Bootstrap SNN Training

The underlying principle for ANN-SNN conversion is that the ReLU activation function (or similar form) approximates the firing rate of an LIF spiking neuron. Consequently, an ANN trained with ReLU activation can be mapped to an equivalent SNN with proper scaling of weights and thresholds. However, as the number of time-steps reduces, the alignment between ReLU activation and LIF spiking rate falls apart mainly due to the following two reasons (especially, for discrete-in-time models like Loihi's CUBA LIF):

- With less time steps, the SNN can assume only a few discrete firing rates.
- Limited time steps mean that the spiking neuron activity rate often saturates to maximum allowable firing rate.

Introducing **Bootstrap training**. An SNN is used to jumpstart an equivalent ANN model which is then used to accelerate SNN training. There is no restriction on the type of spiking neuron or it's reset behavior. It consists of following steps:

- Input output data points are first collected from the network running as an SNN: **SAMPLING mode**.
- The data is used to estimate the corresponding ANN activation as a piecewise linear layer, unique to each layer: **FIT mode**.
- The training is accelerated using the piecewise linear ANN activation: **ANN mode**.
- The network is seamlessly translated to an SNN: **SNN mode**.
- *SAMPLING mode* and *FIT mode* are repeated for a few iterations every couple of epochs, thus maintaining an accurate ANN estimate.

Drawing

Bootstrap training is available as `lava.lib.dl.bootstrap`. The main modules are

- `block`: provides `lava.lib.dl.slayer.block` based network definition interface.
- `ann_sampler`: provides utilities for sampling SNN data points and pievewise linear ANN fit.
- `routine`: `routine.Scheduler` provides scheduling utility to seamlessly switch between SAMPLING | FIT | ANN | SNN mode.
  - It also provides ANN-SNN bootstrap hybrid traiing utility as well (Not demonstrated in this tutorial).

Drawing

MNIST Classification

Here, we will demonstrate botstrap SNN training on the well known MNIST classification problem.

In [1]:
```
import os, sys
import h5py
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import torch
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torchvision import datasets, transforms

# import slayer from lava-dl
import lava.lib.dl.slayer as slayer
import lava.lib.dl.bootstrap as bootstrap

import IPython.display as display
from matplotlib import animation
```

Create Network

The network definition follows standard PyTorch way using `torch.nn.Module`.

`lava.lib.dl.bootstrap` provides **block interface** similar to `lava.lib.dl.slayer.block` - which bundles all these individual components into a single unit. These blocks can be cascaded to build a network easily. The block interface provides additional utilities for normalization (weight and neuron), dropout, gradient monitoring and network export.

In [ ]:
```
class Network(torch.nn.Module):
    def __init__(self, time_steps=16):
        super(Network, self).__init__()
        self.time_steps = time_steps

        neuron_params = {
                'threshold'     : 1.25,
                'current_decay' : 1, # this must be 1 to use batchnorm
                'voltage_decay' : 0.03,
                'tau_grad'      : 1,
                'scale_grad'    : 1,
            }
        neuron_params_norm = {
                **neuron_params,
                # 'norm'    : slayer.neuron.norm.MeanOnlyBatchNorm,
            }

        self.blocks = torch.nn.ModuleList([
                bootstrap.block.cuba.Input(neuron_params, weight=1, bias=0),
                bootstrap.block.cuba.Dense(neuron_params_norm, 28*28, 512, w
                bootstrap.block.cuba.Dense(neuron_params_norm, 512, 512, wei
                bootstrap.block.cuba.Affine(neuron_params, 512, 10, weight_r
            ])

    def forward(self, x, mode):
        N, C, H, W = x.shape
        if mode.base_mode == bootstrap.Mode.ANN:
            x = x.reshape([N, C, H, W, 1])
        else:
            x = slayer.utils.time.replicate(x, self.time_steps)

        x = x.reshape(N, -1, x.shape[-1])

        for block, m in zip(self.blocks, mode):
            x = block(x, mode=m)

        return x

    def export_hdf5(self, filename):
        # network export to hdf5 format
        h = h5py.File(filename, 'w')
        simulation = h.create_group('simulation')
        simulation['Ts'] = 1
        simulation['tSample'] = self.time_steps
        layer = h.create_group('layer')
        for i, b in enumerate(self.blocks):
            b.export_hdf5(layer.create_group(f'{i}'))
```

In [ ]:

```
```

Instantiate Network, Optimizer, DataSet and DataLoader

Here we will use standard *torchvision datasets* to load MNIST data.

In [ ]:

```
trained_folder = 'Trained'
os.makedirs(trained_folder, exist_ok=True)

# device = torch.device('cpu')
device = torch.device('cuda')

net = Network().to(device)

optimizer = torch.optim.Adam(net.parameters(), lr=0.001)

# Dataset and dataLoader instances.
training_set = datasets.MNIST(
        root='data/',
        train=True,
        transform=transforms.Compose([
            transforms.RandomAffine(
                degrees=10,
                translate=(0.05, 0.05),
                scale=(0.95, 1.05),
                shear=5,
            ),
            transforms.ToTensor(),
            transforms.Normalize((0.5), (0.5)),
        ]),
        download=True,
    )

testing_set = datasets.MNIST(
        root='data/',
        train=False,
        transform=transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.5), (0.5)),
        ]),
    )

train_loader = DataLoader(dataset=training_set, batch_size=32, shuffle=True)
test_loader  = DataLoader(dataset=testing_set , batch_size=32, shuffle=True)

stats = slayer.utils.LearningStats()
scheduler = bootstrap.routine.Scheduler()
```

Training Loop

Training loop follows standard PyTorch training structure. `bootstrap.routine.Scheduler` helps simplify the complex routine of periodically switching between different bootstrap modes during training. `scheduler.mode(epoch, i, net.training)` provides an iterator which orchestrates the mode of different blocks/layers.

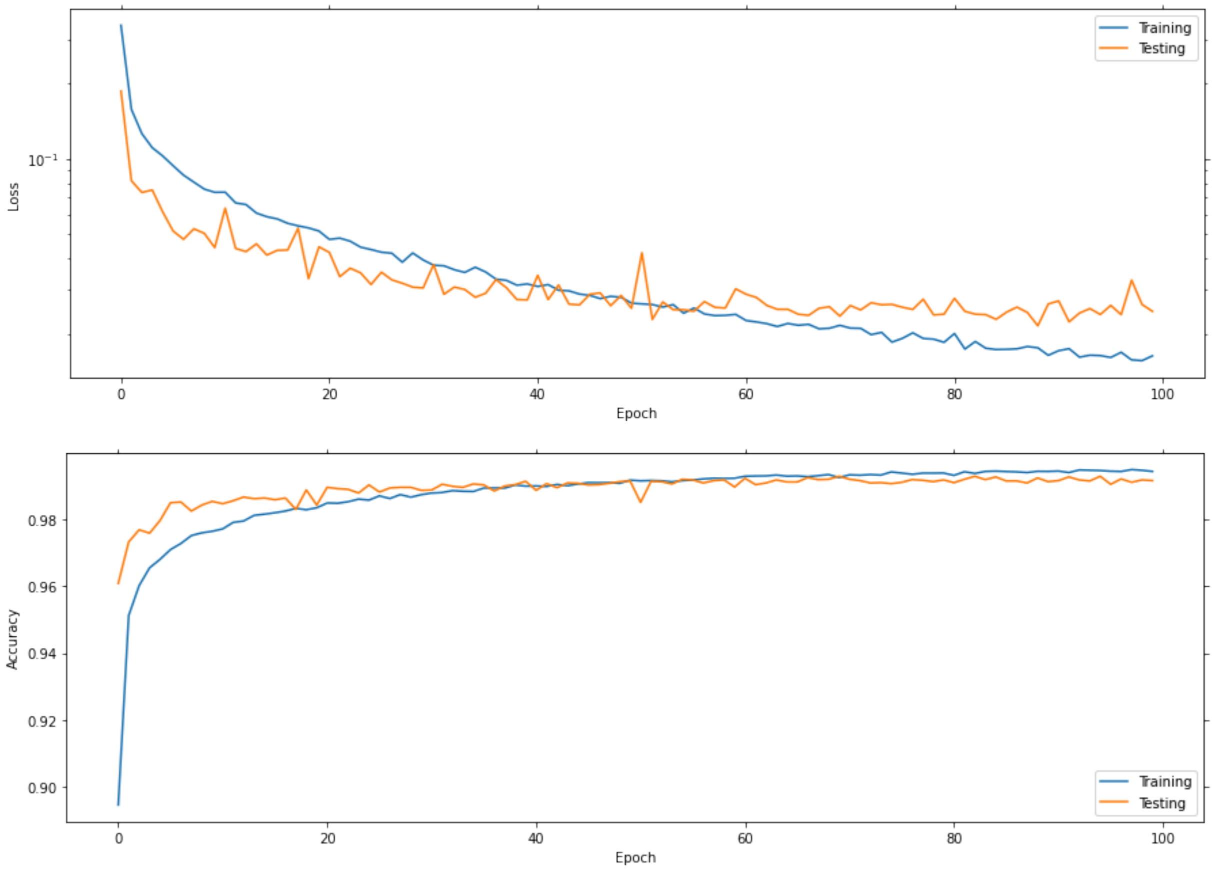
In [ ]:
```
epochs = 50 #changed as per assignment instructions
for epoch in range(epochs):
    for i, (input, label) in enumerate(train_loader, 0):
        net.train()
        mode = scheduler.mode(epoch, i, net.training)

        input = input.to(device)
        output = net.forward(input, mode)
        rate = torch.mean(output, dim=-1).reshape((input.shape[0], -1))

        loss = F.cross_entropy(rate, label.to(device))
        prediction = rate.data.max(1, keepdim=True)[1].cpu().flatten()

        stats.training.num_samples += len(label)
        stats.training.loss_sum += loss.cpu().data.item() * input.shape[0]
        stats.training.correct_samples += torch.sum( prediction == label ).c

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    print(f'\r[Epoch {epoch:2d}/{epochs}] {stats}', end='')

    for i, (input, label) in enumerate(test_loader, 0):
        net.eval()
        mode = scheduler.mode(epoch, i, net.training)

        with torch.no_grad():
            input = input.to(device)
            output = net.forward(input, mode=scheduler.mode(epoch, i, net.tr
            rate = torch.mean(output, dim=-1).reshape((input.shape[0], -1))

            loss = F.cross_entropy(rate, label.to(device))
            prediction = rate.data.max(1, keepdim=True)[1].cpu().flatten()

        stats.testing.num_samples += len(label)
        stats.testing.loss_sum += loss.cpu().data.item() * input.shape[0]
        stats.testing.correct_samples += torch.sum( prediction == label ).da

    print(f'\r[Epoch {epoch:2d}/{epochs}] {stats}', end='')

    if mode.base_mode == bootstrap.routine.Mode.SNN:
        scheduler.sync_snn_stat(stats.testing)
        print('\r', ' '*len(f'\r[Epoch {epoch:2d}/{epochs}] {stats}'))
        print(mode)
        print(f'[Epoch {epoch:2d}/{epochs}]\nSNN Testing: {scheduler.snn_sta

        if scheduler.snn_stat.best_accuracy:
            torch.save(net.state_dict(), trained_folder + '/network.pt')
        scheduler.update_snn_stat()

    stats.update()
    stats.save(trained_folder + '/')
```

```
Mode: SNN
[Epoch  0/100]
SNN Testing: loss =     0.18656                       accuracy = 0.96080

Mode: SNN
[Epoch 10/100]
SNN Testing: loss =     0.06348 (min =     0.18656)    accuracy = 0.98460 (m
ax = 0.96080)

Mode: SNN
[Epoch 20/100]
SNN Testing: loss =     0.04227 (min =     0.06348)    accuracy = 0.98950 (m
ax = 0.98460)

Mode: SNN
[Epoch 30/100]
SNN Testing: loss =     0.03778 (min =     0.04227)    accuracy = 0.98870 (m
ax = 0.98950)

Mode: SNN
[Epoch 40/100]
SNN Testing: loss =     0.03424 (min =     0.03778)    accuracy = 0.98860 (m
ax = 0.98950)

Mode: SNN
[Epoch 50/100]
SNN Testing: loss =     0.04215 (min =     0.03424)    accuracy = 0.98500 (m
ax = 0.98950)

Mode: SNN
[Epoch 60/100]
SNN Testing: loss =     0.02876 (min =     0.03424)    accuracy = 0.99220 (m
ax = 0.98950)

Mode: SNN
[Epoch 70/100]
SNN Testing: loss =     0.02592 (min =     0.02876)    accuracy = 0.99190 (m
ax = 0.99220)

Mode: SNN
[Epoch 80/100]
SNN Testing: loss =     0.02771 (min =     0.02592)    accuracy = 0.99090 (m
ax = 0.99220)

Mode: SNN
[Epoch 90/100]
SNN Testing: loss =     0.02705 (min =     0.02592)    accuracy = 0.99150 (m
ax = 0.99220)
[Epoch 99/100] Train loss =     0.01632 (min =     0.01561)    accuracy = 0.
99425 (max = 0.99480) | Test  loss =     0.02458 (min =     0.02152)    accu
racy = 0.99150 (max = 0.99280)
```

Plot the learning curves

Plotting the learning curves is as easy as calling `stats.plot()` .

```
In [ ]: stats.plot(figsize=(15, 5))
```



Export the best model

Load the best model during training and export it as hdf5 network. It is supported by `lava.lib.dl.netx` to automatically load the network as a lava process.

```
In [ ]: net.load_state_dict(torch.load(trained_folder + '/network.pt'))
        net.export_hdf5(trained_folder + '/network.net')
```

Visualize the network output

Here, we will use `slayer.io.tensor_to_event` method to convert the torch output spike tensor into graded (non-binary) `slayer.io.Event` object and visualize a few input and output event pairs.

```
In [ ]: output = net(input.to(device), mode=scheduler.mode(100, 0, False))
        for i in range(5):
            img = (2*input[i].reshape(28, 28).cpu().data.numpy()-1) * 255
            Image.fromarray(img).convert('RGB').save(f'gifs/inp{i}.png')
```

```
    out_event = slayer.io.tensor_to_event(output[i].cpu().data.numpy().resha
    out_anim = out_event.anim(plt.figure(figsize=(10, 3.5)), frame_rate=240
    out_anim.save(f'gifs/out{i}.gif', animation.PillowWriter(fps=24), dpi=3
```

In [ ]:

```
img_td = lambda gif: f'<td> <img src="{gif}" alt="Drawing" style="height: 15
html = '<table>'
html += '<tr><td align="center"><b>Input</b></td><td><b>Output</b></td></tr>
for i in range(5):
    html += '<tr>'
    html += img_td(f'gifs/inp{i}.png')
    html += img_td(f'gifs/out{i}.gif')
    html += '</tr>'
html += '</tr></table>'
display.HTML(html)
```

Out[ ]:

Input	**Output**
Drawing	Drawing
Drawing	Drawing
Drawing	Drawing
Drawing	Drawing
Drawing	Drawing