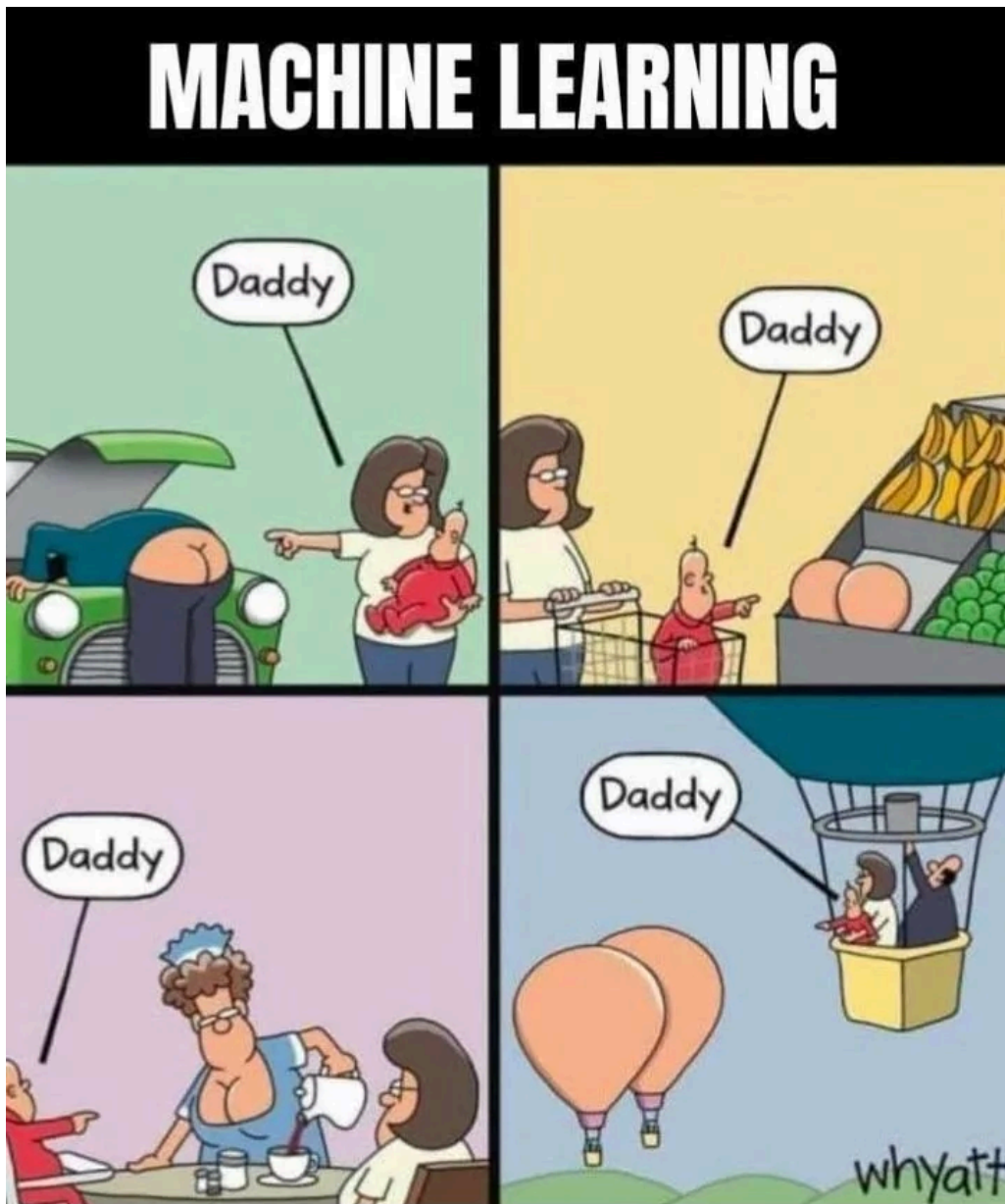


MACHINE LEARNING

Author: Simone Festa, August 2024



SUPERVISED LEARNING

Questa modalità fa uso di *training set* e *testing set*. L'input (che può avere d dimensioni) viene associato a delle *label* (etichette), e per praticità è rappresentato mediante un *vettore* $\langle \text{input}, \text{etichetta} \rangle$. Parliamo di *regressione* se l'etichetta t è un valore reale, o di *classificazione* se t appartiene a insieme discreto.

Nella **classificazione**, un algoritmo è **K-nearest neighbors algorithm**, in cui, dato un nuovo punto da classificare, si prende l'etichetta del punto noto più vicino. Si usa la *distanza euclidea*, trovando quella minima usando la seguente formula. Notiamo che facciamo la distanza per ogni dimensione.

$$\|x^{(a)} - x^{(b)}\|_2 = \sqrt{\sum_{j=1}^d (x_j^{(a)} - x_j^{(b)})^2}$$

Per irrobustire il tutto, visto che potremmo avere come punto più vicino uno classificato male, usiamo K vicini. Se K è piccolo, campioniamo a grana fine, ma troppa precisione vuol dire **overfitting** (quando mi danno i dati veri non sono più così preciso). Se K è grande, sono più stabile, ma posso avere **underfitting**, ovvero semplifico troppo ciò che trovo. K è iperparametro (cioè non fa parte dell'apprendimento), normalmente si prende $k < \sqrt{n}$. Si può provare la configurazione migliore usando *training set*, **validation set** (per vedere come si comporta), e solo dopo aver trovato la configurazione migliore si prova su *testing*.

E' importante *normalizzare* i dati, $\tilde{x}_j = \frac{x_j - \mu_j}{\sigma_j}$. Tutto il tempo è dal testing time, perché calcolo per tutti gli N punti la distanza in D dimensioni, cioè $O(ND)$, più ordinamento punti, $O(N \log N)$.

REGRESSIONE LINEARE

Nella regressione lineare usiamo una funzione lineare tra feature x (cioè gli input) e predizione t , e cerchiamo una funzione y che si avvicini a t . Notiamo il vettore di pesi w e il bias b . Cerchiamo la linea

$$y = f(x) = \sum_j w_j x_j + b$$

migliore prodotta da (w, b) .

Se passiamo in forma **matrice**, e mettiamo una colonna di 1 su x , e una b sui pesi w , cioè:

$$x = \begin{bmatrix} 1 & [x^{(1)}]^T \\ \vdots & \vdots \\ 1 & [x^{(N)}]^T \end{bmatrix} \in \mathbb{R}^{N \times (D+1)} \text{ and } w = \begin{bmatrix} b \\ w_1 \\ \vdots \\ w_N \end{bmatrix} \in \mathbb{R}^{D+1}$$

Il tutto diventa $y = Xw$

Per capire quanto è grave l'errore, si usa la **squared error loss function**, ovvero: $\mathcal{L}(y, t) = \frac{1}{2}(y - t)^2$

la quale amplifica l'errore grande. La derivata è inoltre semplice, e la funzione è **convessa**!

Se facciamo la **media** di tutte le funzioni Loss, otteniamo la **funzione costo**, parliamo di **Mean Squared Error MSE**:

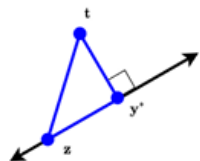
$$J(w, b) = \frac{1}{2N} \sum_{i=1}^N (y^{(i)} - t^{(i)})^2 = \frac{1}{2N} \sum_{i=1}^N (w^T x^{(i)} + b - t^{(i)})^2$$

Il prossimo problema è minimizzare la funzione costo, tipicamente usando disuguaglianze (*algebrico*), o di calcolo (*derivate*). Le soluzioni sono di approccio *diretto* (trovo parametri ottimali) o *iterative* (mi ci avvicino). Quando parliamo di *Loss Function* lavoriamo su un singolo dato, mentre con *Cost Function* parliamo di tutto il dataset, e dell'errore complessivo da minimizzare.

Un approccio diretto è **Linear Algebra**:

Viene generato un sottospazio formato Xw (quindi la retta). Dato un target t , la distanza minore rispetto alla

retta è quella ortogonale. Quindi, data la distanza $y^* - t$, cerchiamo quella perpendicolare, ovvero con prodotto scalare nullo. Questo approccio può essere inesatto, a causa di instabilità.



$$\begin{aligned} \mathbf{X}^T(\mathbf{y}^* - \mathbf{t}) &= 0 \\ \mathbf{X}^T \mathbf{X} \mathbf{w}^* - \mathbf{X}^T \mathbf{t} &= 0 \\ \mathbf{X}^T \mathbf{X} \mathbf{w}^* &= \mathbf{X}^T \mathbf{t} \\ \mathbf{w}^* &= (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{t} \end{aligned}$$

Altro approccio diretto è **Calculus**, in cui deriviamo la funzione **Loss** sia per **w** che per **b** (ovvero la nostra coppia). Così è facile passare alla funzione Costo, mettendo le derivate a 0.

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial w_j} &= \frac{d\mathcal{L}}{dy} \frac{\partial y}{\partial w_j} & \frac{\partial \mathcal{L}}{\partial b} &= \frac{d\mathcal{L}}{dy} \frac{\partial y}{\partial b} \\ &= \frac{d}{dy} \left[\frac{1}{2} (y - t)^2 \right] \cdot x_j & &= y - t \\ &= (y - t) x_j & \frac{\partial \mathcal{J}}{\partial w_j} &= \frac{1}{N} \sum_{i=1}^N (y^{(i)} - t^{(i)}) x_j^{(i)} & \frac{\partial \mathcal{J}}{\partial b} &= \frac{1}{N} \sum_{i=1}^N (y^{(i)} - t^{(i)}) \end{aligned}$$

Possiamo anche qui vettorizzare tutto ottenendo: $\nabla_{\mathbf{w}} \mathcal{J}(\mathbf{w}) = \mathbf{X}^T \mathbf{X} \mathbf{w} - \mathbf{X}^T \mathbf{t} = \mathbf{0}$

Infine, introduciamo il **Feature Mapping**, con il quale cambiamo lo spazio delle caratteristiche per una miglior cattura delle informazioni. Utile se il rapporto input/output non è lineare.

Mappo su un altro spazio $\psi(x) : R^D \rightarrow R^d$, e uso lo spazio di arrivo come input della regressione.

Ad esempio, $\sum_{i=0}^M w_i x^i = 0$, i vari pesi sono collegati in modo lineare, anche se le varie x non lo sono.

Se $M = 0$, il modello è troppo semplice, **underfitting**.

Se M è alto, il modello è troppo complesso, **overfitting**.

M è un *iperparametro*. Per evitare che M aumenti troppo, si introduce un regolarizzatore, che spinge a cercare pesi piccoli. Ne è un esempio L^2 **penalty**.

$$\mathcal{J}_{reg}(\mathbf{w}) = \mathcal{J}(\mathbf{w}) + \lambda \mathcal{R}(\mathbf{w}) = \mathcal{J}(\mathbf{w}) + \frac{\lambda}{2} \sum_j w_j^2$$

Un'alternativa "iterativa" per *minimizzare la funzione costo* è l'algoritmo del **gradiente**.

Inizializziamo i pesi (ad es. 0), calcoliamo la derivata $\frac{\delta \mathcal{J}}{\delta w_j}$

- Se positiva (sale), dobbiamo spostarci verso sinistra.
- Se negativa (scende), dobbiamo spostarci verso destra.

Poi aggiorniamo i pesi sottraendogli la derivata, moltiplicata per l'iperparametro *learning rate*, che ci dice

$$w_j \leftarrow w_j - \alpha \frac{\delta \mathcal{J}}{\delta w_j}$$

quanto velocemente **w** cambia.

Anche qui possiamo mettere tutto sotto forma di vettore (lavoro con **w** e non più con w_j , e al posto della

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{N} \sum_{i=1}^N (y^{(i)} - t^{(i)}) \mathbf{x}^{(i)}$$

funzione costo, nella **regressione lineare**, ho:

Nella linear regression, anche se spesso esiste soluzione diretta, si usa comunque questa tecnica, in quanto semplice ed efficiente rispetto all'inversa di una matrice. Tramite **training curves** tracciamo il *training cost* in funzione delle *iterazioni*: andando avanti, il *training cost* dovrebbe scendere.

$$\frac{\partial \mathcal{J}}{\partial \theta} = \frac{1}{N} \sum_{i=1}^N \frac{\partial \mathcal{L}^{(i)}}{\partial \theta}$$

Chiamiamo $\theta = (w, b)$, possiamo dire , tuttavia se abbiamo tanti *training examples* dovremmo sommarli tutti e non è pratico.

Si passa a **stochastic gradient descent**, si usa un solo *training example* scelto a caso per aggiornare:

$$\theta \leftarrow \theta - \alpha \frac{\delta \mathcal{L}^{(i)}}{\delta \theta}$$

Così non dipendiamo dal numero di training. SGD fornisce stima imparziale, infatti:

$$E \left[\frac{\partial \mathcal{L}^{(i)}}{\partial \theta} \right] = \frac{1}{N} \sum_{i=1}^N \frac{\partial \mathcal{L}^{(i)}}{\partial \theta} = \frac{\partial \mathcal{J}}{\partial \theta}$$

Tuttavia, basarsi su un solo dato aumenta la varianza (aggiornamenti molto diversi tra loro), quindi si prendono **mini-batch** randomici di dimensione M (iperparametro). Se M è grande, varianza bassa ma calcolo più costoso. Spesso $M = 100$.

Qui si preferisce avere all'inizio un *learning rate* elevato che poi decresce. Se troviamo un **punto ottimo locale**, e la funzione è convessa, è un minimo globale.

LOGISTIC REGRESSION

Parliamo di **classificazione**, ovvero prediciamo un target categorico (successo/fallimento, 0/1, animali).

Alla base c'è **Linear Classification**, dove $t = 1$ = *esempio positivo*, $t = 0$ = *esempio negativo*.

Qui la predizione è basata su *funzione lineare* insieme a una soglia r , che se oltrepassata dà come risultato 1, sennò 0.

$$z = \mathbf{w}^T \mathbf{x} + b$$

$$y = \begin{cases} 1 & \text{if } z \geq r \\ 0 & \text{if } z < r \end{cases}$$

$$\mathbf{w}^T \mathbf{x} + b \geq r \rightarrow \mathbf{w}^T \mathbf{x} + \underbrace{b - r}_{w_0} \geq 0$$

Possiamo semplificare con: , se $b = 0$, allora $w_0 = b$ (bias), e quindi aumentiamo di dimensione anche l'input \mathbf{x} , aggiungendoci un 1 come già visto.

$$z = \mathbf{w}^T \mathbf{x}$$

$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

Otteniamo un gradino, che come vedremo non ci piace molto, perché non è continua, derivabile, ...

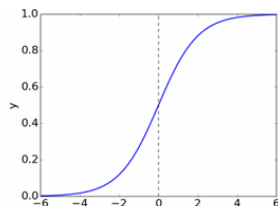
Cerchiamo un qualcosa simile ma più "smooth", e visto che oscilliamo tra 0 e 1, cerchiamo nella *probabilità*.

