Homework Assignment 3: Unsupervised learning and intro to NN

Avinash Pawar

Question 1:

In [1]:
```
%%capture

# Before starting of the program below code ensures that you have all the necessary libraries installed
# if not, it installs them.

!pip install numpy pandas seaborn scikit-learn opendatasets tensorflow
```

In [2]:
```
# Importing all the required libraries
import sys
import time
import math
import numpy as np
import pandas as pd
from pathlib import Path
import sklearn
import pickle
import seaborn as sns
import matplotlib as mpl
from pathlib import Path
import opendatasets as od
import tensorflow as tf
```

```
2023-04-01 02:50:28.276081: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find cuda drivers on your machine, GPU will not b
e used.
2023-04-01 02:50:28.302158: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find cuda drivers on your machine, GPU will not b
e used.
2023-04-01 02:50:28.302677: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use avai
lable CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
2023-04-01 02:50:28.869558: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not find TensorRT
```

In [3]:
```
tf.sysconfig.get_build_info() ##Check Tensorflow
```

Out[3]:
```
OrderedDict([('cpu_compiler', '/dt9/usr/bin/gcc'),
             ('cuda_compute_capabilities',
              ['sm_35', 'sm_50', 'sm_60', 'sm_70', 'sm_75', 'compute_80']),
             ('cuda_version', '11.8'),
             ('cudnn_version', '8'),
             ('is_cuda_build', True),
             ('is_rocm_build', False),
             ('is_tensorrt_build', True)])
```

In [4]:
```
# Declaring image properties
import matplotlib.pyplot as plt

plt.rc('font', size=12)
plt.rc('axes', labelsize=14, titlesize=14)
plt.rc('legend', fontsize=12)
plt.rc('xtick', labelsize=10)
plt.rc('ytick', labelsize=10)
```

In [5]:
```
# importing warnings package and filtering the warnings
import warnings
warnings.filterwarnings('ignore')
```

In [6]:
```
# setting random seed for entire program
np.random.seed(22)
```

In [7]:
```
# Reading the data
data = pd.read_csv('data/sign_mnist_train.csv')
test_data = pd.read_csv('data/sign_mnist_test.csv')
```

In [8]:
```
# Extracting lables from the data and dropping label column from the data
# for better testing -> data = data.iloc[0:12000, ]
labels = data['label']
test_labels = test_data['label']
```

In [9]:
```
data_labels = labels
```

In [10]:
```
# Dropping label column from the datasets
data.drop(columns=['label'], axis = 1, inplace = True)
test_data.drop(columns=['label'], axis = 1, inplace = True)
```

In [11]:
```
from sklearn.model_selection import StratifiedShuffleSplit
```

In [12]:
```
# Split the dataset into training and validation data sets
split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=22)
```

```
training_index, val_index = next(split.split(data, labels))

training_data = data.loc[training_index]
training_labels = labels.loc[training_index]

validation_data = data.loc[val_index]
validation_labels = labels.loc[val_index]
```

In [13]:
```
training_data.shape, training_labels.shape , validation_data.shape, validation_labels.shape
```

Out[13]:
```
((21964, 784), (21964,), (5491, 784), (5491,))
```

In [14]:
```
test_data.shape, test_labels.shape
```

Out[14]:
```
((7172, 784), (7172,))
```

Question : Apply PCA to the training portion of the dataset. How many components do you need to preserve 95% of the variance?

In [15]:
```
from sklearn.decomposition import PCA
```

In [16]:
```
pca = PCA(n_components=0.95)
pca.fit(training_data)
```

Out[16]:
```
PCA(n_components=0.95)
```

In [17]:
```
# Transform training and testing data using PCA
pca_training_data = pd.DataFrame(pca.transform(training_data))
pca_validation_data = pd.DataFrame(pca.transform(validation_data))
```

In [18]:
```
pca_training_data.shape, training_labels.shape , pca_validation_data.shape, validation_labels.shape
```

Out[18]:
```
((21964, 113), (21964,), (5491, 113), (5491,))
```

Answer :

After implementing PCA to the training data, we need to keep 113 components in order to preserve 95% of varience.

Question : Train a Random Forest classifier on the reduced dataset. Was training much faster than in Homework 2? Evaluate the classifier on the test set. How does it compare to the classifier from Homework 2?

Implementing classifier from Homework 2 again

1. Random Forest Classifier (without PCA)

In [19]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.pipeline import make_pipeline
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, confusion_matrix
```

In [20]:
```
estimators = np.arange(50, 780, 50)
depths = np.arange(5, 55, 5)
l_e, l_d = estimators.size, depths.size
```

In [21]:
```
parameters = {
    "training_accuracy" : np.zeros((l_e, l_d)),
    "validation_accuracy" : np.zeros((l_e, l_d)),
    "training_precision" : np.zeros((l_e, l_d)),
    "validation_precision" : np.zeros((l_e, l_d)),
    "training_recall" : np.zeros((l_e, l_d)),
    "validation_recall" : np.zeros((l_e, l_d)),
    "training_f1_score" : np.zeros((l_e, l_d)),
    "validation_f1_score" : np.zeros((l_e, l_d)),
    "best_accuracy" : -math.inf,
    "best_parameters" : {'n_estimators': -1, 'max_depth': -1},
}
```

In [22]:
```
def calculate_metrics(pred_tlabels, training_labels, pred_vlabels,
                      validation_labels, row, column, estimator, depth, parameters):
    # calculating scores on training dataset
    training_precision = precision_score(training_labels, pred_tlabels, average='weighted')
    training_accuracy = accuracy_score(training_labels, pred_tlabels)
    training_recall = recall_score(training_labels, pred_tlabels, average='weighted')
    training_f1_score = f1_score(training_labels, pred_tlabels, average='weighted')

    # calculating scores on validation dataset
    validation_precision = precision_score(validation_labels, pred_vlabels, average='weighted')
    validation_accuracy = accuracy_score(validation_labels, pred_vlabels)
    validation_recall = recall_score(validation_labels, pred_vlabels, average='weighted')
    validation_f1_score = f1_score(validation_labels, pred_vlabels, average='weighted')

    parameters["training_accuracy"][row][column] = training_accuracy
    parameters["validation_accuracy"][row][column] = validation_accuracy
    parameters["training_precision"][row][column] = training_precision
    parameters["validation_precision"][row][column] = validation_precision
```

```
    parameters["training_recall"][row][column] = training_recall
    parameters["validation_recall"][row][column] = validation_recall
    parameters["training_f1_score"][row][column] = training_f1_score
    parameters["validation_f1_score"][row][column] = validation_f1_score

    # Keep track of best hyperparameters and corresponding validation accuracy
    if validation_accuracy > parameters["best_accuracy"]:
        parameters["best_accuracy"] = validation_accuracy
        parameters["best_parameters"] = {'n_estimators': estimator, 'max_depth': depth}
    return parameters
```

In [23]:
```
def save_dict(file_name, dictonary ):
    with open(file_name, 'wb') as fp:
        pickle.dump(dictonary, fp)

def load_dict(file_name):
    with open(file_name, 'rb') as fp:
        dictonary = pickle.load(fp)
    return dictonary
```

In [24]:
```
%%time
file_name = "random_forest_without_pca.pkl"
if Path(file_name).is_file():
    parameters = load_dict(file_name)
else:
    for index_i, estimator in enumerate(estimators):
        for index_j, depth in enumerate(depths):
            # Declaring a model
            rfc = RandomForestClassifier(n_estimators=estimator, max_depth=depth, n_jobs= -1)

            # Fit model on training data
            rfc.fit(training_data, training_labels)

            # Predicting training labels based on trainig data
            pred_tlabels = rfc.predict(training_data)

            # Predicting validation labels based on validation data
            pred_vlabels = rfc.predict(validation_data)

            parameters = calculate_metrics(pred_tlabels, training_labels, pred_vlabels,
                                           validation_labels, index_i, index_j, estimator, depth, parameters)
    save_dict(file_name, parameters)
```

```
CPU times: user 3.46 ms, sys: 197 µs, total: 3.66 ms
Wall time: 437 µs
```

In [25]:
```
print("Best accuracy obtained : ", parameters['best_accuracy'])
print("Optimum value of estimators obtained : ", parameters['best_parameters']['n_estimators'])
print("Optimum value of depth obtained : ", parameters['best_parameters']['max_depth'])
```

```
Best accuracy obtained :  0.9983609542888363
Optimum value of estimators obtained :  550
Optimum value of depth obtained :  45
```

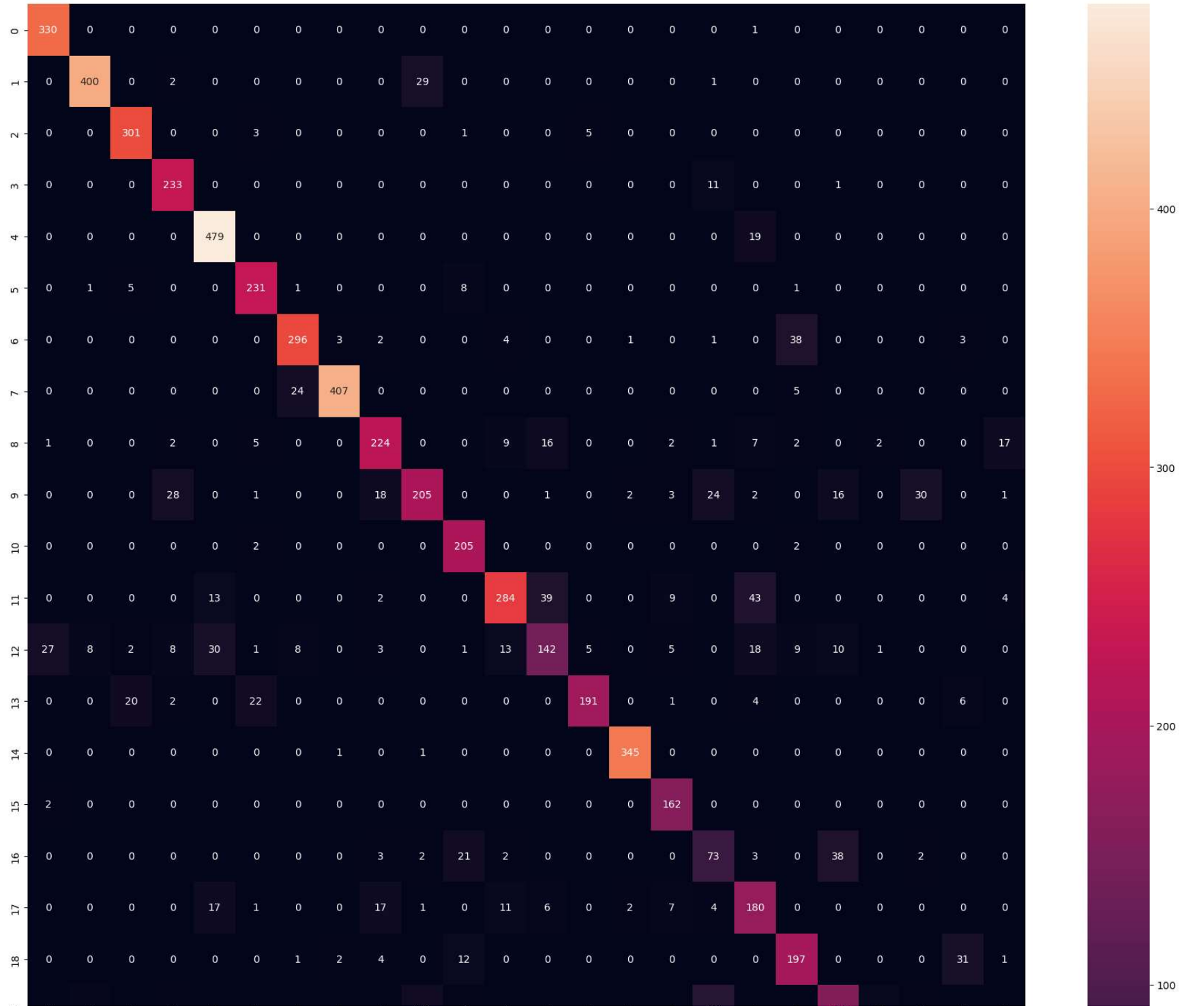
In [26]:
```
# Creating model with best hyperparameters for executing it on test data
rfc_model = RandomForestClassifier(n_estimators=parameters['best_parameters']['n_estimators'], max_depth=parameters['best_parame
```

In [27]:
```
%%time
# save the model after calling .fit method or retriving already saved model
filename = 'rfc_model.sav'

if Path(filename).is_file():
    rfc_model = pickle.load(open(filename, 'rb'))
else:
    rfc_model.fit(training_data, training_labels)
    pickle.dump(rfc_model, open(filename, 'wb'))
```

```
CPU times: user 167 ms, sys: 10.9 ms, total: 178 ms
Wall time: 22.1 ms
```

In [28]:
```
# Predicting test data labels based on trained model
test_labels_pred = rfc_model.predict(test_data)
```

In [29]:
```
# plotting confusion matrix
plt.figure(figsize=(22,22))
plt.rc('font', size=10)
cm = confusion_matrix(test_labels, test_labels_pred)
sns.heatmap(cm, annot=True, fmt='d')
plt.show()
```


0 330 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
1 0 400 0 2 0 0 0 0 0 29 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
2 0 0 301 0 0 3 0 0 0 0 1 0 0 5 0 0 0 0 0 0 0 0 0 0 0
3 0 0 0 233 0 0 0 0 0 0 0 0 0 0 0 0 0 11 0 0 1 0 0 0 0
4 0 0 0 0 479 0 0 0 0 0 0 0 0 0 0 0 0 0 19 0 0 0 0 0 0
5 0 1 5 0 0 231 1 0 0 0 8 0 0 0 0 0 0 0 0 1 0 0 0 0 0
6 0 0 0 0 0 0 296 3 2 0 0 4 0 0 1 0 1 0 38 0 0 0 3 0
7 0 0 0 0 0 0 24 407 0 0 0 0 0 0 0 0 0 0 0 5 0 0 0 0 0
8 1 0 0 2 0 5 0 0 224 0 0 9 16 0 0 2 1 7 2 0 2 0 0 17
9 0 0 0 28 0 1 0 0 18 205 0 0 1 0 2 3 24 2 0 16 0 30 0 1
10 0 0 0 0 0 2 0 0 0 0 205 0 0 0 0 0 0 0 2 0 0 0 0 0
11 0 0 0 0 13 0 0 0 2 0 0 284 39 0 0 9 0 43 0 0 0 0 0 4
12 27 8 2 8 30 1 8 0 3 0 1 13 142 5 0 5 0 18 9 10 1 0 0 0
13 0 0 20 2 0 22 0 0 0 0 0 0 0 191 0 1 0 4 0 0 0 0 6 0
14 0 0 0 0 0 0 0 1 0 1 0 0 0 0 345 0 0 0 0 0 0 0 0 0
15 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 162 0 0 0 0 0 0 0 0
16 0 0 0 0 0 0 0 0 3 2 21 2 0 0 0 0 73 3 0 38 0 2 0 0
17 0 0 0 0 17 1 0 0 17 1 0 11 6 0 2 7 4 180 0 0 0 0 0 0
18 0 0 0 0 0 0 1 2 4 0 12 0 0 0 0 0 0 0 197 0 0 0 31 1
400
300
200
100

In [30]:
```
def print_test_metrics(test_labels, test_labels_pred):
    print(f"Precision Score :  {precision_score(test_labels, test_labels_pred, average='weighted') :.4f} ")
    print(f"Accuracy Score \t:  {accuracy_score(test_labels, test_labels_pred) :.4f} ")
    print(f"Recall Score \t:  {recall_score(test_labels, test_labels_pred, average='weighted') :.4f} ")
    print(f"F1 Score \t:  {f1_score(test_labels, test_labels_pred, average='weighted') :.4f} ")
```

In [31]:
```
print_test_metrics(test_labels, test_labels_pred)
```

```
Precision Score :  0.8137
Accuracy Score  :  0.8001
Recall Score    :  0.8001
F1 Score        :  0.8002
```

2. Random Forest Classifier (with PCA)

In [32]:
```
estimators = np.arange(50, 180, 20)
depths = np.arange(5, 55, 5)
l_e, l_d = estimators.size, depths.size
```

In [33]:
```
parameters_pca = {
    "training_accuracy" : np.zeros((l_e, l_d)),
    "validation_accuracy" : np.zeros((l_e, l_d)),
    "training_precision" : np.zeros((l_e, l_d)),
    "validation_precision" : np.zeros((l_e, l_d)),
    "training_recall" : np.zeros((l_e, l_d)),
    "validation_recall" : np.zeros((l_e, l_d)),
    "training_f1_score" : np.zeros((l_e, l_d)),
    "validation_f1_score" : np.zeros((l_e, l_d)),
    "best_accuracy" : -math.inf,
    "best_parameters" : {'n_estimators': -1, 'max_depth': -1}
}
```

In [34]:
```
%%time
file_name = "random_forest_with_pca.pkl"
if Path(file_name).is_file():
    parameters_pca = load_dict(file_name)
else:
    for index_i, estimator in enumerate(estimators):
        for index_j, depth in enumerate(depths):
            # Declaring a model
            rfc = RandomForestClassifier(n_estimators=estimator, max_depth=depth, n_jobs= -1)

            # Fit model on training data
            rfc.fit(pca_training_data, training_labels)

            # Predicting training labels based on trainig data
            pred_tlabels = rfc.predict(pca_training_data)

            # Predicting validation labels based on validation data
            pred_vlabels = rfc.predict(pca_validation_data)

            parameters_pca = calculate_metrics(pred_tlabels, training_labels, pred_vlabels,
                                               validation_labels, index_i, index_j, estimator, depth, parameters_pca)
    save_dict(file_name, parameters_pca)
```

```
CPU times: user 359 µs, sys: 19 µs, total: 378 µs
Wall time: 307 µs
```

In [35]:
```
print("Best accuracy obtained : ", parameters_pca['best_accuracy'])
print("Optimum value of estimators obtained : ", parameters_pca['best_parameters']['n_estimators'])
print("Optimum value of depth obtained : ", parameters_pca['best_parameters']['max_depth'])
```

```
Best accuracy obtained :  1.0
Optimum value of estimators obtained :  50
Optimum value of depth obtained :  15
```

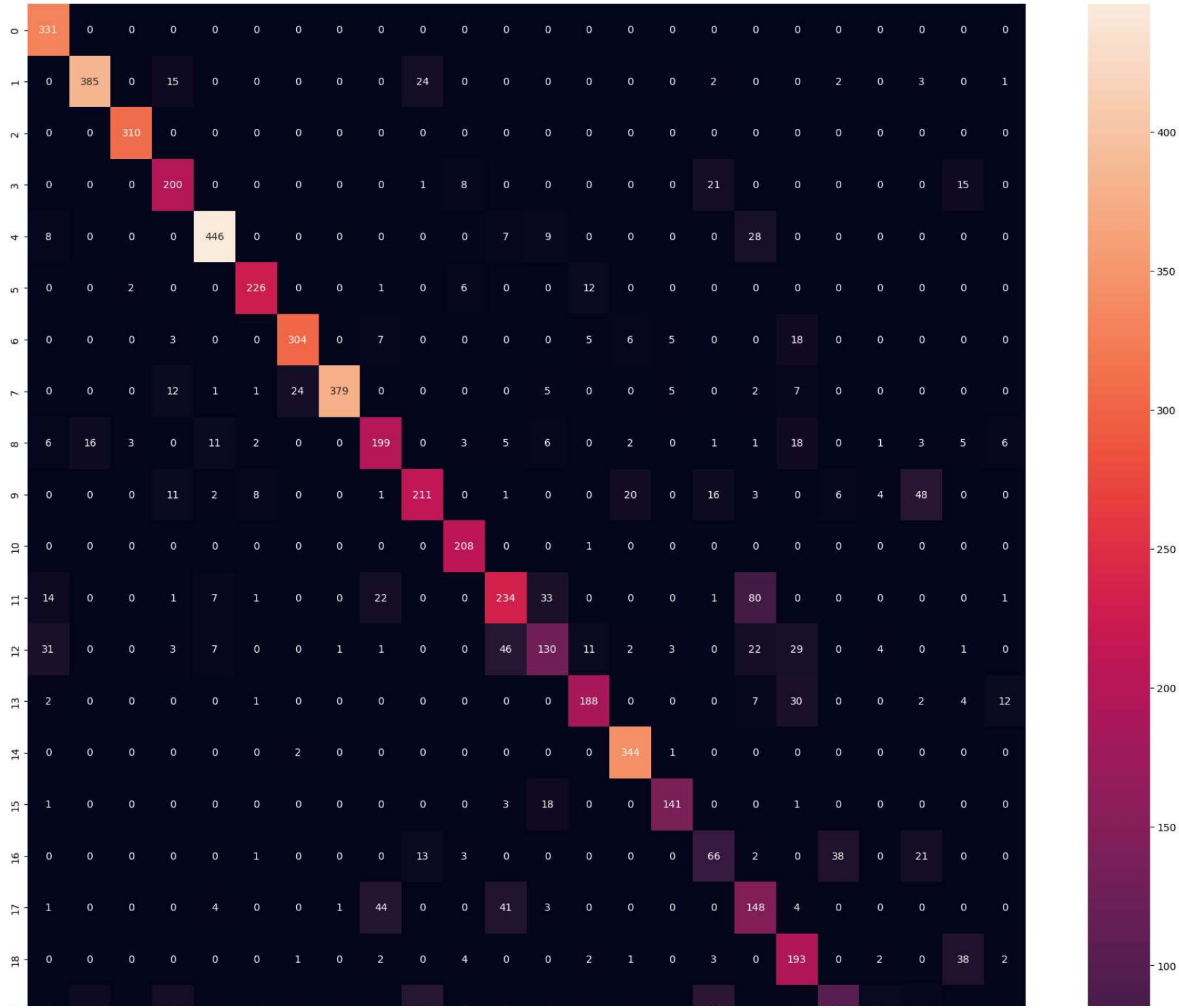
In [36]:
```
# Creating model with best hyperparameters for executing it on test data
rfc_model_pca = RandomForestClassifier(n_estimators=parameters_pca['best_parameters']['n_estimators'], max_depth=parameters_pca[
```

In [37]:
```
%%time
# save the model after calling .fit method or retriving already saved model
filename = 'rfc_model_pca.sav'

if Path(filename).is_file():
    rfc_model_pca = pickle.load(open(filename, 'rb'))
else:
    rfc_model_pca.fit(pca_training_data, training_labels)
    pickle.dump(rfc_model_pca, open(filename, 'wb'))
```

```
CPU times: user 10.5 ms, sys: 28 µs, total: 10.5 ms
Wall time: 10.2 ms
```

In [38]:
```
# Predicting test data labels based on trained model
pca_test_data = pd.DataFrame(pca.transform(test_data))
test_labels_pred = rfc_model_pca.predict(pca_test_data)
```

In [39]:
```
# plotting confusion matrix
plt.figure(figsize=(22,22))
plt.rc('font', size=10)
cm = confusion_matrix(test_labels, test_labels_pred)
sns.heatmap(cm, annot=True, fmt='d')
plt.show()
```


0 331 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 385 0 15 0 0 0 0 0 24 0 0 0 0 0 0 0 2 0 0 2 0 3 0 1
2 0 0 310 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3 0 0 0 200 0 0 0 0 0 1 8 0 0 0 0 0 0 21 0 0 0 0 0 15 0
4 8 0 0 0 446 0 0 0 0 0 0 7 9 0 0 0 0 0 28 0 0 0 0 0 0
5 0 0 2 0 0 226 0 0 1 0 6 0 0 12 0 0 0 0 0 0 0 0 0 0 0
6 0 0 0 3 0 0 304 0 7 0 0 0 0 5 6 5 0 0 18 0 0 0 0 0 0
7 0 0 0 12 1 1 24 379 0 0 0 0 5 0 0 5 0 2 7 0 0 0 0 0 0
8 6 16 3 0 11 2 0 0 199 0 3 5 6 0 2 0 1 1 18 0 1 3 5 6
9 0 0 0 11 2 8 0 0 1 211 0 1 0 0 20 0 16 3 0 6 4 48 0 0
10 0 0 0 0 0 0 0 0 0 0 208 0 0 1 0 0 0 0 0 0 0 0 0 0 0
11 14 0 0 1 7 1 0 0 22 0 0 234 33 0 0 0 1 80 0 0 0 0 0 1
12 31 0 0 3 7 0 0 1 1 0 0 46 130 11 2 3 0 22 29 0 4 0 1 0
13 2 0 0 0 0 1 0 0 0 0 0 0 0 188 0 0 0 7 30 0 0 2 4 12
14 0 0 0 0 0 0 2 0 0 0 0 0 0 0 344 1 0 0 0 0 0 0 0 0 0
15 1 0 0 0 0 0 0 0 0 0 0 3 18 0 0 141 0 0 1 0 0 0 0 0
16 0 0 0 0 0 1 0 0 0 13 3 0 0 0 0 0 66 2 0 38 0 21 0 0
17 1 0 0 0 4 0 0 1 44 0 0 41 3 0 0 0 0 148 4 0 0 0 0 0
18 0 0 0 0 0 0 1 0 2 0 4 0 0 2 1 0 3 0 193 0 2 0 38 2
400
350
300
250
200
150
100

In [40]:
```
print_test_metrics(test_labels, test_labels_pred)
```

