EMS-YOLO

论文《Deep Directly-Trained Spiking Neural Networks for Object Detection》

动机

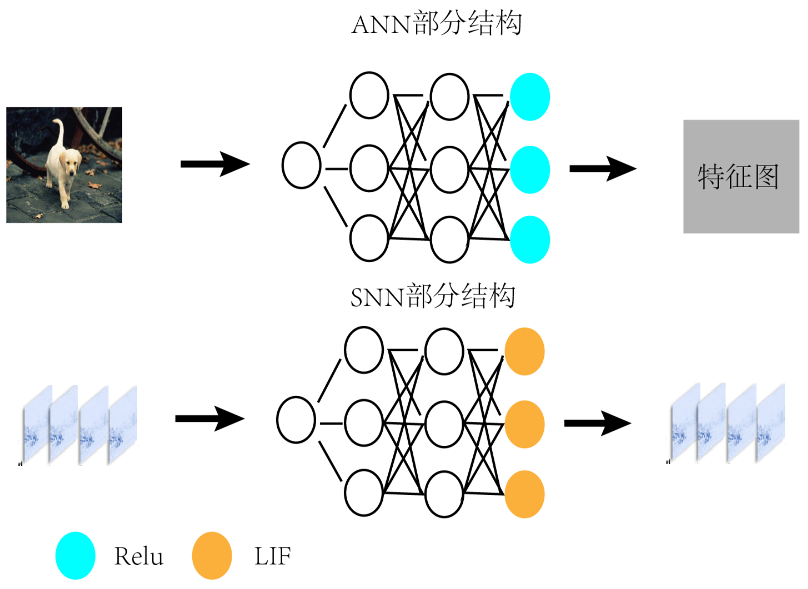
- 大多数应用于目标检测的脉冲神经网络是基于ANN转SNN的，且需要较大的时间步长。
- 大多数转换后的SNN方法只适用于静态图像，不适合DVS数据，因为它们的动力学旨在近似ANN的预期激活，无法捕获DVS数据的时空信息。
- 处理目标检测中的多尺度对象特征，需要网络具有足够的表示能力，要求更深的网络结构。
    - （现有的直接训练的SNN在图像分类任务上有了一些工作MS-ResNet 和 SEW-ResNet，但是通道和维度的多尺度变换将导致由于网络中非尖峰卷积操作而导致的能量消耗增加的问题在目标检测任务中尤为突出。）

贡献

- 提出了一种新的直接训练的脉冲神经网络用于目标检测——EMS-YOLO，实时只需要4个时间步和推理。
- 设计了一个模块Energy-efficient Membrane-Shortcut ResNet，ems-resnet，它可以在网络中实现完整的脉冲传递，从而降低功耗。给出理论上分析，可以被深度训练，避免梯度消失或爆炸。
- 拥有更好的实验效果。

预备知识

1. 脉冲神经网络



   相较于ANN，SNN区别主要在于输入数据以及激活函数的不同，SNN的输入数据为t时间步长的脉冲序列，对比ANN激活层使用的是LIF神经元（输出只有0和1的阶跃函数）。

2. LIF模型

$$V_i^{t+1,n+1} = \tau V_i^{t,n+1}(1 - X_i^{t,n+1}) + \sum_j W_{ij}^n X_j^{t+1,n} \quad (1)$$

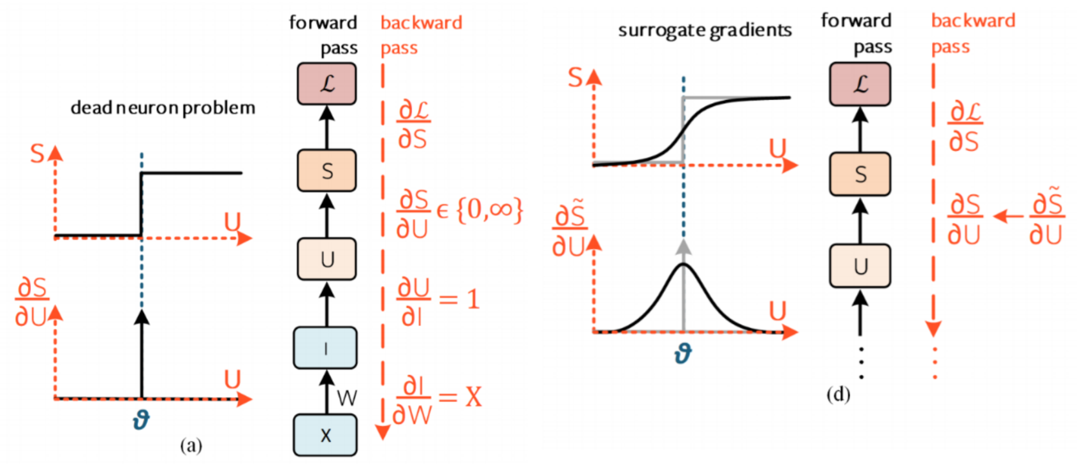
$$X_i^{t+1,n+1} = H(V_i^{t+1,n+1} - V_{th}) \quad (2)$$

where the $V_i^{t,n+1}$ is the membrane potential of the i-th neuron in the $n+1$ layer at the timestep t, τ is a decay factor for leakage. The input to a synapse is the sum of j spikes $X_j^{t+1,n}$ with synaptic weights W_{ij}^n from the previous layer n. $H(\cdot)$ denotes the Heaviside step function which satisfies $H(x) = 1$ for $x \geq 0$, otherwise $H(x) = 0$. As shown in the Figure 2, a firing activity is controlled by the threshold V_{th}, and the $V^{t+1,n+1}$ will be reset to V_{rest} once the neuron emits a spike at time step $t+1$.

- $\mathcal{V}_i^{t,n+1}$--时间步长t中第n+1层第i个神经元的膜电位。
- τ ——衰减因子
- $(1-\mathbf{X}_i^{t,n+1})$--代表着是否要重置电位
- $\sum \mathbf{W}_{ij}^n(\mathbf{X}_j^{t+1,n})$--前一层脉冲总和
- $\mathbf{W}_{ij}^n$--权重
- $\mathbf{H}(\cdot)$--核函数，如果$x>1$,$\mathbf{H}(x)=1$,否则$\mathbf{H}(x)=0$.
- $\mathbf{V}_{th}$--膜电位的阈值

3. 代理梯度

SNN直接训练为什么要用代理梯度?



神经网络的训练需要梯度回传，LIF神经元是阶跃函数，在计算梯度时会出现（a）**死神经元问题**: $\frac{\partial S}{\partial U} \in \{0,\infty\}$的解析解导致梯度无法 进行学习。

（d）在反向传播过程中将脉冲生成函数近似为连续函数，解决梯度计算问题（前向传播过程依然是脉冲过程，只有梯度计算的时候才 会用到代理梯度）。

代理函数

$$\frac{\partial X_i^{t,n}}{\partial V_i^{t,n}} = \frac{1}{a} sign(|V_i^{t,n} - V_{th}| \leq \frac{a}{2}) \quad (3)$$

> Spatio-Temporal Backpropagation for Training High-Performance Spiking Neural Networks

使用上述论文中提出的近似函数作为代理函数。

4. TDBN

```
class tdBatchNorm(nn.BatchNorm2d):
'''
    Args:
        num_features (int): same with nn.BatchNorm2d
        eps (float): same with nn.BatchNorm2d
        momentum (float): same with nn.BatchNorm2d
        alpha (float): an addtional parameter which may change in resblock.
        affine (bool): same with nn.BatchNorm2d
        track_running_stats (bool): same with nn.BatchNorm2d
'''
        def __init__(self, num_features, eps=1e-05, momentum=0.1, alpha=1,
affine=True, track_running_stats=True):
        super(tdBatchNorm, self).__init__(
            num_features, eps, momentum, affine, track_running_stats)
            self.alpha = alpha

        def forward(self, input):
            exponential_average_factor = 0.0

            if self.training and self.track_running_stats:
                if self.num_batches_tracked is not None:
                    self.num_batches_tracked += 1
                    if self.momentum is None:  # use cumulative moving
average
                        exponential_average_factor = 1.0 /
float(self.num_batches_tracked)
                    else:  # use exponential moving average
                        exponential_average_factor = self.momentum

            # calculate running estimates
            if self.training:
                mean = input.mean([0, 2, 3, 4])
                # use biased var in train
                var = input.var([0, 2, 3, 4], unbiased=False)
                n = input.numel() / input.size(1)
                with torch.no_grad():
                    self.running_mean = exponential_average_factor * mean\
                        + (1 - exponential_average_factor) *
self.running_mean
                    # update running_var with unbiased var
                    self.running_var = exponential_average_factor * var * n /
(n - 1)\
                        + (1 - exponential_average_factor) * self.running_var
            else:
                mean = self.running_mean
                var = self.running_var
            input = self.alpha * Vth * (input - mean[None, :, None, None, 
None]) / (torch.sqrt(var[None, :, None, None, None] + self.eps))
            if self.affine:
                input = input * self.weight[None, :, None, None, None] + 
self.bias[None, :, None, None, None]

        return input
```

$$V_i^{t+1,n+1} = \tau V_i^{t,n+1}(1 - X_i^{t,n+1}) + \mathrm{TDBN}(I_i^{t+1}) \quad (4)$$

$$\mathrm{TDBN}(I_i^{t+1}) = \lambda_i \frac{\alpha V_{th}(I_i^{t+1} - \mu_{ci})}{\sqrt{\sigma_{ci}^2 + \epsilon}} + \beta_i \quad (5)$$

where μ_{ci}, σ_{ci}^2 are the mean and variation values for every channel using a mini-batch of sequential inputs $\{I_i^{t+1} = \Sigma_j W_{ij}^n X_j^{t+1,n} | t = 0, ..., T-1\}$, ϵ is a tiny constant to avoid dividing by zero, λ_i, β_i are two trainable parameters, and α is a threshold-dependent hyper-parameter.

- $\alpha, \lambda_i, \beta_i$是超参数
- $\mathcal{V}_{th}$--阈值

使用TDBN去代替网络的BN层，相较于原来的BN层计算公式多了超参数和阈值$\mathcal{V}$

方法

- 流程图

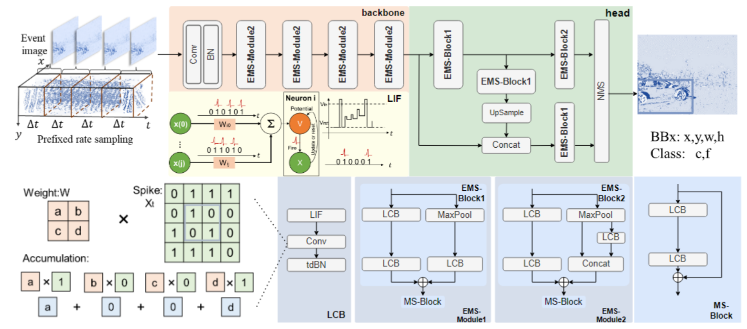
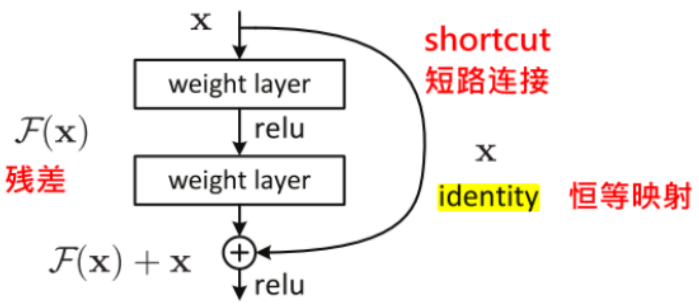


Figure 2. **The proposed directly-trained SNN for object detection (EMS-YOLO).** EMS-YOLO mainly consists of backbone and head, which are mainly composed of EMS-Blocks. EMS-Module1 and EMS-Module2 are EMS-Block1 and MS-Block, EMS-Block2 and MS-Block connections respectively. EMS-Block2 is used when the number of output channels increases, otherwise, EMS-Block1 is used.

- EMS-ResNet 残差块
  - ResNet残差结构



  - EMS-ResNet 残差结构

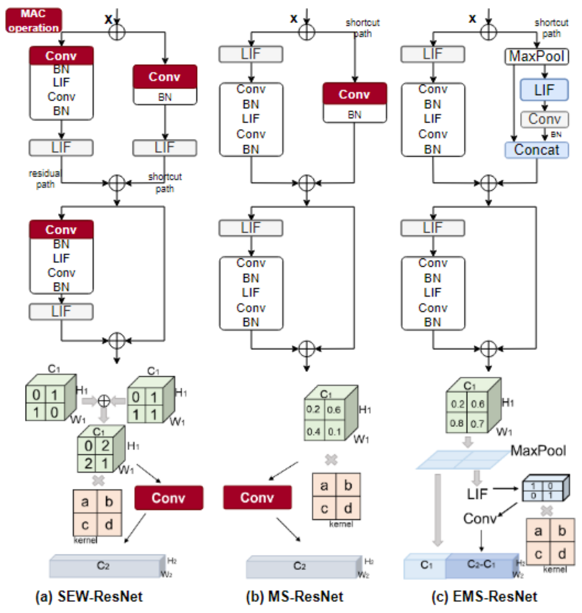


Figure 1. **The Sew-ResNet, the MS-ResNet and proposed EMS-ResNet.** (a) The sum of spikes in SEW-ResNet causes non-spike convolution operations. (b) MS-ResNet introduces non-spike convolution on shortcut paths where the dimensionality and number of channels changes. (c) The full-spike EMS-ResNet.

（a）两个通道会引入额外计算；（b）没有考虑到将脉LIF引入到快捷通道；（c）follow（b）的工作，主要改进就是在快捷通道引入了LIF神经元。