In [1]:
```python
# 加载数据

from PIL import Image
import os
import pandas as pd

def default_loader(path):
    return Image.open(path)

class Dataset():
    def __init__(self, loader=default_loader, transform=None):
        imgs = [] # 用来存放照片路径和打分元组的列表

        folder_path = "../Pytorch_Book_ZhouRUC/dataset/faces/images" # 获取存放照片的文件夹路径
        file_names = os.listdir(folder_path) # 获取每个照片的名字
        # 拼接构成每个图片的路径
        image_paths = [os.path.join(folder_path, file_name) for file_name in file_names]

        # 读取打分文件
        ratings = pd.read_csv("../Pytorch_Book_ZhouRUC/dataset/faces/FaceScore.csv")
        # 提取分数列
        ratings = ratings["Rating"]

        for i in range(len(ratings)):
            # 将照片路径和打分组成一个个元组存储于imgs中
            imgs.append((image_paths[i], ratings[i]))

        self.imgs = imgs
        self.loader = loader
        self.transform = transform

    def __len__(self):  # 作为一个类方法，可以用len(full_data)调用
        return len(self.imgs)

    def __getitem__(self, index):  # 可以用full_data[index]来调用
        image_path, rating = self.imgs[index]
        image = self.loader(image_path)
        image = self.transform(image)
        return image, rating


from torchvision import datasets, transforms
from torch.utils.data import random_split

transform = transforms.Compose([
    transforms.Resize((128, 128)), # 变形为网络所需的输入形状（ 128 * 128）
    transforms.ToTensor(), # 转换为tensor（注意，此处的tensor默认在CPU上储存）
])

full_data = Dataset(transform=transform)  # 加载Dataset实例

# 划分训练集，验证集
train_size = int(len(full_data) * 0.7) # 训练集和验证集比例7：3
val_size = len(full_data) - train_size

train_set, val_set = random_split(full_data, [train_size, val_size])  # 按比例随机划分出训练集、验证集
```

In [3]:
```python
# 构建Dataloader

import torch

batch_size = 64
train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, shuffle=True)
val_loader = torch.utils.data.DataLoader(val_set, batch_size=batch_size, shuffle=True)
```

In [5]:
```python
# 数据展示

from matplotlib import pyplot as plt
from torchvision.utils import make_grid
images, labels = next(iter(train_loader))  # 将train_loader构造成iterable，读取第一个batch中的数据
print(images.shape)
print(labels.shape)
plt.figure(figsize=(12, 20))  # 设置画布大小
plt.axis('off')  # 隐藏坐标轴
plt.imshow(make_grid(images, nrow=8).permute((1, 2, 0)))
# make_grid函数把多张图片一起显示，permute函数调换channel维的顺序
plt.show()
```

```
torch.Size([64, 3, 128, 128])
torch.Size([64])
```

In [4]:
```
# 构建神经网络

import torch.nn as nn
from torchsummary import summary

class Network(nn.Module):  # 建立一个三层神经网络的类
    def __init__(self):
        super().__init__()  # 继承父类的设定
        self.layer1 = nn.Linear(128*128*3, 512)  # 第一层线性层，降维至512
        self.relu = nn.ReLU()  # 将向量通过ReLU，引入非线性
        self.layer2 = nn.Linear(512, 1)  # 第二层线性层，得到一维的打分预测数据

    def forward(self, x):
        x = x.reshape(-1, 128*128*3)  # 先将数据压缩，与线性层维数匹配
        x = self.layer1(x) # 第一层线性变换
        x = self.relu(x) # relu一下
        x = self.layer2(x) # 第二层线性变换
        return x

IMSIZE = 128
network_model = Network().cuda()
summary(network_model, (3, IMSIZE, IMSIZE))  # 模型参数统计
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                  [-1, 512]      25,166,336
              ReLU-2                  [-1, 512]               0
            Linear-3                    [-1, 1]             513
================================================================
Total params: 25,166,849
Trainable params: 25,166,849
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.19
Forward/backward pass size (MB): 0.01
Params size (MB): 96.00
Estimated Total Size (MB): 96.20
----------------------------------------------------------------
```

