VisionTransformer_Visualizing_Embeddings_TCA

April 4, 2022

1 Building Neuron Tensors for ViT

Resources that have been useful: https://huggingface.co/docs/transformers/model_doc/vit https://huggingface.co/docs/transformers/v4.17.0/en/main_classes/output

Resources on Transformers / ViT: * video tutorials: * https://www.youtube.com/watch?v=aButdUV0dxI&list=PLvOO0btloRntpSWSxFbwPIjIum3Ub4GSC * https://www.youtube.com/watch?v=TrdevFK_am4

- connection to convolution:
  - https://epfml.github.io/attention-cnn/
  - https://arxiv.org/pdf/1911.03584.pdf
  - https://openreview.net/forum?id=Gl8FHfMVTZu
  - https://arxiv.org/pdf/2111.01353.pdf
- tutorial:
  - https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial15/Vision_Transforme
  - https://jalammar.github.io/illustrated-transformer/
  - https://lilianweng.github.io/posts/2018-06-24-attention/
- explainability:
  - https://jacobgil.github.io/deeplearning/vision-transformer-explainability#how-do-the-attention-activations-look-like-for-the-class-token-throughout-the-network-
  - https://colab.research.google.com/github/hila-chefer/Transformer-Explainability/blob/main/Transformer_explainability.ipynb#scrollTo=UtHosD9lCgAA (https://github.com/hila-chefer/Transformer-Explainability)
  - visualize attention map: https://github.com/jeonsworld/ViT-pytorch/blob/main/visualize_attention_map.ipynb
- visualizing attention maps!!
  - https://viso.ai/deep-learning/vision-transformer-vit/ and https://arxiv.org/pdf/2106.01548.pdf
  - dino visualization:https://github.com/facebookresearch/dino
  - extremely useful visualization: https://colab.research.google.com/github/hirotomusiker/schwert_colab_
  - also extremely useful in matching the dimensions: https://programmer.group/613ada5f581ff.html

Figure 1. Vision Transformer inference pipeline.
1. Split Image into Patches
The input image is split into 14 x 14 vectors with dimension of 768 by Conv2d (k=16x16) with stride=(16, 16). 2. Add Position Embeddings
Learnable position embedding vectors are added to the patch embedding vectors and fed to the

transformer encoder. 3. Transformer Encoder
The embedding vectors are encoded by the transformer encoder. The dimension of input and output vectors are the same. Details of the encoder are depicted in Fig. 2. 4. MLP (Classification) Head
The 0th output from the encoder is fed to the MLP head for classification to output the final classification results.

Transformer Encoder

Figure 2. Detailed schematic of Transformer Encoder. - N (=197) embedded vectors are fed to the L (=12) series encoders. - The vectors are divided into query, key and value after expanded by an fc layer. - q, k and v are further divided into H (=12) and fed to the parallel attention heads. - Outputs from attention heads are concatenated to form the vectors whose shape is the same as the encoder input. - The vectors go through an fc, a layer norm and an MLP block that has two fc layers.

The Vision Transformer employs the Transformer Encoder that was proposed in the attention is all you need paper.

Implementation Reference:

- tensorflow implementation
- pytorch implementation (timm)

[ ]:
```
!pip install timm
```

[ ]:
```
import os
import matplotlib.pyplot as plt
import numpy as np
import PIL

import torch
import torch.nn.functional as F
import torchvision
import torchvision.transforms as T

from timm import create_model
```

[ ]:
```
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
```

```python
import datetime
from scipy.io import savemat
from scipy.io import loadmat
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
import time
```

[ ]:
```python
import numpy as np
import torch
import random
import os

default_seed = 4142
def seed_everything(seed = 1234):
  random.seed(seed)
  tseed = random.randint(1,1E6)
  tcseed = random.randint(1,1E6)
  npseed = random.randint(1,1E6)
  ospyseed = random.randint(1,1E6)
  torch.manual_seed(tseed)
  torch.cuda.manual_seed_all(tcseed)
  np.random.seed(npseed)
  os.environ['PYTHONHASHSEED'] = str(ospyseed)

seed_everything(default_seed)
```

