400_backward

November 26, 2021

1 400 - gradient et backward

On peut avoir un réseau de neurones comme un gros sandwitch avec de multiples couches de neurones dans lequel on a envie d'insérer sa propre couche mais pour cela il faut interférer avec le gradient. J'ai repris le tutoriel pytorch: defining new autograd functions. L'exemple suivant Extending Torch. J'aimais bien l'API de la version 0.4 mais je ne la trouve plus sur internet. Elle laissait sans doute un peu trop de liberté.

[1]:
```
import time
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
print("torch", torch.__version__)
from torchvision import datasets, transforms
from tqdm import tqdm
```

```
torch 1.5.0+cpu
```

[2]:
```
%matplotlib inline
```

[3]:
```
BATCH_SIZE = 64
TEST_BATCH_SIZE = 64
DATA_DIR = 'data/'
USE_CUDA = False  # switch to True if you have GPU
N_EPOCHS = 2 # 5
```

[4]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[4]:
```
<IPython.core.display.HTML object>
```

1.1 Couche personnalisée

L'exemple suivant illustre comment définir une couche intermédiaire qui obéit à ses propres règles. Elle doit implémenter les deux méthodes `forward` qui calcule la prédiction pour la couche suivante et `backward` pour le calcul du gradient pour la couche précédente. Il reste la variable `ctx` qui permet de stocker des informations qui persistent jusqu'au calcul du gradient.

[5]:
```
class MyReLU(torch.autograd.Function):

    @staticmethod
    def forward(ctx, input):
        ctx.save_for_backward(input)
        return input.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_output):
        input, = ctx.saved_tensors
        grad_input = grad_output.clone()
        grad_input[input < 0] = 0
        return grad_input
```

1.2 exemple avec MNIST

C'est à ce moment que je choisis pour bifurque sur un autre tutoriel pytorch et MNIST qui commence sur un réseau de neurones pour *MNIST* et pas vraiment *MNIST* puisque c'est une régression avec deux sorties.

[6]:
```
dtype = torch.float
device = torch.device("cpu")
```

[7]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 2)

    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features
```

```
net = Net()
print(net)
```

```
Net(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=2, bias=True)
)
```

1.3 Le gradient

On choisit une entrée aléatoire mais aux mêmes dimensions qu'une image et on calcule la sortie du réseau de neurones.

[8]:
```
input = torch.randn(1, 1, 32, 32)
out = net(input)
print(out)
```

```
tensor([[ 0.0984, -0.0259]], grad_fn=<AddmmBackward>)
```

Pour calculer le gradient, il faut choisir une erreur… On prend le MSELoss même si ça n'a rien à voir avec le problème initial mais on s'en fout.

[9]:
```
criterion = nn.MSELoss()
```

On choisit une entrée aléatoire…

[10]:
```
target = torch.randn(2)
```

On calcule une sortie et on redimensionne la cible de sorte qu'elle est la même dimension que la cible.

[11]:
```
output = net(input)
target = target.view(1, -1)  # make it the same shape as output
output, target
```

[11]:
```
(tensor([[ 0.0984, -0.0259]], grad_fn=<AddmmBackward>),
 tensor([[1.0841, 0.6601]]))
```

Maintenantn on calcule l'erreur de prédiction…

[12]:
```
loss = criterion(output, target)
loss
```

[12]:
```
tensor(0.7211, grad_fn=<MseLossBackward>)
```

Et on calcule enfin le gradient :

[13]:
```
loss.backward()
```

Je m'attendais à trouver le gradient sur la forme d'un beau vecteur mais comme c'est un réseau de neurones, le gradient est stocké dans chacune des couches sous la forme d'un gradient et encore en deux morceaux…

[14]:
```
net.conv1.bias.grad
```

[14]:
```
tensor([-0.0044, -0.0129, -0.0120, -0.0106,  0.0108, -0.0080])
```

[15]:
```
net.conv1.weight.grad
```

[15]:
```
tensor([[[[-1.2407e-02, -1.3151e-02,  1.9494e-03, -1.8397e-02, -8.1399e-03],
          [ 2.2697e-02, -2.1370e-02,  1.3234e-02, -4.5362e-02, -4.7617e-03],
          [-2.6007e-03, -3.2106e-02,  5.0279e-03,  1.6232e-02,  3.3929e-03],
          [-1.4467e-03, -8.2939e-03, -8.2464e-03,  9.6412e-03, -5.6548e-03],
          [-8.7655e-03, -1.0408e-02, -1.1384e-02, -8.2852e-03,  2.1430e-04]]],


        [[[ 1.2725e-02,  1.3035e-05,  4.3161e-03,  2.4578e-02,  7.0311e-03],
          [-2.5651e-02,  2.3921e-02,  5.6768e-03,  4.2782e-02,  9.0033e-03],
          [-3.8339e-03, -2.1918e-02, -1.1344e-02, -1.2316e-04, -1.6823e-02],
          [-4.8012e-03, -2.4580e-02, -1.2668e-02, -2.6195e-02, -4.0072e-04],
          [-4.3689e-03, -2.3476e-02, -3.0547e-03,  1.9178e-02, -1.8830e-03]]],


        [[[-6.5927e-03, -1.7511e-02,  4.2962e-03,  6.9383e-03,  3.4250e-02],
          [ 1.6830e-02, -2.1891e-02,  1.6447e-02, -1.4255e-02,  2.5494e-02],
          [ 3.2126e-02,  2.0147e-02,  2.1486e-03, -2.4806e-02,  1.7293e-02],
          [-1.2384e-02,  1.4965e-03, -4.5062e-03,  2.2755e-02,  3.4465e-02],
          [-1.5342e-02, -8.4329e-03,  1.2289e-02,  6.9204e-03,  2.8706e-03]]],


        [[[ 4.3549e-03, -4.0362e-03,  2.5292e-02, -2.1750e-03, -1.9392e-03],
          [ 1.3265e-02, -3.7641e-03, -1.4633e-02,  1.7212e-03,  4.6102e-03],
          [-3.7710e-02, -4.1038e-03,  1.1128e-02, -2.0951e-03,  4.1262e-03],
          [ 1.2689e-02,  1.7853e-02,  3.4458e-02,  2.0659e-02, -1.1101e-02],
          [-9.8923e-03, -3.2189e-03,  4.4341e-02, -2.2615e-02,  1.1134e-03]]],


        [[[ 2.5995e-02, -3.8009e-02,  3.6237e-02, -5.9109e-03, -3.5588e-02],
          [ 8.6424e-03, -1.1644e-02, -3.4126e-03,  6.3851e-03,  4.1707e-03],
          [ 5.8575e-03, -2.5871e-03,  8.4853e-04, -1.3423e-02, -9.6491e-03],
          [ 9.7071e-03,  1.6865e-02, -1.7805e-02,  8.2747e-03,  3.0687e-02],
          [ 1.7748e-02, -2.6293e-02, -3.7462e-02,  1.8656e-02,  3.5240e-02]]],


        [[[ 2.6818e-02, -9.8710e-03, -1.0695e-02, -4.1982e-03,  2.6078e-02],
          [ 3.7741e-02, -7.4138e-03, -1.7361e-02,  1.5816e-02, -2.2532e-02],
          [-4.3165e-04, -8.4519e-03,  1.3581e-02, -2.4493e-02, -1.9400e-03],
          [-5.5297e-04, -2.7260e-02, -3.1624e-02,  2.8114e-03,  3.4903e-02],
          [ 1.8116e-02, -2.3265e-02, -9.1825e-03,  6.5738e-03,  3.7212e-03]]]])
```

1.4 On change de couche

Non, il n'est pas trois heures du matin et personne ne pleure.

[16]:
```
class Net0(nn.Module):

    def __init__(self):
        super(Net0, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
```

```
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 2)

    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = MyReLU(self.fc1(x))
        x = MyReLU(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net0 = Net0()
print(net0)
```

```
Net0(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=2, bias=True)
)
```

Evidemment, ça ne marche jamais du premier coup. On a beaucoup sous-estimé le temps de préparation des cours avec l'informatique. Il est seulement 9h du soir, l'humilité vient seulement après minuit.

[17]:
```
try:
    output = net0(input)
except Exception as e:
    print(e)
```

```
'MyReLU' object has no attribute 'dim'
```

J'avoue que j'ai tout lu en diagonal pensant que ma science infinie me permettrait de combler mon ignorance. Honnêtement, il est temps de dîner !

1.5 After the bug...

J'ai craqué, je n'ai pas eu le courage de recoder à 2h du matin trying to figure out what's going wrong. J'ai mélangé tout ce que j'ai pu trouvé et il est quasiment 2h du matin le lendemain.

[18]:
```
class MyReLUF(torch.autograd.Function):

    @staticmethod
    def forward(ctx, input):
```

```
        ctx.save_for_backward(input)
        res = input.clamp(min=0)
        return res

    @staticmethod
    def backward(ctx, grad_output):
        input, = ctx.saved_tensors
        grad_input = grad_output.clone()
        grad_input[input < 0] = 0
        return grad_input


class MyReLUNN(nn.Module):

    def __init__(self):
        super(MyReLUNN, self).__init__()

    def forward(self, input):
        return MyReLUF.apply(input)

    def extra_repr(self):
        return 'in_features={}'.format(self.in_features)


class Net2(nn.Module):

    def __init__(self):
        super(Net2, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 2)
        self.fct1 = MyReLUNN()
        self.fct2 = MyReLUNN()

    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = self.fct1(self.fc1(x))
        x = self.fct2(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features
```

```
net2 = Net2()
output = net2(input)
output
```

[18]:
```
tensor([[-0.0333,  0.0842]], grad_fn=<AddmmBackward>)
```

[19]:
```
target = target.view(1, -1)  # make it the same shape as output
output, target
```

[19]:
```
(tensor([[-0.0333,  0.0842]], grad_fn=<AddmmBackward>),
 tensor([[1.0841, 0.6601]]))
```

[20]:
```
loss = criterion(output, target)
loss
```

[20]:
```
tensor(0.7901, grad_fn=<MseLossBackward>)
```

[21]:
```
loss.backward()
```

[22]:
```
net2.conv1.bias.grad
```

[22]:
```
tensor([ 0.0256,  0.0139, -0.0511,  0.0475,  0.0105, -0.0168])
```

[23]:
```
net2.conv1.weight.grad
```

[23]:
```
tensor([[[[ 0.0500, -0.0042, -0.0153, -0.0117, -0.0175],
          [-0.0224, -0.0135,  0.0331,  0.0448, -0.0135],
          [-0.0042,  0.0086,  0.0201,  0.0037, -0.0064],
          [ 0.0003, -0.0192,  0.0464,  0.0070,  0.0185],
          [-0.0335, -0.0337, -0.0440,  0.0414,  0.0430]]],


        [[[-0.0214, -0.0110,  0.0022,  0.0427, -0.0296],
          [ 0.0081,  0.0167, -0.0360,  0.0032,  0.0301],
          [ 0.0081,  0.0092, -0.0012, -0.0012, -0.0228],
          [ 0.0122,  0.0110,  0.0293, -0.0106, -0.0213],
          [ 0.0343,  0.0122,  0.0078, -0.0082, -0.0098]]],


        [[[-0.0188,  0.0203,  0.0413,  0.0228, -0.0316],
          [ 0.0266,  0.0005,  0.0027, -0.0052,  0.0008],
          [-0.0393,  0.0069, -0.0042,  0.0063,  0.0075],
          [-0.0143,  0.0155,  0.0342, -0.0302,  0.0274],
          [-0.0002, -0.0492,  0.0197,  0.0126,  0.0019]]],


        [[[-0.0056,  0.0119, -0.0113,  0.0035, -0.0430],
          [-0.0060, -0.0344,  0.0002, -0.0130, -0.0086],
          [-0.0223, -0.0265,  0.0163,  0.0041,  0.0155],
          [ 0.0044,  0.0188, -0.0009, -0.0505, -0.0140],
          [ 0.0070, -0.0174, -0.0054, -0.0245, -0.0113]]],
```

```
[[[ 0.0062, -0.0078,  0.0022, -0.0145,  0.0271],
  [-0.0030, -0.0137, -0.0048,  0.0448,  0.0055],
  [-0.0153, -0.0045,  0.0147, -0.0100,  0.0114],
  [-0.0079, -0.0011, -0.0150, -0.0102, -0.0001],
  [-0.0138,  0.0115, -0.0236, -0.0010,  0.0092]]],


[[[-0.0366,  0.0114, -0.0274, -0.0286,  0.0167],
  [ 0.0094,  0.0126, -0.0124, -0.0069, -0.0165],
  [-0.0034, -0.0061,  0.0168,  0.0144,  0.0080],
  [ 0.0373, -0.0362, -0.0260, -0.0005, -0.0311],
  [-0.0091,  0.0241,  0.0315, -0.0172,  0.0020]]]])
```

1.6 Ca converge ?

Il ne reste plus qu'à voir si cela converge. On crée une cible un peu alambiquée.

[24]:
```
N = 10000
inputs = torch.randn(N, 1, 1, 32, 32)
targets = torch.zeros(N, 2)
for i in range(N):
    s1 = inputs[i, :, :, :16, :].sum() / 10
    s2 = inputs[i, :, :, 16:, :].sum() / 10
    targets[i, 0] = s1
    targets[i, 1] = s2
```

[25]:
```
targets[:5]
```

[25]:
```
tensor([[ 0.1264, -1.6738],
        [-0.7678,  4.4501],
        [-0.3112, -1.8726],
        [ 2.5569, -0.0294],
        [-3.5128,  2.7586]])
```

[26]:
```
torch.sum(targets, 0)
```

[26]:
```
tensor([314.3907,   2.3097])
```

On choisit l'algorithme d'optimisation et la fonction de coût. Rien d'original dans tout ça.

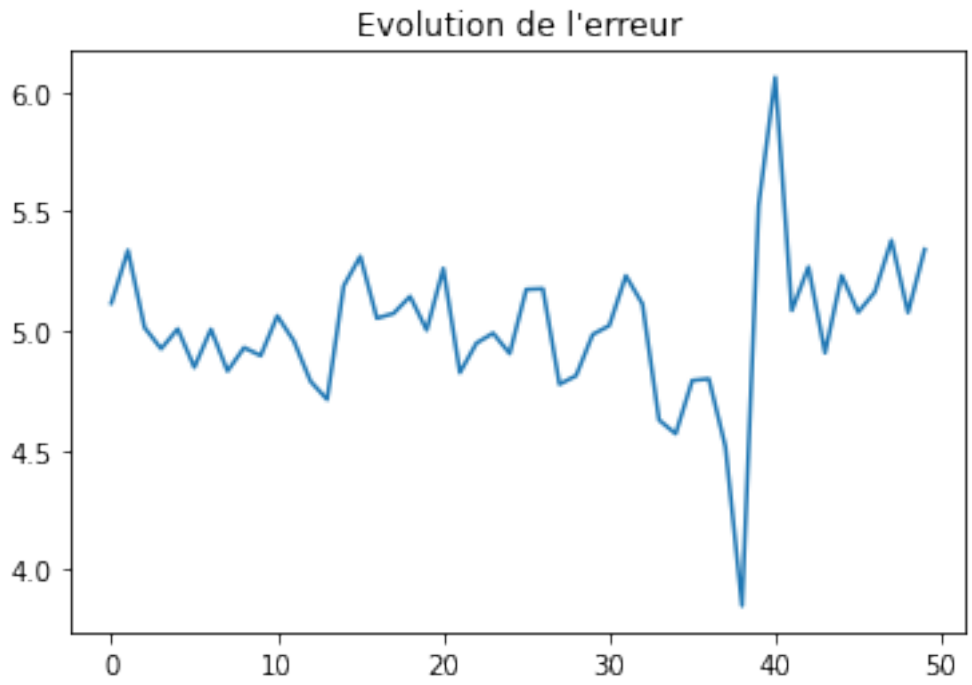
[27]:
```
max_iter = 50
```

[28]:
```
from tqdm import tqdm
import random

