Machine Learning Group
Faculty IV – Electrical Engineering and Computer Science
Technische Universität Berlin

Exercise Sheet 4 (programming part)

In this homework, we will train neural networks on the Boston housing dataset. For this, we will use of the Pytorch library. We will also make use of scikit-learn for the ML baselines. A first part of the homework will analyze the parameters of the network before and after training. A second part of the homework will test some regularization penalties and their effect on the generalization error.

Boston Housing Dataset

The following code extracts the Boston housing dataset in a way that is already partitioned into training and test data. The data is normalized such that each dimension has mean 0 and variance 1.

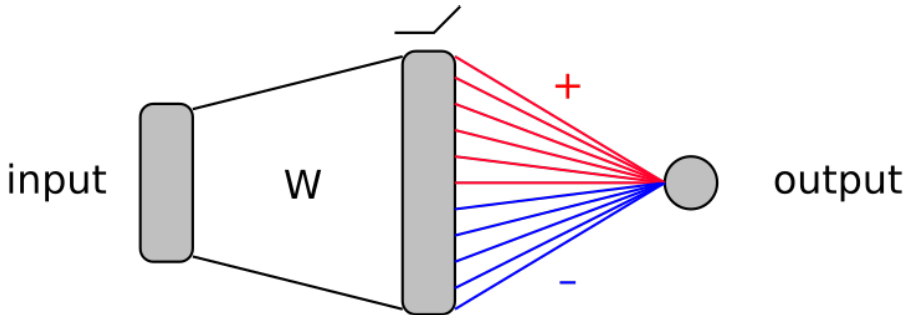
In [1]:
```
import utils

Xtrain,Ttrain,Xtest,Ttest = utils.boston()

nx = Xtrain.shape[1]
nh = 100
```

Neural Network Regressor

In this homework, we will consider a very simple architecture consisting of one linear layer, a ReLU layer applying a nonlinear function element-wise, and a pooling layer computing a fixed weighted sum of the activations obtained in the previous layer. The architecture is shown below:



The class `NeuralNetworkRegressor` implements this network. The function `reg` is a regularizer which we set initially to zero (i.e. no regularizer). Because the dataset is small, the network is optimized in batch mode, using the Adam optimizer.

In [2]:
```
import numpy,torch,sklearn,sklearn.metrics
from torch import nn,optim

class NeuralNetworkRegressor:

    def __init__(self):

        torch.manual_seed(0)

        self.model = nn.Sequential(nn.Linear(nx,nh),nn.ReLU())
        self.pool  = lambda y: 0.1*(y[:,:nh//2].sum(dim=1)-y[:,nh//2:].sum(dim=1))
        self.loss  = nn.MSELoss()

    def reg(self): return 0

    def fit(self,X,T,nbit=10000):

        X = torch.Tensor(X)
        T = torch.Tensor(T)

        optimizer = optim.Adam(self.model.parameters(),lr=0.01)
        for _ in range(nbit):
            optimizer.zero_grad()
            (self.loss(self.pool(self.model(X)),T)+self.reg()).backward()
            optimizer.step()

    def predict(self,X):
        return self.pool(self.model(torch.Tensor(X)))

    def score(self,X,T):
        return sklearn.metrics.r2_score(T,numpy.array(self.predict(X).data))
```

Neural Network Performance vs. Baselines

We compare the performance of the neural network on the Boston housing data to two other regressors: a random forest and a support vector regression model with RBF kernel. We use the scikit-learn implementation of these models, with their default parameters.

In [4]:
```
from sklearn import ensemble,svm

rfr = ensemble.RandomForestRegressor(random_state=0)
rfr.fit(Xtrain,Ttrain)

svr = svm.SVR()
svr.fit(Xtrain,Ttrain)

nnr = NeuralNetworkRegressor()
nnr.fit(Xtrain,Ttrain)
```

In [5]:
```
def pretty(name,model):
    return '> %10s | R2train: %6.3f | R2test: %6.3f'%(name,model.score(Xtrain,Ttrain),model.sc
ore(Xtest,Ttest))

print(pretty('RForest',rfr))
print(pretty('SVR',svr))
print(pretty('NNreg',nnr))
```

```
>    RForest | R2train:  0.977 | R2test:  0.864
>        SVR | R2train:  0.913 | R2test:  0.758
>      NNreg | R2train:  1.000 | R2test:  0.787
```

The neural network performs worse than other regression models on test data due to the absence of regularization. We would instead expect a well-regularized neural network to perform better.

Gradient, and Parameter Norms (30 P)

As a first step towards improving the neural network model, we will measure proxy quantities, that will then be used to regularize the model. We consider the following three quantities:

- $\|W\|_{\text{Frob}} = \sqrt{\sum_{i=1}^{d}\sum_{j=1}^{h} w_{ij}^2}$
- $\|W\|_{\text{Mix}} = h^{-0.5}\sqrt{\sum_{i=1}^{d}\left(\sum_{j=1}^{h}|w_{ij}|\right)^2}$
- $\text{Grad} = \frac{1}{N}\sum_{n=1}^{N}\|\nabla_{\boldsymbol{x}} f(\boldsymbol{x_n})\|$

where d is the number of input features, h is the number of neurons in the hidden layer, and W is the matrix of weights in the first layer (*Note that in PyTorch, the matrix of weights is given in transposed form*). In order for the model to generalize well, the last quantity (Grad) should be prevented from becoming too large. Because the latter depends on the data distribution, we rely instead on the inequality $\text{Grad} \leq \|W\|_{\text{Mix}} \leq \|W\|_{\text{Frob}}$, that we can prove for this model, and will try to control the weight norms instead. The function `Frob(nn)` that computes $\|W\|_{\text{Frob}}$ is already implemented for you.

Tasks:

- Implement the function `Mix(nn)` that receives the neural network as input and returns $\|W\|_{\text{Mix}}$.
- Implement the function `Grad(nn,X)` that receives the neural network and some dataset as input, and computes the averaged gradient norm (Grad).

In [6]:
```
def Frob(nn):
    W = list(nn.model)[0].weight
    return (W**2).sum()**.5

def Mix(nn):
    # ---------------------------------
    # TODO: Replace by your code
    # ---------------------------------
    import solution; return solution.Mix(nn)
    # ---------------------------------

def Grad(nn,X):
    # ---------------------------------
    # TODO: Replace by your code
    # ---------------------------------
    import solution; return solution.Grad(nn,X)
    # ---------------------------------
```

The following code measures these three quantities before and after training the model.

In [7]:
```
def fullpretty(name,nn):
    return pretty(name,nn) + ' | WFrob: %7.3f | WMix: %7.3f | Grad: %7.3f'%(Frob(nn),Mix(nn),G
rad(nn,Xtest))

nnr = NeuralNetworkRegressor()
print(fullpretty('Before',nnr))
nnr.fit(Xtrain,Ttrain)
print(fullpretty('After',nnr))
```

```
>     Before | R2train: -0.503 | R2test: -0.534 | WFrob:   5.890 | WMix:   5.135 | Grad:   0.37
5
>      After | R2train:  1.000 | R2test:  0.787 | WFrob:  29.389 | WMix:  22.781 | Grad:   1.53
9
```

We observe that the inequality $\text{Grad} \leq \|W\|_{\text{Mix}} < \|W\|_{\text{Frob}}$ also holds empirically. We also observe that these quantities tend to increase as training proceeds. This is a typical behavior, as the network starts rather simple and becomes complex as more and more variations in the training data are being captured.

Norm Penalties (20 P)

We consider the new objective $J^{\text{Frob}}(\theta) = \text{MSE}(\theta) + \lambda \cdot \|W\|^2_{\text{Frob}}$, where the first term is the original mean square error objective and where the second term is the added penalty. We hardcode the penalty coeffecient to $\lambda = 0.002$. In principle, for maximum performance and fair comparison between the methods, several of them should be tried (also for other models), and selected based on some validation set. Here, for simplicity, we omit this step.

A downside of the Frobenius norm is that it is not a very tight upper bound of the gradient, that is, penalizing it is does not penalize specifically high gradient. Instead, other useful properties of the model could be negatively affected by it. Therefore, we also experiment with the mixed-norm regularizer $\lambda \cdot \|W\|^2_{\text{Mix}}$, which is a tighter bound of the gradient, and where we also hardcode the penalty coefficient to $\lambda = 0.002$.

Task:

- Create two new regressors by reimplementing the regularization function with the Frobenius norm regularizer and Mixed norm regularizer respectively. You may for this task call the norm functions implemented in the question above, but this time you also need to ensure that these functions can be differentiated w.r.t. the weight parameters.

The code below trains a neural network with the new first layer, and compares the performance with the previous models.

In [8]:
```
import solution

class FrobRegressor(NeuralNetworkRegressor):

    def reg(self):
        # ----------------------------------
        # TODO: Replace by your code
        # ----------------------------------
        import solution; return solution.FrobReg(self)
        # ----------------------------------

class MixRegressor(NeuralNetworkRegressor):

    def reg(self):
        # ----------------------------------
        # TODO: Replace by your code
        # ----------------------------------
        import solution; return solution.MixReg(self)
        # ----------------------------------
```

In [9]:
```
nnfrob = FrobRegressor()
nnfrob.fit(Xtrain,Ttrain)

nnmix = MixRegressor()
nnmix.fit(Xtrain,Ttrain)
```

In [10]:
```
print(pretty('RForest',rfr))
print(pretty('SVR',svr))
print(fullpretty('NN',nnr))
print(fullpretty('NN+Frob',nnfrob))
print(fullpretty('NN+Mix',nnmix))
```

```
>    RForest | R2train:  0.977 | R2test:  0.864
>        SVR | R2train:  0.913 | R2test:  0.758
>         NN | R2train:  1.000 | R2test:  0.787 | WFrob:  29.389 | WMix:  22.781 | Grad:   1.53
9
>    NN+Frob | R2train:  0.958 | R2test:  0.830 | WFrob:   3.929 | WMix:   3.215 | Grad:   0.80
0
>     NN+Mix | R2train:  0.972 | R2test:  0.834 | WFrob:   7.449 | WMix:   3.319 | Grad:   0.89
5
```

It is interesting to observe that this mixed norm penalty more selectively reduced the mixed norm and the gradient, and has let the Frobenius norm take higher values. Here, we observe that the mixed-norm model is also the one that produces the second highest test set accuracy in this benchmark after the random forest.