[ ]:
```python
!pip install git+https://github.com/ahwillia/tensortools
import tensortools as tt
from scipy.ndimage import gaussian_filter
```

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

[ ]:
```python
# Define transforms for test
IMG_SIZE = (224, 224)
NORMALIZE_MEAN = (0.5, 0.5, 0.5)
NORMALIZE_STD = (0.5, 0.5, 0.5)
transforms = [
              T.Resize(IMG_SIZE),
              T.ToTensor(),
              T.Normalize(NORMALIZE_MEAN, NORMALIZE_STD),
              ]

transforms = T.Compose(transforms)
```

2 Load Vision Transformer model and get outputs from the 12 layers:

[ ]:
```
pip install -q git+https://github.com/huggingface/transformers.git
```

[ ]:
```
model_id = 'google/vit-base-patch16-224'
```

[ ]:
```
from transformers import ViTForImageClassification
import torch

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model = ViTForImageClassification.from_pretrained(model_id,␣
↪output_hidden_states=True, output_attentions=True)
model.eval()
model.to(device)
```

[ ]:
```
!pip install huggingface
```

[ ]:
```
!pip install datasets
```

[ ]:
```
from transformers import ViTFeatureExtractor, ViTModel
import torch
from datasets import load_dataset
dataset = load_dataset("huggingface/cats-image")
image = dataset["test"]["image"][0]

# with torch.no_grad():
#     outputs = model(**inputs)
```

[ ]:
```
plt.imshow(image)
```

[ ]:
```
from PIL import Image
import requests

url = 'http://images.cocodataset.org/val2017/000000039769.jpg'
im = Image.open(requests.get(url, stream=True).raw)
im
```

[ ]:
```
## predict the class of the cat image just now
feature_extractor = ViTFeatureExtractor.from_pretrained(model_id)
encoding = feature_extractor(images=image, return_tensors="pt")
pixel_values = encoding['pixel_values'].to(device)

outputs = model(pixel_values)
logits = outputs.logits
```

```
logits.shape

prediction = logits.argmax(-1)
print("Predicted class:", model.config.id2label[prediction.item()])
```

[ ]:
```
pixel_values.shape
```

3 Get attentions and hidden_states:

hidden_states (tuple(torch.FloatTensor), optional, returned when output_hidden_states=True is passed or when config.output_hidden_states=True) — Tuple of torch.FloatTensor (one for the output of the embeddings + one for the output of each layer) of shape (batch_size, sequence_length, hidden_size).

Hidden-states of the model at the output of each layer plus the initial embedding outputs.

attentions (tuple(torch.FloatTensor), optional, returned when output_attentions=True is passed or when config.output_attentions=True) — Tuple of torch.FloatTensor (one for each layer) of shape (batch_size, num_heads, sequence_length, sequence_length).

Attentions weights after the attention softmax, used to compute the weighted average in the self-attention heads.

[ ]:
```
type(image)
```

[ ]:
```
pixel_values
```

[ ]:
```
hidden_states = outputs.hidden_states
attentions = outputs.attentions
```

[ ]:
```
len(attentions)
```

[ ]:
```
len(hidden_states)
## output of the embeddings + 12 outputs for the 12 layers
```

[ ]:
```
print(attentions[0].shape)
attentions[0]
```

[ ]:
```
# the output of the embeddings of shape (batch_size, sequence_length,␣
↪hidden_size)
print(hidden_states[0].shape)
hidden_states[0]
```

[ ]:
```
# the output of the first layer, of shape (batch_size, sequence_length,␣
↪hidden_size)
print(hidden_states[1].shape)
hidden_states[1]
```

[ ]:
```python
# Visualize attention matrix
# There are 12 encoder layers, each layer has 12 attention heads
# i is layer index, j is the index for the attention head
for i in range(12):
  attention_heads_layer_i = attentions[i]
  n_attention_heads = attention_heads_layer_i.shape[1]
  print(n_attention_heads)
  for j in range(n_attention_heads):
    attention_head_j = attention_heads_layer_i[0,j,:,:]
    plt.imshow(attention_head_j.cpu().data)
    # ax.set(xticks = [], yticks = [])
    plt.show()
```

