Python implementation of ProtoNN

Citation

The original repository and code: https://github.com/Microsoft/EdgeML/wiki/ProtoNN
The original paper: https://proceedings.mlr.press/v70/gupta17a.html

Utils library

Used just for the hard-threshold function

Hard threshold function

```
import numpy as np
import torch

def hardThreshold(A: torch.Tensor, s):
      '''
      Hard thresholds and modifies in-palce nn.Parameter A with sparsity s
      '''
      #PyTorch disallows numpy access/copy to tensors in graph.
      #.detach() creates a new tensor not attached to the graph.
      A_ = A.data.cpu().detach().numpy().ravel()
      if len(A_) > 0:
          th = np.percentile(np.abs(A_), (1 - s) * 100.0, interpolation='higher')
          A_[np.abs(A_) < th] = 0.0
      A_ = A_.reshape(A.shape)
      return torch.tensor(A_, requires_grad=True)
```

```
A_ = A.data.cpu().detach().numpy().ravel()
```

- `Tensor.data` was an attribute of `Variable` (object representing `Tensor` with history tracking e.g. for automatic update), not `Tensor`. Actually, `.data` was giving access to the `Variable`'s underlying `Tensor`. However, since PyTorch version `0.4.0`, `Variable` and `Tensor` have been merged (into an updated `Tensor` structure), so `.data` disappeared along the previous `Variable` object (well `Variable` is still there for backward-compatibility, but is deprecated).

> What about .data?
>
> `.data` was the primary way to get the underlying `Tensor` from a `Variable`. After this merge, calling `y = x.data` still has similar semantics. So `y` will be a `Tensor` that shares the same data with `x`, is unrelated with the computation history of `x`, and has `requires_grad=False`.
>
> However, `.data` can be unsafe in some cases. Any changes on `x.data` wouldn't be tracked by `autograd`, and the computed gradients would be incorrect if `x` is needed in a backward pass. A safer alternative is to use `x.detach()`, which also returns a `Tensor` that shares data with `requires_grad=False`, but will have its in-place changes reported by `autograd` if `x` is needed in backward.

- `Tensor.cpu()` returns a copy of this object in CPU memory. If this object is already in CPU memory and on the correct device, then no copy is performed and the original object is returned.
- `Tensor.detach()` returns a new Tensor, detached from the current graph.
- `Tensor.numpy()` returns the tensor as a NumPy `ndarray`
- `numpy.ravel()` return a contiguous flattened array. A 1-D array, containing the elements of the input, is returned

`A_` is a copy of the tensor A in the format of a 1-D NumPy `ndarray`.

Percentile

In statistics, a k-th percentile (percentile score or centile) is a score below which a given percentage k of scores in its frequency distribution falls (exclusive definition) or a score at or below which a given percentage falls

```
th = np.percentile(np.abs(A_), (1 - s) * 100.0, interpolation='higher')
```

- `numpy.percentile(a, q)` Compute the q-th percentile of the data `a`. Returns the `q`-th percentile(s) of the array elements. Interpolation (deprecated) is related to the method (optional parameter, if not specified is linear)
- So `numpy.abs` and `numpy.absolute` are completely identical. It doesn't matter which one you use. Calculate the absolute value element-wise.
- `s` is the sparsity

`th` is the threshold value, if an elements of the matrix is below that it will became 0.

```
A_[np.abs(A_) < th] = 0.0
```

Every elements of `A_` which is less then the threshold value `th` became 0.0

```
A_ = A_.reshape(A.shape)
```

`A_` from 1-D dimensional take back the shape of `A`

```
return torch.tensor(A_, requires_grad=True)
```

- `torch.tensor` constructs a tensor with no autograd history (also known as a “leaf tensor”, see Autograd mechanics) by copying `A_`. `requires_grad (bool, optional)` if autograd should record operations on the returned tensor

Return the created new tensor with the shape and value of `A_` making autograd record operation on it

look at the following documents for the implementation

Microsoft implementation

Graph

```
# Copyright (c) Microsoft Corporation. All rights reserved.
# Licensed under the MIT license.

import torch
import torch.nn as nn
import numpy as np


class ProtoNN(nn.Module):
    def __init__(self, inputDimension, projectionDimension, numPrototypes,
                 numOutputLabels, gamma, W=None, B=None, Z=None):
        '''
        Forward computation graph for ProtoNN.

        inputDimension: Input data dimension or feature dimension.
        projectionDimension: hyperparameter
        numPrototypes: hyperparameter
        numOutputLabels: The number of output labels or classes
        W, B, Z: Numpy matrices that can be used to initialize
            projection matrix(W), prototype matrix (B) and prototype labels
            matrix (B).
            Expected Dimensions:
                W   inputDimension (d) x projectionDimension (d_cap)
                B   projectionDimension (d_cap) x numPrototypes (m)
                Z   numOutputLabels (L) x numPrototypes (m)
        '''
        super(ProtoNN, self).__init__()
        self.__d = inputDimension
        self.__d_cap = projectionDimension
        self.__m = numPrototypes
        self.__L = numOutputLabels

        self.W, self.B, self.Z = None, None, None
        self.gamma = gamma
        self.__validInit = False
        self.__initWBZ(W, B, Z)
        self.__validateInit()

    def __validateInit(self):
        self.__validinit = False
        errmsg = "Dimensions mismatch! Should be W[d, d_cap]"
        errmsg+= ", B[d_cap, m] and Z[L, m]"
        d, d_cap, m, L, _ = self.getHyperParams()
        assert self.W.shape[0] == d, errmsg
        assert self.W.shape[1] == d_cap, errmsg
        assert self.B.shape[0] == d_cap, errmsg
        assert self.B.shape[1] == m, errmsg
        assert self.Z.shape[0] == L, errmsg
        assert self.Z.shape[1] == m, errmsg
        self.__validInit = True

    def __initWBZ(self, inW, inB, inZ):
        if inW is None:
            self.W = torch.randn([self.__d, self.__d_cap])
            self.W = nn.Parameter(self.W)
        else:
            self.W = nn.Parameter(torch.from_numpy(inW.astype(np.float32)))

        if inB is None:
            self.B = torch.randn([self.__d_cap, self.__m])
            self.B = nn.Parameter(self.B)
        else:
            self.B = nn.Parameter(torch.from_numpy(inB.astype(np.float32)))

        if inZ is None:
            self.Z = torch.randn([self.__L, self.__m])
            self.Z = nn.Parameter(self.Z)
        else:
            self.Z = nn.Parameter(torch.from_numpy(inZ.astype(np.float32)))

    def getHyperParams(self):
        '''
        Returns the model hyperparameters:
            [inputDimension, projectionDimension, numPrototypes,
            numOutputLabels, gamma]
        '''
        d =  self.__d
        dcap = self.__d_cap
        m = self.__m
        L = self.__L
        return d, dcap, m, L, self.gamma

    def getModelMatrices(self):
        '''
        Returns model matrices, which can then be evaluated to obtain
        corresponding numpy arrays.  These can then be exported as part of
        other implementations of ProtonNN, for instance a C++ implementation or
        pure python implementation.
        Returns
            [ProjectionMatrix (W), prototypeMatrix (B),
             prototypeLabelsMatrix (Z), gamma]
        '''
        return self.W, self.B, self.Z, self.gamma

    def forward(self, X):
        '''
        This method is responsible for construction of the forward computation
        graph. The end point of the computation graph, or in other words the
        output operator for the forward computation is returned.

        X: Input of shape [-1, inputDimension]
        returns: The forward computation outputs, self.protoNNOut
        '''
        assert self.__validInit is True, "Initialization failed!"

        W, B, Z, gamma = self.W, self.B, self.Z, self.gamma
        WX = torch.matmul(X, W)
        dim = [-1, WX.shape[1], 1]
        WX = torch.reshape(WX, dim)
        dim = [1, B.shape[0], -1]
        B_ = torch.reshape(B, dim)
        l2sim = B_ - WX
        l2sim = torch.pow(l2sim, 2)
        l2sim = torch.sum(l2sim, dim=1, keepdim=True)
        self.l2sim = l2sim
        gammal2sim = (-1 * gamma * gamma) * l2sim
        M = torch.exp(gammal2sim)
        dim = [1] + list(Z.shape)
        Z_ = torch.reshape(Z, dim)
        y = Z_ * M
        y = torch.sum(y, dim=2)
        return y
```

__init__ function

```
class ProtoNN(nn.Module):
    def __init__(self, inputDimension, projectionDimension, numPrototypes,
                 numOutputLabels, gamma, W=None, B=None, Z=None):
        '''
        Forward computation graph for ProtoNN.

        inputDimension: Input data dimension or feature dimension.
        projectionDimension: hyperparameter
        numPrototypes: hyperparameter
        numOutputLabels: The number of output labels or classes
        W, B, Z: Numpy matrices that can be used to initialize
```

```
            projection matrix(W), prototype matrix (B) and prototype labels
            matrix (Z).
            Expected Dimensions:
                W   inputDimension (d) x projectionDimension (d_cap)
                B   projectionDimension (d_cap) x numPrototypes (m)
                Z   numOutputLabels (L) x numPrototypes (m)
        '''
        super(ProtoNN, self).__init__()
        self.__d = inputDimension
        self.__d_cap = projectionDimension
        self.__m = numPrototypes
        self.__L = numOutputLabels

        self.W, self.B, self.Z = None, None, None
        self.gamma = gamma

        self.__validInit = False
        self.__initWBZ(W, B, Z)
        self.__validateInit()
```

The `__init__` method is the Python equivalent of the C++ constructor in an object-oriented approach. The `__init__` function is called every time an object is created from a class. The `__init__` method lets the class initialize the object's attributes and serves no other purpose. It is only used within classes.

```
__init__(self, inputDimension, projectionDimension, numPrototypes,
                 numOutputLabels, gamma, W=None, B=None, Z=None)
```

It takes as input:

- inputDimension: input data dimension or feature dimension. The original number of features of the data, we can call it `d`.
- projectionDimension: is an **hyperparameter** and is the number of transformed features after the dimensionality reduction: `d_cap`.
- numPrototypes: is an **hyperparameter** and is the number of prototypes we want to store to represent our dataset: `m`.
- numOutputLabels: the number of output labels or class: `L`.
- gamma: **hyperparameter** used in the gaussian Kernel function.

- W, B, Z: `numpy.ndarray` that can be used to initialize projection matrix `W`, prototype matrix `B` and prototype labels matrix `Z`. Expected dimension:
    - `W` : inputDimension `d` x projectionDimension `d_cap`
    - `B` : projectionDimension `d_cap` x numPrototypes `m`
    - `Z` : numOutputLabels `L` x numPrototypes `m`

```
super(ProtoNN, self).__init__()
```

The `super` method returns a proxy object that delegates method calls to a parent or sibling class of **type**. This is useful for accessing inherited methods that have been overridden in a class. Or simply, it is used to call the constructor, i.e. the `__init__()` method of the **superclass**, in that case the `torch.nn.Module` class.

```
self.__d = inputDimension
self.__d_cap = projectionDimension
self.__m = numPrototypes
self.__L = numOutputLabels

self.W, self.B, self.Z = None, None, None
self.gamma = gamma
```

Assign to the “private” variable of the class the corresponding value given as input. Assing to the public “variable” `gamma`, and empty object for `W`, `B` and `Z`.

Single and Double Underscore

```
self.__validInit = False
self.__initWBZ(W, B, Z)
self.__validateInit()
```

Define a “private” variable `__validInit` to store True if the input are validate, False otherwise. Call a private function `__initWBZ` to initiate and assign correctly `W`, `B` and `Z`. Call the private function `__validateInit()` to assert if all the input are valid (it store the answer in `__validInit`).

__initWBZ() function

```
def __initWBZ(self, inW, inB, inZ):
        if inW is None:
            self.W = torch.randn([self.__d, self.__d_cap])
            self.W = nn.Parameter(self.W)
        else:
            self.W = nn.Parameter(torch.from_numpy(inW.astype(np.float32)))

        if inB is None:
            self.B = torch.randn([self.__d_cap, self.__m])
            self.B = nn.Parameter(self.B)
        else:
            self.B = nn.Parameter(torch.from_numpy(inB.astype(np.float32)))

        if inZ is None:
            self.Z = torch.randn([self.__L, self.__m])
            self.Z = nn.Parameter(self.Z)
        else:
            self.Z = nn.Parameter(torch.from_numpy(inZ.astype(np.float32)))
```

- in `inW`, `inB` and `inZ` are the three input matrix in the form of `numpy.array`.
- `numpy.ndarray.astype(dtype)` copy of the array, cast to a specified type `dtype`.
- `numpy.float32` alias of `single`. Single-precision floating-point number type, compatible with C `float`. 32-bit-precision floating-point number type: sign bit, 8 bits exponent, 23 bits mantissa.
- `torch.randn(*size)` returns a tensor filled with random numbers from a normal distribution with mean 0 and variance 1 (also called the standard normal distribution). The shape of the tensor is defined by the variable argument `size`.
    - **size** (*int...*) – a sequence of integers defining the shape of the output tensor. Can be a variable number of arguments or a collection like a list or tuple.

- `torch.nn.Parameter(data)` A kind of Tensor that is to be considered a module parameter. Parameters are `Tensor` subclasses, that have a very special property when used with `Module` s - when they're assigned as Module attributes they are automatically added to the list of its parameters, and will appear e.g. in `parameters()` iterator. Assigning a Tensor doesn't have such effect. This is because one might want to cache some temporary state, like last hidden state of the RNN, in the model. If there was no such class as `Parameter`, these temporaries would get registered too.
- `torch.from_numpy(ndarray)` Creates a `torch.Tensor` from a `numpy.ndarray`. The returned tensor and `ndarray` share the same memory. Modifications to the tensor will be reflected in the `ndarray` and vice versa. The returned tensor is not resizable.

