Classificazione (binary classification)

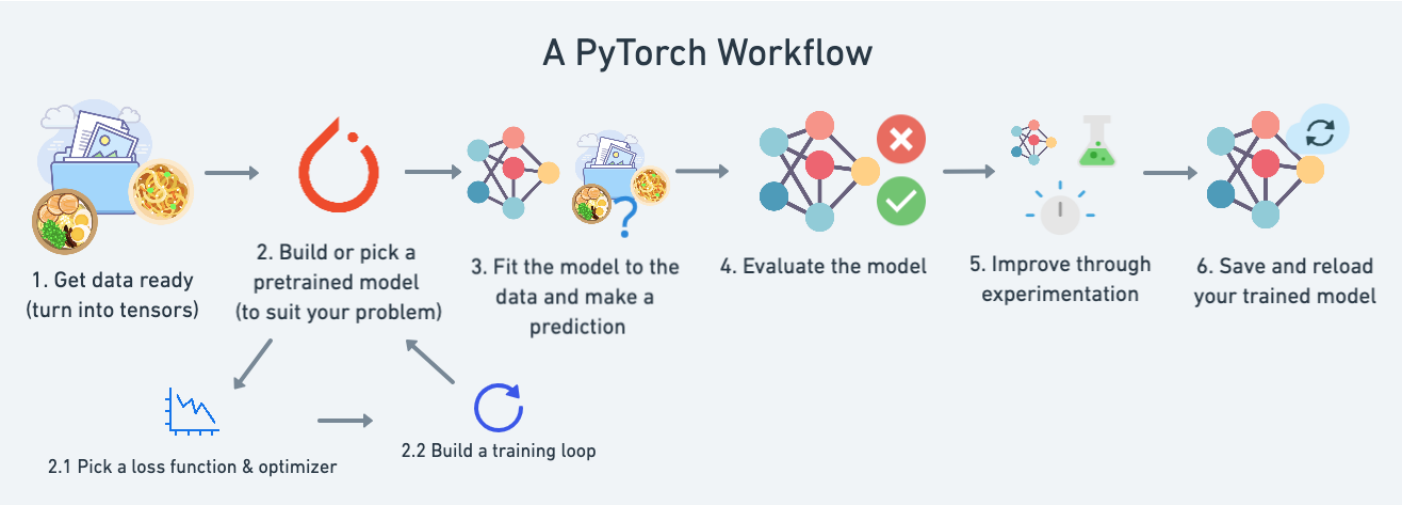
Introduzione

In questa lezione andremo a vedere la classificazione in base a delle tipolgie di dati, differisce quindi dalla regressione che si basa sulla predizione di un valore numero.

La classificazione può essere "binaria" es. cats vs dogs, oppure multiclass classification se abbiamo più di due tipologie da classificare.

Di seguito alcuni esempi di classificazione:

Cosa andreamo a trattare nel coso:



Topic	**Contents**
0. Architecture of a classification neural network	Neural networks can come in almost any shape or size, but they typically follow a similar floor plan.
1. Getting binary classification data ready	Data can be almost anything but to get started we're going to create a simple binary classification dataset.
2. Building a PyTorch classification model	Here we'll create a model to learn patterns in the data, we'll also choose a **loss function**, **optimizer** and build a **training loop** specific to classification.
3. Fitting the model to data (training)	We've got data and a model, now let's let the model (try to) find patterns in the (**training**) data.
4. Making predictions and evaluating a model (inference)	Our model's found patterns in the data, let's compare its findings to the actual (**testing**) data.
5. Improving a model (from a model perspective)	We've trained an evaluated a model but it's not working, let's try a few things to improve it.
6. Non-linearity	So far our model has only had the ability to model straight lines, what about non-linear (non-straight) lines?
7. Replicating non-linear functions	We used **non-linear functions** to help model non-linear data, but what do these look like?
8. Putting it all together with multi-class classification	Let's put everything we've done so far for binary classification together with a multi-class classification problem.

Partiamo con un esempio di classificazione basato su due serie di cerchi che si annidano tra di loro. Utilizziamo sklearn per ottenere questo set di dati:

```
from sklearn.datasets import make_circles
```

Make 1000 samples

n_samples = 1000

```
X, y = make_circles(n_samples,
□□□□□noise=0.03, # a little bit of noise to the dots
                    random_state=42) # keep random state so we get the same values
```

Create circles

proviamo a vedere cosa contengono le X e le y.

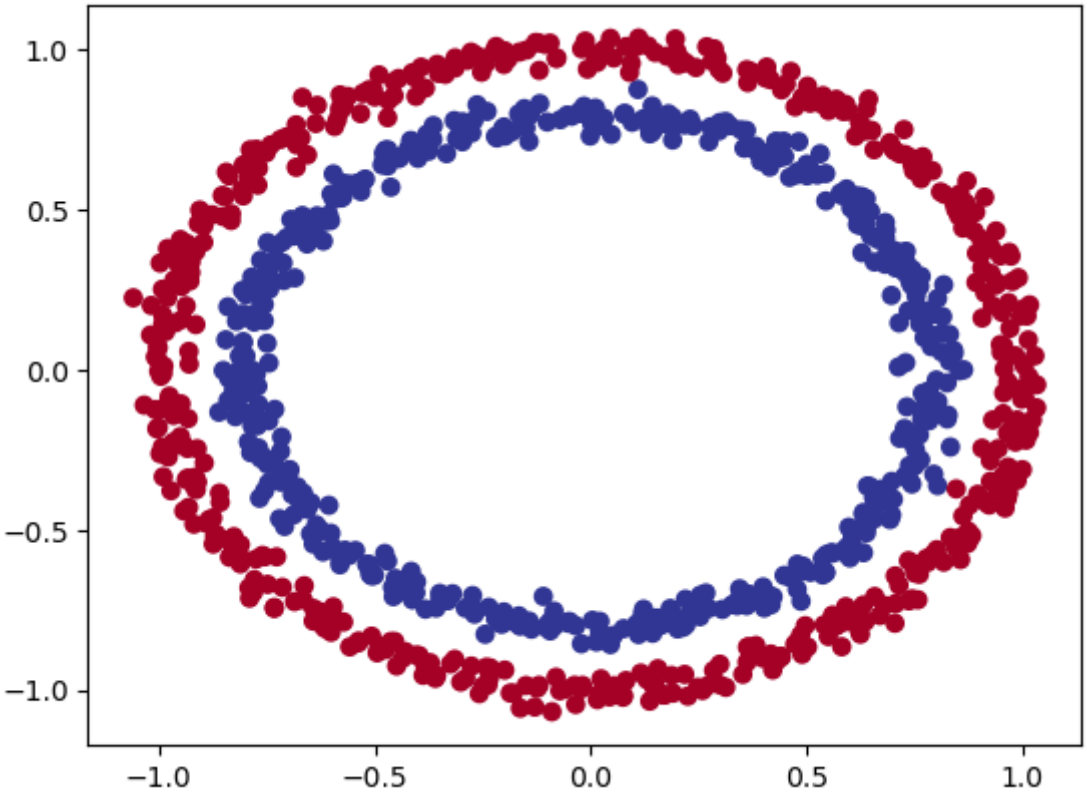
```
print(f"First 5 X features:\n{X[:5]}")
print(f"\nFirst 5 y labels:\n{y[:5]}")
```

