Neural Net Odds and Ends

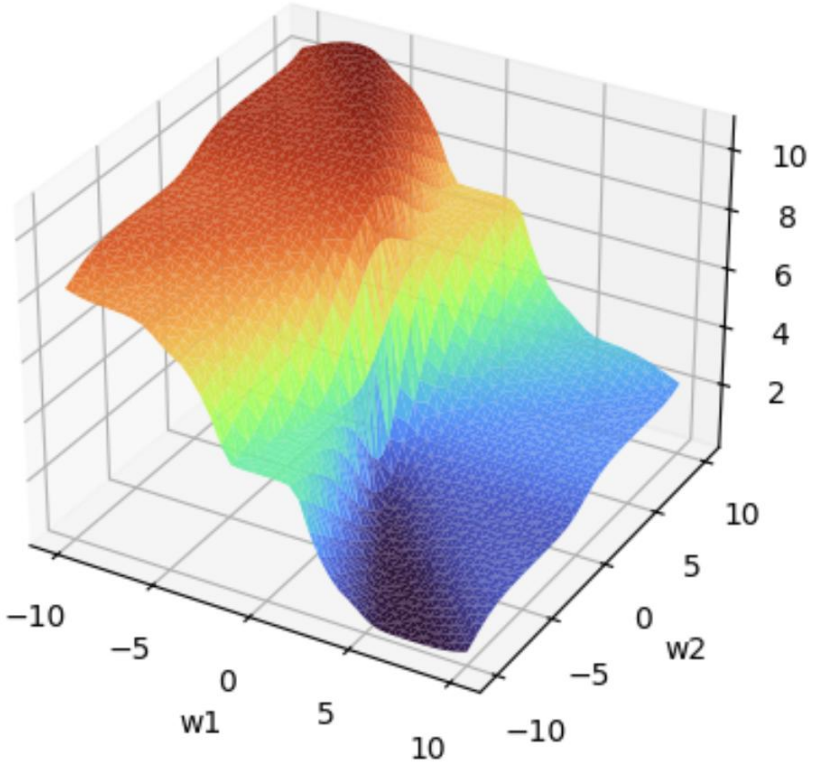
10 March 2026

Alex Lyman

Review

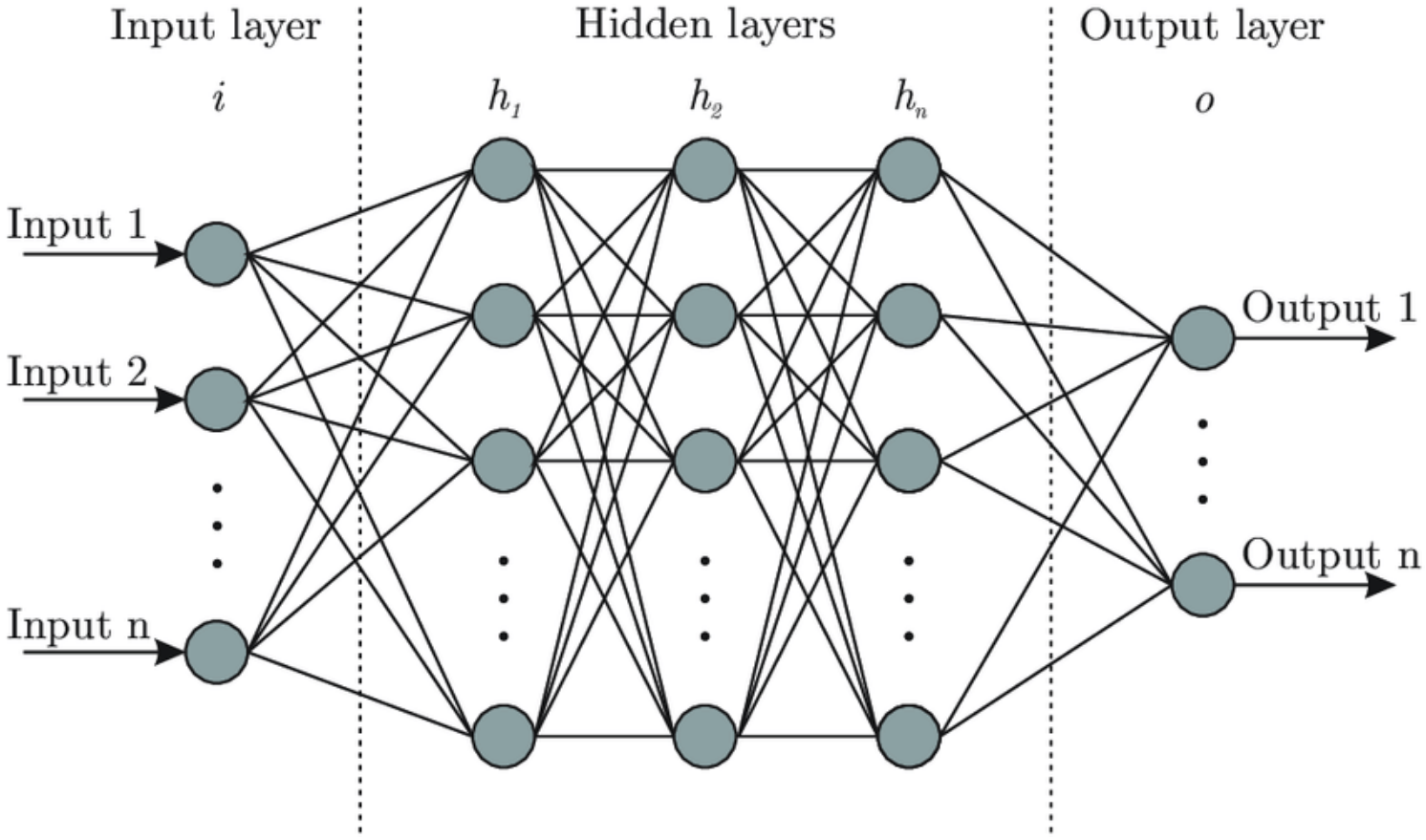
Gradient Descent

- We can plot our error with respect to our weights.
- Finding the lowest point on this graph would minimize error by setting weights to certain values.
- Searching this space is too much work.
- We try to ‘slide down’ to the minimum using **Gradient Descent**
- Snowy mountain at night analogy.



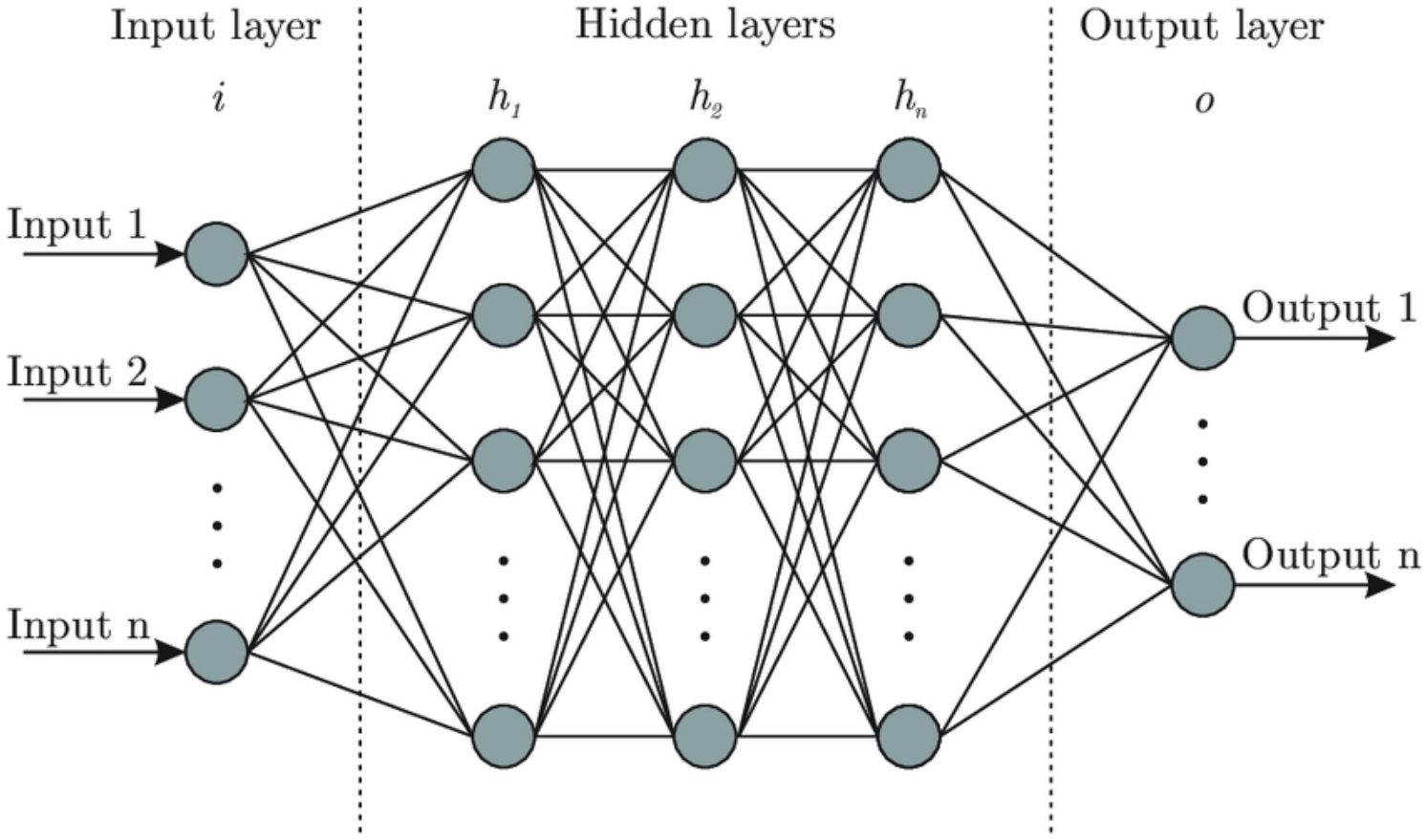
Neural Nets

- Train using Forward and Backward Passes
- Gradient Descent to find error minimum



Neural Net Architecture

- Input layer
- Hidden layers
- Output layers
- Non-linear activation functions let us approximate non-linear functions (better than perceptron)