model = net2
optimizer = torch.optim.SGD(model.parameters(), lr=0.0002)
loss_func = torch.nn.MSELoss()

average_loss = []

for i in tqdm(range(max_iter)):

    ave_loss = 0
```

```
    nb = 0

    optimizer.zero_grad()
    for it in range(inputs.shape[0] // 10):
        h = random.randint(0, inputs.shape[0] - 1)
        x = inputs[h, :, :, :, :]
        y = targets[h, :]

        preds = model(x)
        loss = loss_func(preds, y)

        loss.backward()

        ave_loss += loss
        nb += 1

    optimizer.step()
    ave_loss /= nb
    average_loss.append((i, ave_loss))
```

```
  0%|          | 0/50 [00:00<?, ?it/s]c:\python372_x64\lib\site-
packages\torch\nn\modules\loss.py:432: UserWarning: Using a target size
(torch.Size([2])) that is different to the input size (torch.Size([1, 2])). This
will likely lead to incorrect results due to broadcasting. Please ensure they
have the same size.
  return F.mse_loss(input, target, reduction=self.reduction)
100%|¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ | 50/50 [02:59<00:00,  3.60s/it]
```

[29]:
```
import matplotlib.pyplot as plt
plt.plot([_[0] for _ in average_loss],
         [_[1] for _ in average_loss])
plt.title("Evolution de l'erreur");
```



1.7 On réessaye avec un réseau plus simple

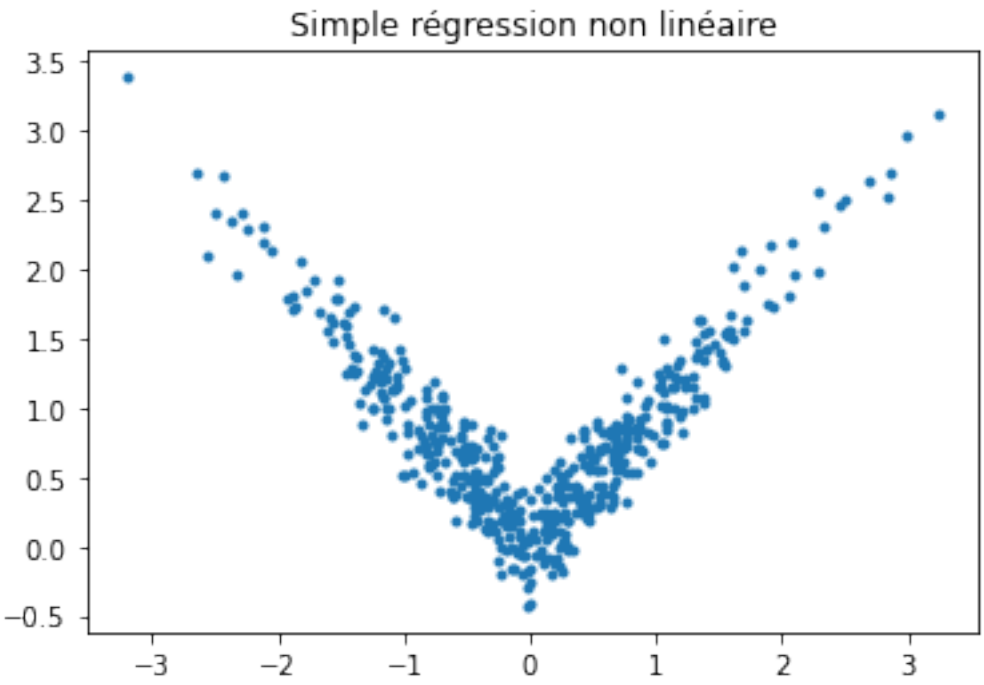
[30]:
```
class MyReLUF(torch.autograd.Function):

