# Autoencoders Lab

visualizing HapMap phase 3 populations

**This is a solution template. Every chunk of code requiring your input will begin with the # TASK comment and all places where you should fill-in with your code are marked by ellipsis (...).**

## Stage 0 — getting the data

First, we will download the data from the linked Dropbox account. The code is hidden as it is not super important here. Double-click below if you are curious to see it!

In [ ]:
```
# Small config to select the GPU in our local machine
import tensorflow as tf

# GPU list
gpus = tf.config.list_physical_devices('GPU')

# Select GPU 1
tf.config.set_visible_devices(gpus[1], 'GPU')
```

In [ ]:
```
#@title Load raw data. Double click to see the code. { display-mode: "form" }

!wget https://www.dropbox.com/s/g7862q1l4ls9z3x/autosomal_5k_matrix.csv
!wget https://www.dropbox.com/s/3lv0062dw20qdqg/autosomal_5k_phenos.csv
!wget https://www.dropbox.com/s/6nzrusxkm536a5j/autosomal_5k_kinship.csv
```

Now, we will load the data and make sure they look as expected. Note, the genotypes per individual (row) are encoded as the count of minor alleles and thus can take values `gt = {0, 1, 2}`.

In [2]:
```
import numpy as np
import pandas as pd
import tensorflow as tf
import seaborn as sns
from tensorflow import keras
from tensorflow.keras import layers
from matplotlib import pyplot as plt

data = pd.read_csv("autosomal_5k_matrix.csv", header=0, index_col=0)
pheno = pd.read_csv("autosomal_5k_phenos.csv", header=0, index_col=0)

print(data)
print(pheno)
```

```
         rs416967  rs17013842  rs13052452  rs11049986  rs10994341  rs1504289  \
NA19919         2           1           0           1           1          0
NA19916         1           0           0           1           1          0
NA19835         0           0           1           1           0          1
NA20282         1           1           0           1           0          0
NA19703         0           0           0           0           1          1
...           ...         ...         ...         ...         ...        ...
NA19119         1           0           1           0           1          1
NA18860         1           0           0           1           1          0
NA19207         2           0           0           0           1          0
NA19103         1           0           1           0           0          0
NA19099         0           0           1           1           0          1

         rs882529  rs3885937  rs537330  rs9372090  ...  rs1558766  rs7818288  \
NA19919         0          1         1          1  ...          0          1
NA19916         1          1         2          1  ...          0          0
NA19835         0          2         2          1  ...          0          1
NA20282         0          0         1          1  ...          0          0
NA19703         0          0         0          2  ...          0          1
...           ...        ...       ...        ...  ...        ...        ...
NA19119         1          1         1          2  ...          0          0
NA18860         0          1         2          1  ...          1          1
NA19207         1          1         2          0  ...          0          0
NA19103         1          1         2          1  ...          0          0
NA19099         0          1         0          0  ...          0          1

         rs1051685  rs11223492  rs789492  rs6557516  rs7313246  rs317892  \
NA19919          0           0         1          0          1         1
NA19916          0           0         0          1          1         1
NA19835          0           0         0          0          2         2
NA20282          0           0         0          0          1         2
NA19703          1           0         1          1          1         1
...            ...         ...       ...        ...        ...       ...
NA19119          0           0         0          0          0         1
NA18860          0           0         0          0          1         0
NA19207          0           0         0          1          0         1
NA19103          0           0         0          0          2         1
NA19099          0           0         0          0          1         1

         rs11937009  rs2806497
NA19919           1          0
NA19916           1          0
NA19835           0          0
NA20282           0          0
NA19703           2          0
...             ...        ...
NA19119           0          0
NA18860           0          1
NA19207           1          0
NA19103           2          0
NA19099           0          0

[1184 rows x 5000 columns]
              id  sex   FID      dad      mom  pheno population
NA19919  NA19919    1  2427  NA19908  NA19909      0        ASW
NA19916  NA19916    1  2431        0        0      0        ASW
NA19835  NA19835    0  2424        0        0      0        ASW
NA20282  NA20282    0  2469        0        0      0        ASW
NA19703  NA19703    1  2368        0        0      0        ASW
...          ...  ...   ...      ...      ...    ...        ...
NA19119  NA19119    1  Y060        0        0      0        YRI
NA18860  NA18860    1  Y012  NA18859  NA18858      0        YRI
NA19207  NA19207    1  Y051        0        0      0        YRI
NA19103  NA19103    1  Y042  NA19101  NA19102      0        YRI
NA19099  NA19099    0  Y105        0        0      0        YRI

[1184 rows x 7 columns]
```

Now, we will create a dictionary and re-name our populations so that the names are a bit more informative:

In [3]:
```python
pop_dict = {'ASW':'African ancestry in SW USA',
            'CEU':'Utah residents with N and W European ancestry',
            'CHB':'Han Chinese in Beijing China',
            'CHD':'Chinese in Metropolitan Denver Colorado',
            'GIH':'Gujarati Indians in Houston Texas',
            'JPT':'Japanese in Tokyo Japan',
            'LWK':'Luhya in Webuye Kenya',
            'MEX':'Mexican ancestry in Los Angeles California',
            'MKK':'Maasai in Kinyawa Kenya',
            'TSI':'Toscans in Italy',
            'YRI':'Yoruba in Ibadan Nigeria'}
pheno2 = pheno.replace({"population": pop_dict})
pheno2
```

Out[3]:

| | id | sex | FID | dad | mom | pheno | population |
|---|---|---|---|---|---|---|---|
| **NA19919** | NA19919 | 1 | 2427 | NA19908 | NA19909 | 0 | African ancestry in SW USA |
| **NA19916** | NA19916 | 1 | 2431 | 0 | 0 | 0 | African ancestry in SW USA |
| **NA19835** | NA19835 | 0 | 2424 | 0 | 0 | 0 | African ancestry in SW USA |
| **NA20282** | NA20282 | 0 | 2469 | 0 | 0 | 0 | African ancestry in SW USA |
| **NA19703** | NA19703 | 1 | 2368 | 0 | 0 | 0 | African ancestry in SW USA |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **NA19119** | NA19119 | 1 | Y060 | 0 | 0 | 0 | Yoruba in Ibadan Nigeria |
| **NA18860** | NA18860 | 1 | Y012 | NA18859 | NA18858 | 0 | Yoruba in Ibadan Nigeria |
| **NA19207** | NA19207 | 1 | Y051 | 0 | 0 | 0 | Yoruba in Ibadan Nigeria |
| **NA19103** | NA19103 | 1 | Y042 | NA19101 | NA19102 | 0 | Yoruba in Ibadan Nigeria |
| **NA19099** | NA19099 | 0 | Y105 | 0 | 0 | 0 | Yoruba in Ibadan Nigeria |

1184 rows × 7 columns

In [4]:
```python
# === TASK 1 ===
print("Example loci / marker names:")
print(data.columns[:10])

print("Total number of individuals:", len(pheno2))

print("\nNumber of individuals per population:")
print(pheno2['population'].value_counts())

print("\nSex distribution (1 = male, 0 = female):")
print(pheno2['sex'].value_counts())

print("\nSex distribution per population:")
print(pheno2.groupby('population')['sex'].value_counts())

# Family structure
family_sizes = pheno2['FID'].value_counts()
num_families = family_sizes.shape[0]

print("\nTotal number of families:", num_families)
print("\nFamily size statistics:")
print(family_sizes.describe())

# Estimation of siblings structure
full_sibs = family_sizes[family_sizes >= 3].count()
half_sibs = family_sizes[family_sizes == 2].count()

print("\nEstimated number of families with full-siblings (size >= 3):", full_sibs)
print("Estimated number of families with half-siblings (size = 2):", half_sibs)
```

```
Example loci / marker names:
Index(['rs416967', 'rs17013842', 'rs13052452', 'rs11049986', 'rs10994341',
       'rs1504289', 'rs882529', 'rs3885937', 'rs537330', 'rs9372090'],
      dtype='object')
Total number of individuals: 1184

Number of individuals per population:
population
Maasai in Kinyawa Kenya                           171
Yoruba in Ibadan Nigeria                          167
Utah residents with N and W European ancestry     165
Luhya in Webuye Kenya                              90
Gujarati Indians in Houston Texas                  88
Toscans in Italy                                   88
Japanese in Tokyo Japan                            86
Chinese in Metropolitan Denver Colorado            85
Han Chinese in Beijing China                       84
African ancestry in SW USA                         83
Mexican ancestry in Los Angeles California         77
Name: count, dtype: int64

Sex distribution (1 = male, 0 = female):
sex
0    595
1    589
Name: count, dtype: int64

Sex distribution per population:
population                                     sex
African ancestry in SW USA                     0      45
                                               1      38
Chinese in Metropolitan Denver Colorado        0      44
                                               1      41
Gujarati Indians in Houston Texas              1      45
                                               0      43
Han Chinese in Beijing China                   0      42
                                               1      42
Japanese in Tokyo Japan                        1      44
                                               0      42
Luhya in Webuye Kenya                          0      45
                                               1      45
Maasai in Kinyawa Kenya                        1      86
                                               0      85
Mexican ancestry in Los Angeles California     0      43
                                               1      34
Toscans in Italy                               0      44
                                               1      44
Utah residents with N and W European ancestry  0      85
                                               1      80
Yoruba in Ibadan Nigeria                       1      90
                                               0      77
Name: count, dtype: int64

Total number of families: 770

Family size statistics:
count    770.000000
mean       1.537662
std        1.033514
min        1.000000
25%        1.000000
50%        1.000000
75%        2.000000
max        6.000000
Name: count, dtype: float64

Estimated number of families with full-siblings (size >= 3): 147
Estimated number of families with half-siblings (size = 2): 60
```

In [5]:

```
# TASK Scaling
# We need to scale our counts data so that it is bound between 0 and 1.
```

```python
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
geno_data = pd.DataFrame(scaler.fit_transform(data), index=data.index, columns=data.columns)
geno_data
```

Out[5]:

| | rs416967 | rs17013842 | rs13052452 | rs11049986 | rs10994341 | rs1504289 | rs882529 | rs38859 |
|---|---|---|---|---|---|---|---|---|
| **NA19919** | 1.0 | 0.5 | 0.0 | 0.5 | 0.5 | 0.0 | 0.0 | ( |
| **NA19916** | 0.5 | 0.0 | 0.0 | 0.5 | 0.5 | 0.0 | 0.5 | ( |
| **NA19835** | 0.0 | 0.0 | 0.5 | 0.5 | 0.0 | 0.5 | 0.0 | ' |
| **NA20282** | 0.5 | 0.5 | 0.0 | 0.5 | 0.0 | 0.0 | 0.0 | ( |
| **NA19703** | 0.0 | 0.0 | 0.0 | 0.0 | 0.5 | 0.5 | 0.0 | ( |
| **...** | ... | ... | ... | ... | ... | ... | ... | |
| **NA19119** | 0.5 | 0.0 | 0.5 | 0.0 | 0.5 | 0.5 | 0.5 | ( |
| **NA18860** | 0.5 | 0.0 | 0.0 | 0.5 | 0.5 | 0.0 | 0.0 | ( |
| **NA19207** | 1.0 | 0.0 | 0.0 | 0.0 | 0.5 | 0.0 | 0.5 | ( |
| **NA19103** | 0.5 | 0.0 | 0.5 | 0.0 | 0.0 | 0.0 | 0.5 | ( |
| **NA19099** | 0.0 | 0.0 | 0.5 | 0.5 | 0.0 | 0.5 | 0.0 | ( |

1184 rows × 5000 columns

In [6]:

```python
# TASK Randomly split into the training and the validation set, so that 80 per-cent of individ

train = geno_data.sample(frac=0.8, random_state=42)  # frac=0.8 → 80% training
test = geno_data.drop(train.index)
train = train.reset_index(drop=True)
test = test.reset_index(drop=True)

# TASK Print some info about the resulting split
print("Total number of individuals:", len(geno_data))
print("\t - training set:", len(train))
print("\t - test set:", len(test))
```

```
Total number of individuals: 1184
         - training set: 947
         - test set: 237
```

In [7]:

```python
# TASK Specify the autoencoder model

train_tensor = train.to_numpy()
print(type(train_tensor))

# TASK Hyperparameters
# use ReLu activations, ADAM optimizer and
# mean squared error as the loss function
hp_loss_fn = 'mse'
hp_act_fn = 'relu'
hp_optimizer = 'adam'
hp_metrics = ['mse','mae','mape']

input_data = keras.Input(shape = (train_tensor.shape[1],))

# TASK Define architecture of the encoder:
# the second layer should be a batch normalization

# Baseline
def Encoder_bl(input):
  # Encoder
  layer1 = layers.Dense(units = 512, activation = hp_act_fn, name='layer1')(input)
  layer2 = layers.BatchNormalization(name='layer2')(layer1)  # batch norm
  layer3 = layers.Dropout(rate = 0.05, name='layer3')(layer2)
  layer4 = layers.Dense(units = 128, activation = hp_act_fn, name='layer4')(layer3)
  layer5 = layers.Dropout(rate = 0.025, name='layer5')(layer4)
  layer6 = layers.Dense(units = 25, activation = hp_act_fn, name='layer6')(layer5)
  bottleneck = layers.Dense(units = 2, name='layer_bottleneck')(layer6)
  return(bottleneck)

# TASK Look at the encoder, complete the decoder function

def Decoder_bl(bottleneck):
  # Decoder
  layer7 = layers.Dense(units = 128, activation = hp_act_fn)(bottleneck)
  layer8 = layers.Dropout(rate=0.025, name='layer8')(layer7)
  layer9 = layers.Dense(units=512, activation=hp_act_fn, name='layer9')(layer8)
  layer10 = layers.Dropout(rate=0.05, name='layer10')(layer9)
  layer11 = layers.Dense(units=1500, activation=hp_act_fn, name='layer11')(layer10)
  layer12 = layers.Dense(units = train_tensor.shape[1], activation = 'sigmoid')(layer11)
  return(layer12)

def Autoencoder_bl(input):
  enc = Encoder_bl(input)
  autoenc = Decoder_bl(enc)
  return(autoenc)

# Smaller
def Encoder_sm(input):
  # Encoder
  layer1 = layers.Dense(units = 512, activation = hp_act_fn, name='layer1')(input)
  layer2 = layers.BatchNormalization(name='layer2')(layer1)  # batch norm
  layer3 = layers.Dropout(rate = 0.05, name='layer3')(layer2)
  layer4 = layers.Dense(units = 128, activation = hp_act_fn, name='layer4')(layer3)
  bottleneck = layers.Dense(units = 2, name='layer_bottleneck')(layer4)
  return(bottleneck)

def Decoder_sm(bottleneck):
  # Decoder
  layer7 = layers.Dense(units = 128, activation = hp_act_fn)(bottleneck)
  layer8 = layers.Dropout(rate=0.025, name='layer8')(layer7)
  layer9 = layers.Dense(units=512, activation=hp_act_fn, name='layer9')(layer8)
  layer10 = layers.Dense(units = train_tensor.shape[1], activation = 'sigmoid')(layer9)
  return(layer10)

def Autoencoder_sm(input):
  enc = Encoder_sm(input)
  autoenc = Decoder_sm(enc)
  return(autoenc)

#Bigger
def Encoder_bg(input):
  # Encoder
  layer1 = layers.Dense(units = 2000, activation = hp_act_fn, name='layer1')(input)
  layer2 = layers.BatchNormalization(name='layer2')(layer1)  # batch norm
  layer3 = layers.Dropout(rate = 0.05, name='layer3')(layer2)
  layer4 = layers.Dense(units = 500, activation = hp_act_fn, name='layer4')(layer3)
  layer5 = layers.Dropout(rate = 0.025, name='layer5')(layer4)
  layer6 = layers.Dense(units = 50, activation = hp_act_fn, name='layer6')(layer5)
  bottleneck = layers.Dense(units = 2, name='layer_bottleneck')(layer6)
  return(bottleneck)

def Decoder_bg(bottleneck):
  # Decoder
  layer7 = layers.Dense(units = 50, activation = hp_act_fn)(bottleneck)
  layer8 = layers.Dropout(rate=0.025, name='layer8')(layer7)
  layer9 = layers.Dense(units=500, activation=hp_act_fn, name='layer9')(layer8)
  layer10 = layers.Dropout(rate=0.05, name='layer10')(layer9)
  layer11 = layers.Dense(units=2000, activation=hp_act_fn, name='layer11')(layer10)
  layer12 = layers.Dense(units = train_tensor.shape[1], activation = 'sigmoid')(layer11)
  return(layer12)

def Autoencoder_bg(input):
  enc = Encoder_bg(input)
  autoenc = Decoder_bg(enc)
  return(autoenc)
```

```
autoencoder_model_bl = keras.Model(inputs = input_data, outputs = Autoencoder_bl(input_data))
autoencoder_model_bl.compile(
  loss = hp_loss_fn,
  optimizer = hp_optimizer,
  metrics = hp_metrics
)

autoencoder_model_sm = keras.Model(inputs = input_data, outputs = Autoencoder_sm(input_data))
autoencoder_model_sm.compile(
  loss = hp_loss_fn,
  optimizer = hp_optimizer,
  metrics = hp_metrics
)
autoencoder_model_bg = keras.Model(inputs = input_data, outputs = Autoencoder_bg(input_data))
autoencoder_model_bg.compile(
  loss = hp_loss_fn,
  optimizer = hp_optimizer,
  metrics = hp_metrics
)

# TASK Visualise the created architecture and summarise its parameters
autoencoder_model_bl.summary()
autoencoder_model_sm.summary()
autoencoder_model_bg.summary()
```

```
<class 'numpy.ndarray'>
```

```
I0000 00:00:1764239057.822997 2332417 gpu_device.cc:2019] Created device /job:localhost/replic
a:0/task:0/device:GPU:0 with 46761 MB memory:  -> device: 1, name: NVIDIA RTX A6000, pci bus i
d: 0000:b3:00.0, compute capability: 8.6
```

```
Model: "functional"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━┓
┃ Layer (type)                     ┃ Output Shape           ┃        Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━┩
│ input_layer (InputLayer)         │ (None, 5000)           │              0 │
├──────────────────────────────────┼────────────────────────┼────────────────┤
│ layer1 (Dense)                   │ (None, 512)            │      2,560,512 │
├──────────────────────────────────┼────────────────────────┼────────────────┤
│ layer2 (BatchNormalization)      │ (None, 512)            │          2,048 │
├──────────────────────────────────┼────────────────────────┼────────────────┤
│ layer3 (Dropout)                 │ (None, 512)            │              0 │
├──────────────────────────────────┼────────────────────────┼────────────────┤
│ layer4 (Dense)                   │ (None, 128)            │         65,664 │
├──────────────────────────────────┼────────────────────────┼────────────────┤
│ layer5 (Dropout)                 │ (None, 128)            │              0 │
├──────────────────────────────────┼────────────────────────┼────────────────┤
│ layer6 (Dense)                   │ (None, 25)             │          3,225 │
├──────────────────────────────────┼────────────────────────┼────────────────┤
│ layer_bottleneck (Dense)         │ (None, 2)              │             52 │
├──────────────────────────────────┼────────────────────────┼────────────────┤
│ dense (Dense)                    │ (None, 128)            │            384 │
├──────────────────────────────────┼────────────────────────┼────────────────┤
│ layer8 (Dropout)                 │ (None, 128)            │              0 │
├──────────────────────────────────┼────────────────────────┼────────────────┤
│ layer9 (Dense)                   │ (None, 512)            │         66,048 │
├──────────────────────────────────┼────────────────────────┼────────────────┤
│ layer10 (Dropout)                │ (None, 512)            │              0 │
├──────────────────────────────────┼────────────────────────┼────────────────┤
│ layer11 (Dense)                  │ (None, 1500)           │        769,500 │
├──────────────────────────────────┼────────────────────────┼────────────────┤
│ dense_1 (Dense)                  │ (None, 5000)           │      7,505,000 │
└──────────────────────────────────┴────────────────────────┴────────────────┘

 Total params: 10,972,433 (41.86 MB)
 Trainable params: 10,971,409 (41.85 MB)
 Non-trainable params: 1,024 (4.00 KB)
Model: "functional_1"
```

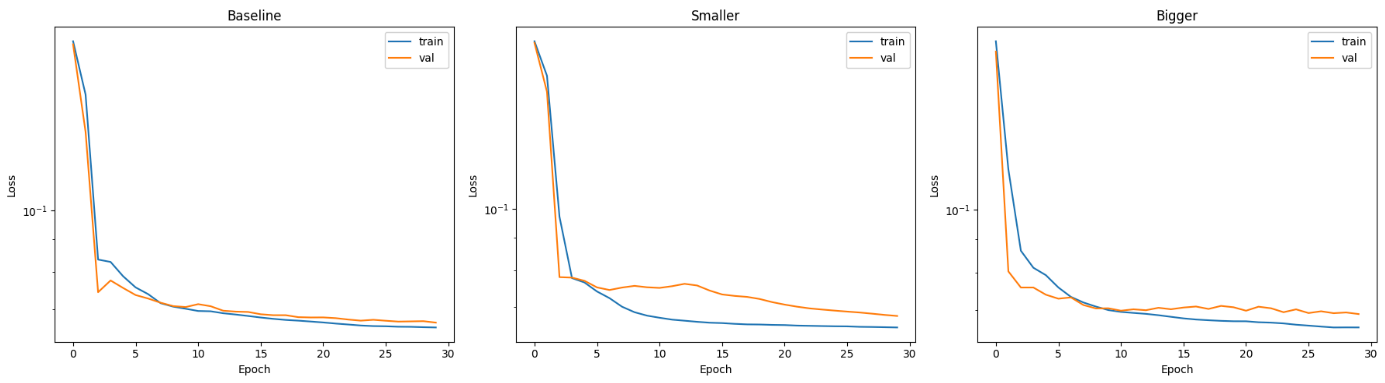
```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                     ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ input_layer (InputLayer)         │ (None, 5000)           │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ layer1 (Dense)                   │ (None, 512)            │     2,560,512 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ layer2 (BatchNormalization)      │ (None, 512)            │         2,048 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ layer3 (Dropout)                 │ (None, 512)            │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ layer4 (Dense)                   │ (None, 128)            │        65,664 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ layer_bottleneck (Dense)         │ (None, 2)              │           258 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_2 (Dense)                  │ (None, 128)            │           384 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ layer8 (Dropout)                 │ (None, 128)            │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ layer9 (Dense)                   │ (None, 512)            │        66,048 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_3 (Dense)                  │ (None, 5000)           │     2,565,000 │
└──────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 5,259,914 (20.06 MB)
 Trainable params: 5,258,890 (20.06 MB)
 Non-trainable params: 1,024 (4.00 KB)
Model: "functional_2"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                     ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ input_layer (InputLayer)         │ (None, 5000)           │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ layer1 (Dense)                   │ (None, 2000)           │    10,002,000 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ layer2 (BatchNormalization)      │ (None, 2000)           │         8,000 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ layer3 (Dropout)                 │ (None, 2000)           │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ layer4 (Dense)                   │ (None, 500)            │     1,000,500 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ layer5 (Dropout)                 │ (None, 500)            │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ layer6 (Dense)                   │ (None, 50)             │        25,050 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ layer_bottleneck (Dense)         │ (None, 2)              │           102 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_4 (Dense)                  │ (None, 50)             │           150 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ layer8 (Dropout)                 │ (None, 50)             │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ layer9 (Dense)                   │ (None, 500)            │        25,500 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ layer10 (Dropout)                │ (None, 500)            │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ layer11 (Dense)                  │ (None, 2000)           │     1,002,000 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_5 (Dense)                  │ (None, 5000)           │    10,005,000 │
└──────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 22,068,302 (84.18 MB)
 Trainable params: 22,064,302 (84.17 MB)
 Non-trainable params: 4,000 (15.62 KB)
```

In [ ]:

```python
# TASK Set hyperparameters for model fitting
# Begin by training for 30 epochs, with mini-batch of 256 and validation set
# having 20 per-cent of examples

hp_epochs = 30
hp_batch_size = 256     # mini-batch size
hp_val_split = 0.2      # 20% of training data for validation

Autoencoder_bl = autoencoder_model_bl.fit(x = train_tensor,
                      y = train_tensor,
                      epochs = hp_epochs,
                      batch_size = hp_batch_size,
                      shuffle = True,
                      validation_split = hp_val_split,
                      verbose=0
                      )

Autoencoder_sm = autoencoder_model_sm.fit(x = train_tensor,
                      y = train_tensor,
                      epochs = hp_epochs,
                      batch_size = hp_batch_size,
                      shuffle = True,
                      validation_split = hp_val_split,
                      verbose=0
                      )

Autoencoder_bg = autoencoder_model_bg.fit(x = train_tensor,
                      y = train_tensor,
                      epochs = hp_epochs,
                      batch_size = hp_batch_size,
                      shuffle = True,
                      validation_split = hp_val_split,
                      verbose=0
                      )
```

Now, let us look at the training process:

In [9]:
```
histories = [Autoencoder_bl, Autoencoder_sm, Autoencoder_bg]
titles = ['Baseline', 'Smaller', 'Bigger']

fig, axs = plt.subplots(1, 3, figsize=(18, 5))

for i, history in enumerate(histories):
    axs[i].plot(history.history['loss'], label='train')
    axs[i].plot(history.history['val_loss'], label='val')
    axs[i].set_title(titles[i])
    axs[i].set_xlabel('Epoch')
    axs[i].set_ylabel('Loss')
    axs[i].set_yscale('log')
    axs[i].legend(loc='upper right')

plt.tight_layout()
plt.show()
```



Now, that the model is trained, we can save the weights and use them to build an encoder. Note that weights are saved for the entire autoencoder, so we need to use `skip_mismatch = True` along with `by_name = True` to initialize weights in our encoder.

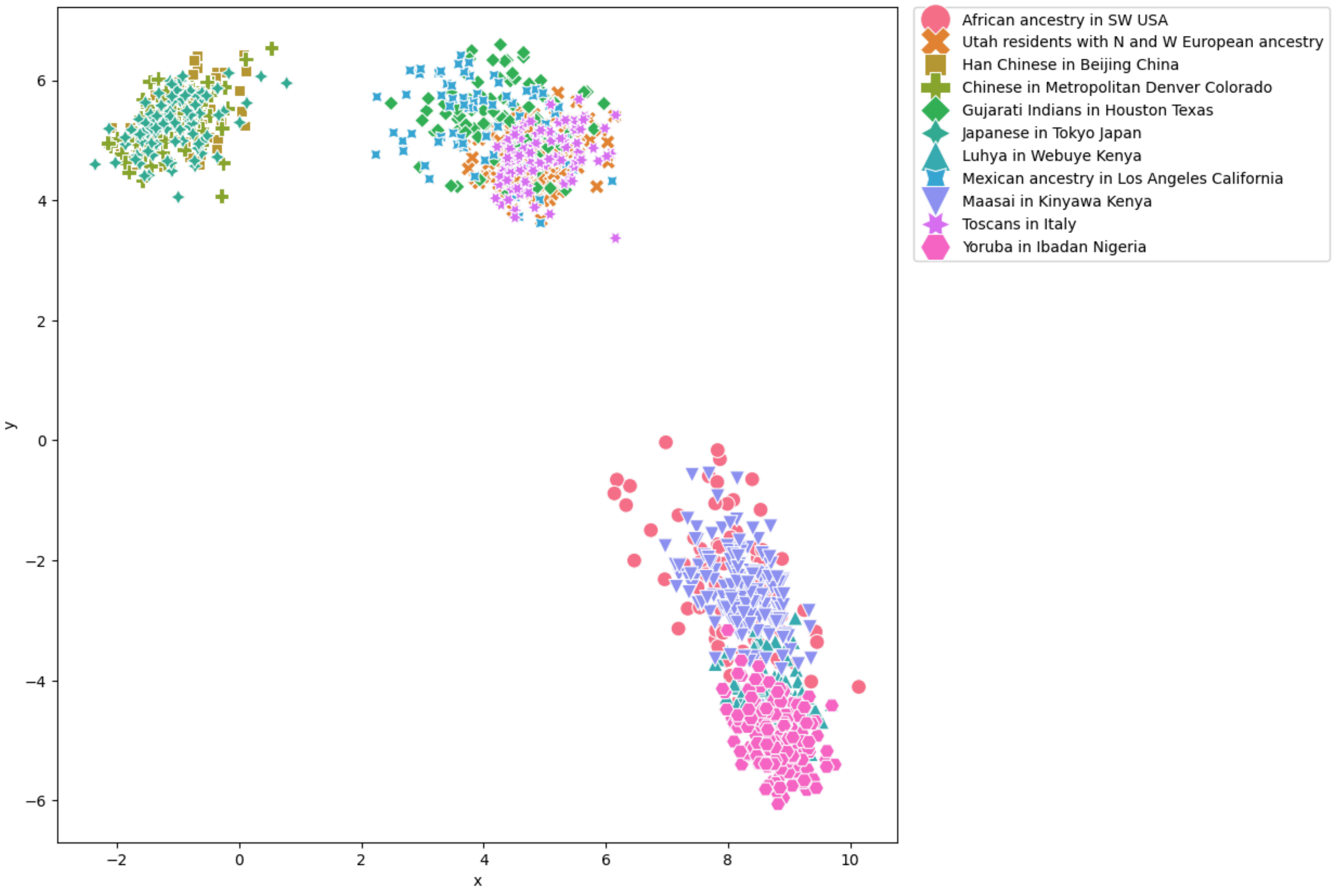
In [10]:
```
autoencoder_model_bl.save_weights('autoencoder_bl_weights.weights.h5',
                               overwrite = True)

encoder_model_bl = keras.Model(inputs = input_data, outputs = Encoder_bl(input_data))
encoder_model_bl.load_weights('autoencoder_bl_weights.weights.h5',
                           skip_mismatch = True)
encoder_model_bl.compile(
  loss = hp_loss_fn,
  optimizer = hp_optimizer,
  metrics = hp_metrics,
)

autoencoder_model_sm.save_weights('autoencoder_sm_weights.weights.h5',
                               overwrite = True)

encoder_model_sm = keras.Model(inputs = input_data, outputs = Encoder_sm(input_data))
encoder_model_sm.load_weights('autoencoder_sm_weights.weights.h5',
                           skip_mismatch = True)
encoder_model_sm.compile(
  loss = hp_loss_fn,
  optimizer = hp_optimizer,
  metrics = hp_metrics,
)

autoencoder_model_bg.save_weights('autoencoder_bg_weights.weights.h5',
                               overwrite = True)

encoder_model_bg = keras.Model(inputs = input_data, outputs = Encoder_bg(input_data))
encoder_model_bg.load_weights('autoencoder_bg_weights.weights.h5',
                           skip_mismatch = True)
encoder_model_bg.compile(
  loss = hp_loss_fn,
  optimizer = hp_optimizer,
  metrics = hp_metrics,
)
```

Let us embed our genotyping data using the encoder we have just constructed. We can also visualise the embedding.

In [11]:
```
embeded_points_bl = encoder_model_bl.predict(geno_data.to_numpy())
print(embeded_points_bl)

x = embeded_points_bl[:,0]
y = embeded_points_bl[:,1]
pop = pheno2['population']
data = {'x':x, 'y':y, 'pop':pop}
plt.figure(figsize = (10,10))
sns.scatterplot(x='x', y='y', data=data, hue='pop', style='pop', s=100)
plt.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0, markerscale=2)
plt.show()

embeded_points_sm = encoder_model_sm.predict(geno_data.to_numpy())
print(embeded_points_sm)

x = embeded_points_sm[:,0]
y = embeded_points_sm[:,1]
pop = pheno2['population']
data = {'x':x, 'y':y, 'pop':pop}
plt.figure(figsize = (10,10))
sns.scatterplot(x='x', y='y', data=data, hue='pop', style='pop', s=100)
plt.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0, markerscale=2)
plt.show()

embeded_points_bg = encoder_model_bg.predict(geno_data.to_numpy())
print(embeded_points_bg)

x = embeded_points_bg[:,0]
y = embeded_points_bg[:,1]
pop = pheno2['population']
data = {'x':x, 'y':y, 'pop':pop}
plt.figure(figsize = (10,10))
sns.scatterplot(x='x', y='y', data=data, hue='pop', style='pop', s=100)
plt.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0, markerscale=2)
plt.show()
```

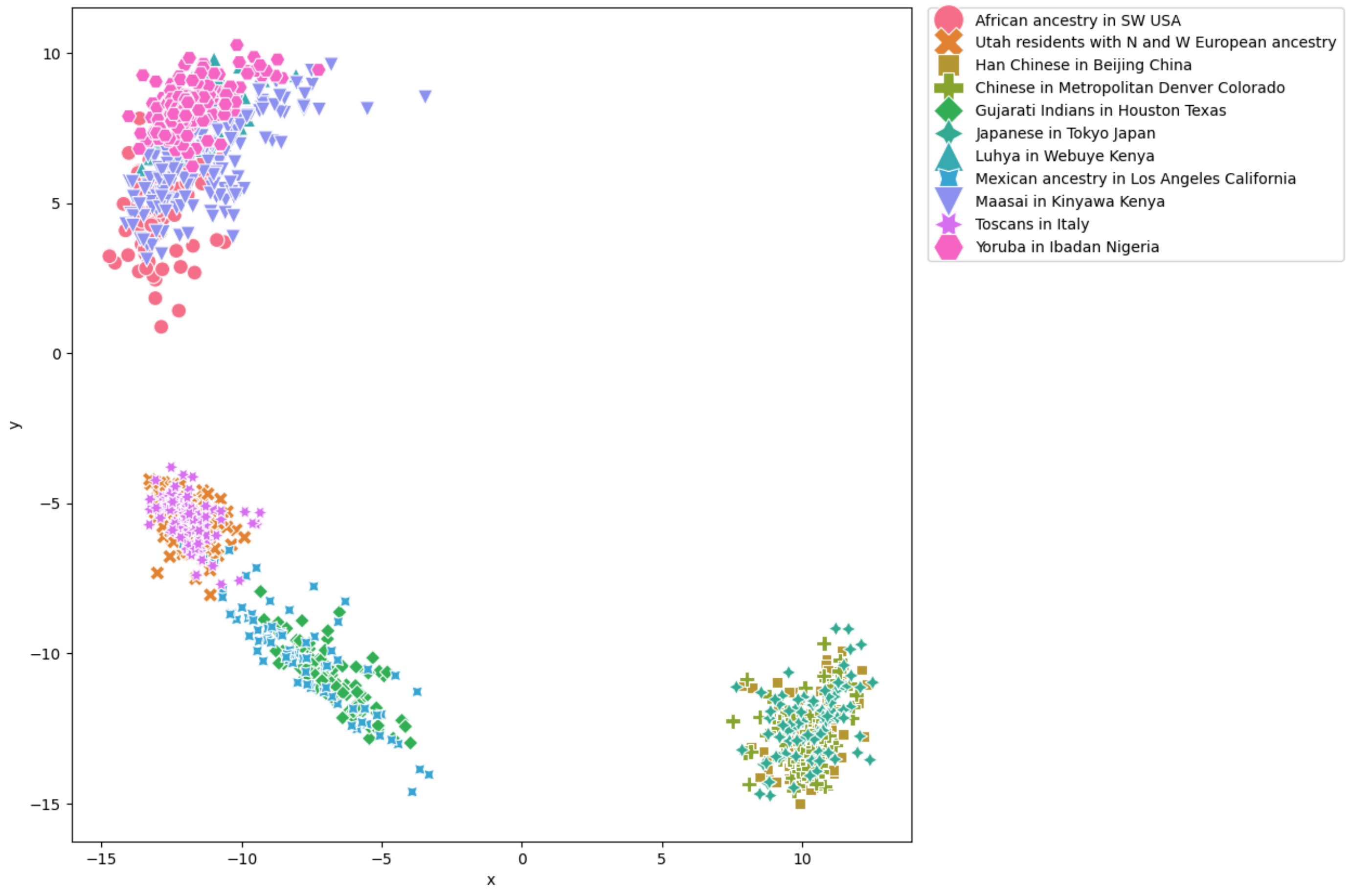
```
2025-11-27 11:26:41.409475: I external/local_xla/xla/stream_executor/cuda/subprocess_compilatio
n.cc:346] ptxas warning : Registers are spilled to local memory in function 'gemm_fusion_dot_4
0', 8 bytes spill stores, 8 bytes spill loads
```

```
37/37 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step
[[ 8.099391  -2.7050934]
 [ 8.437094  -3.3191807]
 [ 8.485687  -1.9565052]
 ...
 [ 8.338465  -4.4818306]
 [ 8.988413  -4.9713206]
 [ 9.060453  -4.947927 ]]
```



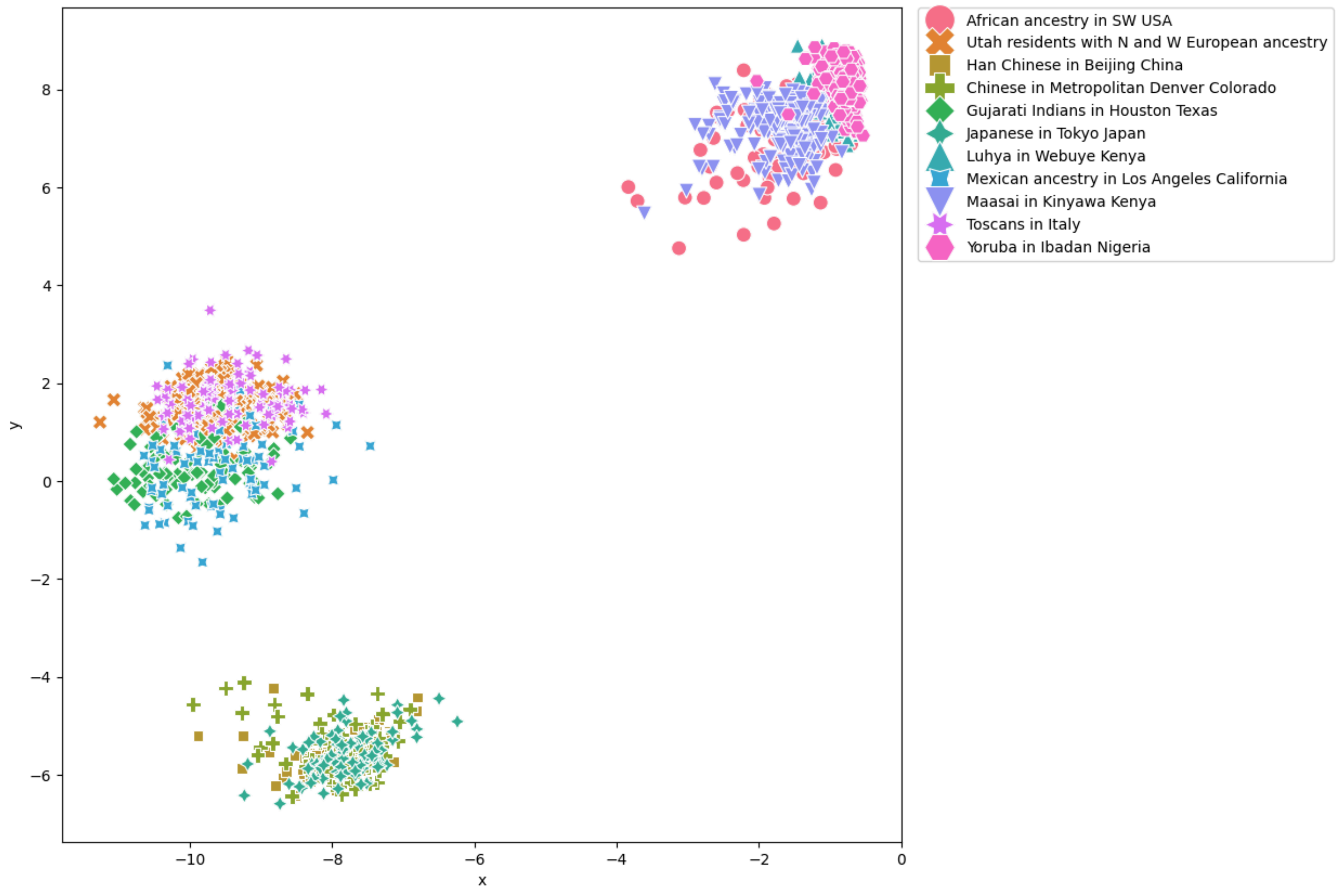
```
2025-11-27 11:26:43.566575: I external/local_xla/xla/stream_executor/cuda/subprocess_compilatio
n.cc:346] ptxas warning : Registers are spilled to local memory in function 'gemm_fusion_dot_3
8', 4 bytes spill stores, 4 bytes spill loads
```

```
37/37 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step
[[-12.144369    7.4149075]
 [-11.999359    6.5924506]
 [-13.135323    5.2313647]
 ...
 [-12.727489    7.248802 ]
 [-11.991169    7.9087834]
 [-10.5780325   8.285355 ]]
```



```
2025-11-27 11:26:46.553873: I external/local_xla/xla/stream_executor/cuda/subprocess_compilatio
n.cc:346] ptxas warning : Registers are spilled to local memory in function 'gemm_fusion_dot_4
0', 160 bytes spill stores, 160 bytes spill loads

2025-11-27 11:26:46.700165: I external/local_xla/xla/stream_executor/cuda/subprocess_compilatio
n.cc:346] ptxas warning : Registers are spilled to local memory in function 'gemm_fusion_dot_4
0', 356 bytes spill stores, 356 bytes spill loads

2025-11-27 11:26:47.275663: I external/local_xla/xla/stream_executor/cuda/subprocess_compilatio
n.cc:346] ptxas warning : Registers are spilled to local memory in function 'gemm_fusion_dot_4
0', 700 bytes spill stores, 700 bytes spill loads
```

```
37/37 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step
[[-1.7340165   7.460173  ]
 [-0.8109024   7.1927357 ]
 [-1.9854507   7.8065553 ]
 ...
 [-1.0569018   8.219454  ]
 [-0.84613174  7.8077846 ]
 [-0.58026487  8.071715  ]]
```



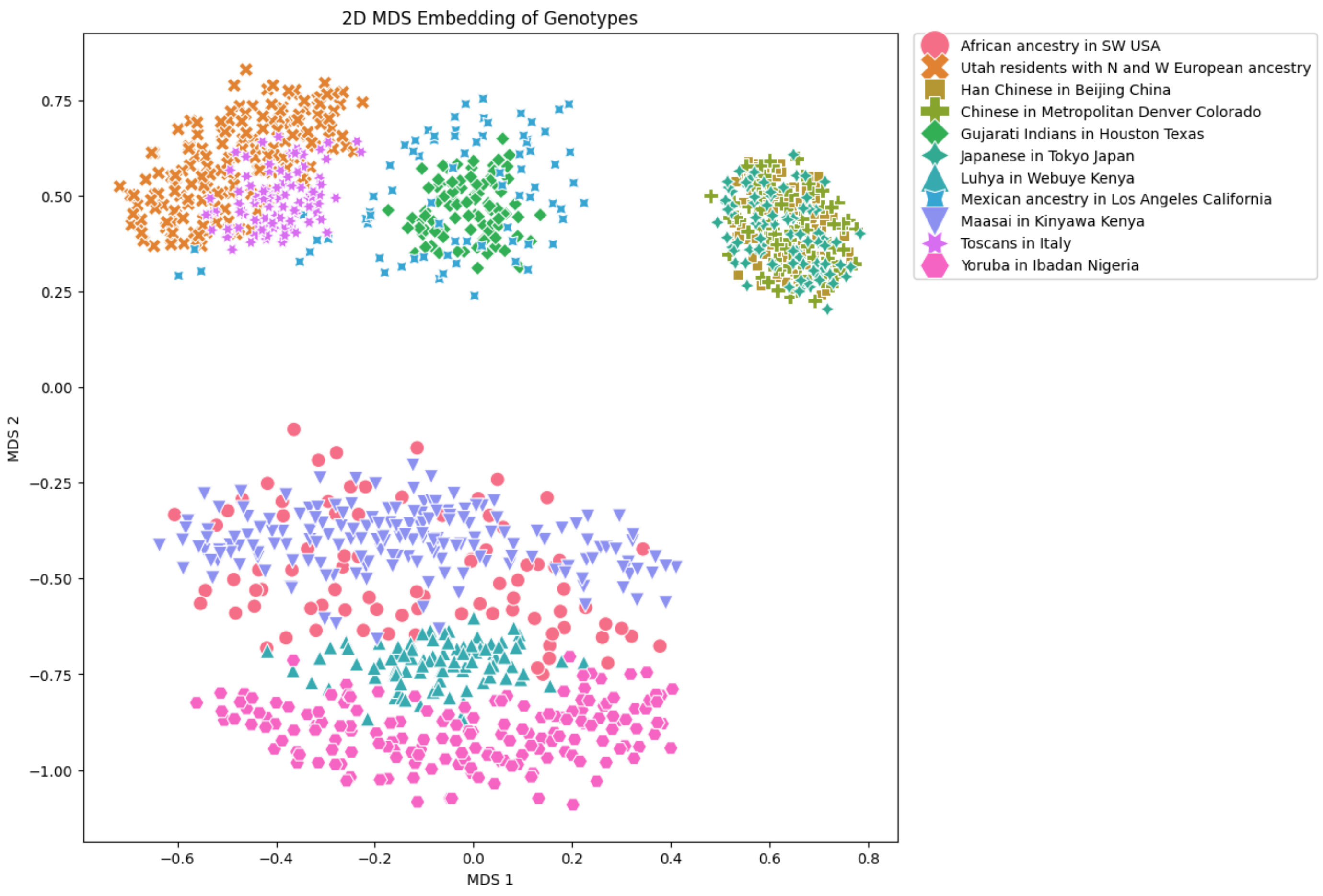
Now, we will compare the result with:

- MDS on the kinship matrix
- PCA perfored directly on raw genotypes

```
In [12]: kinship = pd.read_csv("autosomal_5k_kinship.csv", header=0, index_col=0)
```

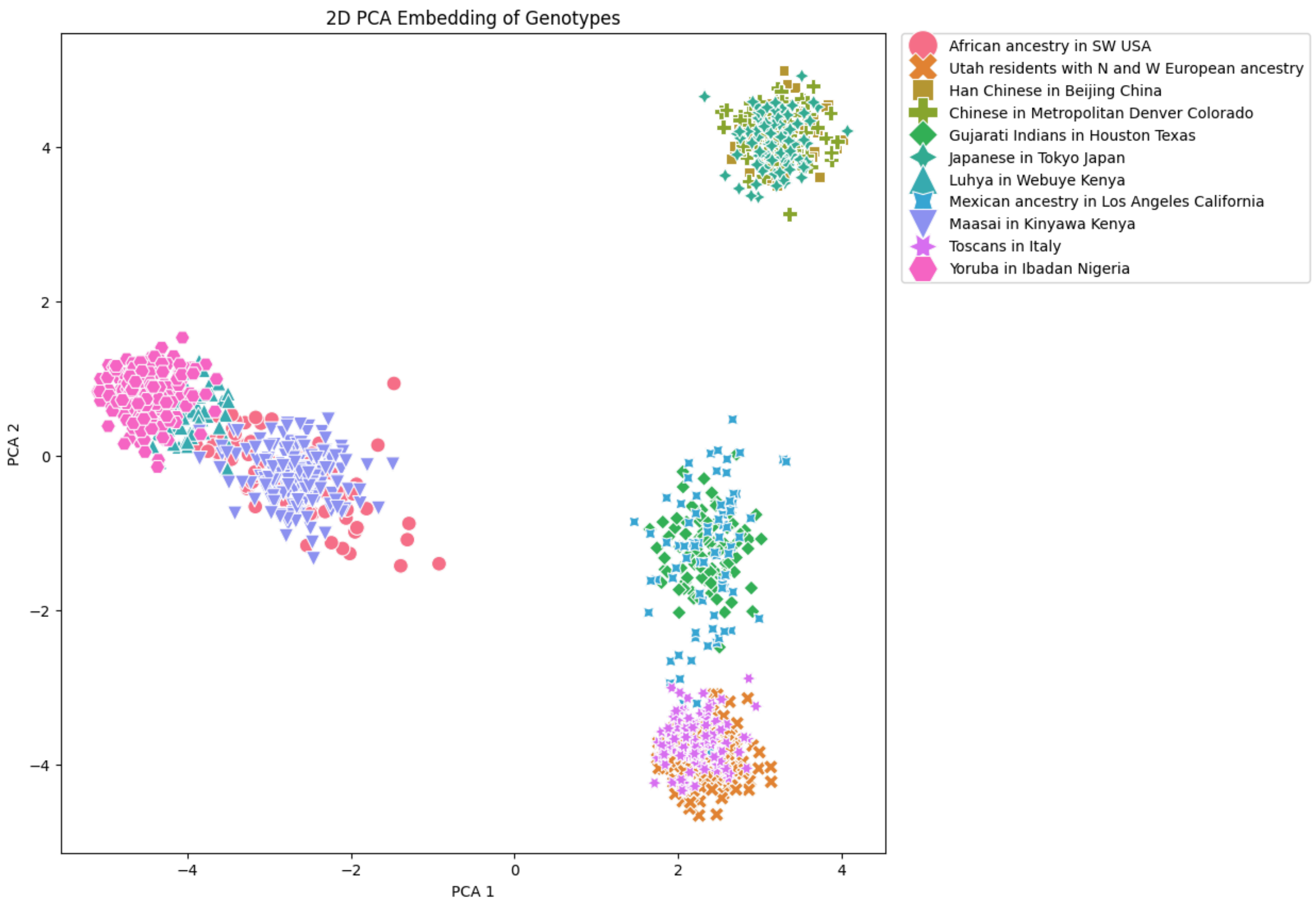
```
In [ ]: from sklearn.manifold import MDS
        embedding = MDS(n_components=2)
        mds_embedding = embedding.fit_transform(kinship)
```

```
In [14]: # TASK Plot MDS embedding in a way similar to plotting autoencoder embeddings
         x = mds_embedding[:,0]
         y = mds_embedding[:,1]
         data = {'x':x, 'y':y, 'pop':pop}

         plot_df = pd.DataFrame({'x': x, 'y': y, 'pop': pop})

         plt.figure(figsize = (10,10))
         sns.scatterplot(x='x', y='y', data=plot_df, hue='pop', style='pop', s=100)
         plt.xlabel('MDS 1')
         plt.ylabel('MDS 2')
         plt.title('2D MDS Embedding of Genotypes')
         plt.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0, markerscale=2)
         plt.show()
```



Finally, we will perform PCA on raw genotypes:

In [15]:
```python
# TASK perform PCA with 2 components on raw genotypes (use geno_data as input but remember it
# Visualise the result.
from sklearn.decomposition import PCA
embedding = PCA(n_components=2)
pca_embedding = embedding.fit_transform(geno_data.to_numpy())

x = pca_embedding[:,0]
y = pca_embedding[:,1]
pop = pheno2['population']

plot_df = pd.DataFrame({'x': x, 'y': y, 'pop': pop})

plt.figure(figsize=(10,10))
sns.scatterplot(x='x', y='y', data=plot_df, hue='pop', style='pop', s=100)
plt.xlabel('PCA 1')
plt.ylabel('PCA 2')
plt.title('2D PCA Embedding of Genotypes')
plt.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0, markerscale=2)
plt.show()
```



## Final comments

- Task 1: Each locus in the dataset is a SNP (one column per locus in the genotype matrix). The first loci are: rs416967, rs17013842, rs13052452, rs11049986, rs10994341, rs1504289, rs882529, rs3885937, rs537330, rs9372090. These rsIDs are unique identifiers but do not show chromosome information. So, we cannot tell the chromosome of a SNP just from its name.
- Task 2: Code implementation.
- Task 3: The autoencoder gives the clearest separation of populations because it can capture non-linear relationships between markers. MDS works fairly well but has more overlap between genetically close groups. PCA, as a linear method, is less able to separate populations with high correlation or gene flow. In general, genetically similar populations overlap more, while the autoencoder is less sensitive to outliers and forms more compact clusters.
- Task 4: All three autoencoder architectures (Baseline, Smaller, Bigger) capture the relative structure of the populations. Clusters are compact and mostly well-separated, with Bigger having the tightest clusters. The absolute positions in 2D differ between models because of random initialization and arbitrary rotations or translations, which is normal. The main difference between architectures is the compactness of clusters and how well closely related populations are resolved, not the overall layout.