Dispense di Intelligenza artificiale

Emanuele Izzo, 0307385

Corso di laurea triennale Informatica
Università Tor Vergata, Facoltà di Scienze MM.FF.NN.
04/08/2020

Documento realizzato in LaTeX

Indice

Capitolo 1

Introduzione

L'intelligenza artificiale si occupa della comprensione e riproduzione del comportamento intelligente, tramite la costruzione di entità intelligenti. Lo studio dell'intelligenza può seguire due approcci principali:

- L'approccio della psicologia cognitiva (dove l'intelligenza artificiale è vista come una scienza [*intelligenza artificiale forte*]), dove l'obiettivo è quello di comprendere l'intelligenza umana tramite la creazione di modelli computazionali e verifiche sperimentali, riuscendo così a risolvere i problemi con gli stessi processi usati dall'uomo;
- L'approccio ingegneristico (dove l'intelligenza artificiale è vista in ambito ingegneristico [*intelligenza artificiale debole*]), dove l'obiettivo è quello di costruire entità dotate di razionalità tramite la codifica del pensiero e comportamento razionale, riuscendo così a risolvere problemi che richiedono intelligenza.

1.1 Agire umanamente: l'approccio del testi di Turing

> L'arte di creare macchine che svolgono funzioni che richiedono intelligenza quando svolte da esseri umani
>
> ---
>
> *Kurzweil, 1990*

Il test di Turing, proposto da Alan Turing nel 1950, è stato concepito per fornire una soddisfacente definizione operativa dell'intelligenza. Un computer passerà il test se un esaminatore umano, dopo aver posto alcune domande in forma scritta, non sarà in grado di capire se le risposte provengono da una persona o no. Programmare una macchina in grado di superare il test applicato in modo rigoroso richiede una sacco di lavoro. Il computer dovrebbe possedere le seguenti capacità:

- Interpretazione del linguaggio naturale;
- Rappresentazione della conoscenza;
- Ragionamento automatico;
- Apprendimento.

Esiste anche un cosiddetto testi di Turing totale che include un segnale video in modo che l'esaminatore possa verificare le capacità percettive del soggetto, e prevede anche la possibilità di passare oggetti fisici attraverso una finestrella. Per superare il test di Turing totale, il computer necessiterà anche di:

- Visione artificiale;
- Robotica.

1.2 Pensare umanamente: l'approccio della modellazione cognitiva

> [L'automazione delle] attività che associamo al pensiero umano, come il processo decisionale, la risoluzione di problemi, l'apprendimento...
>
> *Bellman, 1978*

Se vogliamo dire che un determinato programma ragiona come un essere umano, dobbiamo prima determinare come pensiamo, entrare dentro i meccanismi interni del cervello umano. Ci sono tre modi per farlo:

- L'introspezione, ovvero il tentativo di catturare "al volo" i nostri pensieri mentre scorrono;
- La sperimentazione psicologica, ovvero, ovvero l'osservazione di una persona in azione;
- L'imaging cerebrale, ovvero l'osservazione del cervello in azione.

Una volta che abbiamo formulato una teoria della mente sufficientemente precisa, diventa possibile esprimere sotto forma di un programma per computer. Se il comportamento del software, per quanto riguarda il suo input/output, corrisponde a quello di una persona, potrebbe essere una prova che alcuni meccanismi del programma operano anche negli esseri umani.

1.3 Pensare razionalmente: l'approccio delle "leggi del pensiero"

> Lo studio delle facoltà mentali attraverso l'uso di modelli computazionali
>
> *Charniak e McDermott, 1985*

Il filosofo Aristotele è stato uno dei primi a cercare di codificare formalmente il "pensiero corretto, ovvero i processi di ragionamento irrefutabili. Il loro studio ha dato origine alla disciplina chiamata logica. I logici del XIX secolo hanno sviluppato una notazione precisa per formulare enunciati riguardanti tutti gli oggetti del mondo e le relazioni tra essi. La tradizione logistica, come viene chiamata all'interno dell'intelligenza artificiale, spera di partire da programmi siffatti per costruire sistemi intelligenti.

1.4 Agire razionalmente: l'approccio degli agenti razionali

> Il ramo della scienza dei calcolatori che si occupa dell'automazione del comportamento intelligente
>
> ---
>
> *Luger-Stubblerield, 1993*

Un agente è semplicemente qualcosa che agisce, che fa qualcosa. Naturalmente tutti i programmi per computer fanno qualcosa, tuttavia si suppone che gli agenti artificiali facciano di più: operare autonomamente, essere in grado di percepire l'ambiente, persistere in un attività per un lungo arco di tempo, creare e perseguire o, in condizione di incertezza, il miglior risultato atteso.

1.5 I fondamenti dell'intelligenza artificiale

- Filosofia (ontologia, epistemologia, filosofia del linguaggio, ...);
- Matematica (logica, algebra, analisi funzionale, probabilità, ...);
- Economia (teoria dei giochi, ...);
- Neuroscienze (brain imaging, ...);
- Psicologia (percezione e cognizione, ...);
- Informatica (algoritmica, complessità, rappresentazione della conoscenza, ...);
- Teoria del controllo e cibernetica;
- Linguistica (linguistica computazionale, semiotica, ...).

Capitolo 2

Agenti intelligenti

2.1 Agenti e ambienti

Un **agente** è qualsiasi cosa possa essere vista come un sistema che percepisce il suo **ambiente** attraverso dei **sensori** e agisce su di esso mediante **attuatori**.

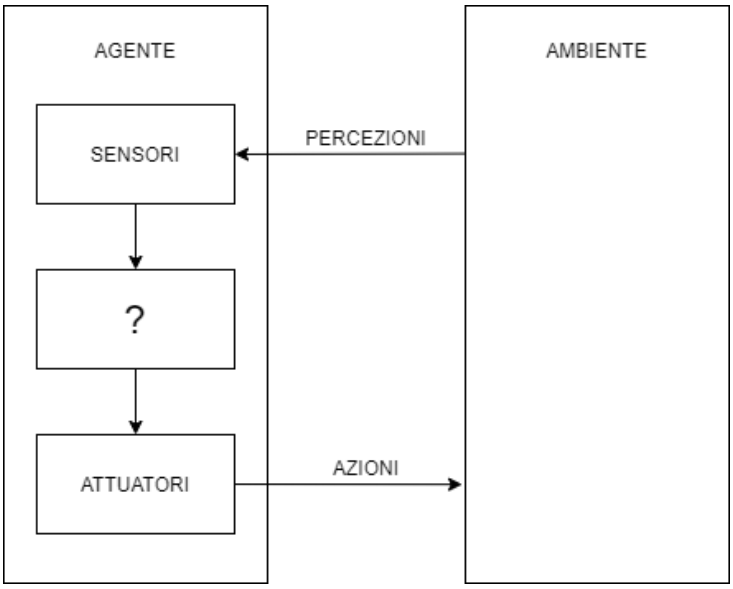

Figura 2.1: Uno schema dell'interazione tra agente ed ambiente

Con **percezione** indichiamo gli input percettivi dell'agente in un dato istante. La **sequenza percettiva** è la storia completa di tutto ciò che l'agente ha percepito nella sua esistenza. In generale, la scelta dell'azione di un agente in un qualsiasi istante può dipendere dall'intera sequenza percettiva osservata fino a quel momento, ma non da qualcosa che non abbia percepito. Il **comportamento** di un agente è descritto dalla funzione agente, che descrive la corrispondenza tra una qualsiasi sequenza percettiva e una specifica azione. La **funzione agente** di un agente artificiale è implementata da un **programma agente**. È importante tenere questi due concetti distinti: la funzione agente è una descrizione matematica astratta; il programma agente è una sua implementazione concreta, in esecuzione all'interno di un sistema fisico.

2.2 Comportarsi correttamente: il concetto di razionalità

Un **agente razionale** è un agente che fa la cosa giusta: dal punto di vista teorico, si potrebbe dire che ogni riga della funzione agente è scritta correttamente. Quando un agente viene inserito in un ambiente, genera una sequenza di azioni in base alle percezioni che riceve. Questa sequenza di azioni porta l'ambiente ad attraversare una sequenza di astati: se tale sequenza è desiderabile, significa che l'agente si è comportato bene. Questa nozione di desiderabilità è catturata da una misura di prestazione che valuta una sequenza di stati dell'ambiente. In un dato momento, ciò che è razionale dipende da quattro fattori:

- La misura di prestazione che definisce il criterio del successo;
- La conoscenza pregressa dell'ambiente da parte dell'agente;
- Le azioni che l'agente può effettuare;
- La sequenza percettiva dell'agente fino all'istante corrente.

Questo porta alla seguente definizione di agente razionale: per ogni possibile sequenza di percezioni, un agente razionale dovrebbe scegliere un'azione che massimizzi il valore atteso della sua misura di precisione, date le informazioni fornite dalla sequenza percettiva e da ogni ulteriore conoscenza dell'agente.

2.3 Onniscenza, apprendimento e autonomia

Dobbiamo distinguere accuratamente il concetto di razionalità e quello di onniscenza. Un agente onnisciente conosce il risultato effettivo delle sue azioni e può agire, ma nella realtà l'onniscenza è impossibile. La nostra definizione di razionalità non richiede l'onniscenza, perché la scelta razionale dipende solo dalla sequenza percettiva fino al momento corrente. Dobbiamo anche assicurarci di non aver inavvertitamente permesso all'agente di intraprendere con risolutezza attività poco intelligenti. Intraprendere azioni mirate a modificare le percezioni future (*information gathering*) è una parte importante della razionalità. Un esempio di information gathering è rappresentato dall'esplorazione che dev'essere intrapresa da un agente posto in un ambiente inizialmente sconosciuto. Un agente razionale non si deve limitare a raccogliere informazioni, ma deve anche essere in grado di apprendere il più possibile sulla base delle proprie percezioni. Quando un agente si appoggia alla conoscenza pregressa inserita dal programmatore invece che alle proprie percezioni, diciamo che manca di autonomia.

2.4 La natura degli ambienti

Gli ambienti sono essenzialmente i "problemi" di cui gli agenti razionali rappresentano le "soluzioni". Descrivere la misura di prestazione, l'ambiente esterno, gli attuatori e i sensori dell'agente possono essere raggruppati nel termine specifico **task environment** o di **descrizione PEAS** (Performance, Environment, Actuators, Sensors).

2.4.1 Proprietà degli ambienti

- *Completamente/parzialmente osservabile*: se i sensori di un agente gli danno accesso allo stato completo dell'ambiente in ogni momento, allora diciamo che l'ambiente operativo è **completamente osservabile**, mentre un ambiente potrebbe essere **parzialmente osservabile** a causa di sensori inaccurati o per la presenza di rumore, o semplicemente perché una parte dei dati non viene rilevata dai sensori;

- *Agente singolo/multiagente*: la distinzione chiave è se, dal punto di vista di un agente A, un oggetto B si può descrivere come il tentativo di massimizzare una misura di prestazione il cui valore dipende dal comportamento dell'agente A (generando quindi un ambiente multiagente **competitivo** o **cooperativo**);

- *Deterministico/stocastico*: se lo stato successivo dell'ambiente è completamente determinato dallo stato corrente e dall'azione eseguita dall'agente, allora si può dire che l'ambiente è **deterministico**, mentre in caso contrario si dice che è **stocastico**; un ambiente è **incerto** se non è completamente osservabile o non è deterministico, mentre un ambiente si dice **non deterministico** se le azioni sono caratterizzate dai loro risultati possibili, ma senza l'associazione alla probabilità;

- *Episodico/sequenziale*: in un ambiente operativo **episodico**, l'esperienza dell'agente è divisa in episodi atomici, e in ogni episodio l'agente riceve una percezione e poi esegue una singola azione, mentre un ambiente si dice **sequenziale** se ogni decisione può influenzare quelle successive;

- *Statico/dinamico*: se l'ambiente può cambiare mentre un agente sta pensando, allora diciamo che è **dinamico** per quell'agente, mentre in caso contrario diciamo che è **statico**; se l'ambiente stesso non cambia al passare del tempo, ma la valutazione della prestazione dell'agente sì, allora diciamo che l'ambiente è **semidinamico**;

- *Discreto/continuo*: la distinzione tra **discreto** e **continuo** si applica allo stato dell'ambiente, al modo in cui è gestito il tempo, alle percezioni e azioni dell'agente;

- *Noto/ignoto*: in un ambiente **noto**, sono conosciuti i risultati (o le corrispondenti probabilità, se l'ambiente è stocastico) per tutte le azioni, mentre, se l'ambiente è **ignoto**, l'agente dovrà apprendere come funzione per poter prendere buone decisioni.

2.5 La struttura degli agenti

Il compito dell'IA è progettare il programma agente che implementa la funzione agente, mentre l'architettura dell'agente comprende i sensori fisici e gli attuatori dell'agente.

2.5.1 Agenti reattivi semplici

Il tipo più semplice è l'**agente reattivo semplice**. Questi agenti scelgono le azioni sulla base della percezione corrente, ignorando tutta la storia percettiva precedente. Gli agenti reattivi semplici hanno l'ammirevole proprietà di essere, appunto, semplici, ma la loro intelligenza è molto limitata, infatti funzionerà solo se si può selezionare la decisione corretta in base alla sola percezione corrente, ovvero solo nel caso in cui l'ambiente sia

completamente osservabile. Evitare cicli infiniti di azioni è possibile quando l'agente è in grado di randomizzare le sue azioni, scegliendole una secondo una componente casuale.

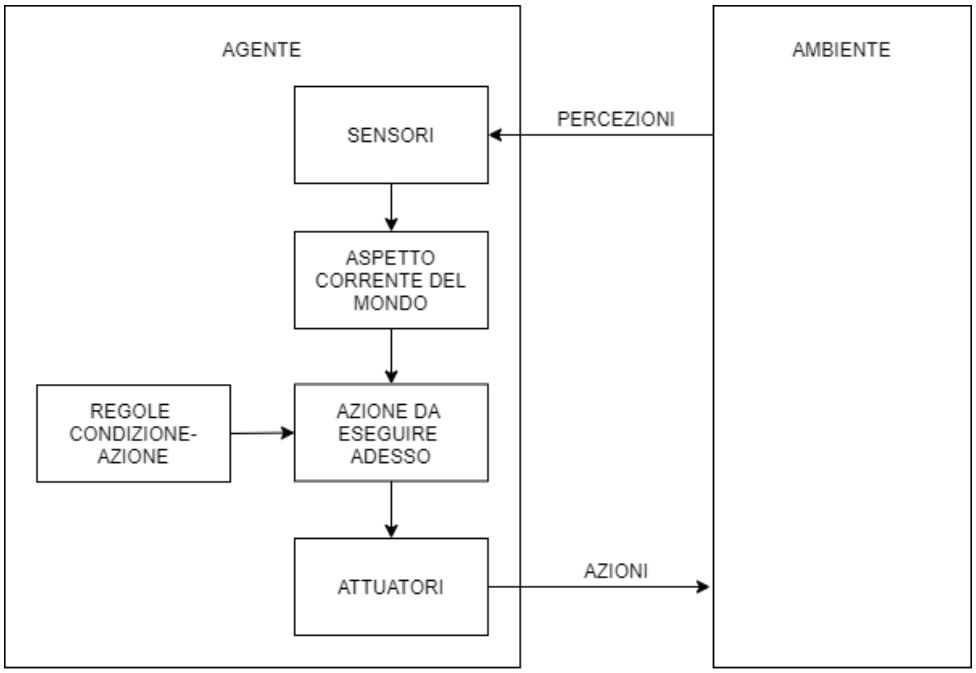

Figura 2.2: Uno schema di un agenti reattivo semplice

2.5.2 Agenti reattivi basati su modello

Il modo più efficace di gestire l'osservabilità parziale, per un agente, è tener traccia della parte del mondo che non può vedere nell'istante corrente. Questo significa che l'agente deve mantenere una sorta di stato interno chi dipende dalla storia delle percezioni e che quindi riflette almeno una parte degli aspetti non osservabili dello stato corrente. Aggiornare le informazioni di stato al passaggio del tempo richiede che il programma agente possieda due tipi di conoscenza. Prima di tutto, deve avere qualche informazione sull'evoluzione del mondo indipendentemente dalle sue azioni. In secondo luogo, sono necessarie delle informazioni sull'effetto che hanno sul mondo le azioni dell'agente stesso. Questa conoscenza sul "funzionamento del mondo", implementata mediante semplici circuiti logici o sviluppata in una teoria scientifica completa, viene chiamata modello del mondo. Un agente che si appoggia a un simile modello prendi il nome, appunto, di **agente basato su modello**.

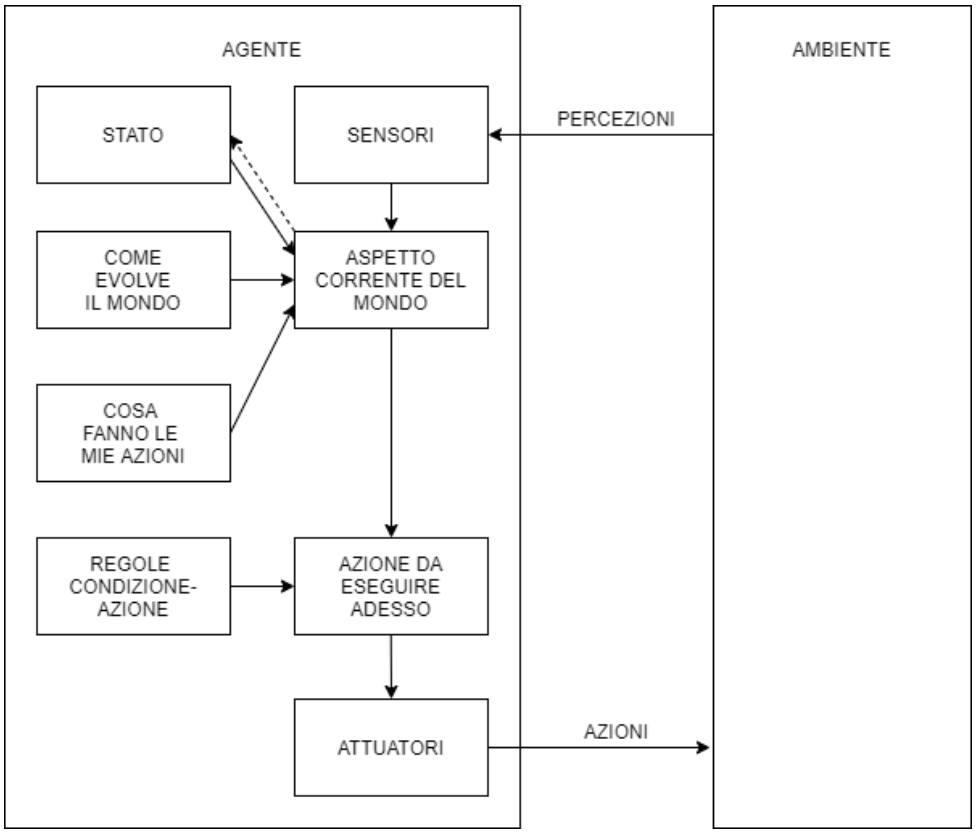

Figura 2.3: Uno schema di un agente basato su modello

2.5.3 Agenti basati su obiettivi

Oltre che della descrizione dello stato corrente l'agente ha bisogno di qualche tipo di informazione riguardante il suo obiettivo aperta (goal), che descriva situazioni desiderabili.

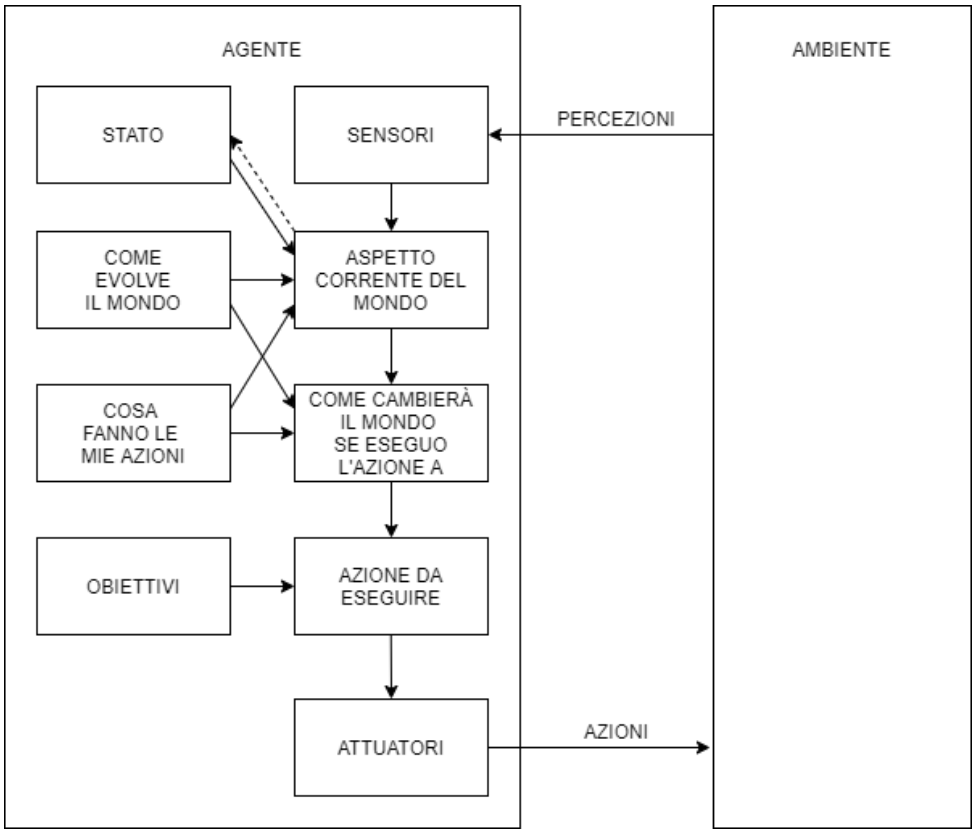

Figura 2.4: Uno schema di un agente basato su obiettivi

2.5.4 Agenti basati sull'utilità

Nella maggior parte degli ambienti gli obiettivi, da soli, non bastano a generare un comportamento di alta qualità. Una funzione di utilità di un agente è, in sostanza, un'internalizzazione della misura di prestazione. In termini tecnici, un **agente** razionale **basato sull'utilità** sceglie l'azione che massimizza l'utilità attesa dei risultati, ovvero l'utilità che l'agente si attende di ottenere, in media, date le probabilità e le utilità di ciascun risultato.

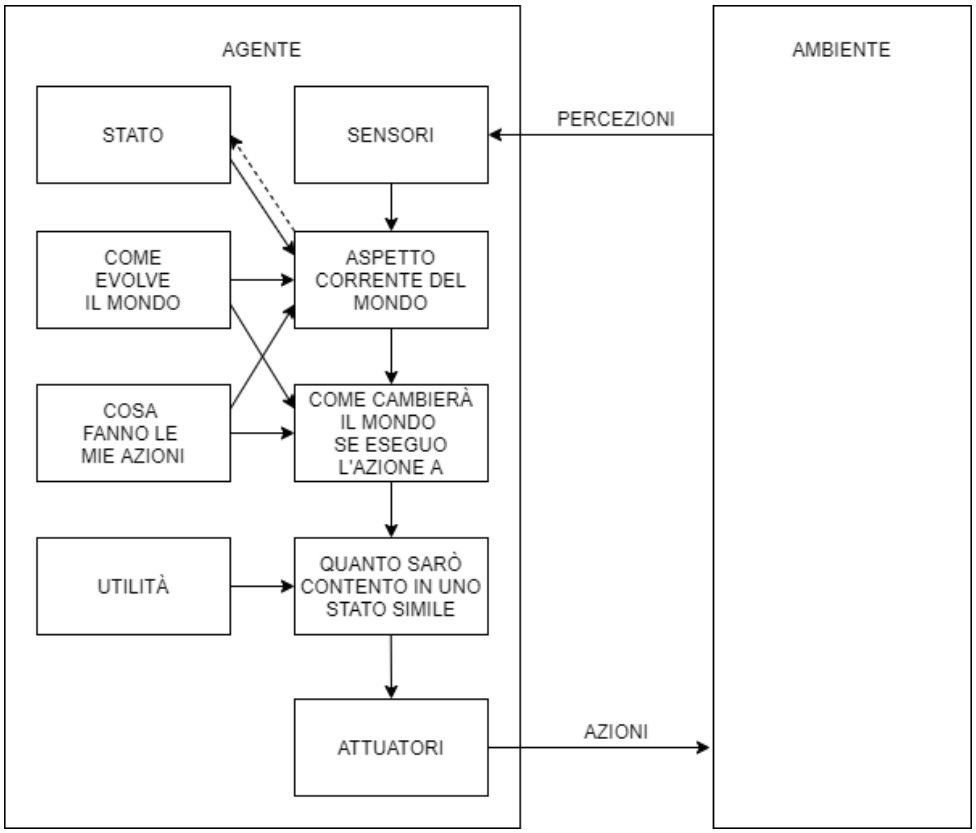

Figura 2.5: Uno schema di un agente basato sull'utilità

2.5.5 Agenti capaci di apprendere

Un **agente capace di apprendere** può essere diviso in quattro componenti astratti. La distinzione più importante è tra l'elemento di apprendimento (learning element), che è responsabile del miglioramento interno, è l'elemento esecutivo (performance element), che si occupa della selezione delle azioni esterne. L'elemento di apprendimento utilizza informazioni provenienti dall'elemento critico riguardo le prestazioni correnti dell'agente e determina se è come modificare l'elemento esecutivo affinché in futuro si comporti meglio. L'ultimo componente di una gente capace di apprendere è il generatore di problemi, il cui scopo è suggerire azioni che portino a esperienze nuove e significative.

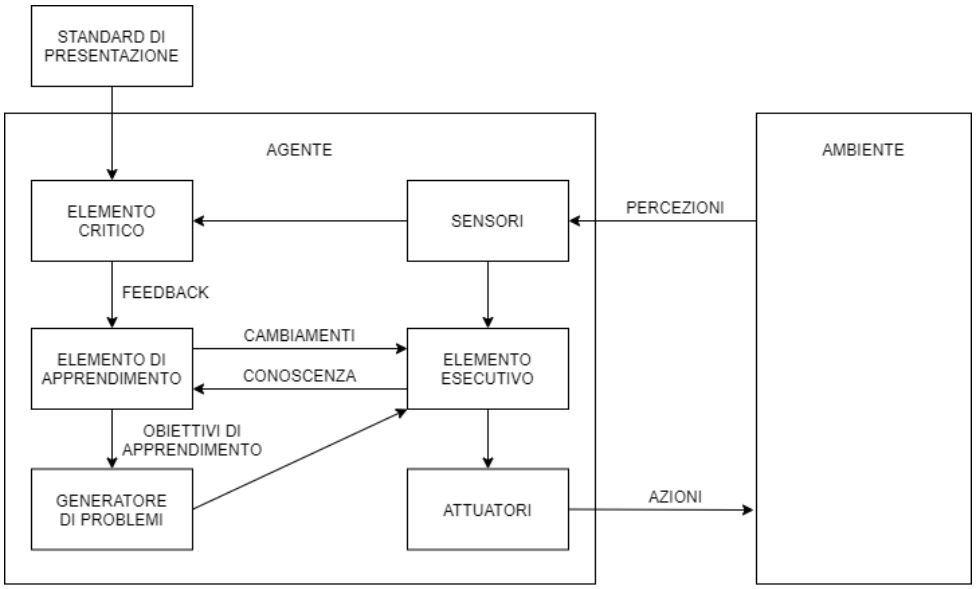

Figura 2.6: Uno schema di un agente capace di apprendere

2.6 Funzionamento dei componenti dei programmi agente

In termini un po' approssimativi, possiamo disporre le rappresentazioni atomica, fattorizzata e strutturata lungo un asse di complessità e potenza espressiva crescente. In una rappresentazione atomica ogni stato del mondo è indivisibile, non ha struttura interna. Una rappresenta fattorizzata suddivide ogni stato in un insieme fissato di variabili o attributi, ognuno dei quali può avere un valore. Una rappresenta strutturata permette di descrivere in modo esplicito gli oggetti insieme alle loro relazioni.

Capitolo 3

Risolvere i problemi con la ricerca

3.1 Agente risolutore di problemi

Un **agente risolutore di problemi** è un agente basato su obiettivi che utilizza una rappresentazione atomica. La formulazione del suo obiettivo, basata sulla situazione corrente e sulla misura di prestazione dell'agente, è il primo passo nella risoluzione di problemi. La formulazione del problema è il processo che porta a decidere, un obiettivo, quali azioni e stati considerare. Un agente che ha a disposizione diverse opzioni immediate di valore sconosciuto può decidere cosa fare esaminando le azioni future che alla fine porteranno a stati di valore conosciuto. Se l'ambiente è osservabile, discreto, noto e deterministico, allora la soluzione di qualsiasi problema è una sequenza fissata di azioni. Il processo che cerca una sequenza di azioni che raggiunge L'obiettivo è detto ricerca. Un algoritmo di ricerca prende un problema come input e restituisce una soluzione sotto forma di una sequenza di azioni. Una volta trovata una soluzione, l'agente può eseguire le azioni raccomandate: questa fase prende il nome di esecuzione.

3.2 Problemi ben definiti e soluzioni

Un problema può essere definito formalmente da cinque componenti:

- Lo stato iniziale in cui si trova la gente;
- Una descrizione delle azioni possibili dell'agente, ognuna delle quali è applicabile in uno stato;
- Una descrizione di ciò che è compiuto da ciascuna azione: Tale descrizione prende il nome di modello di transizione; utilizziamo anche il termine successore per indicare qualsiasi stato raggiungibile da uno Stato mediante una singola azione; insieme, lo stato iniziale, reazioni il modello di transizione definiscono implicitamente lo spazio degli stati del problema; lo spazio degli Stati forma una rete orientata, o grafo;
- Il test obiettivo, determina se un particolare stato è uno stato obiettivo;
- La funzione costo di cammino che assegna un costo numerico ad ogni cammino; il costo di Pap corrisponde al costo di un'azione per passare da uno stato all'altro.

3.3 Cercare soluzioni

Una soluzione è una sequenza di azioni, perciò gli algoritmi di ricerca operano considerando vari e possibili sequenze di azioni. Le possibili sequenze di azioni a partire dallo stato iniziale formano un **albero di ricerca** con lo stato iniziale alla radice; i rami sono azioni e i nodi corrispondono a stati nello spazio degli stati del problema l'idea è quella di espandere lo stato corrente, si applicarvi ogni azione possibile, generare così un nuovo insieme di stati, ciò viene fatto collegando a un nodo padre tutti i suoi nodi figli. Tutti i nodi foglia, ossia i nodi privi di figli che possono essere espansi in un dato punto vengono chiamati **frontiera**. Tutti gli algoritmi di ricerca usano questa struttura di base, ma variano nel modo in cui scelgono il nodo successivo da espandere, nella strategia di ricerca. Uno **stato ripetuto** è uno stato presente almeno due volte nel grafo, viene generalmente generato da un **cammino ciclico**. I cammini ciclici costituiscono un caso speciale del concerto più generale dei **cammini ridondanti**, esistono ogni volta che esistono più modi per passare da uno stato all'altro. Per evitare di addentrarsi in cammini ridondanti è necessario ricordare dove si è passati; a questo scopo, siamo una struttura chiamata *insieme esplorato* (o lista chiusa), che ricorda ogni nodo espanso. Un dettaglio interessante è che la frontiera prepara il grafo degli nella regione esplorata e in quella inesplorata.

3.3.1 Ricerca in ampiezza

La **ricerca in ampiezza** è una semplice strategia nella quale si espande prima il nodo radice, quindi tutti i suoi successori, poi loro successori, e così via. In generale, tutti i nodi a una determinata profondità dell'albero devono essere espansi prima che si possa espandere uno dei nodi al livello successivo. La frontiera è gestita tramite una coda FIFO.

Algorithm 1 WIDTH-SEARCH algorithm

```
1: function WIDTH-SEARCH(G := (V, E), r)
If r it's a goal state return sol(r)
2:     F := {r} FIFO queue;
3:     S := ∅
4:     while F.isEmpty()=false do
5:         u = F.pop()
6:         S = S ∪ {u}
7:         For Each v : (u, v) ∈ E do
8:             if v ∈ V − (F ∪ S) then
If v it's a goal state return sol(v)
Else F.push(v)
9:             end if
10:        end for
11:    end while
12:    return None
13: end function
```

Completezza

Questa ricerca è completa, infatti la ricerca in ampiezza troverà il nodo più vicino alla radice, con altezza d, solo dopo aver espanso tutti i nodi che lo precedono.

Ottimalità

Non sempre, teoricamente la ricerca in ampiezza è ottima solamente se il costo di cammino è una funzione monotona non decrescente della profondità del nodo.

Complessità temporale

$O(b^d)$ (b è il fattore di ramificazione).

Complessità temporale

$O(b^d)$.

3.3.2 Ricerca a costo uniforme

Invece di espandere il nodo meno profondo, la **ricerca a costo uniforme** espande il nodo con il minimo costo di cammino. Questo avviene memorizzando la frontiera come una coda a priorità ordinata secondo il costo di cammino.

Algorithm 2 UNIFORM-COST-SEARCH algorithm

```
1: function UNIFORM-COST-SEARCH(G := (V, E, w), r)
If r it's a goal state return sol(r)
2:     F := {(r, 0)} priority queue;
3:     S := ∅
4:     while F.isEmpty()=false do
5:         u = F.pop()
If u it's a goal state return sol(u)
6:         S ← S ∪ {u}
7:         For Each v : (u, v) ∈ E do
If v ∈ V − (F ∪ S) F.push(v, c(u) + w((u, v)))
If v ∈ F ∧ c(v) > c(u) + w((u, v)) F.update(v, c(u) + w((u, v)))
8:         end for
9:     end while
10:    return None
11: end function
```

Completezza

È garantita purché il costo di ogni passo sia maggiore o uguale a una costante positiva piccola $\epsilon > 0$.

Ottimalità

È ottimale in generale.

Complessità temporale

$O(b^{1+\lfloor C^*/\epsilon \rfloor})$ (C^* è il costo della soluzione ottima).

Complessità temporale

$O(b^{1+\lfloor C^*/\epsilon \rfloor})$.

3.3.3 Ricerca in profondità

La **ricerca in profondità** espande sempre per primo il nodo più profondo nella frontiera corrente dell'albero di ricerca. La ricerca raggiunge immediatamente il livello più profondo dell'albero, pinodj non hanno successori. L'espansione di tali nodi le rimuove dalla frontiera, per cui la ricerca "torna indietro" e considera il nodo più profondo che ha ancora successori non espansi. La ricerca in profondità utilizza una coda lifo, in cui viene scelto per l'espansione l'ultimo nodo generato.

Algorithm 3 DEPTH-SEARCH algorithm

```
 1: function DEPTH-SEARCH(G := (V, E), r)
If r it's a goal state return sol(r)
 2:     F := {r} LIFO stack;
 3:     S := ∅
 4:     while F.isEmpty()=false do
 5:         u = F.pop()
 6:         S ← S ∪ {u}
 7:         For Each v : (u, v) ∈ E do
 8:             if v ∈ V − (F ∪ S) then
If v it's a goal state return sol(v)
Else F.push(v)
 9:             end if
10:         end for
11:     end while
12:     return None
13: end function
```

Completezza

Non è completo.

Ottimalità

Non è ottimale.

Complessità temporale

$O(b^m)$ (m è la profondità massima di un nodo).

Complessità temporale

$O(bm)$.

3.3.4 Ricerca a profondità limitata

La **ricerca a profondità limitata** usa la ricerca in profondità con un limite di profondità predeterminato l. Questo significa che i nodi alla profondità l saranno trattati come se non avessero alcun successore. Questo tipo di ricerca può terminare con due tipi di fallimento: Il valore standard (fallimento) indica che non esiste soluzione, il cosiddetto valore di taglio (taglio) indica che non è stata trovata alcuna soluzione entro il limite di profondità specificato.

Algorithm 4 LIMITED-DEPTH-SEARCH algorithm

```
1: function LIMITED-DEPTH-SEARCH(G := (V, E), r, l)
2:     if r it's a goal state then return sol(r)
3:     else if l = 0 then return Cut
4:     else
5:         happened ← False
6:         For Each v : (r, v) ∈ E do
7:             result ← LIMITED-DEPTH-SEARCH(G, v, l − 1)
 If result = Cut happened ← True
Elif result ≠ Faiulure return result
8:         end for
9:     end if
 If happened = True return Cut
Else return Failure
10: end function
```

Completezza

Non è completo.

Ottimalità

Non è ottimale.

Complessità temporale

$O(b^l)$.

Complessità temporale

$O(b^l)$.

3.3.5 Ricerca ad approfondimento iterativo

La **ricerca ad approfondimento iterativo** è una strategia generale usata spesso in combinazione con la ricerca in profondità per trovare il limite ideale per quest'ultima. Per fare questo il limite viene incrementato progressivamente partendo da 0, finché non viene trovato un nodo obiettivo. Questo avverrà non appena il limite raggiunge d, la profondità del nodo obiettivo più vicino alla radice. In generale, la ricerca ad approfondimento iterativo è il metodo preferito di ricerca non informata quando lo spazio di ricerca è grande e la profondità della soluzione non è nota.

Algorithm 5 ITERATIVE-DEEPENING-SEARCH algorithm

```
1: function ITERATIVE-DEEPENING-SEARCH(G := (V, E), r)
2:     for l = 0 to ∞ do
3:         result ← LIMITED-DEPTH-SEARCH(G, r, l)
If result ≠ Cut return result
4:     end for
5: end function
```

Completezza

Risulta essere completo quando il fattore di ramificazione è finito.

Ottimalità

Risulta essere ottimale quando il costo del cammino è una funzione non decrescente della profondità del nodo.

Complessità temporale

$O(bd)$.

Complessità temporale

$O(b^d)$.

3.3.6 Ricerca bidirezionale

L'idea alla base della **ricerca bidirezionale** ed eseguire due ricerche in parallelo, in avanti la lotta iniziale è un all'indietro partendo dall'obiettivo, speranza che si incontrino a metà strada. La ricerca bidirezionale è implementata sostituendo il test obiettivo con il controllo che le frontiere delle due ricerche si intersechino: In tal caso è stata trovata una soluzione (è importante rendersi conto del fatto che la prima soluzione trovata potrebbe non essere ottima, anche se le due ricerche sono entrambe in ampiezza: è necessario proseguire nella ricerca per assicurarsi che non esista un'altra scorciatoia che colmi il gap). Disegniamo predecessori di uno stato tutti gli stati che lo hanno come successore. La ricerca bidirezionale richiede un metodo per calcolare i predecessori. Quando tutte le azioni dello spazio degli stati sono reversibili, i predecessori di uno stato sono semplicemente i suoi successori. Se lo stato obiettivo è uno solo, la ricerca all'indietro è molto simile a quella in avanti. Se ci sono più stati obiettivo elencati esplicitamente, allora possiamo costruire un nuovo obiettivo fittizio, i cui predecessori immediati sono tutti gli stati obbiettivo reali. Se invece L'obiettivo è una descrizione astratta, allora la ricerca bidirezionale risulta di difficile utilizzo.

Completezza

Risulta essere completo.

Ottimalità

Risulta essere ottimale.

Complessità temporale

$O(b^{d/2})$.

Complessità temporale

$O(b^{d/2})$.

3.4 Confronto tra le strategie di ricerca non informata

RICERCA	COMPLETO?	OTTIMO?	COMPL. TEMPORALE	COMPL. SPAZIALE
In ampiezza	Sì[a]	Sì[c]	$O(b^d)$	$O(b^d)$
A costo uniforme	Sì[a,b]	Sì	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$
In profondità	No	No	$O(b^m)$	$O(bm)$
A profondità limitata	No	No	$O(b^l)$	$O(bl)$
Ad approfondimento iterativo	Sì[a]	Sì[c]	$O(b^d)$	$O(bd)$
Bidirezionale (se applicabile)	Sì[a,d]	Sì[c,d]	$O(b^{d/2})$	$O(b^{d/2})$

Dove:

- b è il fattore di ramificazione;
- d è la profondità della soluzione più vicina alla radice;
- m è la profondità massima dell'albero di ricerca;
- l è il limite alla profondità.

Inoltre:

- (a): completa se b è finito;
- (b): completa se i costi dei passi sono almeno di un ϵ finito;
- (c): ottima se i costi dei passi sono tutti identici;
- (d): se in entrambe le direzioni si esegue una ricerca in ampiezza.

3.5 Strategie di ricerca informata o euristica

Una **strategia di ricerca informata** frutto una conoscenza specifica del problema al di là della tua semplice definizione. La **ricerca best-fit** sceglie il nodo da espandere in base ad una **funzione di valutazione** $f(n)$. La funzione di valutazione è costruita come stima di costo, perciò viene selezionato prima il nodo con la valutazione più bassa. La maggior parte degli algoritmi best-fit include quale componente di f una **funzione euristica** denominata $h(n)$ (costo stimato del cammino più conveniente dallo stato del nodo n a uno stato obiettivo).

3.5.1 Ricerca best-first greedy o "golosa"

La **ricerca best-first greedy** cerca sempre di espandere il nodo più vicino all'obiettivo, sulla base del fatto che è probabile che questo porti rapidamente ad una creazione. Di conseguenza la valutazione dei noti viene fatta applicando direttamente la funzione euristica: $f(n) = h(n)$. La soluzione che trova Però non è detto essere ottima: Ogni passo Cerca sempre di arrivare il più possibile vicino all'obiettivo. È inoltre incompleta persino uno spazio degli stati finito come la ricerca in profondità.