For each parameter `W`, `B` and `Z`, assingn a matrix with random values if is not provided as input. Otherwise it transform the input in float, and create a `torch.Tensor` from each `numpy.ndarray`. Each of the so created Tensor is defined as `Parameter`of the `Module`, which means it will eventually optimized through the gradient.

__validateInit() function

```
def __validateInit(self):
    self.__validinit = False
    errmsg = "Dimensions mismatch! Should be W[d, d_cap]"
    errmsg+= ", B[d_cap, m] and Z[L, m]"
    d, d_cap, m, L, _ = self.getHyperParams()
    assert self.W.shape[0] == d, errmsg
    assert self.W.shape[1] == d_cap, errmsg
    assert self.B.shape[0] == d_cap, errmsg
    assert self.B.shape[1] == m, errmsg
    assert self.Z.shape[0] == L, errmsg
    assert self.Z.shape[1] == m, errmsg
    self.__validInit = True
```

- The `assert` keyword lets you test if a condition in your code returns True, if not, the program will raise an AssertionError. You can write a message to be written if the code returns False:
    - `assert False, "The assertion failed!"`

- `torch.Tensor.shape` is a property of the Tensor class which returns a `torch.Size()` objects containing a list of elements which represent the dimension, you can access to the elements of the list through the brackets.

The function will control the shape of the tensors `W`, `B` and `Z`, asserting they all have the right dimension. If that's the case it will sent the "private" variable `__validInit` to `True`. Otherwise will throw an exception and print the `errmsg`.

getHyperParams() and getModelMatrices()

Simply "public" functions of the class which return the corresponding variable saved in the instance of the class.

forward() - Prediction Function of the Model

```
def forward(self, X):
    '''
    This method is responsible for construction of the forward computation
    graph. The end point of the computation graph, or in other words the
    output operator for the forward computation is returned.

    X: Input of shape [-1, inputDimension]
    returns: The forward computation outputs, self.protoNNOut
    '''
    assert self.__validInit is True, "Initialization failed!"

    W, B, Z, gamma = self.W, self.B, self.Z, self.gamma
    WX = torch.matmul(X, W)
    dim = [-1, WX.shape[1], 1]
    WX = torch.reshape(WX, dim)
    dim = [1, B.shape[0], -1]
    B_ = torch.reshape(B, dim)
    l2sim = B_ - WX
    l2sim = torch.pow(l2sim, 2)
    l2sim = torch.sum(l2sim, dim=1, keepdim=True)
    self.l2sim = l2sim
    gammal2sim = (-1 * gamma * gamma) * l2sim
    M = torch.exp(gammal2sim)
    dim = [1] + list(Z.shape)
    Z_ = torch.reshape(Z, dim)
    y = Z_ * M
    y = torch.sum(y, dim=2)
    return y
```

- `torch.matmul(input, other)` returns a `torch.Tensor` which is the matrix product of two tensors. The behavior depends on the dimensionality of the tensors as follows:
    - If both tensors are 1-dimensional, the dot product (scalar) is returned.
    - If both arguments are 2-dimensional, the matrix-matrix product is returned.
    - If the first argument is 1-dimensional and the second argument is 2-dimensional, a 1 is prepended to its dimension for the purpose of the matrix multiply. After the matrix multiply, the prepended dimension is removed.
    - If the first argument is 2-dimensional and the second argument is 1-dimensional, the matrix-vector product is returned.
    - If both arguments are at least 1-dimensional and at least one argument is N-dimensional (where N > 2), then a batched matrix multiply is returned. If the first argument is 1-dimensional, a 1 is prepended to its dimension for the purpose of the batched matrix multiply and removed after. If the second argument is 1-dimensional, a 1 is appended to its dimension for the purpose of the batched matrix multiple and removed after. The non-matrix (i.e. batch) dimensions are broadcasted (and thus must be broadcastable). For example, if `input` is a (j×1×n×n)(j×1×n×n) tensor and `other` is a (k×n×n)(k×n×n) tensor, `out` will be a (j×k×n×n)(j×k×n×n) tensor.
        - Note that the broadcasting logic only looks at the batch dimensions when determining if the inputs are broadcastable, and not the matrix dimensions. For example, if `input` is a (j×1×n×m)(j×1×n×m) tensor and `other` is a (k×m×p)(k×m×p) tensor, these inputs are valid for broadcasting even though the final two dimensions (i.e. the matrix dimensions) are different. `out` will be a (j×k×n×p)(j×k×n×p) tensor.

```
# batched matrix x broadcasted matrix
>>> tensor1 = torch.randn(10, 3, 4)
>>> tensor2 = torch.randn(4, 5)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([10, 3, 5])
```

- `torch.rehsape(input, shape)` returns a `torch.Tensor` with the same data and number of elements as `input`, but with the specified shape. When possible, the returned tensor will be a view of `input`. Otherwise, it will be a copy. Contiguous inputs and inputs with compatible strides can be reshaped without copying, but you should not depend on the copying vs. viewing behavior. **A single dimension may be -1**, in which case it's inferred from the remaining dimensions and the number of elements in `input`.

- `torch.pow(input, exponent)` takes the power of each element in `input` with `exponent` and returns a `torch.Tensor` with the result. `exponent` can be either a single `float` number or a Tensor with the same number of elements as `input`.
    - When `exponent` is a scalar value, the operation applied is: $out_i = x_i^{exponent}$
    - When `exponent` is a tensor, the operation applied is: $out_i = x_i^{exponent_i}$. In that case, the shapes of `input` and `exponent` must be broadcastable.
- `torch.sum(input, dim, keepdim=False)` returns the sum of each row of the `input` tensor in the given dimension `dim`. If `dim` is a list of dimensions, reduce over all of them. If `keepdim` is `True`, the output tensor is of the same size as `input` except in the dimension(s) `dim` where it is of size 1. Otherwise, `dim` is squeezed, resulting in the output tensor having 1 (or `len(dim)`) fewer dimension(s).
    - `torch.squeeze()` returns a tensor with all the dimensions of `input` of size 1 removed. For example, if input is of shape: (A×1×B×C×1×D) then the out tensor will be of shape: (A×B×C×D). When `dim` is given, a squeeze operation is done only in the given dimension.

```
[ [1, 2, 1], [3, 2, 3] ]
sum dim=0, keepdim=True → [ [ 4, 4, 4 ] ]
sum dim=0, keepdim=False → [ 4, 4, 4 ]
sum dim=1, keepdim=True → [ [ 4 ], [ 8 ] ]
sum dim=1, keepdim=False → [ 4, 8 ]
```

- `torch.Tensor - torch.Tensor` is a (batched) subtraction between `torch.Tensor`. More later.
- `torch.exp(input)` returns a new `torch.Tensor` with the exponential of the elements of the input tensor `input`. $y_i = e^{x_i}$

