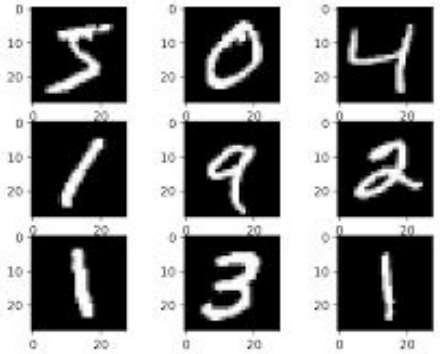
Implementing SNNs: Diving Deep

Shreshth Mehrotra

1)Spike Encoding

The objective is to build an SNN model for **digit recognition** using the MNIST dataset.

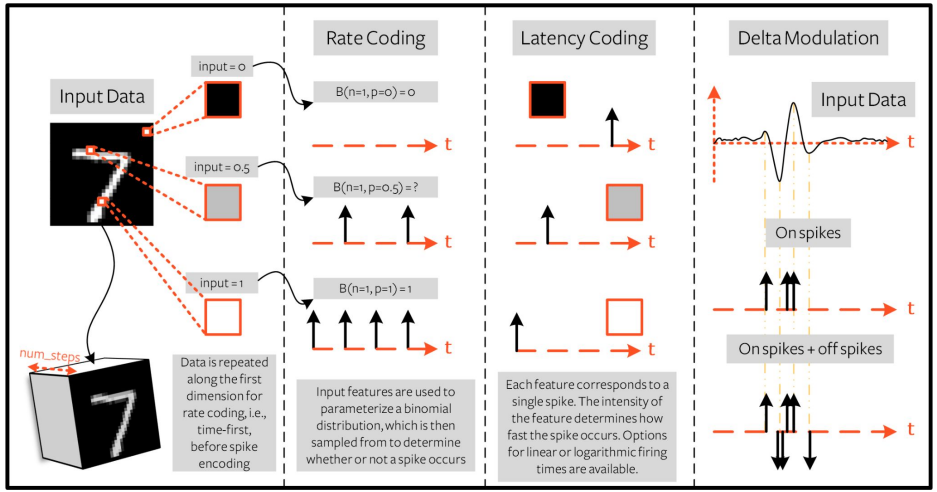
SNNs are made to exploit time varying data. However, MNIST is not a time-varying dataset. It's just a collection of static images.



The first step in building our SNN model, will therefore be to represent our input, (images in this case), as **time varying data.** As SNNs use spikes to communicate information between neurons, we must encode the input data into spikes, and this step is called **spike encoding.**

Some possible ways for spike encoding are:

- **Rate coding:** The spike **frequency** is used to encode input features.
- **Latency coding:** The spike **Timing** is used to encode input features.
- **Delta modulation: Temporal change** of input features is used to generate spikes.



1.1)Rate Coding

Normalize the input image so that the pixel intensities are in [0,1], and these pixel intensities are our input features.

The input values can be treated as the **probability that a spike occurs at any time step**, returning a rate coded value Rij, (which can either be 0 or 1).

This can be modelled as a **bernoulli trial**:

$$R_{ij} = B(n,p) = B(1, X_{ij})$$

$$P(R_{ij} = 1) = X_{ij}$$

1.1.1)Implementation

```
# Temporal Dynamics

num_steps = 10

# create vector filled with 0.5

raw_vector = torch.ones(num_steps)*0.5

# pass each sample through a Bernoulli trial

 rate_coded_vector = torch.bernoulli(raw_vector)
```

```
>>> print(f"Converted vector: {rate_coded_vector}")
Converted vector: tensor([1., 1., 1., 0., 0., 1., 1., 0., 1., 0.])
```

With num steps initialized to 100 :

```
>>> print(f"The output is spiking {rate_coded_vector.sum()*100/len(rate_coded_vector):.2f}% of the time
The output is spiking 48.00% of the time.
```

As num_steps $\rightarrow \infty$ the proportion of spikes approaches the original raw value.

```
from snntorch import spikegen

# Iterate through minibatches

data = iter(train_loader)

data_it, targets_it = next(data)

# Spiking Data

spike_data = spikegen.rate(data_it, num_steps=num_steps)
```

The structure of the input data is `[num steps x batch_size x input dimensions]`, so for eg for a batch size of **128** and num_steps=**100**,the shape will be `[100, 128, 1, 28, 28]`.

Visualization

Reconstruction of the input image by averaging out the spikes over time

1.1.2)Some comments on Rate Coding

The idea of rate coding is actually quite controversial. Although we are fairly confident rate coding takes place at our sensory periphery, we are not convinced that the cortex globally encodes information as spike rates, for the following reasons:

- **Power Consumption:** Nature optimised for efficiency. Multiple spikes are needed to achieve any sort of task, and each spike consumes power. In fact, Olshausen and Field's work in "What is the other 85% of V1 doing?" demonstrates that rate-coding can only explain, at most, the activity of 15% of neurons in the primary visual cortex (V1). It is unlikely to be the only mechanism within the brain, which is both resource-constrained and highly efficient.
- **Reaction Response Times:** We know that the reaction time of a human is roughly around 250ms. If the average firing rate of a neuron in the human brain is on the order of 10Hz, then we can only process about 2 spikes within our reaction timescale.

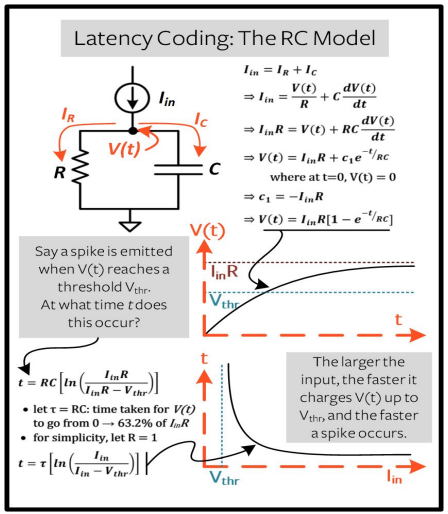
Even if our brain doesn't process data as a rate, we are fairly sure that our biological sensors do. It can be concluded that Rate coding is almost certainly working in conjunction with other encoding schemes in the brain, and we discuss these other encoding mechanisms in the subsequent sections.

