TP NN PART 2, 3 Emilio RAIMOND

2. Implémentation d'un réseau de neurones à deux couches « from scratch »

Consigne 10 :

On initialise les paramètres aléatoirement en respectant les dimensions de Q :

```
def init_randoms_params(input_dim=784, hidden_dim=10, output_dim=3):
    W1 = np.random.randn(input_dim,hidden_dim)*1e-2 #W1 est de dimension DxQ (784x10)
    b1=np.random.randn(hidden_dim,)*1e-2 #b1 est de dimension Q (10)
    W2 = np.random.randn(hidden_dim, output_dim)*1e-2 #W2 est de dimension Qx3
    b2 = np.random.randn(output_dim,)*1e-2 #b2 est de dimension 3
    return W1, b1, W2, b2
```

Question 12 :
On note notre nombre de paramètres nParam.
Nparam28 = (W1 + b1) + (W2 + b2) = (784*10+10) + (10*3+3) = 7883

Si nos images faisaient du 256x256, on n'aurait pas N=28*28 = 784 comme actuellement mais N=265*265

Nparam256 = 256*10+10) + (10*3+3) = 655 403.

Consigne 11 :

On suit le graphe de calcul donné dans l'énoncé pour rajouter le tanh et le Z2.

```
def predict_two_layer_nn(X, parameters):
    W1, b1, W2, b2 = parameters
    Z1 = np.dot(X, W1) + b1 #Z1 est de dimension NxQ
    a1 = np.tanh(Z1) #a1 est de dimension NxQ
    Z2 = np.dot(a1, W2) + b2 #Z2 est de dimension Nx3
    return stable_softmax(Z2) #Yhat est de dimension Nx3
```

Consigne 12 :

On suit ligne à ligne les dérivées partielles données tout en s'assurant la cohérence des dimensions en transposant les bonnes matrices.

```python
def partial_derivative_two_layers_nn(x, y, y_hat, parameters):
    W1, b1, W2, b2 = parameters

    delta3 = y_hat - y
    z1 = np.dot(x, W1) + b1
    a1 = np.tanh(z1)
    da1 = dtanh(z1) #dérivée de a1
    delta2=np.multiply(da1,np.dot(delta3,W2.T))
    dw1 = np.dot(x.T,delta2)
    db1 = np.sum(delta2,axis=0)
    dw2 = np.dot(a1.T,delta3)
    db2 = np.sum(delta3,axis=0)

    return dw1, db1, dw2, db2
```

Consigne 13 :

Je n'ai fait ici que rajouter des paramètres pour créer un graph des loss train et test et j'ai aussi rajouté la modification des nouveaux paramètres par leurs dérivées partielles respectives.

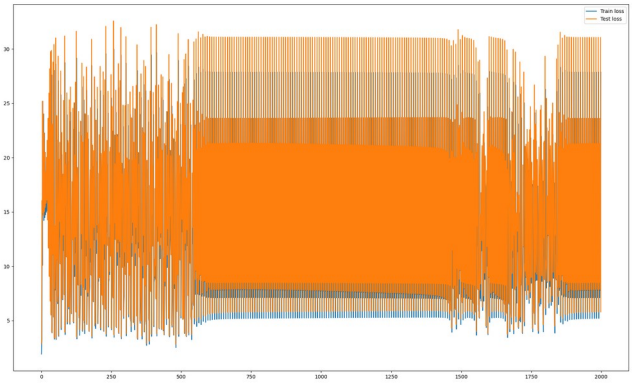
```python
def gradient_descent_two_layers_nn(x_train, y_train, x_test, y_test, hidden_dim=10, num_passes=5000, epsilon=1e-5, x_graph=None, y_graph=None, y_test_graph=None):
    W1, b1, W2, b2 = init_randoms_params(input_dim=x_train.shape[1], hidden_dim=hidden_dim, output_dim=y_train.shape[1])
    for i in range(num_passes):
        y_hat = predict_two_layer_nn(x_train, (W1, b1, W2, b2))
        dw1, db1, dw2, db2 = partial_derivative_two_layers_nn(x_train, y_train, y_hat, (W1, b1, W2, b2))

        W1 = W1 - epsilon*dw1
        b1 = b1 - epsilon*db1
        W2 = W2 - epsilon*dw2
        b2 = b2 - epsilon*db2

        y_graph.append(evaluate_loss(y_train, y_hat))
        y_test_graph.append(evaluate_loss(y_test, predict_two_layer_nn(x_test, (W1, b1, W2, b2))))
        if(i%100==0):
            print("Loss à l'itération ",i," : ",evaluate_loss(y_train, y_hat))
    return W1, b1, W2, b2
```