Nella **Logistic Regression**, vogliamo una funzione $f_w(x)$ che approssima $P(y = 1 | x; w)$,

cioè probabilità condizionata che l'uscita $y = 1$ dato un certo input x , il tutto parametrizzato in base ai pesi:

$$f_w(x) = \sigma(\mathbf{w}^T \mathbf{x})$$

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



Qui il confine lo abbiamo se $z = \mathbf{w}^T \mathbf{x} = 0$, e qui abbiamo la retta di separazione tra $y = 0$ e $y = 1$.

$\mathbf{w}^T \mathbf{x} = w_1 x_1 + w_2 x_2 = 0$ se lavoriamo in R^2 , ma risolvere questa equazione vuol dire trovare una retta.

Data questa funzione, non posso prenderla e metterla nella **Squared Cost Function**, in quanto non è convessa.

$$J(\mathbf{w}) = \frac{1}{2N} \sum_{i=1}^N (f_{\mathbf{w}}(\mathbf{x}^{(i)}) - t^{(i)})^2 = \frac{1}{2N} \sum_{i=1}^N \left(\frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}}} - t^{(i)} \right)^2$$

La soluzione è nel metodo della **massima verosimiglianza**, ovvero la probabilità di osservare **target t** dati gli **input x** (in funzione di **w**, in quanto noi agiamo su loro che influenzano **x**, ma non tocchiamo **x**).

"Dato un evento, quale è la probabilità che esso si verifichi, vedendo gli input?"

(es: se lanciando la moneta esce TTTT, probabilità che esca testa date le uscite non è di sicuro 0.5)

Possiamo riscrivere il tutto come una produttoria tra le verosimiglianze, in quanto il target "**t**" è influenzato solo dall'input "**t**", non da altri.

$$l(\mathbf{w}) = P(\mathbf{t}|\mathbf{x}; \mathbf{w}) = \prod_{i=1}^N P(t^{(i)}|\mathbf{x}^{(i)}; \mathbf{w})$$

Noi cerchiamo la **massima verosimiglianza**, quindi il **w** tale che massimizzi $l(\mathbf{w})$.

Per semplificare il tutto, si usa la funzione monotona **logaritmo**, che ci permette di usare sommatorie:

$$\mathbf{w}_{MLE} = \arg \max_{\mathbf{w}} \sum_{i=1}^N \log P(t^{(i)}|\mathbf{x}^{(i)}; \mathbf{w}) = \arg \max_{\mathbf{w}} \sum_{i=1}^N [t^{(i)} \log P(t^{(i)} = 1|\mathbf{x}^{(i)}; \mathbf{w}) + (1 - t^{(i)}) \log (1 - P(t^{(i)} = 1|\mathbf{x}^{(i)}; \mathbf{w}))]$$

NB: nel caso binario, possiamo spezzare in due.

Nella Logistic Regression, viene usata l'opposto della verosimiglianza come funzione costo (convessa).

Mettiamo dentro la funzione $f_{\mathbf{w}}(\mathbf{x})$ che abbiamo proprio cercato per approssimare la max verosimiglianza.

NB: Possiamo riscrivere la probabilità in modo compatto:

$P(t^{(i)}|\mathbf{x}^{(i)}; \mathbf{w}) = P(t^{(i)} = 1|\mathbf{x}^{(i)}; \mathbf{w})^{t^{(i)}} \cdot (1 - P(t^{(i)} = 1|\mathbf{x}^{(i)}; \mathbf{w}))^{1-t^{(i)}}$, a cui poi applichiamo l'operatore logaritmo.

$$\begin{aligned} \mathcal{J}(\mathbf{w}) &= - \sum_{i=1}^N [t^{(i)} \log P(t^{(i)} = 1|\mathbf{x}^{(i)}; \mathbf{w}) + (1 - t^{(i)}) \log (1 - P(t^{(i)} = 1|\mathbf{x}^{(i)}; \mathbf{w}))] \\ &= - \sum_{i=1}^N [t^{(i)} \log f_{\mathbf{w}}(\mathbf{x}) + (1 - t^{(i)}) \log (1 - f_{\mathbf{w}}(\mathbf{x}))] \\ &= - \sum_{i=1}^N \left[t^{(i)} \log \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}}} + (1 - t^{(i)}) \log \left(1 - \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}}} \right) \right] \end{aligned}$$

Problema: Non esistono soluzioni esplicite per calcolare il minimo.

Soluzione: Usiamo l'algoritmo del gradiente per aggiornare i pesi. Per arrivare alla Cost Function, partiamo dalla **Loss (cross-entropia)**, cioè la componente **singola** della Cost Function.

$$\mathcal{L}_{CE}(y, t) = -t \log(y) - (1 - t) \log(1 - y)$$

$$y = \frac{1}{1 + e^{-z}} \text{ and } z = \mathbf{w}^T \mathbf{x}$$

Noi vogliamo derivare rispetto ai parametri, ovvero **w**, solo che data la funzione, sapremmo farlo solo rispetto **y**, la quale è in funzione di **z**. Infine, **z** è in funzione di **w**. Usiamo quindi la catena di derivate:

$$\frac{\partial \mathcal{L}_{CE}}{\partial w_j} = \frac{\partial \mathcal{L}_{CE}}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial w_j}$$

$$\frac{\partial \mathcal{L}}{\partial y} = -\frac{t}{y} - \frac{(1-t)}{1-y} = \frac{-t+ty+y-ty}{y \cdot (1-y)} = \frac{y-t}{y \cdot (1-y)}$$

$$\frac{\partial y}{\partial z} = \frac{\partial [(1+e^{-z})^{-1}]}{\partial z} = \frac{(-1) \cdot (-e^{-z})}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}} = y \cdot (1-y)$$

$$\frac{\partial z}{\partial w_j} = \frac{\partial \overline{w^T \cdot \overline{x}}}{\partial w_j} = 1_j \cdot \overline{x} = x_j$$

Mettendo tutto insieme si ha: $\frac{\partial \mathcal{L}_{CE}}{\partial w_j} = \frac{(y-t)}{y \cdot (1-y)} \cdot y(1-y) \cdot x_j = (y-t)x_j$

Ora passiamo alla Cost Function, media delle Loss. Notiamo come sia lo stesso risultato della **linear regression**.

$$\begin{aligned} w_j &\leftarrow w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j} \\ &= w_j - \frac{\alpha}{N} \sum_{i=1}^N (y^{(i)} - t^{(i)}) x_j^{(i)} \end{aligned}$$

MULTICLASS CLASSIFICATION

Nel momento in cui abbiamo più target discreti, definiamo “quanto il modello preferisca k come predizione”

$$z_k = \sum_{j=1}^D w_{kj} x_j + b_k \text{ for } k = 1, 2, \dots, K$$

tramite: z_k , che vettorialmente è $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$, che diventa $\mathbf{Z} = \mathbf{W}\mathbf{x}$ se mettiamo \mathbf{b} in \mathbf{W} e $x_0 = 1$ in \mathbf{x} . Usiamo z_k così: $y_i = 1$ solo per la classe dove $i = \text{argMax}(z_k)$, cioè per la classe che massimizza z_k , $y_i = 0$ altrimenti.

Questa y però crea scalini, e quindi si introduce la **softmax**, l'uso di esponenziali amplifica tutto.

$$y_k = \text{softmax}(z_1, \dots, z_K)_k = \frac{e^{z_k}}{\sum_{k'} e^{z_{k'}}}$$

Se lavoriamo con vettori, possiamo usare la **cross-entropia**: (qui c'è solo la parte in cui t non è 0)

$$\mathcal{L}_{CE}(\mathbf{y}, \mathbf{t}) = - \sum_{k=1}^K t_k \log y_k = -\mathbf{t}^T (\log \mathbf{y})$$

Come già visto, vorremmo derivare rispetto w_k , e questo ci riporta a $(y_k - t_k)x$, dove y_k è proprio la softmax.

$$\begin{aligned} \frac{\partial \mathcal{L}_{CE}}{\partial \mathbf{w}_k} &= \frac{\partial \mathcal{L}_{CE}}{\partial \mathbf{z}_k} \cdot \frac{\partial \mathbf{z}_k}{\partial \mathbf{w}_k} = (y_k - t_k) \cdot \mathbf{x} \\ \mathbf{w}_k &\leftarrow \mathbf{w}_k - \alpha \frac{1}{N} \sum_{i=1}^N (y_k^{(i)} - t_k^{(i)}) \mathbf{x}^{(i)} \end{aligned}$$

0 o 1

softmax(Zk)

DECISION TREES

Parliamo sempre di “*supervised*”, l’input è la coppia $\{(x^{(i)}, t^{(i)})\}$, dove le $x^{(i)}$ sono descritte da alcune caratteristiche, e per associarle a $t^{(i)}$ si ha funzione sconosciuta $f: X \rightarrow T$ che mappa dalle istanze alle etichette. Come output abbiamo un’*ipotesi* $h \in H$ (*set di ipotesi*) $= \{h \mid h: X \rightarrow T\}$, quindi la funzione che meglio approssima la funzione obiettivo f . Le ipotesi formano un albero decisionale, in cui ogni *nodo* è un attributo, ogni ramo è *un possibile valore di tale attributo*, e quando arriviamo al *nodo foglia* abbiamo la predizione.

Secondo il **principio di Ockham**, l’albero decisionale migliore (problema NP-hard) è quello più piccolo (in assoluto, anch’esso NP-hard) che classifica bene tutti i training. Noi ne troveremo uno *abbastanza piccolo*.

Introduciamo l’algoritmo greedy **ID3**, che parte dalla radice e un albero vuoto. Si sceglie il “*miglior attributo*” rispetto al quale proseguire, dividendo poi rispetto tale attributo. La scelta del *migliore* può essere random, chi genera meno nodi figlio (o viceversa). ID3 usa in realtà il **Max-Gain**, ovvero sceglie l’attributo che genera figli senza necessità di andare oltre. (Ad esempio, se un attributo mette tutti gli “1” da una parte e tutti gli “0” dall’altra, non devo proseguire. Se questi fossero mischiati, dovrei andare avanti). Per *quantificare* tutto ciò entra in gioco la **teoria dell’informazione**, basata sul concetto di Entropia, che quantifica l’incertezza, se i risultati sono “più certi” essa è bassa, se siamo “sicuri” è 0, perché manco serve osservare. In formula:

$$H(Y) = - \sum_{y \in Y} P(y) \log_2 P(y)$$

Shannon ha anche dimostrato che il risultato di $H(Y)$ ci dice anche quanti *bit* ci servono per memorizzare l’informazione.

Esempio lancio moneta: $p = \frac{1}{2}$, allora $H(Y) = - \frac{1}{2} \log_2(\frac{1}{2}) - \frac{1}{2} \log_2(\frac{1}{2}) = 1$, serve 1 bit

Possiamo anche calcolare:

- Entropia congiunta (tutte combinazioni x, y):
$$H(X, Y) = - \sum_{x \in X} \sum_{y \in Y} P(x, y) \log_2 P(x, y)$$
- Entropia condizionata specifica per singolo x :
$$H(Y|X = x) = - \sum_{y \in Y} P(y|x) \log_2 P(y|x)$$
- Entropia condizionata media su tutte le x :
$$H(Y|X) = \sum_{x \in X} P(x) H(Y|X = x) = - \sum_{x \in X} \sum_{y \in Y} P(x, y) \log_2 p(y|x)$$

Tra le caratteristiche dell’entropia abbiamo: H non negativa, vale la chain rule, se X ed Y sono indipendenti allora X non influisce, conoscendo X l’incertezza può solo diminuire, conoscere già Y rende certa la conoscenza di Y (H nulla).

L’**information gain** è: $IG(Y|X) = H(Y) - H(Y|X)$, cioè incertezza totale (se stiamo nella radice è 1) sottratta quella nota (ovvero gli esempi sotto un nodo divisi gli esempi totali considerati).

Quindi, nell’algoritmo ID3, parto da albero vuoto, per ogni attributo calcolo IG, prendo l’etichetta che lo massimizza, per ogni valore di tale etichetta associo le istanze di training, applicando *iterativamente* ID3 (escludendo le etichette già usate). Termino quando in ogni foglia ho etichette tutte uguali (non devo andare oltre). Ciò vuol dire che si adatta molto bene, leggasi overfitting. Una soluzione, oltre a rimuovere attributi non importanti, è il **pruning**, ovvero valuto sul validation set l’eliminazione di un possibile nodo (tipicamente inferiore), vedendo se la precisione migliora. Passiamo ora al **K fold Cross Validation**, una tecnica per la ricerca di iperparametri. Abbiamo *training set* e *testing set*. Il *training set* viene diviso in k set più piccoli detti “folds”. Uso $k-1$ folds come training, e la restante la uso come *validation set*. Il tutto iterativamente k volte, perché il *validation set* potrebbe essere piccolo, e ovviamente scambiandolo con altri folds. Visti gli iperparametri che si comportano meglio, addestro *tutto* il training set e lo valuto sul *testing set*.

ENSEMBLING METHOD

Gli **ensembling methods** sono tecniche utilizzate per combinare le previsioni di più modelli al fine di ottenere una previsione complessiva più accurata o robusta rispetto all'utilizzo di un singolo modello. Supponiamo di avere diversi training set (tutti $\in P_{sample}$). Scelto un **punto x fisso**, la sua previsione y cambia in funzione del training set fornito. Tali misure si possono **decomporre** in termini di **media** e **varianza**. y è quindi una V.A., e noi vogliamo il vero obiettivo $P(y|x)$.

Vogliamo come sempre minimizzare la Cost Function, che è la media delle Loss, cioè la differenza quadrata tra l'etichetta effettiva e quella calcolata. Sia l'etichetta $t = g(x) + \varepsilon$, ovvero parte deterministica più rumore, mentre $f_w(x)$ è la nostra previsione:

$$E[(t - f_w(x))^2], \text{ aggiungo e tolgo } g(x), \rightarrow E[(t - g(x) + g(x) - f_w(x))^2],$$

chiamo $a = t - g(x)$; $b = g(x) - f_w(x)$ e faccio il quadrato:

$$E[(t - g(x))^2] + E[(g(x) - f_w(x))^2] + 2E[(g(x) - f_w(x))(t - g(x))], \text{ sviluppo i termini:}$$

$$E[(t - g(x))^2] + E[(g(x) - f_w(x))^2] + 2\{E[g(x) \cdot t] - E[g(x)^2] - E[f_w(x) \cdot t] + E[f_w(x) \cdot g(x)]\}$$

Se sviluppo il doppio prodotto, sostituendo le t , scopriamo che questo si toglie tutto.

