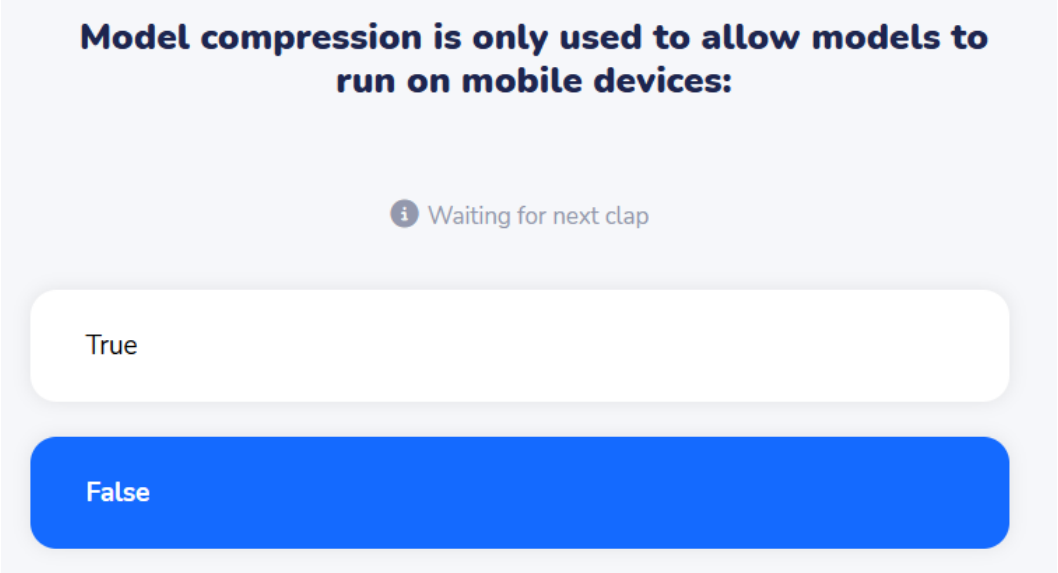
Lezione 10 - RRNs - 29/11/2024

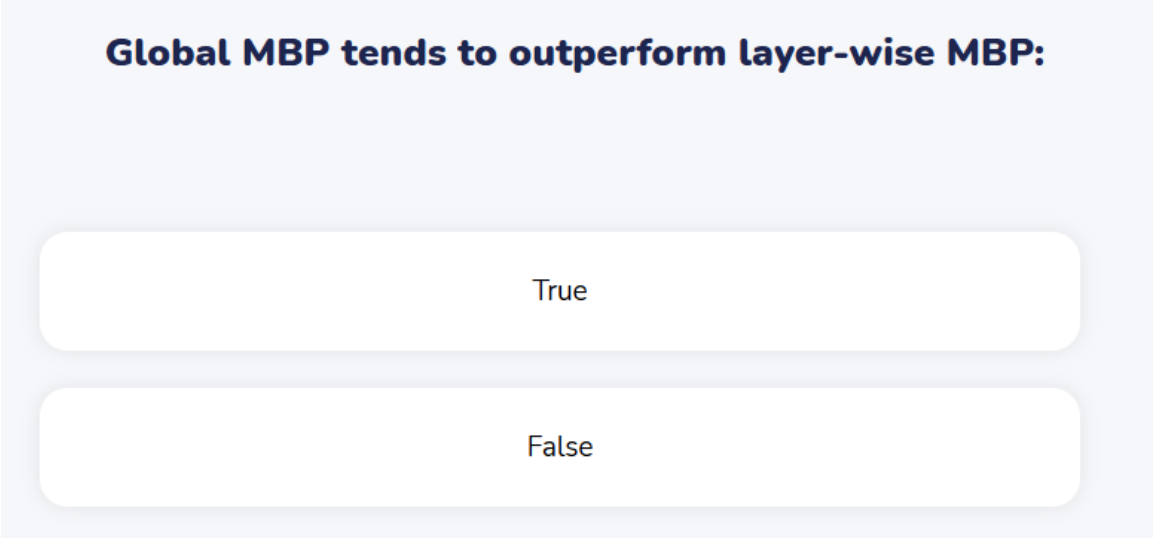


Abbiamo visto che è anche usato per poter fare l'inferenza di modelli molto grandi su pc normali.

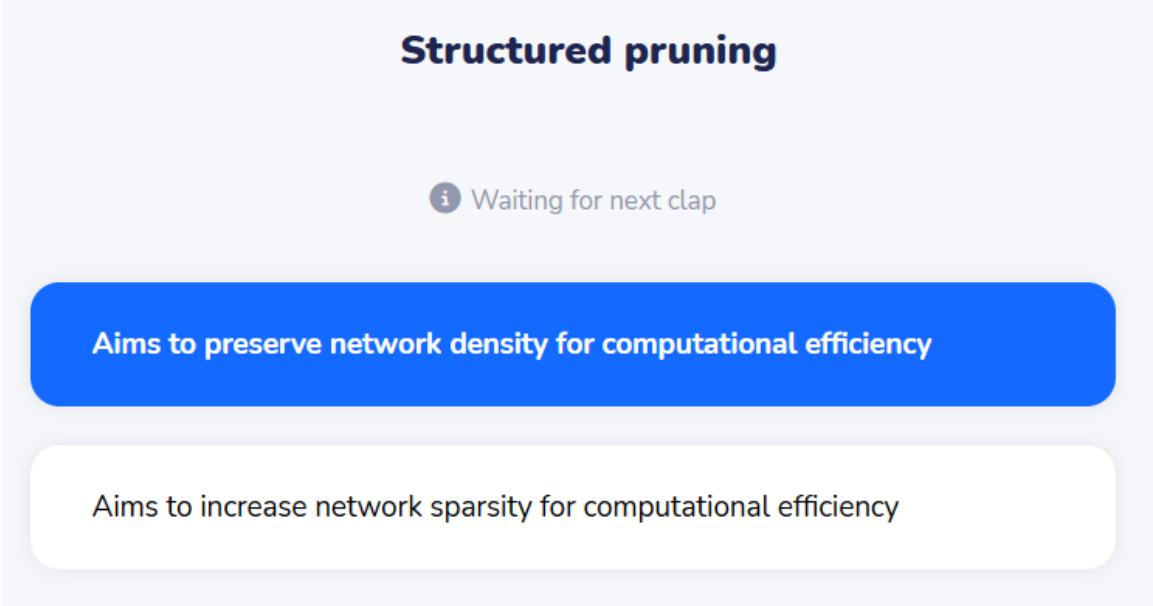
Which is not a model compression technique?

