Appunti del Corso di Machine Learning e Deep Learning

Professore del corso:
Prof. Simone Calderara

Appunti di:
Lorenzo Pioli

Anno Accademico 2024/2025

Sommario

Di seguito riporto i miei appunti del corso di *Machine Learning and Deep Learning* tenuto dal Professor Simone Calderara nell'anno accademico 2024/2025.

Indice

Capitolo 1

Supervised Learning Theory

Nel Supervised Learning si parla di due oggetti:

- **Dati**: sono rappresentati da un elemento in uno spazio k-dimensionale $x \in X^k$. In uno spazio k-dimensionale, l'elemento si descrive con k numeri. Questi k numeri si chiamano attributi o features. Dunque l'elemento x appartenente allo spazio k-dimensionale X^k è: $x = \{x_1, \ldots, x_k\}$. L'insieme dei dati è costituito da questi datapoint e da un oggetto che si chiama **classe**. La classe è un numero associato ad ogni record ed è una categoria definita a priori. Si parla di dataset annotato quando l'insieme di dati è costituito dalle features e dalle classi e contiene la risposta che vogliamo ottenere.

- **Obiettivo**: l'obiettivo è quello di determinare il modello di apprendimento dai dati, che possono essere usati, per predire le classi di nuovi casi.

1.1 Fasi del Supervised Learning

Il Supervised Learning è un processo a due fasi:

1. Training
2. Testing

1.1.1 Training

Durante la fase di Training avviene l'addestramento del modello. Il modello impara dai dati e dalle classi a compiere il task. La fase di training agisce sui parametri del modello. Il modello si può vedere come una funzione che ha degli input e dei parametri.

Pipeline di Training

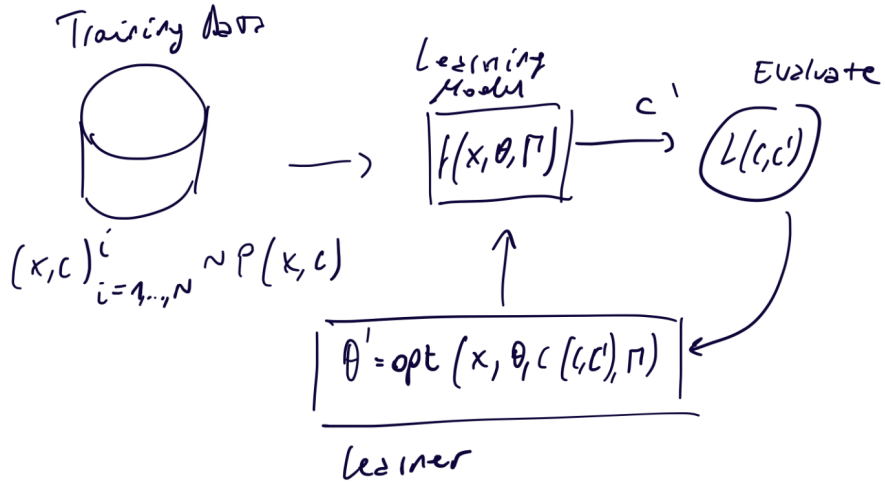


Figura 1.1: Pipeline di Training

Il processo di training è un processo iterativo in cui si cerca di migliorare il modello. La pipeline di training è costituita da quattro elementi:

- **Training Data**: i dati di training sono coppie di vettori di feature x e classi c campionati da una distribuzione $P(x, c)$, rappresentante una certo fenomeno, che non conosciamo. Questi dati vengono forniti ad una determinata funzione che è il modello di learning.

- **Learning Model**: il modello di learning prende in input x, cioè i dati di training, e contiene due set di parametri: Φ e Γ. Questi parametri vengono usati per produrre una possibile classe c'.

- **Valutatore**: il valutatore prende la possibile classe c' fornita dal modello e la confronta con la classe vera c tramite la funzione di Loss $L(c, c')$. Questa funzione indica quanto abbiamo fatto bene/male e fornisce questa informazione all'algoritmo di ottimizzazione.

- **Learner**: l'algoritmo di ottimizzazione ha come obiettivo quello di cambiare i parametri del modello per cercare di farlo andare meglio. Questi nuovi parametri vengono riforniti al modello.

Questo processo viene iterato fino a quando le performance non migliorano più.

1.1.2 Testing

Si fa sempre sui dati che non sono stati presentati al modello durante il training. In questa fase viene misurata l'accuratezza del modello.

L'accuratezza è una misura della performance del modello ed è data da:

$$\text{Acc} = \frac{\text{numero di classificazioni corrette}}{\text{numero totale dei casi}}$$

L'accuratezza non è l'unica misura delle performance. La scelta della misura da utilizzare dipende dal problema: ad esempio l'accuratezza non è una buona scelta quando dobbiamo classificare eventi rari.

L'apprendimento automatico, o machine learning, è sempre dedicato ad apprendere qualcosa e a valutarne le performance.

Pipeline di Testing o di Inference

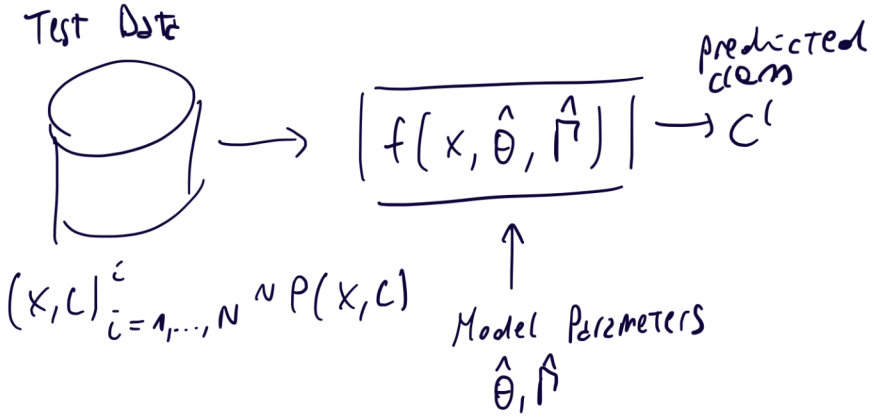


Figura 1.2: Pipeline di Testing

Anche in questo caso si forniscono in input dati annotati. Questa volta, però, non servono per imparare a risolvere un problema ma per valuatare se la predizione del modello è corretta oppure no. La funzione utilizza come parametri gli utimi parametri imparati dal training, quindi i migliori possibili. Quindi la fase di inference non prevede la parte di ottimizzazione ma solo quella di valutazione.

1.2 Learning

1.2.1 Definizione di Learning

L'apprendimento o learning coinvolge tre elementi:

- Dataset D
- Task C
- Misura di Performance L

Da un punto di vista formale si dice che:

Un sistema impara una task C da un dataset D se dopo aver osservato il dataset di training le performance su L migliorano ripetto a quelle che si hanno senza osservare nessun caso.

1.2.2 i.i.d Assumption

Il learning è possibile, se e solo se, la distribuzione da cui provengono i dati di tarining e i dati di test è la stessa $x \sim P(X)$. In tal caso, si dice che i campioni sono indipendentemente e identicamente distribuiti (i.i.d) da training e testing.

Supponiamo di avere una funzione di classificazione $f \in F$, una task $c \in C$ e D è la distribuzione dei dati. Si definisce **errore di generalizzazione**, o **rischio di un classificatore**, il valore medio di tutte le volte che la funzione di classificazione sbaglia a predire la classe.

$$R(f) = P_{x \in D}\big[f(x) \neq c(x)\big] = \mathbb{E}_{x \in D}\big[I_{h(x) \neq c(x)}\big]$$

Questa quantità non si può misurare perchè non conosciamo D. Ciò che si può misurare invece è il **rischio empirico**. Invece di prendere gli elementi da D, si prendono da S, il **sample set**, ossia un insieme di oggetti campionati da D. $S = (x_1, \ldots, x_m)$.

$$\widehat{R_S}(f) = \frac{1}{m}\sum_{i=1}^{m} I_{h(xi)\neq c(xi)}$$

Ciò che si vuole è che l'errore empirico sia il più vicino possibile a quello teorico. La differenza tra errore teorico e errore empirico è detto **gap di generalizzazione**, ossia quanto si perde in termini di performance a causa della limitatezza del set di dati reale.

Si ha che $R(f) = \widehat{R_S}(f) \iff S$ è campionato i.i.d da D.

Dimostrazione

Se S è campionato i.i.d da D, allora:

$$\mathbb{E}_{S\sim D^m}[\widehat{R_S}(f)] = \frac{1}{m}\sum_{i=1}^{m} \mathbb{E}_{S\sim D^m}[I_{h(xi)\neq c(xi)}] = \mathbb{E}_{S\sim D^m}[I_{h(x)\neq c(x)}]$$

Usando il fatto che i campioni sono i.i.d, il valor medio assume lo stesso valore per ogni S. Dunque:

$$\mathbb{E}_{S\sim D^m}[\widehat{R_S}(f)] = \mathbb{E}_{S\sim D^m}[I_{h(x)\neq c(x)}] = \mathbb{E}_{S\sim D}[I_{h(x)\neq c(x)}] = R_f$$

Questa cosa non è sempre vera perchè non sempre è possibile campionare in maniera uniforme da D.

1.3 Misure di Performance

La valutazione delle performance viene effettuata sui dati di test, altrimenti i risultati sono falsati. Le misure delle performance sono le seguenti e vengono applicate a seconda del problema in esame:

1. Accuracy
2. Efficienza
3. Robustezza
4. Scalabilità
5. Interpretabilità
6. Compattezza del modello

1.4 Iperparametri

Un modello ha anche dei parametri che non vengono imparati ma che vengono messi a mano. Questi sono detti iperparametri. Gli iperparametri, dunque, non sono imparati nè sul training nè sul testing. Quindi si ha bisogno di un nuovo set di dati, detto **Validation set**, da tenere da parte.

1.5 Train-Validation-Test

Dato un dataset D lo si partiziona in tre parti:

1. **Training set** (Tr)
2. **Validation set** (V)
3. **Test set** (Te)

Le percentuali possono essere (60,20,20) oppure (80,10,10).

Il modello di machine learning viene addestrato sul Tr per ogni set di iperparametri che sto valutando (a forza bruta). Poi misuro le performance sul V tengo il set di iperparametri che dà le performance migliori. A questo punto congelo gli iperparametri, uso quelli, e riaddestro tutto su Tr+V e ottengo il modello finale. A questo punto il modello finale lo valuto sul test Te.

Per fare in modo che l'assunzione i.i.d sia verificata si dovrebbero mischiare i dati nel dataset in modo casuale. Quindi l'ordine non deve avere importanza.

Se il dataset non è abbastanza grande da spezzrlo in tre allora si può fare la **K-fold Crossvalidation**.

1.5.1 K-fold Crossvalidation

Prendo il dataset D e lo spezzo in K blocchi $B_1, \ldots, B_K$ in modo casuale. Per ogni $k = 1, \ldots, K$, si fa il training su $D - B_k$ e il test su B_k. Questo processo viene ripetututo k volte. Alla fine si hanno K accuracy che mediate danno l'accuracy del modello. Il k-fold crossvalidation può anche essere usato per scegliere gli iperparametri.

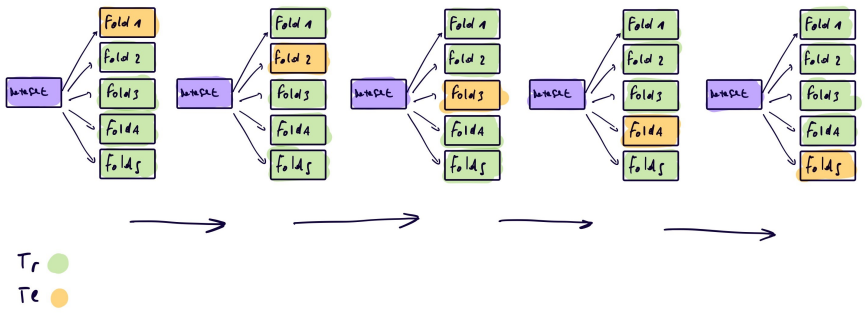


Figura 1.3: K-Fold Crossvalidation

1.6 Performance

Quando voglio classificare un evento raro, l'accuracy non è un parametro valido.

1.6.1 Precision e Recall

La classe di interesse viene chaiamata **classe positiva** e il resto **classi negative**. Prima di introdurre i concetti di precision e recall bisogna introdurre tutti i possibili casi che si possono verificare quando si vuole classificare un evento raro e si ha un concetto di classe positiva e negativa.

- **True Positive**: se prevedo correttamente la classe positiva. (Dico che c'è incendio e c'è)
- **False Positive**: se il classificatore di learning dice che la classe è positiva ma è falso. (Dico che c'è incendio ma non c'è)
- **False Negative**: se il classificatore di learning dice che la classe è negativa ma è falso. (Dico che non c'è incendio ma in realtà c'è)
- **True Negative**: se prevedo correttamente la classe negativa. (Dico che non c'è incendio e non c'è)

Su queste misure che si possono distribuire in una matrice, detta **matrice di confusione**, si possono definire precision e recall.

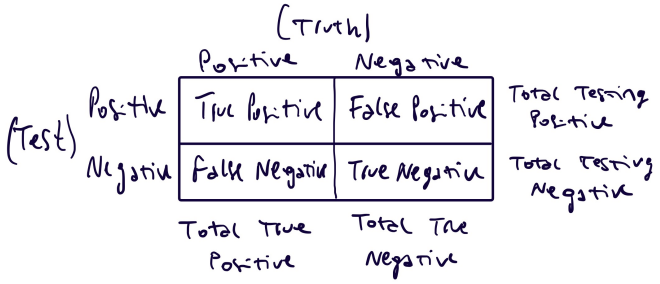


Figura 1.4: Matrice di Confusione

Definizione di Precision

È il numero di veri positivi, cioè il numero di esempi positivi classificati correttamente, diviso per il numero di esempi classificati come positivi.

$$p = \frac{TP}{TP + FP}$$

Definizione di Recall

È una misura di copertura. È il numero di veri positivi diviso il numero di esempi positivi reali.

$$r = \frac{TP}{TP + FN}$$

Definizione di F1-Score

È la media armonica di precision e recall, quindi è più vicina al più basso tra i due.

$$\frac{1}{F_1} = \frac{1}{p} + \frac{1}{r}$$

Curva ROC

Precision e recall normalmente formano una curva, detta **curva ROC**, che misura la loro correlazione. Viene usata tipicamente quando l'output del sistema è uno score. Per ottenere una categoria da uno score abbiamo bisogno di un threashold. Variando il threshhold variano p ed r. Notare che abbassando uno si alza l'altro e viceversa.

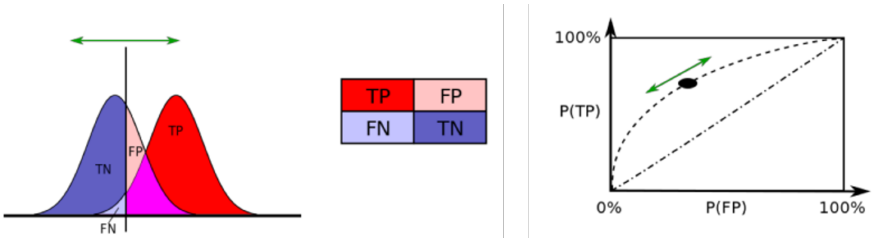


Figura 1.5: Curva ROC

1.6.2 Performance con Output Statistici

Se l'output è una misura di probabilità o distribuzione, le performance sono valutate:

1. utilizzando threshold di probabilità e usando classi discrete.

2. usando la distanza tra distribuzioni.

Misure di distribuzione tra $P(x)$ e $Q(x)$:

1. **Bhattacharrya Coefficient**:

$$BC(P, Q) = \int_x P(x)Q(x)\, dx$$

   Se le distribuzioni sono discrete si usano le sommatorie. Questo coefficiente è un coefficiente di similarietà ed indica quanto le distribuzioni si sovrappongono.

2. **Kubac Liver Divergence**

$$KL(P|Q) = \int_x P(x) \log \frac{P(x)}{Q(x)}\, dx$$

   Misura la perdita di informazione che si verifica quando si usa Q invece che P.

1.6.3 Cross Entropy

È costruita sul concetto di codifica di un messaggio. Valuta il numero medio di bit per scoprire un determinato valore codificato tramite una distribuzione q, quando la distribuzione di origine è p. In sostanza dice quanti bit mi servono per scoprire che dato è, se sto sbagliando la distribuzione da cui quel dato deve provenire, ossia se sto usando un'approssimazione.

$$H(P, Q) = H(P) + KL(P|Q) = -\int_x P(x) \log Q(x)\, dx$$

Dimostrazione I

$$H(P,Q) = H(P) + KL(P|Q)$$

Dalla definizione di Entropia si ha che:

$$H(P) = -\int P(x)log(P(x))\,dx$$

dunque:

$$H(P,Q) = H(P) + KL(P|Q) = -\int (x)log(P(x))\,dx + \int P(x)\log\frac{P(x)}{Q(x)}\,dx$$

dalle proprietà dei logaritmi:

$$\int P(x)\log\frac{P(x)}{Q(x)}\,dx = \int P(x)\log P(X)\,dx - \int P(x)\log Q(X)\,dx$$

Semplificando l'espressione si ottiene infine:

$$H(P,Q) = H(P) + KL(P|Q) = -\int_x P(x)\log Q(x)\,dx$$

Dimostrazione II

Questa dimostrazione fa uso del teorema di Kraft-McMillan. Il teorema enuncia che un valore x_i può essere identificato da l_i bit con probabilità $Q(x_i) = 2^{-l_i}$.

Facendo il valore atteso di l rispetto a $P(x)$ si ottiene:

$$\mathbb{E}_P[l] = \sum_x \log_2\frac{1}{Q(x)} = -\sum_x \log_2 Q(x)$$

Altri utilizzi per la Cross Entropy

La Cross Entropy può essere usata anche come misura di errori relativi alla classificazione:

Si pongono

$$P_i(x) = \begin{cases} 1 & \text{se } x \in \text{classe } i, \\ 0 & \text{altrimenti.} \end{cases}$$

$Q_i(x) =$ score di probabilità che il classificatore attribuisce alla classe i per l'elemento x

Il valore atteso della Cross Entropy discreta sull'intero dataset D di N elementi equiprobabili è:

$$\mathbb{E}[H(P,Q)] = \frac{1}{N} \sum_{x \in D} \left(- \sum_{i \in K} P_i(x) \log Q_i(x) \right)$$

Nel caso binario, ossia quando sono possibili solo due classi, si ha la **binary Cross Entropy**.

$$\frac{1}{N} \sum_{x \in D} \left(- \sum_{i \in \{0,1\}} P_i(x) \log Q_i(x) \right) = -\frac{1}{N} \sum_{x \in D} [P_0(x) \log Q_0(x) + (1 - P_0(x)) \log(1 - Q_0(x))]$$

Dimostrazione

$$\frac{1}{N} \sum_{x \in D} \left(- \sum_{i \in \{0,1\}} P_i(x) \log Q_i(x) \right) = -\frac{1}{N} \sum_{x \in D} [P_0(x) \log Q_0(x) + P_1(x) \log Q_1(x)] =$$

$$-\frac{1}{N} \sum_{x \in D} [P_0(x) \log Q_0(x) + (1 - P_0(x)) \log(1 - Q_0(x))]$$

Capitolo 2

Classificazione

Definizione Task di Classificazione

Dato un vettore $x \in \mathbb{R}^{\mathbb{D}}$, detto vettore di features, appartenente ad una delle $C \in Y$, vogliamo predire a quale classe appartiene x.

Definizione Processo di Apprendimento

È il processo mediante il quale si impara una funzione che, una volta fornito x in ingresso, produce in output la classe.

Errore del Classificatore

Quante volte il classificatore sbaglia a predire le classi.

$$Err(f, D) = \frac{1}{N} \sum_{i=1}^{N} I(y_i \neq f(x_i))$$

Accuracy

Quante volte il classificatore predice le classi correttamente.

$$Acc(f, D) = \frac{1}{N} \sum_{i=1}^{N} I(y_i = f(x_i))$$

Ipotizzando che f sia una probabilità, il classificatore diventa un classificatore probabilistico.

2.1 Classificatore Probabilistico

Un classificatore probabilistico è un oggetto in cui la funzione che attribuisce una classe ad un elemento x dato in ingresso è una probabilità.

Supponiamo di conoscere la **probabilità a priori**, ossia la probabilità di vedere un elemento che appartiene ad una certa classe c sia $P(Y = c) = \pi_c$. Si considera, inoltre, la densità di probabilità di vedere un determinato vettore $x \in \mathbb{R}^{\mathbb{D}}$ che appartiene alla classe c, $p(X = x|Y = c) = \phi_c(x)$. Quest'ultima equazione è la **likelihood** e misura, una volta fissata la classe, quanto quello che sto guardando è coerente con la classe.

Gli elementi chiave del classificatore probabilistico sono dunque:

1. Probabilità a priori: $P(Y = c) = \pi_c$

2. Likelihood $p(X = x|Y = c) = \phi_c(x)$

Date queste quantità, che supponiamo di conoscere correttamente, siamo in grado di determinare la probabilità della classe dato x utilizzando le regole di Bayes.

2.1.1 Regola di Bayes

$$P(Y = c \mid X = x) = \frac{P(X = x \mid Y = c)P(Y = c)}{\sum_{c' \in Y} P(X = x \mid Y = c')P(Y = c')} = \frac{\phi_c(x)\pi_c}{\sum_{c' \in Y} \phi_{c'}(x)\pi_{c'}}$$

Definizione Likelihood

Misura quanto è facile osservare x data una classe c.

$$P(X = x|Y = c)$$

Definizione Probabilità a Priori

Senza tener conto di x misura quanto è probabile osservare c.

$$P(Y = c)$$

Definizione Probabilità a Posteriori

È la probabilità di osservare c dato x. Si ottiene tramite la regola di Bayes.

$$P(Y = c|X = x)$$

Osservazioni sulla regola di Bayes

La likelihhod e la probabilità a priori sono alla ricerca continua di un trade-off tra loro. Supponiamo di fissare la classe c, ad esempio $c = 1$. Supponiamo che x assomigli molto agli elementi della classe c. Ciò implica che $P(Y = c|X = x)$ sia molto alta. Nel caso in cui la classe c sia molto improbabile, il prodotto sarà più basso della likelihood. Quindi il prodotto penalizza le classi poco probabili.

2.2 Strategie per Classificare un Elemento

Supponiamo di avere un dataset $D = (x_i, y_i)_{i=1,\dots,N}$ con k possibili classi c_i, $i = 1, \dots, k$ e un singolo elemento di test $\hat{x}$.

Si hanno due possibilità:

- **Classificatore a Maximum Likelihood**:

  1. Da D imparo una funzione di likelihood $P(x|c_i)$ per ogni classe.
  2. Prendo $\hat{x}$ e lo passo a queste k funzioni di likelihood.
  3. $\hat{c} = \arg\max_c P(\hat{x}|c)$

- **Classificatore a Maximum a-Posteriori**:

  generalmente è più veloce rispetto all'altro.

  1. Da D imparo una funzione di likelihood $P(x|c_i)$ per ogni classe.
  2. Da D imparo anche la probabilità a priori per ogni classe.
  3. Apllico la regola di Bayes dato $\hat{x}$, $P(c_i|x)$.
  4. $\hat{c} = \arg\max_c P(c|\hat{x})$

2.3 Classificatore Bayesiano Ottimo

Questo classificatore utilizza la funzione di classificazione:

$$f_B(x) = \arg\max_{c \in Y} P(Y = c \mid X = x) = \arg\max_{c \in Y} \phi_c(x)\pi_c$$

Quindi è un classificatore a maximum a-posteriori.

Questo classificatore raggiunge il minimo errore possibile tra tutti i classificatori.

$$Acc = 1 - \mathbb{E}_{P(X=x)} \left[\max_{c \in Y} P(Y = c \mid X = x) \right]$$

Non è però utilizzabile nella pratica perchè dalle funzioni imparate.

