Andrea Augello
Department of Engineering, University of Palermo, Italy

Addestrare una rete neurale (SGD)

Università degli Studi di Palermo

Introduzione

Introduzione

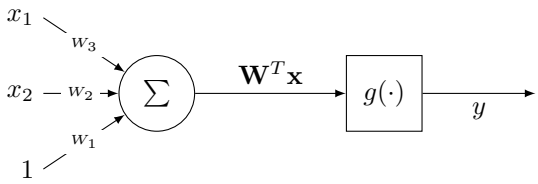
- Addestrare una rete neurale significa trovare i valori dei pesi che minimizzano una funzione di errore sul training set.
- Per trovare i pesi ottimali, è necessario utilizzare un algoritmo di ottimizzazione.
- L'algoritmo di ottimizzazione più semplice è la discesa del gradiente.
- La discesa del gradiente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

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.

Un esempio giocattolo

- Consideriamo una rete neurale con un solo neurone.
- La rete deve imparare a separare due classi linearmente separabili sul piano.
- La rete ha come funzione di attivazione la funzione identità.

$$g(x) = x$$
$$g'(x) = 1$$

Un esempio giocattolo

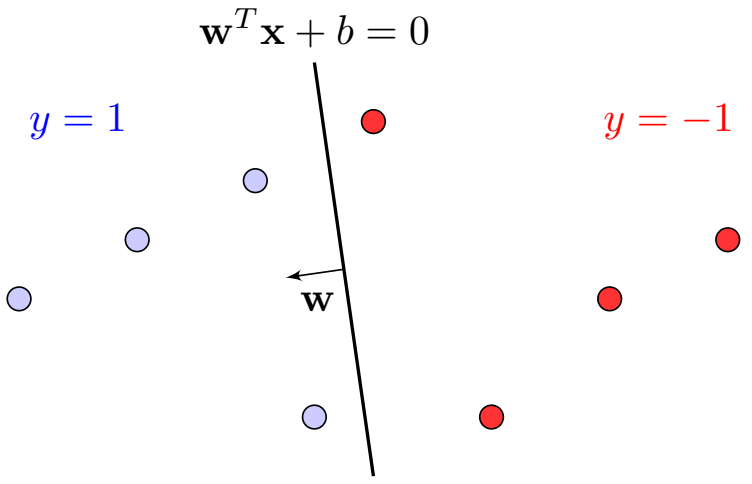
- Consideriamo una rete neurale con un solo neurone.
- La rete deve imparare a separare due classi linearmente separabili sul piano.
- La rete ha come funzione di attivazione la funzione identità.

$$g(x) = x$$
$$g'(x) = 1$$

Dataset:

x_1	x_2	y
2.0	1.0	1
6.0	0.5	-1
2.5	-1.0	1
5.0	0.0	-1
0.0	0.0	1
4.0	-1.0	-1
1.0	0.5	1
3.0	1.5	-1

Un esempio giocattolo



La predizione della rete è: $y = \text{sign}(\mathbf{W}^T\mathbf{x} + b)$

PyTorch

PyTorch

- PyTorch è un framework per il deep learning.
- PyTorch è basato su Tensor, un array multidimensionale.
- PyTorch fornisce un'implementazione efficiente di tensori e operazioni su tensori.
- 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.

Una semplice rete neurale in PyTorch

Proviamo a implementare la rete neurale giocattolo in PyTorch.

Una semplice rete neurale in PyTorch

Proviamo a implementare la rete neurale giocattolo in PyTorch.

```
import torch
import torch.nn as nn

class ToyNet(nn.Module):
    def __init__(self):
        super(ToyNet, self).__init__()
        self.fc1 = nn.Linear(2, 1)

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

`nn.Module` è la classe base per tutti i moduli di reti neurali in PyTorch.

Accedere ai parametri

```
>>> net = ToyNet()
>>> print(net)
ToyNet(
  (fc1): Linear(in_features=2, out_features=1, bias=True)
)

