

Lab Report

Author: 刘佳隆

Student ID: 518010910009

总体介绍

参考来源

- 参考代码1: <https://www.kaggle.com/code/nguyenmanhcuongg/pytorch-video-classification-with-conv2d-lstm>
- 参考代码2: https://github.com/doronharitan/human_activity_recognition_LRCN

代码结构

代码 lab3-cnn-lstm.ipynb 由以下几个部分组成：

1. 定义数据集处理函数

参考 lab3_data_scratch.py 中的代码，定义数据集处理函数，用于将数据集转换为模型所需的数据格式。

2. 定义模型

调用预训练CNN搭建模型

使用预训练的ResNet152模型作为卷积层，搭建CNN-LSTM模型，即 [PretrainedConvLstm.py](#) 中的模型。

使用自行定义的普通CNN搭建模型

自行搭建了一个由卷积层、池化层、全连接层和一个LSTM层组成的CNN-LSTM模型，即 [ConvLstm.py](#) 中的模型。

使用自定义的ResNet和LSTM构建模型

自行搭建了一个由自定义的ResNet和LSTM层组成的CNN-LSTM模型，即 [ResNetLstm.py](#) 中的模型。

3. 定义评估函数

定义了用于评估训练结果的函数，包括计算平均损失和准确率。定义了用于可视化训练过程中的损失和准确率的函数。

4. 定义训练函数

参考代码 1 中的训练函数，对其进行了简化和修改，定义了训练函数，并在训练完成后保存模型的权重。

5. 训练模型

指定模型参数，而后加载数据集和模型，调用训练函数进行训练。

6. 评估模型

调用评估函数评估模型的训练结果，并生成可视化图表。

辅助代码

- [LSTM.py](#): 定义了通用的LSTM模型。
- [ConvLstm.py](#)、[PretrainedConvLstm.py](#)、[ResNetLstm.py](#): 定义了三种模型。
- [classify.py](#): 调用训练结果模型对测试集进行分类。
- [export_model.py](#): 将训练结果模型导出为 ONNX 格式，用于生成可视化模型结构图。

系统设计

模型设计

三种模型的 LSTM 结构均相同，使用 torch.nn.LSTM 搭建，其主要代码和函数解释如下：

```
class Lstm(nn.Module):  
    ...  
    定义LSTM模型  
    :param latent_dim: LSTM的输入维度  
    :param hidden_size: LSTM的隐藏层维度  
    :param lstm_layers: LSTM的层数  
    :param bidirectional: LSTM是否为双向  
    ...  
  
    def __init__(self, latent_dim, hidden_size, lstm_layers, bidirectional):  
        super(Lstm, self).__init__()  
        self.Lstm = nn.LSTM(latent_dim, hidden_size, num_layers=lstm_layers, batch_first=True,  
                           bidirectional=bidirectional)  
        self.hidden_state = None  
  
    # 重置LSTM的隐藏层状态  
    def reset_hidden_state(self):  
        self.hidden_state = None  
  
    def forward(self, x):  
        output, self.hidden_state = self.Lstm(x, self.hidden_state)  
        return output
```

调用预训练CNN的模型设计

预训练的 CNN 使用 torchvision.models.resnet152 加载，固定卷积层的参数，修改最后一层全连接层，将其输出维度设置为 LSTM 的输入维度。

其主要代码和函数解释如下：

```

class PretrainedConv(nn.Module):
    """
    使用预训练的ResNet152模型作为卷积层
    :param latent_dim: 输出的特征维度
    """

    def __init__(self, latent_dim):
        super(PretrainedConv, self).__init__()
        # 使用预训练的ResNet152模型
        self.conv_model = torchvision.models.resnet152(pretrained=True)
        # ===== 固定卷积层的参数 =====
        for param in self.conv_model.parameters():
            param.requires_grad = False
        # ===== 修改最后一层全连接层 =====
        # latent_dim为输出的特征维度，也是LSTM的输入维度
        self.conv_model.fc = nn.Linear(self.conv_model.fc.in_features, latent_dim)

    def forward(self, x):
        return self.conv_model(x)

class PretrainedConvLstm(nn.Module):
    """
    使用预训练的CNN和LSTM构建模型
    :param latent_dim: LSTM的输入维度
    :param hidden_size: LSTM的隐藏层维度
    :param lstm_layers: LSTM的层数
    :param bidirectional: LSTM是否为双向
    :param n_class: 分类的类别数
    """

    def __init__(self, latent_dim, hidden_size, lstm_layers, bidirectional, n_class):
        super(PretrainedConvLstm, self).__init__()
        self.conv_model = PretrainedConv(latent_dim)
        self.Lstm = Lstm(latent_dim, hidden_size, lstm_layers, bidirectional)
        self.output_layer = nn.Sequential(
            nn.Linear(2 * hidden_size if bidirectional == True else hidden_size, n_class),
            nn.Softmax(dim=-1)
        )

    def forward(self, x):
        batch_size, time_steps, channel_x, height, width = x.shape
        conv_input = x.view(batch_size * time_steps, channel_x, height, width)
        conv_output = self.conv_model(conv_input)
        lstm_input = conv_output.view(batch_size, time_steps, -1)
        lstm_output = self.Lstm(lstm_input)
        lstm_output = lstm_output[:, -1, :]
        output = self.output_layer(lstm_output)
        return output

```

自行定义的普通CNN的模型设计

自行定义的普通 CNN 使用 4 层 torch.nn.Conv2d、torch.nn.ReLU、torch.nn.MaxPool2d 和 1 层 torch.nn.Linear 搭建，将其输出维度设置为LSTM 的输入维度。模型其余结构与调用预训练 CNN 的模型相同。

其主要代码和函数解释如下：

```

class Conv(nn.Module):
    """
    自定义的普通CNN模型
    :param latent_dim: 输出的特征维度
    """

    def __init__(self, latent_dim):
        super(Conv, self).__init__()
        self.conv_model = nn.Sequential(
            # 输入维度: (batch_size, 3, 128, 128)
            nn.Conv2d(in_channels=3, out_channels=16, kernel_size=6, stride=2, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # 输出维度: (batch_size, 16, 32, 32)

            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # 输出维度: (batch_size, 32, 16, 16)

            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # 输出维度: (batch_size, 64, 4, 4)

            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # 输出维度: (batch_size, 64, 2, 2)
        )
        self.fc = nn.Linear(64 * 2 * 2, latent_dim)

    def forward(self, x):
        batch_size, time_steps, channel_x, height, width = x.shape
        x = x.view(batch_size * time_steps, channel_x, height, width)
        x = self.conv_model(x)
        x = x.view(batch_size * time_steps, -1)
        x = self.fc(x)
        x = x.view(batch_size, time_steps, -1)
        return x

class ConvLstm(nn.Module):
    """
    使用自定义的CNN和LSTM构建模型
    :param latent_dim: LSTM的输入维度
    :param hidden_size: LSTM的隐藏层维度
    :param lstm_layers: LSTM的层数
    :param bidirectional: LSTM是否为双向
    :param n_class: 分类的类别数
    """

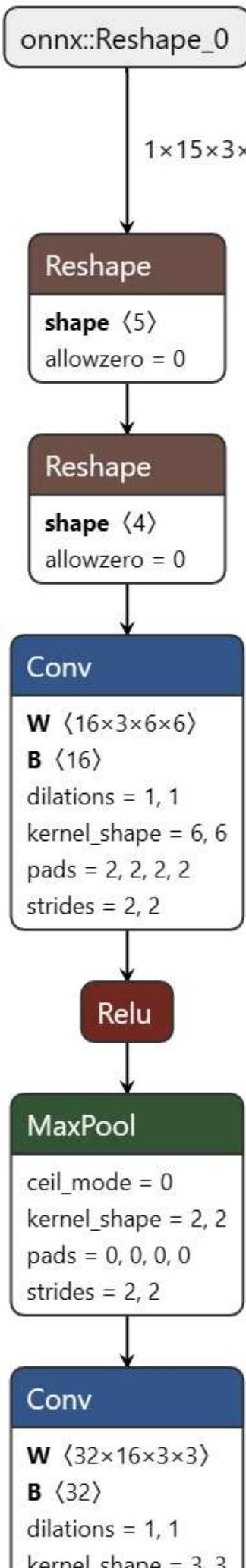
    def __init__(self, latent_dim, hidden_size, lstm_layers, bidirectional, n_class):
        super(ConvLstm, self).__init__()

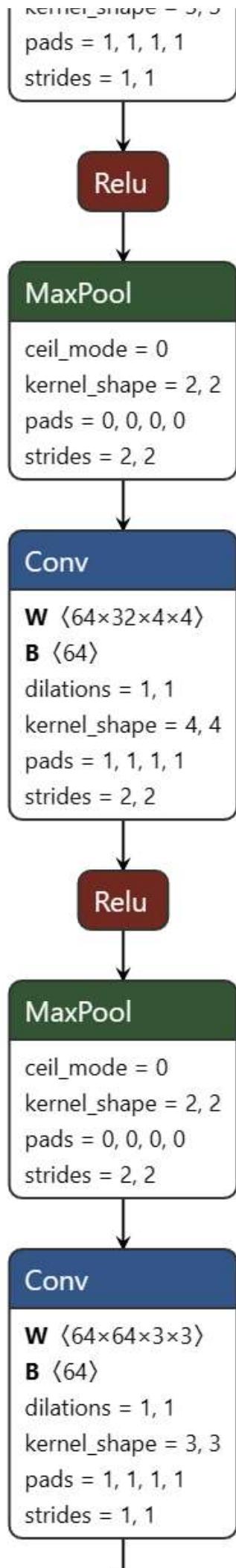
```

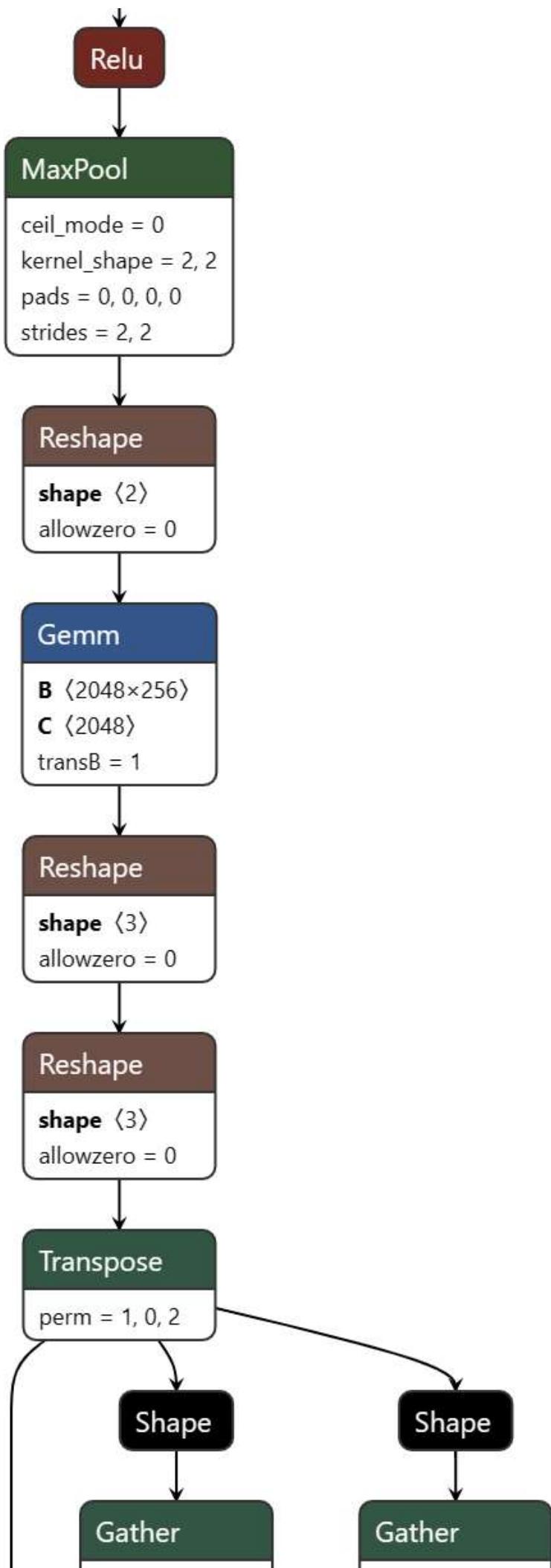
```
self.conv_model = Conv(latent_dim)
self.Lstm = Lstm(latent_dim, hidden_size, lstm_layers, bidirectional)
self.output_layer = nn.Sequential(
    nn.Linear(2 * hidden_size if bidirectional == True else hidden_size, n_class),
    nn.Softmax(dim=-1)
)

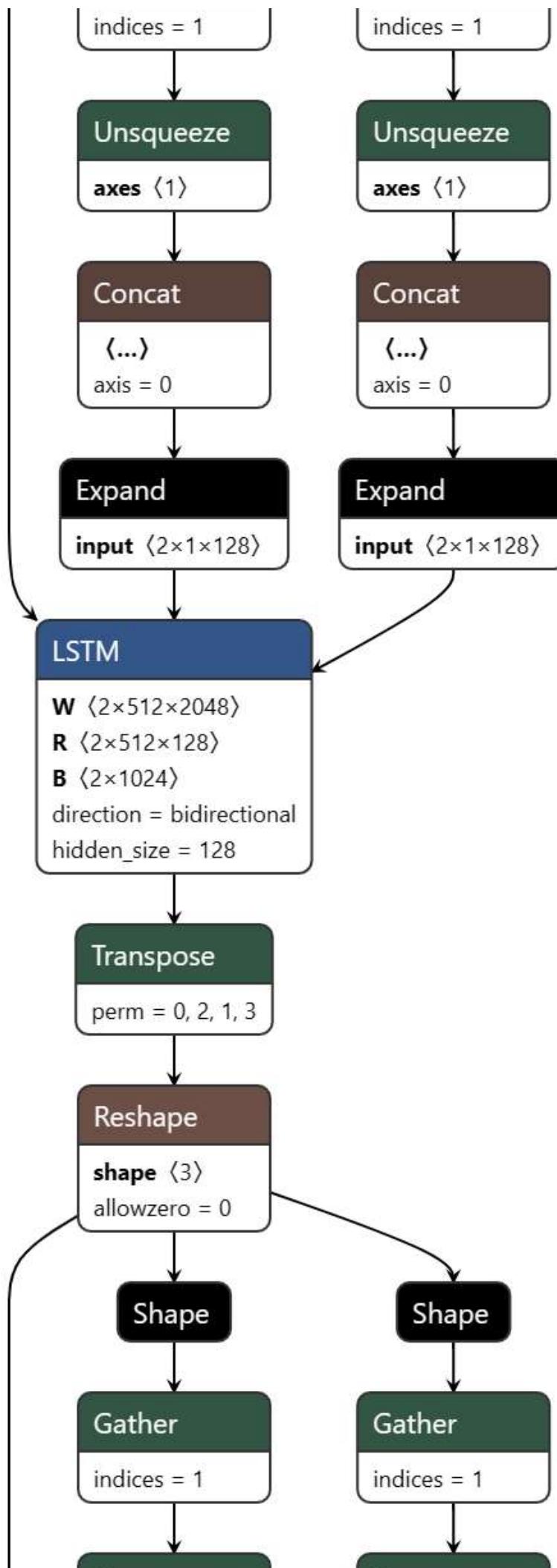
def forward(self, x):
    batch_size, time_steps, channel_x, height, width = x.shape
    conv_input = x.view(batch_size, time_steps, channel_x, height, width)
    conv_output = self.conv_model(conv_input)
    lstm_input = conv_output.view(batch_size, time_steps, -1)
    lstm_output = self.Lstm(lstm_input)
    lstm_output = lstm_output[:, -1, :]
    output = self.output_layer(lstm_output)
    return output
```

