Introduction to machine learning

Giacomo Fantoni
Telegram: @GiacomoFantoni
Github: https://github.com/giacThePhantom/intro2ml

18 maggio 2021

Indice

Capitolo 1

Introduzione

1.1 Definizioni

Si intende per machine learning lo studio di algoritmi che migliorano autonomamente attraverso l'esperienza. È un campo dell'intelligenza artificiale. Stravolge il paradigma convenzionale della programmazione: un algoritmo di machine learning infatti prende come input un insieme di dati e risultati in modo da produrre un programma che fornisce un risultato appropriato. Coinvolge pertanto la scoperta automatica di regolarità nei dati attraverso algoritmi in modo da poter compiere azioni basate su di essi.

1.2 Processo

Il machine learning permette ai computer di acquisire conoscenza attraverso algoritmi che inferiscono e imparano da dati. Questa conoscenza viene rappresentata da un modello che può essere utilizzato su nuovi dati.

1.2.1 Il processo di apprendimento

Il processo di apprendimento in particolare coinvolge diversi passaggi:

- Acquisizione dei dati dal mondo reale attraverso dispositivi di misurazione come sensori o database.
- Preprocessamento dei dati: filtraggio del rumore, estrazione delle feature e normalizzazione.
- Riduzione dimensionale: selezione e proiezione delle feature.
- Apprendimento del modello: classificazione, regressione, clustering e descrizione.
- Test del modello: cross-validation e bootstrap.
- Analisi dei risultati.

1.3 Modello

Un algoritmo di machine learning impara dall'esperienza E in rispetto di una classe di compiti T e di misurazione delle performance P, se la P di T aumenta con E. Si nota pertanto come un compito

di machine learning ben definito possiede una tripla:

$$< T, P, E >$$

1.4 Deep learning

Il deep learning è un sottoinsieme del machine learning che permette a modelli computazionali composti di multipli strati di imparare la rappresentazione di dati con multipli livelli di astrazione. Si utilizza pertanto una rete neurale con diversi strati di nodi tra input e output. Questa serie di strati tra input e output computa caratteristiche rilevanti automaticamente in una serie di passaggi. Questi algoritmi sono resi possibili da:

- Enorme mole di dati disponibili.
- Aumento del potere computazionale.
- Aumento del numero di algoritmi di machine learning e della teoria sviluppata dai ricercatori.
- Aumento del supporto dall'industria.

Capitolo 2

Machine learning basics

2.1 Introduzione

Il machine learning permette ai computer di acquisire conoscenza attraverso algoritmi che imparano e inferiscono dai dati. Tale conoscenza viene rappresentata da un modello che viene poi utilizzato su dati futuri.

2.1.1 Processo di learning

Si individua un processo di learning:

- Acquisizione di dati dal mondo reale attraverso sensori.
- Preprocessamento dei dati: eliminazione del rumore, estrazione delle features e normalizzazione.
- Riduzione di dimensionalità attraverso selezione e proiezione di features.
- Learning del modello: classification, regression, clustering e description.
- Test del modello attraverso cross-validation e bootstrap.
- Analisi dei risultati.

2.2 Dati

I dati disponibili ad un algoritmo di machine learning sono tipicamente un insieme di esempi. Questi esempi sono tipicamente rappresentati come un array di features, caratteristiche dei dati di interesse per lo studio in atto.

2.2.1 Training e test set

In particolare per questi algoritmi si assume sempre che il training e il test set siano distribuiti secondo variabili indipendenti e identicamente distribuite ($i.i.d$) La distribuzione P_{data} è tipicamente sconosciuta ma si può campionare, attraverso un modello probabilistico di learning. In particolare la distribuzione di probabilità di coppie di esempio e label viene detta data generating distribution e sia il training data che il test set sono generati basandosi su di essa.

2.3 Task

Si intende per task una rappresentazione del tipo di predizione che viene svolta per risolvere un problema su dei dati. Viene identificata con un insieme di funzioni che possono potenzialmente risolverla. In generale consiste di una funzione che assegna ogni input $x \in \mathcal{X}$ a un output $y \in \mathcal{Y}$:

$$f : \mathcal{X} \to \mathcal{Y} \qquad \mathcal{F}_{task} \subset \mathcal{Y}^{\mathcal{X}}$$

La natura di $\mathcal{X}, \mathcal{Y}, \mathcal{F}_{task}$ dipende dal tipo di task.

2.4 Modello

Un modello è un programma per risolvere un problema. È cioè l'implementazione di una funzione $f \in \mathcal{F}_{task}$ che può essere computata. Un insieme di modelli formano uno spazio di ipotesi:

$$\mathcal{H} \subset \mathcal{F}_{task}$$

L'algoritmo cerca una soluzione nello spazio di ipotesi.

2.4.1 Target ideale

Il target ideale del modello è quello di minimizzare una funzione di errore (generalizzazione)

$$E(f; P_{data})$$

Questa funzione determina quanto bene una soluzione $f \in \mathcal{F}_{task}$ fitta dei dati. Guida pertanto la selezione della migliore soluzione in $\mathcal{F}_{task}$. Pertanto:

$$f^* \in arg \min_{f \in \mathcal{F}_{task}} E(f; P_{data})$$

2.4.2 Target feasible

Si deve restringere il focus sul trovare funzioni che possono essere implementate e valutate in maniera trattabile. Si definisce pertanto uno spazio di ipotesi del modello $\mathcal{H} \subset \mathcal{F}_{task}$ e si cerca la soluzione all'interno di quello spazio:

$$f^*_{\mathcal{H}} \in arg \min_{f \in \mathcal{H}} E(f; P_{data})$$

Si noti come questa funzione non possa essere computata correttamente in quanto P_{data} è sconosciuta.

2.4.3 Target attuale

Per trovare il target attuale si deve lavorare su un campione di dati o il training set

$$\mathcal{D}_n = \{z_1, \ldots, z_n\}$$

Dove

$$z_i = (x_i, y_i) \in \mathcal{X} \times \mathcal{Y}$$
$$z_i \sim P_{data}$$

Pertanto in:

$$f^*_{\mathcal{H}} \in arg \min_{f \in \mathcal{H}} E(f; P_{data})$$

$E(f; P_{data})$ è il training error.

2.4.4 Funzione di errore

Le funzioni di generalizzazione e di training error possono essere scritte in termini di una pointwise loss $l(f; z)$ che misura l'errore che avviene a f su un esempio di training z.

$$E(f; P_{data}) = \mathbb{E}_{z \sim P_{data}}[l(f; z)]$$

$$E(f; \mathcal{D}_n) = \frac{1}{n} \sum_{i=1}^{n} l(f; z_i)$$

Si nota pertanto come l'algoritmo di learning risolve il problema di ottimizzazione con target:

$$f^*_{\mathcal{H}}(\mathcal{D}_n)$$

2.4.5 Tipi di errore

- Underfitting il modello non fitta sui dati di training, avviene per mancanza di dati.
- Overfitting quando i training data sono rumorosi e il modello li fitta perfettamente, ma impara un modello che si adatta solo su di essi e non fitta dati dal mondo reale.
- Estimation error, indotto imparando da un campione di dati.
- Approximation error, indotto dallo spazio di ipotesi $\mathcal{H}$.
- Irreducible error a causa della variabilità intrinseca.

2.4.6 Stimare l'errore di generalizzazione

L'errore di geenralizzazione può essere stimato utilizzando diversi insiemi di training, validation e test. Si usa il training set per fare training di un modello, quello di validazione per valutarlo e sistemare i suoi iperparametri e dopo di quello si sceglie il modello migliore che si misura attraverso le performance sul test set.

2.4.6.1 Migliorare la generalizzazione

La generalizzazione può essere migliorata:

- Evitando di ottenere il minimo sul training error.
- Riducendo la capacità del modello.
- Cambiando l'obiettivo con un termine di regolarizzazione.
- Iniettando rumore nell'algoritmo.
- Fermando l'algoritmo prima che converga.
- Aumentantdo la quantità di dati.
- Aggiungendo più campioni di training.
- Aumentando il training set con trasformazioni.
- Combinando predizioni da più modelli decorrelati o ensembling.

2.4.6.1.1 Regolarizzazione Si intende per regolarizzazione la modifica della funzione di training error con un termine $\Omega(f)$ che penalizza soluzioni complesse:

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

2.5 Tipi di learning

2.5.1 Supervised learning

Nel supervised learning vengono dati in input a un modello o predittore un insieme di esempi che possiedono una label. Il modello poi impara a creare delle predizioni su un nuovo esempio.

2.5.1.1 Dati

Nel caso del supervised learning i dati creano una distribuzione:

$$p_{data} \in \Delta(\mathcal{X} \times \mathcal{Y})$$

2.5.1.2 Classificazione

In un problema di classificazione si trova un insieme finito di label discrete. In particolare dato un training set $\mathcal{T} = \{(x_1, u_1), \dots (x_m, y_m)\}$, si deve imparare una funzione f per predirre y dato x. f sarà pertanto:

$$f : \mathbb{R}^d \to \{1, 2, \dots, k\}$$

Dove d è la dimensionalità di x e k il numero di labels distinte.

2.5.1.2.1 Task Si deve pertanto trovare una funzione $f \in \mathcal{Y}^{\mathcal{X}}$ che assegna ogni input $x \in \mathcal{X}$ a una label discreta.

$$f(x) \in \mathcal{Y} = \{c_1, \dots, c_k\}$$

2.5.1.3 Regression

Un problema di regressione presenta un insieme di label continue. Dato un training set $\mathcal{T} = \{(x_1, y_1), \dots, (x_m, y_m)\}$, si deve imparare una funzione f per predirre y dato x. f sarà pertanto:

$$f : \mathbb{R}^d \to \mathbb{R}$$

Dove d è la dimensionalità di x.

2.5.1.3.1 Task Si deve trovare una funzione $f(x) \in \mathcal{Y}$ che assegna ogni input a una label continua.

2.5.1.4 Ranking

Il ranking è un tipo particolare di classificazione in cui una label è un ranking.

2.5.2 Unsupervised learning

Nel unsupervised learning vengono dati in input a un modello o predittore un insieme di esempi senza label. Il modello impara a creare delle predizioni su un nuovo esempio.

2.5.2.1 Dati

Nel caso del supervised learning i dati creano una distribuzione:

$$p_{data} \in \Delta(\mathcal{X})$$

2.5.2.2 Clustering

Nel clustering, data $\mathcal{T} = \{x_1, \ldots, x_m\}$ si deve trovare la struttura nascosta che intercorre tra le x o i clusters.

2.5.2.2.1 Task Si deve trovare una funzione $f \in \mathbb{N}^{\mathcal{X}}$ che assegna ogni input $x \in \mathcal{X}$ a un indice di cluster $f(x) \in \mathbb{N}$. Tutti i punti mappati sullo stesso indice formano un cluster.

2.5.2.3 Dimensionality reduction

Nella dimensionalty reduction si tenta di ridurre il numero di variabili sotto considerazione ottenendo un insieme di variabili principali.

2.5.2.3.1 Task Si deve trovare una funzione $f \in \mathcal{Y}^{\mathcal{X}}$ che mappa ogni input di molte dimensioni $x \in \mathcal{X}$ a un output a dimensione minore $f(x) \in \mathcal{Y}$, dove $dim(\mathcal{Y} \ll dim(\mathcal{X})$

2.5.3 Reinforcement learning

Nel reinforcement learning un agente impara dall'ambiente interagendo con esso e ricevendo premi per lo svolgimento di azioni particolari. In particolare, data una sequenza di esempi o stati e una reward dopo il completamento di tale sequenza si impara a predirre l'azione da svolgere per uno stato o esempio individuale.

Capitolo 3

KNN

3.1 Introduzione

Si possono considerare gli esempi come punti in uno spazio n dimensionale dove n è il numero di features. Per classificare un esempio d si può mettere a d una label uguale a quella dell'esempio più vicino a d nel training set. Questo concetto viene esteso nel K-nearest neighbour o *K-NN* in cui per classificare un esempio d si trovano i k esempi più vicini di d e si sceglie la label in maggior numero tra i k vicini più prossimi.

3.2 Misurare la distanza

Misurare la distanza tra due esempi è specifico al problema, ma un modo possibile è la distanza euclidea:

$$D(a,b) = \sqrt{(a_1 - b_1)^2 + \cdots + (a_n - b_n)^2}$$

3.3 Decision boundaries

I decision boundaries sono posti nello spazio delle features dove la classificazione di un punto o un esempio cambia. In particolare *K-NN* definisce dei decision boundaries localmente tra le classi.

3.4 Il ruolo di K

I fattori che determinano la bontà di un algoritmo di machine learning sono la sua abilità di minimizzare il training error e minimizzare il gap tra il training error e il test error. Questi due fattori corrispondono a underfitting e overfitting.

3.4.1 Underfitting

L'underfitting avviene quando il modello non è capace di ottenere un valore di errore abbastanza piccolo sul training set.

3.4.2 Overfitting

L'overfitting avviene quando il gap tra il training error e il test error è troppo grande.

3.5 Scelta di K

Il valore di K comprende euristiche comuni come $3, 5, 7$ o un numero dispari per evitare pareggi. Può essere scelto utilizzando dati di sviluppo. Per classificare un esempio d si trovano i k vicini più prossimi di d e si sceglie la classe più presente nei k scelti.

3.6 Variazioni di K

Invece di scegliere i K vicini più prossimi si possono contare tutti gli esempi in una distanza fissata.

3.6.1 *K-NN* pesata

Si può pesare il voto di tutti gli esempi in modo che esempi più vicini pesino di più. Si usa speso qualche tipo di decadimento esponenziale.

Capitolo 4

Modelli lineari

4.1 Introduzione

Alcuni approcci del machine learning fanno delle forti assunzioni riguardo i dati. Questo avviene in quanto se le assunzioni sono vere si possono raggiungere performance migliori. In caso contrario l'approccio può fallire miseramente. Altri approcci che non fanno molte assunzioni riguardo i dati invece permettono di imparare da dati più vari ma sono più proni a overfitting e richiedono più dati di training.

4.1.1 Bias

Il bias di un modello è quanto forte le assunzioni del modello sono. I classificatori a low-bias fanno delle assunzioni minime riguardo i dati come *k-NN* (che assume unicamente che la vicinanza è correlata alla classe) e *DT*. I classificatori a high-bias fanno assunzioni forti riguardo ai dati.

4.2 Linear separability

L'assunzione strong-bias dei modelli lineari è la linear separability, ovvero che in due dimensioni le classi si possano separare attraverso una linea, mentre in dimensioni maggiori da un hyperplane. Un modello lineare è pertanto un modello che assume che i dati sono linearmente separabili.

4.2.1 Definire una linea

Ogni coppia di valori (w_1, w_2) definisce una linea attraverso l'origine:

$$0 = w_1 f_1 + w_2 f_2$$

Si può inoltre vedere il vettore $\overrightarrow{w} = (w_1, w_2)$ come il vettore dei pesi perpendicolare alla linea. Per classificare i punti rispetto alla linea si considera il segno sostituendo i punti a f_1 e f_2. La positività o negatività indica il lato della linea. Si può estendere l'equazione con:

$$a = w_1 f_1 + w_2 f_2$$

In questo modo la linea interseca l'asse delle y in a.

4.3 Definizione di modello lineare

Si definisce un modello lineare in uno spazio n dimensionale, dove n è il numero di features attraverso $n + 1$ pesi.

$$0 = b + \sum_{i=1}^{n} w_i f_i$$

In un modello lineare si classifica un nuovo esempio moltiplicandolo con il vettore dei pesi, aggiungendo il bias e controllando il segno del risultato. Questo determina la classe dell'esempio.

4.3.1 Training

Il training di un modello lineare avviene online, ovvero a differenza del modo in batch in cui vengono dati i training data come $\{(x_i, y_i) : 1 \leq i \leq n\}$, i data points arrivano uno alla volta. L'algoritmo allora riceve un esempio x_i senza label, predice la classificazione di questo esempio e confronta la predizione con l'effettivo y_i. Infine aggiorna il proprio modello.

4.3.2 Perceptron

4.3.2.1 Numero di iterazioni

Il numero di iterazioni del perceptron viene deciso in base alla convergenza. Inoltre può essere limitato in modo da ridurre l'overfitting. Si noti come in caso di dati non lineramente separabili la convergenza non avviene mai.

4.3.2.2 Ordine dei campioni

I campioni da considerare nel perceptron sono considerati in ordine casuale. In questo modo si produce un modello a low bias.

4.3.2.3 Linear separable sets

Le istanze di training sono lienarmente separabili se esiste un hyperplane che separa le due classi.

4.3.2.4 Algoritmo

4.3.2.5 Calcolo della predizione

4.4 Perceptron e reti neurali

Si può immaginare il perceptron come un neurone artificiale o una funzione parametrizzata non lineare con un valore di attivazione soglia e un range di output ristretto. Le reti neurali sono reti di neuroni artificiali densamente connessi in modo da simulare la rete di neuroni del cervello.

4.4.1 Funzione di attivazione

Una funzione di attivazione può essere una soglia dura: se la somma di tutti gli input è maggiore di un valore allora il perceptron manda il segnale queste permettono di imparare solo modelli lineari. Funzioni di attivazione più interessanti sono le sigmoidi, tangenti iperboliche, ReLU o rectified linear unit o leaky ReLU.

```
: Perceptron()
repeat
    foreach training example (f_1, f_2, ..., f_n, label) do
        %Label = ±1
        check if it is correct based on the current label
        if not correct then
            %update all the weights
            foreach w_i do
                w_i = w_i + f_i * label
                b = b + label
until Convergence
%Or some number of iteration
```

```
: Perceptron()
repeat
    foreach training example (f_1, f_2, ..., f_n, label) do
        %Label = ±1
        prediction = b + sum_{i=1}^{n} w_i f_i
        if prediction is not label then
            %update all the weights
            foreach w_i do
                w_i = w_i + f_i * label
                b = b + label
until Convergence
%Or some number of iteration
```

4.4.2 Storia del perceptron

Il perceptron nasce nel 1958 da parte di Rosemblatt che lo crea con una soglia dura. Questo gli impedisce di imparare modelli non lineari come lo *xor*. Viene superato nel 1986 attraverso perceptrons multi layers e backpropagation e utilizzando una soglia meno dura.

Capitolo 5

Decision Trees

5.1 Struttura

Un decision tree è un modello di predizione con struttura ad albero. È composto da nodi terminali o foglie e nodi non terminali. I nodi non terminali hanno da due a più figli e implementando la funzione di routing. I nodi foglia non hanno figli e implementano la funzione di predizione. Non ci sono cicli e tutti i nodi hanno al massimo un genitore (con esclusione del nodo radice).

5.2 Funzionamento

Un decision tree prende un input $x \in \mathcal{X}$ e lo routa attraverso i nodi fino a che raggiunge un nodo foglia dove avviene la predizione. Ogni nodo non terminale

$$Node(\phi, t_L, t_R)$$

Contiene una funzione di routing $\phi \in \{L, R\}^{\mathcal{X}}$, un figlio destro t_L e un figlio sinistro t_R. Quando x raggiunge il nodo viene spostato sul figlio destro o sinistro in base al valore di $\phi(x) \in \{L, R\}$. Ogni nodo foglia

$$Leaf(h)$$

Contiene una funzione di predizione $h \in \mathcal{F}_{task}$, tipicamente una costante.

5.2.1 Inferenza

Sia f_t la funzione che ritorna la predizione per l'input $x \in \mathcal{X}$ secondo il decision tree t. Questa viene definita come:

$$f_t(x) = \begin{cases} h(t) & \textbf{if } t = Leaf(h) \\ f_{t_{\phi(x)}}(x) & \textbf{if } t = Node(\phi, t_L, t_R) \end{cases}$$

5.3 Decision trees learning algorithm

Dato un training set $\mathcal{D}_n = \{z_1, \ldots, z_n\}$ si deve trovare f_{t^*} dove:

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

Dove $\mathcal{T}$ è l'insieme dei decision trees. Il problema di ottimizzazione è facile se non si impongono constraints, altrimenti potrebbe diventare *NP-hard*. Una soluzione può essere trovata utilizzando una strategia greedy. Pertanto si assume:

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

Ora fissato un insieme di predizioni di foglie

$$\mathcal{H}_{leaf} \subset \mathcal{F}_{task}$$

E fissato un insieme di possibili funzioni di routing o split

$$\Phi \subset \{L, R\}^{\mathcal{X}}$$

La strategia di crescita dell'albero partiziona ricorsivamente il training set e decide se crescere foglie o nodi non terminali.

