COMP 576 Assignment 1

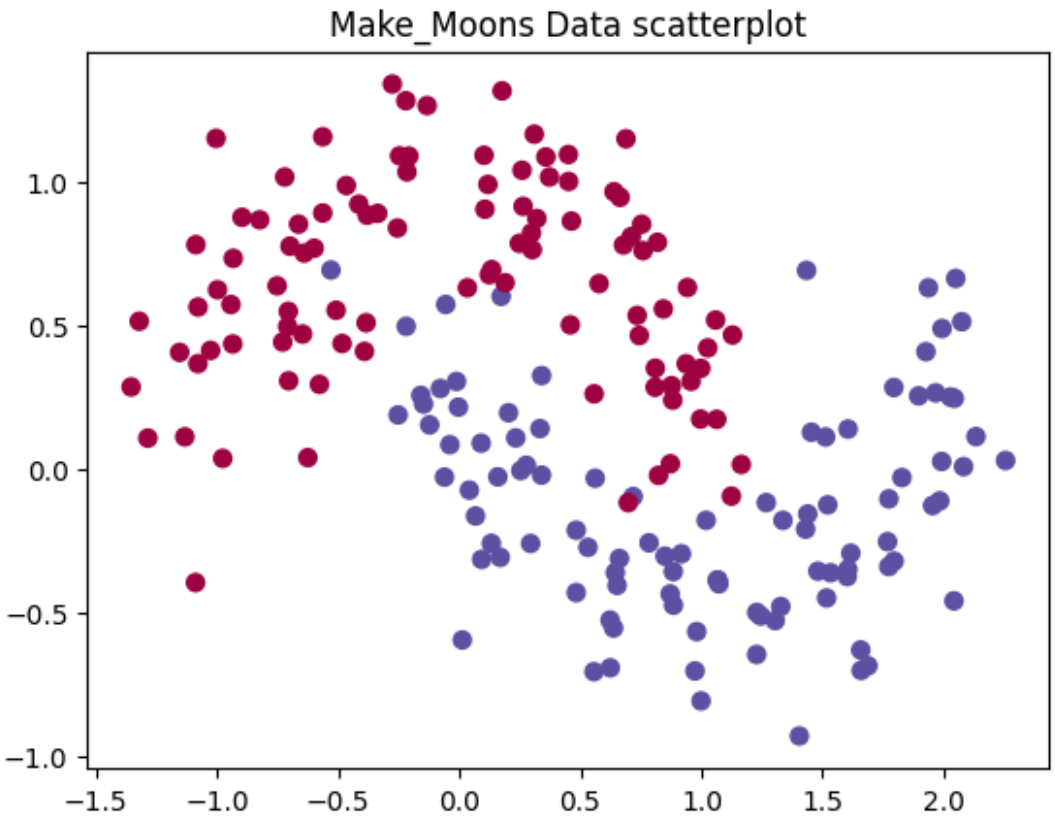
Jerry Yang

Rice University

Due Date: Oct 14, 2025

Contents

1a: Make-Moons dataset



1b: Activation Derivation for Tanh, Sigmoid and ReLU

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\text{sigmoid}(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\text{ReLU}(x) = \max(0, x)$$

derivatives of the above

1. Sigmoid Function

$$\frac{d\sigma}{dz} = \frac{d}{dz}(1 + e^{-z})^{-1} = -(1 + e^{-z})^{-2} \cdot (-e^{-z}) = \frac{e^{-z}}{(1 + e^{-z})^2}$$

Now express it in terms of $\sigma(z)$:

$$\sigma(z) = \frac{1}{1+e^{-z}} \quad \Rightarrow \quad 1-\sigma(z) = \frac{e^{-z}}{1+e^{-z}}$$

$$\sigma'(z) = \sigma(z)\big(1-\sigma(z)\big)$$

—

2. Tanh

$$\frac{d}{dz}\left(\frac{e^z - e^{-z}}{e^z + e^{-z}}\right) = \frac{(e^z+e^{-z})^2\, d(e^z-e^{-z}) - (e^z-e^{-z})(e^z+e^{-z})\, d(e^z+e^{-z})}{(e^z+e^{-z})^2}$$

$$= \frac{(e^z+e^{-z})(e^z+e^{-z}) - (e^z-e^{-z})(e^z-e^{-z})}{(e^z+e^{-z})^2}$$

$$= \frac{(e^z+e^{-z})^2 - (e^z-e^{-z})^2}{(e^z+e^{-z})^2}$$

$$= 1 - \left(\frac{e^z-e^{-z}}{e^z+e^{-z}}\right)^2$$

$$= 1 - \tanh^2(z)$$

—

3.ReLU

The ReLU function is defined as:

$$\mathrm{ReLU}(z) = \max(0,z) = \begin{cases} 0, & z \le 0, \\ z, & z > 0 \end{cases}$$

Differentiate piecewise:

$$\mathrm{ReLU}'(z) = \begin{cases} 0, & z < 0, \\ 1, & z > 0 \end{cases}$$

Thus,

$$f'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x < 0 \text{ or undefined if } x = 0 \end{cases}$$

1c: Feedforward and Calculate Loss Functions Formulas

For Feedforward:

$$z_1 = W_1 X + b_1,$$

$$a_1 = f(z_1),$$

$$z_2 = W_2 a_1 + b_2,$$

$$\hat{y} = \text{softmax}(z_2) = \frac{e^{z_2}}{\sum_{j=1}^{C} e^{z_{2,j}}}$$

`self.a1 = actFun(self.z1)` $\Rightarrow a_1 = f(z_1)$

`self.z2 = np.dot(self.a1, self.W2) + self.b2` $\Rightarrow z_2 = W_2 a_1 + b_2$

`exp_scores = np.exp(self.z2 - np.max(self.z2, axis=1, keepdims=True))` $\Rightarrow e^{z_{2,i} - \max(z_2)}$

`self.probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)` $\Rightarrow \hat{y}_i = \frac{e^{z_{2,i} - \max(z_2)}}{\sum_{j=1}^{C} e^{z_{2,j} - \max(z_2)}}$

For Calculating Loss:

`self.feedforward(X, ...)` $\Rightarrow$ compute $\hat{y} = \text{softmax}(z_2)$

`probs` = `np.exp(self.z2) / np.sum(np.exp(self.z2), axis=1, keepdims=True)` $\Rightarrow \hat{y}_i = \frac{e^{z_{2,i}}}{\sum_{j=1}^{C} e^{z_{2,j}}}$

`data_loss_single` = `-np.log(probs[range(num_examples), y])` $\Rightarrow \ell_n = -\log\left(\hat{y}_{n,y_n}\right)$

`data_loss` = `np.sum(data_loss_single)` $\Rightarrow L_{\text{data}} = \sum_{n=1}^{N} \ell_n$

`data_loss` += `self.reg_lambda / 2 * (np.sum(self.W1**2) + np.sum(self.W2**2))` $\Rightarrow$ regulatization term to loss and return 1/N at the end

1d: Backward Pass

$$\delta_2 = \frac{\partial L}{\partial z_2} = \hat{y} - y$$

$$\frac{\partial L}{\partial W_2} = a_1^\top \delta_2$$

$$\frac{\partial L}{\partial b_2} = \sum_{n=1}^{N} \delta_2$$

$$\delta_1 = \frac{\partial L}{\partial z_1} = (\delta_2 W_2^\top) \odot f'(z_1)$$

$$\frac{\partial L}{\partial W_1} = X^\top \delta_1$$

$$\frac{\partial L}{\partial b_1} = \sum_{n=1}^{N} \delta_1$$

Backpropagation: Mathematical Derivation and Python Implementation

The backpropagation algorithm computes the gradients of the loss function with respect to each parameter of the network, using the chain rule.

$z_1 = W_1 X + b_1$, `self.z1 = np.dot(X, self.W1) + self.b1`

$a_1 = f(z_1)$, `self.a1 = actFun(self.z1)`

$z_2 = W_2 a_1 + b_2$, `self.z2 = np.dot(self.a1, self.W2) + self.b2`

$\hat{y} = \text{softmax}(z_2)$, `self.probs = softmax(self.z2)`

$L = -\frac{1}{N}\sum_{n=1}^{N} \log(\hat{y}_{n,y_n})$, `data_loss = -np.sum(np.log(self.probs[range(N), y])) / N`

$$\delta_3 = \frac{\partial L}{\partial z_2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_2}$$

$$\frac{\partial L}{\partial W_2} = a_1^\top \delta_3$$

$$\frac{\partial L}{\partial b_2} = \sum \delta_3$$

$$\delta_2 = \frac{\partial L}{\partial z_1} = \texttt{diff} * (\delta_3 W_2^T))$$

$$\frac{\partial L}{\partial W_1} = X^\top \delta_2$$

$$\frac{\partial L}{\partial b_1} = \sum \delta_2$$

where diff is the derivative of the activation function evaluated at the hidden layer pre-activation z_1.

