Robert Heeter
ELEC 576 Introduction to Deep Learning
4 October 2023

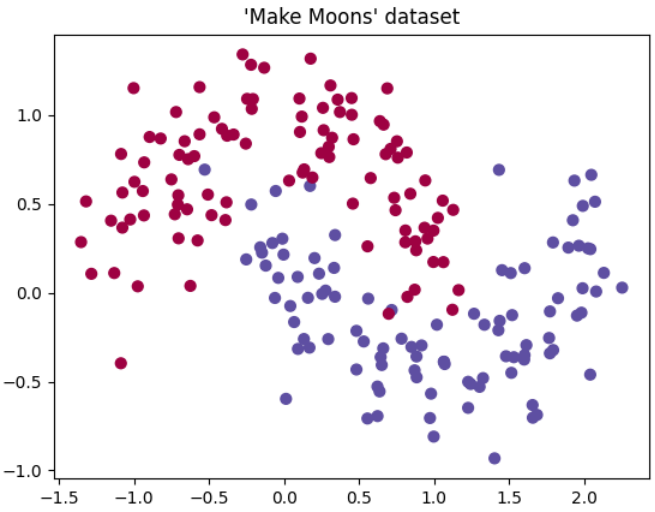
Problem Set #1

I. Backpropagation in a simple neural network

3-layer neural network

Scikit-learn's 'Make Moons' toy dataset was used to test this 3-layer neural network from scratch.

A visualization of the 'Make Moons' 2-class dataset is shown below.



The network was defined as:

- $z_1 = W_1 X + b_1$
- $a_1 = f(z_1)$, where f is an activation function
- $z_2 = W_2 a_1 + b_2$
- $a_2 = y\hat{} = softmax(z_2)$

The loss function was defined as:

- $L(y, y\hat{}) = -\frac{1}{N} \sum_{n \in N} \sum_{i \in C} y_{n,i} log(y\hat{}_{n,i})$, where y are one-hot-encoding vectors and $y\hat{}$ are vectors of probabilities

Hyperbolic tangent (tanh), sigmoid, and rectified linear unit (ReLU) activation functions were used for this neural network, defined as:

- Hyperbolic tangent (tanh):

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

- Sigmoid:
  $f(z) = \frac{1}{1+e^{-z}}$

- Rectified linear unit (ReLU):
  $f(z) = max(0, z)$, so $f(z) = z$ for $z \geq 0$ and $f(z) = 0$ for $z < 0$

The derivatives of these three activation functions were derived for backpropagation:

- Hyperbolic tangent (tanh) derivative:
  $\frac{d}{dz} f(z) = \frac{d}{dz} tanh(z) = sech^2(z) = \frac{2}{e^x + e^{-x}} = 1 - tanh^2(x) = 1 - f^2(z)$, since $sech^2(x) = 1 - tanh^2(x)$ (trigonometric identity)

- Sigmoid derivative:
  $\frac{d}{dz} f(z) = \frac{e^{-x}}{\left(1+e^{-x}\right)^2}$, which algebraically simplifies to
  $\frac{d}{dz} f(z) = \frac{1}{1+e^{-x}} \cdot \left(1 - \frac{1}{1+e^{-x}}\right) = f(z)(1 - f(z))$

- Rectified linear unit (ReLU) derivative:
  $\frac{d}{dz} f(z) = 1$ for $z \geq 0$ and $f(z) = 0$ for $z < 0$

For backpropagation, the following gradients were derived and implemented:

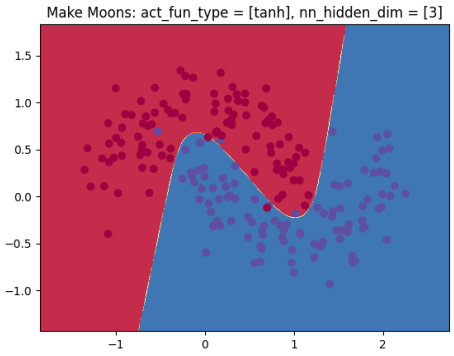
$\Delta_3 = a_2 - y$

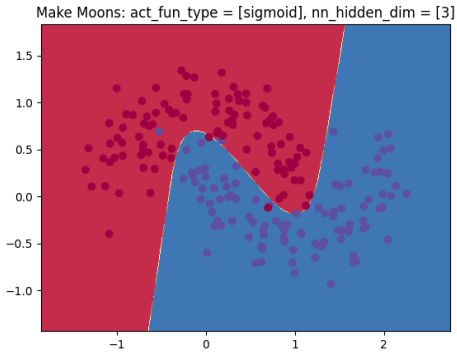
- $\frac{\partial L}{\partial W_2} = a_1^T \cdot \Delta_3$, where W_2, a_1, Δ_3 are matrices or vectors
- $\frac{\partial L}{\partial b_2} = \Sigma \Delta_3$, where b_2, Δ_3 are vectors

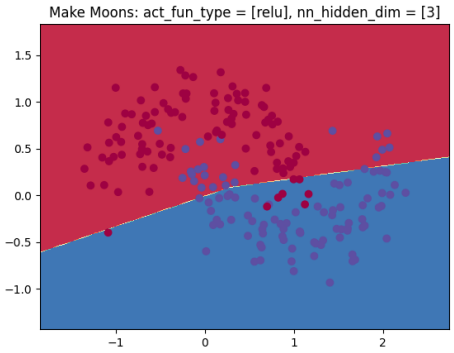
$\Delta_2 = \Delta_3 \cdot W_2^T \times \frac{d}{dz_1} f(z_1)$, where f is an activation function

- $\frac{\partial L}{\partial W_1} = X^T \cdot \Delta_2$, where W_1, X, Δ_2 are matrices or vectors
- $\frac{\partial L}{\partial b_1} = \Sigma \cdot \Delta_2$, where b_1, Δ_2 are vectors

Using 3 hidden dimensions (i.e. nodes/units) in the hidden layer, varying the type of decision boundary (tanh, sigmoid, ReLU) produced the decision boundaries shown below for the Make Moons dataset.

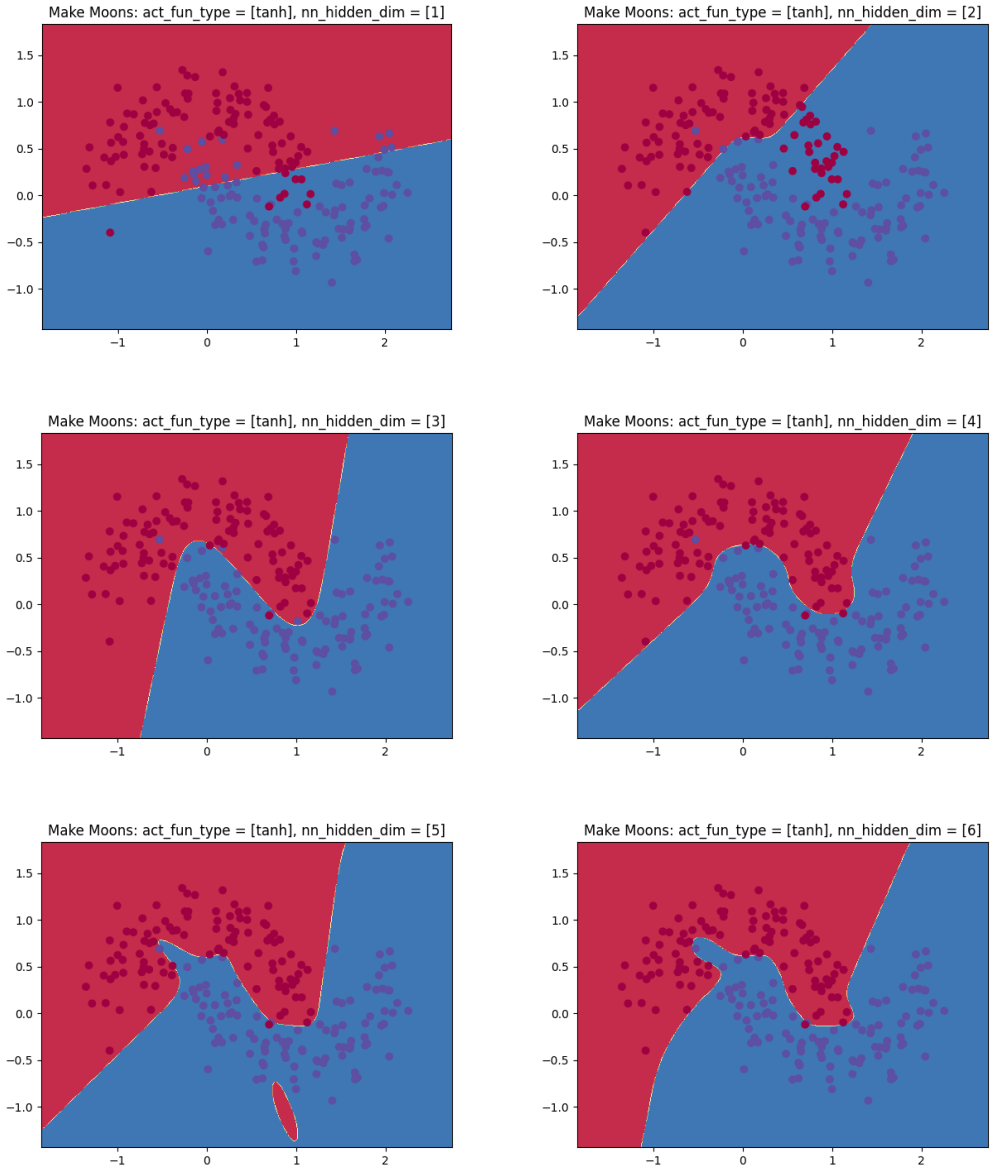






Upon visual inspection, the hyperbolic tangent (tanh) and sigmoid activation functions produce a very similar decision boundary. The decision boundary for both is smooth, reflecting the fact that both activation functions are smooth curves. The rectified linear unit (ReLU) activation function produced a piecewise decision boundary that does not fit the data as well as those from the tanh or sigmoid activation functions given the same number of hidden dimensions (i.e. nodes/units), which was set as 3 for these comparisons.

Using a hyperbolic tangent activation function, varying the number of hidden dimensions from 1 to 6 produced the decision boundaries shown below for the Make Moons dataset.

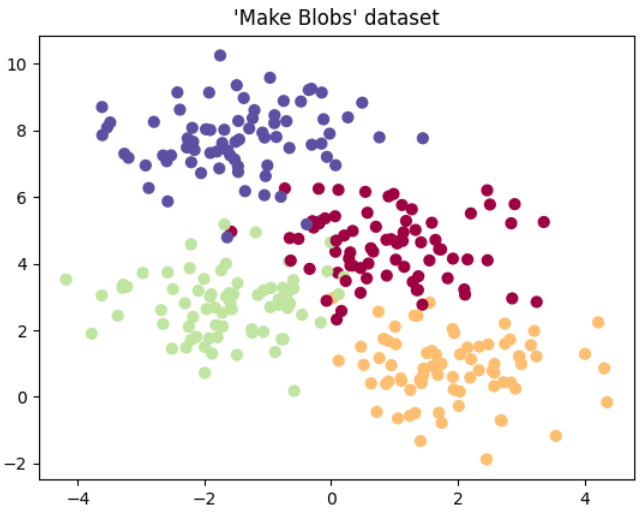


Upon visual inspection, using fewer than 3 hidden dimensions (i.e. nodes/units) for this classification task does not produce a good decision boundary (i.e. the model is underfit). Training the network with 3-4 hidden dimensions produces the most reasonable decision boundary; the model is not fit to noise in the data (i.e. is reasonably fit) and follows the moon-shaped contour of both data clusters. With more than 4 hidden dimensions, the model starts to fit to the noise in the data (i.e. the model is overfit).

N-layer neural network

Scikit-learn's 'Make Moons' and 'Make Blobs' toy datasets were used to test this N-layer neural network from scratch.

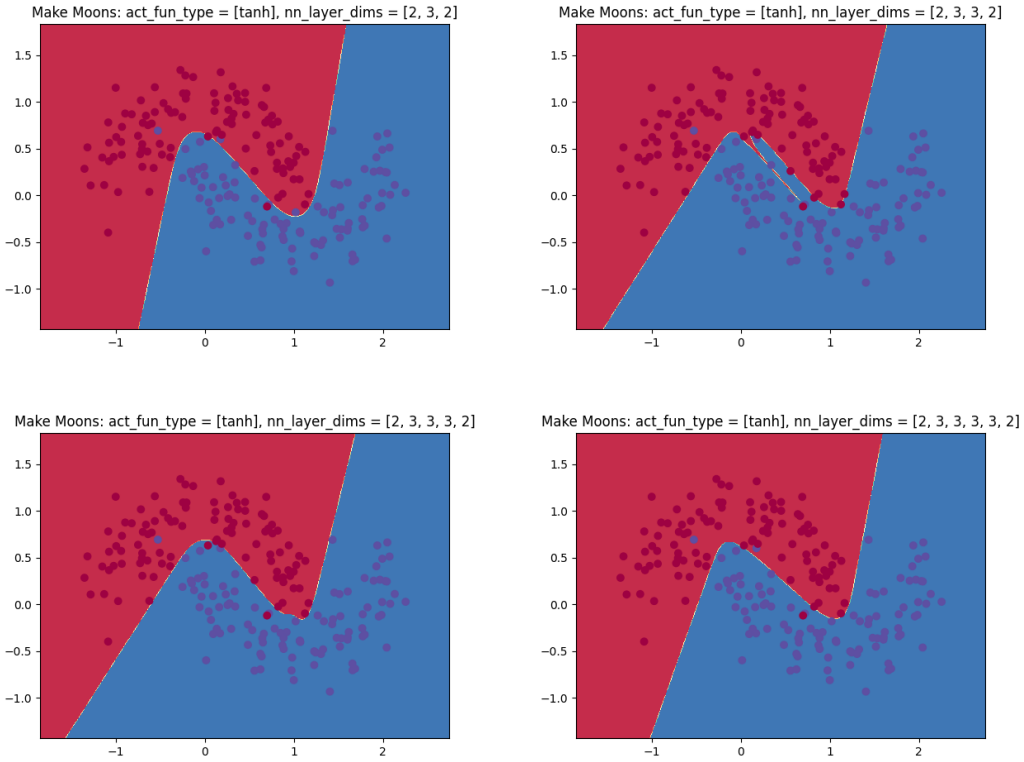
A visualization of the 'Make Blobs' 4-class dataset is shown below.



A new class DeepNeuralNetwork was created that inherits the NeuralNetwork class but modifies its feedforward(), calculate_loss(), backpropagation(), fit_model(), and visualize_decision_boundary() functions. The class generalizes the functionality of NeuralNetwork by taking in a list of layer dimensions as nn_layer_dims. For example, nn_layer_dims = [2,3,3,2] will create a network with 2 hidden layers, each with 3 dimensions (i.e. nodes/units) and the input and output layers will both have 2 dimensions. The DeepNeuralNetwork class uses lists to store the learned weights (W) and biases (b), which it iterates through for the feedforward, backpropagation, and gradient descent steps rather than assigning those and intermediate values (i.e. z, a) to specific variables. This approach was chosen over defining a separate 'Layer' class for overall simplicity (fewer required functions and classes) though special care was required for indexing and updating the weights, biases, and other network parameters.

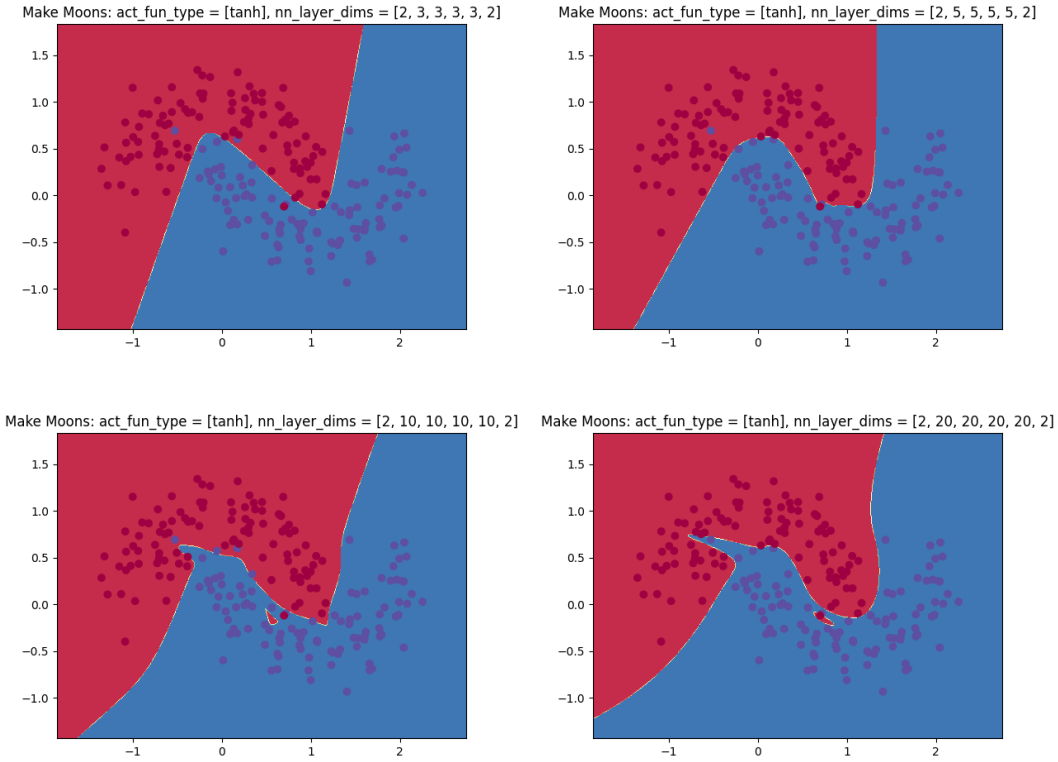
Implementing this class for a given dataset and visualizing the learned decision boundary follows the same methodology as developed in NeuralNetwork. The same network architecture (i.e. loss function and activation functions) were used in the DeepNeuralNetwork class as well.

Using a hyperbolic tangent activation function, varying the number of 3-dimensional hidden layers from 1 to 4 produced the decision boundaries shown below for the Make Moons dataset.



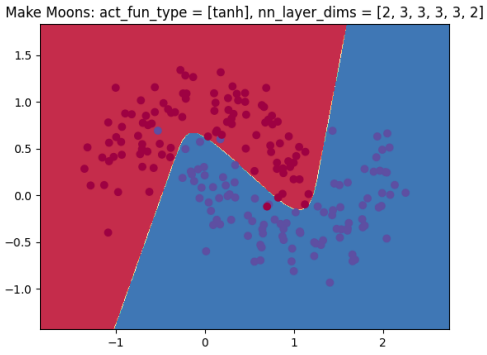
Upon visual inspection, for this simple data set (only 2 features and a trivial decision boundary), the single hidden layer produced the best result with the lowest computational expense. There appeared to be some overfitting in the case of the double hidden layer, where the decision boundary sharply angles to fit noise in the data. Adding additional hidden layers did not seem to produce any better results.

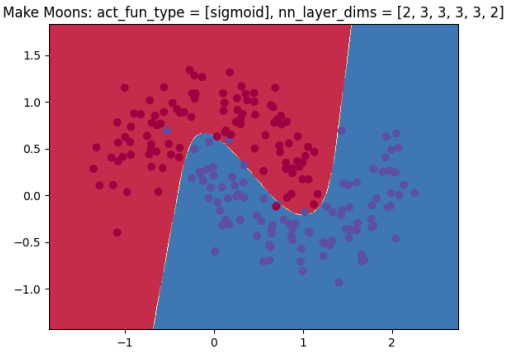
Using a hyperbolic tangent activation function with 4 hidden layers, varying the number of hidden dimensions in each hidden layer from 3 to 20 produced the decision boundaries shown below for the Make Moons dataset.

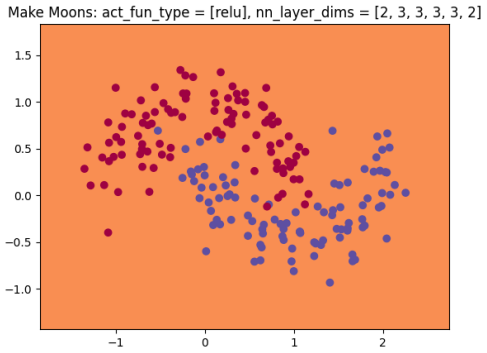


Upon visual inspection, the lower-dimensional hidden layers produced better results (i.e. the 3-dimensional hidden layer case). With higher hidden dimensionality (i.e. neural network width or number of nodes/units), the model became overfitted; the decision boundary was less smooth as noise in the data was fitted. For example, above 10 hidden dimensions in the 4 hidden layers, the model produced an 'island' boundary around one of the outlying data points.

Using 3 hidden dimensions (i.e. nodes/units) in each of 4 hidden layers, varying the type of activation function (tanh, sigmoid, ReLU) produced the decision boundaries shown below for the Make Moons dataset.

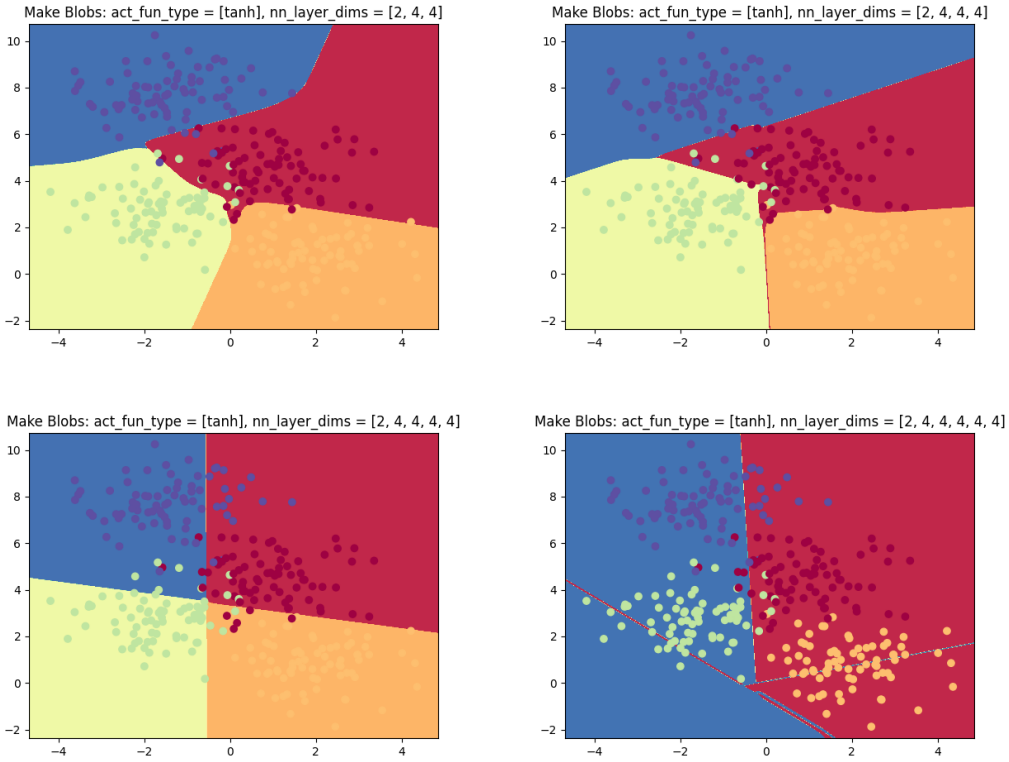






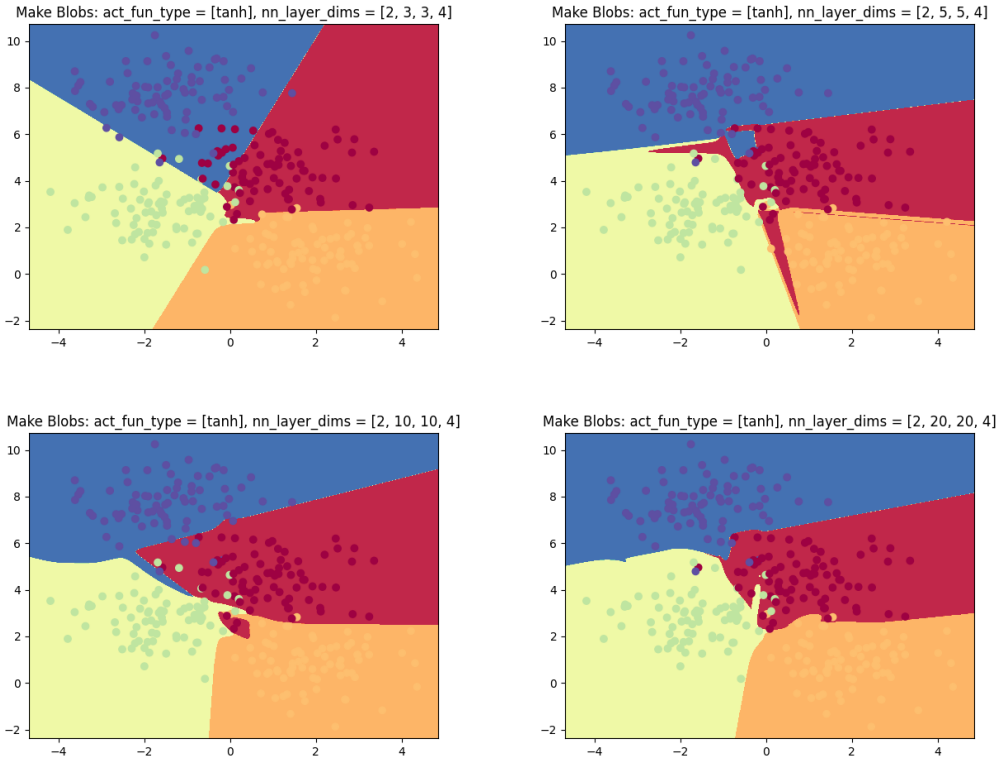
Upon visual inspection, the hyperbolic tangent (tanh) and sigmoid activation functions produced a very similar decision boundary as seen in the 3-layer neural network example. The rectified linear unit (ReLU) activation function did not produce a decision boundary, perhaps due to a shallow gradient or perhaps due to strong regularization.

Using a hyperbolic tangent activation function, varying the number of 4-dimensional hidden layers from 1 to 4 produced the decision boundaries shown below for the Make Blobs dataset.



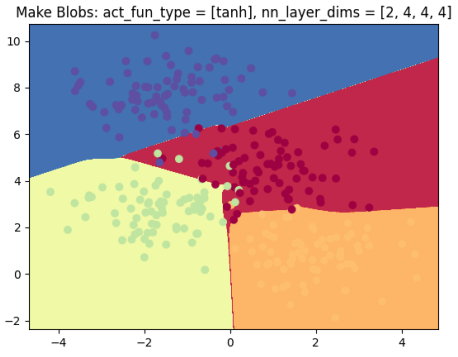
Upon visual inspection, as in the case with the Make Moons dataset, with this relatively simple dataset it seems that fewer hidden layers produced better decision boundaries between the 4 classes. As the number of hidden layers increased, the model appeared to have difficulty in gradient descent, likely due to the high number of tunable parameters relative to the small set of data.

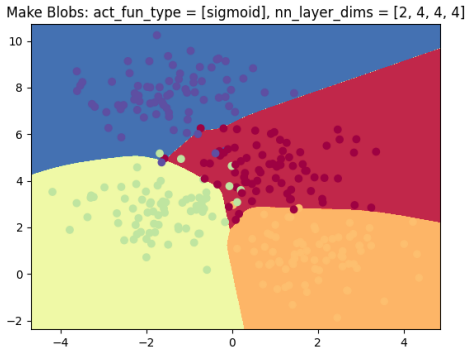
Using a hyperbolic tangent activation function with 2 hidden layers, varying the number of hidden dimensions in each hidden layer from 3 to 20 produced the decision boundaries shown below for the Make Blobs dataset.

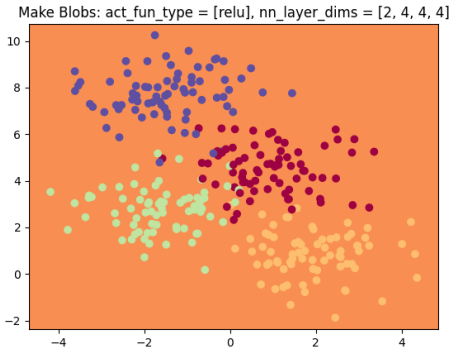


Upon visual inspection, the lower-dimensional hidden layers produced better results (i.e. the 3-dimensional hidden layer case). With higher hidden dimensionality (i.e. neural network width or number of nodes/units), the model became overfitted; the decision boundary was less smooth as noise in the data was fitted. For example, above 10 hidden dimensions in the 2 hidden layers, the model produced multiple 'island' boundaries around outlying data points, in addition to sharp 'cutouts' in the regions of cluster overlap.

Using 4 hidden dimensions (i.e. nodes/units) in each of 2 hidden layers, varying the type of activation function (tanh, sigmoid, ReLU) produced the decision boundaries shown below for the Make Blobs dataset.







Upon visual inspection, the hyperbolic tangent (tanh) and sigmoid activation functions produce a very similar decision boundary as seen in the Make Moon dataset example. Again, the rectified linear unit (ReLU) activation function did not produce a decision boundary, perhaps due to a shallow gradient or perhaps due to strong regularization for this deep neural network with only 4-dimensional hidden and output layers.

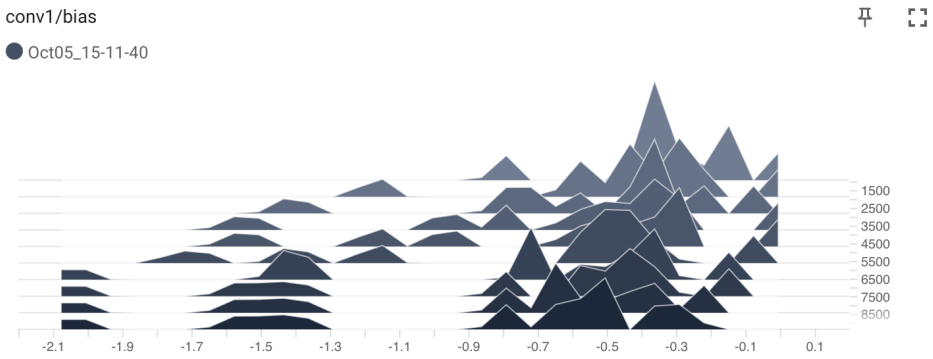
II. Training a simple deep convolutional network on MNIST

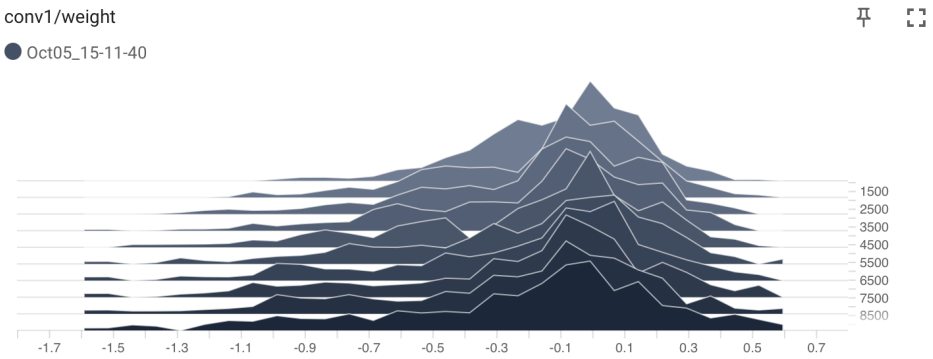
The MNIST dataset was used for training a deep convolutional network built with PyTorch. The handwritten digit images of the numbers 0 to 9 are 28x28 pixels.

A 5-layer deep convolutional network was designed with the following structure in PyTorch using an Adam optimizer and NLLLoss loss criterion:

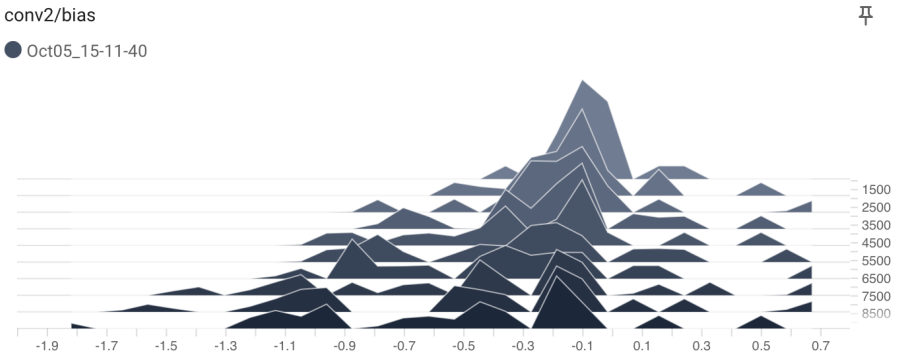
- Convolutional layer 1 (Conv1): 5-5-1-10
- Rectified Linear Unit (ReLU)
- Max-Pooling: 2-2
- Convolutional layer 2 (Conv2): 5-5-10-20
- Rectified Linear Unit (ReLU)
- Max-Pooling: 2-2
- Fully Connected (FC): 320-50
- Rectified Linear Unit (ReLU)
- Dropout: 0.5
- Fully Connected (FC): 50-10
- Softmax: 10

TensorBoard was used for visualizing and monitoring the training process for 10 epochs. The TensorBoard figures for the layer weights and biases, losses, and test accuracy are shown below.

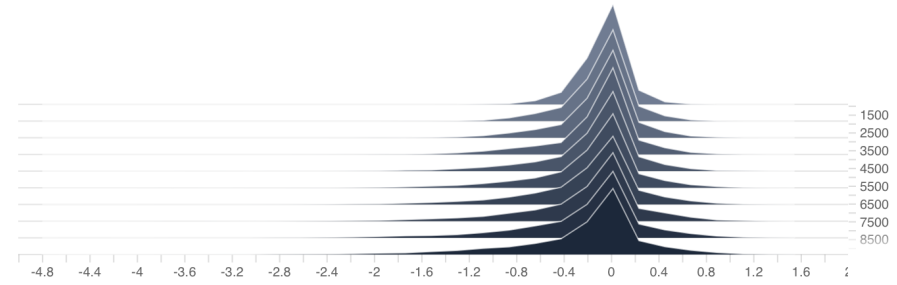




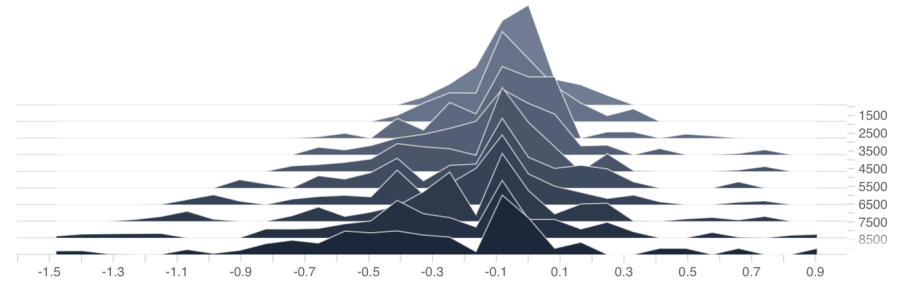
conv2/bias



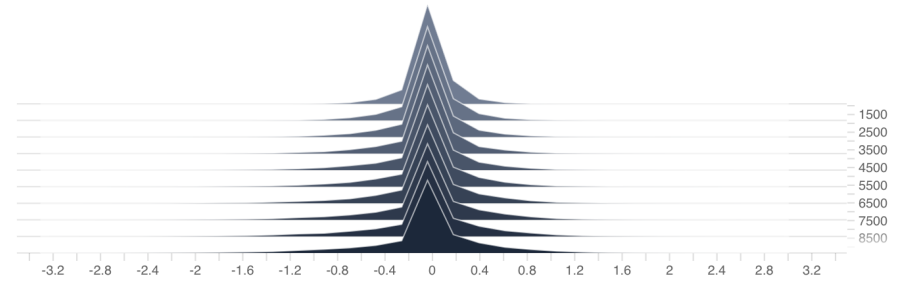
conv2/weight



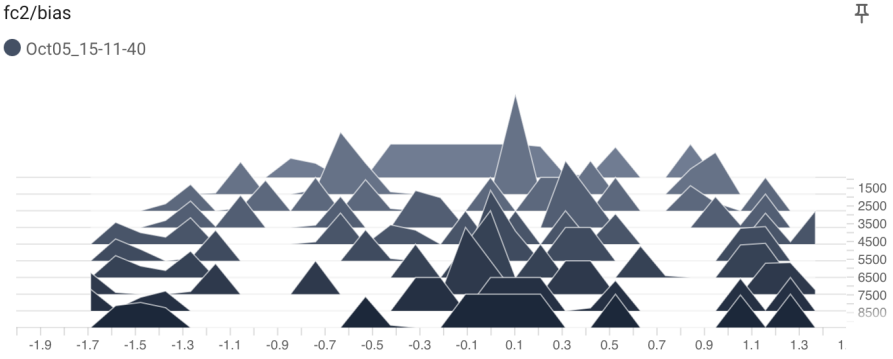
fc1/bias



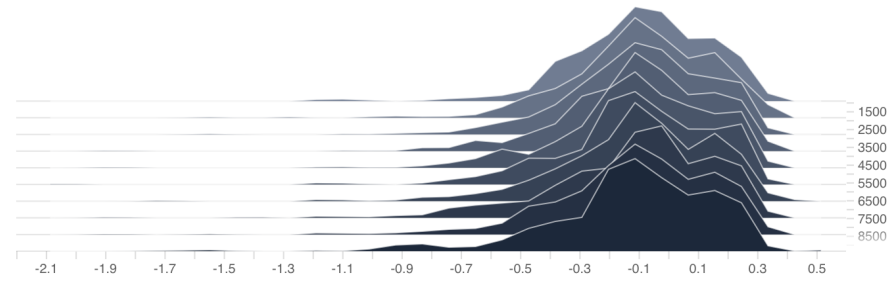
fc1/weight



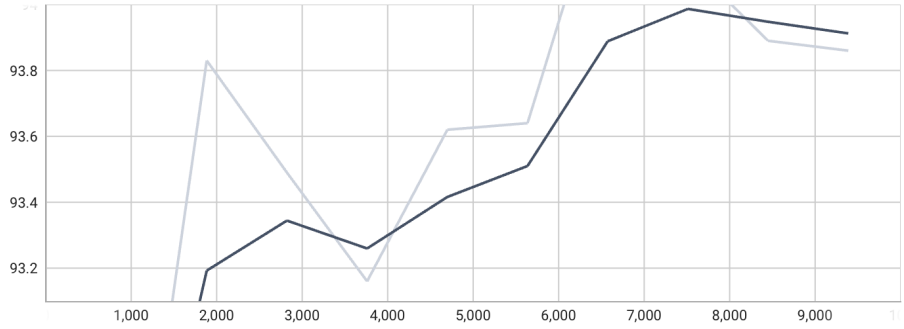
fc2/bias



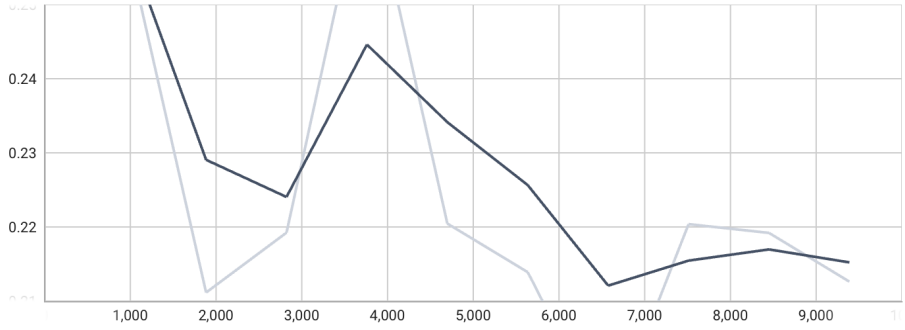
fc2/weight

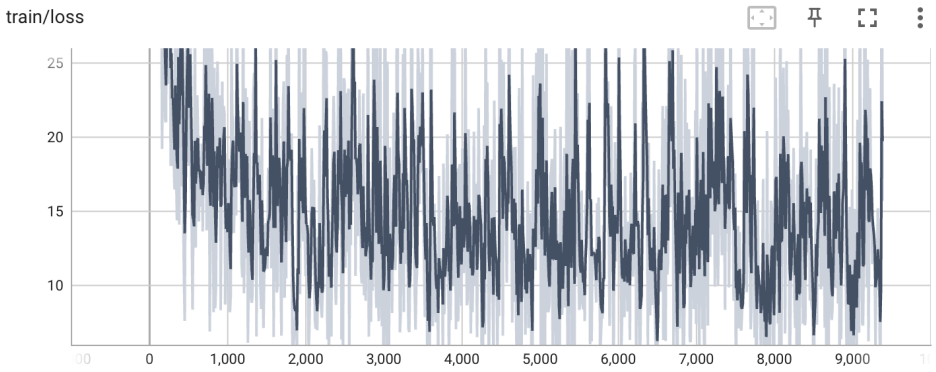


test/accuracy



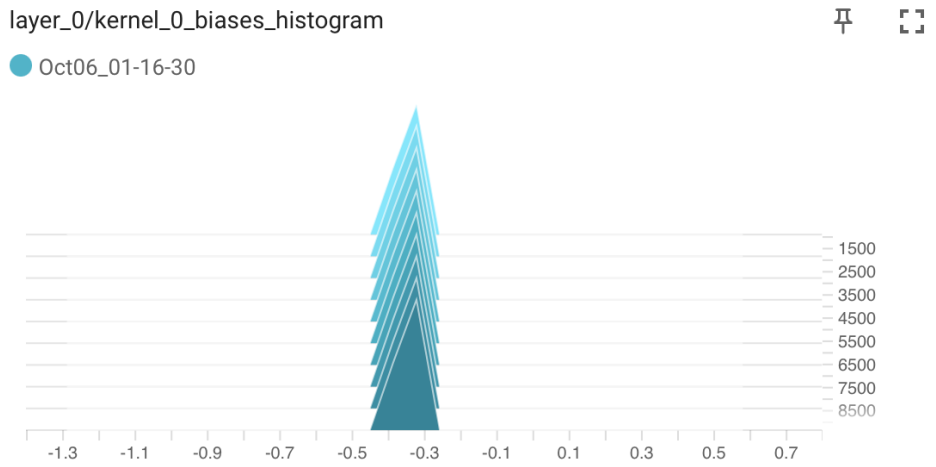
test/loss



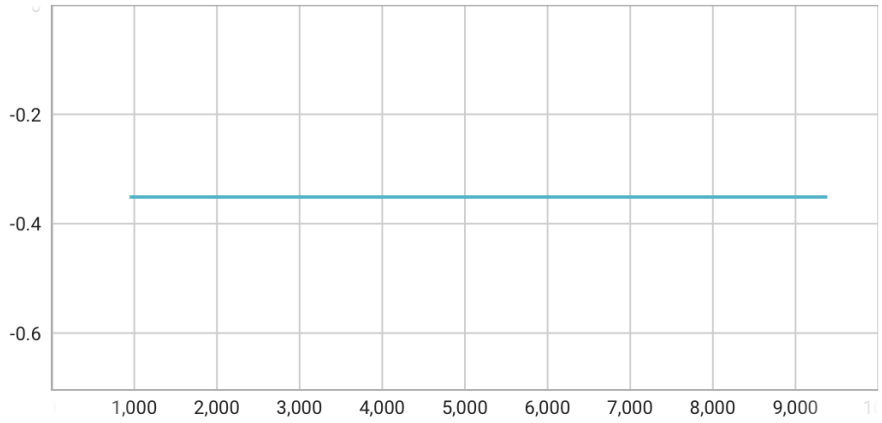


Over 10 epochs (~9,000 iterations), the test and train losses both gradually decreased, albeit with a lot of noise/fluctuation. The test accuracy increased slightly from 93.2% to 93.8%. With more epochs the accuracy may improve further based on the observed trend.

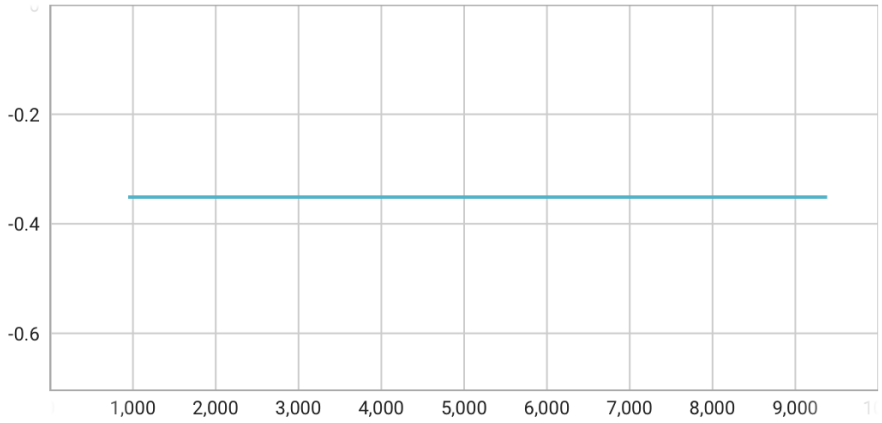
Additional statistics for the min, max, mean, standard deviation, and histogram of the weights, biases, net inputs at each layer, activations after ReLU at each layer, and activations after max-pooling at each layer every 100 iterations are shown below. Note that the convolutional layers (Conv2D layers) used multiple kernels, and only the weights and biases from the first kernel (kernel_0) were used for these figures. Layers 0 and 3 represent the two convolutional layers (Conv2D layers) while layers 7 and 10 represent the two linear (fully connected) layers.



