Andrea Augello
Department of Engineering, University of Palermo, Italy

Approssimatori universali e scelta degli iperparametri

Università degli Studi di Palermo

Il problema della regressione

Il problema della regressione

- La regressione è 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

Il problema della regressione

- 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:

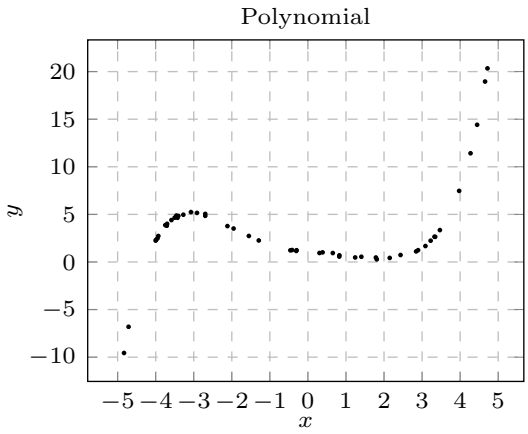
Una funzione polinomiale di grado 5 con rumore gaussiano:

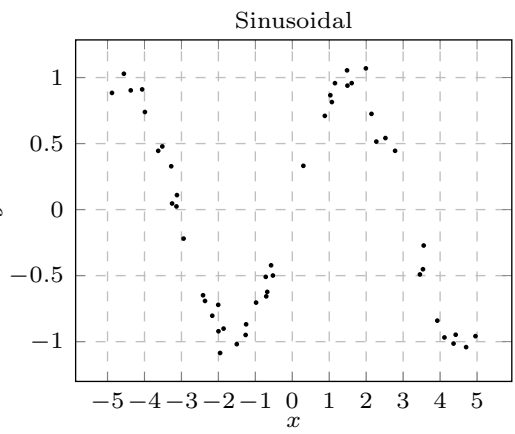
$$y = 0.0125x^5 - 0.125x^3 + 0.25x^2 - 0.5x + 1 + \epsilon$$

Una funzione sinusoidale con rumore gaussiano:

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

I due dataset





Le solite funzioni di utilità

```
def load_data(file):
    data = np.loadtxt(file)
    x = data[:, :-1]
    y = data[:, -1]
    return torch.tensor(x, dtype=torch.float32),\
            torch.tensor(y, dtype=torch.float32)
```

```
def plot(X,y,model, title="", pause=False):
    plt.clf()
    xmin, xmax = min(X), max(X)
    x = np.linspace(xmin, xmax, 100)
    plt.scatter(X, y)
    plt.plot(x, model.forward(torch.tensor(x,
        dtype=torch.float32)).detach().numpy(), color="red")
    plt.title(title)
    plt.pause(0.25)
    if pause:
        plt.show()
```

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

```python
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, decay=True):
        losses = []
        loss_fn = nn.MSELoss()
        optimizer = torch.optim.SGD(self.parameters(), lr=lr)
        if decay:
            scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.995)
        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)
            if decay:
                scheduler.step()
        return losses
```

Scheduler del learning rate

Scheduler del 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 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.

Perché ridurre il learning rate?

Ipotesi sul perché variare il learning rate durante l'addestramento sia utile[1]:

- Un learning rate troppo elevato può far sì che la rete "salti" da un minimo locale all'altro senza convergere.
- Un learning rate più basso consente alla rete di convergere in un minimo più "profondo" ma "stretto".
- Un learning rate iniziale elevato impedisce alla rete di apprendere il rumore nei dati, la riduzione graduale permette di imparare pattern più complessi.

[1] Per approfondimenti: https://arxiv.org/pdf/1908.01878

Perché ridurre il learning rate?

Ipotesi sul perché variare il learning rate durante l'addestramento sia utile[1]:

- Un learning rate troppo elevato può far sì che la rete "salti" da un minimo locale all'altro senza convergere.
- Un learning rate più basso consente alla rete di convergere in un minimo più "profondo" ma "stretto".
- Un learning rate iniziale elevato impedisce alla rete di apprendere il rumore nei dati, la riduzione graduale permette di imparare pattern più complessi.

Strategie popolari per lo scheduling del learning rate:

- step
- cyclic
- exponential

[1] Per approfondimenti: https://arxiv.org/pdf/1908.01878

Scheduler esponenziale

- Ha meno parametri da settare/su cui ragionare rispetto ad altri scheduler.
- Ad ogni epoca, il learning rate viene moltiplicato per un fattore $\gamma \leq 1$.
- Se $\gamma = 1$, il learning rate rimane costante.
- Se avessimo più batch, andremmo a variare il learning rate solo dopo che tutti i batch sono stati processati.

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

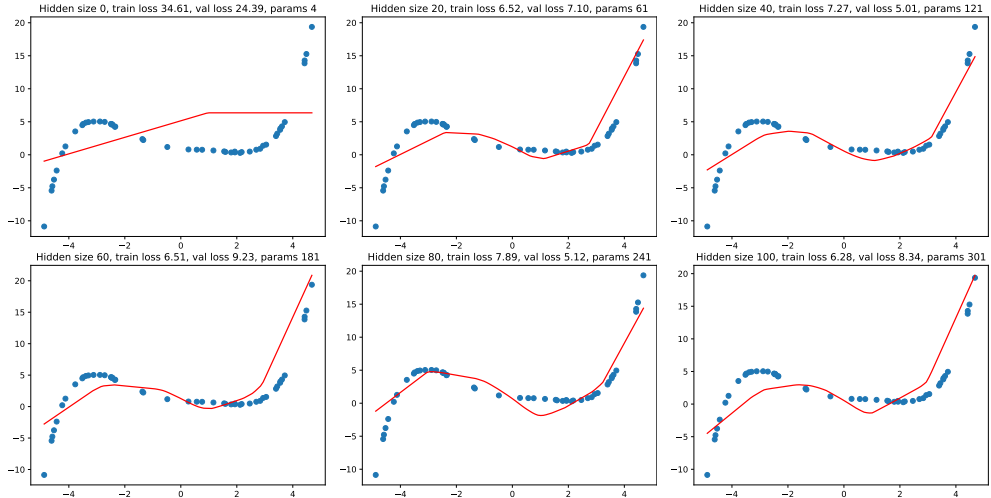
Reti larghe

Primo tentativo: un unico layer nascosto di dimensione variabile.