First check if the internal parameters are valid through the `__validInit` check. Then assign the values to the variable to use them in the function.

```
# X: Input of shape [-1, inputDimension]
```

With shape `-1` it means is inferred, so the shape of `X` can be [number of data points, inputDimension]. X is the matrix of datapoints we want to test it (without the labels).

```
WX = torch.matmul(X, W)
```

`X` shape is `[numDatapoints, inputDimension]` and `W` shape is `[inputDimension, projectionDimension]`, the result will be the matrix of the data points `WX` in the transformed space `[numDatapoints, projectionDimension]`.

```
dim = [-1, WX.shape[1], 1]
WX = torch.reshape(WX, dim)
```

The list `dim` is used to reshape the tensor `WX` and is equal to `[-1, projectionDimension, 1]`, after the function `torch.reshape` the shape of `WX` is `[numDatapoints, projectionDimension, 1]`.

```
dim = [1, B.shape[0], -1]
B_ = torch.reshape(B, dim)
```

We change `dim` into `[1, projectionDimension, -1]` to reshape `B_`, as done in the previous step, into `[1, projectionDimension, numPrototypes]`.

```
l2sim = B_ - WX
```

The l2sim (that will stand for loss squared similarity) is the tensor resulting by the subtraction of `B_` and `WX` : `[1, projectionDimension, numPrototypes]` - `[numDatapoints, projectionDimension, 1]`.

The result's shape is `[numDatapoints, projectionDimension, numPrototypes]`. That means for each data points under testing, is computed the distance for each dimension, between the data points and each prototypes.

$sub_{i,j,k} = B_{1,j,k} - WX_{i,j,1}$

```
l2sim = torch.pow(l2sim, 2)
l2sim = torch.sum(l2sim, dim=1, keepdim=True)
```

The matrix of the distance between all the test points and the prototypes are squared. Then the distance among the different dimension are summed, the new shape of `l2sim` is `[numDatapoints, 1, numPrototypes]`. It represent the Euclidean Squared Distance between each test points and each prototypes.

```
gammal2sim = (-1 * gamma * gamma) * l2sim
```

To the previous `torch.Tensor` we multiply $-\gamma^2$ to each distance.

```
M = torch.exp(gammal2sim)
```

`M` is the `torch.Tensor` representing the similarity (through the Kernel function). Has the shape `[numDatapoints, 1, numPrototypes]`.

```
dim = [1] + list(Z.shape)
Z_ = torch.reshape(Z, dim)
```

We change `dim` into `[1, numOutputLabels, numPrototypes]` to reshape `Z`, obtaining `Z_` with shape `[1, numOutputLabels, numPrototypes]`.

```
y = Z_ * M
```

`Z_` with shape `[1, numOutputLabels, numPrototypes]` is multiplied to `M` with shape `[numDatapoints, 1, numPrototypes]` obtaining `y` with shape `[numDatapoints, numOutputLabels,` `numPrototypes]`. It represent for each datapoints the contribution to the score of each prototypes for each class. Is given by $y_{i,j,k} = Z_{1,j,k} * M_{i,1,k}$

`Z` represent the score of prototypes and `M` the similarity (in function of the distance)

```
y = torch.sum(y, dim=2)
return y
```

We apply the sum on the third dimension, this way we obtain the total score obtained by each datapoints for the given classes. The `torch.Tensor` is returned and we can find the highest score to choose the class to assign to the data point. It will have the following shape: `[numDatapoints, numOutputLabels]`.

The `forward function` is used to compute the graph of the model used to establish the result and by autograd to compute the gradient. We can draw the computational graph to summarise the operation.

```
graph TD
  X--> torch.matmul
  W--> torch.matmul
  torch.matmul--"saved as"-->WX
  WX--reshaped--> WX_
  B--reshaped-->B_
  B_-->B_-WX
  WX_-->B_-WX
  B_-WX--"saved as"-->l2sim
  l2sim-->torch.pow
  torch.pow--"saved as"-->l2sim_
  l2sim_-->torch.sum
  torch.sum--"saved as"-->l2sim__
  gamma-->-1*gamma*gamma*l2sim__
  l2sim__-->-1*gamma*gamma*l2sim__
  -1*gamma*gamma*l2sim__--"saved as"-->gammal2sim
  gammal2sim-->torch.exp
  torch.exp--"saved as"-->M
  Z--reshape-->Z_
  Z_-->Z_*M
  M-->Z_*M
  Z_*M--"saved as"-->y
  y-->torch.sum_
  torch.sum_--"saved as"-->y_
```

Trainer

```
# Copyright (c) Microsoft Corporation. All rights reserved.abs
# Licensed under the MIT license.

import torch
import numpy as np
import os
import sys
import edgeml_pytorch.utils as utils


class ProtoNNTrainer:

    def __init__(self, protoNNObj, regW, regB, regZ, sparcityW, sparcityB,
                 sparcityZ, learningRate, lossType='l2', device=None):
        '''
        A wrapper for the various techniques used for training ProtoNN. This
        subsumes both the responsibility of loss graph construction and
        performing training. The original training routine that is part of the
        C++ implementation of EdgeML used iterative hard thresholding (IHT),
        gamma estimation through median heuristic and other tricks for
        training ProtoNN. This module implements the same in pytorch
        and python.

        protoNNObj: An instance of ProtoNN class defining the forward
            computation graph. The loss functions and training routines will be
            attached to this instance.
        regW, regB, regZ: Regularization constants for W, B, and
            Z matrices of protoNN.
        sparcityW, sparcityB, sparcityZ: Sparsity constraints
            for W, B and Z matrices. A value between 0 (exclusive) and 1
            (inclusive) is expected. A value of 1 indicates dense training.
        learningRate: Initial learning rate for ADAM optimizer.
        X, Y : Placeholders for data and labels.
            X [-1, featureDimension]
            Y [-1, num Labels]
        lossType: ['l2', 'xentropy']

        '''
        self.protoNNObj = protoNNObj
        self.__regW = regW
        self.__regB = regB
        self.__regZ = regZ
        self.__sW = sparcityW
        self.__sB = sparcityB
        self.__sZ = sparcityZ
        self.__lR = learningRate
        self.sparseTraining = True
        if (sparcityW == 1.0) and (sparcityB == 1.0) and (sparcityZ == 1.0):
            self.sparseTraining = False
            print("Sparse training disabled.", file=sys.stderr)
        self.W_th = None
        self.B_th = None
        self.Z_th = None
        self.__lossType = lossType
        self.optimizer = self.__optimizer()
        self.lossCriterion = None
        assert lossType in ['l2', 'xentropy']
        if lossType == 'l2':
            self.lossCriterion = torch.nn.MSELoss()
            print("Using L2 (MSE) loss")
        else :
            self.lossCriterion = torch.nn.CrossEntropyLoss()
            print("Using x-entropy loss")
        self.__validInit = False
        self.__validInit = self.__validateInit()
        if device is None:
            self.device = "cpu"
        else:
            self.device = device

    def __validateInit(self):
        assert self.__validInit == False
        msg = "Sparsity values should be between 0 and 1 (both inclusive)"
        assert 0 <= self.__sW <= 1, msg
        assert 0 <= self.__sB <= 1, msg
        assert 0 <= self.__sZ <= 1, msg
        return True

    def __optimizer(self):
        optimizer = torch.optim.Adam(self.protoNNObj.parameters(),
                                     lr=self.__lR)
        return optimizer

    def loss(self, logits, labels_or_target):
        labels = labels_or_target
        assert len(logits) == len(labels)
        assert len(labels.shape) == 2
        assert len(logits.shape) == 2
        regLoss = (self.__regW * (torch.norm(self.protoNNObj.W)**2) +
                   self.__regB * (torch.norm(self.protoNNObj.B)**2) +
                   self.__regZ * (torch.norm(self.protoNNObj.Z)**2))
        if self.__lossType == 'xentropy':
            _, labels = torch.max(labels, dim=1)
            assert len(labels.shape)== 1
        loss = self.lossCriterion(logits, labels) + regLoss
        return loss

    def accuracy(self, predictions, labels):
        '''
        Returns accuracy and number of correct predictions.
        '''
        assert len(predictions.shape) == 1
        assert len(labels.shape) == 1
        assert len(predictions) == len(labels)
        correct = (predictions == labels).double()
        numCorrect = torch.sum(correct)
        acc = torch.mean(correct)
        return acc, numCorrect

    def hardThreshold(self):
        prtn = self.protoNNObj
        W, B, Z = prtn.W.data, prtn.B.data, prtn.Z.data
        newW = utils.hardThreshold(W, self.__sW)
        newB = utils.hardThreshold(B, self.__sB)
        newZ = utils.hardThreshold(Z, self.__sZ)
        prtn.W.data = torch.FloatTensor(newW).to(self.device)
        prtn.B.data = torch.FloatTensor(newB).to(self.device)
        prtn.Z.data = torch.FloatTensor(newZ).to(self.device)

    def train(self, batchSize, epochs, x_train, x_val, y_train, y_val,
              printStep=10, valStep=1):
        '''
        Performs dense training of ProtoNN followed by iterative hard
        thresholding to enforce sparsity constraints.

        batchSize: Batch size per update
        epochs : The number of epochs to run training for. One epoch is
            defined as one pass over the entire training data.
        x_train, x_val, y_train, y_val: The numpy array containing train and
            validation data. x data is assumed to in of shape [-1,
            featureDimension] while y should have shape [-1, numberLabels].
        printStep: Number of batches between echoing of loss and train accuracy.
        valStep: Number of epochs between evaluations on validation set.
        '''
        d, dcap, m, L, _ = self.protoNNObj.getHyperParams()
        assert batchSize >= 1, 'Batch size should be positive integer'
        assert epochs >= 1, 'Total epochs should be positive integer'
        assert x_train.ndim == 2, 'Expected training data to be of rank 2'
        assert x_train.shape[1] == d, 'Expected x_train to be [-1, %d]' % d
        assert x_val.ndim == 2, 'Expected validation data to be of rank 2'
        assert x_val.shape[1] == d, 'Expected x_val to be [-1, %d]' % d
        assert y_train.ndim == 2, 'Expected training labels to be of rank 2'
        assert y_train.shape[1] == L, 'Expected y_train to be [-1, %d]' % L
        assert y_val.ndim == 2, 'Expected validation labels to be of rank 2'
        assert y_val.shape[1] == L, 'Expected y_val to be [-1, %d]' % L

        trainNumBatches = int(np.ceil(len(x_train) / batchSize))
        valNumBatches = int(np.ceil(len(x_val) / batchSize))
        x_train_batches = np.array_split(x_train, trainNumBatches)
        y_train_batches = np.array_split(y_train, trainNumBatches)
        x_val_batches = np.array_split(x_val, valNumBatches)
        y_val_batches = np.array_split(y_val, valNumBatches)

        for epoch in range(epochs):
            for i in range(len(x_train_batches)):
                x_batch, y_batch = x_train_batches[i], y_train_batches[i]
                x_batch, y_batch = torch.Tensor(x_batch), torch.Tensor(y_batch)
                x_batch, y_batch = x_batch.to(self.device), y_batch.to(self.device)
                self.optimizer.zero_grad()
                logits = self.protoNNObj.forward(x_batch)
                loss = self.loss(logits, y_batch)
                loss.backward()
                self.optimizer.step()
                _, predictions = torch.max(logits, dim=1)
                _, target = torch.max(y_batch, dim=1)
                acc, _ = self.accuracy(predictions, target)
                if i % printStep == 0:
                    print("Epoch %d batch %d loss %f acc %f" % (epoch, i, loss,
                                                                acc))
            # Perform IHT Here.
            if self.sparseTraining:
                self.hardThreshold()
            # Perform validation set evaluation
            if (epoch + 1) % valStep == 0:
                numCorrect = 0
                for i in range(len(x_val_batches)):
                    x_batch, y_batch = x_val_batches[i], y_val_batches[i]
                    x_batch, y_batch = torch.Tensor(x_batch), torch.Tensor(y_batch)
                    x_batch, y_batch = x_batch.to(self.device), y_batch.to(self.device)
                    logits = self.protoNNObj.forward(x_batch)
                    _, predictions = torch.max(logits, dim=1)
                    _, target = torch.max(y_batch, dim=1)
                    _, count = self.accuracy(predictions, target)
                    numCorrect += count
                print("Validation accuracy: %f" % (numCorrect / len(x_val)))
```

