Università Mercatorum

Corso di Laurea in Statistica e Big Data

Classe L41

Convolutional Neural Network: studio e applicazione in Julia

Relatore

Raffaele Miele

Candidato

Mattia Nicolò Careddu

Matr. 0412100021

INDICE

INTRODUZIONE

Le Convolutional Neural Network sono una tipologia di rete neurale particolarmente efficace nel riconoscimento delle immagini.

Questa dissertazione verterà sulle Convolutional Neural Network, con particolare riferimento a due dei principali modelli: AlexNet e ResNet. AlexNet è stato introdotto nel 2012 e ha rappresentato una svolta nello sviluppo delle CNN grazie all'uso di una grande quantità di dati di addestramento e all'impiego di tecniche di regolarizzazione come il *dropout*. ResNet, introdotto nel 2015, ha introdotto il concetto di *skip connection*, che ha permesso di addestrare reti neurali più profonde senza incorrere in problemi di degradazione della performance.

Infine, sarà presentata l'implementazione e l'addestramento di una Convolutional Neural Network per il riconoscimento degli oggetti utilizzando il dataset CIFAR-10. Verrà utilizzato il linguaggio di programmazione Julia, noto per la sua efficienza computazionale e la sua facilità di utilizzo, insieme al *framework* di machine learning Flux.

Deep Residual Learning for Image Recognition e *ImageNet Classification with Deep Convolutional Neural Networks* sono due tra gli articoli più importanti che ho utilizzato nella stesura dell'elaborato; ho inoltre consultato più volte il volume *Deep Learning* di Ian Goodfellow.

1. DEEP NEURAL NETWORK

1.1 Computer Vision

La Computer Vision è un campo dell'intelligenza artificiale che si occupa della capacità dei computer di interpretare, analizzare e comprendere le immagini e i video. Ci sono diverse tipologie di Computer Vision, tra cui il riconoscimento di oggetti, la classificazione di immagini, la segmentazione e la localizzazione di oggetti.

Le reti convoluzionali sono una tipologia di rete neurale artificiale ampiamente utilizzata in Computer Vision. Esse sfruttano il processo di convoluzione per estrarre automaticamente caratteristiche importanti dalle immagini, come ad esempio i bordi, le texture e i pattern. Queste caratteristiche vengono poi passate attraverso una serie di strati di neuroni per effettuare la classificazione o la segmentazione delle immagini.

Le reti convoluzionali sono particolarmente efficaci per l'elaborazione di immagini grazie alla loro capacità di rilevare automaticamente i pattern e le caratteristiche importanti delle immagini, senza che sia necessario specificare manualmente quale sia la rappresentazione migliore per ciascuna immagine. Ciò le rende una scelta ideale per una vasta gamma di applicazioni di Computer Vision, come ad esempio la classificazione di immagini mediche, la rilevazione di oggetti in tempo reale e l'analisi del movimento umano.

1.2 Artificial Neural network

L'obiettivo di una rete neurale è quello di approssimare una funzione **f***. Una rete neurale definisce il suo output nella forma $\mathbf{y = f(x; \theta)}$, dove x rappresenta l'input e $\boldsymbol{\theta}$ il valore del parametro che la rete neurale apprende in modo da avere la migliore approssimazione della funzione **f*.**

Le reti neurali vengono chiamate *reti* in quanto sono rappresentate dalla composizione di diverse funzioni. La composizione più usata è quella nella forma $\mathbf{f^3(f^2(f^1(x))}$ dove $\mathbf{f^1}$ rappresenta il primo strato, $\mathbf{f^2}$ il secondo e così via. Il primo strato viene chiamato *input*

layer, mentre l'ultimo *output layer*. I restanti strati nel mezzo vengono chiamati *hidden layers*.

Il numero di strati presenti nel modello viene definito *depth* (da qui il nome *deep learning*), mentre il numero massimo di unità presenti in uno strato viene definito *width.*

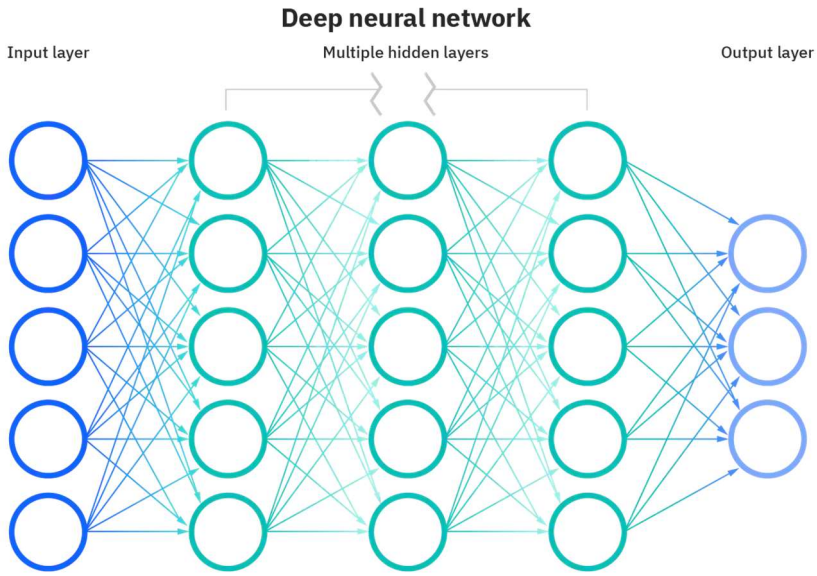

Figura 1: Esempio di rete neurale con 3 *hidden layers*.[1]

L'unità alla base di una rete neurale è il neurone artificiale, così chiamato perché ispirato alla neuroscienza. Il neurone artificiale riceve degli *input* da altre unità e calcola la sua funzione di attivazione.

L'esempio più semplice di neurone artificiale è il *perceptron*, sviluppato, a partire dal 1950, da Frank Rosenblatt. Un *perceptron* riceve *n input,* $\mathbf{x_1, x_2, \ldots x_n}$ e produce un singolo *output* binario. Per calcolare *l'output* il *perceptron* moltiplica ogni *input* per il relativo peso $\mathbf{w_1, w_2, .. w_n}$ e lo compara con un valore di soglia.

$y = 0$ se $\sum_j \quad x_j\, w_j \leq treshold$ altrimenti $y = 1$

[1] https://www.ibm.com/it-it/cloud/learn/neural-networks

Un modo semplificato di esprimere questa equazione è tramite il *dot product* del vettore degli *input* [$\mathbf{x_1, x_2,\ldots x_n}$] e del vettore dei pesi [$\mathbf{w_1, w_2,.. w_n}$] e con l'introduzione della variabile di *bias* che sostituisce il *treshold.*

$y = 0$se $w \cdot x + b \leq 0$ altrimenti $y = 1$

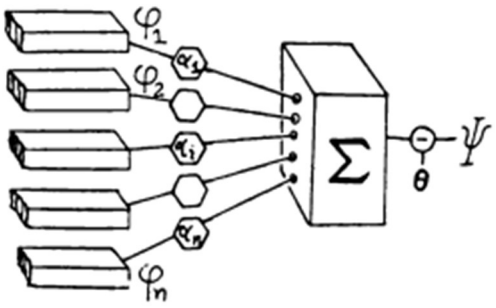

Figura 2: Perceptron[2]

Un'altra tipologia di neuroni artificiali sono i *sigmoid neurons.* Un *sigmoid neuron* riceve *n input* che possono assumere un qualsiasi valore (a differenza dei soli valori binari accettati dal *perceptron*), utilizza dei pesi e un *bias*, e produce un *output* che può assumere un qualsiasi valore tra 0 e 1.

$y = \sigma(w \cdot x + b)$ dove σ rappresenta la funzione sigmoide definita da $\sigma = 1/1 + e^{-z}$

Più in generale, ogni neurone artificiale, definisce una funzione di attivazione che definisce l'output dato uno o più input. Queste funzioni di attivazione ci aiutano a introdurre non linearità nel modello. Grazie a questa caratteristica una rete neurale riesce ad approssimare una qualsiasi funzione **f***.

[2] Minsky Marvin, Papert Seymour A. *Perceptrons: An Introduction to Computational Geometry*, The MIT Press, Cambridge, Massachusetts, and London, England 1988, p. 28.

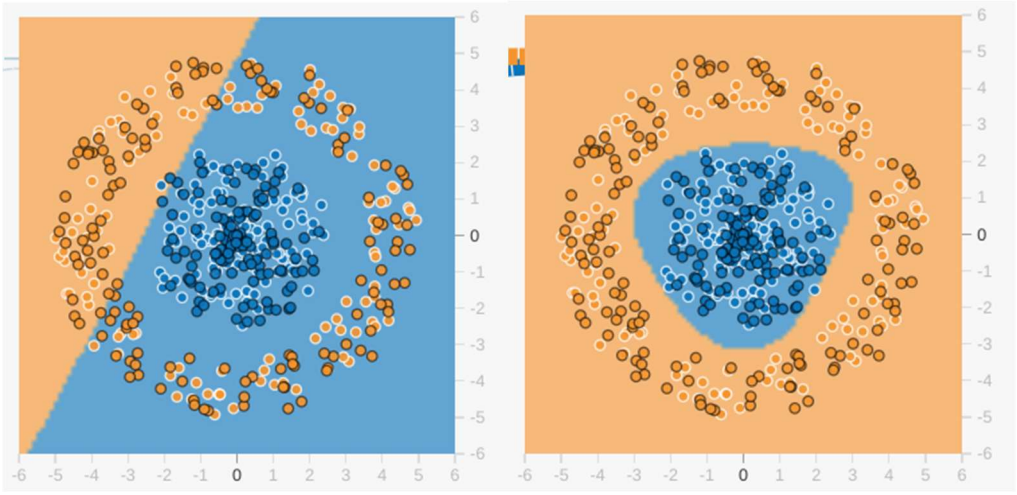

Figura 3: A sinistra la rappresentazione dell'apprendimento di una rete neurale con attivazione lineare, a destra una rete neurale che utilizza come funzione di attivazione la sigmoide.[3]

L'architettura di una rete neurale artificiale (ANN) descrive la sua struttura fondamentale e definisce come i neuroni artificiali sono organizzati e collegati tra loro. Una rete neurale è composta da uno o più strati di neuroni artificiali, ognuno dei quali può avere un diverso numero di unità. Ogni neurone riceve un input dalla somma ponderata dei segnali provenienti dai neuroni del layer precedente, e applica una funzione di attivazione per produrre un output.

Ogni connessione tra i neuroni ha un peso associato, che viene aggiornato durante la fase di addestramento della rete. L'obiettivo dell'addestramento è quello di trovare il set di pesi che minimizza l'errore tra l'output previsto dalla rete e l'output atteso per un determinato input.

Ci sono diversi tipi di architetture di reti neurali, ma le più comuni sono le feedforward neural networks (FFNN), le convolutional neural networks (CNN) e le recurrent neural networks (RNN).

L'addestramento di una rete neurale avviene attraverso un processo iterativo che utilizza il metodo del gradiente per ottimizzare una funzione di costo. Una rete neurale,

[3] https://playground.tensorflow.org/

grazie alla sua non linearità, produce una funzione di costo non convessa. Non c'è quindi garanzia della convergenza della funzione di costo e, per questo motivo, una rete neurale è sensibile all'inizializzazione dei pesi.

L'algoritmo di *backpropagation* calcola il gradiente della funzione di costo rispetto ai pesi della rete neurale. Questo metodo risulta estremamente più efficiente rispetto a calcolare il gradiente per ogni singolo peso della rete.

