Andrea Augello
Department of Engineering, University of Palermo, Italy

Regressione, approssimatori universali, e reti neurali

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:

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

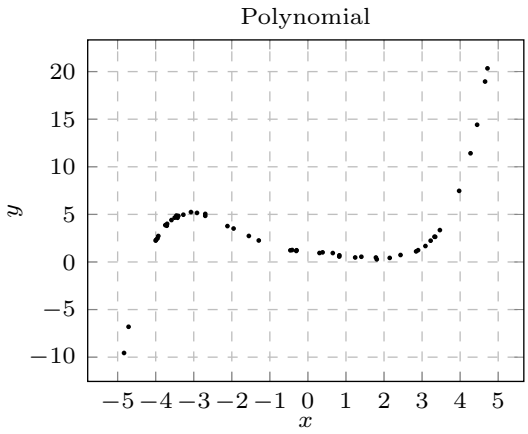
```
def dataset1():
    x = np.random.random(50)*10-5
    y = 0.0125 * x**5 - 0.125 *
        x**3  + 0.25* x**2 - 0.5*
        x + 1
    y += np.random.normal(0, 0.1,
        y.shape)
    for i in range(len(y)):
        print(x[i], y[i])
```

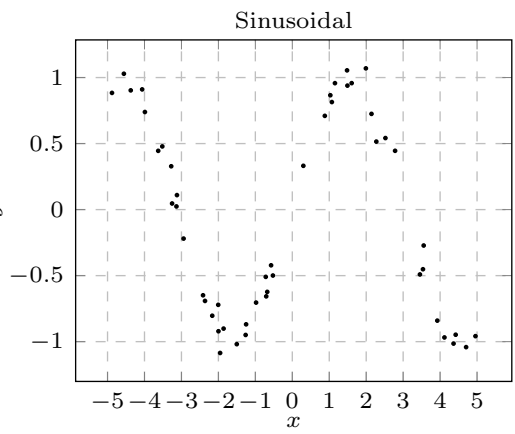
Una funzione sinusoidale con rumore gaussiano:

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

```
def dataset2():
    x = np.random.random(50)*10-5
    y = np.sin(x)
    y += np.random.normal(0, 0.1,
        y.shape)


    for i in range(len(y)):
        print(x[i], y[i])
```

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

```
import torch
import torch.nn as nn

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

    def forward(self, x):
        pass

    def train(self, x, y, lr=0.01, epochs=300, show=False):
        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()
            optimizer.step()
            if show and epoch % 20 == 0:
                utils.plot(x, y, self, f"Epoch {epoch}, loss
                    {loss.item():.2f}", pause=False)
        return loss.item()
```

Codice boilerplate

```
def use_net(dataset):
    x,y = utils.load_data(dataset)
    x_train, y_train = x[:len(x)//2], y[:len(y)//2]
    x_test, y_test = x[len(x)//2:], y[len(y)//2:]
    net = nets.YourNet()
    train_loss = net.train(x_train, y_train)
    val_loss = nn.MSELoss()(net(x_test).squeeze(), y_test)
    utils.plot(x_test, y_test, net, f"Train loss {train_loss:.2f},
        val loss {val_loss:.2f}, params {pytorch_total_params}")

if __name__ == "__main__":
    use_net(sys.argv[1])
```

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.

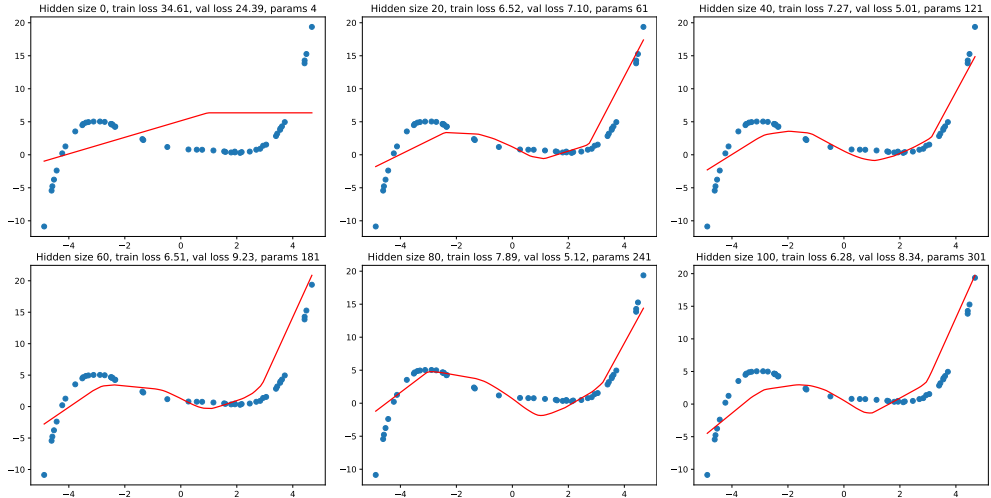
```
class WideNet(Net):
    def __init__(self, hidden_size):
        super().__init__()
        hidden_size = max(1, hidden_size)
        self.fc1 = nn.Linear(1, hidden_size)
        self.fc2 = nn.Linear(hidden_size, 1)
        self.activation = nn.ReLU()

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

Reti larghe

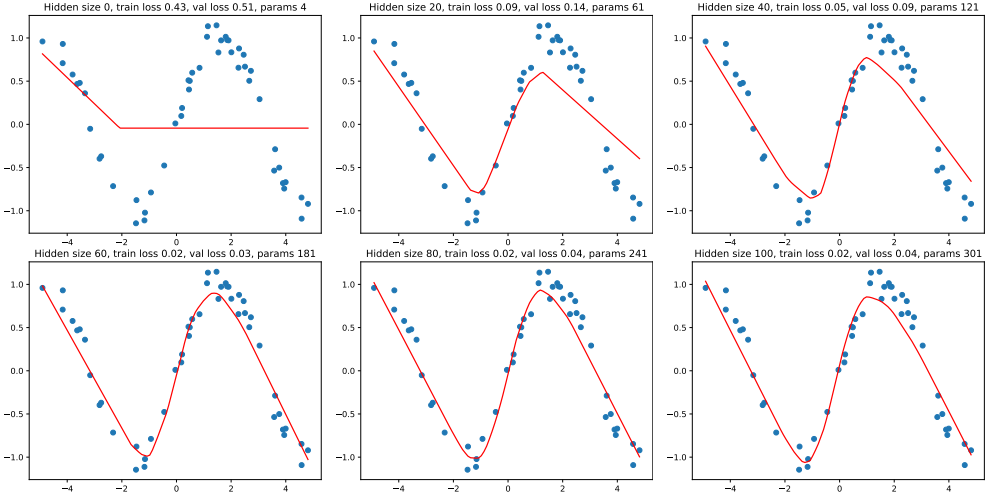
Proviamo diverse dimensioni del layer nascosto:

```
import nets, utils, sys
import torch.nn as nn

def widenet(dataset):
    x,y = utils.load_data(dataset)
    x_train, y_train = x[:len(x)//2], y[:len(y)//2]
    x_test, y_test = x[len(x)//2:], y[len(y)//2:]
    for i in range(0, 101, 20):
        net = nets.WideNet(i)
        pytorch_total_params = sum(p.numel() for p in net.parameters() if
            p.requires_grad)
        train_loss = net.train(x_train, y_train)
        val_loss = nn.MSELoss()(net(x_test).squeeze(), y_test)
        utils.plot(x_test, y_test, net, f"Hidden size {i}, train loss
            {train_loss:.2f}, val loss {val_loss:.2f}, params
            {pytorch_total_params}")

