Glossario Machine Learning

circled-square

June 11, 2024

Definizioni base

- spazio del problema: $\mathcal{F}_{\text{task}} = \{f : X \to Y\} = Y^X$
- spazio di ipotesi $\mathcal{H} \subset \mathcal{F}_{\text{task}}$ contiene le funzioni computabili con il modello scelto
- data generating distribution: p_{data}
- pointwise error: $E(f; P)$ dove $f : X \to Y$ e P è una distribuzione (o insieme)
- target ideale:
$$f^\star \in \arg\min_{f \in \mathcal{F}_{\text{task}}} E(f; p_{\text{data}})$$
- feasible target:
$$f^\star_{\mathcal{H}} \in \arg\min_{f \in \mathcal{H}} E(f; p_{\text{data}})$$
- training set:
$$\mathcal{D}_n = \{z_1, ..., z_n\} \subset \mathcal{X} \times \mathcal{Y} \text{ where } z_i \sim p_{\text{data}}$$
- target ideale sul training set:
$$f^\star(\mathcal{D}_n) \in \arg\min_{f \in \mathcal{H}} E(f; \mathcal{D}_n)$$
- actual target:
$$f^\star_{\mathcal{H}}(\mathcal{D}_n) \in \arg\min_{f \in \mathcal{H}} E(f; \mathcal{D}_n)$$
- underfitting: il modello non approssima bene i dati di training, probabilmente a causa di mancanza di training;

- overfitting: il modello approssima bene i dati di training, ma non altrettanto su p_{data}
- regularizzazione: cambia l'errore di training perché penalizzi funzioni complesse per evitare overfitting. È sempre una funzione complessa (x gradient descent). alcuni esempi possono essere lunghezza o p-norm del vettore dei pesi w. L'iperparametro λ ne controlla l'impatto.

$$E_{reg}(f; \mathcal{D}_n) = E(f; \mathcal{D}_n) + \lambda\Omega(f)$$

Errori

- errore di stima: perché $\mathcal{D}_n \neq p_{\text{data}}$
- error di approssimazione: perché $\mathcal{H} \neq \mathcal{F}_{\text{task}}$
- error irriducibile: a causa di varianza intrinseca, non può essere ridotto cambiando modello ma è possibile che una riselezione delle feature aiuti.

Set vari (proxy di p_{data})

- training set per l'addestramento;
- validation set per aggiustare gli iperparametri;
- test set usato per scegliere il modello migliore;

Tipi di apprendimento:

- supervised learning: dato un insieme di esempi etichettati il modello impara a predire l'etichetta di nuovi esempi.
  - classificazione: supervised learning con un insieme finito e discreto di etichette
    - k-nearest neighbor: l'addestramento memorizza soltanto i dati (lazy learning). Per trovare la classe di un esempio si considera la classe dei k (che è un iperparametro) punti più vicini nel training set, e la classe è sarà quella che compare più volte. (problema della curse of dimensionality TODO)
    - KNN pesato fa una media pesata delle label dei k punti più vicini, dando più peso ai più vicini.

    * percettrone (online), impara un iperpiano che separa due classi; l'iperpiano è progressivamente mosso mano a mano che vengono forniti dati. Il processo si ferma quando converge (che è garantito per dati linearmente separabili). Inference phase: $y = sign(b + w \cdot x)$. Una versione estesa del percettrone è usata nelle reti neurali an extended version of this is used in neural nets
    * Decision Trees: vedi sotto
    * Support Vector Machines: vedi sotto
    * un modello di classificazione binaria può essere convertito a multiclasse atraverso l'utilizzo di tecniche "black box" come OVA e AVA
    * One Versus All (OVA): viene addestrato un modello per ogni classe y che calcola la confidence che la classe di un x sia y. inference phase: x viene dato in pasto a tutti i modelli, è scelta la classe corrispondente al modello con la massima confidenza. Se l'algoritmo non ritorna una confidence è scelta una tra le classi in conflitto.
    * All Versus All (AVA): un classificatore F_{ij} è addestrato per ogni paio di classi $i, j \mid 1 \leq i < j \leq K$
      Inference phase: si può scegliere la maggioranza o fare un voto pesato:
      $\forall i, j \mid 1 \leq i < j \leq K$
      $y = F_{ij}(x)$
      $\text{score}_j += y$
      $\text{score}_i -= y$
    * AVA vs OVA: AVA addestra più modelli, ma con training set più piccoli, quindi è più veloce da addestrare, mentre OVA è più veloce nella inference phase. Tipicamente è preferito OVA.
    * la precisione di un algoritmo multiclasse può essere misurata in due modi:
      · microaveraging: fa la media della precisione del modello su tutto il validation/test set;
      · macroaveraging: fa la media della precisione del modello per ogni classe e poi fa la media del valore di precisione per ogni classe.
  – regressione: supervised learning con uno spazio delle etichette continuo;
  – ranking: classificazione dove l'etichetta è il rank;

