Approfondimento di Intelligenza Artificiale

Studente: Tristano Munini

ANNO ACCADEMICO 2019-2020

1 Introduzione

2 GAN

Come illustrato nella lezione del MIT [**MIT'GEN**], uno tra i primi metodi che ha permesso la generare immagini sintetiche faceva uso di una particolare versione di *Auto-Encoder*(AE), il *Variational Auto-Encoder* (VAE). Come nel caso degli AE classici, i VAE hanno una struttura che ricorda una clessidra: la prima metà della rete permette di comprimere l'input, mappandolo in quello che viene chiamato spazio latente, di minor dimensione rispetto allo spazio di partenza; la seconda metà, invece, prende l'input compresso e lo mappa nello spazio di partenza. Il prodotto della decompressione viene chiamato "input ricostruito". Durante il training si vuole ottimizzare la compressione in modo che non ci sia perdita di informazione, ciò viene effettuato minimizzando la distanza tra input originale ed input ricostruito. Nei VAE, in corrispondenza del punto della rete in cui si raggiunge il livello massimo di compressione (*bottleneck*), invece di essere generato il vettore compresso z, viene prodotta una coppia di vettori σ e μ che descrivono una distribuzione di probabilità dei vettori compressi. In questo modo è possibile campionare z dalla distribuzione appena prima della decompressione. Notare che l'obiettivo dei VAE è differente da quello degli AE: con gli AE si vuole trovare una funzione di compressione in modo *unsupervised*; mentre con i VAE si vuole ottenere z che siano vicini alla compressione del dato di partenza per poi decomprimerli. Solitamente, dopo aver alleato un AE si sarà interessati principalmente nell'encoder, perché si vuole ottenere z, invece quando si allena un VAE si è interessati tanto all'enecoder quanto al decoder perché l'obiettivo è generare dati che siano variazione di quello di partenza. Il campionamento effettuato nel *bottleneck* non è un'operazione differenziabile e questo rende inapplicabile l'algoritmo della *backpropagation*, quindi risulta necessario effettuare quello che viene chiamato *reparametrization trick*. Rappresentando z come $z = \mu + \sigma \odot \varepsilon$ in cui $\varepsilon \sim \mathcal{N}(0, 1)$, ε campionata da una distribuzione normale, è possible effettuare l'operazione di *sampling* all'esterno della rete. In questo modo μ e σ possono essere utilizzati per il calcolo del gradiente e quindi usati durante la *backpropagation*. Un VAE allenato, poiché ha un *bottleneck* non deterministico, permette di generare immagini con *feature* simili a quelle fornite durante il training, inoltre è possibile modificare direttamente i valori del vettore z per osservare che tipo di informazione rappresentano una volta decompressi. Nonostante questo tipo di rete sia in grado di dare risultati interessanti e permetta di comprendere meglio il significato degli spazi latenti, è strettamente legata alla dimensione del *bottleneck*. Quest'ultimo è un *hyperparameter* perché dipende dalla forma della rete, quindi deve essere cercato manualmente.

Le prestazioni dei VAE sono state superate da quelle delle GAN. Le *Generative Adversarial Network* (GAN) sono composte da due modelli che, citando [**GANTF**], *"vengono addestrati simultaneamente da un processo contraddittorio. Un generatore ("l'artista") impara a creare immagini che sembrano reali, mentre un discriminatore ("il critico d'arte") impara a distinguere le immagini reali dai falsi"*. Riformulando la frase si può dire che una GAN, come si vede in Figura 1, è composta da due reti: la prima viene chiamata Generatore G ed il suo scopo è fornire in output un x_{fake} che sembri appartenere alla distribuzione

del *dataset* di dati reali fornito; la seconda, detta Discriminatore D, prende in input un x_{fake} oppure un x_{real} estratto dal *dataset* e deve riuscire determinare se è reale oppure sintetico. Notare che le GAN non sono strettamente limitate ad immagini, infatti lo scopo dell'architettura è quello di produrre una buona approssimazione della distribuzione di probabilità del dataset attraverso una metrica dinamica determinata da D. L'idea di partenza è molto semplice però la convergenza del modello non è scontata e sono noti svariati problemi, alcuni dei quali illustrati in [**HARD˙GAN**].

