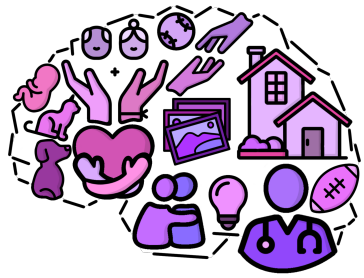
UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INFORMATICA



Corso di Fondamenti di Intelligenza Artificiale

Esistere

Professore:
Ch.mo Prof.
Fabio Palomba

Studenti:
Antonio D'Auria
Mat. 0512114149
Luca Casillo
Mat. 0512113696
Maria Giovanna Della Pietra
Mat. 0512114221
Rosa Carotenuto
Mat. 0512113246

ANNO ACCADEMICO 2023/2024

Indice

Elenco delle figure

1 INTRODUZIONE

1.1 Sistema Attuale

Attualmente, per scoprire se una persona soffre di Alzheimer o meno, sono necessari diverse analisi e controlli riguardanti il corpo e il comportamento. Ad oggi, non esiste una cura per l'Alzheimer, ma solo trattamenti palliativi per provare a rallentare il progresso della stessa. Il progetto *Esistere* nasce dalla necessità di fornire assistenza ai medici durante le prime fasi della diagnosi del morbo di Alzheimer e non di sostituire l'attuale iter diagnostico. Il codice sarà reperibile sulla repository di GitHub.

1.2 Obiettivo

Lo scopo del progetto è sviluppare un modulo di Intelligenza Artificiale basato su un modello di Deep Learning che, analizzando delle immagini TAC, sarà in grado di decretare la presenza di un caso d'Alzheimer. In ambito medico la TAC è una fonte affidabile ai fini della diagnosi precoce, tuttavia in alcuni casi è necessaria ma non sufficiente per stabilire la presenza di Alzheimer.

1.3 Analisi del problema

Il problema della predizione dell'Alzheimer tramite TAC può essere trattato come un problema di classificazione multi-classe o di classificazione binaria.

Classificazione Binaria Nella classificazione binaria, l'obiettivo è prevedere due classi (Demented vs Non Demented). Qui, si utilizza tipicamente la *Binary Cross-Entropy Loss* come funzione di perdita.

Classificazione Multiclasse Nella classificazione multiclasse, l'obiettivo è prevedere due o più classi. La Cross-Entropy Loss è comunemente usata in questo contesto.

Si è scelto di utilizzare una classificazione binaria per i seguenti motivi:

- **Facilità di modellazione**: La classificazione binaria tende a essere più semplice da implementare e ottimizzare.
- **Struttura del modello**: Richiede una struttura di rete più semplice, spesso con un singolo nodo di output e una funzione *sigmoidea* (come utilizzato all'interno del progetto) rendendo il processo di addestramento meno complesso.
- **Facilità di Interpretazione**: L'output del modello, essendo la probabilità di appartenenza a una classe, è diretto e facile da interpretare, il che è cruciale in ambiti come la diagnostica medica dove le decisioni devono essere chiare e giustificabili.

N.B. Un'analisi dettagliata della classificazione, della funzione di attivazione e della funzione di perdita verrà introdotta nelle sezioni successive.

1.4 Tecnologie utilizzate

Le tecnologie utilizzate per lo sviluppo di questo progetto sono:

- **AlexNet**: Una rete neurale composta da otto layer, di cui cinque convoluzionali seguiti da layer di normalizzazione e pooling, mentre gli ultimi tre sono fully connected. Permette l'utilizzo di tecniche di regolarizzazione come il dropout e la parallelizzazione su GPU.
- **PyTorch**: Una libreria open source di machine learning in Python, sviluppata principalmente da Facebook's AI Research lab. Offre ampie capacità per la creazione e l'addestramento di reti neurali profonde con un forte supporto per il calcolo su GPU. PyTorch è noto per la sua flessibilità e la sua interfaccia intuitiva, che facilita la sperimentazione e lo sviluppo di prototipi di modelli complessi di deep learning.

- **OpenCV**: Libreria open-source di computer vision che offre diverse funzionalità per la manipolazione di immagini, la calibrazione, etc.
- **NumPy**: Libreria utilizzata per la manipolazione di array e matrici multidimensionali in Python.
- **PIL (Python Imaging Library)**: Libreria per il trattamento delle immagini, fornisce funzioni per aprire, manipolare e salvare diversi formati di immagine.
- **Scikit-learn**: Libreria che offre strumenti semplici ed efficienti per l'analisi dei dati e la modellazione predittiva e include diversi algoritmi per la classificazione, regressione, clustering, riduzione della dimensionalità e la validazione del modello. All'interno del progetto è stata utilizzata per il calcolo delle metriche.
- **Jupyter**: Utilizzato per testare rapidamente frammenti di codice, eseguire esperimenti e visualizzare i risultati in tempo reale.

Ciò su cui ci soffermeremo maggiormente è l'utilizzo fatto del framework PyTorch [1] e della rete neurale AlexNet [5].

1.4.1 PyTorch. PyTorch rappresenta uno dei framework più utilizzati per la creazione di modelli di deep learning. Nel contesto scientifico, è utilizzato non solo dai più esperti, ma anche da chi sta avendo un primo approccio a tale ambito.

Le caratteristiche che contraddistinguono il framework sono:

- **Intuitività**: offre un'API intuitiva e flessibile che facilita la comprensione e l'implementazione di modelli di deep learning. La sua interfaccia è progettata per permettere di esprimere le idee in codice in modo quasi naturale.
- **Autograd**: è una delle caratteristiche fondamentali delle librerie di deep learning, cruciale per l'apprendimento automatico. Autograd rende possibile il calcolo dei gradienti offrendo un modo per calcolarli automaticamente con un processo chiamato *differenziazione automatica*. La differenziazione avviene *registrando* le operazioni sulle variabili in un grafo dove ogni nodo rappresenta un tensore e ogni arco rappresenta un'operazione che trasforma i nodi di input in output.
  Una volta ottenuto l'output si effettua il calcolo della perdita: Autograd "ritorna indietro" attraverso il grafo per calcolare i gradienti.
  Questo processo si chiama *backpropagation*. Partendo dalla funzione di perdita, Autograd usa la chain rule del calcolo differenziale per propagare i gradienti indietro attraverso il grafo, fino ai nodi di input. Così si effettua la differenziazione automatica.
- **Supporto per GPU**: permette un facile utilizzo delle GPU per accelerare i calcoli, rendendo l'allenamento dei modelli più veloce. Il passaggio da CPU a GPU è semplice e può essere fatto con poche modifiche al codice.

1.4.2 Tensori. Nel cuore di PyTorch e di ogni framework di deep learning ci sono i tensori. I tensori sono fondamentali per la creazione e l'addestramento dei modelli di deep learning per diverse ragioni chiave.

Un tensore è una generalizzazione di scalari, vettori e matrici a dimensioni superiori.

In PyTorch, i tensori sono utilizzati per incapsulare i dati di input, output, e i parametri interni dei modelli, come pesi e bias. I tensori possono variare in dimensione: un tensore a 0 dimensioni rappresenta uno scalare, un tensore a 1 dimensione un vettore, un tensore a 2 dimensioni una matrice.

Ci sono varie motivazioni per cui i tensori vengono utilizzati, le principali sono:

- **Manipolazione di grandi volumi di dati**: i tensori sono ottimizzati per gestire grandi volumi di dati, consentendo operazioni matematiche efficienti.
- **Autograd**: i tensori sono integrati con il sistema Autograd di PyTorch

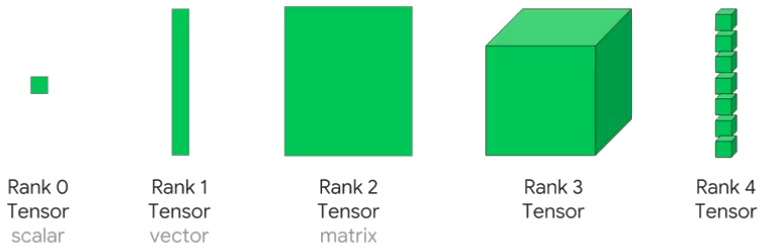


Fig. 1.1. Tensori a diverse dimensioni

1.4.3 Alexnet. AlexNet rappresenta una svolta nel panorama del deep learning. È stata progettata dai dottorandi Alex Krizhevsky e Ilya Sutskever, supervisionati da Geoffrey Hinton.

È composta di otto layer, ognuno dei quali gioca un ruolo fondamentale nella classificazione di immagini. Abbiamo:

- **Input Layer** Riceve i valori grezzi dei pixel delle immagini. L'AlexNet originale prende in input immagini RGB di 227x227 pixel.
  La rete si compone poi di cinque layer convoluzionali, layer di normalizzazione e tre layer fully connected.
  I layer convoluzionali sono il cuore delle reti neurali convoluzionali (CNNs). Questi strati svolgono un ruolo cruciale nell'elaborazione e nell'estrazione automatica delle caratteristiche dalle immagini. L'estrazione delle caratteristiche dominanti delle immagini avviene tramite i kernel, o filtri, ovvero una matrice di pesi utilizzata per eseguire operazioni di convoluzione su dati di input.
  La grandezza dell'output è controllata da tre parametri:
  - **Depth**: corrisponde al numero di filtri usati per le operazioni convoluzionali.
  - **Stride**: lo stride è il numero di pixel con cui viene effettuato lo scorrimento della matrice filtro sulla matrice di input.
  - **Zero-Padding**: aggiunge gli zero attorno al bordo.

  L'applicazione di ogni filtro produce una *feature map* separata che rappresenta le posizioni spaziali di una particolare caratteristica nei dati di input.

- **Convolutional Layer**: AlexNet, si compone di cinque livelli convoluzionali:
  - **CONV1**: il primo strato filtra le immagini 227x227x3 con 96 kernel di taglia 11x11x3 e con uno stride di 4 pixel.
  - **CONV2**: il secondo strato di convoluzione filtra le immagini con 256 kernel di grandezza 5x5x48.
  - **CONV3**: il terzo strato filtra le immagini con 384 kernel di grandezza 3x3x256.
  - **CONV4**: il quarto strato filtra le immagini con 384 kernel di grandezza 3x3x192.

  A seguito dei livelli convoluzionali ci sono i *Pooling Layer*.
  La mappa delle caratteristiche viene sotto-campionata utilizzando un'operazione di pooling che riduce la dimensione spaziale della mappa delle caratteristiche mantenendo le caratteristiche importanti. Le operazioni di pooling comuni includono il *Max pooling* e l'*Average pooling*.
  Nel caso di AlexNet viene utilizzato il Max Pooling dopo il primo, secondo e quinto livello convoluzionale. Il Max Pooling seleziona il valore massimo all'interno di una finestra definita sulla feature map. Questo valore massimo viene poi usato per rappresentare l'intera regione. Attraverso la selezione del valore massimo si tende a catturare le caratteristiche più prominenti o distintive all'interno della finestra di pooling. Questo può aiutare a rendere la rete più invariante alle piccole traslazioni e distorsioni nell'input.
  Inoltre, dopo ogni strato convoluzionale viene applicata la *ReLu Appplication Function*: è una funzione di attivazione, fornisce le proprietà non-lineari alla rete. La funzione ReLu è definita come:

$$f(x) = \max(0, x) \tag{1.1}$$

  L'output delle operazioni convoluzionali e di pooling subisce un'operazione di flattening prima di essere passato ai layer fully connected: l'output viene "appiattito" garantendo un input unidimensionale.
  Arriviamo quindi agli strati *fully-connected*. In uno strato fully connected ogni neurone riceve input da tutti i neuroni presenti nello strato precedente e produce un output basato sul proprio set di learnable weights.
  In Alexnet, nello specifico, abbiamo 3 layer fully-connected:
  - **FC6 e FC7**: composti ognuno da 4096 neuroni. Servono a trasformare le rappresentazioni spaziali apprese dagli strati convoluzionali e di pooling in rappresentazioni ad alto livello che possono essere utilizzate per la classificazione.
  - **FC8**: rappresenta l'ultimo strato fully connected e funge da strato di output per la rete. Questo strato è cruciale per la classificazione finale delle immagini, trasformando le rappresentazioni ad alto livello apprese dagli strati precedenti in probabilità di appartenenza alle classi target. Quindi, è il diretto responsabile della decisione finale di classificazione. L'output di questo layer viene passato ad una funzione *softmax* che permette di interpretare i risultati come distribuzione di probabilità sulle classi possibili.

Softmax viene tipicamente utilizzata come strato di output per convertire i *logit* (i valori raw prodotti dagli strati precedenti della rete) in probabilità che sommano a 1. Questo permette di interpretare direttamente l'output della rete come una distribuzione di probabilità sulle classi possibili.

$$Softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{1.2}$$

Dove:

e^{z_i} = esponenziale del logit z_i per la classe i
$\sum_{j=1}^{K} e^{z_j}$ = somma degli esponenziali di tutti i logit nel vettore z

Per concludere è qui presentata un'immagine rappresentativa dell'architettura AlexNet.

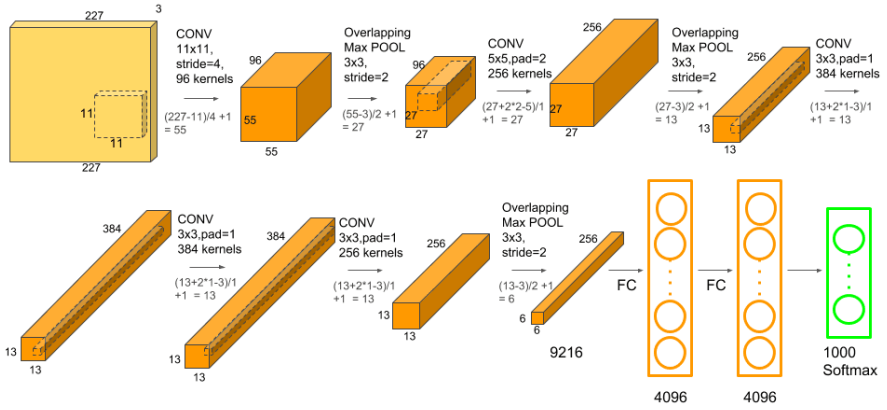


Fig. 1.2. Architettura AlexNet

1.5 Specifica PEAS

Un ambiente è essenzialmente un'istanza di un problema di cui gli agenti razionali rappresentano le soluzioni, in questo caso l'ambiente è un sistema di diagnosi medica.

Un ambiente viene generalmente descritto tramite la formulazione PEAS, ovvero Performance, Environment, Actuators, Sensors.

P. Misura di prestazione adottata per valutare l'operato di un agente.
E. Descrizione degli elementi che formano l'ambiente.
A. Gli attuatori disponibili che l'agente ha a disposizione per intraprendere le azioni.
S. I sensori attraverso i quali riceve gli input percettivi.

Di seguito è riportata la specifica PEAS del progetto.

- **Performance:** La misura di prestazione utilizzata per valutare l'operato dell'agente è la categorizzazione corretta delle immagini.
- **Environment:** L'ambiente in cui opera il sistema è costituito da immagini di TAC di pazienti, acquisite da dispositivi medici appropriati come scanner CT.
- **Actuators:** Gli attuatori del sistema includono CNN, tecniche di apprendimento automatico e sistemi di classificazione per analizzare e categorizzare le immagini di TAC.
- **Sensors:** I sensori del sistema sono dispositivi di acquisizione di immagini TAC fornite in input.

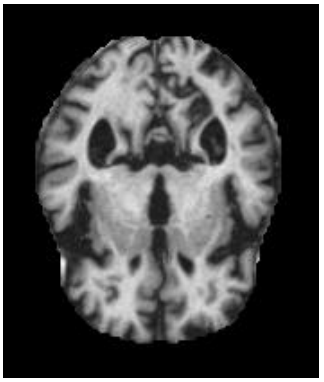
1.6 Caratteristiche dell'ambiente

- **Parzialmente osservabile:** Non tutte le informazioni possono essere catturate da una singola TAC.
- **Stocastico:** Possono esserci elementi di incertezza nella diagnosi, in quanto la qualità dell'immagine e la variabilità biologica possono influenzare i risultati.
- **Sequenziale:** Ogni diagnosi è un episodio a sé stante, ma le informazioni apprese sono utilizzabili per migliorare il sistema nel tempo.
- **Dinamico:** Le condizioni del paziente possono cambiare, di conseguenza si possono rendere disponibili nuove informazioni che possono richiedere rielaborazioni o nuove analisi.
- **Discreto**: La TAC è un dato discreto e il sistema elabora le singole scansioni indipendentemente.
- **Singolo-Agente:** L'ambiente in cui opera l'agente non prevede la presenza di altri agenti che possano ostacolarlo o aiutarlo durante la predizione della presenza di Alzheimer.

