Multi-Layer Perceptron

Dr. Ratna Babu Chinnam

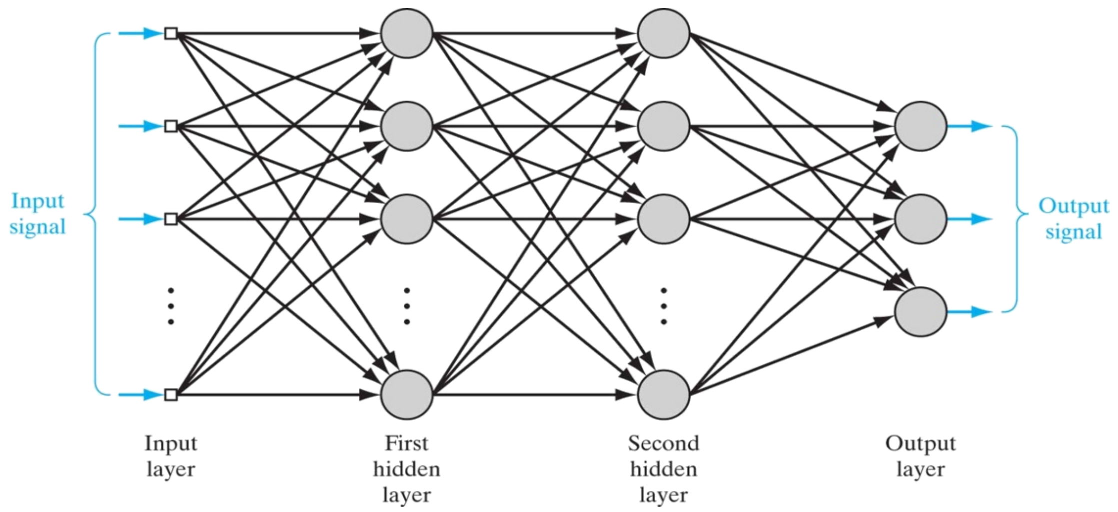
Industrial & Systems Engineering

Wayne State University

Primary Source: Simon Haykin (2009)

Multi-Layer Perceptron

- Consists of ***input layer***, 1 or more ***hidden layers***, and ***output layer***



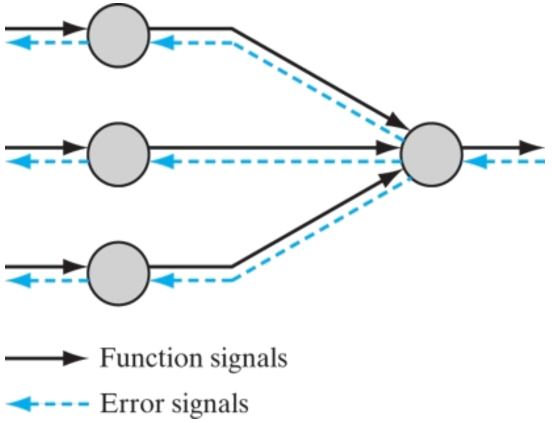
"Fully connected" typically implies that nodes in adjacent layers (not across layers) are fully connected!

Architectural Graph of an MLP Network with Two Hidden Layers

- MLP networks are ***feed-forward networks*** (FFNs)
  - Signal propagates forward, layer-by-layer
- ***Trained in a supervised fashion*** using ***error back-propagation algorithm*** or its variants (based on error-correction rule)

Multi-Layer Perceptron

- ***Back-propagation learning*** consists of ***two passes:***
  - *Forward Pass:* Input signal propagates forward
  - *Backward Pass:* Error signal back-propagates

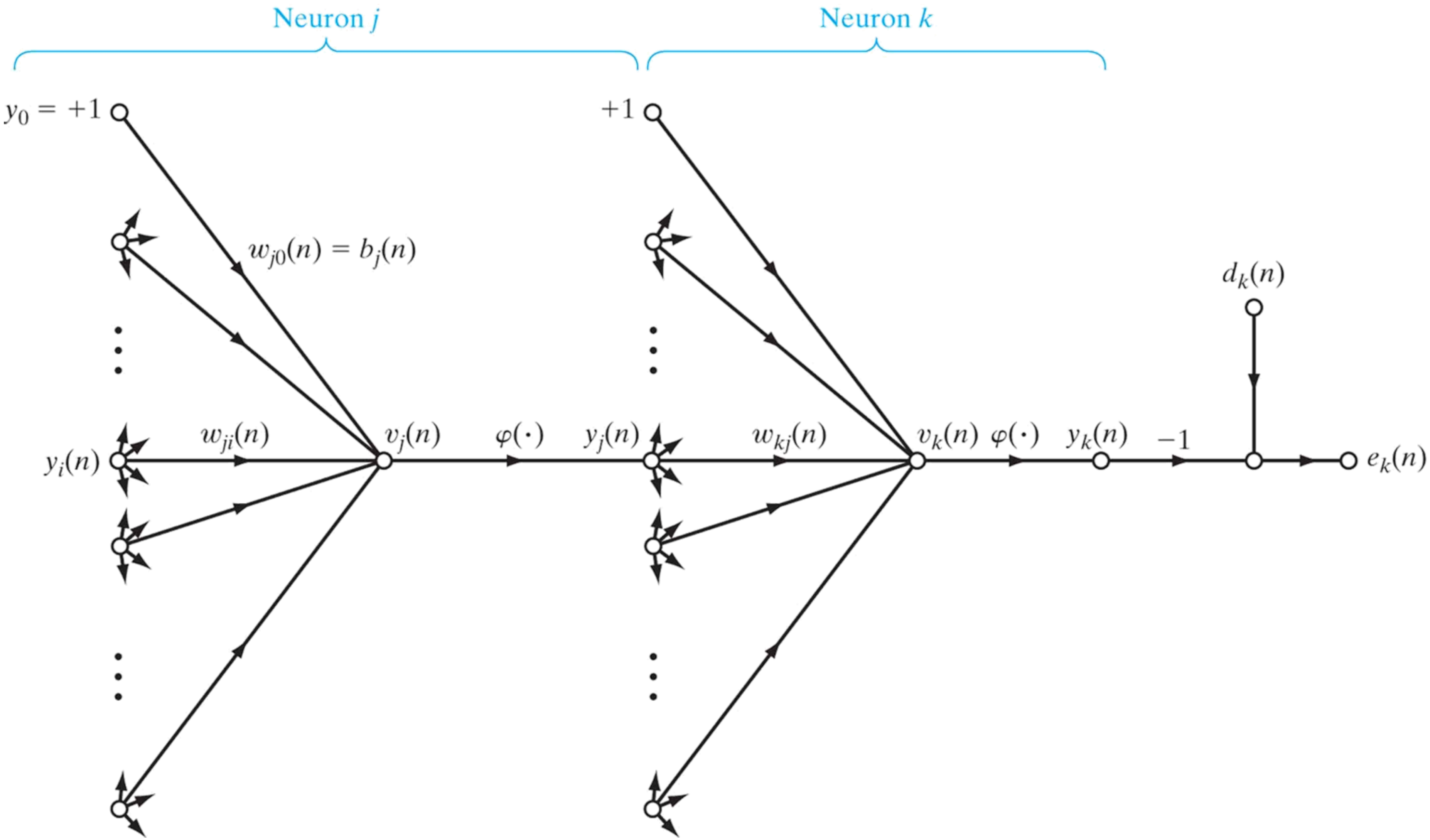


- **Three distinctive characteristics**:
  1. Neuron has a ***differentiable nonlinear activation function***
  2. Network contains 1 or more ***hidden layers***
  3. Network exhibits a high degree of ***connectivity***
- Development of ***back-propagation algorithm*** represents a landmark in that it provides a ***computationally efficient*** method for ***implementing the error-correction rule***

Back-Propagation: Notation

- Indices i, j, and k refer to different neurons in consecutive layers
- $\xi(n)$ denotes "instantaneous" sum of error squares at iteration n
- $y_j(n)$ and $d_j(n)$ denote actual and desired responses, at output neuron j
- $e_j(n)$ denotes neuron j error for iteration n: $e_j(n) = (d_j(n) - y_j(n))$
- $w_{ji}(n)$ denotes weight connecting neuron i to neuron j at iteration n
- $v_j(n)$ denotes activation potential of neuron j at iteration n
- $\varphi_j(\cdot)$ denotes activation function associated with neuron j
- b_j denotes bias for neuron j; represented by $w_{j0} = b_j$ and input ± 1
- η denotes learning rate parameter
- m_l denotes number of nodes in layer l of network; $l = 0,1,\ldots,L$, where L denotes number of hidden and output layers in network
- N denotes total number of patterns in training set

Back-Propagation: Notation



Signal-flow Graph Highlighting the Details of Neuron k Connected to Neuron j

Back-Propagation: Algorithm

- **Errors**: *Instantaneous* and *averaged* sum of squared errors for network

$$\xi(n) = \frac{1}{2}\sum_j e_j(n)^2 \qquad\qquad \xi_{av} = \frac{1}{N}\sum_{n=1}^{N}\xi(n)$$

- **Objective**: Adjust free parameters of network to minimize ξ_{av}
- Let us adjust weights pattern-by-pattern (i.e., "pattern mode" learning)
  - Average of individual weight changes over training set estimate change from directly minimizing ξ_{av} (i.e., "batch mode" learning)
- **Back-Propagation Algorithm:** Employs steepest-descent

$$\Delta w_{ji}(n) \propto \partial\xi(n)/\partial w_{ji}(n)$$

- **Per *delta rule***:

$$\begin{aligned}\Delta w_{ji}(n) &= -\eta\,\frac{\partial\xi(n)}{\partial w_{ji}(n)} \\ &= \eta\left[-\frac{\partial\xi(n)}{\partial v_j(n)}\right]\frac{\partial v_j(n)}{\partial w_{ji}(n)} \qquad \text{(by chain rule)} \\ &= \eta\delta_j(n)y_i(n)\end{aligned}$$

where $\delta_j(n)$ denotes ***local gradient*** of neuron j

Back-Propagation: Algorithm

Case I: Neuron *j* is an Output Node

- According to the chain rule:

$$\frac{\partial \xi(n)}{\partial w_{ji}(n)} = \frac{\partial \xi(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

- From our notation:

$$\frac{\partial \xi(n)}{\partial e_j(n)} = e_j(n) \quad \frac{\partial e_j(n)}{\partial y_j(n)} = -1 \quad \frac{\partial y_j(n)}{\partial v_j(n)} = \varphi'_j(v_j(n)) \quad \frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n)$$

- For current case where neuron *j* is an output node:

$$\delta_j(n) = -\frac{\partial \xi(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)}$$