[ ]:
```python
# Visualize attention map
fig = plt.figure(figsize=(16, 8))
fig.suptitle("Visualization of Attention Heatmap", fontsize=24)
fig.add_axes()

i = 5
attention_heads_layer_i = attentions[i]
for j in range(12):  # visualize the 100th rows of attention matrices in the␣
 ↪0-7th heads
    attention_head_j = attention_heads_layer_i[0,j,:,:] # (197 by 197)
    attention_head_j = attention_head_j.detach().cpu().numpy()
    attn_heatmap = attention_head_j[120, 1:].reshape((14, 14))
    ax = fig.add_subplot(4, 4, j+1)
    ax.imshow(attn_heatmap)
```

[ ]:
```python
activation = {}
def get_activation(name):
    def hook(model, input, output):
      activation[name] = output.detach()
      handle.remove()
    return hook

# register the forward hook
handle = model.vit.encoder.layer[0].output.
 ↪register_forward_hook(get_activation('encoder_queried_layer'))

# pass some data through the model
output = model(pixel_values)

activation['encoder_queried_layer']
print(activation['encoder_queried_layer'].shape)

## for intermediate layer in the first block, out_features=768
```

```
## image size is 224x224 -> (224*224)/(16*16) patches + 1 positional embedding␣
↪= 197
## shape (n_images = 1, n_patches + 1 pos embedding = 197, n_out_features)
activation['encoder_queried_layer']
handle.remove()
```

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

[ ]:
```
n_neurons = 197*768
n_neurons
```

[ ]:
```
hidden_states[1].shape
```

[ ]:
```
plt.matshow(hidden_states[0].squeeze(dim=0).cpu().data)
```

[ ]:
```
x = hidden_states[0].squeeze(dim=0).cpu().data[1:,391]
```

[ ]:
```
plt.imshow(x.reshape(14,14))
```

[ ]:
```
plt.matshow(hidden_states[1].squeeze(dim=0).cpu().data)
```

[ ]:
```
x = hidden_states[1].squeeze(dim=0).cpu().data[1:,391]
plt.imshow(x.reshape(14,14))
```

[ ]:
```
plt.matshow(hidden_states[11].squeeze(dim=0).cpu().data)
```

[ ]:
```
x = hidden_states[11].squeeze(dim=0).cpu().data[1:,391]
plt.imshow(x.reshape(14,14))
```

4 Get images:

[ ]:
```
def imshow(img):
    img = img.cpu().data
    img = img / (img.max()-img.min())    # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
```

[ ]:
```
!python3 -m pip install --upgrade pip
!python3 -m pip install --upgrade Pillow
from PIL import Image, ImageTk
img_array_starfish = np.load('imgnet_starfish.npy')
img_array_strawberry = np.load('imgnet_strawberry.npy')
img_array_husky = np.load('imgnet_husky.npy')
img_array_guitar = np.load('imgnet_guitar.npy')
```

```python
def get_images_selected_classes(num_images, n_classes):
  images_selected_classes = []
  MAX_SIZE = 224
  for i in range(int(num_images/n_classes)):
    im = Image.fromarray(img_array_starfish[i,:,:,:].astype(np.uint8))
    im.thumbnail((MAX_SIZE, MAX_SIZE), Image.ANTIALIAS)
    images_selected_classes.append(np.array(im))
  for i in range(int(num_images/n_classes)):
    im = Image.fromarray(img_array_strawberry[i,:,:,:].astype(np.uint8))
    im.thumbnail((MAX_SIZE, MAX_SIZE), Image.ANTIALIAS)
    images_selected_classes.append(np.array(im))
  for i in range(int(num_images/n_classes)):
    im = Image.fromarray(img_array_husky[i,:,:,:].astype(np.uint8))
    im.thumbnail((MAX_SIZE, MAX_SIZE), Image.ANTIALIAS)
    images_selected_classes.append(np.array(im))
  for i in range(int(num_images/n_classes)):
    im = Image.fromarray(img_array_guitar[i,:,:,:].astype(np.uint8))
    im.thumbnail((MAX_SIZE, MAX_SIZE), Image.ANTIALIAS)
    images_selected_classes.append(np.array(im))
  ##  (#images, #nrow, #ncol, #channels)
  images_selected_classes = np.array(images_selected_classes)

  return images_selected_classes

num_images = 20
n_classes = 4
images_selected_classes = get_images_selected_classes(num_images, n_classes)

images_selected_classes = images_selected_classes.transpose((0,3,1,2))
# images_selected_classes = torch.cuda.FloatTensor(images_selected_classes)

# # Define transforms for test
# NORMALIZE_MEAN = (0.5, 0.5, 0.5)
# NORMALIZE_STD = (0.5, 0.5, 0.5)
# transforms = [
#               T.Normalize(NORMALIZE_MEAN, NORMALIZE_STD),
#               ]
# transforms = T.Compose(transforms)
# images_selected_classes = transforms(images_selected_classes)

# for i in range(num_images):
#   imshow(images_selected_classes[i])
```