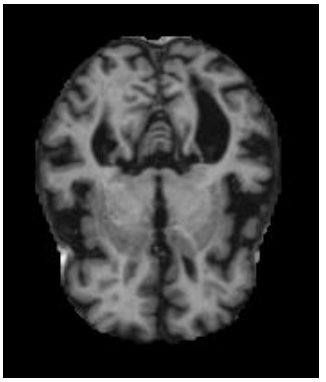
2 ANALISI DEI DATI

2.1 Introduzione al Dataset

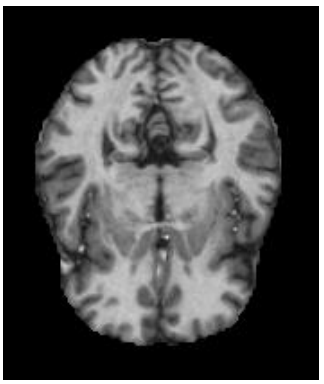
Il dataset sul morbo dell'Alzheimer [6] è stato reperito dalla piattaforma open-source Kaggle ed è non strutturato. Consiste di 6400 immagini di TAC suddivise in quattro classi: Moderate Demented (MOD), Mild Demented (MID), Very Mild Demented (VMD) e Non Demented (ND); le dimensioni delle immagini sono di 176×208 pixel. L'80% del dataset è usato per l'apprendimento mentre il restante 20% per la fase di test.



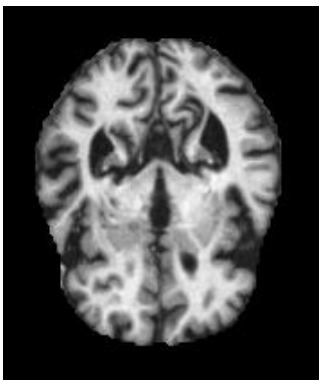
(a) Moderate Demented



(b) Mild Demented



(c) Very Mild Demented



(d) Non Demented

Fig. 2.1. Le quattro classi presenti nel dataset

2.2 Tipo di classificazione

Il Dataset presenta quattro classi differenti, tuttavia, come accennato nella sezione 1.3 stiamo trattando un problema di classificazione binaria, l'obiettivo è distinguere tra la presenza e l'assenza di Alzheimer, indipendentemente dallo stadio specifico. Per ovviare a ciò si è scelto di unire le categorie MID, MOD e VMD in un'unica categoria Demented, lasciando inalterata la categoria Non Demented. Il risultato è un Dataset già bilanciato che presenta 2561 immagini per l'etichetta Demented e 2560 per l'etichetta Non Demented, per un totale di 5121 immagini per il training.

2.2.1 Funzione di perdita. Per guidare il processo di addestramento del modello viene utilizzata una funzione di perdita. La funzione di perdita quantifica la differenza tra il valore predetto dal modello e il valore reale osservato. Essa rappresenta una delle principali valutazioni di prestazione e, naturalmente, l'obiettivo è minimizzarla.

Essendo il modello di classificazione di tipo binario (Dove 1 rappresenta la classe Demented e 0 la classe Non Demented) si è scelto di utilizzare la funzione di perdita *Binary Cross-Entropy Loss (BCE Loss)* [2].

BCE Loss Questa funzione di perdita è utilizzata quando le etichette del modello sono binarie (0 o 1). BCE Loss misura la distanza tra le probabilità previste e le etichette effettive.

$$BCELoss = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)] \tag{2.1}$$

Dove:

N = numero di esempi
y_i = etichetta reale dell'i-esimo esempio
$\hat{y}_i$ = probabilità prevista dell'i-esimo elemento di appartenere alla classe 1

Dalla definizione della BCE Loss si nota quindi che essa è sensibile non solo alla correttezza della classificazione (es., se la classe predetta è giusta o sbagliata) ma anche a quanto il modello è sicuro della sua previsione. Questo aiuta a guidare il modello verso previsioni più accurate e confidenti. [2.1]
Codice:

```
# nn è un modulo di PyTorch
criterion = nn.BCELoss();
```

[2.1] Se l'etichetta reale y è 1 e il modello predice $\hat{y}$ vicino a 1, la perdita sarà bassa e il modello avrà fatto una buona previsione. Se y è 1 ma il modello predice $\hat{y}$ vicino a 0 la perdita sarà alta a causa del termine $\log(\hat{y})$. (Discorso analogo con $y = 0$, $\hat{y}_i = 0 \vee 1$ dove la perdita sarà molto dipendente da $\log(1 - \hat{y})$)

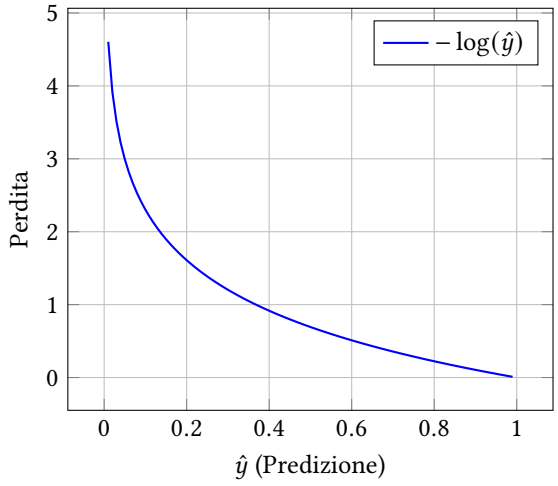


Fig. 2.2. BCELoss per $y = 1$

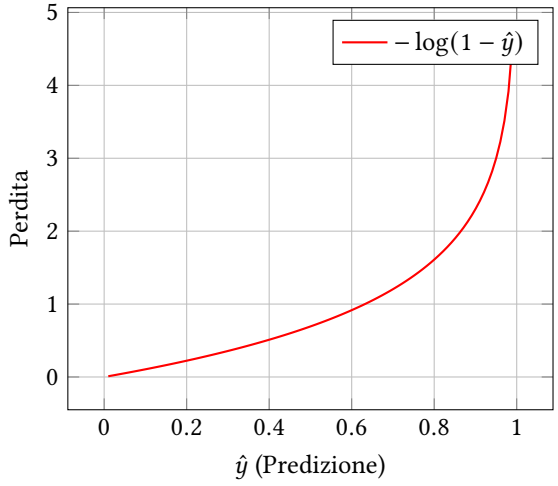


Fig. 2.3. BCELoss per $y = 0$

2.2.2 Funzione di attivazione. Infine, l'ultima scelta inerente all'utilizzo di classificazione binaria è l'uso di una funzione di attivazione. La funzione di attivazione è una funzione che viene applicata all'output di un neurone o di un layer della rete, essa permette di introdurre la non-linearità (così da modellare relazioni non lineari) nel processo di apprendimento, permettendo così alla rete neurale di apprendere e rappresentare complessi pattern nei dati. La funzione di attivazione utilizzata è stata la funzione sigmoide.

Funzione Sigmoide l'output del modello proviene da un singolo neurone che attraverso tale funzione mappa l'output a un valore di probabilità tra 0 e 1. Se l'output x sarà $> 0,5$ la classificazione sarà "Non Demented", viceversa se $x \leq 0,5$ la classificazione sarà "Demented". La funzione sigmoide è definita come:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

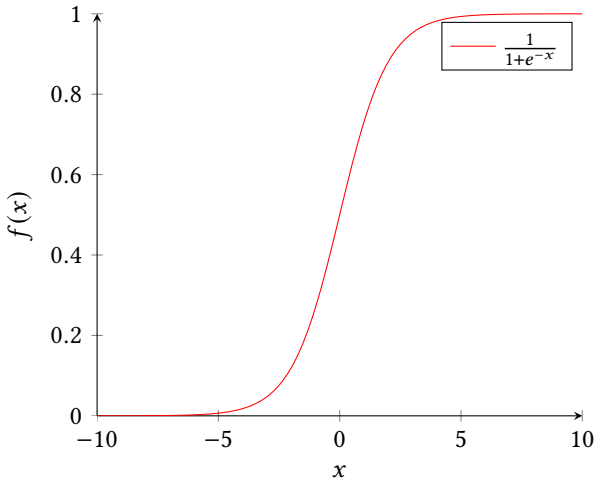


Fig. 2.4. Grafico della funzione sigmoide

Codice:

```
# Aggiunta della funzione di attivazione sigmoide all'ultimo strato
alexnet.classifier.add_module("7", nn.Sigmoid())
```

2.3 Data Augmentation

L'Augmentation è una tecnica utilizzata per aumentare la quantità e la varietà dei dati di addestramento attraverso l'applicazione di trasformazioni che mantengono inalterata l'etichetta dei dati.

Nella seguente sottosezione verranno spiegate le tecniche di Augmentation utilizzate. Come riportato nella sottosezione 2.2 il Dataset presenta un totale di 5121 immagini che risultano più che sufficienti per procedere al training. Nonostante ciò, effettuando vari tentativi di training con Dataset normale e Dataset soggetto ad Augmentation si è riscontrato che l'Augmentation ha riportato un leggero miglioramento al modello, è stato quindi deciso di mantenere tale approccio.

