Neural Networks 2

In this homework, we will train neural networks on the Breast Cancer 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.

Breast Cancer Dataset

The following code extracts the Breast cancer 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. To test the robustness of our learning models, we also artificially inject 4% of mislabelings in the training data.

In [41]:
```
import utils

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

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

Neural Network Classifier

In this homework, we consider the same architecture as the one considered in Exercise 2 of the theoretical part. The class `NeuralNetworkClassifier` 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 can be optimized in batch mode, using the Adam optimizer.

In [42]:
```
import numpy,torch,sklearn,sklearn.metrics
from torch import nn,optim

class NeuralNetworkClassifier:

    def __init__(self):
```

```
        torch.manual_seed(0)

        self.model = nn.Sequential(nn.Linear(nx,nh),nn.ReLU())
        with torch.no_grad(): list(self.model)[0].weight *= 0.1
        self.s = torch.zeros([100]); self.s[:50] = 1; self.s[50:] = -1
        self.pool  = lambda y: y.matmul(self.s)
        self.loss  = lambda y,t: torch.clamp(1-y*t,min=0).mean()

    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):
        Y = numpy.sign(self.predict(X).data.numpy())
        return sklearn.metrics.accuracy_score(T,Y)
```

Neural Network Performance vs. Baselines

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

In [43]:
```
from sklearn import ensemble,svm

rfc = ensemble.RandomForestClassifier(random_state=0)
rfc.fit(Xtrain,Ttrain)
```

```
svc = svm.SVC()
svc.fit(Xtrain,Ttrain)

nnc = NeuralNetworkClassifier()
nnc.fit(Xtrain,Ttrain)
```

In [44]:
```
def pretty(name,model):
    return '> %10s | Train Acc: %6.3f | Test Acc: %6.3f'%(name,model.score(Xtrain,Ttrain),model.score(Xtest,Ttest))

print(pretty('RForest',rfc))
print(pretty('SVC',svc))
print(pretty('NN',nnc))
```

```
>    RForest | Train Acc:  1.000 | Test Acc:  0.940
>        SVC | Train Acc:  0.958 | Test Acc:  0.951
>         NN | Train Acc:  1.000 | Test Acc:  0.884
```

The neural network performs not as good as the baselines. Most likely, the neural network has overfitted its decision boundary, in particular, on the mislabeled training examples.

Gradient, and Parameter Norms (25 P)

For the model to generalize better, we assume that the gradient of the decision function should be prevented from becoming too large. Because the gradient can only be evaluated on the current data distribution (and may not generalize outside the data), we resort to the following inequality we have proven in the theoretical section for this class of neural network models:

$$\text{Grad} \leq \|W\|_{\text{Mix}} \leq \sqrt{h}\|W\|_{\text{Frob}}$$

where

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

and 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*).

As a first step, we would like to keep track of these quantities during training. 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 [77]:
```python
import numpy as np

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

def Mix(nn):
    # ---------------------------------
    # TODO: Replace by your code
    # ---------------------------------
    W = list(nn.model)[0].weight
    return ((W.abs().sum(axis=0))**2).sum()**.5
    # ---------------------------------

def Grad(nn,X):
    # ---------------------------------
    # TODO: Replace by your code
    # ---------------------------------
    if isinstance(X, np.ndarray):
        X = torch.tensor(X, dtype=torch.float32, requires_grad=True)
    else:
        X.requires_grad = True  # Enable gradient computation for PyTorch tensor

    nn.model.zero_grad()
    output = nn.model(X)
    grad_outputs = torch.ones_like(output)  # Gradient of the output w.r.t. itself
    gradients = torch.autograd.grad(outputs=output, inputs=X, grad_outputs=grad_outputs, create_graph=True)[0]
```

```
    grad_norms = gradients.norm(2, dim=1)  # Compute the gradient norms for each input
    avg_grad_norm = grad_norms.mean()      # Compute the average gradient norm
    return avg_grad_norm
    # ---------------------------------
```

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

In [78]:
```
def fullpretty(name,nn):
    return pretty(name,nn) + ' | Grad: %7.3f | WMix: %7.3f | sqrt(h)*WFrob: %7.3f'%(Grad(nn,Xtest),Mix(nn),nh**.5*F

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

```
>     Before | Train Acc:  0.391 | Test Acc:  0.372 | Grad:   0.425 | WMix:   4.966 | sqrt(h)*WFrob:   5.751
>      After | Train Acc:  1.000 | Test Acc:  0.884 | Grad:  10.911 | WMix:  40.103 | sqrt(h)*WFrob:  56.739
```

We observe that the inequality $\mathrm{Grad} \leq \|W\|_{\mathrm{Mix}} \leq \sqrt{h}\|W\|_{\mathrm{Frob}}$ we have proven 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 (15 P)

We consider the new objective $J^{\mathrm{Frob}}(\theta) = \mathrm{MSE}(\theta) + \lambda \cdot (\sqrt{h}\|W\|_{\mathrm{Frob}})^2$, 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 $\lambda = 0.005$. 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 $\lambda \cdot \|W\|^2_{\mathrm{Mix}}$, which is a tighter bound of the gradient, and where we also hardcode the penalty coefficient to $\lambda = 0.025$.

Task:

- Create two new classifiers 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 implements and train neural networks with the new regularizers, and compares the performance with the previous models.

In [79]:
```
class FrobClassifier(NeuralNetworkClassifier):

    def reg(self):
        # ---------------------------------
        # TODO: Replace by your code
        # ---------------------------------
        W = list(self.model)[0].weight
        h = W.shape[0]
        return 0.005*nh*(Frob(self)**2)
        # ---------------------------------

class MixClassifier(NeuralNetworkClassifier):

    def reg(self):
        # ---------------------------------
        # TODO: Replace by your code
        # ---------------------------------
        return 0.025*(Mix(self)**2)
        # ---------------------------------
```

In [80]:
```
nnfrob = FrobClassifier()
nnfrob.fit(Xtrain,Ttrain)

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

In [81]:
```
print(pretty('RForest',rfc))
print(pretty('SVC',svc))
print(fullpretty('NN',nnc))
```

```
print(fullpretty('NN+Frob',nnfrob))
print(fullpretty('NN+Mix',nnmix))
```

```
>    RForest | Train Acc:  1.000 | Test Acc:  0.940
>        SVC | Train Acc:  0.958 | Test Acc:  0.951
>         NN | Train Acc:  1.000 | Test Acc:  0.884 | Grad:  10.911 | WMix:  40.103 | sqrt(h)*WFrob:  56.739
>    NN+Frob | Train Acc:  0.961 | Test Acc:  0.951 | Grad:   0.614 | WMix:   1.719 | sqrt(h)*WFrob:   2.713
>     NN+Mix | Train Acc:  0.951 | Test Acc:  0.954 | Grad:   0.544 | WMix:   1.605 | sqrt(h)*WFrob:   4.097
```

We observe that the regularized neural networks perform on par with the baselines. It is interesting to observe that the mixed norm penalty more selectively reduced the gradient, and has let the Frobenius norm take higher values.