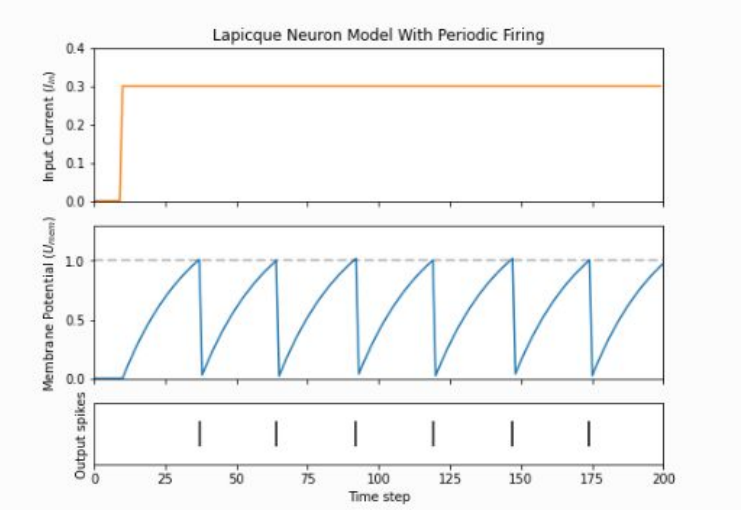
Training Spiking Neural Networks based on DL

Iaroslav Gusev & Yaroslav Abramov

Leaky-Integrate-and-Fire model

$$\tau \frac{dU(t)}{dt} = -U(t) + I_{\text{in}}(t)R$$

$$U(t) = I_{\text{in}}R + [U_0 - I_{\text{in}}R]e^{-\frac{t}{\tau}}$$



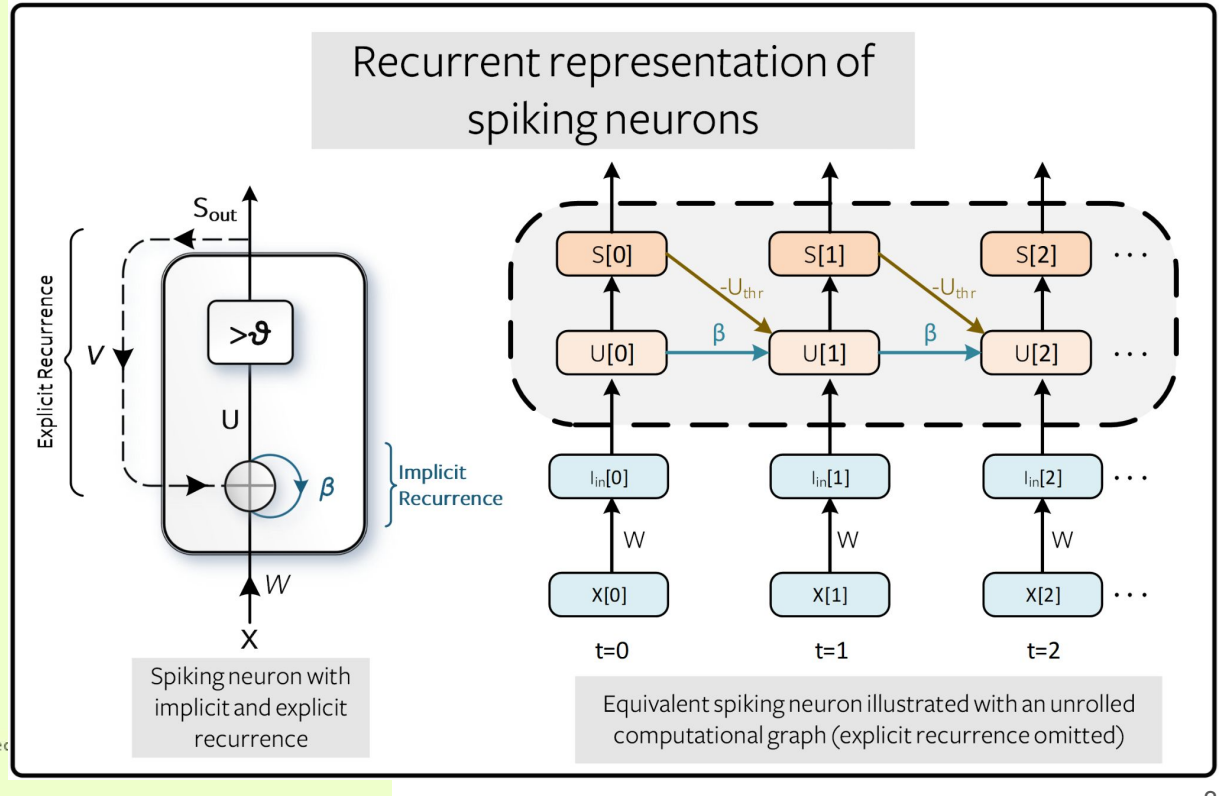
$$U[t] = \underbrace{\beta U[t-1]}_{\text{decay}} + \underbrace{WX[t]}_{\text{input}} - \underbrace{S_{\text{out}}[t-1]\theta}_{\text{reset}}$$

$$S_{\text{out}}[t] = \begin{cases} 1, & \text{if } U[t] > \theta \\ 0, & \text{otherwise} \end{cases}$$

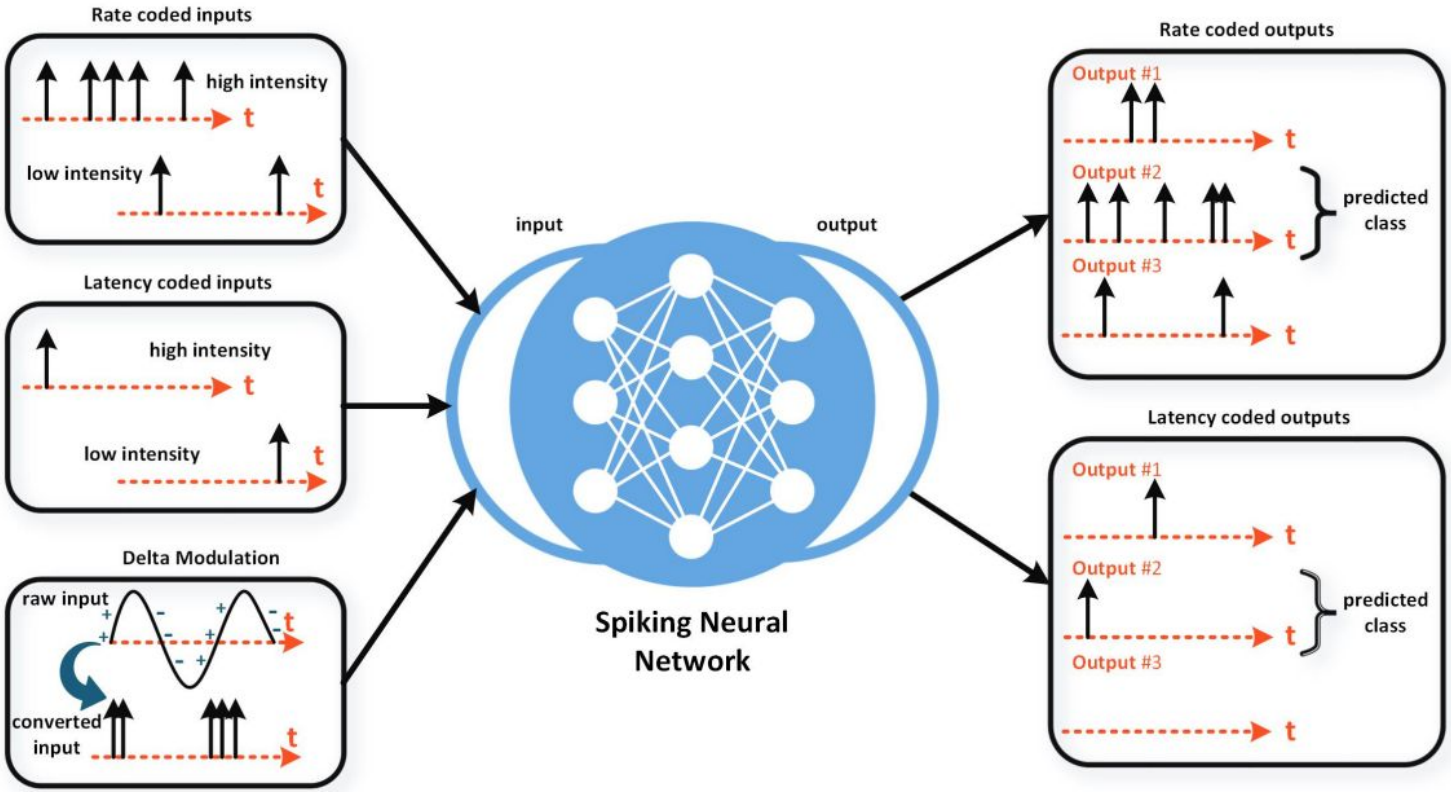
LIF neurons are the most commonly used in the deep learning context. They sit in the sweet spot between biological plausibility and practicality. It is relatively straightforward to adopt techniques used in training recurrent neural networks (RNNs) to SNNs.

Using LIF neurons in artificial neural network

```
# Define Network
class Net(nn.Module):
    def __init__(self):
        super().__init__()

        # Initialize layers
        self.fc1 = nn.Linear(num_inputs, num_hidden)
        self.lif1 = snn.Leaky(beta=beta)
        self.fc2 = nn.Linear(num_hidden, num_outputs)
        self.lif2 = snn.Leaky(beta=beta)

    def forward(self, x):

        # Initialize hidden states at t=0
        mem1 = self.lif1.init_leaky()
        mem2 = self.lif2.init_leaky()

        # Record the final layer
        spk2_rec = []
        mem2_rec = []

        for step in range(num_steps):
            cur1 = self.fc1(x)
            spk1, mem1 = self.lif1(cur1, mem1)
            cur2 = self.fc2(spk1)
            spk2, mem2 = self.lif2(cur2, mem2)
            spk2_rec.append(spk2)
            mem2_rec.append(mem2)

        return torch.stack(spk2_rec, dim=0), torch.stack(mem2_rec

# Load the network onto CUDA if available
net = Net().to(device)
```



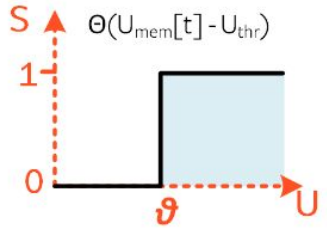
Different input/output encodings



Non-differentiability of spikes

$$S[t] = \Theta(U[t] - U_{\text{thr}})$$

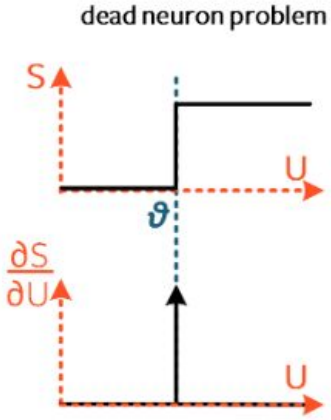
Mathematical Description of Spikes



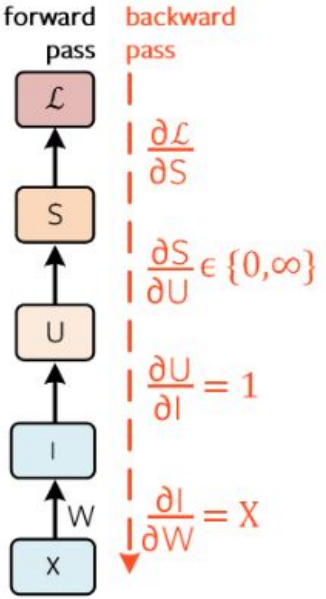
If the membrane potential is greater than the threhsold $U_{thr} = \vartheta$, then $S_{out} = 1$.

Otherwise, $S_{out} = 0$.

Spike Non-differentiability



The derivative of the Heaviside function is 0 almost everywhere, causing $\partial \mathcal{L} / \partial W$ to also become 0



$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial S} \underbrace{\frac{\partial S}{\partial U}}_{\{0,\infty\}} \frac{\partial U}{\partial I} \frac{\partial I}{\partial W}$$

Solution (surrogate gradient):

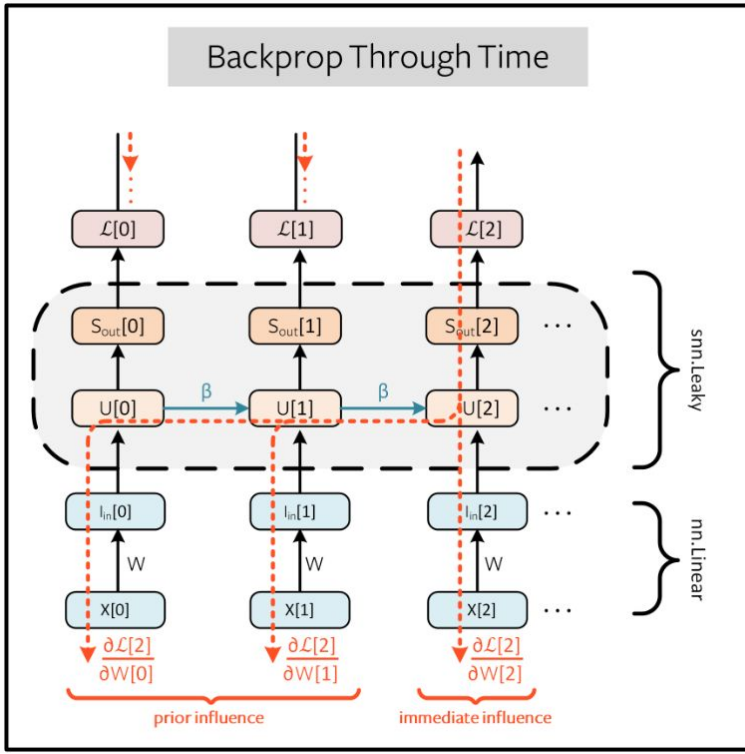
$$\frac{\partial \tilde{S}}{\partial U} \leftarrow \frac{1}{\pi} \frac{1}{(1 + [U\pi]^2)}$$

Backpropagation through time

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_t \frac{\partial \mathcal{L}[t]}{\partial W} = \sum_t \sum_{s \leq t} \frac{\partial \mathcal{L}[t]}{\partial W[s]} \frac{\partial W[s]}{\partial W}$$

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_t \sum_{s \leq t} \frac{\partial \mathcal{L}[t]}{\partial W[s]}$$

$$\frac{\partial \mathcal{L}[t]}{\partial W[t-1]} = \frac{\partial \mathcal{L}[t]}{\partial S[t]} \underbrace{\frac{\partial \tilde{S}[t]}{\partial U[t]}}_{Eq.\ (5)} \underbrace{\frac{\partial U[t]}{\partial U[t-1]}}_{\beta} \underbrace{\frac{\partial U[t-1]}{\partial I[t-1]}}_{1} \underbrace{\frac{\partial I[t-1]}{\partial W[t-1]}}_{X[t-1]}$$



Thankfully, PyTorch does this automatically!

Loss function for MNIST classification problem. (Rate encoding)

- Usually, the spikes of the output neurons considered as “observables” and so the Loss is defined as function of spike train(s).
- However, for the reason of simplicity the authors of the tutorial from snntorch library used the loss function defined as a function of membrane potential, that is usually considered as just “proxy” or auxiliary variable.

$$p_i[t] = \frac{e^{U_i[t]}}{\sum_{i=0}^{C} e^{U_i[t]}} \tag{8}$$

The cross-entropy between p_i and the target $y_i \in \{0, 1\}^C$, which is a one-hot target vector, is obtained using:

$$\mathcal{L}_{CE}[t] = -\sum_{i=0}^{C} y_i \log(p_i[t]) \tag{9}$$

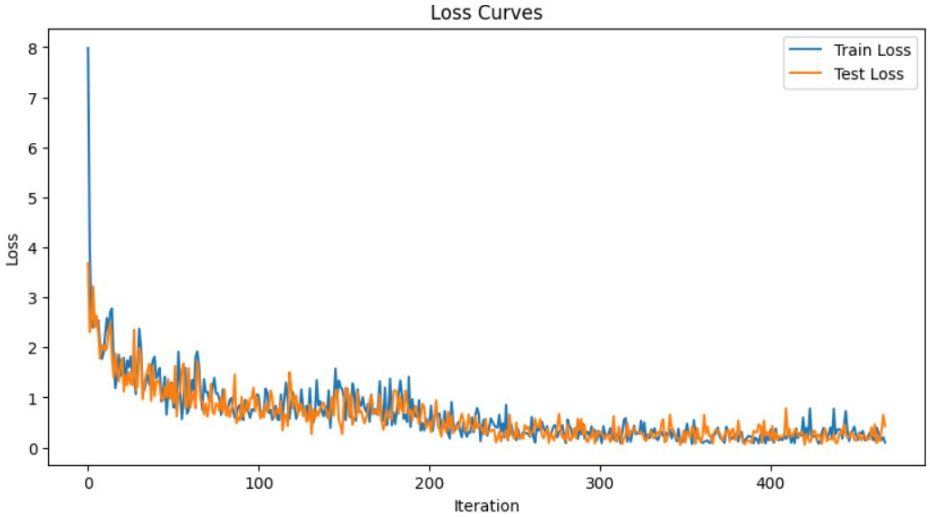
The practical effect is that the membrane potential of the correct class is encouraged to increase while those of incorrect classes are reduced. In effect, this means the correct class is encouraged to fire at all time steps, while incorrect classes are suppressed at all steps. This may not be the most efficient implementation of an SNN, but it is among the simplest.

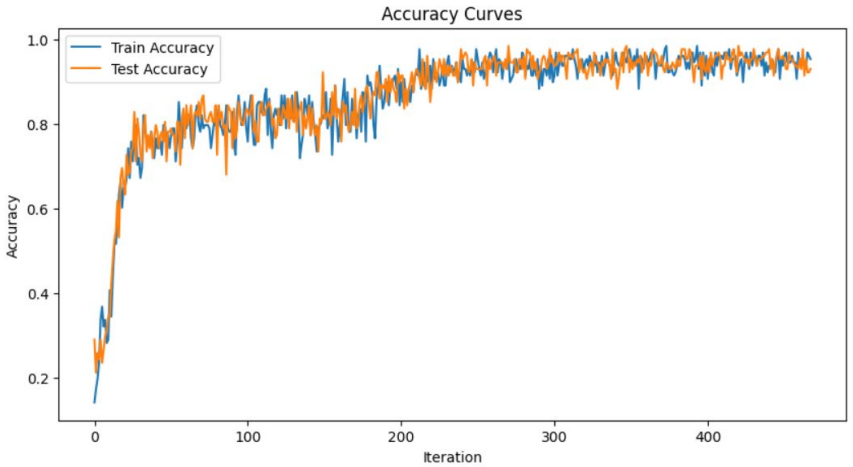
$$\mathcal{L}_{CE} = \sum_{t} \mathcal{L}_{CE}[t]$$

Temporal encoding in MNIST classification (TTFS)

- From spike trains of output neurons we can obtain the timing of first spike for each neuron.
- We wish the neuron that correspond to correct class to fire first, so the smaller value of timing means larger probability, or “logit”.
- So in order to compute “logits” we need to apply a monotonically decreasing function to our spike timings.
- In snntorch two options implemented: y=-x, or y=1/x, we used the first option.
- Corresponding function f(s) is not differentiable, so some approximation is used. Particularly, in snntorch it is df/ds = -1 if s =1, df/ds = 0 otherwise.

Results. MNIST 10-digit classification problem.





- The authors of the tutorial got 93.87% test accuracy (with their rate encoding).
- We got 95.35% test accuracy (with TTFS encoding).

Other tasks. CIFAR-10

You can try some CNNs and just replace activation functions with iterable LIF neurons. But this just give you accuracy around 70% (if you are lucky).

There was an approach from "Fast and Efficient Deep Sparse Multi-Strength Spiking Neural Networks with Dynamic Pruning" by Ruizhi Chen, Hong Ma, Shaolin Xie, Peng Guo, Pin Li, Donglin Wang, which claims 94% accuracy on CIFAR-10, and there spikes were replaced by multi-strength spikes defines as:

$$\theta_{i,t}^{l} := floor(\frac{max(v_i^l(t-1) + z_i^l(t), 0)}{\tau}) \qquad (2)$$

The Multi-Strength LIF neuron integrates input $z_i^l(t)$ until the membrane potential $v_i^l(t)$ exceeds the threshold τ. By subtracting the $\tau\theta_{i,t-1}^l$ to generate the multi-strength spike, the membrane potential of the ith neuron in layer l at time t can be formulated as follows:

$$v_i^l(t) = v_i^l(t-1) + z_i^l(t) - \tau\theta_{i,t-1}^l \qquad (3)$$

where $\theta_{i,t-1}^l$ is a nonnegative integer.