```python
# File main.py:
# Defines activation and error functions
# Initializes and trains networks

import numpy as np
import mycnn # See mycnn.py
import time

trial = 6

def newFunc(z):
    return (z+np.sqrt(z**2+1))*0.5
def newFunc2(z):
    return (z+np.sqrt(z**2+0.1))*0.5
def newFunc3(z):
    return (z+np.sqrt(z**2+0.01))*0.5
def newFuncPrime(z, a):
    return a/(2*a-z)

def relu(z):
    return np.maximum(z, 0)
def reluPrime(z, a):
    return (np.sign(z)+1)/2

def softplus(z):
    return np.log(1+np.exp(z))
def softplusPrime(z, a):
    return 1/(1+np.exp(-z))

def logistic(z):
    return 1/(1+np.exp(-z))
def logisticPrime(z, a):
    return a*(1-a)

def crossEntropy(y, yHat):
    return -np.sum(y*np.log(yHat))
def crossEntropyPrime(y, yHat):
    return -y/yHat

def arraysFromFiles(XFile, yFile, b):

    XString = XFile.read(b*28*28)
    YString = yFile.read(b)
    if XString == '':
        raise ValueError

    xCrop = np.random.randint(5)
    yCrop = np.random.randint(5)

    XRaw = np.fromstring(XString, dtype=np.uint8)
    XShaped = np.transpose(XRaw.astype(float).reshape((b, 1, 28, 28)), (2, 3, 1, 0))
    newX = XShaped[yCrop:yCrop+24,xCrop:xCrop+24]/256

    YLabels = np.fromstring(YString, dtype=np.uint8)
    newY = np.zeros((10, b))
    newY[YLabels, np.arange(b)] = 1
    return newX, newY

def getMNIST(b):

    XTrain = open('../data/train-images-idx3-ubyte', 'r')
    yTrain = open('../data/train-labels-idx1-ubyte', 'r')
    XTest = open('../data/t10k-images-idx3-ubyte', 'r')
    yTest = open('../data/t10k-labels-idx1-ubyte', 'r')

    XTrain.seek(16)
    yTrain.seek(8)
    XTest.seek(16)
    yTest.seek(8)

    t = True
    XFile = XTrain
    yFile = yTrain

    while True:
        try:
            newX, newY = arraysFromFiles(XFile, yFile, b)
            yield newX, newY, t
        except ValueError:
            XFile.seek(16)
            yFile.seek(8)
```

```
            t = not t
            if t:
                XFile = XTrain
                yFile = yTrain
            else:
                XFile = XTest
                yFile = yTest

#x = [[newFunc, newFuncPrime], [relu, reluPrime], [softplus, softplusPrime], [logistic, logisticPrime]]
x = [[newFunc2, newFuncPrime], [newFunc3, newFuncPrime]]

netList = []

numNets = 2
batch = 20
testBatchesPerEpoch = 10000/batch
numEpochs = 50

accuracyData = np.ndarray((numEpochs, numNets))
times = np.ndarray(numNets)

for a in range(numNets):
    allLayers = [{'type': 'conv', 'kernel':   5, 'neurons': 8},
                 {'type': 'act',  'f': x[a][0], 'fPrime': x[a][1]},
                 {'type': 'pool', 'factor':   2},
                 {'type': 'conv', 'kernel':   5, 'neurons': 16},
                 {'type': 'act',  'f': x[a][0], 'fPrime': x[a][1]},
                 {'type': 'pool', 'factor':   2},
                 {'type': 'fc',   'shape':  (10,)},
                 {'type': 'softmax'}]
    newNet = mycnn.CNN(24, 1, batch, crossEntropy, crossEntropyPrime)
    netList.append(newNet)
    for layer in allLayers:
        newNet.addLayer(layer)

    numIterations = 0
    correctInEpoch = 0


    start = time.time()
    for datum in newNet.train(getMNIST):
        correctInEpoch += datum
        numIterations += 1
        if numIterations%testBatchesPerEpoch == 0:
            currentEpoch = numIterations/testBatchesPerEpoch
            accuracyData[currentEpoch-1][a] = correctInEpoch
            correctInEpoch = 0
        if numIterations == testBatchesPerEpoch*numEpochs:
            times[a] = time.time()-start
            break

np.savetxt('../results/accuracy_data_%d.txt'%trial, accuracyData, delimiter=',', fmt='%05u')
np.savetxt('../results/times_%d.txt'%trial, times, fmt='%05u')
```

```
# File mycnn.py:
# Defines the convolutional network class
# Implements forward and backpropagation at the layer level

import numpy as np
import cnnfunctions # See cnnfunctions.py

class CNN:

    def __init__(self, inSize, inNeurons, batchSize, lossFunc, lossPrime):

        inShape = (inSize, inSize, inNeurons, batchSize)

        self.L = [np.ndarray(inShape)]
        self.W = []
        self.b = []

        self.dJdL = [np.ndarray(inShape)]
        self.dJdW = []
        self.dJdb = []

        self.numLayers = 0

        self.dJdWMS = []
        self.WStep = []
        self.WStepMS = []

        self.dJdbMS = []
        self.bStep = []
        self.bStepMS = []

        self.momentum = 0.9
        self.epsilon = 1e-4
        self.decay = 1e-4

        self.params = []

        self.batchSize = batchSize
        self.lossFunc = lossFunc
        self.lossPrime = lossPrime

    def randomize(self):
        for l in range(self.numLayers):
            param = self.params[l]['type']
            normFactor = 1
            if param == 'conv':
                normFactor = kernel*kernel*lastNeurons
            elif param == 'fc':
                normFactor = np.prod(lastShape)/self.batchSize

    def addLayer(self, param):

        LType = param['type']
        lastShape = self.L[-1].shape

        # Default values
        LShape = list(lastShape)
        WShape, bShape = 0, 0

        normFactor = 1

        if LType == 'conv':
            kernel = param['kernel']
            neurons = param['neurons']
            lastNeurons = lastShape[2]
            normFactor = kernel*kernel*lastNeurons
            assert kernel%2 == 1
            LShape[2] = neurons
            WShape = (kernel, kernel, lastNeurons, neurons)
            bShape = (neurons, 1)

        elif LType == 'act':
            pass

        elif LType == 'pool':
            factor = param['factor']
            assert lastShape[0]%factor == 0
            LShape[0] /= factor
            assert lastShape[1]%factor == 0
            LShape[1] /= factor
```

```
        elif LType == 'fc':
            shape = param['shape']
            LShape = shape+(self.batchSize,)
            WShape = lastShape[:-1]+shape
            bShape = shape+(1,)
            normFactor = np.prod(lastShape)/self.batchSize

        elif LType == 'softmax':
            assert len(lastShape)==2

        else:
            print "Invalid layer."

        self.L.append(np.ndarray(LShape))
        self.W.append(np.random.random(WShape)*2/normFactor)
        self.b.append(np.random.random(bShape)-0.5)

        self.dJdL.append(np.ndarray(LShape))
        self.dJdW.append(np.ndarray(WShape))
        self.dJdb.append(np.ndarray(bShape))

        # MS = mean square
        self.dJdWMS.append(np.zeros(WShape))
        self.WStep.append(np.zeros(WShape))
        self.WStepMS.append(np.zeros(WShape))

        self.dJdbMS.append(np.zeros(bShape))
        self.bStep.append(np.zeros(bShape))
        self.bStepMS.append(np.zeros(bShape))

        self.params.append(param)

        self.numLayers += 1

    def forward(self, X):
        self.L[0][:] = X
        for l in range(self.numLayers):
            param = self.params[l]
            LType = param['type']
            if LType == 'conv':
                cnnfunctions.conv(self.L[l], self.W[l], self.L[l+1], bias=self.b[l])
            elif LType == 'pool':
                cnnfunctions.pool(self.L[l], param['factor'], self.L[l+1])
            elif LType == 'act':
                cnnfunctions.act(self.L[l], param['f'], self.L[l+1])
            elif LType == 'fc':
                cnnfunctions.fc(self.L[l], self.W[l], self.b[l], self.L[l+1])
            elif LType == 'softmax':
                cnnfunctions.softmax(self.L[l], self.L[l+1])
            else:
                raise ValueError('Invalid layer type.')

    def backward(self, dJdyHat):
        self.dJdL[-1][:] = dJdyHat
        for m in range(self.numLayers, 0, -1):
            l = m-1
            param = self.params[l]
            LType = param['type']
            if LType == 'conv':
                cnnfunctions.conv(self.dJdL[l], self.W[l], self.dJdL[l+1], forward=False, inL=self.L[l],
WDeriv=self.dJdW[l], bDeriv=self.dJdb[l])
            elif LType == 'pool':
                cnnfunctions.scale(self.dJdL[l], param['factor'], self.dJdL[l+1])
            elif LType == 'act':
                cnnfunctions.actPrime(self.dJdL[l], self.L[l], param['fPrime'], self.L[l+1], self.dJdL[l+1])
            elif LType == 'fc':
                cnnfunctions.fcTranspose(self.dJdL[l], self.L[l], self.W[l], self.dJdL[l+1], self.dJdW[l],
self.dJdb[l])
            elif LType == 'softmax':
                cnnfunctions.softmaxPrime(self.dJdL[l], self.L[l+1], self.dJdL[l+1])
            else:
                raise ValueError('Invalid layer type.')
                break

            cnnfunctions.adadelta(self.dJdW[l], self.dJdWMS[l], self.WStep[l], self.WStepMS[l], self.momentum,
self.epsilon)
            self.W[l] += self.WStep[l]
            self.W[l] *= 1-self.decay

            cnnfunctions.adadelta(self.dJdb[l], self.dJdbMS[l], self.bStep[l], self.bStepMS[l], self.momentum,
self.epsilon)
```

```
            self.b[l] += self.bStep[l]
            self.b[l] *= 1-self.decay

    def train(self, getBatches):
        iterations = 0
        for X, y, t in getBatches(self.batchSize):
            self.forward(X)
            yHat = self.L[-1]
            if t:
                dJdyHat = self.lossPrime(y, yHat)
                self.backward(dJdyHat)
            else:
                pred = np.argmax(yHat, axis=0)
                actual = np.argmax(y, axis=0)
                numRight = np.sum(pred==actual)
                yield numRight
```

```
import numpy as np

def conv(inT, weight, outT, forward=True, bias=None, inL=None, WDeriv=None, bDeriv=None):
    inShape = inT.shape
    inHeight, inWidth = inShape[:2]

    WShape = weight.shape
    WHeight, WWidth = WShape[:2]

    YBorder = (WHeight-1)/2
    XBorder = (WWidth-1)/2

    if forward:
      outT.fill(0)
    else:
      inT.fill(0)

    for y in np.arange(WHeight):
        yOffset = y-YBorder
        inTop = max(-yOffset, 0)
        outTop = max(yOffset, 0)
        inBottom = inHeight+min(-yOffset, 0)
        outBottom = inHeight+min(yOffset, 0)

        for x in np.arange(WWidth):
            xOffset = x-XBorder
            inLeft = max(-xOffset, 0)
            outLeft = max(xOffset, 0)
            inRight = inWidth+min(-xOffset, 0)
            outRight = inHeight+min(xOffset, 0)

            weightPixel = weight[y, x]
            inSec = inT[inTop:inBottom, inLeft:inRight]
            outSec = outT[outTop:outBottom, outLeft:outRight]

            if forward:
              outSec += np.einsum('ijkm,kl', inSec, weightPixel)
            else:
              inSec += np.einsum('ijlm,kl', outSec, weightPixel)
              np.einsum('ijkm,ijlm', inL[inTop:inBottom, inLeft:inRight], outSec, out=WDeriv[y, x])

    if forward:
        outT += bias
    else:
        np.sum(outT, axis=(0, 1, 3), out=bDeriv[:, 0])

def act(inL, f, outL):
    outL[:] = f(inL)

def actPrime(dJdinL, inL, fPrime, outL, dJdoutL):
    dJdinL[:] = dJdoutL * fPrime(inL, outL)

def pool(inL, factor, outL):
    inShape = inL.shape
    outL[:] = np.average(
      inL.reshape(
        (inShape[0]/factor, factor, inShape[1]/factor, factor, inShape[2], inShape[3])),
      axis=(1, 3))

def scale(dJdinL, factor, dJdoutL):
    dJdinL[:] = np.kron(dJdoutL, np.full((factor, factor, 1, 1), factor**-2))

def fc(inL, weight, bias, outL):
    inDims = np.arange(inL.ndim-1)
    outL[:] = np.tensordot(weight, inL, axes=(inDims, inDims))+bias

def fcTranspose(dJdinL, inL, weight, dJdoutL, dJdWeight, dJdbias):
    outDims = dJdoutL.ndim-1
    outAxes = np.arange(outDims)
    WOutAxes = outAxes-outDims
    dJdinL[:] = np.tensordot(weight, dJdoutL, axes=(WOutAxes, outAxes))
    dJdWeight[:] = np.tensordot(inL, dJdoutL, axes=(-1, -1))
    dJdbias[:] = np.expand_dims(np.sum(dJdoutL, axis=-1), -1)

def softmax(inL, outL):
    np.exp(inL-np.amax(inL, axis=0), out=outL)
    outL /= np.sum(outL, axis=0)

def softmaxPrime(dJdinL, outL, dJdoutL):
    a = dJdoutL*outL
    dJdinL[:] = a-(outL*np.sum(a, axis=0))
```

```
def findRMS(param, paramMS, m, e):
    paramMS[:] = m*paramMS + (1-m)*param*param
    return np.sqrt(paramMS+e)

def adadelta(dJda, dJdaMS, aStep, aStepMS, m, e):
    dJdaRMS = findRMS(dJda, dJdaMS, m, e)
    aStepRMS = findRMS(aStep, aStepMS, m, e)
    aStep[:] = -aStepRMS*dJda/dJdaRMS
```