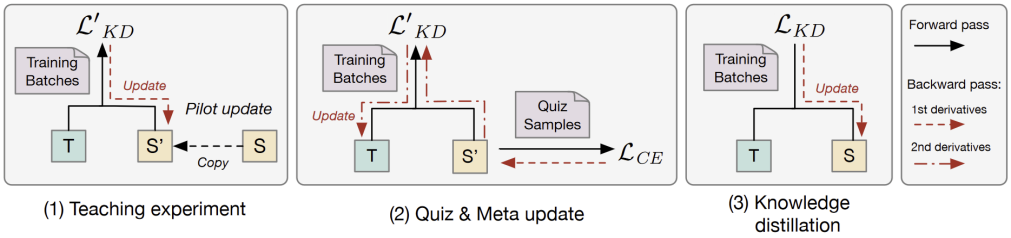


Figura 2.7: MetaDistil workflow [41]: (1) viene effettuata una prova di distillazione nello student sperimentale; (2) Si valutano le prestazioni dello student sperimentale sul quiz set e si aggiorna il teacher sulla base di questo feedback; (3) Si scarta lo student sperimentale e si utilizza il teacher aggiornato per distillare la conoscenza nello student reale.

back dallo student in modo completamente differenziabile. Il workflow appena descritto è rappresentato graficamente in figura 2.7.

Vediamo più formalmente la metodologia proposta, e per fare ciò, partiamo dalla forma generale di una loss di distillazione. Dato un trainset di cardinalità N della forma:$\mathcal{D} = \{(x_1, y_1), ..., (x_N, y_N)\}$ possiamo scrivere la loss di distillazione come:

$$\mathcal{L}_S(\mathcal{D}; \theta_S; \theta_T) = \frac{1}{N} \sum_{i=1}^{N} \left[\alpha \mathcal{L}_{\mathcal{T}}(y_i, S(x_i; \theta_S)) + (1-\alpha) \mathcal{L}_{KD}(T(x_i; \theta_T), S(x_i; \theta_S)) \right] \tag{2.24}$$

Dove α è un iperparametro che regola l'importanza dei due termini, θ_S e θ_T sono i parametri rispettivamente di student e teacher, $\mathcal{L}_{\mathcal{T}}$ si riferisce alla loss specifica per il task di downstream e $\mathcal{L}_{KD}$ si riferisce alla loss di distillazione. Concretamente, l'obiettivo da ottimizzare del teacher sono le performance dello student dopo essere stato sottoposto a distillazione dal teacher stesso. In questo contesto, lo student θ_S rappresenta l'inner-learner, e il teacher θ_T rappresenta il meta-learner. Una volta creata una copia sperimentale dello student, i suoi parametri vengono aggiornati secondo la seguente:

$$\theta'_S(\theta_T)) = \theta_S - \lambda \nabla_{\theta_S} \mathcal{L}_S(x; \theta_S; \theta_T) \tag{2.25}$$

Come possiamo osservare, lo student sperimentale θ'_S, così come anche la loss sul

Require: student θ_S, teacher θ_T, train set $\mathcal{D}$, quiz set $\mathcal{Q}$
Require: λ, μ: learning rate for the student and the teacher
1: **while** not done **do**
2: Sample batch of training data $\boldsymbol{x} \sim \mathcal{D}$
3: Copy student parameter θ_S to student θ'_S
4: Update θ'_S with $\boldsymbol{x}$ and θ_T: $\theta'_S \leftarrow \theta'_S - \lambda \nabla_{\theta'_S} \mathcal{L}_S(x; \theta_S; \theta_T)$
5: Sample a batch of quiz data $\boldsymbol{q} \sim \mathcal{Q}$
6: Update θ_T with $\boldsymbol{q}$ and θ'_S: $\theta_T \leftarrow \theta_T - \mu \nabla_{\theta_T} \mathcal{L}_{\mathcal{T}}(\boldsymbol{q}, \theta'_S(\theta_T))$
7: Update original θ_S with $\boldsymbol{x}$ and the updated θ_T: $\theta_S \leftarrow \theta_S - \lambda \nabla_{\theta_S} \mathcal{L}_S(x; \theta_S; \theta_T)$
8: **end while**

Figura 2.8: Algoritmo MetaDistil [41]

quiz set $l_q = \mathcal{L}_{\mathcal{T}}(\mathbf{q}, \theta'_S(\theta_T))$ calcolata su un batch $\mathbf{q}$ campionato dal quiz set $\mathcal{Q}$, è una funzione dei parametri del teacher θ_T. Di conseguenza, è possibile ottimizzare l_q rispetto a θ_T usando un learning rate μ:

$$\theta_T \leftarrow \theta_T - \mu \nabla_{\theta_T} \mathcal{L}_{\mathcal{T}}(\mathbf{q}, \theta'_S(\theta_T)) \tag{2.26}$$

L'utilizzo di un quiz set separato è quindi fondamentale al fine di evitare l'overfitting sul validation set. In figura 2.8 viene presentato sotto forma di pseudo-codice l'intero processo di distillazione descritto.

Capitolo 3

Distillazione parameter-efficient di LLM tramite meta-learning

Nel precedente capitolo, abbiamo descritto alcune tra le tecniche più avanzate per la distillazione di LLM. Dalla discussione di queste tecniche, tra le altre cose, è emerso che l'utilizzo del meta-learning in combinazione alle tecniche di distillazione ne migliora significativamente la validità. L'utilizzo del meta-learning, però, costituisce anche un elemento di inefficienza all'interno del processo; infatti, l'aggiornamento del modello teacher tramite *backpropagation racchiude l'aggiornamento di centinaia di milioni e in alcuni casi miliardi di parametri.* Collocandosi nel contesto appena descritto, il presente elaborato si pone l'obiettivo di proporre una metodologia di distillazione che faccia utilizzo del meta-learning, per beneficiarne dei vantaggi, ma allo stesso tempo ne limiti il più possibile le inefficienze. Per fare ciò, è stata formulata una strategia di *parameter-efficient meta-learning*, che include all'interno del processo di distillazione una tecnica di meta-learning resa più efficiente da tecniche di *parameter-efficient finetuning (PEFT)* (in particolare la *Low-Rank Adaptation (LoRA)*). Così facendo, ci si aspetta di ottenere una metodologia di distillazione che usi il meta-learning per rendere la conoscenza del teacher meglio distillabile e tecniche di PEFT per rendere il processo di feedback più efficiente. Il presente capitolo sarà così strutturato: nella prima parte vengono introdotte tutte le nozioni preliminari (ad esempio le principali tecniche di PEFT),

mentre nel prosieguo verrà illustrata nel dettaglio la metodologia proposta.

3.1 Parameter-Efficient Finetuning (PEFT)

I Large Language Model hanno dimostrato di avere una grande capacità di generalizzazione, che permette loro di sfruttare la conoscenza acquisita per svolgere nuovi task che non erano inclusi nel processo di pre-training: tale abilità è conosciuta in letteratura con il termine *zero-shot learning*. Tuttavia, il finetuning rimane comunque una operazione essenziale per migliorare ulteriormente questi modelli e personalizzarli su nuovi dati e/o task [12]. A causa della loro enorme scala, una tecnica che viene spesso usata per il finetuning di LLM è quella di selezionare un sottoinsieme sufficientemente piccolo di pesi da aggiornare durante il processo di finetuning. Le strategie di PEFT possono essere complessivamente classificate in quattro categorie [12]: *additive PEFT* che modifica l'architettura del modello iniettando in esso nuovi parametri addestrabili, *selective PEFT* che seleziona un sottoinsieme dei parametri del modello da rendere addestrabili durante il finetuning, *reparameterized PEFT* che trasforma i parametri del modello in uno spazio a bassa dimensionalità (tramite riparametrizzazione) per il training per poi riportarlo alla forma originale per l'inferenza, e *hybrid PEFT* che combina varie tecniche di PEFT insieme. Di seguito verranno discusse brevemente le principali tecniche afferenti a ciascuna di queste categorie.

Additive PEFT Tali tecniche consistono nel mantenere invariata la backbone del modello da sottoporre a finetuning, e contestualmente, iniettare in posizioni strategiche dei nuovi parametri addestrabili. La tecnica *Adapter*, introdotta per la prima volta in [16], consiste nell'inserimento all'interno di un blocco Transformer di un piccolo layer aggiunto chiamato appunto *adapter*. Tipicamente, un adapter consiste di un matrice di proiezione in uno spazio a bassa dimensionalità $W_{down} \in \mathbb{R}^{r \times d}$, seguita da una attivazione non lineare $\sigma(\cdot)$, e da una seconda matrice di proiezione $W_{up} \in \mathbb{R}^{d \times r}$ che porta l'output nuovamente nello spazio di partenza. In questo contesto, d rappresenta la dimensione del layer latente mentre

r è la dimensione dello spazio di proiezione a bassa dimensionalità. Formalmente possiamo descrivere il funzionamento di un adapter secondo la seguente:

$$Adapter(x) = W_{up}\sigma(W_{down} \cdot x) + x \tag{3.1}$$

Il *soft prompt* rappresenta un approccio alternativo per raffinare i modelli attraverso il finetuning. Invece di ottimizzare le rappresentazioni discrete dei token attraverso l'in-context learning, prevale la convinzione che lo spazio di embedding nel continuo possegga un maggiore contenuto informativo. Per cui, alla sequenza in input viene appeso un vettore addestrabile chiamato *soft prompt* che può essere rappresentato come segue:

$$x^{(l)} = [s_1^{(l)}, ..., s_{N_S}^{(l)}, x_1^{(l)}, ..., x_{N_X}^{(l)}] \tag{3.2}$$

$X^{(l)}$ rappresenta l'input del layer l, in cui, $s_i^{(l)}$ sono i token del soft prompt e $x_i^{(l)}$ sono i token dell'input originale. *Il prefix tuning* [24] è una tecnica che prevede l'introduzione di vettori addestrabili da anteporre alle matrici di chiavi e valori nei Transformer layer. Per assicurare la stabilità del processo di training, in genere si usa una strategia di riparametrizzazione che utilizza un layer MLP (fully connected) per generare questi prefissi piuttosto che ottimizzare essi direttamente. Alla fine del processo solo i vettori prefisso vengono conservati per l'inferenza.

Selective PEFT Dato un modello con parametri $\theta = \{\theta_1, ..., \theta_n\}$, il processo di selective PEFT è rappresentato dall'applicazione di una maschera binaria $M = \{m_1, ..., m_n\}$ a questi parametri. Ciascun $m_i \in M$ vale 1 se il parametro θ_i è stato selezionato per il finetuning, oppure 0 altrimenti. Per cui ciascun parametro sarà aggiornato secondo la seguente:

$$\theta_i' = \theta_i - \eta \cdot m_i \cdot \frac{\partial \mathcal{L}}{\partial \theta_i} \tag{3.3}$$

In questa formulazione, solo i parametri θ_i per cui $m_i = 1$ saranno interessati dal processo di backpropagation. *Diff Pruning* [11] è una tecnica che applica una ma-

schera binaria addestrabile ai pesi del modello durante il finetuning, e inoltre, tale maschera viene regolarizzata secondo una approssimazione differenziabile della norma L_0. *FishMask* [34] è una tecnica che sfrutta un'approssimazione dell'informazione di Fisher per determinare l'importanza dei parametri, per poi prendere i top-k parametri (secondo questa informazione) per formare la maschera M. Queste sono solo alcune delle tecniche di selective PEFT presenti in letteratura ma ne esistono molte altre. Più in generale queste tecniche si suddividono in due categorie: *structured parameters masking*, cioè tutte quelle tecniche che applicano le maschere ai pesi secondo dei pattern regolari e ricorrenti; e *unstructured parameters masking*, ciò quell'insieme di tecniche che applicano invece le maschere ai pesi randomicamente.