• unsupervised learning: i dati di training non hanno etichette
  – density estimation: trova una distribuzione f che approssimi p_{data} (deve fittare $\mathcal{X}$)
  – clustering: trova una funzione $f : \mathcal{X} \to \mathbb{N}$

    * K-means:
      scegli posizioni casuali per i centri dei cluster, assegna ogni punto al cluster il cui centro è più vicino, poi per ogni cluster sposta il centro del cluster alla posizione media dei suoi punti. Inference phase: ogni esempio è assegnato al cluster il cui centro è più vicino
    * Expectation Maximisation (EM):
      come K-means, ma fa "soft" clustering: per ogni punto ritorna la probabilità che sarà contenuto in ognuno dei cluster. ogni cluster avrà invece del centro una media e una matrice di covarianze.
    * Spectral Clustering:
      in grado di gestire dati non-gaussiani, considera valori di somiglianza (basati sulla distanza) tra vetici in un grafo e riunisce quelli simili. Può essere utilizzato per Hierarchical Clustering.
    * Hierarchical Clustering: dato un $\mathcal{X}$ trova un dendrogram, cioè un set di cluster annidati organizzati in un albero. non produce un solo clustering ma un insieme.
  - dimensionality reduction: mappa uno spazio di input con molte dimenioni ad uno che ne abbia meno.
    * TODO: principal component analysis
    * auto encoder: vedi sotto, paragrafo neural networks

- reinforcement learning: vedi sotto
- offline vs online learning: L'apprendimento è online se il processo "consuma" un data point alla volta, aggiornando progressivamente i parametri. Utile per dataset enormi, stream di dati e applicazioni privacy-friendly. (esempio: perceptron; controesempio: KNN)

Tipi di distanza e somiglianza

- p-norm: $||x||_p = \sqrt[p]{(|x_1|^p + |x_2|^p + ... + |x_n|^p)}$
- cosine similarity: $S_c(a, b) = \frac{a \cdot b}{|a||b|}$

Gradient descent:

- Gradient descent è un modo per imparare i parametri che minimizzano la loss function.

- Ad ogni step di addestramento viene calcolato il gradiente della loss function rispetto ai parametri, mano a mano aggiornando i parametri seguendo l'inverso del gradiente fino a quando non si raggiunge un minimo locale.

  L'aggiornamento dei pesi avviene nel modo seguente:

  $$w_j = w_j - \eta \frac{d}{dw_i} \text{loss}(w)$$

  dove η è il learning rate, un iperparametro, tipicamente diminuisce progressivamente con ogni iterazione.

- surrogate loss function: il gradient descent necessita che la loss function sia differenziabile, continua e idealmente convessa; la 0/1 loss non è nessuna di queste cose, quindi per utilizzare gradient descent ci appoggiamo a funzioni surrogate che pongono un limite superiore alla 0/1 loss.