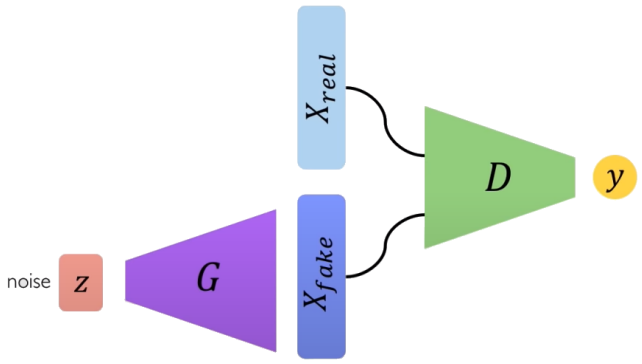

Figura 1: Architettura di una generica GAN (dalle slide in [**MIT˙GEN**])

Il training viene svolto in quattro momenti:

- a partire da del rumore casuale z, il generatore G, basandosi sulle sue conoscenze attuali, produce un x_{fake};
- successivamente D riceve un x_{fake} ed un x_{real} e deve categorizzali correttamente;
- il *ground truth* è noto, quindi è possibile effettuare la *backpropagation* su D;
- similmente G viene aggiornato a partire dalla probabilità $D(x_{fake})$ cioè quanto il discriminatore è convinto che x_{fake} sia reale. Se questo valore è alto significa che il generatore sta imparando correttamente.

Notare come G sia in grado di mappare del rumore casuale nello spazio delle *feature* del dominio e che quindi, una volta allenato, possa essere sfruttato molto facilmente: basterà avere del rumore da cui partire. In questo senso c'è una certa somiglianza con quanto succede nella seconda parte dei VAE: G opera come un decoder in cui lo spazio latente è molto semplice.

Ci si accorge che G e D hanno obiettivi opposti: il primo vuole confondere il secondo, mentre questo vuole evitare di venire confuso. Questo gioco di minmax può essere riassunto in formule con:

$$min_G\, max_D\, V(D,G) = \mathbb{E}_{x\sim P_{real(x)}}[logD(x)] + \mathbb{E}_{z\sim P_z(x)}[log(1 - D(G(z))]$$

D vuole massimizzare il valore atteso della probabilità (in questo caso logaritmica) che un x estratto dal dataset sia effettivamente riconosciuto come reale ed allo stesso tempo vuole massimizzare anche $1 - D(x)$ quando $x = G(z)$ è sintetico. Quindi nel caso del dato reale $D(x)$ dovrà avvicinarsi ad 1, mentre nell'altro caso si vorrà $D(G(z)) \simeq 0$. Al contempo G vuole minimizzare la formula e, dato che può operare solo sul secondo addendo, tenderà ad indurre $D(G(z)) \simeq 1$.

Rispetto ai VAE le GAN risultano non solo più intuitive ma permettono anche di raggiungere prestazioni veramente sorprendenti, come si può vedere in [**GAN˙HD**], in cui anche gli esperti umani vengono ingannati dai prodotti della rete.

Va fatto notare però che le GAN sono notoriamente difficili da allenare, come riportato in [**HARD˙GAN**]. Già ad un livello intuitivo si può capire che partendo da reti senza conoscenze pregresse è molto difficile raggiungere buone prestazioni: entrambe le sotto-reti non hanno alcuna conoscenza del dominio e si chiede loro di guidarsi a vicenda. Quindi in molte applicazioni, ove possibile, conviene effettuare un *pre-train* dei singoli G e D ed collegarli successivamente per formare l'architettura GAN. Questo accorgimento comunque non evita un altro problema ben noto: quando il discriminatore converge rapidamente è possibile che dia una valutazione così bassa al generatore da rendere impossibile che G possa migliorarsi. Quando $D(G(z))$ è pressoché zero si rischia il problema del *vanishing gradient*, quindi i pesi di G vengono aggiranti di un valore troppo piccolo bloccando l'esplorazione dello spazio di ricerca.

Un altro problema difficile da evitare è quello del *model collapse* che si verifica quando G riproduce molto bene soltanto una piccola frazione del dominio. In questo modo riesce ad ottenere punteggi molto alti a discapito della generalità. A seconda del dominio di applicazione il *model collapse* può essere un problema più o meno grave e richiedere accorgimenti specifici. Un caso particolare che preclude la convergenza della GAN si ha quando G si specializza per confondere D in un modo specifico, poi D si aggiorna per difendersi contro quel modo specifico, a quel punto G si specializza su un ulteriore modo specifico. Se i modelli continuano questo "inseguimento" non convergeranno mai ad una soluzione utile.

3 RL

Il *Reinforcement Learning* (RL), assieme al *Supervised Learning* ed all'*Unsupervised Learning*, è il terzo paradigma di apprendimento autonomo. Gli ultimi due paradigmi sfruttano un dataset, rispettivamente con e senza label, per portare a termine un specifico task oppure generare nuova informazione. Nel caso del RL il dataset viene sostituito con un ambiente (*environment*) nel quale il modello può eseguire delle azioni e osservarne le conseguenze, quindi come l'ambiente viene modificato dall'azione appena compiuta. In questo paradigma il modello viene anche chiamato "agente" e l'elenco, discreto oppure continuo, delle azioni eseguibili prende il nome di *action space*. L'agente impara grazie ad una ricompensa (*reward*), positiva o negativa, associata al risultato delle sue azioni. Il suo obiettivo è quello di massimizzare la somma dei *reward* sul lungo termine, quindi si vuole che scopra e adotti una strategia (*policy*) efficace nell'ambiente considerato. Poiché è necessario svolgere numerose iterazioni del tipo *trial-and-error* e considerato che solitamente un fallimento corrisponde anche ad una grande acquisizione di informazione, bisogna creare degli *environment* virtuali che siano il più vicino possibile al dominio di applicazione finale e che permettano un'esecuzione rapida e senza costi aggiuntivi.

Riprendendo quanto illustrato in [**MIT˙RL**] ed in [**Simple˙RL**], una prima modellazione sfrutta una funzione che valuta la qualità dell'azione svolta

$$Q(s_t, a_t) = \mathbb{E}[R_t | s_t, a_t]$$

in cui s_t è lo stato corrente cioè come l'ambiente si presenta all'agente, a_t è l'azione che l'agente svolge all'istante t, mentre a destra dell'uguale si ha il valore atteso del *reward* totale R_t che l'agente potrà ricevere in futuro, quindi negli s_{t+i} con $i = 1, 2, \ldots$ se esegue a_t in s_t. Calcolare R_t risulta critico perché la sua naturale definizione è

$$R_t = \sum_{i=t}^{\infty} \gamma^i r_i = \gamma^{t+1} r_{t+1} + \gamma^{t+2} r_{t+2} + \gamma^{t+3} r_{t+3} + \ldots \gamma^{t+n} r_{t+n} + \ldots$$

in cui $0 < \gamma < 1$ viene chiamato *discount factor* ed indica la *greediness* del modello. In questa casistica risulta utile approssimare $Q(s, a)$ con una rete neurale, in questo modo si evita di dover fissare a mano un *hyperparameter* come, ad esempio, il numero di somme da effettuare per approssimare R_t. L'idea è fornire alla rete, detta anche *Q-Network* [**DQN**], lo stato corrente ed ogni possibile azione lecita e successivamente scegliere l'azione a cui la rete assegna il valore più alto.

$$argmax_a Q(s_t, a)$$

Procedendo in questo modo per ogni stato che si incontra s_{t+i} con $i = 1, 2, \ldots$ è possibile seguire una policy $\pi(s)$ ottimale ad ogni passo. La policy $\pi(s)$ è una politica di scelta che associa uno stato ad un'azione, con lo scopo di ottenere un buon guadagno sul lungo termine. Si trova una strategia ottimale quando ogni scelta fatta seguendo la π è ottimale.

L'approccio appena presentato richiede che l'*action space* sia discreto e di dimensione ridotta, altrimenti sarebbe impossibile o molto costoso iterare su tutte le azioni per selezionarne la migliore. Per ovviare a questo problema si possono utilizzare modelli che provano ad ottimizzare direttamente la *policy* $\pi(s)$, questi modelli prendono il nome di *Policy Learning* e vengono allenati tramite il *Policy Gradient*. In questo modo è possibile ottenere direttamente l'azione migliore a partire soltanto dallo stato. L'idea principale è approssimare π con una distribuzione di probabilità, in questo modo risulta naturale utilizzare un *action space* continuo, inoltre si può ottenere una strategia non deterministica, quindi più flessibile e con maggiori capacità esplorative durante il training. Il non determinismo è introdotto quando, per scegliere l'azione da eseguire, se ne estrae una dalla distribuzione di probabilità data dalla *policy*. In formule, dato uno stato s l'azione a verrà estratta secondo:

$$a = \pi(s) \sim P(a|s) \quad \text{in cui} \int_{a=-\infty}^{\infty} P(a|s) = 1$$

Poiché $P(a|s) = \mathcal{N}(\mu, \sigma^2)$ si può fare in modo che la rete dia in output direttamente il vettore della medie μ e il vettore delle varianze σ^2. L'allenamento incomincia con l'inizializzazione dell'agente e dell'ambiente, poi l'agente segue la sua policy corrente fino a terminazione mentre tutti gli stati, le azioni e i reward vengono registrati. Infine si aumenta la probabilità delle azioni che hanno portato a reward positivi e si decrementa la probabilità delle azioni con reward negativo, fatto ciò si effettua nuovamente l'inizializzazione in modo da valutare la policy aggiornata. Notare che definire il concetto di "terminazione" non è sempre ovvio e può corrispondere, ad esempio, all'esecuzione di un numero limitato di azioni oppure al ricevimento di un grande reward negativo. L'aggiornamento dei pesi della rete, quindi della policy, viene effettuato tramite *gradient descent* usando la *loss*

$$loss = -logP(a_t|s_t)R_t$$

Il logaritmo indica la probabilità logaritmica con cui a_t viene scelta allo stato s_t mentre R_t è il valore atteso del reward totale che si guadagnerà. Se R_t è positivo allora converrà aumentare la probabilità di selezionare a_t, viceversa se è negativo conviene diminuire la probabilità. Il *Policy Gradient* è il gradiente relativo a questa particolare *loss*. Rimane comunque critico il calcolo di R_t. Usando i *Monte Carlo Rollouts* è possibile approssimare questo valore. In particolare, a partire dallo stato corrente, vengono generate ("srotolate" da *rollout*) svariate strategie seguendo la *policy* corrente. Così facendo si può stimare un range di valori raggiungibili ed effettuare una media di questi valori. L'algoritmo appena descritto prende il nome di REINFORCE ed è può essere diviso in quattro passaggi:

1. dallo stato corrente s_t si esegue varie volte ϕs_t fino a terminazione;
2. si effettua la media dei *reward* ottenuti da ogni esecuzione;

3. si usa il valore ottenuto per aggiornarne la rete;

4. si estrae un azione a_t dalla distribuzione ϕs_t e si ripete dal punto 1.

Nell'ambito del *Reinforcement Learning* spesso i termini *explore* ed *exploit* vengono usati per indicare due momenti diversi del train della rete. Una rete viene usata in modalità *explore* quando si vuole che sperimenti strategie nuove. In questa modalità si cerca di trovare soluzioni alternative, sperando che siano migliori di quella corrente. La modalità *exploit*, invece, viene usata quando si vuole sfruttare le conoscenze apprese e portare a termine il compito secondo quella che è ritenuta la strategia migliore.

Quasi obbligatorio citare i risultati ottenuti da AlphaGO [**AlphaGO**] ed AlphaZero [**AlphaZero**] attraverso il RL.

dire quattro parole?

4 GAN+RL per testi

In questa sezione vengono illustrati due modelli capaci di generare testi sintetici sfruttando un'architettura GAN in cui G viene allenato attraverso *Reinforcement Learning*. Il primo modello, chiamato SeqGAN, è stato presentato in [**SeqGAN**] ed illustrato anche in [**GAN˙for˙text**]; il secondo è evoluzione del primo, permette di generare testi più lunghi, prende il nome di LeakGAN ed è descritto in [**LeakGAN**]. Si vuole anche citare [**NetTextGen˙Review**] in cui vengono illustrati alcuni modelli usati prima delle SeqGAN e quelli sviluppati successivamente fino ad arrivare alle LeakGAN. Nell'articolo si trova anche un confronto tra MaliGAN, RankGAN, MaskGAN e TextGAN.

4.1 SeqGAN

Come riportato nell'introduzione dell'articolo [**SeqGAN**], per generare frasi che siano verosimili è necessario allenare un discriminatore che valuti frasi intere e che assegni a queste un punteggio. Purtroppo ciò rende molto difficile allenare il generatore, perché non è possible determinare se un punteggio basso corrisponde all'intera struttura della frase oppure soltanto ad una o poche parole. La problematica è ancora più evidente nel caso in cui il generatore è una RNN rendendo difficile, ad esempio aggiornare efficacemente il modo con cui vengono create le parti iniziali di frasi.

mai introdotte per ora, TODO da fare mini sezione sopra?

Le SeqGAN affrontano il problema in un modo molto interessante: se si considera il punteggio che D fornisce alle frasi come *reward* per G e se questo utilizza come stato la frase generata fino ad ora e come azione la scelta della parola successiva, allora è possibile sfruttare il *Policy Gradient* sul generatore. Di fondamentale importanza la *Monte Carlo Search* con *Rollout* che viene effettuata per valutare la bontà di frasi incomplete, così da alterare efficacemente la distribuzione della parola che ancora deve essere scelta: durante la generazione di una frase, G non può ricevere una valutazione da D perché il discriminatore è in grado di valutare soltanto frasi intere quindi vengono generate N frasi con prefisso la frase generata fino ad ora. Si sfrutta poi D per valutare tutte le N frasi e si effettua una media dei *reward* ottenuti, così si ottiene il valore atteso della bontà della frase che si sta generando. Ci si riferisce a questo furbo accorgimento come *Monte Carlo state-action search*.

Riprendendo i formalismi usati nell'articolo si ha:

- un modello generativo G_θ, con θ si indica i parametri interni, in grado di generare sequenze $Y_{1:T} = (y_1, \dots, y_t, \dots, y_T)$ con gli y_t appartenenti all'insieme dei token validi $\mathbb{T}$;
- al tempo t lo stato s equivale ai token prodotti fino ad ora $(y_1, \dots, y_{t-1})$ mentre l'azione a è il prossimo token da selezionare y_t;
- con $G_\theta(y_t|Y_{1:t-1})$ si indica il modello non deterministico descritto.
- Il modello discriminativo D_ϕ, con parametri ϕ, è in grado di fornire la probabilità $D_\phi(Y_{1:T})$ che $Y_{1:T}$ sia stato estratto dai dati reali.

Prima di continuare con la *loss function* e la formulazione della *Monte Carlo Search*, va sottolineato che il modello RNN è leggermente diverso da quello classico, infatti ad ogni passo la rete prende in input il token generato al passo precedente anziché riceverlo dall'esterno. Si può quindi dire che assomigli ai modelli RNN usati come decoder durante la traduzione di testi, nei quali lo stato interno e l'ultima parola tradotta vengono utilizzati per aggiornare lo stato e generare la parola successiva. Il primo token, o stato di partenza, è un token particolare che si indica con s_0. Lo stato h_0 di partenza può essere fissato oppure selezionato casualmente in modo da modificare il punto di partenza (simili all'input z per i VAE). L'obiettivo del generatore G_θ è quello di produrre una sequenza a partire dallo stato s_0 che massimizzi il *reward* totale, in formule:

$$J(\theta) = \mathbb{E}[R_t|s_0, \theta] = \sum_{y_1 \in \mathbb{T}} G_\theta(y_1|s_0) \cdot Q_{D_\phi}^{G_\theta}(s_0, y_1)$$

in cui $Q_{D_\phi}^{G_\theta}(s, a)$ è la funzione che indica il *reward* accumulabile eseguendo l'azione a allo stato s e seguendo la *policy* G_θ nei passi successivi. Questa funzione dovrà necessariamente essere stimata, perché sappiamo che D_ϕ non può essere sfruttato su sequenze incomplete. Quindi si utilizza una N-*Monte Carlo Search* con *Rollout* per stimare N volte i $T - t$ token mancanti

$$\{Y_{1:T}^1, \ldots, Y_{1:T}^N\} = MC(Y_{1:t}; N)$$

Gli $Y_{t+1:T}^n$ con cui si completa la sequenza parziale sono campionati usando la stessa *policy* G_θ. Quindi la stima del *reward* atteso è data da

$$Q_{D_\phi}^{G_\theta}(s = Y_{1:t-1}, a = y_t) = \begin{cases} \frac{1}{N} \sum_{n=1}^{N} D_\phi(Y_{1:T}^n),\ Y_{1:T}^n \in MC(Y_{1:t}; N) & \text{for } t < T \\ D_\phi(Y_{1:t}) & \text{for } t = T \end{cases}$$