Reparameterized PEFT La riparametrizzazione consiste nel trasformare l'architettura di un modello in un'altra equivalente trasformandone i parametri. In un contesto PEFT questo si traduce nel riparametrizzare i parametri dei modelli, dimodoché, il training sia più efficiente. Dopo il training, il modello può eventualmente essere riportato alla sua parametrizzazione originaria. La più famosa tra queste tecniche è la *Low-Rank Adaptation* [17], che si basa sull'aggiunta di una matrice ΔW che fornisce la conoscenza aggiuntiva specifica per il singolo task. Tale tecnica è parte integrante della metodologia proposta nel presente elaborato, e pertanto, verrà discussa nel dettaglio nel seguito di questo capitolo. La tecnica *Compacter* [27] è un ulteriore esempio di questa categoria, e introduce due moduli Adapter leggeri tramite la parametrizzazione di W_{down} e W_{up} come $W = \sum_{i=1}^{n} A_i \bigotimes B_i$ dove $A_i \in \mathbb{R}^{n \times n}$, $B_i \in \mathbb{R}^{\frac{r}{n} \times \frac{d}{n}}$ e $\bigotimes$ denota il prodotto di Kronecker 3.4. Inoltre, il numero di parametri viene ulteriormente ridotto rendendo A_i una matrice di pesi condivisi e riparametrizzando B_i come il prodotto di due matrici di basso rango; in questo modo, la complessità dei parametri viene ridotta da $O(r \cdot d)$ a $O(r + d)$.

Prodotto di Kronecker

Date due matrici $A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{p\times q}$ il prodotto di Kronecker $\bigotimes$ tra A e B è una matrice a blocchi di dimensione $mp \times nq$ definita come:

$$A \bigotimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix} \tag{3.4}$$

Esplicitando tutti i termini avremo:

$$A \bigotimes B = \begin{bmatrix} a_{11}b_{11} & \cdots & a_{11}b_{1q} & \cdots & \cdots & a_{1n}b_{11} & \cdots & a_{1n}b_{1q} \\ a_{11}b_{21} & \cdots & a_{11}b_{2q} & \cdots & \cdots & a_{1n}b_{21} & \cdots & a_{1n}b_{2q} \\ \vdots & \ddots & \vdots & & & \vdots & \ddots & \vdots \\ a_{11}b_{p1} & \cdots & a_{11}b_{pq} & \cdots & \cdots & a_{1n}b_{p1} & \cdots & a_{1n}b_{pq} \\ \vdots & & \vdots & \ddots & & \vdots & & \vdots \\ \vdots & & \vdots & & \ddots & \vdots & & \vdots \\ a_{m1}b_{11} & \cdots & a_{m1}b_{1q} & \cdots & \cdots & a_{mn}b_{11} & \cdots & a_{mn}b_{1q} \\ a_{m1}b_{21} & \cdots & a_{m1}b_{2q} & \cdots & \cdots & a_{mn}b_{21} & \cdots & a_{mn}b_{2q} \\ \vdots & \ddots & \vdots & & & \vdots & \ddots & \vdots \\ a_{m1}b_{p1} & \cdots & a_{m1}b_{pq} & \cdots & \cdots & a_{mn}b_{p1} & \cdots & a_{mn}b_{pq} \end{bmatrix} \tag{3.5}$$

Notiamo che questo prodotto **non** è una estensione del prodotto riga per colonna.

Hybrid PEFT L'efficacia dei vari metodi di PEFT può differire significativamente al seconda del task a cui vengono applicate, e per questo, di recente molti studi hanno cercato di combinare insieme diverse tecniche di PEFT per beneficiare insieme dei vantaggi di ciascuna di esse. La tecnica *UniPELT* [28] integra insieme LoRA, prefix-tuning e adapter all'interno di un blocco Transformer; e inoltre, per controllare quale di questi verrà attivato, viene introdotto un meccanismo di gating composto da tre reti feed forward di piccole dimensioni. Ciascuna di queste reti produce uno scalare $\mathcal{G} \in (0,1)$ che vengono poi rispettivamente applicati alle

matrici di LoRA, prefix-tuning e adapter. Tecniche di PEFT ibrido sono state studiate anche nel contesto degli LLM, ad esempio, il framework *LLM-Adapters* [18] incorpora varie tecniche di PEFT negli LLM. Tale studio ha dimostrato, tra le altre cose, che l'uso di tecniche di PEFT permette a piccoli LLM di eguagliare in performance le controparti più grandi su alcuni task.

3.1.1 Low-Rank Adaptation (LoRA)

La tecnica *Low-Rank Adaptation* [17] è una tecnica di reparameterized PEFT, che consiste nel *congelare* i parametri del modello (renderli non soggetti ad aggiornamento tramite backpropagation) ed iniettare in esso (a livello dei blocchi Transformer) una decomposizione di matrici di basso rango addestrabile, col fine di ridurre il numero di parametri addestrabili per il task di downstream. Le motivazioni alla base dello sviluppo di questa tecnica sono molteplici:

- Aggiornando solo le matrici LoRA durante il finetuning, il modello di partenza può essere riutilizzato per un numero qualsiasi di task diversi semplicemente sostituendo le matrici LoRA nei blocchi Transformer.
- Aggiornando solo le matrici LoRA, il finetuning del modello diventa più veloce, meno energivoro, e richiede meno risorse computazionali
- LoRA può essere combinata semplicemente con altre tecniche di PEFT come prefix-tuning e Adapters.

Ai fini del presente elaborato, segue una descrizione formale e dettagliata del funzionamento di tale tecnica.

Formulazione del problema Supponiamo di avere un modello autoregressivo, pre-addestrato su un task di language modeling, $P_\phi(y|z)$, parametrizzato da ϕ. L'obiettivo è quello di adattare questo modello su dei task di generezione condizionata di testo (ad esempio summarization, machine traslation etc.). Ciascun task è rappresentanto come $\mathcal{Z} = \{(x_i, y_i)\}_{i=1,\ldots,N}$, dove sia x_i che y_i sono sequenze di token. In un processo di full finetuning quello che avviene è che il modello

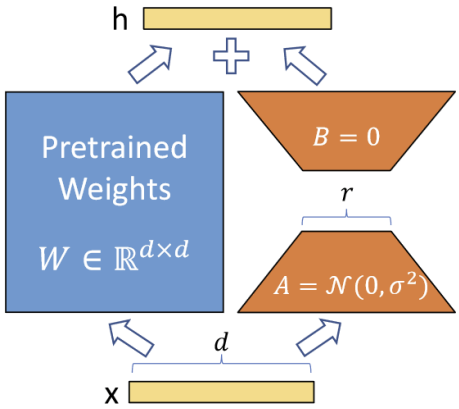


Figura 3.1: Illustrazione grafica della riparametrizzazione applicata da LoRA [17]

viene inizializzato con i parametri di pre-training ϕ_0, e ad ogni iterazione avremo che $\phi_i = \phi_{i-1} + \Delta\phi$ (utilizzando la discesa del gradiente), dimodoché, venga massimizzata la seguente funzione obiettivo:

$$\max_{\phi} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y|} \log(P_\phi(y_t|x,y_{<t})) \tag{3.6}$$

Il problema di questo processo è che per ogni task di downstream bisogna apprendere un insieme di parametri ϕ' con $|\phi'| = |\phi_0|$, il ché rappresenta un problema per modelli molto grandi. Per sopperire a ciò, LoRA propone di codificare l'incremento dei parametri cone $\Delta\phi = \Delta\phi(\theta)$, con $|\theta| \ll |\phi_0|$. La funzione obiettivo si trasforma come segue:

$$\max_{\theta} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y|} \log(P_{\phi_0+\Delta\phi(\theta)}(y_t|x,y_{<t})) \tag{3.7}$$

Metodologia Consideriamo una matrice di pesi pre-addestrata $W_0 \in \mathbb{R}^{d\times k}$, i pesi adattati ad un qualsiasi task possono essere rappresentati come $W_0 + \Delta W$, LoRA vincola questo incremento ad essere dato dal prodotto di due matrici a più basso rango $W_0 + \Delta W = W_0 + BA$, dove $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$ con $r \ll \min(d,k)$. I

parametri W_0 non vengono coinvolti negli aggiornamenti del gradiente, gli unici parametri che vengono aggiornati sono quelli contenuti nelle matrici A e B. Il passo di forward, su input x, sarà calcolato come segue:

$$h = W_0 x + BAx \tag{3.8}$$

La matrice A viene inizializzata in accordo ad una distribuzione normale random $\mathcal{N}(0, \sigma^2)$, mentre la matrice B viene inizializzata a zero. Notiamo che per non pagare costi addizionali in fase di inferenza, è possibile salvare direttamente la matrice $W = W_0 + BA$, potendo recuperare in qualsiasi momento la matrice W_0 invertendo la precedente equazione. La metodologia è illustrata graficamente in figura 3.1

3.2 Metodologia proposta

Veniamo dunque alla descrizione della metodologia di distillazione proposta. Quanto trattato finora, costituisce il background necessario a introdurre quanto discuteremo nel presente capitolo, cioè una procedura *iterativa* di distillazione che sia capace di combinare insieme Meta-Learning, PEFT e metodi classici di distillazione. La presente proposta, costituisce un tentativo di rendere più efficiente il processo di distillazione con feedback, permettendo di effettuare l'aggiornamento dei pesi del teacher sulla base del feedback fornito dallo student, senza pagare in toto il costo che ne consegue. Nel seguito descriveremo per prima cosa le scelte progettuali che si sono dovute intraprendere sia per il modello teacher che per il modello student, e successivamente daremo una descrizione dettagliata della procedura di distillazione proposta.

3.2.1 Scelte architetturali

Passiamo dunque alla discussione delle scelte architetturali riguardanti sia il modello teacher che il modello student. Il presente elaborato intende proporre una metodologia di distillazione che non dipenda dall'architettura dei due mo-

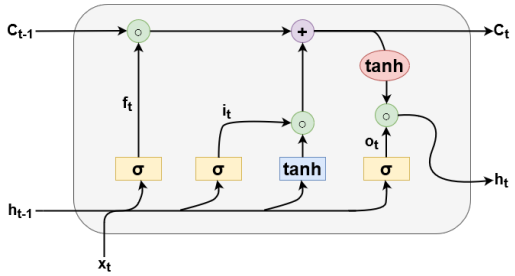


Figura 3.2: Architettura Long Short-Term Memory (LSTM)

delli e che non imponga vincoli su tali architetture. Tuttavia, dovendo fare delle scelte, ai fini di questo elaborato, si è scelto di effettuare una distillazione cross-architecture, e pertanto, si è scelto di adottare una architettura *Transformer* per quanto riguarda il teacher e una architettura *Long Short-Term Memory (LSTM)* per quanto riguarda lo student (scelta a seguito del successo che questo tipo di student ha ottenuto in letteratura, ad esempio in [3]).

Modello Student Per il modello student si è scelto di usare una LSTM bidirezionale, cioè la sequenza in input viene analizzata sia partendo dal token in posizione zero e procedendo in avanti, che partendo dal token finale e procedendo all'indietro. Inoltre, dal momento che il nostro scopo è quello di distillare un modello molto grande in uno molto più piccolo, scegliamo di dotare lo student di un singolo layer. Sempre per mantenere compatta la dimensione del modello distillato, si è scelto, per il layer di embedding antestante al modello LSTM una dimensione pari a 50.