1.2)Latency Coding

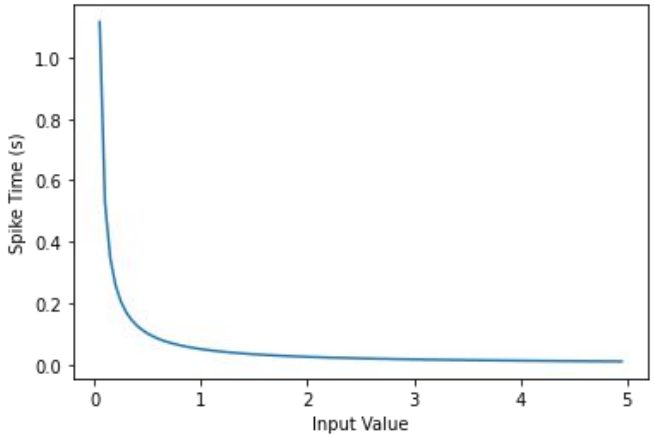
Temporal codes capture information about the precise firing time of neurons; a **single spike carries much more meaning** than in rate codes which rely on firing frequency. While this opens up more susceptibility to noise, it can also decrease the power consumed by the hardware running SNN algorithms by orders of magnitude.

`spikegen.latency` is a function that allows each input to fire at most **once** during the full time sweep. Features closer to `1` will fire earlier and features closer to `0` will fire later. I.e., in our MNIST case, bright pixels will fire earlier and dark pixels will fire later.

1.3)Mathematical Modelling

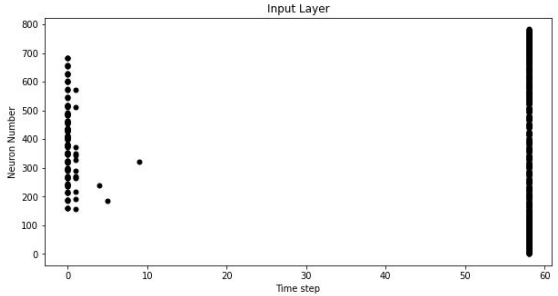


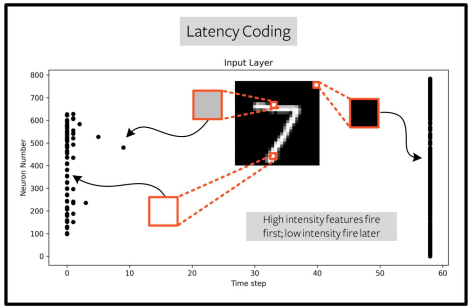
```
def convert_to_time(data, tau=5, threshold=0.01):
  spike_time = tau * torch.log(data / (data - threshold))
   return spike_time
raw_input = torch.arange(0, 5, 0.05) # tensor from 0 to 5
 spike_times = convert_to_time(raw_input)
```



The vector `spike_times` contains the **time** at which spikes are triggered, rather than a sparse tensor that contains the spikes themselves (1's and 0's). When running an SNN simulation, we need the 1/0 representation to obtain all of the advantages of using spikes. This whole process can be automated using `spikegen.latency`, where we pass a minibatch from the MNIST dataset in data_it:

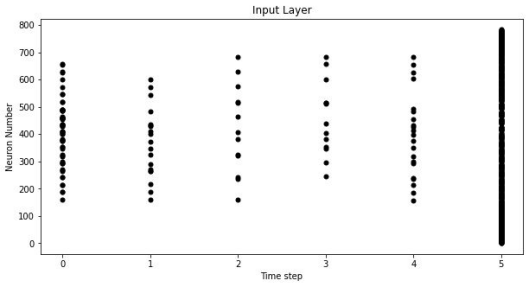
```
spike_data = spikegen.latency(data_it, num_steps=100, tau=5, threshold=0.01)
```

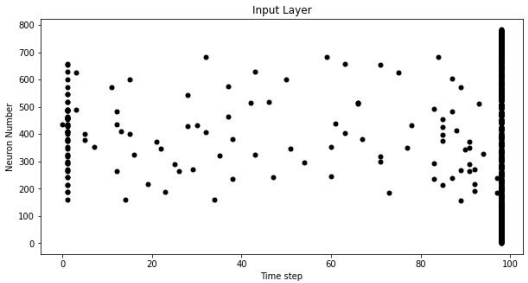


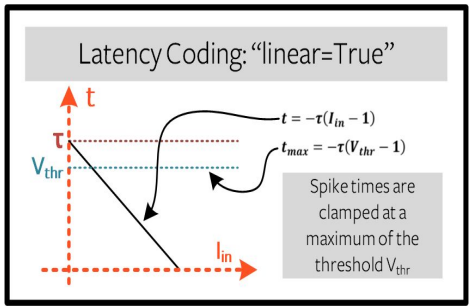


Due to the logarithmic nature, and due to the lack of input features other than 0 and 1, the data has been clustered into groups at the extreme ends. We can spread out the firing times evenly, by using a linear decay instead of logarithmic, using the Linear=true parameter: `spike_data = spikegen.latency(data_it, num_steps=100, tau=5, threshold=0.01, linear=True).`

Also worth noting is the fact that all firing occurs within the first ~5 time steps, whereas the simulation range is 100 time steps. This indicates there are many redundant time steps doing nothing. This can be solved by either increasing `tau` to slow down the time constant, or setting the optional argument `normalize=True` to span the full range of `num_steps`: `spike_data = spikegen.latency(data_it, num_steps=100, tau=5, threshold=0.01, normalize=True, linear=True)`

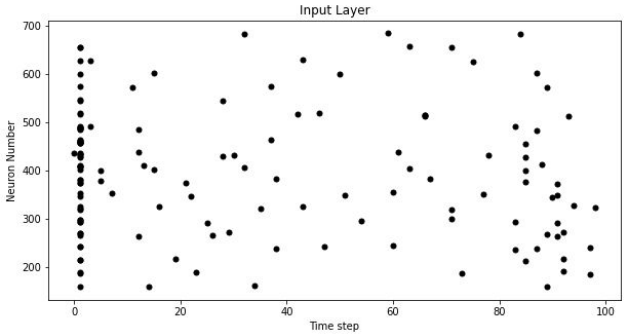






One major advantage of latency coding over rate coding is **sparsity**. If neurons are constrained to firing a maximum of once over the time course of interest, then this **promotes low-power operation**.

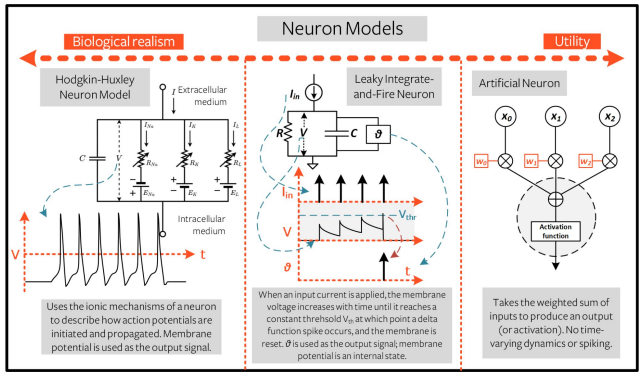
A majority of the spikes occur at the final time step, where the input features fall below the threshold. In a sense, the dark background of the MNIST sample holds **no useful information**. We can remove these redundant features by setting `clip=True`: `spike_data = spikegen.latency(data_it, num_steps=100, tau=5, threshold=0.01,clip=True, normalize=True, linear=True)`



With the input features encoded into spike trains, we now look at the next step: where we look into how the neurons are modelled. As neurons are the basic building block of all neural networks, it is important to understand how neurons react in response to these incoming spike trains.

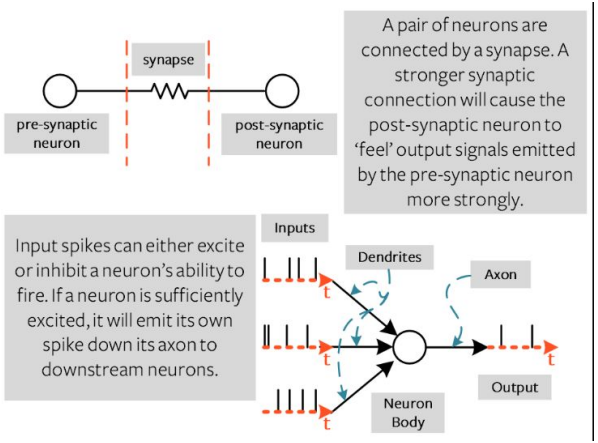
2.)Neuron Modelling

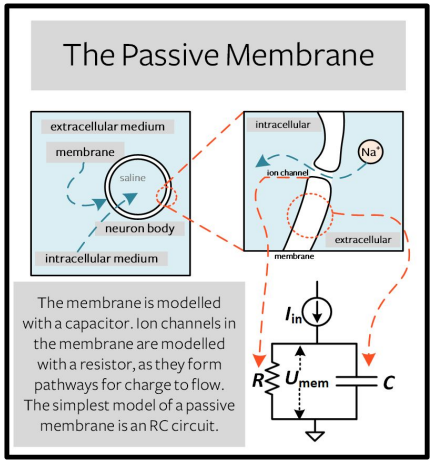
- **Hodgkin-Huxley Neuron Models:**Biophysical model.It can reproduce electrophysiological results with a high degree of accuracy, their complexity makes them difficult to use at present.
- **Artificial Neuron Model:** Extremely easy to use, but not biologically accurate.
- **Leaky Integrate-and-Fire Neuron Models:** Somewhere in the middle of the divide lies the leaky integrate-and-fire (LIF) neuron model. It takes the sum of weighted inputs, much like the artificial neuron. But rather than passing it directly to an activation function, it will **integrate the input over time with a leakage**, much like an RC circuit. If the integrated value exceeds a threshold, then the LIF neuron will emit a voltage spike.



In this report, we use the **Lapicque's RC model** as the model for our LIF neuron.

2.1)LIF Neurons





Like all cells, a neuron is surrounded by a thin membrane. This membrane is a lipid bilayer that insulates the conductive saline solution within the neuron from the extracellular medium. Electrically, the two conductive solutions separated by an insulator act as a **capacitor**.

Another function of this membrane is to control what goes in and out of this cell (e.g., ions such as Na). The membrane is usually impermeable to ions which blocks them from entering and exiting the neuron body. But there are specific channels in the membrane that are triggered to open by injecting current into the neuron. This charge movement is electrically modelled by a **resistor.**

We can now work on this RC circuit and solve for the membrane potential $U_{mem}(t)$:

$$I_{in}(t) = I_R + I_C$$

$$= \frac{U_{mem}(t)}{R} + \frac{dQ_{mem}}{dt}$$

$$= \frac{U_{mem}(t)}{R} + C\frac{dU_{mem}}{dt}$$

Rearranging, and by letting RC=Tau (time constant), we get:

$$\frac{dU_{mem}}{dt} + \frac{U_{mem}(t)}{\tau} = \frac{I_{in}(t)}{C}$$

Which is a linear ODE with integrating factor $e^{t/\tau}$ .We multiply both sides by this factor and simplify:

$$\frac{d(e^{t/\tau}U_{mem}(t))}{dt} = e^{t/\tau}\frac{I_{in}(t)}{C}$$

$$U_{mem}(t) = e^{-t/\tau}\int e^{t/\tau}\frac{I_{in}(t)}{C}dt + Ke^{-t/\tau}$$

The solution is a sum of two terms: The first term **integrates** the input current over time, and the second term exponentially decays with time, which represents **Leakage.** (In the absence of input current, the voltage decays with time, or ‘leaks’ with time)

We have obtained the closed form solution for $U_{mem}(t)$, however it is not very clear how one would implement this in practise. Instead, we can use the **forward Euler method** to solve the previous linear ordinary differential equation (ODE). This approach gives us a discrete, recurrent representation of the LIF neuron:

$$\frac{dU_{mem}}{dt} + \frac{U_{mem}(t)}{\tau} = \frac{I_{in}(t)}{C}$$

$$\approx \frac{U_{mem}(t+\Delta t) - U_{mem}}{\Delta t} + \frac{U_{mem}(t)}{\tau} = \frac{I_{in}(t)}{C}$$

$$U_{mem}(t+\Delta t) = U_{mem}(t) + \frac{\Delta t}{\tau}\big(-U_{mem}(t) + RI_{in}(t)\big)$$

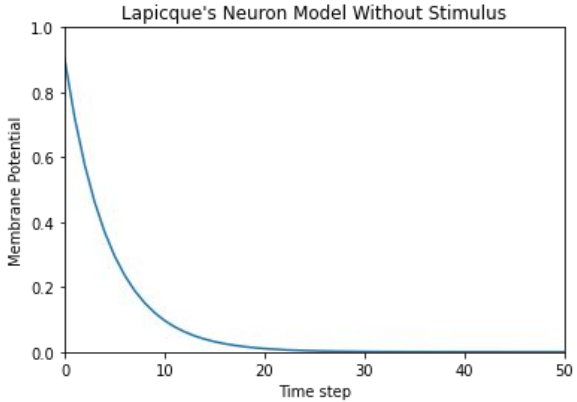
```
def leaky_integrate_neuron(U,
time_step=1e-3, I=0, R=5e7, C=1e-10):

  tau = R*C

  U = U + (time_step/tau)*(-U + I*R)

    return U
```

The LIF model in snnTorch uses this to calculate the membrane potential for the Lapicque LIF Model. We can instantiate it using:

```
time_step = 1e-3

R = 5

C = 1e-3

# leaky integrate and fire neuron, tau=5e-3

  lif1 = snn.Lapicque(R=R, C=C, time_step=time_step)
```

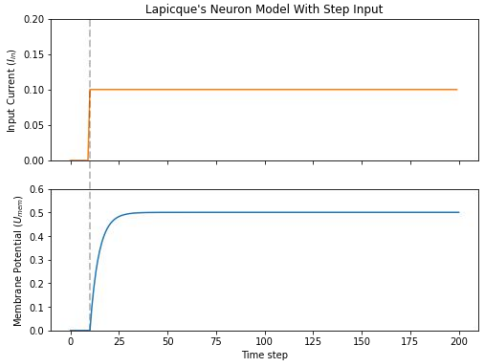
2.1)Using the LIF Neuron:

```
# Initialize membrane, input, and output
mem = torch.ones(1) * 0.9  # U=0.9 at t=0
cur_in = torch.zeros(num_steps)  # I=0 for all t
spk_out = torch.zeros(1)  # initialize output spikes
for step in range(num_steps):

    spk_out, mem = lif1(cur_in[step], mem)
```



For no stimulus,i.e $I_{in}(t) = 0$ we get an exponentially decaying membrane potential: $U_{mem}(t) = U_{mem}(0)e^{-t/\tau}$

We can analyze the potential for a **step input**, say $I_{in}(t) = I_0 u(t - t_0)$ as well:



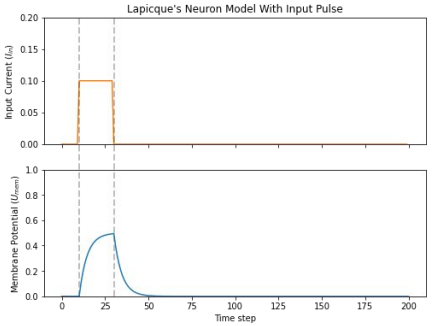
For t<= t_0

$$U_{mem}(t) = e^{-t/\tau} \int e^{t/\tau} \frac{I_{in}(t)}{C} dt + Ke^{-t/\tau} = Ke^{-t/\tau} = U_{mem}(0)e^{t/\tau} = 0$$

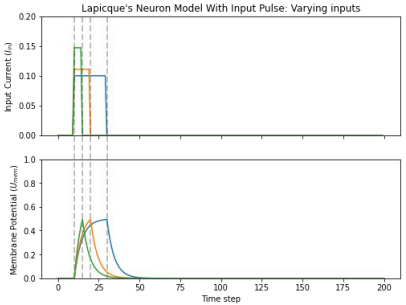
For t>t_0

$$U_{mem}(t) = e^{-t/\tau} \int e^{t/\tau} \frac{I_0}{C} dt + Ke^{-t/\tau} = I_0 R + Ke^{-t/\tau} = I_0 R(1 - e^{(t_0 - t)/\tau})$$

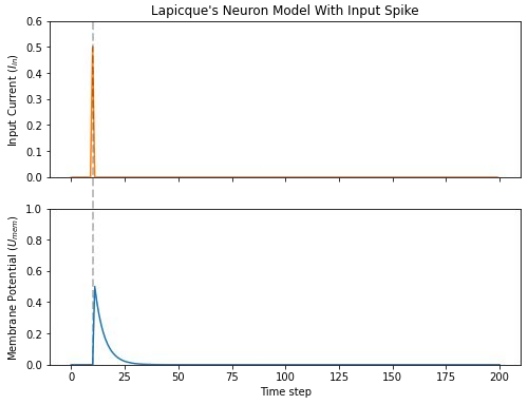
A pulse input is just the difference of two step inputs, and the voltage profile obtained is as shown:



Next, we analyze the voltage profile for various step inputs, with **decreasing width** and **increasing amplitude,**in such a way that the area remains constant.



In the limit of the input current pulse width becoming infinitesimally small, the membrane potential will jump straight up in virtually zero rise time. We can use this as our representation of a spike. Mathematically, this is the **Dirac Delta Function.**



$$I_{in}(t) = Q\delta(t - t_0)$$

$$\int_0^t I_{in}(t)dt = Q\int_0^t \delta(t - t_0)dt = Q\int_{t_0-\epsilon}^{t_0+\epsilon} \delta(t - t_0)dt = Q$$

From our derivation , we have: $\frac{d(e^{t/\tau}U_{mem}(t))}{dt} = e^{t/\tau}\frac{I_{in}(t)}{C} = \frac{Q}{C}e^{t/\tau}\delta(t - t_0)$ .We now integrate both sides from 0 to t. (assuming zero initial conditions:

$$e^{t/\tau}U_{mem}(t) = \frac{Q}{C}\int_{t_0-\epsilon}^{t_0+\epsilon} e^{t/\tau}\delta(t - t_0) = \frac{Q}{C}e^{t_0/\tau} = K$$

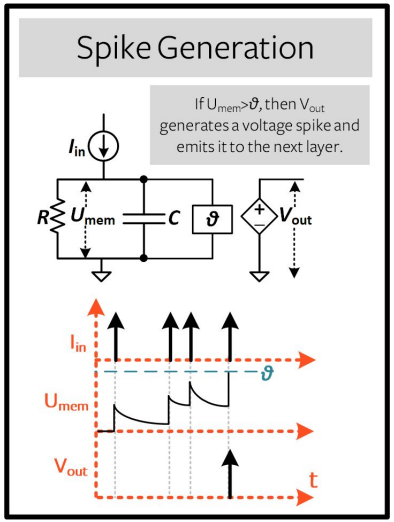
$$U_{mem}(t) = Ke^{-t/\tau}$$

Assuming t>t0. For t<t0, the delta function is zero for the entire region, and so the integral is zero, which implies $U_{mem}(t)$=0.

2.2)Firing

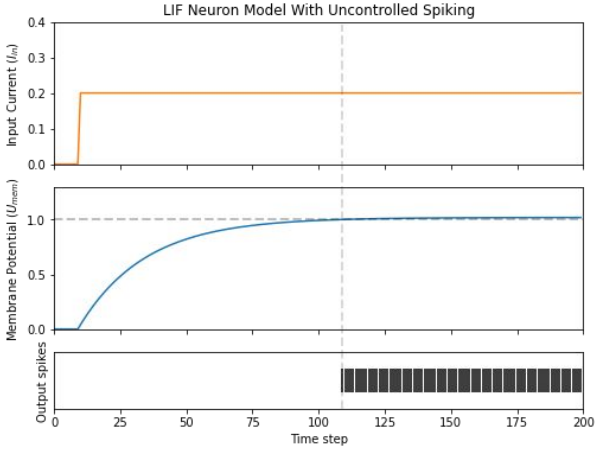
With the leaky, and integrate aspects covered, all that is left is the ‘firing’ aspect: when, and how does a neuron fire, i.e give output spikes?

If the membrane potential exceeds this threshold, then a voltage spike will be generated, external to the passive membrane model.



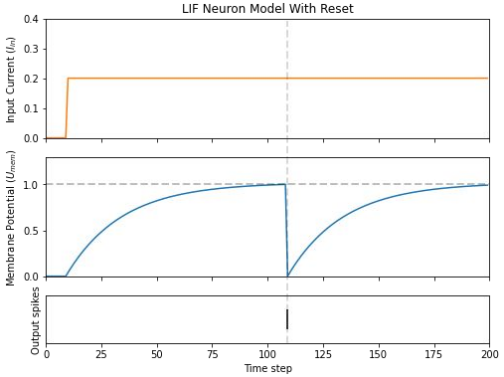

```
def leaky_integrate_and_fire(mem, cur=0,
threshold=1, time_step=1e-3, R=5.1, C=5e-3):
  tau_mem = R*C
  spk = (mem > threshold) # if membrane exceeds
threshold, spk=1, else, 0
  mem = mem + (time_step/tau_mem)*(-mem + cur*R)
    return mem, spk
```

2.3)Reset



The spiking goes out of control, this is because we forgot to add a reset mechanism. In reality, each time a neuron fires, the membrane potential hyperpolarizes back to its resting potential.

```
# LIF w/Reset mechanism
def leaky_integrate_and_fire(mem, cur=0,
threshold=1, time_step=1e-3, R=5.1, C=5e-3):
  tau_mem = R*C
  spk = (mem > threshold)
  mem = mem + (time_step/tau_mem)*(-mem +
cur*R) - spk*threshold  # every time spk=1,
subtract the threhsold
    return mem, spk
```

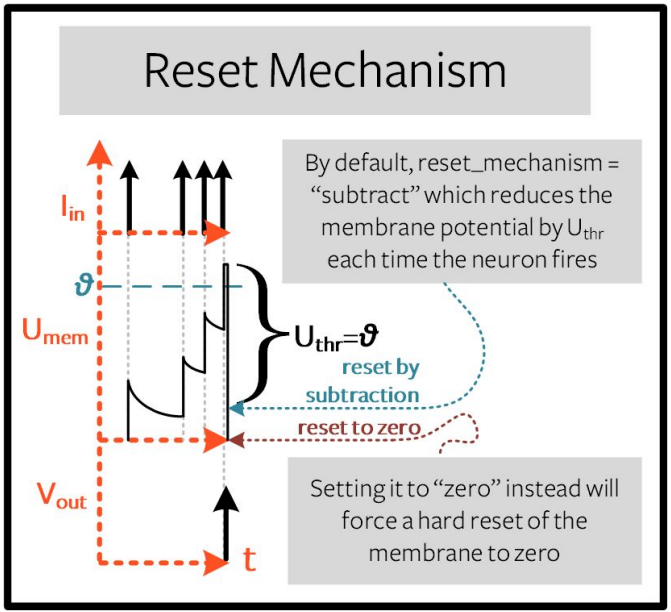



```
lif2 = snn.Lapicque(R=5.1, C=5e-3,
time_step=1e-3,
threshold=1,reset_mechanism="subtrac
t")
```

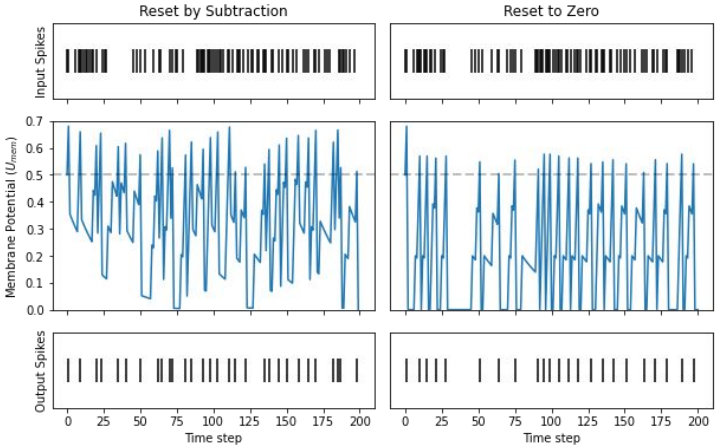
2.4)Choice of reset mechanism

Reset by subtraction: The threshold is subtracted from the membrane potential whenever a spike is generated.

Reset to zero: force the membrane potential to zero each time a spike is generated. (Hard reset to zero).



```
lif3 = snn.Lapicque(R=5.1, C=5e-3, time_step=1e-3, threshold=0.5,reset_mechanism="subtract")
lif4 = snn.Lapicque(R=5.1, C=5e-3, time_step=1e-3, threshold=0.5, reset_mechanism="zero")
```



- Applying `"subtract"` (the default value in `reset_mechanism`) is less lossy, because it does not ignore how much the membrane exceeds the threshold by.
- On the other hand, applying a hard reset with `"zero"` promotes sparsity and potentially less power consumption when running on dedicated neuromorphic hardware.

We can experiment with both depending on the situation.

2.5)Modified LIF neuron (snn.leaky)

The LIF neuron we've discussed till now is unsuitable for large scale deep learning as it involves tuning a lot of hyperparameters. We try to modify it to make it more suitable for large scale DL applications:

$$U(t+\Delta t) = (1-\frac{\Delta t}{\tau})U(t) + \frac{\Delta t}{\tau} I_{\text{in}}(t)R$$

Define the **decay rate** as $\beta = (1-\frac{\Delta t}{\tau})$

We can further reduce hyperparameters by assuming **R=1**. Also, if t represents time instants (i.e time steps in a sequence), then Δt **=1**.Additionally, it is assumed that the input current **instantaneously contributes to the membrane potential**. We then get:

$$U[t+1] = \beta U[t] + (1-\beta) I_{\text{in}}[t+1]$$

We can replace the effect of the input current by the **input features scaled by a learnable parameter W**. We also need to incorporate the reset mechanism. The equation then finally boils down to:

$$U[t+1] = \underbrace{\beta U[t]}_{\text{decay}} + \underbrace{W X[t+1]}_{\text{input}} - \underbrace{S[t] U_{\text{thr}}}_{\text{reset}}$$

W is a learnable parameter, and U_{thr} is generally set to 1 (can also be tuned). This means the only hyperparameter to tune is the decay rate β.

In the absence of an input, we have the solution $U(t) = U_0 e^{-\frac{t}{\tau}}$. By definition, B is the ratio of membrane potentials across two subsequent time steps.

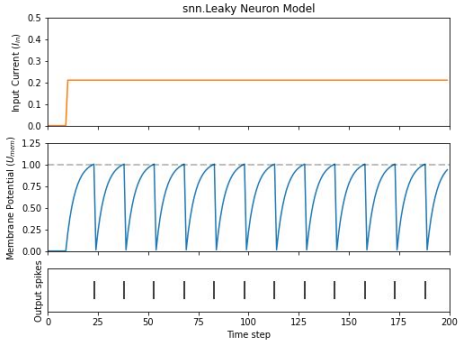
$$\beta = \frac{U_0 e^{-\frac{t+\Delta t}{\tau}}}{U_0 e^{-\frac{t}{\tau}}} = \frac{U_0 e^{-\frac{t+2\Delta t}{\tau}}}{U_0 e^{-\frac{t+\Delta t}{\tau}}} = \dots$$

$$\implies \beta = e^{-\frac{\Delta t}{\tau}}$$

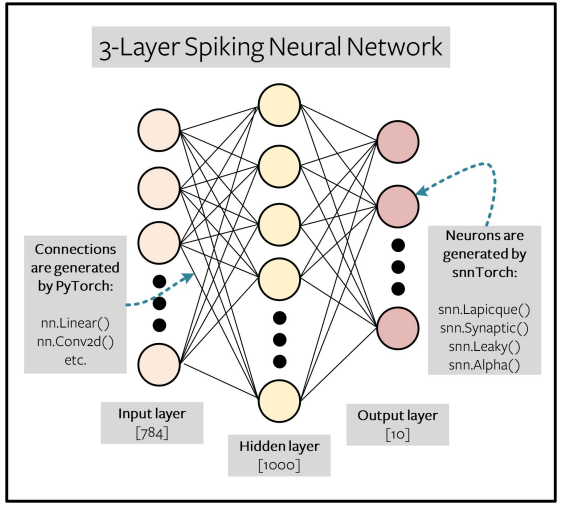
Typically, beta is somewhat close to 1. Typical values can be 0.8,0.9,0.99 etc.

We can use this model by using **snn.leaky** instead of **snn.lapicque:** `lif1 = snn.Leaky(beta=0.8)`

This model has the same optional input arguments of `reset_mechanism` and `threshold` as described for Lapicque's neuron model



3)Feedforward spiking NN



So far, we have only considered how a **single neuron** responds to input stimulus. snnTorch makes it straightforward to scale this up to a deep neural network. In this section, we will create a 3-layer fully-connected neural network of dimensions **784-1000-10**.

We will now discuss the implementation details of this feedforward network using snnTorch.

```
# layer parameters
num_inputs = 784
num_hidden = 1000
num_outputs = 10
beta = 0.99

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

# Initialize hidden states
mem1 = lif1.init_leaky()
mem2 = lif2.init_leaky()

# record outputs
mem2_rec = []
spk1_rec = []
spk2_rec = []

spk_in = spikegen.rate_conv(torch.rand((200, 784))).unsqueeze(1)
```

nn .linear initializes the weights

The static method `init_leaky()` ensures the shape of hidden states are automatically initialized based on the input data dimensions during the first forward pass.

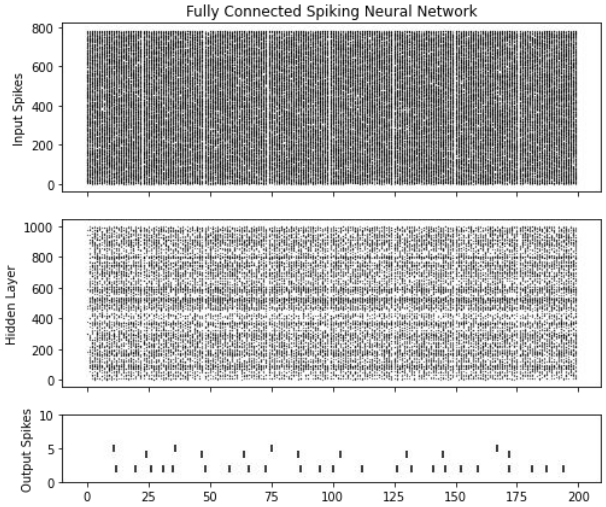
There are 200 time steps to simulate across 784 input neurons, i.e., the input originally has dimensions of 200×784

However, neural nets typically process data in minibatches. snnTorch, uses time-first dimensionality: **[time x batch_size x feature_dimensions]**

- The ith input from `spk_in` to the jth neuron is weighted by the parameter Wij, initialized by **nn.linear**.
- This contributes to the value of U[t+1]
- If U[t+1] >U_thr, then a spike is generated.
- Now, this spike behaves as the input for the second layer, and the above process is repeated for all inputs,weights and neurons. If there is no spike, nothing is passed to the post synaptic neuron.

Implementation:

```
# network simulation
for step in range(num_steps):
    cur1 = fc1(spk_in[step]) # post-synaptic current <-- spk_in x weight
    spk1, mem1 = lif1(cur1, mem1) # mem[t+1] <--post-syn current + decayed membrane
    cur2 = fc2(spk1)
    spk2, mem2 = lif2(cur2, mem2)

    mem2_rec.append(mem2)
    spk1_rec.append(spk1)
    spk2_rec.append(spk2)

# convert lists to tensors
mem2_rec = torch.stack(mem2_rec)
spk1_rec = torch.stack(spk1_rec)
spk2_rec = torch.stack(spk2_rec)

 plot_snn_spikes(spk_in, spk1_rec, spk2_rec, "Fully Connected Spiking Neural Network")
```



At this stage, the spikes don’t have any real meaning. The inputs and weights are all randomly initialized, and no training has taken place.

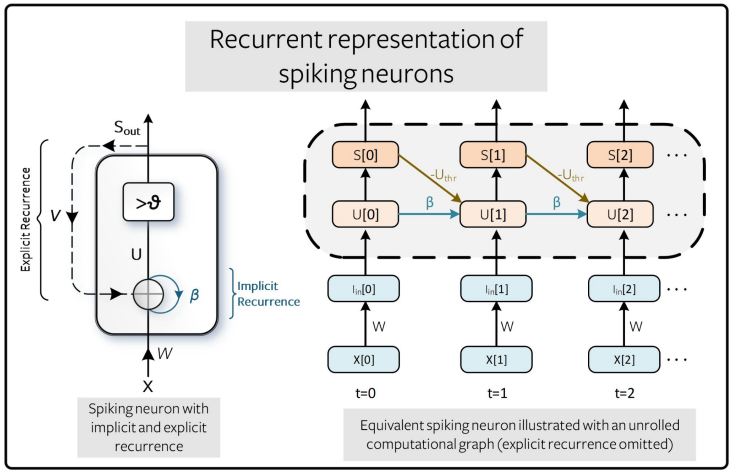
We will now move on the next section, where we discuss how we can train this network properly and update the weights appropriately.

4)Training Spiking Neural Networks

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

$$S[t] = \begin{cases} 1, & \text{if } U[t] > U_{\text{thr}} \\ 0, & \text{otherwise} \end{cases}$$

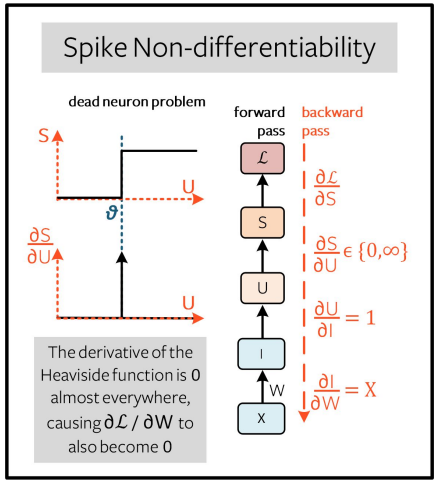
This formulation of a spiking neuron in a discrete, recursive form allows us to take advantage of techniques involved in the training of Recurrent Neural nets, or RNNs.



An equivalent way to represent the spikes can be using the Heaviside step function: $S[t] = \Theta(U[t] - U_{\text{thr}})$

We now define a Loss Function L, based on some combination of the predicted output and expected out[ut. (For eg, MSE, Crossentropy). The goal of ‘training’ is to update the Weights in such a way to minimize the Loss. This is done using gradient descent: W=W-lr*dL/dW

To compute the derivative, the backpropagation algorithm is used, which uses the chain rule:



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

Unless U sits precisely at the threshold(in which case the derivative will be infinity), the derivative will be 0. Effectively, the derivative will be zero all the time, and hence there will be no ‘learning’. This is called the **dead neuron problem.**

The dead neuron problem can be overcome by the **surrogate gradient** approach. The idea is to keep the heaviside function as it is for the forward pass, but replace ds/dU something else : ds~/dU, so that we do not end up with a zero gradient all the time.

The simplest method, called the spike-operator approach , simply sets ds~/dW to be equal to s. If the neuron spikes, the derivative is 1, else its 0.

$$\frac{\partial \tilde{S}}{\partial U} \leftarrow S = \begin{cases} 1, & \text{if } U > U_{\text{thr}} \\ 0, & \text{otherwise} \end{cases}$$

This is already implemented in the **snn.Leaky** method.

With the dead neuron problem now resolved, we can use the backprop algorithm to compute the gradients. But before that, there is another vital aspect which we have to consider:

The backprop equation that we've discussed calculates the gradient only for **a single time step** .

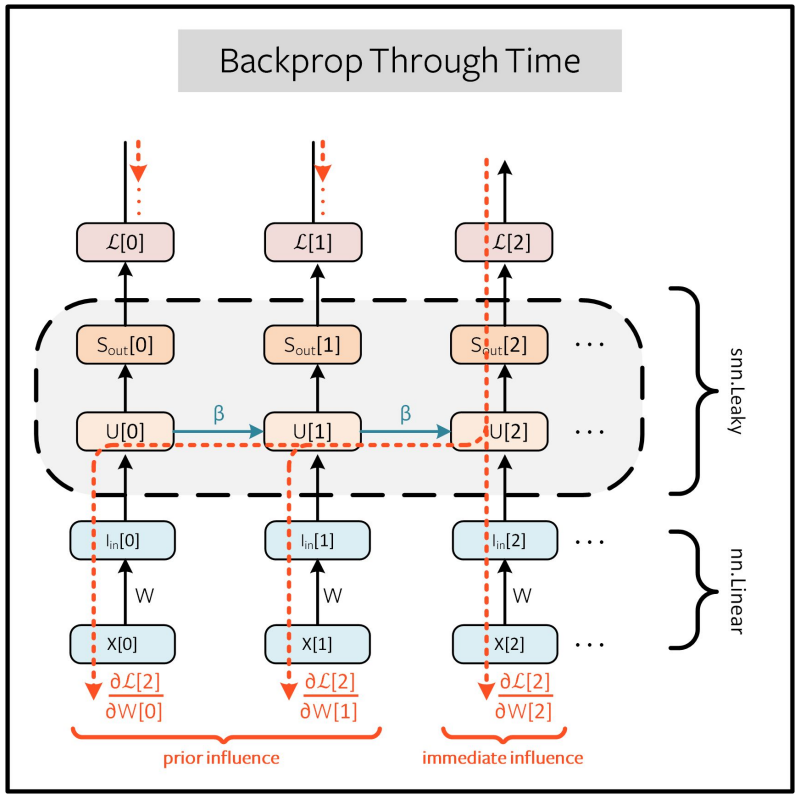
The weight W is applied at every time step, and so a loss is also calculated at every time step. The influence of the weight on **present and historical losses** must be summed together to define the global gradient.

This is achieved using the **Backpropagation Through Time (BPTT).** It calculates the gradient from the loss to all descendants, and sums them together:

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

By constraining s<=t, we only account for the prior and immediate influences on W on the loss. Also, a recurrent system ensures constraints the weight to be shared across all steps: W[0]=W[1].. And so dW[s]/dW=1. We then have:

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



Pytorch's autodiff functionality (torch.autograd) takes care of computing these gradients.

5)Output Decoding

We've built the network, updated the weights, and minimized the loss. We get an output in terms of spikes, and all that is left is to decode this spike information into predictions. This is the reverse of what we did in step 1, where we encoded the input information into spikes. This can be done in a variety of ways:

- **Rate coding:** Take the neuron with the highest firing rate (or spike count) as the predicted class
- **Latency coding:** Take the neuron which fires first as the predicted neuron.

We discuss **Rate coding** in this report. When input data is passed to the network,we want the correct neuron class to emit the most spikes. One way to achieve this is to increase the membrane potential of the correct class to U>Uth, and that of the incorrect class to U<Uth.

This can be implemented using the softmax function: by taking the softmax of the membrane potential for output neurons :

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

We can then define the Loss using cross-entropy between pi and the target yi {0,1} (which is a one-hot encoded vector)

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

Minimizing L will then have the effect of encouraging the mebrane potential of the correct class to increase, and that of the incorrect class to reduce. The correct class will be encouraged to fire at all times, whereas incorrect classes are suppressed.

This target is applied at every time step of the simulation, thus also generating a loss at every step. These losses are then summed together at the end of the simulation:

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

6)Summary

- We first encoded the input images into spikes.
- We then mathematically modelled a single neuron.
- We created a feedforward network using a bunch of these neurons together.
- We used the surrogate gradient approach to properly implement the backprop (through time) algorithm in order to update the weights.
- We used a softmax function and a defined the loss function in terms of cross entropy between the predictions and the output. This was part of the rate-coded approach for output decoding.

Using these steps, we can create a classification model that can classify handwritten digits, with accuracy comparable to that achieved by traditional DNNs, but using far less computational power, making it hardware friendly.

7)References

- Jason K. Eshraghian, Max Ward, Emre Neftci, Xinxin Wang, Gregor Lenz, Girish Dwivedi, Mohammed Bennamoun, Doo Seok Jeong, and Wei D. Lu. “Training Spiking Neural Networks Using Lessons From Deep Learning”

- E. O. Neftci, H. Mostafa, F. Zenke, Surrogate Gradient Learning in Spiking Neural Networks: Bringing the Power of Gradient-based optimization to spiking neural networks