Completezza

Risulta non essere completo.

Ottimalità

Risulta non essere ottimale.

Complessità temporale

$O(b^m)$.

Complessità temporale

$O(b^m)$.

Capitolo 4

Apprendimento tramite esempi

Un agente sta apprendendo se migliora le sue performance su delle task future dopo aver fatto osservazioni riguardo l'ambiente. Qualsiasi componente di un agente può essere migliorato tramite l'apprendimento dai dati. I miglioramenti, e le tecniche usate per effettuarli, dipendono da quattro fattori importanti:

- Quale componente deve essere migliorato;
- Quale conoscenza pregressa l'agente possiede già;
- Quale rappresentazione è usata per i dati e i componenti;
- Quale feedback è disponibile da cui si può apprendere.

I componenti che possono essere appresi sono:

- Una diretta mappatura dalle condizioni dello stato corrente delle azioni;
- Un modo per inferire proprietà rilevanti del mondo tramite la sequenza di percezioni;
- Informazioni riguardo il modo in cui l'ambiente evolve e i risultati delle azioni che l'agente può effettuare;
- Informazioni di utilità che indicano la desiderabilità degli stati dell'ambiente;
- I goal che descrivono le classi di stato dove l'obbiettivo massimizza l'utilità dell'agente.

Apprendere una funzione generale o una regola (probabilmente errata) da specifiche coppie input-output[1] è detto **inductive learning** (**apprendimento induttivo**). Apprendere tramite l'implicazione logica di una nuova regola partendo una regola generale già conosciuta è detto **analytical learning** (**apprendimento analitico**) o **deductive learning** (**apprendimento deduttivo**). Esistono tre tipi di feedback che determinano le tre tipologie principali di apprendimento:

- Nell'**apprendimento non supervisionato** (**unsupervised learning**) l'agente impara dei pattern presenti nell'input seppur non feedback viene fornito. Il task più comune dell'unsupervised learning è quello del **clustering**: trovare, negli input d'esempio forniti, cluster potenzialmente utili.

[1] In questo capitolo gli inputs presentano rappresentazione fratturata (vettore di coppie (*valore*, *attributo*)), e gli outputs possono essere dei valori numerici continui o dei valori discreti.

- Nel **apprendimento per rinforzo** (**reinforcement learning**) l'agente impara tramite una serie di rinforzi-premi o punizioni.
- Nel **apprendimento supervisionato** (**supervised learning**) l'agente osserva alcuni esempi di input-output e impara una funzione che mappa l'output a partire dall'input.

Nella pratica, queste distinzioni non sono sempre così nitide. Nel **semi-supervides learning** ci vengono dati alcuni esempi etichettati e dobbiamo quello che possiamo con una grande collezione di esempi non etichettati. Anche le etichette stesse potrebbero non essere le verità oracolari che speravamo fossero.

4.1 Apprendimento supervisionato

La task dell'apprendimento supervisionato è la seguente: dato un **training set** (**set di addestramento**) di n coppie input-output di esempio

$$(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)^2$$

dove ogni y_i è stato generato da una funzione $y = f(x)$ sconosciuta, trovare una funzione h, detta **ipotesi**, che approssima la funzione f reale. La funzione h è selezionata dallo spazio delle ipotesi $\mathcal{H}$. Per misurare l'accuratezza di un'ipotesi bisogna dargli un **test set** di esempi che è distinto dal training set. Diciamo che un'ipotesi generalizza bene se predice correttamente il valore di y per nuovi esempi. Quando l'output y assume un valore compreso in un set finito di possibili valori, il problema di apprendimento viene detto **classificazione**. Quando y è un numero, il problema di apprendimento è detto **regressione**. Si dice che un'ipotesi è consistente se è in accordo con tutti i dati. Nel caso di poter scegliere tra più ipotesi consistenti, l'idea più semplice è quella di scegliere l'ipotesi consistente con i dati che risulta essere più semplice: questo principio è detto **rasoio di Ockham**. Definire la semplicità, ossia cosa è semplice, non è semplice: infatti, in generale, esiste un tradeoff tra la complessità che si adatta bene ai dati di allenamento e le ipotesi più semplici che potrebbero generalizzare meglio. Diciamo che un problema di apprendimento è realizzabile se lo spazio delle ipotesi contiene la funzione reale. L'apprendimento supervisionato può essere fatto scegliendo l'ipotesi h^* che è più probabile usando i dati iniziali:

$$h^* = \underset{h \in \mathcal{H}}{\operatorname{argmax}} P(h|\text{dati})$$

Secondo la regola di Bayes questo equivale a

$$h^* = \underset{h \in \mathcal{H}}{\operatorname{argmax}} P(\text{dati}|h)P(h)$$

Bisogna notare che esiste un tradeoff tra l'espressività di uno spazio delle ipotesi e la complessità di trovare una buona ipotesi all'interno di questo spazio. Inoltre, esiste un tradeoff tra l'espressività di uno spazio delle ipotesi e il costo temporale di una buona ipotesi all'interno di questo spazio.

[2] I valori x ed y possono avere qualsiasi valore, e pertanto non devono essere per forza dei numeri

4.2 Apprendere i decision tree

I **decision tree** (**alberi di decisione** o **alberi decisionali**) sono una delle forme di machine learning più semplici ma allo stesso tempo più efficienti.

4.2.1 Rappresentazione dei decision tree

Un decision tree rappresenta una funzione che prende come input un vettore di coppie (attributi, valori) e ritorna una "decisione"-un singolo valore di output che può essere discreto o continuo. Una variante è la classificazione booleana, dove ogni input d'esempio viene classificato come vero (un esempio **positivo**) o falso (un esempio **negativo**) tramite un **predicato goal**. Un decision tree raggiunge la sua decisione eseguendo una sequenza di test. Ogni nodo interno nell'albero corrisponde ad un test sul valore di uno degli attributi dell'input, A_i, e i rami che si diramano dal nodo sono etichettati con i possibili valori dell'attributo, $A_i = v_{ik}$. Ogno nodo foglia nell'albero specifica un valore che la funzione deve restituire. Gli esempi sono processati dall'albero partendo dalla radice e seguento le opportune ramificazioni finchè non viene raggiunta una foglia.

4.2.2 Espressività di un decision tree

Un **boolean decision tree** è logicamente equivalente all'asserzione che l'attributo goal è vero se e solo se gli attributi in input soddisfnao uno dei percorsi che portano ad una foglia con il valore true, ossia:

$$Goal \Leftrightarrow (Path_1 \vee Path_2 \vee ...)$$

Dove ogni **Path** è una congiunzione di test attributi-valore richiesti per seguire quel percorso. Per una grande varietà di problemi, il decision tree format produce un risultato piacevole e conciso, Ma alcune funzioni non possono essere rappresentate in modo conciso. I decision tree sono buoni per alcune tipologie di funzioni ma non per altre. Da notare che non esiste una rappresentazione unica che è efficiente per tutte le tipologie di funzioni.

4.2.3 Indurre decision tree tramite esempi

Un esempio per un boolean decision tree consiste in una coppia (x, y), dove x è un vettore di valori per gli attributi di input, e y è un singolo valore booleano di output. Vogliamo un albero che sia consistente con gli esempi e sia il più piccolo possibile. Sfortunatamente, non importa come misuriamo la grandezza, trovare il più piccolo albero consistente è un problema intricato[3]. Tramite semplici euristiche, però, possiamo trovare una buona soluzione approssimata: un piccolo (ma non il più piccolo) albero consistente. L'algoritmo DECISION-TREE-LEARNING adotta una strategia greedy divide-and-conquer: testa sempre l'attributo più importante prima. Questo metodo divide il problema in sotto problemi più piccoli che possono essere risolti ricorsivamente. Con "attributo più importante", si intende l'attributo che crea la maggior differenza nella classificazione degli esempi. In quel modo, speriamo di ottenere la classificazione corretta con un numero piccolo di test, nel senso che tutti i percorsi nell'albero saranno corti e l'albero nel complesso sarà superficiale. Dopo che il test sul primo attributo divide gli esempi, ogni risultato è un nuovo problema di apprendimento di un decision tree a sè stante, con minori esempi ed un attributo in meno. Ci sono quattro casi da considerare per questi problemi ricorsivi:

[3]Questo deriva dal fatto che non esiste una ricerca efficiente tra gli 2^n alberi

- Se gli esempi rimanenti sono tutti positivi (o tutti negativi), possiamo rispondere *true* o *false*;
- Se sono presenti sia esempi positivi che negativi, allora scegli il miglior attributo per dividerli;
- Se non sono rimasti esempi, significa che nessun esempio è stato osservato da questa combinazione di attributi-valori, e viene ritornato il valore calcolato dalla classificazione di pluralità di tutti gli esempi che erano stati usati nel costruire il nodo padre (questi sono passati attraverso la variabile *parent_examples*);
- Se non sono rimasti attributi, ma sono presenti sia esempi positivi che negativi, significa che questi esempi hanno la stessa descrizione, ma differente classificazione, e questo può avvenire nel caso sia presente del **rumore** nei dati.

Algorithm 6 DECISION-TREE-LEARNING algorithm

```
1: function DECISION-TREE-LEARNING(examples, attributes,
   parent_examples)
2:     if examples is empty then
3:         return PLURALITY-VALUE(parent_examples)
4:     else if examples have all the same classification then
5:         return the classification
6:     else if attributes is empty then
7:         return PLURALITY-VALUE(examples)
8:     else
9:         A ← argmax_{a∈attributes} IMPORTANCE(a, examples)
10:        tree ← a new decision tree with root test A
11:        For Each value v_k of A do
12:            exs ← {e : e ∈ examples and e.A = v_k}
13:            subtree ← DECISION-TREE-LEARNING(exs,
   attributes − A, examples)
14:            add a branch to tree with label (A = v_k) and subtree subtree
15:        end forreturn tree
16:    end if
17: end function
```

L'algoritmo DECISION-TREE-LEARNING è mostrato sopra. Il set di esempi è cruciale per costruire l'albero, ma nessuno degli esempi appare nell'albero. Un albero consiste di soli test sugli attributi nei nodi interni, valori degli attributi sui rami, e valori di output sui nodi foglia. Inoltre l'algoritmo di apprendimento osserva gli esempi, piuttosto che la funzione corretta. Bisogna notare che esiste il pericolo di sovra-interpretare l'albero che l'algoritmo sceglie. Quando ci sono varie variabili di simile importanza, la scelta tra di esse è in qualche modo arbitraria. Possiamo valutare l'accuratezza di un algoritmo di apprendimento tramite una **curva di apprendimento**. La curva mostra che, all'aumentare della grandezza del training set, l'accuratezza aumenta (per questa ragione le curve di apprendimento sono anche dette **grafi felici**).

4.2.4 Scelta dei test degli attributi

La ricerca greedy usata nell'algoritmo DECISION-TREE-LEARNING è progettata per minimizzare approssimativamente la profondità dell'albero. L'idea è quella di scegliere l'attributo che permette di avvicinarsi ad una perfetta classificazione degli esempi. Un attributo perfetto divide gli esempi in vari set, ognuno dei quali ha contiene solo esempi positivi o negativi, e perciò ogni set genera un nodo foglia. Pertanto, quello che ci serve è una misura formale di "abbastanza buono" e "veramente inutile" e possiamo implementare la funzione IMPORTANCE. Viene usata la nozione di information gain, la quale è definita in termini di **entropia**, la quantità fondamentale dell'information theory. L'entropia è la misura dell'incertezza di una variabile aleatoria, e viene misurata in bits; l'acquisizione di informazioni corrisponde alla riduzione dell'entropia. In generale, l'entropia di una variabile aleatoria V con valori v_k, ognuna con probabilità $P(v_k)$, è definita come

$$H(V) = \sum_k P(v_k) \log_2 \frac{1}{P(v_k)} = -\sum_k P(v_k) \log_2 P(v_k)$$

Aiuterà definire $B(p)$ come l'entropia di una variabile aleatoria booleana che è vera con probabilità p:

$$B(p) = -(p \log_2 p + (1-p) \log_2 (1-p))$$

Se un training set contiene p esempi positivi ed n esempi negativi, allora l'entropia dell'attributo goal dell'intero set è:

$$B(Goal) = B(\frac{p}{p+n})$$

Un attributo A con d distinti valori divide il training set E in d subset $E_1, E_2, ..., E_d$. Ogni subset E_k ha p_k esempi positivi ed n_k esempi negativi, pertanto se proseguiamo per quel ramo, necessitiamo di ulteriori
$B(p_k/(p_k + n_k))$ bits d'informazione per rispondere alle domande. Un esempio scelto casualmente dal training set possiede il k-esimo valore per l'attributo con probabilità $(p_k+n_k)/(p-n)$, pertanto l'entropia che ci si aspetta rimanga dopo aver testato l'attributo A è

$$Remainder(A) = \sum_{k=1}^{d} \frac{p_k + n_k}{p+n} B(\frac{p_k}{p_k + n_k})$$

L'**information gain** dal test sull'attributo A è la riduzione che ci si aspetta sull'entropia:

$$Gain(A) = B(\frac{p}{p+n}) - Remainder(A)$$

A conti fatti, la funzione $Gain(A)$ è ciò che necessitiamo di dover implementare nella funzione IMPORTANCE.

4.2.5 Generalizzazione e overfitting

Per alcuni problemi l'algoritmo DECISION-TREE-LEARNING genererà un albero molto grosso quando non è presente alcun pattern da trovare: tale problema è detto **overfitting**. Per alcuni decision tree, una tecnica chiamata **decision tree pruning** (**potatura dei deicision tree**) permette di contrastare l'overfitting. La potatura consiste nell'eliminare

i nodi che chiaramente non sono rilevanti. Iniziamo con l'albero completo generato da DECISION-TREE-LEARNING. Prendiamo un nodo test avente solo un nodo foglia come discendente. Se questo test risulta irrilevante[4], allora eliminiamo il test, rimpiazzandolo con un nodo foglia. Ripetiamo questo processo, considerando ogni foglia discendente, finché ogni foglia non viene potata o accettata così com'è. Supponiamo di avere un nodo formato da p esempi positivi ed n esempi negativi. Se l'attributo è irrilevante, ci aspettiamo che divida gli esempi in subset che hanno approssimativamente la stessa proporzione di esempi positivi dell'intero set, $p/(p+n)$, e pertanto l'information gain sarà vicino a zero. Pertanto, l'information gain è un ottimo indizio per l'irrilevanza. Per capire quanto grande deve essere il gain in modo da effettuare la divisione su un certo attributo, possiamo utilizzare un **significance test** statistico. Tale test inizia assumendo che non esiste un pattern sottostante (la cosiddetta **null hypothesis**, o **ipotesi nulla**). A quel punto i dati effettivi vengono analizzati per calcolare l'estensione con la quale deviano dall'assenza perfetta di un pattern. Se il grado di deviazione è statisticamente improbabile (di solito inteso come una probabilità del 5% o meno), allora ciò è considerata una buona prova per la presenza di un pattern significativo nei dati. Le probabilità sono calcolate tramite distribuzioni standard della quantità di deviazione uno si aspetterebbe di vedere in un campionamento casuale. In questo caso, l'ipotesi nulla è che l'attributo sia irrilevante e, pertanto, che l'information gain su un campione incredibilmente grande sarebbe zero. Bisogna calcolare la probabilità che, sotto l'ipotesi nulla, un campione di grandezza $v = n + p$ debba mostrare la deviazione osservata dalla distribuzione prevista di esempi positivi e negativi. Possiamo misurare la deviazione comparando il numero attuale di esempi positivi e negativi in ogni subset, p_k ed n_k, con le quantità previste $\hat{p}_k$ ed $\hat{n}_k$, assumendo una vera irrilevanza:

$$\hat{p}_k = p * \frac{p_k + n_k}{p + n} \qquad \hat{n}_k = n * \frac{p_k + n_k}{p + n}$$

Una misura conveniente della deviazione totale è data da:

$$\Delta = \sum_{k=1}^{d} \frac{(p_k - \hat{p}_k)^2}{\hat{p}_k} + \frac{(n_k - \hat{n}_k)^2}{\hat{n}_k}$$

Sotto l'ipotesi nulla, il valore di Δ è distribuito secondo la distribuzione $\mathcal{X}^2$ ($\mathcal{X}$ *quadrato* o $\mathcal{X}$*-quadro*) con $v - 1$ gradi di libertà. Possiamo usare una tabella $\mathcal{X}^2$ o una routine di una libreria statistica standard per vedere se un Δ particolare conferma o rigetta l'ipotesi nulla (è possibile usare la tabella sotto come riferimento). L'algoritmo DECISION-TREE-LEARNING può implementare questa forma di potatura, conosciuta come $\mathcal{X}^2$ **pruning**. Con la potatura, il rumore nei dati può essere tollerato. Gli alberi potati operano significativamente meglio rispetto agli alberi non potati quando i dati contengono una grande quantità di rumore. Inoltre, gli alberi potati sono di solito molto più piccoli e quindi più facili da capire. $\mathcal{X}^2$ pruning ed information gain possono essere combinati usando un approccio chiamato **early stopping**: l'algoritmo smette di generare nodi quando non ci sono più attributi buoni con cui dividere, piuttosto che prendersi la briga di generare nodi e poi potarli via. Questa variante, però, ci ferma dal riconoscere situazioni in cui non ci sono buoni attributi, ma esiste una combinazione di attributi che fornisce informazioni.

[4] Rileva solo rumore nei dati

Gradi di libertà	Valore $\mathcal{X}^2$					
1	0.004	0.02	0.06	0.15	0.46	1.07
2	0.10	0.21	0.45	0.71	1.39	2.41
3	0.35	0.58	1.01	1.42	2.37	3.66
4	0.71	1.06	1.65	2.20	3.36	4.88
5	1.14	1.61	2.34	3.00	4.35	6.06
6	1.63	2.20	3.07	3.83	5.35	7.23
7	2.17	2.83	3.82	4.67	6.35	8.38
8	2.73	3.49	4.59	5.53	7.34	9.52
9	3.32	4.17	5.38	6.39	8.34	10.66
10	3.94	4.87	6.18	7.27	9.34	11.78
Probabilità	0.95	0.90	0.80	0.70	0.50	0.30

Gradi di libertà	Valore $\mathcal{X}^2$				
1	1.64	2.71	3.84	6.63	10.83
2	3.22	4.61	5.99	9.21	13.82
3	4.64	6.25	7.81	11.34	16.27
4	5.99	7.78	9.49	13.28	18.47
5	7.29	9.24	11.07	15.09	20.52
6	8.56	10.64	12.59	16.81	22.46
7	9.80	12.02	14.07	18.48	24.32
8	11.03	13.36	15.51	20.09	26.12
9	12.24	14.68	16.92	21.67	27.88
10	13.44	15.99	18.31	23.21	29.59
Probabilità	0.20	0.10	0.05	0.01	0.001

Tabella 4.1: Tabella dei valori χ^2 e delle relative probabilità.

4.2.6 Ampliamento dell'applicabilità dei decision tree

In modo da estendere l'induzione di decision tree ad una grande varietà di problemi, è necessario affrontare alcuni problematiche:

- *Dati mancanti*: in alcuni domini, non tutti i valori degli attributi saranno conosciuti per tutti gli esempi (i valori potrebbero non essere stati registrati, oppure sono troppo costosi da registrare);
- *Attributi multivalore*: quando un attributo ha piùvalori possibili, la misura dell'information gain fornisce un'indicazione inappropriata dell'utilità dell'attributo;
- *Attributi di input con valori continui o interi*: gli attributi con valori continui o interi hanno un infinito set di possibili valori, e piuttosto che generare vari rami infiniti, gli algoritmi per l'apprendimento dei decision tree trovano tipicamente lo *split point* che fornisce l'information gain più alto;

- *Attributi di output con valori continui*: Se stiamo cercando di predirre un valore numerico in output, allora necessitiamo di un *regression tree*, le cui foglie possiedono una funzione per ongi subset di valori numerici, piuttosto che di un classification tree, che possiede un solo valore per ogni nodo foglia.

Un sistema di apprendimento di decision tree per problemi del mondo reale deve essere in grado di gestire tutte queste problematiche. Gestire variabili con valori continui è particolarmente importante, perché entrambi i processi fisici e finanziari forniscono dati numerici.

4.3 Valutare e scegliere l'ipotesi migliore

Vogliamo apprendere un'ipotesi che si adatta meglio ai dati futuri. Per renderla precisa dobbiamo definire "dati futuri" e "meglio". Facciamo la *stationary assumption* (*assunzione stazionaria*) che esiste una distribuzione di probabilità su tutti gli esempi che rimane statica nel tempo. Ogni punto dati di esempio è una variabile E_j il cui valore osservato $e_j = (x_j, y_j)$ è ottenuto da quella distribuzione, ed è indipendente degli esempi precedenti:

$$P(E_j|E_{j-1}, E_{j-2}, ...) = P(E_j)$$

Ed ogni esempio ha un'identica distribuzione di probabilità a priori:

$$P(E_j) = P(E_{j-1}) = P(E_{j-2}) = ...$$

Esempi che soddisfano queste assunzioni sono detti indipendenti e identicamente distribuiti o *i.i.d.*. Il passo successivo è quello di definire "la più adatta". Definiamo l'**error rate** di un'ipotesi come il rapporto degli errori che effettua, ossia il rapporto delle volte in cui $h(x) \neq y$ per un esempio (x, y). Ora, solo perché un'ipotesi h ha un basso error rate sul training set non significa che generalizzerà bene. Per ottenere un'accurata valutazione di un'ipotesi, dobbiamo testarla su un set di esempi non ancora usato. L'approccio più semplice è quello di dividere casualmente i dati disponibili in un training set con il quale l'algoritmo di apprendimento produce h ed un test set con il quale è valutata l'accuratezza di h. Questo metodo, chiamato alcune volte **holdout cross-validation**, ha lo svantaggio che fallisce nell'usare tutti i dati disponibili. Se usiamo metà dei dati per il test set, allora effettuiamo il training solo su metà dei dati, e potremmo ottenere un'ipotesi infruttuosa. Dall'altra parte, se usiamo solo il 10% dei dati per il test set, allora potremmo, per caso statistico, ottenere una stima infruttuosa dell'accurateza attuale. Possiamo spremere di più dai dati e ottenere comunque una stima accurata usando una tecnica chiamata **k-fold cross-validation**. L'idea è quella che ogni esempio ha doppio utilizzo, come dato di training e di test. Dividiamo inizialmente i dati in k subset con stessa dimensione. A questo punto effettuiamo k turni di apprendimento; in ogni turno $1/k$ dei dati viene usato come test set e i rimanenti esempi sono usati come dati di training. Il punteggio medio del set di test dei k turni dovrebbe essere una stima migliore rispetto ad un singolo punteggio. Valori comuni per k sono 5 e 10, abbastanza per dare una stima che è statisticamente probabile che sia accurata, al costo che il tempo di computazione è 5 o 10 volte più lungo. Il caso estremo è $k = n$, anche conosciuto come **leave-one-out cross-validation** o **LOOCV**. Nonostante gli sforzi della metodologia statistica, gli utenti invalidano i loro risultati *sbirciando* (*peeking*) inavvertitamente ai dati di test. Il peeking è una conseguenza nell'usare le prestazioni del test set sia per scegliere che per valutare l'ipotesi. Un modo per evitare ciò è quello di tenere da parte il test set, ossia di bloccarlo finché non si è finito

con l'apprendimento e sperare semplicemente di ottenere una valutazione indipendente dell'ipotesi finale. Se il test set è bloccato, ma si vuole lo stesso le prestazione su dati non usati in modo da scegliere una buona ipotesi, allora si può dividere i dati disponibili (escludendo il test set) in un training set e in un **validation set**.

4.3.1 Selezione del modello: complessità contro bontà del modello

La **selezione del modello** definisce lo spazio delle ipotesi e l'**ottimizzazione** trova l'ipotesi migliore all'interno spazio. Un algoritmo che esegue la selezione del modello e l'ottimizzazione è CROSS-VALIDATION-WRAPPER; questa funzione è un **wrapper**[5] che prende un algoritmo di apprendimento `Learner` come parametro (DECISION-TREE-LEARNING, per esempio). Il wrapper enumera i modelli in base ad un parametro, *size*. Per ogni valore di *size*, usa la cross-validation su `Learner` per calcolare l'error rate medio sul training set e test set. La procedura di cross-validation prende il valore di *size* con il valore di errore più basso del validation set. A quel punto viene generata un'ipotesi di quella *size*, utilizzando tutti i dati (senza mantenere nessuno di essi). Alla fine va effettuata una valutazione dell'ipotesi ritornata utilizzando un test set separato. Questo approccio richiede che l'algoritmo di apprendimento accetti un parametro, *size*, e restituisca un'ipotesi di quella *size*. Per l'apprendimento di decision tree, la *size* può essere il numero di nodi. Possiamo modificare DECION-TREE-LEARNING in modo che prenda un numero di nodi come input, crei l'albero in ampiezza piuttosto che in profondità e si fermi quando raggiunge il numero desiderato di nodi.

Algorithm 7 CROSS-VALIDATION-WRAPPER algorithm

```
1: function CROSS-VALIDATION-WRAPPER(Learner, k, examples)
2:     errT, an array, indexed by size, storing training-set error rates
3:     errV, an array, indexed by size, storing validation-set error rates
4:     for size = 1 to ∞ do
5:         errT[size], errV[size]←CROSS-VALIDATION(Learner, size,
   k, examples)
6:         if errT has converged then
7:             best_size←the value of size with minimum errV[size]
8:             return LEARNER(best_size, examples)
9:         end if
10:    end for
11: end function
```

[5]Un wrapper è un programma che ne "riveste" un altro, ossia che funziona da tramite fra i propri clienti (che usano l'interfaccia del wrapper) e il modulo rivestito (che svolge effettivamente i servizi richiesti, su delega dell'oggetto wrapper).

Algorithm 8 CROSS-VALIDATION function

```
1: function CROSS-VALIDATION(Learner, size, k, examples)
2:     fold_errT ← 0
3:     fold_errV ← 0
4:     for fold = 1 to k do
5:         training_set, validation_set ← PARTITION(examples, fold, k)
6:         h ← LEARNER(size, training_set)
7:         fold_errT ← fold_errT+ERROR-RATE(h, training_set)
8:         fold_errV ← fold_errV+ERROR-RATE(h, validation_set)
9:     end for
10:    return fold_errT/k, fold_errV/k
11: end function
```

4.3.2 Da error rate a loss

In machine learning è tradizione esprimere l'utilità tramite una **funzione loss** (**loss function**). La funzione loss $L(x, y, \hat{Y})$ è definita come la quantità di utlità persa prevedendn $h(x) = \hat{y}$ quando la risposta corretta era $f(x) = y$:

$$\begin{aligned} L(x, y, \hat{y}) = &\ Utility(\text{risultato dell'utilizzo di } y \text{ dato un input } x) \\ &- Utility(\text{risultato dell'utilizzo di } \hat{y} \text{ dato un input } x) \end{aligned}$$

Questa è la formulazione più generica della funzione loss. Molto spesso viene usata una versione semplificata, $L(y, \hat{y})$[6], che è indipendente da x. Per funzioni con valori discreti, possiamo enumerare dei valori loss per ogni possibile errore di classificazione, ma non possiamo fare lo stesso per valori reali. In generale sono preferiti errori piccoli piuttosto che errori grandi; due funzioni che implementano quest'idea sono il valore assoluto delle differenze (chiamato loss L_1) e il quadrato delle differenze (detto loss L_2). Se ci accontentiamo con l'idea di minimizzare l'error rate, possiamo usare la funzione loss $L_{0/1}$, che ha un loss di 1 per una risposta sbagliata ed è appropriato per output con valori discreti.

Absolute value loss :	$L_1(y, \hat{y}) = \lvert y - \hat{y} \rvert$
Squared value loss :	$L_2(y, \hat{y}) = (y - \hat{y})^2$
0/1 loss :	$L_{0/1}(y, \hat{y}) = 1$ if $y = \hat{y}$, else 0

L'agente che sta apprendendo può teoricamente massimizzare l'utilità prevista scegliendo l'ipotesi che minimizza il valore di loss previsto su tuttile le coppie input-output che vedrà. Non ha senso parlare di questa previsione senza definire una distribuzione di probabilità $P(X, Y)$ sugli esempi a priori. Sia $\mathcal{E}$ il set di tutti i possibili esempi di input-output. Allora la **generalization loss** per un'ipotesi h (rispetto alla loss function L) è

$$GenLoss_L(h) = \sum_{(x,y)\in\mathcal{E}} L(y, h(x))P(x, y)$$

E l'ipotesi migliore, h^*, è quella con la minor generalization loss prevista:

$$h^* = \operatorname*{argmax}_{h\in\mathcal{H}} GenLoss_L(h)$$

[6]Notare che $L(y, y)$ è sempre zero.

Poiché $P(x, y)$ non è conosciuta, l'agente che sta apprendendo può solo stimare la generalitazion loss con un'**empirical loss** su un set di esempi E:

$$EmpLoss_{L,E}(h) = \frac{1}{N} \sum_{(x,y)\in\mathcal{E}} L(y, h(x))$$

E l'ipotesi migliore stimata $\hat{h}^*$ è quella con la minor empirical loss:

$$\hat{h}^* = \underset{h\in\mathcal{H}}{\operatorname{argmax}}\, EmpLoss_{L,E}(h)$$

Esistono quattro ragioni per cui $\hat{h}^*$ potrebbe differenziare dalla funzione reale f:

- *Irrealizzabilità*: f potrebbe non essere realizzabili, perché potrebbe non essere in $\mathcal{H}$ o perché potrebbe essere presente in un modo che rende altre ipotesi preferite;
- *Varianza*: un algoritmo di apprendimento ritornerà ipotesi differenti per differenti set di esempi, anche se questi sono estratti dalla stessa funzione reale f, e queste ipotesi creeranno differenti predizioni su nuovi esempi;
- *Rumore*: f potrebbe essere non deterministica o *rumorosa*, ossia potrebbe ritornare differenti valori per $f(x)$ ogni volta che x occorre (per definizione, il rumore non può essere previsto);
- *Complessità computazionale*: quando $\mathcal{H}$ è complesso, potrebbe essere computazionalmente intrattabile cercare sistematicamente nell'intero spazio delle ipotesi.

I metodi tradizionali in statistica e i primi anni di machine learning si sono concentrati sullo **small-scale learning**, dove il numero di esempi di training variano dalla dozzina alle poche migliaia; il generalization error derivava principalmente dall'errore di approssimazione dato dal fatto che non conosciamo f nello spazio delle ipotesi, e dall'errore di stima dato dal non avere abbastanza esempi di traning per limitare la varianza. Negli ultimi anni c'è stata molta più enfasi su **large-scale learning**, spesso con milioni di esempi; il generalization error è dominato dai limiti della computazione.

4.3.3 Regularization

Un approccio alternativo è quello ricercare un'ipotesi che minimizza direttamente la somma pesata dell'empirical loss e la complessità dell'ipotesi, che chiameremo il costo totale:

$$Cost(h) = EmpLoss(h) + \lambda\, Complexity(h)$$
$$\hat{h}^* = \underset{h\in\mathcal{H}}{\operatorname{argmin}}\, Cost(h)$$

Dove λ è un parametro, un numero positivo che serve come conversion rate tra la loss e la complessità dell'ipotesi (non sono misurati sulla stessa scala). Questo approccio combina loss e complessità in una sola metrica, permettendoci di trovare l'ipotesi migliore tutto insieme. Sfortunatamente abbiamo ancora bisogno di effettuare una cross-validation search per trovare l'ipotesi che generalizza meglio, ma in questo caso è con valori diversi di λ invece che di *size*. Selezioniamo i valori di λ che ci danno il miglior punteggio del

validation set. Questo progetto di penalizzazione esplicita di ipotesi complesse è chiamato **regularization** (**regolarizzazione**) (poiché cerca una funzione che è più regolare o meno complessa). Notare che la funzione costo ci richiede di fare due scelte: la loss function e misura di complessità, la quale è detta regularization function. La scelta della regularization function dipende dallo spazio delle ipotesi. Un altro modo per semplificare i modelli è quello di ridurre la dimensione con cui il modello lavora. Un processo di **feature selection** può essere effettuato per scartare gli attributi che sembrano irrilevanti. $\mathcal{X}^2$ pruning è un tipo di feature selection. È infatti possibile misurare l'empirical loss e la complessità sulla stessa scala, senza il fattore di conversione λ: possono essere misurati entrambi in bits. L'ipotesi della **minimum description length** (**lunghezza minima di descrizione**, o **MLD**) minimizza il numero totale di bits richiesti. Questo funziona bene per problemi grandi, ma per problemi piccoli si incontrano difficoltà nella scelta della codifica del problema, influenzando il risultato.

4.4 La teoria dell'apprendimento

Computational learning theory (teorema dell'apprendimento computazionale): Ogni ipotesi che è gravemente errata sicuramente verrà "scoperta" con un'alta probabilità dopo un piccolo numero di esempi, dato che fornirà predizioni sbagliate. Pertanto, un'ipotesi che è consistente con un set sufficientemente grande di esempi di training è poco probabile che sia gravemente sbagliata: ovvero, deve essere *probabilmente approssimativamente corretta*.

Qualsiasi algoritmo che ritorna ipotesi che sono probabilmente approssimativamente corrette è detto algoritmo **PAC learning**. I teoremi PAC più semplici hanno a che fare con le funzioni booleane, per le quali la 0/1 loss è appropriata, L'**error rate** di un'ipotesi h, definito informalmente prima, è definito formalmente come il generalization error previsto dagli esempi estratti dalla distribuzione stazionaria:

$$error(h) = GenLoss_{L_{0/1}}(h) = \sum_{x,y} L_{0/1}(y, h(x))P(x, y)$$

In altre parole, $error(h)$ è la probabilità che h classifichi in modo errato un nuovo esempio. Un'ipotesi h è detta **approssimativamente corretta** se $error(h) \leq \epsilon$, dove ϵ è una costante piccola. Possiamo trovare un N tale che, dopo aver osservato N esempi, con una probabilità alta, tutte le ipotesi consistenti sono approssimativamente corrette. Si potrebbe pensare che un'ipotesi approssimativamente corretta è "vicina" alla funzione reale nello spazio delle ipotesi: essa si trova nella cosiddetta **ϵ-ball** intorno alla funzione reale f. Lo spazio delle ipotesi all'esterno della palla è detto $\mathcal{H}_{\text{bad}}$. Possiamo calcolare la probabilità che un'ipotesi "gravemente errata" h_{b} è consistente con i primi N esempi come segue. Sappiamo che $error(h) > \epsilon$, pertanto la probabilità che sia in accordo con un certo esempio è al massimo $1 - \epsilon$. Dato che gli esempi sono indipendenti, il limite per N esempi è

$$P(h_{\text{b}} \text{ è in accordo con } N \text{ esempi}) \leq (1 - \epsilon)^N$$

La probabilità che $\mathcal{H}_{\text{bad}}$ contiene almeno un'ipotesi consistente è legata alla somma delle probabilità individuali:

$$P(\mathcal{H}_{\text{bad}} \text{ contiene un'ipotesi consistente}) \leq |\mathcal{H}_{\text{bad}}|(1-\epsilon)^N \leq |\mathcal{H}|(1-\epsilon)^N$$

Dove abbiamo usato il fatto che $|\mathcal{H}_{\text{bad}}| \leq |\mathcal{H}|$. Vogliamo rendere la probabilità di questo evento minore di un valore piccolo δ:

$$|\mathcal{H}|(1-\epsilon)^N \leq \delta$$

Dato che $1-\epsilon \leq e^{-\epsilon}$, possiamo ottenere ciò se permettiamo all'algoritmo di osservare N esempi, dove:

$$N \geq \frac{1}{\epsilon}(\ln \frac{1}{\delta} + ln|\mathcal{H}|)$$

Pertanto, se l'algoritmo di apprendimento ritorna un'ipotesi che è consistente con tutti questi esempi, allora avrà un errore al massimo ϵ con almeno probabilità $1-\delta$. In altre parole, è probabilmente approssimativamente corretta. Il numero di esempi richiesti, in funzione di ϵ e δ, è chiamato **sample complexity** dello spazio delle ipotesi. Per ottenere una generalizzazione reale degli esempi non visti, necessitiamo di restringere lo spazio delle ipotesi $\mathcal{H}$ in qualche modo; ovviamente, se restringiamo lo spazio, rischiamo di eliminare anche la funzione reale. Ci sono tre modi per uscire da questo dilemma:

- Il primo è quello di indurre una conoscenza preliminare per gestire il problema;
- Il secondo è che l'algoritmo non ritorni una qualsiasi ipotesi consistente, bensì preferibilmente quella più semplice;
- Il terzo è di concentrarsi su un subset apprendibile dell'intero spazio delle ipotesi.

4.4.1 Esempio di PAC learning: apprendere decision lists

Le **decision lists** (**liste decisionali**) consistono in una serie di test, ognuno dei quali è una congiunzione di letterali. Se un test ha successo quando viene applicato sulla descrizione di un esempio, , la decision list specifica il valore da restituire. Se il test fallisce, il processo continua con il prossimo test nella lista. le decision list assomigliano ai decision tree, ma la loro struttura globale è molto più semplice: si ramificano solo in una direzione. In contrasto, i singoli test sono molto più complessi. Se permettiamo ai test di avere una grandezza arbitraria, allora le decision list possono rappresentare qualsiasi funzione booleana. Dall'altra parte, se restringiamo la grandezza di ogni test al massimo a k letterali, allora l'algoritmo di apprendimento può generalizzare con successo tramite un piccolo numero di esempi. Chiamiamo questo linguaggio k**-DL**. k-DL include come subset il linguaggio k**-DT**, il set di tutti i decision tree con profondità al massimo k. È importante ricordarsi che il particolare linguaggio indicato con k-dl dipende degli attributi usato per descrivere gli esempi. Viene usata la notazione k-DL(n) per indicare un linguaggio k-DL che usa n attributi booleani. k-DL è apprendibile: ossia, ogni funzione in k-DL può essere approssimata accuratamente dopo il training su un numero ragionevole di esempi. Dobbiamo calcolare il numero di ipotesi nel linguaggio. Per dimostrarlo, dobbiamo calcolare il numero di ipotesi presenti nel linguaggio. Sia $Conj(n,k)$ il linguaggio dei test, ossia la congiunzione di al massimo k letterali usando n attributi. Dato che una decision list è composta da test, e siccome ogni test può essere attaccato ad un risultato Yes ($True$) o No ($False$) o può essere assente dalla decision list, esistono al massimo $3^{|Conj(n,k)|}$ set distinte di test sui componenti. Ognuno di questi di test può avere qualsiasi ordine, perciò:

$$|k\text{-DL}| \leq 3^{|Conj(n,k)|}|Conj(n,k)|!$$

Il numero di congiunzioni di k letterali da n attributi è dato da

$$|Conj(n,k)| = \sum_{i=0}^{k} \binom{2n}{i} = O(n^k)$$

Possiamo mostrare che il numero di esempi necessari per PAC-apprendere una funzione k-DL è polinomiale in n:

$$N \geq \frac{1}{\epsilon}(\ln \frac{1}{\delta} + O(n^k \log_2(n^k)))$$

Pertanto, un algoritmo che ritorna una decision list consistente PAC-apprenderà una funzione k-DL tramite un numero ragionevole di esempi, per valori picclo di k. Una algoritmo efficiente che ritorna una decision list consistente è l'algoritmo greedy DECISION-LIST-LEARNING che ripetutamente trova un test che è in accordo esattamente cone alcuni subset del training set. Una volta che tale test viene trovato, viene aggiunto alla decision list in costruzione e rimuove gli esempi corrispettivi. Questo viene ripetuto finché non vengono esauriti gli esempi. Questo algoritmo non specifica il metodo per la selezione il test successivo da aggiungere alla decision list. Sembra ragionevole preferire test piccoli che sono in accordo con grandi set di esempi uniformemente classificati, in modo che la decisione complessiva sia la più piccola possibile.

Algorithm 9 DECISION-LIST-LEARNING function

```
1: function DECISION-LIST-LEARNING(examples)
If examples is empty the trivial decision list No
2:     t ← a test that matches a nonempty subset examples_t of examples such that the
   members of examples_t are all positive or all negative
If there is no such t return failure
If the examples in examples_t are positive o ← True
Else o ← False
3:     return a decision list with initial test t and outcome o and remainimg tests given
   by DECISION-LIST-LEARNING(examples − examples_t)
4: end function
```

4.5 Regressione e classificazione con modelli lineari

4.5.1 Regressione lineare univariata

Una **funzione lineare univariata** (una linea dritta) con input x e output y ha la forma $y = w_1x + w_0$, dove w_0 e w_1 sono coefficienti a valori reali da apprendere. Usiamo le lettere w dato che possiamo vedere i coefficienti come **pesi** (**weights**). Definiamo w come il vettore $[w_0, w_1]$, e definiamo

$$h_{\mathrm{w}}(x) = w_1x + w_0$$

Il compito di trovare la funzione h_{w} che si adatta meglio ai dati è detto **regressione lineare** (**linear regression**). Per datattarsi a tutti i dati, tutto quello che dobbiamo fare è trovare i pesi $[w_0, w_1]$ che minimizzano l'empirical loss. È tradizione usare la squared loss function, L_2, sommata su tutti gli esempi di training:

$$Loss(h_{\mathrm{w}}) = \sum_{j=1}^{N} L_2(y_j, h_{\mathrm{w}(x_j)}) = \sum_{j=1}^{N} (y_j - h_{\mathrm{w}}(x_j))^2 = \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2$$

Vogliamo trovare $\mathrm{w} = \mathrm{argmin}_{\mathrm{w}}\ Loss(h_{\mathrm{w}})$. La somma $\sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2$ è minimizzata quando le derivate parziali rispetta $w?0$ e $w?1$ sono zero:

$$\frac{\delta}{\delta w_0} \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2 = 0 \text{ e } \frac{\delta}{\delta w_1} \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2 = 0$$

Queste equazioni hanno un'unica soluzione:

$$w_1 = \frac{N(\sum x_j y_j) - (\sum x_j)(\sum y_j)}{N(\sum {x_j}^2) - (\sum x_j)^2} \text{ e } w_0 = \frac{(\sum y_j - w_1(\sum x_j))}{N}$$

Definiamo come **weight space** (**spazio dei pesi**) lo spazio di tutti i valori che i pesi possono assumere. Per andare oltre i modelli lineari, dobbiamo affrontare il fatto che molto spesso le equazioni che definiscono la minimum loss non hanno nessuna soluzione in formula chiusa, e ci troviamo ad affrontare invece un problema di ricerca di ottimizzazione generale in un weight space continuo. Tale problema può essere risolto tramite l'algoritmo hill climbing che segue il **gradiente** della funzione per essere ottimizzata. In questo caso, dato che stiamo cercando di minimizzare la loss, useremo la **discesa del gradiente**. Scegliamo un qualsiasi punto di partenza nel weight space (in questo caso un punto (w_0, w_1) nel piano), per poi spostarsi in un punto vicino che si trova su una discesa, ripetendo finché non convergiamo alla minima loss possibile:

```
w ← any point in the parameter space
loop until convergence do
    For Each w_i in w do
        w_i ← w_i − α δ/δw_i Loss(w)
    end for
end loop
```

Il parametro α, chiamato **step size**, è di solito definito come **learning rate** quando stiamo minimizzando la loss in un problema di apprendimento. Può essere un valore fissato, o può decadere nel tempo mentre il processo di apprendimento continua. Per la regressione univariata, la loss function è una funzione quadratica, pertanto le derivate parziali saranno delle funzioni lineari. Lavoriamo prima di tutto sulle derivate parziali (i versanti) nel caso semplificato di un solo esempio di traning (x, y):

$$\begin{aligned}\frac{\delta}{\delta w_i} Loss(\mathrm{w}) &= \frac{\delta}{\delta w_i}(y - h_{\mathrm{w}}(x))^2 = \\ &= 2(y - h_{\mathrm{w}}(x)) * \frac{\delta}{\delta w_i}(y - h_{\mathrm{w}}(x)) = \\ &= 2(y - h_{\mathrm{w}}(x)) * \frac{\delta}{\delta w_i}(y - (w_1 x + w_0))\end{aligned}$$

Applicando questa formula a w_0 e w_1 otteniamo:

$$\frac{\delta}{\delta w_0} Loss(\mathrm{w}) = -2(y - h_{\mathrm{w}}(x)) \qquad \frac{\delta}{\delta w_1} Loss(\mathrm{w}) = -2(y - h_{\mathrm{w}}(x)) * x$$
$$w_0 \leftarrow w_0 + \alpha(y - h_{\mathrm{w}}(x)) \qquad w_1 \leftarrow w_1 + \alpha(y - h_{\mathrm{w}}(x)) * x$$

Questi risultati hanno senso: se $h_{\mathrm{w}}(x) > y$, l'output dell'ipotesi è troppo grande, e pertanto w_0 va ridotto, mentre w_1 viene ridotto se x ha valore positivo o incrementato se ha valore negativo. Per N esempi di training, vogliamo minimizzare la somma delle loss individuali per ogni esempio. La derivata di una somma è la somma delle derivate, pertanto abbiamo:

$$w_0 \leftarrow w_0 + \alpha \sum_j (y_j - h_{\mathrm{w}}(x_j)) \qquad w_1 \leftarrow w_1 + \alpha \sum_j (y_j - h_{\mathrm{w}}(x_j)) * x_j$$

Questi aggiornamenti costituiscono regola di apprendimento della **discesa del gradiente in batch** (**batch gradient descend**) per la regressione lineare univariata. Esiste un'altra possibilità, detta **discesa stocastica del gradiente** (**stochastic gradinet descend**), dove consideriamo solo un singolo punto di training alla volta, eseguendo un passaggio dopo l'altro. La discesa stocastica del gradiente può essere usata in un ambiente online, dove nuovi dati vengono ricevuti uno alla volta, oppure offline, dove si cicla sui vari dati tante volte quanto necessario, effettuando i passaggi uno dopo l'altro considerando ogni singolo esempio. È molto più veloce della discesa del gradiente in batch. Con un learning rate fissato α, però, non garantisce la convergenza.

4.5.2 Regressione lineare multivariata

Possiamo estenderci facilmente ai problemi di **regressione lineare multivariata**, in cui ogni esempio x_j è un vettore di n elementi. Il nostro spazio delle ipotesi è il set di funzioni nella forma

$$h_{sw}(\mathrm{x}_j) = w_0 + w_1 x_{j1} + w_2 x_{j2} + ... + w_n x_{jn} = w_0 + \sum_i w_i x_{ji}$$

Il termine w_0, l'intercetta, risulta diverso rispetto agli altri. Possiamo risolvere questa cosa inserendo un valore di input dummy $x_j 0$, che viene definito come sempre uguale a 1. A questo punto h è semplicemente il prodotto scalare dei pesi e il vettore di input:

$$h_{sw}(\mathrm{x}_j) = \mathrm{w} \cdot \mathrm{x}_j = \mathrm{w}^{\mathrm{T}} \mathrm{x}_j = \sum_i w_i x_{ji}$$

Il miglio vettore di pesi, w^*, minimizza l'error loss quadratica sugli esempi:

$$\mathrm{w}^* = \underset{\mathrm{w}}{\operatorname{argmin}} \sum_j L_2(y_j, \mathrm{w} \cdot \mathrm{x}_j)$$

La discesa del gradiente raggiungerà il minimo (unico) della loss function; l'equazione di aggiornamento per ogni peso w_i è

$$w_j \leftarrow w_j + \alpha \sum_j x_{ji}(y_j - h_{\mathrm{w}}(\mathrm{x}_j))$$

È possibile risolvere analiticamente per la w che minimizza la lasso, Sia y il vettore degli output degli esempi di training, e X la **matrice dei dati** (la matrice degli input con un esempio di n valori per riga). Allora la soluzione che minimizza l'errore quadratico è

$$w^* = (\mathrm{X}^\mathrm{T}\mathrm{X})^{-1}\mathrm{X}^\mathrm{T}\mathrm{y}$$

Con la regressione lineare univariata non dobbiamo preoccuparci dell'overfitting, ma con la regressione lineare multivariata in spazi ad alta dimensione è possibile che alcune dimensioni che sono irrilevanti risultano per qualche motivo utili, portando ad un **overfitting**. Pertanto, è comune usare una **regolarizzazione** sulle funzioni lineari multivariate per evitare l'overfitting. Per le funzioni lieari la complessità può essere specificata in funzione dei pesi. Possiamo considerare una famiglia di funzioni di regolarizzazione:

$$Cost(h_\mathrm{w}) = EmpLoss(h_\mathrm{w}) + \lambda\, Complexity(h_\mathrm{w})$$
$$Complexity(h_\mathrm{w}) = L_q(\mathrm{w}) = \sum_i |w_i|^q$$

Come la funzione loss, con $q = 1$ abbiamo l'L_1 regularization, che minimizza la somma dei valori assoluti; con $q = 2$ l'L_2 regularization minimizza la somma dei quadrati. La L_1 regularization ha un vantaggio importante: tende a produrre un **modello sparso**, che setta molti pesi a zero, dichiarando effettivamente i rispettivi attributi come irrilevanti. Le ipotesi che scartano attributi sono più facili da capire per un essere umano, ed è meno probabile che generino un overfit. Minimizzare $Loss(\mathrm{w}) + \lambda\, Complexity(\mathrm{w})$ equivale a minimizzare $Loss(\mathrm{w})$ soggetta al vincolo che $Complexity(\mathrm{w}) \leq c$, per una certa costante c che è legata a λ. Se consideriamo l'L_1 complexity, per una posizione arbitraria del minimo, sarà normale che l'angolo della scatola si trovi più vicino al minimo, poiché i vertici sono punti che hanno zero come valore di una delle dimensioni. Se consideriamo l'L_2 complexity, in generale, non c'è motivo per cui l'intersezione appaia su uno degli assi; pertanto la L_2 regularization non tende a produrre pesi uguali a zero. Pertanto, l'L_1 regularization prende gli assi dimensionali seriamente, mentre l'L_2 li tratta come arbitrari.

4.5.3 Classificatori lineari con un hard threshold

Le funzioni lineari possono essere usate per la classificazione oltre che la regressione. Un **decision boundary** è una linea (o una superficie, in caso di più dimensioni) che separa le due o più classi. Un lienar decision boundary è detto **linear separator** (**separatore lineare**) e i dati che ammettono tale separatore sono detti **linearmente separabili**. Usando la convenzione di un input dummy $x_0 = 1$, possiamo scrivere l'ipotesi di classificazione come

$$h_\mathrm{w}(\mathrm{x}) = 1 \text{ se } \mathrm{w} \cdot \mathrm{x} \geq 0 \text{ e } 0 \text{ altrimenti}$$

Alternativamente, possiamo pensare ad h come il risultato del passaggio della funzione lineare $\mathrm{w} \cdot \mathrm{x}$ tramite una **threshold function** (**funzione di soglia**):

$$h_\mathrm{w}(\mathrm{x}) = Threshold(\mathrm{w} \cdot \mathrm{x}) \text{ dove } Threshold(z) = 1 \text{ se } z \geq 0 \text{ e } 0 \text{ altrimenti}$$

La regola di aggiornamento per la regressione lineare è detta **perceptive leraning route**. Tipicamente la regola di apprendimento è applicata un esempio alla volta, scegliendo esempi in modo casuale (discesa del gradiente stocastico). La **curva di apprendimento** misura la performance del classificatore su di un set di training fissato mentre il processo

procede sullo stesso training set. Il processo di "convergenza" non è esattamente carino, ma funziona sempre, e tipicamente le variazioni tra le varie esecuzioni è molto ampia. È molto comune nel mondo reale che la perceptron linear rule non converge ad un separatore lineare perfetto. In generale, la perceptron learning rule potrebbe non convergere ad una soluzione stabile per un learning rate fisso α, ma se α decade come $O(\frac{1}{t})$ dove t è il numero di tirazioni, allora si può mostrare che la regola converge ad una soluzione con errore minimo quando gli esempi sono presentati in una sequenza casuale.

4.5.4 Classificazione lineare con la regressione logistica

Passare l'uotput di una funzione lienare ad una threshold function crea un classificatore lineare, eppure la forte natura del threshold causa alcuni problemi: l'ipotesi $h_{\mathrm{w}}(\mathrm{x})$ non è differenziabile ed è di fatto una funzione discontinua dei suoi input e dei suoi pesi (questo rende l'apprendimento con la perceptron rule un'avventura molto imprevedibile), inoltre il classificatore lineare annuncia sempre una previsione completamente sicura di 1 o 0 (in molte situazioni necessitiamo di previsioni più graduali). Tutti questi problemi possono essere risolti in larga misura ammorbidendo la threshold function, ossia approssimando l'hard threshold function con una funziona continua e differenziabile. La funzione logistica

$$Logistic(z) = \frac{1}{1 + e^{-z}}$$

ha più proprietà matematiche convenienti. Con la funzione logistica che rimpiazza la threshold function, ora abbiamo

$$h_{\mathrm{w}}(\mathrm{x}) = Logistic(\mathrm{w} \cdot \mathrm{x}) = \frac{1}{1 + e^{-\mathrm{w} \cdot \mathrm{x}}}$$

L'ipotesi forma un soft boundary sullo spazio d'input e fornisce una probabilità di 0.5 per ogni input al centro della regione del confine, e raggiunge 0 ed 1 se ci muoviamo lontano dal confine. Il processo di adattamento dei pesi di questo modello per minimizzare la loss di un set di dati è detto **regressione logistica**. Non esiste alcuna soluzione in forma chiusa per trovare il valore ottimo di w con questo modello, ma la computazione della discesa del gradiente è semplice. Dato che la nostra ipotesi non dà più in output solo 0 o 1, useremo la L_2 loss function. Per un singolo esempio (x, y), la derivata di un gradiente è la stessa della regressione lienare, fino al punto in cui l'attuale forma di h è inserita.

$$\begin{aligned}
\frac{\delta}{\delta w_i} Loss(\mathrm{w}) &= \frac{\delta}{\delta w_i}(y - h_{\mathrm{w}}(\mathrm{x}))^2 = \\
&= 2(y - h_{\mathrm{w}}(\mathrm{x})) \cdot \frac{\delta}{\delta w_i}(y - h_{\mathrm{w}}(\mathrm{x})) = \\
&= -2(y - h_{\mathrm{w}}(\mathrm{x})) \cdot g'(\mathrm{w} \cdot \mathrm{x}) \cdot \frac{\delta}{\delta w_i} \mathrm{w} \cdot \mathrm{x} = \\
&= -2(y - h_{\mathrm{w}}(\mathrm{x})) \cdot g'(\mathrm{w} \cdot \mathrm{x}) \cdot x_i
\end{aligned}$$

La derivata g' della funzione logistica soddisfa $g'(z) = g(z)(1 - g(z))$, pertanto abbiamo

$$g'(\mathrm{w} \cdot \mathrm{x}) = g(\mathrm{w} \cdot \mathrm{x})(1 - g(\mathrm{w} \cdot \mathrm{x})) = h_{\mathrm{w}}(\mathrm{x})(1 - h_{\mathrm{w}}(\mathrm{x}))$$

Pertanto l'aggiornamento dei pesi per la minimizzazione della loss è

$$w_i \leftarrow w_i + \alpha(y - h_{\mathrm{w}}(\mathrm{x})) \cdot h_{\mathrm{w}}(\mathrm{x})(1 - h_{\mathrm{w}}(\mathrm{x})) \cdot x_i$$

In un caso dove i dati sono linearmente separabili, la regressione logistica è alquanto lenta a convergere, ma si comporta in modo molto più prevedibile. Nel caso in cui i dati siano rumorosi e non separabili, la regressione logistica converge molto più velocemente e in modo affidabile. Questi vantaggi vengono riportati nelle applicazioni del mondo reale e la regressione logistica è diventata una delle tecniche di classificazione più popolari per i problemi di medicina, marketing e analisi dei sondaggi, punteggio di credito[7], saluta pubblica e altre applicazioni.

4.6 Reti neurali artificiali

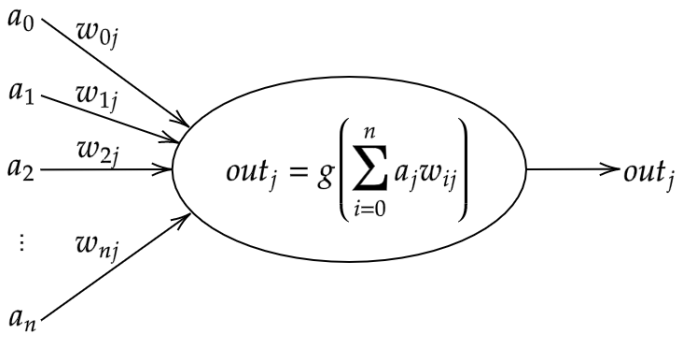

Figura 4.1: Un modello matematico semplice di un neurone. L'attivazione dell'uscita dell'unità è $out_j = g(\sum_{i=0}^{n} a_i w_{ij})$, dove a_i è l'attivatore e $w_i j$ è il peso del collegamento tra a_i e il neurone.

L'attività mentale consiste principalmente nell'attività elettrochimica nelle reti di cellule cerebrali chiamate neuroni. Le **reti neurali** (**neural networks**), grosso modo, "sparano"quando una combinazione lineare dei suoi input eccede qualche (hard o soft) threshold. Per essere precisi, una rete neurale è una collezione di unità connesse tra loro; le proprietà della rete sono determinate dalla sua topologia e della proprietà dei suoi "neuroni".

4.6.1 Struttura delle reti neurali

Le reti neurali sono composte da nodi o **unità** connesse da **link** diretti. Un link da un'unità i all'unità j serve a propagare l'**attivatore**. Ogni link possiede inoltre un **peso** numerico w_{ij} associato ad esso, che determina la forza e il segno della connessione. Come un modello di una regressione lineare, ogni unità possiede un input dummy $a_0 = 1$ con un peso associato w_{0j}. Ogni unità j esegue prima una somma pesata degli input, per poi applicare la **funzione di attivazione** g su questa somma per ottenere l'output:

$$out_j = g(in_j) = g(\sum_{i=0}^{n} a_i w_{ij})$$

[7]Un punteggio di credito è un'espressione numerica basata su un'analisi di livello dei file di credito di una persona, per rappresentare l'affidabilità creditizia di un individuo.

La funzione di attivazione g è tipicamente un hard threshold, in tal caso l'unità è detta **perceptron**, o una funzione logistica, in tal caso viene spesso usato il termine **sigmoid perceptron**. Entrambe le funzioni di attivazione non lienari assicurano l'importante proprietà che l'intera rete di unità può rappresentare una funzione non lineare. Una **rete feed-forward** (**feed-forward network**) ha connessioni tutte dirette in una sola direzione, formando un grafo diretto aciclico; rappresenta pertanto una funzione basata sul solo input, e pertanto non possiede alcuno stato interno se non i pesi stessi. Una **rete ricorrente** (**recurrent network**), d'altra parte, usa l'output prodotto come input all'interno della rete; questo significa che il livello di attivazione della rete forma un sistema dinamico che può raggiungere uno stato stabile o mostrare oscillazioni o anche presentare un comportamento caotico. Inoltre, la risposta della rete dato un certo input dipende dal suo stato iniziale, che può dipendere del precedente input, pertanto le reti ricorrenti possono supportare memoria a breve termine.Le reti feed-forward sono di solito organizzate su **strati** (**layers**), e una rete a più strati possiede uno o più strati di **unità nascoste** (**hidden units**) che non sono connesse all'output della rete.

4.6.2 Reti neurali feed-forward a singolo strato (perceptrons)

Una rete con tutti gli input connessi direttamente agli output è detta **rete neurale a singolo strato** (**single-layer neural network**) o **perceptron network**. La prima cosa da notare è che un perceptron network con m output è in realtà composto da m reti separate, dato che ogni peso lavora su uno solo degli output. pertanto, ci saranno m processi di training separati. inoltre, in base alla funzione di attivazione usata, il processo di training sarà la **perceptron learning rule** o regola di discesa del gradiente delle **regressione logistica**.

4.6.3 Reti neurali feed-forward a più strati

Allenare questa rete risulta semplice se pensiamo la rete nel modo giusto: come una funzione $h_{\mathrm{W}}(\mathrm{x})$ parametrizzata dai pesi W. Dato che la funzione rappresentata dalla reta può essere molto non lineare, composta quindi da soft threshold function nidificate, possiamo vedere la rete neurale come un mezzo per effettuare una **regressione non lineare**.

4.6.4 Apprendere nelle reti a più strati

In caso di interazioni tra i problemi di apprendimento nel caso che la rete possiede output multipli, la rete andrebbe vista come l'implementazione di una funzione vettore $\mathrm{h_w(x)}$ piuttosto che una funzione scalare $h_{\mathrm{w}}(\mathrm{x})$: in questo caso l'output sarà un vettore Y. Seppur un perceptron network si decompone in m problemi di apprendimento separati per un problema ad m output, questa decomposizione fallisce nelle reti a più strati. Questa dipendenza è molto semplice nel caso di una qualsiasi loss function che sia *additiva* attraverso i componenti del vettore di errore $\mathrm{y} - \mathrm{h_w(x)}$. Per la L_2 loss, abbiamo, per qualsiasi peso w,

$$\frac{\delta}{\delta w} Loss(\mathrm{w}) = \frac{\delta}{\delta w}|\mathrm{y} - \mathrm{h_w(x)}|^2 = \frac{\delta}{\delta w}\sum_k (y_k - a_k)^2 = \sum_k \frac{\delta}{\delta w}(y_k - a_k)^2$$

Dove l'index k scorre su tutti i nodi dello strato di output. Ogni termine nella sommatoria finale è solo il gradiente della loss del k-esimo output, calcolato come se gli altri output non

esistessero. Pertanto, possiamo decomporre un problema di apprendimento con m output in m problemi di apprendimento. L'error rate degli strati nascosti sembrano misteriosi dato che i dati di training non dicono quale valore i nodi nascosti devono avere. Possiamo **propagare all'indietro** (**back-propagate**) l'errore dallo strato di output agli strati nascosti. Sia Err_k il k-esimo elemento $\mathrm{y} - \mathrm{h_w(x)}$, sia $\Delta_k = Err_k \cdot g'(in_j)$, l'aggiornamento dei pesi diventa

$$w_{jk} \leftarrow w_{jk} + \alpha \cdot a_j \cdot \Delta_k$$

Per aggiornare la connessione tra le unità d'input e le unità nascoste, dobbiamo definire una quantità analoga al termine d'errore per i nodi di output. L'idea è che il nodo nascosto j è "responsabile" di una frazione dell'errore Δ_k in ognuno dei nodi di output ai quali si connette. Pertanto, il valore Δ_k è diviso in base alla forza della connessione tra il nodo nascosto e il nodo di output ed è propagato al'indietro per fornire i valori Δ_j per gli strati nascosti. La regola di propagazione per i valori Δ_j è la seguente:

$$\Delta_j = g'(in_j) \sum_k w_{jk} \Delta_k$$

Ora la regola di aggiornamento dei pesi per i vari pesi tra gli strati di input e quelli nascosti è essenzialmente identica alla regola di aggiornamento per lo strato di output:

$$w_{ij} \leftarrow w_{ij} + \alpha \cdot a_j \cdot \Delta_j$$

Il processo di propagazione all'indietro può essere riassunto come segue:

- Calcola i valori di Δ per le unità di output, usando l'errore osservato;
- Partendo dallo strato di output, ripete il seguente procedimento per ogni strato della rete, finché non viene raggiunto il primo strato nascosto:
  - Propaga all'indietro il valore di Δ al precedente strato;
  - Aggiorna i pesi tra i due strati.