Long Short-Term Memory (LSTM)

Le reti LSTM sono una archiettura che propone una soluzione agli storici problemi di cui sono afflitte le reti neurali riccorrenti (RNN), come il gradiente che può *"esplodere"* (essere talmente grande da provocare oscillazioni dannose nei pesi), o ancora peggio essere *"evanescente"* (dissolversi durante la retropropagazione lungo la sequenza temporale). L'architettura LSTM nasce proprio per proporre una soluzione a tali problematiche. L'idea alla base di questa architettura è che sia opportuno dotare la rete di una "cella di memoria", sulla quale è possibile leggere, scrivere e cancellare il contenuto. Per regolare le operazioni sulla cella vengono introdotti alcuni **gate**, e ad ogni timestep, una parte del contenuto della cella verrà cancellato, una parte verrà mantenuto, e parte del contenuto dell'input corrente verrà memorizzato all'interno di essa per i timestep successivi. Formalmente, indichiamo con: i_t l'*input gate*, che regola quali informazioni mantenere dall'input corrente; f_t il *forget gate*, che determina quali informazioni cancellare dalla cella; o_t l'*output gate*, che determina quali informazioni devono comporre il prossimo stato interno; h_t rappresenta lo stato interno della rete al timestep t; x_t rappresenta l'input corrente. A questo punto, il funzionamento di una LSTM può essere descritto secondo le seguenti equazioni.

$$\begin{aligned} i_t &= \sigma(W_i h_{t-1} + U_i x_t + b_i) \\ f_t &= \sigma(W_f h_{t-1} + U_f x_t + b_f) \\ o_t &= \sigma(W_o h_{t-1} + U_o x_t + b_o) \\ \tilde{c}_t &= \tanh(W h_{t-1} + U x_t + b) \\ c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\ h_t &= o_t \circ \tanh(c_t) \end{aligned} \tag{3.9}$$

L'architettura risultante è rappresentata graficamente in figura 3.2

Dunque, il modello student sarà costituito da 3 componenti principali: un layer

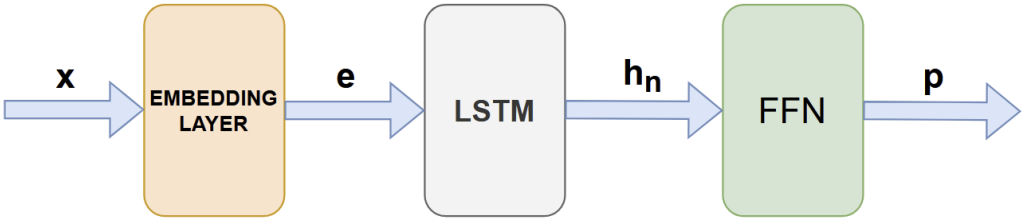


Figura 3.3: Architettura del modello student: x è la sequenza in input, e è l' embedding della sequenza in input, h_n è lo stato interno dell'ultimo elemento della sequenza (se bidirezionale è la concatenazione delle due direzione), e p è il vettore dei logit.

di embedding che si occupa di codificare le parole della sequenza di input in vettori densi, un layer LSTM che elabora tale input e ne costruisce una rappresentazione latente, e un layer denso di classificazione che produce i logit necessari a produrre la previsione. L'architettura dello student è riassunta graficamente dalla figura 3.3.

Modello teacher Per quanto riguarda il modello teacher, questo è un modello di tipo Transformer [36], e in particolare, si tratta del modello BERT (*Bidirectional Encoder Representations from Transformers*) [4]. Tuttavia, ai fini di questo elaborato proponiamo una versione modificata di questo modello che aggiunge le matrici di LoRA in alcuni punti di interesse. Partendo dalla struttura generale del meccanismo di *self-attention*, andiamo ad aggiungere delle matrici LoRA a livello delle matrici di *Query*, *Key* e *Value*. L'idea alla base di questa scelta è che andando ad aggiornare queste matrici di basso rango anzichè quelle appena citate, si ottenga una procedura di training più efficiente e veloce. Sostanzialmente, dato un input x avremo che questo verrà processato secondo la seguente:

$$\begin{aligned} Q &= W_Q x + B_Q A_Q x \\ K &= W_K x + B_K A_K x \\ V &= W_V x + B_V A_V x \end{aligned} \tag{3.10}$$

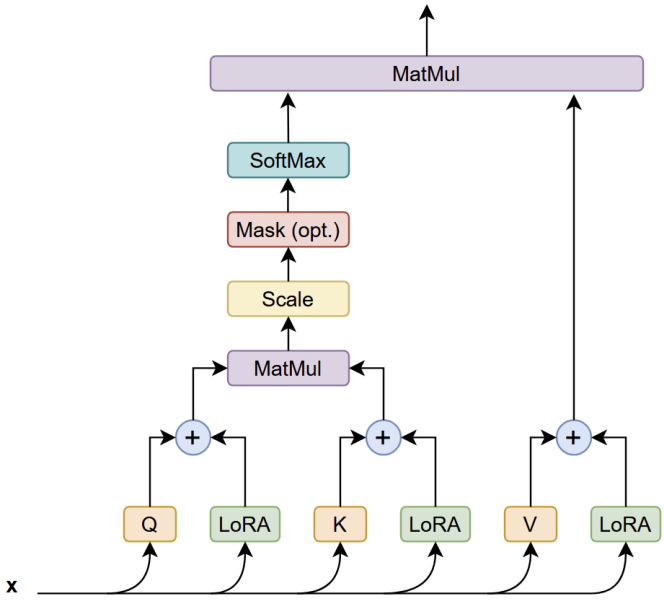


Figura 3.4: Integrazione delle matrici di LoRA all'interno del meccanismo di self attention

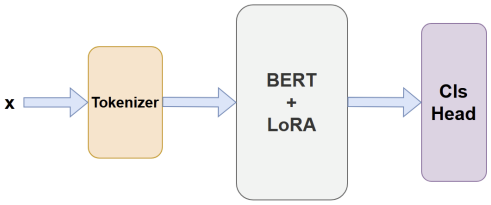


Figura 3.5: Teacher model workflow

La figura 3.4 illustra graficamente come si modifica il meccanismo di self attention introducendo la modifica architetturale appena illustrata.

Nel complesso, dunque, il modello teacher sarà composto da due componenti: la prima è la componente di encoding, rappresentata da BERT (modificato secondo quanto descritto in questo paragrafo); e la seconda è la testa di classificazione, aggiunta in fase di finetuning.

3.2.2 Procedura di distillazione

Prima di procedere alla descrizione *step-by-step* della procedura di distillazione, è opportuno approfondire meglio le motivazioni alla base dell'introduzione della metodologia LoRA all'interno del modello teacher. Consideriamo le equazioni 3.10 e per semplicità lavoriamo solo con la prima equazione, ma il seguente ragionamento vale allo stesso modo anche per le altre. Durante la fase di backpropagation avremo:

$$\theta_T \leftarrow \theta_T - \nabla_{\theta_T} \mathcal{L}_T(x; \theta_S(\theta_T)) \tag{3.11}$$

Se esplicitiamo il contributo dato dai parametri del modello e quello dato dalle matrici LoRA avremo la seguente:

$$\theta_{M \cup LoRA} \leftarrow \theta_{M \cup LoRA} - \nabla_{\theta_{M \cup LoRA}} \mathcal{L}_T(x; \theta_S(\theta_{M \cup LoRA})) \tag{3.12}$$

Dove con M indichiamo i parametri originali del modello. Ma dal momento che i parametri originali del modello sono congelati avremo la seguente equivalenza:

$$\nabla_{\theta_{M \cup LoRA}} \mathcal{L}_T(x; \theta_S(\theta_{M \cup LoRA})) \equiv \nabla_{\theta_{LoRA}} \mathcal{L}_T(x; \theta_S(\theta_{LoRA})) \tag{3.13}$$

A questo punto, se osserviamo che $|\theta_{LoRA}| \ll |\theta_M|$, il vantaggio in termini computazionali, temporali ed energetici è evidente. A questo punto possiamo descrivere il processo di distillazione, che per sommi capi, ripercorre quello proposto in [41]. La procedura si articola in 3 fasi principali: nella prima fase si crea una copia dello student, e si effettua su di essa un esperimento, una prova di distillazione; a questo punto si sottopone a questo studente distillato un quiz (un batch di esempi dedicato), e sulla base delle performance su di esso il teacher viene aggiornato affinchè la sua conoscenza sia più distillabile (in questo caso si aggiornano solamente le matrici LoRA); infine, si utilizza il teacher aggiornato per distillare la conoscenza nello student originale; Dopodiché il processo si ripete.

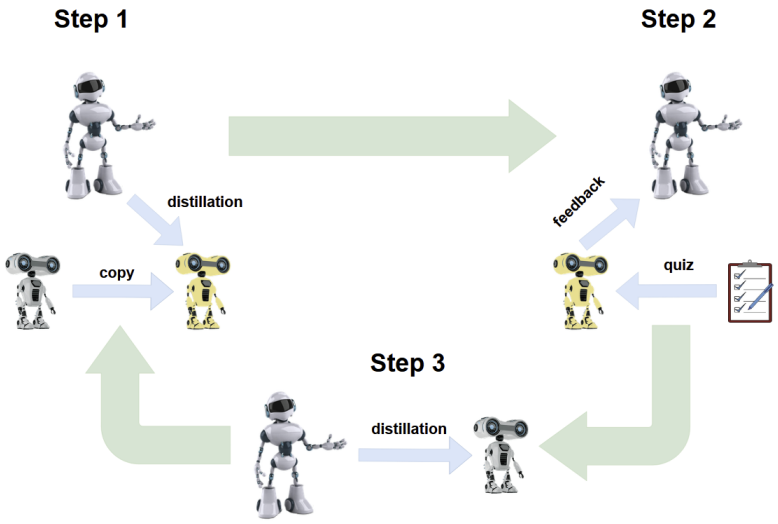


Figura 3.6: Distillation workflow

Loss function Descriviamo brevemente la loss utilizzata durante la distillazione. La forma generale della loss function è la seguente:

$$\mathcal{L}(x, y, \theta_T, \theta_S) = \alpha \mathcal{L}_{KD}(x, \theta_T, \theta_S, \tau) + (1 - \alpha)\mathcal{L}_T(x, y, \theta_S) \tag{3.14}$$

Dove (x, y) è un batch di esempi, $\mathcal{L}_{KD}$ è la loss di distillazione (ad esempio può essere la cross entropy, la KL divergence, l'MSE, etc.), $\mathcal{L}_T$ è la loss specifica del task, e infine, $\alpha \in [0, 1]$ è un iperparametro che regola l'importanza della componente di distillazione rispetto alla loss specifica sul task di downstream.