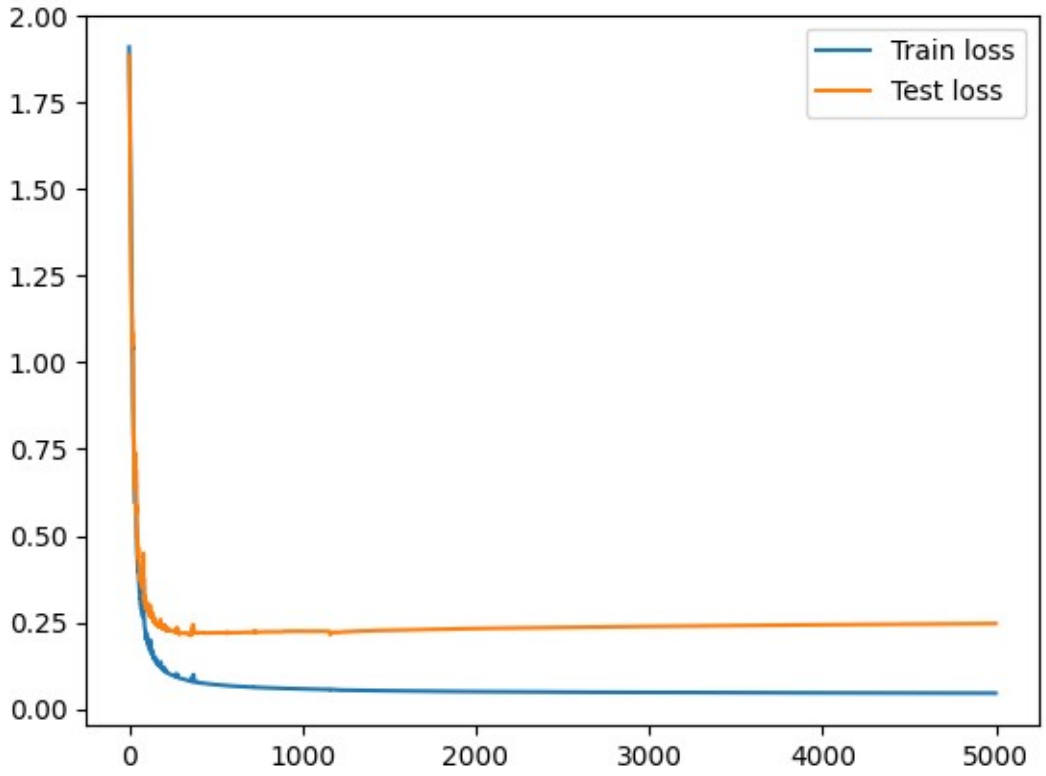
Consigne 14 :

On lance une première fois la descente avec l'epsilon à 1e-3 et on observe un effet yoyo des parmaètres :



Precision pour xTrain: 42.8 %
Precision pour xTest: 36.2 %

On met donc le epsilon est réglé 1e-5 :



Ce qui nous donne un cout qui converge vers ~0.05648 et des précisions de :

Precision pour xTrain : 99.16666666666667
Precision pour xTest: 96.6

Augmenter le nombre de neurones dans la couche cachée n'augmente rien si ce n'est le temps de calcul.

3. En utilisant torch

On reprend le code du tp d'avant.