5.3.1 Crescere una foglia

Sia $\mathcal{D} = \{z_1, \ldots, z_m\}$ il training set che raggiunge un nodo. Il predittore di foglia ottimale viene computato come:

$$h^*_{\mathcal{D}} \in arg \min_{h \in \mathcal{H}_{leaf}} E(h; \mathcal{D})$$

Il valore di errore ottimale o misura di impurità:

$$I(\mathcal{D}) = E(h^*_{\mathcal{D}}; \mathcal{D})$$

Dove $h_{\mathcal{D}}$ è la predizione per il dataset $\mathcal{D}$. L'impurità è computata nel nodo in cui arriva il sottoinsieme del dataset originale $\mathcal{D}$. In quel nodo si calcola il numero di errori che si farebbero classificando tutti i dati del sottoinsieme in una singola classe c, ripetendo il calcolo per ogni classe. Il numero minimo di errori che fai con una certa classe c^* è la misura di impurità del classification error. Se si raggiungono dei criteri si cresce una foglia $Leaf(h^*_{\mathcal{D}})$ Esempi di questi criteri sono la purezza $I(\mathcal{D}) < \epsilon$, la cardinalità minima $|\mathcal{D}| < k$ o un'altezza massima dell'albero.

5.3.2 Crescere un nodo

Se non si raggiunge il criterio si deve trovare una funzione di split ottimale:

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

L'impurità $I_\phi(\mathcal{D})$ di una funzione di split ϕ dato il training set $\mathcal{D}$ viene computata nei termini di impurità dei dati splittati:

$$I_\phi(\mathcal{D}) = \sum_{d \in \{L,R\}} \frac{|\mathcal{D}^\phi_d|}{|\mathcal{D}|} I(\mathcal{D}^\phi_d)$$

Dove

$$\mathcal{D}^\phi_d = \{(x, y) \in \mathcal{D}; \phi(x) = d\}$$

L'impurità di una funzione di split è il più basso errore di training che può essere ottenuto da un albero che consiste di una radice e due figlie. Si cresce pertanto un nodo $Node(\phi^*, t_L, t_R)$ dove ϕ^* è lo split ottimale, mentre t_L e t_R sono ottenuti applicando ricorsivamente l'algoritmo di learning ai training set splits.

5.3.3 Algoritmo

$$Grow(\mathcal{D}) = \begin{cases} Leaf(h^*_{\mathcal{D}}) & \textit{raggiunto criterio di stop} \\ Node(\phi^*_{\mathcal{D}}, Grow(\mathcal{D}^*_L), Grow(\mathcal{D}^*_R)) & \textit{altrimenti} \end{cases}$$

Dove $\mathcal{D}^*_d = \{(x, y) \in \mathcal{D}; \phi^*_{\mathcal{D}}(x) = d\}$.

5.3.4 Split selection

Tipicamente la migliore funzione di split viene data in termini di minimizzazione dell'impurità dello split, ma altre volte nella massimizzazione del guadagno di informazioni o

$$\Delta_\phi(\mathcal{D}) = I(\mathcal{D}) - I_\phi(\mathcal{D})$$

Essendo $\Delta_\phi(\mathcal{D}) \geq 0$ per ogni $\phi \in \{L, R\}^{\mathcal{X}}$ e ogni training set $\mathcal{D} \subset \mathcal{X} \times \mathcal{Y}$, l'impurità non aumentera mai per ogni split scelto casualmente.

5.3.5 Predizione delle foglie

La predizione delle foglie fornisce una soluzione a un problema semplificato coinvolgendo solo dati che la raggiungono. Questa soluzione può essere una funzione arbitraria $h \in \mathcal{F}_{task}$, ma in pratica si restringe a un sottoinsieme di $\mathcal{H}_{leaf}$. Il predittore più semplice è una funzione che ritorna una costante (come una label). L'insieme di tutte le possibili funzioni costanti può essere scritto come:

$$\mathcal{H}_{leaf} = \bigcup_{y \in \mathcal{Y}} \{y\}^{\mathcal{X}}$$

5.4 Misure di impurità per la classificazione

Si consideri per la classificazione:

$$\mathcal{Y} = \{c_1, \ldots, c_k\} \qquad\qquad \mathcal{D} \subset \mathcal{X} \times \mathcal{Y}$$

Sia $\mathcal{D}^y = \{(x, y') \in \mathcal{D} : y = y'\}$, che denota il sottoinsieme di training samples in $\mathcal{D}$ con label y. Considerando la funzione di errore:

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

Se $l(f; (x, y)) = 1_{f(x) \neq y}$ e $\mathcal{H}_{leaf} = \bigcup_y \in \mathcal{Y}\{y\}^{\mathcal{X}}$ la misura di impurità è allora il classification error:

$$I(\mathcal{D}) = 1 - \max_{y \in \mathcal{Y}} \frac{|\mathcal{D}^y|}{|\mathcal{D}|}$$

Se invece $l(f; (x, y)) = \sum_{x \in \mathcal{Y}} [f_c(x) - 1_{c=y}]^2$ e $\mathcal{H}_{leaf} = \bigcup_{\pi \in \Delta(\mathcal{Y})} \{\pi\}^{\mathcal{X}}$ allora la misura di impurità è l'impurità di Gini:

$$I(\mathcal{D}) = 1 - \sum_{y \in \mathcal{Y}} \left(\frac{|\mathcal{D}^y|}{|\mathcal{D}|} \right)^2$$

Infine se $l(f;(x,y)) = -\log f_y(x)$ e $\mathcal{H}_{leaf} = \bigcup_{\pi \in \Delta(\mathcal{Y})} \{\pi\}^{\mathcal{X}}$, con una distribuzione costante di label come predizione di foglie, allora la misura di impurità è l'entropia:

$$I(\mathcal{D}) = -\sum_{y \in \mathcal{Y}} \frac{|\mathcal{D}^y|}{|\mathcal{D}|} \log\log \frac{|\mathcal{D}^y|}{|\mathcal{D}|}$$

5.5 Misure di impurità per la regressione

Si consideri per la regressione:

$$\mathcal{Y} \subset \mathbb{R}^d \qquad\qquad \mathcal{D} \subset \mathcal{X} \times \mathcal{Y}$$

Se $l(f;(x,y)) = ||f(x) - y||_2$ e $\mathcal{H}_{leaf} = \bigcup_{y \in \mathcal{Y}} \{y\}^{\mathcal{X}}$ allora la misura di impurità è la varianza:

$$I(\mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{(x,y) \in \mathcal{D}} ||x - \mu_{\mathcal{D}}||^2$$

Dove $\mu_{\mathcal{D}} = \frac{1}{|\mathcal{D}|} \sum_{(x,y) \in \mathcal{D}} x$

5.6 Data features e attributi

Un data point $x \in \mathcal{X}$ potrebbe essere d dimensionale con ogni dimensione con tipi di valori eterogenei come discreti o continui e avere un ordinamento o no, rispettivamente ordinali o nominali.

5.7 Funzioni di split o routing

Il routing o split $\phi \in \{L, R\}^{\mathcal{X}}$ determina se un data point $x \in \mathcal{X}$ deve muoversi a destra o sinistra. La possibile funzione di split è ristretta in un insieme predefinito $\Phi \subset \{L, R\}^{\mathcal{X}}$ in base alla natura dello spazio di features. La funzione di split prototipica per un input d dimensionale prima seleziona una dimensione e poi applica un criterio di split 1 dimensionale.

5.7.1 Features discrete e nominali

Si assumano features discrete e nominali con valori in $\mathcal{K}$. La funzione di split può essere implementata data una partizione di $\mathcal{K}$ in $\mathcal{K}_R$ e $\mathcal{K}_L$:

$$\phi(x) = \begin{cases} L & \textbf{if} x \in \mathcal{K}_L \\ R & \textbf{if} x \in \mathcal{K}_R \end{cases}$$

Trovare lo split ottimale richiede testare $2^{|\mathcal{K}|-1} - 1$ bi-partizioni.

5.7.2 Features ordinali

Si assumano features ordinali con valori in $\mathcal{K}$. La funzione di split può essere implementata dando una soglia $r \in \mathcal{K}$:

$$\phi(x) = \begin{cases} L & \textbf{if} x \leq r \\ R & \textbf{if} x > r \end{cases}$$

Se $|\mathcal{K}| \leq |\mathcal{D}|$ trovare lo split ottimale richiede il test di $|\mathcal{K}| - 1$ soglie. Se $|\mathcal{K}| > |\mathcal{D}|$ si deve ordinare i valori di input in $\mathcal{D}$ dove $\mathcal{D}$ è il training set che raggiunge il nodo e testare $|\mathcal{D}| - 1$ soglie.

5.7.3 Obliquo

A volte è conveniente fare split considerando più features alla volta. Tali funzioni lavorano con features continue e sono dette oblique in quanto generano decision boundaries obliqui. Se $x \in \mathbb{R}^d$ allora la funzione di split può essere implementata dato $w \in \mathbb{R}^d$ e $r \in \mathbb{R}$:

$$\phi(x) = \begin{cases} L & \textbf{if} w^T x \leq r \\ R & \textbf{altimenti} \end{cases}$$

Si nota come questa funzione sia più difficile da ottimizzare.

5.8 Decision trees e overfitting

I decision trees sono modelli non parametrici con una struttura determinata dai dati. Per questo sono flessibili e possono facilmente fare fit sul training set, con un alto rischio di overfitting. Tecniche standard per migliorare la generalizzazione si applicano ai decision trees:

- Early stopping.
- Complexity reduction.
- Regularization.
- Data augmentation.
- Ensembling.

Una tecnica per ridurre la complessità a posteriori è detta pruning.

5.9 Random forest

Le random forest sono ensembles di decision trees. Ogni algero è tipicamente trained con una versione bootstrapped del training set campionata con sostituzione. Le funzioni di split sono ottimizzate su features campionate a caso o completamente a caso (extremely randomized trees). Questo aiuta ad ottenere decision trees decorrelati. La predizione finale della foresta è ottenuta facendo la media delle predizioni per ogni albero nell'ensemble $\mathcal{Q} = \{t_1, \ldots, t_T\}$.

$$f_{\mathcal{Q}}(x) = \frac{1}{T} \sum_{j=1}^{T} f_t(x)$$

5.10 Confronto con KNN

Si nota come a differenza dei decision trees KNN non richiede nessun training, ma la classificazione è più veloce per i decision trees in quanto KNN per ogni esempio deve calcolare k distanze. A differenza di KNN che tratta tutte le features in maniera uguale i decision trees permettono una selezione delle features più importanti facendo scelte pesate.

