带有约束的神经网络

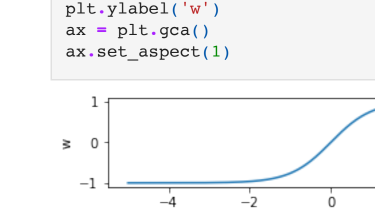
神经网络中的约束大致分为2类：显性约束和隐性约束。显性约束在这里指的是直接独立作用在可学习参数上的约束，例如约束可学习参数的值域。而隐性约束在这里指的是对通过若干权重算出来的值进行约束。

解决显性约束的问题比较简单直观，我们可以灵活运用各种映射（函数）来实现。

范围约束

例如，当我们训练一个神经网络时，我们希望所有的权重都在-1和+1之间。这时候我们可以引入中间变量来解决这个问题：我们可以回顾tanh函数的值域为$[-1,1]$，那么我们可以让可学习参数W'为无约束的变量，然后让$W=\tanh(W')$，这样W的范围就在$[-1,1]$之间了，见下图。然后我们让W参与加权求和的运算。这样，我们学到的W'所对应的W就是最优的用于加权求和的值，也就是权重。那么这里的W'可以看作是一个辅助的变量。下面我们看代码

In [23]:
```python
w_ = torch.linspace(-5,5,1000)
w = torch.tanh(w_)
plt.plot(w_.numpy(), w.numpy())
plt.xlabel('w_')
plt.ylabel('w')
ax = plt.gca()
ax.set_aspect(1)
```



前置代码：导入package，准备数据集等

In [18]:
```python
import torch
import pickle
import sys
sys.path.append('./others/codes/')
import training as T
import evaluation as E
import matplotlib.pyplot as plt
from torch.utils.data import TensorDataset, DataLoader

with open('./others/datasets/Dataset_breastcancerwisc.p', 'rb') as f:
    data = pickle.load(f)

X_train    = data['X_train']
y_train    = data['y_train']
X_valid    = data['X_valid']
y_valid    = data['y_valid']
X_test     = data['X_test']
y_test     = data['y_test']
data_name  = data['name']

N_class    = data['n_class']
N_feature  = data['n_feature']
N_train    = X_train.shape[0]
N_valid    = X_valid.shape[0]
N_test     = X_test.shape[0]

print(f'Dataset "{data_name}" has {N_feature} input features and {N_class} classes.\nThere are {N_train} training examp

train_data = TensorDataset(X_train, y_train)
valid_data = TensorDataset(X_valid, y_valid)
test_data  = TensorDataset(X_test, y_test)
train_loader = DataLoader(train_data, batch_size=len(train_data))
valid_loader = DataLoader(valid_data, batch_size=len(valid_data))
test_loader  = DataLoader(test_data, batch_size=len(test_data))
```

```
Dataset "breastcancerwisc" has 9 input features and 2 classes.
There are 418 training examples, 139 valid examples, and 140 test examples in the dataset.
```

前置代码：搭建正常的网络用于对比，实验中都用sigmoid作为激活函数，用9-5-2的网络结构。

In [2]:
```python
torch.manual_seed(0)
net = torch.nn.Sequential(torch.nn.Linear(N_feature, 5),
                          torch.nn.Sigmoid(),
                          torch.nn.Linear(5, N_class),
                          torch.nn.Sigmoid())
lossfunction = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=0.5)
```

In [3]:
```python
net, train_loss, valid_loss = T.train_nn(net,
                                         train_loader, valid_loader,
                                         lossfunction, optimizer)
```

```
The ID for this training is 1667243727.
| Epoch:        0 | Train loss: 0.66036 | Valid loss: 0.56218 |
| Epoch:      500 | Train loss: 0.32769 | Valid loss: 0.34996 |
| Epoch:     1000 | Train loss: 0.32764 | Valid loss: 0.35111 |
| Epoch:     1500 | Train loss: 0.32763 | Valid loss: 0.35249 |
| Epoch:     2000 | Train loss: 0.32762 | Valid loss: 0.35366 |
| Epoch:     2500 | Train loss: 0.32762 | Valid loss: 0.35452 |
| Epoch:     3000 | Train loss: 0.32762 | Valid loss: 0.35513 |
| Epoch:     3500 | Train loss: 0.32762 | Valid loss: 0.35555 |
| Epoch:     4000 | Train loss: 0.32762 | Valid loss: 0.35585 |
| Epoch:     4500 | Train loss: 0.32762 | Valid loss: 0.35606 |
| Epoch:     5000 | Train loss: 0.32762 | Valid loss: 0.35621 |
Early stop.
Finished.
```

In [4]:
```python
acc_train = E.ACC(net, X_train, y_train)
acc_valid = E.ACC(net, X_valid, y_valid)
acc_test  = E.ACC(net, X_test, y_test)
print(f'The accuracy on train set is {acc_train:.4f}, on valid set is {acc_valid:.4f}, on test set is {acc_test:.4f}.')
```

```
The accuracy on train set is 0.9833, on valid set is 0.9640, on test set is 0.9643.
```

下面正式开始搭建网络

In [5]:
```python
class ConstraintLayer(torch.nn.Module):
    def __init__(self, N_in, N_out):
        super().__init__()
        self.W_ = torch.nn.Parameter(torch.rand(N_in+1, N_out), requires_grad=True)
    @property
    def W(self):
        return torch.tanh(self.W_)

    def forward(self, X):
        X_extend = torch.hstack([X, torch.ones(X.shape[0],1)])
        Z = torch.matmul(X_extend, self.W)
        return torch.sigmoid(Z)
```

In [6]:
```python
torch.manual_seed(0)
net = torch.nn.Sequential(ConstraintLayer(N_feature, 5),
                          ConstraintLayer(5, N_class))
lossfunction = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=0.5)
```

In [7]:
```python
net, train_loss, valid_loss = T.train_nn(net,
                                         train_loader, valid_loader,
                                         lossfunction, optimizer)
```

```
The ID for this training is 1667243741.
| Epoch:        0 | Train loss: 0.68445 | Valid loss: 0.64514 |
| Epoch:      500 | Train loss: 0.45215 | Valid loss: 0.46640 |
| Epoch:     1000 | Train loss: 0.45214 | Valid loss: 0.46640 |
| Epoch:     1500 | Train loss: 0.45175 | Valid loss: 0.46614 |
| Epoch:     2000 | Train loss: 0.45175 | Valid loss: 0.46609 |
| Epoch:     2500 | Train loss: 0.45173 | Valid loss: 0.46616 |
| Epoch:     3000 | Train loss: 0.45173 | Valid loss: 0.46614 |
| Epoch:     3500 | Train loss: 0.45173 | Valid loss: 0.46613 |
| Epoch:     4000 | Train loss: 0.45173 | Valid loss: 0.46614 |
| Epoch:     4500 | Train loss: 0.45173 | Valid loss: 0.46614 |
| Epoch:     5000 | Train loss: 0.45173 | Valid loss: 0.46614 |
| Epoch:     5500 | Train loss: 0.45173 | Valid loss: 0.46613 |
| Epoch:     6000 | Train loss: 0.45173 | Valid loss: 0.46624 |
| Epoch:     6500 | Train loss: 0.45173 | Valid loss: 0.46614 |
| Epoch:     7000 | Train loss: 0.45173 | Valid loss: 0.46615 |
Early stop.
Finished.
```

In [8]:
```python
acc_train = E.ACC(net, X_train, y_train)
acc_valid = E.ACC(net, X_valid, y_valid)
acc_test  = E.ACC(net, X_test, y_test)
print(f'The accuracy on train set is {acc_train:.4f}, on valid set is {acc_valid:.4f}, on test set is {acc_test:.4f}.')
```

```
The accuracy on train set is 0.9641, on valid set is 0.9424, on test set is 0.9714.
```

可以看到，网络的效果也不差。要注意，这个网络中参与加权求和计算的是 `W` 而不是 `W_`，所以权重可以认为是 `W`。让我们来看一下权重，他们都在：

In [9]:
```python
for layer in net:
    print(layer.W)
```

```
tensor([[-0.2705, -0.2643,  0.2612,  0.2402, -0.2281],
        [ 1.0000,  1.0000, -1.0000, -1.0000,  1.0000],
        [ 1.0000,  1.0000, -1.0000, -1.0000,  1.0000],
        [ 1.0000,  1.0000, -1.0000, -1.0000,  1.0000],
        [-0.4232, -0.4251,  0.4133,  0.4092, -0.4396],
        [ 1.0000,  1.0000, -1.0000, -1.0000,  1.0000],
        [ 0.7494,  0.7408, -0.7193, -0.6951,  0.9529],
        [ 1.0000,  1.0000, -1.0000, -1.0000,  1.0000],
        [ 1.0000,  1.0000, -1.0000, -1.0000,  1.0000],
        [-1.0000, -1.0000,  1.0000,  1.0000, -1.0000]],
       grad_fn=<TanhBackward0>)
tensor([[-1.0000,  1.0000],
        [-1.0000,  1.0000],
        [ 1.0000, -1.0000],
        [ 1.0000, -1.0000],
        [-1.0000,  1.0000],
        [ 1.0000, -1.0000]], grad_fn=<TanhBackward0>)
```

对于范围约束，我们可以充分且灵活的运用各种函数，其实sigmoid也可以用于这个目的。甚至还可以进行变形。例如我们想让权重在[0,1]之间，我们当然可以直接用sigmoid进行映射，我们仍然还可以用tanh的变形$(\tanh+1)/2$：

In [10]:
```python
class ConstraintLayer(torch.nn.Module):
    def __init__(self, N_in, N_out):
        super().__init__()
        self.W_ = torch.nn.Parameter(torch.rand(N_in+1, N_out), requires_grad=True)
    @property
    def W(self):
        return (torch.tanh(self.W_) + 1.) / 2.

    def forward(self, X):
        X_extend = torch.hstack([X, torch.ones(X.shape[0],1)])
        Z = torch.matmul(X_extend, self.W)
        return torch.sigmoid(Z)
```

In [11]:
```python
torch.manual_seed(0)
net = torch.nn.Sequential(ConstraintLayer(N_feature, 5),
                          ConstraintLayer(5, N_class))
lossfunction = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=0.5)
```

In [12]:
```python
net, train_loss, valid_loss = T.train_nn(net,
                                         train_loader, valid_loader,
                                         lossfunction, optimizer)
```

```
The ID for this training is 1667244990.
| Epoch:        0 | Train loss: 0.69135 | Valid loss: 0.67705 |
| Epoch:      500 | Train loss: 0.64255 | Valid loss: 0.63861 |
| Epoch:     1000 | Train loss: 0.64253 | Valid loss: 0.63862 |
| Epoch:     1500 | Train loss: 0.64253 | Valid loss: 0.63862 |
| Epoch:     2000 | Train loss: 0.64253 | Valid loss: 0.63862 |
| Epoch:     2500 | Train loss: 0.64253 | Valid loss: 0.63862 |
| Epoch:     3000 | Train loss: 0.64253 | Valid loss: 0.63862 |
| Epoch:     3500 | Train loss: 0.64253 | Valid loss: 0.63863 |
| Epoch:     4000 | Train loss: 0.64253 | Valid loss: 0.63858 |
| Epoch:     4500 | Train loss: 0.64253 | Valid loss: 0.63860 |
| Epoch:     5000 | Train loss: 0.64253 | Valid loss: 0.63866 |
Early stop.
Finished.
```

In [13]:
```python
acc_train = E.ACC(net, X_train, y_train)
acc_valid = E.ACC(net, X_valid, y_valid)
acc_test  = E.ACC(net, X_test, y_test)
print(f'The accuracy on train set is {acc_train:.4f}, on valid set is {acc_valid:.4f}, on test set is {acc_test:.4f}.')
```

```
The accuracy on train set is 0.6411, on valid set is 0.6619, on test set is 0.6929.
```

In [14]:
```python
for layer in net:
    print(layer.W)
```

```
tensor([[2.8849e-05, 1.6844e-04, 9.9996e-01, 9.9996e-01, 1.0000e+00],
        [1.0000e+00, 9.9998e-01, 1.0000e+00, 9.9999e-01, 9.9999e-01],
        [9.9999e-01, 9.9997e-01, 9.9994e-01, 9.9996e-01, 9.9996e-01],
        [9.9999e-01, 9.9998e-01, 9.9997e-01, 9.9996e-01, 9.9996e-01],
        [3.2693e-05, 4.0650e-05, 9.9996e-01, 9.9998e-01, 9.9999e-01],
        [9.9962e-01, 9.9993e-01, 9.9996e-01, 9.9997e-01, 9.9998e-01],
        [3.6299e-05, 3.2236e-01, 9.9996e-01, 9.9997e-01, 1.0000e+00],
        [1.0000e+00, 9.9998e-01, 1.0000e+00, 1.0000e+00, 1.0000e+00],
        [1.0000e+00, 9.9999e-01, 9.9999e-01, 9.9999e-01, 9.9998e-01],
        [1.2219e-05, 1.4701e-04, 9.9997e-01, 9.9997e-01, 1.0000e+00]],
       grad_fn=<DivBackward0>)
tensor([[9.9996e-01, 9.9996e-01],
        [9.9995e-01, 2.2411e-01],
        [9.9997e-01, 4.6015e-05],
        [9.9997e-01, 5.4002e-05],
        [9.9994e-01, 1.5652e-04],
        [9.9997e-01, 1.5765e-05]], grad_fn=<DivBackward0>)
```