La figura 3.8 illustra più dettagliatamente come si propaga il feedback dallo student verso il teacher: x_Q rappresenta un campione dal quiz set, sul quale lo student effettua delle previsioni; su queste previsioni viene calcolata la loss scelta per il task di downstream, che verrà poi utilizzata per aggiornare i parametri del teacher; a questo punto, i parametri di base del modello teacher saranno *frozen*, e quindi non saranno interessati dalla retropropagazione del gradiente, mentre i parametri di LoRA saranno *trainable* e subiranno l'aggiornamento. Osserviamo che,

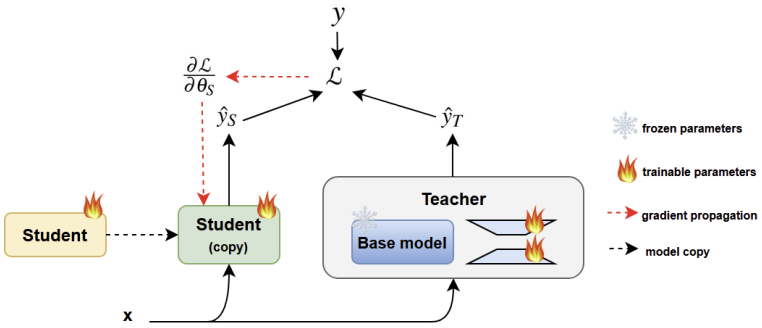


Figura 3.7: Step 1 della procedura di distillazione

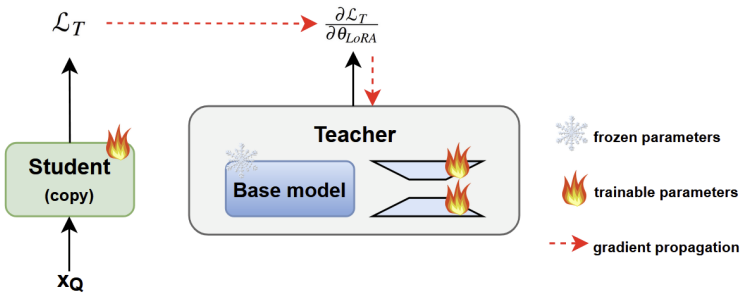


Figura 3.8: Step 2 della procedura di distillazione

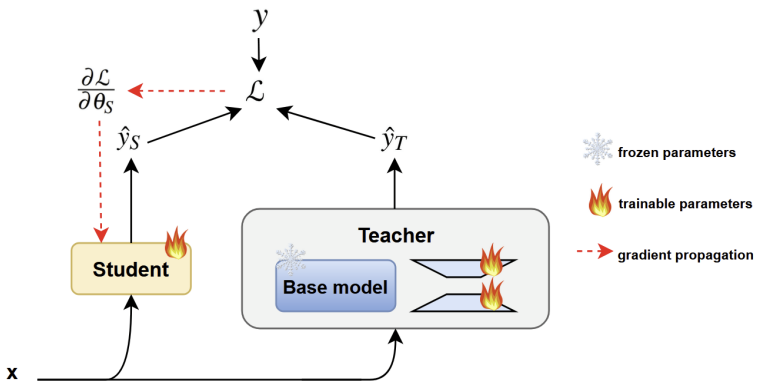


Figura 3.9: Step 3 della procedura di distillazione

anche se all'apparenza la loss così calcolata non dipende dai parametri del teacher (in quanto le predizioni le effettua lo student), in realtà questa dipende da questi in quanto lo student è il risultato di un "prova di distillazione" più formalmente possiamo descrivere questa dipendenza secondo le seguenti:

$$\begin{aligned} \theta'_S &\leftarrow \theta'_S - \alpha \nabla_{\theta_S} \mathcal{L}_{KD}(x, \theta'_S, \theta_T) \quad (step\ 1) \\ \theta_T &\leftarrow \theta_T - \alpha \nabla_{\theta_T} \mathcal{L}_T(x, \theta'_S) \qquad (step\ 2) \end{aligned} \tag{3.15}$$

Sostanzialmente accade che θ_T dipende da θ'_S che a sua volta dipende da θ_T,

Algorithm 1 Procedura di distillazione

Require: Student θ_S, teacher θ_T, Dataset $\mathcal{D}$, Quizset $\mathcal{Q}$, student learning rate α_S, teacher learning rate α_T

1: **while** not done **do**
2: Sample $(x, y) \sim \mathcal{D}$
3: $\theta'_S \leftarrow \theta_S$ ▷ student copy
4: $\theta'_S \leftarrow \theta'_S - \alpha_S \nabla_{\theta'_S} \mathcal{L}(x, y, \theta_{M \cup LoRA}, \theta_S)$ ▷ distillation experiment (Step 1)
5: Sample $(x_Q, y_Q) \sim \mathcal{Q}$
6: $\theta_{LoRA} \leftarrow \theta_{LoRA} - \alpha_T \nabla_{\theta_{LoRA}} \mathcal{L}_T(x_Q, y_Q, \theta'_S(\theta_{M \cup LoRA}))$ ▷ feedback propagation (Step 2)
7: $\theta_S \leftarrow \theta_S - \alpha_S \nabla_{\theta_S} \mathcal{L}(x, y, \theta_{M \cup LoRA}, \theta_S)$ ▷ distillation pass (Step 3)
8: **end while**

di conseguenza il gradiente è non nullo e quindi il feedback può essere retropropagato. La figura 3.9 illustra più dettagliatamente il terzo step della procedura di distillazione, in questo caso l'input viene elaborato da entrambi i modelli, i quali producono le predizioni $\hat{y}_S$ per lo student e $\hat{y}_T$ per il teacher. Tali predizioni, vengono utilizzate congiuntamente alle *hard labels* del dataset per calcolare la funzione loss totale, la quale verrà poi utilizzata per l'aggiornamento dei pesi dello student durante la fase di backpropagation. L'algoritmo 1 illustra lo pseudocodice che racchiude l'intero processo appena descritto. Le righe due, tre e quattro rappresentano lo step uno del processo, in cui si copia lo student e si effettua un esperimento di distillazione; le righe cinque e sei rappresentano lo step due del processo, in cui si propaga il feedback aggiornando opportunamente i parametri del teacher; e infine, la riga sette rappresenta lo step tre del processo, in cui si

utilizza il teacher aggiornato per distillare la conoscenza nello student originale. Volendo riassumere, la proposta appena presentata combina 3 diverse tecniche di training per modelli di deep learning: il meta-learning viene utilizzato per rendere la conoscenza del teacher meglio distillabile attraverso un meccanismo di feedback, la distillazione viene utilizzata per trasferire la conoscenza dal modello teacher al modello student (allineando i logit dei due modelli), e la *Low Rank Adaptation (LoRA)* viene utilizzata per rendere più veloce ed efficiente l'intero processo. Nel capitolo che segue, verranno descritti gli esperimenti condotti per testare la validità della metodologia proposta, e verranno presentati alcuni risultati sperimentali che dimostrano come l'introduzione delle matrici di LoRA nel modello teacher possano effettivamente portare un beneficio significativo al processo di distillazione.

Capitolo 4

Valutazione sperimentale

Veniamo dunque alla sperimentazione della metodologia proposta nel capitolo precedente. Gli esperimenti condotti, mirano alla verifica della validità della metodologia, in termini di performance, tempo di training e consumo energetico. Il capitolo sarà organizzato secondo la seguente struttura: la prima sezione sarà dedicata al *design degli esperimenti*, presenteremo quindi il dataset e il task di downstream utilizzati, i principali iperparametri e le metriche di valutazione utilizzate per la validazione della metodologia. La seconda sezione, invece, sarà dedicata alla *presentazione dei risultati* ottenuti, discuteremo di come è stato fatto il tuning degli iperparametri, vedremo un confronto tra la metodologia proposta e lo stato dell'arte cercando di fare emergere eventuali punti di forza e/o di debolezza della metodologia, e discuteremo di alcuni aspetti e conseguenze secondarie ma non meno rilevanti dell'utilizzo combinato di meta-learning e tecniche di distillazione. Tutto ciò allo scopo di fornire tutte le informazioni necessarie a trarre delle conclusioni esaustive sul metodo proposto, e a definire quali potrebbero essere degli ulteriori miglioramenti futuri alla metodologia. L'approccio utilizzato nelle sezioni che seguono, oltre ad essere finalizzato alla presentazione degli esperimenti condotti, lascerà ampio spazio a considerazioni teoriche sui risultati che si osservano, in modo da fornire maggiore autorevolezza alle conclusioni inferite dagli esperimenti condotti.

4.1 Design degli esperimenti

Iniziamo dunque definendo il design degli esperimenti. La presente sezione si articola in tre parti: nella prima viene descritto *Twitter financial news sentiment*, un dataset di notizie finanziarie opportunamente etichettato per un task di classificazione utilizzato negli esperimenti condotti; nella seconda vedremo quali sono i principali iperparametri degli eseprimenti, discuteremo degli ottimizzatori, delle funzioni di loss, dei learning rate e di altri iperparametri selezionati; nella terza parte, infine, presenteremo le metriche di valutazione utilizzate per determinare le performance del modello.

4.1.1 Dataset: Twitter financial news sentiment

Twitter financial news sentiment è un dataset di testi scritti in lingua inglese, ciascuno di questi testi è annotato con una etichetta che indica il sentimento espresso nel testo. Il dataset è composto da *11932* coppie testo-etichetta, di cui *9938* compongono il trainset e *9486* compongono il validation set (che negli esperimenti condotti per questo elaborato ha ricoperto il ruolo di testset).

Etichette Ciascun testo contenuto nel dataset ha associata una etichetta rappresentativa del sentimento espresso nel testo. Questo è un dataset di classificazione multiclasse, e in particolare, l'insieme delle etichette è composto da 3 etichette:

- **Bearish** a cui è associata l'etichetta 0, un testo annotato con questa etichetta contiene un sentimento *negativo*, dovuto all'abbassamento del valore di una certa azione.
- **Bullish** a cui è associata l'etichetta 1, un testo annotato con questa etichetta contiene un sentimento *positivo*, dovuto al rialzo del valore di una certa azione.
- **Neutral** a cui è associata l'etichetta 2, un testo annotato con questa etichetta contiene un sentimento *neutrale*, cioè il testo non si esprime rispetto alla notizia che riporta ma i fatti in essa vengono riportati in modo neutrale.

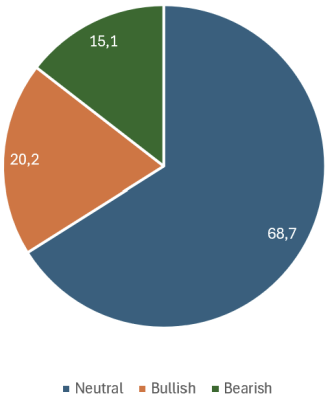


Figura 4.1: Distribuzione delle etichette nel dataset

Inoltre, come mostrato in figura 4.1, il dataset è fortemente sbilanciato a favore dell'etichetta Neutral (che rappresenta il 68,7% del totale), rispetto all'etichetta Bullish (che rappresenta il 20,2% del totale), e ancor più rispetto all'etichetta Bearish (che rappresenta solo il 15,1% del totale).

Preprocessing I testi contenuti nel dataset sono caratterizzati dall'abbondante presenza di rumore che potrebbero inficiare la qualità del training dei modelli. Infatti, nei testi sono contenuti molti collegamenti ipertestuali, emoji e altri elementi che non risultano essere utili ai fini della sentiment analysis. Pertanto, sui testi è stata effettuata una fase di preprocessing per ripulirli da tutti gli elementi di disturbo. In particolare, per prima cosa tutti i testi vengono trasformati in carattere minuscolo e viene applicata la seguente espressione regolare $htt\,p\backslash S+$ per la rimozione dei link a contenuti sul web, successivamente viene applicata una seconda espressione regolare della forma $[\hat{}a-zA-Z\$0-9\backslash d\backslash s!?]+$ per rimuovere tutti i caratteri indesiderati e fare in modo che i testi siano composti solo da lettere, cifre, spazi, il simbolo $ e i simbolo ! e ?. Infine, vengono rimossi gli spazi superfluei agli estremi delle stringhe di testo.