L'algoritmo di *backpropagation* divenne popolare dopo la pubblicazione del *paper Learning representations by back-propagating errors* del 1986 di Rumelhart, D., Hinton, G. & Williams R. L'algoritmo inizia calcolando la derivata parziale per ogni unità di *output*, arrivando così a conoscere quanto una modifica all'*input* x dell'unità di *output* ha effetto sull'errore. L'*input* x è il risultato della funzione di attivazione delle unità degli strati precedenti: possiamo quindi facilmente calcolare quanto viene influenzato l'errore da una modifica al peso w. Prendendo quindi in considerazione tutte le connessioni emanate dall'unità i troviamo:

$$\partial E/\partial yi = \sum_j \quad \partial E/\partial xi \cdot wji \quad [4]$$

Ripetendo questa procedura per tutti gli strati precedenti possiamo quindi utilizzare il gradiente per aggiornare i pesi della rete neurale. Possiamo aggiornare i pesi dopo un singolo caso *input-output* (dati di addestramento), oppure accumulare il gradiente per tutti i casi di *input-output* prima di fare l'aggiornamento.

1.2 Convolutional Neural Network

Le *Convolutional Neural Network* sono reti neurali specializzate per processare dati matriciali, come ad esempio *time-series* (1D) oppure immagini (2D). Il termine *Convolutional* indica che la rete neurale utilizza un'operazione matematica chiamata convoluzione. Il ruolo di una *Convolutional Neural Network* è quello di ridurre la

[4] Rumelhart, D., Hinton, G. & Williams, R. *Learning representations by back-propagating errors*. Nature 323, 533–536 (1986) pag 535.

dimensionalità spaziale dell'*input* senza perdere le caratteristiche necessarie ad ottenere una buona previsione.

Una *Convolutional Neural Network* è composta da tre differenti strati: *input layer, convolutional layer, fully-connected layers* (questi ultimi agiscono come una *artificial neural network*).

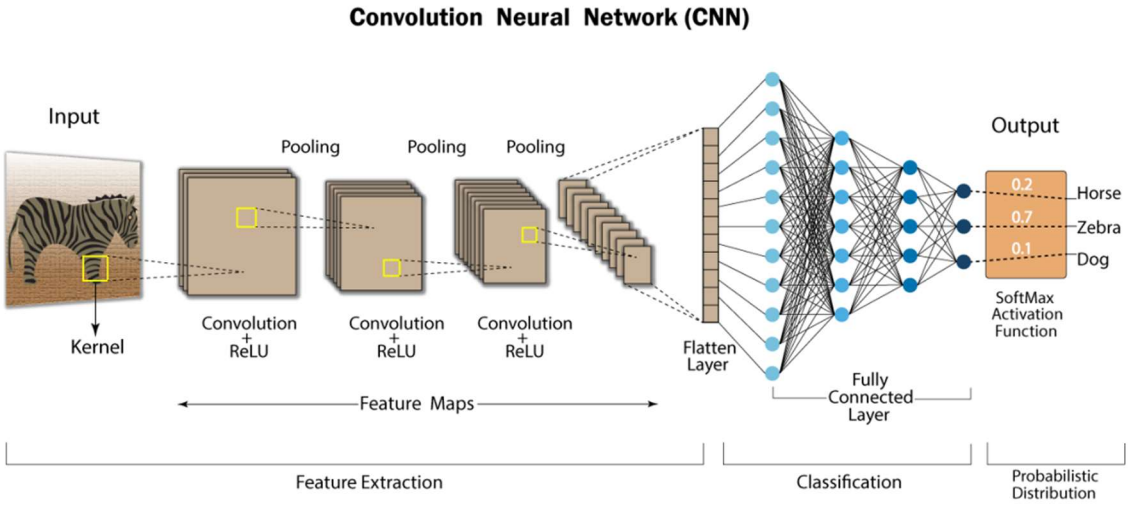

Figura 4: Esempio di architettura di una *Convolutional Neural Network* per la classificazione di immagini[5]

Un *Convolutional Layer* è solitamente composto da tre fasi: durante la prima fase vengono eseguite operazioni di convoluzione che producono delle attivazioni lineari; durante la seconda fase ogni attivazione lineare passa attraverso l'attivazione di funzioni non lineari, come la *rectified linear activation function;* durante la terza fase viene utilizzata una funzione di *pooling* al fine di ridurre la dimensionalità spaziale.

Durante il processo di convoluzione viene utilizzato un *kernel* che corrisponde ad una matrice, solitamente di piccole dimensioni spaziali, che si estende per tutto *l'input*. Durante ogni iterazione viene eseguito il prodotto *element-wise* tra le due matrici.

[5] https://developersbreach.com/convolution-neural-network-deep-learning/

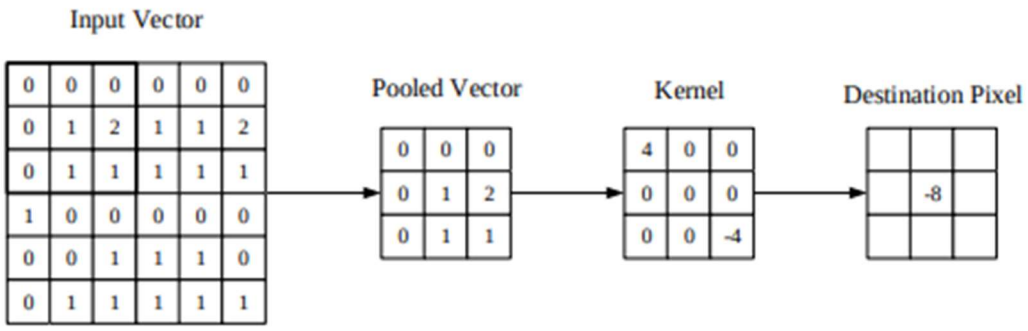

Figura 5: rappresentazione grafica di un *convolutional layer*[6].

Una funzione di *pooling* sostituisce l'*output* della rete in una determinata posizione con una statistica di sintesi degli *output* vicini. Le tre principali funzioni di *pooling* sono: *max pooling*, *min pooling* e *average pooling*.

[6] O'Shea Keiron, Nash Ryan, *An Introduction to Convolutional Neural Networks.* Department of Computer Science, Aberystwyth University, Ceredigion 2015

2. ARCHITETTURE DI CONVOLUTIONAL NEURAL NETWORK

2.1 AlexNet

AlexNet è il nome di un'architettura di una *convolutional neural network* sviluppata da Alex Krizhevsky in collaborazione con Ilya Sutskever e Geoffrey Hinton.

La rete neurale, che ha 60 milioni di parametri e 650.000 neuroni, è composta da cinque *convolutional layer* seguiti da un *max pooling layer,* tre *fully-connected layer* e un *output layer* con attivazione *softmax*. È stata addestrata sul dataset *ImageNet LSVRC-2010* composto da 1,2 milioni di immagini ad alta definizione con 1000 classi differenti.

Per i *convolutional layer* è stata utilizzata la funzione non lineare ReLU (Rectified Lineare Units) dove $f(x) = max(x, 0)$. Questo ha permesso alla rete di essere addestrata molto più velocemente rispetto alla funzione non lineare *than* dove $f(x) = than(x)$

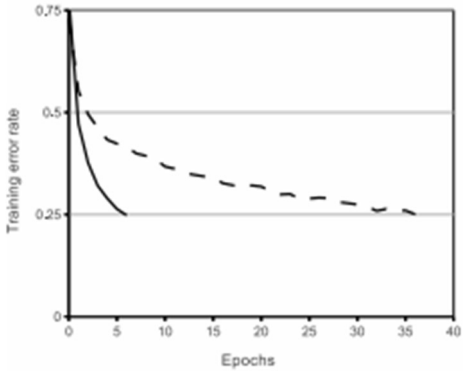

Figura 6: Una *convolutional neural network* a quattro strati con attivazione *ReLU* (linea spessa) raggiunge il 25% di errore sui dati di training *CIFAR-10* sei volte più velocemente rispetto a una convolutional neural network con attivazione *than* (linea tratteggiata).[7]

[7] Krizhevsky Alex, Sutskever Ilya, Hinton Geoffrey E. *ImageNet Classification with Deep Convolutional Neural Networks*, University of Toronto, 2012, p. 3.

I *kernel* del secondo, quarto e quinto *convolutional layer* sono connessi solo ai *kernel* dello strato precedente che risiedono sulla stessa GPU. I *kernel* del terzo *layer* sono connessi a tutti i *kernel* del *layer* precedente. I neuroni dei *fully-connected layers* sono connessi a tutti i neuroni dello strato precedente. I *layer* di normalizzazione seguono il primo e il secondo *convolutional layer.* I *layer* di max-pooling seguono i *layer* di normalizzazione e il quinto *layer* della *convolutional neural network.* A ogni *output* è applicata la funzione di attivazione *ReLU*.

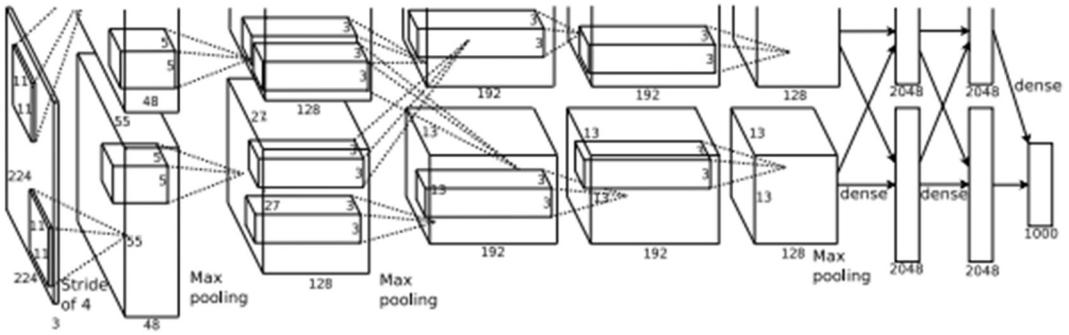

Figura 7: Architettura di AlexNet[8]

Il primo *convolutional layer* filtra *l'input* di dimensione 224 x 224 x 3 con 96 *kernels* di dimensione 11 x 11 x 3 con *stride* di 4 pixel; il secondo filtra *l'output* del primo *layer* (normalizzato e passato attraverso la funzione di *pooling*) con 256 *kernels* di 5 x 5 x 48; il terzo *layer* ha 384 *kernels* 3 x 3 x 256; il quarto ha 384 *kernels* 3 x 3 x 192 e il quinto *layer* ha 256 *kernels* di 3 x 3 x 192. Ogni strato del *fully-connected layer* ha 4096 neuroni.

Per ridurre *l'overfitting* sono state applicate due tecniche principali: *data augmentation* e *droput.* Si parla di *overfitting* quando un modello statistico si adatta ai dati osservati poiché ha un numero eccessivo di parametri rispetto al numero di osservazioni. Questo porta il modello ad avere un errore molto basso sui dati di *training* e molto alto sui dati di test.

[8] Krizhevsky Alex, Sutskever Ilya, Hinton Geoffrey E. *ImageNet Classification with Deep Convolutional Neural Networks,* University of Toronto, 2012, p. 5.

L'*overfitting* causa la modifica di alcune unità in modo da correggere l'errore di altre unità portando a un complesso coadattamento che impedisce al modello di generalizzare l'apprendimento su dati non osservati.

Per l'addestramento di AlexNet sono state usate due tecniche differenti per il *data augmentation*. Il primo metodo consiste nel traslare e riflettere l'immagine, mentre il secondo metodo consiste nell'alterazione dell'intensità dei *pixel* RGB dell'immagine.

Il *dropout* è una tecnica che comporta la rimozione dei nodi dall'*input layer* e dagli *hidden layers* rimuovendo temporaneamente tutte le connessioni di quei nodi durante il *forward pas*s e la *backpropagation.* I nodi sono rimossi con una probabilità *p.* Questo permette di creare una nuova sotto-architettura durante ogni iterazione. AlexNet utilizza una probabilità di *dropout* di $p = 0.5$.

2.2 ResNet

Dopo AlexNet, tutte le vittorie della competizione di ImageNet furono ottenute grazie ad architetture sempre più profonde, che utilizzavano quindi sempre più *layers* per ridurre l'errore. Aumentando però il numero di *layer* in una rete neurale andiamo incontro al problema del *vanishing/exploding gradient*. Questo problema porta il gradiente ad assumere valori o troppo grandi o prossimi allo zero, facendo divergere la rete nel primo caso, o non facendola mai convergere nel secondo.