这里我们看到效果就差多了，因为负权重是必要的，没有了他们就没法表现负相关的信息。但是这是约束条件所导致的，跟网络本身的训练无关。

离散的值域约束

通过刚才的例子，我们发现，只要找到一个合适的函数，能把任意的 `W_` 映射到约束范围内的 `W`，就可以实现约束条件。但是我们发现，刚才的例子都是给定了一个范围的约束，在约束里，值域仍然是连续的。如果对于值域的约束是离散的，比如我们要求权重都是0.1的整数倍数，比如-0.1, 0.2, 1.4，我们再借助上面的方法就会出现问题。

我们先想象一个函数，能把任意的数字转换成证数，然后再乘以0.1就能满足上述约束了。这样的函数比如说四舍五入或者向上向下取整，让我们试一下：

In [15]:
```python
class ConstraintLayer(torch.nn.Module):
    def __init__(self, N_in, N_out):
        super().__init__()
        self.W_ = torch.nn.Parameter(torch.rand(N_in+1, N_out), requires_grad=True)
    @property
    def W(self):
        N = torch.round(self.W_)
        return N * 0.1

    def forward(self, X):
        X_extend = torch.hstack([X, torch.ones(X.shape[0],1)])
        Z = torch.matmul(X_extend, self.W)
        return torch.sigmoid(Z)
```

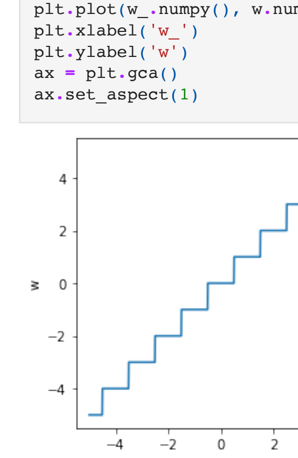
In [16]:
```python
torch.manual_seed(0)
net = torch.nn.Sequential(ConstraintLayer(N_feature, 5),
                          ConstraintLayer(5, N_class))
lossfunction = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=0.5)
```

In [17]:
```python
net, train_loss, valid_loss = T.train_nn(net,
                                         train_loader, valid_loader,
                                         lossfunction, optimizer)
```

```
The ID for this training is 1667245702.
| Epoch:        0 | Train loss: 0.68993 | Valid loss: 0.68936 |
| Epoch:      500 | Train loss: 0.68993 | Valid loss: 0.68936 |
| Epoch:     1000 | Train loss: 0.68993 | Valid loss: 0.68936 |
| Epoch:     1500 | Train loss: 0.68993 | Valid loss: 0.68936 |
| Epoch:     2000 | Train loss: 0.68993 | Valid loss: 0.68936 |
| Epoch:     2500 | Train loss: 0.68993 | Valid loss: 0.68936 |
| Epoch:     3000 | Train loss: 0.68993 | Valid loss: 0.68936 |
| Epoch:     3500 | Train loss: 0.68993 | Valid loss: 0.68936 |
| Epoch:     4000 | Train loss: 0.68993 | Valid loss: 0.68936 |
| Epoch:     4500 | Train loss: 0.68993 | Valid loss: 0.68936 |
| Epoch:     5000 | Train loss: 0.68993 | Valid loss: 0.68936 |
Early stop.
Finished.
```

我们可惜地发现，网络无法被训练。原因其实很简单，取整操作的梯度为0，切断了梯度的方向传播，如下图所示：

In [21]:
```python
w_ = torch.linspace(-5,5,1000)
w = torch.round(w_)
plt.plot(w_.numpy(), w.numpy())
plt.xlabel('w_')
plt.ylabel('w')
ax = plt.gca()
ax.set_aspect(1)
```



这时候我们会思考，尽管每一个点的梯度都为0（或者梯度不存在），也就是说，无论往右走还是往左走一小步，值都不会变化。但是我们仍然知道，这个函数关系的大趋势是：越往左值越小，越往右值越大。尽管在局部没有这个关系，我们仍然可以给一个大概的指引。也就是说，我们需要人为给定一个关系，比如$w_ \approx w$，这样也就是$dw_/dw \approx 1$用于指导训练的进行。那么怎么才能实现这个forward和backward用不同函数的功能呢？我们需要用到 `.detach()`。