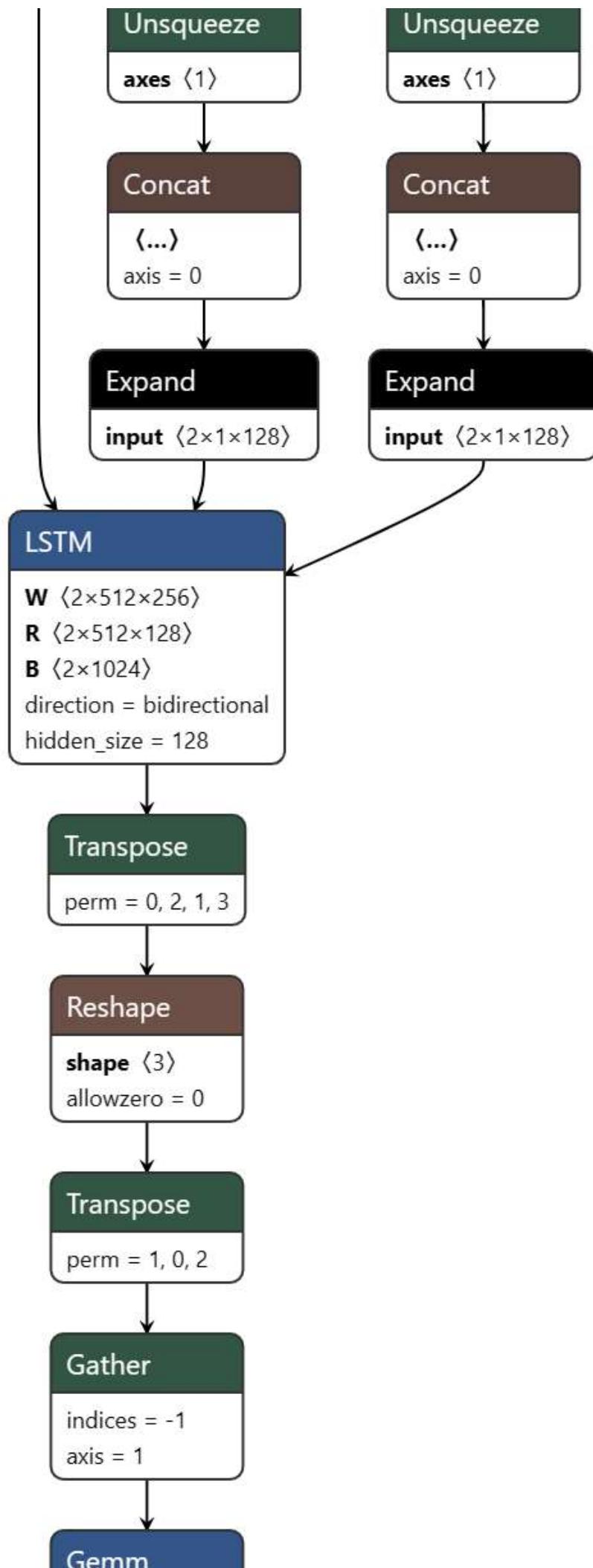
模型结构图如下：

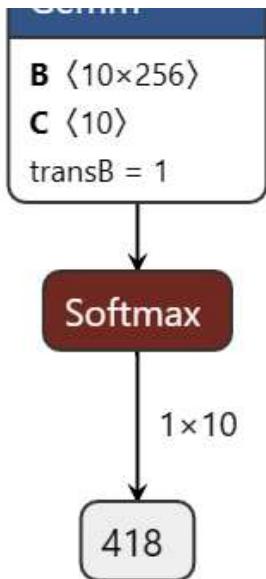












自定义的ResNet和LSTM构建模型设计

当前在图像分类任务上表现较好的CNN模型之一是ResNet（Residual Network）。ResNet通过引入残差连接（residual connection）解决了深层网络训练过程中的梯度消失和表示瓶颈问题。因此自定义搭建了一个基于ResNet的ResNetLSTM模型用于视频分类任务。

其主要代码和函数解释如下：

```

class ResidualBlock(nn.Module):
    """
    定义ResNet的残差块。该残差块包含两个卷积层，每个卷积层后面跟着一个批归一化层。
    :param in_channels: 输入的通道数
    :param out_channels: 输出的通道数
    :param stride: 卷积的步长，默认为1
    """

    def __init__(self, in_channels, out_channels, stride=1):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,
                            stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
                            stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        out += self.shortcut(residual)
        out = self.relu(out)
        return out

class ResNet(nn.Module):
    """
    定义ResNet模型，由多个残差块组成。
    :param latent_dim: 输出的特征维度
    """

    def __init__(self, latent_dim):
        super(ResNet, self).__init__()
        self.in_channels = 8
        self.conv1 = nn.Conv2d(3, 8, kernel_size=7, stride=2, padding=3, bias=False)
        self.bn1 = nn.BatchNorm2d(8)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self.make_layer(8, 2)
        self.layer2 = self.make_layer(16, 2, stride=2)
        self.layer3 = self.make_layer(32, 2, stride=2)
        self.layer4 = self.make_layer(64, 2, stride=2)