First 5 X features: [[0.75424625 0.23148074] [-0.75615888 0.15325888] [-0.81539193 0.17328203] [-0.39373073 0.69288277] [0.44220765 -0.89672343]]

`First 5 y labels:[1 1 1 1 0]`

quindi le X contengono delle coordinate metre le y si suddividono in valori zero e uno. Quindi siamo di fronte ad una classificazione binaria, ma vediamola graficamente:

```
import matplotlib.pyplot as plt
plt.scatter(x=X[:, 0],
            y=X[:, 1],
            c=y,
            cmap=plt.cm.RdYlBu);
```



Quindi riassimento le X contengo le coordinate del cerchio, mentre le y il colore. Dalla figura si vede che i cerchi sono suffidivisi in due macrogruppi posizionati uno all'interno dell'altro.

Vediamo le shape:

```
# Check the shapes of our features and labels
X.shape, y.shape
```

```
((1000, 2), (1000,))
```

X ha una shape di due, mentre le y non ha uno shape in quanto è uno scalare di un valore.

Ora converiamo da numpy a tensori

```
# Turn data into tensors
# Otherwise this causes issues with computations later on
import torch
X = torch.from_numpy(X).type(torch.float)
y = torch.from_numpy(y).type(torch.float)
```

View the first five samples

print (X[:5], y[:5])

`(tensor([[ 0.7542,  0.2315],[-0.7562,  0.1533],[-0.8154,  0.1733],[-0.3937,  0.6929],[ 0.4422, -0.8967]]),tensor([1., 1., 1., 1., 0.]))`

lo converiamo in float32 (float) perchè numpy è in float64

splittiamo i dati in training e test

```
# Split data into train and test sets
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42) #
make the random split
```

La funziona "**train_test_split**" splitta le featurues e le label per noi. :)

Bene, ora costruiamo il modello:

```
# Standard PyTorch imports
import torch
from torch import nn

# Make device agnostic code
device = "cuda" if torch.cuda.is_available() else "cpu" device
```

Construct a model class that subclasses nn.Module

```
class CircleModelV0(nn.Module):
  def init(self):
    super().init()
    # 2. Create 2 nn.Linear layers capable of handling X and y input and output shapes
    self.layer_1 = nn.Linear(in_features=2, out_features=5)

        # takes in 2 features (X), produces 5 features
        self.layer_2 = nn.Linear(in_features=5, out_features=1) # takes in 5 features,
produces 1 feature (y)

# 3. Define a forward method containing the forward pass computation
def forward(self, x):
```

```
    # Return the output of layer_2, a single feature, the same shape as y
    return self.layer_2(self.layer_1(x)) # computation goes through layer_1 first then the output of layer_1 goes through layer_2
```

Create an instance of the model and send it to target device

`model_0 = CircleModelV0().to(device)model_0`

NB: una regola per settare il numero di feautres in **input** è fallo coincidere con le features del dataset. Idem per le features di **output**.

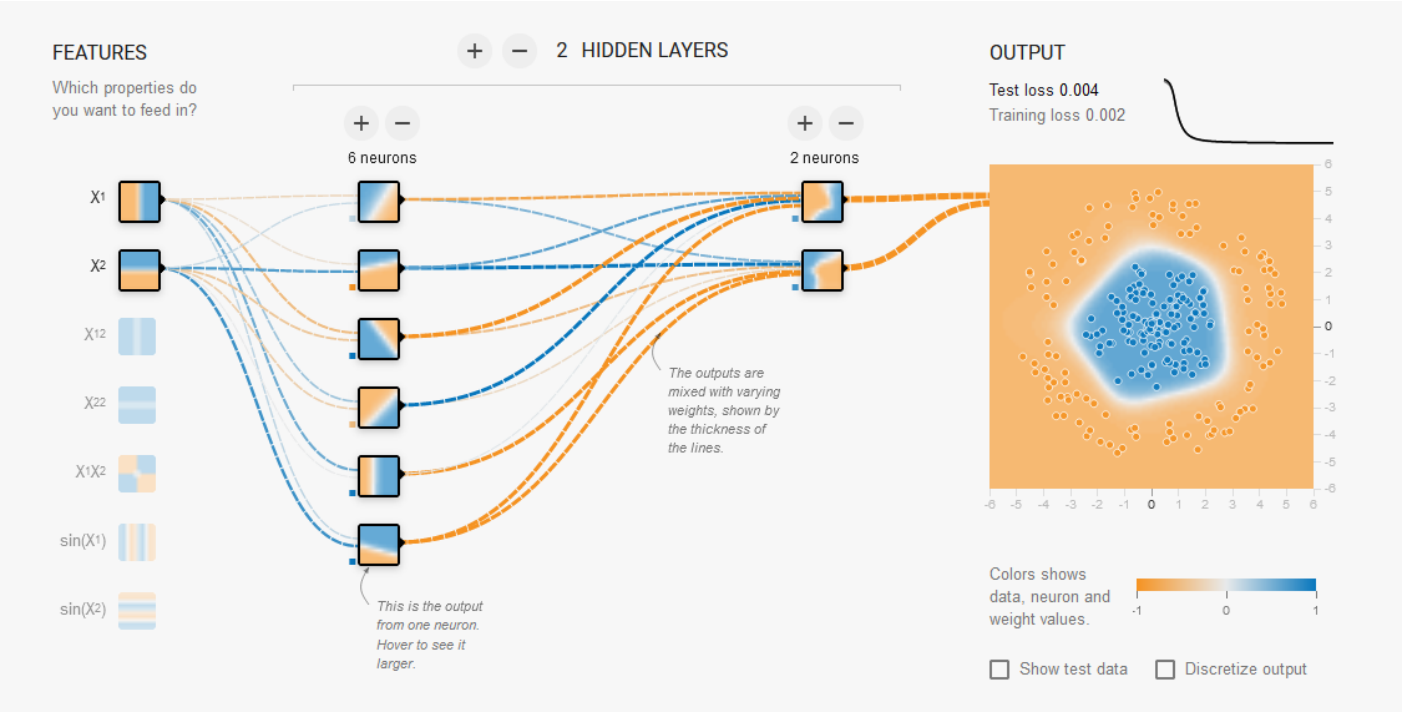
esiste inoltre un altro modo per rappresentare il modello in stile "Tensorflow", es:

```
# costruisco il modello
model_0 = nn.Sequential(
    nn.Linear(in_features=2, out_features=6),
    nn.Linear(in_features=6, out_features=2),
    nn.Linear(in_features=2, out_features=1)
).to(device)
```

