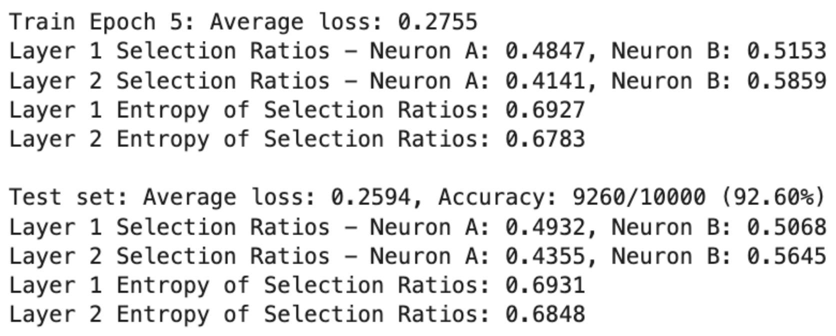
Folder : /workspace/fun/multi_neuron_models

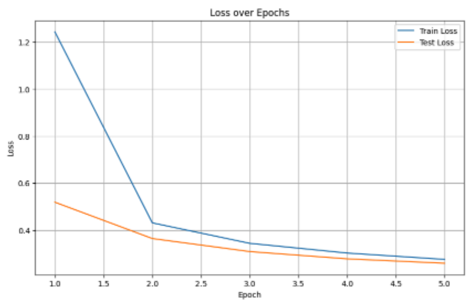
Working on methods to increase dimensionality of the computation graph of NN without changing the fundamental architecture.

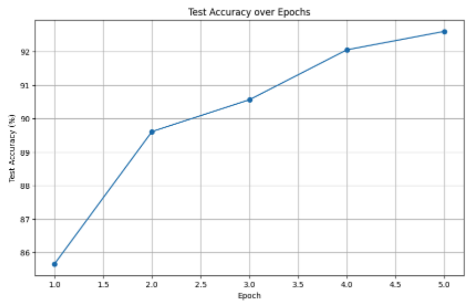
Training on Mnist I found that not only does the model train and with high accuracy >90% (92% after 5 epochs)

But also that the selection neuron is effectively using both computation neurons at the same time.

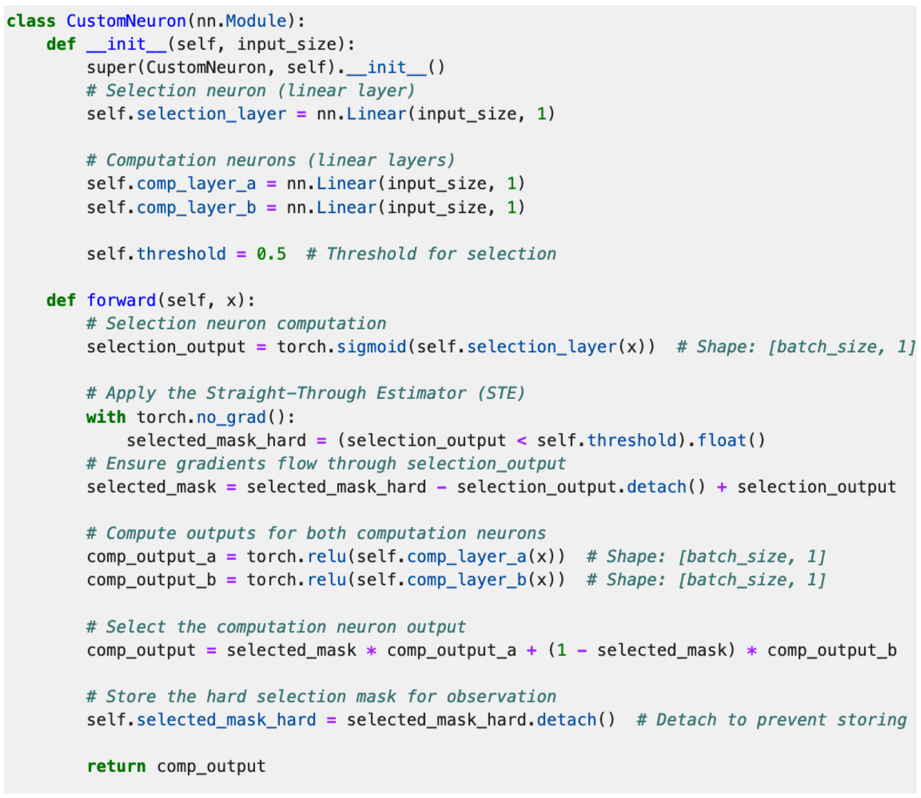


```
Train Epoch 5: Average loss: 0.2755
Layer 1 Selection Ratios - Neuron A: 0.4847, Neuron B: 0.5153
Layer 2 Selection Ratios - Neuron A: 0.4141, Neuron B: 0.5859
Layer 1 Entropy of Selection Ratios: 0.6927
Layer 2 Entropy of Selection Ratios: 0.6783

Test set: Average loss: 0.2594, Accuracy: 9260/10000 (92.60%)
Layer 1 Selection Ratios - Neuron A: 0.4932, Neuron B: 0.5068
Layer 2 Selection Ratios - Neuron A: 0.4355, Neuron B: 0.5645
Layer 1 Entropy of Selection Ratios: 0.6931
Layer 2 Entropy of Selection Ratios: 0.6848
```

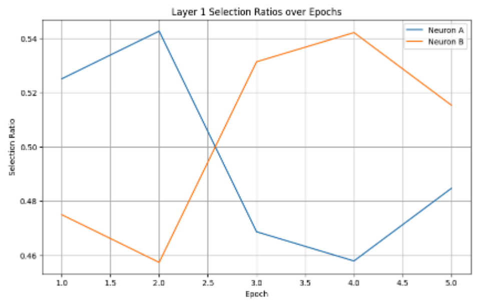


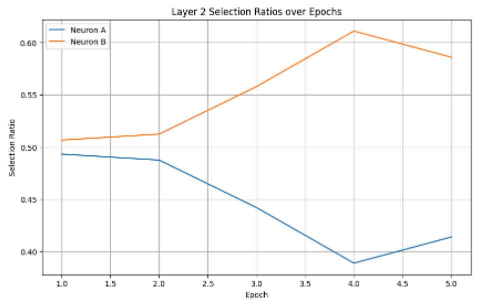


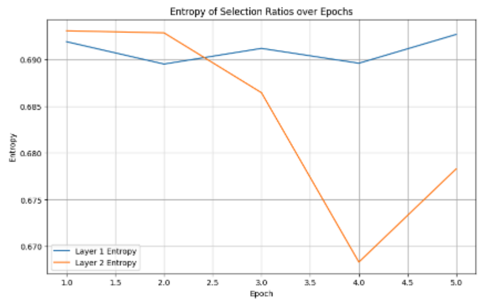
These were done with two neuron sets A and B in a selector
And computation neuron setup.



```
class CustomNeuron(nn.Module):
    def __init__(self, input_size):
        super(CustomNeuron, self).__init__()
        # Selection neuron (linear layer)
        self.selection_layer = nn.Linear(input_size, 1)

        # Computation neurons (linear layers)
        self.comp_layer_a = nn.Linear(input_size, 1)
        self.comp_layer_b = nn.Linear(input_size, 1)

        self.threshold = 0.5  # Threshold for selection

    def forward(self, x):
        # Selection neuron computation
        selection_output = torch.sigmoid(self.selection_layer(x))  # Shape: [batch_size, 1]

        # Apply the Straight-Through Estimator (STE)
        with torch.no_grad():
            selected_mask_hard = (selection_output < self.threshold).float()
        # Ensure gradients flow through selection_output
        selected_mask = selected_mask_hard - selection_output.detach() + selection_output

        # Compute outputs for both computation neurons
        comp_output_a = torch.relu(self.comp_layer_a(x))  # Shape: [batch_size, 1]
        comp_output_b = torch.relu(self.comp_layer_b(x))  # Shape: [batch_size, 1]

        # Select the computation neuron output
        comp_output = selected_mask * comp_output_a + (1 - selected_mask) * comp_output_b

        # Store the hard selection mask for observation
        self.selected_mask_hard = selected_mask_hard.detach()  # Detach to prevent storing

        return comp_output
```







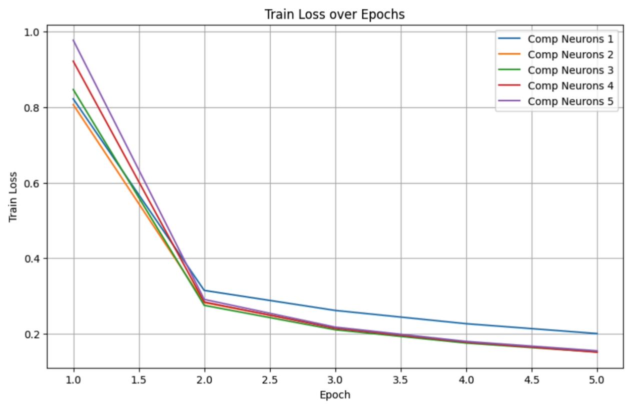
Overall it is seemingly working very straightforward without any hiccup:

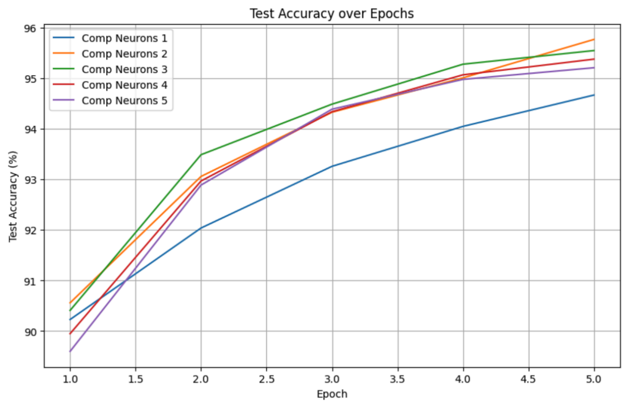
- The higher dimensions of these extra neurons seem to simply act as a regulator of the model not really an issue for stability which is odd.
- This test above and info below are on mnist on a simple model so these results may not translate to larger models.
- Before switching to a more complex scenario (transformers) I am looking at the cosine similarity of the two computational parameter sets.
- I need to test growing or changing the number of computation neurons over time to see the effect
- Need to see the effect in the same CL scenario as done before with EWC

Cosine similarity (the graphing broke) repeat of the above just with cosine monitoring
- Mild rise in cosine similarity over the 5 epochs but nothing major (0.1 -> 0.2) 93% accuracy

```
Train Epoch 1: Average loss: 1.3620
Test set: Average loss: 0.5612, Accuracy: 8458/10000 (84.58%)
Layer 1 Cosine Similarity between Neuron A and B: 0.1199
Layer 2 Cosine Similarity between Neuron A and B: 0.1024

Train Epoch 2: Average loss: 0.4422
Test set: Average loss: 0.3588, Accuracy: 9007/10000 (90.07%)
Layer 1 Cosine Similarity between Neuron A and B: 0.1649
Layer 2 Cosine Similarity between Neuron A and B: 0.1469

Train Epoch 3: Average loss: 0.3377
Test set: Average loss: 0.3097, Accuracy: 9108/10000 (91.08%)
Layer 1 Cosine Similarity between Neuron A and B: 0.1879
Layer 2 Cosine Similarity between Neuron A and B: 0.1668

Train Epoch 4: Average loss: 0.3006
Test set: Average loss: 0.2814, Accuracy: 9186/10000 (91.86%)
Layer 1 Cosine Similarity between Neuron A and B: 0.2068
Layer 2 Cosine Similarity between Neuron A and B: 0.1854

Train Epoch 5: Average loss: 0.2669
Test set: Average loss: 0.2449, Accuracy: 9312/10000 (93.12%)
Layer 1 Cosine Similarity between Neuron A and B: 0.2201
Layer 2 Cosine Similarity between Neuron A and B: 0.1991
```

Performing a an increase number of computation neurons test 1-5 so 5 tests overall:
- seems to have near identical training accuracy and loss performances over the different epochs.
    - Overall training loss is nearly the same even though all the neurons are evenly being activated

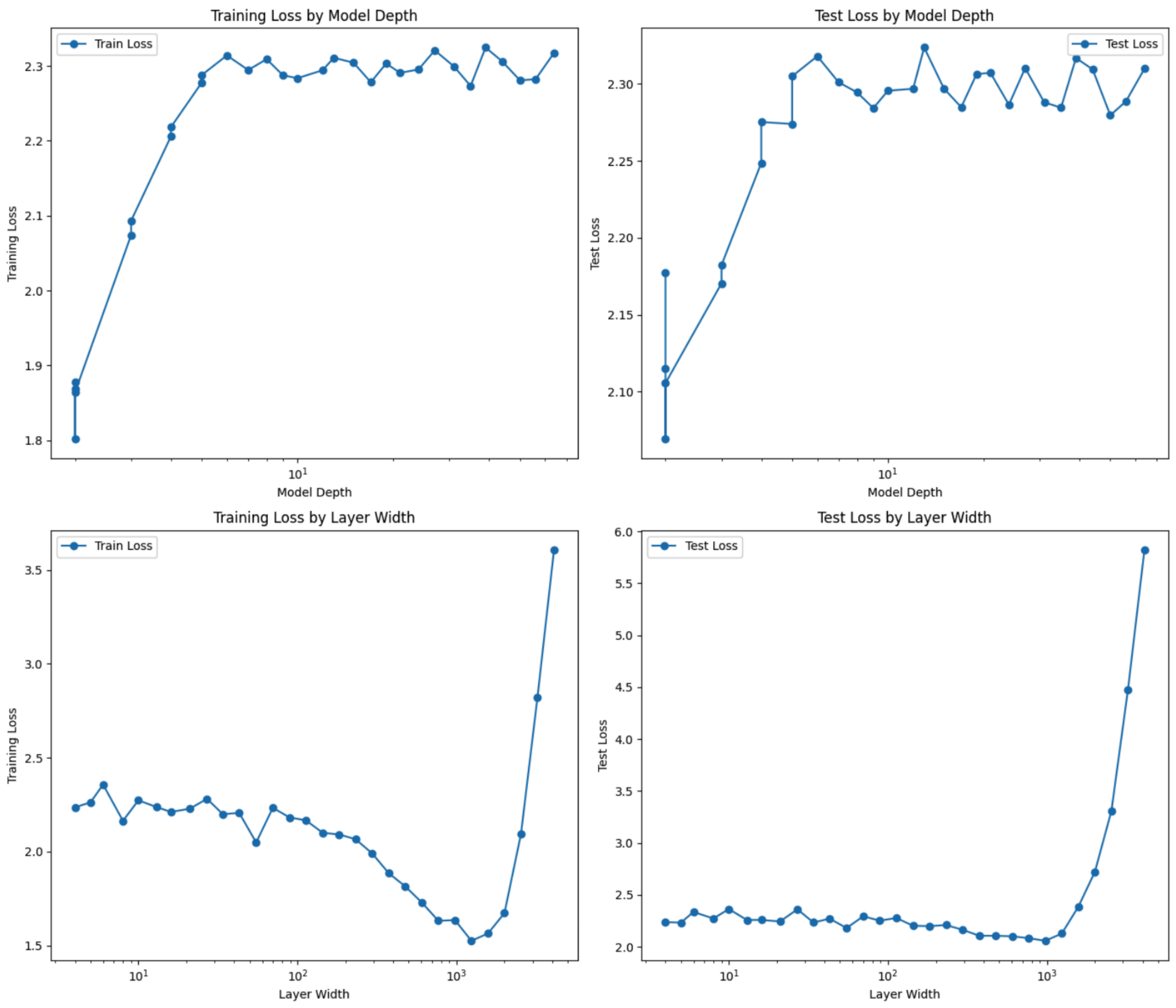




Didn’t save the models for cosine similarity but I hope they are similar to the 2 neuron version as this took forever. The selection and bias and entropy all looks good to me though.

Random tangent: how does the size of a model effect the early training performance and decent. Does the depth and width increasing have a massive effect?

Interesting results. Depth improves early training for a decent increase but then massively fails and levels off. Depth can go to incredible widths and still be decreasing initial loss on the 10 examples but then explodes.



Initial work with a transformer model:

I believe I got it to work.

With 2 computation neurons:
Epoch [1/5], Average Loss: 8.3382

Base standard model:
Epoch [1/1], Average Loss: 8.1562

There is an update in this to allow it to keep an identical transformers architecture.

Forward Prop:

- input goes to the selection neuron
- selects a computation neuron to use
- original input goes to the computation neuron and then out

Back prop:
- gradient goes to the computation neuron
- this is then back propped to the selection neuron
- but the original gradient from the computation neuron is what is sent backward effectively skipping the selection neuron while still allowing it to update.

This effectively creates a hidden selection method while still "looking" identical to a transformer model.

Even more interesting is how the losses are near identical to each other just like the mnist previous testing.

However this was only replacing the linear layers with this new method. I am now trying to reimplement the attention layers the same way.
Still works!
Epoch [1/5], Average Loss: 8.2926
Increasing number of computation neurons:
5: Epoch [1/5], Average Loss: 8.3156
50: Epoch [1/1], Average Loss: 8.3408
75: Epoch [1/1], Average Loss: 8.3711
I run OOM in more but I don't understand why this is working. I think something is wrong and will need to debug it. Chat sees nothing wrong though so it's probably subtle.

Monitoring and verification run:
- seems to somewhat be working but some layers just have low entropy while others have high. Seems disbalanced.
- I may want to implement some early stage of training randomness to the selector neurons to encourage early stage exploration.

Layer	Selection Frequency (%)	Average Entropy	Average Gradient Magnitude per Neuron
layers.0.self_attn.q_linear	[20.26, 19.94, 20.10, 19.75, 19.95]	0.1973	[248.52, 246.74, 247.47, 249.14, 246.45]
layers.0.self_attn.k_linear	[19.96, 19.94, 19.82, 20.20, 20.07]	0.1855	[83.60, 82.21, 84.27, 85.08, 84.61]
layers.0.self_attn.v_linear	[20.17, 20.05, 19.97, 19.91, 19.89]	0.1774	[9.45, 9.56, 9.37, 9.18, 9.44]
layers.0.self_attn.out_proj	[20.98, 20.05, 20.39, 17.86, 20.72]	0.1594	[5.47, 5.41, 5.10, 4.46, 4.93]
layers.0.linear1	[19.53, 19.21, 18.63, 21.30, 21.33]	1.4446	[14.59, 14.69, 14.78, 14.66, 15.64]
layers.0.linear2	[21.51, 19.13, 20.36, 19.72, 19.28]	1.2696	[4.19, 4.20, 4.04, 3.70, 3.94]
layers.1.self_attn.q_linear	[18.65, 20.84, 17.90, 21.25, 21.35]	1.3825	[0.20, 0.22, 0.18, 0.23, 0.21]
layers.1.self_attn.k_linear	[15.71, 21.46, 21.39, 22.05, 19.39]	1.2469	[0.21, 0.27, 0.25, 0.25, 0.24]
layers.1.self_attn.v_linear	[18.84, 21.47, 19.53, 19.92, 20.24]	1.2617	[3.50, 4.12, 3.77, 4.16, 3.82]
layers.1.self_attn.out_proj	[18.80, 22.53, 16.96, 20.18, 21.54]	1.4093	[3.31, 3.94, 2.95, 3.75, 4.12]
layers.1.linear1	[20.35, 19.09, 19.90, 21.47, 19.18]	1.4803	[5.57, 5.14, 5.21, 5.52, 5.35]
layers.1.linear2	[18.58, 17.39, 21.74, 20.98, 21.31]	1.5308	[6.21, 5.72, 6.76, 6.68, 6.78]
linear_out	[20.10, 19.84, 19.96, 20.12, 19.98]	0.2563	[79.22, 66.48, 72.29, 73.92, 91.31]

Ran a repeat with 16 layers stacked instead of 2:
Epoch [1/1], Average Loss: 8.2488

Same effect of early layers (first layer basically) having low entropy and all later layers having high.

Final layer is also very very low entropy.

10-16-24:Performing more tests and check on new architecture

Forward Pass Goals:

1. **Direct Input to Selection and Computation Neurons:**
   - The **raw input** is sent directly to both the **selection neuron** and the **computation neurons**.
2. **Selection Neuron Chooses a Computation Neuron:**
   - The selection neuron processes the **raw input** to generate selection logits, which determine which computation neuron will be active.
3. **Input Routed to the Selected Computation Neuron:**
   - Based on the selection, only the chosen computation neuron processes the **raw input** to produce an output.
   - All other computation neurons are ignored in this step.
4. **Output Solely from Selected Computation Neuron:**
   - The final output of the `CustomNeuron` layer is generated only by the selected computation neuron. This output is then passed on to the next layer.

Backward Pass Goals:

1. **Gradient Flow from Computation Neuron:**
   - During backpropagation, gradients flow **back through the selected computation neuron only**.
   - No gradients are calculated for the inactive computation neurons.
2. **Gradient Flow to Selection Neuron:**
   - After exiting the computation neuron, the gradient flows in reverse to the **selection neuron**.
   - The selection neuron receives updates based on these gradients, allowing it to improve its selection criteria over time.
3. **Gradient Flow to Next Layer:**
   - The gradient also flows from the output of the selected computation neuron **to the next layer in the network**.
   - This ensures that the gradients follow the reverse path of the forward pass, passing through both the computation neuron and the selection neuron before reaching previous layers.

Seems back prop is not working correctly and is updating all the computation neurons every single time. This doesn’t necessarily negate the previous findings but I will need reverification.

CL test on the same method I did the ewc system, better but still bad results:

Base model:
```
Accuracy on Task 1: 0.00%
Accuracy on Task 2: 49.46%
Accuracy on Task 3: 0.00%
Accuracy on Task 4: 0.00%
Accuracy on Task 5: 0.00%
```

EWC Base:
```
Accuracy on Task 1: 92.11%
Accuracy on Task 2: 72.85%
Accuracy on Task 3: 18.52%
Accuracy on Task 4: 24.45%
Accuracy on Task 5: 11.64%
```

Previous attempts did not update the selection neuron, revising and retrying.

After updating it’s actually significantly worse at CL and unlike base model it basically just gets perfect score on whatever it is directly training on at the time with no memory of past events.

I’m seeing the forgetting and remembering again but now for the entire training set at times.