Module 29 : Feed forward Networks with Pytorch

```
>> import numpy as np
>> import math
>> import matplotlib.pyplot as plt
>> import matplotlib.colors
>> import pandas as pd.
>> from sklearn.model_selection import train_test_split
>> from sklearn.metrics import accuracy_score , mean_squared_error,
                                               log_loss
>> from tqdm import tqdm_notebook
>> import seaborn as sns.
>> import time
>> from IPython.display import HTML
>> import warnings
warnings.filterwarnings('ignore')

from sklearn.preprocessing import OneHotEncoder
from sklearn.datasets import make_blobs
import torch
```

`torch.manual_seed(0)` → creating a seed at the start of the session

```
my_cmap = matplotlib.colors.LinearSegmentedColormap.
                  from_list("", ["red", "yellow",
                           "green"])
```

*) Generating Dataset

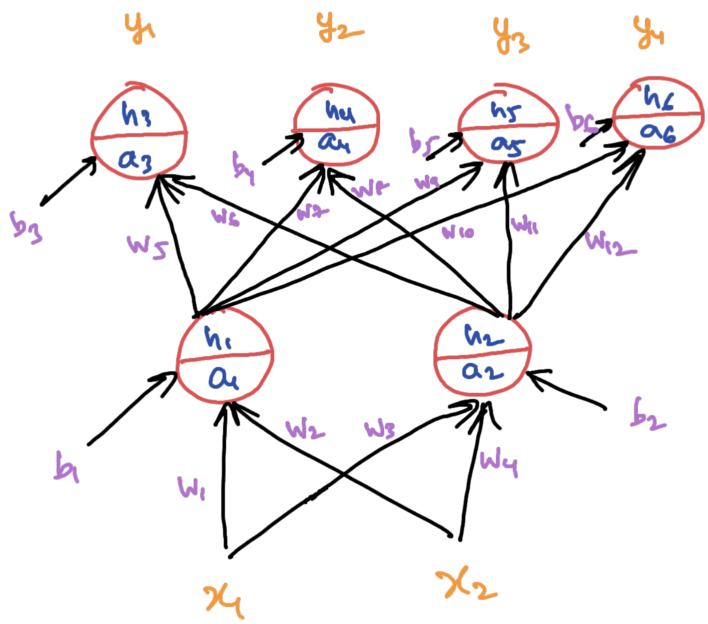
```
>> data , labels = make_blobs (n_samples = 1000,
                               centers = 4 ,
                               n_features = 2 ,
                               random_state = 0)

>> plt . scatter ( data [ : , 0 ], data [ : , 1 ] , c = labels,
                                           cmap = my_cmap )

>> plt . show ( )
```

(Train - test split)

```
>> X_train , X_val , Y_train , Y_val = train_test_split (data ,
                                labels , stratify = labels , random_state = 0)
```



b_3	w_5	w_6	b_4	w_7	w_8	b_5	w_9	w_{10}	b_6	w_{11}	w_{12}
0	0	0	b_1	w_1	w_2	b_2	w_3	w_4	0	0	0

Using torch tensors & autograd

mapping dataset to torch. Tensor

```
X_train, Y_train, X_test, Y_test = map(torch.tensor,
                                        (X_train,
                                         Y_train,
                                         X_val,
                                         Y_val))
```

no. of samples.

```
def model(x):
                                                        # (N, 2) * (2, 2)
    a1 = torch.matmul(x, weights1) + bias1              → (N, 2)
    h1 = a1.sigmoid()  # (N, 2)                         # (N, 2) * (2, 4)
    a2 = torch.matmul(h1, weights2) + bias2             → (N, 4)
    h2 = a2.exp() / a2.exp().sum(-1).unsqueeze(-1)

    return h2  # (N, 4)
```

dimension to sum about ↓ -1 refers to the last index

converts a tensor of dim N to (N,1)

```
a = torch.rand(2, 4)
print(a)
print(a.exp())
# print(a.exp().sum(0))        along cols.
print(a.exp().sum(-1))         along rows → N size tensor
print(a.exp().sum(-1).unsqueeze(-1))
print(a.exp() / a.exp().sum(-1).unsqueeze(-1))
```

size of tensor part of the operation should match at the non-singleton dimension.

```
tensor([[0.5936, 0.4158, 0.4177, 0.2711],
        [0.6923, 0.2038, 0.6833, 0.7529]])
tensor([[1.8105, 1.5155, 1.5185, 1.3114],
        [1.9983, 1.2261, 1.9804, 2.1231]])
torch.Size([2, 4])
tensor([6.1559, 7.3278])
torch.Size([2])
tensor([[6.1559],
        [7.3278]])
torch.Size([2, 1])
tensor([[0.2941, 0.2462, 0.2467, 0.2130],
        [0.2727, 0.1673, 0.2703, 0.2897]])
```

→ shape

→ shape

→ shape

2, 4

y_hat = torch.tensor([[0.1, 0.2, 0.3, 0.4],
[0.8, 0.1, 0.05, 0.05]])

y = torch.tensor([2, 0])

y_hat[range(y_hat.shape[0]), y]

(-y_hat[range(y_hat.shape[0]), y].log()).mean()
.item()

↓

cross entropy loss

torch.argmax(y_hat,
dim = 1)

↓

returns the index corresponding
to the element with max
value about the given dim.

torch.argmax(y_hat, dim) == y

```
x = torch.rand(5,4,3)
print(x)
x[[0,2],[0,2],[0,1]]
```

```
tensor([[[0.6808, 0.5633, 0.4963],
         [0.4012, 0.5627, 0.3858],
         [0.4965, 0.5638, 0.1089],
         [0.2379, 0.9037, 0.0942]],

        [[0.4641, 0.9946, 0.6806],
         [0.5142, 0.0667, 0.7477],
         [0.1439, 0.3581, 0.3322],
         [0.4260, 0.5055, 0.9124]],

        [[0.5624, 0.9478, 0.8059],
         [0.1839, 0.7243, 0.1466],
         [0.2881, 0.6471, 0.6651],
         [0.8751, 0.3390, 0.5008]],

        [[0.7574, 0.0165, 0.8615],
         [0.0865, 0.5069, 0.4150],
         [0.2367, 0.5661, 0.9135],
         [0.3538, 0.2032, 0.3151]],

        [[0.0044, 0.7257, 0.2599],
         [0.1663, 0.2119, 0.7875],
         [0.7648, 0.8838, 0.6814],
         [0.3330, 0.3603, 0.6477]]])
tensor([0.6808, 0.6471])
```

```
def loss_fn(y_hat, y):
    return -(y_hat[range(y_hat.shape(0)), y])

def accuracy(y_hat, y):
    pred = torch.argmax(y_hat, dim=1)
    return (pred==y).float().mean()
```

```
# Defining weights used in the model

torch.manual_seed(0)
weights1 = torch.randn(2, 2) / math.sqrt(2)
```

no. of i/ps

xavier initialization

```
weights1.requires_grad = True
bias1 = torch.zeros(2, requires_grad = True)

weights2 = torch.randn(2, 4) / math.sqrt(2)
weights2.requires_grad = True
bias2 = torch.zeros(4, requires_grad = True)

learning_rate = 0.2
epochs = 10000

X_train = X_train.float()
Y_train = Y_train.long()
```

→ being used as index, hence needs to be integer

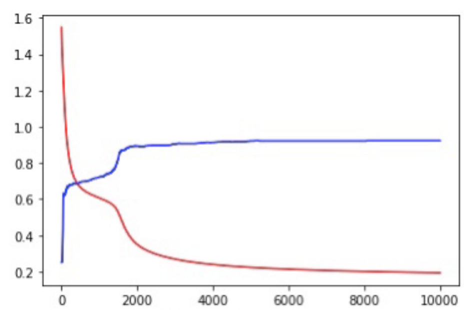
```
loss_arr = []
acc_arr = []
```

for book keeping purposes

```
for epoch in range(epochs):
    y_hat = model(x_train)
    loss = loss_fn(y_hat, y_train)
    loss.backward()
    loss_arr.append(loss.item())
    acc_arr.append(accuracy(y_hat, y_train))

    with torch.no_grad():
        weights1 -= weights1.grad() * learning_rate
        bias1 -= bias1.grad() * learning_rate
        weights2 -= weights2.grad() * learning_rate
        bias2 = bias2.grad() * learning_rate
        weights1.grad.zero()
        bias1.grad.zero()
        weights2.grad.zero()
        bias2.grad.zero()
```

→ only value modification, no structural changes




```
Loss before training 1.5456441640853882
Loss after training 0.19288592040538788
CPU times: user 10.1 s, sys: 346 ms, total: 10.4 s
Wall time: 10.6 s
```

```
plt.plot(loss_arr, 'r-')
plt.plot(acc_arr, 'b-')
plt.show()

print('Loss before training', loss_arr[0])
```

→ Using NN. functional (allows the use of std. fns req. in DL, like cross entropy)

import torch.functional.nn as F

everything else remains the same except the use of

F.cross_entropy (y_hat, Y_train) in place of loss fn.