让我们先来看一个$y=x^2+1, x=2$的例子：

In [33]:
```python
x = torch.nn.Parameter(torch.tensor(2.), requires_grad=True)
def f(x):
    result = x ** 2 + 1
    return result
y = f(x)
y
```

Out[33]:
```
tensor(5., grad_fn=<AddBackward0>)
```

结果是显而易见的。那么让我们看一下反向传播：经过计算我们知道此时$dy/dx=2x=4$，让我们检验一下：

In [34]:
```python
y.backward()
x.grad
```

Out[34]:
```
tensor(4.)
```

跟我们预料到一样。但是如果我们把 `result` 加上 `detach()`，就会出现下面的现象：

In [36]:
```python
x = torch.nn.Parameter(torch.tensor(2.), requires_grad=True)
def f(x):
    result = x ** 2 + 1
    return result.detach()
y = f(x)
y
```

Out[36]:
```
tensor(5.)
```

向前传递没有任何问题，但是向后传递时报错：

In [37]:
```python
y.backward()
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Input In [37], in <cell line: 1>()
----> 1 y.backward()

File ~/miniconda3/envs/ML/lib/python3.8/site-packages/torch/_tensor.py:363, in Tensor.backward(self, gradient, retain_gr
aph, create_graph, inputs)
    354 if has_torch_function_unary(self):
    355     return handle_torch_function(
    356         Tensor.backward,
    357         (self,),
   (...)
    361         create_graph=create_graph,
    362         inputs=inputs)
--> 363 torch.autograd.backward(self, gradient, retain_graph, create_graph, inputs=inputs)

File ~/miniconda3/envs/ML/lib/python3.8/site-packages/torch/autograd/__init__.py:173, in backward(tensors, grad_tensors,
retain_graph, create_graph, grad_variables, inputs)
    168     retain_graph = create_graph
    170 # The reason we repeat same the comment below is that
    171 # some Python versions print out the first line of a multi-line function
    172 # calls in the traceback and some print out the last line
--> 173 Variable._execution_engine.run_backward(  # Calls into the C++ engine to run the backward pass
    174     tensors, grad_tensors_, retain_graph, create_graph, inputs,
    175     allow_unreachable=True, accumulate_grad=True)

RuntimeError: element 0 of tensors does not require grad and does not have a grad_fn
```

原因在于 `detach` 的变量不会向后传播梯度，相当于切断了和其他部分的联系，所以 `y` 此时被切断了，也就谈不上反向传播了。下面我们做这个改动（在返回值后面再加一个 `x`）：

In [38]:
```python
x = torch.nn.Parameter(torch.tensor(2.), requires_grad=True)
def f(x):
    result = x ** 2 + 1
    return result.detach() + x
y = f(x)
y
```

Out[38]:
```
tensor(7., grad_fn=<AddBackward0>)
```

前向传播和之前预计的一样，等于$y=x^2+1+x=2^2+1+2=7$没有任何问题。这时候我们再反向传播：

In [39]:
```python
y.backward()
x.grad
```

Out[39]:
```
tensor(1.)
```

梯度为1，因为这时候$y=x^2+1+x$中，只有x项可以向后传播梯度，而x^2+1被切断了。所以再反向传播时，梯度无异于$dy/dx=dx/dx=1$。

有了这个功能，我们就可以自由的组合前向和反向传播了。但是要注意的是，这种人为给定的梯度，应当具有足够的代表性，给出变量被优化的大概走向，而不能背道而驰。至少在反向和正向的增减性上要相同。

那么我们现在就应该思考，应该怎么组合前向和后向传播呢？应该是这样： `return y_forward.detach() - y_backward.detach() + y_backward`
让我们来分析：前向传播时，后两项的值相反，所以抵消了，只有 `y_forward` 传递了过去。而在反向传播时，前两项对梯度没有影响，所以是按照 `y_backward` 传播。

这就是Straight Through Estimator

让我们应用在刚才的例子中：

In [40]:
```python
class ConstraintLayer(torch.nn.Module):
    def __init__(self, N_in, N_out):
        super().__init__()
        self.W_ = torch.nn.Parameter(torch.rand(N_in+1, N_out), requires_grad=True)
    @property
    def W(self):
        N_forward = torch.round(self.W_)
        N_backward = self.W_
        return (N_forward.detach() - N_backward.detach() + N_backward) * 0.1

    def forward(self, X):
        X_extend = torch.hstack([X, torch.ones(X.shape[0],1)])
        Z = torch.matmul(X_extend, self.W)
        return torch.sigmoid(Z)
```