```

```

self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
self.fc = nn.Linear(64, latent_dim)

def make_layer(self, out_channels, num_blocks, stride=1):
    layers = []
    layers.append(ResidualBlock(self.in_channels, out_channels, stride))
    self.in_channels = out_channels
    for _ in range(1, num_blocks):
        layers.append(ResidualBlock(out_channels, out_channels))
    return nn.Sequential(*layers)

def forward(self, x):
    batch_size, time_steps, channel_x, height, width = x.shape
    conv_input = x.view(batch_size * time_steps, channel_x, height, width)
    x = self.conv1(conv_input)
    x = self.bn1(x)
    x = self.relu(x)
    x = self.maxpool(x)
    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)
    x = self.layer4(x)
    x = self.avgpool(x)
    x = x.view(x.size(0), -1)
    x = self.fc(x)
    return x

class ResNetLstm(nn.Module):
    """
    使用自定义的ResNet和LSTM构建模型
    :param latent_dim: LSTM的输入维度
    :param hidden_size: LSTM的隐藏层维度
    :param lstm_layers: LSTM的层数
    :param bidirectional: LSTM是否为双向
    :param n_class: 分类的类别数
    """

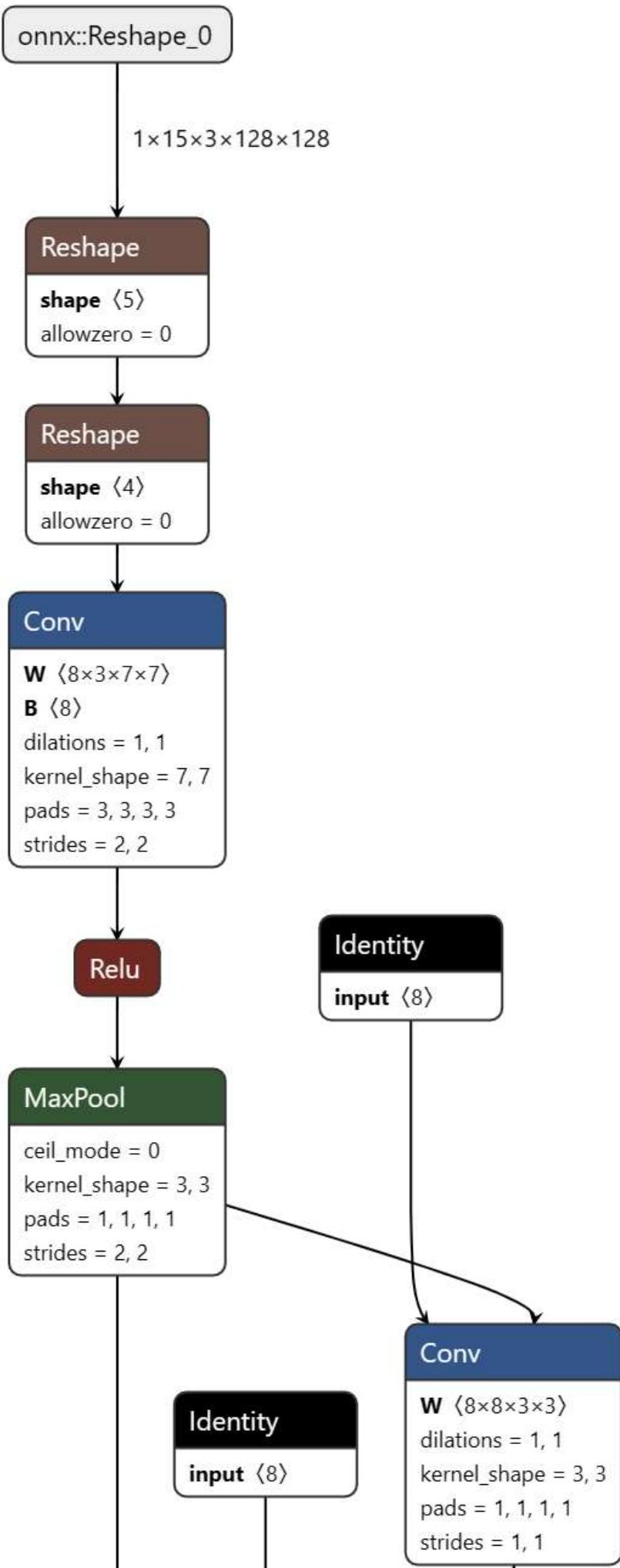
    def __init__(self, latent_dim, hidden_size, lstm_layers, bidirectional, n_class):
        super(ResNetLstm, self).__init__()
        self.conv_model = ResNet(latent_dim=latent_dim)
        self.Lstm = Lstm(latent_dim, hidden_size, lstm_layers, bidirectional)
        self.output_layer = nn.Sequential(
            nn.Linear(2 * hidden_size if bidirectional == True else hidden_size, n_class),
            nn.Softmax(dim=-1)
        )

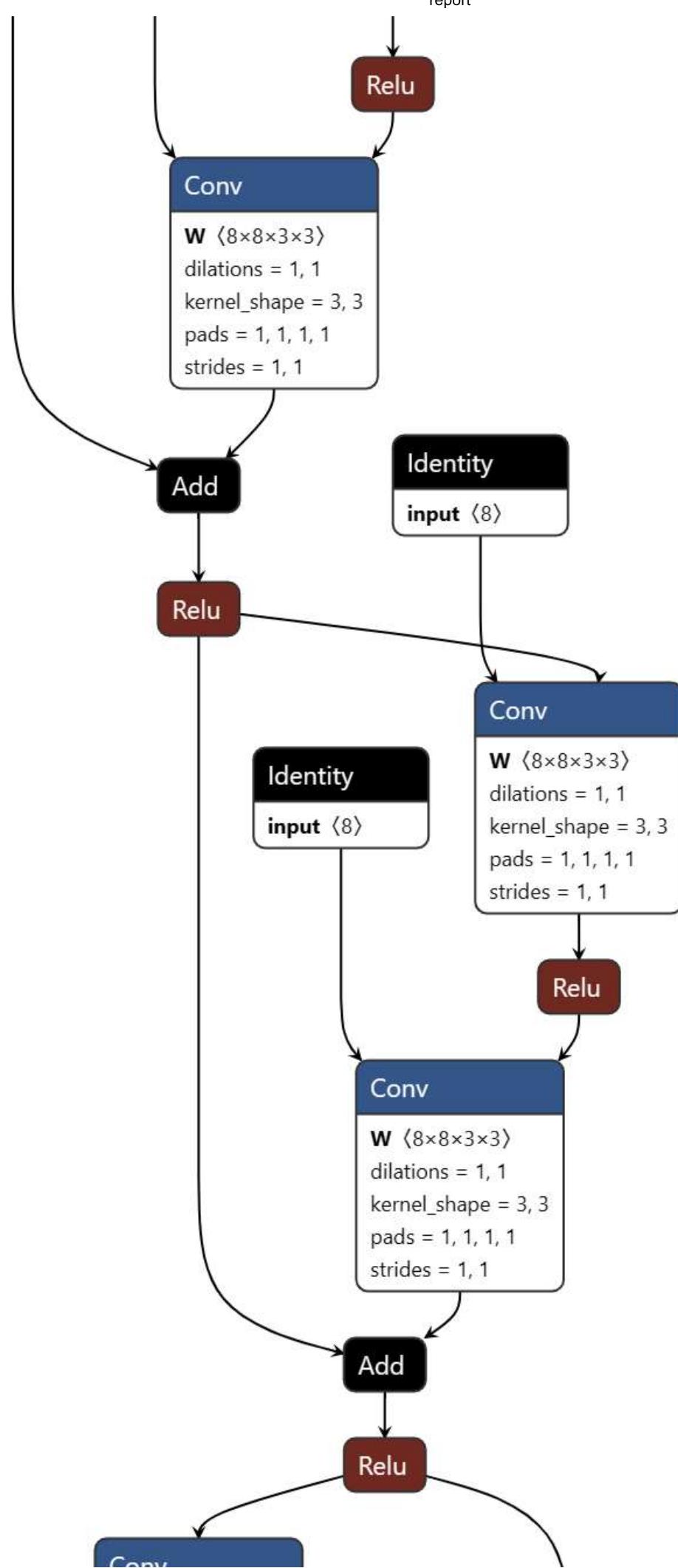
    def forward(self, x):
        batch_size, time_steps, channel_x, height, width = x.shape
        conv_input = x.view(batch_size, time_steps, channel_x, height, width)
        conv_output = self.conv_model(conv_input)
        lstm_input = conv_output.view(batch_size, time_steps, -1)
        lstm_output = self.Lstm(lstm_input)
        lstm_output = lstm_output[:, -1, :]

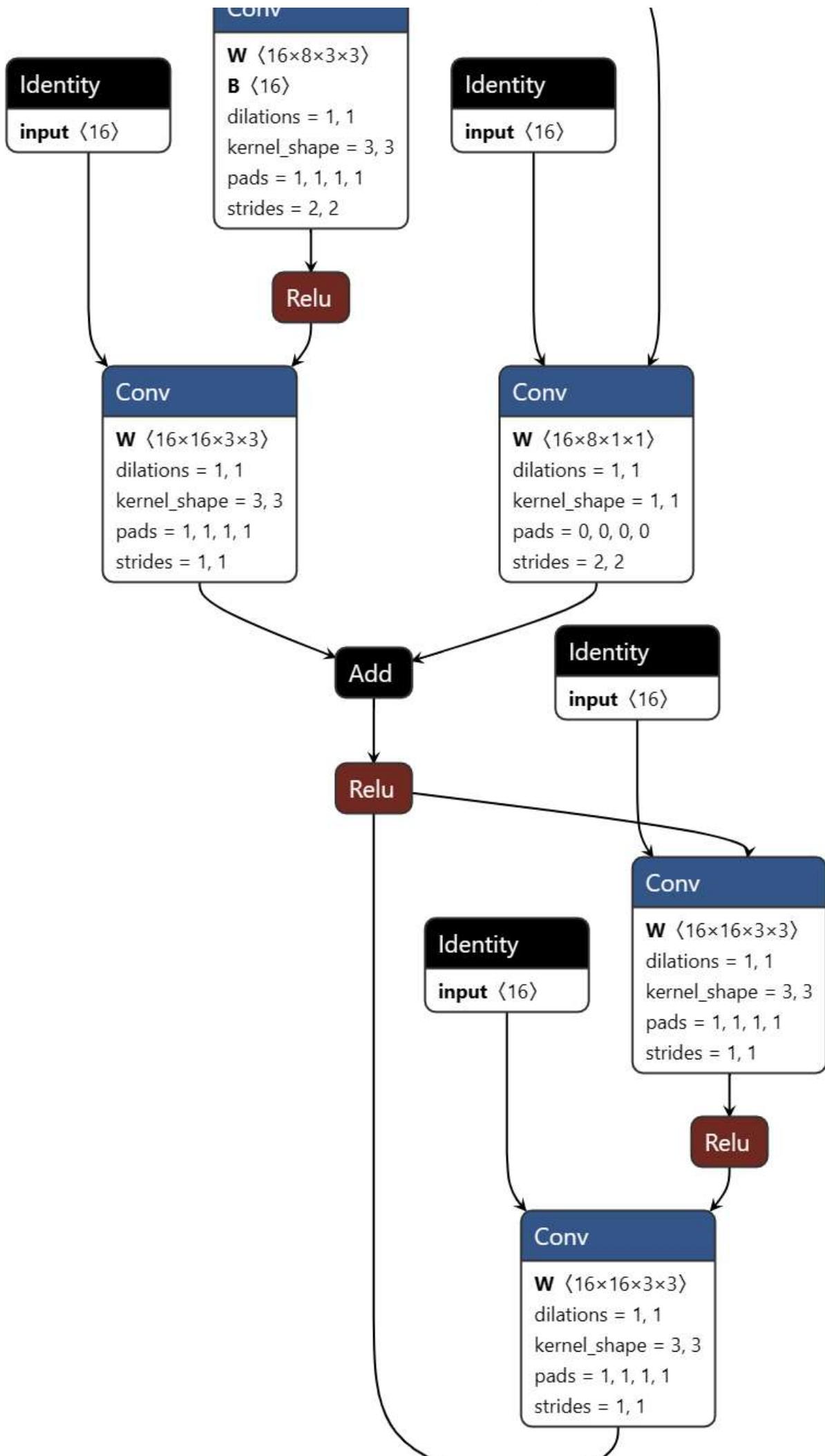
```

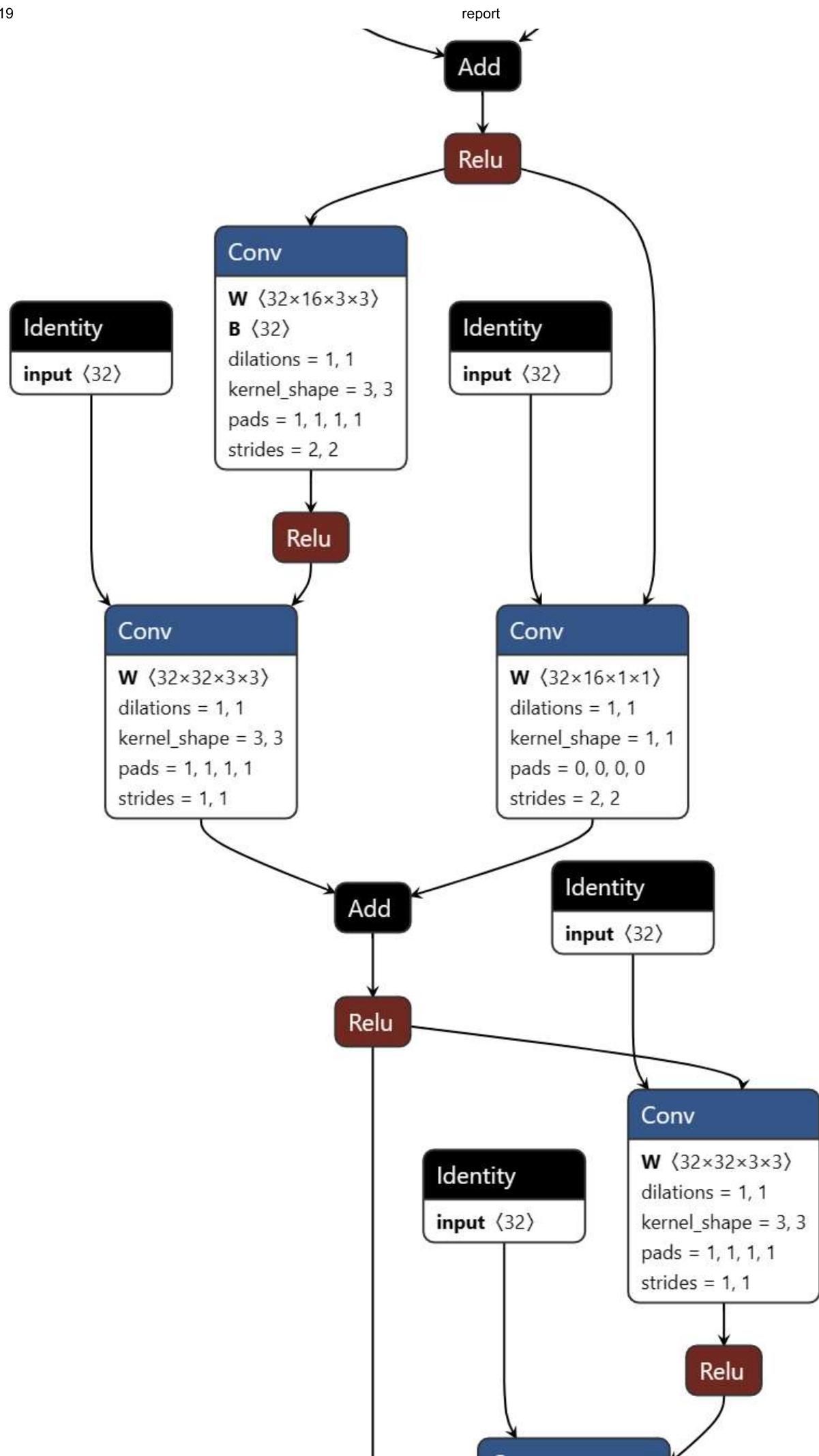
```
output = self.output_layer(lstm_output)
return output
```

