CRNN

April 4, 2022

Resources:

Convolutional recurrent neural networks:

https://github.com/bgshih/crnn

paper: https://arxiv.org/pdf/1507.05717.pdf

pytorch implementation: https://github.com/meijieru/crnn.pytorch

ocr image dataset: http://iapr-tc11.org/mediawiki/index.php?title=ICDAR_2003_Robust_Reading_Competitio

Recurrence is required to capture the representational dynamics of the human visual system: https://www.pnas.org/doi/pdf/10.1073/pnas.1905544116 https://cbmm.mit.edu/video/its-about-time-modelling-human-visual-inference-deep-recurrent-neural-networks

Other related resources:

Recurrent Attention Models: https://github.com/kevinzakka/recurrent-visual-attention

https://github.com/Element-Research/rnn/blob/master/examples/recurrent-visual-attention.lua

https://github.com/jlindsey15/RAM

[ ]:
```
import torch
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.python.keras.layers import InputLayer, Input
from keras.models import Model
from keras.layers import Dense
from keras.layers import Flatten
from tensorflow.python.keras import backend as K
import math
import pathlib
import datetime
from scipy.io import savemat
from scipy.io import loadmat
```

```
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
import time
```

[ ]:
```
from google.colab import drive
drive.mount('/content/drive/')
%cd '/content/drive/My Drive/Embeddings/rnn'
```

1 Define CRNN model:

[ ]:
```
import torch.nn as nn


class BidirectionalLSTM(nn.Module):

    def __init__(self, nIn, nHidden, nOut):
        super(BidirectionalLSTM, self).__init__()

        self.rnn = nn.LSTM(nIn, nHidden, bidirectional=True)
        self.embedding = nn.Linear(nHidden * 2, nOut)

    def forward(self, input):
        recurrent, _ = self.rnn(input)
        T, b, h = recurrent.size()
        t_rec = recurrent.view(T * b, h)

        output = self.embedding(t_rec)  # [T * b, nOut]
        output = output.view(T, b, -1)

        return output


class CRNN(nn.Module):

    def __init__(self, imgH, nc, nclass, nh, n_rnn=2, leakyRelu=False):
        super(CRNN, self).__init__()

        assert imgH % 16 == 0, 'imgH has to be a multiple of 16'

        ks = [3, 3, 3, 3, 3, 3, 2]
        ps = [1, 1, 1, 1, 1, 1, 0]
        ss = [1, 1, 1, 1, 1, 1, 1]
        nm = [64, 128, 256, 256, 512, 512, 512]

        cnn = nn.Sequential()
```

```
        def convRelu(i, batchNormalization=False):
            nIn = nc if i == 0 else nm[i - 1]
            nOut = nm[i]
            cnn.add_module('conv{0}'.format(i),
                           nn.Conv2d(nIn, nOut, ks[i], ss[i], ps[i]))
            if batchNormalization:
                cnn.add_module('batchnorm{0}'.format(i), nn.BatchNorm2d(nOut))
            if leakyRelu:
                cnn.add_module('relu{0}'.format(i),
                               nn.LeakyReLU(0.2, inplace=True))
            else:
                cnn.add_module('relu{0}'.format(i), nn.ReLU(True))

        convRelu(0)
        cnn.add_module('pooling{0}'.format(0), nn.MaxPool2d(2, 2))  # 64x16x64
        convRelu(1)
        cnn.add_module('pooling{0}'.format(1), nn.MaxPool2d(2, 2))  # 128x8x32
        convRelu(2, True)
        convRelu(3)
        cnn.add_module('pooling{0}'.format(2),
                       nn.MaxPool2d((2, 2), (2, 1), (0, 1)))  # 256x4x16
        convRelu(4, True)
        convRelu(5)
        cnn.add_module('pooling{0}'.format(3),
                       nn.MaxPool2d((2, 2), (2, 1), (0, 1)))  # 512x2x16
        convRelu(6, True)  # 512x1x16

        self.cnn = cnn
        self.rnn = nn.Sequential(
            BidirectionalLSTM(512, nh, nh),
            BidirectionalLSTM(nh, nh, nclass))

    # def init_hidden(self,):
    #     return (torch.zeros(512, self.nh, self.nh))

    def forward(self, input):
        # conv features
        conv = self.cnn(input)
        b, c, h, w = conv.size()
        assert h == 1, "the height of conv must be 1"
        conv = conv.squeeze(2)
        conv = conv.permute(2, 0, 1)  # [w, b, c]
        # self.hidden = self.init_hidden()
        # self.hidden = self.rnn[0](input, self.hidden)

        # rnn features
        output = self.rnn(conv)
```

```
        return output
```

[ ]:
```
import torch
from torch.autograd import Variable
# import utils
# import dataset
from PIL import Image


model_path = 'crnn.pth'
img_path = 'demo.png'
alphabet = '0123456789abcdefghijklmnopqrstuvwxyz'
```

[ ]:
```
model = CRNN(32, 1, 37, 256)
if torch.cuda.is_available():
    model = model.cuda()
print('loading pretrained model from %s' % model_path)
model.load_state_dict(torch.load(model_path))
```

[ ]:
```
import torch
import torch.nn as nn
from torch.autograd import Variable
# import collections


class strLabelConverter(object):
    """Convert between str and label.
    NOTE:
        Insert `blank` to the alphabet for CTC.
    Args:
        alphabet (str): set of the possible characters.
        ignore_case (bool, default=True): whether or not to ignore all of the␣
↪case.
    """

    def __init__(self, alphabet, ignore_case=True):
        self._ignore_case = ignore_case
        if self._ignore_case:
            alphabet = alphabet.lower()
        self.alphabet = alphabet + '-'  # for `-1` index

        self.dict = {}
        for i, char in enumerate(alphabet):
            # NOTE: 0 is reserved for 'blank' required by wrap_ctc
            self.dict[char] = i + 1
```

```python
    def encode(self, text):
        """Support batch or single str.
        Args:
            text (str or list of str): texts to convert.
        Returns:
            torch.IntTensor [length_0 + length_1 + ... length_{n - 1}]: encoded␣
↪texts.
            torch.IntTensor [n]: length of each text.
        """
        if isinstance(text, str):
            text = [
                self.dict[char.lower() if self._ignore_case else char]
                for char in text
            ]
            length = [len(text)]
        elif isinstance(text, collections.Iterable):
            length = [len(s) for s in text]
            text = ''.join(text)
            text, _ = self.encode(text)
        return (torch.IntTensor(text), torch.IntTensor(length))

    def decode(self, t, length, raw=False):
        """Decode encoded texts back into strs.
        Args:
            torch.IntTensor [length_0 + length_1 + ... length_{n - 1}]: encoded␣
↪texts.
            torch.IntTensor [n]: length of each text.
        Raises:
            AssertionError: when the texts and its length does not match.
        Returns:
            text (str or list of str): texts to convert.
        """
        if length.numel() == 1:
            length = length[0]
            assert t.numel() == length, "text with length: {} does not match␣
↪declared length: {}".format(t.numel(), length)
            if raw:
                return ''.join([self.alphabet[i - 1] for i in t])
            else:
                char_list = []
                for i in range(length):
                    if t[i] != 0 and (not (i > 0 and t[i - 1] == t[i])):
                        char_list.append(self.alphabet[t[i] - 1])
                return ''.join(char_list)
        else:
            # batch mode
```

```
            assert t.numel() == length.sum(), "texts with length: {} does not␣
↪match declared length: {}".format(t.numel(), length.sum())
            texts = []
            index = 0
            for i in range(length.numel()):
                l = length[i]
                texts.append(
                    self.decode(
                        t[index:index + l], torch.IntTensor([l]), raw=raw))
                index += l
            return texts
```

[ ]:
```
import torchvision.transforms as transforms
class resizeNormalize(object):

    def __init__(self, size, interpolation=Image.BILINEAR):
        self.size = size
        self.interpolation = interpolation
        self.toTensor = transforms.ToTensor()

    def __call__(self, img):
        img = img.resize(self.size, self.interpolation)
        img = self.toTensor(img)
        img.sub_(0.5).div_(0.5)
        return img
```

[ ]:
```
converter = strLabelConverter(alphabet)
image = Image.open(img_path).convert('L')
transformer = resizeNormalize((100, 32))
image = transformer(image)
if torch.cuda.is_available():
    image = image.cuda()
image = image.view(1, *image.size())
image = Variable(image)
```

[ ]:
```
model.eval()
```

2 Make images:

[ ]:
```
images_selected_classes = []
num_images = 20
for i in range(num_images):
  img_path = str(i+1) + '.jpg'
  converter = strLabelConverter(alphabet)
  image = Image.open(img_path).convert('L')
  transformer = resizeNormalize((100, 32))
```

```
  image = transformer(image)
  image = image.view(1, *image.size())
  image = Variable(image)
  images_selected_classes.append(image)
images_selected_classes = np.vstack(images_selected_classes)
images_selected_classes.shape
```

3 Predict one demo image and take output from intermediate rnn-lstm layer:

[ ]:
```
Image.open('demo.png')
```

3.1 output of lstm after embedding:

[ ]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
image.to(device)
model.to(device)
```

[ ]:
```
layer_index = 0
cnn_out = model.cnn(image)
print('cnn shape:')
print(cnn_out.shape)
## before the linear embedding step:
lstm_before_embedding = model.rnn[layer_index].rnn
b, c, h, w = cnn_out.size()
assert h == 1, "the height of conv must be 1"
cnn_out = cnn_out.squeeze(2)
cnn_out = cnn_out.permute(2, 0, 1)  # [w, b, c]
all_hidden_states, (last_hidden, last_cell) = lstm_before_embedding(cnn_out)

# # last_hidden.shape: [2, 1, 256]
# # last_cell.shape: [2, 1, 256]
# # all_hidden_states.shape: [26, 1, 512]
```

[ ]:
```
all_hidden_states.shape
```

[ ]:
```

```

3.2 output of lstm after embedding:

[ ]:
```
layer_index = 0
print(image.shape)

activation = {}
def get_activation(name):
```

```
    def hook(model, input, output):
      activation[name] = output.detach()
      handle.remove()
    return hook
# register the forward hook
handle = model.rnn[layer_index].embedding.
 ↪register_forward_hook(get_activation('encoder_queried_layer'))
# pass some data through the model
output = model(image)
layer_out = activation['encoder_queried_layer']
handle.remove()
print(layer_out)

# https://discuss.pytorch.org/t/understanding-output-of-lstm/12320
# https://pytorch.org/docs/master/generated/torch.nn.LSTM.html#torch.nn.LSTM
# https://stackoverflow.com/questions/48302810/
 ↪whats-the-difference-between-hidden-and-output-in-pytorch-lstm

cnn_out = model.cnn(image)
print('cnn shape:')
print(cnn_out.shape)
lstm = model.rnn[layer_index]
b, c, h, w = cnn_out.size()
assert h == 1, "the height of conv must be 1"
cnn_out = cnn_out.squeeze(2)
cnn_out = cnn_out.permute(2, 0, 1)  # [w, b, c]

# all_hidden_states, (last_hidden, last_cell) = lstm(cnn_out)
print(lstm(cnn_out))
handle.remove()
```

[ ]:
```
layer_out.shape
```

[ ]:
```
handle.remove()
```

[ ]:
```
image.shape
```

[ ]:
```
preds = model(image)
print(preds.shape)
```

[ ]:
```
type(preds)
```

[ ]:
```
_, preds = preds.max(2)
preds.shape
```

[ ]:
```
preds = preds.transpose(1, 0).contiguous().view(-1)
preds
```

```
[ ]: preds_size = Variable(torch.IntTensor([preds.size(0)]))
     raw_pred = converter.decode(preds.data, preds_size.data, raw=True)
     sim_pred = converter.decode(preds.data, preds_size.data, raw=False)
     print('%-20s => %-20s' % (raw_pred, sim_pred))
```

4 Create 2D tensor:

```
[ ]: def apply_all_shifts(im, shift_step):
         '''
         arg(s):
             DEBUG NOTE: PYTORCH IS (#channels, rows, cols)!!
             im, an image of shape (3, im_size, im_size)
         return:
             im_all_shifts, a list of all shifted images from the input image
             n_shifts, number of shifted images
         '''
         # im is of shape (1, 32, 100)

         ## vertical size might not be the same as the horizontal, note that the␣
      ↪channel for tf and pytorch are in different dimension
         im_size_vertical = im.shape[1]
         im_size_horizontal = im.shape[2]

         n_shifts_vertical = int(math.ceil(im_size_vertical/ shift_step))
         n_shifts_horizontal = int(math.ceil(im_size_horizontal/ shift_step))
         n_shifts = n_shifts_vertical * n_shifts_horizontal

         im_all_shifts = []
         # start with the unshifted im
         im_shift = im
         im_all_shifts.append(np.expand_dims(im_shift,axis=0))
         # im_all_shifts.append(np.expand_dims(im_shift,axis=0))
         for i in range(n_shifts_vertical):
             ## for pytorch axis = 1, for tensorflow axis = 0
             im_shift = np.roll(im_shift, shift = shift_step, axis=1)
             im_all_shifts.append(np.expand_dims(im_shift,axis=0))
             for j in range(n_shifts_horizontal):
                 ## for pytorch axis = 2, for tensorflow axis = 1
                 im_shift = np.roll(im_shift, shift = shift_step, axis=2)
                 im_all_shifts.append(np.expand_dims(im_shift,axis=0))

         im_all_shifts = np.stack( im_all_shifts, axis=0 )
         im_all_shifts = torch.cuda.FloatTensor(im_all_shifts)
         return im_all_shifts, n_shifts
```

