Assignment 2

Question 2: Triplet networks & one-shot learning (10pt)

In practice 4b.4, we train a Siamese network for one-shot learning task on the Omniglot dataset. In this assignment, we will work on the same data set with the same task but extend it to triplet networks, we will also compare our model performance under different triplet selection method. The assignment contains the following 4 tasks

Import packages and mount data

Before everything, we need to import packages and mount data, *HINT: you could use the dataset in practice 4b.4 directly*

In [1]:
```
import tensorflow as tf
from tensorflow.keras import backend as K
from tensorflow.keras.layers import Input, Conv2D, Lambda, Dense, Flatten, MaxPooling2D, Dropout,Concatenate, BatchNor
malization
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.regularizers import l2
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import binary_crossentropy
import numpy as np
import os
import pickle
import matplotlib.pyplot as plt
from sklearn.utils import shuffle

from typing import Tuple, Dict, List
```

In [2]:
```
PATH = os.path.join("data", "omniglot")

with open(os.path.join(PATH, "omniglot_train.p"), "rb") as f:
    (X_train, c_train) = pickle.load(f)

with open(os.path.join(PATH, "omniglot_test.p"), "rb") as f:
    (X_test, c_test) = pickle.load(f)

print("X_train shape:", X_train.shape)
print("X_test shape:", X_test.shape)
print("")
print("training alphabets")
print([key for key in c_train.keys()])
print("test alphabets:")
print([key for key in c_test.keys()])
```

```
X_train shape: (964, 20, 105, 105)
X_test shape: (659, 20, 105, 105)

training alphabets
['Japanese_(katakana)', 'N_Ko', 'Japanese_(hiragana)', 'Bengali', 'Tagalog', 'Futurama', 'Braille', 'Arcadian', 'Earl
y_Aramaic', 'Korean', 'Grantha', 'Inuktitut_(Canadian_Aboriginal_Syllabics)', 'Tifinagh', 'Greek', 'Blackfoot_(Canadi
an_Aboriginal_Syllabics)', 'Gujarati', 'Ojibwe_(Canadian_Aboriginal_Syllabics)', 'Syriac_(Estrangelo)', 'Hebrew', 'An
glo-Saxon_Futhorc', 'Asomtavruli_(Georgian)', 'Mkhedruli_(Georgian)', 'Burmese_(Myanmar)', 'Armenian', 'Latin', 'Cyri
llic', 'Sanskrit', 'Alphabet_of_the_Magi', 'Malay_(Jawi_-_Arabic)', 'Balinese']
test alphabets:
['Tibetan', 'Aurek-Besh', 'Keble', 'Oriya', 'Kannada', 'ULOG', 'Syriac_(Serto)', 'Malayalam', 'Atemayar_Qelisayer',
'Manipuri', 'Old_Church_Slavonic_(Cyrillic)', 'Gurmukhi', 'Sylheti', 'Angelic', 'Tengwar', 'Glagolitic', 'Avesta', 'A
tlantean', 'Ge_ez', 'Mongolian']
```

Task 2.1: Build the triplet network (3pt)

We will define a triplet Network for use with the Omniglot dataset. Each branch of the triplet is a "convnet" model that transforms data to an embeddings space.

HINT: you may need "Concatenate" from keras.layer to merge the output layer

In [10]:
```python
# define a convnet model to transforms data to an embeddings space.
# === COMPLETE CODE BELOW ===
from tensorflow.keras.layers import Layer
class L2Normalization(Layer):
    def __init__(self):
        super(L2Normalization, self).__init__()

    def call(self, inputs):
        return inputs / tf.linalg.norm(inputs, axis=1, keepdims=True)

arch_convnet = "schroff" # available choices: ["hoffer", "schroff", "practical", "custom"]

if arch_convnet == "hoffer":
    # Hoffer: filter size {5,3,3,2}, and feature map dimensions {3,64,128,256,128}
    convnet = Sequential([
        Conv2D(3, (5, 5), strides=3, activation="relu", padding="same", input_shape=(105, 105, 1)),
        MaxPooling2D(),
        Conv2D(64, (3, 3), activation="relu", padding="same"),
        MaxPooling2D(),
        Conv2D(128, (3, 3), activation="relu"),
        MaxPooling2D(),
        Conv2D(256, (2, 2), activation="relu"),
        MaxPooling2D(),
        Flatten(),
        Dense(4096, activation="sigmoid")
    ])
elif arch_convnet == "schroff":
    convnet = Sequential([
        Conv2D(64, (1, 1), strides=1, activation="relu", input_shape=(105, 105, 1)),
        Conv2D(64, (3, 3), strides=1, activation="relu"),
        BatchNormalization(),
        MaxPooling2D(),

        Conv2D(192, (1, 1), strides=1, activation="relu"),
        Conv2D(192, (3, 3), strides=1, activation="relu"),
        MaxPooling2D(),

        Conv2D(384, (1, 1), strides=1, activation="relu"),
        Conv2D(384, (3, 3), strides=1, activation="relu"),

        Conv2D(256, (1, 1), strides=1, activation="relu"),
        Conv2D(256, (3, 3), strides=1, activation="relu"),

        Conv2D(256, (1, 1), strides=1, activation="relu"),
        Conv2D(256, (3, 3), strides=1, activation="relu"),

        MaxPooling2D(),
        Flatten(),
        Dense(128),
        L2Normalization()
    ])
elif arch_convnet == "custom":
    convnet = Sequential([
        Conv2D(64, (10, 10), activation="relu", padding="same", input_shape=(105, 105, 1)),
        MaxPooling2D(),

        Conv2D(128, (7, 7), activation="relu", padding="same"),
        MaxPooling2D(),

        Conv2D(256, (4, 4), activation="relu", padding="same"),
        MaxPooling2D(),

        Conv2D(256, (4, 4), activation="relu", padding="same"),
        MaxPooling2D(),

        Dropout(0.2),

        Flatten(),
        Dense(4098, activation="sigmoid"),
        Lambda(lambda x: x / tf.linalg.norm(x, axis=1, keepdims=True))
    ])
else: # also arch_convnet == "practical"
    # Practical
    convnet = Sequential([
        Conv2D(64, (10,10), activation='relu', input_shape=(105, 105, 1), kernel_regularizer=l2(2e-4)),
        MaxPooling2D(),
        BatchNormalization(),
        Dropout(0.25),
        Conv2D(128, (7,7), activation='relu', kernel_regularizer=l2(2e-4)),
        MaxPooling2D(),
        BatchNormalization(),
        Dropout(0.25),
        Conv2D(128, (4,4), activation='relu', kernel_regularizer=l2(2e-4)),
        MaxPooling2D(),
        BatchNormalization(),
        Dropout(0.25),
        Conv2D(256, (4,4), activation='relu', kernel_regularizer=l2(2e-4)),
        Flatten(),
        BatchNormalization(),
        Dropout(0.25),
        Dense(4096, activation="sigmoid", kernel_regularizer=l2(1e-3))
    ])
convnet.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_10 (Conv2D)           (None, 105, 105, 64)      128       
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 103, 103, 64)      36928     
_________________________________________________________________
batch_normalization_1 (Batch (None, 103, 103, 64)      256       
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 51, 51, 64)        0         
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 51, 51, 192)       12480     
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 49, 49, 192)       331968    
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 24, 24, 192)       0         
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 24, 24, 384)       74112     
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 22, 22, 384)       1327488   
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 22, 22, 256)       98560     
_________________________________________________________________
conv2d_17 (Conv2D)           (None, 20, 20, 256)       590080    
_________________________________________________________________
conv2d_18 (Conv2D)           (None, 20, 20, 256)       65792     
_________________________________________________________________
conv2d_19 (Conv2D)           (None, 18, 18, 256)       590080    
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 9, 9, 256)         0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 20736)             0         
_________________________________________________________________
dense_1 (Dense)              (None, 128)               2654336   
_________________________________________________________________
l2_normalization_1 (L2Normal (None, 128)               0         
=================================================================
Total params: 5,782,208
Trainable params: 5,782,080
Non-trainable params: 128
_________________________________________________________________
```

In [11]:
```
# define a Triplet network

# The anchor, positive, negative image are merged together, as the input of the triplet network, then got split to get
each one's neural codes.
generated = Input(shape=(3, 105, 105, 1), name='input')

anchor  = Lambda(lambda x: x[:,0])(generated)
pos     = Lambda(lambda x: x[:,1])(generated)
neg     = Lambda(lambda x: x[:,2])(generated)


anchor_embedding    = convnet(anchor)
pos_embedding       = convnet(pos)
neg_embedding       = convnet(neg)

# merge the anchor, positive, negative embedding together,
# let the merged layer be the output of triplet network



# === COMPLETE CODE BELOW ===
merged_output = Concatenate()([anchor_embedding, pos_embedding, neg_embedding])

triplet_net = Model(inputs=generated, outputs=merged_output)
triplet_net.summary()
```

```
Model: "model_1"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input (InputLayer)              [(None, 3, 105, 105, 0

__________________________________________________________________________________________________
lambda_3 (Lambda)               (None, 105, 105, 1)  0           input[0][0]
__________________________________________________________________________________________________
lambda_4 (Lambda)               (None, 105, 105, 1)  0           input[0][0]
__________________________________________________________________________________________________
lambda_5 (Lambda)               (None, 105, 105, 1)  0           input[0][0]
__________________________________________________________________________________________________
sequential_1 (Sequential)       (None, 128)          5782208     lambda_3[0][0]
                                                                 lambda_4[0][0]
                                                                 lambda_5[0][0]
__________________________________________________________________________________________________
concatenate_1 (Concatenate)     (None, 384)          0           sequential_1[1][0]
                                                                 sequential_1[2][0]
                                                                 sequential_1[3][0]
==================================================================================================
Total params: 5,782,208
Trainable params: 5,782,080
Non-trainable params: 128
__________________________________________________________________________________________________
```

Task 2.2: Define triplet loss (2pt)

You can find the formula of the triplet loss function in our lecture note. When training our model, make sure the network achieves a smaller loss than the margin and the network does not collapse all representations to zero vectors.

HINT: If you experience problems to achieve this goal, it might be helpful to tinker the learning rate, you can also play with the margin value to get better performance

In [12]:
```
# Notice that the ground truth variable is not used for loss calculation.
# It is used as a function argument to by-pass some Keras functionality.
# This is because the network structure already implies the ground truth for the anchor image with the "positive" imag
e.
def triplet_loss(ground_truth, network_output):

    anchor, positive, negative = tf.split(network_output, num_or_size_splits=3, axis=1)


    # === COMPLETE CODE BELOW ===
    margin = 0.2 # as specified in Schroff
    # shape (64, 128)
    anchor_negative = tf.reduce_sum((anchor - negative) ** 2, axis=1)
    anchor_positive = tf.reduce_sum((anchor - positive) ** 2, axis=1)
#     anchor_negative = tf.constant([1.]) - tf.linalg.diag_part(tf.tensordot(anchor, negative, axes=[1, 1]))
#     anchor_positive = tf.constant([1.]) - tf.linalg.diag_part(tf.tensordot(anchor, positive, axes=[1, 1]))
    loss = tf.maximum(
        anchor_positive - anchor_negative + margin,
        0
    )

    return tf.reduce_mean(loss)

triplet_net.compile(
    optimizer="adam",
    loss=triplet_loss
)
```

Task 2.3: Select triplets for training (3pt)

Different selection method

We have two different options for the triplet selection method, and we will compare the model performance under these two methods after building our model.

(1) Random triplets selection, including the following steps:

- Pick one random class for anchor
- Pick two different random picture for this class, as the anchor and positive images
- Pick another class for Negative, different from anchor_class
- Pick one random picture from the negative class.

(2) Hard triplets selection. For easy implement, for a picked anchor, positive pair, we will choose the hardest negative to form a hard triplet, that means, after picking an anchor, positive image, we will choose the negative image which is nearest from anchor image from a negative class, ie: "- d(a,n)" can get the maximum value. The whole process including the following steps:

- Pick one random class for anchor
- Pick two different random picture for this class, as an anchor and positive images
- Pick another class for negative, different from anchor_class
- Pick one hardest picture from the negative class.

HINT: when picking the hardest negative, you may need the model.predict to get the embedding of images, the calculate the distances

In [6]:
```python
# Notice that the returned  1 * np.zeros(batch_size) is to by-pass some Keras functionality, corresponding to ground_t
ruth in tripletloss
# We use a variable hard_selection to control which method we are going to use. If we set hard_selection == False, we
 will select triplets random,If we set the variable hard_selection == True, we will select hard triplets.

# === COMPLETE CODE BELOW ===
def get_batch(
    X: np.ndarray,
    batch_size: int = 64,
    hard_selection: bool = False,
    convnet: Model = None,
) -> Tuple[np.ndarray, np.ndarray]:

    while True:

        n_classes, n_examples, w, h = X.shape
        # initialize result
        triplets = np.zeros((batch_size, 3, w, h, 1))
        for i in range(batch_size):
            #Pick one random class for anchor
            anchor_class = np.random.randint(0, n_classes)

            #Pick two different random pics for this class => idx_A and idx_P
            [idx_A,idx_P] = np.random.choice(n_examples,size=2,replace=False)

            #Pick another class for negative, different from anchor_class
            # === COMPLETE CODE BELOW ===
            negative_class = anchor_class
            while negative_class == anchor_class:
                negative_class = np.random.randint(0, n_classes)

            if not hard_selection:
                #Pick a random pic from this negative class => N

                # === COMPLETE CODE BELOW ===
                idx_N = np.random.randint(0, n_examples)

            else:
                #Pick a hardest pic from this negative class => N
                # === COMPLETE CODE BELOW ===
                negative_embeddings = convnet(X_train[negative_class], training=False)
                anchor_embedding = convnet(X_train[anchor_class][[idx_A]], training=False)

                anchor_negative_distance = tf.reduce_sum((anchor_embedding - negative_embeddings) ** 2, axis=1)

                idx_N = tf.argmin(anchor_negative_distance)

            triplets[i][0] = X[anchor_class][idx_A].reshape(w, h, 1)
            triplets[i][1] = X[anchor_class][idx_P].reshape(w, h, 1)
            triplets[i][2] = X[negative_class][idx_N].reshape(w, h, 1)

        yield triplets, 1 * np.zeros(batch_size)
```

Task 2.4: One-shot learning with different selection method (2pt)

Function "make_oneshot_task" that can randomly setup such a one-shot task from a given test set (if a language is specified, using only classes/characters from that language), i.e. it will generate N pairs of images, where the first image is always the test image, and the second image is one of the N reference images. The pair of images from the same class will have target 1, all other targets are 0.

The function "test_oneshot" will generate a number (k) of such one-shot tasks and evaluate the performance of a given model on these tasks; it reports the percentage of correctly classified test images

In "test_oneshot", you can use embeddings extracted from the triplet network with L2-distance to evaluate one-shot learning. i.e. for a given one-shot task, obtain embeddings for the test image as well as the support set. Then pick the image from the support set that is closest (in L2-distance) to the test image as your one-shot prediction.

HINT you can re-use some code from practice 4b.4

In [7]:
```python
def make_oneshot_task(N, X, c, language=None):
    """Create pairs of (test image, support set image) with ground truth, for testing N-way one-shot learning."""
    n_classes, n_examples, w, h = X.shape
    indices = np.random.randint(0, n_examples, size=(N,))
    if language is not None:
        low, high = c[language]
        if N > high - low:
            raise ValueError("This language ({}) has less than {} letters".format(language, N))
        categories = np.random.choice(range(low,high), size=(N,), replace=False)
    else:  # if no language specified just pick a bunch of random letters
        categories = np.random.choice(range(n_classes), size=(N,), replace=False)
    true_category = categories[0]
    ex1, ex2 = np.random.choice(n_examples, replace=False, size=(2,))
    test_image = np.asarray([X[true_category, ex1, :, :]]*N).reshape(N, w, h, 1)
    support_set = X[categories, indices, :, :]
    support_set[0, :, :] = X[true_category, ex2]
    support_set = support_set.reshape(N, w, h, 1)
    targets = np.zeros((N,))
    targets[0] = 1
    targets, test_image, support_set = shuffle(targets, test_image, support_set)
    pairs = [test_image, support_set]
    return pairs, targets
```

In [8]:
```python
def test_oneshot(
    model: Model,
    X: np.ndarray,
    N: int,
    k: int,
    c: Dict[str, List[int]],
    verbose: bool = True
):
    # === COMPLETE CODE BELOW ===
    if verbose:
        print(f"Evaluating model on {k} random {N}-way one-shot learning tasks...")

    n_correct = 0
    for i in range(k):
        inputs, targets = make_oneshot_task(N, X, c)
        test_embedding = model.predict(inputs[0][[0]]) # All first images in inputs are the same, so we can calculate
 just one
        support_embeddings = model.predict(inputs[1])

        # Calculate sqaured Euclidean distance of normalised embeddings
        squared_l2_distances = np.sum((test_embedding - support_embeddings) ** 2, axis=1)

        # Calculate cosine distance as 1 - cosine similarity
        # squared_l2_distances = 1 - np.tensordot(support_embeddings, test_embedding, axes=[1, 1]).flatten()

        if np.argmin(squared_l2_distances) == np.argmax(targets):
            n_correct += 1

    percent_correct = 100 * n_correct / k

    if verbose:
        print(f"Average accuracy of {percent_correct}% for {N}-way one-shot learning.")

    return percent_correct
```

With different triplets selecting method (random and hard), we will train our model and evaluate the model by one-shot learning accuracy.

- You need to explicitly state the accuracy under different triplets selecting method
- When evaluating model with test_oneshot function, you should evaluate on 20 way one-shot task, and set the number (k) of evaluation one-shot tasks to be 250, then calculate the average accuracy

HINT: After training our model with random selection method, before train model under hard triplets selection, we should re-build our model (re-run the cell in Task 2.1) to initialize our model and prevent re-use the trained model of random selection

Evaluate one-shot learning with random triplets selection

In [9]:
```
# hard_selection == False, selcet triplets randomly
# Train our model and evaluate the model by one-shot learning accuracy.
loops = 10
best_acc = 0
for i in range(loops):
    print("=== Training loop {} ===".format(i+1))
    # === ADD CODE HERE ===
    triplet_net.fit(
        get_batch(X_train, batch_size=64, hard_selection=False), steps_per_epoch=100, epochs=1
    )
    avg_accuracy = test_oneshot(convnet, X_test, 20, 250, c_test)
    best_acc = avg_accuracy if avg_accuracy > best_acc else best_acc
print(f"Best accuracy: {best_acc}%")
```

```
=== Training loop 1 ===
100/100 [==============================] - 45s 446ms/step - loss: 0.0754
Evaluating model on 250 random 20-way one-shot learning tasks...
Average accuracy of 42.0% for 20-way one-shot learning.
=== Training loop 2 ===
100/100 [==============================] - 45s 446ms/step - loss: 0.0642
Evaluating model on 250 random 20-way one-shot learning tasks...
Average accuracy of 40.0% for 20-way one-shot learning.
=== Training loop 3 ===
100/100 [==============================] - 45s 445ms/step - loss: 0.0502
Evaluating model on 250 random 20-way one-shot learning tasks...
Average accuracy of 54.4% for 20-way one-shot learning.
=== Training loop 4 ===
100/100 [==============================] - 45s 445ms/step - loss: 0.0390
Evaluating model on 250 random 20-way one-shot learning tasks...
Average accuracy of 51.2% for 20-way one-shot learning.
=== Training loop 5 ===
100/100 [==============================] - 45s 446ms/step - loss: 0.0269
Evaluating model on 250 random 20-way one-shot learning tasks...
Average accuracy of 48.8% for 20-way one-shot learning.
=== Training loop 6 ===
100/100 [==============================] - 45s 446ms/step - loss: 0.0214
Evaluating model on 250 random 20-way one-shot learning tasks...
Average accuracy of 62.0% for 20-way one-shot learning.
=== Training loop 7 ===
100/100 [==============================] - 45s 446ms/step - loss: 0.0193
Evaluating model on 250 random 20-way one-shot learning tasks...
Average accuracy of 62.4% for 20-way one-shot learning.
=== Training loop 8 ===
100/100 [==============================] - 45s 446ms/step - loss: 0.0202
Evaluating model on 250 random 20-way one-shot learning tasks...
Average accuracy of 64.4% for 20-way one-shot learning.
=== Training loop 9 ===
100/100 [==============================] - 45s 445ms/step - loss: 0.0175
Evaluating model on 250 random 20-way one-shot learning tasks...
Average accuracy of 60.8% for 20-way one-shot learning.
=== Training loop 10 ===
100/100 [==============================] - 45s 446ms/step - loss: 0.0161
Evaluating model on 250 random 20-way one-shot learning tasks...
Average accuracy of 63.2% for 20-way one-shot learning.
Best accuracy: 64.4%
```

Evaluate one-shot learning with hard triplets selection

In [13]:
```python
# hard_selection == True, selcet hard triplets
# Train our model and evaluate the model by one-shot learning accuracy.
loops = 10
best_acc = 0
for i in range(loops):
    print("=== Training loop {} ===".format(i+1))
    # === ADD CODE HERE ===
    triplet_net.fit(
        get_batch(X_train, batch_size=64, hard_selection=True, convnet=convnet), steps_per_epoch=100, epochs=1
    )
    avg_accuracy = test_oneshot(convnet, X_test, 20, 250, c_test)
    best_acc = avg_accuracy if avg_accuracy > best_acc else best_acc

print(f"Best accuracy: {best_acc}%")
```

```
=== Training loop 1 ===
100/100 [==============================] - 253s 3s/step - loss: 0.1501
Evaluating model on 250 random 20-way one-shot learning tasks...
Average accuracy of 44.4% for 20-way one-shot learning.
=== Training loop 2 ===
100/100 [==============================] - 254s 3s/step - loss: 0.0810
Evaluating model on 250 random 20-way one-shot learning tasks...
Average accuracy of 60.4% for 20-way one-shot learning.
=== Training loop 3 ===
100/100 [==============================] - 254s 3s/step - loss: 0.0567
Evaluating model on 250 random 20-way one-shot learning tasks...
Average accuracy of 65.6% for 20-way one-shot learning.
=== Training loop 4 ===
100/100 [==============================] - 254s 3s/step - loss: 0.0473
Evaluating model on 250 random 20-way one-shot learning tasks...
Average accuracy of 66.8% for 20-way one-shot learning.
=== Training loop 5 ===
100/100 [==============================] - 255s 3s/step - loss: 0.0384
Evaluating model on 250 random 20-way one-shot learning tasks...
Average accuracy of 72.0% for 20-way one-shot learning.
=== Training loop 6 ===
100/100 [==============================] - 262s 3s/step - loss: 0.0340
Evaluating model on 250 random 20-way one-shot learning tasks...
Average accuracy of 72.4% for 20-way one-shot learning.
=== Training loop 7 ===
100/100 [==============================] - 264s 3s/step - loss: 0.0309
Evaluating model on 250 random 20-way one-shot learning tasks...
Average accuracy of 74.0% for 20-way one-shot learning.
=== Training loop 8 ===
100/100 [==============================] - 264s 3s/step - loss: 0.0258
Evaluating model on 250 random 20-way one-shot learning tasks...
Average accuracy of 79.6% for 20-way one-shot learning.
=== Training loop 9 ===
100/100 [==============================] - 258s 3s/step - loss: 0.0223
Evaluating model on 250 random 20-way one-shot learning tasks...
Average accuracy of 72.4% for 20-way one-shot learning.
=== Training loop 10 ===
100/100 [==============================] - 264s 3s/step - loss: 0.0210
Evaluating model on 250 random 20-way one-shot learning tasks...
Average accuracy of 77.6% for 20-way one-shot learning.
Best accuracy: 79.6%
```