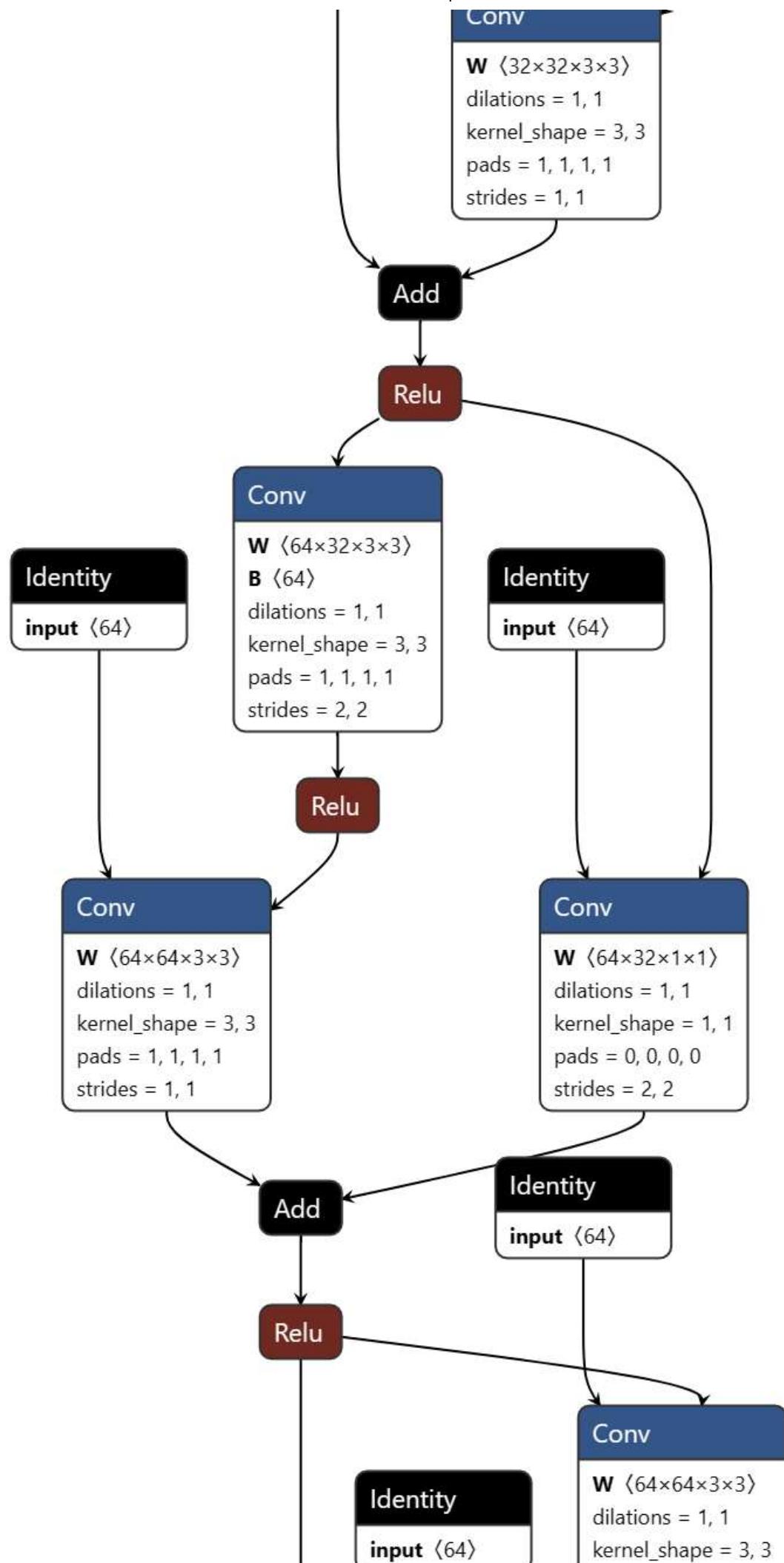
模型结构图如下：

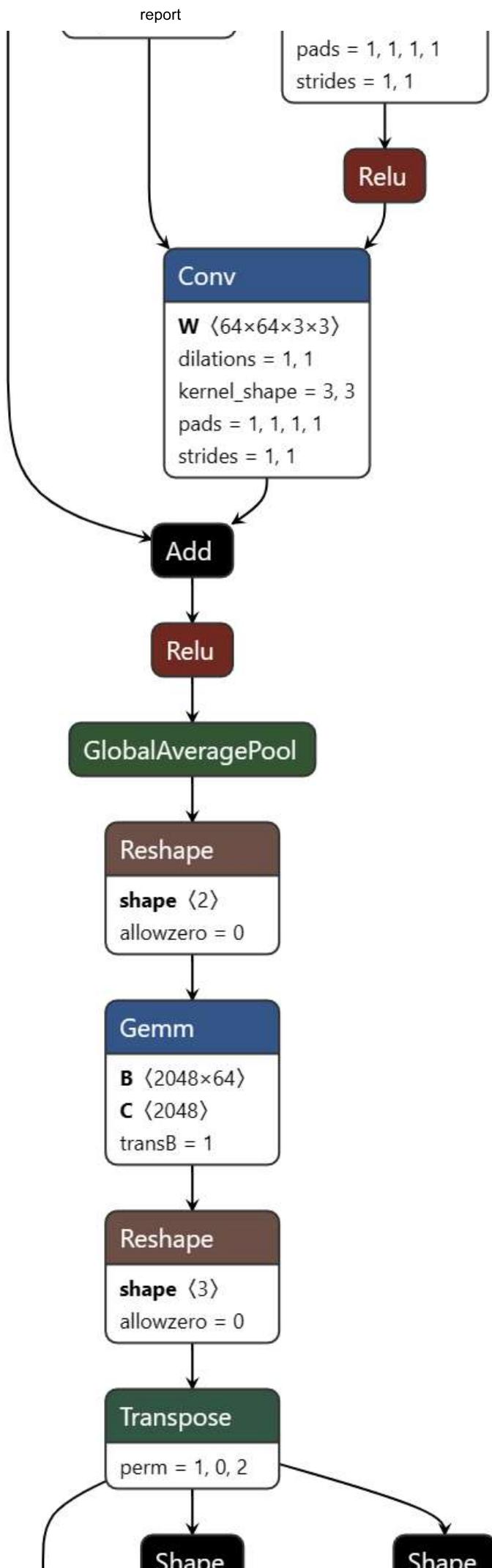


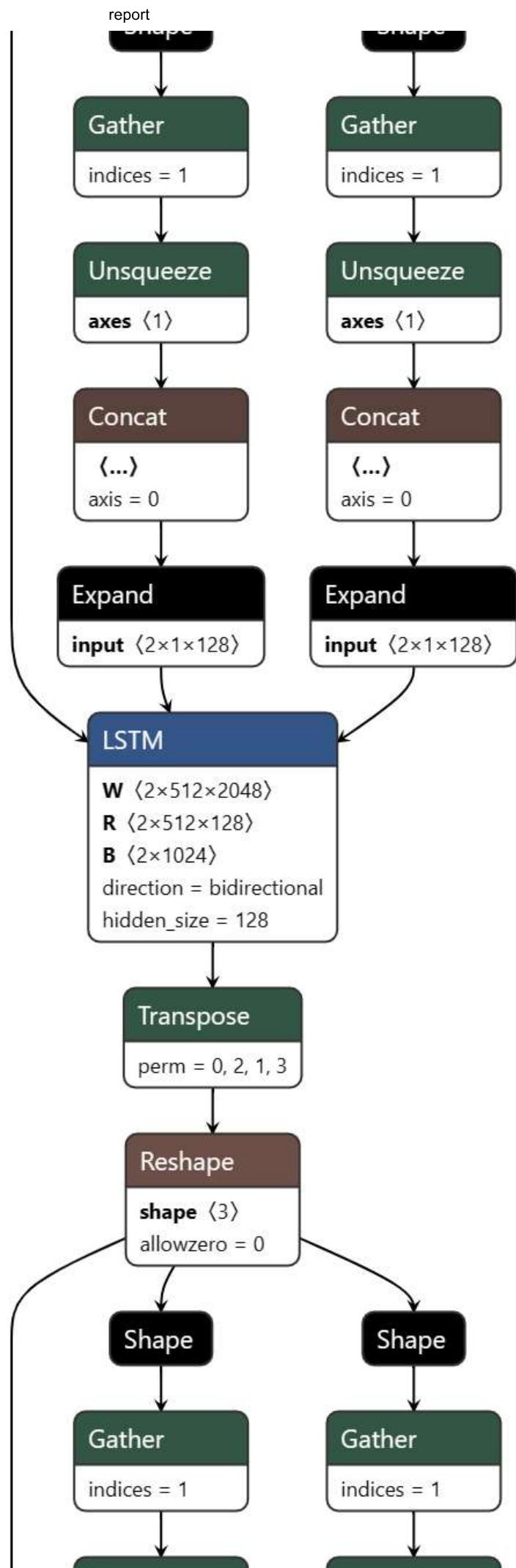


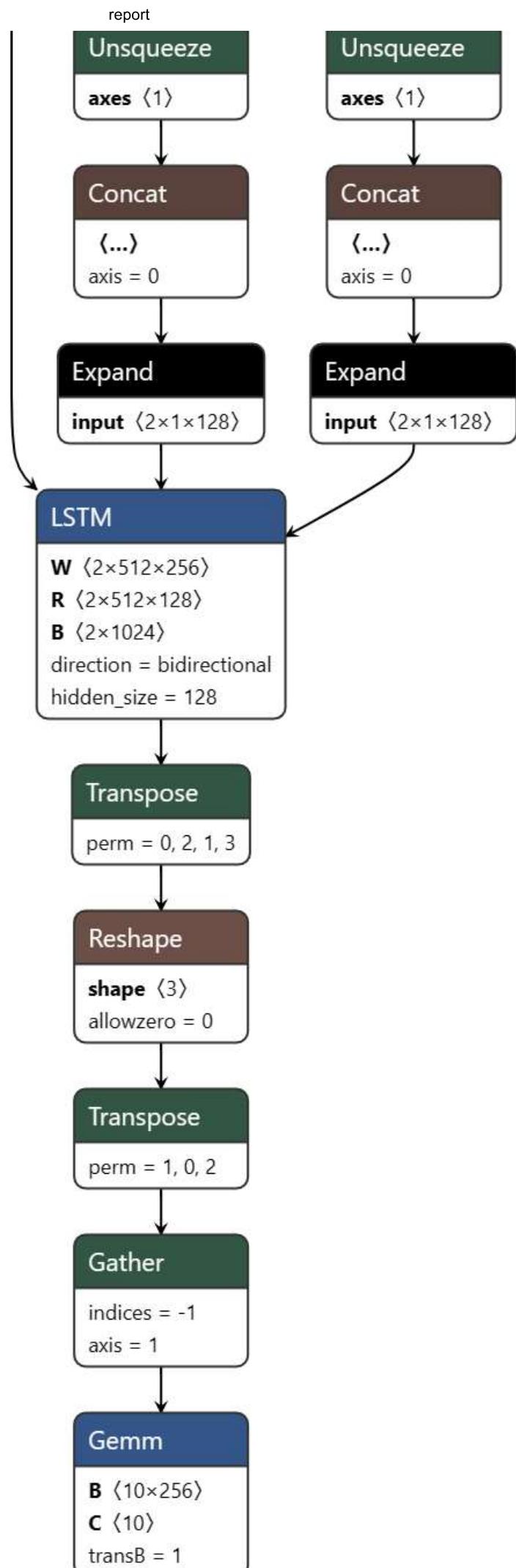


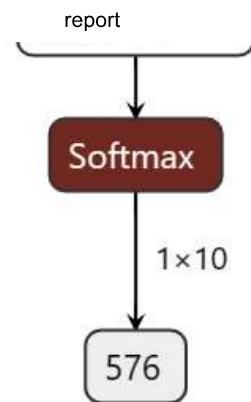












训练方法

训练函数主要包括：

1. 训练前先对模型进行评估，而后设定缺省的训练参数，而后开始训练。
2. 训练时先通过模型计算输出，而后计算损失，而后反向传播计算梯度，而后更新参数。
3. 每轮训练完成后，对模型进行评估，并根据学习率调度器调整学习率。
4. 训练完成后保存模型的权重。

主要代码解释如下：

```

def train(model, train_data, loss_fn, optimizer, epochs, device, save_last_weights_path=None,
          save_best_weights_path=None, steps_per_epoch=None,
          validation_data=None, scheduler=None):
    """
    训练模型

    :param model: 要训练的模型。
    :param train_data: 训练数据集的数据加载器。
    :param loss_fn: 损失函数。
    :param optimizer: 优化器。
    :param epochs: 训练的轮数。
    :param device: 训练设备。
    :param save_last_weights_path: 可选参数，保存最后模型权重的路径。
    :param save_best_weights_path: 可选参数，保存最佳模型权重的路径。
    :param steps_per_epoch: 可选参数，每个epoch的步数。
    :param validation_data: 可选参数，用于验证的数据加载器。
    :param scheduler: 可选参数，学习率调度器。
    :return:
    """

    if save_best_weights_path:
        # 评估当前模型在验证数据集上的损失
        best_loss, _ = evaluate(model, validation_data, loss_fn, device)

    if steps_per_epoch is None:
        # 如果没有指定每个epoch的步数，则将其设置为训练数据集的长度
        steps_per_epoch = len(train_data)

    num_steps = len(train_data)
    iterator = iter(train_data)
    count_steps = 1

    history = {
        'train_loss': [],
        'train_acc': [],
        'val_acc': [],
        'val_loss': []
    }

    # 将模型移动到设备上
    model = model.to(device)

    # 遍历每个epoch
    for epoch in range(1, epochs + 1):

        running_loss = 0.
        train_correct = 0
        train_total = steps_per_epoch * train_data.batch_size

        model.train()

        for step in tqdm(range(steps_per_epoch), desc=f'epoch: {epoch}/{epochs}: ', ncols=100):
            img_batch, label_batch = next(iterator)
            img_batch, label_batch = img_batch.to(device), label_batch.to(device)