→ Using NN.Parameter

import torch.nn as nn

↓ writing a class for our network

```
class firstNetwork (nn.Module):

    def __init__ (self):

        super().__init__()

        torch.manual_seed(0)
        self.weights1 = nn.Parameter (torch.randn(2,2) /
                                          math.sqrt(2))

        self.bias1 = nn.Parameter (torch.zeros(2))

        self.weights2 = nn.Parameter (torch.randn(2,4) /
                                          math.sqrt(2))

        self.bias2 = nn.Parameter (torch.zeros(4))

    def forward(self):

        a1 = torch.matmul (x, self.weights1) + self.bias1
        h1 = a1.sigmoid()
        a2 = torch.matmul (h1, self.weights2) + self.bias2
        h2 = a2.exp()/ a2.exp().sum(-1).unsqueeze(-1)
```

no need to explicitly tell requires_grad

```
        return h2


def fit( epochs = 1000, learning_rate = 1) :

    loss_arr = []

    acc_arr = []

    for epoch in range(epochs):
        y_hat = fn(x_train)
        loss = F.cross_entropy(y_hat, Y_train)
        loss_arr.append(loss.item())
        acc_arr.append(accuracy(Y_hat, Y_train))

        loss.backward()
        with torch.no_grad() :
            for param in fn.parameters() :
                param -= learning_rate * param.grad
            fn.zero_grad()
```

model object in global scope

will execute forward

handles all parameter updation on its own

nn.Parameter provides a shorthand to reference the parameters of a network.

The class pattern allows for a separation of concerns between the model definition, inference & fit fn.

```
fn = FirstNetwork()
fit()
```

defined in global scope

→ using NN.Linear & optim

abstracts the process of having to separately define weight matrices & biases. by just specifying the no. of input units & no. of output units.

```
def firstNetwork_v1(nn.Module):

    def __init__(self):
        super().__init__()
        torch.manual_seed(0)
        self.lin1 = nn.Linear(2, 2)
        self.lin2 = nn.Linear(2, 4)

    def forward(self, x):
        a1 = self.lin1(x)
        h1 = a1.sigmoid()
        a2 = self.lin2(h1)
        h2 = a2.exp()/a2.exp().sum(-1).unsqueeze(-1)
```

the way the weights are getting init. is diff.

will automatically internally have weights & biases

→ input to first linear layer

→ input to second input layer

*) from torch import optim → abstract away the need to write loops for parameter updation

```
def fit_v1(epochs=1000, learning_rate=1):
    loss_arr = []
    acc_arr = []
    opt = optim.SGD(fn.parameters(), lr=learning_rate)
```

defining opt hyperparameters & algo.

parameters to optimize

learning rate to be used for optim

```
for epoch in range(epochs):
    y_hat = fn(X_train)
    loss = f.cross_entropy(y_hat, Y_train)
    loss_arr.append(loss.item())
    acc_arr.append(y_hat, Y_train)

    loss.backward()   # backpropagation
    opt.step()        # update step
    opt.zero_grad()   # gradient reset
```

→ Using NN.Sequential → allows us to define a sequence through which our data goes through.

```
def FirstNetwork_v1(nn.Module):

    def __init__(self):
        super().__init__()
        torch.manual_seed(0)
        self.net = nn.sequential(
                nn.Linear(2, 2),   → highlights a linear transformation from 2, 2
                nn.sigmoid(),      ← activation fn. applied after the transformation
                nn.Linear(2, 4),   → highlights a linear transformation from 2, 4
                nn.softmax()
        )

    def forward(self, X):
        return self.net(X)
```

due to the way GPUs are organised

↑

better performance if 2^x

↑

(2, 1024)

exercise : eval fn. which accepts test data & returns test accuracy.

```
fn = firstNetwork_v2()
loss_fn = F.cross_entropy()
opt = optim.SGD(fn.parameters(), lr=0.2)
fit_v2(X_train, Y_train, fn, opt)
```

→ Invoking the network

*) migrating to CUDA

to operate the data on any device, we need to push the data to it using .to(device). For example

```
X_train = X_train.to(device)
```

↓

casts the data to the device sent in as argument.

```
device = torch.device("cpu") or
         torch.device("cuda")
```

Pytorch

↓

vectorization across i/ps, 1000 times faster than native python

```
X_train = X_train.to(device)
Y_train = Y_train.to(device)
fn = FirstNetwork_v2()
fn.to(device)
tic = time.time()
print(fit_v2(X_train, Y_train, fn, opt, loss_fn))
```

```
toc = time.time()
print (toc - tic )
```

→ exercises

i) Try out a deeper neural network, ex. 2 hidden layers
ii) different parameters in the optimizer, in optim.SGD
iii) different optimization methods supported in optim
iv) different initialisation methods, supported in nn.init
v) tuning of hyperparameters to improve model accuracy.

optim.SGD (momentum
dampening
weight_decay
nesterov)

~~RMSProp~~
~~ASGD~~
Adamax
~~Adam~~
~~Adagrad~~
~~Adadelta~~

state_dict → returns the state of the optimizer as a dict
load_state_dict → loads the optimiser_state

---

different initialisation methods : nn.init

nn.init.uniform_ (tensor, a, b) $U(a, b)$
normal_ (tensor, mean, std)
xavier_uniform_ (tensor, gain)
xavier_normal_ (tensor, gain)
kaiming_uniform_(tensor, a, mode, nonlinearity)

kaiming_normal_(tensor, a, mode, nonlinearity)

*) initialising weights for sequential defined models :

defining another fn.

```
def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)
        nn.init.constant_(m.bias, 0)
```