5 Create 3D neuron 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 (3, 224, 224)

         ## 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))
         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.vstack(im_all_shifts)
         im_all_shifts = torch.cuda.FloatTensor(im_all_shifts)
         return im_all_shifts, n_shifts
```

Note: output size = [(W−K+2P)/S]+1 = (128-3)/3+1 = ~43

e.g. conv_layer W is the input volume - 128 K is the Kernel size - 3 P is the padding - 0 S is the stride - 3

edge_neuron = floor(filter_size /stride) = floor(3/3) = 1

```
[ ]: def compute_neuron_output(model, 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)
    layer = model.vit.encoder.layer[layer_index]

    neuron_output = []
    # go through all the batches in the dataset
    for i in range(n_shifts):
      # forward pass -- getting the outputs
      im = 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.vit.encoder.layer[layer_index].output.
↪register_forward_hook(get_activation('encoder_queried_layer'))
      # pass some data through the model
      output = model(im.unsqueeze(dim=0))

      ## shape (n_shifts, n_patches + 1 cls token = 197, n_out_features)
      layer_out = activation['encoder_queried_layer']
      handle.remove()
      relu = torch.nn.ReLU()
      layer_out = relu(layer_out)
      layer_out = layer_out.cpu().data
      layer_out = layer_out[:,1:,:]
      neuron_output.append(layer_out)

      # out = model(im)
      # # collect the activations in the correct list
```

```
        # # +1 is because hidden_states[0] stores the conv2d embeddings before␣
↪the encoder layers
        # layer_out = hidden_states[layer_index + 1].cpu().data
        # print(layer_out.shape)
        # ## force the entries to be nonnegative
        # relu = torch.nn.ReLU()
        # layer_out = relu(layer_out)
        # ## layer_out is of shape (1,197,768) 197=1 cls + 14*14 neurons
        # layer_out = layer_out[:,1:,:]
        # neuron_output.append(layer_out)
    neuron_output = np.vstack(neuron_output)
    print("neuron_output shape: ")
    print(neuron_output.shape)

    ## ------ IF NEED TO REMOVE THE NEURONS AT THE EDGES ------
    # filter_size = 3
    # shift_step = 3
    # edge_neuron = math.floor(filter_size / shift_step)
    # edge_neuron = int(0.1 * n_row)
    # neuron_output = neuron_output[:, edge_neuron:(n_row - edge_neuron),␣
↪edge_neuron:(n_col - edge_neuron),:]
    # n_shifts, n_row, n_col, n_feature_maps = neuron_output.shape[:]
    # neuron_output = neuron_output.reshape((n_shifts, n_row * n_col,␣
↪n_feature_maps))
    # ## number of neurons for each feature map is nrow * ncol
    # n_neurons = n_row * n_col
    ## ------ END ------