```

```

# 将梯度置零
optimizer.zero_grad()
# 前向传播计算输出
output_batch = model(img_batch)
# 计算损失
loss = loss_fn(output_batch, label_batch.long())
# 反向传播计算梯度
loss.backward(retain_graph=True)
# 更新参数
optimizer.step()
# 预测标签
_, predicted_labels = torch.max(output_batch.data, dim=1)
# 统计正确预测的数量
train_correct += (label_batch == predicted_labels).sum().item()
# 计算平均损失
running_loss += loss.item()
# 打印训练损失和准确率
if count_steps == num_steps:
    # 循环迭代器，以便继续训练数据集的下一个epoch
    count_steps = 0
    iterator = iter(train_data)
    count_steps += 1

train_loss = running_loss / steps_per_epoch
train_accuracy = train_correct / train_total

if scheduler:
    # 如果提供了学习率调度器，则根据训练损失调整学习率
    scheduler.step(train_loss)

history['train_loss'].append(float(train_loss))
history['train_acc'].append(float(train_accuracy))

# 评估模型在验证数据集上的性能
val_loss, val_acc = evaluate(model, validation_data, loss_fn, device)
# 打印训练损失和准确率
print(
    f'epoch: {epoch}, train_accuracy: {train_accuracy:.2f}, loss: {train_loss:.3f},
    val_accuracy: {val_acc:.2f}, val_loss: {val_loss:.3f}')

if save_best_weights_path:
    if val_loss < best_loss:
        # 如果验证损失更小，则保存模型的权重
        best_loss = val_loss
        torch.save(model.state_dict(), save_best_weights_path)
        print(f'Saved successfully best weights to:', save_best_weights_path)
history['val_loss'].append(float(val_loss))
history['val_acc'].append(float(val_acc))

if save_last_weights_path:
    # 如果提供了保存最后权重的路径，则保存模型的权重
    torch.save(model.state_dict(), save_last_weights_path)
    print(f'Saved successfully last weights to:', save_last_weights_path)

```

```
return model, history
```

训练过程 & 调参实验及结果

三种模型中，训练效果最好的分别为：

模型	训练次数	训练轮数	测试集Loss	测试集Acc	训练集Loss	训练集Acc
调用预训练CNN的模型	第一次训练	70	1.668	0.820	1.588	0.885
自定义的普通CNN的模型	第五次训练	120	1.730	0.750	1.523	0.945
自定义的ResNet模型	第四次训练	70	1.893	0.580	1.499	0.968

主要参数说明

在训练过程中不变的参数设置和说明如下：

参数名	参数值	说明
num_classes	10	视频分类的类别数
batch_size	4	每个batch的大小
num_workers	4	数据加载器的线程数
device	cuda	训练设备
epochs	70	训练的轮数

与训练过程相关的参数说明如下：

参数名	说明
num_frames	每个视频的帧数
img_size	图像的像素大小
latent_dim	CNN的输出维度
hid_size	LSTM的隐藏层维度
num_lstm_layers	LSTM的层数
learning_rate	学习率

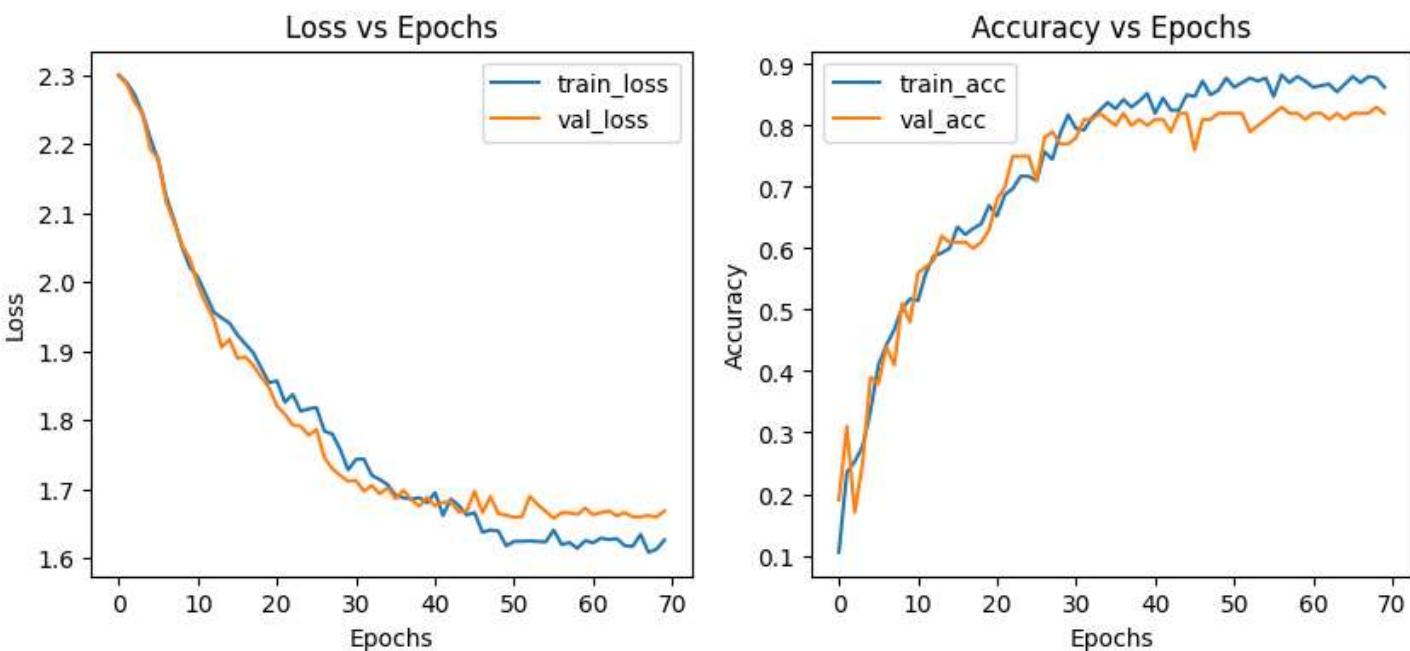
调用预训练CNN的模型训练过程及结果

第一次训练

参数设置如下：

参数名	参数值
num_frames	15
img_size	(128, 128)
latent_dim	2048
hid_size	128
num_lstm_layers	2
learning_rate	2e-5

训练过程和评估结果如下：



测试集：Loss: 1.668, Acc: 0.820

训练集：Loss: 1.588, Acc: 0.885

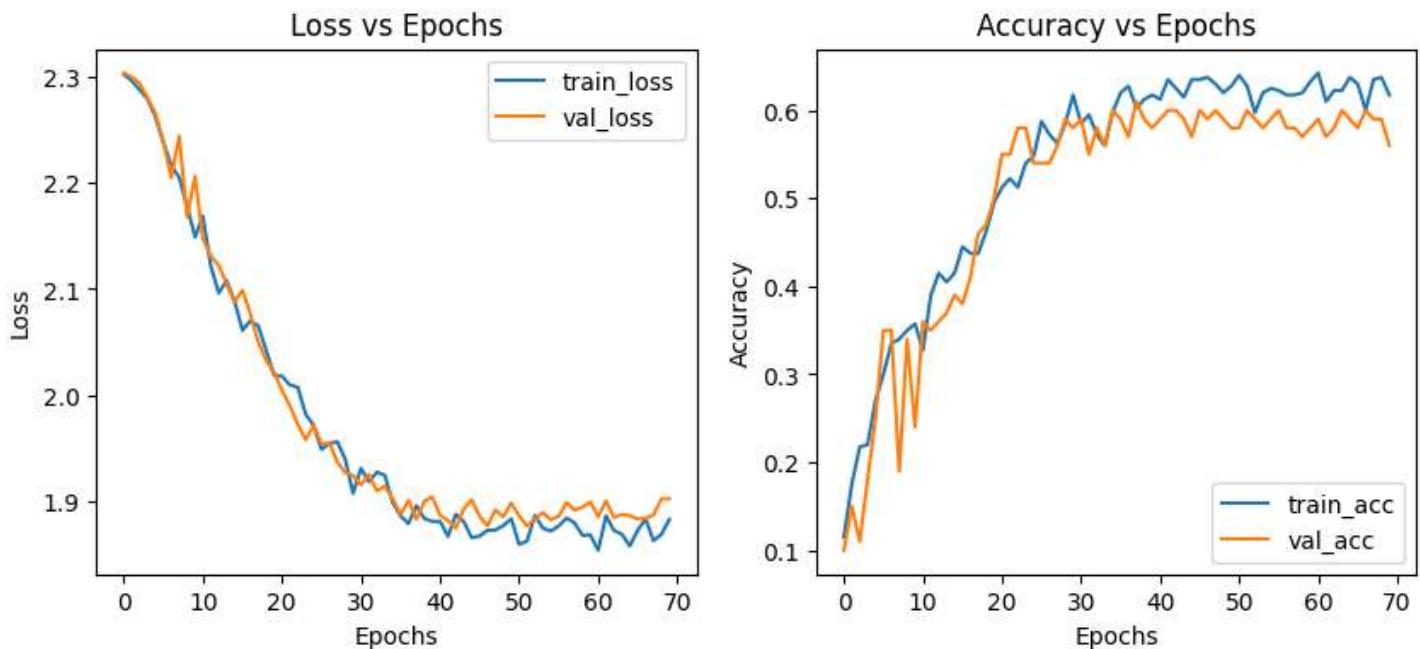
训练时长：24 min 38 s

第二次训练

参数设置如下：

参数名	参数值
num_frames	20
img_size	(64, 64)
latent_dim	2048
hid_size	128
num_lstm_layers	2
learning_rate	2e-5

训练过程和评估结果如下：



测试集：Loss: 1.903, Acc: 0.560

训练集：Loss: 1.844, Acc: 0.647

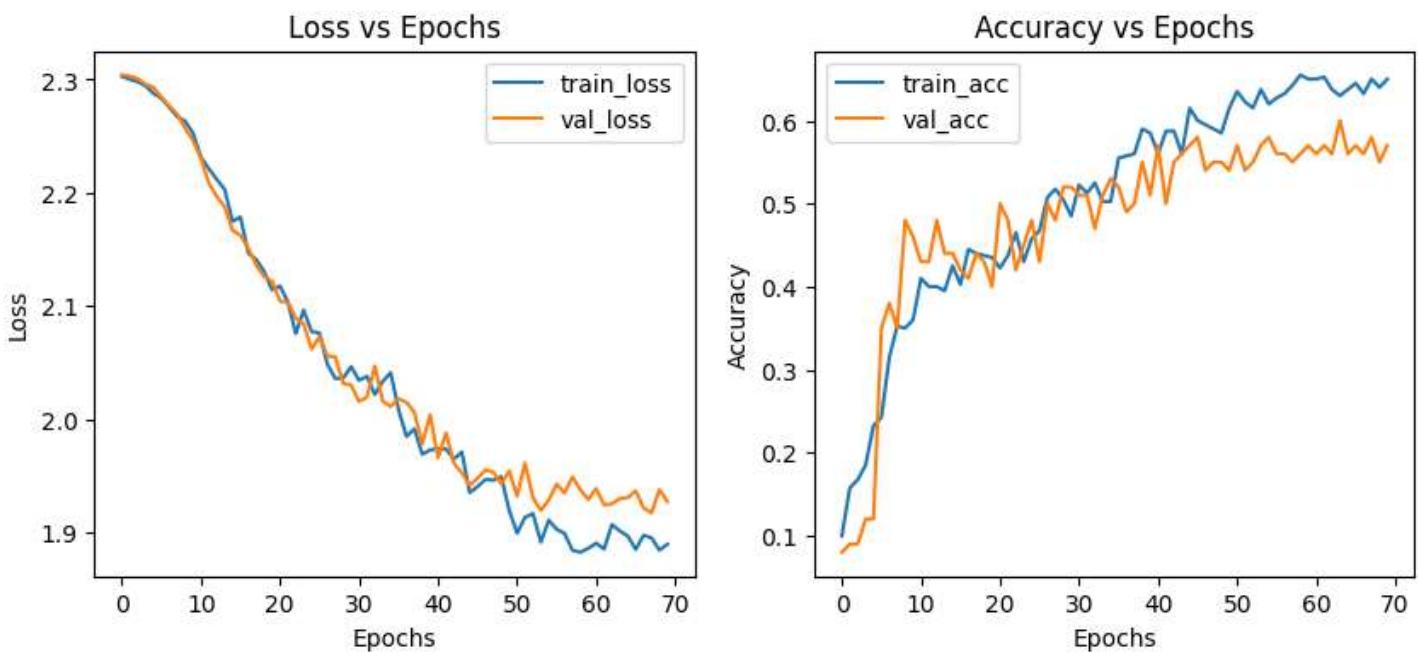
训练时长：17 min 27 s

第三次训练

参数设置如下：

参数名	参数值
num_frames	20
img_size	(64, 64)
latent_dim	1024
hid_size	64
num_lstm_layers	2
learning_rate	2e-5

