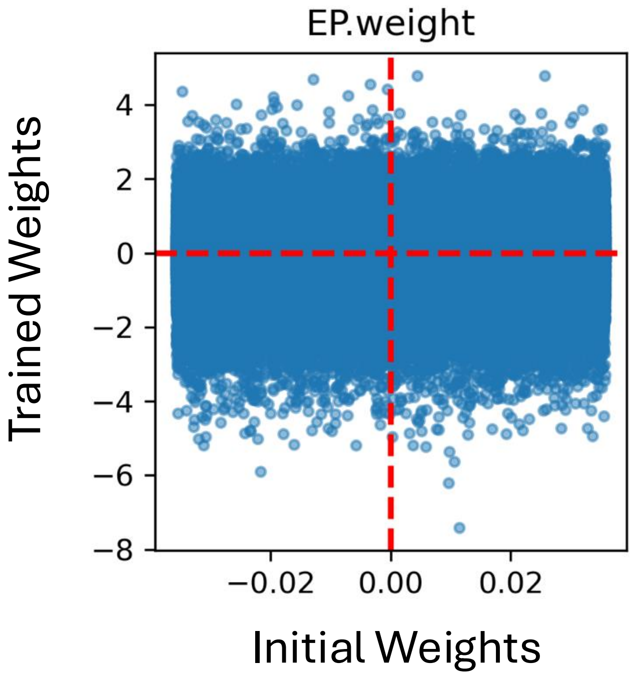
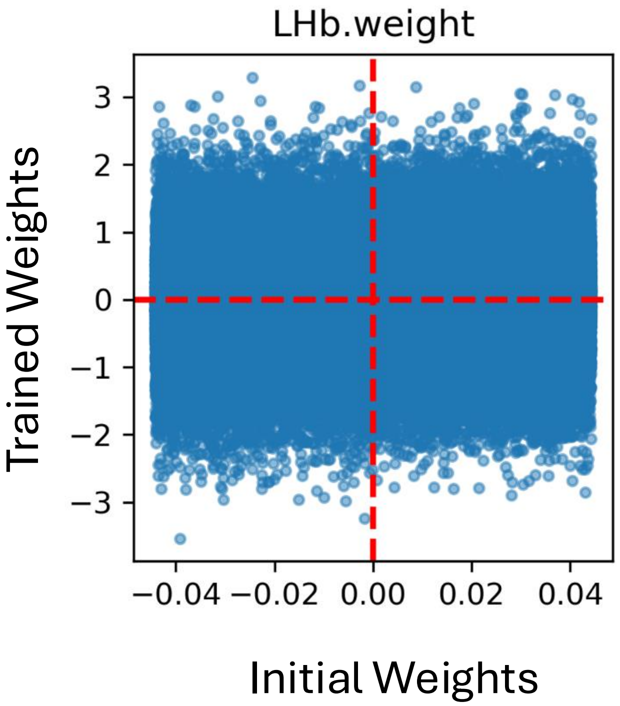
Is there a functional advantage for co-releasing in learning?

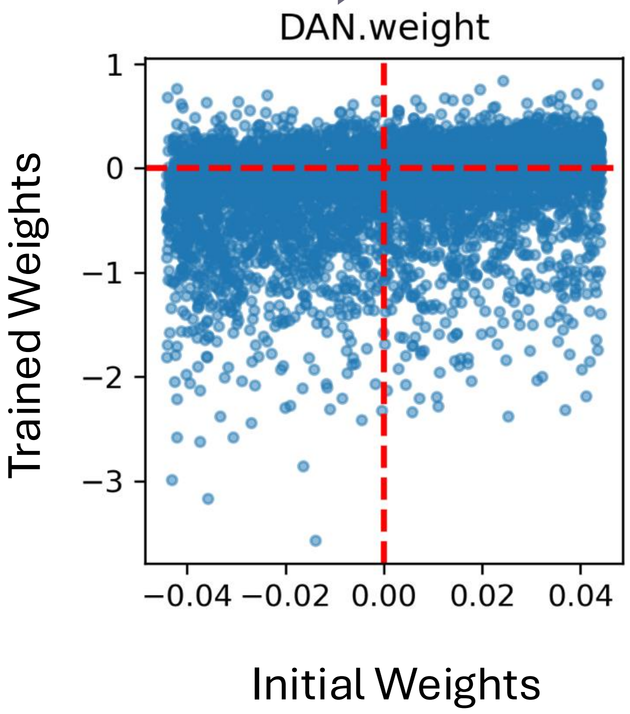
Elliot Jerng
Sabatini Lab Meeting

Standard MLP: Control

Input:
784 Neurons
Ctx & Str → Hidden Layer 1:
512 Neurons
EP → Hidden Layer 2:
512 Neurons
LHb → Output Layer:
10 features: fMNIST dataset
DAN







Standard MLP Code

Constructor init function

```
def __init__(self, in_features=784, h1=512, h2=512, out_features=10, dropout_rate=0.5, real = False, combine_EI = False, dales_law = False):
    super().__init__()
    self.real = real
    self.dales_law = dales_law
    # create layers
    self.EP = nn.Linear(in_features, h1)
    self.bn1 = nn.BatchNorm1d(h1)
    self.LHb = nn.Linear(h1, h2)
    self.bn2 = nn.BatchNorm1d(h2)
    self.DAN = nn.Linear(h2, out_features)
```

Forward function

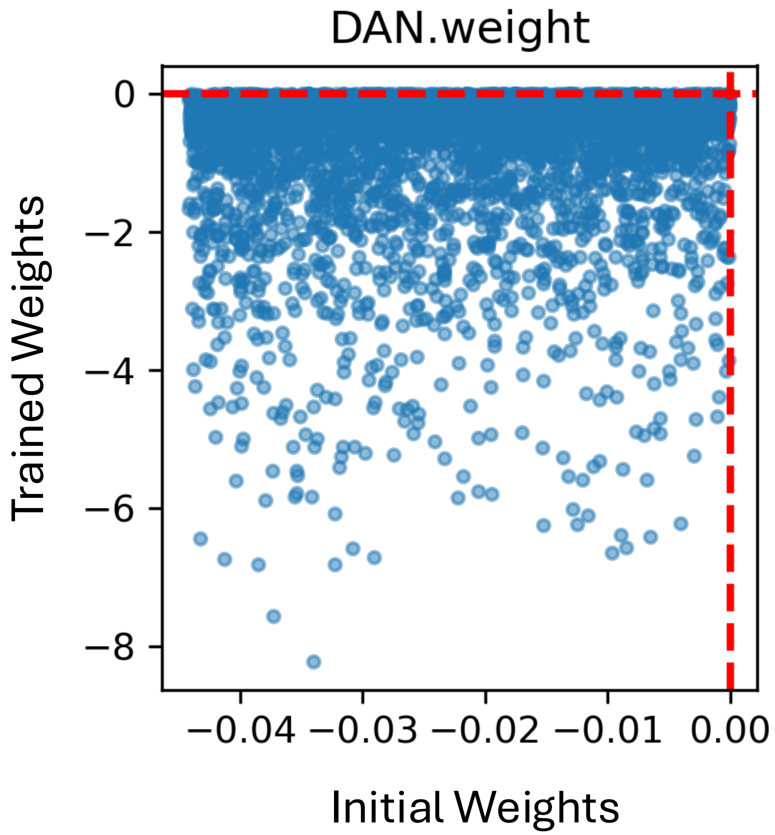
```
def forward(self, x):
    x = x.view(x.size(0), -1)
    x = F.relu(self.bn1(self.EP(x)))
    x = F.relu(self.bn2(self.LHb(x)))

    # pure negative -> DAN
    if self.real == True:
        x = -torch.abs(x)
    x = self.DAN(x)

    return x
```

True LHb ⊣ DAN

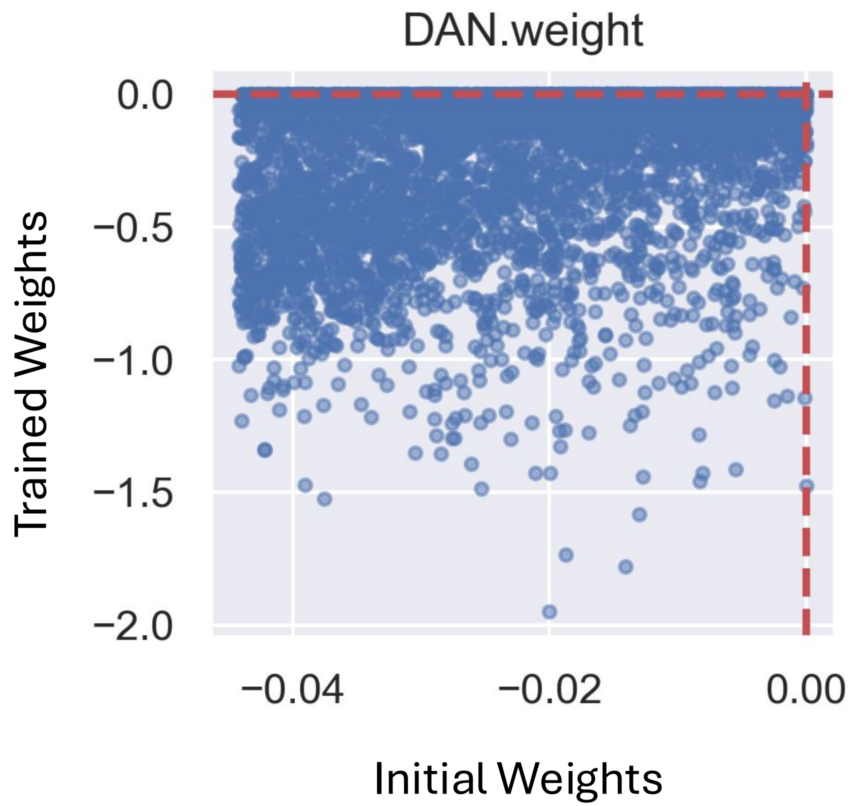
Hidden Layer 2:
512 Neurons

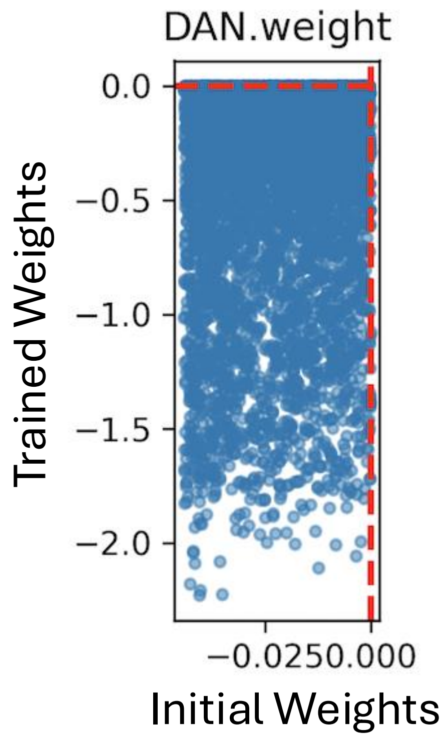
Output Layer:
10 features

Ctx & Str → EP → LHb ⊣ DAN



True LHb ⊣ DAN





Shun Experiment Model

LHb ⊣ DAN

Canonical Brain Circuit V2/V3: Split Stream

LHb ⊣ DAN

LHb ⊣ DAN

Shun Experiment Model Code: True LHb-DAN

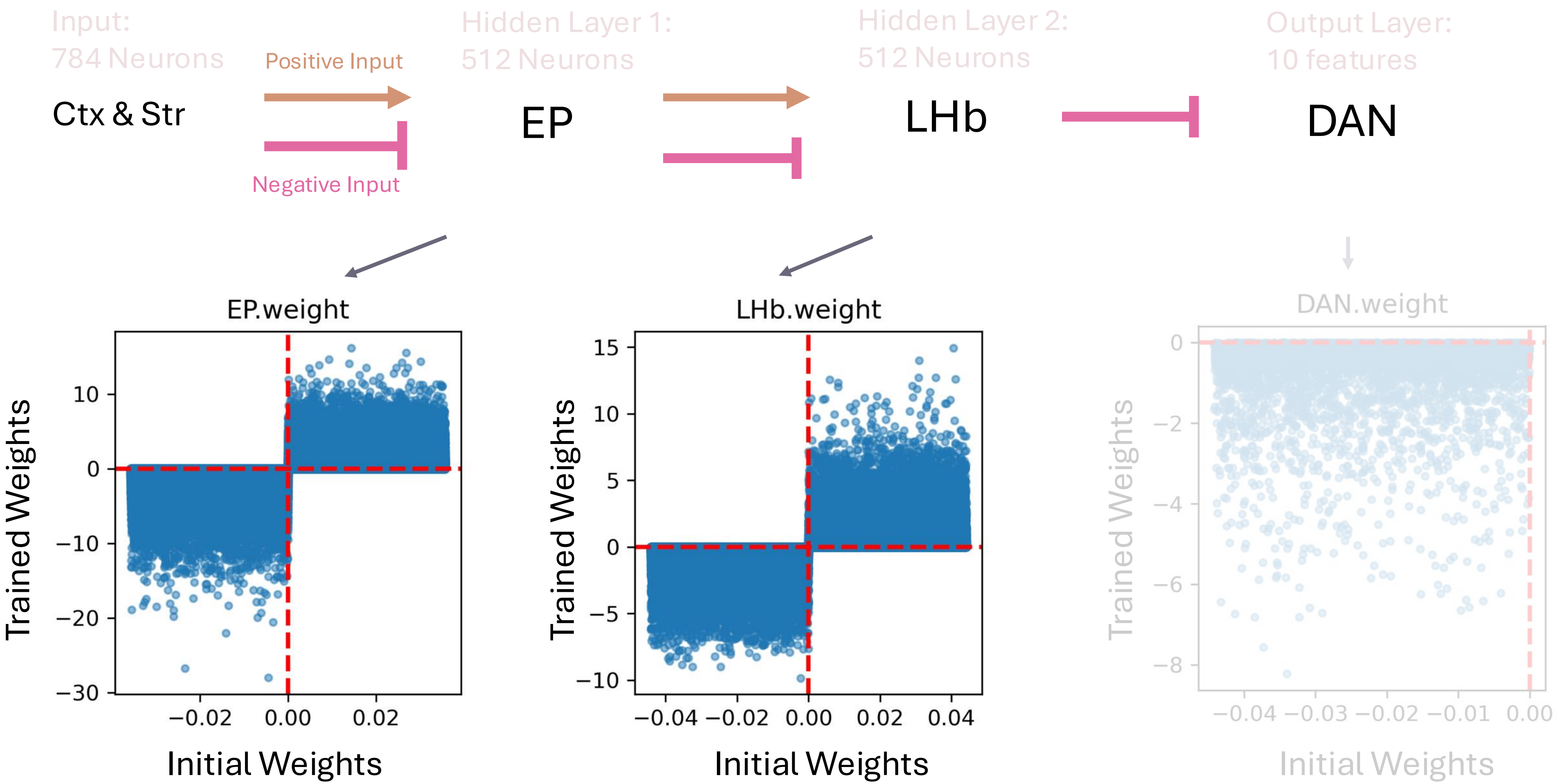
Constructor init function

```
def __init__(self, in_features=784, h1=512, h2=512, out_features=10, dropout_rate=0.5, real = False, combine_EI = False, dales_law = False):
    super().__init__()
    self.real = real
    self.dales_law = dales_law
    # create layers
    self.EP = nn.Linear(in_features, h1)
    self.bn1 = nn.BatchNorm1d(h1)
    self.LHb = nn.Linear(h1, h2)
    self.bn2 = nn.BatchNorm1d(h2)
    self.DAN = nn.Linear(h2, out_features)

    # initialize DAN as purely inhibitory
    if self.real == True:
        self.apply(self.absolute_val)
```

Forward function

```
def forward(self, x):
    x = x.view(x.size(0), -1)
    x = F.relu(self.bn1(self.EP(x)))
    x = F.relu(self.bn2(self.LHb(x)))

    # pure negative -> DAN
    if self.real == True:
        x = -torch.abs(x)
    x = self.DAN(x)

    return x
```

Fixed-Sign Excitatory and Inhibitory: Combined Stream

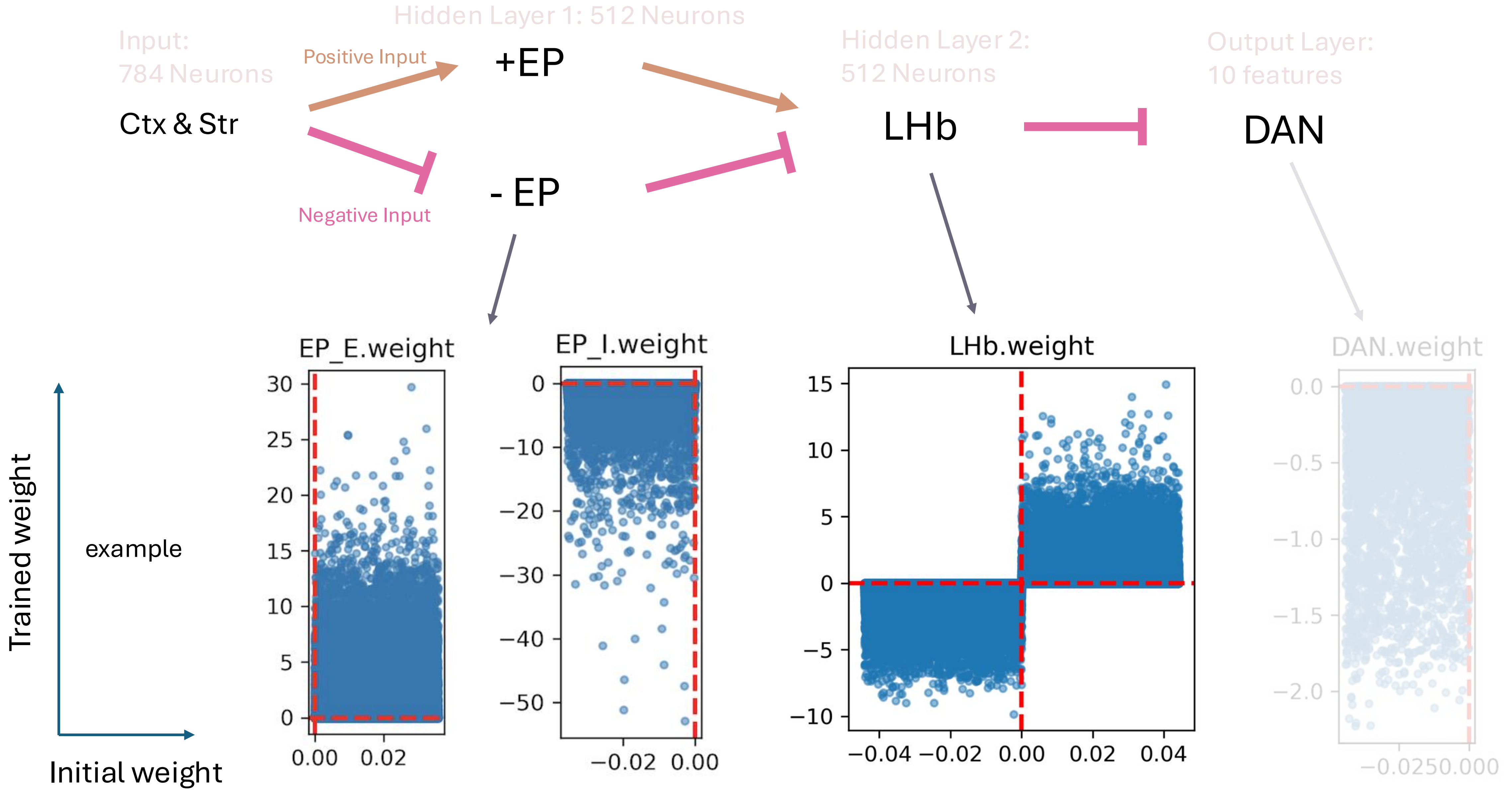


Fixed-Sign: Combined Stream

```python
    def __init__(self, in_features=784, h1=512, h2=512, out_features=10, dropout_rate=0.5, real = False, combine_EI = False, dales_law = False):
        super().__init__()
        self.real = real
        self.dales_law = dales_law
        # create layers
        self.EP = nn.Linear(in_features, h1)
        self.bn1 = nn.BatchNorm1d(h1)
        self.LHb = nn.Linear(h1, h2)
        self.bn2 = nn.BatchNorm1d(h2)
        self.DAN = nn.Linear(h2, out_features)

        # initialize DAN as purely inhibitory
        if self.real == True:
            self.apply(self.absolute_val)

        # combined EI/ dale's law
        EP_LHb_DAN_pos_neurons, EP_LHb_DAN_neg_neurons = {}, {}
        DAN_pos_neurons, DAN_neg_neurons = {}, {}


        # neurons will only project pure excitatory/inhibitory
        with torch.no_grad():
            for name, param in self.named_parameters():
                if combine_EI == True:
                    print(combine_EI)
                    if "weight" in name:
                        # categorize neurons as excitatory/inhibitory
                        EP_LHb_DAN_pos_neurons[name] = torch.sum(param.data, axis = 0) >= 0
                        EP_LHb_DAN_neg_neurons[name] = torch.sum(param.data, axis = 0) < 0

                        # make neuron all excitatory/inhibitory
                        param.data[:, EP_LHb_DAN_pos_neurons[name]] = torch.sign(param[:, EP_LHb_DAN_pos_neurons[name]]) * param[:, EP_LHb_DAN_pos_neurons[name]]
                        param.data[:, EP_LHb_DAN_neg_neurons[name]] = -torch.sign(param[:, EP_LHb_DAN_neg_neurons[name]]) * param[:, EP_LHb_DAN_neg_neurons[name]]
                elif self.real == True:
                    if "DAN.weight" in name:
                        DAN_pos_neurons[name] = torch.sum(param.data, axis = 0) >= 0
                        DAN_neg_neurons[name] = -torch.sum(param.data, axis = 0) < 0

                        # make neuron all excitatory/inhibitory
                        param.data[:, DAN_pos_neurons[name]] = torch.sign(param[:, DAN_pos_neurons[name]]) * param[:, DAN_pos_neurons[name]]
                        param.data[:, DAN_neg_neurons[name]] = -torch.sign(param[:, DAN_neg_neurons[name]]) * param[:, DAN_neg_neurons[name]]


        # keep track of weights
        self.EP_LHb_DAN_pos_neurons = EP_LHb_DAN_pos_neurons
        self.EP_LHb_DAN_neg_neurons = EP_LHb_DAN_neg_neurons

        self.DAN_pos_neurons = DAN_pos_neurons
        self.DAN_neg_neurons = DAN_neg_neurons
```

```python
def forward(self, x):
    x = x.view(x.size(0), -1)
    x = F.relu(self.bn1(self.EP(x)))
    x = F.relu(self.bn2(self.LHb(x)))

    # pure negative -> DAN
    if self.real == True:
        x = -torch.abs(x)
    x = self.DAN(x)

    return x
```

Fixed-Sign Excitatory and Inhibitory: Split Stream



Shun Experiment Model Code: Split Stream EP only

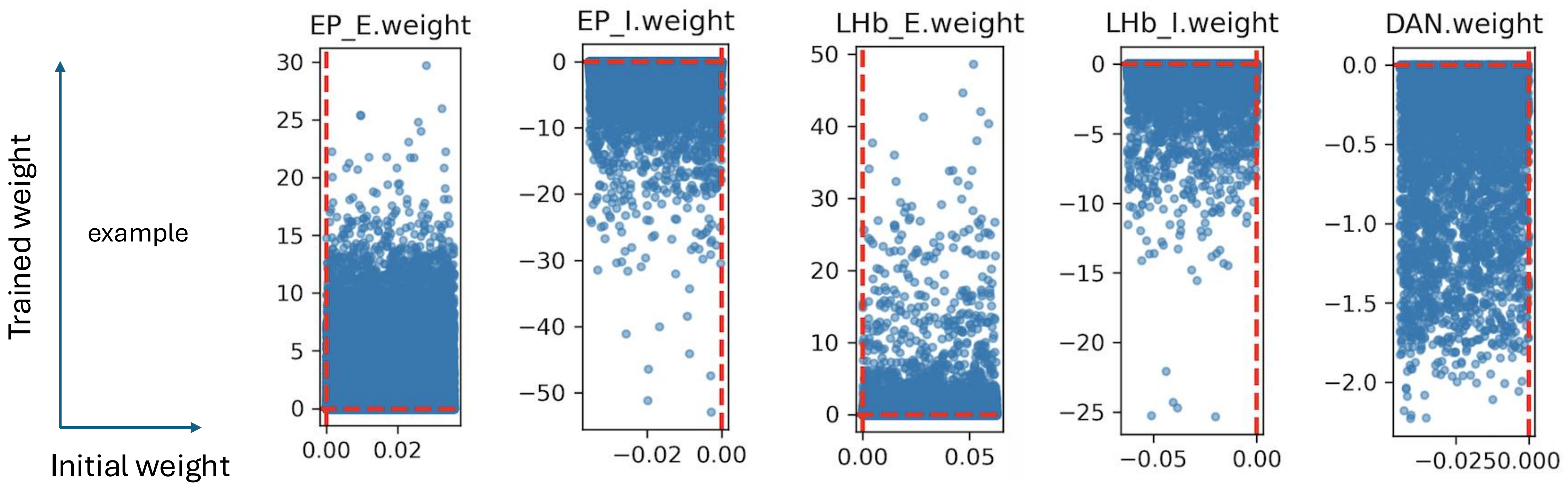
Constructor init function

```
def __init__(self, in_features=784, h1=512, h2=512, out_features=10, dropout_rate=0.5):
    super().__init__()

    # 50% E and I
    num_excitatory_h1 = int(0.5 * h1)
    num_inhibitory_h1 = h1 - num_excitatory_h1

    num_excitatory_h2 = int(0.5 * h2)
    num_inhibitory_h2 = h2 - num_excitatory_h2

    # Create layers
    self.EP_E = nn.Linear(in_features, h1)
    self.EP_I = nn.Linear(in_features, h1)
    self.bn1_E = nn.BatchNorm1d(h1)
    self.bn1_I = nn.BatchNorm1d(h1)
    self.LHb = nn.Linear(h1, h2)
    self.bn2 = nn.BatchNorm1d(h2)
    self.DAN = nn.Linear(h2, out_features)

    # initialize EP_E, EP_I, DAN as strictly E or I
    self.apply(self.absolute_val)

    # keep track of weights
    self.init_weights = self.record_params(calc_sign=False)
```

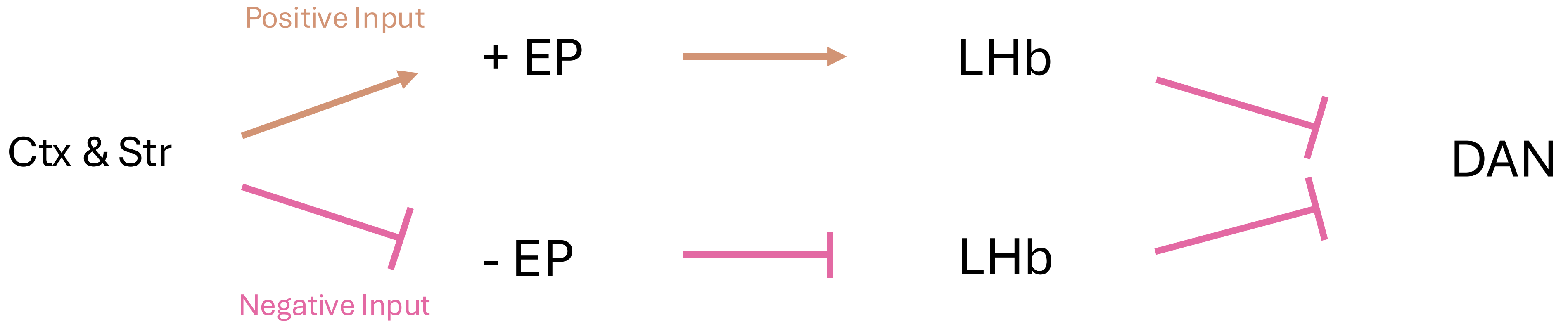
Forward function

```
def forward(self, x):
    x = x.view(x.size(0), -1)

    # EP
    x_e = F.relu(self.bn1_E(self.EP_E(x)))
    x_i = F.relu(self.bn1_I(self.EP_I(x)))

    # Converge into LHb
    x = x_e + x_i
    x = F.relu(self.bn2(self.LHb(x)))

    # Pure Negative LHB to DAN
    x = -torch.abs(x)
    x = self.DAN(x)

    return x
```

Fixed-Sign Excitatory and Inhibitory: Split Stream (EP and LHb)



Canonical Brain Circuit V2: Split Stream



Shun Experiment Model Code: Split EP and LHb

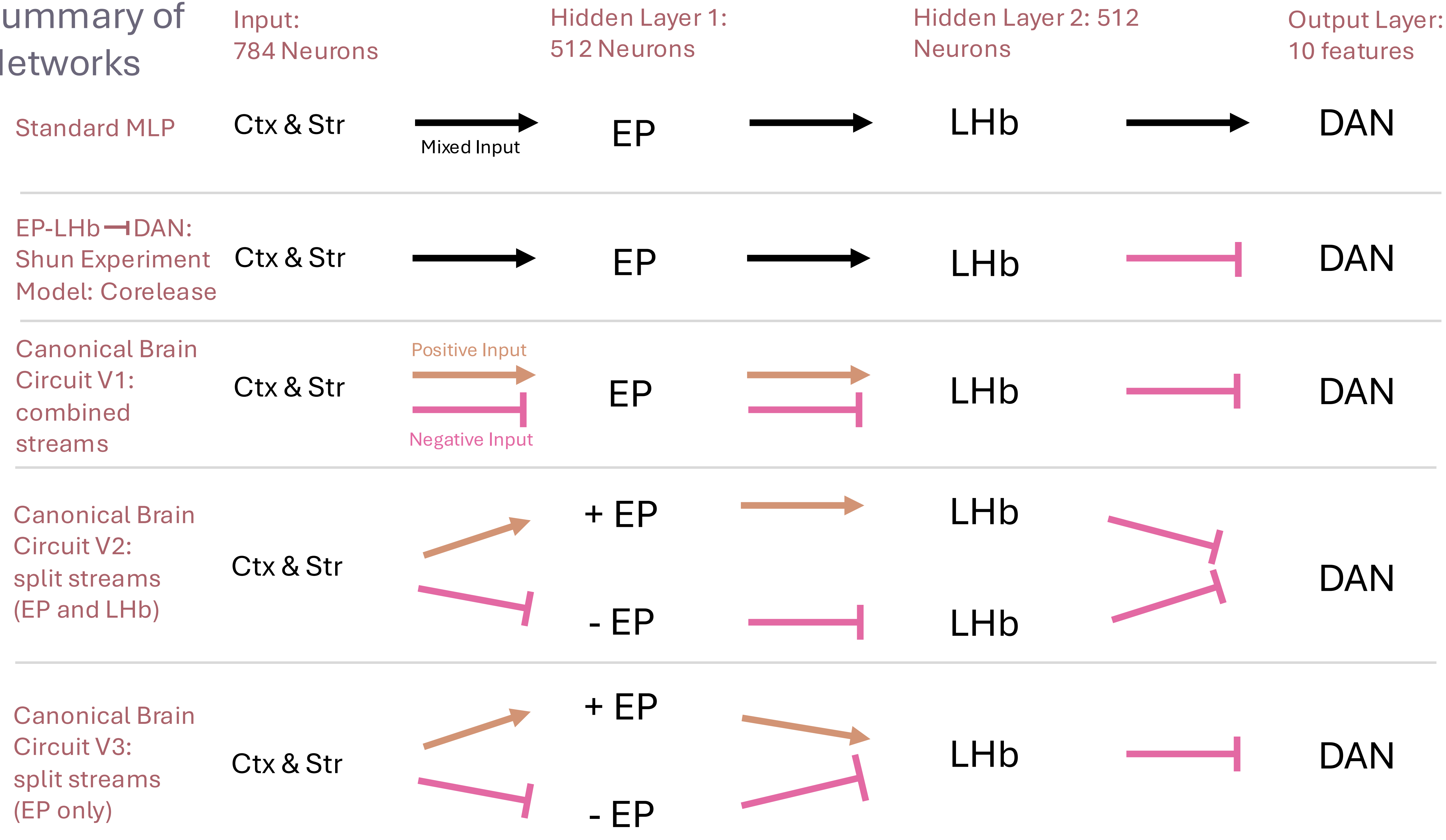
Constructor init function

```
def __init__(self, in_features=784, h1=512, h2=512, out_features=10, dropout_rate=0.5):
    super().__init__()

    # 50% E and I
    num_excitatory_h1 = int(0.5 * h1)
    num_inhibitory_h1 = h1 - num_excitatory_h1

    # Create layers
    self.EP_E = nn.Linear(in_features, num_excitatory_h1)
    self.EP_I = nn.Linear(in_features, num_inhibitory_h1)
    self.bn1_E = nn.BatchNorm1d(num_excitatory_h1)
    self.bn1_I = nn.BatchNorm1d(num_inhibitory_h1)
    self.LHb_E = nn.Linear(num_excitatory_h1, h2)
    self.LHb_I = nn.Linear(num_inhibitory_h1, h2)
    self.bn2_E = nn.BatchNorm1d(h2)
    self.bn2_I = nn.BatchNorm1d(h2)
    self.DAN = nn.Linear(h2, out_features)

    # initialize EP_E, EP_I, LHb_E, LHb_I, DAN as E or I
    self.apply(self.absolute_val)

    # keep track of weights
    self.init_weights = self.record_params(calc_sign=False)
```

Forward function

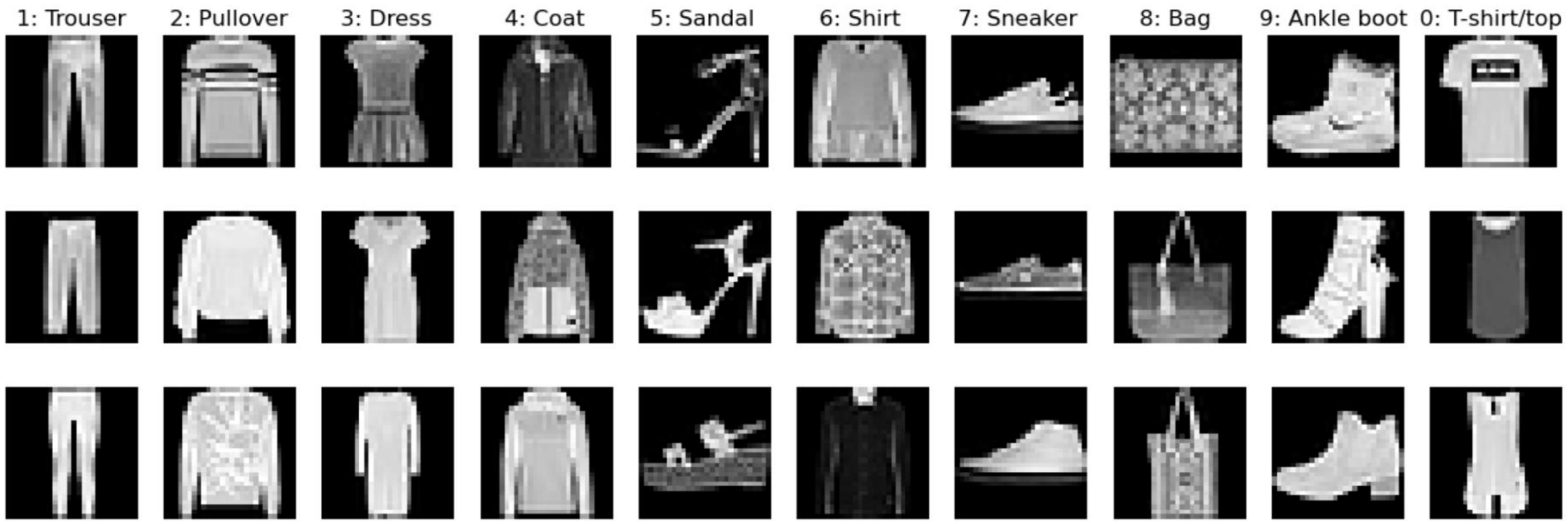
```
def forward(self, x):
    x = x.view(x.size(0), -1)

    # EP
    x_e = F.relu(self.bn1_E(self.EP_E(x)))
    x_i = F.relu(self.bn1_I(self.EP_I(x)))

    # LHb
    x_e = F.relu(self.bn2_E(self.LHb_E(x_e)))
    x_i = F.relu(self.bn2_I(self.LHb_I(x_i)))

    # converge to DAN
    x = x_e + x_i

    # Pure Negative LHB to DAN
    x = -torch.abs(x)

    x = self.DAN(x)

    return x
```

Summary of Networks

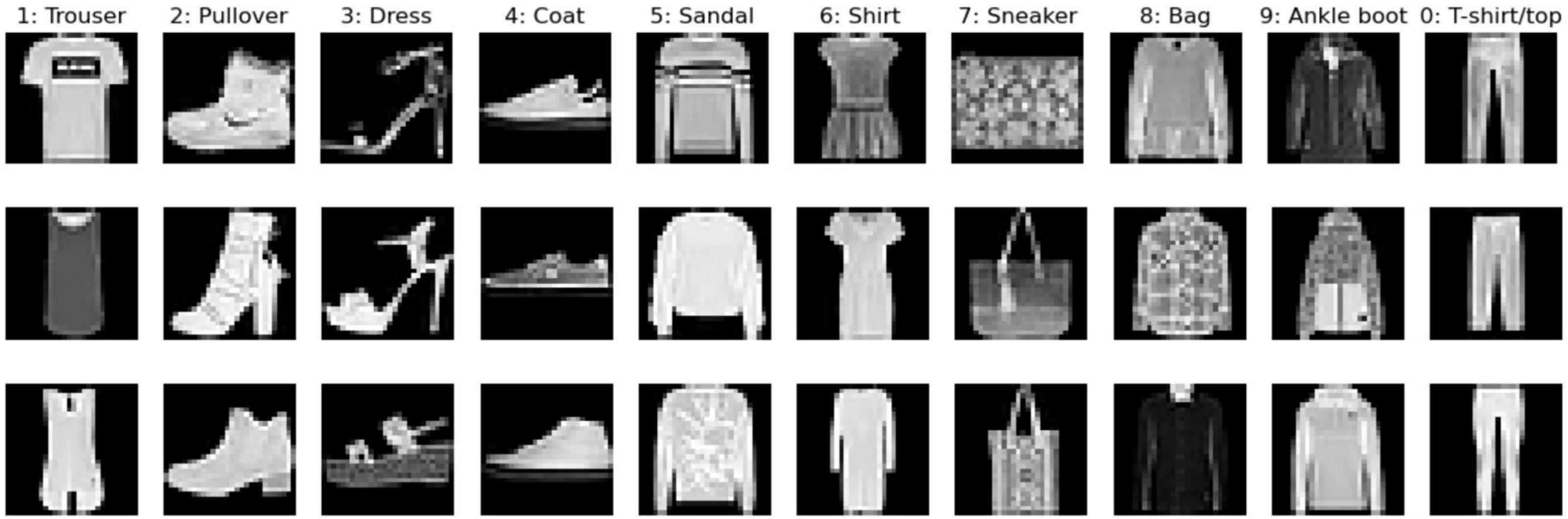


Task for the network: F-MNIST

Part 1:
original fMNIST



Part 2:
shuffled fMNIST



Average Test Accuracy