Decision Trees

- modello di classificazione multiclass, con struttura ad albero.
- le foglie si dicono nodi terminali, gli altri nodi sono non-terminali. I nodi non-terminali hanno (generalmente) 2 figli e implementano una funzione di routing, mentre i terminali implementano la funzione di predizione.
- i non terminali sono: Node(ϕ, t_L, t_R) dove ϕ è la funzione di routing e t_X sono i figli. $\phi(x) \in \{L, R\}$
- i terminali sono: Leaf(h) dove $h \in \mathcal{F}_{\text{task}}$ o equivalentemente $h : \mathcal{X} \to \mathcal{Y}$, ma generalmente è una costante.
- inference phase è banale: basta attraversare l'albero, ad ogni passo visitare il figlio indicato da $\phi(x)$ fino a raggiungere un terminale; la $h(x)$ del terminale sarà la classe di x.
- learning phase: vogliamo trovare

  $$t^\star \in \arg\min_{t \in \mathcal{T}} E(f_t; \mathcal{D}_n)$$

  dove $\mathcal{D}_n \subset \mathcal{X} \times \mathcal{Y}$ e

  $$E(f; \mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{z \in \mathcal{D}} l(f; z)$$

Per trovare l'albero che minimizza l'errore possiamo usare una soluzione greedy: cresci un terminale se

$$I(\mathcal{D}) < \epsilon$$

dove ϵ è un iperparametro, $\mathcal{D}$ è il training set che raggiunge un nodo, e $I(\mathcal{D})$ è la sua impurità, cioè il minimo errore rispetto a $\mathcal{D}$ che una funzione h scelta per $\mathcal{D}$ avrebbe.

Altrimenti (a meno che un'altezza massima per l'albero o una minima dimensione per $\mathcal{D}$ non siano stati fissati) verrà fatto crescere un non-terminale. La funzione di routing ottimale sarà scelta:

$$\phi^\star_{\mathcal{D}} \in \arg\min_{\phi \in \Phi} I_\phi(\mathcal{D})$$

dove $I_\phi(\mathcal{D})$ è l'impurità della funzione di routing, che è la media pesata delle impurità dei due insiemi figli, con come pesi le loro cardinalità.

Dopo aver cresciuto un non-terminale l'algoritmo sarà lanciato ricorsivamente sugli insiemi figli.

- questo algoritmo può anche essere applicato a problemi di regressione cambiando la funzione di impurità in modo che ritorni la varianza delle etichette contenute in $\mathcal{D}$.
- soggetto a overfitting.
- random forest: un'estensione di DT che utilizza un array di diversi DT e a inference phase ritorna la media delle inferenze dei vari alberi. Alberi diversi sono addestrati su feature differenti estratte a caso e diversi subset del training set campionati casualmente.

Support Vector Machines

- SVM è un modello di classificazione binaria che trova l'iperpiano ottimale per separare le due classi. L'addestramento è online.
- SVM vs perceptron:
  - perceptron non trova necessariamente l'iperpiano di separazione ottimale (per ottimale si intende con la massima distanza dai punti più vicini).
- hyperplane margin: qualunque punto oltre il margine viene assegnato una classe con confidenza totale, altrimenti c'è incertezza riguardo alla classe del punto.

- support vectors: punti con vicinanza massima al margine, per n dimensioni saranno $n+1$. Sono gli unici ad essere necessari a definire sia l'iperpiano che i margini.

  La dimensione del margine è $m = \frac{1}{|w|}$; il margine deve essere massimizzato, quindi $|w|$ deve essere minimizzato soggetto a $y_i(b + w \cdot x_i) >= 1$ in modo che nessun punto cada all'interno dei margini.

- soft margin: ci permette di classificare dati non linearmente separabili con una SVM permettendole di commettere errori nella classificazione. La funzione da ottimizzare diventa

$$\arg\min_{w,b} ||w||^2 + \mathcal{C} \sum_i \zeta_i$$

  soggetta a

$$y_i(wx_i + b) \geq 1 - \zeta_i \; \forall i \quad \wedge \quad \zeta_i \geq 0 \; \forall i$$

  ζ_i sono dette slack variables, e permettono al modello di commettere errori sul training set che sono però conteggiati nella loss function da minimizzare.
  $\mathcal{C}$ è un iperparametro che regola il peso degli errori sulla loss function, quindi un valore basso significa alta tolleranza per gli errori (e margine più grande).

- kernel trick: permette la classificazione di dati non linearmente separabili con una SVM attraverso la mappatura del feature space a uno spazio con più dimensioni in cui le classi sono linearmente separabili.

Neural Networks

- (feed-forward) NNs (originariamente Multi-Layer Perceptron o MLP sono un modello composto di svariati percettroni su diversi layer (un input layer, uno o più hidden layers e un output layer).

- grazie ai multipli layer può gestire dati non linearmente separabili, a differenza di perceptron e SVM.

- la funzione di attivazione è applicata all'output di ogni neurone e restringe il range dell'output a [-1, 1] o [0, 1]. Il primo layer spesso ha una funzione di attivazione diverso dagli altri.

  – sigmoid/htan: mappa interamente $\mathbb{R}$, ma difficili da derivare;

  – ReLU (REctifier Linear Unit): $h(z) = max(0, z)$;
  Facile da derivare (utile per gradient descent)

  – Leaky ReLU: $h(z) = max(\alpha z, z)$;
  α è fissato oppure casuale nel caso di Randomized Leaky ReLU

- tutto bello ma non si può addestrare come un perceptron, visto che non conosciamo la funzione obbiettivo per nessun layer a parte l'output. (first AI winter)
- Backpropagation: permette l'addestramento delle NN; 3 fasi:
  - forward propagation: da input in pasto alla NN e li passa avanti fino all'output layer, che ci da un risultato
  - error estimation: il valore calcolato è passato a una loss function che ci ritorna il segnale d'errore;
  - back propagation: il segnale d'errore viene propagato all'indietro nella NN per calcolare il suo gradiente in funzione di tutti i pesi per poi correggere i pesi.
- secondo AI winter: Le NN sono inclini a overfitting e "vanishing gradient": è più difficile correggere i pesi nei primi layer perché il gradiente rispetto ai pesi dei primi layer diventa mano a mano significativo mano a mano che aumentano i layer. (anche la potenza computazionale e i dataset disponibili erano insufficienti)
- per evitare overfitting nelle (feed-fwd) NN si fa early stopping
- come loss function si può usare squared loss
- Gradient Descent con NN:
  - Batch Gradient Descent: ad ogni passo il gradiente della loss è calcolato per l'intero training set; questo garantisce stabilità ma ha un costo computazionale esoso.
  - Stochastic Gradient Descent: il gradiente è calcolato su un solo esempio casuale estratto da $\mathcal{D}_n$; molto più veloce ma instabile. Miglioramenti:
    * Mini-batches: il gradiente è calcolato su "mini-batches" di dati estratti casualmente da $\mathcal{D}_n$ di dimensione indipendente da $|\mathcal{D}_n|$ (spesso 2^n a causa di proprietà di calcolo delle GPU). molto meno instabile ma comunque abbastanza veloce.
    * Momentum: una media decadente degli ultimi valori del gradiente, viene sommato al gradiente per rendere la discesa meno erratica e permette di sfuggire ai punti sella.
    * Adaptive learning rate (o Adagrad): il learning rate è diverso per ogni peso e viene aggiornato in base a quanto sono grandi le derivate passate per quel peso, in pratica rallenta la discesa nelle direzioni ad alta derivata, che a quanto pare è una buona idea per dataset sparsi.
- Convolutional NNs:

  - utili per dati strutturati come immagini, audio o segnali in generale.
  - struttura diversa dalle altre NN: ogni layer è uno di 3 tipi (che sono sempre posti in questo ordine)
    * convolutional layer: Ogni nodo è un filtro convoluzionale applicato ai segnali dallo scorso layer; i pesi della convoluzione (il kernel) sono imparati nel training; un singolo nodo ha kernel diversi per ogni canale sorgente.
    * non linearity: Una funzione di attivazione (come sigmoid o ReLU) è applicata all'intero segnale, aumentando la non linearità e aumentando quindi la generalità della CNN;
    * pooling: rende il segnale più piccolo, essenzialmente downsampling. Metodi comuni sono max-pooling e mean-pooling.

- Long-Short Term Memory NN: TODO

- Auto Encoders:
  - utilizzata per dimensionality reduction automatica
  - due NN con strutture simmetriche l'una all'altra, una chiamata encoder e l'altra decoder.
  - Addestrate attraverso gradient descent con backpropagation. L'errore è dato dalla differenza tra l'input e l'output.
  - a inference time si scarta il decoder e si utilizza soltanto l'encoder, a volte a monte di un'altra NN.
  - Variational Auto Encoder: AE non può essere utilizzato come modello generativo, perché nonostante sia possibile campionare Ω e passare il risultato al decoder il risultato sarà probabilmente privo di significato. Questo è perché l'AE non ha garanzie riguardo alla distribuzione dei dati nello spazio latente Ω, per questa ragione ci serve il VAE.

    L'objective function è:

    $$\theta^\star \in \arg\min_{\theta\in\Theta} d(q_\theta,\ p_{data})$$

    dove

    $$q_\theta(x) = \mathbb{E}_{\omega\sim p_\omega}[q_\theta(x \mid \omega)]$$

    $q_\theta(x|\omega)$ è il decoder, quindi l'obbiettivo è cercare parametri che massimizzino la similarity tra p_{data} e la distribuzione di una gaussiana predefinita p_ω mappata dal decoder.

Nota: il decoder di un VAE è diverso da quello di un normale AE perché per ogni ω ritorna una distribuzione gaussiana su $\mathcal{X}$, invece di un $x \in \mathcal{X}$.

Nella funzione obbiettivo serve un modo per misurare la differenza tra due distribuzioni; un modo è la "KL-divergence":

$$d_{\mathrm{KL}}(p, q) = \mathbb{E}_{x \sim p}\left[\log \frac{p(x)}{q(x)}\right]$$

questa espressione purtroppo è intrattabile quando applicata al nostro caso, ma in qualche modo possiamo collegarci il nostro encoder $q_\psi(\omega \mid x)$ per ottenere ciò che segue:

$$d_{\mathrm{KL}}(p_{\mathrm{data}}, q_\theta) \leq \mathbb{E}_{x \sim p_{\mathrm{data}}}[\text{reconstruction} + \text{regularizer}] + \text{const}$$

dove regularizer $= d_{\mathrm{KL}}(q_\psi(\cdot \mid x), p_\omega)$ e reconstruction è un'espressione complicata. regularizer regola il comportamento dell'encoder; reconstruction regola il decoder in modo che il suo output somigli ai dati di training.

– VQ-VAE ?

- Generative Adversarial Networks:

  – simili a VAE, ma aggiunge un'altra NN chiamata "discriminator" che è un classifier che viene addestrato per predire se un campione è stato generato dal generatore o se è campionato da p_{data}.
  la funzione obbiettivo è la seguente:

$$\theta^\star \in \arg\min_\theta \max_\phi \{\mathbb{E}_{x \sim p_{\mathrm{data}}}[\log t_\phi(x)] + \mathbb{E}_{x \sim q_\theta}[\log(1 - t_\phi(x))]\}$$

  dove θ sono i pesi del generatore q_θ e ϕ sono i pesi del discriminator t_ϕ.
  il primo termine della somma minmaxxa la precisione della predizione del discriminator quando il sample è da p_{data}, il secondo quando è da q_θ
  A inference time il discriminator non è necessario e può essere scartato.
  le GAN sono ovviamente molto sensibili alle architetture scelte per le due NN che le compongono.
  Come per VAE è possibile fare aritmetica sullo spazio latente e ottenere risultati significativi. (per esempio: uomo con occhiali - uomo senza occhiali + donna senza occhiali = donna con gli occhiali)
  problemi

    - instabilità nell'addestramento: i parametri a volte oscillano senza mai convergere;
    - mode collapse: il generatore può imparare a generare perfettamente pochi esempi dal training set;
    - vanishing gradient: se il discriminator diventa molto preciso prima del generatore il gradiente del generator sui suoi pesi è zero, visto che nessun piccolo cambiamento nei pesi può essere sufficiente a sorpassare la precisione del discriminatore;
    - loss function più difficile da calcolare rispetto a quella di VAE (ma più precisa)

  è possibile dare in pasto al training anche le label dei dati (generati o campionati da p_{data}) sia al generatore che al discriminator.

  Progressive GANs: funzionano facendo upscaling (generatore) o downscaling (discriminator) per poter riprodurre immagini di dimensione arbitraria.

  VAE-GAN: ibrido dei due

- Diffusion:
  - aggiungi progressivamente rumore a un segnale in input, e allena una NN a predire la precedente iterazione a partire da un qualsiasi step.
  - con l'aggiunta di rumore la distribuzione diventa sempre più simile a una gaussiana; la NN ha bisogno di poter mappare questa gaussiana a una distribuzione che approssima p_{data} attraverso gaussian convolution.
  - la NN di denoising riceve anche informazioni sul tempo, che viene prima mappato attraverso un segnale sinusoidale (o comunque periodico);
  - la loss è complicata e difficile da calcolare, quindi usiamo un proxy;
  - il campionamento è molto più lento che per GAN o VAE, perché la NN deve essere chiamata molte volte (una per ogni step di denoising)
  - tecniche avanzate di diffusion come stable diffusion danno anche testo codificato in un vettore in pasto ai vari stadi di denoising, per esempio un prompt.

Reinforcement Learning

- un agente interagisce con l'ambiente e riceve ricompense basate sulle sue azioni;
- più formalmente, ad ogni tempo t:
  - l'ambiente fornisce all'attore lo stato s_t (basato s_{t-1} e a_{t-1})

  - l'attore reagisce con un'azione a_t
  - l'ambiente fornisce la ricompensa r_t

- È un Markov Decision Process(MDP):
  - composto da:
    * Stati s_t che iniziano da s_0;
    * Azioni a;
    * Transition model $P(s'|s, a)$;
    * Reward function $r(s)$
  - MDP è utile per ricavare la Policy $\pi(s)$ (azione dell'agente in funzione dello stato)
  - s_0, r_t, s_{t+1} sono probabilistici, cioè campionati da distribuzioni (che prendono in considerazione s_t e a_t se rilevanti).
  - l'obbiettivo è trovare la policy $\pi^\star$ che massimizza il Cumulative Discounted Reward.

    cumulative discounted reward:

    $$\sum_{t\geq 0} \gamma^t r(s_t) \;\; | \;\; 0 < \gamma \leq 1$$

    il discount factor γ regola quanto l'agente prioritizza ricompense immediate rispetto a ricompense a lungo termine

- problemi (relativamente a supervised learning in generale):
  - s_{t+1} dipende da a_t
  - supervised input (cioè il reward) è spesso "sparse"
  - la ricompensa non è differenziabile rispetto ai parametri del modello

- Value-based Reinforcement Learning: il modello cerca di stimare il valore di ogni stato, invece di cercare di modellare la policy $\pi^\star$

  Q-value è una funzione di valore che prende in considerazione non solo lo stato ma anche l'azione.

  La Bellman Equation ci permette di usare dynamic programming per riempire una tabella con tutti i Q-values in funzione dei Q-values degli stati a cui portano; questa tabella in realtà è conosciuta solamente all'ambiente, mentre l'agente deve calcolarla in base all'esperienza.

per spazi degli stati più complessi la funzione Q può essere approssimata con una NN, che prende in input uno stato e ritorna il Q-value per ogni azione quando accoppiata a quello stato.

- Policy Gradient: se l'action space è continuo (per esempio nel caso del movimento robotico) possiamo addestrare una NN stimando il gradiente della ricompensa futura rispetto alla policy e aggiornare i parametri della policy con gradient ascent.