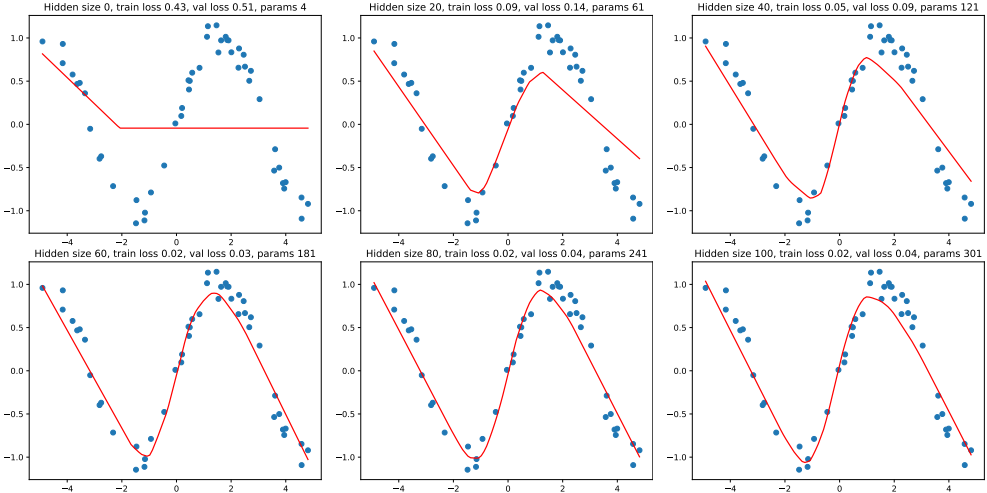
```
class WideNet(Net):
    def __init__(self, hidden_size):
        super().__init__()
        hidden_size = max(1, hidden_size)
        self.fc1 = nn.Linear(1, hidden_size)
        self.fc2 = nn.Linear(hidden_size, 1)
        self.activation = nn.ReLU()

    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

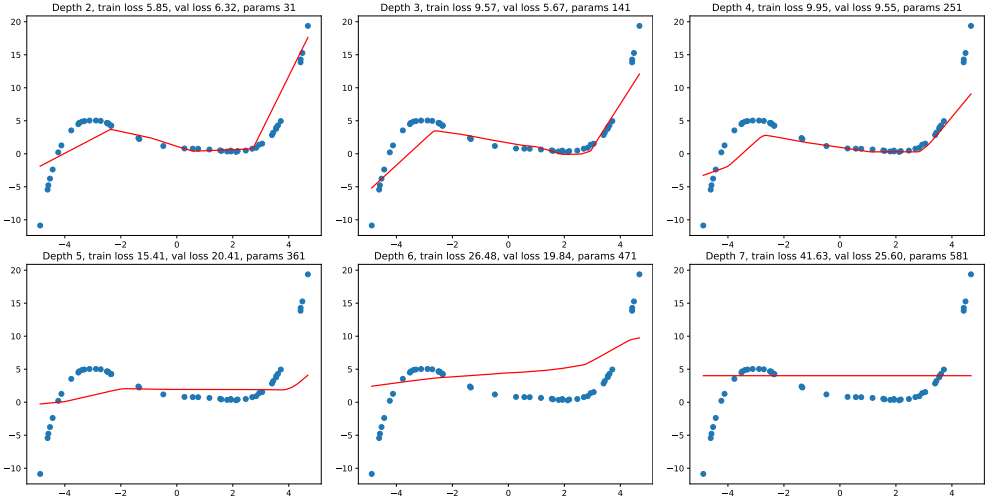
- Una rete molto profonda può ottenere feature di più alto livello e “ragionare” ad un livello più astratto
- Questo tipo di rete può soffrire con più facilità del problema dell’esplosione/scomparsa del gradiente

Rete più profonda

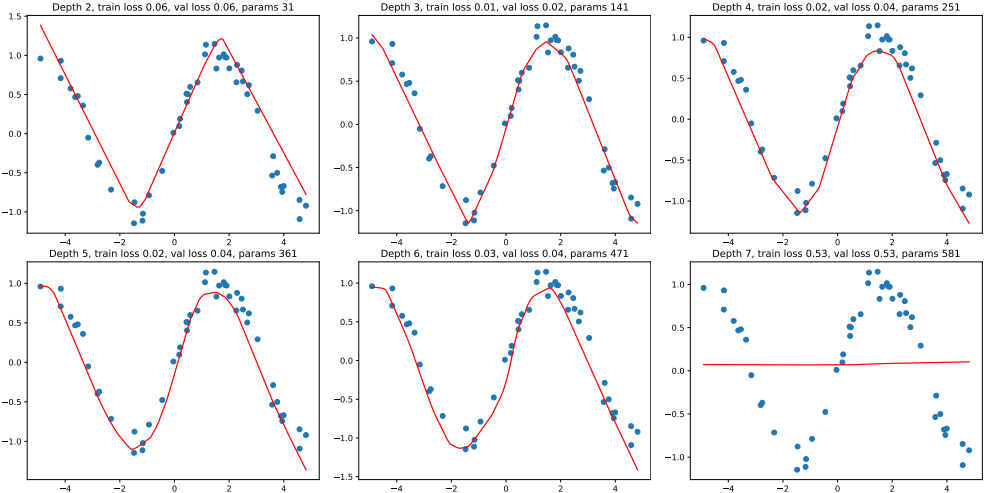
```
class DeepNet(Net):
    def __init__(self, depth, hidden_size):
        super().__init__()
        hidden_size = max(1, hidden_size)
        depth = max(1, depth)
        self.activation = nn.ReLU()
        layers = [nn.Linear(1, hidden_size), self.activation]
        for _ in range(depth-2):
  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