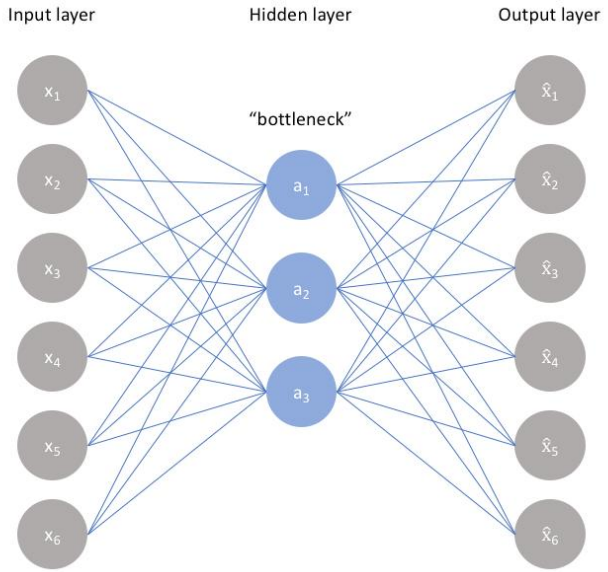


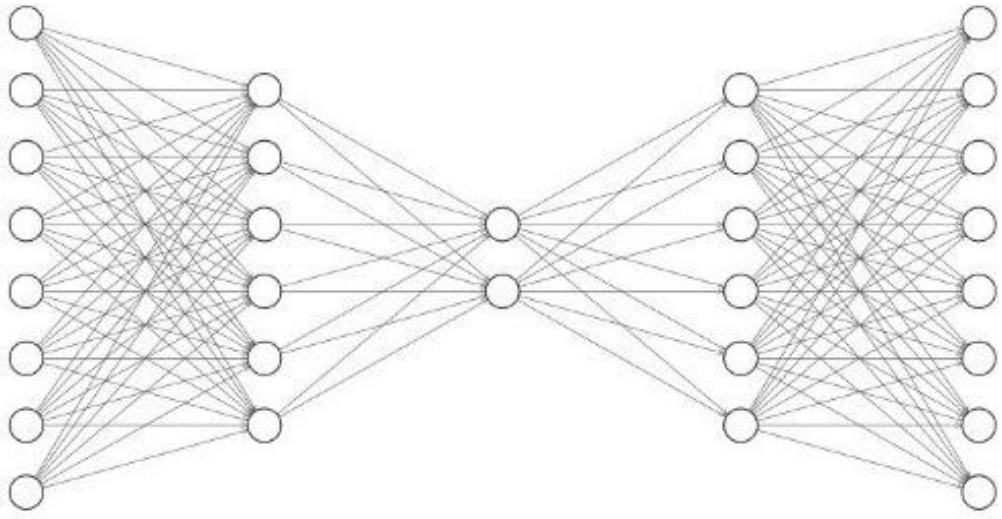
NN Topology

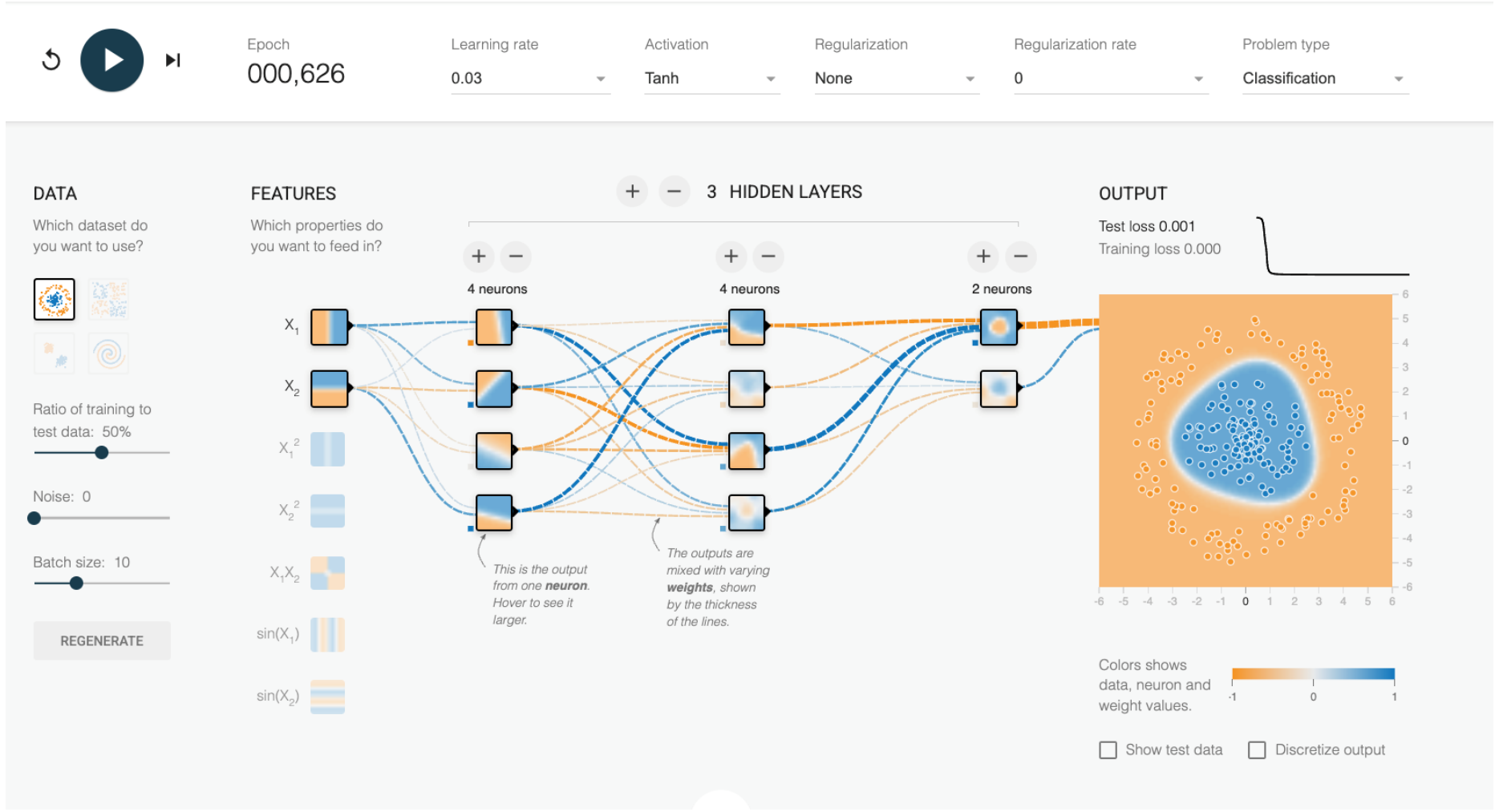
What do Hidden Nodes Do?

What are the Hidden Nodes Doing?

- Higher order features vs 1st order features (perceptron/Us)
  - The real power of machine learning (exponential # of variations)
- Hidden nodes discover new *higher order* features which are fed into subsequent layers
- Compression






Epoch
000,626
Learning rate
0.03
Activation
Tanh
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Noise: 0
Batch size: 10
REGENERATE
FEATURES
Which properties do you want to feed in?
3 HIDDEN LAYERS
4 neurons
4 neurons
2 neurons
This is the output from one neuron. Hover to see it larger.
The outputs are mixed with varying weights, shown by the thickness of the lines.
OUTPUT
Test loss 0.001
Training loss 0.000
Colors shows data, neuron and weight values.
Show test data
Discretize output

Weight Initialization

Inductive Bias & Weight Initialization

- We talked about random initialization – What does that look like? Why do we do it?
- Node Saturation - Avoid early, but all right later
  - With small weights all nodes have low confidence at first (linear range)
  - When saturated (confident), the output changes only slightly as the net changes. An incorrect output node can still have low error.
  - Start with weights close to 0. Once nodes saturate, hopefully have learned correctly, others still looking for their niche.
  - Saturated error even when wrong? – Multiple training set loss drops
  - Don’t start with equal weights (can get stuck), random small Gaussian/uniform with 0 mean
- Inductive Bias
  - Start with simple net (small weights, initially linear changes)
  - Gradually build a more complex until accurate enough without getting too complex

Weight Initialization

- Not available by default in MLPClassifier
- Typically small random values within a range defined by layer sizes
- Can override, but… advanced

```
# new class
class MLPClassifierOverride(MLPClassifier):
# Overriding _init_coef method
def _init_coef(self, fan_in, fan_out):
    if self.activation == 'logistic':
        init_bound = np.sqrt(2. / (fan_in + fan_out))
    elif self.activation in ('identity', 'tanh', 'relu'):
        init_bound = np.sqrt(6. / (fan_in + fan_out))
    else:
        raise ValueError("Unknown activation function %s" %
                         self.activation)
    coef_init = ### place your initial values for coef_init here

    intercept_init = ### place your initial values for intercept_init here

    return coef_init, intercept_init
```

How do we get Output?

MLP for Classification & Regression

- The output of an MLP is a continuous value
- How we use the output determines what we are doing

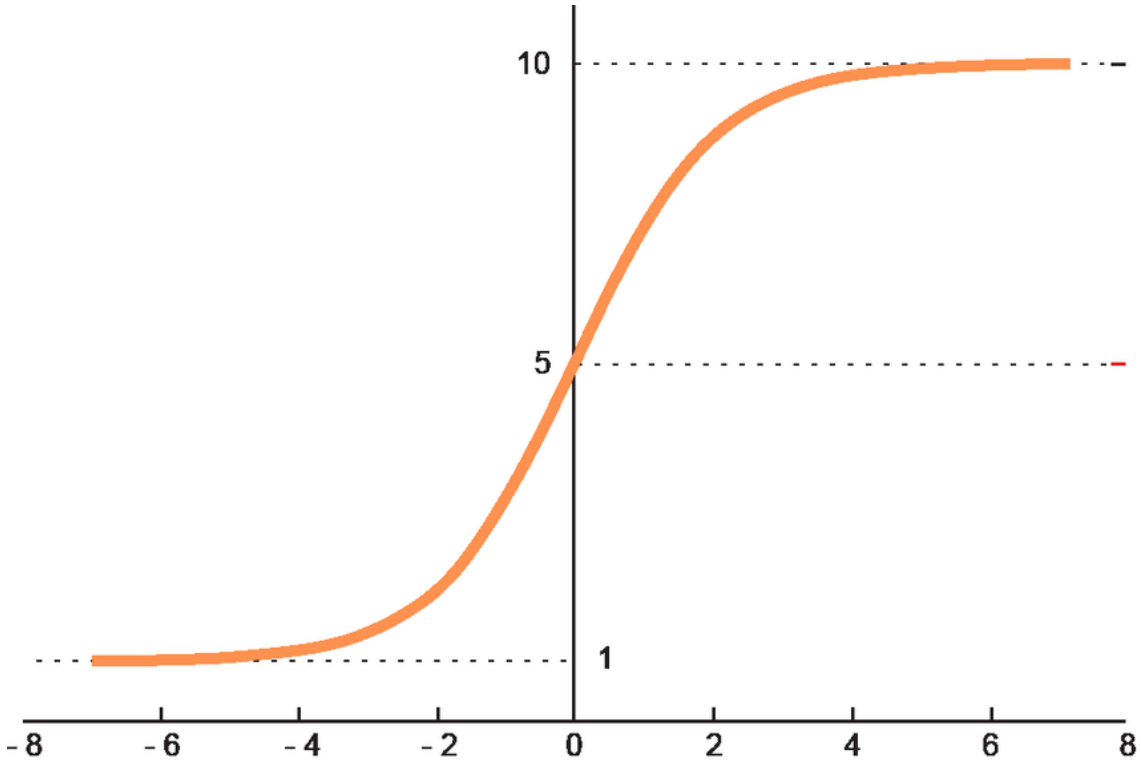
- Classification uses the output to determine a class
  - We must then interpret the output values to classes

- Regression is trying to produce the output value that is close to the value we want

- The way we calculate error should correlate with our task

Classification – the Softmax

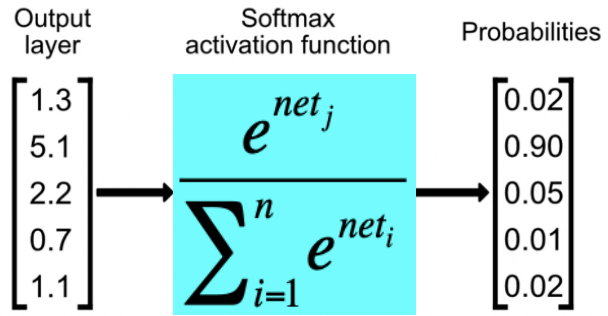
- Softmax is a sigmoid that generalizes to multiple dimensions
- We use it to convert our raw NN outputs (logits) to a probability distribution.