`model_0`

Questo tipo di definizione del modello è "limitato" dal fatto che è sequenziale e quindi meno flessibile rispetto a reti più articolate.

Il modello può essere rappresentato graficamente come sotto riportato:



playground.tensorflow.org

ora, prima di fare il training del modello proviamo a passare i dati di test per vedere che output viene generato. (ovviamente essendo un modello non "allenato" saranno dati casuali)

```
# Make predictions with the model
with torch.inference_mode():
untrained_preds = model_0(X_test.to(device))
print(f"Length of predictions: {len(untrained_preds)}, Shape: {untrained_preds.shape}")
print(f"Length of test samples: {len(y_test)}, Shape: {y_test.shape}")
print(f"\nFirst 10 predictions:\n{untrained_preds[:10]}")
print(f"\nFirst 10 test labels:\n{y_test[:10]}")
```

Length of predictions: 200, Shape: torch.Size([200, 1]) Length of test samples: 200, Shape: torch.Size([200])

First 10 predictions: tensor([[-0.7534], [-0.6841], [-0.7949], [-0.7423], [-0.5721], [-0.5315], [-0.5128], [-0.4765], [-0.8042], [-0.6770]], device='cuda:0', grad_fn=<SliceBackward0>)

`First 10 test labels:tensor([1., 0., 1., 0., 1., 1., 0., 0., 1., 0.])`

Possiamo notare che che l'output non è zero oppure uno come invce sono le labels... come mai? lo vedremo più avanti...

Prima di fare il training settiamo la "loss function" e "l'optimizer".

Setup loss function and optimizer

La domanda che ci si pone di sempre quale loss function e optimizer utilzzare?

Per la classfificazione in genere si utilizza la binary cross entropy, vedi tabella esempio sotto ripotata:

Loss function/Optimizer	Problem type	PyTorch Code
Stochastic Gradient Descent (SGD) optimizer	Classification, regression, many others.	torch.optim.SGD()(https://pytorch.org/docs/stable/generated/torch.optim.SGD.html)
Adam Optimizer	Classification, regression, many others.	torch.optim.Adam() `https://pytorch.org/docs/stable/generated/torch.optim.Adam.html)
Binary cross entropy loss	Binary classification	torch.nn.BCELossWithLogits(https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html) or [`torch.nn.BCELoss`](https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html)
Cross entropy loss	Mutli-class classification	[`torch.nn.CrossEntropyLoss`](https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html)
Mean absolute error (MAE) or L1 Loss	Regression	[`torch.nn.L1Loss`](https://pytorch.org/docs/stable/generated/torch.nn.L1Loss.html)
Mean squared error (MSE) or L2 Loss	Regression	[`torch.nn.MSELoss`](https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html#torch.nn.MSELoss)

> Riassumento la **loss function misura** quanto il modello si distanzia dai valori attesi.

> Mentre per gli **optimizer** servono per migliorare il modello che poi attraverso la loss funzion verrà valutato.

In genere si utilizza **SGD** o **Adam**..

Ok creiamo la loss e l'optimizer:

```
# Create a loss function
# loss_fn = nn.BCELoss() # BCELoss = no sigmoid built-in
loss_fn = nn.BCEWithLogitsLoss() # BCEWithLogitsLoss = sigmoid built-in
```

Create an optimizer

`optimizer = torch.optim.SGD(params=model_0.parameters(),  lr=0.1)`

Accuracy e Loss function

> Definiamo anche il concetto di "**accuracy**".

La loss functuon misura quanto le preduzioni si allontanano dai valori desierati, mentre la **Accuracy** indica la **percentuale** con la quale il modello fa delle previsioni corrette. La differenza è sottile, e in questo momento non mi è chiara, credo che l'accuracy dipenda dalla loss e che indichi con una percentuale quello che la loss esprime in valori numerici specifici per il modello. Ad ogni modo vengono utilizzate entramb le misure per verificare la buona qualità del modello.

Implementiamo la accuracy

```
# Calculate accuracy (a classification metric)
def accuracy_fn(y_true, y_pred):
    correct = torch.eq(y_true, y_pred).sum().item() # torch.eq() calculates where two tensors
are equal
    acc = (correct / len(y_pred)) * 100
    return acc
```

Logits

I logits rappresentano l'output "grezzo" del modello. I logits devono essere convertiti nella previsione probabilistica passandoli ad una "funzione di attivazione". (es. sigmoid per la "binari cross entropy", softmax per la multiclass classificazion) Per noi essere "discretizzati" (i valori probabilistici) mediante l'uso di funzioni come "round".