    @staticmethod
    def forward(ctx, input):
        ctx.save_for_backward(input)
        res = input.clamp(min=0)
        return res

    @staticmethod
    def backward(ctx, grad_output):
        input, = ctx.saved_tensors
        grad_input = grad_output.clone()
        grad_input[input < 0] = 0
        return grad_input


class MyReLUNN(nn.Module):

    def __init__(self):
        super(MyReLUNN, self).__init__()

    def forward(self, input):
        return MyReLUF.apply(input)

    def extra_repr(self):
        return 'in_features={}'.format(self.in_features)
```

```
class Net11(nn.Module):

    def __init__(self):
        super(Net11, self).__init__()
        self.lay1 = nn.Linear(1, 10)
        self.fct1 = MyReLUNN()
        self.lay2 = nn.Linear(10, 1)
        self.fct2 = MyReLUNN()

    def forward(self, x):
        x = self.fct1(self.lay1(x))
        x = self.fct2(self.lay2(x))
        return x


net11 = Net11()
input = torch.randn(1, 1)
output = net11(input)
output
```

[30]:
```
tensor([[0.]], grad_fn=<MyReLUFBackward>)
```

[31]:
```
N = 500
inputs = torch.randn(N, 1)
rnd = torch.randn(N, 1)
targets = torch.zeros(N, 1)
for i in range(N):
    y = abs(inputs[i, 0]) + rnd[i, 0] / 5
    targets[i, 0] = y
```

[32]:
```
plt.plot(list(inputs[:, 0]), list(targets[:, 0]), '.')
plt.title("Simple régression non linéaire");
```



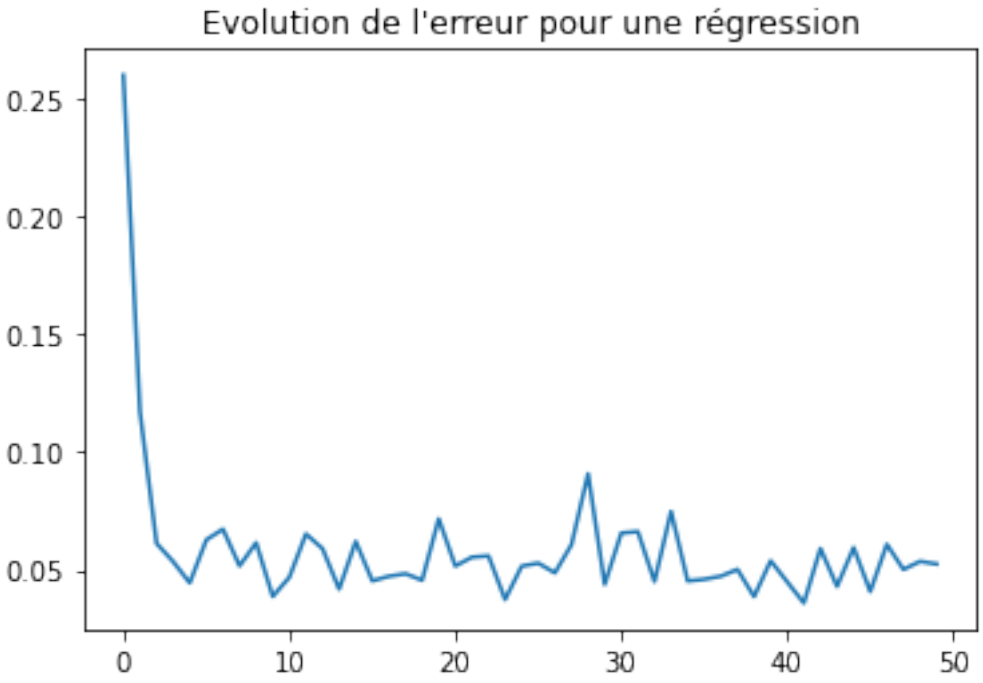
[33]:
```
max_iter = 50
```

[34]:
```
model = net11
optimizer = torch.optim.SGD(model.parameters(), lr=0.02)
loss_func = torch.nn.MSELoss()