>>> for name, param in net.named_parameters():
...     print(name, param)
fc1.weight Parameter containing:
tensor([[-0.6990,  0.4320]], requires_grad=True)
fc1.bias Parameter containing:
tensor([-0.0255], requires_grad=True)
```

Modificare i parametri

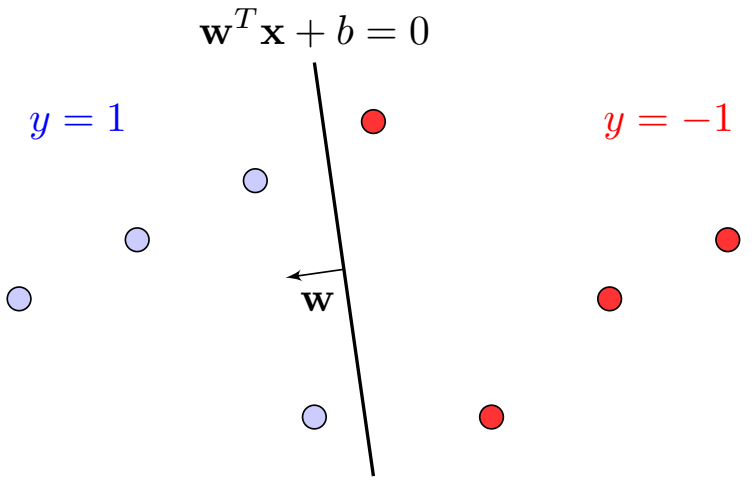
È possibile modificare i parametri della rete neurale accedendo direttamente ai tensori.

```
>>> print(net.fc1.weight)
tensor([[-0.6990,  0.4320]], requires_grad=True)
>>> net.fc1.weight.data = torch.nn.Parameter(torch.tensor([[1.0,
    0.0]]))
>>> print(net.fc1.weight)
tensor([[1.0,  0.0]], requires_grad=True)
```

Nella pratica, è molto scomodo e i parametri non vengono modificati manualmente.

Calcolare i pesi

Il caso studio



La predizione della rete è: $y = \text{sign}(\mathbf{W}^T\mathbf{x} + b)$

Caricare i dati e visualizzarli

```
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)
```

Caricare i dati e visualizzarli

