

1 Autograd and SGD

1.1 What is the principle behind using Stochastic Gradient Descent (SGD) for optimizing a function w.r.t. its parameters?

A differenza di un approccio esatto, come la discesa del gradiente esatta che utilizza tutti i dati del training per valutare la loss del modello e cercare di minimizzarla, lo SGD cerca di ridurre il numero di dati di utilizzo, utilizzando un batch. Questo permette di valutare la loss con meno accuratezza rispetto a un metodo esatto, ma con una complessità sicuramente minore. Anche perché spesso valutare la loss su tutti i punti del dataset potrebbe essere impraticabile. L'aggiornamento dei pesi viene fatto utilizzando sempre la backpropagation, utilizzando la delta rule:

$$w_{new} = w_{old} - \alpha \cdot \frac{\partial L}{\partial w}$$

1.2 Explain the concept of function composition in neural networks and how it relates to layers in a model.

Una funzione composta, è una funzione $h(x)$ tale per cui, date le funzioni $f(x)$ e $g(x)$ posso scrivere:

$$h(x) = f(g(x))$$

Si può generalizzare per n funzioni diverse, ognuna dipendente da x . Una rete neurale è per sua natura una funzione composta, dove ogni uscita di un livello è l'ingresso del successivo. Per questo possiamo dire che, dato l'ingresso $X \in \mathbb{R}^{D \times B}$, e dati i livelli nascosti h_1, \dots, h_n , posso dire che l'uscita Y :

$$Y = h_n(h_{n-1}(\dots h_1(X)\dots))$$

Tenendo ben presente che ogni livello è composto da una somma pesata degli ingressi e un livello di non linearità dato dalla funzione di attivazione.

1.3 How do you calculate the derivative of a composed function w.r.t. its inputs?

La derivata di una funzione composta $h(x) = f(g(x))$ è: $h'(x) = f'(g(x)) \cdot g'(x)$.

Essa può essere anche calcolata con il meccanismo della Chain Rule. Data la funzione composta $h(x) = f(g(x))$, posso calcolare la derivata come:

$$\frac{\partial h}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x}$$

1.4 Describe the unfolding process of a function and its impact on derivative calculation.

Una funzione composta $o(x) = l_3(l_2(l_1(x)))$ può rappresentare un modello di rete neurale, dove ogni l_i è un layer della rete. Il processo di "unfolding" di una funzione composta del genere, è quello di dividerla in funzioni più semplici, per poi usare la chain rule delle derivate delle funzioni della scomposizione, per calcolare la derivata della funzione composta. In questo caso la scomposizione potrebbe essere

$$w_1 = l_1(x)$$

$$w_2 = l_2(w_1)$$

$$w_3 = l_3(w_2)$$

Questo ci permetterà di calcolare la derivata di $o(x)$ come, prima valutando le derivate delle funzioni della scomposizione, e poi moltiplicandole tra loro:

$$\frac{\partial o}{\partial x} = \frac{\partial w_3}{\partial w_2} \cdot \frac{\partial w_2}{\partial w_1} \cdot \frac{\partial w_1}{\partial x}$$

1.5 Provide an example of applying the chain rule with multiple variables and binary operators in differentiation.

Un operatore binario è una regola che combina due elementi detti operandi, per produrre un terzo elemento detto risultato (es. $+$, $-$, $*$, $/$). Possiamo definire un operatore binario come una funzione:

$$f : A \times B \rightarrow C$$

Un operatore binario è un'operazione **chiusa** se il dominio è $A \times A$ e il codominio è A .

Un esempio di applicazione della chain rule con un operatore binario è il seguente:

$$o(x, y) = (x + y) * \sin(x)$$

Inizialmente scomponiamo la funzione in:

$$w_1 = x + y$$

$$w_2 = \sin(x)$$

$$o = w_1 \cdot w_2$$

Calcoliamo le derivate delle funzioni della scomposizione:

$$\frac{\partial o}{\partial x} = \frac{\partial w_1 \cdot w_2}{\partial x} = w_2 \frac{\partial w_1}{\partial x} + w_1 \frac{\partial w_2}{\partial x}$$

Da notare che l'operatore binario ha l'effetto di dividere il flusso di derivazione in due rami. Sostituendo le derivate col loro calcolo effettivo ($\frac{\partial w_1}{\partial x} = 1$, e $\frac{\partial w_2}{\partial x} = \cos x$), otteniamo:

$$\frac{\partial o}{\partial x} = w_2 + \cos x$$

$$\frac{\partial o}{\partial x} = \sin x + (x + y) \cdot \cos x$$

1.6 Explain the derivation process for a function involving a binary operator and how it splits the derivation flow.

Per poter calcolare la derivata di una funzione composta con un operatore binario, bisogna scomporre la funzione in funzioni più semplici, per poi calcolare le derivate delle funzioni nate dalla scomposizione. Se è presente un operatore binario, esso causerà una biforcazione nel flusso di derivazione, in quanto la derivata di questo tipo di funzione composta è data dalla somma pesata delle derivate delle due funzioni con i pesi che dipendano dal tipo di operatore binario. Per capire come organizzare i pesi, basta calcolare la derivata della funzione composta con l'operatore binario:

- Addizione: $\frac{\partial(f(x)+g(x))}{\partial x} = 1 \frac{\partial f(x)}{\partial x} + 1 \frac{\partial g(x)}{\partial x}$
- Sottrazione: $\frac{\partial(f(x)-g(x))}{\partial x} = 1 \frac{\partial f(x)}{\partial x} - 1 \frac{\partial g(x)}{\partial x}$
- Moltiplicazione: $\frac{\partial(f(x) \cdot g(x))}{\partial x} = f(x) \cdot \frac{\partial g(x)}{\partial x} + g(x) \cdot \frac{\partial f(x)}{\partial x}$
- Divisione: $\frac{\partial \frac{f(x)}{g(x)}}{\partial x} = \frac{1}{g(x)} \frac{\partial f(x)}{\partial x} - \frac{f(x)}{g(x)^2} \frac{\partial g(x)}{\partial x}$

1.7 What is reverse mode differentiation, and how is it applied to compute derivatives in computational graphs?

La "Reverse mode differentiation" è un modo per calcolare la derivata dell'uscita rispetto all'ingresso di un grafo computazionale. Dopo aver costruito il grafo, bisogna calcolare la derivata di ogni nodo rispetto al suo arco entrante. Per calcolare la derivata dell'intera funzione descritta dal grafo computazionale, bisogna scegliere un'uscita e seguire tutti i percorsi che portano a un ingresso, moltiplicando i valori delle derivate trovate. Se sono presenti biforcazioni, bisogna sommare il prodotto delle derivate di ogni percorso *Out* → *In*. In questo modo possiamo calcolare la derivata di un'uscita rispetto a tutti gli ingressi in un solo passaggio.

1.8 Discuss the concept of forward path and backward path in the context of computational graphs and differentiation.

Il "Forward path" è il percorso che parte da uno degli ingressi del grafo computazionale e arriva a una delle uscite, mentre il "Backward path" è il percorso che parte da una delle uscite e arriva a uno degli ingressi. Sono entrambi utili nel calcolo delle derivate, nel contesto di un grafo computazionale.

Per calcolare la derivata in "Backward", bisogna per prima cosa calcolare la derivata di ogni nodo rispetto al suo arco entrante, e poi calcolare la derivata dell'intera funzione descritta dal grafo computazionale, scegliendo un'uscita e seguendo tutti i percorsi, sommando eventuali biforazioni.

Per calcolare la derivata in "Forward", bisogna inizialmente scegliere un ingresso per il quale si vuole calcolare la derivata, e poi portare in ingresso al grafo la derivata su ogni ingresso rispetto all'ingresso scelto (Sarà un vettore con tutti zeri e un 1 nella posizione dell'ingresso scelto). Dopo aver calcolato la derivata di ogni nodo rispetto al suo arco entrante, per avere la derivata dell'uscita rispetto all'ingresso scelto, bisogna moltiplicare le derivate che si incontrano negli archi, e sommare quando due percorsi che partono da input diversi si incontrano.

1.9 Describe the differences between forward mode differentiation and reverse mode differentiation in terms of computational efficiency and application.

Data una funzione $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$:

In Reverse Mode Differentiation:

- Ogni arco del grafo computazionale ha un solo valore di derivata associato da calcolare.
- Per il punto precedente, i valori delle derivate nel grafo computazionale sono condivise fra le derivate parziali di un ingresso rispetto a un'uscita.
- Possiamo calcolare la derivata di un'uscita rispetto a tutti gli ingressi in un solo passaggio.
- Ideale quando il numero di ingressi è maggiore del numero di uscite ($n \gg m$).

In Forward Mode Differentiation:

- Ogni nodo del grafo computazionale ha diversi valori, uno per ogni derivata parziale (Per ogni ingresso vanno ricalcolate).
- Per il punto precedente, i valori delle derivate nel grafo computazionale **non** sono condivise fra le varie derivate parziali.

- Possiamo calcolare la derivata di un ingresso rispetto a tutte le uscite in un solo passaggio.
- Ideale quando il numero di uscite è maggiore del numero di ingressi ($m \gg n$).

1.10 How do machine learning frameworks utilize reverse mode differentiation, and what advantages does this offer for building complex architectures?

In machine learning abbiamo a che fare con modelli che imparano ad approssimare funzioni del tipo $f : \mathbb{R}^n \rightarrow \mathbb{R}$, dove n può essere esageratamente grande (Anche nell'ordine dei miliardi). Per questo motivo, la "Reverse Mode Differentiation" è la scelta migliore, perché permette di calcolare la derivata di un'uscita rispetto a tutti gli ingressi in un solo passaggio. I modelli di machine learning utilizzano la "Reverse Mode Differentiation" per calcolare le derivate delle funzioni di loss rispetto ai pesi del modello, e propagare all'indietro l'errore per aggiornare i pesi.

1.11 Draw a one-dimensional loss function w.r.t. one parameter. Perform one step of SGD

Possiamo per esempio costruire una funzione di loss come:

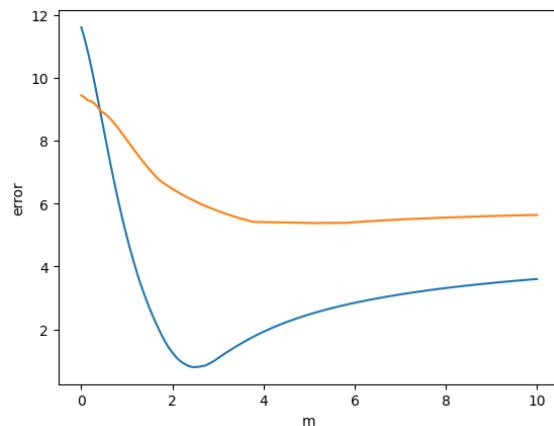


Figura 1: Funzione di loss valutata su tutti i punti del dataset in blu, e in arancione valutata in un batch.

I passi per eseguire un'iterazione di SGD, deciso quale sia il batch di dati da utilizzare e la loss function L :

- calcolare la predizione del modello sul batch
- valutare l'errore del modello sul batch
- calcolare la derivata della loss rispetto ai pesi del modello: $\frac{\partial L}{\partial w}$

- aggiornare i pesi del modello con la formula: $w = w - \alpha \cdot \frac{\partial L}{\partial w}$

2 Tensor algebra and PyTorch

2.1 Explain the concept of a tensor in PyTorch.

Un Tesore è un array multidimensionale, che generalizza i concetti di vettori e matrici verso più ampie dimensioni ($N \geq 1$). In PyTorch, i tensori sono la struttura dati fondamentale, e possono essere utilizzati per rappresentare dati i gresso, in uscita e i pesi di un modello. I tensori possono essere elaborati attraverso l'algebra tensoriale, che include operazioni come: addizione, moltiplicazione, rearrange, reshape, reduction, ecc.

In PyTorch i tensori possono essere creati con la funzione `torch.tensor()`, passando come argomento, una lista di valori, una lista di liste di valori, un numpy array, ecc. Oppure possiamo creare dei tensori pieni di zeri o di uno con le funzioni `torch.zeros()` e `torch.ones()`, specificando le dimensioni desiderate. Ogni tensore può essere elaborato con l'algebra tensoriale in PyTorch.

2.2 How do the addition and multiplication between tensors work?

L'addizione e la moltiplicazione tra tensori funzionano in modo simile all'addizione e alla moltiplicazione tra matrici. Vengono sommati (moltiplicati) gli elementi del tensore indicizzati nello stesso modo. In PyTorch esiste l'operazione implicita di broadcasting, che automaticamente replica un tensore verso una dimensione che non è uguale a quella dell'altro tensore con cui si vuole fare l'operazione algebrica. Questo permette di fare operazioni tra tensori di dimensioni diverse

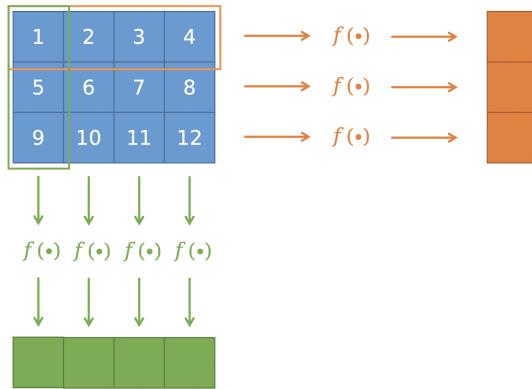
2.3 What is the difference between the 'reshape' and 'view' methods in tensor manipulation?

Sebbene il risultato delle operazioni di "reshape" e "view" sia lo stesso, ovvero quello di cambiare l'ordine degli elementi del tensore, la differenza sta nel fatto che "reshape" va a modificare effettivamente l'ordinamento del tensore in memoria, mentre "view" si limita a modificare l'indicizzazione.

"Reshape" riarrangia il tensore in modo più lento, ma fornisce un accesso ad esso più veloce, mentre "View" ha prestazione opposte, riarrangia velocemente ma consente un accesso più lento.

2.4 Explain the use of reduction operations like '.sum', '.mean', '.max', etc., in tensor algebra.

L'operazione di riduzione va a modificare la forma stessa del tensore, riducendola secondo un certo criterio di aggregazione. Le operazioni di riduzione come ".sum", ".mean", ".max" vanno a sommare, calcolare la media, e trovare il massimo rispettivamente, in una dimensione di un tensore definita dall'utente. Per esempio:



dove $f(\cdot)$ è uno degli aggregatori sopra citati.

2.5 Describe the process of creating a custom dataset using 'torch.utils.data.Dataset'.

In PyTorch possiamo definire un dataset personalizzato attraverso la classe `torch.utils.data.Dataset`. Questo è utile per poter definire a piacimento come i dati vengono caricati, trasformati e restituiti al modello. Per creare un dataset personalizzato, bisogna creare una classe che erediti da `torch.utils.data.Dataset`, e implementare i metodi:

- `__init__(self)`: Inizializza il dataset, e carica i dati in memoria.
- `__len__(self)`: Restituisce la lunghezza del dataset.
- `__getitem__(self, idx)`: Restituisce l'elemento del dataset all'indice `idx`.

2.6 What is the 'torch.utils.data.IterableDataset' and how is it used?

Il `torch.utils.data.IterableDataset` è una classe di PyTorch che permette di creare dataset che seguono il modello iterativo di Python. Questo ci permette di creare dataset che esso stesso è un iteratore, e quindi possiamo iterare su di esso con un ciclo for. Per creare un `IterableDataset`, bisogna creare una classe che erediti da `torch.utils.data.IterableDataset`, e implementare il metodo:

- `__iter__(self)`: Restituisce un iteratore sul dataset.
- `__next__(self)`: Restituisce l'elemento successivo del dataset.

2.7 How does the 'torch.utils.data.DataLoader' work in PyTorch?

Il `torch.utils.data.DataLoader` è una classe di PyTorch utile a creare un dataset iterabile, a partire da un dataset personalizzato. Il *Dataloader* supporta sia map-style dataset che l'iterable-style dataset, con singoli o multipli processi di caricamento, ci permette di personalizzare l'ordine di caricamento dei dati, e di definire la dimensione del batch.

Gli argomenti principali del *DataLoader* sono:

- **dataset**: Il dataset da caricare.
- **batch_size**: La dimensione del batch.
- **shuffle**: Se i dati devono essere caricati in ordine casuale. Questo funziona solo per i map-style dataset.
- **num_workers**: Il numero di processi che caricano i dati. Ogni processo carica un dato.
- **sampler**: Il metodo di campionamento dei dati.

2.8 Describe the structure and purpose of the 'torch.nn.Module' class.

La classe *torch.nn.Module* è la classe base per tutti i modelli la cui archittettura è definita in PyTorch. Esso serve per specificare la struttura del modello, e come esso si deve comportare in "Forward". La classe *torch.nn.Module* ha i seguenti metodi principali:

- **`__init__(self)`**: Inizializza il modello, e definisce i layer che lo compongono.
- **`forward(self, x)`**: Definisce come i dati devono essere elaborati dal modello in "Forward".
- **`parameters(self)`**: Restituisce i parametri del modello.
- **`to(self, device)`**: Sposta il modello su un dispositivo specificato.

Ogni modello può comprendere parametri, rappresentati da tensori racchiusi in *torch.nn.Parameter*. Ogni attributo del modello che è definito come *torch.nn.Parameter* è incluso nella lista dei parametri del modello, e può essere aggiornato durante l'ottimizzazione. Anche tutti i parametri dei sub-module che vengono definiti, sono racchiusi in *torch.nn.Parameter*. PyTorch dispone di molti moduli come: *torch.nn.Linear*, *torch.nn.Conv2d*, *torch.nn.Sequential*, ecc.

Un esempio di come definire un modello in PyTorch che calcola una funzione quadratica è:

```
import torch
import torch.nn as nn

class QuadraticModel(nn.Module):
    def __init__(self):
        self.a = nn.Parameter(torch.randn(1))
        self.b = nn.Parameter(torch.randn(1))
        self.c = nn.Parameter(torch.randn(1))

    def forward(self, x):
        return self.a * x**2 + self.b * x + self.c
```

2.9 What are the key methods in a PyTorch module, and how are they implemented?

I metodi principali di un modulo PyTorch sono:

- **`__init__(self)`**: Inizializza il modello, e definisce i layer che lo compongono.
- **`forward(self, x)`**: Definisce come i dati devono essere elaborati dal modello in "Forward".
- **`parameters(self)`**: Restituisce i parametri del modello.
- **`to(self, device)`**: Sposta il modello su un dispositivo specificato.

2.10 Discuss the training process in PyTorch using the Stochastic Gradient Descent (SGD) algorithm.

Il processo di training in PyTorch con l'algoritmo di ottimizzazione SGD è il seguente:

- Definire il modello, e i dati di training e di test.
- Definire la funzione di loss.
- Definire l'ottimizzatore, in questo caso SGD.
- Definire il numero di epoche.
- Per ogni epoca:
 - Per ogni batch:
 - * Calcolare la predizione del modello.
 - * Calcolare l'errore del modello.
 - * Calcolare la derivata della loss rispetto ai pesi del modello.
 - * Aggiornare i pesi del modello.

Un esempio di codice può essere:

```
import torch
import torch.nn as nn
import torch.optim as optim

# Definire il modello
model = Model()
loss_fn = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Definire il numero di epoche
for epoch in range(num_epochs):
    for batch in dataloader:
        # Calcolare la predizione del modello
        output = model(batch)
        # Calcolare l'errore del modello
        loss = loss_fn(output, target)
        # Calcolare la derivata della loss rispetto ai pesi del modello
        optimizer.zero_grad()
        loss.backward()
        # Aggiornare i pesi del modello
        optimizer.step()
```

2.11 Explain the concept and application of batch size in model training.

Il batch è un numero di dati di training che vengono passati al modello contemporaneamente per addestrarlo. Questo è utile quando si hanno dataset molto grandi, che non possono essere caricati in memoria in un unico momento, così si caricano frammentati in batch.

2.12 How do you implement a simple linear regression model in PyTorch?

Un modello di regressione lineare in PyTorch può essere implementato come:

```
import torch
import torch.nn as nn

# Definire il modello
class LinearRegression(nn.Module):
    def __init__(self):
        super(LinearRegression, self).__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, x):
        return self.linear(x)

# Training
model = LinearRegression()
loss_fn = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

for epoch in range(num_epochs):
    for batch in dataloader:
        output = model(batch)
        loss = loss_fn(output, target)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

2.13 Discuss the implementation of a custom model in PyTorch and the steps to train it on a dataset.

Per implementare un modello personalizzato in PyTorch, bisogna creare una classe che erediti da `torch.nn.Module`, e definire i layer:

```
import torch
import torch.nn as nn

# Definire il modello
class CustomModel(nn.Module):
    def __init__(self):
        super(CustomModel, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(10, 20),
            nn.ReLU(),
            nn.Linear(20, 1)
        )

    def forward(self, x):
        return self.model(x)

# Training
dataloaders = nn.DataLoader(dataset, batch_size=32, shuffle=True)
model = CustomModel()
loss_fn = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

for epoch in range(num_epochs):
    for batch in dataloaders:
        output = model(batch)
        loss = loss_fn(output, target)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

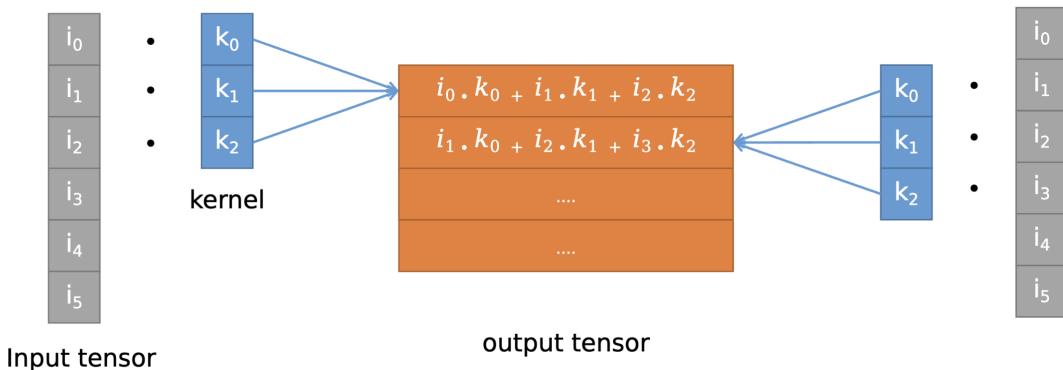
3 Convolutional Neural Networks and ResNets

3.1 What is a CNN?

Una Convolutional Neural Network (CNN) è un tipo particolare di rete neurale profonda, adatta per l'elaborazione di dati strutturati in griglie, come le immagini. Le CNN si basano sul concetto di convoluzione, che permette di estrarre automaticamente le caratteristiche salienti dei dati in ingresso. Le CNN sono composte da diversi strati, tra cui i layer di convoluzione, pooling e fully connected.

