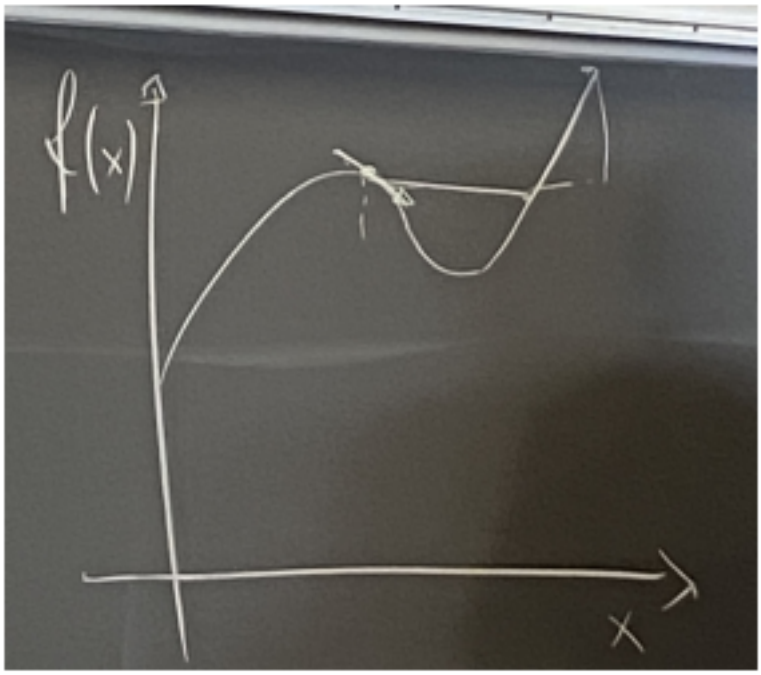
Lez10_ottimizzazione_gradiente

November 10, 2023

1 Lezione 10

1.1 Stochastic Gradient Descent

Posso interrompere quando raggiungo qualche criterio di convergenza, ad esempio non vedo diminuzione loss. Spesso ci fermiamo dopo epoche. In ogni iterazione campiono mini batch di esempi presi nel training set. Presi questi, faccio *forward pass* (calcolo uscite rete e loss, oltre a funzione costo), nella *backward pass* calcolo gradiente G rispetto tutti i parametri della rete. Poi sottraggo un valore pari al gradiente per un learning rate. Voglio diminuire costo, vado in direzione opposta al gradiente, Se mi sposto troppo verso una direzione, non è detto che io migliori. Ad esempio, potrei avere una discesa più rapida andando verso destra, anche se poi il punto di minimo era verso sinistra.



Fino ad ora, il *learning* rate era una costante, anche se nella realtà cambia durante l'esecuzione dell'algoritmo:

Stochastic Gradient Descent (SGD) at epoch k

while *stopping criterion not met* **do**
  Sample minibatch $\mathcal{B}$ randomly from the training set
  $\boldsymbol{g} \leftarrow \frac{1}{|\mathcal{B}|} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha_k \boldsymbol{g}$
end

Si ha α_k, in funzione dell'epoca. Essa dovrebbe decrescere durante l'avanzamneto, più è grande α_k più cambio i parametri. Allora più mi avvicino al minimo locale, meno dovrei cambiare.

1.2 SGD - Learning Rate

Come scelgo α adatto? Innanzitutto, devo rispettare le due condizioni per la **convergenza**:

- $\sum_{k=1}^{\infty} \alpha_k = \infty$
- $\sum_{k=1}^{\infty} \alpha_{2k} < \infty$

Normalmente si sceglie un valore di partenza α_0 (massimo). Scegliamo α_τ (minimo), α decresce fino ad arrivare al tempo τ al valore minimo α_τ

Normalmente $\alpha_\tau = \frac{\alpha_0}{100}$.

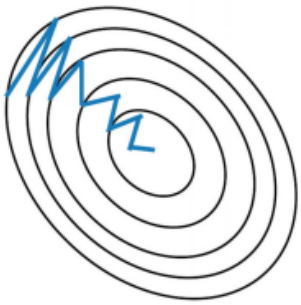
La cosa migliore è vedere come varia la loss durante l'apprendimento.

[ ]:
```
import tensorflow as tf

s = tf.keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=1e-2,
    decay_steps=10000, #tau
    decay_rate=0.9     #tipo esponenziale
)

opt = tf.keras.optimizers.SGD(learning_rate=s)
```

1.3 Momentum

SGD può essere lento. Cerchiamo alternative, tra cui il **Momentum**. Se seguiamo alla lettera SGD, parto da un punto, e tramite mini-batch seguo ciò che ci dice il gradiente. La strada verso il minimo è tortuosa.

Stochastic Gradient Descent **withhout** Momentum



Stochastic Gradient Descent **with** Momentum

Col Momentum teniamo media mobile dei gradienti visti in passato. Se prima sono andato verso sinistra, e ora il gradiente mi dice di andare verso destra, andrò in un certo punto medio. Se invece tendo a muovermi dritto, allora avrò più spinta nel seguire quella direzione. Molto spesso, tale algoritmo è più rapido, converge in meno passi. Il concetto deriva dalla fisica, e non dal bug su FIFA. Un altro iperparametro è $\beta \in [0, 1)$, che ci dice quanto veloce è il contributo.

Stochastic Gradient Descent (SGD) with momentum

while *stopping criterion not met* **do**
  Sample minibatch $\mathcal{B}$ randomly from the training set
  $\boldsymbol{g} \leftarrow \frac{1}{|\mathcal{B}|} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
  Update velocity: $\boldsymbol{v} \leftarrow \beta \boldsymbol{v} - \alpha_k \boldsymbol{g}$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end

Moltiplichiamo la velocità precedente per β, quindi va a diminuire per ogni step. Poi sommo opposto del gradiente (cioè sottraggo) $\alpha_k g$, quindi lo spostamento è una somma pesata tra ciò che mi dice il gradiente, compensato dalla velocità che avevo in precedenza. Quindi posso rallentare dove vado, ma non cambio direzione.

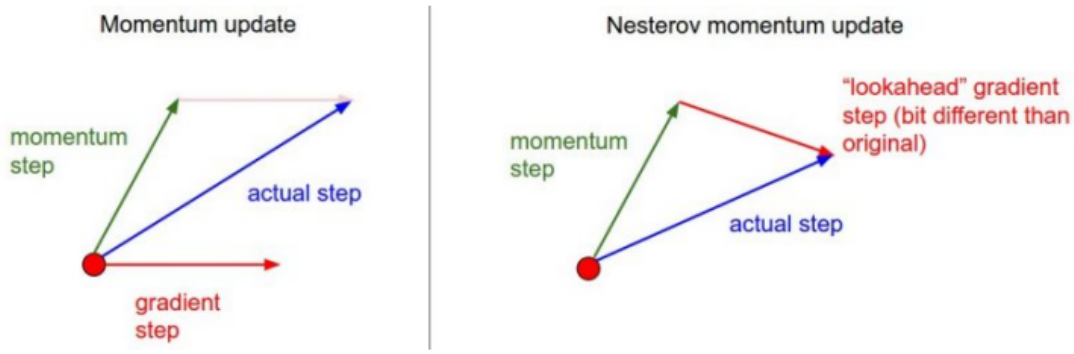
Cambia solo come calcolo la velocità, qui β è fisso, tipicamente $0,5$; $0,9$; $0,99$

Lo passiamo a `keras` specificando:

```
SGD(momentum = beta, ...):
```

1.4 Nesterov Momentum

Nel Momentum, il nostro *actual step*, dipende dalla velocità *momentum step* e da *gradient step*, cioè media delle direzioni.



Nesterov usa il metodo *punta-coda*, quindi prima si muove del momentum, e poi il gradient. Il cambiamento è piccolo, ma empiricamente sembra andare meglio. Questo perchè i gradienti spesso non sono discordanti, quindi muovendomi sto "un passetto avanti".

Nell'algoritmo, cambio subito i parametri, poi calcolo gradiente funzione su questi nuove parametri, e poi il resto è uguale.

Stochastic Gradient Descent (SGD) with Nesterov momentum

while *stopping criterion not met* **do**
  Sample minibatch $\mathcal{B}$ randomly from the training set
  Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \beta \boldsymbol{v}$
  $\boldsymbol{g} \leftarrow \frac{1}{|\mathcal{B}|} \nabla_{\boldsymbol{\theta}} J(\tilde{\boldsymbol{\theta}})$
  Update velocity: $\boldsymbol{v} \leftarrow \beta \boldsymbol{v} - \alpha_k \boldsymbol{g}$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end

Possiamo usarlo con:

```
SGD(momentum=beta, nesterov=True,...):
```

1.5 Ragionamento sul Learning Rate

Il **Learning rate** è un parametro difficile da settare, se lo cambio, cambia il comportamento dell'algoritmo, potrebbe anche non convergere. E' uno strumento delicato, può velocizzarmi o meno. Col Momentum è utile, però introduce β, un altro iperparametro. E se usassi un β per ogni parametro del modello? Quindi una gestione più a grana fine. Non sto aumentando gli iper-parametri? Ovviamente facciamo riferimento ad ottimizzazioni automatiche. Vediamone qualcuno.

1.5.1 AdaGrad

Lo step-size che uso è inversamente proporzionale alla radice quadrata della somma di tutti i valori al quadrato del gradiente. In italiano comprensibile, se un parametro è stato molto aggiornato, ora

so che devo aggiornarlo di meno. Più valori del gradiente sono alti, più è rapida la decrescita.

AdaGrad

$\boldsymbol{r} \leftarrow 0$ /* Gradient accumulation */
while *stopping criterion not met* **do**
  Sample minibatch $\mathcal{B}$ randomly from the training set
  $\boldsymbol{g} \leftarrow \frac{1}{|\mathcal{B}|}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
  $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + -\frac{\alpha}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$
end

Algoritmo Prendo minibatch, calcolo gradiente, lo metto nel vettore r in cui considero il gradiente al quadrato (moltiplico elemento per elemento, non è prodotto tra matrici). Accumulo i quadrati dei gradienti. Poi, non sottraggo semplicemente αg, ma divido per $\sqrt{r}$. Quindi step size diminuisce se r grande. δ è un valore piccolissimo che evita di dividere per 0.

1.5.2 RMSProp

RMSProp

$\boldsymbol{r} \leftarrow 0$ /* Gradient accumulation */
while *stopping criterion not met* **do**
  Sample minibatch $\mathcal{B}$ randomly from the training set
  $\boldsymbol{g} \leftarrow \frac{1}{|\mathcal{B}|}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
  $\boldsymbol{r} \leftarrow \rho\boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + -\frac{\alpha}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$
end

Molto usato, modifica di AdaGrad. L'accumulazione dei gradienti diventa *media mobile esponenziale*, gestendo meglio problemi non convessi. C'è un nuovo iperparametro ρ, ovvero "come valuto i pesi nuovi rispetto a quelli vecchi". Nell'algoritmo infatti moltiplichiamo per $(1-\rho)$, i vecchi gradienti pesano sempre di meno. Questo aiuta a far diventare lo step-size più grande quando necessario.

1.6 RMSProp + Nesterov

RMSProp with Nesterov momentum

$\boldsymbol{r} \leftarrow 0$ /* Gradient accumulation */
while *stopping criterion not met* **do**
  Sample minibatch $\mathcal{B}$ randomly from the training set
  Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \beta \boldsymbol{v}$
  $\boldsymbol{g} \leftarrow \frac{1}{|\mathcal{B}|} \nabla_{\theta} J(\tilde{\boldsymbol{\theta}})$
  $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho) \boldsymbol{g} \odot \boldsymbol{g}$
  $\boldsymbol{v} \leftarrow \beta \boldsymbol{v} - \frac{\alpha}{\sqrt{r+\delta}} \odot \boldsymbol{g}$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end

Cambia poco, dopo aver preso il minibatch, prima aggiorno la velocità (da *Nesterov*), e poi il gradiente lo calcolo su questi nuovi parametri. Accumulo in r (come in RMSProb) e gli aggiornamenti seguono quest'ultimo algoritmo.

1.7 Adam - Adaptive Moments

Adam

$t \leftarrow 0, \boldsymbol{v} \leftarrow 0, \boldsymbol{r} \leftarrow 0$ /* Initialization */
while *stopping criterion not met* **do**
  Sample minibatch $\mathcal{B}$ randomly from the training set
  $\boldsymbol{g} \leftarrow \frac{1}{|\mathcal{B}|} \nabla_{\theta} J(\boldsymbol{\theta})$
  $t \leftarrow t + 1$
  $\boldsymbol{v} \leftarrow \beta_1 \boldsymbol{v} + (1-\beta_1) \boldsymbol{g}$ /* Momentum update */
  $\boldsymbol{r} \leftarrow \beta_2 \boldsymbol{r} + (1-\beta_2) \boldsymbol{g} \odot \boldsymbol{g}$ /* 2nd moment update */
  $\hat{\boldsymbol{v}} \leftarrow \frac{\boldsymbol{v}}{1-\beta_1^t}, \hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\beta_2^t}$ /* Init. bias correction */
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \frac{\hat{\boldsymbol{v}}}{\sqrt{\hat{r}+\delta}}$
end

Variante di *RMSProp+Nesterov*. Oltre a stimare il primo momento del gradiente (*momentum*), stima anche il secondo momento. Si parte ponendo le tre variabili t, v, r a 0. Nella ricorsione, aggiorniamo i *momenti*:

- v prende valore $v \leftarrow \beta_1 v + (1-\beta_1) g$. Prima era solo beta.

- In r mettiamo $r \leftarrow \beta_2 r + (1-\beta_2) g \odot g$, con $\odot$ che indica il prodotto elemento per elemento, e non il prodotto tra matrici.

Poi corregge le stime per eliminare bias di inizializzazione. Infatti, all'inizio ($t = 0$), le tre variabili sono poste a 0, i momenti v ed r risultano piccoli. Quando applichiamo la correzione del bias, ovvero:

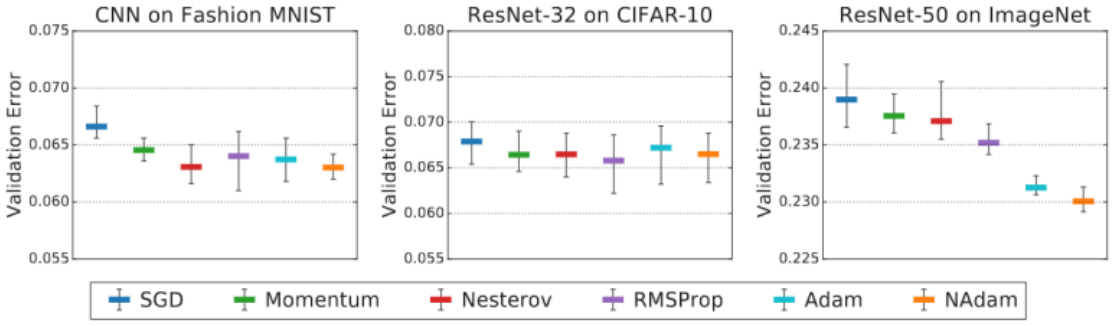
$\bar{v} \leftarrow \frac{v}{1-\beta_1^t}, \bar{r} \leftarrow \frac{r}{1-\beta_2^t}$

Queste variabili (*correzioni dei momenti*) tenderanno a diventare valori grandi, perchè dividiamo per dei valori β^t piccoli. Tuttavia c'è l'esponente t, e quindi andando avanti tra le epoche è come se dividessimo per 1. Questa correzione assicura che i momenti siano inizialmente meno influenti, contribuendo a una migliore stabilità e convergenza dell'ottimizzazione.

Gli altri aggiornamenti sono identici, come vediamo in riga 9. Tipicamente, $\beta 1 = 0.9$ e $\beta 2 = 0.999$, poichè il secondo momento impatta, cerco di essere conservativo.

1.8 Scelta dell'algoritmo

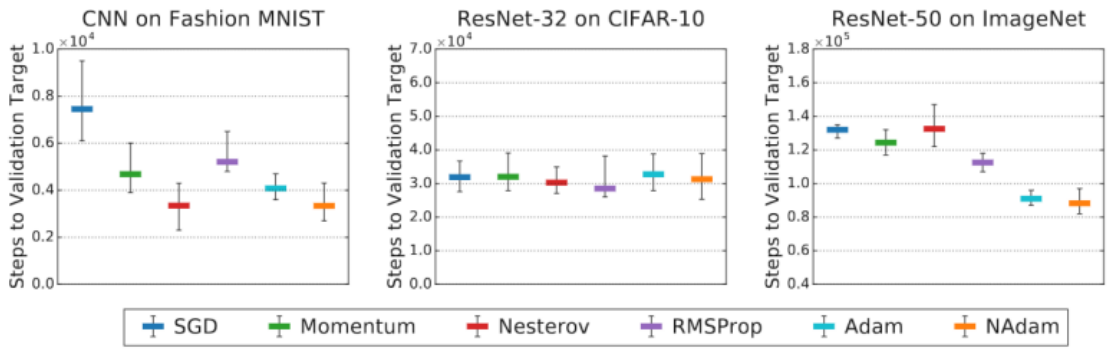
Non abbiamo dimostrato nulla sull'efficienza di questi algoritmi. Però non vogliamo lavorare alla cieca. Non esiste risposta unica. Un criterio importante è la familiarità con l'algoritmo, la quale mi permette di ottimizzare meglio gli iperparametri. Esistono valori suggeriti, ma non sempre sono i migliori.



I grafici (o reti) rappresentano errori sul *validation set*: *nb*: più in basso sono, meglio è.

- Primo grafico: meglio Nesterov, ma più o meno siamo li.
- Secondo grafico: meglio RSProb.
- Terza rete: Adam e NAdam molto meglio.

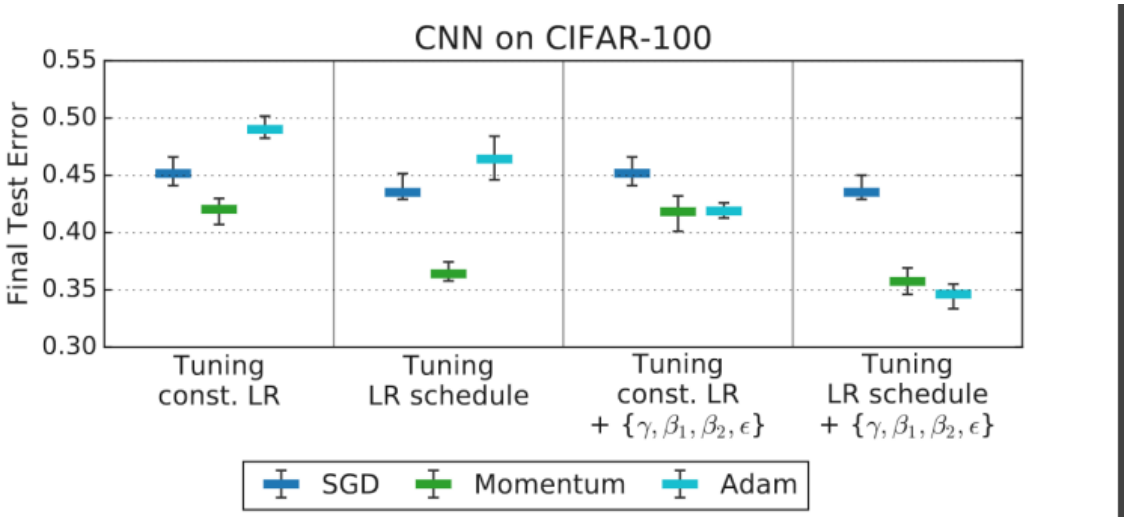
Spesso infatti si usa proprio Adam. E' possibile vedere anche il numero di iterazioni per arrivare a questi risultati. *nb*: più in basso sono, meglio è.



SGD ci mette molto, mentre Adam è più veloce nel convergere. Quindi useremo sempre e per sempre Adam? Ovviamente no!

1.8.1 Algoritmi e iperparametro

La bontà di *Adam* dipende dalla scelta dell'iperparametro.



Nel primo: α/learning rate costante, Adam peggiore.

Nel secondo: α/learning rate varia durante training. Ma non usiamo nè ottimizziamo altri iperparametri. Adam peggiore. Momentum migliore.

Terzo: Adam ha altri iperparametri, se ottimizzati migliora, con α/learning rate costante.

Quarto: Adam con α/learning rate variabile. Ora è il migliore.

1.9 Inizializzazione

Da che parametri parto? come inizializzo? Li metto tutti a 0? Pessima idea, già visto nel Perceptron, sarà tutto 0. Una loro scelta sbagliata potrebbe portare a non convergenza.

Le euristiche sono basate su metodi randomizzati, con distribuzioni ben precise. Tra gli approcci più noti:

- Usiamo distribuzione normale o uniforme per mettere tutti i pesi.
- Usiamo il metodo di **Glorot**, con pesi basati su distribuzione gaussiana, con media e varianza seguenti: $\mu = 0$; $\sigma^2 = \frac{2}{n_{\text{IN}}+n_{\text{OUT}}}$ con n_{IN} ed n_{OUT} che denotano il numero di input ed output nel layer. Più il livello ha interconnessioni, minore sarà la varianza. In `keras`:

  ```
  init = keras.initializers.GlorotNormal()
  layer = keras.layers.Dense(3, kernel_initializer=init)
  ```

- **He** inizialization: cambiamo leggermente la varianza, funziona bene con RELU.

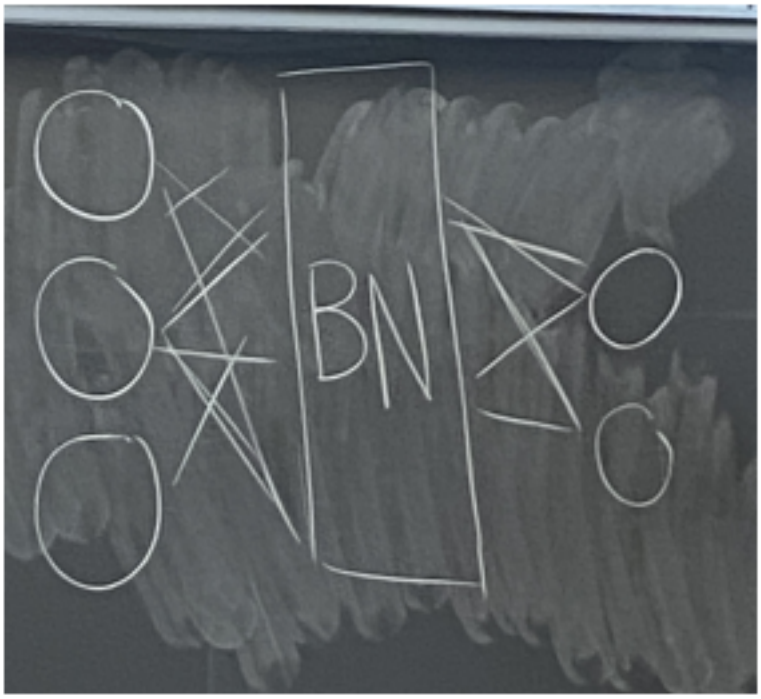
  $\mu = 0$; $\sigma^2 = \frac{2}{n_{\text{IN}}}$ con n_{IN} numero di input nel layer. Questi valori sono livello per livello. In `keras`:

  ```
  init = keras.initializers.HeNormal()
  layer = keras.layers.Dense(3, kernel_initializer=init)
  ```

1.10 Batch Normalization

Tecnica a metà tra ottimizzazione e regolarizzazione. Proposta per migliorare convergenza degli algoritmi che usano reti neurali deep. Regolarizzazione è un effetto secondario. Si applica ai singoli livelli, ma nulla vieta di applicarli a tutti i livelli.

Quando faccio scaling input e lo standardizziamo (es: media = 0 centrata, var = 1), gli algoritmi funzionano meglio, poichè i parametri variano sulla stessa scala di valori. Allora anche i livelli interni della rete potrebbero lavorare meglio con cose standardizzate. Tra livelli, dovremmo standardizzare. Questa è l'idea dietro della *Batch Normalization*.



Considerato un minibatch B, un input $x \in B$, i vettori media $\bar{\mu}_B$ e varianza $\bar{\sigma}_B$, si ha:

$BN(x) = \frac{x-\bar{\mu}_B}{\bar{\sigma}_B}$

1.10.1 Idea: applicare BN agli hidden layers

Sappiamo che $h = \phi(Wx + b)$, con x input livello.

Allora prima di applicare ϕ, applichiamo *Batch Normalization BN*.

Otterremmo quindi $h = \phi(BN(Wx + b))$.

Nella pratica, viene fatto il contrario, cioè $h = BN(\phi(Wx + b))$ (come è stato fatto in figura). Quindi un livello fa le sue cose, produce mini batch di output, che viene normalizzato per diventare mini batch di input.

1.10.2 Capacità espressiva

Un problema nel fare ciò è che ogni livello è limitato nella sua capacità espressiva. Magari mi servono davvero valori non centrati in 0. Si introducono i valori γ e β. Entrambi sono vettori, essendo x vettore.

$BN(x) = \gamma \cdot \frac{x - \bar{\mu}_B}{\bar{\sigma}_B} + \beta$

Che media avrà? La media sarà β, mentre γ influisce sulla varianza.

Cioè ora gli ho ridato "libertà di espressione", allora che senso ha avuto forzarla a 0?

La miglioria è che ora *media* e *varianza* dipendono **solo** da γ e β per ogni livello.

Nel caso base, avrei dovuto toccare tutti i livelli precedenti, e le relazioni sono complesse, quindi se tocco livello 1 non so come influirà sui livelli successivi.

Questa operazione cambia per ogni batch, cambiano μ e σ, mentre γ e β sono parametri della rete. Salverò questi μ e σ a fine addestramento, e li userò nella fase di inferenza (leggasi *predizione*) per nuovi mini batch.

Se addestrassi rete con BN su mini batch di dimensione 1? (Quindi non sto usando mini batch). Che varianza avrei? che media avrei? sè stesso. Quindi avrei tutti 0. Quindi non posso usare BN *senza* mini batch. In `keras`:

```
model = keras.Sequential()
model.add(keras.layers.Dense(16))
model.add(keras.layers.BatchNormalization()) #nuovo livello intermedio
```