Consigne 15 :

On modifie la fonction init_random_param de la manière suivante :

```
def init_randoms_params(input_dim=784, hidden_dim=10, output_dim=3):
    W1 = np.random.randn(input_dim,hidden_dim)*1e-3 #W1 est de dimension DxQ (784x10)
    b1=np.random.randn(1,hidden_dim)*1e-3 #b1 est de dimension Q (10)
    W2 = np.random.randn(hidden_dim, output_dim) #W2 est de dimension Qx3
    b2 = np.random.randn(1,output_dim) #b2 est de dimension 3

    #Puis on convertit les arrays numpy en tensors torch
    W1 = torch.tensor(W1,dtype=dtype ,requires_grad=True)
    b1 = torch.tensor(b1,dtype=dtype, requires_grad=True)
    W2 = torch.tensor(W2, dtype=dtype, requires_grad=True)
    b2 = torch.tensor(b2, dtype=dtype, requires_grad=True)
    return W1, b1, W2, b2
```

Ce qui nous permet de désormais manipuler des tensors pytorch.

Consigne 16 et 17:

On modifie aussi les opérations car on n'a plus des numpy.arrays :

```
def predict_two_layer_nn(x, params):
    W1, b1, W2, b2 = params
    Z1 = torch.matmul(x, W1) + b1
    a1 = torch.tanh(Z1)
    Z2 = torch.matmul(a1, W2) + b2
    yhat = torch.softmax(Z2,1)
    return yhat

#y et yhat sont des tensors torch
def evaluate_loss_torch(y, y_hat):
    return -(1/y.shape[0])*torch.sum(y*torch.log(y_hat+1e-16)+(1-y)*torch.log((1-y_hat)+1e-16))
```

Consigne 18 :

Comme on s'est débarrassé de l'écriture des dérivées partielles, on doit demander à pytorch de faire la rétropropagation.
On crée pour cela des variables loss (sur le train évidemment). Le calcul des gradients se fera donc par le biais de la méthode .backward(), on modifie ensuite la valeur des paramètres à l'aide de ce calcul. On réinitialise ensuite les gradients pour ne pas les accumuler.

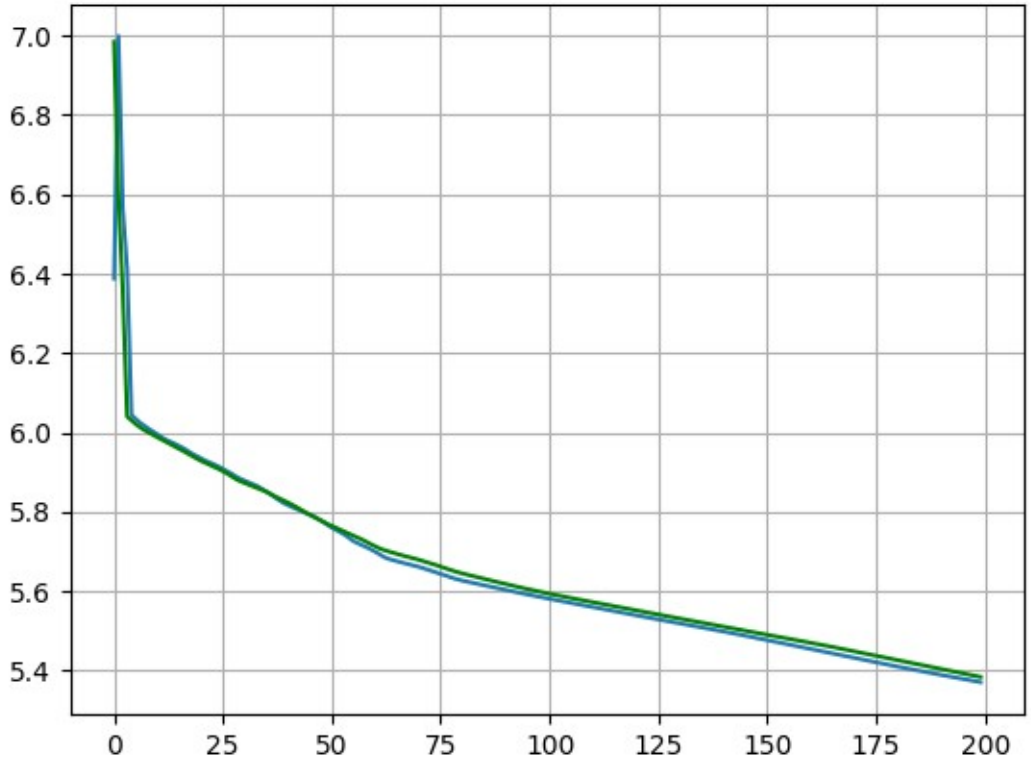
```
def gradient_descent_torch(x_train, y_train, x_test, y_test, parameters, model=None,
    W1, b1, W2, b2 = parameters
    for i in range(num_passes):
        y_hat=predict_two_layer_nn(x_train, (W1, b1, W2, b2))
        loss=evaluate_loss_torch(y_train, y_hat)
        loss.backward() #calcul des gradients

        #calcul des gradients pour les paramètres
        W1.data -= epsilon * W1.grad.data
        b1.data -= epsilon * b1.grad.data
        W2.data -= epsilon * W2.grad.data
        b2.data -= epsilon * b2.grad.data

        W1.grad.data.zero_()
        b1.grad.data.zero_()
        W2.grad.data.zero_()
        b2.grad.data.zero_()

        y_graph.append(loss.item())
```

