Pipeline de um processo de Deep Learning implementado em PyTorch:

1. Preparar os Dados
2. Definir o Modelo
3. Treinar o Modelo
4. Avaliar o Modelo
5. Usar o Modelo

MLP para classificação Multiclass

Iris flowers multiclass classification dataset
https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv (https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv)
https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.names (https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.names)
Previsão da especie de flor dadas medidas das flores.

Imports

In [2]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix, classification_report
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.utils.data import random_split
from torch import Tensor
from torch.nn import Linear
from torch.nn import ReLU
from torch.nn import Softmax
from torch.nn import Module
from torch.optim import SGD, Adam
from torch.nn import CrossEntropyLoss
from torch.nn.init import kaiming_uniform_
from torch.nn.init import xavier_uniform_
```

In [3]:
```
# Constants

#path = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master
/iris.csv'
PATH = 'iris.csv'

# não estão a ser utilizados para já
device = torch.device("cpu") #torch.device("cuda" if torch.cuda.is_a
vailable() else "cpu")
#model.to(device)

EPOCHS = 50
BATCH_SIZE = 32
LEARNING_RATE = 0.01
```

1. Preparar os Dados

In [4]:
```
# definição classe para o dataset
class CSVDataset(Dataset):
    # ler o dataset
    def __init__(self, path):
        # ler o ficheiro csv para um dataframe
        df = pd.read_csv(path, header=None)
        # separar os inputs e os outputs
        self.X = df.values[:, :-1]
        self.y = df.values[:, -1]
        # garantir que os inputs sejam floats
        self.X = self.X.astype('float32')
        # fazer o encoding dos outputs (label) e garantir que sejam
floats
        self.y = LabelEncoder().fit_transform(self.y) # faz o fit e
transforma no self.y o 'Iris-setosa','Iris-versicolor'  e 'Iris-virg
inica' em 0, 1,2


    # número de casos no dataset
    def __len__(self):
        return len(self.X)

    # retornar um caso
    def __getitem__(self, idx):
        return [self.X[idx], self.y[idx]]

    # retornar índices para casos de treino e casos de teste
    def get_splits(self, n_test=0.33):
        # calcular tamanho para o split
        test_size = round(n_test * len(self.X))
        train_size = len(self.X) - test_size
        # calcular o split do houldout
        return random_split(self, [train_size, test_size])#, generat
or=torch.Generator().manual_seed(42))

# preparar o dataset
def prepare_data(path):
    # criar uma instância do dataset
    dataset = CSVDataset(path)
    # calcular o split
    train, test = dataset.get_splits()
    # preparar data loaders
    train_dl = DataLoader(train, batch_size=32, shuffle=True) #32 le
n(train)
    test_dl = DataLoader(test, batch_size=1024, shuffle=False)
    train_dl_all = DataLoader(train, batch_size=len(train), shuffle=
False)
    test_dl_all = DataLoader(test, batch_size=len(test), shuffle=Fal
se)
    return train_dl, test_dl, train_dl_all, test_dl_all

# preparar os dados
train_dl, test_dl,  train_dl_all, test_dl_all = prepare_data(PATH)
```

2. Definir o Modelo

In [6]:

```
from torchinfo import summary

# Definição classe para o modelo
class MLP(Module):
    # definir elementos do modelo
    def __init__(self, n_inputs):
        super(MLP, self).__init__()
        # input para a primeira camada
        self.hidden1 = Linear(n_inputs, 10)
        kaiming_uniform_(self.hidden1.weight, nonlinearity='relu') #
He initialization
        self.act1 = ReLU()
        # segunda camada
        self.hidden2 = Linear(10, 8)
        kaiming_uniform_(self.hidden2.weight, nonlinearity='relu')
        self.act2 = ReLU()
        # terceira camada e output
        self.hidden3 = Linear(8, 3) # um nodo para cada class
        xavier_uniform_(self.hidden3.weight) # Glorot initialization
        # função de ativação
        self.act3 = Softmax(dim=1) # softmax visto ser multiclass

    # sequência de propagação do input
    def forward(self, X):
        # input para a primeira camada
        X = self.hidden1(X)
        X = self.act1(X)
         # segunda camada
        X = self.hidden2(X)
        X = self.act2(X)
        # terceira camada e output
        X = self.hidden3(X)
        X = self.act3(X)
        return X

# definir a rede neuronal
model = MLP(4)
# visualizar a rede
print(summary(model, input_size=(BATCH_SIZE, 4), verbose=0)) #verbos
e=2 Show weight and bias layers in full detail
model.to(device)
```

```
==========================================================================
========================
Layer (type:depth-idx)                   Output Shape
Param #
==========================================================================
========================
├─Linear: 1-1                            [32, 10]
50
├─ReLU: 1-2                              [32, 10]
--
├─Linear: 1-3                            [32, 8]
88
├─ReLU: 1-4                              [32, 8]
--
├─Linear: 1-5                            [32, 3]
27
├─Softmax: 1-6                           [32, 3]
--
==========================================================================
========================
Total params: 165
Trainable params: 165
Non-trainable params: 0
Total mult-adds (M): 0.00
==========================================================================
========================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.01
Params size (MB): 0.00
Estimated Total Size (MB): 0.01
==========================================================================
========================
```

Out[6]:
```
MLP(
  (hidden1): Linear(in_features=4, out_features=10, bias=True)
  (act1): ReLU()
  (hidden2): Linear(in_features=10, out_features=8, bias=True)
  (act2): ReLU()
  (hidden3): Linear(in_features=8, out_features=3, bias=True)
  (act3): Softmax(dim=1)
)
```

3. Treinar o Modelo

In [7]:
```
# treino do modelo
def train_model(train_dl, model):
    # definir o loss e a função de otimização
    criterion = CrossEntropyLoss() # neste caso implementa a sparse_
categorical_crossentropy
    #nn.CrossEntropyLoss accepts ground truth labels directly as int
egers
    #in [0, N_CLASSES[ (no need to onehot encode the labels)
    optimizer = SGD(model.parameters(), lr=LEARNING_RATE, momentum=
0.9) # stochastic gradient descent
    # iterar as epochs
    for epoch in range(EPOCHS):
        # iterar as batchs
        for i, (inputs, targets) in enumerate(train_dl): # backpropa
gation
            # inicializar os gradientes
            optimizer.zero_grad() #coloca os gradientes de todos os
parâmetros a zero
            # calcular o output do modelo
            yprev = model(inputs)
            # calcular o loss
            loss = criterion(yprev, targets)
            # atribuição alteraçoes "In the backward pass we receive
a Tensor containing the gradient of the loss
            # with respect to the output, and we need to compute the
gradient of the loss with respect to the input.
            loss.backward()
            # update pesos do modelo
            optimizer.step()

# treinar o modelo
train_model(train_dl, model)
```

4. Avaliar o Modelo

In [8]:
```
# Avaliar o modelo
def evaluate_model(test_dl, model):
    predictions = list()
    actual_values = list()
    for i, (inputs, labels) in enumerate(test_dl):
        # avaliar o modelo com os casos de teste
        yprev = model(inputs)
        # retirar o array numpy
        yprev = yprev.detach().numpy()
        actual = labels.numpy()
        # converter para a class dos labels
        yprev = np.argmax(yprev, axis=1)
        # reshape for stacking
        actual = actual.reshape((len(actual), 1))
        yprev = yprev.reshape((len(yprev), 1))
        # guardar
        predictions.append(yprev)
        actual_values.append(actual)
        break
    predictions, actual_values = np.vstack(predictions), np.vstack(a
ctual_values)
    return predictions, actual_values

def display_confusion_matrix(cm):
    plt.figure(figsize = (16,8))
    sns.heatmap(cm,annot=True,xticklabels=['Iris-setosa','Iris-versi
color','Iris-virginica'],yticklabels=['Iris-setosa','Iris-versicolor
','Iris-virginica'], annot_kws={"size": 12}, fmt='g', linewidths=.5)

    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()

# avaliar o modelo
predictions, actual_values = evaluate_model(test_dl, model)

acertou=0
falhou = 0
for r,p in zip(actual_values, predictions):
    print(f'real:{r} previsão:{p}')
    if r==p: acertou+=1
    else: falhou+=1

# calcular a accuracy
acc = accuracy_score(actual_values, predictions)
print(f'Accuracy: {acc:0.3f}\n')
print(f'acertou:{acertou} falhou:{falhou}')
```

```
real:[2] previsão:[1]
real:[2] previsão:[1]
real:[0] previsão:[0]
real:[0] previsão:[0]
real:[2] previsão:[1]
real:[0] previsão:[0]
real:[1] previsão:[1]
real:[0] previsão:[0]
real:[1] previsão:[1]
real:[0] previsão:[0]
real:[0] previsão:[0]
real:[2] previsão:[1]
real:[2] previsão:[1]
real:[1] previsão:[1]
real:[2] previsão:[1]
real:[0] previsão:[0]
real:[1] previsão:[1]
real:[0] previsão:[0]
real:[2] previsão:[1]
real:[2] previsão:[1]
real:[0] previsão:[0]
real:[2] previsão:[1]
real:[2] previsão:[1]
real:[2] previsão:[1]
real:[0] previsão:[0]
real:[1] previsão:[1]
real:[2] previsão:[1]
real:[0] previsão:[0]
real:[2] previsão:[1]
real:[1] previsão:[1]
real:[2] previsão:[1]
real:[0] previsão:[0]
real:[0] previsão:[0]
real:[2] previsão:[1]
real:[0] previsão:[0]
real:[1] previsão:[1]
real:[2] previsão:[1]
real:[2] previsão:[1]
real:[1] previsão:[1]
real:[1] previsão:[1]
real:[2] previsão:[1]
real:[2] previsão:[1]
real:[1] previsão:[1]
real:[1] previsão:[1]
real:[2] previsão:[1]
real:[1] previsão:[1]
real:[1] previsão:[1]
real:[2] previsão:[1]
real:[1] previsão:[1]
real:[2] previsão:[1]
Accuracy: 0.560

acertou:28 falhou:22
```

5. Usar o Modelo

In [9]:
```
# fazer uma previsão utilizando um caso
def predict(row, model):
    # converter row para tensor
    row = Tensor([row])
    # fazer a previsão
    yprev = model(row)
    # retirar o array numpy
    yprev = yprev.detach().numpy()
    return yprev

# fazer uma unica previsão (classe esperada=1)
row = [5.1,3.5,1.4,0.2]
yprev = predict(row, model)
print('Predicted: %s (class=%d)' % (yprev, np.argmax(yprev)))
```

```
Predicted: [[9.9767727e-01 2.1665678e-03 1.5611171e-04]] (class=0)
```

In [ ]: