Sapienza - Università di Roma

Facoltà di ingegneria

Corso di laurea in ingegneria informatica e automatica

Algoritmo di ottimizzazione per reti neurali su sistemi distribuiti

Relatore: Maria Fiora Pirri

Presentato da: Riccardo Viviano

Contents

1 Introduzione

Sin dell'avvento di nuovi algoritmi di machine learning che fanno riferimento alla branca del deep learning il principale ostacolo incontrato è la limitatezza computazionale dei dispositivi sui quali si fanno girare tali algoritmi. Questo problema è stato affrontato più volte da esperti del settore [1] e si è cercato di risolverlo affidandosi ad hardware molto potente come GPU e TPU e parallelizzando l'allenamento delle reti neurali usando batch di dimensioni elevate quando l'allenamento viene affidato allo stochastic gradient descent (SGD) [2]. Tuttavia questo metodo potrebbe non essere sufficiente a velocizzare il "training" degli algoritmi di deep learning nel caso in cui il dataset sui cui si sta allenando la rete neurale o la rete neurale stessa siano piuttosto grandi. Per ovviare a questa difficoltà i ricercatori hanno trovato diverse soluzioni per rendere questo processo più veloce usando hardware parallelizzabile e cluster di devices. Usando diverse tecniche di ottimizzazione come distributed Newton method [3] o semplicemente usando l' SGD in un cluster di nodi e facendo successivamente la media dei pesi [4]. Questa tesi si concentrerà sulla parellelizzazione del *training* usando un cluster di

nodi e un nuovo tipo di algoritmo di ottimizzazione creato appositamente per un sistema distribuito.

2 Deep learning

Il deep learning è una branca del machine learning che studia algoritmi e modelli ispirati alla struttura e al funzionamento del cervello, e che prendono il nome di artificial neural network (ANNs) o in italiano, reti neurali artificiali. La capacità di ottimizzazione di queste strutture è incomparabile con modelli classici di machine learning quando si ha a disposizione un' enorme quantità di dati. Yoshua Bengio, uno dei principali esponenti della materia, parla del deep learning in termini della sorprendente capacità degli algoritmi di trovare e imparare le correlazioni attraverso il *feature learning* (apprendimento delle proprietà)[14].

2.1 Reti Neurali

Le ANNs sono il cuore pulsante del deep learning e sono strutture o modelli matematici che si ispirano alla biologia e utilizzano metodologie di elaborazione dei dati analoghe a quelle dei neuroni: Le reti neruali artificiali imparano dalle esperienze, le generalizzano a nuovi input e sono in grado di prendere decisioni. Le Artificial Neural Networks sono composte da neuroni, i neuroni artificiali imitano il comportamento dei neuroni bioligici. Essi infatti ricevono in input un segnale lo rielaborano e inviano ad altri neuroni un diverso tipo di segnale. Strutturalmente parlando le ANNs sono composte da tre differenti Layers (o livelli): l'input layer, gli hidden layers (possono essere uno o più) e l'output layer.

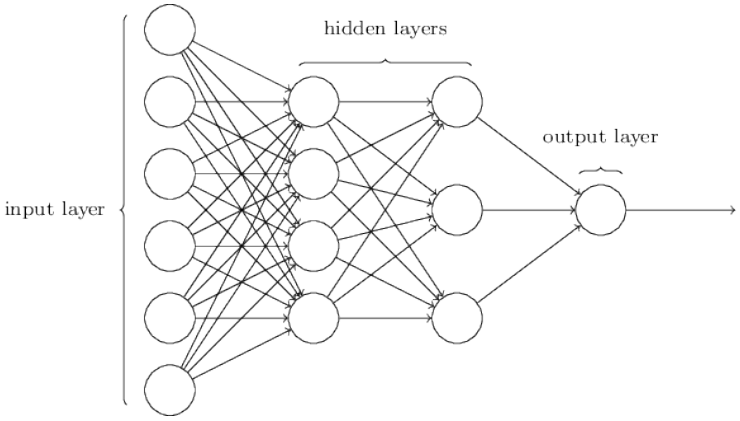

Figure 1: Modello fully-connected di rete neurale

In un modello fully-connected ogni neurone j di ciascun livello l è collegato tramite un peso o weight $w_{i,j}$ a ciascun neurone i del layer $l-1$. Una rete neurale può essere vista come una funzione matematica che mappa le variabili $x_1, x_2, ..., x_n$ corrispondenti ai neuroni di input nella variabile di output y corrispondente al segnale d'uscita del/i neurone/i di output. Il processo che mappa le varibili di input nella/e variabile/i di output si chiama feed-forward. Nell'architettura appena descritta ciascun neurone d' output j di ciascun layer l durante il feed-forward computa le seguenti equazioni:

$$z_j = \sum_{(i)} x_i \cdot w_{i,j} + b_j$$

$$a_j = f(z_j)$$

Dove x_i è l'input i e $w_{i,j}$ è il peso che connette l'input i con l'output j. b_j è il bias del neurone d' output j e $f(z_j)$ è una funzione di attivazione non lineare. a_j corrisponde al segnale di uscita del neurone j che verrà trattato come input x_i dai neuroni del layer successivo.

Insieme al modello fully-connected una delle architetture neurali più utilizzate e più efficienti grazie alla capacità di "estrarre proprietà locali" [15] sono le convolutional neural network (CNNs).
A differenza del modello fully-connected nelle convolutional neural networks i pesi e bias sono condivisi tra i neuroni del layer l. Le CNNs sono responsabili per l'individuazione di importanti features e possono essere viste come la semplice applicazione di un filtro bidimensionale o tridimensionale ad un determinato input [16]. Gli stessi neuroni vengono disposti in una matrice, o nel caso venissero applicati più filtri o kernel, in un tensore.