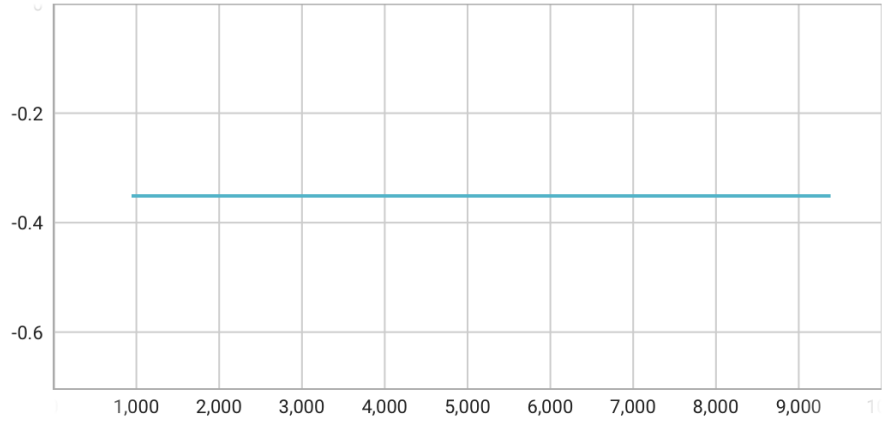
layer_0/kernel_0_biases_max



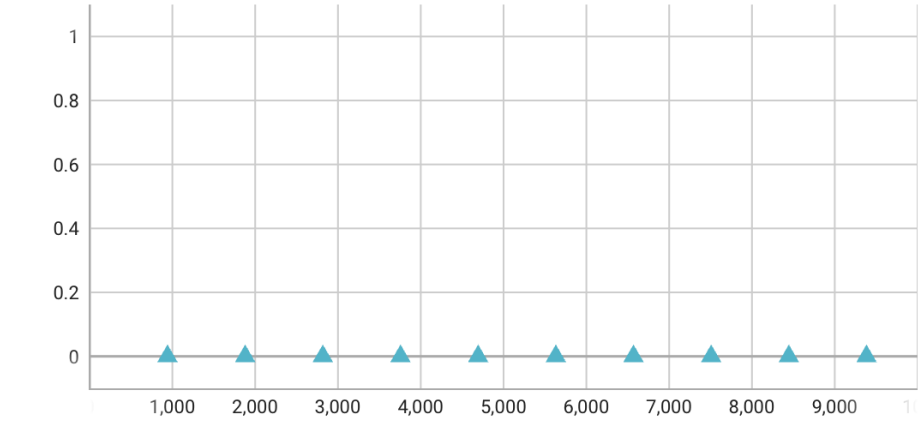
layer_0/kernel_0_biases_mean



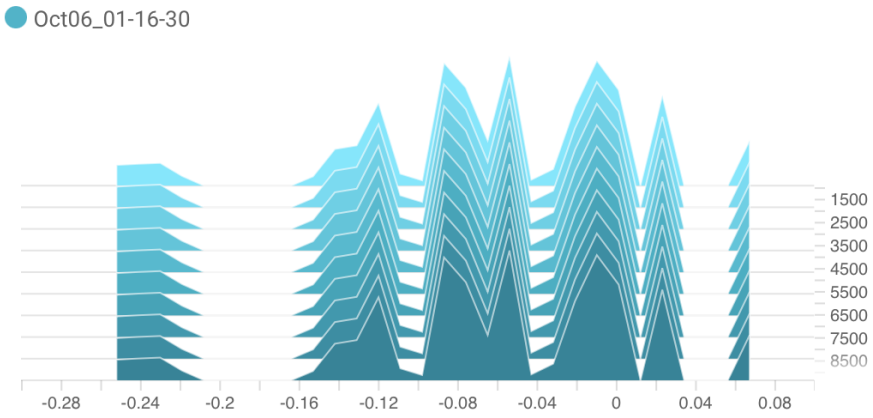
layer_0/kernel_0_biases_min



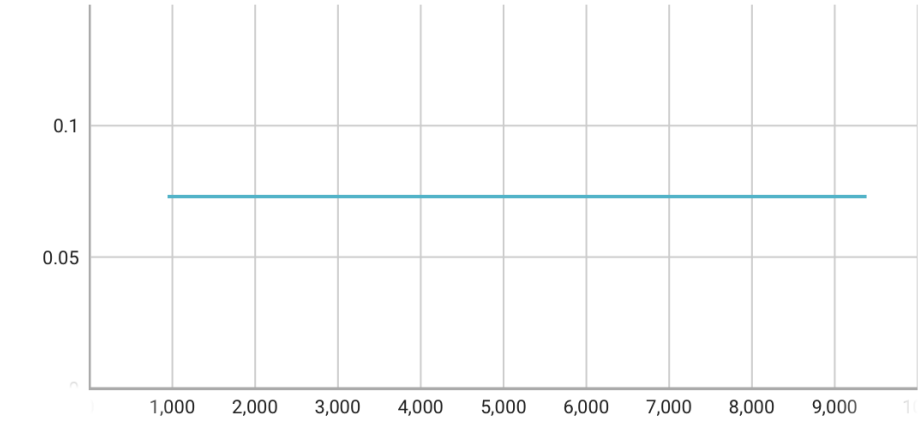
layer_0/kernel_0_biases_stdev



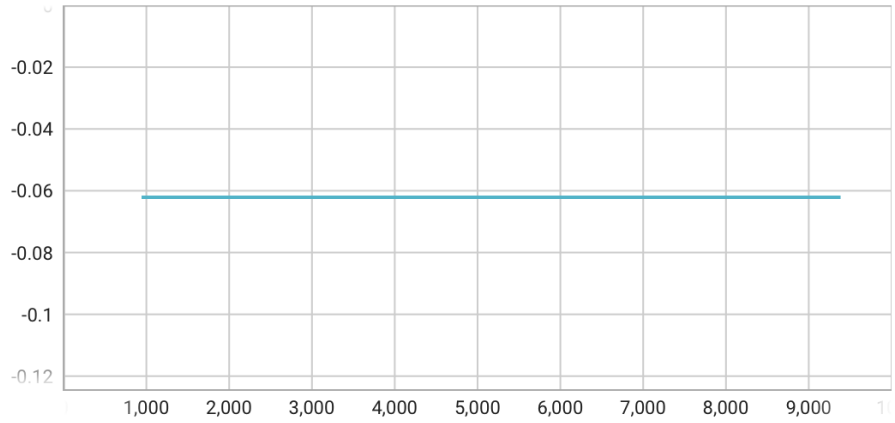
layer_0/kernel_0_weight_histogram



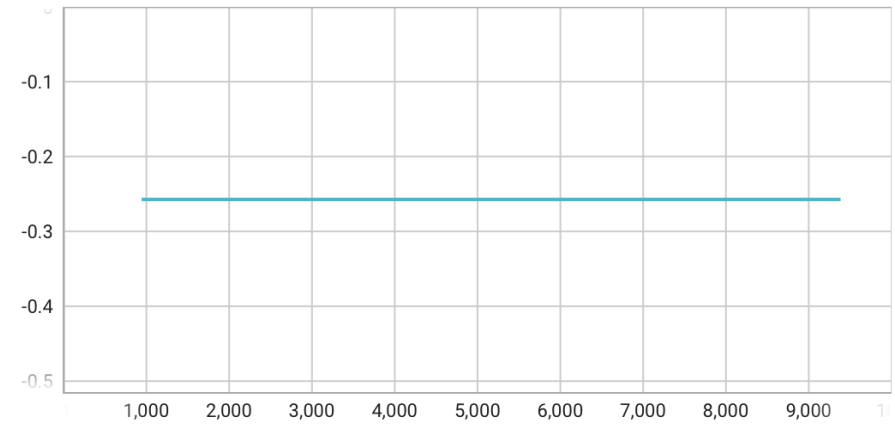
layer_0/kernel_0_weight_max



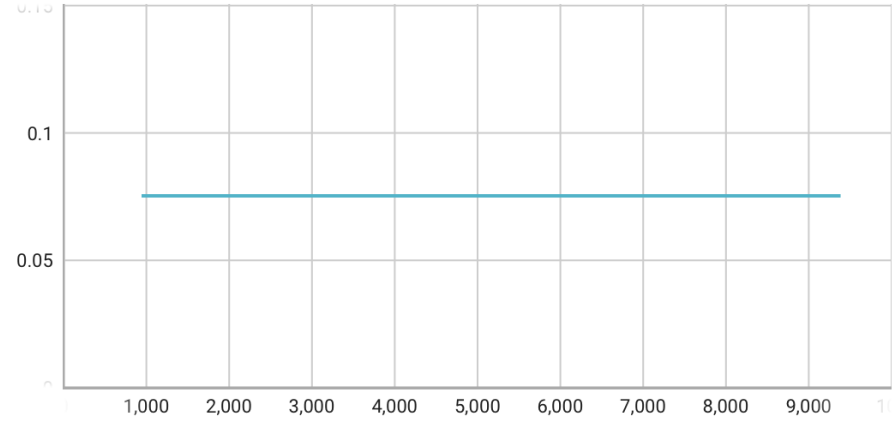
layer_0/kernel_0_weight_mean



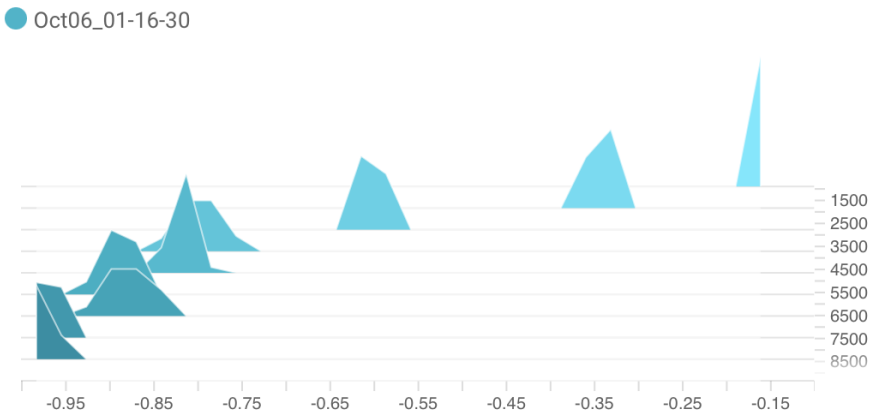
layer_0/kernel_0_weight_min



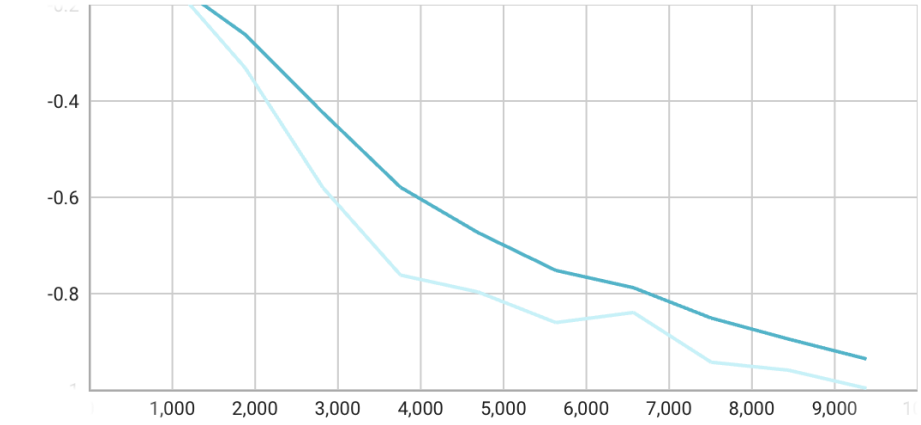
layer_0/kernel_0_weight_stdev



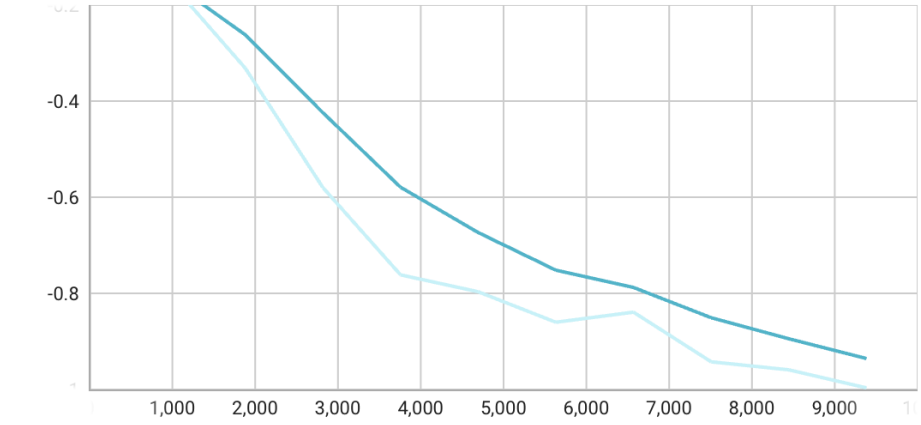
layer_3/kernel_0_biases_histogram



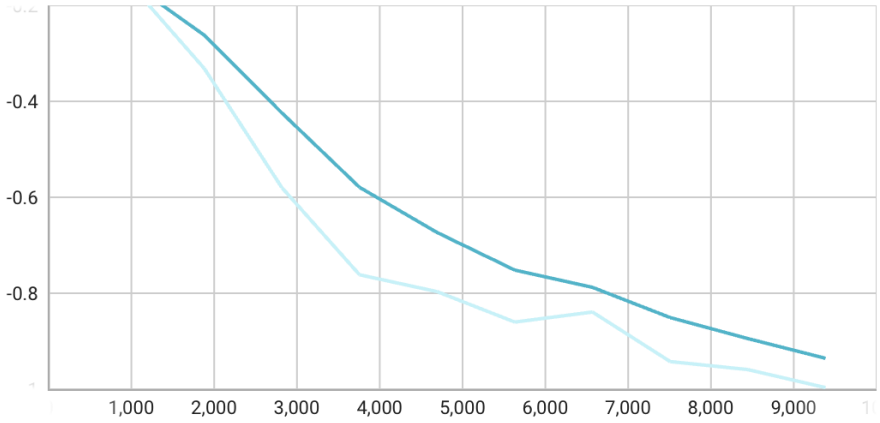
layer_3/kernel_0_biases_max



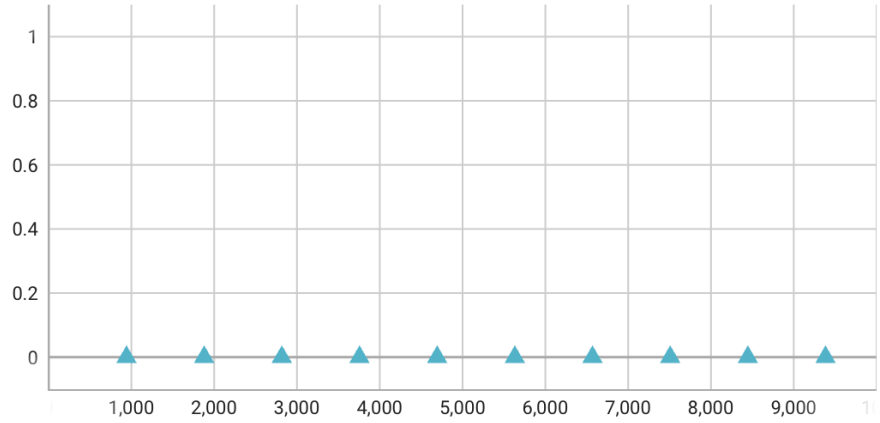
layer_3/kernel_0_biases_mean



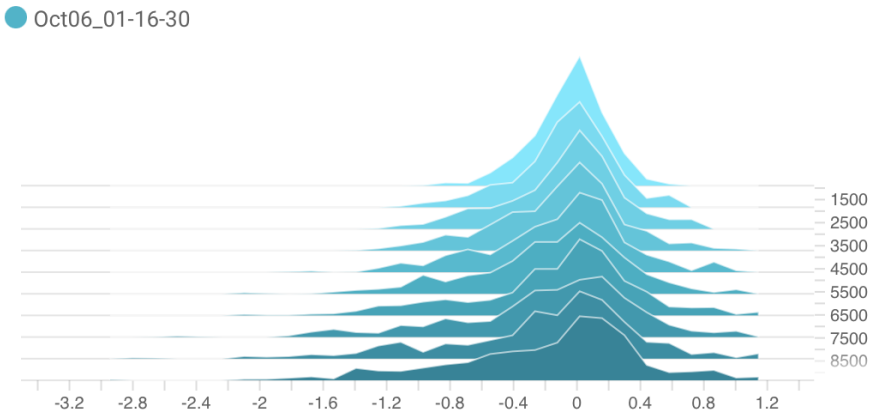