3.2 How does the convolution operation work in CNNs?

L'operatore di convoluzione funziona moltiplicando una piccola finestra dei dati con un kernel, e sommando i risultati per produrre un singolo valore. Questa operazione viene ripetuta facendo scorrere il kernel lungo tutte le dimensioni dell'ingresso, producendo un tensore di uscita che i cui elementi sono il risultato delle varie convoluzioni con kernel. Questo permette di estrarre le caratteristiche salienti dei dati in ingresso. La seguente immagine mostra come funziona l'operazione di convoluzione:



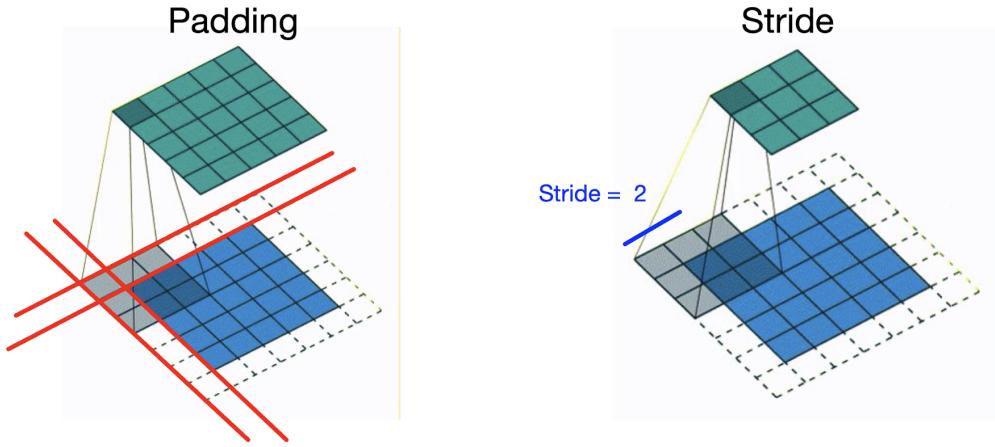
In una CNN i valori dei pesi di un livello corrispondono ai valori del kernel o dei kernel usati, e vengono appresi durante il training. I risultati di kernel diversi vengono a formare i canali del tensore di uscita.

3.3 Explain the significance of kernel size, padding, and stride in convolutional layers.

Per **kernel size** si intende la dimensione del kernel o dei kernel usati nella convoluzione. Le dimensioni del kernel determinano l'area dell'input che viene considerata durante la convoluzione, e quindi influenzano la dimensione del tensore di uscita.

Per **padding** si intende l'aggiunta di extra pixel all'input così da garantire che la convoluzione processi l'intero ingresso, specialmente nei bordi.

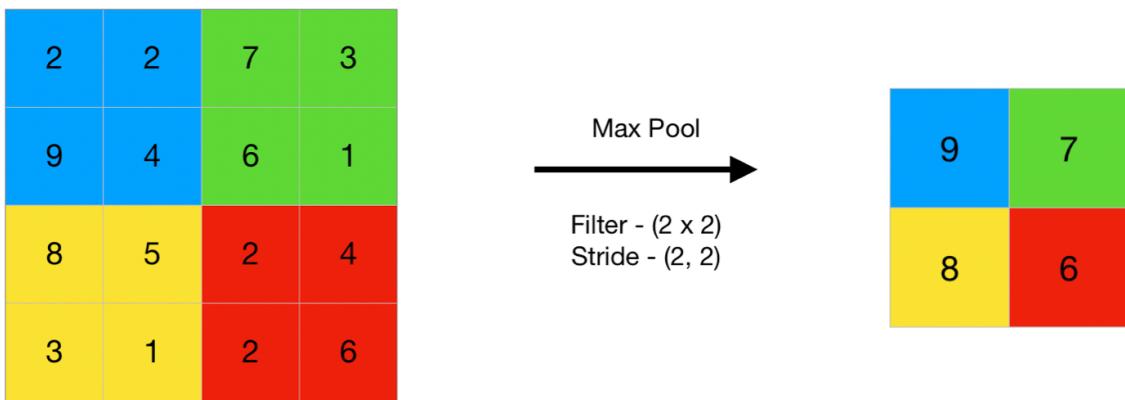
Per **stride** si intende il numero di pixel che il kernel salta mentre scorre l'input. Più il passo è grande, più la dimensione del tensore di uscita sarà ridotta.



3.4 What are the roles of pooling layers in CNNs?

I **pooling layers** sono utilizzati per ridurre la dimensione spaziale di un tensore, semplicemente utilizzando un aggregatore. Viene scelta un area di pooling, e viene applicata una funzione di aggregazione (come il massimo o la media) per ridurre la dimensione. Questo ci permette di ridurre le dimensioni del tensore che va in input ai livelli successivi.

Un esempio di pooling layer è il *MaxPooling*, che restituisce il valore massimo dell'area di pooling:



3.5 Discuss the function of activation functions in CNNs.

Le **funzioni di attivazione** sono utilizzate per introdurre non linearità in una CNN, permettendo al modello di approssimare funzioni complesse. Questo viene fatto perché di per se la convoluzione è un'operazione lineare, e dunque una composizione di convoluzione può essere ridotta ad una singola convoluzione. Facendo seguire

ogni operazione lineare da una funzione di attivazione, si risolve questo problema. Le funzioni di attivazione più comuni sono la *ReLU*, la *Sigmoid* e la *Tanh*, ecc.

3.6 How does a Conv2D layer in PyTorch operate?

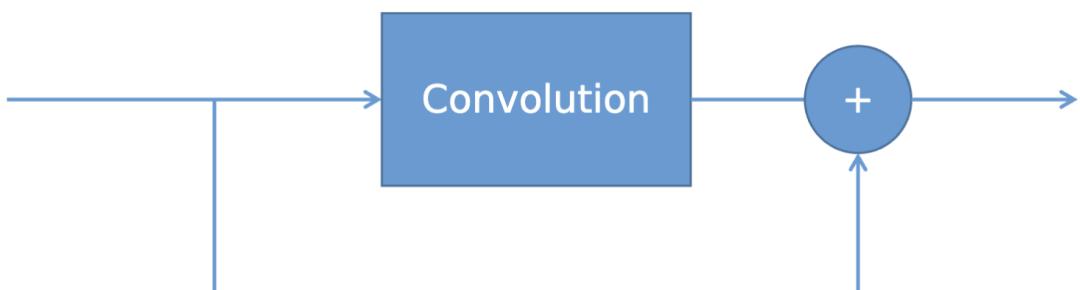
In PyTorch, un layer di convoluzione 2D è definito con la classe `torch.nn.Conv2d`. La convoluzione è calcolata in parallelo essendo la stessa operazione da fare su input diversi. Può essere usato il paradigma *SIMD*, e le GPU sono perfette per questo tipo di operazione.

3.7 Explain the concept of channels in CNNs and their significance.

I **canali** in una CNN sono i diversi "strati" del tensore di uscita di un layer di convoluzione. Ogni canale corrisponde al risultato di una convoluzione con un kernel diverso. I canali permettono alla CNN di catturare diverse caratteristiche dell'input, e di apprendere gerarchie di feature complesse.

3.8 What are skip connections in CNNs, and how do they function?

Le **skip connections** sono connessioni dirette fra due layer di una rete neurale. Questo tipo di soluzione è stata introdotta per cercare di superare il problema del "vanishing gradient" e del "exploding gradient" nelle reti profonde. Le skip connections permettono al gradiente di propagarsi direttamente attraverso i layer, senza essere attenuato:



Questo collegamento diretto fra ingresso e uscita è poi sommato all'uscita del layer, permettendo al gradiente di propagarsi direttamente. Abbiamo però il problema che il tensore di uscita, una volta elaborato dalla rete neurale, avrà potenzialmente dimensioni che non corrispondono più a quelle del tensore di input, e quindi è impossibile fare la somma. Questo problema viene superato con due passaggi:

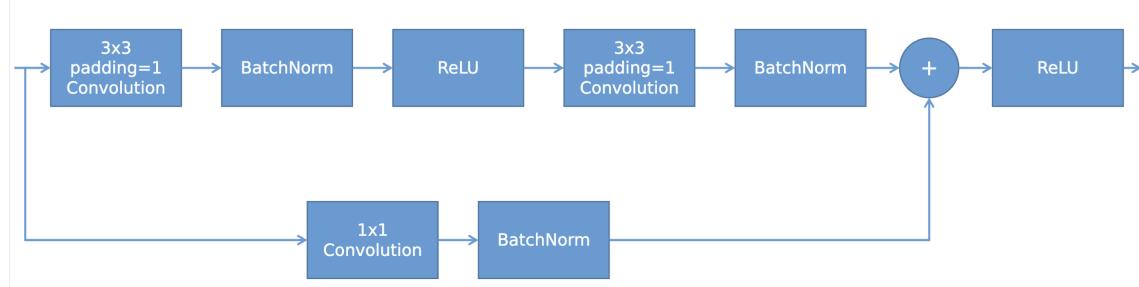
- **Half-Padding:** Si calcola la convoluzione col kernel, con un padding pari alla metà della dimensione del kernel, così da preservare le dimensioni spaziali del tensore.
- **1x1 Convolution:** Si applica una convoluzione con kernel 1x1 in skip connection, così da riportare il tensore di ingresso allo stesso numero di canali del tensore d'uscita. Un altro modo più banale è quello di usare lo **Zero-Padding**.

3.9 Define a Residual Network (ResNet) and its advantages in deep learning.

Una **Residual Network (ResNet)** è un tipo di rete neurale profonda che utilizza skip connections per superare il problema del "vanishing gradient" e del "exploding gradient". Questo tipo di rete neurale è composta da blocchi residui, che contengono skip connections che permettono al gradiente di propagarsi direttamente attraverso i layer. Questo permette di addestrare reti molto più profonde.

3.10 Draw a diagram of a ResNet and its computational graph

Ogni rete che presenta una skip connection è una ResNet. La ResNet più standard è:



3.11 Explain the concept of feature maps in CNNs

Le **feature maps** sono i tensori di uscita di un layer di convoluzione di una CNN. Ogni canale di un tensore di uscita corrisponde al risultato di una convoluzione con un kernel diverso, esse infatti rappresentano la risposta di quel filtro convoluto con l'ingresso. Ogni elemento di una feature map rappresenta l'attivazione di uno specifico neurone della rete, e il suo valore rappresenta il grado con cui quella specifica caratteristica è presente nell'input.

Per esempio nei livelli meno profondi di una CNN, le feature maps catturano caratteristiche di basso livello, come linee e bordi, mentre nei livelli più profondi catturano caratteristiche di alto livello, come forme e oggetti.

Il numero di feature maps in un layer convoluzionale è un iperparametro che può essere scelto in fase di progettazione della rete. Incrementare il numero di feature

maps permette alla rete di catturare più caratteristiche dell'input, ma aumenta il numero di parametri da allenare, e quindi la complessità della rete. Inoltre potrebbe portare ad un overfitting.

Le feature maps catturano quindi le caratteristiche salienti dell'input, e vengono passate ai layer successivi per l'elaborazione.