Softmax Output Layer Activation

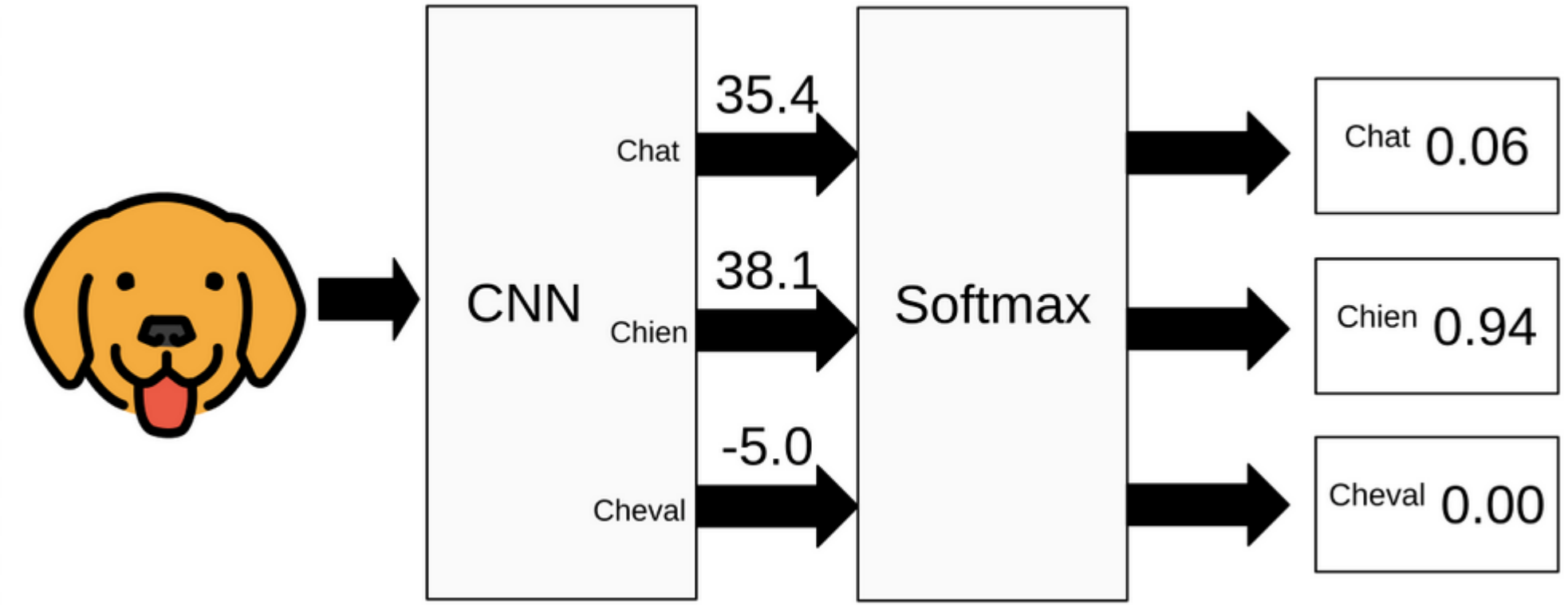
- For *classification* problems we generally use the *softmax* activation function, just at the output layer
- Softmax (softens) 1 of *n* targets to mimic a probability vector for the output nodes

$$f(net_j) = \frac{e^{net_j}}{\sum_{i=1}^{n} e^{net_i}}$$

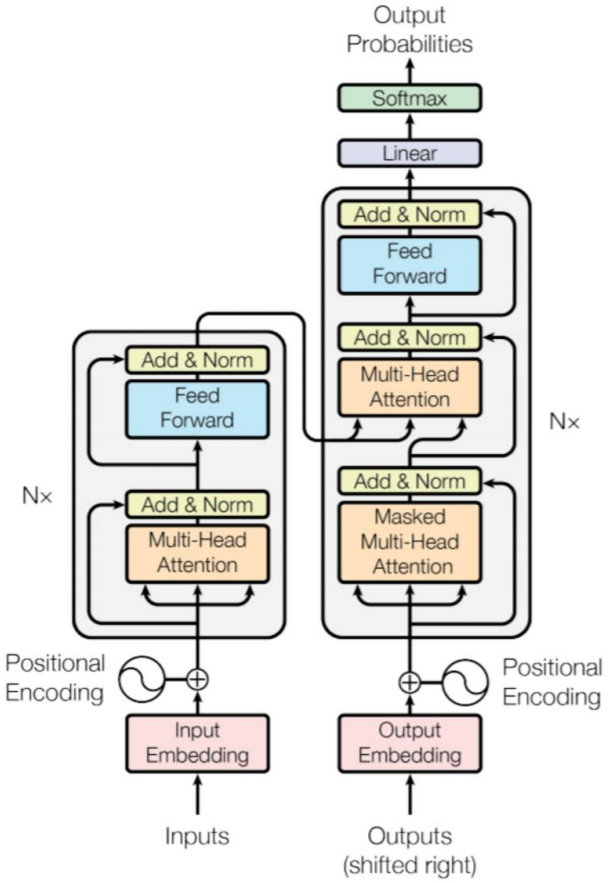


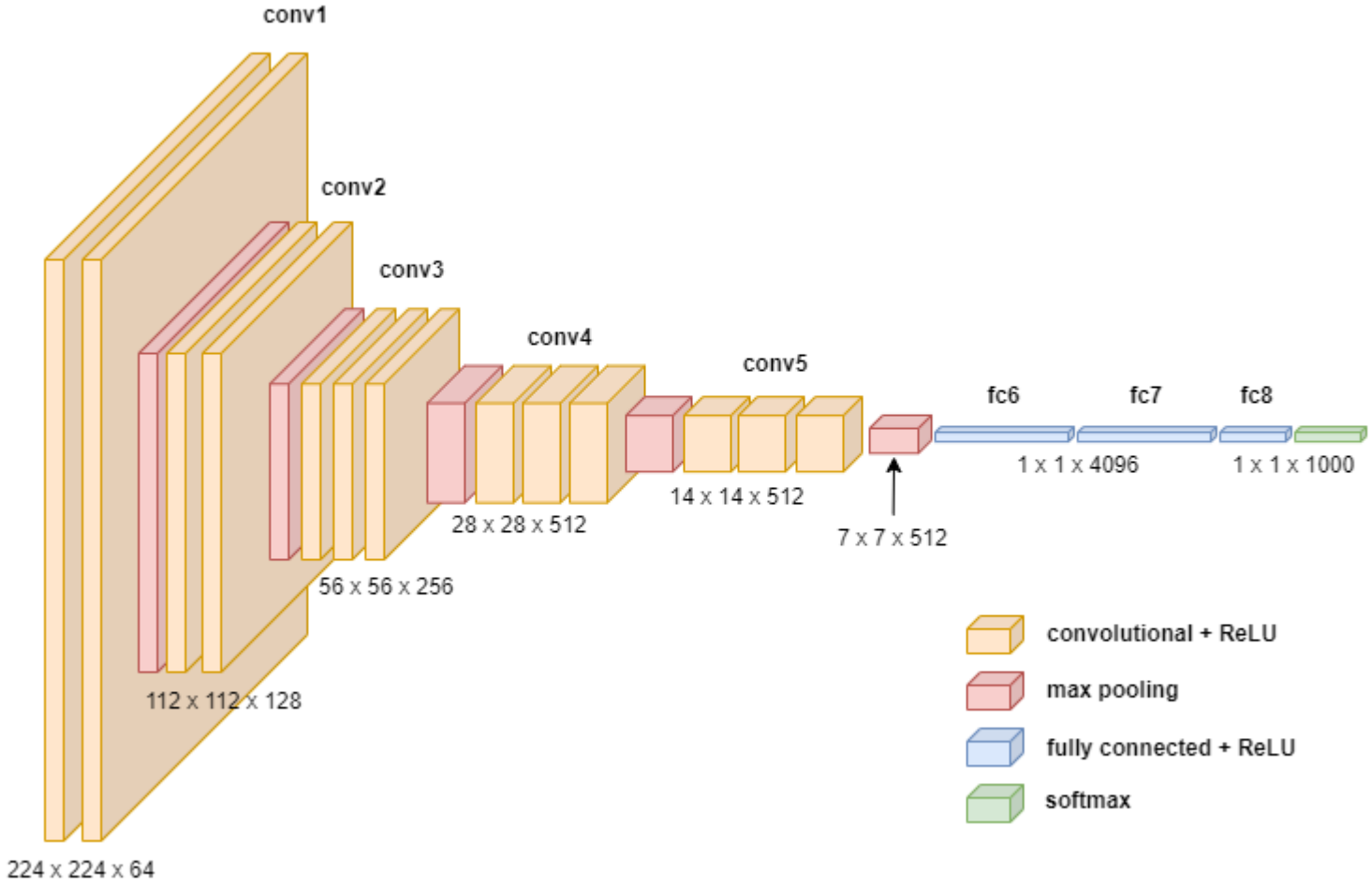
- If there were 3 output nodes with net values 0, .5, and 1 then the outputs for each node would be
  - 1/5.37, 1.65/5.37, 2.72/5.37 = .18, .31, .51
  - sums to 1 and can be considered as probability estimates
- All the hidden nodes still do a standard activation function such as logistic, hyperbolic tangent, or ReLU
- Sklearn automatically uses softmax at the output layer for MLP classification, and you choose the activation function for hidden nodes

Softmax Example 1



Softmaxes in the wild





Regression with MLP/BP

- For regression in MLPs we use the sum-squared error (L2) loss function. More natural for regression than for classification.
- Output nodes use a linear activation (i.e. identity function which just passes the *net* value through). This naturally supports unconstrained regression.
  - Don’t typically normalize output
- The output error is still $(t - z)\, f'(net)$, but since $f'(net)$ is 1 for the linear activation, the output error is just (*target* – *output*)
- Hidden nodes still use a non-linear activation function (such as logistic) with the standard $f'(net)$
- This is how sklearn always does MLP regression

Loss

Loss and Multiclass Classification

- Loss in binary classification is straightforward
  - t - z
- Loss in multiclass classification
  - We need a single number that represents a distance
  - But we have probability distributions
- How do we calculate a distance?

- Information Theory

t	*z*
0	0.2
1	1
1	.9
1	.5
1	.1

Entropy

- Information theory metric that quantifies the number of bits required to transmit or encode an event.
  - Quantifies the *surprise* of an event
  - High probability events are less surprising – less bits needed
  - Low probability events are more surprising – more bits needed
- Calculated as
  - h(x) = - log(P(x)) - negative log because P(x) < 1
- For a set of instances
  - $H(X) = -\sum P(x)\log(P(x))$
- Cross Entropy builds on this idea and calculates the difference in entropy between two probability distributions

Cross-Entropy and Softmax

- Cross-entropy measures the difference (in entropy) between two distributions.
  - $H(P,Q) = -\sum P(x)\log(Q(x))$
  - P(x) becomes t_i
  - Log(P(x)) becomes $\log(z_i)$ or $\ln(z_i)$
- Assumes that the outputs are probability distributions

$$Loss_{CrossEntopy} = -\sum_{i=1}^{n} t_i \ln(z_i)$$

t	*z*	CE
0	0.01	0
1	1	0
1	.9	.11
1	.5	.69
1	.1	2.30
		3.1

- Use the cross entropy as the loss for output nodes
  - Replaces (t – z) and there is no *f '(net)*
- The hidden layers still update as usual and include *f* '(*net*) for their activation function
- Sklearn always uses this approach for MLP classification