__init__ function

```
class ProtoNNTrainer:

    def __init__(self, protoNNObj, regW, regB, regZ, sparcityW, sparcityB,
                 sparcityZ, learningRate, lossType='l2', device=None):
        '''
        A wrapper for the various techniques used for training ProtoNN. This
        subsumes both the responsibility of loss graph construction and
        performing training. The original training routine that is part of the
        C++ implementation of EdgeML used iterative hard thresholding (IHT),
        gamma estimation through median heuristic and other tricks for
        training ProtoNN. This module implements the same in pytorch
        and python.

        protoNNObj: An instance of ProtoNN class defining the forward
            computation graph. The loss functions and training routines will be
            attached to this instance.
        regW, regB, regZ: Regularization constants for W, B, and
```

```
            Z matrices of protoNN.
        sparcityW, sparcityB, sparcityZ: Sparsity constraints
            for W, B and Z matrices. A value between 0 (exclusive) and 1
            (inclusive) is expected. A value of 1 indicates dense training.
        learningRate: Initial learning rate for ADAM optimizer.
        X, Y : Placeholders for data and labels.
            X [-1, featureDimension]
            Y [-1, num Labels]
        lossType: ['l2', 'xentropy']

        '''
        self.protoNNObj = protoNNObj
        self.__regW = regW
        self.__regB = regB
        self.__regZ = regZ
        self.__sW = sparcityW
        self.__sB = sparcityB
        self.__sZ = sparcityZ
        self.__lR = learningRate
        self.sparseTraining = True
        if (sparcityW == 1.0) and (sparcityB == 1.0) and (sparcityZ == 1.0):
            self.sparseTraining = False
            print("Sparse training disabled.", file=sys.stderr)
        self.W_th = None
        self.B_th = None
        self.Z_th = None
        self.__lossType = lossType
        self.optimizer = self.__optimizer()
        self.lossCriterion = None
        assert lossType in ['l2', 'xentropy']
        if lossType == 'l2':
            self.lossCriterion = torch.nn.MSELoss()
            print("Using L2 (MSE) loss")
        else :
            self.lossCriterion = torch.nn.CrossEntropyLoss()
            print("Using x-entropy loss")
        self.__validInit = False
        self.__validInit = self.__validateInit()
        if device is None:
            self.device = "cpu"
        else:
            self.device = device
```

The `ProtoNNTrainer` will extend the `ProtoNN` class (containing as a variable), is a wrapper for the various techniques used for training ProtoNN. This subsumes both the responsibility of loss graph construction and performing training.

```
def __init__(self, protoNNObj, regW, regB, regZ, sparcityW, sparcityB,
                 sparcityZ, learningRate, lossType='l2', device=None)
```

Input:

- protoNNObj: An instance of ProtoNN class defining the forward computation graph. The loss functions and training routines will be attached to this instance. It has to be initialized already.
- regW, regB, regZ: regularization constants for `W`, `B`, and `Z` matrices of ProtoNN.
- sparsityW, sparsityB, sparsityZ: sparsity constraints for `W`, `B`, and `Z` matrices. A value between 0 (exclusive) and 1 (inclusive) is expected. A value of 1 indicates dense training. This is how is enforced the space constraints.
- learningRate: intial learning rate for the optimizer.
- lossType: can be loss squared `l2` or cross entropy `xentropy`.
- device: our case will be cpu.

```
self.protoNNObj = protoNNObj
self.__regW = regW
self.__regB = regB
self.__regZ = regZ
self.__sW = sparcityW
self.__sB = sparcityB
self.__sZ = sparcityZ
self.__lR = learningRate
self.sparseTraining = True
if (sparcityW == 1.0) and (sparcityB == 1.0) and (sparcityZ == 1.0):
    self.sparseTraining = False
    print("Sparse training disabled.", file=sys.stderr)
self.W_th = None
self.B_th = None
self.Z_th = None
self.__lossType = lossType
```

It will assign the input variables to the “private scope”. Check the input sparsity constraints and if is `1.0` on all field set the variable `sparseTraining` to `False` to explicitly say is not requested.

Not clear the usage of `self.W_th`, `self.B_th` and `self.Z_th`.

```
self.optimizer = self.__optimizer()
```

Call the “private” function to assign the method for the optimization of the parameters.

```
self.lossCriterion = None
assert lossType in ['l2', 'xentropy']
if lossType == 'l2':
    self.lossCriterion = torch.nn.MSELoss()
    print("Using L2 (MSE) loss")
else :
    self.lossCriterion = torch.nn.CrossEntropyLoss()
    print("Using x-entropy loss")
```

Define the loss type as either `l2` or `xentropy` and the `lossCriterion`.

- `torch.nn.MSELoss()`: Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.
    - `loss = torch.nn.MSELoss()`, then can be called `loss(x, y)` with the predicted and true class of the test points to compute the loss Tensor.
- **Criterion**: objective function aka criterion - is a function to be minimized or maximized. In this case is an error function so it has to be minimized.

```
self.__validInit = False
self.__validInit = self.__validateInit()
```

Call `__validateInit()` to check if the sparsity input are valid.

```
if device is None:
    self.device = "cpu"
else:
    self.device = device
```

Assign the input variable `device` to the class, or set it as `cpu` if not specified.

__valideateInit() function

```
def __validateInit(self):
    assert self.__validInit == False
    msg = "Sparsity values should be between 0 and 1 (both inclusive)"
    assert 0 <= self.__sW <= 1, msg
    assert 0 <= self.__sB <= 1, msg
    assert 0 <= self.__sZ <= 1, msg
    return True
```

Check if the sparsity values are valid, otherwise print an error message.

__optimizer() function

```
def __optimizer(self):
    optimizer = torch.optim.Adam(self.protoNNObj.parameters(),
                                 lr=self.__lR)
    return optimizer
```

- `torch.optim` is a package implementing various optimization algorithms. To use `torch.optim` you have to construct an optimizer object, that will hold the current state and will update the parameters based on the computed gradients. To construct an `optimizer` you have to give it an iterable containing the **parameters** (all should be `variables`) to optimize. Then, you can specify optimizer-specific options such as the learning rate, weight decay, etc.
    - `optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)`
    - `optimizer = optim.Adam([var1, var2], lr=0.0001)`

Define the optimizer of the class as `torch.optim.Adam` passing the parameter of the instance of the `protoNN` saved and the learning rate.

loss() function

```
def loss(self, logits, labels_or_target):
    labels = labels_or_target
    assert len(logits) == len(labels)
    assert len(labels.shape) == 2
    assert len(logits.shape) == 2
    regLoss = (self.__regW * (torch.norm(self.protoNNObj.W)**2) +
               self.__regB * (torch.norm(self.protoNNObj.B)**2) +
               self.__regZ * (torch.norm(self.protoNNObj.Z)**2))
    if self.__lossType == 'xentropy':
        _, labels = torch.max(labels, dim=1)
        assert len(labels.shape)== 1
    loss = self.lossCriterion(logits, labels) + regLoss
    return loss
```

It has two parameters:

- `logits`: the raw predictions which come out of the forward computation.
- `labels_or_target`: the right outputs for the tested points.

Regularization

Regularization refers to techniques that are used to calibrate machine learning models in order to minimize the adjusted loss function and prevent overfitting or underfitting.

The one used here is the **Ridge (L2) Regularization**, where the cost function is the following: $Loss + \lambda \sum ||w||^2$

λ is the penalty for the errors, w is the weights. In that case this is the finally cost function:

$$Loss + reg_W(\sum ||W||)^2 + reg_B(\sum ||B||)^2 + reg_Z(\sum ||Z||)^2$$

```
regLoss = (self.__regW * (torch.norm(self.protoNNObj.W)**2) +
           self.__regB * (torch.norm(self.protoNNObj.B)**2) +
           self.__regZ * (torch.norm(self.protoNNObj.Z)**2))
```

- `torch.norm()`: returns the matrix norm or vector norm of a given tensor. Is deprecated and may be removed in a future PyTorch release

The regularization term `regLoss` is obtained by the sum of the squared norm of the parameters.

```
loss = self.lossCriterion(logits, labels) + regLoss
return loss
```

After controlling the shape of the input it returns the loss function with the regularizations term. The returned object is a `torch.Tensor` with one value, the scalar resulted after the computation of the loss function.

accuracy() function

```
def accuracy(self, predictions, labels):
    '''
    Returns accuracy and number of correct predictions.
    '''
    assert len(predictions.shape) == 1
    assert len(labels.shape) == 1
    assert len(predictions) == len(labels)
    correct = (predictions == labels).double()
    numCorrect = torch.sum(correct)
    acc = torch.mean(correct)
    return acc, numCorrect
```

It takes two parameters:

- `predictions`: should be a 1-dimensional `torch.Tensor` containing the predicted class for the input values inserted in the `forward` function.
- `labels`: should be a 1-dimensional `torch.Tensor` containing the right class of the input values inserted in the `forward` function.

Functions used:

- `len()`: returns the number of items in an object.
- `Tensor.double()`: returns a Tensor with the specified `torch.float64` type.
- `torch.Tensor == torch.Tensor`: used on same shape Tensor returns `torch.Tensor` with the elements are the result of the boolean equation: $new_i = \text{True if } (a_{i,j} == b_{i,j}), \text{False otherwise}$
- `torch.mean(input)`: returns the mean value of all elements in the `input` tensor.

Given the prediction and the right class of the tested points (after the forward function), sum the number of right prediction and returns it with the mean.

hardThreshold() function

```
def hardThreshold(self):
    prtn = self.protoNNObj
    W, B, Z = prtn.W.data, prtn.B.data, prtn.Z.data
    newW = utils.hardThreshold(W, self.__sW)
    newB = utils.hardThreshold(B, self.__sB)
    newZ = utils.hardThreshold(Z, self.__sZ)
    prtn.W.data = torch.FloatTensor(newW).to(self.device)
    prtn.B.data = torch.FloatTensor(newB).to(self.device)
    prtn.Z.data = torch.FloatTensor(newZ).to(self.device)
```

- `utils.hardThreshold`: modify the tensor and returns a copy which satisfy the sparsity constraints by keeping only the highest absoulte values in the tensor.
- `tensor.to()`: performs Tensor dtype and/or device conversion. A `torch.dtype` and `torch.device` are inferred from the arguments of `self.to(*args, **kwargs)`.

Apply the sparsity constraints to the parameters `W`, `B` and `Z`, overwrite them in the `protoNNObj`.

train() function

```
def train(self, batchSize, epochs, x_train, x_val, y_train, y_val,
              printStep=10, valStep=1):
    '''
    Performs dense training of ProtoNN followed by iterative hard
    thresholding to enforce sparsity constraints.

    batchSize: Batch size per update
    epochs : The number of epochs to run training for. One epoch is
        defined as one pass over the entire training data.
    x_train, x_val, y_train, y_val: The numpy array containing train and
        validation data. x data is assumed to in of shape [-1,
        featureDimension] while y should have shape [-1, numberLabels].
    printStep: Number of batches between echoing of loss and train accuracy.
    valStep: Number of epochs between evaluations on validation set.
    '''
    d, dcap, m, L, _ = self.protoNNObj.getHyperParams()
    assert batchSize >= 1, 'Batch size should be positive integer'
    assert epochs >= 1, 'Total epochs should be positive integer'
    assert x_train.ndim == 2, 'Expected training data to be of rank 2'
    assert x_train.shape[1] == d, 'Expected x_train to be [-1, %d]' % d
    assert x_val.ndim == 2, 'Expected validation data to be of rank 2'
    assert x_val.shape[1] == d, 'Expected x_val to be [-1, %d]' % d
    assert y_train.ndim == 2, 'Expected training labels to be of rank 2'
    assert y_train.shape[1] == L, 'Expected y_train to be [-1, %d]' % L
    assert y_val.ndim == 2, 'Expected validation labels to be of rank 2'
    assert y_val.shape[1] == L, 'Expected y_val to be [-1, %d]' % L

    trainNumBatches = int(np.ceil(len(x_train) / batchSize))
    valNumBatches = int(np.ceil(len(x_val) / batchSize))
    x_train_batches = np.array_split(x_train, trainNumBatches)
    y_train_batches = np.array_split(y_train, trainNumBatches)
    x_val_batches = np.array_split(x_val, valNumBatches)
    y_val_batches = np.array_split(y_val, valNumBatches)

    for epoch in range(epochs):
        for i in range(len(x_train_batches)):
            x_batch, y_batch = x_train_batches[i], y_train_batches[i]
            x_batch, y_batch = torch.Tensor(x_batch), torch.Tensor(y_batch)
            x_batch, y_batch = x_batch.to(self.device), y_batch.to(self.device)
            self.optimizer.zero_grad()
            logits = self.protoNNObj.forward(x_batch)
            loss = self.loss(logits, y_batch)
            loss.backward()
            self.optimizer.step()
            _, predictions = torch.max(logits, dim=1)
            _, target = torch.max(y_batch, dim=1)
            acc, _ = self.accuracy(predictions, target)
            if i % printStep == 0:
                print("Epoch %d batch %d loss %f acc %f" % (epoch, i, loss,
                                                            acc))
        # Perform IHT Here.
        if self.sparseTraining:
            self.hardThreshold()
        # Perform validation set evaluation
        if (epoch + 1) % valStep == 0:
            numCorrect = 0
            for i in range(len(x_val_batches)):
                x_batch, y_batch = x_val_batches[i], y_val_batches[i]
                x_batch, y_batch = torch.Tensor(x_batch), torch.Tensor(y_batch)
                x_batch, y_batch = x_batch.to(self.device), y_batch.to(self.device)
                logits = self.protoNNObj.forward(x_batch)
                _, predictions = torch.max(logits, dim=1)
                _, target = torch.max(y_batch, dim=1)
                _, count = self.accuracy(predictions, target)
                numCorrect += count
            print("Validation accuracy: %f" % (numCorrect / len(x_val)))
```

Performs dense training of ProtoNN followed by iterative hard thresholding to enforce sparsity constraints.

Several parameters:

- `batchSize`: batch size per update
- `epochs`: the number of epochs to run training for. One epoch is defined as one pass over the entire training data.

- `x_train, x_val, y_train, y_val`: the numpy array containing train and validation data. `x` data is assumed to in of shape `[-1, featureDimension]` while `y` should have shape `[-1, numberLabels]`.
- `printStep`: number of batches between echoing of loss and train accuracy.
- `valStep`: number of epochs between evaluations on validation set.

```
assert batchSize >= 1, 'Batch size should be positive integer'
assert epochs >= 1, 'Total epochs should be positive integer'
assert x_train.ndim == 2, 'Expected training data to be of rank 2'
assert x_train.shape[1] == d, 'Expected x_train to be [-1, %d]' % d
assert x_val.ndim == 2, 'Expected validation data to be of rank 2'
assert x_val.shape[1] == d, 'Expected x_val to be [-1, %d]' % d
assert y_train.ndim == 2, 'Expected training labels to be of rank 2'
assert y_train.shape[1] == L, 'Expected y_train to be [-1, %d]' % L
assert y_val.ndim == 2, 'Expected validation labels to be of rank 2'
assert y_val.shape[1] == L, 'Expected y_val to be [-1, %d]' % L
```

- `Tensor.ndim`: alias of `Tensor.dim`, returns the number of dimensions of `self` tensor.