3.12 Describe the architecture of a typical CNN.

Una CNN tipica è composta da diversi layer, tra cui layer di convoluzione, pooling, e fully connected. I layer di convoluzione estraggono le caratteristiche salienti dell'input, i layer di pooling riducono la dimensione spaziale del tensore, e i layer fully connected elaborano le feature maps per la classificazione, questi ultimi sono utili per effettuare il task che la rete si presuppone di fare, perché i layers precedenti servono solo per estrarre e ridurre le features dell'input. Un esempio di architettura di una CNN che classifica digit di MNIST è in codice:

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__(input_w, input_h, num_classes)
        self.cnn = nn.Sequential(
            nn.Conv2d(kernel_size=(3, 3),
                      in_channels=1,
                      out_channels=32),
            nn.LeakyReLU(),
            nn.MaxPool2d(kernel_size=(2, 2)),
            nn.Conv2d(kernel_size=(3, 3),
                      in_channels=32,
                      out_channels=64),
            nn.LeakyReLU(),
            nn.MaxPool2d(kernel_size=(2, 2)),
            nn.Flatten()
        )

        # Fully connected layers
        self.ff = nn.Sequential(
            nn.Linear(64 * 6 * 6, 128),
            nn.LeakyReLU(),
            nn.Linear(128, num_classes)
        )

    def forward(self, x):
        x = self.cnn(x)
        x = self.ff(x)
        return x
```

3.13 What are the common challenges in training deep CNNs?

Alcuni dei problemi comuni nell'addestramento di CNN profonde sono:

- **Vanishing Gradient:** Il gradiente diventa troppo piccolo per aggiornare i pesi dei layer iniziali.
- **Exploding Gradient:** Il gradiente diventa troppo grande, causando oscillazioni e instabilità nell'addestramento.

- **Overfitting:** La rete impara troppo bene i dati di training, e non generalizza bene su nuovi dati.
- **Computational Complexity:** Le CNN profonde richiedono molte risorse computazionali per l'addestramento.

3.14 How do residual blocks in ResNets mitigate the vanishing gradient problem?

I **residual blocks** in ResNets mitigano il problema del "vanishing gradient" permettendo al gradiente di propagarsi direttamente attraverso la skip connection, senza essere attenuato durante all'attraversamento inverso della rete. Questo permette di addestrare reti molto più profonde.

3.15 Discuss the application of CNNs in image classification tasks, with an example like MNIST.

Le CNN sono ampiamente utilizzate per il riconoscimento di immagini, e sono particolarmente efficaci per questo tipo di task, perché riescono a estrarre feature salienti tramite la convoluzione fra l'immagine e i vari kernel, allenati per riconoscere feature specifiche (dipendentemente dal ground truth con cui si allena la rete). Un esempio di applicazione di CNN per il riconoscimento di immagini è il dataset MNIST, che contiene immagini di cifre, in codice:

```
# Modello CNN per MNIST
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.cnn = torch.nn.Sequential(
            torch.nn.Linear(28 * 28, 300),
            torch.nn.LeakyReLU(),
            torch.nn.Linear(300, 300),
            torch.nn.LeakyReLU(),
            torch.nn.Linear(300, 10),
            torch.nn.Softmax(dim=1)
        )

    def forward(self, x):
        return self.cnn(x)

# Training
dataloaders = nn.DataLoader(dataset_MNIST, batch_size=32, shuffle=True)
model = CNN()
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

for epoch in range(num_epochs):
    for digit, cls in dataloaders:
        output = model(digit)
        loss = loss_fn(output, cls)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

3.16 How does the structure of a ResNet differ from a standard CNN?

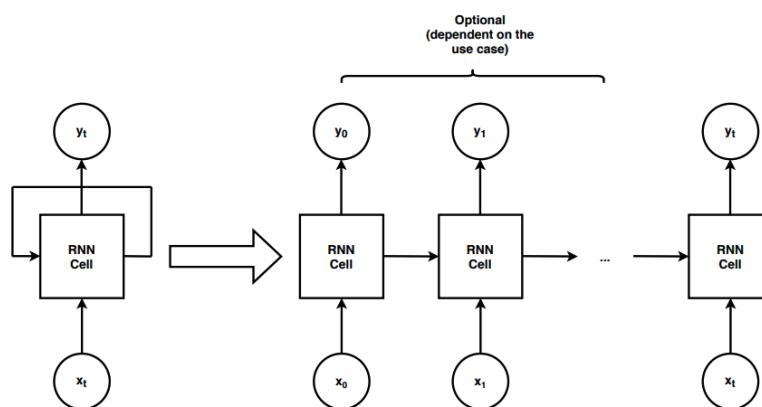
La struttura di una ResNet differisce da una CNN standard per la presenza di skip connections, che permettono al gradiente di propagarsi direttamente, senza il rischio di attenuazione durante l'attraversamento inverso della rete. Questo permette di addestrare reti molto più profonde.

4 Recurrent Neural Networks

4.1 What is a Recurrent Neural Network (RNN), and how does it work?

Una **Recurrent Neural Network (RNN)** è un tipo di rete neurale che si basa sul concetto di ricorrenza, ovvero sul concetto di applicare gli stessi pesi più volte su input che cambiano nel tempo. Questo ci permette di creare una rete che impara da sequenze di dati di qualsiasi lunghezza, come sequenze di testo, audio, video, ecc.

Le RNN sono composte da uno schema principale, detto **cella**, che elabora le sequenze di dati. In particolare ogni cella ha un input, uno stato nascosto, e un output. L'input è il dato da elaborare, lo stato nascosto è il risultato dell'elaborazione precedente, e l'output è il risultato dell'elaborazione corrente:



4.2 Explain the concept of time-varying inputs and outputs in RNNs.

Le RNN sono progettate per elaborare sequenze di dati, che possono variare nel "tempo". Questo significa che l'input e l'output della rete possono cambiare ad ogni passo temporale. Ad esempio, in una RNN che elabora sequenze di testo, l'input è una parola alla volta, e l'output è la predizione della parola successiva. A ogni passo temporale viene passato l'hidden state al passo successivo, così che la rete riesca a tenere conto delle informazioni passate.

4.3 Describe two major application families of RNNs: Sequence to Task and Sequence to Sequence.

Esistono due famiglie principali di applicazioni per le RNN:

- **Sequence to Task:** In questo tipo di applicazione, la RNN riceve una sequenza di dati in ingresso, e produce un'uscita singola, che è poi usata da un'altra rete

neurale per compiere un task specifico, come la classificazione di una sequenza di input.

- **Sequence to Sequence:** In questo tipo di applicazione, la RNN riceve una sequenza di dati in ingresso, e produce una sequenza di risultati che vengono passati a un'altra RNN che li elabora. Sono ottime per elaborare sequenze di dati di lunghezza non definita, come la traduzione di testo.

4.4 How do RNNs capture temporal dependencies and patterns in sequences?

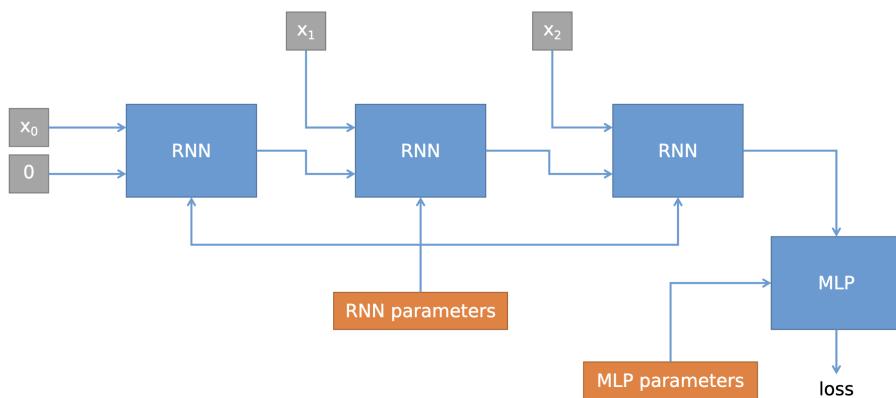
Le RNN catturano le dipendenze temporali e i pattern nelle sequenze attraverso l'uso dello stato nascosto. Lo stato nascosto contiene informazioni riguardo le elaborazioni temporalmente precedenti.

4.5 Discuss the vanishing gradient problem in RNNs and its impact on learning from long sequences.

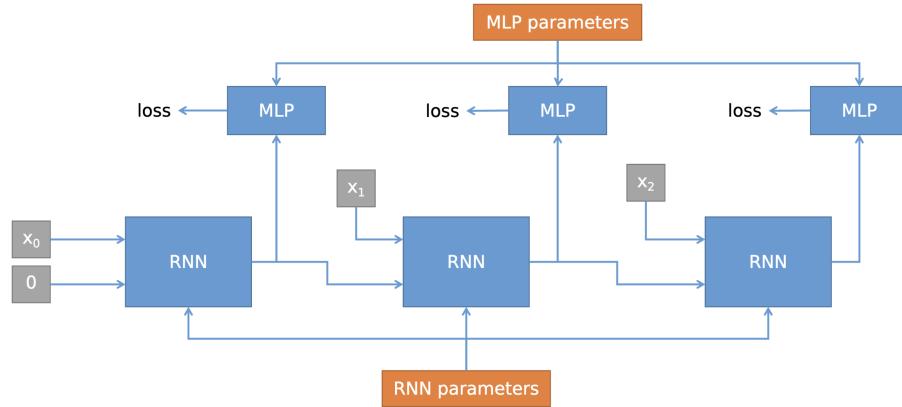
Le RNN, come le Deep Neural Networks, soffrono il problema del "vanishing gradient", che si verifica quando il gradiente diventa troppo piccolo per aggiornare i pesi. Nelle RNN il problema si manifesta perché, in fase di training, il gradiente deve essere propagato più volte attraverso la stessa cella. Nel caso specifico degli stati nascosti, il gradiente della rete dipende dal gradiente di ogni stato nascosto generato dalla rete stessa, e questo facilita il problema del "vanishing gradient", soprattutto quando si ha a che fare con lunghe sequenze di dati che costringono la rete a generare molti stati nascosti.

4.6 Draw a diagram of a RNN and its computational graph.

Caso Seq2Task:

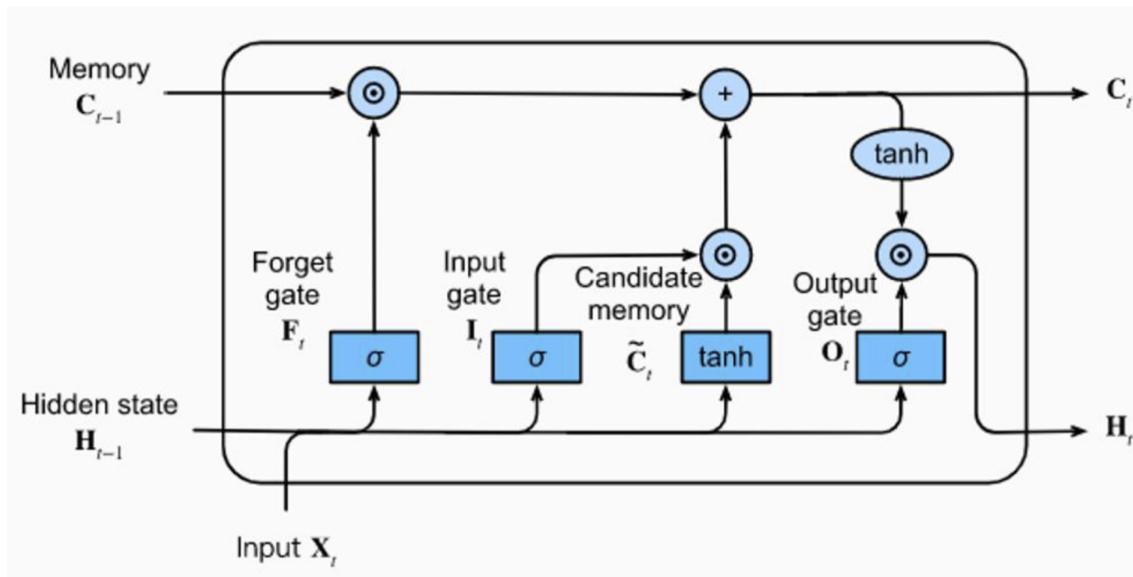


Caso Seq2Seq:



4.7 What are Long Short-Term Memory (LSTM) networks and how do they address the vanishing gradient problem?

Le **Long Short-Term Memory (LSTM)** sono un tipo di RNN che sono progettate per catturare dipendenze temporali a lungo termine. Le LSTM mitigano il problema del "vanishing gradient" attraverso l'uso di un meccanismo chiamato "gating", che permette di controllare quanto il gradiente viene propagato attraverso la rete. Questo permette alle LSTM di apprendere dipendenze temporali a lungo termine. Lo schema di una LSTM è il seguente:



In formulare i vari gate sono:

- **Input Gate:** $i_t = \sigma(W_{ii} \cdot x_t + b_{ii} + W_{hi} \cdot h_{t-1} + b_{hi})$
- **Forget Gate:** $f_t = \sigma(W_{if} \cdot x_t + b_{if} + W_{hf} \cdot h_{t-1} + b_{hf})$

