Pytorch for dummy

PyTorch: a deep learning framework

- One of the most popular frameworks
- Originally developed by Meta AI, now part of Linux Foundation
- Intuitive and user-friendly
- Similarities with NumPy

Originariamente impletato da META ora fa parte della Linux foundation

La base di tutto è il tensore, che non è altro che una matrice (o un array) sulla quale PT consente tutta una serie di operazioni, un po' come numpy, es:

Tensor attributes

- **Tensor shape**

```
my_list = [[1, 2, 3], [4, 5, 6]]
tensor = torch.tensor(my_list)
print(tensor.shape)
```

```
torch.Size([2, 3])
```

- **Tensor data type**

```
print(tensor.dtype)
```

```
torch.int64
```

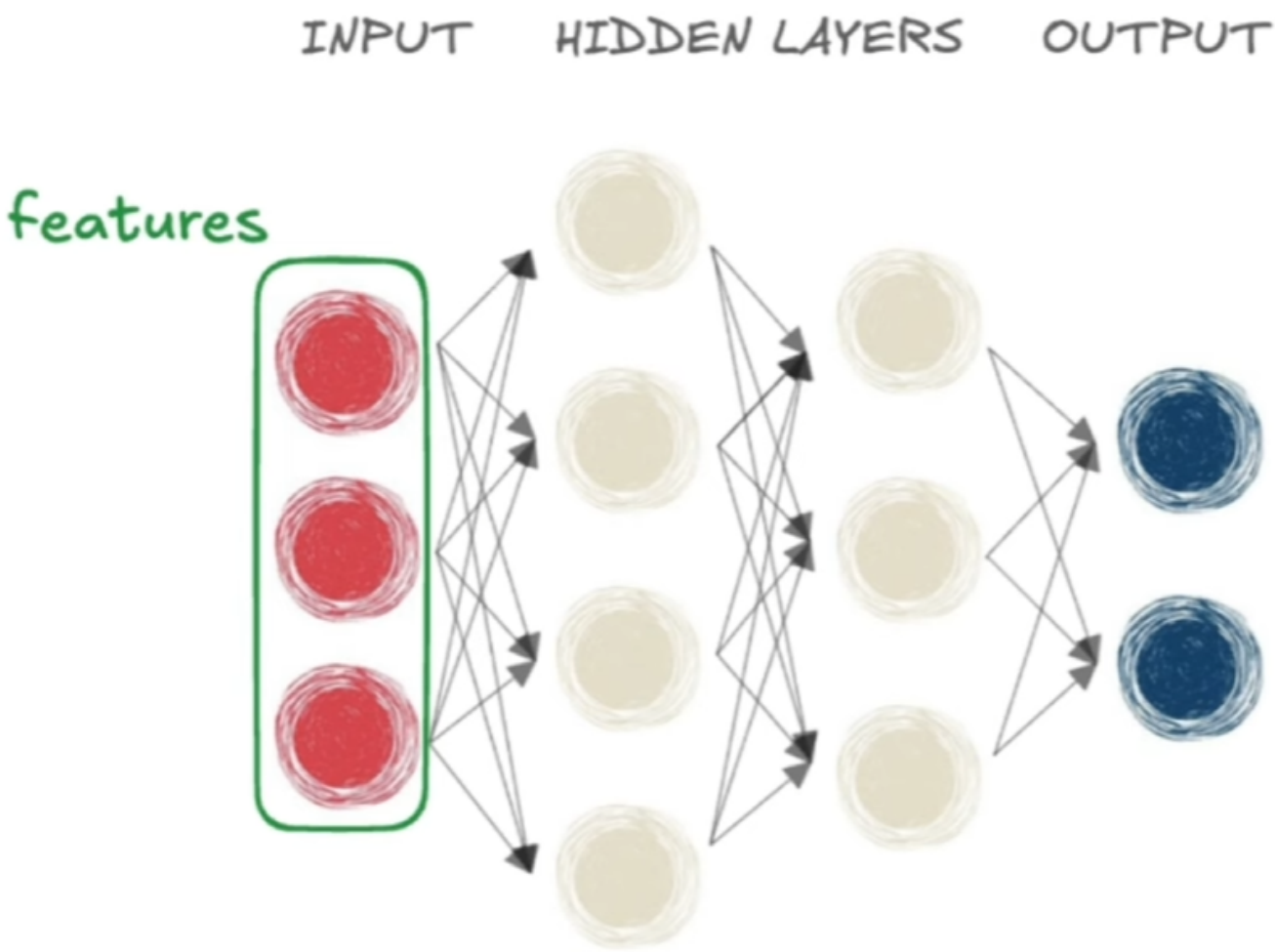
es:

Element-wise multiplicatior

```
a = torch.tensor([[1, 1],
                  [2, 2]])
b = torch.tensor([[2, 2],
                  [3, 3]])
print(a * b)
```