average_loss = []

for i in tqdm(range(max_iter)):

    ave_loss = 0
    nb = 0

    for it in range(inputs.shape[0] // 10):
        h = random.randint(0, inputs.shape[0] - 1)
        x = inputs[h, :]
        y = targets[h, :]

        preds = model(x)

        loss = loss_func(preds, y)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        ave_loss += loss
        nb += 1
```

```
    ave_loss /= nb
    average_loss.append((i, ave_loss))
```

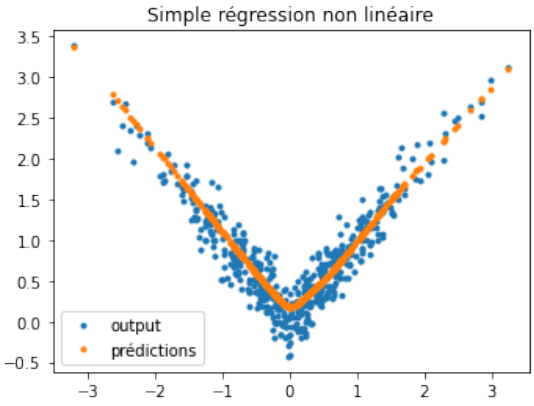
```
100%|¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ | 50/50 [00:01<00:00, 27.32it/s]
```

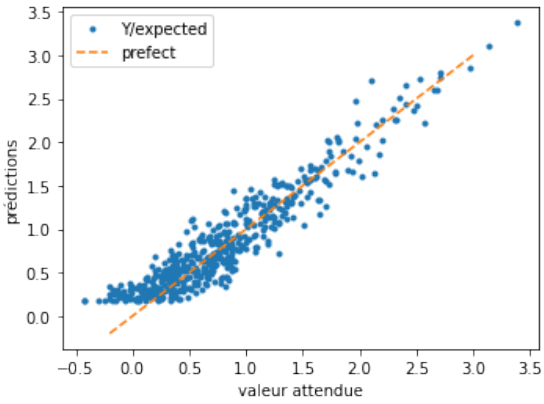
[35]:
```
import matplotlib.pyplot as plt
plt.plot([_[0] for _ in average_loss],
         [_[1] for _ in average_loss])
plt.title("Evolution de l'erreur pour une régression");
```



[36]:
```
predictions = model(inputs)
```

[37]:
```
fig, ax = plt.subplots(1, 2, figsize=(12, 4))
ax[0].plot(list(inputs[:, 0]), list(targets[:, 0]), '.', label="output")
ax[0].plot(list(inputs[:, 0]), list(predictions[:, 0]), '.', label="prédictions")
ax[0].legend()
ax[0].set_title("Simple régression non linéaire")
ax[1].plot(list(targets[:, 0]), list(predictions[:, 0]), '.', label="Y/expected")
ax[1].set_xlabel("valeur attendue")
ax[1].set_ylabel("prédictions")
ax[1].plot([-0.2, 3], [-0.2, 3], '--', label="prefect")
ax[1].legend();
```





Pas trop mal mais j'ai la flemme d'aller corriger ma première expérience qui n'avait aucun sens de toute façon.

[38] :

[39] :