Teacher tokenizer Il modello teacher utilizza un tokenizzatore basato su *WordPiece*, adattato opportunamente ai testi contenuti nel dataset di riferimento. Il numero massimo di token oltre il quale la sequenza viene troncata è 512, la tronca-

tura viene applicata senza *stride* e secondo una politica *LongestFirst* (rimuove un token alla volta dalla sequenza più lunga della coppia, ricordando che il modello BERT per il task di pretraining lavora con input composti da coppie di testi). Inoltre il tokenizzatore è dotato dei seguenti token speciali: $[CLS]$ token di classificazione, $[SEP]$ token per la separazione di segmenti nell'input (ad esempio premessa e conseguenza), $[UNK]$ token utilizzato quando una sub-word non è contenuta nel vocabolario del tokenizzatore, $[MASK]$ token speciale usato per mascherare alcuni pezzi di testo nel task di *Masked Language Modeling*. La dimensione totale del vocabolario del tokenizzatore utilizzato è pari a $30,873$ token.

Word Piece

Word Piece è un algoritmo di segmentazione del testo in *pezzi di parole* utilizzato spesso nel *Natural Language Processing*. Il vocabolario è inizializzato con i singoli caratteri del linguaggio da apprendere e successivamente viene applicata una procedura iterativa di fusione dei token secondo il seguente schema:

1. `inizializza il vocabolario con tutti i caratteri presenti nel testo.`
2. `costruisce un language model sul vocabolario attuale`
3. `genera un nuovo token combinando due token dell'attuale vocabolario in modo che questo fornisca il massimo incremento della funzione di verosimiglianza calcolata sui dati di training`
4. `applica iterativamente i passi 2 e 3 fino a quando la funzione di verosimiglianza non supera un threshold fissato oppure il vocabolario ha raggiunto la dimensione desiderata`

Student tokenizer Nel caso del modello student, si è deciso di utilizzare degli embeddings preaddestrati, e in particolare, *GloVe* nella versione con dimensione degli embeddings pari a 50, quindi in questo caso si è effettuato manualmente un mapping tra le parole del dataset e il relativo embedding, aggiungendo poi un token $[PAD]$ per il padding con embedding nullo, e un token $[UNK]$ il cui embedding è pari alla media di tutti gli altri embedding. Dunque, nel caso del modello student, avremo che il testo sarà pre-tokenizzato e non viene usato un vero e proprio tokenizzatore.

4.1.2 Principali iperparametri

Passiamo dunque alla descrizione dei principali iperparametri che sono stati selezionati nel corso della definizione degli esperimenti condotti. Tali iperparametri coinvolgono le funzioni di loss, sia per la distillazione che per il task di downstream; il parametro di temperatura; il rango delle matrici LoRA e gli ottimizzatori utilizzati. La prima cosa di cui ci occuperemo sono proprio gli ottimizzatori, e procederemo ad illustrarne alcuni che sono di particolare interesse ai fini di questo elaborato.

Stochastic Gradient Descent (SGD) È alla base di tutti gli ottimizzatori utilizzati in applicazioni di machine e deep learning, l'idea alla base è che se il gradiente di una funzione descrive la sua direzione di crescita, allora muovendosi *per step* nella direzione opposta al gradiente prima o poi si giungerà in un punto di ottimo. Se indichiamo con $\mathcal{L}(\theta)$ la loss function da minimizzare, allora indicheremo il gradiente come segue:

$$\nabla_\theta \mathcal{L}(\theta) = \left[\frac{\partial \mathcal{L}(\theta)}{\partial \theta_1}, \frac{\partial \mathcal{L}(\theta)}{\partial \theta_2}, \ldots, \frac{\partial \mathcal{L}(\theta)}{\partial \theta_n} \right]^T \tag{4.1}$$

Dove n è il numero di parametri del modello. A questo punto i parametri vengono aggiornati spostandosi lungo la direzione dell'anti-gradiente di un suo sotto multiplo:

$$\theta = \theta - \eta \nabla_\theta \mathcal{L}(\theta) \tag{4.2}$$

Dove η è il learning rate, ovvero la dimensione del passo che si effettua lungo l'anti-gradiente. È opportuno precisare che l'aggiornamento del gradiente non avviene sull'intero dataset, ma su un batch campionato da esso, e il processo si ripete iterativamente fino a convergenza. Inoltre, nelle versioni di SGD utilizzate nella pratica vengono aggiunti alcuni perfezionamenti: il *weight decay* che aiuta a prevenire l'overfitting penalizzando la complessità del modello; e il *momento di nesterov* che permette di accelerare la convergenza considerando nell'aggiornamento anche i gradienti passati.

Algorithm 2 Stochastic Gradient Descent (SGD)

Require: η (learning rate), $\mathcal{L}(\theta)$ (loss function), λ (weight decay), μ (momento), τ (dampening)

for $t = 1$ **to** $\ldots$ **do**
  $g_t \leftarrow \nabla_\theta \mathcal{L}_t(\theta_{t-1})$
  if $\lambda \neq 0$ **then**
    $g_t \leftarrow g_t + \lambda\,\theta_{t-1}$
  end if
  if $\mu \neq 0$ **then**
    if $t > 1$ **then**
      $\mathbf{b}_t \leftarrow \mu \mathbf{b}_{t-1} + (1-\tau) g_t$
    else
      $\mathbf{b}_t \leftarrow g_t$
    end if
  end if
  $g_t \leftarrow g_t + \mu \mathbf{b}_t$
  $\theta_t \leftarrow \theta_{t-1} - \eta g_t$
end for
return θ_t

Root Mean Square Propagation (RMSprop) È un algoritmo di ottimizzazione che scaturice da un miglioramento di un altro algoritmo chiamato *Rprop*, che proponeva l'aggiustamento adattivo del learning rate di ogni singolo peso rispetto al segno dei gradiente alle ultime due iterazioni. Il ragionamento alla base di questa metodologia è che se i gradienti alle iterazioni precedenti hanno lo stesso segno allora la direzione è quella giusta e quindi il learning rate può essere aumentato, contrariamente, se i due gradienti hanno segni opposti significa che il learning rate deve essere decrementato. Tuttavia, Rprop non funziona bene in contesti reali,

Algorithm 3 Root Mean Square Propagation (RMSprop)

Require: α, η (learning rate), $\mathcal{L}(\theta)$ (loss function), λ (weight decay), μ (momento), *centered* (booleano)

for $t = 1$ **to** ... **do**
  $g_t \leftarrow \nabla_\theta \mathcal{L}_t(\theta_{t-1})$
  if $\lambda \neq 0$ **then**
    $g_t \leftarrow g_t + \lambda \theta_{t-1}$
  end if
  $v_t \leftarrow \alpha v_{t-1} + (1-\alpha) g_t^2$
  $\tilde{v}_t \leftarrow v_t$
  if *centered* **then**
    $g_t^{ave} \leftarrow \alpha g_{t-1}^{ave} + (1-\alpha) g_t$
    $\tilde{v}_t \leftarrow \tilde{v}_t - (g_t^{ave})^2$
  end if
  if $\mu > 0$ **then**
    $\mathbf{b}_t \leftarrow \mu \mathbf{b}_{t-1} + \frac{g_t}{\sqrt{\tilde{v}_t} + \varepsilon}$
    $\theta_t \leftarrow \theta_{t-1} - \eta \mathbf{b}_t$
  else
    $\theta_t \leftarrow \theta_{t-1} - \eta \frac{g_t}{\sqrt{\tilde{v}_t} + \varepsilon}$
  end if
  return θ_t
end for

quando si lavora con dataset molto grandi che richiedono un aggiornamento in mini-batch. Questo perchè guardando solo ai segni dei pesi, Rprop non è in grado di mediare efficaciemente i gradienti su mini-batch successivi. Per risolvere questo problema, RMSprop aggiusta i gradienti utilizzando la media mobile dei quadrati dei gradienti, che è un metodo che permette di mediare meglio i gradienti su mini-batch successivi. Formalmente avremo quanto segue:

$$\begin{aligned} \mathbb{E}[g^2]_t &= \alpha \mathbb{E}[g^2]_{t-1} + (1-\alpha) \left(\frac{\partial \mathcal{L}}{\partial \theta} \right)^2 \\ \theta_t &= \theta_{t-1} - \frac{\eta}{\sqrt{\mathbb{E}[g^2]_t}} \frac{\partial \mathcal{L}}{\partial \theta} \end{aligned} \tag{4.3}$$

Inoltre, nell'aggiornamento può anche essere considerata la varianza dei gradienti, in questi casi si parla di una versione *centrata* di RMSprop.

Adam Introdotto in [21], *Adam* è uno dei più famosi algoritmi di ottimizzazione utilizzati in machine e deep learning. L'idea chiave alla base di Adam è il calcolo adattivo dei learning rate per i parametri attraverso una stima dei momenti del primo e del secondo ordine, mantenendo efficienza computazionale e basso impiego di memoria. Questo ottimizzatore nasce dalla combinazione di altri ottimizzatori come *RMSprop* (che abbiamo appena descritto) e *AdaGrad* [5]. Vediamo nel dettaglio come funziona Adam. Consideriamo una funzione scalare stocastica $\mathcal{L}$, differenziabile rispetto ai parametri θ. Siamo interessati a minimizzare $\mathbb{E}[\mathcal{L}(\theta)]$ rispetto ai parametri θ, e indichiamo con $\mathcal{L}_1(\theta), \mathcal{L}_2(\theta), ..., \mathcal{L}_T(\theta)$ le realizzazioni della funzione stocastica negli istanti temporali $1, 2, ..., T$ (la stocasticità nel nostro caso è dovuta al campionamento dei rando dei mini-batch). Indichiamo con $g_t = \nabla_\theta \mathcal{L}_t(\theta)$. L'algoritmo aggiorna le medie mobili esponenziali (m_t) e il gradiente al quadrato (v_t) e l'utilizzo di due iperparametri $\beta_1, \beta_2 \in (0,1)$ controllano i rate di decadimento di queste medie mobili, che osserviamo essere una stima del momento del primo e del secondo ordine del gradiente (e sono inizializzate come vettori nulli). Osserviamo che nelle righe 9 e 10 dell'algoritmo 4 viene

Algorithm 4 Adam

Require: η (learning rate), $\beta_1, \beta_2 \in [0,1)$ (rates di decadimento esponenziale), $\mathcal{L}(\theta)$ (funzione da ottimizzare), θ_0 (vettore dei parametri iniziali)

1: $m_0 \leftarrow 0$
2: $v_0 \leftarrow 0$
3: $t \leftarrow 0$
4: **while** θ_t not converged **do**
5: $\quad t \leftarrow t + 1$
6: $\quad g_t \leftarrow \nabla_\theta \mathcal{L}_t(\theta_{t-1})$
7: $\quad m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
8: $\quad v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
9: $\quad \hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$
10: $\quad \hat{v}_t \leftarrow \frac{v_t}{(1-\beta_2^t)}$
11: $\quad \theta_t \leftarrow \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$
12: **end while**
13: **return** θ_t

effettuata una correzzione delle stime, questo perchè l'inizializzazione a zero di queste, soprattutto nelle prime iterazioni, tende a spingerle molto verso lo zero.

Oltre alla versione standard presentata qui, esitono molte versioni alternative di Adam, ai fini di questo elaborato è opportuno citare la versione *AdamW*. AdamW aggiunge ad Adam un termine di regolarizzazione L_2, trasformando la formula di aggiornamento nella seguente:

$$\theta_t = \theta_{t-1} + \eta \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon} + w_t \theta_{t-1} \right) \tag{4.4}$$

Cross Entropy Loss (CE) Terminata la presentazione degli ottimizzatori, presentiamo adesso alcune loss function di nostro interesse, iniziando dalla *Cross Entropy*. Date due distribuzioni di probabilità p e q, la *Cross Entropy* di q rispetto a p su insieme dato equivale alla seguente:

$$H(p,q) = -\mathbb{E}_p[\log q] \tag{4.5}$$

Se consideriamo un contesto discreto, avremo che su un set $\mathcal{X}$ la cross entropy può essere scritta come:

$$H(p,q) = -\sum_{x \in \mathcal{X}} p(x) \log q(x) \tag{4.6}$$

Nel machine learning, questa loss si applica a molti task. Se consideriamo ad esempio un task di classificazione con C classi avremo che la Cross Entropy può essere calcolata secondo la seguente:

$$CE = -\sum_{(x,y) \in \mathcal{D}} \sum_{i=1}^{C} y_i \log p_i(x) \tag{4.7}$$

In questo caso y_i è la ground truth per la classe i e $p_i(x)$ è la probabilità restituita dal modello per la classe i. Nel contesto di questo elaborato la cross entropy viene utilizzata con una duplice funzione: la prima è quella classica di loss per la classificazione sul task di downstream, e la seconda è quella di candidato come loss di allineamento tra i logit del modello teacher e del modello student.

Kullback-Leibler divergence (KL) La *divergenza di Kullback-Leibler* ($KL(p||q)$) è una misura della differenza tra due distribuzioni di probabilità. In particolare, si tratta di una misura della perdita di informazione dovuta all'approssimazione di p attraverso q. È importante osservare che questa misura non è una *metrica*, infatti, essa non è simmetrica. Nel caso discreto ha la seguente forma:

$$KL(P||Q) = \sum_i P(i) \log_2 \left(\frac{P(i)}{Q(i)} \right) \tag{4.8}$$

Nel contesto di questo elaborato, questa loss risulta essere un buon candidato come loss di distillazione, in quanto permette di calcolare la differenza tra la distribuzione del modello teacher e quella del modello student, permettendo così di allinearle.

Mean Squared Error (MSE) È una misura molto nota in statistica utilizzata per la verifica della qualità di uno stimatore. Nel machine learning viene utilizzata come loss function per molti task diversi, ad esempio la distillazione o l'*anomaly detection*. L'MSE ha la seguente forma:

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (Y_i - \hat{Y}_i)^2 \tag{4.9}$$

ovvero per ogni item del dataset si misura il quadrato della differenza tra il valore reale della variabile aleatoria (Y_i) ed il valore predetto ($\hat{Y}_i$), e di questi si calcola la media. Nel contesto di questo elaborato, l'MSE risulta essere un buon candidato come componente di distillazione in quanto permette di calcolare la differenza media tra i logit dei due modelli.

Temperatura Terminata la presentazione delle funzioni di loss coinvolte, discutiamo brevemente l'iperparametro temperatura. La temperatura è un iperparametro introdotto nel contesto dei Large Language Modeling col fine di rendere meno sparse le distribuzioni sulle varie classi. Nel contesto della distillazione, questo iperparametro assuma particolare importanza, questo perchè se la distribuzione

del teacher è meno sparsa sarà più informativa. Formalmente avremo:

$$\mathcal{L}_{KD}\left(\frac{\hat{y}_S}{\tau}, \frac{\hat{y}_T}{\tau}, x, y\right) \tag{4.10}$$

Dove τ è il parametro di temperatura, $\hat{y}_T$ sono i logit del teacher, $\hat{y}_S$ sono i logit dello studet e (x,y) sono esempi ed etichette del mini-batch. Per chiarire meglio questo aspetto consideriamo un task di classificazione a 3 classi, se il teacher produce una previsione del tipo $(0.998, 0.001, 0.001)$ allora l'informazione che lo student può recepire è solo che la classe 0 è quella corretta, mentre se il teacher produce una predizione del tipo $(0.85, 0.1, 0.05)$ allora l'informazione che lo student può recepire è che la classe 0 è quella corretta, ma anche che la classe 1 è molto più verosimile della classe 2.

LoRA rank L'ultimo iperparametro che consideriamo è il rango delle matrici LoRA. Come già illustrato in precedenza la *Low-Rank Adaptation* può essere rappresentata secondo la seguente:

$$h = Wx + \Delta Wx = Wx + BAx \tag{4.11}$$

Dove $B \in \mathbb{R}^{d \times r}$ e $A \in \mathbb{R}^{r \times k}$ con $r \ll min(d,k)$. r è il rango delle matrici *LoRA* e determina la dimensione dello spazio di proiezione. Sciegliere opportunamente il rango permette di avere il giusto trade off tra efficienza e performance. Avere un rango molto grande fornisce al modello maggiore espressività al prezzo di una efficienza minore in termini di tempo di training e occupazione di memoria. AL contrario, un rango molto piccolo fornisce maggiore efficienza al prezzo di una minore espressività.

4.1.3 Metriche di valutazione

Per concludere questa prima sezione del capitolo discutiamo brevemente le metriche di valutazione utilizzate per misurare le performance del modello distillato.

	Predicted positive	**Predicted negative**
Observed positive	TP	FN
Observed negative	FP	TN

Tabella 4.1: Confusion matrix

Accuracy L'*accuratezza* è una delle metriche di valutazione più frequentemente utilizzate per task come la classificazione. Informalmente, possiamo definire all'*accuracy* di un modello come la frazione degli esempi di un testset correttamente classificati:

$$Accuracy = \frac{\#\,Esempi\ classificati\ correttamente}{\#\,Totale\ di\ esempi} \tag{4.12}$$

Più formalmente, consideriamo un task di classificazione binaria e costruiamo una *confusion matrix* (tabella 4.1) dove: **TP** (*True Positive*) è il numero di esempi positivi classificati correttamente dal modello, **TN** (*True Negative*) è il numero di esempi negativi classificati correttamente dal modello, **FP** (*False Positive*) è il numero di esempi negativi classificati erroneamente come positivi dal modello, e infine **FN** (*False Negative*) è il numero di esempi positivi classificati erroneamente come negativi dal modello. A questo punto definiamo formalmente l'accuracy come segue:

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \tag{4.13}$$

Notiamo come questa misura da sola non è molto informativa in quanto non tiene conto dello sbilanciamento tra le classi. Osserviamso che l'estensione di questa misura ad un contesto multiclasse è immediata, basterà considerare cumulativamente le predizioni corrette per ogni classe.

Precision e Recall Descriviamo ora due misure strettamente correlate all'accuracy ma che sono più adatte al valutazione delle performance per problemi con classi sbilanciate. *Precision e Recall* scaturiscono, così come l'accuracy, dalla

matrice di confusione, e in particolare, sono definite come segue:

$$\begin{aligned} Precision &= \frac{TP}{TP+FP} \\ Recall &= \frac{TP}{TP+FN} \end{aligned} \tag{4.14}$$

Sostanzialmente la *Precision* calcola la frazione di esempi classificati come positivi che sono stati classificati correttamente, mentre la *Recall* calcola la frazione di esempi positivi classificati correttamente. L'estensione al più di due classi si effettua andando a considerare una classe per volta come classe positiva e tutte le altre come classi negative, e aggregando alla fine i valori di precision e/o recall ottenuti.

Inoltre, l'aggregazione può avvenire secondo due diverse politiche: *Micro Averaging* e *Macro Averaging*. Nel primo caso, *Micro Averaging* andiamo a considerare singolarmente il contributo in termini di TP, TN, FP ed FN di ogni singola classe:

$$\begin{aligned} Micro\ Average\ Precision &= \frac{\sum_{i=1}^{C} TP_i}{\sum_{i=1}^{C} TP_i + \sum_{i=1}^{C} FP_i} \\ Micro\ Average\ Recall &= \frac{\sum_{i=1}^{C} TP_i}{\sum_{i=1}^{C} TP_i + \sum_{i=1}^{C} FN_i} \end{aligned} \tag{4.15}$$

Dove con TP_i si intende il numero di True Positive nella matrice di confusione costruita considerando la classe i come positiva (il discorso è analogo per gli altri valori). Nel secondo caso, *Macro Averaging*, andiamo a fare la media dei valori di Precision/Recall calcolati per le classi singolarmente:

$$\begin{aligned} Macro\ Average\ Precision &= \frac{\sum_{i=1}^{C} Precision_i}{C} \\ Macro\ Average\ Recall &= \frac{\sum_{i=1}^{C} Recall_i}{C} \end{aligned} \tag{4.16}$$

Dove $Precision_i$ indica il valore di Precision calcolato sulla matrice di confusione che considera la classe i come positiva (discorso analogo per la Recall). Ai

fini di questo elaborato utilizziamo la versione *Macro Averaging*, questo perchè desideriamo trattare equamente tutte le classi, e inoltre, il dataset utilizzato non è bilanciato.

F1 score Si tratta di una misura che combina Precision e Recall facendone la *media armonica*. Viene introdotta per bilanciare Precision e Recall, in quanto, da un lato potrebbe accadere che il modello sia incentivato a rispondere sempre con la classe positiva in modo da massimizzare la recall (con la conseguente penalizzazione della precision), e dall'altro la precision non considera i falsi negativi. L'*F1 score* ha la seguente forma:

$$F1\ score = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \tag{4.17}$$

Osserviamo che $F1\ score \in [0,1]$, e vale 0 quando una delle due metriche vale 0 mentre vale 1 quando entrambe le metriche valgono 1.

4.2 Risultati ottenuti

In questa sezione presenteremo i risultati finali dei test condotti. Per prima cosa discuteremo il tuning degli iperparametri, vedremo come questi sono stati impostati e cercheremo, se possibile, di spiegare da un punto di vista teorico la consistenza del setting ottenuto. Vedremo inoltre un confronto con lo stato dell'arte, non solo in termini di metriche di valutazione ma anche di efficienza energetica del training. Infine, discuteremo delle implicazioni che l'utilizzo del meta-learning ha avuto sui modelli.

4.2.1 Tuning degli iperparametri

Come fatto nella sezione 4.1.2 procediamo discutendo uno per volta i principali iperparametri.

Metodologia	Optim. student	Optim. teacher	Accuracy
Distillazione classica	Adam	-	0.7902
Distillazione classica	RMSprop	-	**0.8149**
Meta-Distillazione Full-finetuning	Adam	AdamW	0.8375
Meta-Distillazione Full-finetuning	RMSprop	AdamW	**0.8442**
Meta-Distillazione con LoRA	Adam	AdamW	0.8438
Meta-Distillazione con LoRA	Adam	Adam	0.8329
Meta-Distillazione con LoRA	RMSprop	AdamW	**0.8480**
From Scratch	Adam	-	0.8057
From Scratch	RMSprop	-	**0.8082**

Tabella 4.2: Confronto tra gli ottimizzatori Adam e RMSprop per le varie metodologie di training (con lo stesso setting dei restanti iperparametri), viene riportata l'accuracy massima ottenuta su 20 epoche di training

Ottimizzatore student I test condotti per il tuning sono riassunti nella tabella 4.2, che riporta per ciascuna metodologia, e per ciascun ottimizzatore, l'accuracy massima ottenuta su 20 epoche di training. Come si può osservare dai valori evidenziati in grassetto, RMSprop per il modello student risulta essere più performante rispetto ad Adam.

Ottimizzatore teacher Per quanto rigurda l'ottimizzatore del modello teacher, questo riguarda solo le metodologie di distillazione con feedback ovvero quella proposta in questo elaborato e lo stato dell'arte basato su full finetuning. Per quanto riguarda la metodologia basata su full finetuning non è stato effettuato alcun tuning dell'ottimizzatore ma si è scelto di utilizzare l'ottimizzatore proposto in letteratura, ovvero AdamW. Per quanto rigurda invece la metodologia proposta, sono stati effettuati alcuni esperimenti utilizzando Adam e AdamW (tabella 4.2), a fronte dei quali AdamW si è riconfermato essere la scelta più adatta.

Loss di distillazione Per la *distillazione classica* si è scelto di seguire il setting proposto in letteratura effettuando i test con tutte e tre le loss presentate nella sezione 4.1.2: Cross Entropy, Mean Squared Error e Kullback-Leibler Divergence. Per quanto riguarda invece le metodologie di distillazione con feedback sono stati condotti alcuni esperimenti riassunti nella tabella 4.3. Come si può osserva-

Metodologia	Loss	Accuracy
Meta-Distillazione full-finetuning	Cross Entropy	**0.8183**
Meta-Distillazione full-finetuning	Kullback-Leibler Divergence	0.8107
Meta-Distillazione full-finetuning	Mean Squared Error	0.8262
Meta-Distillazione con LoRA	Cross Entropy	**0.8438**
Meta-Distillazione con LoRA	Kullback-Leibler Divergence	0.8405
Meta-Distillazione con LoRA	Mean Squared Error	0.8354

Tabella 4.3: Confronto tra le diverse loss di distillazione per le metodologie con feedback. Per ciascuna è riportato il miglior valore di accuracy del modello student ottenuto su 15 epoche di training (a parita di setting per tutti gli altri iperparametri).

re, la loss che sembra avere i risultati migliori è la Cross Entropy, seguita dalla Kullback-Leibler Divergence e infine dal Mean Squared Error.

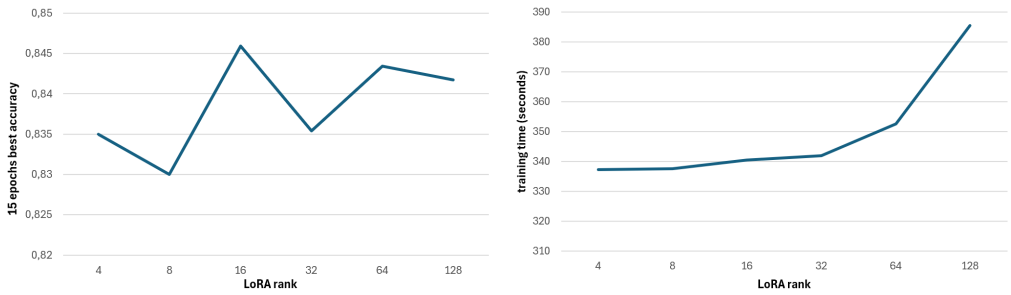


Figura 4.2: Test effettuati sul rango delle matrici LoRA. (Sinistra) Miglior valore di accuracy su 15 epoche al variare del rango delle matrici LoRA. (Destra) Tempo di training (15 epoche) al variare del rango delle matrici LoRA

LoRA rank Al fine di effettuare il tuning del rango delle matrici LoRA, sono stati condotti alcuni esperimenti volti a capire quale fosse il rango con il miglior trade off tra guadagno in termini di tempo di training e accuracy del modello distillato. I test condotti sono riportati sui grafici in figura 4.2, e da essi possiamo trarre le seguenti conclusioni: sebbene in termini di accuracy le oscillazioni al variare del rango siano poco pronunciate, l'utilizzo di un rango troppo elevato non risulta essere conveniente per via del significativo aumento del tempo di training

(in accordo anche agli esperimenti condotti in [17]). Concludiamo che il rango con il miglior trade off tra accuracy e tempo di training è pari a 16.

4.2.2 Confronto con lo stato dell'arte

Veniamo dunque al confronto della metodologia proposta con lo stato dell'arte. Considereremo tre diverse metodologie presenti in letteratura: *Training from scratch*, *Distillazione classica* (con tutte e tre le loss di distillazione utilizzate in letteratura), e *Meta-Distillazione* con full-finetuning del teacher. Per ciascuna metodologia considereremo le metriche Accuracy ed F1 score prese alla *quinta*, alla *decima*, alla *quindicesima* ed alla *ventesima* epoca. Successivamente andremo a confrontare le varie metodologie anche rispetto ad altri parametri.

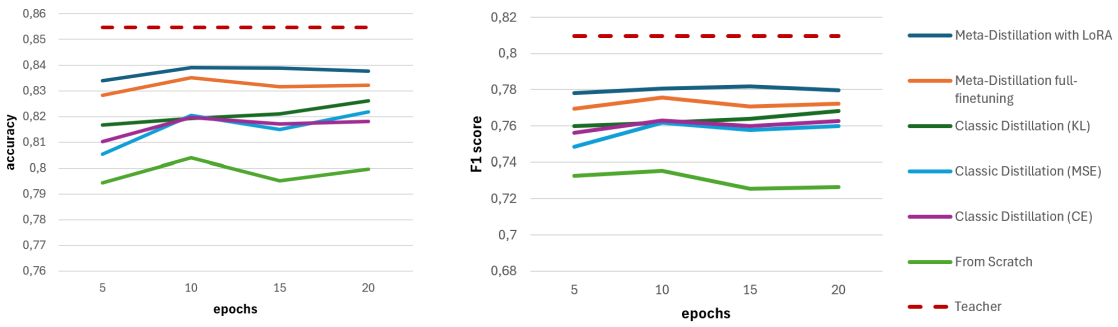


Figura 4.3: Confronto tra le metodologie in termini di accuracy (Sinistra) e f1 score (destra)

Performance sul task di downstream Per ciascuna metodologia, sono stati effettuati 5 run di training, e al termine di ciascuno sono state calcolate le metriche. Alla fine, per ciascuna metodologia sono state calcolate le performance medie in corrispondenza della quinta, decima, quindicesima e ventesima epoca. I risultati ottenuti sono rappresentati graficamente nella figura 4.3, come possiamo osservare la metodologia proposta risulta avere performance significativamente migliori rispetto alle tecniche di distillazione classica e di training from scratch. Inoltre, migliorano le performance anche rispetto alla metodologia di distillazione con

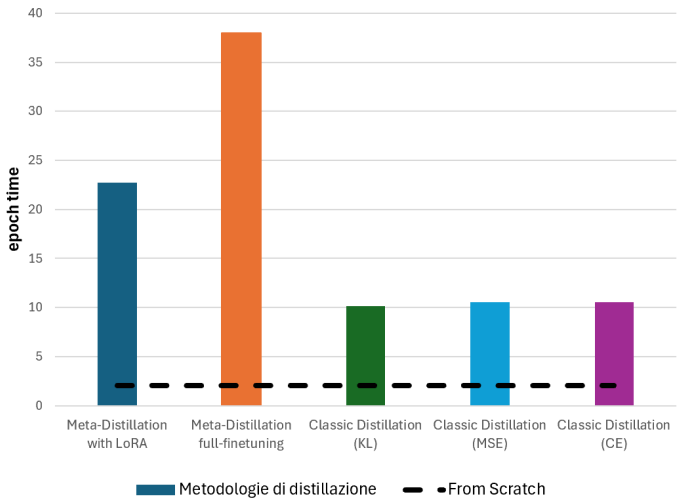


Figura 4.4: Confronto tra il tempo (in secondi) necessario a concludere un'epoca di training per ciascuna metodologia.

full-finetuning, anche se il gap è meno accentuato rispetto a quello osservato con le altre metodologie.

Tempo di training Vediamo adesso un confronto tra il tempo medio necessario a concludere un'epoca di ciascuna metodologia. A tal fine, consideriamo i tempi riportati in figura 4.4. Come da aspettative, il training from scratch risulta essere in assoluto la metodologia più veloce, al prezzo di performance significativamente peggiori rispetto alle altre metodologie. Anche la distillazione classica risulta essere significativamente più veloce rispetto alla metodologia di Meta-Distillazione, questo perchè coinvolgere il teacher nel processo di aggiornamento dei pesi rappresenta un overhead necessario per il miglioramento delle performance del modello distillato. Di particolare interesse, invece, è il confronto tra le due metodologie di distillazione con feedback. L'integrazione di LoRA nel processo di distillazione, permette di abbassare significativamente il costo di utilizzo del meccanismo di feedback, e allo stesso tempo, migliora leggermente le performance finali del modello distillato (come discusso nel paragrafo precedente). Al fine di validare in modo più accurato la metodologia proposta, sono stati condotti dei test per valutarne il comportamento in contesti *time costrained*. Fissato un bud-

get temporale per il training, confrontiamo le performance in termini di accuracy delle varie metodologie dopo un training temporalmente limitato al budget fissato.

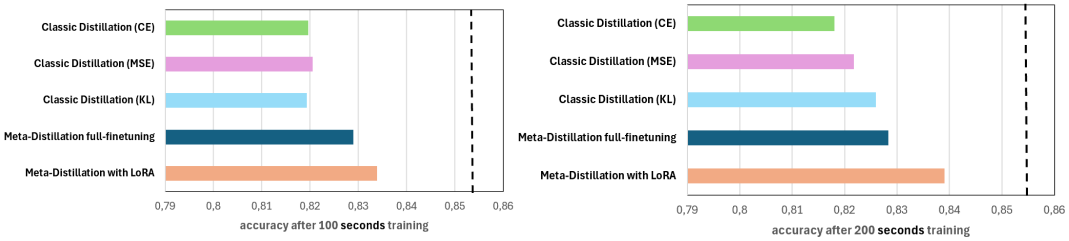


Figura 4.5: Valori di accuracy delle varie metodologie dopo un training della durata di rispettivamente 100 secondi (sinistra) e 200 secondi (destra). La linea tratteggiata nera rappresenta il target desiderabile, cioè, la performance del teacher.

La figura 4.5 riassume i risultati ottenuti sperimentando due limiti temporali: 100 e 200 secondi. In entrambi i casi, nonostante la metodologia di Meta-Distillazione basata su full finetuning performi meglio rispetto a quelle che non utilizzano il feedback, l'utilizzo di LoRA risulta ancora una volta essere la scelta più adeguata, permettendo di raggiungere performance migliori nello stesso tempo di training.

4.2.3 Implicazioni del meta-learning

In questa sezione approfondiremo alcuni effetti secondari, ma non meno importanti, derivanti dall'integrazione del meta-learning all'interno del processo di distillazione. In particolare, ci concentreremo su due aspetti: gli effetti del meta-learning sulla sensibilità del training alla temperatura (sezione 4.1.2), e gli effetti del meta-learning sulle prestazioni del teacher sul task di downstream.

Effetti del meta-learning sulla sensibilità alla temperatura Il tuning appropriato della temperatura, in un contesto di distillazione classica è fondamentale. Questo perchè, come già discusso in precedenza, se il teacher distribuisce meglio le probabilità sulle varie classi, il processo di distillazione viene agevolato. L'idea antestante gli esperimenti descritti di seguito è che l'integrazione del meccaniso

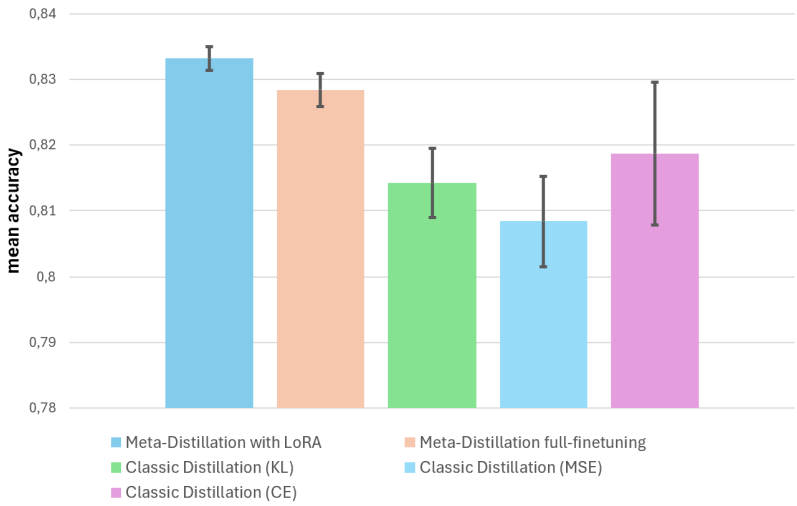


Figura 4.6: Valori medi di accuracy al variare della temperatura con relativa barra di errore.

di feedback tramite meta-learning possa consentire al teacher di aggiustare la sua distribuzione, rendendo meno rilevante il tuning della temperatura. Per esplorare questo fenomeno sono stati fissati quattro valori di temperatura: due, tre, quattro e cinque. Per ciascun valore di temperatura, e per ciascuna metodologia di distillazione sono stati effettuati cinque run. Raccolti tutti i dati è stata calcolata la *deviazione standard* sulle performance medie ottenute al variare della temperatura. I risultati ottenuti sono riassunti in figura 4.6, come possiamo osservare le metodologia che utilizzano il meta-learning (Meta-Distillation full-finetuning e Meta-Distillation with LoRA) hanno una deviazione standard molto più piccola rispetto alle metodologie che non ne fanno uso. Di conseguenza, dai test condotti è emerso che effettivamente, l'utilizzo del meta-learning rende il processo di distillazione meno sensibile alle variazioni di temperatura.

Effetti del meta-learning sulle performance del teacher sul task di downstream Riassumiamo brevemente cosa avviene durante il processo di distillazione con feedback. Per prima cosa il teacher distilla la sua conoscenza nello student, dopodiché lo student viene sottoposto ad un test sulla base del quale restituisce un feedback al teacher, il quale aggiusta i propri parametri in modo da rendere la sua

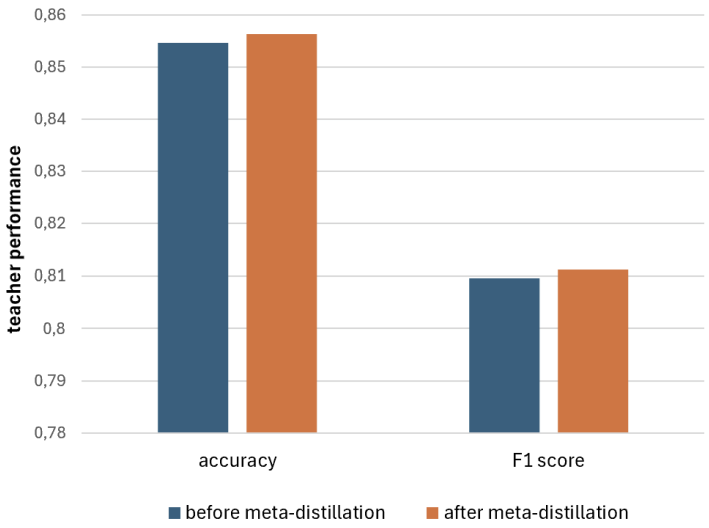


Figura 4.7: Performance del teacher prima e dopo il il processo di meta-distillazione con LoRA

conoscenza meglio distillabile. Dunque, durante l'intero processo il teacher non si addestra mai per performare meglio sul task di downstream. Tuttavia, il feedback introdotto sui parametri del teacher potrebbe avere anche un effetto indiretto sulle performance di tale modello sul task di downstream. In effetti, i logit in uscita dal teacher risentono direttamente della modifica indotta dal feedback, e una migliore capacità di distillazione potrebbe causare a sua volta un leggero miglioramento sul task rispetto al quale la distillazione stessa avviene. Effettivamente durante i test condotti, e in particolare per la metodologia di meta distillazione proposta in questo elaborato, si è osservato che le prestazioni del teacher sul task di downstream prima del training ammontavano a 0.8547 per la metrica accuracy e 0.8097 per la metrica F1 score. Dopo 15 epoche di meta-distillazione con LoRA, le performance del teacher sul task di downstream ammontano rispettivamente a 0.8564 per la metrica accuracy e 0.8112 per la metrica F1 score. Dunque, entrambe le metriche hanno registrato un incremento positivo.

Conclusioni

Questo elaborato ha discusso ampiamente l'importanza che i Large Language Model ricoprono attualmente nel campo dell'intelligenza artificiale, costituendo lo stato dell'arte su molti task afferenti all'ambito del deep learning. Sono state inoltre presentate le problematiche conseguenti dall'utilizzo di questi modelli, come le loro grandi dimensioni che ne impediscono il deploy in dispositivi a basse prestazioni, oppure il fatto che siano estremamente energivori e poco compatibili con uno sviluppo tecnologico ecosostenibile. Per risolvere questi problemi, è stata quindi presentata una nuova metodologia di distillazione, che rientra tra le metodologie di meta-distillazione di tipo white-box, e che permetta di avere un processo di distillazione più rapido ed efficiente senza sacrificare le performance del modello distillato. Nel primo capitolo sono state introdotte le principali strategie per la compressione dei modelli presenti in letteratura, distinguendo tra tecniche di pruning, di quantizzazione e di distillazione; presentando pro e contro di ciascuna. Nel secondo capitolo sono state presentate con un maggiore livello di dettaglio alcune particolari metodologie di distillazione, spaziando dall'integrazione di tecniche di eXplainable AI nel processo, fino all'introduzione di un meccanismo di feedback in un framework del secondo ordine per la meta-distillazione di modelli come gli LLM. Nel terzo capitolo, punto focale dell'elaborato, è stata delineata nel dettaglio la metodologia proposta. In particolare, è stato descritto come utilizzare il meta-learning in combinazione alla tecnica di finetunig efficiente Low-Rank Adaptation (LoRA), ed è stato mostrato come questo possa rendere la procedura di meta-distillazione più veloce ed efficiente. Nel quarto capitolo, dedicato alla valutazione sperimentale, è stato mostrato come la metodologia proposta

sia in grado di ridurre significativamente il tempo necessario alla distillazione, con la conseguente riduzione del consumo energetico del processo, migliorando allo stesso tempo le performance rispetto alla controparte che non fa uso delle tecniche di PEFT. Inoltre, sono stati osservati alcuni interessanti effetti derivanti dall'utilizzo del meta-learning nella procedura di distillazione, come un leggero aumento delle performance del modello teacher sulle metriche per il task di downstrem, e una minore sensibilità alle variazione nell'iperparametro temperatura. In conclusione, il presente elaborato vuole essere una proposta di avanzamento dello stato dell'arte in materia di distillazione di modelli di deep learning, con l'obiettivo di dirigersi verso uno sviluppo futuro in cui l'intelligenza artificiale possa essere di supporto in diversi ambiti del mondo reale mantenendo allo stesso tempo i requisiti in termini di risorse e di ecosostenibilità.

Elenco delle figure

Elenco delle tabelle

Bibliografia

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, and et al. Language models are few-shot learners, 2020.

[3] Riccardo Cantini, Alessio Orsino, and Domenico Talia. Xai-driven knowledge distillation of large language models for efficient deployment on low-resource devices. *Journal of Big Data*, 11, 05 2024.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[5] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011.

[6] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks, 2017.

[7] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot, 2023.

[8] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023.

[9] Elias Frantar, Sidak Pal Singh, and Dan Alistarh. Optimal brain compression: A framework for accurate post-training quantization and pruning, 2023.

[10] Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. Minillm: Knowledge distillation of large language models, 2024.

[11] Demi Guo, Alexander M. Rush, and Yoon Kim. Parameter-efficient transfer learning with diff pruning, 2021.

[12] Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, and Sai Qian Zhang. Parameter-efficient fine-tuning for large models: A comprehensive survey, 2024.

[13] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015.

[14] Namgyu Ho, Laura Schmid, and Se-Young Yun. Large language models are reasoning teachers, 2023.

[15] Lu Hou, Zhiqi Huang, Lifeng Shang, Xin Jiang, Xiao Chen, and Qun Liu. Dynabert: dynamic bert with adaptive width and depth. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[16] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp, 2019.

[17] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, and et al. Lora: Low-rank adaptation of large language models, 2021.

[18] Zhiqiang Hu, Lei Wang, Yihuai Lan, Wanyu Xu, Ee-Peng Lim, and et al. Llm-adapters: An adapter family for parameter-efficient fine-tuning of large language models, 2023.

[19] Yukun Huang, Yanda Chen, Zhou Yu, and Kathleen McKeown. In-context learning distillation: Transferring few-shot learning ability of pre-trained language models, 2022.

[20] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. Tinybert: Distilling bert for natural language understanding, 2020.

[21] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[22] Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard Hovy. Race: Large-scale reading comprehension dataset from examinations. *arXiv preprint arXiv:1704.04683*, 2017.

[23] Shiyang Li, Jianshu Chen, Yelong Shen, Zhiyu Chen, Xinlu Zhang, Zekun Li, Hong Wang, Jing Qian, Baolin Peng, Yi Mao, Wenhu Chen, and Xifeng Yan. Explanations from large language models make small reasoners better, 2022.

[24] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation, 2021.

[25] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration, 2024.

[26] Christos Louizos, Max Welling, and Diederik P. Kingma. Learning sparse neural networks through l0 regularization. In *International Conference on Learning Representations*, 2018.

[27] Rabeeh Karimi Mahabadi, James Henderson, and Sebastian Ruder. Compacter: Efficient low-rank hypercomplex adapter layers, 2021.

[28] Yuning Mao, Lambert Mathias, Rui Hou, Amjad Almahairi, Hao Ma, and et al. Unipelt: A unified framework for parameter-efficient language model tuning, 2022.

[29] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one?, 2019.

[30] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, and et al. Gpt-4 technical report, 2024.

[31] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2020.

[32] Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. Patient knowledge distillation for bert model compression, 2019.

[33] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3319–3328. PMLR, 06–11 Aug 2017.

[34] Yi-Lin Sung, Varun Nair, and Colin Raffel. Training neural networks with fixed sparse masks, 2021.

[35] Raphael Tang, Yao Lu, Linqing Liu, Lili Mou, Olga Vechtomova, and Jimmy Lin. Distilling task-specific knowledge from bert into simple neural networks, 2019.

[36] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 2023.

[37] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding, 2019.

[38] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions, 2023.

[39] Ziheng Wang, Jeremy Wohlwend, and Tao Lei. Structured pruning of large language models. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2020.

[40] Chuanpeng Yang, Wang Lu, Yao Zhu, Yidong Wang, Qian Chen, Chenlong Gao, Bingjie Yan, and Yiqiang Chen. Survey on knowledge distillation for large language models: Methods, evaluation, and application, 2024.

[41] Wangchunshu Zhou, Canwen Xu, and Julian McAuley. Bert learns to teach: Knowledge distillation with meta learning, 2022.