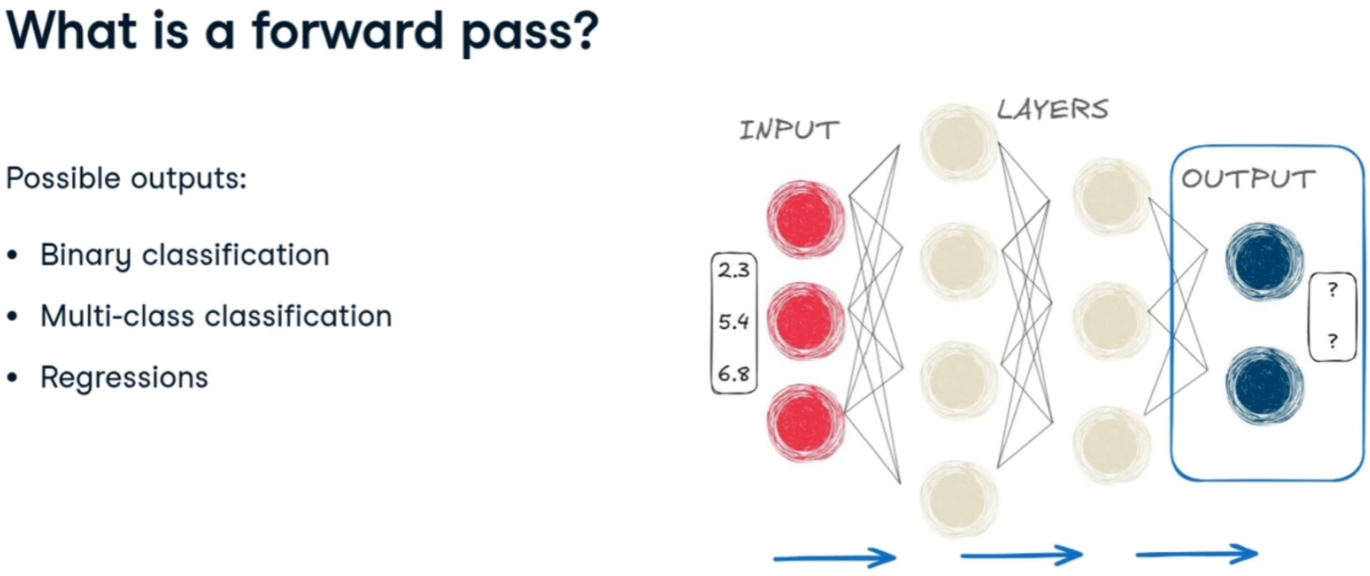
```
tensor([[2,  2],
        [6, 6]])
```

Layer della rete neurale



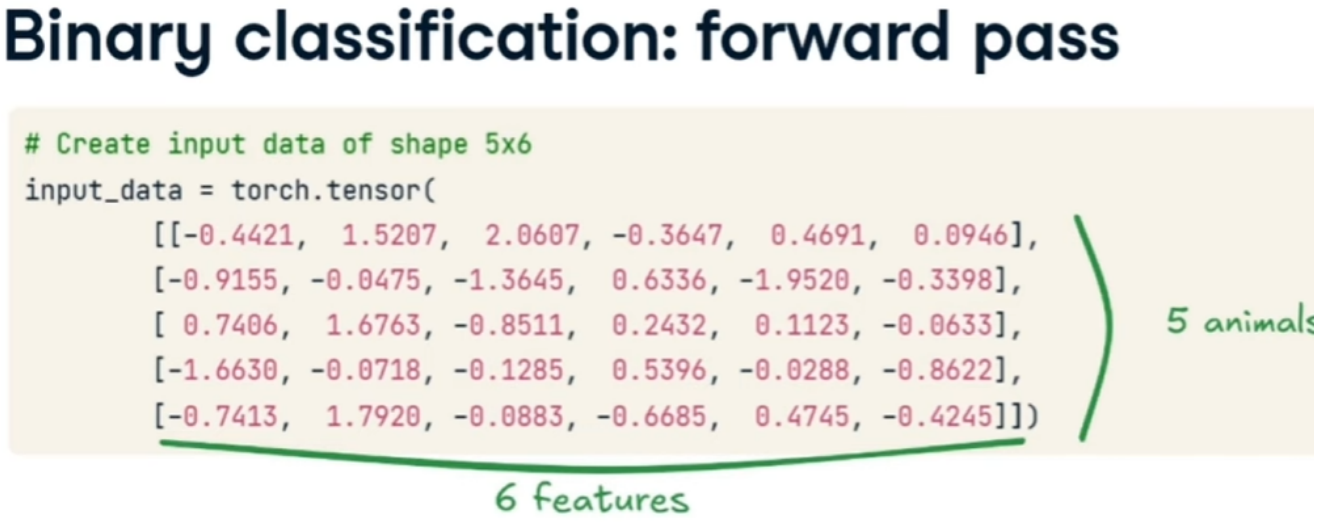
la parte in rosso sono gli input della rete, detta "features", la parte in grigio sono i layer "nascosi", mentra la parte di blu è l'output layer ovvero l'output desiderato.