Waiting for next clap

- Weight sharing
- Network pruning
- Low rank matrix decomposition
- **Dropout**
- Knowledge distillation
- Quantization



True, perché a quello globale non interessa dove siano i collegamenti con i pesi più bassi, toglie quella percentuale a prescindere da dove siano, anche se sono tutti sullo stesso layer. Mentre invece quello layer based vuole togliere lo stesso numero da ciascun layer.



Quando facciamo pruning strutturato stiamo togliendo neuroni interi, e questo significa che potremmo ridurre un'intera colonna o riga nella matrice, quindi la matrice rimane densa.

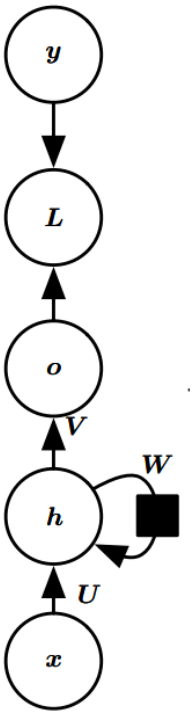
Recurrent and Recursive Networks

Sono chiamate RNNs, sono una famiglia di NN create per processare dati sequenziali. Nello stesso modo in cui abbiamo CNN create per processare dati in un grid regolare, come immagini, abbiamo che le RNNs sono state create per processare dati sequenziali.

Nello stesso modo che le CNN possono scalare immagini che sono molto grandi, e alcune anche se hanno grandezze variabili, le RRNs possono scalare quanto saranno lunghe le sequenze (quindi le sequenze hanno lunghezza variabile).

Le RNNs sono definite con il loro grafo computazionale, usato per calcolare la training loss, e mappare una sequenza di valori x in input, in una sequenza di valori di output.

La loss L misura quanto è diverso il nostro output O rispetto al target y.



- When using softmax outputs, we assume o is the unnormalized log probabilities.
- The loss L internally computes $\hat{y} = softmax(o)$ and compares this to the target y.

Le lettere maiuscole sono le connessioni parametrizzate dalla matrice di pesi. Quindi U mappa l'input nell'attivazione nascosta h, e poi V che mappa verso l'output. E poi abbiamo una mappa W che ha come input e output la connessione nascosta.

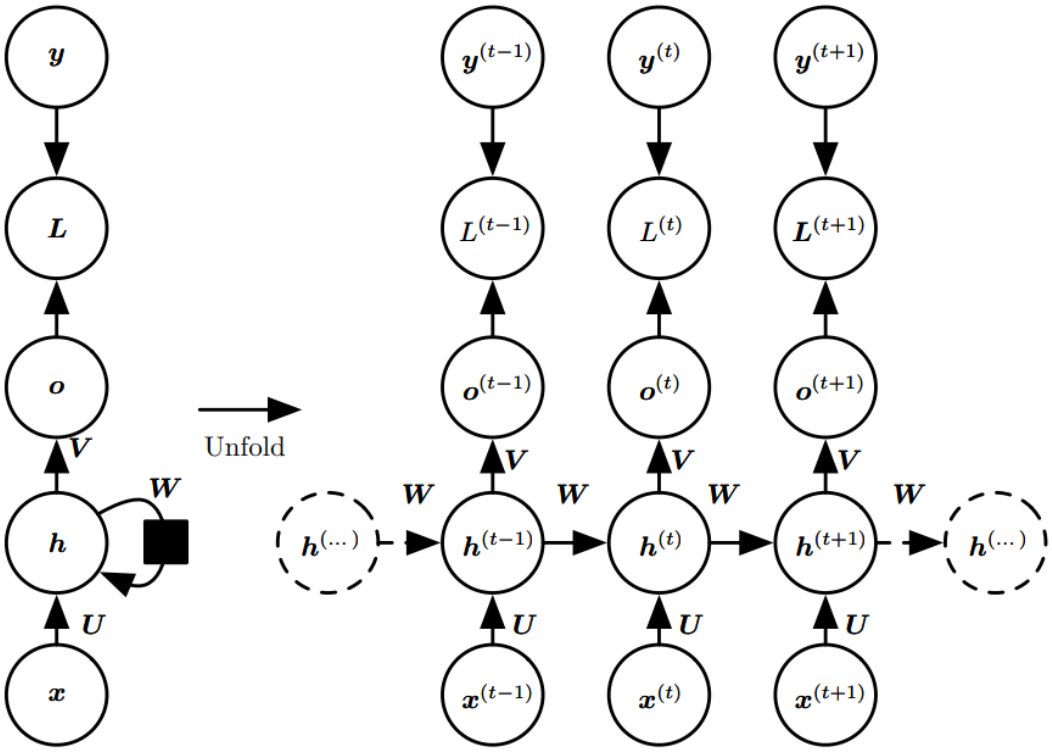
Possiamo vedere le mappe come una fully connected.

Questo è un modo compatto di rappresentare una RRN.

Supponiamo di avere l'input al tempo x t-1, mappo l'input nel layer nascosto, e poi ho dell'informazione che arriva dall'hidden layer del tempo precedente.

Da una parte questo produce l'output, e dall'altro è propagato al prossimo istante di tempo, così che possiamo usare quest'informazione per fare una predizione migliore al prossimo istante di tempo.

Per esempio se dobbiamo predire la temperatura minuto per minuto, possiamo prendere lo storico di cosa è successo ai minuti precedenti.



Quindi l'hidden layer produce un output e manda dell'informazione al prossimo istante di tempo.

Recap:

L'idea di base è quella di usare informazioni sequenziali. In una rete tradizionale feedforward è assunto che tutti gli input e tutti gli output sono indipendenti tra di loro.

Le RNN sono chiamate ricorrenti perché eseguono lo stesso compito per ogni elemento di una sequenza, con l'output che dipende dai calcoli precedenti.

Un altro modo di pensare alle RRNs è quello di immaginare che hanno una sorta di memoria che salva qualcosa dalla computazioni precedenti, che può essere usato per migliorare le prossime predizioni.

Lo stato hidden può essere considerato questa memoria.

L'output al tempo t è calcolato usando solamente la memoria al tempo t, non c'è informazione che arriva dalla prossima istanza nel tempo.

Nelle reti tradizionali feedforward, per ogni layer abbiamo diversi parametri. Nelle RRNs invece condivide gli stessi parametri U,V,W in tutti gli steps, ma con un input

diverso.