Code provided below:

```
num_examples = len(X)
delta3 = self.probs
delta3[range(num_examples), y] -= 1
```

```
# dL/dW2j
dW2 = (self.a1.T).dot(delta3)
db2 = np.sum(delta3, axis=0, keepdims=True)# dL/db2
delta2 = delta3.dot(self.W2.T) * (self.diff_actFun(self.a1,type=self.actFun_t
#dL/dW1
dW1 = np.dot(X.T,delta2)
#dL/db1
db1 = np.sum(delta2,axis = 0)
```

Results of activation functions + change in tanh hidden layer analysis

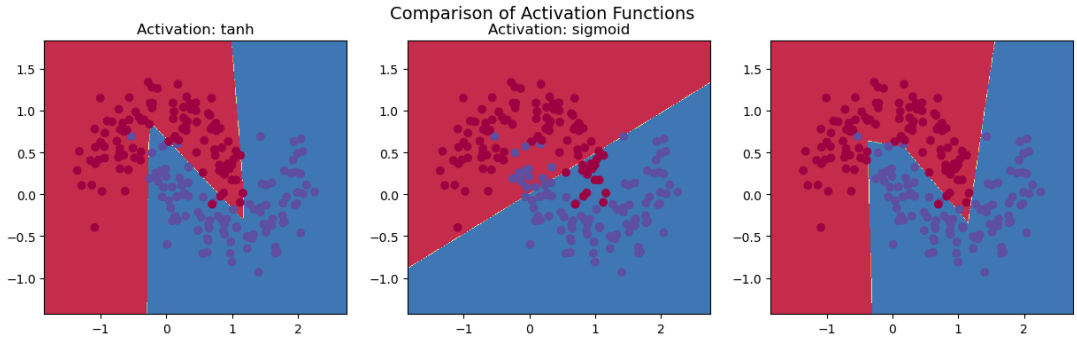
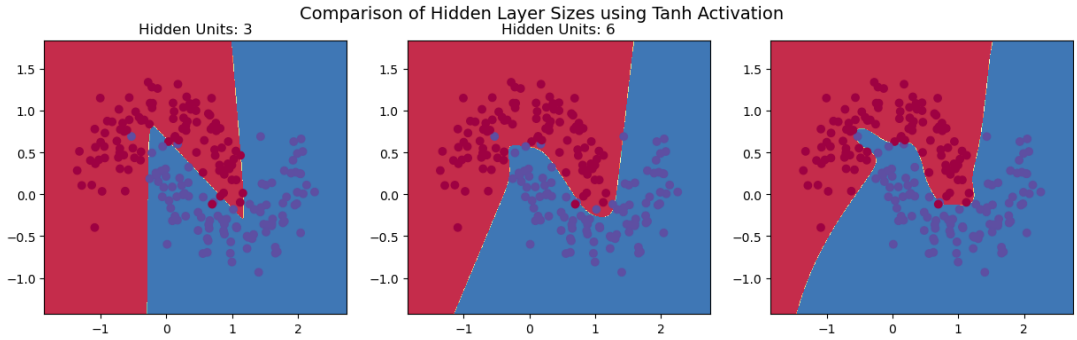


Figure above illustrates the decision boundaries produced by three activation functions: **tanh**, **sigmoid**, and **ReLU**. The **tanh** activation yields a smooth and flexible boundary that successfully captures the nonlinear separation of the two half-moon clusters, reflecting its zero-centered output and stronger gradient propagation. In contrast, the **sigmoid** activation produces a softer, less adaptive boundary that underfits the data, as its outputs saturate near 0 and 1, causing vanishing gradients. The **ReLU** activation, while capable of capturing the class separation, forms sharper, piecewise-linear boundaries due to its linear behavior for positive activations. Tthese visual differences demonstrate how the nonlinearity of the activation function governs the expressiveness and shape of the learned decision boundary.

Change in tanh hidden layer Finding



Above demonstrates the effect of varying the number of hidden units (3, 6, and 9) on the decision boundary of a neural network using the **tanh** activation function. As you can see, the number of hidden units increases, the model transitions from underfitting (overly simple boundary) to a well-balanced fit (smooth and accurate boundary), and finally to mild overfitting with overly complex curves capturing noise. We can tell, in this case, that the increase in the number of hidden layers benefits in capturing non-linearity, but reduces generalization as it goes.

1f: Extending the Neural Network Architecture to n-layer

Recap: 3-layer implemenation

The 3-layer implementation repeated similar operations for each layer:

$$z^{(l)} = a^{(l-1)}W^{(l)} + b^{(l)}, \quad a^{(l)} = f(z^{(l)})$$

To support arbitrary depth, we needed to:

- Create a flexible way to initialize weights and biases for any layer count.
- Dynamically manage the feedforward and backpropagation sequences.
- Isolate layer-specific operations (activation, gradient, weight updates).

This was achieved by modularizing computation into a separate `Layer` class and letting the `DeepNeuralNetwork` act as a controller that sequences these layers.

—

Parameter Initialization for an n-Layer Network

Instead of explicitly defining $W_1, W_2, \ldots$, we iterate through the list of dimensions `nn_dims`:

$$\text{for } i = 1 \ldots L-1: \quad W^{(i)} \sim \mathcal{N}(0, \tfrac{1}{\sqrt{d_i}}), \quad b^{(i)} = 0$$

Code Listing 1: Deep Neural Network Initialization (Generalized)

```
class DeepNeuralNetwork:
    def __init__(self, nn_dims, actFun_type='tanh', reg_lambda=0.01, seed=
        self.layers = []
        self.reg_lambda = reg_lambda
        self.actFun_type = actFun_type
        for i in range(len(nn_dims) - 1):
            self.layers.append(
                Layer(
                    nn_dims[i], nn_dims[i + 1],
                    actFun_type=actFun_type,
                    reg_lambda=reg_lambda,
                    seed=seed
                )
            )
```

This generalization enables any structure (e.g., [2, 3, 2], [2, 4, 3, 2], or [2, 8, 6, 4, 2]) by simply passing a list of dimensions.

—

Introducing the `Layer` Class

To modularize computation, we introduced a new class that encapsulates the feedforward and backpropagation logic of a single layer.

Code Listing 2: Layer Class for Modular Computation

```
class Layer:
    def __init__(self, input_dim, output_dim, actFun_type='tanh', reg_lambd
        np.random.seed(seed)
        self.W = np.random.randn(input_dim, output_dim) / np.sqrt(input_dim
        self.b = np.zeros((1, output_dim))
        self.actFun_type = actFun_type
        self.reg_lambda = reg_lambda

    def feedforward(self, X):
        self.input = X
        self.z = np.dot(X, self.W) + self.b
        self.a = self.actFun(self.z)
        return self.a

    def backprop(self, delta_next, W_next=None):
        if W_next is not None:
            delta = np.dot(delta_next, W_next.T) * self.diff_actFun(self.z)
        else:
            delta = delta_next * self.diff_actFun(self.z)
        dW = np.dot(self.input.T, delta) + self.reg_lambda * self.W
        db = np.sum(delta, axis=0, keepdims=True)
        return delta, dW, db
```

Each layer now independently computes its linear transformation, activation, and local gradients. The deep network simply orchestrates these computations across multiple layers.

—

Feedforward: Sequential Layer Composition

The 3-layer version explicitly coded two transformations. In the new implementation, this process is fully looped and generalized:

$$a^{(0)} = X, \quad a^{(l)} = f(W^{(l)}a^{(l-1)} + b^{(l)})$$

Code Listing 3: Generalized Feedforward Process

```
def feedforward(self, X):
    a = X
    for layer in self.layers:
        a = layer.feedforward(a)
    self.z_out = self.layers[-1].z
    shifted_logits = self.z_out - np.max(self.z_out, axis=1, keepdims=True)
    exp_scores = np.exp(shifted_logits)
    self.probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
    return self.probs
```

This design supports any number of layers without code modification.

—

Backpropagation: Modular Gradient Propagation

Previously, the gradient equations were explicitly written for two weight matrices (W_1, W_2). Now, gradients are propagated recursively through layers:

$$\delta^{(L)} = \hat{y} - y, \quad \delta^{(l)} = (\delta^{(l+1)}W^{(l+1)^T}) \odot f'(z^{(l)})$$

Code Listing 4: Generalized Backpropagation

```
def backprop(self, X, y):
    num_examples = len(X)
    delta = self.probs.copy()
    delta[range(num_examples), y] -= 1
    dW, db = [], []
    next_delta = delta
    next_W = None
    for layer in reversed(self.layers):
        next_delta, dW_i, db_i = layer.backprop(next_delta, W_next=next_W)
```

```
            dW.insert(0, dW_i)
            db.insert(0, db_i)
            next_W = layer.W
        return dW, db
```

The network iterates backwards through layers, allowing fully dynamic gradient computation regardless of depth.

—

Loss Function and Regularization

The total loss combines cross-entropy with L2 regularization across all layers:

$$\mathcal{L} = -\frac{1}{N}\sum_{i}\log(\hat{y}_{i,y_i}) + \frac{\lambda}{2}\sum_{l}\|W^{(l)}\|^2$$

Code Listing 5: Cross-Entropy Loss with L2 Regularization

```
def calculate_loss(self, X, y):
    num_examples = len(X)
    probs = self.feedforward(X)
    correct_logprobs = -np.log(probs[range(num_examples), y] + 1e-9)
    data_loss = np.sum(correct_logprobs)
    W_sum = sum([np.sum(np.square(layer.W)) for layer in self.layers])
    data_loss += self.reg_lambda / 2 * W_sum
    return (1. / num_examples) * data_loss
```

—

Training Loop

The training loop now performs sequential updates across all layers:

Code Listing 6: Gradient Descent Parameter Update

```
def fit_model(self, X, y, epsilon=0.01, num_passes=20000, print_loss=True):
    for i in range(num_passes):
        self.feedforward(X)
```

```
        dW, db = self.backprop(X, y)
        for j, layer in enumerate(self.layers):
            layer.W -= epsilon * dW[j]
            layer.b -= epsilon * db[j]
        if print_loss and i % 1000 == 0:
            print(f Loss after iteration {i}: {self.calculate_loss(X, y):.6
```

Effect of Activation Function: ReLU, Tanh, Sigmoid

Empirical Observation

For three activations, we used configurations as following: [2, 3, 2], [2, 6, 2], [2, 4, 3, 2], and [2, 4, 3, 2, 2]. Each network was trained on the Make Moons dataset with identical training parameters and learning rate.

