Table of Contents

Deep Feature Selection

In this notebook, we will demonstrate how to implement our method on the nonlinear simulation examples from our paper.

User Guide on nonlinear example

In this example, a high dimensional dataset with 500 covariates and 300 observations is generated using the following equation

$$y = \begin{cases} 1, & e^{x_1} + x_2^2 + 5\sin(x_3 x_4) - 3 > 0 \\ 0, & \text{otherwise,} \end{cases}$$

i.e. among 500 covariates, only the first 4 variables actually contributed to the response. Our task is to correctly select the important variables. Please see section 5.2 of the paper for detailed generation process.

In [1]:
```
import sys
sys.path.append("../../src")
from time import clock
import numpy as np
import math
import pandas as pd
import matplotlib.pyplot as plt
import torch
import torch.nn.functional as F
from torch.autograd import Variable
from torch.autograd import grad
from torch.nn.parameter import Parameter
from utils import data_load_n, data_load_l, measure, accuracy
from models import Net_nonlinear, Net_linear
from dfs import DFS_epoch, training_n
```

Data Preparation

We will load our data in the following chunk. The data, both covariates and response, need to be load as `pytorch Tensor` objects to be fed into the DFS algorithm. The function `data_load_n` will read in dataset and split it into training and test set so that both sets have same number of positive and negative samples.

In [2]:
```
# load and prepare datasets
dirc = "../../data/nonlinear/p_500_N_600_s_4/"
k = 0 # dataset number from 0 to 9
X, Y, X_test, Y_test = data_load_n(k, directory=dirc)
N, p = X.shape
print("The covariates is of type:", type(X))
print("The response is of type:", type(Y))
print()
print("The dimension of training set:", X.shape)
print("    The number of positive sample:", len(np.where(Y==1)[0]))
print("    The number of negative sample:", len(np.where(Y==0)[0]))
print()
print("The dimension of test set:", X.shape)
print("    The number of positive sample:", len(np.where(Y_test==1)[0]))
print("    The number of negative sample:", len(np.where(Y_test==0)[0]))
```

```
The covariates is of type: <class 'torch.Tensor'>
The response is of type: <class 'torch.Tensor'>

The dimension of training set: torch.Size([300, 500])
    The number of positive sample: 150
    The number of negative sample: 150

The dimension of test set: torch.Size([300, 500])
    The number of positive sample: 150
    The number of negative sample: 150
```

DFS with fixed hyper-parameters

In this section, we demonstrate how to run DFS with one given set of hyper-parameters. The hyper-parameters includes:

- `s`, the number of variables to be selected;
- `c`, the tunning parameters to control the magnitude of λ_1 and λ_2;
- `epochs`, the number of DFS iterations to be run;
- `n_hidden1` & `n_hidden2`, the number of neurons in the fully connect neural networks;
- `learning_rate`, the learning rate for optimizer;
- `Ts` & `step`, the parameters to control the optimization on given support

Among the above hyper-parameters, `s` is the most important parameters, and the selection of s will be demonstrated in next sections. `c` can be selection through a sequence of candidates that returns the smallest loss function. Others mostly are meant to help the convergence of the optimization steps.

In [3]:
```
# specify hyper-paramters
s = 4
c = 1
epochs = 10
n_hidden1 = 50
n_hidden2 = 10
learning_rate = 0.05
Ts = 25 # To avoid long time waiting, this parameter has been shorten
step = 5


# Define Model
torch.manual_seed(1) # set seed
# Define a model with pre-specified structure and hyper parameters
model = Net_nonlinear(n_feature=p, n_hidden1=n_hidden1, n_hidden2=n_hidd
en2, n_output=2)
# Define another model to save the current best model based on loss func
tion value
# The purpose is to prevent divergence of the training due to large lear
ning rate or other reason
best_model = Net_nonlinear(n_feature=p, n_hidden1=n_hidden1, n_hidden2=n
_hidden2, n_output=2)


# Define optimizers for the optimization with given support
# optimizer to separately optimize the hidden layers and selection layer
s
# the selection layer will be optimized on given support only.
# the optimzation of hidden layers and selection layer will take turn in
iterations
optimizer = torch.optim.Adam(list(model.parameters()), lr=learning_rate,
weight_decay=0.0025*c)
optimizer0 = torch.optim.Adam(model.hidden0.parameters(), lr=learning_ra
te, weight_decay=0.0005*c)


# Define loss function
lf = torch.nn.CrossEntropyLoss()


# Allocated some objects to keep track of changes over iterations
hist = []
SUPP = []
LOSSES = []
supp_x = list(range(p)) # initial support
SUPP.append(supp_x)


### DFS algorithm
start = clock()
for i in range(epochs):
    # One DFS epoch
    model, supp_x, LOSS = DFS_epoch(model, s, supp_x, X, Y, lf, optimize
r0, optimizer, Ts, step)
    LOSSES = LOSSES + LOSS
    supp_x.sort()
```

```
    # Save current loss function value and support
    hist.append(lf(model(X), Y).data.numpy().tolist())
    SUPP.append(supp_x)
    # Prevent divergence of optimization over support, save the current
 best model
    if hist[-1] == min(hist):
        best_model.load_state_dict(model.state_dict())
        best_supp = supp_x
    # Early stop criteria
    if len(SUPP[-1]) == len(SUPP[-2]) and (SUPP[-1] == SUPP[-2]).all():
        break

end = clock()
print("Training finished in" , len(SUPP)-1, "epochs, and took", end-star
t, "seconds")
```

```
Training finished in 2 epochs, and took 229.32999999999998 seconds
```

In the following chunk, we will demonstrate the results from the DFS algorithm, in terms of selected support, training error and test error for **one step** procedure.