In [41]:
```python
torch.manual_seed(0)
net = torch.nn.Sequential(ConstraintLayer(N_feature, 5),
                          ConstraintLayer(5, N_class))
lossfunction = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=0.5)
```

In [42]:
```python
net, train_loss, valid_loss = T.train_nn(net,
                                         train_loader, valid_loader,
                                         lossfunction, optimizer)
```

```
The ID for this training is 1667247349.
| Epoch:        0 | Train loss: 0.68993 | Valid loss: 0.68067 |
| Epoch:      500 | Train loss: 0.33184 | Valid loss: 0.35149 |
| Epoch:     1000 | Train loss: 0.32905 | Valid loss: 0.35098 |
| Epoch:     1500 | Train loss: 0.32825 | Valid loss: 0.35057 |
| Epoch:     2000 | Train loss: 0.32795 | Valid loss: 0.35055 |
| Epoch:     2500 | Train loss: 0.32781 | Valid loss: 0.35099 |
| Epoch:     3000 | Train loss: 0.32774 | Valid loss: 0.35063 |
| Epoch:     3500 | Train loss: 0.32770 | Valid loss: 0.35102 |
| Epoch:     4000 | Train loss: 0.32767 | Valid loss: 0.35101 |
| Epoch:     4500 | Train loss: 0.32766 | Valid loss: 0.35084 |
| Epoch:     5000 | Train loss: 0.32764 | Valid loss: 0.35075 |
| Epoch:     5500 | Train loss: 0.32764 | Valid loss: 0.35093 |
| Epoch:     6000 | Train loss: 0.32763 | Valid loss: 0.35102 |
| Epoch:     6500 | Train loss: 0.32763 | Valid loss: 0.35136 |
Early stop.
Finished.
```

In [43]:
```python
acc_train = E.ACC(net, X_train, y_train)
acc_valid = E.ACC(net, X_valid, y_valid)
acc_test  = E.ACC(net, X_test, y_test)
print(f'The accuracy on train set is {acc_train:.4f}, on valid set is {acc_valid:.4f}, on test set is {acc_test:.4f}.')
```

```
The accuracy on train set is 0.9856, on valid set is 0.9640, on test set is 0.9571.
```

In [44]:
```python
for layer in net:
    print(layer.W)
```

```
tensor([[-3.0000, -3.0000, -2.8000, -2.6000, -2.9000],
        [-3.2000, -3.1000, -3.0000, -3.0000, -3.0000],
        [-8.7000, -8.1000, -7.7000, -7.9000, -7.8000],
        [ 0.3000,  0.2000,  0.2000,  0.2000,  0.1000],
        [-6.9000, -6.3000, -6.1000, -6.2000, -6.2000],
        [-8.0000, -7.5000, -7.3000, -7.5000, -7.5000],
        [-7.2000, -6.8000, -6.7000, -6.7000, -6.8000],
        [-3.3000, -3.2000, -3.1000, -2.9000, -3.1000],
        [-5.1000, -4.7000, -4.7000, -5.2000, -4.8000],
        [ 9.1000,  8.5000,  8.2000,  8.1000,  8.4000]], grad_fn=<MulBackward0>)
tensor([[ 6.0000, -5.8000],
        [ 5.7000, -5.7000],
        [ 4.5000, -4.4000],
        [ 4.5000, -4.4000],
        [ 5.0000, -5.0000],
        [-7.7000,  7.7000]], grad_fn=<MulBackward0>)
```

可以看出，用于加权求和的值都是0.1的倍数了，而且网络效果也很不错。

总结

这里我们讲了2种考虑约束的方法。其中straight through estimator十分重要，而且可以被灵活运用在各种地方，最主要的应用就是给没有梯度的正向函数人为指定一个具有梯度的反向传播函数用于指导训练。

从优化的术语来看，这也可以叫做松弛反向传播（我自己起的名字）。松弛在优化中的意思就是弱化约束，应用于对问题的简化和估计。比如本来一个变量只能是0-10之间的整数，这是一个困难的整数优化问题，经过松弛之后它可以时0-10之间的任意数字，这样就变成了一个连续的优化问题，简单了许多。这种松弛常用与估计一个问题优化的上下界，也就是说，一个问题如果在[0,10]内有最大值y^*，那么它在$\{0,1,2,...,10\}$的最大值肯定小于等于y^*，如果连y^*都达不到要求的话，那么这个问题也就没有解的必要了。