Deep Neural Networks: Forward and Backward Propagation

In this homework, our goal is to test different approaches to program neural networks, from the simplest and least general one to the most complex and general. Here, we will be focusing on programming forward and gradient computations. Training neural networks will be left for Homework 2. The first implementation we consider is a hard-coded neural network made of one input layer, three hidden layers, and one output layer. We use the ReLU nonlinearity at each hidden layer. The neural network is depicted below:

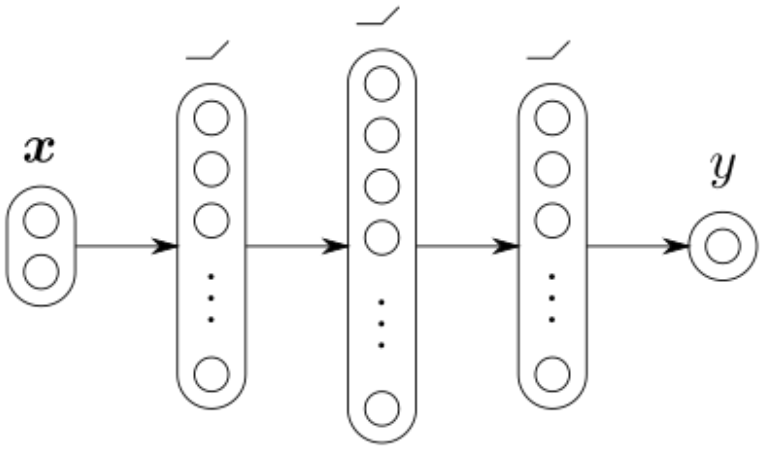

Figure 1:

The following implementation performs forward and gradient computations using `for` loops. The code is compact but only works for this specific architecture. It is hard to extend it to new layers without significantly refactoring and complexifying the code.

In [1]:
```
import numpy,numpy.linalg

def implementation1(X,T,W,B):

    # 1. Initialize some data structures
    A,Z = [X],[]
    DW,DB = [],[]

    # 2. Run the forward pass
    for i,(w,b) in enumerate(zip(W,B)):
        Z.append(A[-1].dot(w)+b)
        if i in [0,1,2]: A.append(numpy.maximum(0,Z[-1]))
    Y = Z[-1]

    # 3. Compute the error
    Y = Z[-1]
    err = ((Y-T)**2).mean()
    grad = 2*(Y-T)

    # 4. Gradient propagation
    for w,b,a in zip(W[::-1],B[::-1],A[::-1]):
        DW.insert(0,a.T.dot(grad)/len(a))
```

```
            DB.insert(0,grad.mean(axis=0))
            grad = grad.dot(w.T)*(a>0)

        # 5. Return error and gradient
        return err,DW
```

The code below applies this function to a given dataset `X,T` at a given position `W,B` in parameter space, where these two variables contain the weight and bias parameters at each layer. It then prints the prediction mean square error, and the gradient norm at each layer.

In [2]:
```
import utils

X,T = utils.getdata(100)
W,B = utils.getparameters([X.shape[1],10,15,10,T.shape[1]])

err,DW = implementation1(X,T,W,B)

print(err,map(numpy.linalg.norm,DW))
```

```
(1.1593903177325451, [0.29622485370120832, 0.6949589001193901, 1.3754967914824301, 0.80223630277879099]
```

These numbers are not specially interesting (the network has not been trained), however, they are useful for debugging purposes, in order to verify that the next implementations work as expected.

Object-Oriented Implementation (15 P)

The following implmentation adopts an object oriented approach to the forward and gradient computations. Each layer is an object with methods implementing the forward and backward pass for this layer. Objects are defined in the file `layers.py`. Overall, the code is longer, but it is better structured and easier to extend.

In [3]:
```
import layers

def implementation2(X,T,W,B,NonLin):

    # 1. Build the neural network
    nnlayers = []
    for w,b in zip(W,B): nnlayers += [layers.Linear(w,b)] + ([NonLin()] if len(b)!=1 else [])
    nn = layers.Sequential(nnlayers)

    # 2. Compute the error and its gradient
    Y = nn.forward(X)
    err = ((Y-T)**2).mean()
    nn.backward(2*(Y-T))

    # 3. Return them
    return err,[nn.layers[i].DW for i in [0,2,4,6]]
```

The code below computes and prints the prediction error, and the gradient norm at each layer.

In [4]:
```
import utils

err,DW = implementation2(X,T,W,B,layers.Tanh)
print(err,map(numpy.linalg.norm,DW))
```

```
(1.0609541173391692, [0.24642012799448015, 0.49094047912050492, 0.66230622343522649, 0.4731456439941019
```

Here, although the data and parameters have not been changed, the numbers are different as for `implementation1`, as we have now made use of the `Tanh` nonlinearity instead of `ReLU`.

Tasks:

- **Define a new layer `ReLU` to be used in replacement to the `Tanh` layer. This makes the architecture equivalent to the neural network implemented by the function `implementation1`.**
- **Run the code below to verify that the error and gradient are indeed the same for `implementation1` and `implementation2`.**

In [5]:
```
# ------------------------------------
# TODO: REPLACE BY YOUR OWN CODE
# ------------------------------------
import solution; ReLU = solution.ReLU
# ------------------------------------

err,DW = implementation1(X,T,W,B);      print(err,map(numpy.linalg.norm,DW))
err,DW = implementation2(X,T,W,B,ReLU); print(err,map(numpy.linalg.norm,DW))
```

```
(1.1593903177325451, [0.29622485370120832, 0.6949589001193901, 1.3754967914824301, 0.80223630277879099])
(1.1593903177325451, [0.29622485370120832, 0.6949589001193901, 1.3754967914824301, 0.80223630277879099])
```

Implementation for General Graphs (15 P)

The implementation below is more complex but is also applicable to a broader set of structures than simple feed-forward network. Here, the neural network can be seen as a set of nodes (defined in `nodes.py`) that are organized in a graph. Prediction and computation of gradients are obtained by traversing the graph using recursion.

In [6]:
```
import nodes

def implementation3(X,T,W,B):

    # 1. Build the neural network
    W,B = utils.getparameters([X.shape[1],10,15,10,T.shape[1]])

    nodeX = nodes.Input()
    nodeA = nodeX
    nodesW = [nodes.Weight(w) for w in W]
    nodesB = [nodes.Bias(b) for b in B]

    for i,(nodeW,nodeB) in enumerate(zip(nodesW,nodesB)):
        nodeZ = nodes.Linear(nodeA,nodeW,nodeB)
        if i in [0,1,2]: nodeA = nodes.Tanh(nodeZ)
    nodeY = nodes.Output(nodeZ)

    # 2. Compute the error and its gradient
    nodeX.feed(X)
    Y = nodeY.evaluate()
    err = ((Y-T)**2).mean()
    nodeY.feed(2*(Y-T))

    # 3. Return them
    return err,[nodeW.grad() for nodeW in nodesW]
```

The code below applies the new implementation on the same dataset and parameters as before, and the error and gradients are compared for correctness to those of `implementation2`.

```
In [7]: err,DW = implementation2(X,T,W,B,layers.Tanh); print(err,map(numpy.linalg.norm,DW))
        err,DW = implementation3(X,T,W,B);             print(err,map(numpy.linalg.norm,DW))

(1.0609541173391692, [0.24642012799448015, 0.49094047912050492, 0.66230622343522649, 0.4731456439941019
(1.0609541173391692, [0.24642012799448015, 0.49094047912050492, 0.66230622343522649, 0.4731456439941019
```

We now would like to modify the neural network architecture by including a shortcut connection. Shortcut connections can be useful when the prediction requires a combination of simple and more abstract features.

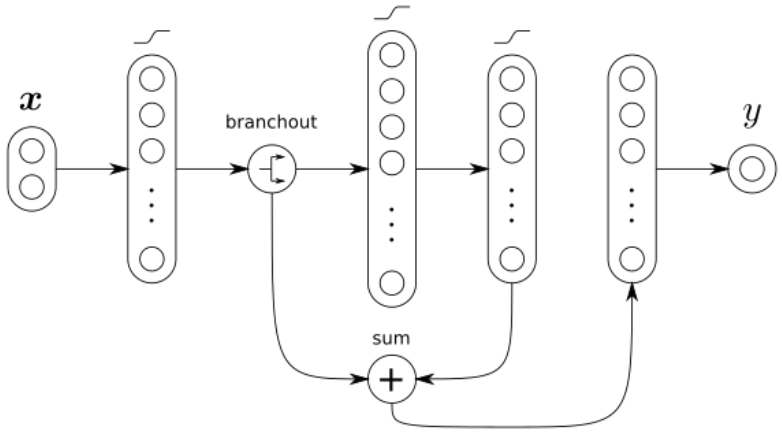

Figure 2:

The implementation below assumes two types of nodes, `Sum` and `BranchOut`. Both are needed to implement the network above.

```
In [8]: def implementation3B(X,T,W,B,Sum,BranchOut):

            # 1. Build the neural network
            W,B = utils.getparameters([X.shape[1],10,15,10,T.shape[1]])

            nodeX = nodes.Input()
            nodesW = [nodes.Weight(w) for w in W]
            nodesB = [nodes.Bias(b) for b in B]

            nodeZ1 = nodes.Linear(nodeX,nodesW[0],nodesB[0])
            nodeA1 = nodes.Tanh(nodeZ1)
            nodeQ1 = BranchOut(nodeA1)

            nodeZ2 = nodes.Linear(nodeQ1,nodesW[1],nodesB[1])
            nodeA2 = nodes.Tanh(nodeZ2)
            nodeZ3 = nodes.Linear(nodeA2,nodesW[2],nodesB[2])
            nodeA3 = nodes.Tanh(nodeZ3)
            nodeS3 = Sum([nodeQ1,nodeA3])

            nodeZ4 = nodes.Linear(nodeS3,nodesW[3],nodesB[3])
            nodeOut = nodes.Output(nodeZ4)

            # 2. Compute the error and its gradient
            nodeX.feed(X)
            Y = nodeOut.evaluate()
```

```
        err = ((Y-T)**2).mean()
        nodeOut.feed(2*(Y-T))

        # 3. Return them
        return err,[nodeW.grad() for nodeW in nodesW]
```

Tasks:

- **Create the nodes `Sum` and `BranchOut` needed for implementing the new architecture (i.e. define two new classes, and implement for each class the required methods).**
- **Run the code below to display the error and gradient information for the dataset and current parameters.**

In [9]:
```
## ---------------------------
## REPLACE BY YOUR OWN CODE
## ---------------------------
import solution
Sum = solution.Sum
BranchOut = solution.BranchOut
## ---------------------------

err,DW = implementation3B(X,T,W,B,Sum,BranchOut)
print(err,map(numpy.linalg.norm,DW))
```

```
(1.1922440307543112, [0.52149785222097844, 0.85734586001317348, 1.4576722381826273, 1.6381730369377567]
```