Per questo nel 2015 viene introdotta una nuova tipologia di reti neurali: le *Residual Network* (ResNet). L'implementazione di ResNet ottenne un errore del 3,6% durante la competizione ImageNet del 2015.

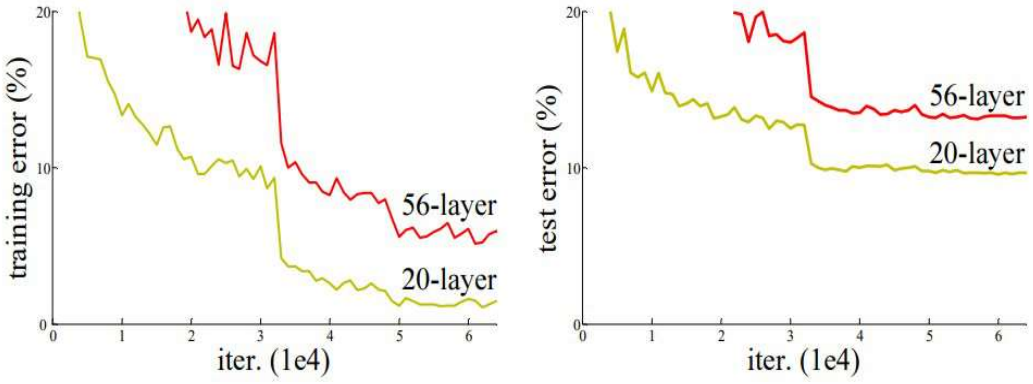

Figura 8: A sinistra l'errore sui dati di training, a destra l'errore sui dati di test sul dataset CIFAR-10 di due reti neurali da 20 e 56 *layers*[9]

Per risolvere il problema sopra citato l'architettura di ResNet introduce il concetto di *residual block,* utilizzando una tecnica chiamata *skip connections.* Questa tecnica connette l'attivazione di un *layer* con quella di alcuni *layer* successivi, saltando uno o più *layer*.

In una *neural network, l'input x* viene moltiplicato per i pesi del *layer* di riferimento a cui viene aggiunto il *bias,* ottenendo quindi*:*

$$H(x) = F(wx + b) \text{ o } H(x) = F(x)$$

Con l'introduzione delle *skip connections* abbiamo invece

$$H(x) = F(x) + x$$

La formula $F(x) + x$ può essere ottenuta con l'utilizzo di *shortcut connections,* che permettono di saltare uno o più *layer.* ResNet implementa le *shortcut connections* utilizzando la funzione identità ($f(x) = x$) aggiungendo *l'output* a quello degli altri *layer.*

[9] He Kaiming, Zhang Xiangyu, Ren Shaoqing, Sun Jian, *Deep Residual Learning for Image Recognition*, Microsoft Research, 2015, p. 1.

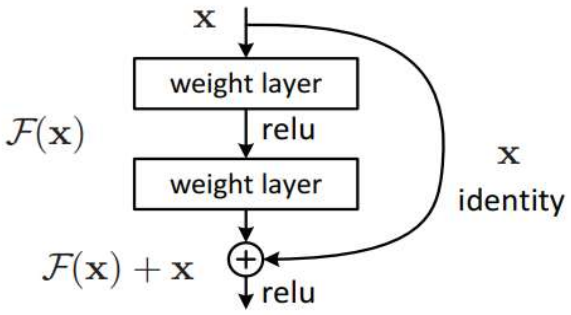

Figura 9: rappresentazione grafica di una *shortcut connection.*[10]

ResNet utilizza una rete con 34 *layer* (ispirata all'architettura della rete VGG-19) a cui sono state aggiunte le *shortcut connections.* Queste connessioni rendono la rete una *residual network.*

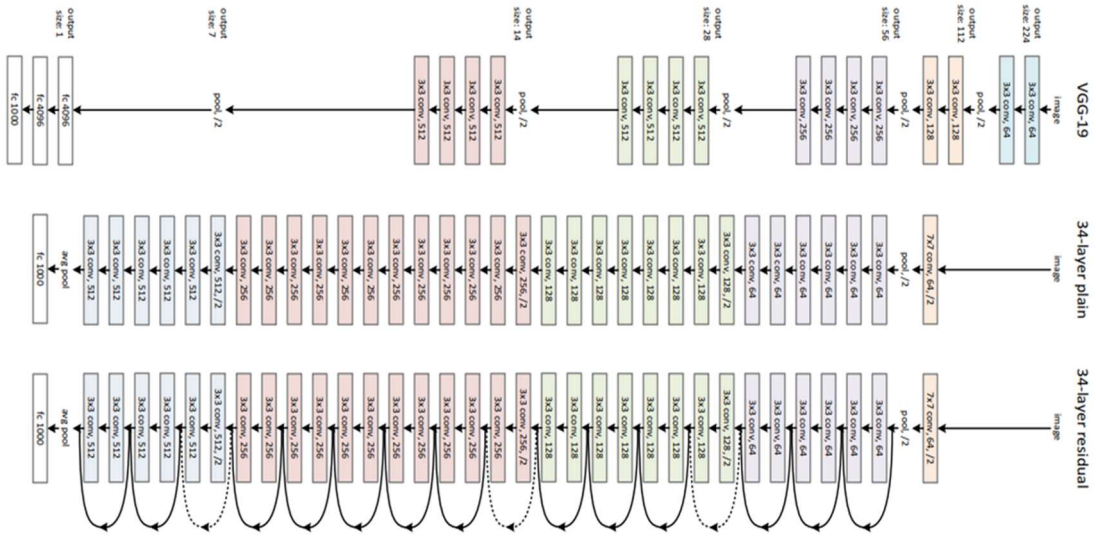

Figura 10: Architettura di ResNet a confronto con VGG-19[11].

[10] Ivi, p.2.
[11] Ivi, p.4.

3. IMPLEMENTAZIONE DI UNA CONVOLUTIONAL NEURAL NETWORK IN FLUX

3.1 Introduzione

In questo capitolo della dissertazione implementerò una *convolutional neural network* (ispirata a LeNet) usando il linguaggio di programmazione Julia nella versione 1.8.5 e Flux nella versione 0.13.12 utilizzando CIFAR-10 come *dataset* di addestramento. Utilizzerò Pluto.jl, una libreria che permette di scrivere *notebooks* reattivi in Julia. La versione di Pluto.jl utilizzata è la 0.19.22.

Infine, per importare il *dataset* CIFAR-10 utilizzerò il progetto MLDatasets.jl alla versione v0.7.9, una libreria che consente di accedere ai dataset di Machine Learning più utilizzata in maniera semplice.

3.1 Julia e Flux

Julia è un linguaggio di programmazione ad alto livello, dinamico, *open source*, pubblicato nel 2012. Utilizza il compilatore JIT (Just In Time) tramite il *framework* LLVM. È stato sviluppato appositamente per la computazione scientifica a numerica, nonostante possa essere applicato a qualunque situazione. Grazie a queste caratteristiche Julia è un linguaggio efficiente (performance simili al C) e molto utilizzato per progetti di *data mining* e *big data*.

Flux è una libreria scritta in Julia per il Machine Learning che supporta nativamente l'utilizzo di CUDA (Compute Unified Device Architecture) una piattaforma di calcolo parallelo sviluppata da NVIDIA che permette di sfruttare la potenza di calcolo delle GPU.

3.2 Il Dataset CIFAR-10

Il *database* CIFAR-10 consiste in 60.000 immagini RGB di dimensione 32x32 suddivise in 10 classi, con 6000 immagini per classe. Sono state collezionate e etichettate da Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.

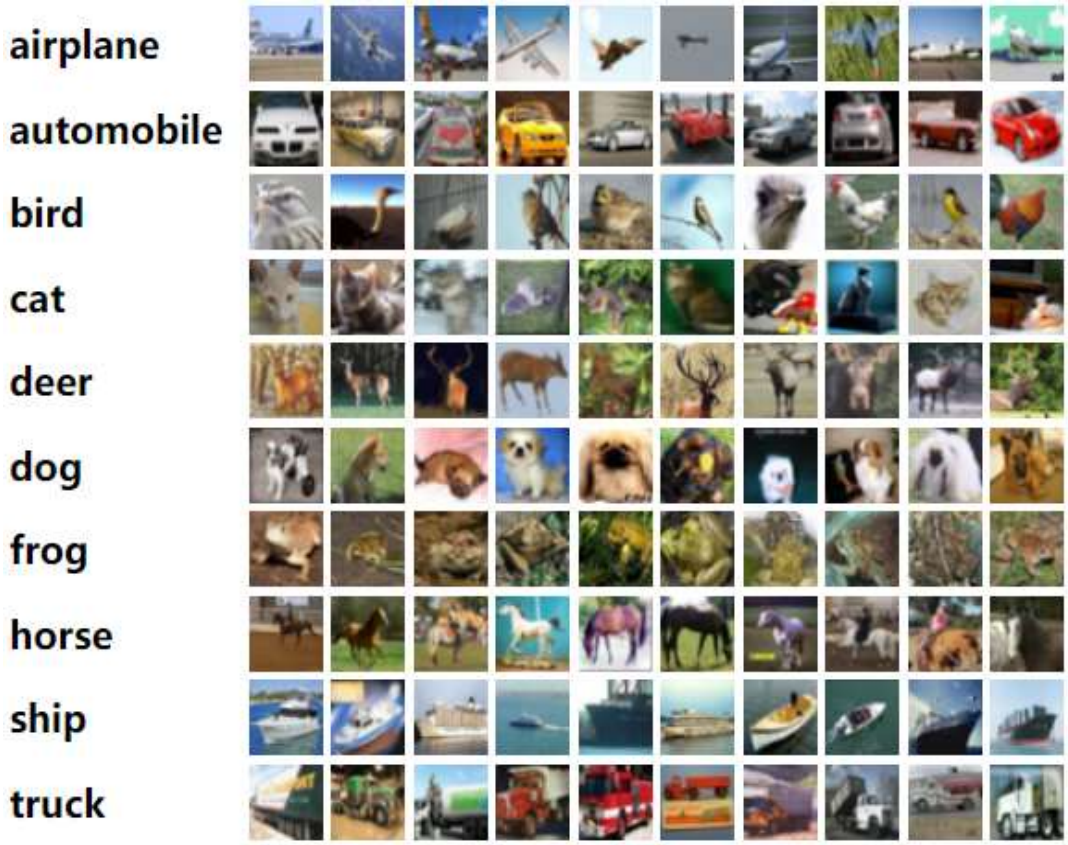

Figura 11: Esempio di immagini presenti nel database CIFAR-10.[12]

3.3 Implementazione e addestramento

Iniziamo importando le librerie necessarie e scaricando il dataset CIFAR-10.

[12] https://paperswithcode.com/dataset/cifar-10

```
begin
    using Flux ✓, MLDatasets ✓, Statistics ✓
    using Flux: onehotbatch, onecold, logitcrossentropy, params
    using Base.Iterators: partition
    using Printf ✓, BSON ✓
    using CUDA ✓
    using ImageShow ✓
    using Images ✓
    CUDA.allowscalar(true)
end
```

21.1 s

```
("truck", )
```

```
MLDatasets.CIFAR10().metadata["class_names"][MLDatasets.CIFAR10().targets[2] + 1],
convert2image(MLDatasets.CIFAR10(), 2)
```

Definiamo ora una struttura dati contenente tutti i parametri che utilizzeremo per l'addestramento: il *learning rate,* il numero di epoche per cui verrà addestrato il modello, la dimensione di ogni *batch* di dati e il percorso che verrà utilizzato per salvare automaticamente il modello.

```
Base.@kwdef mutable struct TrainArgs
    lr::Float64 = 1e-2
    epochs::Int = 100
    batch_size = 32
    savepath::String = "./"
end
```

Definiamo ora due funzioni che utilizzeremo per caricare in memoria i dati di *training* e i dati di test andando a creare poi dei *batch* che utilizzeremo per l'addestramento.

```
make_minibatch (generic function with 1 method)
```

```
function make_minibatch(X, Y, idxs)
    X_batch = Array{Float32}(undef, size(X)[1:end-1]..., length(idxs))
    for i in 1:length(idxs)
        X_batch[:, :, :, i] = Float32.(X[:,:,:,idxs[i]])
    end
    Y_batch = onehotbatch(Y[idxs], 0:9)
    return (X_batch, Y_batch)
end
```

```
get_processed_data (generic function with 1 method)
```

```
function get_processed_data(args)
    train_imgs, train_labels = MLDatasets.CIFAR10(:train)[:]
    mb_idxs = partition(1:length(train_labels), args.batch_size)
    train_set = [make_minibatch(train_imgs, train_labels, i) for i in mb_idxs]

    test_imgs, test_labels = MLDatasets.CIFAR10(:test)[:]
    test_set = make_minibatch(test_imgs, test_labels, 1:length(test_labels))

    return train_set, test_set
end
```

Definiamo ora una funzione che costruirà la *convolutional neural network.* La rete sarà formata da due *layer* di convoluzione con attivazione relu, seguiti entrambi da una funzione di max *pooling.* Avremo successivamente una *fully-connected layer* composta da quattro *layer* con rispettivamente 400, 120, 256, 84 neuroni con attivazione relu. Infine avremo il *layer* di *output* con attivazione *softmax* e dieci neuroni, corrispondenti al numero di classi presenti nel database CIFAR-10.

```
function build_model(args; imgsize=(32, 32, 3), nclasses=10)
    model = Chain(
          Conv((5,5), imgsize[end]=>16, relu),
          MaxPool((2,2)),
          Conv((5,5), 16=>8, relu),
          MaxPool((2,2)),
          x -> reshape(x, :, size(x, 4)),
          Dense(200, 120),
          Dense(120, 84),
          Dense(84, 10), softmax)
    model
end
```

Definiamo ora tre funzioni di utilità: *augment* ci permetterà di applicare del rumore casuale alle immagini al fine di evitare *overfitting*; *anynan* ci permetterà di validare ogni iterazione del modello; *accuracy* calcola l'accuratezza del modello sui dati di test.

```
accuracy (generic function with 1 method)
begin
    augment(x) = x .+ gpu(0.1f0*randn(eltype(x), size(x)))
    anynan(x) = any(y -> any(isnan, y), x)

    accuracy(x, y, model) = mean(Flux.onecold(cpu(model(x)), 0:9) .==
    Flux.onecold(cpu(y), 0:9))
end
```

Infine, scriviamo una funzione per addestrare il modello. In questa funzione caricheremo i *dataset* e il modello sulla gpu, in modo da sfruttare la sua potenza computazionale per l'addestramento. Successivamente definiremo una funzione di *loss* utilizzando la funzione di *logit cross entropy.* Andremo poi a iterare per tutte le epoche definite inizialmente per addestrare il modello. Ad ogni iterazione verrà calcolata l'accuratezza sul dataset di test e il modello verrà salvato automaticamente utilizzando il formato BSON (*Binary JavaScript Object Notation)* ogni volta che l'accuratezza migliora. Se dopo cinque iterazioni successive non abbiamo ottenuto un'accuratezza migliore aggiorniamo automaticamente il *learning rate* dividendo il valore attuale per 10.

```
function train(; kws...)
    args = TrainArgs(; kws...)

    @info("Loading data set")
    train_set, test_set = get_processed_data(args)

    @info("Building model...")
    model = build_model(args)

    # Load model and datasets onto GPU
    train_set = gpu.(train_set)
    test_set = gpu.(test_set)
    model = gpu(model)

    # Pre-compile the model
    model(train_set[1][1])

    # define the loss function
    function loss(x, y)
        x̂ = augment(x)
        ŷ = model(x)
        return crossentropy(ŷ, y)
    end

    opt = Momentum(args.lr)

    @info("Beginning training loop...")
    best_acc = 0.0
    last_improvement = 0
    for epoch_idx in 1:args.epochs
        Flux.train!(loss, params(model), train_set, opt)

        if anynan(Flux.params(model))
            @error "NaN params"
            break
        end

        @info("Computing accuracy...")
        acc = accuracy(test_set..., model)

        @info(@sprintf("[%d]: Test accuracy: %.4f", epoch_idx, acc))
        if acc >= 0.999
            @info(" -> Early-exiting: We reached our target accuracy of 99.9%")
            break
        end

        # If this is the best accuracy we've seen so far, save the model out
        if acc >= best_acc
            @info(" -> New best accuracy! Saving model out to cifar_10.bson")
            BSON.@save joinpath(args.savepath, "cifar_10.bson") params = cpu.
(params(model)) epoch_idx acc
            best_acc = acc
            last_improvement = epoch_idx
        end

        # drop out the learning rate if no improvement
        if epoch_idx - last_improvement >= 5 && opt.eta > 1e-6
            opt.eta /= 10.0
            @warn(" -> Haven't improved in a while, dropping learning rate to
$(opt.eta)!")

            last_improvement = epoch_idx
        end

        if epoch_idx - last_improvement >= 10
            @warn(" -> model converged.")
            break
        end
    end
end
```

Invochiamo la funzione *train()* per addestrare il modello.

```
train()
```

```
Loading data set
Building model...
Beginning training loop...
100%
Computing accuracy...
[1]: Test accuracy: 0.4720
 -> New best accuracy! Saving model out to cifar_10.bson
100%
Computing accuracy...
[2]: Test accuracy: 0.5208
 -> New best accuracy! Saving model out to cifar_10.bson
100%
Computing accuracy...
[3]: Test accuracy: 0.5294
```

Per poter utilizzare il modello addestrato ci basterà caricare i parametri dal file BSON salvato automaticamente durante l'addestramento e effettuare una predizione su nuovi dati.

```
test (generic function with 1 method)
```

```
function test(; kws...)
    args = TrainArgs(; kws...)

    _,test_set = get_processed_data(args)

    model = build_model(args)

    BSON.@load joinpath(args.savepath, "cifar_10.bson") params

    Flux.loadparams!(model, params)

    test_set = gpu.(test_set)
    model = gpu(model)
    @show accuracy(test_set...,model)
end
```

3.4 Performance di un modello di classificazione

Per valutare il modello addestrato ho utilizzato l'accuratezza, una delle metriche più comunemente utilizzate. Essa rappresenta la percentuale di campioni di test correttamente classificati rispetto al totale dei campioni di test. Il modello ha raggiunto un'accuratezza del 65% con un addestramento di cento epoche.

