2-Layer TNN

Tuesday, April 21, 2020 5:50 PM

2 layer TNN
2 columns feeding into 3rd col.
Each of the 1st two cols get all of the input image at once.

For STDP parameters:
capture and backoff are the same.
I increase search for the 2nd layer because it receives far sparser input and needs more 'encouragement'.
2-WTA is used for the two base columns, but 1-WTA is used for the output column.

rTNN V2 (Multi-clustering, receptive slices, 2 buffers)

Thursday, April 23, 2020 1:55 PM

2 input clustering sections, with overlapping receptive slices
These 2 clustering columns feed into a recurrent layer, which has two chained buffer layers each feeding back into it.

Params:
input_size = 28
input_slice = 16
tnn_layer_sz = 50
rtnn_layer_sz = 100
num_timesteps = 16
tnn_thresh = 64
rtnn_thresh = 16
max_weight = num_timesteps
num_winners_tnn = 1
num_winners_rtnn = rtnn_layer_sz//10
stdp_tnn_params = {
"ucapture": 8/128,
"uminus": 8/128,
"usearch": 2/128,
"ubackoff": 96/128,
"umin": 4/128,
"maxweight": max_weight
}
stdp_rtnn_params = {
"ucapture": 10/128,
"uminus": 10/128,
"usearch": 30/128,
"ubackoff": 96/128,
"umin": 16/128,
"maxweight": max_weight
}
w_eye_rtnn = max_weight * torch.diag(torch.ones(rtnn_layer_1.n))

Accuracy of the model on 1 test images: 11.32 %

After trying again with diff test and train sets:
Training the read out): 2it [00:34, 13.57s/it]
Epoch: 10/10, Loss: 1.0982: 10it [00:03, 2.85it/s]:56, 1.54it/s]

Accuracy of the model on 1000 test images: 9.51 %
--Return--10, Loss: 1.0982: 10it [00:03, 2.84it/s]

rTNN V3 (Multiple Clustering and Recurrence, all STDP)

Saturday, April 25, 2020 5:56 PM

STDP enabled for recurrent and feedworward connections.
Also changed STDP parameters.

```
input_size = 28
input_slice = 20
tnn_layer_sz = 20
rtnn_layer_sz = 50
num_timesteps = 16
tnn_thresh = 64
rtnn_thresh = 8
max_weight = 16
max_weight_rtnn = 16
num_winners_tnn = 1
num_winners_rtnn = rtnn_layer_sz//10


stdp_tnn_params = {
   "ucapture":  8/128,
   "uminus":    8/128,
   "usearch":   2/128,
   "ubackoff":  96/128,
   "umin":      4/128,
   "maxweight": max_weight
}
stdp_rtnn_params = {
   "ucapture":  15/128,
   "uminus":    15/128,
   "usearch":   10/128,
   "ubackoff":  96/128,
   "umin":      1/128,
   "maxweight": max_weight_rtnn
}
```

The idea here is to reduce the size of the network and actially get the recurrent connections to train somehow.

```
❯ python3.6 rc_buff_seq_V3.py --plot --examples 1000

Pre-Training progress: (1000 / 1000): 1001it [10:36,  1.54it/s]Press enter to continue to plotting...

 Training the read out): 2it [00:16,  7.45s/it]
Readout Training progress: (1000 / 1000): 1001it [07:07,  2.37it/s]


Epoch: 4/10, Loss: 0.9008: 4it [00:28,  7.12s/it]

Epoch: 10/10, Loss: 0.8976: 10it [01:11,  7.13s/it]
```

Accuracy of the model on 1000 test images: 14.00 %
--Return--10, Loss: 0.8976: 10it [01:11, 7.18s/it]
> /mnt/c/Users/markp/Repos/18847-rTNN/rc_buff_seq_V3.py(297)<module>()->None

rTNN V4 (Multiple clustering into recurrent, 2 buffers)

Saturday, April 25, 2020 7:51 PM

```
input_size = 28
input_slice = 20
tnn_layer_sz = 20
rtnn_layer_sz = 50
num_timesteps = 16
tnn_thresh = 64
rtnn_thresh = 8
max_weight = 16
max_weight_rtnn = 16
num_winners_tnn = 1
num_winners_rtnn = rtnn_layer_sz//10

time = num_timesteps

torch.manual_seed(seed)

# build network:
network = Network(dt=1)
input_layer_a = Input(n=input_slice)
input_layer_b = Input(n=input_slice)

tnn_layer_1a = TemporalNeurons(
        n=tnn_layer_sz,
        timesteps=num_timesteps,
        threshold=tnn_thresh,
        num_winners=num_winners_tnn
        )
tnn_layer_1b = TemporalNeurons(
   n=tnn_layer_sz,
   timesteps=num_timesteps,
   threshold=tnn_thresh,
   num_winners=num_winners_tnn
   )
rtnn_layer_1 = TemporalNeurons(
   n=rtnn_layer_sz,
   timesteps=num_timesteps,
   threshold=rtnn_thresh,
   num_winners=num_winners_rtnn
   )

buffer_layer_1 = TemporalBufferNeurons(n=rtnn_layer_sz, timesteps=num_timesteps)
buffer_layer_2 = TemporalBufferNeurons(n=rtnn_layer_sz, timesteps=num_timesteps)

stdp_tnn_params = {
   "ucapture":  10/128,
   "uminus":    10/128,
   "usearch":   2/128,
   "ubackoff":  96/128,
```

```
    "umin":     4/128,
    "maxweight": max_weight
}
stdp_rtnn_params = {
    "ucapture":  15/128,
    "uminus":    15/128,
    "usearch":   10/128,
    "ubackoff":  96/128,
    "umin":     1/128,
    "maxweight": max_weight_rtnn
}

# Feed-forward connections
w_rand_l1 = 0.1 * max_weight * torch.rand(input_layer_a.n, tnn_layer_1a.n)
w_rand_l2 = 0.1 * max_weight * torch.rand(tnn_layer_1a.n, rtnn_layer_1.n)
FF1a = Connection(source=input_layer_a, target=tnn_layer_1a,
        w = w_rand_l1, timesteps = num_timesteps,
    update_rule=TNN_STDP, **stdp_tnn_params)
FF1b = Connection(source=input_layer_b, target=tnn_layer_1b,
    w = w_rand_l1, timesteps = num_timesteps,
    update_rule=TNN_STDP, **stdp_tnn_params )
FF2a = Connection(source=tnn_layer_1a, target=rtnn_layer_1,
    w = w_rand_l2, timesteps = num_timesteps,
    update_rule=TNN_STDP, **stdp_rtnn_params )
FF2b = Connection(source=tnn_layer_1b, target=rtnn_layer_1,
    w = w_rand_l2, timesteps = num_timesteps,
    update_rule=TNN_STDP, **stdp_rtnn_params )

# Recurrent connections
w_eye_rtnn = torch.diag(torch.ones(rtnn_layer_1.n))
rTNN_to_buf1 = Connection(source=rtnn_layer_1, target=buffer_layer_1,
        w = w_eye_rtnn, update_rule=None)
buf1_to_buf2 = Connection(source=buffer_layer_1, target=buffer_layer_2,
    w = w_eye_rtnn, update_rule=None)

buf1_to_rTNN = Connection(
        source=buffer_layer_1,
        target=rtnn_layer_1,
        w = 0.5 * max_weight * torch.rand(rtnn_layer_1.n, rtnn_layer_1.n),
    timesteps = num_timesteps,
    update_rule=TNN_STDP, **stdp_rtnn_params )

buf2_to_rTNN = Connection(
    source=buffer_layer_2,
    target=rtnn_layer_1,
    w = 0.5 * max_weight * torch.rand(rtnn_layer_1.n, rtnn_layer_1.n),
    timesteps = num_timesteps,
    update_rule=TNN_STDP, **stdp_rtnn_params )

# Add all nodes to network:
network.add_layer(input_layer_a, name="I_a")
network.add_layer(input_layer_b, name="I_b")
network.add_layer(tnn_layer_1a, name="TNN_1a")
```

```
network.add_layer(tnn_layer_1b, name="TNN_1b")
network.add_layer(buffer_layer_1, name="BUF_1")
# network.add_layer(buffer_layer_2, name="BUF_2")
network.add_layer(rtnn_layer_1, name="rTNN_1")

# Add connections to network:
# (feedforward)
network.add_connection(FF1a, source="I_a", target="TNN_1a")
network.add_connection(FF1b, source="I_b", target="TNN_1b")
network.add_connection(FF2a, source="TNN_1a", target="rTNN_1")
network.add_connection(FF2b, source="TNN_1b", target="rTNN_1")
# (Recurrences)
network.add_connection(rTNN_to_buf1, source="rTNN_1", target="BUF_1")
# network.add_connection(buf1_to_buf2, source="BUF_1", target="BUF_2")
network.add_connection(buf1_to_rTNN, source="BUF_1", target="rTNN_1")
# network.add_connection(buf2_to_rTNN, source="BUF_2", target="rTNN_1")
```

Results:

```
Pre-Training progress: (1000 / 1000): 1001it [08:53,  1.86it/s]Press enter to continue to plotting...

 Training the read out): 2it [00:17,  7.75s/it]
Epoch: 10/10, Loss: 0.8977: 10it [01:02,  6.29s/it]5:59,  2.81it/s]

 Accuracy of the model on 1000 test images: 12.51 %
--Return--10, Loss: 0.8977: 10it [01:02,  6.24s/it]
> /mnt/c/Users/markp/Repos/18847-rTNN/rc_buff_seq_V4.py(303)<module>()->None
-> pdb.set_trace()
```

Second run:

```
 Accuracy of the model on 1000 test images: 13.07 %
```

V5 (Clustering into Recurrent)

Sunday, April 26, 2020 11:21 AM

Going back to a simpler design:
A clustering column feeding into another column with a recurrent buffer.

```
input_size = 28
input_slice = 28
tnn_layer_sz = 50
rtnn_layer_sz = 100
num_timesteps = 16
tnn_thresh = 32
rtnn_thresh = 32
max_weight = 16
max_weight_rtnn = 32
num_winners_tnn = 2
num_winners_rtnn = 10

time = num_timesteps

torch.manual_seed(seed)

# build network:
network = Network(dt=1)
input_layer_a = Input(n=input_slice)

tnn_layer_1a = TemporalNeurons(
        n=tnn_layer_sz,
        timesteps=num_timesteps,
        threshold=tnn_thresh,
        num_winners=num_winners_tnn
        )
rtnn_layer_1 = TemporalNeurons(
   n=rtnn_layer_sz,
   timesteps=num_timesteps,
   threshold=rtnn_thresh,
   num_winners=num_winners_rtnn
   )

buffer_layer_1 = TemporalBufferNeurons(n=rtnn_layer_sz, timesteps=num_timesteps)
buffer_layer_2 = TemporalBufferNeurons(n=rtnn_layer_sz, timesteps=num_timesteps)

stdp_tnn_params = {
   "ucapture":  20/128,
   "uminus":    20/128,
   "usearch":   2/128,
   "ubackoff":  96/128,
   "umin":      4/128,
   "maxweight": max_weight
}
stdp_rtnn_params = {
   "ucapture":  20/128,
```

```
   "uminus":    20/128,
   "usearch":   30/128,
   "ubackoff":  96/128,
   "umin":      16/128,
   "maxweight": max_weight_rtnn
}

# Feed-forward connections
w_rand_l1 = 0.1 * max_weight * torch.rand(input_layer_a.n, tnn_layer_1a.n)
w_rand_l2 = 0.1 * max_weight * torch.rand(tnn_layer_1a.n, rtnn_layer_1.n)
FF1a = Connection(source=input_layer_a, target=tnn_layer_1a,
        w = w_rand_l1, timesteps = num_timesteps,
   update_rule=TNN_STDP, **stdp_tnn_params)
FF2a = Connection(source=tnn_layer_1a, target=rtnn_layer_1,
   w = w_rand_l2, timesteps = num_timesteps,
   update_rule=TNN_STDP, **stdp_rtnn_params )

# Recurrent connections
w_eye_rtnn = torch.diag(torch.ones(rtnn_layer_1.n))
rTNN_to_buf1 = Connection(source=rtnn_layer_1, target=buffer_layer_1,
        w = w_eye_rtnn, update_rule=None)

# Force recurrent connectivity to be sparse, but strong.
w_recur = max_weight * torch.rand(rtnn_layer_1.n, rtnn_layer_1.n)
w_recur[torch.rand(rtnn_layer_1.n, rtnn_layer_1.n) < 0.90] = 0
buf1_to_rTNN = Connection(
        source=buffer_layer_1,
        target=rtnn_layer_1,
        w = w_recur,
   timesteps = num_timesteps,
   update_rule=None )


# Add all nodes to network:
network.add_layer(input_layer_a, name="I_a")
network.add_layer(tnn_layer_1a, name="TNN_1a")
network.add_layer(buffer_layer_1, name="BUF_1")
network.add_layer(rtnn_layer_1, name="rTNN_1")

# Add connections to network:
# (feedforward)
network.add_connection(FF1a, source="I_a", target="TNN_1a")
network.add_connection(FF2a, source="TNN_1a", target="rTNN_1")
# (Recurrences)
network.add_connection(rTNN_to_buf1, source="rTNN_1", target="BUF_1")
network.add_connection(buf1_to_rTNN, source="BUF_1", target="rTNN_1")
```