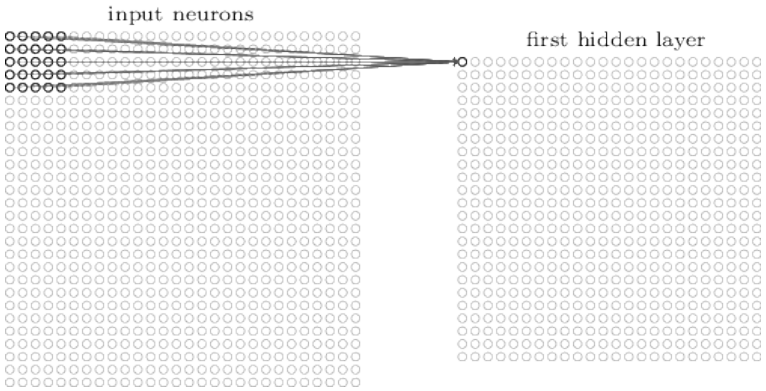

Figure 2: Applicazione convoluzionale di un layer l in una neural network

In una layer convoluzionale l ciascun neurone d'output j, k durante il feed-forward computa

le seguenti equazioni:

$$z_{j,k} = b + \sum_{n}^{N} \sum_{m}^{M} x_{j+n,k+m} \cdot w_{n,m}$$

$$a_{j,k} = f(z_{j,k})$$

Dove N x M è la dimensione del kernel considerato, $x_{j+n,k+m}$ è l'input e $w_{n,m}$ è il peso del kernel. $f(z_{j,k})$ è una funzione di attivazione non lineare. $a_{j,k}$ corrisponde al segnale di uscita del neurone j, k che verrà trattato come input dai neuroni del layer successivo.

2.2 Loss function e Gradient descent

Come abbiamo già accennato, una rete neurale può essere vista come una funzione che mappa le variabili di input $x_1, x_2, ..., x_n$ nella variabile o più variabili di output y. Possiamo definire formalmente questa funzione di mapping come:

$$y = y(v, x)$$

Dove $x \in R^n$ sono gli input n dimensionali, v corrisponde ai pesi e bias all'interno della rete neurale e $y(v, x)$ è il processo di feed-forward precedentemente menzionato. $y \in R^d$ è l'output d'uscita della neural network.
I pesi e bias all'interno della rete vengono considerati parametri adattabili. Durante il *training* questi parametri assumeranno valori diversi al fine di approssimare l'output y ad un certo valore t definito *target value*.
Per fare ciò viene definita una funzione d'errore anche chiamata *Loss function* che determina l'errore dell'output y rispetto al valore del *target value*.

La scelta della *Loss function* è puramente arbitraria, generalmente per problemi di regressione viene utilizzata la MSE (*Mean squared error*) e per problemi di classificazione la *Cross Entropy*.
Immaginando di avere un dataset D di dimensione N in cui associamo ad ogni input x_i un *target value* t_i possiamo definire l'errore complessivo in questa maniera:

$$L = \frac{1}{2} \sum_{i}^{N} (y(v, x_i) - t_i)^2 \qquad \text{(MSE)}$$

$$L = -\sum_{i}^{N} t_i log(y(v, x_i)) + (1 - t_i) log(1 - y(v, x_i)) \qquad \text{(Cross Entropy Loss)}$$

Dove L è la *Loss function*.

L'obiettivo da realizzare è quello di trovare i valori ottimali v^* tramite un processo definito *training* che minimizzano la *Loss*.
L' errore può essere visto come un' ipersuperficie che ci aspettiamo sia una funzione convessa, pertanto per trovare un minimo sulla superficie è necessario fare le opportune derivazioni. A

questo fine viene utilizzato l' algoritmo *Gradient descent* tramite l'uso della *Back propagation* [6]. Per ogni timestep t con la *Back propagation* si calcolano le derivate parziali dei parametri adattibili rispetto alla *Loss* calcolata per poi applicare la formula del *Gradient descent*:

$$v_t = v_{t-1} - a\frac{\delta L}{\delta v_t}$$

$$v_{t+1} = v_t - a\frac{\delta L}{\delta v_{t+1}}$$

...

Dove a è definito come *learning rate* e solitamente assume un valore compreso tra 0 e 1. Il *learning rate* definisce la velocità del cambiamento di valore per ogni parametro v. Un *learning rate* troppo piccolo può causare un *training* poco efficiente, uno troppo grande può provocare una mancata soluzione ottimale per i parametri v.

L'intuizione di base del *Gradient descent* sta nel provare a raggiungere il minimo globale della funzione d'errore L attraverso step successivi t, $t+1$, ..., $t+n$. Di seguito è presentata la dimostrazione della sua correttezza.
Poniamo di avere una funzione d'errore differenziabile su due variabili.

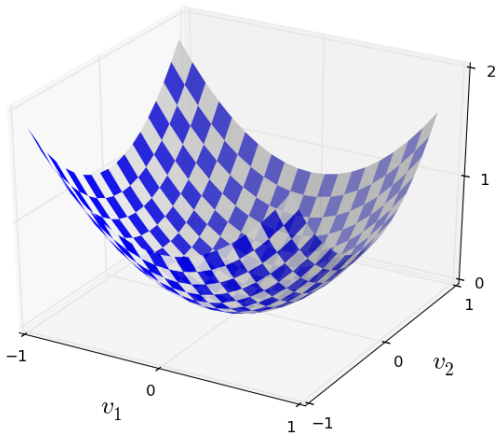

Figure 3: Funzione d'errore a due variabili

Il minimo globale di questa funzione si trova nel punto $v1, v2$. Immaginiamo ora di trovarci su un punto qualsiasi della funzione e di volerci spostare di un valore $\Delta v1$ in direzione $v1$ e di un valore $\Delta v2$ in direzione $v2$. In tal caso possiamo approssimare la differenza della funzione d' errore come:

$$\Delta L \approx \frac{\delta L}{\delta v1}\Delta v1 + \frac{\delta L}{\delta v2}\Delta v2$$

Poichè il gradiente si muove nella direzione di incremento della funzione, per minimizzare l'errore dobbiamo considerare la sua direzione negativa. Definiamo ora i vettori:

$$\nabla L \equiv (\frac{\delta L}{\delta v1}, \frac{\delta L}{\delta v2})$$

$$\Delta v \equiv (\Delta v1, \Delta v2)$$

Da cui possiamo riscrivere ΔL come:

$$\Delta L \approx \nabla L \cdot \Delta v$$

Se adesso definiamo Δv come:

$$\Delta v = -a\nabla L$$

Avremo:

$$\Delta L \approx -a||\nabla L||^2$$

Essendo $||\nabla L||^2 \geq 0$ allora $\Delta L \leq 0$. Da questa formula possiamo ricavarci il vettore v come:

$$v = v - a\frac{\delta L}{\delta v}$$

2.3 Training

Il training o learning (apprendimento) è il processo attraverso il quale un modello di rete neurale cerca di minimizzare l'errore L dell'output di uscita y rispetto dei *target value* t. Esistono diverse tipologie di apprendimento: *supervised learning*, *unsupervised learning*, $reinforcement\ learning$. Come caso d'esempio prenderemo in considerazione il supervised learning (apprendimento supervisionato).

Nell'apprendimento supervisionato si ha a disposizione un dataset D di dimensione N in cui a ciascun input x_i vi è associato un *target value* t_i. Il processo di *training* verrà così definito:

Input: rete neurale θ con parametri inizializzati, dataset D di dimensione N
Result: Rete neurale ottimizzata θ^*

$\theta^* \leftarrow \theta$
for $i \leftarrow 0$, **to** *Epochs* **do**
  shuffle(D)
  for $j \leftarrow 0$, **to** N **do**
    $y \leftarrow feedForward(\theta^*, x_j)$
    $L = \text{Loss}(y, t_j)$
    update(θ^*,L)
  end
end
return θ^*

Algorithm 1: Training algorithm

Nell'algoritmo sopra descritto per ogni istanza del dataset viene effettuato il feed-forward della rete neurale, il calcolo della *Loss*, la *Back propagation* e il *Gradient descent* (tramite $update(\theta^*, L)$). Questo processo viene poi ripetuto *Epochs* volte.

Spesso durante il training viene utilizzato una variante del *Gradient descent*: Lo *Stochastic gradient descent*(SGD)[2]. L'SGD anzichè aggiornare i parametri v attraverso il calcolo della derivata parziale della *Loss* di una singola istanza del dataset, effettua l'update tramite la media delle derivate parziali di un *batch* di istanze del dataset.

$$v = v - a\frac{1}{M}\sum_{i}^{M}\frac{\delta L_i}{\delta v}$$

Dove M è la dimensione del *batch* preso in considerazione.
Questa variante comporta due benefici: la parallelizzazione del processo di feed-forward e *Backpropagation* per ogni *batch* e la facilitazione della ricerca del minimo globale della funzione d'errore.

3 Metodo

In questa sezione verrà presentato un innovativo algoritmo di *training* e il suo utilizzo in un nuovo metodo per l'ottimizzazione di una rete neurale tramite un sistema distribuito.

3.1 Edge-Popup

Recentemente è stato pubblicato un nuovo tipo di algoritmo (Edge-popup) [5] che permette di associare ad ogni peso all'interno di una rete neurale uno score o punteggio e che determina

l'efficacia di quel peso all'interno di un subnetwork della rete neurale rispetto alla loss calcolata. Le assunzioni che verranno fatte ora per un fully-connected neural network possono essere estese anche ad un convolutional neural network. Richiamando il capitolo 1.1, in una architettura neurale fully-connected ciascun neurone d' output j di ciascun layer l durante il feed-forward computa le seguenti equazioni:

$$z_j = \sum_{(i)} x_i \cdot w_{i,j} + b_j$$

$$a_j = f(z_j)$$

Dove x_i è l'input i e $w_{i,j}$ è il peso che connette l'input i con l'output j. b_j è il bias del neurone d' output j e $f(z_j)$ è una funzione di attivazione non lineare. Il metodo più comunemente utilizzato per allenare ciascun peso all'interno della rete neurale è il Gradient descent tramite la *Back Propagation* [6]. L'algoritmo precedentemente menzionato utilizza lo stesso metodo, ma anzichè aggiornare i pesi aggiorna degli score associati ad essi. L' Edge-Popup sceglie per ciascun layer l un parametro k che determina la percentuale di pesi del layer l che devono essere utilizzati durante il feed-forward. Ciascun neurone d'output cambierà così le sue equazioni a seconda del nuovo parametro k:

$$z_j = \sum_{(i)} x_i \cdot w_{i,j} \cdot h(s_{i,j}) + b_j$$

$$a_j = f(z_j)$$

Dove $s_{i,j}$ è lo score associato al peso che connette il neurone d'input i col neurone d output j e $h(s_{i,j}) = 1$ se lo score $s_{i,j}$ è tra la percentuale dei k migliori score del layer l altrimenti vale 0. Per aggiornare lo score $s_{i,j}$ viene applicata la *Back propagation* tramite lo straight-through gradient estimator [7], nel quale h è trattato come l' identità durante il "backwards pass". In tal senso ogni score $s_{i,j}$ indipendentemente se sia stato usato o meno durante il feed-forward verrà aggiornato, se ignoriamo il momentum e il weight decay [8], tramite la seguente equazione:

$$\tilde{s}_{i,j} = s_{i,j} - a\frac{\delta L}{\delta z_j} w_{i,j} x_i$$

Dove L denota la loss della rete rispetto un' instanza del dataset. Per quanto riguarda invece architetture convoluzionali gli score o punteggi non vengono affidati ai singoli pesi, bensì ai kernel che contengono un insieme di pesi.
Durante il processo di *training* attraverso l'Edge-popup, il subnetwork che verrà scelto per il feed-forward verrà selezionato in base ai migliori k pesi di ciascun layer l.