In [16]:

```
# 模型训练

device = torch.device('cuda')

import time

# 验证模型
def validate(model, val_loader):
    model.eval()  # 示意进行模型预测
    val_loss = 0
    # 将模型用于验证集预测一下，检测在验证集上的表现
    for inputs, labels in val_loader:
        inputs, labels = inputs.to(device), labels.to(device)  # 将输入张量和标签放到GPU上
        outputs = model(inputs)  # 得到预测张量
        loss = torch.nn.MSELoss()(outputs.view(-1), labels.to(torch.float32))  # 计算预测值与真值之间的均方误差
        val_loss += loss

    val_loss /= len(val_loader)
    return val_loss

# 将每个epoch训练结果输出
def printlog(epoch, train_time, train_loss, val_loss, epochs=10):
    print(f"Epoch [{epoch}/{epochs}], time: {train_time:.2f}s, train_loss: {train_loss:.4f}, val_loss: {val_loss:.4f}")

# 训练模型
def train(model, optimizer, train_loader, val_loader, epochs=1):
    train_losses = []
    val_losses = []  # 分别用于记录训练和验证过程的损失，最后返回
    model.train()  # 示意进行模型训练
    for i in range(epochs):  # 执行epoch次
        train_loss = 0
        val_loss = 0  # 设定初值
        start = time.time()  # 记录时间
        for inputs, labels in train_loader:  # 获取输入张量和标签
            inputs, labels = inputs.to(device), labels.to(device)  # 挪至GPU
            optimizer.zero_grad()  # 清空optimizer的梯度
            outputs = model(inputs)  # 得到输出张量
            loss = torch.nn.MSELoss()(outputs.view(-1), labels.to(torch.float32))  # 计算损失
            train_loss += loss.item()  # item获取loss的数值，以免梯度值被加入
            loss.backward()  # 反向传播
            optimizer.step()  # 模型优化，梯度下降

        end = time.time()  # 记录结束时间
        train_time = end - start  # 计算单个epoch用时
        train_loss /= len(train_loader)  # 平均一个batch的损失
        val_loss = validate(model, val_loader)  # 得到验证集上平均一个batch的损失
        train_losses.append(train_loss)  # 将上两个损失记录进列表中
        val_losses.append(val_loss)
        printlog(i+1, train_time, train_loss, val_loss)  # 打印训练结果
    return train_losses, val_losses

lr = 1e-3  # 学习率
epochs = 10  # epoch数
optimizer = torch.optim.Adam(network_model.parameters(), lr=lr)  # 选取Adam优化器
history = train(network_model, optimizer, train_loader, val_loader, epochs)  # 整个训练的执行
```

```
Epoch [1/10], time: 10.64s, train_loss: 0.9670, val_loss: 0.3545
Epoch [2/10], time: 10.07s, train_loss: 0.3143, val_loss: 0.3167
Epoch [3/10], time: 10.84s, train_loss: 0.2992, val_loss: 0.3155
Epoch [4/10], time: 10.36s, train_loss: 0.3020, val_loss: 0.3282
Epoch [5/10], time: 10.44s, train_loss: 0.2964, val_loss: 0.3024
Epoch [6/10], time: 10.52s, train_loss: 0.3564, val_loss: 0.3146
Epoch [7/10], time: 10.47s, train_loss: 0.3146, val_loss: 0.3484
Epoch [8/10], time: 9.76s, train_loss: 0.3034, val_loss: 0.3193
Epoch [9/10], time: 9.95s, train_loss: 0.2944, val_loss: 0.3116
Epoch [10/10], time: 9.92s, train_loss: 0.3104, val_loss: 0.3101
```