Vediamo quindi come rivedere la fase di training in funzione dei logits. NB per capire la rappresentazione dei logits vedi il commento nel training loop.

```
for epoch in range(epochs):
    ### Training
```

```
    # 0. imposto la modalità in Training (da fare ad ogni epoca)
    model_0.train()

    # 1. calcolo l'output con i parametri del modello, NB devo gare la "squeeze" percheè va ritdotta di una dimensione
    # quanto l'output del modello ne aggiunge una.
    # I logits sono i valori "grezzi" che, nella caso delle classificazioni BINARIE, NON possono essere comparati
    # con i valori discreti 0/1 delle t_test.
    # I logits quindive dobranno essere convertiti attraverso le funzioni come per la esempio la sigmoing, che
    # non fa altro che ricondurli a valori compresi tra zero e uno che, poi andranno "discretizzati" a 0/1 atttraverso
    # l'uso di funzioni di arrotondamento come per es. la round.
    y_logits = model_0(X_train).squeeze() #

    # pred. logits -> pred. probabilities -> labels 0/1
    y_pred = torch.round(torch.sigmoid(y_logits))

    # 2. calculate loss/accuracy
    # calcolo la loss, da nota che viene utilizzata come loss function la "BCEWithLogitsLoss" che vuole in input
    # dirattamente i logits anzichè i valori predetti, in quanto gli applica la sigmoid e la round in automatico
    # per poi paragonli con le y_train "discrete".
    loss = loss_fn(y_logits, y_train) # nn.BCEWithLogitsLoss()

    # calcololiamo anche la percentuale di accuratezza.
    acc = accuracy_fn(y_true=y_train, y_pred=y_pred)

    # 3. reinizializzo l'optimizer in quanto tende ad accumulare i valori
    optimizer.zero_grad()

    # 4. effettua la back propagation, nella pratica Pytorch tiene traccia dei valori associati alla discesa del gradiente
    #    Quindi calcola la derivata parziale per determinare il minimo della curva dei delta tra valori predetti e valori di test
    loss.backward()

    # 5. ottimizza i parametri (una sola volta) e in base al valore "lr".
```

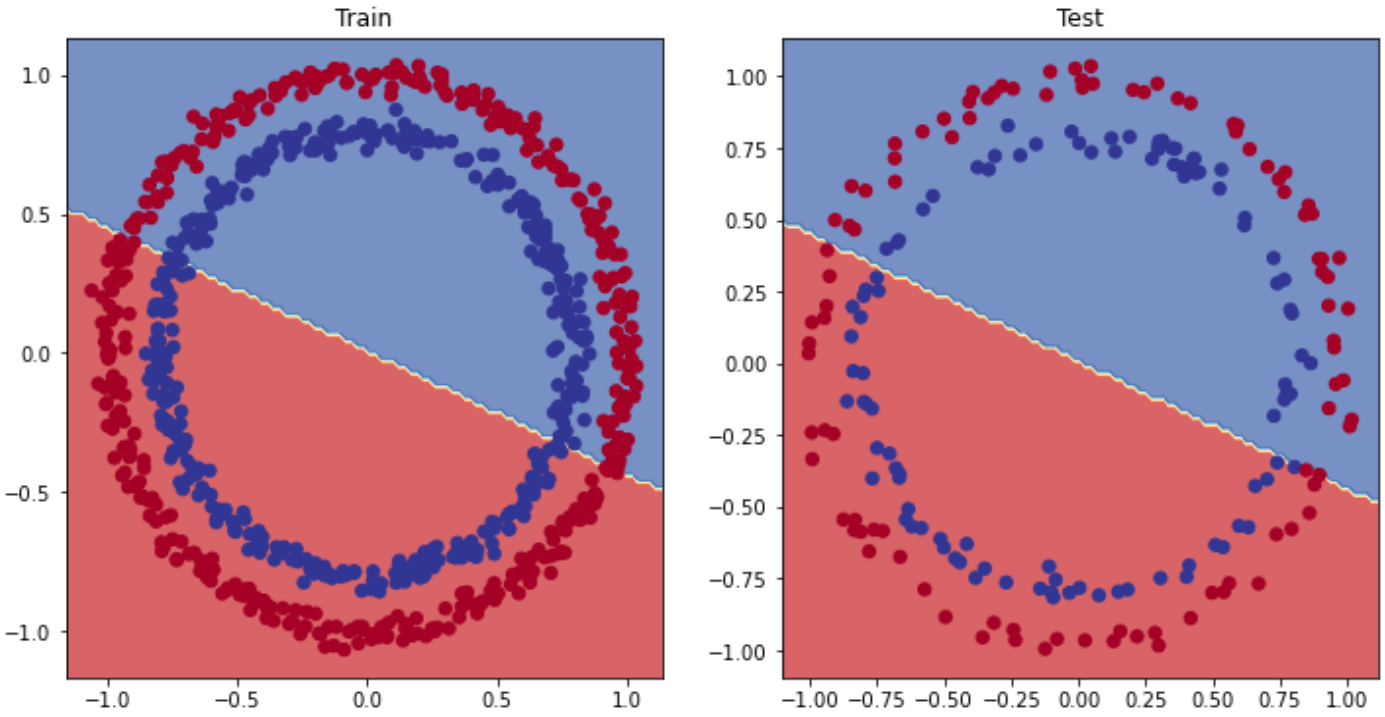
```
    #  NB: cambia quindi i valori dei tensori per cercare di farli avvicinare ai valori
ottimali
    optimizer.step()

    ### Testing (in questa fase vengono passati i valori non trainati di test)

    # indico a Pytrch che la fase di training è terminata e che ora devo valutare i parametri
e paragonarli con i valori attesi
    model_0.eval()
    with torch.inference_mode(): # disabilito la fase di training

        test_logits = model_0(X_test).squeeze()  #

        # pred. logits -> pred. probabilities -> labels 0/1
        test_pred = torch.round(torch.sigmoid(test_logits))

        # per poi paragonli con le y_train "discrete".
        test_loss = loss_fn(test_logits, y_test)  # nn.BCEWithLogitsLoss()

        # calcololiamo anche la percentuale di accuratezza.
        test_acc = accuracy_fn(y_true=y_test, y_pred=test_pred)

        # Print out what's happening
        if epoch % 10 == 0:
            print(f"Epoch: {epoch} | Train -> Loss: {loss:.5f} , Acc: {acc:.2f}% | Test ->
Loss: {test_loss:.5f}%. Acc: {test_acc:.2f}% ")
```

L'output della funzione sarà:

```
Python 3.10.8 | packaged by conda-forge | (main, Nov 24 2022, 14:07:00) [MSC v.1916 64 bit
(AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.7.0 -- An enhanced Interactive Python. Type '?' for help.
PyDev console: using IPython 8.7.0
Python 3.10.8 | packaged by conda-forge | (main, Nov 24 2022, 14:07:00) [MSC v.1916 64 bit
(AMD64)] on win32
runfile('C:\\lavori\\formazione_py\\src\\formazione\\DanielBourkePytorch\\02_classification.py
```

```
', wdir='C:\\lavori\\formazione_py\\src\\formazione\\DanielBourkePytorch')
Epoch: 0 | Train -> Loss: 0.70155 , Acc: 50.00% | Test -> Loss: 0.70146%. Acc: 50.00%
Epoch: 10 | Train -> Loss: 0.69617 , Acc: 57.50% | Test -> Loss: 0.69654%. Acc: 55.50%
Epoch: 20 | Train -> Loss: 0.69453 , Acc: 51.75% | Test -> Loss: 0.69501%. Acc: 54.50%
Epoch: 30 | Train -> Loss: 0.69395 , Acc: 50.38% | Test -> Loss: 0.69448%. Acc: 53.50%
Epoch: 40 | Train -> Loss: 0.69370 , Acc: 49.50% | Test -> Loss: 0.69427%. Acc: 53.50%
Epoch: 50 | Train -> Loss: 0.69358 , Acc: 49.50% | Test -> Loss: 0.69417%. Acc: 53.00%
Epoch: 60 | Train -> Loss: 0.69349 , Acc: 49.88% | Test -> Loss: 0.69412%. Acc: 52.00%
Epoch: 70 | Train -> Loss: 0.69343 , Acc: 49.62% | Test -> Loss: 0.69409%. Acc: 51.50%
Epoch: 80 | Train -> Loss: 0.69337 , Acc: 49.25% | Test -> Loss: 0.69408%. Acc: 51.50%
Epoch: 90 | Train -> Loss: 0.69333 , Acc: 49.62% | Test -> Loss: 0.69407%. Acc: 51.50%
Backend MacOSX is interactive backend. Turning interactive mode on.
```

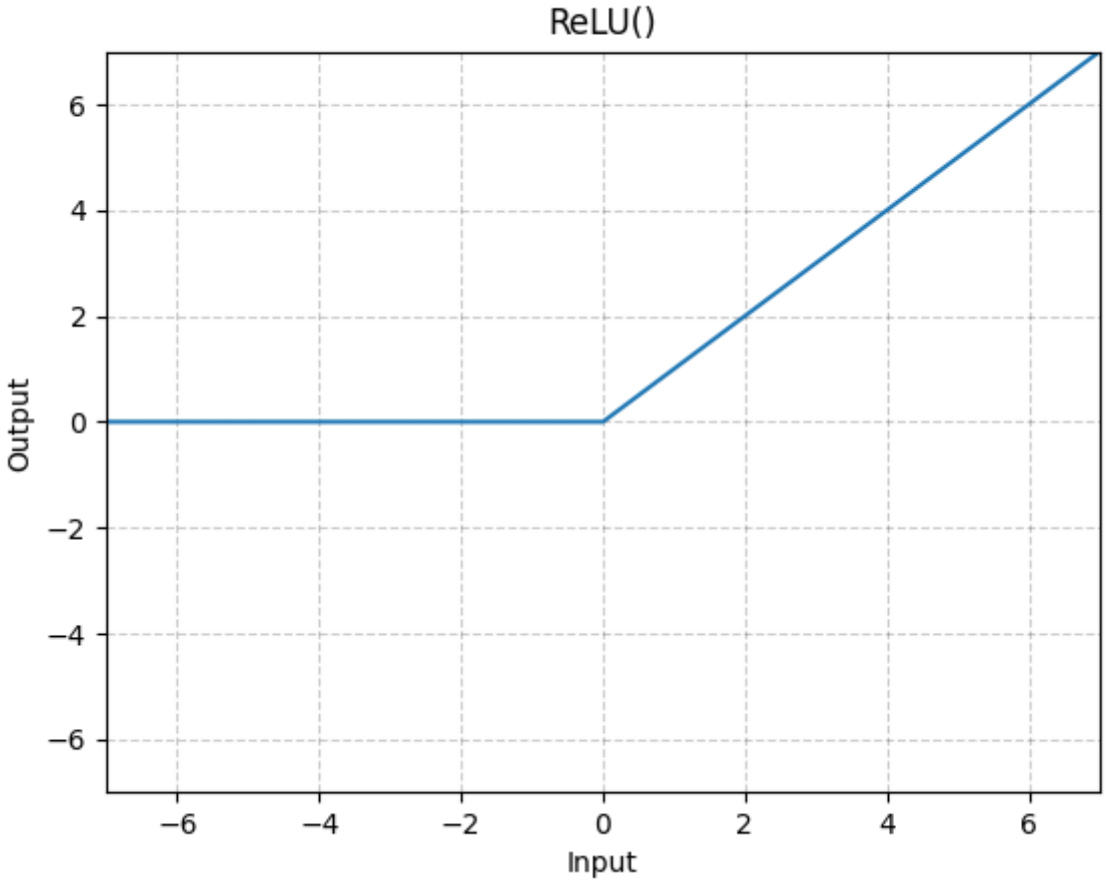
che è pessimo in quanto il modello utilizza un "linear model" che sostanzialmente rappresenta una linea che negli assi cartesiani ha un'intercetta e una direzione e quindi non riscurà mai a rappresentare i dati.