3.2 Stato dell'arte

L' Edge-popup è un tipo di algoritmo estremamente potente che permette, una volta inizializzata una rete neurale, di trovare una sotto-rete o subnetwork all'interno della rete randomicamente inizializzata che performi il task affidatogli con una notevole accuratezza.

Lo stato dell'arte attuale mostra, sotto alcune condizioni, come la ricerca dei parametri della rete permetta di raggiungere un livello di ottimizzazione efficiente quanto lo stato dell'arte attuale ottenuto tramite un classico processo di *training*.

Di seguito verrà presentato lo stato dell'arte dell'Edge-popup sul dataset CIFAR-10 [9] tramite l'utilizzo di reti neurali implementate sul modello VGG, lo stesso modello utilizzato anche da *Frankle and Carbin* [10] e *Zhou et al.* [11].

L'inizializzazione della rete neurale come mostrato da *Ramanujan et al.* è fondamentale per una performance ottimale ottenuta dall'algoritmo Edge-popup. *Ramanujan et al.* utilizzano due differenti tipi di distribuzioni come inizializzazione dei parametri della rete:

• Kaiming Normal [12], denotata con $\mathcal{N}_k$ e definita come $\mathcal{N}_k = \mathcal{N}(0, \sqrt{2/n_{l-1}})$ dove $\mathcal{N}$ costituisce la distribuzione normale.
• Signed Kaiming Constant, denotata con $\mathcal{U}_k$. Ciascun peso viene scelto come costante e il suo segno è scelto casualmente.

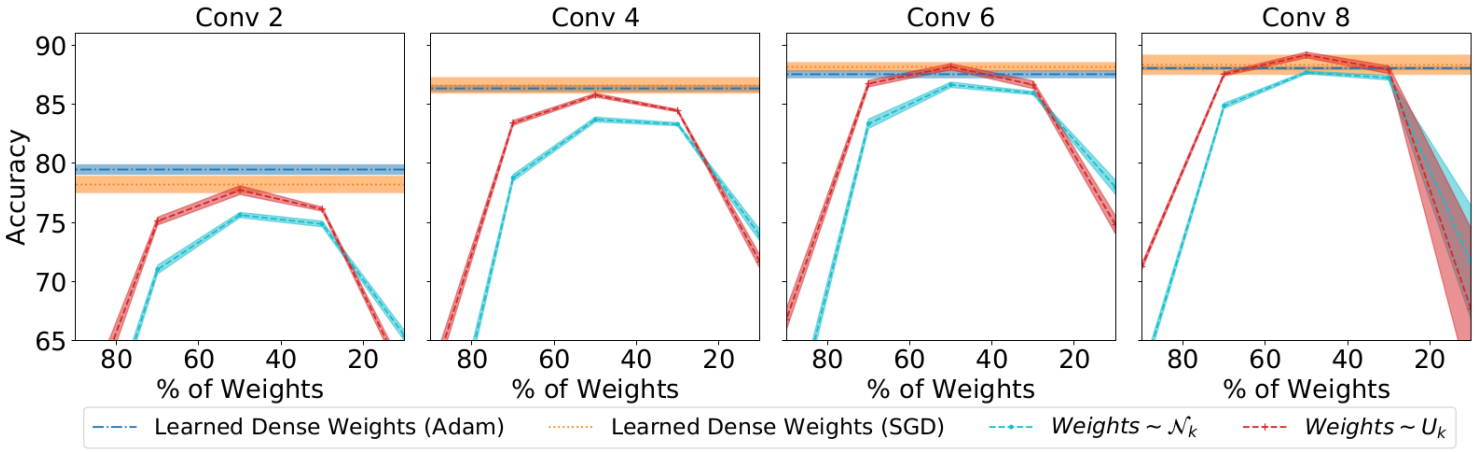

Figure 4: Esperimenti sul Dataset CIFAR-10 [9]. La linea orizzontale corrisponde alle reti neurali ottenute tramite un *training* classico. Il % Weights indica il valore k (percentuale di pesi presi in considerazione durante il feed-forward), per ogni layer l di ciascuna rete.

Nella Figure 4 possiamo notare le prestazioni ottenute da differenti tipologie di reti neurali tramite l'utilizzo dell'Edge-popup. Il training per ciascuna rete è stato portato avanti per 100 Epochs. Quando ottimizzata con l' Adam [18] (variante del gradient descent) il *learning*

rate utilizzato è stato portato a 0.1, mentre con l'SGD a 0.01. Per entrambi gli algoritmi di ottimizzazione è stato utilizzato un batch size di 128 e un *momentum* [8] (parametro aggiuntivo per il *Gradient descent*) di 0.9.

Dai test risulta che la distribuzione $\mathcal{U}_k$ sia migliore della Kaiming Normal su questo tipo di rete e con questo tipo di algoritmo. Inoltre l'efficienza migliore per ciascuna Neural Network si ha quando il parametro k, che denota la percentuale di pesi per ciascun layer l utilizzati durante il feed-forward sia tra il 30% e 70%, questo è dovuto al fatto che intorno a quella percentuale la rete ha una più ampia gamma di possibili subnetwork da generare.