Classificatori



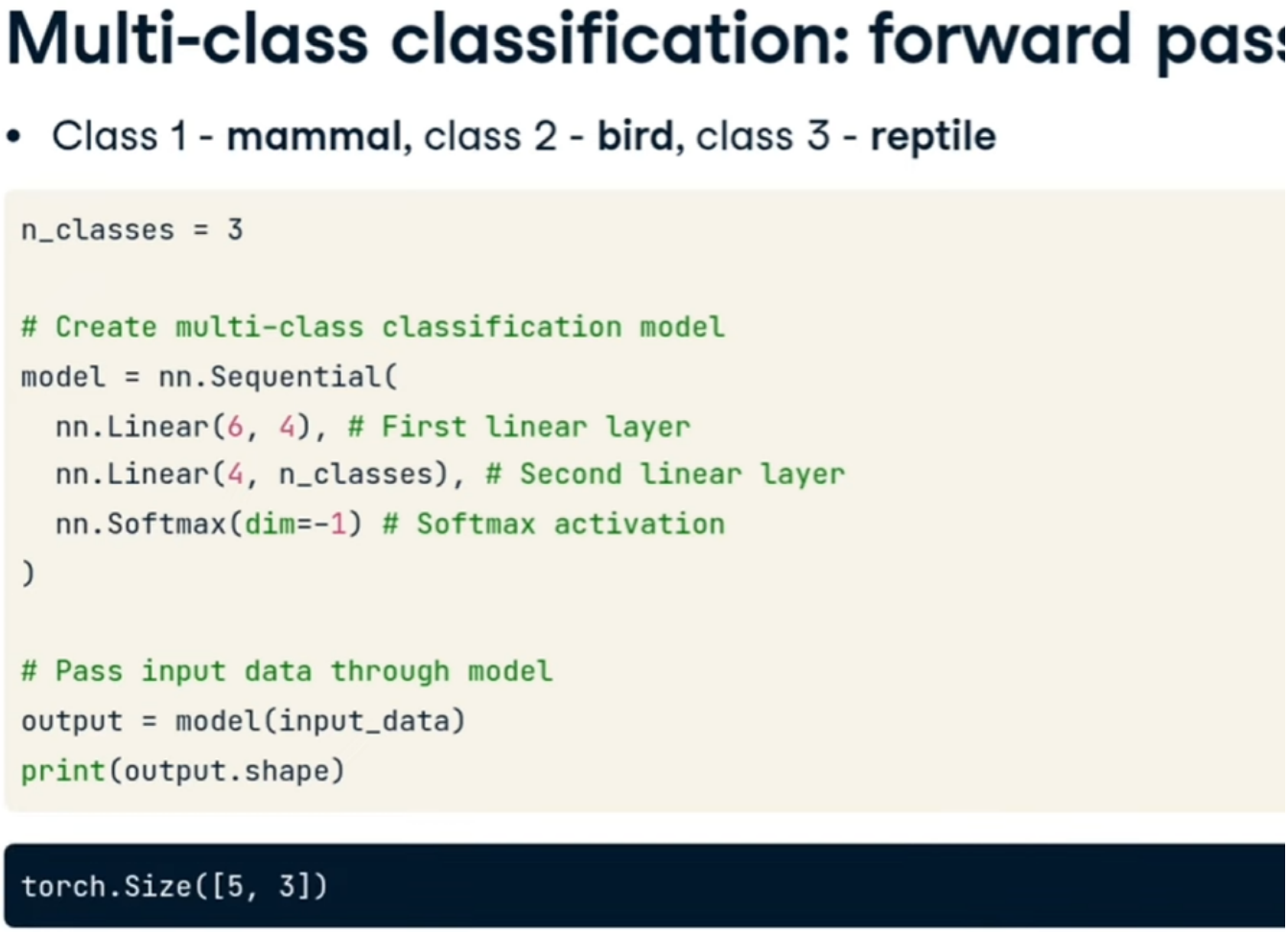
Le funzioni possono essere:

- **Sigmoid** per la classificazione binaria (un unico output con un valore compreso tra 0 e 1)




```
# Create binary classification model
model = nn.Sequential(
  nn.Linear(6, 4), # First linear layer
  nn.Linear(4, 1), # Second linear layer
  nn.Sigmoid() # Sigmoid activation function
)
```

- **Softmax** per la multi classificazione, va messo come ultimi layer della rete neurale. (dove l'ultimo livelo di neurino definice il numero di valori da classificre)



- yy per la regressione, ovvero per predirre un flusso continuo di valori numerici, in questo caso non verrò inserita nessuna funzione di attivazione

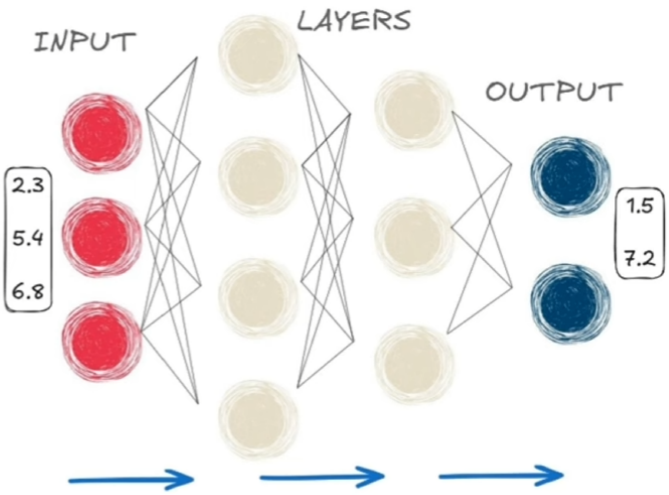
Forward pass

è l'operazione di passaggio dei pesi e del bias da un layer della rete a quello successivo

What is a forward pass?

