Latest_Training_AlexNet_Visualizing_Embeddings_TCA

April 4, 2022

1 Overview:

- AlexNet Training
- Saved models + saved tensors
- With tensor tools integration
- Streamlined workflow
- 3D neuron tensor
- 2D neuron tensor

References:

https://deeplearning.neuromatch.io/projects/Neuroscience/blurry_vision.html?highlight=alexnet

https://www.kaggle.com/asilvaigor/learning-alexnet

https://www.kaggle.com/drvaibhavkumar/alexnet-in-pytorch-cifar10-clas-83-test-accuracy

https://discuss.pytorch.org/t/how-to-extract-features-of-an-image-from-a-trained-model/119/3?u=klory

https://distill.pub/2017/feature-visualization/

Note that in pytorch linear laryers are initialized in this way:

```
stdv = 1. / math.sqrt(self.weight.size(1))
self.weight.data.uniform_(-stdv, stdv)
if self.bias is not None:
    self.bias.data.uniform_(-stdv, stdv)
```

[ ]:
```
import tensorflow as tf
import numpy as np
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 matplotlib.pyplot as plt
import math
```

```
import time
```

[ ]:
```
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
# printout versions
print(f"Tensor Flow Version: {tf.__version__}")
print(f"numpy Version: {np.version.version}")
```

[ ]:
```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
from torchvision.datasets import CIFAR10
from torch.utils.data import Subset
import torch.optim as optim

print(torch.__version__)
print(torchvision.__version__)
```

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

2 Set seed for reporducibility:

[ ]:
```
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)
```

3 Load CIFAR-10 data:

[ ]:
```
transform = transforms.Compose([
    transforms.Resize(128),
    # transforms.Resize(256),
    # transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
```

[ ]:
```
train_data = CIFAR10(root='./data', train=True, download=True,␣
 ↪transform=transform)
trainloader = torch.utils.data.DataLoader(train_data, batch_size=4,␣
 ↪shuffle=True, num_workers=2)

test_data = CIFAR10(root='./data', train=False, download=True,␣
 ↪transform=transform)
testloader = torch.utils.data.DataLoader(test_data, batch_size=4,␣
 ↪shuffle=False, num_workers=2)

classes = ('Airplane', 'Car', 'Bird', 'Cat', 'Deer', 'Dog', 'Frog', 'Horse',␣
 ↪'Ship', 'Truck')
```

[ ]:
```
N_train = 50000
N_test = 10000
img_shape = (3, 32, 32) #use a tuple in the fomat: (n_channels, height, width)
n_classes = 10
print(N_train,N_test,img_shape,n_classes)
```

[ ]:
```
def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

# get some random training images
dataiter = iter(trainloader)
images, labels = dataiter.next()

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join('%5s' % classes[labels[j]] for j in range(4)))
```

4 Define AlexNet Model:

[ ]:
```
# Define AlexNet with different modules representing different brain areas
class AlexNet(nn.Module):

  def __init__(self, num_classes=1000, downscale=1):
    """
    Args:
      num_classes: int
      downscale: int
    """
    super(AlexNet, self).__init__()

    ##torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1,␣
↪padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros',␣
↪device=None, dtype=None
    self.conv_layer1 = nn.Sequential(
        nn.Conv2d(3, 64//downscale, kernel_size=11, stride=4, padding=2),
        nn.ReLU(inplace=True),
        nn.BatchNorm2d(64//downscale),
        nn.MaxPool2d(kernel_size=3, stride=2),
    )
    self.conv_layer2 = nn.Sequential(
        nn.Conv2d(64//downscale, 192//downscale, kernel_size=5, padding=2),
        nn.ReLU(inplace=True),
        nn.BatchNorm2d(192//downscale),
        nn.MaxPool2d(kernel_size=3, stride=2),
    )
    self.conv_layer3 = nn.Sequential(
        nn.Conv2d(192//downscale, 384//downscale, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.BatchNorm2d(384//downscale),
    )
    self.conv_layer4 = nn.Sequential(
        nn.Conv2d(384//downscale, 256//downscale, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.BatchNorm2d(256//downscale),
    )
    self.conv_layer5 = nn.Sequential(
        nn.Conv2d(256//downscale, 256//downscale, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.BatchNorm2d(256//downscale),
        nn.MaxPool2d(kernel_size=3, stride=2),
    )
    self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
    self.it = nn.Sequential(
        nn.Dropout(),
```

```
            nn.Linear(256//downscale * 6 * 6, 4096//downscale),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096//downscale, 4096//downscale),
            nn.ReLU(inplace=True)
        )
        self.classifier = nn.Linear(4096//downscale, num_classes)

    def forward(self, x):
        """
        Args:
          x: torch.Tensor
        Returns:
          x: torch.Tensor
        """
        x = self.conv_layer1(x)
        x = self.conv_layer2(x)
        x = self.conv_layer3(x)
        x = self.conv_layer4(x)
        x = self.conv_layer5(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.it(x)
        x = self.classifier(x)
        return x
```

[ ]:
```
AlexNet_Model = AlexNet(num_classes=10, downscale=2)
AlexNet_Model.eval()
```

[ ]:
```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(AlexNet_Model.parameters(), lr=0.001, momentum=0.9)
```

[ ]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

[ ]:
```
print(device)
```

5 Train and save model:

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

[ ]:
```
AlexNet_Model_epoch10 = AlexNet(num_classes=10, downscale=2)
AlexNet_Model_epoch10.load_state_dict(torch.load('/content/drive/My Drive/
 ↪Embeddings/code/AlexNet_Model_epoch10.pt')['state_dict'])
```

```
optimizer.load_state_dict(torch.load('/content/drive/My Drive/Embeddings/code/
↪AlexNet_Model_epoch10.pt')['optimizer'])
AlexNet_Model_epoch10.to(device)
AlexNet_Model = AlexNet_Model_epoch10
AlexNet_Model_epoch7 = AlexNet(num_classes=10, downscale=2)
AlexNet_Model_epoch7.load_state_dict(torch.load('/content/drive/My Drive/
↪Embeddings/code/AlexNet_Model_epoch7.pt')['state_dict'])
AlexNet_Model_epoch7.to(device)
AlexNet_Model_epoch5 = AlexNet(num_classes=10, downscale=2)
AlexNet_Model_epoch5.load_state_dict(torch.load('/content/drive/My Drive/
↪Embeddings/code/AlexNet_Model_epoch5.pt')['state_dict'])
AlexNet_Model_epoch5.to(device)
AlexNet_Model_epoch0 = AlexNet(num_classes=10, downscale=2)
AlexNet_Model_epoch0.load_state_dict(torch.load('/content/drive/My Drive/
↪Embeddings/code/AlexNet_Model_epoch0.pt')['state_dict'])
AlexNet_Model_epoch0.to(device)
AlexNet_Model_epoch3 = AlexNet(num_classes=10, downscale=2)
AlexNet_Model_epoch3.load_state_dict(torch.load('/content/drive/My Drive/
↪Embeddings/code/AlexNet_Model_epoch3.pt')['state_dict'])
AlexNet_Model_epoch3.to(device)
AlexNet_Model_epoch1 = AlexNet(num_classes=10, downscale=2)
AlexNet_Model_epoch1.load_state_dict(torch.load('/content/drive/My Drive/
↪Embeddings/code/AlexNet_Model_epoch1.pt')['state_dict'])
AlexNet_Model_epoch1.to(device)
AlexNet_Model_epoch2 = AlexNet(num_classes=10, downscale=2)
AlexNet_Model_epoch2.load_state_dict(torch.load('/content/drive/My Drive/
↪Embeddings/code/AlexNet_Model_epoch2.pt')['state_dict'])
AlexNet_Model_epoch2.to(device)
AlexNet_Model_epoch4 = AlexNet(num_classes=10, downscale=2)
AlexNet_Model_epoch4.load_state_dict(torch.load('/content/drive/My Drive/
↪Embeddings/code/AlexNet_Model_epoch4.pt')['state_dict'])
AlexNet_Model_epoch4.to(device)
AlexNet_Model_epoch8 = AlexNet(num_classes=10, downscale=2)
AlexNet_Model_epoch8.load_state_dict(torch.load('/content/drive/My Drive/
↪Embeddings/code/AlexNet_Model_epoch8.pt')['state_dict'])
AlexNet_Model_epoch8.to(device)
AlexNet_Model_epoch15 = AlexNet(num_classes=10, downscale=2)
AlexNet_Model_epoch15.load_state_dict(torch.load('/content/drive/My Drive/
↪Embeddings/code/AlexNet_Model_epoch15.pt')['state_dict'])
AlexNet_Model_epoch15.to(device)
```

[ ]:

```
# torch.save({
#             'epoch': 0,
#             'state_dict': AlexNet_Model.state_dict(),
#             'optimizer' : optimizer.state_dict(),
```

```
#            }, '/content/drive/My Drive/Embeddings/code/AlexNet_Model_epoch0.
↪pt')
```

[ ]:
```
for epoch in range(5):  # loop over the dataset multiple times

    running_loss = 0.0
    start_time = time.time()
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data[0].to(device), data[1].to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        output = AlexNet_Model(inputs)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()

        #Time
        end_time = time.time()
        time_taken = end_time - start_time

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss /␣
↪2000))
            print('Time:',time_taken)
            running_loss = 0.0

print('Finished Training of AlexNet: Epoch 10')
```

[ ]:
```
torch.save({
            'epoch': 15,
            'state_dict': AlexNet_Model.state_dict(),
            'optimizer' : optimizer.state_dict(),
            }, '/content/drive/My Drive/Embeddings/code/AlexNet_Model_epoch15.
↪pt')
```

6 Load saved tensors:

```
[ ]: # N_epoch0_fix = loadmat("neuron_output_3D_epoch0.mat")['neuron_output_3D']
# N_epoch3_fix = loadmat("neuron_output_3D_epoch3.mat")['neuron_output_3D']
# N_epoch5_fix = loadmat("neuron_output_3D_epoch5.mat")['neuron_output_3D']
# N_epoch7_fix = loadmat("neuron_output_3D_epoch7.mat")['neuron_output_3D']
# N_epoch10_max = loadmat("neuron_output_3D_epoch10.mat")['neuron_output_3D']
# N_epoch10_conv4 = loadmat("neuron_output_3D_epoch10_conv4.
↪mat")['neuron_output_3D']
# # N_epoch0_fix = loadmat("neuron_output_3D_epoch0.mat")['neuron_output_3D']
# N_epoch1_max = loadmat("neuron_output_3D_epoch1_max.mat")['neuron_output_3D']
# N_epoch5_max = loadmat("neuron_output_3D_epoch5_max.mat")['neuron_output_3D']
# N_epoch7_max = loadmat("neuron_output_3D_epoch7_max.mat")['neuron_output_3D']

N_epoch0_2D_max = loadmat("neuron_output_2D_epoch0_conv1_max.
↪mat")['neuron_output_2D']
N_epoch1_2D_max = loadmat("neuron_output_2D_epoch1_conv1_max.
↪mat")['neuron_output_2D']


N_epoch0_2D = loadmat("neuron_output_2D_epoch0_conv1.mat")['neuron_output_2D']
N_epoch1_2D = loadmat("neuron_output_2D_epoch1_conv1.mat")['neuron_output_2D']
N_epoch2_2D = loadmat("neuron_output_2D_epoch2_conv1.mat")['neuron_output_2D']
N_epoch3_2D = loadmat("neuron_output_2D_epoch3_conv1.mat")['neuron_output_2D']
N_epoch4_2D = loadmat("neuron_output_2D_epoch4_conv1.mat")['neuron_output_2D']
N_epoch5_2D = loadmat("neuron_output_2D_epoch5_conv1.mat")['neuron_output_2D']
N_epoch7_2D = loadmat("neuron_output_2D_epoch7_conv1.mat")['neuron_output_2D']
N_epoch8_2D = loadmat("neuron_output_2D_epoch8_conv1.mat")['neuron_output_2D']
N_epoch5_2D_max = loadmat("neuron_output_2D_epoch5_conv1_max.
↪mat")['neuron_output_2D']
N_epoch10_2D = loadmat("neuron_output_2D_epoch10_conv1.mat")['neuron_output_2D']
N_epoch15_2D_max = loadmat("neuron_output_2D_epoch15_conv1_max.
↪mat")['neuron_output_2D']
N_epoch15_2D = loadmat("neuron_output_2D_epoch15_conv1.mat")['neuron_output_2D']
```

7 Streamline:

```
[ ]: def get_tensor_factors(N, dim = 3, n_vertical_shifts = 22, 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',
    },
    '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_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 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)
    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.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=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.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))
  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()
    print(neuron_factor.shape)
    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()
```

7.1 For 3D epoch 0 fix:

[ ]:
```
ensembles_0 = get_tensor_factors(N_epoch0_fix, ranks = [100,150,200])
```

[ ]:
```
from scipy.io import savemat
mdic = {"ensemble": ensembles_0}
mdic
savemat("ensembles_0.mat", mdic)
```

[ ]:
```
rep = 0
ranks = [125]
tensor_factors = []
for rank in ranks:
  tensor_factors.append(ensembles_0['ncp_hals'].results[rank][rep].factors)
tensor_factors_best_rank = tensor_factors[0]
```

[ ]:
```
get_tensor_factors_plot(tensor_factors_best_rank, best_rank=125,
 ↪n_max_feature_maps=10)
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=10)
```

[ ]:
```
ensembles_1 = get_tensor_factors(N_epoch1_max, ranks = [100,150,200])
```

```
[ ]: rep = 0
ranks = [200]
tensor_factors = []
for rank in ranks:
  tensor_factors.append(ensembles_1['ncp_hals'].results[rank][rep].factors)
tensor_factors_best_rank = tensor_factors[0]

get_tensor_factors_plot(tensor_factors_best_rank, best_rank=200,␣
 ↪n_max_feature_maps=10)
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=10)
```

7.2 For 3D epoch3 fix:

```
[ ]: ensembles_3 = get_tensor_factors(N_epoch3_fix, ranks = [100,150,200])
```

```
[ ]: from scipy.io import savemat
mdic = {"ensemble": ensembles_3}
mdic
savemat("ensembles_3.mat", mdic)

rep = 0
ranks = [200]
tensor_factors = []
for rank in ranks:
  tensor_factors.append(ensembles_3['ncp_hals'].results[rank][rep].factors)
tensor_factors_best_rank = tensor_factors[0]

get_tensor_factors_plot(tensor_factors_best_rank, best_rank=200,␣
 ↪n_max_feature_maps=10)
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=10)
```

7.3 For 3D epoch 5 fix:

```
[ ]: ensembles_5 = get_tensor_factors(N_epoch5_fix, ranks = [100,150,200])
```

```
[ ]: from scipy.io import savemat
mdic = {"ensemble": ensembles_5}
mdic
savemat("ensembles_5.mat", mdic)
```

[ ]:
```
rep = 0
ranks = [200]
tensor_factors = []
for rank in ranks:
  tensor_factors.append(ensembles_5['ncp_hals'].results[rank][rep].factors)
tensor_factors_best_rank = tensor_factors[0]
```

[ ]:
```
get_tensor_factors_plot(tensor_factors_best_rank, best_rank=200,␣
 ↪n_max_feature_maps=10)
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=10)
```

[ ]:
```
ensembles_5 = get_tensor_factors(N_epoch5_max, ranks = [100,150,200])
```

[ ]:
```
rep = 0
ranks = [200]
tensor_factors = []
for rank in ranks:
  tensor_factors.append(ensembles_5['ncp_hals'].results[rank][rep].factors)
tensor_factors_best_rank = tensor_factors[0]
get_tensor_factors_plot(tensor_factors_best_rank, best_rank=200,␣
 ↪n_max_feature_maps=10)
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=10)
```

7.4 For 3D epoch 7 fix:

[ ]:
```
ensembles_7 = get_tensor_factors(N_epoch7_fix, ranks = [100,150,200])
```

[ ]:
```
from scipy.io import savemat
mdic = {"ensemble": ensembles_7}
mdic
savemat("ensembles_7.mat", mdic)
```

[ ]:
```
rep = 0
ranks = [200]
tensor_factors = []
for rank in ranks:
  tensor_factors.append(ensembles_7['ncp_hals'].results[rank][rep].factors)
tensor_factors_best_rank = tensor_factors[0]
get_tensor_factors_plot(tensor_factors_best_rank, best_rank=200,␣
 ↪n_max_feature_maps=10)
```

```
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=10)
```

[ ]:
```
ensembles_7 = get_tensor_factors(N_epoch7_max, ranks = [100,150,200])
```

[ ]:
```
rep = 0
ranks = [200]
tensor_factors = []
for rank in ranks:
  tensor_factors.append(ensembles_7['ncp_hals'].results[rank][rep].factors)
tensor_factors_best_rank = tensor_factors[0]
get_tensor_factors_plot(tensor_factors_best_rank, best_rank=200,␣
 ↪n_max_feature_maps=10)
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=10)
```

7.5 For 3D epoch 10 max:

[ ]:
```
ensembles_10 = get_tensor_factors(N_epoch10_conv4, ranks = [100,150,200])
```

[ ]:
```
rep = 0
ranks = [200]
tensor_factors = []
for rank in ranks:
  tensor_factors.append(ensembles_10['ncp_hals'].results[rank][rep].factors)
tensor_factors_best_rank = tensor_factors[0]
get_tensor_factors_plot(tensor_factors_best_rank, best_rank=200,␣
 ↪n_max_feature_maps=10)
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=10)
```

[ ]:
```
ensembles_10 = get_tensor_factors(N_epoch10_max, ranks = [100,150,200])
```

[ ]:
```
rep = 0
ranks = [200]
tensor_factors = []
for rank in ranks:
  tensor_factors.append(ensembles_10['ncp_hals'].results[rank][rep].factors)
tensor_factors_best_rank = tensor_factors[0]
```

```
[ ]: get_tensor_factors_plot(tensor_factors_best_rank, best_rank=200,␣
     ↪n_max_feature_maps=10)
     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=10)
```

7.6 For 3D epoch 12 fix:

```
[ ]: N_epoch12_fix.shape
```

```
[ ]: ensembles_12 = get_tensor_factors(N_epoch12_fix, ranks = [50, 75,100,125])
```

```
[ ]: from scipy.io import savemat
     mdic = {"ensemble": ensembles_12}
     mdic
     savemat("ensembles_12.mat", mdic)
```

```
[ ]: rep = 0
     ranks = [125]
     tensor_factors = []
     for rank in ranks:
       tensor_factors.append(ensembles_12['ncp_hals'].results[rank][rep].factors)
     tensor_factors_best_rank = tensor_factors[0]
```

```
[ ]: get_tensor_factors_plot(tensor_factors_best_rank, best_rank=125,␣
     ↪n_max_feature_maps=5, feature_map_side=13)
     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)
```

7.7 For 3D epoch 14 fix:

```
[ ]: N_epoch14_fix.shape
```

```
[ ]: ensembles_14 = get_tensor_factors(N_epoch14_fix, ranks = [50, 75,100,125])
```

```
[ ]: from scipy.io import savemat
     mdic = {"ensemble": ensembles_14}
     mdic
     savemat("ensembles_14.mat", mdic)
```

```
[ ]: rep = 0
     ranks = [125]
     tensor_factors = []
```

```
for rank in ranks:
  tensor_factors.append(ensembles_14['ncp_hals'].results[rank][rep].factors)
tensor_factors_best_rank = tensor_factors[0]
```

[ ]:
```
get_tensor_factors_plot(tensor_factors_best_rank, best_rank=125,␣
↪n_max_feature_maps=5, feature_map_side=13)
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)
```

7.8 For 3D epoch 16 max:

[ ]:
```
N_epoch16_max.shape
```

[ ]:
```
im_ind = 0
n_max_feature_maps = 5
feature_map_side = 13
# f_i, axes = plt.subplots(1,n_max_feature_maps,figsize=(10,1))
neuron_matrix = N_epoch16_max[:,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 = 256
  feature_map_side = 13
  for t in range(N_frames):
    neuron_matrix_fi_t = neuron_matrix_fi[:, t].reshape((feature_map_side,␣
↪feature_map_side))
    print(neuron_matrix_fi_t)
    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_16 = get_tensor_factors(N_epoch16_max, ranks =  [50, 75,100,125])
```

[ ]:
```
from scipy.io import savemat
mdic = {"ensemble": ensembles_16}
mdic
savemat("ensembles_16.mat", mdic)
```

[ ]:
```
rep = 0
ranks = [125]
tensor_factors = []
for rank in ranks:
```

```
  tensor_factors.append(ensembles_16['ncp_hals'].results[rank][rep].factors)
tensor_factors_best_rank = tensor_factors[0]
```

[ ]:
```
get_tensor_factors_plot(tensor_factors_best_rank, best_rank=125,␣
 ↪n_max_feature_maps=5, feature_map_side=13)
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)
```

7.9 For 2D epoch 0 fix:

[ ]:
```
ensembles_2D = get_tensor_factors(N_epoch0_2D, dim =2, ranks=[1,2,3,4,5])
```

[ ]:
```
rep = 0
ranks=[2]
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=2,␣
 ↪n_max_feature_maps=10)
neuron_output_highest_with_shifts_PCA =␣
 ↪get_embeddings_2D(tensor_factors_best_rank, neuron_labels, n_dim_PCA = 2)
get_spatial_order_plot_2D(1, neuron_output_highest_with_shifts_PCA,␣
 ↪neuron_labels, n_max_feature_maps=10)
```

[ ]:
```
ensembles_2D = get_tensor_factors(N_epoch1_2D, dim =2, ranks=[1,2,3,4,5])
```

[ ]:
```
rep = 0
ranks=[2]
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=2,␣
 ↪n_max_feature_maps=10)
neuron_output_highest_with_shifts_PCA =␣
 ↪get_embeddings_2D(tensor_factors_best_rank, neuron_labels, n_dim_PCA = 2)
get_spatial_order_plot_2D(1, neuron_output_highest_with_shifts_PCA,␣
 ↪neuron_labels, n_max_feature_maps=10)
```

[ ]:
```
ensembles_2D = get_tensor_factors(N_epoch2_2D, dim =2, ranks=[1,2,3,4,5])
```

```
[ ]: rep = 0
     ranks=[2]
     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=2,␣
      ↪n_max_feature_maps=10)
     neuron_output_highest_with_shifts_PCA =␣
      ↪get_embeddings_2D(tensor_factors_best_rank, neuron_labels, n_dim_PCA = 2)
     get_spatial_order_plot_2D(1, neuron_output_highest_with_shifts_PCA,␣
      ↪neuron_labels, n_max_feature_maps=10)
```

```
[ ]: ensembles_2D = get_tensor_factors(N_epoch3_2D, dim =2, ranks=[1,2,3,4,5])
     rep = 0
     ranks=[2]
     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=2,␣
      ↪n_max_feature_maps=10)
     neuron_output_highest_with_shifts_PCA =␣
      ↪get_embeddings_2D(tensor_factors_best_rank, neuron_labels, n_dim_PCA = 2)
     get_spatial_order_plot_2D(1, neuron_output_highest_with_shifts_PCA,␣
      ↪neuron_labels, n_max_feature_maps=10)
```

```
[ ]: ensembles_2D = get_tensor_factors(N_epoch4_2D, dim =2, ranks=[1,2,3,4,5])
     rep = 0
     ranks=[2]
     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=2,␣
      ↪n_max_feature_maps=10)
     neuron_output_highest_with_shifts_PCA =␣
      ↪get_embeddings_2D(tensor_factors_best_rank, neuron_labels, n_dim_PCA = 2)
     get_spatial_order_plot_2D(1, neuron_output_highest_with_shifts_PCA,␣
      ↪neuron_labels, n_max_feature_maps=10)
```

7.10 For 2D epoch 5 fix:

[ ]:
```
ensembles_2D = get_tensor_factors(N_epoch5_2D, dim =2, ranks=[1,2,3,4,5])
```

[ ]:
```
rep = 0
ranks=[2]
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=2,␣
 ↪n_max_feature_maps=10)
neuron_output_highest_with_shifts_PCA =␣
 ↪get_embeddings_2D(tensor_factors_best_rank, neuron_labels, n_dim_PCA = 2)
get_spatial_order_plot_2D(1, neuron_output_highest_with_shifts_PCA,␣
 ↪neuron_labels, n_max_feature_maps=10)
```

7.11 For 2D epoch 7 fix:

[ ]:
```
ensembles_2D = get_tensor_factors(N_epoch7_2D, dim =2, ranks=[1,2,3,4,5])
rep = 0
ranks=[2]
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=2,␣
 ↪n_max_feature_maps=10)
neuron_output_highest_with_shifts_PCA =␣
 ↪get_embeddings_2D(tensor_factors_best_rank, neuron_labels, n_dim_PCA = 2)
get_spatial_order_plot_2D(1, neuron_output_highest_with_shifts_PCA,␣
 ↪neuron_labels, n_max_feature_maps=10)
```

[ ]:
```
ensembles_2D = get_tensor_factors(N_epoch8_2D, dim =2, ranks=[1,2,3,4,5])
rep = 0
ranks=[2]
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=2,␣
 ↪n_max_feature_maps=10)
neuron_output_highest_with_shifts_PCA =␣
 ↪get_embeddings_2D(tensor_factors_best_rank, neuron_labels, n_dim_PCA = 2)
```

```
get_spatial_order_plot_2D(1, neuron_output_highest_with_shifts_PCA,␣
↪neuron_labels, n_max_feature_maps=10)
```

7.12 For 2D epoch 10:

[ ]:
```
ensembles_2D = get_tensor_factors(N_epoch10_2D , dim =2, ranks=[1,2,3,4,5])
```

[ ]:
```
rep = 0
ranks=[2]
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=2,␣
↪n_max_feature_maps=10)
neuron_output_highest_with_shifts_PCA =␣
↪get_embeddings_2D(tensor_factors_best_rank, neuron_labels, n_dim_PCA = 2)
get_spatial_order_plot_2D(1, neuron_output_highest_with_shifts_PCA,␣
↪neuron_labels, n_max_feature_maps=10)
```

[ ]:
```
ensembles_2D = get_tensor_factors(N_epoch15_2D_max , dim =2, ranks=[1,2,3,4,5])
rep = 0
ranks=[2]
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=2,␣
↪n_max_feature_maps=10)
neuron_output_highest_with_shifts_PCA =␣
↪get_embeddings_2D(tensor_factors_best_rank, neuron_labels, n_dim_PCA = 2)
get_spatial_order_plot_2D(1, neuron_output_highest_with_shifts_PCA,␣
↪neuron_labels, n_max_feature_maps=10)
```

[ ]:
```
ensembles_2D = get_tensor_factors(N_epoch15_2D , dim =2, ranks=[1,2,3,4,5])
rep = 0
ranks=[2]
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=2,␣
↪n_max_feature_maps=10)
```

```
neuron_output_highest_with_shifts_PCA =␣
↪get_embeddings_2D(tensor_factors_best_rank, neuron_labels, n_dim_PCA = 2)
get_spatial_order_plot_2D(1, neuron_output_highest_with_shifts_PCA,␣
↪neuron_labels, n_max_feature_maps=10)
```

[ ]:
```
!pip install tensorly
```

[ ]:
```
import numpy as np
import tensorly as tl
from tensorly.decomposition import non_negative_parafac,␣
↪non_negative_parafac_hals
from tensorly.decomposition._nn_cp import initialize_nn_cp
from tensorly.cp_tensor import CPTensor
import time
from copy import deepcopy
```

[ ]:
```
tic = time.time()
weights_init, factors_init = initialize_nn_cp(N_epoch16_max10_2D,␣
↪init='random', rank=10)
cp_init = CPTensor((weights_init, factors_init))
tensor_hals, errors_hals = non_negative_parafac_hals(N_epoch16_max10_2D,␣
↪rank=10, init=deepcopy(cp_init), return_errors=True)
cp_reconstruction_hals = tl.cp_to_tensor(tensor_hals)
time_hals = time.time()-tic
```

[ ]:
```
import matplotlib.pyplot as plt
def each_iteration(a, title):
    fig=plt.figure()
    fig.set_size_inches(10, fig.get_figheight(), forward=True)
    plt.plot(a)
    plt.title(str(title))
    plt.legend(['HALS'], loc='upper left')

each_iteration(errors_hals, 'Error for each iteration')
```

[ ]:
```
tic = time.time()
weights_init, factors_init = initialize_nn_cp(N_epoch16_max10, init='random',␣
↪rank=10)
cp_init = CPTensor((weights_init, factors_init))
tensor_hals_3D, errors_hals_3D = non_negative_parafac_hals(N_epoch16_max10,␣
↪rank=10, init=deepcopy(cp_init), return_errors=True)
cp_reconstruction_hals_3D = tl.cp_to_tensor(tensor_hals_3D)
time_hals = time.time()-tic
```

[ ]:
```
import matplotlib.pyplot as plt
def each_iteration(a, title):
```

```
    fig=plt.figure()
    fig.set_size_inches(10, fig.get_figheight(), forward=True)
    plt.plot(a)
    plt.title(str(title))
    plt.legend(['HALS'], loc='upper left')

each_iteration(errors_hals_3D, 'Error for each iteration')
```

[ ]:
```
N = N_epoch16_max10
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((32,32)), sigma=1).
 ↪reshape((1024,))
    N_filtered[i,j,:] = filtered[:]
tic = time.time()
weights_init, factors_init = initialize_nn_cp(N_filtered, init='random',␣
 ↪rank=10)
cp_init = CPTensor((weights_init, factors_init))
tensor_hals_3D, errors_hals_3D = non_negative_parafac_hals(N_filtered, rank=10,␣
 ↪init=deepcopy(cp_init), return_errors=True)
cp_reconstruction_hals_3D = tl.cp_to_tensor(tensor_hals_3D)
time_hals = time.time()-tic
```

[ ]:
```
import matplotlib.pyplot as plt
def each_iteration(a, title):
    fig=plt.figure()
    fig.set_size_inches(10, fig.get_figheight(), forward=True)
    plt.plot(a)
    plt.title(str(title))
    plt.legend(['HALS'], loc='upper left')

each_iteration(errors_hals_3D, 'Error for each iteration')
```

8 Get accuracy:

[ ]:
```
!pip install tqdm
import tqdm
```

[ ]:
```
# define function to calculate current accuracy with a given dataloader
def accuracy(net, dataloader, device='cpu'): #Get the accuracies
  net.eval()
  correct = 0
  count = 0
  for data, target in tqdm.tqdm(trainloader):
    data = data.to(device).float()
```

```
    target = target.to(device).long()
    data = data.type(torch.cuda.FloatTensor)
    target = target.type(torch.cuda.FloatTensor)

    prediction = net(data)
    _, predicted = torch.max(prediction, 1)
    count += target.size(0)
    correct += (predicted == target).sum().item()

  acc = 100 * correct / count
  return count, acc


# define function to evaluate and print training and test accuracy
def evaluate(net, device='cpu', title=""):
  net.eval()
  train_count, train_acc = accuracy(net, train_data, device=device)
  test_count, test_acc = accuracy(net, test_data, device=device)
  print(f'Accuracy on the {train_count} training samples {title}: {train_acc:0.
 ↪2f}')
  print(f'Accuracy on the {test_count} testing samples {title}: {test_acc:0.
 ↪2f}')
```

[ ]:
```
evaluate(AlexNet_Model_epoch10)
```

[ ]:
```
evaluate(AlexNet_Model_epoch10)
```

[ ]:
```
evaluate(AlexNet_Model_epoch12)
```

[ ]:
```
evaluate(AlexNet_Model_epoch14)
```

[ ]:
```
evaluate(AlexNet_Model_epoch16)
```

9 Visualizing filters:

[ ]:
```
def show_weights(layer, i=0):
    filters = layer[0].weight.cpu().data # [0] is to get the conv_2d layer

    # normalize filter values to 0-1 so we can visualize them
    f_min, f_max = filters.min(), filters.max()
    filters = (filters - f_min) / (f_max - f_min)
    print(filters.shape)

    fig, axs = plt.subplots(5, 5, figsize =(20, 10))
    for i , ax in enumerate(axs.flatten()):
        image = filters[i]
```

```
        ax.imshow(image.permute(1, 2, 0))
        ax.axis('off')
    plt.tight_layout
    # img = torchvision.utils.make_grid(filters)
    # npimg = img.numpy()
    # print(npimg.shape)
    # print(np.transpose(npimg, (1, 2, 0)).shape)
    # plt.imshow(np.transpose(npimg, (1, 2, 0)))
    # plt.show()
```

[ ]:
```
show_weights(AlexNet_Model_epoch0.conv_layer3)
```

[ ]:
```
show_weights(AlexNet_Model_epoch1.conv_layer1)
```

[ ]:
```
show_weights(AlexNet_Model_epoch10.conv_layer1)
```

[ ]:
```
show_weights(AlexNet_Model_epoch0.conv_layer1)
```

[ ]:
```
show_weights(AlexNet_Model_epoch1.conv_layer1)
```

[ ]:
```
show_weights(AlexNet_Model_epoch2.conv_layer1)
```

[ ]:
```
show_weights(AlexNet_Model_epoch3.conv_layer1)
```

[ ]:
```
show_weights(AlexNet_Model_epoch4.conv_layer1)
```

[ ]:
```
show_weights(AlexNet_Model_epoch5.conv_layer1)
```

[ ]:
```
show_weights(AlexNet_Model_epoch10.conv_layer1)
```

[ ]:
```

```

10 Visualizing outputs given an image:

[ ]:
```
dataiter = iter(trainloader)
images, labels = dataiter.next()
```

[ ]:
```
imshow(images[2])
```

[ ]:
```
images = images.type(torch.cuda.FloatTensor)
im = images[0].unsqueeze(0)
```

[ ]:
```
im.shape
```

[ ]:
```
!pip install torch_intermediate_layer_getter
```

```
[ ]: ## DEBUG NOTE: https://discuss.pytorch.org/t/
     ↪how-can-i-extract-intermediate-layer-output-from-loaded-cnn-model/77301/12
     ## The forward hook registered in model.fc returns the "pre-relu" activation,␣
     ↪since negative values are shown.

     import torchvision
     from torchvision.models._utils import IntermediateLayerGetter
     return_layers = {
         # "<name of layer in the AlexNet Class>" : "<key for the layer output in␣
     ↪the returned dictionary>"
           'conv_layer1': 'conv_layer1',
           'conv_layer3': 'conv_layer3',
           'conv_layer5': 'conv_layer5',
       }
     model_with_multuple_layer = IntermediateLayerGetter(AlexNet_Model_epoch7,␣
     ↪return_layers=return_layers)
     intermediate_output = model_with_multuple_layer(im)
     intermediate_output
```

```
[ ]: # from torch_intermediate_layer_getter import IntermediateLayerGetter as␣
     ↪LayerGetter
     # return_layers = {
     #     # "<name of layer in the AlexNet Class>" : "<key for the layer output in␣
     ↪the returned dictionary>"
     #       'conv_layer1': 'conv_layer1',
     #       'conv_layer3': 'conv_layer3',
     #       'conv_layer5': 'conv_layer5',
     #   }
     # net = AlexNet_Model_epoch7
     # LayerGetter_ = LayerGetter(net, return_layers=return_layers)
     # intermediate_output = LayerGetter_(im)
     # print(intermediate_output['conv_layer1'].shape)
     # intermediate_output['conv_layer1'].min()
```

```
[ ]: fig, axs = plt.subplots(5, 5, figsize = (10,10))
     for i , ax in enumerate(axs.flatten()):
         ax.imshow(intermediate_output['conv_layer1'][1,i,:,:].cpu().data)
     fig.suptitle('Visualizing Outputs for first 25 filters in conv_layer1',␣
     ↪fontsize=20)
```

11 Visualizing embeddings (3D tensor):

```
[ ]: !python3 -m pip install --upgrade pip
     !python3 -m pip install --upgrade Pillow
     from PIL import Image, ImageTk
```

```python
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')

def get_images_selected_classes(num_images, n_classes):
  images_selected_classes = []
  MAX_SIZE = 64
  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)
for i in range(20):
  plt.imshow(images_selected_classes[i])
  plt.show()

images_selected_classes = images_selected_classes.transpose((0,3,1,2))
print(images_selected_classes.shape)
```

[ ]:
```python
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
```

```
    '''

    ## 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(im)
    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(im_shift)
        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(im_shift)

    im_all_shifts =  np.array(im_all_shifts)

    return im_all_shifts, n_shifts
```

[ ]:
```
AlexNet_Model_epoch0.conv_layer1[0].
↪register_forward_hook(getActivation('conv_layer1'))
```

[ ]:
```
# https://web.stanford.edu/~nanbhas/blog/forward-hooks-pytorch/

model = AlexNet_Model_epoch16
# a dict to store the activations
activation = {}

def getActivation(name):
  # the hook signature
  def hook(model, input, output):
    activation[name] = output.detach()
  return hook

# register forward hooks on the layers of choice
h1 = model.conv_layer1[0].register_forward_hook(getActivation('conv_layer1'))

conv_layer1_list = []
```

```
# go through all the batches in the dataset

# forward pass -- getting the outputs
im = torch.tensor(images_selected_classes[0])
out = model(im.unsqueeze(dim=0).to(device).float())
# collect the activations in the correct list
conv_layer1_list.append(activation['conv_layer1'])

# detach the hooks
h1.remove()
```

[ ]:
```
conv_layer1_list[0][0].shape
```

[ ]:
```
conv_layer1_list[0].shape

plt.imshow(conv_layer1_list[0][0,3,:,:].cpu().data)
```

[ ]:
```
# https://web.stanford.edu/~nanbhas/blog/forward-hooks-pytorch/

model = AlexNet_Model_epoch16
# a dict to store the activations
activation = {}

def getActivation(name):
  # the hook signature
  def hook(model, input, output):
    activation[name] = output.detach()
  return hook

# register forward hooks on the layers of choice
h1 = model.conv_layer1[0].register_forward_hook(getActivation('conv_layer1'))

conv_layer1_list = []
# go through all the batches in the dataset
for i in range(20):
  # forward pass -- getting the outputs
  im = torch.tensor(images_selected_classes[i])
  out = model(im.unsqueeze(dim=0).to(device).float())
  # collect the activations in the correct list
  conv_layer1_list.append(activation['conv_layer1'])

# detach the hooks
h1.remove()
```

[ ]:
```
len(conv_layer1_list)
```

[ ]:
```
for i in range(20):
  plt.imshow(conv_layer1_list[i][0,3,:,:].cpu().data)
  plt.show()
```