    ## ------- IF NEED TO PLOT THE EDGE NEURONS ------
    # neuron_index = np.empty((n_row,n_col),dtype=int)
    # index = 0
    # for i in range(n_row):
    #     for j in range(n_col):
    #         neuron_index[i, j] = index
    #         index += 1
    # ## obtain the index of the neurons at the edges
    # neuron_edge_index = np.hstack((neuron_index[[0,1,n_row-2,n_row-1],:].
↪reshape((4*32,1)), neuron_index[:,[0,1,n_col-2,n_col-1]].reshape((4*32,1))))
    # neuron_edge_index = neuron_edge_index.reshape((256,1))
    # ## re-label the neurons at the edge with a different color
    # neuron_labels = []
    # for i in range(10):
    #     neuron_labels = np.hstack((neuron_labels, [i] * n_neurons))
    #     neuron_labels = np.array(neuron_labels)
    #     neuron_labels[n_neurons * i + neuron_edge_index-1] = 15
```

```
    # neuron_output = neuron_output.reshape((n_shifts, n_neurons,␣
↪n_feature_maps))
    ## ------- END ------

    n_shifts, n_neurons, n_features =  neuron_output.shape
    ## transpose to organize by feature maps
    ## the shape of neuron_output_by_fm is (n_shifts, n_feature_maps, #neurons)
    neuron_output_by_fm = neuron_output.transpose(0, 2, 1)

    ## 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))
    feature_map_side = int(np.sqrt(n_neurons))

    for i in range(n_shifts):
        if max_indices is None:
            max_indices = np.argsort(-1*fm_avg[i], axis = 0)[:
↪n_max_feature_maps]
        temp = neuron_output_by_fm[i, max_indices, :]
        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 = temp[f_i, :].max()
          if normalizing_constant == 0:
            neuron_output_highest[i, (f_i*feature_map_side**2):
↪((f_i+1)*feature_map_side**2)] = 0
          else:
```

```
            neuron_output_highest[i, (f_i*feature_map_side**2):
↪((f_i+1)*feature_map_side**2)] /= 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_3D(model, layer_index, images_selected_classes, shifts,␣
↪max_indices, n_images_selected_classes = 20, shift_step = 11,␣
↪n_max_feature_maps = 500, 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_without_shifts = []
        for i in range(n_images_selected_classes):
            im = np.expand_dims(images_selected_classes[i], axis=0)
            print(im)
            ## neuron_output_highest is of shape (n_shifts = 1, n_neurons *␣
↪n_max_feature_maps)
```

```
            neuron_output_highest, fm_avg, neuron_labels, max_indices,␣
↪n_neurons = compute_neuron_output(
                model, layer_index, np.expand_dims(im, axis=0), max_indices,␣
↪n_max_feature_maps, plot_activity)
            neuron_output_highest_without_shifts.append(neuron_output_highest)
        neuron_output_highest_without_shifts = np.
↪array(neuron_output_highest_without_shifts)
        neuron_output_highest_final = neuron_output_highest_without_shifts
    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, neuron_labels, max_indices,␣
↪n_neurons = compute_neuron_output(
                model, 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_shifts, 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, neuron_labels, max_indices
```

[ ]:
```
neuron_output_highest_with_shifts, fm_avg, neuron_labels, max_indices =␣
↪show_stimuli_3D(
    model, 0, images_selected_classes, shifts = True,  max_indices = None,␣
↪n_images_selected_classes = num_images, n_max_feature_maps=20, plot_activity␣
↪= True)
```

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

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

```
[ ]: data = np.transpose(neuron_output_highest_with_shifts, (2, 0, 1))
from scipy.io import savemat
mdic = {"neuron_output_3D": data}
mdic
savemat("neuron_output_3D_vit_layer1_shifts.mat", mdic)
```

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

```
[ ]: fm_avg.shape
# for last image: shifts by n_fm
```

```
[ ]: for f_i in range(768):
  plt.plot(fm_avg[:,f_i]/max(fm_avg[:,f_i]))
plt.show()
```

```
[ ]: plt.plot(fm_avg[:,0]/max(fm_avg[:,0]))
```

6 Create 2D neuron tensor:

```
[ ]: def show_stimuli_2D(model, layer_index, images_selected_classes, shifts,␣
 ↪max_indices, n_images_selected_classes, shift_step = 11, 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, 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)
            # im_all_shifts = torch.Tensor(im_all_shifts)
```

```
            ## 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, 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_all_layers_with_shifts,␣
↪neuron_labels, max_indices = show_stimuli_2D(
    model, 11, images_selected_classes, shifts = True,  max_indices = None,␣
↪n_images_selected_classes = num_images, n_max_feature_maps=20, plot_activity␣
↪= False)
```

```
[ ]: neuron_output_highest_with_shifts.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_vit_layer12_shifts.mat", mdic)
```

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

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

7 Streamline tensor factorization:

```
[ ]: from scipy.io import loadmat
     # N_vit_3D_layer1 = loadmat("neuron_output_3D_vit_layer1.
      ↪mat")["neuron_output_3D"]
     # N_vit_3D_layer1_20fm = loadmat("neuron_output_3D_vit_layer1_20fm.
      ↪mat")["neuron_output_3D"]
     N_vit_3D_layer1_shifts = loadmat("neuron_output_3D_vit_layer1_shifts.
      ↪mat")["neuron_output_3D"]
     N_vit_2D_layer1_shifts = loadmat("neuron_output_2D_vit_layer1_shifts.
      ↪mat")["neuron_output_2D"]
     N_vit_2D_layer12_shifts = loadmat("neuron_output_2D_vit_layer1_shifts.
      ↪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.
       )
```

```python
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',
    },
    '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_embeddings_2D(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)
    neuron_output_highest_with_shifts_PCA.shape
    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.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(
```

```
        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 =␣
↪5):
  n_neurons = int(neuron_labels.shape[0]/n_max_feature_maps)
  feature_map_side = int(np.sqrt(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]
```

```python
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.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 = plt.figure(figsize=(12, 12))
ax = fig.add_subplot(projection='3d')
data = neuron_cluster
ax.scatter(
        data[:,0],
        data[:,1],
        data[:,2],
        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],
            data[:,2],
            c = colorFromUnivariateData(ys, cmap1 = plt.cm.Reds),
            s = 100,
            alpha= 1
            )
    plt.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()
```

##3D:

```
[ ]: N_vit_3D_layer1_shifts.shape
```

```
[ ]: im_ind = 0
     n_max_feature_maps = 20
     feature_map_side = 14
     # f_i, axes = plt.subplots(1,n_max_feature_maps,figsize=(10,1))
     neuron_matrix = N_vit_3D_layer1_shifts[:,im_ind,:]
```

```
[ ]: for f_i in range(n_max_feature_maps):
       neuron_matrix_fi =  neuron_matrix[(f_i * feature_map_side ** 2): ((f_i+1) *
      ↪feature_map_side ** 2)]
       N_frames = 484
       feature_map_side = 14
       for t in range(N_frames):
         neuron_matrix_fi_t = neuron_matrix_fi[:, t].reshape((feature_map_side,
      ↪feature_map_side))
         vmin = neuron_matrix_fi_t.min()
         vmax = neuron_matrix_fi_t.max()
         plt.imshow(neuron_matrix_fi_t, vmin = vmin, vmax = vmax)
         # plt.set(xticks = [], yticks = [])
         plt.show()
```

```
[ ]: ensembles_3D = get_tensor_factors(N_vit_3D_layer1_shifts, dim =3,
      ↪ranks=[20,25,30])
```

```
[ ]: rep = 0
     ranks = [25]
     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=25,
      ↪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,n_max_feature_maps=20)
```

7.1 2D:

[ ]:
```
N_vit_2D_layer1_shifts.shape
```

[ ]:
```
3920/20
```

[ ]:
```
for f_i in range(5):
  plt.matshow(N_vit_2D_layer1_shifts[196*f_i : 196*(f_i + 1), :].
↪reshape(196,20).T)
```

[ ]:
```
ensembles_2D = get_tensor_factors(N_vit_2D_layer1_shifts, dim =2,␣
↪ranks=[1,2,3,4,5])
```

[ ]:
```
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=20)
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=20)
```

[ ]:
```
ensembles_2D = get_tensor_factors(N_vit_2D_layer12_shifts, dim =2,␣
↪ranks=[1,2,3,4,5])
```

[ ]:
```
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=20)
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=20)
```

```
[ ]:
```