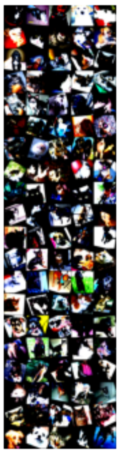
In [1]:
```python
from utils import *
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchsummary import summary
from matplotlib import pyplot as plt
import os
from torchvision.utils import make_grid
from torchvision import transforms
import torchvision
import time
```

In [2]:
```python
IMSIZE = 128  # 图片尺寸
batch_size = 100  # 小批量大小
data_dir = "../dataset/CatDog" # 数据集位置

# with DA
train_transform1 = transforms.Compose([
    transforms.Resize((IMSIZE, IMSIZE)), # 先将图片调整尺寸
    transforms.RandomCrop(IMSIZE, padding = 8), # 加上padding后, 对图像进行随机裁剪
    # 对图像进行一系列变换, 随机旋转、沿x y轴平移, 缩放, 错切
    transforms.RandomAffine(degrees = 30, translate = (0.1, 0.1), scale = (0.8, 1.2), shear = 10),
    transforms.RandomHorizontalFlip(p = 0.5), # 以1/2的概率水平翻转每张图片
    transforms.ToTensor(), # 将图片向量转化为张量
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]) # 对像素进行归一化
])

# without DA
train_transform2 = transforms.Compose([
    transforms.Resize((IMSIZE, IMSIZE)), # 调整图片尺寸
    transforms.ToTensor(), # 转换为张量
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]) # 归一化
])

# 验证集的数据变化
test_transform = transforms.Compose([
    transforms.Resize((IMSIZE, IMSIZE)), # 调整图片尺寸
    transforms.ToTensor(), # 转换为张量
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]) # 归一化
])
```

In [36]:
```python
# 加载数据
train_set_1 = torchvision.datasets.ImageFolder(root = os.path.join(data_dir, "train"), transform = train_transform1) # 带DA的训练集
train_set_2 = torchvision.datasets.ImageFolder(root = os.path.join(data_dir, "train"), transform = train_transform2) # 不带DA的训练集
test_set = torchvision.datasets.ImageFolder(root = os.path.join(data_dir, "validation"), transform = test_transform) # 加载验证集

train_loader_1 = torch.utils.data.DataLoader(train_set_1, batch_size = batch_size, shuffle = True, num_workers = 4) # 构造训练集的两
train_loader_2 = torch.utils.data.DataLoader(train_set_2, batch_size = batch_size, shuffle = True, num_workers = 4)
test_loader = torch.utils.data.DataLoader(test_set, batch_size = batch_size, shuffle = False, num_workers = 4) # 构造验证集的loader
```

In [45]:
```python
images, labels = next(iter(train_loader_1))  # 获取训练集第一个批次中的图片及相应标签
print(images.shape) # 打印一个图片batch的大小
print(labels.shape) # 打印一个标签batch的大小
plt.figure(figsize=(12, 6))  # 设置画布大小
plt.axis('off')  # 隐藏坐标轴
plt.imshow(make_grid(images, nrow=5).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()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
torch.Size([100, 3, 128, 128])
torch.Size([100])
```



In [34]:
```python
# with BN
class CNN_1(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Sequential(
            # 第一层卷积，16个卷积核，每个卷积核channel数为3，大小为5，故参数量为（5*5*3 + 1）*16 = 1216
            nn.Conv2d(in_channels=3, out_channels=16, kernel_size=5, stride=1, padding=1),
            # 归一化时每一个channel有两个参数需训练，故参数量为2*16 = 32
            nn.BatchNorm2d(16),
            nn.ReLU(),
            # 最大值池化
            nn.MaxPool2d(kernel_size=2, stride=2),
            # 第二层卷积，32个卷积核，每个卷积核channel数为16，大小为5，故参数量为（5*5*16 + 1）*32 = 12832
            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5, stride=1, padding=1),
            # 归一化时每一个channel有两个参数需训练，故参数量为2*32 = 64
            nn.BatchNorm2d(32),
            nn.ReLU(),
            # 最大值池化
            nn.MaxPool2d(kernel_size=2, stride=2),
            # 第三层卷积，64个卷积核，每个卷积核channel数为32，大小为4，故参数量为（4*4*32 + 1）*64 = 32832
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride = 1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            # 最大值池化
            nn.MaxPool2d(kernel_size=2, stride=2),
            # 第四层卷积，128个卷积核，每个卷积核channel数为64，大小为2，故参数量为（2*2*64 + 1）*128 = 32896
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=2, stride = 1, padding=0),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            # 最大值池化
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        self.fc = nn.Sequential(
            # 参数量为（128*6*6 + 1）*1024 = 4719616
            nn.Linear(128*6*6, 1024),
            nn.ReLU(),
            # 以0.2的概率舍弃神经元，增加模型泛化程度
            nn.Dropout(0.2),
            # 参数量为（1024 + 1）*128 = 131200
            nn.Linear(1024, 128),
            nn.ReLU(),
            # 以0.1的概率舍弃神经元，增加模型泛化程度
            nn.Dropout(0.1),
            # 参数量为（128 + 1）* 64 = 8256
            nn.Linear(128, 64),
            nn.ReLU(),
            # 参数量为（64 + 1）*2 = 130
            nn.Linear(64, 2),
            nn.ReLU()
        )

    def forward(self, x):
        feature = self.conv(x) # 通过卷积层，得到[-1, 128, 6, 6]大小的特征张量
        output = feature.view(x.size()[0], -1) # 将特征拉长，变成[-1, 128*6*6]的张量通过全连接层
        return self.fc(output)

    # 权重初始化
    def initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d): # 如果为卷积层
                torch.nn.init.xavier_normal_(m.weight.data) # 对卷积层权重以xavier正态分布方法初始化
                if m.bias is not None: # 如果bias不为None
                    m.bias.data.zero_() # 将bias置为0
            elif isinstance(m, nn.BatchNorm2d): # 如果为批量归一化层
                m.weight.data.fill_(1) # 对权重赋1
                m.bias.data.zero_() # 对bias赋0
            elif isinstance(m, nn.Linear): # 如果为全连接层
                torch.nn.init.normal_(m.weight.data, 0, 0.01) # 将权重初始化为均值为0，标准差为0.01的正态分布数据
                # m.weight.data.normal_(0,0.01)
                m.bias.data.zero_() # 将bias赋0
        print("Initialization Finished! ")

model_1 = CNN_1() # 生成带BN的神经网络实例
model_1.initialize_weights() # 初始化model_1的权重
model_1.to(torch.device("cuda")) # 将model_1的参数移至GPU
summary(model_1, (3, IMSIZE, IMSIZE))
```

```
Initialization Finished!
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 16, 126, 126]           1,216
       BatchNorm2d-2         [-1, 16, 126, 126]              32
              ReLU-3         [-1, 16, 126, 126]               0
         MaxPool2d-4           [-1, 16, 63, 63]               0
            Conv2d-5           [-1, 32, 61, 61]          12,832
       BatchNorm2d-6           [-1, 32, 61, 61]              64
              ReLU-7           [-1, 32, 61, 61]               0
         MaxPool2d-8           [-1, 32, 30, 30]               0
            Conv2d-9           [-1, 64, 29, 29]          32,832
      BatchNorm2d-10           [-1, 64, 29, 29]             128
             ReLU-11           [-1, 64, 29, 29]               0
        MaxPool2d-12           [-1, 64, 14, 14]               0
           Conv2d-13          [-1, 128, 13, 13]          32,896
      BatchNorm2d-14          [-1, 128, 13, 13]             256
             ReLU-15          [-1, 128, 13, 13]               0
        MaxPool2d-16            [-1, 128, 6, 6]               0
           Linear-17                 [-1, 1024]       4,719,616
             ReLU-18                 [-1, 1024]               0
          Dropout-19                 [-1, 1024]               0
          Linear-20                  [-1, 128]         131,200
            ReLU-21                  [-1, 128]               0
         Dropout-22                  [-1, 128]               0
          Linear-23                   [-1, 64]           8,256
            ReLU-24                   [-1, 64]               0
          Linear-25                    [-1, 2]             130
            ReLU-26                    [-1, 2]               0
================================================================
Total params: 4,939,458
Trainable params: 4,939,458
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.19
Forward/backward pass size (MB): 11.13
Params size (MB): 18.84
Estimated Total Size (MB): 30.16
----------------------------------------------------------------
```

In [44]:

```
# without BN
class CNN_2(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Sequential(
            # 第一层卷积，16个卷积核，每个卷积核channel数为3，大小为5，故参数量为（5*5*3 + 1）*16 = 1216
            nn.Conv2d(in_channels=3, out_channels=16, kernel_size=5, stride=1, padding=1),
            # 归一化时每一个channel有两个参数需训练，故参数量为2*16 = 32
            nn.ReLU(),
            # 最大值池化
            nn.MaxPool2d(kernel_size=2, stride=2),
            # 第二层卷积，32个卷积核，每个卷积核channel数为16，大小为5，故参数量为（5*5*16 + 1）*32 = 12832
            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5, stride=1, padding=1),
            # 归一化时每一个channel有两个参数需训练，故参数量为2*32 = 64
            nn.ReLU(),
            # 最大值池化
            nn.MaxPool2d(kernel_size=2, stride=2),
            # 第三层卷积，64个卷积核，每个卷积核channel数为32，大小为4，故参数量为（4*4*32 + 1）*64 = 32832
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride = 1, padding=1),
            nn.ReLU(),
            # 最大值池化
            nn.MaxPool2d(kernel_size=2, stride=2),
            # 第四层卷积，128个卷积核，每个卷积核channel数为64，大小为2，故参数量为（2*2*64 + 1）*128 = 32896
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=2, stride = 1, padding=0),
            nn.ReLU(),
            # 最大值池化
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        self.fc = nn.Sequential(
            # 参数量为（128*6*6 + 1）*1024 = 4719616
            nn.Linear(128*6*6, 1024),
            nn.ReLU(),
            # 以0.2的概率舍弃神经元，增加模型泛化程度
            nn.Dropout(0.2),
            # 参数量为（1024 + 1）*128 = 131200
            nn.Linear(1024, 128),
            nn.ReLU(),
            # 以0.1的概率舍弃神经元，增加模型泛化程度
            nn.Dropout(0.1),
            # 参数量为（128 + 1）* 64 = 8256
            nn.Linear(128, 64),
            nn.ReLU(),
            # 参数量为（64 + 1）*2 = 130
            nn.Linear(64, 2),
            nn.ReLU()
        )

    def forward(self, x):
        feature = self.conv(x) # 通过卷积层，得到[-1, 128, 6, 6]大小的特征张量
        output = feature.view(x.size()[0], -1) # 将特征拉长，变成[-1, 128*6*6]的张量通过全连接层
        return self.fc(output)

    # 权重初始化
    def initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d): # 如果为卷积层
                torch.nn.init.xavier_normal_(m.weight.data) # 对卷积层权重以xavier正态分布方法初始化
                if m.bias is not None: # 如果bias不为None
                    m.bias.data.zero_() # 将bias置为0
            elif isinstance(m, nn.BatchNorm2d): # 如果为批量归一化层
                m.weight.data.fill_(1) # 对权重赋1
                m.bias.data.zero_() # 对bias赋0
            elif isinstance(m, nn.Linear): # 如果为全连接层
                torch.nn.init.normal_(m.weight.data, 0, 0.01) # 将权重初始化为均值为0，标准差为0.01的正态分布数据
                # m.weight.data.normal_(0,0.01)
                m.bias.data.zero_() # 将bias赋0
        print("Initialization Finished! ")

model_2 = CNN_2() # 创造不带BN的CNN_2的实例
model_2.initialize_weights() # 初始化model_2的权重
model_2.to(torch.device("cuda")) # 将model_2的参数移至GPU
summary(model_2, (3, IMSIZE, IMSIZE))
```

```
Initialization Finished!
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
```

```
================================================================
            Conv2d-1         [-1, 16, 126, 126]           1,216
              ReLU-2         [-1, 16, 126, 126]               0
         MaxPool2d-3           [-1, 16, 63, 63]               0
            Conv2d-4           [-1, 32, 61, 61]          12,832
              ReLU-5           [-1, 32, 61, 61]               0
         MaxPool2d-6           [-1, 32, 30, 30]               0
            Conv2d-7           [-1, 64, 29, 29]          32,832
              ReLU-8           [-1, 64, 29, 29]               0