In alcuni casi, specialmente quando le classi sono sbilanciate, l'accuratezza non è la metrica più accurata da utilizzare. In queste situazioni si utilizzano metriche come precisione, richiamo e *F1 score.* La precisione indica la percentuale di campioni positivi classificati correttamente dal modello rispetto al totale dei campioni positivi predetti dal modello. Il richiamo indica la percentuale di campioni positivi correttamente identificati dal modello rispetto al totale dei campioni positivi presenti nel set di dati. L'*F1 score* rappresenta la media armonica tra precisione e richiamo.

La matrice di confusione è un altro strumento importante per valutare le performance di un modello. La matrice di confusione visualizza il numero di campioni classificati correttamente ed erroneamente dal modello per ogni classe presente nel set di dati di test. È importante perché consente di valutare in modo dettagliato le performance del modello per ogni classe presente nel set di dati, individuando eventuali errori o problemi di classificazione che potrebbero influire negativamente sulle performance del modello.

Attualmente, i migliori modelli di CNN per la classificazione di oggetti possono raggiungere un'accuratezza superiore al 99%. Ad esempio, il modello ResNet-50, che utilizza una rete a 50 strati convoluzionali, ha raggiunto un'accuratezza del 99,2% sul dataset di classificazione di immagini CIFAR-10.

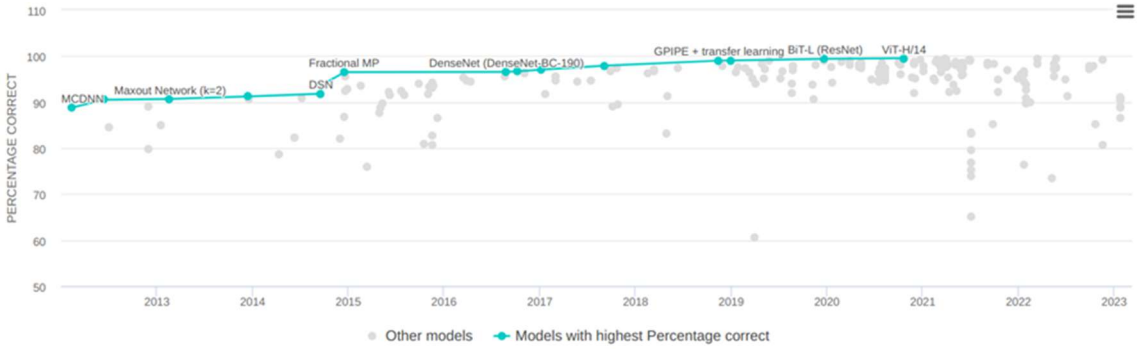

Figura 12: Accuratezza dei modelli sul dataset CIFAR-10.[13]

[13] https://paperswithcode.com/sota/image-classification-on-cifar-10

CONCLUSIONE

In conclusione, le *Convolutional Neural Network* hanno dimostrato di essere una tecnologia essenziale per molti compiti di visione artificiale, analisi di immagini e altri problemi di elaborazione di dati con strutture spaziali. Questi modelli sono stati utilizzati con successo in molte applicazioni, come la classificazione di immagini, il riconoscimento facciale, la segmentazione dell'immagine e la generazione di immagini. La loro capacità di gestire dati complessi con strutture spaziali li rende ideali per molte applicazioni future in una vasta gamma di settori. Con l'evoluzione continua delle tecnologie di *deep learning*, ci aspettiamo che le *Convolutional Neural Network* continueranno a giocare un ruolo sempre più importante nello sviluppo dell'Intelligenza Artificiale.

BIBLIOGRAFIA

- Cireşan Dan C., Meier Ueli, Masci Jonathan, Gambardella Luca M., Schmidhuber Jürgen, *High-Performance Neural Networks for Visual Object Classification,* Technical Report No. IDSIA-01-11 2011.
- Goodfellow Ian, Bengio Yoshua, Courville Aaron, *Deep Learning,* MIT Press 2016.
- He Kaiming, Zhang Xiangyu, Ren Shaoqing, Sun Jian, *Deep Residual Learning for Image Recognition,* Microsoft Research, 2015.
- Hinton G. E., Srivastava N., Krizhevsky A., Sutskever I., Salakhutdinov R. R., *Improving neural networks by preventing co-adaptation of feature detectors,* Department of Computer Science, University of Toronto 2012.
- Krizhevsky Alex, Sutskever Ilya, Hinton Geoffrey E., *ImageNet Classification with Deep Convolutional Neural Networks,* University of Toronto, 2012.
- Minsky Marvin, Papert Seymour A., *Perceptrons: An Introduction to Computational Geometry*, The MIT Press, Cambridge, Massachusetts, and London, England 1988.
- Nielsen Michael, *Neural Networks and Deep Learning*, 2019, http://neuralnetworksanddeeplearning.com/
- O'Shea Keiron, Nash Ryan, *An Introduction to Convolutional Neural Networks.* Department of Computer Science, Aberystwyth University, Ceredigion 2015.
- Redmon Joseph, Divvala Santosh, Girshick Ross, Farhadi Ali, *You Only Look Once: Unified, Real-Time Object Detection,* University of Washington Allen Institute for AI, Facebook AI Research 2015.
- Rumelhart, D., Hinton, G. & Williams, R. *Learning representations by back-propagating errors* in Nature 323, 533–536 1986.

SITOGRAFIA

- https://www.ibm.com/it-it/cloud/learn/neural-networks
- https://playground.tensorflow.org/
- https://julialang.org/
- https://fluxml.ai
- https://github.com/JuliaML/MLDatasets.jl
- https://paperswithcode.com/sota/image-classification-on-cifar-10

RINGRAZIAMENTI

Vorrei inoltre ringraziare la mia amica Berecca per avermi aiutato a superare gli esami, mio fratello Jacopo per non aver fatto niente e i miei genitori per il regalo.