训练过程和评估结果如下：



测试集: Loss: 1.928, Acc: 0.570

训练集: Loss: 1.870, Acc: 0.640

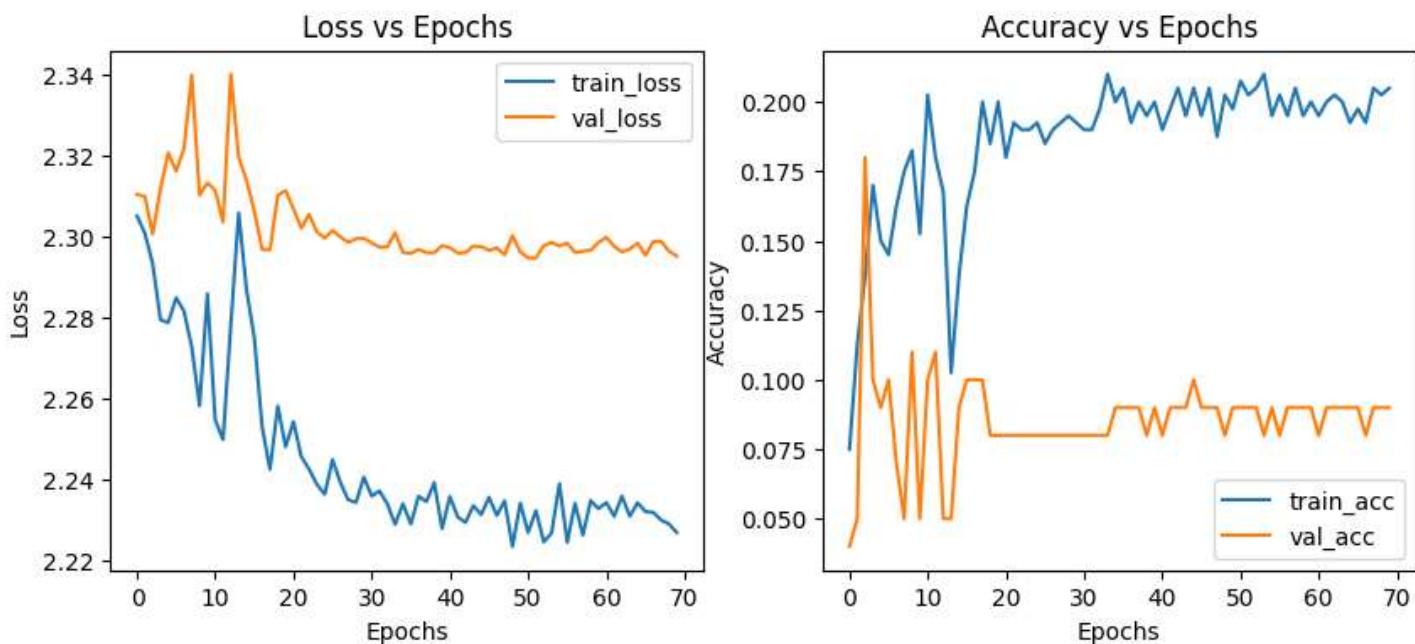
训练时长: 13 min 49 s

第四次训练

参数设置如下:

参数名	参数值
num_frames	15
img_size	(128, 128)
latent_dim	2048
hid_size	128
num_lstm_layers	2
learning_rate	2e-4

训练过程和评估结果如下:



测试集: Loss: 2.295, Acc: 0.090

训练集: Loss: 2.222, Acc: 0.212

训练时长: 24 min 32 s

对比

参数名	第一次训练	第二次训练	第三次训练	第四次训练
num_frames	15	20	20	15
img_size	(128, 128)	(64, 64)	(64, 64)	(128, 128)
latent_dim	2048	2048	1024	2048
hid_size	128	128	64	128
num_lstm_layers	2	2	2	2
learning_rate	2e-5	2e-5	2e-5	2e-4
测试集Loss	1.668	1.903	1.928	2.295
测试集Acc	0.820	0.560	0.570	0.090
训练集Loss	1.588	1.844	1.870	2.222
训练集Acc	0.885	0.647	0.640	0.212
训练时长	24 min 38 s	17 min 27 s	13 min 49 s	24 min 32 s

对比第一次训练和第二次训练:

修改参数:

- num_frames: 15 → 20
- img_size: (128, 128) → (64, 64)

第一次训练的训练集、测试集准确率远高于第二次训练，但训练时长较长，说明输入视频的帧数和图像的像素大小对模型的训练效果有较大影响。

对比第二次训练和第三次训练：

修改参数：

- latent_dim: 2048 → 1024
- hid_size: 128 → 64

第二次训练的训练集、测试集准确率与第三次训练相当，但训练时长较长，说明 CNN 的输出维度和 LSTM 的隐藏层维度对模型的训练效果影响不大。

对比第一次训练和第四次训练：

修改参数：

- learning_rate: 2e-5 → 2e-4

两次训练的时长相同，但第一次训练的训练集、测试集准确率远高于第四次训练，说明在当前模型结构下，学习率为 2e-5 的效果更好。

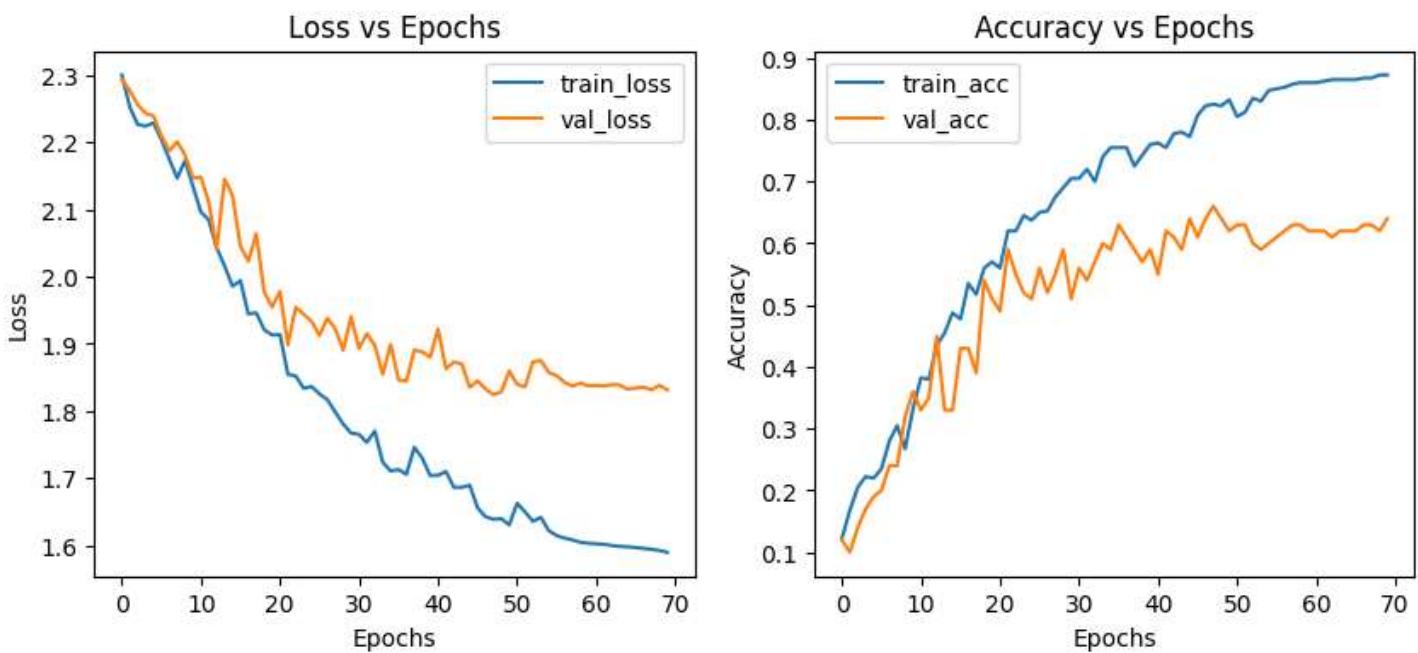
调用自行定义的普通CNN的模型训练过程及结果

第一次训练

参数设置如下：

参数名	参数值
num_frames	15
img_size	(128, 128)
latent_dim	2048
hid_size	128
num_lstm_layers	2
learning_rate	2e-4

训练过程和评估结果如下：



测试集: Loss: 1.831, Acc: 0.640

训练集: Loss: 1.587, Acc: 0.875

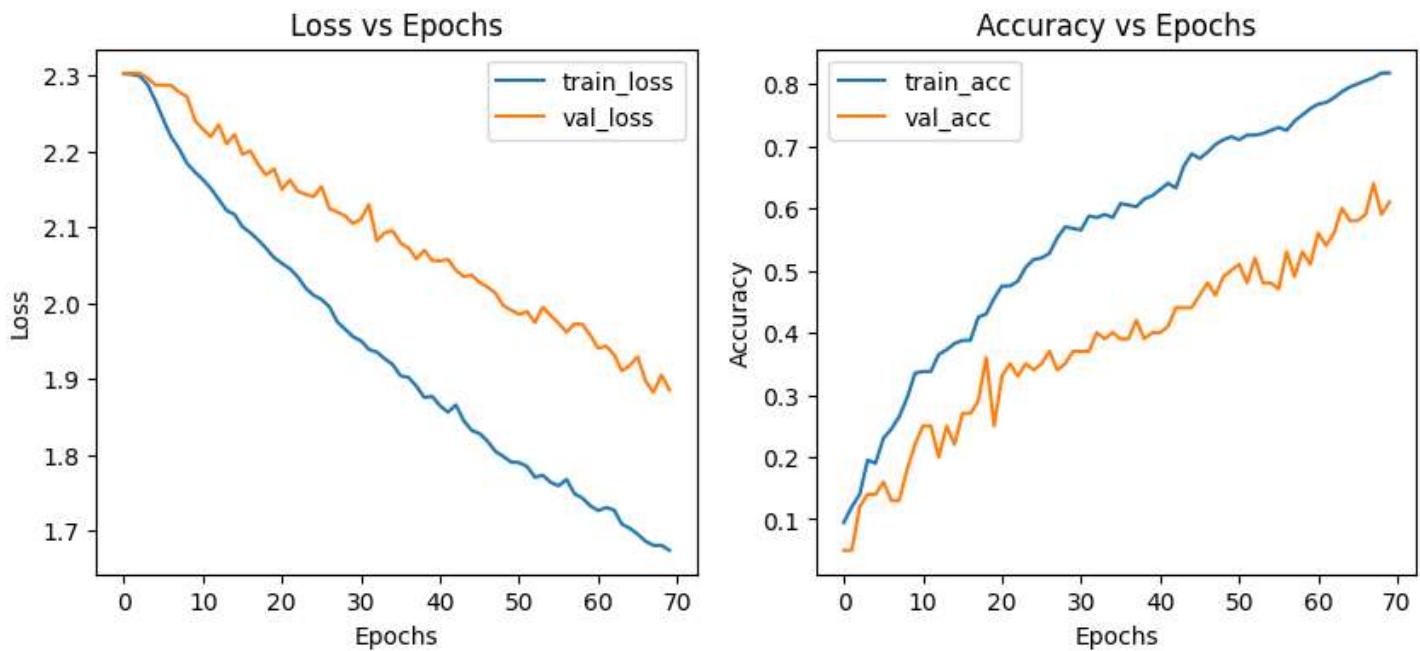
训练时长: 31 min 14 s

第二次训练

参数设置如下:

参数名	参数值
num_frames	15
img_size	(128, 128)
latent_dim	2048
hid_size	128
num_lstm_layers	2
learning_rate	2e-5

训练过程和评估结果如下:



测试集: Loss: 1.883, Acc: 0.610

训练集: Loss: 1.660, Acc: 0.825

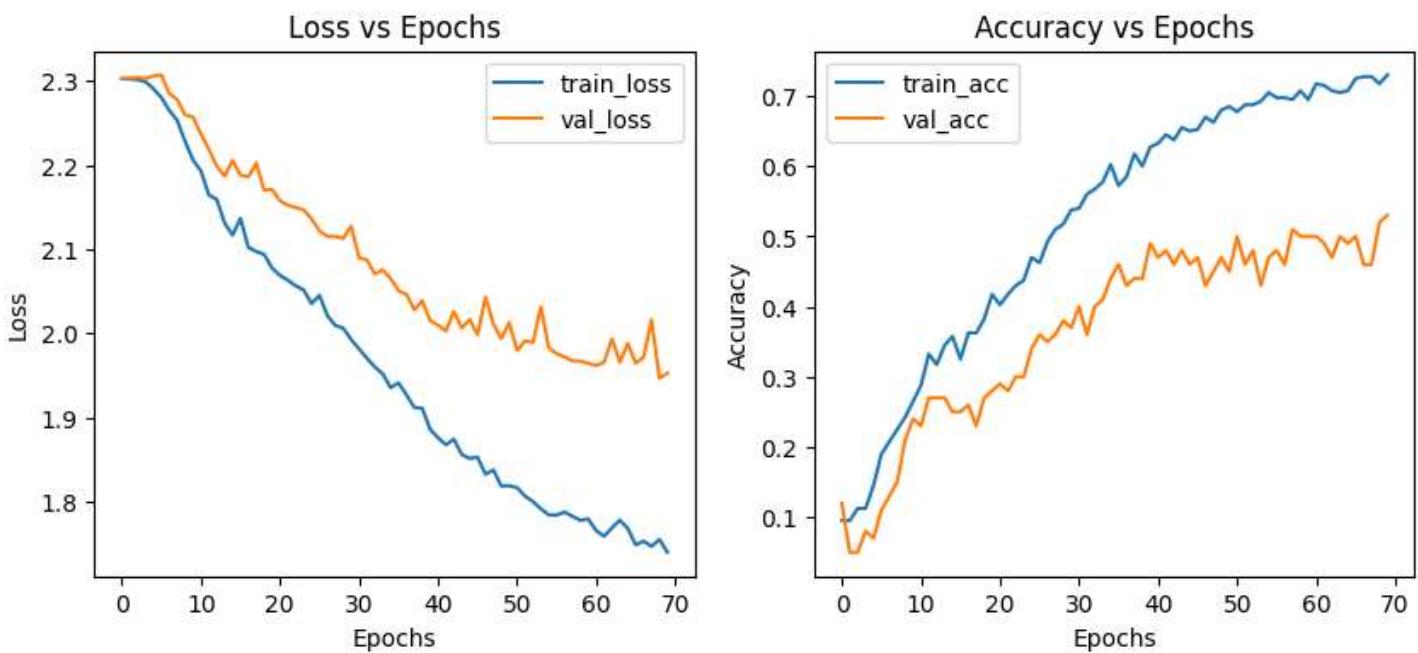
训练时长: 31 min 24 s

第三次训练

参数设置如下:

参数名	参数值
num_frames	15
img_size	(128, 128)
latent_dim	2048
hid_size	128
num_lstm_layers	3
learning_rate	2e-5

训练过程和评估结果如下:



测试集: Loss: 1.952, Acc: 0.530

训练集: Loss: 1.735, Acc: 0.735

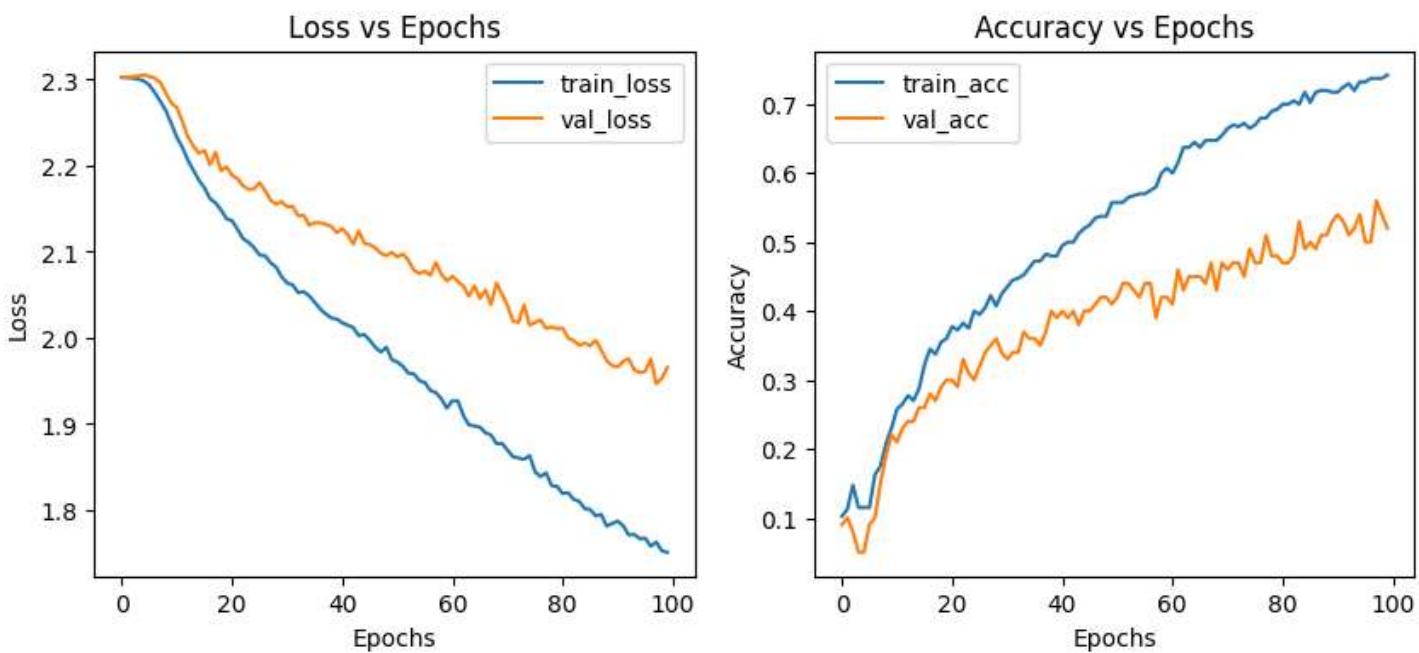
训练时长: 33 min 41 s

第四次训练

参数设置如下:

参数名	参数值
num_frames	15
img_size	(128, 128)
latent_dim	2048
hid_size	128
num_lstm_layers	2
learning_rate	1e-5

训练过程和评估结果如下:



测试集：Loss: 1.966, Acc: 0.520

训练集：Loss: 1.754, Acc: 0.738

训练时长：33 min 41 s

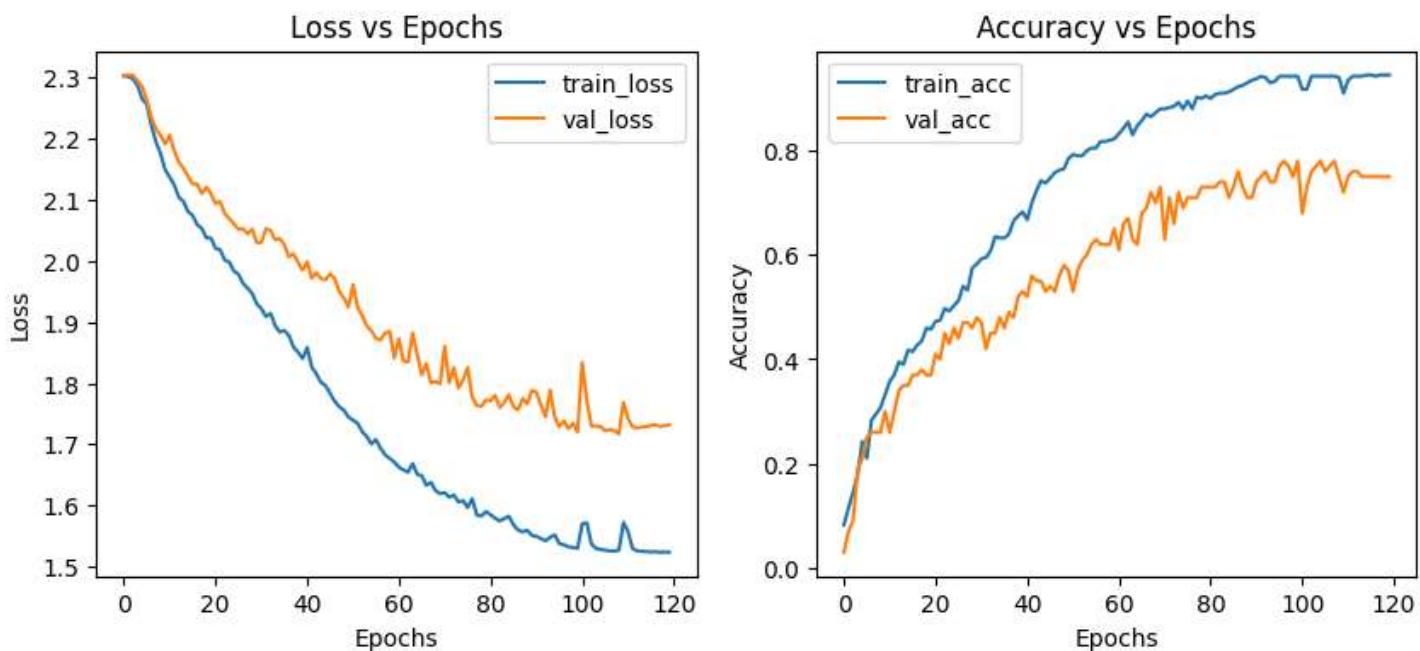
第五次训练

通过观察发现第二次训练随着训练轮数的增加，训练的效果在不断上升，因此采用与第二次相近的训练参数（将 num_frames 由 15 改为 20），将训练轮数更改为 120，以使自定义模型达到精度要求。

参数设置如下：

参数名	参数值
num_frames	20
img_size	(128, 128)
latent_dim	2048
hid_size	128
num_lstm_layers	2
learning_rate	2e-5

训练过程和评估结果如下：



测试集：Loss: 1.730, Acc: 0.750

训练集：1.523, Acc: 0.945

训练时长：62 min

对比

参数名	第一次训练	第二次训练	第三次训练	第四次训练	第五次训练
num_frames	15	15	15	15	20
img_size	(128, 128)	(128, 128)	(128, 128)	(128, 128)	(128, 128)
latent_dim	2048	2048	2048	2048	2048
hid_size	128	128	128	128	128
num_lstm_layers	2	2	3	2	2
learning_rate	2e-4	2e-5	2e-5	1e-5	2e-5
测试集Loss	1.831	1.883	1.952	1.966	1.730
测试集Acc	0.640	0.610	0.530	0.520	0.750
训练集Loss	1.587	1.660	1.735	1.754	1.523
训练集Acc	0.875	0.825	0.735	0.738	0.945
训练时长	31 min 14 s	31 min 24 s	33 min 41 s	33 min 41 s	62 min

对比第一次、第二次、第四次训练：

这三次训练均只修改了学习率，三次训练时间相近，第一次和第二次的训练效果优于第三次。

第一次的训练准确度在 20 轮之后趋于稳定，第二次的训练准确度随着训练次数的增加在不断上升，说明在自定义的模型下，学习率为 2e-4 训练更快，但是存在瓶颈，而学习率为 2e-5 的训练效果更好。

对比第二次、第三次训练：

修改参数：

- num_lstm_layers: 2 → 3

两次训练时间相近，第二次的训练准确度优于第三次。

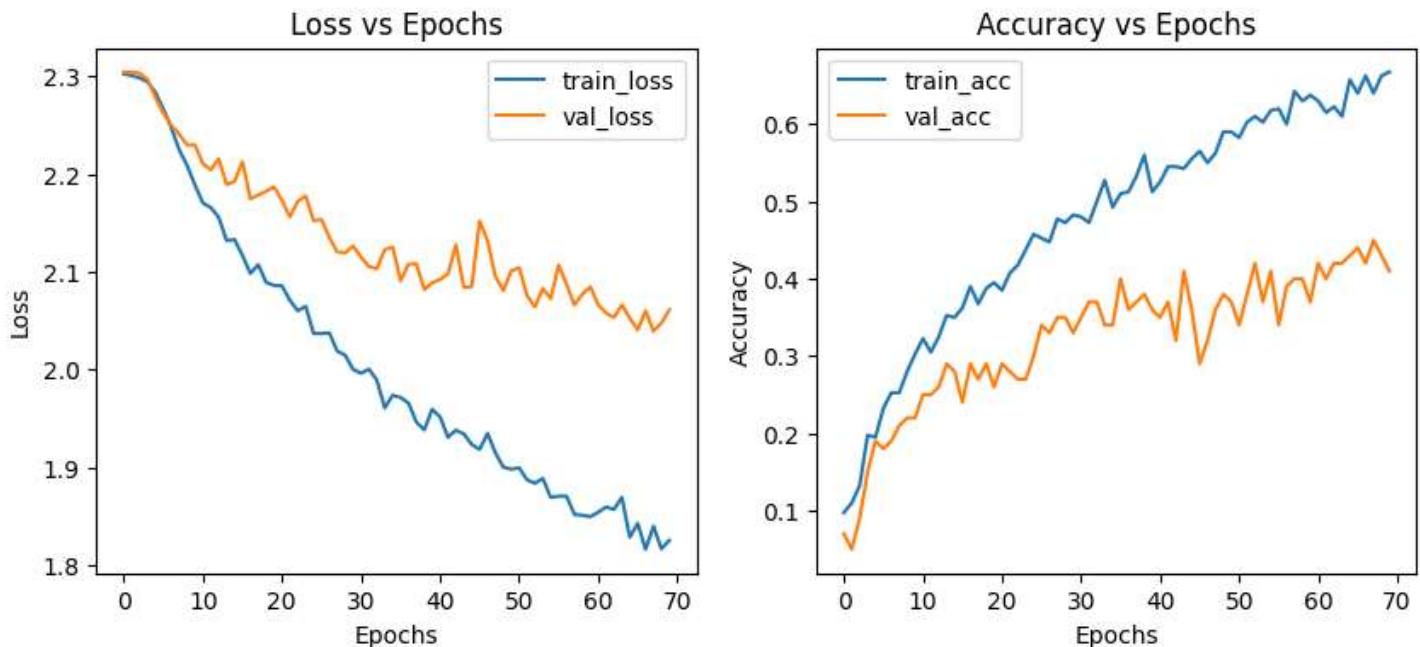
使用自定义的ResNet和LSTM构建模型训练过程及结果

第一次训练

参数设置如下：

参数名	参数值
num_frames	20
img_size	(128, 128)
latent_dim	2048
hid_size	128
num_lstm_layers	2
learning_rate	2e-5