Per quanto riguarda il discriminatore D_ϕ viene specificato che l'aggiornamento dei suoi parametri ϕ viene effettuato solo quando il generatore ha creato un numero sufficiente di sequenze. In questo modo è possibile avere un discriminatore che si adatta e migliora assieme al generatore, pur lasciandogli il tempo di perfezionarsi. In formule D_ϕ viene allenato secondo:

$$min_\phi - \mathbb{E}_{Y \sim p_{real}}[logD_\phi(Y)] - \mathbb{E}_{Y \sim G_\theta}[log(1 - D_\phi(Y))]$$

L'algoritmo del train illustrato in [**SeqGAN**] è riportato in Algorithm 1. È molto importante sottolineare il *pre-train* effettuato per inizializzare la SeqGAN con alcune conoscenze basilari. In questo modo G e D saranno già capaci di svolgere i loro compiti e potranno migliorarsi più efficacemente. Il *pre-train* del generatore viene effettuato usando la *Maximum Likelihood Estimation*(MLE) sul dataset di sequenze reali, G tenterà quindi di imitare nel miglior modo possibile la distribuzione dei token delle sequenze date. Mentre D viene allenato come un classificatore attraverso la *Cross Entropy Loss* su dati reali e dati generati dal G appena creato. Ovviamente il *train* di D viene sempre effettuato su un

Algorithm 1 Sequence Generative Adversarial Nets

Initialize G_θ, D_ϕ with random weights θ, ϕ
Pre-train G_θ using MLE on real data
Generate negative samples using G_θ for training D_ϕ
Pre-train D_ϕ via minimizing the cross entropy
repeat
    for g-steps **do**
        Generate a sequence $Y_{1:T} = (y_1, \ldots, y_T) \sim G_\theta$
        for t in $1 : T$ **do**
            Compute $Q_{D_\phi}^{G_\theta}(s = Y_{1:T}; a = y_t)$
        end for
        Update generator parameters via policy gradient
    end for
    for d-steps **do**
        Use current G_θ to generate negative examples and combine with given positive examples
        Train D_ϕ for k epochs
    end for
until SeqGAN converges

insieme di sequenze per metà generato e per metà reale, così da non introdurre sbilanciamenti nelle probabilità. Interessate sottolineare che il discriminatore non è una *Deep Neural Network* (DNN), come ci si potrebbe aspettare, ma una *Convolutional Neural Network* (CNN). Nell'articolo viene spiegato come queste riescano a mantenere un'informazione localizza e quindi a creare legami tra parole vicine. Per poter applicare una CNN risulta necessario organizzare le frasi forma matriciale.

maggiori dettagli sulla matrice

In [**SeqGAN**] vengono utilizzate anche tecniche come *Dropout* e *L2 regularization* per evitare l'*over-fitting*. La prima è una tecnica molto conosciuta che permette di evitare che la rete impari "a memoria" la distribuzione target. Con il *Dropout* si va ad azzerare casualmente una percentuale dei pesi della rete, in questo modo la si obbliga ad astrarre maggiormente l'informazione. Inoltre questo rafforza la resistenza e l'efficienza della rete perché sarà in grado di portare a termine il compito anche in mancanza di nodi interni. Con la *L2 regularization* si effettua una scolatura dell'errore così da evitare il *gradient vanishing*. In [**SeqGAN**] è anche possibile trovare una valutazione dettagliata delle prestazioni delle SeqGAN rispetto ad altri modelli e su tre casid'uso differenti.

ripassare L2

4.2 LeakGAN

Le LeakGAN sono state create per far fronte alla principale debolezza delle SeqGAN, ossia la difficoltà nel generare sequenze lunghe che siano convincenti. Se gli esperimenti delle SeqGAN mostravano affidabilità con sequenze fino a 20 to-

ken, le LeakGAN riescono a raggiungere lunghezze di 40 token, pur mantenendo coerenza e verosimiglianza. Queste reti vengono presentate in [**LeakGAN**] e si differenziano dalle precedenti per due motivi:

- si introduce una "perdita" (*leak*) di informazione dal discriminatore al generatore. Le *feature* che il primo estrae e su cui poi baserà la valutazione vengono fornite al secondo in modo da ricevere un'informazione molto più ricca di un semplice punteggio;
- si introduce anche un nuovo modulo all'interno del generatore in modo da elaborare l'informazione che giunge da D ed utilizzarla per poi decidere il token successivo.

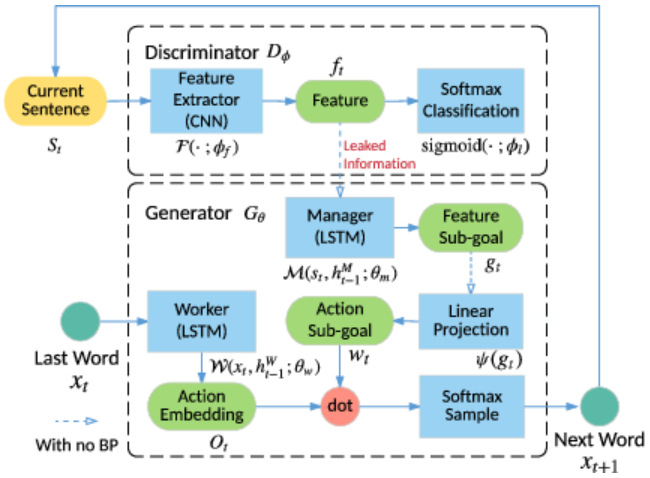

Figura 2: Architettura di una LeakGAN [**LeakGAN**]

Va subito fatto notare come la seconda modifica renda il generatore un generatore gerarchico, quindi composto da sottomoduli con specifici compiti. È altrettanto importante sottolineare che i sotto-compiti che il Manager richiede al Worker sono auto-determinati, infatti i risultati ottenuti dimostrano che il Manger è in grado di richiedere al Worker la generazione di punteggiature e particolari strutture.

La *Linear Projection* presente nello schema effettua una trasformazione lineare ψ, con pesi W_ψ, su un numero c di goal g_t recenti, così da generare un vettore w_t di dimensione adeguata per l'esecuzione del prodotto con O_t.
In formule si ha

$$w_t = \psi\left(\sum_{i=1}^{c} g_{t-1}\right) \tag{1}$$

$$O_t, h_t = Worker(x_t, h_{t-1}; \theta) \tag{2}$$

$$x_{t+1} = G_\theta(\cdot|s_t) = softmax(O_t \cdot w_t/\alpha) \tag{3}$$

in cui h_t è *hidden state* del Worker, mentre α viene usato per bilanciare esplorazione e sfruttamento (*exploration and exploitation*). In generale avrà un valore alto durante il *training* per favorire l'esplorazione, avrà invece un valore basso durante la generazione di quelle sequenze che poi verranno usate per allenare D.

spiegare meglio train G? Rescaled R_t?

Un'altra importante modifica riguarda l'*interleaved training*: si alternano allenamento tramite GAN ed allenamento tramite metodo supervisionato (MLE), anziché effettuare soltanto GAN dopo il *pre-train*. Questo evita il *mode collapse* obbligando G a rimanere aderente alla vera distribuzione degli esempi reali, evitando quindi che si specializzi su casi particolari con cui confondere D. La modifica viene mostrata in modo esplicito a riga 4 del pseudocodice **Algorithm 2**. Notare come vengano mostrati esplicitamente i parametri del Worker e del Manager, specificando che vengono aggiornati indipendentemente.

Algorithm 2 Adversarial Training with Leaked Information

1: Initialize G_{θ_m,θ_w}, D_ϕ with random weights θ_m,θ_w, ϕ
2: Pre-train D_ϕ on real data and generated data
3: Pre-train G_{θ_m,θ_w} using leaked information from D_ϕ
4: Perform the two parts of pre-training interleavingly until convergence
5: **repeat**
6: **for** g-steps **do**
7: Generate a sequence $Y_{1:T} = (y_1, \ldots, y_T) \sim G_{\theta_m,\theta_w}$
8: **for** t in $1:T$ **do**
9: Store leaked information from D_ϕ
10: Get $Q(f_t, g_t)$ by Monte Carlo Search
11: Get the computed direction g_t from MANAGER
12: Update WORKER parameters θ_w, ψ, softmax
13: Update MANAGER parameters θ_m
14: **end for**
15: **end for**
16: **for** d-steps **do**
17: Use current G_{θ_m,θ_w} to generate negative examples
18: Train D_ϕ for k epochs on generated examples and real data
19: **end for**
20: **until** LeakGAN converges

I risultati riportati in [**LeakGAN**] mostrano come le LeakGAN siano in grado di superare le già buone prestazioni delle SeqGAN su sequenze di 20 token e come le superino notevolmente con sequenze lunghe 40 token. Interessante anche il grafico ottenuto tramite PCA che permette di ottenere una visualizzazione della capacità delle LeakGAN di raggiungere lo spazio delle feature dei dati reali.

In Tabella 1 vengono esposti quattro esempi generati da LeakGAN e SeqGAN allenate con le didascalie del dataset COCO.

LeakGAN
(1) A man sitting in front of a microphone with his dog sitting oh his shoulder. (2) A young man is holding a bottle of wine in his hand.
SeqGAN
(1) A couple of kids in a bathroom that is in a bathroom. (2) A bathroom with tiled walls ad shower on it.

Tabella 1: Esempi generati da modelli alleanti con *COCO Image Captions*

5 Implementazione

Il codice delle LeakGAN può essere trovato nella repository GitHub https://github.com/CR-Gjx/LeakGAN. Nonostante ci siano implementazioni più recenti in `Python3` ed in `PyTorch` (https://github.com/nurpeiis/LeakGAN-PyTorch si è preferito analizzare il codice originale. Come indicato nella pagina principale del repository si hanno le seguenti dipendenze

- `Tensorflow r1.2.1`
- `Python 2.7`
- `CADA 7.5+ (For GPU)`

Per poter eseguire il codice è stato utilizzata un'immagine docker scaricabile ed eseguibile con

```
> docker run --rm -p 8888:8888 -v ~/LeakGAN:/LeakGAN
    tensorflow/tensorflow:1.2.1-gpu
```

L'immagine viene eseguita con accesso alla porta `8888`, sulla quale viene lanciato automaticamente Jupyter Notebook (http://localhost:8888/tree). L'immagine contiene già alcuni notebook con dei tutorial di TensorFlow. Usando il terminale di Jupyter ci si può spostare con `cd /LeakGAN`, cartella esterna all'immagine, collegata tramite il comando `-v ~/LeakGAN:/LeakGAN`. La repository fornisce tre esempi (Image COCO, No Temperature e Synthetic Data) nonostante i modelli `LeakGANModel.py` e `Discriminator.py` siano pressoché identici in ogni esempio. Gli esempi possono essere eseguiti con `python Main.py` nella relativa cartella.

Si vuole subito sottolineare che non è stato possible svolgere un train completo della rete, infatti la macchina su cui è stato eseguito il codice rendeva i tempi d'attesa inaccettabili. Si è anche tentato di ridurre la dimensione del dataset ImageCOCO (quindi anche della quantità di dati sintetici generati) ma senza riscontri positivi riguardo al tempo d'esecuzione.

La classe `Discriminator` del file `Discriminator.py` espone metodi per effettuare la *feature extraction* del vettore f_t e la classificazione di un input dato. La classe `LeakGAN` del file `LeakGAN.py` espone molti più metodi, tra cui si possono trovare:

- il costruttore della classe permette di specificare vari parametri tra cui il discriminatore che si vuole usare, la lunghezza delle sequenze, la dimensione del vocabolario dei token e la dimensione dello spazio latente a cui il vettore z appartiene;
- una funzione per effettuare il *rollout* secondo le modalità descritte sopra, specificando input di partenza ed N;
- una funzione che permette di generare una sequenza sintetica;
- funzioni per la creazione di Worker e Manager.