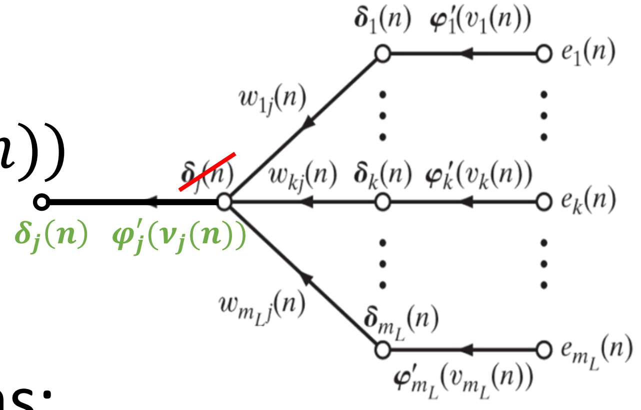
$$= e_j(n) \varphi'_j(v_j(n))$$

Back-Propagation: Algorithm

Case II: Neuron *j* is a Hidden Node

- No specified desired response for neuron (hidden node)
  - Error to be determined in terms of errors of neurons to which it is connected
- Local gradient $\delta_j(n)$ for hidden neuron j is:

$$\delta_j(n) = -\frac{\partial \xi(n)}{\partial y_j(n)}\frac{\partial y_j(n)}{\partial v_j(n)} = -\frac{\partial \xi(n)}{\partial y_j(n)}\varphi'_j(v_j(n))$$



- Instantaneous error: $\xi(n) = \frac{1}{2}\sum_k e_k(n)^2$
- Local gradient $\delta_j(n)$ for hidden neuron j can be rewritten as:

$$\delta_j(n) = -\varphi'_j(v_j(n)) \cdot \sum_k \left[e_k(n)\frac{\partial e_k(n)}{\partial y_j(n)}\right] = -\varphi'_j(v_j(n)) \cdot \sum_k \left[e_k(n)\frac{\partial e_k(n)}{\partial v_k(n)}\frac{\partial v_k(n)}{\partial y_j(n)}\right]$$

$$= -\varphi'_j(v_j(n)) \cdot \sum_k \left[e_k(n)\frac{\partial e_k(n)}{\partial y_k(n)}\frac{\partial y_k(n)}{\partial v_k(n)}\frac{\partial v_k(n)}{\partial y_j(n)}\right] = -\varphi'_j(v_j(n)) \cdot \sum_k \left[e_k(n)(-1)\varphi'_k(v_k(n))\frac{\partial v_k(n)}{\partial y_j(n)}\right]$$

$$= \varphi'_j(v_j(n)) \cdot \sum_k \left[e_k(n)\varphi'_k(v_k(n))\frac{\partial \sum_j w_{kj}(n) y_j(n)}{\partial y_j(n)}\right] = \varphi'_j(v_j(n)) \cdot \sum_k [\delta_k(n) w_{kj}(n)]$$

- $\delta_j(n)$ can be shown to be valid for any neuron in any hidden layer

Activation Function

- Local gradient requires derivative of activation function ($\varphi'(\cdot)$)
- ***Logistic Function:***

$$\varphi_j(v_j(n)) = \frac{a}{1+\exp(-bv_j(n))} \quad (a,b) > 0 \quad -\infty < v_j(n) < +\infty$$

$$\varphi'_j(v_j(n)) = \frac{ab\exp(-bv_j(n))}{\left[1+\exp(-bv_j(n))\right]^2} = \frac{b}{a}\varphi_j(v_j(n))\left[a - \varphi_j(v_j(n))\right]$$

- ***Hyperbolic Tangent Function:***

$$\varphi_j(v_j(n)) = a \cdot \tanh(bv_j(n)) \quad (a,b) > 0 \quad -\infty < v_j(n) < +\infty$$

$$= a \cdot \frac{1-\exp(-bv_j(n))}{1+\exp(-bv_j(n))}$$

$$\varphi'_j(v_j(n)) = ab \cdot \operatorname{sech}^2(bv_j(n))$$

$$= ab(1-\tanh^2(bv_j(n))) = \frac{b}{a}\left[a + \varphi_j(v_j(n))\right]\left[a - \varphi_j(v_j(n))\right]$$

Activation Function

- ***Softmax Function:***

$$p_j = \text{softmax}(\boldsymbol{y} = [y_1, \ldots, y_K]) = \frac{e^{y_j}}{\sum_{k=1}^{K} e^{y_k}}$$

$$\frac{\partial p_j(n)}{\partial y_i(n)} = \begin{cases} p_j(1 - p_i) & if\ j = i \\ -p_j \cdot p_i & if\ j \neq i \end{cases}$$

See Full Derivation Here: [Link]()

- ***ReLu Functions:***

Rectifer:

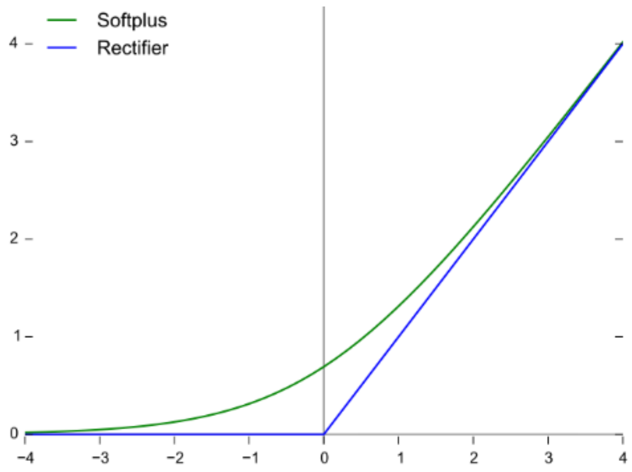
$$\varphi_j(v_j(n)) = \max(0, v_j(n))$$

$$\varphi'_j(v_j(n)) = \begin{cases} 1 & if\ v_j(n) > 0 \\ 0 & if\ v_j(n) \leq 0 \end{cases}$$

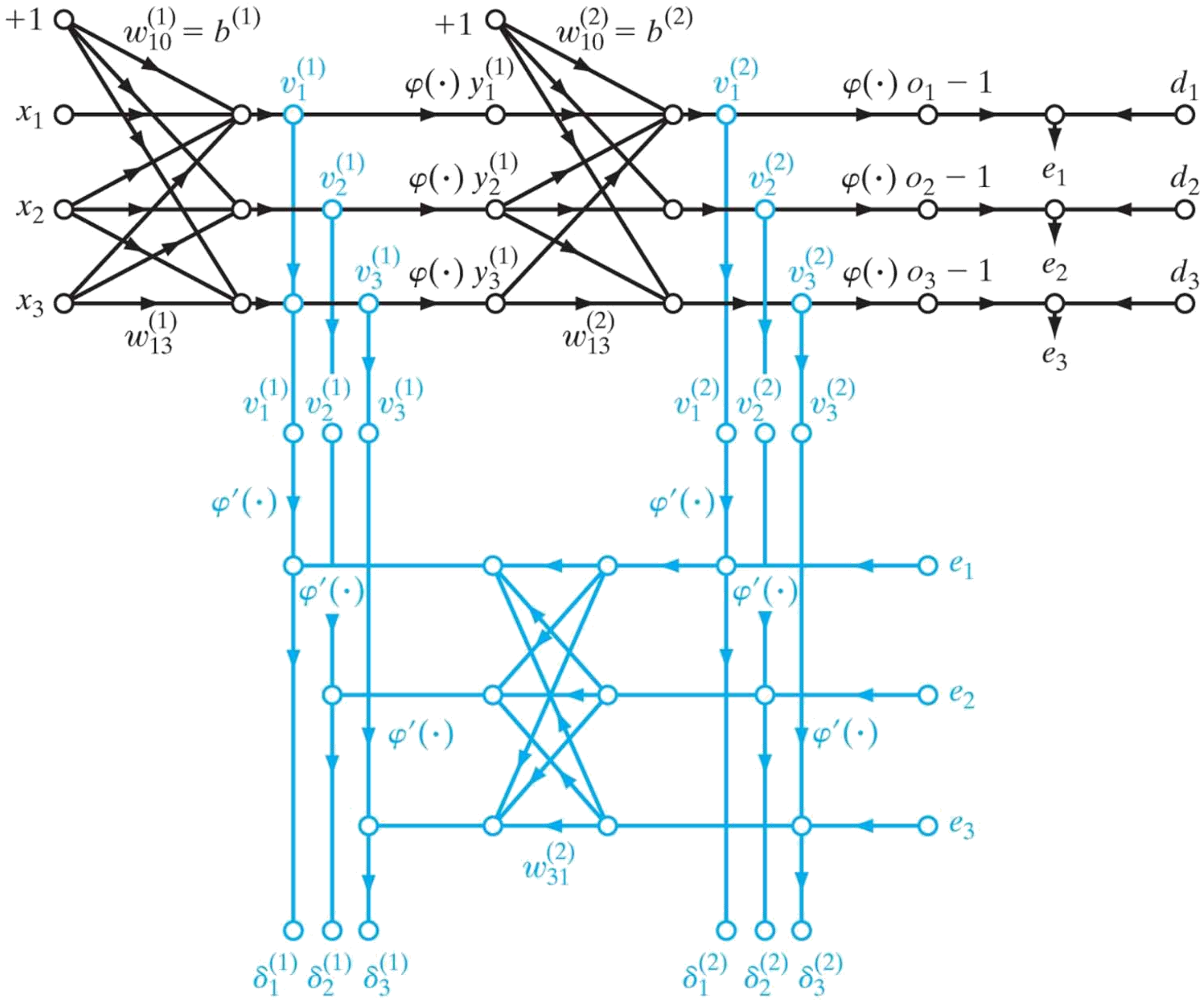
Softplus:

$$\varphi_j(v_j(n)) = \log(1 + e^{v_j(n)})$$

$$\varphi'_j(v_j(n)) = \frac{e^{v_j(n)}}{1+e^{v_j(n)}}$$



Signal-flow Summary of Back-propagation Algorithm



Representation: Top Part→Forward Pass. Bottom Part→Backward Pass.

Rate of Learning: “Momentum” and $\eta(n)$

- Back-propagation algorithm provides an “approximation” to steepest descent during pattern-mode learning: **Stochastic Gradient Descent**
- **Rate of Learning**:
  - Small η: Smooth trajectory attained at the cost of a slower rate of learning
  - Large η: Faster learning but there is a danger network may become unstable
- **Momentum**: One can increase rate of learning while avoiding instability:

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n) \quad 0 \leq |\alpha| < 1$$

$$= \eta \sum_{t=0}^{n} \alpha^{n-t} \delta_j(t) y_i(t)$$

  - Current adjustment represents sum of an exponentially weighted time series
- Tends to *accelerate descent* in steady downhill directions and has a *stabilizing effect* in directions that oscillate in sign
- **Learning Rate**: Can be adapted as well: $\eta(n)$ Ideas?

Sequential (Pattern) & Batch Modes of Training

- **Sequential Mode**: Adjusts network parameters for each example

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n)$$

  - Search in the weight space tends to be *stochastic* avoiding limit cycles
  - Order of presentation of training examples has to be randomized
  - Reduced memory storage
  - Increased potential for avoiding local minima
- **Batch Mode**: Adjusts network parameters after presenting all examples
  - Local gradients have to be calculated after presentation of each example

$$\Delta w_{ji} = \frac{\eta}{N} \sum_{n=1}^{N} \delta_j(n) y_i(n)$$

  - Easier to establish theoretical conditions for algorithm convergence
- **Note**: Computed $\delta_j(n)$ and $y_i(n)$ tend to differ for the two modes of training since the weights are updated at different frequencies

Useful Criteria for "Stopping" Network Training

- **Epochs**: Predetermined number of "epochs"
  - Epoch: A single presentation of ALL training patterns to network
- **Computation Time**: Predetermined processor computation time
- **Learning Error**: Acceptable minimum learning error
- **Learning Rate**: Acceptable minimum learning rate
- **Generalization**:
  - Acceptable generalization performance
  - Apparent peak (actually, a "valley" in error plot) in generalization performance

"Heuristics" for Improving Back-propagation Algorithm

- ***Sequential versus batch update***. Sequential update leads to stochastic gradient search and can be more effective in practice
- ***Mini-batches provide a better balance*** between accuracy and computational efficiency
- ***Nature of activation function***. Antisymmetric functions generally help (i.e., $\varphi(-v) = -\varphi(v)$).
- ***Target values***. Linear output neurons might be safer for regression.
- ***Normalization of inputs and outputs***. Is crucial.
- ***Weight initialization***. Synaptic weights should be initialized uniformly with mean zero and variance equal to the reciprocal of the number of synaptic connections of a neuron.
- ***Learning rate parameters***. All neurons should learn at the same rate. Since the last layers have usually larger local gradients, their learning rate parameters should be smaller.
- ***A priori information***. Utilize this knowledge to achieve invariance properties, symmetries, etc.

"Heuristics" for Improving Back-propagation Algorithm

Batch Normalization for Hidden Layers

- Stochastic/mini-batch gradient is effective, but requires tuning of hyper-parameters
  - Learning rate, network initialization
- Training is complicated for each layer is affected by parameters of preceding layers
  - Small changes to network parameters amplify as network becomes deeper
- **Covariate shift**: Change in distributions of layers' inputs presents a problem
  - Layers need to continuously adapt to new distributions over the course of training
    - Batch to batch variation
    - Learnt "black" cats and now learning "white" cats
  - **Batch normalization helps overcome covariate shift** for hidden layers
- If γ and β parameters are calculated from the mini-batch, the noise introduced also has a **slight regularization effect**

For Any Given Hidden Layer

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

Gradients for Back-Propagation

$$\frac{\partial \ell}{\partial \widehat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$

$$\frac{\partial \ell}{\partial \sigma_\mathcal{B}^2} = \sum_{i=1}^m \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_\mathcal{B}) \cdot \frac{-1}{2}(\sigma_\mathcal{B}^2 + \epsilon)^{-3/2}$$

$$\frac{\partial \ell}{\partial \mu_\mathcal{B}} = \left(\sum_{i=1}^m \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}\right) + \frac{\partial \ell}{\partial \sigma_\mathcal{B}^2} \cdot \frac{\sum_{i=1}^m -2(x_i - \mu_\mathcal{B})}{m}$$

$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_\mathcal{B}^2} \cdot \frac{2(x_i - \mu_\mathcal{B})}{m} + \frac{\partial \ell}{\partial \mu_\mathcal{B}} \cdot \frac{1}{m}$$

$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^m \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i$$

$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^m \frac{\partial \ell}{\partial y_i}$$

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Source: Link

Sergey Ioffe
Google Inc., *sioffe@google.com*

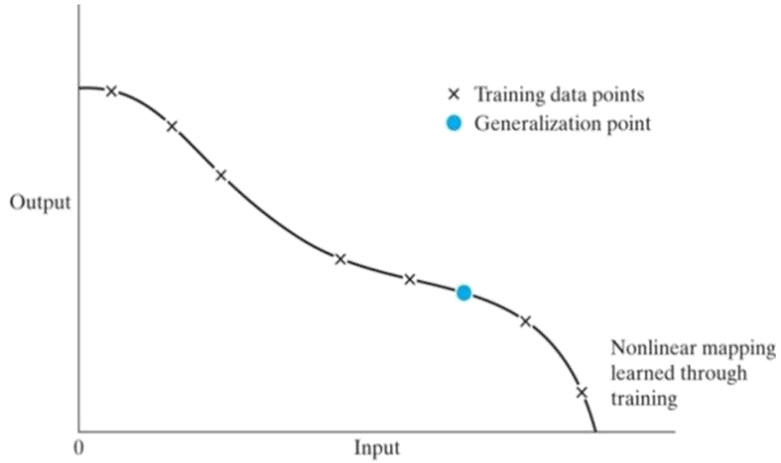
Christian Szegedy
Google Inc., *szegedy@google.com*

Batch Normalization in TensorFlow: Link

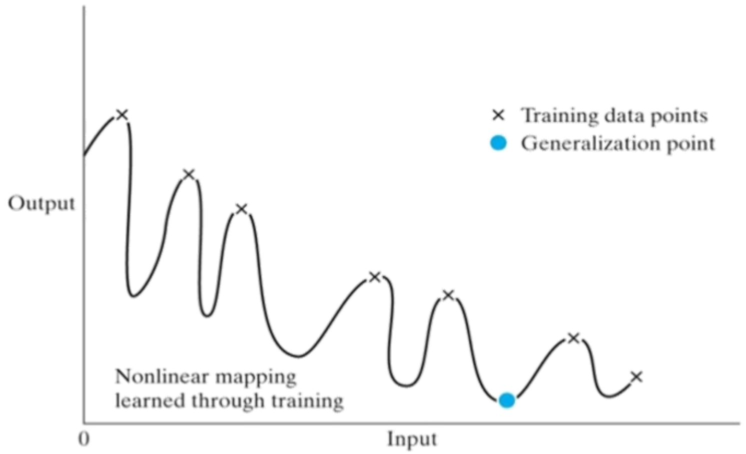
Why Batch Norm Works: Link

Achieving “Generalization” with MLP

- Network is said to *generalize* well when the mapping is correct for test data
  - Assumption: Test data are drawn from the same process/population
- Curve-fitting Example: Equivalent to good interpolation
- *Overtraining* or *overfitting* can result if network starts “memorizing” training data
- Can be viewed from different perspectives:
  - Architecture is fixed, what should be the size of the training data?
    - VC dimension provides the theoretical basis
  - Training data set is fixed, what is the best architecture for the network?
    - More common approach



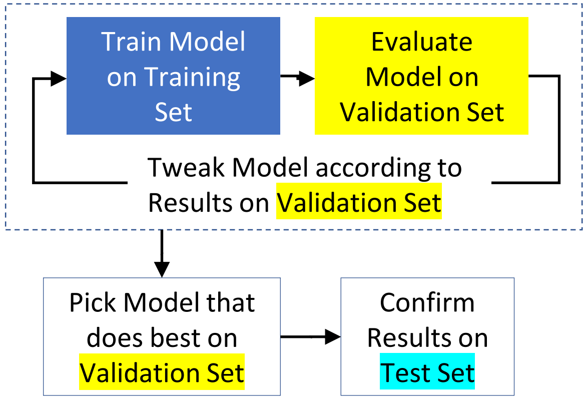
Mapping with good generalization



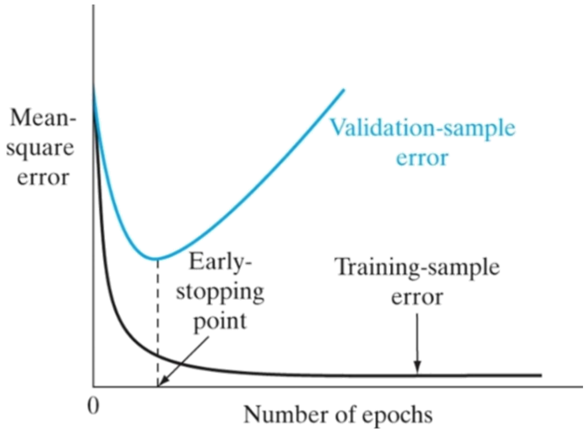
Mapping with poor generalization

Achieving Generalization through “Cross-Validation”

- “Training” can be seen as making the network “learn enough” to generalize
- **Cross-Validation**: A standard tool from statistics is valuable
  - Dataset is partitioned into ***training*** (N patterns) and ***test*** sets (M patterns)
  - Training set is further partitioned into two disjoint subsets:
    - ***Estimation*** subset to "learn" the model (with $(1-r)N$ patterns, where $0 < r < 1$)
    - ***Validation*** subset to "validate" the model (with rN patterns)
  - Possible for model to overfit validation subset. Measure performance on test set.
  - As target function becomes complex relative to N, choice of r^* becomes important and should decrease in value (Kearns, 1996)



- **Early Stopping:** A method of training using cross-validation
  - Periodically stop network training process and test network
  - Permanently stop training if an increase is noted in validation error over successive evaluations.
  - Effectiveness decreases if $N \gg W$



Variants of Cross-Validation

- For relatively small training data sets, two variants of cross-validation that offer more promise include:
- ***Multifold Method***
  - Involves dividing training set into K subsets, $K > 1$
  - Model is trained on all subsets except for one, and validation error is measured by testing it on the subset left out
  - Procedure is repeated for a total of K trials, each time using a different subset for validation
  - Model performance is assessed by averaging squared error under validation over all trials of the experiment
  - Increases computational burden
- ***Leave-one-out Method***
  - When data set is severely small, one can use extreme form of multifold cross-validation where K equals N

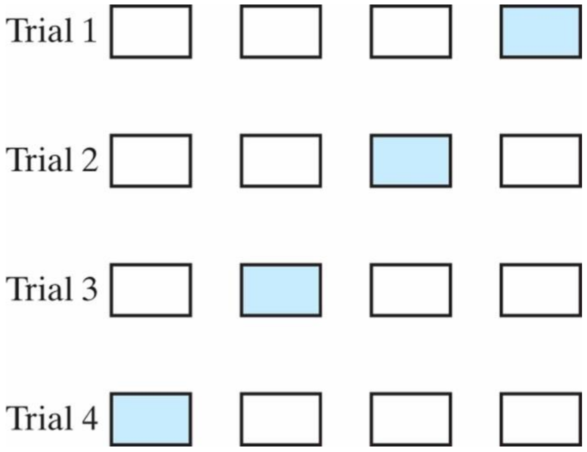


Illustration of the Hold-Out Method of Cross-Validation

Implementing Cross-Validation: Heuristic Guidelines

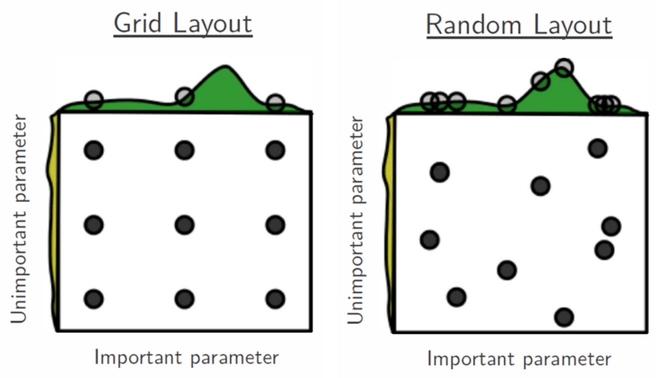
- **Large Sample Size (data points >>>> model parameters): No need for cross-validation**
  - Optimize network configuration and model parameters using full dataset
  - Occam's Razor Principle is still relevant (seek smaller models over complex models)
- **Decent Sample Size (data points >> model parameters): Simple Cross-Validation** (partition dataset into ***training, validation,*** *and* ***test*** sets)
  - Optimize network configuration using best validation error as criteria
  - Evaluate errors on test set to ensure that they are comparable to errors seen under cross-validation
- **Small Sample Size (data points > model parameters): Implement Multifold Method**
  - Optimize network configuration using best pooled validation error as criteria
    - Aggregate errors over "all" validation trials of the experiment
  - Retrain network using **entire training and validation sets** and rely on early-stopping to stop training once training error reaches the training error corresponding to best validation error
- **Very Small Sample Size (data points < model parameters): Leave-one-out Method**
  - Optimize network configuration using best pooled validation error as criteria
  - Retrain network using **entire dataset** and rely on early-stopping to stop the training once the error reaches the training error corresponding to best validation error
- **Sampling for Data Partition**: Seek stratified (classification) sets with full coverage
  - To the extent possible, all datasets should have cases from all classes (classification) and/or cover the full range of the input space (regression)

Complexity-Regularization: Network Pruning/Growing

- **Idea:** A network with minimum/optimal size is less likely to learn the idiosyncrasies or noise in the training data (offering good generalization)
- **Two Approaches**: *Network Growing & Network Pruning*
  - Aim to tackle the bias-variance dilemma
- **Complexity-Regularization:** Minimize risk $R(\mathbf{w}) = \xi_S(\mathbf{w}) + \lambda\xi_C(\mathbf{w})$
  - $\xi_C(\mathbf{w})$ denotes complexity penalty and depends on the network (model) alone
  - $\xi_S(\mathbf{w})$ denotes the standard performance measure (such as MSE) and depends on network (model) and input data
  - λ denotes complexity regularization parameter ($0 \leq \lambda \leq \infty$)
- ***K*th-Order Smoothing Requirement**: $\xi_C(\mathbf{w}, k) = \frac{1}{2}\int \left\| \frac{\partial^k}{\partial \mathbf{x}^k} F(\mathbf{x}, \mathbf{w}) \right\|^2 \mu(\mathbf{x}) d\mathbf{x}$
  - $F(\mathbf{x}, \mathbf{w})$ is input-output mapping performed by the model
  - $\mu(\mathbf{x})$ is some weighting function determining region of input space over which function $F(\mathbf{x}, \mathbf{w})$ is required to be smooth
  - Larger the k, smoother (i.e., less complex) the function $F(\mathbf{x}, \mathbf{w})$
- **Weight Decay Method**: $\xi_C(\mathbf{w}) = \|\mathbf{w}\|^2$
- **Weight Elimination Method**: $\xi_C(\mathbf{w}) = \sum_{\forall i} \frac{(w_i/w_0)^2}{1+(w_i/w_0)^2}$ (w_0 is a parameter)

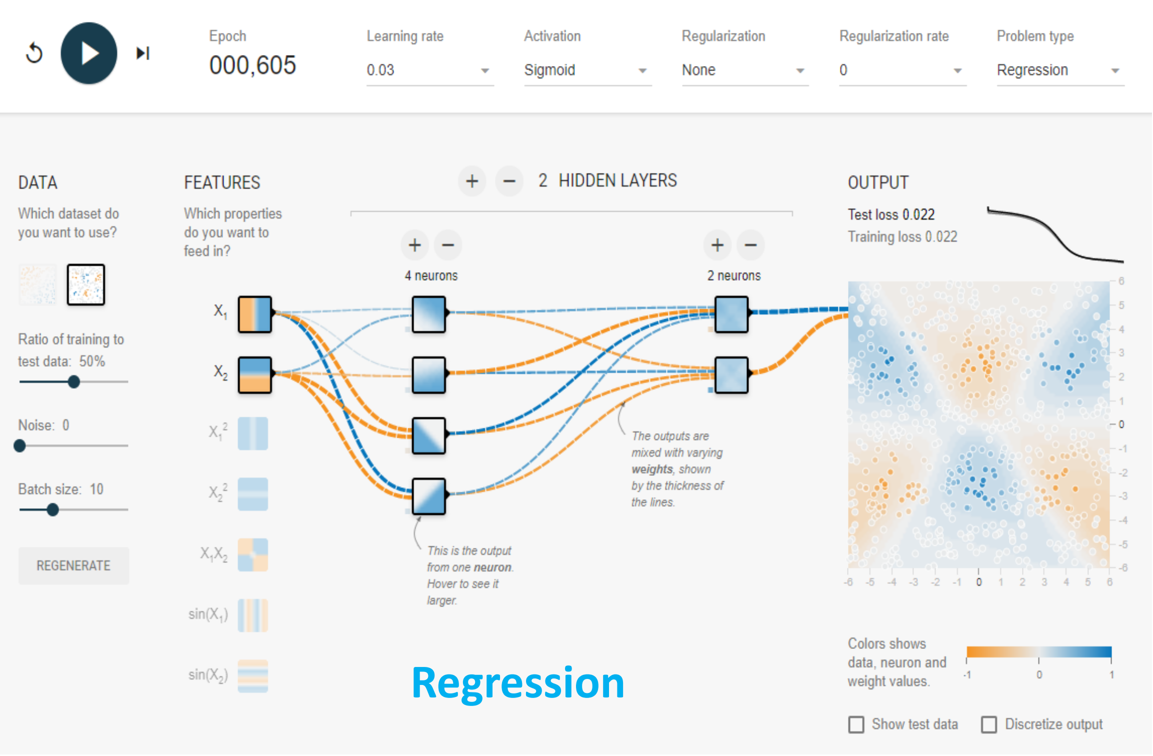
Hyper-Parameter Tuning

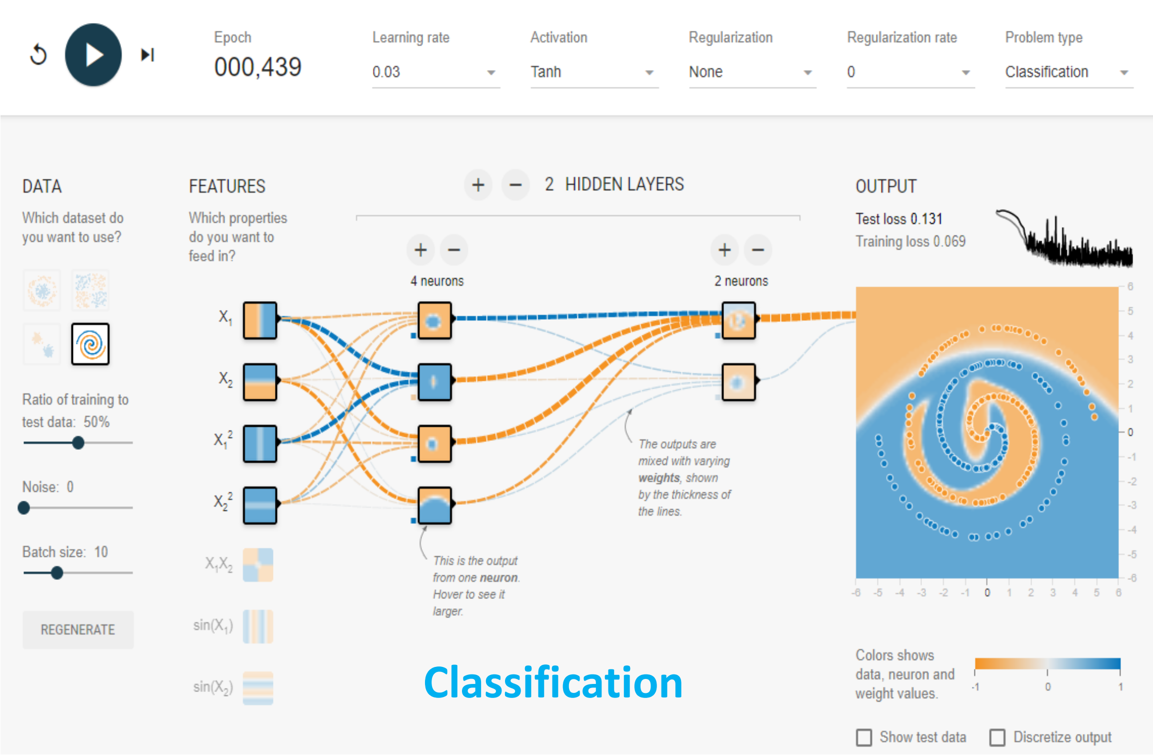
- **Need to optimize network structure and learning parameters**
  - Number of hidden layers
  - Nodes per layer
  - Transfer function/parameters
  - Learning rate/decay
  - Regularization parameters
  - Others …
- **Joint optimization is necessary**
  - Total Enumeration (Grid Search): Can be expensive
  - Random Search: Little more practical
  - Bayesian Optimization: Smart exploration & exploitation
- **Example Packages:**
  - Matlab: bayesopt | Link
  - Python: skopt | Link | Example; bayesopt | Example
  - Keras: Talos | Link
  - Google Cloud: Bayesian Optimization | Link



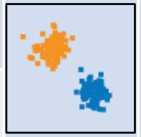
Neural Network Playground: Website

- Daniel Smilkov and Shan Carter created a neural network playground (using TensorFlow) to demystify MLP networks

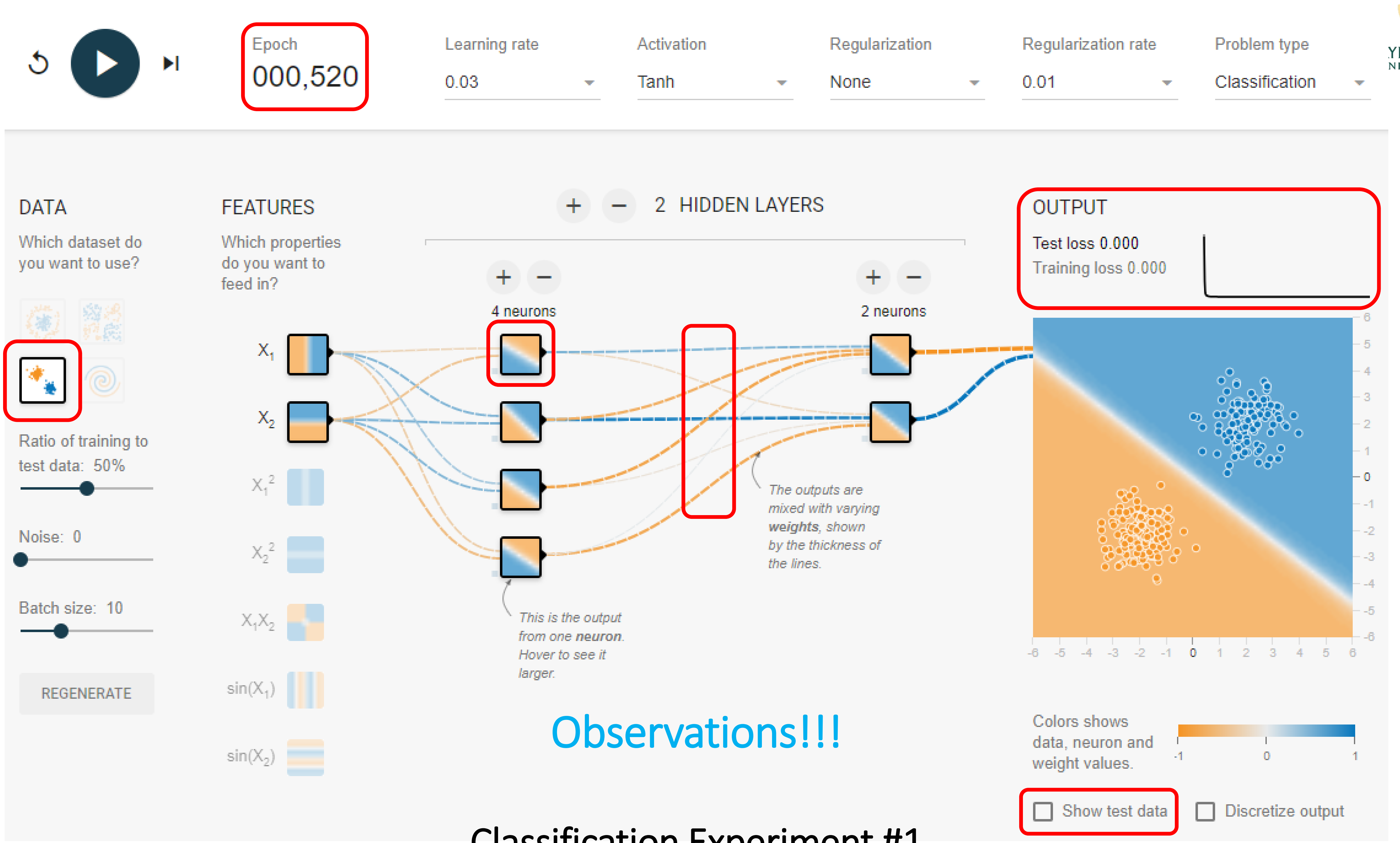




Neural Network Playground: Classification Experiments

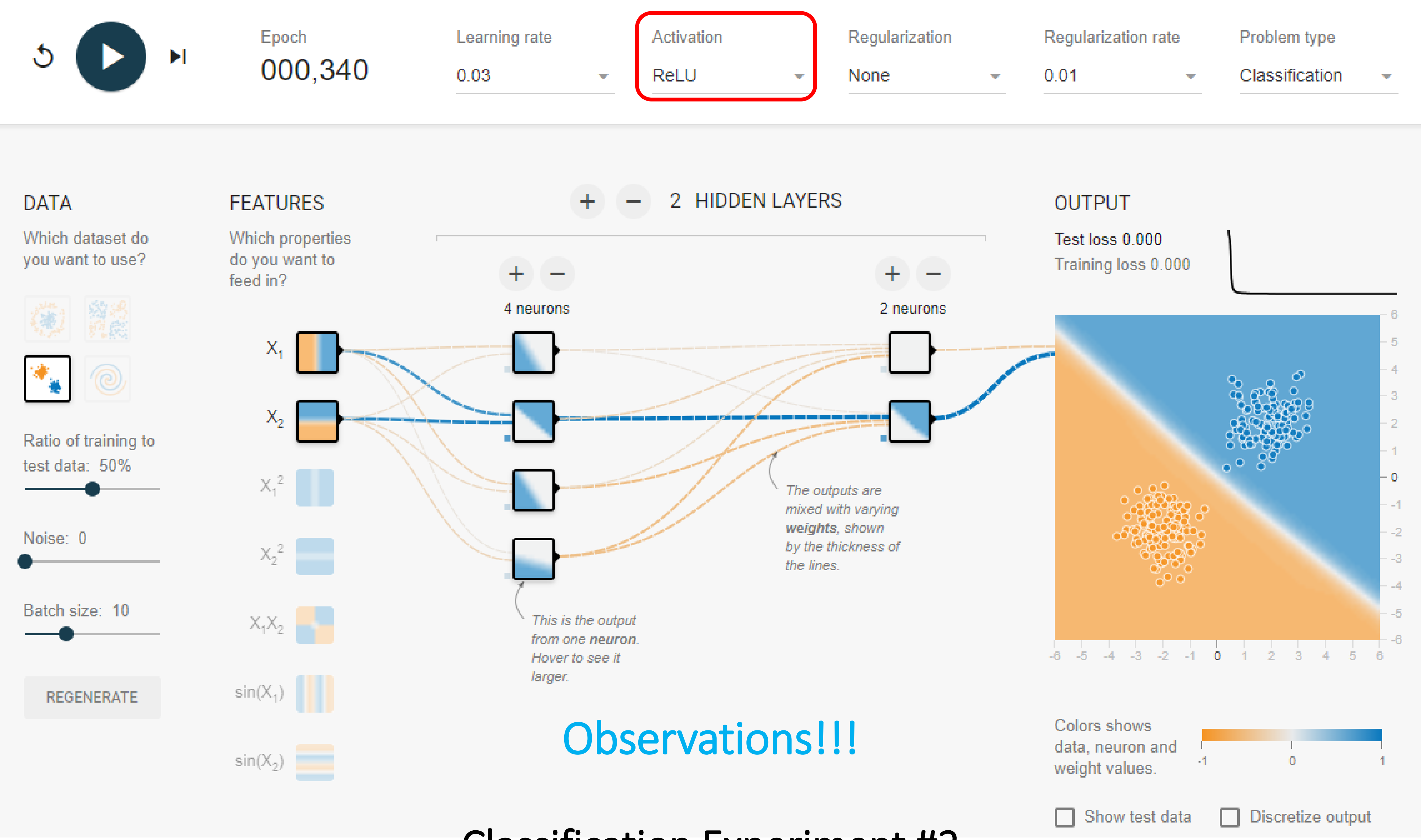
#	Data	Features	Hidden Layers & Nodes	Activation & Learning Rate	Regularization Type & Rate	OBSERVATIONS: Accuracy/Loss (Train/Test), Convergence Rate, Robustness, Others
1		X_1, X_2	2: 4, 2	Tanh, 0.03	None	
2				ReLU, 0.03		
3				Tanh, 0.03	L2, 0.01	
4					L1, 0.01	
5		$+X_1^2, X_2^2, X_1X_2$			None	
6					L2, 0.01	
7		X_1, X_2	2: 4, 2	Tanh, 0.03	None	
8			2: 8, 4			
9			3: 8, 4, 2			
10		$+ Sin(X_1), Sin(X_2)$	2: 4, 2			
11		X_1, X_2	5: 6,5,4,3,2			Batch Size: 30
12			3: 8, 5, 3			

Other Settings: Ratio of Training to Test Data=50%, Noise=0, Batch Size=10



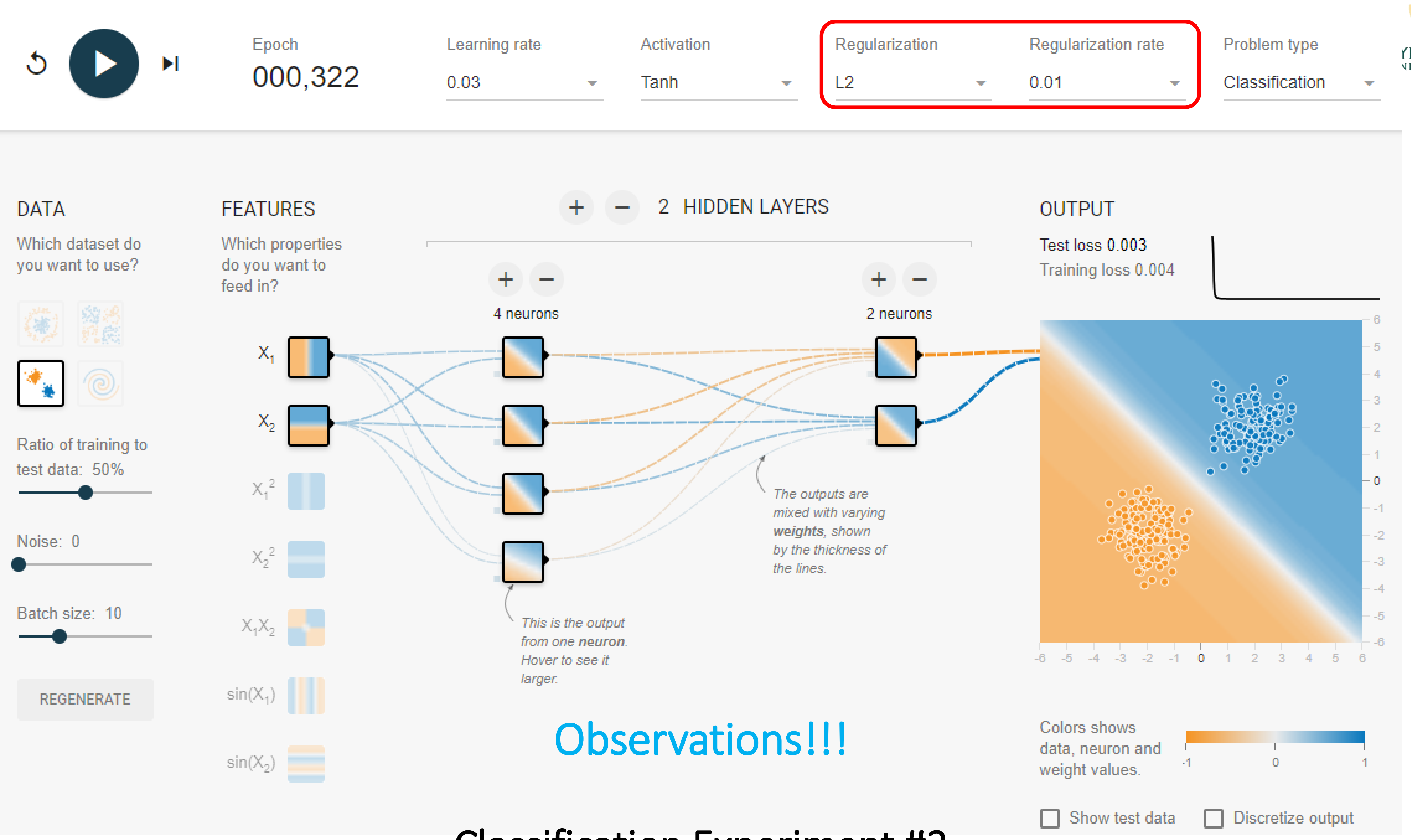
Observations!!!