```
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)
```

```
import matplotlib.pyplot as plt
def plot(x,y, net):
    plt.scatter(x[:, 0], x[:, 1], c=y)
    w = net.fc1.weight.data
    b = net.fc1.bias.data
    x1 = np.linspace(min(x[:, 0]), max(x[:, 0]), 100)
    # w_0*x + w_1*y + b = 0
    x2 = -(w[0, 0]*x1 + b[0])/w[0, 1]
    plt.plot(x1, x2)
    plt.ylim(min(x[:, 1])-1, max(x[:, 1])+1)
    plt.show()
```

Algoritmo del percettrone

Intuizione: correggere l’errore corrente.

Algoritmo del percettrone

Intuizione: correggere l'errore corrente.

- Errore su esempio positivo: $\mathbf{W}^T\mathbf{x} < 0$

$$\begin{aligned}\mathbf{W}_{t+1}^T x &= (\mathbf{W}_t + \mathbf{x}_t)^T \mathbf{x}_t \\ &= \mathbf{W}_t^T \mathbf{x}_t + \mathbf{x}_t^T \mathbf{x}_t \\ &= \mathbf{W}_t^T \mathbf{x}_t + ||\mathbf{x}_t||^2\end{aligned}$$

- Errore su esempio negativo: $\mathbf{W}^T\mathbf{x} > 0$

$$\begin{aligned}\mathbf{W}_{t+1}^T x &= (\mathbf{W}_t - \mathbf{x}_t)^T \mathbf{x}_t \\ &= \mathbf{W}_t^T \mathbf{x}_t - \mathbf{x}_t^T \mathbf{x}_t \\ &= \mathbf{W}_t^T \mathbf{x}_t - ||\mathbf{x}_t||^2\end{aligned}$$

Algoritmo del percettrone

Intuizione: correggere l'errore corrente.

- Errore su esempio positivo: $\mathbf{W}^T\mathbf{x} < 0$

$$\begin{aligned}\mathbf{W}_{t+1}^T x &= (\mathbf{W}_t + \mathbf{x}_t)^T \mathbf{x}_t \\ &= \mathbf{W}_t^T \mathbf{x}_t + \mathbf{x}_t^T \mathbf{x}_t \\ &= \mathbf{W}_t^T \mathbf{x}_t + ||\mathbf{x}_t||^2\end{aligned}$$

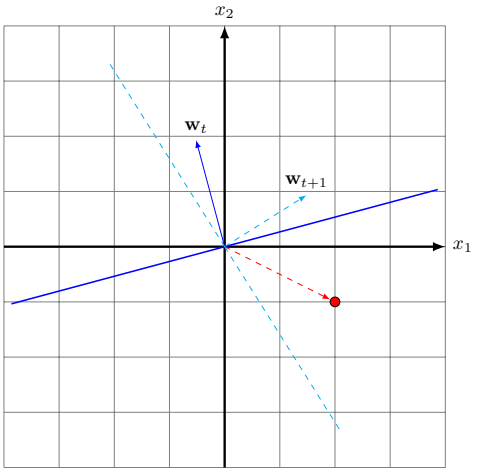
- Errore su esempio negativo: $\mathbf{W}^T\mathbf{x} > 0$

$$\begin{aligned}\mathbf{W}_{t+1}^T x &= (\mathbf{W}_t - \mathbf{x}_t)^T \mathbf{x}_t \\ &= \mathbf{W}_t^T \mathbf{x}_t - \mathbf{x}_t^T \mathbf{x}_t \\ &= \mathbf{W}_t^T \mathbf{x}_t - ||\mathbf{x}_t||^2\end{aligned}$$

1: $\mathbf{W_1} \leftarrow \mathbf{0}$
2: $t \leftarrow 1$
3: **while** non convergente **do**
4: $\quad \delta \leftarrow \frac{y - \text{sign}(\mathbf{W}_t^T \mathbf{x}_t)}{2}$
5: $\quad \mathbf{W}_{t+1} \leftarrow \mathbf{W}_t + \delta \mathbf{x}_t$
6: $\quad t \leftarrow t + 1$
7: **end while**

Interpretazione geometrica

Interpretazione geometrica: ruotare l'iperpiano di decisione fino a che i punti sono correttamente classificati.



Convergenza

SE i dati sono linearmente separabili **ALLORA** l'algoritmo converge in un numero finito di passi.

Altrimenti, l'algoritmo non convergerà.

Implementazione

Addestramento

Valutiamo l'errore su un campione ed aggiorniamo i pesi.

1: $\mathbf{W_1} \leftarrow \mathbf{0}$
2: $t \leftarrow 1$
3: **while** non convergente **do**
4: $\quad \delta \leftarrow \frac{y - \text{sign}(\mathbf{W}_t^T \mathbf{x}_t)}{2}$
5: $\quad \mathbf{W}_{t+1} \leftarrow \mathbf{W}_t + \delta \mathbf{x}_t$
6: $\quad t \leftarrow t + 1$
7: **end while**

Addestramento

Valutiamo l'errore su un campione ed aggiorniamo i pesi.

1: $\mathbf{W_1} \leftarrow \mathbf{0}$
2: $t \leftarrow 1$
3: **while** non convergente **do**
4: $\quad \delta \leftarrow \frac{y - \text{sign}(\mathbf{W}_t^T \mathbf{x}_t)}{2}$
5: $\quad \mathbf{W}_{t+1} \leftarrow \mathbf{W}_t + \delta \mathbf{x}_t$
6: $\quad t \leftarrow t + 1$
7: **end while**

```
class ToyNet(nn.Module):
    def __init__(self):
        super(ToyNet, self).__init__()
        self.fc1 = nn.Linear(2, 1)

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

    def train_sample_perceptron(self, x,y):
        y_hat = torch.sign(self.forward(x))
        delta = (y - y_hat)/2
        self.fc1.weight.data += delta * x
        self.fc1.bias.data += delta
```

Loop di addestramento

```
net = ToyNet.ToyNet()
x, y = load_data(DATA)
while True:
    # shuffle x and y
    indices = torch.randperm(len(x))
    x = x[indices]
    y = y[indices]

    for i in range(len(x)):
        net.train_sample_perceptron(x[i], y[i])

    plot(x, y, net)

    y_hat = torch.sign(net.forward(x))
    if torch.all(y_hat.flatten() == y):
        break
print("w: ", net.fc1.weight.data)
print("b: ", net.fc1.bias.data)
```

Discesa del gradiente

Discesa del gradiente

- L’algoritmo del percettrone non garantisce la convergenza in caso di dati non linearmente separabili.

Discesa del gradiente

- L'algoritmo del percettrone non garantisce la convergenza in caso di dati non linearmente separabili.
- L'apprendimento di una rete neurale è un problema di ottimizzazione.

Discesa del gradiente

- L’algoritmo del percettrone non garantisce la convergenza in caso di dati non linearmente separabili.
- L’apprendimento di una rete neurale è un problema di ottimizzazione.
- I parametri della rete neurale sono ottimizzati per minimizzare una funzione di errore sul training set.

Discesa del gradiente

- L'algoritmo del percettrone non garantisce la convergenza in caso di dati non linearmente separabili.
- L'apprendimento di una rete neurale è un problema di ottimizzazione.
- I parametri della rete neurale sono ottimizzati per minimizzare una funzione di errore sul training set.
- Tipiche funzioni di errore:

Discesa del gradiente

- L'algoritmo del percettrone non garantisce la convergenza in caso di dati non linearmente separabili.
- L'apprendimento di una rete neurale è un problema di ottimizzazione.
- I parametri della rete neurale sono ottimizzati per minimizzare una funzione di errore sul training set.
- Tipiche funzioni di errore:
  - Cross-entropy (problemi di classificazione): $E = -\sum_i y_i \log(\hat{y}_i)$

Discesa del gradiente

- L'algoritmo del percettrone non garantisce la convergenza in caso di dati non linearmente separabili.
- L'apprendimento di una rete neurale è un problema di ottimizzazione.
- I parametri della rete neurale sono ottimizzati per minimizzare una funzione di errore sul training set.
- Tipiche funzioni di errore:
  - Cross-entropy (problemi di classificazione): $E = -\sum_i y_i \log(\hat{y}_i)$
  - Mean squared error (problemi di regressione): $E = \frac{1}{2}(y - \hat{y})^2$

Discesa del gradiente

- L'algoritmo del percettrone non garantisce la convergenza in caso di dati non linearmente separabili.
- L'apprendimento di una rete neurale è un problema di ottimizzazione.
- I parametri della rete neurale sono ottimizzati per minimizzare una funzione di errore sul training set.
- Tipiche funzioni di errore:
  - Cross-entropy (problemi di classificazione): $E = -\sum_i y_i \log(\hat{y}_i)$
  - Mean squared error (problemi di regressione): $E = \frac{1}{2}(y - \hat{y})^2$
- Tipicamente si utilizza un sottoinsieme dei dati per l'addestramento (training set).

L'algoritmo SGD

1. Si inizializzano i pesi $\mathbf{W}$ in modo casuale.
2. Per ogni elemento $\mathbf{x}$ del training set:
   - 2.1 Si calcola l'output della rete $\hat{y} = g(\mathbf{W}^T\mathbf{x})$.
   - 2.2 Si calcola l'errore commesso attraverso la funzione di loss scelta.
   - 2.3 Si calcola il gradiente della loss rispetto ai pesi $\frac{\partial L}{\partial W_i}$.
   - 2.4 Si aggiornano i pesi nella direzione opposta al gradiente.
3. Si ripete il processo fino a che l'errore non è sufficientemente basso o il numero di epoche non è sufficientemente alto.

Calcolo del gradiente

- Usiamo come funzione di loss il MSE.

$$L = \frac{1}{2}Err^2 \equiv \frac{1}{2}(y - \hat{y})^2 = \frac{1}{2}(y - g(\sum_i W_i x_i))^2$$

Calcolo del gradiente

- Usiamo come funzione di loss il MSE.

$$L = \frac{1}{2}Err^2 \equiv \frac{1}{2}(y - \hat{y})^2 = \frac{1}{2}(y - g(\sum_i W_i x_i))^2$$

- L'ottimizzazione avviene attraverso la discesa del gradiente.

Calcolo del gradiente

- Usiamo come funzione di loss il MSE.

$$L = \frac{1}{2}Err^2 \equiv \frac{1}{2}(y - \hat{y})^2 = \frac{1}{2}(y - g(\sum_i W_i x_i))^2$$

- L'ottimizzazione avviene attraverso la discesa del gradiente.
- Deriviamo la loss rispetto all'output:

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y = Err$$

Calcolo del gradiente

- Usiamo come funzione di loss il MSE.

$$L = \frac{1}{2}Err^2 \equiv \frac{1}{2}(y - \hat{y})^2 = \frac{1}{2}(y - g(\sum_i W_i x_i))^2$$

- L'ottimizzazione avviene attraverso la discesa del gradiente.
- Deriviamo la loss rispetto all'output:

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y = Err$$

- Deriviamo l'output rispetto al peso W_i:

$$\frac{\partial \hat{y}}{\partial W_i} = \frac{\partial g(\sum_i W_i x_i)}{\partial W_i} = g'(\sum_i W_i x_i) \times x_i$$

Nel nostro caso, essendo $g(\cdot)$ l'identità, $g'(\cdot) = 1$, quindi $\frac{\partial \hat{y}}{\partial W_i} = x_i$

Calcolo del gradiente

- Usiamo come funzione di loss il MSE.

$$L = \frac{1}{2}Err^2 \equiv \frac{1}{2}(y - \hat{y})^2 = \frac{1}{2}(y - g(\sum_i W_i x_i))^2$$

- L'ottimizzazione avviene attraverso la discesa del gradiente.
- Deriviamo la loss rispetto all'output:

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y = Err$$

- Deriviamo l'output rispetto al peso W_i:

$$\frac{\partial \hat{y}}{\partial W_i} = \frac{\partial g(\sum_i W_i x_i)}{\partial W_i} = g'(\sum_i W_i x_i) \times x_i$$

  Nel nostro caso, essendo $g(\cdot)$ l'identità, $g'(\cdot) = 1$, quindi $\frac{\partial \hat{y}}{\partial W_i} = x_i$
- Regola di aggiornamento dei pesi:

$$W_i = W_i - \alpha \times (\hat{y} - y) \times x_i$$

Discesa del gradiente

```
class ToyNet(nn.Module):
    ...
    def manual_sdg_train(self, x, y, epochs=40, lr=0.05):
        # train using SGD optimizer and MSE loss
        for _ in range(epochs):
            for i in range(len(x)):
                y_hat = self.forward(x[i])
                error = y_hat - y[i]
                self.fc1.weight.data -= lr * error * x[i]
                self.fc1.bias.data -= lr * error * 1
```

La soluzione non artigianale

La soluzione non artigianale

```
class ToyNet(nn.Module):
    ...
    def train(self, x, y, epochs=40, lr=0.05):
        # train using SGD optimizer
        optimizer = torch.optim.SGD(self.parameters(), lr=lr)
        criterion = nn.MSELoss()
        for _ in range(epochs):
            optimizer.zero_grad()
            y_hat = self.forward(x).flatten()
            loss = criterion(y_hat, y)
            loss.backward()
            optimizer.step()
```

Avvertimenti

- Abbiamo utilizzato un esempio giocattolo per introdurre i concetti.

Avvertimenti

- Abbiamo utilizzato un esempio giocattolo per introdurre i concetti.
- In pratica, non si utilizza l'algoritmo come lo abbiamo implementato noi ma varianti più efficienti.

Avvertimenti

- Abbiamo utilizzato un esempio giocattolo per introdurre i concetti.
- In pratica, non si utilizza l'algoritmo come lo abbiamo implementato noi ma varianti più efficienti.
  - Non si calcola l'errore su un singolo esempio ma su un batch di esempi (Da qui discesa stocastica).

Avvertimenti

- Abbiamo utilizzato un esempio giocattolo per introdurre i concetti.
- In pratica, non si utilizza l'algoritmo come lo abbiamo implementato noi ma varianti più efficienti.
  - Non si calcola l'errore su un singolo esempio ma su un batch di esempi (Da qui discesa stocastica).
  - Si utilizzano varianti dell'algoritmo del gradiente (momentum, Nesterov, Adam, …).

Avvertimenti

- Abbiamo utilizzato un esempio giocattolo per introdurre i concetti.
- In pratica, non si utilizza l'algoritmo come lo abbiamo implementato noi ma varianti più efficienti.
  - Non si calcola l'errore su un singolo esempio ma su un batch di esempi (Da qui discesa stocastica).
  - Si utilizzano varianti dell'algoritmo del gradiente (momentum, Nesterov, Adam, …).
- Sarebbe stata più appropriata una funzione di attivazione come la sigmoide.

Avvertimenti

- Abbiamo utilizzato un esempio giocattolo per introdurre i concetti.
- In pratica, non si utilizza l'algoritmo come lo abbiamo implementato noi ma varianti più efficienti.
  - Non si calcola l'errore su un singolo esempio ma su un batch di esempi (Da qui discesa stocastica).
  - Si utilizzano varianti dell'algoritmo del gradiente (momentum, Nesterov, Adam, …).
- Sarebbe stata più appropriata una funzione di attivazione come la sigmoide.
- Per un problema di classificazione, non si utilizza la funzione di errore MSE ma la cross-entropy.

Avvertimenti

- Abbiamo utilizzato un esempio giocattolo per introdurre i concetti.
- In pratica, non si utilizza l'algoritmo come lo abbiamo implementato noi ma varianti più efficienti.
  - Non si calcola l'errore su un singolo esempio ma su un batch di esempi (Da qui discesa stocastica).
  - Si utilizzano varianti dell'algoritmo del gradiente (momentum, Nesterov, Adam, …).
- Sarebbe stata più appropriata una funzione di attivazione come la sigmoide.
- Per un problema di classificazione, non si utilizza la funzione di errore MSE ma la cross-entropy.
- La scelta degli iperparametri (numero di epoche, learning rate, …) è molto importante.

Prossimi passi

Prossimi passi

- Modificare il dataset in modo che non sia più linearmente separabile.
  - Come si comporta l'algoritmo del percettrone?
  - Come si comporta la discesa del gradiente?
- Consultando la documentazione online di PyTorch, provare a:
  - Aggiungere una sigmoide come funzione di attivazione.
  - Utilizzare la cross-entropy come funzione di errore.
    - Soltanto la soluzione built-in di PyTorch funzionerà correttamente, modificare le altre è al di là dello scopo di questo esercizio.
    - Come cambia l'iperpiano di decisione?
  - Provare ad aggiungere un layer nascosto.
  - Provare a modificare gli iperparametri (numero di epoche, learning rate, …).

Prossimi passi

Per plottare classificatori non lineari/con più layer, usare la seguente funzione:

```
def plot2(X,y, model, title=""):
    # define bounds of the domain
    min1, max1 = X[:, 0].min()-1, X[:, 0].max()+1
    min2, max2 = X[:, 1].min()-1, X[:, 1].max()+1
    # define the x and y scale
    x1grid , x2grid = np.arange(min1, max1, 0.025), np.arange(min2, max2, 0.025)
    # create all of the lines and rows of the grid
    xx, yy = np.meshgrid(x1grid, x2grid)
    # flatten each grid to a vector
    r1, r2 = xx.flatten(), yy.flatten()
    # horizontal stack vectors to create x1,x2 input for the model
    grid = [[x1, x2] for x1, x2 in zip(r1,r2)]
    # make predictions for the grid
    yhat = model.forward(torch.tensor(grid, dtype=torch.float32)).detach().numpy()
    # reshape the predictions back into a grid
    zz = yhat.reshape(xx.shape)
    # plot the grid of x, y and z values as a surface
    plt.contourf(xx, yy, zz, cmap='viridis')
    # create scatter plot for samples from each class
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis', edgecolors='black')
    plt.title(title)
    # show the plot
    plt.pause(2)
    plt.clf()
```