训练过程和评估结果如下：



测试集：Loss: 2.062, Acc: 0.410

训练集：Loss: 1.769, Acc: 0.705

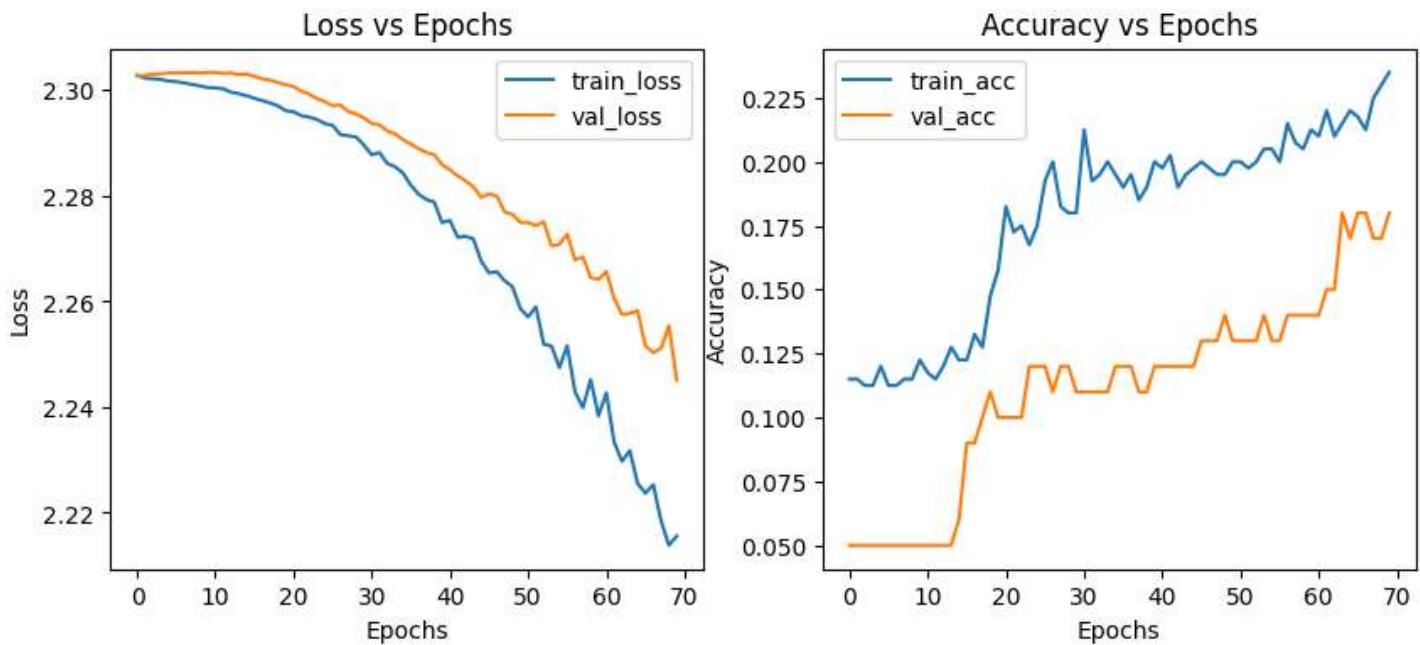
训练时长：33 min 41 s

第二次训练

参数设置如下：

参数名	参数值
num_frames	20
img_size	(128, 128)
latent_dim	2048
hid_size	128
num_lstm_layers	2
learning_rate	2e-6

训练过程和评估结果如下：



测试集：Loss: 2.245, Acc: 0.190

训练集：Loss: 2.196, Acc: 0.300

训练时长：56 min 8 s

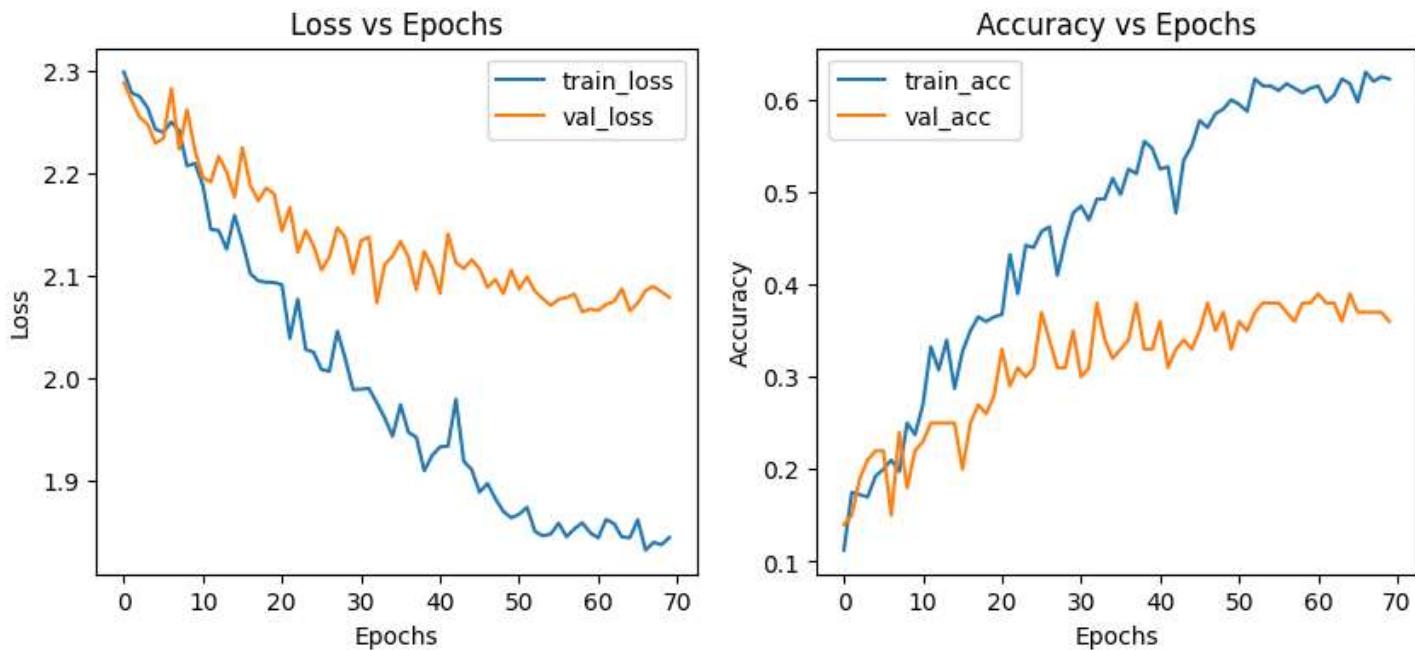
第三次训练

参数设置如下：

参数名	参数值
num_frames	20
img_size	(128, 128)
latent_dim	2048
hid_size	128

参数名	参数值
num_lstm_layers	2
learning_rate	2e-4

训练过程和评估结果如下：



测试集：Loss: 2.080, Acc: 0.360

训练集：Loss: 1.820, Acc: 0.637

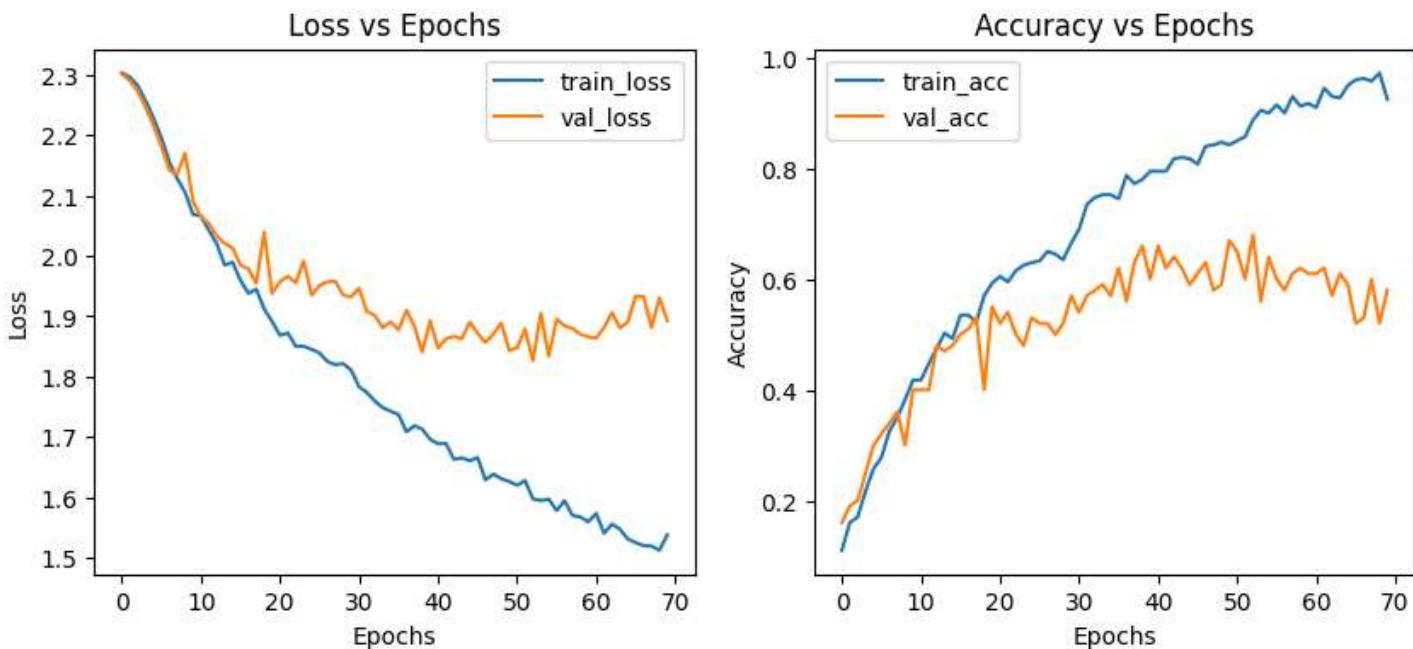
训练时长：55 min 54 s

第四次训练

参数设置如下：

参数名	参数值
num_frames	20
img_size	(128, 128)
latent_dim	2048
hid_size	128
num_lstm_layers	2
learning_rate	5e-5

训练过程和评估结果如下：



测试集：Loss: 1.893, Acc: 0.580

训练集：Loss: 1.499, Acc: 0.968

训练时长：55 min 58 s

对比

参数名	第一次训练	第二次训练	第三次训练	第四次训练
learning_rate	2e-5	2e-6	2e-4	5e-5
测试集Loss	2.062	2.245	2.080	1.893
测试集Acc	0.410	0.190	0.360	0.580
训练集Loss	1.769	2.196	1.820	1.499
训练集Acc	0.705	0.300	0.637	0.968
训练时长	33 min 41 s	55 min 10 s	55 min 54 s	55 min 58 s

四次训练均只改变了学习率，通过观察发现在学习率为 5e-5 时，训练集准确率达到 0.968，测试集准确率达到 0.580，均为最高，因此在自定义的 ResNetLSTM 模型下，学习率为 5e-5 的效果最好。

采用残差网络的 ResNetLSTM 模型在训练集上的准确率均远大于测试集，训练效果也并不好，说明该模型的结构存在问题，也存在过拟合现象，导致准确率低于使用普通 CNN 的模型。

视频分类效果展示

在分类时使用第五次训练的自定义CNN模型进行分类任务，通过运行 `classify.py` 文件，可以对指定的视频进行分类。

一次样例运行结果如下：

```
73 # 加载预训练的模型权重
74 model_path = "D:/course/MachineLearning/lab/lab3/result/best_model_ConvLstm.pth"
75 model = ConvLstm(latent_dim=latent_dim, hidden_size=hid_size, lstm_layers=num_lstm_layers,
76                  bidirectional=True, n_class=num_classes)
77 model.load_state_dict(torch.load(model_path, map_location=device))
78
79 # 对视频进行分类
80 video_path = "D:/course/MachineLearning/lab/lab3/data/HandstandWalking/v_HandstandWalking_g04_c01.avi"
81 predicted_class = predict_video_category(video_path, model, transform)
82 print("Predicted class:", predicted_class)
83
```

The screenshot shows a terminal window with the following details:

- Tab bar: 运行 (Run) - classify
- Icon bar: Run, Stop, Minimize, Maximize, Close, More
- Text area:
 - 命令: D:\Program\python-3.10.9\python.exe D:\course\MachineLearning\lab\lab3\classify.py
 - 输出: Predicted class: HandstandWalking
 - 结束语: 进程已结束, 退出代码为 0