Classification Experiment #1



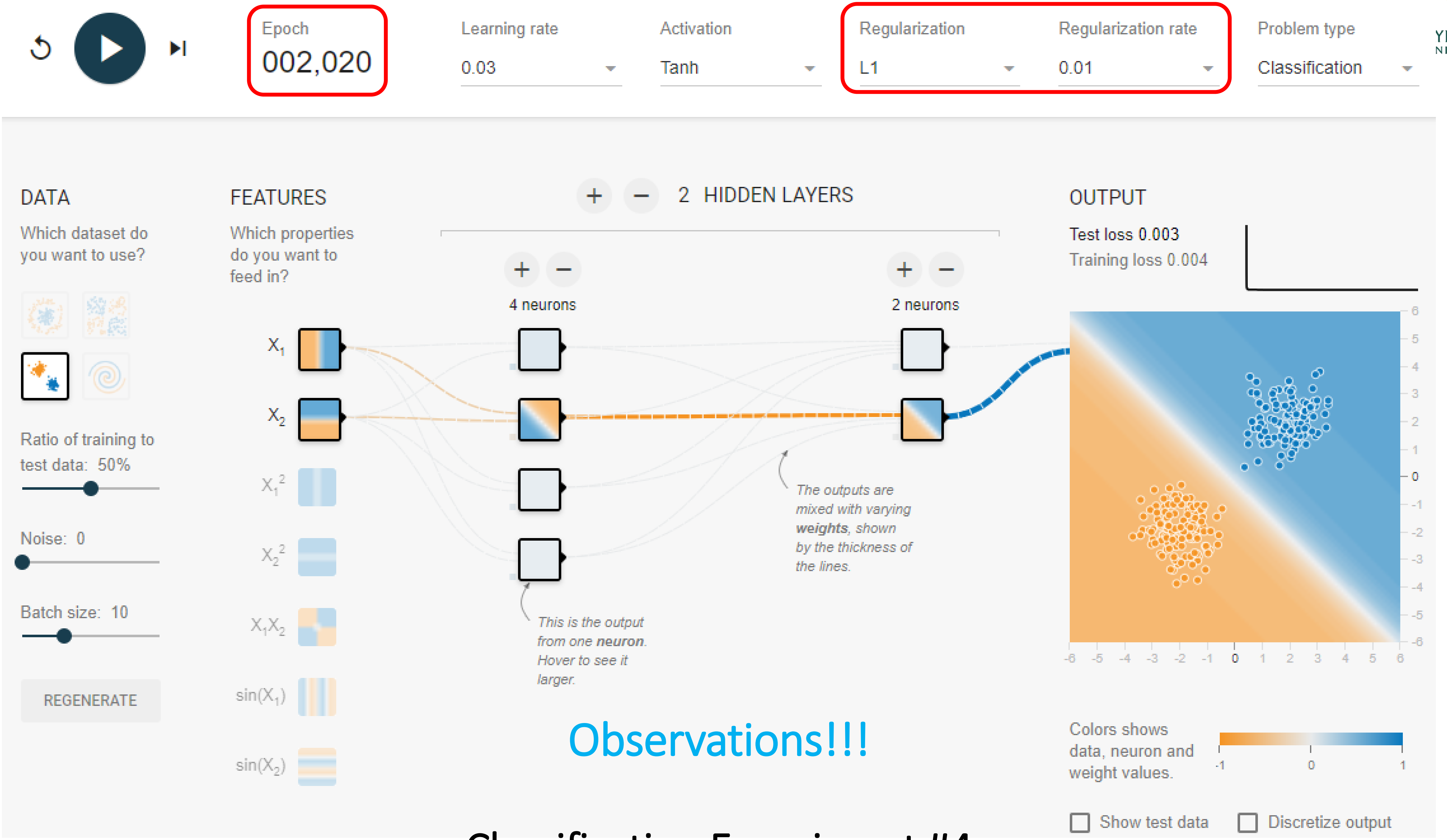
Observations!!!

Classification Experiment #2



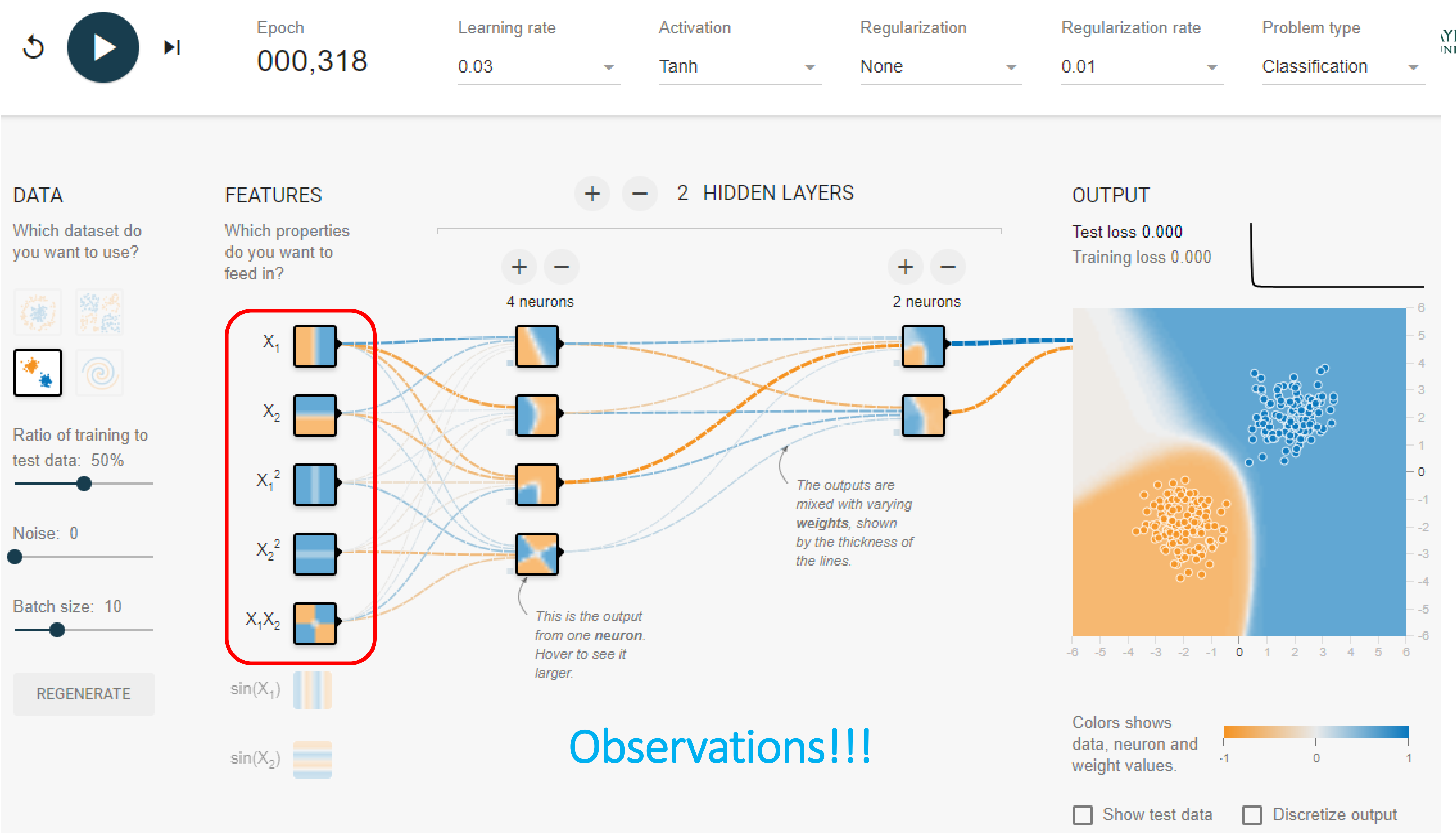
Observations!!!

Classification Experiment #3

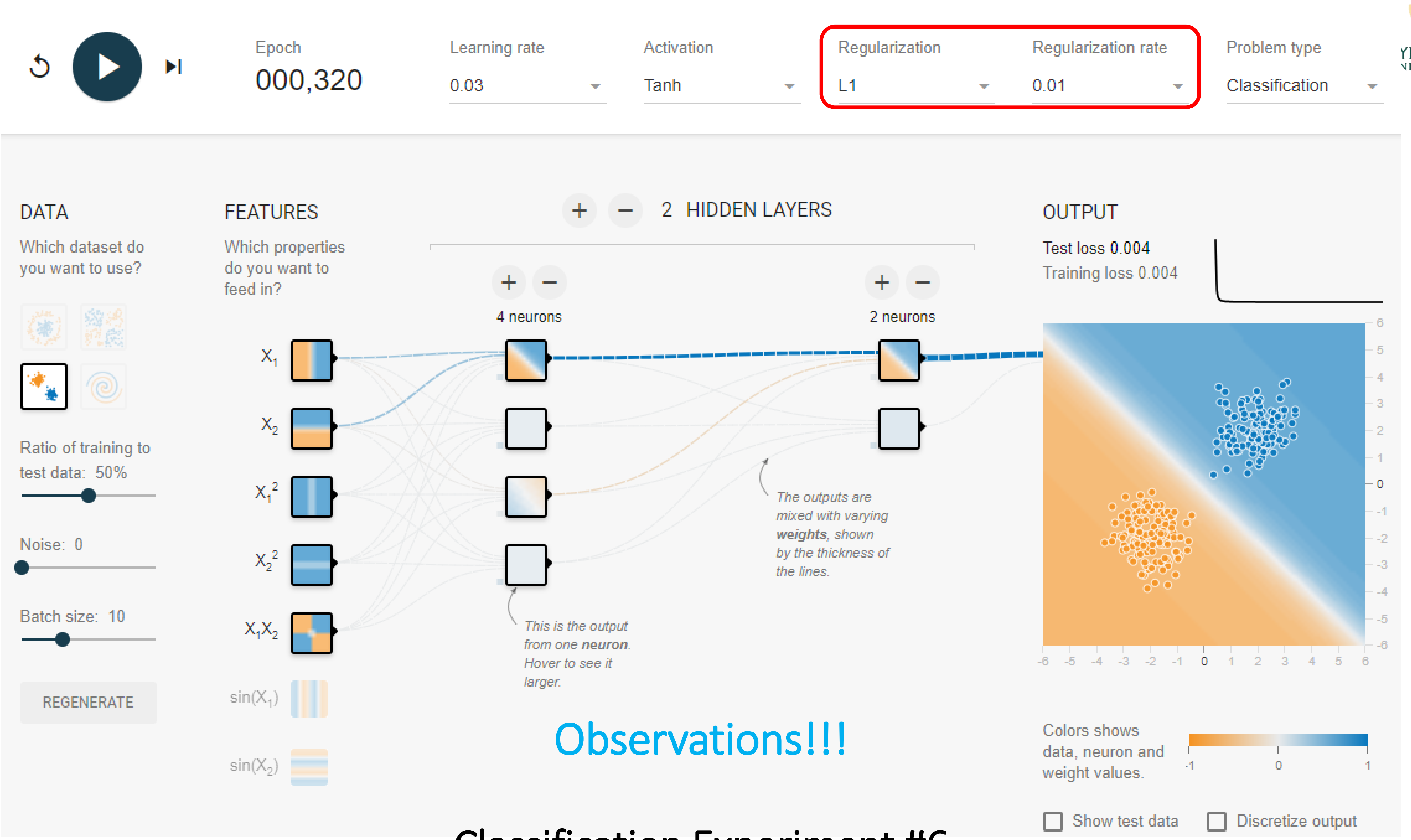


Observations!!!

Classification Experiment #4

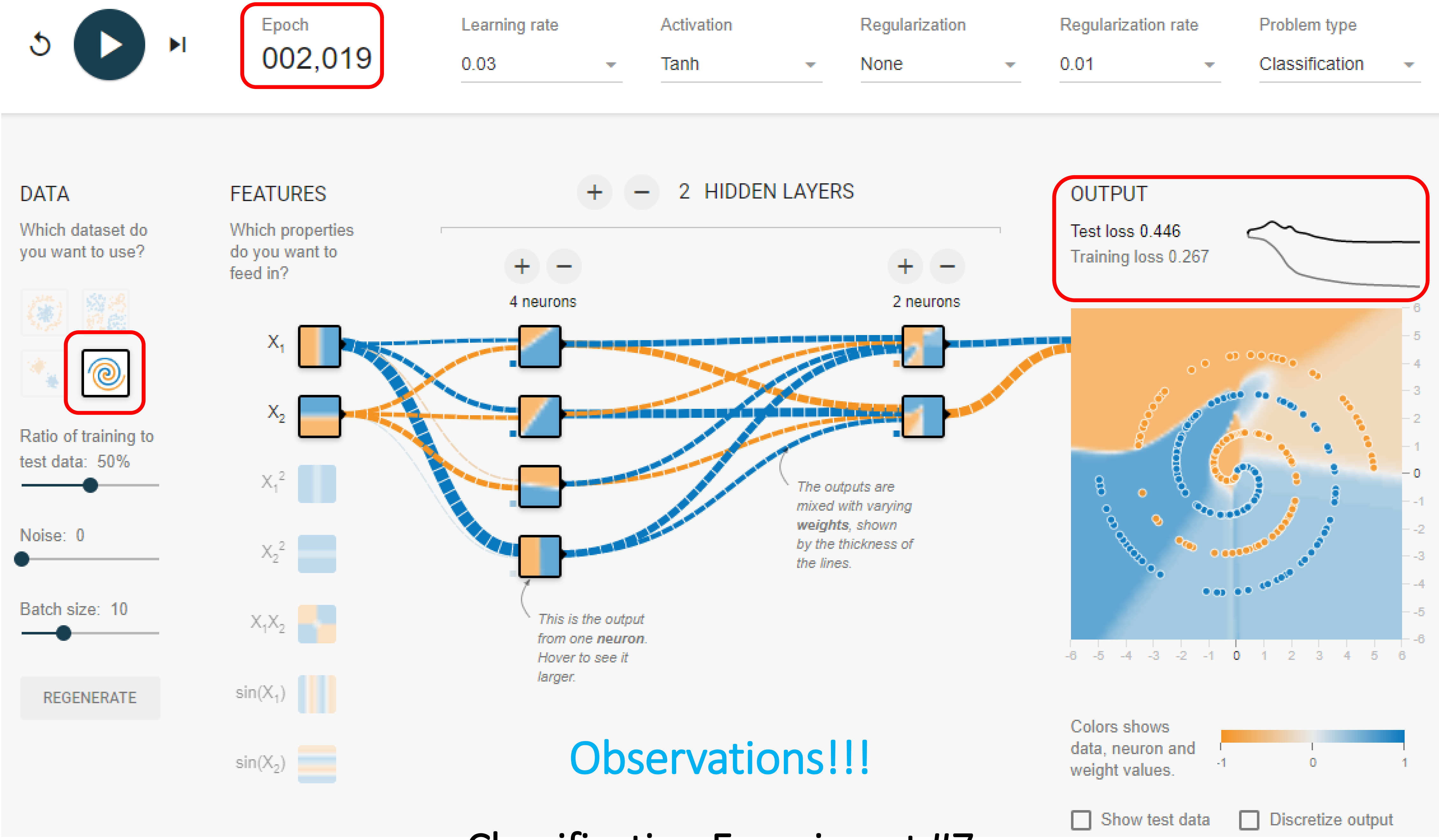


Classification Experiment #5



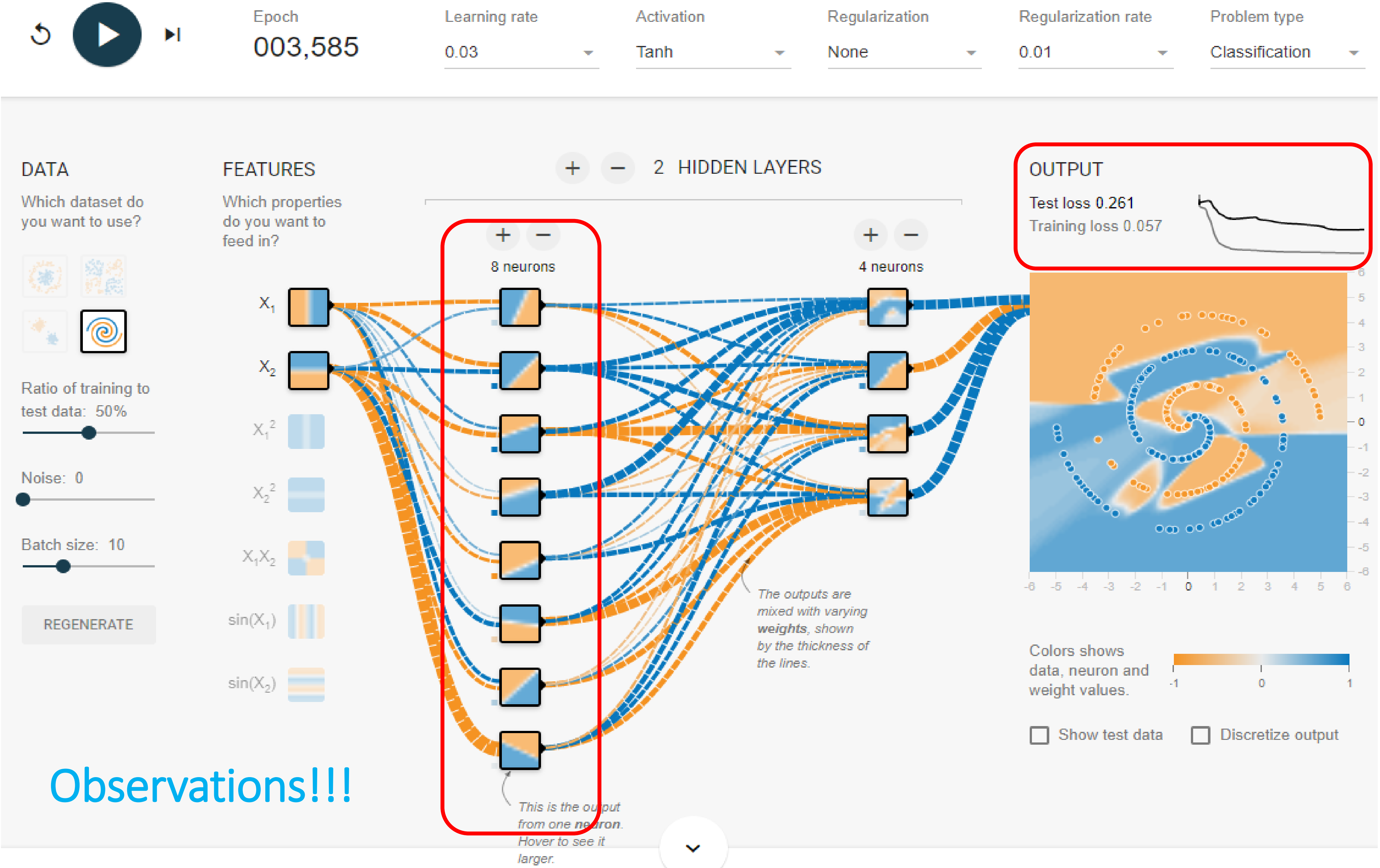
Observations!!!

Classification Experiment #6



Observations!!!

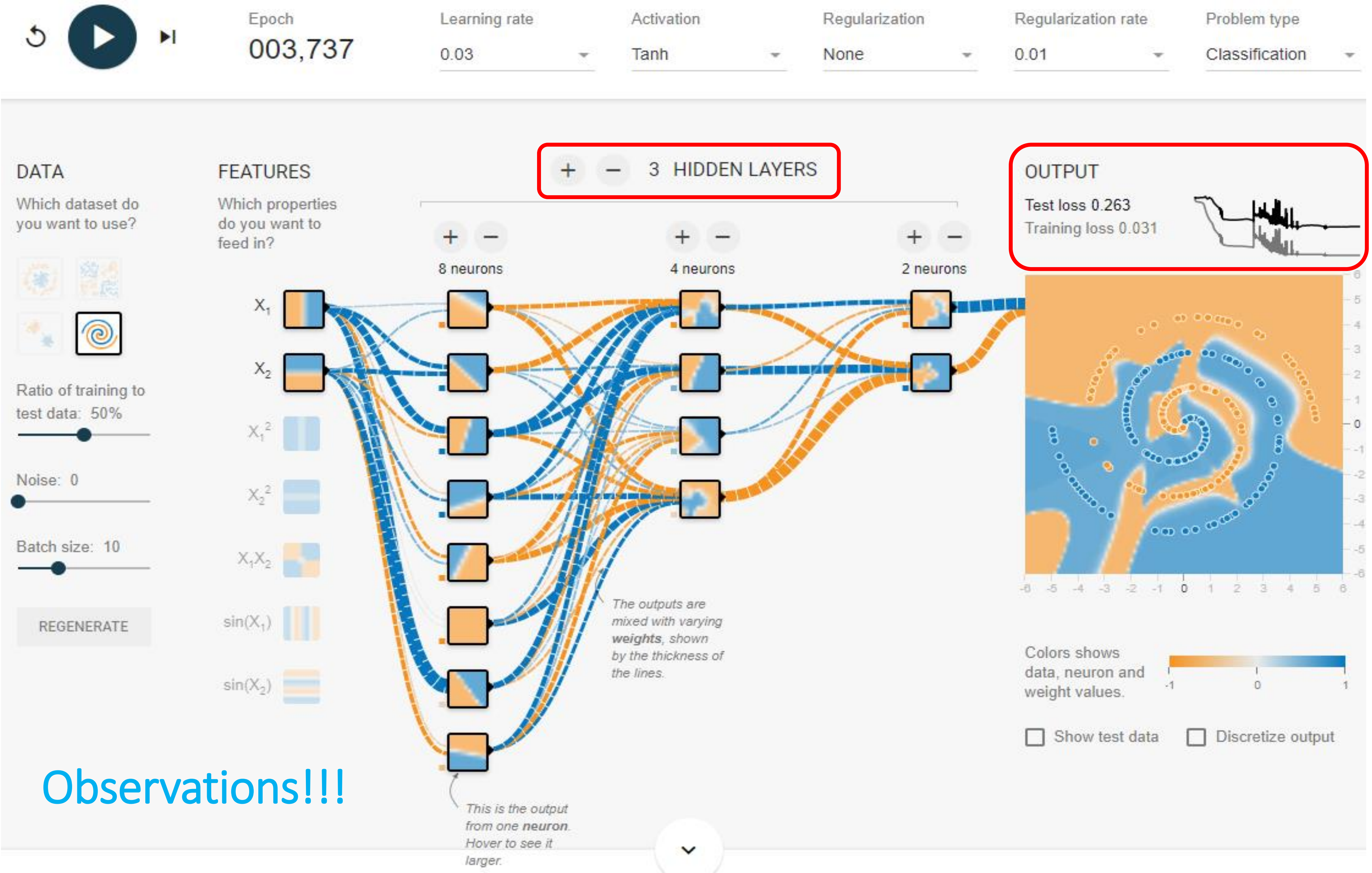
Classification Experiment #7



WAYNE STATE UNIVERSITY

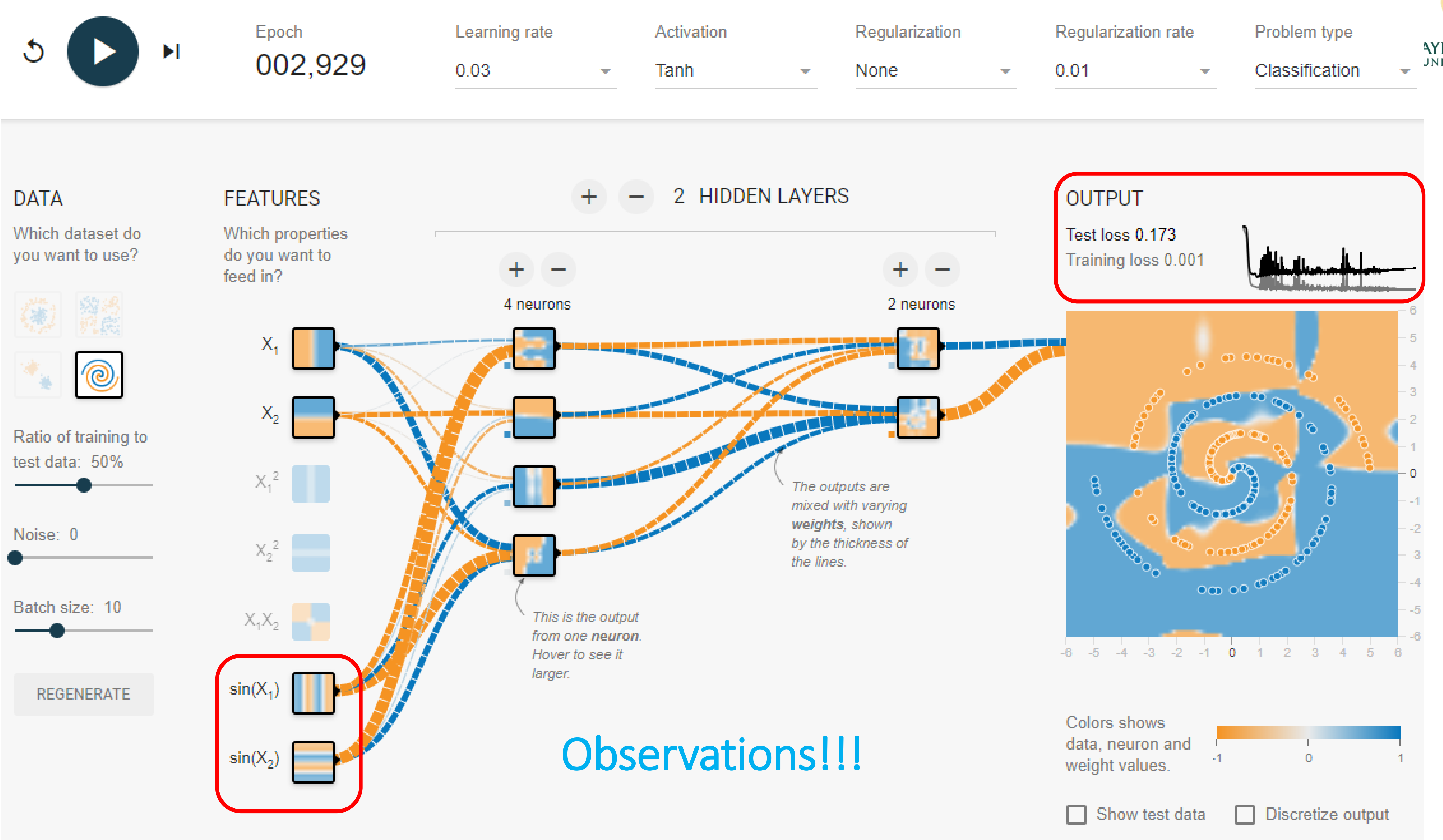
Observations!!!

Classification Experiment #8



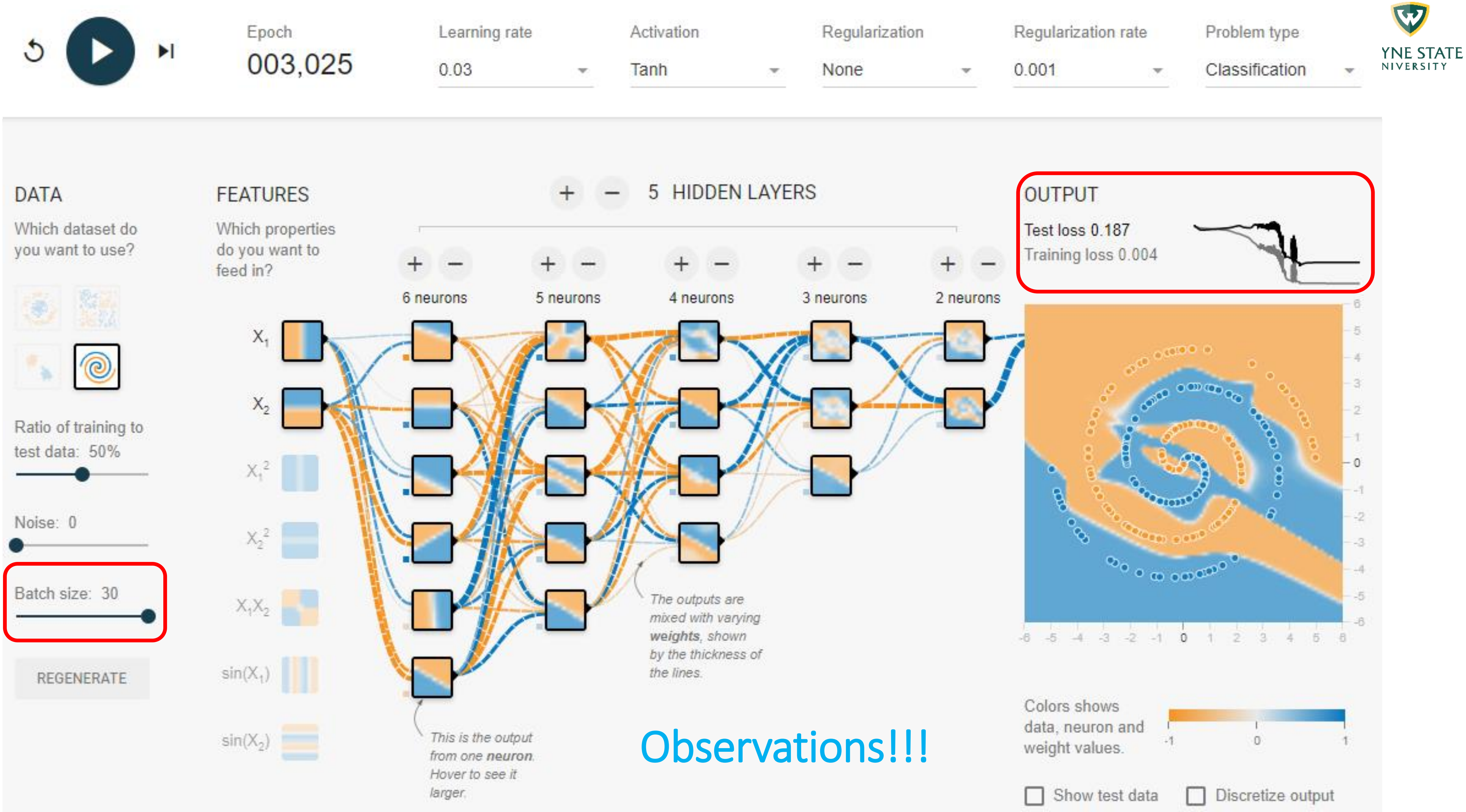
Observations!!!

Classification Experiment #9



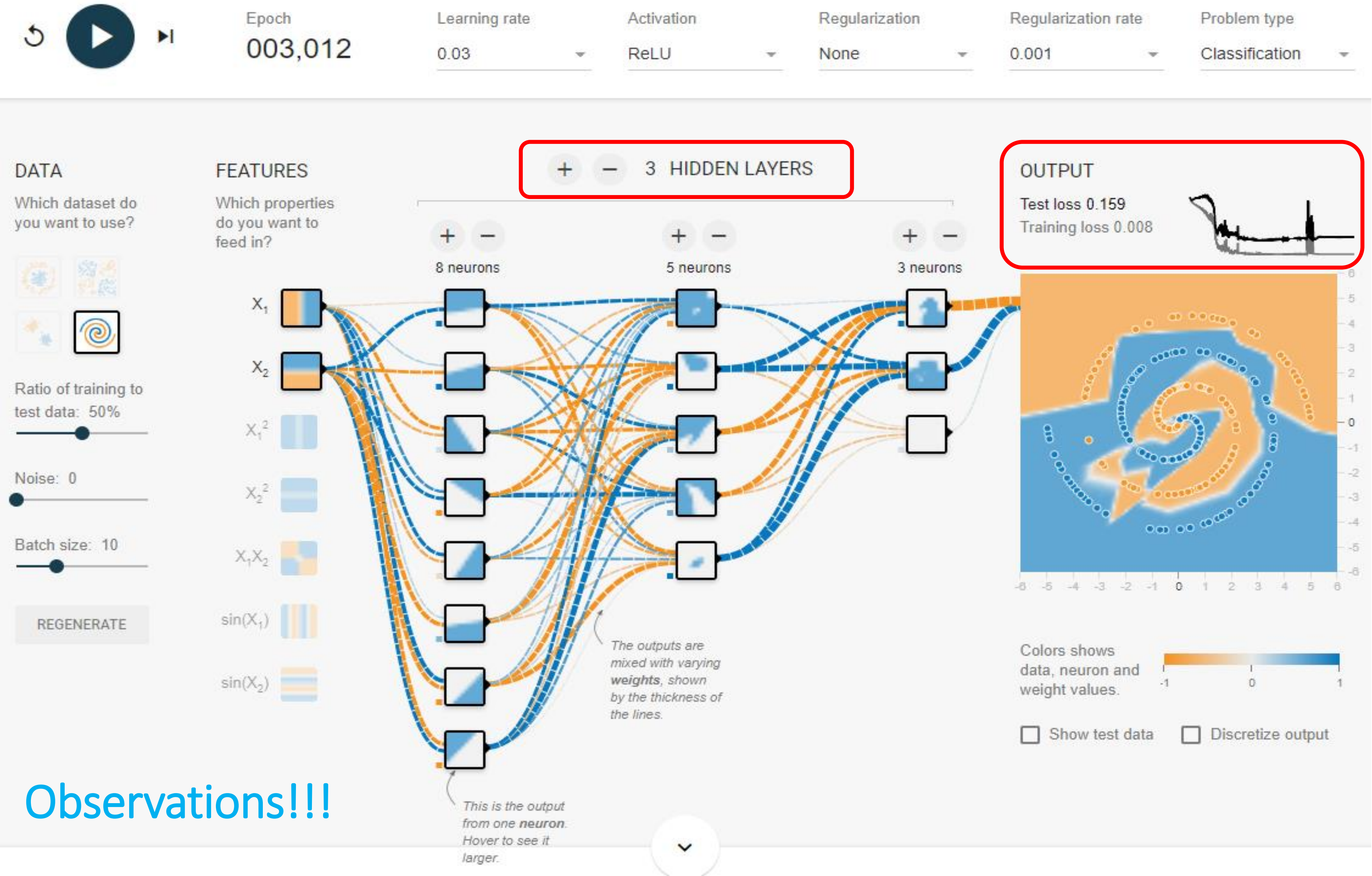
Observations!!!

Classification Experiment #10



Classification Experiment #11

WAYNE STATE UNIVERSITY



Observations!!!

Classification Experiment #12