Capitolo 6

Multi class classification

6.1 Introduzione

6.1.1 Classificazione binaria

Si è definita la classificazione binaria come la task in cui dati:

- Uno spazio di input $\mathcal{X}$.
- Una distribuzione sconosciuta $\mathcal{D}$ su $\mathcal{X} \times \{-1, +1\}$.
- Un training set D campionato da $\mathcal{D}$

Si deve computare una funzione f che minimizza $\mathbb{E}_{(x,y)\sim\mathcal{D}}[f(x) \neq y]$

6.1.2 Classificazione multi classe

La classificazione multiclasse è l'estensione naturale della classificazione binaria. L'obiettivo è quello di assegnare una label discreta a degli esempi. La differenza è che ora ci sono $k > 2$ classi da cui scegliere. Dati pertanto:

- Uno spazio di input $\mathcal{X}$ e un numero di classi K. $[K]$.
- Una distribuzione sconosciuta di $\mathcal{D}$ su $\mathcal{X} \times [K]$.
- Un training set D campionato da $\mathcal{D}$.

Si deve computare una funzione f che minimizza $\mathbb{E}_{(x,y)\sim\mathcal{D}}[f(x) \neq y]$.

6.1.2.1 K nearest neighbours

Si noti come una *K-NN* per classificare un esempio d trova i k vicini di d e sceglie la label in presenza maggiore tra i k vicini più prossimi. Non necessita pertanto di cambi algoritmici nel caso della multi class classification.

6.1.2.2 Decision tree

I decision tree non richiedono cambi algoritmici per la multi class classification.

6.1.3 Approccio black box alla multi class classification

Dato un classificatore binario questo si può usare per risolvere il problema della multiclass classification. Si noti come un perceptron oltre al risultato può anche dare un punteggio di confidenza. Inoltre siccome una linea non è sufficiente per dividere le classi se ne possono usare diverse.

6.2 One versus all *OVA*

Nell'approccio *OVA* nel training si definisce per ogni label L un problema binario in cui:

- Tutti gli esempi con la label L sono positivi.
- Tutti gli altri esempi sono negativi.

In pratica si imparano L diversi modelli di classificazione. Si ricordi come il classificatore divide il piano in due semipiani.

6.2.1 Ambiguità

In questo caso si formano pertanto delle zone in cui si creano delle ambiguità. Se il classificatore non fornisce confidence e c'è ambiguità si sceglie una delle label in conflitto. Nella maggior parte dei casi i classificatori forniscono confidence, allora in questo caso si:

- Si sceglie il positivo con confidence maggiore.
- Se nessuno è positivo si sceglie il negativo con confidence minore.

La confidence nel perceptron si calcola come distanza dall'iperpiano stabilito dalla prediction.

6.2.2 Algoritmi

: `OneVersusAllTrain(`$D^{multiclass}$`, BinaryTrain())`

for $i = 1$ **to** K **do**
  D^{bin} = relabel $D^{multiclass}$ in modo che i è positivo e $\neq i$ è negativo
  f_i = `BinaryTrain(`D^{bin}`)`
Return $f_1, \ldots, f_K$

: `OneVersusAllTest(`$f_1, \ldots, f_K$`,` $\hat{x}$`)`

score = $\langle 0, \ldots, 0 \rangle$ %`Inizializza` K `score a` 0
for $i = 1$ **to** K **do**
  y = $f_i(\hat{x})$
  $score_i = score_i + y$
Return `max(`*score*`)`

6.3 All versus all *AVA*

Un approccio alternativo consiste nel gestire il problema della classificazione multi classe decomponendolo in problemi di classificazione binaria. Questo approccio viene detto anche all pairs. Si classificano $\frac{K(K-1)}{2}$ classificatori in modo che

$$F_{ij}, 1 \leq i < j \leq K$$

Sia il classificatore che discrimina la classe i contro la classe j. Questo classificatore riceve tutti gli esempi della classe i come positivi e tutti gli esempi della classe j come negativi. Quando arriva un punto di test si valuta su tutti i classificatori F_{ij}. Ogni volta che F_{ij} predice positivo la classe i prende un voto, altrimenti lo prende j. Dopo aver eseguito tutti i $\frac{K(K-1)}{2}$ classificatori la classe con più voti decide la label.

6.3.1 *AVA* training

Per ogni coppia di label si addestra un classificatore che le distingue:

`: AllVersusAllTrain()`

for i = *1* **to** *number of labels* **do**
  for $j = i + 1$ **to** *number of labels* **do**
    train a classifier F_{ij} to distinguish between $label_j$ and $label_i$
    create a dataset with all examples with $label_j$ labeled positive and with $label_i$ negative
    Train classifier F_{ij} su questo sottoinsieme di dati

6.3.2 *AVA* classification

Per classificare un esempio x lo si classifica per ogni classificatore F_{ij}. Per scegliere la classe finale si può:

- Considerare la maggioranza.
- Considerare un voto pesato basato sulla confidence:
  - $y = F_{ij}(x)$.
  - $score_j += y$.
  - $score_i -= y$.

Lo score viene cambiato in quanto se y è positivo il classificatore lo pensa di tipo j, se negativo lo pensa di tipo i e pertanto lo score viene aggiornato di conseguenza.

6.3.3 Algoritmi

: `AneVersusAllTrain(`$D^{multiclass}$`, BinaryTrain())`

$f_{ij} = \emptyset, \forall 1 \leq i < j \leq K$
for $i = 1$ **to** $K - 1$ **do**
  $D^{pos} =$ all $x \in D^{multiclass}$ labeled i
  for $j = j + 1$ **to** K **do**
    $D^{neg} =$ all $x \in D^{multiclass}$ labeled j
    $D^{bin} = \{(x, +1) : x \in D^{pos}\} \cup \{(x, -1) : x \in D^{neg}\}$
    $f_{ij} =$ `BinaryTrain(`D^{bin}`)`
Return all f_{ij}

: `AllVersusAllTest(`*all* f_{ij}, $\hat{x}$`)`

score $= \langle 0, \ldots, 0 \rangle$ %`Inizializza` K `score a` 0
for $i = 1$ **to** $K - 1$ **do**
  for $j = i + 1$ **to** K **do**
    y $= f_{ij}(\hat{x})$
    $score_i = score_i + y$
    $score_j = score_j - y$
Return `max(`*score*`)`

6.4 Confronto tra *OVA* e *AVA*

6.4.1 Tempo di training

AVA impara più classificatori ma il training set è molto più piccolo pertanto tende ad essere più veloce se le label sono equamente bilanciate.

6.4.2 Tempo di test

Avendo *AVA* più classificatori è tipicamente più lenta.

6.4.3 Errori

AVA fa training con data sets più bilanciata, ma avendo più classificatori i test tendono ad avere più possiblità di errori.

6.5 Riassunto

Se vengono usati classificatori binari viene tipicamente utilizzata *OVA*, altrimenti si usa un classificatore che permette label multiple come *DT* o *K-NN*, nonostante altri metodi più sofisticati siano meglio.

6.6 Multiclass evaluation

Per computare l'accuratezza nella predizione di una classe c si può usare la misura di predizione $Pr\frac{TP}{TP+FP}$ dove TP sono le predizioni corrette e FP le predizioni scorrette.

6.6.1 Microaveraging

Nel microaveraging si fa la media sugli esempi. Considerando n classi si calcola come:

$$Pr = \frac{\sum_{i=1}^{n} TP_{c_i}}{\sum_{i=1}^{n} (TP_{c_i} + FP_{c_i})}$$

6.6.2 Macroaveraging

Nel macroaveraging si calcola lo score di valutazione o accuratezza per ogni label e poi si fa la media tra le label. Questo in quanto dà più enfasi a label più rare e permette un'altra dimensione di analisi.

$$Pr = \frac{\sum_{c \in C} Pr_c}{|C|}$$

6.6.3 Confusion matrix

La confusion matrix è una matrice in cui (i, j) rappresenta il numero di esempi con label i predetti avere label j. Viene spesso espressa come percentuale. Nel caso di una classificazione a k classi sarà di dimensione $k \times k$. La performance è buona nel caso in cui la diagonale presenti le percentuali più alte.

Capitolo 7

Ranking

7.1 Classificazione multiclasse e multilabel

Nella classificazione multi classe ogni esempio ha esattamente una label che sono pertanto mutualmente esclusive. Nella classificazione multi label ogni esempio ha zero o più labels, dette anche annotazioni. Per svolgere una classficazione multi label basterebbe fare training su un modello per ogni label e applicarli a ogni nuovo esempio, ma ci sono altri metodi più sofisticati ed efficaci come il joint learning.

7.2 Problema del ranking

I dati di training sono divisi in K categoria ognuna delle quali corrispondente a un ranking. Si deve insegnare a un modello che quando riceve un insieme di esempi li fitti nel ranking corretto o ritorni un ordinamento per questo nuovo esempio.

7.2.1 Preference function

La preference function o binary classifier è un'implementazione di ranking. Data una query q e due campioni x_i e x_j il classificatore predice se x_i deve essere preferito a x_j rispetto alla query q. Il classificatore prende pertanto due campioni e dà in output 1 se il primo è più alto o -1 se è più basso. In questo modo si ottiene una funzione di ordinamento atomica che si può estendere a diversi esempi.

7.2.1.1 Perceptron

Per implementare questo algoritmo si può utilizzare il perceptron creando il vettore delle feature combinate dei due esempi:

$$f'_i = a_i - b_i$$

$$f'_i = \{1 \; if \; a_i > b_i, \; 0 \; altrimenti\}$$

Questa funzione viene sviluppata in maniera dipendente dall'applicazione. Un modo per computare il ranking numerico per i campioni può essere risolto utilizzando la preference function sommando i valori di ritorno di ogni classificatore e ottenendo uno score per ognuno di essi.

7.3 Utilizzo del ranking e della preference function

Con gli algoritmi visti precedentemente si potrebbe pesare il ranking di un esempio utilizzando la distanza dagli altri. SI possono usare diversi metodi di distanza dati che sono consistenti. La distanza a tempo di testing è calcolata come la confidenza della predizione del perceptron e dà un ordinamento degli esempi. In questo modo si ottiene una forma di ranking più precisa rispetto a quella vista prima. Per un algoritmo di ranking sofisticato si devono incorporare queste osservazioni a tempo di training: se un problema ritorna un'alta differenza in preferenza tra due esempi dovrebbe avere un peso più alto.

7.4 Ordinamento e ω-ranking

Nonostante la veloce soluzione dell'ordinamento contro la funzione di preference si può utilizzarla come funzione di sorting. Si definisce un ranking come una funzione σ che mappa gli oggetti alla posizione nella lista di ranking desiderata $(1, \ldots, M)$. Se $\sigma_u < \sigma_v$ allora u è preferito a v. Dati i dati con ranking osservati σ l'obiettivo è di imparare a predire i ranking per nuovi oggetti σ^*. Si definisce $\sum_M$ l'insieme di tutti gli ordinamenti di ranking in M. Si vuole modellare il fatto che uno sbaglio su alcune coppie è peggiore rispetto ad altre, pertanto si implementa una nuova funzione di errore per questo scopo. Si definisce una funzione di costo ω dove $\omega(i, j)$ è il costo di mettere qualcosa nella posizione j quando dovrebbe essere in i. Tale funzione deve essere:

- Simmetrica: $w(i, j) = w(j, i.$
- Monotona: $i < j < l \vee i > j > k \Rightarrow \omega(i, j) \leq \omega(i, k)$.
- Soddisfi la disuguaglianza triangolare: $\omega(i, j) + \omega(j, k) \leq \omega(i, k)$.

Un esempio potrebbe essere:

$$\omega(i, j) = \{1 \; if\{i, j\} \leq K \wedge i \neq j \;\; 0 \; altrimenti\}$$

Questa viene detta task di ω ranking. Dati:

- Uno spazio di input $\mathcal{X}$.
- Una distribuzione sconosciuta $\mathcal{D}$ su $\mathcal{X} \times \sum_M$.
- Un training set D campionato da $\mathcal{D}$.

Si deve computare una funzione $f : \mathcal{X} \to \sum_M$ che minimizzi:

$$\mathbb{E}_{(\mathcal{X},\sigma)\sim\mathcal{D}}\left[\sum_{u \neq v}[\sigma_u < \sigma_v][\hat{\sigma}_v < \hat{\sigma}_u]\omega(\sigma_u, \sigma_v)\right]$$

A tempo di testing invece di predirre score e poi ordinare la lista come negli algoritmi prima si utilizza un algoritmo di ordinamento utilizzando la funzione imparata come funzione di ordinamento. La differenza è che qua la funzione di comparazione è probabilistica.

Capitolo 8

Gradient descent

8.1 Model based machine learning

Nel model based machine learning si sceglie un modello definito da un insieme di parametri. In particolare si nota come:

- Per i decision trees i parametri sono la struttura dell'albero, quali features ogni nodo divide e le predizioni delle foglie.
- Per il perceptron i parametri sono i pesi e il valore di b.

Dopo aver scelto il modello si deve scegliere un criterio da ottimizzare o la funzione obiettivo come per esempio il training error. Infine si sviluppa un algoritmo di learning che deve cercare di minimizzare il criterio, spesso in maniera euristica.

8.1.1 Modelli lineari

Nei modelli lineari il modello è:

$$0 = b + \sum_{j=1}^{m} w_j f_j$$

Si deve scegliere il criterio da ottimizzare.

8.1.1.1 Notazioni

8.1.1.1.1 Funzione indicatrice Una funzione indicatrice trasforma valori di *Vero* e *Falso* in numeri e conte.

$$1[x] = \begin{cases} 1 & \textbf{if } x = True \\ 0 & \textbf{if } x = False \end{cases}$$

8.1.1.1.2 Dot-product Utilizzando una notazione vettoriale si rappresenta un esempio $f_1, \ldots, f_m$ come un vettore singolo $\overrightarrow{x}$ in cui j indicizza la feature e i indicizza un dataset di esempi. Si possono rappresentare anche i pesi $w_1, \ldots, w_m$ come un vettore $\overrightarrow{w}$. Il dot-product tra due vettori a e b viene definito come:

$$a \cdot b = \sum_{j=1}^{m} a_j b_j$$

8.1.1.2 Funzione obiettivo

Il criterio da ottimizzare o funzione obiettivo può essere:

$$\sum_{i=1}^{n} 1[y_i(w \cdot x_i + b) \leq 0]$$

Si devono pertanto trovare w e b tali che minimizzano questa funzione, ovvero:

$$argmin_{w,b} \sum_{i=1}^{n} 1[y_i(w \cdot x_i + b) \leq 0]$$

8.2 Loss functions

8.2.1 Loss $0/1$

Una funzione di loss $0/1$ è una funzione nella forma:

$$\sum_{i=1}^{n} 1[y_i * w \cdot x_i + b) \leq 0]$$

Dove tra le quadre si trova se la predizione e la label sono d'accordo, con vero se non lo fanno e tra le tonde la distanza dall'iperpiano, di cui il segno è la predizione. Questa funzione ritorna il numero di sbagli.

8.2.1.1 Minimizzare la loss $0/1$

Per minimizzare una funzione $0/1$ si deve, ogni volta cambiare un valore di w in modo che l'esempio è corretto o scorretto la perdita aumenta o diminuisce. Si nota come a ogni feature aggiunta si aggiunge una nuova dimensione allo spazio. Il minimo si trova trovando w e b che minimizzano la perdita. Questo è un problema *NP-hard*. Sue difficoltà comprendono il fatto che piccoli cambi in ogni w possono portare a grandi cambi nella perdita in quanto il cambio non è continuo. Ci possono essere molti minimi locali. Ad ogni punto non si hanno informazioni che direzionano verso il minimo. Pertanto si nota come una loss function ideale sia continua e differenziabile in modo da avere un'indicazione verso la direzione di minimizzazione e un unico minimo.

8.2.1.1.1 Loss function ideale Una loss function ideale dovrebbe essere:

- Continua in modo da ottenere informazioni riguardo la direzione della minimizzazione.
- Avere un solo minimo.
- Misurare la distanza tra la predizione reale e quella predetta.

8.2.2 Funzioni convesse

In una funzione convessa il segmento tra qualsiasi due punti della funzione si trova al di sopra della funzione.

8.2.3 Surrogate loss function

Per molte applicazioni si vuole minimizzare la loss 0/1. Una surrogate loss function è una loss function che fornisce un limite superiore alla loss function attuale. Si vuole identificare un surrogato convesso della loss function in modo da facilitarne la minimizzazione. Chiave a una loss function è come verifica la differenza tra la label y effettiva e la predizione y'.

8.2.3.1 Alcune surrogate loss function

- 01 *loss*: $l(y, y') = 1[yy' \leq 0]$.
- *Hinge* $l(y, y') = \max(0, 1 - yy')$.
- Exponential: $l(y, y') = \exp(-yy')$
- *Squared loss*: $l(y, y') = (y - y')^2$.

8.3 Gradient descent

Il gradient descent è un modo per trovare il minimo di una funzione: le derivate parziali danno un slope o direzione dove muoversi in tale dimensione. Questo approccio consiste di scegliere un punto di partenza e a ripetizione di: scegliere una dimensione e muoversi di una piccola quantità verso il minimo utilizzando la derivata. Pertanto si:

- Sceglie un punto di inizio w.
- Sceglie una dimensione.
- si muove di una piccola quantità verso la diminuzione della loss utilizzando la derivata.

Questo ciclo si ripete fino a che la loss non diminuisce in nessuna dimensione.

8.3.1 Spostamento in direzione della minimizzazione dell'errore

Il movimento in direzione della minimizzazione dell'errore è pertanto:

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

Dove η è il learning rate.

8.3.1.1 Calcolo dello spostamento per la loss function esponenziale

Si deve pertanto calcolare:

$$\begin{aligned} \frac{d}{dw_j} loss &= \frac{d}{dw_j} \sum_{i=1}^{n} \exp(-y_i(w \cdot x_i + b)) \\ &= = \sum_{i=1}^{n} \frac{d}{dw_j} [-y_i(w \cdot x_i + b)] \exp(-y_i(w \cdot x_i + b)) \end{aligned}$$

Si consideri pertanto ora:

$$\begin{aligned}\frac{d}{dw_j}[-y_i(w \cdot x_i + b)] &= -\frac{d}{dw_j}[-y_i(w \cdot x_i + b)] = \\ &= -\frac{d}{dw_j} y_i(w_1 x_{i1} + \cdots + w_m x_{im} + b) = \\ &= -y_i x_{ji}\end{aligned}$$

Si nota pertanto come:

$$\frac{d}{dw_j} loss = \sum_{i=1}^{n} -y_i x_{ij} \exp(-y_i(w \cdot x_i + b))$$

Si aggiorna pertanto w_j:

$$w_j = w_j - \eta \sum_{i=1}^{n} -y_i x_{ij} \exp(-y_i(w \cdot x_i + b))$$

Questo viene fatto per ogni esempio x_i.

8.3.2 Learning algorithm del perceptron

Si nota pertanto come considerando il perceptron nell'ambito del gradient descent si aggiorna sempre il vettore dei pesi. Si noti come in questo caso η rappresenta il learning rate, y_i la label e $(w \cdot x_i + b))$

```
: Perceptron()
repeat
    foreach training example (f_1, f_2, ..., f_n, label) do
        %Label = ±1
        prediction = b + Σ_{i=1}^{n} w_i f_i
        foreach w_i do
            w_j = w_j + η y_i x_ij exp(−y_i(w · x_i + b))  b = b + label
until Convergence
```

la predizione. Questi generano una costante c

8.3.3 Costante c

Nella costante c se label e predizione hanno lo stesso segno quando gli elementi predetti aumentano gli aggiornamenti diventano minori. Se invece sono diversi più diversi lo sono, maggiore l'aggiornamento.

8.3.4 Gradiente

Il gradiente è il vettore delle derivate parziali rispetto a tutte le coordinate dei pesi:

$$\nabla L = \left[\frac{\delta L}{\delta w_1} \cdots \frac{\delta L}{\delta w_N}\right]$$

: `GradientDescent(`$\mathcal{F}$, K, η_1, ...`)`

$z^{(0)} \rightarrow < 0, 0, \ldots, 0$%`Inizializza la variabile da ottimizzare`
for $k = 1$ **to** K **do**
  $g^{(k)} \rightarrow \nabla_z \mathcal{F}|_{z^{(k-1)}}$%`Computa il gradiente nella posizione corrente`
  $z^{(k)} \rightarrow z^{(k-1)} - \eta^{(k)} g^{(k)}$%`scendi il gradiente`
return $z^{(k)}$

Ogni derivata parziale misura quanto veloce la perdita cambia in una direzione. Quando il gradiente è zero la perdita non sta cambiando in nessuna direzione.

Nei problemi in cui il problema di ottimizzazione è non convesso si trovano dei minimi locali, questi non permettono all'algoritmo di proseguire in quanto non distingue tra minimi locali e minimi globali. Un altro punto è un punto a sella, in cui certe direzioni curvano verso l'alto e altre verso il basso. In tali punti il gradiente è 0 e l'algoritmo si blocca. Un modo per uscire da un punto à sella è spostarsi a lato un po' in modo da uscirne. Si nota come i punti a sella sono molti comuni in alte dimensioni. Il learning rate è molto importante in quanto permette di decidere la distanza coperta da uno spostamento determinando velocità di avvicinamento al minimo e precisione dell'algoritmo.

Capitolo 9

Regularization

9.1 Introduzione

Si noti come con il gradient descent si calcola il valore minimo della loss function sul training set. Ci si deve pertanto preoccupare del overfitting. Il minimo w e b sul training set infatti non sono tipicamente il minimo per il test set. Questo problema viene risolto attraverso la regolarizzazione.

9.2 Regolarizzatori

I regolarizzatori sono criteri addizionali alla loss function in modo da evitare overfitting. Prova a mantenere i parametri regolari o normali. È un bias sul modello che forza che il learning preferisca certi tipi di pesi rispetto agli altri.

$$argmin_{w,b} \sum_{i=1}^{n} loss(yy') + \lambda regularizer(w,b)$$

Tipicamente non si vogliono pesi molto grandi in quanto un piccolo cambio in una feature potrebbe causare un cambio nella predizione. Si potrebbe anche preferire pesi di 0 per features che non sono utili.

9.2.1 Regolarizzatori comuni

9.2.1.1 Somma dei pesi

Il regolarizzatore somma dei pesi penalizza di più piccoli valori e si calcola come:

$$r(w,b) = \sum_{w_j} |w_j|$$

Si dice anche 1-norm.

9.2.1.2 Somma quadratica dei pesi

La somma quadratica dei pesi penalizza di più valori grandi e si calcola come:

$$r(w,b) = \sqrt{\sum_{w_j} |w_j|^2}$$

Si dice anche 2-norm.

9.2.1.3 P-norm

Si intende per p-norm:

$$r(w,b) = \sqrt[p]{\sum_{w_j} |w_j|^p} = ||w||^p$$

Valori più piccoli di $p < 2$ incoraggiano vettori più sparsi, mentre valori più grandi scoraggiano pesi più grandi creando pertanto vettori con pesi più simili.

9.3 Gradient descent e regolarizzazione

Si nota come se scelto un modello e dimostrato che $loss + regularizer$ è una funzione convessa si può ancora utilizzare il gradient descent. Si nota come per costruzione le p-norms sono convesse per $p \geq 1$ e pertanto:

$$\frac{d}{dw_j} objective = \frac{d}{dw_j} \sum_{i=1}^{n} \exp(-y_i(w \cdot x_i + b)) + \frac{\lambda}{2}||w||^2 =$$

$$\vdots$$

$$= -\sum_{i=1}^{n} y_i x_{ij} \exp(-y_i(w \cdot x_i + b)) + \lambda w_j$$

Pertanto con il regolarizzatore l'aggornamento è

$$w_j = w_j + \eta y_i x_{ij} \exp(-y_i(w \cdot x_i + b)) - \eta\lambda w_j$$

Si noti pertanto come la regolarizzazione, se w_i è positiva la riduce, mentre se è negativa lo aumenta muovendo w_i verso lo 0.

9.4 Regolarizzazione con le p-norms

9.4.1 L1

$$w_j = w_j + \eta(lossCorrection - \lambda sign(w_j))$$

Popolare in quanto tende a dare soluzioni sparse, ma non è differenziabile e lavora unicamente per risolutori a gradient descent.

9.4.2 L2

$$w_j = w_j + \eta(lossCorrection - \lambda w_j)$$

Popolare in quanto per qualche loss function può essere risolta direttamente.

9.4.3 Lp

$$w_j = w_j + \eta(lossCorrection - \lambda c w_j^{p-1})$$

Meno popolare in quanto non riduce i pesi abbastanza.

9.5 Metodi di machine learning con regolarizzazione

- Ordinario: least squares: squared loss.
- Ridge regression: squared loss with L2 regularization
- Lasso regression: squared loss with L1 regularization
- Elastic regression: squared loss with L2 and L1 regularization
- Logistic regression: logistic loss.

Capitolo 10

Support vector machines

10.1 Introduzione

Le support vector machines permettono di trovare l'iperpiano ottimo che separa i training data. Si nota come fino ad ora c'era una grande variabilità nell'iperpiano che il classificatore lineare trova cominciando da diversi punti nell'iperspazio. Inoltre quando i dati non sono linearmente separabili questa variabilità aumenta grandemente.

10.1.1 Considerazioni su perceptron e gradient descent

Come visto il perceptron se i dati sono linearmente separabili trova un qualche iperpiano che li separa, altrimenti continuerà ad aggiustarsi iterando attraverso gli esempi e l'iperpiano dipenderà dall'ultimo esempio visto. Il gradient descent invece trova l'iperpiano che minimizza $loss + regularization$ in entrambi i casi.

10.1.2 Idea delle support vector machines

Le support vector machines cercano di trovare il migliore iperpiano che lo fa. Per definirlo vengono introdotti i margini di un iperpiano.

10.2 Margini

I margini di un iperpiano sono la distanza dal punto più vicino. Maggiore il margine meglio l'iperpiano separa le classi. Il fatto che il modello trovato da una SVM ha il maggior margine possibile aumenta l'abilità di generalizzazioen del modello.

10.2.1 Support vectors

I support vectors sono i data points più vicini al margine. Per n dimensioni ci saranno almeno $n+1$ support vectors.

10.2.2 Calcolare il margine

Il margine può essere pertanto calcolato come la distanza dal support vector. La distanza di un punto x da un'iperpiano viene calcolata come $d(x) = \frac{wx+b}{||w||}$. Il margine viene calcolato pertanto come $\frac{c}{||w||}$, dove c è la traslazione dell'iperpiano in modo che la sua distanza dal support vector sia 0. Si nota come scalando il vettore dei pesi w la distanza dall'iperpiano rimane la stessa. Inoltre essendo c e w strettamente correlati si può assumere $c = 1$ sempre.

10.3 Problema di ottimizzazione

Si vuole pertanto massimizzare il margine ma classificando correttamente ogni esempio.

$$argmax(margin(w, b)) \qquad \wedge \qquad y_i(wx_i + b) \geq 1 \forall i$$

L'errore viene tenuto basso dal fatto di avere tutti i data point al di fuori dal margine come stabilito dalla seconda equazione.

10.3.1 Massimizzare il margine

Per massimizzare il margine si vuole massimizzare

$$\frac{1}{||w||}$$

Tenendo d'occhio l'errore. Si vuole pertanto avere il vettore di pesi minore possibile w. $c = 1$ è un'assunzione che si fa in quanto altrimenti si starebbe imparando una versione scalata dello stesso problema (spiegazione nelle slides). In realtà si tenta di minimizzare:

$$||w||^2 = \sqrt{\sum_{j \in I} |w_j|^2}$$

Soggetto a $y_i(wx_i + b) \geq 1$. Questo è lo stesso problema con il vantaggio che è una funzione convessa con lo stesso minimo della norma di w. In questo modo diventa un problema di ottimizzazione quadratica, di risoluzione semplice.

10.4 Soft margin classification

La soft margin classification permette di usare SVM quando i punti non sono linearmente separabili. Quando qualche dato non è lineramente separabile non si riescono a soddisfare i due costraint necessari per train le SVM e pertanto si necessita di modificare la funzione obiettivo. Per farlo si permette al modello di fare delle predizione sbagliate, ma si aggiunge alla funzione obiettivo una penalità per ogni esempio classificato erroneamente. Queste sono dette slack penalties e sono usate per ottenere un modello che accetta con una certa tolleranza errori di classificazione. La funzione da ottimizzare diventa pertanto:

$$||w||^2 + \mathcal{C} \sum_i \zeta_i \qquad subject\ to \qquad y_i(wx_i + b) \geq 1 - \zeta_i \forall i$$

Con il costraint $\zeta_i \geq 0 \forall i$. Si nota come ζ_i sono le slack variables e se ne trova una per ogni esempio nel dataset e sono utilizzate per correggere classificazioni sbagliate, ma poi il loro peso è aggiunto

alla funzione obiettivo. SI vuole pertanto trovare un trade-off tra minimizzare il quadrato della norma di w e il valore di penalità dato dalla slack variable. La somma di ζ_i misura la tolleranza agli errori. Si nota come:

- Piccoli valori di $\mathcal{C}$ permettono più errori che consistono in un margine più grande.
- Grandi valori di $\mathcal{C}$ danno agli errori di classificazione più peso restringendo il margine.
- $\mathcal{C} = \infty$ impone tutti i costraint e riduce a un hard margin.

I valori di ζ_i sono imparati insieme a w e b. SI nota come questo è ancora un problema di ottimizzazione quadratica con costraint lineari, ma il numero di calcoli con un training set medio grande è molto elevato.

10.4.1 Risolvere il problema delle SVM

Data la soluzione ottimale (w, b) si può calcolare la slack penalty per ogni punto. Per tutti gli esempi correttamente classificati al di fuori del margine il valore dovrà essere 0. Per gli esempi correttamente classificati all'interno del margine sarà la distanza dal punto e la linea del margine, che può essere calcolata come $1 - (\textit{valore della funzione di decision})$, un valore compreso tra 0 e 1, in particolare:

$$\zeta_i = 1 - y_i(wx_i + b)$$

I dati classificati con errore il valore è dato dalla somma della distanza dall'iperpiano più la distanza dal maargine, con la stessa formula come nel caso precedente. Si riassumne la formula in:

$$\zeta_i = \max(0, 1 - yy')$$

Che si nota che è la hinge loss function. Trasformando la funzione obiettivo di un SVM in una funzione di hinge loss si trasforma il problema in uncostrained in quanto entrami i costraint sono tenuti in considerazione dalla funzione. La nuova funzione obiettivo pertanto sarà:

$$\min_{w,b} ||w||^2 + \mathcal{C} \sum_i \max(0, 1 - y_i * wx_i + b))$$

Che può essere considerata come un problema di gradient descent con Hinge loss function e regolarizzatore $||w||^2$. La soluzione a questo problema trova pertanto l'iperpiano con il margine più grande possible permettendo un soft margin.

10.5 Data non lineramente separabile

Per separare i dati non linearmente separabili si possono utilizzare spazi con dimensioni maggiori. Prima si tentava di risolvere un problema di ottimizzazione quadratica soggetto a un'insieme di costraint lineari. Questo è il problema primario delle SVM. Si può riscrivere questo problema nella forma di un dual problem. Si noti come i problemi di ottimizzazione quadratica sono una classe ben conosciuta di problemi di programmazione per cui esistono diversi algoritmi non banali. Una possibile soluzione coinvolge costruire un dual problem dove un moltiplicatore di Lagrange a_i è associato con ogni costraint di ineguaglianza nel problema originario. Si deve pertanto trovare $a_1, \ldots, a_n$ tale che:

$$Q(a) = \sum a_i - \frac{1}{2} \sum \sum a_i a_j y_i y_j x_i^T x_j$$

è massimizzato e $\sum a_i y_i = 0$ e $a_i \geq 0 \forall a_i$. Il problema è pertanto massimizzare $Q(a)$, un problema di ottimizzazione quadratico con un paio di costraint lineari. Un buon risolutore scalabile è SM). Una volta risolto il dual problem e ottenuto tutti i valori di a_i posso computare w e b:

$$w = \sum a_i y_i x_i$$

$$b = y_k - \sum a_i y_i^T x_k$$

Per ogni $a_k > 0$. Pertanto la funzione classificatrice è:

$$f(x) = \sum (a_i y_i x_i^T x) + b$$

Si nota come tutte le a_i degli esempi che non sono support vector hanno valore 0, pertanto non si necessita di esplicitare w se si conosce a. In quanto x compare solo nel dot product nella funzione di predizione e di ottimizzazione e a è nulla per ogni esempio non support vector il loro impatto è nullo: una volta trainata la funzione di predizione può mantenere solo i support vector risparmiando memoria.

10.5.1 Soft margin classifier

Il dual problem è simile nel caso del soft margin classifier: si deve trovare $a_i, \ldots, a_n$ tali che:

- Massimizzano $Q(a) = \sum a_i - \frac{1}{2} \sum \sum a_i a_j y_i y_j x_i^T x_j$.
- $\sum a_i y_i = 0$
- $0 \leq a_i \leq \mathcal{C} \forall a_i$

Si nota come x_i con non zero a_i sono support vectors. Pertanto la predizione dipende solo dai support vectors. Con entrambi gli approcci si impara un iperpiano che separa i datapoints e in entrambi solo i data points rilevanti per la prediction function sono i support vector.

10.5.2 Identificare i support vector

I support vector sono identificati dagli algoritmi di ottimizzazione quadratica come i training point cono non zero lagrangian multipliers a_i in quanto i training point appaiono unicamente all'interno prodotti interni, pertanto non li necessito ma solo i loro dot-product.

10.5.3 Utilizzo di SVN in maniera non lineare

Se i dati non sono linearmente separabili si possono portare a uno spazio a dimensioni maggiori rendendoli lineramente separabili. Lo spazio delle features originali può essere sempre mappato a uno spazio di features con dimessone maggiore dove il training set è separabile. Il problema di ottimizzazione e la funzione di predizione dipendono solo dal prodotto dei vettori di features degli esempi. Il classificatore lineare dipende sul prodotto interno di questi vettori:

$$K_{linear}(x_i, x_j) = x_i^T x_j$$

Se ogni data point è mappato in uno spazio con maggiore dimensione attraverso una qualche trasformazeion$\phi : x \to \phi(x)$, il prodotto interno diventa:

$$K_{for\ function\ \phi}(x_i, x_J) = \phi(x_i)^T \phi(x_j)$$

10.5.3.1 Kernel function

Una kernel function è una funzione equivalente a un prodotto interno in uno spazio di features. In generale non si è interessati a ϕ ma alla funzione kernel in quanto mappa i dati a uno spazio a maggiore dimensione implicitamente. Ogni kernel ha un iperparametro a parte la lineare.

- Linear: $K(x_i, x_j) = x_i^T x_j$
- Polinomial of power p: $K(x_i, x_j) = (1 + x_i^T x_j)^p$.
- Gaussian o radial-basis: $K(x_i, x_j) = e^{\frac{|x_i = x_j|^2}{2\sigma^2}}$. In questo caso ogni punto è mappato a una funzione gaussiana e $\phi(x)$ ha infinite dimensioni. La combinazione delle funzioni per i support vectors è il separatore.

Lo spazio a maggiore dimensioni ha una idmensionalità d intrinseca ma il separatore lineare in esso corrisponde a un separatore non lineare nello spazio originale.

10.5.3.2 Soluzione del problema scelto il kernel

Una volta scelto il kernel il problema diventa trovare $a_1, \ldots, a_n$ tali che:

- Massimizzano $Q(a) = \sum a_i - \frac{1}{2} \sum \sum a_i a_j y_i y_j K(x_i, x_j)$.
- $\sum a_i y_i = 0$
- $a_i \geq 0 \forall a_i$

La soluzione è:

$$f(x) = \sum a_i y_i K(x_i, x) + b$$

La tecnica di ottimizzazione per a_i rimane la stessa. Si nota come risolvendo il dual problem con il kernel giusto si risolve il problema SVM portando i dati in uno spazio a dimensione maggiore in cui è linearmente separabile.

Capitolo 11

Neural Networks

11.1 Introduzione

È stato visto come il perceptron può imparare un modello lineare utilizzando la funzione attivatrice e il peso degli input. Funzioni attivatrici tradizionali sono non lineari come la sigmoide $h(x) = \frac{1}{1+e^{-x}}$ e la tangente iperbolica $h(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, mentre funzioni attivatrici più moderne sono la rectified linear unit *ReLU*: $h(x) = \max(0, x)$ e la Leaky ReLU $h(x) = \max(\alpha x, x)$ con α costante piccola.

11.2 Il perceptron multilayer

L'idea del multilayer perceptron è di connettere un grande numero di perceptron creando layers che sono densamente connessi tra di loro. Questa struttura presenta un input layer, hidden layers e un output layer. Questa composizione di perceptron causa una composizione di funzioni non lineari.

11.2.1 Feed forward NN

Si intende per feed forward NN una rete neurale in cui l'informazione fluisce dall'input all'output passando in un DAG. Si nota un singolo processamento tra I/) che avviene in ogni singolo neurone simile a come avviene in un perceptron normale. I nodi ora sono connessi e l'output di uno di essi è l'input degli altri. La NN può computare una o più funzioni non lineari componendo funzioni algebriche implementate nella forma di connessioni, pesi e bias dei layer hidden e output. Gli hidden layer sono una rappresentazione intermedia con diversi gradi di complessità.

11.2.1.1 Nodo

Un nodo riceve in input un insieme di output di altri nodi θ_n che vengono calcoalti in $Z = \sum_i \theta_i x_i + \theta 0$ e passati alla funzione soglia $h(Z)$. Considerando ora un layer si hanno multipli Z e i calcoli vanno replicati su più canali di output in cui ogni canale ha il proprio insieme di pesi e bias. Inoltre ogni output ha un insieme di pesi $\theta_{i,j}$ e il bias $\theta_{0,j}$.

11.2.1.2 Single layer NN

In una single layer NN ogni neurone in un layer n computa la sua attivazione utilizzando:

- L'attivazione di tutti i neuroni del layer $n-1$ con gli insiemi di pesi e bias.
- La funzione di threshold.

È comune usare nelle NN la stessa attivazione per tutti gli hidden layer ma diversa per gli output.

11.2.1.3 Training backpropagation

Si nota come non si può utilizzare il metodo di learning del perceptron in quanto gli hidden layer non hanno informazioni riguardo gli output desiderati e pertanto non sanno come modificare i propri pesi. Il problema viene risolto attraverso la backpropagation, un modo efficiente di computare gradienti per aggiornare i pesi in tutti i layer delle NN.

- Si fa forward propagation si sommano gli input, si produce l'attivazione, feed-forward e si ottiene l'output.
- Si stima l'errore comparando la predizione e le label reali.
- La backpropagation invia il segnale d'errore indietro all'intera architettura della NN per aggiornare i propri pesi.

Inviare l'errore all'indietro permette di aggiornare i pesi attraverso una formula di ottimizzazione o obiettivo simile a quelle già viste:

$$\sum_i L(y_i, f(x_i, \Theta))$$

Dove si prova a minimizzare l'errore minimizzando Θ che rappresenta i parametri della NN. Per fare backpropagation si deve computare il gradiente: dati il training set si vuole imparare tutti i pesi della rete per minimizzare la loss function. Si è liberi nella scelta della loss function. Si possono aggiornare i pesi con il gradient descent e la backpropagation è la tecnica che permette di computare efficientemente il gradiente per ogni nodo.

11.3 Second AI winter

Anche con la backpropagation le reti neurali hanno grandi problemi:

- Molti layer rendono le NN prone a fare overfitting.
- Vanishing gradient: il gradiente continua a diventare sempre più piccolo mentre si fa backpropagation rendendo impossibile aggiornare i pesi.

Inoltre non si trovavano dataset abbastanza grandi e l'hardware non era abbastanza potente. Questo ha causato una diffusione di SVM in quanto hanno accuratezza simile, possiedono molto meno euristiche e parametri e hanno una prova di generalizzazione. Nel 2006 grazie a un nuovo modo di inizializzare i pesi: si fa training di ogni layer attraverso unsupervised learning e si fa fine-tuning su di es attraverso un round di supervised learning. Una NN è pertanto una composizione di moduli in cui si imparano un insieme di pesi insieme che sono anche le features. I deep model sono i migliori in quanto si hanno molti dati e potenza di calcolo.

11.4 Feed Forward Networks

Una FFNN viene usata per approssimare una funzione $f^* : X \to Y$ dove f^* è il classificatore ottimale. Le FFNN devono definire una mappatura parametrica $y = f(x, \theta)$ in modo da ottenere una buona classificazione. Ogni layer viene rappresentato da una certa funzione e la composizione delle funzioni può essere rappresentata da un DAG. La profondità della rete è il numero di layer, mentre la larghezza di un layer è il numero di perceptron. In quanto viene implementata una funzione complessa non si può garantire sia convessa e pertanto che si arrivi alla convergenza. Per applicare il gradient descent si deve specificare un modello o architettura, una funzione di costo loss e la struttura del layer di output.

11.4.1 Funzione di costo

La funzione di costo deve calcolare la discrepanza tra la predizione e le labels reali. Le loss già viste vanno bene per le NN. Il meccanismo di classificazione più comune consiste nel convertire gli output della rete in probabilità di appartenere a una classe. La normalizzazione a una probabilità è ottenuta attraverso softmax: ogni classe ottiene un voto dal layer di output e con log normalization questi sono pesati e trasformati in probabilità. La loss function più comune con softmax è la cross-entropy loss function:

$$\mathcal{L} = -\sum_k y_k \log(S(l_k)) = -\log(S(l))$$

In cui si compara la label con il log di softmax. La combinazione di loss e output è basata sulla comparazione del vettore delle probabilità con il vettore di label per ogni campbione. La scelta della loss è correlata alla scelta dell'output layer. Un'altra opzione possibile è di applicare pesi e bieas all'output con un output layer lineare:

$$y = W^T f + b = \sum_j w_j f_j + b$$

Dove y è l'output, W il vettore dei pesi f il vettore di features e b il bias. Non satura e questo è un bene in quanto mantiene il gradiente lontano da 0. I linear output layer producono probabilità log non normalizzate. IN caso della classificazione la scelta di output tipica è softmax che trasforma l'output in probabilità normalizzate:

$$softmax(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Nel caso delle hidden unit queste prendono un input x, computano una trasformazione $z = W^T x + b$, applicano una funzione $h(z)$ non lineare e producono l'output. La scelta di $h()$ è artigianale e la più utilizzata è la ReLU:

$$h(z) = \max(0, z)$$

IL suo gradiente è 0 o 1, ottimo per l'ottimizzazione in quanto il calcolo è molto semplice e annulla il problema dei gradienti piccoli. Non è derivabile in 0 pertanto si sceglie a caso. Diverse variazioni di ReLU risolvono il problmea che quando il gradiente diventa 0 il neurone muore, ma hanno il problema di saturazione.