3.3 Algoritmo distribuito

Come già anticipato nell'introduzione, uno degli ostacoli più ostici del deep learning è la limitatezza computazionale delle risorse che processano questo tipo di algoritmi. Idee che permettano una parallelizzazione delle risorse sono già state proposte in passato. Come l'idea di fare la media dei pesi di una stessa rete neurale allenata separatamente da più dispositivi, usando tecniche come la nonlinear cross validation (NCV) [13], che si affida però su l'assunzione secondo cui il modello neurale deve essere stato allenato completamente sull'intero dataset per almeno un' epoca.
Un' implementazione che abbia una performance simile può essere ottenuta applicando l'Edge-popup come algoritmo di *training*. Si può infatti dividere il dataset in K subset e affidare il training a K nodi, per poi successivamente mantenere il miglior punteggio di ciascun parametro della rete ottenuto dai K nodi. L'algoritmo che viene ora illustrato basa il suo funzionamento proprio su quest'idea.

Data una rete neurale θ, un dataset D di dimensione N, dove N è il numero di instanze che possiede, un insieme di nodi K con $NmodK = 0$ e un nodo centrale A, si può affidare ad ogni nodo K_i un subset $S_i \subset D$ con M istanze, dove $NmodM = 0$ ma non necessariamente $N/K = M$, in modo tale che ogni nodo K_i esegua l'algoritmo Edge-popup sulle istanze del subset affidatogli, per poi riportare al nodo centrale A il punteggio di ciascun peso rispetto a S_i. Successivamente A aggiorna gli score con i miglior punteggi riportati da ciascun nodo K_i, genera il submodel $\tilde{\theta}$ e si itera il processo.

Pseudocodice:

Input: Modello θ, k_l percentuale di pesi utilizzati per i layer l, K nodi,
Dataset D di dimensione N, K subset S_i di dimensione M
Result: submodel ottimizzato $\tilde{\theta}$

do
  for $i \leftarrow 0$, **to** K **do**
    $S_i \leftarrow getSample(D, M)$
    $Scores_i \leftarrow assign(S_i, K_i, \theta)$
  end
  Scores$_K$ ←getBestScores(Scores)
  $\tilde{\theta} \leftarrow getBestSubnetwork(\theta, Scores_K)$
while $\tilde{\theta}$ *Not convergence*;
return $\tilde{\theta}$

Algorithm 2: Algoritmo Distribuito

La funzione $assign(S_i, K_i, \theta)$ viene svolta in parallelo con le altre funzioni $assign(...)$ dai nodi K_i e consiste nell'implementazione dell' algoritmo Edge-popup da parte del nodo K_i sul subset S_i.

4 Test

Nella fase di test verranno presi in considerazioni i vantaggi computazionali al pari di quelli di ottimizzazione dei modelli derivati dall'algoritmo distribuito definito nella sezione 3.3.

4.1 Setup

I test sono stati effettuati su una singola macchina: RAM 15,6 GB, processore Intel Core i7-7700K CPU @ 4.20GHz x 8, OS Ubuntu 16.04 LTS 64 bit. Il tempo di processamento per l'algoritmo distribuito è una stima del tempo nel caso in cui il training venisse processato su più dispositivi con le stesse caratteristiche.
I modelli di rete neurale scelti sono due: una rete convoluzionale basata sul modello VGG con la sola modifica dell'inglobamento dei layer convoluzionali in un residual layer e una rete neurale fully-connected.

L'inizializzazione utilizzata per i modelli è la Signed Kaiming Costant che campiona da una distribuzione di probabilità uniforme nell'insieme $\{-\sigma_k, \sigma_k\}$ dove σ_k è la deviazione standard della Kaiming Normal. In ciascun modello di training il batch size utilizzato è 4, così come il numero di thread. É stato scelto l'SGD con i seguenti parametri: Il *learning rate* a 0.01 senza alcun decremento durante il processo di learning. Il momentum a 0.9. L'orizzonte

temporale del numero di epoche è stato settato a 10.
A causa del basso numero di epoche l'accuratezza non è il fattore di comparazione più adeguato, pertanto come termine di paragone è stata scelta la funzione d'errore.
La *Loss* presa in considerazione è la funzione *Focal Loss* [17] definita come:

$$p_t = \begin{cases} y(v, x_i), & \text{if } t_i = 1 \\ 1 - y(v, x_i), & \text{otherwise} \end{cases}$$

$$L = -(1 - p_t)^{\gamma} log(p_t)$$

Dove L è la *Loss*, $y(v, x_i)$ corrisponde all'output della rete, t_i è il *target value* e γ è un parametro che è stato settato a 2. Il *training* è stato eseguito sul dataset CIFAR-10 [9] diviso adeguatamente in training set (50.000 instanze) e test set (10.000 istanze). Per i layer convoluzionali, al fine di aumentare la performance, è stato scelto di non considerare lo score complessivo di ciascun kernel bensì quelli singoli di ciascun peso all'interno dei filtri. Il codice è reperibile su github: `https://github.com/VivianoRiccardo/TesiTriennale` vi è un'implementazione sia in C che in Python tramite il modulo Cython. Il test è diviso in una parte server e una parte client che corrisponde al *training* effettuato dai K_i nodi.

4.2 Risultati

Il modello fully-connected è composto da 5 layers. La funzione di attivazione usata è la Relu [19] per i primi 4 layers e la Softmax [20] per l'ultimo layer. Nessun tipo di normalizzazione o di tecnica di regolarizzazione è stata usata durante il *training*. La rete convoluzionale è invece composta da un layer convoluzionale seguito dal max-pooling, due residual layers, ciascuno con due layer convoluzionali al loro interno, 2 fully-connected layers. Come per il modello fully-connected l'ultimo layer ha come funzione d'attivazione la Softmax, mentre tutti gli altri livelli utilizzano la funzione Leaky Relu [21].
La percentuale k di pesi mantenuti per ciascun livello per entrambe le reti neurali è stato settato a 0.5 (50%) eccetto che per l'ultimo layer in cui è stato deciso di mantenere tutti i parametri durante il feed-forward (k = 1). Nella Table 1 viene descritta più accuratamente l'architettura dei due modelli.

Layers	fully-connected model: number of neurons	convolutional model: number of neurons
Layer 1	200	20 14 14, padd, pool
Layer 2	200	begin residual, 40 14 14, padd
Layer 3	200	20 14 14, end residual, padd
Layer 4	200	begin residual, 40 14 14, padd
Layer 5	10	20 14 14, end residual, padd
Layer 6		100
Layer 7		10

Table 1

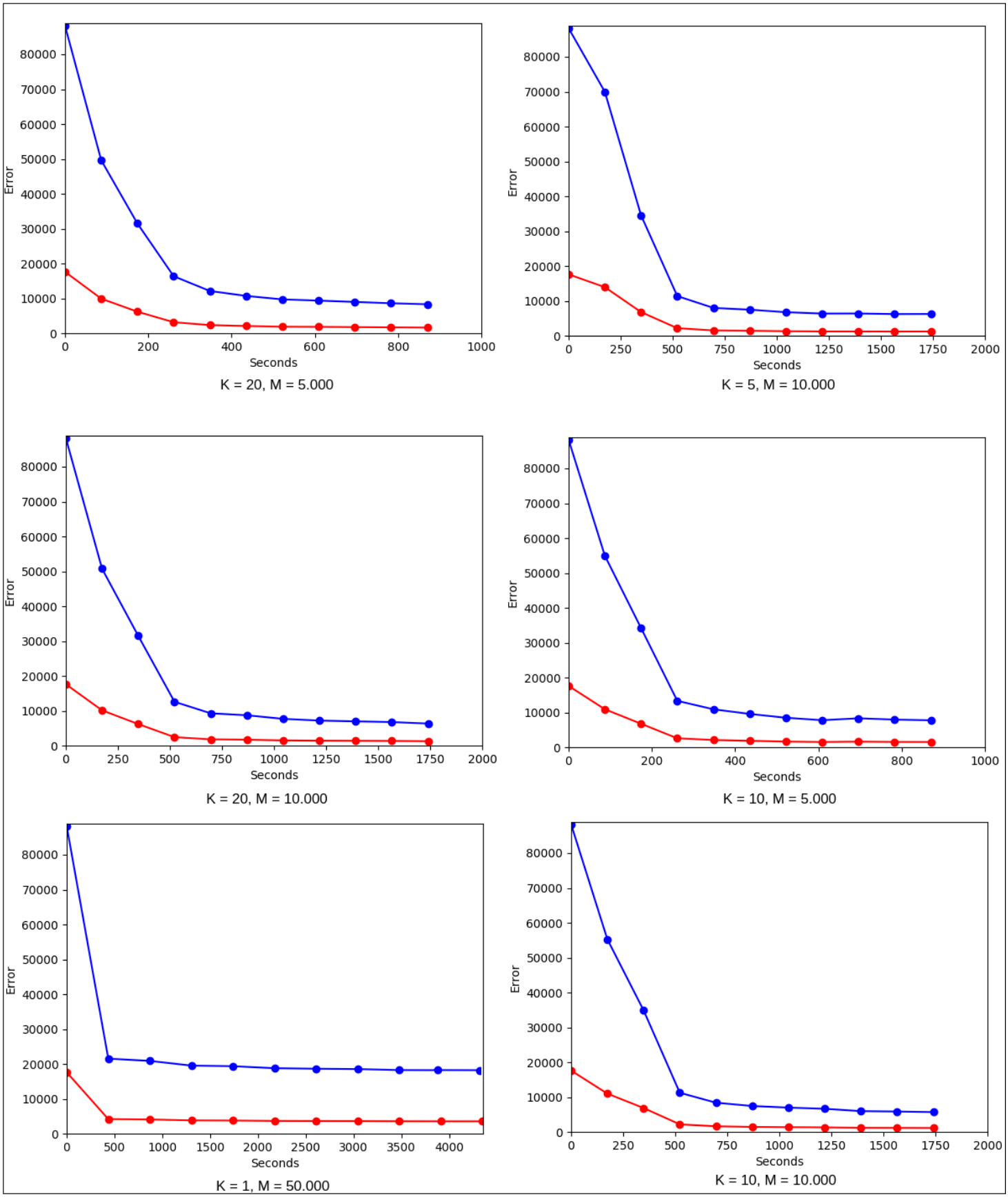

Figure 5: I grafici disposti rappresentano gli errori per ciascuna epoca del modello fully-connected per valori differenti di K e M. La blue line corrisponde all'errore sul training set, la red line corrisponde all'errore calcolato sul test set. Ogni dot corrisponde ad un epoca differente. K indica il numero di nodi utilizzati durante il training, M indica il numero di istanze del training set utilizzate da ciascun nodo K durante l'allenamento.

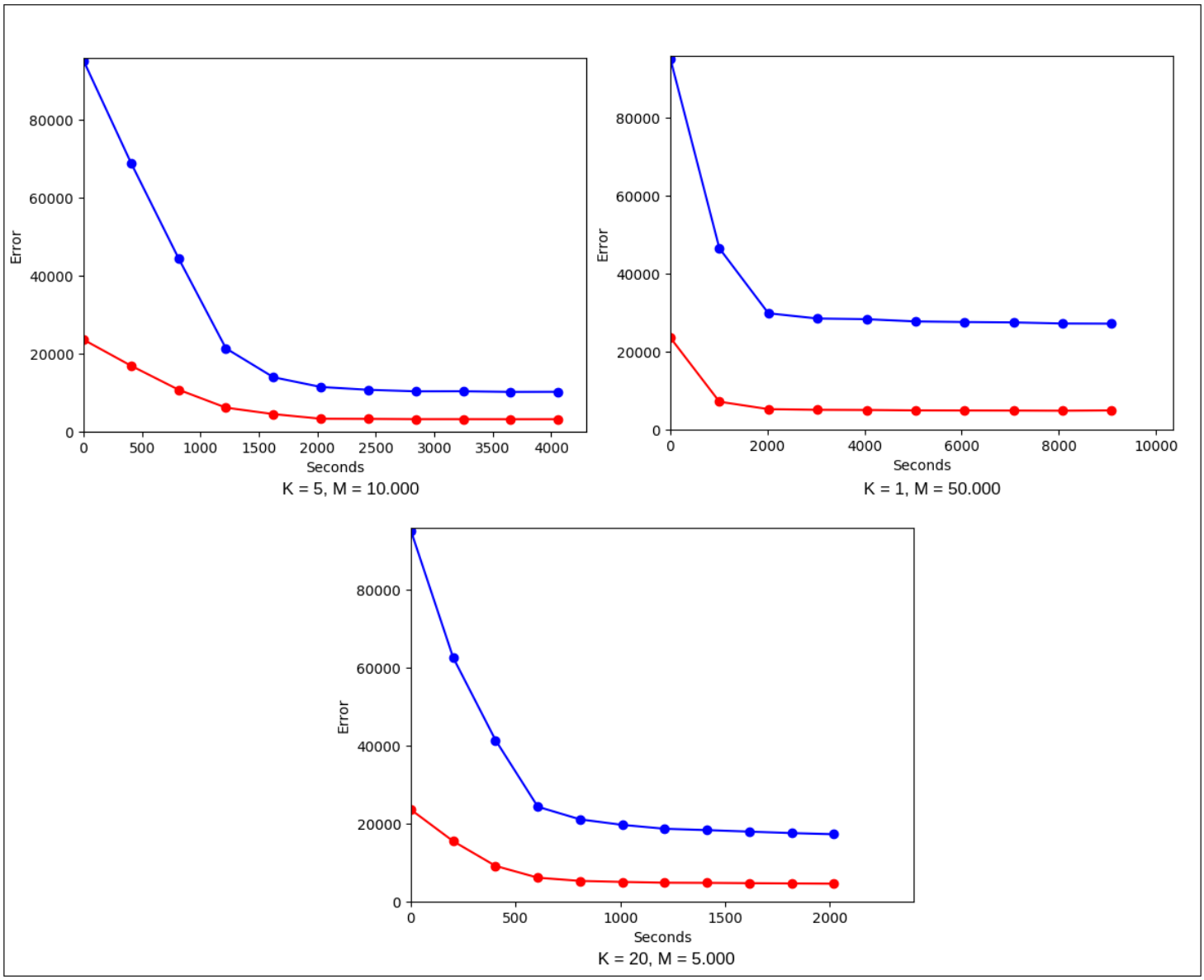

Figure 6: I grafici disposti rappresentano gli errori per ciascuna epoca del modello convoluzionale per valori differenti di K e M. Come per la figure 5 La blue line corrisponde all'errore sul training set, la red line corrisponde all'errore calcolato sul test set. Anche in questo caso ogni dot corrisponde ad un epoca differente. K indica il numero di nodi utilizzati durante il training, M indica il numero di istanze del training set utilizzate da ciascun nodo K durante l'allenamento.

La Figure 5 mostra sei diversi *training* per il modello fully-connected. La performance peggiore sia in termini di ottimizzazione dei parametri che temporale è data quando K, numero di nodi utilizzati, corrisponde ad 1 e M, numero di istanze utilizzate dall'unico nodo K_1, è 50.000. Un tale settaggio dei parametri K e M corrisponde al *training* classico effettuato dall'algoritmo Edge-Popup. I restanti 5 grafici ottengono una performance migliore sia temporale che di ottimizzazione e fanno riferimento all'utilizzo dell'algoritmo distribuito menzionato nel capitolo 3.3. In particolare, quando K = 1 e M = 50.000 dopo la prima epoca il modello neurale raggiunge una $Loss \approx 4217$ e ottiene alla decima epoca un valore ≈ 3576 sul test set. I *training* che fanno riferimento invece all'algoritmo distribuito raggiungono un valore minore di 4217 intorno alla terza epoca, tuttavia continuando l'allenamento riescono ad ottenere prestazioni più efficienti fino ad avere una $Loss \leq 1600$. Graficamente parlando, il *training* classico tramite l'Edge-popup mostra la sua efficacia dopo già la prima epoca, ma allo stesso tempo dimostra anche la facilità con cui rimane bloccato in minimi locali. L'algoritmo distribuito invece supera questa condizione evitando di incagliarsi in minimi locali e ottenendo prestazioni efficienti in tempi più brevi.
La performance migliore considerando il valore temporale e di minimizzazione della funzione d'errore è ottenuta nel quarto grafico in cui la Loss raggiunge un valore di ≈ 1562 in un intervallo temporale di 882 secondi a dispetto dell'algoritmo classico ottenuto dal quinto grafico con una Loss ≈ 3576 in 4997 secondi.

Nella figure 6 sono stati effettuati test sul modello convoluzionale, in questo caso in tutti i test i risultati della performance è peggiore rispetto alla rete neurale precedente in quanto, a causa del basso numero di parametri di ciascun layer rispetto al modello fully-connected, la varietà di subnetwork che si possono generare è nettamente inferiore. Tuttavia come per il modello completamente connesso gli errori calcolati sul training set e sul test set hanno una performance nettamente migliore sia temporale che di ottimizzazione nel caso dell'algoritmo distribuito a dispetto di quello non distribuito (K = 1, M = 50.000).

5 Conclusione

5.1 Considerazioni finali e idee future

In questa tesi abbiamo presentato l'algoritmo Edge-Popup (sezione 3.1) e come, tramite i dovuti accorgimenti, il suo utilizzo può essere applicato su un sistema distribuito (sezione 3.3). I risultati mostrano un'ottimizzazione temporale e sulla performance in generale, ottenendo un miglioramento nelle prestazioni fino a due volte maggiori rispetto alla versione non distribuita ed un vantaggio temporale fino a 5 volte più efficiente.
Su progetti più grandi con una mole di dati decisamente più elevata l'algoritmo in questione può sicuramente raggiungere prestazioni temporali ancora più vantaggiose in base al numero di nodi in cui si decide di dividere il proprio dataset. In particolare questo tipo di algoritmo può venir preso seriamente in considerazione nel caso di progetti di reinforcement learning in

cui i dati su cui i modelli effettuano il *training* sono, in alcuni casi, potenzialmente infiniti.

È stata quindi proposta una soluzione semplice ed efficace per effettuare un *training* più rapido su algoritmi di deep learning e che presenta anche ampi margini di miglioramento sulla sua implementazione. Infatti, l'algoritmo distribuito mostrato nella sezione 3.3 può essere affiancato e migliorato con altre tecniche di ottimizzazione che si concentrano sulla reinizializzazione dei pesi con un low score all'interno del modello che si sta allenando. A questo proposito si può pensare ad algoritmi genetici o che fanno riferimento alla branca della "Swarm Intelligence" e che, attraverso le opportune tecniche e alla fine di ogni iterazione, decidono di assegnare valori diversi ai pesi che hanno ottenuto un punteggio basso (e che quindi non vengo presi in considerazione nel subnetwork) durante il processo di allenamento. Nulla esclude in futuro la realizzazione di tecniche più elaborate che hanno come base di partenza questo algoritmo.

5.2 Ringraziamenti

Volevo ringraziare in primis la professoressa Maria Fiora Pirri per avermi avvicinato a questo campo tramite il suo corso Metodi quantitativi per l'informatica e sopratutto per avermi fatto da relatrice per questa tesi. Un rigraziamento anche a mio fratello che mi ha incoraggiato a seguire questa direzione e ai miei genitori per avermi supportato e sopportato in questo periodo universitario.

Bibliografia

[1] Neil C. Thompson, Kristjan Greenewald, Keeheon Lee, Gabriel F. Manso: "**The Computational Limits of Deep Learning**", (ArXiv), 2020.

[2] Shun-ichiAmari: "**Backpropagation and stochastic gradient descent method**", (Dept. of Mathematical Engineering, University of Tokyo, Bunkyo-ku, Hongo, Tokyo 113, Japan), 1992

[3] Chien-Chih Wang, Kent Loong Tan, Chun-Ting Chen, Yu-Hsiang Lin, S. Sathiya Keerthi, Dhruv Mahajan, S. Sundararajan, Chih-Jen Lin, "**Distributed Newton Methods for Deep Neural Networks**",(Department of Computer Science, National Taiwan University,Department of Physics, National Taiwan University,Microsoft,Facebook Research),2018.

[4] Robert K. L. Kennedy, Taghi M. Khoshgoftaar , Flavio Villanustre,Timothy Humphrey: "**A parallel and distributed stochastic gradient descent implementation using commodity clusters**", (Journal of Big Data),2019.

[5] RVivek Ramanujan, Mitchell Wortsman, Aniruddha Kembhavi, Ali Farhadi, Mohammad Rastegari: "**What's Hidden in a Randomly Weighted Neural Network?**", (ArXiv),2019.

[6] B. B. Le Cun, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, L. D. Jackel, "**Handwritten digit recognition with a back-propagation network**", (NIPS), 1989.

[7] Yoshua Bengio, Nicholas Lonard, and Aaron Courville., "**timating or propagating gradients through stochastic neurons for conditional computation**", 2013.

[8] Anders Krogh and John A. Hertz, "**A simple weight decay can improve generalization**",(NIPS), 1991.

[9] Alex Krizhevsky,"**Learning multiple layers of features from tiny images**",(Technical report, University of Toronto), 2009.

[10] Jonathan Frankle and Michael Carbin,"**The lottery ticket hypothesis: Finding sparse, trainable neural networks**" (ICLR 2019), 2019

[11] Hattie Zhou, Janice Lan, Rosanne Liu, and Jason Yosinski, "**Deconstructing lottery tickets: Zeros, signs, and the supermask**" 2019.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, "**Delving deep into rectifiers: Surpassing human-level performance on imagenet classification**".(IEEE International Conference on Computer Vision,ICCV),2015.

[13] Joachim Utans, "**Weight Averaging for Neural Networks and Local Resampling Schemes**",(London Business School Sussex Place, Regent's Park),1996.

[14] Yoshua Bengio, "**Deep Learning of Representations for Unsupervised and Transfer Learning**",(JMLR Workshop and Conference Proceedings),2012.

[15] Yoshua Bengio, Lecun, Yann "**Convolutional Networks for Images, Speech, and Time-Series**",(JOUR),1977.

[16] Goodfellow I, Bengio Y, Courville A.: "**Deep Learning**", (MIT Press), 2016.

[17] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, Piotr Dollár: "**Focal Loss for Dense Object Detection**", (Arxiv), 2018.

[18] Diederik P. Kingma, Jimmy Ba: "**Adam: A Method for Stochastic Optimization**", (Arxiv), 2014.

[19] Abien Fred Agarap: "**Deep Learning using Rectified Linear Units (ReLU)**", (Arxiv), 2018.

[20] Bridle, J. S. : "**Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition**", (Neurocomputing. Springer Berlin Heidelberg), 1990.

[21] Fayyaz ul Amir Afsar Minhas, Amina Asif : "**Learning Neural Activations**", (Arxiv), 2019.