- Input data **flows** through layers
- **Calculations** performed at each layer
- Final layer generates **outputs**

- Outputs produced based on **weights** and **biases**

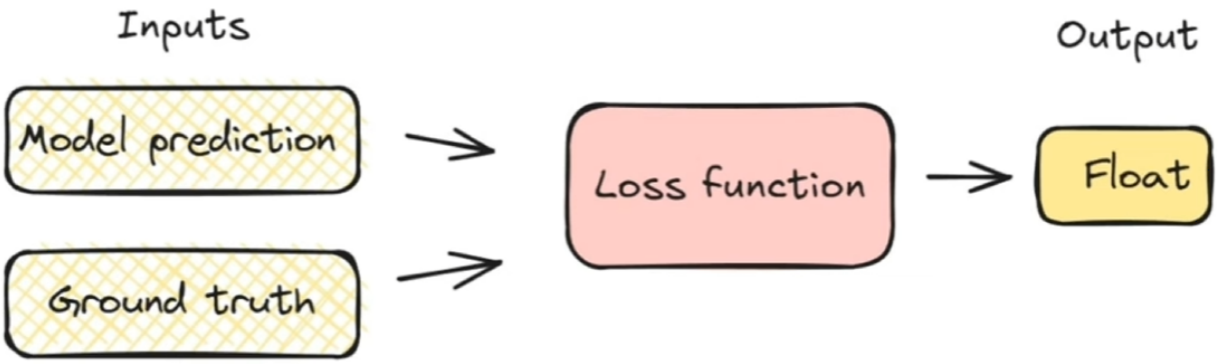


Loss function

La LF indica quanto il modello è efficace nel predirre i valori durante la fase di training.

Why do we need a loss function?

- Tells us how good our model is during training
- Takes a model **prediction** $\hat{y}$ and **ground truth** y
- Outputs a float



La funzione di "loss" indicata come F, riceve in input i valori corretti associati alle features utilizzate durante il training e quelli generati dal modello -> F(y,**y'**)

L'outuput è un valore numerico

Transforming labels with one-hot encoding

```
import torch.nn.functional as F

print(F.one_hot(torch.tensor(0), num_classes = 3))
```

```
tensor([1, 0, 0])
```

```
print(F.one_hot(torch.tensor(1), num_classes = 3))
```

```
tensor([0, 1, 0])
```

```
print(F.one_hot(torch.tensor(2), num_classes = 3))
```

```
tensor([0, 0, 1])
```

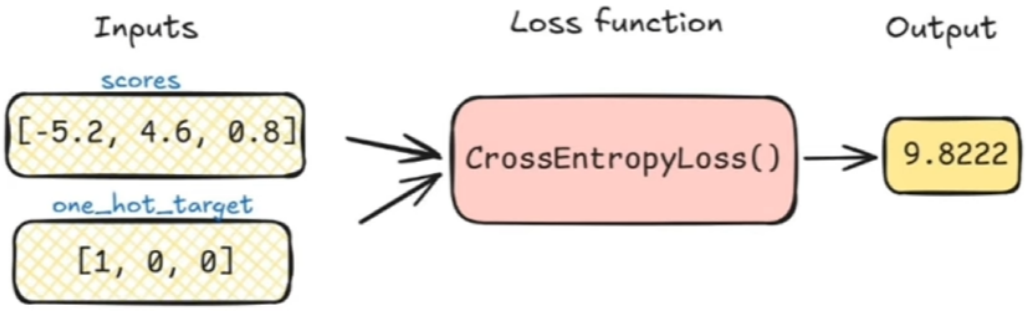
Una delle funzioni di loss function è la *CrossEntropyLoss* che vuole in input i valori calcolati dalla rete e le label che rappresentano il valore "vero". L'ouput è il valore di "**loss**" vero e proprio che, attraverso la backpropagation bisogna minimizzare.

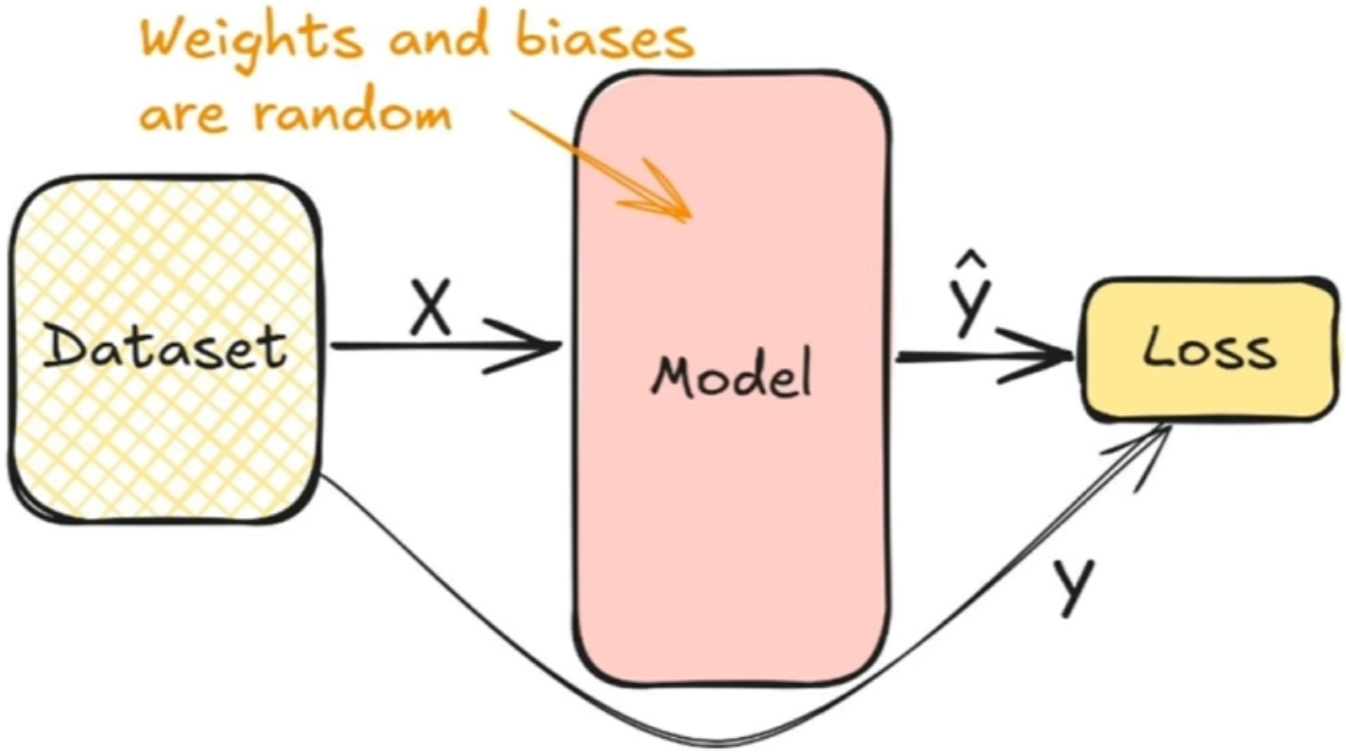
Loss function takes:

- **scores** - model predictions **before** the final softmax function
- **one_hot_target** - one hot encoded ground truth label

Loss function outputs:

- **loss** - a single **float**

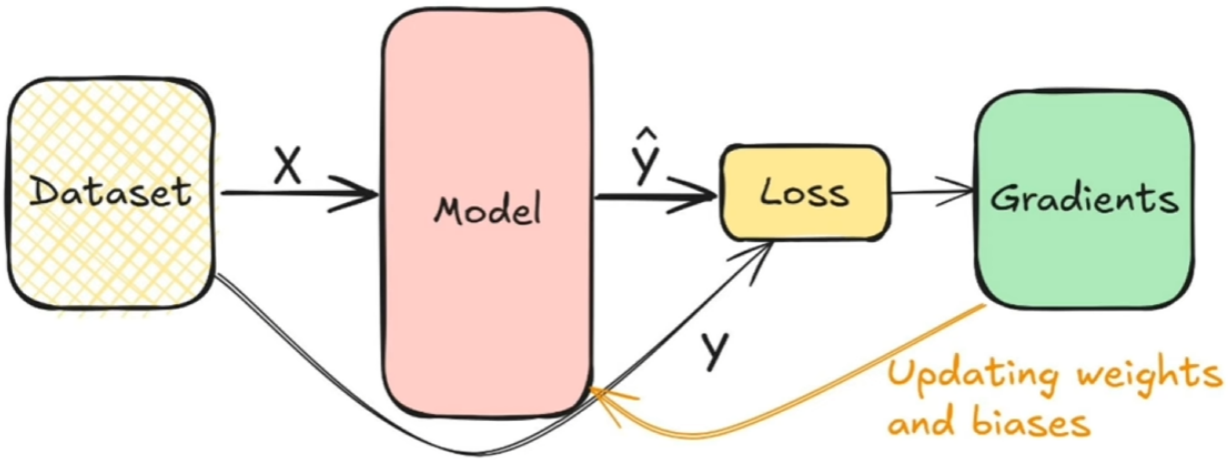




La Backpropagation

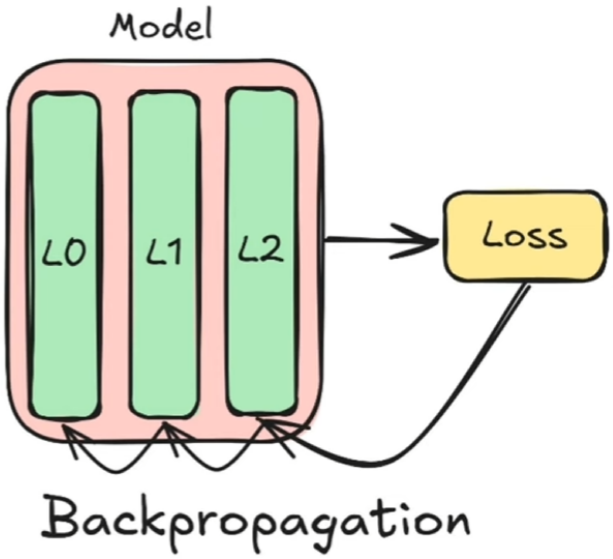
Una volta calcolati i pesi e i bias della rete neurale, si prende il valore generato y' e si effettua un'operazione di backpropagation che, attraverso il calcolo della discesa del gradiente va a ricalcolare i pesi e i bias a ritroso per ciascun layar, al fine dir minimizzare l'errore.

- Gradients help minimize **loss**, tune layer **weights** and **biases**



Backpropagation concepts

- Consider a network made of three layers:
  - Begin with loss gradients for $L2$
  - Use $L2$ to compute $L1$ gradients
  - Repeat for all layers $(L1, L0)$



vediamolo in PT:

Gradient descent

- For non-convex functions, we will use **gradient descent**
- PyTorch simplifies this with **optimizers**
  - Stochastic gradient descent (SGD)

```
import torch.optim as optim

# Create the optimizer
optimizer = optim.SGD(model.parameters(), lr=0.001)

# Perform parameter updates
optimizer.step()
```

← OTTIMIZZA

Preparazione dei dati per il training

Ci sono 4 passi fondamentali prima di "allenare" la rete neurale, ovvero:

prendiamo per esempio un dataset di animali dove le prime colonne (esclusa la zero che è puramente decrittiva) rappresentano le "features" mentre l'ultima indica il tipo di animale:

Our animals dataset

```
import pandas as pd
animals = pd.read_csv('animal_dataset.csv')
```

animal_name	hair	feathers	eggs	milk	predator	legs	tail	type
sparrow	0	1	1	0	0	2	1	0
eagle	0	1	1	0	1	2	1	0
cat	1	0	0	1	1	4	1	1
dog	1	0	0	1	0	4	1	1
lizard	0	0	1	0	1	4	1	2