2.3.1 Come imparare la Likelihood

Devo poter stimare una densità di probabilità a partire da un dataset. Si tenta sempre di stimare la **joint probability**, o probabilità congiunta, $P(X = x, Y = c)$, ossia la probabilità che $X = x$ ed $Y = c$.

La differenza tra $P(X = x|Y = c)$ e $P(X = x, Y = c)$ è che nel primo caso l'evento $Y = c$ è già accaduto, mentre nel secondo i due eventi si verificano contemporaneamente.

Regola del prodotto

La probabilità congiunta è data da:

$$P(x, y) = P(x|y)p(y)$$

Da cui si deriva la likelihood:

$$P(x|y) = \frac{P(x, y)}{P(y)}$$

Per determinare la probabilità a posteriori invece si utilizza la regola di Bayes:

$$P(y|x) = \frac{P(x|y)P(y)}{P(x)} = \frac{P(x, y)}{P(x)}$$

Quindi se avessi la probabilità congiunta potrei calcolare quello che voglio. Però è difficile da trovare e spesso causa **overfitting**. Siccome il dataset non rappresenta il fenomeno ma un'osservazione di quel fenomeno, non copre tutti i possibili casi, quindi alcune combinazioni (x, y) potrebbero non apparire mai nel dataset producento una probabilità 0. Una soluzione a questo problema può essere quella di stimare la probabilità $P(x|c)$

2.3.2 Come stimare una Densità di Probabilità

Ci sono due soluzioni che dipendono dal tipo di dato che sto affrontando.

1. **Parametric Maximum Likelihood Density Estimation**: Si utilizza se x è continuo. Battezzo una certa funzione $p(x|\Phi)$ dove Φ è un set di parametri. La funzione è nota (es. Gaussiana). Poi devo imparare sul dataset quali sono i parametri che massimizzano al meglio i miei dati. Massimizzo la likelihood, quindi cerco il set $\hat{\Phi}$ di parametri tale che:

$$\hat{\Phi} = \arg\max \Phi \prod_{x_i \in D} p(x_i \mid \Phi) = \arg\max \Phi \sum_{x_i \in D} \log(p(x_i \mid \Phi))$$

   Si può notare che la likelihood su un dataset è il prodotto della likelihood di ogni singolo elemento del dataset.

2. **Stima non parametrica**: si utilizza nel caso x discreto. Si costruisce un istogramma $H(x)$ contando il numero di valori che assume la variabile x. Si normalizza l'istogramma per ottenere la probabilità.

Differenze tra tra densità parametrica e non parametrica

Nel momento in cui stimo una densità di probabilità con un metodo parametrico, mi aspetto che i dati seguano la funzione che ho scelto quindi faccio un'approssimazione. In generale, invece, un modello non parametrico è sempre "giusto" perchè non faccio nessuna assunzione su una funzione in particolare. A volte si preferisce una stima parametrica perchè nell'istogramma c'è un rischio di overfitting e di discretizzazione.

Prova da solo

Dato un dataset D composto da N elementi, l'equazione della distribuzione parametrica della Gaussiana 1D è $p(x|\mu, \sigma^2)$.

$$p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{\sigma^2}}$$

La Log-likelihhod $L(\mu, \sigma^2)$ si ottiene come:

$$L(\mu, \sigma^2) = \log \prod_{x_i \in D} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{\sigma^2}} =$$

$$= \sum_{x_i \in D} \log \left(\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x_i-\mu)^2}{\sigma^2}} \right) =$$

$$= \sum_{x_i \in D} \log \frac{1}{\sqrt{2\pi\sigma^2}} + \log e^{-\frac{(x_i-\mu)^2}{\sigma^2}}$$

Si passa alle derivate:

$$\frac{\partial L}{\partial \mu} = 0$$

$$\sum_{x_i \in D} \log e^{-\frac{(x_i-\mu)^2}{\sigma^2}} = \sum_{x_i \in D} -\frac{(x_i - \mu)^2}{\sigma^2}$$

$$\sum_{x_i \in D} \frac{2(x_i - \mu)}{\sigma^2} = 0$$

$$\frac{1}{\sigma^2} \sum_{x_i \in D} 2(x_i - \mu) = 0$$

$$\sum_{x_i \in D} (x_i - \mu) = \sum_{x_i \in D} (x_i) - \sum_{x_i \in D} (\mu) = 0$$

$$\sum_{x_i \in D} (x_i) = \sum_{x_i \in D} (\mu)$$

Ricordando che D è composto da N elementi:

$$\sum_{x_i \in D} (x_i) = N(\mu)$$

$$\mu = \frac{1}{N} \sum_{i=1}^{N} (x_i)$$

Che è la media.

Ora si deve fare la stessa cosa per σ^2

$$\sum_{x_i \in D} \log \frac{1}{\sqrt{2\pi\sigma^2}} + \log e^{-\frac{(x_i-\mu)^2}{\sigma^2}}$$

$$\frac{\partial L}{\partial \sigma^2} = 0$$

$$\sum_{x_i \in D} \log e^{-\frac{(x_i-\mu)^2}{\sigma^2}} = \sum_{x_i \in D} -\frac{(x_i-\mu)^2}{\sigma^2}$$

$$\frac{\partial}{\partial \sigma^2}\left(\log \frac{1}{\sqrt{2\pi\sigma^2}}\right) = -\frac{1}{\sigma^2}$$

$$\frac{\partial}{\partial \sigma^2}\left(-\frac{(x_i-\mu)^2}{\sigma^2}\right) = \frac{(x_i-\mu)^2}{\sigma^4}$$

$$\sum_{x_i \in D} (x_i-\mu)^2 = \sum_{x_i \in D} \sigma^2$$

Ricordando che D è composto da N elementi:

$$\sum_{x_i \in D} (x_i-\mu)^2 = \sum_{x_i \in D} \sigma^2 = N\sigma^2$$

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N} (x_i-\mu)^2$$

Stimatore di Densità Naive

È uno stimatore banale. Permette di gestire l'overfitting.

Se ho k variabili aleatorie, questo stimatore considera ogni variabile come indipendente dalle altre.

$$P(x_1, x_2, \ldots, x_k|c) = \prod_{i=1}^{k} P(x_i|c)$$

In questo modo non dipendo più dalle possibili configurazioni delle coppie ma dalle configurazioni del singolo.

2.4 Naive Bayes

Approssima il classificatore Bayesiano ottimo considerando ogni attributo del vettore come indipendente dagli altri, effettuanda una Naive DE Assumption.

$$\phi_c(x) = p(X = x|Y = c) = \prod_{d=1}^{k} p(X_d = x_d|Y = c) = \prod_{d=1}^{k} \phi_{cd}(x_d)$$

La forma generale della funzione di classificazione è:

$$f_{NB}(x) = \arg\max_{c \in Y} \pi_c \prod_{d=1}^{k} \phi_{cd}(x_d)$$

2.4.1 Class Conditional Distribution

Le funzioni $p(X_d = x_d|Y = c) = \phi_{cd}(x_d)$ sono dette marginal class conditional distribution.

- Per valori reali di x_d, $p(X_d = x_d|Y = c)$ è tipicamente modellato come una normale:

$$\phi_{cd}(x_d) = N(x_d; \mu_{dc}; \sigma^2_{dc}) = \frac{1}{\sqrt{2\pi\sigma^2_{dc}}} e^{-\frac{(x_d - \mu_{dc})^2}{2\sigma^2_{dc}}}$$

- Per valori binari di x_d si usa la distribuzione di Bernoulli:

$$\phi_{cd}(x_d) = \theta_{dc}^{x_d}(1 - \theta_{dc})^{(1-x_d)}$$

- Per valori generali categorici x_d, si utilizza una distribuzione categorica:

$$\phi_{cd}(x_d) = \prod_{v \in x_d} \theta_{vdc}^{x_d = v}$$

2.4.2 Learning per il Naive Bayes

π_c e $\phi_{cd}(x_d)$ vengono imparate per maximum likelihood su dati di training $D = (x_i, y_i), i = 1, \ldots, n$.

Si calcola $\pi_c = \frac{1}{N} \sum_{i=1}^{N} [y_i = c]$ e i parametri in base alle funzioni di likelihood scelte.

2.4.3 Interpretazione Geometrica

Il classificatore divide lo spazio in k partizioni, una partizione per ogni classe.

Supponiamo:

$x \in \mathbb{R}^2$ $\bar{x} = (x_1, x_2)$,

$c_0, c1$,

$p(\bar{x}|c_i) = \prod_{i=1}^{2} p(x_i|c_i)$

Dato che ci sono due classi, il classificatore dividerà lo spazio in due regioni.

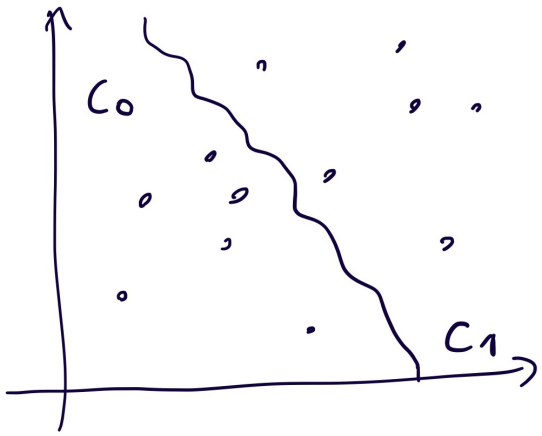


Figura 2.1: Interpretazione geometrica del classificatore

Per trovare la geometria di questo oggetto devo determinare il luogo dei punti per cui la funzione della classe c_1 è uguale a quella della classe c_0.

$$f_{NB}(x) = \arg\max_{c \in Y} \pi_c \prod_{d=1}^{k} \phi_{cd}(x_d) =$$

$$= \arg\max_{c \in Y} \log \pi_c + \sum_{d=1}^{k} \log \phi_{cd}(x_d) =$$

Supponendo di usare una distribuzione gaussiana normale per le marginal class conditional distribution si ottiene:

$$= \arg\max_{c \in Y} \log(\pi_c) + \sum_{d=1}^{D} \left[-\frac{1}{2} \log(2\pi\sigma_{dc}^2) - \frac{1}{2\sigma_{dc}^2}(x_d - \mu_{dc})^2 \right]$$

Il **decision boundary** è costituito dai punti tali che:

$$(I) = \log(\pi_c 0) + \sum_{d=1}^{2} \left[-\frac{1}{2} \log(2\pi\sigma_{d0}^2) - \frac{1}{2\sigma_{d0}^2}(x_d - \mu_{d0})^2 \right]$$
$$(II) = \log(\pi_c 1) + \sum_{d=1}^{2} \left[-\frac{1}{2} \log(2\pi\sigma_{d1}^2) - \frac{1}{2\sigma_{d1}^2}(x_d - \mu_{d1})^2 \right]$$
$$(I) = (II)$$

Si può notare che il decision boundary è una funzione quadrica con forma:

$$\sum_{d=1}^{D} (a_d x_d^2 + b_d x_d) + c = 0$$

Nel caso in esame il decison boundary partiziona il piano in parabola o iperbole:

$$\sum_{d=1}^{2} (a_d x_d^2 + b_d x_d) + c = 0$$

2.4.4 Trade-Offs del Naive Bayes

1. **Velocità**: sia il learning che la classifcazione hanno complessità computazionale molto bassa.

2. **Memoria**: il modello richiede solo $O(DC)$ parametri. Raggiunge una grande compressione dei dati di training.

3. **Interpretabilità**: Il modello ha una buona interpretabilità poiché i parametri di $\phi(x)$ corrispondono alle medie class conditional.

4. **Accuracy**: Ciò che si fa è un'approssimazione con l'assunzione di indipendenza delle feature e con le forme caninche dei class conditianl marginals, quindi non si adatterà mai perfettamente ai casi reali. Quindi l'accuracy non è molto alta.

5. **Dati**: Bisogna stare attenti nella stima dei parametri nel caso discreto quando i dati sono scarsi.

2.5 Linear Discriminant Analysis

L'LDA è una tecnica di classificazione creata da Fisher nel 1930.

Può essere vista come una diversa approssimazione rispetto al Bayes Optimal Classifier per dati reali.

Invece di un prodotto di normali indipendenti come nel Naive Bayes, l'LDA assume la densità di classe condizionale come normali multivariate con una matrice di covarianza comune.

$$\phi_c(x) = N(x; \mu_c, \Sigma) = \frac{1}{|(2\pi)\Sigma|^{\frac{1}{2}}} \exp^{\left(-\frac{1}{2}(x-\mu_c)^T \Sigma^{-1}(x-\mu_)\right)}$$

La funzione di classificazione è:

$$f_{LDA}(x) = \arg\max_{c \in Y} \pi_c \phi_c(x)$$

Come nel Naive Bayes, i parametri sono imparati a maximum likelihood.

Probabilità di classe:

$$\phi_c = \frac{1}{n} \sum_{i=1}^{n} [y_i = c]$$

Media di classe:

$$\mu_c = \frac{\sum_{i=1}^{n} [y_i = c] x_i}{\sum_{i=1}^{n} [y_i = c]}$$

Covarianza condivisa:

$$\Sigma = \frac{1}{n} \sum_{i=1} (x_i - \mu_{y_i})(x_i - \mu_{y_i})^T$$

2.5.1 Interpretazione Geometrica dell'LDA nel caso binario

Il decision boundary corrisponde al set di punti x per cui:

$$\log(\pi_0) - \frac{1}{2}\log|2\pi\Sigma| - \frac{1}{2}\log(x-\mu_0)^T\Sigma^{-1}(x-\mu_0) -$$
$$-\log(\pi_1) + \frac{1}{2}log|2\pi\Sigma| + \frac{1}{2}\log(x-\mu_1)^T\Sigma^{-1}(x-\mu_1) = 0$$

Per via della matrice di covarianza comune possiamo eliminare molti termini e otteniamo:

$$(\log(\pi_0) - 0.5\mu_0^T\Sigma^-1\mu_0 + \mu_0^T\Sigma^-1x) - (\log(\pi_1) - 0.5\mu_1^T\Sigma^-1\mu_1 + \mu_1^T\Sigma^-1x) = 0$$

La prima parentesi è il boundary per la classe 0, la seconda è il boundary per la classe 1.

Ciò mostra che il decision boundary è lineare in x. Quindi l'LDA è un classificatore lineare.

2.5.2 Masking

Il fenomeno del masking nell'LDA si verifica quando le caratteristiche altamente correlate, o ridondanti, nascondono l'importanza di altre caratteristiche rilevanti. Questo accade principalmente perché l'LDA utilizza la matrice di covarianza condivisa per stimare la separazione tra le classi, e ciò può portare a problemi nel distinguere correttamente le classi.

L'LDA può evitare il masking usando due boundary diversi.

2.5.3 Quadratic Discriminant Analysis

Rilassando la condizione di covarianza condivisa, le class conditional probability densities (sempre gaussiane multivariate) possono avere matrici di covarianza diverse.

Boundary classe 0:

$$(\log(\pi_0) - 0.5\mu_0^T\Sigma_0^-1\mu_0 + \mu_0^T\Sigma_0^-1x)$$

Boundary classe 1:

$$-(\log(\pi_1) - 0.5\mu_1^T\Sigma^-1\mu_1 + \mu_1^T\Sigma^-1x) = 0$$

2.5.4 Trade-Offs dell'LDA

1. **Velocità**: la dipendenza quadratica su D rende l'LDA più lento del Naive Bayes di un fattore D durante il learning e la classificazione.

2. **Memoria**: l'LDA richiede $O(D^2C)$ parametri, che può rappresentare una buona compressione dei dati quando $D << N$.

3. **Interpretabilità**: l'LDA ha una buona interpretabilità dato che μ_c corrisponde alla media delle class conditional.

4. **Accuracy**: le assunizioni fatte dall'LDA non sono sempre estendibili a casi reali e ciò provoca una accuracy non molto alta.

5. **Dati**: generalmente richiede più dati del Naive Bayes perchè deve stimare $O(D^2)$ parametri nella matrice di covarianza.

2.6 Generative VS Discriminative Classifiers

Il Bayes Optimal Classifier, il Naive Bayes e l'LDA sono detti classificatori generativi perchè modellano esplicitamente la joint distribution $P(X, Y)$.

Generative significa che, una volta imparato il classificatore, è sempre possibile campionare esempi x data la classe $y = c$ utilizzando la classe condizionale $p(x|y = c)$ o marginalizzando la joint $P(X, Y)$.

In ogni caso, per costruire un classificatore probabilistico, l'unica cosa che ci serve modellare è $P(y|X)$

I classificatori che si basano su stimare direttamente $P(y|X)$ sono detti classificatori discriminativi perchè ignorano la distribuzione di x e si concentrano solo su $y = c$.

2.7 Logistic Regression

La Logistic Regression è un classificatore probabilistico discriminativo.

Nel caso binario modella la probabilità a posteriori come:

$$P(Y = 0|x) = \frac{e^{w^T x+b}}{1 + e^{w^T x+b}}$$

$$P(Y = 1|x) = \frac{1}{1 + e^{w^T x+b}}$$

Il classificatore LR è lineare, infatti il decision boundary è:

$$\log P(Y = 0|x) - \log P(Y = 1|x) = w^T x + b$$

$w^T x + b$ corrisponde all'equazione di una retta/piano/iperpiano.

La funzione di classificazione è:

$$f_{LR} = \arg\max_{c \in Y} P(Y = c|x)$$

2.7.1 Funzione Logistica

Si usa quando si vuole trasformare uno score in una probabilità. È una funzione "a S". È la versione continua di una soglia. Soglia lo score ottenuto da $w^T x + b$.

$$f_l = \frac{1}{1 + e^{-2x}}$$

2.7.2 Multiclass Logistic Regression

La LR può anche essere estesa al caso multiclasse con k classi.

$$P(Y = c|x) = \frac{e^{w_c^T x + b_c}}{1 + \sum_{l=1}^{k-1} e^{w_l^T x + b_l}}$$

Notare che:

1. Sia a numeratore che a denominatore si ha uno scalare.
2. w_c^T è un vettore riga
3. Si ha un vettore per ogni classe.
4. b è il bias.

$$P(Y = k|x) = \frac{1}{1 + \sum_{l=1}^{k-1} e^{w_l^T x + b_l}}$$

La funzione di classificazione rimane la stessa:

$$f_{LR} = \arg\max_{c \in Y} P(Y = c|x)$$

2.7.3 Learning per la Logistic Regression

Si addestra a Maximum Likelihood, quindi i parametri $\theta = \{(w_c, b_c) \mid c \in Y\}$ devono essere scelti per massimizzare la likelihood (o la log likelihood).

Dato un dataset $D = \{(x_i, y_i) \mid i = 1, \ldots, n\}$, la likelihood è:

$$\sum_{i=1}^{n} \log P(Y = y_i | X = x_i)$$

$$\theta^* = \arg\max L(\theta, D) = \arg\max \sum_{i=1}^{n} \log P(Y = y_i | X = x_i)$$

Si deve cercare di avere lo score della classe corretta per tutti i punti.

$L(\theta, D)$ non può essere massimizzata qualitativamente quindi deve essere massimizzata in modo ricorsivo. Sugli altri score non c'è bisogno di fare niente perchè la somma è sempre 1 (uno si alza, gli altri si abbassano).

$$D(x_i, y_i), \text{ n coppie}$$

$$P(Y = c|X)$$

$$L = \prod_{i=1}^{n} P(y_i|x_i)$$

$$\log L = \sum_{i=1}^{n} \log P(y_i|x_i)$$

$$\log P(y_i|x_i) = \log \frac{e^{w_c^T x + b_c}}{1 + \sum_{l=1}^{k-1} e^{w_l^T x + b_l}} = w_c^T x_i + b - \log(1 + \sum_{l=1}^{k-1} e^{w_l^T x_i + b_l})$$

Nel caso a due classi:

$$P(Y=1|x) = \frac{e^{w^T x}}{1+e^{w^T x}}$$

$$P(Y=0|x) = \frac{1}{1+e^{w^T x}}$$

$$L(\theta, D) = \sum_{i=1}^{n} y_i \log P(Y=1|x) + (1-y_i)\log(y=0|x) =$$

Notare che è il negato della binary cross entropy.

$$= \sum_{i=1}^{n} y_i w^T x_i - \log(1+e^{w^T x_i})$$

Per massimizzare devo utilizzare una strategia approssimata: il **Gradient Descent**.

Calcolare la derivata della funzione obiettivo rispetto ad uno qualsiasi dei parametri.

$$\frac{\partial L(x_i, \theta)}{\partial w_k} = (y_i - \frac{e^{w^T x_i}}{1+e^{w^T x_i}})x_i, k$$

y_i è la label, $\frac{e^{w^T x}}{1+e^{w^T x}}$ è lo score della classe 1 valutata in x_i.

- Se $label = 1$ e $score = 1$: perfetto.
- Se $label = 1$ e $score = 0$: sbagliato.

Operativamente:

$$\frac{\partial L(x_i, \theta)}{\partial w^T} = \frac{\partial(y_i w^T x_i - \log(1+e^{w^T x_i}))}{\partial w^T} = (I) - (II)$$

$$(I) = \frac{\partial(y_i w^T x_i)}{\partial w^T} = y_i x_i$$

$$(II) = \frac{\partial(\log(1+e^{w^T x_i}))}{\partial w^T} = \frac{x_i e^{w^T x_i}}{1+e^{w^T x_i}}$$

$$\frac{\partial L(x_i, \theta)}{\partial w^T} = (I) - (II) = \left(y_i x_i - \frac{x_i e^{w^T x_i}}{1+e^{w^T x_i}} \right) = \left(y_i - \frac{e^{w^T x_i}}{1+e^{w^T x_i}} \right) x_i$$

2.7.4 Gradient Descent

Procedura iterativa per trovare i minimi di una funzione. Seguo la pendenza della funzione, data dalla derivata della funzione nel punto in cui mi trovo. A questo punto posso fidarmi della derivata per trovare il minimo.

Il Gradient Descent è spesso utilizzato in Machine Learning per minimizzare una funzione costo, detta **Loss function**. Questa funzione viene utilizzata per valutare l'efficienza di un modello: minimizzare la loss significa massimizzare l'efficienza.

Dico che la soluzione del problema di minimo è un valore a caso θ_j e inizio un processo iterativo.

$$\theta_j \text{iniziale} \rightarrow \theta_j = \theta_j - \alpha \frac{\partial f(\theta)}{\partial \theta_j}$$

- α è il **learning rate** e indica quanto mi fido della derivata. $(0 <= \alpha <= 1)$ se è basso ci fidiamo poco e quindi freniamo, se alto ci fidiamo molto e saltiamo. Se freno troppo ci metto molto tempo ad arrivare a convergenza; se salto troppo rischio di mancare il minimo superandolo.

- $\frac{\partial f(\theta)}{\partial \theta_j}$ è la pendenza o gradiente.

- L'algoritmo arriva a convergenza quando il gradiente è uguale a 0, quindi possiamo trovare il massimo globale $\iff$ la funzione è convessa.

Il Gradient Descent si può applicare a tutti i problemi di minimizzazione. Per l'update posso usare o un solo elemento come il gradiente in un punto, oppure tutto il dataset o pezzi di dataset. L'ottimo sarebbe il gradiente medio su tutto il dataset. Il problema è così facendo ci metto un sacco di tempo perchè ad ogni step devo ricalcolare su tutto il dataset.

Con il **minibatch stochastic gradient descent** calcolo il gradiente su un sottoinsieme del dataset.

2.7.5 Stochastic Gradient Descent for Binary Logist Regression

È il caso estremo in cui un esempio random del training set viene considerato per fare lo step di update.

Algorithm 1: Stochastic Gradient Descent with Logistic Loss

Input: Dataset D of N elements, $y \in \{0, 1\}$, learning rate α

Output: Parameter vector $\mathbf{w}$

Initialize parameter vector $\mathbf{w}$ at random;

while *maximum iteration not reached or convergence not achieved* **do**

  Pick at random a sample $(\mathbf{x}_i, y_i)$ from D;

  Compute vectorized derivatives $\nabla L(\mathbf{x}_i, \mathbf{w})$:;

  Recall

$$\frac{\partial L(\mathbf{x}_i, \mathbf{w})}{\partial w_k} = \left(y_i - \frac{\exp(\mathbf{w}^\top \mathbf{x}_i)}{1 + \exp(\mathbf{w}^\top \mathbf{x}_i)} \right) x_{i,k}, \quad \forall k = 1, \ldots, K$$

  Apply vectorized update rule:;

$$\mathbf{w}_{\text{new}} = \mathbf{w} - \alpha \nabla L(\mathbf{x}_i, \mathbf{w})$$

  Set $\mathbf{w} \leftarrow \mathbf{w}_{\text{new}}$;

end

2.7.6 Interpretazione Geometrica della LR

La LR ha un decision boundary lineare nel caso bianario. Nel caso multiclasse il decision boundary è lineare a tratti. L'LR ha la stessa capacità rappresentazionale dell'LDA.

2.7.7 Trade-Offs della Logistic Regression

1. **Velocità**: per la classificazione LR è più veloce di NB e LDA. Per il learning LR ha bisogno di ottimizzazione numerica che può essere più lenta dell'NB.

2. **Memoria**: richiede $O(DC)$ parametri.

3. **Interpretabilità**: l'importanza di feature variables diverse x_d può essere vista in termini dei loro pesi w_{dc}. Infatti ad ogni feature è attribuito uno score.

4. **Accuracy**: tende ad essere migliore dell'LDA quando ha pochi dati.

2.8 SVM

L'SVM introduce il concetto di **margine**, ossia lo spazio intorno alla retta di separazione in cui non cadono punti. Per semplicità il margine viene sempre costruito simmetrico.

2.8.1 Classificatore Lineare e Margine

La classificazione binaria può essere vista come la task di separare le classi in feature space.

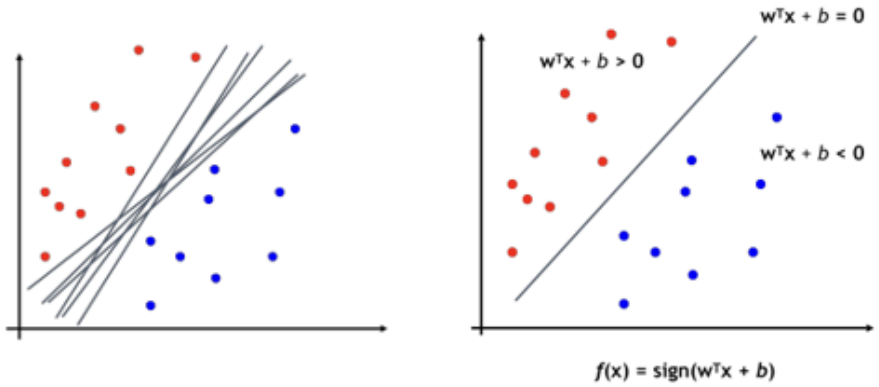


Figura 2.2: separatori

Più separatori sono possibili, tuttavia quello ottimale è quello lontano dai punti perchè permette di costruire una banda di sicurezza tra le due classi.

2.8.2 Costruzione del margine

1. Utlizzo l'equazione della distanza punto retta $r_i = \frac{w^T x+b}{\|w\|}$.

2. Voglio massimizzare questa distanza rispetto ai punti più vicini alla retta.

Non tutti i punti sono uguali, e non concorrono allo stesso modo nella scelta della retta migliore. I punti migliori sono detti **support vector**.

Matematicamente

È un problema di classificazione supervisionata, quindi ho il dataset annotato $D = (x_i, y_i)\ i \in [1, \ldots, N]\ y_i \in \{+1, -1\}$

Dico che ρ è un ipotetico margine. Voglio che i punti che stanno sopra alla retta ci stiano per più di $\frac{\rho}{2}$ e che tutti quelli che stanno sotto ci stiano per meno di $\frac{\rho}{2}$:

$$\begin{cases} w^T x_i + b \geq \frac{\rho}{2} & \text{se } y_i = +1 \\ w^T x_i + b \leq \frac{\rho}{2} & \text{se } y_i = -1 \end{cases}$$

Utilizzando la label posso riscrivere queste due equazioni in una sola:

$$y_i(w^T x_i + b) \geq \frac{\rho}{2} \quad \forall x_i \in D$$

A questo punto si deve trovare una retta che divida i punti e che abbia ρ più grande possibile.

Per i support vector la diseguaglianza in realtà è un uguaglianza:

$$w^T x_s + b = \frac{\rho}{2}$$

Quindi richiamando r_i e assorbendo ρ in $\|w\|$ si ottiene:

$$r_s = \frac{w^T x_s + b}{\|w\|} = \frac{\rho}{2\|w\|} = \frac{1}{\|\hat{w}\|}$$

- w sono i parametri del classificatore e sono da imparare.
- prendo w e divido ogni coordinata per $\frac{\rho}{2}$ e lo chiamo $\hat{w}$.

$$\rho = 2r_s = \frac{2}{\|\hat{w}\|}$$

2.8.3 SVM Quadratic Problem

- Massimizzare il margine.
- I punti devono essere classificati bene.

Massimizzo il margine:

$$\arg\max_{w,b} \frac{2}{\|\hat{w}\|}$$

Seleziono la retta:

$$y_i(w^T x_i + b) \geq 1 \quad \forall (x_i, y_i) \in D$$

Tutto ciò è equivalente al problema QP:

$$\arg\min_{w,b} \|\hat{w}\|^2$$

$$y_i(w^T x_i + b) \geq 1 \quad \forall (x_i, y_i) \in D$$

QP solvers

Pacchetti software che risolvono questi problemi. Uno dei più utilizzati è Sequential Minimum Optimization SMO. Viene risolto il problema duale

2.8.4 Ottimizzazione Vincolata

Bisogna imparare il vincolo nell'equazione che vogliamo risolvere.

Per calcolare il minimo di una funzione soggetta a vincoli si utilizzano i moltiplicatori di Lagrange.

Problema da risolvere:

$$\min f(x)$$

Soggetto al vincolo:

$$h_k(x) >= 0 \quad k \in 1, \ldots, K$$

Si utilizza la funzione Lagrangiana

$$L(x, \mu) = f(x) - \sum_{i}^{k} \alpha_i h_i(x) \quad \alpha_i \geq 0$$

Notare che si è introdotto un parametro α_i per ogni vincolo, e si trasforma il problema da risolvere.

Se per x_i, $h(x_i) > 0$ allora il vincolo è già soddisfatto e si dice che è **inattivo**, inoltre si ha che la soluzione è:

$x^* | \nabla f(x) = 0$

Se, invece, x_i, $h(x_i) = 0$ allora il vincolo si dice **attivo** e la soluzione è:

$x^* | \nabla f(x) - \alpha^T \nabla h(x) = 0$

Notare che se $\alpha_i = 0$ per ogni i allora $\nabla f(x) = 0$ che è la soluzione ideale. Se però si ha sia degli α uguali a 0 che degli alpha positivi:

$$\frac{\partial}{\partial x}\left(f(x) + \sum_{i=1}^{k} \alpha_i h_i(x)\right) = \nabla f(x) + \sum_{i=1}^{k} \alpha_i \frac{\partial h_i(x)}{\partial x}$$

Da cui:

$$\nabla f(x) - \alpha^T \nabla h(x) = 0$$

Mettendo tutto insieme abbiamo le KKT ossia le condizioni necessarie affinchè il problema sia risolvibile

$$(I) = \nabla f(x^*) - \alpha^T \nabla h(x^*) = 0$$
$$(II) = h(x^*) <= 0$$
$$(III) = \alpha_i \geq 0$$
$$(IV) = \alpha^T \nabla h(x) = 0$$

Notare che:

- (II): La soluzione deve rispettare i vincoli
- (III): Se uno dei vincoli non è nullo il moltiplicatore deve essere 0. Se h è nullo il moltiplicatore deve essere positivo.
- Se il vincolo è lineare ed f è convessa, le KKT sono condizioni necessarie e sufficienti per avere un'unica soluzione.

Nel caso dell'SVM quindi si ottiene:

$$\min_{w,b,\alpha} L(w, b, \alpha) = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{N} \alpha_i \left[y_i(w^T x_i + b) - 1\right]$$

Le KKT sono:

$$\alpha_i \geq 0$$
$$y_i(w^T x_i + b) - 1 \geq 0$$
$$\alpha_i \left[y_i(w^T x_i + b) - 1\right] = 0$$

Si deriva rispetto a tre variabili w, b, α e si pone uguale a 0.

1. Rispetto a w:

$$w = \sum_{i=1}^{N} \alpha_i y_i x_i$$

   Mette in relazione w e il moltiplicatore, il vincolo si attiva (α_i>0) solo per i support vector.

2. Rispetto a b:

$$\sum_{i=1}^{N} \alpha_i y_i = 0$$

Sostituendo w nella Lagrangiana otteniamo:

$$L(w, b, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j (x_i^T x_j)$$

Che risulta nella forma duale di WOLFE:

$$\max_{\alpha} \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j (x_i^T x_j)$$

Soggetta a:

$$\sum_{i=1}^{N} \alpha_i y_i = 0$$
$$\alpha_i \geq 0$$

- $\alpha_i = 0$ x_i è irrilevante
- $\alpha_i > 0$ X_i è un support vector (quindi è critico)

La funzione di classificazione dell'SVM diventa quindi (assorbendo b in w):

Primale:

$$f(x) = w^T x$$

Duale:

$$f(x) = \sum_{i=1}^{N} \alpha_i y_i x_i^T x$$

2.8.5 Non Separable Soft Margin

Se non trovo una retta che separa il dataset perfettamente ci deve essere qualche vincolo che devo violare. Quindi si utilizzano variabili in più per gestire vincoli che posso violare:

Ad ogni elemento (≥ 0) associo una slack variable ϵ_i (=0 se l'elemento non viola il vincolo, >0 se lo viola)che permette di traslare l'elemento del minimo necessario.

$$\min \frac{1}{2}\|w\|^2 + C \sum_{i}^{N} \epsilon_i$$

Soggetto a

$$y_i(x_i w + b) \geq 1 - \epsilon_i$$

C è un'iperparametro che indica quanto è importante violare i vincoli rispetto a minimizzare il margine.

Il problema di questo approccio è che è un trucco.

2.8.6 SVM non lineare

Se i dati non sono separabili linearmente si cambia lo spazio di classificazione in modo da far diventare il boundary lineare.

L'SVM è l'unico classificatore che ha la possibilità di diventare non lineare.

Soluzione: aumentare il numero di feature, quindi il numero di coordinate. Mi sposto in uno spazio a numero di dimensioni superiore.

Devo trovare una funizone che mi porta da un feature vector x ad un feature vector $\rho(x)$.

$$\Phi : x \to \rho(x)$$

Il nuovo spazio deve avere più dimensioni perchè aumentando il numero di dimensioni il problema tende a linearizzarsi.

L'SVM permette di definire questo mapping in modo implicito.

Prendo l'SVM duale:

$$f(x) = \sum_{i=1}^{N} \alpha_i y_i x_i^T x$$

$$L(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j (x_i^T x_j)$$

E applico il mapping

$$\Phi : x \to \rho(x)$$

$$\Phi(x_i)^T \Phi(x_j) = K(x_i, xj)$$

E applicando il mapping risulta:

$$f(x) = \sum_{i=1}^{N} \alpha_i y_i K(x_i, x_j)$$

- Notare che $x_i T x_j$ è uno scalare

- K è la funzione di kernel

- $K(x_i, x_j)$ dice qual è il prodotto scalare nello spazio trasformato, però non dice come fare questa trasformazione.

Kernel Trick

Bisogna capire quali funzioni possono essere considerate dei Kernel.

Teorema di Mercer

Ogni matrice simmetrica semi positiva è una matrice di Gram ed è un Kernel

$$\mathbf{K} = \begin{bmatrix} K(\mathbf{x}_1, \mathbf{x}_1) & K(\mathbf{x}_1, \mathbf{x}_2) & \cdots & K(\mathbf{x}_1, \mathbf{x}_N) \\ K(\mathbf{x}_2, \mathbf{x}_1) & K(\mathbf{x}_2, \mathbf{x}_2) & \cdots & K(\mathbf{x}_2, \mathbf{x}_N) \\ \vdots & \vdots & \ddots & \vdots \\ K(\mathbf{x}_N, \mathbf{x}_1) & K(\mathbf{x}_N, \mathbf{x}_2) & \cdots & K(\mathbf{x}_N, \mathbf{x}_N) \end{bmatrix}$$

Kernel Popolari

1. Lineare: nessuna trasformazione di spazio di feature

$$K(x, x_i) = x^T x$$

2. Polinomiale: mappa in $\binom{d+p}{d}$ dimensioni

$$K(x, x_i) = (1 + x^T x)^p$$

3. RBF Kernel Gaussiano: mappa in uno spazio infinito-dimensionale

$$K(x, x_i) = e^{-\frac{\|x - x_i\|^2}{2\sigma^2}}$$

   σ^2 è la **banda del Kernel**. Se è < 1 la distanza viene amplificata, se è >1 la distanza viene ammorbidita.

Dimostrazione

Il Kernel Gaussiano è la normalizzata di $e^{\frac{x^T x_i}{\sigma^2}}$. Utilizzando $e^x = \sum_{n=1}^{\infty} \frac{x^n}{n!}$

$$e^{\frac{x^T x_i}{\sigma^2}} = \sum_{n=1}^{\infty} \frac{(x^T x)^n}{\sigma^{2n} n!}$$

Che è una sommatoria infinita, quindi ∞ dimensioni.

Se uso i Kernel non posso usare l'SVM primale ma solo la duale. Utilizzare i kernel è meno efficiente perchè devo calcolare, tramite il kernel, molte distanze.

Una volta che ho classificato posso tornare nello spazio originale e vedere come vengono i boundaries. Saranno non lineari nello spazio originale ma lineari in quello trasformato.

Se faccio la linea di separazione troppo complicata sono sensibile all'overfitting, ossia che il classificatore stia imparando a memoria il dataset.

2.8.7 Trade-Offs dell'SVM

- **Velocità vs Accuratezza**
  - **Pro**: Gli SVM sono molto accurati per dataset di dimensioni moderate e alta dimensionalità.
  - **Contro**: Con dataset molto grandi, l'addestramento può diventare lento, specialmente con kernel complessi (es. RBF).
- **Memoria vs Prestazioni**
  - **Pro**: Gli SVM sfruttano solo i vettori di supporto, quindi memorizzano meno dati rispetto ad altri algoritmi.

  - **Contro**: Per dataset molto grandi, la complessità computazionale e la memoria richiesta possono crescere rapidamente $O(n^2)$ o peggio.

- **Dati vs Generalizzazione**

  - **Pro**: Gli SVM si comportano bene in dataset con molte feature (alta dimensionalità).

  - **Contro**: Con dataset piccoli o rumorosi, rischiano di sovradattarsi, richiedendo una buona ottimizzazione dei parametri.

- **Interpretabilità vs Complessità**

  - **Pro**: Con kernel lineare, gli SVM producono modelli interpretabili (pesi delle feature).

  - **Contro**: Con kernel non lineari, il modello diventa una "scatola nera", difficilmente interpretabile.

- **Scelta del Kernel vs Robustezza**

  - **Pro**: Gli SVM sono versatili grazie ai diversi kernel (lineare, polinomiale, RBF).

  - **Contro**: La scelta del kernel e dei relativi iperparametri richiede tempo ed esperienza.

Capitolo 3

Metodi di Insieme

I metodi di insieme servono per assemblare più classificatori insieme per avere perferomance migliori rispetto a quelle che si avrebbero con un singolo classificatore.

Ci sono due tecniche:

- **Bagging**
- **Boosting**

3.1 Definizione di Ensamble o Insieme

Una collezione di modelli che sono trainati per risolvere lo stesso problema di classificazione.

Nel 90% dei casi il tipo di tecnica viene mantenuto identico nell'insieme.

Normalmente l'insieme va meglio del singolo classificatore.

Supponiamo di voler risolvere un problema di classificazione binario, e supponiamo di avere K classificatorin addestrati. Supponiamo che mediamente tutti i classificatori abbiano all'incirca lo stesso error-rate < 0.5 e che gli errori siano indipendenti. Con

una semplice **majority voting** si può migliorare la classificazione diminuendo la varianza nel setting.

3.1.1 Dimostrazione della Majority Voting

Supponiamo di avere un insieme di K classificatori con accuracy $\alpha > 0.5$;

L'insieme majority voting produce un errore quando $\frac{K}{2} + 1$ classificatori producono un errore, ossia una predizione sbagliata. Gli errori seguono la distribuzione binomiale cumulativa $P(x \leq k) = \sum_0^k \binom{K}{k} \alpha^i (1-\alpha)^{K-i}$ la quale è sicuramente minore di quella del singolo classificatore.

3.1.2 Come ottenere la condizione di Errori Indipendenti

Si ottiene per costruzione. Si dovrebbe addestrare ciascun classificatore su un set di dati diverso dagli altri classificatori. La debolezza di questo approccio è che il dataset deve essere molto grande.

3.2 Bagging

È un'approssimazione del metodo precedente. Gioca con la randomness campionando a caso dal training set.

Ciò che fa è prendere un singolo training set Tr e ne estrae casualmente K sottoinsiemi (con ripetizione) per formare K set di addestramento $T_{r1}, \ldots, T_{rK}$.

Ognuno di questi training set è utilizzato per addestrare un'istanza dello stesso classificatore. Si ottengono così K funzioni di classificazione: $f_1(x), f_2(x), \ldots, f_K(x)$.

3.2.1 Perchè il Bagging funziona

Supponiamo che ogni classificatore impari una funzione:

$$y = f_i(x) + e_i$$

L'errore medio di M classificatori sul dataset D è:

$$\epsilon_{AV} = \frac{1}{M} \sum_{i=1}^{M} \mathbb{E}_D[e_i]$$

$\mathbb{E}_D[e_i]$ non dipende da quale elemento sbaglio. Quando faccio bagging **non** sono in questa situazione.

$$y_{bagged} = \frac{1}{M} \sum_{i=1}^{M} f_i(x) + e_i(x)$$

$$\epsilon = \mathbb{E}_D \left[\frac{1}{M} \sum_{i=1}^{M} e_i(x) \right]$$

Notare che questa volta si tiene conto di quali elementi si sbaglia.

$$\epsilon = \mathbb{E}_D \left[\frac{1}{M} \sum_{i=1}^{M} e_i(x) \right] \leq \epsilon_{AV}$$

I due valori sono uguali solo se i valori attesi non sono indipendenti.

3.3 Boosting

È considerato uno dei più grandi sviluppi nel machine learning. Il principio alla base del boosting è che i classificatori hanno visibilità su quello che è successo ai classificatori precedenti.

Non tutti i classificatori sono uguali ma ciascuno ha una reputazione in base a quanto sbaglio.

3.3.1 Idea alla base del Boosting

1. Associo ad ogni elemento del dataset un peso uniforme. Se ho N esempi avrò $w_k = \frac{1}{N}$.

2. Addestro un primo classificatore e misuro la sua accuracy α_i che rappresenta la reputazione del classificatore.

3. Guardo gli esempi del training set che sono stati classificati bene e non li tocco. A quelli classificati male aumento il peso e normalizzo.

4. Il secondo classificatore lavorerà su un dataset campionato usando questi esempi come probabilità di considerare un esempio.

In questo modo, ad ogni step della catena, più un elemento è stato sbagliato più è probabile che finisca nel dataset dell'elemento successivo.

La regola di decisione finale del boosting è:

$$y(x_{new}) = sign\left(\sum_{i}^{M} \alpha_i f_i(x_{new})\right)$$

3.3.2 Come Campionare

Si costruisce una tabella che ha due colonne per ogni esempio del dataset.

- **Input**: devo avere una distribuzione discreta $H(x)$ con $x \in \{1, \ldots, N\}$ e voglio campionare K elementi secondo la distribuzione.

- **Output**: K indici.

Ci sono due step:

1. **Step 1**: per ogni x:
$$T_x = \left[\sum_0^{x-1} H(x), \sum_0^{x} xH(x) \right]$$

2. **Step 2**: Si pescano gli indici.

(a) **While** (numero_iter $= K$) **do:**
(b) Pesca un numero casuale $n \in [0, 1]$.
(c) Trova k tale che $T_k[0] \leq n < T_k[1]$.
(d) Output: k.
(e) **end while.**

3.3.3 Adaboost

Metodo di boosting in cui si cerca di aggirare il metodo di campionamento. Invece che realizzare i dataset con esempi proporzionali al loro peso, interpreto il peso come un'iportanza dell'esempio. Parte come il boosting quindi tutti gli elementi hanno peso $\frac{1}{N}$. Voglio fare k classificazioni in serie, traino il primo classificatore su tutto il dataset e misuro l'errore del classificatore come:

$$\epsilon_k = \sum_{i=1}^{N} w_K^i I(f_k(x_i) \neq y^i)$$

Il classificatore per sbagliare il meno possibile deve fare bene quelli che hanno peso più alto.

$$\alpha_k = \log\left(\frac{1-\epsilon_{ki}}{\epsilon_k}\right)$$

Per aggiornare i pesi:

$$\begin{cases} w_k e^{-\lambda k} & \text{se } f_k(x^i) = y^i \\ w_k e^{+\lambda k} & \text{se } f_k(x^i) \neq y^i \end{cases}$$

Poi si rinormalizza.

È importante che ogni classificatore abbia un valore di errore, che tiene conto dei pesi, minore di 0.5, altrimenti la catena viene riavviata.

Ogni stage mantiene il trust:

$$\alpha_k = \log\left(\frac{1-\epsilon_{ki}}{\epsilon_k}\right)$$

$$y_i = sign\left(\sum_{k=1}^{M} \alpha_k f_k(x_i)\right)$$

3.3.4 Classificatori utilizzabili

I classificatori devono essere semplici, come ad esempio il **Decision Stump**.

Ripeti fino a $\epsilon < 0.5$:

1. Scegli a caso una feature.
2. Scegli a caso una soglia.
3. Valuta se l'accuracy α è $<$ del 50%.

3.4 Random Forest

Metodo costruito sul bagging. Il cuore del Random Forest è l'albero decisionale binario. Ad ogni nodo l'albero pone un quesito.

È un modello completatamente interpretabile perchè posso ricostruire il ragionamento del classificatore.

3.4.1 The Basics: Binary decision Trees

Parametri del classificatore:

1. feature vector $v \in \mathbb{R}^n$
2. split function $f_n(v) : \mathbb{R}^n \to \mathbb{R}$
3. threshold $t_n \in \mathbb{R}$
4. classifications $P_n(c)$

Durante il training dobbiamo imparare la funzione di split e la soglia.

Nelle foglie non c'è un valore ma un istogramma: n barre = n classi. In ordinata ho il numero di elementi di training che hanno seguito quel percorso e sono finiti in quella foglia con quel valore di classe.

Left Split:

$$I_l = i \in I_n | f(v_i) < t$$

Right Split:

$$I_r = I_n - I_l$$

Per la funzione di split possiamo scegliere il Random Stump → **Extreme Randomized Random Forest**

$$t \in (\min_i f(v_i), \max_i f(v_i))$$

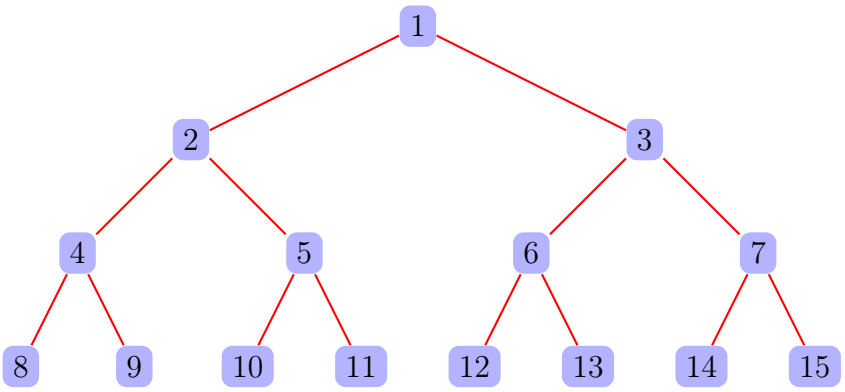
Si deve scegliere t ed f per massimizzare l'information gain.

$$\Delta E = -\frac{|I_l|}{|I_n|}E(I_l) - \frac{|I_r|}{|I_n|}E(I_r)$$

- $|I_l|$ = cardinalità elemnti di sinistra.
- $\frac{|I_l|}{|I_n|}$ = quanti elementi sono finiti nello split a sinistra applicando i parametri che sto valutando.
- $E(I_l)$ = entropia della label che sto valutando.
- per lo split di destra è uguale.

L'entropia E è calcolata dall'istogramma delle label in I.

3.4.2 Entropia di un uno split



Al nodo 1 faccio vedere tutto il training set un elemento alla volta.

1. Per prima cosa scelgo un indice di feature che sto considerando, con il decision stump.
2. Scelgo tot valori di soglia su quella feature.
3. Tra questi scelgo quello che preferisco.
4. Conto quanti elementi si hanno a destra e sinistra.

5. Valuto l'information gain.

L'istogramma delle label viene fatto per ogni split I_l e I_r e l'entropia si trova tramite l'istogramma.

$$E = -\sum_{i}^{n_l abel} h_i \log_2(h_i)$$

h_i è il valore dell'istograamma.

Se tutte le classi hanno la stessa probabilità allora l'entropia è 1, e quindi si ha massimo disordine.

Se una classe è da sola allora l'entropia è 0 e si ha ordine massimo.

Massimizzare l'information Gain corrisponde a minimizzare l'entropia di uno split.

Si allena un nodo alla volta tramite un algoritmo ricorsivo.

3.4.3 Quante feature e t provare

1. Solo uno: extremely randomized.

2. alcune → veloce, potrebbe underfittare, forse troppo profondo.

3. molte → lento, può overfittare.

3.4.4 Quando fermare l'albero

1. maximum depth.

2. entropia minima ottenuta.

3. delta class distribution.

4. pruning di livello o singolo.

3.4.5 Come trasformare l'albero in un metodo d'insieme

Si vuole fare un bag di alberi, quindi sommo tutti gli istogrammi e normalizzo.

Il Random Forest è un bag di alberi decisionali binari.

I bag possono essere fatti in diversi modi:

1. A caso.
2. A mano.
3. Bag con overlap.
4. Bag a livello di feature.
5. Bagging non uniforme.

Capitolo 4

Clustering

È anche detto learning non supervisionato. Organizza i dati in classi in modo che ci sia:

1. Alta similarità intra-classe.

2. Bassa similarità inter-classe.

A differenza della classificazione, trova le label delle classi e il numero delle classi direttamente dai dati.

Informalmente si può dire che trova raggruppamenti naturali tra gli oggetti.

Il problema del clustering è soggettivo, dipende dalle caratteristiche su cui si fa clustering. Inoltre, la similarità non è univoca ma dipende dalle caratteristiche che guardiamo.

Abbiamo due modi per misurare se due elementi sono simili:

1. Misurare una distanza. (matematicamente è più solido)

2. Misurare una similarità.

4.1 Proprietà della distanza

1. **Simmetria**:
$$D(A, B) = D(B, A)$$

2. **Self-Similarity**:
$$D(A, A) = 0$$

3. **Positività**:
$$D(A, B) = 0 \iff A = B$$

4. **Disuguaglianza triangolare**:
$$D(A, B) < D(A, C) + B(B, C)$$

Il clustering può essere:

- **Gerarchico**: è iterativo. Si uniscono e separano gli elementi in modo gerarchico. Ciò che si fa è costruire un albero di unioni o separazioni.

- **Partizionale**: in un unico step divide gli elementi nei gruppi che vogliamo ottenere.

4.2 Proprietà dell'algoritmo di Clustering

1. Scalabilità

2. Capacità di gestire dati diversi

3. Meno conoscenza possibilie del dominio

4. Gestire rumore e outliers

5. Possibilità di inserire vincoli

6. Non dipendere dall'ordine dei dati

7. Interpretabilità e usabilità

4.3 Clustering Gerarchico

Ci sono due possibilità a seconda di come viene costruita la gerarchia:

- **Bottom-Up (Agglomerativo)**: inizialmente considero gli elementi tutti disgiunti e poi farò operazioni di merge, cioè fondo i gruppi.

- **Top-Down (Divisivo)**: Parto dalla radice in cui tutti gli elementi sono insieme e stacco un elemento alla volta.

Il numero di dendogrammi con n foglie è:

$$\frac{(2n-3)!}{[(2^{n-2})(n-2)!]}$$

In ogni caso all'inizio si costruisce una matrice dove per righe e colonne si hanno gli elementi e nella casella di intersezione le distanze o similarità degli elementi. La matrice è triangolare perchè sia distaza che similarità sono simmetriche.

4.3.1 Bottom-Up

Parto con tutti gli elementi disgiunti e con già definita una misura. A questo punto per tutte le possibili coppie degli elementi misuro la loro distanza/simiarità. Scelgo la coppia per cui la distanza è minima e la unisco. A questo punto ho tuuti gli elementi disgiunti tranne la coppia. Si deve ripetere questa procedura fino a quando non ho unito tutto. Si deve capire come elaborare la distanza gruppo-gruppo ed elemento-gruppo.

4.3.2 Strategie di Linkage

1. Single Linkage (alta varianza): la distanza gruppo-gruppo ed elemento-gruppo è la distanza tra gli elementi più vicini. Porta ad una forma più elongata ed incentiva l'unione tra elementi.

2. Complete Linkage (bassa varianza): la distanza considerate è quella tra gli elementi più lontani. Porta a forme più compatte.

3. Average Linkage: la media delle distanze degli elementi di A con quelli di B.

4. Wards Linkage: $\frac{Media}{Varianza}$

Posso smettere di aggregare quando la distanza è sopra ad una certa soglia.

4.3.3 Caratteristiche del Clustering Gerarchico

- Non bisogna specificare il numero di cluster in anticipo
- Non scala bene: si ha almeno $O(n^2)$ dove n è il numero di elementi.
- Gli ottimi locali sono un problema
- L'interpretazione del risultato è soggettiva

4.3.4 Clustering Partizionale

Non creo sottogruppi iteramente ma partiziono subito l'insieme nel numero di gruppi che voglio.

4.4 K-Means

È l'algoritmo di clustering partizionale più famoso.

Dati un set di punti $(x_1, x_2, \ldots, x_k)$ e k set $S = S_1, S_2, \ldots, S_k$ la funzione obiettivo è:

$$\arg\min_S \sum_{i=1}^{k} \sum_{x \in S_i} \|x - \mu_i\|^2$$

μ_i è la media dei punti in S_i.

Il che è uguale a:

$$\arg\min_S \sum_{i=1}^{k} \sum_{x \in S_i} |S_i| Var S_i = \arg\min_S \sum_{i=1}^{k} \sum_{x,y \in S_i} \|x - y\|^2$$

4.4.1 Dimostrazione

$$\sum_{i=1}^{N} \sum_{j=i}^{N} (x_i - x_j)^2 = \sum_{i=1}^{N} \sum_{j=i}^{N} (x_i + \hat{x} - \hat{x} - x_j)^2 =$$

$$= \sum_{i=1}^{N} \sum_{j=i}^{N} (x_i + \hat{x})^2 + (\hat{x} - x_j)^2 + (x_i + \hat{x})(x_j + x) = 2N \sum_{j=i}^{N} (x - \hat{x})^2$$

4.4.2 K-means Naive Procedure

Algorithm 2: K-Means Clustering Algorithm

Input: Dataset D with N points, number of clusters K, maximum iterations `maxiter`

Output: Cluster centers $\{\mu_{S_1}, \mu_{S_2}, \ldots, \mu_{S_K}\}$

Initialize K random cluster centers $\{\mu_{S_1}, \mu_{S_2}, \ldots, \mu_{S_K}\}$;

while *cluster centers are not updated* **or** `maxiter` *is reached* **do**

  Assign every point to the closest cluster center;

  foreach *cluster* S_i **do**

    Compute the new center as the average of the assigned points:;

$$\mu_{S_i}^{\text{new}} = \frac{1}{|S_i|} \sum_{x \in S_i} x$$

    Update the cluster center:;

$$\mu_{S_i} \leftarrow \mu_{S_i}^{\text{new}}$$

  end

end

4.4.3 Commenti sul K-Means

- **Forza**:
  1. Abbastanza efficiente $O(tkn)$;
  2. Spesso termina in un ottimo locale.
- **Debolezza**:
  1. Applicabile solo quando si buò definire la media.
  2. Bisogna specificare il numero di cluster in anticipo.
  3. Non gestisce outliers e rumore.

4. Non può essere usato per scoprire cluster che non abbiano una forma convessa (per cluster di forma non convessa si dovrebbe utilizzare il clustering spettrale).

4.5 Partitioning around Medoids (PAM)

Trovare oggetti rappresentabili, detti medoids, nei cluster. Si utilizza nel caso non si possa fare la media.

Parte da un set iniziale di medoidi e li rimpiazza iterativamente uno a uno con un medoide se migliora la distanza totale del clustering risultante.

Non crea nuovi punti ma utilizza i punti del dataset come partenza.

Sceglie k punti del dataset e ogni elemento viene assegnato al centro più vicino. L'update fatto in questo modo: Il nuovo centro, invece di essere la media delle coordinate dei punti assegnati al cluster, è il punto, tra quelli assegnati al cluster, la cui somma delle distanze da tutti gli altri punti è più bassa. Questo viene fatto per tutti i punti del cluster.

4.6 Elementi di teoria dei grafi

Un grafo $G = (V, E)$ consiste di un set di vertici V e di un set di archi E.

Un grafo può essere:

- **Diretto**: la connessione tra due vertici ha un verso.
- **Indiretto**: le connessioni non hanno verso.

Un grafo è bipartito se c'è la possibilità di tagliare il grafo e dividerlo in due gruppi disgiunti.

Ogni vertice può avere uno scalare che corrisponde ad un peso.

Ogni arco può avere uno scalare che corrisponde al valore dell'arco.

4.6.1 Campo di un grafo

Operatore $L_2(v)f : V \to \mathbb{R}$ è un vettore di funzioni/coefficienti che associa ad ogni vertice un numero. Permette di definire il prodotto scalare.

$$< f, g >_{L_2(v)} = \sum_{i_i nv} a_i f_i g_i$$

Corrisponde allo **spazio di Hilbert**, ossia uno spazio equipaggiato dal prodotto scalare, ovvero uno spazio in cui si possono calcolare le distanze.

f e g sono due campi mentre a_i è il coefficiente del vertice.

4.6.2 Definizione di Gradiente

Funzione definita dai vertici agli archi:

$$L_2(V) \to L_2(E)$$

$$(\nabla f)_{ij} = \sqrt{w_{ij}}(f_i - f_j)$$

4.6.3 Definizione di Divergenza

Funzione definita dagli archi ai vertici:

$$L_2(E) \to L_2(V)$$

$$div(F)_i = \frac{1}{a_i} \sum_{(i,j) \in E} \sqrt{w_{ij}}(F_{ij} - F_{ji})$$

È l'opposto del gradiente.

4.6.4 Operatore Laplaciano

Va dai vertici ai vertici

$$L_2(V) \to L_2(V)$$
$$\Delta(F)_i = \frac{1}{a_i} \sum_{j:(i,j)} w_{ij}(f_i - f_j)$$

È la somma del gradiente per tutti gli archi entranti in un determinato vertice. In forma matriciale si presenta come una matrice NxN.

$$\Delta = A^{-1}(D - W)$$

Dove:

- A^{-1} è la matrice dei coefficienti dei vertici. È una matrice diagonale.
- D è la matrice di grado, è una matrice diagonale e contiene la somma degli archi entranti in un vertice.
- W è la matrice dei pesi degli archi.
- Se $A = 1$ il Laplaciano non è normalizzato.
- Se $A = D$ il Laplaciano è Random Walk.

La matrice Laplaciana è:

$$L = D - W$$

È una matrice simmetrica definita positiva, quindi ammette n autovalori non negativi e autovalori reali ortogonali.

$$\nabla\Phi_k(x) = \lambda_k\Phi_k(x) \quad k = 1, \ldots$$

4.6.5 Prodotto scalare tra gradiente

Energia di Dirichlet:

$$E_Dir(f) =< \nabla f, \nabla f >_{L^2(\epsilon)} =< \Delta f, f >_{L^2(v)}$$

$$E_Dir(f) = f^T \Delta f$$

Per dimostrarlo basta costruire il gradient di f. La formula

$$(\nabla f)_{ij} = \sqrt{w_{ij}}(f_i - f_j)$$

misura quanto la funzione è morbita.

Trovare set di vettori ortogonali tra loro tali per cui l'energia di Dirichler sia minima significa trovare i vertici del grafo che sono collegati nel modo più smooth possibile.

La soluzione di questo problema sono i primi n autovettori del laplaciano (che vengono dati in ordine).

Gli autovettori danno dei possibili campi ortogonali tra loro che minimizzano l'energia di Dirichler.

4.7 Come fare cluistering con i grafi

Definiamo un grafo dove sui vertici mettiamo gli elementi e sugli archi la similarità tra gli elementi (w_{ij}) che decido io. Una volta definito il grafo, clusterizzare significa trovare una partizione di questo grafo in gruppi in cui all'interno del gruppo gli elementi siano molto simili tra loro. Se sono fuori dal gruppo devono essere poco simili.

4.7.1 Partizione di 2 Gruppi

Devo scegliere quale funzione risolvere.

$$cut(A, B) = \sum_{i \in A, i \in B} w_{ij}$$

che equivale alla somma dei valori degli archi tagliati.

Due funzioni obiettivo comuni sono:

1. **Minimum cut**: $mincut(A, B)$, favorisce gli elementi isolati e produce partizioni poco bilanciate quando ci sono elementi da soli.

2. **Normalized cut (Malik)**: $minNcut(A, B) = \frac{cut(A,B)}{vol(A)} + \frac{cut(A,B)}{vol(B)}$, vol(A) è la somma degli archi in A (volume di A). Favorisce volumi alti e cut bassi.

4.7.2 Come risolvere questo problema

Si definisce un vettore p tale che:

$$p = \begin{cases} +1 & \text{se } vertice \in A \\ -1 & \text{se } vertice \in B \end{cases}$$

Il vettore p ha tanti elementi quanti sono i vertici. p associa ogni vertice ad uno scalare. Quindi è un campo.

$$mincut(A, B) = \arg\min_{p} \sum_{V} w_{ij}(p_i - p_j)^2$$

Partizionare in due un grafo significa trovare il campo ad energia di Dirichler minima, ossia trovare la prima soluzione della minimizzazione di E.

$$mincut(A, B) = \arg\min_{p} p^T L_p$$

Questo problema è NP completo quindi bisogna rilassare p.

4.7.3 Cosa fare per il clustering con un grafo

1. Costruire il grafo.

2. Costruire W.

3. Costruire D.

4. D-W.

5. Calcolare autovettori del Laplaciano.

6. Prendere il secondo autovettore del Laplaciano, ossia il campo che risolve il problema. (autovalore = valore del cut)

4.8 K-Way Spectral Clustering

Con più di due cluster non posso usare un autovettore del laplaciano e aspettarmi che generi una partizione che ha più di due gruppi.

Si fa in questo modo:

1. Recursive Partitioning: si partiziona iterativamente i gruppi.

2. Cluster Multiple Eigenvectors: Si utilizzano gli autovettori del laplaciano come un nuovo spazio in cui fare clustering.

4.8.1 Cluster multiple Eigenvectors

1. Preprocessare il dataset e costruire W.

2. Trovare i primi K autovettori di L.

3. Costruire un nuovo spazio euclideo in cui le coordinate per ogni punto sono gli autovettori del laplaciano. Prendo gli autovettori, li organizzo per colonne, dei questi ne prendo K (ossia il numero di cluster che voglio fare).

4. Faccio il k-means.

4.8.2 Euristica dell'EigenVector

Calcola per ogni coppia di autovalori, ordinati in ordine crescente, la differenza in modulo. La differenza in modulo è la differenza in termini di energia di Dirichler se usassi un autovettore al posto dell'altro.

$$\Delta k = |\lambda_k - \lambda_{k-1}|$$

Il miglior numero di cluster è l'indice dell'eigengap massimo.

Capitolo 5

Deep Learning

Una rete deep, all'aumentare dei dati di training continua a migliorare a differenza dei tradizionali algoritmi di learning. Quindi ad un certo punto posso fissare il modello e lavorare soltanto sui dati.

Nel caso di un metodo di machine learning tradizionale, all'inizio del metodo c'è sempre una fase di feature extraction. Questa parte di estrazione non è collegata all'ottimizzazione dei parametri ma viene fatta prima. Questa cosa non implica che ci possano essere delle feature migliori che facilitano la parte di apprendimento. Nel caso del Deep Learning, la parte di feature extraction e di apprendimento dei parametri non sono disaccoppiate. Accoppiando queste due parti non cambia il classificatore, cambiano le feature.

In generale i metodi di machine learning dipendono da feature di tipo hardcrafted, che hanno il limite di essere sensate per chi le estrae. Non è detto quindi che siano le migliori per risolvere il problema.

Il Deep Learning permette di estrarre la rappresentazione migliore direttamente dai dati.

5.0.1 Partenza da Classificatore Lineare

Supponiamo di voler classificare un'immagine classificata come:

$$x_i \in \mathbb{R}^D \quad i = \ldots, N$$

E supponiamo di voler classificare in K classi.

Il training set sarà composto dalle coppie:

$$(x_i, y_i) \quad \text{dove } y \in 1, \ldots, K$$

L'obiettivo è quello di definire una funzione $f : \mathbb{R}^D \to \mathbb{R}^K$ che mappa le immagini in uno score per ogni classe.

Supponiamo che ogni immagine sia 32x32x3, $x_i \in \mathbb{R}^{3072}$. Possiamo definire una mappatura lineare:

$$f(x_i, W, b) = Wx_i + b$$

Supponiamo di voler imparare 10 funzioni lineari, avremo che:

$$W \in \mathbb{R}^{10x3072}$$

$$b \in \mathbb{R}^{10}$$

Problema: Gli score possono andare da -inf a +inf, quindi sono complicati da confrontare tra loro. Voglio quindi che il classificatore fornisca un valore in un intervallo limitato, come [0,1].

Supponiamo che le classi siano 2 e non 10. Aggiungo all'output della funzione lineare la funzione non lineare σ.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad \text{funzione sigmoide}$$

$$f(x_i, W, b) = \sigma(Wx_i + b)$$

Si ottiene dunque la Logistic Regression, che può essere trainata ottimizzando la binary cross entropy loss:

$$L(w, b) = -\frac{1}{m} \sum_{i=1}^{m} [y_i \log(f(x_i)) + (1 - y_i) \log(x - f(x_i))]$$

5.0.2 Softmax Classifier

Generalizza la Logistic Regression in multi-class classification.

$$softmax_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

Prende un vettore di score arbitrari z e li mette in un vettore di valori tar 0 e 1 che sommano a 1.

Per ottimizzare questo classificatore si ottimizza la cross entropy.

$$L = -\sum_i y_i^{true} \log(y_i^{pred})$$

Questa cosa equivale in modo lasco a quello che fa un neurone.

5.1 Neurone

Un neurone è costituito da:

- **Nucleo**: è la cellula vera e propria.
- **Dendriti**: connettono il nucleo con l'esterno.
- **Sinapsi**: gelatina in cui sono immersi i dendriti. Hanno lo scopo di amplificare i segnali che arrivano ai dendriti. Il nucleo poi valuta il segnale.

Il segnale può essere:

- **Forte**: quindi il neurone è attivo e l'output è un segnale elettrico.
- **Debole**: quindi il neurone è spento e non c'è nessun output.

5.1.1 Dal punto di vista matematico

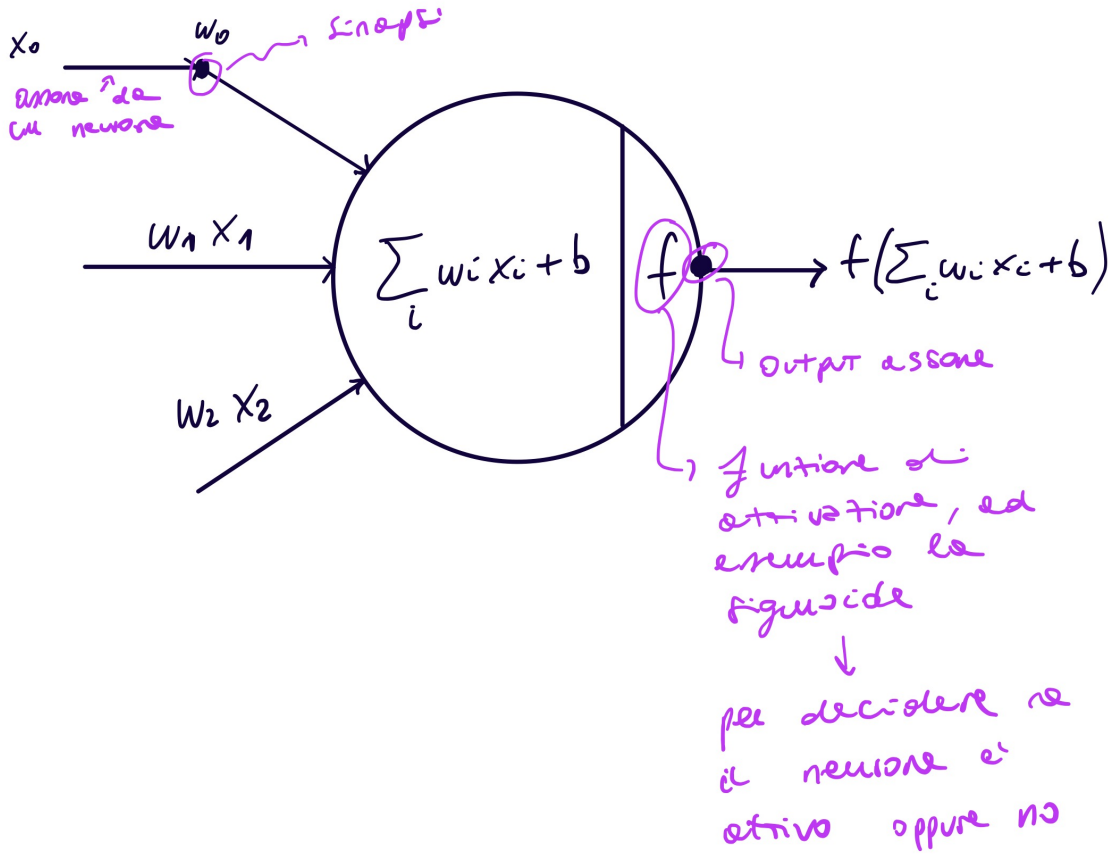


Figura 5.1: Neurone dal punto di vista matematico

Ogni neurone può avere input multipli. Un singolo neurone può essere utilizzato per implementare un classificatore binario. Quando una cross entropy loss è applicata all'output del neurone possiamo ottimizzare un binary softmax classifier.

5.1.2 Funzione di attivazione

Si possono utilizzare diverse funzioni di attivazione:

Sigmoide

$$\sigma(x) = \frac{1}{a + e^{-x}}$$

Se uso la sigmoide, il neurone diventa esattamente una Logist Regression.

Ad oggi la sigmoide è la meno utilizzata perchè:

1. L'output non è centrato sull'ingresso (se 0 in input, dà 0.5 in output).

2. Satura e uccide il gradiente, quindi da un certo punto in poi le uscite sono uguali indipendentemente dagli ingressi. Si perdono dunque informazioni importanti.

Per risolvere il primo problema, si può utilizzare la funzione $tanh = 2\sigma(x) - 1$ che centra la sigmoide sullo 0. Anchè questa funzione però satura e uccide il gradiente.

ReLu

$$f(x) = max(0, x)$$

La ReLu non satura mai. In uscita dà un valore di score (se x>0) non una probabilità. È la funzione di attivazione più utilizzata nei neuroni intermedi (non in quelli terminali).

5.1.3 Perchè utilizzare funzioni d'attivazione non lineari

La composizione di funzioni lineari è una funzione lineare. Senza funzioni d'attivazione non lineari, le reti neurali si ridurrebbero ad una singola Logistic Regression.

Supponiamo di avere la seguente funzione:

$$f(x) = \sigma\left(W_2\sigma(W_1x)\right)$$

Se eliminassimo la funzione non lineare avremmo:

$$W_2W_1x = Wx$$

Che è ancora lineare. La funzione d'attivazione evita quindi la moltiplicazione tra matrici.

5.2 Rete Neurale

Una rete neurale è la combinazione di neuroni organizzati in strati (layer) dove l'output del neurone allo strato precedente diventa l'input di un neurone allo strato successivo.

Il numero di parametri è alto.

5.2.1 Forward Propagation

La Forward Propagation è il processo di calcolo degli output della rete dati gli input.

"È il processo con cui metto i valori di ingresso e guardo lo score di uscita."

5.2.2 Potere Rappresentazionale

Le reti neurali, con almeno uno strato nascosto (hidden layer), possono rappresentare, a meno di ϵ, qualsiasi funzione continua esistente. Sono approssimatori universali. Nella pratica, più strati si hanno meglio si approssima.

5.2.3 Settare gli iperparametri

Una rete neurale ha una serie di iperparametri:

1. Quanti strati.

2. Quanti neuroni per strato.

Più neuroni può significare una migliore capacità della rete e quindi una migliore approssimazione ma può anche causare overfitting.

In generale, le reti vengono fatte grandi il più possibile e l'overfitting viene prevenuto tramite regole di regolarizzazione.

5.2.4 Loss Function

È la funzione obiettivo e misura quanto stiamo classificando bene.

In questo caso la funzione di Loss è arbitraria e dipende dal problema. La forma della funzione è questa:

$$L(\theta; x, y) = \frac{1}{N} \sum_i L_i(\theta; x_i, y_i) + \lambda \Omega(\theta)$$

Il **data term** è:

$$\frac{1}{N} \sum_i L_i(\theta; x_i, y_i)$$

è calcolato come la media degli esempi individuali e misura quanto sono buone le predizioni del modello.

Il **termine di regolarizzazione** è:

$$\lambda \Omega(\theta)$$

serve per prevenire l'overfitting. Chiede al modello di risolvere il problema con la minor capacità possibile, quindi con meno parametri possibili.

Esempi di termini di regolarizzazione

Esempi di termini di regolarizzazione sono:

1. Norma L^2: $\Omega(\theta) = \sum \theta^2$

2. Norma L^1: $\Omega(\theta) = \sum |\theta|$, ha come derivata 1 se $\theta \neq 0$ e 0 se $\theta = 0$. Questa somma viene minimizzata quando il numero di parametri uguali a 0 è massimo. La norma L^1 introduce il concetto di sparsità, ovvero un gran numero di parametri uguali a 0 causa una diminuzione di capacità.

Lo svantaggio dell'L^1 è che il modulo non è derivabile quindi devo metterci un if. Per ovviare a questo problema si utilizza la norma L^2 che è una versione rilassata del modulo (rinuncio al concetto di sparsità hard). La norma L^2 accetta di avere parametri con valori molto bassi

Esempi di Funzioni di Loss

Tra le varie loss che si possono utilizzare, tre sono famose:

- Per la **Classificazione**:

  1. **Hinge Loss**: si usa quando vogliamo essere sicuri della classe.

  $$L_i = \sum_{j \neq y_i} \max(0, f_j - f_y + 1)$$

  2. **Softmax loss**: è praticamente la cross entropy.

  $$L_i = -\log\left(\frac{e^f_{y_i}}{\sum_j e^f_j}\right)$$

- Per la **Regressione**:

  1. **Mean squared error**:

  $$L_i = \|f - y_i\|_2^2$$

5.2.5 Imparare i parametri

Per imparare i parametri dobbiamo risolvere questo problema:

$$\theta^* = \arg\min \left(\frac{1}{N} L_i(\theta; x_i, y_i) + \lambda \Omega(\theta) \right)$$

Per farlo si utilizza l'algoritmo di **Backpropagation**. È l'algoritmo con cui le reti neurali riescono a svolgere localmente, cioè layer per layer, tutta la parte necessaria per fare training.

Per fare Backpropagation bisogna utilizzare la regola della catena:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \quad \text{dove:} \quad z = f(g(x)) \;\; \text{e} \;\; y = g(x)$$

e procede a ritroso rispetto al flusso di calcoli effettuati per calcolare la perdita stessa.

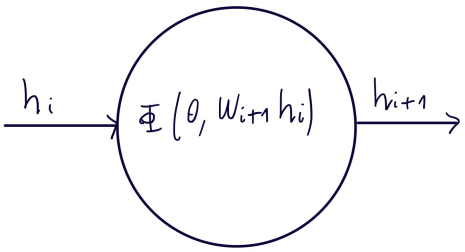
La backpropagation è un processo locale. I neuroni non conoscono la topologia della rete di cui fanno parte. Quindi, per far funzionare la backpropagation, ogni neurone deve essere in grado di calcolare solo due cose:

1. Derivata del suo output rispetto ai suoi pesi:

$$\frac{\partial o}{\partial \overline{w}} = \frac{\partial h_{i+1}}{\partial \overline{W_{i+1}}} = [\overline{x}]$$

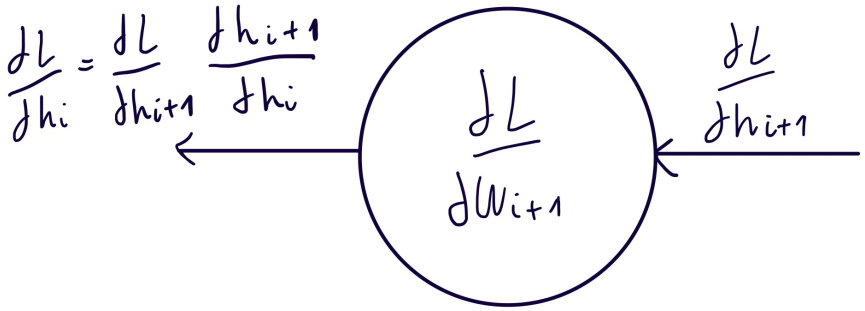
2. Derivata del suo output rispetto al suo input:

$$\frac{\partial o}{\partial \overline{x}} = \frac{\partial h_{i+1}}{\partial \overline{h_i}} = [\overline{w}]$$



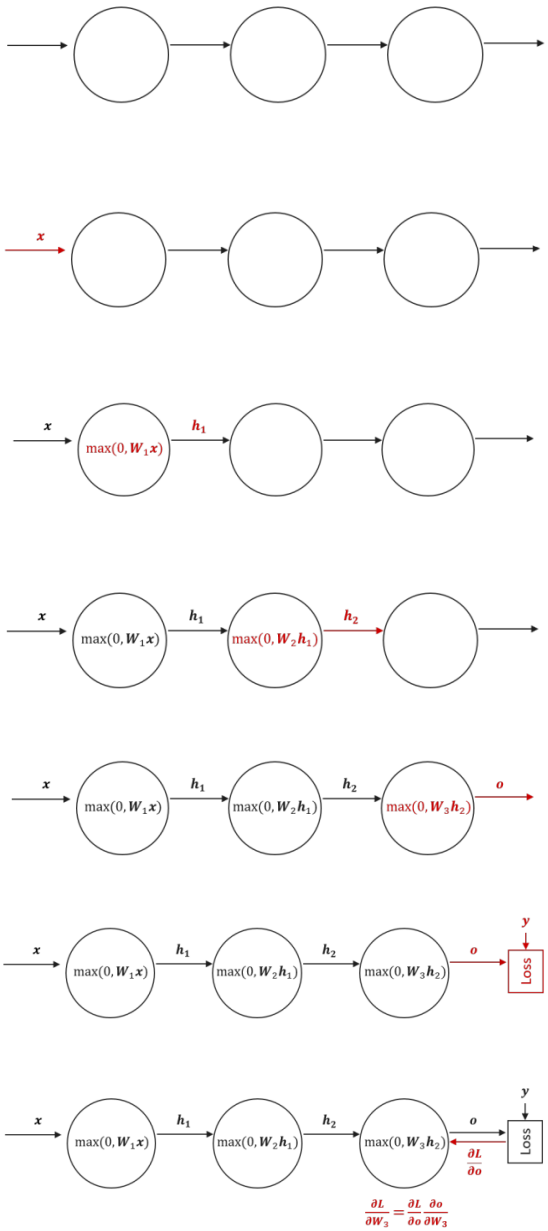
Durante la backpropagation, il neurone scoprirà l'importanza del valore del suo output sull'output dell'intera rete, cioè riceverà:

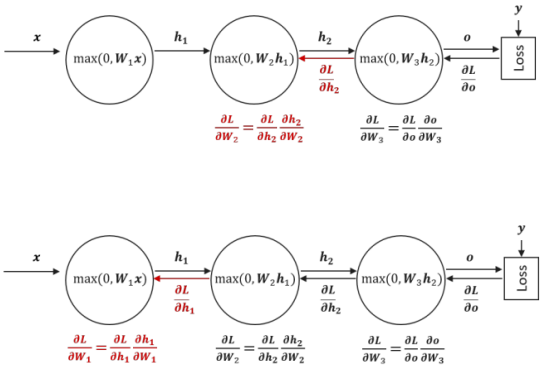
$$\frac{\partial L}{\partial h_{i+1}}$$



Questo gradiente viene quindi concatenato al gradiente locale e passato ai neuroni precedenti per continuare il processo di backpropagation.

5.2.6 Esempio di backpropagation





5.2.7 Inizializzare i pesi

Quando si inizializzano i pesi nella rete neurale, risulta che l'inizializzazione è casuale molto importante per rompere la simmetria.

Infatti, se tutti i pesi fossero inizializzati allo stesso valore, tutti i neuroni calcolerebbero lo stesso output, lo stesso gradiente e porterebbero allo stesso update. Quindi una pratica comune è inizializzare i pesi a numeri bassi centrati sullo zero a caso e tutti i bias a zero.

5.2.8 Regolarizzazione

Se c'è il termine di regolarizzazione, alla derivata della Loss bisogna aggiungere la derivata del termine di regolarizzazione, che dipende solo dal set di parametri che stiamo considerando. Ci sono anche tecniche di regolarizzazione non esplicite.

5.2.9 Dropout

È un layer con l'obiettivo di evitare l'overfitting. Durante il training arbitrariamente, con probabilità p, spegne un neurone. Quindi quando faccio forward, ogni neurone ha probabilità p di essere spento. Nel test il dropout viene disattivato.

5.2.10 Data Augmentation

Si trasformano i dati di input per aumentare la dimensione del dataset. In questo modo l'overfitting diventa difficile perchè imparare a memoria un dataset grande è complicato.

5.2.11 Early stopping

Consiste nello stoppare il training quando la loss sulla validation cessa di decresce. È un processo subottimo perchè il training cessa prima di imparare i parametri ottimi.

5.3 Reti Neurali Convolutive (CNN)

Sono molto simili alle reti neurali ordinarie:

- Sono costituite da neuroni che hanno pesi e bias imparabili.
- Ogni neurone riceve un input, performa un prodotto scalare e, opzionalmente, lo fa seguire da una non-linearità.
- L'intera rete esprime ancora una singola funzione differenziale.

Assumo che l'input siano immagini, o cumunque dati che hanno un supporto spaziale.

L'obiettivo delle CNN è quello di ridurre il numero di parametri supponendo che l'input siano immagini.

Le immagini hanno tre dimensioni:

1. **Altezza** (H)
2. **Larghezza** (W)
3. **Numero di canali** (C): possiamo vedere un'immagine come tre immagini, una in scala di verde, una di rosso e una di blu.

Le reti convolutive sono considerate dei black-box perchè non sappiamo cosa succede nei layer intermedi per via del numero delle convoluzioni, filtri, ecc.

5.3.1 Obiettivo delle CNN

Da un'immagine si vuole diminuire l'altezza e la larghezza e aumentare il numero di canali fino ad arrivare ad un caso estremo dove $W, H = 1$ e C è alto. Una volta arrivato a questo punto ho rappresentato l'immagine con 1024 valori.

Si elaborano dunque immagini attraverso operazioni derivabili, fino ad estrarre un

vettore di feature.

5.3.2 Layer Convolutivo

Il layer convolutivo è il blocco operativo di una rete convolutiva. Partendo da un certo numero di W,H,D ha l'obiettivo di arrivare ad un nuovo numero di W,H,D dove generalmente W,H diminuiscono e D aumenta.

Un layer convolutivo è equipaggiato da un set di filtri si possono imparare. Un **filtro** è un oggetto di forma quadrata, una griglia, che ha sempre larghezza e altezza dispari (3x3,5x5,...) perchè deve esistere il concetto di centro (casella di centro). Se istanzio un filtro 3x3 istanzio nove parametri. Il numero di filtri che istanzio corrisponde al numero di canali in uscita dall'operazione della convoluzione. Ad esempio, istanziando 6 filtri ottengo 6 nuove immagini in uscita dal layer convolutivo.

Le grandezze fondamentali in gioco sono dunque:

1. **Numero di filtri**: corrisponde al numero di canali in uscita.

2. **Dimensione di un filtro k**

3. **Stride**: il filtro scorre sull'immagine, lo stride indica di quanti pixel mi sposto quando applico un filtro. È responsabile della riduzione di W e H al layer successivo.

Un altro elemento che si può specificare in un layer convolutivo è il **padding** P. Serve perchè non è detto che, selezionato un filtro, questo stia nell'immagine un numero finito di volte. Il Padding è una cornice di valori fasulli che metto vicino all'immagine quando voglio applicare il filtro anche se sono vicino ai bordi.

5.3.3 Strategie di Padding

Il Padding non riduce W e H.

Ci sono tre strategie di padding:

1. **0-padding**: ci si mette degli 0.

2. **Padding same**: valori dell'ultima riga copiata.

3. **Padding mirror**: tutte le righe dall'ultima indietro.

C'è il rischio che la rete impari anche nel padding quindi bisogna fare in modo di non creare discontinuità tra immagine e padding.

5.3.4 Operazione di convoluzione discreta

L'operazione di convoluzione discreta è una combinazione di somme e moltiplicazioni della sorgente I e del kernel K:

$$F(i,j) = \sum_m \sum_n I(i-m, j-n)k(m,n)$$

Ho il filtro e lo vado a superimporre ad una porzione dell'immagine. Prendo il valore dell'immagine e lo moltiplico per il valore del filtro che corrisponde a quel determinato pixel. Poi sommo tutti questi valori eli sostituisco nella posizione centrale del filtro.

Molto spesso questa operazione è implementata come cross-correlazione flippando il kernel: non devo gestire coordinate positive o negative e ho un prodotto riga per colonna.

$$F(i,j) = \sum_m \sum_n I(i+m, j+n)k(m,n)$$

Quindi non si ospita più nel centro del filtro ma nel pixel in alto a sinistra.

Per aumentare la D questa cosa viene fatta con kernel diversi.

Localmente i neuroni di una CNN realizzano l'operazione

$$\sum_i w_i x_i + b$$

Quindi è un piccolo perceptron che scorre su tutta l'immagine. Ad ogni valore di uscita nell'immagine successiva corrispondono nell'immagine precedente kxk valori. Tuttavia, nelle CNN i neuroni sono connessi solo localmente ai volumi di input. La

regione che ogni valore di output di un'immagine d'uscita da una convoluzione vede nell'immagine precedente è detta **receptive field**. Preso un layer qualunque e preso un qualunque valore in una qualunque immagine di uscita di quel layer, la regione nell'immagine iniziale che è servita per calcolare quel valore è il receptive field. Tra due layer successivi ha la stessa dimensione del kernel, tra ad esempio il primo e il quarto è invece molto più grande (piramide). il receptive field è moltiplicativo.

Un kernel risponde con un valore alto quando il pattern dentro al kernel si sovrappone bene al pattern che stiamo cercando nell'immagine. Quando siamo al layer successivo il nuovo kernel lavora sui nuovi valori. È la rete neurale che impara i pattern da applicare.

La convoluzione è un'operazione translation invariant.

5.3.5 Canali

I canali sono filtrati da filtri indipendenti K^c e sommati insieme:

$$F(i,j) = \sum_c \sum_m \sum_n I(i+m, j+n, c) K^c(m,n)$$

Notare che si ha un kernel per ogni canale.

5.3.6 Numero di parametri imparabili

Ogni layer ha un certo numero di parametri che dipende dal numero di filtri (che è uguale al numero di canali di output), K=dimensione del kernel, e il nuemero di canali in ingresso.

$$tot\_param\_layer = N * K * K * C_1 + N$$

$$tot\_param\_filtro = K * K * C_1 + 1$$

5.3.7 Pooling Layers

In una rete convolutiva ci possono essere anche layer il cui unico scopo è quello di ridurre H e W. Questi layer sono detti layer di pooling.

L'operazione di pooling non ha parametri. Consiste nello far scorrere una finestra kxk pari (2x2, 4x4,...) sull'immagine e applicare ai pixel nella finestrella un'operazione.

Lo stride è come nella normale convoluzione. Le operazioni che si possono fare sono operazioni di base come ad esempio min, max, *average*. La trasformazione, in questo caso, non è imparata.

Si fa pooling perchè:

1. Si riduce i costi computazionali perchè le immagini sono più piccole.

2. Man mano che riduco le immagini il receptive field aumenta.

3. Previene l'overfitting.

4. Guadagno robustezza sulla localizzazione delle feature.

Ci sono casi in cui il pooling non conviene, come ad esempio nei problemi di segmentazione.

5.3.8 Activation Layers

Gli activation layers calcolano la funzione di attivazione non lineare in modo elementare sul volume di input. Le funzioni di attivazioni sono le stesse viste in precedenza: sigmoide, tanh e ReLu. La ReLu vince.

L'activation layer non cambia nessuna dimensione dell'immagine.

5.3.9 Calcolare la dimensione dell'immagine di output

Supponiamo di avere un'immagine di dimensioni H_1, W_1, C_1. L'output di un layer con N filtri, kernel size K, stride S e zero-padding P è un volume di forma H_2xW_2xC_2 dove:

$$H_2 = (H_1 - K + 2P)/S + 1$$

$$W_2 = (W_1 - K + 2P)/S + 1$$

$$C_2 = N$$

5.3.10 Output del pooling

$$H_2 = (H_1 - K)/S + 1$$

$$W_2 = (W_1 - K)/S + 1$$

$$C_2 = C_1$$

5.3.11 Advanced CNN Architecture

Architetture CNN più avanzate hanno recentemente dimostrato di performare meglio delle tradizionali con architettura del tipo (conv→ReLu→pooling).

Queste architetture generalmente presentano diverse tipologie di grafo e altre strutture di connettività.

5.3.12 VGG

L'input sono immagini 224x224(x3). Ad esse viene tolto il valore medio del training set. In questo modo si centrano i punti rispetto al sistema di riferimento.

I filtri sono tutti 3x3 e lo stride è 1.

Lo spatial pooling è 2x2 ed $S = 2$. Quindi dimezza altezza e larghezza.

Per l'attivazione utilizza le ReLu alla fine di ogni layer.

I layer fully connected presentano 4096 neuroniognuno dei quali è seguito da una ReLu. L'ultimo fully connected Layer ha 1000 neuroni ed è seguito da un'attivazione softmax.

VGG 16 ha 138 milioni di parametri imparabili. Ogni immagine richiede 93 MB di memoria per la forward. Si deve sempre salvare gli output dei layer precedenti. Il 70% dei parametri imparabili sono condensati nei fully connected layer.

5.3.13 Visualizzare le attivazioni

Serve per avere un'idea se di fatto il training sia andato a buon fine o meno.

5.3.14 Visualizzare i filtri

Vale solo per i primi layer. Visualizzare i filtri significa vedere che tipi di pattern la rete sta imparando.

I filtri sono buoni se:

1. Non sono correlati, cioè ogni filtro impara cose diverse.

2. Hanno una struttura (hanno direzioni).

3. Non sono rumorosi.

5.3.15 Partially occluding the images

Serve per vedere quali porzioni dell'immagine di ingresso la rete ha usato per prendere una determinata decisione.

5.3.16 Transfer Learning

Nella maggior parte dei casi non si traina mai una rete convolutiva da zero ma si cerca di utilizzare i primi layer di reti convolutive note. I primi layer hanno infatti filtri generali che vanno bene per tutte le immagini.

Decidere quale porzione della rete ri-trainare è una scelta importante che influenzerà le performance finali del modello. Due fattori influenzano la decisione:

1. Grandezza del dataset.

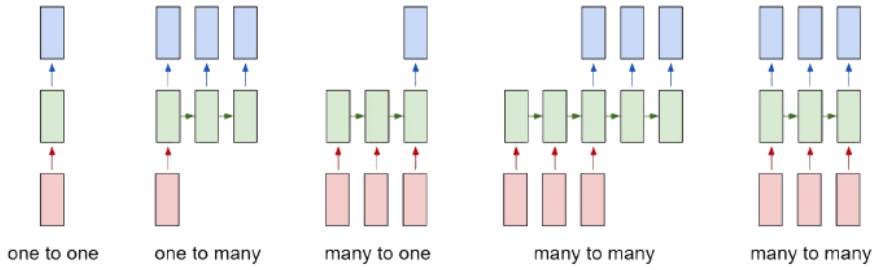
2. Similarità dei dati del mio dataset e di quello su cui è stato trainato il modello.

5.4 Recurrent Neural Network

Si utilizzano i tipi di dato temporale. Per tipo di dato temporale si considere che gli input del modello abbiano, tra gli stessi valori che rappresentano il vettore di feature, anche il tempo.

L'input non è un singolo vettore di feature statico ma è una sequenza di vettori di feature in un certo ordine (sequenza ordinata).

Le reti neurali ricorrenti non sono una struttura ma sono un'API quindi possono dare origine a tante strutture differenti.



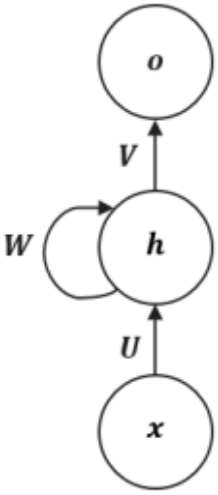
- Rosso: input.

- Verde: unità ricorrente.

- Blu: output.

5.4.1 Cella ricorrente di tipo Vanilla

È la cella ricorrente di base.



- **o**: uscita.
- **V**: set di parametri che tratta l'uscita (si imparano).
- **W**: set di parametri che dallo stato precedente porta quello corrente (si imparano).
- **h**: è lo stato, ossia la rappresentazione delle informazioni durante il tempo che tiene conto sia dell'ingresso all'istante temporale t che dello stato all'istante $t-1$. È ciò che tratta la ricorrenza.
- **U**: set di parametri che tratta l'ingresso.
- **x**: feature vector all'istante temporale t.

Le funzioni di update sono:

$$\begin{cases} h^{(t)} = \Phi(Wh^{(t-1)} + Ux^{(t)}) \\ o^{(t)} = Vh^{(t)} \end{cases}$$

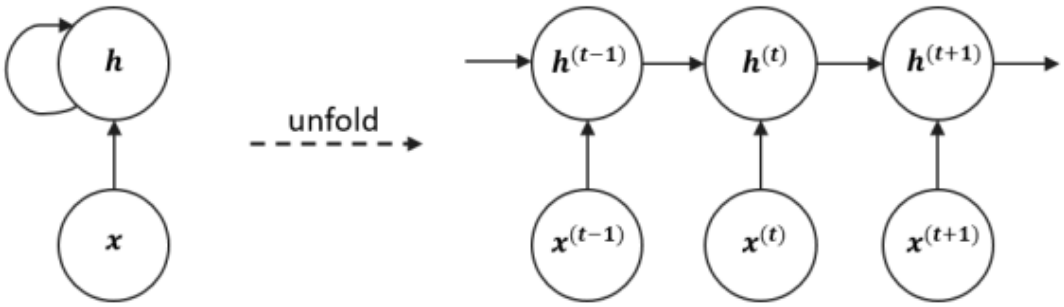
Lo stato h rappresenta un riassunto di tutto quello che la rete ha imparato fino allo stato temporale corrente.

Visto che una sequenza di input di arbitraria lunghezza $(x^1, x^2, x^3, \ldots, x^t)$ è mappata in un vettore di grandezza fissa h^t, si ha una perdita di informazioni (processo Markoviano).

5.4.2 Sviluppo del grafo computazionale

Quando si deve applicare un oggetto di questo tipo, questa struttura può andare avanti un numero di step indefinito. Quindi si srotola in funzione del numero di time step.

In modo formale si dice che: un grafo computazionale ricorrente può essere srotolato in una sequenza di grafi computazionali con una struttura ripetitiva.



$$h^{(t)} = f(h^{(t-1)}, x^{(t)}, \theta)$$

La cella ricorrente non dipende quindi da un numero finito di input perchè viene srotolata al momento in funzione di quante x ha la sequenza.

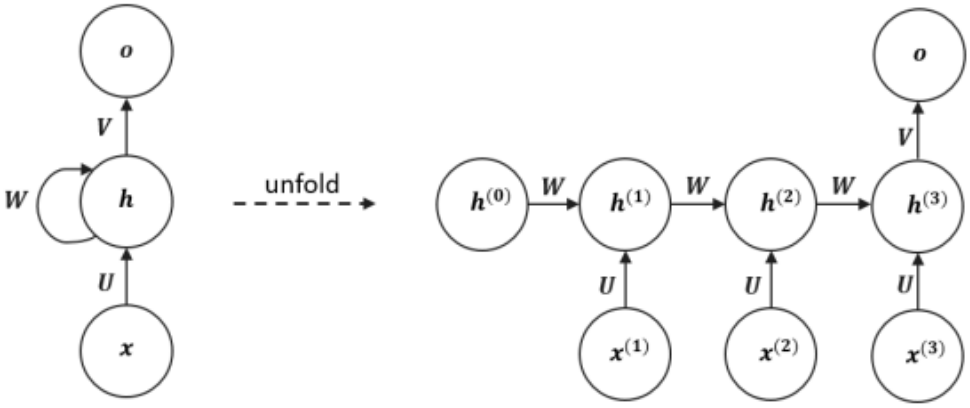
W, V, U sono unici per tutti gli istanti temporali.

5.4.3 Backpropagation through time

La backpropagation non è quella tradizionale perchè bisogna tener conto della natura ricorrente della rete.

Poiché il gradiente concettualmente fluisce all'indietro attraverso il tempo invece che attraverso i livelli, questo algoritmo è solitamente definito come backpropagation through time (BPTT).

Supponiamo di avere la cella ricorrente e di fare il training consequenze lunghe 3. Il srotolato diventerebbe un oggetto di questo tipo:



La loss function si misura in O.

Dobbiamo calcolare la derivata della loss rispetto ad un certo set di parametri e poi possiamo applicare il gradient descent.

Nella prima non devo considerare il tempo.

$$\frac{\partial L}{\partial V} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial V}$$

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial h^{(3)}}\sum_{k=0}^{3}\left(\frac{\partial h^{(3)}}{\partial h^{(k)}}\frac{\partial h^{(k)}}{\partial W}\right)$$

$$\frac{\partial L}{\partial U} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial h^{(3)}}\sum_{k=0}^{3}\left(\frac{\partial h^{(3)}}{\partial h^{(k)}}\frac{\partial h^{(k)}}{\partial U}\right)$$

Si può notare che:

- $\frac{\partial L}{\partial V}$ dipende solo dallo stato corrente.

- $\frac{\partial L}{\partial W}$ e $\frac{\partial L}{\partial U}$ dipendono da tutti gli stati precedenti.

5.4.4 The challenge of long-term dependencies

Guardando più da vicino, vediamo che i termini $\frac{\partial h^{(t)}}{\partial h^{(k)}}$ devono essere anch'essi calcolati tramite la regola della catena. Ad esempio, possiamo ottenere:

$$\frac{\partial h^{(3)}}{\partial h^{(1)}} = \frac{\partial h^{(3)}}{\partial h^{(2)}} \cdot \frac{\partial h^{(2)}}{\partial h^{(1)}}.$$

Si scopre che quando si utilizzano attivazioni tanh e sigmoide, la norma L^2 di queste matrici di Jacobiani è limitata superiormente da 1 e $1/4$ rispettivamente. Pertanto, è facile che si finisca per moltiplicare numeri sempre più piccoli fino a quando i gradienti diventano zero.

Questo problema è noto come il **problema del gradiente che scompare** (vanishing gradient problem) e causa seri problemi quando si cerca di apprendere dipendenze a lungo termine nelle sequenze di input, poiché i contributi dei "passi lontani" diventano zero.

A seconda dei parametri della rete e della scelta delle funzioni di attivazione, può insorgere il problema opposto. Questo è chiamato il **problema del gradiente esplosivo** (exploding gradient problem) e si verifica quando i gradienti diventano sempre più grandi fino a che problemi numerici distruggono il processo di ottimizzazione.

Entrambi i problemi possono essere mitigati attraverso una corretta inizializzazione dei pesi, una scelta accurata delle funzioni di attivazione e il **clipping dei gradienti**.

Questi problemi possono verificarsi anche nelle reti feedforward profonde; tuttavia, sono più comuni nelle architetture ricorrenti perché questi modelli sono solitamente molto profondi (solitamente, profondi quanto la lunghezza della sequenza di input).

5.4.5 Architetture ricorrenti avanzate

Le **Long Short-Term Memory** (LSTM) e le **Gated Recurrent Unit** (GRU) sono architetture ricorrenti più complesse che sono state proposte per superare i problemi nel flusso del gradiente e per facilitare l'apprendimento delle dipendenze a lungo termine grazie all'introduzione di meccanismi di gating apprendibili.

Complicano il modo in cui si calcola lo stato.

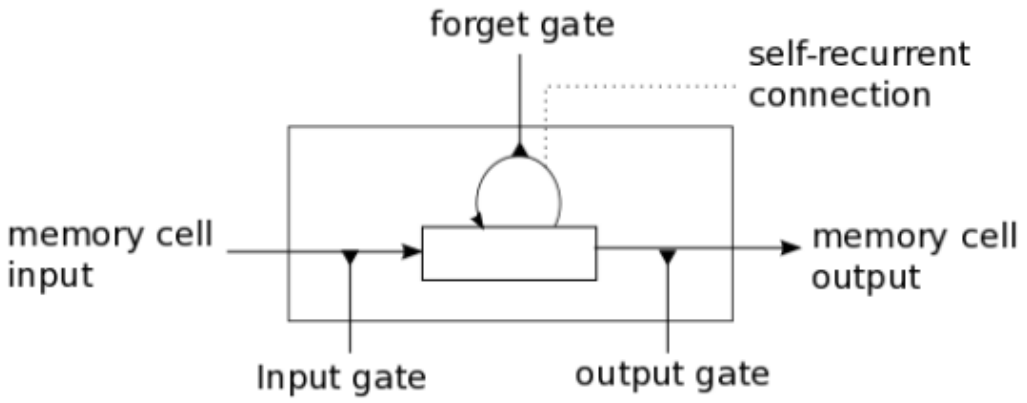


Figura 5.2: Cella LSTM

Le equazioni sono molto più complicate:

$$\begin{cases} i = \sigma\left(x^{(t)}U_i + s^{(t-1)}W_i\right) \\ f = \sigma\left(x^{(t)}U_f + s^{(t-1)}W_f\right) \\ o = \sigma\left(x^{(t)}U_o + s^{(t-1)}W_o\right) \\ g = tanh\left(x^{(t)}U_g + s^{(t-1)}W_g\right) \\ c^{(t)} = c^{(t-1)} \odot f + g \odot i \\ s^{(t)} = tanh(c^{(t)}) \odot o \end{cases}$$

I primi tre sono rispettivamente i gate di input, forget e output. Ogni gate ha la stessa dimensione dello stato nascosto. I gate vengono moltiplicati elemento per elemento con altre funzioni dell'LSTM, agendo così come interruttori continui e differenziabili grazie alla funzione di attivazione sigmoide.

Il gate di input e quello di output controllano rispettivamente quanto dell'input deve essere lasciato passare e quanto dello stato interno deve essere esposto all'esterno. Al contrario, il gate di forget controlla quanto della memoria del passo temporale precedente deve essere sovrascritta.

g calcola quello che potrebbe essere intuitivamente descritto come uno stato candidato. Infatti, l'equazione è praticamente la stessa che abbiamo dell'architettura RNN tradizionale. Tuttavia, la quantità di influenza di g sulla cella di memoria LSTM è controllata dal gate di input i.

La penultima equazione calcola l'aggiornamento per la cella di memoria c. Il gate di forget controlla quanto della memoria dei passi precedenti $c^{(t)}$ deve essere mantenuto. Il gate di input supervisiona la quantità di stato appena calcolato g che deve fluire nella memoria.

Infine, l'ultima equazione calcola lo stato nascosto di output dalla memoria corrente. Il gate di output regola la quantità di informazione che deve essere esposta agli strati successivi.

Capitolo 6

Unsupervised Learning

Per avere la supervision e i dati devono essere annotati e questa è una penalità. Se esistessero delle tecniche per estrarre informazioni dai dati senza avere le annotazioni, sarebbero molto utili.

Perchè fare apprendimento non supevisionato: la quantità di dati non annotati è diversi ordine di grandezza più grande di quella di dati annotati. Inoltre, all'interno dei dati, se rappresentano la stessa cosa o campionamenti dello stesso processo, esistono strutture che sono ricorrenti e che dipendono da quel processo, a prescindere dalla presenza di un'etichetta. Le reti neurali soffrono di un peccato originale, cioè overfittano o underfittano tanto. L'apprendimento non supervisionato può essere uno strumento per non partire con una rete da zero. Potrei all'inizio avere una qualche straegia per addestrare la rete in modo non supervisionato e poi fare solo il fine tuning, ossia la classificazione fine usando il dataset annotato.

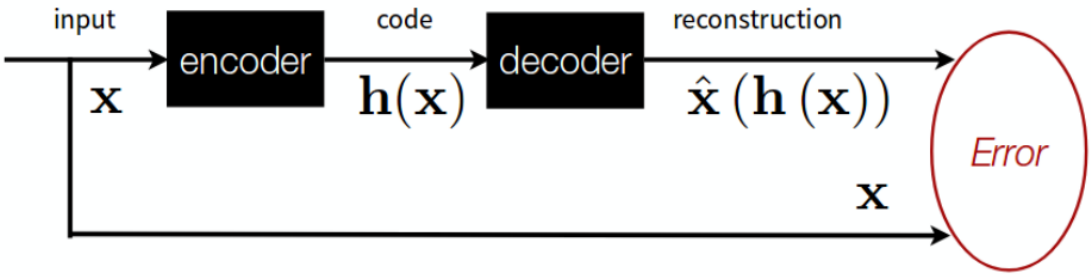
manifold assumption: i dati del problema non spaziano l'intero spazio ma solo una figura geometrica in cui ci sono solo le configurazioni valide. Se trovo questa struttura avrò sempre a disposizione configurazioni valide. Una task importante è quindi trovare un manifold buono (semplice e non deve dipendere dalle task). Per trovarlo si può utilizzare ad esempio la PC ma è un metodo debole perchè è un metodo lineare e quindi permette solo di ruotare assi e buttare via degli assi.

Se volessi implementare la PCA sulle reti neurali, rilassando alcuni vincoli potri implementarla così: considerare una serie di layer che prendono l'input e lo mappano su un certo numero di neuroni più basso dei neuroni di input (come nella PCA). Poi però voglio anche tornare indietro, quindi partendo dalle nuove feature vorrei ricostruire l'input. Questa cosa non la posso fare con la PCA ma la posso fare con le reti neurali. Un'altra differenza con la PCA è che in questo caso, nello spazio delle feature ridotte, non ho la sicurezza che queste siano ortogonali tra loro.

6.1 Autoencoder

Un autoencoder è una rete neurale di tipo free forward (convolutiva o densa ma non ricorrente) che è addestrata per prendere l'input, processarlo con alcuni layer, ottenere una rappresentazione ridotta detta **codice** e poi ricostruire l'input attraverso un'altra rete (di decodifica) a partire dal codice.

6.1.1 Struttura generale di un autoencoder



Siccome il metodo non è supervisionato, non ho le label. Quindi la Loss agisce confrontando la ricostruzione con l'input. L'obiettivo è quello di avere la ricostruzione il più vicino possibile all'input.

Come tutte le reti neurali, l'autoencoder può overfittare. L'autoencoder quando overfitta impara la funzione identità. Quindi impara ad usare tutti i neuroni per prendere l'input e copiarlo in output. Questo succede quando l'autoencoder ha troppa capacità, ad esempio quando il codice nascosto ha dimensione molto maggiore dell'input. In tal caso, il codice non rappresenterebbe nulla.

Servono delle strategie per evitare l'overfitting

6.1.2 Autoencoder undercompleto

Per evitare di copiare in output l'input, decido che nel layer di bottleneck, ossia il layer che collega encoder a decoder, devo avere un numero di neuroni molto minore del numero di neuroni di input. Lo svantagio è che l'autoencoder potrebbe utilizzare il bottleneck come un indice per trovare i valori.

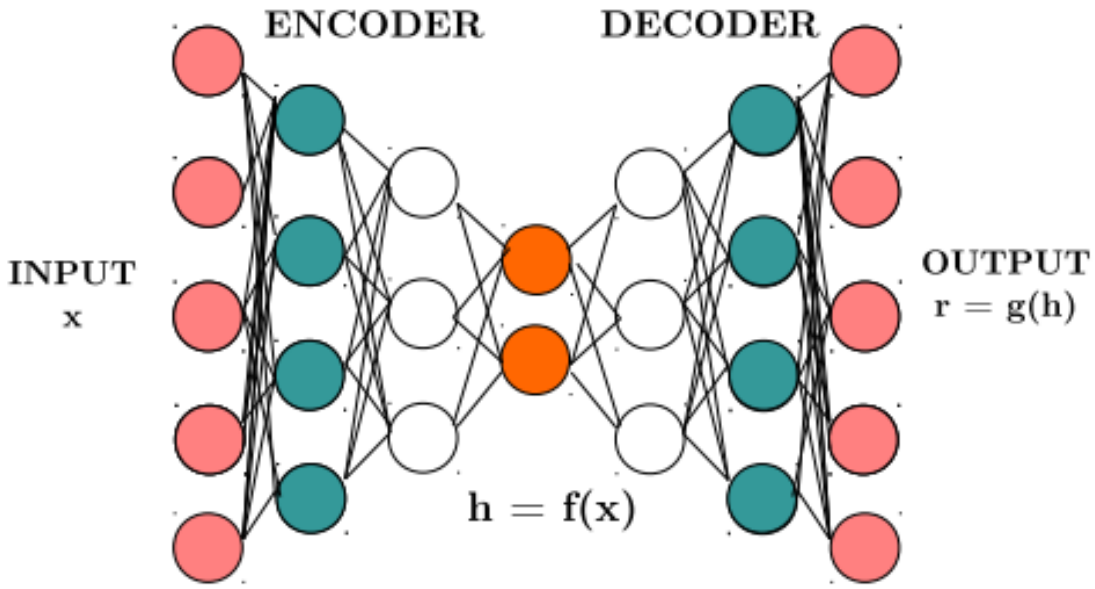


Figura 6.1: Schema dell'autoencoder undercompleto

Il processo di learning mira a minimizzare:

$$L(x, g(f(x))$$

L è la funzione di loss che penalizza $g(f(x))$ per essere dissimile da x.

L'autoencoder undercompleto può essere considerato come una più potente generalizzazione non lineare del PCA

Se all'encoder e al decoder viene data troppa capacità, l'autoencoder potrebbe imparare a copiare direttamente l'input in output senza estrarre informazioni importanti.

6.1.3 Un principio del Deep Learning

- Se il modello ha troppi pochi parametri, tenderà ad underfittare.
- Se il numero di parametri è troppo alto, tenderà ad overfittare.

Se stiamo lavorando con un autoencoder, ciò si traduce nell'apprendimento della funzione identità.

Come selezionare la giusta capacità di un modello di apprendimento automatico? Un principio di Deep Learning raccomanda di fornire abbastanza capacità per adattarsi a funzioni molto complesse e, allo stesso tempo, di vincolare il modello affinché utilizzi saggiamente il potere che ha a disposizione. Di solito la migliore performance deriva da un modello grande ben regolarizzato.

6.1.4 Autoencoder Regolarizzati

Per evitre che gli autoencoder imparino la funzione identità, vengono ristretti in modo da permettere loro di copiare solo parzialmente.

Gli autoencoder regolarizzati usano una funzione di loss che incoraggia il modello ad avere altre proprietà oltre all'abilità di copiare l'input in output. Visto che il modello è forzato a cercare e determinare quali aspetti dell'input prioritizzare per la copia, imparerà spesso importanti informazioni sui dati. Può catturare la struttura della distribuzione che genera i dati grazie all'opposizione tra la distribuzione di ricostruzione e il regolarizzatore.

6.1.5 Autoencoder Sparso

È un autoencoder regolarizzato. Ha il numero di neuroni di input pari alla dimensione del dato. Il numero di neuroni nel codice è superiore a quelli di input.

Un autoencoder sparso è un autoencoder il cui criterio di addestramento prevede una penalità aggiuntiva di sparseness sullo strato di codice h.

$$L_{SAE} = D(x, g(f(x))) + \lambda|h|$$

- $\lambda|h|$: termine di sparsità.

- h: uscita dei neuroni nel bottleneck

Grazie alla sparseness penalty, possiamo costruire una rappresentazione overcompleta per i dati di ingresso senza correre il rischio di imparare la funzione identità.

L'obiettivo è quello di ricostruire il meglio possibile ma evendo nel bottleneck il maggiorn numero di neuroni spenti.

Gli autoencoder sparsi sono utili per costruire dei dizionari. Ad esempio se dò in ingresso ad un autoencoder sparso delle immagini naturali, l'autoencoder cercherà di accendere lo stesso neurone per immagini della stessa categoria.

L'autoencoder sparso può scoprire rappresentazioni che spiegano la maggior parte o tutta l'informazione di un segnale con una combinazione di un numero ridotto di concetti elementari, ricorrenti e solo parzialmente correlati. Dato il segnale in ingresso, l'idea generale è di trovare una lista corta di termini che descrivono meglio l'ingresso, da un dizionario appreso. La nuova rappresentazione h ha la dimensionalità del dizionario, molto più alta della dimensionalità dell'ingresso (overcompletezza).

L'autoencoder sparso costruisce vettori di codice molto legati alla categoria e poco al contenuto. Quindi poco è poco legato alla semantica.

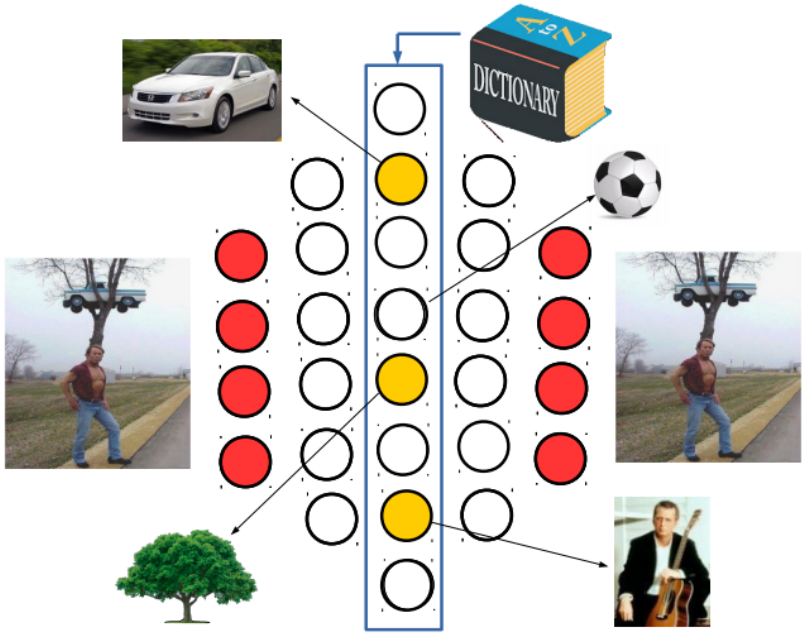


Figura 6.2: Autoencoder Sparso

6.1.6 Autoencoder Denoising

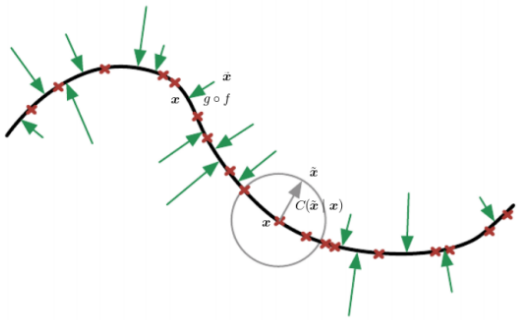
In input non viene data l'immagine originale ma l'immagine originale a cui viene aggiunto, pixel per pixel, del rumore gaussiano. Ciò viene fatto per forzare lo strato nascosto a scoprire caratteristiche più robuste ed evitare di imparare semplicemente la funzione identità.

Ciò che viene richiesto al decoder questa volta non è ricostruire l'input, ma ricostruire l'immagine senza rumore.

$$L_{DAE} = D(x, g(f(\hat{x}))) \quad \hat{x} \sim N(x, \sigma^2)$$

L'overfitting viene evitato perchè per l'autoencoder non è conveniente, in termini di loss function, copiare l'ingresso in uscita. Questo perchè l'ingresso è molto diverso dall'immagine con cui si vuole fare il confronto.

Un DAE è in grado di riportare $\hat{x}$ al punto sulla curva in cui risiede la versione originale di x. Impara un campo vettoriale che punta verso regioni di maggiore probabilità. Conoscendo per ogni punto vicino alla curva dove deve essere proiettato su di essa, un DAE ha implicitamente appreso la struttura della varietà sottostante responsabile del processo generativo dei dati.



Il training su immagini con rumore porta l'encoder e il decoder a imparare implicitamente la struttura della distribuzione di probabilità $p_{data}(x)$

6.1.7 Autoencoder Contrattivo

Anche nell'autoencoder contrattivo c'è una penalità di regolarizzazione. Questo autoencoder ricostruisce l'input dal codice e rende il codice insensibile all'input.

$$L_{CAE} = D(x, g(f(x))) + \lambda \left\| \frac{\partial f(x)}{\partial x} \right\|^2$$

- $\frac{\partial f(x)}{\partial x}$: è la derivata del codice rispetto all'input e deve essere il più bassa possibile. Quindi deve esserci poca differenza tra i due valori.

