Andrea Augello
Department of Engineering, University of Palermo, Italy

Addestrare una rete neurale

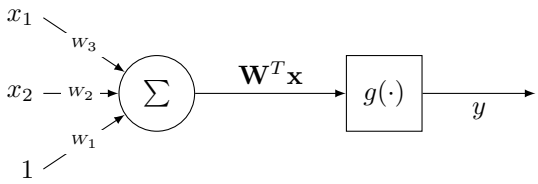
Università degli Studi di Palermo

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

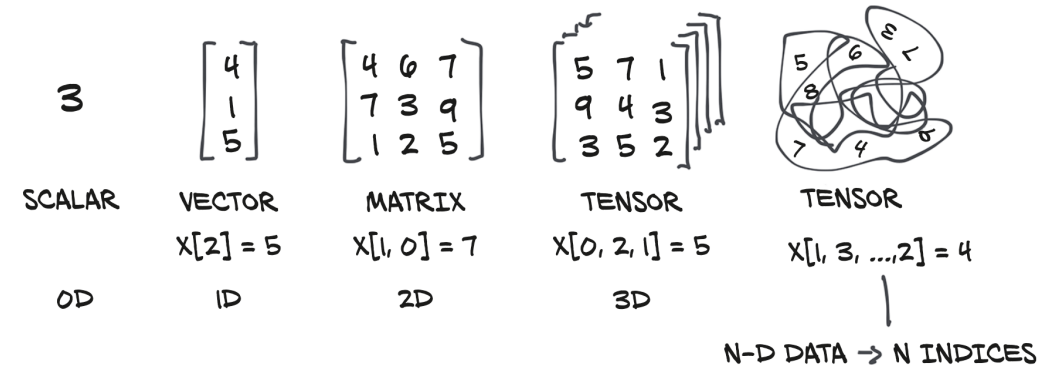
Useremo principalmente le seguenti componenti:

- `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.

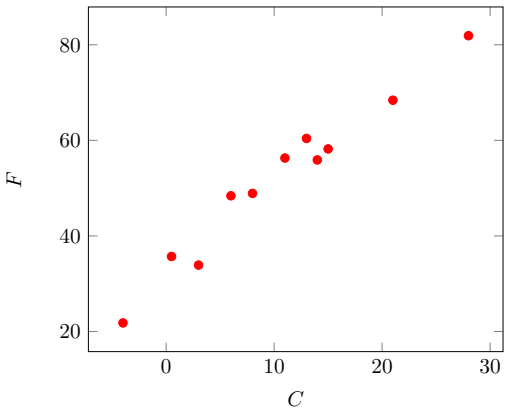
Esempio giocattolo di una rete neurale

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.

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.

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

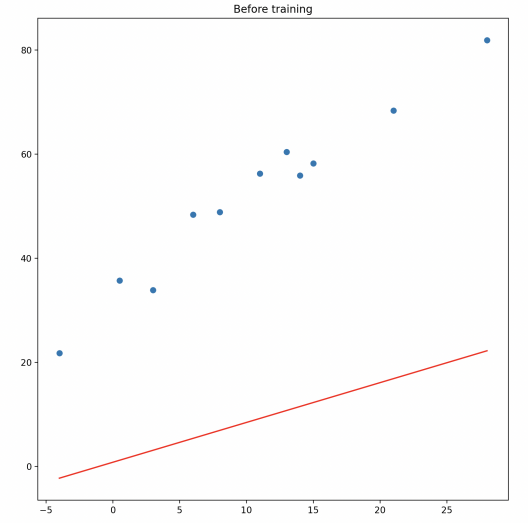
```
dataset = torch.Tensor([ [0.5, 35.7],
                         [14.0, 55.9],
                         [15.0, 58.2],
                         [28.0, 81.9],
                         [11.0, 56.3],
                         [8.0, 48.9],
                         [3.0, 33.9],
                         [-4.0, 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)
```

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.

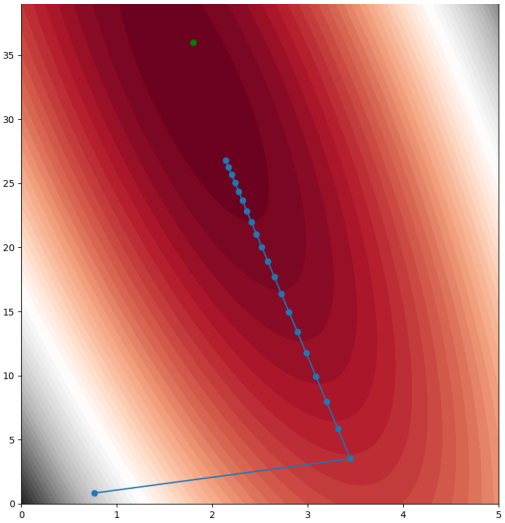
```
tensor([[ 1.2123],
        [11.5335],
        [12.2981],
        [22.2371],
        [ 9.2399],
        [ 6.9463],
        [ 3.1236],
        [-2.2281],
        [ 5.4172],
        [10.7690],
        [16.8853]], grad_fn=<AddmmBackward0>)
```

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:

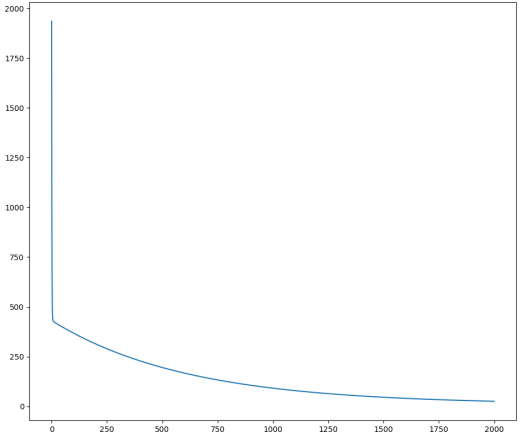
- `torch.optim.SGD` implementa la discesa del gradiente stocastica. Vuole in input i parametri da ottimizzare e il learning rate.
- `self.parameters()` restituisce tutti i parametri del modello per passarli all'ottimizzatore.
- `nn.MSELoss` implementa l'errore quadratico medio, da minimizzare per i problemi 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.

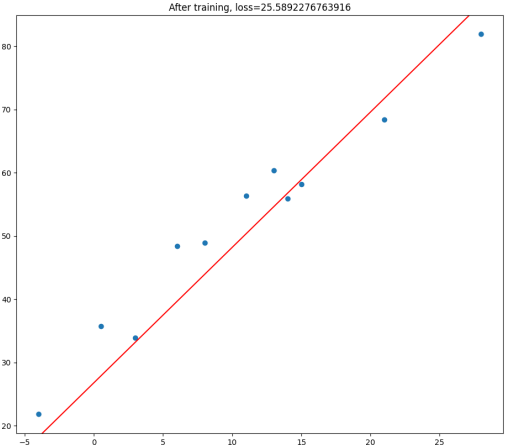
Aggiungiamo un metodo fit

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



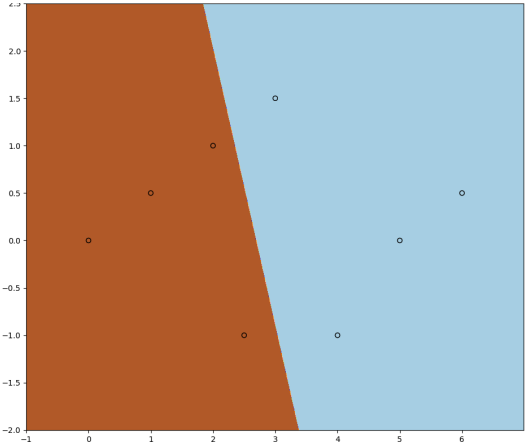


- Ground truth: Fahrenheit = 1.8 * Celsius + 32
- Legge appresa: Fahrenheit = 2.14 * Celsius + 26.79

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.

Scelta degli iperparametri

Scelta degli iperparametri

- Ci concentreremo su un problema di apprendimento supervisionato
- L'obiettivo è quello di approssimare una funzione f che mappa un vettore di input $\vec{x}$ in un valore reale y:

$$f : \mathbb{R}^n \to \mathbb{R}$$

- La funzione f è sconosciuta, ma si hanno a disposizione m coppie $(\vec{x}_i, y_i)$, dette esempi di addestramento, che sono estratti da f e che vengono utilizzati per approssimarla

Scelta degli iperparametri

- Per un'ampia classe di funzioni è possibile ottenere una approssimazione arbitrariamente precisa attraverso reti neurali sufficientemente grandi.
- Nella pratica, per n sufficientemente piccolo, è possibile ottenere buoni risultati con reti neurali di dimensione ridotta a patto di calcolare delle feature appropriate a partire dai dati di input.

Nota bene

Questo ragionamento può essere esteso anche a problemi di classificazione, considerando la funzione f come il confine di decisione (non necessariamente lineare) tra le classi.

I due dataset

I due dataset

Useremo come esempio due dataset sintetici generati con le seguenti funzioni:

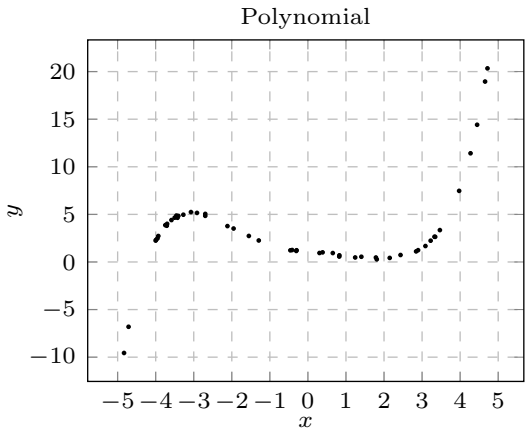
Una funzione polinomiale di grado 5 con rumore gaussiano:

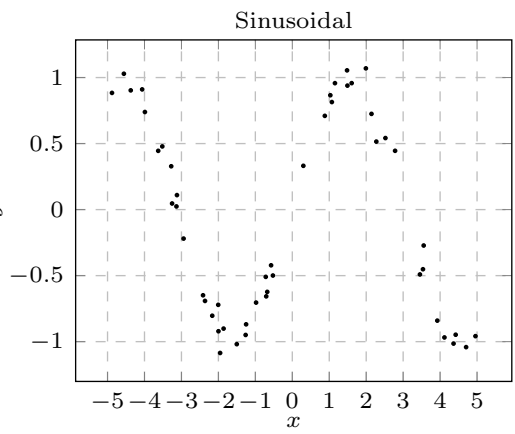
$$y = 0.0125x^5 - 0.125x^3 + 0.25x^2 - 0.5x + 1 + \epsilon$$

Una funzione sinusoidale con rumore gaussiano:

$$y = \sin(x) + \epsilon$$

I due dataset





Approccio base

Approccio base

- Definiamo una serie di reti neurali di dimensione crescente, e addestriamo ciascuna di esse sui due dataset.
- Per ciascuna rete, valutiamo la bontà dell'approssimazione attraverso il mean squared error (MSE):

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

- Useremo mezzo dataset per l'addestramento e mezzo per la validazione

Classe base

```
import torch
import torch.nn as nn

class Net(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        pass

    def fit(self, x, y, lr=0.01, epochs=300, show=False):
        losses = []
        loss_fn = nn.MSELoss()
        optimizer = torch.optim.SGD(self.parameters(), lr=lr)
        for epoch in range(epochs):
            loss = loss_fn(self.forward(x).squeeze(), y)
            optimizer.zero_grad()
            loss.backward()
            losses.append(loss.item())
            optimizer.step()
            if show :
                utils.plot(x, y, self, f"{epoch}, {loss.item():.2f}", pause=False)
        return losses
```

Il learning rate

Il learning rate

- Il learning rate è un iperparamentro che tipicamente varia nel range $[10^{-6}, 10^{-1}]$, con 0.01 come valore molto comune.
- Un learning rate troppo elevato può far sì che la rete “salti” da un minimo locale all’altro senza convergere.
- Un learning rate troppo piccolo rallenta eccessivamente l’addestramento, tipicamente si preferisce iniziare con il massimo learning rate possibile che non faccia divergere l’addestramento.
- Nelle reti moderne è comune utilizzare strategie per ridurre il learning rate durante l’addestramento.

Le epoche di addestramento

- Il tuning del numero di epoche è sostanzialmente gratuito: si lascia addestrare la rete per un numero elevato di epoche e si ferma quando la funzione di loss smette di migliorare.
- Lo stop può essere automatico (utilizzando un dataset di validazione separato da quello di addestramento per evitare l'overfitting) o manuale, vedendo a partire da che epoca la loss smette di migliorare, e tagliando l'addestramento a partire da quella epoca.
- Cosa costituisce un numero di epoche "elevato" dipende dal problema e dalla dimensione del dataset, potrebbero essere sufficienti poche decine/centinaia di epoche o potrebbero essere necessarie qualche migliaio di epoche.

Reti larghe

Reti larghe

- Un elevato numero di neuroni permette alla rete di apprendere numerose features
- In caso di parallelizzazione su GPU, il minor numero di operazioni sequenziali permette di sfruttare meglio le capacità di calcolo della scheda grafica
- La backpropagation è più semplice
- Possibile problema: overfitting

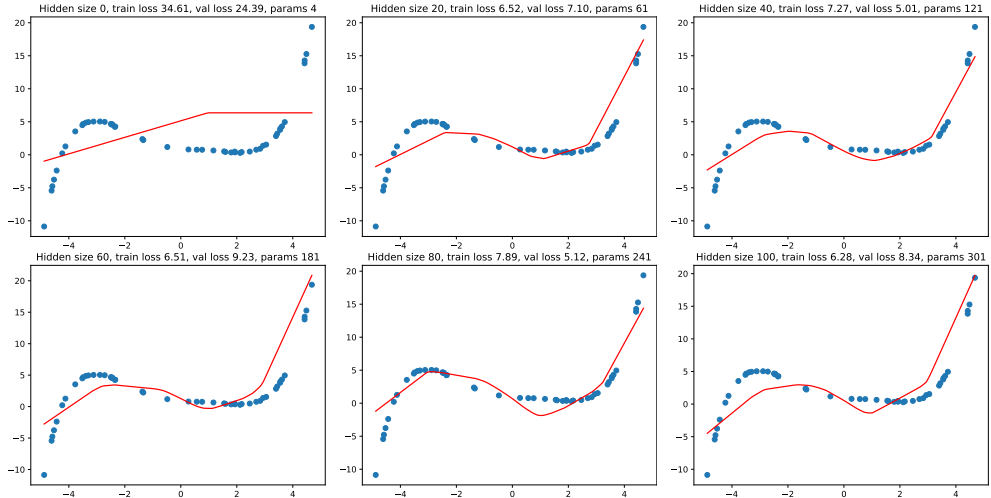
Reti larghe

Primo tentativo: un unico layer nascosto di dimensione variabile.

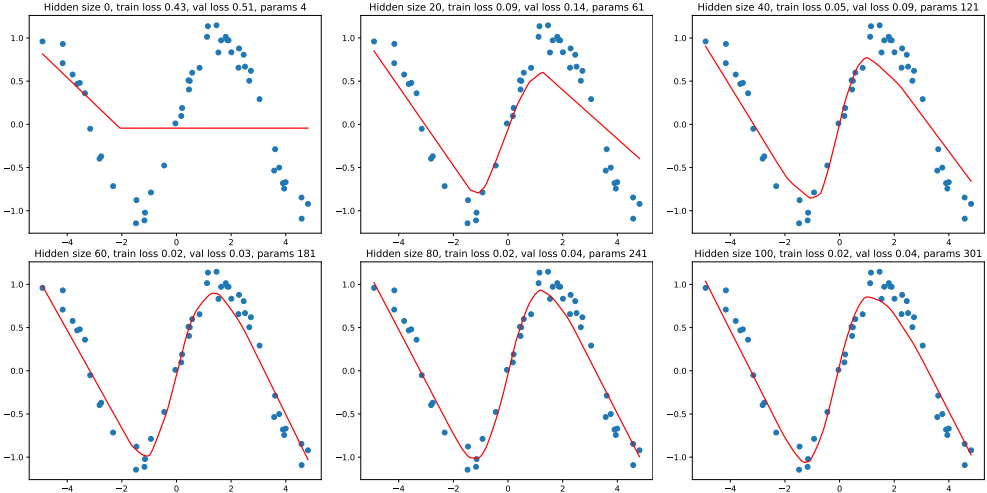
```
class WideNet(Net):
    def __init__(self, hidden_size):
        super().__init__()
        hidden_size = max(1, hidden_size)
        self.fc1 = nn.Linear(1, hidden_size)
        self.fc2 = nn.Linear(hidden_size, 1)
        self.activation = nn.ReLU()

    def forward(self, x):
        x = self.activation(self.fc1(x))
        x = self.fc2(x)
        return x
```

Reti larghe — Polinomio



Reti larghe — Sinusoide



Reti profonde

Reti profonde

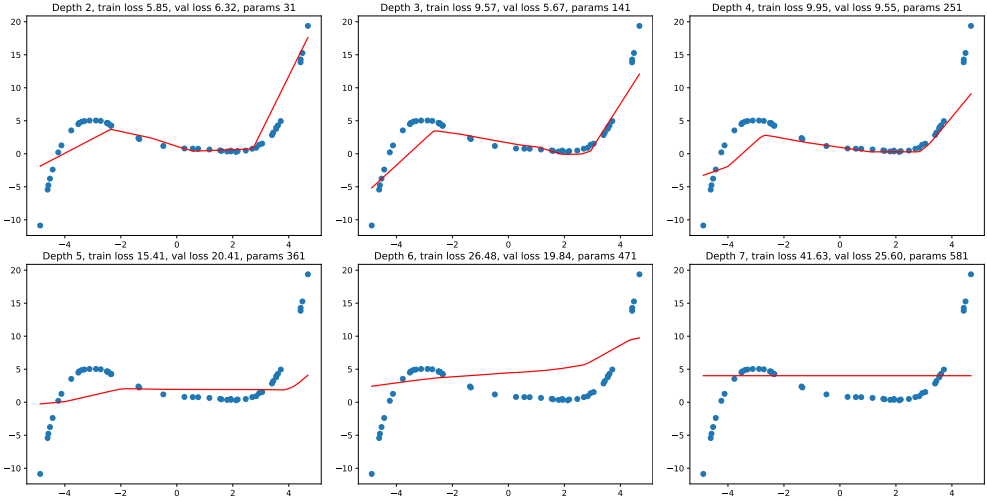
- Una rete molto profonda può ottenere feature di più alto livello e “ragionare” ad un livello più astratto
- Questo tipo di rete può soffrire con più facilità del problema dell’esplosione/scomparsa del gradiente

Rete più profonda

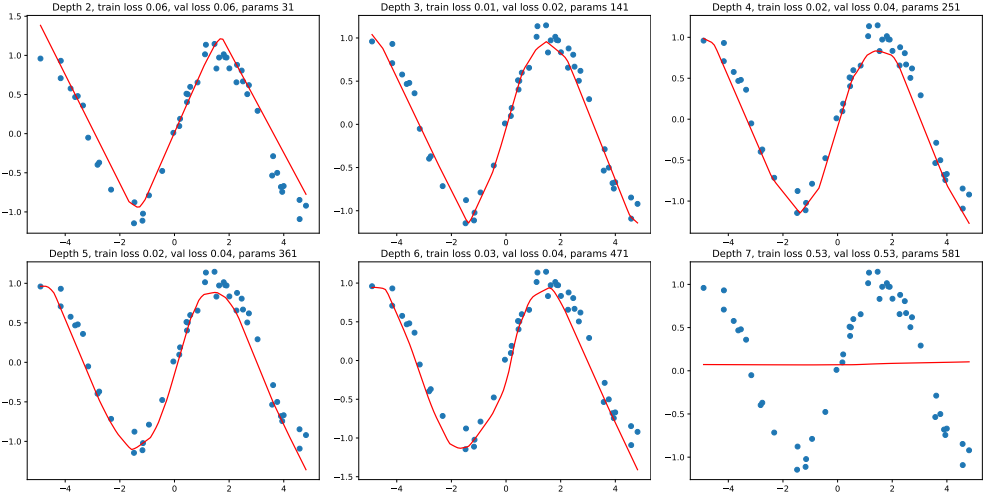
```
class DeepNet(Net):
    def __init__(self, depth, hidden_size):
        super().__init__()
        hidden_size = max(1, hidden_size)
        depth = max(1, depth)
        self.activation = nn.ReLU()
        layers = [nn.Linear(1, hidden_size), self.activation]
        for _ in range(depth-2):
            layers += [nn.Linear(hidden_size, hidden_size),
                self.activation]
        self.layers = nn.Sequential(*layers)
        self.output = nn.Linear(hidden_size, 1)

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

Reti profonde — Polinomio



Reti profonde — Sinusoide



Nota bene

- Questa distinzione tra reti larghe e reti profonde non è così netta nella pratica.
- Una rete può essere molto larga nei primi layer e successivamente può avere numerosi livelli con meno neuroni.
- Non esiste una regola generale per la scelta della dimensione e del numero dei layer.

Dimensionare la rete

Dimensionare la rete

Non esiste una regola generale per la scelta della dimensione e del numero dei layer. Ci sono però alcune linee guida empiriche che si possono tenere a mente nel lavoro “artigianale” di progettazione di una rete:

- Spesso le reti con il primo strato nascosto più largo dell’input funzionano meglio di reti “undercomplete”.
- Avere tutti i layer centrali con lo stesso numero di neuroni da spesso risultati equivalenti, se non migliori, di reti con un numero di neuroni crescente/decrescente.
- “Sbagliare” usando strati troppo larghi tipicamente ha meno effetti negativi che “sbagliare” usando strati troppo stretti (oltre al costo di addestramento maggiore).

Esempio pratico

Esempio pratico

Dato il codice nel file `main2.py`, scrivere il codice mancante per creare la rete e addestrarla sul dataset `dataset3.dat`. Determinare una architettura adeguata per la rete e individuare i parametri migliori per l'addestramento.
Si tenga presente che la rete deve risolvere un problema di classificazione su tre classi con due feature in input.

Obbiettivo: ottenere un'accuratezza sul dataset di test maggiore del 94.5%.