Optimization_HNFG_IMDB_V1

April 27, 2021

1 Computational Intelligence Project: Sentiment Analysis on IMDB dataset (Part III)

[Musab - 19030008]

In this Notebook, I have done implementing Hybrid Neuro Genetic Fuzzy System. In this approach, an optimization is applied to Neuro-fuzzy inference system using genetic algorithm. Neuro-fuzzy is also called ANFIS. Genetic Algorithm is used to optimize the hybrid model using different parameters i.e., 1. Number of layers 2. Number of parameters in Dense Layer specificly. 3. Different optimizers i.e., Adam, RMSProp,Adagrad, SDG 4. Different Activation Functions i.e., Sigmoid, Relu

1.0.1 Imports

[ ]:
```
#%%
import re
import keras
import numpy as np
import random
import logging
from tqdm import tqdm
from functools import reduce
from operator import add
from keras.datasets import imdb
from keras import regularizers
from keras import backend as K
from keras.engine.topology import Layer
import matplotlib.pyplot as plt
from keras.layers import Input, Dense, Dropout
from tensorflow.python.client import device_lib
from keras.utils import to_categorical
from keras.models import Model, Sequential
from keras.models import load_model
import matplotlib.pyplot as plt
from sklearn.metrics import f1_score, confusion_matrix
from keras.callbacks import EarlyStopping
```

1.1 Step 1 | Loading Dataset

[ ]:
```
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=5000)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/imdb.npz
17465344/17464789 [==============================] - 0s 0us/step

<string>:6: VisibleDeprecationWarning: Creating an ndarray from ragged nested
sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with
different lengths or shapes) is deprecated. If you meant to do this, you must
specify 'dtype=object' when creating the ndarray
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/keras/datasets/imdb.py:159:
VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences
(which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths
or shapes) is deprecated. If you meant to do this, you must specify
'dtype=object' when creating the ndarray
  x_train, y_train = np.array(xs[:idx]), np.array(labels[:idx])
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/keras/datasets/imdb.py:160:
VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences
(which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths
or shapes) is deprecated. If you meant to do this, you must specify
'dtype=object' when creating the ndarray
  x_test, y_test = np.array(xs[idx:]), np.array(labels[idx:])
```

Dataset statistics

[ ]:
```
print("train_data ", x_train.shape)
print("train_labels ", y_train.shape)
print("_"*100)
print("test_data ", x_test.shape)
print("test_labels ", y_test.shape)
print("_"*100)
print("Maximum value of a word index ")
print(max([max(sequence) for sequence in x_train]))
print("Maximum length num words of review in train ")
print(max([len(sequence) for sequence in x_train]))
```

```
train_data  (25000,)
train_labels  (25000,)
________________________________________________________________________________
____________________
test_data  (25000,)
test_labels  (25000,)
________________________________________________________________________________
____________________
Maximum value of a word index
```

```
4999
Maximum length num words of review in train
2494
```

1.2 Step 2 | Splitting Dataset

As dataset contains the 50,000 reviews and is classified to positive and negative classes. Genetic Algorithm expands the training as it tries multiple generation, and population to ooptimize the network.

[ ]:
```
x_train = x_train[:10000]
y_train = y_train[:10000]
print(x_train.shape)

x_test = x_test[:10000]
y_test = y_test[:10000]
print(x_test.shape)
```

```
(10000,)
(10000,)
```

Vectorizing the input makes the learning of model faster that's why i have applied Vectorization

[ ]:
```
def vectorize_sequences(sequences, dimension=5000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results
```

[ ]:
```
x_train = vectorize_sequences(x_train)
x_test = vectorize_sequences(x_test)

print("x_train ", x_train.shape)
print("x_test ", x_test.shape)
```

```
x_train  (10000, 5000)
x_test  (10000, 5000)
```

[ ]:
```
y_train = np.asarray(y_train).astype('float32')
y_test = np.asarray(y_test).astype('float32')
print("y_train ", y_train.shape)
print("y_test ", y_test.shape)
```

```
y_train  (10000,)
y_test  (10000,)
```

1.3 Step 3 | Fuzzy System & Neural Network

1.3.1 Fuzzy Layer | Custom layer

[ ]:
```
class FuzzyLayer(Layer):

    def __init__(self,
                 output_dim,
                 initializer_centers=None,
                 initializer_sigmas=None,
                 **kwargs):
        if 'input_shape' not in kwargs and 'input_dim' in kwargs:
            kwargs['input_shape'] = (kwargs.pop('input_dim'),)
        self.output_dim = output_dim
        self.initializer_centers = initializer_centers
        self.initializer_sigmas = initializer_sigmas
        super(FuzzyLayer, self).__init__(**kwargs)

    def build(self, input_shape):
        self.input_dimensions = list(input_shape)[:-1:-1]
        self.c = self.add_weight(name='c',
                                 shape=(input_shape[-1], self.output_dim),
                                 initializer= self.initializer_centers if self.
↪initializer_centers is not None else 'uniform',
                                 trainable=True)
        self.a = self.add_weight(name='a',
                                 shape=(input_shape[-1], self.output_dim),
                                 initializer=self.initializer_sigmas if self.
↪initializer_sigmas is not None else 'ones',
                                 trainable=True)
        super(FuzzyLayer, self).build(input_shape)

    def call(self, x):

        aligned_x = K.repeat_elements(K.expand_dims(x, axis = -1), self.
↪output_dim, -1)
        aligned_c = self.c
        aligned_a = self.a
        for dim in self.input_dimensions:
            aligned_c = K.repeat_elements(K.expand_dims(aligned_c, 0), dim, 0)
            aligned_a = K.repeat_elements(K.expand_dims(aligned_a, 0), dim, 0)

        xc = K.exp(-K.sum(K.square((aligned_x - aligned_c) / (2 * aligned_a)),␣
↪axis=-2, keepdims=False))
        #sums = K.sum(xc,axis=-1,keepdims=True)
        #less = K.ones_like(sums) * K.epsilon()
        return xc# xc / K.maximum(sums, less)
```

```
    def compute_output_shape(self, input_shape):
        return tuple(input_shape[:-1]) + (self.output_dim,)
```

1.3.2 Neural Network

[ ]:
```
class Network():
  def __init__(self, params=None):
    """
    Arguments:
        Parameters for the network, includes:
            nb_neurons (list): [64, 128 ..]
            nb_layers (list): [1, 2 .. ]
            activation (list): ['relu'...]
            optimizer (list): ['adam'..]
    """
    self.accuracy = 0.
    self.params = params
    self.network = {}

  def create_random(self):
      for key in self.params:
          self.network[key] = random.choice(self.params[key])

  def create_set(self, network):
      self.network = network

  def train(self, dataset):
      if self.accuracy == 0.:
          self.accuracy = train_and_score(self.network, dataset)

  def print_network(self):
      logging.info(self.network)
      logging.info("Network accuracy: %.2f%%" % (self.accuracy * 100))
```

1.3.3 Training Adaptation

[ ]:
```
early_stopper = EarlyStopping(patience=5)

def compile_model(network, input_shape):

    # Get our network parameters.
    nb_layers = network['nb_layers']
    nb_neurons = network['nb_neurons']
    activation = network['activation']
    optimizer = network['optimizer']
```

```
    model = Sequential()

    for i in range(nb_layers):
        # Need input shape for first layer.
        if i == 0:
            model.add(Dense(nb_neurons, activation=activation,␣
↪input_shape=input_shape))
        else:
            model.add(Dense(nb_neurons, activation=activation))

        model.add(Dropout(0.2))  # hard-coded dropout
    model.add(Dense(2,activation=activation))
    model.add(FuzzyLayer(100))

    model.add(Dense(1, activation='linear'))

    model.compile(loss='mse', optimizer=optimizer,
                  metrics=['accuracy'])

    return model

def train_and_score(network, dataset):

    batch_size = 32
    input_shape = (5000,)
    x_train, x_test, y_train, y_test = dataset

    model = compile_model(network, input_shape)

    model.fit(x_train, y_train,
              batch_size=batch_size,
              epochs=100,
              verbose=1,
              validation_data=(x_test, y_test),
              callbacks=[early_stopper])

    score = model.evaluate(x_test, y_test, verbose=1)

    return score[1]
```

1.4 Step 4 | Genetic Algorithm

For implementaion i have followed this tutorial; http://lethain.com/genetic-algorithms-cool-name-damn-simple/

[ ]:

```
class GeneticAlgorithm():
    def __init__(self, params, retain=0.4,random_select=0.1, mutate_chance=0.2):
        """
        Arguments:
            params: Possible network paremters
            retain: Percentage of population to retain after each generation
            random_select : Probability of a rejected network remaining in the␣
 ↪population
            mutation: Probability a network will be randomly mutated
        """
        self.mutate_chance = mutate_chance
        self.random_select = random_select
        self.retain = retain
        self.params = params

    def create_population(self, count):

        pop = []
        for _ in range(0, count):
            # Create a random network.
            network = Network(self.params)
            network.create_random()

            # Add the network to our population.
            pop.append(network)

        return pop

    @staticmethod
    def fitness(network):
        """Return the accuracy, which is our fitness function."""
        return network.accuracy

    def grade(self, pop):
        summed = reduce(add, (self.fitness(network) for network in pop))
        return summed / float((len(pop)))

    def breed(self, mother, father):
        children = []
        for _ in range(2):
            child = {}
            # Loop through the parameters and pick params for the kid.
            for param in self.params:
                child[param] = random.choice(
                    [mother.network[param], father.network[param]]
                )
```

```
            # Now create a network object.
            network = Network(self.params)
            network.create_set(child)

            # Randomly mutate some of the children.
            if self.mutate_chance > random.random():
                network = self.mutate(network)

            children.append(network)

        return children

    def mutate(self, network):
        # Choose a random key.
        mutation = random.choice(list(self.params.keys()))

        # Mutate one of the params.
        print("Mutation!!")
        network.network[mutation] = random.choice(self.params[mutation])

        return network

    def evolve(self, pop):
        # Get scores for each network.
        graded = [(self.fitness(network), network) for network in pop]

        # Sort on the scores.
        graded = [x[1] for x in sorted(graded, key=lambda x: x[0], 
↪reverse=True)]

        # Get the number we want to keep for the next gen.
        retain_length = int(len(graded)*self.retain)

        # The parents are every network we want to keep.
        parents = graded[:retain_length]

        # For those we aren't keeping, randomly keep some anyway.
        for individual in graded[retain_length:]:
            if self.random_select > random.random():
                parents.append(individual)

        # Now find out how many spots we have left to fill.
        parents_length = len(parents)
        desired_length = len(pop) - parents_length
        children = []

        # Add children, which are bred from two remaining networks.
```

```
        while len(children) < desired_length:

            # Get a random mom and dad.
            male = random.randint(0, parents_length-1)
            female = random.randint(0, parents_length-1)

            # Assuming they aren't the same network...
            if male != female:
                male = parents[male]
                female = parents[female]

                # Breed them.
                babies = self.breed(male, female)
                print("Evolution!!")
                # Add the children one at a time.
                for baby in babies:
                    # Don't grow larger than desired length.
                    if len(children) < desired_length:
                        children.append(baby)

        parents.extend(children)
        return parents
```

[ ]:
```
logging.basicConfig(
    format='%(asctime)s - %(levelname)s - %(message)s',
    datefmt='%m/%d/%Y %I:%M:%S %p',
    level=logging.DEBUG,
    filename='log.txt'
)

def train_networks(networks, dataset):
    """Train each network.
    Args:
        networks (list): Current population of networks
        dataset (str): Dataset to use for training/evaluating
    """
    pbar = tqdm(total=len(networks))
    for network in networks:
        network.train(dataset)
        pbar.update(1)
    pbar.close()

def get_average_accuracy(networks):
    # Getting accuracy for a group of networks.

    total_accuracy = 0
    for network in networks:
```

```python
        total_accuracy += network.accuracy

    return total_accuracy / len(networks)

def generate(generations, population, params, dataset):
    """
    Generate a network with the genetic algorithm.
    Args:
        generations: Number of times to evole the population
        population : Number of networks in each generation
        params: Parameter choices for networks
        dataset: Dataset to use for training/evaluating
    """
    ga = GeneticAlgorithm(params)
    networks = ga.create_population(population)

    # Evolve the generation.
    for i in range(generations):
        logging.info("***Doing generation %d of %d***" % (i + 1, generations))

        # Train and get accuracy for networks.
        train_networks(networks, dataset)

        # Get the average accuracy for this generation.
        average_accuracy = get_average_accuracy(networks)

        # Print out the average accuracy each generation.
        logging.info("Generation average: %.2f%%" % (average_accuracy * 100))
        logging.info('-'*80)

        # Evolve, except on the last iteration.
        if i != generations - 1:
            # Do the evolution.
            networks = ga.evolve(networks)

    # Sort our final population.
    networks = sorted(networks, key=lambda x: x.accuracy, reverse=True)

    # Print out the top 5 networks.
    print_networks(networks[:5])

def print_networks(networks):

    logging.info('-'*80)
    for network in networks:
        network.print_network()
```