Two task network

Network has some inputs

1. The fixation.
2. The first context mod.
3. The second ontext mod.

Network has five outputs

1. The fixation.
2. The first output.
3. The second output

> Learning rule: superspike
>
> Neuron type: Lif + refrac
>
> Task: dm

In [ ]:

```
import torch
import numpy as np
import torch.nn as nn
import matplotlib.pyplot as plt  # for analys
from cgtasknet.net.lifrefrac import SNNLifRefrac
from cgtasknet.tasks.reduce import DMTask
from norse.torch.functional.lif_refrac import LIFRefracParameters
from norse.torch.functional.lif import LIFParameters

# from norse.torch import LIF
```

Step -1: Create dataset

In [ ]:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# device = torch.device('cpu')
print(f'Device: {("gpu (cuda)" if device.type=="cuda" else "cpu")}')
```

```
Device: gpu (cuda)
```

In [ ]:

```
batch_size = 100
number_of_tasks = 1
#tasks = dict(task_list)
Task = DMTask(batch_size=batch_size)
Task_test = DMTask(batch_size=1)
```

Step 1.1: Create model

In [ ]:

```
feature_size, output_size = Task.feature_and_act_size
hidden_size = 400

neuron_parameters = LIFRefracParameters(
    LIFParameters(tau_mem_inv=torch.as_tensor(1/0.01).to(device),
        alpha=torch.as_tensor(100), method="super", v_th=torch.as_tensor(0.65
    ),
    rho_reset=torch.as_tensor(1),
)
model = SNNLifRefrac(
    feature_size,
    hidden_size,
    output_size,
    neuron_parameters=neuron_parameters,
    tau_filter_inv=500,
).to(device)
```

Step 1.2: Save pre-learning weights

In [ ]:

```
weights_pre_l = []
with torch.no_grad():
    for name, param in model.named_parameters():
        if param.requires_grad:
            weights_pre_l.append((param).cpu().numpy())
```

Step 2: loss and creterion

In [ ]:

```
learning_rate = 1e-3


class RMSELoss(nn.Module):
    def __init__(self):
        super().__init__()
        self.mse = nn.MSELoss()

    def forward(self, yhat, y):
        return torch.sqrt(self.mse(yhat, y))


criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
#optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Step 3: Train loop

In [ ]:

```
%matplotlib
plt.ion
fig = plt.figure()
ax = fig.add_subplot(111)

ax.set_title("lif")
inputs, target_outputs = Task.dataset(number_of_tasks)
(line1,) = ax.plot(np.arange(0, len(target_outputs)), target_outputs[:, 0, 1]
(line2,) = ax.plot(np.arange(0, len(target_outputs)), target_outputs[:, 0, 2]
(line3,) = ax.plot(np.arange(0, len(target_outputs)), target_outputs[:, 0, 1]
(line4,) = ax.plot(np.arange(0, len(target_outputs)), target_outputs[:, 0, 2]
ax.set_ylim([-0.5, 1.5])
ax.set_xlim([0, len(inputs)])
running_loss = 0
fig.canvas.draw()
fig.canvas.flush_events()
for i in range(300):
    inputs, target_outputs = Task.dataset(number_of_tasks)
    #inputs = inputs * 2 - 1
    inputs += np.random.normal(0, 0.01, size=(inputs.shape))
    inputs = torch.from_numpy(inputs).type(torch.float).to(device)
    target_outputs = torch.from_numpy(target_outputs).type(torch.float).to(de
    #with torch.no_grad():
    #    inputs[:, :, 1:3] = layer_inputs(inputs[:, :, 1:3])[0]
    # zero the parameter gradients
    optimizer.zero_grad()

    # forward + backward + optimize
    outputs, states = model(inputs)

    loss = criterion(outputs, target_outputs)
    loss.backward()
    optimizer.step()

    # print statistics
    running_loss += loss.item()
    if i % 10 == 9:
        print("epoch: {:d} loss: {:0.5f}".format(i + 1, running_loss / 10))
        running_loss = 0.0
        with torch.no_grad():
            inputs, target_outputs = Task_test.dataset(number_of_tasks)

            inputs = torch.from_numpy(inputs).type(torch.float).to(device)
            target_outputs = (
                torch.from_numpy(target_outputs).type(torch.float).to(device)
            )
            outputs, states = model(inputs)
            loss = criterion(outputs, target_outputs)

            print("test loss: {:0.5f}".format(loss.item()))
        for_plot = outputs.detach().cpu().numpy()[:, 0, :]
        line1.set_xdata(np.arange(0, len(for_plot), 1))
        line2.set_xdata(np.arange(0, len(for_plot), 1))
        line3.set_xdata(np.arange(0, len(for_plot), 1))
        line4.set_xdata(np.arange(0, len(for_plot), 1))

        line1.set_ydata(for_plot[:, 1])
        line2.set_ydata(for_plot[:, 2])
```

```
Using matplotlib backend: TkAgg
epoch: 10 loss: 0.00644
test loss: 0.00484
epoch: 20 loss: 0.01082
test loss: 0.00472
epoch: 30 loss: 0.01173
test loss: 0.00460
epoch: 40 loss: 0.00790
test loss: 0.00405
epoch: 50 loss: 0.00921
test loss: 0.00524
epoch: 60 loss: 0.00928
test loss: 0.00925
epoch: 70 loss: 0.00977
test loss: 0.00407
epoch: 80 loss: 0.00976
test loss: 0.00465
epoch: 90 loss: 0.00878
test loss: 0.00407
epoch: 100 loss: 0.01001
test loss: 0.00494
epoch: 110 loss: 0.00797
test loss: 0.00440
epoch: 120 loss: 0.00839
test loss: 0.00414
epoch: 130 loss: 0.01126
test loss: 0.00508
epoch: 140 loss: 0.01058
test loss: 0.00494
epoch: 150 loss: 0.01406
test loss: 0.00493
epoch: 160 loss: 0.01175
test loss: 0.00503
epoch: 170 loss: 0.00948
test loss: 0.00588
epoch: 180 loss: 0.01307
test loss: 0.00718
epoch: 190 loss: 0.00976
test loss: 0.00719
epoch: 200 loss: 0.01087
test loss: 0.00696
epoch: 210 loss: 0.00876
test loss: 0.20322
epoch: 220 loss: 0.01408
test loss: 0.08334
epoch: 230 loss: 0.01474
test loss: 0.00549
epoch: 240 loss: 0.01418
test loss: 0.00643
epoch: 250 loss: 0.00915
test loss: 0.00484
epoch: 260 loss: 0.00805
test loss: 0.00516
```

```
epoch: 270 loss: 0.00867
test loss: 0.00414
epoch: 280 loss: 0.00883
test loss: 0.00641
epoch: 290 loss: 0.00973
test loss: 0.00479
epoch: 300 loss: 0.00726
test loss: 0.00424
```

In [ ]:

```
torch.save(model.state_dict(), "Only_dm_lif_refrac_net")
```

In [ ]:

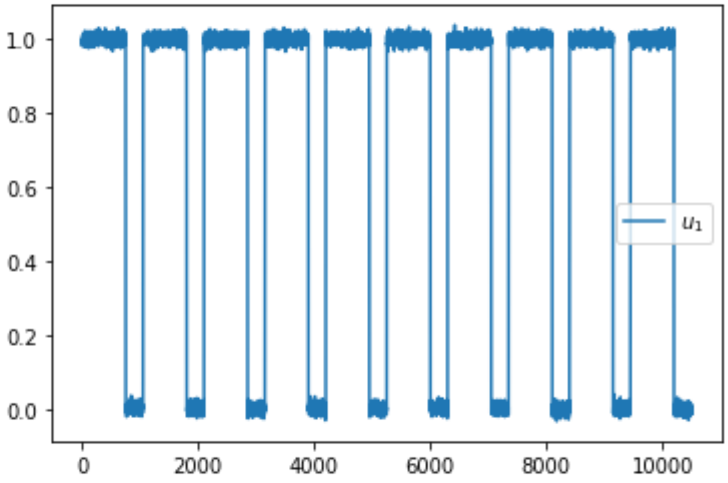
```
if False:
    model.load_state_dict(
        torch.load("Only_dm_lif_net")
    )
```

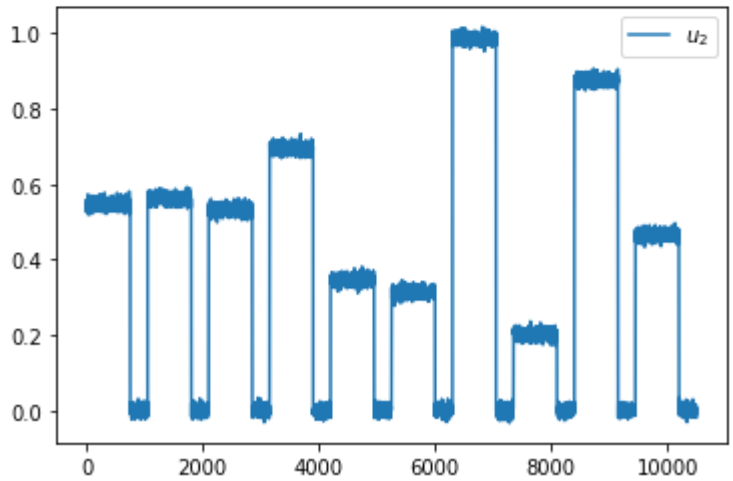
In [ ]:

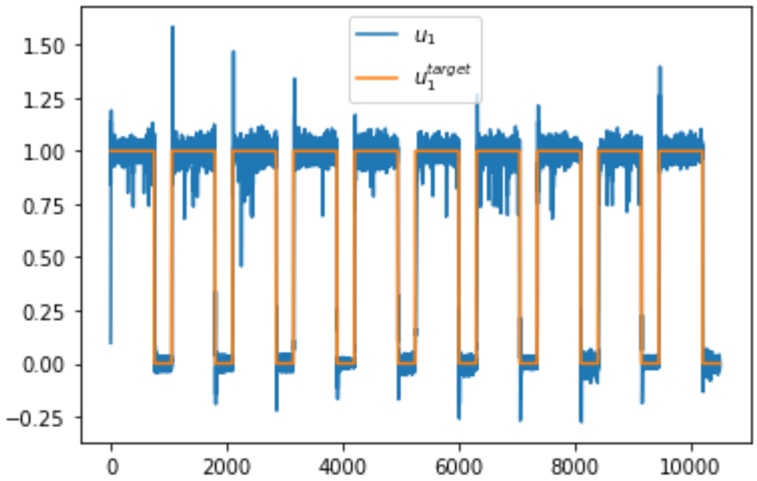
```
Taskplot = DMTask(batch_size=1)
inputs, target_outputs = Taskplot.dataset(10)
inputs += np.random.normal(0, 0.01, size=(inputs.shape))
inputs = torch.from_numpy(inputs).type(torch.float).to(device)
target_outputs = torch.from_numpy(target_outputs).type(torch.float).to(device
outputs, states = model(inputs)
```

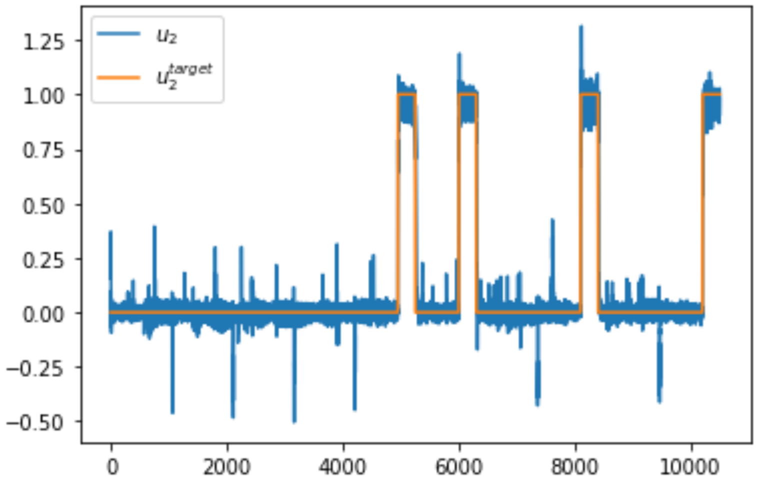
In [ ]:

```
%matplotlib inline
for i in range(inputs.shape[2]):
    plt.plot(inputs[:, 0, i].detach().cpu().numpy(), label=fr"$u_{i + 1}$")
    plt.legend()
    plt.show()
    plt.close()
for i in range(outputs.shape[2]):
    plt.plot(outputs[:, 0, i].detach().cpu().numpy(), label=fr"$u_{i + 1}$")
    plt.plot(
        target_outputs[:, 0, i].detach().cpu().numpy(), label=fr"$u^{{target}
    )
    plt.legend()
    plt.show()
    plt.close()
```




u_2
1.0
0.8
0.6
0.4
0.2
0.0
0
2000
4000
6000
8000
10000

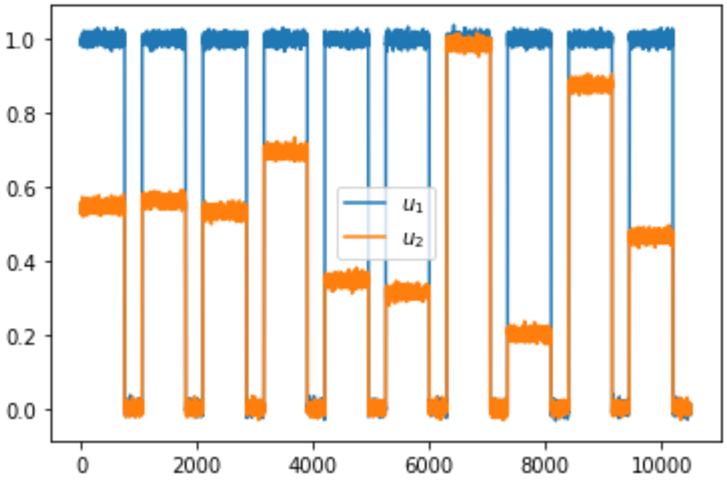

u_1
u_1^{target}
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
0
2000
4000
6000
8000
10000


u_2
u_2^{target}
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
0
2000
4000
6000
8000
10000

In [ ]:

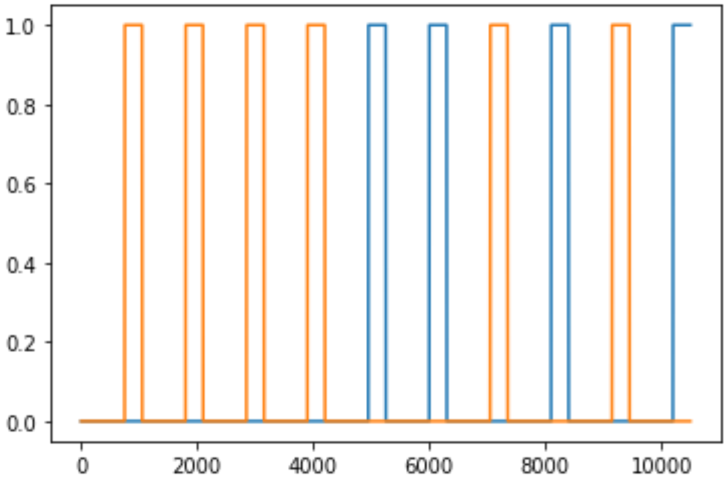
```
%matplotlib inline
for i in range(inputs.shape[2]):
    plt.plot(inputs[:, 0, i].detach().cpu().numpy(), label=fr"$u_{i + 1}$")

plt.legend()
plt.show()
plt.close()
plt.plot(target_outputs[:, 0, 1].detach().cpu().numpy(), label=fr'$y_{2}$')
plt.plot(target_outputs[:, 0, 2].detach().cpu().numpy(), label=fr'$y_{3}$')
```



Out[ ]:

```
[<matplotlib.lines.Line2D at 0x7fbbc85a7af0>]
```

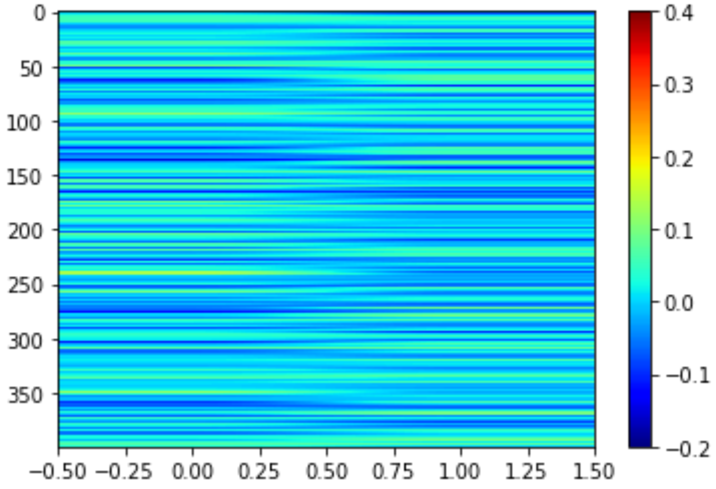


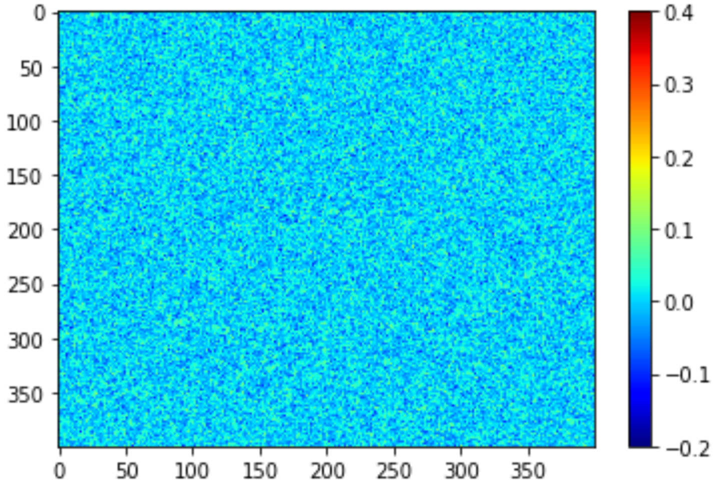
In [ ]:

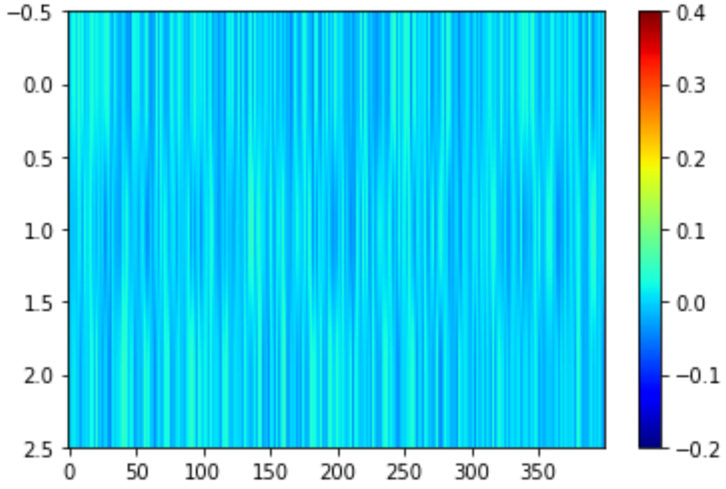
```
weights_post_l = []
with torch.no_grad():
    for name, param in model.named_parameters():
        if param.requires_grad:
            weights_post_l.append((param).cpu().numpy())
```

In [ ]:

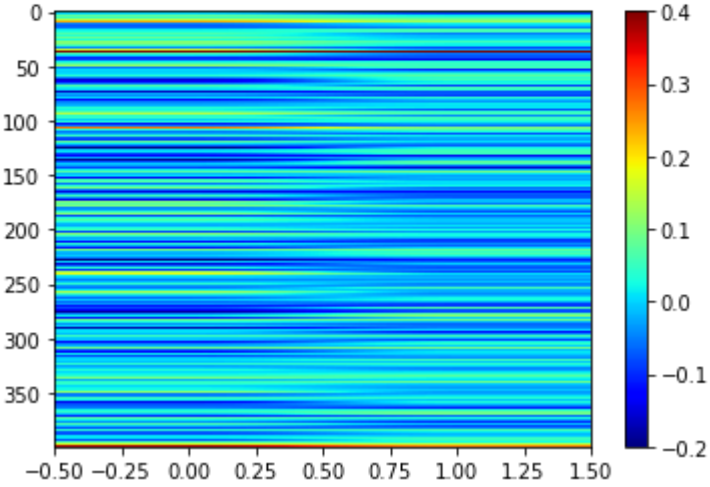
```
%matplotlib inline
for i in range(len(weights_pre_l) - 1):
    plt.imshow((weights_pre_l[i]), aspect='auto',cmap='jet', vmin=-.2, vmax=0
    plt.colorbar()
    plt.show()
```

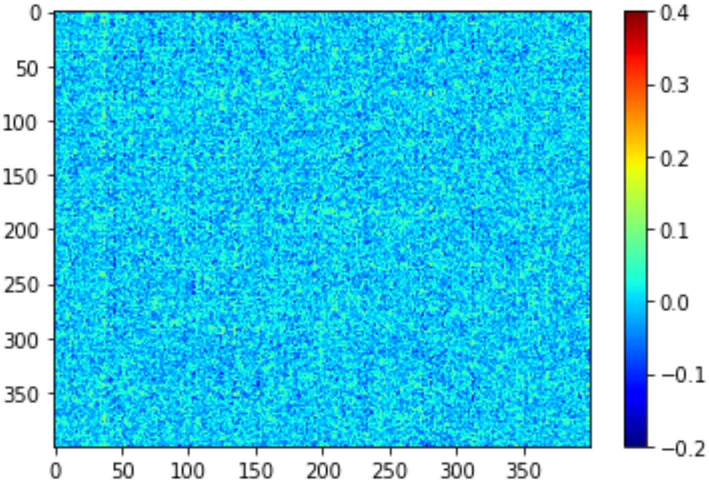


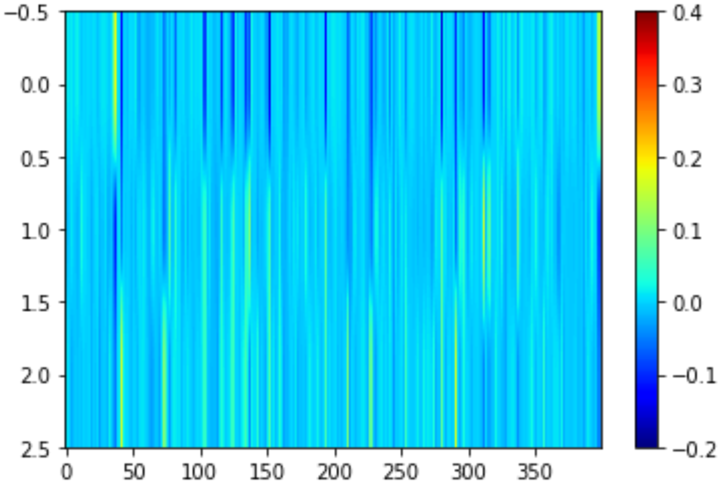




```
In [ ]: %matplotlib inline
        for i in range(len(weights_pre_l) - 1):
            plt.imshow((weights_post_l[i]), aspect='auto', cmap='jet', vmin=-.2, vmax
            plt.colorbar()
            plt.show()
```

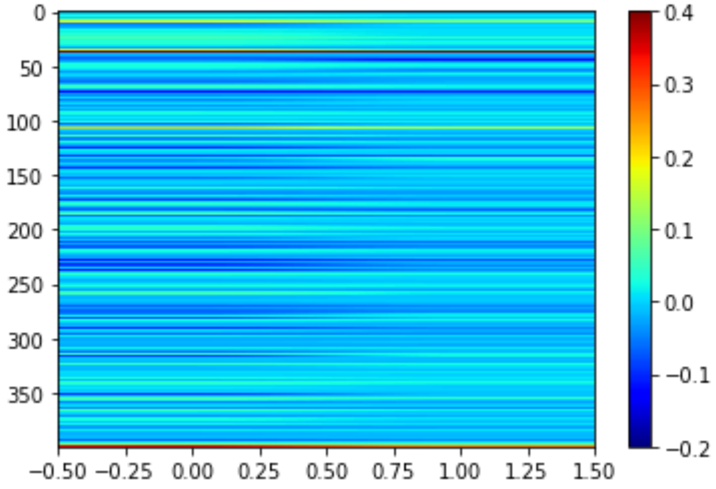


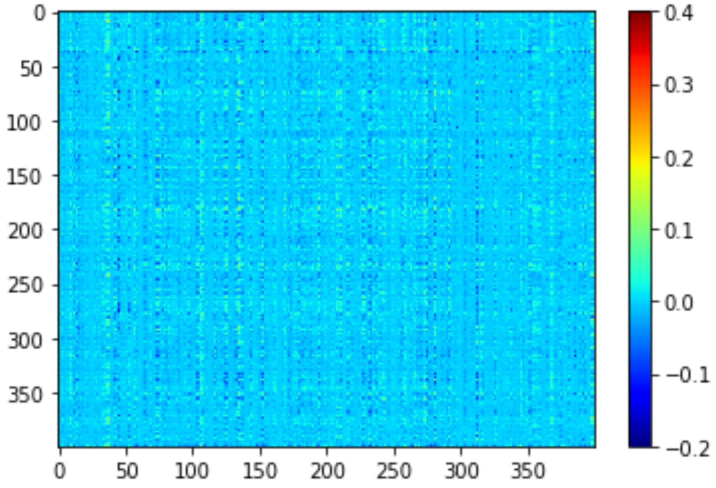


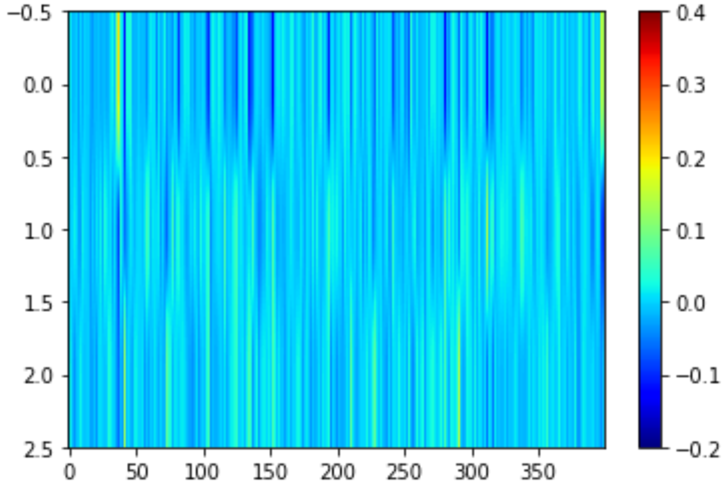


In [ ]:

```
%matplotlib inline
for i in range(len(weights_pre_l) - 1):
    plt.imshow((weights_post_l[i] - weights_pre_l[i]), aspect='auto', cmap='j
    plt.colorbar()
    plt.show()
```







In [ ]: