TP2

October 10, 2019

1 TP 2: Graphe de calcul, Optimiseur et Module

Ismaël Bonneau & Issam Benamara

Le vrai code avec en particulier les import, chargement des données, etc se trouve dans le fichier tp2.py

1.0.1 Descente de gradient sans optimiseur:

On doit mettre à jour nous même les paramètres.

```
In [ ]: class LinearRegressor(torch.nn.Module):
            def __init__(self, inputSize):
                super(LinearRegressor, self).__init__()
                #self.linear = torch.nn.Linear(inputSize, 1)
                self.W = torch.nn.Parameter(torch.randn(inputSize, 1), requires_grad=True)
                self.b = torch.nn.Parameter(torch.randn(1), requires_grad=True)
            def forward(self, x):
                #y_pred = self.linear(x)
                y_pred = x @ self.W + self.b
                return y_pred
```

```
In [ ]: model = LinearRegressor(6)
        lossfn = torch.nn.MSELoss()

        for e in range(epochs):
            ohplai = [] # pour calculer ensuite la moyenne des loss sur tous les batches
            for x_batch, y_batch in trainloader: # batch
                model.train()
                # forward
                mult = model(x_batch)
                loss = lossfn(mult, y_batch)
                ohplai.append(loss.item())
                loss.backward()
                with torch.no_grad():
                    model.W -= learningRate * model.W.grad
                    model.b -= learningRate * model.b.grad
                model.W.grad.zero_() #remise a zero des gradients
                model.b.grad.zero_()
```

```
            with torch.no_grad():
                # compute validation error
                model.eval()
                arouf = model(torch.from_numpy(X_test))
                loss_arouf = lossfn(arouf, torch.from_numpy(Y_test))

            if (e % 10) == 0:
                print("epoch %d " % e , "train MSE: ", np.array(ohplai).mean(), "val MSE: ", los
```

1.0.2 Descente de gradient avec optimizer (SGD):

Maintenant c'est l'optimizer qui se charge de mettre à jour les paramètres.

```
In [ ]: model = LinearRegressor(6)
        lossfn = torch.nn.MSELoss()
        optimizer = torch.optim.SGD(model.parameters(), lr = learningRate)

        for e in range(epochs):
            for x_batch, y_batch in trainloader:

                model.train()
                # forward
                mult = model(x_batch)
                loss = lossfn(mult, y_batch)
                loss.backward()

                optimizer.step()
                optimizer.zero_grad()
```

1.0.3 Version avec un NN simple à deux couches:

La procédure pour la descente de gradient est la même que précédemment.

Le conteneur pytorch *Sequential* permet d'encapsuler plusieurs modules pytorch, dans un certain ordre. L'avantage: les paramètres de ces modules seront automatiquement enregistrés dans le registre des paramètres pytorch, ce qui n'est pas le cas si on utilise une liste par exemple. Autre avantage: il dispose d'une méthode forward, un seul appel permet d'exécuter en chaine un forward sur tous les modules qu'il contient. Il suffira d'un seul

```
return mon_sequential(x)
```

Sans conteneur Sequential:

```
In [ ]: class Perceval1(torch.nn.Module):
            def __init__(self, inputSize):
                super(Perceval, self).__init__()
                self.linear1 = torch.nn.Linear(inputSize, 16)
                self.linear2 = torch.nn.Linear(16, 1)
                self.activation = torch.nn.Tanh()
            def forward(self, x):
                return self.linear2(self.activation(self.linear1(x)))
```

Avec conteneur Sequential:

```
In [ ]: class Perceval(torch.nn.Module):
            def __init__(self, inputSize):
                super(Perceval, self).__init__()
                self.mlp = torch.nn.Sequential(torch.nn.Linear(inputSize, 16), torch.nn.Tanh(),
            def forward(self, x):
                return self.mlp(x) # un seul forward pour faire linear->tanh->linear!
```