if __name__ == "__main__":
    widenet(sys.argv[1])
```

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

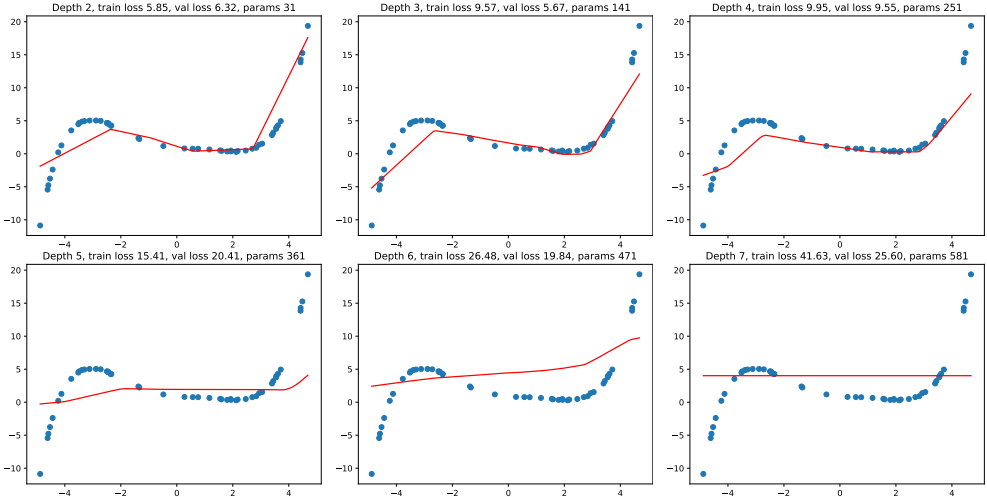
Rete più profonda

```
class DeepNet(Net):
    def __init__(self, depth, hidden_size):
        super().__init__()
        hidden_size = max(1, hidden_size)
        depth = max(1, depth)
        self.fc1 = nn.Linear(1, hidden_size)
        for i in range(2,depth):
            setattr(self, f'fc{i}', nn.Linear(hidden_size,
                hidden_size))
        setattr(self, f'fc{depth}', nn.Linear(hidden_size, 1))

    def forward(self, x):
        x = F.relu(self.fc1(x))
        for i in range(2, len(self._modules)):
            x = F.relu(getattr(self, f'fc{i}')(x))
        x = getattr(self, f'fc{len(self._modules)}')(x)
        return x
```

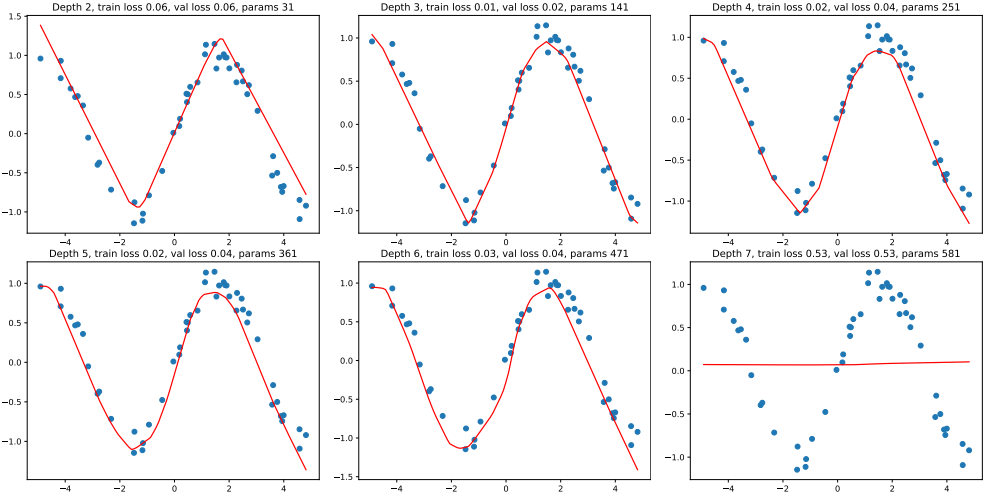
Reti profonde

```
def deepnet(dataset):
    x,y = utils.load_data(dataset)
    x_train, y_train = x[:len(x)//2], y[:len(y)//2]
    x_test, y_test = x[len(x)//2:], y[len(y)//2:]
    for i in range(2, 8, 1):
        net = nets.DeepNet(i,10)
        pytorch_total_params = sum(p.numel() for p in
            net.parameters() if p.requires_grad)
        train_loss = net.train(x_train, y_train, lr=0.01)
        val_loss = nn.MSELoss()(net(x_test).squeeze(), y_test)
        utils.plot(x_test, y_test, net, f"Depth {i}, train loss
            {train_loss:.2f}, val loss {val_loss:.2f}, params
            {pytorch_total_params}")
```

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.

Feature engineering

Feature engineering

Il feature engineering è un processo di trasformazione dei dati di input in modo da renderli più adatti all'apprendimento automatico.

- In molti casi è reso superfluo dall'uso di reti neurali sufficientemente profonde
- Può comunque ridurre la complessità della rete e velocizzare l'addestramento

Feature di Taylor

La serie di Taylor

Una funzione continua $f(x)$ può essere approssimata con una serie di Taylor:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

Vale per x vicino ad a. Includendo abbastanza termini, è possibile ottenere una buona approssimazione un dominio più ampio.

Feature di Taylor

L'idea di base è che la funzione f che vogliamo approssimare può essere approssimata con una serie di Taylor centrata in un punto a:

- Daremo in input alla rete i valori di $(x)^n$ per n crescente
- La rete (idealmente) apprenderà i coefficienti della serie di Taylor
- Il punto a è un iperparametro della rete
- Visti gli elevamenti a potenza, è necessario scalare l'input in modo opportuno

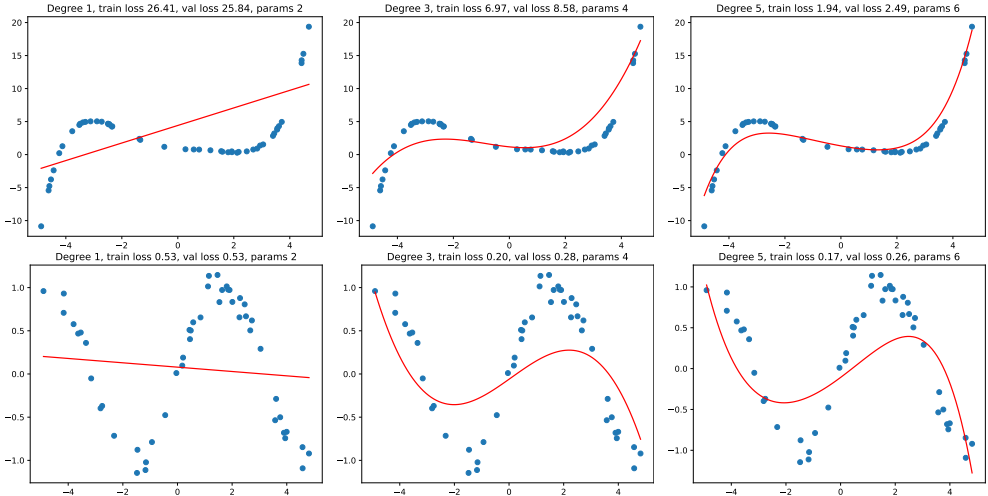
Feature di Taylor

```
class TaylorNet(Net):
    def __init__(self, degree):
        super().__init__()
        self.degree = degree
        self.fc1 = nn.Linear(degree, 1)

    def forward(self, x):
        x = torch.cat([(x**i-self.mean[i-1])/self.std[i-1] \
                for i in range(1, self.degree+1)], dim=1)
        return self.fc1(x)

    def train(self, x, y, lr=0.01, epochs=300):
        self.set_scaler(x)
        return super().train(x, y, lr, epochs)

    def set_scaler(self, x):
        self.mean = []
        self.std = []
        for i in range(1, self.degree+1):
            self.mean.append(torch.mean(x**i))
            self.std.append(torch.std(x**i))
```

Feature di Taylor

