Tutorial September 26, 2019

This document contains rough code outline that describes several key steps in training and evaluating a neural network model, using PyTorch. It describes the dataset class, the trainloader class, and the training and evaluation loop.

Please note that this code HAS NOT been compiled and run, and so may contain errors.

```
# Assume that, at this point, we have a DataFrame object, called allData,
# that contains both the input data/features (all in numerical form
# at this point, having converted any categorical features
# to 1-hot and normalized the continuous features) and the
# labels for each sample (= row in the original csv file)
# To be clear, to get to this point you will need to have done the
# processing described in Assignment 3 from Section 3.1 through 3.6

# Let's keep it simple, and assume there are only 4 numerical
# input features and one (binary) output *label* which is
# either 0 or 1;  (in assignment 3 you should have over 100 input
# features)


# This data needs to be split in two ways:
# 1. We need to split the set into training and validation sets
# 2. We need to separate the input features and output labels
# We should also convert these into numpy array types, as that is the
# input to the Dataloader class requires


import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

import torch
from torch.utils.data import DataLoader

# Below is also rough outline of the model definition
# Previous descriptions in class should help you with this part

class MultiLayerPerceptron(nn.Module):

    def __init__(self, input_size, Other_NN_HyperParameters):

        super(MultiLayerPerceptron, self).__init__()

        self.input_size = input_size
        self.output_size = 1

     #  YOU WILL NEED OTHER FUNCTIONS NEEDED TO DEFINE THE MODEL
     #  HERE, as discussed previously

    def forward(self, features):

```

```
        x = SOMETHING (features)
        x = SOMETHINGELSE (x)

      #  CONTINUE TO DEFINE MODEL as discussed previously in Lecture 8

        return x

# Separate features from labels

# 'labelcol' is the heading in the csv file of the binary encoded label
allLabels = allData["labelcol"]

#convert DataFrame object to numpy array because dataloader requires
#a numpy array type as its input

allLabels = allLabels.values

# remove label column from DataFrame object, leaving just the features

features  = allData.drop(columns=["labelcol"])

features = features.values  # also convert DataFrame to numpy 2D array


# now, separate into training and validation set, randomly,
# With 20% going to validation set
# Should set random seed so that program is the same each execution

seed = 0

feat_train, feat_valid, label_train, label_valid =

      train_test_split(features, labels, test_size=0.2,random_state=seed)


# Need to set up the DataLoader - which feeds the
# training and validation loops with the data
#
# DataLoader requires the Dataset object:
# A Dataset contains both the features and the labels
# it requires the methods: __init__,  __len__ (number of features)
# and __getitem_ - get one sample

import torch.utils.data as data

class myDataset(data.Dataset):

    def __init__(self, features, label):

        # these must be numpy array as mentioned above
        # this gives methods in this class access to features and labels

        self.features = features
        self.label = label
```

```

    def __len__(self):
        return len(self.features)

    # getitem returns an individual sample's features and label
    # if there is more than one feature, it should be a numpy array

    def __getitem__(self, index):
        features  = self.features[index]

        label =self.label[index]

        return features, label

# The code below sets up the separate training and validation datasets
# putting them into a DataLoader object

# First, instantiate an object in the dataset class defined above, and
# fill it with the training data

train_dataset = myDataset(feat_train, label_train)

# create a callable object that will provide 'batches' of the samples
# later on, when asked for the training data
# this converts (invisibly) the data to be PyTorch-compatible tensors
# This is also where the batch_size is set
# Shuffle = True re-orders data every Epoch

train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)

# similarly, do the same for the validation data set.

valid_dataset = myDataset(feat_valid, label_valid)

valid_loader = DataLoader(valid_dataset, batch_size=batch_size,
shuffle=False)

# skip validation function below for the moment,
# will return to it after training loop

# a function to run the validation data set through the model
# This will be done every so often during the training loop below

def evaluate(model, valid_loader):
    total_corr = 0

    for i, vbatch in enumerate(valid_loader):
        feats, label = vbatch     #feats will have shape (batch_size,4)

      # run the neural network model on the data!
      # note that there are batch_size samples being run through
      # the model, not just one sample

```

```
        prediction = model(feats)

      # if a prediction is OVER 0.5 - that is considered to be 1
      # otherwise answer is 0.
      # Squeeze: shape is (batchsize,1) results - don't want
      # that 1 dimension
      # use long because that is the basic integer type

        corr = (prediction > 0.5).squeeze().long() == label

      # sum up the number of correct predictions

        total_corr += int(corr.sum())

    return float(total_corr)/len(valid_loader.dataset)

# NOW, heading towards the training loop!!!

# Choose the loss function to be Mean Squared Errir
# it is a callable object; will describe intuition later.

loss_function = torch.nn.MSELoss()

# Instantiate the callable object that is the neural NETWORK model
# defined in the model section above
#


model = MultiLayerPerceptron(4, OTHERPARAMETERS....)

# Choose the optimization method - Stochastic Gradient Descent
# model.parameters() contains all of the weights and biases defined in
# the model definition

# lr is the value of the learning rate that you're setting

optimizer = torch.optim.SGD(model.parameters(), lr=args.learningrate)


#
#  THE TRAINING OPTIMIZING LOOP
#

    t = 0   # used to count batch number putting through the model


    for epoch in range(MaxEpochs): # Epoch: one pass through all training
        accum_loss = 0
        tot_corr = 0

        for i, batch in enumerate(train_loader):    #from DataLoader
          # this gets one "batch" of data;p

            feats, label = batch  #feats will have shape (batch_size,4)
```

```

            # need to send batch through model and do a gradient opt step;
            # first set all gradients to zero

              optimizer.zero_grad()

            # Run the neural network model on the batch, and get answers

              predictions = model(feats)   # has shape (batch_size,1)

            # compute the loss function (MSE as above) using the
            # correct answer for the entire batch
            # label was an int, needs to become a float

              batch_loss = loss_fnc(input=predictions.squeeze(),
target=label.float())

            accum_loss += batch_loss

            # computes the gradient of loss with respect to the parameters
            # pytorch keeps all kinds of information in the Tensor object
            # to make this possible;  uses back-propagation

              batch_loss.backward()

            # Change the parameters  in the model with one 'step' guided by
            # the learning rate.  Recall parameters are the weights & bias

              optimizer.step()

            # calculate number of correct predictions

              corr = (predictions > 0.5).squeeze().long() == label

              tot_corr += int(corr.sum())


            # evaluate model on the validation set every eval_every steps

              if (t+1) % args.eval_every == 0:
                  valid_acc = evaluate(model, valid_loader)

                  print("Epoch: {}, Step {} | Loss: {}| Valid acc:
{}".format(epoch+1, t+1, accum_loss / eval_every, valid_acc))


                  accum_loss = 0

              t = t + 1
# output final, post-training training accuracy

print("Train acc:{}".format(float(tot_corr)/len(train_loader.dataset)))
```