In [4]:

```
### Metric calculation
err_train_1 = 1-accuracy(best_model, X, Y)
err_test_1 = 1-accuracy(best_model, X_test, Y_test)
print("The support selected is:", best_supp)
print("The index of non-zero coefficents on selection layer:",
      np.where(best_model.hidden0.weight != 0)[0])
print("The training error is:", err_train_1)
print("The test error is:", err_test_1)
```

```
The support selected is: [0 1 2 3]
The index of non-zero coefficents on selection layer: [0 1 2 3]
The training error is: 0.010000000000000009
The test error is: 0.043333333333333335
```

From the results above, we have successfully selected the right support, i.e. the first 4 variables. (Note in `python` starting index is 0)

In the following chunk, we will perform a two-step procedure to train the `best_model` on the given support.

Two-step procedure is used for two reasons, to get better predictive performance and to get better estimation of bic which is important in selection of optimal s.

As we demonstrated on the above chunk, the selection layer of `best_model` has non-zero coefficients on given support. In the second step, we treat `best_model` as our initial model and update parameters only in hidden layer.

In [5]:
```
# Define optimizer only update parameters in hidden layer.
_optimizer = torch.optim.Adam(list(best_model.parameters())[1:], lr=0.01
, weight_decay=0.0025)
# Training
for _ in range(100):
    out = best_model(X)
    loss = lf(out, Y)
    _optimizer.zero_grad()
    loss.backward()
    _optimizer.step()

### metric calculation
acc_train = accuracy(best_model, X, Y)
acc_test = accuracy(best_model, X_test, Y_test)
print("The training accuracy of two step is: ", acc_train*100, "%", sep=
"")
print("The test accuracy of two step is: ", acc_test*100, "%", sep="")
```

```
The training accuracy of two step is: 100.0%
The test accuracy of two step is: 96.0%
```

The result has shown that the predictive performance of our model is increased.

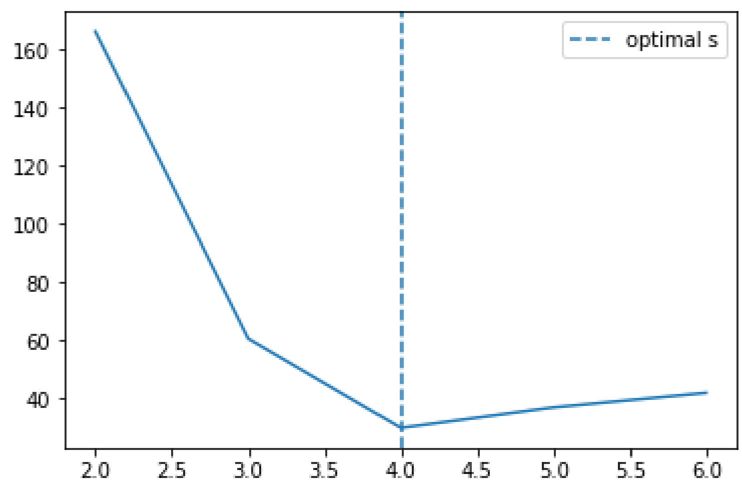
All good results shown above is based on the correct given s. However, in reality, s is unknown for most of the time. So the next thing would be finding the optimal s

Selection of s

In this section, we demonstrate the procedure of selection of optimal s. We have wrapped up the training procedure above in a function `training_n`. For each given s, bic, defined as $-2 \cdot \log \hat{L} + s \cdot \log n$, of the model will be automatically calculated by `training_n`, also the trained model with the given s will also be returned.

In [6]:
```
Ss = list(range(2, 7)) # We shorten the candidates list in the notebooks
BIC = [] # Store the bic for different s
best_model = Net_nonlinear(n_feature=p, n_hidden1=n_hidden1, n_hidden2=n
_hidden2, n_output=2)
for i, s in enumerate(Ss):
    # Training dataset k with given s
    model, supp, bic, _, [err_train, err_test] = training_n(X, Y, X_test
, Y_test, c, s,
                                                            epochs=10, T
s=25)
    # Store bic values
    BIC.append(bic)
    # if current bic is the smallest, save the trained model, support an
d other metric
    if bic == min(BIC):
        best_model.load_state_dict(model.state_dict())
        best_supp = supp
        best_err_train, best_err_test = err_train, err_test # one step m
odel training and testing error

idx = np.argmin(BIC)
best_s = Ss[idx]
plt.plot(Ss, BIC)
plt.axvline(x=best_s, ls='--', label="optimal s")
plt.legend()
plt.show()
```



From the graph above, we can tell $s = 4$ is the optimal s, and the corresponding model is stored in `best_model` which is the same model showed in section 1.2