Poniamo di cacolare solo il gradiente per $Loss_k = (y_k - a_k)^2$ al k-esimo output. Il gradiente di questa loss riguardo ai pesi che connettono lo strato nascosto con lo strato di output saranno zero tranne il peso w_{jk} che connette con la k-esima unità di output. Per questi pesi, abbiamo

$$\begin{aligned}
\frac{\delta Loss_k}{\delta w_{jk}} &= -2(y_k - a_k)\frac{\delta a_k}{\delta w_{jk}} = -2(y_k - a_k)\frac{\delta g(in_k)}{\delta w_{jk}} = \\
&= -2(y_k - a_k)g'(in_k)\frac{\delta in_k}{\delta w_{jk}} = -2(y_k - a_k)g'(in_k)\frac{\delta}{\delta w_{jk}}(\sum_j w_{jk}a_j) = \\
&= -2(y_k - a_k)g'(in_k)a_j = -a_j\Delta_k
\end{aligned}$$

Per ottenere il gradiente riguardo ai pesi w_ij che connetto lo strato di input allo strato nascosto, dobbiamo espandere gli attivatori a_j:

$$\begin{aligned}
\frac{\delta Loss_k}{\delta w_{ij}} &= -2(y_k - a_k)\frac{\delta a_k}{\delta w_{ij}} = -2(y_k - a_k)\frac{\delta g(in_k)}{\delta w_{ij}} = \\
&= -2(y_k - a_k)g'(in_k)\frac{\delta in_k}{\delta w_{ij}} = -2\Delta_k \frac{\delta}{\delta w_{jk}}(\sum_j w_{jk}a_j) = \\
&= -2\Delta_k w_{jk}\frac{\delta a_j}{\delta w_{jk}} = -2\Delta_k w_{jk}\frac{\delta g(in_j)}{\delta w_{jk}} = \\
&= -2\Delta_k w_{jk}g'(in_j)\frac{\delta in_j}{\delta w_{jk}} = \\
&= -2\Delta_k w_{jk}g'(in_j)\frac{\delta}{\delta w_{jk}}(\sum_j w_{ij}a_i) = \\
&= -2\Delta_k w_{jk}g'(in_j)a_i = -a_i\Delta_j
\end{aligned}$$

4.6.5 Apprendere la struttura delle reti neurali

Le reti neurali sono soggette all'**overfitting** quando sono presenti troppi parametri nel modello. Se ci atteniamo alle reti neurali completamente connesse, l'unica scelta da fare riguarda il numero di strati nascosti e la loro grandezza. L'approccio usuale è quello di fare varie prove e tenere da parte i risultati migliori. Le tecniche di **cross validation** sono necessarie per evitare il **peeking** al set di test. Se vogliamo considerare reti che non sono completamente connesse, allora dobbiamo trovare metodi efficaci all'intero dell'enorme spazio delle possibili topologie di connessione. Uno di questi algoritmi è l'**optical brain damage**. L'algoritmo di **tiling**, invece, ricorda il decision-list learning, e sfrutta l'idea di partire con una sola unità che fà del suo meglio per produrre il risultato corretto sul maggior numero di esempi di training possibile, per poi aggiungere sequenzialmente varie unità per occuparsi dei vari esempi che la prima unità ha sbagliato.

4.7 Modelli non parametrici

La regressione lineare e le reti neurali usano i dati di training per stimare un set fissato di parametri w. Ciò definisce la nostra ipotesi $h_{\mathrm{w}}(\mathrm{x})$, e a quel punto possiamo eliminare i dati di training, dato che sono riassunti da w. Un modello di apprendimento che riassume i dati con un set di parametri dalla grandezza fissata (indipendente dal numero di esempi di training) è detto **modello parametrico**. Un **modello non parametrico** è un modello che non può essere caratterizzato da un set finito di parametri. un esempio di approccio per modelli non parametrici è **l'apprendimento basato su istanze** (**istance-based learning**) o **apprendimento basato su memoria** (**memory-based learning**). Il metodo più semplice di apprendimento basato su istanza è il **table lookup**: prende tutti gli esempi di training e li pone in una tabella di ricerca, e quando viene richiesto $h(\mathrm{x})$, controlla se x è presente nella tabella; se sì, ritorna la corrispondente y. Il problema con questo metodo è che non generalizza bene: quando x non è nella tabella tutto quello che si può fare è ritornare un valore di default.

4.7.1 Modello nearest neighbor

Il k-**nearest neighbors** lookup è una variazione del table lookup: data una query x_q trova i k esempi che sono più *vicini* ad x_q. Sia $NN(k, \mathrm{x}_q)$ il set dei k vicini più vicini, si prende il voto di pluralità dei vicini (che è il voto di maggioranza nel caso di classificazione binaria). Per evitare pareggi, k è sempre un numero dispari. Per effettuare una regressione, possiamo prendere la media o la mediana dei k vicini, o possiamo risolvere un problema di regressione lineare sui vicini. I metodi non parametrici rimangono soggetti all'underfitting e all'overfitting, come i metodi parametrici. Per misurare la distanza da una query x_q ad un punto d'esempio x_j possiamo usare la **distanza di Minkowsky** o la norma L^p, definita come

$$L^p(\mathrm{x}_j, \mathrm{x}_q) = (\sum_i |x_{ji} - x_{qi}|^p)^{\frac{1}{p}}$$

Con $p = 2$ abbiamo la distanza euclidea[8] e con $p = 1$ abbiamo la distanza di Manhattan[9]. Con attributi a valori booleani, il numero di attributi per cui due punti differiscono è detta **distanza di Hamming**. Viene spesso usato la distanza euclidea se le dimensioni misurano proprietà simili, come le grandezze, mentre la distanza di Manhattan viene usata se sono dissimili. Se usiamo numeri grezzi da ciascuna dimensione allora la distanza totale sarà afflitta da un cambio di scala su qualsiasi dimensione. Per evitare ciò, è comune applicare la **normalizzazione** alle misure in ogni dimensione. Un approccio semplice è quello di calcolare la media μ_i e la deviazione standard σ_i dei valori in ciascuna dimensione, e scalarli in modo che x_{ij} diventa $\frac{x_{ij} - \mu_i}{\sigma_i}$. Una metrica più complessa conosciuta come la **distanza di Mahalanobis** prende in considerazione la covarianza tra le dimensione. In spazi con poche dimesioni e molti dati, nearest neighbor lavora molto bene. Ma quando il numero di dimensioni aumenta incontriamo un problema: i vicini più vicini in spazi ad alta dimensione non sono di solito molto vicini. Consideriamo k vicini più vicini in un set di dati composto da N punti distribuiti in modo uniforme all'interno di un ipercubo n-dimensionale con lunghezza 1. Sia l la lunghezza laterale media di un vicinato. Allora il volume del vicinato (che contiene k punti) è l^n e il volume dell'intero cubo (che contiene N punti) è 1. Pertanto, in media $l^n = \frac{k}{N}$. Prendendo la n-esima radice di entrambi i lati abbiamo $l = (\frac{k}{N})^{\frac{1}{n}}$. Questo problema viene chiamato la **maledizione della dimensionalità**. La funzione $NN(k, \mathrm{x}_q)$ è concettualmente banale: dato un set di N esempi e una query x_q, itera tra i vari esempi, misura per ciascuno la distanza da x_q, e mantieni i migliori k. Se siamo soddisfatti di un'implementazione che richiede $O(n)$ tempo di esecuzione, allora quella è la fine della storia. Ma i metodi basati sulle istanze sono impiegati per enormi set di dati, pertanto è preferibile un algoritmo con tempo di esecuzione sublineare. L'analisi elementare degli algoritmi ci dice che il tempo di una ricerca esatta nella tabella è $O(n)$ con una tabella sequenziale, $O(\log n)$ con un albero binario, e $O(1)$ con una tabella hash.

[8] $L^2(\mathrm{x}_j, \mathrm{x}_q) = \sqrt{\sum_i (x_{ji} - x_{qi})^2}$

[9] $L^1(\mathrm{x}_j, \mathrm{x}_q) = \sum_i |x_{ji} - x_{qi}|$

4.7.2 Trovare i vicini più vicini con i k-d tree

Un albero binario bilanciato su dei dati con un numero arbitrario di dimensioni è detto **k-d tree**, per alberi k-dimensionali. La costruzione di k-d tree è simile alla costruzione di un albero binario bilanciato ad una dimensione. Iniziamo con un set di esempi e al nodo radice andiamo a dividerli lungo la i-esima dimensione testando se $x_i \leq m$. Scegliamo il valore m come la mediana degli esempi lungo la I-esima dimensione; così metà degli esempi sarà nel ramo sinistro dell'albero e metà in quello destro. A questo punto creiamo ricorsivamente un albero dal set sinistro e destro di dati fermandoci quando ci sono meno di due esempi rimanenti. Per scegliere una dimensione su cui dividere ogni nodo dell'albero, si può scegliere la dimensione $imodn$ al livello i dell'albero. Un'altra strategia è quella di dividere sulla dimensione che ha la più ampia diffusione di valori. Ogni ricerca su un k-d treeè come una ricerca su un albero binario. Ma la ricerca dei vicini più vicini è molto più complicata. Mentre scendiamo tra i rami, dividendo gli esempi a metà, in alcuni casi possiamo scartare l'altra metà degli esempi. Ma non sempre è possibile. In alcuni casi il punto che stiamo interrogando cade molto vicino al confine di divisione, Il punto di query potrebbe trovarsi sul ramo sinistro del confine, ma uno o più dei k vicini più vicini potrebbero essere sul lato destro. Dobbiamo verificare per questa possibilità calcolando la distanza del punto di query dal confine di divisione, e cerca a quel punto su entrambi i lati se non troviamo k esempi sul lato sinistro che sono più vicini della distanza calcolata. A causa di questo problema, k-d tree sono appropriati solo quando abbiamo molti più esempi che dimensioni, preferibilmente almeno 2^n esempi.

4.7.3 Hashing sensibile alla località

Le tabelle hash hanno il potenziale do fornire ricerche ancora più veloci rispetto agli alberi binari. I codici hash distribuiscono casualmente i valori nei vari bidoni, ma vogliamo avere i punti vicini raggruppati insieme nello stesso bidone; vogliamo un **hashing sensibile alla località** (**locality-sensitive hashing** o **LSH**). Definiamo il problema dei **vicini approssimativi** (**approximate near-neighbors**): dato un set di dati di punti d'esempio e un punto di query x_q, trovare, con alta probabilità un punto (o punti) d'esempio vicini a x_q. Per essere precisi, richiediamo che se esiste un punto x_j entro un raggio r di x_q, allora l'algoritmo troverà un punto $\mathrm{x}_{j'}$ con alta probabilità entro un raggio $c \cdot r$ di x_q. Se non esiste alcun punto entro un raggio r allora l'algoritmo potrà riportare un fallimento. I valori di c e "alta probabilità" sono parametri dell'algoritmo. Per risolvere questo problema, necessitiamo di una funzione hash $g(\mathrm{x})$ che possiede la proprietà che, que due punti qualsiasi x_j e $\mathrm{x}_{j'}$, la probabilità che abbiano lo stesso codice hash è minore se la loro distanza è maggiore di $c \cdot r$, ed è maggiore se la loro distanza è minore di r. L'intuizione su cui ci basiamo è che se due punti sono vicini in uno spazio n-dimensionale, allora saranno necessariamente vicini quando proiettati in uno spazio ad una dimensione (una linea). Infatti, possiamo discretizzare la linea in bidoni, o secchi hash, in modo che, con alta probabilità, punti vicini siano all'interno dello stesso bidone. Il trucco dell'LSH è di creare *multiple* proiezioni casuali e combinarle. Una proiezione è un subset casuale della rappresentazione della stringa di bit. Scegliamo l differenti proiezioni casuali e creiamo l tabelle hash $g_1(\mathrm{x}), g_2(\mathrm{x}), ..., g_l(\mathrm{x})$. A questo punto inseriamo tutti gli esempi nelle tabelle hash. Quando viene fornito un punto di query x_q, recuperiamo il set di punti nel bidone $g_k(\mathrm{x})$ per ogni k, e uniamo questi set insieme in un unico set di punti candidati C. A questo punto calcoliamo la distanza effettiva da x_q per ognuno dei punti di C e ritorniamo i k punti più vicini. Con alta probabilità, ognuno dei punti che sono vicini

a x_q sarà presente in almeno ognuno dei bidoni, e seppure saranno presenti punti molto lontani, possiamo ignorarli.

4.7.4 Regressione non parametrica

Il metodo noto informalmente come "collega i punti" e arrogantemente come "regressione lineare non parametrica a pezzi" crea una funzione $h(x)$ che, data una query x_q, risolve il normale problema di regressione lineare con solo due punti: gli esempi di training immediatamente a sinistra e a destra di x_q. Quando il rumore è basso, questo metodo non è male. La **regressione con k-nearest neighbors** migliora rispetto al metodo "collega i punti". Invece di usare solo i due punti a destra e a sinistra di x_q, usiamo i k vicini più vicini. Un grande valore di k tende ad appianare la grandezza dei picchi, seppur la funzione risultante presenta discontinuità. Nella *k-nearest-neighbors average*, $h(x)$ è la media dei k punti, ossia

$$h(x) = \frac{\sum_{j=1}^{k} y_j}{k}$$

Mentre nella *k-nearest-neighbors regression* viene trovata la migliore linea usando k esempi. La **regressione locale pesata** (**locally weighted regression**) ci dà il vantaggio di nearest neighbors senza le discontinuità. Per evitare discontinuità in $h(x)$, dobbiamo evitare discontinuità nel set di esempi che usiamo per stimare $h(x)$. L'idea della regressione locale pesata è che ad ogni punto di query x_q, gli esempi che sono vicini ad x_q hanno peso maggiore, e gli esempi che sono ontani hanno peso minore o uguale a zero. Il decremento del peso in base alla distanza è sempre graduale e non improvviso. Il **kernel** è una funzione il cui grafico ricorda un bernoccolo. Non possiamo scegliere qualsiasi funzione come kernel. Notare prima di tutto che chiamare una funzione kernel $\mathcal{K}$ con $\mathcal{K} * Distance(\mathrm{x}_j, \mathrm{x}_q)$, dove x_q è un punto di query che è a una data distanza da x_j, e vogliamo sapere quanto vale il peso da assegnare. Pertanto $\mathcal{K}$ dovrebbe essere simmetrica intorno allo 0 e deve valere massimo in 0, Dobbiamo fare attenzione all'ampiezza del kernel (questo è un parametro del modello che è scelto tramite cross-validation). Per un dato punto di query x_q bisogna risolvere il seguente problema di regressione pesato usando la discesa del gradiente:

$$\mathrm{w}^* = \underset{\mathrm{w}}{\operatorname{argmin}} \sum_j \mathcal{K}(Distance(\mathrm{x}_q, \mathrm{x}_j))(y_i - \mathrm{w} \cdot \mathrm{x}_j)^2$$

Dove *Distance* è una qualsiasi distanza metrica discussa per i vicini più vicini. Allora la risposta è $h(\mathrm{x}_q) = \mathrm{w}^* \cdot \mathrm{x}_q$ Notare che è necessario risolvere un nuovo problema di regressione per ogni punto di query (ecco perché è detto *locale*). A mitigare contro questo lavoro extra è il fatto che ogni problema di regressione sarà più semplice, dato che coinvolge solo gli esempi con pesi non uguali a zero (gli esempi il cui kernel si sovrappone al punto di query). Quando la grandezza del kernel è piccola, i punti considerati sono pochi. Moli modelli non parametrici hanno il vantaggio che è più semplice fare leave-one-out cross-validation senza dover ricalcolare tutto.

4.8 Macchine a supporto di vettori

Le **macchine a vettori di supporto** (**support vector machine**), o **SVM** framework, sono attualmente l'approccio più popolare per l'apprendimento supervisionato "disponibile subito".Ci sono tre proprietà che rendono SVM attrattivo:

- Gli SVM costruiscono un separatore con un margine massimo, ossia un confine decisionale con la più grande distanza possibile dei punti di esempio, e ciò permette di generalizzare bene.
- Gli SVM creano un iperpiano lineare separatore, ma hanno l'abilità di inserire i dati in uno spazio con maggiori dimensioni usando il cosiddetto kernel trick.
- Gli SVM sono metodi non parametrici, Ossia trattengono alcuni esempi di training e necessitano potenzialmente di immagazzinarli tutti. In pratica di solito finiscono per trattenere solo una piccola frazione del numero di esempi, alcune volte Solo un multiplo del numero delle dimensioni, pertanto le SVM combinano i vantaggi dei modelli non parametrici e parametrici, Avendo la flessibilità di rappresentare funzioni complesse rimanendo resistenti all'overfitting.

L'intuizione chiave delle SVM è che alcuni esempi sono più importanti di altri e portare maggiore attenzione ad essi può portare ad una migliore generalizzazione. Le SVM affrontano questo problema: invece di minimizzare l'*empirical loss* prevista sui dati di training, provano a minimizzare la generalization loss prevista. Sotto l'ipotesi probabilistica che gli esempi sono estratti dalla stessa distribuzione di quelli visti prima, Si può minimizzare la *generalization loss* scegliendo il separatore che è il più lontano dagli esempi che abbiamo visto finora: questo separatore prende il nome di **separatore con margine massimo**, dove il **margine** è il doppio della distanza dal separatore al punto di esempio più vicino. Tradizionalmente gli SVM usano la convenzione che le etichette delle classi sono $+1$ e -1 invece dei $+1$ e 0 che abbiamo usato finora. Inoltre Le SVM non pongono l'intercetta all'interno del vettore dei pesi w, bensì in un parametro separato b. Con questo in mente, il separatore è definito come il set di punti $\{\mathrm{x} : \mathrm{w} \cdot \mathrm{x} + b = 0\}$. Possiamo effettuare una ricerca nello spazio di w e b con la discesa del gradiente per trovare i parametri che massimizzano il margine mentre classificano correttamente tutti gli esempi. Esiste una rappresentazione alternativa, detta rappresentazione doppia, in cui la soluzione ottima è trovata risolvendo

$$\underset{\alpha}{\operatorname{argmax}} \sum_j \alpha_j - \frac{1}{2} \sum_{j,k} \alpha_j \alpha_k y_j y_k (\mathrm{x}_j \cdot \mathrm{x}_k)$$

Soggetta alle costanti $\alpha_j \geq 0$ e $\sum_j \alpha_j y_j = 0$. Questo è un problema di ottimizzazione della **programmazione quadratica**, Per il quale esistono buoni pacchetti di software. Una volta che abbiamo trovato il vettore α possiamo tornare a $\mathrm{w} = \sum_j \alpha_j \mathrm{x}_j$, o possiamo rimanere nella rappresentazione duale. Esistono tre proprietari importanti riguardo questa equazione:

- L'espressione è convessa ossia a un singolo massimo globale che può essere trovato efficientemente;

- I dati entrano nell'espressione solo sotto forma di prodotto scalare di coppie di punti, infatti una volta che l'α_j ottimo è stato calcolato, abbiamo

$$h(\mathrm{x}) = \mathrm{sign}(\sum_j \alpha_j y_j (\mathrm{x} \cdot \mathrm{x}_j) - b)$$

- I pesi α_j associati ad ogni punto di dati sono uguali a zero esclusi quelli per i **vettori di supporto**, ossia i punti più vicini al separatore.

Se i dati sono mappati all'interno di uno spazio con dimensione sufficientemente alta, allora saranno sicuramente linearmente separabili, infatti se si osserva un punto da abbastanza direzioni si troverà un modo di farli allineare. In generale, con alcuni casi speciali previsti, se abbiamo N punti di dati allora saranno sempre separabili in spazi con almeno $N-1$ dimensioni. Di solito non ci aspettiamo di trovare un separatore lineare nello spazio di input x, ma possiamo trovare separatori lineari nello spazio delle caratteristiche ad alta dimensione $F(\mathrm{x})$ semplicemente rimpiazzando $\mathrm{x}_j \cdot \mathrm{x}_k$ con $F(\mathrm{x}_j) \cdot F(\mathrm{x}_k)$. Spesso $F(\mathrm{x}_j) \cdot F(\mathrm{x}_k)$ può essere calcolato senza dover calcolare prima F per ciascun punto. La **funzione kernel**, scritta di solito come $K(\mathrm{x}_j, \mathrm{x}_k)$, può essere applicata ad una coppia di dati di input per valutare il prodotto scalare in uno spazio delle caratteristiche corrispondente. Pertanto, possiamo trovare separatori dimensionali in spazi delle caratteristiche ad alte dimensioni $F(\mathrm{x})$ semplicemente rimpiazzando $\mathrm{x}_j \cdot \mathrm{x}_k$ con una funzione kernel $K(\mathrm{x}_j, \mathrm{x}_k)$. Pertanto, possiamo apprendere nello spazio ad alta dimensioni, ma calcoleremo soltanto funzioni kernel piuttosto che tutta la lista di caratteristiche per ogni punto di dati. Il **teorema di Mercer** (1909) ci dice che ogni funzione kernel "ragionevole" corrisponde ad *alcuni* spazi delle caratteristiche: per esempio, il **kernel polinomiale** $K(\mathrm{x}_j, \mathrm{x}_k) = (1 + \mathrm{x}_j \cdot \mathrm{x}_k)^d$ corrisponde a uno spazio delle caratteristiche la cui dimensione è esponenziale in d. Il kernel trick consiste nell'inserire questi kernel nell'equazione, in modo che il separatore lineare ottimo può essere trovato efficientemente in spazi di caratteristiche con miliardi (o, in alcuni casi, infinite) dimensioni. Nel caso di dati intrinsecamente rumorosi, ci piacerebbe avere una superficie di decisione in uno spazio con poche dimensioni che non divide chiaramente le classi, ma riflette la realtà dei dati rumorosi. Questo è possibile con i **soft margin classifier**, che permettono agli esempi di cadere dalla parte sbagliata del confine, ma assegnargli una penalità proporzionale alla distanza richiesta per riportarli nella parte corretta. Il metodo del kernel può essere applicato con qualsiasi algoritmo che può essere riformulato per lavorare non solo con prodotti scalari di punti di dati. Una volta fatto, il prodotto scalare è rimpiazzato da una funzione kernel, avendo così una versione **kernelizzata** dell'algoritmo.

4.9 Apprendimento d'insieme

L'idea dei metodi di **apprendimento d'insieme** (**ensemble learning**) è quello di scegliere una collezione, o **insieme**, di ipotesi dallo spazio delle ipotesi e combinare le loro predizioni. È possibile vedere l'insieme di ipotesi come una singola ipotesi e lo spazio delle ipotesi come tutti i possibili insiemi di ipotesi costruibili dalle ipotesi nello spazio originale. Se lo spazio delle ipotesi originale consente un algoritmo di apprendimento semplice ed efficiente, allora il metodo d'insieme fornisce un modo per apprendere una classe di ipotesi molto più espressiva senza incorrere in una complessità di calcolo o algoritmica maggiore. Il metodo di apprendimento d'insieme più usato è detto **boosting**. Un **set di training**

pesato (**weighted training set**) associa ad ogni esempio un peso $w_j \geq 0$. Più alto il peso di un esempio, maggiore è la sua importanza durante l'apprendimento di un ipotesi. Il boosting parte sempre con $w_j = 1$ per tutti gli esempi. Da qui, genera la prima ipotesi, h_1. Questa ipotesi classificherà alcuni degli esempi di training correttamente ed altri incorrettamente. Vorremmo che la prossima ipotesi facesse meglio con gli esempi classificati male, pertanto incrementiamo i loro pesi mentre decrementiamo i pesi di quelle classificate correttamente. Da questo nuove set di training pesato, generiamo l'ipotesi h_2. Questo processo continua così finché non generiamo K ipotesi, dove K è un input dell'algoritmo di boosting. L'insieme finale delle ipotesi è una combinazione a maggioranza ponderata di tutte le ipotesi K, ognuna pesata in base a come si è comportata con il set di training. L'algoritmo AdaBoost ha una proprietà interessante: se l'algoritmo di apprendimento in input L è un algoritmo di **apprendimento debole**, ossia che L restituisce un'ipotesi con un accuratezza del set di training che è leggermente meglio della scelta casuale, allora AdaBoost ritornetà un'ipotesi che *classifica i dati di training perfettamente* per valodi di K abbastanza grandi. Pertanto, l'algoritmo *incentiva* l'accuratezza dell'algoritmo di apprendimento originale sul set di training.

Algorithm 10 AdaBoost function

```
1: function AdaBoost(examples, L, K)
2:     for k = 1 to K do
3:         h[k] ← L(examples, w)
4:         error ← 0
5:         for j = 1 to N do
6:             if h[k](x_j) ≠ y_j then
7:                 error ← error + w[j]
8:             end if
9:         end for
10:        for j = 1 to N do
11:            if h[k](x_j) = y_j then
12:                w[j] ← w[j] · error / (1 − error)
13:            end if
14:        end for
15:        w ← Normalize(w)
16:        z[k] ← log (1 − error) / error
17:    end for
18:    return Weighted-Majority(h, z)
19: end function
```

Un esempio di struttura su cui applicare l'algoritmo di boosting sono i **ceppi decisionali** (**decision stumps**), ossia alberi decisionali con un solo test alla radice. Un dettaglio interessante è che la performance del set di test continua a crescere anche dopo che l'errore del set di training ha raggiunto lo zero. Inoltre, le predizioni *migliorarno* man mano che l'insieme delle ipotesi diventa sempre più complesso. Il rasoio di Ockmar ci dice di non creare ipotesi più complesse del necessario, ma nel caso del boosting le predizioni *migliorano* man mano che l'insieme delle ipotesi diventa sempre più complesso. Un modo per spiegare ciò è che il boosting approssima il **Bayesian learning**, che si può mostrare essere un un algoritmo di apprendimento ottimale, e l'approssimazione migliora quanto le

iptesi che vengono aggiunte. Un'altra possibile spiegazione è che l'addizione di ulteriori ipotesi permette all'insieme di essere *più definito* nella distinzione tra esempi positivi e negativi, il che aiuta quando è necessario classificare nuovi esempi.

4.9.1 Online learning

Quando i dati non sono i.i.d., è possibile utilizzare l'**online learning**, che usa la seguente idea: un agente riceve un input x_j dalla natura, predice la corrispondente y_j, e gli viene detta poi la risposta corretta. Questo procedimento viene ripetuto con x_{j+1} e così via. L'**algoritmo di maggioranza pesato e casuale** (**randomized weighted majority algorithm**) implementa l'online learning su K agenti nel seguente modo:

- Inizializza un set di pesi $\{w_,w_2, ..., w_K\}$ tutti ad 1;
- Riceve le predizioni $\{\hat{y}_1, haty_2, ..., haty_K\}$ dagli agenti;
- Sceglie casualmente un agente k^*, in proporzione al suo peso (la probabilità di scegliere un agente k è $P(k) = \dfrac{w_k}{\sum_{k'} w_{k'}}$);
- Predice $\hat{y}_{k^*}$;
- Riceve la risposta corretta y;
- Per ogni agente k tale che $\hat{y}_{k^*} \neq y$ aggiorna $w_k \leftarrow \beta w_k$.

Dove $0 < \beta < 1$ è un numero che indica quanto penalizzare una esperto per una scelta sbagliata. Misuriamo il successo di questo algoritmo in termini di **rimorso**, che è definito come il numero di errori addizionali che facciamo comparati con l'agente $k \in [1, K]$ con il miglior sotirco delle predizioni. Sia M^* il numero di errori fatti da h. Allora il numero di errori M fatto dall'algoritmo di maggioranza pesato e casuale è limitato da

$$M < \frac{M^* \ln \frac{1}{\beta} + \ln K}{1 - \beta}$$

Questo limite vale per una *qualsiasi* sequenza di esempi, anche quelle scelte da avversari che stanno cercando di fare del loro peggio. In generale, se β è vicino ad 1 allora siamo reattivi ai cambiamenti nel lungo periodo; se il miglior agente cambia, lo riprenderemo presto. Tuttavia, paghiamo una penalità all'inizio, quando iniziamo con tutti gli agenti con pesi uguali; potremmo accettare la predizione di un cattivo agente per troppo tempo. Quando β è vicino a 0, questi due fattori sono invertiti. Possiamo scegliere β in modo che sia asintoticamente vicino a M^* nel lungo periodo; questo è detto **algoritmo senza rimpianti** (**no-regret algorithm**). L'online learning è utile quando i dati possono cambiare rapidamente nel tempo. È utile anche per applicazioni che usano una grande collezione di dati che aumenta costantemente, anche se i cambiamenti sono graduali.

Capitolo 5

Apprendimento per rinforzo

Con **reward** (**ricompensa**) o **rifonzo** si indica un feedback fornito all'agente per indicargli se ha raggiunto un esito positivo o negativo. il compito dell'**apprendimento per rinforzo** (**reinforcement learning**) è di usare le reward osservate per apprendere una policy ottima (o quasi ottima) per l'ambiente. L'apprendimento per rinforzo può essere considerato per incoraggiare tutte le IA: un agente è piazzato in un ambiente e deve imparare a comportarsi con successo al suo interno. Ci concentreremo su ambienti semplici e agenti con design semplice, inoltre assumeremo che l'ambiente è completamente osservabile, cosicché lo stato corrente è fornito da ogni percezione. Inoltre assumeremo che l'agente non conosce in che modo l'ambiente lavora o cosa fa la sua azione, e permetteremo e consentiremo risultati probabilistici dell'azione. pertanto, l'agente deve affrontare un problema decisionale di Markov sconosciuto. Considereremo tre design di agenti:

- Un **utility-based agent** impara una funzione di utility sugli stati e la usa per selezionare azioni che massimizzano l'utility in output prevista;
- Un **Q-learning agent** impara un'**action-utility function**, o **Q-function**, restituendo l'utility prevista in base all'azione e lo stato dati in input;
- Un **reflex agent** impara una policy che mappa direttamente dagli stati alle azioni.

Un utility-based agent deve avere inoltre un modello dell'ambiente in modo da poter effettuare decisioni, dato che deve conoscere lo stato in cui l'azione porterà. Un Q-learning agente può comparare l'utility prevista delle varie scelte disponibili senza dover sapere i loro risultati, pertanto non richiede un modello dell'ambiente. Poiché i Q-learning agent non sanno dove portano le loro azioni, non possono guardare avanti. Nell'**apprendimento passivo** la policy dell'agente è fissa e il compito è quello di imparare l'utility degli stati (o le coppie azione-stato); questo può anche coinvolgere l'apprendimento di un modello del mondo. Nell'**apprendimento attivo** l'agente deve anche apprendere cosa fare. Il problema principale è l'**esplorazione**: un agente deve fare più esperienza possibile sull'ambiente in modo da capire come comportarsi al suo interno.

5.1 Apprendimento per rinforzo passivo

Iniziamo dal caso di un passive learning agent usando una rappresentazione basata sugli stati in un ambiente completamente osservabile. Nell'apprendimento passivo, la policy dell'agente π è fissa: in uno stato s, viene sempre eseguita l'azione $\pi(s)$. Il suo

obiettivo è semplicemente imparare quanto è buona la policy, ossia imparare la funzione di utility $U^{\pi}(s)$. Chiaramente, l'attività di apprendimento passivo è simile all'attività di **valutazione delle policy**. La differenza principale è che il passive learning agent non conosce il **modello di transizione** $P(s|s,a)$, che specifica la probabilità di raggiungere uno stato s' dallo stato s dopo aver effettuato un'azione a; non conosce neanche la **funzione di reward** $R(s)$, che specifica la reward per ogni stato. L'agente esegue una serie di **prove** nell'ambiente usando la policy π. Lo scopo è quello di usare le informazioni sulle reward per imparare l'utility prevista $U^{pi}(s)$ associata ad ogni stato non deterministico s. L'utility è definita come la somma prevista delle reward (scontate) ottenute se viene seguita la policy π:

$$U^{\pi}(s) = E[\sum_{t=0}^{\infty} \gamma^t R(S_t)]$$

Dove $R(s)$ è la reward per uno stato, S_t (una variabile casuale) è lo stato raggiunto al tempo t quando viene eseguita la policy π, e $S_0 = s$. Includeremo un **fattore di sconto** γ in tutte le nostre equazioni.

5.1.1 Stima diretta dell'utility

un metodo semplice per la **stima diretta dell'utility** è stato inventato alla fine degli anni '50 nell'area della **teoria del controllo adattivo** da Widrow e Hoff (1960). L'idea è che l'utility di uno stato è il reward totale previsto da quello stato in poi (chiamata la **reward-to-go** prevista), e ogni prova fornisce un *campione* di questa quantità per ogni stato visitato. Alla fine di ogni sequenza, l'algoritmo calcola la reward-to-go osservata per ogni stato e aggiorna l'utility prevista per ogni stato in accordo, semplicemente mantenendo una media corrente per ogni stato in una tabella. Nel limite di infinite prove diverse, la media dei campioni convergerà al valore atteso. È chiaro che la stima diretta dell'utility è solo un istanza dell'apprendimento supervisionato dove ogni esempio a lo stato come input e la reward-to-go osservata come output appunto questo significa che abbiamo ridotto l'apprendimento per rinforzo ha un problema standard di apprendimento induttivo. La stima diretta dell' utility succede nel ridurre il problema di apprendimento per rinforzo in un problema di apprendimento induttivo, per quello che noi sappiamo. Sfortuna nettamente, non considera una fonte di informazioni molto importante, ossia il fatto che le utility degli stati non sono indipendenti. L' utility di ogni stato è uguale alla sua reward più l'utility prevista per il suo stato successivo, come indicato nell'equazione di Bellman:

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s)) U^{\pi}(s')$$

Ignorando la connessione tra gli stati, la stima diretta dell'utility perde occasione per l'apprendimento.

5.1.2 Adaptive dynamic programming

Un **adaptive dynamic programming** (o **ADP**) agent prende il vantaggio dei vincoli insieme alle utility degli stati prendendo il modello di transizione che li connette e risolvendo il corrispettivo processo decisionale di Markov usando un metodo di programmazione dinamica. Per un passive learning agent, questo significa inserire il modello di

transizione appreso e le reward osservate all'interno dell'equazione di Bellman per calcolare le utility degli stati. Queste equazioni sono lineari (non involgono massimizzazione) quindi possono essere risolte utilizzando un qualsiasi package di algebra lineare. Alternativamente, possiamo adottare l'approccio di **iterazione della policy modificata** (**modified policy iteration**), usando un processo di iterazione del valore semplificato per aggiornare le stime dell'utility dopo ogni cambio sul modello ha appreso. Poiché il modello di solito cambia soltanto leggermente con ogni osservazione, il processo di iterazione del valore può usare le precedenti stime delle utility come valori iniziali e dovrebbe convergere abbastanza velocemente. Il processo di apprendimento del modello stesso è semplice, poiché l'ambiente è completamente osservabile. Questo significa che abbiamo un task di apprendimento supervisionato dove l'input è una copia stato-azione e l'output è lo stato risultante. Nel caso più semplice, possiamo rappresentare il modello di transizione che come una tabella di probabilità. Teniamo traccia di quanto spesso ogni risultato di un'azione occorre e stimare la probabilità di transizione $P(s'|s,a))$ dalla frequenza con cui s' è raggiunto quando viene eseguito a in s. In termini di quanto velocemente il valore previsto migliora, l'ADP agent è limitato soltanto dalla sua abilità di imparare il modello di transizione. In questo senso, fornisce uno standard contro il quale misurare altri algoritmi di apprendimento rinforzato. È tuttavia intrattabile per enormi spazi di stati. L'algoritmo utilizza la stima di massima verosomiglianza per apprendere il modello di transizione; inoltre, scegliendo una policy basata solamente sul modello previsto si comporta *come se* il modello fosse corretto. Questa non è necessariamente una buona idea. L'**apprendimento per rinforzo bayesiano** (**beyesian reinforced learning**) assume una probabilità a priori $P(h)$per ogni ipotesi h riguardo a quale sia il vero modello; la proprietà a posteriori $P(h|\mathrm{e})$ è ottenuta nel solito moto tramite la regola di Bayes data dalle osservazioni finora. Poi, se l'agente ha deciso di non apprendere più, la policy ottima è quella che ritorna il valore di utility previsto più alto. Sia ${u_h}^\pi$ l'utility prevista, media su tutti i possibili stati di avvio, ottenuta eseguendo la policy π nel modello h. Allora avremo

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \sum_h P(h|\mathrm{e}){u_h}^\pi$$

In alcuni casi speciali, questa policy può essere anche calcolata. Però, se l'agente continuerà ad apprendere nel futuro, allora trovare una polizza ottima diventa considerabilmente più difficile, poiché la gente deve considerare gli effetti delle future osservazioni sulle sue credenze riguardo il modello di transizione. L'approccio derivato dalla **teoria del controllo robusto** consente un *set* di possibili modelli $\mathcal{H}$ e definisce come policy ottimale robusta quella che restituisce il miglior risultato nel caso peggiore su $\mathcal{H}$:

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \min_h {u_h}^\pi$$

Risolvere il sottostante MDP non è l'unico modo per portare le equazioni di Bellman per influenzare l'algoritmo di apprendimento. Un altro modo è usare le transazioni osservate per aggiustare le utility degli stati osservati in modo che siano in accordo con le condizioni dell'equazione. Quando una transazione avviene da uno stato s ad uno stato s', applichiamo il seguente aggiornamento di $U^\pi(s)$:

$$U^\pi(s) \leftarrow U^\pi(s) + \alpha(R(s) + \gamma\, U^\pi(s') - U^\pi(s))$$

Qui, α è il parametro del learning rate. Dato che questa regola di aggiornamento usa le differenze delle utility tra stadi successivi, è spesso chiamata l'equazione di **differenza dei temporale** (**temporal-difference** equation), o TD. Tutti i metodi di differenza

temporale lavorano aggiustando l'utility prevista tramite l'equilibrio ideale che tiene localmente quando l'utilità prevista è corretta. Notare che gli aggiornamenti comprendono solo i successori s' osservati, mentre la condizione di equilibrio attuale coinvolge tutti i possibili stati successivi. Poiché transizione rare avvengono solo raramente, il valore medio di $U^\pi(s)$ convergerà al valore corretto. Inoltre, se cambiamo α da un valore fisso ha una funzione che decresce come il numero di volte che uno stato viene visitato cresce, allora $U^\pi(s)$ stesso convergerà al valore corretto. L'agente TD non impara velocemente come un agente ADP e mostra maggiore variabilità, ma è molto più semplice e richiede meno calcoli per osservazione. Notare che *TD non richiede un modello di transizione per eseguire gli aggiornamenti*. L' approccio ADP e l'approccio TD sono in realtà connessi. Entrambi provano a fare degli aggiustamenti locali sulle utility previste in modo da rendere ogni stato "in accordo" con i suoi successori. Una differenza è che TD aggiusta uno stato per essere d'accordo con il suo successore *osservato*, mentre ADP aggiusta lo stato per essere in accordo con *tutti* i successori che potrebbero presentarsi, pesati dalle loro probabilità. Una differenza più importante è che mentre TD effettua un singolo aggiustamento per ogni transizione osservata, ADP effettua vari aggiustamenti quanti ne servono per restaurare le consistenze tra le utility previste U e il modello dell'ambiente P. Ogni aggiustamento fatto ADP può essere visto, dal punto di vista di TD, come il risultato una "pseudoesperienza" creata simulando il corrente modello dell'ambiente. È possibile estendere l'approccio TD per usare un modello dell'ambiente per generare varie pseudoesperienze. In questo modo l'utility risultante prevista sarà sempre più vicina a quella di ADP, portando però ad un incremento del tempo di esecuzione. In modo analogo, possiamo generare versioni più efficienti di ADP approssimando direttamente l'algoritmo per l'interazione dei valori o per le iterazioni della policy. Un possibile approccio per generare velocemente risposte ragionevolmente buone è di limitare il numero di aggiustamenti fatti dopo ogni transizione osservata. Si può anche usare un'euristica per classificare i possibili aggiustamenti in modo che vengono effettuati solo quelli più importanti. la **prioritized sweeping** heuristic preferisce fare aggiustamenti a stati i cui successori *probabili* sono stati sottoposti ad un grande aggiustamento sulla loro utility prevista. Usando euristiche come queste, approssimare algoritmi ADP di solito possono imparare più o meno velocemente quanto un full ADP, in termini di numero di sequenze di transizioni, ma può essere più efficiente per vari ordini di magnitudo in termini di computazione.

5.2 Apprendimento per rinforzo attivo

Un agente attivo deve decidere quale azione effettuare. Prima di tutto, l'agente deve imparare un modello completo con tutte le probabilità risultanti per tutte le azioni piuttosto che il modello per policy fissata. Successivamente, dobbiamo tenere in conto il fatto che l'agente ha una scelta di azioni. L'utility che necessita di imparare sono quelle definite dalla policy ottimale:

$$U(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s,a)U(s')$$

Questa equazione può essere risolta ottenendo la funzione di utility U usando gli algoritmi di iterazione dei valori o della policy. Dopo aver ottenuto una funzione di utility U ottimale per il modello appreso, l'agente può estrarre un'azione ottimale effettuando una previsione in avanti di un solo passo per massimizzare l'utility prevista.

5.2.1 Esplorazione

Un agente ADP che segue la raccomandazione della policy ottimale per il modello appreso ad ogni step *non* impara le vere utility o la vera policy ottimale: questo agente viene detto **agente goloso** (**greedy agent**). Ripetuti esperimenti mostrano che gli agenti golosi *molto raramente* convergono alla policy ottimale per questo ambiente e alcune volte convergono a delle policy veramente orrende. Il modello appreso non è lo stesso dell'ambiente reale, ciò che è ottimale nel modello appreso potrebbe essere subottimale nell'ambiente reale. L'agente non conosce come è fatto l'ambiente reale, pertanto non può calcolare l'azione ottimale per l'ambiente reale. Quello che l'agente goloso ha trascurato è che le azioni fanno molto di più che fornire rewards in accordo al modello appreso corrente, contribuiscono anche ad apprendere il modello reale influenzando le percezioni che sono ricevute. Un agente deve fare un compromesso tra lo **sfruttamento** (**exploitation**) per massimare la reward, come riflesso dalle stime correnti dell'utility, e l'**esplorazione** (**exploration**) per massimizzare il benessere a lungo termine. Il puro sfruttamento rischia di bloccarsi in un solco. La pura esplorazione per migliorare la conoscenza non è di aiuto se non viene mai messa in pratica. Uno schema *ragionevole* che porterà eventualmente ad un comportamento ottimale dall'agente deve essere goloso nel limite dell'esplorazione infinita, o **GLIE**. Uno schema GLIE deve provare ogni azione in ogni stato un numero di volte illimitato per evitare di avere una probabilità finita che un'azione ottimale è mancata a causa di una serie inusuale di risultati. Un agente ADP che usa tale scheda apprenderà eventualmente il modello corretto dell'ambiente. Una schema GLIE deve anche eventualmente diventare goloso, in modo che le azioni dell'agente diventano ottimali rispetto al modello appreso. Esistono vari schemi GLIE; uno dei più semplici è di avere che un agente sceglie un'azione casuale in una frazione di tempo $\frac{1}{t}$ e seguire la policy golosa altrimenti. Seppur convergerà eventualmente ad una policy ottimale, può richiedere risultare molto lento. Un approccio molto più sensibile darà alcuni pesi alle azioni che l'agente non ha provato molto spesso, mentre tenderà ad evitare azioni che sono credute essere di poca utilità. Usiamo $U^+(s)$ per indicare la stima ottimistica dell'utility dello stato s, e sia $N(s,a)$ il numero di volte l'azione a è stata provata nello stato s. Supponiamo che stiamo usando un'iterazione del valore in un agente ADP che apprende. La sua equazione di aggiornamento sarà:

$$U^+(s) \leftarrow R(s) + \gamma \max_a f(\sum_{s'} P(s'|s,a)U^+(s'), N(s,a))$$

Dove $f(u,n)$ è detta la **funzione di esplorazione**. Determina quanto la golosità (preferenza per alti valori di u) viene scambiata con la curiosità (preferenza per le azioni che non sono state provate spesso e hanno un valore n basso). La funzione $f(u,n)$ dovrebbe essere crescente in u e decrescente in n. Esistono varie possibili funzioni che soddisfano queste condizioni. Una definizione particolare e molto semplice è

$$f(u,n) = \begin{cases} R^+ & \text{se } n < N_e \\ u & \text{altrimenti} \end{cases}$$

Dove R^+ è una stima ottimistica della miglior reward possibile ottenibile in qualsiasi stato e N_e è un parametro fissato. Questo avrà l'effetto di portare l'agente a provare coppia stato-azione almeno N_e volte. Il fatto che U^+ appare nella parte destra dell'equazione piuttosto che U è molto importante. Mentre l'esplorazione procede, gli stati e le azioni

vicini allo stato iniziale saranno stati provati un gran numero di volte. Se usassimo U, la stima delle utility più pessimista, allora l'agente diventerebbe incline a non esplorare più lontano. L'uso di U^+ significa che che i benefit dell'esplorazione sono propagati all'indietro dai confini delle regioni non esplorate, cosicché le azioni che portano *in direzione* di zone inesplorate hanno pesi maggiori, rispetto alle azioni che sono loro stesso non familiari.

5.2.2 Apprendere un'action-utility function

Un active temporal-difference learning agent non è più equipaggiato con un policy fissata, pertanto, se apprenderà una funzione di utility U necessiterà di apprendere un modello in modo di poter scegliere un'azione in base ad U effettuando una previsione in avanti di un solo passo. Il problema di acquisizione del modello per l'agente TD è identico a quello per l'agente ADP. La regola di aggiornamento del TD rimane invariata. Si può mostrare che l'algoritmo TD convergerà agli stessi valori di ADP se il numero di sequenze di training tende all'infinito. Esiste un metodo TD alternativo, chiamato **Q-learning**, il quale apprende una rappresentazione azione-utility invece di apprendere le utility. Useremo la notazione $Q(s, a)$ per indicare l'effettuazione di un'azione a nello stato s. I Q-values sono direttamente correlati ai valori di utility come segue:

$$U(s) = \max_a Q(s, a)$$

Le Q-function hanno una proprietà davvero interessante: *un agente TD che apprende una Q-function non necessita di un modello nella forma $P(s'|s, a)$, sia per apprendere che per scegliere un'azione.* Per questa ragione, Q-learning è detto modello **model-free**. Per quello che riguarda le utility, possiamo scrivere un'equazione di vincolo che deve mantenere un equilibrio quando i Q-values sono corretti:

$$Q(s, a) = R(s) + \gamma \sum_{s'} P(s'|s, a) \max_a Q(s', a')$$

Come nell'agente ADP, possiamo usare quest'equazione direttamente come un'equazione di aggiornamento per un processo di iterazione che calcola i Q-values esatti, dato un modello stimato. Questo, però, richiede che venga appreso un modello, poiché l'equazione usa $P(s'|s, a)$. Gli approcci a differenza temporale, d'altro canto, non richiedono un modello per le transizioni di stato, poiché necessitano solo dei Q-values. L'equazione di aggiornamento per il TD Q-learning è

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

La quale viene calcolata ogni volta che un'azione a viene eseguita nello stato s portando allo stato s'. Q-learning ha un parente chiamato **SARSA** (per State-Action-Reward-State-Action). La regola di aggiornamento per SARSA è

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma \text{ max } Q(s', a') - Q(s, a))$$

Dove a' è l'azione *effettivamente eseguita* nello stato s'. Questa regola è applicata alla fine di ogni quintupla s, a, r, s', a' (da qui prende il nome). La differenza dal Q-learning è abbastanza sottile: mentre Q-learning propaga all'indietro il *miglior* Q-value dallo stato raggiunto nella transizione osservata, SARSA aspetta finché non viene effettivamente eseguita un'azione e propaga all'indietro il Q-value per quell'azione. Poiché Q-learning usa il miglio Q-value, non presta attenzione alla policy attuale che viene seguita, pertanto

è un algoritmo di apprendimento **off-policy**, mentre SARSA è un algoritmo **on-policy**. Q-learning è molto più flessibile del SARSA, nel senso che un agente Q-learning può imparare come comportarsi bene anche quando è guidato da un policy di eslorazione casuale o avversario. D'altro canto, SARSA è molto più realistico: è meglio imparare una Q-function per quello che succederà effettivamente piuttosto che quello che l'agente vorrebbe che accadesse. Sia Q-learning che SARSA apprendono una policy ottimale, ma lo fanno con una velocità molto più lenta di un agente ADP. Questo perché gli aggiornamenti locali non rafforzano la coerenza tra tutti i Q-values tramite il modello.

5.3 Generalizzazione nell'apprendimento per rinforzo

Abbiamo assunto che le funzioni di utility e le Q-function apprese dagli agenti sono rappresentate in forma tabellare con un output per ogni tupla in input. Tale approccio lavora ragionevolmente bene per spazi di stati molto piccoli, ma il tempo di convergenza e (per gli ADP) il tempo per l'iterazione aumenta rapidamente man mano che lo spazio degli stati aumenta. Un modo per trattare tali problemi è usare l'**approssimazione della funzione**, che significa semplicemente usare un qualsiasi metodo di rappresentazione per la Q-function piuttosto che la ricerca nella tabella. La funzione o Q-function può essere rappresentata nella forma scelta. Una **funzione di valutazione** è rappresentata come una funzione lneare pesata di un set di **features** (o **funzioni basi**) $f_1, f_2, ..., f_n$:

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + ... + \theta_n f_n(s)$$

Un algoritmo di apprendimento per rinforzo può apprendere dei valori per i parametri $\theta = \theta_1, \theta_2, ..., \theta_n$ in modo che la funzione di valutazione $\hat{U}_\theta$ approssima la funzione di utility reale. L'approssimazione della funzione rende pratico rappresentare la funzione di utility per enormi spazi degli stati, ma non è il suo vantaggio principale. *La compressione ottenuta da un approssimatore della funzione permette all'agente che sta apprendendo di generalizzare dagli stati che ha visitato agli stati che non ha visitato.* L'aspetto più importante dell'approssimazione della funzione non è che richiede meno spazio, ma che permette una generalizzazione induttiva sugli stati di input. C'è il problema che potrebbe non esserci alcuna funzione nello spazio delle ipotesi scelto che approssima sufficientemente bene la funzione di utility reale. Come in tutti gli apprendimenti induttivi, esiste un tradeoff tra la grandezza dello spazio delle ipotesi e il tempo necessario ad apprendere la funzione. Uno spazio delle ipotesi molto grande incrementa la possibilità di trovare una buona approssimazione, ma significa anche che molto probabilmente la convergenza sarà ritardata. La stima diretta dell'utility con l'approssimazione della funzione è un'istanza dell'**apprendimento supervisionato**. Data una collezione di prove, otteniamo un set di valori d'esempio di $\hat{U}_\theta(x, y)$, e possiamo trovare il più adatto, nel senso di minimizzare lo squared error, usando la regressione lineare standard. Per l'apprendimento per rinforzo, ha più senso usare un algoritmo di apprendimento *online* che aggiorna i parametri dopo ogni prova. Come con l'apprendimento delle reti neurali, scriviamo un'error function e calcoliamo il suo gradiente rispetto ai parametri. Se $u_j(s)$ è la reward totale osservata dallo stato s in avanti nella j-esima prova, allora l'errore è definito come la metà della differenza al quadrato tra il totale predetto e il totale attuale: $E_j(s) = \dfrac{(\hat{U}_\theta(s) - u_j(s))^2}{2}$. Il

rapporto di cambio dell'errore rispetto ad ogni parametro θ_i è $\frac{\delta E_j}{\delta \theta_i}$, pertanto per muovere il parametro in modo da diminuire l'errore vogliamo

$$\theta_i \leftarrow \theta_i - \alpha \frac{\delta E_j}{\delta \theta_i} = \theta_i + \alpha(u_j(s) - \hat{U}_\theta(s)) \frac{\delta \hat{U}_\theta(s)}{\delta \theta_i}$$

Questa è detta la **regola di Widrof-Hoff**, o **regola delta**, per minimi quadrati online. Notare che *cambiare i parametri θ in risposta ad una transizione osservata tra due stati cambia anche i valori di $\hat{U}_\theta$ per ogni altro stato.* Ci aspettiamo che l'agente apprenda velocemente se usa un approssimatore della funzione, fornito in modo che lo spazio delle ipotesi non sia troppo grande, ma includa alcune funzioni che sono ragionevolmente una buon adattamento della funzione di utility reale. Ciò che importa per l'approssimazione di funzioni lineari è che la funzione sia lineare nei *parametri*, pertanto le feature possono essere arbitrariamente funzioni non lineari per le variabili di stato. Possiamo applicare queste idee altrettanto bene ai temporal-difference learners. Tutto ciò che dobbiamo fare è aggiustare i parametri per provare a ridurre la differenza temporale tra gli stati successivi. La nuova versione delle equazioni di TD e del Q-learning, rispettivamente per le utility e per i Q-values, sono date da

$$\theta_i \leftarrow \theta_i + \alpha[R(s) + \gamma\, \hat{U}_\theta(s') - \hat{U}_\theta(s)] \frac{\delta \hat{U}_\theta(s)}{\delta \theta_i}$$

$$\theta_i \leftarrow \theta_i + \alpha[R(s) + \gamma\, \underset{a'}{max}\, \hat{Q}_{theta}(s', a') - \hat{Q}_\theta(s, a)] \frac{\delta \hat{Q}_\theta(s, a)}{\delta \theta_i}$$

Per l'apprendimento TD passivo, si può mostrare che la regola di aggiornamento converge all'approssimazione più vicina possibile della funzione reale quando l'approssimatore della funzione è *lineare* nei parametri. Con l'apprendimento attivo e funzioni *non lineari* come le reti neurali, tutte le scommesse sono chiuse. Esistono algoritmi più sofisticati che evitano questi problemi, ma tutt'ora l'apprendimento per rinforzo con approssimazione generale delle funzioni rimane un'arte delicata. L'approssimazione delle funzioni può essere molto utile per imparare un modello dell'ambiente. Qualsiasi metodo di apprendimento supervisionato può essere usato, con aggiustamenti adatti per il fatto che necessitiamo di predire la descrizione completa dello stato piuttosto che una classificazione booleana o un singolo valore reale. Per un ambiente *parzialmente osservabile*, il problema di apprendimento è molto più difficile.

5.4 Ricerca della policy

La **ricerca della policy** è il metodo più semplice: l'idea è quella di continuare a modificare la policy finché le performance migliorano, per poi fermarsi. Un policy π è una funzione che associa gli stati alle azioni. Siamo interessati principalmente alle rappresentazioni *parametrizzati* di π chi hanno molti meno parametri rispetto agli stati nello spazio degli stati. Per esempio, possiamo rappresentare π tramite una collezione di Q-functions parametrizzate, una per ogni azione, e prendere a'azione con il valore più alto previsto:

$$\pi(s) = \underset{a}{max}\, \hat{Q}_\theta(s, a)$$

Ogni Q-function può essere una funzione lineare dei parametri θ, o può essere una funzione non lineare come una rete neurale. La ricerca della policy aggiusterà i parametri θ per migliorare la policy/ Notare che se la policy è rappresentata da Q-functions, allora la ricerca della policy diventa un processo che apprende Q-functions. *Questo processo non è lo stesso del Q-learning* Nel Q-learning con l'apporssimazione delle funzioni, l'algoritmo trova un valore di θ tale che $\hat{Q}_\theta$ è "vicino" a Q^*, la Q-function ottimale. La ricerca della policy, invece, trova un valore di θ che porta ad una buona performance; i valori trovati dai due metodi possono differenziare di molto. Un problema con la rappresentazione delle policy del tipo dato sopra è che la policy è una funzione *discontinua* sui parametri quando le azioni sono discrete. Pertanto, ci saranno valori di θ tale che un cambio infinitesimale in θ che portano la policy scegliere un'azione invece che un'altra. Questo significa che il valore delle policy potrebbe cambiare discontinuamente, il che rende la ricerca basata sul gradiente difficile. Per questa ragione, i metodi di ricerca della policy usano spesso la rappresentazione di una **policy stocastica** $\pi_\theta(s, a)$, che specifica la *probabilità* di selezionare un'azione a nello stato s. Una rappresentazione popolare è la **softmax function**:

$$\pi_\theta(s, a) = \frac{e^{Q_\theta(\hat{s},a)}}{\sum_{a'} e^{Q_\theta(\hat{s},a')}}$$

Softmax diventa quasi deterministica se una delle azioni è migliore rispetto alle altre, ma darà sempre una funzione differenziabile in θ; pertanto, il valore della policy è una funzione differenziabile in θ. Partiamo dal caso semplice: una policy deterministica e un ambiente deterministico. Sia $\rho(\theta)$ il **valore della policy**, per esempio la reward-to-go prevista quando π_θ viene eseguita. Se possiamo derivare un'espressione per $\rho(\theta)$ in forma chiusa, allora abbiamo un problema di ottimizzazione standard. Possiamo seguire il vettore del **gradiente della policy** $\nabla_{\theta\rho}(\theta)$ purché $\rho(\theta)$ è differenziabile. Se $\rho(\theta)$ non è disponibile in forma chiusa, possiamo valutare π_θ semplicemente eseguendola e osservano la reward accumulata. Possiamo seguire il **gradiente empirico** tramite hill climbing. Quando l'ambiente (o la policy) è stocastica, le cose si fanno più complesse. Supponiamo che stiamo provando ad effettuare hill climbing, che richiede la comparazione tra $\rho(\theta)$ e $\rho(\theta + \Delta\theta)$ per un qualche $\Delta\theta$ piccolo. La reward totale per ogni prova può variare enormemente, pertanto le stime dei valori della policy ottenuta da un piccolo numero di prove sarà abbastanza inaffidabile; provare a comparare due di queste stime sarà ancora di più inaffidabile. Una soluzione è semplicemente quella di eseguire una marea di prove, misurando la varianza degli esempi e usandola per determinare che sono state eseguite abbastanza prove per ottenere un'indicazione della direzione di miglioramento di $\rho(\theta)$. Sfortunatamente, ciò non è pratico per molti problemi reali, dove ogni prova può essere costosa, lunga, e in alcuni casi pericolosa. Per il caso di una policy stocastica $\pi_\theta(s, a)$, è possibile ottenere una stima imparziale del gradiente in θ, $\nabla_{\theta\rho}(\theta)$, direttamente dai risultati delle prove eseguite in θ. Per semplicità, deriveremo la stima per il caso semplice di un ambiente non sequenziale dove la reward $R(a)$ è ottenuta immediatamente dopo aver effettuato l'azione a nello stato s_0. In questo caso, il valore della policy è solo il valore previsto della reward, e abbiamo

$$\nabla_{\theta\rho}(\theta) = \nabla_\theta \sum_a \pi_\theta(s_0, a)R(a) = \sum_a (\nabla_\theta \pi_\theta(s_0, a))R(a)$$

Ora effettuiamo un semplice trucco in modo che la somma può essere approssimata dei campioni generati dalla distribuzione probabilistica definita da $\pi_\theta(s_0, a)$. Supponiamo di avere N prove in tutto e e l'azione effettuata nella j-esima prova è a_j. Allora

$$\nabla_{\theta\rho}(\theta) = \sum_a \pi_\theta(s_0, a) \cdot \frac{(\nabla_\theta \pi_\theta(s_0, a))R(a)}{\pi_\theta(s_0, a)} \approx \frac{1}{N} \sum_{j=1}^{N} \frac{(\nabla_\theta \pi_\theta(s_0, a_j))R(a)_j}{\pi_\theta(s_0, a_j)}$$

Pertanto, il gradiente reale del valore di policy è approssimato dalla dei termini coinvolgendo il gradiente della della probabilità di selezione dell'azione in ogni prova. Per il caso sequenziale, questo generalizza per ogni stato s visitato:

$$\nabla_{\theta\rho}(\theta) \approx \frac{1}{N} \sum_{j=1}^{N} \frac{(\nabla_\theta \pi_\theta(s, a_j))R_j(a)_j}{\pi_\theta(s, a_j)}$$

Dove a_j è eseguita in s nella j-esima prova e $R_j(s)$ è la reward totale ricevuta dallo stato s in avanti nella j-esima prova. L'algoritmo risultante è chiamato REINFORCE; è usualmente molto più efficace rispetto ad hill climbing usando molte prove per ogni valore di θ. Tuttavia, è molto più lento del necessario. Il **campionamento correlato** (**correlated sampling**) è alla base di un algoritmo di ricerca della policy chiamato PEGASUS. L'algoritmo è applicabile in dini in cui è disponibile un simulatore in modo che i risultati "casuali" delle azioni possono essere ripetuti. Lálgoritmo lavora generando in anticipo N sequenze di numeri casuali, ognuna delle quali può essere usata per effettuare una prova di una policy qualsiasi. La ricerca della policy viene effettuata valutando ogni policy candidata usando lo stesso *set* di sequenza casuali per determinare i risultati dell'azione. Il numero di sequenze casuali richiesto per assicurare che il valore di *ogni* policy sia ben stimato dipende solo dalla complessità dello spazio della policy.