Andrea Augello
Department of Engineering, University of Palermo, Italy

Addestrare una rete neurale

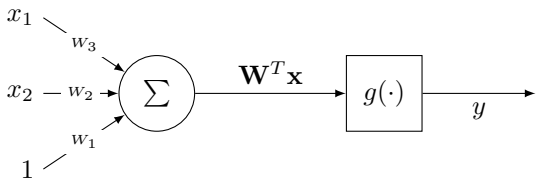
Università degli Studi di Palermo

Reti neurali

Reti neurali

- Una rete neurale è un modello matematico ispirato al sistema nervoso.
- Una rete neurale è composta da un insieme di neuroni artificiali.
- Un neurone è un modello matematico ispirato al neurone biologico.
- Un neurone è composto da:
  - un insieme di connessioni in ingresso
  - una funzione di aggregazione
  - una funzione di attivazione

Un neurone artificiale



- Il neurone riceve un insieme di input $\mathbf{x} = (x_1, x_2, \ldots, x_n, 1)$.
- Ogni input x_i è moltiplicato per un peso W_i.
- I pesi $\mathbf{W}$ sono i parametri della rete neurale.
- I pesi $\mathbf{W}$ sono inizializzati casualmente.
- I pesi $\mathbf{W}$ sono aggiornati durante l'addestramento.

Reti neurali

- Addestrare una rete neurale significa trovare i valori dei pesi che minimizzano una funzione di errore sul training set.
- Per trovare i pesi ottimali, è necessario utilizzare un algoritmo di ottimizzazione.
- L'algoritmo di ottimizzazione più semplice è la discesa del gradiente.
- La discesa del gradiente è un algoritmo iterativo che può essere applicato a qualsiasi funzione differenziabile.

PyTorch

PyTorch

- PyTorch è un framework per il deep learning particolarmente diffuso nella comunità scientifica.
- PyTorch supporta il calcolo su GPU.
- PyTorch gestisce automaticamente la differenziazione e la backpropagation.
- PyTorch fornisce un'implementazione efficiente di reti neurali e numerosi algoritmi di ottimizzazione.

PyTorch

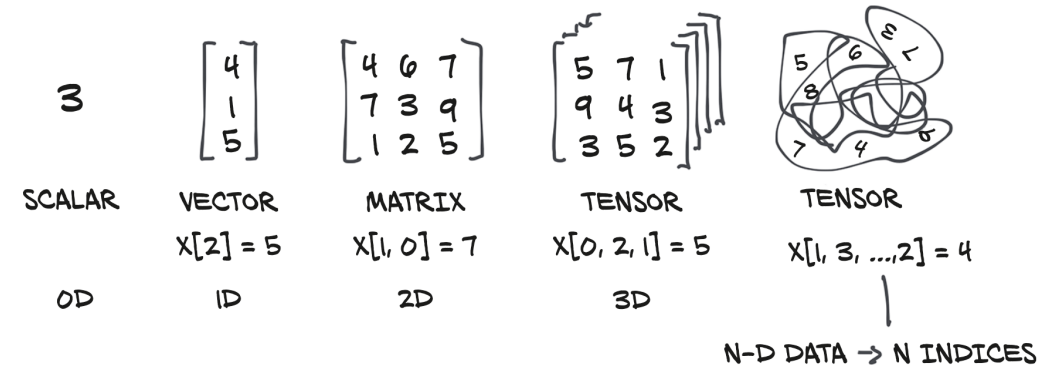
Useremo principalmente le seguenti componenti:

- `torch.tensor`: array multidimensionali.
- `torch.nn`: moduli per la definizione di reti neurali.
  - funzioni di attivazione
  - layer
  - loss
- `torch.optim`: moduli per l'ottimizzazione.

Tutto inizia con un tensore

Tutto inizia con un tensore

Le reti neurali operano su numeri reali (IA sub-simbolica). PyTorch utilizza i Tensori (array multidimensionali) come struttura dati principale.



Differenza tra tensori e liste di liste

- I tensori contengono un singolo tipo di dato e hanno dimensioni fisse (più efficienti da manipolare).
- Un tensore può risiedere nella RAM o nella GPU.
- L'oggetto tensore ha una serie di attributi e metodi aggiuntivi.
- I tensori supportano indicizzazioni avanzate.
- Un tensore può ricordare la storia delle operazioni che lo hanno generato.

In questo corso, quasi tutto il codice che va a manipolare direttamente i tensori sarà già fornito.

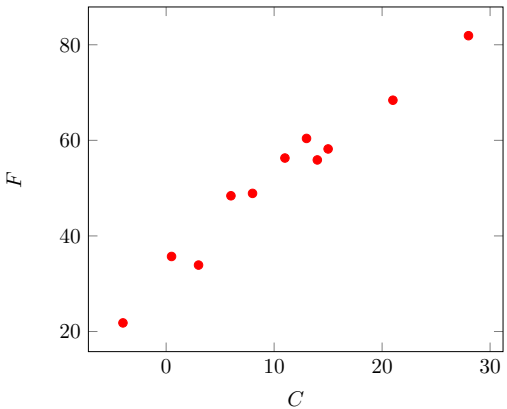
Esempio giocattolo di una rete neurale

Esempio giocattolo di una rete neurale

Proviamo a implementare una rete neurale giocattolo in PyTorch per un semplice problema di regressione: imparare a convertire gradi Celsius in gradi Fahrenheit.

Il dataset delle osservazioni (rumorose):

C	F
0.5	35.7
14.0	55.9
15.0	58.2
28.0	81.9
11.0	56.3
8.0	48.9
3.0	33.9
-4.0	21.8
6.0	48.4
13.0	60.4
21.0	68.4



Una semplice rete neurale in PyTorch

Creiamo una rete neurale con un singolo neurone:

```
import torch
import torch.nn as nn

class ToyNet(nn.Module):
    def __init__(self):
        super(ToyNet, self).__init__()
        self.fc1 = nn.Linear(1, 1)

    def forward(self, x):
        x = self.fc1(x)
        return x
```

`nn.Module` è la classe base per tutte le reti neurali in PyTorch.

nn.Module

Un modulo può contenere come attributi uno o più parametri che devono essere ottimizzati.

Un modulo può avere come attributi anche altri moduli, PyTorch gestirà automaticamente la propagazione del gradiente anche a questi moduli.

Nota bene: PyTorch non gestisce automaticamente la propagazione del gradiente per gli attributi che non sono moduli, ovvero per moduli all'interno di altre strutture dati (liste, dizionari, etc.).

Proviamo la nostra rete neurale

```
dataset = torch.Tensor([ [0.5, 35.7],
                         [14.0, 55.9],
                         [15.0, 58.2],
                         [28.0, 81.9],
                         [11.0, 56.3],
                         [8.0, 48.9],
                         [3.0, 33.9],
                         [-4.0, 21.8],
                         [6.0, 48.4],
                         [13.0, 60.4],
                         [21.0, 68.4]])
model = ToyNet()
X = dataset[:, 0].unsqueeze(1)
y = dataset[:, 1].unsqueeze(1)
plot(X, y, model, "Before training", temp=False)
```

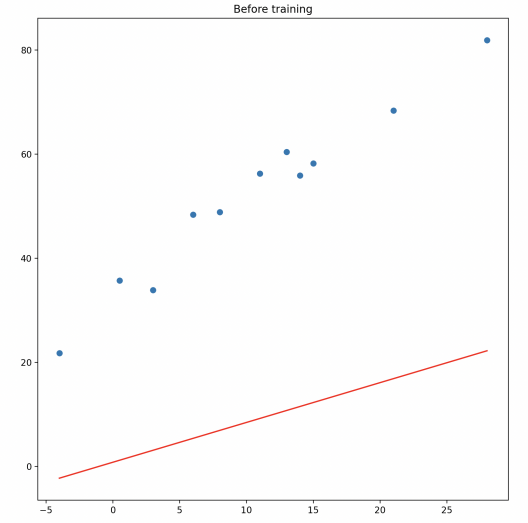
Nota bene: Chiamare direttamente il metodo `forward` di un modulo è un errore: non viene eseguita una serie di operazioni che PyTorch gestisce automaticamente in background.

Proviamo la nostra rete neurale

Se proviamo a stampare l'output del modello (`print(model(X))`), otteniamo un tensore particolare: oltre al valore numerico, contiene anche informazioni sul gradiente.

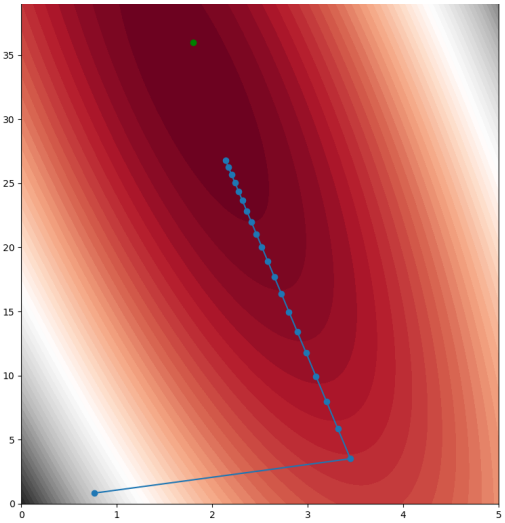
```
tensor([[ 1.2123],
        [11.5335],
        [12.2981],
        [22.2371],
        [ 9.2399],
        [ 6.9463],
        [ 3.1236],
        [-2.2281],
        [ 5.4172],
        [10.7690],
        [16.8853]], grad_fn=<AddmmBackward0>)
```

La rete non addestrata lascia a desiderare



La discesa del gradiente

- La discesa del gradiente è un algoritmo iterativo per minimizzare una funzione differenziabile.
- L'algoritmo calcola il gradiente della funzione in un punto e si sposta nella direzione opposta.
- Il passo è proporzionale al gradiente e ad un parametro chiamato *learning rate*.



Aggiungiamo un metodo fit

```
    def fit(self, x, y, epochs=100, lr=0.01):
        losses = []
        optimizer = torch.optim.SGD(self.parameters(), lr=lr)
        criterion = nn.MSELoss()
        for _ in range(epochs):
            optimizer.zero_grad()
            output = self(x)
            loss = criterion(output, y)
            losses.append(loss.item())
            loss.backward()
            optimizer.step()
        return losses
```

Aggiungiamo un metodo fit

```
    def fit(self, x, y, epochs=100, lr=0.01):
        losses = []
        optimizer = torch.optim.SGD(self.parameters(), lr=lr)
        criterion = nn.MSELoss()
        for _ in range(epochs):
            optimizer.zero_grad()
            output = self(x)
            loss = criterion(output, y)
            losses.append(loss.item())
            loss.backward()
            optimizer.step()
        return losses
```

Cosa sta succedendo?

Aggiungiamo un metodo fit

Passi preliminari:

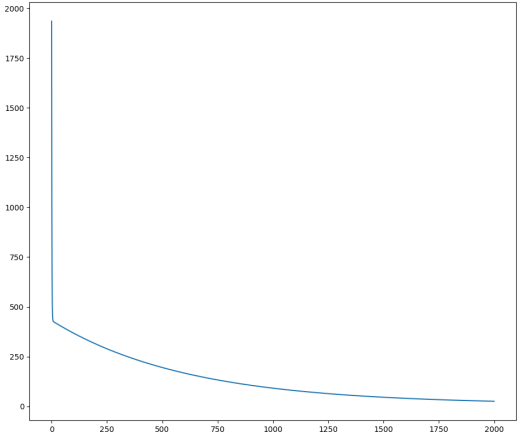
- `torch.optim.SGD` implementa la discesa del gradiente stocastica. Vuole in input i parametri da ottimizzare e il learning rate.
- `self.parameters()` restituisce tutti i parametri del modello per passarli all'ottimizzatore.
- `nn.MSELoss` implementa l'errore quadratico medio, da minimizzare per i problemi di regressione come il nostro. PyTorch implementa molte altre funzioni di loss da utilizzare a seconda del problema. È anche possibile definire una propria funzione di loss.
- `lossess=[]` è una lista che conterrà i valori dell'errore ad ogni iterazione. Non è strettamente necessaria, ma può essere utile per monitorare l'andamento dell'addestramento.
- `epochs` determina il numero di iterazioni dell'algoritmo di ottimizzazione.

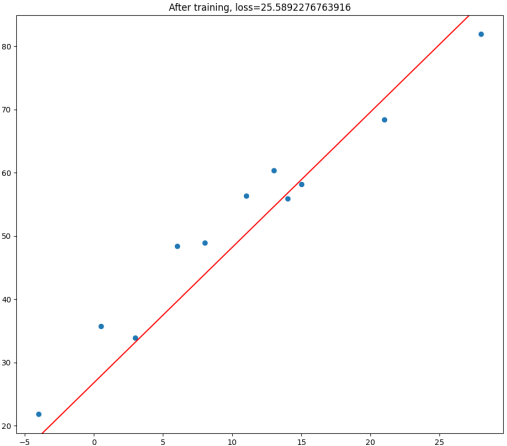
Aggiungiamo un metodo fit

Per ogni epoca di addestramento:

- `optimizer.zero_grad()` azzera i gradienti dei parametri calcolati nella precedente epoca.
- `output = self(x)` calcola l'output della rete neurale per l'input `x`.
- `loss = criterion(output, y)` calcola l'errore tra l'output della rete e il target `y`.
- la loss è un tensore che contiene il valore dell'errore e la storia delle operazioni che hanno portato a quel valore. Con `.item()` si estrae il valore numerico dell'errore che verrà aggiunto alla lista `losses`.
- `loss.backward()` accumula il gradiente della loss sui parametri della rete.
- `optimizer.step()` aggiorna i parametri della rete in base al gradiente accumulato di una certa quantità proporzionale al learning rate. Algoritmi di ottimizzazione diversi possono richiedere parametri aggiuntivi e, per esempio, possono tenere conto del gradiente accumulato in più epoche.

Il risultato dell'addestramento



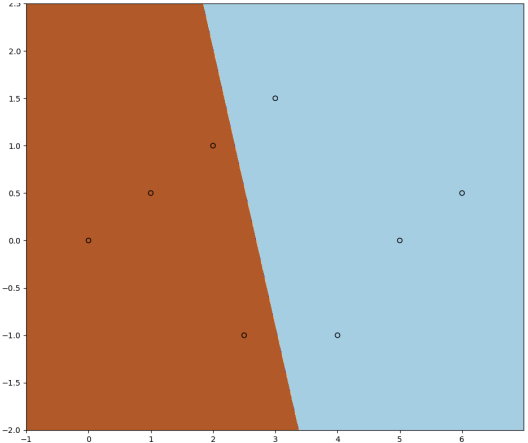


- Ground truth: Fahrenheit = 1.8 * Celsius + 32
- Legge appresa: Fahrenheit = 2.14 * Celsius + 26.79

Un problema di classificazione

Un problema di classificazione

Proviamo ora a implementare una rete neurale per un problema di classificazione: riconoscere i punti appartenenti ad una di due classi:



Un problema di classificazione

Differenze chiave rispetto al problema di regressione:

- La rete neurale deve avere un numero di output pari al numero di classi.
- L'ingresso della rete adesso è composto da due feature.
- La funzione di loss deve essere la cross-entropy.

Minibatch e dataloader

Discesa del gradiente stocastica

Quando lavoriamo con dataset meno triviali, non è possibile caricare in memoria l'intero dataset per addestrare la rete.

Per ovviare a questo problema, possiamo suddividere il dataset in mini-batch e addestrare la rete su un mini-batch alla volta.

Per batch sufficientemente grandi, il gradiente del mini-batch è una buona approssimazione del gradiente sul dataset completo, accelerando l'addestramento. Batch troppo piccoli, invece, possono portare instabilità nell'addestramento.

Dataloader

- PyTorch fornisce una classe `DataLoader` per gestire i dataset.
- `DataLoader` permette di caricare i dati in mini-batch, randomizzare l'ordine delle osservazioni, e gestire il caricamento in parallelo.
- Il dataloader è iterabile e restituisce un mini-batch alla volta, recuperando i campioni del dataset in maniera "lazy".
- È possibile creare un `DataLoader` per il proprio dataset creando una classe che estende `torch.utils.data.Dataset`.

Dataset

Una classe che estende dataset deve implementare tre metodi:

- `__init__`: inizializza il dataset, per esempio inizializzando le cartelle in cui andare a cercare i dati.
- `__len__`: restituisce la lunghezza del dataset.
- `__getitem__`: restituisce l'elemento `i`-esimo del dataset sotto forma di tupla `(input, target)` (se `target` è presente).