layer_3/kernel_0_biases_min



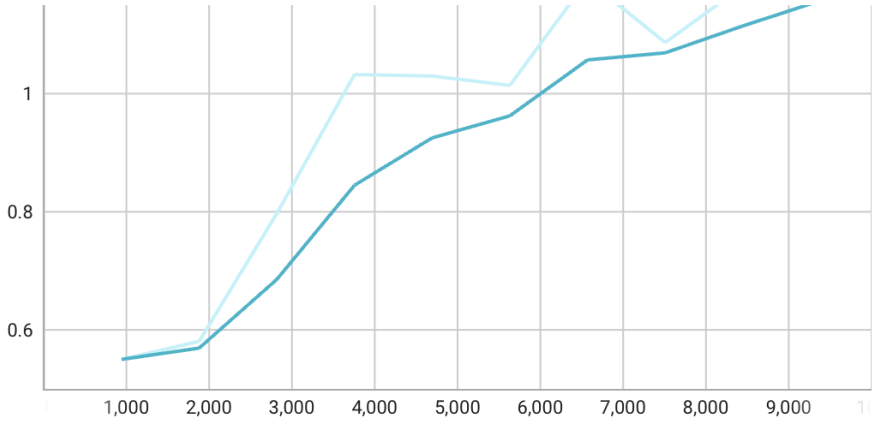
layer_3/kernel_0_biases_stdev



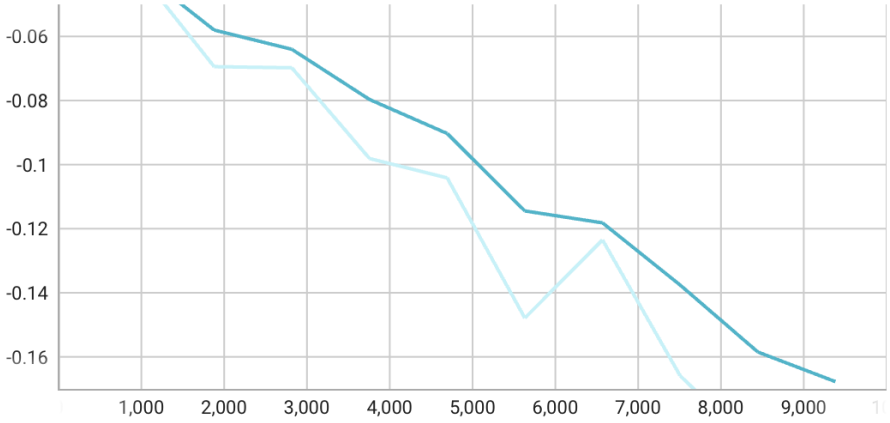
layer_3/kernel_0_weight_histogram



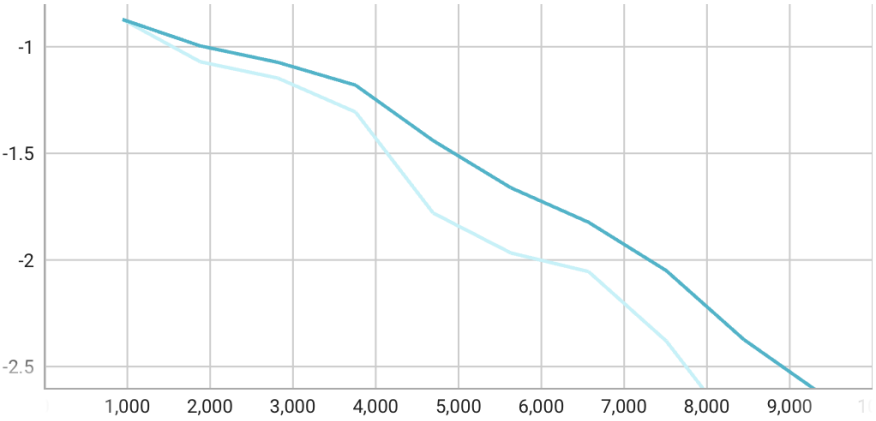
layer_3/kernel_0_weight_max



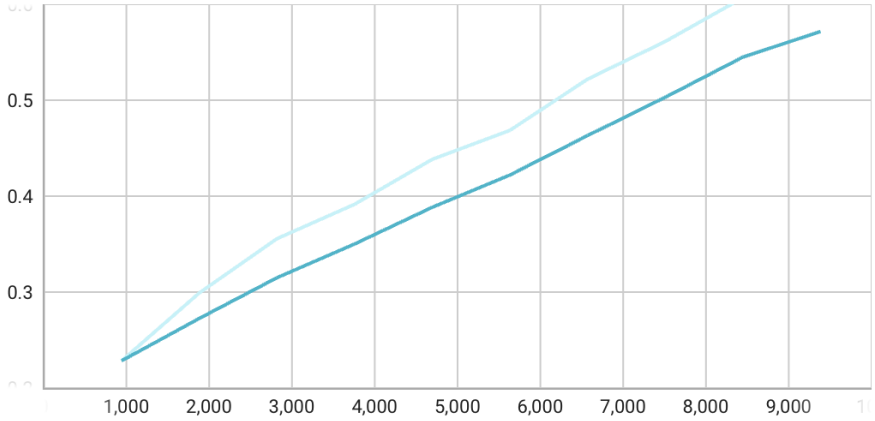
layer_3/kernel_0_weight_mean



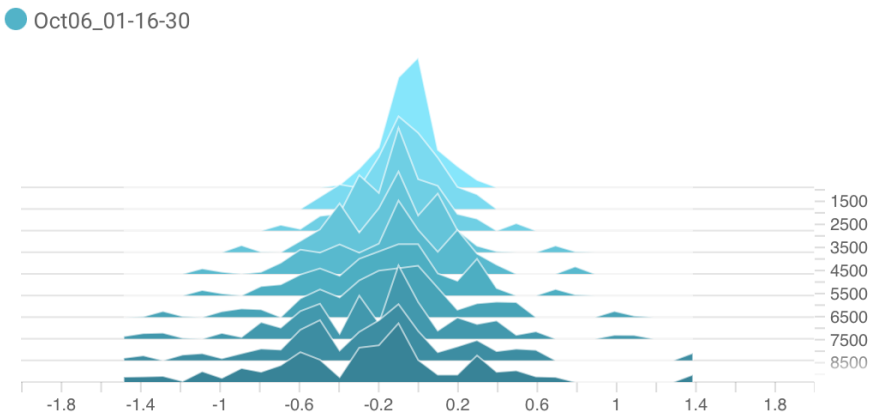
layer_3/kernel_0_weight_min



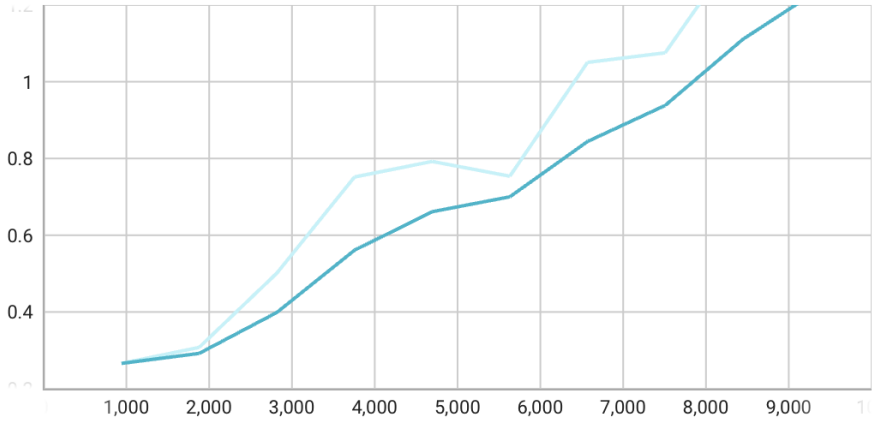
layer_3/kernel_0_weight_stdev



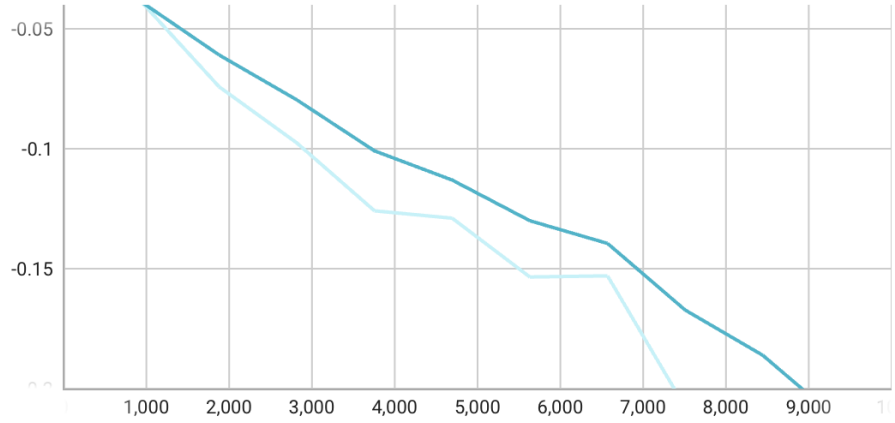
layer_7_biases_histogram



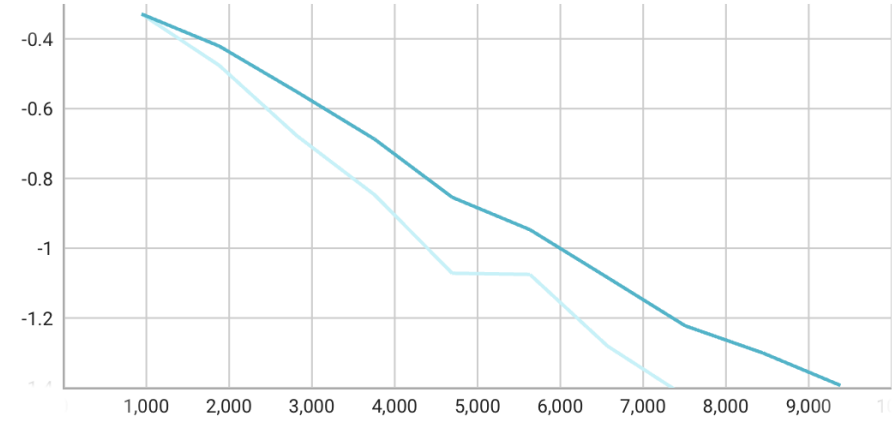
layer_7_biases_max



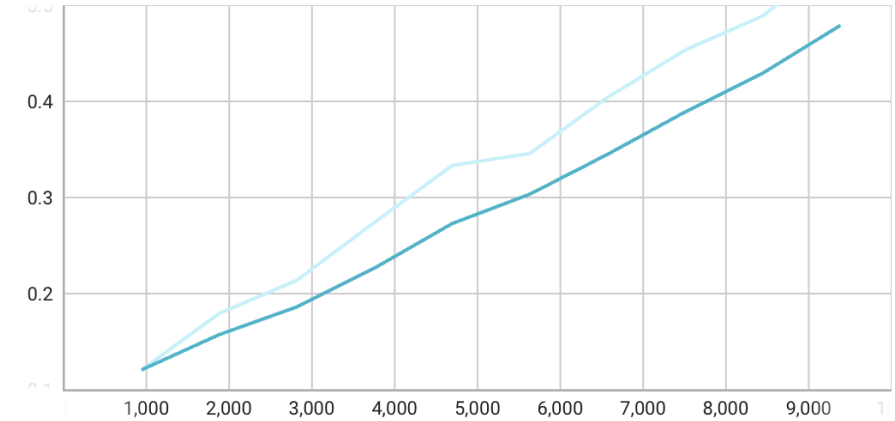
layer_7_biases_mean



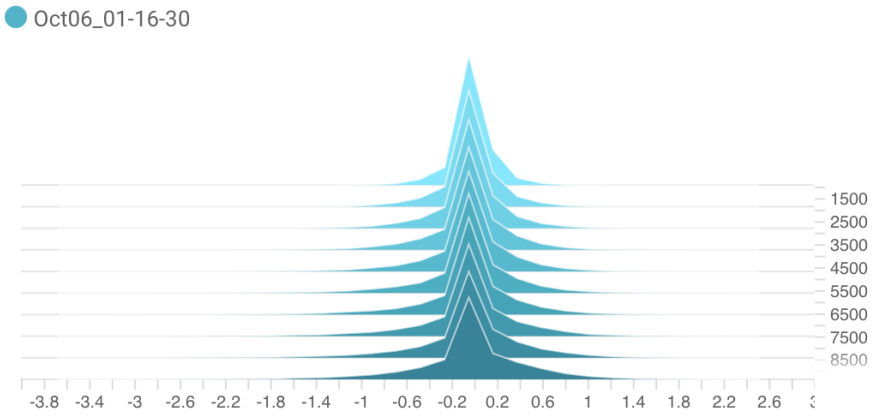
layer_7_biases_min



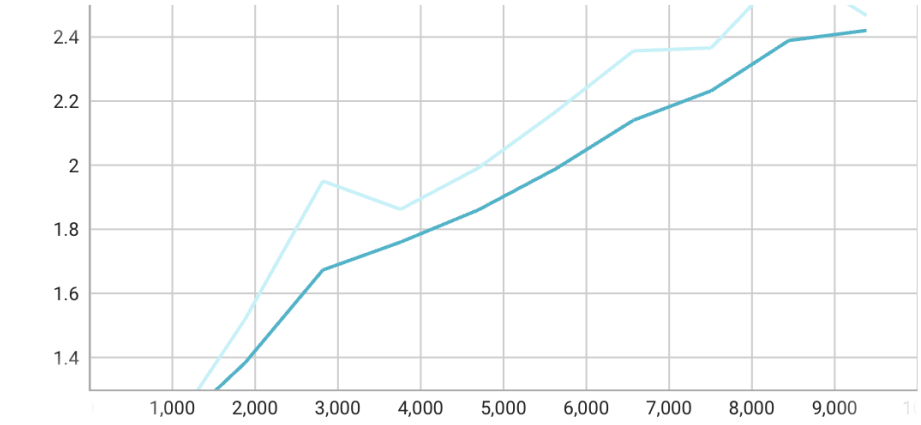
layer_7_biases_stdev



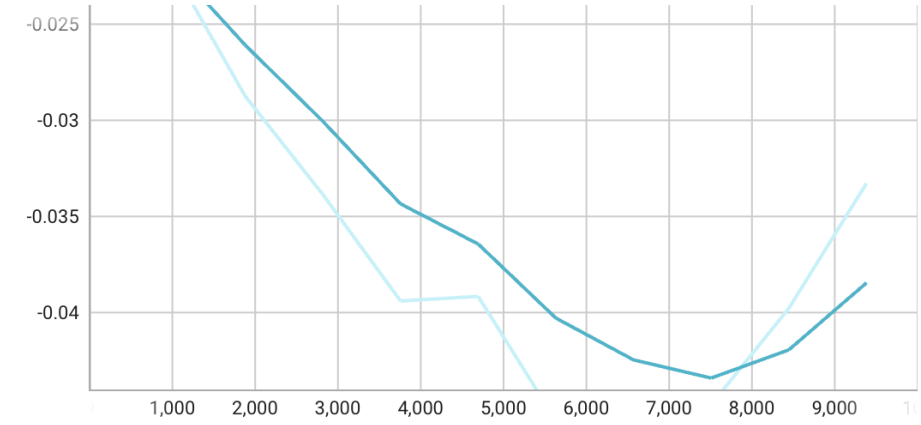
layer_7_weight_histogram



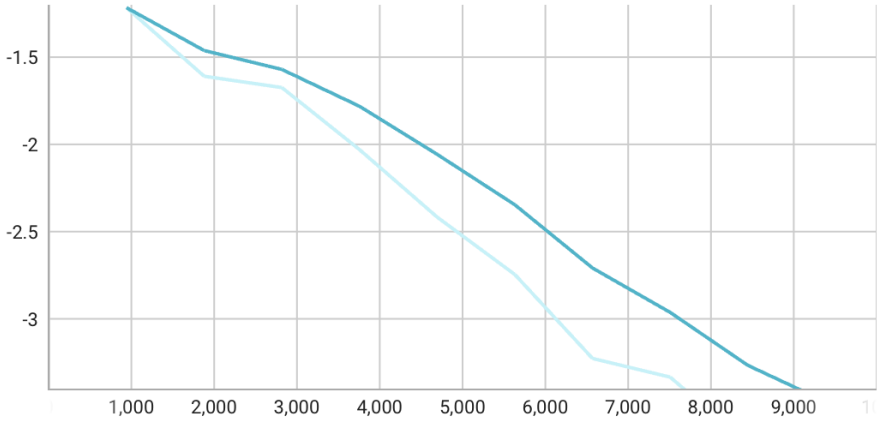
layer_7_weight_max



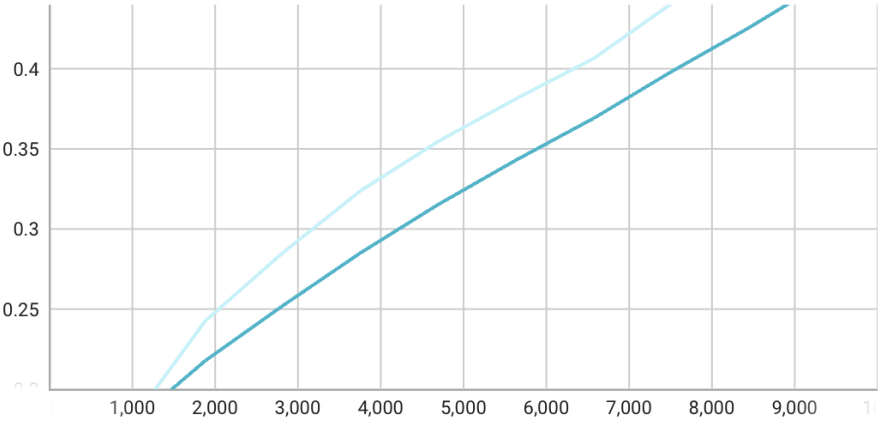
layer_7_weight_mean



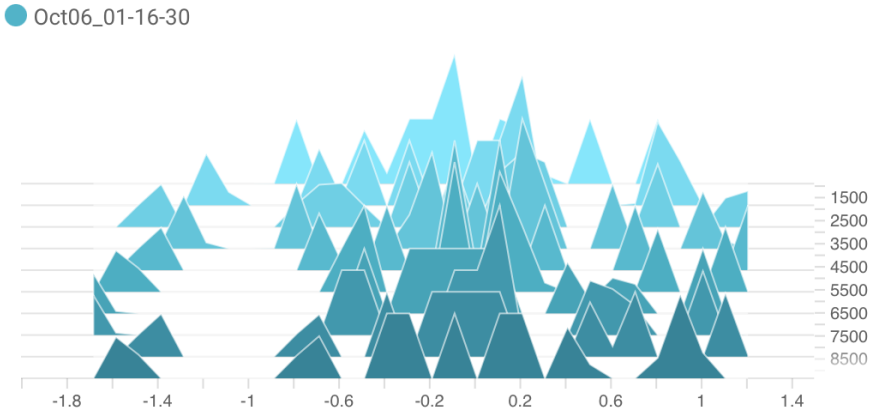
layer_7_weight_min



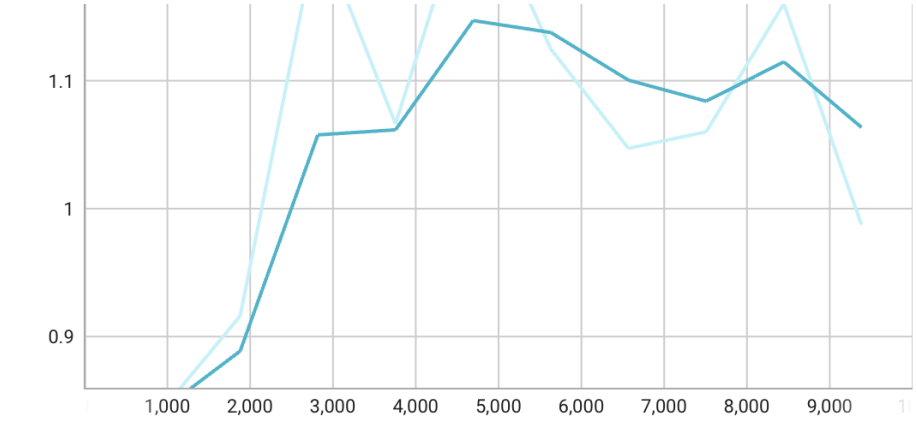
layer_7_weight_stdev



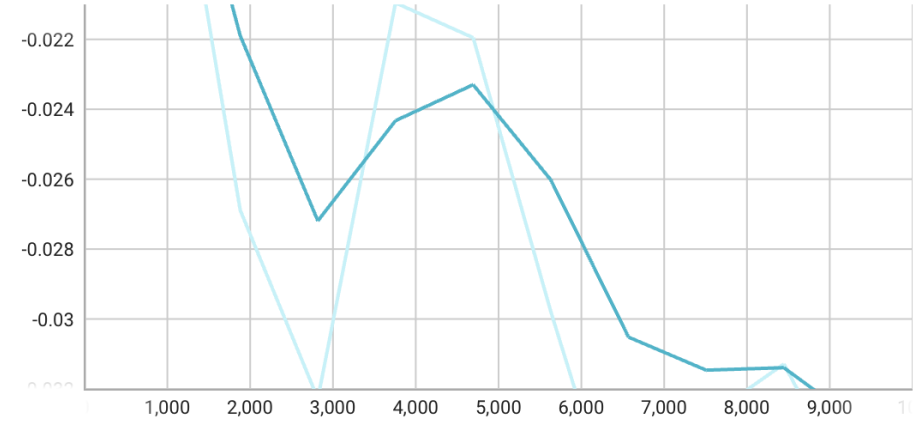
layer_10_biases_histogram



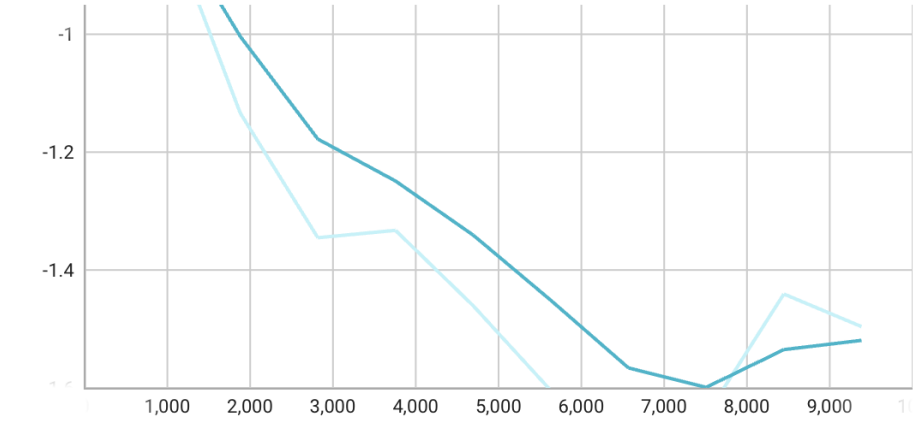
layer_10_biases_max



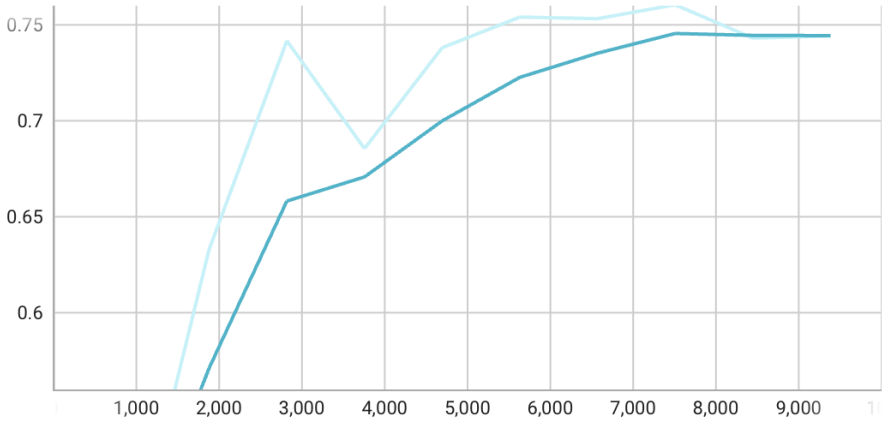
layer_10_biases_mean



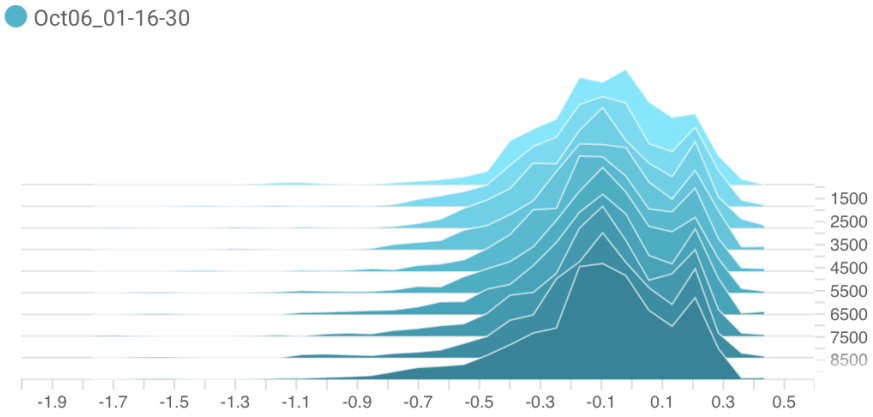
layer_10_biases_min



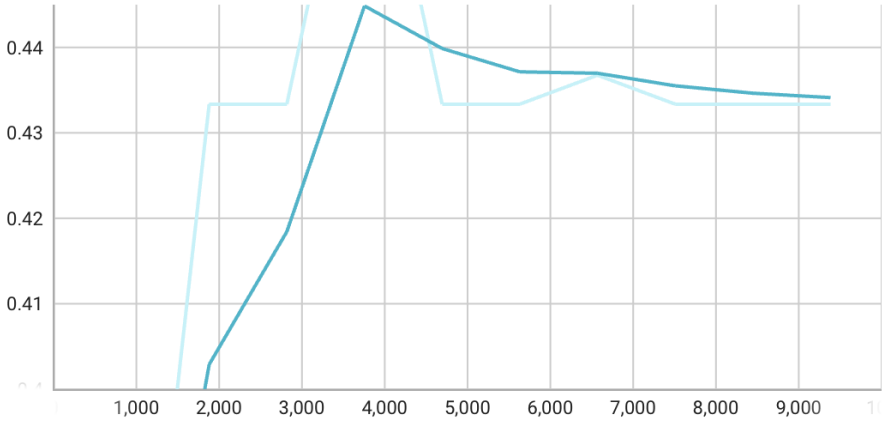
layer_10_biases_stdev



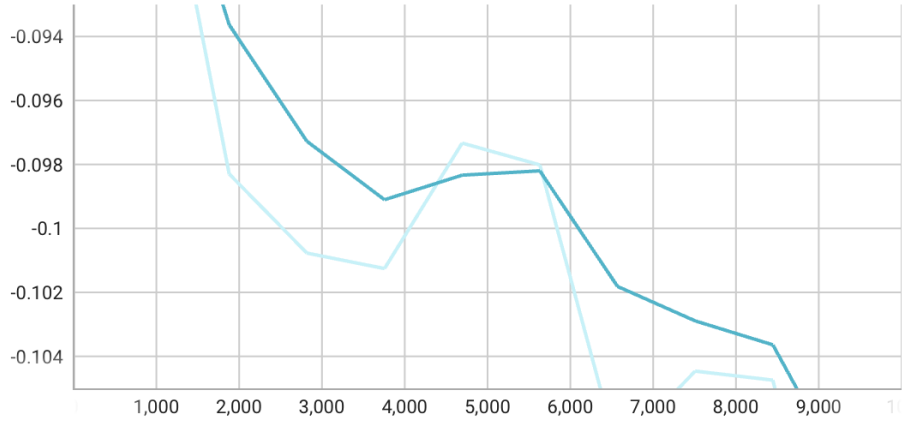
layer_10_weight_histogram



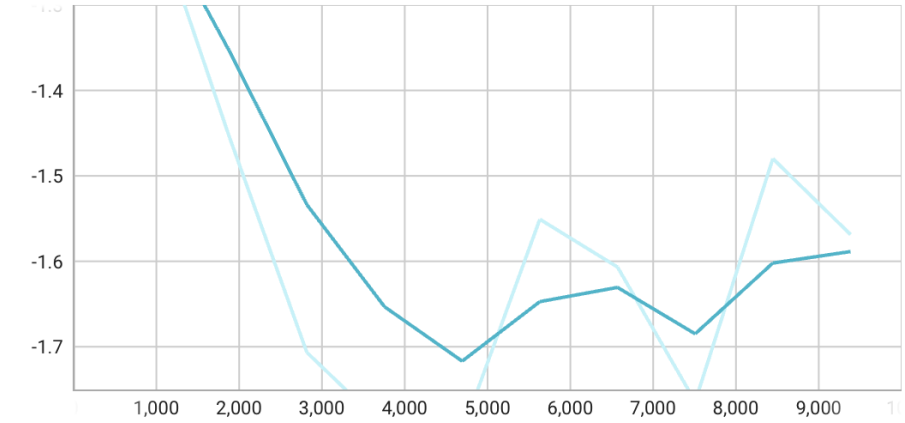
layer_10_weight_max



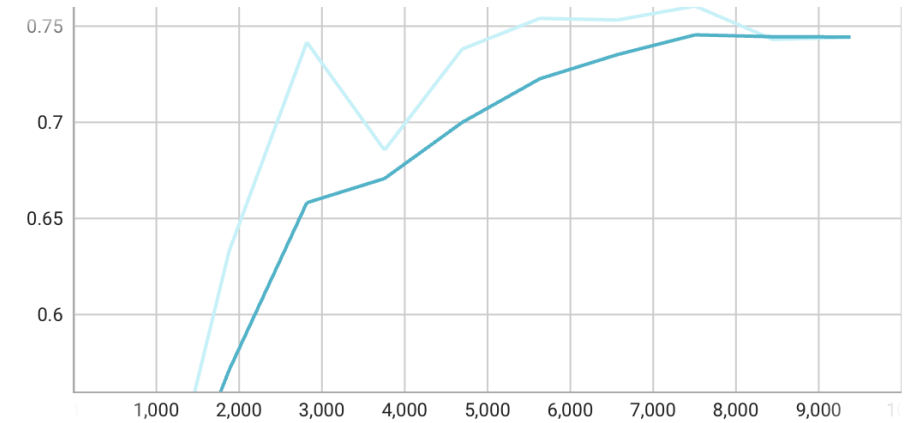
layer_10_weight_mean



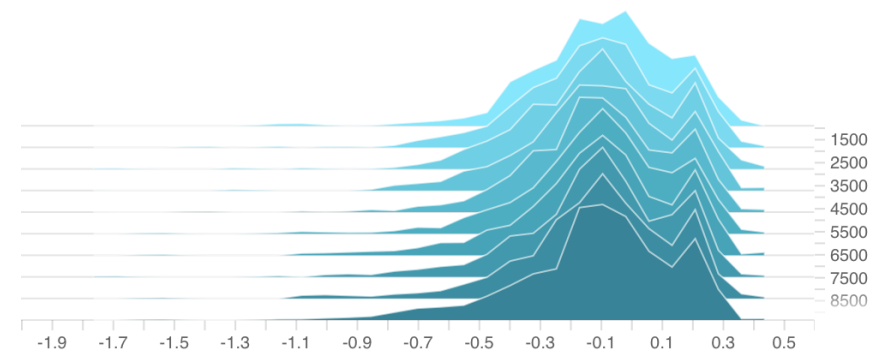
layer_10_weight_min



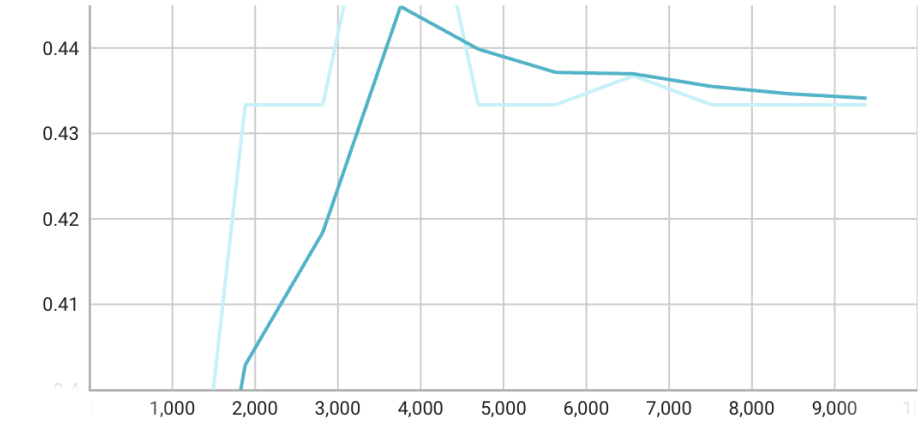
layer_10_biases_stdev



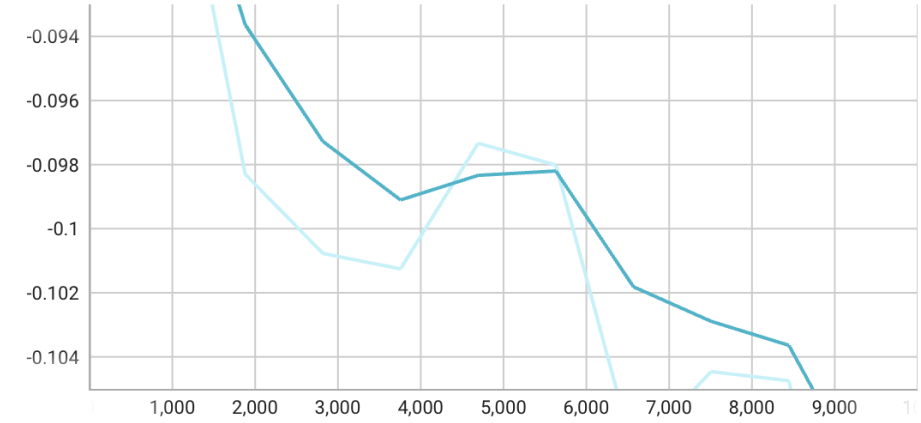
layer_10_weight_histogram



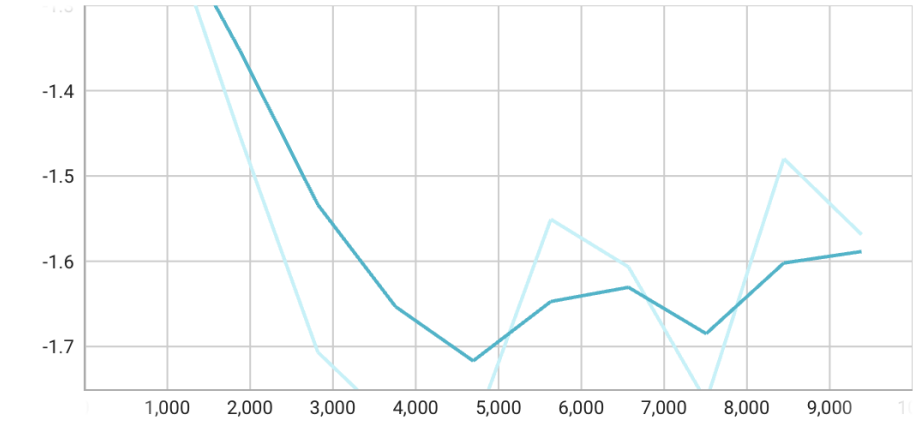