```
[ ]: def compute_neuron_output(model, block,  layer_index, im_all_shifts, ␣
    ↪max_indices = None, n_max_feature_maps = 500, plot_activity = False):

        '''
        arg(s):
            layer_names: list of strings indicating the names of the layers we want␣
    ↪to take neuron outputs from
            im_all_shifts: all shifts of one particular image in the for loop
        return:

        '''
        print("im_all_shifts shape: ")
        print(im_all_shifts.shape)
        n_shifts = im_all_shifts.shape[0]
        neuron_output_highest = []
        im_all_shifts = torch.cuda.FloatTensor(im_all_shifts)

        neuron_output = []
        # go through all the batches in the dataset
        if block == 'RNN':
          for i in range(n_shifts):
          # forward pass -- getting the outputs
            image = im_all_shifts[i]

            # https://discuss.pytorch.org/t/understanding-output-of-lstm/12320
            # https://pytorch.org/docs/master/generated/torch.nn.LSTM.html#torch.nn.
    ↪LSTM
            # https://stackoverflow.com/questions/48302810/
    ↪whats-the-difference-between-hidden-and-output-in-pytorch-lstm

            # activation = {}
            # def get_activation(name):
            #     def hook(model, input, output):
            #       activation[name] = output.detach()
            #       handle.remove()
            #     return hook
            # # register the forward hook
            # handle = model.rnn[layer_index].embedding.
    ↪register_forward_hook(get_activation('encoder_queried_layer'))
            # # pass some data through the model
            # output = model(image)
            # layer_out = activation['encoder_queried_layer']
            # handle.remove()
            layer_index = 0
            cnn_out = model.cnn(image)
            print('cnn shape:')
            print(cnn_out.shape)
```

```
        ## before the linear embedding step:
        lstm_before_embedding = model.rnn[layer_index].rnn
        b, c, h, w = cnn_out.size()
        assert h == 1, "the height of conv must be 1"
        cnn_out = cnn_out.squeeze(2)
        cnn_out = cnn_out.permute(2, 0, 1)  #[w, b, c]
        all_hidden_states, (last_hidden, last_cell) =␣
↪lstm_before_embedding(cnn_out)
        ## all_hidden_states is of shape [26, 1, 512]
        all_hidden_states = all_hidden_states.squeeze(dim=1)
        neuron_output.append(all_hidden_states.cpu().data)

      neuron_output = np.stack(neuron_output, axis=0)
      print("neuron_output shape: ")
      print(neuron_output.shape)
      n_shifts, n_features, n_neurons =  neuron_output.shape
      ## the shape of neuron_output_by_fm is (n_shifts, n_feature_maps,␣
↪#neurons)
      neuron_output_by_fm = neuron_output
    ## --- END OF RNN ---

    if block == 'CNN':
      for i in range(n_shifts):
      # forward pass -- getting the outputs
        image = im_all_shifts[i]

        activation = {}
        def get_activation(name):
            def hook(model, input, output):
              activation[name] = output.detach()
              handle.remove()
            return hook
        # register the forward hook
        handle = model.cnn.relu1.
↪register_forward_hook(get_activation('encoder_queried_layer'))
        # pass some data through the model
        output = model(image)
        layer_out = activation['encoder_queried_layer']
        handle.remove()

        layer_out = layer_out.cpu().data
        neuron_output.append(layer_out.squeeze(dim=0))
      neuron_output = np.stack( neuron_output, axis=0 )
      print("neuron_output shape: ")
      print(neuron_output.shape)
      n_shifts, n_features, n_rows, n_cols =  neuron_output.shape
      n_neurons = n_rows * n_cols
```

```
    # # ------ IF NEED TO REMOVE THE NEURONS AT THE EDGES ------

    # edge_neuron_row = int(0.1 * n_rows)
    # edge_neuron_col = int(0.1 * n_cols)
    # neuron_output = neuron_output[:, :, edge_neuron_row :(n_rows -␣
↪edge_neuron_row ), edge_neuron_col:(n_cols - edge_neuron_col)]
    # n_shifts, n_features, n_rows, n_cols =  neuron_output.shape
    # n_neurons = n_rows * n_cols
    # print(n_rows)
    # print(n_cols)

    # # ------ END ------

    ## the shape of neuron_output_by_fm is (n_shifts, n_feature_maps,␣
↪#neurons)
    neuron_output = neuron_output.reshape((n_shifts,n_features,n_rows*n_cols))
    neuron_output_by_fm = neuron_output.
↪reshape((n_shifts,n_features,n_rows*n_cols))
  ## --- END OF CNN ---

  ## compute avg neuron firing rate in each feature map
  ## fm_avg is of shape (n_shifts, n_feature_maps)
  fm_avg = neuron_output_by_fm.sum(axis=2) / neuron_output_by_fm.shape[2]

  # if plot_activity == True:
  #     print("Indices of FM with highest average firing rate in response to␣
↪each image: " )
  #     max_fm_ind = np.argmax(fm_avg,axis = 1)
  #     print(max_fm_ind)
  #     print("#neurons in the FM with highest average firing rate: " +␣
↪str(n_neurons))
  #     # for f_i in range(n_features):
  #     #   plt.plot(fm_avg[:,f_i]/max(fm_avg[:,f_i]))
  #     plt.plot(fm_avg[:,0]/max(fm_avg[:,0]))
  #     plt.show()
  #     print('----------------------------------------------------\n')

  # neuron_output_highest = neuron_output_by_fm.reshape((n_shifts, n_features␣
↪* n_neurons))

  neuron_output_highest = np.empty((n_shifts, n_max_feature_maps * n_neurons))

  for i in range(n_shifts):
      if max_indices is None:
          ## get the max magnitude (most negative and most positive firing)
```

```
            max_indices = np.argsort(-1*abs(fm_avg[i]))[:n_max_feature_maps]
        neuron_output_highest[i] = neuron_output_by_fm[i, max_indices, :].
↪reshape((1, n_max_feature_maps * n_neurons))

        for f_i in range(n_max_feature_maps):
          normalizing_constant = abs(neuron_output_highest[i, (f_i*n_neurons):
↪((f_i+1)*n_neurons)]).max()

          if normalizing_constant == 0:
            neuron_output_highest[i, (f_i*n_neurons):((f_i+1)*n_neurons)] = 0
          else:
            neuron_output_highest[i, (f_i*n_neurons):((f_i+1)*n_neurons)] /=␣
↪normalizing_constant

    # if plot_activity == True:
    #     neuron_output_highest_by_fm = neuron_output_highest.
↪reshape((n_shifts, n_max_feature_maps, n_neurons))
    #     fm_avg = neuron_output_highest_by_fm.sum(axis=2) /␣
↪neuron_output_highest_by_fm.shape[2]

    #     print("Indices of FM with highest average firing rate in response to␣
↪each image: " )
    #     max_fm_ind = np.argmax(fm_avg,axis = 1)
    #     print(max_fm_ind)
    #     print("#neurons in the FM with highest average firing rate: " +␣
↪str(n_neurons))
    #     for f_i in range(n_max_feature_maps):
    #       plt.plot(fm_avg[:,f_i])
    #     plt.show()
    #     print('----------------------------------------------------\n')

    neuron_labels = []
    for i in range(n_max_feature_maps):
      neuron_labels += [i] * n_neurons
    neuron_labels = np.array(neuron_labels)

    return neuron_output_highest, fm_avg, neuron_labels, max_indices, n_neurons
```

[ ]:
```
def show_stimuli_2D(model, block, layer_index, images_selected_classes, shifts,␣
↪max_indices, n_images_selected_classes, shift_step = 3, n_max_feature_maps =␣
↪5, plot_activity = False):
    '''
    arg(s):
        layer_indices, interested layers
        images_selected_classes, all the selected images
        shifts = True/False
```

```
    return:
        neuron_output_shifts_avg, (n_images, n_neurons)
    '''


    if shifts is False:
        neuron_output_highest_final, fm_avg_all_layers, neuron_labels,␣
↪max_indices, n_neurons = compute_neuron_output(
            model, block, layer_index, images_selected_classes, max_indices,␣
↪n_max_feature_maps, plot_activity)


    else:
        # generate shifts for each selected image and then stack:
        neuron_output_highest_with_shifts = []
        for i in range(n_images_selected_classes):
            im = images_selected_classes[i]
            im_all_shifts, n_shifts =  apply_all_shifts(im, shift_step)

            ## neuron_output_highest is of shape (n_shifts, n_neurons *␣
↪n_max_feature_maps)
            neuron_output_highest, fm_avg_all_layers, neuron_labels,␣
↪max_indices, n_neurons = compute_neuron_output(
                model, block, layer_index, im_all_shifts, max_indices,␣
↪n_max_feature_maps, plot_activity)


            ## take the average over all shifts of im
            neuron_output_highest_shifts_avg = neuron_output_highest.
↪sum(axis=0) / neuron_output_highest.shape[0]
            neuron_output_highest_shifts_avg = neuron_output_highest_shifts_avg.
↪reshape((1, neuron_output_highest_shifts_avg.shape[0]))


            fm_shifts_avg = fm_avg_all_layers.sum(axis=0) / fm_avg_all_layers.
↪shape[0]
            fm_shifts_avg = fm_shifts_avg.reshape((1,fm_shifts_avg.shape[0]))

            if i == 0:
              neuron_output_highest_with_shifts =␣
↪neuron_output_highest_shifts_avg
              fm_avg_all_layers_with_shifts = fm_shifts_avg
            else:
              neuron_output_highest_with_shifts = np.
↪vstack((neuron_output_highest_with_shifts, neuron_output_highest_shifts_avg))
```

```
            fm_avg_all_layers_with_shifts = np.
↪vstack((fm_avg_all_layers_with_shifts, fm_shifts_avg))

        ## out of for loop!
        ## neuron_output_highest_with_shifts is of shape␣
↪(n_images_selected_classes, n_neurons * n_max_feature_maps)
        neuron_output_highest_with_shifts = np.
↪array(neuron_output_highest_with_shifts)
        neuron_output_highest_final = neuron_output_highest_with_shifts
        fm_avg_all_layers = fm_avg_all_layers_with_shifts

    return neuron_output_highest_final, fm_avg_all_layers, neuron_labels,␣
↪max_indices
```

[ ]:
```
neuron_output_highest_with_shifts, fm_avg, neuron_labels, max_indices =␣
↪show_stimuli_2D(
    model, 'RNN', 0, images_selected_classes, shifts = True,  max_indices =␣
↪None, n_images_selected_classes = num_images, n_max_feature_maps=20,␣
↪plot_activity = False)
```

[ ]:
```
data = np.transpose(neuron_output_highest_with_shifts, (1,0))
from scipy.io import savemat
mdic = {"neuron_output_2D": data}
mdic
savemat("neuron_output_2D_crnn.mat", mdic)
```

[ ]:
```
data.shape
```

5 Tensor factorization:

[ ]:
```
N_crnn_2D = loadmat("neuron_output_2D_crnn.mat")["neuron_output_2D"]
```

[ ]:
```
!pip install git+https://github.com/ahwillia/tensortools
import tensortools as tt
```

[ ]:
```
def get_tensor_factors(N, dim = 3, ranks = [10, 20, 30], reps = 1):
## note that for 2D tensor (ie a matrix), rank + nullity = num_columns => rank␣
↪<= num_columns
  if dim == 3:
    N_filtered = N
    # N_filtered = np.empty(N.shape)
    # for i in range(N.shape[0]):
    #   for j in range(N.shape[1]):
    #     filtered = gaussian_filter(N[i,j,:].reshape((n_vertical_shifts,␣
↪n_vertical_shifts)), sigma=1).reshape((n_vertical_shifts *␣
↪n_vertical_shifts,))
```

```python
  #     N_filtered[i,j,:] = filtered[:]
else:
  N_filtered = N.reshape((N.shape[0], N.shape[1],1))

# Fit ensembles of tensor decompositions:
methods = (
  "cp_als",  # Fits CP Decomposition using Alternating Least Squares (ALS).
)

ensembles = {}
for m in methods:
    ensembles[m] = tt.Ensemble(fit_method=m,  fit_options=dict(tol=1e-5))
    ensembles[m].fit(N_filtered, ranks=ranks, replicates=reps)
    ## replicates: int, number of models to fit at each rank

## plot objective, similarity, factors:
"""
Customized plotting routines for CP decompositions
"""

# Plotting options for the unconstrained and nonnegative models.
plot_options = {
  'cp_als': {
    'line_kw': {
      'color': 'blue',
      'alpha': 1,
      'label': 'cp_als',
    },
    'scatter_kw': {
      'color': 'blue',
      'alpha': 1,
      's': 1,
    },
  },
}

def plot_objective(ensemble, partition='train', ax=None, jitter=0.1,
                  scatter_kw=dict(), line_kw=dict()):
    """Plots objective function as a function of model rank.
    Parameters
    ----------
    ensemble : Ensemble object
        holds optimization results across a range of model ranks
    partition : string, one of: {'train', 'test'}
        specifies whether to plot the objective function on the training
        data or the held-out test set.
    ax : matplotlib axis (optional)
```

```
        axis to plot on (defaults to current axis object)
    jitter : float (optional)
        amount of horizontal jitter added to scatterpoints (default=0.1)
    scatter_kw : dict (optional)
        keyword arguments for styling the scatterpoints
    line_kw : dict (optional)
        keyword arguments for styling the line
    """

    if ax is None:
        ax = plt.gca()

    if partition == 'train':
        pass
    elif partition == 'test':
        raise NotImplementedError('Cross-validation is on the TODO list.')
    else:
        raise ValueError("partition must be 'train' or 'test'.")

    # compile statistics for plotting
    x, obj, min_obj = [], [], []
    for rank in sorted(ensemble.results):
        # reconstruction errors for rank-r models
        o = ensemble.objectives(rank)
        obj.extend(o)
        x.extend(np.full(len(o), rank))
        min_obj.append(min(o))

    print(o)
    print(obj)
    print(x)
    # add horizontal jitter
    ux = np.unique(x)
    x = np.array(x) + (np.random.rand(len(x))-0.5)*jitter

    # make plot
    # customized: plot objectives for all iterations
    ax.scatter(x, obj, **scatter_kw)
    ax.plot(ux, min_obj, **line_kw)
    ax.set_xlabel('model rank')
    ax.set_ylabel('objective')

    return ax


def plot_similarity(ensemble, ax=None, jitter=0.1,
                    scatter_kw=dict(), line_kw=dict()):
```

```python
    """Plots similarity across optimization runs as a function of model rank.
    Parameters
    ----------
    ensemble : Ensemble object
        holds optimization results across a range of model ranks
    ax : matplotlib axis (optional)
        axis to plot on (defaults to current axis object)
    jitter : float (optional)
        amount of horizontal jitter added to scatterpoints (default=0.1)
    scatter_kw : dict (optional)
        keyword arguments for styling the scatterpoints
    line_kw : dict (optional)
        keyword arguments for styling the line
    References
    ----------
    Ulrike von Luxburg (2010). Clustering Stability: An Overview.
    Foundations and Trends in Machine Learning.
    https://arxiv.org/abs/1007.1075
    """

    if ax is None:
        ax = plt.gca()

    # compile statistics for plotting
    x, sim, mean_sim = [], [], []
    for rank in sorted(ensemble.results):
        # reconstruction errors for rank-r models
        s = ensemble.similarities(rank)[1:]
        sim.extend(s)
        x.extend(np.full(len(s), rank))
        mean_sim.append(np.mean(s))

    # add horizontal jitter
    ux = np.unique(x)
    x = np.array(x) + (np.random.rand(len(x))-0.5)*jitter

    # make plot
    # customized: plot similarities for all iterations
    ax.scatter(x, sim, **scatter_kw)
    ax.plot(ux, mean_sim, **line_kw)

    ax.set_xlabel('model rank')
    ax.set_ylabel('model similarity')
    ax.set_ylim([0, 1.1])

    return ax
```

```python
    # Plot similarity and error plots.
    plt.figure()
    for m in methods:
        plot_objective(ensembles[m], **plot_options[m])
    plt.legend()

    # plt.figure()
    # for m in methods:
    #     plot_similarity(ensembles[m], **plot_options[m])
    # plt.legend()

    plt.show()

    return ensembles ## A LIST!
```

[ ]:
```python
def get_embeddings(tensor_factors_best_rank,neuron_labels,n_dim_PCA = 10):
    neuron_factor = tensor_factors_best_rank[0]
    # PCA on tensor factors
    pca = PCA(n_dim_PCA)
    neuron_output_highest_with_shifts_PCA = pca.fit_transform(neuron_factor)
    vectors, lambdas = pca.components_, pca.explained_variance_
    plt.plot(pca.explained_variance_ratio_)
    plt.show()

    # plot embeddings:
    data = neuron_output_highest_with_shifts_PCA
    import plotly.graph_objects as go
    import plotly.express as px

    fig = go.Figure()
    traces = []
    colors_palette = px.colors.qualitative.Dark24
    data = neuron_output_highest_with_shifts_PCA
    print(data.shape)
    for i, label in enumerate(set(neuron_labels)):
        mask = (neuron_labels == label)
        print(mask.shape)
        print(label, sum(mask))
        traces.append(go.Scatter3d(
            x=data[mask,0],
            y=data[mask,1],
            z=data[mask,2],
            mode='markers',
            marker=dict(
                size=4,
                color=colors_palette[int(label)],
                opacity=1,
```

```
                #showscale= True,
            )))

    for trace in traces:
        fig.add_trace(trace)
    fig.update_layout(

      width=700,
      margin=dict(r=20, l=10, b=10, t=10))

    fig.update_layout(margin=dict(l=0, r=0, b=0, t=0),showlegend=True,
                      # scene = dict(xaxis = dict(range=[-1.5,2.5],),
                      #              yaxis = dict(range=[-1.5,1.5],),
                      #              zaxis = dict(range=[-1.5,1.5],),),
                      )
    fig.show()
    return neuron_output_highest_with_shifts_PCA
```

[ ]:

```
def colorFromUnivariateData(Z1, cmap1 = plt.cm.Reds):
    # Rescale values to fit into colormap range (0->255)
    Z1_plot = np.array(255*(Z1-Z1.min())/(Z1.max()-Z1.min()), dtype=np.int)

    Z1_color = cmap1(Z1_plot)

    # Color for each point
    Z_color = np.array(Z1_color[:,0:3])
    return Z_color

## ## https://stackoverflow.com/questions/49871436/
↪scatterplot-with-continuous-bivariate-color-palette-in-python

def colorFromBivariateData(Z1,Z2,cmap1 = plt.cm.Blues, cmap2 = plt.cm.Reds):
    # Rescale values to fit into colormap range (0->255)
    Z1_plot = np.array(255*(Z1-Z1.min())/(Z1.max()-Z1.min()), dtype=np.int)
    Z2_plot = np.array(255*(Z2-Z2.min())/(Z2.max()-Z2.min()), dtype=np.int)

    Z1_color = cmap1(Z1_plot)
    Z2_color = cmap2(Z2_plot)

    # Color for each point
    Z_color = np.sum([Z1_color , Z2_color ], axis=0)/2.0
    Z_color = np.array(Z_color[:,0:3])
    return Z_color

def get_spatial_order_plot(cluster_index,␣
↪neuron_output_highest_with_shifts_PCA,␣
↪neuron_labels,n_max_feature_maps,n_rows,n_cols):
```

```
n_neurons = int(neuron_labels.shape[0]/n_max_feature_maps)
print(n_neurons)
neuron_cluster = neuron_output_highest_with_shifts_PCA[cluster_index *␣
↪n_neurons : (cluster_index + 1) * n_neurons]
neuron_labels_cluster = neuron_labels[cluster_index * n_neurons :␣
↪(cluster_index + 1) * n_neurons]
xs, ys = np.mgrid[0:n_rows,0:n_cols]
xs = xs.reshape((n_neurons,))
ys = ys.reshape((n_neurons,))

import plotly.graph_objects as go
import plotly.express as px

fig = go.Figure()
traces = []
colors_palette = px.colors.qualitative.Dark24
data = neuron_cluster

for i, label in enumerate(set(neuron_labels)):
    mask = (neuron_labels_cluster == label)
    print(label, sum(mask))
    traces.append(go.Scatter3d(
        x=data[mask,0],
        y=data[mask,1],
        z=data[mask,2],
        mode='markers',
        marker=dict(
            size=4,
            color=colors_palette[int(label)],
            opacity=1,
            #showscale= True,
        )))
for trace in traces:
    fig.add_trace(trace)
fig.update_layout(margin=dict(l=0, r=0, b=0, t=0),showlegend=True,)
fig.show()


fig = go.Figure(go.Scatter3d(
  x = data[:,0],
  y = data[:,1],
  z = data[:,2],
  mode='markers',
  marker=dict(
      size=10,
      color=colorFromUnivariateData(xs, cmap1 = plt.cm.Blues),
      opacity=1
```

```
    )
  ))
  fig.show()

  fig = go.Figure(go.Scatter3d(
  x = data[:,0],
  y = data[:,1],
  z = data[:,2],
  mode='markers',
  marker=dict(
      size=10,
      color=colorFromUnivariateData(ys, cmap1 = plt.cm.Reds),
  )
  ))
  fig.show()
```

```
[ ]: def get_spatial_order_plot_2D(cluster_index,␣
     ↪neuron_output_highest_with_shifts_PCA, neuron_labels, n_max_feature_maps=10):
      n_neurons = int(neuron_labels.shape[0]/n_max_feature_maps)
      neuron_cluster = neuron_output_highest_with_shifts_PCA[cluster_index *␣
     ↪n_neurons : (cluster_index + 1) * n_neurons]
      neuron_labels_cluster = neuron_labels[cluster_index * n_neurons :␣
     ↪(cluster_index + 1) * n_neurons]
      feature_map_side = int(np.sqrt(n_neurons))
      xs, ys = np.mgrid[0:feature_map_side,0:feature_map_side]
      xs = xs.reshape((n_neurons,))
      ys = ys.reshape((n_neurons,))

      import plotly.graph_objects as go
      import plotly.express as px

      fig = go.Figure()
      traces = []
      colors_palette = px.colors.qualitative.Dark24
      data = neuron_cluster

      for i, label in enumerate(set(neuron_labels)):
          mask = (neuron_labels_cluster == label)
          print(label, sum(mask))
          traces.append(go.Scatter(
              x=data[mask,0],
              y=data[mask,1],
              mode='markers',
              marker=dict(
                  size=4,
                  color=colors_palette[int(label)],
                  opacity=1,
```

```
                #showscale= True,
            )))
    for trace in traces:
        fig.add_trace(trace)
    fig.update_layout(margin=dict(l=0, r=0, b=0, t=0),showlegend=True,)
    fig.show()

    fig = plt.figure(figsize=(12, 12))
    ax = fig.add_subplot(projection='3d')
    data = neuron_cluster
    ax.scatter(
            data[:,0],
            data[:,1],
            c = colorFromUnivariateData(xs, cmap1 = plt.cm.Blues),
            s = 100,
            alpha= 1
            )
    plt.show()

    fig = plt.figure(figsize=(12, 12))
    ax = fig.add_subplot(projection='3d')
    data = neuron_cluster
    ax.scatter(
            data[:,0],
            data[:,1],
            c = colorFromUnivariateData(ys, cmap1 = plt.cm.Reds),
            s = 100,
            alpha= 1
            )
    plt.show()
```

[ ]:
```
def get_tensor_factors_plot(tensor_factors_best_rank, best_rank,␣
 ↪n_max_feature_maps):
    ## neuron_factor shape: (#neuron, #factors=best_rank)
    neuron_factor = tensor_factors_best_rank[0]
    ## neuron_factor shape: (#shifts, #factors=best_rank)
    time_factor = tensor_factors_best_rank[2]

    # neuron_factor_first = neuron_factor[:,0]
    plt.plot(neuron_factor)
    plt.show()

    # time_factor_first = time_factor[:,0]
    plt.plot(time_factor)
    plt.show()

    n_neurons = neuron_factor.shape[0]
```

```
  feature_map_side = int(np.sqrt(n_neurons/n_max_feature_maps))
  print(feature_map_side)
  for i in range(best_rank):
    neuron_factor_i_th = neuron_factor[:,i]
    vmin = neuron_factor_i_th.min()
    vmax = neuron_factor_i_th.max()
    f_i, axes = plt.subplots(1,n_max_feature_maps,figsize=(10,1))
    for f_i, ax in enumerate(axes):
      feature_map_matrix = neuron_factor_i_th[(f_i * feature_map_side ** 2):
↪((f_i+1) * feature_map_side ** 2)].reshape((feature_map_side,
↪feature_map_side))
      ax.imshow(feature_map_matrix, vmin = vmin, vmax = vmax)
      ax.set(xticks = [], yticks = [])
    plt.show()
```

[ ]:
```
ensembles_2D = get_tensor_factors(N_crnn_2D, dim =2, ranks=[5,10,15])
```

[ ]:
```
rep = 0
ranks = [10]
tensor_factors = []
for rank in ranks:
  tensor_factors.append(ensembles_2D['cp_als'].results[rank][rep].factors)
tensor_factors_best_rank = tensor_factors[0]

get_tensor_factors_plot(tensor_factors_best_rank, best_rank=10,
↪n_max_feature_maps=20)
neuron_output_highest_with_shifts_PCA =
↪get_embeddings(tensor_factors_best_rank, neuron_labels, n_dim_PCA = 10)
# get_spatial_order_plot(1, neuron_output_highest_with_shifts_PCA,
↪neuron_labels)
```

6 Sanity check plot the CNN layer:

[ ]:
```
neuron_output_highest_with_shifts, fm_avg, neuron_labels, max_indices =
↪show_stimuli_2D(
    model, 'CNN', 0, images_selected_classes, shifts = True,  max_indices =
↪None, n_images_selected_classes = num_images, n_max_feature_maps=5,
↪plot_activity = False)
```

[ ]:
```
neuron_labels.shape
```

[ ]:
```
data = np.transpose(neuron_output_highest_with_shifts, (1,0))
from scipy.io import savemat
mdic = {"neuron_output_2D": data}
mdic
```

```
savemat("neuron_output_2D_cnn.mat", mdic)
```

[ ]:
```
data.shape
```

[ ]:
```
data.min()
```

[ ]:
```
N_cnn_2D = loadmat("neuron_output_2D_cnn.mat")["neuron_output_2D"]
```

[ ]:
```
def get_tensor_factors(N, dim = 3, ranks = [10, 20, 30], reps = 1):
## note that for 2D tensor (ie a matrix), rank + nullity = num_columns => rank␣
↪<= num_columns
  if dim == 3:
    N_filtered = N
    # N_filtered = np.empty(N.shape)
    # for i in range(N.shape[0]):
    #   for j in range(N.shape[1]):
    #     filtered = gaussian_filter(N[i,j,:].reshape((n_vertical_shifts,␣
↪n_vertical_shifts)), sigma=1).reshape((n_vertical_shifts *␣
↪n_vertical_shifts,))
    #     N_filtered[i,j,:] = filtered[:]
  else:
    N_filtered = N.reshape((N.shape[0], N.shape[1],1))

  # Fit ensembles of tensor decompositions:
  methods = (
    'ncp_hals',  # fits nonnegative tensor decomposition.
  )

  ensembles = {}
  for m in methods:
      ensembles[m] = tt.Ensemble(fit_method=m,  fit_options=dict(tol=1e-5))
      ensembles[m].fit(N_filtered, ranks=ranks, replicates=reps)
      ## replicates: int, number of models to fit at each rank

  ## plot objective, similarity, factors:
  """
  Customized plotting routines for CP decompositions
  """

  # Plotting options for the unconstrained and nonnegative models.
  plot_options = {
    'ncp_hals': {
      'line_kw': {
        'color': 'blue',
        'alpha': 1,
        'label': 'ncp_hals',
      },
```

```python
      'scatter_kw': {
        'color': 'blue',
        'alpha': 1,
        's': 1,
      },
    },
}

def plot_objective(ensemble, partition='train', ax=None, jitter=0.1,
                   scatter_kw=dict(), line_kw=dict()):
    """Plots objective function as a function of model rank.
    Parameters
    ----------
    ensemble : Ensemble object
        holds optimization results across a range of model ranks
    partition : string, one of: {'train', 'test'}
        specifies whether to plot the objective function on the training
        data or the held-out test set.
    ax : matplotlib axis (optional)
        axis to plot on (defaults to current axis object)
    jitter : float (optional)
        amount of horizontal jitter added to scatterpoints (default=0.1)
    scatter_kw : dict (optional)
        keyword arguments for styling the scatterpoints
    line_kw : dict (optional)
        keyword arguments for styling the line
    """

    if ax is None:
        ax = plt.gca()

    if partition == 'train':
        pass
    elif partition == 'test':
        raise NotImplementedError('Cross-validation is on the TODO list.')
    else:
        raise ValueError("partition must be 'train' or 'test'.")

    # compile statistics for plotting
    x, obj, min_obj = [], [], []
    for rank in sorted(ensemble.results):
        # reconstruction errors for rank-r models
        o = ensemble.objectives(rank)
        obj.extend(o)
        x.extend(np.full(len(o), rank))
        min_obj.append(min(o))
```

```
    print(o)
    print(obj)
    print(x)
    # add horizontal jitter
    ux = np.unique(x)
    x = np.array(x) + (np.random.rand(len(x))-0.5)*jitter

    # make plot
    # customized: plot objectives for all iterations
    ax.scatter(x, obj, **scatter_kw)
    ax.plot(ux, min_obj, **line_kw)
    ax.set_xlabel('model rank')
    ax.set_ylabel('objective')

    return ax


def plot_similarity(ensemble, ax=None, jitter=0.1,
                    scatter_kw=dict(), line_kw=dict()):
    """Plots similarity across optimization runs as a function of model rank.
    Parameters
    ----------
    ensemble : Ensemble object
        holds optimization results across a range of model ranks
    ax : matplotlib axis (optional)
        axis to plot on (defaults to current axis object)
    jitter : float (optional)
        amount of horizontal jitter added to scatterpoints (default=0.1)
    scatter_kw : dict (optional)
        keyword arguments for styling the scatterpoints
    line_kw : dict (optional)
        keyword arguments for styling the line
    References
    ----------
    Ulrike von Luxburg (2010). Clustering Stability: An Overview.
    Foundations and Trends in Machine Learning.
    https://arxiv.org/abs/1007.1075
    """

    if ax is None:
        ax = plt.gca()

    # compile statistics for plotting
    x, sim, mean_sim = [], [], []
    for rank in sorted(ensemble.results):
        # reconstruction errors for rank-r models
        s = ensemble.similarities(rank)[1:]
```

```
            sim.extend(s)
            x.extend(np.full(len(s), rank))
            mean_sim.append(np.mean(s))

        # add horizontal jitter
        ux = np.unique(x)
        x = np.array(x) + (np.random.rand(len(x))-0.5)*jitter

        # make plot
        # customized: plot similarities for all iterations
        ax.scatter(x, sim, **scatter_kw)
        ax.plot(ux, mean_sim, **line_kw)

        ax.set_xlabel('model rank')
        ax.set_ylabel('model similarity')
        ax.set_ylim([0, 1.1])

        return ax

    # Plot similarity and error plots.
    plt.figure()
    for m in methods:
        plot_objective(ensembles[m], **plot_options[m])
    plt.legend()

    # plt.figure()
    # for m in methods:
    #     plot_similarity(ensembles[m], **plot_options[m])
    # plt.legend()

    plt.show()

    return ensembles ## A LIST!
```

[ ]:

```
def get_embeddings(tensor_factors_best_rank,neuron_labels,n_dim_PCA = 10):
    neuron_factor = tensor_factors_best_rank[0]
    # PCA on tensor factors
    pca = PCA(n_dim_PCA)
    neuron_output_highest_with_shifts_PCA = pca.fit_transform(neuron_factor)
    vectors, lambdas = pca.components_, pca.explained_variance_
    plt.plot(pca.explained_variance_ratio_)
    plt.show()

    # plot embeddings:

    import plotly.graph_objects as go
    import plotly.express as px
```

```
fig = go.Figure()
traces = []
colors_palette = px.colors.qualitative.Dark24
data = neuron_output_highest_with_shifts_PCA
print(data.shape)
for i, label in enumerate(set(neuron_labels)):
    mask = (neuron_labels == label)
    print(mask.shape)
    print(label, sum(mask))
    traces.append(go.Scatter3d(
        x=data[mask,0],
        y=data[mask,1],
        z=data[mask,2],
        mode='markers',
        marker=dict(
            size=4,
            color=colors_palette[int(label)],
            opacity=1,
            #showscale= True,
        )))

for trace in traces:
    fig.add_trace(trace)
fig.update_layout(

  width=700,
  margin=dict(r=20, l=10, b=10, t=10))

fig.update_layout(margin=dict(l=0, r=0, b=0, t=0),showlegend=True,
                  # scene = dict(xaxis = dict(range=[-1.5,2.5],),
                  #              yaxis = dict(range=[-1.5,1.5],),
                  #              zaxis = dict(range=[-1.5,1.5],),),
                  )
fig.show()
return neuron_output_highest_with_shifts_PCA
```

[ ]:
```
def get_tensor_factors_plot(tensor_factors_best_rank, best_rank,␣
↪n_max_feature_maps):
  ## neuron_factor shape: (#neuron, #factors=best_rank)
  neuron_factor = tensor_factors_best_rank[0]
  ## neuron_factor shape: (#shifts, #factors=best_rank)
  time_factor = tensor_factors_best_rank[2]

  # neuron_factor_first = neuron_factor[:,0]
  plt.plot(neuron_factor)
  plt.show()
```

```
# time_factor_first = time_factor[:,0]
plt.plot(time_factor)
plt.show()

n_neurons = neuron_factor.shape[0]
feature_map_side = int(np.sqrt(n_neurons/n_max_feature_maps))
print(feature_map_side)
for i in range(best_rank):
  neuron_factor_i_th = neuron_factor[:,i]
  vmin = neuron_factor_i_th.min()
  vmax = neuron_factor_i_th.max()
  f_i, axes = plt.subplots(1,n_max_feature_maps,figsize=(10,1))
  for f_i, ax in enumerate(axes):
    feature_map_matrix = neuron_factor_i_th[(f_i * feature_map_side ** 2):
↪((f_i+1) * feature_map_side ** 2)].reshape((feature_map_side,
↪feature_map_side))
    ax.imshow(feature_map_matrix, vmin = vmin, vmax = vmax)
    ax.set(xticks = [], yticks = [])
  plt.show()
```

[ ]:
```
N_cnn_2D.shape
```

[ ]:
```
ensembles_2D = get_tensor_factors(N_cnn_2D, dim =2, ranks=[3,4,5])
```

[ ]:
```
rep = 0
ranks = [3]
tensor_factors = []
for rank in ranks:
  tensor_factors.append(ensembles_2D['ncp_hals'].results[rank][rep].factors)
tensor_factors_best_rank = tensor_factors[0]

get_tensor_factors_plot(tensor_factors_best_rank, best_rank=3,
↪n_max_feature_maps=5)
neuron_output_highest_with_shifts_PCA =
↪get_embeddings(tensor_factors_best_rank, neuron_labels, n_dim_PCA = 3)
get_spatial_order_plot(1, neuron_output_highest_with_shifts_PCA, neuron_labels,
↪n_max_feature_maps=5)
```

First conv layer:

[ ]:
```
```

7 CNN 3D

[ ]:
```
def show_stimuli_3D(model, block, layer_index, images_selected_classes, shifts,␣
↪max_indices, n_images_selected_classes, shift_step = 3, n_max_feature_maps =␣
↪10, plot_activity = False):
    '''
    arg(s):
        layer_indices, interested layers
        images_selected_classes, all the selected images
        shifts = True/False
    return:
        neuron_output_shifts_avg, (n_images, n_neurons)
    '''



    if shifts is False:
        neuron_output_highest_final, fm_avg_all_layers, neuron_labels,␣
↪max_indices, n_neurons = compute_neuron_output(
            model, block, layer_index, images_selected_classes, max_indices,␣
↪n_max_feature_maps, plot_activity)



    else:
        # generate shifts for each selected image and then stack:
        neuron_output_highest_with_shifts = []
        for i in range(n_images_selected_classes):
            im = images_selected_classes[i]
            im_all_shifts, n_shifts =  apply_all_shifts(im, shift_step)

            ## neuron_output_highest is of shape (n_shifts, n_neurons *␣
↪n_max_feature_maps)
            neuron_output_highest, fm_avg_all_layers, neuron_labels,␣
↪max_indices, n_neurons = compute_neuron_output(
                model, block, layer_index, im_all_shifts, max_indices,␣
↪n_max_feature_maps, plot_activity)

            ## instead of taking average, we create a dimension for all shifts␣
↪(analogous to the time dimension)
            neuron_output_highest_with_shifts.append(neuron_output_highest)

        ## out of for loop!
        ## neuron_output_highest_with_shifts is of shape␣
↪(n_images_selected_classes, n_neurons * n_max_feature_maps)
        neuron_output_highest_with_shifts = np.
↪array(neuron_output_highest_with_shifts)
        neuron_output_highest_final = neuron_output_highest_with_shifts
```

```
    return neuron_output_highest_final, fm_avg_all_layers, neuron_labels,␣
 ↪max_indices
```

[ ]:
```
neuron_output_highest_with_shifts, fm_avg, neuron_labels, max_indices =␣
 ↪show_stimuli_3D(
    model, 'CNN', 0, images_selected_classes, shifts = True, max_indices =␣
 ↪None, n_images_selected_classes = num_images, shift_step = 3,␣
 ↪n_max_feature_maps=5, plot_activity = False)
```

[ ]:
```
neuron_output_highest_with_shifts.shape
```

[ ]:
```
data = np.transpose(neuron_output_highest_with_shifts, (2,0,1))
data.shape
```

[ ]:
```
data.min()
```

[ ]:
```
from scipy.io import savemat
mdic = {"neuron_output_3D": data}
mdic
savemat("neuron_output_3D_crnn.mat", mdic)
```

[ ]:
```
data.shape
```

[ ]:
```
N_crnn_3D = loadmat("neuron_output_3D_crnn.mat")["neuron_output_3D"]
```

[ ]:
```
ensembles_3D = get_tensor_factors(N_crnn_3D, dim =3, ranks=[150])
```

[ ]:
```
rep = 0
ranks = [150]
tensor_factors = []
for rank in ranks:
  tensor_factors.append(ensembles_3D['ncp_hals'].results[rank][rep].factors)
tensor_factors_best_rank = tensor_factors[0]

get_tensor_factors_plot(tensor_factors_best_rank, best_rank=150,␣
 ↪n_max_feature_maps=5)
neuron_output_highest_with_shifts_PCA =␣
 ↪get_embeddings(tensor_factors_best_rank, neuron_labels, n_dim_PCA = 10)
get_spatial_order_plot(1, neuron_output_highest_with_shifts_PCA, neuron_labels,␣
 ↪n_max_feature_maps=5, n_rows = 16, n_cols = 50)
```

[ ]:
```
get_spatial_order_plot(3, neuron_output_highest_with_shifts_PCA, neuron_labels,␣
 ↪n_max_feature_maps=5, n_rows = 16, n_cols = 50)
```

[ ]:
```

```