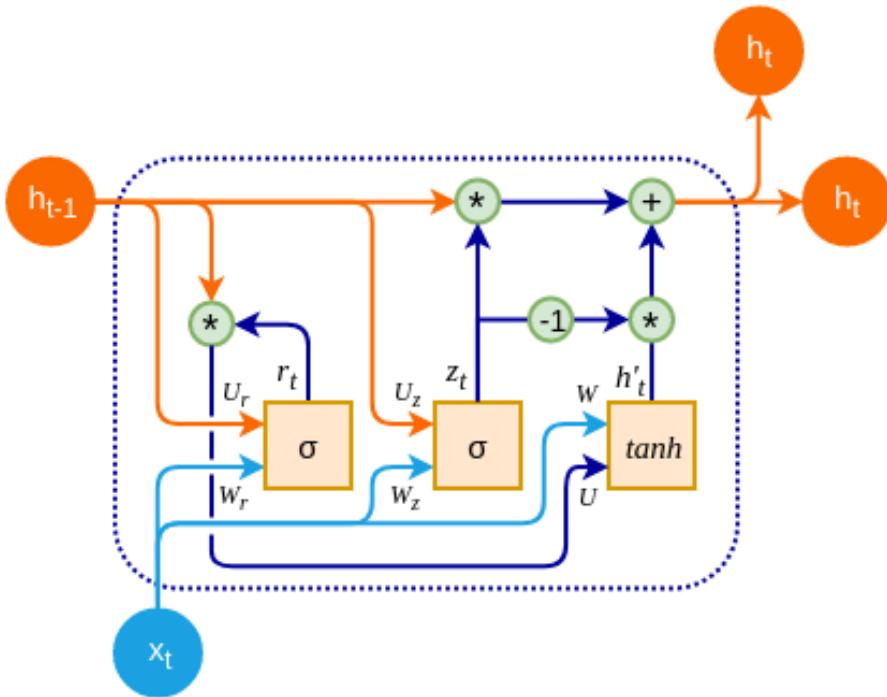
- **Cell State:** $g_t = \tanh(W_{ig} \cdot x_t + b_{ig} + W_{hg} \cdot h_{t-1} + b_{hg})$
- **Output Gate:** $o_t = \sigma(W_{io} \cdot x_t + b_{io} + W_{ho} \cdot h_{t-1} + b_{ho})$
- **Nest Memory:** $c_t = o_t \cdot c_{t-1} + i_t \cdot b_{ho}$
- **Next Hidden:** $h_t = o_t \cdot \tanh(c_t)$

4.8 What are Gated Recurrent Unit (GRU) networks and how do they differ from LSTMs?

Le **Gated Recurrent Unit (GRU)** sono un tipo di RNN che, come le LSTM, cercano di mitigare il problema del "vanishing gradient", attraverso un'architettura più complessa. Le GRU sono però più semplici delle LSTM, infatti presentano solo tre gate: Reset, Update e New.

In formule i vari gate sono:

- **Reset Gate:** $r_t = \sigma(W_{ir} \cdot x_t + b_{ir} + W_{hr} \cdot h_{t-1} + b_{hr})$
- **Update Gate:** $z_t = \sigma(W_{iz} \cdot x_t + b_{iz} + W_{hz} \cdot h_{t-1} + b_{hz})$
- **New Gate:** $n_t = \tanh(W_{in} \cdot x_t + b_{in} + r_t \cdot (W_{hn} \cdot h_{t-1} + b_{hn}))$
- **Next Hidden:** $h_t = (1 - z_t) \cdot n_t + z_t \cdot h_{t-1}$



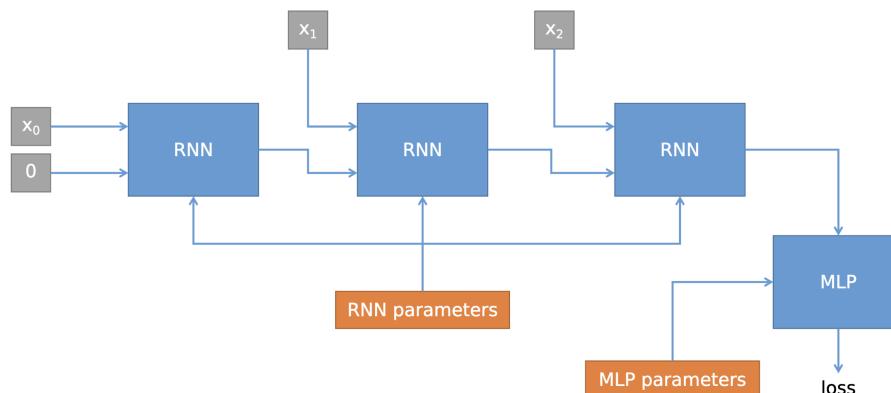
4.9 Describe the functionality of the three gates in GRUs: reset, update, and new gate.

I tre gate delle GRU sono:

- **Reset Gate:** Determina quanto del passato deve essere dimenticato, esso è attivato dalla sigmoide, e se tende a 0, il passato viene dimenticato.
- **Update Gate:** Determina quanto del passato deve essere mantenuto, anch'esso è attivato dalla sigmoide, e se tende a 0, il passato viene dimenticato.
- **New Gate:** Determina quanto del nuovo stato deve essere aggiunto, ed è dipendente dai due gate precedenti. Esso è attivato dalla tangente iperbolica, e determina il nuovo stato nascosto fra -1 e 1.

4.10 Explain how to train an RNN for a sequence classification problem.

Per addestrare una RNN per un problema di classificazione di sequenze, bisogna definire il modello della RNN e poi il modello della rete di classificazione che si vuole usare. Lo schema da usare è il seguente:



In codice un esempio potrebbe essere:

```
class RNNModel(torch.nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Creo lo schema della cella RNN
        self.rnn = torch.nn.Sequential(
            torch.nn.Linear(input_size + hidden_size, hidden_size),
            torch.nn.LeakyReLU(),
            torch.nn.Linear(hidden_size, hidden_size),
            torch.nn.LeakyReLU(),
            torch.nn.Linear(hidden_size, hidden_size),
        )
```

```

# Creo il layer di output full-forward per la classificazione
self.ff = torch.nn.Linear(hidden_size, output_size)

def forward(self, x):
    hidden_state = torch.zeros(x.shape[0], self.hidden_size, device=x.device)

    for t in range(x.shape[1]):
        # Concateno il punto della sequenza con lo stato nascosto
        input = torch.cat([x[:, t, :], hidden_state], dim=1)
        # Applico la cella RNN
        hidden_state = self.rnn(input)

    return self.ff(hidden_state)

# Training
dataloaders = nn.DataLoader(dataset, batch_size=32, shuffle=True)
model = Seq2Task()
loss_fn = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

for epoch in range(num_epochs):
    for input, cls in dataloaders:
        output = model(input)
        loss = loss_fn(output, cls)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

```

4.11 Discuss the considerations for setting up RNNs, LSTMs, and GRUs for a classification task

4.12 Discuss the challenges in training RNNs and how they can be mitigated.

La grande difficoltà nell'allenare le RNN è il problema del "vanishing gradient", che si verifica quando il gradiente diventa troppo piccolo. Questo problema può essere mitigato utilizzando architetture più complesse, come le LSTM e le GRU, che permettono di controllare quanto il gradiente viene propagato attraverso la rete. Altre tecniche per mitigare il problema del "vanishing gradient" sono l'utilizzo di funzioni di attivazione come la ReLU, che per sua natura ha un gradiente costante e quindi non rischia di vanificarsi durante la backpropagation.

5 Autoencoders and VAEs

5.1 What is an autoencoder, and what are its primary components?

Un Autoencoder è un'archittettura di rete neurale disegnata per problemi non supervisionati. Essa è composta da due parti principali: l'encoder e il decoder. L'encoder prende in input un dato e lo trasforma in una rappresentazione compatta, in un nuovo spazio, detto **latent space**. Il decoder prende la rappresentazione compatta e la trasforma nuovamente nel dato originale. L'obiettivo dell'autoencoder è quello di minimizzare l'errore di ricostruzione, ovvero la differenza tra l'input originale e l'output predetto dall'encoder.

5.2 Describe the roles of the encoder and decoder in an autoencoder.

L'**encoder** è la parte dell'autoencoder che trasforma l'input in una rappresentazione compatta, detta **latent space**. Questa è una rappresentazione compressa dell'input originale, che cattura le caratteristiche salienti dei dati. L'**Decoder** è una rete neurale che prende in ingresso i dati trasformati nello spazio latente e cerca di ricreare l'input originale.

5.3 What is meant by the "latent space" in an autoencoder?

Il **latent space** è lo spazio di rappresentazione compatta in cui l'encoder trasforma l'input. Questo spazio contiene le caratteristiche salienti dei dati, e permette di rappresentare i dati in modo più efficiente.

5.4 How does an autoencoder learn a compact representation of input data?

Un autoencoder apprende una rappresentazione compatta dei dati attraverso l'allenamento. Durante l'allenamento, l'autoencoder cerca di minimizzare l'errore di ricostruzione, ovvero la differenza tra l'input originale e l'output predetto dall'encoder. Questo processo permette all'autoencoder di apprendere le caratteristiche salienti dei dati e di trasformarle in una rappresentazione compatta nello spazio latente. Un esempio di autoencoder in PyTorch è il seguente:

```
# Modello
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__(input_size, hidden_size)
        self.encoder = nn.Sequential(
            nn.Linear(input_size, 300),
            nn.LeakyReLU(),
            nn.Linear(300, 300),
            nn.LeakyReLU(),
```

```

        nn.Linear(300, hidden_size)
    )

    self.decoder = nn.Sequential(
        nn.Linear(hidden_size, 300),
        nn.LeakyReLU(),
        nn.Linear(300, 300),
        nn.LeakyReLU(),
        nn.Linear(300, input_size)
    )

    def forward(self, x):
        latent = self.encoder(x)
        reconstruction = self.decoder(latent)
        return reconstruction

# Training
dataloaders = nn.DataLoader(dataset, batch_size=32, shuffle=True)
model = Autoencoder()
loss_fn = nn.L1Loss() # Mean Absolute Error
optimizer = optim.Adam(model.parameters(), lr=0.01)

for epoch in range(num_epochs):
    for input in dataloaders:
        output = model(input)
        loss = loss_fn(output, input)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

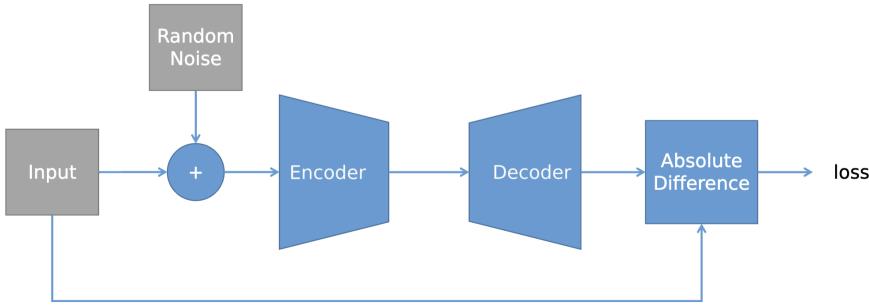
```

5.5 Discuss the concept of reconstruction error in autoencoders.

L'errore di ricostruzione è la differenza assoluta fra l'input originale e l'output predetto dall'encoder. Questo errore misura quanto bene l'autoencoder riesce a ricostruire l'input originale a partire dalla sua rappresentazione compatta nello spazio latente. Minimizzare l'errore di ricostruzione è l'obiettivo principale dell'autoencoder durante l'allenamento, e permettere all'encoder di creare una rappresentazione compatta dei dati, che riesca a cogliere le caratteristiche salienti.