Bisoga quindi cambiare modello.

In particolare bisogna introdurre una funziona **non lineare** come per es. la ReLU che nella prarica ritorna zero se i valori sono <=0 oppure il valore stesso se >0.

Di seguito il grafico della funzione non lineare ReLU.



Modifichiamo quindi il modello aggiungendo dopo l'hidden layer la funzione di attivazione non lineare come nell'esempio di seguito:

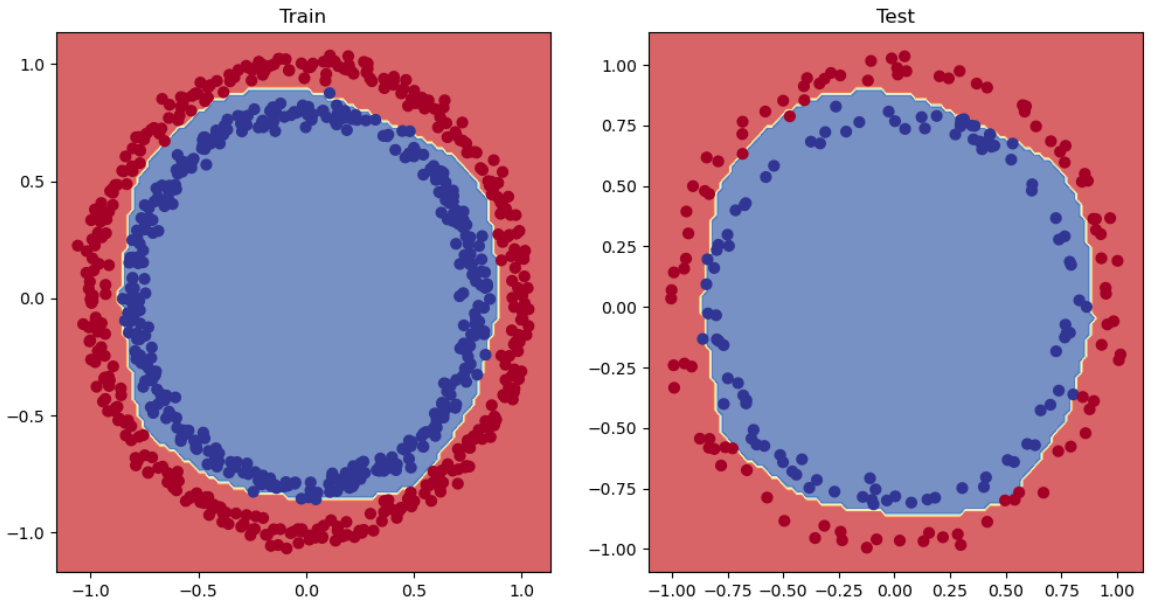
```
# costruisco il modello
model_0 = nn.Sequential(

                      nn.Linear(in_features=2, out_features=10),
                      nn.ReLU(),
                      nn.Linear(in_features=10, out_features=10),
                      nn.ReLU(),
                      nn.Linear(in_features=10, out_features=1))
```

che produce risultati decisamente migliori:

Epoch: 0 | Train -> Loss: 0.69656 , Acc: 47.38% | Test -> Loss: 0.69921%. Acc: 46.00%
Epoch: 10 | Train -> Loss: 0.69417 , Acc: 46.00% | Test -> Loss: 0.69735%. Acc: 43.00%
Epoch: 20 | Train -> Loss: 0.69257 , Acc: 49.62% | Test -> Loss: 0.69603%. Acc: 49.50%
Epoch: 30 | Train -> Loss: 0.69123 , Acc: 50.38% | Test -> Loss: 0.69486%. Acc: 48.50%
Epoch: 40 | Train -> Loss: 0.69000 , Acc: 51.00% | Test -> Loss: 0.69374%. Acc: 49.50%
Epoch: 50 | Train -> Loss: 0.68884 , Acc: 51.50% | Test -> Loss: 0.69266%. Acc: 49.50%
Epoch: 60 | Train -> Loss: 0.68772 , Acc: 52.62% | Test -> Loss: 0.69162%. Acc: 48.50%
Epoch: 70 | Train -> Loss: 0.68663 , Acc: 53.00% | Test -> Loss: 0.69060%. Acc: 49.00%
Epoch: 80 | Train -> Loss: 0.68557 , Acc: 53.25% | Test -> Loss: 0.68960%. Acc: 48.50%
Epoch: 90 | Train -> Loss: 0.68453 , Acc: 53.25% | Test -> Loss: 0.68862%. Acc: 48.50%
Epoch: 100 | Train -> Loss: 0.68349 , Acc: 54.12% | Test -> Loss: 0.68765%. Acc: 49.00%
Epoch: 110 | Train -> Loss: 0.68246 , Acc: 54.37% | Test -> Loss: 0.68670%. Acc: 48.50%
Epoch: 120 | Train -> Loss: 0.68143 , Acc: 54.87% | Test -> Loss: 0.68574%. Acc: 49.00%
Epoch: 130 | Train -> Loss: 0.68039 , Acc: 54.75% | Test -> Loss: 0.68478%. Acc: 49.00%
Epoch: 140 | Train -> Loss: 0.67935 , Acc: 55.50% | Test -> Loss: 0.68382%. Acc: 50.00%
Epoch: 150 | Train -> Loss: 0.67829 , Acc: 55.62% | Test -> Loss: 0.68285%. Acc: 50.50%
Epoch: 160 | Train -> Loss: 0.67722 , Acc: 57.25% | Test -> Loss: 0.68188%. Acc: 53.50%
Epoch: 170 | Train -> Loss: 0.67614 , Acc: 59.62% | Test -> Loss: 0.68090%. Acc: 57.50%
Epoch: 180 | Train -> Loss: 0.67504 , Acc: 61.62% | Test -> Loss: 0.67991%. Acc: 59.00%
Epoch: 190 | Train -> Loss: 0.67390 , Acc: 63.75% | Test -> Loss: 0.67891%. Acc: 59.50%
Epoch: 200 | Train -> Loss: 0.67275 , Acc: 65.50% | Test -> Loss: 0.67789%. Acc: 60.50%
Epoch: 210 | Train -> Loss: 0.67156 , Acc: 66.50% | Test -> Loss: 0.67686%. Acc: 60.50%
Epoch: 220 | Train -> Loss: 0.67036 , Acc: 68.62% | Test -> Loss: 0.67580%. Acc: 63.50%
Epoch: 230 | Train -> Loss: 0.66912 , Acc: 70.75% | Test -> Loss: 0.67473%. Acc: 64.50%
Epoch: 240 | Train -> Loss: 0.66787 , Acc: 72.00% | Test -> Loss: 0.67363%. Acc: 66.00%
Epoch: 250 | Train -> Loss: 0.66658 , Acc: 73.75% | Test -> Loss: 0.67252%. Acc: 67.50%
Epoch: 260 | Train -> Loss: 0.66526 , Acc: 74.88% | Test -> Loss: 0.67139%. Acc: 69.00%
Epoch: 270 | Train -> Loss: 0.66392 , Acc: 75.75% | Test -> Loss: 0.67025%. Acc: 69.50%
Epoch: 280 | Train -> Loss: 0.66256 , Acc: 77.62% | Test -> Loss: 0.66909%. Acc: 72.00%
Epoch: 290 | Train -> Loss: 0.66118 , Acc: 78.75% | Test -> Loss: 0.66791%. Acc: 72.50%
Epoch: 300 | Train -> Loss: 0.65978 , Acc: 79.75% | Test -> Loss: 0.66672%. Acc: 75.50%
Epoch: 310 | Train -> Loss: 0.65835 , Acc: 80.75% | Test -> Loss: 0.66552%. Acc: 76.00%
Epoch: 320 | Train -> Loss: 0.65689 , Acc: 81.88% | Test -> Loss: 0.66431%. Acc: 77.00%
Epoch: 330 | Train -> Loss: 0.65540 , Acc: 82.75% | Test -> Loss: 0.66309%. Acc: 77.50%
Epoch: 340 | Train -> Loss: 0.65390 , Acc: 84.38% | Test -> Loss: 0.66183%. Acc: 78.50%
Epoch: 350 | Train -> Loss: 0.65237 , Acc: 85.12% | Test -> Loss: 0.66056%. Acc: 78.50%
Epoch: 360 | Train -> Loss: 0.65083 , Acc: 85.25% | Test -> Loss: 0.65927%. Acc: 81.00%
Epoch: 370 | Train -> Loss: 0.64925 , Acc: 85.88% | Test -> Loss: 0.65797%. Acc: 81.50%
Epoch: 380 | Train -> Loss: 0.64763 , Acc: 86.38% | Test -> Loss: 0.65664%. Acc: 83.00%
Epoch: 390 | Train -> Loss: 0.64599 , Acc: 87.00% | Test -> Loss: 0.65530%. Acc: 83.50%
Epoch: 400 | Train -> Loss: 0.64430 , Acc: 87.38% | Test -> Loss: 0.65394%. Acc: 84.50%
Epoch: 410 | Train -> Loss: 0.64258 , Acc: 88.75% | Test -> Loss: 0.65256%. Acc: 85.00%
Epoch: 420 | Train -> Loss: 0.64083 , Acc: 89.50% | Test -> Loss: 0.65115%. Acc: 86.00%
Epoch: 430 | Train -> Loss: 0.63904 , Acc: 89.62% | Test -> Loss: 0.64971%. Acc: 86.50%

