Training on your own Model and your own Data

Example:

In order to use the code for creating your own model on your own dataset, you would have to edit the code under the headings '*Data'* , '*Architecture*' and '*Training*' in ''CNN_Training_Code.ipynb' and run the file. You would have to make an object of each class (each layer has a class) and pass a list of these objects to the constructor of the 'Network' class. The code would look something like this:

Data:
```
path_to_data = '/content/drive/MyDrive/PBC_dataset_normal_DIB'
data = Dataset(path_to_data, resize=(50,50), zero_center='image' )
```

Architecture:
```
Conv4 = Conv(number_of_filters=4, filter_size=11, stride=1,
zero_padding=1)
LReLU4= LeakyReLU()
Pool4= Pool('max', filter_size=2, stride=2)
Conv8= Conv(number_of_filters=8, filter_size=3, stride=1, zero_padding=1)
LReLU8= LeakyReLU()
Pool8= Pool('max', filter_size=3, stride=2)
FC200 = FC(neurons=200)
LReLU200 = LeakyReLU()
FC8 = FC(neurons=8)
SM = Softmax ( )
CE = CELoss (number_of_classes=8)
layers=[Conv4, LReLU4, Pool4, Conv8, LReLU8, Pool8, FC200,LReLU200, FC8,
SM, CE]
```

Training:
```
weights_init='He'
bias_init=0.01
eval_metric = [ClassAccu, ConfMatrix]
optimizer = AD( beta1=0.9, beta2=0.999)
```

```
train_batch_size = 768
valid_batch_size = 1024
epochs = 2
lmbda = 0.001
lr = 0.0001


model=Network(layers=layers, weights_init=weights_init,
bias_init=bias_init, optimizer=optimizer, eval_metric=eval_metric)

train_cost_iter, train_accu_iter, valid_cost_iter, valid_accu_iter =
model.train(data, train_batch_size, valid_batch_size, epochs, lmbda, lr)
```

Or you can simply edit these blocks in the provided code according to your own needs and make your life easier.

Layers:

A detailed information on which layers are supported by the code and what they expect is provided below:

Dataset

```
class Dataset:
 '''
   Returns an object of "Dataset" class that is used as input to 'Network'
class.

   Paramters:

     'path' : str
       path to main folder containing subfolders of images in a specific
structure. For more info, see notes.

     'resize' : tuple of ints (width, height) (optional)
```

```
       shape of image in which it would be resized to. If no value is
passed, no resizing will be done.

     'split_ratio' : tuple of int or float (train_size_percent,
valid_size_percent, test_size_percent) Default = (60,20,20) (optional)
       ratio in which data will be split. The sum of the values must be
equal to 100.

     'zero_center': str = 'per_channel' or 'image'. Default = None
(optional)
       defines zero centering of the data. It can take one of the
folllowing values:

           'per_channel' : Subtract the mean per channel calculated over
all images (like in VGG)
           'image' : Subtract mean image calculated over all images (like
in AlexNet)

       If no value is passed, zero centering of the data will not be done

   Notes:

     - The class expects images to be in certain architecture of folders.
There should be one main folder whose path will be passed
       to the parameter 'path'. Inside this main folder, there should be,
equal to number of classes, folders. Each folder should be named
       with class label and inside it, there should be only images (in
jpeg format, each having same size with shape(W,H,3)) of this class.
       Make sure there are no hidden folders.Everything else except
folders and images will be ignored.

     - The class expects images to be in uint8 format as it will map these
values between 0 and 1 for better processing.

 '''
```

Convolutional Layer:

```
class Conv:
 '''
   Returns an object of "Conv" class that is used as Convolution Layer in
CNNs.

   Paramters:

       'number_of_filters' : non-zero, positive, int
         Number of filters that will be convolved with the activation maps

       'filter_size' : non-zero, positive, odd, int
         size of the square filter

       'stride' : non-zero, positive, int. Default=1 (optional)
         size of step to take in horizontal and vertical direction

       'zero_padding' : positive, int. Default=0 (i.e. no zero padding)
(optional)
         number of rows and columns of zeros that need to be added around
the activation maps.
           For example, if it is specified to be as 2 and activations has
a shape of (batch, channels, rows, cols), then
           after zero padding, the new shape of activations will be
(batch, channels, rows + 4, cols + 4) with zeros above and below, right
and left of the axis -1 and -2.


   Notes:

     -   Combination of stride, zero padding, last two axis of both
filters and activation maps must be
         such that it results an whole number for the following equation:

                For each axis:
                        O = ( ( R - F + 2P ) / S ) + 1
                where
                        O = size of output axis
                        R = size of activation map axis
                        F = size of filter
                        P = amount of zero padding
                        S = stride

 '''
```

Activation Functions:

Hyperbolic Tangent Function:

```
class Tanh:
 '''
   Returns an object of class "Tanh" that can be used as activation
function in CNNs
  '''
```

ReLU:

```
class ReLU:
 '''
   Returns an object of "ReLU" class that can be used as activation
function in CNNs
  '''
```

Leaky ReLU:

```
class LeakyReLU:
 '''
   Returns an object of "LeakyRelU" class that can be used as activation
function in CNNs

   Parameters:
     'slope' : int or float. Default=0.01 (optional)
       slope of the function for negative values of input i.e it
represents 'a' in the following formula:

           f(x) = max(ax, x)
  '''
```

Softmax:

```
class Softmax:
 '''
   Returns an object of "Softmax" class that can be used with cross
entropy loss in CNNs
  '''
```

Pooling Layer:

```
class Pool:
 '''
   Returns an object of "Pool" class that can be used as Pooling Layer in
CNNs.

   Paramters:

     'pooling_type' : str = 'max', 'average' or 'global'
       It defines which type of pooling to be applied on the activation
maps. It can take one of the following values:

         'max': outputs maximum value of region of activation map under
the window
         'average' or 'global': outputs mean value of region of activation
map under the window

     'filter_size' : non-zero, positive, int (optional when using
'pooling_type' = 'global')
```

```
       size of the square window that needs to operate over each
activation map

     'stride' : non-zero, positive, int. Default=2 (optional)
       size of step to take in horizontal and vertical direction

     'zero_padding' : positive, int. Default=0 (i.e. no zero padding)
(optional)
       number of rows and columns of zeros that needs to be added around
each activation map.
         For example, if it is specified to be as 2 and activations has a
shape of (batch, channels, rows, cols), then
         after zero padding, the new shape of activations will be (batch,
channels, rows + 4, cols + 4) with zeros above and below, right and left
of the axis -1 and -2.

 Notes:

   -  Combination of stride, zero padding, last two axis of both filter
and activation maps must be
   such that it results in an whole number for the following equation:

             For each axis:
                     O = ( ( R - F + 2P ) / S ) + 1
             where
                     O = size of output axis
                     R = size of activation map axis
                     F = size of filter
                     P = amount of zero padding
                     S = stride

   -  When using 'global' as 'pooling type', following parameters will be
set as:

             'filter_size' = size of last two axis of activation map
(rows, cols)
             'stride' = 1
             'zero_padding' = 0
```

```
      Thus, these parameters are not required. In case, if they are
provided then they will be overwritten.
  '''
```

Fully Connected Layer:

```
class FC:
 '''
   Returns an object of "FC" class that can be used as fully connected
layer in CNNs

   Parameters:

     'neurons' : non-zero, positive, int
       Number of Neurons in the Fully Connected Layer
 '''
```

Cross Entropy Loss:

```
class CELoss:
 '''
   Returns an object of "CELoss" class that can be used as cross entropy
loss for Multiclass Classification in CNNs

   Parameters:

     'number_of_classes' : non-zero, positive, int
       Number of classes present in the dataset

     Notes:

       - 'actual_labels' even though not one hot encoded, are used in
calculations in such a way that they work like one hot encoded values.

 '''
```

Evaluation Metrics:

These are just functions whose handles need to be passed.

Classification Accuracy:

```
def ClassAccu(activations, actual_labels):
 '''
    Returns number of correctly classified examples in the batch

    Parameters:

      'activations' : numpy array of shape (batch, number of classes)
        activations of output layer (layer before loss function).

      'actual_labels' : numpy array of shape (batch,1)
        actual labels of each example

     Notes:

       - 'actual_labels', even though not one hot encoded, are used in
calculations in such a way that they work like one hot encoded values.

 '''
```

Confusion Matrix:

```
def ConfMatrix(activations, actual_labels):
 '''
    Returns Confusion Matrix (numpy array) of shape (number of classes,
number of classes)
        Rows represent predicted labels and Columns represent actual
labels.

    Parameters:

      'activations' : numpy array of shape (batch, number of classes)
```

```
        activations of output layer (layer before loss function).

      'actual_labels' : numpy array of shape (batch,1)
        actual labels of each example

   Notes:

   - 'actual_labels', even though not one hot encoded, are used in
calculations in such a way that they work like one hot encoded values.
 '''
```

Optimizers:

Gradient Descent:

```
class GD:
 '''
   Returns an object of "GD" class that can be used as optimizer in CNNs.
   It uses Vanilla Gradient Descent algorithm to update the parameters.
   Further, it may be noted that it uses L2 regularization

 '''
```

Gradient Descent with Momentum:

```
class GDM:
'''
   Returns an object of "GDM" class that can be used as optimizer in CNNs.
   It uses Gradient Descent with Momentum algorithm to update the
parameters.
   Further, it may be noted that it uses L2 regularization

   Parameters:

     'rho' : int or float
       value of "friction"
 '''
```

Nesterov Momentum:

```
class NM:
 '''
   Returns an object of "NM" class that can be used as optimizer in CNNs.
   It uses Nestrov Momentum algorithm to update the parameters.
   Further, it may be noted that it uses L2 regularization

   Parameters:

     'rho' : int or float
       value of "friction"
 '''
```

AdaGrad:

```
class AG:
 '''
   Returns an object of "AG" class that can be used as optimizer in CNNs.
   It uses AdaGrad algorithm to update the parameters.
   Further, it may be noted that it uses L2 regularization

 '''
```

RMSProp:

```
class RP:
 '''
   Returns an object of "RP" class that can be used as optimizer in CNNs.
   It uses RMSProp algorithm to update the parameters.
   Further, it may be noted that it uses L2 regularization

   Parameters:
     'decay_rate' : int or float
       value of rate of decay of learning rate
 '''
```

Adam:

```
class AD:
 '''
   Returns an object of "AD" class that can be used as optimizer in CNNs.
   It uses Adam algorithm to update the parameters.
   Further, it may be noted that it uses L2 regularization

   Parameters:

     'beta1' : int or float

     'beta2' : int or float

 '''
```

Network:

```
class Network:

 '''
   Returns an object of 'Network' class that is used for building and
training the CNN model.

   Parameters:

     'layers' : list of class objects.
       This list defines the architecture of the network.
       It can be have objects of classes:
         'Conv', 'ReLU', 'Tanh', 'LeakyReLU', 'Pool', 'FC', 'Softmax' or
'CELoss'.


     'weights_init' : int, float, 'Gauss', 'Xavier' or 'He'
```

```
       This paramter determines how to initialize the weights in the
network.
       It can take one of the following values:

           int or float: all weights will be initialized with this value
           'Gauss' : pick random numbers from simple Gaussian Distribution
with specified mean and standard deviation
           'Xavier' : same as 'Gauss' except mean = 0 and standard
deviation = 1/sqrt(number of input neurons)
           'He' : same as 'Gauss' except mean = 0 and standard deviation =
1/sqrt(number of input nerons/2)

     'bias_init' : int, float or 'Gauss'
       This parameter determines how to initialize the biases in the
network.

     'mean' : int / float (not optional when using 'Gauss')
       Mean of the Gaussian distribution

     'std' : non negative float (not optional when using 'Gauss')
       Standard Deviation of the Gaussian distribution

     'optimizer' : object of a class
       The methodology to use for updating weights. It can be an object of
one of the following classes:
         'GD', 'GDM', 'NM', 'AG', 'RP', or 'AD'

     'eval_metric' : list of functions: ClassAccu or ConfMatrix
(optional).
       metrics on which the model will be evaluated. The list can contain
upto two values. For more info, see notes.

   Notes:

     - For a CNN with two convolution layers, two pooling layers and two
fully connected layer with ReLU as activation function
       and Cross entropy as Loss function, the parameter 'layers' with
these classes' objects would look like as follows:
```

```
         [Conv, ReLU, Pool, Conv, ReLU, Pool, FC, ReLU, FC, Softmax,
CELoss]

       The layer before loss function is output layer and it must have
neurons equal to number of classes.

     - When using 'Gauss', weights and biases will use same mean and same
standard deviation for initialization.
       When using only for weights (or biases), they must be provided.

     - 'Xavier' performs Xavier initialization and 'He' performs He
initialization of weights. In either case, parameter 'mean'
       and 'std' are not required. If provided, they will be overwritten.

     - When using 'Xavier' or 'He', the number of input neurons for
Convolution layer will be equal to filter_size*filter_size*channels
       and for Fully Connected layer, it will be equal to number of
neurons in the previous layer.

     - If no value is passed to 'eval_metric', no evaluation method will
be used. This parameter must be a list.
       It can take any one of the forms:
[ClassAccu],[ConfMatrix],[ClassAccu, ConfMatrix] or [ConfMatrix,ClassAccu]

 '''
 def __init__(self, layers, weights_init, bias_init,  optimizer,
mean=None, std=None, eval_metric=None):
```

Training:

For training this method must be called on object of Network class

```
def train(self, data, train_batch_size, valid_batch_size, epcohs, lmbda,
lr, grad_check=False):
   '''
     Performs training of the model and returns training and validation
cost and accuracy for each iteration

     Parameters:
       data: object of class 'Dataset'
         the data which will be used for training and validation

       train_batch_size: int
         size of batch for training data. This determines the number of
examples after which weights wiil be updated

       valid_batch_size: int
         size of batch for validation data.

       epochs: int
         number of times whole dataset needs to be passed through the
network during training

       lmbda: int or float
         value of regulariztion parameter.

       lr: int or float
         value of learning rate

       grad_check: bool
         whether to calculate and check numerically calculated gradients
with analytically calculated gradients

     Notes:

       - The optimizer algorithm uses L2 regularization.

   '''
```