Rimaniamo con: $E[(t - f_w(x))^2] = E[(t - g(x))^2] + E[(g(x) - f_w(x))^2]$, anche qui possiamo sostituire:

$$E[(t - f_w(x))^2] = E[(g(x) + \varepsilon - g(x))^2] + E[(g(x) - E[f_w(x)] + E[f_w(x)] - f_w(x))^2] =$$

qui come prima aggiungo e tolgo $E[f_w(x)]$, chiamo $a = g(x) - E[f_w(x)]$ e $b = E[f_w(x)] - f_w(x)$

$$E[\varepsilon^2] + E[(g(x) - E[f_w(x)])^2] + E[(E[f_w(x)] - f_w(x))^2] + 2E[(E[f_w(x)] - f_w(x)) \cdot (g(x) - E[f_w(x)])]$$

come prima, il doppio prodotto si toglie, così rimane:

$$E[(t - f_w(x))^2] = \text{var}[\varepsilon] + E[(g(x) - E[f_w(x)])^2] + E[(E[f_w(x)] - f_w(x))^2], \text{ che riscrivo come:}$$

$$E[(t - f_w(x))^2] = \text{bias}(f_w(x))^2 + \text{var}(f_w(x)) + \sigma^2, \text{ l'ultimo termine è l'imprevedibilità, } \mathbf{Bayes\ error}.$$

Bagging

Se potessimo campionare m *training set* indipendenti da una probabilità P_{sample} , potrei per ogni campione trovare una previsione y_i , e considerare la previsione "finale" y come *la media di queste y_i* .

Il bias, cioè quanto è sbagliata la previsione attesa (underfitting), non cambia, essendo la media di y non cambia.

Ciò che cambia è la varianza (quanto variano le previsioni, cioè la complessità del modello, overfitting).

$$\text{Var}[y] = \text{Var}\left[\frac{1}{m} \sum_{i=1}^m y_i\right] = \frac{1}{m^2} \sum_{i=1}^m \text{Var}[y_i] = \frac{1}{m} \text{Var}[y_i]$$

Partire da una distribuzione e ottenere più dati è costoso. Una soluzione è data dal **bagging**, in cui invece partiamo dai dati che abbiamo, vi introduciamo una distribuzione per campionarli (anche con reintroduzione), creiamo più training set, classifichiamo su questi e poi facciamo una media.

Quindi, se ad esempio ho 7 dati, posso creare 3 training set con 6 dati ciascuno, con qualche dato che appare più volte. Nulla vieta di crearli in parallelo. L'unica limitazione è che i dataset non sono indipendenti, e quindi, data la correlazione ρ , si ha $\text{Var}[y] = \frac{1}{m} (1 - \rho) \sigma^2 + \rho \sigma^2$

Quindi il **bagging**:

- Riduce l'overfitting/varianza perché fa la media.
- Non riduce il bias/underfitting perché i modelli sono tutti uguali.

Un esempio di albero decisionale con bagging è dato dal **Random Forest**, in cui si sceglie casualmente un insieme di attributi per un albero (ID3 lavora su tutti, e quindi produrrebbe alberi simili), che lavorerà solo con quelli.

Una caratteristica comune di ciò che abbiamo visto è l'uso, per tutti i membri, degli stessi "pesi".

Con la tecnica di **Boosting**, addestriamo più classificatori in sequenza, e all'iterazione successiva ci si concentra su ciò che è stato sbagliato in precedenza, usando dei pesi diversificati. Essendo iterativo, non può operare in parallelo. Come posso pesare?

Abbiamo un *missclassification rate* $\frac{1}{N} \sum_{n=1}^N \mathbb{I}[h(x^{(n)}) \neq t^{(n)}]$ che, dato il classificatore h , pesa equamente le predizioni *sbagliate*. Noi vogliamo pesare di più ciò che è sbagliato, introducendo un peso

positivo, tale che la somma di tutti i pesi sia 1: $\sum_{n=1}^N w^{(n)} \mathbb{I}[h(x^{(n)}) \neq t^{(n)}]$

Un algoritmo *sequenziale* che si basa su tale logica è **AdaBoost**:

Si parte da un *classificatore debole ma computazionalmente efficiente* (ad esempio che classifichi 51% sì, 49% no) e lo si migliora enfatizzando i suoi errori. Gli step sono:

A ogni iterazione, alleno il classificatore coi pesi correnti, vedo il suo comportamento assegnandogli "un peso in funzione di come si è comportato", dato da:

$\beta_t = \frac{1}{2} \ln\left(\frac{1-\varepsilon_t}{\varepsilon_t}\right)$, dove $\varepsilon_t = \sum_{i: y_i \neq h_t(x_i)} w_{t,i}$ è la somma dei pesi classificati male: se è piccolo fa aumentare

l'importanza del classificatore. Poi aggiorno i pesi $w_{t+1,i} = w_{t,i} \cdot e^{-\beta_t y_i h_t(x_i)} \forall i$, li normalizzo: $w_{t+1,i} = \frac{w_{t+1,i}}{\sum_{j=1}^n w_{t+1,j}}$

Alla fine, ritorno l'ipotesi: $H(x) = \text{sign}\left(\sum_{t=1}^T \beta_t h_t(x)\right)$

Andando avanti il bias (underfitting) si riduce, la varianza (overfitting) può aumentare, perché il modello diventa più complesso. Il modello finale è la combinazione di tutti i classificatori in questi T passi, il risultato non è lineare (sarà un'area). Negli algoritmi di *regressione logistica* (classificatori) posso incorporare questi pesi nella funzione di costo. In casi come ID3 in cui non c'è supporto ai pesi, posso sostituire il campione originale con uno "pesato", ovvero moltiplicato per la probabilità di essere scelto.

NEURAL NETWORKS AND DEEP LEARNING

Un modello basico è il **Perceptron**, in cui lavoro con numeri reali, ogni input ha un *peso* e ogni output è ottenuto mediante una *funzione di attivazione* ϕ , consideriamo anche un *bias* (che "aiuta" a predire qualcosa, come un valore >0 , quando i dati posseduti non ci riescono, perché ad esempio sono nulli). Possiamo vederlo come classificatore binario in $\{-1, +1\}$, in cui, dati i pesi inizializzati a 0, si usa un *training instance* alla volta per predire y , e se la predizione è sbagliata si rinforza la connessione:

$$w_j \leftarrow w_j + (t^{(i)} - y^{(i)}) x_j^{(i)}$$

Cioè stiamo "regolando" la retta che separa i due casi.

Successivamente, è stato rivisto come applicazione del SGD, anche se in origine non era stocastico, è stata aggiunta una **funzione di perdita** per la singola istanza:

$$\begin{aligned} L_{0,1}^{(i)} &= \frac{1}{2} (t^{(i)} - \text{sign}\{w^T x^{(i)}\})^2 = \frac{1}{2} (t^{(i)2} - 2t^{(i)} \text{sign}\{w^T x^{(i)}\} + \text{sign}\{w^T x^{(i)}\}^2) = \\ &= \frac{1}{2} (1 - 2t^{(i)} \text{sign}\{w^T x^{(i)}\} + 1) = 1 - t^{(i)} \text{sign}\{w^T x^{(i)}\}, \text{ infatti al quadrato sia } \{-1, +1\} \text{ danno } 1. \end{aligned}$$

Ricordiamo che $\text{sign}(x) = -1$ se $x < 0$; 0 se $x = 0$; 1 se $x > 0$.

Il problema è che la funzione segno non è differenziabile, e quindi si usa una funzione che la approssima, la **Loss Function Surrogata** $L^{(i)} = \max\{0, -t^{(i)}(w^T x^{(i)})\}$, e calcolare SGD e aggiornamento, come sempre:

$$\nabla_w \mathcal{L}^{(i)} = (y^{(i)} - t^{(i)})x^{(i)} \quad \text{e} \quad w \leftarrow w - \eta \nabla_w \mathcal{L}^{(i)} = w + \eta(t^{(i)} - y^{(i)})x^{(i)}, \text{ con } \eta \text{ learning rate.}$$

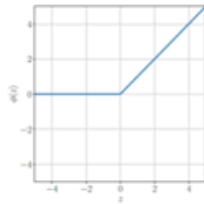
Il Perceptron ha un singolo “neurone”, ne vorremmo di più.

Nelle **Reti Neurali Feedforward** si ha un vettore di input \mathbf{x} , dei livelli nascosti (livello i produce output $h^{(i)}$) caratterizzati da *width* (unità nel singolo livello nascosto) e *depth* (quanti livelli nascosti ho). Un livello nascosto i applica la stessa *funzione di attivazione* per tutte le sue unità, livelli diversi hanno funzioni diverse, e alla fine producono il vettore output \mathbf{y} .

Parlando di funzioni di attivazione, generalmente abbiamo, per il livello i , $h^{(i)} = \phi^{(i)}(W^{(i)}\mathbf{h}^{(i-1)} + \mathbf{b}^i)$, dove $\mathbf{h}^{(i-1)}$ è ciò che è stato fatto al livello precedente, a cui applico il peso associato alla connessione dell'unità precedente all'unità attuale. Le funzioni di attivazione principali sono:

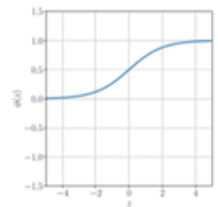
Per la Regression (costo casa)
ReLU (Rectified Linear Unit)

$$\phi(z) = \max\{0, z\}$$



Per una classificazione binaria
Logistic sigmoid

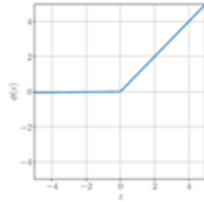
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



Parametric ReLU (PReLU)

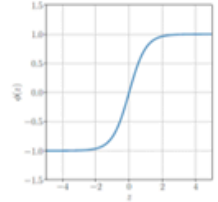
$$\phi(z) = \begin{cases} z & z > 0 \\ pz & z \leq 0 \end{cases}$$

e.g., $p = 0.01$



Mapa in [-1,+1], dati centrati in 0
Hyperbolic Tangent

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



Per una classificazione multi - classe
possiamo usare la SOFTMAX

$$z = W^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}$$

$$y_i = \frac{e^{z_i}}{\sum_{k=1}^K e^{z_k}}$$

Sostanzialmente, nella **Neural Network** stiamo calcolando una funzione composta da tutte le attivazioni presenti nei vari livelli. Nell'ultimo livello lavoriamo con un vettore che non è \mathbf{x} , ma un vettore calcolato dai livelli precedenti. Le funzioni sono *non-lineari*, altrimenti sarebbe come lavorare con un unico livello.

Quanto possono essere potenti queste reti?

Il **Teorema di approssimazione universale** afferma che, una rete *feedforward* con un livello di uscita lineare ($w_0x_0 + w_1x_1 + \dots$), con una qualsiasi funzione di attivazione avente *output* in un intervallo finito e un numero sufficiente di *hidden units* (quindi si parla di un solo hidden layer con una certa *width*) *può approssimare qualsiasi funzione misurabile di Borel*.

Il problema è non ci viene detto come fare il training per ottenerla (potremmo averne una errata), né ci dice quante unità ci servono.

Nella pratica si scopre che si possono ottenere buoni risultati con più livelli nascosti (*deep*) composti da poche unità, quindi parliamo di **Deep Learning** sulle **Neural Networks**. La differenza rispetto Machine Learning è che ora vogliamo apprendere sia la rappresentazione che la mappatura dei dati, mentre in ML

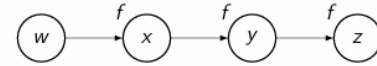
abbiamo già la rappresentazione. Come si allenano?

Facciamo sempre la stessa cosa, ovvero definiamo la Loss Function, la Cost Function e una funzione di regolarizzazione, che è pesata in base a un iperparametro: $J(\theta) = \sum_{i=1}^N \frac{1}{2} (t^{(i)} - y^{(i)})^2 + \lambda \Omega(\theta)$

Potremmo quindi usare la SGD, però il problema è nell'uso di funzioni *non lineari* e *non convesse*.

Si usa quindi l'algoritmo di **Backpropagation** per calcolare il gradiente, esso ha due fasi:

- **Forward**: Da un'istanza del training set, passando per i vari livelli, produco la previsione che userò per *calcolare la funzione costo*.



Introduciamo un *computational graph*, struttura di questo tipo

che

rende più ordinate operazioni come $z = f(y) = f(f(x)) = f(f(f(w)))$ e $\frac{\partial z}{\partial w} = f'(y)f'(x)f'(w)$

Se diciamo che le prime k variabili sono di input, quelle successive sono *funzioni* e l'ultima n è di output, allora ciò che facciamo è applicare i pesi e bias all'output del livello precedente:

$a^{(k)} \leftarrow W^{(k)} h^{(k-1)} + b^{(k)}$ e poi applicare la funzione di attivazione $h^{(k)} \leftarrow f^{(k)}(a^{(k)})$.

Alla fine avrò $J(\theta) = L(y, t) + \lambda \Omega(\theta)$. A noi interesserà solo $L(y, t)$, l'altra è indipendente.

- **Backward**: Calcolo il gradiente della funzione Loss rispetto ai parametri, percorrendo la NN in verso opposto. Sia g il *gradiente della funzione di costo rispetto allo strato k* , ovvero $g \leftarrow \nabla_{h^{(k)}} L$, all'inizio sarà $g \leftarrow \nabla_y L$, cioè rispetto l'output. Andando indietro, vogliamo passare da $h^{(k)}$ al valore di preattivazione $a^{(k)}$, sempre stesso livello, infatti sussiste $a^{(k)} = W^{(k)} h^{(k-1)} + b^{(k)}$ (cioè cosa entra nel livello k) e $h^{(k)} = f^{(k)}(a^{(k)})$ (cosa esce dal livello k). Quindi, ciclicamente fino ad arrivare al livello iniziale, vogliamo $g = \nabla_{a^{(k)}} L$, che sviluppando è:

$$g \leftarrow \frac{\partial L}{\partial a^{(k)}} = \frac{\partial L}{\partial h^{(k)}} \cdot \frac{\partial h^{(k)}}{\partial a^{(k)}} = g \odot f^{(k)'}(a^{(k)}). \text{ NB: notiamo che in } g \text{ appare sè stessa e altro.}$$

A sua volta, $a^{(k)}$ è in funzione dei parametri $W^{(k)}$ e $b^{(k)}$, quindi andiamo a derivare J rispetto loro, ovvero

Non siamo scesi di livello, siamo solo passati alla pre-attivazione $a^{(k)}$, la quale è in funzione dei parametri $W^{(k)}$ e $b^{(k)}$, quindi deriviamo J rispetto loro, ad esempio per $W^{(k)}$:

$$\frac{\partial J}{\partial W^{(k)}} = \frac{\partial L}{\partial W^{(k)}} + \frac{\partial \Omega(\theta)}{\partial W^{(k)}} = \frac{\partial L}{\partial a^{(k)}} \cdot \frac{\partial a^{(k)}}{\partial W^{(k)}} + \nabla_{W^{(k)}} \Omega(\theta) = g \cdot h^{(k-1)^T} + \nabla_{W^{(k)}} \Omega(\theta), \text{ dove la } g \text{ che abbiamo messo in realtà include } g \odot f^{(k)'}(a^{(k)}).$$

Per concludere la parte nel ciclo, abbiamo lavorato sempre sullo stesso livello, ma in $a^{(k)}$ c'è anche

$$h^{(k-1)} \text{ del livello precedente, e quindi } \frac{\partial L}{\partial h^{(k-1)}} = \frac{\partial L}{\partial a^{(k)}} \cdot \frac{\partial a^{(k)}}{\partial h^{(k-1)}} = g \cdot W^{(k)^T}$$

Finora abbiamo fatto le considerazioni su una singola istanza di input, ma è meglio lavorare con *minibatch*, potendo sfruttare il parallelismo.

GUIDE PER NEURAL NETWORK

Per **REGRESSIONE**: Usiamo *un neurone* per ogni feature (spazio 3D, 3 feature, 3 neuroni), tra 1 e 5 livelli nascosti aventi dalle 10 alle 100 unità, in proporzione alle feature comunque. Le unità di output dipendono dai valori da predire, come funzione di attivazione per *input* e *hidden layers* si usa ReLU, mentre per il livello di uscita *ReLU* per output positivi, *tanh* se lavoro in intervalli, oppure *nulla*. Come Loss usiamo errore quadratico medio MSE o errore medio assoluto MAE.

Per **CLASSIFICAZIONE**: Per il livello di ingresso e gli hidden layers si usa sempre ReLU. Le differenze sono avere un'unità di input per ogni classe (una sola se binario), usare la *sigmoid* in uscita se l'output è tra 0 e 1, altrimenti per multi-classe *softmax*, e come Loss usiamo *Cross-Entropy Loss*.

EVALUATION

Per quantificare le prestazioni nella regressione usiamo la **Root Mean Square Error**:

$$\sqrt{\frac{1}{N} \sum (y^{(i)} - t^{(i)})^2} = \sqrt{J(y, t)}$$

Per quantificare le prestazioni in un contesto di classificazione usiamo l'**accuracy**, ovvero il rapporto percentuale tra le istanze correttamente classificate e quelle totali. Abbiamo anche la **confusion matrix**, che sulla diagonale principale ci dice, in percentuale, quante volte un'istanza *c* è stata ben identificata. Error e Accuracy non bastano, perché non considerano tutti gli aspetti possibili.

Nel caso di classificatore binario introduciamo:

- **Precision**: $\frac{TP}{TP+FP}$, ovvero, di tutte le volte che ho detto "vero", quante volte ho etichettato bene.
- **Recall**: $\frac{TP}{TP+FN}$, ovvero, di tutte le volte che l'istanza è "vera", quante volte ho etichettato bene.
- Spesso vengono combinate in **F1 score**: $2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$, elevato solo quando lo sono entrambe.

Tutto ciò ha senso finché ho delle etichette che mi dicono quale sia la risposta corretta. Se non ci fossero, dovrei parlare di **clustering**, per individuare gruppi di punti vicini che probabilmente sono simili.

Per definire questi gruppi usiamo l'algoritmo **K-means**, che ha il compito di trovare sia il centro del cluster sia a quale cluster assegnare i vari punti, per ridurre al minimo le distanze, in formula, assumendo come centri m_k e come appartenenza al centro k il valore, che opera come "indicatrice": $r_k^{(n)} = [0_1, \dots, 1_k, \dots, 0_n]$

$$\min_{\{m_k\}, \{r^{(n)}\}} \mathcal{J}(\{m_k\}, \{r^{(n)}\}) = \min_{\{m_k\}, \{r^{(n)}\}} \sum_{n=1}^N \sum_{k=1}^K r_k^{(n)} \|m_k - x^{(n)}\|^2$$

In pratica stiamo calcolando la distanza euclidea $\sqrt{(m_1 - k_1)^2 + (m_2 - k_2)^2 \dots}$

La soluzione ottimale è NP-hard, e la difficoltà sta nel fatto di trovare insieme i due parametri. Con K-means:

- Inizializziamo in modo casuale i centri m_1, \dots, m_k dei K cluster.
- Assegnazione dei punti $x^{(n)}$ ai centri più vicini tramite $r^{(n)}$ (è lui che li setta a 0 o 1, quindi mi porto tutto il vettore), vedendo quindi la distanza minore dai vari centri e quindi assegnandolo a chi ha distanza minore. Ho ottimizzato rispetto $r^{(n)}$.
- Ogni centro viene reimpostato sulla media dei dati, ovvero: $m_i = \frac{\sum_n r_i^{(n)} x^{(n)}}{\sum_n r_i^{(n)}}$, il centro è stato spostato

facendo la derivata rispetto ∂m_i e ponendo a 0 la formula con il minimo scritta poco sopra. Al numeratore abbiamo la somma delle coordinate dei punti in "i", sotto il numero dei punti in "i".

Il numero di iterazioni da fare si può determinare osservando quando le assegnazioni smettono di cambiare. Per il numero di cluster K , un valore più alto aumenta la precisione, rendendo i centri dei cluster più centrali. Scegliamo K in modo che per i K precedenti si ha una diminuzione dell'errore rapido e per i K successivi tale la riduzione diventi molto più lenta. Si parla di **elbow method**. (gomito)

REGULARIZATION

Vogliamo che gli algoritmi si comportino bene sia sul training set, ma anche davanti nuovi dati.

Vengono quindi introdotte le **funzioni di regolarizzazione**, che possono aggiungere vincoli o penalità, ciò che vogliamo è generalizzare il modello, per avere un comportamento migliore sul testing, a scapito del training. Ad esempio, si considera una penalità nella funzione obiettivo $J(\theta; X; y) = L(\theta; X; y) + \lambda \Omega(\theta)$,

dove $\lambda \in (0, \infty)$ è un iperparametro. Spesso si usano **Norma L2** $\Omega(\theta) = \frac{1}{2} \|w\|_2^2$ o **Norma L1** $\Omega(\theta) = \|w\|_1$, questo per evitare di avere pesi troppo grandi, e quindi connessioni non necessarie.

Per generalizzare, abbiamo bisogno comunque di tanti dati spesso non disponibili.

Con la **Dataset Augmentation**, spesso usato nelle immagini, in cui creiamo dei *fake data* (con il dubbio che potrebbero essere irrealistici) applicando *rumore all'input* (tocco pixel), ai *pesi* o alle *unità nascoste*.

Un altro approccio è **Early Stopping**, in cui fermiamo il training prima che si cada in overfitting.

Abbiamo tre parametri: n è il numero di step di addestramento che vogliamo fare tra due verifiche, P è la pazienza, ovvero quante volte di fila posso vedere che l'errore sul validation set peggiori prima di fermarsi,

θ_0 sono i parametri iniziali, θ i parametri attuali, θ^* i migliori parametri trovati fino a ora, infine j indica il numero di volte che abbiamo visto il *validation error* (salvato in v , inizializzato a ∞) aumentare.

Lo svolgimento è il seguente:

θ si aggiorna per n iterazioni, aggiorniamo a priori $j \leftarrow j + 1$, calcolo il *validation error* v' , lo confronto col precedente, se $v' < v$ (ho migliorato), allora imposto questo v' come validation error di riferimento, $\theta^* \leftarrow \theta$, resetto j e metto in i^* le iterazioni passate per trovare questi parametri ottimi.

Se non entrassi nel confronto, j non si resetta, anzi continua a crescere, se supera P ho terminato.

Come visto, richiede *validation set* (che prelevo dal training), quindi posso *addestrare da zero tutto il training set per i^* iterazioni* oppure *continuo da dove ho terminato usando tutti i dati*.

Altra tecnica è il **Dropout**, basata su Bagging (creo dataset diversi usando più volte gli stessi dati), presenta un numero *esponenziale* di reti neurali diverse. Per farlo, basta scartare qualche neurone a caso (metto a 0 l'uscita) a *ogni step di addestramento*. La scelta su chi scartare è fatta mediante **dropout probability**, che con probabilità p lascia presente l' i -esima unità $h_i^{(l)}$, altrimenti la pone a 0. Ciò viene fatto *solo nel training*, mentre passando nel *testing* devo scalare tutti i parametri per una probabilità p , cioè $W_{test}^{(l)} = p^{(l)} W^{(l)}$, questo perché altrimenti *training e testing* potrebbero lavorare su range diversi.

OPTIMIZATION ALGORITHMS FOR NNs

Nelle reti neurali non abbiamo finora considerato il costo di allenare tutto ciò che abbiamo detto.

Entrano in gioco fattori di *ottimizzazione*, infatti in ML abbiamo sempre lavorato con funzioni *convesse*.

Nei **deep models** abbiamo tanti *minimi locali* e anche delle *selle*. Come "ottimizziamo" ?

Possiamo partire dallo **Stochastic Gradient Descent SGD**, leggermente modificato:

Definite delle *epoche*, campioniamo un *minibatch* dal training set ed eseguiamo *backpropagation*, e aggiorniamo i parametri $\theta \leftarrow \theta - a_k g$, con il learning rate variabile ad ogni epoca. Crea rumore.

Come varia? La cosa più semplice è, impostando che $\sum_{k=1}^{\infty} a_k = \infty$ e $\sum_{k=1}^{\infty} a_k^2 < \infty$ (α nè troppo grande nè troppo piccolo), allora si fa decrescere un valore α_k che parte dal massimo α_0 fino a un altro valore α_T , raggiunto dopo T iterazioni, le cui successive non modificano più α_k . Non esistono regole fisse per scegliere i bound, e la convergenza è spesso lenta, quindi vediamo altre idee:

La tecnica del **Momentum** affronta la lentezza della convergenza mantenendo una *media mobile dei gradienti passati*, così se tutti i gradienti puntano nella stessa direzione, aumentiamo la velocità. La modifica principale nell'algoritmo SGD consiste quindi nell'introdurre la velocità $v \leftarrow \beta v - \alpha_k g$ usata per aggiornare $\theta \leftarrow \theta + v$, dove $\beta \in (0, 1)$ è un iperparametro che ci informa quanto "sfumano" i gradienti precedenti.

Un'altra versione è **Nesterov Momentum**, in cui prima si effettua il movimento aggiornando i parametri in funzione di v , e dopo calcoliamo la nuova velocità per aggiornare i futuri parametri. Empiricamente va meglio.

Una problematica di tutto ciò che abbiamo visto è la dipendenza di tutti gli algoritmi dal *learning rate*, se preso male rovina tutto!

Con **AdaGrad (Adaptive Gradient)** si vuole usare un tasso di apprendimento separato per ogni parametro del modello, che viene adattato automaticamente durante l'addestramento. L'idea è che se un parametro è stato aggiornato molto in passato, ora va aggiornato di meno. Prendiamo sempre un *minibatch* e salviamo in g il *gradiente della funzione Costo*. In r accumuliamo il *l'element wise del gradiente* ($g \odot g$) e aggiorniamo $\theta \leftarrow \theta - \frac{\alpha}{\delta + \sqrt{r}} \odot g$, dove δ è una costante molto piccola (evita la divisione per 0). Comporta overhead.

NB: L'element wise è il prodotto elemento per elemento, simile a fare il quadrato, ma formalmente non lo è.

RMSProp

Qui ciò che si cambia è l'accumulazione dei gradienti, fatta tramite *media mobile esponenziale tra il nuovo gradiente e la media precedente*. Infatti sia ρ il peso che do ai vecchi gradienti, allora:

$r \leftarrow \rho r + (1 - \rho)g \odot g$ che useremo, come già visto, in $\theta \leftarrow \theta - \frac{\alpha}{\delta + \sqrt{r}} \odot g$.

Adam

E' RMSProp con momentum. Si inizializzano t , v , r a 0. Si prende un *minibatch*, si calcola in g il gradiente, incremento le epoche $t \leftarrow t + 1$. Aggiorno poi il primo gradiente v e il secondo gradiente r , con β vicini a 1:

```

v ← β1v + (1 - β1)g /* Momentum update */
r ← β2r + (1 - β2)g ⊙ g /* 2nd moment update */
v̂ ←  $\frac{v}{1 - \beta_1^t}$ , r̂ ←  $\frac{r}{1 - \beta_2^t}$  /* Init. bias correction */
θ ← θ - α  $\frac{\hat{v}}{\sqrt{\hat{r} + \delta}}$ 

```

Correggiamo il bias (altrimenti saremmo lenti all'inizio), $1 - \beta$ è piccolo, al denominatore dà valori grandi, mentre l'esponente t , che cresce con le epoche, riduce andando avanti le β .

Parliamo spesso di "Inizializzazione", però non è mai stata approfondita. In realtà, la scelta degli *initial point* può essere fondamentale per la convergenza. Nel metodo **Glorot/Xavier** si prendono gli input da una gaussiana con *media* μ nulla e *varianza* $\sigma^2 = \frac{2}{N_{IN} + N_{OUT}}$, dove al denominatore abbiamo il numero di *input* e di *output in uno specifico livello*, quindi ogni livello sarà diverso. Esiste anche **He initialization**, dove consideriamo solo N_{IN} .

Procediamo con **Batch Normalization**, il quale non si occupa di ottimizzare, ma migliora la convergenza nei contesti Deep Neural Networks. Essa viene applicata *livello per livello* e cerca di standardizzare.

Prendiamo un *minibatch* B e un suo input $x \in B$, allora lo standardizziamo $BN(x) = \frac{x - \mu_B}{\sigma_B}$ per avere media nulla e varianza unitaria. Questo lo facciamo per i livelli nascosti, quando un output esce da un livello, viene applicata la normalizzazione, che diventa input per il livello dopo. In realtà, così facendo limitiamo la capacità espressiva di ogni livello, quindi si introducono γ e β (parametri da apprendere nell'addestramento), ovvero $BN(x) = \gamma \frac{x - \mu_B}{\sigma_B} + \beta$ in modo che la libertà espressiva rimanga, e che media e varianza dipendano solo da questi due parametri, non da cose interne. Tutto ciò va bene nel *training*, no nel *testing* (magari lavoriamo per predire un solo dato, non un batch), e quindi qui si usa una *media mobile della media* e una *media mobile della deviazione standard*.

CONVOLUTIONAL NEURAL NETWORKS

Abbiamo visto numerose ottimizzazioni, ciò non toglie che in alcuni casi, come la classificazione delle immagini, abbiamo troppi pixel da gestire e questo è un problema.

La soluzione è data dalle **reti neurali convoluzionali**, buone con strutture a griglia (come foto), esse devono presentare *almeno un livello di convoluzione*.

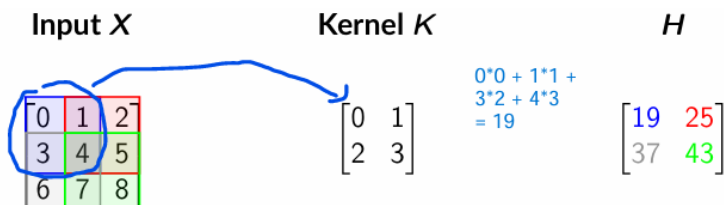
Il calcolo della convoluzione è utile per calcolare una stima che tenga conto dei cambiamenti, enfatizzando quelli più recenti. Ad esempio, un percorso in auto in cui campioniamo delle posizioni (che possono subire del rumore) le quali vengono corrette da una "media dei dati passati". Questa si scrive così:

$$s(t) = (x * w)(t) = \int_{-\infty}^{+\infty} x(u) w(t - u) du$$

In realtà, lavorando con array multidimensionali, usiamo una *sommatoria*, e calcoliamo la convoluzione tra due tensori, uno per l'input del livello e uno (**kernel/feature map**) che è un parametro del livello di convoluzione. In pratica: $(X * K)(i, j) = \sum_a \sum_b X(a, b) K(i - a, j - b)$.

Quindi un *livello convoluzionale* prende in input un tensore X e tramite i parametri calcolati nel training, rappresentati da K , produce in output un tensore H . Questo processo "ingloba" il concetto dei pesi w . Nella pratica si usa la **cross-correlazione**, $(X * K)(i, j) = \sum_a \sum_b K(a, b) X(i + a, j + b)$, non una vera convoluzione. Qui diciamo che sono i parametri che cambiano nel training.

Piccolo esempio:



Vediamo come l'output sia più piccolo. Per evitare ciò si usa il *padding*, ovvero si aggiungono elementi nulli per aumentare la dimensione dell'input, per evitare la perdita dei dati. Noi vogliamo il caso in cui l'output abbia le stesse dimensioni dell'input originale, né più grande, né più piccolo.

Le dimensioni sarebbero: $(input_{righe} + padd_{righe} - kernel_{righe} + 1) \cdot (input_{col} + padd_{col} - kernel_{col} + 1)$ e noi impostiamo $padd_{righe} = kernel_{righe} - 1$ e $padd_{col} = kernel_{col} - 1$

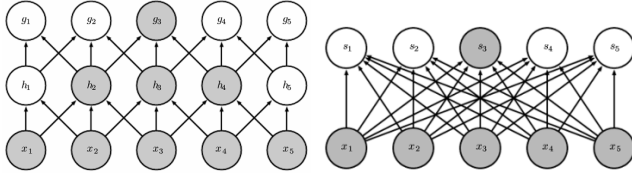
Spesso i Kernel hanno dimensioni dispari, se aggiungiamo un padding di dimensioni pari, evitiamo asimmetrie e centriamo l'input.

Quali sono le caratteristiche alla base della convoluzione?

La prima, **Sparse Interaction**, evita che ogni unità interagisca con *tutti* gli input, ma solo una parte.

Questo viene realizzato proprio grazie al *kernel* che opera su una sotto porzione.

Graficamente, il *receptive field* mostra i collegamenti tra questi. Nel primo caso, *convoluzionale sparse interaction*, anche con meno connessioni rispetto al *fully connected*, abbiamo comunque un'interazione *indiretta* con tutti gli input, utilizzando alla fine gli stessi parametri.



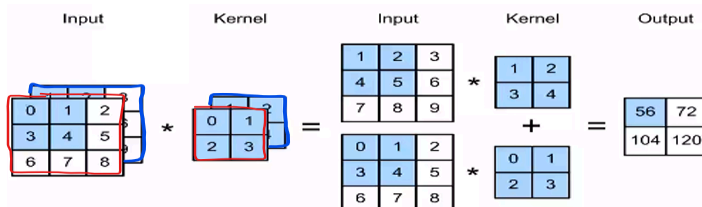
Altra caratteristica è il **Parameter Sharing**, dove, a differenza delle reti Neural Networks che associano pesi diversi a unità diverse, qui nelle CNN il set di *parametri* (e quindi pesi) è condiviso da tutte le unità.

Tutte le reti appena trattate sono robuste rispetto alla traslazione, per altre trasformazioni non vi è certezza. Questo terzo aspetto è detto **Equivariant Representation**, conseguenza del *Parameter Sharing*.

Torniamo al concetto delle immagini, le quali hanno una trattazione leggermente diversa dal resto.

Se si lavora con immagini a colori, queste presentano 3 *canali RGB*, e sono sempre dei tensori (o matrici bidimensionali) associati all'intensità del colore. L'immagine non è più bidimensionale (solo se in bianco e nero) ma l'input sarà "*#channels × #righe × #colonne*". Questo vuol dire che anche il *kernel* deve

essere tridimensionale per eseguire: $H_{ij} = \sum_a \sum_b \sum_c K_{a,b,c} X_{i+a,j+b,c}$, ciò che sto facendo è, per ogni "canale", effettuare la convoluzione tra il singolo Kernel *K* 3D e input *X*, ottenendo un *output bidimensionale*.



Se usassi c_o Kernel diversi, potrei "estrarre" c_o output diversi. Se ragioniamo su un singolo livello, avremo:

In input il tensore *X* di dimensione $n_h \times n_w \times c_i$, posso vedere i vari Kernel come un unico Kernel di dimensioni $k_h \times k_w \times c_i \times c_o$ dove c_i sono i canali in input e c_o quelli in output e ciò che produco è, tramite una funzione di attivazione *non-lineare* (es: ReLU) un *tensore H* di dimensioni $n_h \times n_w \times c_o$.

Questo risultato è dato da: $\varphi(\sum_{u,v,c} K_{u,v,c} X_{i+u,j+v,c} + b_k)$

Fino a ora, lo "scorrimento" è avvenuto di una sola posizione alla volta. Tramite il parametro **Stride** possiamo definire un valore arbitrario per lo spostamento. Questo è utile per ridurre la risoluzione.

Pooling

Spesso i livelli convoluzionali sono seguiti dai *livelli di pooling*, che usano una finestra di dimensione fissata (quindi niente parametri) per generare un singolo valore in output. E' quindi simile alla convoluzione, ma ha il compito di *aggregare* i valori dei pixel, magari per ottenere una media o un massimo. Con più canali, lo applico a ciascun canale separatamente, e ciò che ottengo ha gli stessi canali dell'input.

LeNet e Transfer Learning

LeNet5 fu la prima rete convoluzionale presentata. Usata per riconoscere cifre scritte a mano.

La rete è composta da: una parte convoluzionale, seguita da pooling (riduzione dimensioni), un'altra convoluzione seguita da pooling (ulteriore riduzione) e una parte *dense* composta da tre livelli completamente connessi. La parte "convoluzione+pooling" estrae e comprime le informazioni più importanti, ma queste sono sempre rispetto una "griglia", e non è direttamente usabile per la parte di classificazione. La parte "dense" prende il risultato precedente, che è ancora organizzato spazialmente, e lo "appiattisce" in un vettore lineare di valori, considerabile per operazioni di classificazione.

Successivamente si ha avuto AlexNet e GoogLeNet, dove non c'è bisogno di scegliere la dimensione del Kernel.

Partendo la LeNet ci chiediamo: posso usare un modello già pronto, con dati già analizzati, come "base" per la predizione di un qualcosa di nuovo inerente?

Magari ho un dataset di animali, e voglio aggiungere una specie nuova.

Ciò è alla base della tecnica di **Transfer Learning**:

Non devo cambiare troppo: Si inizia prendendo i layer di un modello precedentemente addestrato, congelo questi livelli e pesi per preservare le informazioni apprese (altrimenti le perderei con il nuovo addestramento), aggiungo i nuovi layer "sopra quelli congelati", addestrandoli sul dataset specifico.

Opzionalmente, sblocco l'interno modello (operazione di *fine-tuning*) e lo riaddestro sui nuovi dati con un tasso di apprendimento basso.

Classificazione Immagini

Possiamo fare di più di una semplice classificazione, in particolare:

Localizzazione: Vogliamo sapere se c'è un oggetto, e dove si trova. Questa seconda richiesta è ritornata mediante ***bounding-box***, identificata da *coordinate* (x,y), *altezza* e *larghezza*. E' quindi una regressione perché tratto valori numerici continui. La modifica da attuare è nella parte finale, dove abbiamo due *livelli completamente connessi*: uno per classificare, uno per il *bounding box*. Per questi problemi è difficile avere molti dati (bounding box già tracciate). Ammesso di averle, si usa l'**errore quadratico medio** (MSE) nell'addestramento di *bounding box* perché misura la differenza tra le coordinate previste e quelle reali. Tuttavia, quando si tratta di valutare la qualità complessiva delle *bounding box* previste, l'**Intersection over Union** (IoU) è una metrica migliore, che calcola il rapporto tra le box (totalmente uguali = 1, disgiunte = 0).

Object Detenction: Qui possiamo avere n oggetti nell'immagine. Un primo approccio basico è **Naive Sliding Window**, in cui addestriamo la CNN per localizzare un singolo oggetto, e poi facciamo scorrere la CNN sull'immagine reale, facendo previsioni ad ogni passaggio. Si associa il *confidence score*, cioè la probabilità che l'immagine contenga l'oggetto classificato. Spesso coincide con la probabilità di appartenere a una certa classe. (Es: classifico al 99% gatto, sono confidente al 99%). Scala poco. E' richiesto post-processamento, ad esempio per togliere le box di cui sono meno sicuro oppure quelle sovrapposte (segnerei lo stesso oggetto più volte).

Altro approccio è **YOLO**, nella quale processiamo l'immagine una volta sola, dividendola in una griglia e predicendo per ogni cella (lavorando quindi in parallelo) un certo numero di *bounding box* e un *vettore di probabilità* per l'appartenenza alle classi. Anche qui c'è il confidence score, calcolato con *Intersection Over Union*. Inoltre, una cella può "consultare" le celle vicine.

Semantic Segmentation: Abbiamo degli oggetti, e vogliamo sapere i *pixel che formano l'oggetto*. Adesso sto quindi classificando i singoli pixel, bisogna essere precisi. L'output finale deve avere le stesse dimensioni dell'input in quanto non posso perdere informazioni. La dimensione dell'output dipende quindi da quella dell'input. Nelle reti CNN, in cui abbiamo alla fine un livello *dense* totalmente connesso, l'output è un

vettore di dimensioni fisse, indipendente dalle dimensioni dell'input. Questo ha senso se facciamo predizioni “globali”, in cui la spazialità non ci interessa. Per esempio, in un compito di classificazione, l'output è un vettore “appiattito” di probabilità che rappresenta le classi. A noi questo non va bene, in quanto abbiamo necessità di predizioni “spaziali” (in quanto dove si trova il pixel è rilevante). Si introduce allora la **Rete completamente convoluzionale FCN**, in cui non abbiamo più la parte “dense” a favore di un altro livello convoluzionale in output. Esso è robusto rispetto alle dimensioni diverse dei vari input. Si parte da detection per farsi un'idea, poi si fa mapping.

Reti Convoluzionali vs Reti Feedforward

	Livelli compl. connessi	Struttura Spaziale	Tipo di input	Operazioni di Pooling
Convoluzionali	NO	SI	2D/3D	SI
Feed Forward	SI	NO	Lineare	NO

RECURRENT NEURAL NETWORKS

Le RNN sono dei modelli in grado di catturare le dinamiche di sequenze di dati tramite *connessioni ricorrenti*. Non abbiamo l'obbligo di andare sempre avanti. Possiamo vederle come delle *Feed Forward* srotolate. Assumiamo un livello di input, un livello nascosto ricorrente e un livello di output.

Nelle sequenze di dati, dovremmo “portarci dietro” tutte le informazioni degli step precedenti a quello attuale, ma ciò si semplifica introducendo un *hidden state* tale che: $P(x_t | x_{t-1}, x_{t-2}, \dots, x_1) \approx P(x_t | h_{t-1})$.

Questa informazione viene processata da *hidden layers*: $h_t = \varphi(W_t x_t + W_h h_{t-1} + b)$, dove vediamo la dipendenza dall'hidden state precedente. Spesso si usa *sigmoid* o *tanh*. Le analisi ottenibili sono diverse: Posso ottenere una sequenza, data una sequenza (come le previsioni), posso ottenere un vettore partendo da una sequenza (dal testo di una canzone al titolo) o anche il viceversa.

L'addestramento avviene mediante l'algoritmo già noto **Backpropagation** (visto nelle Reti Neurali FeedForward), con **la differenza** di considerare il tempo oltre agli archi: se calcolo ∇L_{W_t} dovrò sommarlo a quello del tempo precedente $\nabla L_{W_{t-1}}$.

Mettiamoci in un modello semplice, senza funzione di attivazione né bias:

$h_t = W_x x_t + W_h h_{t-1}$, l'output è $y_t = W_y h_t$ e la Loss Function (sembra J ma è somma gradienti nei vari istanti temporali) $L = \frac{1}{T} \sum_{t=1}^T l(t, y_t)$.

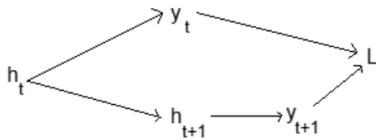
Ora, ad ogni istante possiamo calcolare $\frac{\partial L}{\partial y_t} = \frac{1}{T} \frac{\partial l(t, y_t)}{\partial y_t}$, per quanto detto scomponiamo in W_y e h_T :

$$\frac{\partial \mathcal{L}}{\partial W_y} = \sum_{t=1}^T \frac{\partial \mathcal{L}}{\partial y_t} \frac{\partial y_t}{\partial W_y} = \frac{1}{T} \sum_{t=1}^T \frac{\partial \ell(t, y_t)}{\partial y_t} h_t^\top \quad \text{e} \quad \frac{\partial \mathcal{L}}{\partial h_T} = \frac{\partial \mathcal{L}}{\partial y_T} \frac{\partial y_T}{\partial h_T} = W_y^\top \frac{\partial \mathcal{L}}{\partial y_T}$$

La parte più complessa è per gli istanti temporali precedenti (ossia abbiamo $t < T$):

In questo caso, “scomponiamo” h_t in y_t e h_{t+1} , che scopriremo essere un'operazione iterativa.

Infine, potremo vedere le componenti di h_t ovvero W_x e W_h .



$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}_t} \frac{\partial \mathbf{y}_t}{\partial \mathbf{h}_t} + \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}} \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t} = \mathbf{W}_y^\top \frac{\partial \mathcal{L}}{\partial \mathbf{y}_t} + \mathbf{W}_h^\top \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}} = \mathbf{W}_y^\top \frac{\partial \mathcal{L}}{\partial \mathbf{y}_t} + \mathbf{W}_h^\top \left(\mathbf{W}_y^\top \frac{\partial \mathcal{L}}{\partial \mathbf{y}_{t+1}} + \mathbf{W}_h^\top \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+2}} \right) =$$

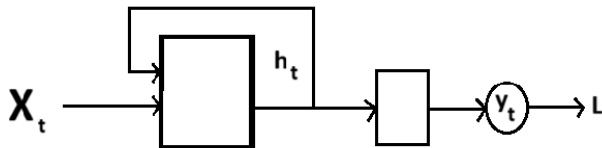
$$= \sum_{i=0}^{T-t} \left(\mathbf{W}_h^\top \right)^i \mathbf{W}_y^\top \frac{\partial \mathcal{L}}{\partial \mathbf{y}_{t+i}}$$

Notiamo come la tipologia di formula si ripeta andando avanti.

Per “tornare indietro” dobbiamo vedere cosa include \mathbf{h}_t , ovvero \mathbf{W}_x e \mathbf{W}_h , che aggiornano nel training:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h} = \sum_{t=1}^T \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_h} = \sum_{t=1}^T \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \mathbf{h}_{t-1}^\top \quad \text{e} \quad \frac{\partial \mathcal{L}}{\partial \mathbf{W}_x} = \sum_{t=1}^T \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_x} = \sum_{t=1}^T \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \mathbf{x}_t^\top$$

Qui è più complesso perché all’inizio L lo esprimo in funzione dell’uscita y_t , ma andando indietro non posso usarla in quanto non tratto più le uscite.



Deep RNN

Se necessitiamo di più espressività, cioè di processare l’input più volte, possiamo mettere in colonna più livelli concorrenti. Orizzontalmente, la rete è la stessa ma per più istanti temporali. Posso avere anche livelli completamente connessi. Se le sequenze sono troppo lunghe possiamo avere il problema di *vanishing and exploding gradients*. Anche avere funzioni che non hanno bound (come ReLu) è un problema. Spesso si usano *sigmoid* o *tanh*.

Un input attraverso reti ricorrenti potrebbe “svanire” andando avanti nell’esecuzione. A volte è un pro (previsioni di una settimana fa per predire oggi) altre è un contro (insieme di parole). Per prevenire il problema del gradiente le RNN hanno una *long-term memory* nei pesi (cambiano poco) e una *short-term memory* negli hidden state (sovrascrivo l’informazione precedente).

La tecnica **Long short term memory LSTM** introduce delle *celle di memoria* nei neuroni. Queste celle memorizzano l’input se deve impattare lo stato interno (*input gate*), la possibilità di resettare lo stato interno (*forget gate*), e se lo stato interno deve impattare l’output (*output gate*). Questi gate sono come livelli completamente connessi. A questo aggiungiamo l’*input node* (quale input considero) e lo *stato interno precedente*.

Per lavorare sulle sequenze posso usare anche *reti convoluzionali*, non per forza *reti ricorrenti*.

PROBABILISTIC MODELS

Sappiamo che il problema di ipotizzare una probabilità dato un campione è affrontato con la massima verosimiglianza. Il nostro obiettivo è quello di trovare il valore θ che la massimizzi, calcolando la derivata e

ponendola a 0. Nel caso di un lancio della moneta $\sum_{i=1}^N -x_i \log(\theta) - (1-x_i) \log(1-\theta)$ troviamo che ciò che

massimizza l'osservazione sono proprio i casi favorevoli rapportati quelli totali.

Parlando di **classificazione** abbiamo due approcci: **discriminator approach**, in cui dati gli esempi individuo un separatore (retta), oppure **generative approach**, in cui modelliamo la distribuzione (concetto assente prima) degli input data la classe, e parliamo di **Bayes Classifier**. Prima mi chiedevo come separare le classi, ora mi chiedo come sono fatte. Caso importante è la **classificazione delle mail spam**, che affrontiamo proprio con il classificatore di Bayes. Il focus è capire se una mail è spam ($c = 1$) o no ($c = 0$). Identifichiamo un *bag-of-words*, cioè una lista di parole tipiche di email spam, in modo da ottenere un vettore binario \mathbf{x} che, per ogni mail, ci dica se una parola è presente o meno.

Noi vogliamo, dato il vettore \mathbf{x} , calcolare la probabilità che una mail sia o meno spam. In altre parole:

$$P(c | \mathbf{x}) = \frac{P(c \cap \mathbf{x})}{P(\mathbf{x})} = \frac{P(\mathbf{x} | c) P(c)}{P(\mathbf{x})}$$

Possiamo identificare una *probabilità a priori* $P(c)$ (es: il 30% delle mail è spam, senza guardare il documento) e una *probabilità a posteriori* (che invece dipende dal documento) $P(\mathbf{x} | c)$, che ci dice la probabilità di avere la parola x_j nel documento data una classe c (es: probabilità di avere "AS Roma" se parlo di calcio).

Il denominatore, è dato dalle *probabilità totali*: $P(\mathbf{x}) = P(\mathbf{x} | c = 0) \cdot P(c = 0) + P(\mathbf{x} | c = 1) \cdot P(c = 1)$, anche se andando avanti non ci servirà, in quanto voglio massimizzare a parità di denominatore.

Possiamo quindi vedere il tutto come: $P(c | \mathbf{x}) = \text{Prob. Posteriori} = \frac{\text{verosimiglianza} \cdot \text{Prob. Priori}}{\text{evidenza}}$

Ci servono dunque $P(\mathbf{x} | c)$ e $P(c)$.

A questo problema ci viene incontro **Naive Bayes**:

Se avessimo la probabilità congiunta $P(c \cap \mathbf{x}) = P(c, x_1, \dots, x_D) = P(c) \cdot P(x_1, \dots, x_D | c)$ potremmo

calcolarla, però avrei una distribuzione congiunta su $D + 1$ parametri binari, ovvero $2^{D+1} - 1$ entries (– 1 perchè l'ultima posso vederla in "funzione" delle altre) e sarebbe costoso.

Bayes assume che le componenti di \mathbf{x} siano *condizionalmente indipendenti dalle classi* (non indipendenti in senso assoluto) ovvero $P(c, x_1, \dots, x_D) = P(c) \cdot P(x_1 | c) \cdot \dots \cdot P(x_D | c)$, passando a $2D + 1$ entries, in quanto ogni probabilità vale 0 o 1, sono D totali, e considero anche c .

Grazie alla verosimiglianza, il **learning** è facile:

$$\begin{aligned} \ell &= \sum_{i=1}^N \log P(c^{(i)}, \mathbf{x}^{(i)}) = \sum_{i=1}^N \log (P(\mathbf{x}^{(i)} | c^{(i)}) P(c^{(i)})) = \sum_{i=1}^N \log \left(P(c^{(i)}) \prod_{j=1}^D P(x_j^{(i)} | c^{(i)}) \right) \\ &= \sum_{i=1}^N \left(\log P(c^{(i)}) + \sum_{j=1}^D \log P(x_j^{(i)} | c^{(i)}) \right) = \sum_{i=1}^N \log P(c^{(i)}) + \sum_{j=1}^D \sum_{i=1}^N \log P(x_j^{(i)} | c^{(i)}) \end{aligned}$$

Questi due termini possono essere affrontati separatamente, ma l'obiettivo è lo stesso.

Per il primo termine, chiamiamo $P(c^{(i)} = 1) = \pi$ e $P(c^{(i)} = 0) = 1 - \pi$ le probabilità a priori (es: mail è spam). Se volessi *compattare la notazione* scrivo: $P(c^{(i)}) = \pi^{c^{(i)}} \cdot (1 - \pi)^{1-c^{(i)}}$, in quanto $c^{(i)}$ è un valore binario e quindi "annulla" solo una delle due componenti alla volta. Si avrà quindi, grazie ai logaritmi:

$$\sum_{i=1}^N \log P(c^{(i)}) = \sum_{i=1}^N c^{(i)} \log \pi + \sum_{i=1}^N (1 - c^{(i)}) \log(1 - \pi)$$

da cui, calcolando la derivata rispetto π posta a 0, troviamo $\theta_{\max} = \frac{\#spam \text{ nel dataset}}{\#dataset \text{ completo}}$.

Per l'altro termine, facciamo una cosa simile: sia $P(x_j = 1|c) = \theta_{jc}$ la probabilità condizionata di una parola data una classe (es: la parola "prezzo" appare in spam). Compattando, si avrà:

$P(x_j^{(i)}|c) = \theta_{jc}^{(x_j^{(i)})} \cdot (1 - \theta_{jc})^{(1-x_j^{(i)})}$ in quanto anche qui $x_j^{(i)}$ è un valore binario.

$$\sum_{i=1}^N \log P(x_j^{(i)}|c^{(i)}) = \sum_{i=1}^N c^{(i)} (x_j^{(i)} \log \theta_{j1} + (1 - x_j^{(i)}) \log(1 - \theta_{j1})) + \sum_{i=1}^N (1 - c^{(i)}) (x_j^{(i)} \log \theta_{j0} + (1 - x_j^{(i)}) \log(1 - \theta_{j0}))$$

Impostando le derivate a 0, tramite verosimiglianza si ha $\theta_{\max j,c} = \frac{\# \text{ volte che parola "j" appare in spam}}{\#spam \text{ nel dataset}}$ per $c = 1$.

Per il **testing** ci chiediamo, data una mail arrivata, come essa deve essere classificata:

$$P(c|\mathbf{x}) = \frac{P(c)P(\mathbf{x}|c)}{\sum_{c'} P(c')P(\mathbf{x}|c')} = \frac{P(c) \prod_{j=1}^D P(x_j|c)}{\sum_{c'} P(c') \prod_{j=1}^D P(x_j|c')}, \text{ come già detto non ci interessa il denominatore, quindi}$$

ci concentriamo sul $h(\mathbf{x}) = \arg \max_c P(c) \prod_{j=1}^D P(x_j|c)$, ovvero per un *input* \mathbf{x} voglio vedere la *classe che massimizza* la probabilità. Applicando la massima verosimiglianza, e generalizzando per k classi:

$$h(\mathbf{x}) = \arg \max_{c_k} \log \left(P(c = c_k) \prod_{j=1}^D P(x_j|c = c_k) \right) = \arg \max_{c_k} \log P(c = c_k) + \sum_{j=1}^D \log P(x_j|c = c_k)$$

L'uso di queste tecniche è sicuramente molto economico, stimo parametri con verosimiglianza e nel testing la regola di Bayes ci aiuta molto. Tuttavia, supporre *l'indipendenza condizionata* non sempre è ottimale. Se un evento non si verifica nei dati forniti, la verosimiglianza gli associa probabilità 0 (il **data sparsity** genera overfitting). La soluzione è **Laplace Smoothing**, in cui aggiungiamo + 1 *ad ogni elemento del conteggio*. Ad esempio $\theta_{ML} = \frac{(N_{testa}+1)}{(N_{testa}+1) + (N_{croce}+1)}$

Contesto più generico è la **classificazione di documenti** in base al testo, qui abbiamo però k classi.

Trattiamo il documento come un insieme di parole \mathbf{x} (ovvero un dizionario di lunghezza D) dove x_j ci dice quante volte è occorsa la parole. L'ordine non conta. Cioè, supponiamo di avere diverse classi c (ad esempio, "sport", "politica"). Per ogni classe c , abbiamo un insieme di parametri $\theta_c = \{\theta_{c1}, \dots, \theta_{c|D|}\}$ dove θ_{cj} rappresenta la probabilità che la parola j appaia in un documento appartenente alla classe c .

La massima verosimiglianza del documento d , caratterizzato dal vettore \mathbf{x} (è come fossero sinonimi) *dato che appartenga alla classe "c"* è:

$$P(d|c) = \frac{(\sum_j x_j)!}{\prod_j x_j!} \prod_j (\theta_{cj})^{x_j} \quad \text{dove:}$$

- $\prod_j (\theta_{cj})^{x_j}$ è la probabilità di vedere esattamente x_j occorrenze della parola j nel documento, dato che appartiene alla classe c . Per ogni classe, osservo j dato che il documento è di classe c .
- $(\sum_j x_j)!$ sono le permutazioni totali delle parole che posso avere.
- $\prod_j x_j!$ normalizza perché non ci interessa l'ordine, soprattutto di quelle che appaiono più volte.

$$P(c|d) = \frac{P(d|c)P(c)}{P(d)} \propto P(d|c)P(c) \propto \prod_j (\theta_{cj})^{x_j} P(c)$$

Come prima, con Bayes:

in cui applichiamo il log:

$$h(d) = \arg \max_c \left(\log P(c) + \sum_{j=1}^{|D|} x_j \log \theta_{jc} \right)$$

Nella fase di **training**, nella pratica:

Calcoliamo il dizionario D sul training, stimiamo la probabilità *prior*, ovvero quanti documenti appartengono a una certa classe rapportati sul totale dei documenti. Stimiamo le *probabilità condizionali* che una generica parola j , appartenente alla classe c , appaia nel documento (e applicando anche *Laplace Smoothing*).

Per un **nuovo documento**, avremo $h(d) = \arg \max_c (\log P^{max}(c) + \sum_j x_j \log \theta_{jc}^{max})$

Alcune ottimizzazioni consistono nel rimuovere parole superflue (stop-words), e usiamo il *termine di frequenza*, che misura l'importanza di un termine nel documento rispetto a quante volte occorre (se *boolean* basta che appaia una volta, se *raw counts* conto le volte che appare). Possiamo normalizzarlo rispetto alla lunghezza del documento (*Normalized Counts*), o dare priorità alle parole che appaiono poco (magari più significative), parlando dunque di *Inverse Document Frequency*.

REINFORCEMENT LEARNING

E' una branca del ML che gestisce *processi decisionali sequenziali*.

Abbiamo un protagonista *agent*, che interagisce con l'*environment* compiendo delle *actions*, ricevendo un *reward*. In ogni momento l'agente può vedere lo stato corrente dell'ambiente (*state*).

L'obiettivo è massimizzare le ricompense/minimizzare i costi. La tecnica più semplice è *try-and-error*, cioè l'agente a forza di provare capisce le decisioni migliori.

Formalmente, per modellare tutto ciò si parla di **Processi Decisionali di Markov MDP**.

Ad ogni step t l'agente si trova in uno $state_t$ e compie un'*action* a_t (tra quelle disponibili) che lo porta in un nuovo stato s_{t+1} (non per forza deterministico), ottenendo un *reward* o penalità.

Nel processo di Markov abbiamo quindi un *insieme di stati* S , un insieme di azioni A (entrambe non sempre finite se lavoro con numeri reali), una *probabilità* $p(s'|s, a) = P[S_{t+1} = s' | S_t = s, A_t = a]$ per la transizione

di stato e una *funzione di ricompensa o costo* $r(s, a)$, che può essere in funzione dello stato attuale

$r(s, a) = E[R_t | S_t = s, A_t = a]$ o dello stato futuro $r(s, a, s') = E[R_t | S_t = s, A_t = a, S_{t+1} = s']$

I processi di Markov ereditano la **proprietà di Markov**, cioè sono indipendenti dal passato.

Dipendiamo solo dallo stato attuale, ovvero $P[S_{t+1} | S_1, \dots, S_t] = P[S_{t+1} | S_t]$

L'**Expected cumulative discounted return** $G_t = R_t + \gamma R_{t+1} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k}$ è il valore atteso partendo

dal tempo t corrente. Noi vogliamo massimizzarlo. Possiamo anche vederla in maniera opposta:

Se $r(s, a) = -c(s, a)$ allora $G_t = C_t + \gamma C_{t+1} + \dots = \sum_{k=0}^{\infty} \gamma^k C_{t+k}$ è da minimizzare.

E' stato introdotto il termine di **discount** $\gamma \in [0, 1)$ perché valutiamo di più scelte corrette fatte all'inizio.

Senza sarebbe un comune *expected return*.

Introduciamo la **policy** π , una funzione o una distribuzione che specifica quale azione un agente dovrebbe eseguire in un determinato stato. Nel caso **probabilistico**, allora $\pi(a|s) = p(A_t = a | S_t = s)$, cioè la

probabilità di compiere l'azione a stando nello stato s . Se **deterministica**, è il mapping $\pi: S \rightarrow A$, ed è anche una politica ottima.

Essa definisce *completamente* il comportamento dell'agente.

Per predire i costi futuri introduciamo una **Value Function**, che dato uno stato e le sue possibili azioni, predice il *valore futuro del reward/cost* in funzione delle azioni. Ad esempio, se nello stato s_1 posso compiere l'azione a_1 che mi porta un costo di 3, e un'azione a_2 che mi porta un costo di 10, scelgo la prima.

Trattasi di una predizione dei costi (valori attesi), non è $r(s, a)$. Differenziamo due value function:

- **Action Value Function/Q Function** $Q_\pi(s, a)$: Rappresenta l'aspettativa della ricompensa cumulativa futura G_t , che l'agente si aspetta partendo da s , eseguendo l'azione a e poi seguendo la politica π .

In formule: $Q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a]$.

Questa ricompensa futura la posso scomporre tra il *costo immediato* C_t e i *costi scontati*

(perché c'è γ^k) *all'infinito* a partire dallo stato S_{t+1} . Ciò che si ottiene è:

$$\begin{aligned} Q_\pi(s, a) &= E_\pi[C_t + \gamma C_{t+1} + \gamma^2 C_{t+2} + \dots | S_t = s, A_t = a] = \\ &= E_\pi[C_t + \gamma(C_{t+1} + \gamma C_{t+2} + \dots | S_t = s, A_t = a)] = \\ &= E_\pi[C_t + \gamma G_{t+1} | S_t = s, A_t = a] = c(s, a) + \gamma E_\pi[G_{t+1} | S_t = s, A_t = a] \end{aligned}$$

Questa può essere rappresentata, tramite l'**equazione di Bellman Ford**, in termini di valori attesi condizionati sugli stati successivi:

$$Q_\pi(s, a) = c(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) \cdot Q_\pi(s', \pi(s')), \text{ ovvero abbiamo il prodotto tra la } \textit{probabilità di passare da } s' \text{ a } s \text{ compiendo } a \text{ e il valore atteso futuro partendo da } s'.$$

- **State Value Function** $V_\pi(s) = E_\pi[G_t | S_t = s]$: Rappresenta l'aspettativa della ricompensa cumulativa futura G_t , che l'agente si aspetta partendo da s e seguendo la politica π . In formule:

$$V_\pi(s) = E_\pi[G_t | S_t = s]$$

Qui possiamo fare esattamente la stessa cosa tramite l'**equazione di Bellman Ford**:

$$\begin{aligned} V_\pi(s, a) &= E_\pi[C_t + \gamma C_{t+1} + \gamma^2 C_{t+2} + \dots | S_t = s] = E_\pi[C_t + \gamma(C_{t+1} + \gamma C_{t+2} + \dots | S_t = s)] = \\ &= E_\pi[C_t + \gamma G_{t+1} | S_t = s] = c(s, a) + \gamma E_\pi[G_{t+1} | S_t = s] = c(s, a) + \gamma \sum_{s' \in S} p(s'|s, \pi(s)) \cdot V_\pi(s') \end{aligned}$$

L'unica differenza è il considerare o meno l'azione a da compiere.

Ovviamente, siamo interessati alle funzioni che calcolo la predizione dei *costi futuri minimi*:

$Q^*_\pi(s, a) = \min_\pi Q_\pi(s, a)$ e $V^*_\pi(s) = \min_\pi V_\pi(s)$, che ovviamente possiamo mettere in **Bellman Ford**:

$$Q^*(s, a) = c(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) \cdot \min_{a'} Q^*(s', a') = c(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) \cdot V^*(s') \text{ dove in pratica}$$

$\min_{a'} Q^*(s', a') = V^*(s')$ perché per trovare il valore ottimale di uno stato, considero tutte le azioni.

Prima di procedere, è utile soffermarci su alcune caratteristiche distintive di algoritmi usabili in questi casi:

- **Ho un modello di riferimento?** Nell'approccio **model-free** non ho modelli, è tutta esperienza, mentre in **model-based** ho un modello (anche parziale) che aumenta la velocità di apprendimento.
- **Calcolo la politica ottima, o cerco di apprenderla interagendo con l'ambiente?**
Nel caso **Value-based** apprendo la *optimal value function* tramite esperienza e poi derivò la politica (es: robot apprende la funzione andando a sbattere o camminando, compiendo azioni).
Nel caso **Policy-based** apprendo direttamente la politica ottimale tramite esperienza, no calcoli. (es: il robot apprende la strategia, non si basa sulle azioni). Esiste anche l'approccio ibrido.
- **Se lavoro con Value Function, l'aggiornamento di Q deve seguire le "stesse regole" usate per scegliere le azioni?**
Negli algoritmi di tipo **off-policy** come aggiornare Q e come scegliere le azioni vengono fatte con policy diverse.
Negli algoritmi di tipo **on-policy** siamo più vincolati, in quanto uso la stessa politica sia per l'azione successiva sia per aggiornare Q.

Nei processi di Markov MDP, spesso non andiamo a *calcolare* la politica ottima, bensì cerco di apprenderla interagendo con l'ambiente.

Policy ottimale

Sappiamo che la policy ottima (quale azione l'agente deve eseguire) è data da $\pi^*(s) = \arg \min_{a \in A} Q^*(s, a)$,

però se non conosciamo $Q^*(s, a)$? Ci aiuta l'algoritmo *model-based* e *value-based* **Value Iteration**.

E' model-based perchè devo conoscere l'ambiente, ovvero il reward sulle transizioni $c(s, a)$, in quanto per ogni stato vedo "dove posso andare" e quale "conviene", per poi migliorare questa valutazione.

Ecco come funziona:

Inizializziamo $Q_i(s, a) = 0$ per tutti gli stati e azioni. Poi, per ogni *stato*, e *per ogni azione*, iterando finché la differenza $Q_i(s, a)$ e $Q_{i+1}(s, a)$ sia infinitesimale (ovvero converga), in particolare usiamo **Bellman Ford**:

$$Q_{i+1}(s, a) \leftarrow c(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) \min_{a' \in A(s')} Q_i(s', a')$$

Il valore futuro è calcolato prendendo in considerazione tutte le possibili transizioni allo stato successivo s' , ponderate per la probabilità di transizione. Alla fine la policy sarà data, per ogni stato, dalle azioni che minimizzano la Q-function. $\pi^*(s) = \operatorname{argmin}_a Q_i(s, a), \forall s \in S$

Un esempio pratico è il **Cloud Auto-Scaling**, in cui vogliamo che il sistema si svegli in dei momenti fissi e, in base al tasso di arrivo discreto λ_i e il numero attuale di macchine VM k_i capisca se l'azione migliore è aggiungere (fino a k^{max}), rimuovere (fino a 1) o non far nulla riguardo le VM. La transizione è modellata da una *probabilità* $p = ((k', \lambda') | (k, \lambda), a] = P[\lambda_{t+1} = \lambda' | \lambda_t = \lambda] \Leftrightarrow k' = k + a$, dove $a = \{-1, 0, +1\}$

La funzione di costo $c(s, a, s')$ è visibile come la somma tra:

- **resource cost** $W_{res} \frac{k+a}{K^{max}}$: costo dovuto all'uso delle risorse, normalizzato.
- **performance** $W_{perf} \frac{1}{\{R(s,a,s') > R^{max}\}}$: Vale 1 (penalizzazione) se supero tempo risposta massimo.
- **reconfiguration** $W_{rcf} \frac{1}{\{a \neq 0\}}$: Evita che l'agenti scali in continuazione senza conseguenze.

W_{res} , W_{perf} e W_{rcf} sono dei coefficienti.

Un algoritmo *value-based* e *model-free* è il **Q-learning**, che molto spesso converge lentamente alla politica ottima. Ad ogni step, ci chiediamo se è meglio *esplorare* (eseguo azioni mai fatte, so di più dell'ambiente), o *exploitation* (uso la conoscenza per l'azione migliore). Il problema è che non posso fare sempre uno o sempre l'altro, e per questo c'è ϵ – *Greedy Exploitation*, dove con probabilità ϵ esploro, e $1 - \epsilon$ scelgo l'azione migliore. La ϵ decade, quindi all'inizio esploro di più. Altrimenti c'è *Softmax Action Selection*, in cui assegno a tutte le azioni una probabilità $\pi(a|s) = \frac{\exp(\frac{Q(s,a)}{\tau})}{\sum_{a' \in A} \exp(\frac{Q(s,a')}{\tau})}$, dove τ è la “temperatura”, se piccolo agisce in maniera greedy, se è grande sceglie casualmente, anche qui c'è decadenza.

Sono in s_1 , compio azione a_1 , e ricevo *reward*, cioè una variabile aleatoria perché l'agente RL non sa quanto sarà, ma comunque sappiamo che è un valore atteso. Diciamo $r_t = 5$, non sappiamo se ci è andata bene. Potremmo finire in svariati stati s_1, \dots, s_n , con una probabilità e stima sul costo futuro. Il Q-learning non sa nulla degli stati futuri, non sa le probabilità. Ha visto solo che è entrato nello stato s_1 , si basa solo su quello, per calcolare la stima in arancione, considera il reward campionato (cioè osservato e non atteso) r_t e il massimo tra le Q partendo dallo stato s_{t+1} (ottenuto grazie alle esperienze) a cui sottrae la Q attuale. Quindi:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \underbrace{\alpha_t}_{\text{Learning Rate}} \left[\underbrace{r_t + \gamma \max_{a' \in A} Q(s_{t+1}, a')}_{\text{Target}} - Q(s_t, a_t) \right]$$

dove nelle parentesi abbiamo il target, ovvero la *stima puntuale del costo futuro in base al reward appena ottenuto r_t e dai costi futuri conseguenti a tale scelta*; a cui sottraggo la stima fatta finora (errore, di quanto si discostano). Quindi stiamo dicendo che, se ci troviamo in s_t e compiamo a_t , otterremo r_t e potenzialmente valutiamo l'azione migliore da fare arrivati nel suo stato s_{t+1} , sottraendo l'errore.

Un esempio di algoritmo *value-based* e *on-policy* è **SARSA**, in cui siamo più vincolati (uso stessa politica sia per l'azione successiva sia per aggiornare Q).

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t [r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

Esempio: Se un robot esplora casualmente e urta (penalità), continuerà comunque con questa politica random, mentre nel Q-learning posso esplorare secondo un approccio e aggiornare Q rispetto un altro.

Deep Recurrent Learning DRL - Deep RL

Con tanti *stati* viene richiesta molta memoria, e si può generalizzare poco. Usare Q-learning con tanti stati lo fa diventare esoso in termini di risorse. Si crea quindi un'approssimazione *parametrica della value function*

$\hat{V}(s, w) \approx V_{\pi}(s)$, che dipende da una variabile di stato e un vettore di parametri w . Con $\hat{V}(s, w)$

approssimazione di $V(s)$, generalizzo e risparmio memoria. Questa funzione da approssimare noi però non la conosciamo, e neanche il vettore dei pesi. Si parla di **Function Approximation** e non di regressione (perché qui ho tutti i dati, nella regressione alcuni devo trovarli).

Un primo approccio per valutare l'approssimazione è minimizzare l'errore quadratico medio su *tutti* gli stati:

$$J(\mathbf{w}) = \sum_{s \in \mathcal{S}} \mu(s) [V_{\pi}(s) - \hat{V}(s, \mathbf{w})]^2$$

dove $\mu(s) \geq 0$ è una distribuzione che riflette l'importanza degli stati (come una frequenza di visite).

Anche qui l'idea è aggiornare i parametri \mathbf{w} usando il gradiente discendente $\mathbf{w} \leftarrow \mathbf{w}_t + \frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w}_t)$, dove però il primo problema è la *somma di tutti gli stati* (magari non servono) e il secondo problema è che dentro $J(\mathbf{w})$ c'è $V_{\pi}(s)$ che noi non abbiamo, non possiamo confrontare con qualcosa che non conosciamo.

La soluzione al problema degli stati è nell'uso dello *stochastic gradient descent*, mentre per la Value Function ignota possiamo mettere una stima U_t al posto del valore vero $V_{\pi}(s)$, parlando quindi di

Stochastic semi-gradient descent:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha [U_t - \hat{V}(s_t, \mathbf{w})] \nabla_{\mathbf{w}} \hat{V}(s_t, \mathbf{w})$$

Nella parentesi quadra, stiamo proporzionando l'aggiornamento del gradiente alla quantità dell'errore, infatti se tale differenza è elevata, l'aggiornamento è maggiore.

L'approssimazione più semplice è $\hat{V}(s, \mathbf{w}) = \mathbf{w}^T \phi(s) = \sum_{i=1}^d w_i \phi_i(s)$ dove abbiamo i pesi $\mathbf{w} \in R^d$ e le feature

$\phi: S \rightarrow R^d$, le quali servono a definire le informazioni che riteniamo importanti a partire dallo stato in cui ci troviamo, applicando $\phi(s)$.

Per definire U_t usiamo il **Q-learning** in cui però aggiorniamo i pesi, non le tabelle.

Un aspetto importante è la *perdita della garanzia di convergenza*.

La soluzione è nell'utilizzo delle reti **Deep** per approssimare la **Value function**, in particolare usando la rete **Deep Q Network**: questa prende in input uno *stato* e in *output* abbiamo per ogni azione (quindi non vale nel contesto continuo) un *valore* $Q(s_t, a_t)$.

Idealmente, avremmo bisogno di molti esempi di interazioni per addestrare la rete neurale in modo efficace, tuttavia l'agente ottiene i reward solo *mentre fa le azioni*, quindi non abbiamo un set di dati.

La soluzione è sfruttare l'interazione con l'ambiente, eseguendo azioni e osservando i risultati. Ogni interazione viene memorizzata come una tupla $\langle s_t, a_t, r_t, s_{t+1} \rangle$ memorizzate in un **Experience Replay**

Buffer FIFO di dimensione fissa B. Durante l'addestramento, si estraggono *casualmente* mini-batch di esperienze dal buffer per aggiornare la rete neurale. Questo ci permette di non dimenticare azioni svolte nel passato.

Altro problema è la *divergenza*, in quanto i *target* e i pesi \mathbf{w} continuano a cambiare.

Usiamo quindi una seconda rete neurale per *stabilizzare gli obiettivi*.

Vediamo adesso la **Policy-based RL**, dove cerchiamo di apprendere *direttamente* la policy, non *value function*. Esse apprendono una policy parametrizzata (differenziabile). Parliamo di *policy gradient*.

$\pi(a|s, \theta) = P(A_t = a | S_t = s, \theta_t = \theta)$, dove $\theta \in R^m$ ha lo stesso ruolo dei parametri \mathbf{w} , quindi si avrà sempre $\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t)$. Seguiamo questa via perché spesso la policy è più semplice da approssimare, e abbiamo convergenza maggiore.

$$\pi(a|s, \theta) = \frac{e^{h(s,a,\theta)}}{\sum_{a'} e^{h(s,a',\theta)}}$$

Una approssimazione della policy può essere la softmax dove all'esponente abbiamo un valore numerico di preferenza per ogni coppia *state-action*. Sostanzialmente, θ si adatta per aumentare le probabilità delle azioni più remunerative.

Questi θ ricordano molto $Q(s,a)$. Tra due azioni, la differenza tra due $Q(s,a)$ può essere minima (perché magari, in un percorso lungo, una singola scelta può essere poco significativa), e viene associata con un parametro detto temperatura per aumentare questo divario. Le preferenze, invece, non necessitano di convergere a specifici valori, ma semplicemente al miglior valore per la policy che stiamo apprendendo.

Task episodici (terminano entro un tempo finale T)

Consideriamo un task episodico e assumiamo di partire da uno stato s_0 e di valutarlo con la metrica

$J(\theta) = V_{\pi_\theta}(s_0)$. Vogliamo aggiornare θ (dipendente da stati, azione, ambiente) per migliorare le prestazioni.

Tale metrica dipende dall'azione scelta e dalla distribuzione degli stati.

Ci viene in aiuto il **Policy Gradient theorem**:

$$\nabla_\theta J(\theta) \propto \sum_s \mu(s) \sum_a Q_\pi(s, a) \nabla_\theta \pi(a|s, \theta)$$

Ovvero abbiamo una proporzionalità rispetto alla *probabilità* $\mu(s)$ di stare in s (e ciò vale \forall stato), in s posso compiere delle azioni a in modo parametrizzato $\pi(a|s, \theta)$ (di cui faccio il gradiente perché sto derivando in θ) e ottenere una ricompensa attesa $Q_\pi(s, a)$ in funzione dello stato e dell'azione.

Abbiamo quindi *proporzionalità rispetto alla lunghezza dell'episodio*. La dimostrazione è la seguente:

$$\begin{aligned} \nabla V_\pi(s) &= \nabla \left[\sum_a \pi(a|s) Q_\pi(s, a) \right] = \sum_a \nabla [\pi(a|s) Q_\pi(s, a)] = \sum_a [\nabla \pi(a|s) Q_\pi(s, a) + \pi(a|s) \nabla Q_\pi(s, a)] = \\ &= \sum_a \left[\nabla \pi(a|s) Q_\pi(s, a) + \pi(a|s) \nabla \sum_{s', r'} p(s', r'|s, a) (r + V_\pi(s')) \right], \end{aligned}$$

soffermiamoci sulla parte in rosso ottenuta tramite Bellman Ford, ovvero il reward, che non dipende da θ (ma solo da r), quindi la derivata rispetto a θ scompare. Poi scomponiamo la sommatoria in s', r' , cioè due sommatorie. Neanche r' è in funzione di θ , quindi tale sommatoria non impatta:

$$\sum_{s'} \sum_{r'} p(s', r'|s, a) V_\pi(s') = \sum_{s'} p(s'|s, a) V_\pi(s') = \sum_a \left[\nabla \pi(a|s) Q_\pi(s, a) + \pi(a|s) \sum_{s'} p(s'|s, a) \nabla V_\pi(s') \right]$$

C'è ricorsività, infatti, riscrivendo:

$$\begin{aligned} \sum_a \left[\nabla \pi(a|s) Q_\pi(s, a) + \pi(a|s) \sum_{s'} p(s'|s, a) \cdot \sum_{a'} \left(\nabla \pi(a'|s') Q_\pi(s', a') + \pi(a'|s') \sum_{s''} p(s''|s', a') \nabla V_\pi(s'') \right) \right] = \\ = \sum_{x \in S} \sum_{k=0}^{\infty} P(s \rightarrow x, k, \pi) \sum_a [\nabla \pi(a|x) Q_\pi(x, a)] \end{aligned}$$

Potremmo continuare a srotolare, ciò ci porta a:

$$\nabla J(\theta) = \nabla V_\pi(s_0) = \sum_s \sum_{k=0}^{\infty} P(s_0 \rightarrow s, k, \pi) \sum_a [\nabla \pi(a|s) Q_\pi(s, a)] = \sum_s \eta(s) \sum_a [\nabla \pi(a|s) Q_\pi(s, a)]$$

dove $\eta(s)$ sono gli step

medi passati in s durante un episodio.

$$= \sum_{s'} \eta(s') \sum_s \frac{\eta(s)}{\sum_{s'} \eta(s')} \sum_a [\nabla \pi(a|s) Q_\pi(s, a)] = \sum_{s'} \eta(s') \sum_s \mu(s) \sum_a [\nabla \pi(a|s) Q_\pi(s, a)]$$

, da cui finalmente:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a [\nabla \pi(a|s) Q_\pi(s, a)]$$

Introduzione all'algoritmo policy-based Reinforce

$\mu(s)$ è la distribuzione *on-policy* degli stati seguendo la politica π , possiamo toglierla mettendo E_π , allora:

$$\nabla_\theta J(\theta) \propto \sum_s \mu(s) \sum_a Q_\pi(s, a) \nabla_\theta \pi(a|s, \theta) = E_\pi \left[\sum_a Q_\pi(s_t, a) \nabla_\theta \pi(a|s_t, \theta) \right]$$

Adesso vorremmo togliere la somma di tutte le azioni:

$$\nabla_{\theta} J(\theta) \propto E_{\pi} \left[\sum_a Q_{\pi}(s_t, a) \nabla_{\theta} \pi(a|s_t, \theta) \right] = E_{\pi} \left[\sum_a \pi(a|s_t, \theta) Q_{\pi}(s_t, a) \frac{\nabla_{\theta} \pi(a|s_t, \theta)}{\pi(a|s_t, \theta)} \right] = E_{\pi} \left[Q_{\pi}(s_t, a_t) \frac{\nabla_{\theta} \pi(a_t|s_t, \theta)}{\pi(a_t|s_t, \theta)} \right] = E_{\pi} \left[G_t \frac{\nabla_{\theta} \pi(a_t|s_t, \theta)}{\pi(a_t|s_t, \theta)} \right]$$

Alla fine, lavorando istante per istante, non serve più Σ , e quindi passo a lavorare con G_t , potendo poi usare lo stochastic gradient descent per ogni istante.

Possiamo quantificare $G_t \forall \text{ steptime}$: $\theta_{t+1} \leftarrow \theta_t + \alpha G_t \nabla_{\theta} \ln \pi(a_t|s_t, \theta)$

L'algoritmo è il seguente, notiamo l'iterazione interna da 0 a T , cioè fino alla fine dell'episodio: Non aggiorniamo i pesi man mano, perchè il ritorno G_t è noto solo a fine episodio, non durante.

```

1 Initialize  $\theta$  (e.g., to 0)
2 Loop
3   generate episode  $s_0, a_0, r_1, s_1, \dots, r_T$  following  $\pi$ 
4   for  $t=0, 1, \dots, T$  do
5      $G_t \leftarrow \sum_{k=t+1}^T \gamma^{k-t-1} r_k$ 
6      $\theta \leftarrow \theta + \alpha \gamma^t G_t \nabla_{\theta} \ln \pi(a_t|s_t, \theta)$ 
7   end
8 EndLoop

```

Esiste anche una generalizzazione, realizzata mediante **baseline** $b(s)$, che per ciascuno stato ci dà una stima del ritorno per quello stato, ovvero “quanto è buono quello stato”. Il teorema *policy gradient* continua a valere, e ci dice che “invece di imparare un valore $Q(s, a)$ per ogni stato e azione, l'algoritmo apprende la differenza tra $Q(s, a)$ e la baseline”, il tutto è espresso tramite:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a (Q(s, a) - b(s)) \nabla \pi(a|s, \theta)$$

La baseline non viene appresa, bensì è un qualcosa che conosco.

Per ridurre l'alta varianza generata dal **Reinforce** abbiamo l'**Actor Critic**:

Anziché utilizzare il valore ritorno G_t , si sfrutta un'altra informazione detta “**one step return** $G_{t:t+1}$ ”: viene preso il reward ottenuto al tempo t e il resto del ritorno lo si stima a partire dalla value function.

$$G_{t:t+1} = r_t + \gamma V(s_{t+1})$$

Quando la value function viene utilizzata in questo modo diciamo che svolge il ruolo di *critic*. Questo approccio viene chiamato **actor-critic**: da un lato stiamo apprendendo una policy che ha il ruolo di *actor* (ci dice come agire) e dall'altro stiamo apprendendo una value function su cui valutiamo il ritorno e che ci dice quanto è “buono” quello che facciamo (*critic*).

Il REINFORCE era usabile solo a fine episodio con tutti i reward. Questo approccio, con singolo reward, mi permette con one-step-return di aggiornare i pesi:

$$\theta_{t+1} = \theta_t + \alpha (G_{t:t+1} - \hat{V}(s_t, \mathbf{w})) \nabla \ln \pi(a_t|s_t, \theta) = \theta_t + \alpha (r_{t+1} + \gamma \hat{V}(s_{t+1}, \mathbf{w}) - \hat{V}(s_t, \mathbf{w})) \nabla \ln \pi(a_t|s_t, \theta)$$

Nell'algoritmo, quindi, andremo a inizializzare due vettori di parametri. Il loop è “infinito” perché dipende da quanto tempo vogliamo dedicare all'addestramento dell'agente. Nel loop andiamo avanti fino a quando l'episodio non termina, ossia fino a quando s non è uno stato terminale. Ecco l'*Actor Critic*:

```

1 Initialize  $\theta$  and  $\mathbf{w}$  (e.g., to 0)
2 Loop
3   Initialize  $s$  as first state of the episode
4    $l \leftarrow 1$ 
5   while  $s$  not terminal do
6     choose action  $a$  according to  $\pi(\cdot|s, \theta)$ 
7     observe  $s'$  and  $r$ 
8      $\delta \leftarrow r + \gamma \hat{V}(s', \mathbf{w}) - \hat{V}(s, \mathbf{w})$ 
9      $\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \nabla \hat{V}(s, \mathbf{w})$ 
10     $\theta \leftarrow \theta + l \alpha \delta \nabla \ln \pi(a|s, \theta)$ 
11     $s \leftarrow s'$ 
12  end
13 EndLoop

```

La grandezza δ (che altro non è che una variabile di appoggio) viene utilizzata per aggiornare entrambi i parametri col metodo del gradiente. Possiamo usare due reti neurali: una per approssimare la politica e una per approssimare la value function, andando poi ad ogni step ad addestrare entrambe le reti.

AUTOENCODERS

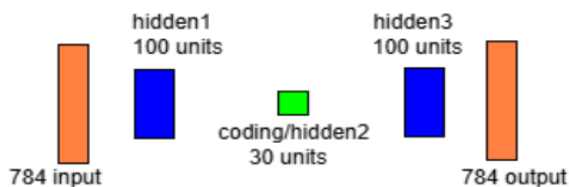
Esse sono *reti neurali unsupervised* addestrate per produrre un vettore \mathbf{x}' simile all'input \mathbf{x} . Internamente, un *hidden layer* \mathbf{h} con poche unità rispetto l'input, ne calcola la codifica, ovvero riassume più informazioni possibili per ricostruire l'input di partenza. Quindi $\mathbf{h} = f(\mathbf{x})$ ci dà una rappresentazione compatta (*encoder*) e poi si ricostruisce $\mathbf{x}' = g(\mathbf{h})$ (*decoder*). Usiamo gli *autoencoders* per avere dimensionalità ridotte, per ridurre del rumore o per individuare anomalie nell'input.

Nel **Simple Autoencoder** abbiamo in sequenza *encoder* e *decoder*, con singolo livello nascosto, e il livello di output del decoder ha lo stesso numero di unità del livello di input dell'encoder.

Per addestrarlo usiamo **reconstruction loss**, ovvero confrontiamo con l'input: $L(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|_2^2$

Nei **Deep autoencoders** abbiamo più livelli nascosti, per apprendere codifiche più complesse. Tuttavia, non dobbiamo puntare a una codifica *perfetta*, che sarebbe inutile sui nuovi dati.

Spesso sono simmetrici con hidden layer centrale. Per ridurre il numero di parametri facciamo in modo che i livelli *specchiati* nell'encoder e decoder siano gli stessi ma trasposti: $W_{N-l+1} = W_l^T$



Un'alternativa è addestrare un autoencoder superficiale (shallow autoencoder) alla volta, un approccio noto come **addestramento strato per strato** o **Stack Autoencoders**

1. **Primo Autoencoder (AE):** Si addestra un autoencoder con solo il primo strato dell'encoder e l'ultimo strato del decoder. Questo AE impara a ricostruire gli input originali.
2. **Successivi Autoencoder:** Le codifiche apprese dal primo AE vengono utilizzate come input per il successivo AE, che coinvolge il secondo strato dell'encoder e il penultimo strato del decoder. Questo AE impara a ricostruire le codifiche, piuttosto che gli input originali.
3. **Ripetizione:** Questo processo viene ripetuto fino a che tutti i livelli dell'autoencoder profondo sono stati addestrati.

Unsupervised Pretraining

Se si hanno molti dati senza etichette e pochi dati etichettati, si può addestrare un *autoencoder* sui dati non etichettati per imparare delle *feature* utili. In pratica, sfrutta la parte *encoding* per apprendere rappresentazioni utili (come orecchie e zampe di animali).

Successivamente, dopo *coding* metto un livello di classificazione per le immagini usando i pochi dati etichettati. Congelando i pesi dell'encoder, si evita l'overfitting sugli esempi etichettati.

Convolutional Autoencoders

Nel caso delle immagini, un autoencoder utilizza un decoder con *convoluzione trasposta* per ricostruire l'immagine originale, aumentando la dimensione dell'output. Lo *stride* in questo caso è inferiore a 1, quindi ogni pixel dell'input contribuisce più volte alla ricostruzione.

Autoencoder undercomplete e overcomplete

Le trattazioni fatte fino ad ora riguardano *autoencoder undercomplete*, i quali hanno una codifica di dimensione inferiori all'input, costringendo il modello a imparare solo le caratteristiche rilevanti per la ricostruzione.

Tuttavia, esistono anche *autoencoder overcomplete* (magari per estrarre caratteristiche più complesse), dove la codifica può essere grande quanto o più dell'input, usando altri vincoli per evitare che il modello semplicemente copi l'input all'output.

Denosing Autoencoders DAE

Aggiungiamo rumore agli input, con l'autoencoder che cerca di ricostruire l'input originale privo di rumore. Il rumore può corrispondere a puro rumore gaussiano aggiunto agli input. Ciò permette all'autoencoder di generalizzare. In alternativa, il rumore può corrispondere agli input messi *off* casualmente (ossia, *dropout*).

Sparse Autoencoders

Gli sparse autoencoders estraggono feature *riducendo il numero di neuroni attivi* nel livello di codifica. Nella pratica, invece di limitare il numero di neuroni, si aggiunge una penalità durante l'addestramento. Un approccio comune è usare una *funzione di attivazione sigmoid* nel livello di codifica e applicare una regolarizzazione *L1* alle attivazioni, non ai pesi, per minimizzare il numero di neuroni attivi (non nulli).

Variational Autoencoders VAE

Sono modelli probabilistici generativi che creano nuove istanze *simili* a quelle del training set, dunque non semplici riproduzioni.

Il VAE non comprime semplicemente l'input, ma genera nuovi dati campionando da una distribuzione probabilistica.

Identifichiamo:

- *Encoder*: Invece di comprimere l'immagine, l'encoder calcola i parametri di una distribuzione probabilistica, tipicamente una Gaussiana, cioè le medie e varianze per ogni unità del livello nascosto.
- *Coding*: Utilizza questi parametri per campionare valori casuali secondo la distribuzione calcolata. In pratica, calcola $z = \mu + \sigma \epsilon$, dove ϵ è un rumore casuale. Questo passaggio consente di fare *backpropagation* e approssimare i dati.
- *Decoder*: Il decoder ricostruisce l'output come in un autoencoder tradizionale, ma partendo da z .

La *Cost Function* tiene conto sia della differenza tra l'oggetto creato e l'input, sia del fatto che il coding debba produrre un qualcosa che segua il campionamento scelto.