  Ciò incoraggia il modello a:

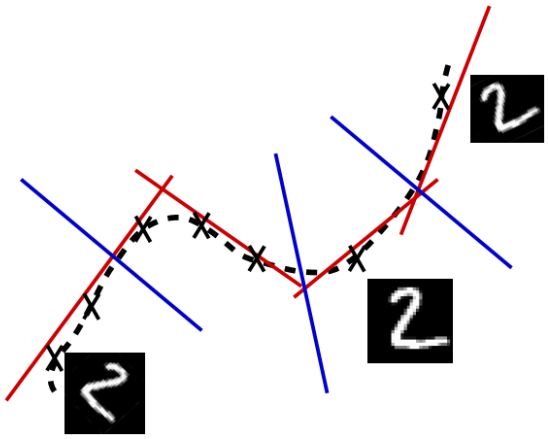
  1. Apprendere una funzione che non cambia molto quando x varia leggermente o è influenzato da piccole perturbazioni.

  2. Ridurre il numero di gradi di libertà effettivi della rappresentazione.

  3. Rappresentare solo le variazioni necessarie per ricostruire gli esempi di addestramento.

Vogliamo estrarre caratteristiche che riflettano solo le variazioni osservate nel set di addestramento.

Con la linea rossa indichiamo le direzioni delle variazioni a cui l'encoder deve essere sensibile per ricostruire correttamente l'input. Con la linea blu indichiamo le direzioni delle variazioni a cui l'encoder non deve essere sensibile, poiché non sono osservate nel set di addestramento.



Il nome contractive deriva dal modo in cui il CAE deforma lo spazio: poiché il CAE è addestrato a resistere alle perturbazioni del suo input, è incoraggiato a mappare un intorno dei punti di input in un intorno più piccolo di punti di output. Grazie al termine contractive, il modello diventa invariante a piccole perturbazioni vicino ai punti. Come accade per i denoising autoencoders, un CAE conosce i piani tangenti in ogni punto della varietà sottostante. In questo modo, può catturare la struttura della varietà dei dati.

6.2 Generative Modeling

Nei modelli generativi è possibile campionare i dati. Per poter campionare i dati, è necessario che il modello modelli effettivamente la distribuzione dei dati.

6.2.1 Generative Modeling e Autoencoder

Dall'autoencoder che ha imparato a rappresentare dei volti, vorrei poter campionare per ottenere altri volti che pur sempre appartengono alla distribuzione del dataset.

Il DAE, il SAE, e il CAE possono imparare implicitamente com'è fatta $p(x)$ ma non ne hanno un'esplicita modellazione. Quindi non si ha la facoltà di accedere a questa distribuzione e non si può tantomeno campionarla.

Una possibile soluzione è quella di chiedere all'autoencoder che il codice, quindi il vettore di feature che gioca nello spazio latente, segua una distribuzione di probabilità nota. Una volta che riusciamo a imparare questo vincolo nel codice allora possiamo anche campionare.

Ciò è più semplice a dirsi che a farsi perchè normalmente noi non accediamo alla distribuzione $p(x)$ ma alla distribuzione delle latenti $p(z|x)$ visto che sono definite su tutti i possibili dati.

6.2.2 Variational Inference

Invece di utilizzare distribuzioni a cui non posso accedere uso distribuzioni note (con forme che voglio) e chiedo che queste distribuzioni si avvicinino il più possibile alla distriuzione reale minimizzando la distanza tra distribuzioni.

Chiamo z il codice (set di variabili latenti) e decido che queste z sono campionabili da una probabilità a priori $p(z)$ (ad esempio una gaussiana multivariata) . La distribuzione congiunta tra input e codice è:

$$p(x, z) = p(x|z)p(z)$$

Usando Bayes posso calcolare la probabilità del codice data una certa x in input:

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)}$$

$$p(x) = \int_z p(x,z)dx$$

$p(x)$ è intrattabile. Devo approssimare $p(z|x)$ usando la variational inference. Quindi dico che $p(z|x)$ ha una certa forma parametrica che decido io.

Si approssima $p(z|x)$ con un'altra distribuzione che si chiama $q_\lambda(z|x)$ (λ sono i parametri della distribuzione). Adesso voglio minimizzare la distanza, ossia la divergenza, tra $p(z|x)$ e $q_\lambda(z|x)$.

Si minimizza la divergenza KL:

$$KL(q(z|x), p(z|x)) = \mathbb{E}_q \log(q(z|x)) - \mathbb{E}_q \log(p(x,z)) + \log(p(x))$$

È ancora intrattabile a causa di $p(x)$. Quindi definisco una quantità, detta ELBO (Empirical Lower Bound Objective) tale che:

$$ELBO_\lambda = \mathbb{E}_q \log(p(x,z)) - \mathbb{E}_q \log(q(z|x))$$

E si ha che:

$$\log(p(x)) = ELBO_\lambda + KL(q(z|x), p(z|x))$$

- $\log(p(x))$ è sempre non negativa.
- $KL(q(z|x), p(z|x))$ è sempre non negativa.

Minimizzare KL significa porre

$$\log(p(x)) = ELBO_\lambda$$

Analogamente, ciò significa massimizzare l'ELBO.

Dimostrazione

$$\log(p(x)) = \log \int_z p(x|z) = \log \int_z p(x|z) \frac{q(z|x)}{q(z|x)} = \log \mathbb{E}_q \frac{p(x,z)}{q(z|x)}$$

Usando la Disugaglianza di Jensen:

$$\mathbb{E}f(x) \leq f(\mathbb{E}x)$$

$$\log(p(x)) \geq \mathbb{E}_q \log fracq(z|x)q(z|x) = \mathbb{E}_q \log(p(x,z)) - \mathbb{E}_q \log(q(z|x)) = ELBO_\lambda$$

$$\log(p(x)) \geq ELBO_\lambda$$

Quindi massimizzando l'ELBO minimizzo implicitamente KL.

L'autoencoder generativo sfrutta questo discorso e cerca di massimizzare l'ELBO.

6.2.3 Come massimizzare l'ELBO

$$ELBO_\lambda = \mathbb{E}_q \log(p(x,z)) - \mathbb{E}_q \log(q(z|x)) = \mathbb{E}_q \log(p(x|z)p(z)) - \mathbb{E}_q \log(q(z|x)) =$$

$$= \mathbb{E}_q \left(\log(p(x|z)) + \log(p(z))\right) - \mathbb{E}_q \log(q(z|x))$$

Da cui:

$$ELBO_\lambda = \mathbb{E}_q(p_\phi(x_i|z)) - KL(q_\theta(z|x_i), p(z))$$

- ϕ e θ sono parametri della rete, rispettivamente decoder (osservo il codice e chiedo x) ed encoder (osservo x ed ottengo z). L'aspettazione è trattata utilizzando l'approssimazione di campo medio, assumendo un insieme di parametri λ per ogni punto dati.

- Il campo medio sostituisce l'integrale nell'aspettazione con una somma sul batch e considera una diversa $q(z \mid x_i)$ per ogni punto dati x_i.

6.2.4 Variational Autoencoder

Un variational autoencoder è una rete neurale allenata per massimizzare:

$$\mathbb{E}_{z\sim Q(z|x)}[\log P(x|z)] - \lambda D_{KL}[Q(z|x)||P(Z)]$$

Il primo termine è il **Reconstruction Term**, il secondo è il **Regularization Term**.

- $Q(z \mid X)$ è l' encoder, che proietta i dati X nello spazio delle variabili latenti.

- z è la variabile latente.

- $P(X \mid z)$ è il decoder, che genera i dati data una variabile latente.

- $D_{\mathrm{KL}}[Q(z \mid X) \| P(z)]$ è una misura di quanto $Q(z \mid X)$ diverge da $P(z)$, una distribuzione di probabilità attesa per la variabile latente (tipicamente scelta come una distribuzione Gaussiana).

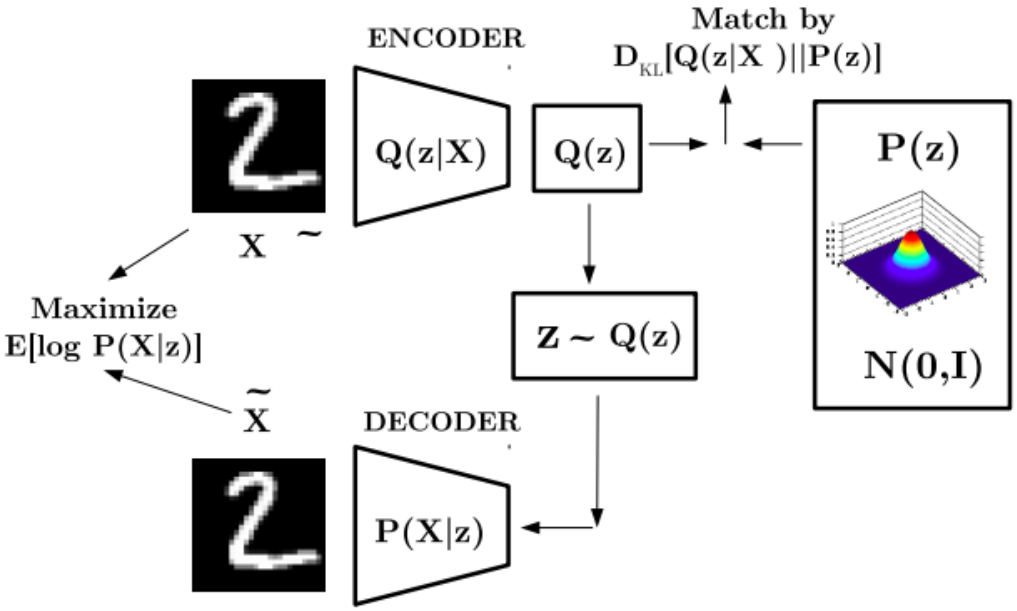


Figura 6.3: Schema di un variational autoencoder

Reparametrization trick

Il campionamento non è derivabile quindi devo fare un trucco: chiedo all'encoder di darmi il codice e σ e μ; per campionare prendo un numero a caso da una gaussiana N(0,1) e lo moltiplico per σ. Il nuovo z, cioè quello campionato, è il risultato di questa operazione.

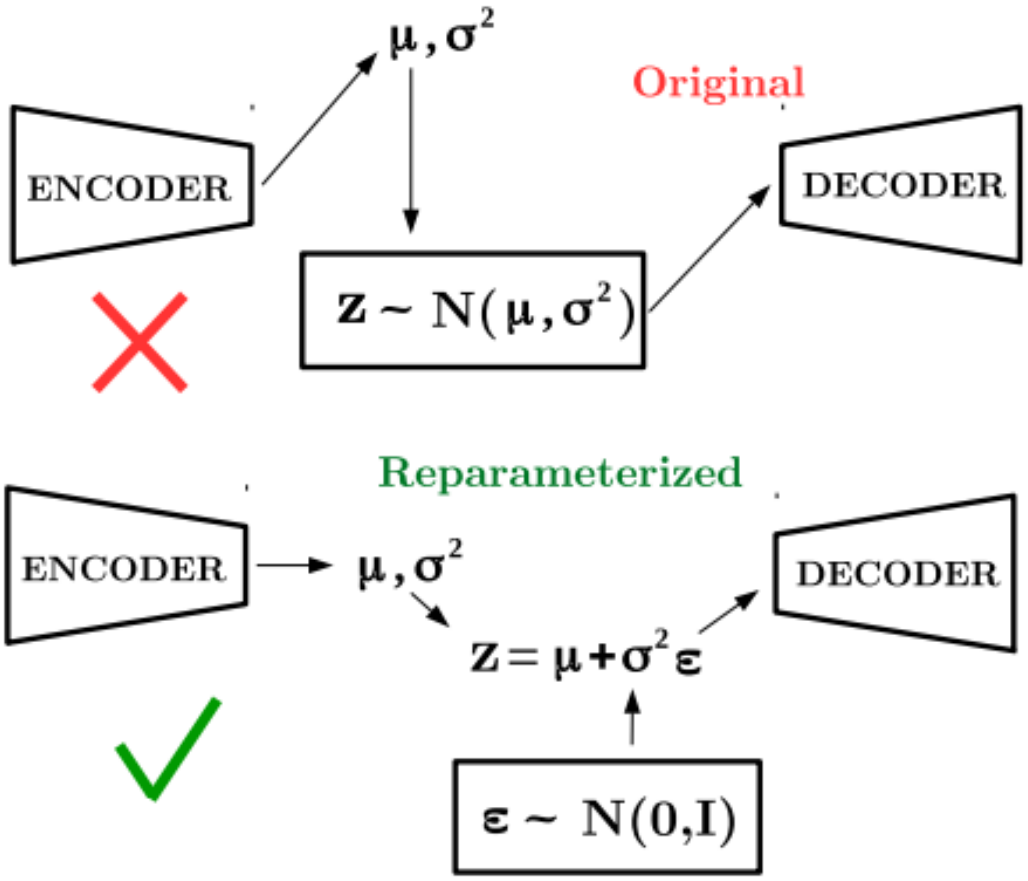


Figura 6.4: Reparametrization Trick

6.2.5 Generative Adversarial Network (GAN)

L'approccio in questo caso è puramente generativo. Non si modella in esplicito la distribuzione di probabilità $p(x)$ da cui provengono i dati. Si vuole solo campionare da $p(x)$.

Questa cosa viene realizzata tramite due reti che competono.

La prima rete è il **Generatore**: ha per ingresso un vettore di numeri, campionato da una distribuzione di probabilità nota $Z \sim p_z(z)$. Il generatore prende z e prova a generare un oggetto che ha la stessa dimensione del dato che vogliamo generare.

La seconda rete è il **Discriminatore**: gli ingressi hanno la stessa dimensione del dato generato. L'uscita è diversa dall'ingresso. Gli ingressi possono essere **real** o **fake**.

L'addestramento di una GAN avviene attraverso un gioco di competizione: si vuole che il generatore diventi bravo a generare immagini; che il discriminatore sia bravo a distinguere immagini real e fake; che ad un certo punto il generatore diventi così bravo da poter ingannare il discriminatore.

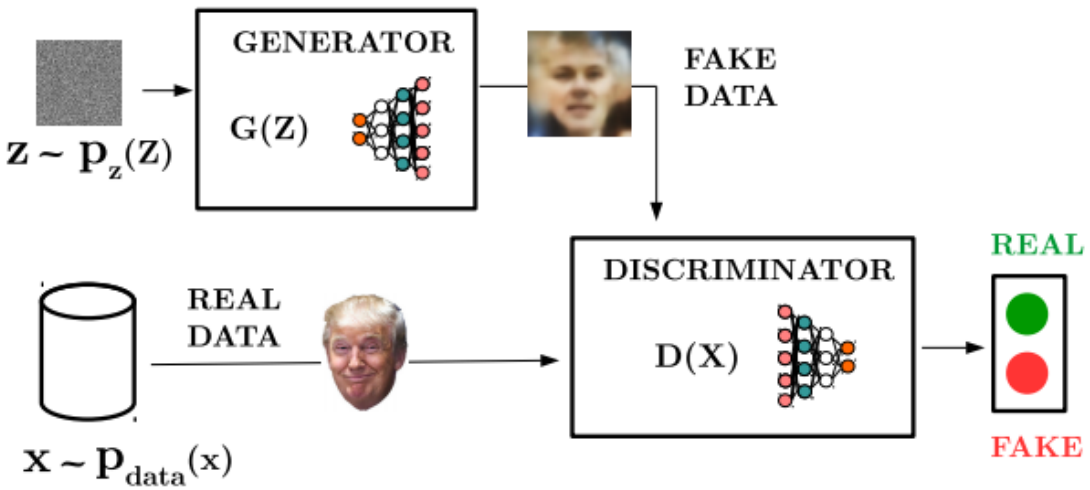


Figura 6.5: Architettura di una GAN

La Loss è:

$$\min_G \max_D V(G, D) = \mathbb{E}_{x \in p_{data}} \log D(x) + \mathbb{E}_{x \in p_{model}} \log(1 - D(x))$$

Il primo termine indica la probabilità di dire real quando il dato proviene dal dataset; il secondo termine indica la probabilità di dire fake quando il dato proviene dal generatore.

Provo a riscriverlo formalmente come da paper:

La struttura delle generative adversarial network è più semplice da applicare quando le due reti sono entrambe multilayer perceptron. Per imparare la distribuzione p_g del generatore sul dato x, si definisce una prior $p_z(z)$ sugli input noise variables, poi si rappresenta un mapping sullo spazio dati come $G(z; \theta_g)$ dove G è una funzione differenziabile rappresentante il multilayer perceptron di parametri θ_g. Si definisce, inoltre, un secondo multilayer perceptron $D(x, \theta_d)$ il cui output è un singolo scalare che indica la probabilità che l'esempio appartenga alla distribuzione dei dati (quindi 0= sicuramente non appartiene alla distribuzione dei dati; 1=appartiene sicuramente alla distribuzione dei dati). $D(x)$ rappresenta la probabilità che x provenga dalla distribuzione dei dati piuttosto che da quella del modello. Noi alleniamo D per massimizzare la probabilità di assegnare la label corretta sia agli esempi provenienti dal training che da quelli provenienti da G. Simultaneamente alleniamo G per minimizzare $\log(D(G(z)))$.

In altre parole G e D giocano ad un 2-player minmax game dove G tenta di ingannare D e D cerca di non farsi ingannare. La loss function dunque si scrive in questo modo:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log [1 - D(G(z))]]$$

Dato che sia G che D sono due multilayer perceptron, il training si può fare con la backpropagation. Questa cosa è molto conveniente perchè più veloce rispetto alle catene di Markov.

Il training si fa in questo modo:

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
  • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  • Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Figura 6.6: Training GAN

Il problema è che, ad esempio, nelle fasi iniziali dell'algoritmo possiamo trovarci con il discriminatore molto più bravo del generatore e quindi con $D(G(z)) \approx 0$, e quindi $\log[1 - D(G(z))]$ satura. Quindi l'equazione di loss potrebbe non garantire gradiente sufficiente a G per imparare bene. Quindi invece che allenare G a minimizzare $\log[1 - D(G(z))]$, la si può allenare a massimizzare $\log D(G(z))$. In questo modo la dinamica è la stessa ma il gradiente risulta più forte nelle fasi iniziali dell'apprendimento.

Per G fissato, il discriminatore ottimale è:

$$D^*_G(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)}$$

Si può dimostrare in questo modo: il criterio di training per il discriminatore D, dato un qualsiasi G, è quello di massimizzare la quantità V(G,D)

$$V(G, D) = \int_x p_{data}(x) \log D(x) dx + \int_z p_z(z) \log[1 - D(G(z))]\, dz =$$

$$= \int_x p_{data}(x) \log D(x) + p_g(x) \log \left[1 - D(x)\right] dx$$

Per ogni $(a, b) \in \mathbb{R}^2 - \{0, 0\}$, la funzione $y \to a \log(y) + b \log(1 - y)$ raggiunge il suo massimo in [0,1] in $\frac{a}{a+b}$. Il discriminatore non ha bisogno di essere definito fuori da $Supp(p_{data}) \cup Supp(p_g)$, quindi la dimostrazione è conclusa.

Si può notare che'obiettivo di training per D può essere interpretato come massimizzare la log-likelihood per stimare la probabilità condizionale $P(Y = y|x)$ dove Y indica se x proviene da p_data (y=1) o da p_g (y=0). Il minmax game può essere quindi riscritto in questo modo:

$$C(G) = max_D V(G, D)) = \mathbb{E}_{x \sim p_{data}}[\log D^*_G(x)] + \mathbb{E}_{z \sim p_z}\left[\log\left[1 - D^*_G(G(z))\right]\right] =$$

$$= \mathbb{E}_{x \sim p_{data}}[\log D^*_G(x)] + \mathbb{E}_{z \sim p_g}\left[\log\left[1 - D^*_G(x)\right]\right] =$$

$$= \mathbb{E}_{x \sim p_{data}}\left[\log \frac{p_{data}(x)}{p_{data}(x) + p_g(x)}\right] + \mathbb{E}_{x \sim p_g}\left[\log \frac{p_g(x)}{p_{data}(x) + p_g(x)}\right]$$

Il minimo globale del criterio di training virtuale C(G) è raggiunto se e solo se $p_g = p_{data}$. A quel punto $C(G) = -\log 4$.

Pro e Contro delle GAN

- **Pro**:

    1. Costruibili in termini di reti neurali.

    2. Loss Function è imparata.

    3. Non ha bisogno di fare in esplicito il campionamento.

- **Contro**:

    1. Generatore e Discriminatore devono progredire insieme.

    2. Durante il training bisogna supervisionare manualmente.

    3. Le metriche di performance sono qualitative quindi non si hanno garanzie.

Capitolo 7

Reinforcement Learning

- Non c'è supervisione, solo reward signal.
- Il feedback è ritardato, non istantaneo.
- L'agente è attivo, le sue azioni inflenzano l'ambiente in cui vive.

7.1 Reward

Una reward R_t è un numero associato ad un time step t. Indica quanto un agente sta facendo beno la task allo step t.

Una reward può essere:

1. **Alta**: l'agente sta facendo bene.
2. **Bassa**: l'agente sta facendo male,

7.2 All'interno di un RL Agent

7.2.1 Sequential Decision Making

L'obiettivo dell'agente è quello di massimizzare la reward cumulativa su un episodio.

Le azioni svolte dall'agente possono avere conseguenze a lungo termine, inoltre le reward possono essere ritardate. Può essere quindi conveniente sacrificare reward immediate per ottenere maggiori reward a lungo termine.

7.2.2 Agent and Environment

Ad ogni time step:

- L'agente riceve l'osservazione O_t dall'ambiente, la reward R_t ed esegue l'azione A_t.
- L'ambiente riceve l'azione A_t, emette l'osservazione O_{t+1} ed emette una reward R_{t+1}.

7.2.3 History & State

La History è la sequenza delle osservazioni, azioni, rewards.

$$H_t = O_1, A_1, R_1, \ldots, O_t, A_t, R_t$$

Lo State è l'informazione usata per determinare cosa accadrà dopo. È una funzione della storia.

$$S_t = f(H_t)$$

7.2.4 Agent and Environments states

Lo stato dell'agente è qualsiasi informazione S_t^a esso usi per scegliere la prossima azione da compiere. È l'informazione utilizzata dagli algoritmi di RL.

Lo stato dell'ambiene è qualsiasi dato che l'ambiente utilizza per scegliere la prossima osservazione/reward. Generalmente non è visibile all'agente.

Se l'agente osserva direttamente lo stato dell'ambiente si parla di **full observability**. Se, al contrario, l'agente osserva indirettamente lo stato dell'ambiente si parla di **partial observability**.

7.2.5 Componenty di un agent

L'agent può includere uno o più di questi componenti:

- **Policy**: è la funzione di comportamento dell'agente.
- **Value Function**: quanto buono/a è ogni stato/azione.
- **Model**: rappresentazione dell'ambiente.

Policy

La policy è il comportamento dell'agente. È una mappa dallo stato all'azione.

Si possono avere due tipi di policy:

- **Deterministic Policy**:
$$a = \pi(s)$$
- **Stochastic policy**:
$$\pi(a|s) = P(A_t = a|S_t = s)$$

Return

Il ritorno è la reward complessiva al time step t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

- Lo sconto $\gamma \in [0, 1]$ rappresenta il valore attuale dei reward futuri.

- Il valore di ricevere un reward R dopo $k+1$ time-step è $\gamma^k R$.
- Questo dà priorità ai reward immediati rispetto a quelli ritardati.
- Un valore di γ vicino a 0 porta a una valutazione miope.
- Un valore di γ vicino a 1 porta a una valutazione lungimirante.

Value Function

È una predizione delle future reward. È utilizzata per valutare la "bontà" di uno stato e per scegliere le azioni possibili.

Definizione

La funzione di valore dello stato $v_\pi(s)$ è il ritorno atteso a partire dallo stato s, seguendo poi la policy π.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$

Definizione

La funzione di valore d'azione $q_\pi(s, a)$ è il ritorno atteso a partire dallo stato s, eseguendo l'azione a, e seguendo poi la politica π.