```
def taylornet(dataset):
    x,y = utils.load_data(dataset)
    x_train, y_train = x[:len(x)//2], y[:len(y)//2]
    x_test, y_test = x[len(x)//2:], y[len(y)//2:]
    for i in range(1, 7, 2):
        net = nets.TaylorNet(i)
        pytorch_total_params = sum(p.numel() for p in
            net.parameters() if p.requires_grad)
        train_loss = net.train(x_train, y_train, lr=0.01)
        val_loss = nn.MSELoss()(net(x_test).squeeze(), y_test)
        utils.plot(x_test, y_test, net, f"Degree {i}, train loss
            {train_loss:.2f}, val loss {val_loss:.2f}, params
            {pytorch_total_params}")
```

Risultato


Degree 1, train loss 26.41, val loss 25.84, params 2
Degree 3, train loss 6.97, val loss 8.58, params 4
Degree 5, train loss 1.94, val loss 2.49, params 6
Degree 1, train loss 0.53, val loss 0.53, params 2
Degree 3, train loss 0.20, val loss 0.28, params 4
Degree 5, train loss 0.17, val loss 0.26, params 6

Feature di Taylor

Se l'input è di dimensione maggiore di 1, è possibile utilizzare le feature di Taylor per approssimare funzioni di più variabili.

In questo caso bisogna ricordarsi che vanno considerate tutte le possibili combinazioni di termini, ovvero:

$$f(x_1, \ldots, x_n) = \sum_{i_1=0}^{d} \cdots \sum_{i_n=0}^{d} c_{i_1,\ldots,i_n} (x_1 - a_1)^{i_1} \ldots (x_n - a_n)^{i_n}$$

Feature di Fourier

Feature di Fourier

- Per funzioni periodiche, è possibile utilizzare le feature di Fourier
- Le feature di Fourier sono ottime per approssimare funzioni periodiche, ma possono anche approssimare funzioni non periodiche avendo cura di scalare l'input in modo opportuno

La serie di Fourier

Una funzione periodica $f(x)$ con periodo 2π può essere approssimata con una serie di Fourier:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(nx) + b_n \sin(nx)$$

Valori elevati di n corrispondono a frequenze più alte, spesso è possibile ottenere una buona approssimazione con un numero limitato di termini.

Feature di Fourier

Preso in input un valore x, la rete calcola i valori di $\sin(nx)$ e $\cos(nx)$ per n crescente fino ad un limite dato, e li concatena in un unico vettore.

- La rete apprende i coefficienti a_n e b_n della serie di Fourier ed il bias $a_0/2$.
- Il numero di termini della serie di Fourier è un iperparametro della rete.
- A differenza della serie di Taylor, l'approssimazione non è centrata in un punto, ma è valida per tutto il dominio.
- Se la funzione è periodica, bisogna assicurarsi che il periodo venga correttamente scalato.
- Se la funzione NON è periodica, bisogna scalare l'input per essere sicuri che il dominio non ecceda $[-\pi, \pi]$.

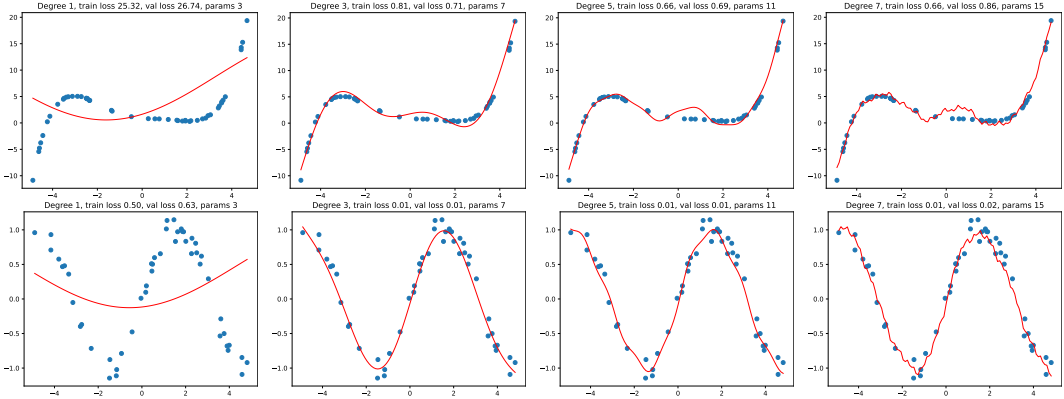
Feature di Fourier

```
class FourierNet(Net):
    def __init__(self, degree):
        super().__init__()
        self.degree = degree
        self.fc1 = nn.Linear(2*degree, 1)

    def forward(self, x):
        x = (x-self.bias)*self.scale
        x = torch.cat([torch.sin(x*i)) for i in range(0, self.degree)] +\
                [torch.cos(x*i)) for i in range(0, self.degree)], dim=1)
        return self.fc1(x)

    def set_scaler(self, x):
        #make the wole domain [0, pi]
        min_x, max_x = torch.min(x), torch.max(x)
        self.bias = -min_x
        self.scale = torch.pi/(max_x-min_x)

    def train(self, x, y, lr=0.1, epochs=300):
        self.set_scaler(x)
        return super().train(x, y, lr, epochs)
```

Solito codice

```
def fouriernet(dataset):
    x,y = utils.load_data(dataset)
    x_train, y_train = x[:len(x)//2], y[:len(y)//2]
    x_test, y_test = x[len(x)//2:], y[len(y)//2:]
    for i in range(1, 10, 2):
        net = nets.FourierNet(i)
        pytorch_total_params = sum(p.numel() for p in
            net.parameters() if p.requires_grad)
        train_loss = net.train(x_train, y_train, lr=0.1)
        val_loss = nn.MSELoss()(net(x_test).squeeze(), y_test)
        utils.plot(x_test, y_test, net, f"Degree {i}, train loss
            {train_loss:.2f}, val loss {val_loss:.2f}, params
            {pytorch_total_params}")
```

Risultati


Degree 1, train loss 25.32, val loss 26.74, params 3
Degree 3, train loss 0.81, val loss 0.71, params 7
Degree 5, train loss 0.66, val loss 0.69, params 11
Degree 7, train loss 0.66, val loss 0.86, params 15
Degree 1, train loss 0.50, val loss 0.63, params 3
Degree 3, train loss 0.01, val loss 0.01, params 7
Degree 5, train loss 0.01, val loss 0.01, params 11
Degree 7, train loss 0.01, val loss 0.02, params 15

Limitazione

Il numero di parametri tende ad esplodere con la dimensione dell'input. Già con un input di dimensione 2, una serie di Fourier di grado d è

$$
\begin{aligned}
f(x,y) = &\sum_{n=0}^{d}\sum_{m=0}^{d} a_{n,m}\sin(nx)\cos(my) \\
&+ \sum_{n=0}^{d}\sum_{m=0}^{d} b_{n,m}\cos(nx)\sin(my) \\
&+ \sum_{n=0}^{d}\sum_{m=0}^{d} c_{n,m}\sin(nx)\sin(my) \\
&+ \sum_{n=0}^{d}\sum_{m=0}^{d} d_{n,m}\cos(nx)\cos(my)
\end{aligned}
$$

Aumentando la dimensione dell'input, il numero di termini cresce esponenzialmente.

Feature di Fourier

Per mitigare il problema occorrono tecniche di riduzione della dimensionalità…

Feature di Fourier

Per mitigare il problema occorrono tecniche di riduzione della dimensionalità…

…di cui non ci occuperemo!

Esempio più realistico

Esempio più realistico

Vogliamo una rete che impari la legge di Coulomb:

$$F = \frac{1}{4\pi\epsilon_0}\frac{q_1 q_2}{r^2}$$

```
def coulomb():
    epsilon_0 = 8.8541878128e-12
    for _ in range(20000):
        q1 = 3*(np.random.random()-0.5)*1e-7
        q2 = 3*(np.random.random()-0.5)*1e-8
        r = np.random.uniform(0.01, 0.005)
        F = 1/(4*np.pi*epsilon_0) * q1*q2/r**2
        print(q1, q2, r, F)
```

Definizione della rete

```
class CoulombNet(Net):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(3, 20)
        self.fc2 = nn.Linear(20, 20)
        self.fc3 = nn.Linear(20, 10)
        self.fc4 = nn.Linear(10, 10)
        self.fc5 = nn.Linear(10, 1)
        self.activation = nn.ReLU()

    def set_scaler(self, x,y):
        self.mean = [ torch.mean(x[:,i]) for i in range(x.shape[1])]
        self.std  = [ torch.std(x[:,i])  for i in range(x.shape[1])]
        self.miny , self.maxy = torch.min(y), torch.max(y)

    def train(self, x, y, lr=0.01, epochs=300):
        self.set_scaler(x,y)
        return super().train(x, y, lr, epochs)
```

Definizione della rete

Scaliamo anche l'output così che la rete lavori con valori compresi tra 0 e 1:

```
class CoulombNet(Net):
    #...

    def forward(self, x):
        x = torch.stack([(x[:,i]-self.mean[i])/self.std[i] for i in
            range(x.shape[1])], dim=1)
        x = self.activation(self.fc1(x))
        x = self.activation(self.fc2(x))
        x = self.activation(self.fc3(x))
        x = self.activation(self.fc4(x))
        x = nn.Sigmoid()(self.fc5(x))
        x = x*(self.maxy-self.miny)+self.miny
        return x
```

La rete in azione

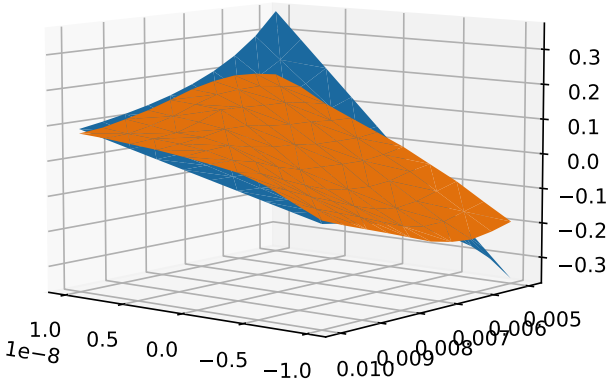
```
def coulombnet():
    x,y = utils.load_data("coulomb.dat")
    x_train, y_train = x[:3*len(x)//4], y[:3*len(y)//4]
    x_test, y_test = x[3*len(x)//4:], y[3*len(y)//4:]
    net = nets.CoulombNet()
    pytorch_total_params = sum(p.numel() for p in net.parameters() if
        p.requires_grad)
    train_loss = net.train(x_train, y_train, lr=0.1, epochs=2000)
    val_loss = nn.MSELoss()(net(x_test).squeeze(), y_test)
    print(f"Train loss {train_loss:.3f}, val loss {val_loss:.3f},
        params {pytorch_total_params}")
    #...
```

La rete in azione

```
for q1 in np.linspace(-1e-7, 1e-7, 10):
    q2 = np.linspace(-1e-8, 1e-8, 10)
    r  = np.linspace(0.01, 0.005, 10)
    x = np.array([[q1, q2_s, r_s] for q2_s in q2 for r_s in r])
    y_hat = net(torch.tensor(x,
        dtype=torch.float32)).squeeze().detach().numpy()
    print(np.mean(y_hat), np.std(y_hat))
    #coulomb law ground truth
    epsilon_0 = 8.8541878128e-12
    y = np.array([1/(4*np.pi*epsilon_0) * q1*q2_s/r_s**2 for q1,
        q2_s, r_s in x])
    #3d mesh plot with q2, r, y
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.plot_trisurf(x[:,1], x[:,2], y,     label="ground truth")
    ax.plot_trisurf(x[:,1], x[:,2], y_hat, label="prediction")
    plt.show()
```

Risultato

Ai bordi del dominio, la qualità dell'approssimazione è scarsa, ma nel resto del dominio la rete approssima bene la legge di Coulomb.



Prossimi passi

Prossimi passi

- Provare ad affrontare la regressione con input di dimensione maggiore. Potete usare funzioni analitiche note per generare i dati di addestramento.
- Provare a risolvere un problema di classificazione con classi non linearmente separabili (Si consiglia attivazione sigmoide dopo l'ultimo layer). Potete usare funzioni analitiche note per generare i dati di addestramento:

$$\begin{cases} y = 1 & \text{se } x_2 = f(x_1) + \epsilon \\ y = 0 & \text{se } x_2 = f(x_1) - \epsilon \end{cases}$$

- Provare a combinare gli approcci di feature engineering con reti aventi layer nascosti.
  - Ordini relativamente bassi saranno probabilmente sufficienti
  - Potrebbe risultare superfluo considerare tutte le possibili combinazioni di termini