Tramite tale procedura si è raggiunto un totale di 6000 immagini, 3000 per l'etichetta Demented e 3000 per l'etichetta Non Demented.

Le trasformazioni di Augmentation utilizzate sono le seguenti:

- **Flip Orizzontale casuale**: L'immagine può essere capovolta orizzontalmente con una probabilità del 50%, determinata dalla scelta casuale tra True e False.
- **Rotazione casuale**: L'immagine viene ruotata di un angolo casuale compreso tra -10 e 10 gradi. Questo viene fatto calcolando una matrice di rotazione attorno al centro dell'immagine.

Di seguito è riportata la definizione della funzione utilizzata per applicare l'Augmentation alle immagini:

```
def augment_image(input_image_path, output_image_path):
    image = cv2.imread(input_image_path)
    # Random flip
    flip_hor = random.choice([True, False])
    if flip_hor:
        image = cv2.flip(image, 1)  # Flip orizzontale
    # Random rotation
    angle = random.randint(-10, 10)
    rows, cols, _ = image.shape
    M = cv2.getRotationMatrix2D((cols / 2, rows / 2), angle, 1)
    image = cv2.warpAffine(image, M, (cols, rows))
    # Salva l'immagine augmentata
    cv2.imwrite(output_image_path, image)
```

2.4 Trasformazioni applicate

Le trasformazioni vengono utilizzate per effettuare una pre-elaborazione delle immagini prima di utilizzarle in un modello di Deep Learning in PyTorch [1]. Le trasformazioni utilizzate, oltre l'Augmentation, mirano a preparare le immagini in un formato adeguato e ottimale per l'input a una rete neurale, in questo caso specifico, AlexNet [5], ma con una particolarità: la conversione delle immagini in scala di grigi.

Vediamo in dettaglio le trasformazioni applicate:

- **transforms.Resize((227, 227))**: È stato effettuato un ridimensionamento delle immagini in dimensione di 227×227 pixel in maniera tale da adattarle al modello pre-addestrato.
- **transforms.Grayscale(num_output_channels=1)**: Grayscale è stato utilizzato per convertire le immagini in scala di grigi in maniera tale da ridurre il numero di canali da 3 (RGB) a 1 (intensità di grigio). Non è necessario avere 3 canali poiché stiamo utilizzando immagini di TAC.

- **transforms.ToTensor()**: Converte le immagini PIL (Python Imaging Library) o array di NumPy in tensori PyTorch, scalando i valori dei pixel nell'intervallo $[0, 1]$. Questo passaggio è necessario perché PyTorch lavora con dati sotto forma di tensori.
- **transforms.Normalize(mean=[0.485], std=[0.229])**: Normalizza i tensori delle immagini sottraendo la media e dividendo per la deviazione standard, canale per canale. In questo caso, poiché le immagini sono state convertite in scala di grigi, si utilizza un solo valore per la media e uno per la deviazione standard, anziché i tre valori tipicamente utilizzati per immagini RGB.

2.4.1 Normalizzazione dei tensori. Tra i vari punti, la normalizzazione dei tensori richiede ulteriori chiarimenti.

La deviazione standard è una misura statistica che quantifica la dispersione o la variabilità dei dati in un insieme. Indica quanto i valori in un insieme di dati si discostano dalla media (o valore medio) dell'insieme stesso. Una deviazione standard bassa significa che i dati sono raggruppati vicino alla media, mentre una deviazione standard alta indica che i dati sono più distribuiti e quindi ci sono valori che si discostano di più dalla media. Tale processo di normalizzazione (o standardizzazione) mira a modificare le caratteristiche delle immagini in modo che la distribuzione dei loro pixel abbia una media (μ) di 0 e una deviazione standard (σ) di 1. Questo viene fatto per ogni canale di colore separatamente.

Il processo segue la formula dello Z-score:

$$Z = \frac{X - \mu}{\sigma} \tag{2.3}$$

Dove:

Z = il valore normalizzato
X = valore originale del pixel
μ = media dei valori dei pixel
σ = deviazione standard dei valori dei pixel

Il processo di normalizzazione, oltre a facilitare la convergenza dell'algoritmo, serve anche a prevenire il bias nei pesi: aiuta a prevenire che alcune caratteristiche dominino l'addestramento del modello a causa della loro scala. Senza normalizzazione, caratteristiche con valori più grandi potrebbero influenzare sproporzionatamente l'addestramento.

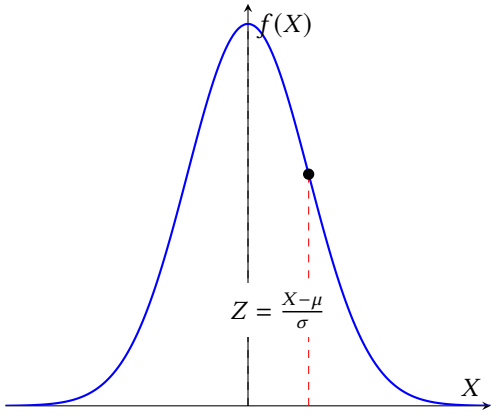


Fig. 2.5. Distribuzione normale standardizzata con $\mu = 0, \sigma = 1$

2.5 Funzione di ottimizzazione

Le funzioni di ottimizzazione, note anche come algoritmi di ottimizzazione o ottimizzatori, hanno il compito di minimizzare (o in alcuni casi massimizzare) una funzione obiettivo, che solitamente è la funzione di perdita del modello.

L'obiettivo principale nel processo di addestramento di una rete è l'ottimizzazione. Questo processo si affida a tre concetti chiave: il gradiente, la backpropagation e il Gradient Descent, un algoritmo di ottimizzazione.

Iniziamo col parlare del *gradiente*. In matematica, un gradiente è un vettore che indica la velocità e la direzione del maggior aumento della funzione. Nel contesto in cui ci troviamo, è definito dalla derivata parziale della funzione di perdita rispetto a ciascun peso della rete. Il gradiente è inteso come una guida per ottimizzare i pesi di una rete neurale, indicando la direzione in cui dovremmo muoverci per ridurre il valore della funzione di perdita.

Passiamo adesso a definire il processo di *backpropagation*. Consente di aggiornare i pesi della rete in modo efficiente, minimizzando la differenza tra l'output previsto dal modello e l'output reale (etichette). La backpropagation si basa su concetti di calcolo differenziale, in particolare sulla chain rule, per calcolare i gradienti della funzione di perdita rispetto a ogni peso nella rete.

La procedura può essere divisa in varie fasi:

- **Forward Pass**: Durante questo passaggio, l'input viene trasformato attraverso vari strati della rete, utilizzando i pesi attuali, fino a generare un output. Questo output è poi confrontato con l'etichetta reale per calcolare il valore della funzione di perdita, che misura l'errore del modello.
- **Calcolo del Gradiente della Funzione di Perdita**: Si calcola il gradiente della funzione di perdita rispetto all'output della rete.
- **Propagazione all'indietro degli errori**: Utilizzando la chain rule, si calcolano i gradienti dei pesi di ogni strato Questo sta ad indicare quanto ogni singolo peso all'interno della rete contribuisca all'errore totale. L'errore, quindi, viene propagato all'indietro attraverso la rete, strato per strato, partendo dallo strato di output fino ad arrivare agli strati di input.
- **Aggiornamento dei Pesi**: I pesi vengono aggiornati in direzione opposta ai gradienti calcolati, utilizzando un algoritmo di discesa del gradiente o una delle sue varianti. Viene utilizzato il learning rate per determinare la dimensione del passo.
- **Iterazioni**: I passi descritti precedentemente vengono ripetuti iterativamente per il numero di epoche, con l'obiettivo di ridurre progressivamente la funzione di perdita e migliorare la capacità del modello di predire correttamente l'output.

Parliamo adesso del *Gradient Descent*, nominato anche nella descrizione della backpropagation. È una strategia di ottimizzazione che viene utilizzata insieme alla backpropagation nell'addestramento: utilizza i gradienti calcolati dalla backpropagation per aggiornare i pesi in modo da minimizzare la funzione di perdita.

Definiti questi concetti, passiamo adesso al problema in esame. Durante lo sviluppo si è deciso di utilizzare l'optimizer *Adam (Adaptive Moment Estimation)* [3]. Adam estende il concetto di Gradient Descent.

Adam Si tratta di uno degli ottimizzatori più popolari per l'addestramento delle reti neurali. È noto per essere efficiente in termini computazionali, richiedere poca memoria, e per essere adatto a problemi che sono grandi in termini di dati o parametri.
Adam adatta il tasso di apprendimento per ciascun peso del modello (altri modelli lo fanno per tutti i pesi) utilizzando le stime dei momenti. Inoltre, è stato dimostrato essere efficace in pratica su una vasta gamma di problemi di apprendimento automatico. È anche relativamente facile da configurare, con i suoi valori predefiniti delle impostazioni che funzionano bene in molte situazioni, sebbene possano comunque essere regolati per specifiche esigenze.
L'algoritmo di Adam aggiorna i pesi della rete neurale nel seguente modo:
- Calcola il gradiente della funzione di perdita rispetto ai pesi.
- Calcola le medie mobili dei gradienti e dei loro quadrati.
- Corregge queste medie mobili per eliminare un bias verso zero all'inizio dell'addestramento.
- Aggiorna i pesi utilizzando queste medie mobili corrette.

Per una descrizione dettagliata del funzionamento dell'algoritmo si faccia riferimento alla documentazione di Adam [3].

Codice:

```
optimizer = optim.Adam(alexnet.parameters(), lr=0.001)
```

Il primo parametro fornisce i pesi e bias che l'ottimizzatore dovrà aggiornare. Ciò che ci interessa maggiormente è il parametro *lr* rappresentante il Learning Rate (tasso di apprendimento).

Il learning rate viene utilizzato dall'ottimizzatore e controlla la dimensione dei passi che l'algoritmo compie verso il minimo della funzione di perdita. Determina quindi quanto velocemente o lentamente il modello aggiorna i suoi pesi in risposta all'errore osservato durante l'addestramento.

- **Troppo Alto**: Se il learning rate è troppo alto, l'ottimizzatore potrebbe "saltare" oltre il minimo, portando a una convergenza instabile o addirittura a una divergenza, dove l'errore di addestramento inizia ad aumentare invece di diminuire.
- **Troppo Basso**: Al contrario, se il learning rate è troppo basso, l'addestramento diventa molto lento perché l'ottimizzatore fa passi molto piccoli. Ciò può portare a un lungo tempo di addestramento e c'è il rischio che l'addestramento si arresti in un minimo locale anziché globale, a seconda della natura della funzione di perdita.

Il valore $0,001$ è il valore consigliato e di default di Adam che è stato utilizzato durante il pre-addestramento e funziona in maniera ottima nella maggior parte dei casi. Sono stati effettuati tentativi di variazione del learning rate ma il valore predefinito ha portato a risultati più stabili, di conseguenza si è scelto di mantenerlo.

Tale valore è sufficientemente basso da evitare oscillazioni eccessive ma abbastanza alto da garantire una convergenza efficiente senza rimanere bloccati in minimi locali o richiedere troppe iterazioni.

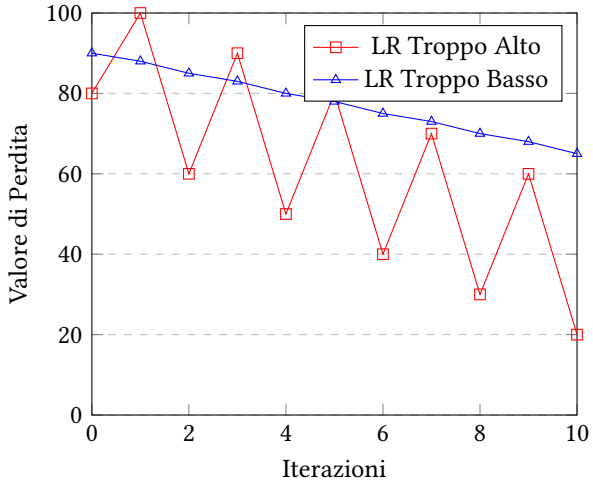


Fig. 2.6. Effetto del Learning Rate sull'ottimizzazione

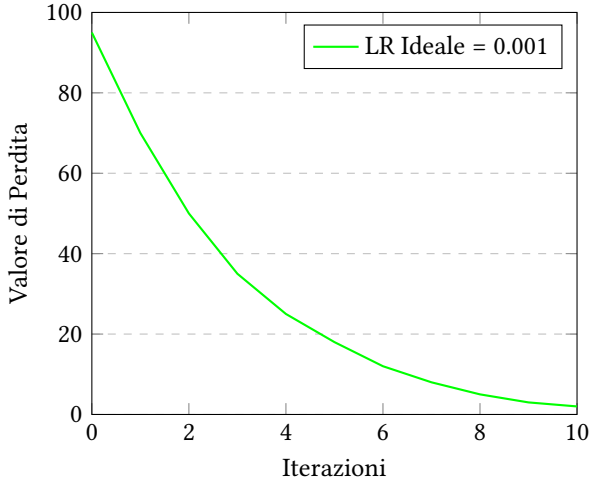


Fig. 2.7. Convergenza con Learning Rate Ideale

3 MODIFICHE APPORTATE AL MODELLO

L'idea alla base del progetto è stata applicare la tecnica del *transfer learning*.

Il transfer learning è una tecnica che consente ai modelli di trarre vantaggio dalle conoscenze acquisite in precedenti attività di addestramento per affrontare nuovi compiti correlati con maggiore efficienza.

Abbiamo adottato il transfer learning per adattare il modello di rete neurale preesistente a un nuovo compito specifico. La rete di partenza, progettata per la classificazione di immagini RGB in multiple classi, necessitava di modifiche sostanziali per soddisfare i requisiti del nostro progetto, che includeva lavorare con immagini in scala di grigio e indirizzare una classificazione binaria.

Le principali modifiche apportate riguardano:

- **Caricamento della rete preaddestrata**:
  Abbiamo iniziato caricando una versione preaddestrata di AlexNet.
  ```
  weights = AlexNet_Weights.DEFAULT
  alexnet = alexnet(weights=weights)
  ```
  Questa scelta che ci ha permesso di sfruttare una base di conoscenze visive già ampia, riducendo così il tempo e le risorse necessarie per l'addestramento da zero.
- **Modifica del primo strato convoluzionale**:
  Poiché il nostro compito coinvolgeva immagini in scala di grigio anziché RGB, abbiamo adattato il primo strato convoluzionale per accettare un singolo canale di input:
  ```
  alexnet.features[0] = nn.Conv2d(1, 64, kernel_size=11,
      stride=4, padding=2)
  ```
  Questo adattamento ci ha permesso di mantenere l'architettura profonda della rete pur modificandola per le nostre esigenze specifiche.
- **Inizializzazione tramite euristica**:
  Per evitare i problemi legati all'inizializzazione casuale e migliorare la convergenza del modello, abbiamo optato per la *Kaiming Initialization*, o *He Initialization*. Si faccia riferimento a [4] per informazioni dettagliate.
  Codice:
  ```
  nn.init.kaiming_normal_(alexnet.features[0].weight,
      mode='fan_in', nonlinearity='relu')
  if alexnet.features[0].bias is not None:
      nn.init.constant_(alexnet.features[0].bias, 0)
          stride=4, padding=2)
  ```
  La scelta di questo metodo è dovuta all'utilizzo della funzione di attivazione ReLu sui risultati dei layer convoluzionali. Il metodo di inizializzazione He permette di mantenere la varianza dei gradienti attraverso i diversi strati della rete. Il principio di base è di inizializzare i pesi in modo tale che la varianza dell'input di ogni strato sia uguale alla varianza dell'output, mantenendo così i gradienti in un intervallo ragionevole durante la backpropagation.

La formula per la Kaiming initialization è la seguente:

$$W = \text{random\_values} * \sqrt{\frac{2}{n_l}} \tag{3.1}$$

Dove:

W = pesi da inizializzare
random_values = valori estratti da una distribuzione normale con $\mu = 0$ e $\sigma^2 = 1$
n_l = numero di unità nell'input del livello l

In sostanza, la Kaiming initialization aiuta a garantire che all'inizio dell'addestramento, ogni neurone abbia una varianza dell'output simile e contribuisca a mantenere i gradienti in un intervallo gestibile.

- **Modifiche dell'ultimo strato fully-connected**:
  Per adattare la rete alla classificazione binaria, abbiamo sostituito l'ultimo strato fully-connected per produrre un singolo output, invece delle multiple classi gestite originariamente dalla rete.

```
alexnet.classifier[6] = nn.Linear(4096, 1)
```

- **Aggiunta della Sigmoid**:
  L'introduzione della funzione Sigmoid (2.2.2) all'ultimo strato ha reso possibile l'uso della rete per compiti di classificazione binaria, fornendo un output nel range [0,1] che indica la probabilità di appartenenza alla classe di interesse.

```
alexnet.classifier.add_module("7", nn.Sigmoid())
```

- **Congelamento dei pesi nei primi strati**: Con "congelamento dei pesi dei primi strati", ci riferiamo al fatto che i parametri (pesi e bias) dei primi strati della rete neurale non verranno aggiornati durante il processo di addestramento.

```
for param in alexnet.features.parameters():
    param.requires_grad = False
```

  Questo approccio risulta molto utile con reti neurali preaddestrate perché i primi strati di queste reti tendono ad estrarre caratteristiche di basso livello. Questo è stato ottenuto impostando il parametro *requires_grad* su *False* e per tutti i parametri nei livelli convoluzionali della rete AlexNet.