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$$

La value function può essere scomposta in due parti:

- reward immediato R_{t+1}
- discounted value dello stato successivo $\gamma v(S_{t+1})$

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] =$$
$$= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma R_{t+3} + \ldots |S_t = s] =$$
$$= \mathbb{E}_\pi[R_{t+1} + \gamma(R_{t+2} + R_{t+3})|S_t = s] =$$
$$= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] =$$
$$= \mathbb{E}_\pi[R_{t+1} + \gamma v(S_{t+1})|S_t = s]$$

Anche l'action-value function può essere scomposta in modo simile:

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma q(S_{t+1}, A_{t+1})|S_t = s, A_t = a]$$

Model

Predice cosa farà l'ambiente. P predice il prossimo stato. R predice il prossimo reward immediato.

$$P^a ss' = P[S_{t+1} = s' \mid S_t = s, A_t = a]$$

$$R_s^a = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$$

7.3 Model free Prediction

Non viene costruito esplicitamente un modello dell'ambiente. Stima la value function data una policy in un ambiente non osservabile.

Si hanno due possibilità:

- **Monte-Carlo Learning**
- **Temporal-Difference Learning**

7.3.1 Monte-Carlo reinforcement learning

I metodi Monte Carlo (MC) apprendono direttamente dagli episodi di esperienza. Non richiedono una conoscenza esplicita dei meccanismi dell'ambiente, rendendoli quindi model-free. L'apprendimento avviene utilizzando informazioni tratte dagli episodi completi, ma con un limite: possono essere applicati solo ad ambienti episodici, dove tutti gli episodi terminano. Il principio alla base di MC è semplice: il valore viene calcolato come media del ritorno ottenuto.

MC Policy evaluation

L'obiettivo è quello di imparare v_π da episodi di esperienza sotto la policy π. Durante l'apprendimento si osserva una sequenza di stati, azioni, ricomense:

$$S_1, A_1, R_2, \ldots, S_k \sim \pi$$

Il ritorno totale è la discounted total reward:

$$G_t = R_{t+1} + \gamma + R_{t+2} + \cdots + \gamma^{T-1} + R_T$$

La value function è il valore atteso del ritorno. In altre parole, è la media delle ricompense future che ci si aspetta di ottenere, considerando tutte le possibili traiettorie a partire da s secondo π.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$

La valutazione della policy Monte Carlo utilizza la media empirica dei ritorni osservati al posto del ritorno atteso. Questo significa che, invece di calcolare il valore teorico basato su una distribuzione probabilistica, si stima $v_\pi(s)$ osservando direttamente gli episodi e calcolando la media delle ricompense totali per ciascuno stato.

In sintesi: La valutazione Monte Carlo consente di stimare la funzione di valore $v_\pi(s)$ osservando le esperienze reali senza bisogno di conoscere il modello dell'ambiente. La chiave è sostituire la stima teorica con una media basata sui dati raccolti.

Every-Visit MC Policy evaluation

Questa procedura utilizza un approccio empirico per stimare il valore di uno stato. Ogni volta che uno stato è visitato in un episodio, si accumula il ritorno totale osservato e si aggiorna una media basata sul numero di visite. Nel lungo periodo, con un numero sufficiente di visite, la stima della media empirica tende al valore atteso teorico.

Questo metodo è particolarmente utile in contesti in cui non si conosce il modello dell'ambiente, ma si ha accesso a dati episodici da cui trarre informazioni.

Per valutare uno stato s: Ogni volta che lo stato s viene visitato durante un episodio, si compiono i seguenti passi:

1. Si incrementa un contatore $N(s)$ che tiene conto di quante volte lo stato s viene visitato:

$$N(s) \leftarrow N(s) + 1$$

2. Si incrementa il totale dei ritorni aggiungendo il ritorno osservato:

$$S(s) \leftarrow S(s) + G_t$$

3. La stima del valore dello stato viene calcolata come media dei ritorni osservati:

$$V(s) = \frac{S(s)}{N(s)}$$

4. Per la legge dei grandi numeri:

$$V(s) \to v_\pi(s) \quad \text{quando } N(s) \to \infty$$

Incremental MC updates

In questo approccio, la stima del valore $V(s)$ viene aggiornata in modo incrementale dopo ogni episodio, evitando di conservare l'intera cronologia delle esperienze.

Gli aggiornamenti incrementali Monte Carlo evitano di memorizzare tutti i ritorni e semplificano il calcolo del valore stimato. Questo approccio è utile in ambienti dinamici o con grandi spazi di stato, dove un approccio basato su una memoria completa sarebbe inefficiente. L'uso di una media mobile permette di "dimenticare" gradualmente gli episodi più vecchi, adattandosi meglio a cambiamenti nell'ambiente o nella policy.

La procedura è la seguente:

1. Dopo aver completato l'episodio $S_1, A_1, R_2, \ldots, S_T$, si calcola G_t.

2. Per ciascuno stato S_t visitato nell'episodio:

   - Si incrementa il contatore delle visite:

$$N(S_t) \leftarrow N(S_t) + 1$$

   - Si aggiorna il valore $V(S_t)$ usando la media incrementale:

$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t))$$

Questo approccio tiene conto di tutti i ritorni osservati per uno stato, ma assegna un peso progressivamente minore alle osservazioni più vecchie.

Spesso si utilizza una versione semplificata che applica una media mobile con un fattore di apprendimento α ($0 < \alpha < 1$):

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$$

Qui α controlla quanto velocemente i nuovi dati sovrascrivono i valori precedenti. Un valore alto di α dà più peso alle osservazioni recenti, mentre un valore basso conserva maggiormente le stime passate. In questo modo non è necessario mantenere un contatore delle visite riducendo il calcolo ad un singolo aggiornamento per ogni stato visitato.

Blackjack example

Nel gioco del Blackjack, l'obiettivo è avere un punteggio il più vicino possibile a 21 senza superarlo. Il giocatore può scegliere tra due azioni:

- Stick: il giocatore si ferma (smette di ricevere carte) e termina il turno.
- Twist: il giocatore chiede un'altra carta.

Gli stati sono definiti da tre variabili:

1. **Somma attuale** delle carte carte del giocatore (valori da 12 a 21, poiché al di sotto di 12 il giocatore pesca automaticamente).
2. **Carta visibile del dealer** (valori da Asso a 10)
3. **Possesso di un asso utilizzabile** (Sì/No): un asso utilizzabile è un asso che può valere 11 senza superare 21.

In totale ci possono essere 200 possibili stati.

Le reward sono diverse a seconda che si scelga Stick o Twist:

- Stick:
  - +1: il punteggio del giocatore è maggiore di quello del dealer.

  - 0: il punteggio del giocatore è uguale a quello del dealer.
  - -1: il punteggio del giocatore è minore di quello del dealer.

- Twist:
  - -1: se il totale supera 21.
  - 0: altrimenti.

Transizioni: se la somma delle carte del giocatore è minore a 12 allora il giocatore è obbligato a prendere.

Una policy iniziale può essere:

- Il giocatore sceglie stick se la somma delle carte è maggiore o uguale a 20.
- Altrimenti sceglie twist.

Monte Carlo viene utilizzato per stimare la funzione di valore $V(s)$ per ciascuno stato s. Questo avviene osservando gli episodi giocati, accumulando i ritorni associati agli stati visitati, e calcolando la media empirica dei ritorni.

Dopo l'apprendimento, la funzione di valore $V(s)$ rappresenta una mappa dei ritorni attesi per ciascuno stato. L'obiettivo è migliorare la politcy utilizzando queste stime, adottando strategie che massimizzano il ritorno atteso per ogni stato.

7.3.2 Temporal Difference

TD è un metodo di apprendimento model-free che combina idee di Monte Carlo e programmazione dinamica. Aggiorna le stime di valore dopo ogni passo dell'episodio, utilizzando stime intermedie.

Il più semplice algoritmo di TD è il TD(0).

L'obiettivo è Aggiornare $V(S_t)$ in base ad una stima del ritorno futuro $R_{t+1} + \gamma V(S_{t+1})$ anziché attendere l'intero episodio. La formula di aggiornamento è:

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$

- $R_{t+1} + \gamma V(S_{t+1})$ è detto TD target. Combina la ricompensa immediata e il valore scontato del successivo stato.

- $\delta = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ è il TD error. Questo errore misura quanto la stima attuale si discosta dall'osservazione e guida l'aggiornamento.

La procedure è dunque questa: dopo ogni passo dell'episodio si calcola il TD target e il TD error. Si usano queste informazioni per aggiornare $V(S_t)$, rendendo così l'apprendimento incrementale.

7.3.3 MC e TD

In sintesi:

- **MC**: è basato su ritorni osservati e richiede episodi completi, risultando preciso ma meno flessibile.

- **TD(0)**: aggiorna le stime in modo incrementale, permettendo di apprendere senza attendere l'intero episodio e lavorare in ambienti non episodici.

7.4 Model free Control

L'obiettivo è quello di cercare una buona policy in un ambiente non osservabile.

Ci sono varie possibilità:

- **On-Policy Monte-Carlo Control**

- **On-Policy Temporal-Difference Learning**

- **Off-Policy Learning**

7.4.1 Come migliorare la policy

Data una policy π la si deve valutare:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \ldots | S_t = s]$$

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \ldots | S_t = s, A_t = a]$$

Queste funzioni forniscono una misura della bontà degli stati o delle combinazioni stato-azione sotto una data policy. Una volta valutata la policy, è possibile migliorarla agendo in modo greedy rispetto alla funzione di valore:

$$\pi' = greedy(v_\pi)$$

$$\pi' = greedy(q_\pi)$$

Nei contesti model-free, l'apprendimento è basato solo sui dati esperienziali, utilizzando la funzione $Q(s, a)$. Questo approccio iterativo è alla base del reinforcement learning per migliorare le policy e massimizzare le ricompense cumulative.

7.4.2 ϵ-greedy exploration

L'ϵ-greedy exploration è una tecnica semplice ed efficace per garantire che l'agente esplori continuamente l'ambiente, evitando di rimanere bloccato su azioni subottimali.

L'obiettivo è quello di Bilanciare l'esplorazione (provare azioni nuove) e lo sfruttamento (scegliere l'azione migliore secondo le conoscenze attuali dell'agente). L'esplorazione è necessaria per scoprire nuove azioni che potrebbero portare a ricompense migliori. Lo sfruttamento serve per massimizzare il ritorno atteso utilizzando ciò che l'agente ha già appreso.

Come funziona:

1. L'agente sceglie l'azione:

   - Con probabilità $1 - \epsilon$ esegue l'azione greedy cioè quella che massimizza il valore stimato $Q(s, a)$. Cioè:

$$a^* = \arg\max_{a \in A} Q(s, a)$$

- Con probabilità ϵ sceglie un'altra azione random.

2. La policy $\pi(a|s)$, ossia la probabilità di scegliere l'azione a quando si è nello stato s è:

$$\pi(a|s) = \begin{cases} \frac{\epsilon}{m} + (1-\epsilon) & \text{se } a^* = \arg\max_{a \in A} Q(s,a) \\ \frac{\epsilon}{m} & \text{altrimenti} \end{cases}$$

7.4.3 On and Off Policy

Il confronto tra on-policy learning e off-policy learning riguarda il modo in cui un agente apprende informazioni per migliorare la propria policy in base alle esperienze raccolte.

Nell'On-Policy, l'agente impara una policy π (quella su cui intende basarsi) utilizzando esperienze campionate seguendo proprio quella stessa policy. Si dice che l'apprendimento è sul campo. L'agente segue la policy corrente per esplorare l'ambiente; durante questa esplorazione, raccoglie dati (stati, azioni, ricompense); utilizza questi dati per migliorare la stessa policy. Riprendendo l'esempio del blackjack, l'agente si fermava quando raggiungeva 20 punti (questa era la policy), questa regola viene poi aggiornata guardando i risultati ottenuti. Richiede un'esplorazione esplicita per evitare che l'agente rimanga bloccato in comportamenti subottimali.

Nell'Off-Policy, l'agente impara una policy π (quella che intende ottimizzare) utilizzando esperienze raccolte seguendo una policy diversa μ. Si dice che si impara sulle spalle di qualcun altro. La policy di comportamento μ viene usata per esplorare l'ambiente (potrebbe essere casuale o basata su un'altra regola). I dati raccolti da μ vengono usati per ottimizzare una policy target π, che potrebbe differire da μ. Ad esempio, la policy target π può essere: "Scopri la strategia ottimale per vincere al Blackjack". La policy di comportamento μ è: "Gioca a caso per esplorare ogni possibile scenario". Gli esiti del comportamento casuale (μ) sono usati per migliorare π. È più flessibile, poiché permette di apprendere più policy target usando esperienze raccolte con μ. Può riutilizzare esperienze passate o osservate da altri agenti. Può essere applicato in scenari dove esplorazione e sfruttamento sono separati dato che **non** richiede che $\pi = \mu$.

Quindi: On-policy è diretto e intuitivo, ideale per situazioni in cui esplorazione e sfruttamento sono strettamente legati. Off-policy è più flessibile e potente, poiché consente di apprendere da esperienze diverse o di riutilizzare esperienze passate, ma è più complesso da implementare.

MC vs TD control

Il TD presenta diversi vantaggi rispetto al MC:

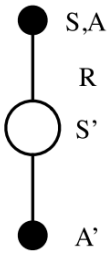
1. ha una varianza minore, quindi le stime risultano più precise.

2. è online.

3. può utilizzare sequenze incomplete.

Un'idea naturale dunque può essere quella di usare il TD al posto dell'MC nel ciclo di controllo. Quindi si applica TD a $Q(S, A)$, si usa una greedy-policy improvement e si fa l'update ad ogni time step.

7.4.4 Updating action-value with SARSA



$$Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + Q(S', A') - Q(S, A)\right]$$

Questa equazione aggiorna la stima del valore di una coppia stato-azione (S,A) basandosi sulle informazioni osservate da una singola transizione nell'ambiente.

Algorithm 3: SARSA Algorithm for On-Policy Control

Initialize $Q(s, a)$ arbitrarily, $\forall s \in S, a \in A(s)$
Set $Q(\text{terminal-state}, \cdot) = 0$ **for** *each episode* **do**
    Initialize S
    Choose A from S using policy derived from Q (e.g., ε-greedy) **while** *episode is not terminated* **do**
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g., ε-greedy)
        Update $Q(S, A)$:

$$Q(S, A) \leftarrow Q(S, A) + \alpha\big(R + \gamma Q(S', A') - Q(S, A)\big)$$

        Update $S \leftarrow S'$, $A \leftarrow A'$

nel caso dell'off policy, per valutare la politica target $\pi(a, s)$, l'obiettivo è calcolare la funzione di valore di stato $v_\pi(s)$ o la funzione di valore azione-stato $q_\pi(s, a)$:

- L'agente segue una *policy di comportamento* $\mu(a|s)$ per campionare esperienze.
- La sequenza di esperienze raccolte è rappresentata come:

$$\{S_1, A_1, R_2, \ldots, S_T\} \sim \mu$$

  dove l'agente interagisce con l'ambiente seguendo μ per raccogliere dati.

L'uso di una politica di comportamento μ per valutare una policy target π è rilevante per diversi motivi:

1. **Imparare osservando esseri umani o altri agenti:** Seguendo μ, l'agente può raccogliere dati osservando le azioni eseguite da esseri umani o altri agenti intelligenti nell'ambiente.
2. **Riutilizzare esperienze generate da vecchie politiche:** Le esperienze passate raccolte utilizzando politiche precedenti $\pi_1, \pi_2, \ldots, \pi_{t-1}$ possono essere

sfruttate per valutare o migliorare la politica target corrente π.

3. **Imparare la politica ottimale seguendo una politica esplorativa:** La politica di comportamento μ può essere progettata per incoraggiare l'esplorazione (ad esempio, usando un approccio ε-greedy), mentre la politica target π si concentra sull'ottenere prestazioni ottimali.

4. **Imparare più politiche seguendone una sola:** Un'unica politica di comportamento μ può essere utilizzata per raccogliere esperienze utili per valutare e migliorare più politiche target contemporaneamente.

7.4.5 Q-Learning

Ora consideriamo l'off-policy learning di azione-valori $Q(s, a)$. L'azione successiva è scelta utilizzando la policy di comportamento: $A_{t+1} \sim \mu(\cdot, S_t)$, l'azione alternativa è $A' \sim \pi(\cdot, S_t)$.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A') - Q(S_t, A_t))$$

- Ora consentiamo di migliorare sia la policy di comportamento (μ) che la policy obiettivo (π).

- La policy obiettivo π è greedy rispetto a $Q(s, a)$:

$$\pi(S_{t+1}) = \arg\max_{a'} Q(S_{t+1}, a')$$

- La policy di comportamento μ è ϵ-greedy rispetto a $Q(s, a)$.

- Il target del Q-learning si semplifica come segue:

$$R_{t+1} + \gamma Q(S_{t+1}, A')$$

$$= R_{t+1} + \gamma Q(S_{t+1}, \arg\max_{a'} Q(S_{t+1}, a'))$$

$$= R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a')$$

il Q-learning per l'off policy control si fa in questo modo:

- Inizializza $Q(s, a)$ per ogni stato $s \in S$ e azione $a \in A(s)$ arbitrariamente, e $Q(\text{terminal-state}, \cdot) = 0$

- Per ogni episodio:

  - Inizializza lo stato S

  - Per ogni passo dell'episodio:

    * Scegli A dallo stato S usando una politica derivata da Q (ad esempio, ϵ-greedy)

    * Esegui l'azione A, osserva la ricompensa R e il nuovo stato S'

    * Aggiorna $Q(S, A)$:

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left(R + \gamma \max_{a'} Q(S', a') - Q(S, A) \right)$$

    * Aggiorna lo stato: $S \leftarrow S'$

7.5 Function Approssimation

Il reinforcement learning può essere usato per risolvere problemi di grandi dimensioni. Come scalare i metodi model free visti in precedenza? La soluzione è quella di stimare la funzione valore con la function approssimation.

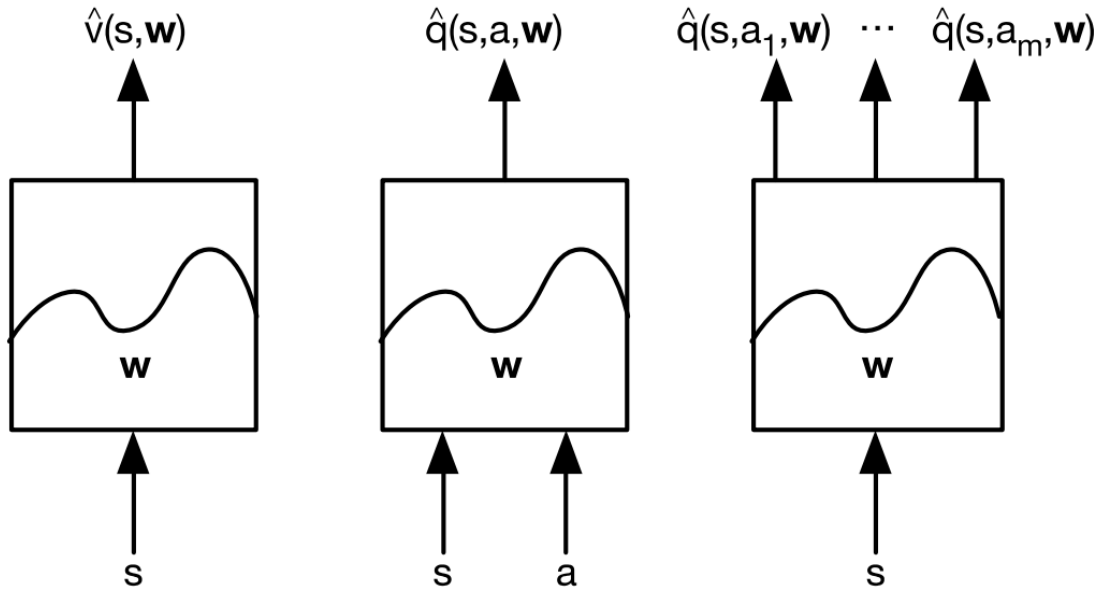
$$\hat{v}(s, w) \approx v_\pi(s)$$

$$\hat{q}(s, a, w) \approx q_\pi(s, a)$$

Si generalizza da stati visti a non visti e si aggiorna il parametro w utilizzando l'MC learning o il TD learning.

7.5.1 Tipi di value-function approximation

Ci possono essere diversi tipi di funzione di approssimazione.



7.5.2 Tipi di function approximator

- Linear combinations of features
- Neural network
- Decision tree
- Nearest neighbour
- Fourier / wavelet bases
- ...

Noi considereremo le prime due, ossia i differentiable function approximators. Inoltre, avremo bisogno di un metodo di apprendimento per dati non stazionari e non i.i.d.

7.5.3 Metodi incrementali

Capitolo 8

Domade Frequenti

Si riportano alcune domande fatte durante l'esame orale:

1. Cosa vuol dire fare classificazione lineare e che differenza c'è tra classificatori lineari e non lineari? Qual è l'equazione di classificazione?

2. Spiega il K-Means. Quando lo utilizziamo?

3. Divergenza KL e la sua relazione con la Cross Entropy.

4. Spiega il K-Means. È un algoritmo ottimo?

5. Scrivi la Loss dei Contractive Autoencoder.

6. Cos'è la likelihood e come si approssima in un classificatore a maximum likelihood? Spiegare la stima parametrica.

7. Spiegare il Denoising Autoencoder.

8. Regola di Bellman, dove viene usata maggiormente?

9. Equazione di un classificatore LDA e mostrare che è lineare. Equazione gaus-

siana multivariata, come si trova il boundary?

10. Strategie per prevenire l'overfitting nel Deep Learning. Differenza tra norma L1 ed L2. Perché in genere una rete neurale overfitta di più di un classificatore?

11. Spiegare il Q-Learning. Cosa vuol dire che il Q-Learning è un metodo off-policy? Nella formula dove si vede che è off-policy?

12. Spiegare la BackPropagation (con disegno).

13. Fare un esempio di metodo di riduzione lineare.

14. Relazione tra w e α nell'SVM.

15. Possibili configurazioni che si possono usare in una rete ricorrente. Spiegare per cosa possono essere usate le varie tipologie.

16. Loss di un Autoencoder Sparso.

17. In inferenza come si può sfruttare il primale-duale dell'SVM per ottenere il Kernel Trick? In termini di occupazione di spazio ed efficienza è meglio il primale o il duale?

18. Spiegare il clustering gerarchico.

19. Algoritmo di k-face (Lab).

20. Classificatore bayesiano di tipo Naive. Perché si dice che questa procedura previene l'overfitting?

21. Spiegare la Logistic Regression.

22. Spiegare la Binary Cross Entropy.

23. Spiegare il K-means.

24. Relazione tra perceptron e LR, in quale caso sono la stessa cosa?

25. Logistic Regression, mostrare che è lineare e relazione con la Binary Cross Entropy.

26. Spiegare le GAN, scrivere la funzione di Loss.

27. Algoritmo di Temporal Difference, in che punto abbiamo applicato Bellman?

28. Spiegare l'Adaboost.

29. Precision e Recall, cosa sono e come si usano.

30. Come si usa un Validation Set per trovare gli iperparametri.

31. Loss del Contractive Autoencoder.

32. Spiegare i Variational Autoencoder ma solo la struttura che si implementa.

33. Adaboost.

34. In un classificatore a boosting qual è la principale causa di overfitting?

35. Spiegare l'algoritmo SARSA.

36. Spiegare il Bagging. Perché un classificatore a bag ha un errore minore o uguale a quello del classificatore singolo? Quali sono le condizioni perché questo avvenga?

37. Assunzione i.i.d. Cosa succede al classificatore sul test se violiamo la condizione i.i.d?

38. BackPropagation through time, disegno e far vedere $\frac{dL}{dW}$. Qual è il termine che porta il problema dell'exploding gradient e per quale motivo.