Quand on teste avec 200 itérations, le résultat est complètement dépendant de l'init random des paramètres (ce qu'on ne veut pas). 200 de num_passes est donc insuffisant.



Comme le prouve la précision digne d'un lancé de pièce ici :

```
Loss à l'itération  0  :  tensor(6.3880, grad_fn=<MulBackward0>)
Loss à l'itération  100  :  tensor(5.5812, grad_fn=<MulBackward0>
Precision pour train :  33.333333333333333
Precision pour test :  33.33333333333333
```

On augmente le nombres de passes et on utilise torh.relu au lieu de la tangeant hyperbolique :

```
Loss à l'itération  9400  :  tensor(0.0002, grad_fn=<MulBackward0>)
Loss à l'itération  9500  :  tensor(0.0002, grad_fn=<MulBackward0>)
Loss à l'itération  9600  :  tensor(0.0002, grad_fn=<MulBackward0>)
Loss à l'itération  9700  :  tensor(0.0002, grad_fn=<MulBackward0>)
Loss à l'itération  9800  :  tensor(0.0002, grad_fn=<MulBackward0>)
Loss à l'itération  9900  :  tensor(0.0002, grad_fn=<MulBackward0>)
Precision pour train :  100.0
Precision pour test :  97.33333333333334
```

Les losses atteignent un niveau largement plus désirable tout comme la précision qui augmente.

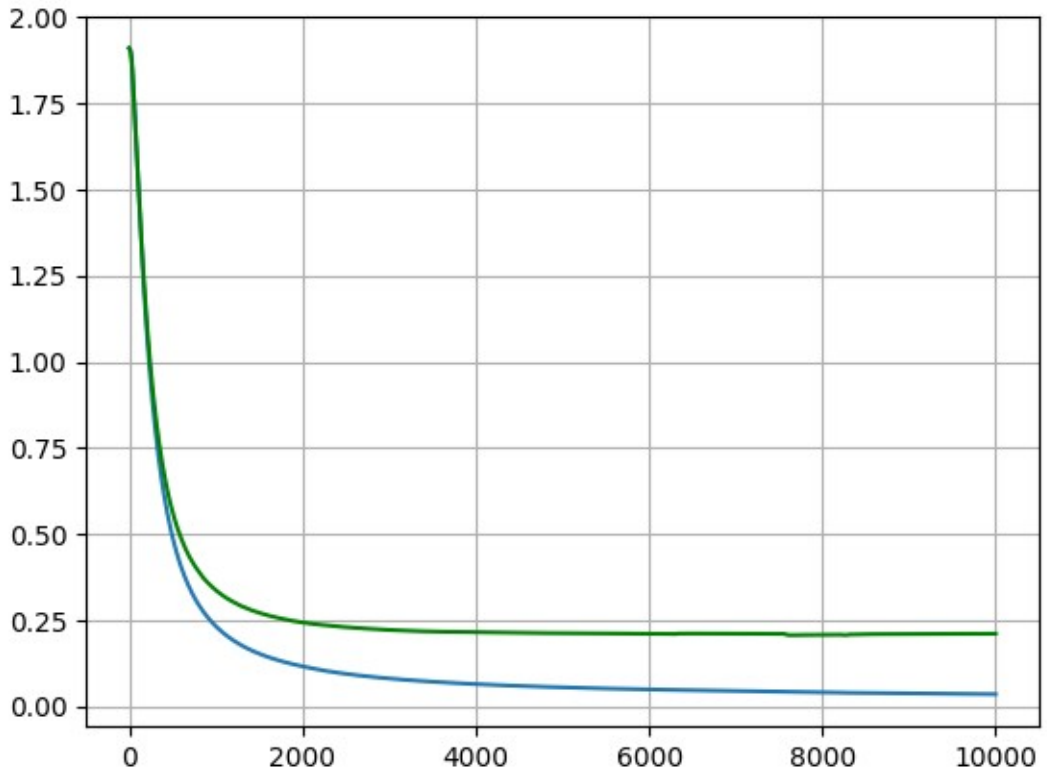
Consigne 19 :

On remplace notre code par l'optimizer SGD :

```
def gradient_descent_torch(x_train, y_train, x_test, y_test, parameters, mod
    W1, b1, W2, b2 = parameters
    optimizer=torch.optim.SGD(params=parameters, lr=epsilon, momentum=0.9)

    for i in range(num_passes):
        #on utilise le modèle
        y_hat=predict_two_layer_nn(x_train, (W1, b1, W2, b2))
        loss=evaluate_loss_torch(y_train, y_hat)

        loss.backward() #calcul des gradients
        optimizer.step() #mise à jour des paramètres
        optimizer.zero_grad() #réinitialisation des gradients
```



« Precision pour train : 99.6, Precision pour test : 96.86666666666667 »
(ces tests ont été fait avec tanh)

Consigne 20 :
On écrit la classe MLP héritant de torch.nn.Module, on crée c1 et c2 nos deux couches dans l'__init__ de type torch.Linear et on fait notre fonction forward. On pense

```python
class MLP(torch.nn.Module):
    def __init__(self, input_dim=784, hidden_dim=100, output_dim=3):
        super(MLP, self).__init__()

        self.c1 = torch.nn.Linear(input_dim, hidden_dim)
        self.c1.weight.data.normal_(0, 1e-3) #initialisation des poid
        self.c1.bias.data.normal_(0, 1e-3)
        self.c2 = torch.nn.Linear(hidden_dim, output_dim)
        self.c2.weight.data.normal_(0, 1e-3)
        self.c2.bias.data.normal_(0, 1e-3)

    def forward(self, x):
        x = self.c1(x)
        x = torch.relu(x)
        x = self.c2(x)
        x = torch.softmax(x,1)
        return x
```

On vérifie que la somme des prédictions donne bien 1 et on peut continuer :

```python
#PARTIE 3 : Implémentation d'un réseau de neurones à deux couches avec PyTorch
#on utilise la classe MLP définie plus bas
y_graph = [] #loss sur le train
y_test_graph = [] #loss sur le test
model = MLP()

optimizer = torch.optim.SGD(model.parameters(), lr=1e-4, momentum=0.9)
for i in range(10_000):
    y_hat = model(x_train)
    loss = evaluate_loss_torch(y_train, y_hat)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    y_graph.append(loss.item())
    y_test_graph.append(evaluate_loss_torch(y_test, model(x_test)).data.numpy())
    if(i%100==0):
        print("Loss à l'itération ",i," : ",evaluate_loss_torch(y_train, y_hat))
```

On utilise donc désormais le modèle au lieu de définir les paramètres manuellement.

```python
#PARTIE 3 : Implémentation d'un réseau de neurones à deux couches avec PyTorch
#on utilise la classe MLP définie plus bas
y_graph = [] #loss sur le train
y_test_graph = [] #loss sur le test
model = MLP()

optimizer = torch.optim.SGD(model.parameters(), lr=1e-4, momentum=0.9)
for i in range(10_000):
    y_hat = model(x_train)
    loss = evaluate_loss_torch(y_train, y_hat)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Ce qui nous donne :