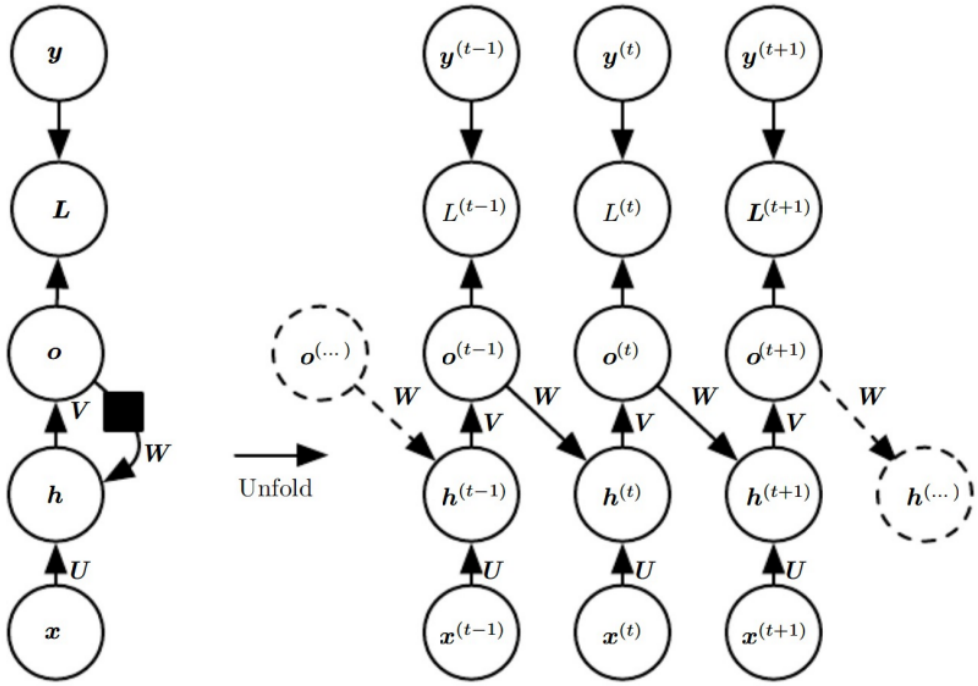
Questo riduce moltissimo il numero di parametri, e permette di processare sequenze che hanno qualsiasi lunghezza, perchè possiamo ripetere l'operazione quante volte vogliamo.

Questa è un'architettura molto semplice, e in particolare con questa stiamo producendo un output a ciascuno time stamp, ma questo potrebbe non essere necessario. Per esempio se sto analizzando un video per capire che tipo di azione sta succedendo, avere una predizione per ogni singolo frame può non avere senso, potremmo voler avere solo un output alla fine del video.

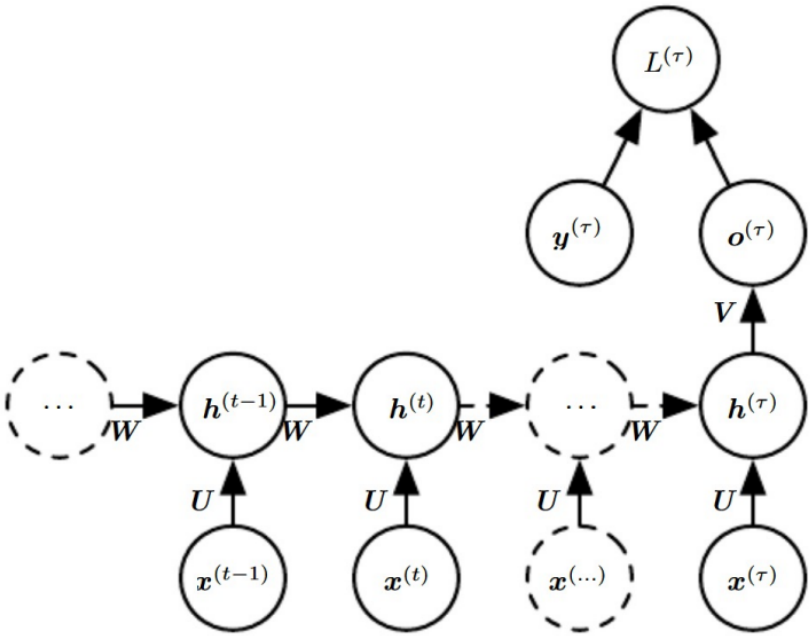
Alcune delle architetture RNN più rilevanti sono:

- RNN che producono un output a ogni passo temporale e hanno connessioni ricorrenti tra le unità nascoste (ad esempio quella che abbiamo visto ora).
- RNN che producono un output a ogni passo temporale e hanno connessioni ricorrenti solo dall'output di un passo temporale alle unità nascoste del passo successivo.
- RNN con connessioni ricorrenti tra le unità nascoste, che analizzano un'intera sequenza e poi producono un singolo output.

Secondo tipo:



Quindi non abbiamo la connessione tra hidden, ma invece è l'output del time stamp precedente che influenza il prossimo.



Qui invece l'informazione arriva dal precedente hidden, ma diamo l'output solo quando arriviamo alla fine della sequenza, che viene confrontato con la ground truth con la loss.

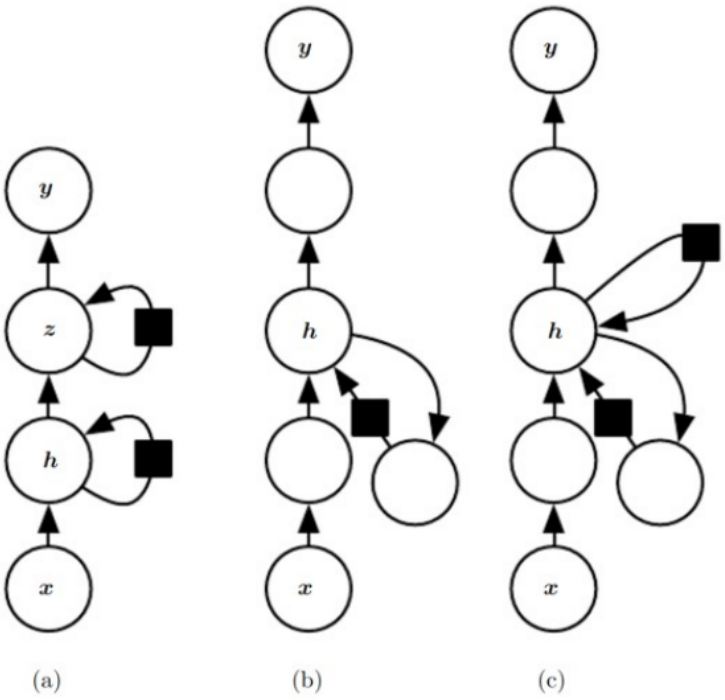
La computazione delle RRNs è scomposta in **3 diversi blocchi**:

- Trasformazioni dall'input all'hidden state (potremmo avere più hidden layers).
- Trasformazioni dal precedente hidden state al prossimo, o più in generale dall'informazione precedente al prossimo hidden state (anche qui potremmo avere più layers)
- Trasformazioni dall'hidden state all'output (anche qui potremmo avere multipli layers).

Abbiamo una matrice associata a ciascun blocco.

Abbiamo un vantaggio, se introduciamo più layers in ciascun blocco? Non ci sono prove, ma da esperimenti si vede che può essere utile per alcuni mappings.

Deep RNNs:



C'è molta flessibilità nel design di queste architetture.

In (a) l'input mappa nell'hidden layer che è connesso a quello al timestamp precedente, che si connette ad un altro hidden layer che è connesso al timestamp precedente. Quindi abbiamo 2 blocchi ricorrenti, aggiungendo un livello di astrazione.

In (b) abbiamo l'input che va in un layer intermedio, prima di andare nell'hidden layer, che è connesso all'hidden layer del timestamp precedente usando un altro layer intermedio. Quindi c'è più profondità prima di arrivare all'hidden layer, ma anche nel collegamento al timestamp precedente, e anche nel layer intermedio che è successivo all'hidden layer.

Questa ricorrenza permette di catturare dipendenze temporali più profonde e complesse. Ogni livello può elaborare sia le informazioni provenienti dal livello sottostante sia quelle derivanti dal passato.

In (c), propaghiamo 2 tipi di informazioni diversa in due percorsi diversi

---

Traning

Come facciamo ad allenare questi modelli?

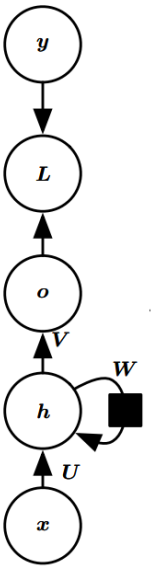
In modo simile ad un normale modello NN. Si usa la backpropagation, con alcune differenze. Visto che i parametri sono condivisi dalla rete in tutti i timesteps, abbiamo che il gradiente a ciascun timestamp dipende anche dal calcolo del time stamp precedente (e quindi ricorsivamente tutti quelli precedenti).

Quindi se vogliamo calcolare il gradiente al time stamp 4, ho bisogno anche i gradienti dei 3 time stamps precedenti.

Per questo motivo, si chiama **Backpropagation Through Time (BPTT)**

Quindi è la stessa backpropagation, con in più le dipendenze dei time stamps precedenti.

BPTT algorithm



- For notation purposes let's use $o^{(t)} = \hat{y}^{(t)}$
- Then consider:

$$h^{(t)} = \tanh\left(Ux^{(t)} + Wh^{(t-1)}\right)$$

$$\hat{y}^{(t)} = softmax\left(Vh^{(t)}\right)$$

- And the cross entropy as the loss (i.e., error) function to minimize:

$$E^{(t)}\left(y^{(t)}, \hat{y}^{(t)}\right) = -y^{(t)} log\left(\hat{y}^{(t)}\right)$$

$$E(y, \hat{y}) = \sum_t E^{(t)}\left(y^{(t)}, \hat{y}^{(t)}\right) = -\sum_t y^{(t)} log\left(\hat{y}^{(t)}\right)$$

La non linearità è data dal tanh dell'input x moltiplicato con la matrice U, e l'informazione del timestamp precedente moltiplicato per la matrice W.

L'attivazione è softmax dell'hidden state per la matrice V (che mappa l'hidden state all'output).

Al tempo t abbiamo che la loss è calcolata tra la groud truth corrente e la predizione corrente (al tempo t). Ma vogliamo minimizzare la loss tra tutte le istanze di tempo, e quindi sommiamo nel tempo la cross-entropy calcolata.

L'obiettivo della backpropagation through time è quello did calcolare i gradienti degli errori su U,V,W e poi di imparare parametri migliori ussando la stochastic gradient descent.

- Just like we sum up the errors, we also sum up the gradients at each time step for one training example:

$$\frac{\partial E}{\partial U} = \sum_t \frac{\partial E^{(t)}}{\partial U}, \quad \text{and} \quad \frac{\partial E}{\partial V} = \sum_t \frac{\partial E^{(t)}}{\partial V}, \quad \text{and} \quad \frac{\partial E}{\partial W} = \sum_t \frac{\partial E^{(t)}}{\partial W}$$

L'errore è la combinazione nel tempo dei gradienti. U non ha t perchè è uguale per tutti gli istanti di tempo.

Per calcolare questo gradiente, la chain-rule standard è usata anche qui (come nei NN normali). Quindi abbiamo i gradienti che tornano indietro nella rete.

Nel nostro caso, abbiamo che il nostro output arriva solo dall'hidden state corrente. Quindi possiamo calcolare il gradiente semplicemente, però in realtà visto che vogliamo calcolarlo anche rispetto a W e U, dove abbiamo che l'informazione arriva dalla sequenza (fino a t=0), il calcolo del gradiente è più difficile.

Exploding & vanishing gradients problem

Calcolare il gradiente rispetto alla prima istanza, ha una ripetizione di calcolo di gradienti. Se abbiamo che molti valori sono più grandi di 1, stiamo moltiplicando qualcosa che diventa sempre più grande, quindi abbiamo il problema dell'**exploding gradient**. Questo è di solito risolto clippando il gradiente, ovvero dandogli un massimo.

- It may happen that we have many values > 1:
  → Exploding gradients

  How can we solve them?
  - Gradient clipping to scale big gradients

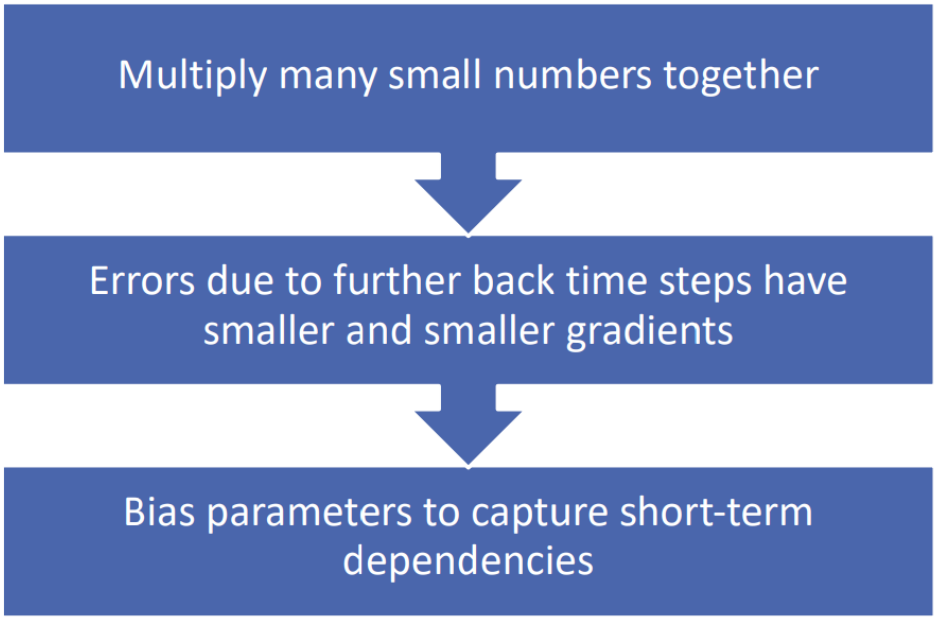
Altrimenti potremmo avere tanti valori minori di 1, quindi la computazione va a 0 molto velocemente, abbiamo il problema dei **vanishing gradient**. Questo è un problema più difficile da diagnosticare, perchè non ci danno errori dal punto di vista numerico.

- It may happen that we have many values < 1:
  → Vanishing gradients

  How can we solve them?
  1. Activation function
  2. Weight initialization
  3. Network architecture

Il modo più efficace di risolverlo è quello di avere un design di un'architettura che previene questo problema.

Vediamo perchè i vanishing gradients sono un problema.



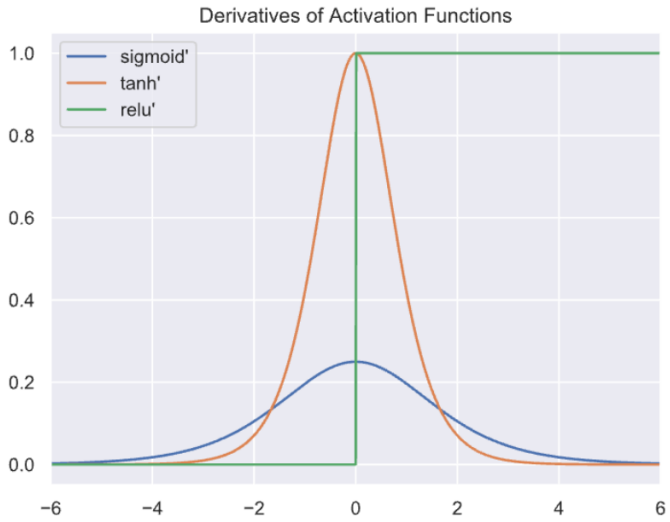
Le RRNs non riescono ad imparare dipendenze a lungo termine, perchè il gradiente diventa quasi 0 man mano che si va indietro nel tempo.

Le RRNs tendono ad essere molto profonde (profonde come la lunghezza della sequenza), il che rende il problema più frequente.

Per esempio in un file audio, che ha 40k samples al secondo, quindi abbiamo 40k layers su cui dobbiamo back-propagare. è un grosso problema.

Vanishing gradient solutions

Soluzione 1: cambiare la funzione di attivazione



ReLU prevents f' to shrink gradients when $x > 0$

ReLU evita che il gradiente si riduca quando abbiamo che l'input è più grande di 1.

Soluzione 2: inizializzazione dei parametri

- Initialize weights to identity matrix
- Initialize biases to zero
- This helps preventing the weigths from shrinking to zero

$$I_n = \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix}$$

Soluzione 3: architettura della rete

Questo è il metodo più usato. Nello stesso modo in cui abbiamo usato le ResNet, anche qui abbiamo un'unità più complessa che ha lo stesso vantaggio della skip connection della ResNet.

Gated RNNs

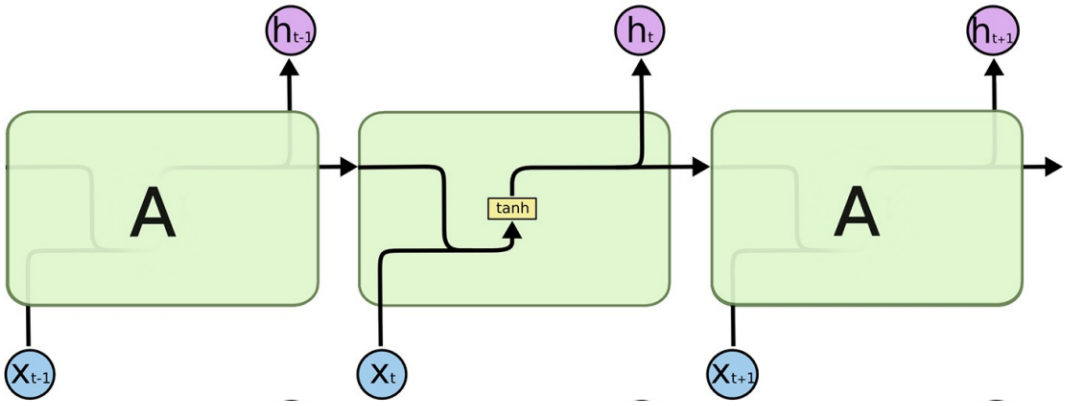
I modelli di sequenza più efficaci utilizzati nelle applicazioni pratiche sono le cosiddette RNN con **gate**. Due modelli famosi sono **Long Short-Term Memory (LSTM)** e **Gated Recurrent Unit (GRU)**.

I Gated RNNs sono basati nell'idea di creare dei path nel tempo che hanno derivate che non hanno problemi di vanishing o exploding.

Per raggiungere questo obiettivo, usano dei pesi di connessione che potrebbero cambiare ad ogni time step.

LSTM Long Short-Term Memory

Esempio: qui vediamo la rete orizzontalmente, il time step precedente, quello corrente, e quello successivo.



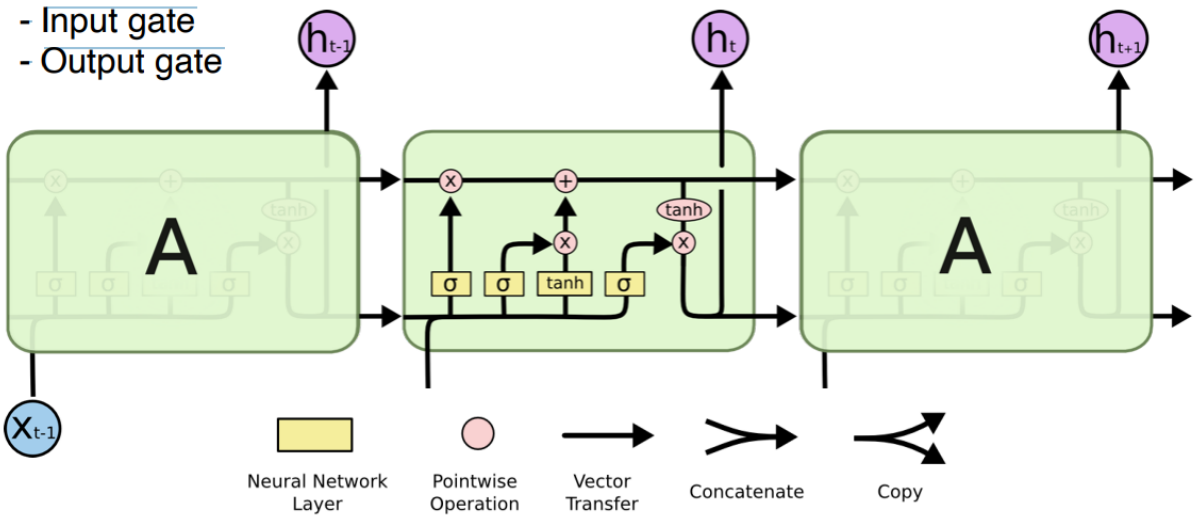
L'input è combinato con informazione che arriva dallo step precedente, per creare l'hidden state al tempo t, che sarà utilizzato (sopra) da altri layers che calcoleranno l'output, e sarà passato al prossimo time step.

Qui vediamo che l'input e l'hidden state precedente vengono combinati con una non linearità.

Modelli LSTM sono organizzati in celle che hanno alcune operazioni. Hanno una variabile di stato interna, che viene passata da una cella all'altra e modificata tramite i seguenti **gates operativi**:

- **Forget gate**
- **Input gate**
- **Output gate**

Quindi abbiamo diversi gates che aggiungono o rimuovono informazione per essere propagata al prossimo gate. I gate cambiano la variabile interna decidendo come estrarre informazioni dall'input e dell'informazione dello step precedente.

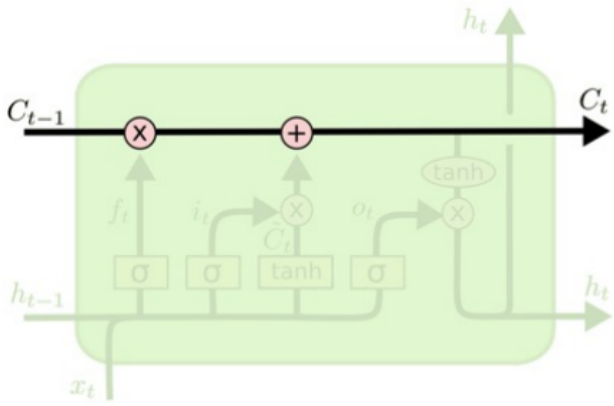


Nell'immagine precedente vediamo la notazione delle componenti.

Cell state

Mantiene un vettore C_t che ha la stessa dimensionalità dell'hidden state h_t.

Le informazioni possono essere aggiunge o rimosse da questo vettore di stato tra il forget gate o l'input gate (il forget gate è il primo, che farà un'operazione di moltiplicazione con la memoria del timestamp precedente, mentre l'input gate è il secondo, che si combina tramite la somma)



Forget Gate

Il forget gate è il primo. é un layer di sigmoide che prende l'hidden state del precedente time step e l'input corrente. Li concatena, e applica una

trasformazione lineare (box giallo) seguita da un'attivazione tramite sigmoide.

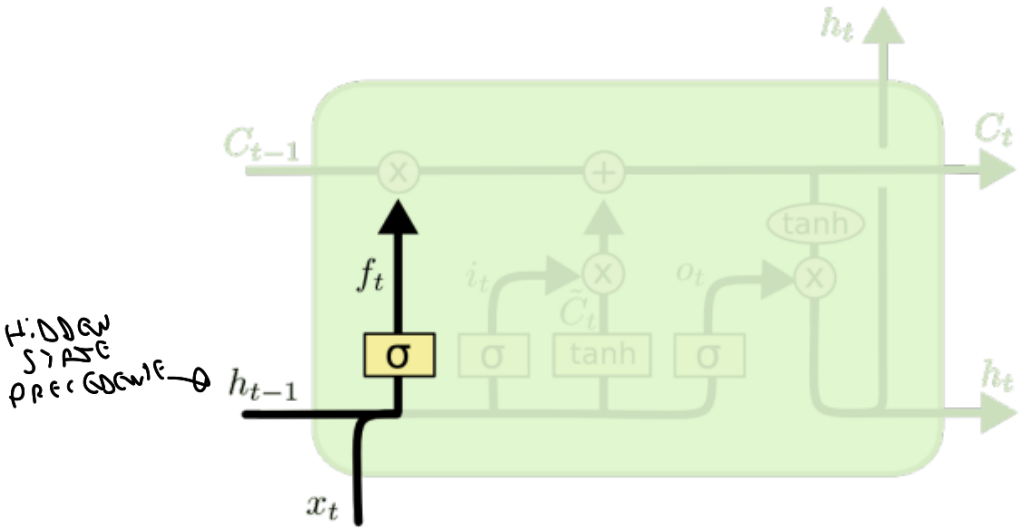
Nella formula abbiamo che la concatenazione (parentesi quadre) dell'hidden state precedente e l'input corrente viene moltiplicata per la matrice W_f (è la matrice dei pesi del forget gate, è appresa durante il training e determina la contribuzione dell'input e dell'hidden state precedente nel forget gate). Poi si somma il bias (anche questo è imparato), e poi tutto viene passato per la funzione di attivazione sigmoide.

$$f^{(t)} = \sigma(W_f[h^{(t-1)}, x^t] + b_f)$$

L'output del forget gate è quindi tra 0 è 1 (grazie alla sigmoide):

- se $f^{(t)}$ è 0, allora lo stato interno del time step precedente C_{t-1} è completamente dimenticato
- se $f^{(t)}$ è 1, allora lo stato interno del time step precedente C_{t-1} passa inalterato.

Questo accade in quell'operazione di moltiplicazione (che però fa parte dell'update state gate).



Input gate

è il secondo gate, decide quali informazioni provenienti dall'ingresso corrente e dal passato (input e hidden state precedente) devono essere conservate nella memoria a lungo termine (cell state C_t)

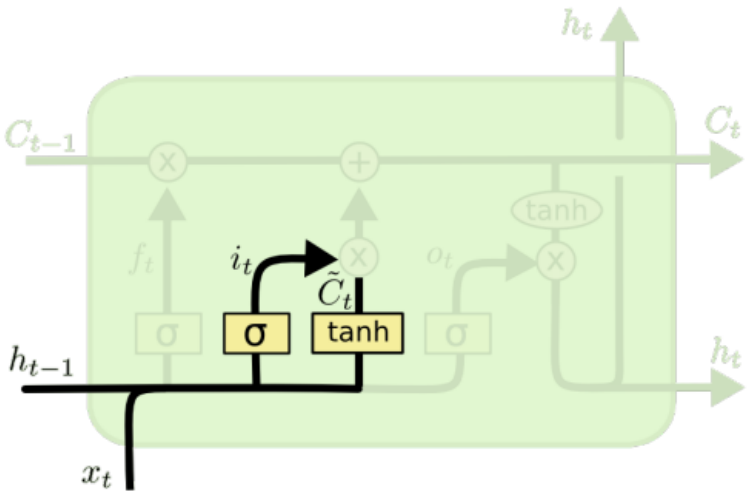
Innanzitutto, determina quali elementi dello stato della cella aggiornare, calcolando una sigmoidale compresa tra 0 e 1. Sottolineo che la matrice W_i è diversa da quella precedente, sono tutte matrici diverse apprese durante il training.

$$i^{(t)} = \sigma(W_i[h^{(t-1)}, x^t] + b_i)$$

Poi, determina quale quantità aggiungere o sottrarre da questi elementi, calcolando la funzione tanh (compresa tra -1 e 1) della concatenazione tra input e hidden state precedente (dopo aver fatto la solita trasformazione lineare con la matrice W_c e il bias). Anche qui la matrice è diversa.

$$\tilde{C}^{(t)} = \tanh(W_C[h^{(t-1)}, x^t] + b_C)$$

C (tilde) rappresenta le nuove informazioni che vengono candidate per entrare nella memoria C_t.



Update the Cell State

Lo stato precedente è moltiplicato con il forget gate, e poi sommato a quella parte del nuovo candidato che è stata fatta passare dall'input gate.

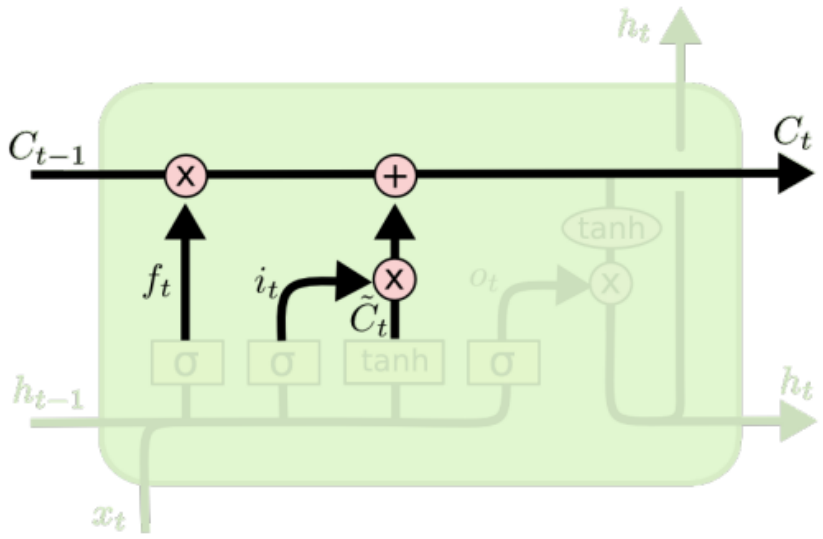
$$C^{(t)} = f^{(t)}C^{(t-1)} + i^{(t)}\tilde{C}^{(t)}$$

Quindi abbiamo una parte che rimuove informazione dello stato precedente, e poi della informazione nuova viene aggiunta, dall'intput.

Nel pezzo che arriva dall'input gate:

- **Effetto della moltiplicazione**:
  - Gli elementi di $\tilde{C}^{(t)}$ vengono **ridimensionati** (o "filtrati") in base ai valori di $i^{(t)}$.
    - Se $i^{(t)} \approx 1$: Le nuove informazioni $\tilde{C}^{(t)}$ vengono mantenute quasi intatte.
    - Se $i^{(t)} \approx 0$: Quelle informazioni vengono quasi ignorate (moltiplicazione per 0).

La somma invece combina la memoria precedente, dopo essere stata modulata dalla forget gate, e le nuove informazioni elaborate e filtrate dalla porta di input.



In altre parole, lo stato della cella viene aggiornato utilizzando una moltiplicazione componente per componente del vettore per "dimenticare" e un'addizione vettoriale per «includere» nuove informazioni.

Output Gate

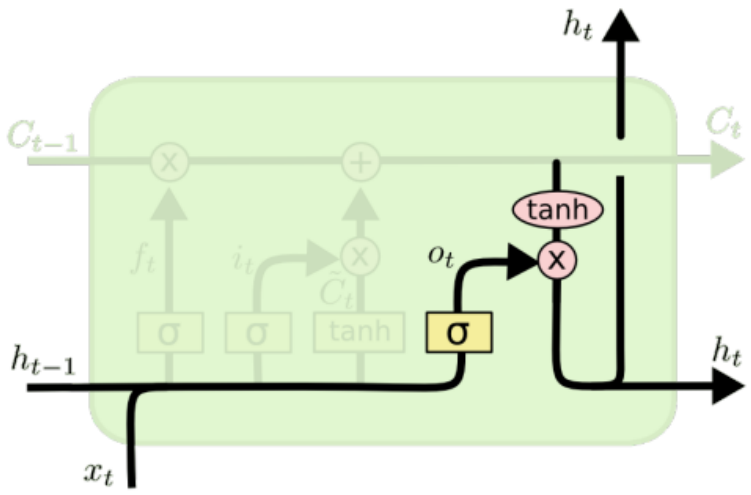
In questo gate lo stato nascosto h_t è aggiornato in base ad una versione "filtrata" dello stato della cella (memoria), che è filtrata tramite tanh.

Quindi prendiamo in input una copia dello cell state che va nel tanh, e poi dopo un'operazione ci darà in output il nuovo hidden state. L'altra copia del cell state viene data così com'è al prossimo time step.

L'output gate calcola una funzione sigmoide sulla concatenazione tra input e il precedente hidden state, per determinare quali elementi dello stato della cella mandare in output.

$$o^{(t)} = \sigma\left(W_o\left[h^{(t-1)}, x^t\right] + b_o\right)$$

$$h^{(t)} = o^{(t)} \tanh C^{(t)}$$



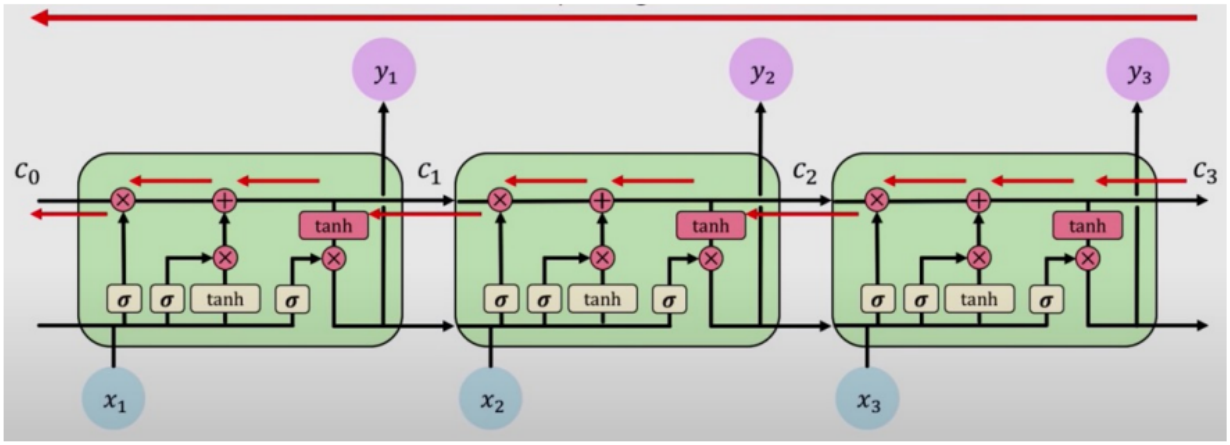
In particolare, abbiamo che l'output della sigmoide è moltiplicato elemento per elemento, per il cell state dopo che è passato per l'operazione tanh.

Tanh serve a comprimere i valori tra -1 e 1, di modo che l'output non diventi troppo grande o instabile, migliorando l'apprendimento del modello. Inoltre, permette di rappresentare relazioni sia **positive** che **negative** nel flusso delle informazioni, e aiuta a prevenire i problemi dell'exploding e vanishing gradient, riducendo il range delle sue attivazioni durante la propagazione.

Concetti principali di LSTM

Usano i gates per controllare il flow dell'informazione (cosa aggiungere, cosa togliere, ...):

- I forget gates tolgono informazioni irrilevanti
- Abbiamo l'operazione di salvataggio dell'informazione rilevante dall'input corrente
- E poi aggiorna selettivamente lo stato della cella.
- Infine, l'output gate ritorna una versione filtrata del cell state.

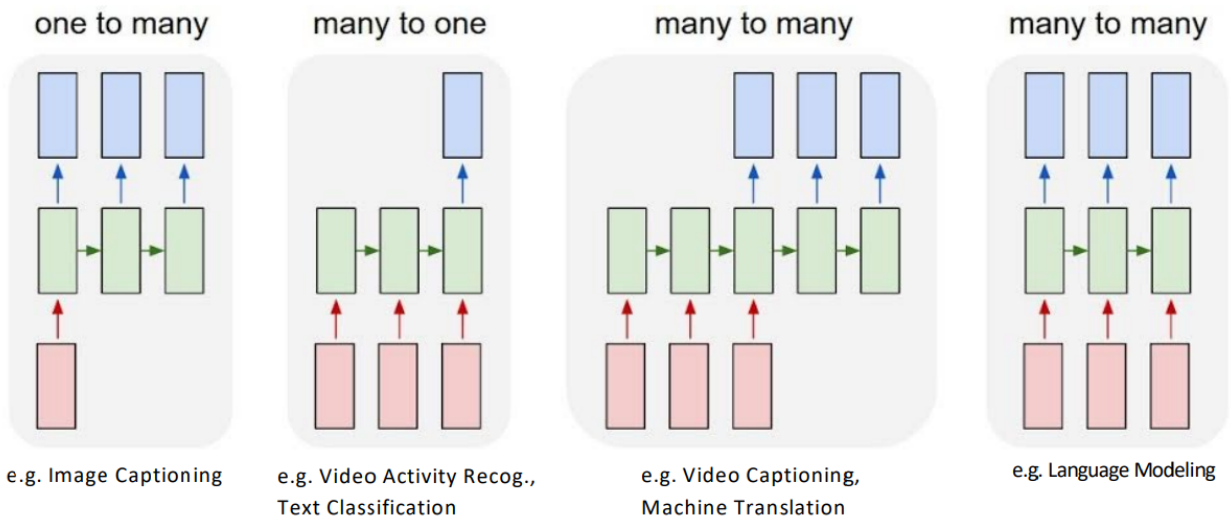


In questo modo la backpropagation nel tempo ha un path che può seguire (nei cell states, frecce rosse) senza cambiamenti. Questa è la stessa idea di ResNet.

Per fare questo, dobbiamo aggiungere la contribuzione dei gradienti che arrivano da tutte le diverse direzioni (i gates sotto).

Diverse architetture

In base a quanti input e quanti output abbiamo, e se abbiamo l'output allineato all'input corrispondente o se abbiamo un delay, abbiamo diverse architetture.



Nel caso **one-to-many** per esempio vogliamo una descrizione testuale (sequenza di parole) che descrivono il contenuto dell'immagine in input.

Nel caso **many-to-one** , potrebbe essere l'input di diverse parole, e l'output del topic. Oppure come input abbiamo i diversi frames del video, e come output abbiamo il label.

Abbiamo diversi tipi di architetture **many-to-many**, in base a se abbiamo lo stesso output in input, oppure se abbiamo un delay, perché abbiamo bisogno di tempo per capire.

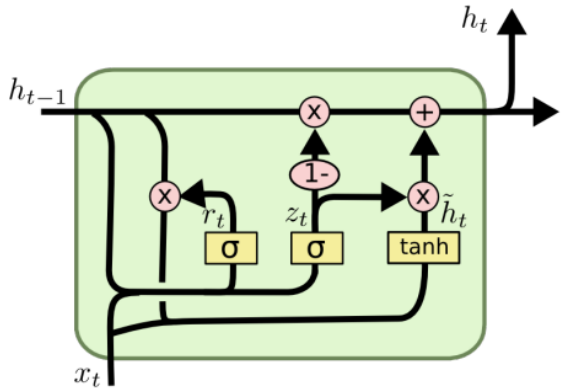
Nella prima many to many per esempio abbiamo bisogno di più frames per capire una palla che si muove nell'immagine in che direzione si sta muovendo. Oppure per la traduzione, di solito non abbiamo un mapping 1-1 tra parole di diverse parole, ma invece in base al contesto le stesse parole possono avere un significato diverso, abbiamo quindi bisogno di più contesto.

Nelle architetture many-to-many dove l'output è allineato all'input. Per esempio sono usati per i language models.

GRU - gated recurrent unit

Questa è la processing cell di GRU. è un'alternativa a LSTM che usa meno gates, perchè combina il forget e input gates nell'**update gate**.

Quest'architettura elimina il vettore cell state, infatti abbiamo un solo output, non 2 righe come prima.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Anche qui abbiamo un modo di back-propagare senza andare in layers di NN.

Grazie al fatto che ha meno gates, ha meno parametri e quindi è più veloce da allenare.

In base alla task, questo metodo potrebbe funzionare meglio o peggio rispetto a quella precedente.