layer_10_weight_max



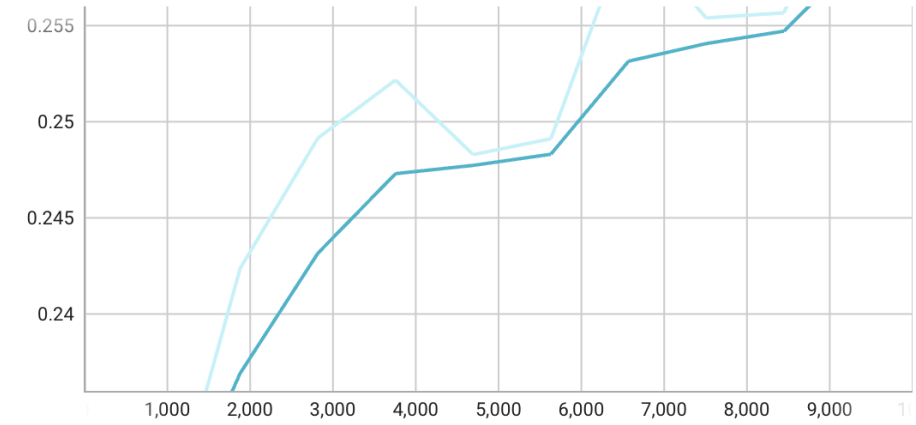
layer_10_weight_mean



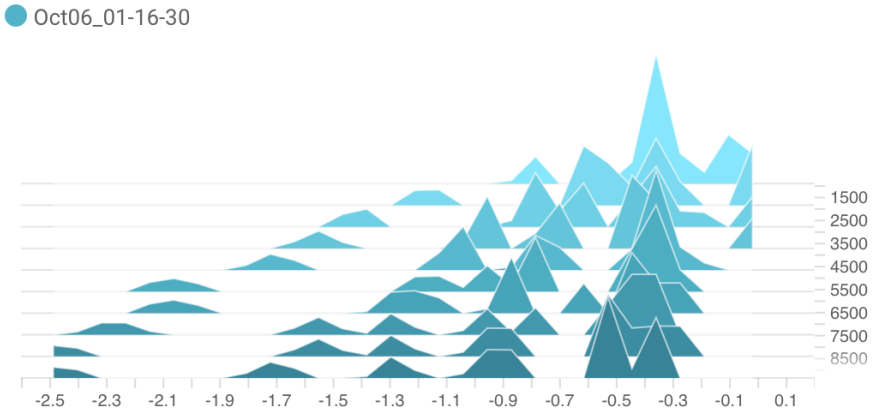
layer_10_weight_min



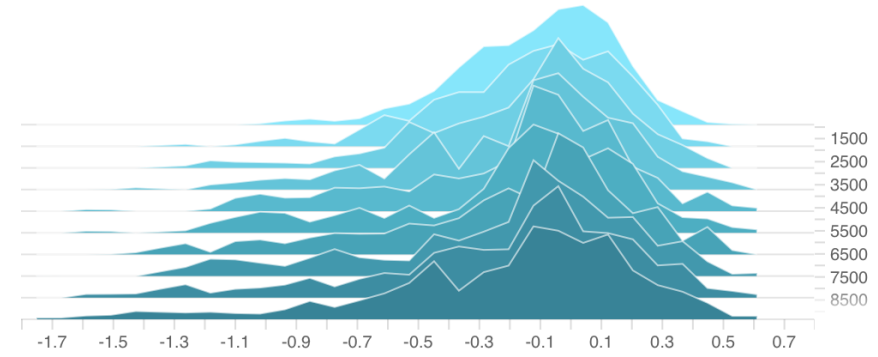
layer_10_weight_stdev



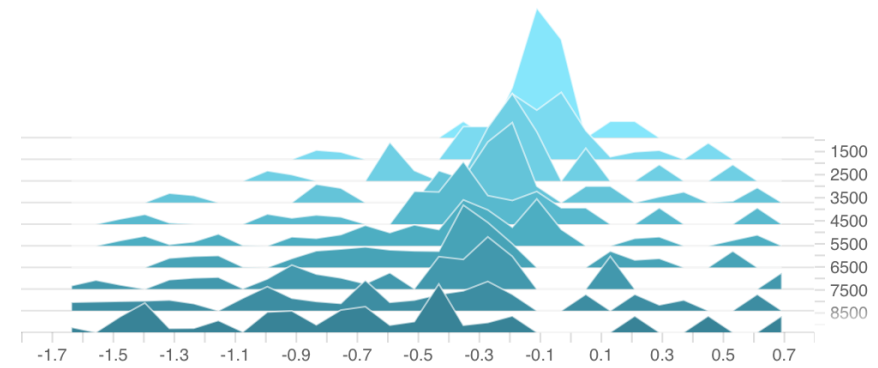
layers.0/bias



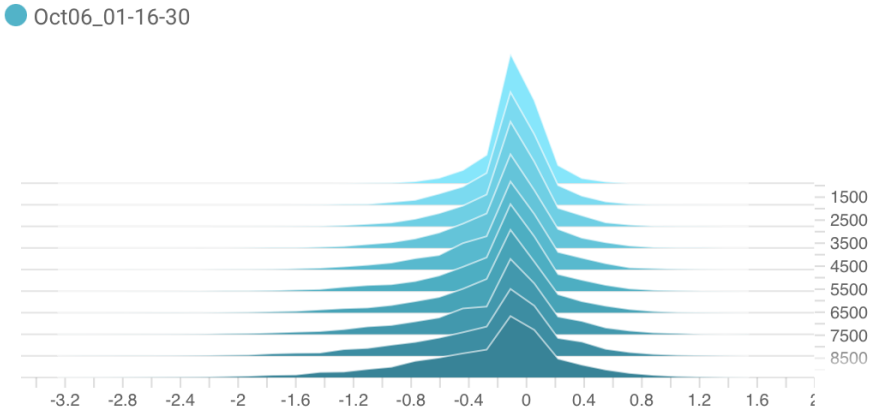
layers.0/weight



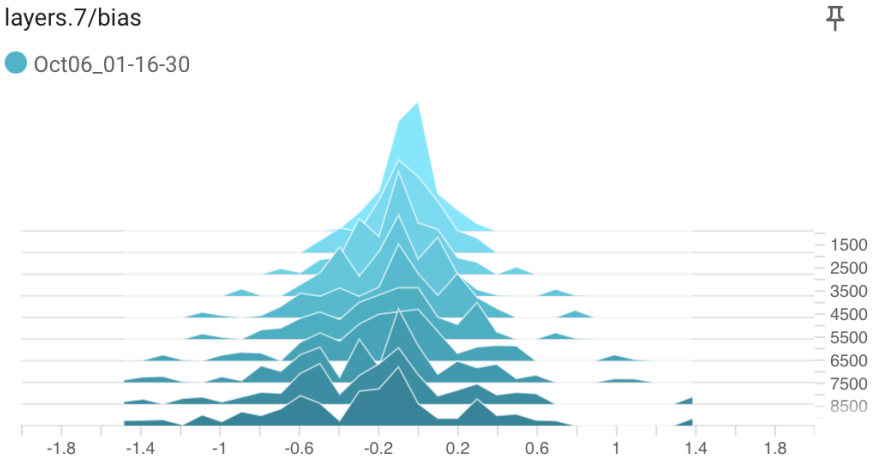
layers.3/bias



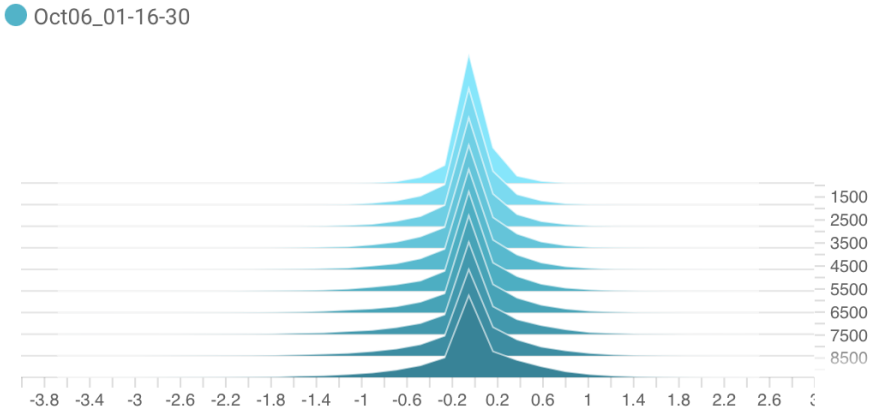
layers.3/weight

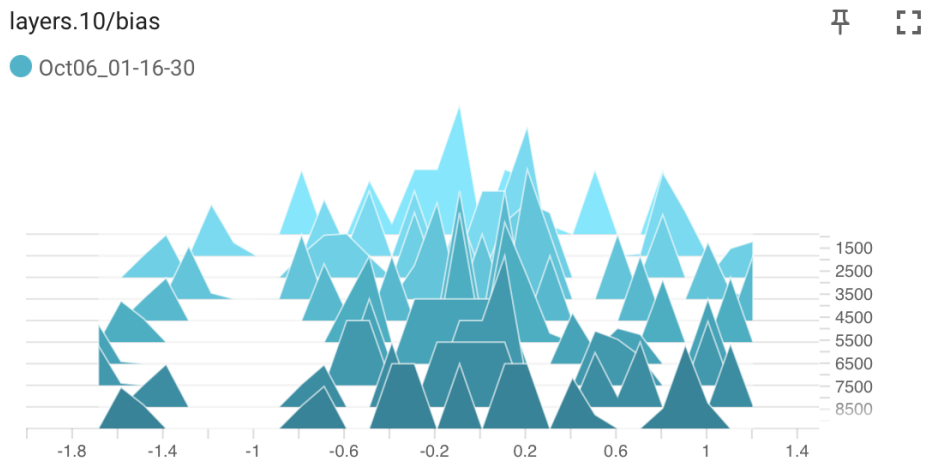


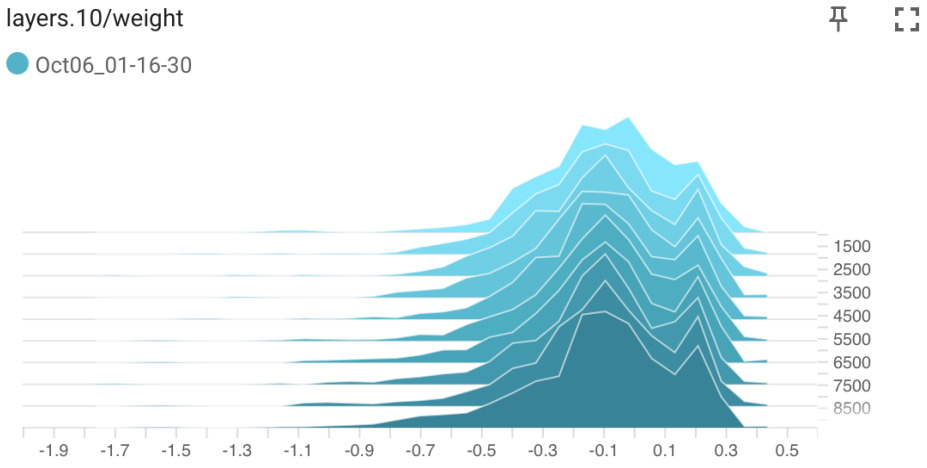
layers.7/bias



layers.7/weight



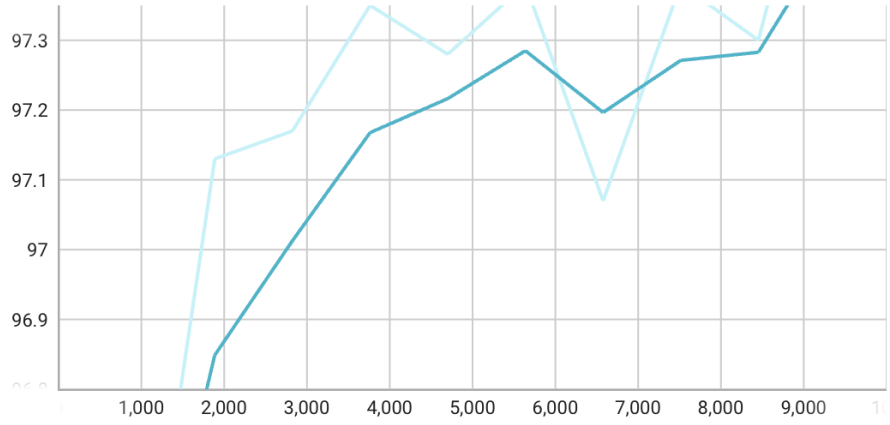




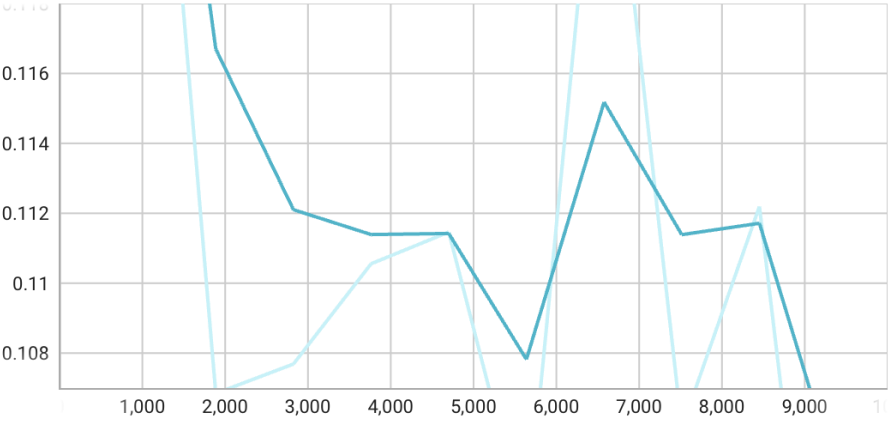
In the various network layers, the different metrics (mean, max, min, standard deviation, and histogram) show how the weights and biases change over the 10 epochs

The test and validation error after every 1100 iterations (each epoch) are shown below).

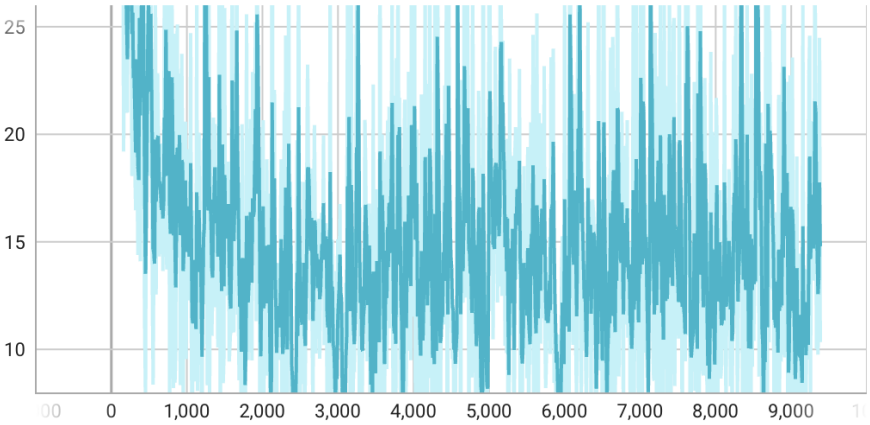
test/accuracy



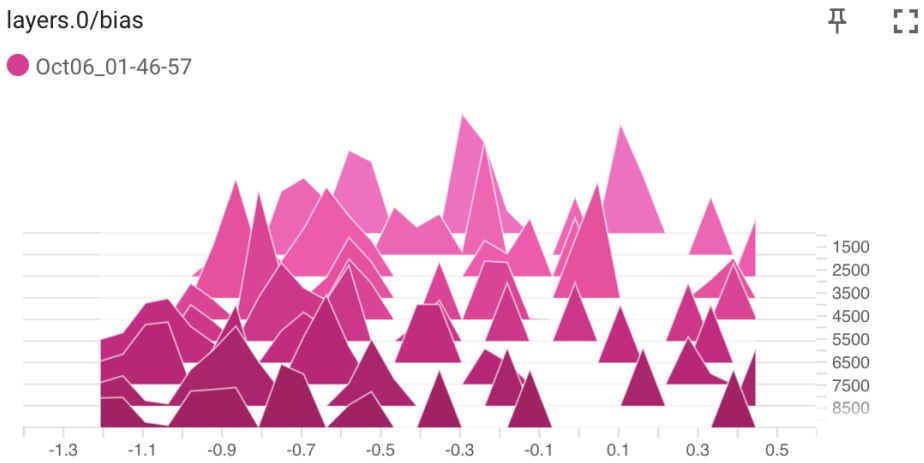
test/loss

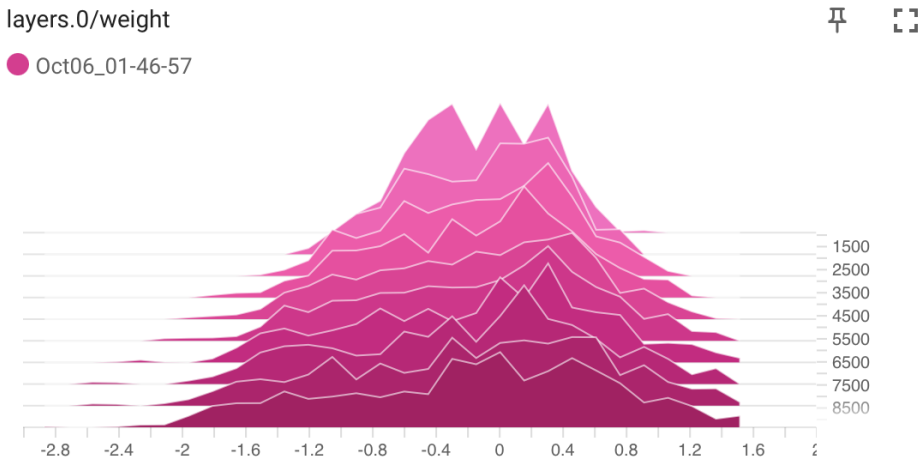


train/loss



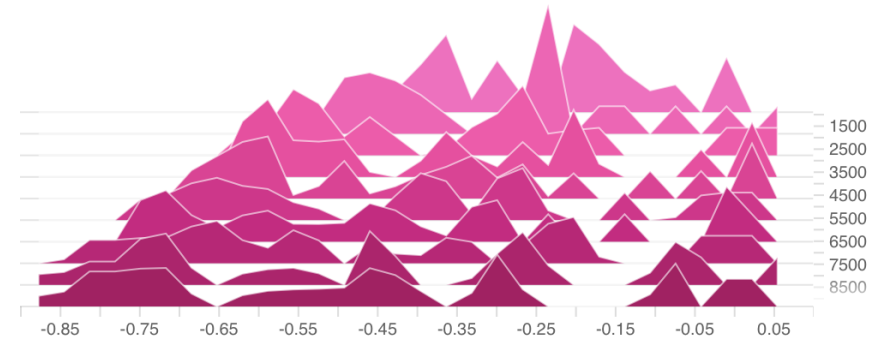
Instead of an Adam training algorithm (optimizer), an SGD was tested as a training algorithm. In addition, a hyperbolic tangent (tanh) non-linearity (activation function) was used instead of a rectified linear unit (ReLU) activation function. Statistics and figures using these new model parameters are shown below.





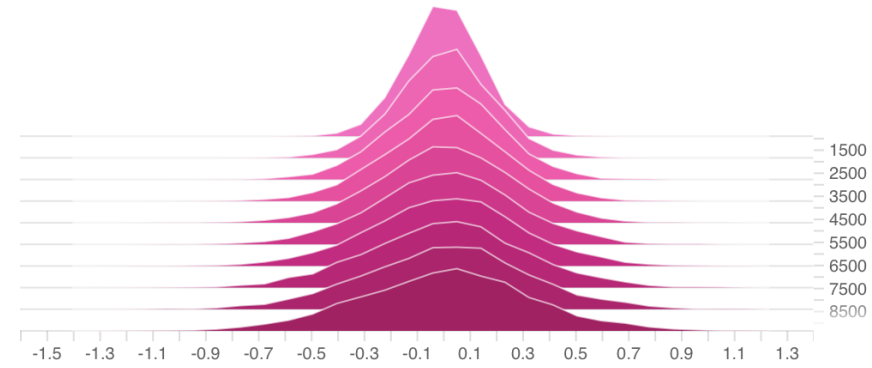
layers.3/bias

Oct06_01-46-57



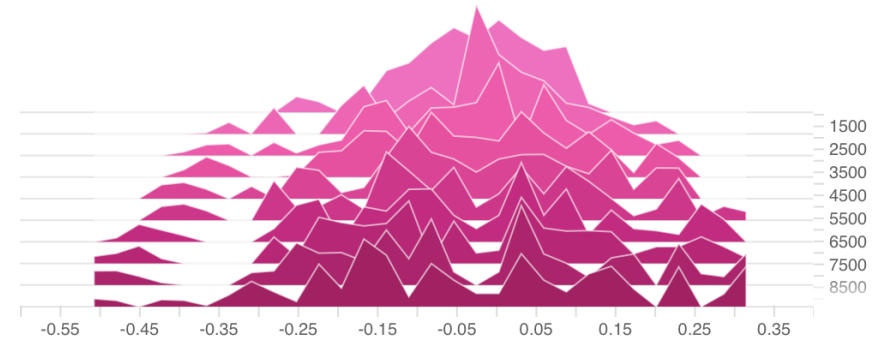
layers.3/weight

Oct06_01-46-57



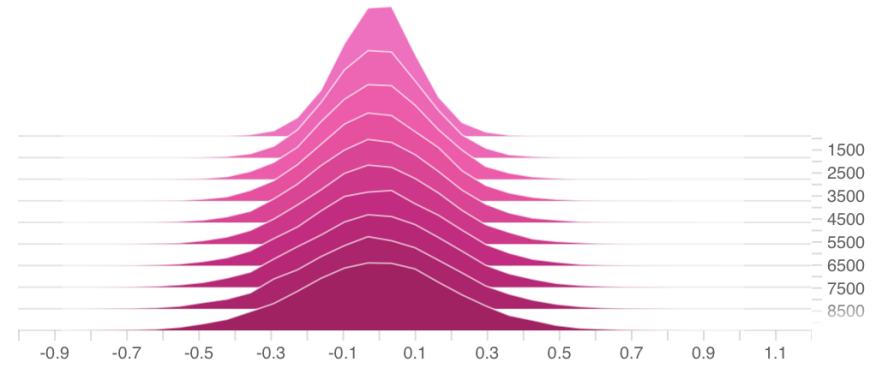
layers.7/bias

Oct06_01-46-57



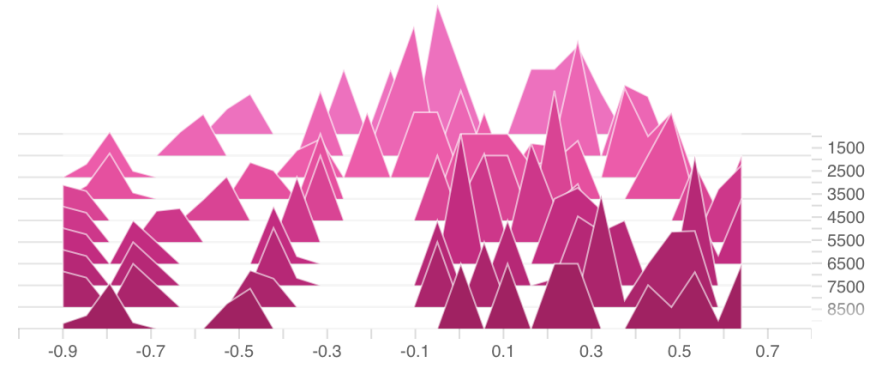
layers.7/weight

Oct06_01-46-57



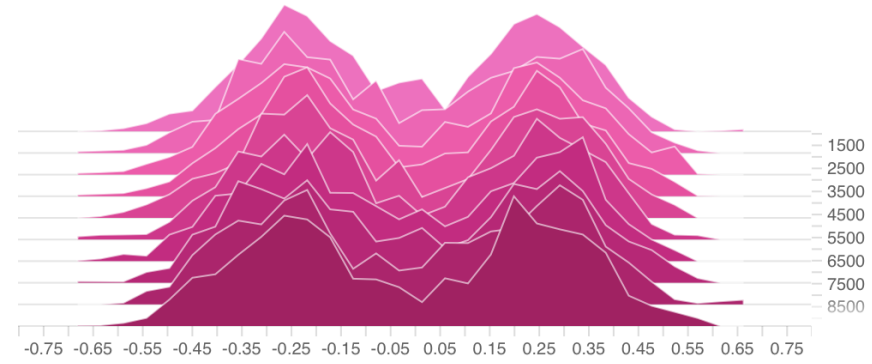
layers.10/bias

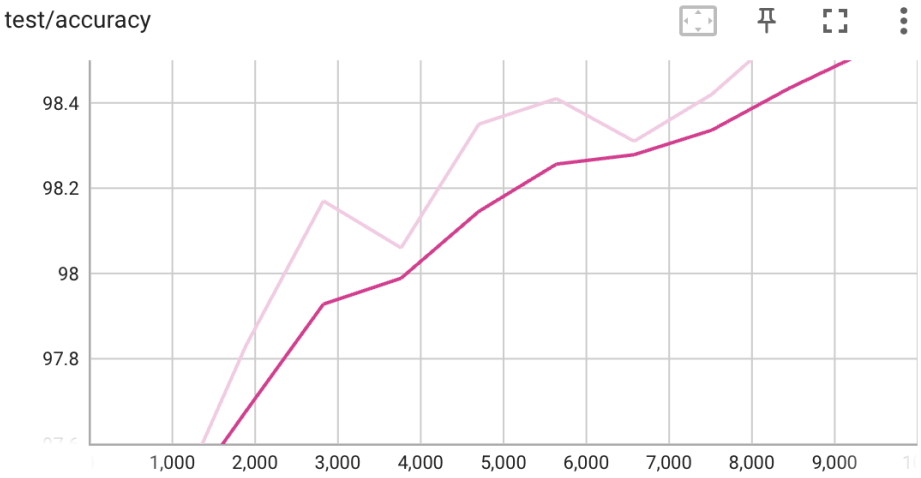
Oct06_01-46-57

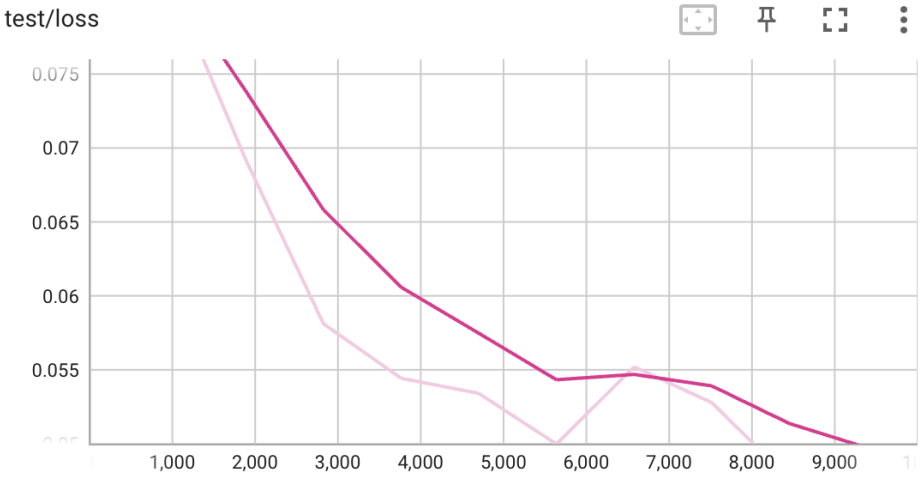


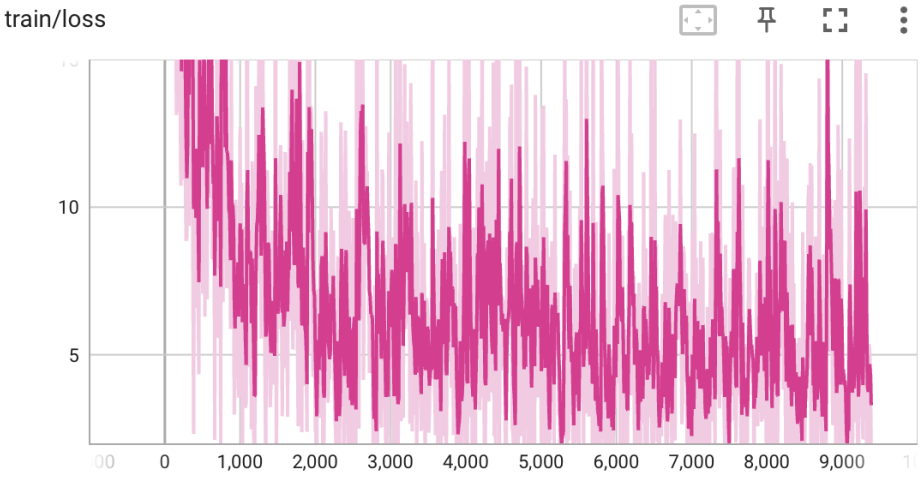
layers.10/weight

Oct06_01-46-57









This new set of parameters produced a higher test accuracy (around 98.5% compared to the 97.4% with the Adam optimizer and ReLU activation function), with different weights and bias histograms during the training process,

particularly in the final linear (fully connected) layer. The test loss decreased more smoothly and the test accuracy increased more smoothly than the previous case as well, perhaps reflective of the smooth shape of the hyperbolic tangent activation function compared to the piecewise rectified linear unit activation function, similar to the decision boundary results in the first task.