sheet2

June 11, 2018

1 Exercise sheet 2: Training and Regularization

In this homework, we will train neural networks on the Boston housing dataset. For this, we will use the modular framework developed in Homework 1. 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.

1.1 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 [21]: import utils

         Xtrain,Ttrain,Xtest,Ttest = utils.getBostonHousingData()
```

1.2 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 function `getarch` shown below generates an architecture specific to the Boston housing dataset, with 13 input features, `h` intermediate neurons, and one output. It takes as a parameter the first layer type, which is usually `layers.Linear`. Later in this homework, we will replace it by variants of `layers.Linear` that incorporate weight norm penalties.

```
In [22]: import layers

         def getarch(FirstLayerType):
             h = 100
             return layers.Sequential([
                 FirstLayerType(13,h,seed=0),
                 layers.ReLU(),
```

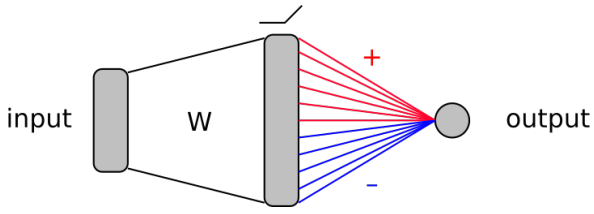

Diagram of the Neural Network Regressor used in this homework

```
            layers.Pooling(h)
        ])
```

The class `NeuralNetworkRegressor` shown below takes a neural network architecture, trains it on the data using gradient descent with momentum, and predict new data points. Because the dataset is rather small, we do not need to use stochastic gradient descent. We choose instead the batch variant, where we follow the gradient of the true mean square error.

```
In [23]: import sklearn,sklearn.metrics

         class NeuralNetworkRegressor:

             def __init__(self,arch): self.arch = arch

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

                 for i in range(nbit):
                     self.arch.backward(2*(self.arch.forward(X)-T))
                     for l in self.arch.layers: l.computepgrad(); l.pgradstep(0.01)

             def predict(self,X):
                 return self.arch.forward(X)

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

1.2.1 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 [24]: import sklearn.ensemble,sklearn.svm

         rfr = sklearn.ensemble.RandomForestRegressor()
         rfr.fit(Xtrain,Ttrain)

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

         nnr = NeuralNetworkRegressor(getarch(layers.Linear))
         nnr.fit(Xtrain,Ttrain)

In [25]: def pretty(name,model):
             return '> %10s | R2train: %6.3f | R2test: %6.3f'%(name,model.score(Xtrain,Ttrain),model.sc

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

>    RForest | R2train:  0.973 | R2test:  0.863
>        SVR | R2train:  0.913 | R2test:  0.758
>      NNreg | R2train:  0.978 | R2test:  0.856
```

The neural networks performs on par with the other regression models although it has in principle the added ability to learn its own internal features. We would therefore expect a well-trained neural network to perform better.

1.3 Gradient, and Parameter Norms (20 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:

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

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. 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.

Tasks:

- Implement the function `WMix(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 [26]: def WFrob(nn):
             W = nn.arch.layers[0].W
             return (W**2).sum()**.5

         def WMix(nn):
             W = nn.arch.layers[0].W
             return (1 / W.shape[1] * (abs(W).sum(axis = 1) ** 2).sum())**0.5
         import numpy as np

         def Grad(nn,X):
             W = nn.arch.layers[0].W
             B = nn.arch.layers[0].B
             def grad1(X,W,B):
                 s = np.zeros(W.shape[0])
                 for j in range(W.shape[1]):
                     I = X.dot(W[:,j]) + B[j] > 0
                     s += W[:,j] * I
                 return (s**2).sum()**0.5
             ret = np.mean([grad1(x,W,B) for x in X])
             return ret
```

The following experiment computes these three quantities before and after training the model.

```
In [27]: def fullpretty(name,nn):
             return pretty(name,nn) + ' | WFrob: %7.3f | WMix: %7.3f | Grad: %7.3f'%(WFrob(nn),WMix(nn)

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

```
>     Before | R2train: -0.043 | R2test: -0.006 | WFrob:   9.834 | WMix:   7.845 | Grad:  13.562
>      After | R2train:  0.978 | R2test:  0.856 | WFrob:   8.188 | WMix:   4.214 | Grad:   8.821
```

We observe that the inequality 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.

1.4 Frobenius-Norm Penalty (10 P)

The first penalty we experiment with is the squared Frobenius norm. 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 $\lambda = 10^{-3}$. 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.

Tasks:

- Create a new layer, that replaces the original `layers.Linear` layer and incorporates the proposed penalty. *(Hint: This can be achieved by extending layers.Linear, and rewriting its method `computepgrad`.)*

```
In [28]: import copy
         LinearFrob = copy.deepcopy(layers.Linear)
         def computepgrad (self,lam = 1/10**3):
             self.DW = np.dot(self.A.T,self.DZ)/len(self.A) + lam *2 * self.W
             self.DB = self.DZ.mean(axis=0)
         LinearFrob.computepgrad = computepgrad

         # ----------------------------------------
```

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

```
In [29]: nnfrob = NeuralNetworkRegressor(getarch(LinearFrob))
         nnfrob.fit(Xtrain,Ttrain)

In [30]: print(pretty('RForest',rfr))
         print(pretty('SVR',svr))
         print(fullpretty('NN',nnr))
         print(fullpretty('NN+Frob',nnfrob))
```

```
>    RForest | R2train:  0.973 | R2test:  0.863
>        SVR | R2train:  0.913 | R2test:  0.758
>         NN | R2train:  0.978 | R2test:  0.856 | WFrob:   8.188 | WMix:   4.214 | Grad:   8.821
>    NN+Frob | R2train:  0.967 | R2test:  0.833 | WFrob:   5.119 | WMix:   4.139 | Grad:   7.263
```

As we can see, the penalty has the effect of strongly reducing the Frobenius norm of the weight matrix. However, the test accuracy is not improved.

1.5 Mixed-Norm Penalty (10 P)

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. In the following, we experiment with the mixed-norm regularizer, which is a tighter bound of the gradient.

The objective function is now given by $J^{\text{Mix}}(\theta) = \text{MSE}(\theta) + \lambda \cdot \|W\|^2_{\text{Mix}}$, where the first and second terms are again the original objective and the penalty term. As for the previous exercise, we hardcode the latter coefficient to 10^{-3}.

Tasks:

- Using the same technique as before, create a new layer that incorporates the Mixed norm penalty.

The differential of the Mix norm for element $W_{l,k}$ is $\frac{\partial \|W\|^2_{\text{Mix}}}{\partial W_{l,k}} = 2h^{-1}\sum_{i=1}^{h} W_{l,j}\text{sign}(W_{l,k}) = 2h^{-1}W\_\{l,:\}|1sign(W\{l,k\})$ thus $\frac{\partial \|W\|^2_{\text{Mix}}}{\partial W} = 2h^{-1}\tilde{W}_{(1)}1_h^T \odot \text{sign}(W)$ with $\tilde{W}_{(1)} := (\|W_{1,:}\|_1, ..., \|W_{d,:}\|_1)^T$. Reminder: $\odot$ represents the operator for the elemente-wise matrix multiplication.

```
In [31]: LinearMix = copy.deepcopy(layers.Linear)
         def computepgrad (self,lam = 1/10**3):
             _W2 = np.outer(abs(self.W).sum(axis= 1),np.ones(self.W.shape[1]))
             self.DW = np.dot(self.A.T,self.DZ)/len(self.A)
             self.DW+= lam * 2 / self.W.shape[1] * np.multiply(_W2,np.sign(self.W))
             self.DB = self.DZ.mean(axis=0)
         LinearMix.computepgrad = computepgrad
```

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

```
In [32]: nnmix = NeuralNetworkRegressor(getarch(LinearMix))
         nnmix.fit(Xtrain,Ttrain)

In [33]: print(pretty('RForest',rfr))
         print(pretty('SVR',svr))
         print(fullpretty('NN',nnr))
         print(fullpretty('NN+Frob',nnfrob))
         print(fullpretty('NN+Mix',nnmix))
```

```
>    RForest | R2train:  0.973 | R2test:  0.863
>        SVR | R2train:  0.913 | R2test:  0.758
>         NN | R2train:  0.978 | R2test:  0.856 | WFrob:   8.188 | WMix:   4.214 | Grad:   8.821
>    NN+Frob | R2train:  0.967 | R2test:  0.833 | WFrob:   5.119 | WMix:   4.139 | Grad:   7.263
>     NN+Mix | R2train:  0.978 | R2test:  0.856 | WFrob:   8.188 | WMix:   4.214 | Grad:   8.821
```

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. The mixed-norm model is also the one that produces the highest test set accuracy.