Check if the parameters are valid and satisfy the dimentional condition.

```
trainNumBatches = int(np.ceil(len(x_train) / batchSize))
valNumBatches = int(np.ceil(len(x_val) / batchSize))
x_train_batches = np.array_split(x_train, trainNumBatches)
y_train_batches = np.array_split(y_train, trainNumBatches)
x_val_batches = np.array_split(x_val, valNumBatches)
y_val_batches = np.array_split(y_val, valNumBatches)
```

- `numpy.ceil()`: return the ceiling of the input, element-wise. The ceil of the scalar *x* is the smallest integer *i*, such that `i >= x`. It is often denoted as $\lceil x \rceil$.
- `numpy.array_split(array, indices_or_sections)`: split an array into multiple sub-arrays. Please refer to the `split` documentation. The only difference between these functions is that `array_split` allows `indices_or_sections` to be an integer that does *not* equally divide the axis. For an array of length l that should be split into n sections, it returns l % n sub-arrays of size l//n + 1 and the rest of size l/n.

Split the test and validation sets and their related output into `batchSize`'s array and save into corresponding arrays. E.g., `x_train_batches` is an array of array, for each batch we will use one of these contained array.

```
for epoch in range(epochs):
    for i in range(len(x_train_batches)):
        x_batch, y_batch = x_train_batches[i], y_train_batches[i]
        x_batch, y_batch = torch.Tensor(x_batch), torch.Tensor(y_batch)
        x_batch, y_batch = x_batch.to(self.device), y_batch.to(self.device)
        self.optimizer.zero_grad()
        logits = self.protoNNObj.forward(x_batch)
        loss = self.loss(logits, y_batch)
        loss.backward()
        self.optimizer.step()
        _, predictions = torch.max(logits, dim=1)
        _, target = torch.max(y_batch, dim=1)
        acc, _ = self.accuracy(predictions, target)
        if i % printStep == 0:
            print("Epoch %d batch %d loss %f acc %f" % (epoch, i, loss,
                                                        acc))
    # Perform IHT Here.
    if self.sparseTraining:
        self.hardThreshold()
    # Perform validation set evaluation
    if (epoch + 1) % valStep == 0:
        numCorrect = 0
        for i in range(len(x_val_batches)):
            x_batch, y_batch = x_val_batches[i], y_val_batches[i]
            x_batch, y_batch = torch.Tensor(x_batch), torch.Tensor(y_batch)
            x_batch, y_batch = x_batch.to(self.device), y_batch.to(self.device)
            logits = self.protoNNObj.forward(x_batch)
            _, predictions = torch.max(logits, dim=1)
            _, target = torch.max(y_batch, dim=1)
            _, count = self.accuracy(predictions, target)
            numCorrect += count
        print("Validation accuracy: %f" % (numCorrect / len(x_val)))
```

- `range()`: function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

Repeat the **optimization and evaluation procedure** for the requested `epochs` times.

The following part is the one responsible of computing the loss and doing a step in the optimization algorithm:

```
for i in range(len(x_train_batches)):
    x_batch, y_batch = x_train_batches[i], y_train_batches[i]
    x_batch, y_batch = torch.Tensor(x_batch), torch.Tensor(y_batch)
    x_batch, y_batch = x_batch.to(self.device), y_batch.to(self.device)
    self.optimizer.zero_grad()
    logits = self.protoNNObj.forward(x_batch)
    loss = self.loss(logits, y_batch)
    loss.backward()
    self.optimizer.step()
    _, predictions = torch.max(logits, dim=1)
    _, target = torch.max(y_batch, dim=1)
    acc, _ = self.accuracy(predictions, target)
    if i % printStep == 0:
        print("Epoch %d batch %d loss %f acc %f" % (epoch, i, loss, acc))
```

- `optimizer.zero_grad()` : Sets the gradients of all optimized `torch.Tensor` s to zero.
- `protoNN.forward()` : prediction function of the protoNN, it compute the forward graph used by autograd to calculate the derivative and optimize the parameters of the model.
- `Tensor.backward()` : computes the gradient of current tensor w.r.t. graph leaves. The graph is differentiated using the chain rule. If the tensor is non-scalar (i.e. its data has more than one element) and requires gradient, the function additionally requires specifying `gradient` . It should be a tensor of matching type and location, that contains the gradient of the differentiated function w.r.t. `self` . This function accumulates gradients in the leaves - you might need to zero `.grad` attributes or set them to `None` before calling it.

  After doing the backward pass, the graph will be freed to save memory. This means that you have to compute again the graph through the `forward` function before being able to use again the backward function (except if specific optional parameters are used).
- `torch.optim.optimizer.step` : performs a single optimization step (parameter update).
- `torch.max` : returns the maximum value of all elements in the `input` tensor.

For each batch in our train set, compute the prediction and later the loss. Then computes the gradiet w.r.t each parameter, thanks to the autograd graph, and update the parameters. Lastly we take the argmax of the prediction to compute the accuracy and print to the user. The graph necessarily to the derivative is computed through the `forward` function and is kept in memory since is based on `torch.Tensor` with the attribute `requires_grad` set to `True`.

```
# Perform IHT Here.
if self.sparseTraining:
    self.hardThreshold()
```

If the sparsity constraint are requested apply the iterative hard threshold to the parameters.

```
# Perform validation set evaluation
if (epoch + 1) % valStep == 0:
    numCorrect = 0
    for i in range(len(x_val_batches)):
        x_batch, y_batch = x_val_batches[i], y_val_batches[i]
        x_batch, y_batch = torch.Tensor(x_batch), torch.Tensor(y_batch)
        x_batch, y_batch = x_batch.to(self.device), y_batch.to(self.device)
        logits = self.protoNNObj.forward(x_batch)
        _, predictions = torch.max(logits, dim=1)
        _, target = torch.max(y_batch, dim=1)
        _, count = self.accuracy(predictions, target)
        numCorrect += count
    print("Validation accuracy: %f" % (numCorrect / len(x_val)))
```

After the requested `epochs` iterations, it evaluates the test set one batch at time. To evaluate it computes the prediction through the forward function, take the highest value for each test and select the corresponding class. Use the `accuracy` function to get the number of the corrected predictions for that batch and sum it to the total sum of the right predictions so far. Lastly prints the overall accuracy: number of the total corrected prediction over the total of prediction done.