11.4.2 L'architettura

Scegliere l'architettura consiste nel scegliere il numero e le dimensioni dei layer. La maggior parte delle volte si segue un'euristica, ma si trova un teorema: una rete a 2 layer con linear output con squashing (riduce l'output a un range ristretto) non lineare nelle hidden unit può approssimare ogni funzione continua su dominio compatto a una ccuratezza arvitraria. Questo implica che a prescindere dalla funzione che si vuole imparare si sa che un large MLP rappresenterà questa funzione, ma non garantisce che l'algoritmo sarà in grado di impararla in quanto deve gestire ottimizzazione e overfitting. Gli esperimenti notano come è meglio creare reti profonde rispetto a larghe.

11.4.3 Backpropagation

La backpropagation avviene dopo aver ottenuto l'output da un input x e aver computato l'errore e un costo scalare dipendenti dalla loss function. Si usa il costo per computare un gradiente della perdita con rispetto ai pesi e si aggiornano con il gradient descent per ogni layer attraverso backpropagation. IN modo da usare il gradient descent si devono computare tutte le derivate degli errori per tutti i pesi nella rete. Dal training data non si sa cosa le unità nascoste dovrebbero fare ma si può computare quanto veloce l'errore cambia cambiando un'attività nascosta. Per farlo si calcolano le derivate degli errori con rispetto alle attività nascosta. Ogni unità nascosta può avere effetto su molte unità di output e avere effetti separati sull'errore. Si dovrebbero combinare questi effetti. Si possono computare le derivate degli errori efficientemente con una regola ricorsiva e quando si hanno le derivate degli errori per queste attività nascoste è facile avere le derivate degli errori per i pesi che vanno all'interno. Calcolo dell'output:

$$\hat{y}(x; w) = f\left(\sum_{j=1}^{m} w_j^{(2)} h\left(\sum_{i=1}^{d} w_{ij}^{(1)} x_i + w_{0j}\right) + w_0^{(2)}\right)$$

Calcolo dell'errore della rete su un training set:

$$L(X; w) = \sum_{i=1}^{N} \frac{1}{2}(y_i - \hat{y}(x_i; w))^2$$

Qui si comparano le predizioni con la label reali attraverso la squared loss. Per utilizzare il gradient descent si deve computare la derivata della loss con rispetto ai pesi per ogni nodo. Per calcolare la derivata della loss dell'output layer in caso è lineare:

$$\frac{dL(x_i)}{dw_j} = (\hat{y}_i - y_i)x_{ij}$$

Nel terzo passaggio si vuole computare il gradient descent per tutte le unità nascoste della rete. Si deve computare pertanto la derivata della perdita con rispetto ai pesi. Si consideri $t \in T$ il livello corrente, $s \in S$ quello seguente e $j \in J$ quello precedente. L'obettivo è la derivate della loss function con rispetto ai pesi che arrivano dall'unità precedente in un'unità t del livello corrente

$$\frac{dL}{dw_{jt}}$$

Si devinisce l'attivazione a_t l'input della funzione di attivazione di un'unità di livello t, la somma di $w_{jt} \times z_j$ dove z_j è l'output della funzione di attivazione della j-esemia unità:

$$a_t = \sum_j w_{jt} z_j$$

Si nota come al loss L dipende da w_{jt} solo attraverso l'attivazione a_t, pertanto si manipola la formula in modo da computare la derivata della loss con rispetto di a_t:

$$\frac{dL}{dw_{jt}} = \frac{dL}{da_t}\frac{da_t}{dw_{jt}}$$

Si nota come $\frac{da_t}{dw_{jt}} = z_j$, pertanto:

$$\frac{dL}{dw_{jt}} = \frac{dL}{da_t}z_j$$

Il termine a destra viene detto errore locale ed è quello computato, mentre la prima frazione del termine destro viene detta δ_t:

$$\frac{dK}{dw_jt} = \delta_t z_j$$

Si ha ora una relazione ricorsiva che permette di calcolare δ_t dato un livello successivo S e tutte le sue unità nascoste s è:

$$\delta_t = \sum_{s\in S}\frac{dL}{da_s}\frac{da_s}{da_t}$$

Si nota come il primo termine nella somma è δ_s, il termine ricorsivo. Nella seconda invece a_s è l'input della funzione attivatrice di $s \in S$, ovvero:

$$a_s = \sum_{t:t\to s} w_{ts}h(a_t)$$

w_{ts} è il peso del link di un nodo t di T a un nodo s di S. Pertanto se si vuole computare l'attivazione di una unità specifica s da S si deve moltiplicare l'output di attivazione $h(a_t) = z_t$ volte il peso del link da t a s per tutte le unità di T. In quanto si ha $\frac{da_s}{da_t}$, l'operatore di derivazione ha effetto solo su $h(a_t)$ in quanto i pesi sono costanti in rispetto di a_t, pertanto:

$$\frac{da_s}{da_t} = h'(a_t)\sum_{t:t\to s} w_{ts}$$

Si unisce questo nel modello ricorsivo e si ottiene la formula:

$$\delta_t = \sum_{s\in S}\frac{dL}{da_s}\frac{da_s}{da_t} = h'(a_t)\sum_{t':t'\to s} w_{t's}\delta_s$$

In questa formula t' è ogni nodo di T connesso con il nodo s. Ora si ha una formula che permette di computare il gradiente per tutte le unità nascoste di tuti i layer. Si necessita pertanto ora di un caso migliore. In quanto questo inizia al layer di output si deve calcolare δ a tale layer. Quando si ha una funzione di output lineare il termine è la differenza tra la predizione e la label corretta:

$$\delta_{out} = \hat{y} - y$$

Ora con questi elementi si computa il gradiente e si applica il gradient descent. Si hanno due regole per aggiornare i pesi, una per il layer di output e una per i layer nascosti. Quando si calcola δ_j la derivata della loss function è importante in quanto se arriva troppo vicina a 0 si arriva a un problema di saturazione fermando l'aggiornamento dei pesi. In caso al rete abbia più di un output si ripete il processo per ogni output.

11.4.4 Scelta di un ottimizzatore

Il gradient descent trova l'insieme di parametri che rendono la loss il più piccola possibile e i cambi di parametri dipendono sul gradiente della loss con rispetto dei pesi della rete. Si analizzano altri miglioramenti come Stochastic gradient descent. Il gradient descent Nelle reti neurali può essere calcolato in diversi modi.

11.4.4.1 Batch Gradient Descent (BGD)

In BGD i gradienti sono computati su ogni update per l'intero training con un alto costo computazionale ma garantendo una grande stabilità nella stima del gradiente. Il learning rate ε_k può cambiare nel tempo.

: `Batch Gradient Descent at iteration K`

return: Learning rate ε_k
return: Initial Parameter θ
while *stopping criteria not met* **do**
  %`Compute gradient estimate over` N `examples`
  g $= +\frac{1}{N}\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)}$
  %`Apply update`
  $\theta = \theta - \varepsilon_k g$

11.4.4.2 Stochastic gradient descent (SDG)

In SDG si computa il gradiente solo su un campione e non sull'intero training set in modo da ottenere performance migliori.

: `Stochastic Gradient Descent at iteration K`

return: Learning rate ε_k
return: Initial Parameter θ
while *stopping criteria not met* **do**
  %`Compute gradient estimate over sample example (`$x^{(i)}, y^{(i)}$`) from training set`
  g $= +\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)}$
  %`Apply update`
  $\theta = \theta - \varepsilon_k g$

11.4.4.3 Minibatches

Le minibatches risvono il problema di SDG rispetto ai dati rumorosi e permettono una parallelizzazione rispetto alle minibatches. Sono tipicamente di dimensione 2^n per le proprietà di calcolo della GPU.

11.4.4.4 Momento

Un problema di BGD e SGD è il fatto che minimizzano l'errore in molto tempo. Una soluzione diversa è data dalla tecnica del momento, che introduce un vettore velocità v di aggiornamenti, una media con decay esponenziale per i gradienti utilizzato per aggiornare i pesi. Introduce un vettore momento che regola il trade-off tra il gradiente allo step corrente e quelle vecchie.

: `Stochastic Gradient Descent with momentum`

return: Learning rate ε_k
return: Momentum parameter α
return: Initial Parameter θ
return: Initial velocity v
while *stopping criteria not met* **do**
  %`Compute gradient estimate over sample example` $(x^{(i)}, y^{(i)})$ `from training set`
  g $= +\nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)}$
  %`Compute velocity update`
  $v = \alpha v - \varepsilon_k g$
  %`Apply update`
  $\theta = \theta - \varepsilon_k g$

11.4.4.5 Adaptive learning rate method

Alcune volte è bene usare un learning rate diverso per ogni peso. Un metodo che lo implementa è Adagrad o adaptive gradient optimizer. Questo fa downscale un parametro del modello della radice della somma dei quadrati dei valori storici, in questo modo parametri con una grande derivata parziale hanno learning rate che si abbassano rapidametne. Si adatta al learning rate dei parametri svolgendo aggiornamenti minori associati con feature che più frequenti e grandi per feature meno frequenti. Utile per gestire dati sparsi.

: `AdaGrad`

return: Learning rate ε_k
return: Initial Parameter θ, δ
r $= 0$
while *stopping criteria not met* **do**
  %`Compute gradient estimate over sample example` $(x^{(i)}, y^{(i)})$ `from training set`
  g $= +\nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)}$
  %`Compute velocity update`
  r $=$ r $+ f \circ g$ %`Compute update`
  $\Delta\theta = -\frac{\varepsilon_k}{\delta + \sqrt{r}} \circ g$
  $\theta = \theta + \Delta\theta$

Capitolo 12

Reinforcement learning

Capitolo 13

Unsupervised Learning

Capitolo 14

Generative Models