ReLu: Analysis and Interpretation

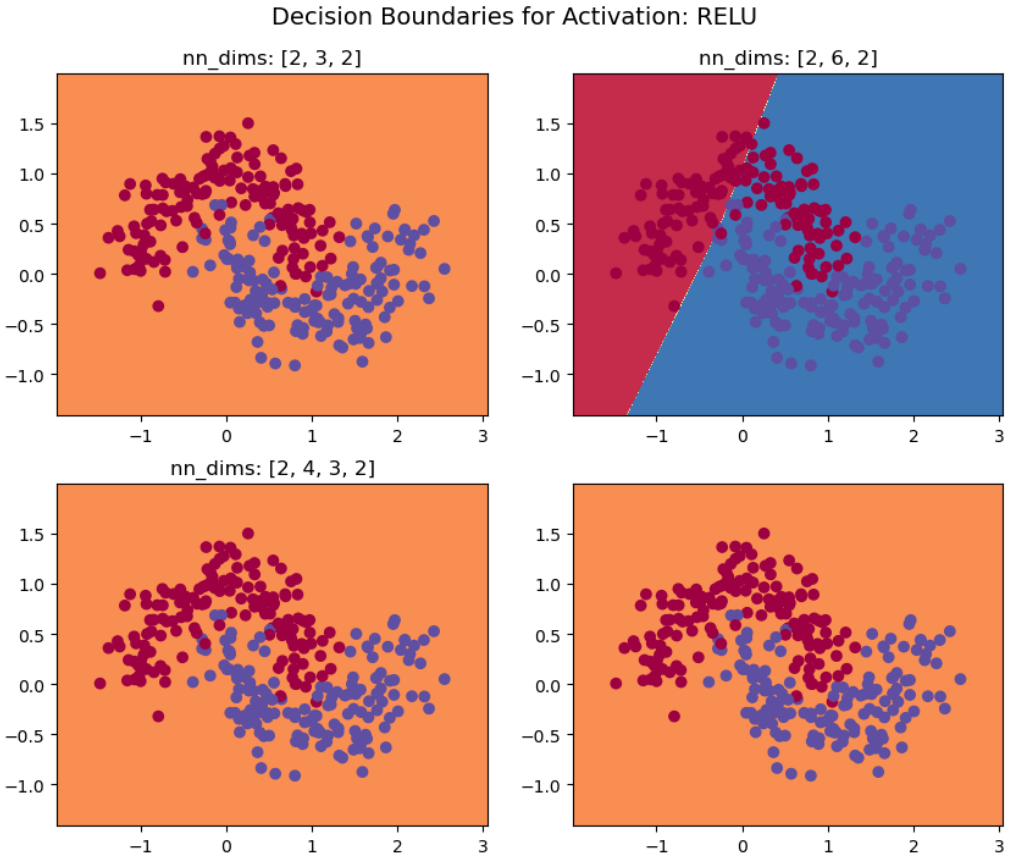


Figure 1: Decision boundaries of ReLU-activated networks across increasing depth and width. Top row: shallow architectures with one hidden layer; bottom row: deeper networks with two or more hidden layers.

So not sure what's going on here, the only framework that can result in something is with [2,6,2]. I searched online and it says its likely due to inactive (dead) neurons caused by negative pre-activations. I believe there's something wrong in my code but I tried to solve it and I can't.

Effect of Activation Function: Tanh

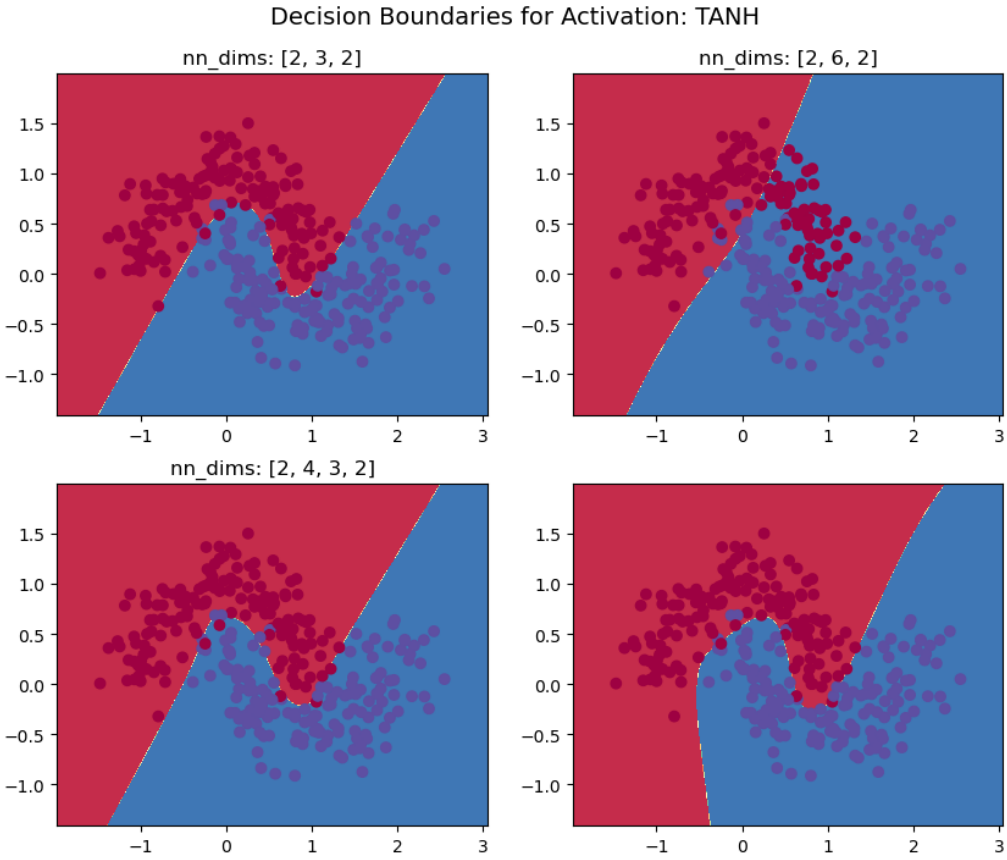


Figure 2: Decision boundaries for Tanh activation across different architectures.

I noticed that increasing the number of layers and neurons slightly refined the boundary but did not drastically change the overall pattern. The Tanh functions smooth, symmetric shape helped the model generalize well without producing abrupt separations, and it showed consistent performance even in shallower configurations.

Effect of Activation Function: Sigmoid

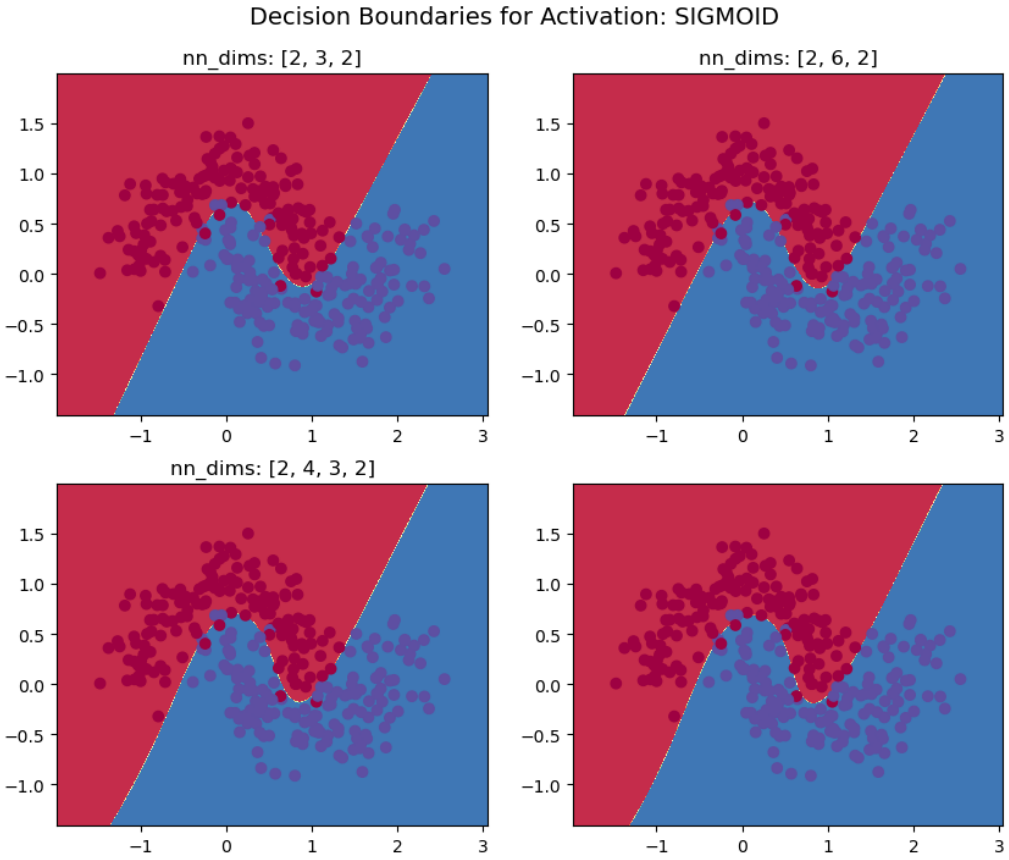


Figure 3: Decision boundaries for Tanh activation across different architectures.

The deeper architectures made only marginal improvements in boundary smoothness. While Sigmoid captured the overall data pattern well, its gradients tended to saturate near 0 or 1, which likely slowed learning compared to Tanh. Overall, it performed consistently but less sharply, confirming its tendency to train slower and produce smoother decision surfaces.

2a: Train 4-layer DCN

Data Import

What I did here is to downloads the MNIST dataset (70,000 images of digits 09, each 2828 pixels). Converts them into PyTorch tensors (values in [0,1]). Normalizes pixel intensities using the datasets mean (0.1307) and std (0.3081) for stability.

Splits data into 55,000 images for training, 5,000 images for validation, and 10,000 images for testing.

Code Listing 7: Data Import

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
full_train = datasets.MNIST('./data', train=True, download=True,
transform=transform)
train_ds, val_ds = random_split(full_train, [55000, 5000])
train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True)
val_loader   = DataLoader(val_ds,   batch_size=test_batch_size)
test_loader  = DataLoader(datasets.MNIST('./data', train=False,
transform=transform),batch_size=test_batch_size)
```

Build Network

conv1(5-5-1-32) - ReLU - maxpool(2-2) - conv2(5-5-32-64) - ReLU - maxpool(2-2) - fc(1024) - ReLU - DropOut(0.5) - Softmax(10)

Code Listing 8: Build Network

```
def _act(x, kind):
    if kind == relu : return F.relu(x, inplace=False)
    if kind == leakyrelu : return F.leaky_relu(x, negative_slope=0.1)
    if kind == tanh : return torch.tanh(x)
    if kind == sigmoid : return torch.sigmoid(x)
    return x

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5, stride=1, padding=2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5, stride=1, padding=2)
```

```
        self.conv2_drop = nn.Dropout(p=0.5)  # dropout after fc per forward
        self.fc1 = nn.Linear(64 * 7 * 7, 1024)
        self.fc2 = nn.Linear(1024, 10)
        self.cache = {}

    def forward(self, x):
        z1 = self.conv1(x)
        a1 = _act(z1, ACT_CONV)
        p1 = F.max_pool2d(a1, 2)

        z2 = self.conv2(p1)
        a2 = _act(z2, ACT_CONV)
        p2 = F.max_pool2d(a2, 2)

        f = torch.flatten(p2, 1)
        z3 = self.fc1(f)
        a3 = _act(z3, ACT_FC)
        a3 = self.conv2_drop(a3)  # dropout after fc
        z4 = self.fc2(a3)

        self.cache = { z1 : z1,  a1 : a1,  p1 : p1,  z2 : z2,  a2 : a2,  p2
         z3 : z3,  a3 : a3,  z4 : z4}
        return F.softmax(z4, dim=1)

model = Net().to(device)

def _init_weights(m):
    if isinstance(m, nn.Conv2d):
        nonlin =  relu  if ACT_CONV in ( relu , leakyrelu ) else  linear
        nn.init.kaiming_normal_(m.weight, mode= fan_out , nonlinearity= lea
        if m.bias is not None: nn.init.zeros_(m.bias)
    elif isinstance(m, nn.Linear):
```

```
        nn.init.xavier_normal_(m.weight)
        if m.bias is not None: nn.init.zeros_(m.bias)

model.apply(_init_weights)
```

Training

Code Listing 9: Starter Code

```
if OPTIMIZER == adam :
optimizer = optim.Adam(model.parameters(), lr=lr)
elif OPTIMIZER == sgd :
optimizer = optim.SGD(model.parameters(), lr=lr)
elif OPTIMIZER == momentum :
optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.9)
else:
optimizer = optim.Adam(model.parameters(), lr=lr)
def train(epoch):
    model.train()
    criterion = nn.NLLLoss()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(torch.log(output + eps), target)
        loss.backward()
        optimizer.step()

        if batch_idx % logging_interval == 0:
            pred = output.argmax(dim=1)
            acc = (pred == target).float().mean().item()
            n_iter = (epoch - 1) * len(train_loader) + batch_idx
            print(f Train Epoch: {epoch} [{batch_idx*len(data)}/
```

```
------------{len(train_loader.dataset)}]-
            f Loss:-{loss.item():.4f}--Acc:-{acc*100:.2f}% )
        writer.add_scalar( train/loss , loss.item(), n_iter)
        writer.add_scalar( train/acc , acc, n_iter)
        writer.add_scalar( train/error , 1.0 - acc, n_iter)
        train_steps.append(n_iter); train_losses.append(loss.item());
        train_accs.append(acc)

        for name, param in model.named_parameters():
            tag = f {cfg_tag}/param/{name}
            _update_stats(tag, param, n_iter)
            writer.add_histogram(f {tag}/hist , param.detach().cpu()
            .numpy(), n_iter)
            writer.add_scalar(f {tag}/min ,
            STAT[tag][ min ][-1], n_iter)
            writer.add_scalar(f {tag}/max , STAT[tag][ max ][-1],
n_iter)
            writer.add_scalar(f {tag}/mean , STAT[tag][ mean ][-1]
            , n_iter)
            writer.add_scalar(f {tag}/std , STAT[tag][ std ][-1],
n_iter)
        for key, val in getattr(model, cache , {}).items():
            tag = f {cfg_tag}/act/{key}
            _update_stats(tag, val, n_iter)
            writer.add_histogram(f {tag}/hist , val.detach().cpu()
            .numpy(), n_iter)
            writer.add_scalar(f {tag}/min , STAT[tag][ min ][-1],
n_iter)
            writer.add_scalar(f {tag}/max , STAT[tag][ max ][-1],
            n_iter)
            writer.add_scalar(f {tag}/mean , STAT[tag][ mean ][-1], n_
            writer.add_scalar(f {tag}/std , STAT[tag][ std ][-1],
```

```
n_iter)

@torch.no_grad() elf.conv2_drop = nn.Dropout(0.5)
```

Accuarcy Reporting

The network learned very quickly and reached high accuracy early in the process when I started training. The training loss dropped sharply within the first few iterations, showing that the model was able to minimize errors efficiently. The error curve showed brief spikes but consistently moved downward, indicating steady progress despite minor fluctuations. The accuracy curve climbed rapidly and stabilized near 100%, suggesting that the network was able to capture the patterns in the MNIST dataset effectively. Overall, the ReLUReLUAdam setup demonstrated fast and adaptive convergence, with strong and responsive weight updates throughout training.

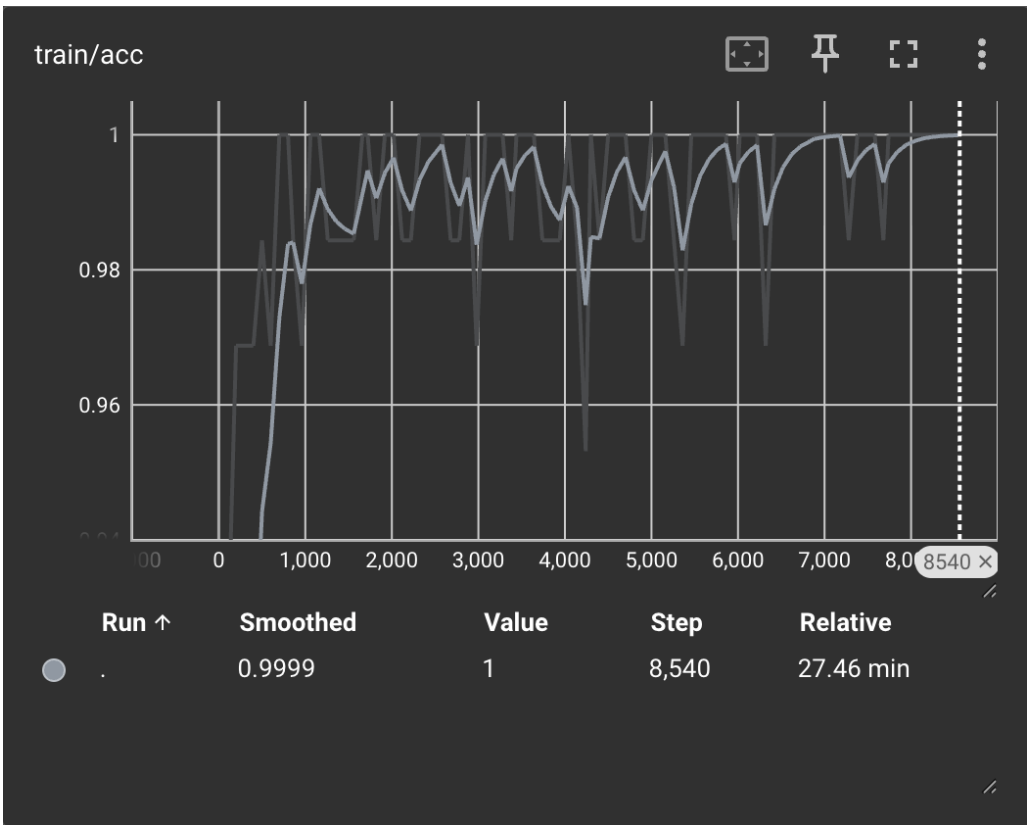


Figure 4: train accuracy result.

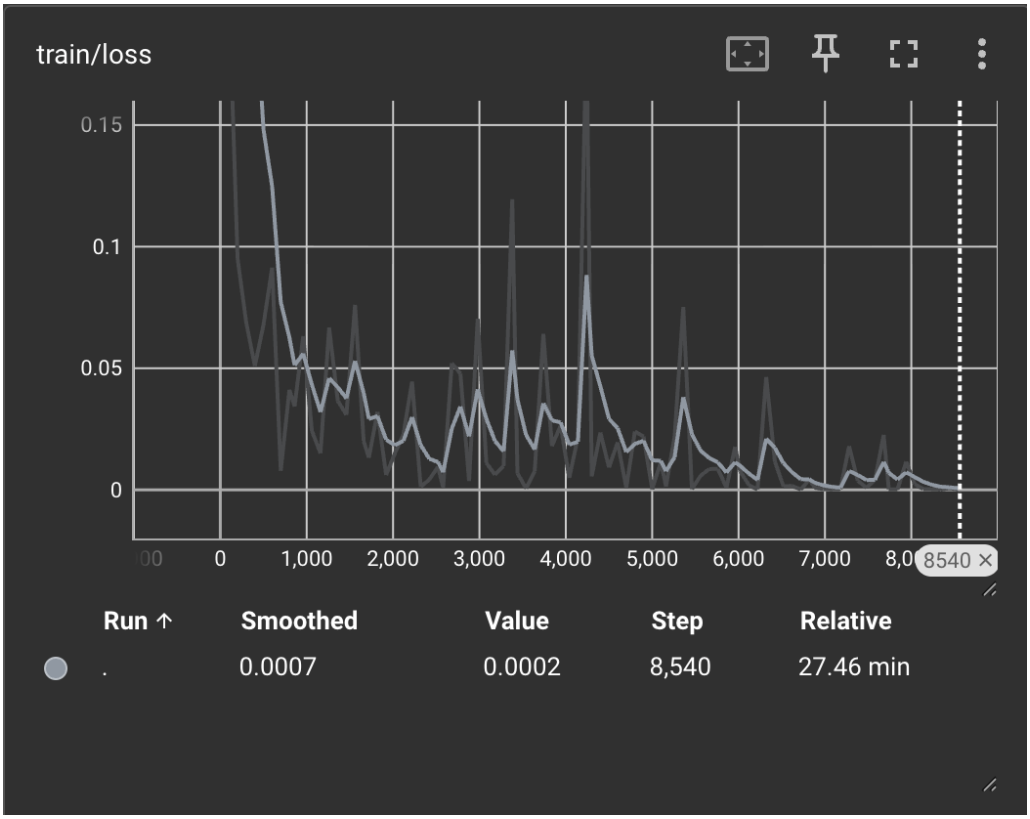


Figure 5: train loss result

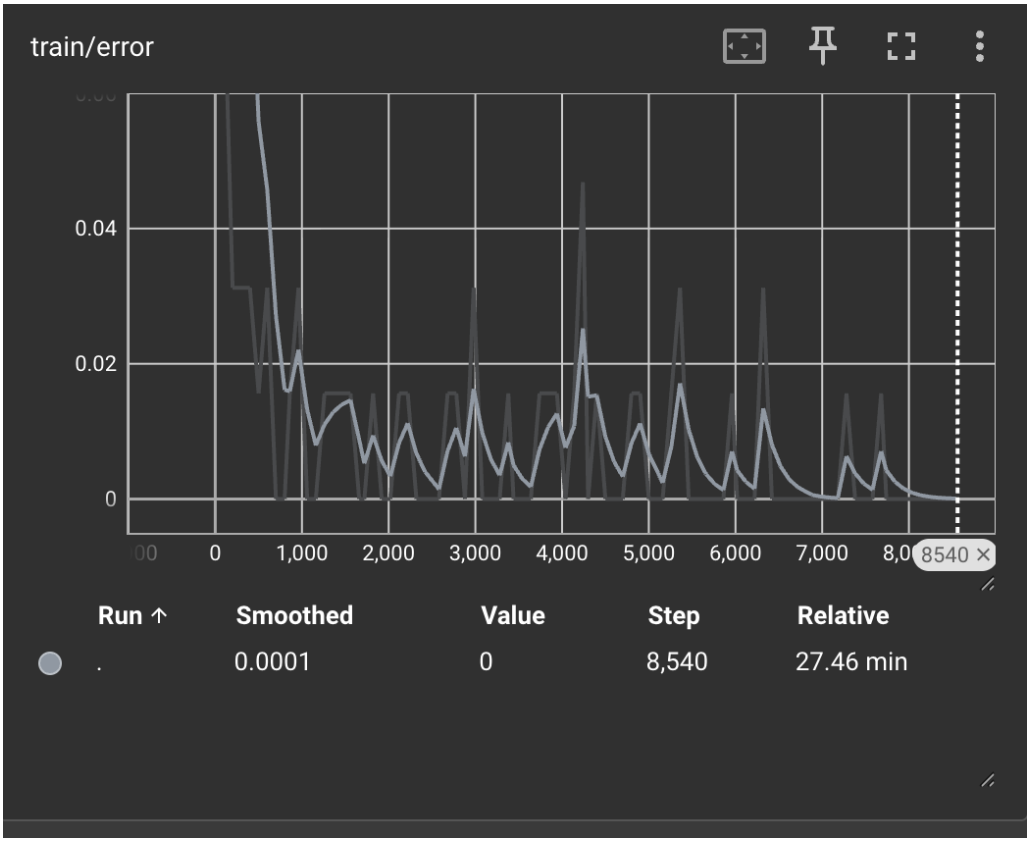


Figure 6: train error result

Relu-Relu-Adam test result

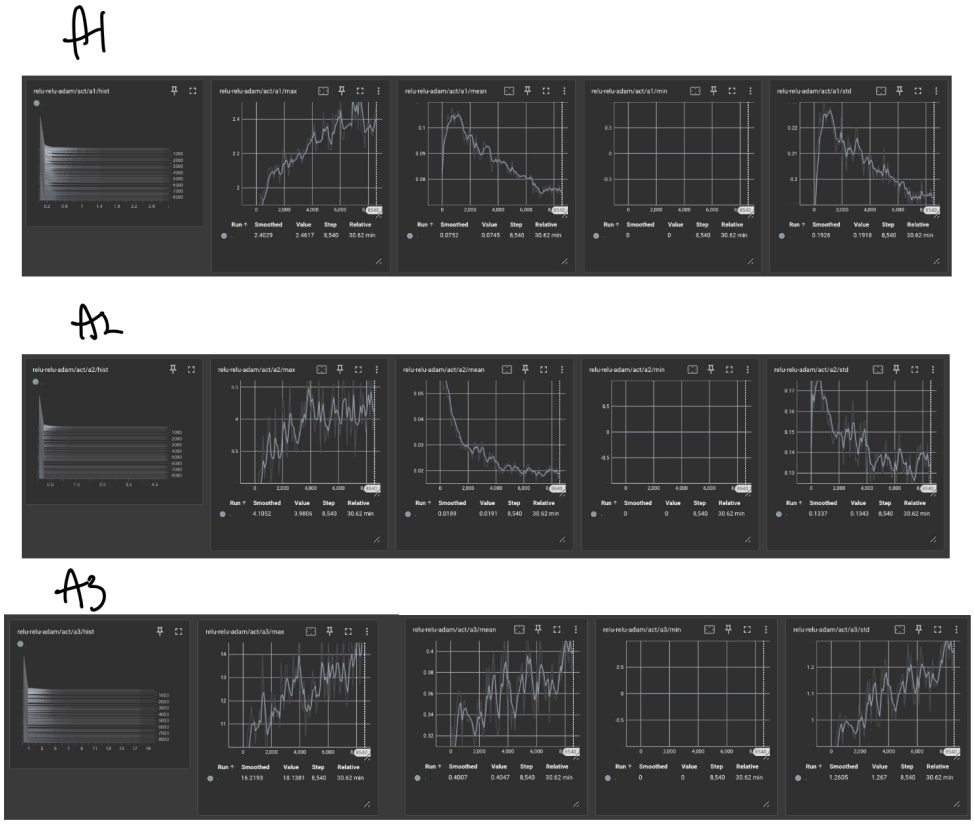


Figure 7: a statistics

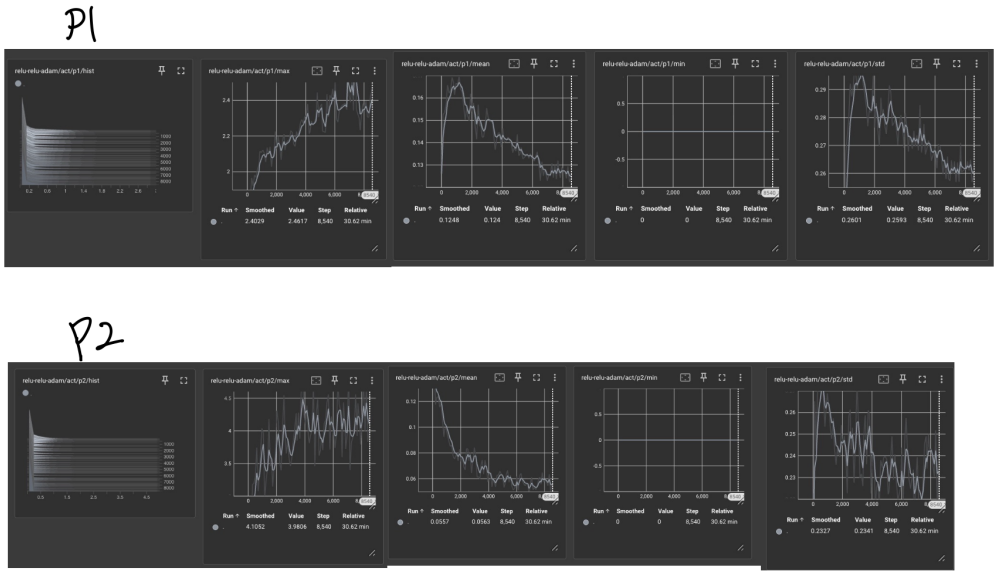


Figure 8: p statistics

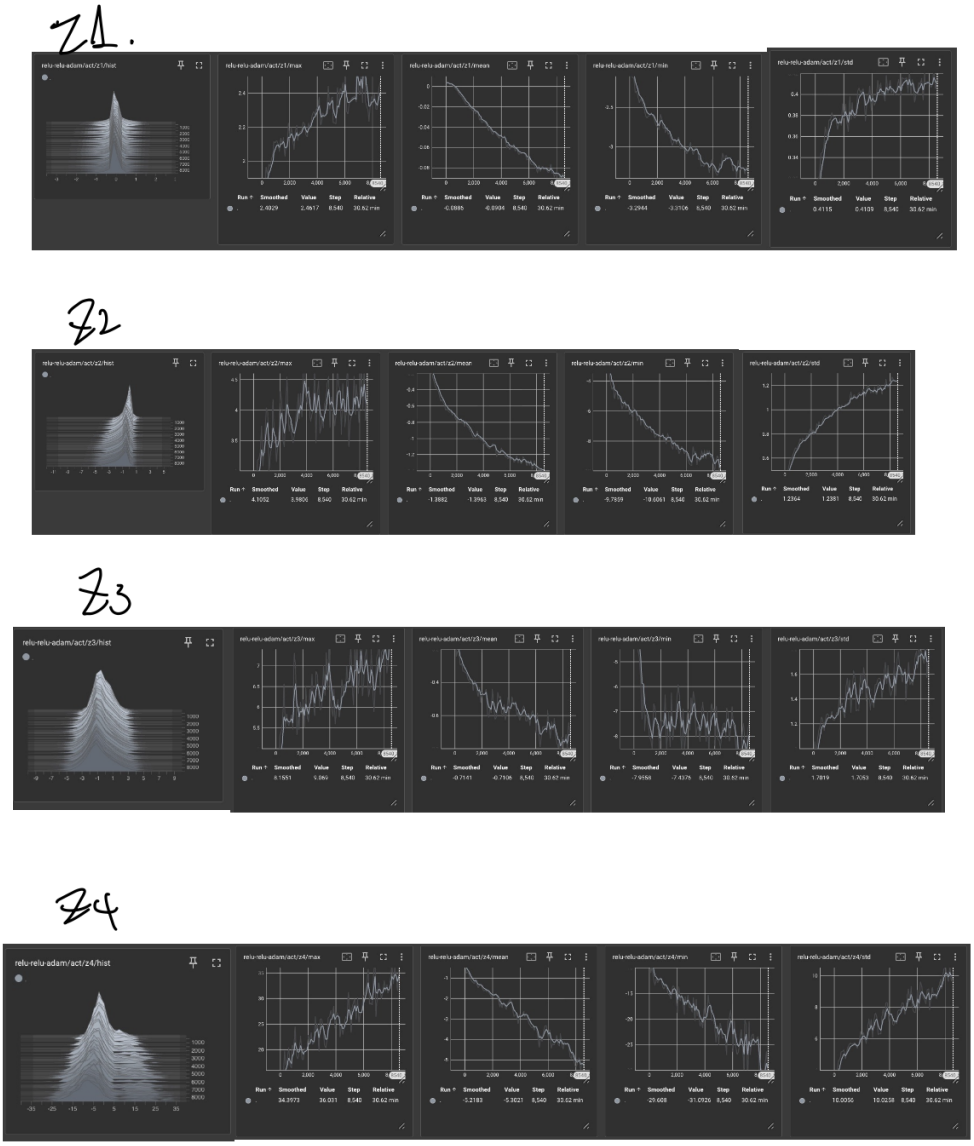


Figure 9: z statistics

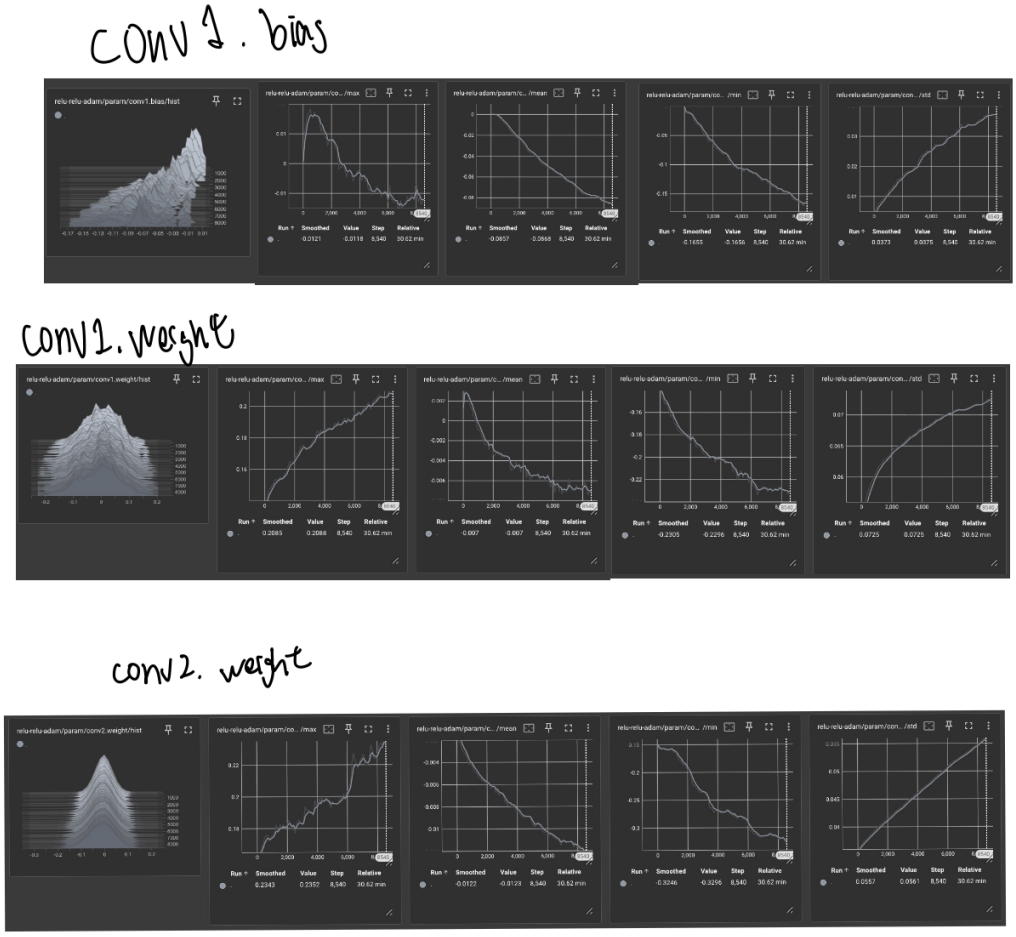


Figure 10: conv statistics

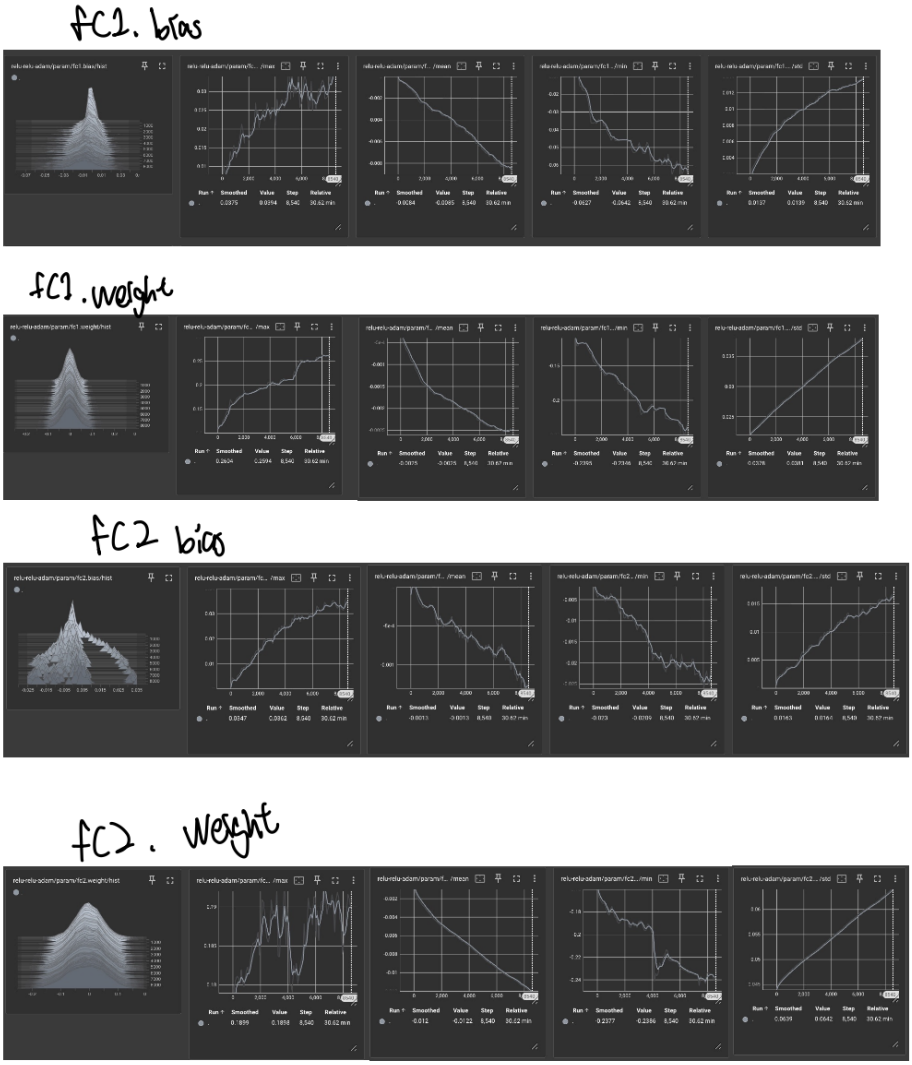


Figure 11: fc statistics

tanh-tanh-momentum results

When I switched from the ReLUReLUAdam setup to TanhTanhMomentum, I noticed that the learning behavior changed noticeably. The loss curve under the TanhMomentum configuration declined more gradually and smoothly, without the sharp early drop seen in the ReLUAdam run. The error trend was steadier, showing fewer spikes, and the overall accuracy increased at a slower pace before stabilizing close to 99%. I believe it is because tanhs smaller gradients and the momentum-based optimizer both promote smoother, more stable learning, but at the cost of slower convergence. In short, the

TanhMomentum model trained more conservatively but still reached high accuracy with reduced oscillations compared to the faster and more aggressive ReLUAdam combination.

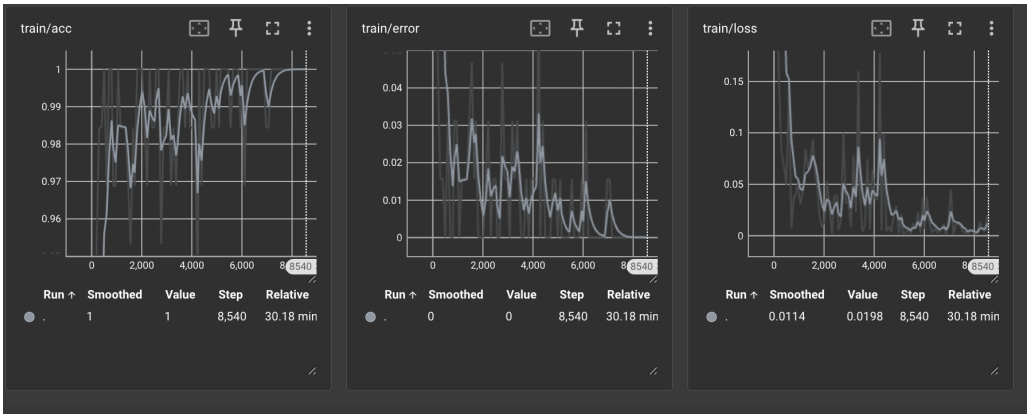


Figure 12: accuracy table result

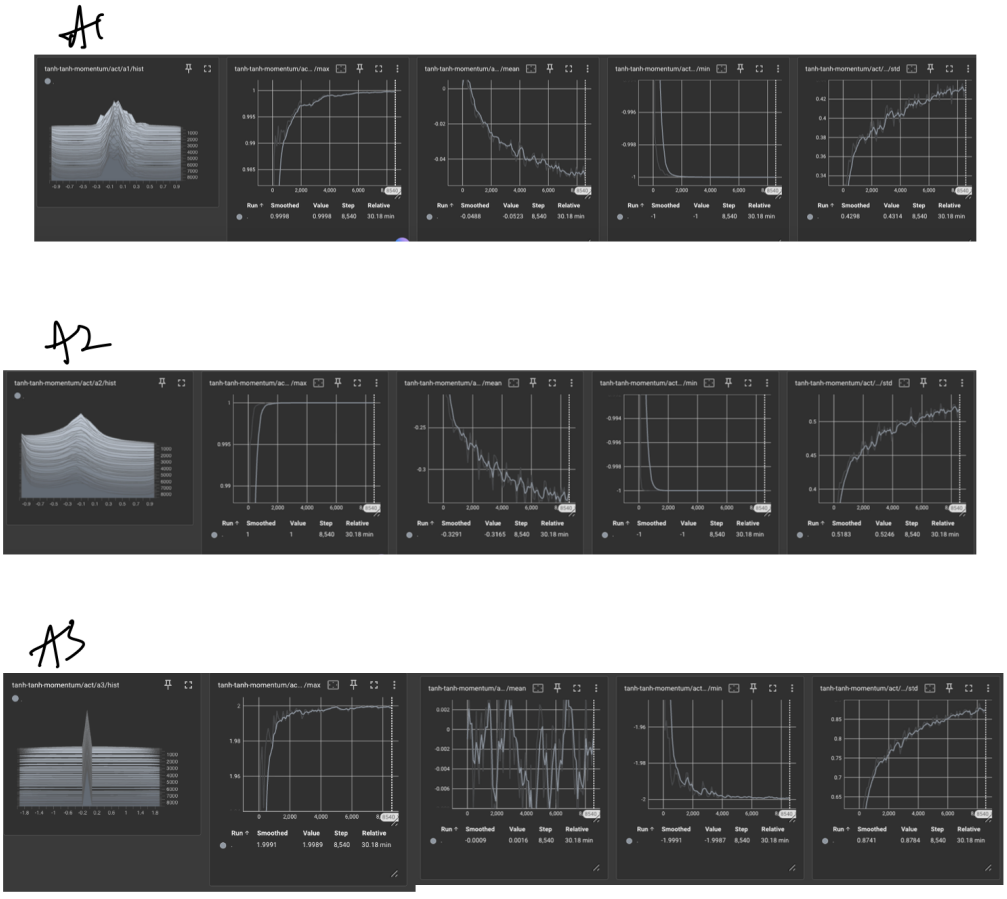


Figure 13: a statistics result

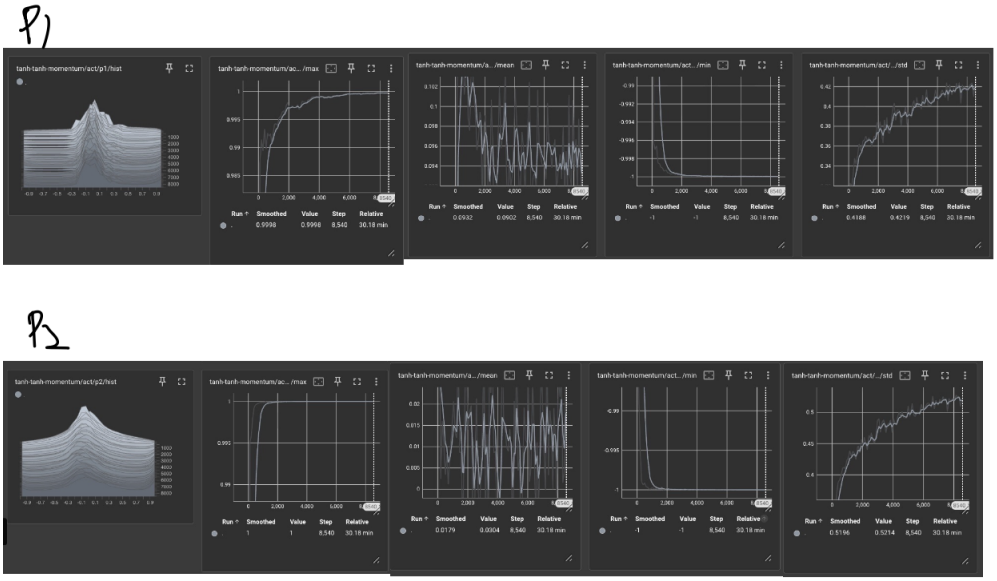


Figure 14: p result

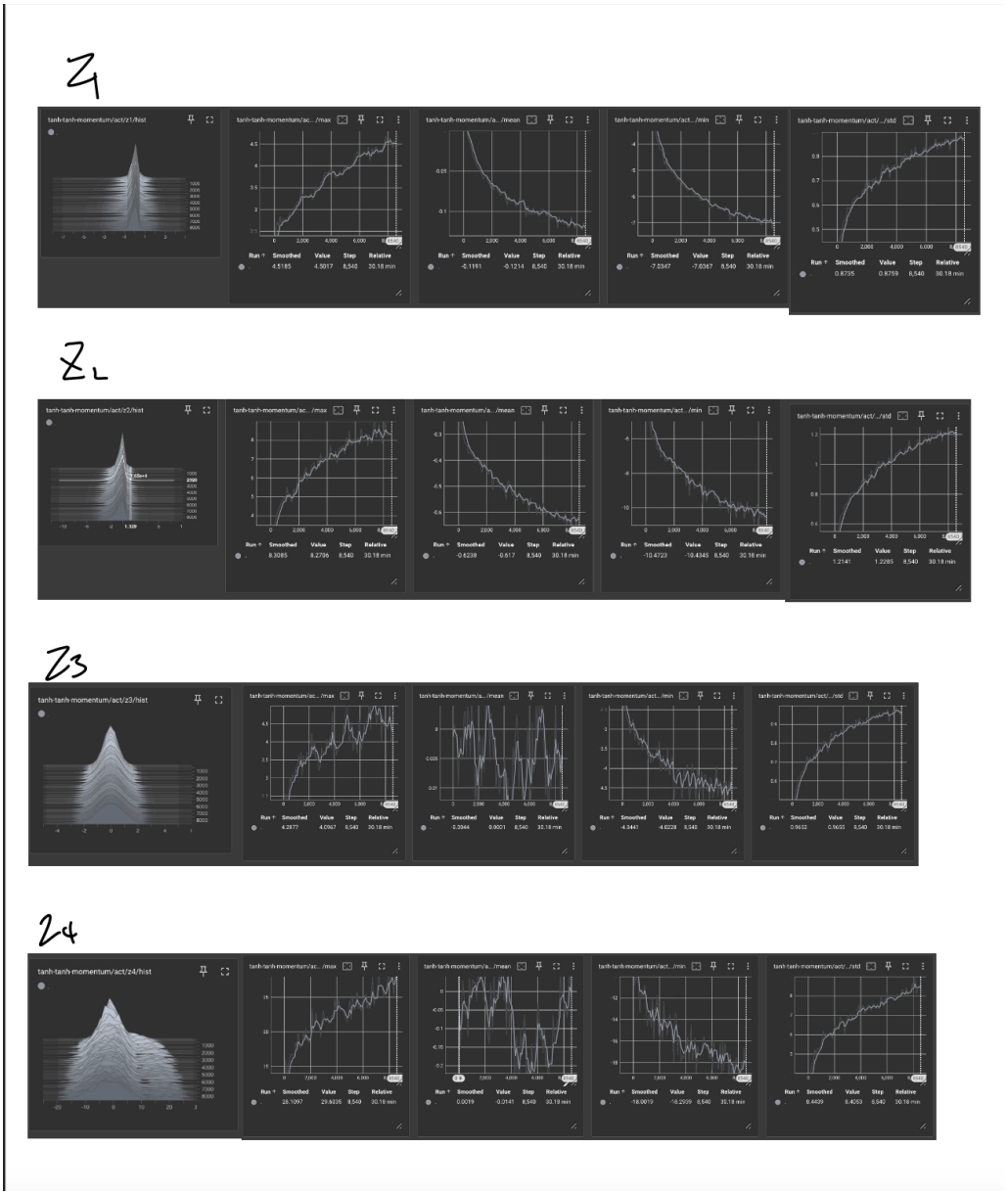


Figure 15: z statistics

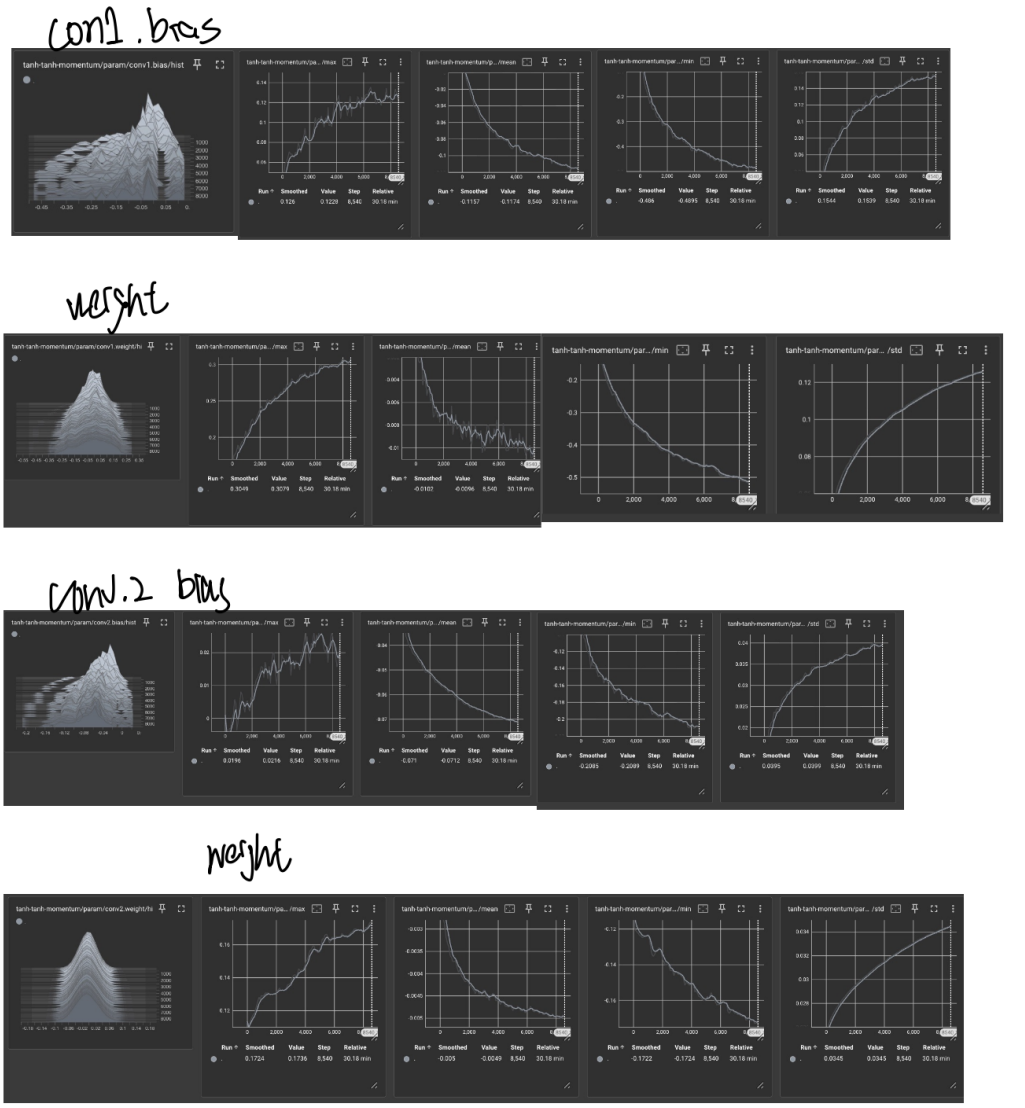


Figure 16: Conv1.bias and weight result

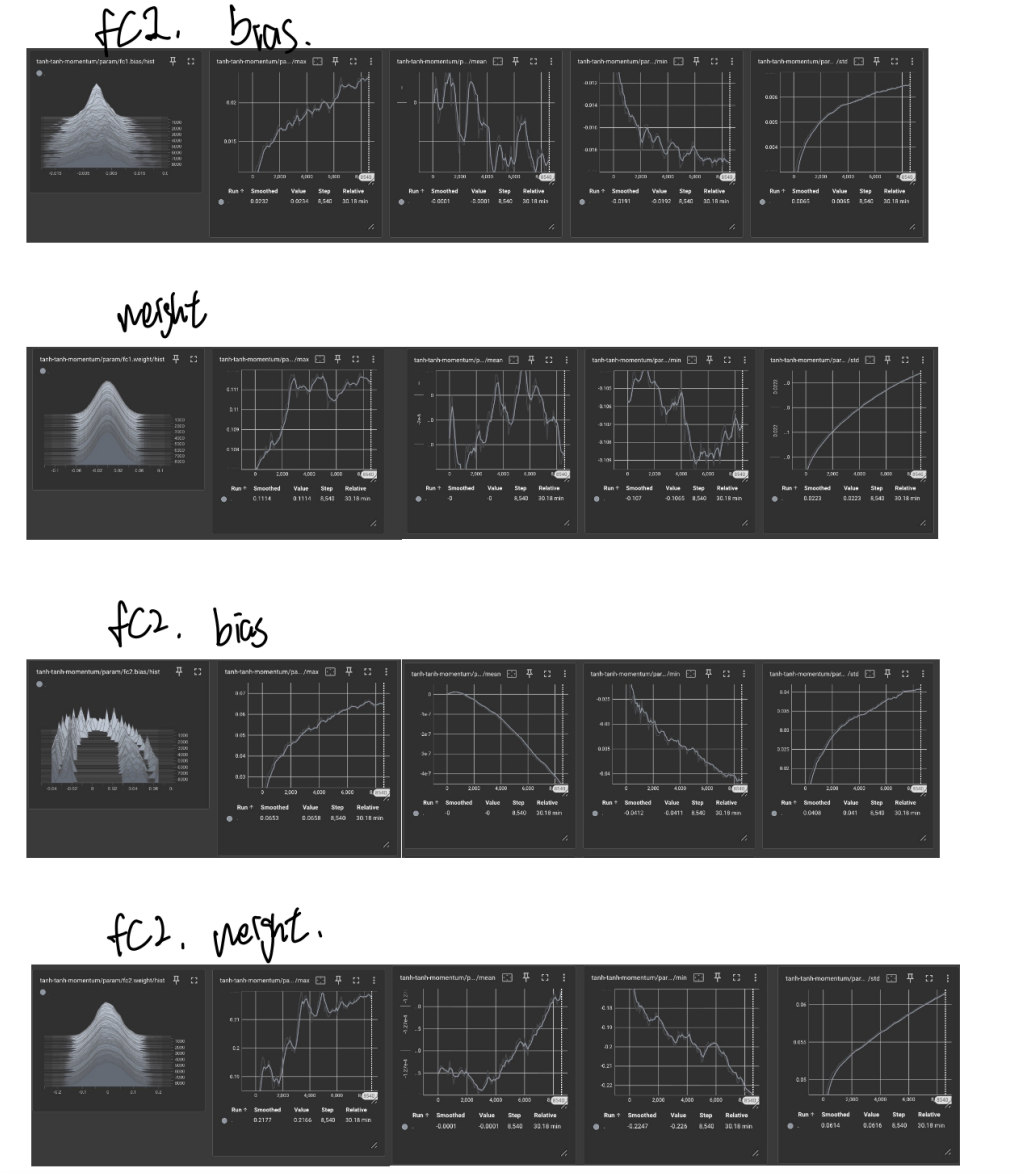


Figure 17: Fc2.bias and weight result