Type categories: **bird** (0), **mammal** (1), **reptile** (2)

selezioniamo le fetures:

Our animals dataset: defining features

```
import numpy as np

# Define input features
features = animals.iloc[:, 1:-1]

X = features.to_numpy()
print(X)
```

```
[[0 1 1 0 0 2 1]
 [0 1 1 0 1 2 1]
 [1 0 0 1 1 4 1]
 [1 0 0 1 0 4 1]
 [0 0 1 0 1 4 1]]
```

ora le labels

Back to our animals dataset: defining target values

```
# Define target values (ground truth)
target = animals.iloc[:, -1]
y = target.to_numpy()
print(y)
```

```
[0 0 1 1 2]
```

Ora utilizziamo l'oggetto TensorDataset per caricare le x e le y:

TensorDataset

```
import torch
from torch.utils.data import TensorDataset

# Instantiate dataset class
dataset = TensorDataset(torch.tensor(X), torch.tensor(y))

# Access an individual sample
input_sample, label_sample = dataset[0]
print('input sample:', input_sample)
print('label_sample:', label_sample)
```

ora creiamo il dataloader per gestire il carico dei dati efficacemente durante il training

DataLoader

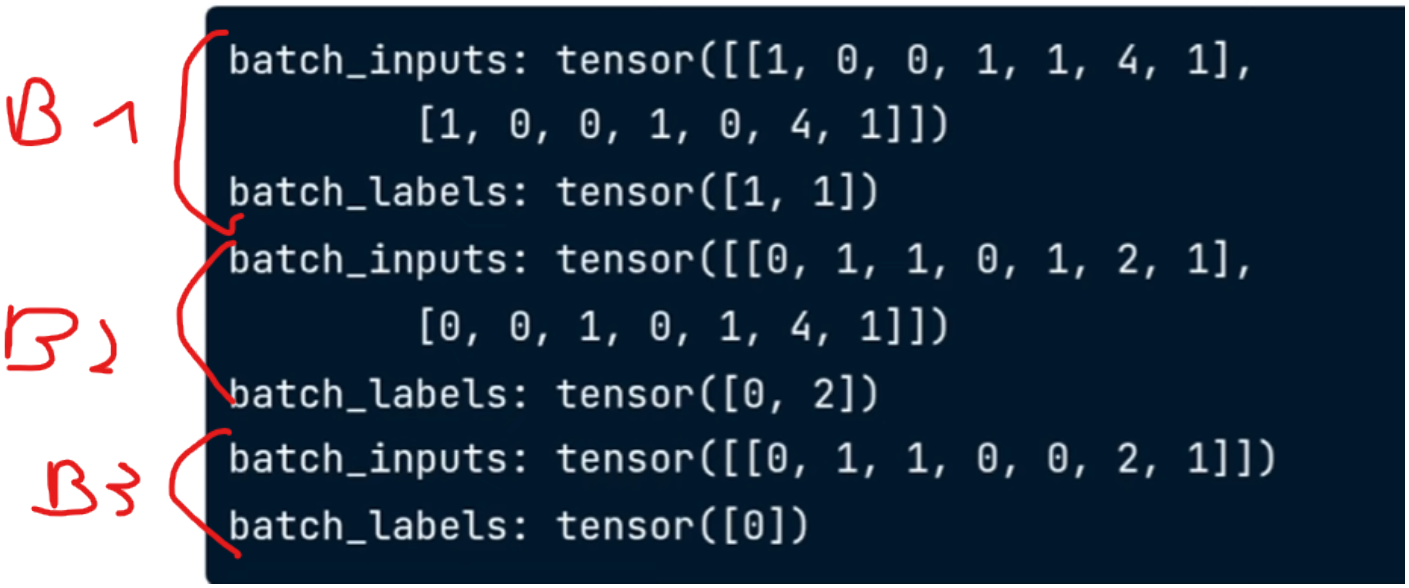
```
from torch.utils.data import DataLoader

batch_size = 2
shuffle = True

# Create a DataLoader
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)
```

- **Epoch:** one full pass through the training dataloader
- **Generalization:** model performs will with unseen data

avendo setto il batch size a 2 ad ogni iterazione del dataloader estrarrò solo un bach di due elementi (in questo 2 carratteristiche di animali e il tipo), come sotto riportato:

DataLoader

```
# Iterate over the dataloader
for batch_inputs, batch_labels in dataloader:
    print('batch_inputs:', batch_inputs)
    print('batch_labels:', batch_labels)
```

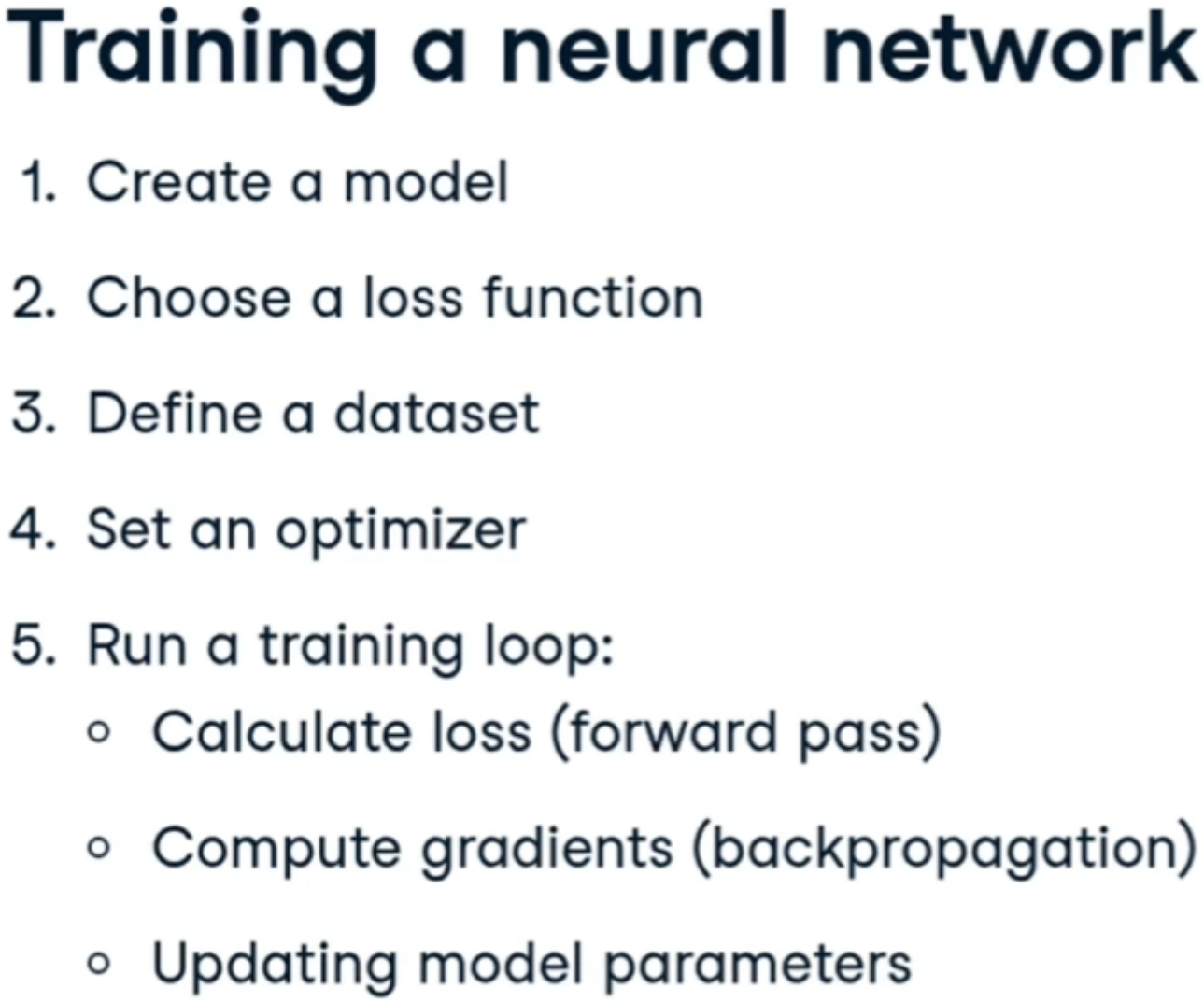


essemdo solo 5 animali si può notarer come l'ultimo batch contenga un solo animale.

Quindi il cliclo for fa passare tutto il dataset.

Training

ora possiamo procedere con il training, che consiste in:



Il training è molto importante perchè consenti di minimizzare la loss e di appore delle modifiche al training stesso.

Regressione

La regressione consente di avere un valore lineare come output.

Per la regressione so utilizza in genere la funzione di loss MSE (mean square error)

Mean Squared Error Loss

- MSE loss is the mean of the squared difference between predictions and ground truth

```
def mean_squared_loss(prediction, target):
  return np.mean((prediction - target)**2)
```

- in PyTorch:

```
criterion = nn.MSELoss()
# Prediction and target are float tensors
loss = criterion(prediction, target)
```

facciamo un esempio di regression i gli stipendi dei data scientist:

Introducing the Data Science Salary dataset

experience_level	employment_type	remote_ratio	company_size	salary_in_usd
0	0	0.5	1	0.036
1	0	1.0	2	0.133
2	0	0.0	1	0.234
1	0	1.0	0	0.076
2	0	1.0	1	0.170

- Features: categorical, target: salary (USD)
- Final output: linear layer
- Loss: regression-specific

creiamo la rete neurale

Before the training loop

```
# Create the dataset and the dataloader
dataset = TensorDataset(torch.tensor(features).float(),
                        torch.tensor(target).float())

dataloader = DataLoader(dataset, batch_size=4, shuffle=True)

# Create the model
model = nn.Sequential(nn.Linear(4, 2),
                      nn.Linear(2, 1))

# Create the loss and optimizer
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.001)
```

adesso loppiamo su tutto il dataset

```
# The training loop
for epoch in range(num_epochs):
  for data in dataloader:
    # va azzerato ad ogni epoca
    optimizer.zero_grad()

    # Get feature and target from the data loader
    feature, target = data

    # Run a forward pass
    pred = model(feature)

    # Compute loss and gradients
    loss = criterion(pred, target)
    loss.backward()
```

```
    # Update the parameters
    optimizer.step()
```

Utilizzo Softmax vs ReLU.

E' emerso che per gli hidden layer è meglio utilizzare la **funzione di attivazione** ReLU, mentre per l'output layer si può utilizzare anche la Softmax.

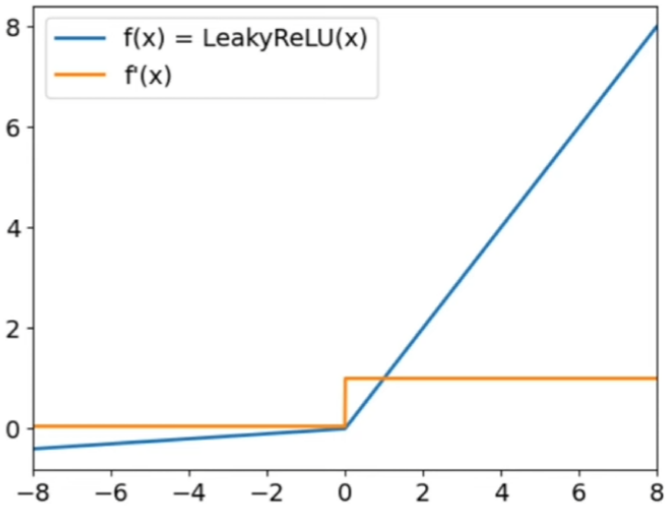
Leaky ReLU

Migliora la ReLU moltiplicando i valori di input per un coefficiente che evita i casi di disattivazione totale del neurone che causa lo stop dell'apprendimento.

Leaky ReLU

Leaky ReLU:

- **Positive** inputs behave like ReLU
- **Negative** inputs are scaled by a small coefficient (default 0.01)
- Gradients for negative inputs are **non-zero**



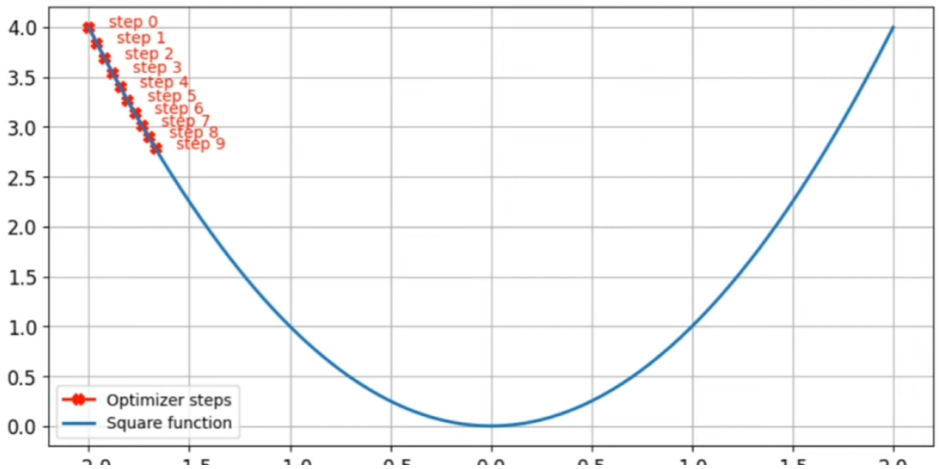
In PyTorch:

```
leaky_relu = nn.LeakyReLU(
  negative_slope = 0.05)
```

Learing rate e momentum

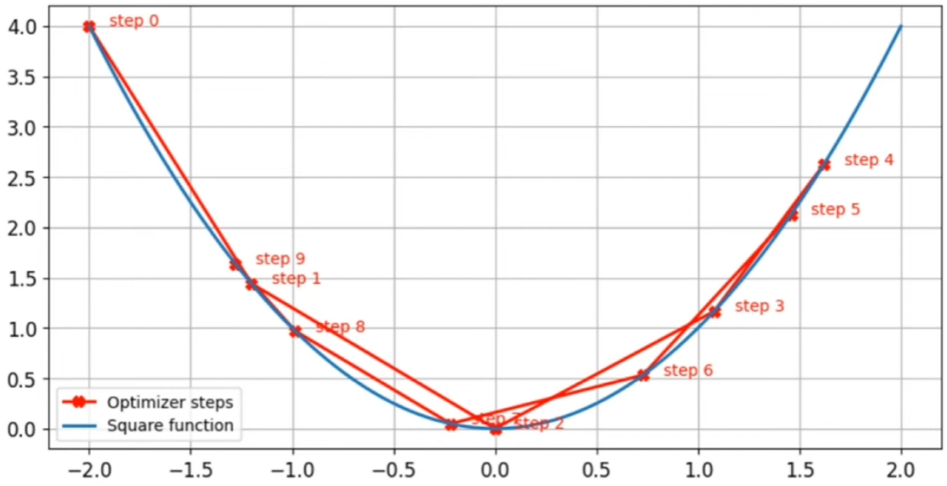
Il LR è il passo utilizzato per arrivare al mimimo durante la fase della discesa del gradiente, se è troppo piccolo non arrivieremo al minimo, come qui:

Impact of the learning rate: small learning rate



se è troppo grande, continua a rimbalzare senza trovare cmq il minimo, come qui:

Impact of the learning rate: high learning rate



Il "momento" invece rappresenta l'inzeria con la quale si effettuano i passi, serve per evitare di fermarsi ad un "minimo locale", in sintesi:

Summary

Learning Rate	Momentum
Controls the step size	Controls the inertia
Too high → poor performance	Helps escape local minimum
Too low → slow training	Too small → optimizer gets stuck
Typical range: 0.01 (10^{-2}) and 0.0001 (10^{-4})	Typical range: **0.85 to 0.99**

Valutazione del modello

https://www.youtube.com/watch?v=IFsVsXAqPto

47:37 Evaluating Models with Training and Validation Data

Revision #7
Created 2025-12-08 16:02:10 UTC by marco
Updated 2025-12-08 17:59:59 UTC by marco