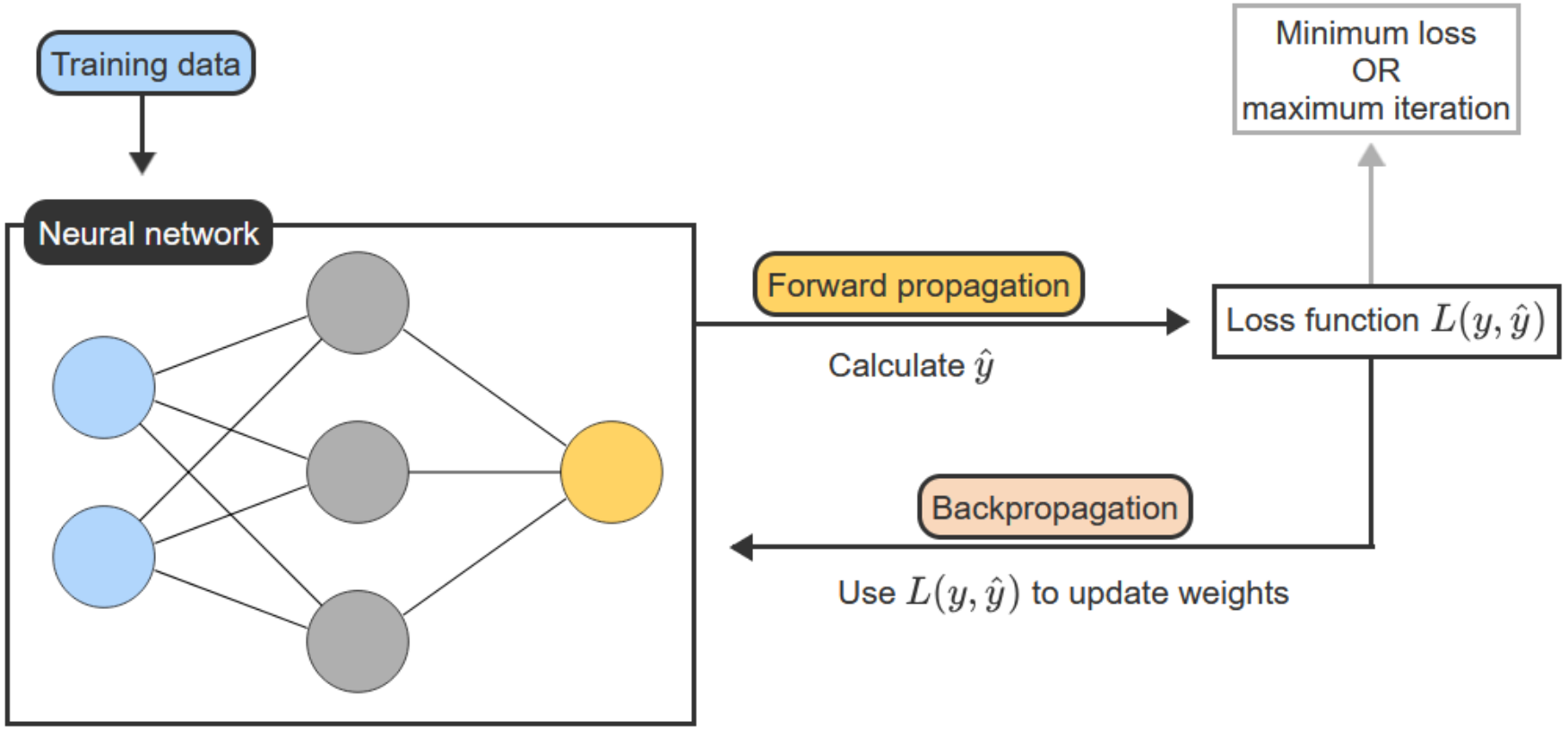
Network Size/Shape

Number of Hidden Nodes

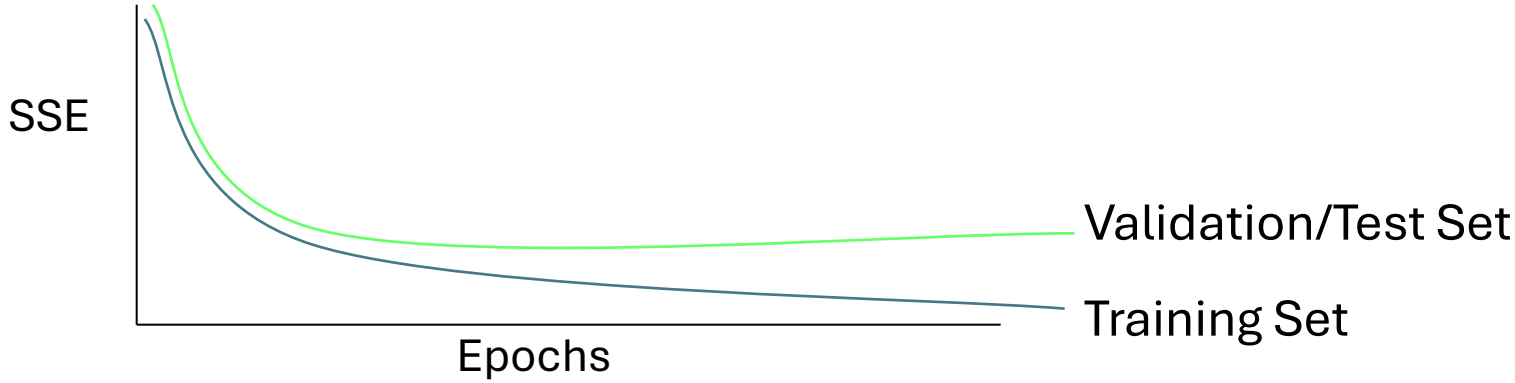
- How many needed is a function of how hard the task is
- Common to use one fully connected hidden layer. Initial number could be ~2n hidden nodes where n is the number of inputs.
- In practice we train with a small number of hidden nodes, then keep doubling, etc. until no more significant improvement on test sets
  - Too few will underfit
  - Too many nodes can make learning slower and could overfit
    - Having somewhat too many hidden nodes is preferable if using reasonable regularization; avoids underfit and should ignore unneeded nodes
- Each output and hidden node should have its own bias weight

Training Neural Nets

Training Neural Nets



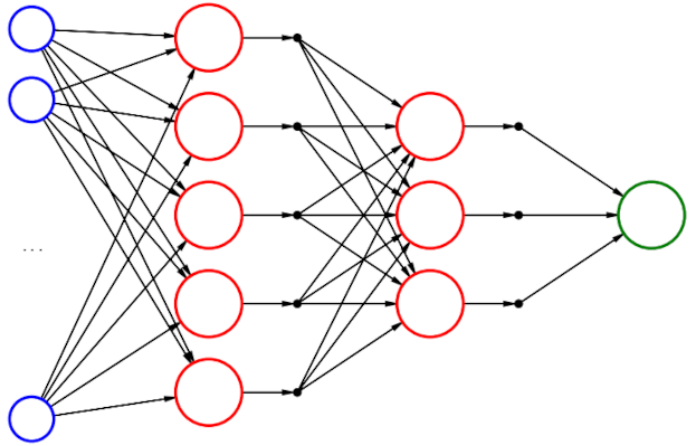
How do We Know When to Stop?



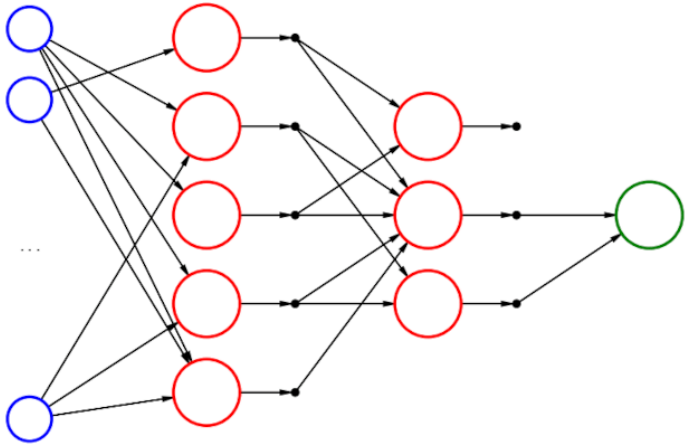
- More Training Data (vs. overtraining - One epoch in the limit)
- Validation Set
- Test accuracy after every n epochs to avoid overfit.
- How else can we avoid overfitting?
  - Dropout
  - Regularization

Dropout

- Fully connected Neural Networks love to overfit to data.
- Randomly slicing off some connections (dropout) helps a lot!



Regular Connections



Applying Dropout

Dropout

```
class torch.nn.Dropout(p=0.5, inplace=False) #
```

Improving neural networks by preventing co-adaptation of feature detectors

G. E. Hinton*, N. Srivastava, A. Krizhevsky, I. Sutskever and R. R. Salakhutdinov
Department of Computer Science, University of Toronto,
6 King's College Rd, Toronto, Ontario M5S 3G4, Canada

Regularization in MLP

- Remember: Regularization refers to a method to avoid overfitting
- Methods:
  - Data
    - Get more data
    - Get better data – more representative
  - Early stopping
  - Model methods – keep the model simple
  - Loss regularization – L1 and L2
  - Dropout

Backpropagation Regularization

- How to avoid overfit – Keep the model simple
  - Keep decision surfaces smooth
  - Smaller overall weight values lead to simpler models with less overfit
- Early stopping with validation set is a common approach to avoid overfitting (since weights don't have time to get too big)
- Could make complexity an explicit part of the loss function
  - Then we don’t need early stopping (though sometimes one is better than the other and we can even do both simultaneously)
- Regularization approach: Model (h) selection
  - Minimize $F(h) = Error(h) + \lambda \cdot Complexity(h)$
  - Tradeoff accuracy vs complexity
- Two common approaches
  - Lasso (L1 regularization)
  - Ridge (L2 regularization)

L1 (Lasso) Regularization

- Standard BP update is based on the derivative of the loss function with respect to the weights. We can add a model complexity term directly to the loss function such as:
  - $L(\boldsymbol{w}) = \text{Error}(\boldsymbol{w}) + \lambda\Sigma|w_i|$
  - λ is a hyperparameter which controls how much we value model simplicity vs training set accuracy
  - Gradient of $L(\boldsymbol{w})$: Gradient of Error($\boldsymbol{w}$) + λ
  - To make it gradient descent we negate the Gradient: ($-\nabla$error($\boldsymbol{w}$) $-\lambda$)
    - This is also called weight decay
    - Gradient of Error is just equations we have used if Error($\boldsymbol{w}$) is TSS, but may differ for other error functions
- Weights that really should be significant stay large enough, but weights just being nudged by a few data instances go to 0

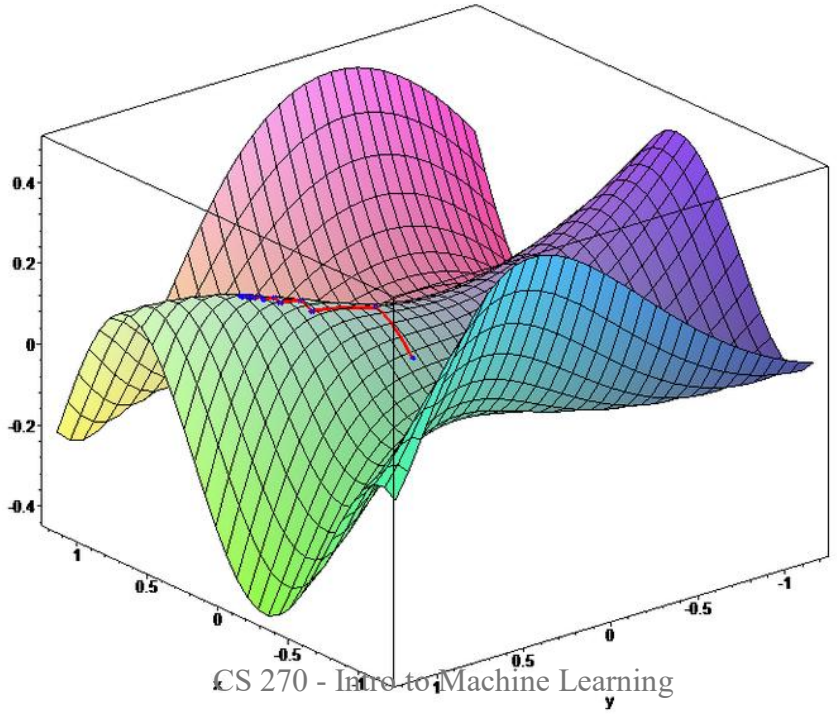
L2 (Ridge) Regularization

- $L(\boldsymbol{w})$ = Error($\boldsymbol{w}$) + $\lambda \Sigma w_i^2$
- -Gradient of $L(\boldsymbol{w})$: -Gradient of Error($\boldsymbol{w}$) - $2\lambda w_i$
- Regularization portion of weight update is scaled by weight value (fold 2 into λ)
  - Decreases change when weight small (<0), otherwise increases
  - λ is % of weight change, .03 means 3% of the weight is decayed each time
- L1 vs L2 Regularization
  - L1 drives many weights all the way to 0 (Sparse representation and feature reduction)
  - L1 more robust to large weights (outliers), while L2 makes larger decay with large weights
  - L1 leads to simpler models, but L2 often more accurate with more complex problems which require a bit more complexity

Neural Network Issues

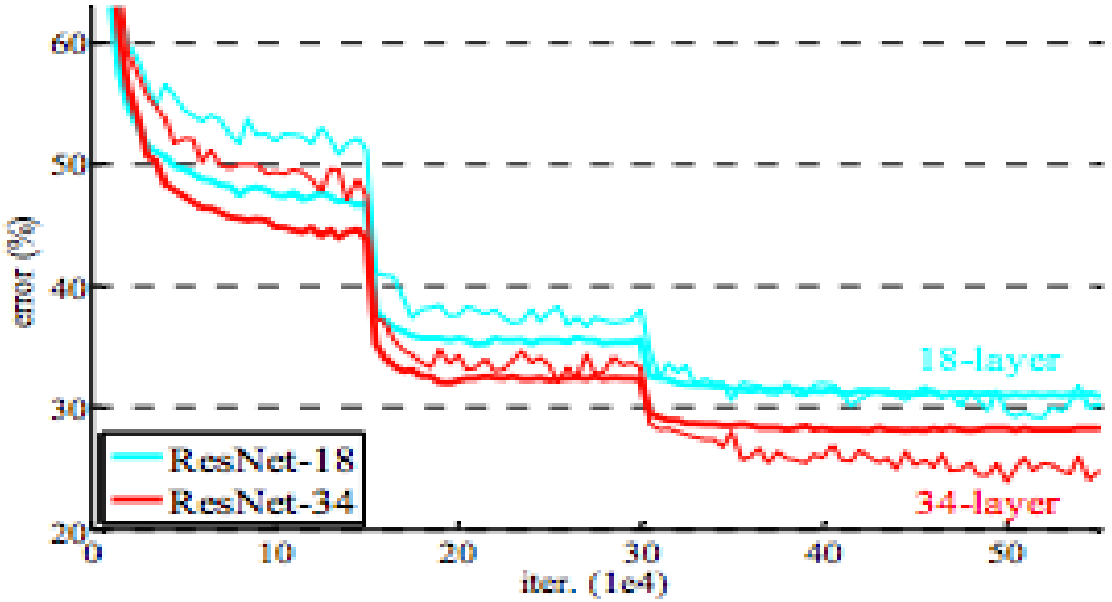
Local Minima and Neural Networks

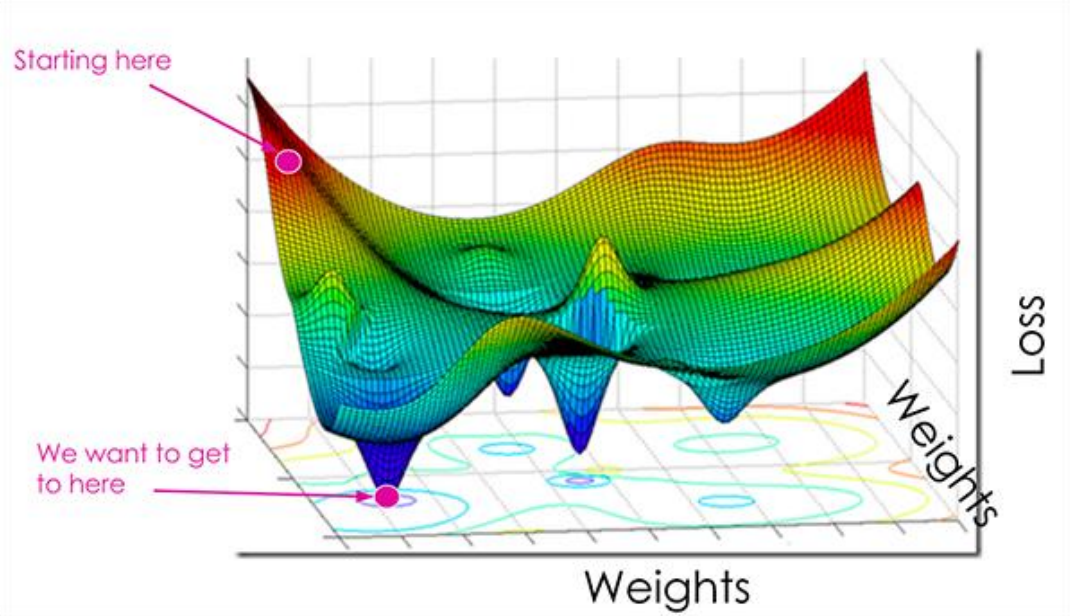
- Neural Network can get stuck in local minima for small networks, but for most large networks (many weights), local minima rarely occur in practice
- This is because with so many dimensions of weights it is unlikely that we are in a minima in every dimension simultaneously – almost always a way down



Local Minima

- Most algorithms which have difficulties with simple tasks get much worse with more complex tasks
- Good news with MLPs
- Many dimensions make for many descent options
- Local minima more common with simple/toy problems, rare with larger problems and larger nets
- Even if there are occasional minima problems, could simply train multiple times and pick the best
- Some algorithms add noise to the updates to escape minima


60
50
40
30
20
error (%)
0
10
20
30
40
50
iter. (1e4)
18-layer
34-layer
ResNet-18
ResNet-34


Starting here
We want to get
to here
Loss
Weights
Weights

Vanishing and Exploding Gradients

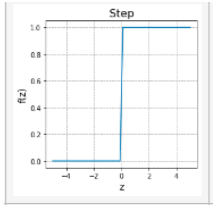
- When we do a pass through the network, we multiply gradients across each layer.
- If our activation derivatives are small, multiplying small numbers together makes our gradients disappear as we work through the network!
- If our values are too large, (much larger than one), the gradients will explode to an enormous number.
- ReLU solves these (derivative is 1 for positive numbers)

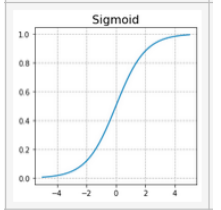
MLP Hyperparameters

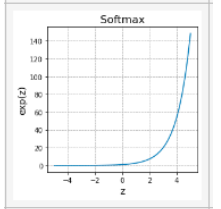
Activation Functions

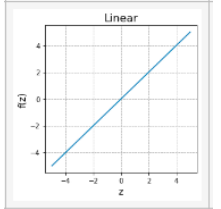
Activation Function

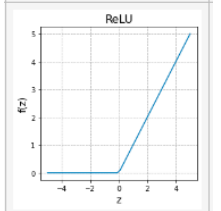
- MLPClassifier has 4 options
  - identity
  - logistic (sigmoid)
  - tanh (hyperbolic tangent)
  - relu (rectified linear unit)
- logistic (sigmoid) was the choice in the 1990s, tanh became the choice in the early 2000s
  - tanh can perform better than sigmoid
  - Both can suffer from vanishing gradient
- ReLU is most common now
  - Allows for discontinuities
- New ones are being used more – leaky relu

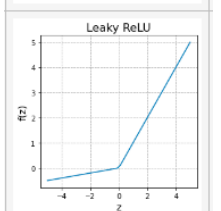


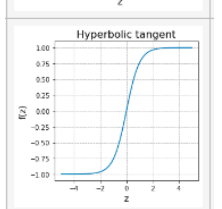




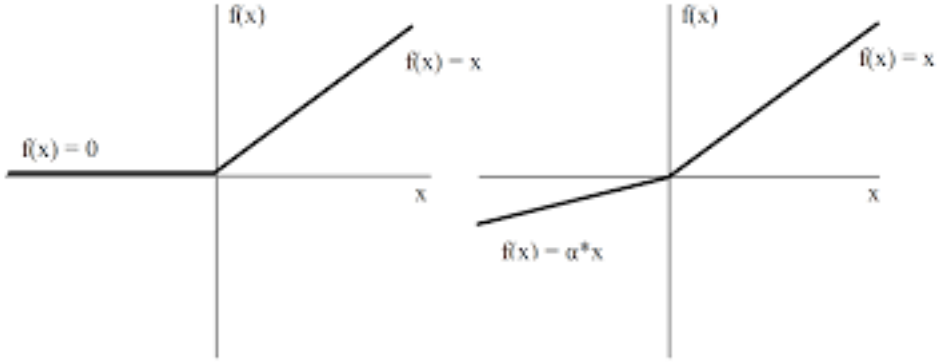








Rectified Linear Units

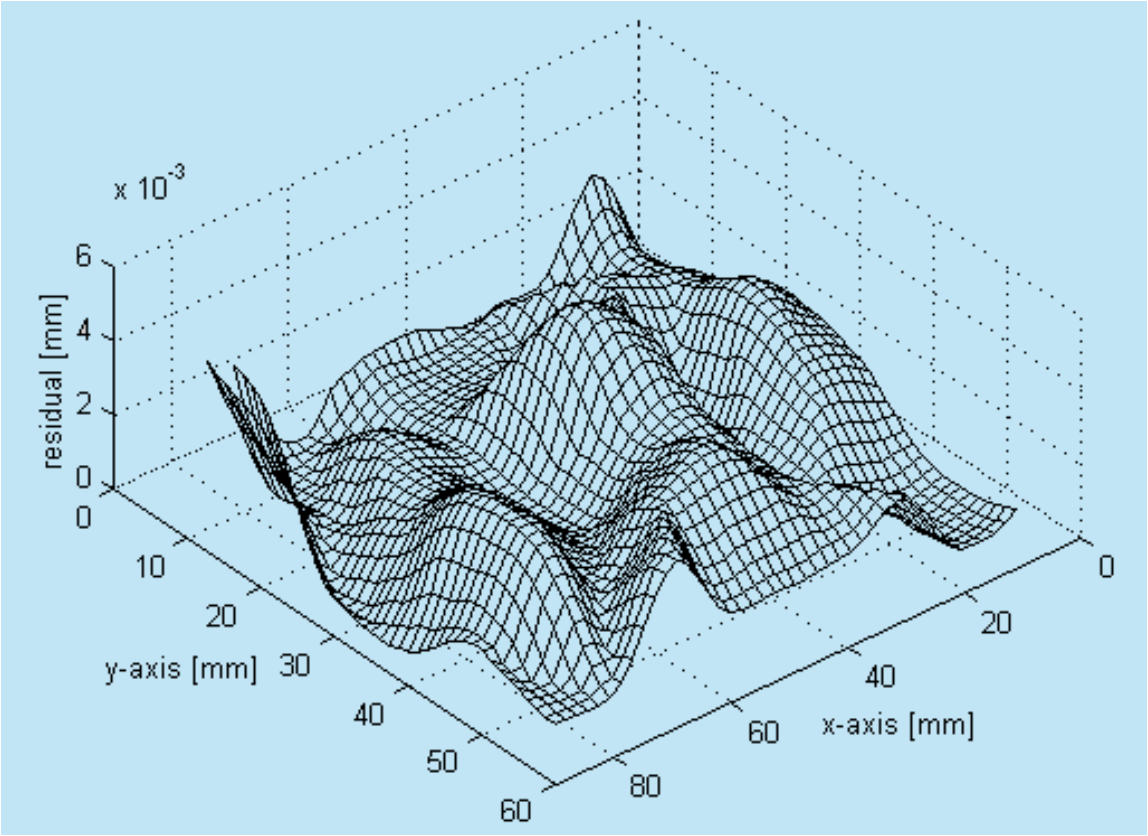


- BP can work with any differentiable non-linear activation function (e.g. sine)
- *ReLU* is common these days especially with deep learning: $f(x) = \text{Max}(0,x)$
  - More efficient computation: Only comparison, addition and multiplication
  - $f'(net)$ is 0 or constant, just fold into learning rate
- Leaky ReLU $f(x) = x$ if $x > 0$, else ax, where $0 \leq a <= 1$, so for net < 0 the derivate is not 0 and can do some learning (does not "die").
  - Lots of other variations
- Not differentiable but we just "cheat" and include the discontinuity point with either side of the linear part of the ReLU function – piecewise linear

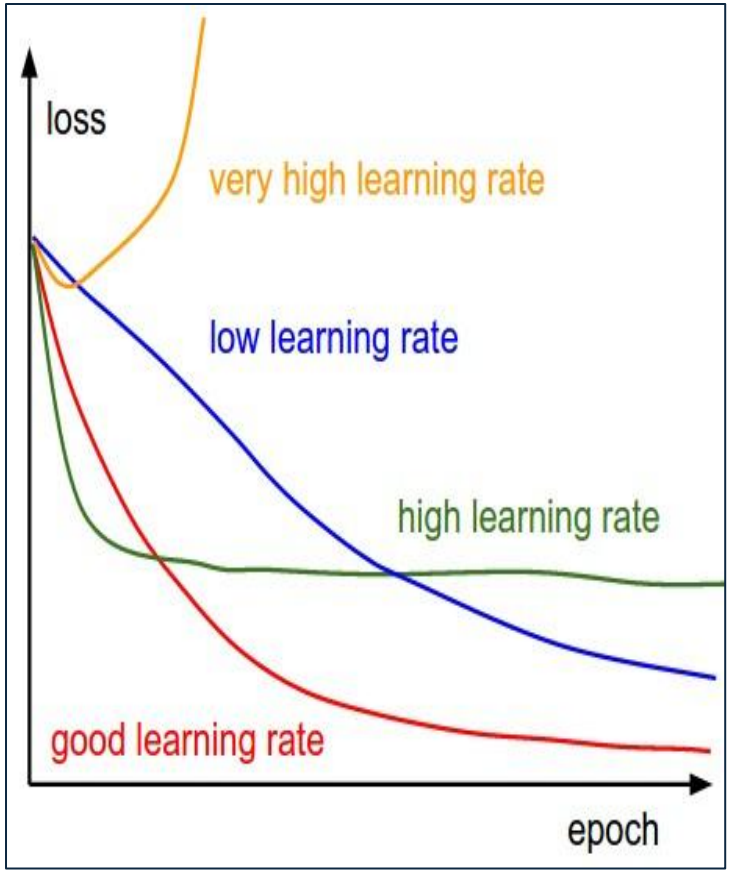
Learning Rate

Learning Rate

- Learning Rate - Relatively small (.01 - .5 common), if too large BP will not converge or be less accurate, if too small it is just slower with no accuracy improvement as it gets even smaller
- Gradient – computed only where you are, avoid too big of jumps



Learning Rate



Learning Rate

- If learning rate is too large can jump around global minimum
- If too small, will get to minimum, but will take a longer time

- Can decrease learning rate over time to give higher speed and still attain the global minimum (although exact minimum is still just for training set and thus…)

- Many algorithms for following the gradient and adjusting the learning rate. (Learning rates often not constant)

Batch vs Stochastic Gradient Descent

Batch vs Stochastic Gradient Descent

- Batch

Batch Update

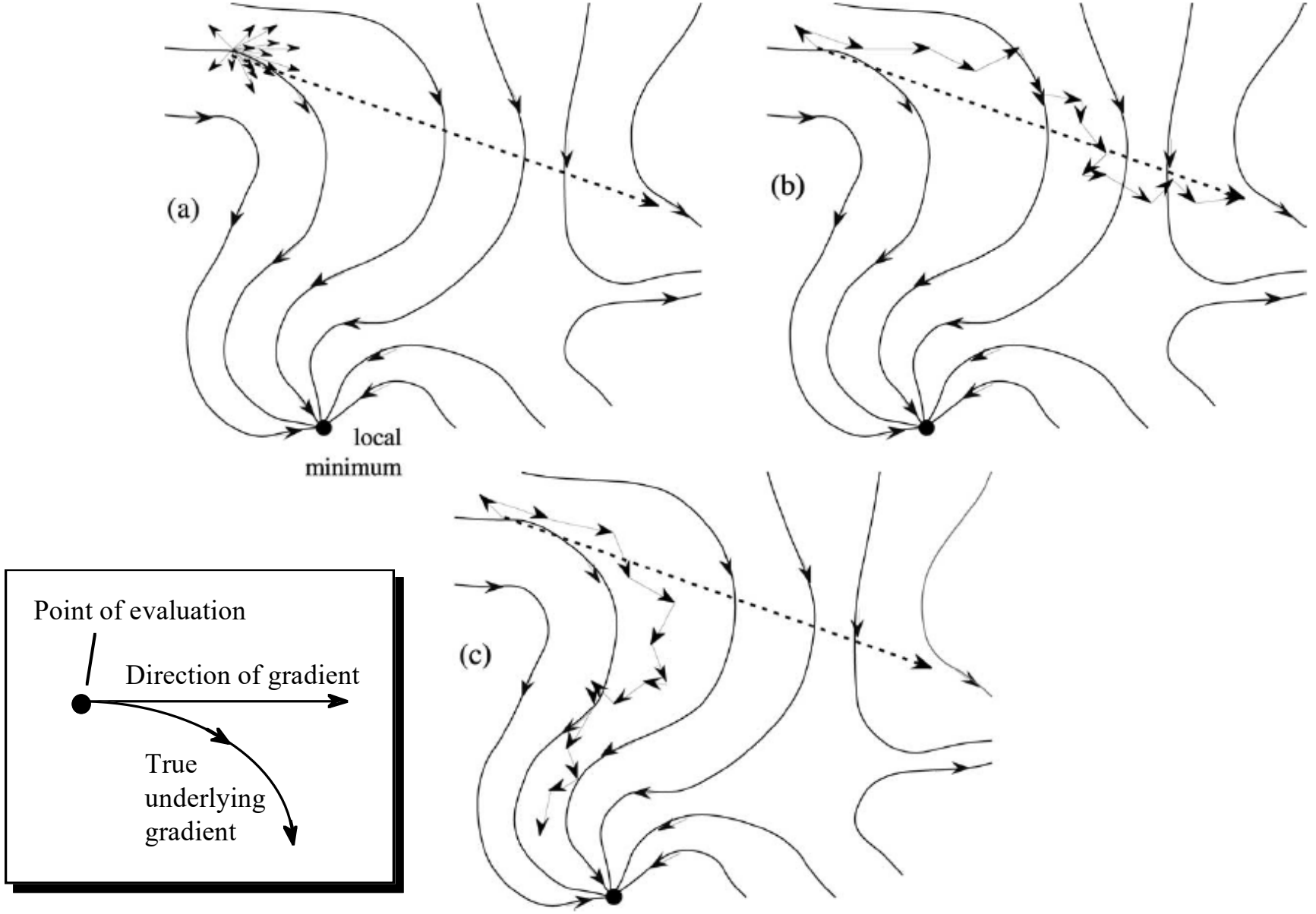
- With On-line (stochastic) update we update weights after every pattern
- With Batch update we accumulate the changes for each weight, but do not update them until the end of each batch
- Batch update gives a better estimated direction of the gradient for the data set, while on-line could do some weight updates in directions quite different from the average gradient of the entire data set
  - Based on noisy instances and also just that specific instances will not usually be at the average gradient
- Proper approach? -
  - Most assumed batch more appropriate, but batch/on-line a non-critical decision with similar results
- We show that batch is less efficient
  - Wilson, D. R. and Martinez, T. R., The General Inefficiency of Batch Training for Gradient Descent Learning, *Neural Networks*, vol. **16**, no. 10, pp. 1429-1452, 2003

Batch vs Stochastic Update

- To get the true gradient, we need to sum errors over the entire training set and only update weights at the end of each epoch

- Batch (gradient) vs stochastic (on-line, incremental)
  - SGD (Stochastic Gradient Descent)
  - With the stochastic gradient descent algorithm, you update after every pattern, just like with the perceptron algorithm (even though that means each change may not be along the true gradient)
  - Stochastic is more efficient and best to use in almost all cases, though not all have figured it out yet
  - We'll talk about this in more detail when we get to Backpropagation

Adaptive Learning


(a)
local
minimum
(b)
(c)
Point of evaluation
Direction of gradient
True
underlying
gradient

Adaptive Learning Rate Approaches

- Momentum is a type of adaptive learning rate mechanism

$$\Delta w_{ij}(t) = C\delta_j z_i + \alpha \Delta w_{ij}(t\text{-}1)$$

- Adaptive Learning rate methods
  - Start LR small
  - As long as weight change is in the same direction, increase a bit (e.g. scalar multiply > 1, etc.)
  - If weight change changes directions (i.e. sign change) reset LR to small, could also backtrack for that step, or …

- Use mini-batch rather than single instance for better gradient estimate – *Sometimes* helpful if SGD variation more sensitive to bad gradient, and also for some parallel (GPU) implementations.

Momentum

- Simple speed-up modification (type of adaptive learning rate)

$$\Delta w_{ij}(t) = C\delta_j z_i + \alpha \Delta w_{ij}(t\text{-}1)$$

- Save $\Delta w_{ij}(t)$ for each weight to be used as next $\Delta w_{ij}(t\text{-}1)$
- A large momentum (e.g. 0.9) will mean that the update is strongly influenced by the previous update, whereas a modest momentum (0.2) will mean very little influence.
- Weight update maintains momentum in the direction it has been going
  - Significant speed-up, common value $\alpha \approx .9$
  - Effectively increases learning rate in areas where the gradient is consistently the same sign. (Which is a common approach in adaptive learning rate methods which we will mention later).
  - Can overshoot
- These types of terms make the algorithm less pure in terms of gradient descent.

Speed up variations of SGD

- Standard Momentum
  - Note that these approaches already do an averaging of gradient, also making mini-batch less critical
- Nesterov Momentum – Calculate point you would go to if using normal momentum. Then, compute gradient at that point. Do normal update using *that* gradient and momentum.
- Rprop – Resilient BP, if gradient sign inverts, decrease it's individual LR, else increase it – common goal is faster in the flats, variants that backtrack a step, etc.
- Adagrad – Scale LRs inversely proportional to sqrt(sum(historical values))
- RMSprop – Adagrad but uses exponentially weighted moving average, older updates basically forgotten
- Adam (Adaptive moments) –Momentum terms on both gradient and squared gradient (uncentered variance) (1^{st} and 2^{nd} moments) – updates based on a moving average of both - Popular

Hyperparameter Selection

Hyperparameter Selection

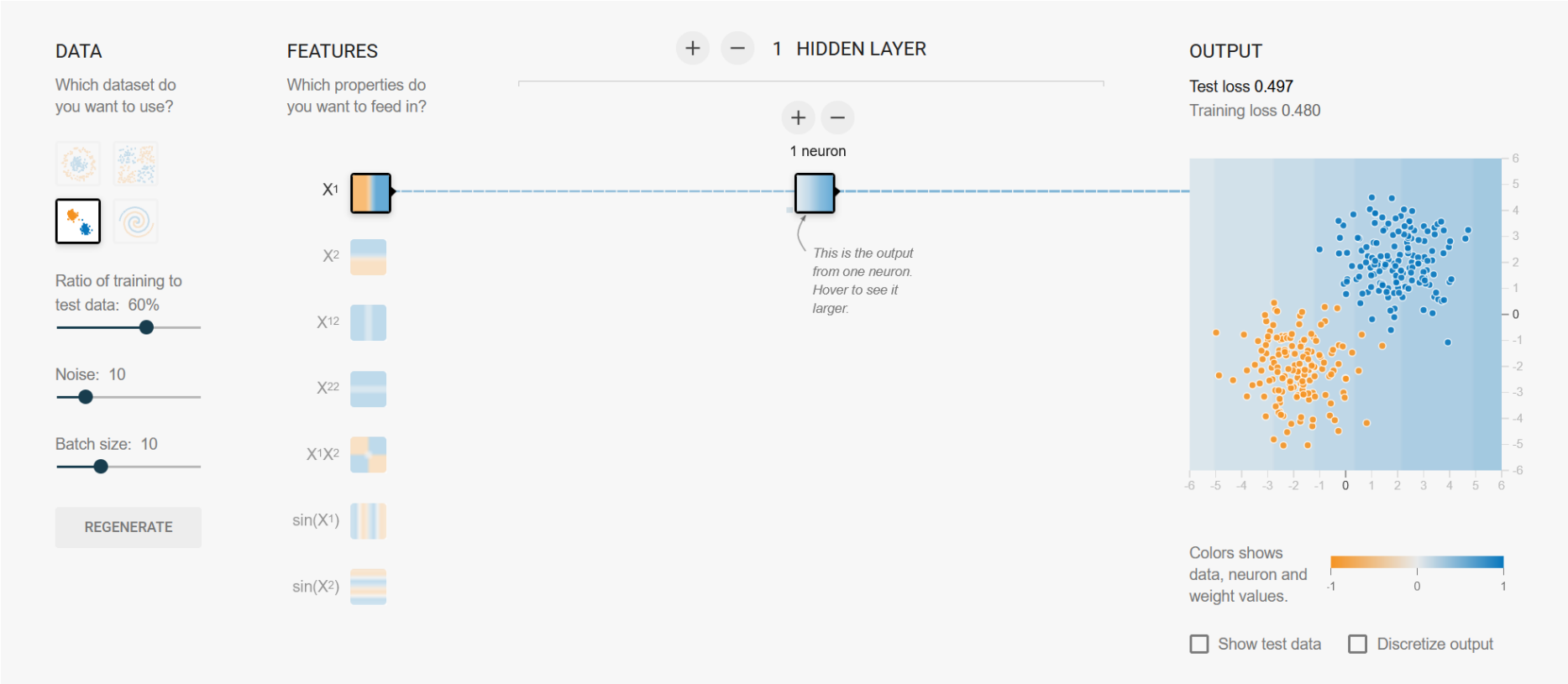
- LR (e.g. .1)
- Momentum – (.5 … .99)
- Connectivity: fully connected between layers
- Number of hidden nodes: Problem dependent
  - Less hidden nodes NOT a great approach because may underfit
- Number of layers: 1 (common) or 2 hidden layers which are usually sufficient for good results, attenuation makes learning very slow – modern deep learning approaches show significant improvement using many layers and many hidden nodes
- Manual CV can be used to set hyperparameters: trial and error runs
  - Often sequential: find one hyperparameter value with others held constant, freeze it, find next hyperparameter, etc.
- Hyperparameters could be learned by the learning algorithm in which case you must take care to not overfit the training data – always use a cross-validation technique to measure hyperparameters

Neural Net Activity

https://playground.tensorflow.org/

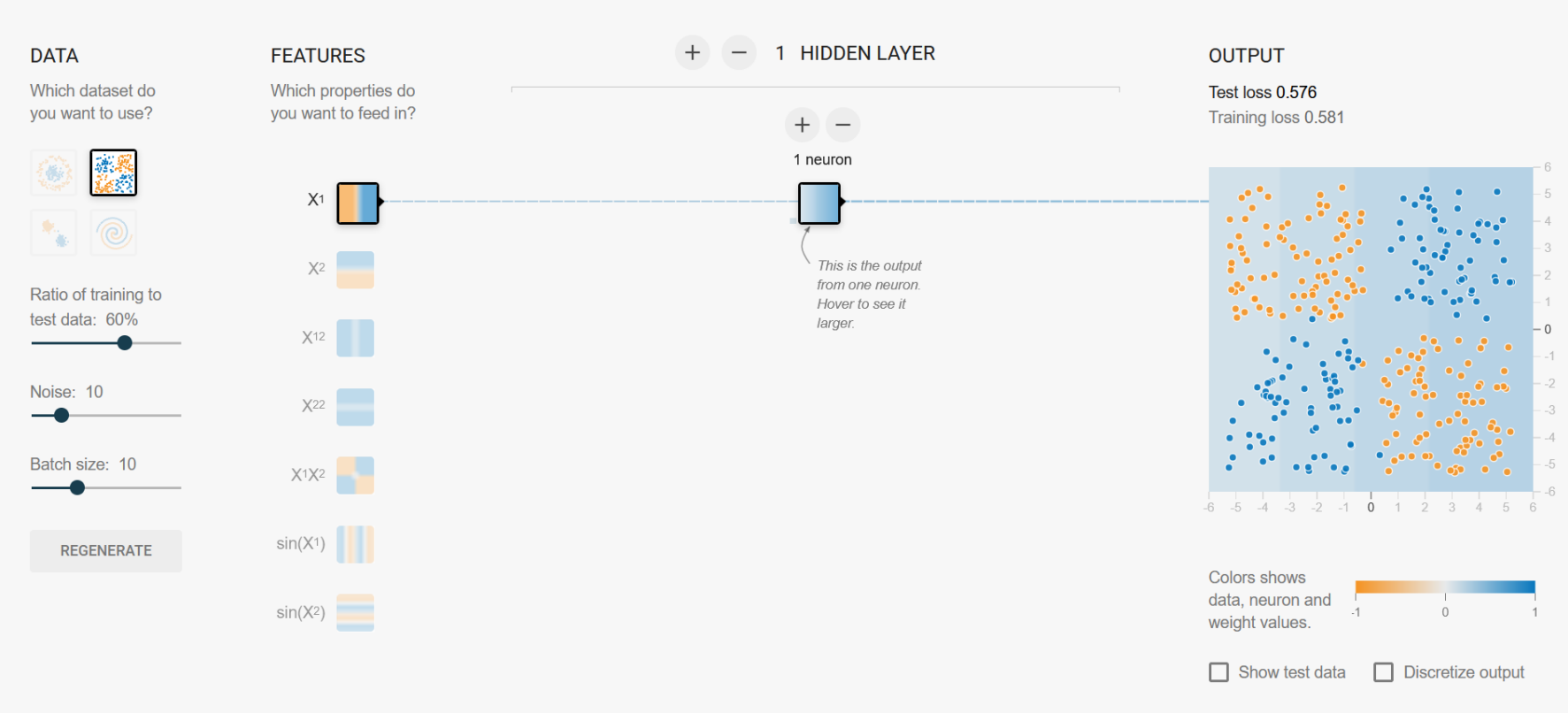
Challenge 1 - Basically a Perceptron

- Can you separate the cluster data with one single hidden neuron?
- How many inputs do you need?



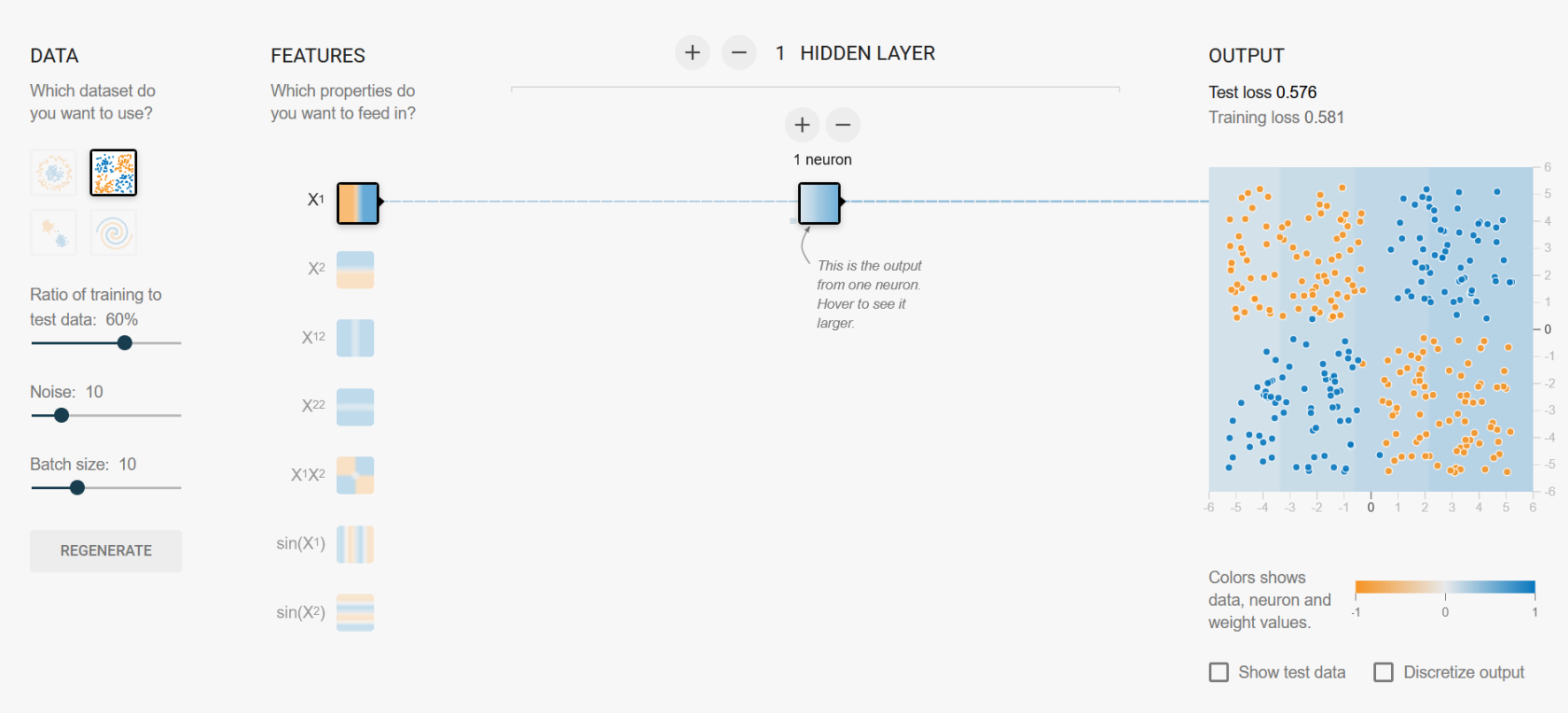
Challenge 2- Basically a Perceptron

- Can you separate the checker data with one single hidden neuron?
- Try noisier data.



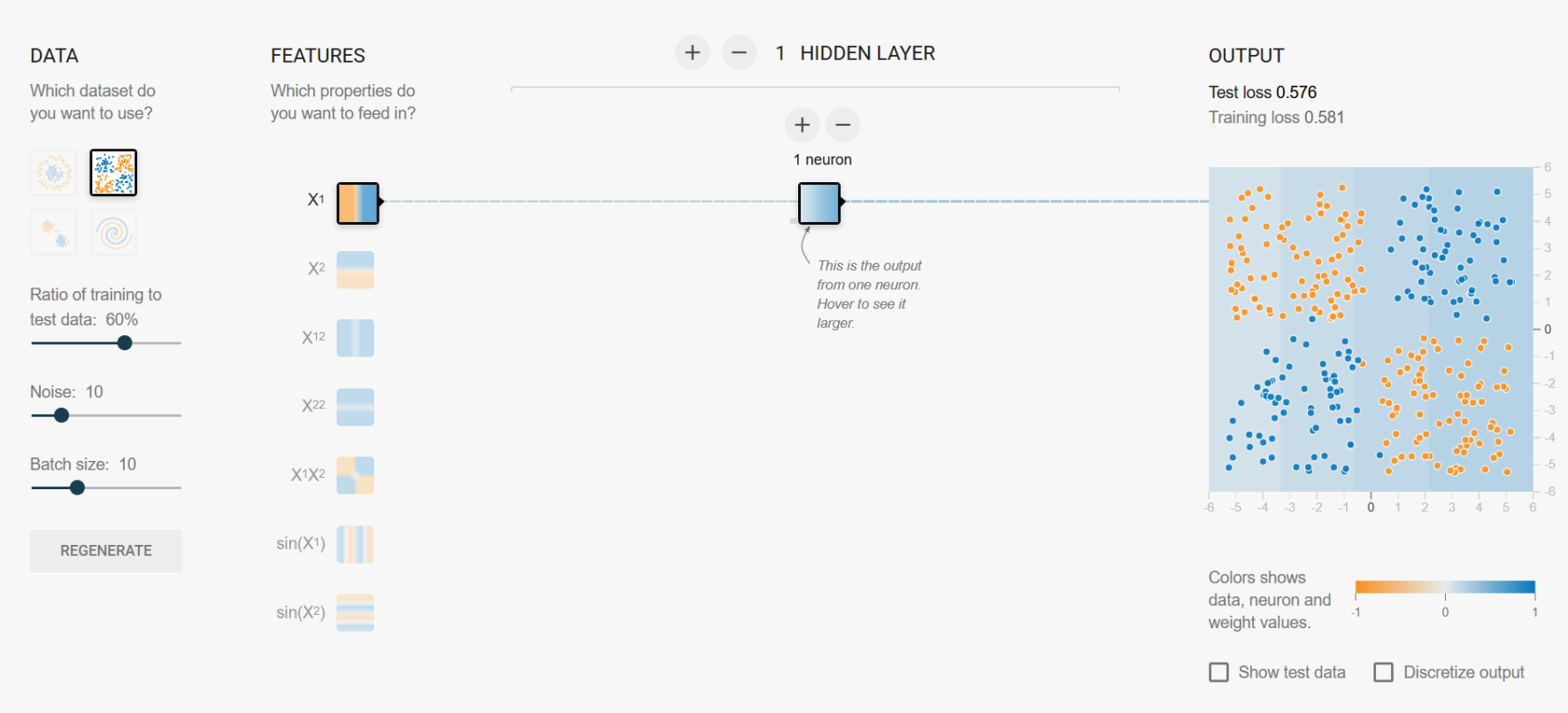
Challenge 3

- Can you separate the checker data with only X1 and X2 as inputs?



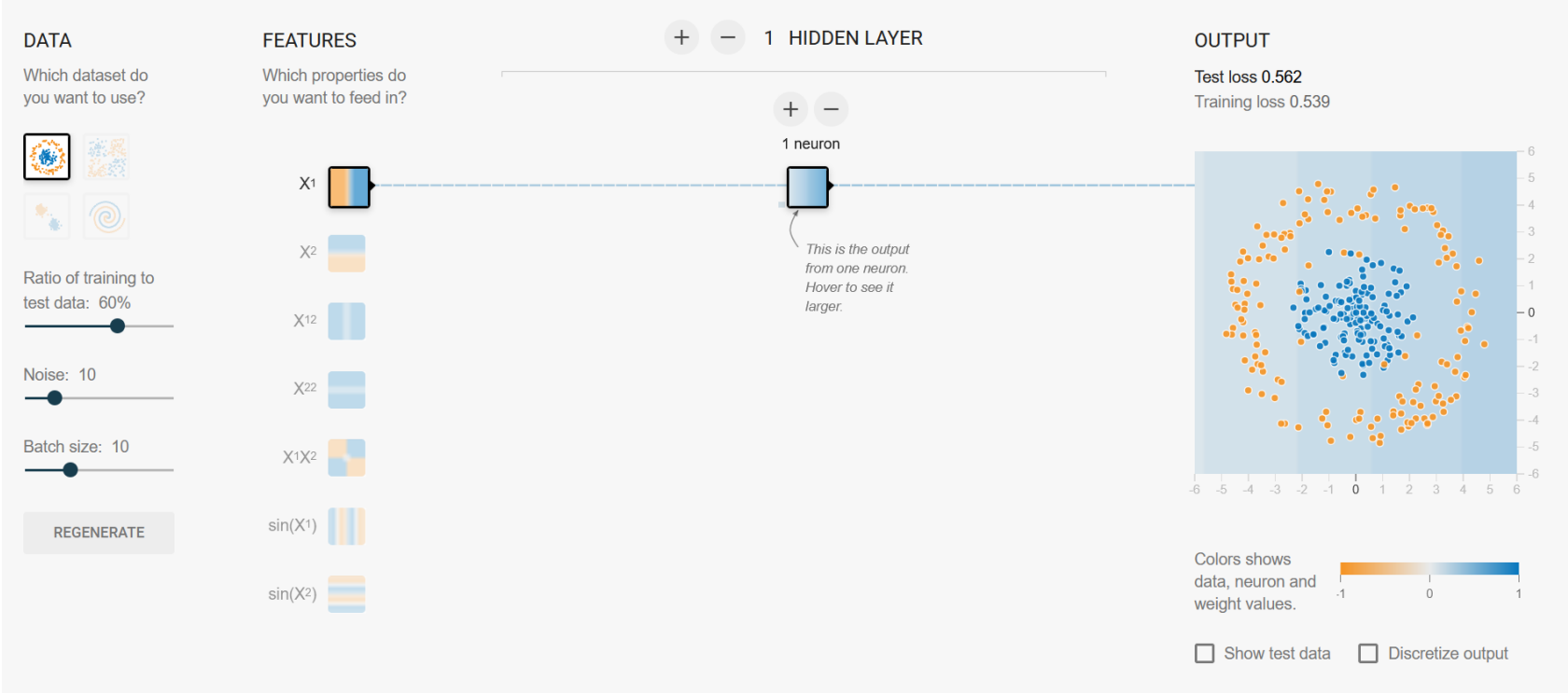
Challenge 3.5

- Try multiple activation functions



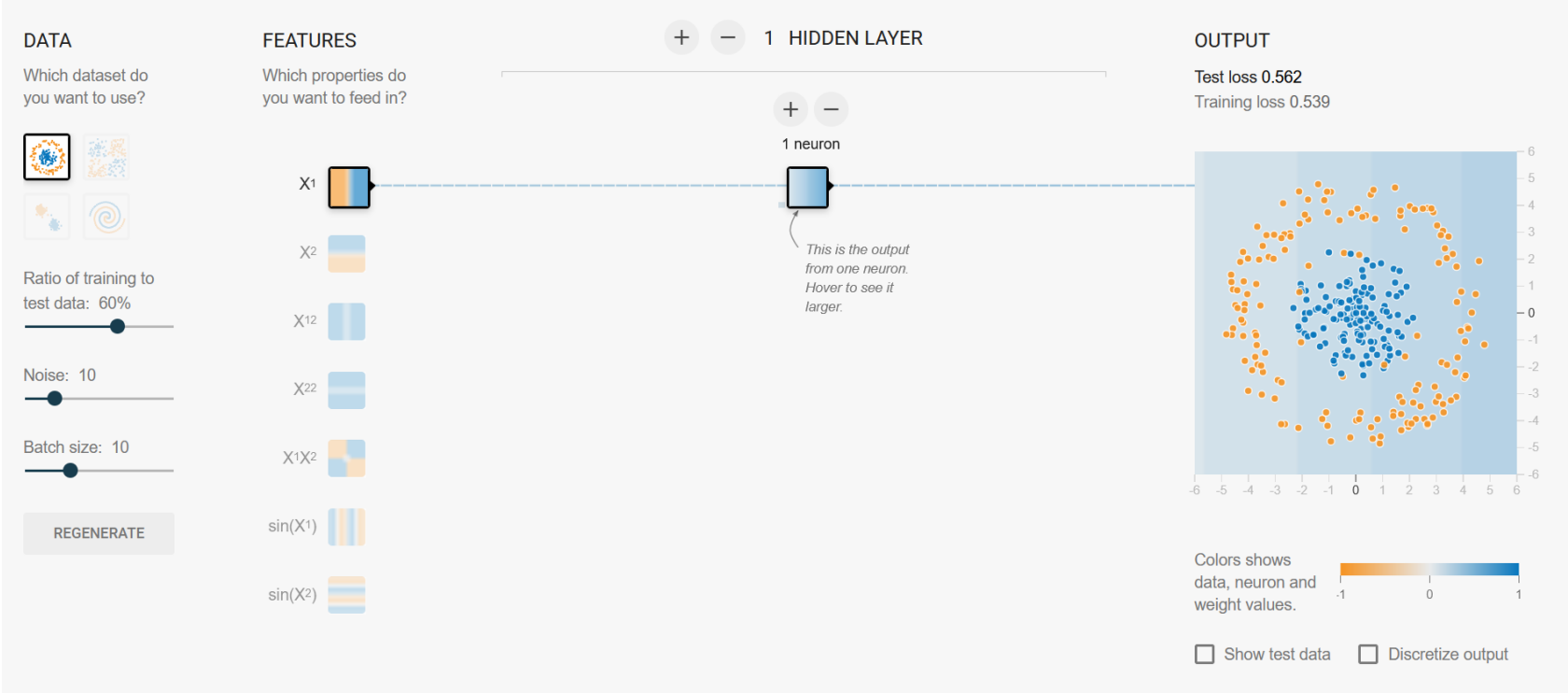
Challenge 4

- Can you separate the circle data with a single layer?
- Which inputs did you use?



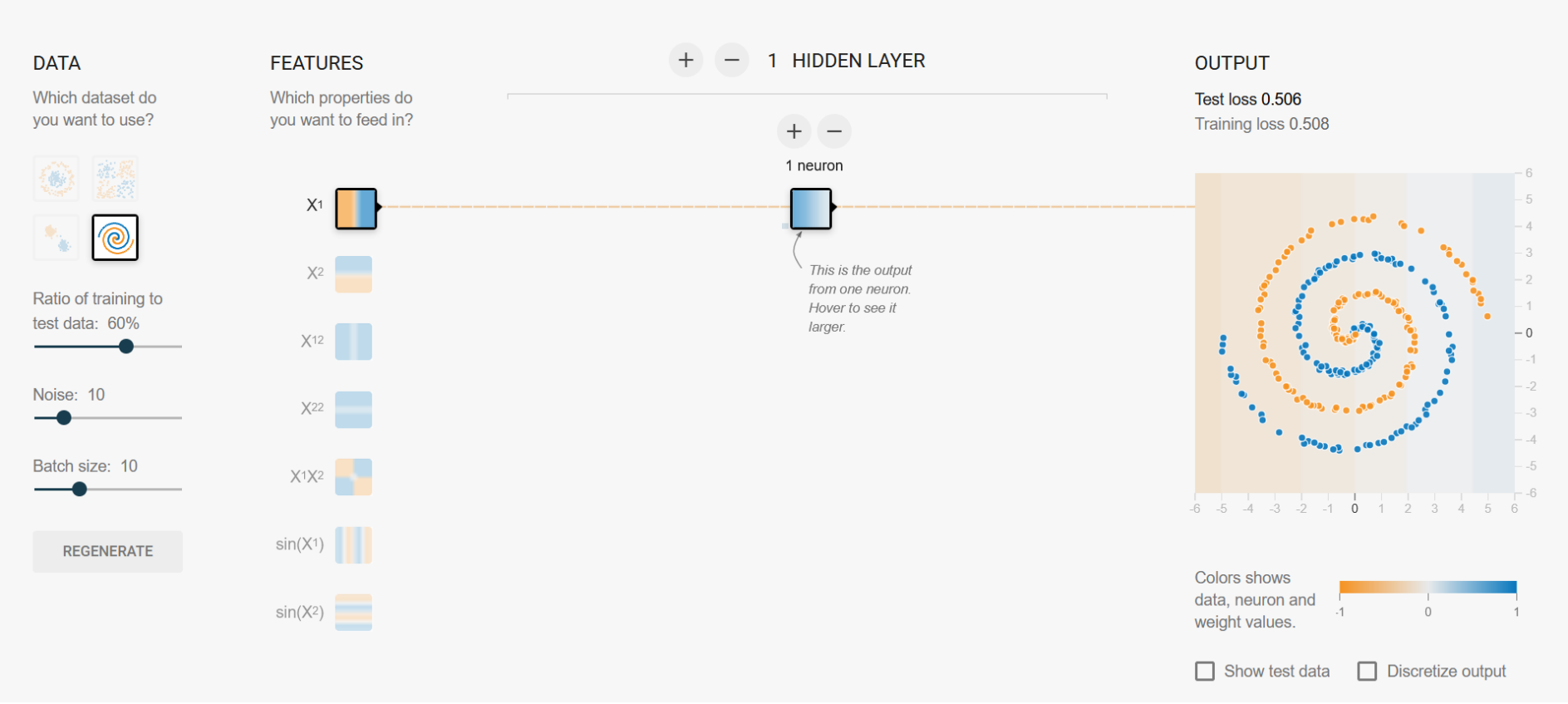
Challenge 4.5

- Can you separate the circle data with the minimum number of hidden nodes? (any number of layers acceptable)



Challenge 5 – Win a prize

- Can you separate the spiral data?
- Noise = 20
- Training/test ratio = 70



Quiz Time!