Exercises for the course
**Artifical Intelligence**
Summer Semester 2024

G. Montavon
Institute of Computer Science
**Department of Mathematics and Computer Science**
Freie Universität Berlin

# Exercise Sheet 9

In [1]:
```
import numpy
import utils
import torch
import scipy
import scipy.signal

na = numpy.newaxis
```

## Exercise 1: Learning a Neural Network Classifier (20 P)

We would like to build a machine learning model that classifies breast cancer samples into malignant or benign. For this, we will make use of the breast cancer dataset provided as part of scikit-learn, and which we make available in verious forms from the class `utils.Environment`.

In [2]:
```
env = utils.Environment()

X,T = env.get_data()
```

The dataset comes in the form of a data matrix $X$ of size $N \times d$, where $N$ is the number of data points and $d$ is the number of features, and a vector $T$ of size $N$ containing the class associated to each data point (encoded as $-1$ for malignant or $+1$ for benign). The dataset is already normalized and it can be directly fed to a neural network. A neural network can be generated with the function below, where the argument determines the number of outputs.

In [3]:
```
utils.getnn(1)
```

Out[3]:
```
Sequential(
  (0): Linear(in_features=30, out_features=10, bias=True)
  (1): ReLU()
  (2): Linear(in_features=10, out_features=1, bias=True)
)
```

The output of a neural network produces a real value representing the predicted log-likelihood ratio between the two classes. As shown in the slides, the likelihood of the data in the light of the neural network prediction can be maximized by solving the following optimization problem:

$$\max_{\theta} \frac{1}{N} \sum_{i=1}^{N} -\log(1+\exp(-tz))$$

where $\theta$ are the parameters of the neural network. There are two problems with this formulation. First, the PyTorch optimization framework considers the *minimization* of objectives. Furthermore, the exponential function is numerically instable for large input values. Thus, one first observes that the function $\log(1+\exp(t))$ is also known as the softplus function, and the latter comes with more stable implementation. Also, maximizing an objective is equivalent to minimizing the negative of the same objective. This gives us the optimization problem:

$$\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \text{softplus}(-tz)$$

We now would like to minimize this objective using gradient descent. For this, we first need to implement the error function. Note that for gradient descent to work, your implementation should be differentiable (i.e. all computations expressed in PyTorch, without conversion to numpy).

In [4]:
```
from torch.nn.functional import softplus

def getE(D,nn):
    # Unpack the dataset into a matrix of inputs and a vector o
    X,T = D
    z = nn(X)[:, 0] # if we don't reduce the dimensions from (1
```

```
    E = softplus(-T * z).mean()
    return E
```

The code below runs a gradient descent procedure, at the end of which the neural network becomes capable of accurately solving the classification task.

In [5]:
```
nn = utils.train(env.get_data(),utils.getnn(1),getE,[1,10,100,1
```

```
   1 0.654
  10 0.448
 100 0.090
1000 0.016
```

# Exercise 2: Extracting Probabilities (20 P)

We would like to integrate the prediction of our neural network (the log-likelihood ratio, and the class probabilities that can be derived from it) into a Bayes inference procedure. Unlike the previous Bayesian inference exercises, we now have $N$ data points to process. We will therefore work with multidimensional arrays where the first dimensions is reserved for representing the various data points. As a reminder, the class probability can be extracted from the log-likelihood ratio via the sigmoid function:

$$p(t=-1|x) = \mathrm{sigm}(-z)$$
$$p(t=+1|x) = \mathrm{sigm}(+z)$$

Your task is to implement a function `getP` that receives a vector of $N$ log-likelihood ratios (the output of the network) and conver them into probabilities for each class. The output of this function should be an array of size $N \times 2$.

In [6]:
```
def getP(Z):

    Z = Z.data.numpy()
    Z = Z[:, na]
    P = numpy.concatenate(
        [
            utils.sigm(-Z),
            utils.sigm(Z),
        ],
```

```
        axis=1
    )

    return P
```

# Exercise 3: Taking Optimal Decisions (20 P)

We now would like to infer for each data point what is the action that delivers the highest expected utility. The utility of performing specific actions depends on the actual class of the data point. Such relation between class and actions is assumed to be known and given in the following table.

| | $t = -1$ (malignant) | $t = +1$ (benign) |
|---|---|---|
| $a = 0$ (malignant) | $0$ | $-1$ |
| $a = 1$ (benign) | $-5$ | $0$ |
| $a = 2$ (retrain) | $-0.25$ | $-0.25$ |

The content of this table is stored in the array `utils.R`. Your task is to implement a function that generates an vector `A` of size $N$ containing for each data point the best action to take (i.e. the action that maximizes the expected utility, where `P` is the array storing class probabilities for each data point).

In [7]:
```
def getA(P):
    # we transpose the matrix to get the classes as the first d
    A = P @ utils.R.T
    A = A.argmax(axis=1)

    return A
```

We can now focus on the utility one gets by taking the actions above. As a first option, we can look at the utility as estimated from the the model (i.e. according the the class probabilities the model produces).