         MaxPool2d-9           [-1, 64, 14, 14]               0
           Conv2d-10          [-1, 128, 13, 13]          32,896
             ReLU-11          [-1, 128, 13, 13]               0
        MaxPool2d-12            [-1, 128, 6, 6]               0
           Linear-13                 [-1, 1024]       4,719,616
             ReLU-14                 [-1, 1024]               0
          Dropout-15                 [-1, 1024]               0
           Linear-16                  [-1, 128]         131,200
             ReLU-17                  [-1, 128]               0
          Dropout-18                  [-1, 128]               0
           Linear-19                   [-1, 64]           8,256
             ReLU-20                   [-1, 64]               0
           Linear-21                    [-1, 2]             130
             ReLU-22                    [-1, 2]               0
================================================================
Total params: 4,938,978
Trainable params: 4,938,978
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.19
Forward/backward pass size (MB): 7.71
Params size (MB): 18.84
Estimated Total Size (MB): 26.73
----------------------------------------------------------------
```

In [11]:

```python
# with DA & BN
lr = 1e-3 # 设置学习率
optimizer = torch.optim.Adam(model_1.parameters(), lr=lr) # 设置优化器
history_1 = train(model_1, optimizer, train_loader_1, test_loader, epochs=30) # 带DA/BN的模型的训练结果
```

```
Epoch [1/30], time: 13.86s, loss: 0.6241, acc: 0.6416, val_loss: 0.6036, val_acc: 0.6977
Epoch [2/30], time: 14.41s, loss: 0.6160, acc: 0.6591, val_loss: 0.5590, val_acc: 0.7174
Epoch [3/30], time: 14.20s, loss: 0.5451, acc: 0.7233, val_loss: 0.4775, val_acc: 0.7714
Epoch [4/30], time: 14.25s, loss: 0.5214, acc: 0.7405, val_loss: 0.4606, val_acc: 0.7782
Epoch [5/30], time: 14.52s, loss: 0.4825, acc: 0.7677, val_loss: 0.4137, val_acc: 0.8104
Epoch [6/30], time: 14.03s, loss: 0.4568, acc: 0.7809, val_loss: 0.4131, val_acc: 0.8173
Epoch [7/30], time: 14.40s, loss: 0.4376, acc: 0.7934, val_loss: 0.4114, val_acc: 0.8075
Epoch [8/30], time: 14.60s, loss: 0.4094, acc: 0.8056, val_loss: 0.3436, val_acc: 0.8497
Epoch [9/30], time: 14.29s, loss: 0.3821, acc: 0.8280, val_loss: 0.3243, val_acc: 0.8544
Epoch [10/30], time: 14.12s, loss: 0.3660, acc: 0.8341, val_loss: 0.3550, val_acc: 0.8374
Epoch [11/30], time: 14.00s, loss: 0.3536, acc: 0.8441, val_loss: 0.3119, val_acc: 0.8672
Epoch [12/30], time: 13.85s, loss: 0.3408, acc: 0.8441, val_loss: 0.2707, val_acc: 0.8850
Epoch [13/30], time: 13.78s, loss: 0.3322, acc: 0.8508, val_loss: 0.3417, val_acc: 0.8328
Epoch [14/30], time: 14.73s, loss: 0.3130, acc: 0.8611, val_loss: 0.2733, val_acc: 0.8811
Epoch [15/30], time: 13.85s, loss: 0.3037, acc: 0.8663, val_loss: 0.2504, val_acc: 0.8932
Epoch [16/30], time: 14.57s, loss: 0.2974, acc: 0.8715, val_loss: 0.2572, val_acc: 0.8954
Epoch [17/30], time: 14.12s, loss: 0.2893, acc: 0.8697, val_loss: 0.2585, val_acc: 0.8829
Epoch [18/30], time: 13.79s, loss: 0.2794, acc: 0.8750, val_loss: 0.2418, val_acc: 0.8952
Epoch [19/30], time: 13.93s, loss: 0.2807, acc: 0.8795, val_loss: 0.2406, val_acc: 0.8980
Epoch [20/30], time: 13.88s, loss: 0.2699, acc: 0.8817, val_loss: 0.2450, val_acc: 0.8990
Epoch [21/30], time: 13.87s, loss: 0.2591, acc: 0.8885, val_loss: 0.2412, val_acc: 0.9019
Epoch [22/30], time: 14.11s, loss: 0.2671, acc: 0.8831, val_loss: 0.2350, val_acc: 0.9082
Epoch [23/30], time: 13.98s, loss: 0.2580, acc: 0.8862, val_loss: 0.2207, val_acc: 0.9063
Epoch [24/30], time: 13.88s, loss: 0.2501, acc: 0.8935, val_loss: 0.2302, val_acc: 0.9053
Epoch [25/30], time: 13.93s, loss: 0.2459, acc: 0.8931, val_loss: 0.2527, val_acc: 0.8911
Epoch [26/30], time: 14.67s, loss: 0.2474, acc: 0.8925, val_loss: 0.2061, val_acc: 0.9143
Epoch [27/30], time: 13.91s, loss: 0.2363, acc: 0.8982, val_loss: 0.2226, val_acc: 0.9099
Epoch [28/30], time: 14.30s, loss: 0.2272, acc: 0.9017, val_loss: 0.2187, val_acc: 0.9087
Epoch [29/30], time: 13.84s, loss: 0.2310, acc: 0.9024, val_loss: 0.1983, val_acc: 0.9178
Epoch [30/30], time: 14.32s, loss: 0.2363, acc: 0.8988, val_loss: 0.2090, val_acc: 0.9096
```

最高精度 91.78%

In [41]:

```python
# only with DA
lr = 1e-3
optimizer = torch.optim.Adam(model_2.parameters(), lr=lr)
history_2 = train(model_2, optimizer, train_loader_1, test_loader, epochs=30) # 只有DA的模型的训练结果
```

```
Epoch [1/30], time: 14.73s, loss: 0.6688, acc: 0.5797, val_loss: 0.6261, val_acc: 0.6548
Epoch [2/30], time: 14.36s, loss: 0.6117, acc: 0.6631, val_loss: 0.5794, val_acc: 0.6928
Epoch [3/30], time: 14.44s, loss: 0.5615, acc: 0.7089, val_loss: 0.4981, val_acc: 0.7685
Epoch [4/30], time: 14.01s, loss: 0.5269, acc: 0.7384, val_loss: 0.4867, val_acc: 0.7667
Epoch [5/30], time: 14.50s, loss: 0.5062, acc: 0.7567, val_loss: 0.4560, val_acc: 0.7862
Epoch [6/30], time: 14.38s, loss: 0.4871, acc: 0.7629, val_loss: 0.4360, val_acc: 0.8061
Epoch [7/30], time: 13.52s, loss: 0.4760, acc: 0.7737, val_loss: 0.4219, val_acc: 0.8100
Epoch [8/30], time: 14.23s, loss: 0.4687, acc: 0.7797, val_loss: 0.4416, val_acc: 0.7944
Epoch [9/30], time: 14.31s, loss: 0.4520, acc: 0.7903, val_loss: 0.3993, val_acc: 0.8222
Epoch [10/30], time: 14.26s, loss: 0.4423, acc: 0.7919, val_loss: 0.3849, val_acc: 0.8298
Epoch [11/30], time: 14.37s, loss: 0.4324, acc: 0.7972, val_loss: 0.4104, val_acc: 0.8151
Epoch [12/30], time: 13.80s, loss: 0.4358, acc: 0.7947, val_loss: 0.3777, val_acc: 0.8341
Epoch [13/30], time: 14.36s, loss: 0.4103, acc: 0.8161, val_loss: 0.3656, val_acc: 0.8396
Epoch [14/30], time: 14.71s, loss: 0.4101, acc: 0.8152, val_loss: 0.3900, val_acc: 0.8220
Epoch [15/30], time: 13.92s, loss: 0.4067, acc: 0.8137, val_loss: 0.3660, val_acc: 0.8374
Epoch [16/30], time: 13.89s, loss: 0.4021, acc: 0.8145, val_loss: 0.3464, val_acc: 0.8489
Epoch [17/30], time: 13.82s, loss: 0.3855, acc: 0.8267, val_loss: 0.3465, val_acc: 0.8502
Epoch [18/30], time: 14.11s, loss: 0.3909, acc: 0.8231, val_loss: 0.3584, val_acc: 0.8407
Epoch [19/30], time: 14.05s, loss: 0.3799, acc: 0.8279, val_loss: 0.3385, val_acc: 0.8551
Epoch [20/30], time: 13.58s, loss: 0.3778, acc: 0.8289, val_loss: 0.3415, val_acc: 0.8504
Epoch [21/30], time: 13.82s, loss: 0.3822, acc: 0.8306, val_loss: 0.3375, val_acc: 0.8532
Epoch [22/30], time: 14.05s, loss: 0.3667, acc: 0.8382, val_loss: 0.3324, val_acc: 0.8573
Epoch [23/30], time: 13.47s, loss: 0.3618, acc: 0.8388, val_loss: 0.3257, val_acc: 0.8596
Epoch [24/30], time: 13.69s, loss: 0.3611, acc: 0.8362, val_loss: 0.3711, val_acc: 0.8401
Epoch [25/30], time: 14.71s, loss: 0.3582, acc: 0.8347, val_loss: 0.3293, val_acc: 0.8608
Epoch [26/30], time: 14.33s, loss: 0.3586, acc: 0.8387, val_loss: 0.3069, val_acc: 0.8689
Epoch [27/30], time: 14.36s, loss: 0.3575, acc: 0.8395, val_loss: 0.3108, val_acc: 0.8695
Epoch [28/30], time: 14.48s, loss: 0.3427, acc: 0.8450, val_loss: 0.3144, val_acc: 0.8629
Epoch [29/30], time: 13.91s, loss: 0.3435, acc: 0.8455, val_loss: 0.3006, val_acc: 0.8744
Epoch [30/30], time: 14.36s, loss: 0.3379, acc: 0.8498, val_loss: 0.3020, val_acc: 0.8713
```

In [33]:

```
# only with BN
lr = 1e-3
model_3 = CNN_1() # 创造另一个带BN的CNN_1实例
model_3.initialize_weights() # 初始化参数
model_3.to(torch.device("cuda")) # 移至GPU
optimizer = torch.optim.Adam(model_3.parameters(), lr=lr)
history_3 = train(model_3, optimizer, train_loader_2, test_loader, epochs=30) # 只带BN的模型的训练结果
```

```
Initialization Finished!
Epoch [1/30], time: 13.94s, loss: 0.5826, acc: 0.6858, val_loss: 0.5377, val_acc: 0.7406
Epoch [2/30], time: 13.37s, loss: 0.4790, acc: 0.7693, val_loss: 0.4432, val_acc: 0.7928
Epoch [3/30], time: 13.24s, loss: 0.4146, acc: 0.8063, val_loss: 0.4192, val_acc: 0.8130
Epoch [4/30], time: 13.66s, loss: 0.3595, acc: 0.8388, val_loss: 0.3874, val_acc: 0.8326
Epoch [5/30], time: 13.06s, loss: 0.3104, acc: 0.8642, val_loss: 0.3753, val_acc: 0.8344
Epoch [6/30], time: 13.30s, loss: 0.2763, acc: 0.8825, val_loss: 0.4453, val_acc: 0.8231
Epoch [7/30], time: 13.32s, loss: 0.2073, acc: 0.9127, val_loss: 0.4366, val_acc: 0.8425
Epoch [8/30], time: 13.35s, loss: 0.1515, acc: 0.9396, val_loss: 0.4651, val_acc: 0.8388
Epoch [9/30], time: 13.40s, loss: 0.1049, acc: 0.9592, val_loss: 0.4675, val_acc: 0.8387
Epoch [10/30], time: 13.11s, loss: 0.0783, acc: 0.9695, val_loss: 0.5231, val_acc: 0.8291
Epoch [11/30], time: 13.61s, loss: 0.0576, acc: 0.9780, val_loss: 0.6717, val_acc: 0.8374
Epoch [12/30], time: 13.24s, loss: 0.0421, acc: 0.9839, val_loss: 0.7117, val_acc: 0.8364
Epoch [13/30], time: 13.42s, loss: 0.0400, acc: 0.9862, val_loss: 0.7127, val_acc: 0.8373
Epoch [14/30], time: 13.37s, loss: 0.0251, acc: 0.9911, val_loss: 0.7818, val_acc: 0.8391
Epoch [15/30], time: 12.96s, loss: 0.0244, acc: 0.9911, val_loss: 0.7528, val_acc: 0.8359
Epoch [16/30], time: 13.59s, loss: 0.0191, acc: 0.9937, val_loss: 0.8434, val_acc: 0.8406
Epoch [17/30], time: 12.68s, loss: 0.0155, acc: 0.9947, val_loss: 0.9167, val_acc: 0.8369
Epoch [18/30], time: 13.57s, loss: 0.0218, acc: 0.9919, val_loss: 0.8985, val_acc: 0.8368
Epoch [19/30], time: 13.78s, loss: 0.0169, acc: 0.9943, val_loss: 0.6194, val_acc: 0.8364
Epoch [20/30], time: 13.24s, loss: 0.0167, acc: 0.9939, val_loss: 1.0817, val_acc: 0.8471
Epoch [21/30], time: 13.49s, loss: 0.0135, acc: 0.9955, val_loss: 0.8988, val_acc: 0.8336
Epoch [22/30], time: 13.03s, loss: 0.0184, acc: 0.9931, val_loss: 1.2277, val_acc: 0.8352
Epoch [23/30], time: 13.18s, loss: 0.0229, acc: 0.9921, val_loss: 0.8498, val_acc: 0.8360
Epoch [24/30], time: 13.20s, loss: 0.0264, acc: 0.9913, val_loss: 0.9976, val_acc: 0.8347
Epoch [25/30], time: 12.86s, loss: 0.0215, acc: 0.9934, val_loss: 0.8967, val_acc: 0.8414
Epoch [26/30], time: 12.63s, loss: 0.0135, acc: 0.9957, val_loss: 1.0854, val_acc: 0.8231
Epoch [27/30], time: 13.57s, loss: 0.0149, acc: 0.9949, val_loss: 0.8444, val_acc: 0.8359
Epoch [28/30], time: 13.09s, loss: 0.0103, acc: 0.9965, val_loss: 1.2024, val_acc: 0.8278
Epoch [29/30], time: 13.80s, loss: 0.0158, acc: 0.9951, val_loss: 1.1251, val_acc: 0.8379
Epoch [30/30], time: 13.46s, loss: 0.0130, acc: 0.9955, val_loss: 0.9991, val_acc: 0.8394
```

In [42]:

```
# without DA and BN
lr = 1e-3
model_4 = CNN_2() # 创造另一个不带BN的CNN_2的实例
model_4.initialize_weights() # 初始化参数
model_4.to(torch.device("cuda")) # 移至GPU
optimizer = torch.optim.Adam(model_4.parameters(), lr=lr)
history_4 = train(model_4, optimizer, train_loader_2, test_loader, epochs=30) # 不带DA和BN的模型的训练结果
```

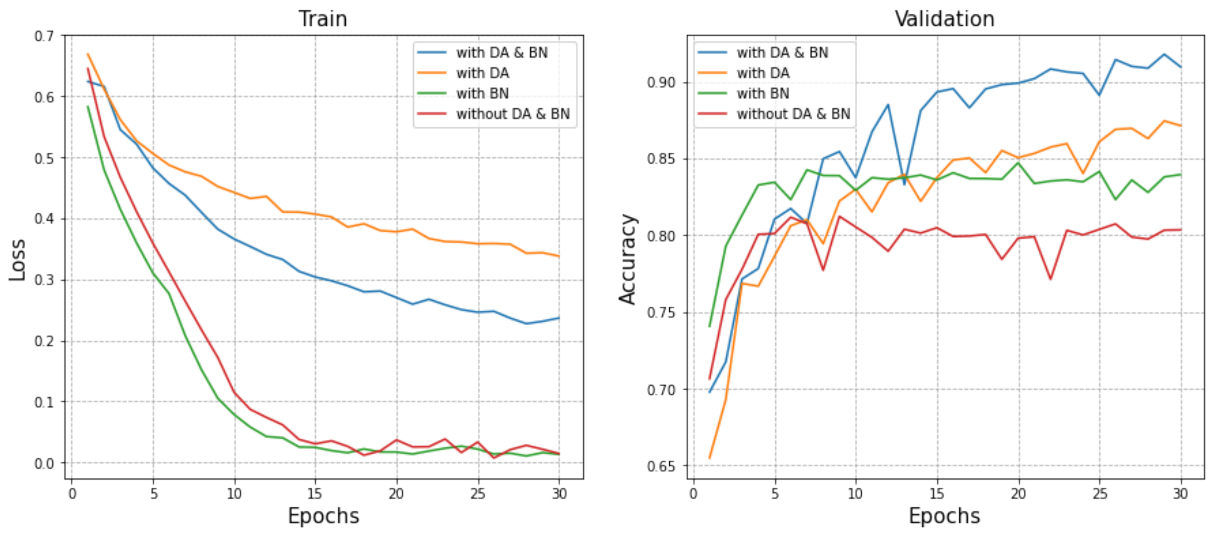
```
Initialization Finished!
Epoch [1/30], time: 13.34s, loss: 0.6451, acc: 0.6161, val_loss: 0.5790, val_acc: 0.7064
Epoch [2/30], time: 12.83s, loss: 0.5338, acc: 0.7343, val_loss: 0.5022, val_acc: 0.7580
Epoch [3/30], time: 13.00s, loss: 0.4669, acc: 0.7801, val_loss: 0.4663, val_acc: 0.7777
Epoch [4/30], time: 13.33s, loss: 0.4108, acc: 0.8116, val_loss: 0.4338, val_acc: 0.8005
Epoch [5/30], time: 13.40s, loss: 0.3592, acc: 0.8391, val_loss: 0.4310, val_acc: 0.8010
Epoch [6/30], time: 13.15s, loss: 0.3119, acc: 0.8667, val_loss: 0.4288, val_acc: 0.8116
Epoch [7/30], time: 13.34s, loss: 0.2639, acc: 0.8865, val_loss: 0.4505, val_acc: 0.8074
Epoch [8/30], time: 13.23s, loss: 0.2169, acc: 0.9108, val_loss: 0.5893, val_acc: 0.7771
Epoch [9/30], time: 13.41s, loss: 0.1718, acc: 0.9305, val_loss: 0.5347, val_acc: 0.8122
Epoch [10/30], time: 13.21s, loss: 0.1147, acc: 0.9563, val_loss: 0.6369, val_acc: 0.8053
Epoch [11/30], time: 13.40s, loss: 0.0867, acc: 0.9675, val_loss: 0.7090, val_acc: 0.7985
Epoch [12/30], time: 13.13s, loss: 0.0735, acc: 0.9733, val_loss: 0.7936, val_acc: 0.7895
Epoch [13/30], time: 13.18s, loss: 0.0612, acc: 0.9776, val_loss: 0.7812, val_acc: 0.8038
Epoch [14/30], time: 12.84s, loss: 0.0374, acc: 0.9877, val_loss: 0.8872, val_acc: 0.8013
Epoch [15/30], time: 13.69s, loss: 0.0302, acc: 0.9911, val_loss: 1.0000, val_acc: 0.8048
Epoch [16/30], time: 13.08s, loss: 0.0351, acc: 0.9882, val_loss: 1.1319, val_acc: 0.7991
Epoch [17/30], time: 13.28s, loss: 0.0260, acc: 0.9921, val_loss: 1.1006, val_acc: 0.7994
Epoch [18/30], time: 13.46s, loss: 0.0116, acc: 0.9971, val_loss: 1.2997, val_acc: 0.8004
Epoch [19/30], time: 13.24s, loss: 0.0186, acc: 0.9939, val_loss: 1.3531, val_acc: 0.7842
Epoch [20/30], time: 13.03s, loss: 0.0365, acc: 0.9870, val_loss: 1.1118, val_acc: 0.7980
Epoch [21/30], time: 13.09s, loss: 0.0252, acc: 0.9922, val_loss: 1.2715, val_acc: 0.7989
Epoch [22/30], time: 12.30s, loss: 0.0255, acc: 0.9916, val_loss: 1.4614, val_acc: 0.7712
Epoch [23/30], time: 13.26s, loss: 0.0381, acc: 0.9869, val_loss: 1.2705, val_acc: 0.8031
Epoch [24/30], time: 13.53s, loss: 0.0159, acc: 0.9955, val_loss: 1.3456, val_acc: 0.8001
Epoch [25/30], time: 12.91s, loss: 0.0331, acc: 0.9881, val_loss: 1.3340, val_acc: 0.8037
Epoch [26/30], time: 13.58s, loss: 0.0070, acc: 0.9983, val_loss: 1.4001, val_acc: 0.8073
Epoch [27/30], time: 13.59s, loss: 0.0205, acc: 0.9929, val_loss: 1.4256, val_acc: 0.7987
Epoch [28/30], time: 13.41s, loss: 0.0277, acc: 0.9897, val_loss: 1.4656, val_acc: 0.7974
Epoch [29/30], time: 13.36s, loss: 0.0213, acc: 0.9933, val_loss: 1.3822, val_acc: 0.8032
Epoch [30/30], time: 13.01s, loss: 0.0144, acc: 0.9949, val_loss: 1.4818, val_acc: 0.8034
```

In [43]:

```
# (训练集loss, 验证集精度)
fig, ax = plt.subplots(1, 2) # 将画布分为一行两列
fig.set_figwidth(15) # 设置画布宽度
fig.set_figheight(6) # 设置画布高度

ax[0].grid(linestyle = 'dashed')                          # 添加网格线
ax[0].plot(torch.arange(1, len(history_1[0])+1), history_1[0])   # 绘制使用DA和BN的实验结果
ax[0].plot(torch.arange(1, len(history_2[0])+1), history_2[0])  # 绘制只使用DA的实验结果
ax[0].plot(torch.arange(1, len(history_3[0])+1), history_3[0])  # 绘制只使用BN的实验结果
```

```
ax[0].plot(torch.arange(1, len(history_4[0])+1), history_4[0])  # 不用DA和BN的实验结果
ax[0].set_xlabel("Epochs", fontsize = 15)                      # 添加 x-label文字
ax[0].set_ylabel("Loss", fontsize = 15)                        # 添加 y-label文字
ax[0].set_title("Train", fontsize = 15)                        # 添加标题
ax[0].legend(labels = ['with DA & BN', 'with DA', 'with BN', 'without DA & BN']) # 添加图例

ax[1].grid(linestyle = 'dashed')                               # 添加网格线
ax[1].plot(torch.arange(1, len(history_1[3])+1), history_1[3])   # 绘制使用DA和BN的实验结果
ax[1].plot(torch.arange(1, len(history_2[3])+1), history_2[3])  # 绘制只使用DA的实验结果
ax[1].plot(torch.arange(1, len(history_3[3])+1), history_3[3])  # 绘制只使用BN的实验结果
ax[1].plot(torch.arange(1, len(history_4[3])+1), history_4[3])  # 不用DA和BN的实验结果
ax[1].set_xlabel("Epochs", fontsize = 15)                      # 添加 x-label文字
ax[1].set_ylabel("Accuracy", fontsize = 15)                        # 添加 y-label文字
ax[1].set_title("Validation", fontsize = 15)                        # 添加标题
ax[1].legend(labels = ['with DA & BN', 'with DA', 'with BN', 'without DA & BN']) # 添加图例
```

```
Out[43]: <matplotlib.legend.Legend at 0x7f130cd041d0>
```



观察到在验证集精度方面，同时有DA和BN的神经网络模型优于其他模型，而只有DA的模型预测精度略好于只有BN的模型，DA和BN都没有的模型预测精度最差，体现了数据增强以及批量归一化对模型精度的重要贡献。然而，与此同时，观察到了带有数据增强的模型比不带有数据增强的模型在减小训练集损失上更慢，而带有批量归一化的模型相较于不带有批量归一化的模型，加快了训练集上损失的减小。这可能是由于数据增强改变了原始样本，使模型泛化能力增强，但降低了在训练集上的收敛速度；批量归一化避免了训练过程的梯度爆炸或梯度消失，使训练更加稳定可控，增加了收敛速度。