Epoch: 440 | Train -> Loss: 0.63723 , Acc: 90.75% | Test -> Loss: 0.64825%. Acc: 87.00%
Epoch: 450 | Train -> Loss: 0.63540 , Acc: 91.38% | Test -> Loss: 0.64678%. Acc: 87.00%
Epoch: 460 | Train -> Loss: 0.63354 , Acc: 92.38% | Test -> Loss: 0.64529%. Acc: 87.00%
Epoch: 470 | Train -> Loss: 0.63165 , Acc: 93.00% | Test -> Loss: 0.64377%. Acc: 88.00%
Epoch: 480 | Train -> Loss: 0.62974 , Acc: 93.38% | Test -> Loss: 0.64222%. Acc: 89.00%
Epoch: 490 | Train -> Loss: 0.62780 , Acc: 93.50% | Test -> Loss: 0.64065%. Acc: 91.00%
Epoch: 500 | Train -> Loss: 0.62585 , Acc: 94.38% | Test -> Loss: 0.63905%. Acc: 91.50%
Epoch: 510 | Train -> Loss: 0.62386 , Acc: 94.75% | Test -> Loss: 0.63746%. Acc: 92.50%
Epoch: 520 | Train -> Loss: 0.62183 , Acc: 95.25% | Test -> Loss: 0.63584%. Acc: 92.50%
Epoch: 530 | Train -> Loss: 0.61979 , Acc: 95.50% | Test -> Loss: 0.63421%. Acc: 92.50%
Epoch: 540 | Train -> Loss: 0.61773 , Acc: 95.75% | Test -> Loss: 0.63255%. Acc: 92.50%
Epoch: 550 | Train -> Loss: 0.61564 , Acc: 95.62% | Test -> Loss: 0.63088%. Acc: 93.00%
Epoch: 560 | Train -> Loss: 0.61351 , Acc: 96.00% | Test -> Loss: 0.62917%. Acc: 93.50%
Epoch: 570 | Train -> Loss: 0.61136 , Acc: 96.00% | Test -> Loss: 0.62742%. Acc: 94.00%
Epoch: 580 | Train -> Loss: 0.60919 , Acc: 96.12% | Test -> Loss: 0.62565%. Acc: 94.50%
Epoch: 590 | Train -> Loss: 0.60699 , Acc: 96.00% | Test -> Loss: 0.62385%. Acc: 94.50%
Epoch: 600 | Train -> Loss: 0.60477 , Acc: 96.50% | Test -> Loss: 0.62203%. Acc: 94.50%
Epoch: 610 | Train -> Loss: 0.60253 , Acc: 96.50% | Test -> Loss: 0.62020%. Acc: 94.00%
Epoch: 620 | Train -> Loss: 0.60026 , Acc: 96.75% | Test -> Loss: 0.61833%. Acc: 94.00%
Epoch: 630 | Train -> Loss: 0.59796 , Acc: 97.00% | Test -> Loss: 0.61643%. Acc: 94.50%
Epoch: 640 | Train -> Loss: 0.59563 , Acc: 97.25% | Test -> Loss: 0.61449%. Acc: 94.50%
Epoch: 650 | Train -> Loss: 0.59327 , Acc: 97.38% | Test -> Loss: 0.61253%. Acc: 94.50%
Epoch: 660 | Train -> Loss: 0.59086 , Acc: 97.62% | Test -> Loss: 0.61053%. Acc: 94.50%
Epoch: 670 | Train -> Loss: 0.58843 , Acc: 97.62% | Test -> Loss: 0.60850%. Acc: 94.00%
Epoch: 680 | Train -> Loss: 0.58595 , Acc: 97.88% | Test -> Loss: 0.60642%. Acc: 95.00%
Epoch: 690 | Train -> Loss: 0.58343 , Acc: 97.88% | Test -> Loss: 0.60429%. Acc: 95.50%
Epoch: 700 | Train -> Loss: 0.58088 , Acc: 97.88% | Test -> Loss: 0.60211%. Acc: 95.50%
Epoch: 710 | Train -> Loss: 0.57830 , Acc: 98.00% | Test -> Loss: 0.59991%. Acc: 96.00%
Epoch: 720 | Train -> Loss: 0.57569 , Acc: 98.25% | Test -> Loss: 0.59767%. Acc: 96.50%
Epoch: 730 | Train -> Loss: 0.57305 , Acc: 98.38% | Test -> Loss: 0.59541%. Acc: 96.50%
Epoch: 740 | Train -> Loss: 0.57037 , Acc: 98.50% | Test -> Loss: 0.59309%. Acc: 96.50%
Epoch: 750 | Train -> Loss: 0.56766 , Acc: 98.50% | Test -> Loss: 0.59073%. Acc: 96.50%
Epoch: 760 | Train -> Loss: 0.56493 , Acc: 98.62% | Test -> Loss: 0.58835%. Acc: 97.00%
Epoch: 770 | Train -> Loss: 0.56216 , Acc: 98.62% | Test -> Loss: 0.58594%. Acc: 97.00%
Epoch: 780 | Train -> Loss: 0.55935 , Acc: 98.62% | Test -> Loss: 0.58352%. Acc: 97.00%
Epoch: 790 | Train -> Loss: 0.55652 , Acc: 98.88% | Test -> Loss: 0.58106%. Acc: 97.00%
Epoch: 800 | Train -> Loss: 0.55365 , Acc: 98.88% | Test -> Loss: 0.57855%. Acc: 97.00%
Epoch: 810 | Train -> Loss: 0.55075 , Acc: 98.88% | Test -> Loss: 0.57604%. Acc: 97.00%
Epoch: 820 | Train -> Loss: 0.54782 , Acc: 98.88% | Test -> Loss: 0.57351%. Acc: 97.00%
Epoch: 830 | Train -> Loss: 0.54485 , Acc: 99.00% | Test -> Loss: 0.57095%. Acc: 97.00%
Epoch: 840 | Train -> Loss: 0.54185 , Acc: 99.00% | Test -> Loss: 0.56836%. Acc: 97.00%
Epoch: 850 | Train -> Loss: 0.53884 , Acc: 99.00% | Test -> Loss: 0.56572%. Acc: 97.50%
Epoch: 860 | Train -> Loss: 0.53580 , Acc: 99.00% | Test -> Loss: 0.56307%. Acc: 97.50%
Epoch: 870 | Train -> Loss: 0.53274 , Acc: 99.00% | Test -> Loss: 0.56040%. Acc: 97.50%

```
Epoch: 880 | Train -> Loss: 0.52964 , Acc: 99.00% | Test -> Loss: 0.55773%. Acc: 97.50%
Epoch: 890 | Train -> Loss: 0.52652 , Acc: 99.12% | Test -> Loss: 0.55503%. Acc: 97.50%
Epoch: 900 | Train -> Loss: 0.52337 , Acc: 99.25% | Test -> Loss: 0.55228%. Acc: 97.50%
Epoch: 910 | Train -> Loss: 0.52019 , Acc: 99.25% | Test -> Loss: 0.54952%. Acc: 97.50%
Epoch: 920 | Train -> Loss: 0.51700 , Acc: 99.25% | Test -> Loss: 0.54673%. Acc: 97.00%
Epoch: 930 | Train -> Loss: 0.51378 , Acc: 99.25% | Test -> Loss: 0.54393%. Acc: 97.00%
Epoch: 940 | Train -> Loss: 0.51053 , Acc: 99.25% | Test -> Loss: 0.54110%. Acc: 97.50%
Epoch: 950 | Train -> Loss: 0.50726 , Acc: 99.25% | Test -> Loss: 0.53826%. Acc: 97.50%
Epoch: 960 | Train -> Loss: 0.50398 , Acc: 99.25% | Test -> Loss: 0.53538%. Acc: 97.50%
Epoch: 970 | Train -> Loss: 0.50067 , Acc: 99.38% | Test -> Loss: 0.53248%. Acc: 97.50%
Epoch: 980 | Train -> Loss: 0.49734 , Acc: 99.38% | Test -> Loss: 0.52956%. Acc: 98.00%
Epoch: 990 | Train -> Loss: 0.49399 , Acc: 99.38% | Test -> Loss: 0.52664%. Acc: 98.00% 3
```



(vedi sorgente completo in attachement a questa pagina 02_classification.py)

Revision #16
Created 2023-03-11 17:08:04 UTC by marco
Updated 2023-05-09 20:10:34 UTC by marco