In [8]:
```
X,T = env.get_data()

P = getP(nn.forward(X)[:,0])        # get class probabilities
A = getA(P)                         # generate rational actions
```

```
print('Predicted utility: %.3f'%(utils.R[A]*P).sum(axis=1).mean
```

```
Predicted utility: -0.022
```

Alternatively, we can look at some empirical measure of utility by applying the model above on some new data points, and return the utility one gets on average.

In [9]:
```
X,T = env.get_test_data()

P = getP(nn.forward(X)[:,0])        # get class probabilities
A = getA(P)                         # generate rational actions
Y = numpy.eye(2)[(T*.5+.5).long()] # one-hot encoding of target

print('Actual utility: %.3f'%(utils.R[A]*Y).sum(axis=1).mean()
```

```
Actual utility: -0.094
```

We observe that the actual utility is worse than what the model predicts. This effect can be attributed to overfitting: the model has become overconfident in its own predictions, and fails to generate sensible probability scores for data points it has not seen before.

# Exercise 4: Learning with Utility Targets (20 P)

So far, we have combined a standard logistic classifier with some action mechanism selection based on some known utility function. We will now consider the scenario where the data comes in the form of observations along with a particular action and associated utility. We will consider a neural network that predicts utilities associated to various observation-action pairs. In particular, we will consider a neural network with three outputs predicting the utility associated to all three possible actions.

The training objective is then to minimize the squared error between the predicted utility of the given action and the true utility associated to this action. Your task is to implement this error function. As for the first exercise, your implementation of the error function should be differentiable so that gradient descent procedure can be easily applied.

In [10]:
```
mse = torch.nn.MSELoss()

def getE2(D,nn):

    X,A,U = D

    pred_U = nn(X)

    # select only the predictions for the used actions
    relevant_pred_U = pred_U[range(A.shape[0]), A]

    E = ((relevant_pred_U - U) ** 2).mean()
    return E
```

A model that minimizes the objective you have implemented is obtained by running the code below.

In [11]:
```
nn = utils.train(env.get_data_xau(),utils.getnn(3),getE2,[1,10,
```

```
   1 2.971
  10 0.517
 100 0.148
1000 0.038
```

As expected, the utility predictions become highly accurate after a sufficient number of training steps. Similarly to what we have done before, we can either display the associated utility as predicted by the model itself, or we can measure the actual utility on some test set different from the one used for training.

In [12]:
```
X,T = env.get_data()

U = nn.forward(X).amax(axis=1)      # maximal utility

print('Predicted utility: %.3f'%U.mean())


X,T = env.get_test_data()

A = nn.forward(X).argmax(axis=1)   # action that maximizes util
Y = numpy.eye(2)[(T*.5+.5).long()] # one-hot encoding of target
U = (utils.R[A]*Y).sum(axis=1)

print('Actual utility: %.3f'%U.mean())
```

```
Predicted utility: 0.059
Actual utility: -0.083
```

We observe that the new model performs on par with the approach based on logistic regression.

# Exercise 5: Reinforcement Learning (20 P)

We will now consider the setting where there are no labels or reward information that is available a priori. Instead, the agent needs to perform actions in the environment in order to understand the utility associated to these actions. Our reinforcement learning procedure we will consist of 20 iterations. At each iteration, the agent receives a batch of 20 examples. It needs to take actions for each of then, and then receive receive a reward from these actions. The agent can in a second step use this newly acquired ground-truth to extend the dataset and improve the model (e.g. by retraining it). The overall reinforcement learning procedure is given below. The part consisting of taking action on the new data points and extracting utilities is left blank and needs to be implemented. Your implementation should

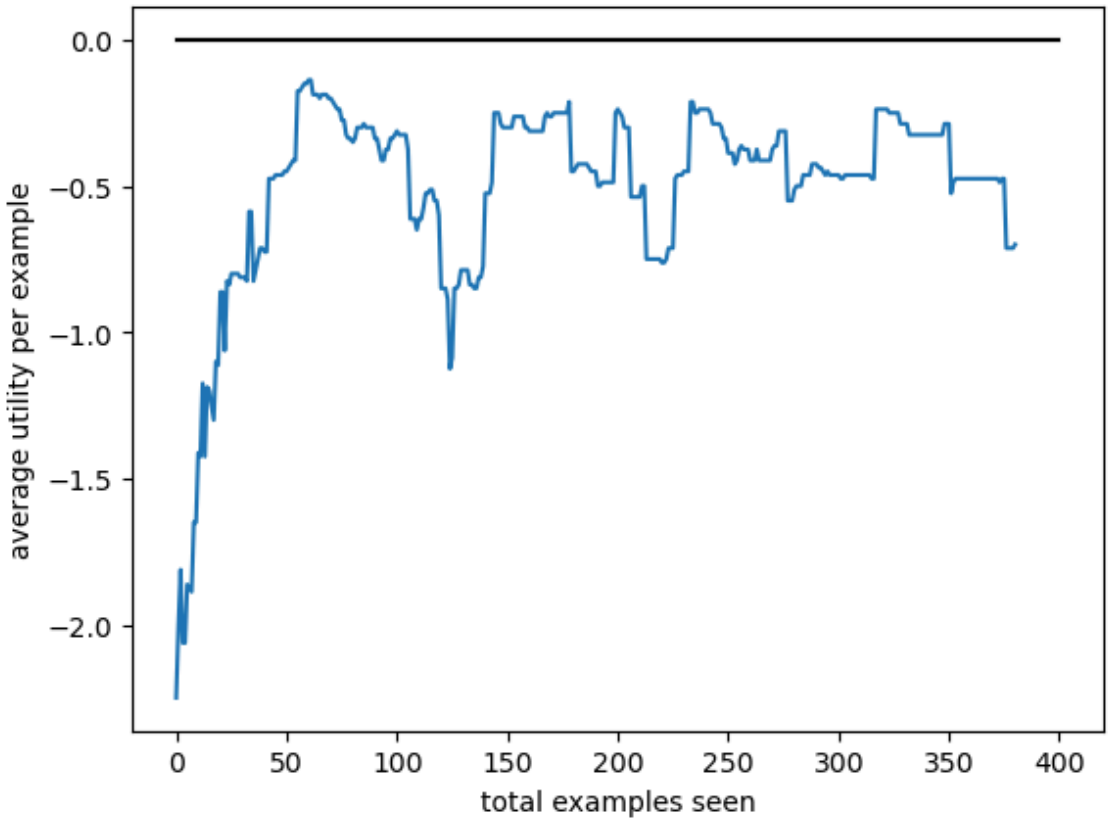
make use of the functions `get_new_batch()` and `act_and_get_utility(a)` of the class `Environment`.

In [23]:
```
import matplotlib
from matplotlib import pyplot as plt

env = utils.Environment()

nn = utils.getnn(3)

X = torch.tensor([])
A = torch.tensor([]).long()
U = torch.tensor([])

for i in range(20):

    # =========================================================
    # STEP 1: Carry and action and get feedback
    # =========================================================

    x = env.get_new_batch()
    pred_u = nn(x).detach().numpy()
    a = numpy.argmax(pred_u, axis=1)
    u = env.act_and_get_utility(a)

    a = torch.tensor(a)
    u = torch.tensor(u)

    # =========================================================
    # STEP 2: Augment and retrain
    # =========================================================
    X = torch.cat((X,x))
    A = torch.cat((A,a))
    U = torch.cat((U,u))

    nn = utils.train((X,A,U),utils.getnn(3),getE2,[1000])
```

```
1000 0.000
1000 0.000
1000 0.002
1000 0.000
1000 0.000
1000 0.004
1000 0.005
1000 0.013
1000 0.012
1000 0.024
1000 0.037
1000 0.043
1000 0.077
1000 0.079
1000 0.062
1000 0.178
1000 0.101
1000 0.066
1000 0.093
1000 0.050
```

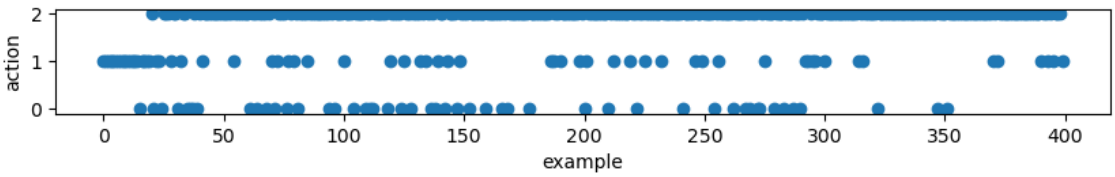
The results of the training process can be analyzed. In particular, we can show the utility values that have been produced at each time step. The latter indicate the overall gain/cost the agent has experienced over the different stages of data acquisition and training. The plot is smoothed for easier visualization.

In [24]:
```
plt.plot([0,len(U)],[0,0],color='black')
plt.plot(scipy.signal.convolve(U.data,numpy.ones([20])/20,'val
plt.xlabel('total examples seen')
plt.ylabel('average utility per example')
plt.show()
```




We observe that in the first few iterations, there have been a rather low utility per data point. It seems that the model was not able initially to take appropriate actions. After a few iterations, the network learns a better policy, and the utility improves. However, the curve does not reach the level of utility we have obtained in the previous exercises. To shed light on this limitation, we can look at the actual actions throughout training.

In [25]:
```
plt.figure(figsize=(10,1))
plt.scatter(numpy.arange(len(A)),A)
plt.xlabel('example')
plt.ylabel('action')
plt.yticks([0,1,2])
plt.show()
```




We observe that the model is often choosing the failback action 2 which is to refrain from classifying. Such fallback on this "safe" action, which is never bad but also not optimal, can be explained by the absence of a specific exploration mechanism in our procedure.

In [ ]: