Sheet

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from PIL import Image
import matplotlib.pyplot as plt
from scipy.signal import savgol_filter

%matplotlib inline
```

Method 1 - Remove features with PCA

Setup

Dataset: Subset of CIFAR10 which includes 4 classes: plane, car, bird and cat.

Net: Resnet18

training with colab GPU.

Experiment hyperparameters

We found the following hyperparameters to work best:

- learning rate = 0.1
- batch size = 100
- SGD optimizer
  - Weight decay = 5e-4
  - Momentumm = 0.9
- StepLR scheduler:
  - step size = 5
  - gamma = 0.1

Learning rate decay = 1

In this setup we followed the following steps:

1. Extract k principle components from Train dataset for every channel (RGB)
2. Project images of train and test into train subspace (from 1), of 3 channels.
3. Run standard transformation on images (to increase generalization).
4. Add 0.2 label noise.
5. Load pretrained Resnet18
6. Train Resnet18 with new projected datasets for ~130 epochs, with above hyperparametrs.

Important comment - the following plots and code correspond to figure 4 in the paper

```
## Fig4 from paper

plt.figure(figsize=(10,10))
plt.axis('off')
plt.imshow(mpimg.imread('fig4 - paper.JPG'))
```

```
<matplotlib.image.AxesImage at 0x7f3cb0a50760>
```

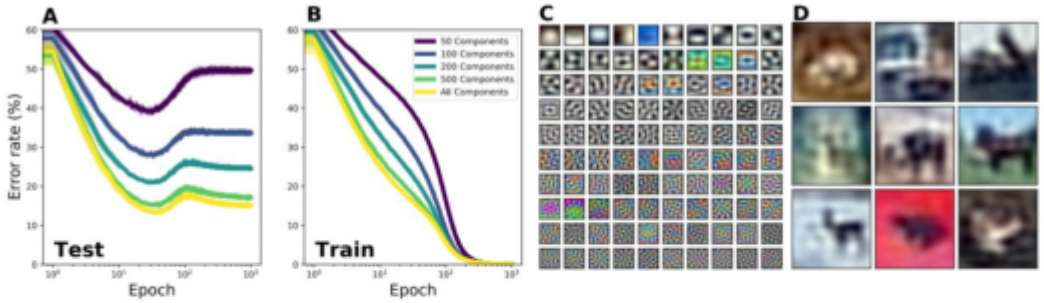


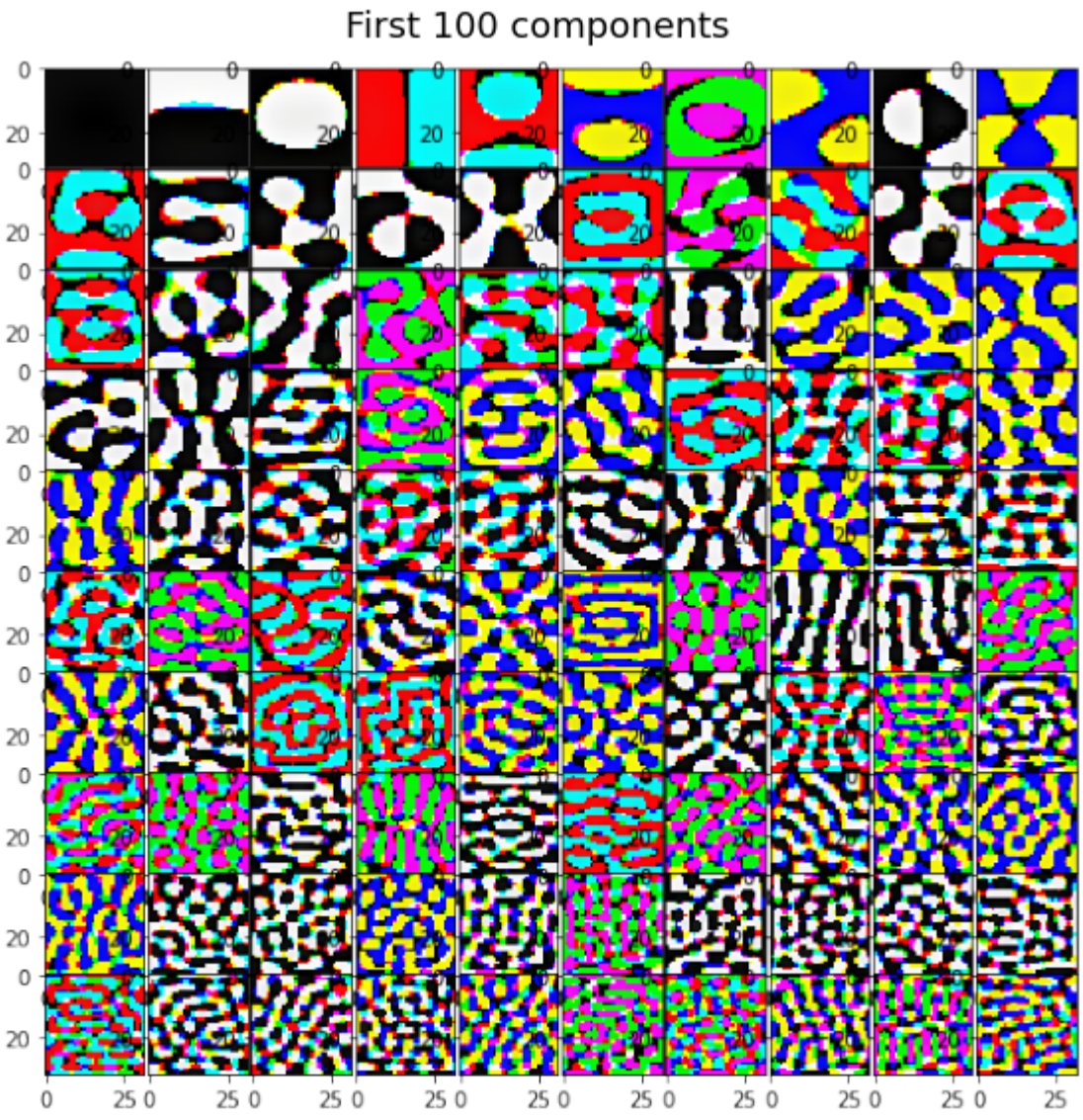
Figure 4: **A:** Generalization error of ResNet18 trained on CIFAR10 with 20% label noise ($\epsilon = 0.2$) as an increasing number of the smaller principal components are removed. When ≤ 100 components remain, epochwise double descent disappears. **B:** Training error as an increasing number of the smaller principal components are removed. **C:** Visualization of the first 100 principal components showing that the larger principal components correspond to larger scale features. **D:** Examples of the training data when only the 100 largest principal components are kept. All lines show mean (bold) $\pm$ std (shaded) across 10 random seeds.

100 first principle components

We shall note that in correspondence to the paper, the smaller scale features look indeed 'less important'. In addition, it is interesting to see that as the index of the component gets bigger (i.e the component matches a bigger eigenvalue), the image is more clear and defined.

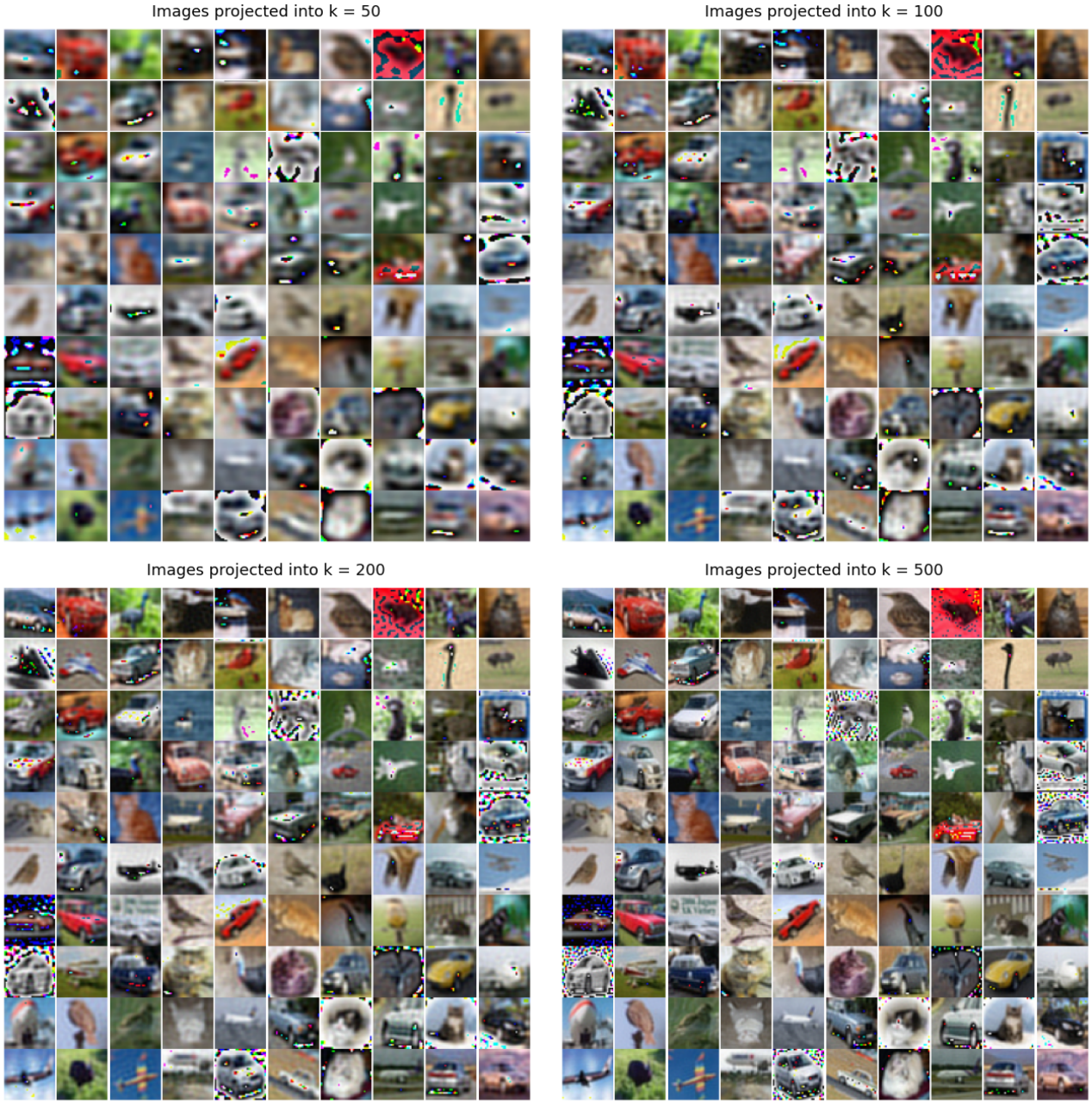
```
plt.figure(figsize=(10,10))
plt.axis('off')
plt.title("First 100 components")
plt.imshow(mpimg.imread('comps.png'))
```

```
<matplotlib.image.AxesImage at 0x7fee96dd3610>
```



Show reconstruction of images - projected for different K's

```
fig, axes = plt.subplots(nrows = 2, ncols = 2, figsize= (18, 18))
plt.rcParams['font.size'] = '15'
k_list = [50, 100, 200, 500]
for k, ax in zip(k_list, axes.flatten()) :
    ax.imshow(mpimg.imread(f'images_proj_k={k}.png'))
    ax.set_title(f'Images projected into k = {k}')
    ax.set_axis_off()
plt.tight_layout()
```



Show results of train and test errors which correspond to

```
##load results of make_cnn network
acc_train_k100 = np.load('acc_train_k100.npy')
acc_test_k100 = np.load('acc_test_k100.npy')
acc_train_k500 = np.load('acc_train_k500.npy')
acc_test_k500 =  np.load('acc_test_k500.npy')
```

Intro:

In the following figures we will see that in correspondence to the results in the paper, there are a few interesting phenomena:

1. For k > 100 we can see a clear epoch-wise double descent.
2. For k= 50 we can see that there is no obvious double descent.

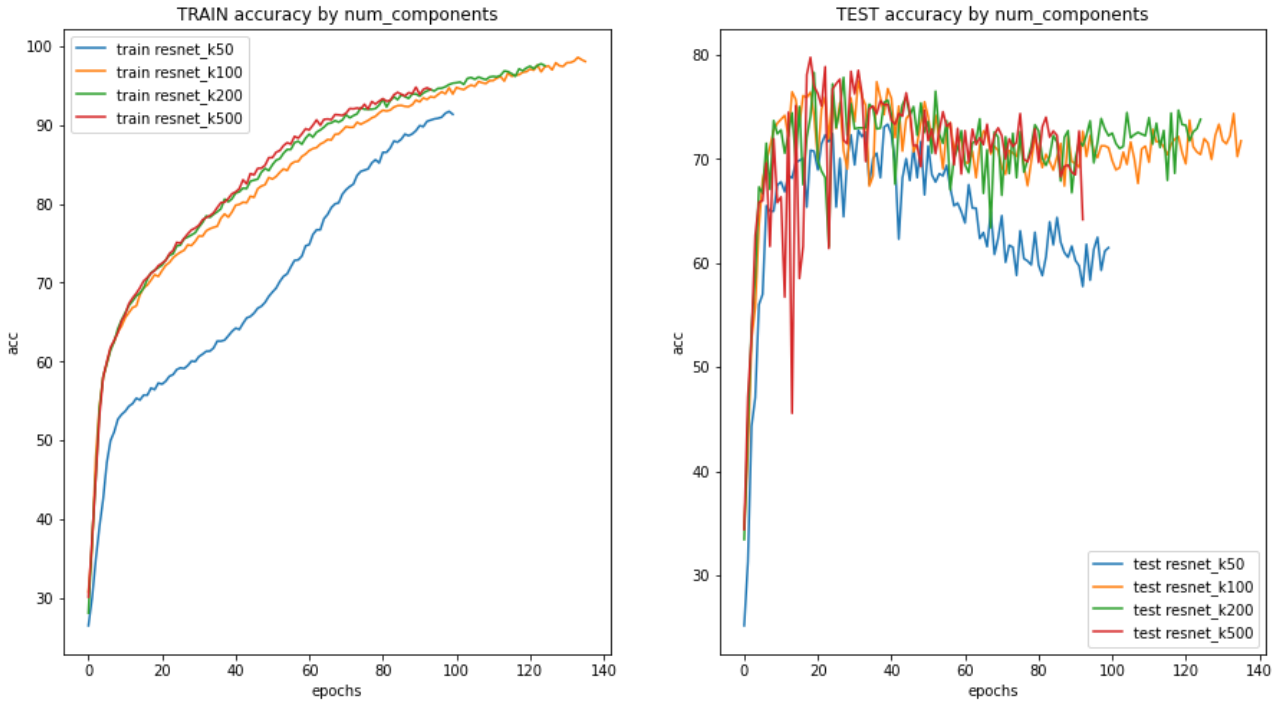
We shall note that because we used a large learning rate (0.1), the error is not smooth, thus the results are not exactly like the ones presented in the paper and for the purpose of seeing a clear trend we smoothed them.

```
import numpy as np
##load result of resnet18
acc_train_dict = {'resnet_k50':  np.load('acc_train_resnet_k50.npy'),
                  'resnet_k100': np.load('acc_train_resnet_k100.npy'),
                 'resnet_k200': np.load('acc_train_resnet_k200.npy'),
                 'resnet_k500': np.load('acc_train_resnet_k500_new.npy')}

acc_test_dict = {'resnet_k50': np.load('acc_test_resnet_k50.npy'),
                  'resnet_k100': np.load('acc_test_resnet_k100.npy'),
                 'resnet_k200': np.load('acc_test_resnet_k200.npy'),
                 'resnet_k500': np.load('acc_test_resnet_k500_new.npy')}
```

PLOTS

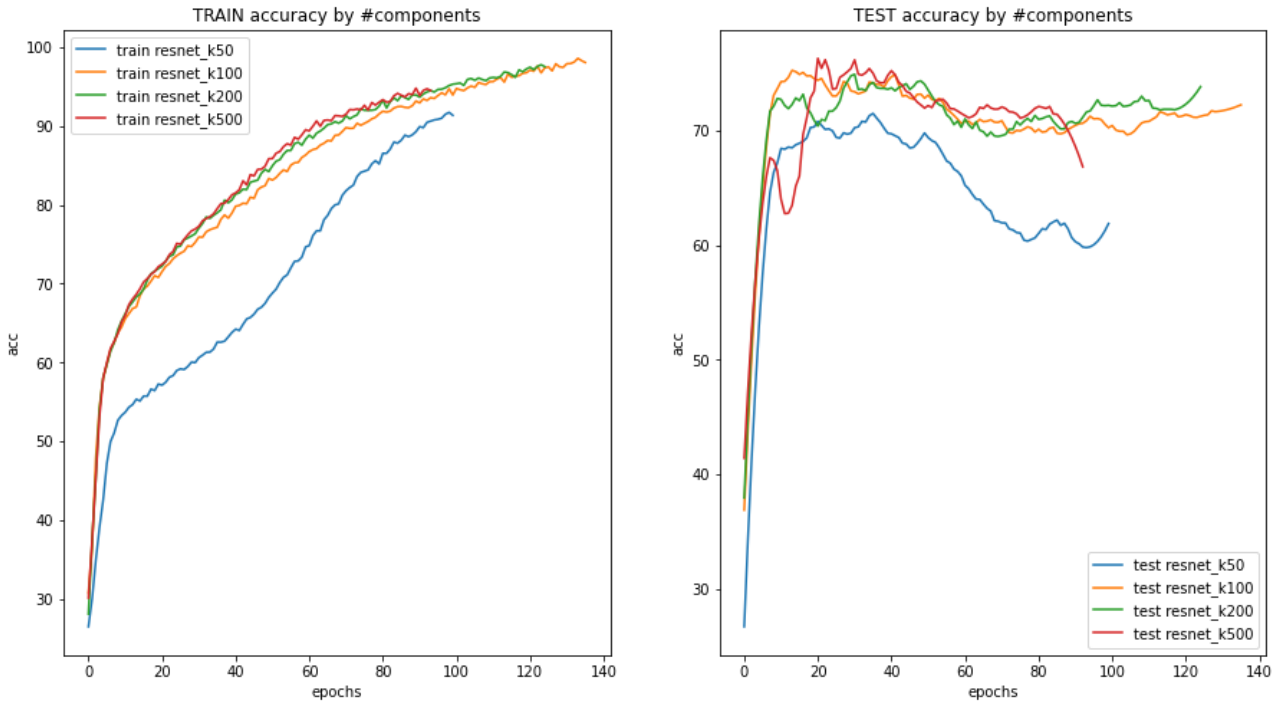
Accuracy plots before smoothing:

```
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(15,8))
for key in acc_train_dict.keys():
    axes[0].plot(acc_train_dict[key], label = f'train {key}')
    axes[0].set_xlabel('epochs')
    axes[0].set_ylabel('acc')
    axes[0].set_title('TRAIN accuracy by num_components')
    axes[0].legend()

for key in acc_test_dict.keys():
    axes[1].plot(acc_test_dict[key], label = f'test {key}')
    axes[1].set_xlabel('epochs')
    axes[1].set_ylabel('acc')
    axes[1].set_title('TEST accuracy by num_components')
    axes[1].legend()
```



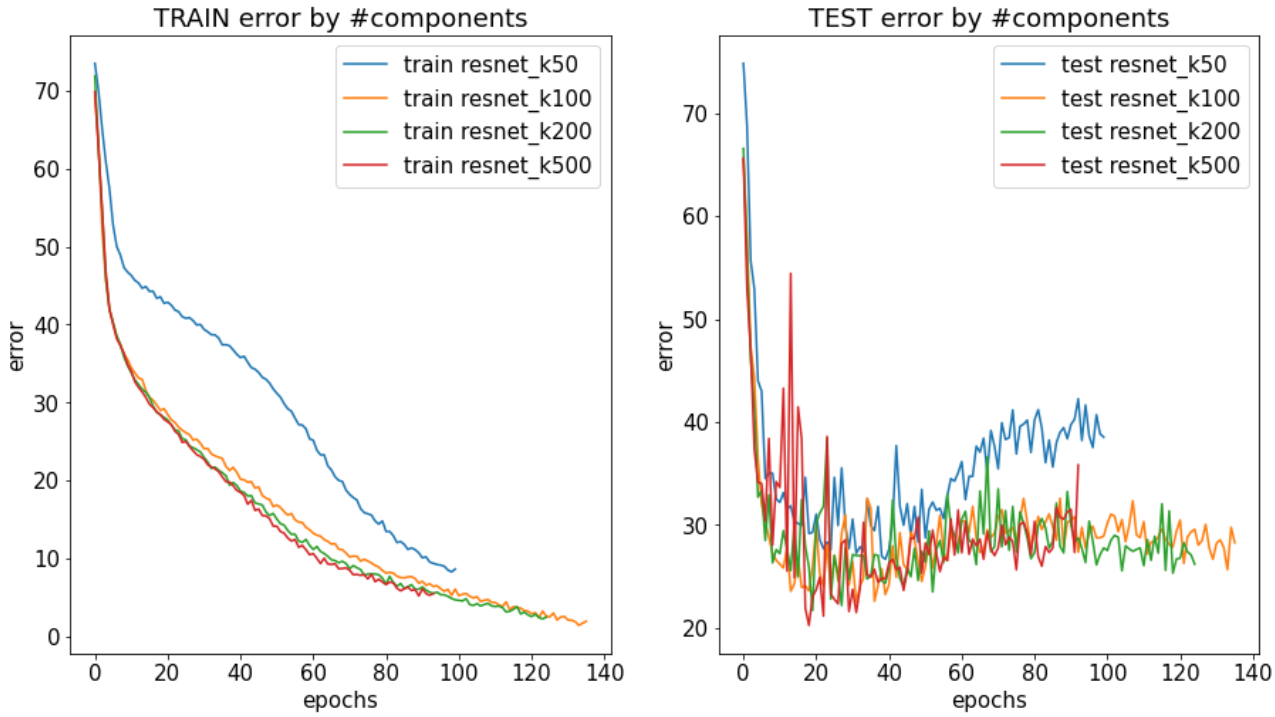
Accuracy plots after smoothing:

```
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(15,8))
for key in acc_train_dict.keys():
    axes[0].plot(acc_train_dict[key], label = f'train {key}')
    axes[0].set_xlabel('epochs')
    axes[0].set_ylabel('acc')
    axes[0].set_title('TRAIN accuracy by #components')
    axes[0].legend()

for key in acc_test_dict.keys():
    new_acc= savgol_filter(acc_test_dict[key], 15,2)
    axes[1].plot(new_acc, label = f'test {key}')
    axes[1].set_xlabel('epochs')
    axes[1].set_ylabel('acc')
    axes[1].set_title('TEST accuracy by #components')
    axes[1].legend()
```



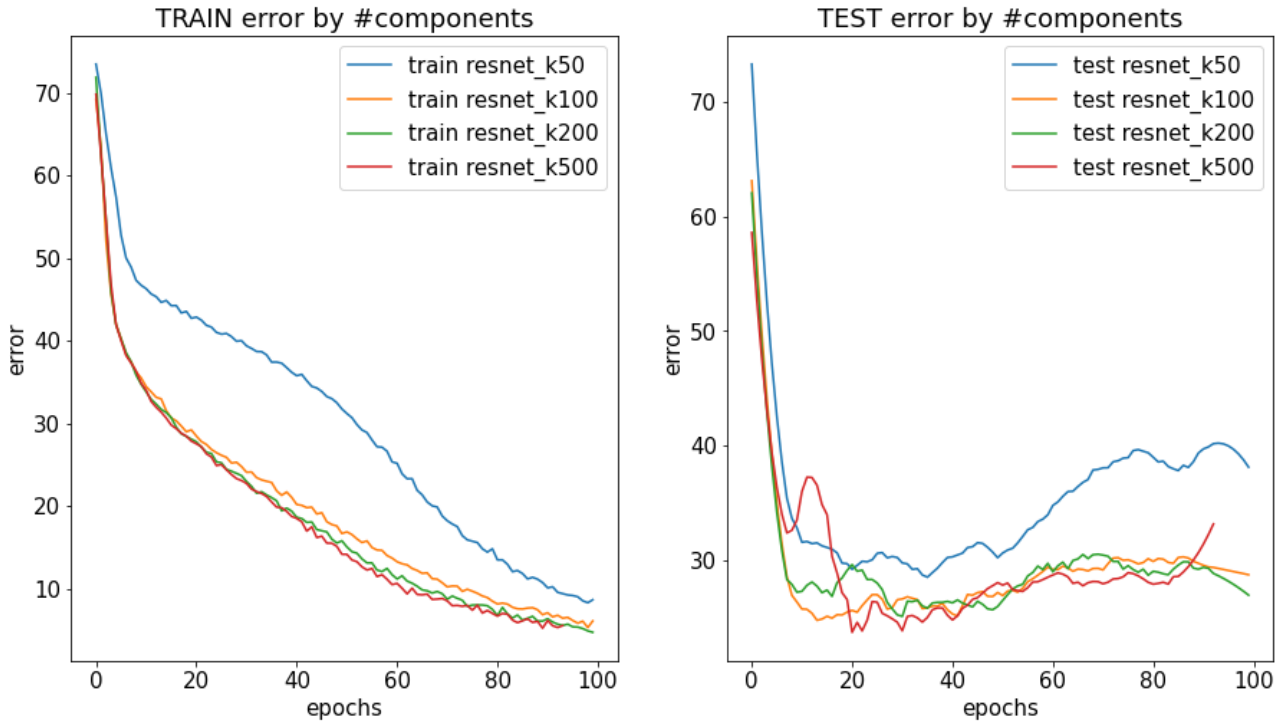
Error plots before smoothing:

```
#
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(15,8))
for key in acc_train_dict.keys():
    l = acc_train_dict[key]
    error_list= [100-i for i in l]
    axes[0].plot(error_list, label = f'train {key}')
    axes[0].set_xlabel('epochs')
    axes[0].set_ylabel('error')
    axes[0].set_title('TRAIN error by #components')
    axes[0].legend()

for key in acc_test_dict.keys():
    l= acc_test_dict[key]
    error_list= [100-i for i in l]
    axes[1].plot(error_list, label = f'test {key}')
    axes[1].set_xlabel('epochs')
    axes[1].set_ylabel('error')
    axes[1].set_title('TEST error by #components')
    axes[1].legend()
```



Error plots after smoothing:

```
from scipy.signal import savgol_filter

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(15,8))
for key in acc_train_dict.keys():
    l = acc_train_dict[key]
    error_list= [100-l[i] for i in range(min(len(l),100))]
    axes[0].plot(error_list, label = f'train {key}')
    axes[0].set_xlabel('epochs')
    axes[0].set_ylabel('error')
    axes[0].set_title('TRAIN error by #components')
    axes[0].legend()

for key in acc_test_dict.keys():
    l= acc_test_dict[key]
    error_list= [100-l[i] for i in range(min(len(l),100))]
    new_error_list= savgol_filter(error_list, 15,2)
    axes[1].plot(new_error_list, label = f'test {key}')
    axes[1].set_xlabel('epochs')
    axes[1].set_ylabel('error')
    axes[1].set_title('TEST error by #components')
    axes[1].legend()
```



Summary and Conclusions:

- As we mentioned before, we can see that in the test set of k=50 we can perform early stopping because we don't see the EDD phenomena - there is only one major descent.

- In k=500 that captures most of Data's variation we can see the EDD phenomena- we have first descent on epoch 10 and second one on epoch 20, Thus we can not perform early stopping on that Data set.
- In therms of "BEST DATA SET" we would take k=100. It doesn't experienced the EDD phenomena and also has very low error already on epoch~15 thus we can perform early stopping on that data set.

To sum it up, we can see that the PCA method for reducing EDD is indeed working well in practice.

Notes:

We have also tried other approaches using a smaller net and different datasets but they didn't show EDD properties, thus we didn't elaborate on their results.

Method 2 - Remove features by replacing the last layer's weights with converged weights

Set up:

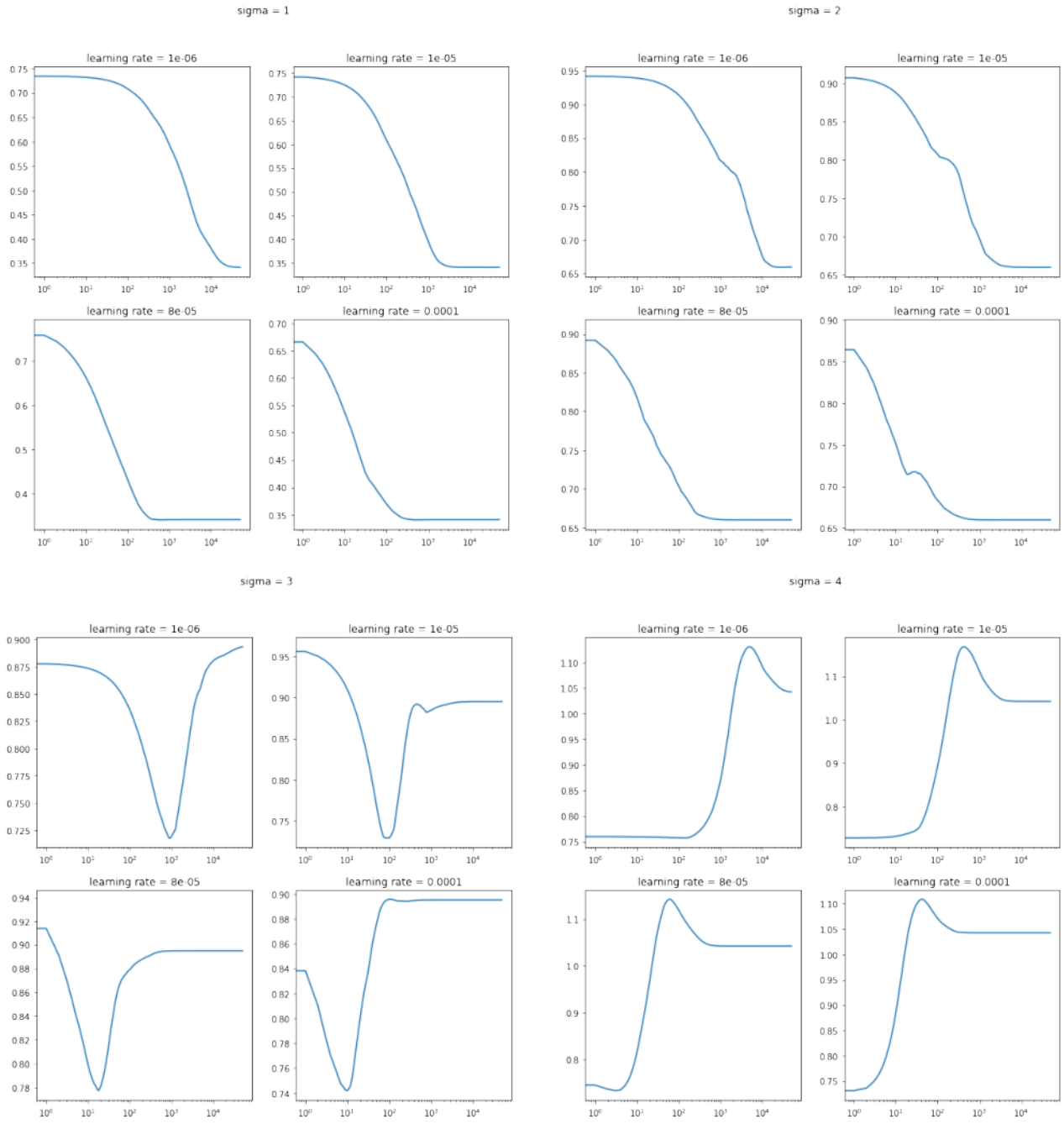
In the following section we generated different sets of gaussian data, with noise added to eigenvectors matching large eigenvalues (large= greater than 1, we added the noise only to the train set). We examine for which parameters of learning rate and sigma (std of noise) we see epoch-wise double descent, on a simple two layer network (as proposed by Heckel & Yilmaz (2020))

Parameters:

- N (= 100) size of test set and train set
- D(=50) features
- C(=1) classes
- n_dense_unit (=250)

We trained the network for T(=10) epochs. After that we replaced the weights of the last layer with corresponding converged weights.

```
fig, axes = plt.subplots(nrows = 2, ncols = 2, figsize= (15, 15))
plt.rcParams['font.size'] = '15'
sigma_list = [1, 2, 3, 4]
for s, ax in zip(sigma_list, axes.flatten()) :
    ax.imshow(mpimg.imread(f'sigma={s} plot.png'))
    # ax.set_title(f'Images projected into k = {k}')
    ax.set_axis_off()
plt.tight_layout()
```



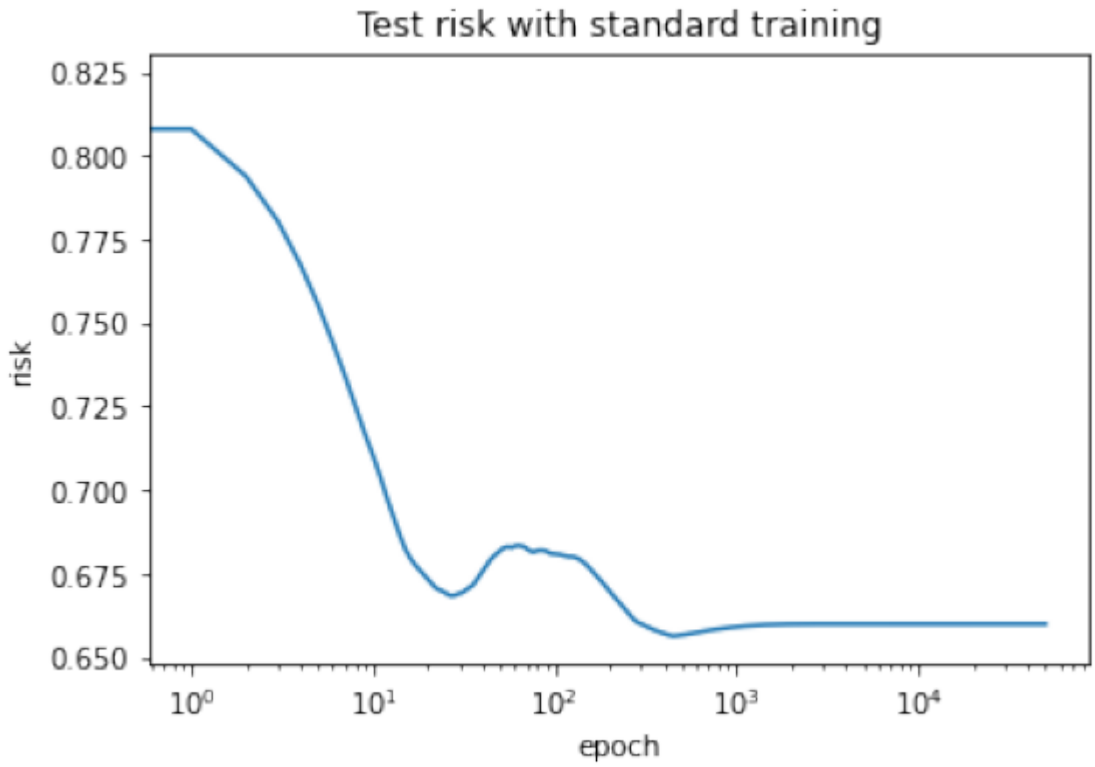
Results summary:

1. We can see that for small sigma - there is no EDD
2. We can see that according to the paper contributions, there is a critical level of noise (sigma), where EDD occurs.
   - In our case it is for sigma = 2.
3. As been mentioned in the paper we can see that there is a critical amount of noise needed for EDD to occur.

After seeing the results we decided to use learning rate = 0.00005 and sigma=2

```
plt.figure(figsize=(10,10))
plt.axis('off')
plt.imshow(mpimg.imread('sigma=2 lr=0.00005.png'))
```

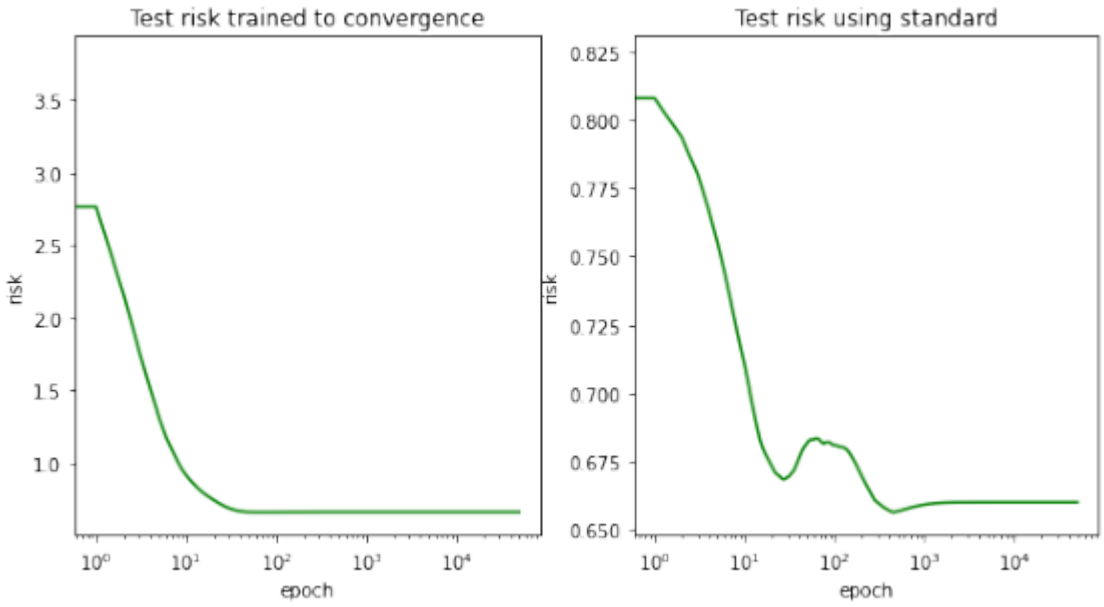
```
<matplotlib.image.AxesImage at 0x7fee96231490>
```



In the above figure we can see clearly the EDD phenomena, thus we chose those parameters to prove that after replacing the weights the EDD will disappear.

```
plt.figure(figsize=(10,10))
plt.axis('off')
plt.imshow(mpimg.imread('converged vs standard.png'))
```

```
<matplotlib.image.AxesImage at 0x7fee96bfb460>
```



Summary and Conclusions:

1. We saw that EDD occurs as function of a few hyperparametrs:

- learning rate
- amount of noise
- training time

   There is an obvious connection between all of the above.

2. We saw that using converged weights and training the network further from the point where standard training stopped leads to elimination of EDD phenomena, very similar to our papers' main results.

    - Because the scale may be miss leading, we shall note that the loss after replacing the weights didn't damaged.
    - We can see that using the converged weights we can perform early stopping before 100 epochs and in the standard case we need to wait ~1000 epochs, almost 10 times more.

3. We saw that the results above are not reproducible in every run, in contrast to what the paper suggested.

4. Using the theoretical equations on real life data (such as CIFAR10), may result in similar conclusions.

Notes:

- In the implementation (you can see notebook of method2) we implemented 2 ways to calculate the converged weights:
  - One way is relevant to MSE loss - regression problem (that's the way we used to produce our results)
  - The other way fits Cross Entropy loss - classification problem (will be relevant for CIFAR10 for example)