Qui è riportato il risultato ottenuto dopo aver effettuato le modifiche:

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(1, 64, kernel_size=(11, 11), stride=(4, 4),
           padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0,
            dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1),
           padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0,
            dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1),
           padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1),
           padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
           padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0,
             dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=1, bias=True)
    (7): Sigmoid()
  )}
```

4 FASE DI TRAIN E TEST

Questa sezione a dedicata all'illustrazione e spiegazione di come sono stati effettuati il train e il test del modello.

L'obiettivo della fase di addestramento è di insegnare al modello a riconoscere pattern, relazioni e a compiere specifiche azioni basandosi sui dati forniti.

La fase di test ha lo scopo di valutare la capacità del modello di generalizzare le sue conoscenze a nuovi dati, mai visti durante la fase di addestramento. Si mira a stimare l'accuratezza e l'efficacia del modello nel mondo reale.

Prima di mostrare tali processi analizziamo le misure standard utilizzate per valutare la performance del modello, sia nella fase di train che nella fase di test, ovvero le metriche.

4.1 Metriche

Le metriche in machine learning sono standard di misura che valutano le performance dei modelli. Sono cruciali per capire quanto bene un modello esegue compiti specifici, come la classificazione o la regressione.

Utilizzo durante il Train Usare le metriche durante il train aiuta a monitorare come il modello apprende e, a fare aggiustamenti se necessario, per esempio modificando i parametri, la struttura del modello o le tecniche di pre-elaborazione dei dati. Tuttavia, un'elevata performance sul training set potrebbe indicare un overfitting, ovvero che il modello ha imparato a memoria i dati di addestramento, ma non generalizza bene su nuovi dati.

Utilizzo durante il Test Valutare il modello sul test set con le metriche fornisce un'indicazione più realistica della sua capacità di generalizzazione su dati non visti. È fondamentale per assicurarsi che il modello sia davvero efficace e pronto per essere utilizzato in applicazioni reali.

Si noti che la funzione di perdita è cruciale per visualizzare l'andamento e l'apprendimento del modello (2.2.1), tuttavia le metriche sono adatte per fornire un'indicazione complementare per la valutazione del modello di classificazione.

Abbiamo quindi utilizzato le metriche sia in fase di train che di test visualizzando l'andamento del modello ed apportando modifiche in risposta ad esso.

È stata utilizzata la matrice di confusione per avere una visione d'insieme delle performance del modello, mostrando il numero di predizioni corrette e errate fatte per ciascuna classe. La matrice di confusione permette di individuare in modo chiaro e dettagliato i tipi di errori compiuti dal modello. Questo è utile per comprendere dove il modello sta sbagliando e quali classi sono più difficili da distinguere.

Parliamo quindi della matrice di confusione, anche detta tabella di errata classificazione, la quale restituisce una rappresentazione dell'accuratezza di un classificatore.

	Istanze realmente positive	Istanze realmente negative
Istanze predette come positive	Veri positivi	Falsi positivi
Istanze predette come negative	Falsi negativi	Veri negativi

Tabella 4.1. Matrice di confusione

La matrice di confusione è una matrice con cui poter indicare se ed in quanti casi il classificatore ha predetto correttamente o meno il valore di un'etichetta. Sulla base dei valori della matrice di confusione, possiamo poi calcolare diverse metriche.

Definiamo come TP il numero di veri positivi, come FP il numero di falsi positivi, TN il numero di veri negativi e come FN il numero di falsi negativi.

Le metriche usate nella classificazione del modello sono state: accuracy, precision e recall.

4.1.1 Accuracy. L'accuracy misura la frazione di previsioni corrette (sia positive che negative) rispetto al totale delle previsioni fatte. Si calcola come:

$$\frac{TP + TN}{TP + TN + FP + FN} \tag{4.1}$$

L'accuracy è utile quando le classi sono bilanciate, ma può essere fuorviante in dataset sbilanciati, dove una classe è molto più frequente dell'altra.

4.1.2 Precision. La precision misura la frazione di previsioni corrette positive rispetto al totale delle previsioni positive. In altre parole, indica quanto sia affidabile il modello quando afferma che un'istanza appartiene alla classe positiva. Si calcola come:

$$\frac{TP}{TP + FP} \tag{4.2}$$

La precision è particolarmente importante in situazioni dove il costo di un falso positivo è elevato.

4.1.3 Recall. La recall misura la frazione di positivi reali che sono stati correttamente identificati dal modello. Valuta quindi la capacità del modello di trovare tutte le istanze positive. Si calcola come:

$$\frac{TP}{TP + FN} \tag{4.3}$$

La recall è cruciale quando è importante identificare tutti i casi positivi.

4.2 Train e Test

Durante l'addestramento, il modello tenta di ottimizzare i suoi parametri interni (pesi) per minimizzare la funzione di perdita, che misura quanto le predizioni del modello si discostano dai risultati attesi. L'addestramento può richiedere molteplici iterazioni (epoche) attraverso il dataset per migliorare la sua accuratezza.

Prima di addestrare il modello utilizzando PyTorch, è necessario effettuare un processo di preparazione dei dati, è qui mostrato il codice:

```
train_dataset = datasets.ImageFolder(data_dir_train,
    transform=standard_transforms)

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)

print(train_dataset)
print(train_loader.batch_size)
```

Output:

```
Dataset ImageFolder
    Number of datapoints: 6000
    Root location: Alzheimer_s_Dataset/train
    StandardTransform
Transform: Compose(
               Resize(size=(227, 227),
                   interpolation=bilinear, max_size=None, antialias=warn)
               Grayscale(num_output_channels=1)
               ToTensor()
               Normalize(mean=[0.485], std=[0.229])
           )
32
```

Spiegazione del codice:

- **datasets.ImageFolder**: si tratta di una funzione fornita da PyTorch per caricare facilmente il dataset di immagini. Si aspetta che le immagini siano organizzate in una struttura di cartelle, dove ogni cartella rappresenta una classe. Restituirà alla variabile *train_dataset* una struttura del dataset organizzata a seconda delle etichette.
  Richiede i seguenti parametri:
  - **data_dir_train**: variabile che contiene il percorso della directory dove si trovano i dati di addestramento.
  - **transform=standard_transforms**: (opzionale) questo argomento specifica le trasformazioni da applicare a ciascuna immagine del dataset.
    La variabile *standard_transforms* definisce la composizione di trasformazioni applicate (2.4), utilizzando *torchvision.transforms*.
- **DataLoader**: il DataLoader si occupa di organizzare la struttura del dataset sottoforma di batch (lotti) per velocizzare e ottimizzare l'addestramento. Restituisce il suo contenuto alla variabile *train_loader*.
  Richiede i seguenti parametri:
  - **dataset**: nel nostro caso forniamo il dataset risultante da *datasets.ImageFolder*.

  - **batch_size**: indica il numero di batch di dati con il quale effettuare i passi di addestramento nel modello. Le reti neurali generalmente lavorano meglio con lotti di dati e non utilizzando direttamente l'intero dataset.
    È stato scelto 32 perché rappresenta un ottimo compromesso tra velocità di addestramento e bontà dei risultati. Con 64 l'addestramento era più veloce ma, anche se in minima parte, più impreciso.
- **shuffle**: può essere impostato a True o False, indica se effettuare un mescolamento dei dati ad ogni iterazione (epoch). Nella fase di training vogliamo utilizzare lo shuffle per aggiungere una componente casuale che permetta di differenziare i vari training, quindi abilitiamo lo shuffle. È utile soprattutto perché alcuni dataset contengono i tipi di etichette in maniera sequenziale e ciò non permetterebbe un buon addestramento del modello.

4.2.1 Funzione di training. Dopo il processo di preparazione dei dati procediamo con la definizione della funzione di training.

Le prime righe del codice contengono l'inizializzazione del criterion (2.2.1), l'inizializzazione dell'optimizer (2.5) e la definizione del numero di epoche:

```
criterion = nn.BCELoss()
optimizer = optim.Adam(alexnet.parameters(), lr=0.001)

num_epochs = 30
```

In seguito, la definizione della funzione di training:

```
def test_model(model, train_loader, criterion, device)
```

La funzione prende come parametri il modello, il train_loader, il criterion, inizializzato precedentemente e il device su cui viene eseguito il train

Inizia, quindi, definendo l'inizializzazione del ciclo di addestramento:

```
for epoch in range(num_epochs):
    model.train()
```

Viene iterato per il numero di epoche definito precedentemente e si pone il modello in modalità di addestramento abilitando caratteristiche specifiche del modello, come ad esempio il *dropout*.

Il dropout consiste in una tecnica di regolarizzazione utilizzata nel training delle reti neurali per prevenire l'overfitting: mitiga questo adattamento "disattivando" casualmente un sottoinsieme di neuroni in uno o più strati della rete. Nel caso di Alexnet si trova dopo gli strati fully-connected e prima dell'ultimo strato di output. La probabilità di dropout è impostata al 50% ed è stata mantenuta tale durante il corso dell'addestramento.

Dopo aver posto il modello in modalità di addestramento, c'è l'inizializzazione delle variabili per l'addestramento:

```
train_loss = 0.0
train_preds, train_labels = [], []
```

Si inizializza la variabile per il calcolo del loss totale, l'array per raccogliere le predizioni generate dal modello, quindi i valori stimati dal modello basati sugli input correnti accumulati per tutti i batch del dataset, e l'array delle etichette reali, cioè i valori effettivi che si stanno cercando di prevedere.

Si passa, quindi, al ciclo sui dati di addestramento:

```
for inputs, labels in train_loader:
    inputs, labels = inputs.to(device), labels.to(device)
```

Si itera sul dataset di addestramento utilizzando il DataLoader e si spostano i dati, cioè gli *inputs* e le *label* sul dispositivo usato per l'addestramento, nel nostro caso una GPU.

Viene eseguito l'azzeramento dei gradienti:

```
optimizer.zero_grad()
```

Azzera i gradienti del modello prima del calcolo del gradiente per l'attuale step di addestramento, per evitare l'accumulo di gradienti da più backward pass. L'azzeramento dei gradienti è necessario prima di ogni iterazione di addestramento poiché i valori calcolati nei backward pass si accumulerebbero portando a gradienti non rappresentativi dell'ultimo set di dati processato. Questo interferirebbe con l'aggiornamento dei pesi, allontanando la rete dalla direzione ottimale di apprendimento.

Abbiamo il forward pass:

```
outputs = model(inputs)
```

E la regolazione delle dimensioni di output e labels:

```
if outputs.dim() > 1 and outputs.size(1) == 1:
    outputs = outputs.squeeze(1)
if labels.dim() > 1:
    labels = labels.squeeze()
labels = labels.float()
```

Le uscite della rete possono avere grandezze diverse a seconda dell'architettura. Se le dimensioni di outputs e labels non corrispondono, il calcolo della perdita non può procedere.

Passiamo al calcolo del loss e del backward pass:

```
loss = criterion(outputs, labels)
train_loss += loss.item() * inputs.size(0)
loss.backward()
optimizer.step()
```

Si calcola il loss utilizzando il criterion scelto, basato sugli *outputs* del modello e le *labels* reali. Aggiorna il loss totale di addestramento, esegue il backward pass per calcolare i gradienti del modello e aggiorna i pesi del modello utilizzando l'ottimizzatore (optimizer).

Si raccolgono le predizioni e le etichette:

```
train_preds.extend(outputs.detach().cpu().numpy())
train_labels.extend(labels.detach().cpu().numpy())
```

Si convertono in array numpy per il successivo calcolo delle metriche.
Calcolo delle metriche:

```
train_loss /= len(train_loader.dataset)
train_accuracy = accuracy_score(train_labels,
    [p > 0.5 for p in train_preds])
train_precision = precision_score(train_labels,
    [p > 0.5 for p in train_preds])
train_recall = recall_score(train_labels,
    [p > 0.5 for p in train_preds])
```

Viene effettuato un calcolo de loss medio di addestramento per epoca e calcolo delle metriche scelte, quali accuracy, recall e precision, utilizzando le predizioni e le etichette raccolte.
Infine si ha la stampa delle metriche:

```
print(f"Epoch {epoch+1}/{num_epochs}")
print(f"Loss: {train_loss:.4f} Accuracy: {train_accuracy:.4f}
    Precision: {train_precision:.4f} Recall: {train_recall:.4f}")
```

La funzione restituisce il modello addestrato:

```
return model
```

4.2.2 Funzione di test. Durante la fase di test di un modello di machine learning l'obiettivo è valutare quanto bene il modello addestrato generalizza su dati non visti durante l'addestramento. A differenza dell'addestramento, in questa fase non si cercano di ottimizzare ulteriormente i pesi del modello; invece, si utilizzano i parametri (pesi) già appresi durante l'addestramento per fare previsioni su un nuovo set di dati, comunemente noto come set di test.

Analogamente a come descritto nella fase di training è necessario il processo di preparazione dei dati:

```
test_dataset = datasets.ImageFolder(data_dir_test,
    transform=standard_transforms)

test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

print(test_dataset)
print(test_loader.batch_size)
```

Output:

```
Dataset ImageFolder
Number of datapoints: 1279
Root location: Alzheimer_s_Dataset/test
StandardTransform
Transform: Compose(
           Resize(size=(227, 227),
               interpolation=bilinear, max_size=None, antialias=warn)
           Grayscale(num_output_channels=1)
           ToTensor()
           Normalize(mean=[0.485], std=[0.229])
       )
32
```

La differenza sostanziale durante la preparazione dei dati (oltre il fatto che naturalmente viene fornito il dataset di test) dalla fase di train è che il parametro *shuffle* è impostato su False. Nella fase di test, l'ordine in cui i dati vengono presentati al modello non influisce sul calcolo delle metriche di valutazione complessive. Non mescolare i dati assicura che ogni volta che si esegue la valutazione, i dati vengono presentati al modello nello stesso ordine. Questo rende più facile riprodurre i risultati del test e confrontare le prestazioni tra diversi modelli o configurazioni in maniera consistente.

Anche la funzione di test è piuttosto simile a quella di train, mostriamone i cambiamenti. Inanzitutto, come ovvio che sia, il modello deve essere impostato in modalità valutazione:

```
model.eval()  # Imposta il modello in modalità di valutazione
```

Le funzioni *optimizer.zero_grad()*, *loss.backward()* e *optimizer.step()* non sono presenti poiché specifici per la fase di addestramento essendo direttamente coinvolti nel processo di ottimizzazione dei pesi del modello. Anche durante il test, la loss viene similmente calcolata per valutare la performance del modello, ma non influisce sull'aggiornamento dei pesi.

Infine, le metriche di performance sono calcolate per valutare in modo imparziale la capacità di generalizzazione del modello su dati non visti e sono calcolate su tutto il set di dati di test per fornire una valutazione comprensiva delle prestazioni del modello.

Codice completo della funzione di test:

```
from sklearn.metrics import precision_score, recall_score, accuracy_score
import torch

def test_model(model, test_loader, criterion, device):
    model.eval()  # Imposta il modello in modalità di valutazione

    test_loss = 0.0
    all_labels = []
    all_predictions = []

    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)

            # Forward pass
            outputs = model(inputs)

            # Controlla e regola le dimensioni dell'output
            if outputs.dim() > 1 and outputs.size(1) == 1:
                outputs = outputs.squeeze(1)

            # Converte in float
            labels = labels.float()

            # Calcola la loss
            loss = criterion(outputs, labels)
            test_loss += loss.item() * inputs.size(0)

            # Converti gli output in predizioni binarie
            predicted = (outputs > 0.5).float()
            all_labels.extend(labels.cpu().numpy())
            all_predictions.extend(predicted.cpu().numpy())

    test_loss /= len(test_loader.dataset)

    test_accuracy = accuracy_score(all_labels, all_predictions)
    test_precision = precision_score(all_labels, all_predictions)
    test_recall = recall_score(all_labels, all_predictions)

    print(f"Test Loss: {test_loss:.4f}")
    print(f"Test Accuracy: {test_accuracy:.4f}")
    print(f"Test Precision: {test_precision:.4f}")
    print(f"Test Recall: {test_recall:.4f}")
```

5 TENTATIVI DI MIGLIORAMENTO EFFETTUATI

Di seguito sono riportati i cambiamenti effettuati che hanno portato miglioramenti al modello (e sono stati quindi mantenuti) e i cambiamenti che invece non hanno portato alcun miglioramento o l'hanno addirittura peggiorato (di conseguenza non sono stati matenuti).

5.1 Cambiamenti aggiunti

Le modifiche qui riportate hanno migliorato i parametri della loss e delle metriche.

- **Augmentation dei dati**: per avere un dataset più ampio.
- **Congelamento dei pesi per i primi strati**: per evitare l'overfitting.
- **Aggiunta dell'euristica per l'inizializzazione dei pesi**: per non avere pesi totalmente casuali e migliorare l'apprendimento.
- **Passare da 10 epoche a 30 epoche**: per dare più tempo al modello di migliorare.

5.2 Cambiamenti non aggiunti

Le modifiche qui riportate portavano ad overfitting o ad un aumento del parametro della loss con diminuzione dei valori delle metriche.

- **Modifica del DropOut**: da $0,5$ a $0,6$ per migliorare l'apprendimento del modello.
- **Diminuzione degli strati convoluzionali**: da 5 a 3 poiché pensavamo che le immagini fossero troppo semplici per la rete.
- **Aumento e diminuzione del learning rate**: per migliorare l'apprendimento del modello.
- **Utilizzo della CLAHE (Contrast Limited AHE)**: per l'aumento del contrasto, in maniera tale da aiutare la rete a distinguere maggiormente i dettagli.
- **Aggiunta del *weight_decay* all'optimizer**: per evitare l'overfitting.
- **Utilizzo di un threshold**: per bilanciare il miglioramento verso l'ottimizzazione della recall.
- **Varie combinazioni**: di ogni punto precedente.

Tutto ciò evidenzia come la fase di sperimentazione dei modelli di machine learning sia cruciale per identificare le strategie più efficaci per migliorare le prestazioni. È fondamentale un approccio iterativo, che prevede la valutazione dell'impatto di ogni singola modifica in un contesto controllato, per distinguere chiaramente tra le modifiche che offrono miglioramenti reali e quelle che possono invece essere controproducenti.

6 DEPLOYMENT

Nell'ambito dello sviluppo e dell'implementazione di soluzioni basate sull'intelligenza artificiale, la capacità di esportare efficacemente un modello addestrato è cruciale per facilitare il suo utilizzo in applicazioni reali e ambienti di produzione. La funzione di esportazione del modello di IA è stata progettata con l'obiettivo di rendere questo processo il più semplice e versatile possibile, garantendo al contempo la massima efficienza e compatibilità attraverso diversi ambienti e piattaforme. Funzione di esportazione del modello:

```
def save_model(model_state_dict, file_name,):
    # Definizione del dizionario da salvare
    model_info = {
        'architettura': 'AlexNet Modificato',
        'model_state_dict': model_state_dict,
        'optimizer_state_dict': optimizer.state_dict(),
        'criterion_state_dict': criterion.state_dict(),
        'epoch': num_epochs,
        'preprocessing': {
            'resize': (227, 227),
            'mean': [0.485],
            'std': [0.229],
            'grayscale': True
        },
    }

    # Salvataggio del dizionario in un file
    torch.save(model_info, file_name)
    print(f"Modello salvato come {file_name}")
```

Il parametro *model_state_dict* è un oggetto Python dict che mappa ciascun layer al suo tensor di parametri (pesi e bias) del modello. Questo dizionario è essenziale per la serializzazione e deserializzazione dello stato di un modello, permettendo agli sviluppatori di salvare, caricare, trasferire e ripristinare modelli in modo efficiente.

Un discorso analogo vale per l'*optimizer.state_dict* e il *criterion.state_dict*. Tali operazioni permettono il salvataggio dello stato dell'optimizer e del criterion così da poterli riutilizzare successivamente.

Dopodiché vengono salvate alcune informazioni globali quali:

- Il nome dell'architettura.
- Il numero di epoch.
- Le informazioni di preprocessing.

Infine tramite un parametro che definisce il nome del file sottoforma di percorso, il modello viene esportato nel file system.

Il caricamento del modello sarà possibile grazie alle seguenti istruzioni:

```
# Caricamento delle informazioni salvate
loaded_info = torch.load('model/saved_model.pth')
# Caricamento stato modello
alexnet.load_state_dict(loaded_info['model_state_dict'])
# Caricamento stato optimizer
optimizer.load_state_dict(loaded_info['optimizer_state_dict'])
# Caricamento stato criterion
criterion.load_state_dict(loaded_info['criterion_state_dict'])
```

Si noti che sarà necessario riconfigurare AlexNet prima di caricare le varie informazioni salvate. Tuttavia, prima di tale processo, il modello dovrà essere definito in tale modo:

```
alexnet = alexnet(weights=None)
```

Il parametro *weights=None* riferisce alla creazione di un'istanza del modello AlexNet senza caricare i pesi preaddestrati.

In conclusione, sarà quindi possibile impostare il modello in modalità di valutazione e procedere con le classificazioni (si faccia riferimento al blocco note Jupyter, contenuto nella repository, per il codice completo).

6.1 Integrazione del modello con la piattaforma web

Il modello esportato è stato poi caricato nel server Flask, preparandolo per l'inferenza in tempo reale.

Flask è un microframework per applicazioni web in Python che è stato scelto per la sua semplicità e flessibilità. Il server Flask è stato configurato per gestire richieste HTTPS POST, permettendo agli utenti di caricare in modo sicuro le immagini TAC attraverso la piattaforma web di *Esistere*. La scelta di utilizzare HTTPS garantisce che i dati trasmessi, potenzialmente sensibili, siano criptati e protetti durante il trasferimento.

Flusso di Richiesta e Risposta:

- **Caricamento immagine**: gli utenti accedono alla piattaforma web e selezionano un'immagine TAC dal loro dispositivo per il caricamento. L'immagine viene inviata al server tramite una richiesta POST HTTPS.
- **Elaborazione del modello**: il server Flask riceve l'immagine e la passa al modello di IA per l'elaborazione. Il modello analizza l'immagine e genera una predizione basata sul suo addestramento.
- **Risposta**: la predizione viene quindi inviata indietro al client tramite il server Flask. L'utente riceve la risposta "Demented" o "Non Demented" a seconda della classificazione.

Il codice è reperibile all'interno della repository ed è contenuto nel file *server.py*.

7 CONCLUSIONE

Nel nostro progetto incentrato sull'impiego di AlexNet per la diagnosi dell'Alzheimer da immagini TAC, le principali sfide affrontate hanno riguardato non solo l'adattamento dell'architettura pre-addestrata alle nostre specifiche necessità ma anche il conseguimento di risultati promettenti dal modello. La personalizzazione di AlexNet per rispondere efficacemente alle caratteristiche uniche del nostro dataset si è rivelata cruciale per superare le limitazioni imposte da un ambiente differente. Il migliori risultati ottenuti dopo svariati train e test sono stati i seguenti:

```
# Modello 10
Test Loss: 0.5496
Test Accuracy: 0.7084
Test Precision: 0.7834
Test Recall: 0.5766

# Modello 11
Test Loss: 0.5574
Test Accuracy: 0.7349
Test Precision: 0.7400
Test Recall: 0.7250

# Modello 12
Test Loss: 0.5662
Test Accuracy: 0.7021
Test Precision: 0.7085
Test Recall: 0.6875
```

Il nostro progetto ha esplorato le potenzialità dell'intelligenza artificiale, ciò ci ha permesso non solo di applicare le conoscenze acquisite durante il corso [7] ma anche di esplorare nuovi meccanismi e sfide in tale ambito. Attraverso questo lavoro, abbiamo compreso in modo più profondo come teorie e principi discussi in aula si traducano in pratica nel superare ostacoli reali, come il bilanciamento dei dati, l'adottare i vari tipi classificazione, il preprocessing, l'utilizzo di funzioni di perdita, di tecniche di normalizzazione dei dati, l'applicazione corretta di metriche, e di tutto ciò che è stato appresso in maniera individuale riguardo le reti convoluzionali. Questa esperienza ha arricchito il nostro apprendimento, dimostrando il valore dell'applicazione pratica nel consolidare ed espandere la nostra comprensione teorica.

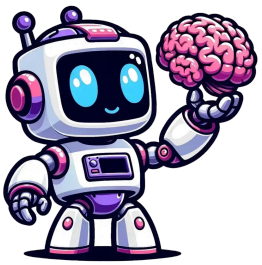


Fig. 7.1. Grazie!

RIFERIMENTI BIBLIOGRAFICI

[1] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanano. 2016. PyTorch. https://pytorch.org.
[2] PyTorch Contributors. 2023. PyTorch Documentation: torch.nn.BCELoss. https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html. Accessed: 2023-02-10.
[3] PyTorch Contributors. 2023. PyTorch Documentation: torch.optim.Adam. https://pytorch.org/docs/stable/generated/torch.optim.Adam.html. Accessed: 2023-02-10.
[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. https://arxiv.org/abs/1502.01852. *Computer Science* arXiv:1502.01852 (2015), 11.
[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., n.a., 1097–1105.
[6] n.a. 2021. Dataset_Alzheimer. Kaggle. https://www.kaggle.com/datasets/yasserhessein/dataset-alzheimer
[7] Prof. Fabio Palomba. 2023-2024. Contenuto del Corso di Fondamenti di Intelligenza Artificiale. https://docenti.unisa.it/027888/home. Accessed: 2023-2024.