```
Precision Score :  0.7836
Accuracy Score  :  0.7691
Recall Score    :  0.7691
F1 Score        :  0.7704
```

It appears that for same number of trees and with same depth, the random forest with PCA takes relatively less time then the random forest without PCA. This is because the dimentionality is reduced and thus the complexity in the data is also reduced.

If we look at the accuracy produced by both models, there is no huge gap in between. So, reducing dimentionality to save time makes sense here.

Question: Plot 10 random images in the original form (without PCA) and then plot them after you kept 95% of variance using PCA.

In [41]:
```
# function to plot digits into an image format
def plot_data(image_data):
    image = image_data.reshape(28, 28)
    plt.imshow(image, cmap="binary")
    plt.axis("off")
```

Random 10 samples from Training data before implementing PCA

In [42]:
```
random_sample_indexes = training_data.sample(n=10, random_state=22).index
```

In [43]:
```
plt.figure(figsize=(10, 4))
for index_i, image_index in enumerate(random_sample_indexes):
    plt.subplot(2, 5, index_i + 1)
    plot_data(training_data.loc[image_index].to_numpy())
```

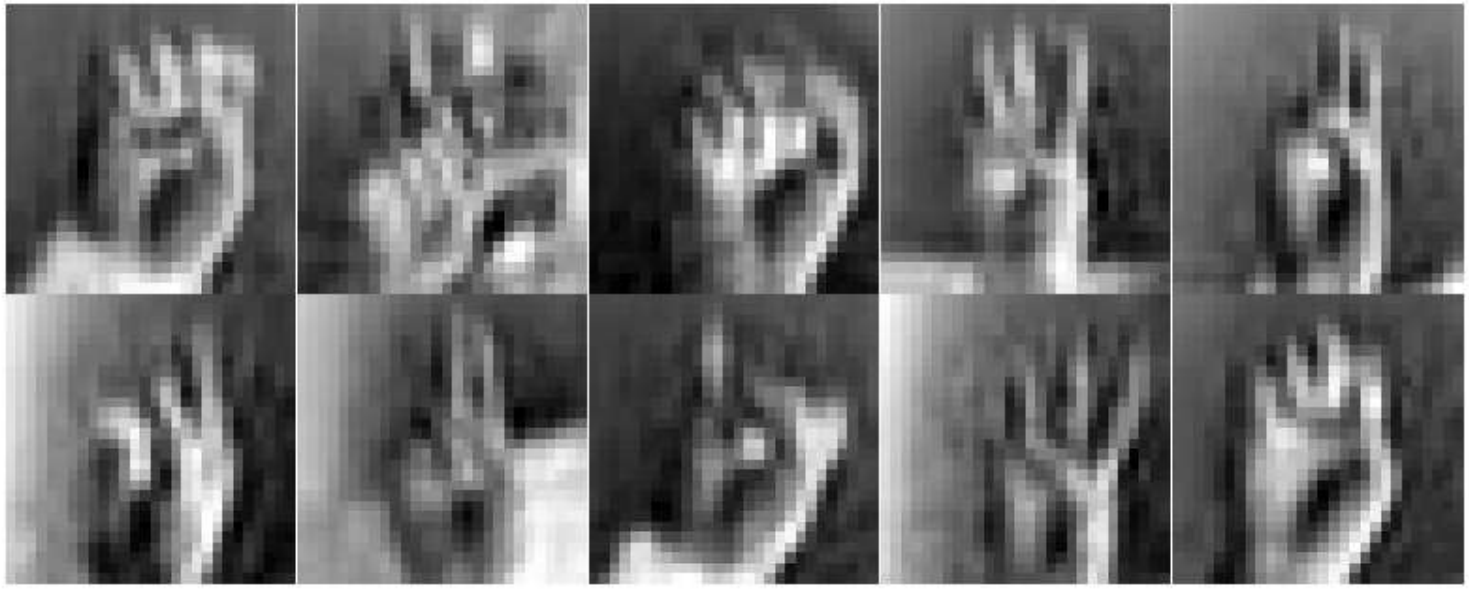
```
plt.subplots_adjust(wspace=0, hspace=0)
plt.show()
```

Random 10 samples from Training data After implementing PCA

In [44]:
```
training_data_recovered = pd.DataFrame(pca.inverse_transform(pca_training_data))
```

In [45]:
```
random_sample_indexes = training_data_recovered.sample(n=10, random_state=22).index
```

In [46]:
```
plt.figure(figsize=(10, 4))
for index_i, image_index in enumerate(random_sample_indexes):
    plt.subplot(2, 5, index_i + 1)
    plot_data(training_data_recovered.loc[image_index].to_numpy())
plt.subplots_adjust(wspace=0, hspace=0)
plt.show()
```



Question: How much of the variance is explained with the first two principal components?

In [47]:
```
print(f"The first component explains {pca.explained_variance_ratio_[0] * 100:.4f} % variance")
```

```
The first component explains 31.0377 % variance
```

In [48]:
```
print(f"The Second component explains {pca.explained_variance_ratio_[1] * 100:.4f} % variance")
```

```
The Second component explains 9.5145 % variance
```

Question:

Use PCA to reduce dimensionality to only 2 dimensions. Plot 1000 random images from the training set in the 2D space spanned by the first two principal components. Use a scatterplot with 10 different colors to represent each image's target class. Repeat the process and create the same type of plots for t-SNE, LLE and MDS. [6 points] Which of the visualizations do you prefer and why? [1 point]

In [49]:
```
from sklearn.preprocessing import MinMaxScaler
from matplotlib.offsetbox import AnnotationBbox, OffsetImage

def plot_digits(X, y, min_distance=0.04, images=None, figsize=(13, 10)):
    # Let's scale the input features so that they range from 0 to 1
    X_normalized = MinMaxScaler().fit_transform(X)
    # Now we create the list of coordinates of the digits plotted so far.
    # We pretend that one is already plotted far away at the start, to
```

```
    # avoid `if` statements in the loop below
    neighbors = np.array([[10., 10.]])
    # The rest should be self-explanatory
    plt.figure(figsize=figsize)
    cmap = plt.cm.jet
    digits = np.unique(y)
    for digit in digits:
        plt.scatter(X_normalized[y == digit, 0], X_normalized[y == digit, 1],
                    c=[cmap(float(digit) / 24)], alpha=0.5)
    plt.axis("off")
    ax = plt.gca()  # get current axes
    for index, image_coord in enumerate(X_normalized):
        closest_distance = np.linalg.norm(neighbors - image_coord, axis=1).min()
        if closest_distance > min_distance:
            neighbors = np.r_[neighbors, [image_coord]]
            if images is None:
                plt.text(image_coord[0], image_coord[1], str(int(y[index])),
                         color=cmap(float(y[index]) / 24),
                         fontdict={"weight": "bold", "size": 16})
            else:
                image = images[index].reshape(28, 28)
                imagebox = AnnotationBbox(OffsetImage(image, cmap="binary"),
                                          image_coord)
                ax.add_artist(imagebox)
```

In [50]:
```
# Plot the first two principal components
def plot_principal_components(data, labels, figsize=(13, 10)):
    plt.figure(figsize=figsize)
    for i in range(25):
        plt.scatter(data[labels == i][:, 0], data[labels == i][:, 1], label=str(i), cmap="jet", alpha=0.5)
    plt.axis('off')
    plt.colorbar()
    plt.legend()
    plt.show()
```

1. PCA

In [51]:
```
X = training_data
y = training_labels
```

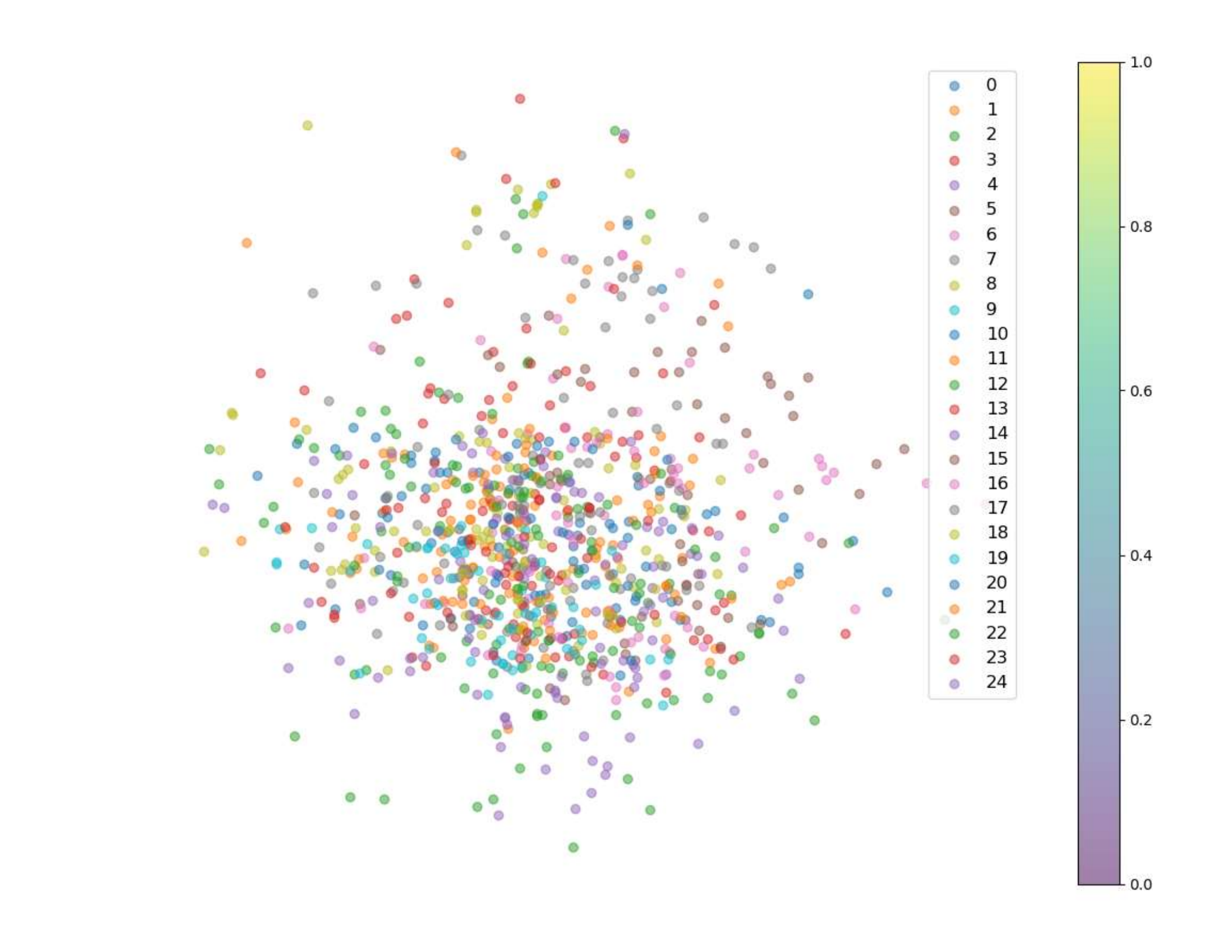
In [52]:
```
X.shape, y.shape
```

Out[52]:
```
((21964, 784), (21964,))
```

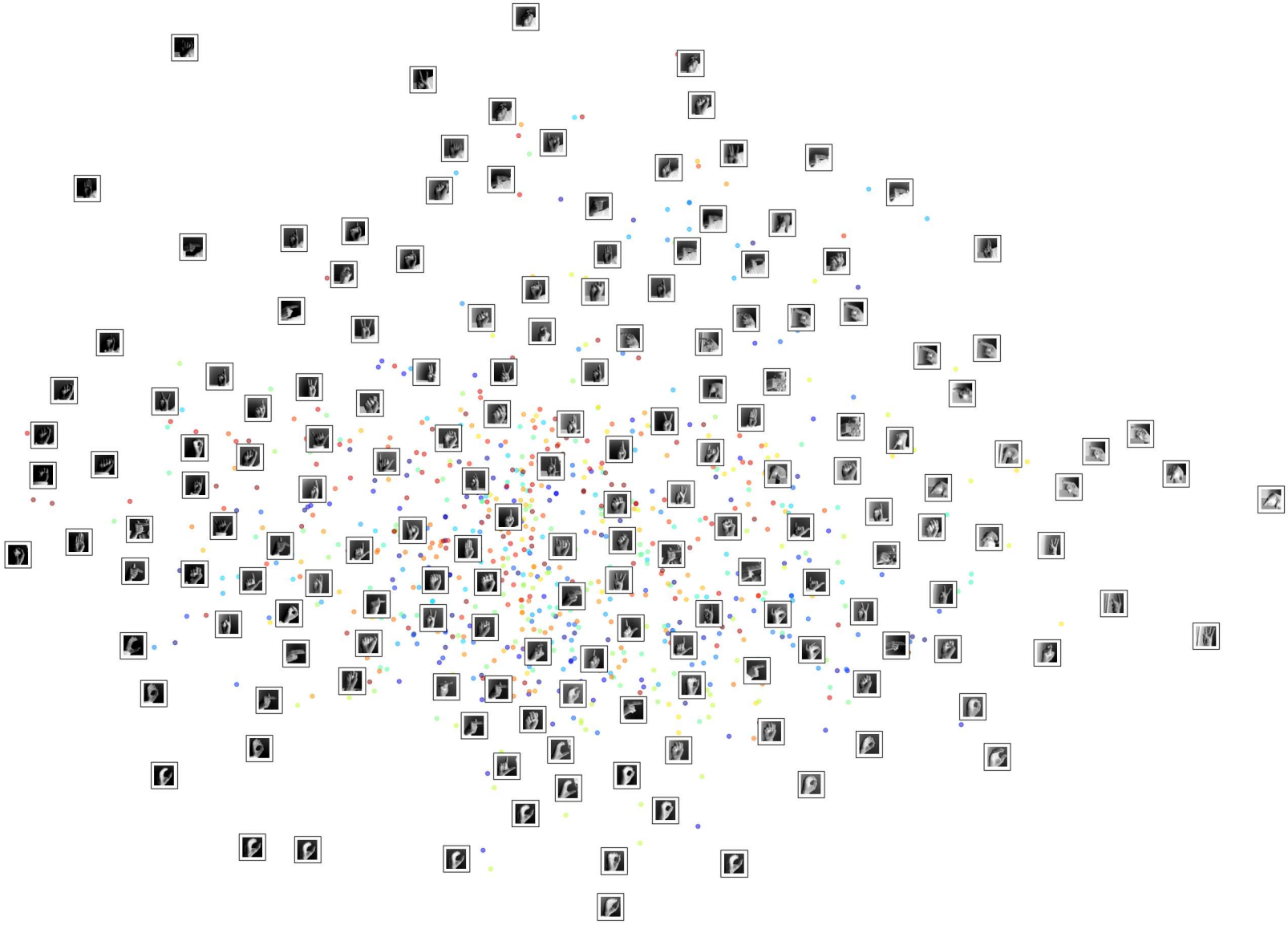
In [53]:
```
pca = PCA(n_components=2, random_state=22)
```

In [54]:
```
X_pca_reduced = pca.fit_transform(X)
```

In [55]:
```
plot_principal_components(X_pca_reduced[:1000], y[:1000])
```


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
1.0
0.8
0.6
0.4
0.2
0.0

In [56]:
```
plot_digits(X_pca_reduced[:1000], y[:1000], images=X.values[:1000], figsize=(35, 25))
```

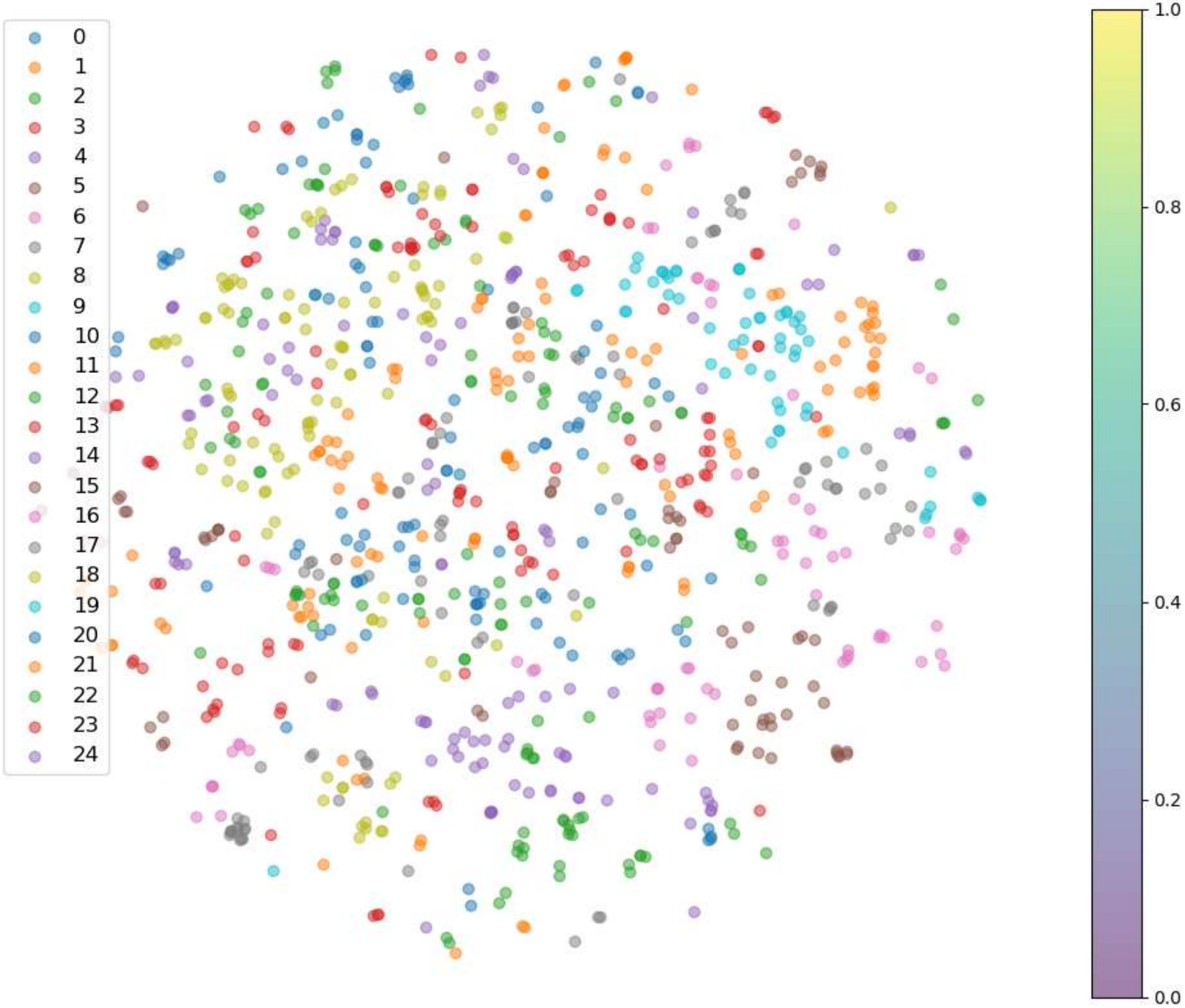


2. TSNE

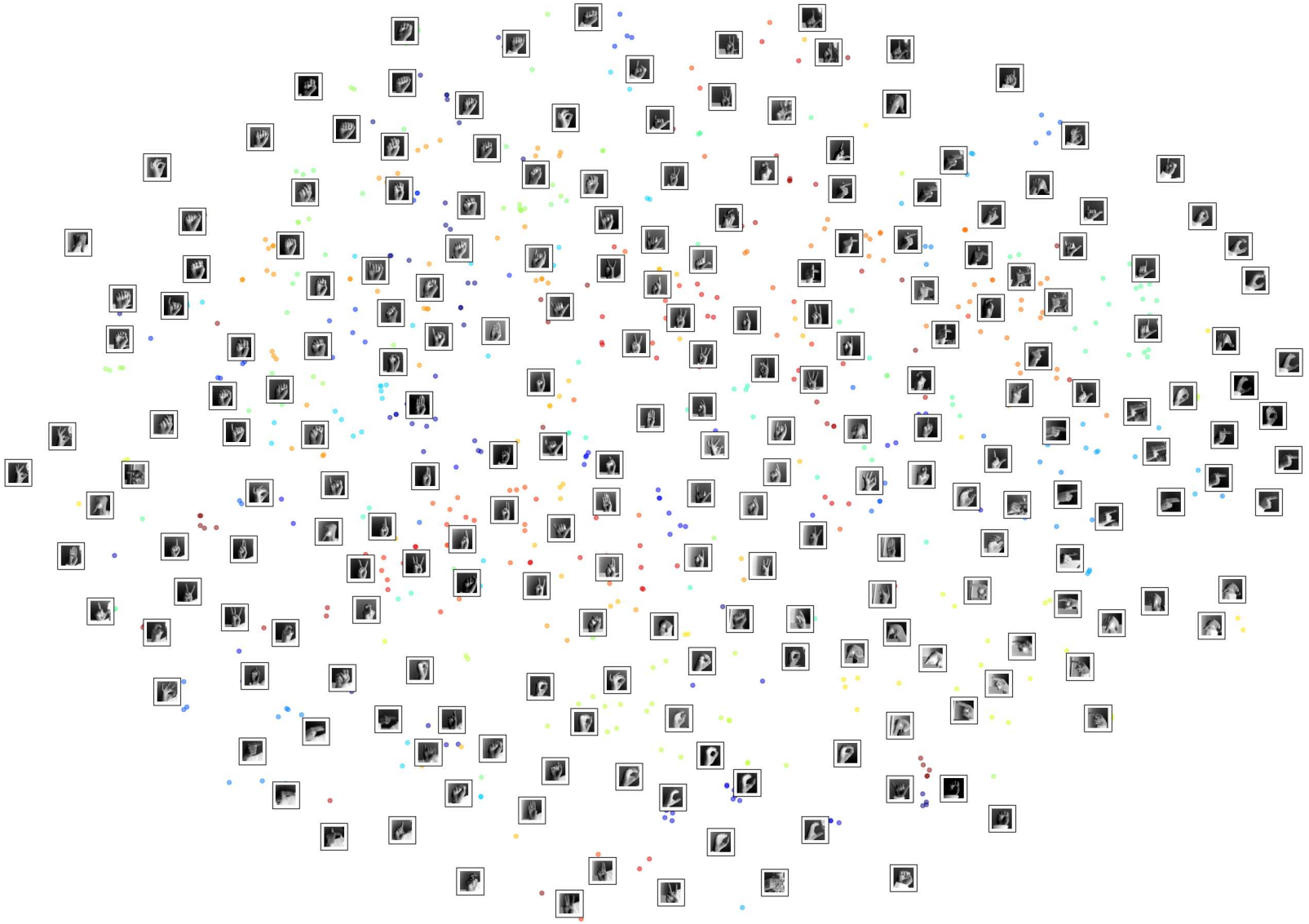
In [57]:
```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, init="random", learning_rate="auto", random_state=22)
```

In [58]:
```
X_tsne_reduced = tsne.fit_transform(X)
```

In [59]:
```
plot_principal_components(X_tsne_reduced[:1000], y[:1000])
```


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
1.0
0.8
0.6
0.4
0.2
0.0

In [60]:
```
plot_digits(X_tsne_reduced[:1000], y[:1000], images=X.values[:1000], figsize=(35, 25))
```

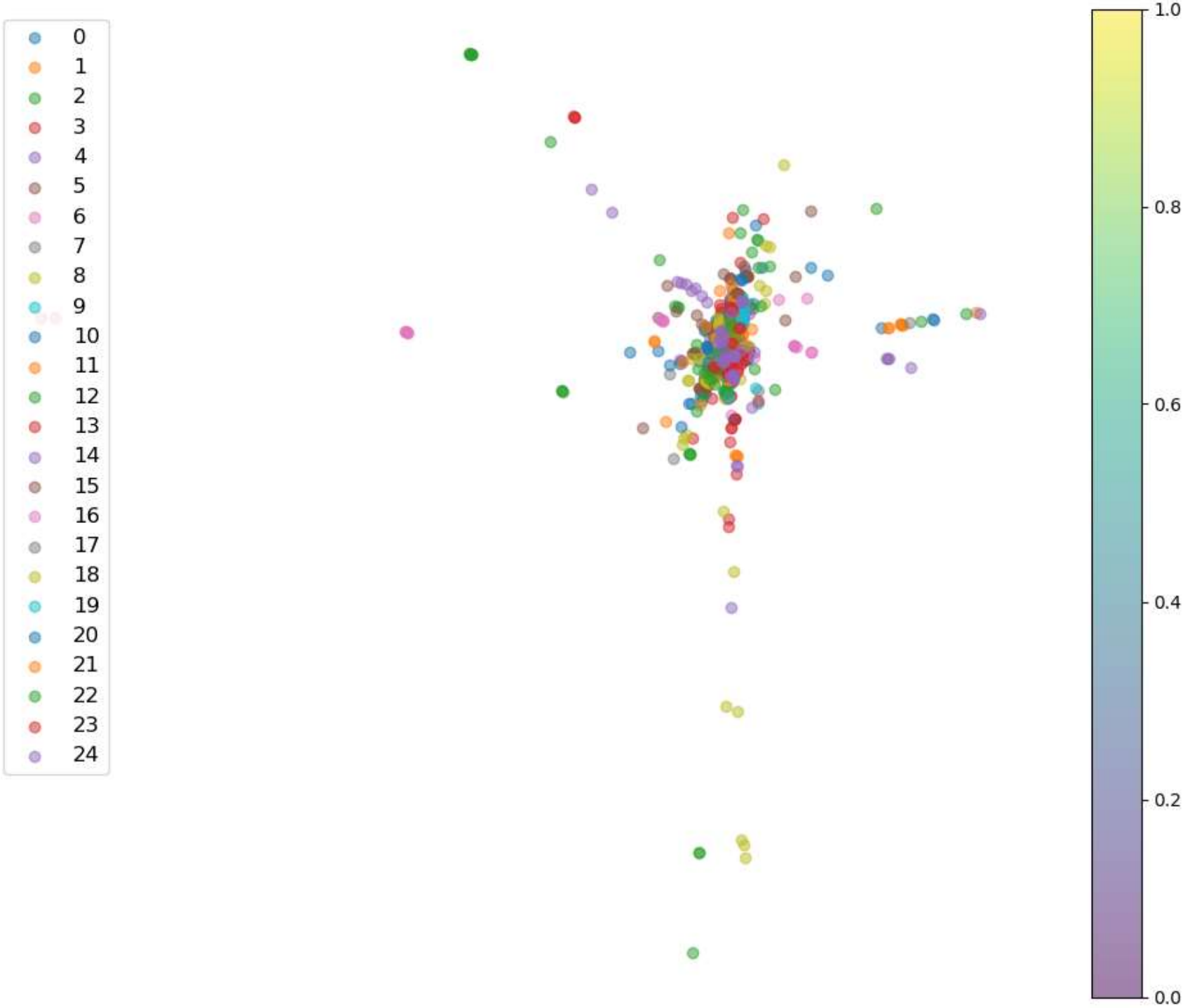


3. LLE

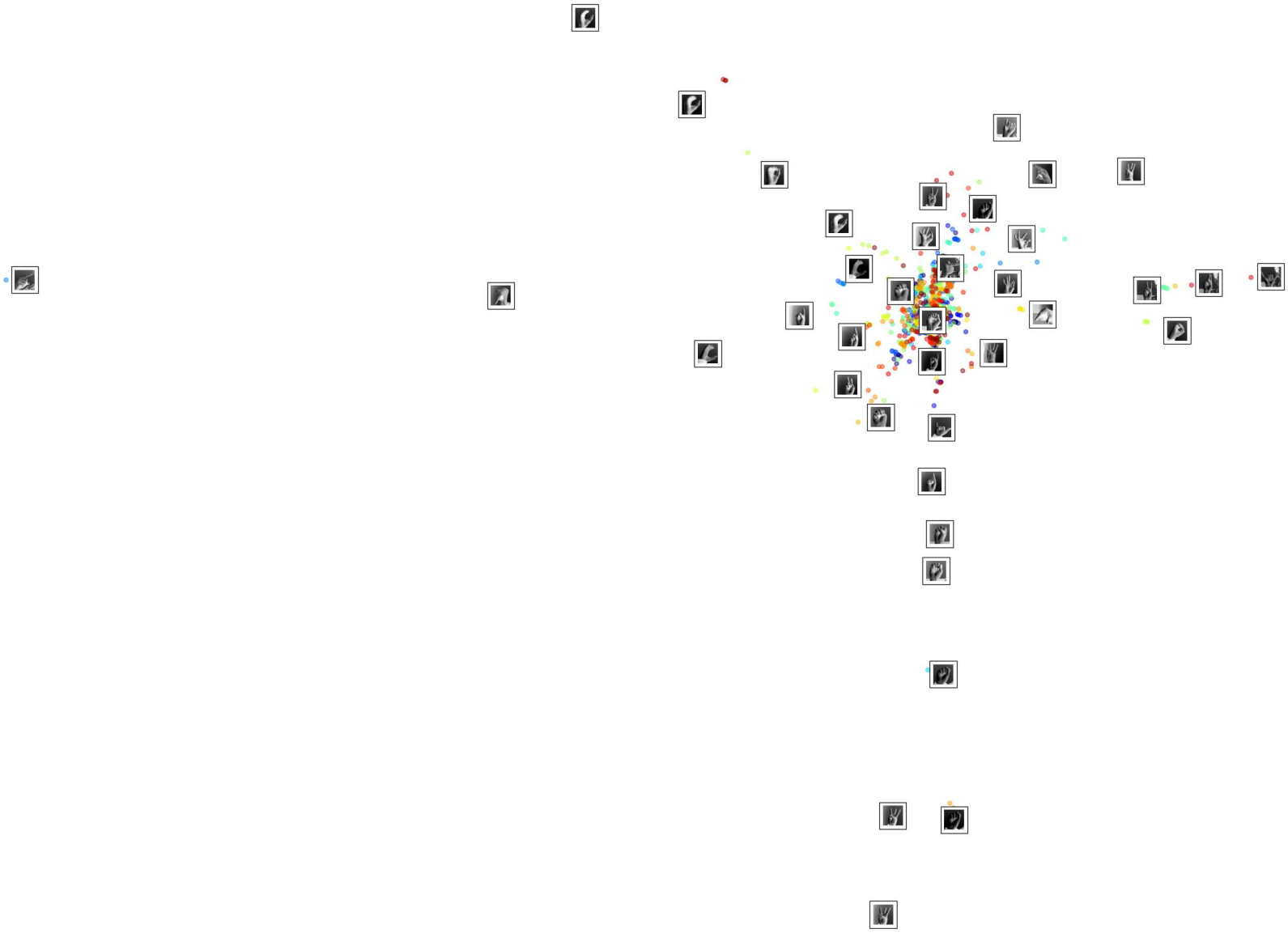
In [61]:
```
from sklearn.manifold import LocallyLinearEmbedding

lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10, random_state=22)
```

In [62]:
```
X_lle_reduced = lle.fit_transform(X)
```

In [63]:
```
plot_principal_components(X_lle_reduced[:1000], y[:1000])
```


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
1.0
0.8
0.6
0.4
0.2
0.0

In [64]:
```
plot_digits(X_lle_reduced[:1000], y[:1000], images=X.values[:1000], figsize=(35, 25))
```



4. MDS

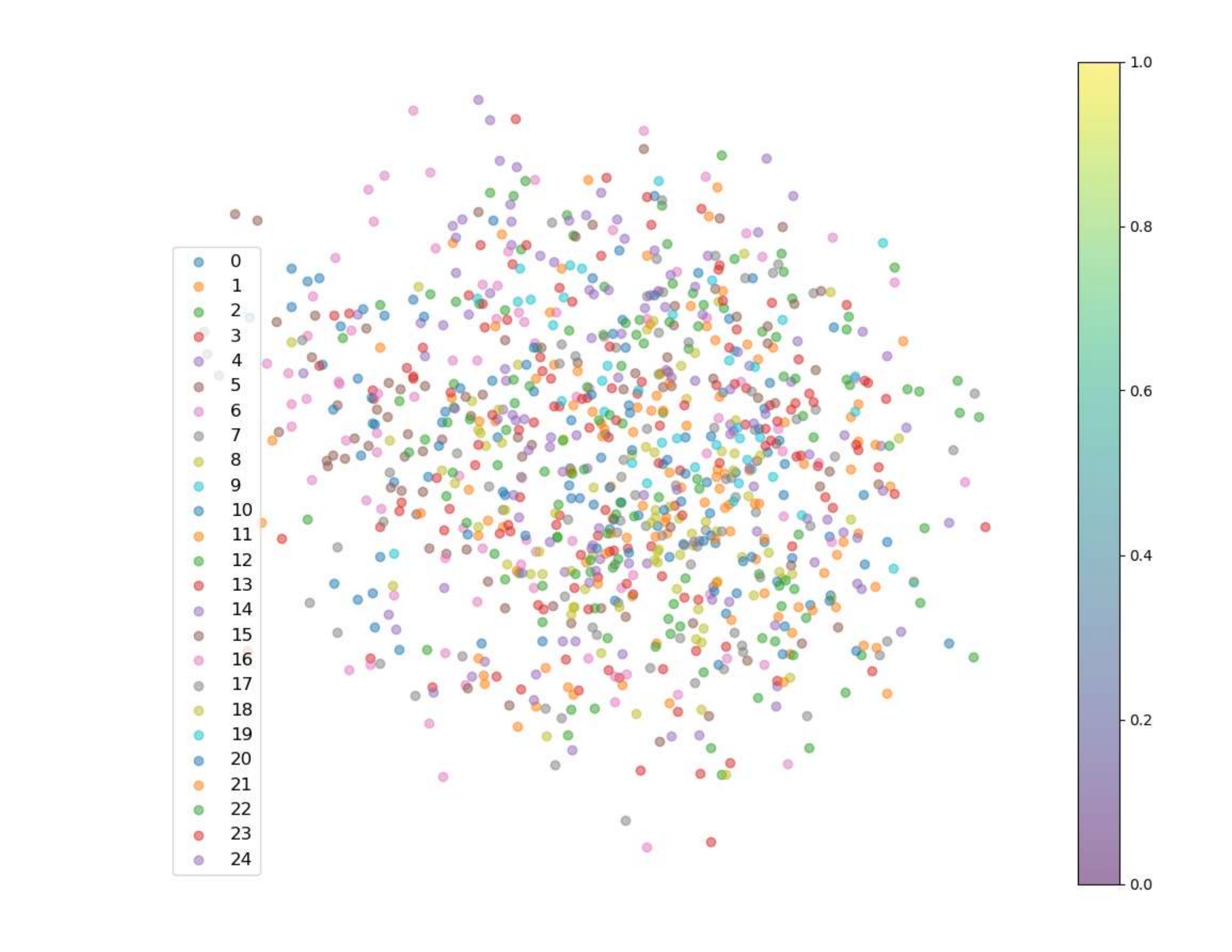
In [65]:
```
from sklearn.manifold import MDS

mds = MDS(n_components=2, random_state=22)
```

In [66]:
```
random_sample = training_data.sample(n = 5000, random_state = 22)
```

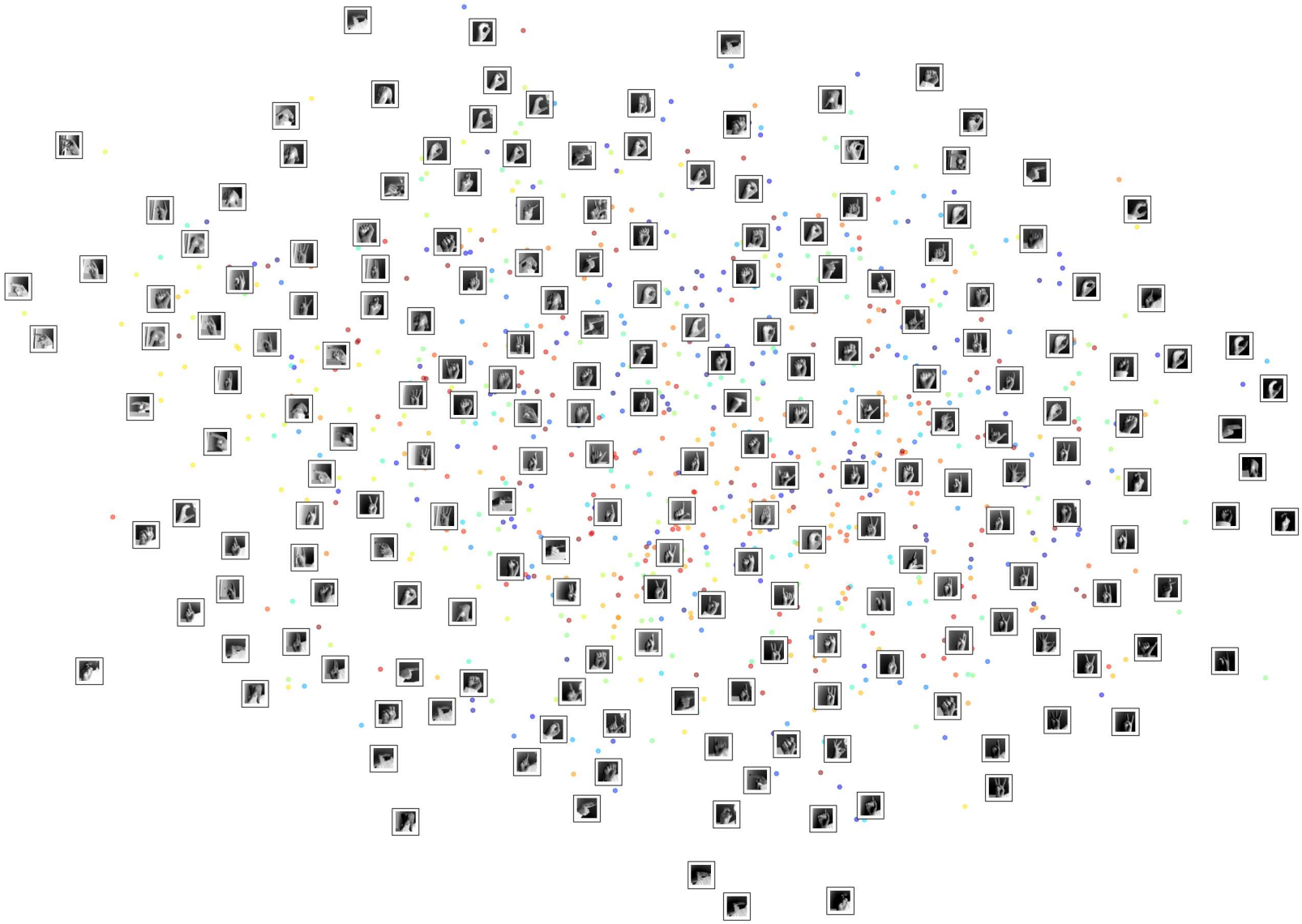
In [67]:
```
X = random_sample
y = training_labels[random_sample.index]
```

In [68]:
```
X_mds_reduced = mds.fit_transform(X)
```

In [69]:
```
plot_principal_components(X_mds_reduced[:1000], y[:1000])
```


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
1.0
0.8
0.6
0.4
0.2
0.0

In [70]:
```
plot_digits(X_mds_reduced[:1000], y[:1000], images=X.values[:1000], figsize=(35, 25))
```



Answer:

PCA, t-SNE, LLE and MDS are different dimentionality reduction techniques. We have visualized output of each method. Every visualization tries to show different clustering patterns between the data. For this example, we would generally prefer the visualization given by TSNE. The data seems not that mutch visually separable, and is arranged in the form of parse clusters. If we look closely enough, TSNE managed to cluster all the signs together. It has better performance over all other methodologies.

==========

Question:

Take 10000 samples of the training portion of fashion MNIST dataset and cluster the images using K-Means. To speed up the algorithm, use PCA to reduce the dimensionality of the dataset. Ensure that you have a good number of clusters using one of the techniques we discussed in class. [4 points] Visualize the clusters (you can show only a subset of images): do you see similar clothing items in each cluster?

In [71]:
```
random_sample = training_data.sample(n = 10000, random_state = 22)
```

In [72]:
```
# Extracting 10000 samples of the trainig data
# Split the dataset into training and validation data sets
X_train = random_sample
y_train = training_labels[random_sample.index]
```

In [73]:
```
pca = PCA(0.95)
X_train_pca = pca.fit_transform(X_train)
```

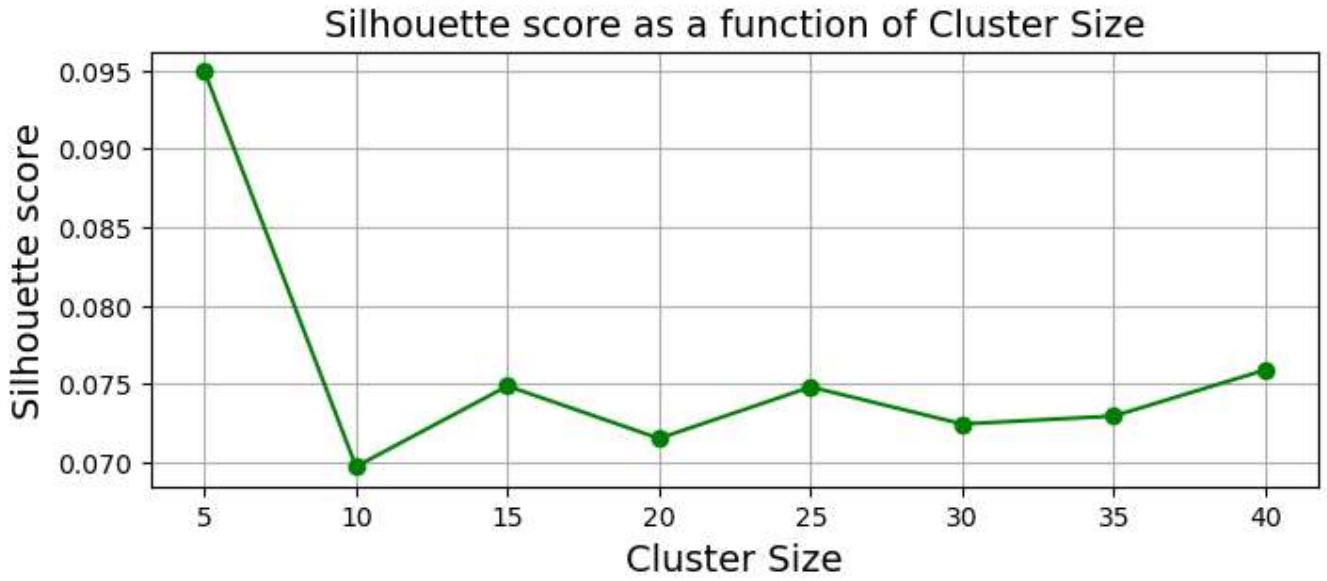
First of all, we will run K-Means algorithm for different size of clusters. We can use silhouette score and inertia as one of the metrics to determine the optimum cluster size.

As suggested in problem statement, we will directly use the reduced data given by PCA insted of using the full dimention data.

In [74]:
```
from sklearn.cluster import KMeans

cluster_size_range = [5,10,15,20, 25,30,35,40]
kmeans_per_k = []
for k in cluster_size_range:
    print(f"Fitting data for Cluster Size={k}")
    kmeans_obj = KMeans(n_clusters=k, random_state=42, init="random", n_init=50)
    kmeans_obj.fit(X_train_pca)
    kmeans_per_k.append(kmeans_obj)
```

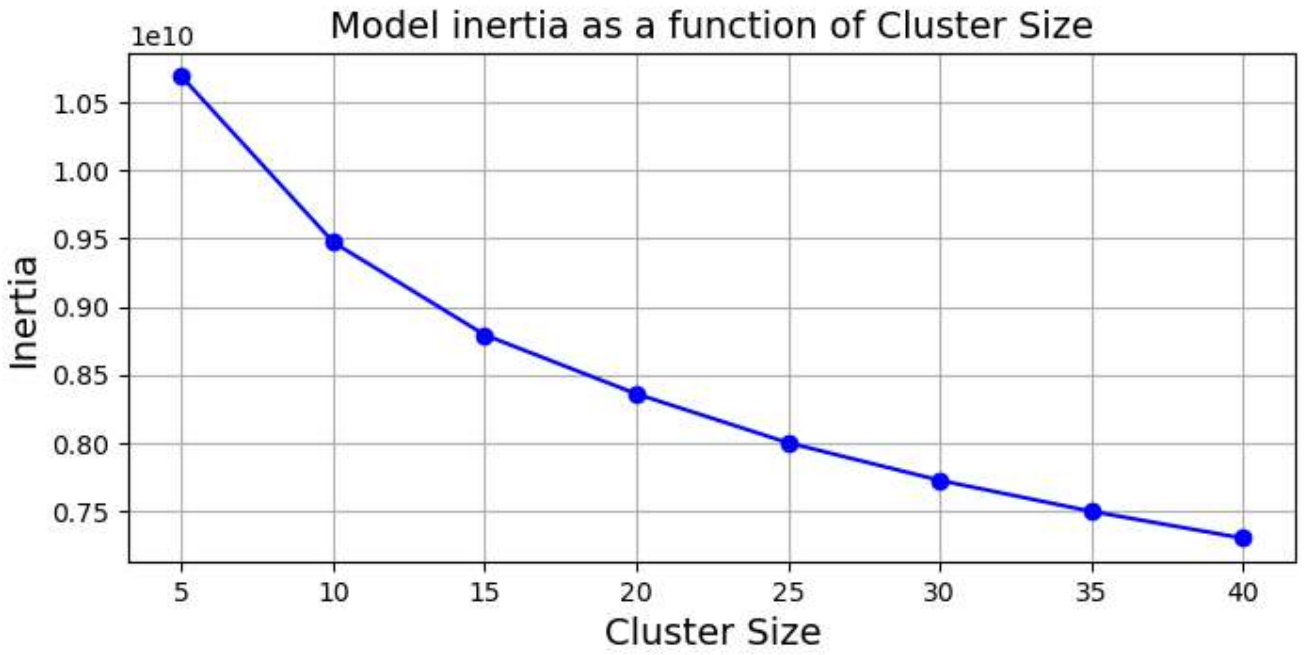
```
Fitting data for Cluster Size=5
Fitting data for Cluster Size=10
Fitting data for Cluster Size=15
Fitting data for Cluster Size=20
Fitting data for Cluster Size=25
Fitting data for Cluster Size=30
Fitting data for Cluster Size=35
Fitting data for Cluster Size=40
```

In [75]:
```python
# plotting the silhouette score
from sklearn.metrics import silhouette_score

silhouette_scores = [silhouette_score(X_train_pca, model.labels_) for model in kmeans_per_k]

plt.figure(figsize=(8, 3))
plt.plot(cluster_size_range, silhouette_scores, "go-")
plt.title("Silhouette score as a function of Cluster Size")
plt.xlabel("Cluster Size")
plt.ylabel("Silhouette score")
plt.grid()
plt.show()
```



In [76]:
```python
# plotting the intertia
inertias = [model.inertia_ for model in kmeans_per_k]

plt.figure(figsize=(8, 3.5))
plt.plot(cluster_size_range, inertias, "bo-")
```

```
plt.title("Model inertia as a function of Cluster Size")
plt.xlabel("Cluster Size")
plt.ylabel("Inertia")
plt.grid()
plt.show()
```



By looking at the model inertia, the elbow is clearly visible at the cluster size 25

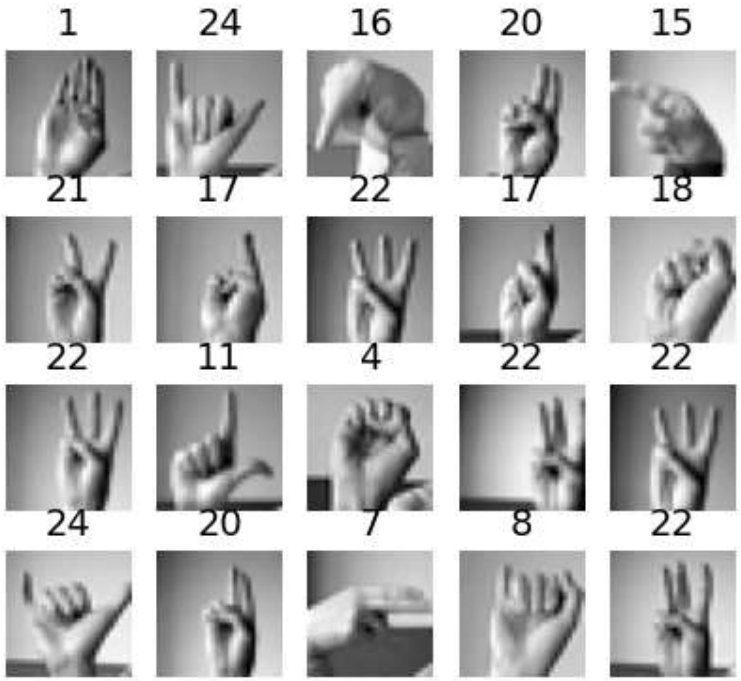
In [77]:
```
best_model = kmeans_per_k[4]
```

In [78]:
```
def plot_items(items, labels, n_cols=5):
    n_rows = (len(items) - 1) // n_cols + 1
    plt.figure(figsize=(n_cols, n_rows * 1.1))
    for index, (item, label) in enumerate(zip(items, labels)):
        item = item.reshape(28,28)
        plt.subplot(n_rows, n_cols, index + 1)
        plt.imshow(item, cmap="gray")
        plt.axis("off")
        plt.title(label)
    plt.show()
```

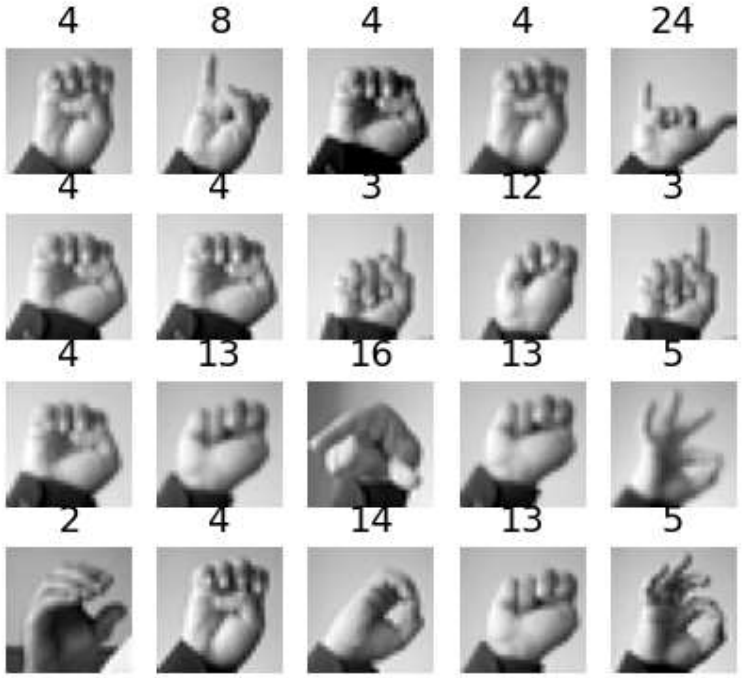
For convinience we have plotted first 20 images per cluster.

In [79]:
```
for cluster_id in np.unique(best_model.labels_):
    print("Cluster", cluster_id)
    in_cluster = best_model.labels_==cluster_id
    items = X_train[in_cluster]
    labels = y_train[in_cluster]
    plot_items(items.values[:20], labels[:20])
```

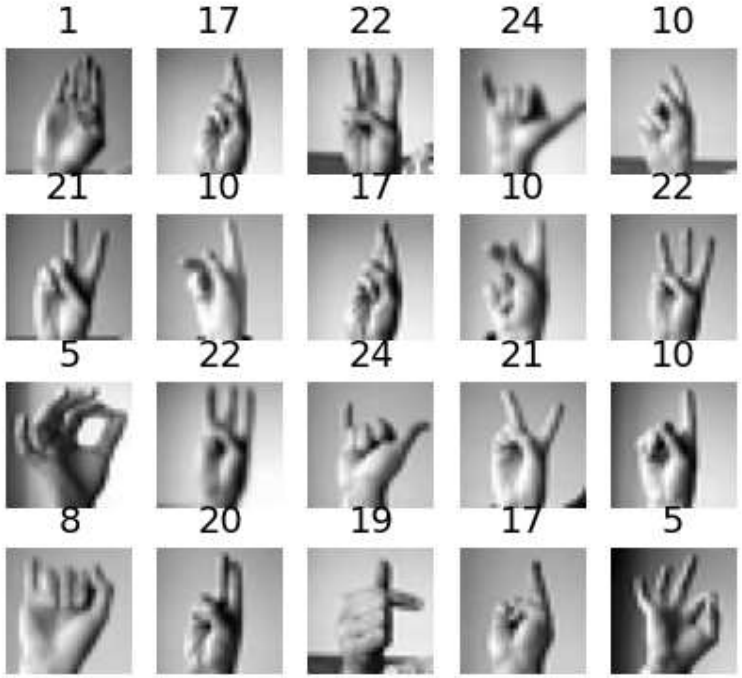
```
Cluster 0
```



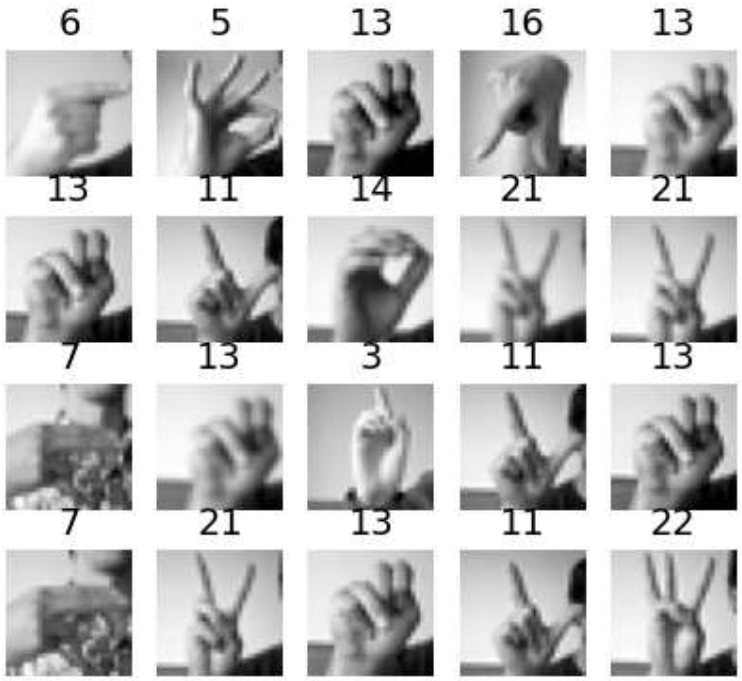

```
Cluster 1
```


4 8 4 4 24
4 4 3 12 3
4 13 16 13 5
2 4 14 13 5

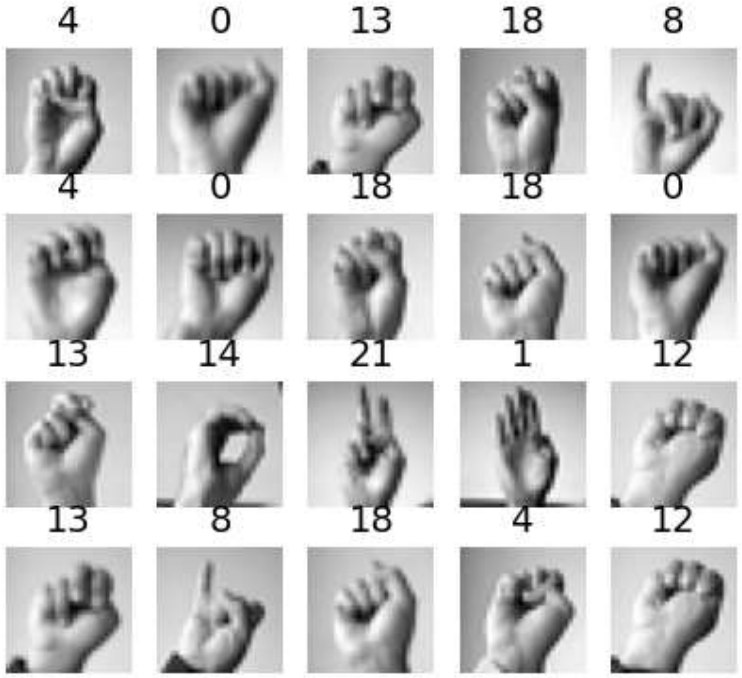
Cluster 2


1 17 22 24 10
21 10 17 10 22
5 22 24 21 10
8 20 19 17 5

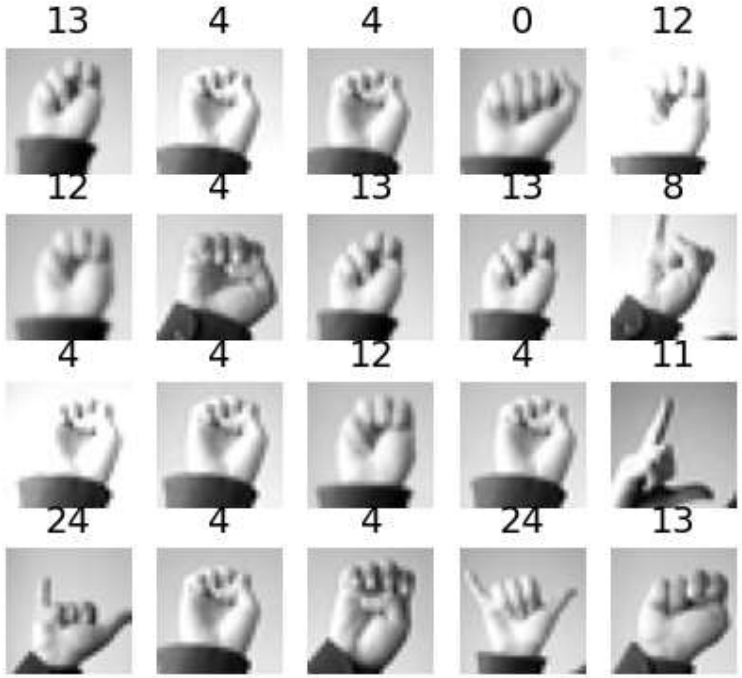
Cluster 3



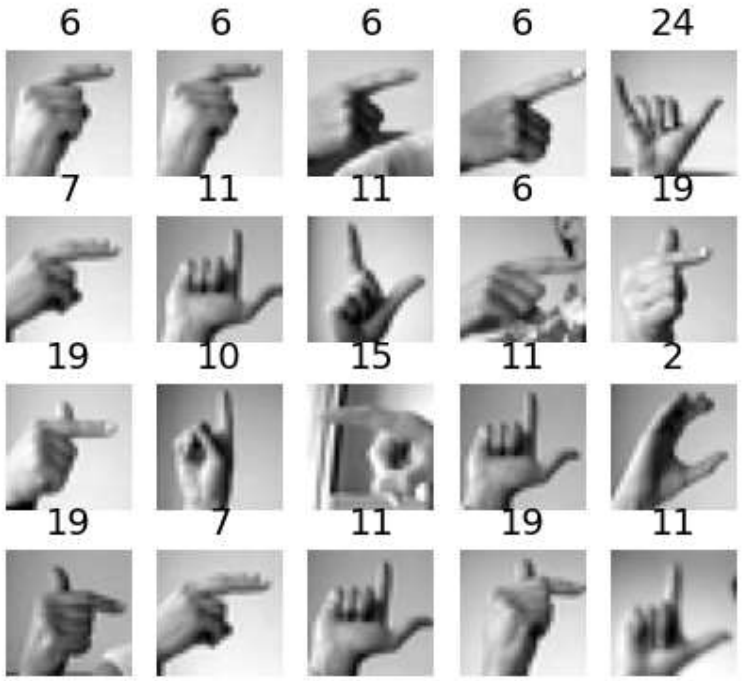
Cluster 4


4 0 13 18 8
4 0 18 18 0
13 14 21 1 12
13 8 18 4 12

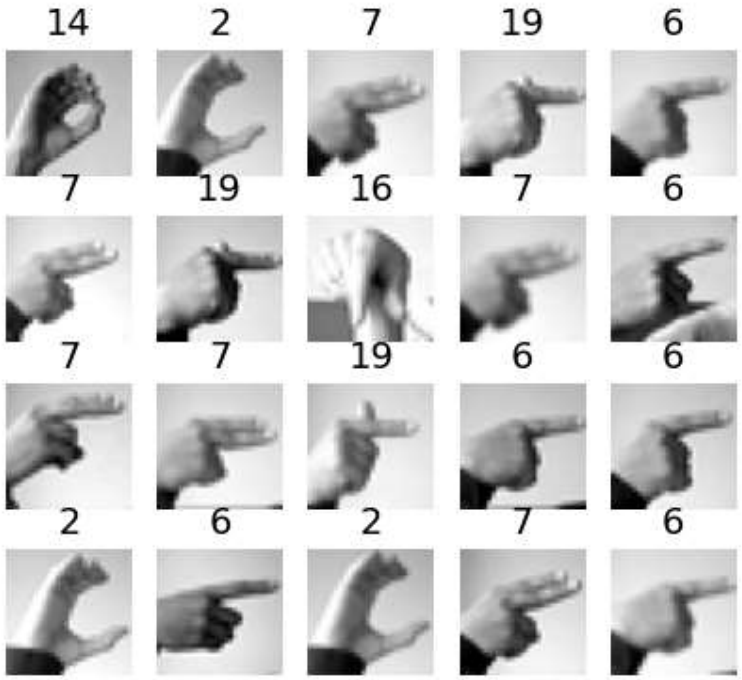
Cluster 5


13 4 4 0 12
12 4 13 13 8
4 4 12 4 11
24 4 4 24 13

Cluster 6



Cluster 7

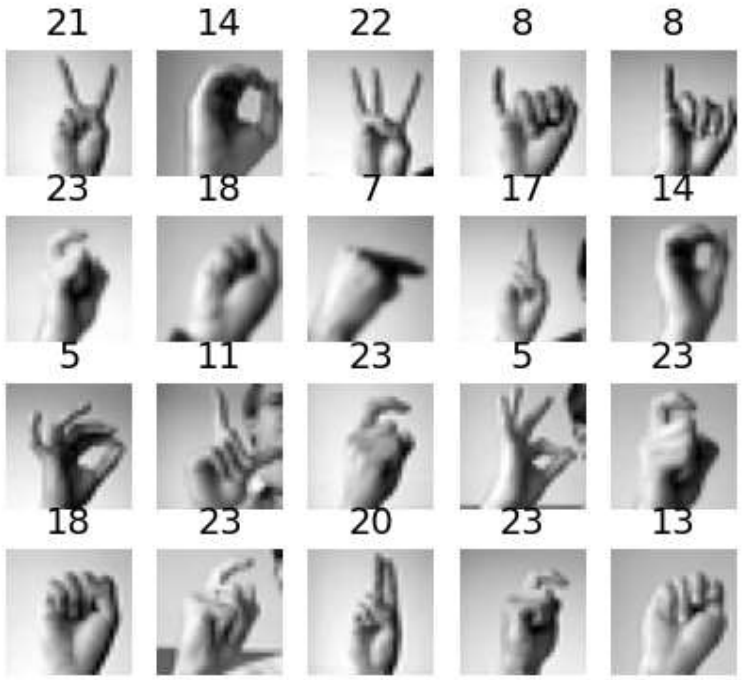

14 2 7 19 6
7 19 16 7 6
7 7 19 6 6
2 6 2 7 6

Cluster 8

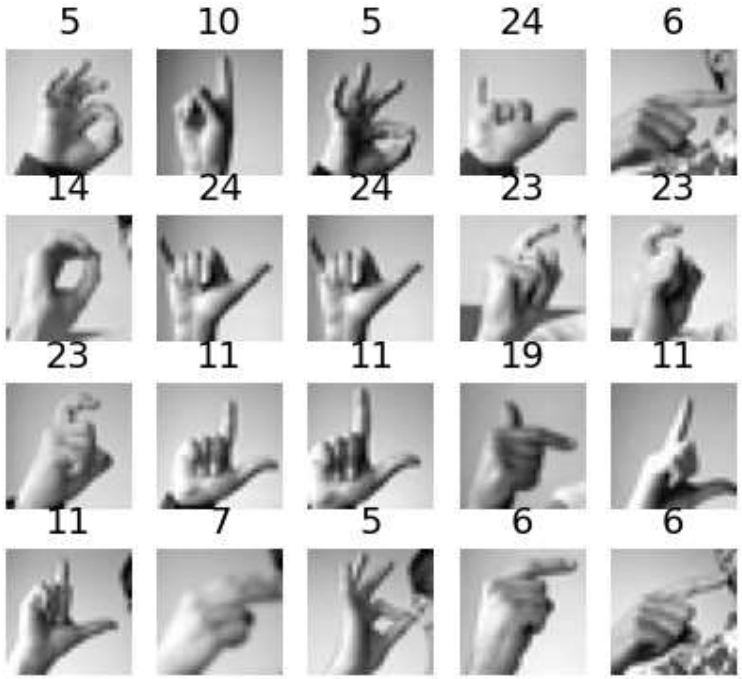

21 23 21 19 16
0 21 17 19 10
11 20 1 19 22
1 2 24 21 12

Cluster 9

Cluster 10

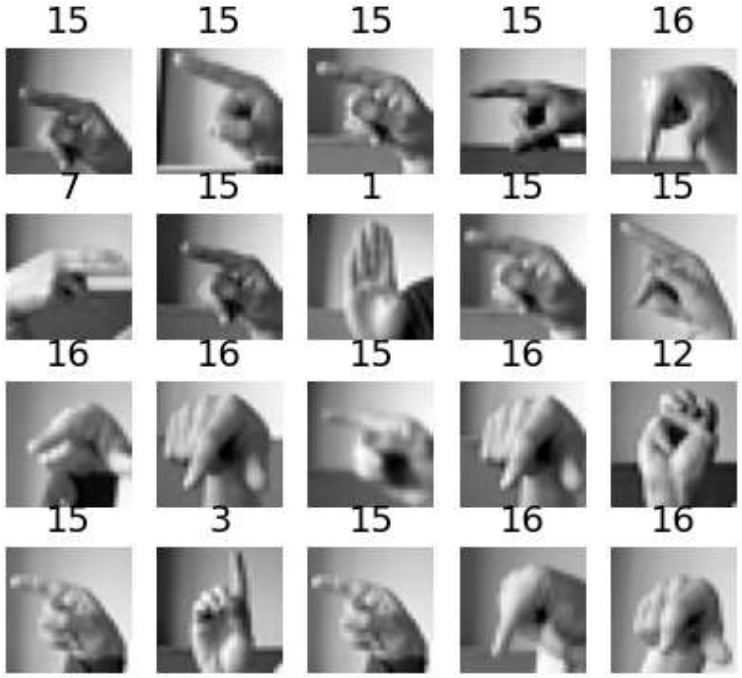

21 14 22 8 8
23 18 7 17 14
5 11 23 5 23
18 23 20 23 13

Cluster 11


5 10 5 24 6
14 24 24 23 23
23 11 11 19 11
11 7 5 6 6

Cluster 12

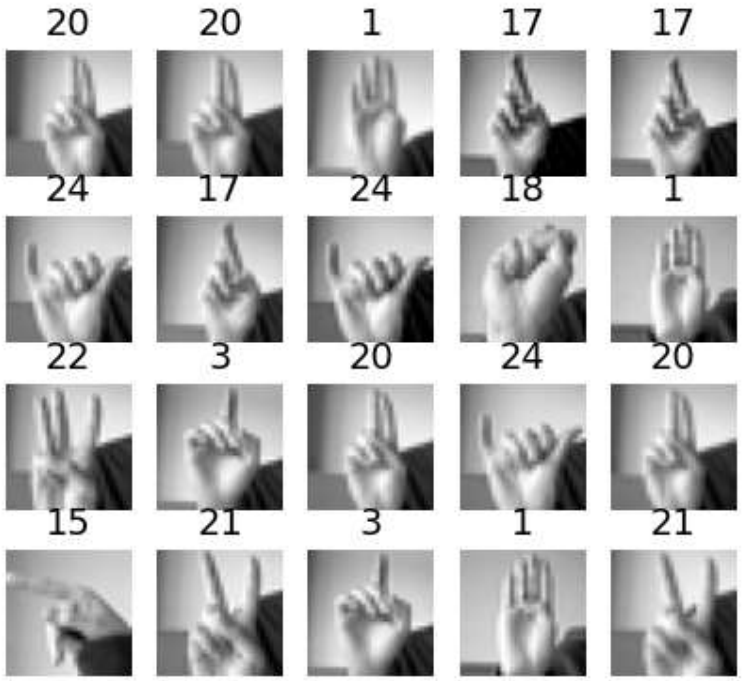
Cluster 13


15 15 15 15 16
7 15 1 15 15
16 16 15 16 12
15 3 15 16 16

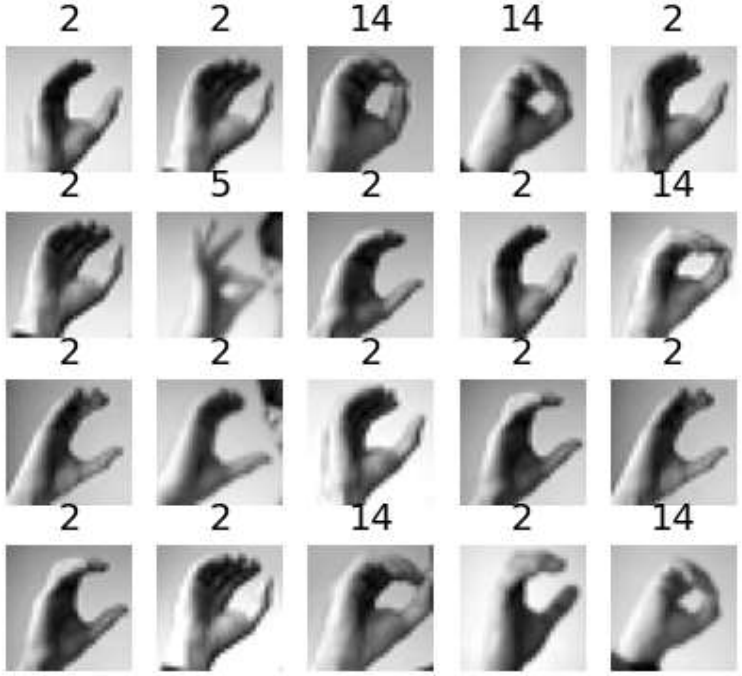
Cluster 14


18 14 3 4 12
13 0 1 14 0
13 23 14 13 13
2 13 2 0 18

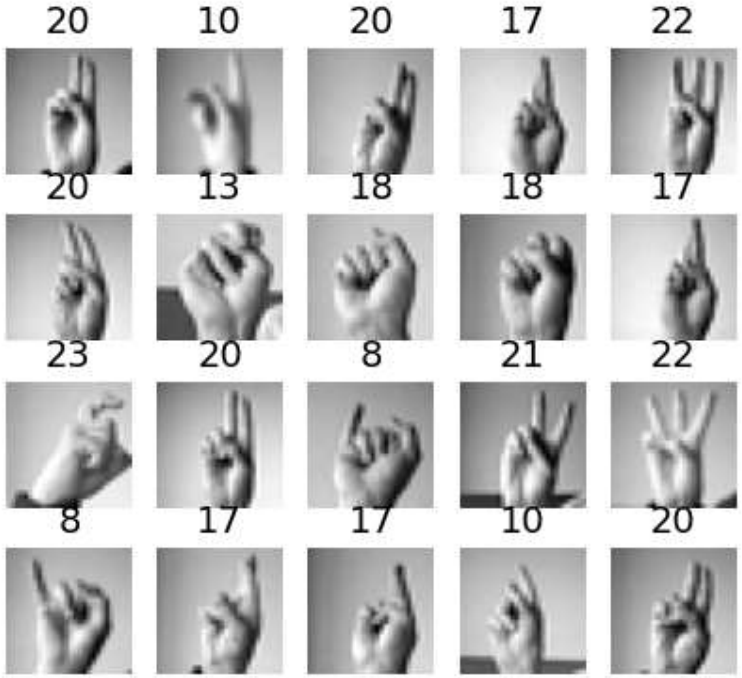
Cluster 15



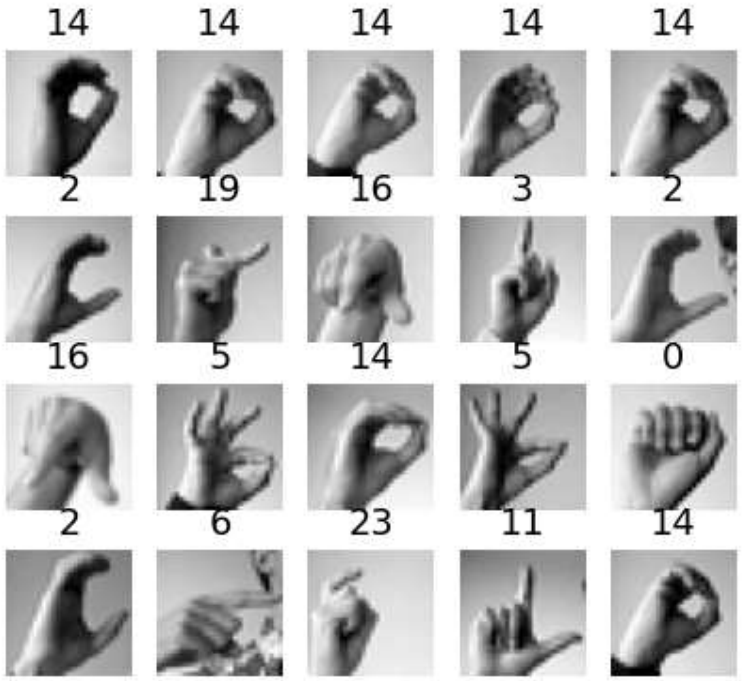
Cluster 16


2 2 14 14 2
2 5 2 2 14
2 2 2 2 2
2 2 14 2 14

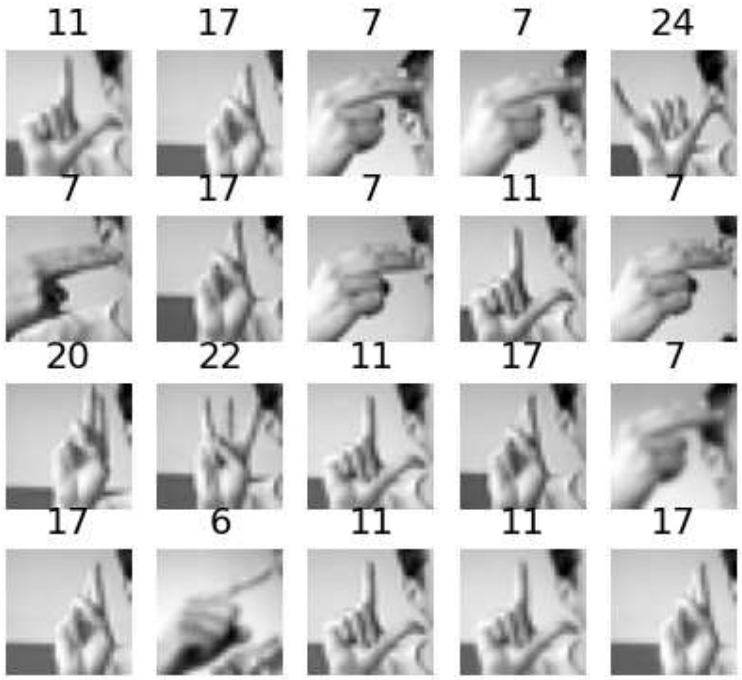
Cluster 17


20 10 20 17 22
20 13 18 18 17
23 20 8 21 22
8 17 17 10 20

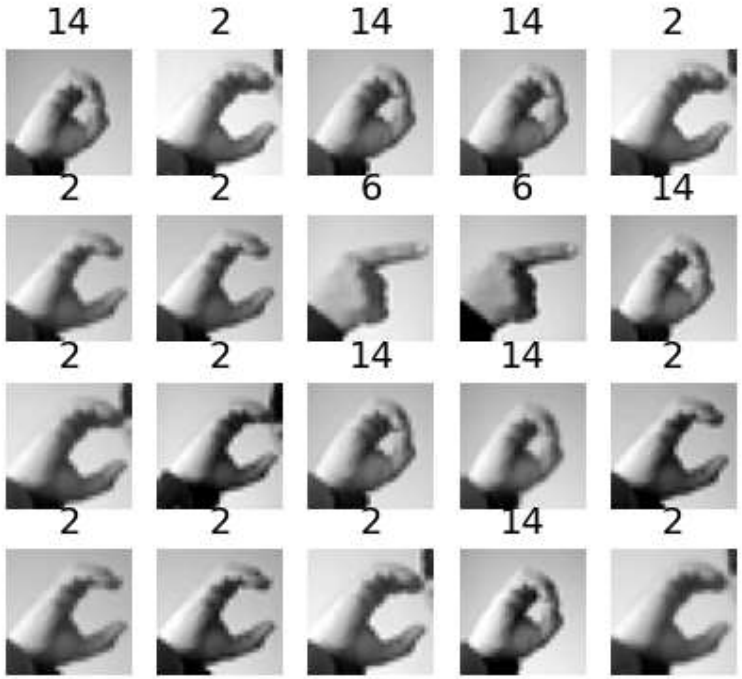
Cluster 18



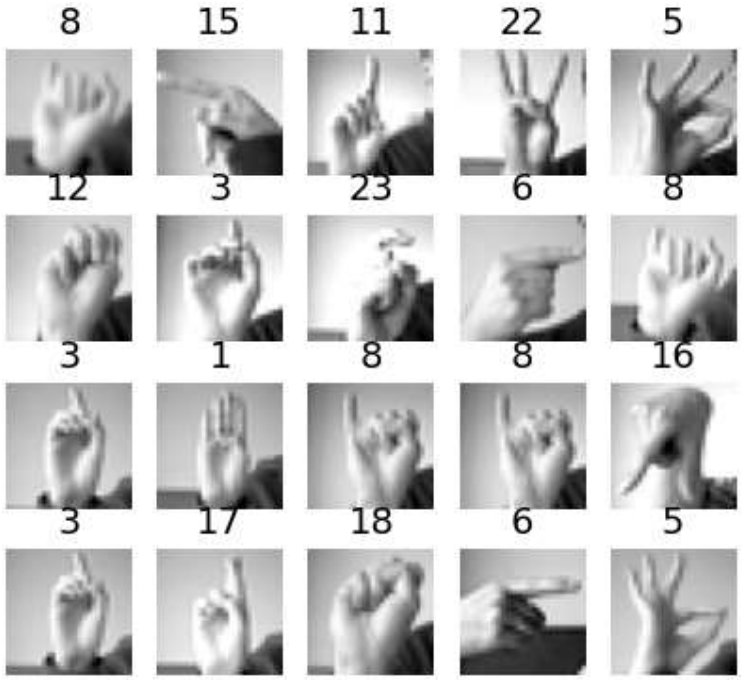
Cluster 19


11 17 7 7 24
7 17 7 11 7
20 22 11 17 7
17 6 11 11 17

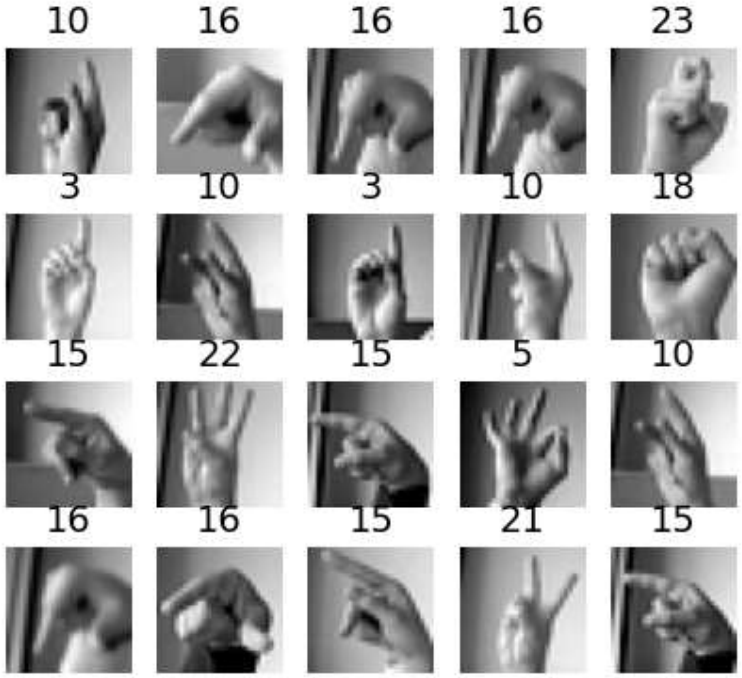
Cluster 20


14 2 14 14 2
2 2 6 6 14
2 2 14 14 2
2 2 2 14 2

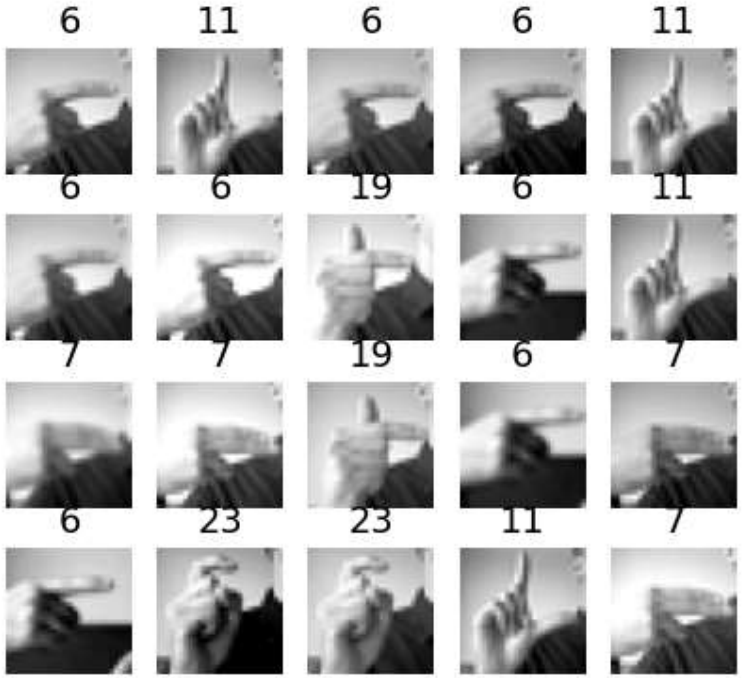
Cluster 21



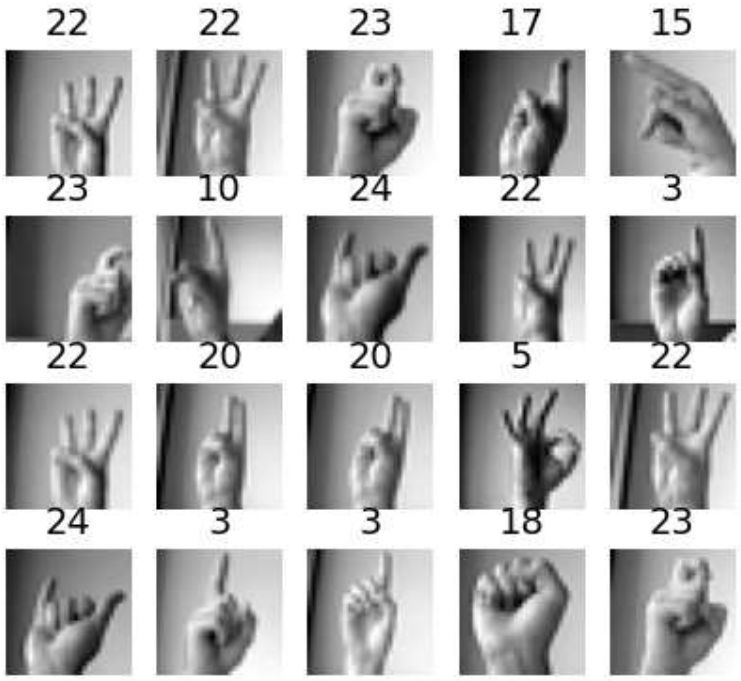
Cluster 22


10 16 16 16 23
3 10 3 10 18
15 22 15 5 10
16 16 15 21 15

Cluster 23


6 11 6 6 11
6 6 19 6 11
7 7 19 6 7
6 23 23 11 7

```
Cluster 24
```



Answer:

There are certain clusters like cluster 12,13,16,20 which contains most of the similar clothing types. Also, if we take the generality into consideration, the K-means clustering has managed to cluster the items with similar hand guestures together. However, the algorithm has managed to cluster closed fists, circled fists and raised fingers together. But there are some clusters(cluster 3, cluster 6, cluster 7), which has incorrectly classified the data.

==========

Question:

Take 10000 samples of the training portion of fashion MNIST dataset and cluster the images using a Gaussian mixture model. To speed up the algorithm, use PCA to reduce the dimensionality of the dataset. Ensure that you have a good number of clusters using one of the techniques we discussed in the class.

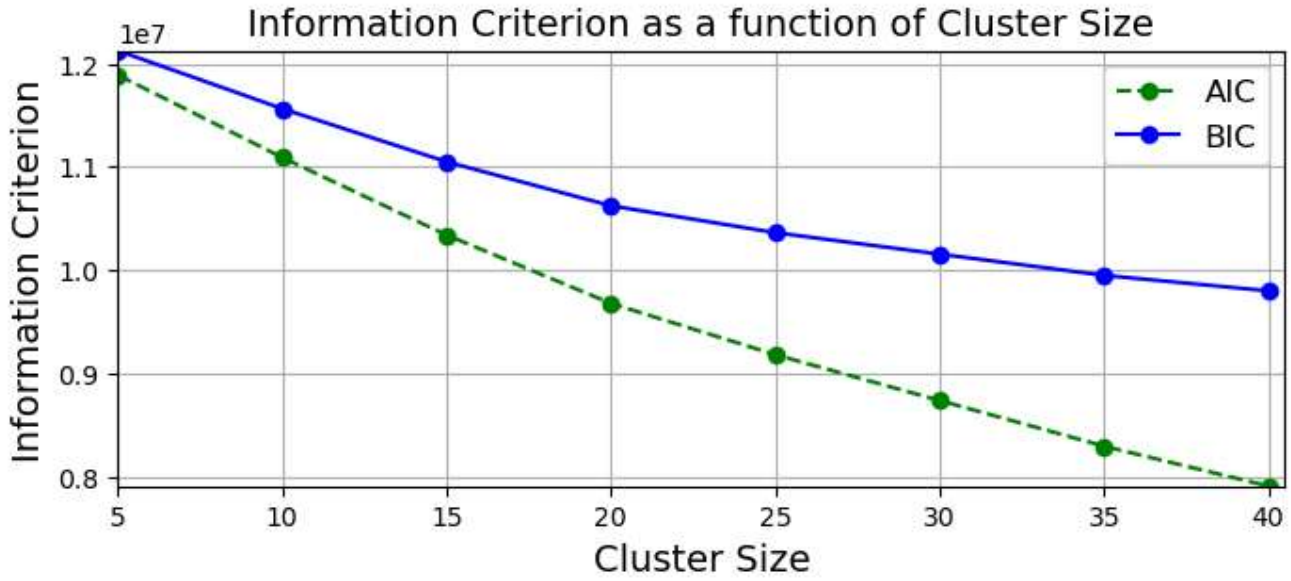
In [80]:
```
from sklearn.mixture import GaussianMixture
```

In [81]:
```
cluster_size_range = [5,10,15,20, 25,30,35,40]
gms_per_k = []
for k in cluster_size_range:
    print(f"Fitting data for Cluster Size={k}")
    gmm_model = GaussianMixture(n_components=k, n_init=10, random_state=42)
    gmm_model.fit(X_train_pca)
    gms_per_k.append(gmm_model)
```

```
Fitting data for Cluster Size=5
Fitting data for Cluster Size=10
Fitting data for Cluster Size=15
Fitting data for Cluster Size=20
Fitting data for Cluster Size=25
Fitting data for Cluster Size=30
Fitting data for Cluster Size=35
Fitting data for Cluster Size=40
```

In [82]:
```
BICS = [model.bic(X_train_pca) for model in gms_per_k]
AICS = [model.aic(X_train_pca) for model in gms_per_k]
```

In [83]:
```
plt.figure(figsize=(8, 3))
plt.plot(cluster_size_range, AICS, "go--", label="AIC")
plt.plot(cluster_size_range, BICS, "bo-", label="BIC")
plt.title("Information Criterion as a function of Cluster Size")
plt.xlabel("Cluster Size")
plt.ylabel("Information Criterion")
plt.axis([min(cluster_size_range), max(cluster_size_range)+0.5, min(AICS) - 50, max(BICS) + 50])

plt.legend()
plt.grid()
plt.show()
```



If we look at the above figure, for Cluster Size = 25, both BIC and AIC scores are minimum. After Cluster Size = 25, BIC score decreases with AICs. If we have to find the optimum K, then it will be 25. Thus, we will select the number of clusters as 25.

In [84]:
```
best_gmm_model = gms_per_k[4]
```

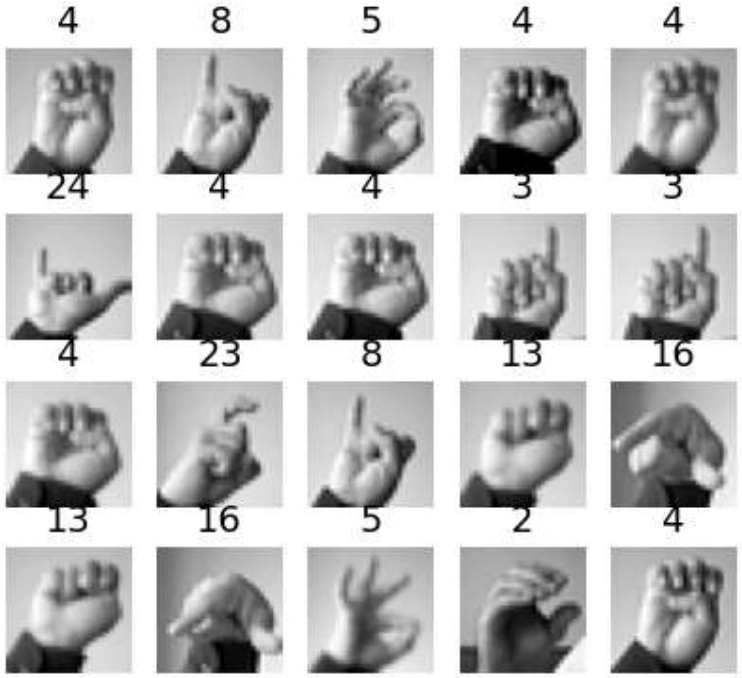
In [85]:
```
best_gmm_model.n_components
```

Out[85]:
```
25
```

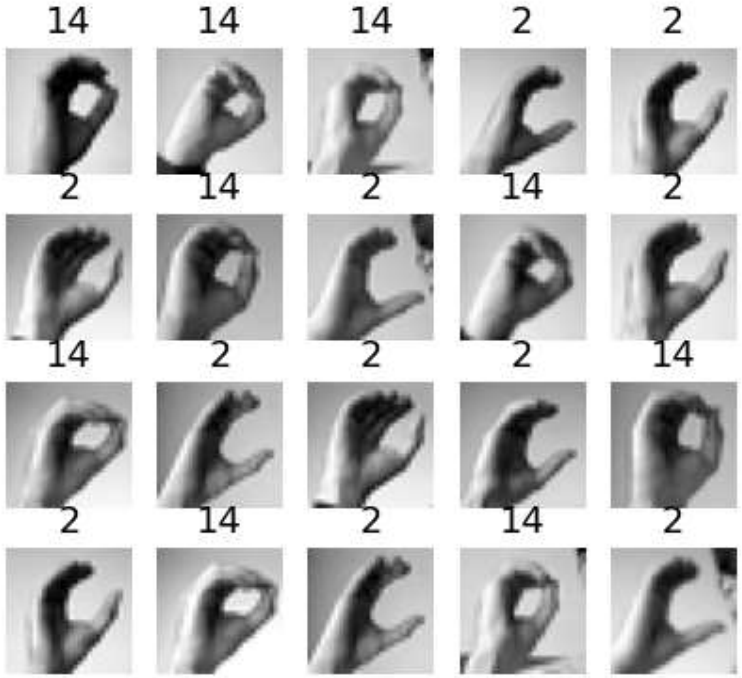
For convinience we have plotted first 20 images per cluster.

In [86]:
```
for cluster_id in range(best_gmm_model.n_components):
    print("Cluster", cluster_id)
    in_cluster = best_gmm_model.predict(X_train_pca) == cluster_id
    items = X_train[in_cluster]
    labels = y_train[in_cluster]
    plot_items(items.values[:20], labels[:20])
```

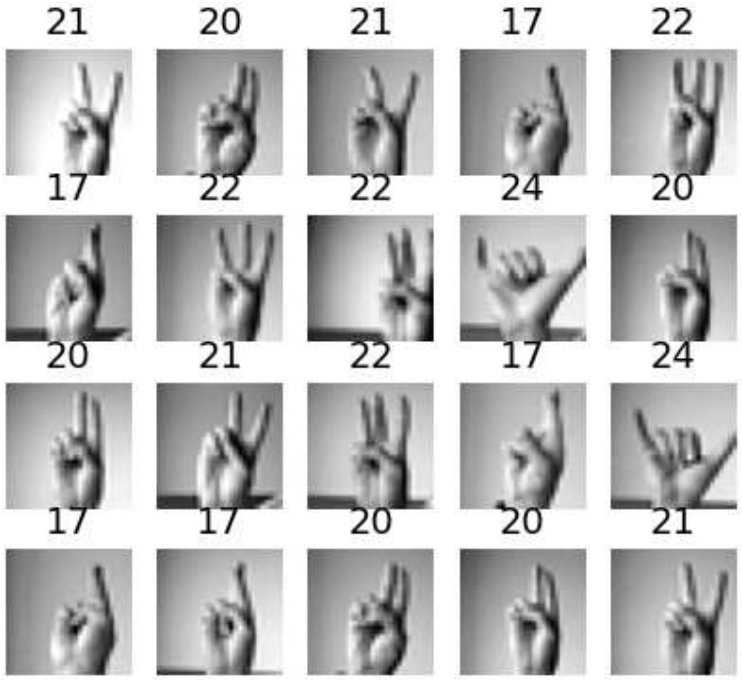
```
Cluster 0
```


4 8 5 4 4
24 4 4 3 3
4 23 8 13 16
13 16 5 2 4

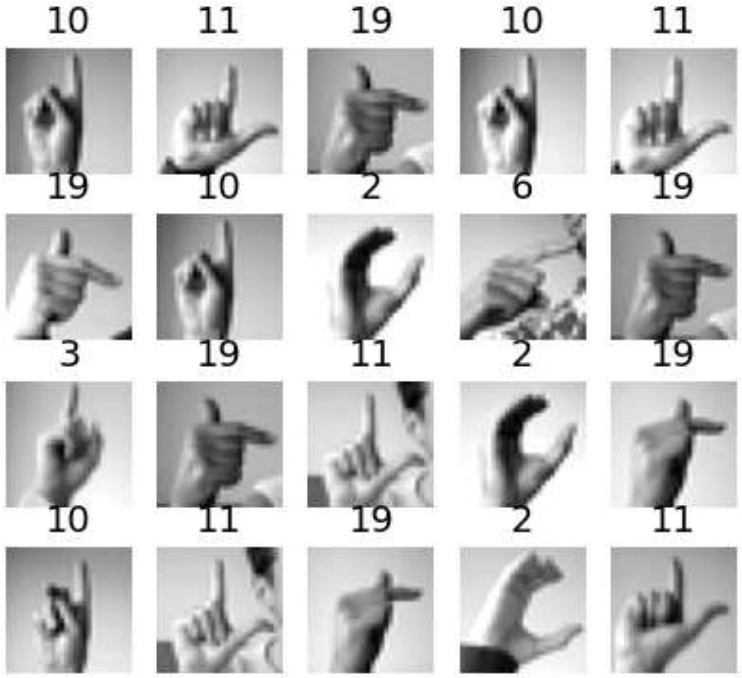
Cluster 1


14 14 14 2 2
2 14 2 14 2
14 2 2 2 14
2 14 2 14 2

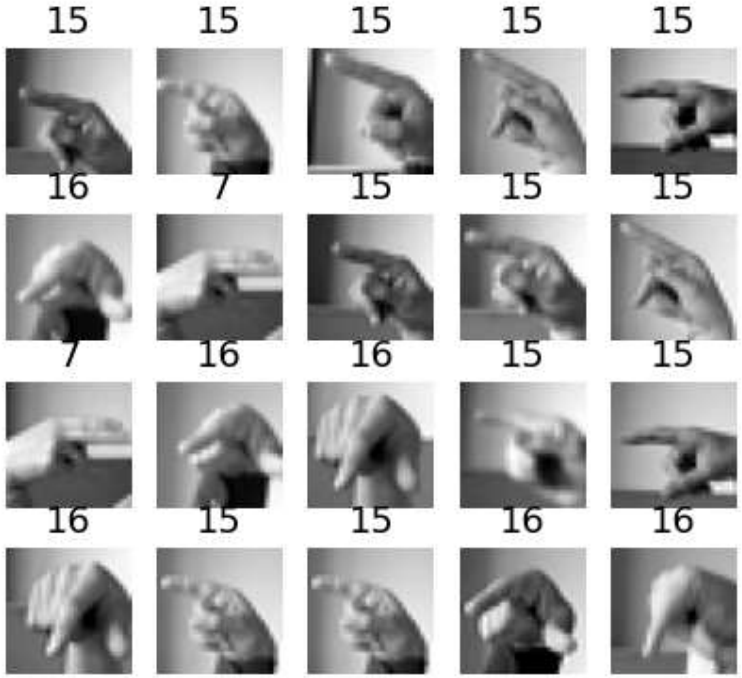
Cluster 2



Cluster 3

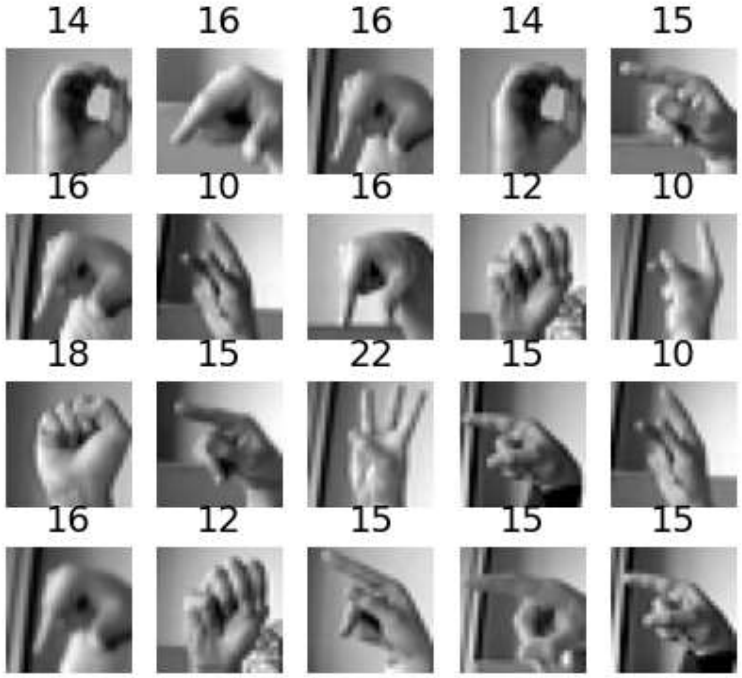

10 11 19 10 11
19 10 2 6 19
3 19 11 2 19
10 11 19 2 11

Cluster 4


15 15 15 15 15
16 7 15 15 15
7 16 16 15 15
16 15 15 16 16

Cluster 5

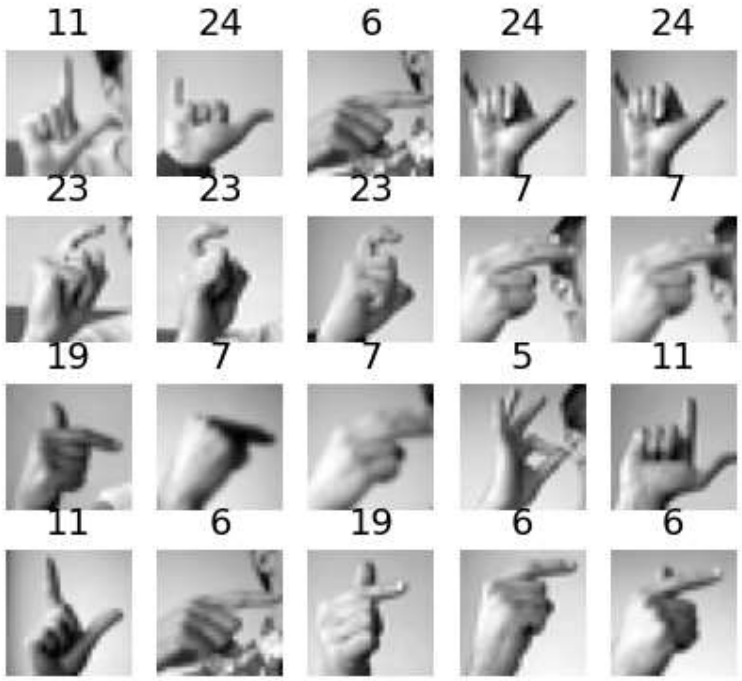
Cluster 6


14 16 16 14 15
16 10 16 12 10
18 15 22 15 10
16 12 15 15 15

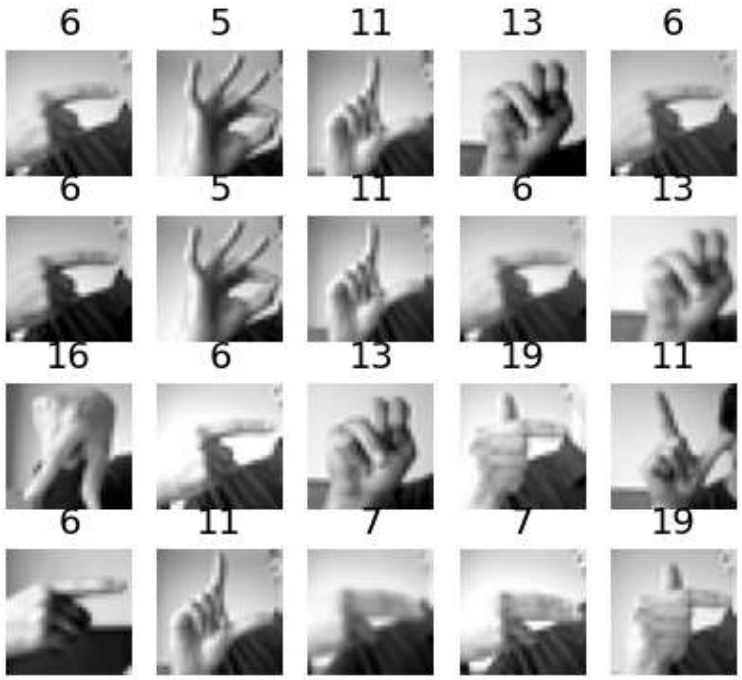
Cluster 7


11 21 23 22 3
17 1 19 8 23
24 17 20 17 17
11 23 21 19 5

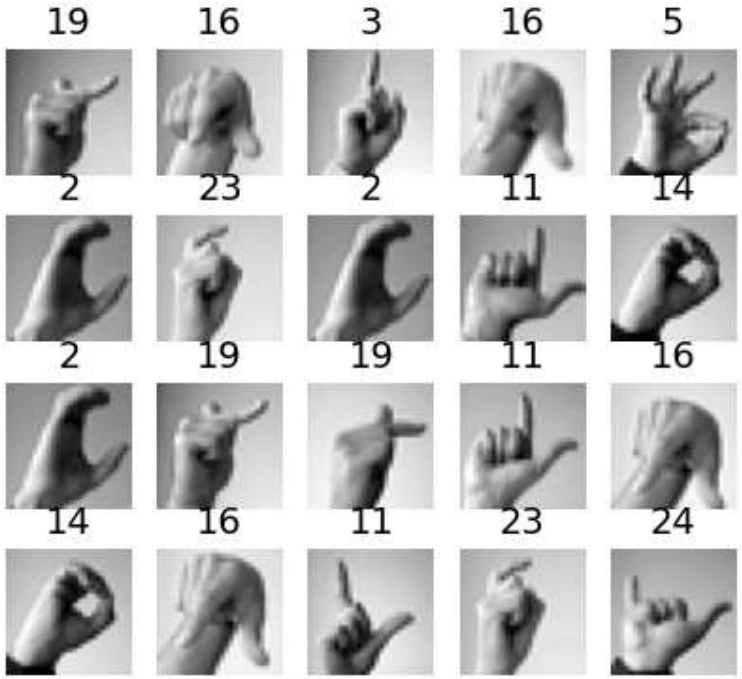
Cluster 8



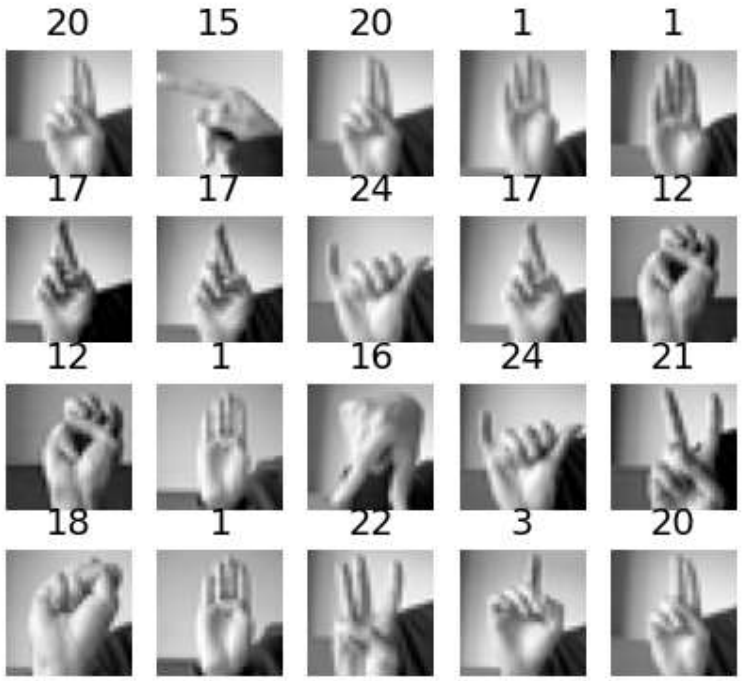
Cluster 9


6 5 11 13 6
6 5 11 6 13
16 6 13 19 11
6 11 7 7 19

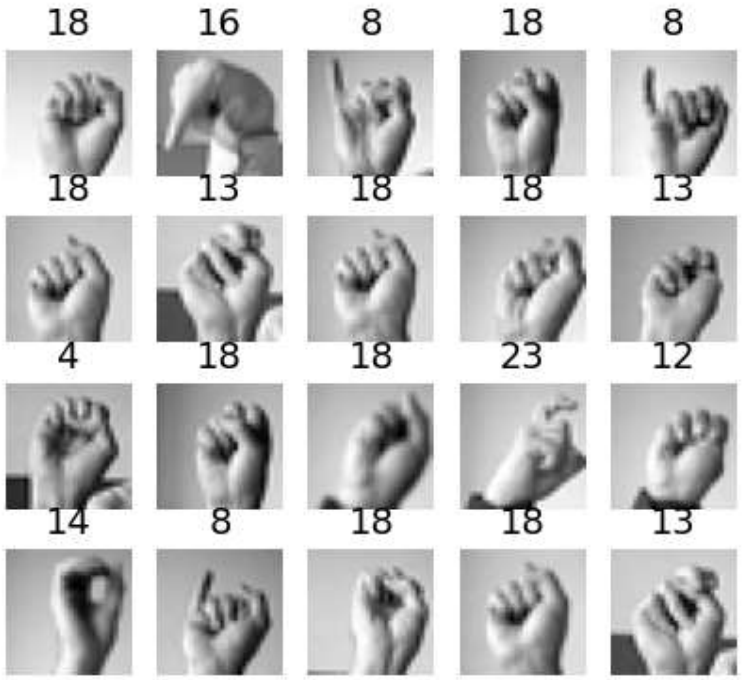
Cluster 10


19 16 3 16 5
2 23 2 11 14
2 19 19 11 16
14 16 11 23 24

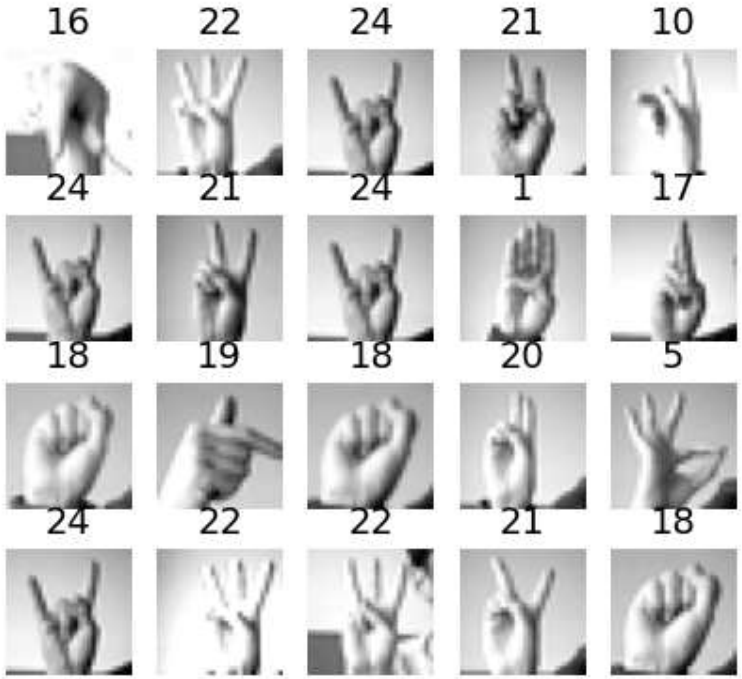
Cluster 11



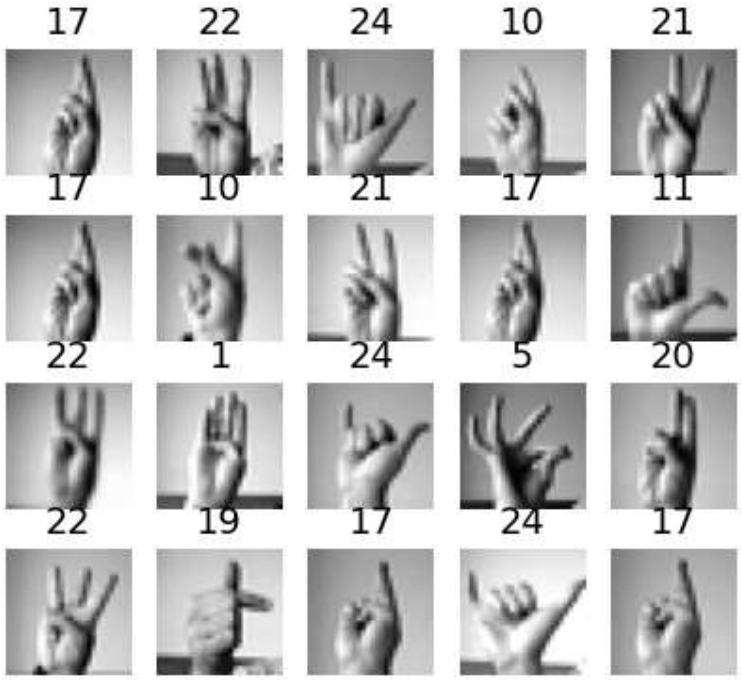
Cluster 12


18 16 8 18 8
18 13 18 18 13
4 18 18 23 12
14 8 18 18 13

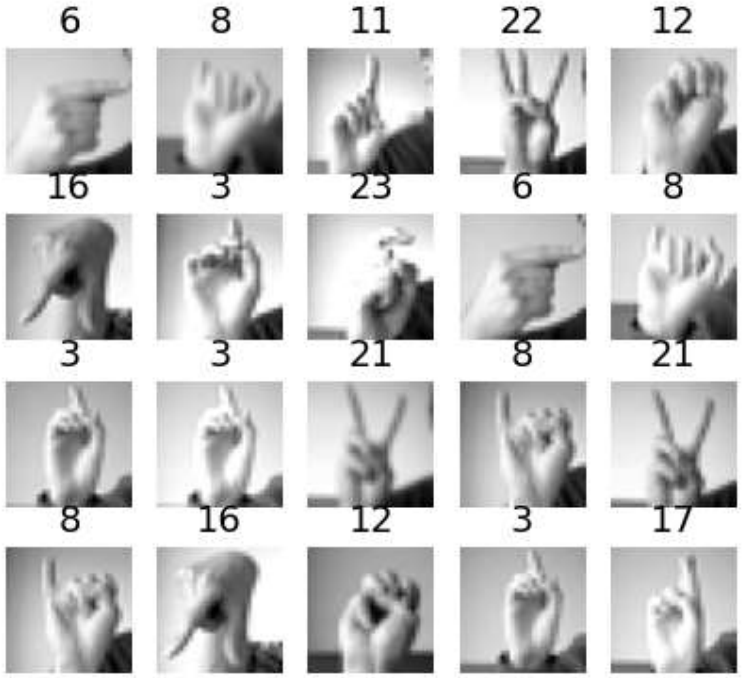
Cluster 13


16 22 24 21 10
24 21 24 1 17
18 19 18 20 5
24 22 22 21 18

Cluster 14



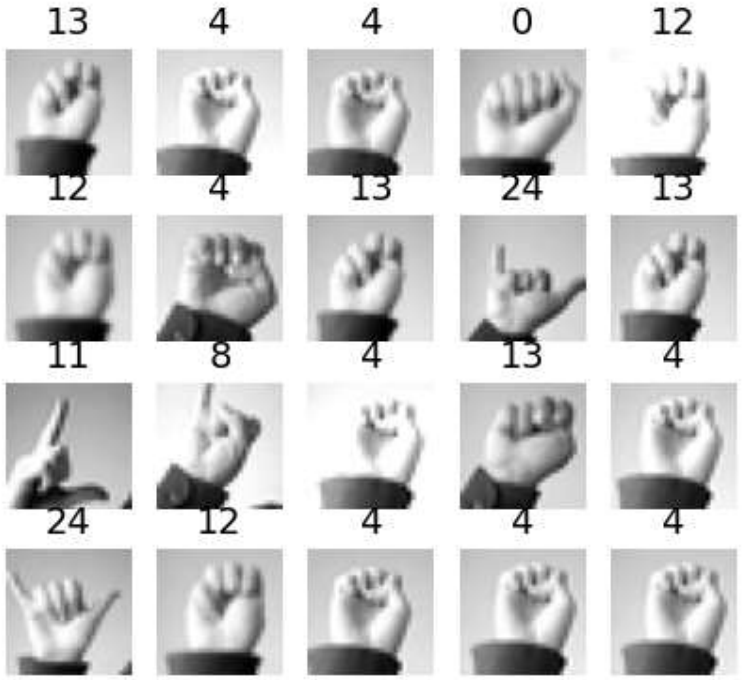
Cluster 15


6 8 11 22 12
16 3 23 6 8
3 3 21 8 21
8 16 12 3 17

Cluster 16


14 18 14 4 8
18 14 4 13 12
0 14 0 14 23
14 8 1 5 4

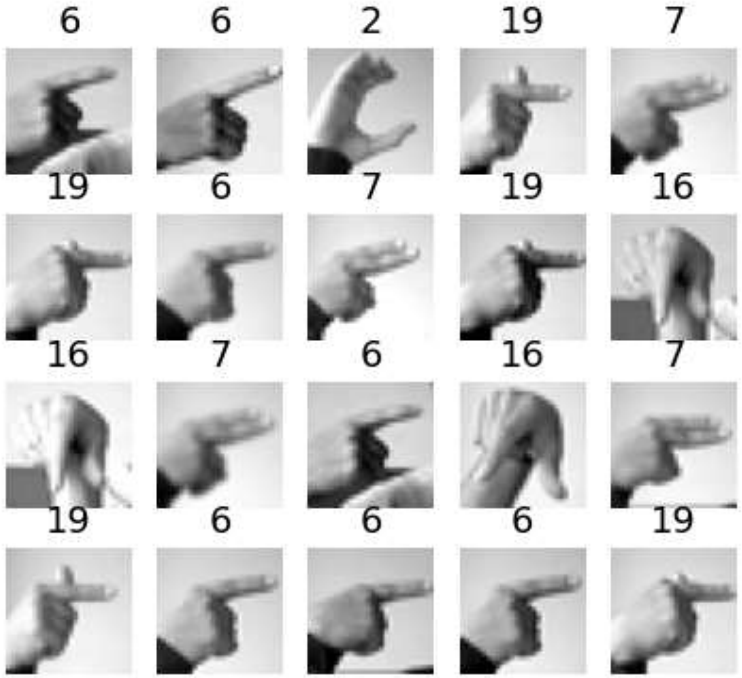
Cluster 17



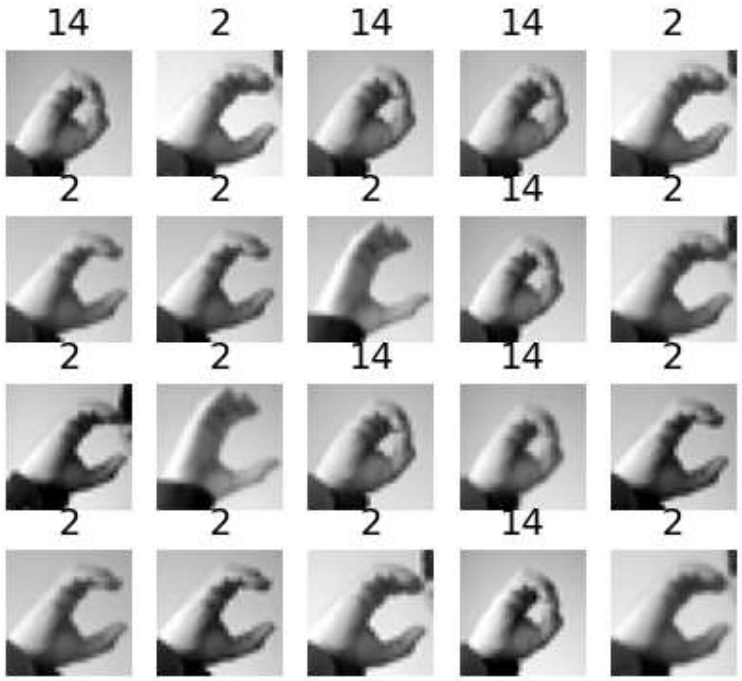
Cluster 18


22 24 10 16 22
5 23 24 13 10
8 5 17 3 21
18 22 10 22 16

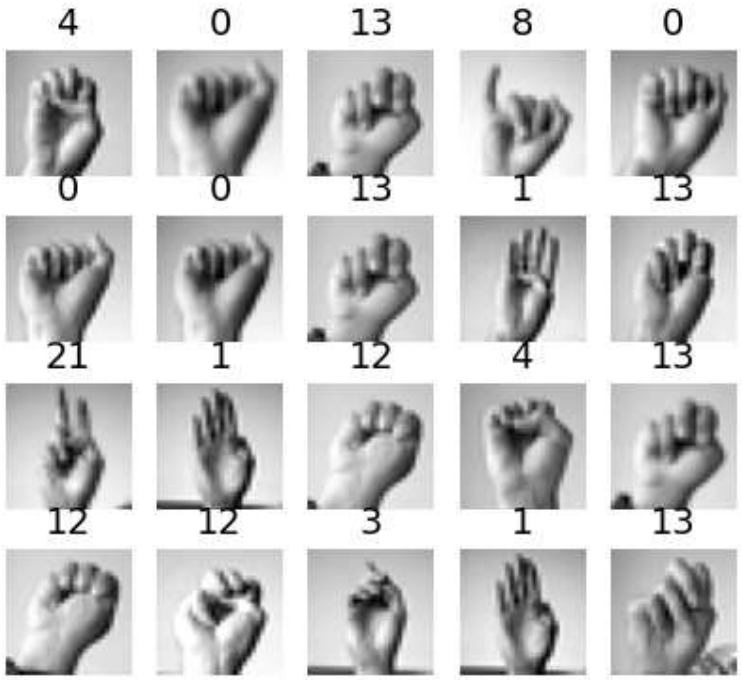
Cluster 19


6 6 2 19 7
19 6 7 19 16
16 7 6 16 7
19 6 6 6 19

Cluster 20



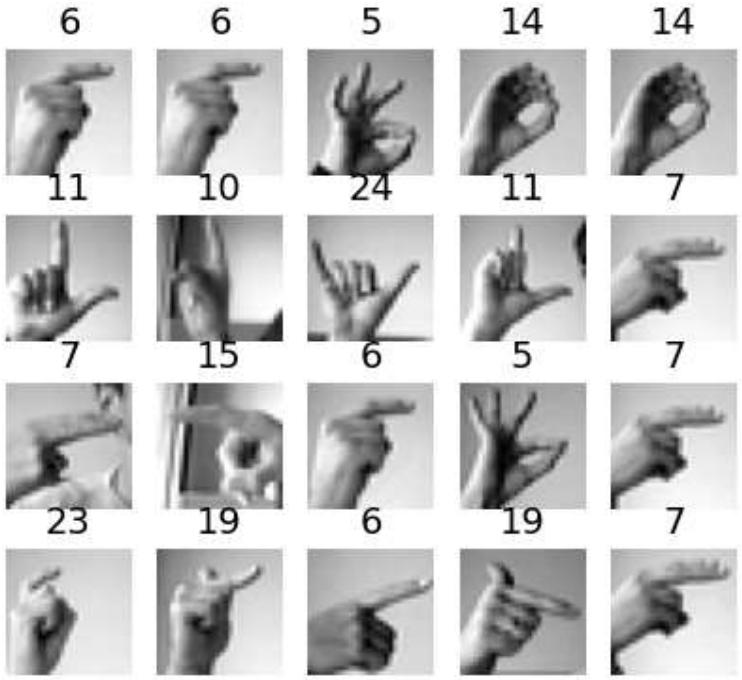
Cluster 21


4 0 13 8 0
0 0 13 1 13
21 1 12 4 13
12 12 3 1 13

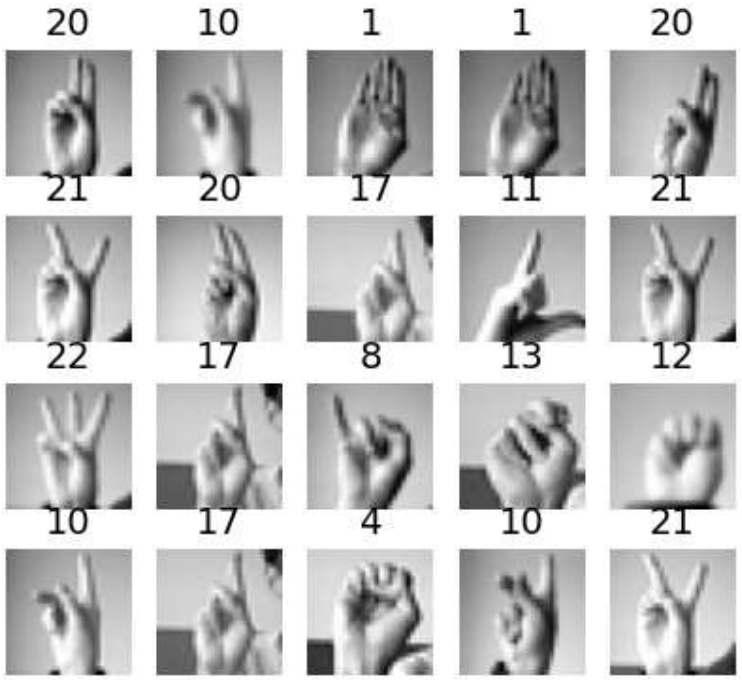
Cluster 22


10 22 22 23 17
23 3 22 23 3
22 22 3 22 22
5 20 4 20 24

Cluster 23



Cluster 24



Question: Do you see similar clothing items in each cluster?

There are certain clusters like cluster 12,13,16,20 which contains most of the similar clothing types. Also, if we take the generality into consideration, the K-means clustering has managed to cluster the items with similar clothing type together. However, the heels, sandles and shooes are footware and the algorithm has managed to cluster those together. But there are some clusters(cluster 3, cluster 6, cluster 7), which has incorrectly classified the data.

Question:

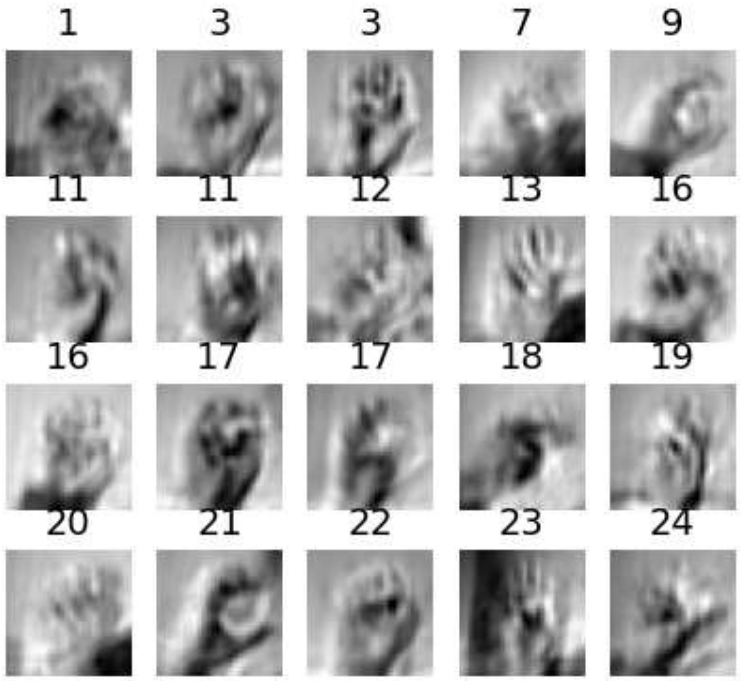
Use the model to generate 20 new clothing items (using the sample() method), and visualize them (since you used PCA, you will need to use its inverse_transform() method).

In [87]:
```
gmm_model = GaussianMixture(n_components=best_gmm_model.n_components, n_init=10, random_state=22)
```

In [88]:
```
y_pred = gmm_model.fit_predict(X_train_pca)
```

In [89]:
```
n_items = 20
gen_items_reduced, y_gen_items = gmm_model.sample(n_samples=n_items)
gen_items = pca.inverse_transform(gen_items_reduced)
```

In [90]:
```
plot_items(gen_items, y_gen_items)
```



Thus we have generated images using GMM

Question :

Build a fully connected (dense) feedforward neural network with two hidden layers using Keras (within Tensorflow) and train it on 50k Fashion MNIST training images. First hidden layer should contain 200 neurons and second hidden layer should contain 50 neurons. The hidden layers should have ReLU activation function. Train the network for 100 epochs. Plot training and validation loss and accuracy as a function of training epochs. Try three different learning rates of your choice (make the plots for each learning rate). [8 points]

In [91]:
```
# Split the dataset into training and validation data sets
split = StratifiedShuffleSplit(n_splits=1, test_size=1/6, random_state=22)
train_index, val_index = next(split.split(data, data_labels))

X_train = data.loc[train_index]
y_train = data_labels.loc[train_index]

X_valid = data.loc[val_index]
y_valid = data_labels.loc[val_index]

X_test = test_data
y_test = test_labels
```

In [92]:
```
print(X_train.shape, y_train.shape)
print(X_valid.shape, y_valid.shape)
print(X_test.shape, y_test.shape)
```

```
(22879, 784) (22879,)
(4576, 784) (4576,)
(7172, 784) (7172,)
```

In [93]:
```
from sklearn.decomposition import PCA

pca = PCA(0.95)
X_train_pca = pca.fit_transform(X_train)
X_valid_pca = pca.transform(X_valid)
X_test_pca = pca.transform(X_test)
```

In [94]:
```
pca.n_components_
```

Out[94]:
```
113
```

In [95]:
```
np.unique(y_test)
```

Out[95]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24])
```

In [96]:
```
def train_neural_net(model_name, X_train, y_train, X_valid, y_valid, learning_rate, activation_functions):
    tf.keras.backend.clear_session()
    tf.random.set_seed(42)
    if Path(model_name).is_dir():
        model = tf.keras.models.load_model(model_name)
        history=np.load(model_name+'\\trainHistory.npy',allow_pickle='TRUE').item()
    else:
        norm_layer = tf.keras.layers.Normalization(input_shape=X_train.shape[1:])
        model = tf.keras.Sequential([
        norm_layer,
```

```
        tf.keras.layers.Dense(200, activation=activation_functions["hidden_layer1"]),
        tf.keras.layers.Dense(50, activation=activation_functions["hidden_layer2"]),
        tf.keras.layers.Dense(25, activation=activation_functions["ouput_layer"])
        ])
        optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
        model.compile(loss=tf.keras.metrics.sparse_categorical_crossentropy, optimizer=optimizer, metrics=[tf.keras.metrics.spar
        norm_layer.adapt(X_train)
        history = model.fit(X_train, y_train, epochs=100, validation_data=(X_valid, y_valid))
        model.save(model_name, save_format="tf")
        np.save(model_name+'\\trainHistory.npy',history)
    return model, history
```

In [97]:
```
def plot_metrics(history):
    pd.DataFrame(history.history).plot(figsize=(8, 5))
    plt.grid(True)
    plt.gca().set_ylim(0, 1)
    plt.show()
```

In [98]:
```
def print_test_metrics(model, X_test, y_test, learning_rate):
    loss_test, accuracy_test = model.evaluate(X_test, y_test)
    print("\nFor learning Rate : ", learning_rate)
    print("\t Reported Loss on Test set : ", round(loss_test, 4))
    print("\t Reported accuracy on Test set : ", round(accuracy_test, 4))
```

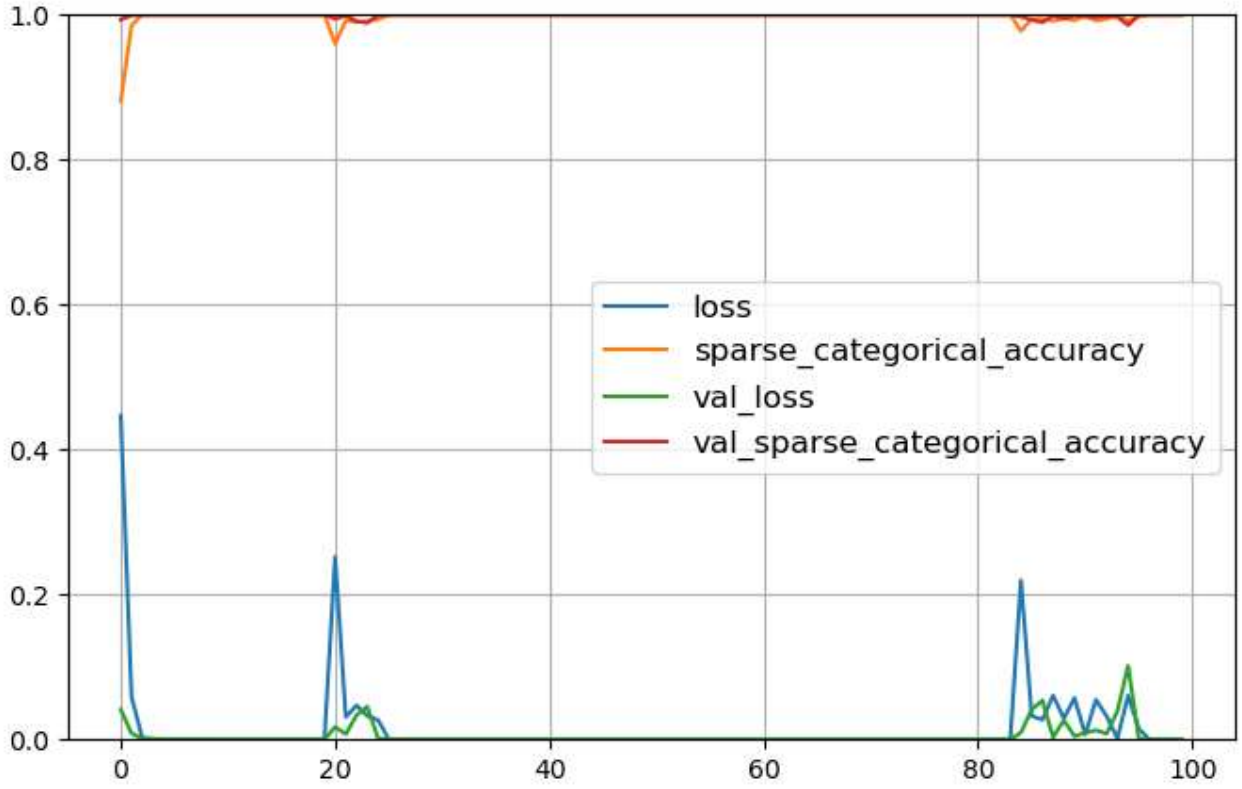
Using Learning rate as 0.001 i.e. 1e-3

In [99]:
```
activation_functions = {'hidden_layer1': 'relu',
                        'hidden_layer2': 'relu',
                        'ouput_layer': 'softmax'}
```

In [100…
```
%%capture
learning_rate = 1e-3
model, history = train_neural_net("neural_net_LR1e-03", X_train, y_train, X_valid, y_valid, learning_rate, activation_functions)
```

```
2023-04-01 03:05:57.493030: I tensorflow/compiler/xla/stream_executor/cuda/cuda_gpu_executor.cc:982] could not open file to read
NUMA node: /sys/bus/pci/devices/0000:01:00.0/numa_node
Your kernel may have been built without NUMA support.
2023-04-01 03:05:57.506916: W tensorflow/core/common_runtime/gpu/gpu_device.cc:1956] Cannot dlopen some GPU libraries. Please ma
ke sure the missing libraries mentioned above are installed properly if you would like to use GPU. Follow the guide at https://w
ww.tensorflow.org/install/gpu for how to download and setup the required libraries for your platform.
Skipping registering GPU devices...
INFO:tensorflow:Assets written to: neural_net_LR1e-03/assets
```

In [101…
```
plot_metrics(history)
```



In [102…
```
print_test_metrics(model, X_test, y_test, learning_rate)
```

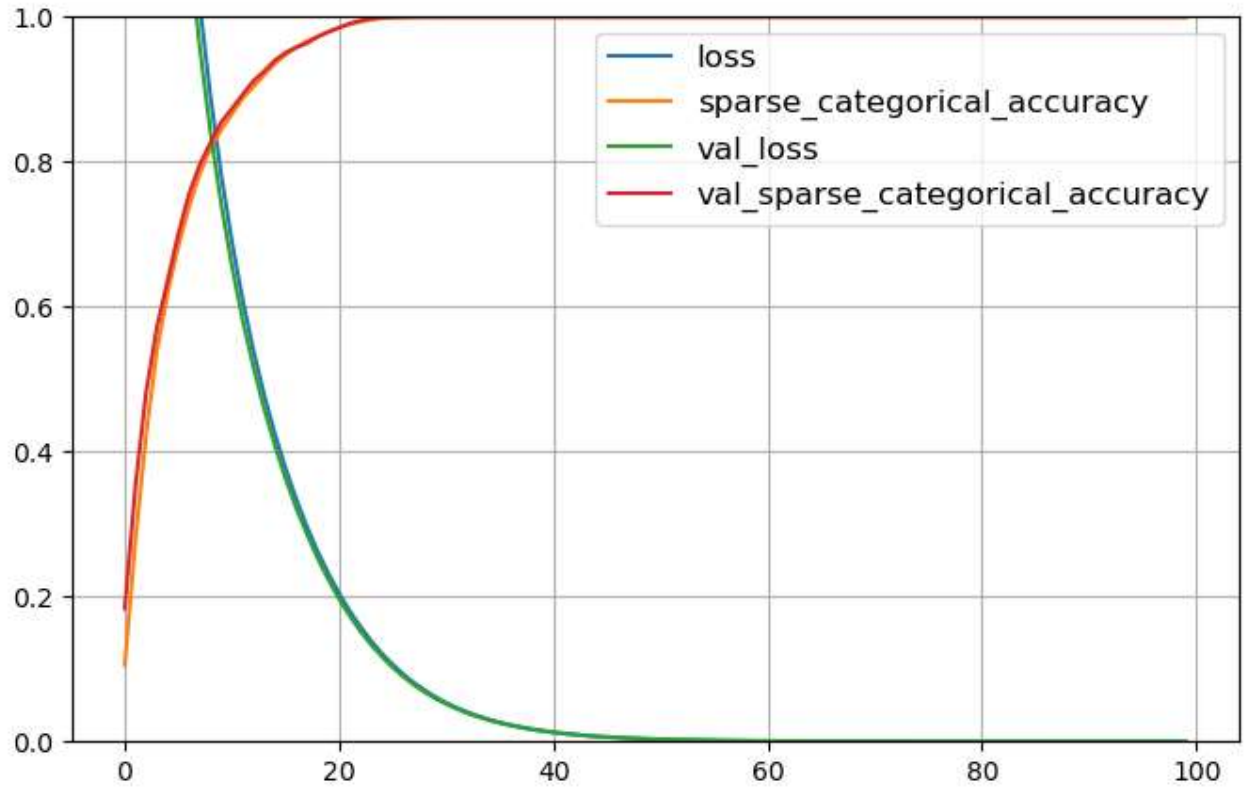
```
225/225 [==============================] - 0s 828us/step - loss: 3.6797 - sparse_categorical_accuracy: 0.7876

For learning Rate :  0.001
         Reported Loss on Test set :  3.6797
         Reported accuracy on Test set :  0.7876
```

Using Learning rate as 0.00001 i.e. 1e-5

In [103…
```
%%capture
learning_rate = 1e-5
model, history = train_neural_net("neural_net_LR1e-5", X_train, y_train, X_valid, y_valid, learning_rate, activation_functions)
```

```
INFO:tensorflow:Assets written to: neural_net_LR1e-5/assets
```

In [104…
```
plot_metrics(history)
```



In [105…
```
print_test_metrics(model, X_test, y_test, learning_rate)
```

```
225/225 [==============================] - 0s 811us/step - loss: 1.2073 - sparse_categorical_accuracy: 0.8070

For learning Rate :  1e-05
         Reported Loss on Test set :  1.2073
         Reported accuracy on Test set :  0.807
```

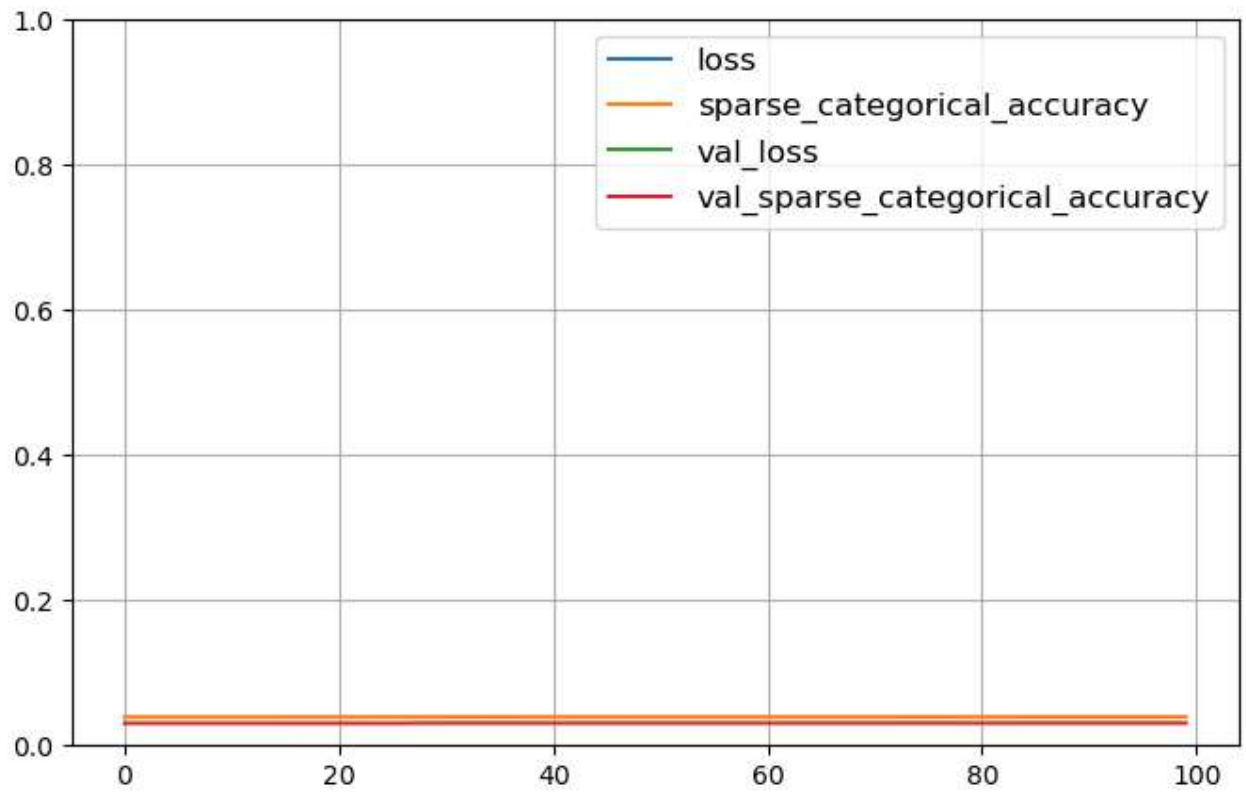
Using Learning rate as 0.000000001 i.e. 1e-9

In [106…
```
tf.keras.backend.clear_session()
tf.random.set_seed(42)
```

In [107…
```
%%capture
learning_rate = 1e-9
model, history = train_neural_net("neural_net_LR1e-9", X_train, y_train, X_valid, y_valid, learning_rate, activation_functions)
```

```
INFO:tensorflow:Assets written to: neural_net_LR1e-9/assets
```

In [108…
```
plot_metrics(history)
```



In [109…
```
print_test_metrics(model, X_test, y_test, learning_rate)
```

```
225/225 [==============================] - 0s 828us/step - loss: 3.4369 - sparse_categorical_accuracy: 0.0393

For learning Rate :  1e-09
         Reported Loss on Test set :  3.4369
         Reported accuracy on Test set :  0.0393
```

Question:

Run the network on the test portion of the dataset using best-performing learning rate and report loss and accuracy. [2 points] How many parameters does the network have? How many of those parameters are bias parameters?

The best performing learning rate is 0.00001 i.e. 1e-5 with reported loss = 120.73% Accuracy : 80.7%

The criteria used for selecting the best performing model for this example is to select model with minimal loss and highest accuracy

The model is trained under below architecture

In [110…
```
model = tf.keras.Sequential([
tf.keras.layers.Normalization(input_shape=X_train.shape[1:]),
tf.keras.layers.Dense(200, activation=activation_functions["hidden_layer1"]),
tf.keras.layers.Dense(50, activation=activation_functions["hidden_layer2"]),
tf.keras.layers.Dense(25, activation=activation_functions["ouput_layer"])
])
```

In [111…
```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 normalization_1 (Normalizat  (None, 784)              1569
 ion)

 dense_3 (Dense)             (None, 200)               157000

 dense_4 (Dense)             (None, 50)                10050

 dense_5 (Dense)             (None, 25)                1275

=================================================================
Total params: 169,894
Trainable params: 168,325
Non-trainable params: 1,569
_________________________________________________________________
```

Answers:

1. The total parameters available are 169129, out of those the trainable parameters are 167560 and non trainable parameters are 1569.
2. There are in total 200 bias parameters for hidden layer 1, 50 bias parameters for hidden layer 2 and 10 bias parameters for output layer, which makes in total 260 bias parameters.

Question:

Repeat everything from the previous step but make the hidden layers have linear activation functions. [5 points] Discuss how this impacts accuracy and why. [2 points]

Using Learning rate as 0.001 i.e. 1e-3

In [112…

```
linear_activation_functions = {'hidden_layer1': 'linear',
                               'hidden_layer2': 'linear',
                               'ouput_layer': 'softmax'}
```

In [113…

```
# %%capture
learning_rate = 1e-3
model, history = train_neural_net("neural_net_LR1e-3_linear", X_train, y_train, X_valid, y_valid, learning_rate, linear_activati
```

```
Epoch 1/100
715/715 [==============================] - 2s 2ms/step - loss: 0.6132 - sparse_categorical_accuracy: 0.8277 - val_loss: 0.2253 -
val_sparse_categorical_accuracy: 0.9307
Epoch 2/100
715/715 [==============================] - 1s 2ms/step - loss: 0.1625 - sparse_categorical_accuracy: 0.9512 - val_loss: 0.0673 -
val_sparse_categorical_accuracy: 0.9821
Epoch 3/100
715/715 [==============================] - 1s 2ms/step - loss: 0.1166 - sparse_categorical_accuracy: 0.9671 - val_loss: 0.4762 -
val_sparse_categorical_accuracy: 0.8918
Epoch 4/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0821 - sparse_categorical_accuracy: 0.9783 - val_loss: 0.0064 -
val_sparse_categorical_accuracy: 0.9991
Epoch 5/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0022 - sparse_categorical_accuracy: 1.0000 - val_loss: 0.0026 -
val_sparse_categorical_accuracy: 0.9998
Epoch 6/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0847 - sparse_categorical_accuracy: 0.9877 - val_loss: 1.4477 -
val_sparse_categorical_accuracy: 0.7815
Epoch 7/100
715/715 [==============================] - 1s 2ms/step - loss: 0.2050 - sparse_categorical_accuracy: 0.9593 - val_loss: 0.0137 -
val_sparse_categorical_accuracy: 0.9965
Epoch 8/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0139 - sparse_categorical_accuracy: 0.9965 - val_loss: 0.0041 -
val_sparse_categorical_accuracy: 0.9991
Epoch 9/100
715/715 [==============================] - 1s 2ms/step - loss: 0.1437 - sparse_categorical_accuracy: 0.9655 - val_loss: 0.0577 -
val_sparse_categorical_accuracy: 0.9880
Epoch 10/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0832 - sparse_categorical_accuracy: 0.9799 - val_loss: 0.0116 -
val_sparse_categorical_accuracy: 0.9965
Epoch 11/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0503 - sparse_categorical_accuracy: 0.9876 - val_loss: 0.1555 -
val_sparse_categorical_accuracy: 0.9659
Epoch 12/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0922 - sparse_categorical_accuracy: 0.9786 - val_loss: 0.0722 -
val_sparse_categorical_accuracy: 0.9784
Epoch 13/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0591 - sparse_categorical_accuracy: 0.9857 - val_loss: 0.0573 -
val_sparse_categorical_accuracy: 0.9851
Epoch 14/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0710 - sparse_categorical_accuracy: 0.9826 - val_loss: 0.0406 -
val_sparse_categorical_accuracy: 0.9908
Epoch 15/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0052 - sparse_categorical_accuracy: 0.9983 - val_loss: 0.0148 -
val_sparse_categorical_accuracy: 0.9965
Epoch 16/100
715/715 [==============================] - 1s 2ms/step - loss: 0.1406 - sparse_categorical_accuracy: 0.9741 - val_loss: 0.0118 -
val_sparse_categorical_accuracy: 0.9961
```

```
Epoch 17/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0210 - sparse_categorical_accuracy: 0.9947 - val_loss: 0.0521 -
val_sparse_categorical_accuracy: 0.9865
Epoch 18/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0750 - sparse_categorical_accuracy: 0.9851 - val_loss: 0.0153 -
val_sparse_categorical_accuracy: 0.9972
Epoch 19/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0808 - sparse_categorical_accuracy: 0.9855 - val_loss: 0.3895 -
val_sparse_categorical_accuracy: 0.9543
Epoch 20/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0407 - sparse_categorical_accuracy: 0.9926 - val_loss: 0.0043 -
val_sparse_categorical_accuracy: 0.9989
Epoch 21/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0219 - sparse_categorical_accuracy: 0.9959 - val_loss: 0.1166 -
val_sparse_categorical_accuracy: 0.9731
Epoch 22/100
715/715 [==============================] - 1s 2ms/step - loss: 0.1104 - sparse_categorical_accuracy: 0.9798 - val_loss: 0.0624 -
val_sparse_categorical_accuracy: 0.9869
Epoch 23/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0457 - sparse_categorical_accuracy: 0.9905 - val_loss: 0.0207 -
val_sparse_categorical_accuracy: 0.9939
Epoch 24/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0042 - sparse_categorical_accuracy: 0.9988 - val_loss: 0.0120 -
val_sparse_categorical_accuracy: 0.9967
Epoch 25/100
715/715 [==============================] - 1s 2ms/step - loss: 0.1246 - sparse_categorical_accuracy: 0.9787 - val_loss: 0.0016 -
val_sparse_categorical_accuracy: 0.9989
Epoch 26/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0791 - sparse_categorical_accuracy: 0.9875 - val_loss: 0.1180 -
val_sparse_categorical_accuracy: 0.9827
Epoch 27/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0183 - sparse_categorical_accuracy: 0.9969 - val_loss: 0.0054 -
val_sparse_categorical_accuracy: 0.9983
Epoch 28/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0838 - sparse_categorical_accuracy: 0.9880 - val_loss: 0.2311 -
val_sparse_categorical_accuracy: 0.9653
Epoch 29/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0372 - sparse_categorical_accuracy: 0.9923 - val_loss: 0.0772 -
val_sparse_categorical_accuracy: 0.9873
Epoch 30/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0064 - sparse_categorical_accuracy: 0.9987 - val_loss: 0.0046 -
val_sparse_categorical_accuracy: 0.9985
Epoch 31/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0840 - sparse_categorical_accuracy: 0.9869 - val_loss: 0.1883 -
val_sparse_categorical_accuracy: 0.9740
Epoch 32/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0616 - sparse_categorical_accuracy: 0.9900 - val_loss: 0.0130 -
val_sparse_categorical_accuracy: 0.9952
```

```
Epoch 33/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0323 - sparse_categorical_accuracy: 0.9955 - val_loss: 0.0899 -
val_sparse_categorical_accuracy: 0.9832
Epoch 34/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0564 - sparse_categorical_accuracy: 0.9892 - val_loss: 0.0048 -
val_sparse_categorical_accuracy: 0.9985
Epoch 35/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0385 - sparse_categorical_accuracy: 0.9947 - val_loss: 0.0728 -
val_sparse_categorical_accuracy: 0.9867
Epoch 36/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0527 - sparse_categorical_accuracy: 0.9914 - val_loss: 0.0028 -
val_sparse_categorical_accuracy: 0.9987
Epoch 37/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0478 - sparse_categorical_accuracy: 0.9920 - val_loss: 0.0792 -
val_sparse_categorical_accuracy: 0.9827
Epoch 38/100
715/715 [==============================] - 2s 2ms/step - loss: 0.0445 - sparse_categorical_accuracy: 0.9922 - val_loss: 0.0063 -
val_sparse_categorical_accuracy: 0.9985
Epoch 39/100
715/715 [==============================] - 2s 3ms/step - loss: 0.0550 - sparse_categorical_accuracy: 0.9912 - val_loss: 0.0266 -
val_sparse_categorical_accuracy: 0.9924
Epoch 40/100
715/715 [==============================] - 2s 3ms/step - loss: 0.0391 - sparse_categorical_accuracy: 0.9933 - val_loss: 0.0644 -
val_sparse_categorical_accuracy: 0.9897
Epoch 41/100
715/715 [==============================] - 2s 2ms/step - loss: 0.0388 - sparse_categorical_accuracy: 0.9932 - val_loss: 0.0662 -
val_sparse_categorical_accuracy: 0.9893
Epoch 42/100
715/715 [==============================] - 2s 2ms/step - loss: 0.0755 - sparse_categorical_accuracy: 0.9892 - val_loss: 0.0985 -
val_sparse_categorical_accuracy: 0.9816
Epoch 43/100
715/715 [==============================] - 2s 2ms/step - loss: 0.0413 - sparse_categorical_accuracy: 0.9922 - val_loss: 0.0145 -
val_sparse_categorical_accuracy: 0.9972
Epoch 44/100
715/715 [==============================] - 2s 2ms/step - loss: 0.0439 - sparse_categorical_accuracy: 0.9945 - val_loss: 0.2632 -
val_sparse_categorical_accuracy: 0.9644
Epoch 45/100
715/715 [==============================] - 2s 2ms/step - loss: 0.0499 - sparse_categorical_accuracy: 0.9913 - val_loss: 0.0098 -
val_sparse_categorical_accuracy: 0.9969
Epoch 46/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0491 - sparse_categorical_accuracy: 0.9934 - val_loss: 0.1058 -
val_sparse_categorical_accuracy: 0.9856
Epoch 47/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0386 - sparse_categorical_accuracy: 0.9937 - val_loss: 0.0615 -
val_sparse_categorical_accuracy: 0.9919
Epoch 48/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0797 - sparse_categorical_accuracy: 0.9899 - val_loss: 0.0302 -
val_sparse_categorical_accuracy: 0.9945
```

```
Epoch 49/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0205 - sparse_categorical_accuracy: 0.9963 - val_loss: 0.0109 -
val_sparse_categorical_accuracy: 0.9978
Epoch 50/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0112 - sparse_categorical_accuracy: 0.9980 - val_loss: 0.0124 -
val_sparse_categorical_accuracy: 0.9972
Epoch 51/100
715/715 [==============================] - 1s 2ms/step - loss: 0.1007 - sparse_categorical_accuracy: 0.9883 - val_loss: 0.0406 -
val_sparse_categorical_accuracy: 0.9904
Epoch 52/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0359 - sparse_categorical_accuracy: 0.9948 - val_loss: 0.0706 -
val_sparse_categorical_accuracy: 0.9910
Epoch 53/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0221 - sparse_categorical_accuracy: 0.9961 - val_loss: 0.0014 -
val_sparse_categorical_accuracy: 0.9996
Epoch 54/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0409 - sparse_categorical_accuracy: 0.9937 - val_loss: 0.1982 -
val_sparse_categorical_accuracy: 0.9755
Epoch 55/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0568 - sparse_categorical_accuracy: 0.9906 - val_loss: 0.0953 -
val_sparse_categorical_accuracy: 0.9886
Epoch 56/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0330 - sparse_categorical_accuracy: 0.9955 - val_loss: 0.0038 -
val_sparse_categorical_accuracy: 0.9987
Epoch 57/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0570 - sparse_categorical_accuracy: 0.9932 - val_loss: 0.1146 -
val_sparse_categorical_accuracy: 0.9847
Epoch 58/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0490 - sparse_categorical_accuracy: 0.9938 - val_loss: 0.0021 -
val_sparse_categorical_accuracy: 0.9993
Epoch 59/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0426 - sparse_categorical_accuracy: 0.9937 - val_loss: 0.0483 -
val_sparse_categorical_accuracy: 0.9913
Epoch 60/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0396 - sparse_categorical_accuracy: 0.9946 - val_loss: 0.1815 -
val_sparse_categorical_accuracy: 0.9889
Epoch 61/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0539 - sparse_categorical_accuracy: 0.9940 - val_loss: 0.4612 -
val_sparse_categorical_accuracy: 0.9611
Epoch 62/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0317 - sparse_categorical_accuracy: 0.9955 - val_loss: 0.0190 -
val_sparse_categorical_accuracy: 0.9965
Epoch 63/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0198 - sparse_categorical_accuracy: 0.9969 - val_loss: 0.0316 -
val_sparse_categorical_accuracy: 0.9948
Epoch 64/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0800 - sparse_categorical_accuracy: 0.9899 - val_loss: 0.0972 -
val_sparse_categorical_accuracy: 0.9889
```

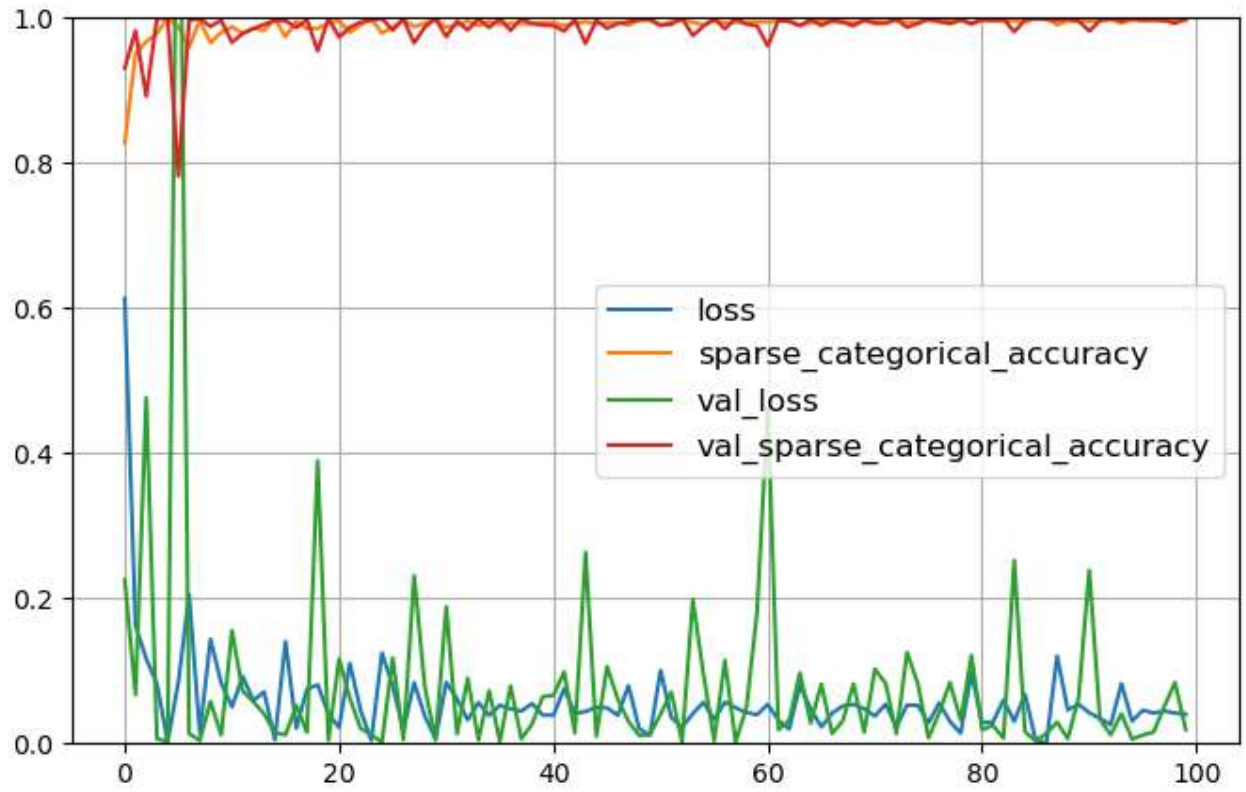
```
Epoch 65/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0476 - sparse_categorical_accuracy: 0.9931 - val_loss: 0.0277 -
val_sparse_categorical_accuracy: 0.9978
Epoch 66/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0227 - sparse_categorical_accuracy: 0.9961 - val_loss: 0.0820 -
val_sparse_categorical_accuracy: 0.9893
Epoch 67/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0416 - sparse_categorical_accuracy: 0.9944 - val_loss: 0.0137 -
val_sparse_categorical_accuracy: 0.9963
Epoch 68/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0515 - sparse_categorical_accuracy: 0.9941 - val_loss: 0.0312 -
val_sparse_categorical_accuracy: 0.9941
Epoch 69/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0540 - sparse_categorical_accuracy: 0.9937 - val_loss: 0.0822 -
val_sparse_categorical_accuracy: 0.9889
Epoch 70/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0478 - sparse_categorical_accuracy: 0.9948 - val_loss: 0.0156 -
val_sparse_categorical_accuracy: 0.9972
Epoch 71/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0377 - sparse_categorical_accuracy: 0.9950 - val_loss: 0.1025 -
val_sparse_categorical_accuracy: 0.9921
Epoch 72/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0542 - sparse_categorical_accuracy: 0.9942 - val_loss: 0.0827 -
val_sparse_categorical_accuracy: 0.9924
Epoch 73/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0232 - sparse_categorical_accuracy: 0.9970 - val_loss: 0.0135 -
val_sparse_categorical_accuracy: 0.9976
Epoch 74/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0529 - sparse_categorical_accuracy: 0.9937 - val_loss: 0.1252 -
val_sparse_categorical_accuracy: 0.9869
Epoch 75/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0525 - sparse_categorical_accuracy: 0.9942 - val_loss: 0.0823 -
val_sparse_categorical_accuracy: 0.9919
Epoch 76/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0284 - sparse_categorical_accuracy: 0.9967 - val_loss: 0.0076 -
val_sparse_categorical_accuracy: 0.9983
Epoch 77/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0557 - sparse_categorical_accuracy: 0.9941 - val_loss: 0.0424 -
val_sparse_categorical_accuracy: 0.9952
Epoch 78/100
715/715 [==============================] - 2s 2ms/step - loss: 0.0302 - sparse_categorical_accuracy: 0.9960 - val_loss: 0.0840 -
val_sparse_categorical_accuracy: 0.9919
Epoch 79/100
715/715 [==============================] - 2s 2ms/step - loss: 0.0141 - sparse_categorical_accuracy: 0.9981 - val_loss: 0.0322 -
val_sparse_categorical_accuracy: 0.9967
Epoch 80/100
715/715 [==============================] - 2s 2ms/step - loss: 0.0974 - sparse_categorical_accuracy: 0.9906 - val_loss: 0.1212 -
val_sparse_categorical_accuracy: 0.9921
```

```
Epoch 81/100
715/715 [==============================] - 2s 2ms/step - loss: 0.0287 - sparse_categorical_accuracy: 0.9966 - val_loss: 0.0186 -
val_sparse_categorical_accuracy: 0.9972
Epoch 82/100
715/715 [==============================] - 2s 2ms/step - loss: 0.0290 - sparse_categorical_accuracy: 0.9962 - val_loss: 0.0249 -
val_sparse_categorical_accuracy: 0.9967
Epoch 83/100
715/715 [==============================] - 2s 2ms/step - loss: 0.0599 - sparse_categorical_accuracy: 0.9951 - val_loss: 0.0074 -
val_sparse_categorical_accuracy: 0.9987
Epoch 84/100
715/715 [==============================] - 2s 2ms/step - loss: 0.0305 - sparse_categorical_accuracy: 0.9963 - val_loss: 0.2523 -
val_sparse_categorical_accuracy: 0.9806
Epoch 85/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0678 - sparse_categorical_accuracy: 0.9939 - val_loss: 0.0161 -
val_sparse_categorical_accuracy: 0.9974
Epoch 86/100
715/715 [==============================] - 2s 2ms/step - loss: 0.0050 - sparse_categorical_accuracy: 0.9994 - val_loss: 0.0037 -
val_sparse_categorical_accuracy: 0.9991
Epoch 87/100
715/715 [==============================] - 1s 2ms/step - loss: 2.1803e-04 - sparse_categorical_accuracy: 0.9999 - val_loss: 0.01
36 - val_sparse_categorical_accuracy: 0.9978
Epoch 88/100
715/715 [==============================] - 1s 2ms/step - loss: 0.1203 - sparse_categorical_accuracy: 0.9905 - val_loss: 0.0293 -
val_sparse_categorical_accuracy: 0.9965
Epoch 89/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0461 - sparse_categorical_accuracy: 0.9955 - val_loss: 0.0066 -
val_sparse_categorical_accuracy: 0.9991
Epoch 90/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0540 - sparse_categorical_accuracy: 0.9952 - val_loss: 0.0623 -
val_sparse_categorical_accuracy: 0.9956
Epoch 91/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0422 - sparse_categorical_accuracy: 0.9955 - val_loss: 0.2386 -
val_sparse_categorical_accuracy: 0.9816
Epoch 92/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0345 - sparse_categorical_accuracy: 0.9962 - val_loss: 0.0332 -
val_sparse_categorical_accuracy: 0.9963
Epoch 93/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0258 - sparse_categorical_accuracy: 0.9979 - val_loss: 0.0118 -
val_sparse_categorical_accuracy: 0.9987
Epoch 94/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0824 - sparse_categorical_accuracy: 0.9927 - val_loss: 0.0407 -
val_sparse_categorical_accuracy: 0.9961
Epoch 95/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0307 - sparse_categorical_accuracy: 0.9969 - val_loss: 0.0059 -
val_sparse_categorical_accuracy: 0.9989
Epoch 96/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0457 - sparse_categorical_accuracy: 0.9955 - val_loss: 0.0116 -
val_sparse_categorical_accuracy: 0.9978
```

```
Epoch 97/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0420 - sparse_categorical_accuracy: 0.9963 - val_loss: 0.0156 -
val_sparse_categorical_accuracy: 0.9978
Epoch 98/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0447 - sparse_categorical_accuracy: 0.9955 - val_loss: 0.0509 -
val_sparse_categorical_accuracy: 0.9961
Epoch 99/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0418 - sparse_categorical_accuracy: 0.9965 - val_loss: 0.0841 -
val_sparse_categorical_accuracy: 0.9924
Epoch 100/100
715/715 [==============================] - 1s 2ms/step - loss: 0.0403 - sparse_categorical_accuracy: 0.9964 - val_loss: 0.0192 -
val_sparse_categorical_accuracy: 0.9987
INFO:tensorflow:Assets written to: neural_net_LR1e-3_linear/assets
```

In [114…

```
plot_metrics(history)
```



In [115…

```
print_test_metrics(model, X_test, y_test, learning_rate)
```

```
225/225 [==============================] - 0s 819us/step - loss: 22.7573 - sparse_categorical_accuracy: 0.6464

For learning Rate :  0.001
         Reported Loss on Test set :  22.7573
         Reported accuracy on Test set :  0.6464
```

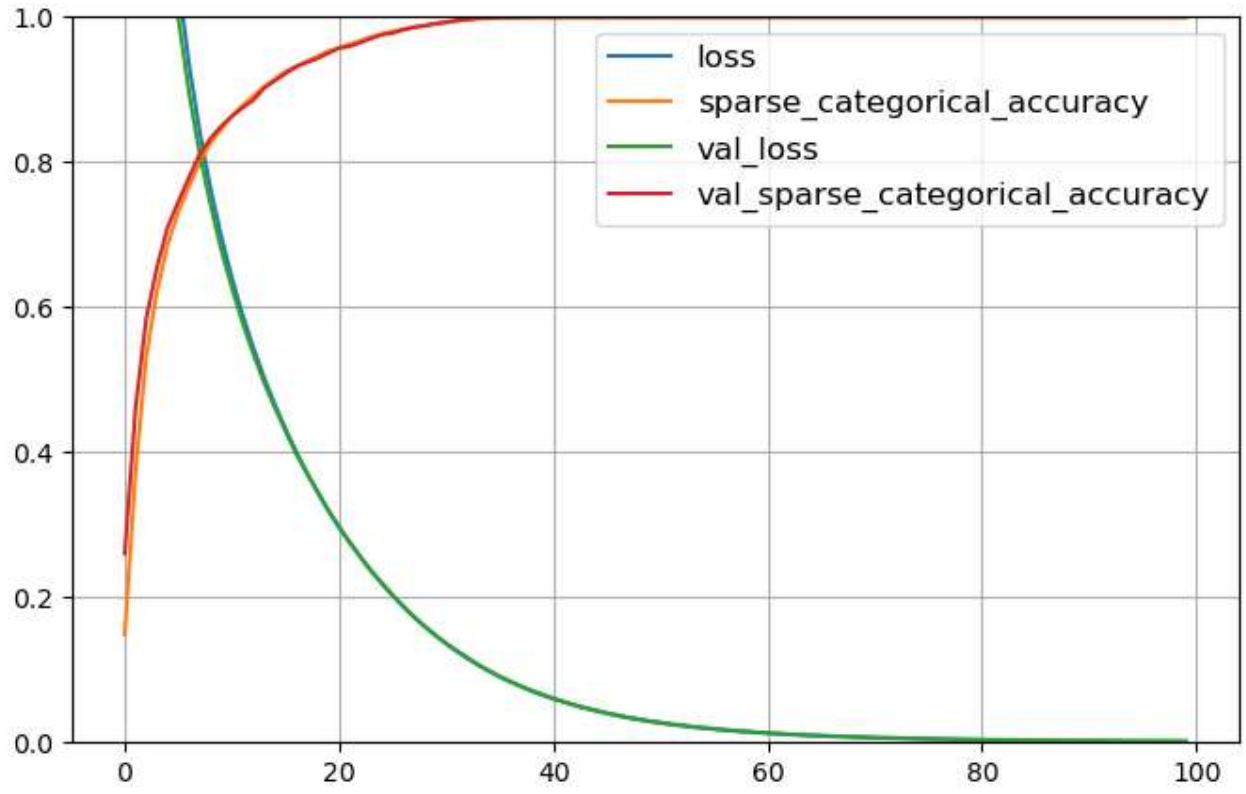
Using Learning rate as 0.00001 i.e. 1e-5

In [116…
```
tf.keras.backend.clear_session()
tf.random.set_seed(42)
```

In [117…
```
%%capture
learning_rate = 1e-5
model, history = train_neural_net("neural_net_LR1e-5_linear", X_train, y_train, X_valid, y_valid, learning_rate, linear_activati
```

```
INFO:tensorflow:Assets written to: neural_net_LR1e-5_linear/assets
```

In [118…
```
plot_metrics(history)
```



In [119…
```
print_test_metrics(model, X_test, y_test, learning_rate)
```

```
225/225 [==============================] - 0s 831us/step - loss: 2.2231 - sparse_categorical_accuracy: 0.6937

For learning Rate :  1e-05
         Reported Loss on Test set :  2.2231
         Reported accuracy on Test set :  0.6937
```

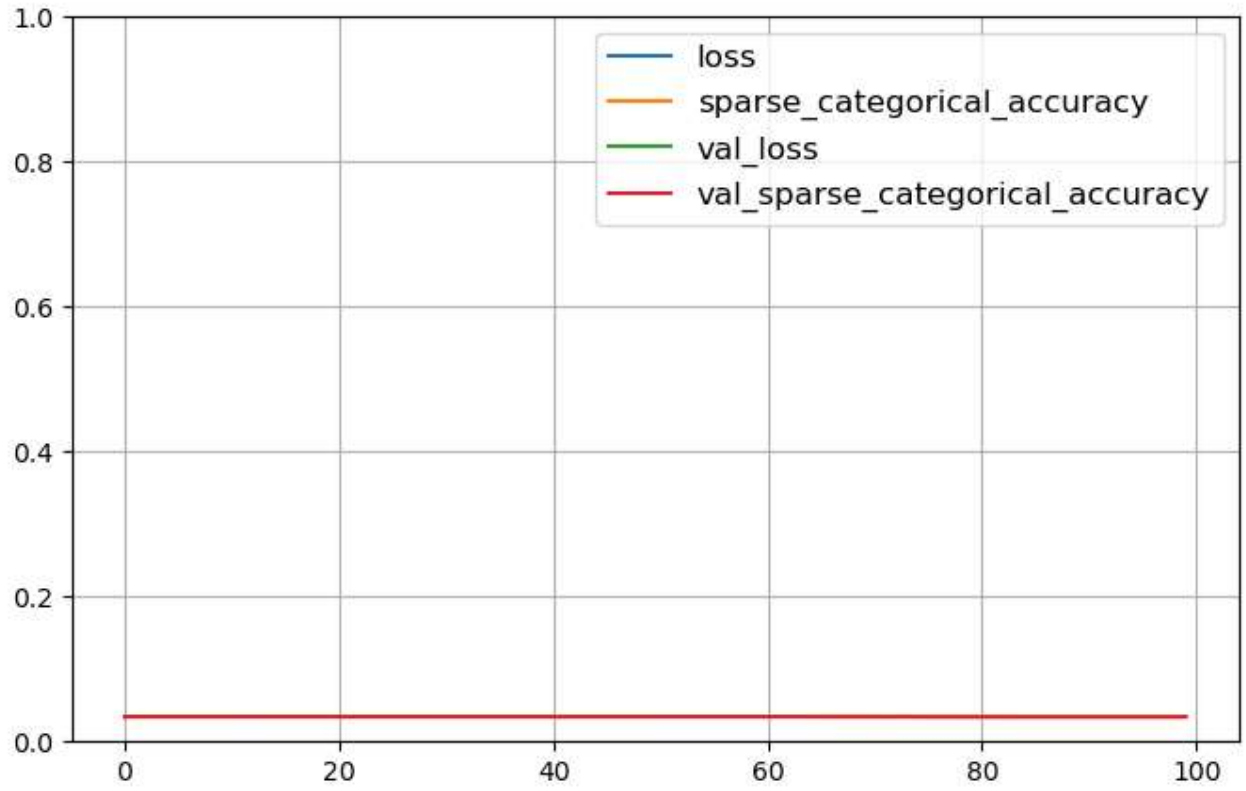
Using Learning rate as 0.000000001 i.e. 1e-9

In [120…
```
tf.keras.backend.clear_session()
tf.random.set_seed(42)
```

In [121…
```
%%capture
learning_rate = 1e-9
model, history = train_neural_net("neural_net_LR1e-9_linear", X_train, y_train, X_valid, y_valid, learning_rate, linear_activati
```

```
INFO:tensorflow:Assets written to: neural_net_LR1e-9_linear/assets
```

In [122…
```
plot_metrics(history)
```



In [123…

```
print_test_metrics(model, X_test, y_test, learning_rate)
```

```
225/225 [==============================] - 0s 844us/step - loss: 4.5873 - sparse_categorical_accuracy: 0.0371

For learning Rate :  1e-09
         Reported Loss on Test set :  4.5873
         Reported accuracy on Test set :  0.0371
```

There is no reliable model with linear activation function function as the loss is high in each case.

Question:

Discuss how using linear activation function impacts accuracy and why. [2 points]

Here, we have used linear activation function. This can be done using setting activation function as `None` or `Linear`.

The linear activation function is set for the 2 hidden layers of the neural network. We can observe following things:

For Learning rate as 0.001 i.e. 1e-3 :

ReLU activation : loss : 3.6797 Accuracy : 0.7876

Linear Activation: loss : 22.7573 Accuracy : 0.6464

For Learning rate as 0.00001 i.e. 1e-5 :

ReLU activation : loss : 1.2073 Accuracy : 0.807

Linear Activation: loss : 2.2231 Accuracy : 0.6937

For Learning rate as 0.000000001 i.e. 1e-9 :

ReLU activation : loss : 3.4369 Accuracy : 0.0393

Linear Activation: loss : 4.5873 Accuracy : 0.0371

Observations :

1. The model with ReLU performs better for the 2 learning rates (1e-3 and 1e-5) than a model with Linear activation. While for other learninng rates the loss is high. This might happen because of the neural network we have created is shallow.(contains only 2 hidden layers). If the network is deep enough, we might have been able to see some differences.
2. Also, as our dataset is MNIST like, one of the possibility is that the model might be learned the linear combination of features.
3. One more possibility is that, the ReLU might have overfitted on the training data causing it to give huge loss on the test data.
4. The loss in my case for both the models is high and more hyperparameter tuning is required to minimize the loss and imoprove accuracy

Acknowledgements:

We would like to thank Prof. Zoran Tiganj for his guidance on the plotting scatterplot using principle components for different dimentionality reduction methodologies. Also, we have worked according to his suggestion about following code on the github repository of Machine Learning Notebooks by A. Geron.

References :

[1] Geron, A. (2019). Hands-on machine learning with scikit-learn, keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems (2nd ed.). O'Reilly Media.

[2] Machine Learning Notebooks(8,9,10) on GitHub by A. Geron. https://github.com/ageron/handson-ml2 , https://github.com/ageron/handson-ml3

[2] API Reference. (n.d.). Scikit-learn. https://scikit-learn.org/stable/modules/classes.html

[3] API Reference. (n.d.). Pandas. https://pandas.pydata.org/docs/user_guide/index.html

[4] API Reference. (n.d.). NumPy. https://numpy.org/doc/stable/user/index.html#user

[5] How to save dictionaries : https://stackoverflow.com/questions/40219946/python-save-dictionaries-through-numpy-save