Result:

```
 Training the read out10): 11it [02:39, 13.45s/it]
Epoch: 10/10, Loss: 0.8955: 10it [01:07,  6.78s/it]4:41,  3.52it/s]
```

Accuracy of the model on 1000 test images: 13.92 %
--Return--10, Loss: 0.8955: 10it [01:07, 6.81s/it]
> /mnt/c/Users/markp/Repos/18847-rTNN/rc_buff_seq_V5.py(281)<module>()->None
-> pdb.set_trace()

Again:
Training the read out10): 11it [02:29, 13.19s/it]
Epoch: 10/10, Loss: 1.1097: 10it [00:02, 4.35it/s]4:33, 3.62it/s]

Accuracy of the model on 1000 test images: 13.89 %
--Return--10, Loss: 1.1097: 10it [00:02, 4.37it/s]
> /mnt/c/Users/markp/Repos/18847-rTNN/rc_buff_seq_V5.py(281)<module>()->None
-> pdb.set_trace()

V6 (Cheat mode activated)

Sunday, April 26, 2020 1:29 PM

Idea here: collect output spikes over all time waves and use these in the logreg unit. Might be cheating…?

Training the read out10): 11it [02:29, 13.55s/it]
Epoch: 10/10, Loss: 0.1585: 10it [00:02, 3.69it/s]4:39, 3.59it/s]

Accuracy of the model on 1000 test images: 27.07 %
--Return--10, Loss: 0.1585: 10it [00:02, 3.68it/s]

V7,8,9,10 (Cheat mode: saving output at each wave)

Monday, April 27, 2020 11:30 AM

For all tests, 1000 input examples. Log Reg took concatenated outputs from all 28 waves per input sample digit.

V7: remove the clustering column, random recurrent weights

Training the read out10): 11it [02:57, 15.26s/it]
Epoch: 10/10, Loss: 0.2194: 10it [00:03, 3.20it/s]3:38, 4.59it/s]

Accuracy of the model on 1000 test images: 68.63 %
--Return--10, Loss: 0.2194: 10it [00:03, 3.18it/s]

Rerun:
Training the read out10): 11it [03:13, 16.29s/it]
Epoch: 10/10, Loss: 0.2681: 10it [00:16, 1.68s/it]7:06, 3.86it/s]

Accuracy of the model on 4000 test images: 68.73 %
--Return--10, Loss: 0.2681: 10it [00:16, 1.70s/it]
> /mnt/c/Users/markp/Repos/18847-rTNN/rc_buff_seq_V7.py(264)<module>()->None
-> pdb.set_trace()

Rerun with definitely different training and test sets:
Training the read out10): 11it [02:25, 13.86s/it]
Epoch: 10/10, Loss: 0.2746: 10it [00:02, 4.10it/s]:59, 4.27it/s]

Accuracy of the model on 1000 test images: 55.26 %
--Return--10, Loss: 0.2746: 10it [00:02, 4.11it/s]

V8: remove the clustering column, STDP recurrent weights

Training the read out10): 11it [03:02, 15.64s/it]
Epoch: 10/10, Loss: 0.1103: 10it [00:03, 3.05it/s]4:20, 3.90it/s]

Accuracy of the model on 1000 test images: 67.03 %
--Return--10, Loss: 0.1103: 10it [00:03, 3.01it/s]

Rerun:
Train progress: (10 / 10): 10it [03:10, 17.81s/it]
Train progress: (10 / 10): 11it [07:21, 87.63s/it]
Training the read outess: (0 / 4000): 0it [00:00, ?it/s]
Epoch: 10/10, Loss: 0.1287: 10it [00:14, 1.49s/it]9:01, 3.53it/s]

Accuracy of the model on 4000 test images: 74.53 %
--Return--10, Loss: 0.1287: 10it [00:14, 1.48s/it]
> /mnt/c/Users/markp/Repos/18847-rTNN/rc_buff_seq_V8.py(264)<module>()->None
-> pdb.set_trace()

Training the read out10): 11it [04:23, 16.63s/it]
Epoch: 10/10, Loss: 0.1207: 10it [00:13, 1.40s/it]6:15, 4.11it/s]

Accuracy of the model on 4000 test images: 75.31 %
--Return--10, Loss: 0.1207: 10it [00:13, 1.39s/it]
> /mnt/c/Users/markp/Repos/18847-rTNN/rc_buff_seq_V8.py(264)<module>()->None
-> pdb.set_trace()

Note: weights are not training down to 0 in the recurrent layer. See image below for 4000 examples trained.

Rerun with definitely different training and test sets:
Training the read out10): 11it [02:24, 13.64s/it]
Epoch: 10/10, Loss: 0.0899: 10it [00:02, 3.75it/s]:23, 3.45it/s]

Accuracy of the model on 1000 test images: 67.27 %
--Return--10, Loss: 0.0899: 10it [00:02, 3.80it/s]

V9: remove the clustering column, random recurrent weights, poisson encoding.
Training the read out10): 11it [02:32, 14.15s/it]
Epoch: 10/10, Loss: 0.2866: 10it [00:03, 2.99it/s]3:52, 4.37it/s]

Accuracy of the model on 1000 test images: 33.47 %
--Return--10, Loss: 0.2866: 10it [00:03, 2.99it/s]

retest with 4000 inputs:
❯ python3.6 rc_buff_seq_V9.py --examples 4000

Pre-Training progress: (4000 / 4000): 4001it [21:00, 3.31it/s]
Training the read outess: (4000 / 4000): 4001it [16:26, 4.38it/s]
Epoch: 10/10, Loss: 0.3802: 10it [00:16, 1.62s/it]

Accuracy of the model on 4000 test images: 43.61 %
--Return--sting progress: (4000 / 4000): 4001it [16:39, 4.29it/s]
> /mnt/c/Users/markp/Repos/18847-rTNN/rc_buff_seq_V9.py(264)<module>()->None
-> pdb.set_trace()

Rerun with definitely different training and test sets:
Training the read out10): 11it [02:14, 13.13s/it]
Epoch: 10/10, Loss: 0.2854: 10it [00:02, 3.77it/s]:42, 4.55it/s]

Accuracy of the model on 1000 test images: 32.03 %
--Return--10, Loss: 0.2854: 10it [00:02, 3.77it/s]

V10: same as V9, but with STDP on recurrent weights.
Note: recurrent weights trained to nothing.
Training the read out10): 11it [02:56, 14.48s/it]
Epoch: 10/10, Loss: 0.2432: 10it [00:02, 3.61it/s]3:22, 5.02it/s]

Accuracy of the model on 1000 test images: 40.96 %
--Return--10, Loss: 0.2432: 10it [00:02, 3.61it/s]

Rerun with 4000 inputs:
❯ python3.6 rc_buff_seq_V10.py --examples 4000

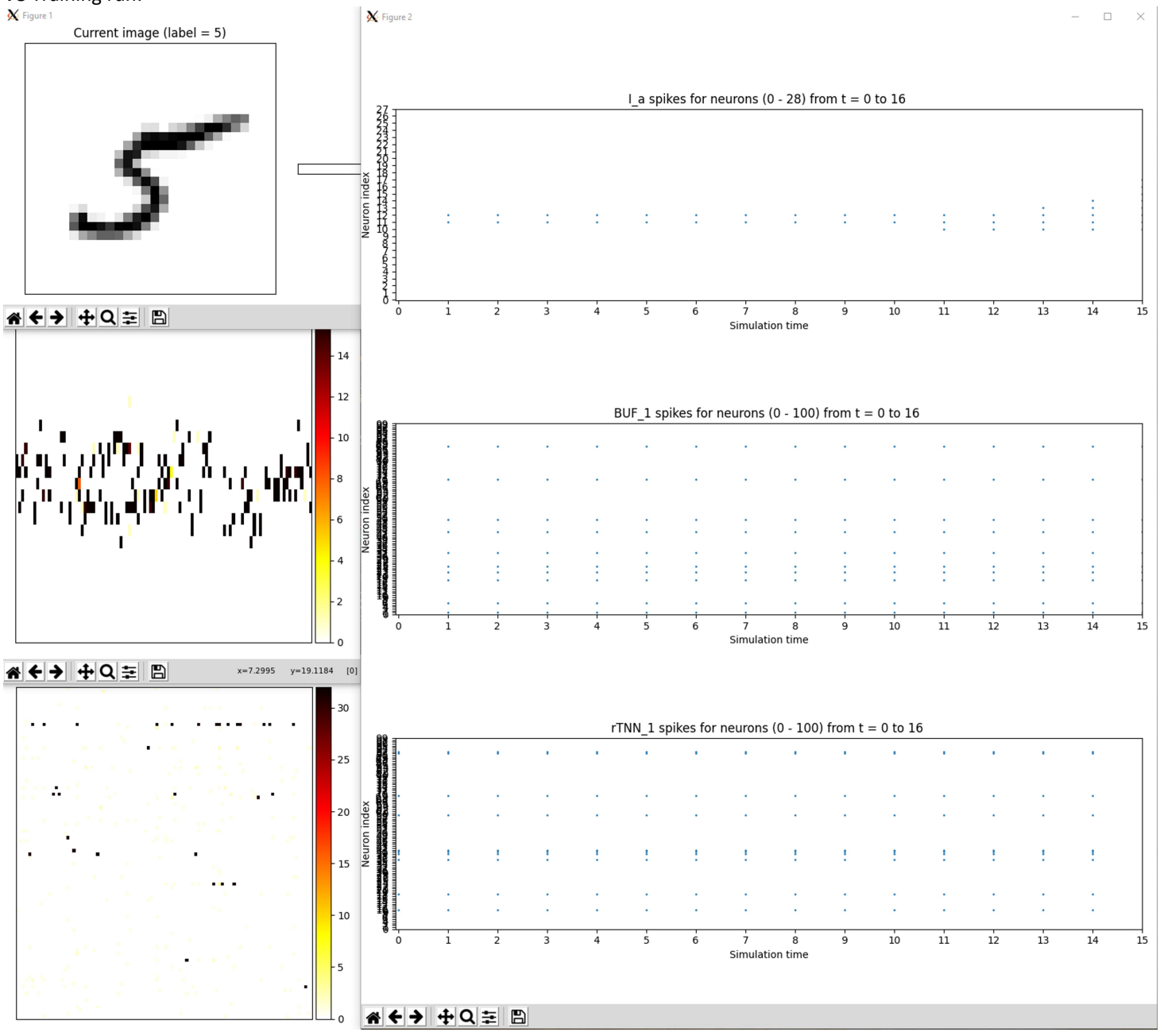
Pre-Training progress: (4000 / 4000): 4001it [28:24, 2.22it/s]
Training the read outess: (4000 / 4000): 4001it [17:16, 3.75it/s]
Epoch: 10/10, Loss: 0.3713: 10it [00:16, 1.69s/it]

Accuracy of the model on 4000 test images: 48.34 %
--Return--sting progress: (4000 / 4000): 4001it [17:18, 3.84it/s]
> /mnt/c/Users/markp/Repos/18847-rTNN/rc_buff_seq_V10.py(264)<module>()->None
-> pdb.set_trace()

Rerun with definitely different training and test sets:
Training the read out10): 11it [02:24, 13.83s/it]
Epoch: 10/10, Loss: 0.2124: 10it [00:02, 3.86it/s]:03, 4.18it/s]

Accuracy of the model on 1000 test images: 43.44 %
--Return--10, Loss: 0.2124: 10it [00:02, 3.85it/s]

Images:
V8 Training run:



V11, 12, 13, 14 (Encoding and recurrence, w/ and w/out STDP)

Monday, April 27, 2020 11:51 AM

Same as V7-10, but with only the last wave saved for log reg. I want to see if it's the removal of the clustering layer or if it is saving spikes from all waves that made things work in that last set.

V11:

Training the read out10): 11it [02:17, 13.38s/it]
Epoch: 10/10, Loss: 1.0842: 10it [00:02, 4.24it/s]:59, 4.61it/s]

Accuracy of the model on 1000 test images: 21.42 %
--Return--10, Loss: 1.0842: 10it [00:02, 4.27it/s]

V12:

Training the read out10): 11it [02:20, 13.65s/it]
Epoch: 10/10, Loss: 1.0781: 10it [00:02, 4.36it/s]:11, 4.60it/s]

Accuracy of the model on 1000 test images: 19.82 %
--Return--10, Loss: 1.0781: 10it [00:02, 4.34it/s]

V13:

Training the read out10): 11it [02:04, 10.87s/it]
Epoch: 10/10, Loss: 1.1278: 10it [00:02, 4.35it/s]:26, 4.37it/s]

Accuracy of the model on 1000 test images: 12.81 %
--Return--10, Loss: 1.1278: 10it [00:02, 4.38it/s]

V14:

Training the read out10): 11it [02:22, 13.58s/it]
Epoch: 10/10, Loss: 1.1337: 10it [00:02, 4.31it/s]:00, 4.21it/s]

Accuracy of the model on 1000 test images: 12.01 %
--Return--10, Loss: 1.1337: 10it [00:02, 4.17it/s]

One more test:
How well does log reg with the flattened image do?

Result with 100 epochs:

❯ python3.6 log_reg_only.py --examples 1000
Training the read out
Epoch: 100/100, Loss: 0.1821: 100it [00:00, 310.21it/s]

Accuracy of the model on 1000 test images: 22.22 %
❯ python3.6 log_reg_only.py --examples 4000

Training the read out

Epoch: 100/100, Loss: 0.0914: 100it [00:00, 322.11it/s]

Accuracy of the model on 4000 test images: 11.11 %

With only 10 epochs:

❯ python3.6 log_reg_only.py --examples 1000

Training the read out
Epoch: 10/10, Loss: 0.1830: 10it [00:00, 302.37it/s]

Accuracy of the model on 1000 test images: 11.11 %

❯ python3.6 log_reg_only.py --examples 4000

Training the read out
Epoch: 10/10, Loss: 0.2737: 10it [00:00, 279.51it/s]

Accuracy of the model on 4000 test images: 55.56 %

V15 (More buffers, early version)

Tuesday, April 28, 2020 4:02 PM

STDP weights, 2 buffers, 100 neurons, saving just spikes from last wave.

Training the read out10): 11it [03:16, 16.38s/it]
Epoch: 10/10, Loss: 1.0953: 10it [00:02, 3.87it/s]:15, 3.80it/s]

Accuracy of the model on 1000 test images: 20.22 %
--Return--10, Loss: 1.0953: 10it [00:02, 3.89it/s]

With another buffer layer:
Pre-Training progress: (5000 / 5000): 5001it [1:01:33, 1.45it/s]Press enter to continue to plotting...
Training the read out10): 11it [04:09, 23.92s/it]
Epoch: 10/10, Loss: 0.8674: 10it [00:15, 1.55s/it]1:42, 2.64it/s]

Accuracy of the model on 5000 test images: 16.22 %
--Return--10, Loss: 0.8674: 10it [00:15, 1.54s/it]