5.6 What is a denoising autoencoder, and how does it differ from a traditional autoencoder?

Un **denoising autoencoder** è una variazione del semplice autoencoder che è progettata gestire dati rumorosi. Questo tipo di autoencoder è allenato per ricostruire l'input originale a partire da una versione rumorosa dell'input. Questo permette all'autoencoder di apprendere caratteristiche più robuste e generiche dei dati. Lo schema di un denoising autoencoder è il seguente:



in PyTorch un esempio di denoising autoencoder è il seguente:

```
# Modello
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__(input_size, hidden_size)
        self.encoder = nn.Sequential(
            nn.Linear(input_size, 300),
            nn.LeakyReLU(),
            nn.Linear(300, 300),
            nn.LeakyReLU(),
            nn.Linear(300, hidden_size)
        )

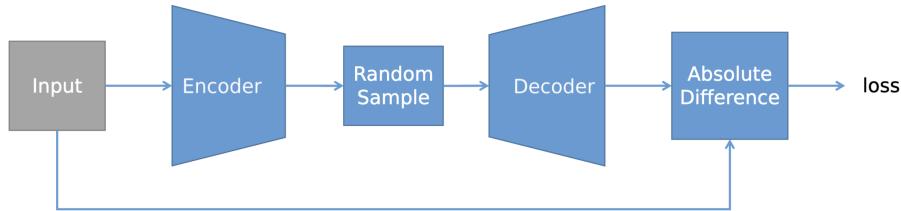
        self.decoder = nn.Sequential(
            nn.Linear(hidden_size, 300),
            nn.LeakyReLU(),
            nn.Linear(300, 300),
            nn.LeakyReLU(),
            nn.Linear(300, input_size)
        )

    def forward(self, x):
        noise_x = x + torch.randn(x.shape)
        latent = self.encoder(noise_x)
        reconstruction = self.decoder(latent)
        return reconstruction
```

5.7 What are Variational Autoencoders, and how do they differ from regular autoencoders?

I **Variational Autoencoders (VAEs)** sono un tipo di autoencoder disegnato non solo per imparare una rappresentazione compatta dei dati, ma anche per cercare di generare nuovi dati a partire dalla rappresentazione nello spazio latente dell'input.

Per fare ciò i VAEs utilizzano un approccio probabilistico, che permette di campionare nuovi punti nello spazio latente e generare nuovi dati. L'encoder di un VAE non trasforma direttamente l'input in una rappresentazione compatta, ma invece produce una distribuzione di probabilità dell'input. Il decoder prende un punto campionato dalla distribuzione e lo trasforma in un'output. Questo permette ai VAEs di generare nuovi dati a partire dalla rappresentazione compatta nello spazio latente.



5.8 How does the Reparametrization Trick work?

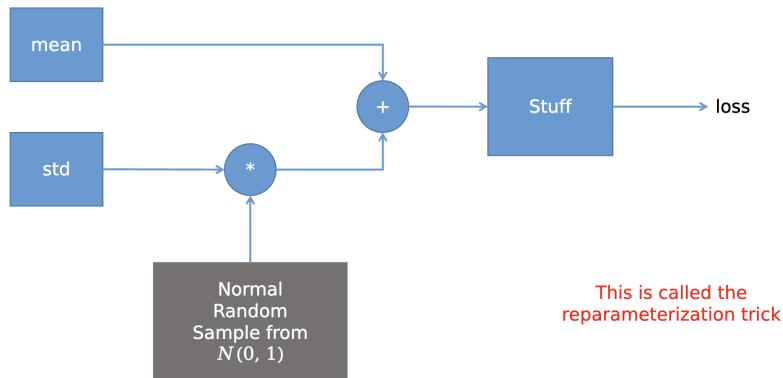
Il problema dei VAEs è che, la parte di sampling non è differenziabile, e quindi non può essere utilizzata durante la backpropagation. Per superare questo problema, i VAEs utilizzano un trucco chiamato **Reparametrization Trick**, che permette di campionare nuovi punti nello spazio latente in modo differenziabile. L'idea è calcolare i gradienti rispetto ai parametri della distribuzione di probabilità, che nel caso di una distribuzione normale, sono la media e la deviazione standard. Una variabile randomica $z \sim N(\mu, \sigma)$ può essere trasformata in $\hat{z} \sim N(0, 1)$ usando:

$$\hat{z} = \frac{z - \mu}{\sigma}$$

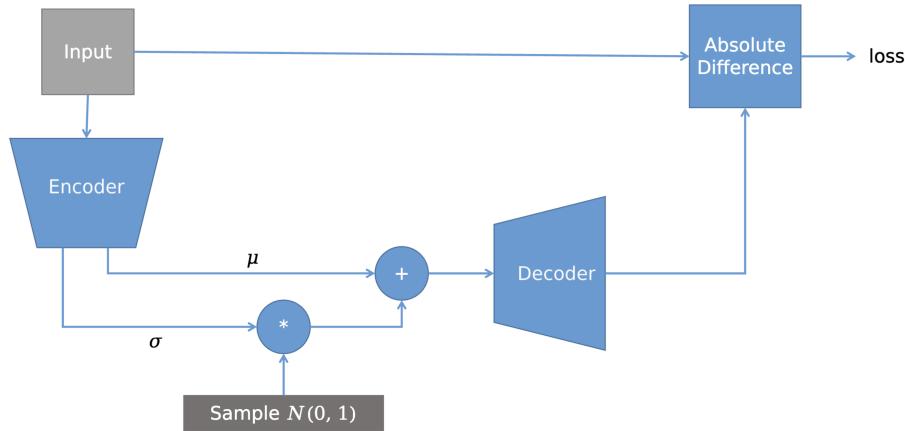
Nello stesso modo posso una variabile randomica $\hat{z} \sim N(0, 1)$ può essere trasformata in $z \sim N(\mu, \sigma)$ usando:

$$z = \mu + \sigma \cdot \hat{z}$$

Dove sia la somma che la moltiplicazione sono operazioni differenziabili, e quindi possono essere utilizzate durante la backpropagation.



5.9 Draw a diagram of a VAE and its computational graph.



5.10 Why VAEs are a 'generative' architecture?

I VAEs sono un'architettura generativa perché sono progettati per generare nuovi dati a partire dalla rappresentazione compatta nello spazio latente. Questo è possibile per via del campionamento nello spazio latente tramite una distribuzione di probabilità, per cui il decoder ricostruisce non proprio l'input originale, ma un input simile a quello originale, comunque nuovo.

5.11 What is the Kullback-Leibler divergence, and how is it used in VAEs?

La **Kullback-Leibler divergence** è una misura che quantifica quanto due distribuzioni di probabilità divergono:

$$D_{KL}(P||Q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} dx$$

In un VAE, la KL divergenza è utilizzata per misurare la differenza tra la distribuzione di probabilità della rappresentazione compatta e una distribuzione ben definita, come una distribuzione normale. Questo permette di regolarizzare la distribuzione della rappresentazione compatta, e di generare nuovi dati più realistici.

5.12 Describe a practical implementation of training a simple autoencoder on the MNIST dataset.

```

# Training
dataloaders = nn.DataLoader(dataset_MNIST, batch_size=32, shuffle=True)
model = Autoencoder()
loss_fn = nn.L1Loss() # Mean Absolute Error
optimizer = optim.Adam(model.parameters(), lr=0.01)
  
```

```

for epoch in range(num_epochs):
    for input, _ in dataloaders:
        output = model(input)
        loss = loss_fn(output, input)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

```

5.13 Explain the procedure for training a Variational Autoencoder on the MNIST dataset.

```

# Modello
class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_size, 300),
            nn.LeakyReLU(),
            nn.Linear(300, 300),
        )

        self.mu = nn.Linear(300, hidden_size)
        self.std = nn.Linear(300, hidden_size)

        self.decoder = nn.Sequential(
            nn.Linear(hidden_size, 300),
            nn.LeakyReLU(),
            nn.Linear(300, 300),
            nn.LeakyReLU(),
            nn.Linear(300, input_size)
        )

    def encode(self, x):
        latent = self.encoder(x)
        mu = self.mu(latent)
        std = torch.abs(self.std(latent))
        return mu, std

    def generate(self, mu, std):
        z = mu + std * torch.randn_like(mu)
        return self.decoder(z)

    def forward(self, x, kl=False):
        mu, std = self.encode(x)
        reconstruction = self.generate(mu, std)

        if kl:
            kl_divergence = 0.5*(std**2+mu**2-torch.log(std**2)-1).sum(1).mean(dim=0)
            return reconstruction, kl_divergence

        return reconstruction

```

5.14 How does the 'Face Swap' algorithm work?

Il **Face Swap** è un algoritmo che utilizza VAEs per generare nuove immagini a partire da due immagini di volti. L'idea è di utilizzare due VAEs, che condividono lo stesso Encoder, ma hanno Decoder diversi. In fase di treaining si passano due

immagini di volti diversi, e ci allenano due Decoder diversi uno a riconoscere il volto A e l'altro il volto B. In fase di inferenza, si passa un'immagine di un volto A all'encoder e si passa il risultato al decoder del volto B, e viceversa. Questo permette di generare nuove immagini di volti che combinano le caratteristiche dei due volti originali.

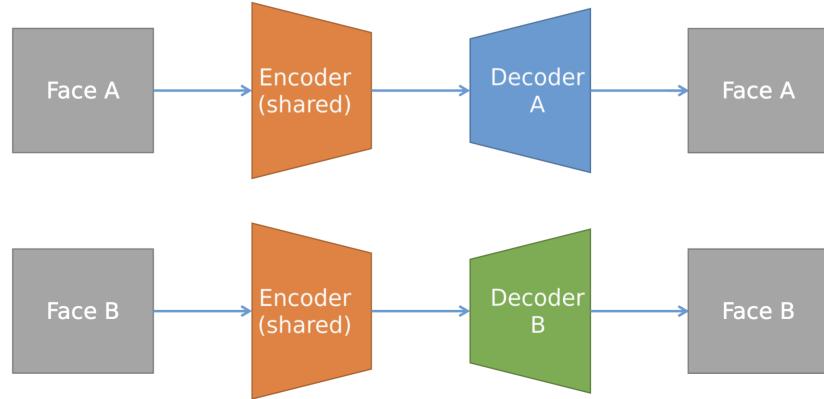


Figura 2: Face Swap Training Procedure

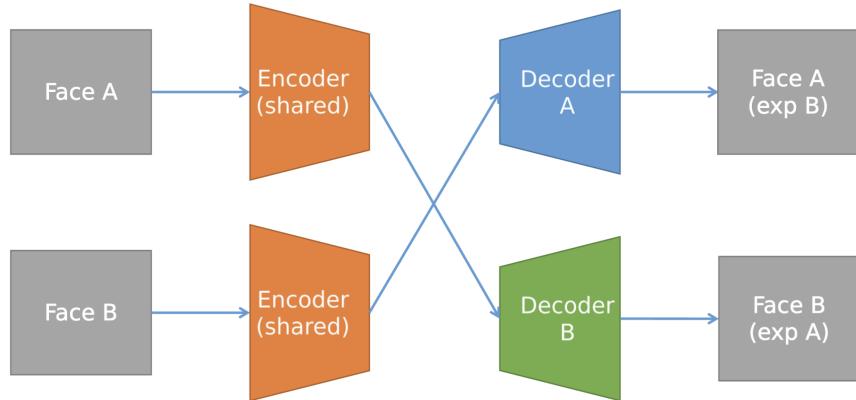
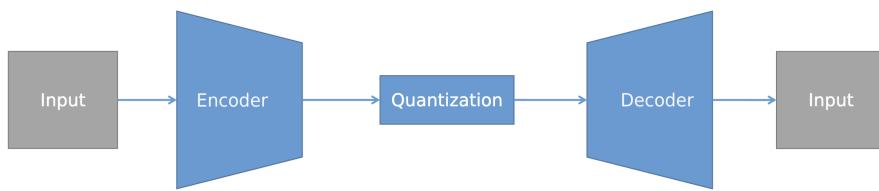


Figura 3: Face Swap Inference Procedure

6 Vector Quantized Variational Autoencoders

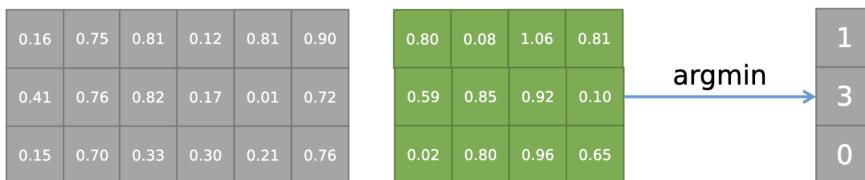
6.1 What is a Vector Quantized Variational Autoencoder (VQ-VAE)?

I **Vector Quantized Variational Autoencoders (VQ-VAEs)** sono un tipo rete neurale che combinano il concetto di VAEs con quello di Vector Quantization per imparare una rappresentazione discreta dei dati nello spazio latente.



Similmente ai VAEs, i VQ-VAE sono progettati per codificare i dati di input attraverso un Encoder che mappa i dati da una rappresentazione continua a una rappresentazione discreta, associandoli agli elementi di una specifica entità detta **codebook**. La rappresentazione continua è riorganizzata in una matrice, che ha le stesse colonne del codebook, e il numero di righe è uguale al numero di valori della codifica dell'input.

Viene poi associato ogni codifica dell'input a ogni vettore del codebook, alla ricerca di quello più simile. Questa operazione è detta **Vector Quantization**. Per effettuare questa operazione semplicemente si calcola la distanza fra gli elementi del codebook e i vettori dell'input, e si associa l'input al vettore del codebook più vicino.

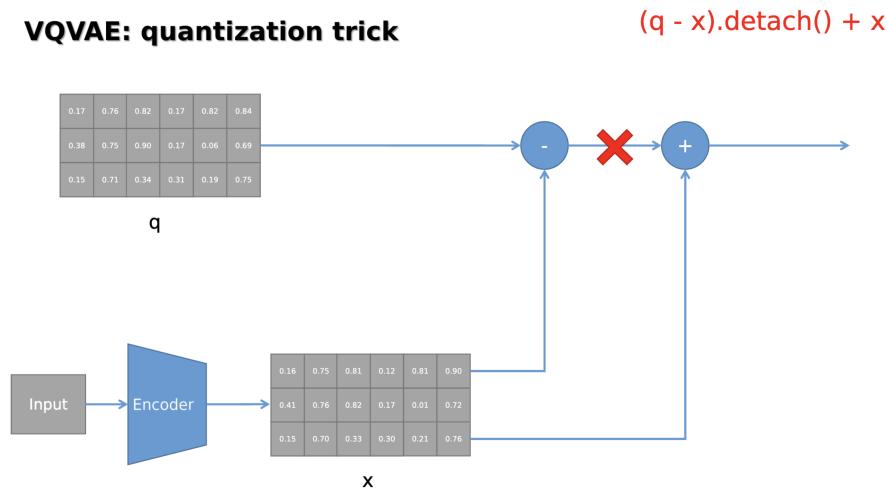


0.15	0.17	0.89	0.38
0.71	0.76	0.92	0.75
0.34	0.82	0.56	0.90
0.31	0.17	0.72	0.17
0.19	0.82	0.43	0.06
0.75	0.84	0.94	0.69

- 6.2 Explain the process of mapping input data to a continuous latent space and then to discrete codes in a VQ-VAE.
- 6.3 How does the vector quantization process work in a VQ-VAE, and what is the role of the codebook?
- 6.4 How does the quantization trick work?

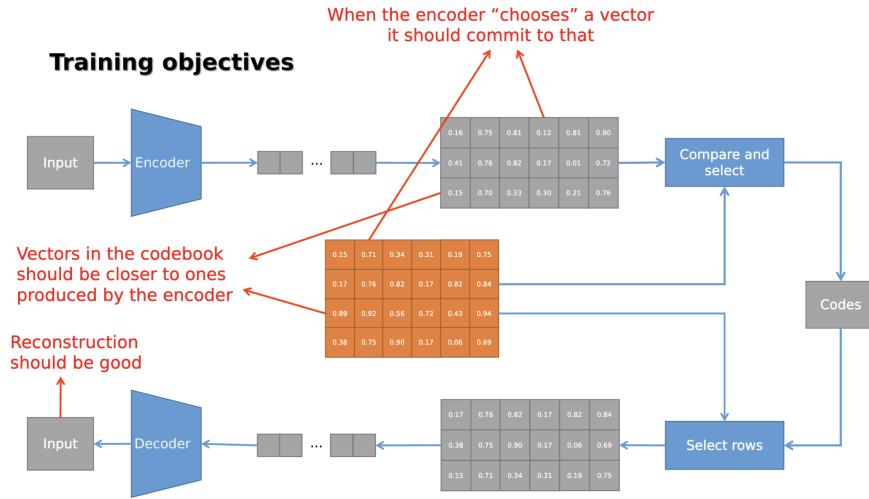
Il **quantization trick** è un trucco utilizzato nei VQ-VAE per permettere la back-propagation durante l'allenamento. Il problema è che la, durante la quantizzazione, bisogna cercare il minimo all'interno del vettore delle distanze fra input e codebook, e questa operazione non è differenziabile in fase di backward. Per superare questo problema, si utilizza il quantization trick, che consiste nel scollegare dal grafo computazionale l'arco che calcola la differenza.

Chiamiamo x l'uscita dell'encoder, e q la matrice del codebook associata a x , quello che vogliamo fare è trasformare x in q . Per farlo bisogna fare l'operazione di `detach()`, calcolando la differenza fra il vettore del codebook e dell'input come: $(q - x).detach() + x$:



Da notare che questo trucco non ha fondamenti matematici, infatti funziona solo se il codebook è molto vicino ai valori che l'encoder produce.

6.5 Draw a diagram of a VQ-VAE and its computational graph



6.6 Describe the function of the 'cdist' function in PyTorch in the context of VQ-VAEs.

La funzione **cdist** di PyTorch è una funzione che calcola la distanza tra due tensori e crea una matrice dove ogni elemento è la distanza fra due righe dei tensori. Vengono confrontate tutte le righe del primo tensore con tutte le righe del secondo tensore.

6.7 What is the responsibility of the decoder in a VQ-VAE, and how does it utilize discrete codes for data reconstruction or generation?

Il Decocer di un VQ-VAE è responsabile di ricostruire l'input a partire dalla rappresentazione discreta dello spazio latente, che sono poi i vettori del codebook più vicini all'input. Il decoder prende i vettori del codebook e li trasforma in un'output, che è la ricostruzione dell'input originale.

6.8 Discuss the types of losses used in training a VQ-VAE, specifically reconstruction, codebook, and commitment loss.

Per allenare un VQ-VAE vengono utilizzati tre tipi di loss:

- **Reconstruction Loss:** Misura la differenza tra l'input originale e l'output predetto dal decoder.
- **Codebook Loss:** Misura la differenza tra i vettori del codebook e i vettori dell'input.

- **Commitment Loss:** Misura la differenza tra i vettori dell'input e i vettori del codebook, e serve a regolarizzare la rappresentazione dello spazio latente, forzandola ad essere più simile ai vettori del codebook.

6.9 Explain the practical steps involved in training a VQ-VAE on the MNIST dataset.

```

# Modello
class VQVAE(torch.nn.Module):
    def __init__(self, *, input_dim, codebook_dim, vector_dim, num_embeddings):
        super(VQVAE, self).__init__()

        self.input_dim = input_dim
        self.codebook_dim = codebook_dim
        self.vector_dim = vector_dim
        self.num_embeddings = num_embeddings

        # ENCODER
        self.encoder = torch.nn.Sequential(
            torch.nn.Linear(input_dim, 300),
            torch.nn.LeakyReLU(),
            torch.nn.Linear(300, 300),
            torch.nn.LeakyReLU(),
            torch.nn.Linear(300, vector_dim*num_embeddings),
        )

        self.vq = VectorQuantize(
            dim = vector_dim,
            codebook_size = codebook_dim,
            decay = 0.8,
            commitment_weight = 1.0
        )

        # DECODER
        self.decoder = torch.nn.Sequential(
            torch.nn.Linear(vector_dim*num_embeddings, 300),
            torch.nn.LeakyReLU(),
            torch.nn.Linear(300, 300),
            torch.nn.LeakyReLU(),
            torch.nn.Linear(300, input_dim),
            torch.nn.Sigmoid()
        )

    def encode(self, x):
        embeddings = embeddings.view(embeddings.shape[0],
                                      self.num_embeddings,
                                      self.vector_dim)
        return embeddings

    def quantize(self, embeddings):
        quantized, codes, commit_loss = self.vq(embeddings)
        return quantized, codes, commit_loss

    def decode(self, x):
        return self.decoder(x)

    def decode_from_codes(self, codes):
        embeddings = self.vq.codebook[codes]
        flat_quantized = embeddings.view(embeddings.shape[0], -1)
        return self.decoder(flat_quantized)

```

```

def forward(self, x, return_codes=False):
    embeddings = self.encode(x)
    quantized, codes, commit_loss = self.quantize(embeddings)
    flat_quantized = quantized.view(quantized.shape[0], -1)
    reconstructed = self.decode(flat_quantized)

    if return_codes:
        return reconstructed, codes, commit_loss
    return reconstructed, commit_loss

```

6.10 Explain how to generate images from random codes in a VQ-VAE and the insights it provides.

Per generare immagini a partire da codici casuali in un VQ-VAE, bisogna passare i codici randomici, codificarli col codebook, e poi passarli al Decoder che genererà la nuova immagine. In codice:

```

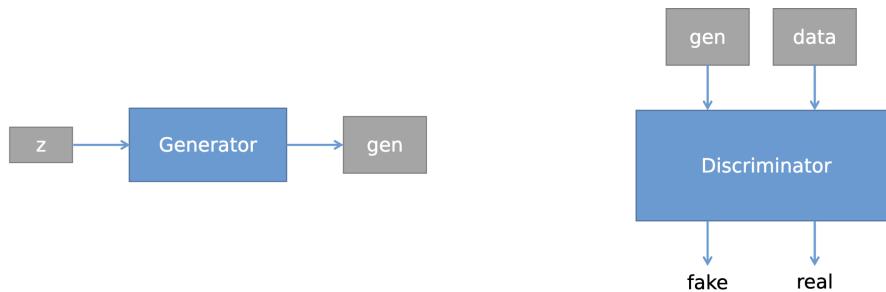
random_codes = torch.randint(0, 8, (8, 8, 16)) # 8x8 random codes
for i in range(8):
    for j in range(8):
        reconstructed = model.decode_from_codes(random_codes[i, j].view(1, 16))
        plt.subplot(8, 8, i*8+j+1)

```

7 Generative Adversarial Networks

7.1 What is a Generative Adversarial Network (GAN)?

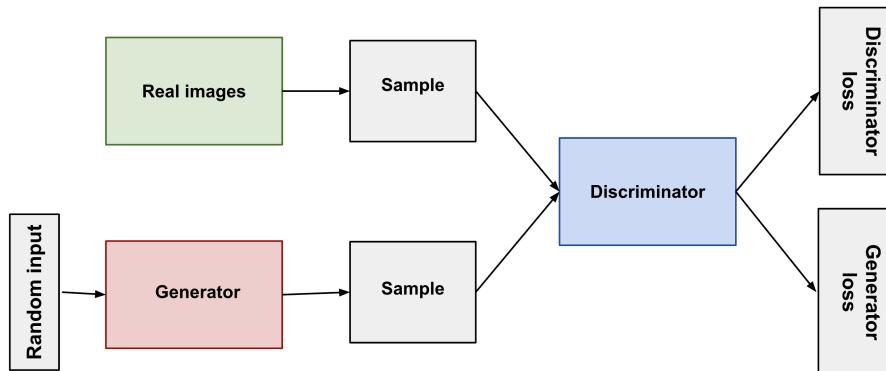
Una **Generative Adversarial Network (GAN)** è un tipo di deep neural network che è progettata per generare nuovi dati. Essa è composta da due reti neurali distinte: il **generatore** e il **discriminatore**. Il generatore prende in input un vettore di rumore casuale e cerca di generare nuovi dati, mentre il discriminatore prende in input un dato e cerca di distinguere se è un dato reale o generato dal generatore. Le due reti sono addestrate in competizione tra loro, dove il generatore cerca di ingannare il discriminatore generando dati sempre più realistici, e il discriminatore cerca di distinguere i dati reali da quelli generati.



7.2 Describe the architecture of GANs, including the roles of the generator and discriminator.

Il **generatore** di un GAN è responsabile di generare nuovi dati a partire da un vettore di rumore casuale. Esso prende in input un vettore di rumore casuale e cerca di generare dati sempre più realistici, tendendo di ingannare il discriminatore.

Il **discriminatore** di un GAN è responsabile di distinguere i dati reali da quelli generati dal generatore, esso è un classificatore binario in grado di distinguere i dati reali da quelli generati.



7.3 Explain how the discriminator functions as a binary classifier in GANs.

Il **discriminatore** di un GAN funziona come un classificatore binario, che prende in input un dato e cerca di distinguere se è un dato reale o generato dal generatore. Il discriminatore è addestrato per classificare i dati in due classi: real o fake. Durante l'allenamento, il discriminatore cerca di massimizzare la sua accuratezza, distinguendo i dati reali da quelli generati dal generatore.

7.4 Outline the steps involved in training the discriminator in a GAN.

Il **discriminatore** di un GAN è addestrato per distinguere i dati reali da quelli generati dal generatore. Durante l'allenamento, il discriminatore cerca di massimizzare la sua accuratezza, distinguendo i dati reali da quelli generati dal generatore. Il discriminatore è addestrato indipendentemente dal generatore, ma in competizione con esso.

```
def train_discriminator(*, optimizer, generator, discriminator, real_images):
    batch_size = real_images.shape[0]
    device = real_images.device

    z = torch.randn(batch_size, generator.latent_dim).to(device)
    fake_images = generator(z)

    fake_label = torch.zeros(batch_size).to(device) # 0 = fake
    real_label = torch.ones(batch_size).to(device) # 1 = real

    fake_logits = discriminator(fake_images)[:, 0]
    real_logits = discriminator(real_images)[:, 0]

    fake_loss = torch.nn.functional.binary_cross_entropy(fake_logits, fake_label)
    real_loss = torch.nn.functional.binary_cross_entropy(real_logits, real_label)

    loss = (fake_loss + real_loss) / 2

    loss.backward()
    optimizer.step()

    return fake_loss.item(), real_loss.item(), fake_accuracy.item(), real_accuracy.item()
```

7.5 Explain the training process of the generator in a GAN.

Il **generatore** di un GAN è addestrato per generare dati sempre più realistici. Durante l'allenamento, il generatore cerca di ingannare il discriminatore generando dati che il discriminatore non riesce a distinguere dai dati reali:

```
def train_generator(*, optimizer, generator, discriminator, real_images):
    batch_size = real_images.shape[0]
    device = real_images.device

    z = torch.randn(batch_size, generator.latent_dim)
    fake_images = generator(z)
```

```

real_label = torch.ones(batch_size).to(device)
fake_logits = discriminator(fake_images)[:, 0]

loss = torch.nn.functional.binary_cross_entropy(fake_logits, real_label)
loss.backward()
optimizer.step()

```

La cosa importatne è azzerare i gradienti del discriminatore prima di fare il backpropagation del generatore e viceversa:, altrimenti i gradienti del discriminatore verrebbero aggiornati durante il backpropagation del generatore (e viceversa):

```

generator = Generator(latent_dim=64, output_dim=28).to("mps")
discriminator = Discriminator(input_dim=28).to("mps")

train_dl = torch.utils.data.DataLoader(train_mnist, batch_size=64, shuffle=True)

generator_optimizer = torch.optim.Adam(generator.parameters(), lr=2e-4) # lr = 0.0002
discriminator_optimizer = torch.optim.Adam(discriminator.parameters(), lr=2e-4)

history = []

for epoch in range(0, 20):
    for real_images, _ in tqdm(train_dl):
        real_images = real_images.to("mps")

        discriminator_optimizer.zero_grad()
        generator_optimizer.zero_grad()
        d_fake_loss, d_real_loss, d_fake_acc, d_real_acc = train_discriminator(
            generator=generator,
            discriminator=discriminator,
            optimizer=discriminator_optimizer,
            real_images=real_images
        )

        discriminator_optimizer.zero_grad()
        generator_optimizer.zero_grad()
        g_loss, g_accuracy = train_generator(
            generator=generator,
            discriminator=discriminator,
            optimizer=generator_optimizer,
            real_images=real_images
        )
    
```

7.6 Draw a diagram of a GAN and its computational graph.

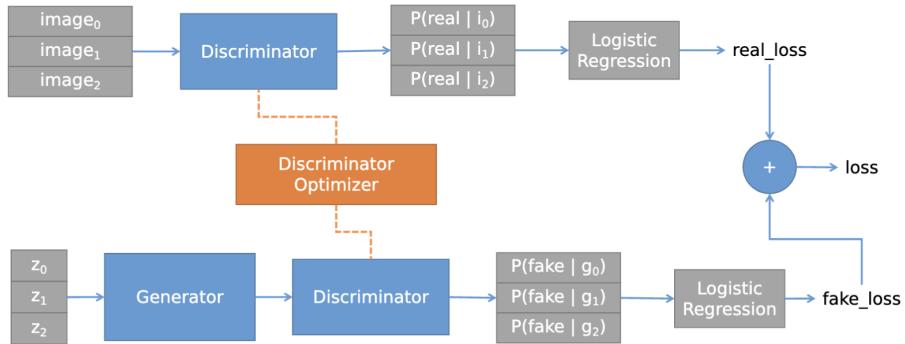


Figura 4: GAN Computational Graph - Discriminator

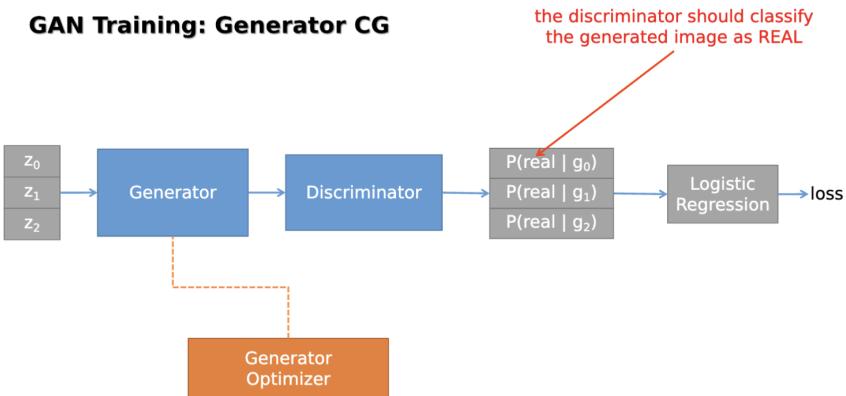


Figura 5: GAN Computational Graph - Generator

7.7 What is an Auxiliary Classifier GAN (ACGAN), and how does it differ from standard GANs?

Un **Auxiliary Classifier GAN (ACGAN)** è una variante di GAN dove il discriminatore non solo cerca di distinguere i dati reali da quelli generati, ma anche di classificare le immagini in diverse classi. Questo permette di generare immagini che appartengono a classi specifiche.

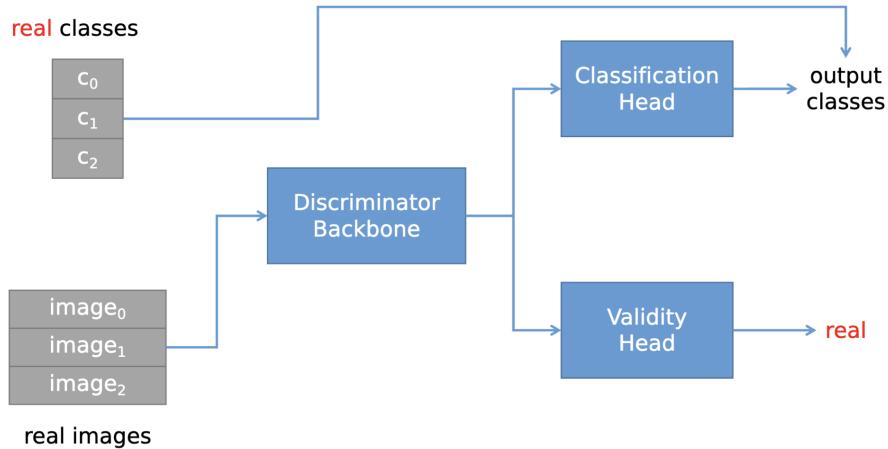


Figura 6: GAN Computational Graph - Discriminator

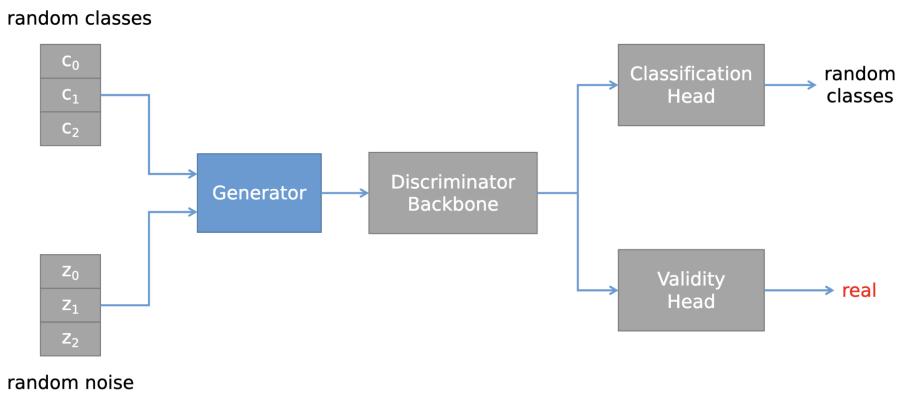


Figura 7: GAN Computational Graph - Generator

7.8 Discuss the challenges in training GANs and strategies to overcome them.

Allenare una GAN è un processo molto delicato, perché non è facile trovare un equilibrio tra il generatore e il discriminatore. Le due reti per essere ben addestrate non devono predominare l'una dall'altra, ed è facile che ciò accada. Esistono però dei trucchi per aiutare l'allenamento di una GAN:

- **Normalizzare** l'immagine fra -1 e 1
- Preferire l'uso di **CNN** rispetto a **MLP**
- Mettere un *BatchNorm2d* dopo ogni layer
- Non usare MaxPooling, ma preferire l'uso dello **stride**
- Inizializzare i pesi delle convoluzionali a quasi 0 (es. $N(0, 0.02)$)
- Inizializzare i pesi delle batchnorm a quasi 1 (es. $N(1, 0.02)$)

7.9 Describe the process of training a Deep Convolutional GAN (DCGAN) on the MNIST dataset.

7.10 How can the balance between randomness and class information be maintained in an ACGAN?

Questo è bilanciato dall'utilizzo di due loss: una per la classificazione binaria e una per la classificazione delle classi delle immagini. Questo permette di mantenere un equilibrio tra la generazione di immagini casuali e la generazione di immagini appartenenti a classi specifiche, perché il discriminatore è addestrato a classificare le immagini in base anche alla classe di appartenenza.

Risposta Domenico

Questo può essere fatto semplicemente sfruttando il *torch.nn.Embedding* che permette di proiettare in un vettore il numero di classi memorizzandolo all'interno di un dizionario. Nell'operazione forward del generatore, per il quale passeremo le classi e un po' di rumore casuale come input, creeremo innanzitutto l'embedding a partire dalle classi e poi lo moltiplicheremo per il rumore.

8 Advanced Architectures

- 8.1 What is YOLO (You Only Look Once) in the context of object detection, and how does it perform real-time detection?
- 8.2 Explain how YOLO divides an image into a grid for object detection and how it predicts bounding boxes and class probabilities.
- 8.3 Describe Non-Maximum Suppression (NMS) and its role in object detection.
- 8.4 What is Faster R-CNN, and how does it combine a Region Proposal Network (RPN) with a CNN?
- 8.5 Explain the U-Net architecture and its application in semantic image segmentation.
- 8.6 What is CLIP (Contrastive Language-Image Pretraining) by OpenAI, and how does it combine vision and language?
- 8.7 Discuss Denoising Diffusion Probabilistic Models (DD-PM) and Latent Diffusion, and their role in generating high-quality samples.
- 8.8 Describe the DALL-E 2 architecture