[ ]:
```
conv_layer1_list[1].shape
```

[ ]:
```
A = np.random.randn(4,5)
A
```

[ ]:
```
B = np.random.randn(4,5)
B
```

[ ]:
```
np.concatenate((A,B), axis = 1)
```

[ ]:
```
def compute_neuron_output(model, layer_names, im_all_shifts,  max_indices =␣
 ↪None, n_max_feature_maps = 10, 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)
    n_shifts = im_all_shifts.shape[0]
    n_layers = len(layer_names)
    neuron_output_highest = []
    all_fm_avg = []
    im_all_shifts = torch.tensor(np.array(im_all_shifts))
    im_all_shifts = im_all_shifts.type(torch.cuda.FloatTensor)

    is_first_layer = True
    for layer_name in layer_names:
        ## note: always take the images as inputs

        # a dict to store the activations
        activation = {}

        def getActivation(name):
          # the hook signature
          def hook(model, input, output):
            activation[name] = output.detach()
          return hook
        # register forward hooks on the layers of choice
```

```
        h1 = model.conv_layer1[0].
↪register_forward_hook(getActivation('conv_layer1'))

        conv_layer1_list = []
        # go through all the batches in the dataset
        for i in range(n_shifts):
          # forward pass -- getting the outputs
          im = im_all_shifts[i]
          out = model(im.unsqueeze(dim=0))
          # collect the activations in the correct list
          conv_layer1_list.append(activation['conv_layer1'])

        # detach the hooks
        h1.remove()

        # return_layers = {layer_name: layer_name}
        # LayerGetter = torchvision.models._utils.
↪IntermediateLayerGetter(model, return_layers=return_layers)
        # neuron_output = []
        # for i in range(n_shifts):
        #     im = im_all_shifts[i]
        #     im = torch.reshape(im, (1, im.shape[0],im.shape[1],im.shape[2]))
        #     output = LayerGetter(im)[layer_name]
        #     relu = torch.nn.ReLU()
        #     output = relu(output)
        #     neuron_output.append(output)
        # neuron_output =  torch.stack(neuron_output)
        # neuron_output = neuron_output.reshape((neuron_output.shape[0],␣
↪neuron_output.shape[2], neuron_output.shape[3], neuron_output.shape[4]))

        n_feature_maps, n_row, n_col = conv_layer1_list[0][0,:,:,:].shape[:]
        neuron_output = np.empty((n_shifts, n_feature_maps, n_row, n_col))
l
        for i in range(n_shifts):
          neuron_output[i,:,:,:] = conv_layer1_list[i][0,:,:,:].cpu().data

        print(neuron_output.shape)
        ## neuron_output has shape (n_shifts, n_feature_maps, nrow, ncol)
        neuron_output = np.transpose(neuron_output, (0, 2, 3, 1))
        ## after permuting, neuron_output has shape (n_shifts, nrow, ncol,␣
↪n_feature_maps)
        # n_shifts, n_row, n_col, n_feature_maps = neuron_output.shape[:]

        ## remove the neurons at the edges

        # based on the AlexNet model structure:
        filter_size = 11
```

```
        shift_step = 4
        edge_neuron = math.floor(filter_size / shift_step)
        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

        # 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))

        ## transpose to organize by feature maps
        ## the shape of neuron_output_by_fm is (n_shifts, n_feature_maps,␣
↪#neurons)
        neuron_output_by_fm = np.transpose(neuron_output, (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 is_first_layer is True:
            fm_avg_all_layers = fm_avg
            neuron_output_by_fm_all_layers = neuron_output_by_fm
        else:
```

```
            fm_avg_all_layers = np.hstack((fm_avg_all_layers, fm_avg))
            neuron_output_by_fm_all_layers = np.
↪concatenate((neuron_output_by_fm_all_layers,neuron_output_by_fm), axis = 1)
            is_first_layer = False
        # if plot_activity == True:
        #     layer = model.layers[layer_index]
        #     print("Current layer: " + layer.name)
        #     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_row * n_col))
        #     print("Average activity for all feature maps in " + layer.name)
        #     plt.matshow(fm_avg)
        #     plt.show()
        #     print('-----------------------------------------------------\n')

    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_all_layers[i], axis = 0)[:
↪n_max_feature_maps]
        temp = neuron_output_by_fm_all_layers[i, max_indices, :]
        neuron_output_highest[i] = neuron_output_by_fm_all_layers[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

    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_all_layers, neuron_labels,␣
↪max_indices, n_neurons
```

```
[ ]: def show_stimuli_3D(model, layer_names, images_selected_classes, shifts,␣
     ↪max_indices, n_images_selected_classes, shift_step = 4, 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, layer_names, 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_names, 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)

                 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:
                     fm_avg_all_layers_with_shifts = fm_shifts_avg
                 else:
```

```
                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_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
        fm_avg_all_layers = fm_avg_all_layers_with_shifts

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

[ ]:
```
# def get_images_selected_classes():
#   airplane_indices, automobile_indices, cat_indices, dog_indices = [], [],␣
↪[], []
#   airplane_idx, automobile_idx, cat_idx, dog_idx = train_data.
↪class_to_idx['airplane'], train_data.class_to_idx['automobile'], train_data.
↪class_to_idx['cat'], train_data.class_to_idx['dog']

#   i = 0
#   n_images_per_class = 5
#   while(len(airplane_indices) < n_images_per_class or len(automobile_indices)␣
↪< n_images_per_class or len(cat_indices) < n_images_per_class or␣
↪len(dog_indices) < n_images_per_class):
#       current_class = train_data[i][1]
#       if current_class == airplane_idx:
#           airplane_indices.append(i)
#       elif current_class == automobile_idx:
#           automobile_indices.append(i)
#       elif current_class == cat_idx:
#           cat_indices.append(i)
#       elif current_class == dog_idx:
#           dog_indices.append(i)
#       i += 1

#   airplane_indices = airplane_indices[0:n_images_per_class]
#   automobile_indices = automobile_indices[0:n_images_per_class]
#   cat_indices = cat_indices[0:n_images_per_class]
#   dog_indices = dog_indices[0:n_images_per_class]
#   selected_train_data = Subset(train_data, airplane_indices +␣
↪automobile_indices + cat_indices + dog_indices)
#   n_images_selected_classes = n_images_per_class * 4

#   # get some random training images
```

```
#   trainloader_images_selected_classes = torch.utils.data.
↪DataLoader(selected_train_data, batch_size=n_images, shuffle=False,␣
↪num_workers=2)
#   dataiter_images_selected_classes = iter(trainloader_images_selected_classes)
#   images_selected_classes, labels_selected_classes =␣
↪dataiter_images_selected_classes.next()

#   return images_selected_classes
```

11.1 Save and load .mat data for TCA:

[ ]:
```
neuron_output_highest_with_shifts, fm_avg_all_layers_with_shifts,␣
↪neuron_labels, max_indices = show_stimuli_3D(
    AlexNet_Model_epoch10, ['conv_layer4'], images_selected_classes, shifts =␣
↪True,  max_indices = None, n_images_selected_classes = num_images,␣
↪n_max_feature_maps=10, plot_activity = True)
```

[ ]:
```
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_epoch10_conv4.mat", mdic)
```

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

Note: output size = [(W−K+2P)/S]+1 = (128-11)/4+1 = ~30

-> after downsampling = 15

-> minus edge neuron = 13

thus, 5 * 13 * 13 = 845

e.g. conv_layer W is the input volume - 128 K is the Kernel size - 11 P is the padding - not sure (assuming 0) S is the stride - 4

edge_neuron = floor(filter_size /stride) = floor(11/4) = 2

shifts = 16 * 16 = 256

[ ]:
```
max_indices
```

[ ]:
```
max_indices = [13, 30,  7, 31,  3,  2,  0,  1, 12, 26]
```

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

12 Visualizing embeddings (2D tensor):

```
[ ]: def show_stimuli_2D(model, layer_names, 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, layer_names, 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_names, 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(
    AlexNet_Model_epoch15, ['conv_layer1'], images_selected_classes, shifts = 
 ↪True,  max_indices = max_indices, n_images_selected_classes = num_images, 
 ↪n_max_feature_maps=10, plot_activity = True)
```

[ ]:
```
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_epoch15_conv1.mat", mdic)
```

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

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

[ ]:
```
max_indices
```

[ ]:
```

```