

Lecture 1

Feed Forward Neural Network Design

DSCI-789: Neural Networks for Data Science

Travis Desell (tjdvse@rit.edu)
Associate Professor
Department of Software Engineering



ROCHESTER INSTITUTE OF TECHNOLOGY

Overview

- What is a Neural Network?
- Layers in a Neural Network
 - Input
 - Hidden
 - Output
- Weights and Biases
- Activation Functions
- The Forward Pass
- The "Problem"

What is a Neural Network?

What are Neural Networks?

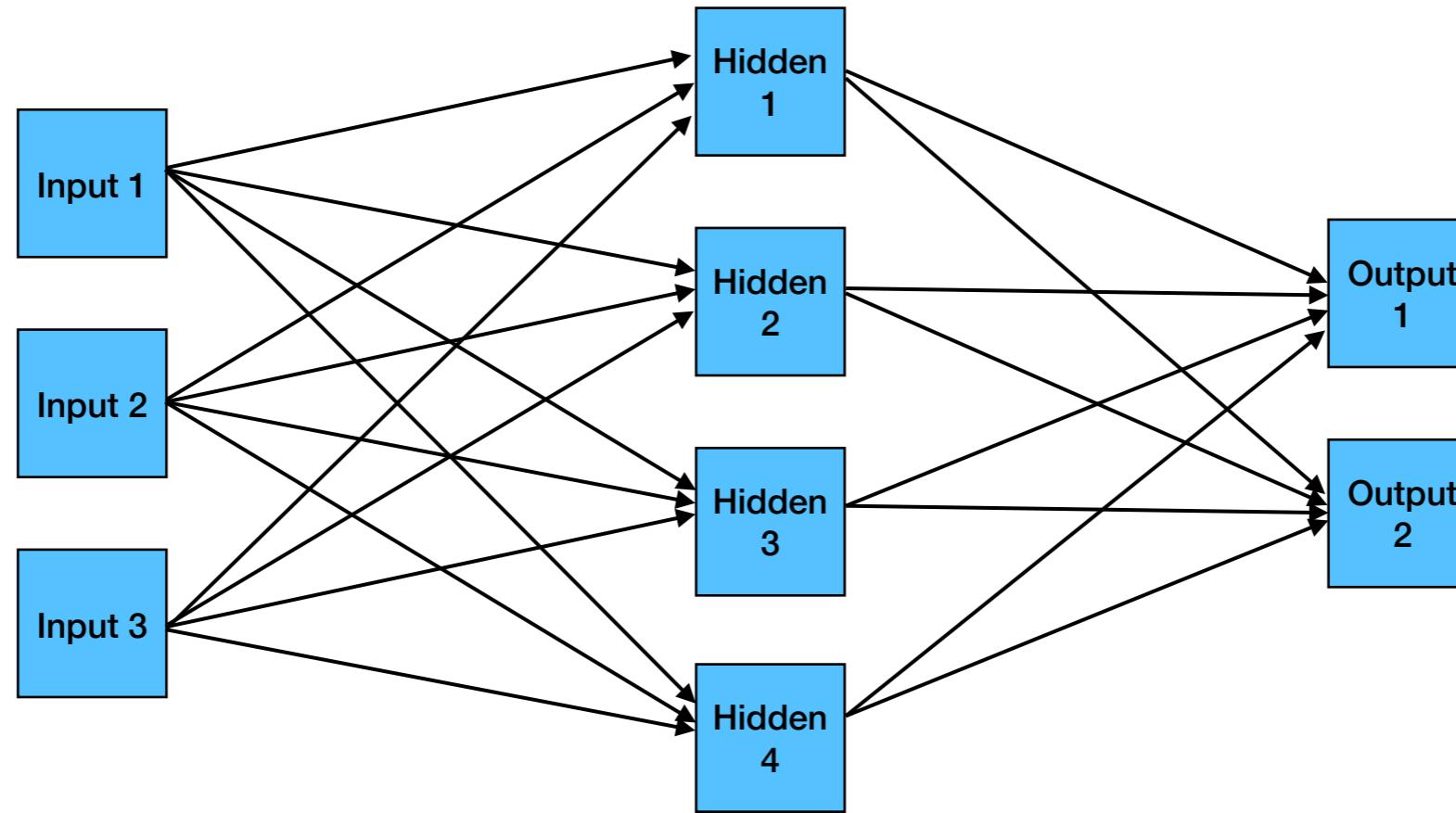
- Neural networks take given inputs and calculate an output or set of outputs.
- Neural networks are typically used for supervised learning, e.g:
 - Classification - given an input, predict a class/type.
 - Prediction - given an input, predict an expected output.

What are Neural Networks?

- Neural Networks (NNs) are directed graphs:
 1. Values at input nodes (vertices) are set by input data.
 2. Data flows through the edges of the network through hidden nodes.
 3. Finally, data reaches the output nodes which are the predictions of the neural network.
- The nodes/vertices of a NN are typically (but not always) organized into input, hidden and output layers.

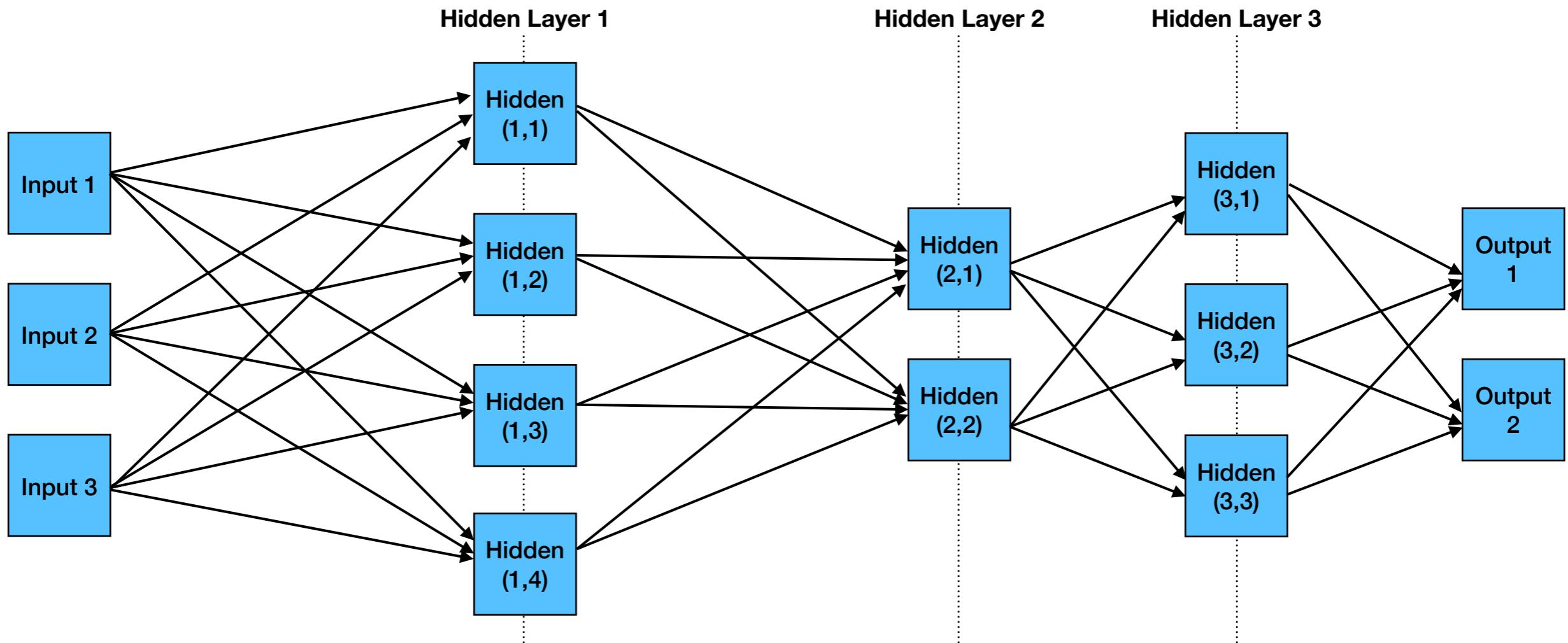
Layers in a Neural Network

Layers in a Neural Network



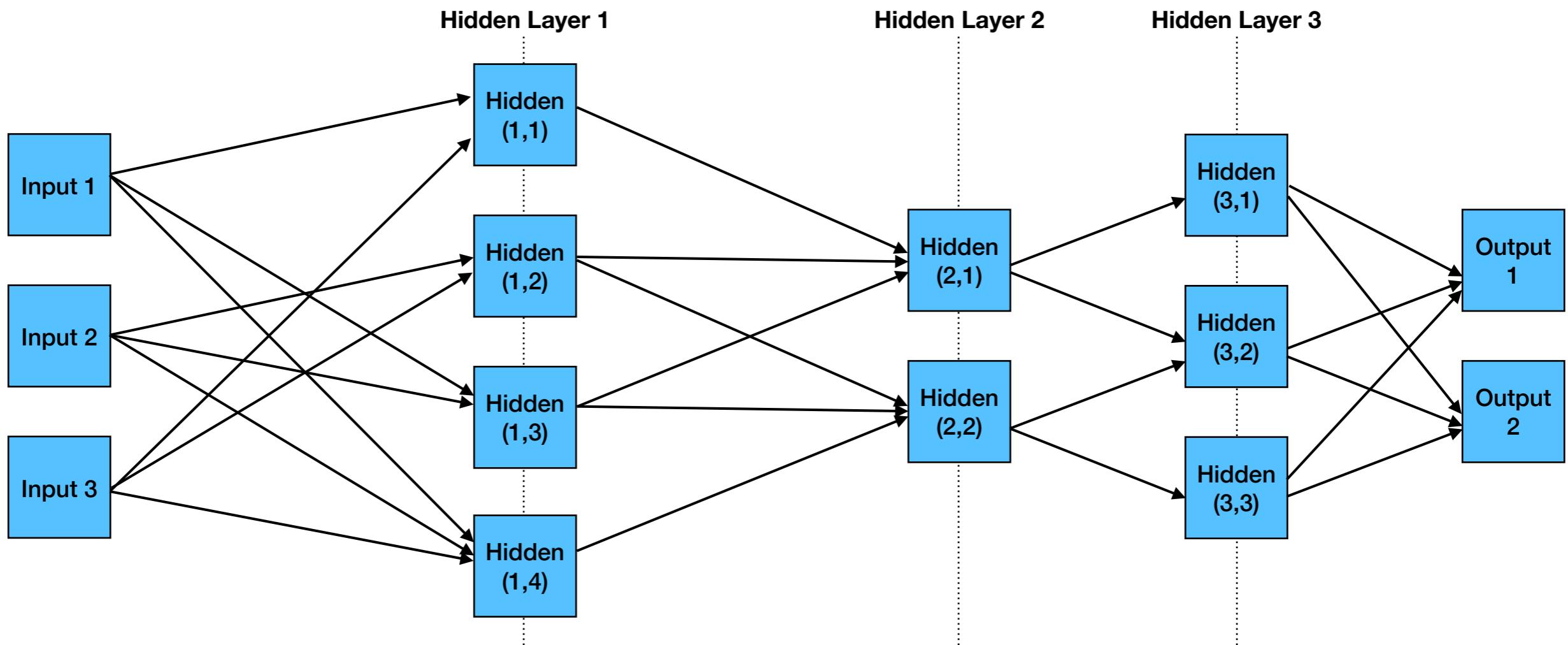
- The above is a feed forward neural network with 3 input nodes, 4 hidden nodes and 2 output nodes.
- The number of input and output nodes is determined by your data.
- *You determine the number of hidden nodes.*

Layers in a Neural Network



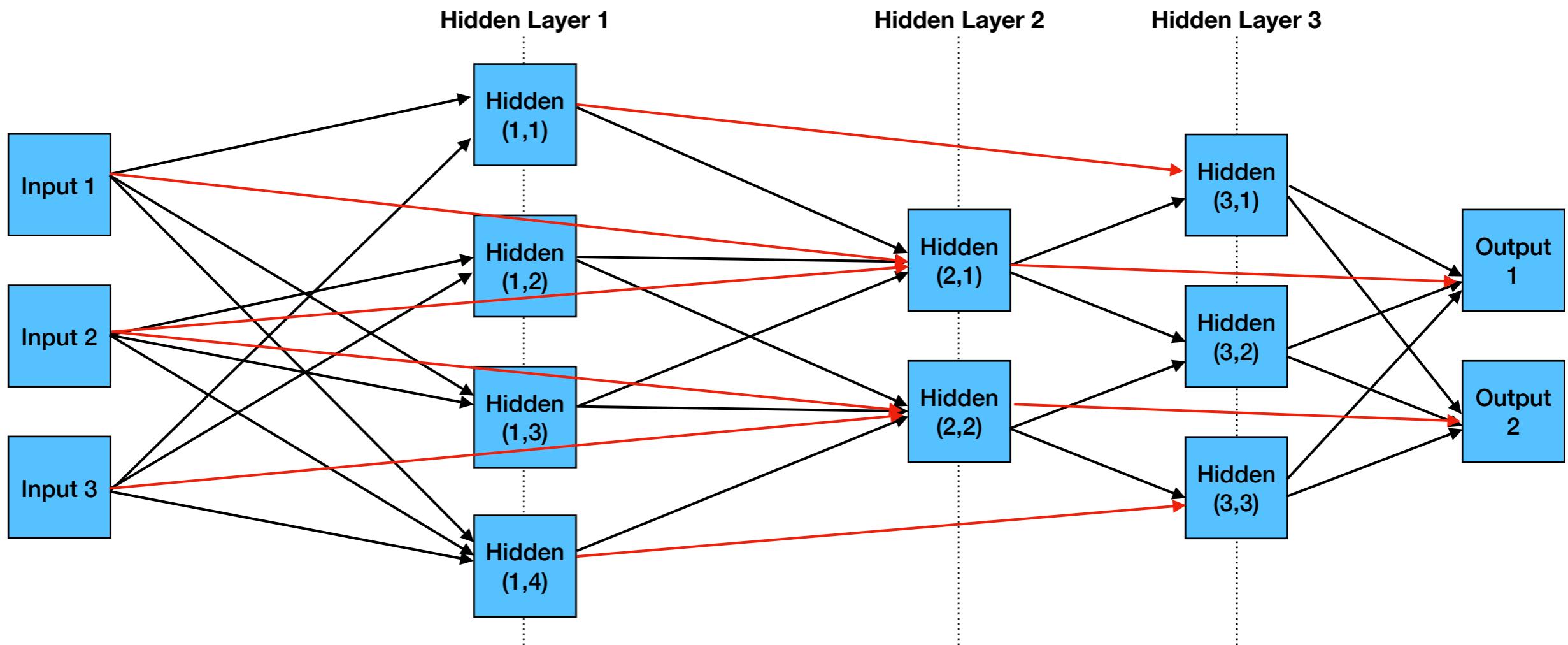
- NNs may also have any number of layers.
- Layers do not necessarily need to have the same number of nodes.
- *You determine the number of layers and the number of nodes in each.*

Layers in a Neural Network



- The previous two networks were *fully connected*. Each node in a layer is connected by an each to each node in the next layer.
- Networks are not necessarily fully connected (see above). *You determine how the nodes are connected.*

Layers in a Neural Network



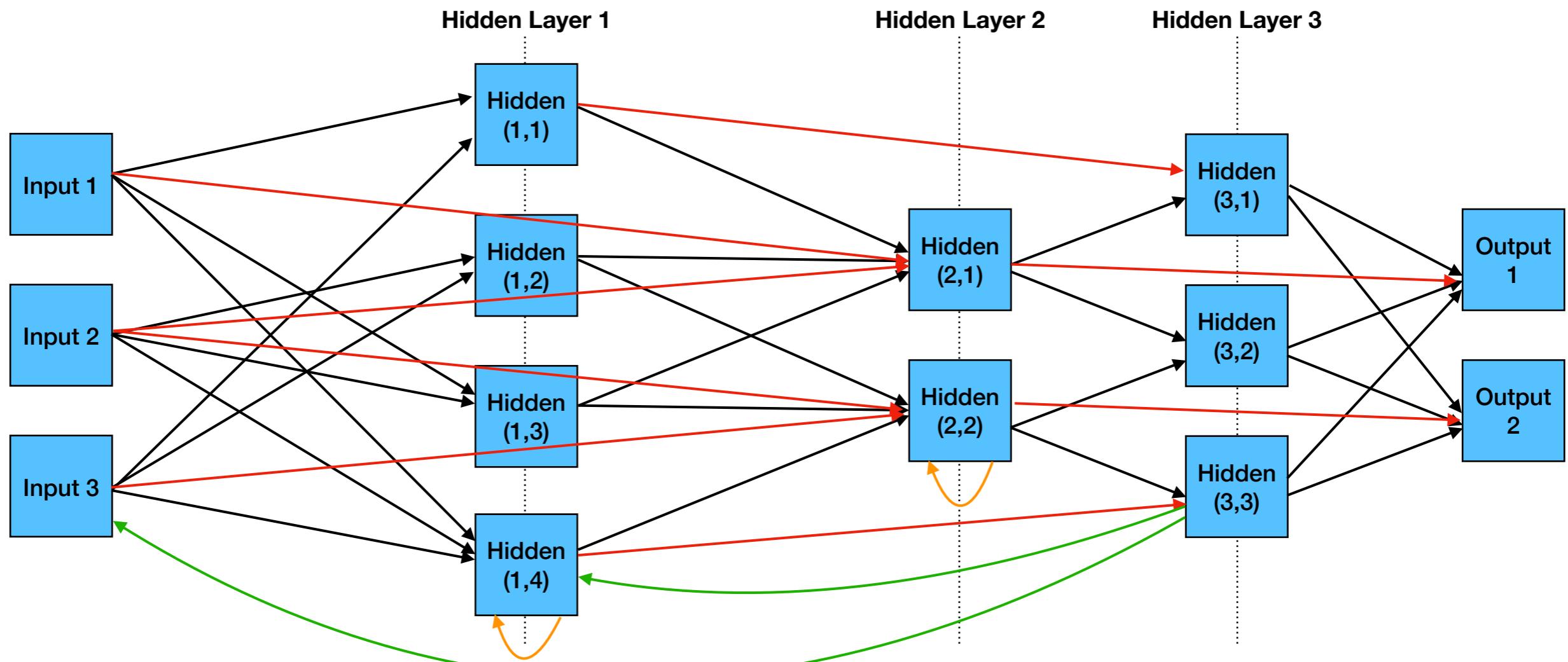
- Edges do not necessarily need to be between sequential layers.
- Edges can span layers (see edges in red). This has gained popularity with *residual networks* such as ResNet [1] and its variants [2,3].
- *You determine how the nodes are connected.*

[1] He, Kaiming, et al. "Deep residual learning for image recognition." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.

[2] https://en.wikipedia.org/wiki/Residual_neural_network

[3] <https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035>

Layers in a Neural Network



- Feed forward neural networks are acyclic (there are no backwards connections which would make a cycle).
- Recurrent neural networks, however, may have backward connections (green) or even looping connections (orange) - but don't worry about those yet.

What Architecture Should I Use?

- It currently is not possible to *a priori* know how well an architecture will perform.
- Just because an architecture works well on one data set also does not mean it will necessarily work well on another. In fact there is a whole sub-field of neural network study called *transfer learning* [4], which focuses on repurposing NNs trained on one data set to another.
- This is what makes machine learning and neural networks just as much of an art as a science.
- Also why being an ML expert/data scientist means decent job security for awhile.

[4] https://en.wikipedia.org/wiki/Transfer_learning

What Architecture Should I Use?

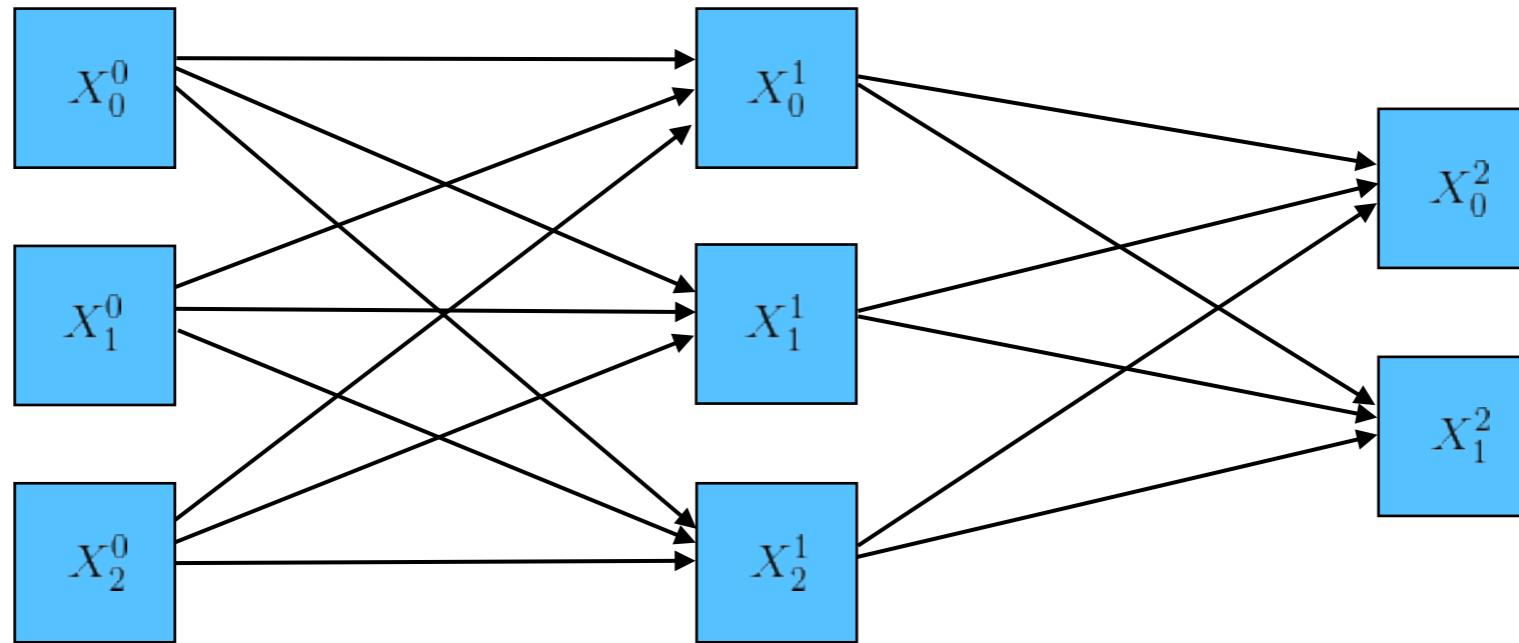
- Don't get too comfortable!
- *Neural Architecture Search* (NAS) is a growing area of research which seeks to automate the design of neural networks. Approaches include neuro-evolution, reinforcement learning, and even neural networks to design neural networks.
- However, NAS is still computationally expensive (it can involve training thousands or millions of neural networks) and can require large scale compute resources.
- Some approaches are being studied to speed the process which estimate the potential performance of a NN without training it.

Tips for NN Design

- Be scientific:
 - Change one thing at a time.
 - Training a NN is stochastic (randomized) so you need to train multiple times with new settings to determine if they help or not.
 - To be rigorous you can use statistical tests for significance.
- Larger neural networks are harder and more expensive to train.
- Smaller networks may not be complex enough.
- Adding layers helps with learning, however more layers greatly increases training difficulty:
 - For a long time it was thought that more than 2 layers were not possible to train due to vanishing gradients however advances brought on the age of *deep learning* allowing deeper networks
- Over time you will gain experience in understanding how different changes will effect your neural network's performance.

Weights and Biases

Weights and Biases

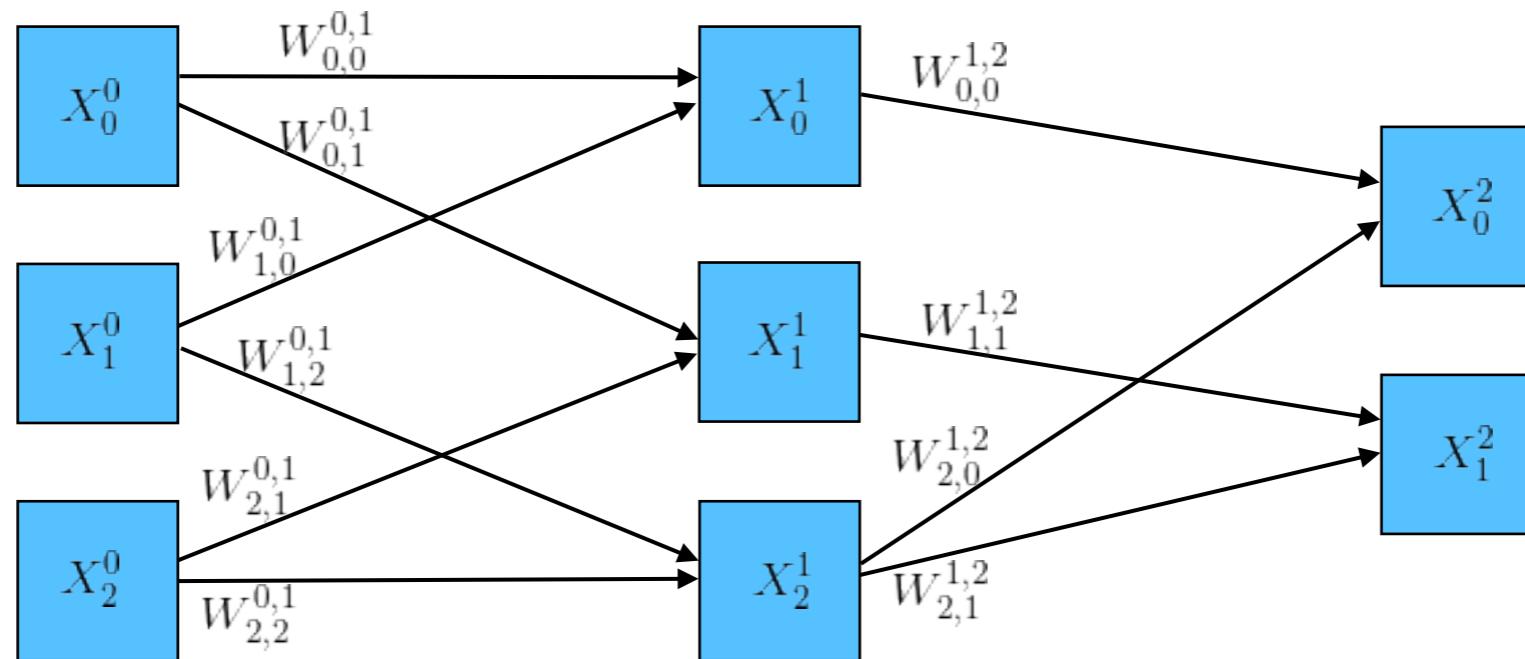


- We can define nodes as X with a little math notation. The super script is the layer number and the subscript is the node number in the layer:

$$X_{\text{number}}^{\text{layer}}$$

- Each node in layer 0 will be initialized from the input data, and the values of the subsequent nodes will be calculated using *weights*, values belonging to each edge.
- For now, each node will hold one value.

Weights and Biases



- We can define edges and weights with some more notation. Where the superscript gives the input layer and output layer, and the subscript gives the input node number and output node number:

$$W_{\text{input node}, \text{output node}}^{\text{input layer}, \text{output layer}}$$

- For now, each edge will hold one weight value.
- Many representations leave out the output layer as they assume there are no layer skipping connections.

Weights and Biases

- We can then formulate how the values of each node (after the input nodes have been assigned) as follows:

$$X_j^{i+1} = \sum_{k=0}^n X_k^i * W_{k,j}^{i+1}$$

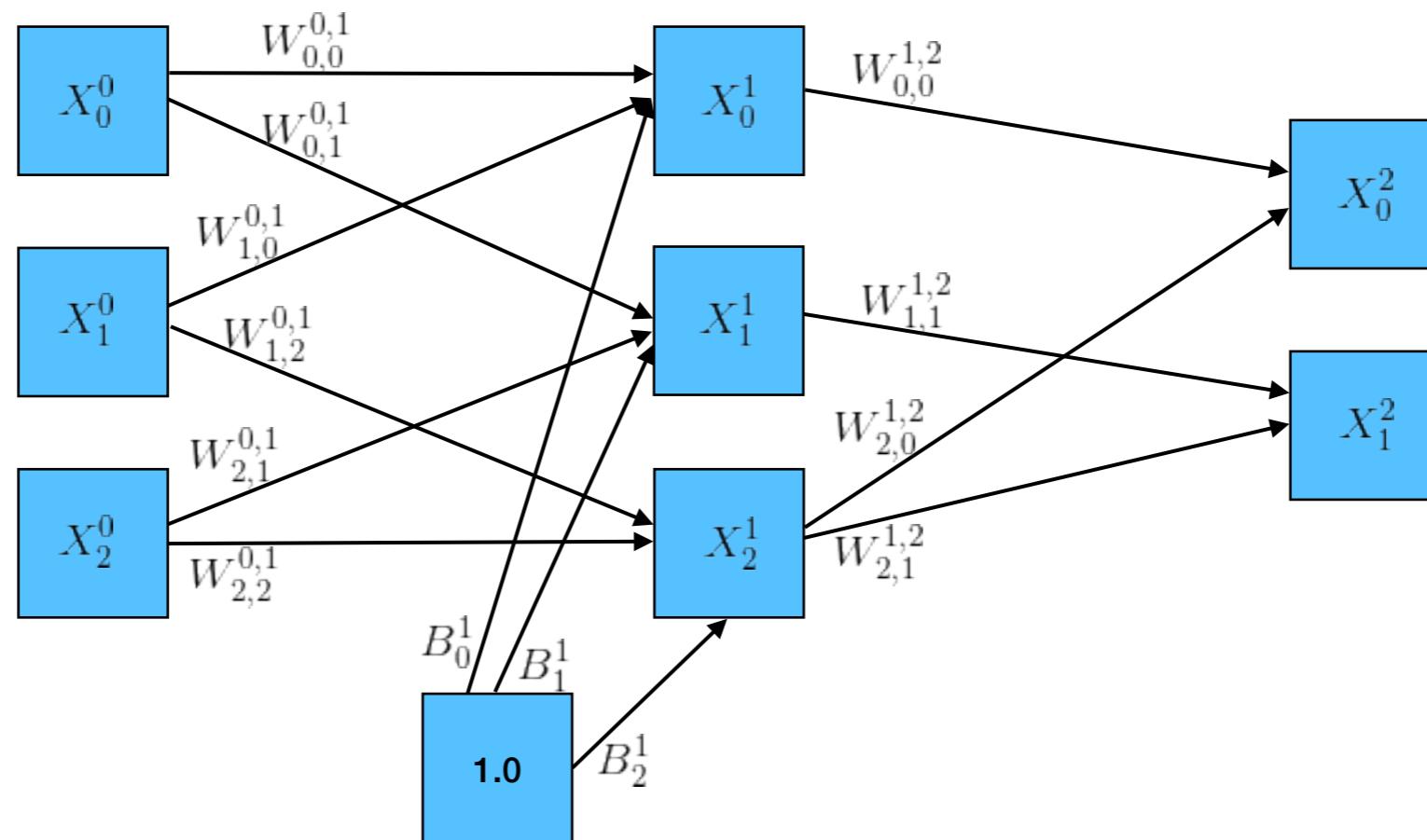
- Where i is the input layer, j is the output layer node number, and k goes from 0 to n , where n is the number of input nodes.
- Note that this assumes that there are no layer skipping edges and also that the network is fully connected.
- In some representations you will see this presented as a vector multiplication, where each layer is a vector, and each node in that vector has a vector of weights. It also can be further abstracted to matrix multiplication if we are calculating multiple inputs at the same time (more on that later).

Weights and Biases

$$X_j^{i+1} = \sum_{k=0}^n X_n^i * W_{k,j}^{i,i+1}$$

- In short, the value for each hidden and output node is calculated as the sum of the value of each node connected to it on previous layers multiplied by the weight on the edge from that node to the target node.
- We are not quite done yet though, as nodes typically have *bias* and apply an *activation function*.

Weights and Biases



- We also typically add a bias to each hidden node. This is similar to adding a weight to a node which has a fixed value of 1.0.
- Bias can be represented as the following, where the superscript is the layer and the subscript is the node number:

$$B_{number}^{layer}$$

- Bias is not added to the input or output nodes.

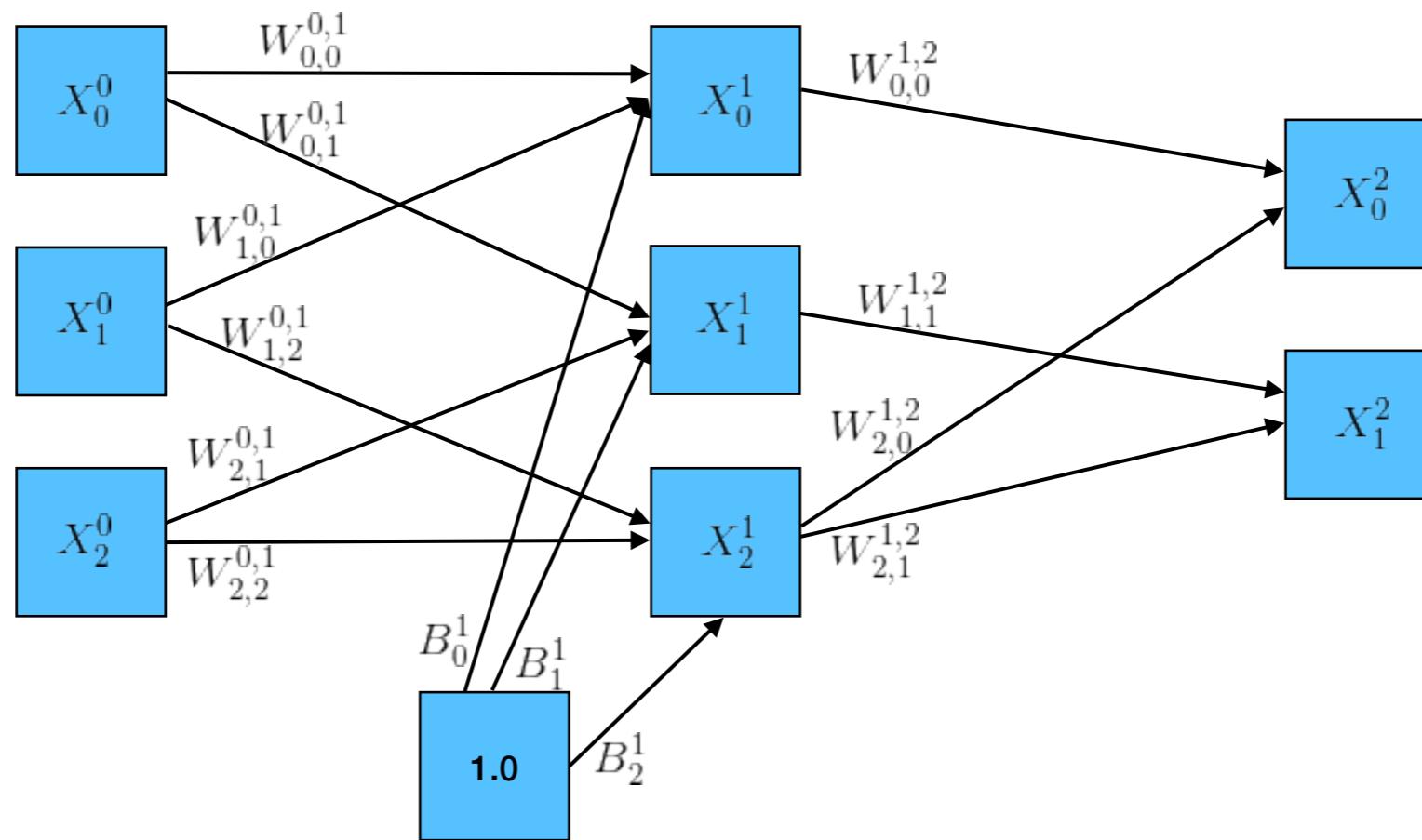
Weights and Biases

- We update the calculation for each node's value by adding its bias:

$$X_j^{i+1} = B_j^{i+1} + \sum_{k=0}^n X_n^i * W_{k,j}^{i,i+1}$$

- Bias is often removed from neural network diagrams as it is assumed each hidden node has one.
- After this we need to apply *activation functions* to each node.

Weights and Biases



- To recap, we can calculate the hidden and output node values for the above neural network (assuming no activation function) as follows:

$$X_0^1 = (1.0 * B_0^1) + (X_0^0 * W_{0,0}^{0,1}) + (X_1^0 * W_{1,0}^{0,1})$$

$$X_1^1 = (1.0 * B_1^1) + (X_0^0 * W_{0,1}^{0,1}) + (X_2^0 * W_{2,1}^{0,1})$$

$$X_2^1 = (1.0 * B_2^1) + (X_1^0 * W_{1,2}^{0,1}) + (X_2^0 * W_{2,2}^{0,1})$$

$$X_0^2 = (X_0^1 * W_{0,0}^{1,2}) + (X_2^1 * W_{2,0}^{1,2})$$

$$X_1^2 = (X_0^1 * W_{1,1}^{1,2}) + (X_2^1 * W_{2,1}^{1,2})$$

Activation Functions

Activation Functions

- After the sum of the bias and products of the inputs times their weights has been calculated, an activation function is typically applied to provide the final value for a node.
- Activation functions are typically denoted $g(x)$.
- Common activation functions are:

- identity (i.e., do nothing):

$$g(x) = x$$

- ReLU (rectified linear unit):

$$g(x) = \max(0, x)$$

- sigmoid:

$$g(x) = \frac{1}{1 + e^{-x}}$$

- ReLU-6:

$$g(x) = \min(\max(0, x), 6)$$

- tanh:

$$g(x) = \tanh(x)$$

- leaky ReLU:

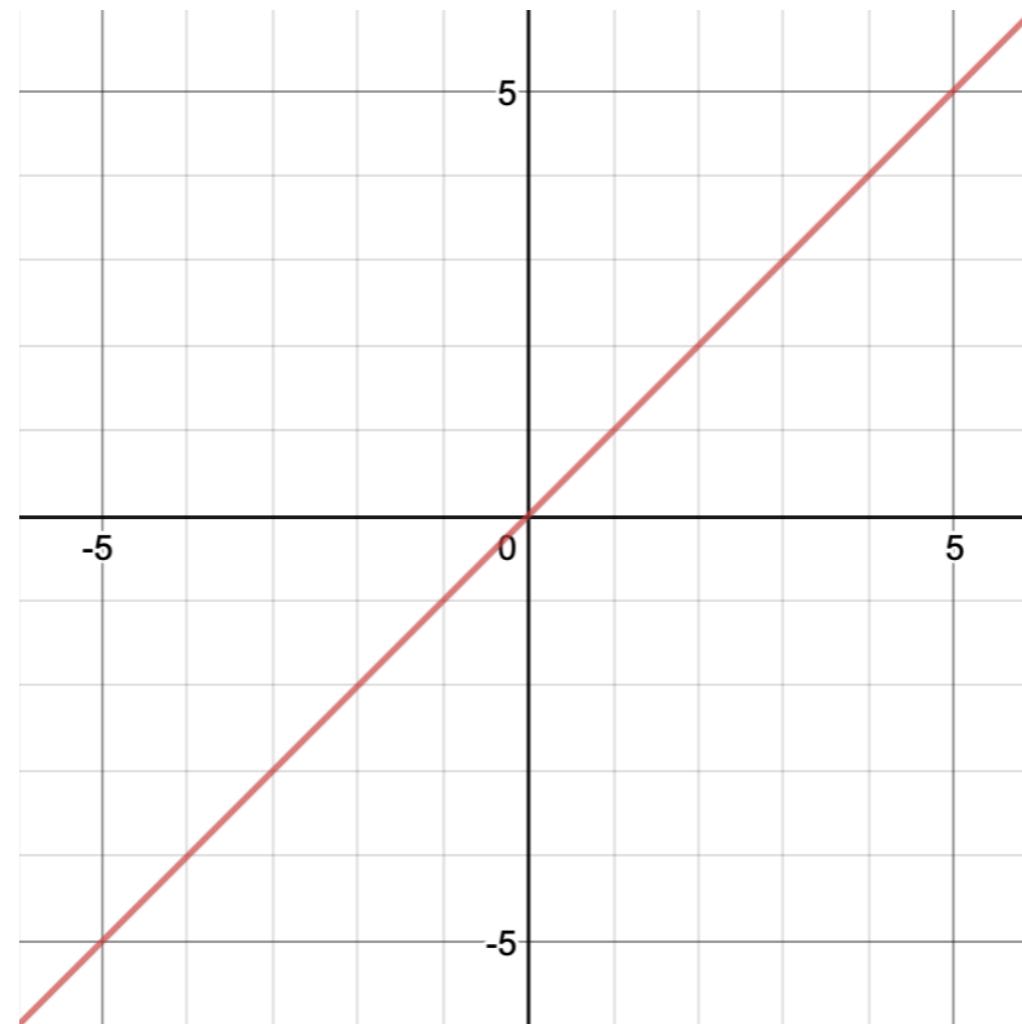
$$f(x) = \begin{cases} 0.01x, & \text{if } x \leq 0 \\ x, & \text{otherwise} \end{cases}$$

- Leaky ReLU and ReLU-6 can be parameterized (i.e., values other than 0.01 and 6) as well as combined.

Identity Function

$$g(x) = x$$

- Pros:
 - Computationally minimal.
 - Has a derivative of 1, keeps error signal strong in backpropagation.
- Cons:
 - Unbounded

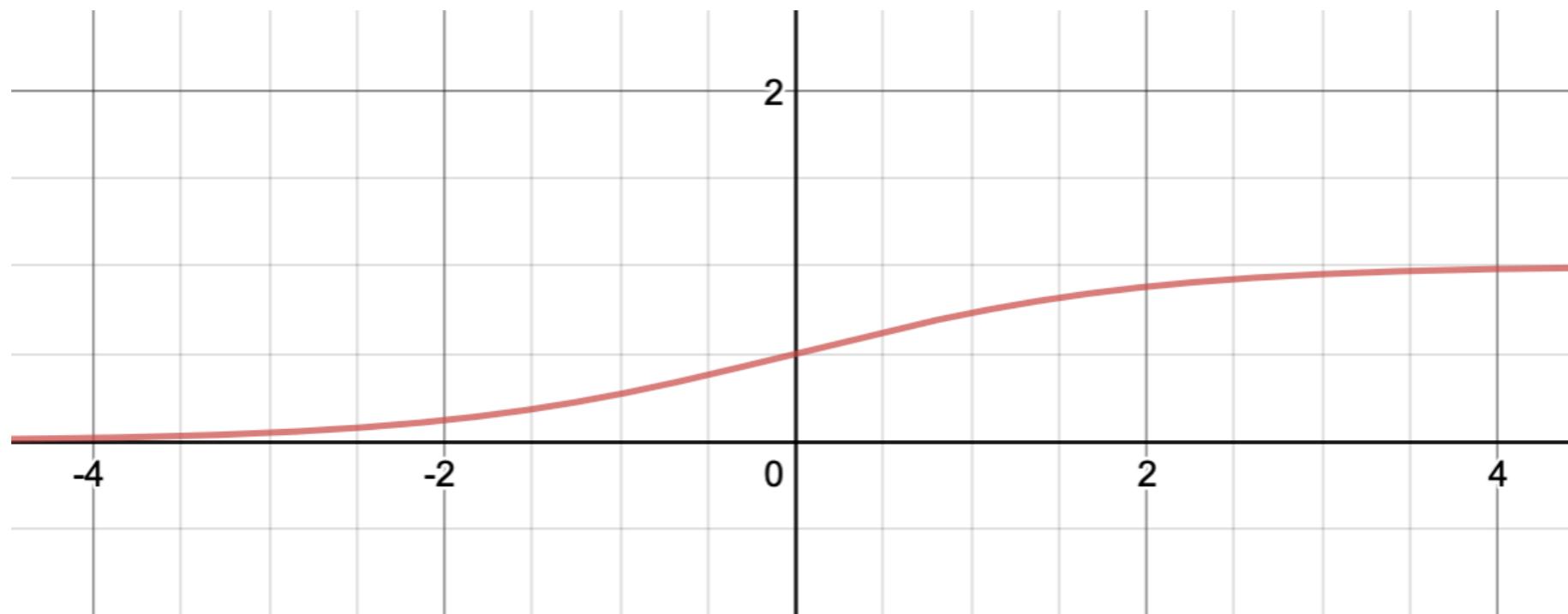


Sigmoid Function

$$g(f) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

- Pros:
 - Bounds output between 0 and 1
 - Simple derivative:
- Cons:
 - Reduces error signal in back propagation (due to small derivatives) which can slow training.

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

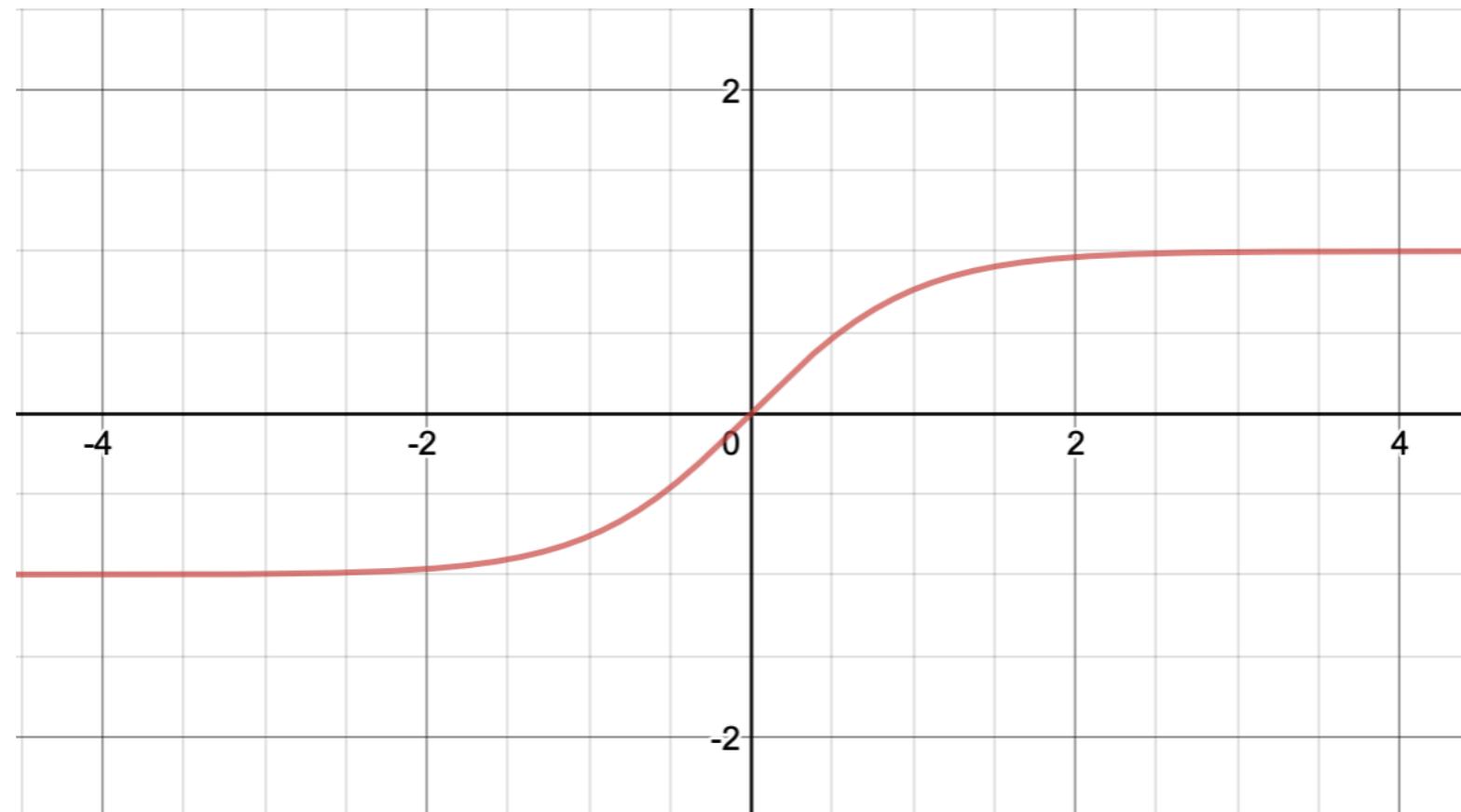


Tanh Function

$$g(x) = \tanh(x)$$

- Pros:
 - Rescaled sigmoid function
 - Bounded between -1 and 1
 - Simple derivative:
- Cons:
 - Reduces error signal in back propagation (due to small derivatives) which can slow training.

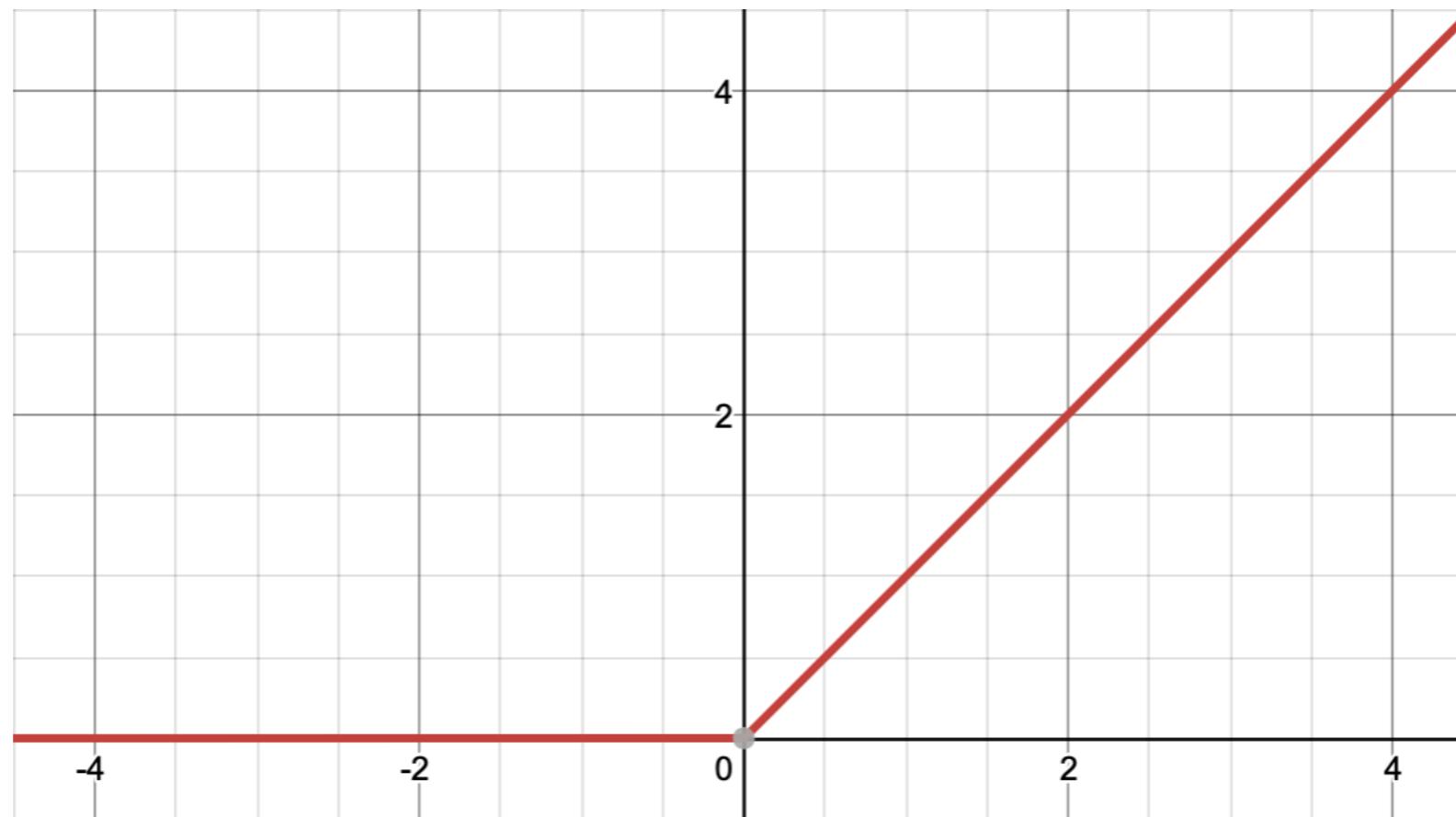
$$\tanh'(x) = 1 - \tanh^2(x)$$



ReLU Function

$$g(x) = \max(0, x)$$

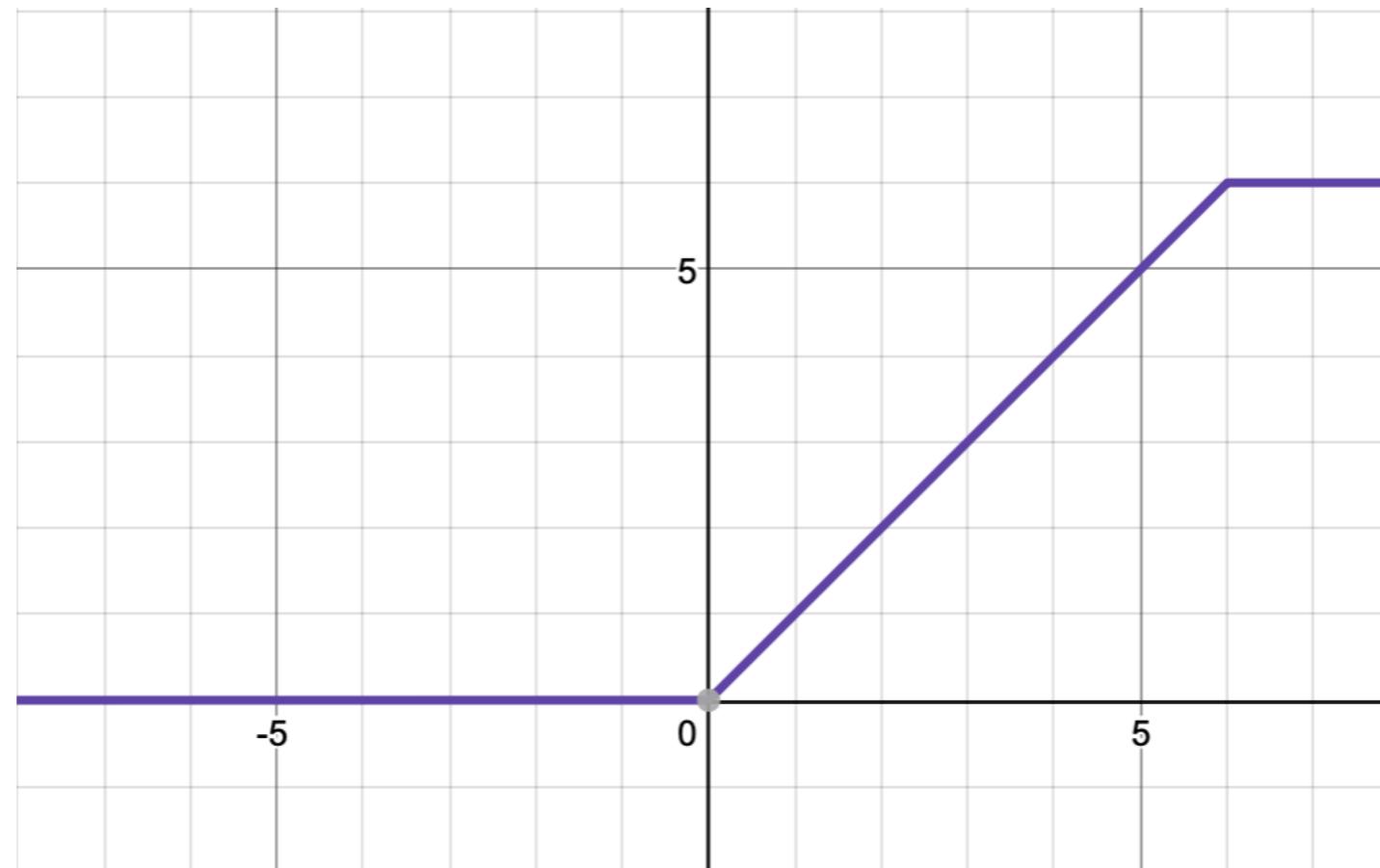
- Pros:
 - Simple to compute
 - Derivative of 0 or 1 (also simple to compute)
- Cons:
 - Partially unbounded
 - Causes neurons to get stuck and "die" (if sum of inputs + bias drops below 0)



ReLU-6 Function

$$g(x) = \min(\max(0, x), 6)$$

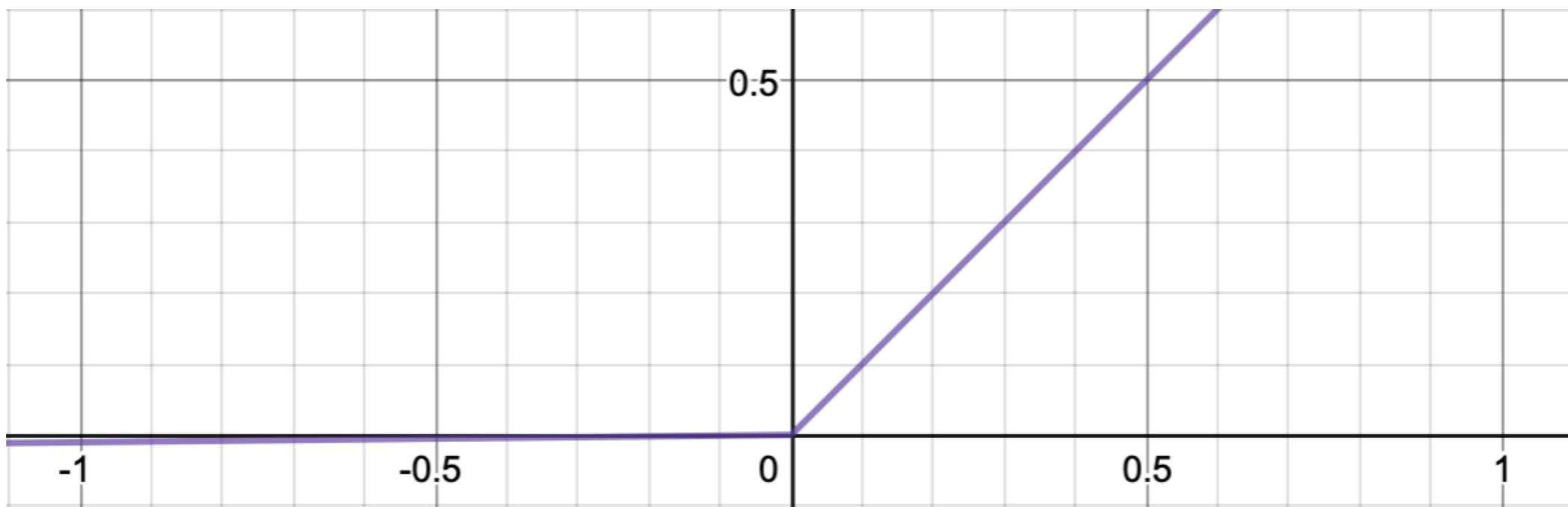
- Pros:
 - Simple to compute (but less so than ReLU)
 - Derivative of 0 or 1 (also simple to compute)
 - Bounded between 0 and 6
- Cons:
 - Neurons can die (if inputs + bias <= 0)
 - Neurons can get stuck (if inputs + bias >= 6)



Leaky ReLU Function

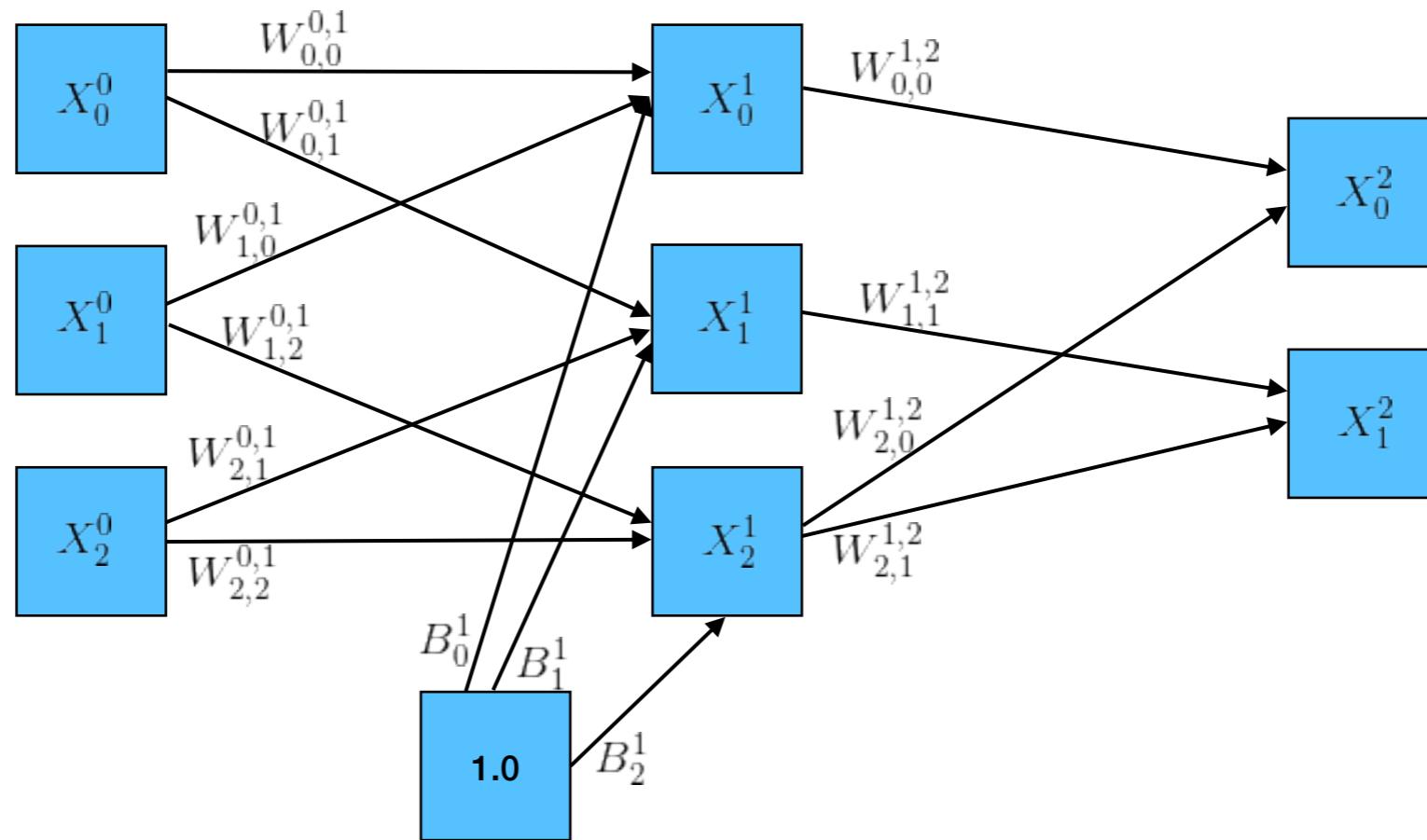
$$f(x) = \begin{cases} 0.01x, & \text{if } x \leq 0 \\ x, & \text{otherwise} \end{cases}$$

- Pros:
 - Simple to compute (but less so than ReLU)
 - Derivative of -0.01 or 1 (also simple to compute)
 - Prevents neurons dying
- Cons:
 - Partially unbounded



The Forward Pass

The Forward Pass



- Now we can add the activation function into the calculation of our node values:

$$X_0^1 = g((1.0 * B_0^1) + (X_0^0 * W_{0,0}^{0,1}) + (X_1^0 * W_{1,0}^{0,1}))$$

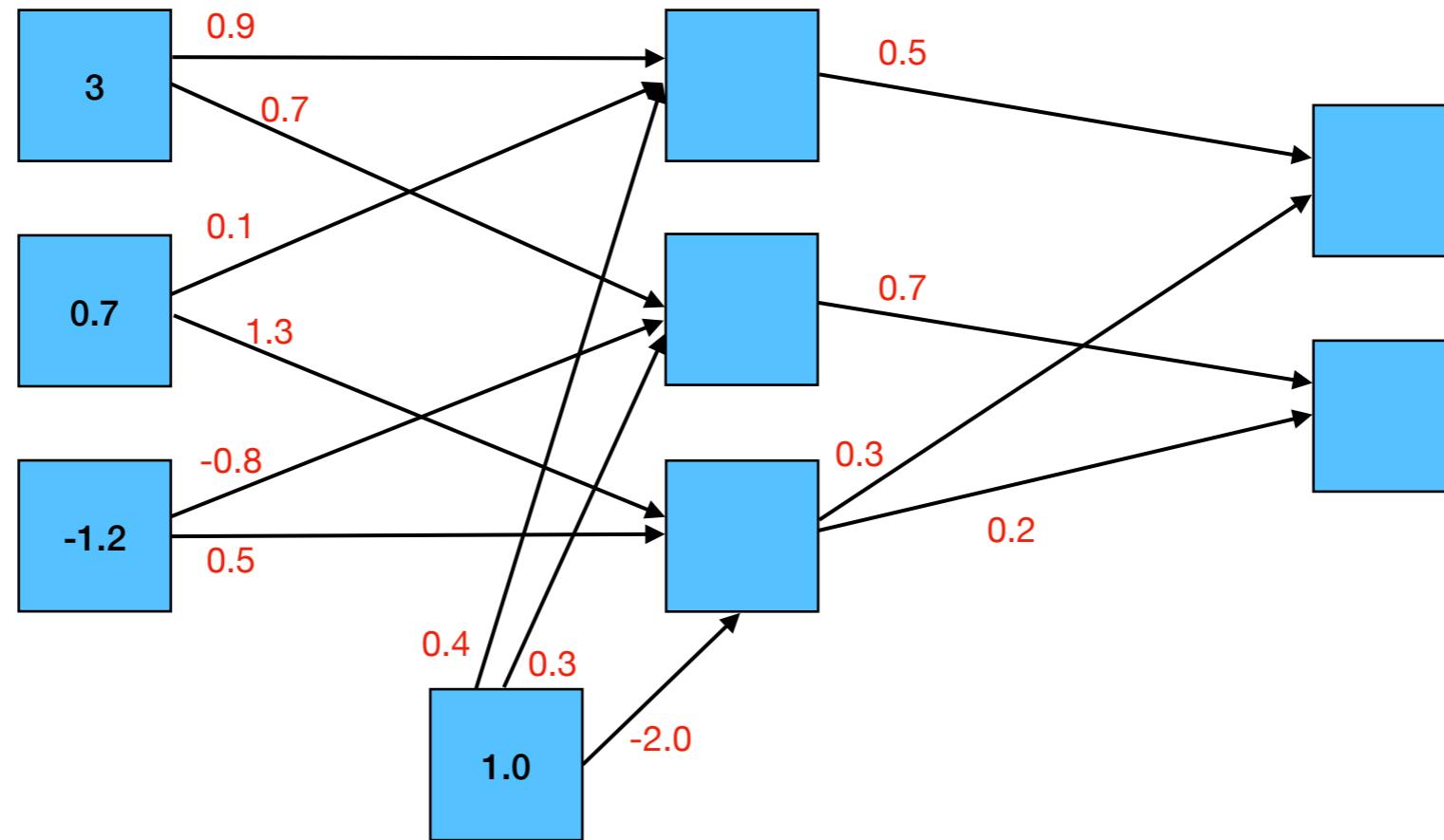
$$X_1^1 = g((1.0 * B_1^1) + (X_0^0 * W_{0,1}^{0,1}) + (X_2^0 * W_{2,1}^{0,1}))$$

$$X_2^1 = g((1.0 * B_2^1) + (X_1^0 * W_{1,2}^{0,1}) + (X_2^0 * W_{2,2}^{0,1}))$$

$$X_0^2 = g((X_0^1 * W_{0,0}^{1,2}) + (X_2^1 * W_{2,0}^{1,2}))$$

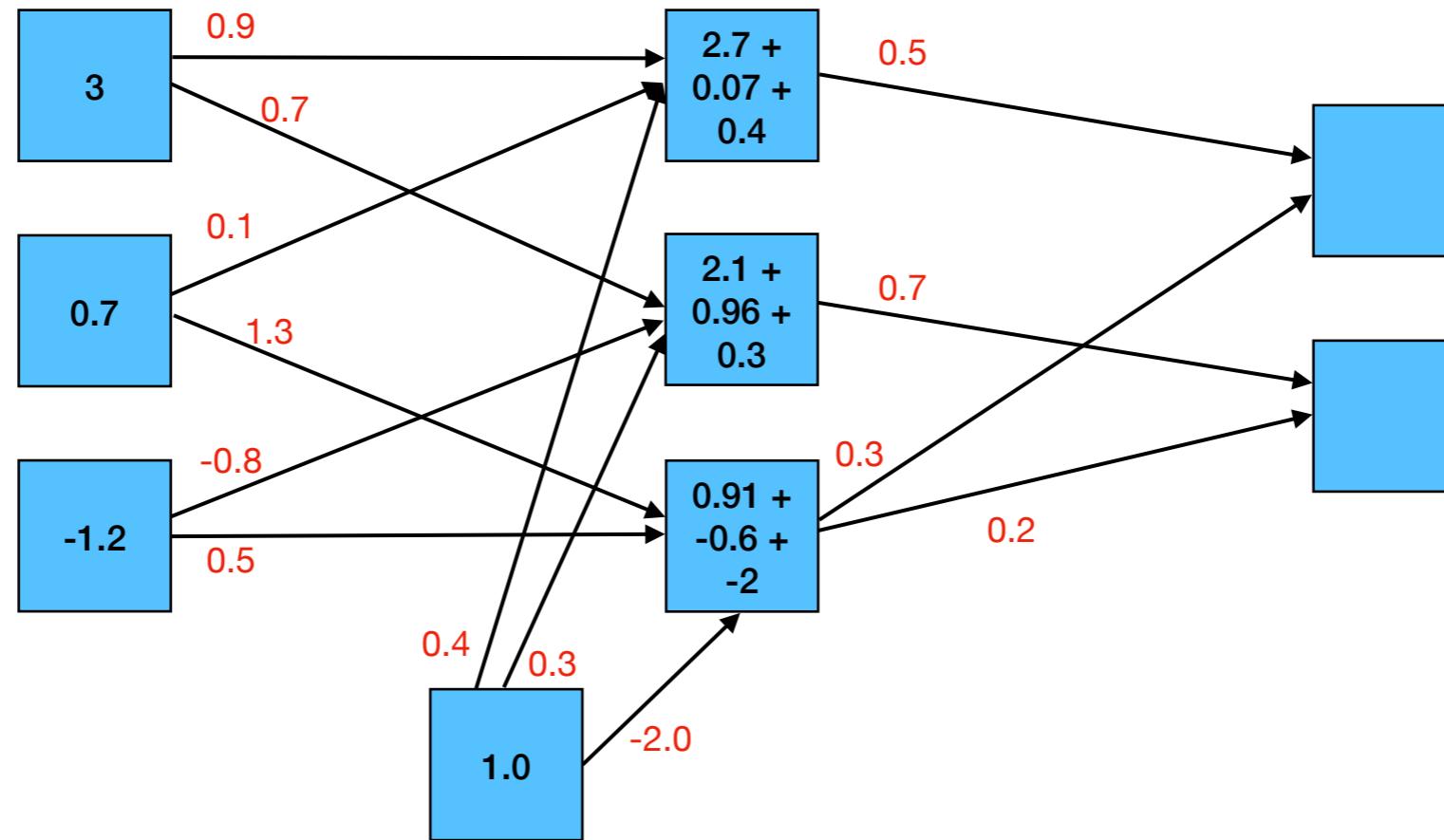
$$X_1^2 = g((X_0^1 * W_{0,1}^{1,2}) + (X_2^1 * W_{2,1}^{1,2}))$$

The Forward Pass - An example



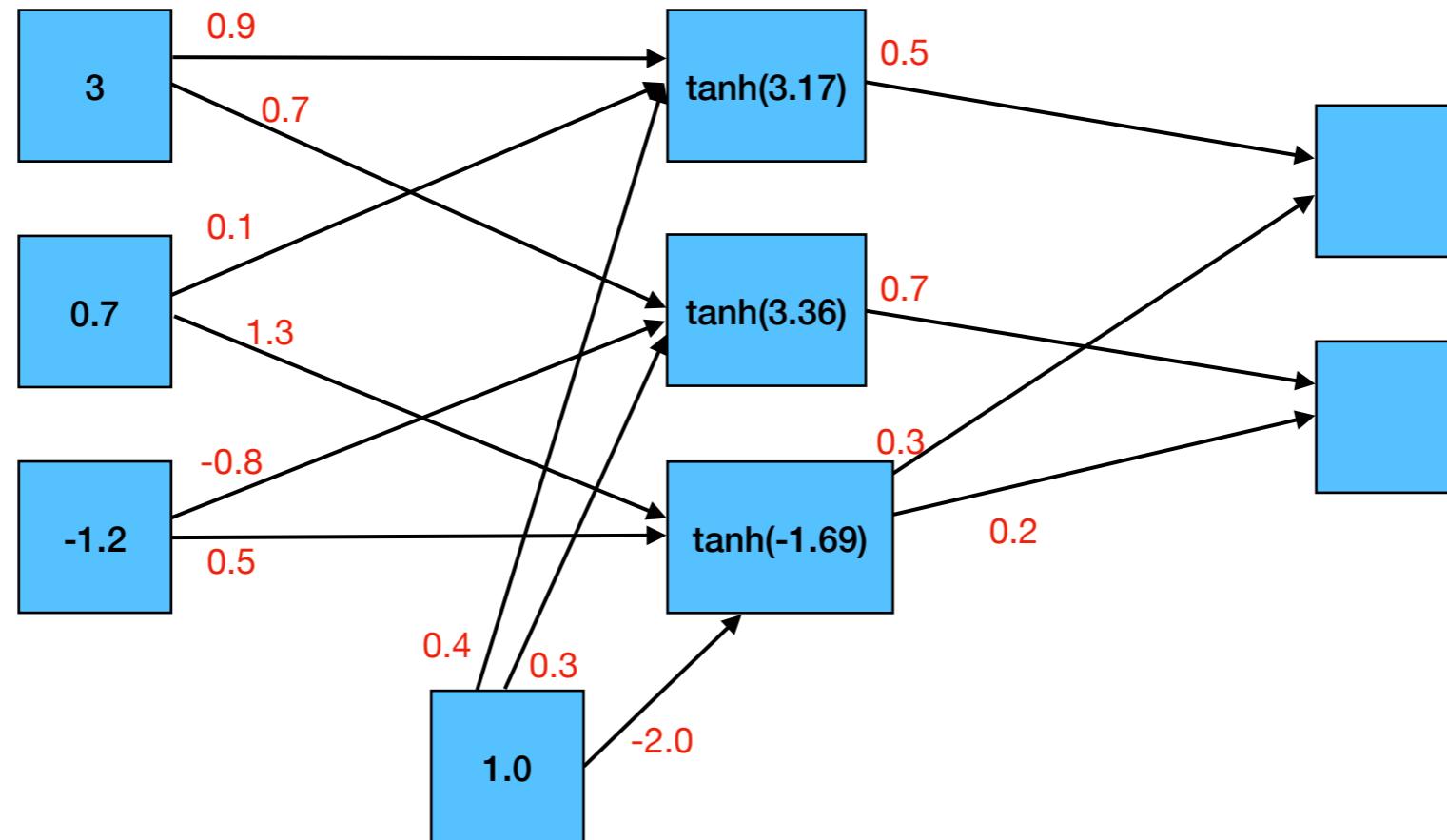
- We set the input node values from our input data to 3, 0.7, and -1.2.
- Weights for each edge have already been assigned (in red).
- Each node will use the tanh activation function.

The Forward Pass - An example



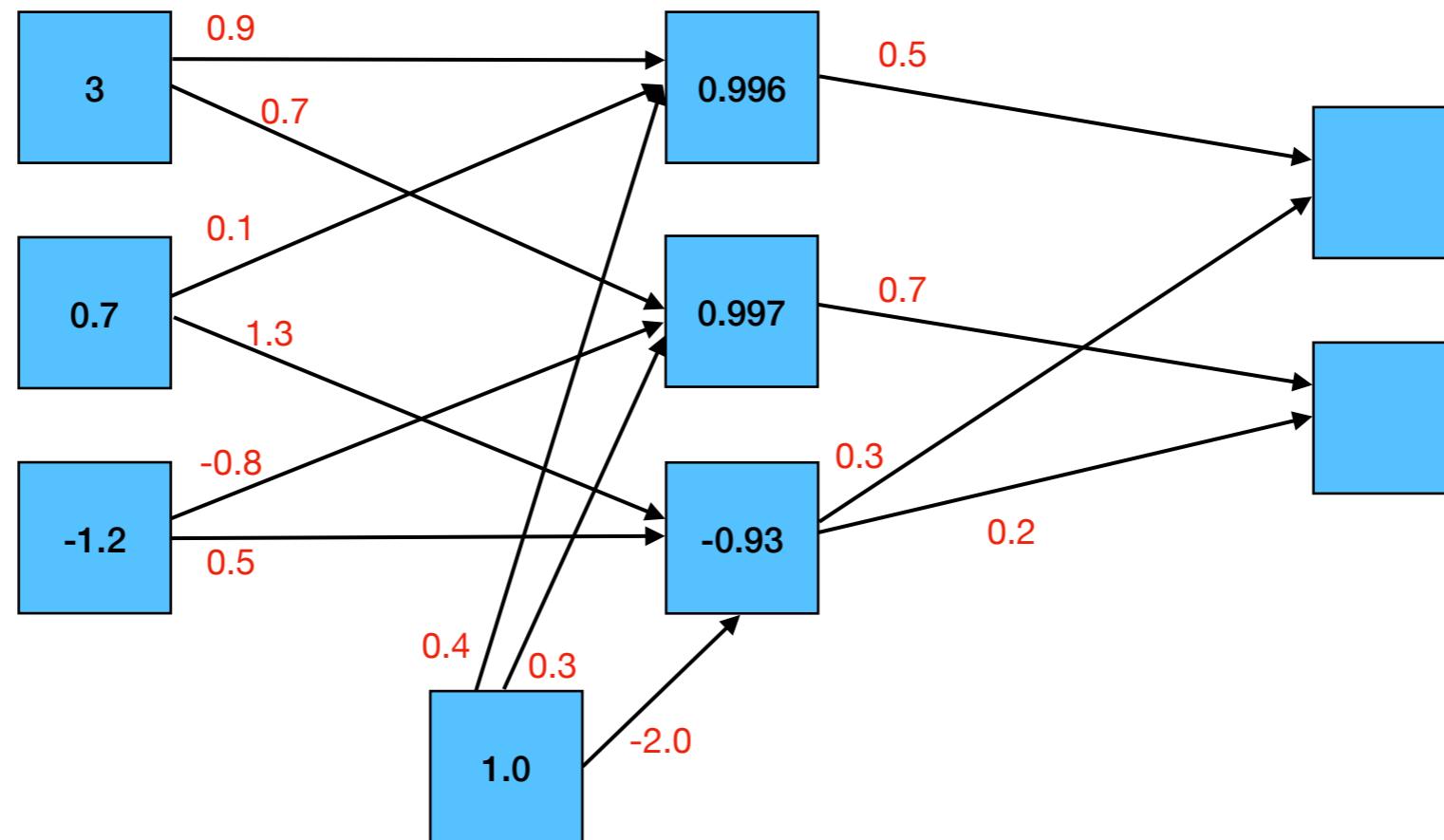
- We multiply each node the outgoing weight and sum these on the target nodes in the hidden layer.

The Forward Pass - An example



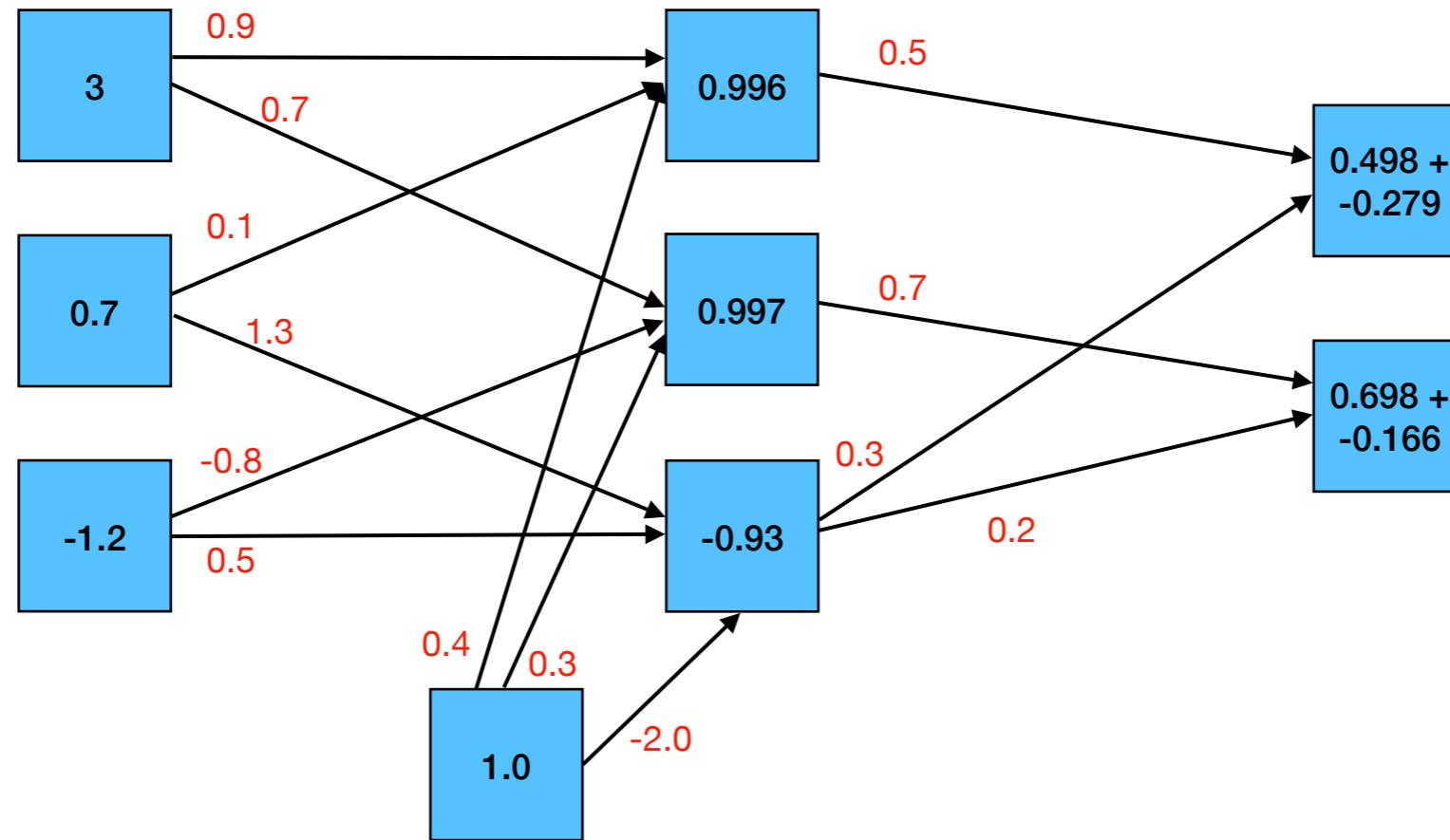
- We calculate the sums before applying the activation function.

The Forward Pass - An example



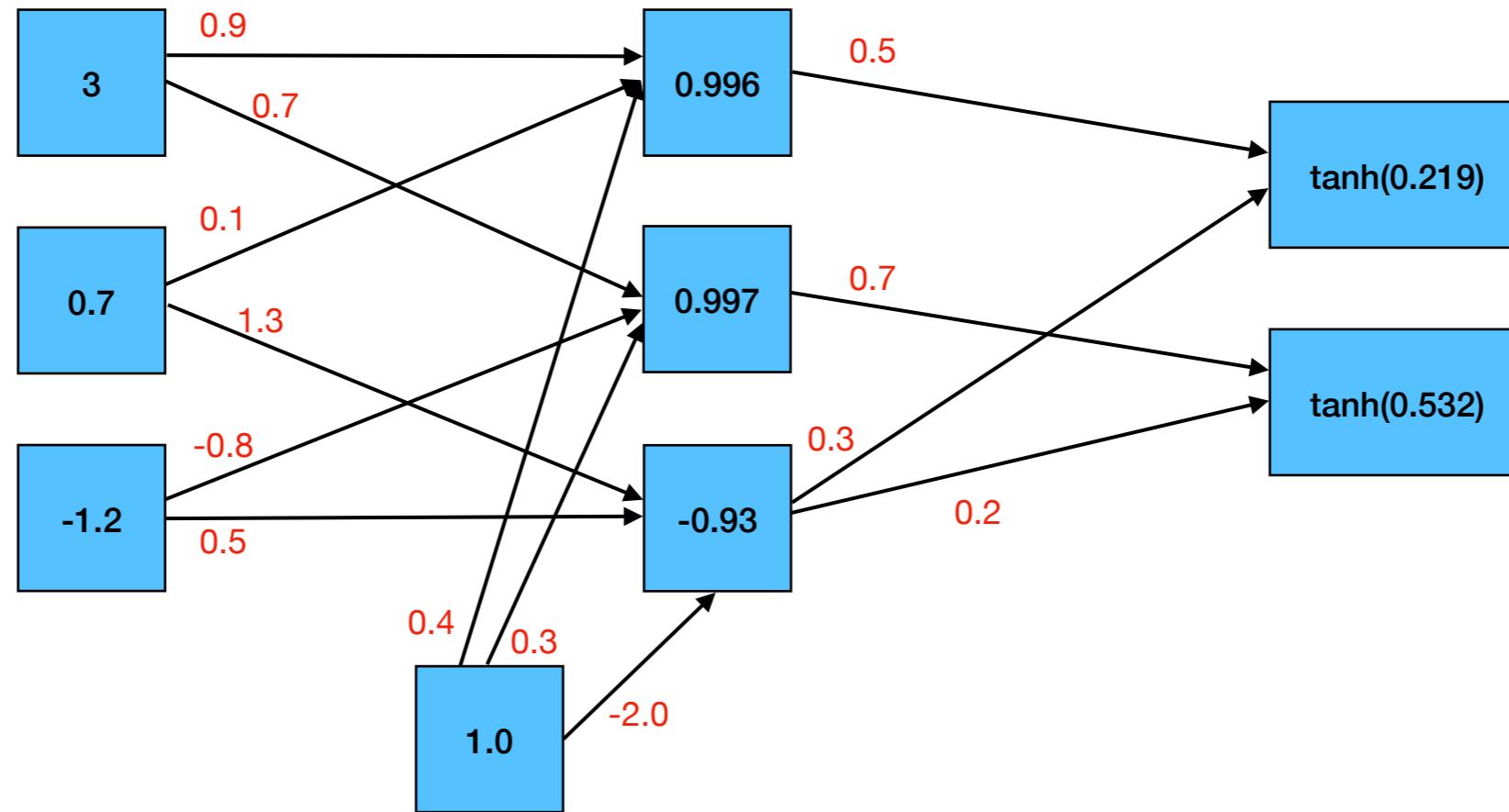
- Apply the activation function.

The Forward Pass - An example



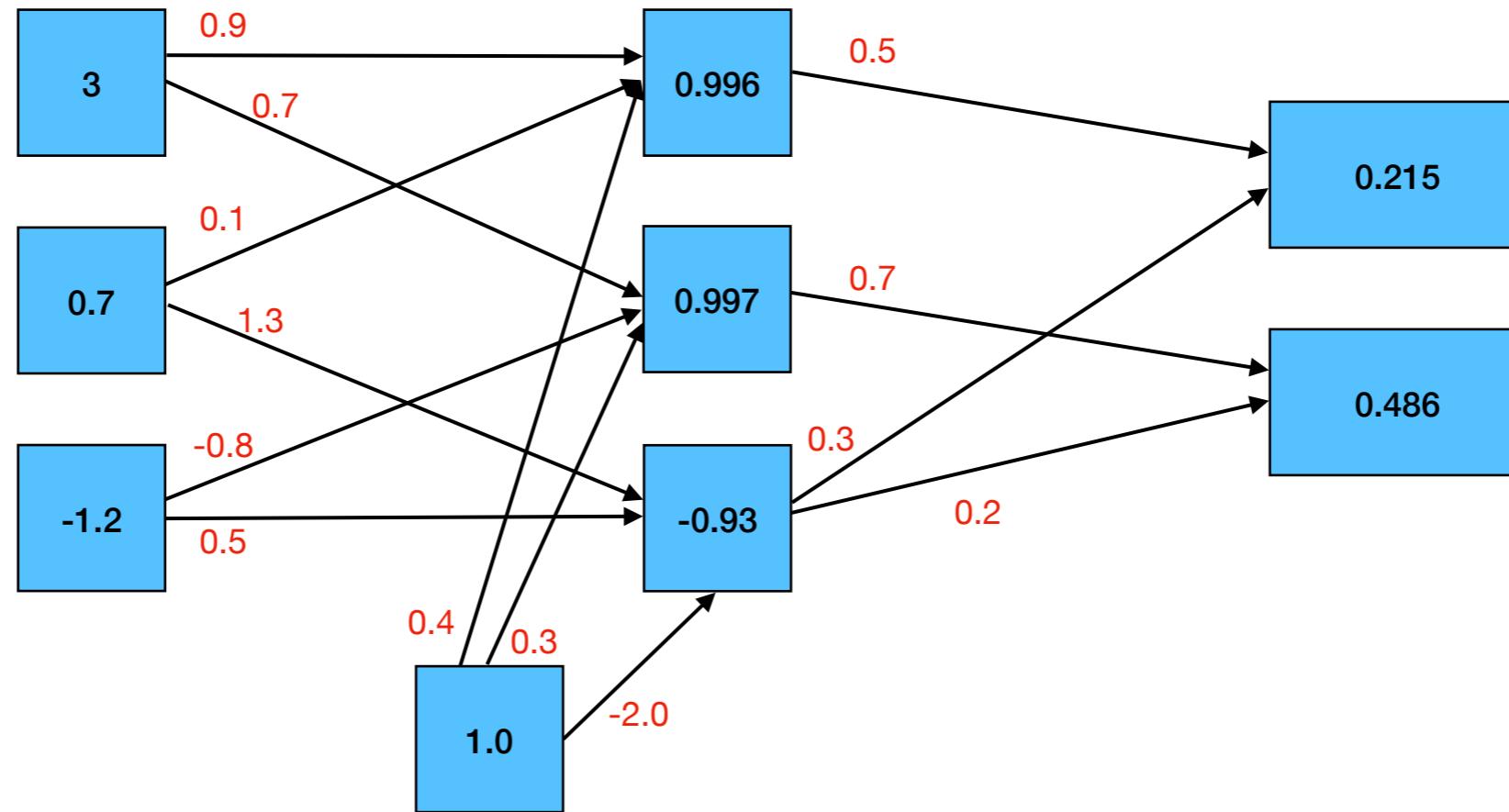
- Now multiply the values in the hidden nodes by their outgoing weights and add them in the output nodes.

The Forward Pass - An example



- Calculate the sum before applying the activation function

The Forward Pass - An example



- Calculate the activation function and we have our outputs.

The Forward Pass

- We can generate some pseudo-code to initialize a simple fully connected feed forward neural network and perform the forward pass for calculating the output:

```
//The layer sizes of the neural network:  
//input layer: 5 nodes  
//hidden layer 1: 5 nodes  
//hidden layer 2: 4 nodes  
//hidden layer 2: 3 nodes  
//hidden layer 2: 4 nodes  
//output layer: 2 nodes  
layer_sizes = [5, 5, 4, 3, 4, 2];  
  
nodes = array(layer_sizes.length)  
//we will have a bias value for each node  
bias = array(layer_sizes.length)  
  
//we will make the NN fully connected  
weights = array(layer_sizes.length)  
  
for (i = 0; i < layer_sizes.length; i++) {  
    //for each layer in the network,  
    //append an array of size N where N  
    nodes[i] = array(layer_sizes[i])  
    bias[i] = array(layer_sizes[i])  
  
    if (i == layer_sizes.length - 1) {  
        //the output layer doesn't have  
        //any output weights  
        break  
    }  
  
    //each node will have its own set of weights  
    //to each node in the next layer  
    weights[i] = array(layer_sizes[i])  
    for (j = 0; j < layer_sizes[i]; j++) {  
        weights[i][j] = array(layer_sizes[i+1])  
    }  
}  
  
//FORWARD PASS  
//set the input values from the input data  
for (i = 0; i < layer_sizes[0]; i++) {  
    nodes[0][i] = input_data[i]  
}  
  
//nodes[LAYER][NUMBER]  
//bias[LAYER][NUMBER]  
//weights[INPUT LAYER][INPUT NUMBER][OUTPUT NUMBER]  
  
set all hidden and output node values to 0  
  
//calculate the values for every node in the  
//hidden and output layers  
for (i = 1; i < layer_sizes.length; i++) {  
  
    //iterate over the nodes in the current layer  
    for (j = 0; j < layer_sizes[i]; j++) {  
  
        //iterate over the nodes in the previous layer  
        for (k = 0; k < layer_sizes[i-1]; k++) {  
            nodes[i][j] += nodes[i-1][k] * weights[i-1][k][j]  
        }  
  
        //add the bias for the node  
        nodes[i][j] += bias[i][j]  
  
        //apply the activation function  
        nodes[i][j] = g(nodes[i][j])  
    }  
}  
  
//The outputs of the neural network are the two nodes  
//in the last (5th) layer  
output[0] = nodes[5][0]  
output[1] = nodes[5][1]
```

The Forward Pass

- We can also take an object oriented approach to clean this up a bit:

```
class Node {  
    int depth;  
  
    double pre_activation;  
    double post_activation;  
    double bias;  
  
    Edge[] input_edges;  
    Edge[] output_edges;  
  
    Node(int depth) { this.depth = depth; }  
  
    void reset() {  
        pre_activation = 0;  
        post_activation = 0;  
    }  
  
    void propagateForward() {  
        if (depth > 0) {  
            //input nodes don't apply an activation function  
            post_activation = activation_function(pre_activation + bias);  
        }  
  
        for (i = 0; i < output_edges.size; i++) {  
            output_edges[i].output_node.pre_activation += post_activation * output_edges[i].weight;  
        }  
    }  
  
    class Edge {  
        Node input_node;  
        Node output_node;  
        double weight;  
  
        Edge(Node input_node, Node output_node) {  
            this.input_node = input_node;  
            this.output_node = output_node;  
  
            //add the edge to the output edges of the  
            //input node and the input edges of the  
            //output node  
            input_node.output_edges.append(this);  
            output_node.input_edges.append(this);  
        }  
    }  
}  
  
nodes = []  
layer_sizes = [5, 5, 4, 3, 4, 2];  
  
for (i = 0; i < layer_sizes.length; i++) {  
    nodes.append([]); //append an empty array for this layer  
  
    for (j = 0; j < layer_sizes[i]; j++) {  
        nodes[i].append(new Node(i));  
    }  
  
    if (i >= 1) {  
        //add an edge between every node in the previous  
        //layer and every edge in the current layer  
        for (j = 0; j < layer_sizes[i]; j++) {  
            for (k = 0; k < layer_sizes[i-1]; k++) {  
                new Edge(nodes[i-1][k], nodes[i][j]);  
            }  
        }  
    }  
}  
  
//set the input node values  
for (i = 0; i < nodes[0].length; i++) {  
    //set the values for the input nodes  
    nodes[0][i].post_activation = input_data[i]  
}  
  
//FORWARD PASS  
for (i = 0; i < nodes.length; i++) {  
    for (j = 0; j < nodes[i].length; j++) {  
        nodes[i][j].propagateForward();  
    }  
}  
  
//The outputs of the neural network are the post activation  
//values of the two nodes in the last layer  
output[0] = nodes[5][0].post_activation;  
output[1] = nodes[5][1].post_activation;
```

The "Problem"

The "Problem"

- For a given architecture, it is fairly straightforward to calculate the outputs or predictions given the inputs, a set of weights (biases are just a special case of weights).
- Aside from challenges in determining a good architecture, why are neural networks such a challenging and hot field of study?

How do we find the best weights?

- Given a set of inputs and expected outputs, along with a neural network architecture -- how do we find the weights which will give us the best predictions?
- Each output node value (z_i) has an expected value (e_i) from the training data set:

$$\text{error}(\text{inputs}, W) = \sum_{i=1}^K z_i - e_i$$

- As the inputs are fixed, we now have a *numerical optimization* problem to find the set of weights W that minimize the error function across all inputs.
- Determining the best weights is the process of *training* a neural network.

How do we find the best weights?

- But it is even more complicated than that!
- Doing well on known inputs in our training data is good -- but even more importantly we need our neural network to perform well on data it has not seen before.
- We typically leave aside some data as *testing* data (and sometimes an additional set of *validation* data -- more on those in Lecture 3) which we do not train on.
- We need to make sure our network does well on unseen data (this is called *generalizability*). So we can't simply minimize the error on our training data.

How do we find the best weights?

- Training a neural network is typically done using an algorithm called *backpropagation*, which is a specialized version of *gradient descent* - a common numerical optimization technique.
- However, gradient descent/backpropagation is not without its own set of issues, and researching other strategies for finding the best weights (e.g., gradient free methods, etc) is another area of research.
- We will go into training methods in the next lecture.

Lecture 2

Training Options and The Backward Pass

DSCI-789: Neural Networks for Data Science

Travis Desell (tjdvse@rit.edu)
Associate Professor
Department of Software Engineering



ROCHESTER INSTITUTE OF TECHNOLOGY

Overview

- How can you train a neural network?
- Calculating a Numerical Gradient
- The Chain Rule
- The Backward Pass
- A Backward Pass Example
- Backward Pass with Multiple Training Samples
- Validating Gradients

How can you train a neural network?

The problem (again)

- We have a function f which takes a number of parameters
 $x_1 \dots x_n : f(x_1 \dots x_n)$
- In this case, the output of our function f is the error of our neural network predictions.
- $x_1 \dots x_n$ are the weights and biases.
- We want to find the values of x which minimize the output of f (the error), i.e.: $\arg \min f(x)$
- This is a numerical optimization problem.

How can you train a neural network?

- What options do we have to perform this numerical optimization?
- We're going to review our options, slowest to fastest:
 - Grid search
 - Random search
 - Evolutionary/Bio-Inspired Algorithms
 - Gradient Descent

Grid Search

- This is mostly to demonstrate the way **not** to perform numerical optimization.
- Select a grid size for each parameter:
 - $x_1: [-1.0, 0.9, 0.8 \dots, 0.0, 0.1, 0.2, 0.3, 0.4, \dots 1.0]$
 - $x_2: [-1.0, 0.9, 0.8 \dots, 0.0, 0.1, 0.2, 0.3, 0.4, \dots 1.0]$
 - ...
 - $x_n: [-1.0, 0.9, 0.8 \dots, 0.0, 0.1, 0.2, 0.3, 0.4, \dots 1.0]$
- Evaluate every combination and keep the best.
- This is **very** computationally expensive. In the above example, with n parameters the runtime is $O(21^n)$, and our parameter space is very limited.
- Pros:
 - very easy to implement
 - embarrassingly parallel (can evaluate each x independently)
 - gradient calculation not required
- Cons:
 - Prohibitively expensive
 - Limited in possible x values

Random Search

- This is mostly to demonstrate another way **not** to perform numerical optimization.
- Assign each parameter randomly:
 - `for (i = 1..n): xi = Math.random(-1.0, 1.0);`
- Repeatedly evaluate random parameters, keeping the best.
- The runtime of this probably faster than grid search, and also isn't limited to particular increments. But the runtime is also random. Can be pretty useful for testing your forward pass.
- Pros:
 - very easy to implement
 - embarrassingly parallel (can evaluate each x independently)
 - gradient calculation not required
- Cons:
 - Prohibitively expensive (but not as bad as grid search)

Evolutionary/Nature Inspired Algorithms

- General strategy:
 - Create a population of randomly initialized x values. 
 - Progressively generate new populations based using various **heuristics** (e.g., mutation, reproduction, selection).
 - Repeat until a good solution is found.
- Examples worth looking into:
 - Differential evolution (https://en.wikipedia.org/wiki/Differential_evolution)
 - Particle swarm optimization (https://en.wikipedia.org/wiki/Particle_swarm_optimization)
 - CMA-ES (<https://en.wikipedia.org/wiki/CMA-ES>) 
- Difficult to determine how long they will take to find a solution. Typically 1000 to 20000 populations (each with 20-100 different x values), so 20,000 to 2,000,000 forward passes (each forward pass calculates an x value). Also do not scale well to a large number of parameters (n values > 200 , where x is $x_1 \dots x_n$) -- although this is an area of research.
- Pros:
 - can be easy to implement
 - easy to parallelize (especially if using an asynchronous strategy)
 - **gradient calculation not required** (can have non-differentiable activation functions)
- Cons:
 - Still computationally expensive
 - Runtime can be fairly unbounded with large n

Gradient Descent

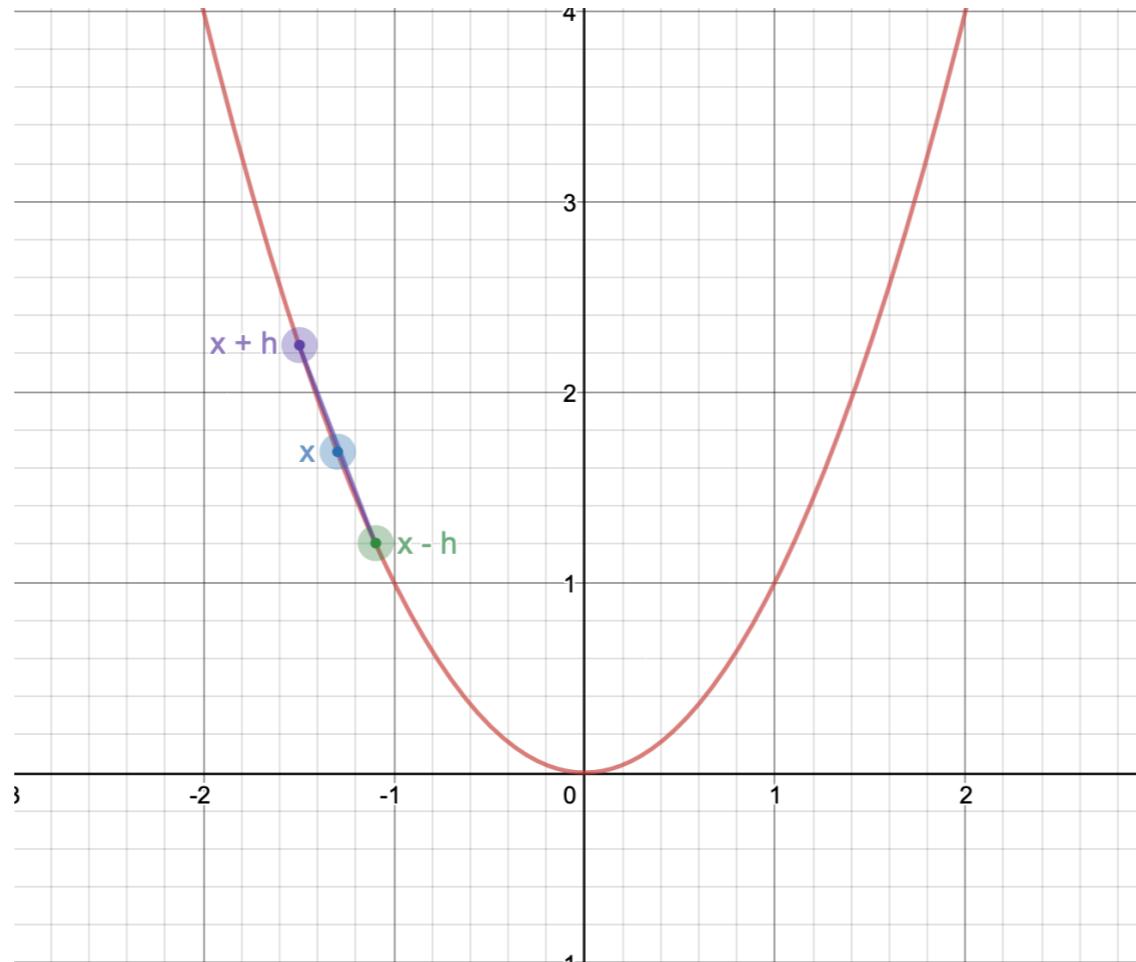
- General strategy:
 - Start at a randomly (or quasi-randomly) selected x
 - Calculate the gradient (g) for that x (i.e., what is the slope of each x_i)
 - Update each value of x by subtracting the gradient times a small learning rate (n):
 - $x_i = x_i - (n * g_i)$
 - Repeat the gradient calculation and x update until the error is minimized.
- Can be very fast, for feed forward NNs, can minimize error in 50-100 epochs (more on those later). RNNs are slower, \sim 1000-2000 epochs.
- Each epoch involves a forward pass and gradient calculation for each training sample (instance).
- Numeric gradient calculation costs $2n$ forward passes. Gradient calcuation (via backpropagation) costs ~ 2 forward passes.
- Pros:
 - Fastest method we know of.
- Cons:
 - Backpropagation can be tricky to implement.
 - Not (currently) parallelizable (need to parallelize the forward pass instead, which can work well for CNNs and RNNs on GPUs but not all architectures).
 - Can get stuck in local minima (a problem for RNNs, there is research that suggests feed foward NNs are convex and do not have local minima).

Gradient Descent

- Gradient descent revolves around knowing the gradient of the function f at a point x .
- So how do we calculate the gradient?

Calculating a Numeric Gradient

Numeric Gradients



- If you remember some calculus, one way to estimate the gradient is called the **finite difference method** (in this case x is our weights and biases):

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h}$$

- <https://www.desmos.com/calculator/hawp8cseji>

Numeric Gradients

- As a word of warning, you may also come across this method, as it is one way the finite difference method is commonly proven and shown:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

- While this is mathematically correct, when implementing it on a computer it tends to have numeric issues. So even though you may look at it and think you can save N forward passes, the results will not be sufficiently approximate to compare to your backward pass.

Numeric Gradients

- We need to do this for a vector, not just a single x value however.
- This is fairly straight forward, we can repeat the same process for each element in x (where x are our weights and biases):

$$\frac{\partial f}{\partial x_i} = \frac{f(x|x_i=x_i+h) - f(x|x_i=x_i-h)}{2h}$$

- So for each x_i in x we can calculate the derivative for that parameter by doing a forward pass keeping all other x's the same, but adding and subtracting h from that parameter.

Numeric Gradients

- In code this is really simple:

```
Instance y = ... //y is the input instance for our neural network
w = getWeights(); //w is each weight in the neural network
w_test = w.copy();
for (i = 0; i < w.length; i++) {
    w_test[i] = w[i] + h;
    //do a forward pass using the modified weight
    output1 = forward_pass(y, w_test);

    w_test[i] = w[i] - h;
    //do a forward pass using the modified weight
    output2 = forward_pass(y, w_test);

    gradient[i] = (output1 - output2) / (2.0 * h)
    w_test[i] = w[i]; 
```

- Note we need to have a small h (0.0001 is not bad).
- We can calculate the gradient for a neural network even faster however, and that's where the backward pass of backpropagation comes in.
- But it is good to implement it this was so we can verify we're calculating the gradient the backward pass correctly.

Numeric Gradients

- Depending on how we're doing backpropagation (more on that next lecture), we may need to calculate the gradient over a set of inputs (the training samples or instances). We can define a training sample as y .
- We could calculate the gradient of the y values however these are fixed and we can't change them so it's not worthwhile.
- Luckily this is pretty straightforward, we just sum up the outputs for each instance y .

```
Instance[] ys = ...  
  
w_test = w.copy(); //w are the weights and biases in our network  
for (i = 0; i < w.length; i++) {  
    w_test[i] = w[i] + h;  
    output1 = 0;  
    foreach (Instance y in ys) {  
        output1 += forward_pass(y, w_test);  
    }  
  
    w_test[i] = w[i] - h;  
    output2 = 0;  
    foreach (Instance y in ys) {  
        output2 += forward_pass(y, w_test);  
    }  
  
    gradient[i] = (output1 - output2) / (2.0 * h)  
    w_test[i] = w[i]; //reset w_test[i]  
}
```

The Chain Rule

The Chain Rule

- First let's remember the literal definition of a gradient. If we have a function $f(x)$, the gradient of f at x , $\nabla f(x)$ or $f'(x)$ is the slope of that function at x .
- Basically, if we have some tiny value z , it will be  much the output of $f(x)$ will change if we add z to x .
- What backpropagation does is calculate $\nabla f(x)$ recursively by the **chain rule**.

Partial Derivatives

- In neural networks we typically have additions, multiplications (and later in CNNs, max/min operations).
- We have various rules we can use to calculate partial derivatives of a function with multiple variables with respect to each variable:

$$f(x_1, x_2) = x_1 * x_2 \rightarrow \frac{\partial f}{\partial x_1} = x_2 \quad \frac{\partial f}{\partial x_2} = x_1$$

$$f(x_1, x_2) = x_1 + x_2 \rightarrow \frac{\partial f}{\partial x_1} = 1 \quad \frac{\partial f}{\partial x_2} = 1$$

$$f(x_1, x_2) = \max(x_1, x_2) \rightarrow \frac{\partial f}{\partial x_1} = \begin{cases} 1, & \text{if } x_1 \geq x_2 \\ 0, & \text{otherwise} \end{cases} \quad \frac{\partial f}{\partial x_2} = \begin{cases} 1, & \text{if } x_2 \geq x_1 \\ 0, & \text{otherwise} \end{cases}$$

$$f(x_1, x_2) = \min(x_1, x_2) \rightarrow \frac{\partial f}{\partial x_1} = \begin{cases} 0, & \text{if } x_1 \geq x_2 \\ 1, & \text{otherwise} \end{cases} \quad \frac{\partial f}{\partial x_2} = \begin{cases} 0, & \text{if } x_2 \geq x_1 \\ 1, & \text{otherwise} \end{cases}$$

Partial Derivatives: Multiplication

$$f(x_1, x_2) = x_1 * x_2 \rightarrow \frac{\partial f}{\partial x_1} = x_2 \quad \frac{\partial f}{\partial x_2} = x_1$$



- What happens to the output of f when we increase x_1 by 1? It increases by x_2 .
- What happens to the output of f when we increase x_2 by 1? it increases by x_1 .
- In this regard, this rule for multiplication makes a lot of sense, the derivative of f with respect of x_1 is x_2 , and vice versa.

Partial Derivatives: Addition

$$f(x_1, x_2) = x_1 + x_2 \quad \rightarrow \quad \frac{\partial f}{\partial x_1} = 1 \quad \frac{\partial f}{\partial x_2} = 1$$

- What happens to the output of f when we increase x_1 by 1? It increases by 1.
- What happens to the output of f when we increase x_2 by 1? It also increases by 1.
- So this rule also makes a lot of sense, the derivative of f with respect to x_1 is 1, as is the derivative of f with respect to x_2 . Both effect the output equally.

Partial Derivatives: min/max

$$f(x_1, x_2) = \max(x_1, x_2) \rightarrow \frac{\partial f}{\partial x_1} = \begin{cases} 1, & \text{if } x_1 \geq x_2 \\ 0, & \text{otherwise} \end{cases} \quad \frac{\partial f}{\partial x_2} = \begin{cases} 1, & \text{if } x_2 \geq x_1 \\ 0, & \text{otherwise} \end{cases}$$

$$f(x_1, x_2) = \min(x_1, x_2) \rightarrow \frac{\partial f}{\partial x_1} = \begin{cases} 0, & \text{if } x_1 \geq x_2 \\ 1, & \text{otherwise} \end{cases} \quad \frac{\partial f}{\partial x_2} = \begin{cases} 0, & \text{if } x_2 \geq x_1 \\ 1, & \text{otherwise} \end{cases}$$

- How does the $f = \max(x_1, x_2)$ change when we increase x_1 by 1? In this case it depends - it increases by 1 if $x_1 \geq x_2$. Same for vice versa.
- The opposite logic can be applied for min.

The chain rule

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

- The chain rule gives us a way to "chain" nested expressions together to find a partial derivative.
- If we know the derivative of f with respect to q , and we know the derivative of q with respect to x ; the derivative of f with respect to q is their product (we multiply them together).

The chain rule: An Example

$$f(x, y, z) = z(x + 3y)$$

- With the above function, let's assume z is our fixed input and we want to find the derivative of f with respect to x and y, i.e., how much will the output change if we add 1 to x and how much will it change if we add 1 to y.
- We apply the rules recursively, from outer expressions to inner expressions and apply the chain rule to chain things together.

The chain rule: an example

$$f(x, y, z) = z(x + 3y)$$

$$q = x + 3y$$

- We first apply the multiplication rule (let's substitute $x + 3y$ with q)
- This makes our function: $f(z, q) = z * q$

$$\frac{\partial f}{\partial z} = q$$

$$\frac{\partial f}{\partial q} = z$$

- z is fixed so we don't care about its gradient (we can't modify z), but now we can figure out the partial derivatives of q (the $x + 3y$ part)

The chain rule: an example

$$q(x, y) = x + 3y$$

$$r = 3y$$

$$q = x + r$$

- Now we do the addition rule (let's substitute r for 3y):

$$\frac{\partial q}{\partial r} = 1$$

$$\frac{\partial q}{\partial x} = 1$$

- Now we're just worried about the partial derivatives of r.

The chain rule: an example

$$r = 3y$$

- We do another multiplication rule:

$$\frac{\partial r}{\partial y} = 3$$

- We don't do the derivative with respect to 3 because that's a fixed value.

The chain rule: an example

- Now we have all the pieces:

$$\frac{\partial f}{\partial z} = q$$

$$\frac{\partial q}{\partial r} = 1$$

$$\frac{\partial r}{\partial y} = 3$$

$$\frac{\partial f}{\partial q} = z$$

$$\frac{\partial q}{\partial x} = 1$$

- We can apply the chain rule to get the derivative of f with respect to y and the derivative of f with respect to x :

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial r} \frac{\partial r}{\partial y} = z * 1 * 3 = 3z$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} = z * 1 = z$$

The chain rule: an example

$$f(x, y, z) = z(x + 3y)$$

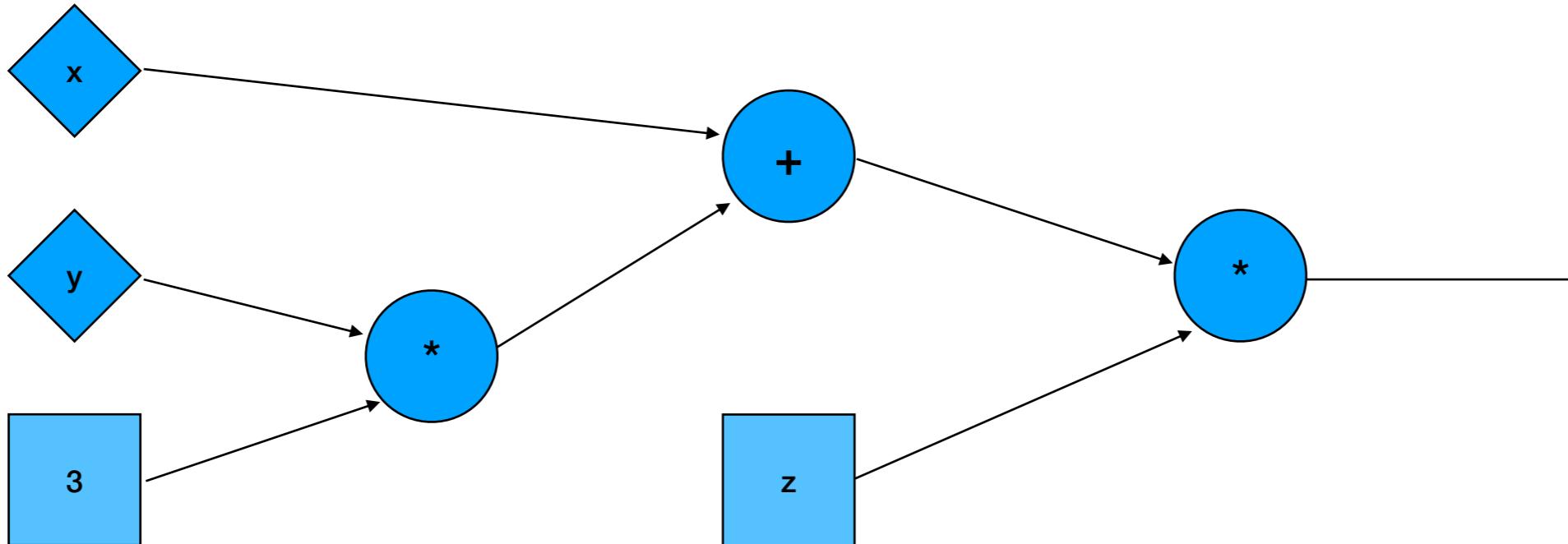
$$\frac{\partial f}{\partial x} = z$$

$$\frac{\partial f}{\partial y} = 3z$$

- So does this make sense?
- What happens when we increase x by 1? It's multiplied by z so the total output of f is increased by z .
- What happens when we increase y by 1? It's multiplied by 3 and then by z , so the total output of f is increased by $3z$.
- Looks like we calculated the gradient with respect to y and x correctly!

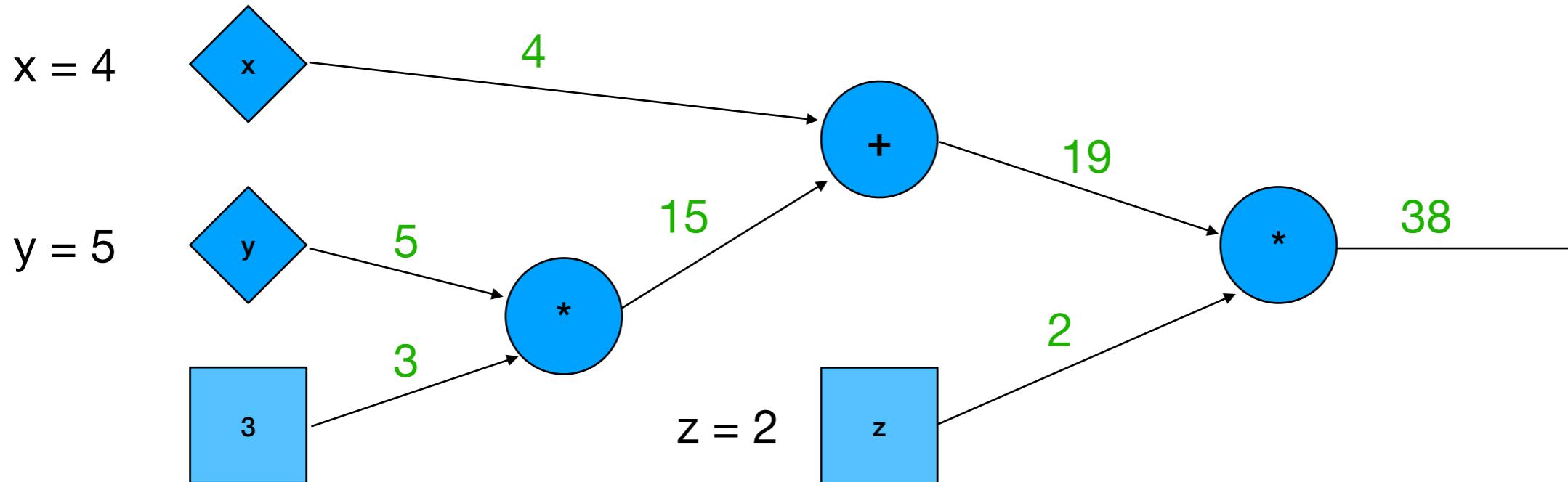
The Backwards Pass

The Backwards Pass



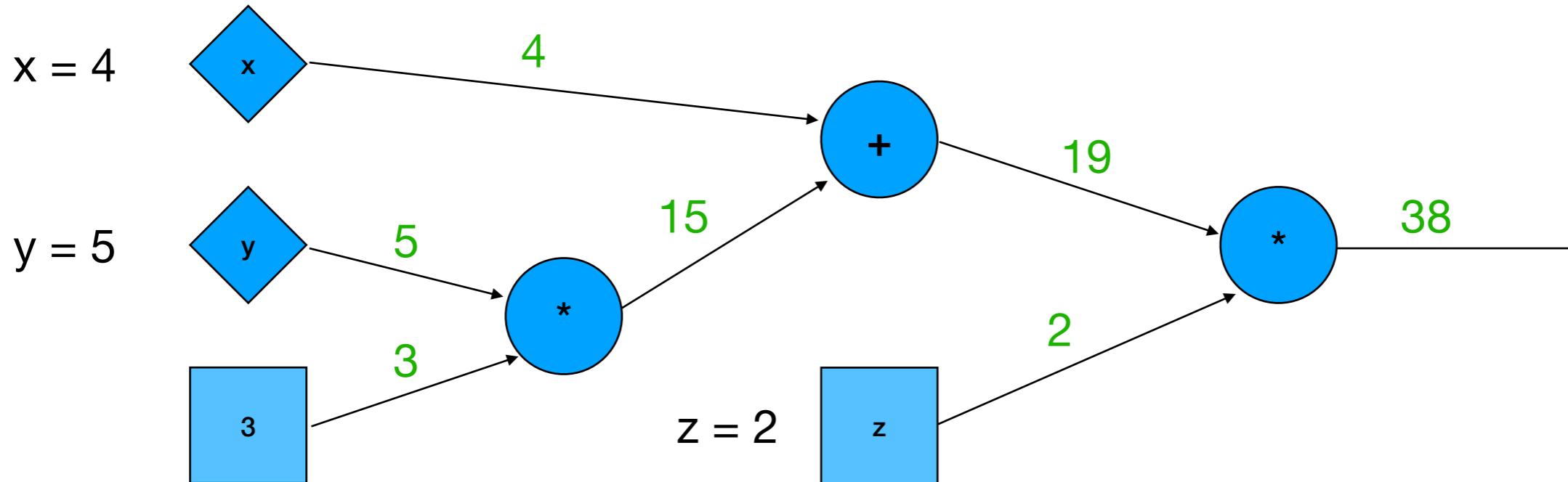
- We can diagram the chain rule out with a nice circuit diagram (i.e., a graph!).
- We'll denote modifiable parameters with a diamond, fixed parameters with a square, and operations with a circle.
- The above represents our $f(x,y,z) = z(x + 3y)$ function.

The Backwards Pass



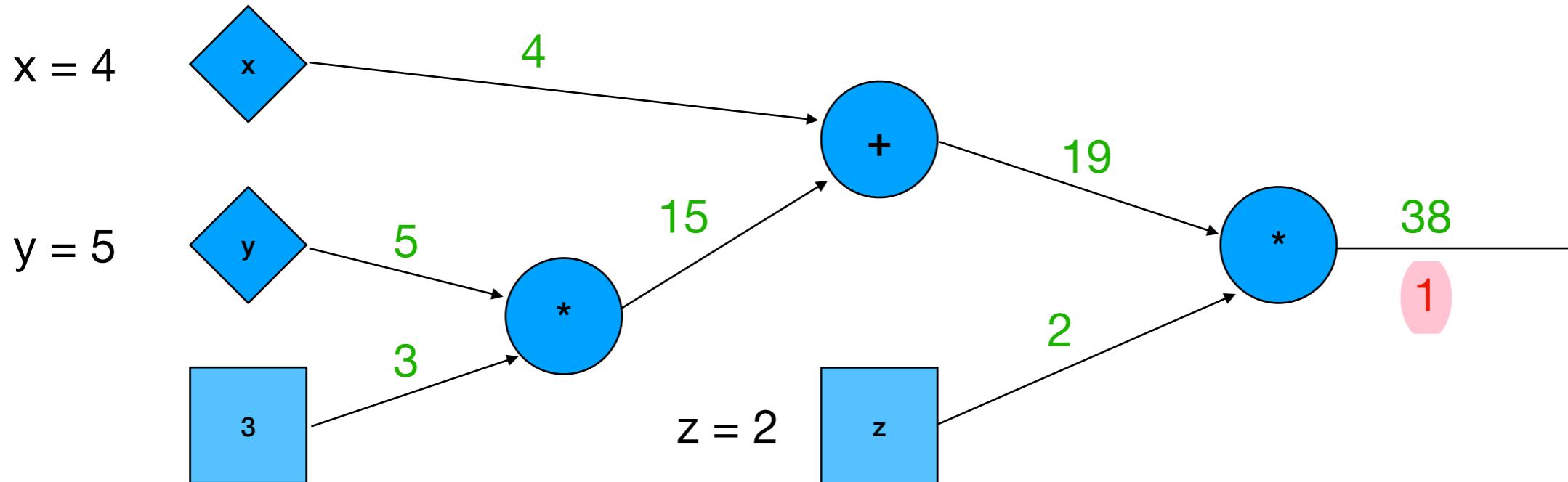
- Lets propagate values forward through the function. The forward pass will be above the lines in green.
- $f(4,5,2) = 2(4 * 3 * 5) = 38$
- Note this is already starting to look like a neural network!

The Backwards Pass



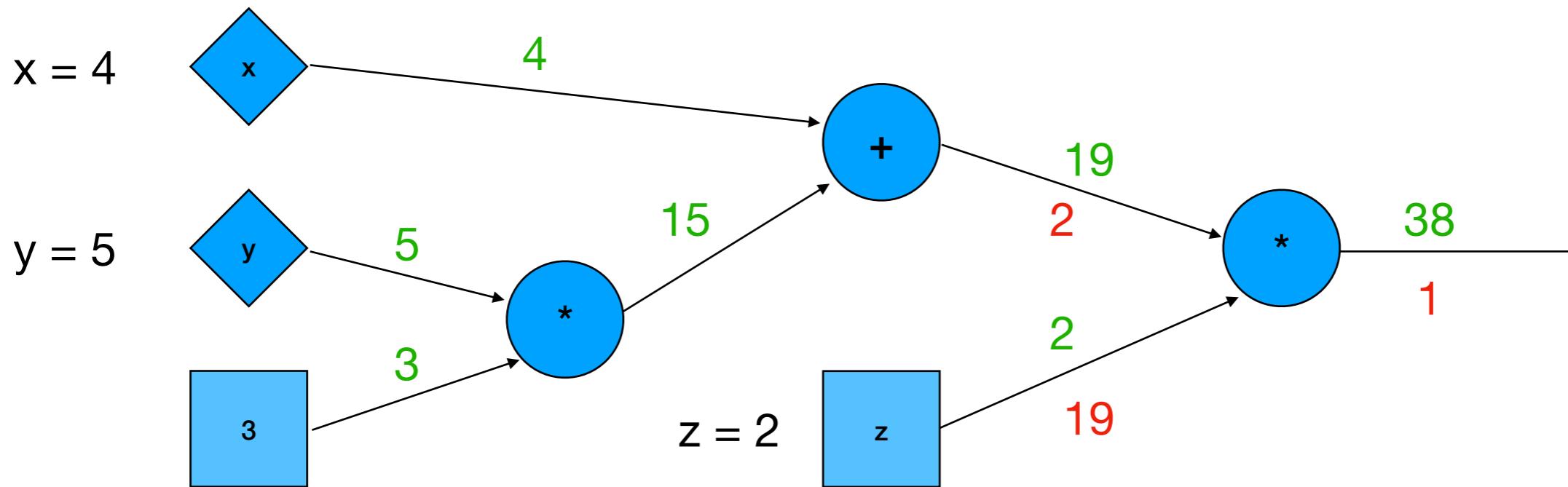
- Now we can do a backwards pass calculating the gradients (below the lines in red) working our way from right to left.

The Backwards Pass



- The gradient of the output is one (after the multiplication if that value is increased by 1 the output is (obviously) increased by 1).

The Backwards Pass

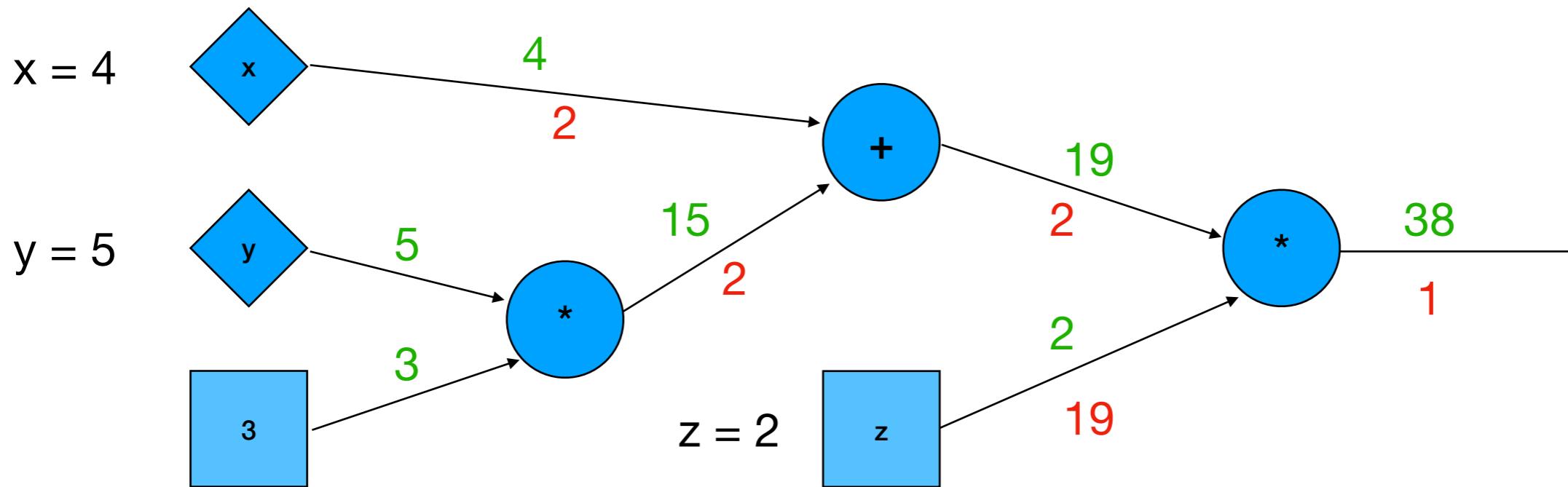


- Given that we have a multiplication, we can apply the multiplication rule on the graph:
 - If we add 1 to z , the **output** will be increased by 19 (the value coming out of the plus).
 - If we add 1 to the value coming out of the plus, the final output will be increased by 2.
- This is the same as the multiplication partial derivative rule:

$$f(x_1, x_2) = x_1 * x_2 \rightarrow \frac{\partial f}{\partial x_1} = x_2 \quad \frac{\partial f}{\partial x_2} = x_1$$

- In the case of multiplication, for any incoming connection we backpropagate the product of the other incoming connections multiplied by the outgoing connection.

The Backwards Pass

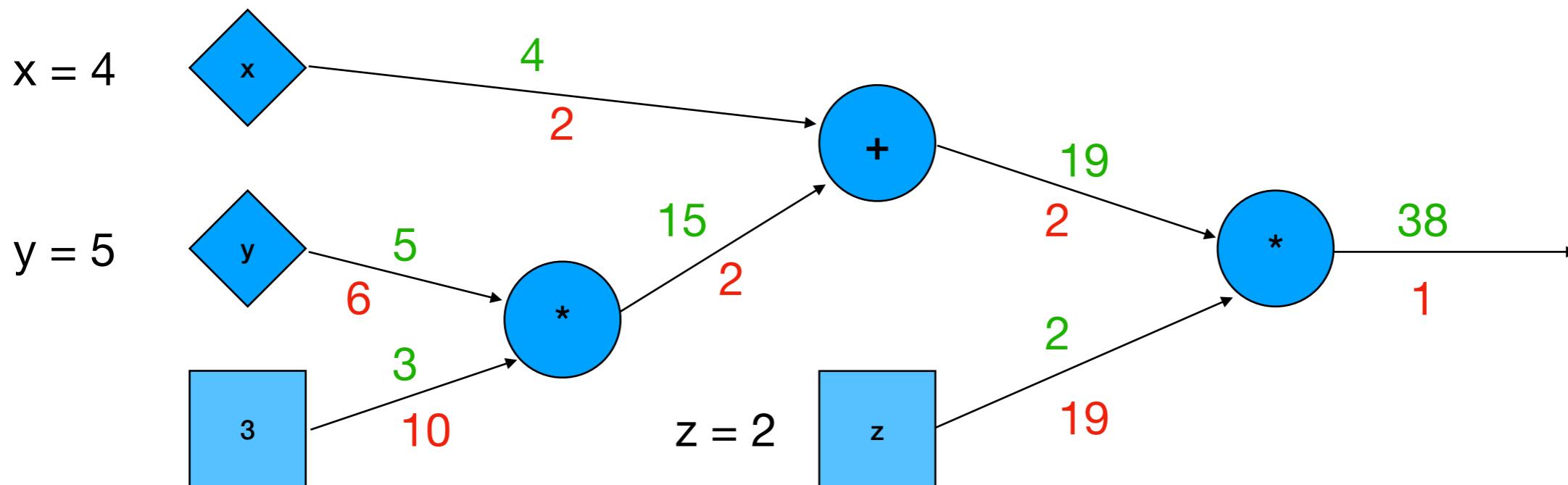


- Now we can propagate backwards through the addition:
 - If we increase the 4 to 5, it will be multiplied by the outgoing gradient (2) and the final output will be increased by 2.
 - Likewise, if we increase the 15 to 16, it will be multiplied by the outgoing gradient (2) and the final output will be increased by 2.
- This is combining the chain rule with the addition rule:

$$f(x_1, x_2) = x_1 + x_2 \rightarrow \frac{\partial f}{\partial x_1} = 1 \quad \frac{\partial f}{\partial x_2} = 1$$

- In the case of addition, we just propagate the incoming error backwards across all incoming edges.

The Backwards Pass

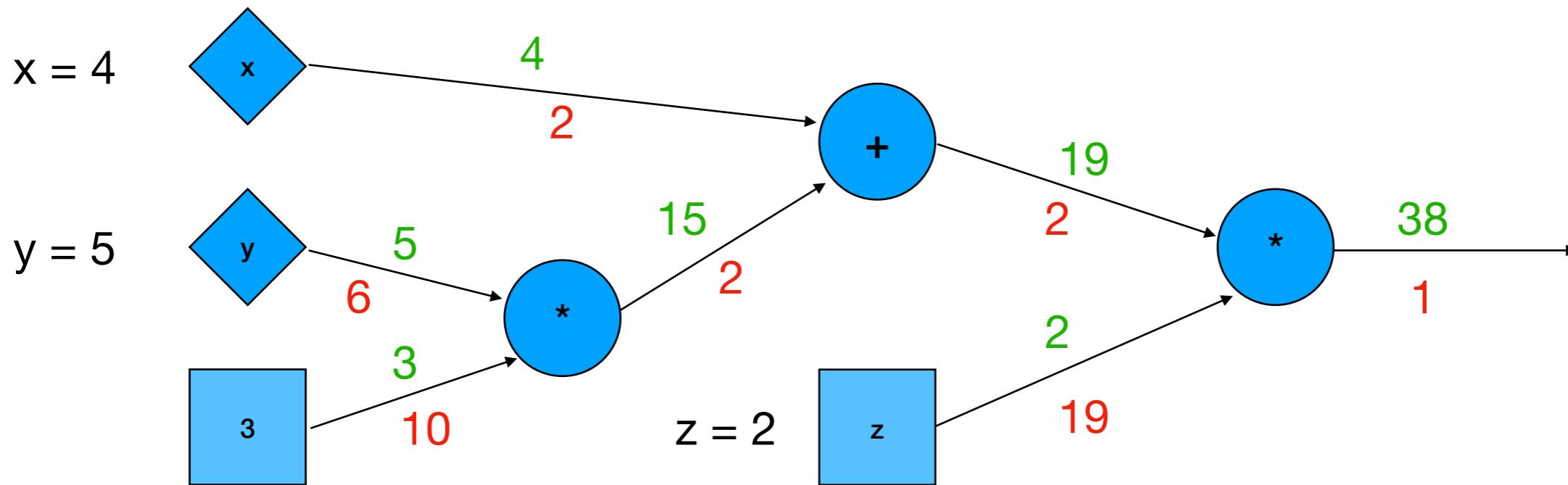


- Now we backpropagate by multiplication again:
 - If we could modify 3, increasing it by 1 would increase the final output by 10. $(y = 5) * 1 * (z = 2) = 10$
 - Increasing y by 1 will be multiplied by 3, then later by z, increasing the final output by 6.

$$f(x_1, x_2) = x_1 * x_2 \rightarrow \frac{\partial f}{\partial x_1} = x_2 \quad \frac{\partial f}{\partial x_2} = x_1$$

- In the case of multiplication, for any incoming connection we backpropagate the product of the other incoming connections multiplied by the outgoing connection.

The Backwards Pass

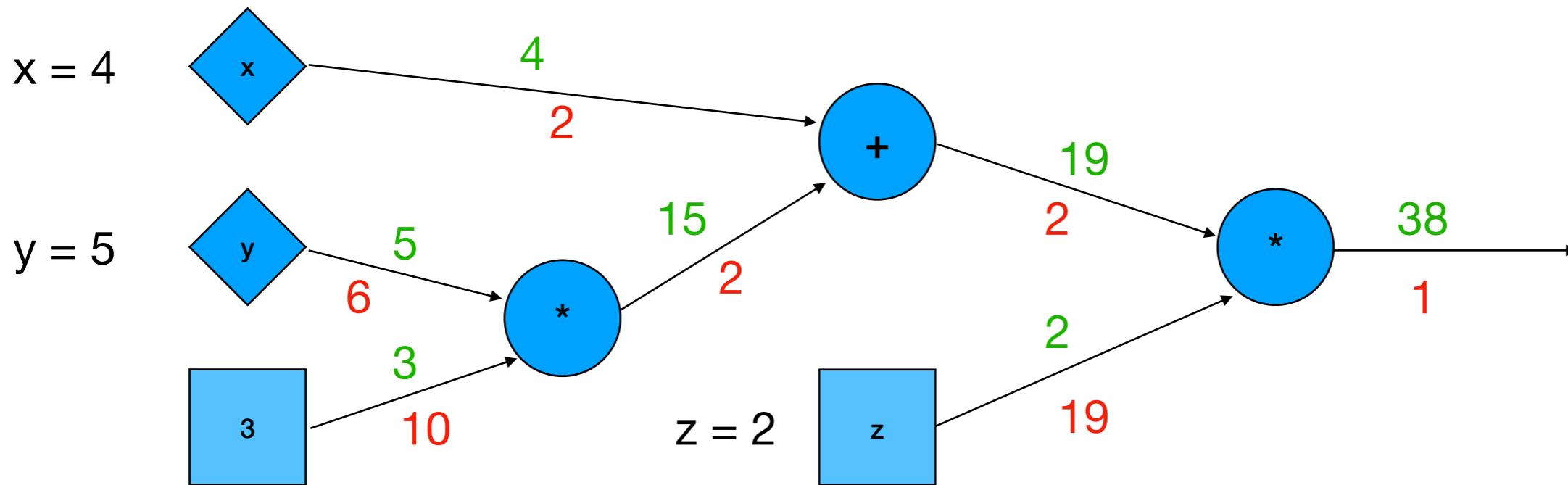


- Remember what we calculated before:

$$f(x, y, z) = z(x + 3y) \quad \frac{\partial f}{\partial x} = z \quad \frac{\partial f}{\partial y} = 3z$$

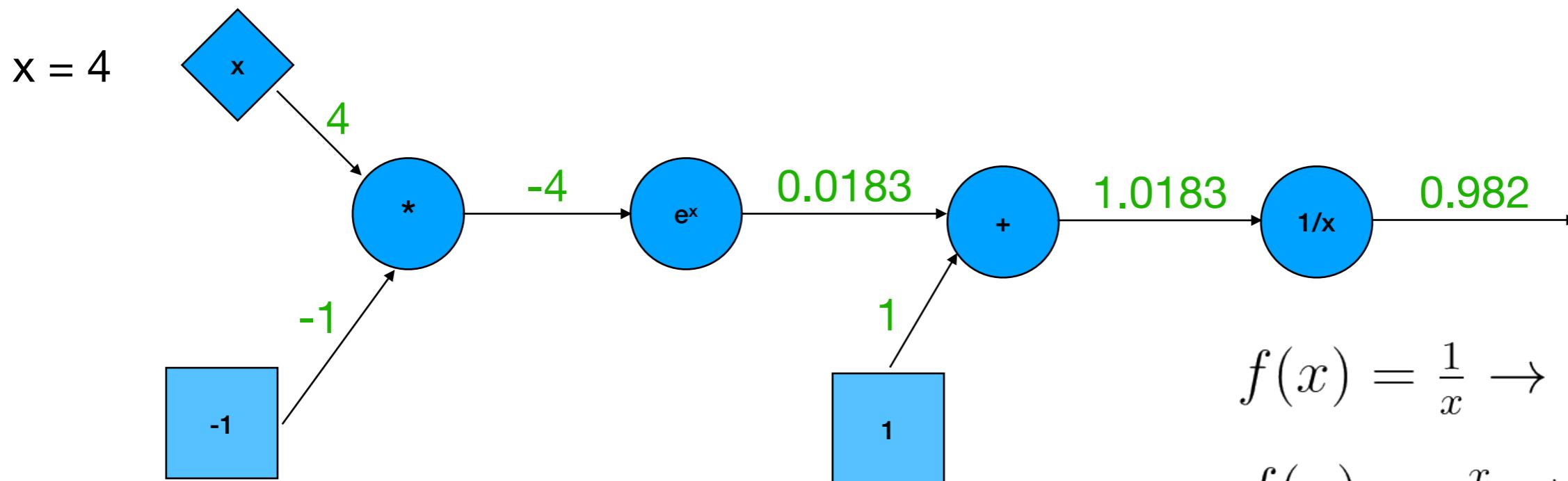
- Doing backpropagation we found the derivative of f with respect to x is 2, and the derivative of f with respect to y is 6.
- $z = 2$, so $df/dx = 2$ and $df/dy = 6$.
- We got the same result!

The Backwards Pass



- This approach is not only conceptually easy to understand but very powerful!
- If we draw out the network for any large function we can calculate all the partial derivatives for a given set of inputs just by doing a backward pass.
- In practice a backwards pass is computationally about the same cost as 2 forward passes -- which is even faster than approximating the gradient with finite differences. It is a constant 2 cost vs. a linearly scaling $2n$ cost (where n is the number of parameters).
- Now we are just missing how we do this for the activation functions.

The Backwards Pass: Activation Functions



$$f(x) = \frac{1}{1+e^{-x}}$$

$$f(x) = \frac{1}{x} \rightarrow \frac{\partial f}{\partial x} = \frac{-1}{x^2}$$

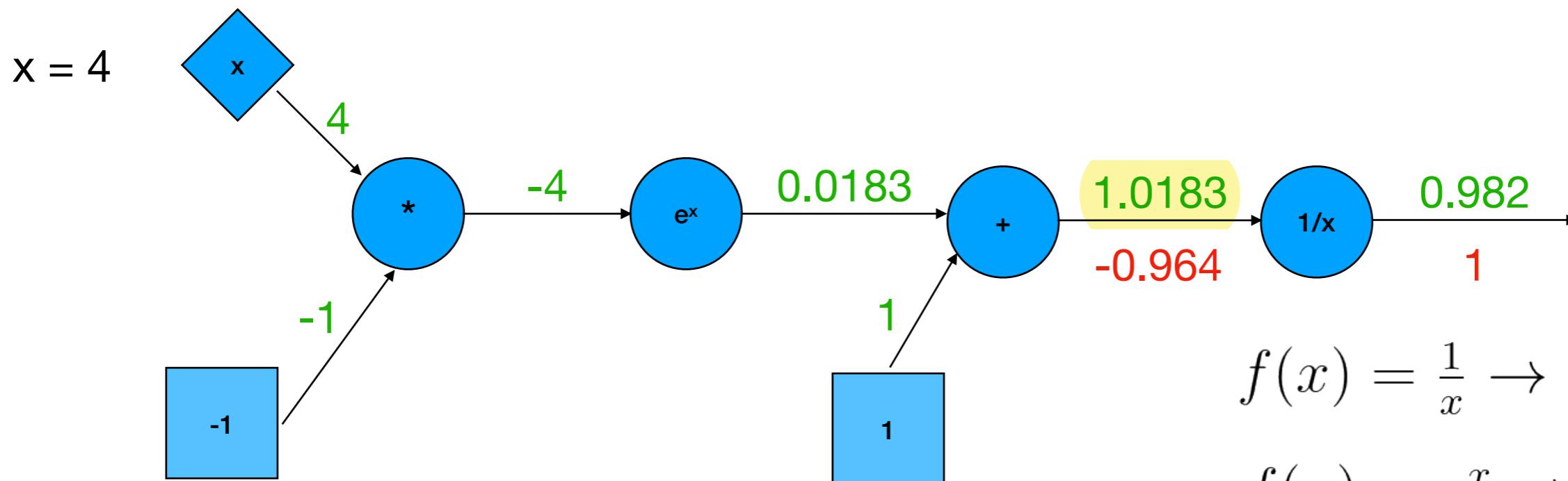
$$f(x) = e^x \rightarrow \frac{\partial f}{\partial x} = e^x$$

$$f_a(x) = ax \rightarrow \frac{\partial f}{\partial x} = a$$

$$f_c(x) = c + x \rightarrow \frac{\partial f}{\partial x} = 1$$

- Lets do this for sigmoid (above left).
- We have some helpful derivatives above on the right.

The Backwards Pass: Activation Functions



$$f(x) = \frac{1}{1+e^{-x}}$$

$$f(x) = \frac{1}{x} \rightarrow \frac{\partial f}{\partial x} = \frac{-1}{x^2}$$

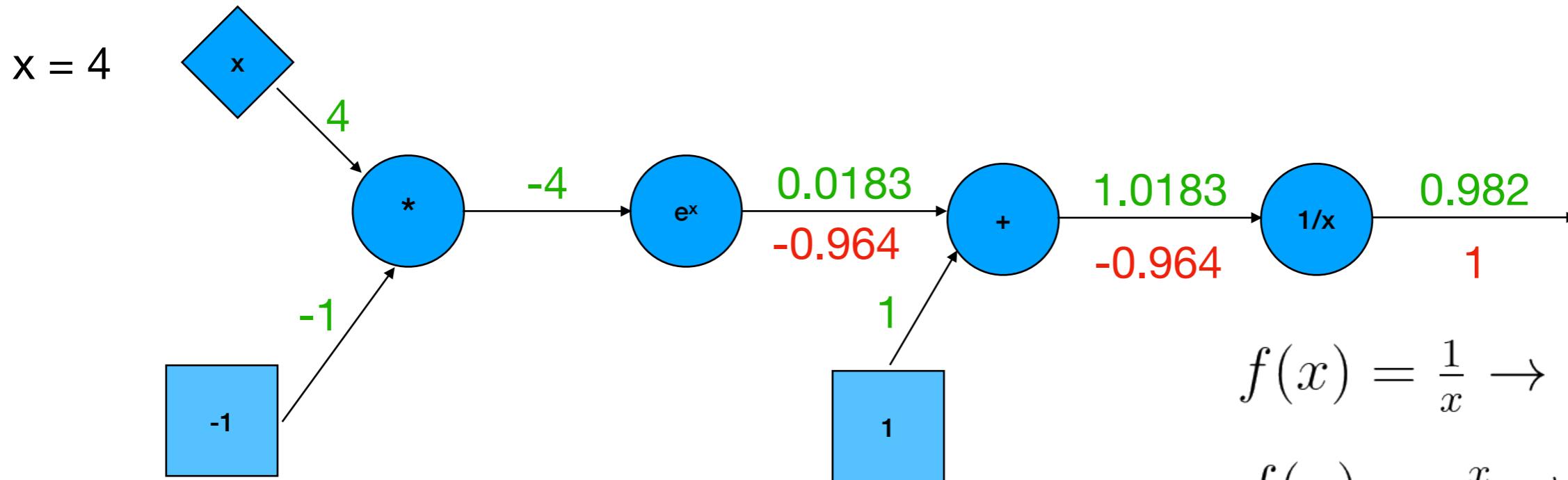
$$f(x) = e^x \rightarrow \frac{\partial f}{\partial x} = e^x$$

$$f_a(x) = ax \rightarrow \frac{\partial f}{\partial x} = a$$

$$f_c(x) = c + x \rightarrow \frac{\partial f}{\partial x} = 1$$

- The derivative of $1/x$ is $-1/x^2$
- We know the incoming x is 1.0183
- $-1/(1.0183^2) = -0.964..$
- We multiply this by the outgoing gradient (1) to get the incoming gradient.

The Backwards Pass: Activation Functions



$$f(x) = \frac{1}{1+e^{-x}}$$

$$f(x) = \frac{1}{x} \rightarrow \frac{\partial f}{\partial x} = \frac{-1}{x^2}$$

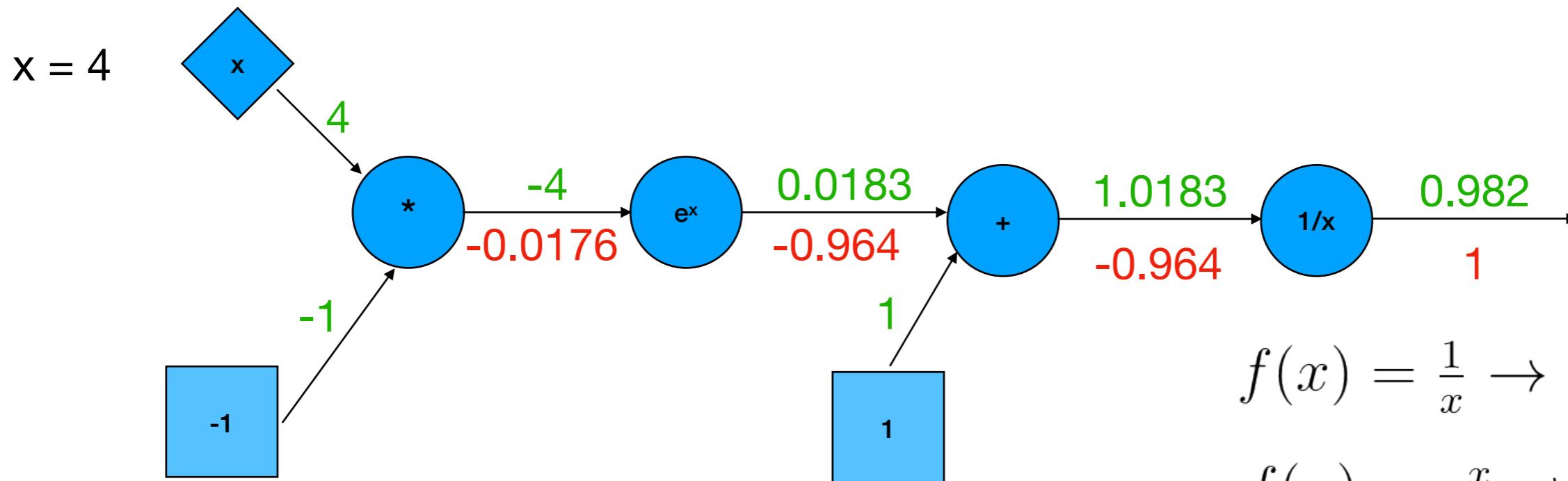
$$f(x) = e^x \rightarrow \frac{\partial f}{\partial x} = e^x$$

$$f_a(x) = ax \rightarrow \frac{\partial f}{\partial x} = a$$

$$f_c(x) = c + x \rightarrow \frac{\partial f}{\partial x} = 1$$

- We can do the plus rule, the partial gradient is 1 and we multiply it by the outgoing gradient because of the chain rule.

The Backwards Pass: Activation Functions



$$f(x) = \frac{1}{1+e^{-x}}$$

$$f(x) = \frac{1}{x} \rightarrow \frac{\partial f}{\partial x} = \frac{-1}{x^2}$$

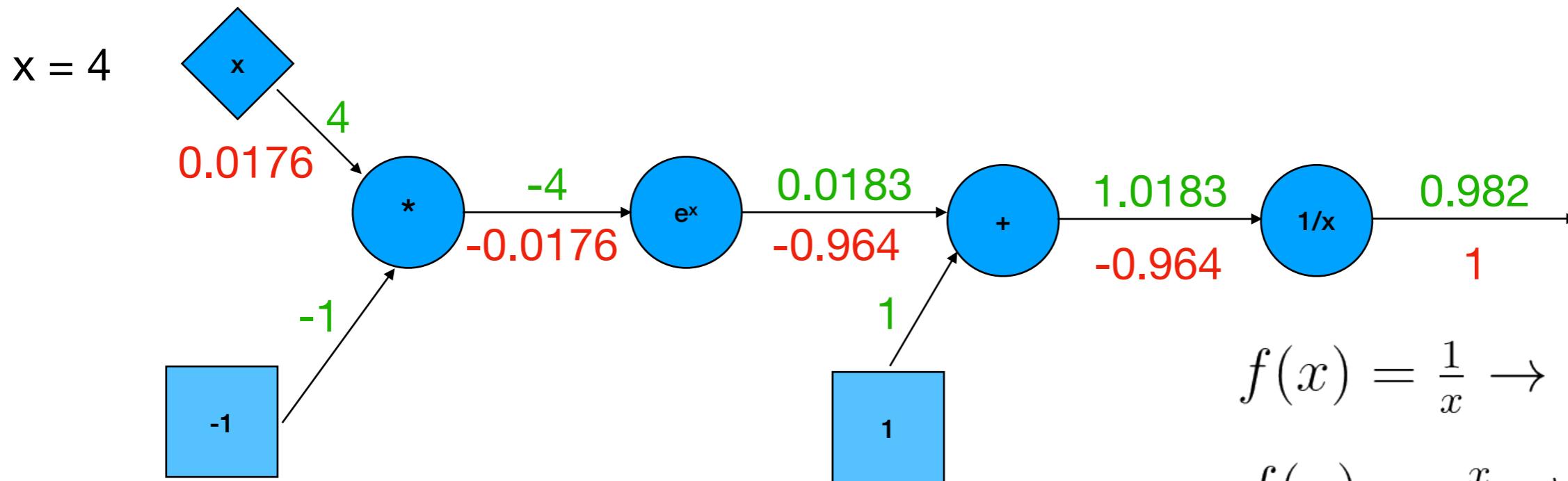
$$f(x) = e^x \rightarrow \frac{\partial f}{\partial x} = e^x$$

$$f_a(x) = ax \rightarrow \frac{\partial f}{\partial x} = a$$

$$f_c(x) = c + x \rightarrow \frac{\partial f}{\partial x} = 1$$

- The derivative of e^x is e^x .
- e^{-4} is 0.0183 so we multiply this by the outgoing gradient because of the chain rule = -0.0176

The Backwards Pass: Activation Functions



$$f(x) = \frac{1}{1+e^{-x}}$$

$$f(x) = \frac{1}{x} \rightarrow \frac{\partial f}{\partial x} = \frac{-1}{x^2}$$

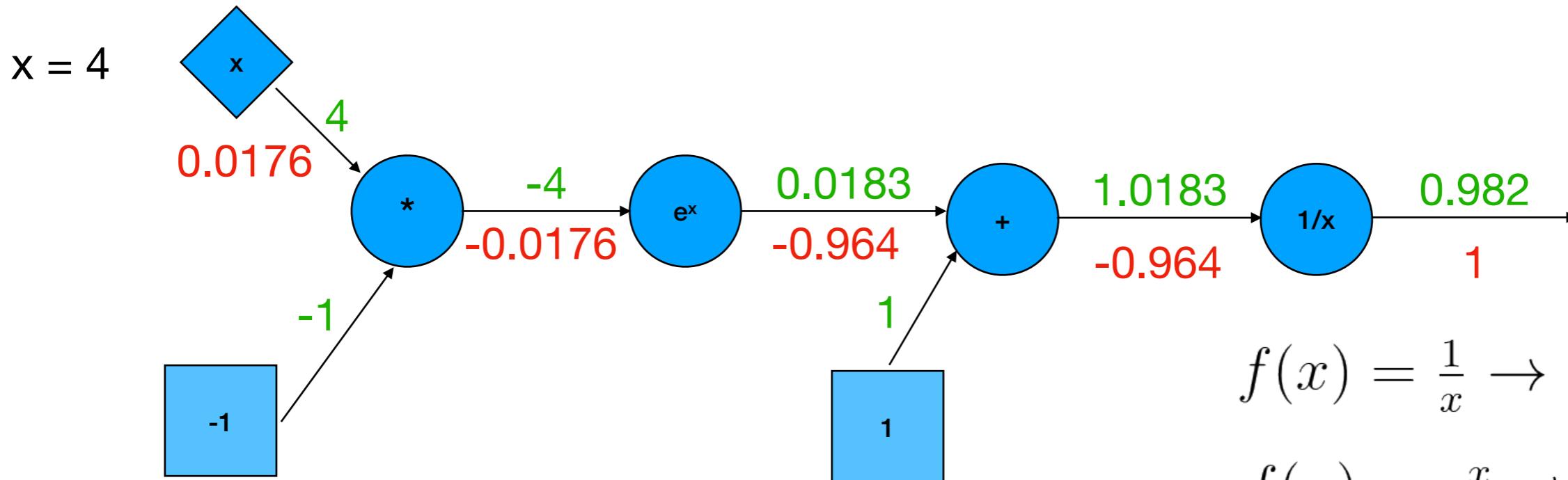
$$f(x) = e^x \rightarrow \frac{\partial f}{\partial x} = e^x$$

$$f_a(x) = ax \rightarrow \frac{\partial f}{\partial x} = a$$

$$f_c(x) = c + x \rightarrow \frac{\partial f}{\partial x} = 1$$

- We can now do the multiplication rule again, we multiply the outgoing gradient by the other incoming value (-1).
- Now we have the gradient of sigmoid with respect to x when $x = 4$.

The Backwards Pass: Activation Functions



$$f(x) = \frac{1}{1+e^{-x}}$$

$$f(x) = \frac{1}{x} \rightarrow \frac{\partial f}{\partial x} = \frac{-1}{x^2}$$

$$f(x) = e^x \rightarrow \frac{\partial f}{\partial x} = e^x$$

$$f_a(x) = ax \rightarrow \frac{\partial f}{\partial x} = a$$

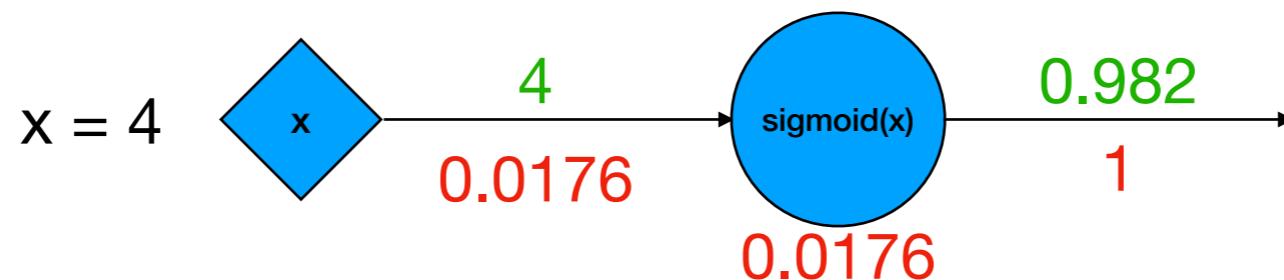
$$f_c(x) = c + x \rightarrow \frac{\partial f}{\partial x} = 1$$

- Lets check our math, we know the derivative of sigmoid is:

$$\sigma(x) = \frac{1}{1+e^{-x}} \quad \frac{\partial \sigma}{\partial x} = \sigma(x)(1 - \sigma(x))$$

- $0.982 * (1 - 0.982) = 0.017676$. So we were right (I removed some significant figures for the slides, sigmoid(4) is actually 0.98201).

The Backward Pass: Activation Function

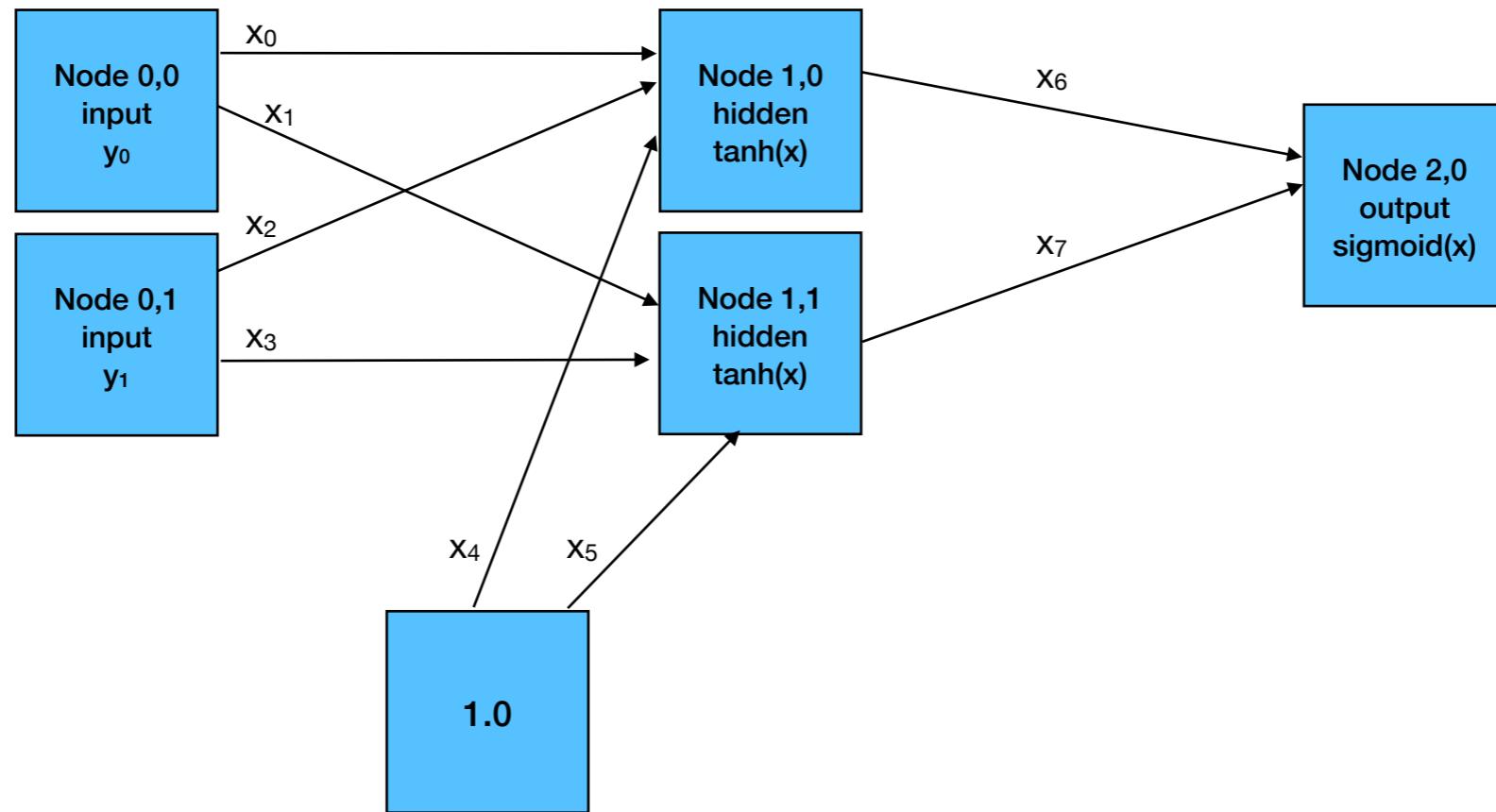


$$\sigma(x) = \frac{1}{1+e^{-x}} \quad \frac{\partial \sigma}{\partial x} = \sigma(x)(1 - \sigma(x))$$

- So in actuality we can simplify a sigmoid activation function in our forward/backward pass network (or any other activation function that we know the derivative for).
- In the forward pass, we know x so we can calculate the sigmoid output.
- We also know the derivative with respect to x for sigmoid = $\text{sigmoid}(x) * (1 - \text{sigmoid}(x))$.
- All we need to do is calculate the gradient for x when we calculate the sigmoid on the forward pass so we can use it to multiply the outgoing gradient by in the backwards pass. We can do this for any differentiable function.

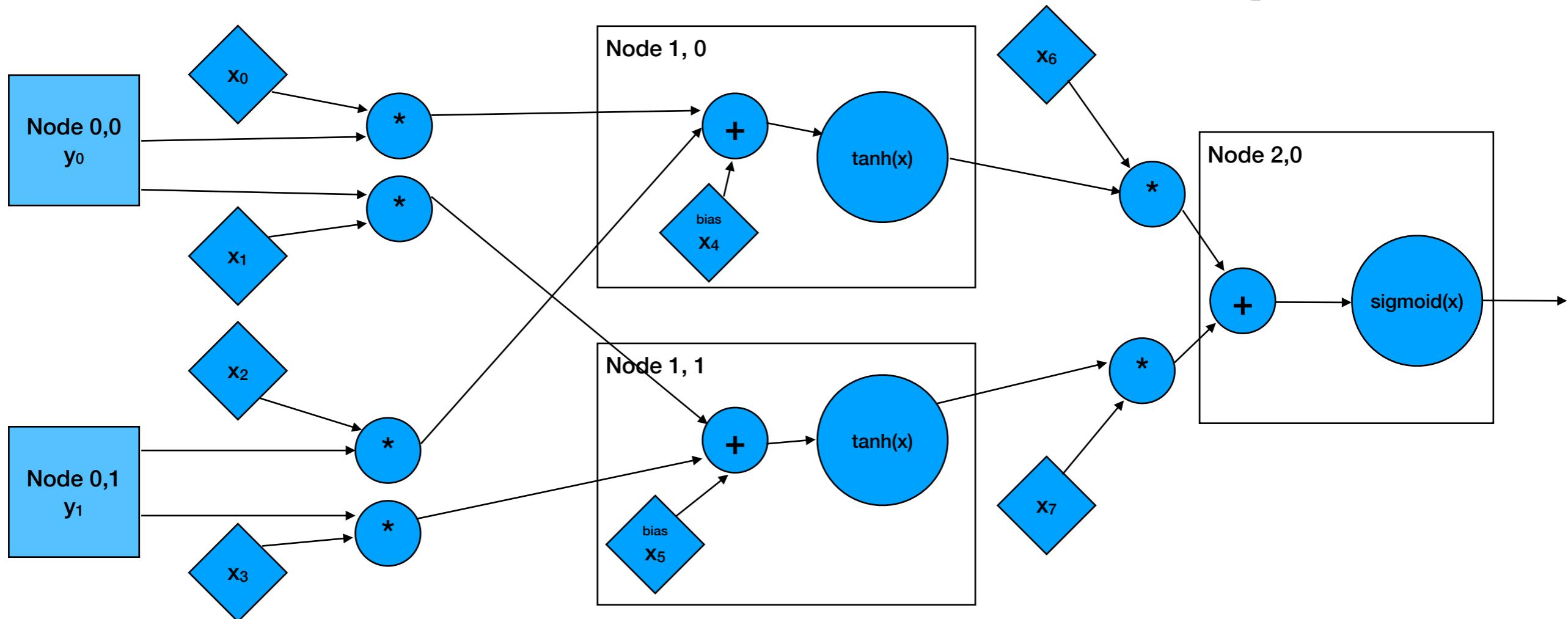
An Example

The Backward Pass: An Example



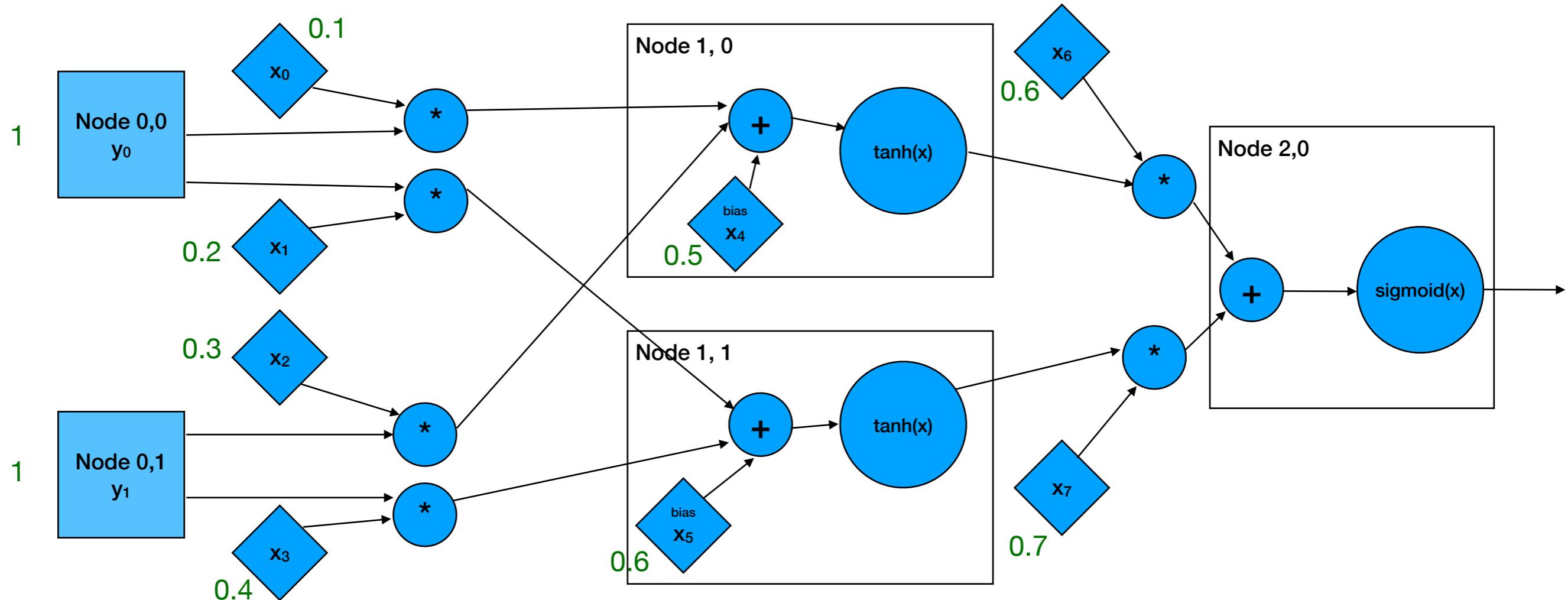
- Lets look at a simple network which we could use to calculate xor.
- The hidden nodes are tanh activation functions and the output node is a sigmoid function

The Backward Pass: An Example



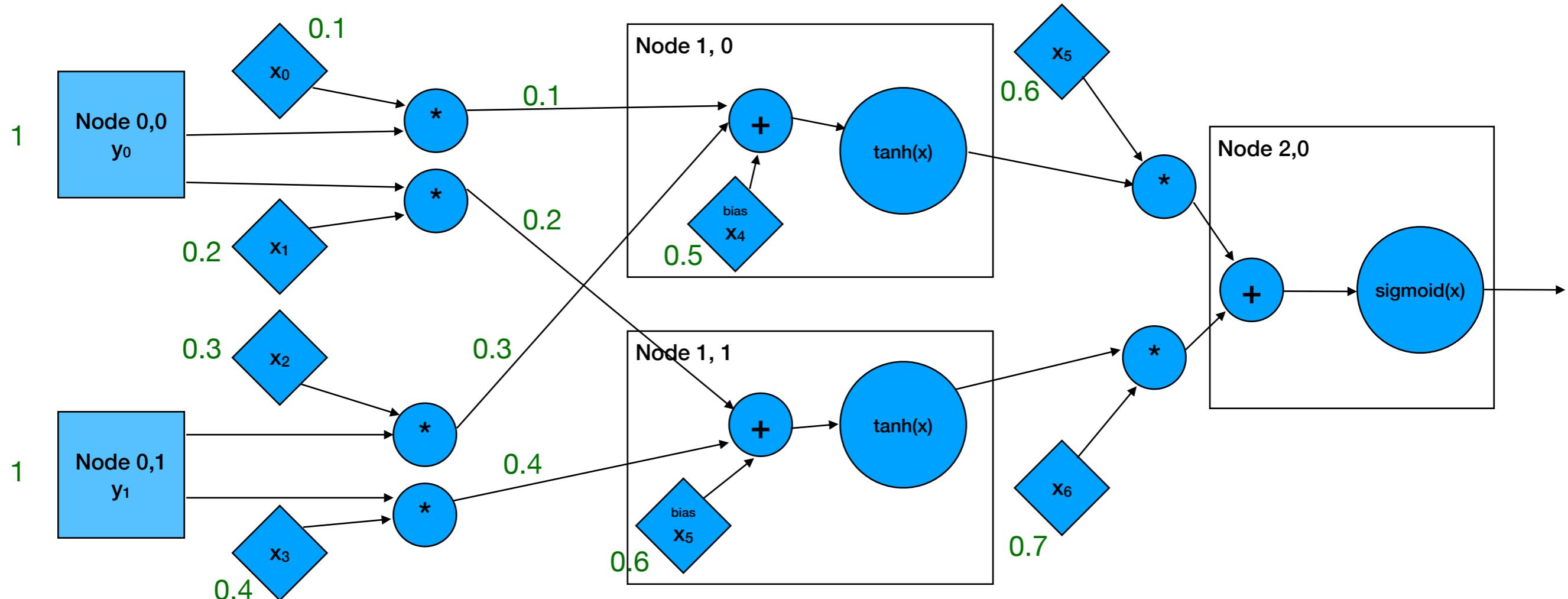
- We can expand this network out to a similar circuit diagram as before to do back propagation on it.

The Backward Pass: An Example



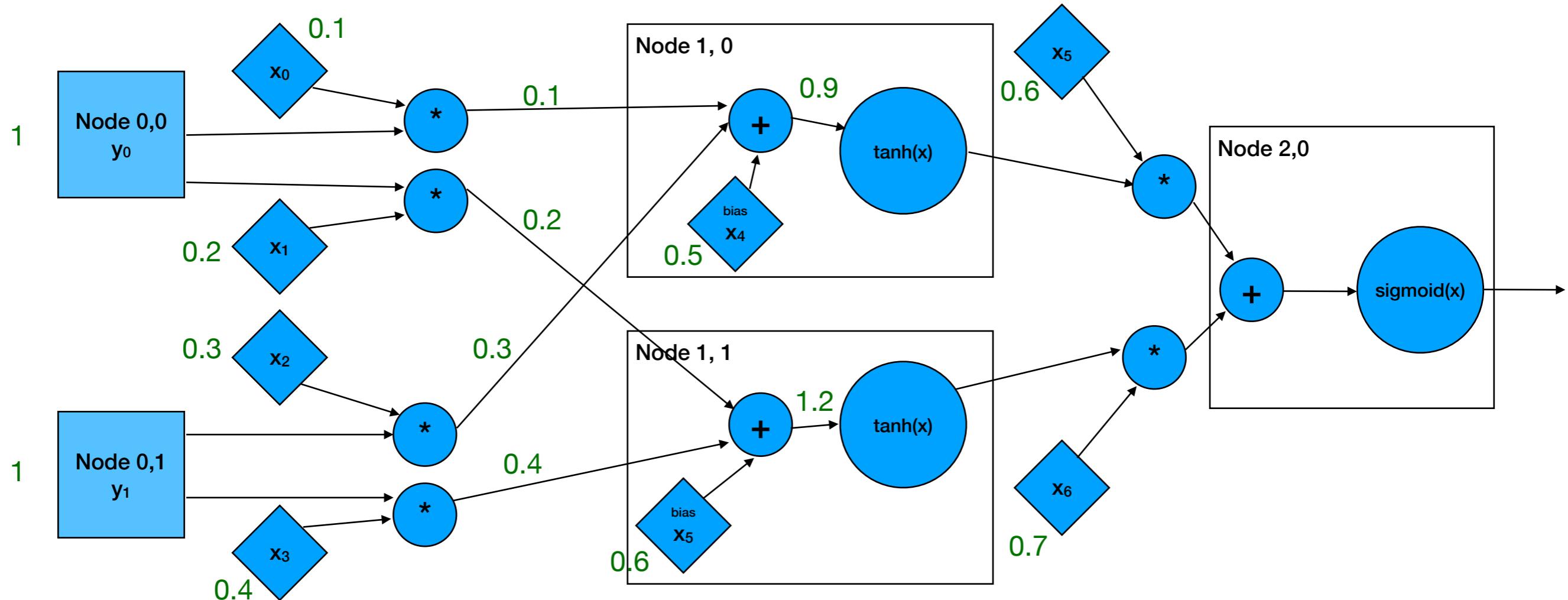
- Lets use $y_0 = 1, y_1 = 1, x_0 = 0.1, x_1 = 0.2, x_2 = 0.3, x_3 = 0.4, x_4=0.5, x_5=0.6, x_6 = 0.6, x_7=0.7$.

The Backward Pass: An Example



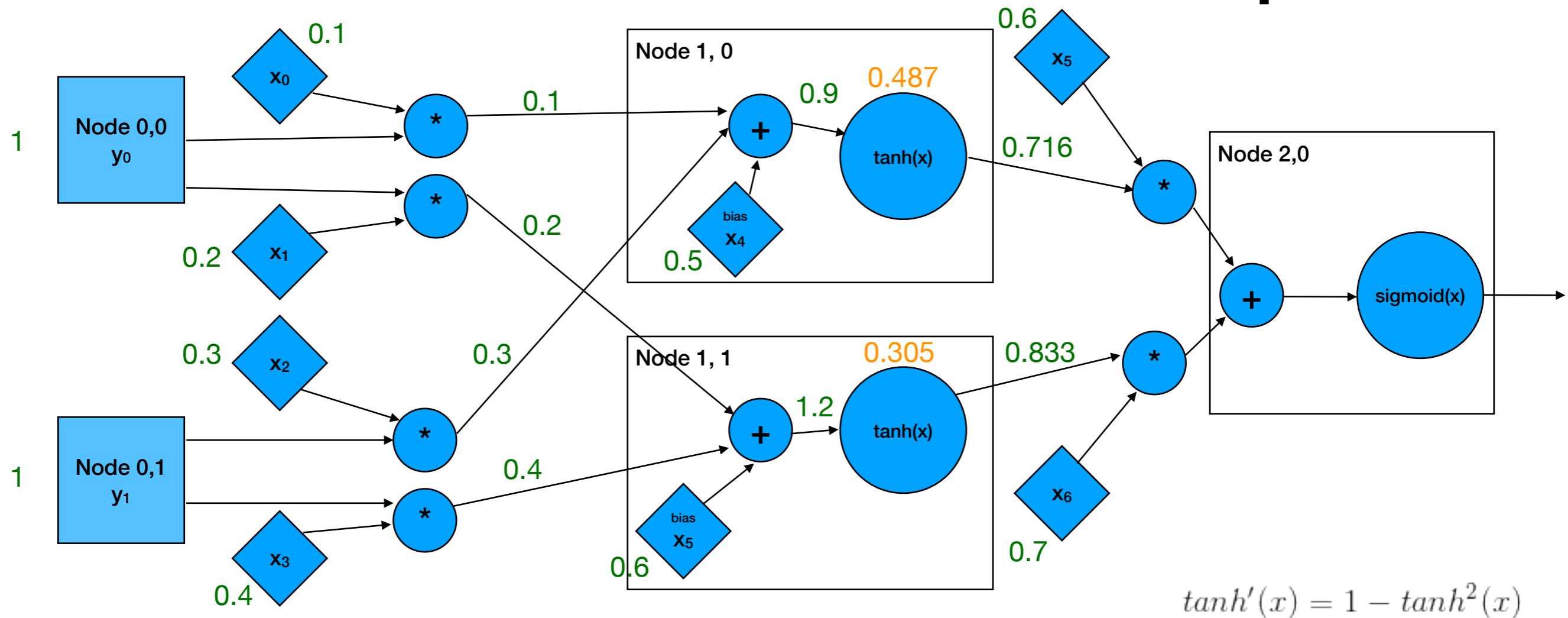
- Now we can forward pass the first layer.

The Backward Pass: An Example



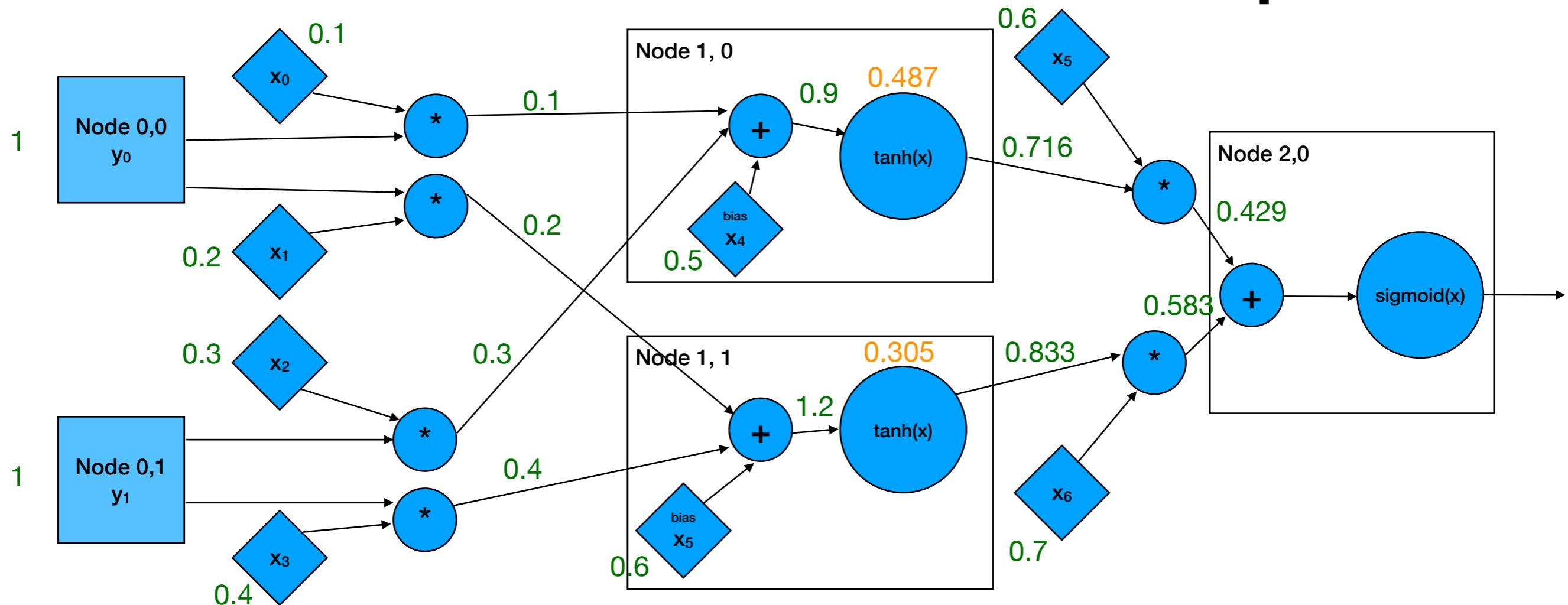
- Sum things up in the hidden layer nodes.

The Backward Pass: An Example



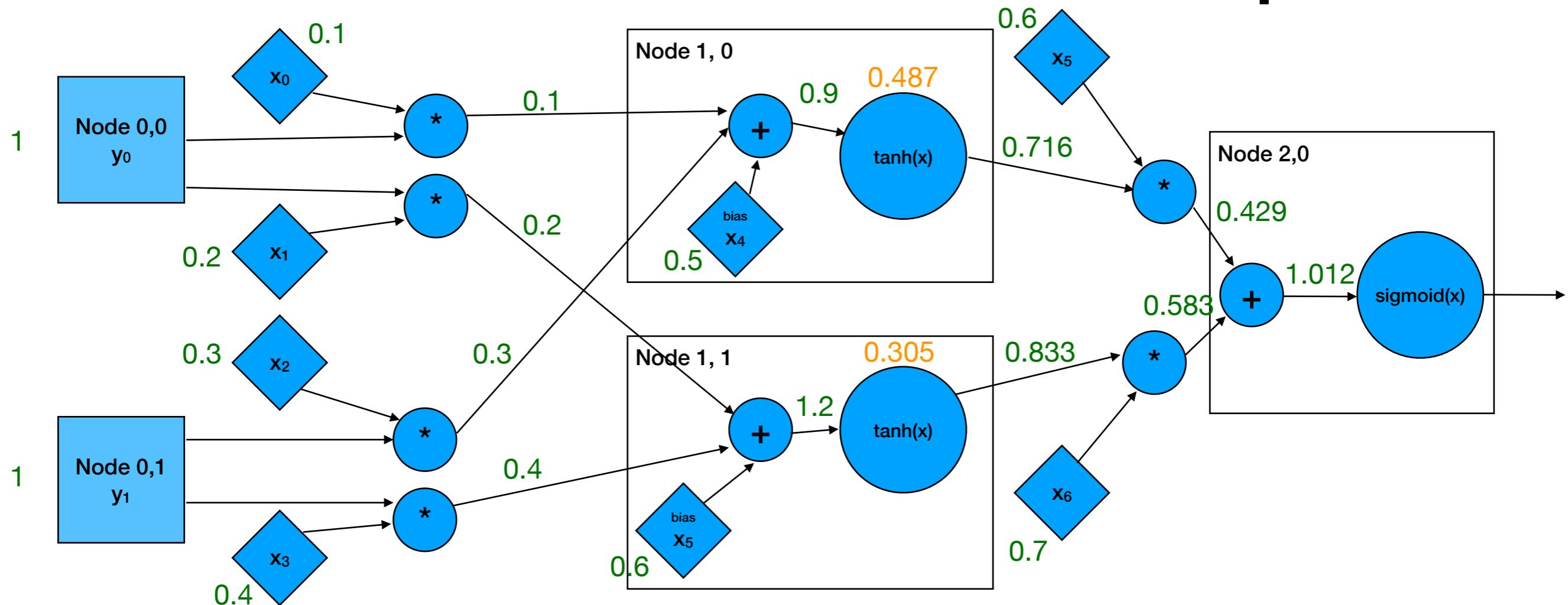
- Apply the activation functions.
- While we're propagating the data through the activation function we can calculate their derivatives so we can use them later for the backward pass.
- $1 - \tanh^2(0.9) = 0.487$ and $1 - \tanh^2(1.2) = 0.305$. We'll write these in orange and save them for later.

The Backward Pass: An Example



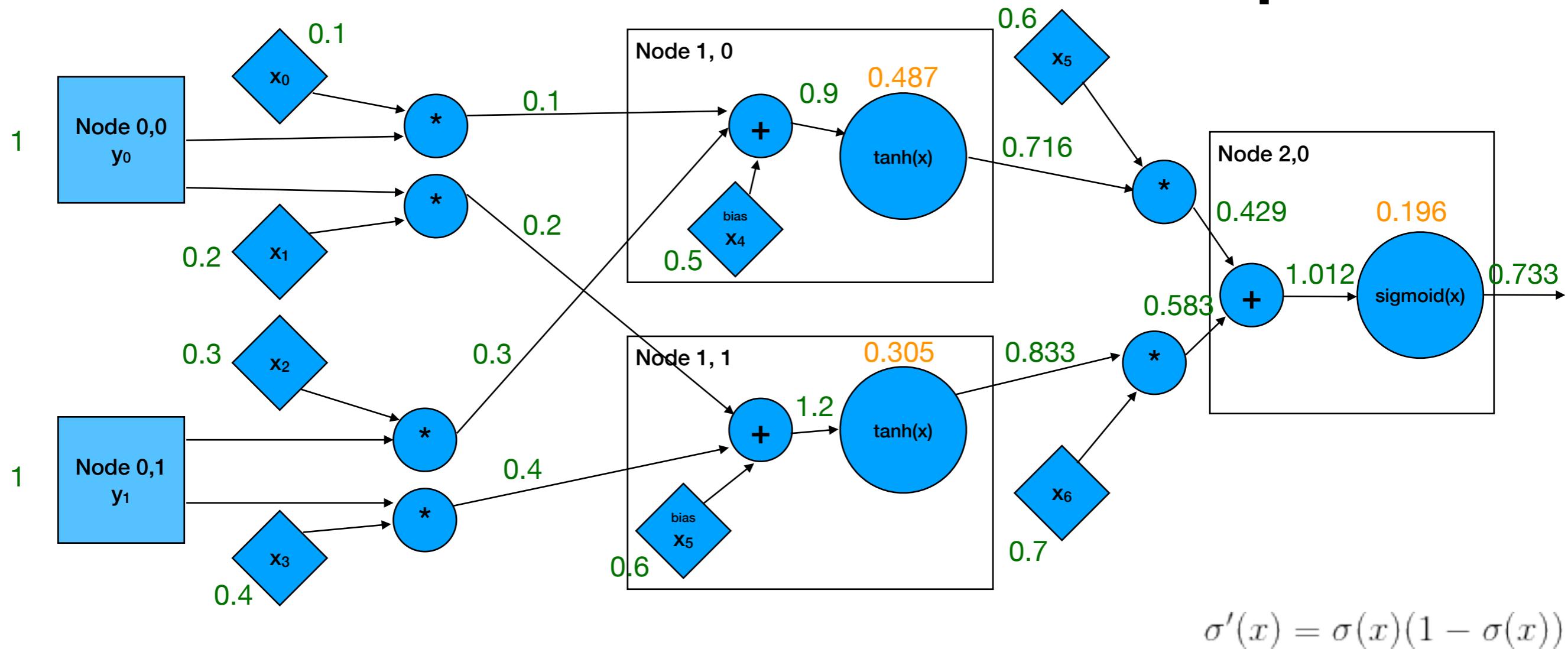
- Now the next level of weights.

The Backward Pass: An Example



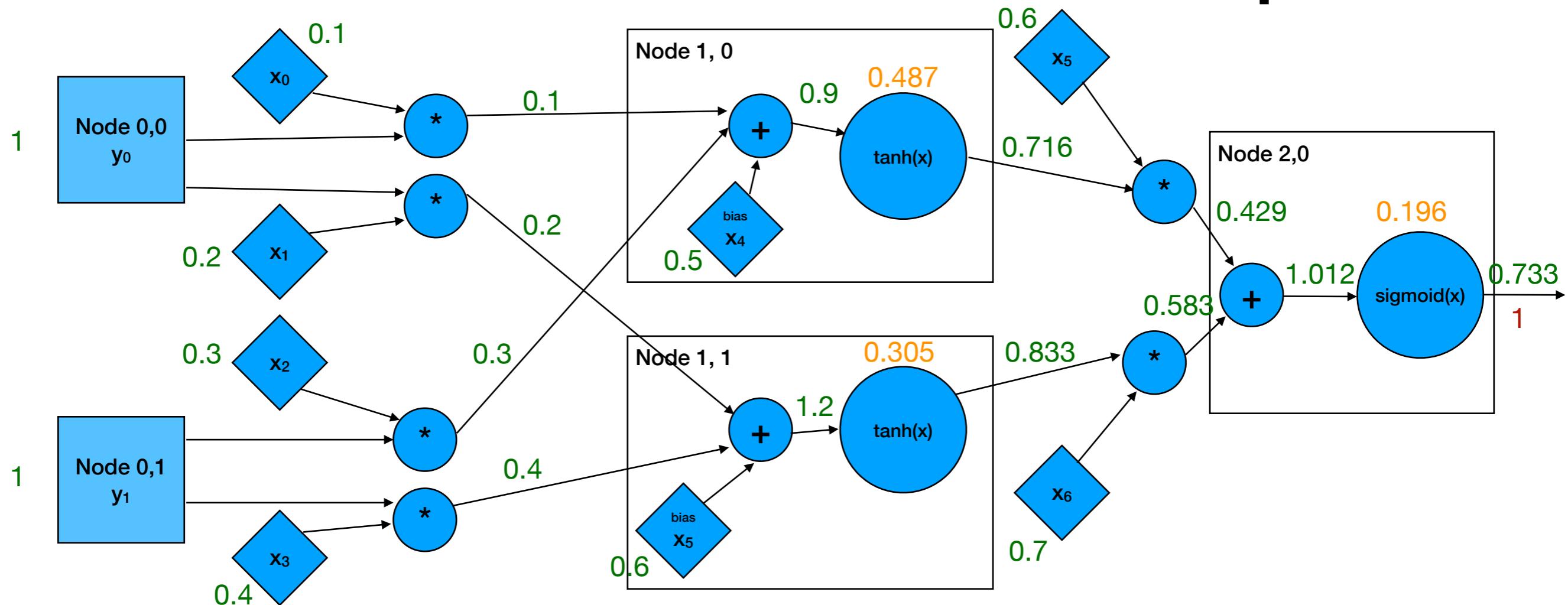
- Sum up at the output node.

The Backward Pass: An Example



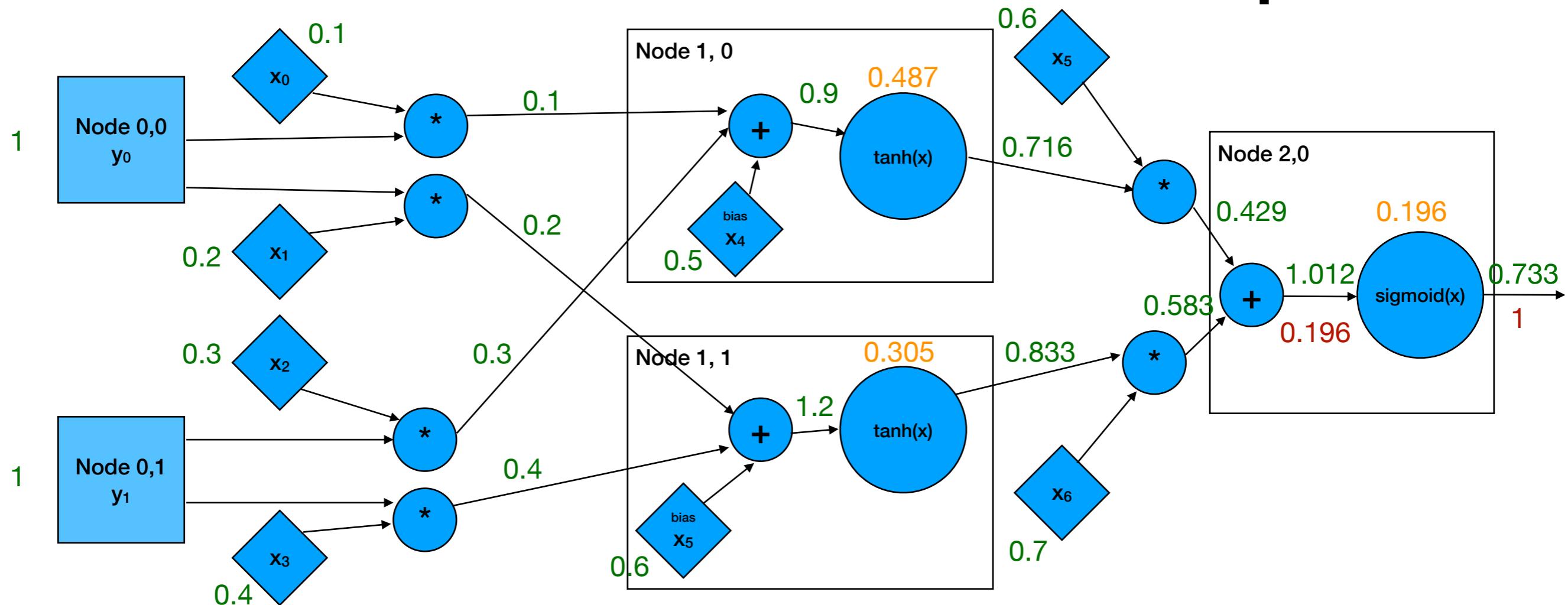
- Apply the sigmoid activation function.
- Again lets calculate the gradient to save for the backwards pass.
- $0.733 * (1 - 0.733) = 0.196$

The Backward Pass: An Example



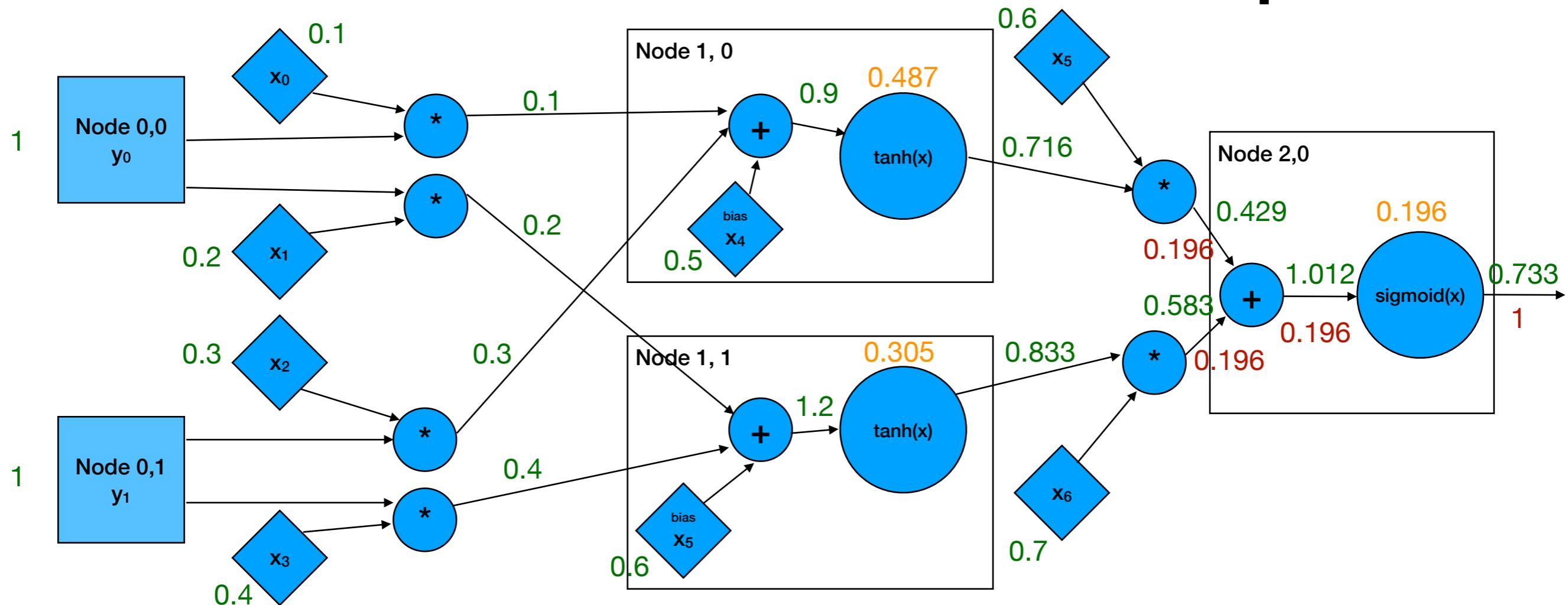
- Now that the forward pass is complete we can do the backward pass.
- Start with a 1 for the incoming gradient (we'll call this a *delta* which is the common term for it in backpropagation).

The Backward Pass: An Example



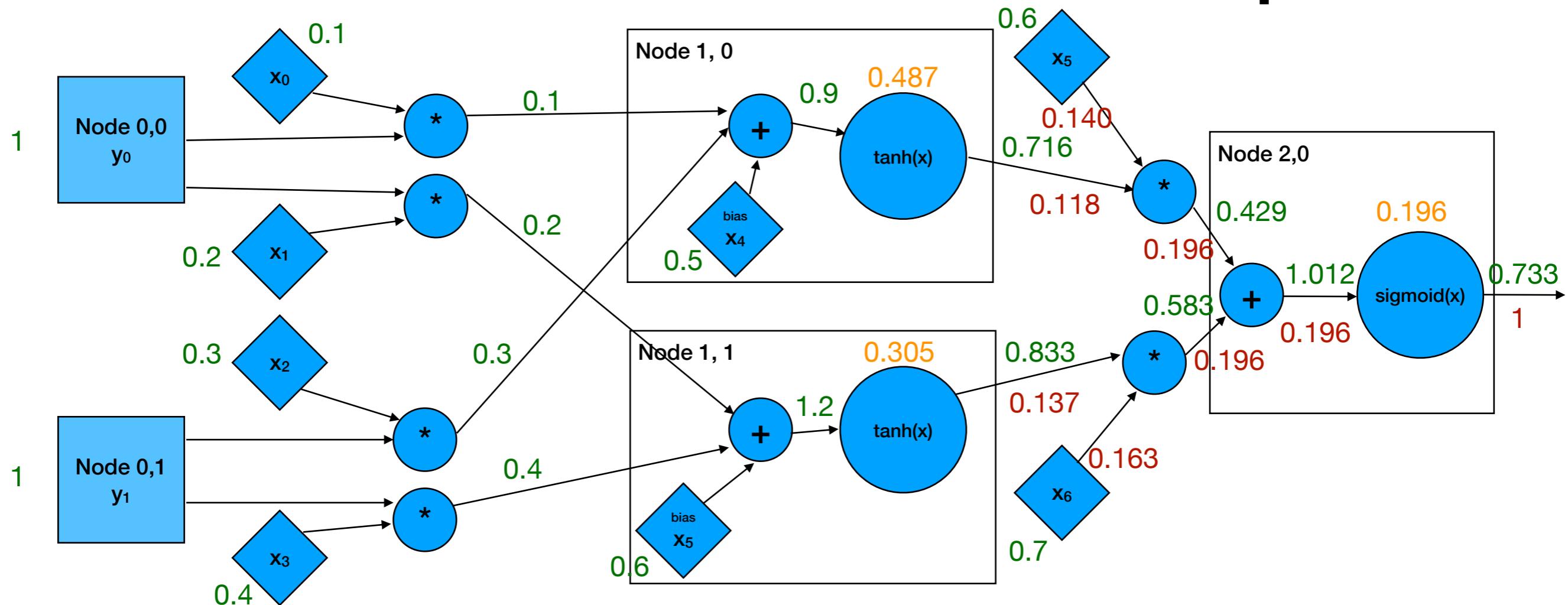
- To backpropagate through the sigmoid activation function we multiply it by the derivative of sigmoid we calculated in the forward pass.

The Backward Pass: An Example



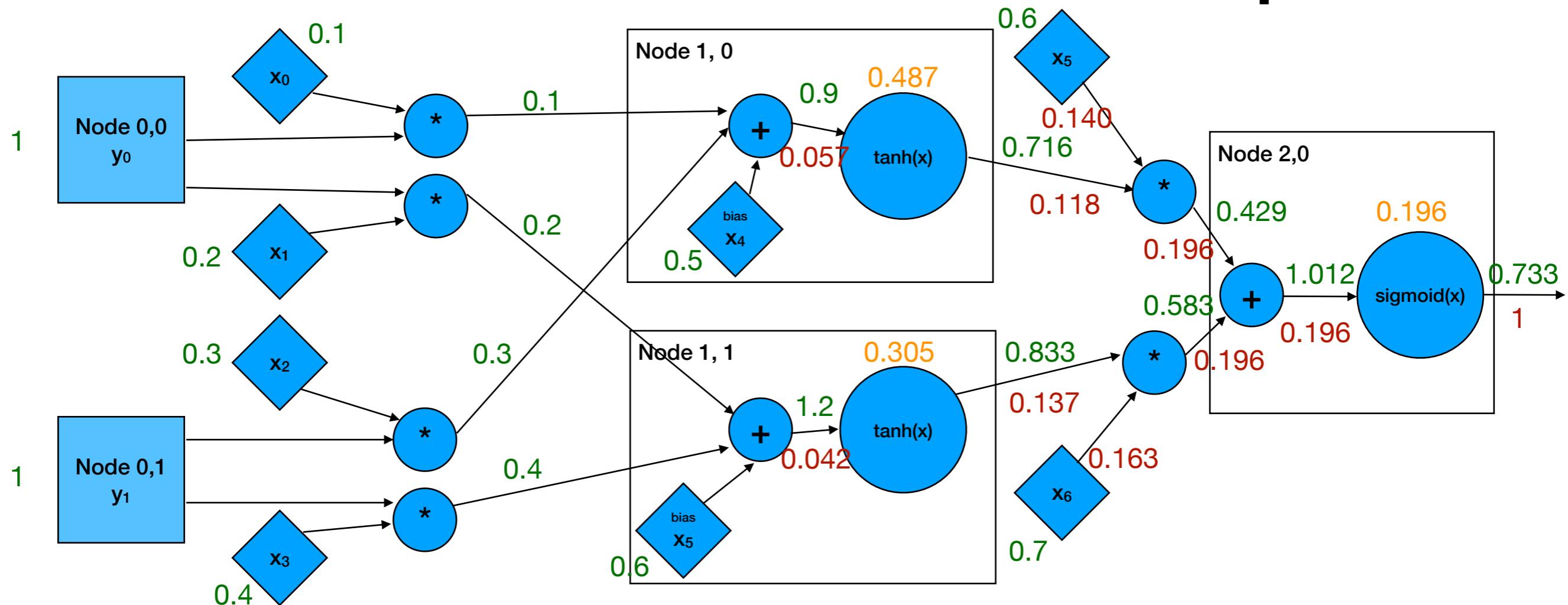
- We use the addition rule to pass the delta back on both incoming connections.

The Backward Pass: An Example



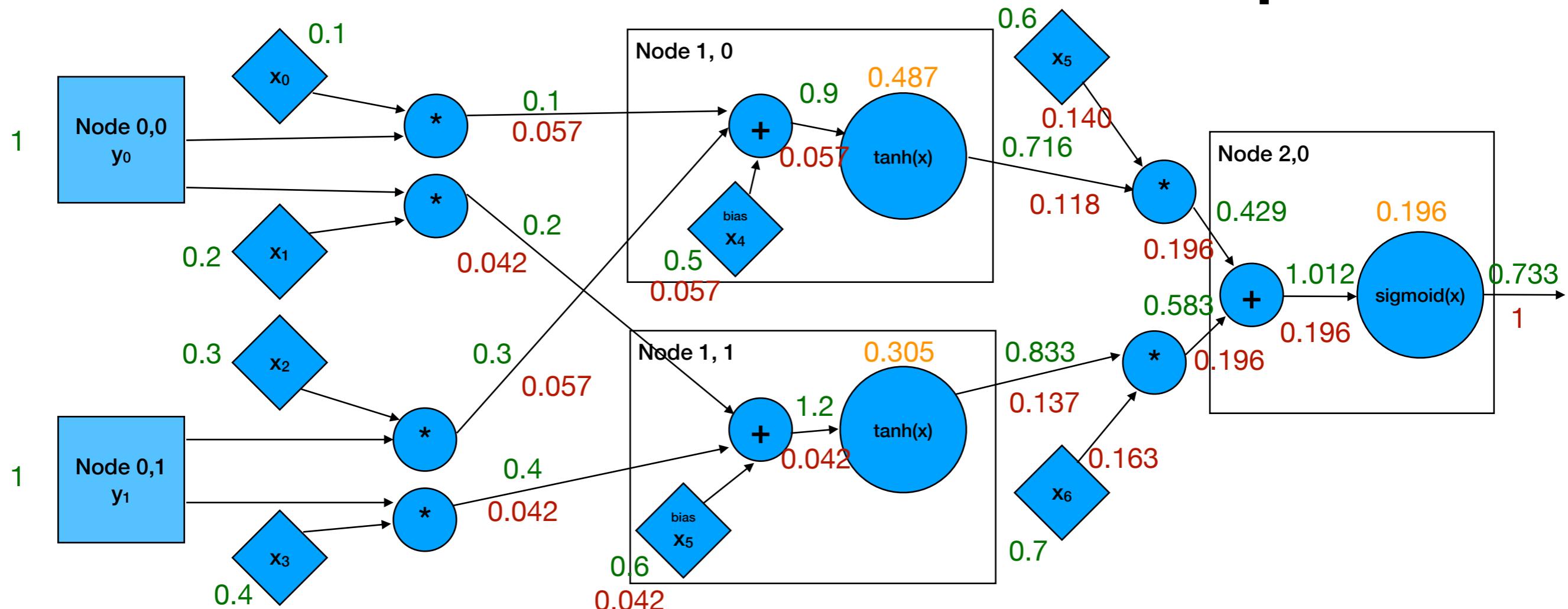
- Now we use the multiplication rule to back propagate on each of the multiplies.
- We multiply the incoming delta by the other input edge to the multiply.

The Backward Pass: An Example



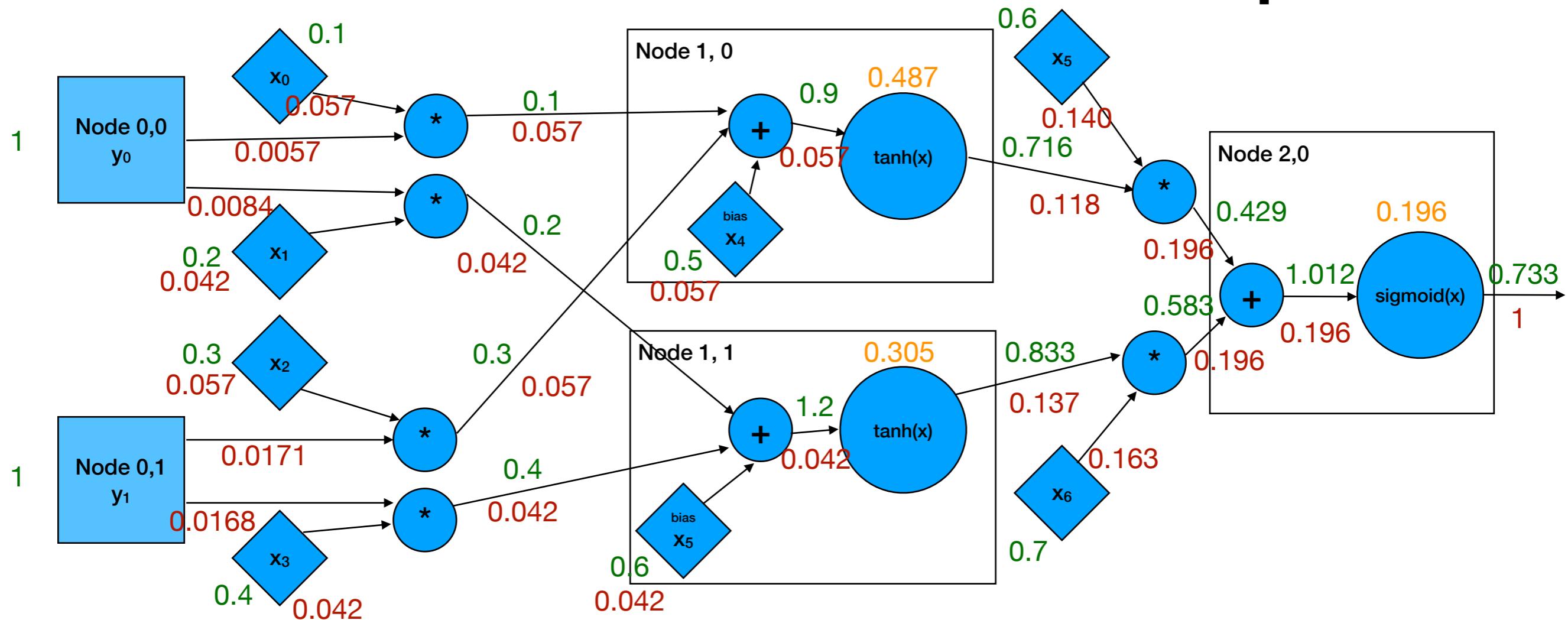
- Now we pass through the tanh activation functions by multiplying it by the incoming delta.
- $0.118 * 0.487 = 0.057$
- $0.137 * 0.305 = 0.042$

The Backward Pass: An Example



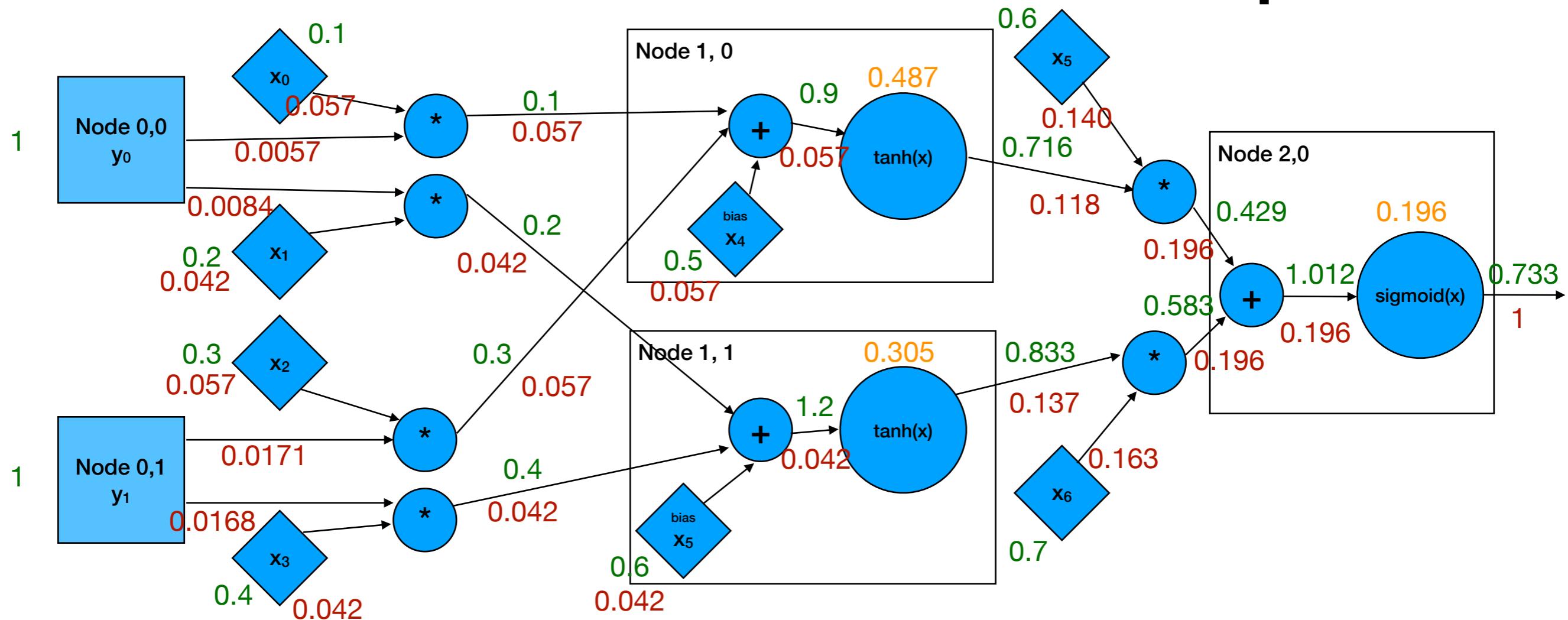
- Now since we're passing through an addition, we just keep the same value back to each edge

The Backward Pass: An Example



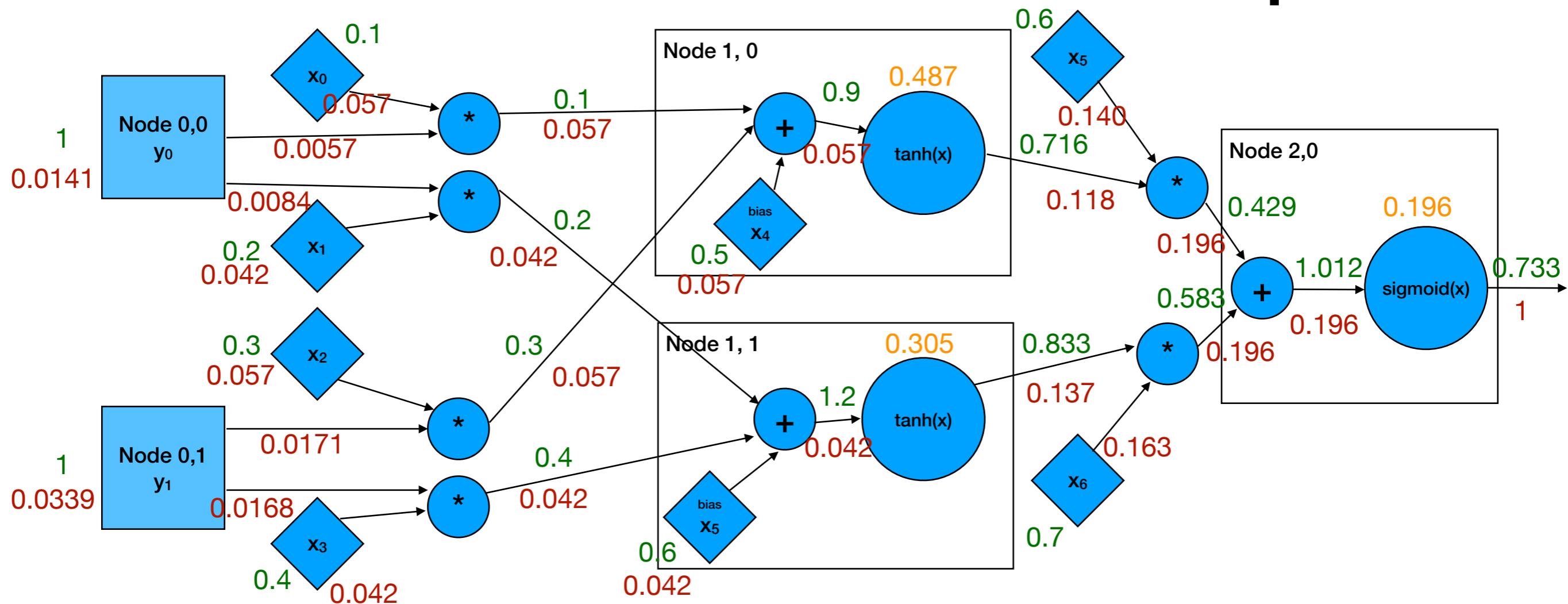
- Now we can pass back through the multiplications.

The Backward Pass: An Example



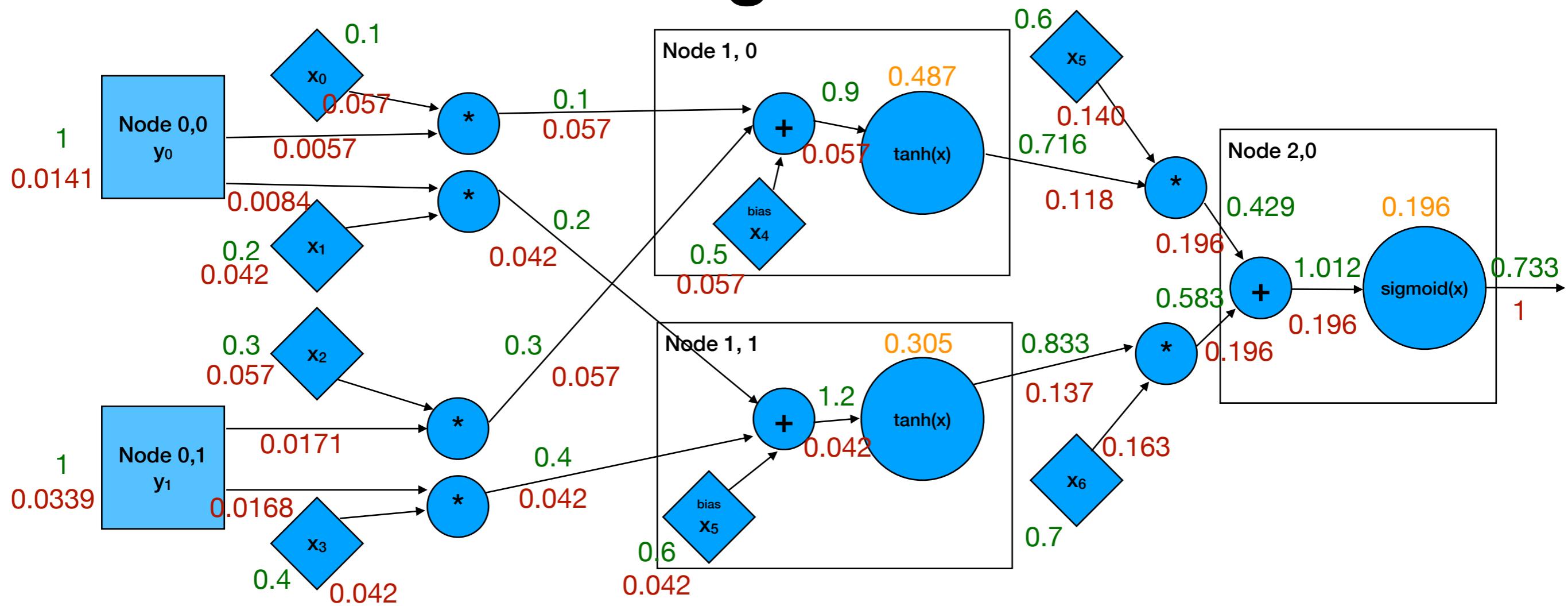
- Note that we've hit a point where we have multiple deltas coming into the same node. For this network it doesn't matter because we can't change the input y values but this will happen in most networks (especially ones with more than one hidden layer).

The Backward Pass: An Example



- If multiple deltas come into the same node (or operation) we use the sum of those deltas to propagate backwards.
- So if we were propagating backwards through node 0,0 and node 0,1, we would use 0.0141(the sum of the incoming deltas) for node 0,0 and 0.0339 for node 0,1 (the some of its incoming deltas).

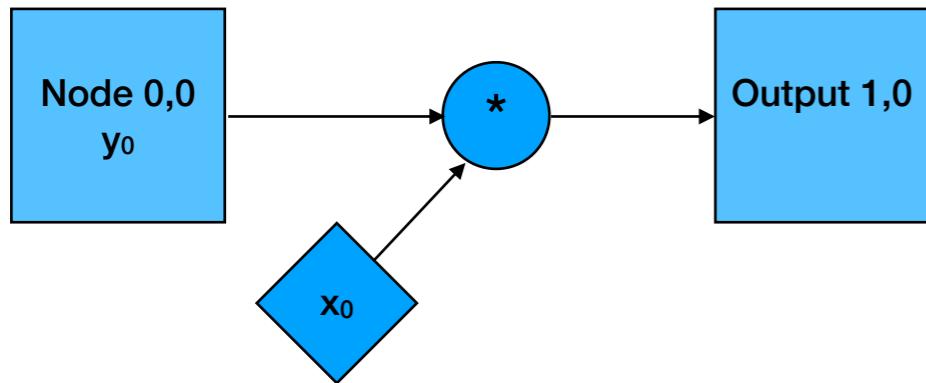
Vanishing Gradients



- You may have noticed that our gradients got fairly small. If the output of our sigmoid function is close to 0 or 1, or the output of our tanh function is close to -1 or 1; their gradients are very small.
- Since we always are multiplying deltas through the backward pass, gradients can get very small very quick.
- When this happens our hill climbing gets stuck because the moves its making are too small because the gradient is too small.

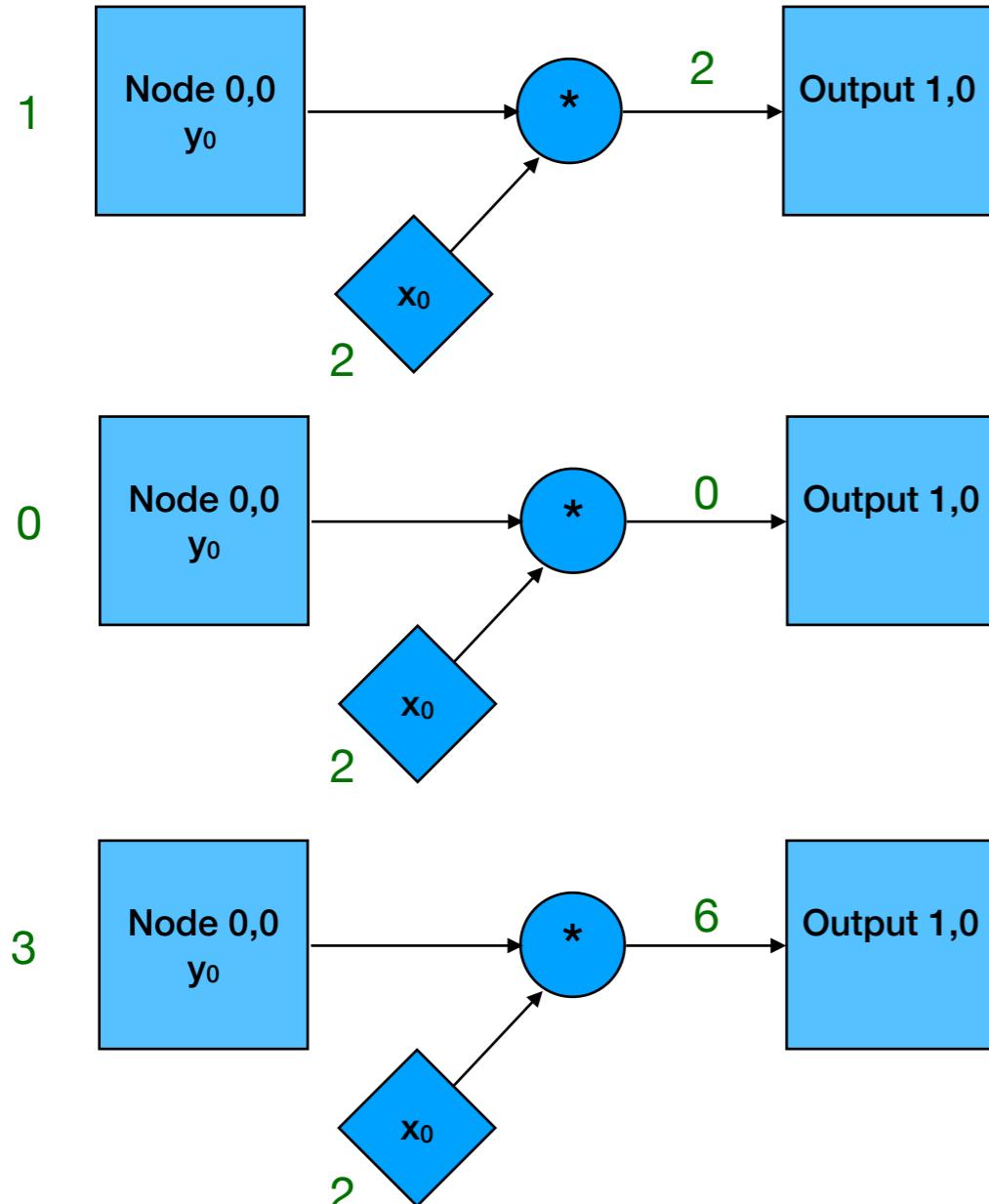
Backwards Pass with Multiple Samples

Multiple Instances/Samples



- With numerical gradients we saw it was possible to calculate the gradient for multiple instances simultaneously.
- We can do the same with backpropagation and use our circuits to motivate it.
- Let's use a very simple minimal neural network with one weight.

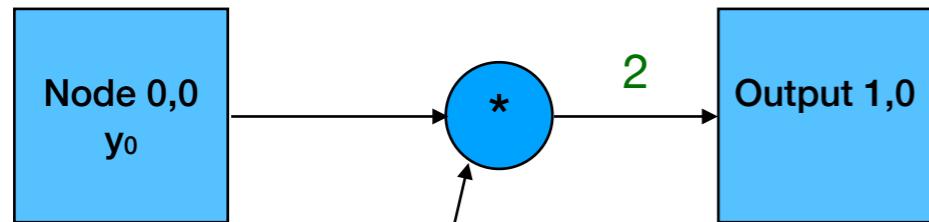
Multiple Instances/Samples



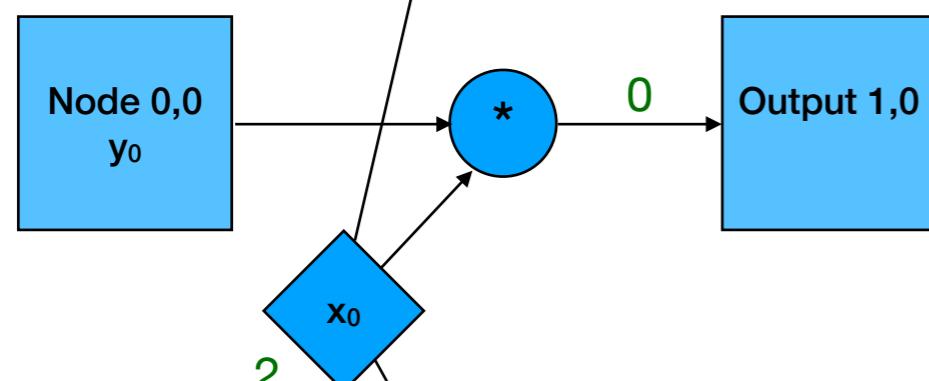
- We can do forward passes for each three using the same weights (in this case just x_0).

Multiple Instances/Samples

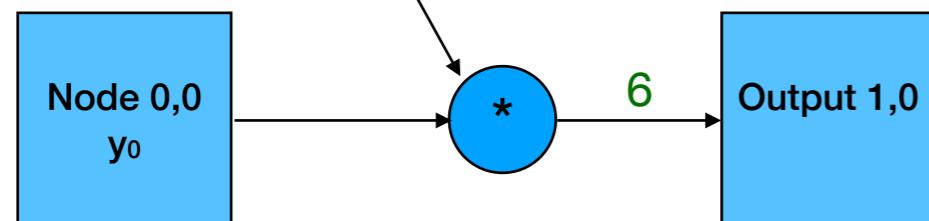
1



0



3



- Note that the x_0 value is the same for all of them. We can actually redraw this as seen on the left.
- As described previously, if we have multiple deltas going into the same node, we simply sum them.
- So in this case, to calculate the gradient across multiple samples with the same weights, we can do backpropagation on each of them individually and then simply sum up the gradients for each weight/bias.

Validating Gradients

Multiple Instances/Samples

- While you can compare the numeric and backward pass gradients directly, due to floating point precision implementations results can raise issues.
- Further, the relative size of the gradients can cause issues. If two gradients are close to 1.0 and differ by 1e4 this is not problematic. But if two gradients are close to 1e5 or lower and differ by 1e4 then the magnitude of that difference is much more significant.

Multiple Instances/Samples

- Due to this it makes more sense to calculate a relative error (where f'_a is the analytic gradient, or the gradient calculated by the backward pass, and f'_n is the numeric gradient calculated by the finite difference method):

$$error = \frac{|f'_a(x) - f'_n(x)|}{\max(|f'_a(x)|, |f'_n(x)|)}$$

Multiple Instances/Samples

$$\text{error} = \frac{|f'_a(x) - f'_n(x)|}{\max(|f'_a(x)|, |f'_n(x)|)}$$

- In this case we can use the following metrics to determine the similarity of the gradient values:
 - $\text{error} > 1e^{-2}$ usually means one gradient is wrong
 - $1e^{-2} > \text{error} > 1e^{-4}$ is not very good and could indicate a problem
 - $1e^{-4} > \text{error}$ is usually okay if you have objective functions like ReLU which have a "kink" (i.e., a max/min or if value), but if not (if you're using tanh or sigmoid) it is too high.
 - $1e^{-7} > \text{error}$ means you've calculated things correctly.
- Also note that the deeper the network the higher the relative errors will be, so these numbers can be adjusted a bit depending on size.

Lecture 3

Loss Functions and Gradient Descent

DSCI-789: Neural Networks for Data Science

Travis Desell (tjdvse@rit.edu)
Associate Professor
Department of Software Engineering



ROCHESTER INSTITUTE OF TECHNOLOGY

Overview

- Calculating Error
- Regression
- Classification
- Weight Initialization
- Gradient Descent
- Batch
- Mini-batch
- Stochastic

Calculating the Error

Calculating the Error

- Up to now, we were just using a forward pass to calculate an output of the network, and then a backward pass (or finite difference method) to calculate the gradient.
- Depending on what we're using the neural network for (e.g., regression, classification, attribute classification or something else) we will add an additional loss function to our output nodes.
- The general idea is that the gradient gives us an idea how to modify the weights to change the output, but in general we want to minimize the error of the network so we need to also compare the output of the network to the expected output(s) of or training sample(s).

Regression

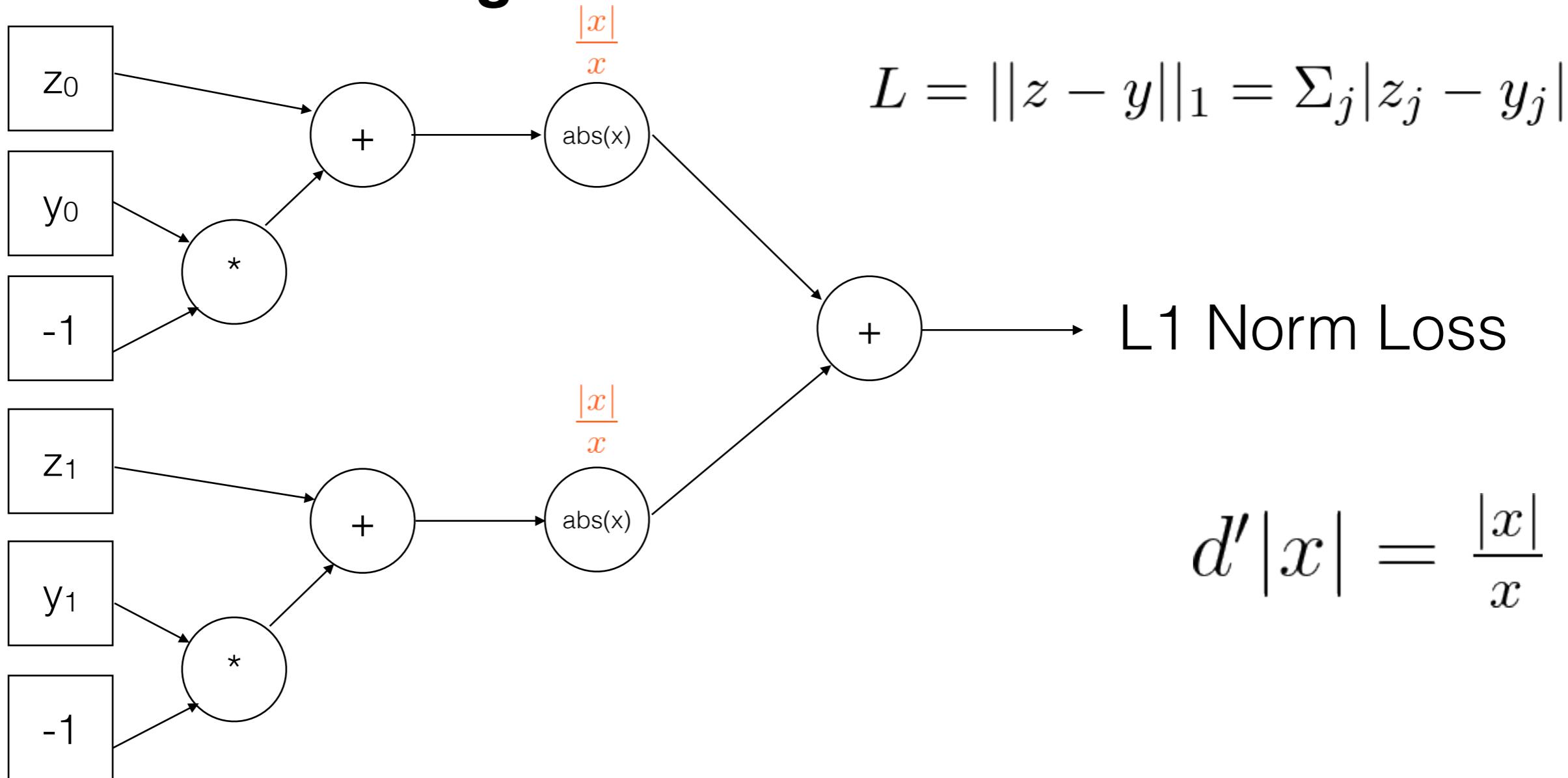
L1 Norm: $L = ||z - y||_1 = \sum_j |z_j - y_j|$

L2 Norm: $L_2 = ||z - y||_2^2 = (\sum_j |z_j - y_j|^2)^{\frac{1}{2}} = \sqrt{(\sum_j |z_j - y_j|^2)}$

L_p Norm: $L_p = ||x||_p = (\sum_j |x_j|^p)^{\frac{1}{p}}$

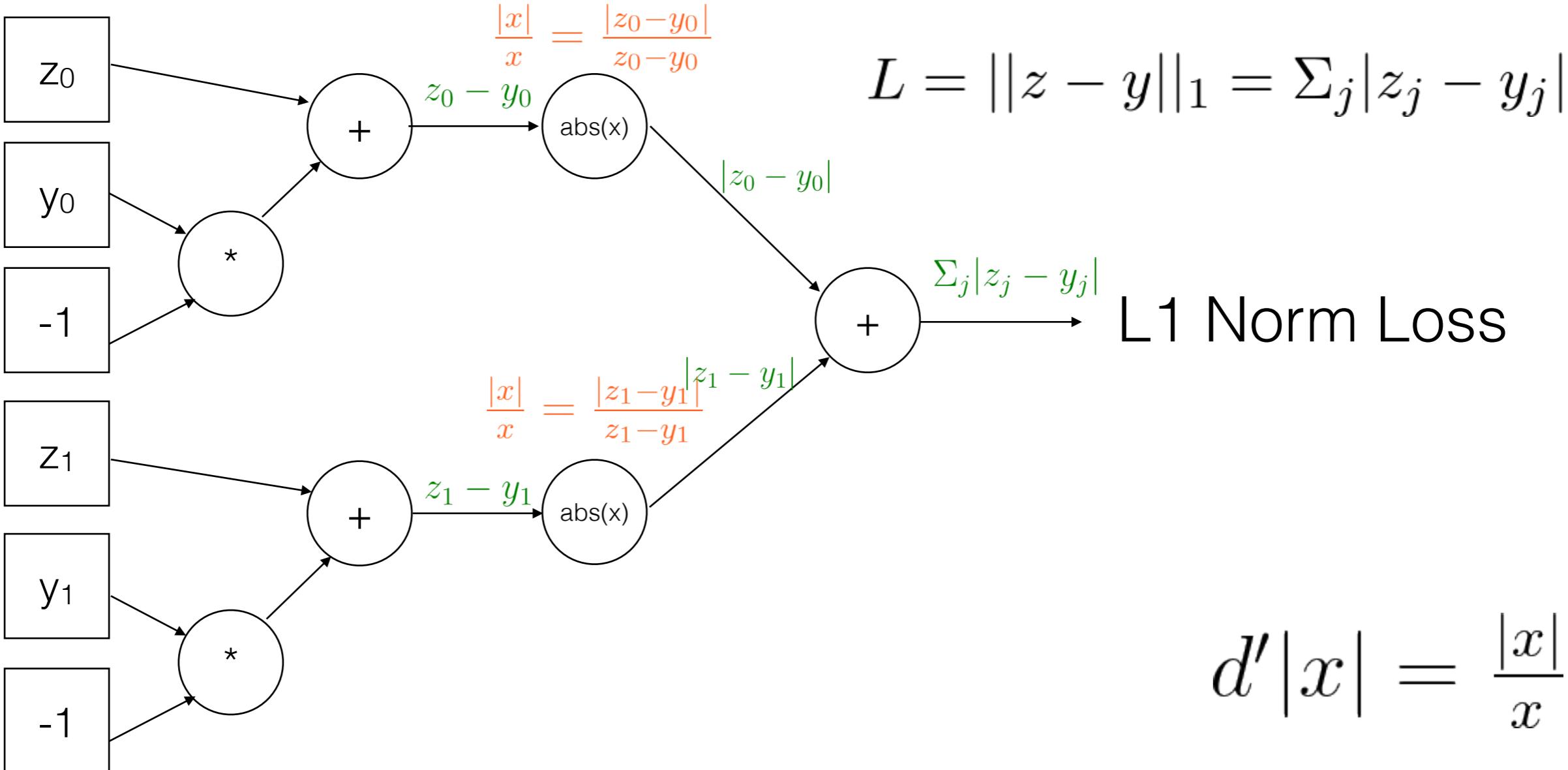
- When we are trying to predict a target value (e.g., in our simple xor case) we can use either the L1 norm (sum of absolute differences) or L2 norm (sum of squares) to calculate the error.
- In this case, L is the total loss (error) that we're trying to minimize, z is the vector of outputs from the neural network, and y is the vector of expected outputs from our training sample. j is the jth parameter of the output vector/expected output vector.

Regression - L1 Norm



- We can add these calculations to our network circuit and calculate the derivative similarly.
- The above is an example from 2 outputs, where the z_0 and z_1 are, e.g., the outputs from the sigmoid activation function of our output layer.
- Note that the derivative of $\text{abs}(x)$ is $\frac{|x|}{x}$ and we can calculate this on the forward pass to use on the backward pass similar to any other activation function. **This is undefined if $x = 0$.**
- If there are more outputs we simply connect them to the summation (the far right +).

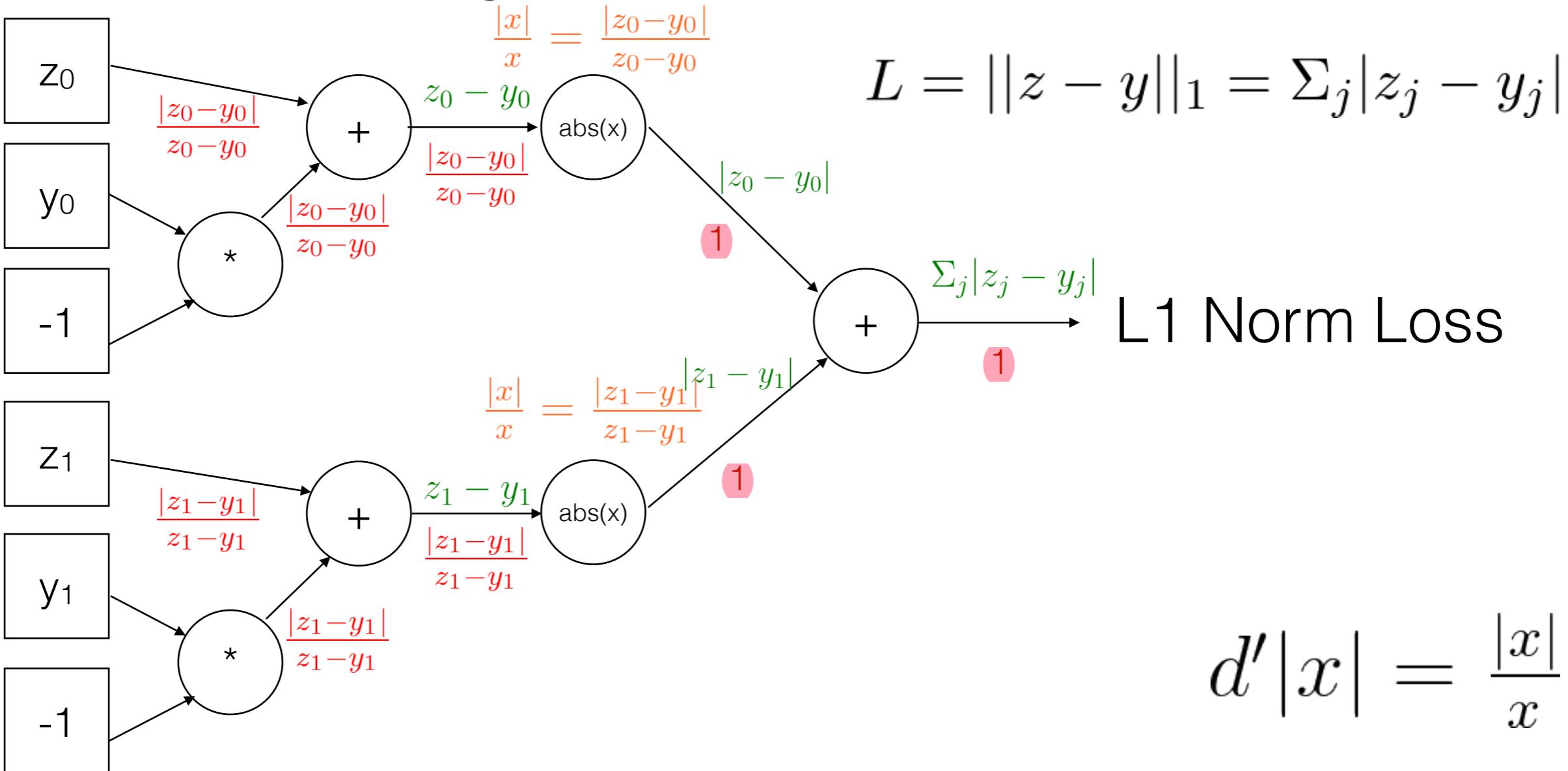
Regression - L1 Norm



$$d' |x| = \frac{|x|}{x}$$

• \

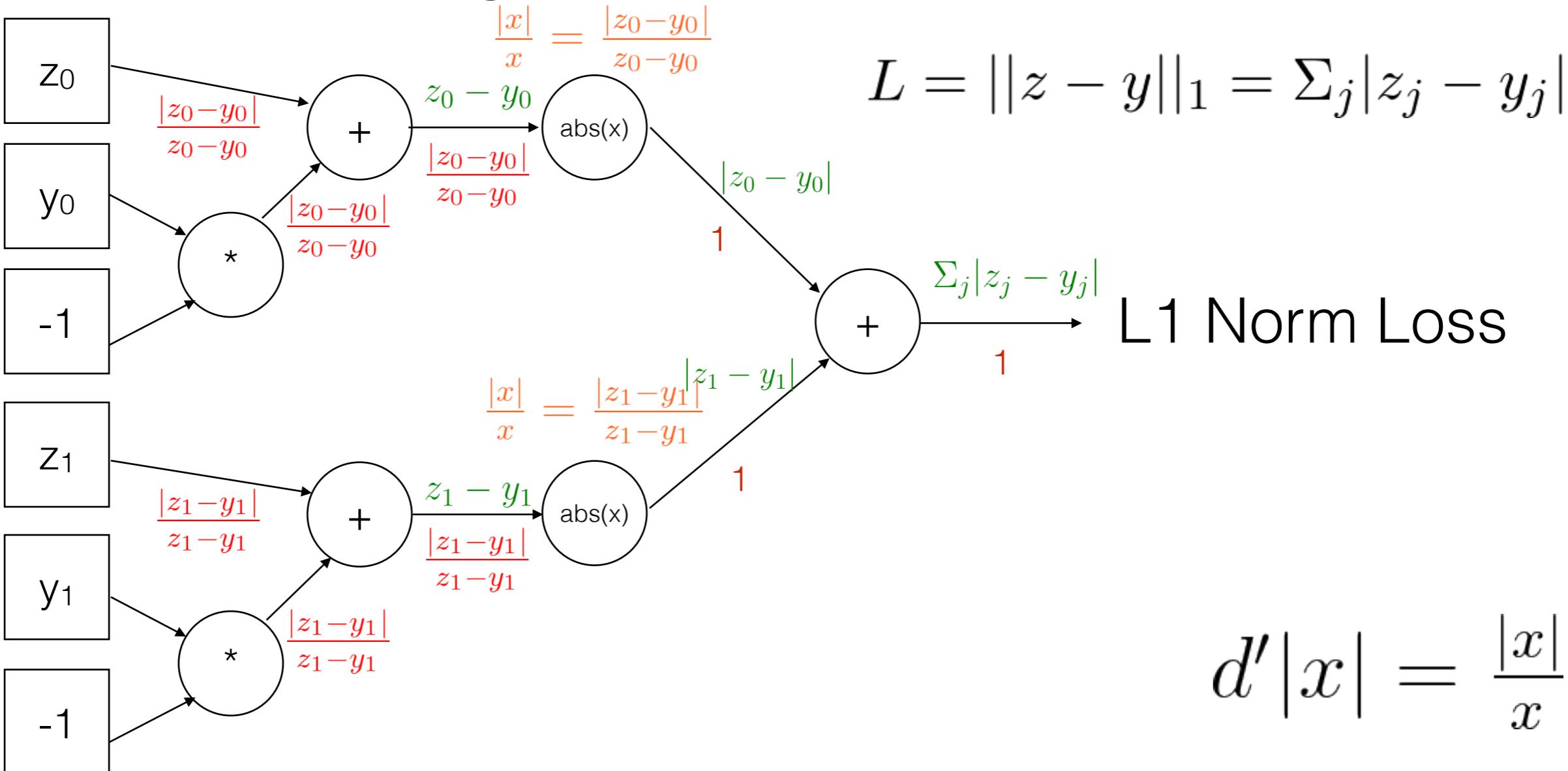
Regression - L1 Norm



- Doing a backwards pass we can simplify this a bit to determine the gradients right at the z values:

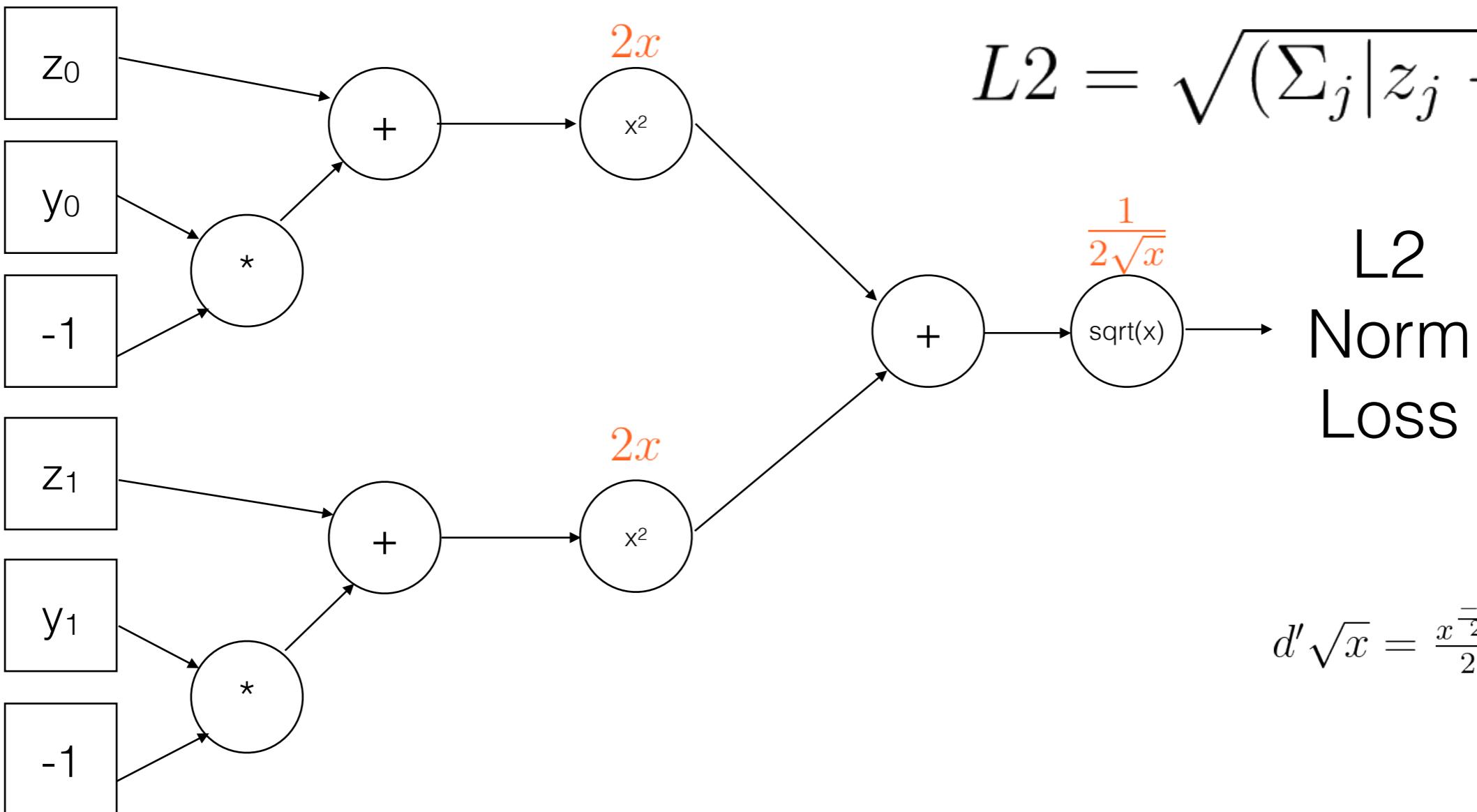
$$z'_i = abs(z_i - y_i) / (z_i - y_i)$$
- Note that $abs(z_i - y_i) / (z_i - y_i)$ simplifies to -1 if $z_i < y_i$ and 1 if $z_i > y_i$.

Regression - L1 Norm



- The L1 norm is also called the sum of absolute error, and in many cases it's divided by n (the number of output parameters) and becomes the mean absolute error (MAE).

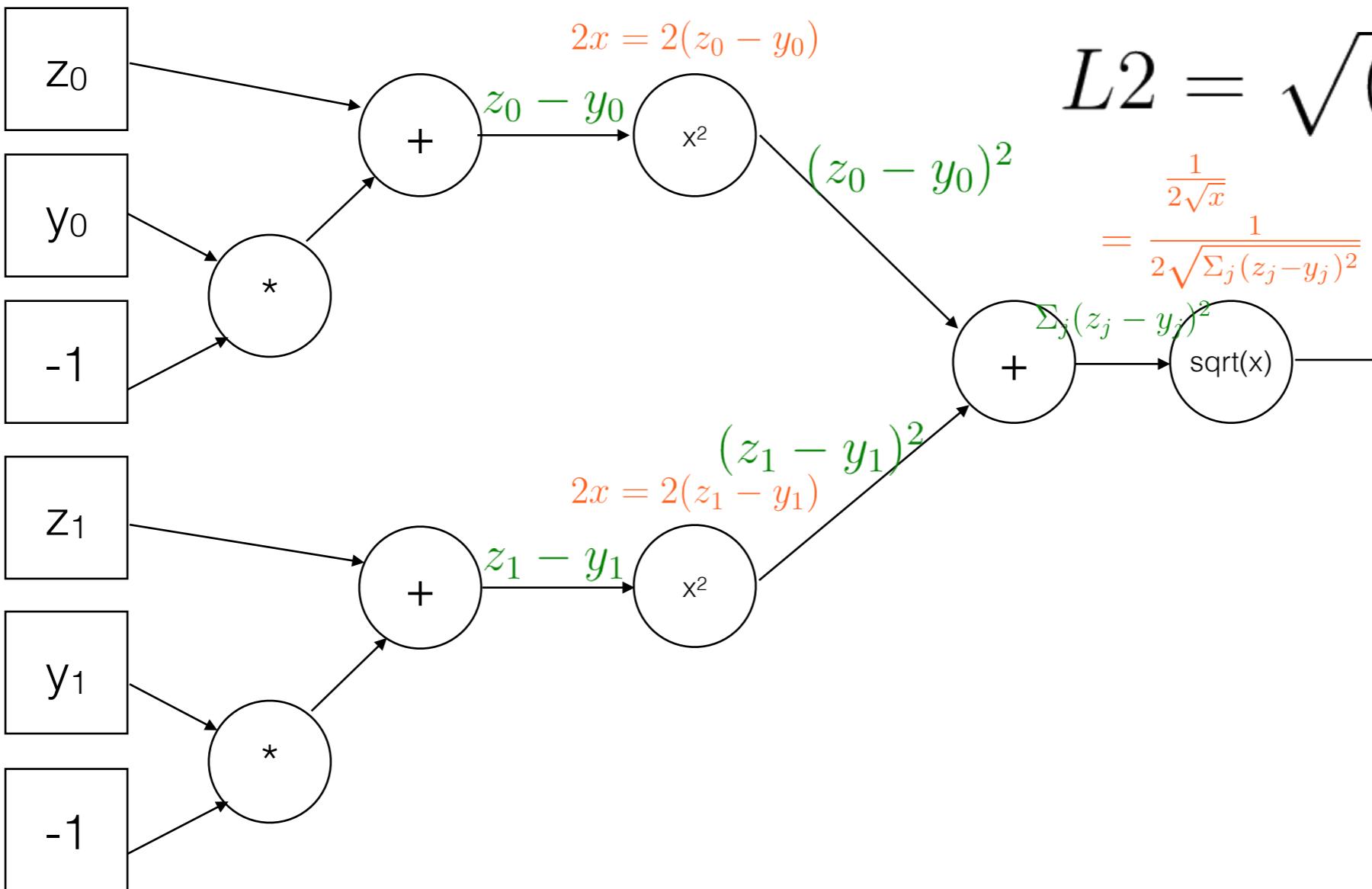
Regression - L2 Norm



$$d' \sqrt{x} = \frac{x^{-\frac{1}{2}}}{2} = \frac{1}{2\sqrt{x}}$$

- We can add these calculations to our network circuit and calculate the derivative similarly.
- The above is an example from 2 outputs, where the z_0 and z_1 are, e.g., the outputs from the sigmoid activation function of our output layer.
- Note that the derivative of x^2 is $2x$ and we can calculate this on the forward pass to use on the backward pass similar to any other activation function. This is safer than the L1 norm as it is never undefined and is actually faster to compute.
- If there are more outputs we simply connect them to the summation (the far right +).

Regression - L2 Norm



$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$

$$= \frac{1}{2\sqrt{x}}$$

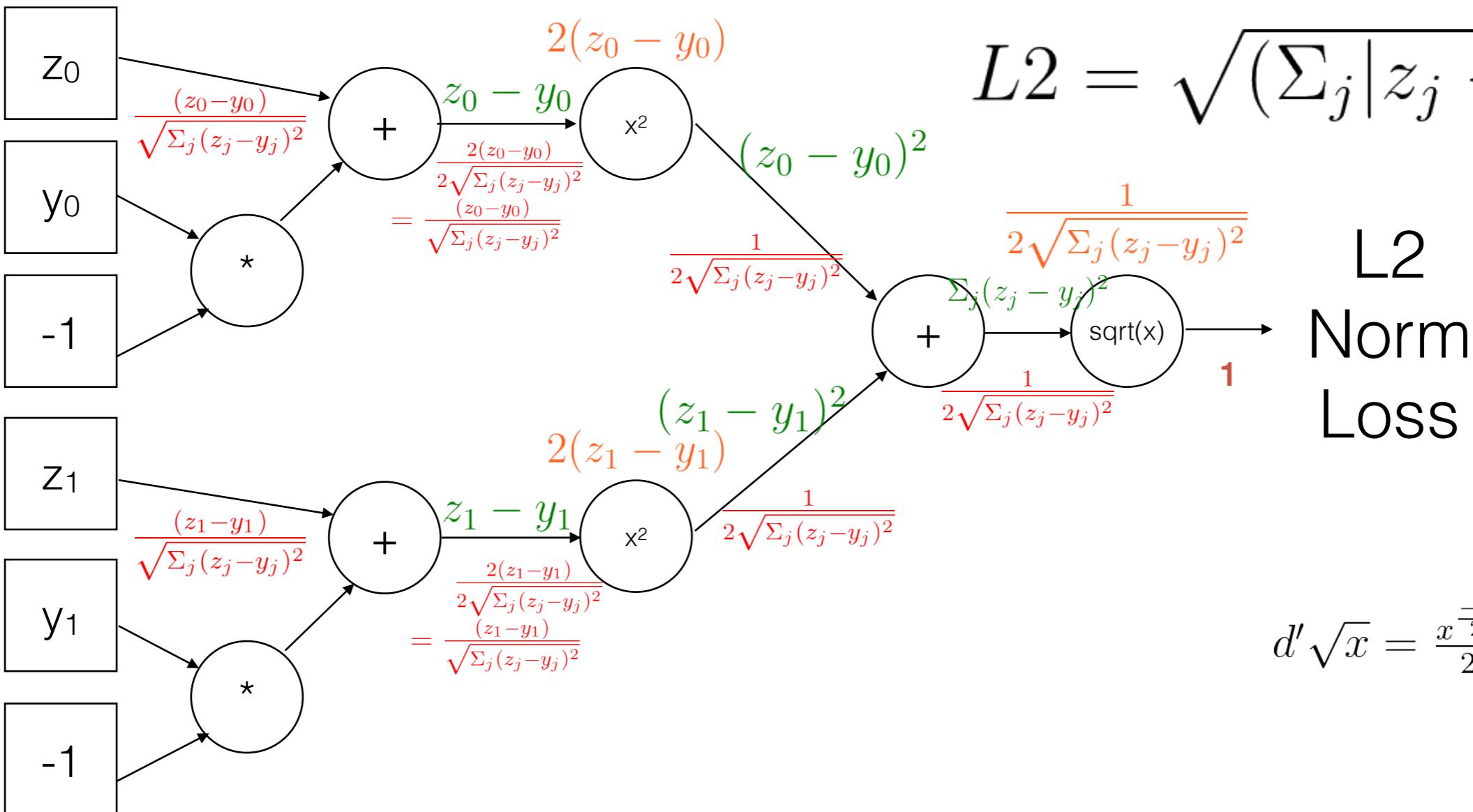
$$= \frac{1}{2\sqrt{\sum_j (z_j - y_j)^2}}$$

L2
Norm
Loss

$$d' \sqrt{x} = \frac{x^{-\frac{1}{2}}}{2} = \frac{1}{2\sqrt{x}}$$

- We can do a forward pass to calculate the derivatives going through the x^2 and \sqrt{x} functions.

Regression - L2 Norm



$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$

L2
Norm
Loss

$$d' \sqrt{x} = \frac{x^{-\frac{1}{2}}}{2} = \frac{1}{2\sqrt{x}}$$

- Similarly we can do a backward pass to calculate the derivative for the z values. This ends up being fairly straightforward:
- The L2 Norm is also called the sum of squares error, or mean squared error (MSE).

$$d' z_i = \frac{(z_i - y_i)}{\sqrt{\sum_j (z_j - y_j)^2}}$$

Classification - SVM

SVM

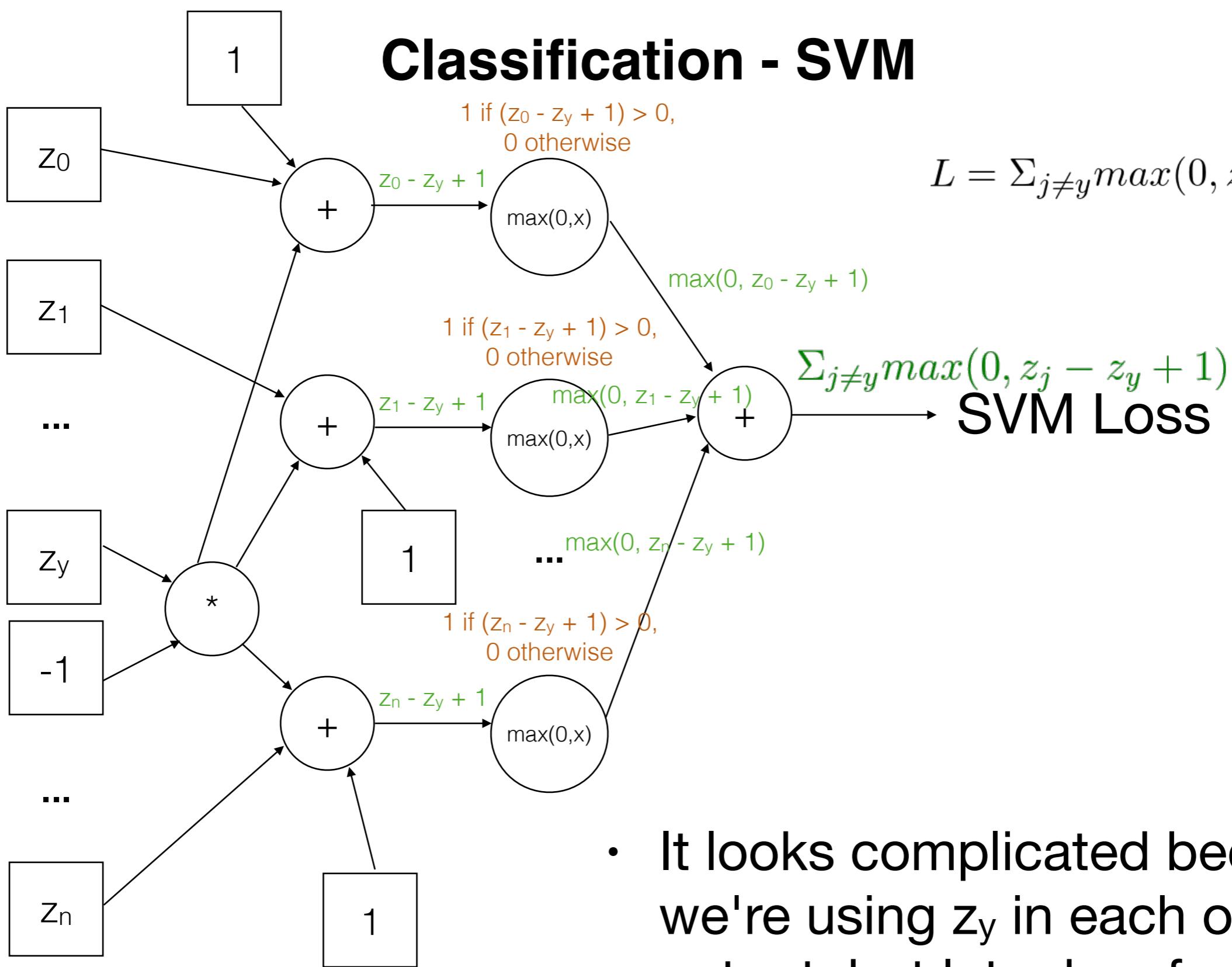
$$L = \sum_{j \neq y} \max(0, z_j - z_y + 1)$$

Squared Hinge Loss

$$L = \sum_{j \neq y} \max(0, z_j - z_y + 1)^2$$

- Here we can define each output z_j as the network's confidence in the j^{th} class. y is the expected class.
- So here we're comparing the network's output for a given class j to its output for the expected class y .
- Some work reports the square hinge loss working better than SVM.
- Why do we have the $+1$? If $L \geq 1$ it belongs to that class, if $L = 0$ it belongs to another class, in between is not sure.

Classification - SVM

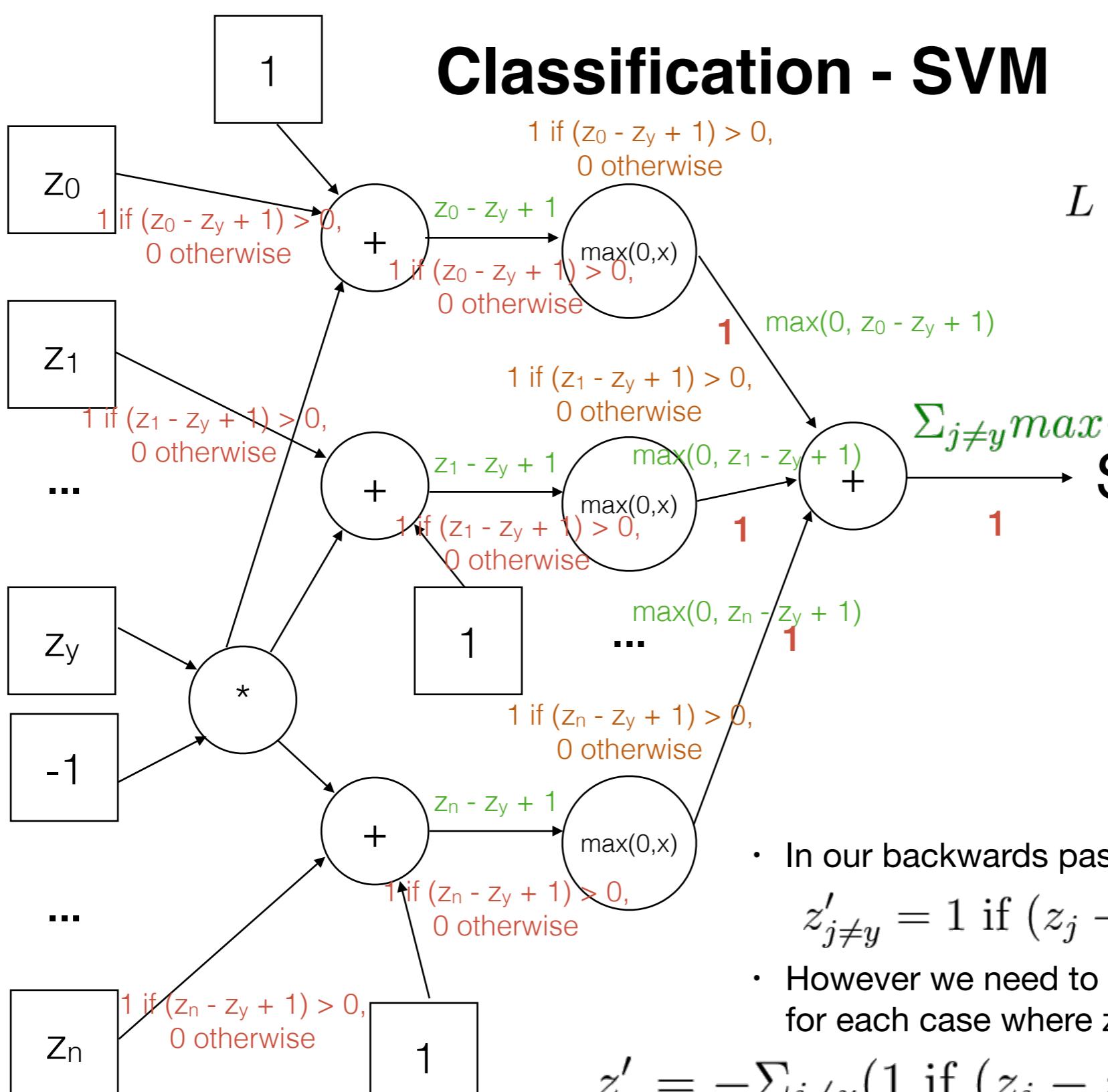


$$L = \sum_{j \neq y} \max(0, z_j - z_y + 1)$$

$$\sum_{j \neq y} \max(0, z_j - z_y + 1) \quad \text{SVM Loss}$$

- It looks complicated because we're using z_y in each other output, but let's do a forward pass to show what's going on.

Classification - SVM



$$L = \sum_{j \neq y} \max(0, z_j - z_y + 1)$$

SVM Loss

- In our backwards pass when $j \neq y$ it is straightforward:

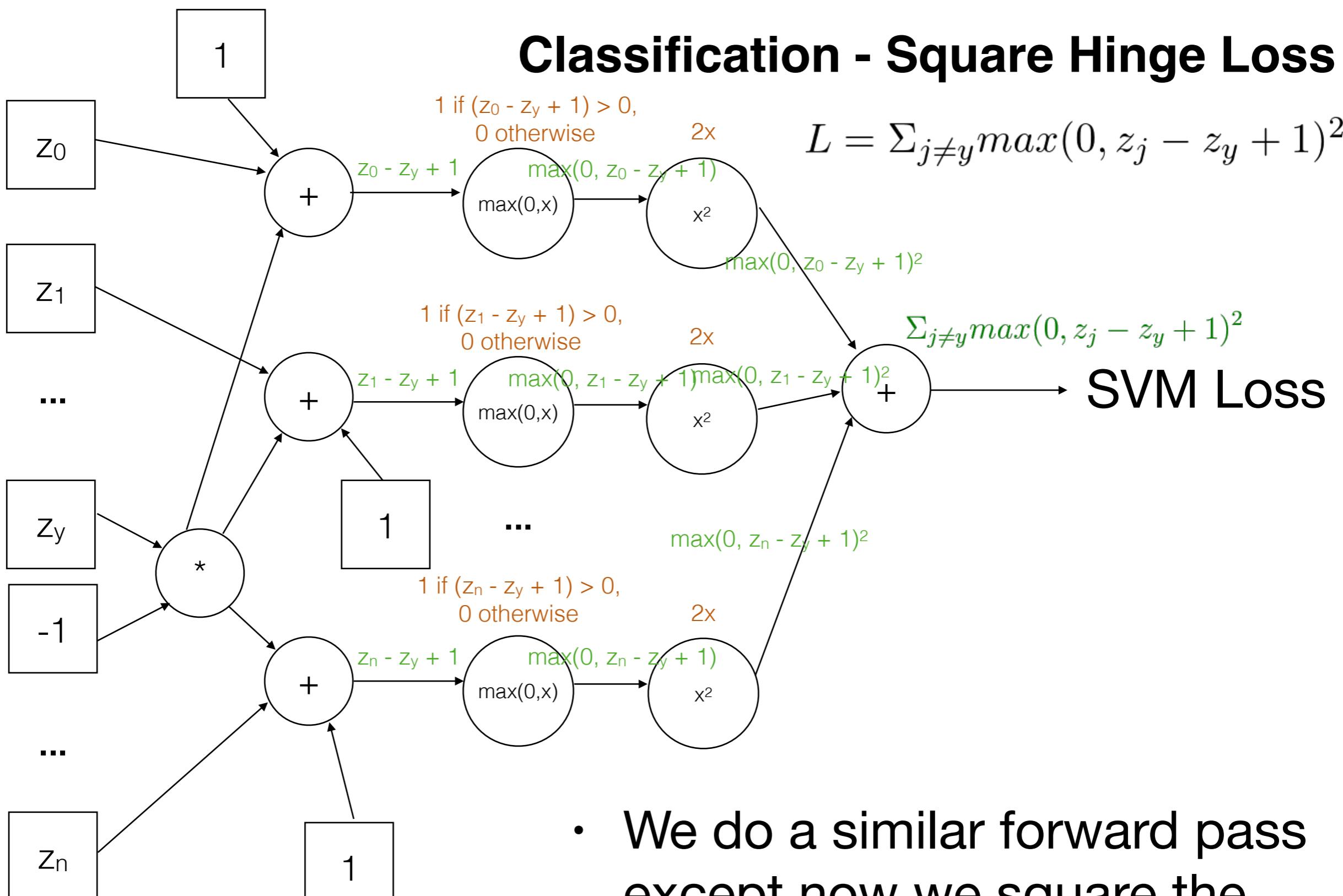
$$z'_{j \neq y} = 1 \text{ if } (z_j - z_y + 1) > 0, 0 \text{ otherwise}$$

- However we need to accumulate the gradients into z_y for each case where $z_j - z_y + 1 > 0$:

$$z'_y = -\sum_{j \neq y} (1 \text{ if } (z_j - z_y + 1) > 0, 0 \text{ otherwise})$$

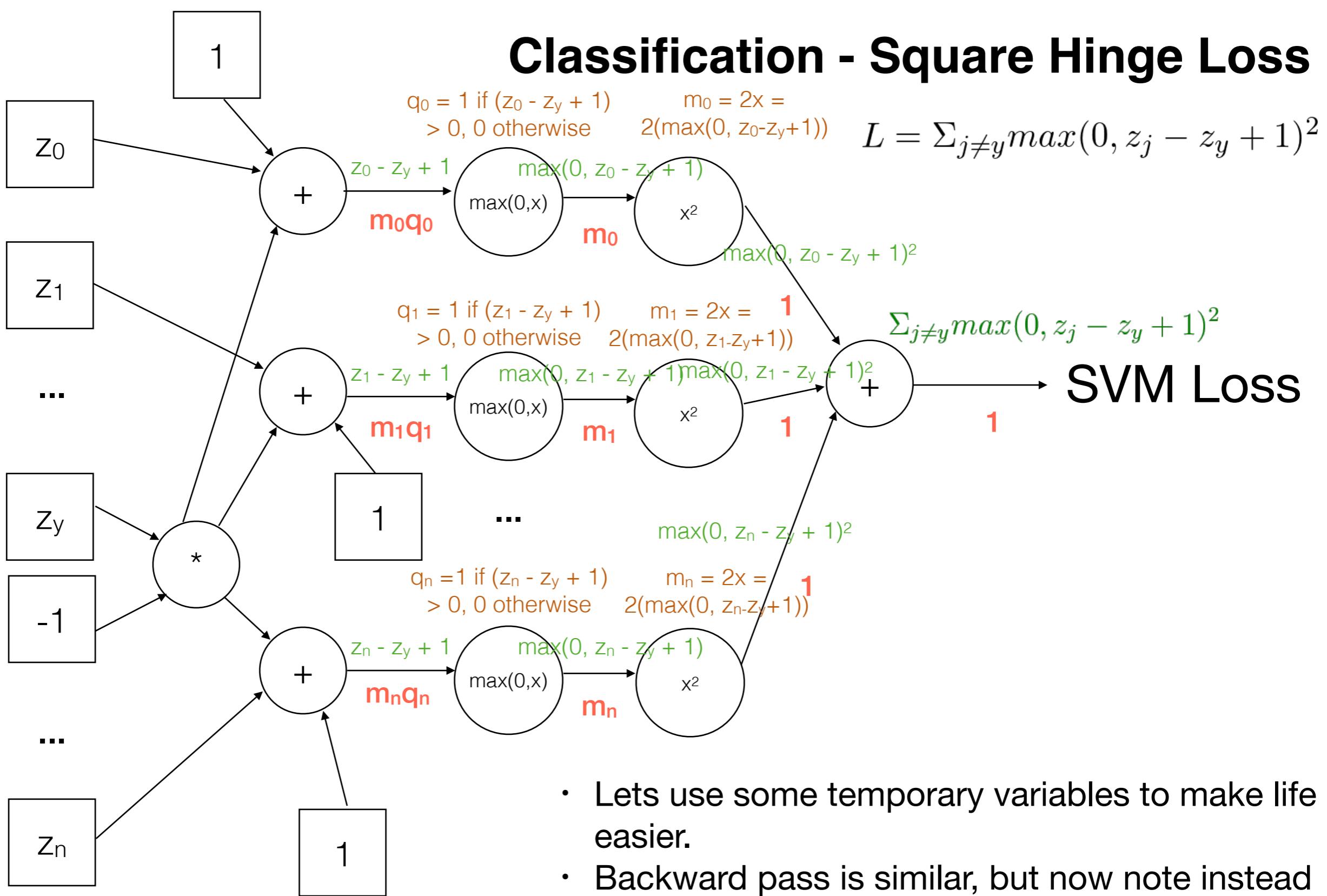
which is just: $z'_y = -\sum z'_{j \neq y}$

Classification - Square Hinge Loss



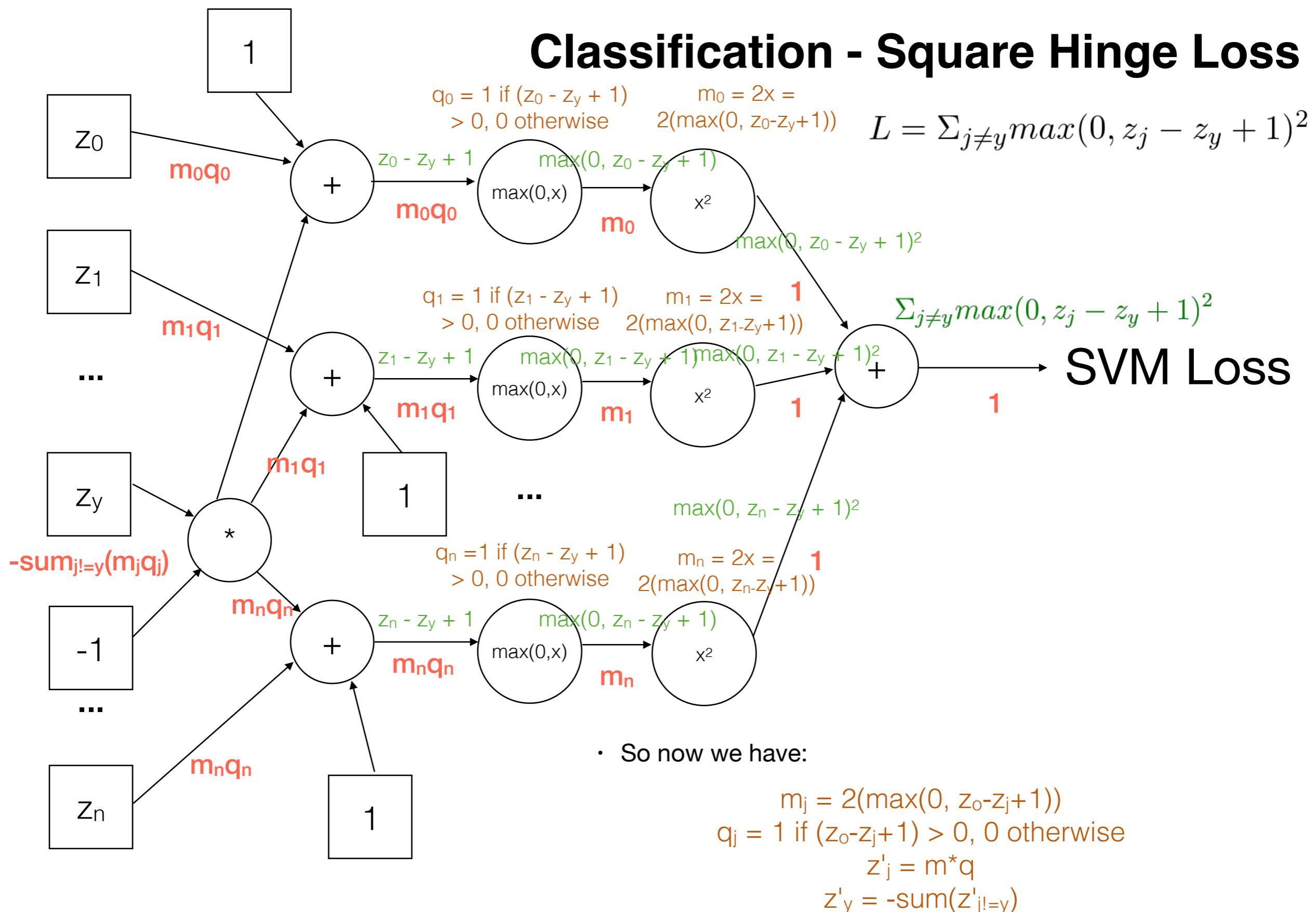
- We do a similar forward pass except now we square the results of the max operation.

Classification - Square Hinge Loss



- Lets use some temporary variables to make life easier.
- Backward pass is similar, but now note instead of 1 being the delta into the max operation, it is now $2x$ where $x = \max(0, z_j - z_y + 1)$ otherwise.

Classification - Square Hinge Loss



Classification - Softmax

Softmax

$$L = -\log\left(\frac{e^{z_y}}{\sum_j e^{z_j}}\right)$$

- Again, we can define each output z_i as the network's confidence in the i^{th} class. y is the expected class.
- What this does is calculate the -log likelihood of the network's output for the expected class.
- By summing all z_j 's the softmax function converts all outputs to a percentage that sums up to 1.

Softmax Function

- Why would we use a softmax function on the output layer?
- It is great for classification as it converts all outputs into probabilities which sum to 1.0. After that we just take the - log likelihood of the output for the target class.
- Example:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \text{ for } i = 1, \dots, K \text{ and } z = (z_1, \dots, z_K)$$

$$z_0 = 8$$

$$z_1 = 2$$

$$z_3 = 5$$

$$e^{z_0} = 2980.95$$

$$e^{z_1} = 7.39$$

$$e^{z_2} = 148.41$$

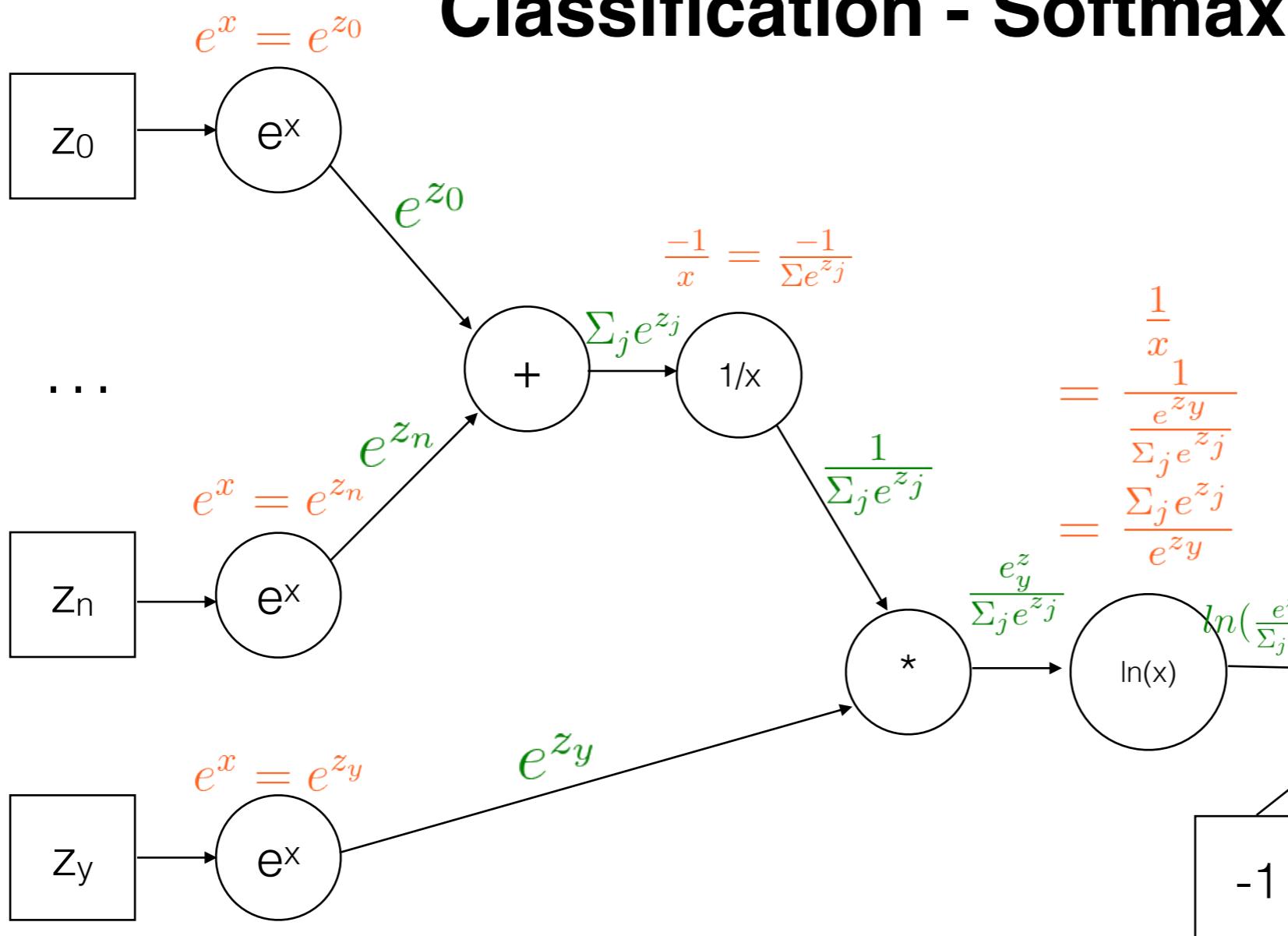
$$\sum_{j=1}^K e^{z_j} = 3136.75$$

$$\sigma(z_0) = \frac{2980.95}{3136.75} = 0.9503$$

$$\sigma(z_1) = \frac{7.39}{3136.75} = 0.0023$$

$$\sigma(z_2) = \frac{148.41}{3136.75} = 0.0473$$

Classification - Softmax



$$L = -\log\left(\frac{e^{z_y}}{\sum_j e^{z_j}}\right)$$

$$d'e^x = e^x$$

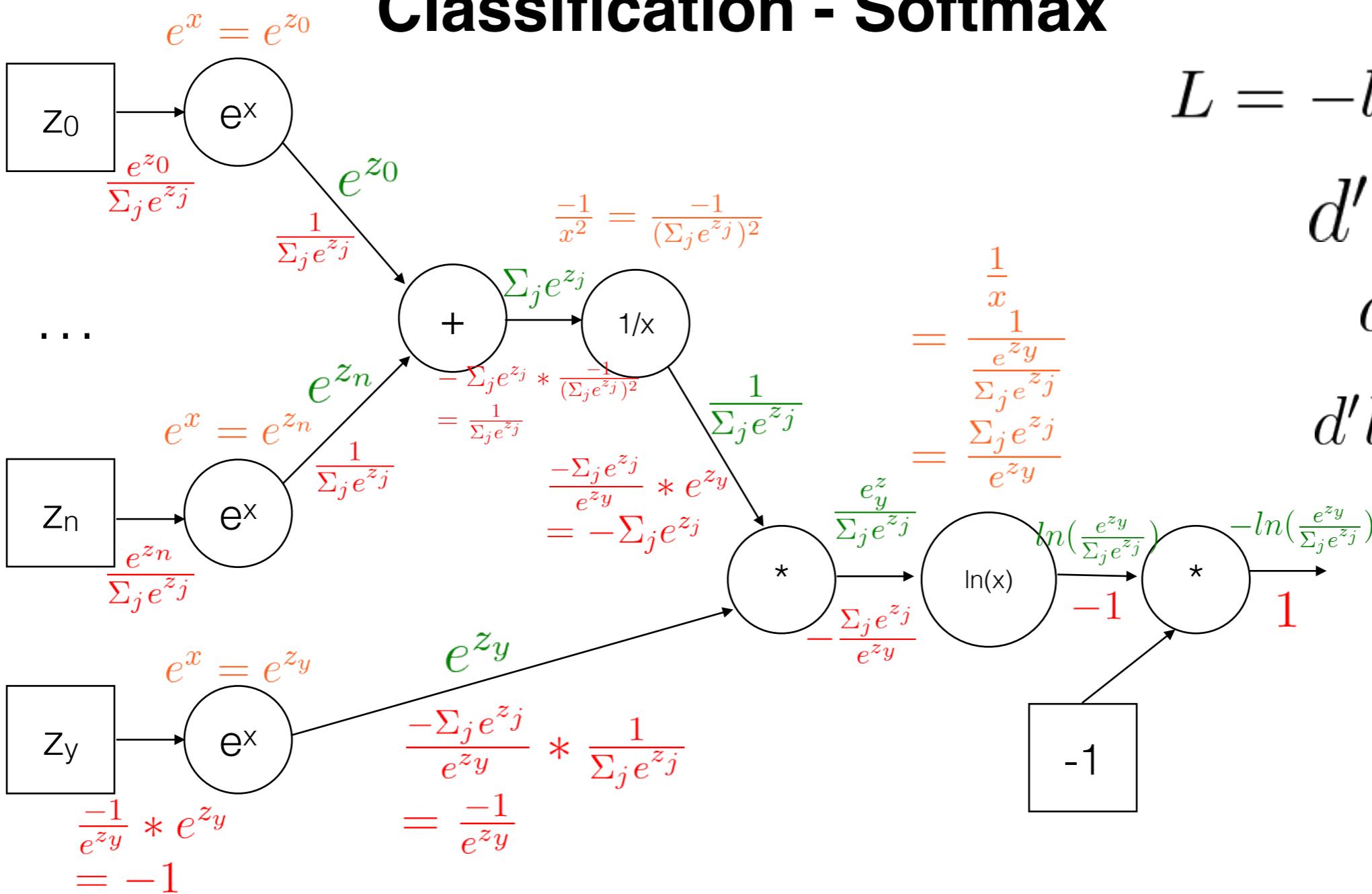
$$d' \frac{1}{x} = \frac{-1}{x^2}$$

$$d' \ln(x) = \frac{1}{x}$$

Softmax Loss

- Similar to before we can compute the derivatives for the various functions during our forward pass through the network.

Classification - Softmax



$$L = -\log\left(\frac{e^{z_y}}{\sum_j e^{z_j}}\right)$$

$$d'e^x = e^x$$

$$d' \frac{1}{x} = \frac{-1}{x^2}$$

$$d' \ln(x) = \frac{1}{x}$$

Softmax Loss

- Now we can do the backward pass again.
- Things actually simplify out quite nicely (we just need to note that the delta for z_y is added to the z_i from the top where $i = y$):

$$d' z_{i \neq y} = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

$$d' z_y = \frac{e^{z_y}}{\sum_j e^{z_j}} - 1$$

Classification - Softmax

$$d'z_{i \neq y} = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

$$d'z_y = \frac{e^{z_y}}{\sum_j e^{z_j}} - 1$$

- So after all that math this is actually quite intuitive. $e^{z_i}/\sum(e^{z_j})$ is always going to be between 0 and 1.
- For the outputs that are not for the expected class, the derivative is going to be > 0 so we need to **reduce** those outputs.
- For the outputs that are for the expected class, the derivative is going to be < 0 so we need to **increase** those outputs.

Classification - Softmax (Numerical Issues)

$$L = -\log\left(\frac{e^{z_y}}{\sum_j e^{z_j}}\right)$$

$$L = -\log\left(\frac{e^{z_j - \max(z)}}{\sum_j e^{z_j - \max(z)}}\right)$$

$$d' z_{i \neq y} = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

$$d' z_y = \frac{e^{z_y}}{\sum_j e^{z_j}} - 1$$

- You may find that your softmax function runs into numerical issues when z_y becomes too large. This won't happen with sigmoid or tanh activation functions, but it will with linear and ReLU functions (because the potential output is not bounded).
- An easy fix to this is to first calculate the max output (z) value and subtract it from each output. The gradient actually remains the same (I'll leave it to an exercise for you to circuit diagram it out and prove it to yourself).

Regression is Harder than Classification

- Optimizing for L2 loss is much more challenging than for SVM or softmax loss. For L2 loss the neural network is trying to learn an *exact* value.
- For SVM and softmax, the output for the right class just needs to be proportionally higher than the other outputs.
- If you can make your problem a classification problem instead of a regression problem you will get better results, e.g., instead of getting a single output neuron to predict 1, 2, 3, or 4 where those are your four classes, have four output neurons and use a SVM or softmax loss function.

Gradient Descent

Useful Terminology

- *convex function*: a function is convex if it has no local minima. It may have flat valleys with small but non-zero gradients, but for every point except the global minima there should be a non zero gradients.
- *monotonic function*: the output of a function can be said to be monotonically increasing if every subsequent output is larger than the last one, or monotonically decreasing if every subsequent output is smaller than the last one.
- Monotonicity is related to convexity but they are not quite the same. We want to have the search space for gradient descent to be convex if possible as it makes learning via hill climbing/gradient descent easier. We typically want our loss to be monotonically decreasing (i.e., always getting better).

Gradient Descent

```
while(true):
```

```
    gradient = neural_network.get_gradient();
```

```
    weights -= step_size * gradient;
```

- In essence gradient descent is a very simple concept (see above), but in practice it has many variations and hyperparameters.
- Note that we subtract the gradient from the weights. The gradient gives us the direction to move the weights to increase the output. Typically we are minimizing the error so we want to move in the opposite direction.
- You'll also note a step_size parameter, which is the first hyper parameter we'll deal with.

Step Size (or Learning Rate)

- From the previous lecture (and PA1-2) you may have noticed that the H value for the finite difference method can have a pretty big effect on the accuracy of the gradients.
- We get a similar issue when it comes to determining the gradient and there's a reason the same terminology is used. The farther we move from the point we calculated the gradient, the less accurate it is.

Step Size (or Learning Rate)

- I tested this with our toy xor problem on the sample test networks. While ideal step size here is much larger than what you usually want (due to this being a toy problem) the effect is demonstrated.
- After a point increasing the step size can degrade the improvement of the step.

```
[INFO ] initial output: size: 1.104871961909497
[INFO ] output with step size: 0.000000001000000 -- 1.104871961897984, difference from initial: 0.00000000011513
[INFO ] output with step size: 0.000000010000000 -- 1.104871961794369, difference from initial: 0.000000000115128
[INFO ] output with step size: 0.000000100000000 -- 1.104871960758218, difference from initial: 0.0000000001151279
[INFO ] output with step size: 0.000001000000000 -- 1.104871950396711, difference from initial: 0.000000011512786
[INFO ] output with step size: 0.000010000000000 -- 1.104871846781643, difference from initial: 0.000000115127853
[INFO ] output with step size: 0.000100000000000 -- 1.104870810631603, difference from initial: 0.000001151277894
[INFO ] output with step size: 0.001000000000000 -- 1.104860449194418, difference from initial: 0.000011512715079
[INFO ] output with step size: 0.001000000000000 -- 1.104756841150768, difference from initial: 0.000115120758728
[INFO ] output with step size: 0.010000000000000 -- 1.103721399705812, difference from initial: 0.001150562203685
[INFO ] output with step size: 0.100000000000000 -- 1.093437033261385, difference from initial: 0.011434928648111
[INFO ] output with step size: 1.000000000000000 -- 1.003234880418295, difference from initial: 0.101637081491202
[INFO ] output with step size: 10.000000000000000 -- 1.352625900522723, difference from initial: -0.247753938613227
[INFO ] output with step size: 100.000000000000000 -- 1.730117583069014, difference from initial: -0.625245621159517
[INFO ] output with step size: 1000.000000000000000 -- 1.999999999567319, difference from initial: -0.895128037657822
[INFO ] output with step size: 10000.000000000000000 -- 2.000000000000000, difference from initial: -0.895128038090503
```

Step Size (or Learning Rate)

- To provide some other intuition, consider the below search space (red is higher error, blue is lower error). If we calculate the gradient from the point and step too far we'll go past the minima along the line.
- In general we want to take multiple small steps as whenever we move the gradient changes.

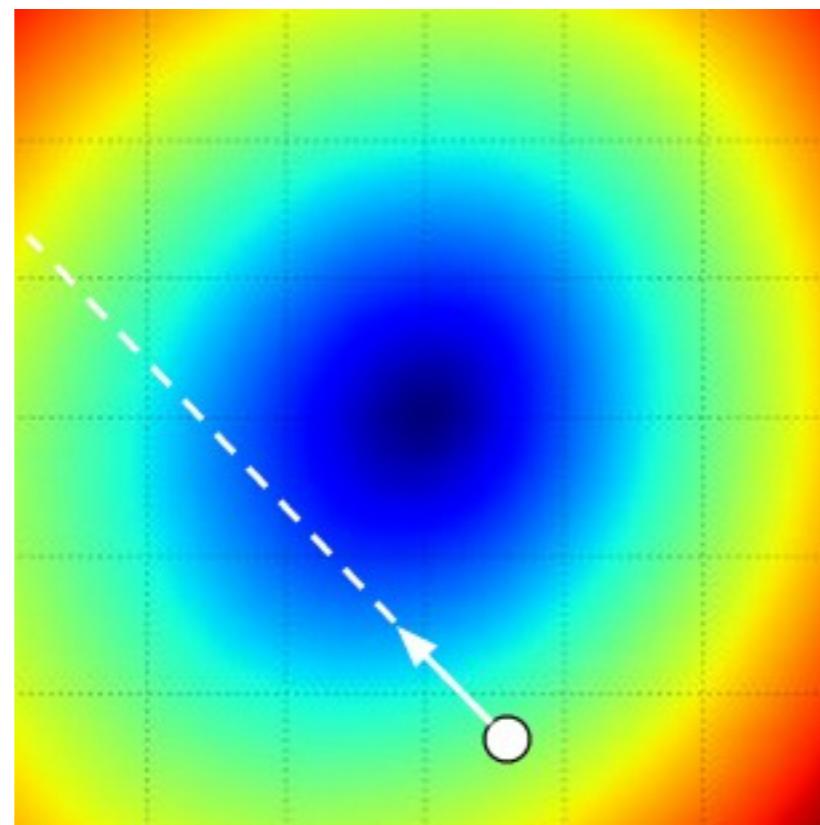


Image from: <http://cs231n.github.io/assets/stepsizes.jpg> (a great online course resource).

Batch Gradient Descent

```
while(true):
```

```
    gradient = neural_network.get_gradient(all instances);  
    weights -= step_size * gradient;
```

- The different flavors of gradient descent depend on the number of training instances we calculate the gradient over.
- If we use all training instances then this is typically denoted batch gradient descent (or sometimes just gradient descent).

Pros:

- Gradient descent is not stochastic (randomized). If you use the same initial weights it will follow the same trajectory through the search space.

Cons:

- This is typically the slowest method. Only one move is made over a full pass of the training instances. If the gradient over all samples averages close to 0 (which is not uncommon) this will end up in very slow learning.

Mini-Batch Gradient Descent

```
while(true):
```

```
    Instance[] instances = data.random_sample(N);  
    gradient = neural_network.get_gradient(instances);  
    weights -= step_size * gradient;
```

- Mini-batch gradient descent (sometimes also called batch gradient descent or stochastic gradient descent just to add to the confusion) selects random samples of the training instances and calculates the gradient over the group.
- This can lead to some interesting performance improvements (batch normalization) and is very common in convolutional neural networks (CNNs). It also aids in effectively utilizing GPUs as the forward and backward passes can be calculated in parallel on one. Batch size is a hyperparameter, and a common value is 256 for large scale datasets.

Pros:

- Batching allows for performance improvements (GPUs).
- Faster than batch gradient descent as we can do weight updates more frequently.
- Allows performance improving tricks like batch normalization.

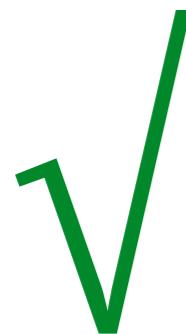
Cons:

- This is a stochastic method, the batches are randomly selected so even when starting with the same weights and inputs you can get different results.
- Batch size becomes another hyper parameter to optimize for.

Mini-Batch Gradient Descent



```
while (true):
    Instance[] instances = data.getNRandomInstances(batchSize);
    gradient = neuralNetwork.getGradient(instances)
    weights -= stepSize * gradient
```



```
while (true):
    //each iteration of the while loop is an epoch
    data.shuffle() //reshuffle the data before each epoch
    for (i = 0; i < numberSamples; i += batchSize):
        Instance[] instances = data.getInstances(i, batchSize)
        gradient = neuralNetwork.getGradient(instances)
        weights -= stepSize * gradient
```

- It is important when implementing mini-batch gradient descent that you do not simply repeatedly randomly select N instances to do a weight update with as this could bias the results (not all samples will be evaluated equally).
- Instead first shuffle the data, and then progressively select batchSize instances. This ensures both random selections of instances (which can help break out of local minima) but also that samples are not over or under sampled.
- You will need to handle the edge case where the number of samples is not evenly divisible by the batch size.

Stochastic Gradient Descent

```
while(true):
```

```
    Instance instance = data.get_random_sample();  
    gradient = neural_network.get_gradient(instance);  
    weights -= step_size * gradient;
```

- Stochastic gradient descent (sometimes also called online gradient descent) selects a random sample at a time, calculates the gradient and does the weight update..
- Simpler to implement than batch gradient descent, but not as capable of taking advantage of modern GPUs.

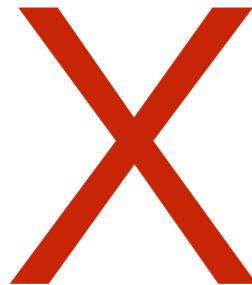
Pros:

- Simple to implement.
- Can train quickly (especially for non-CNNs) or training sets with less samples.

Cons:

- This is a stochastic method, the batches are randomly selected so even when starting with the same weights and inputs you can get different results.
- Not as efficient in some cases (esp CNNs) as mini-batch gradient descent.

Stochastic Gradient Descent



```
while (true):
    Instance instance = data.getRandomInstance();
    gradient = neuralNetwork.getGradient(instance)
    weights -= stepSize * gradient
```



```
while (true):
    //each iteration of the while loop is an epoch
    data.shuffle() //reshuffle the data before each epoch
    for (i = 0; i < numberSamples; i++):
        Instance instance = data.getInstance(i)
        gradient = neuralNetwork.getGradient(instance)
        weights -= stepSize * gradient
```

- Similar to mini-batch gradient descent we shouldn't just randomly sample from the training instances.
- For each epoch, shuffle the data and then go through every training sample.

Some Gradient Descent Notes

- A common method to somewhat unify the learning rate is to divide it by the number of samples used:
 - stochastic gradient descent uses the learning rate, lr
 - minibatch gradient descent uses $lr/batchSize$
 - batch gradient descent uses $lr/numberInstances$
- In practice this actually does not work very well due to how the gradients sum up across the training examples, so typically each method will have its own “optimal” learning rate.

Weight Initialization

Weight Initialization

- Before we get to tricks for improving our gradient descent we also need to think about how we initialize the weights.
- We actually have more than a few options.

Weight Initialization: All Zeros

- This is another example of what **not** to do.
- Maybe you take the assumption that in a properly trained network with a properly normalized data (more on normalization next week) set we might expect to have approximately half the weights positive and half the weights negative with approximately the same magnitudes.
- If that's the case, setting everything to 0 might seem sensible.
- Unfortunately, this will cause all the neurons to compute the same outputs, and every neuron will have the same gradients in backprop and have the exact same weight updates, so the neurons will not differentiate and learn different things.
- Zero weights are also bad because they will propagate 0 into the preceding node, zeroing any further gradients.
- So we need some source of asymmetry in our weights!

Weight Initialization: Small Random Values

- So we want our weights close to zero, but we also need a source of **asymmetry** to them. If we think about a value close to zero going into a sigmoid or tanh function this is the "sweet spot" of either function - the gradient is close to 1, and it's right in the middle of its potential outputs minimizing bias to a large or small output.
- A good way to do this is to use a normal (or gaussian) distribution with a mean of zero and standard deviation of 1 (i.e., a unit standard deviation), and then multiply it by a number < 1 to make it smaller, typically:

```
w[i] = 0.01 * rand_normal_distribution(0, 1)
```

Weight Initialization: Small Random Values

- Note that close to zero isn't *always* good. Remember that gradients can vanish very quickly, so if we have a lot of very small weights our gradients will vanish that much quicker.
- Also, in regression problems initializing to very small weights may also slow down learning.

Weight Initialization: Calibration

```
w[i] = 0.01 * rand_normal_distribution(0, 1)
```

- This approach still has an issue. Lets think about the variance of the inputs to a given node.
- If one node has 100 input edges, and another has 5:
 - For both, the mean of all the incoming weights will be 0 due to the random normal distribution.
 - However, the variance of the incoming weights to the node with 100 input edges will be much higher!
- This empirically will hinder learning and the convergence of gradient descent.

Weight Initialization: Calibration

- We can consider a node's *fan in* and *fan out*:
 - *fan in*: the number of incoming edges
 - *fan out*: the number of outgoing edges
- We can use the fan in of a node to calibrate it's incoming edges. So for any given node, the weights of the incoming edges can be generated using the same normal distribution, but we can divide by $\text{sqrt}(n)$ where n is the fan in.
- This will ensure that each node now has the same variance for it's incoming weights and empirically this improves the convergence rate.

Weight Initialization: Bias Initialization

- Biases can be initialized differently. Some options (as opposed to random) are:
 - Initialize all to 0 - this is okay as long as our weights aren't zero and breaking asymmetry.
 - Initialize all to 0.1 or another small value - this is nice for ReLUs because it can ensure each node fires initially (i.e., its incoming value is not ≤ 0); however there hasn't been any clear studies showing this is any better (at least for CNNs).

Lecture 4

Data Preprocessing and Training Tricks

DSCI-789: Neural Networks for Data Science

Travis Desell (tjdvse@rit.edu)
Associate Professor
Department of Software Engineering



ROCHESTER INSTITUTE OF TECHNOLOGY

Overview

- Training, Validation and Testing data
- Data Preprocessing
 - Mean Subtraction
 - Normalization
 - One-Hot Encoding
 - PCA and Whitening
- Training Tricks
 - Momentum
 - Annealing the Learning Rate
 - Per-Parameter Adaptive Learning Rates

Training, Validation and Testing Data

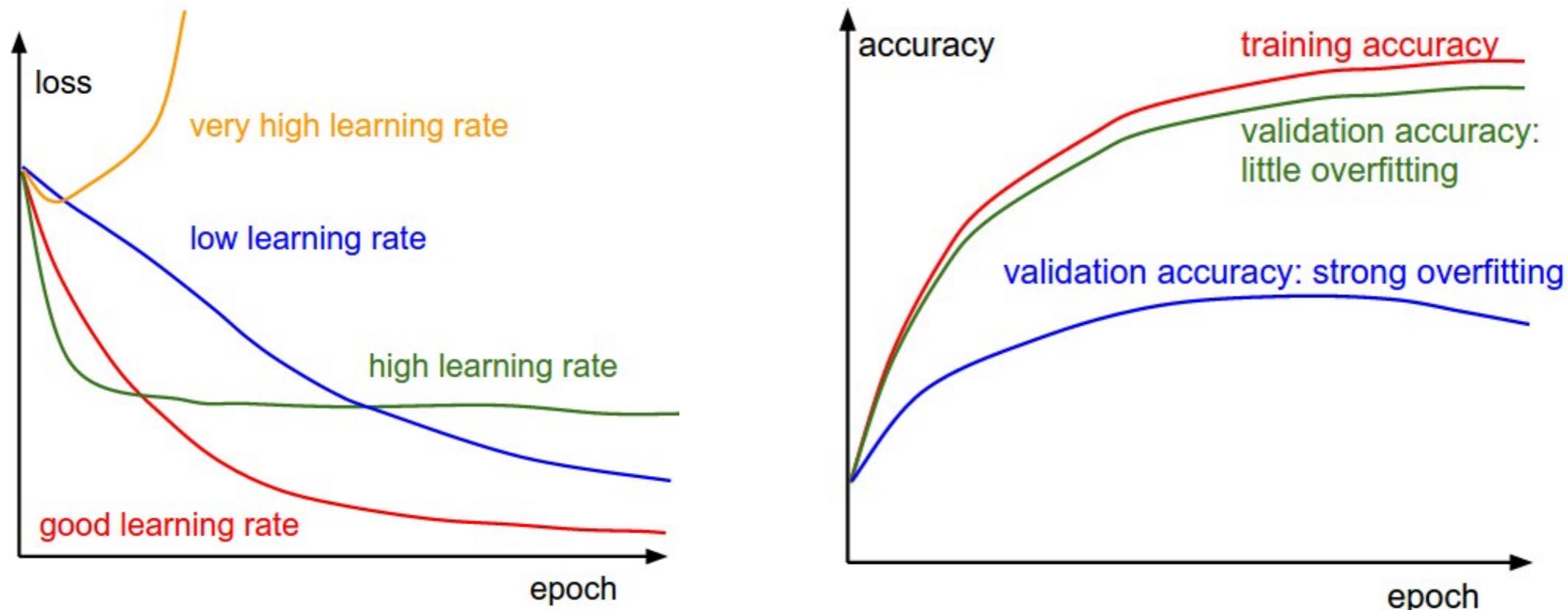
Training, Validation and Test Data

- When using non-toy data sets we typically want to split our data up into three different sets:
 - *training data*: this is the data we train our neural network on.
 - *validation data*: this is additional data we evaluate our neural network on during the training process to make sure we are not overfitting (i.e., our network is generalizing well. We will also use this to validate hyperparameters.
 - *test data*: this is the data we leave aside, and after everything is done (hyperparameter optimization, training, etc) check to see how well our network works on unseen data.

Training, Validation and Test Data

- Often times you will see data only split up into training and test data, however this carries problems and risk.
- The dataset you determine your hyperparameters for (learning rate, network structure, mu, etc) can bias those values. So if you finely optimize your network and hyperparameters to the test data there is a chance you will not do so well on other unseen data.
- This is why it is the most rigorous to leave aside the test data and only check on that at the end (to make sure you haven't overfit your hyperparameters either).

Watching the Loss and Accuracy



- During training you want to watch (and plot) your loss and accuracy.
- Loss:
 - If your loss is linear, your learning rate is probably too low.
 - If your loss looks exponential, it is probably too high.
- Accuracy:
 - You need to make sure you are not overfitting on your data (which NNs can do surprisingly easily).
 - If your validation accuracy is wildly diverging from your training accuracy you need to apply remedies like regularization (L1/L2 regularization, dropout, etc -- more on those later).

Images from CS234n: <http://cs231n.github.io/neural-networks-2/>

DSCI-789: Neural Networks for Data Science
2020 Spring

Ratios of Weights to Updates

```
paramScale = norm(weights) //usually L1 norm  
updateScale = norm(-learningRate * gradient)
```

ratio = updateScale / paramScale

- You can also track the ratio of the weight updates to the weights.
- If your weight updates are close to or larger than the weights then your position in the search space is bouncing around noisily.
- Ideally you should be making small updates to the weights and a good ratio is ~1e-3: 
 - If it is lower, the learning rate might be too low.
 - If it is higher, the learning rate is probably too high.
- For RNNs we can use modifications on this (gradient clipping and gradient boosting) where we scale the weight update up or down depending on how large it's norm is.

Class Imbalance

- Another thing to keep in mind is class balance. Let's take a simple example:
 - Class 1 - 10000 samples
 - Class 2 - 100 samples
- We need to be careful in training here. If we do standard epochs we'll be sampling from class 1 100 times more than from class 2. If the neural network just always predicts class 1 we'll have 99% accuracy even though we haven't learned anything.

Class Imbalance

- Class 1 - 10000 samples
- Class 2 - 100 samples
- There are a couple methods to work with here:
 - Upsample class 2 - for every epoch select from class 2 100 times more (e.g., make 100 copies of class 2 and shuffle them in with everything else).
 - Downsample from class 1 - do smaller epochs where you sample all 100 from class 2 but only 100 from class 1. Make sure you shuffle class 1 every pass through all of class 1 (not after every pass through 100).

Class Imbalance

- Class 1 - 10000 samples
- Class 2 - 100 samples
- There are even more advanced methods like SMOTE [1] which use distortions and permutations (usually of imagery) to make synthetic samples.

[1] <https://arxiv.org/pdf/1106.1813.pdf>

Class Imbalance - Training/Validation/Test Data

- Class 1 - 10000 samples 
- Class 2 - 100 samples
- You also need to be careful when generating your test data and validation sets. You cannot sample randomly from the entire data set, in the above case your test and validation set could easily be entirely class 1.
- Ideally you have the same number of samples from each class in your test and validation data. But in cases of extreme imbalance this isn't possible, so at the very least keep the ratio the same and be very careful about analyzing your accuracy.

Data Preprocessing

Data Preprocessing

- There are a number of ways to preprocess data:
 - Converting categorical data to a one-hot encoding.
 - Mean Subtraction
 - Normalization
 - PCA and Whitening

One Hot Encoding



```
p,x,s,n,t,p,f,c,n,k,e,e,s,s,w,w,p,w,o,p,k,s,u  
e,x,s,y,t,a,f,c,b,k,e,c,s,s,w,w,p,w,o,p,n,n,g  
e,b,s,w,t,l,f,c,b,n,e,c,s,s,w,w,p,w,o,p,n,n,m  
p,x,y,w,t,p,f,c,n,n,e,e,s,s,w,w,p,w,o,p,k,s,u  
e,x,s,g,f,n,f,w,b,k,t,e,s,s,w,w,p,w,o,e,n,a,g  
e,x,y,y,t,a,f,c,b,n,e,c,s,s,w,w,p,w,o,p,k,n,g  
e,b,s,w,t,a,f,c,b,g,e,c,s,s,w,w,p,w,o,p,k,n,m
```

- Above is example data from the agaricus-leiota.data file from the UCI Machine Learning Repository [1] which you'll need to pre-process as part of your next programming assignment.

[1] <https://archive.ics.uci.edu/ml/index.php>

One Hot Encoding

- If we look at the data, there are two output classes (e and p)
- Each column contains a varying number of classes as well.
- The number of classes per attribute is: 1(6), 2(4), 3(10), 4(2), 5(9), 6(4), 7(3), 8(2), 9(12), 10(2), 11(7), 12(4), 13(4), 14(9), 15(9), 16(2), 17(4), 18(3), 19(8), 20(9), 21(6), 22(7).
- We can convert this from having 22 data points per sample, each with a category, to (in this case) to 126 binary values.

7. Attribute Information: (classes: edible=e, poisonous=p)
1. cap-shape:
2. cap-surface:
3. cap-color:
4. bruises?:
5. odor:
6. gill-attachment:
7. gill-spacing:
8. gill-size:
9. gill-color:
10. stalk-shape:
11. stalk-root:
12. stalk-surface-above-ring:
13. stalk-surface-below-ring:
14. stalk-color-above-ring:
15. stalk-color-below-ring:
16. veil-type:
17. veil-color:
18. ring-number:
19. ring-type:
20. spore-print-color:
21. population:
22. habitat:

bell=b,conical=c,convex=x,flat=f,
knobbed=k,sunken=s
fibrous=f,grooves=g,scaly=y,smooth=s
brown=n,buff=b,cinnamon=c,gray=g,green=r,
pink=p,purple=u,red=e,white=w,yellow=y
bruises=t,no=f
almond=a,anise=l,creosote=c,fishy=y,foul=f,
musty=m,none=n,pungent=p,spicy=s
attached=a,descending=d,free=f,notched=n
close=c,crowded=w,distant=d
broad=b,narrow=n
black=k,brown=n,buff=b,chocolate=h,gray=g,
green=r,orange=o,pink=p,purple=u,red=e,
white=w,yellow=y
enlarging=e,tapering=t
bulbous=b,club=c,cup=u,equal=e,
rhizomorphs=z,rooted=r,missing=?
fibrous=f,scaly=y,silky=k,smooth=s
fibrous=f,scaly=y,silky=k,smooth=s
brown=n,buff=b,cinnamon=c,gray=g,orange=o,
pink=p,red=e,white=w,yellow=y
brown=n,buff=b,cinnamon=c,gray=g,orange=o,
pink=p,red=e,white=w,yellow=y
partial=p,universal=u
brown=n,orange=o,white=w,yellow=y
none=n,one=o,two=t
cobwebby=c,evanescent=e,flaring=f,large=l,
none=n,pendant=p,sheathing=s,zone=z
black=k,brown=n,buff=b,chocolate=h,green=r,
orange=o,purple=u,white=w,yellow=y
abundant=a,clustered=c,numerous=n,
scattered=s,several=v,solitary=y
grasses=g,leaves=l,meadows=m,paths=p,
urban=u,waste=w,woods=d

[1] <https://archive.ics.uci.edu/ml/index.php>

One Hot Encoding

```
p,x,s,n,t,p,f,c,n,k,e,e,s,s,w,w,p,w,o,p,k,s,u  
e,x,s,y,t,a,f,c,b,k,e,c,s,s,w,w,p,w,o,p,n,n,g  
e,b,s,w,t,l,f,c,b,n,e,c,s,s,w,w,p,w,o,p,n,n,m  
p,x,y,w,t,p,f,c,n,n,e,e,s,s,w,w,p,w,o,p,k,s,u  
e,x,s,g,f,n,f,w,b,k,t,e,s,s,w,w,p,w,o,e,n,a,g  
e,x,y,y,t,a,f,c,b,n,e,c,s,s,w,w,p,w,o,p,k,n,g  
e,b,s,w,t,a,f,c,b,g,e,c,s,s,w,w,p,w,o,p,k,n,m
```

- To take an example, the first column can have the values:
bell=b,conical=c,convex=x,flat=f,knobbed=k,sunken=s
- We expand this into 6 columns where the first is yes (1) or no (0) if the value is b, the second is yes (1) or no (0) if the value is c, the third is yes (1) or no (0) if the value is x, and so on.
- We do this for each column in the categorical data. While we can end up with significantly more input values, networks will learn much better than if you instead kept things as a single value and had 0 = b, 1 = c, 2 = x, 3 = f, etc., because in reality these different categories are not related. This will actually mean that b and x are more similar to c than they are to s.

[1] <https://archive.ics.uci.edu/ml/index.php>

Mean Subtraction

```
5.1,3.5,1.4,0.2,Iris-setosa  
4.9,3.0,1.4,0.2,Iris-setosa  
4.7,3.2,1.3,0.2,Iris-setosa  
4.6,3.1,1.5,0.2,Iris-setosa  
5.0,3.6,1.4,0.2,Iris-setosa  
5.4,3.9,1.7,0.4,Iris-setosa  
4.6,3.4,1.4,0.3,Iris-setosa
```

- In the case of non-categorical real valued data (like the iris data set above, also from the UCI ML repository) other normalizations are useful.

[1] <https://archive.ics.uci.edu/ml/index.php>

Mean Subtraction

```
5.1,3.5,1.4,0.2,Iris-setosa  
4.9,3.0,1.4,0.2,Iris-setosa  
4.7,3.2,1.3,0.2,Iris-setosa  
4.6,3.1,1.5,0.2,Iris-setosa  
5.0,3.6,1.4,0.2,Iris-setosa  
5.4,3.9,1.7,0.4,Iris-setosa  
4.6,3.4,1.4,0.3,Iris-setosa
```

Summary Statistics:

	Min	Max	Mean	SD	Class	Correlation
sepal length:	4.3	7.9	5.84	0.83		0.7826
sepal width:	2.0	4.4	3.05	0.43		-0.4194
petal length:	1.0	6.9	3.76	1.76		0.9490 (high!)
petal width:	0.1	2.5	1.20	0.76		0.9565 (high!)

- From the data file (iris.names) we can see that the different real values are not centered around 0. If we think about our various activation functions (sigmoid, tanh) having such large values as input can lead to low gradients and will also bias our outputs towards 1.
- Subtracting the mean of each parameter will center it around zero making the learning process easier.

[1] <https://archive.ics.uci.edu/ml/index.php>

Normalization

```
5.1,3.5,1.4,0.2,Iris-setosa  
4.9,3.0,1.4,0.2,Iris-setosa  
4.7,3.2,1.3,0.2,Iris-setosa  
4.6,3.1,1.5,0.2,Iris-setosa  
5.0,3.6,1.4,0.2,Iris-setosa  
5.4,3.9,1.7,0.4,Iris-setosa  
4.6,3.4,1.4,0.3,Iris-setosa
```

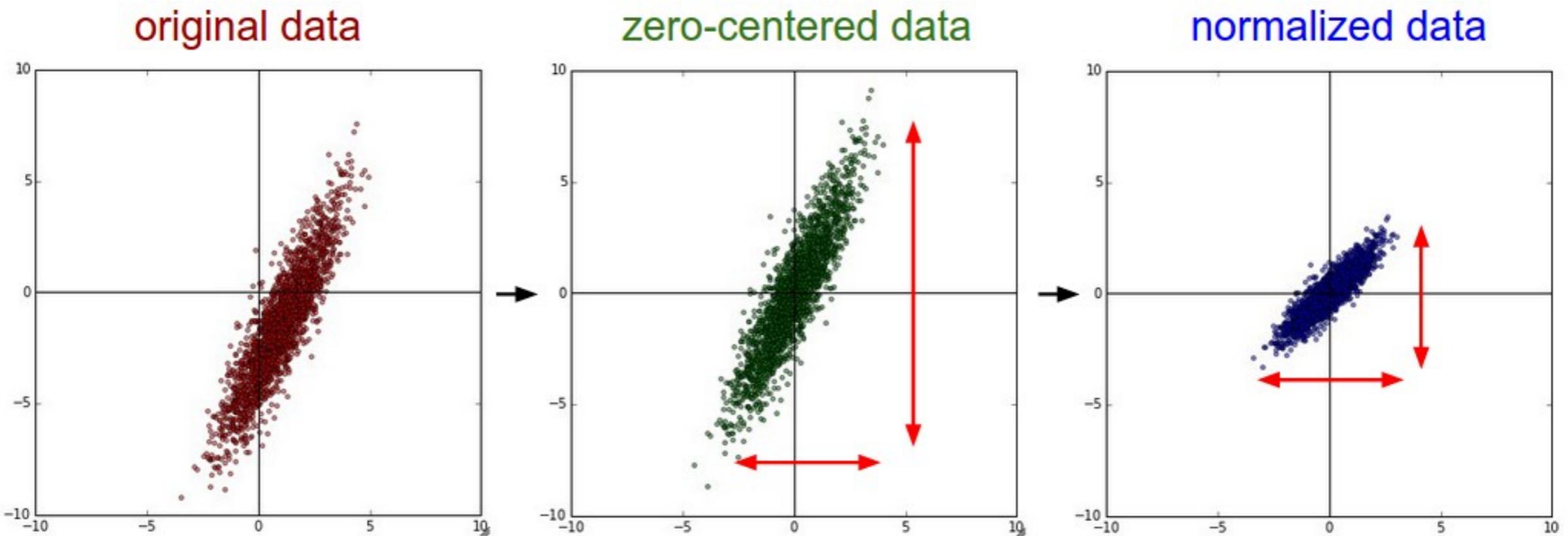
Summary Statistics:

	Min	Max	Mean	SD	Class	Correlation
sepal length:	4.3	7.9	5.84	0.83		0.7826
sepal width:	2.0	4.4	3.05	0.43		-0.4194
petal length:	1.0	6.9	3.76	1.76		0.9490 (high!)
petal width:	0.1	2.5	1.20	0.76		0.9565 (high!)

- Beyond mean subtraction, each different parameter (or feature) has a different standard deviation. This means that changes to weights for certain inputs will have greater effects on the output, and likewise the varying inputs will have greater or lesser effects on their outputs due to their distance from zero.
- By dividing each parameter (after mean subtraction) by its standard deviation it makes sure that each parameter will have an equal importance (at least initially) in the learning process.

[1] <https://archive.ics.uci.edu/ml/index.php>

0-1 Normalization



- To visualize things, if we have two parameters (x and y):
 - mean subtraction zero centers the data.
 - division by standard deviation normalizes (unifies the scale) of the data.
- You almost always want to do both.

Image from CS234n: <http://cs231n.github.io/neural-networks-2/>

0-1 Normalization

$$p_i = \frac{p_i - \min(\forall p_i)}{\max(\forall p_i) - \min(\forall p_i)}$$

- Another form of normalization is to scale the parameters between 0 and 1 (or sometimes -1 and 1). This is more common in preparing time series data for prediction by recurrent neural networks.
- For a given parameter column i (p_i) you subtract by the minimum value for that parameter and then divide by the maximum minus the minimum value of that parameter. This re-scales the values between 0 and 1; however it does not give every parameter the same standard deviation -- which retains information about the relative magnitudes of each parameter.

PCA

$$X = X - \text{mean}(X)$$
$$C = \frac{X^\top \cdot X}{n}$$

- In this case, X is an n by d matrix of our data, it has n samples each with d parameters.
- For principal component analysis (PCA) we first zero-center the data, and then compute the covariance matrix C .
- In the covariance matrix, the (i,j) element is the covariance between the i^{th} and j^{th} dimension (parameters).

PCA

$$U, S, V = svd(C)$$

$$X_{rot} = X \cdot U$$

- Following this we perform a singular value decomposition (SVD) factorization on the covariance matrix to get U, S and V.
- The columns of U are the eigenvectors¹ of X and S is a 1-dimensional array of the singular values.
- X_{rot} is now the decorrelated version of X.

¹An eigenvector is a characteristic vector of a linear transformation is a nonzero vector that changes at most by a scalar factor when that linear transformation is applied to it.

PCA Dimensionality Reduction

$$U, S, V = svd(C)$$

$$X_{rot} = X \cdot U[n, 0..j]$$

- A nice property of SVD (as it is usually computed) is that in U , the eigenvector columns are sorted by their eigenvalues, so they are ordered by the dimensions that have the most variance.
- By selecting only the first j columns in U , we can discard the dimensions that have the least variance performing a dimensionality reduction.
- This will reduce the data matrix X from being n by d to being n by j .

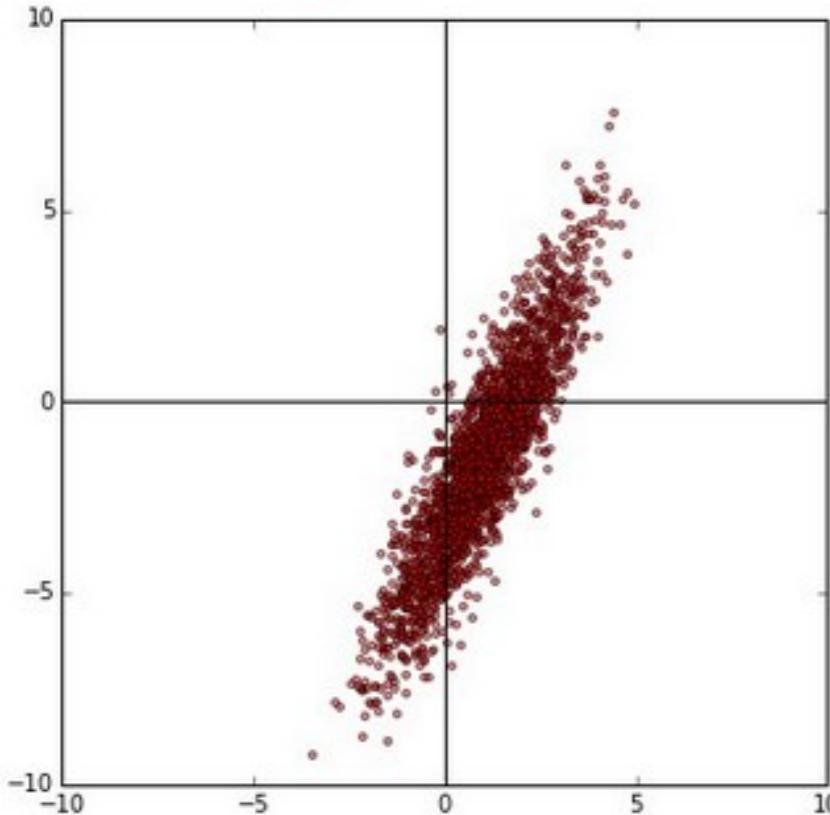
Whitening

$$X_{white} = \frac{x_{rot}}{\sqrt{S+1e^{-5}}}$$

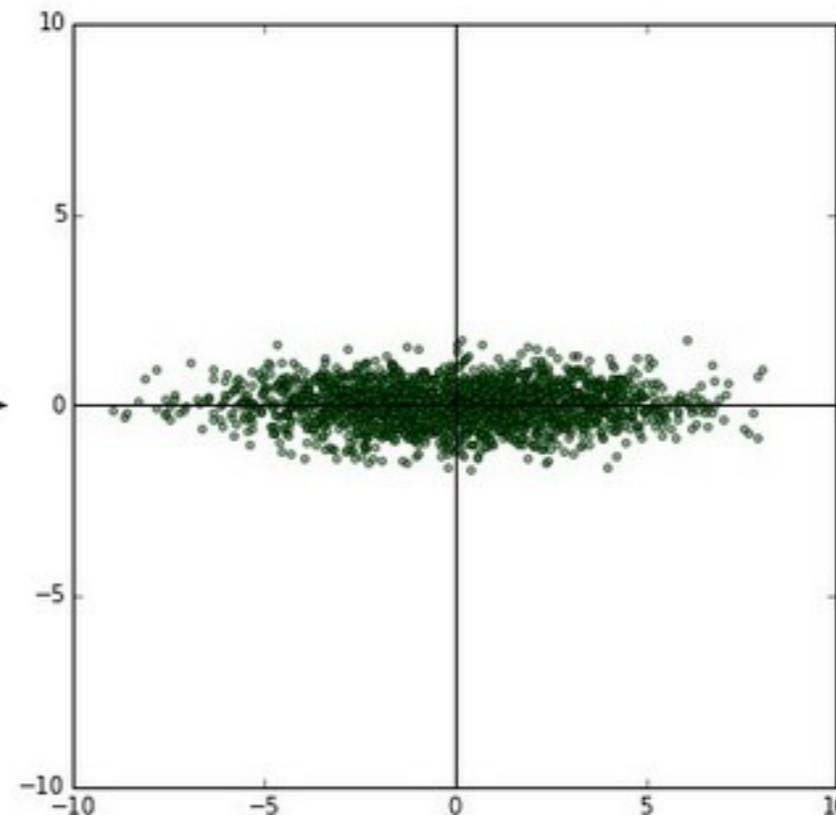
- To whiten the decorrelated PCA data (X_{rot}) we then divide it by the eigenvalues, which we compute from S .
- This basically means if the data is normally distributed (it is a multivariable gaussian) then the whitened data will have a zero mean and identity covariance matrix (i.e., all dimensions/parameters now have the same covariance).
- We typically add a small value (1e-5) to prevent division by zero, unfortunately this can really exaggerate the noise in the data however this can be mitigated by using a larger number.

PCA and Whitening

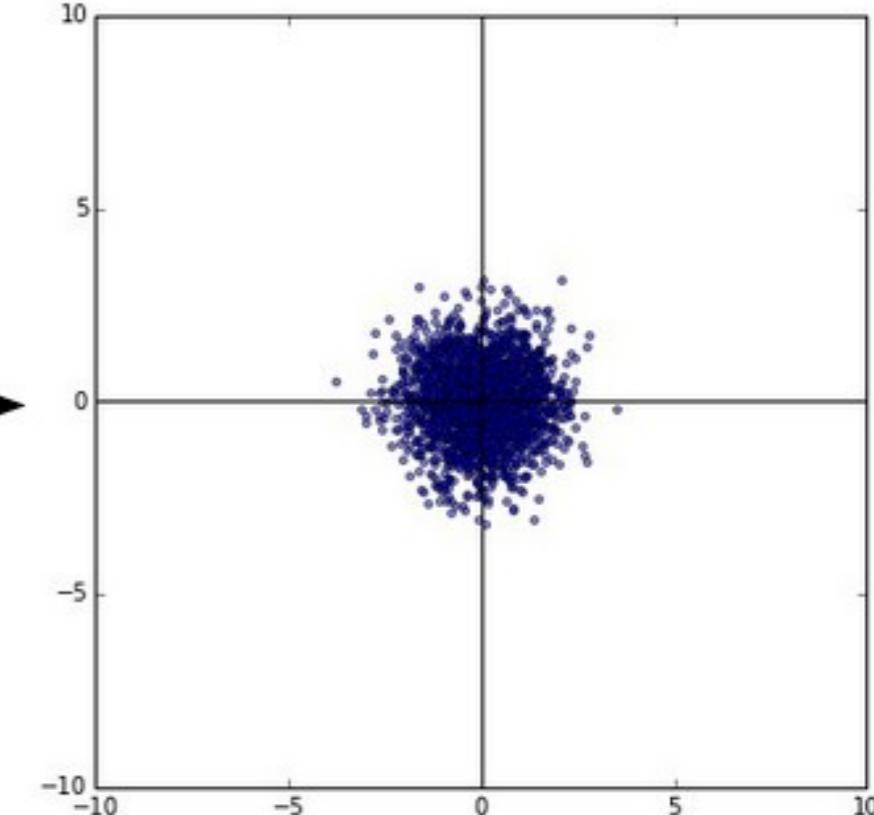
original data



decorrelated data



whitened data



- So to visualize this:
 - PCA decorrelates the data
 - Whitening stretches/squeezes the data to be a uniform gaussian (normally distributed) blob.

Image from CS234n: <http://cs231n.github.io/neural-networks-2/>

What To Use When

- Generally speaking:
 - Use one-hot encoding if you have categorical data.
 - Use mean subtraction and standard deviation normalization if you are doing classification and have real valued data. This is very common to use on the pixel values before feeding them into convolutional neural networks.
 - Use 0-1 normalization if doing time series data prediction with real-valued data.
 - PCA and dimensionality reduction can be useful if you have categorical data with a large number of parameters -- it is very common in NLP. Whitening may help here as well.

Normalization with Training, Validation and Testing Data

- It is extremely important that when doing normalization or any pre-processing; you calculate the appropriate values (mins, maxes, standard deviation) on the **training data only** and then use the values from the training data on the test and validation data.
- **DO NOT INCLUDE THE VALIDATION AND TEST DATA IN THE PRE-PROCESSING.**
- The *only* exception to this is in some cases doing time series data prediction you can use domain knowledge to know an exact minimum and maximum value a parameter can have, and then use those for 0-1 normalization.

Training Tricks: Momentum

Vanilla Update

```
weights -= learningRate * gradient
```

- Up until now we've been using the "vanilla" way to update the weights, we simply subtract the gradient by a small learning rate.
- While this guarantees that if our learning rate is small enough and our gradient is correct and evaluated on the full data set we will always improve our error until we find a minimum - we can actually improve on this in a few ways.
- One way is through using **momentum**.

Momentum Update

```
velocity = mu * velocityPrev - learningRate * gradient  
weights += velocity  
velocityPrev = velocity
```

- The motivation behind velocity is the same as a ball rolling down a hill. As it continues downwards it picks up velocity speeding up as long as it's traveling on a downward slope.
- In this case, we have *mu* as a new hyperparameter which essentially serves as *friction*. For each gradient calculation we now also add the previous weight update scaled by mu (typically 0.9 but other values commonly used are 0.5, 0.95 and 0.99).
- If the gradients keep pointing in the same direction, they will accumulate and speed up moving that way.

Momentum Update

velocity = mu * velocityPrev - learningRate * gradient

weights += velocity

velocityPrev = velocity

mu = 0.9

lr = 0.3

g1 = [1.0, 1.0, 1.0]

v1 = [0.0 - 0.3, 0.0 - 0.3, 0.0 - 0.3]

w += [-0.3, -0.3, -0.3]

//do forward pass and backward pass to calculate new gradient

g2 = [1.0, 1.0, 1.0]

v2 = [-0.27 - 0.3, -0.27 - 0.3, -0.27 - 0.3]

w += [-0.57, -0.57, -0.57]

//do forward pass and backward pass to calculate new gradient

g3 = [1.0, -1.0, 1.0]

v3 = [-0.513 - 0.3, -0.513 + 0.3, -0.57 - 0.3]

w += [-0.813, -0.213, -0.85]

...

Here's a simple example, where we in three steps gradient descent we got gradients of [1, 1, 1], [1, 1, 1] and [1, -1, 1].

The weight updates in the direction of the gradient keep increasing, and when we start to get a different gradient then they will start decreasing once the prior momentum runs out.

Momentum Update

```
velocity = mu * velocityPrev - learningRate * gradient
```

```
weights += velocity
```

```
velocityPrev = velocity
```

- You need to make sure you keep the momentum < 1 , otherwise your gradients will explode (they will keep getting larger and larger).
- Another thing we can do (more on this later) is to *anneal* the μ parameter. Typically we want it smaller at the beginning of gradient descent (when the gradients are larger and the search space a bit more chaotic), and then increase it as we get closer to the optimum, where gradients are smaller. This can also help us break out of valleys.

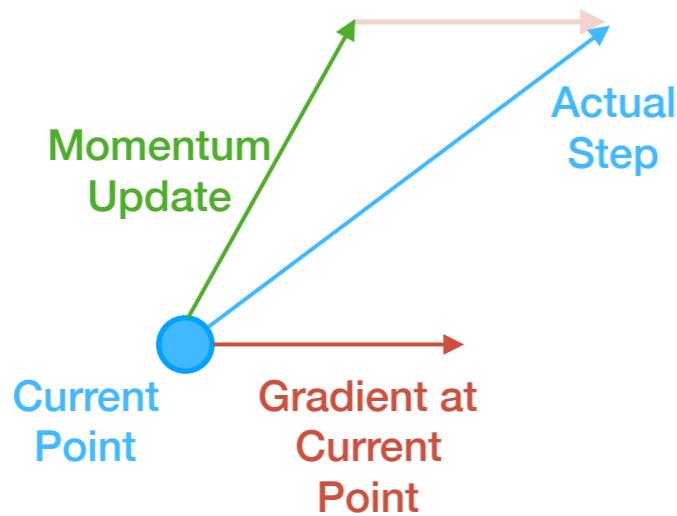
Nesterov Momentum

- We can improve further over a regular momentum update.
- *Nesterov momentum* provides better theoretical guarantees on convex functions (functions without local minima) and in practice almost always works better.

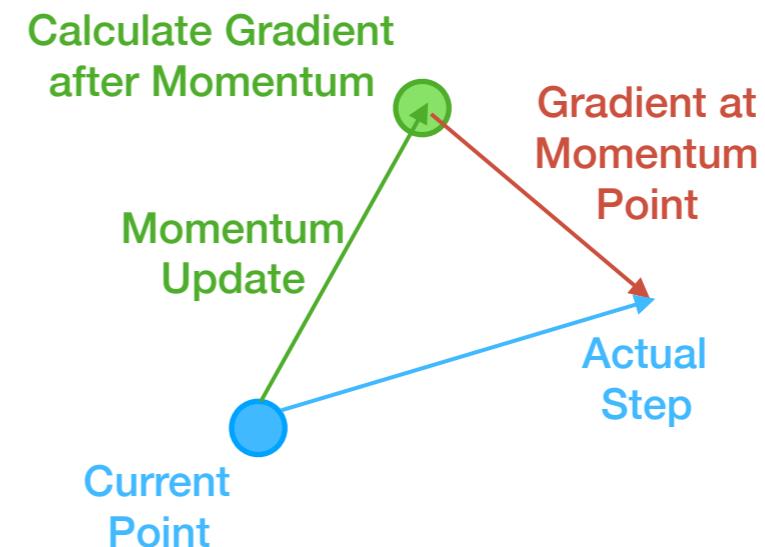
Nesterov Momentum

- The general idea behind Nesterov momentum is that we know where momentum is going to move us, so instead of calculating the gradient at the current x , we calculate it where we'll be moved by momentum and then take a step:

Standard Momentum



Nesterov Momentum



Nesterov Momentum

```
weightsNext = weights + mu * velocity  
//calculate gradient at weightsNext instead of x  
velocity = mu * velocity - learningRate * gradientNext  
weights += velocity
```

- We can actually reformulate this so that it looks similar to a regular momentum or vanilla update:

```
velocityPrev = velocity  
velocity = mu * velocity - learningRate * gradient  
weights += (-mu * velocityPrev) + ((1 + mu) * velocity)
```

- What this essentially does is calculate the velocity for the next version of the weights, but then do the parameter update for the current weights.

Training Tricks: Annealing the Learning Rate

Learning Rate Annealing

- Annealing in engineering is to and allow a glass or metal to cool slowly, in order to remove internal stresses and toughen it.
- In the machine learning context, it means to gradually reduce a hyperparameter (sometimes increase) during the learning process to achieve better results.

Learning Rate Annealing

- There are three commonly used methods of *annealing* the learning rate:
 - Step Decay
 - Exponential Decay
 - $1/t$ Decay

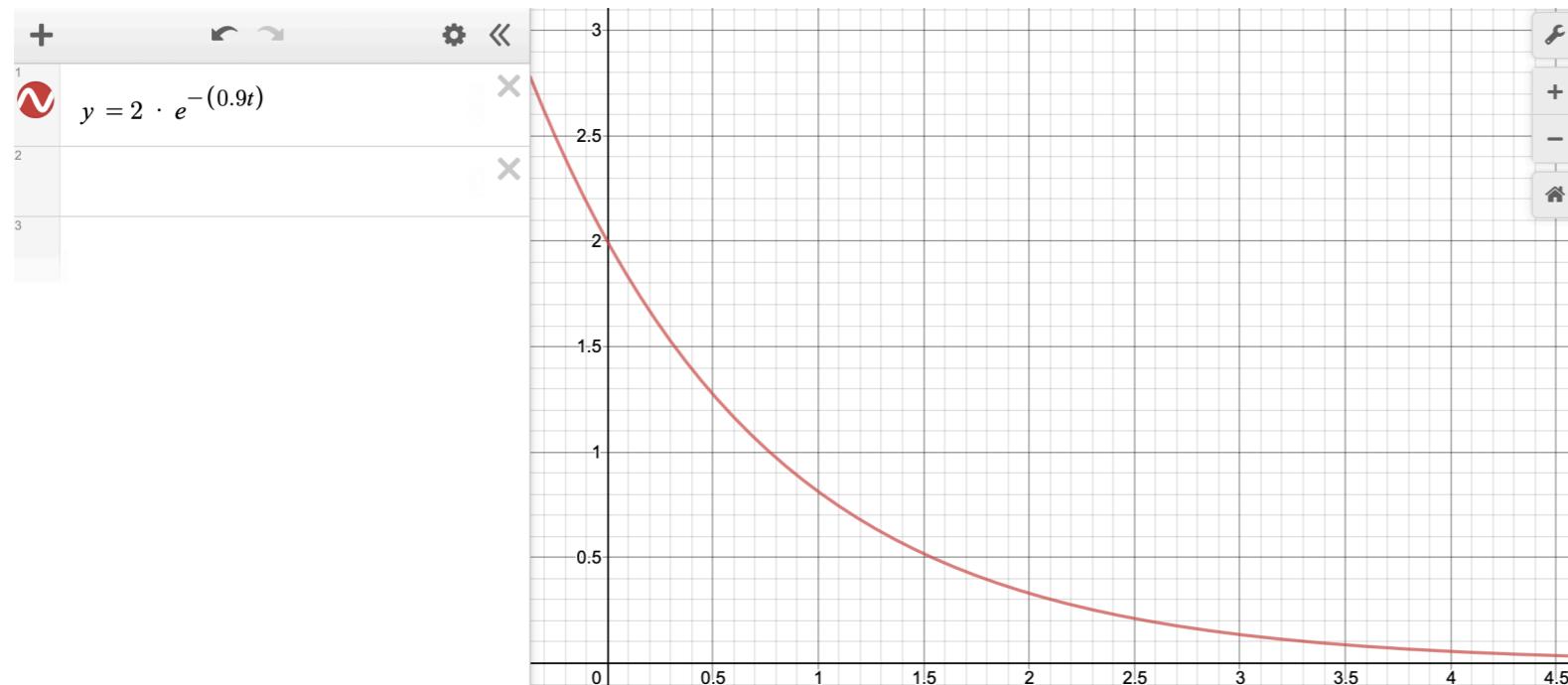
Learning Rate Annealing: Step Decay

```
if ((epoch % step) == 0) {  
    learningRate *= annealingRate;  
}
```

- Step decay is the simplest method, every `ew` epochs reduce the learning rate.
- Typical strategies include reducing the learning rate by half every 5 epochs, or by 0.1 every 20 epochs.
- You may also see people reducing the learning rate when the validation error (more on that later) stops improving.

Learning Rate Annealing: Exponential Decay

$$\text{learningRate} = \text{initialRate} * e^{(-k*t)}$$



$$k = 0.9$$
$$\text{initialRate} = 1$$

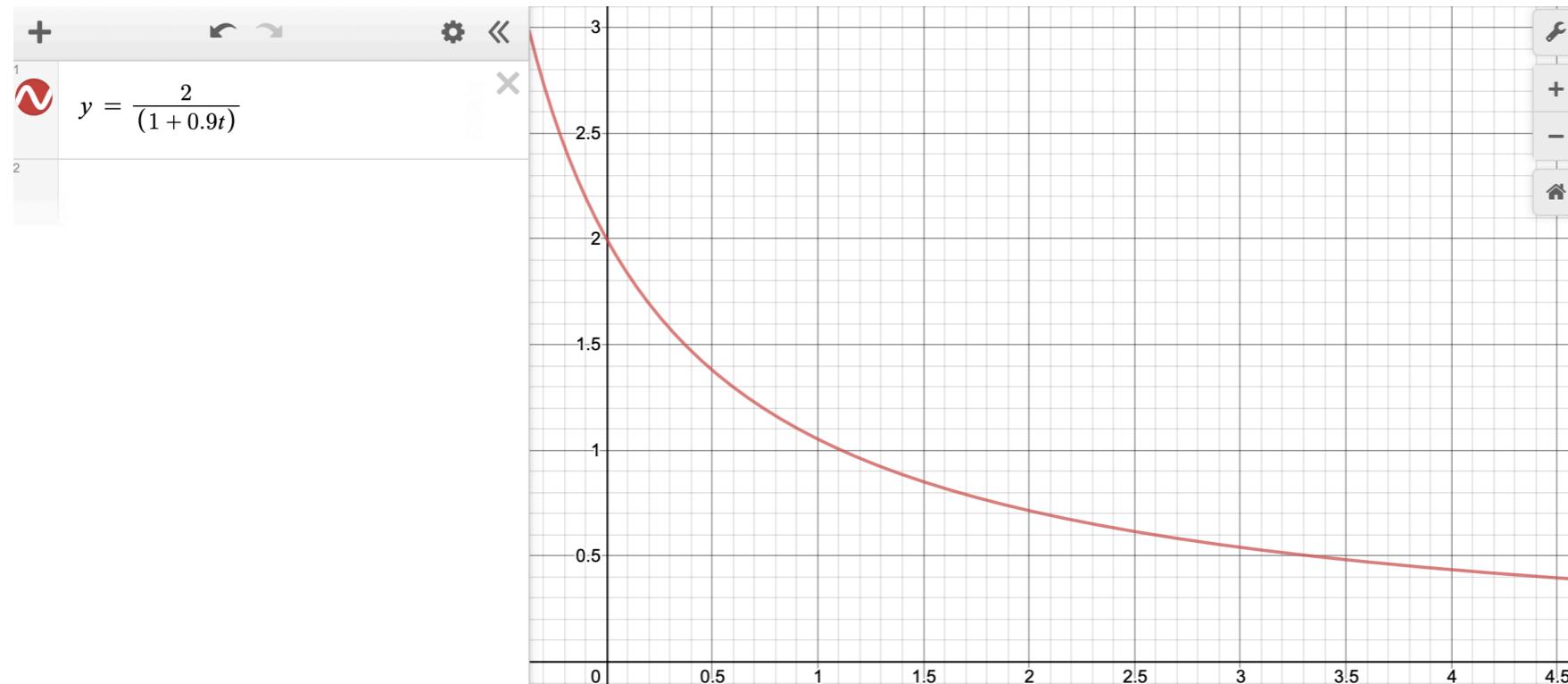
$$\text{lr}_0 = 1$$
$$\text{lr}_1 = 0.4065$$
$$\text{lr}_2 = 0.1652$$
$$\text{lr}_3 = 0.0672$$
$$\text{lr}_4 = 0.0273$$
$$\text{lr}_5 = 0.0111$$

...

- This method uses two hyperparameters, an initial learning rate, and k. t is either the iteration number or epoch (the first value of t is 0, which means the first learning rate will be set to initialRate because $e^0 = 1$).
- This progressively reduces the learning rate towards 0 (but never to actually 0).

Learning Rate Annealing: 1/t Decay

$$\text{learningRate} = \text{initialRate}/(1 + kt)$$



$$k = 0.9$$
$$\text{initialRate} = 2$$

$$\begin{aligned} lr_0 &= 2 \\ lr_1 &= 1.0526 \\ lr_2 &= 0.7142 \\ lr_3 &= 0.5405 \\ lr_4 &= 0.4347 \\ lr_5 &= 0.3636 \\ \dots \end{aligned}$$

- This method is similar, with similar hyperparameters but the reduction in the learning rate is not as swift as exponential decay.

Training Tricks: Per-parameter adaptive learning rates

Per-parameter Adaptive Learning Rates

- A few common per-parameter learning methods are:
 - Adagrad
 - RMSprop
 - Adam
- These are more advanced than the previous rates as they calculate a different learning rate *for each parameter* in the gradient.

Adagrad

```
cache[i] += gradient[i]2
```

```
weight[i] -= (learningRate / (sqrt(cache[i]) + eps)) * gradient[i]
```

- Note that here we add an additional cache variable (with the same number of elements as the gradient/weights) and keep track of the sum of squares of the previous gradients.
- eps is the hyperparameter, typically small ($1\text{e-}4$ to $1\text{e-}8$)
- Each weight update we scale the learning rate down by the square root of the cache value plus some epsilon (which prevents dividing by zero).
- A potential downside to adagrad is that it monotonically reduces the learning rate and this may be too aggressive especially for deep learning.

RMSprop

```
cache[i] = decayRate * cache[i] + (1 - decayRate) * gradient[i]2
weight[i] -= (learningRate / (sqrt(cache[i]) + eps)) * gradient[i]
```

- RMSprop adjusts adagrad so that it doesn't have a monotonically decreasing learning rate.
- Instead of the cache containing the sum of squares of all previous gradients, it instead uses a moving average, so the scaling of the learning rate is more based on the recent gradients.

Interesting aside: this method was not published, so everyone just cites Geoffrey Hinton's Coursera class (slide 29 of Lecture 6):
http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Adam

$$m[i] = \beta_1 * m[i] + (1 - \beta_1) * \text{gradient}[i]$$

$$v[i] = \beta_2 * v[i] + (1 - \beta_2) * \text{gradient}[i]^2$$

$$\text{weights}[i] \leftarrow \text{learningRate} * m[i] / (\sqrt{v[i]} + \epsilon)$$

- Adam is an effort to combine RMSprop with momentum, and often works better. That being said stochastic (or minibatch) gradient descent and Nesterov momentum also can do just as well or better for some problems.
- Recommended hyperparameters are $\epsilon = 1e-8$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

Adam paper: <https://arxiv.org/abs/1412.6980>

Adam with Bias Correction

$$m[i] = \beta_1 * m[i] + (1 - \beta_1) * \text{gradient}[i]$$

$$mt[i] = m[i] / (1 - \beta_1^t)$$

$$v[i] = \beta_2 * v[i] + (1 - \beta_2) * \text{gradient}[i]^2$$

$$vt[i] = v[i] / (1 - \beta_2^t)$$

$$\text{weights}[i] \leftarrow \text{learningRate} * mt[i] / (\sqrt{vt[i]} + \epsilon)$$

- The full adam update also contains a bias correction (above).
- This compensates for the fact that the $m[i]$ and $v[i]$ are initialized to 0 and that introduces some bias.
- This essentially reduces the m and v values a bit initially and allows the method to "warm up" (as $1 - \beta_1^t$) will eventually be 1 (for a large t , where t is the iteration or epoch).

Adam paper: <https://arxiv.org/abs/1412.6980>

Lecture 5

Recurrent Neural Networks and the RNN Forward Pass

DSCI-789: Neural Networks for Data Science

Travis Desell (tjdvse@rit.edu)
Associate Professor
Department of Software Engineering



ROCHESTER INSTITUTE OF TECHNOLOGY

Overview

- What are RNNs?
- Traditional RNN Architectures
- A Generic Recurrent Connection Model
- RNN Forward Pass and Loss Functions

What are Recurrent Neural Networks (RNNs)?

What are RNNs?

- RNNs are neural networks designed to operate on sequential/temporal data.
- Until now the data we have worked with did not have any sequential or temporal correlation. Each training sample was independent.
- One of the most important features of RNNs is that they allow the same network to be trained on sequences of varying lengths. Other approaches (CNNs or linearizing the data into a single training sample) require all samples to be the same length.

What are they used for?

- Common use cases for RNNs include (but are not limited to):
 - Time series data prediction
 - Time series classification
 - Character/word prediction
 - Text classification
 - Generative models

Time Series Data Prediction

Alt A GL	E1 CHT1	E1 CHT2	E1 CHT3	E1 CHT4	E1 EGT1	E1 EGT2	E1 EGT3	E1 EGT4	E1 OilP	E1 OilT	E1 RPM	FQ tyL	FQ tyR	G nd Sp d	IA S	La tAc	No rm Ac	OA T	Pit ch	Ro II	TA S	vo lt1	vo lt2	VS pd	VS pd G
2.0	154.68	158.26	163.92	184.88	1118.15	1051.39	1079.77	1083.17	63.74	153.61	1225.8	19.48	21.04	0.0	0.0	-0.01	0.02	25.0	1.37	-0.07	0.0	28.0	27.9	-30.12	-3.9
2.0	155.15	158.59	164.2	185.28	1120.67	1054.03	1082.75	1086.21	63.3	153.56	1219.5	19.48	20.99	0.0	0.0	-0.01	0.0	25.0	1.4	-0.04	0.0	28.0	27.9	-35.91	-3.9
2.0	155.68	159.03	164.67	185.9	1118.25	1056.28	1083.58	1087.25	60.19	153.5	1083.6	19.48	21.04	0.0	0.0	0.0	0.02	25.0	1.47	-0.04	0.0	27.9	27.9	-25.92	-3.9
1.0	156.12	159.41	165.02	186.3	1105.33	1055.26	1076.52	1081.81	59.1	153.44	987.6	19.48	21.04	0.0	0.0	0.0	0.0	25.0	1.49	-0.04	0.0	27.9	27.9	-30.26	0.0
2.0	156.59	159.81	165.35	186.77	1092.08	1053.14	1065.49	1076.38	58.87	153.37	958.7	19.48	21.04	0.0	0.0	-0.01	0.0	25.0	1.51	0.0	0.0	27.8	27.8	-7.05	0.0
2.0	157.11	160.16	165.73	187.19	1079.67	1052.24	1056.0	1069.26	58.8	153.31	954.3	19.48	21.04	0.0	0.0	0.0	0.0	25.0	1.53	-0.01	0.0	27.8	27.8	-0.28	0.0
2.0	157.65	160.62	166.24	187.77	1068.77	1052.49	1047.71	1063.16	58.95	153.24	954.2	19.51	21.04	0.0	0.0	0.0	0.01	24.8	1.51	-0.03	0.0	27.9	27.8	-14.42	0.0
1.0	158.17	161.02	166.63	188.21	1059.82	1052.43	1040.47	1057.89	58.88	153.17	954.9	19.51	21.07	0.0	0.0	-0.01	0.02	24.8	1.49	-0.04	0.0	27.9	27.9	-15.95	0.0
2.0	158.66	161.41	166.98	188.7	1052.0	1051.41	1033.99	1052.41	58.95	153.11	957.8	19.51	21.07	0.0	0.0	-0.01	0.01	24.8	1.48	0.02	0.0	27.9	27.8	-14.68	-3.9
3.0	159.17	161.81	167.37	189.09	1045.21	1049.61	1029.56	1048.0	58.89	153.04	956.7	19.48	21.07	0.0	0.0	0.0	0.01	24.8	1.48	0.03	0.0	27.8	27.8	-51.43	0.0
3.0	159.69	162.2	167.76	189.6	1039.97	1048.27	1026.94	1044.84	58.86	152.97	956.0	19.51	21.07	0.0	0.0	-0.02	0.0	24.8	1.45	0.01	0.0	27.9	27.8	-40.56	-3.9
3.0	160.32	162.71	168.32	190.11	1037.56	1048.85	1022.7	1041.83	58.89	152.91	958.8	19.51	21.07	0.0	0.0	0.0	0.0	24.8	1.44	0.01	0.0	27.9	27.8	-28.76	0.0
3.0	160.98	163.23	168.87	190.71	1035.58	1050.04	1019.0	1039.76	59.06	152.84	963.2	19.56	21.07	0.0	0.0	0.0	0.01	24.5	1.47	-0.01	0.0	27.9	27.9	3.07	0.0

- Times series data typically comes from various sensors or data feeds.
- Time series data prediction involves predicting what the next value of one or more columns of data will be depending on all prior seen data points:
 - Given the above flight data, what will the engine's next RPM value be (highlighted in bold).
 - Given a set of stock ticker data from various companies, what will the next value of a particular company's stock be?

Time Series Data Prediction

Alt A GL	E1 CHT1	E1 CHT2	E1 CHT3	E1 CHT4	E1 EGT1	E1 EGT2	E1 EGT3	E1 EGT4	E1 OilP	E1 OilT	E1 RPM	FQ tyL	FQ tyR	G nd Sp d	IA S	La tAc	No rm Ac	OA T	Pit ch	Ro II	TA S	vo lt1	vo lt2	VS pd	VS pd G
2.0	154.68	158.26	163.92	184.88	1118.15	1051.39	1079.77	1083.17	63.74	153.61	1225.8	19.48	21.04	0.0	0.0	-0.01	0.02	25.0	1.37	-0.07	0.0	28.0	27.9	-30.12	-3.9
2.0	155.15	158.59	164.2	185.28	1120.67	1054.03	1082.75	1086.21	63.3	153.56	1219.5	19.48	20.99	0.0	0.0	-0.01	0.0	25.0	1.4	-0.04	0.0	28.0	27.9	-35.91	-3.9
2.0	155.68	159.03	164.67	185.9	1118.25	1056.28	1083.58	1087.25	60.19	153.5	1083.6	19.48	21.04	0.0	0.0	0.0	0.02	25.0	1.47	-0.04	0.0	27.9	27.9	-25.92	-3.9
1.0	156.12	159.41	165.02	186.3	1105.33	1055.26	1076.52	1081.81	59.1	153.44	987.6	19.48	21.04	0.0	0.0	0.0	0.0	25.0	1.49	-0.04	0.0	27.9	27.9	-30.26	0.0
2.0	156.59	159.81	165.35	186.77	1092.08	1053.14	1065.49	1076.38	58.87	153.37	958.7	19.48	21.04	0.0	0.0	-0.01	0.0	25.0	1.51	0.0	0.0	27.8	27.8	-7.05	0.0
2.0	157.11	160.16	165.73	187.19	1079.67	1052.24	1056.0	1069.26	58.8	153.31	954.3	19.48	21.04	0.0	0.0	0.0	0.0	25.0	1.53	-0.01	0.0	27.8	27.8	-0.28	0.0
2.0	157.65	160.62	166.24	187.77	1068.77	1052.49	1047.71	1063.16	58.95	153.24	954.2	19.51	21.04	0.0	0.0	0.0	0.01	24.8	1.51	-0.03	0.0	27.9	27.8	-14.42	0.0
1.0	158.17	161.02	166.63	188.21	1059.82	1052.43	1040.47	1057.89	58.88	153.17	954.9	19.51	21.07	0.0	0.0	-0.01	0.02	24.8	1.49	-0.04	0.0	27.9	27.9	-15.95	0.0
2.0	158.66	161.41	166.98	188.7	1052.0	1051.41	1033.99	1052.41	58.95	153.11	957.8	19.51	21.07	0.0	0.0	-0.01	0.01	24.8	1.48	0.02	0.0	27.9	27.8	-14.68	-3.9
3.0	159.17	161.81	167.37	189.09	1045.21	1049.61	1029.56	1048.0	58.89	153.04	956.7	19.48	21.07	0.0	0.0	0.0	0.01	24.8	1.48	0.03	0.0	27.8	27.8	-51.43	0.0
3.0	159.69	162.2	167.76	189.6	1039.97	1048.27	1026.94	1044.84	58.86	152.97	956.0	19.51	21.07	0.0	0.0	-0.02	0.0	24.8	1.45	0.01	0.0	27.9	27.8	-40.56	-3.9
3.0	160.32	162.71	168.32	190.11	1037.56	1048.85	1022.7	1041.83	58.89	152.91	958.8	19.51	21.07	0.0	0.0	0.0	0.0	24.8	1.44	0.01	0.0	27.9	27.8	-28.76	0.0
3.0	160.98	163.23	168.87	190.71	1035.58	1050.04	1019.0	1039.76	59.06	152.84	963.2	19.56	21.07	0.0	0.0	0.01	0.01	24.5	1.47	-0.01	0.0	27.9	27.9	3.07	0.0

- In some cases prediction tasks involve predicting multiple next time step values (e.g., RPM, AGL and CHTs bolded above).
- Some tasks may also involve predicting farther out into the future than the next seen value (the farther out the more difficult this becomes).

Time Series Classification

- Another task involving time series is classification.
- If we have multiple time series and we know their target classes, can we train a neural network to predict the class of things we haven't seen?
- This can be used for tasks such as predictive maintenance (e.g., some time series represent a mechanical failure while others don't).

Character/Word Prediction

aer banknote berlitz calloway centrust cluett fromstein gitano guterman hydro-quebec ipo kia memotec mlx nahb punts rake regatta rubens sim
snack-food ssangyong swapo wachter
pierre <unk> N years old will join the board as a nonexecutive director nov. N
mr. <unk> is chairman of <unk> n.v. the dutch publishing group
rudolph <unk> N years old and former chairman of consolidated gold fields plc was named a nonexecutive director of this british industrial conglomerate
a form of asbestos once used to make kent cigarette filters has caused a high percentage of cancer deaths among a group of workers exposed to it more than N years ago researchers reported
the asbestos fiber <unk> is unusually <unk> once it enters the <unk> with even brief exposures to it causing symptoms that show up decades later researchers said
<unk> inc. the unit of new york-based <unk> corp. that makes kent cigarettes stopped using <unk> in its <unk> cigarette filters in N although preliminary findings were reported more than a year ago the latest results appear in today 's new england journal of medicine a forum likely to bring new attention to the problem

- The above is example training data from penn-treebank.
- A common benchmark problem is predicting the next character or word to appear in a sentence.
- Each sentence is of a varying length and in the case of penn-treebank there are 42068 sentences in the training data.

Character/Word Prediction

aer banknote berlitz calloway centrust cluett fromstein gitano guterman hydro-quebec ipo kia memotec mlx nahb punts rake regatta rubens sim
snack-food ssangyong swapo wachter
pierre <unk> N years old will join the board as a nonexecutive director nov. N
mr. <unk> is chairman of <unk> n.v. the dutch publishing group
rudolph <unk> N years old and former chairman of consolidated gold fields plc was named a nonexecutive director of this british industrial conglomerate
a form of asbestos once used to make kent cigarette filters has caused a high percentage of cancer deaths among a group of workers exposed to it more than N years ago researchers reported
the asbestos fiber <unk> is unusually <unk> once it enters the <unk> with even brief exposures to it causing symptoms that show up decades later researchers said
<unk> inc. the unit of new york-based <unk> corp. that makes kent cigarettes stopped using <unk> in its <unk> cigarette filters in N although preliminary findings were reported more than a year ago the latest results appear in today 's new england journal of medicine a forum likely to bring new attention to the problem

- Both involve one hot encoding:
 - character prediction becomes 26 inputs (assuming punctuation is ignored)
 - word prediction becomes over 40k inputs (one for each word!) - this is where PCA/whitening and dimensionality reduction can be very helpful.

Generative Models

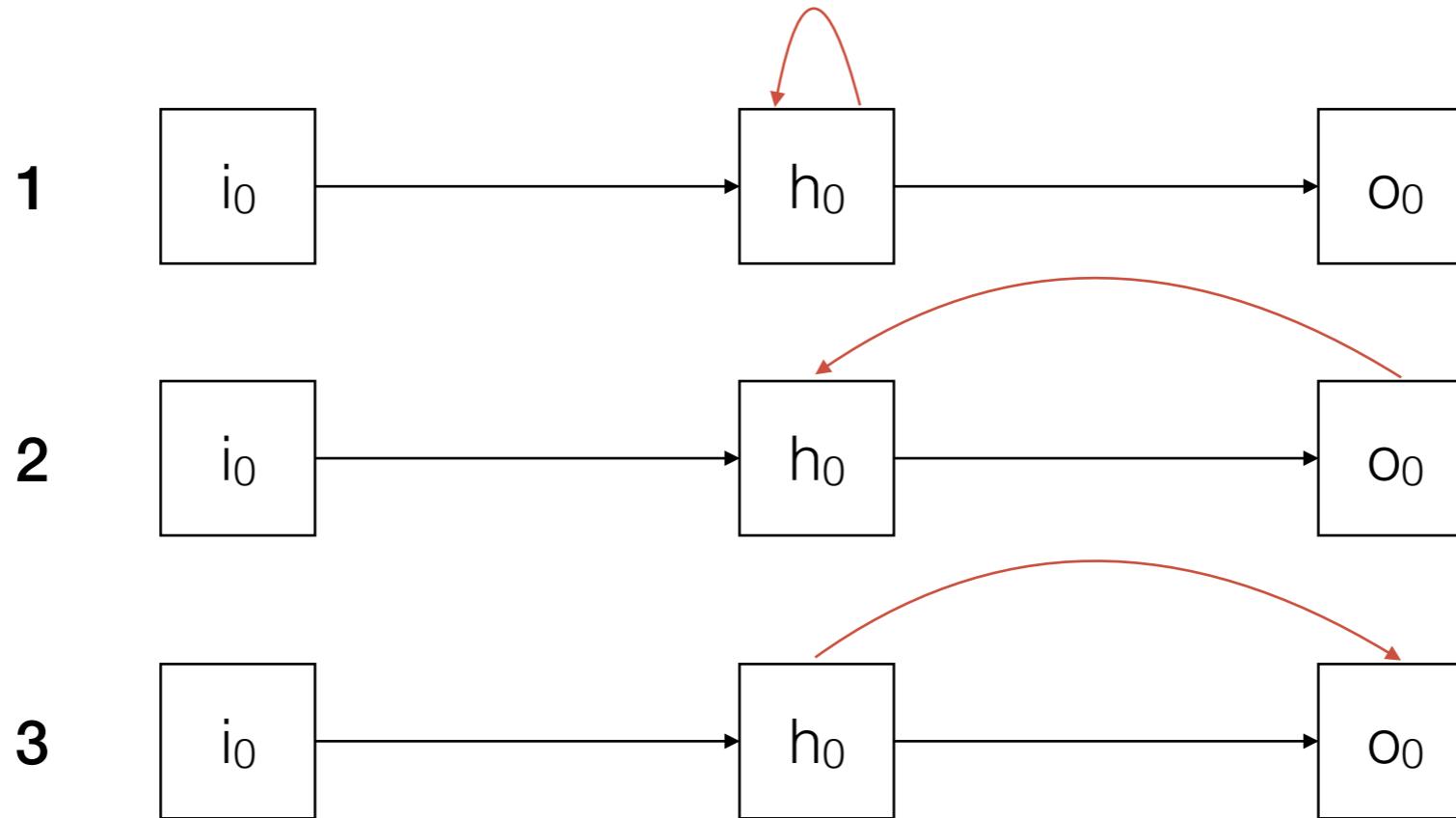
- One very cool thing that RNNs can do is that they can be used as *generative models*.
- If you design an RNN which predicts *all* of its inputs, then after being trained you can feed it's own output back into itself as it's own input.
- You can give it starting inputs and then it can generate its own data; creating new sequences.
- This can be done for time series or character/word prediction. It has even been used to generate music (audio is time series data).

Traditional RNN Architectures

Recurrent Connections

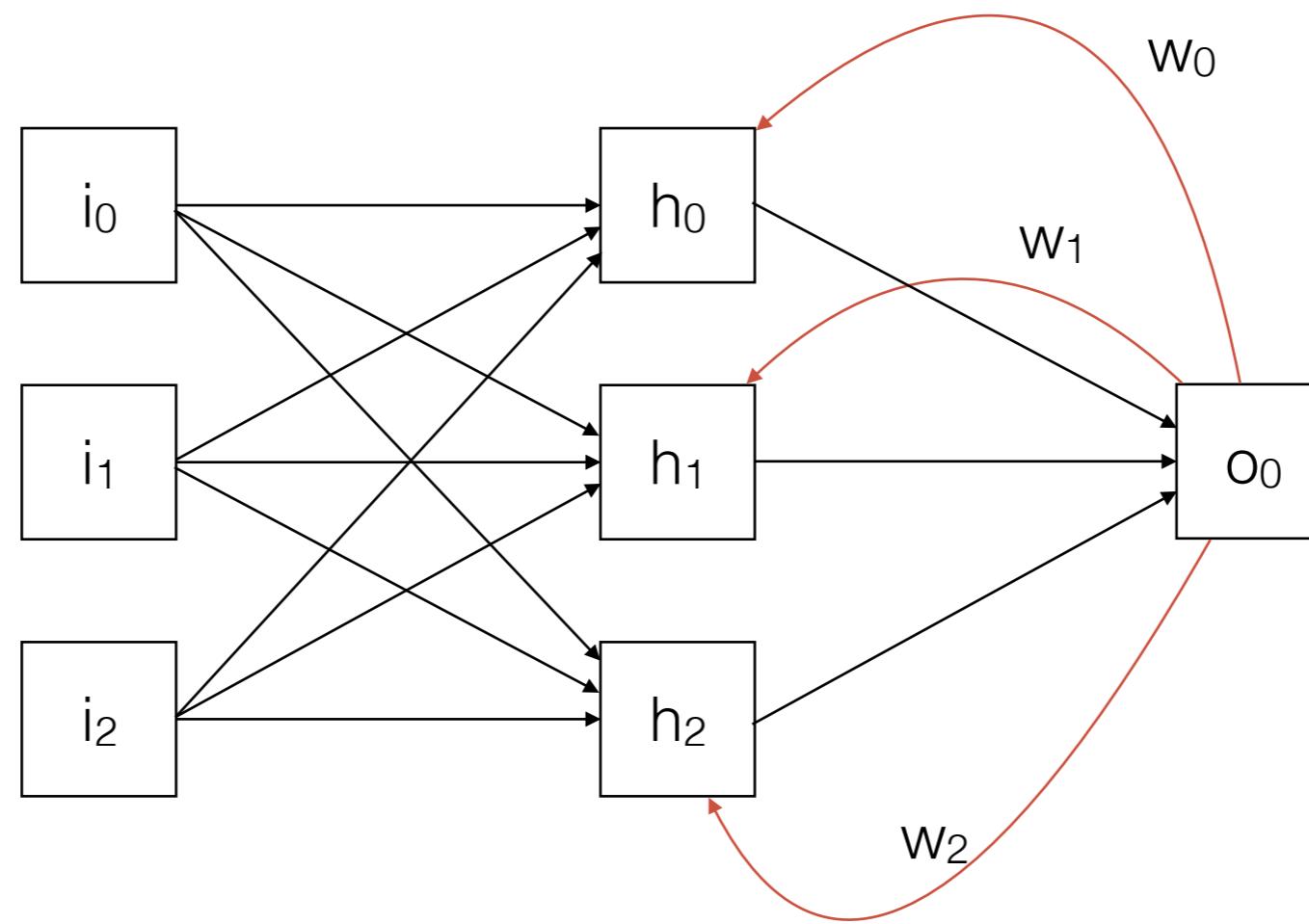
- Until now, we dealt with feedforward neural networks which were restricted to being directed, acyclic graphs (i.e., no edges could loop back and create cycles within the network graph).
- Recurrent neural networks remove this restriction by allowing recurrent connections which loop back into the network, using weighted connections to pass information from previous passes in the network.

Recurrent Connections



- Above shows three possibilities (in red) for recurrent connections:
 1. A node can be connected to itself.
 2. A node can be connected to a node in a prior layer in the network.
 3. We can also have forward recurrent connections (which will make more sense when we unroll our networks later).
- The first two are loopback connections which create a cycle in a graph. These cycles become undone when we unroll the network over time.

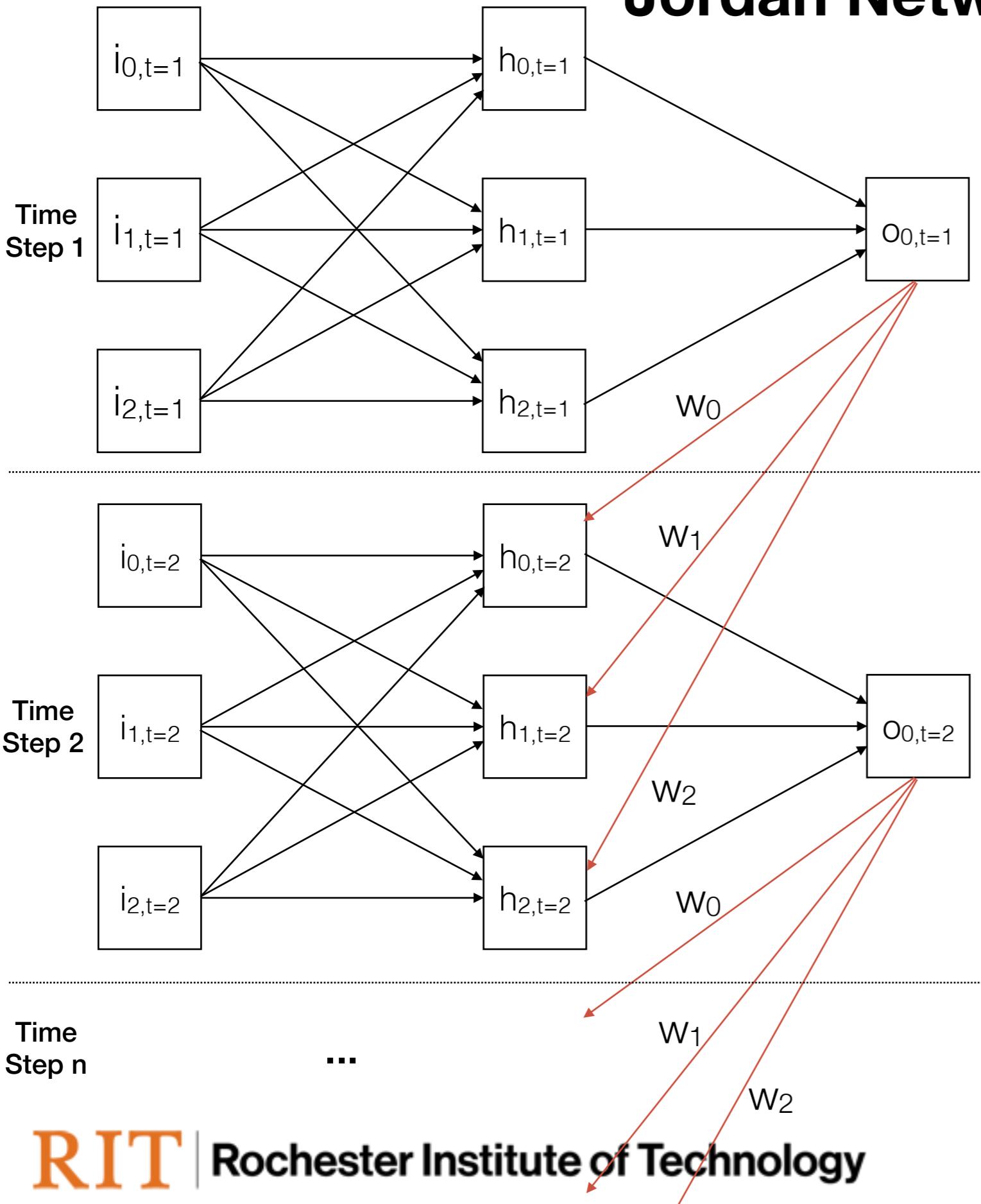
Jordan Networks



- Jordan RNNs were proposed by Michael I. Jordan in 1986 [1] and have been commonly used for time series data prediction.
- In a Jordan RNN the output from the previous pass through the RNN (for a given sequence) is fed into the hidden nodes after multiplying it by weights on those connections (in this case, w_0 , w_1 and w_2).
- Jordan networks typically have only one hidden layer, which can have any number of nodes (although an equal number to the number of input nodes is a common choice).

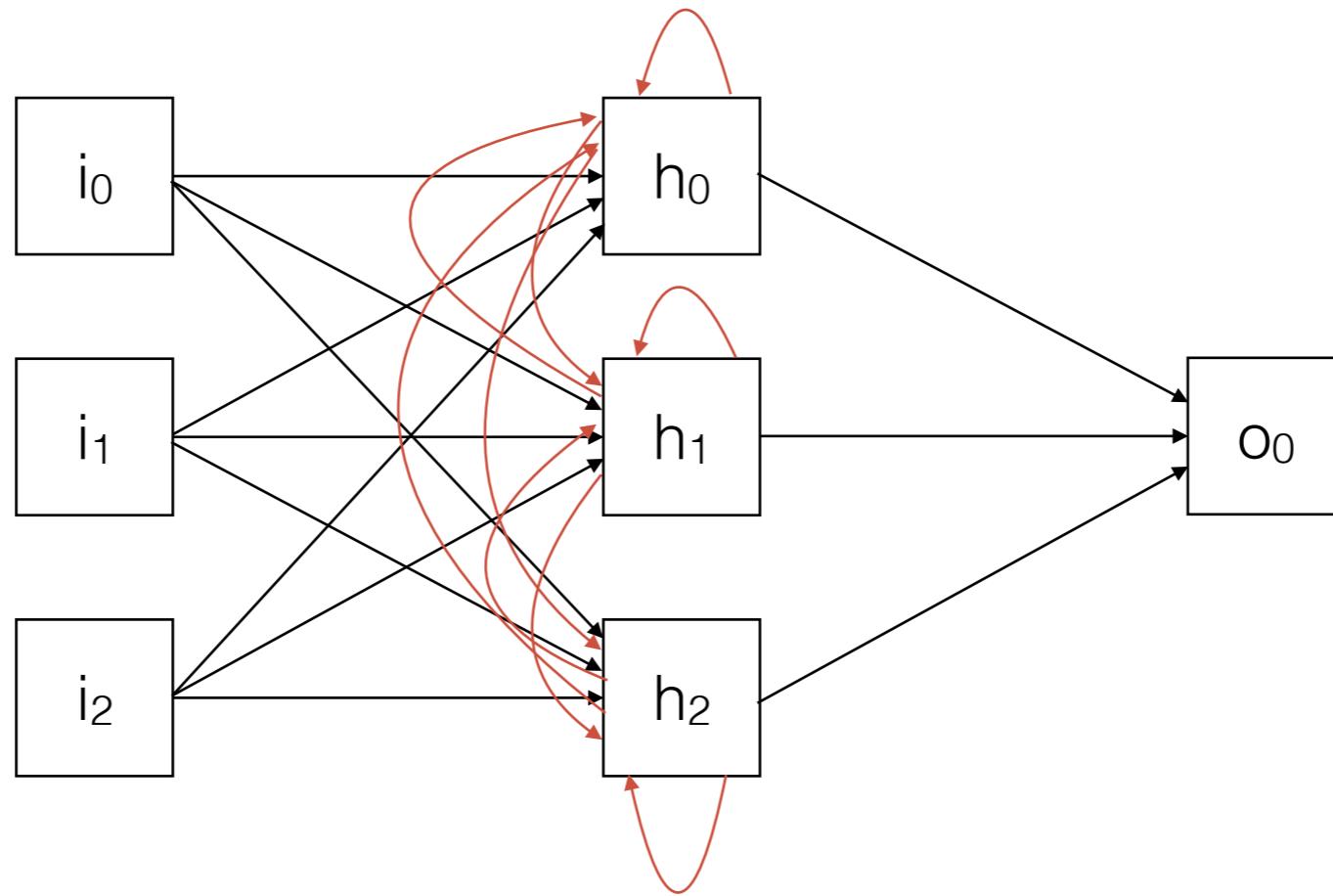
[1] Michael I. Jordan. **Serial order: A parallel distributed processing approach.** Technical Report 8604, Institute for Cognitive Science, University of California, San Diego, 1986.

Jordan Networks



- When recurrent connections are used, we can think about the network "*unrolling through time*".
- A copy of the network is made (with the same weights) for pass through the network for a given sequence.
- When we use backpropagation through time (more on that later) this needs to be done explicitly to calculate the gradients from the backward pass.
- Gradient free methods (like evolutionary algorithms) can avoid the unrolling process for some big computational and memory saving benefits -- one of the reasons they are popular for training RNNs.

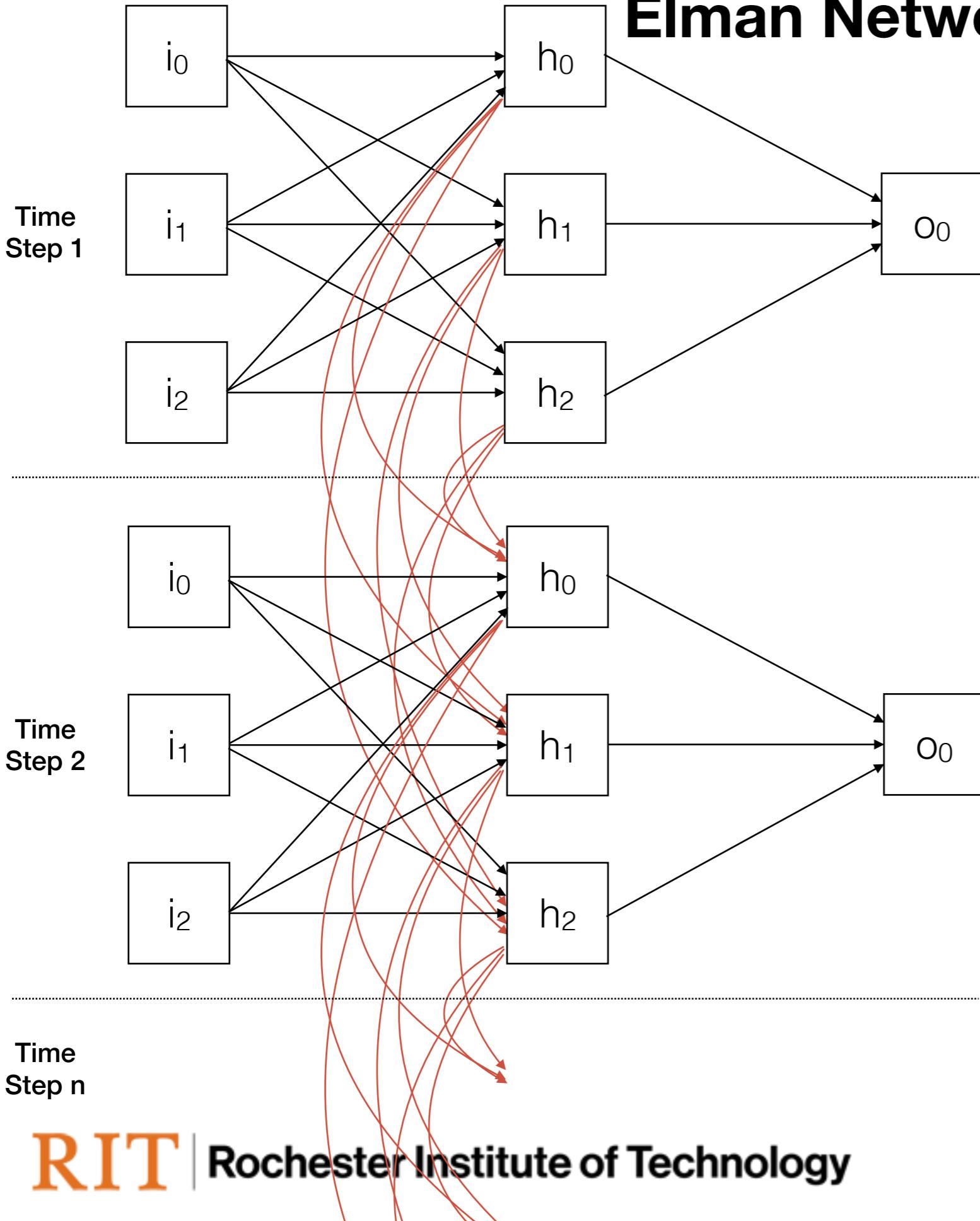
Elman Networks



- Elman networks were proposed in 1990 by Jeffrey L. Elman [2].
- Instead of passing the output from a previous pass through the network, they fully connect the hidden layer to itself with recurrent connections (each with their own weight).
- Whereas in the Jordan network (with a similar 3 inputs and 3 hidden nodes) only 3 new recurrent edges and weights were added, in the case of an Elman network, 9 new recurrent edges and weights are added.

[2] Jeffrey L. Elman. **Finding structure in time**. *Cognitive science*, 14(2):179–211, 1990.

Elman Networks

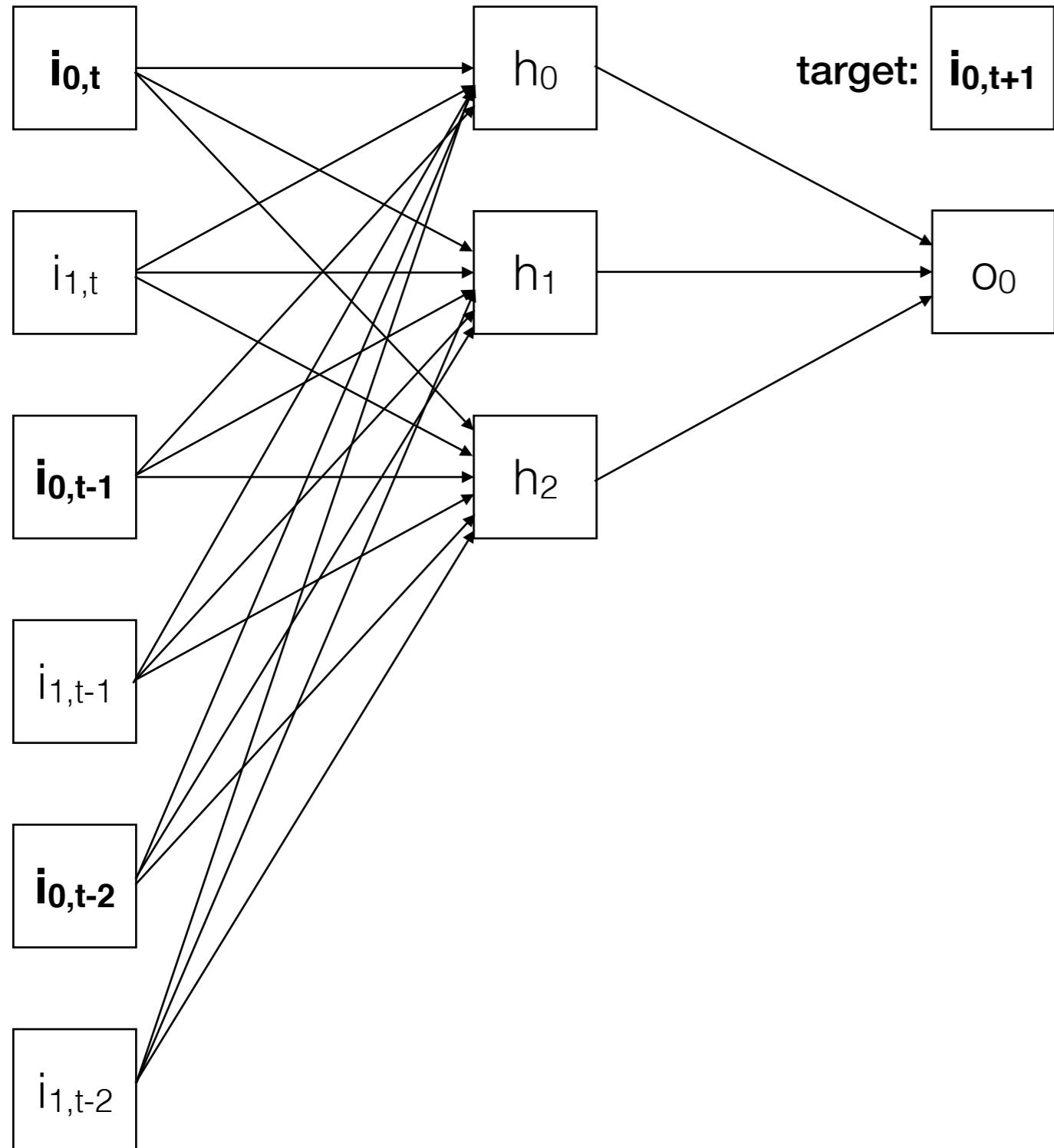


- We can unroll an Elman network in a similar fashion. Each hidden node has a weighted connection to each hidden node in the subsequent pass through the neural network for the sequence.

Nonlinear Models

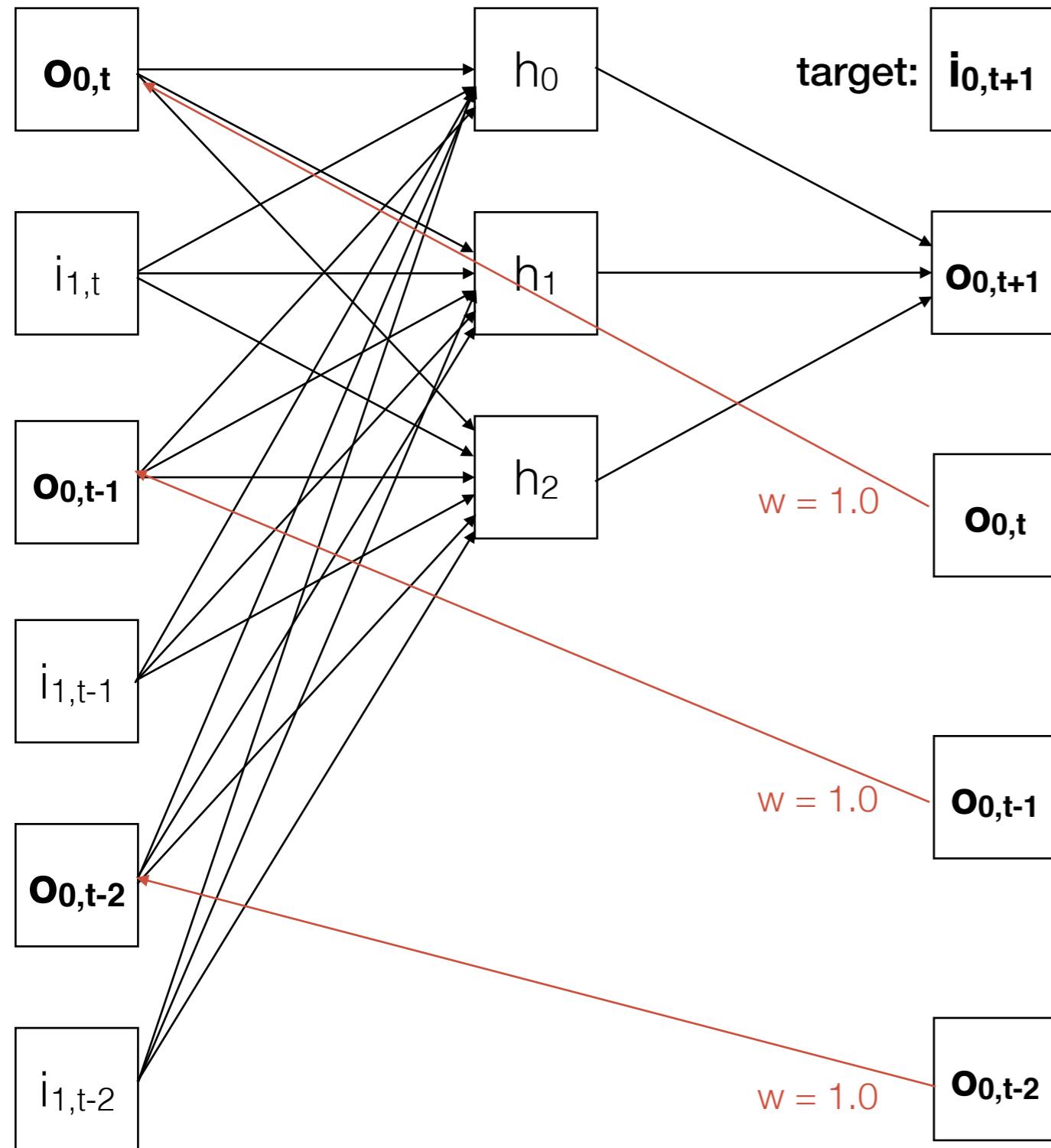
- Other recurrent neural network models have come out of statistical methods for time series data prediction (e.g., auto-regressive moving average (ARMA) models).
- These use the concept of *lagged* or *windowed* inputs, where not only the parameters for time t are passed into the network, but also parameters from time $t - 1, t - 2, \dots, t - n$ where n is the lag or window size.

Nonlinear Output Error (NOE) Network



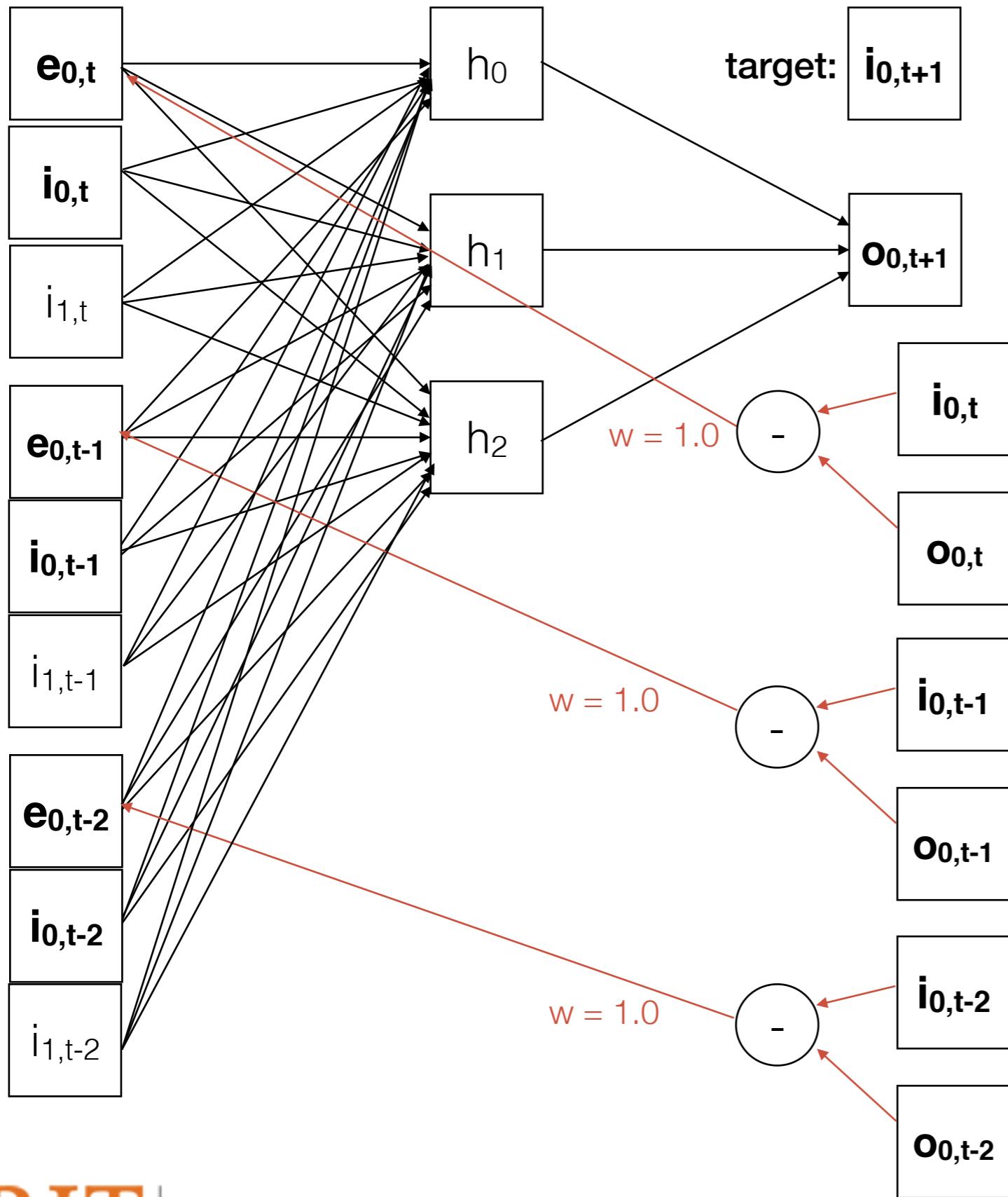
- An NOE network is actually not really a recurrent neural network (as it does not have recurrent connections) but is typically used in a similar manner.
- This example is for two time lags/windows ($n = 2$)
- In an NOE network, previous version of the prediction target (e.g., input 0 at time $t+1$) are fed into the network along with the previous windows.
- NOEs can train as easily as FFNNs because they do not require unrolling.

Nonlinear AutoRegressive with eXogenous inputs (NARX) Network



- In a NARX network, instead of passing in the prediction parameter (i_0) the previous predicted values are passed from the previous passes through the neural network.
- These are over a recurrent connection but it's weight is fixed at 1 (it is not trainable).

Nonlinear Box Jenkins (NBJ) Network

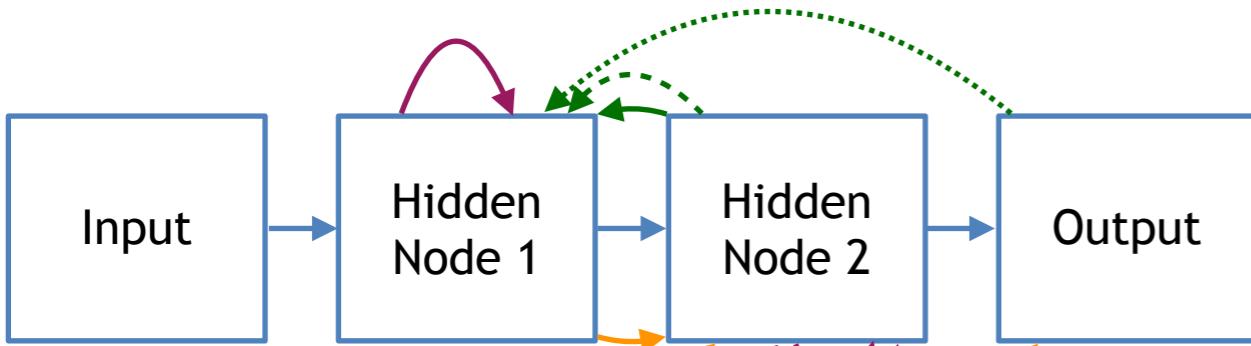


- In an NBJ model, similar to an NOE model the actual prediction parameter (the i_0 node, assuming $i_{0,t+1}$ is what we're predicting) at time t is passed into the network.
- In addition, the error (the e nodes) between the prediction and the actual value are also passed in as an input parameter via a recurrent connection (with an untrainable weight = 1).

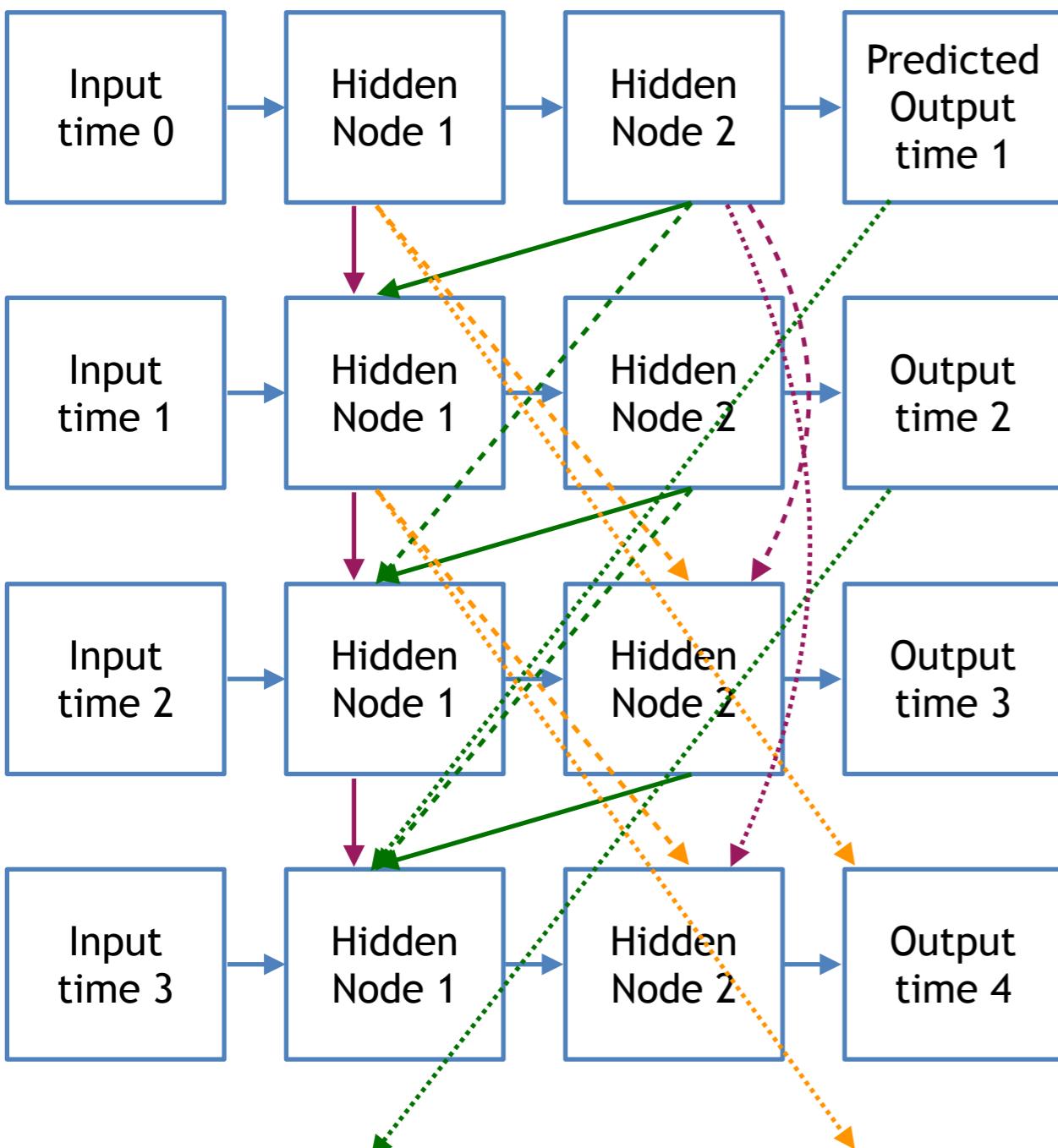
A Generic Recurrent Connection Model

Generic Recurrent Connections

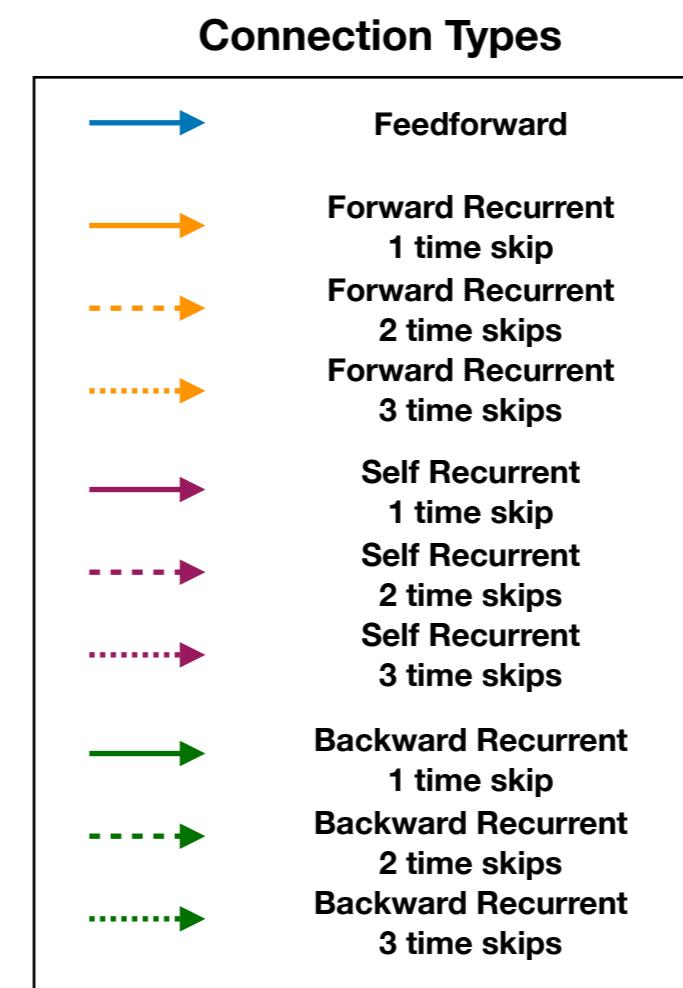
- The basic idea behind RNNs is passing information from previous passes through the neural network to future passes via weighted connections.
- When we unroll a network we can see we have a number of different options.



Recurrent Network



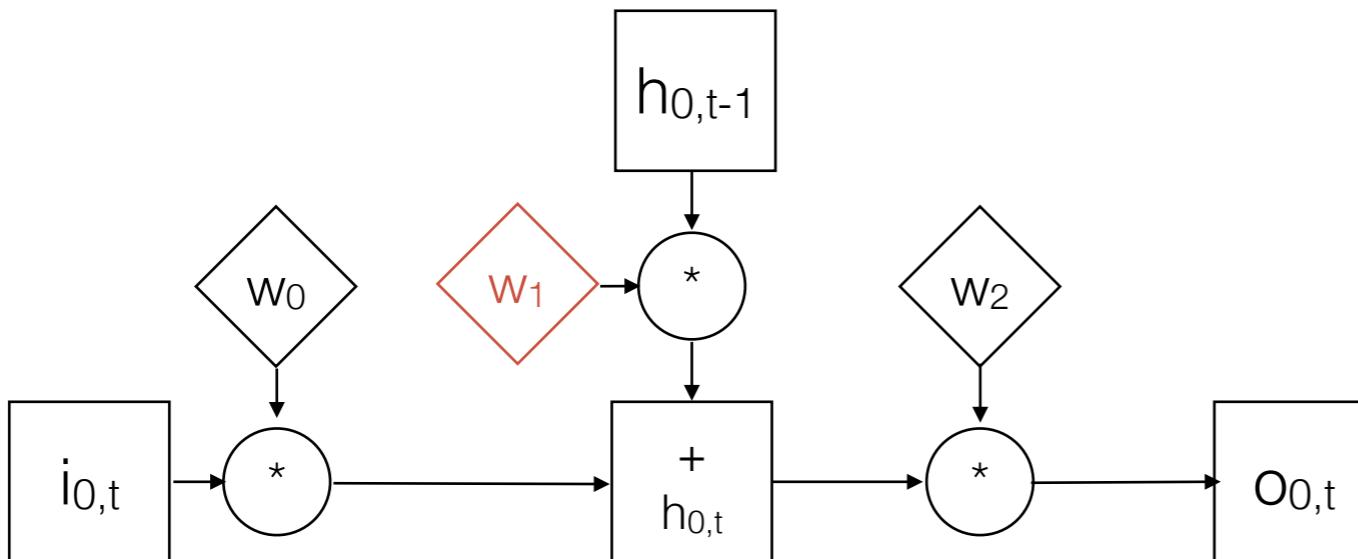
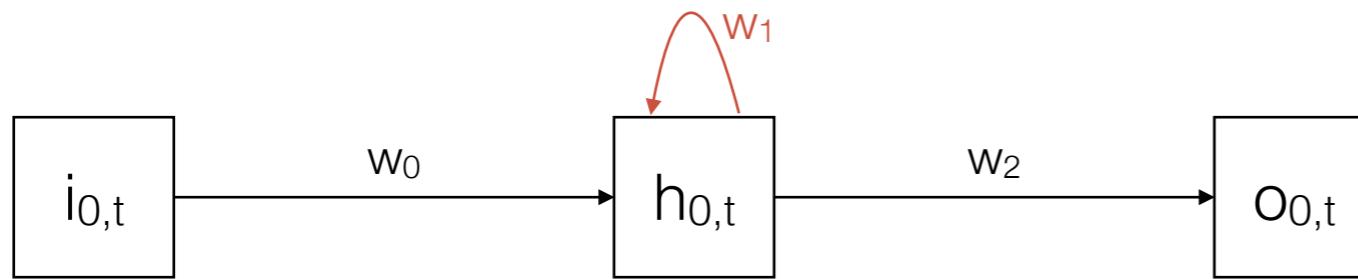
Unrolled Recurrent Network



- It is possible to have a wide variety of potential recurrent connections. The networks we've seen until now focus on either self recurrent or backward recurrent connections with 1 time skip.
- So we can have three different types of recurrent connections - backward, self, and forward. And they can actually span multiple time steps.
- This makes the possible search space of RNNs much larger than that of FFNNs.

RNN Forward Pass and Loss Functions

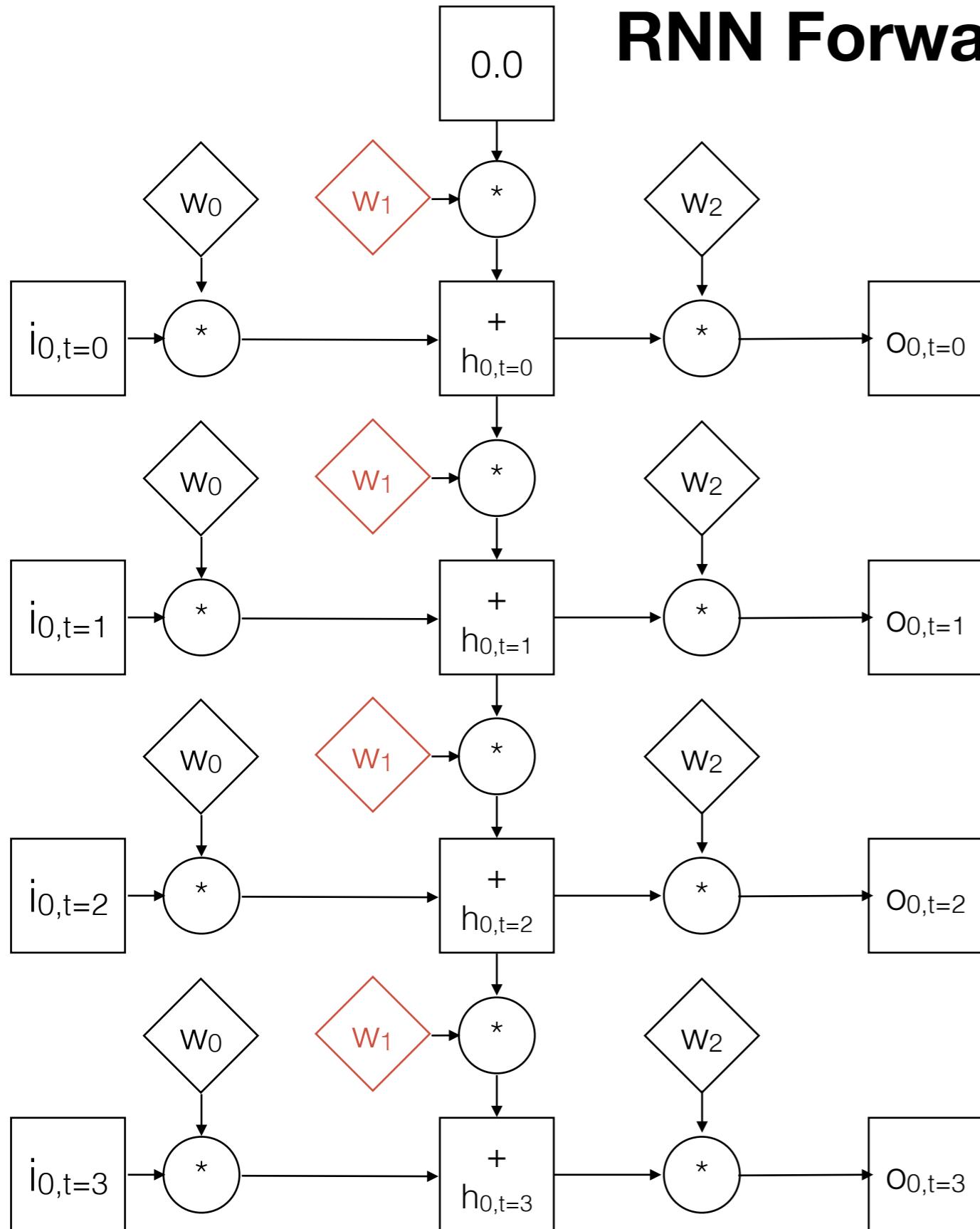
RNN Forward Pass



time (t)	Input	Output
0	1	2
1	2	4
2	3	8
3	4	16

- We'll work with the simplest Elman network as an example of how to do a forward pass, and how to calculate the loss functions.
- Lets convert a single time step into a circuit diagram.
- We'll use the $f(x) = x^2$ function as our sequential input data.
- For simplicity we'll use a linear activation function $f(x) = x$

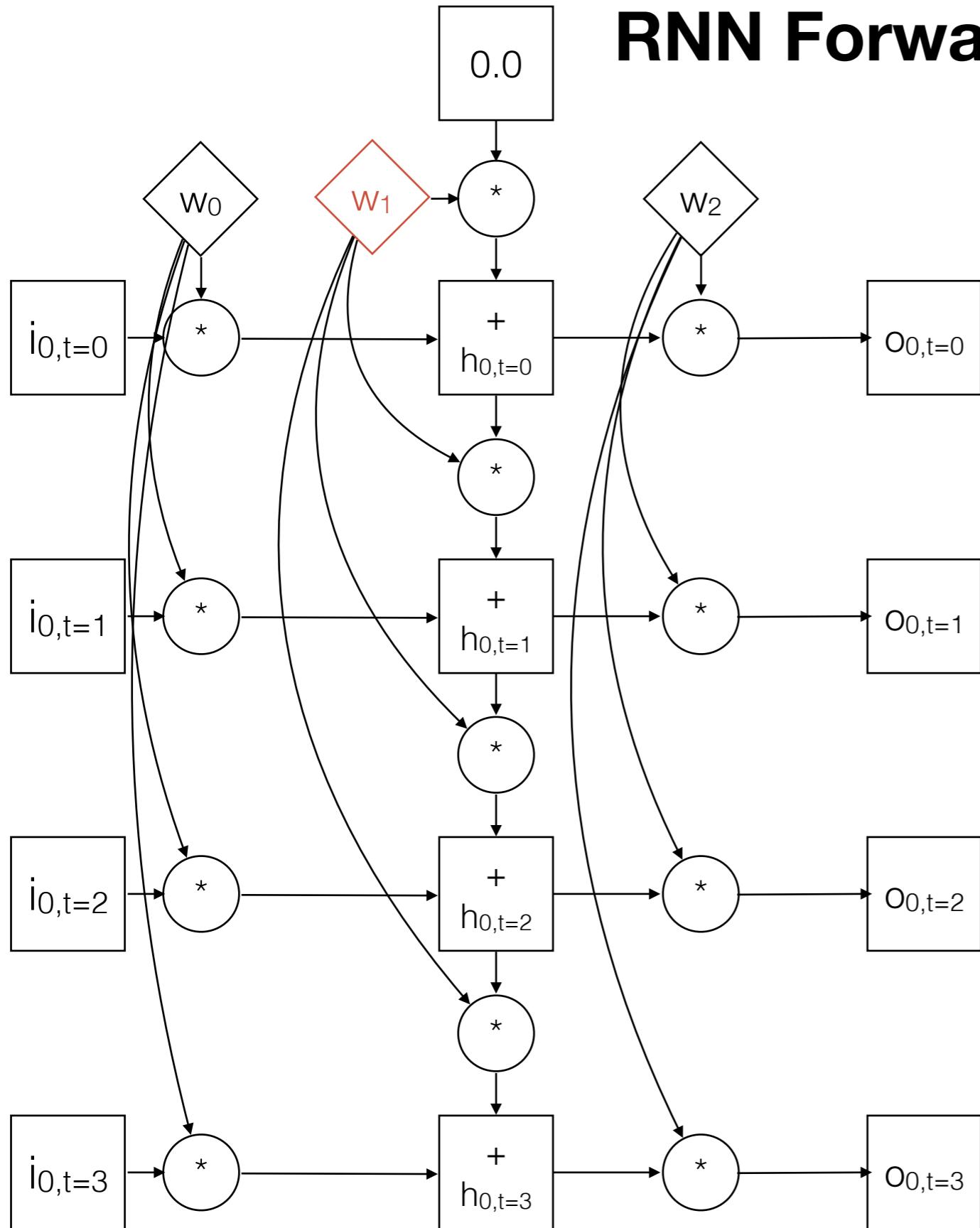
RNN Forward Pass



time (t)	Input	Output
0	1	2
1	2	4
2	3	8
3	4	16

- Because our sequence has 4 time steps we can unroll the network four times.
- Since there is no previous hidden node value for $t=0$, we can set it to 0.0

RNN Forward Pass

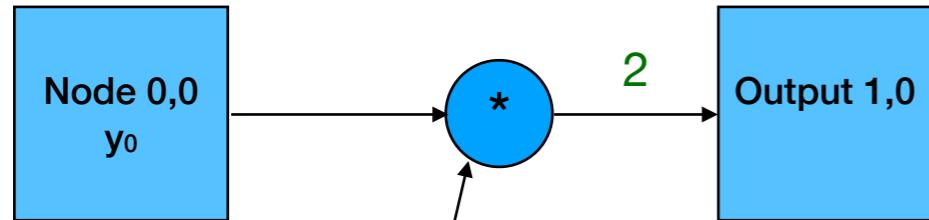


time (t)	Input	Output
0	1	2
1	2	4
2	3	8
3	4	16

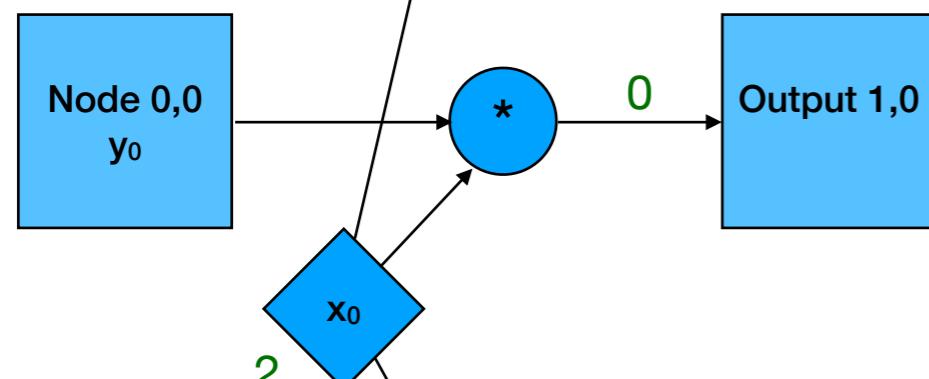
- Note that we re-use the same weights for each pass through the neural network.
- We can update our circuit accordingly.
- You may note this looks very similar to how we calculated a gradient for multiple instances at the same time.

Multiple Instances/Samples (Flashback!)

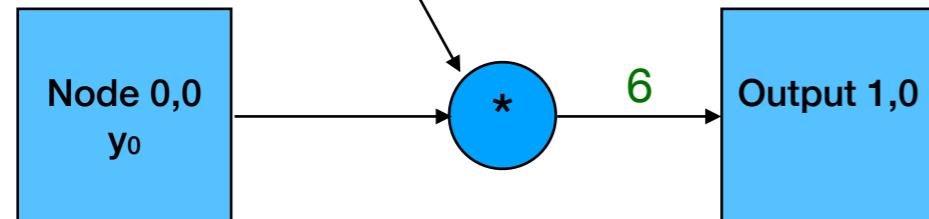
1



0

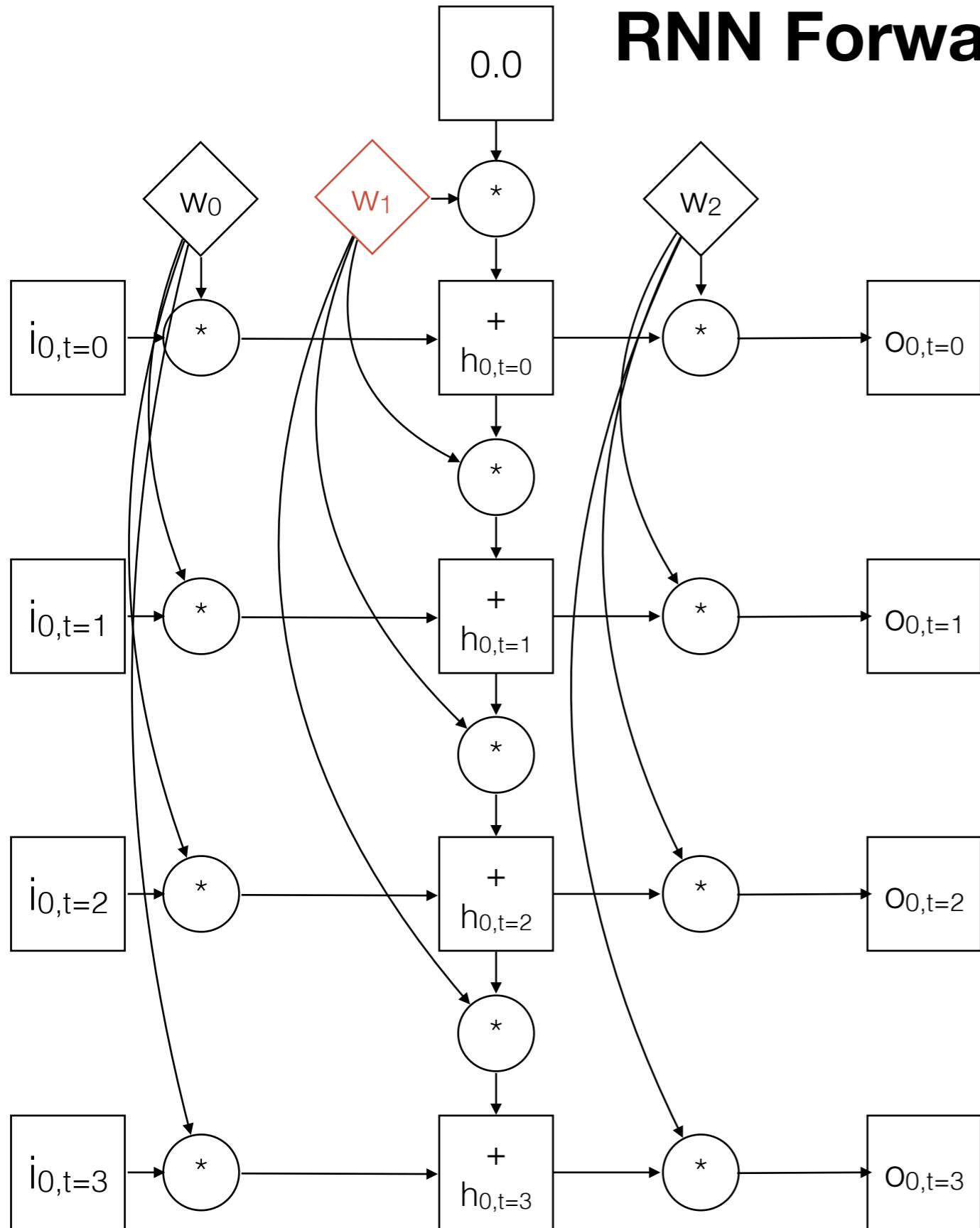


3



- Note that the x_0 value is the same for all of them. We can actually redraw this as seen on the left.
- As described previously, if we have multiple deltas going into the same node, we simply sum them.
- So in this case, to calculate the gradient across multiple samples with the same weights, we can do backpropagation on each of them individually and then simply sum up the gradients for each weight/bias.

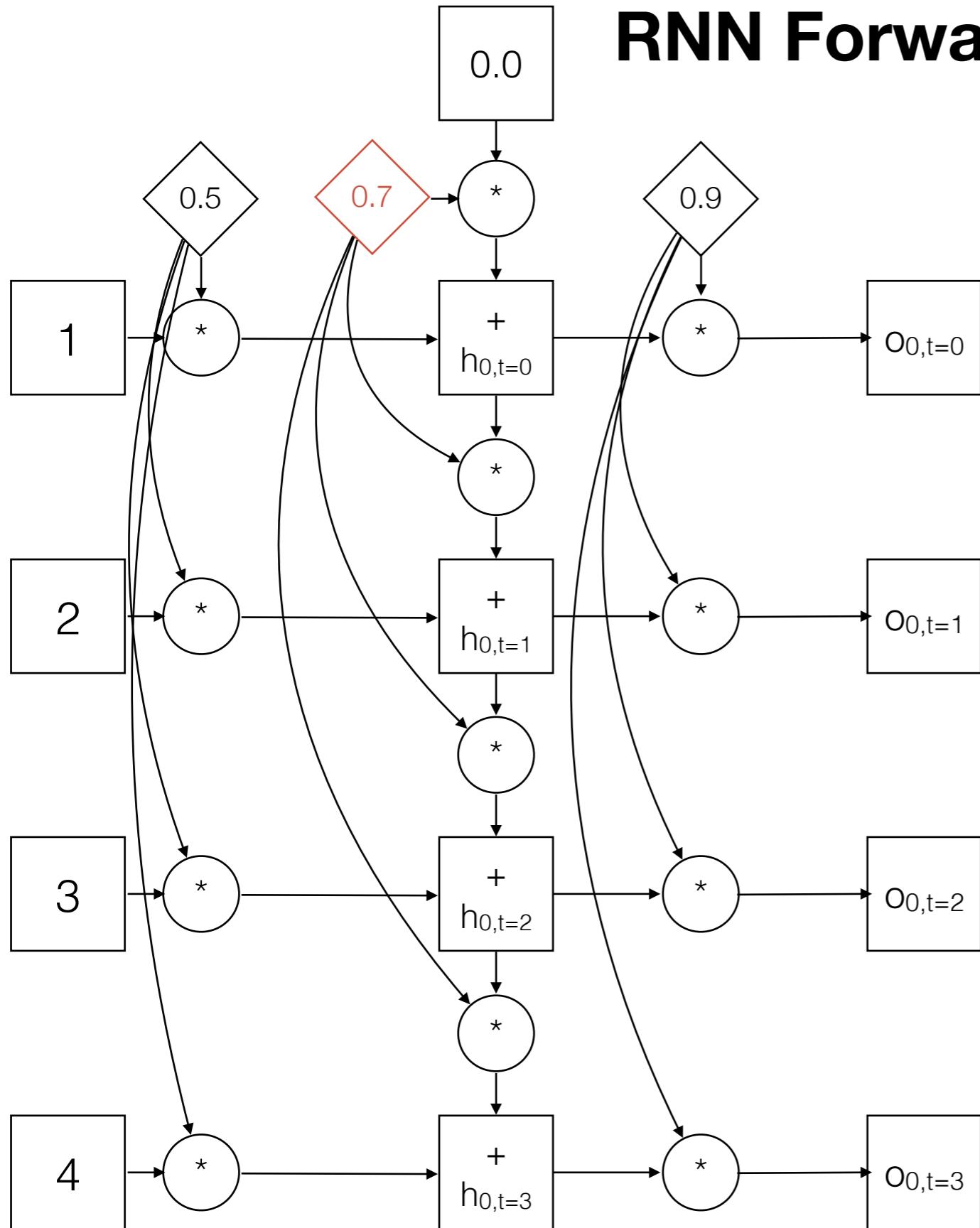
RNN Forward Pass



time (t)	Input	Output
0	1	2
1	2	4
2	3	8
3	4	16

- Unrolling the network over time took our directed cyclic graph and converted it to a directed acyclic graph.
- We can now train our network the same we train a FFNN. It will just be much deeper.
- We also need to unroll more or less depending on how many time steps in the sequences we are training on.

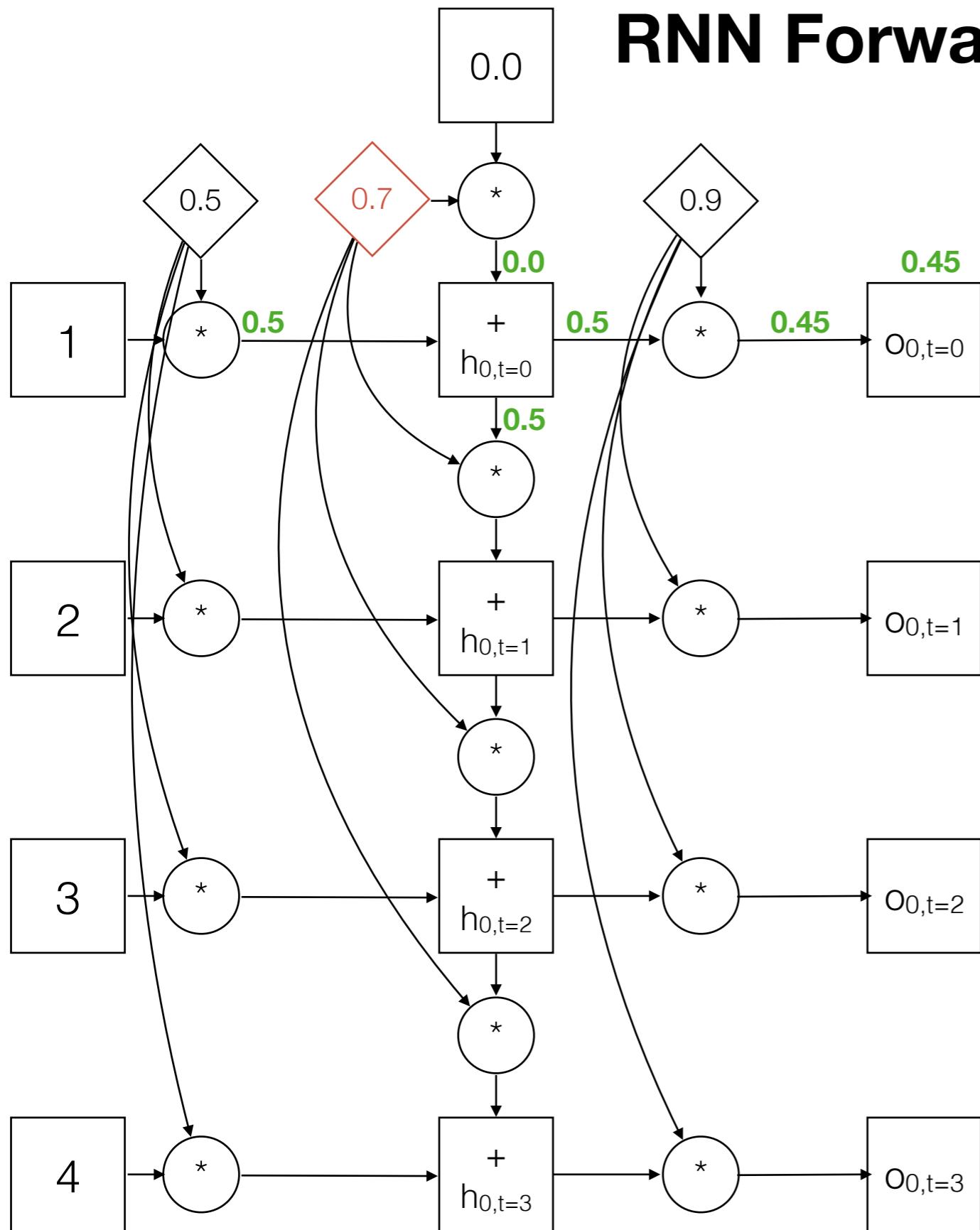
RNN Forward Pass



time (t)	Input	Output
0	1	2
1	2	4
2	3	8
3	4	16

- Now lets do a forward pass with weights $w_0 = 0.5$, $w_1 = 0.7$ and $w_2 = 0.9$.

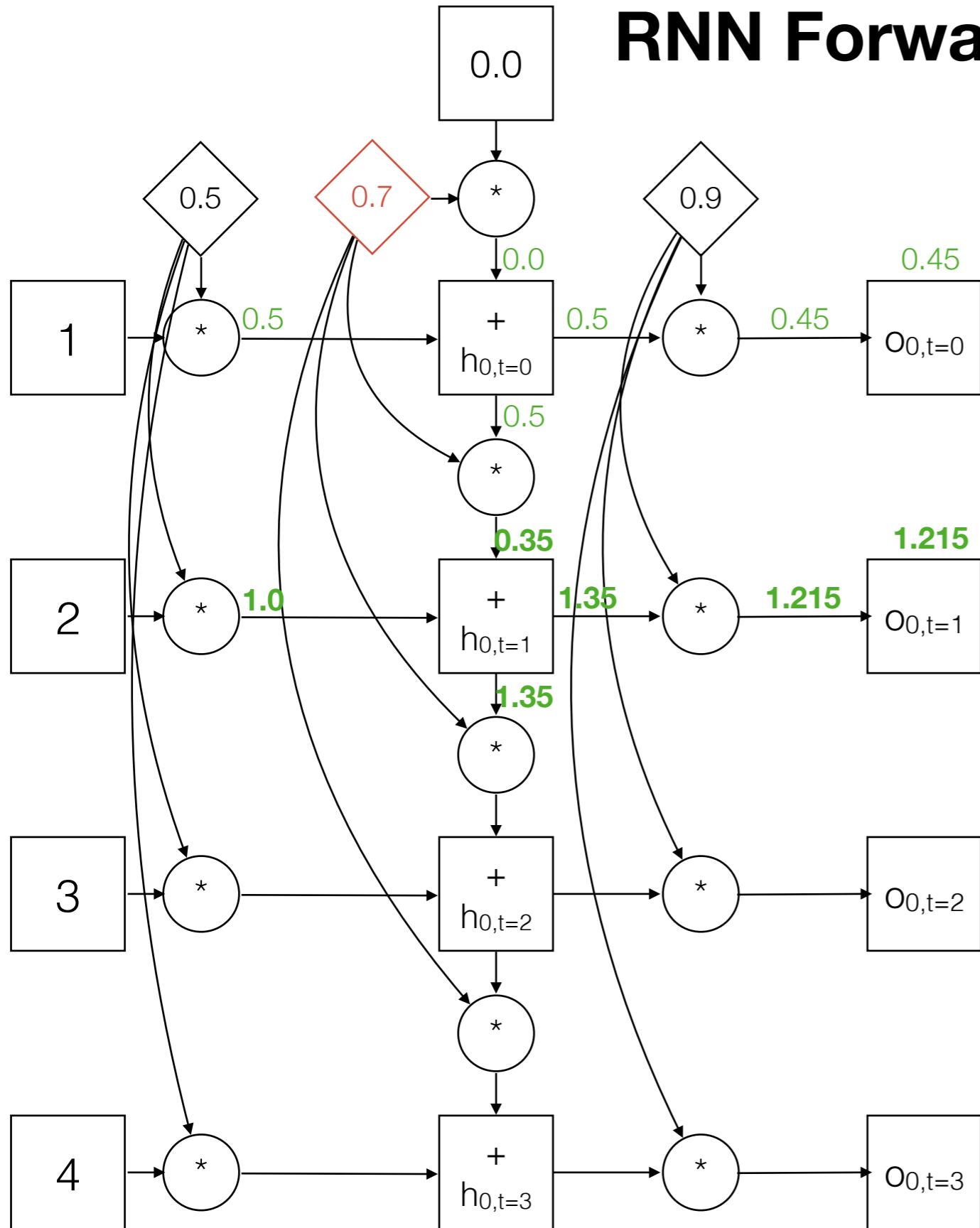
RNN Forward Pass



time (t)	Input	Output
0	1	2
1	2	4
2	3	8
3	4	16

- We need to do each time step sequentially as values from the previous time step are required for the subsequent time step.
- Here's t=1

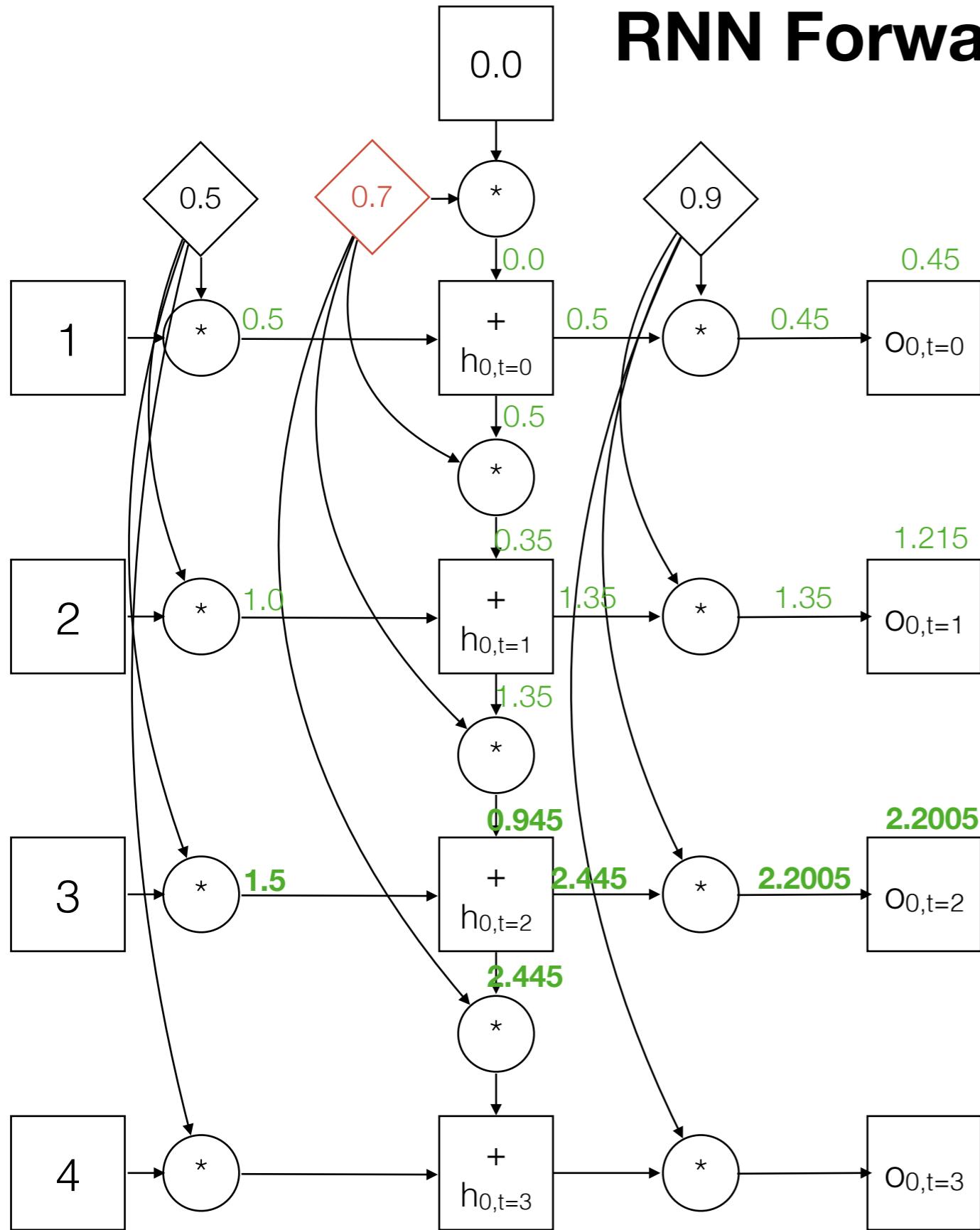
RNN Forward Pass



time (t)	Input	Output
0	1	2
1	2	4
2	3	8
3	4	16

- For time step $t = 2$, we need to also pass in the value from the hidden node multiplied by the recurrent weight

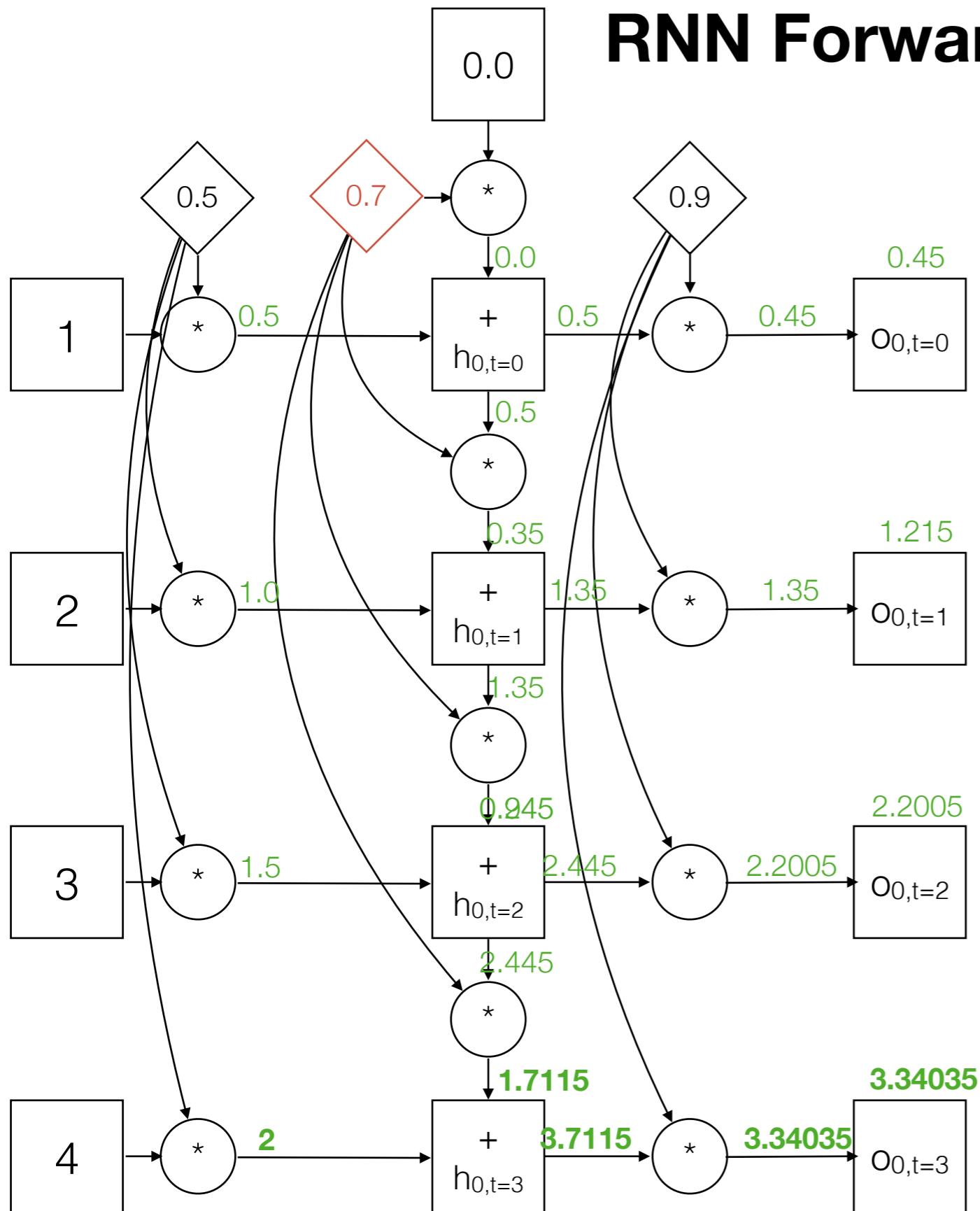
RNN Forward Pass



- Things progress similarly for $t=3$

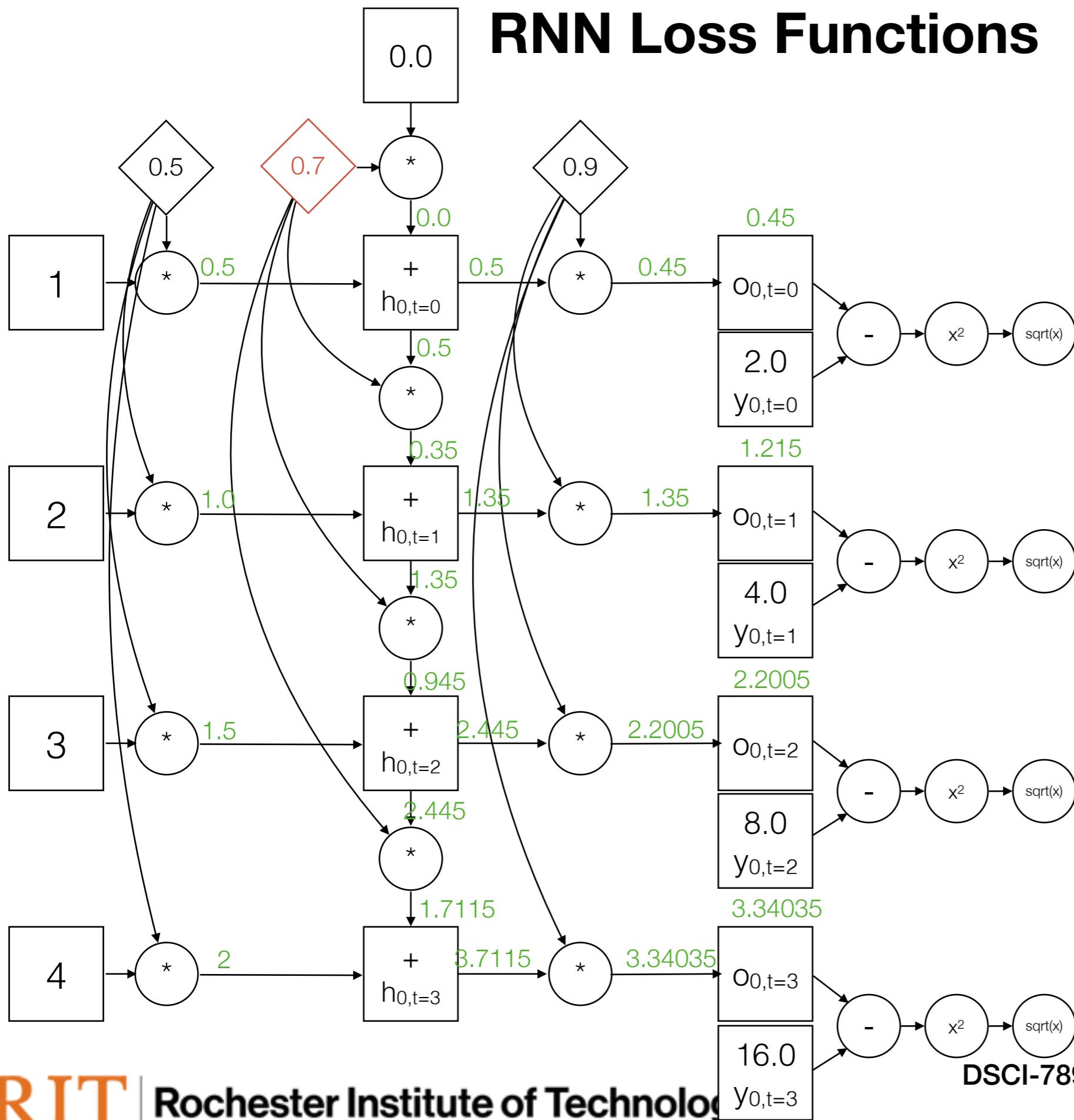
time (t)	Input	Output
0	1	2
1	2	4
2	3	8
3	4	16

RNN Forward Pass



- And for $t=4$.
- Now that we've done a pass through the network we also need to consider the loss function.
- As this sample problem is a regression problem, we can use an L2 norm loss function.

RNN Loss Functions

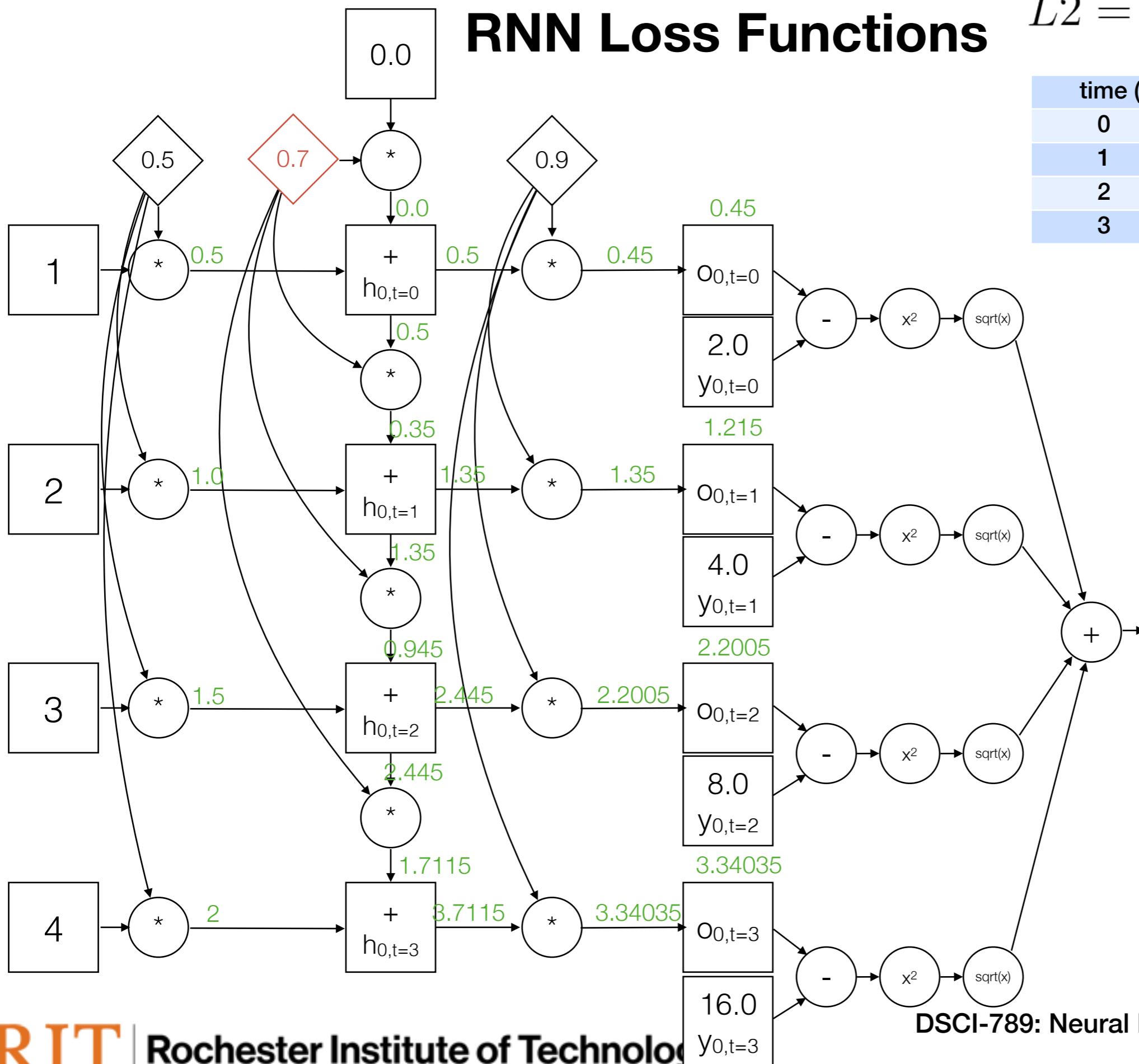


$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$

time (t)	Input	Output
0	1	2
1	2	4
2	3	8
3	4	16

- Since the L2 norm is only operating on one variable it is pretty straightforward:
 $L2(x) = \sqrt{x^2}$
- We can add this to each output, however we now have an error value for each output.

RNN Loss Functions

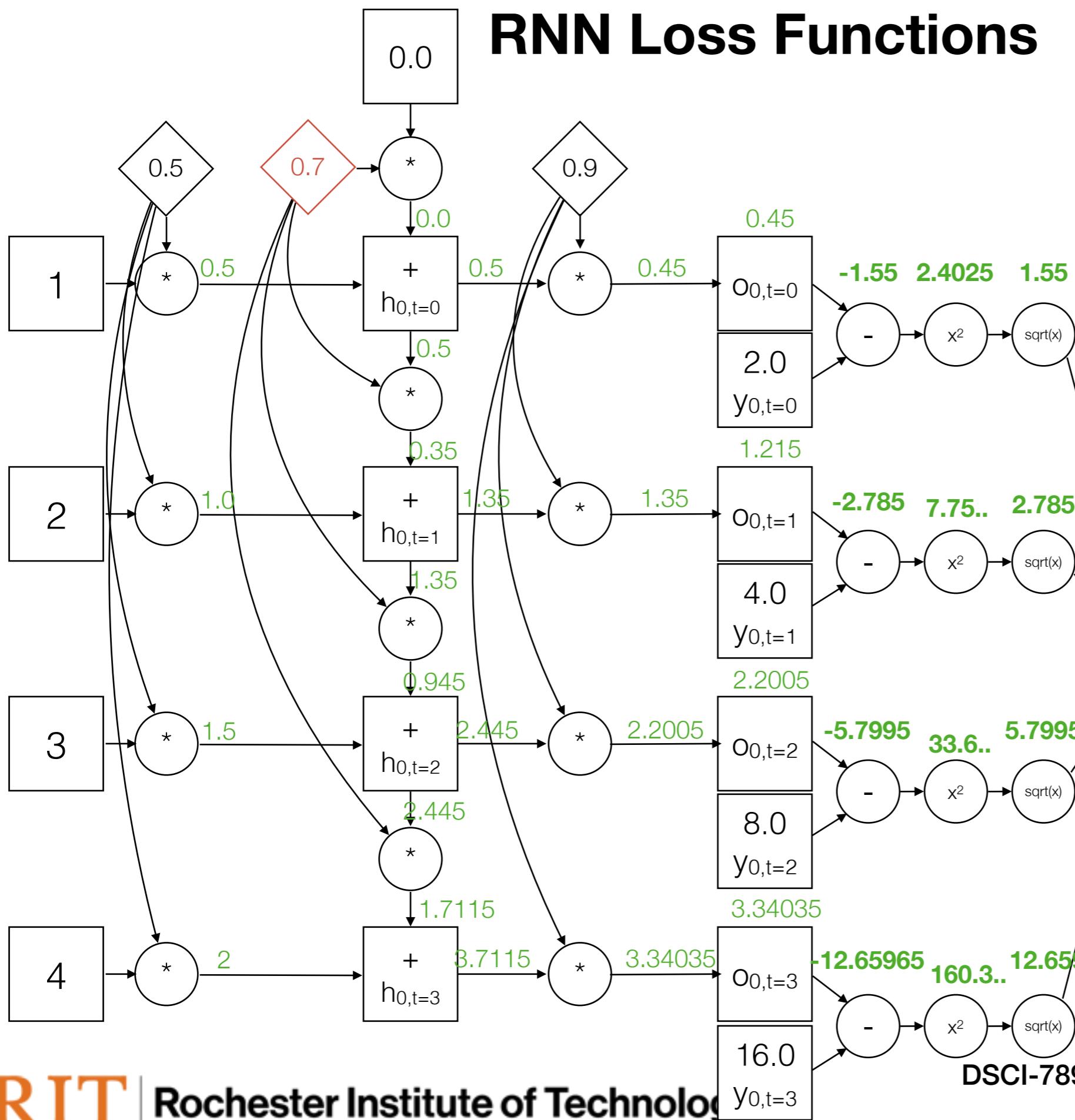


$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$

time (t)	Input	Output
0	1	2
1	2	4
2	3	8
3	4	16

- To calculate the final loss simply sum the loss over each output.

RNN Loss Functions



$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$

time (t)	Input	Output
0	1	2
1	2	4
2	3	8
3	4	16

- Now we can do the calculations for the final loss.
- This sets us up to do the backward pass just as before!

RNN Forward Pass and Loss Functions

- So to summarize, the process for the forward pass:
 1. Determine the length of the training sequence.
 2. Unroll the neural network, creating a copy for each time step in the sequence.
 3. Do a forward pass iteratively for each time step.
 4. Apply the loss function on each output, and sum the result of the loss function for each output.

Lecture 6

RNN Backward Pass, Weight Initialization and Dealing with Numerical Issues

DSCI-789: Neural Networks for Data Science

Travis Desell (tjdvse@rit.edu)
Associate Professor
Department of Software Engineering



ROCHESTER INSTITUTE OF TECHNOLOGY

Overview

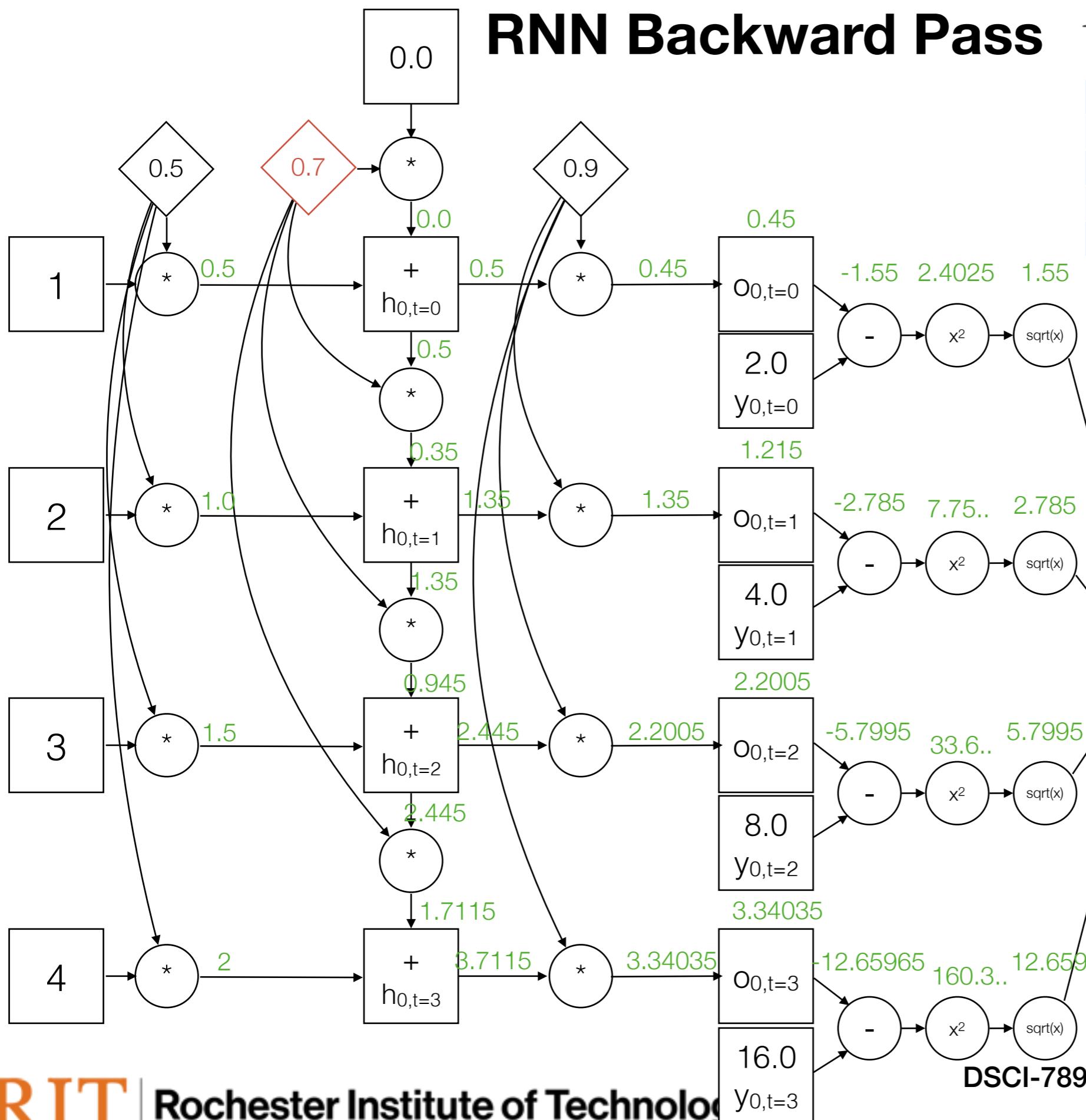
- RNN Backward Pass
- RNN Weight Initialization
- Gradient Clipping, Scaling and Boosting

RNN Backward Pass

RNN Backward Pass

- Let's start where we left off at the end of the last lecture with a forward pass done through an RNN.

RNN Backward Pass

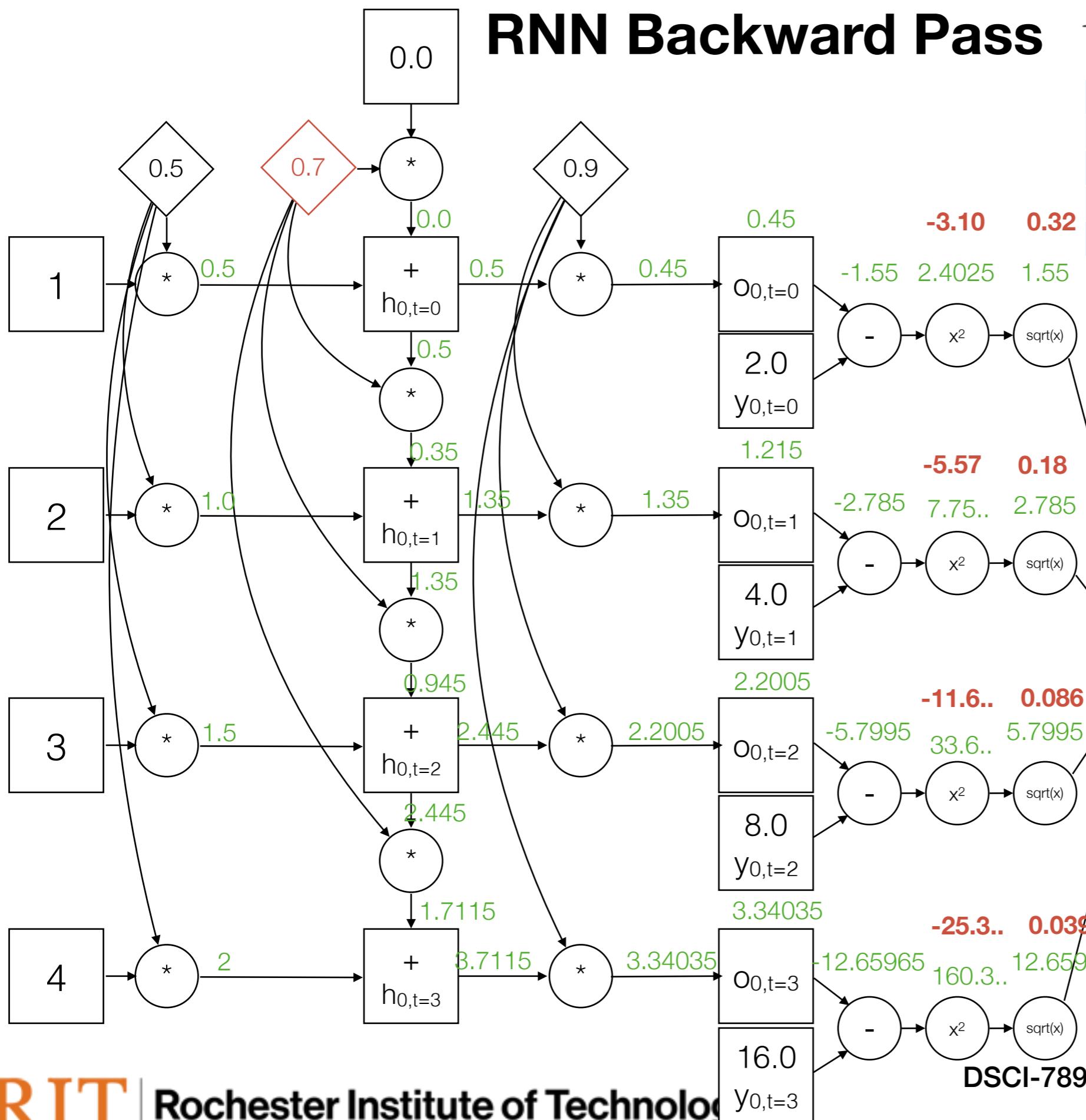


$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$

time (t)	Input	Output
0	1	2
1	2	4
2	3	8
3	4	16

- We've done a forward pass and calculated all the forward values.

RNN Backward Pass



$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$

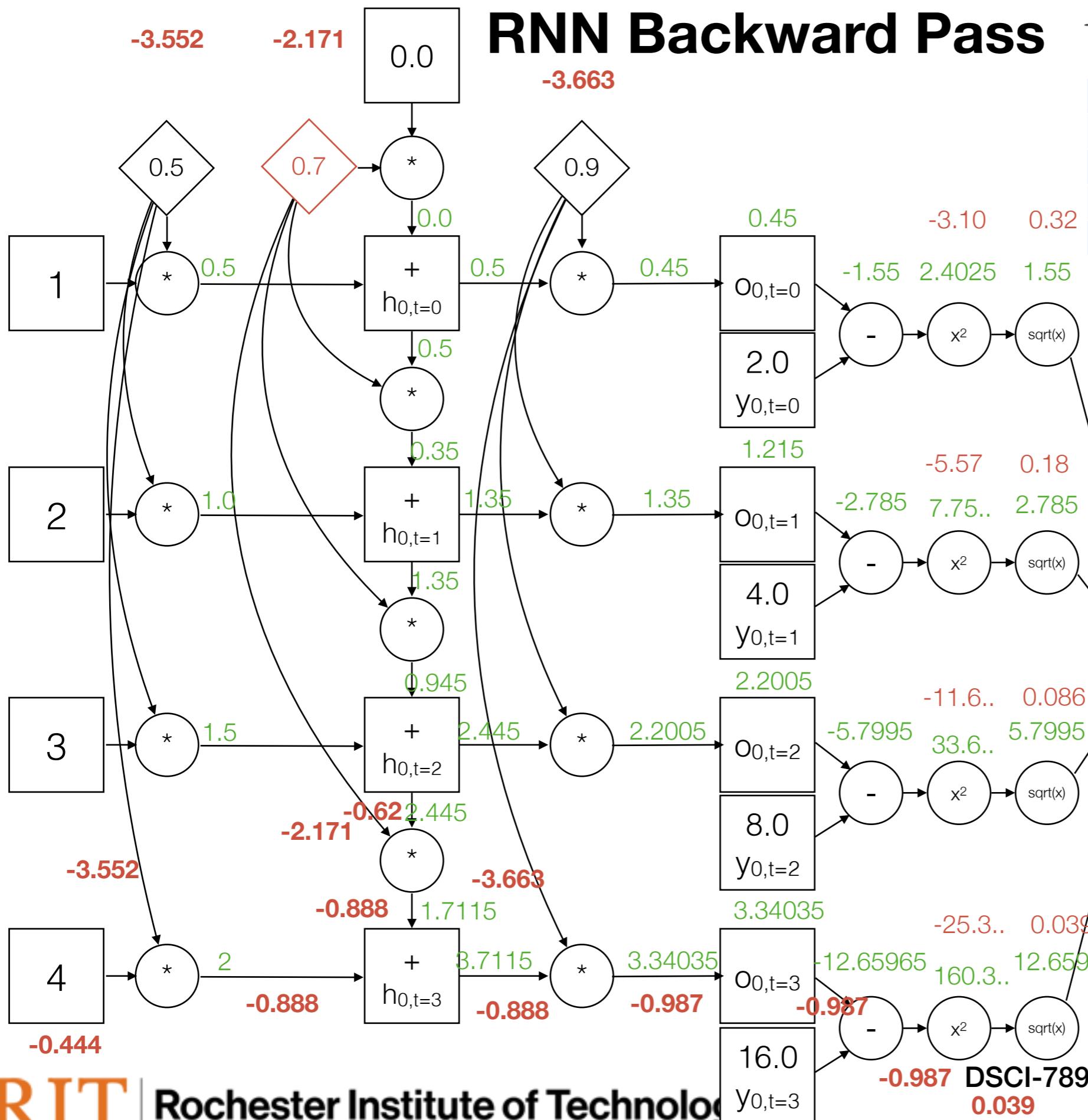
time (t)	Input	Output
0	1	2
1	2	4
2	3	8
3	4	16

- We can now calculate the derivatives we need to do the backward pass (in red).

$$\sqrt{x}' = \frac{1}{2\sqrt{x}}$$

$$x^{2'} = 2x$$

RNN Backward Pass



- Now we can start to back propagate the error.
- We need to start at the last time step through the RNN.

$$\sqrt{x'} = \frac{1}{2\sqrt{x}}$$

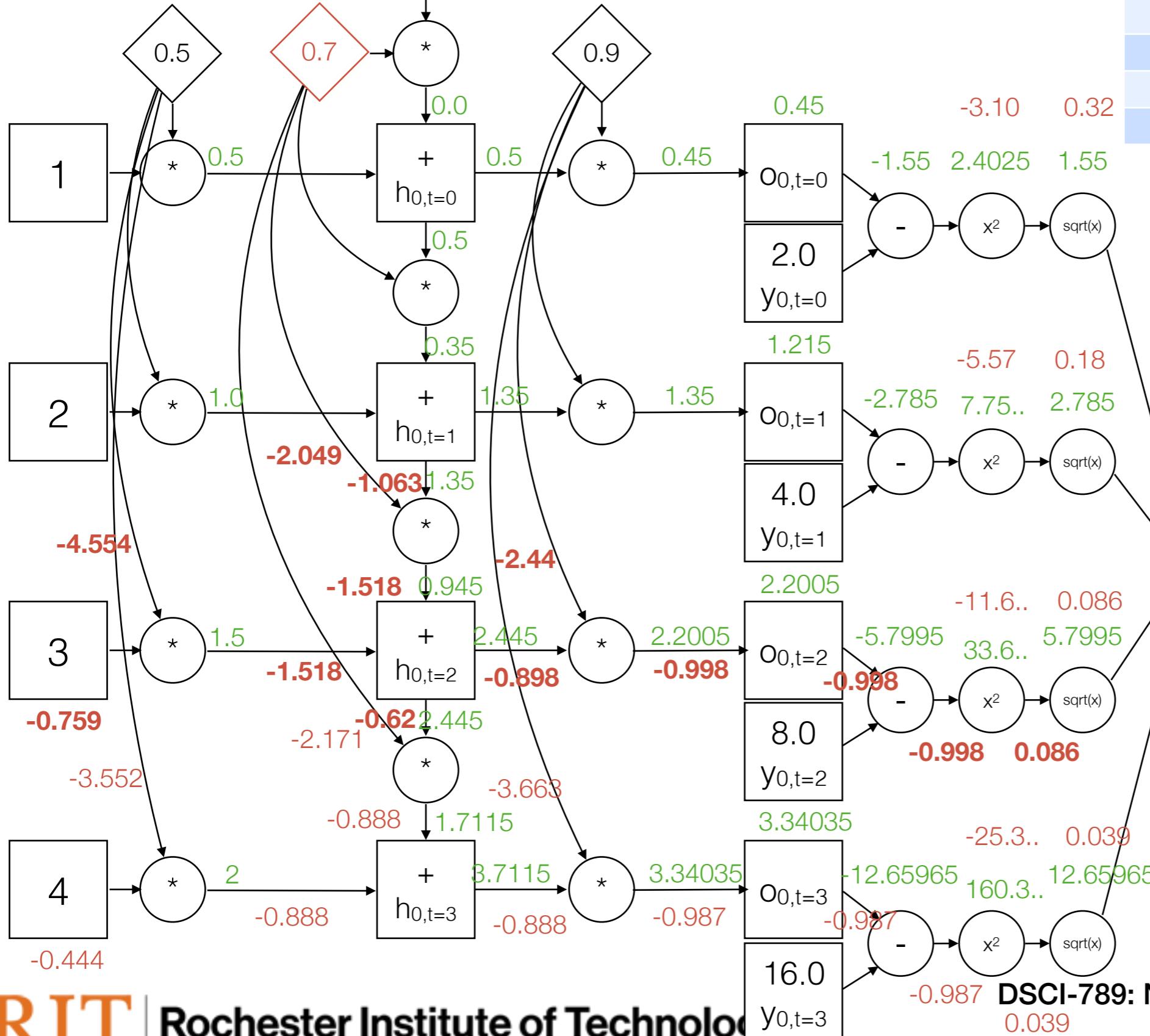
$$x^{2'} = 2x$$

$$-3.663 + -4.554$$

$$-2.171 + -2.049$$

$$0.0$$

$$-3.663 + -2.44$$

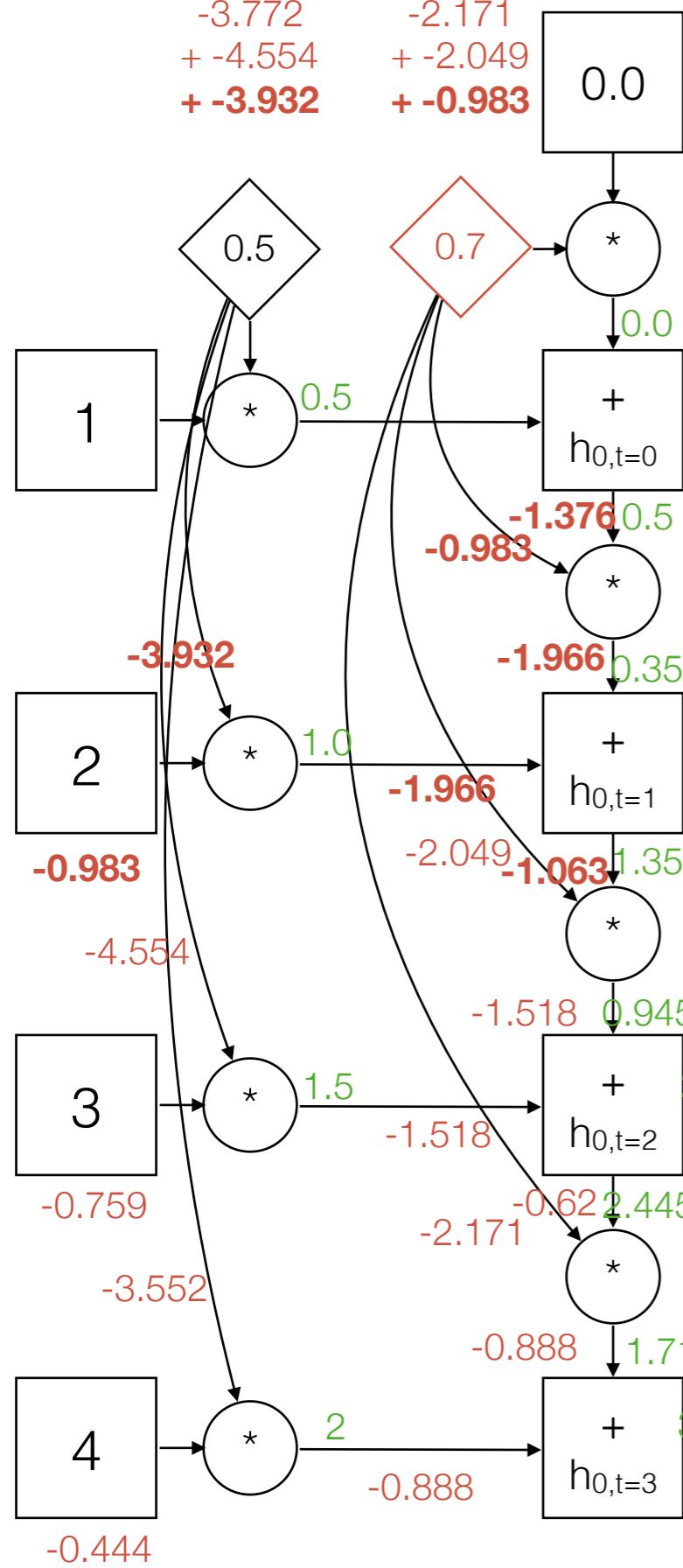


- Now we do the time step before that.
- Note that the -0.62... from the last time step backward pass gets fed into the hidden node in the 2nd to last pass, and similarly its delta gets passed up into the previous time step as well.
- Additionally, note that the deltas for each weight are being accumulated over every time step's backward pass

$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$

$$\begin{aligned} -3.772 \\ + -4.554 \\ + -3.932 \end{aligned}$$

$$\begin{aligned} -2.171 \\ + -2.049 \\ + -0.983 \end{aligned}$$

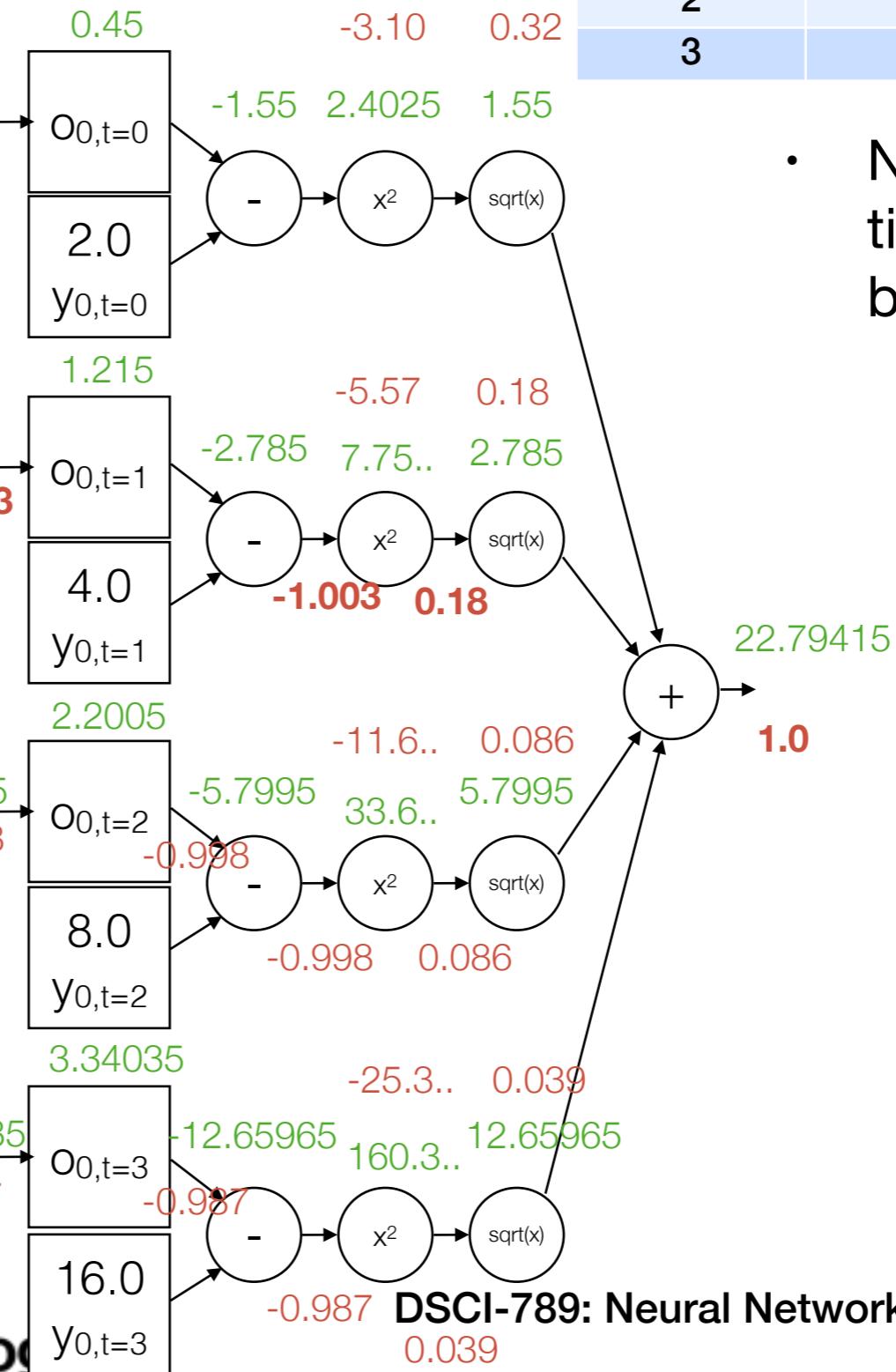


RNN Backward Pass

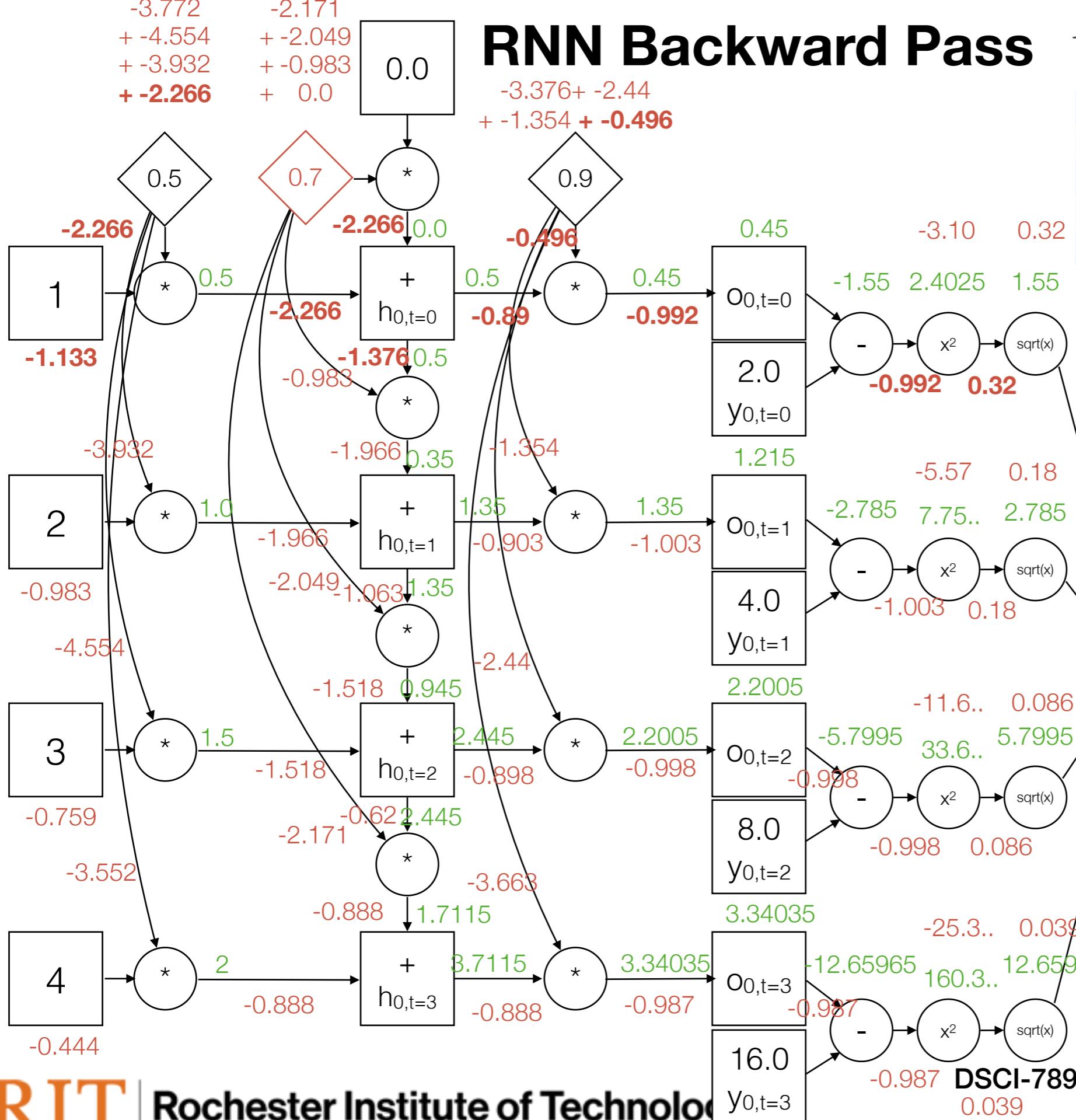
$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$

time (t)	Input	Output
0	1	2
1	2	4
2	3	8
3	4	16

- Now we do the time step before that.



$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$



- And finally we're back to the first time step.
- We can sum up the deltas for each of the weights.

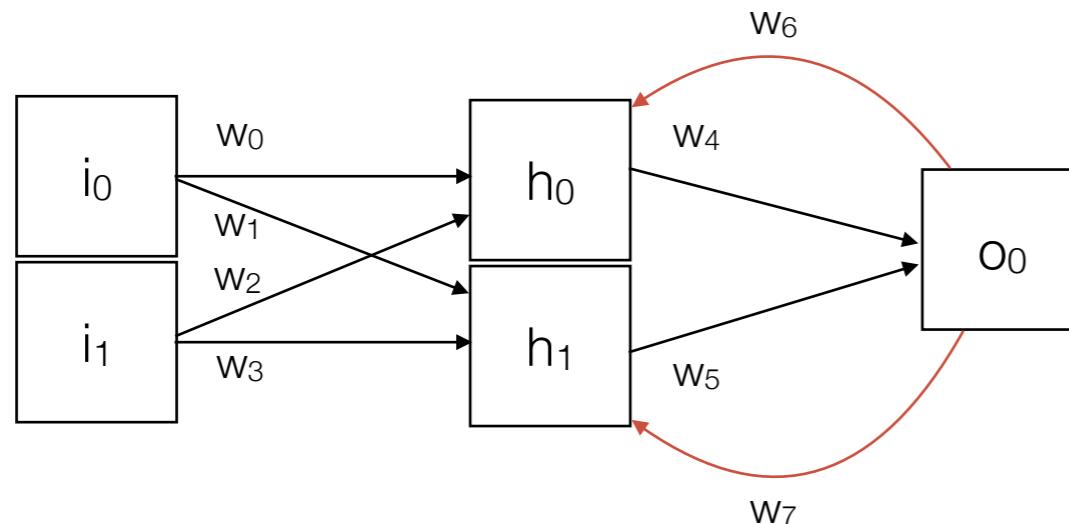
RNN Backward Pass

- Let's try another example with a Jordan network with two hidden nodes. We'll use a more condensed circuit to fit things in a slide.

RNN Backward Pass 2

$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$

time (t)	Input 1	Input 2	Output
0	0.5	0.2	2
1	0.3	0.4	3
2	0.4	0.1	3



- This time we'll use a Jordan network with two hidden nodes and two output nodes (as left).
- Hidden nodes will use the tanh activation function, and the output nodes will use the identity activation function, which will then pass into a L2 norm loss function for the output.
- Inputs and outputs will be as above.
- We can use the following as derivatives for the forward pass:

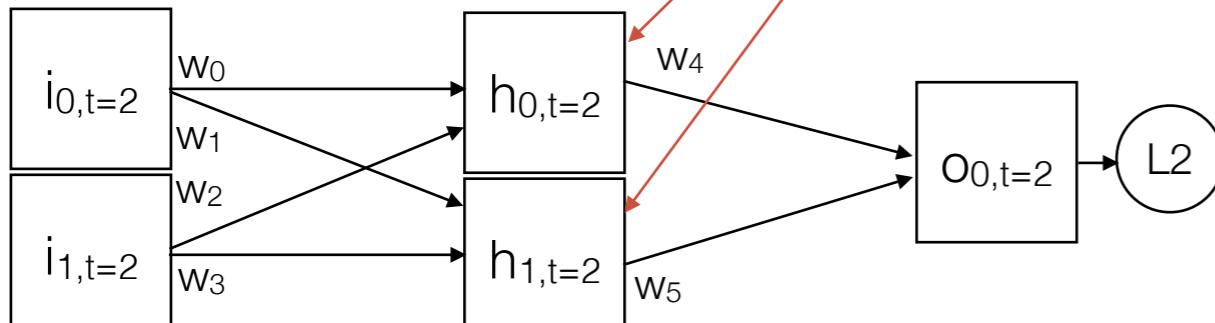
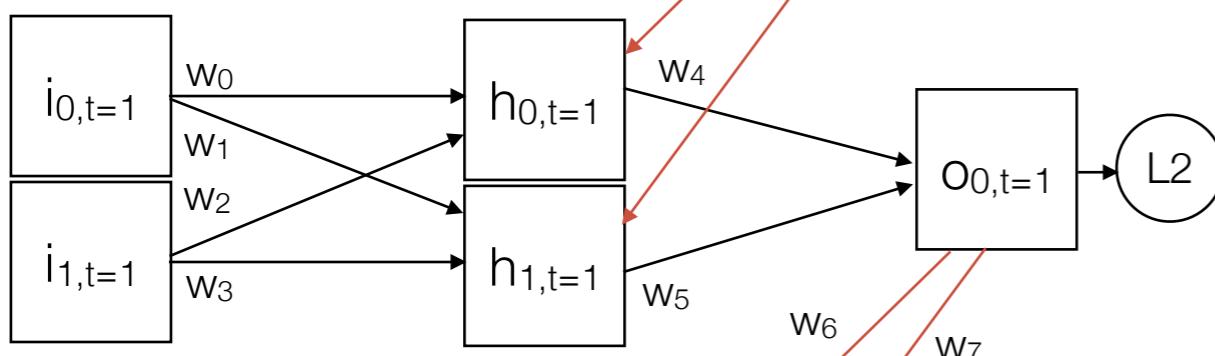
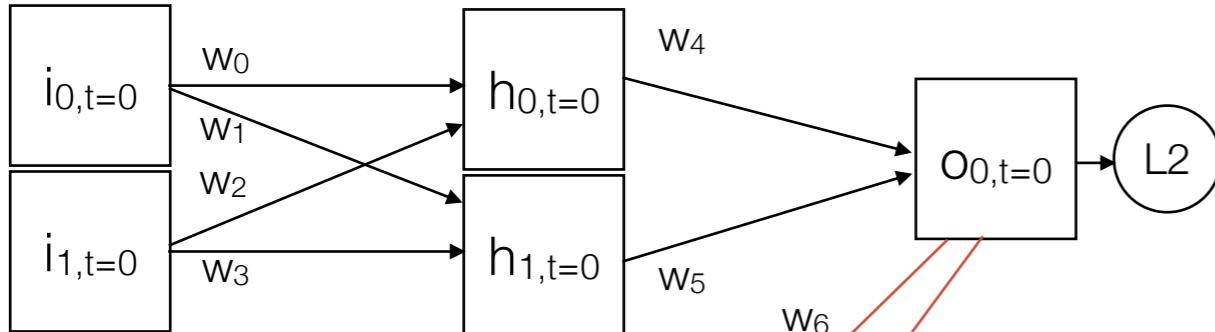
$$\tanh' * (x) = 1 - \tanh^2(x)$$
$$L2'(x) = \frac{x}{\sqrt{x}}, \text{ where size}(x) = 1$$

$$\tanh'(x) = 1 - \tanh^2(x)$$

$$L2'(x) = \frac{x}{\sqrt{|x|}}, \text{ where } \text{size}(x) = 1 \text{ and } x = z_j - y_j$$

RNN Backward Pass 2

$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$



- Let's unroll the network over the 3 time steps.
- We'll use the below to track our weights, derivatives and node values.

time (t)	Input 1	Input 2	Target
0	0.5	0.2	2
1	0.3	0.4	3
2	0.4	0.1	3

Time, Number	H	H d	O	O d
0, 0				
0, 1				
1, 0				
1, 1				
2, 0				
2, 1				

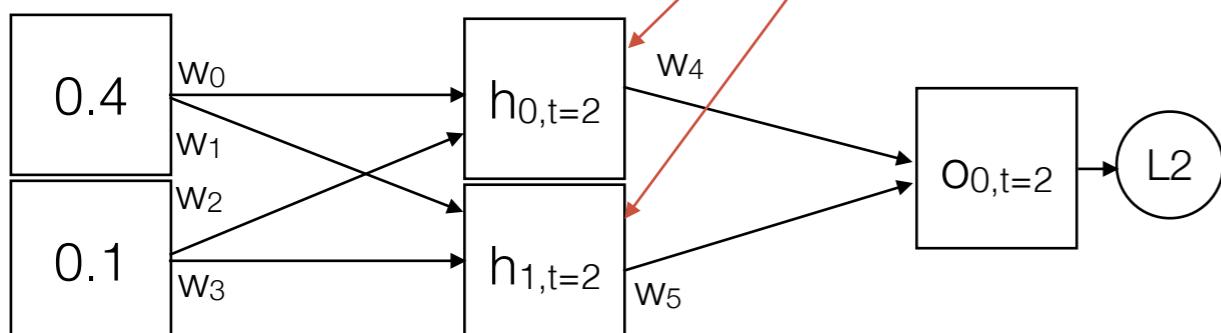
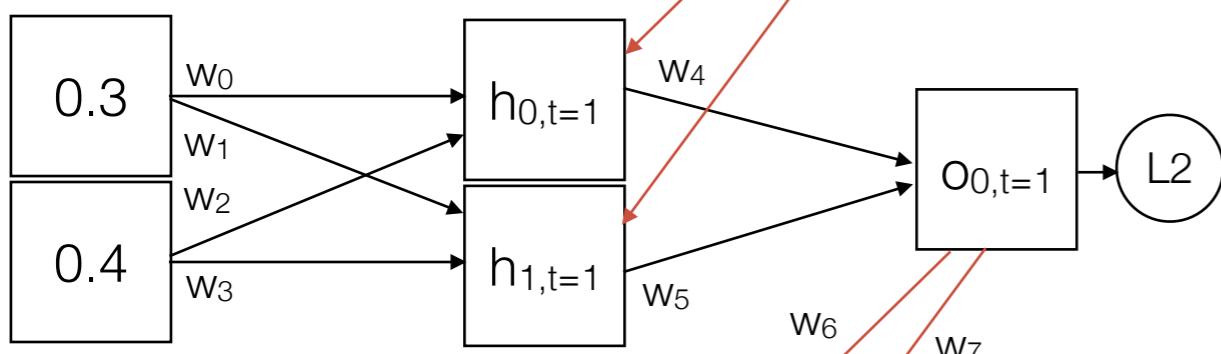
Weight	Value	Delta
1	0.3	0
2	-0.4	0
3	-0.5	0
4	0.2	0
5	-0.1	0
6	-0.6	0
7	0.1	0
8	0.3	0

$$\tanh'(x) = 1 - \tanh^2(x)$$

$$L2'(x) = \frac{x}{\sqrt{|x|}}, \text{ where } \text{size}(x) = 1 \text{ and } x = z_j - y_j$$

RNN Backward Pass 2

$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$



Note that we're multiplying the output of the output nodes before the L2 loss function by the weights into the next time step.

time (t)	Input 1	Input 2	Target
0	0.5	0.2	2
1	0.3	0.4	3
2	0.4	0.1	3

Weight	Value	Delta
0	0.3	0
1	-0.4	0
2	-0.5	0
3	0.2	0
4	-0.1	0
5	-0.6	0
6	0.1	0
7	0.3	0

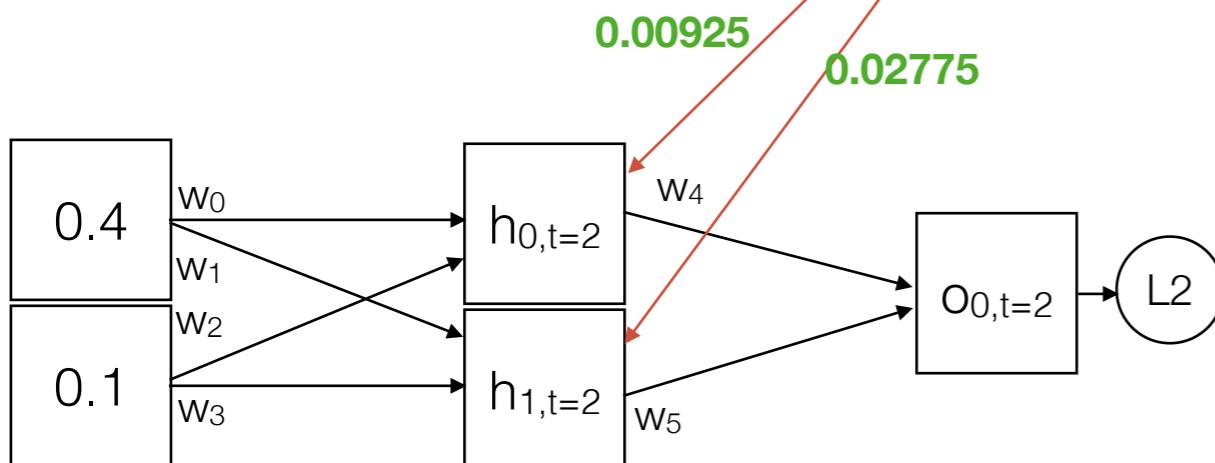
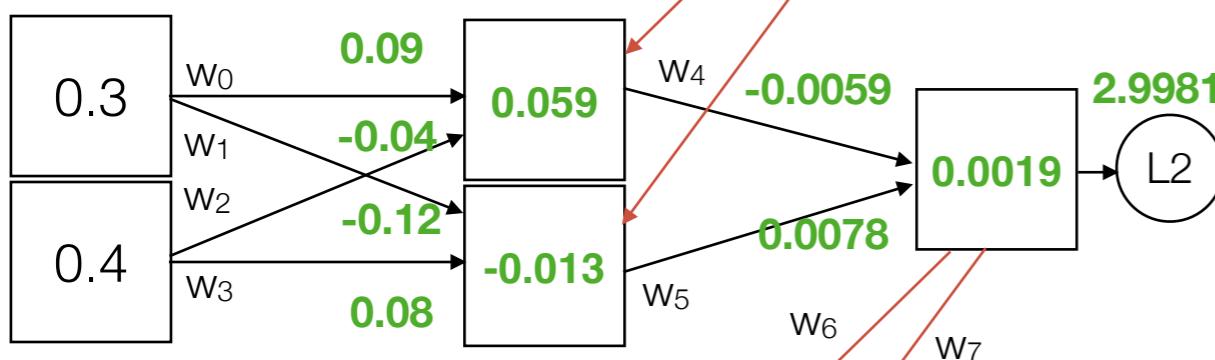
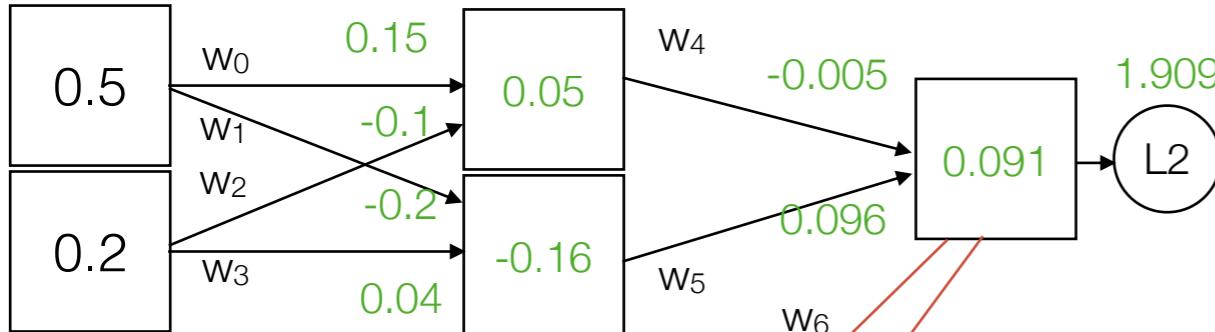
Time, Number	H	H d	O	L2 d
0, 0	$\tanh(0.05) = 0.05$	0.99	0.091	$0.091 / \sqrt{0.091} = 0.30$
0, 1	$\tanh(-0.16) = -0.16$	0.97		
1, 0				
1, 1				
2, 0				
2, 1				

$$\tanh'(x) = 1 - \tanh^2(x)$$

$$L2'(x) = \frac{x}{\sqrt{|x|}}, \text{ where } \text{size}(x) = 1 \text{ and } x = z_j - y_j$$

RNN Backward Pass 2

$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$



Now we do a forward pass for the second time step.

- Don't forget for the inputs to the hidden nodes we need to add the values coming from the previous layer before doing the activation function.

time (t)	Input 1	Input 2	Target
0	0.5	0.2	2
1	0.3	0.4	3
2	0.4	0.1	3

Weight	Value	Delta
0	0.3	0
1	-0.4	0
2	-0.5	0
3	0.2	0
4	-0.1	0
5	-0.6	0
6	0.1	0
7	0.3	0

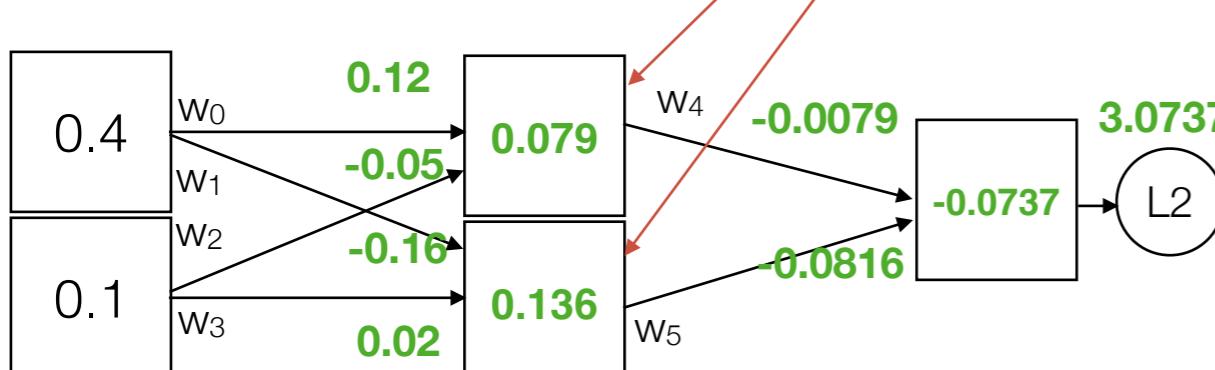
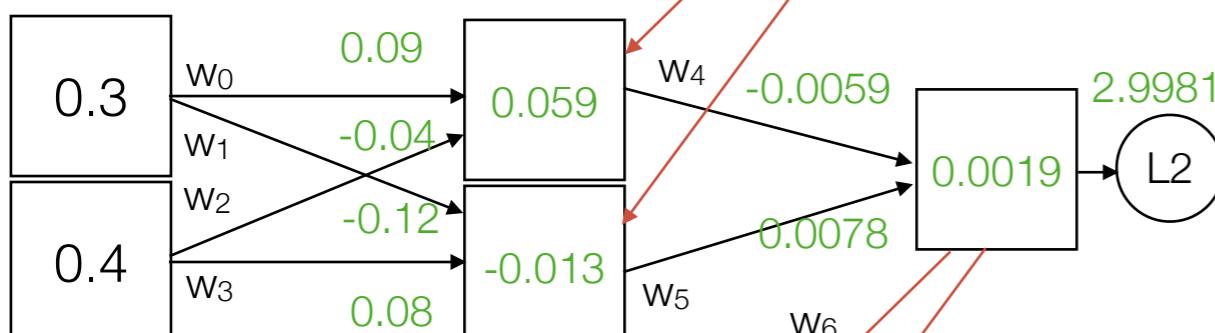
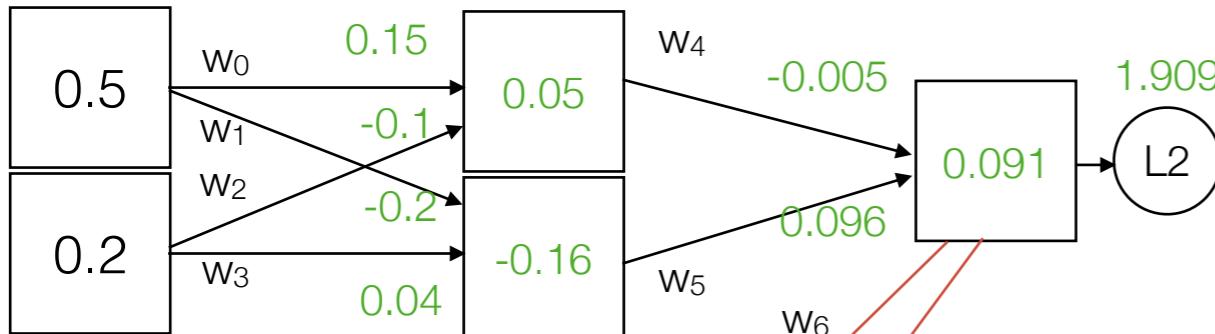
Time, Number	H	H d	O	L2 d
0, 0	$\tanh(0.05) = 0.05$	0.99	0.091	0.30
0, 1	$\tanh(-0.16) = -0.16$	0.97		
1, 0	$\tanh(0.09 - 0.04 + 0.0091) = \tanh(0.059) = 0.059$	0.99	0.0019	$0.0019 / \sqrt{0.0019} = 0.44$
1, 1	$\tanh(-0.12 + 0.08 + 0.0273) = \tanh(-0.0127) = -0.0127$	0.99		
2, 0				
2, 1				

$$\tanh'(x) = 1 - \tanh^2(x)$$

$$L2'(x) = \frac{x}{\sqrt{|x|}}, \text{ where } \text{size}(x) = 1 \text{ and } x = z_j - y_j$$

RNN Backward Pass 2

$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$



Now we repeat for the 3rd time step.

time (t)	Input 1	Input 2	Target
0	0.5	0.2	2
1	0.3	0.4	4
2	0.4	0.1	3

Weight	Value	Delta
0	0.3	0
1	-0.4	0
2	-0.5	0
3	0.2	0
4	-0.1	0
5	-0.6	0
6	0.1	0
7	0.3	0

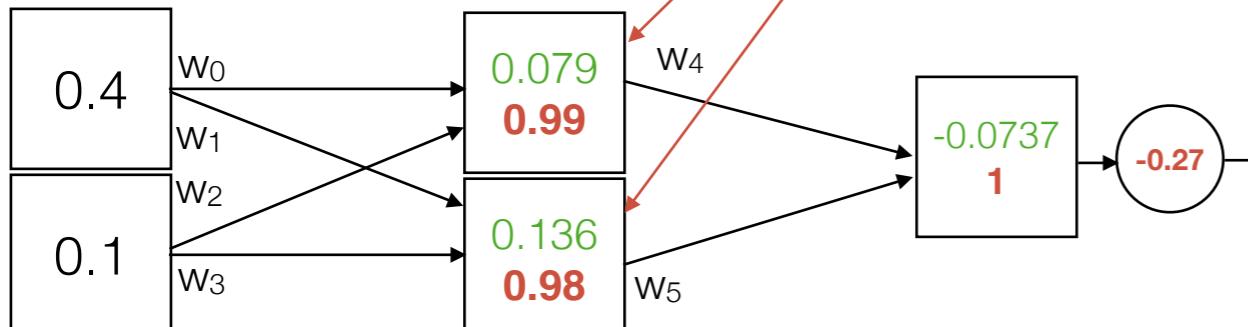
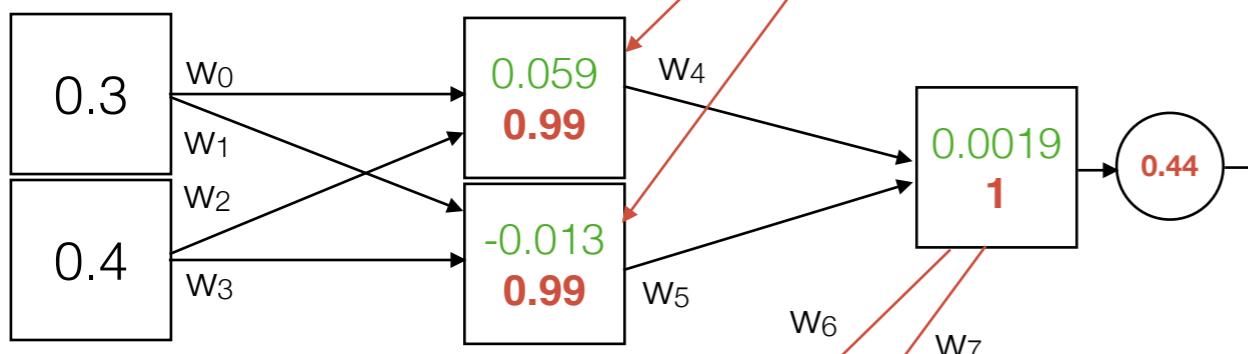
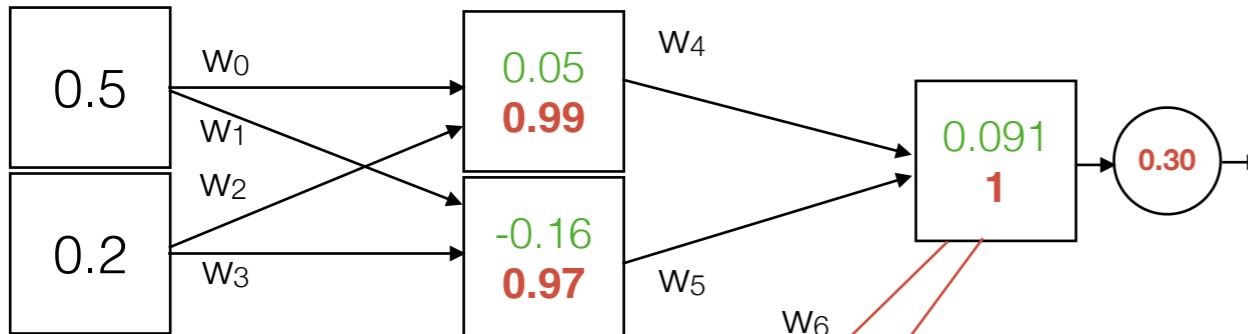
Time, Number	H	H d	O	L2 d
0, 0	$\tanh(0.05) = 0.05$	0.99	0.091	0.30
0, 1	$\tanh(-0.16) = -0.16$	0.97		
1, 0	$\tanh(0.059) = 0.059$	0.99	0.0019	0.44
1, 1	$\tanh(-0.0127) = -0.0127$	0.99		
2, 0	$\tanh(0.00925 + 0.12 - 0.05) = \tanh(0.07925) = 0.079$	0.99	-0.0737	$-0.0737 / \sqrt{0.0737} = -0.27$
2, 1	$\tanh(0.02775 - 0.16 + 0.02) = \tanh(0.1375) = 0.136$	0.98		

$$\tanh'(x) = 1 - \tanh^2(x)$$

$$L2'(x) = \frac{x}{\sqrt{|x|}}, \text{ where } \text{size}(x) = 1 \text{ and } x = z_j - y_j$$

RNN Backward Pass 2

$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$



Now we can go through, and set up all the derivatives we need to calculate the backward pass.

- We need to know the output of each node to calculate the deltas to the weights, but all the other forward pass values we can ignore.

time (t)	Input 1	Input 2	Target
0	0.5	0.2	2
1	0.3	0.4	4
2	0.4	0.1	3

Weight	Value	Delta
0	0.3	0
1	-0.4	0
2	-0.5	0
3	0.2	0
4	-0.1	0
5	-0.6	0
6	0.1	0
7	0.3	0

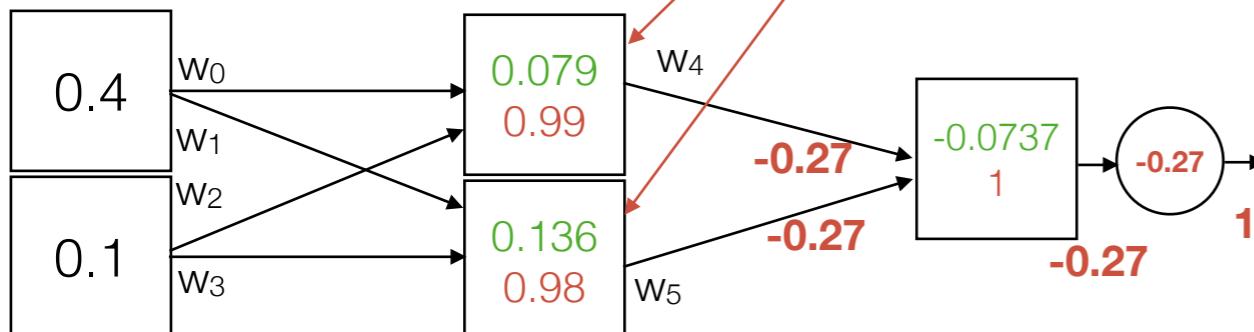
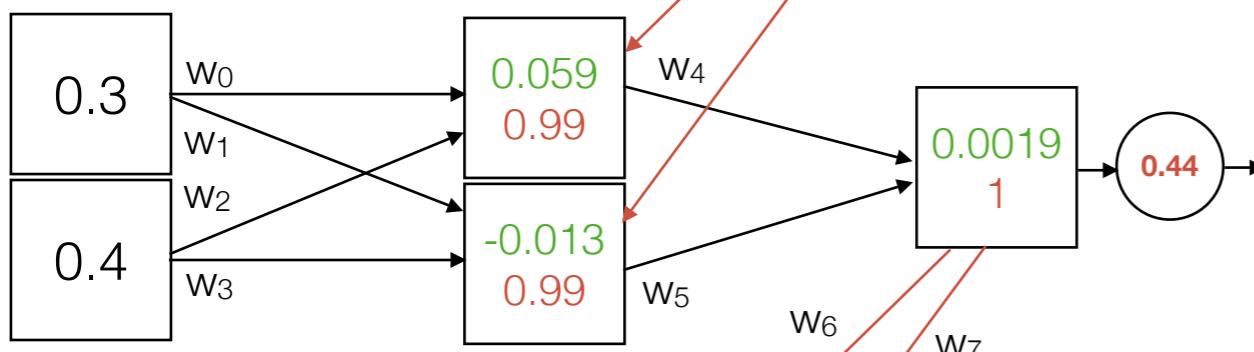
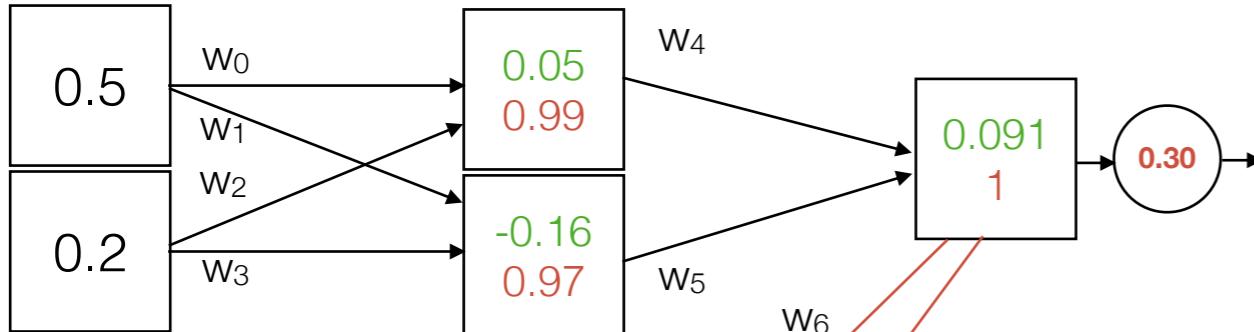
Time, Number	H	H d	O	L2 d
0, 0	0.05	0.99	0.091	0.30
0, 1	-0.16	0.97		
1, 0	0.059	0.99	0.0019	0.44
1, 1	-0.0127	0.99		
2, 0	0.079	0.99	-0.0737	-0.27
2, 1	0.136	0.98		

$$\tanh'(x) = 1 - \tanh^2(x)$$

$$L2'(x) = \frac{x}{\sqrt{|x|}}, \text{ where } \text{size}(x) = 1 \text{ and } x = z_j - y_j$$

RNN Backward Pass 2

$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$



Now let's do a backward pass starting at the last time step.

time (t)	Input 1	Input 2	Target
0	0.5	0.2	2
1	0.3	0.4	4
2	0.4	0.1	3

Weight	Value	Delta
0	0.3	0
1	-0.4	0
2	-0.5	0
3	0.2	0
4	-0.1	0
5	-0.6	0
6	0.1	0
7	0.3	0

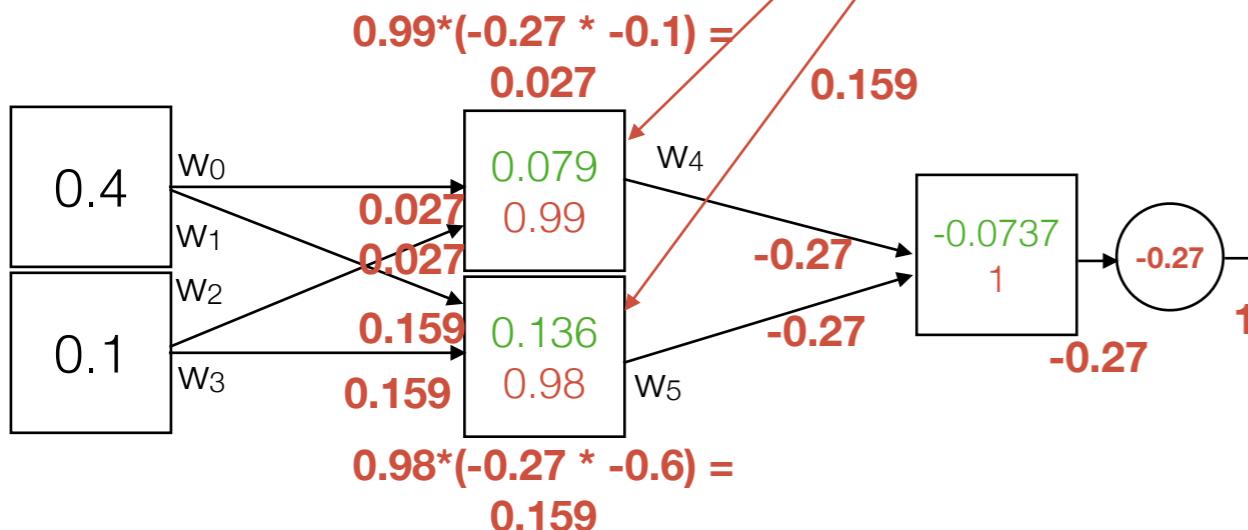
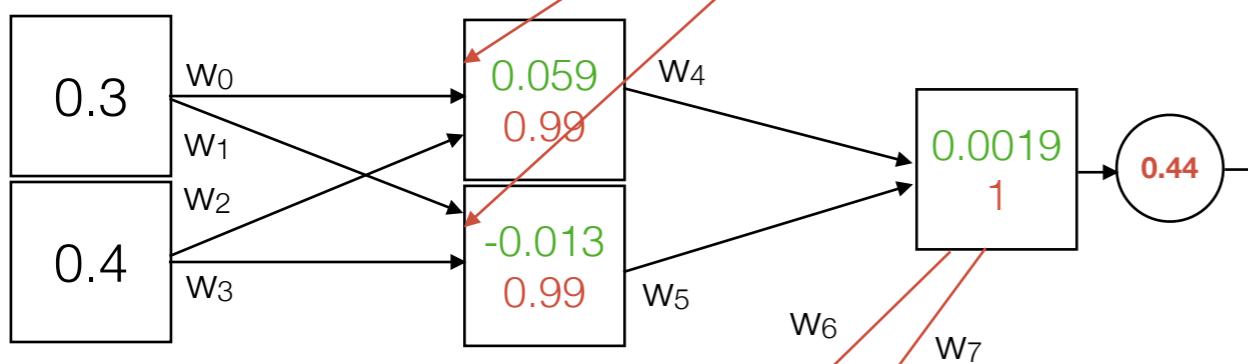
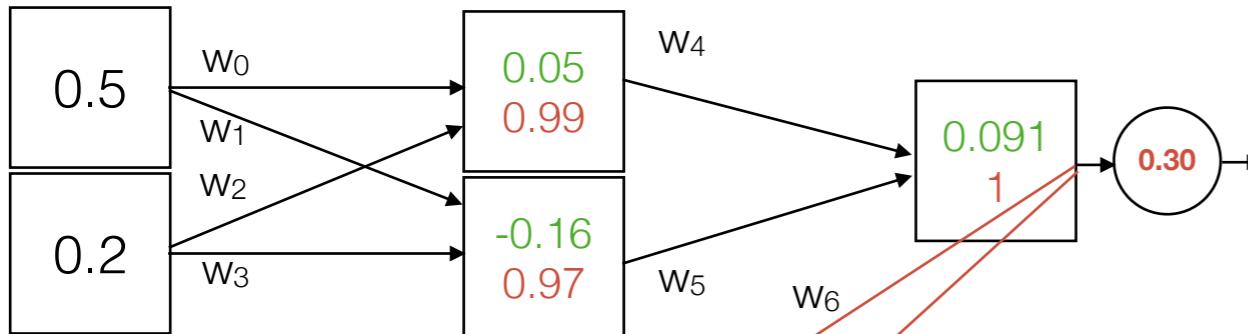
Time, Number	H	H d	O	L2 d
0, 0	0.05	0.99	0.091	0.30
0, 1	-0.16	0.97		
1, 0	0.059	0.99	0.0019	0.44
1, 1	-0.0127	0.99		
2, 0	0.079	0.99	-0.0737	-0.27
2, 1	0.136	0.98		

$$\tanh'(x) = 1 - \tanh^2(x)$$

$$L2'(x) = \frac{x}{\sqrt{|x|}}, \text{ where } \text{size}(x) = 1 \text{ and } x = z_j - y_j$$

RNN Backward Pass 2

$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$



- The delta into the hidden nodes is equal to the incoming delta multiplied by the respective weights.
- The delta into the weights is the incoming delta multiplied by the output of the hidden nodes.
- Note that I am removing significant figures just so things show up on the slides nicer.

time (t)	Input 1	Input 2	Target
0	0.5	0.2	2
1	0.3	0.4	4
2	0.4	0.1	3

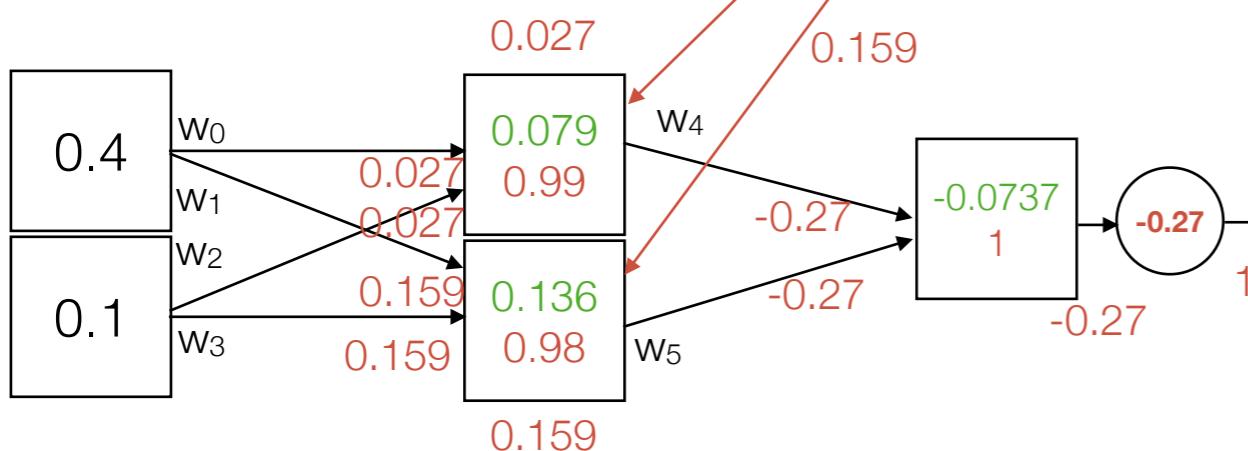
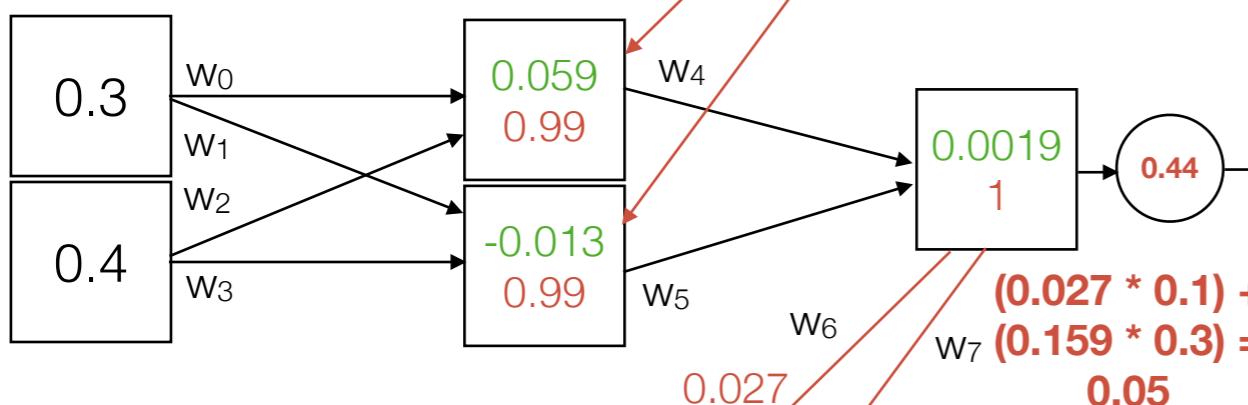
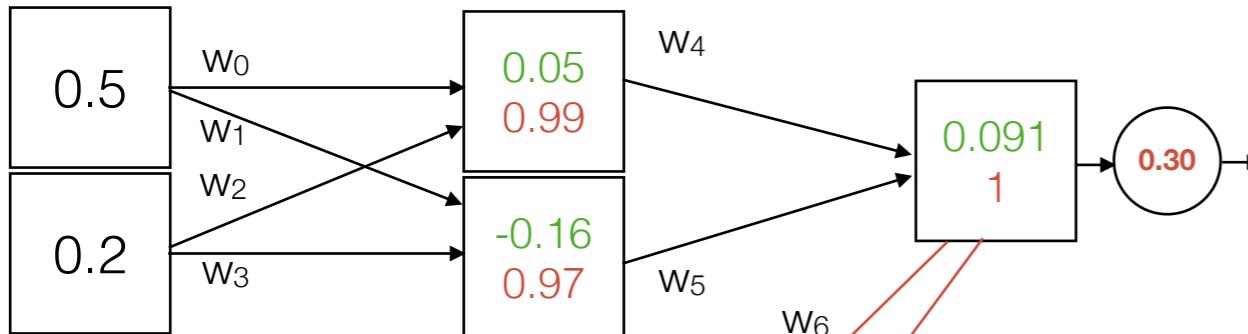
Weight	Value	Delta
0	0.3	0
1	-0.4	0
2	-0.5	0
3	0.2	0
4	-0.1	$-0.27 * 0.079 = -0.021$
5	-0.6	$-0.27 * 0.136 = -0.037$
6	0.1	0
7	0.3	0

$$\tanh'(x) = 1 - \tanh^2(x)$$

$$L2'(x) = \frac{x}{\sqrt{|x|}}, \text{ where } \text{size}(x) = 1 \text{ and } x = z_j - y_j$$

RNN Backward Pass 2

$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$



- Now we can backprop into weights 0-4 and weights 6 and 7.
- Note the error into the output node from the two jordan recurrent edges is the sum of the incoming deltas times those weights.

time (t)	Input 1	Input 2	Target
0	0.5	0.2	2
1	0.3	0.4	4
2	0.4	0.1	3

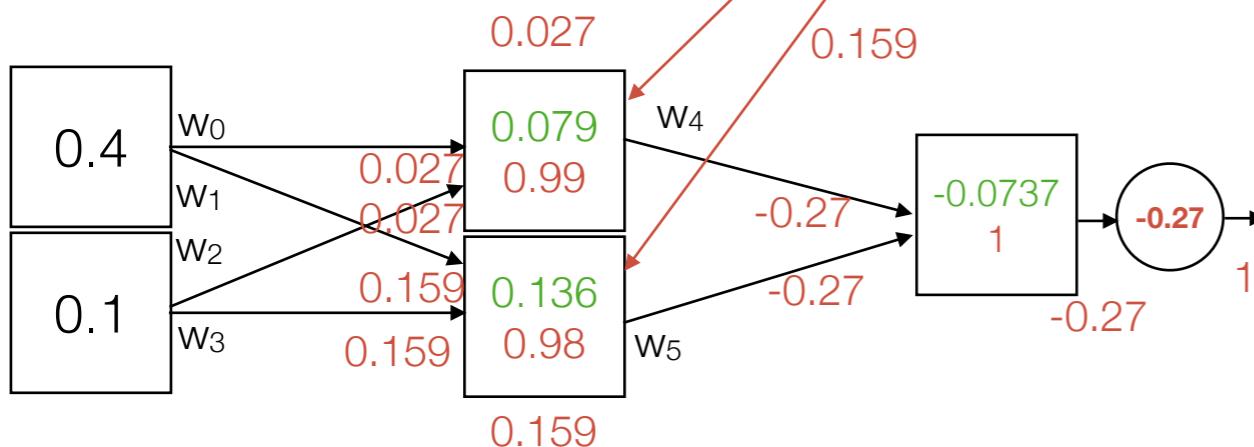
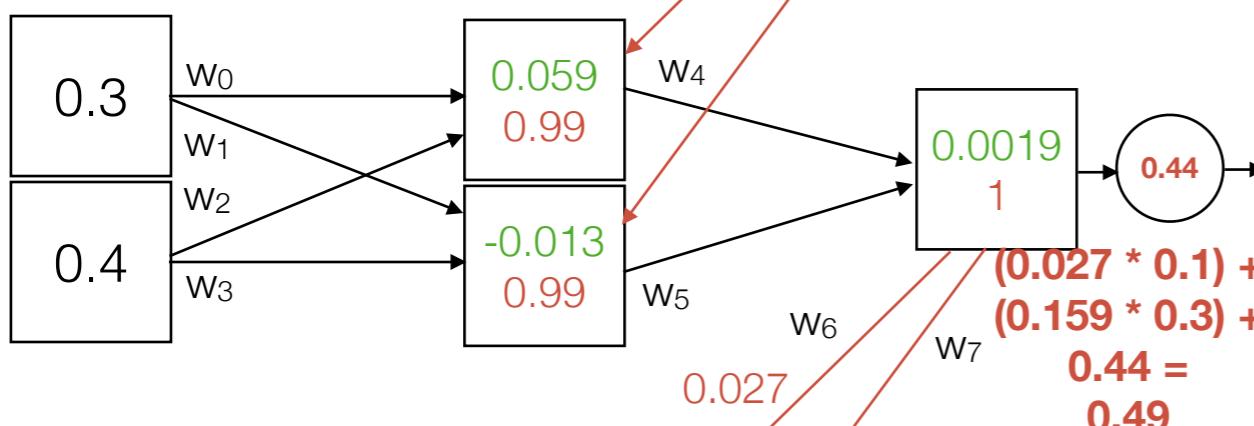
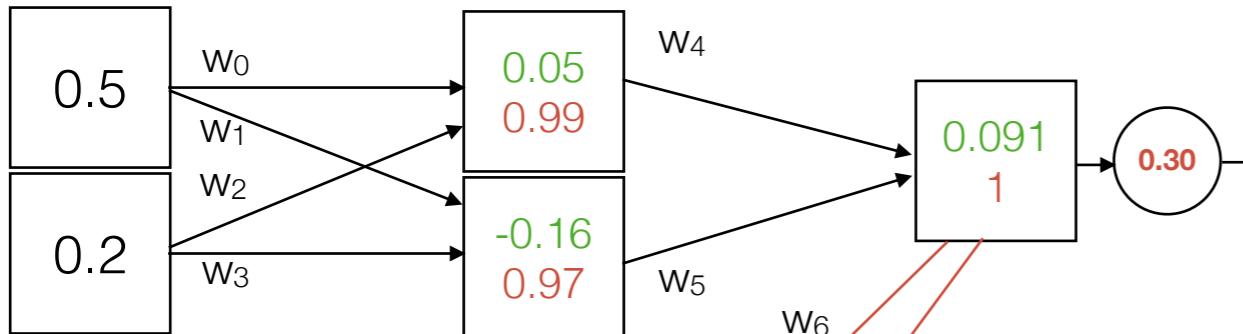
Weight	Value	Delta
0	0.3	(0.027 * 0.4) = 0.011
1	-0.4	(0.159 * 0.4) = 0.064
2	-0.5	(0.027 * 0.1) = 0.0027
3	0.2	(0.159 * 0.1) = 0.0016
4	-0.1	-0.021
5	-0.6	-0.037
6	0.1	(0.027 * 0.0019) = 0.0000513
7	0.3	(0.159 * 0.0019) = 0.0003021

$$\tanh'(x) = 1 - \tanh^2(x)$$

$$L2'(x) = \frac{x}{\sqrt{|x|}}, \text{ where } \text{size}(x) = 1 \text{ and } x = z_j - y_j$$

RNN Backward Pass 2

$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$



- Doing backprop on the second time step we need to remember to add the incoming deltas from the last time step we did to the incoming delta from the loss function.

time (t)	Input 1	Input 2	Target
0	0.5	0.2	2
1	0.3	0.4	4
2	0.4	0.1	3

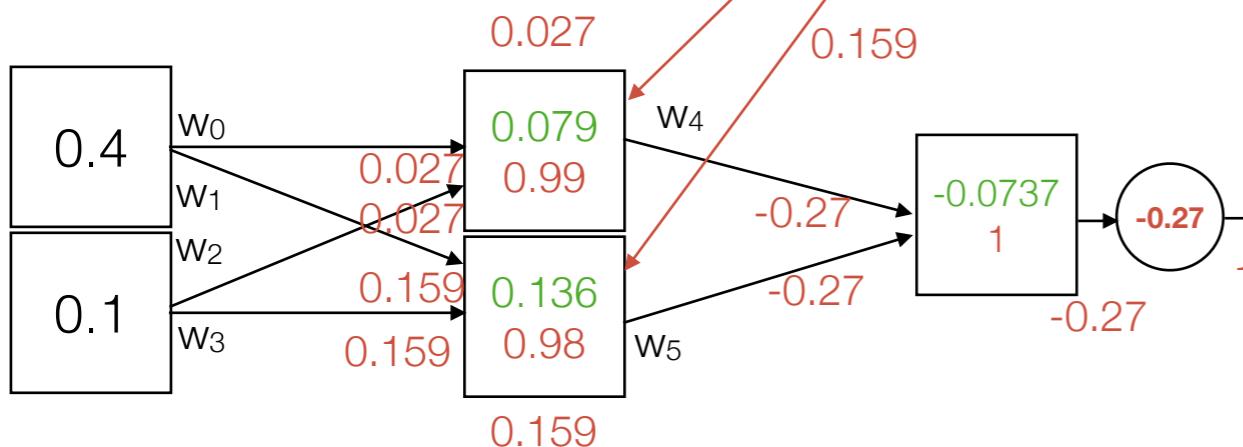
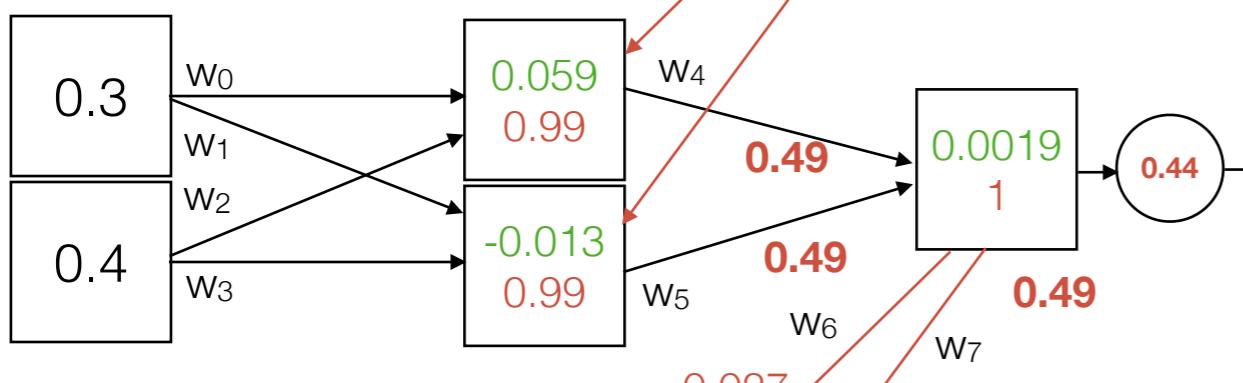
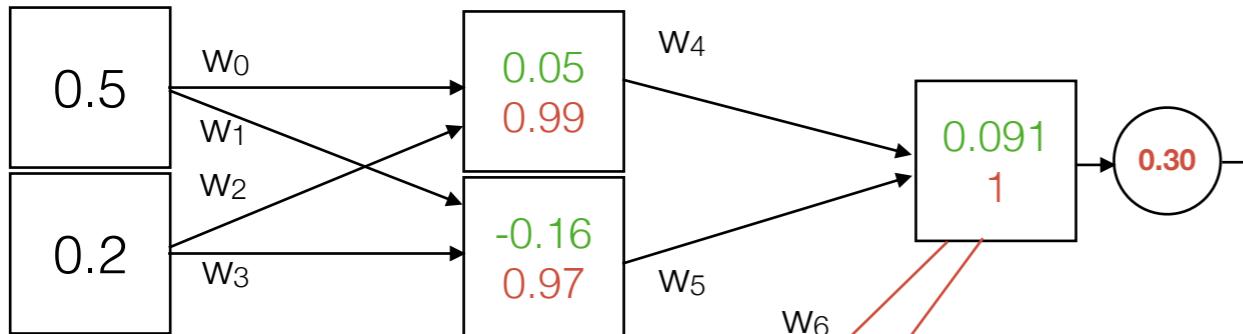
Weight	Value	Delta
0	0.3	0.011
1	-0.4	0.064
2	-0.5	0.0027
3	0.2	0.0016
4	-0.1	-0.021
5	-0.6	-0.037
6	0.1	0.0000513
7	0.3	0.0003021

$$\tanh'(x) = 1 - \tanh^2(x)$$

$$L2'(x) = \frac{x}{\sqrt{|x|}}, \text{ where } \text{size}(x) = 1 \text{ and } x = z_j - y_j$$

RNN Backward Pass 2

$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$



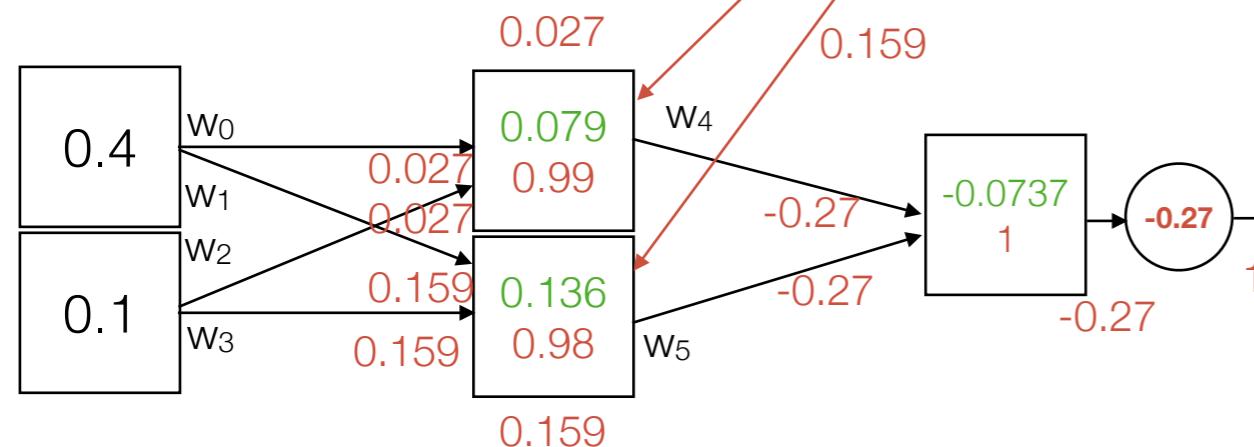
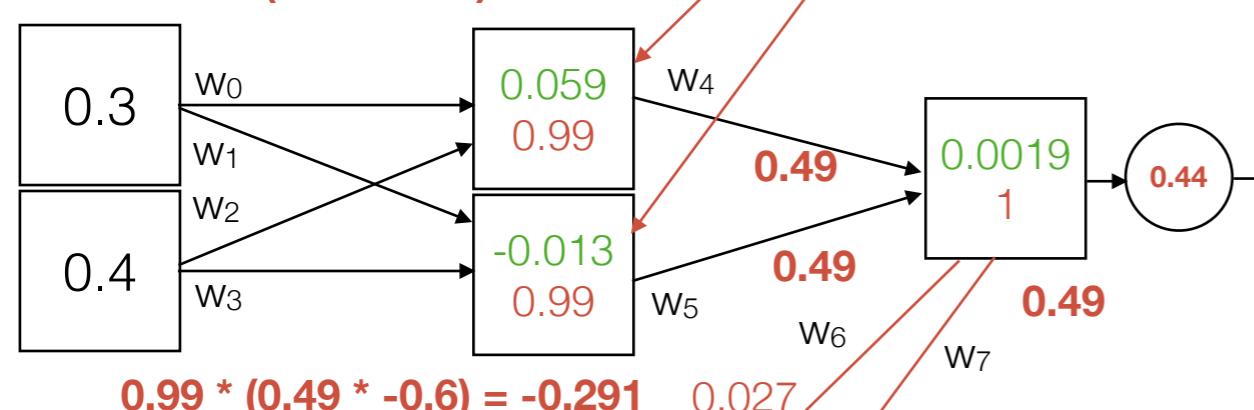
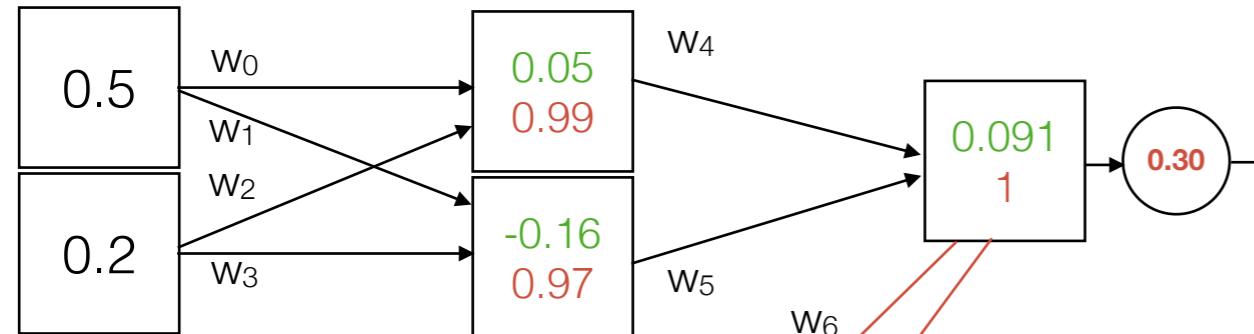
- Doing backprop on the second time step we need to remember to add the incoming deltas from the last time step we did to the incoming delta from the loss function.

time (t)	Input 1	Input 2	Target
0	0.5	0.2	2
1	0.3	0.4	4
2	0.4	0.1	3

Weight	Value	Delta
0	0.3	0.011
1	-0.4	0.064
2	-0.5	0.0027
3	0.2	0.0016
4	-0.1	-0.021
5	-0.6	-0.037
6	0.1	0.0000513
7	0.3	0.0003021

$$\tanh'(x) = 1 - \tanh^2(x)$$

$$L2'(x) = \frac{x}{\sqrt{|x|}}, \text{ where } \text{size}(x) = 1 \text{ and } x = z_j - y_j$$



RNN Backward Pass 2

$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$

time (t)	Input 1	Input 2	Target
0	0.5	0.2	2
1	0.3	0.4	4
2	0.4	0.1	3

- Now we do a similar process to update the weight deltas for weight 4 and 5 (multiply the output of the hidden node by the incoming delta) as well as the deltas into the hidden nodes (multiply the weight by the incoming delta).

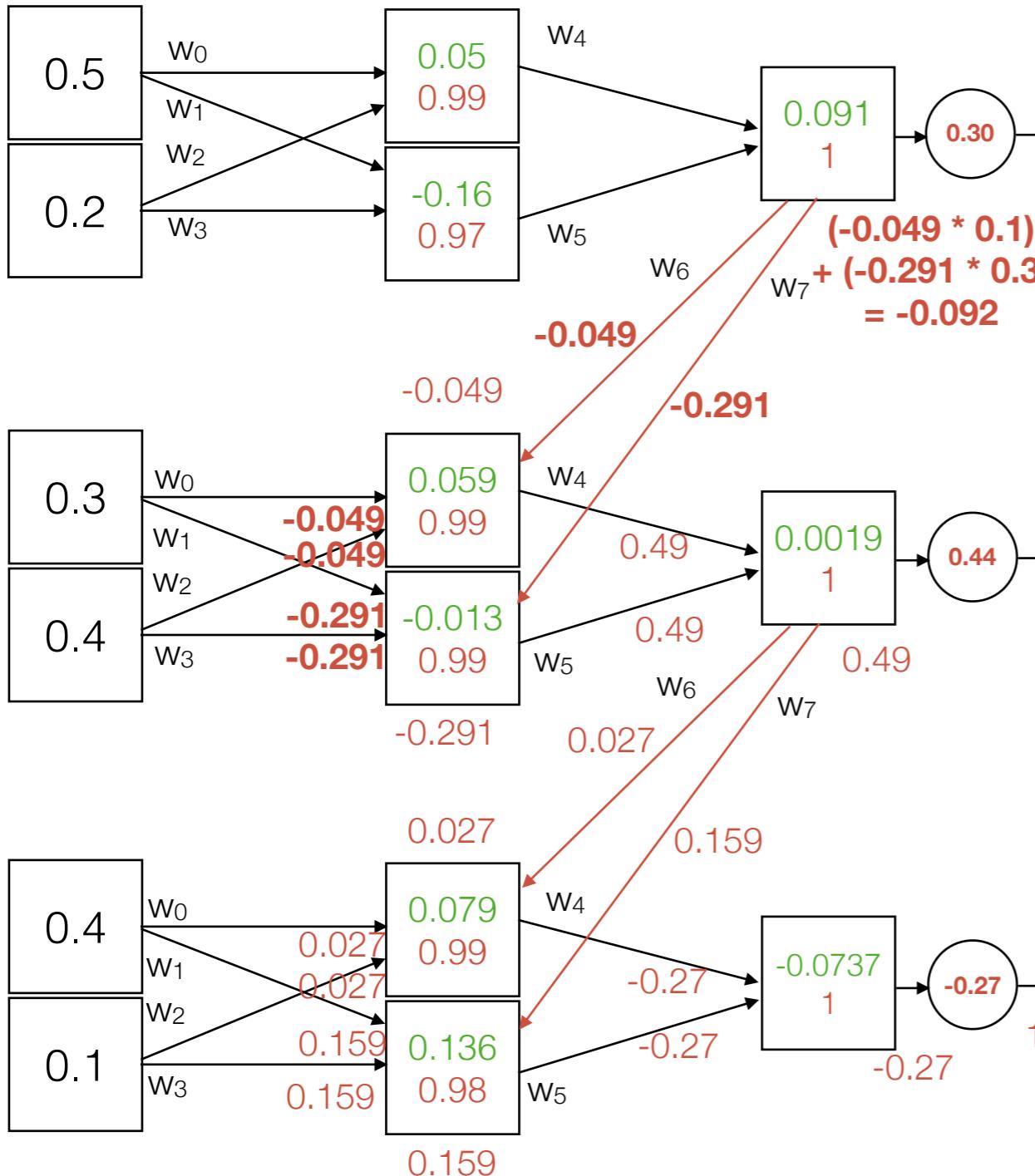
Weight	Value	Delta
0	0.3	0.011
1	-0.4	0.064
2	-0.5	0.0027
3	0.2	0.0016
4	-0.1	-0.021 + (0.49 * 0.059) = -0.008
5	-0.6	-0.037 + (0.49 * -0.013) = -0.043
6	0.1	0.0000513
7	0.3	0.0003021

$$\tanh'(x) = 1 - \tanh^2(x)$$

$$L2'(x) = \frac{x}{\sqrt{|x|}}, \text{ where } \text{size}(x) = 1 \text{ and } x = z_j - y_j$$

RNN Backward Pass 2

$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$



- We can then backprop into weights 0-4 and weights 6 and 7 for the recurrent edges.
- We can also calculate the delta into the previous time step's output node.

time (t)	Input 1	Input 2	Target
0	0.5	0.2	2
1	0.3	0.4	4
2	0.4	0.1	3

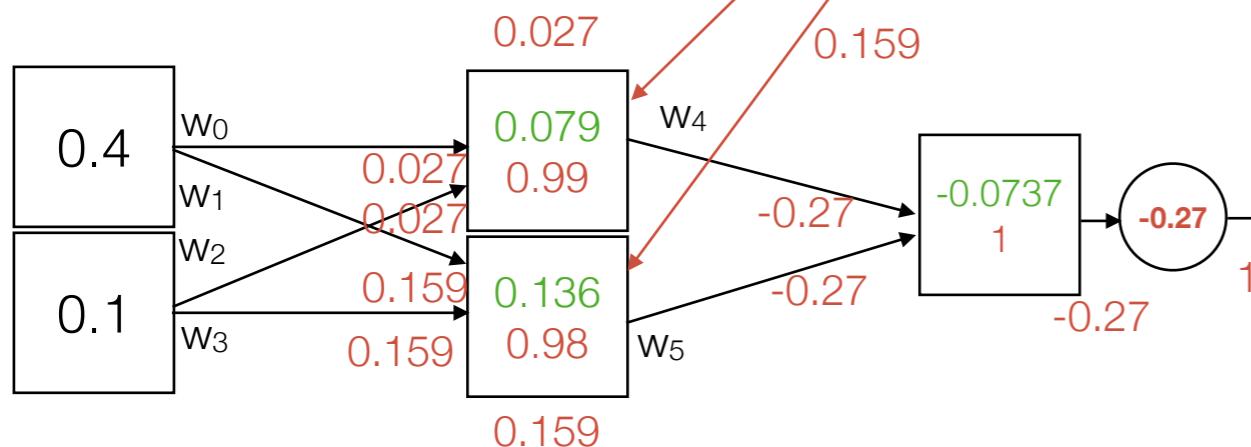
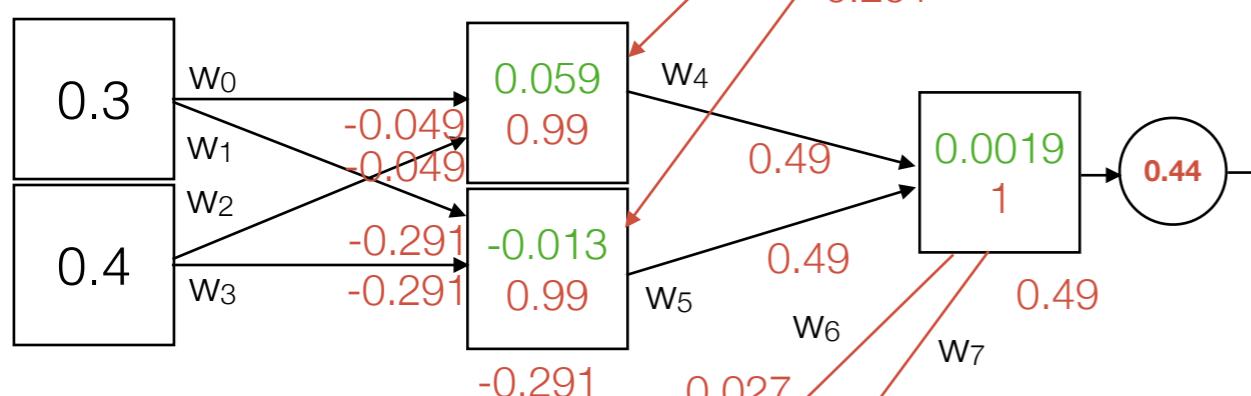
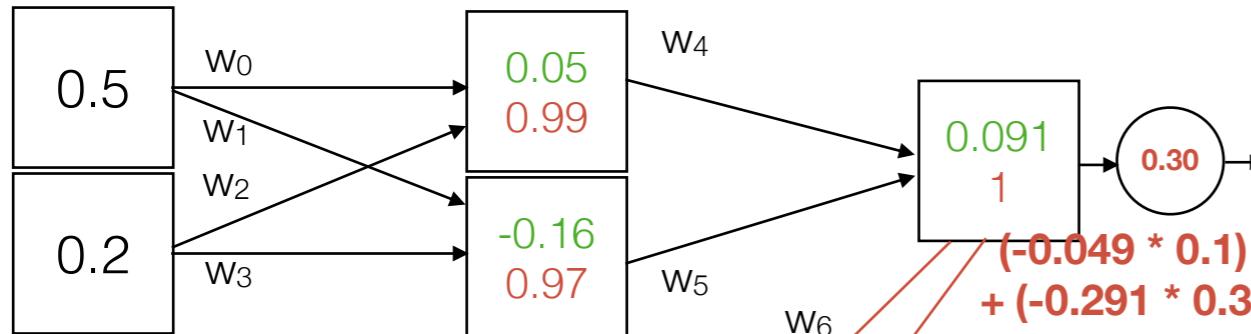
Weight	Value	Delta
0	0.3	$0.011 + (0.3 * -0.049) = -0.004$
1	-0.4	$0.064 + (0.3 * -0.291) = -0.023$
2	-0.5	$0.0027 + (0.4 * -0.049) = -0.017$
3	0.2	$0.0016 + (0.4 * -0.291) = -0.115$
4	-0.1	-0.008
5	-0.6	-0.043
6	0.1	$0.0000513 + (0.091 * -0.049) = -0.0044077$
7	0.3	$0.0003021 + (0.091 * -0.291) = -0.0261789$

$$\tanh'(x) = 1 - \tanh^2(x)$$

$$L2'(x) = \frac{x}{\sqrt{|x|}}, \text{ where } \text{size}(x) = 1 \text{ and } x = z_j - y_j$$

RNN Backward Pass 2

$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$



- Now we can backprop the first time step, again don't forget to add the incoming delta from the proceeding time step with the delta from the loss function into the output node.

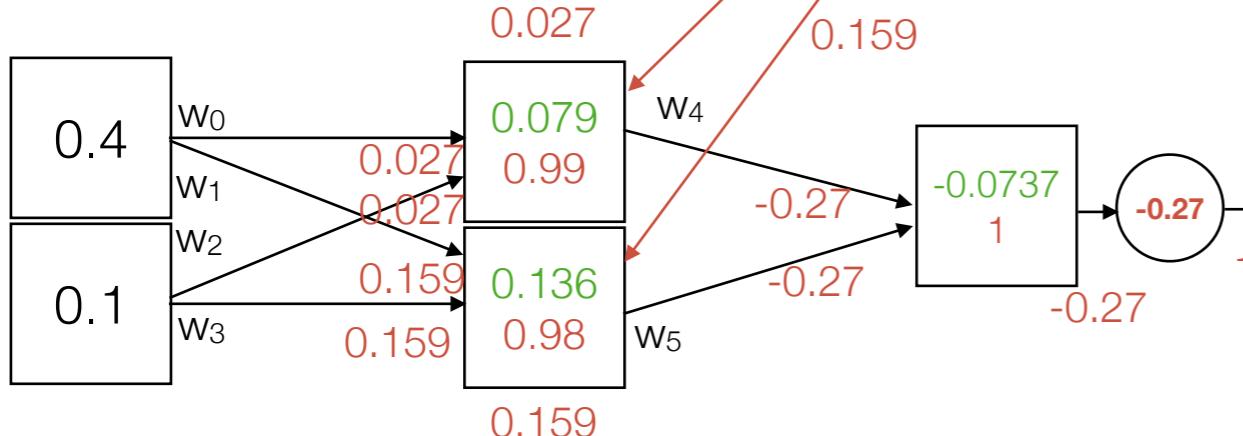
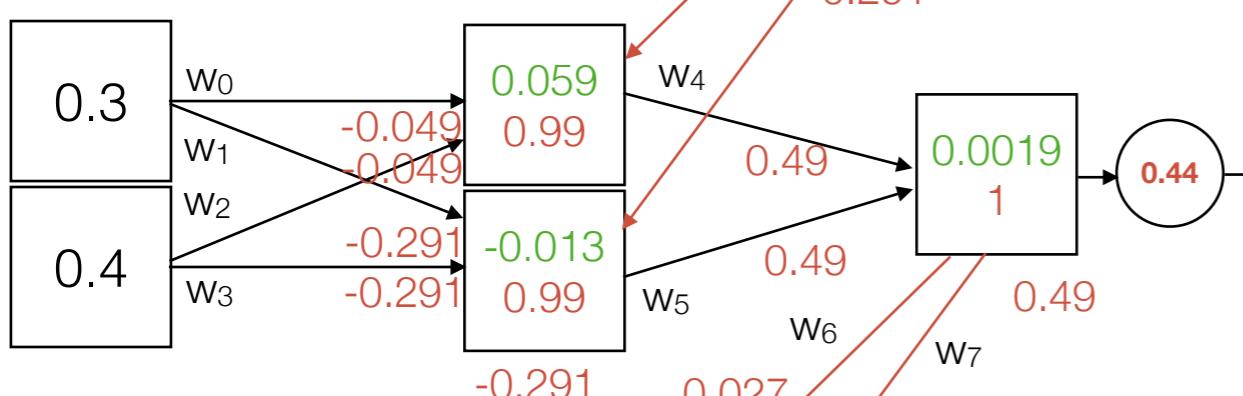
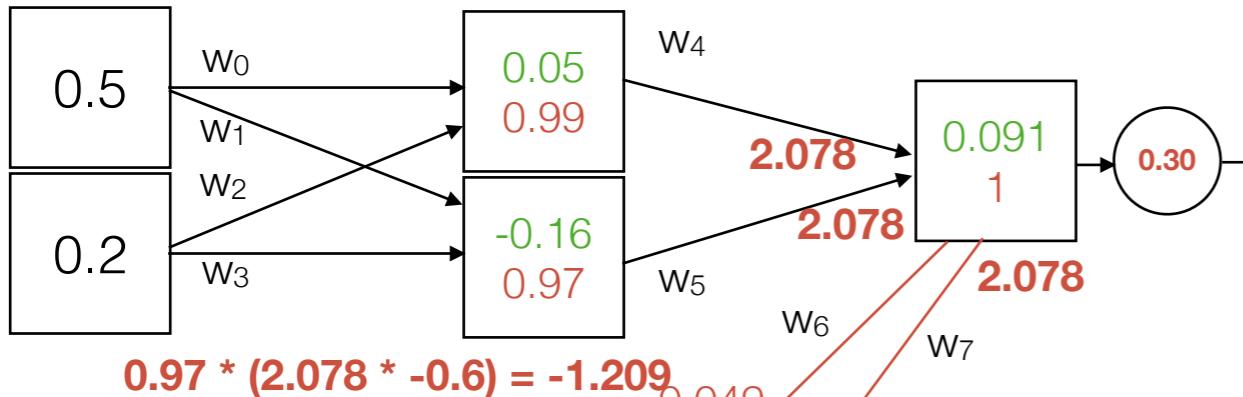
time (t)	Input 1	Input 2	Target
0	0.5	0.2	2
1	0.3	0.4	4
2	0.4	0.1	3

Weight	Value	Delta
0	0.3	-0.004
1	-0.4	-0.023
2	-0.5	-0.017
3	0.2	-0.115
4	-0.1	-0.008
5	-0.6	-0.043
6	0.1	-0.0044077
7	0.3	-0.0261789

$$\tanh'(x) = 1 - \tanh^2(x)$$

$$L2'(x) = \frac{x}{\sqrt{|x|}}, \text{ where } \text{size}(x) = 1 \text{ and } x = z_j - y_j$$

$$0.99 * (2.078 * -0.1) = -0.206$$



RNN Backward Pass 2

$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$

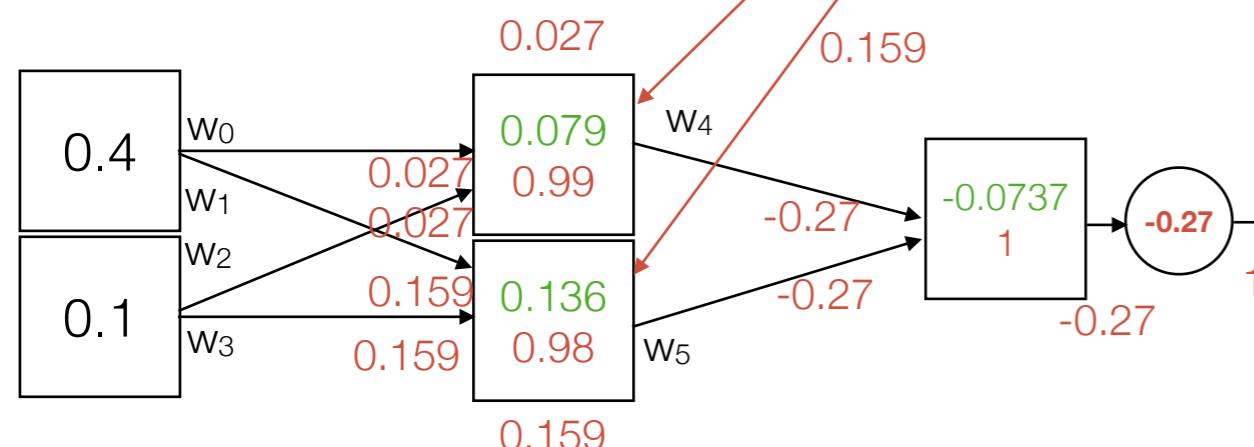
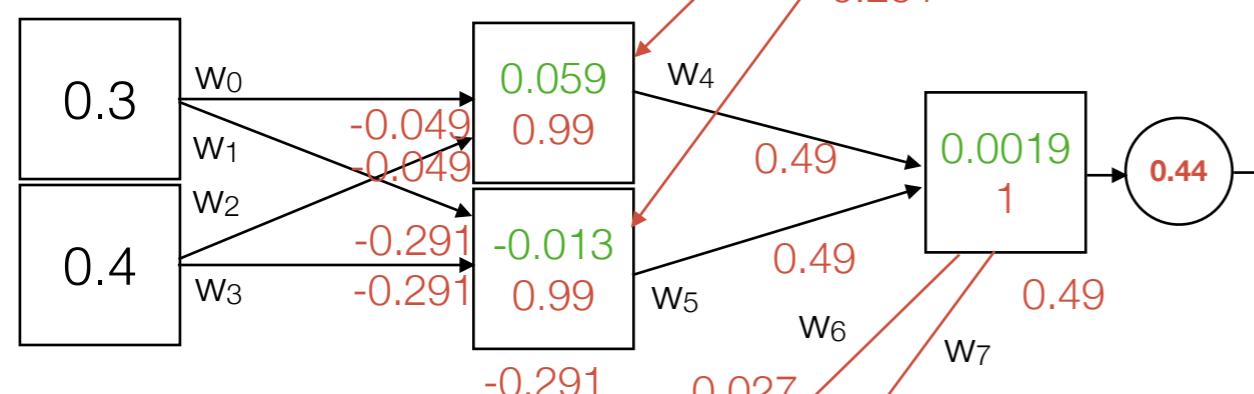
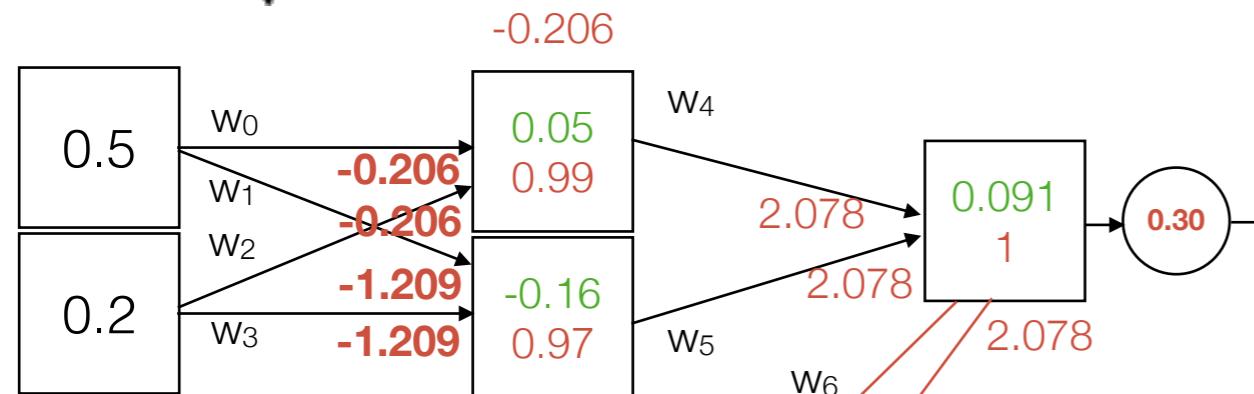
time (t)	Input 1	Input 2	Target
0	0.5	0.2	2
1	0.3	0.4	4
2	0.4	0.1	3

- We can now do the final delta update for weights 4 and 5 and calculate the delta into the hidden nodes.

Weight	Value	Delta
0	0.3	-0.004
1	-0.4	-0.023
2	-0.5	-0.017
3	0.2	-0.115
4	-0.1	$-0.008 + (2.078 * 0.05) = 0.096$
5	-0.6	$-0.043 + (2.078 * -0.16) = -0.375$
6	0.1	-0.0044077
7	0.3	-0.0261789

$$\tanh'(x) = 1 - \tanh^2(x)$$

$$L2'(x) = \frac{x}{\sqrt{|x|}}, \text{ where } \text{size}(x) = 1 \text{ and } x = z_j - y_j$$



RNN Backward Pass 2

$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$

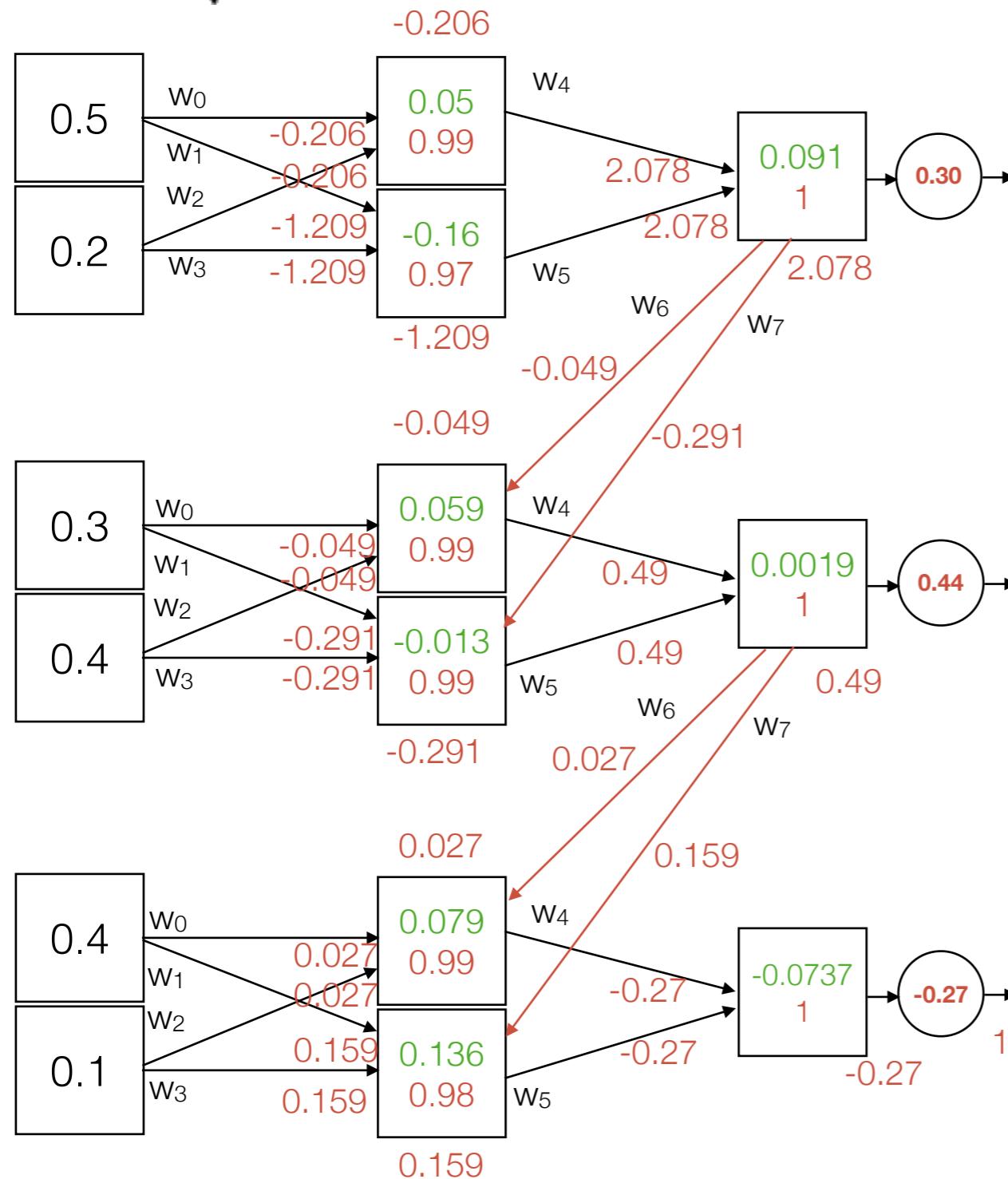
time (t)	Input 1	Input 2	Target
0	0.5	0.2	2
1	0.3	0.4	4
2	0.4	0.1	3

- And then back into weights 0-4.
- Note we do not need to have to do weights 6 and 7 because there was not a previous pass.

Weight	Value	Delta
0	0.3	-0.004 + (-0.206 * 0.5) = -0.107
1	-0.4	-0.023 + (-1.209 * 0.5) = -0.628
2	-0.5	-0.017 + (-0.206 * 0.2) = -0.058
3	0.2	-0.115 + (-1.209 * 0.2) = -0.357
4	-0.1	0.096
5	-0.6	-0.375
6	0.1	-0.0044077
7	0.3	-0.0261789

$$\tanh'(x) = 1 - \tanh^2(x)$$

$$L2'(x) = \frac{x}{\sqrt{|x|}}, \text{ where } \text{size}(x) = 1 \text{ and } x = z_j - y_j$$



RNN Backward Pass 2

$$L2 = \sqrt{(\sum_j |z_j - y_j|^2)}$$

time (t)	Input 1	Input 2	Target
0	0.5	0.2	2
1	0.3	0.4	4
2	0.4	0.1	3

- And now we've completed the backward pass and calculated all the deltas (gradients) for the weights.
- Note that we need to keep track of the output values for each node as well as the derivative at each time step (this is why these are arrays in the code you're working with, one value per time step).
- However, you do not need to keep an array for the weight deltas, as you can just accumulate them in the backward pass.

Weight	Value	Delta
0	0.3	-0.107
1	-0.4	-0.628
2	-0.5	-0.058
3	0.2	-0.357
4	-0.1	0.096
5	-0.6	-0.375
6	0.1	-0.0044077
7	0.3	-0.0261789

Vanishing and Exploding Gradients

- While the Jordan network was small and short, with weights and inputs carefully chosen, we didn't get much in the way of vanishing/exploding gradients.
- On the other hand, in the Elman network we saw some big differences between gradients from the different time steps -- the gradient signals from the later time steps were much larger than the earlier time steps.
- That being said it is a significant problem to understand, which makes training RNNs the most challenging of all network types.
- To delve even deeper the following classic paper is a good read about the problems of training RNNs:

Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio. "On the difficulty of training recurrent neural networks." International conference on machine learning. 2013.

RNN Weight Initialization

RNN Weight Initialization

- A common older strategy for initializing RNNs is similar strategy as we discussed before.
- We can initialize the weights on incoming edges to a node with a *uniform* (not normal) distribution [-1,1], and then scaling the random numbers by $1/\sqrt{f_{in}}$ where f_{in} is the fan-in to that node.

$$w_i = \frac{U[-1,1]}{\sqrt{f_{in}}}$$

- For this method biases are commonly set to 0.

Xavier Initialization

- However back in 2010 Xavier and Bengio found that this heuristic does not work too well for networks with a large number of layers. The activation gradients vanish to almost zero and it prevents networks from learning.
- They instead propose a different method which they call *normalized initialization* which works much better in practice, utilizing both the *fan in* (f_{in}) and the *fan out* (f_{out}) of a layer:

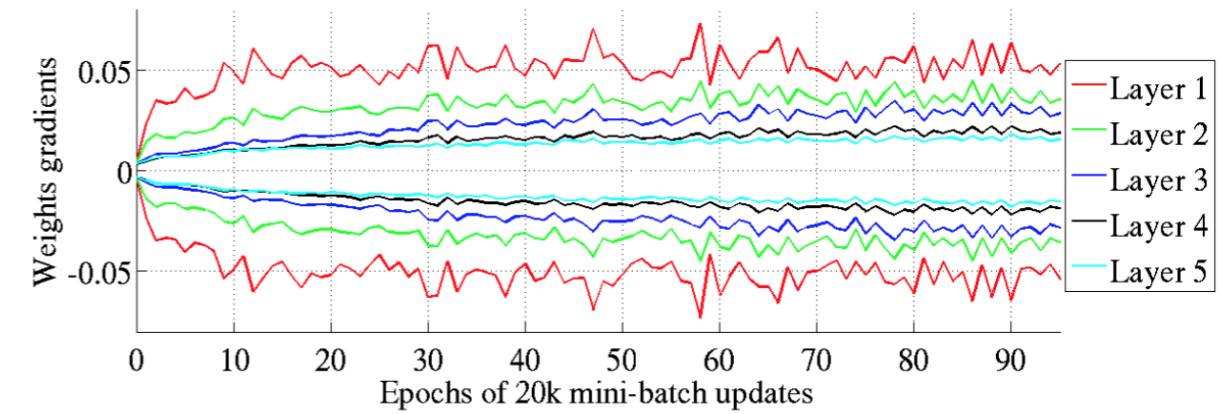
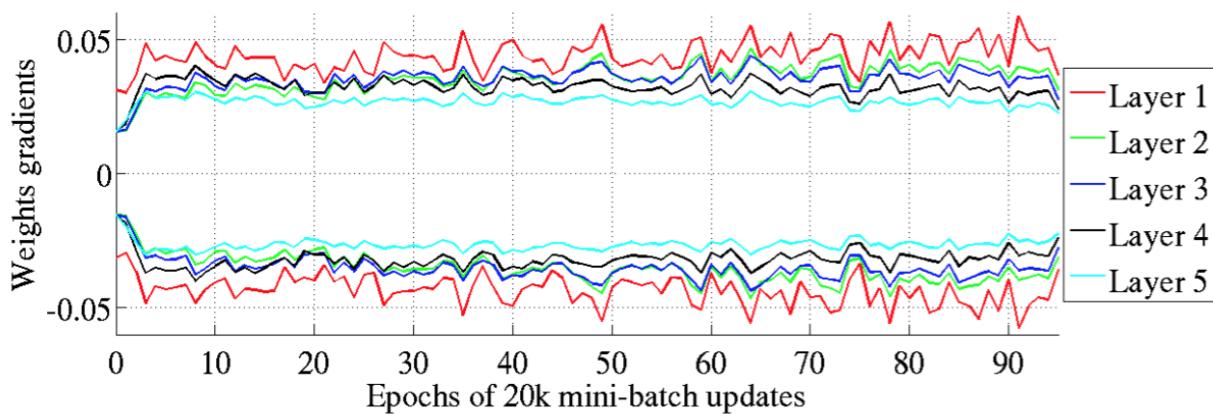
$$w_{j+1,i} = U\left[-\frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}\right]$$

- Where n_j is the number of weights fanning in to layer j , and n_{j+1} is the number of weights fanning out of layer j .

[1] Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." Proceedings of the thirteenth international conference on artificial intelligence and statistics. 2010.

Xavier Initialization

- Xavier and Bengio found that this method of initialization would maintain the variance of activations and gradients across many layers (they tested up to 5) whereas the previous method had differing variances for different layers, see figures from their work:



[1] Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." Proceedings of the thirteenth international conference on artificial intelligence and statistics. 2010.

Kaiming Initialization

- Unfortunately the work on Xavier weight initialization was done using symmetric activation functions (sigmoid, tanh and softsign):

$$\text{softsign}(x) = \frac{x}{1+|x|}$$

- However nowadays many activation functions are non-symmetric (e.g., ReLUs), so the results may not hold.

Kaiming Initialization

- In 2015, He. et al. [2] showed that deep networks (they used a 22 layer CNN) would converge much earlier if they utilized a different strategy:
 - Initialize bias to 0.

$$w_i = N(0, 1) * \frac{\sqrt{2}}{\sqrt{f_{in}}}$$

- Numbers generated from a normal distribution (mean = 0, std deviation = 1) are multiplied by $\sqrt{2}/\sqrt{f_{in}}$ for the weights on a given layer.

[2] He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." Proceedings of the IEEE international conference on computer vision. 2015.

Which to use?

- For RNNs it may be worth trying the different above strategies depending on your activation functions.
- If they are symmetric then Xavier may outperform Kaiming initialization. If you're regression (using L1 or L2 loss functions) you can even try the old standard strategy.
- Different data sets and architectures will perform differently so its up to you to investigate which weight initialization strategies give you the best results.

Gradient Clipping, Scaling and Gradient Boosting

Gradient Clipping and Scaling

- You'll find that when training RNNs backprop will perform fine and then suddenly gradient values will become NaN (not a number) or Inf (infinity).
- From the suggested text:

"The difficulty that arises is that when the parameter gradient is very large, a gradient descent parameter update could throw the parameters very far, into a region where the objective function is larger, undoing much of the work that had been done to reach the current solution."

-- Deep Learning, Pg 413

Gradient Clipping and Scaling

- This will even happen with the different memory cell based architectures (which we'll discuss next week). As noted by Alex Graves [2]:

"One difficulty when training LSTM with the full gradient is that the derivatives sometimes become excessively large, leading to numerical problems. To prevent this, [we] clipped the derivative of the loss with respect to the network inputs to the LSTM layers (before the sigmoid and tanh functions are applied) to lie within a predefined range." [2]

[2] Graves, Alex. "Generating sequences with recurrent neural networks." arXiv preprint arXiv:1308.0850 (2013). <https://arxiv.org/pdf/1308.0850>

Gradient Clipping and Scaling

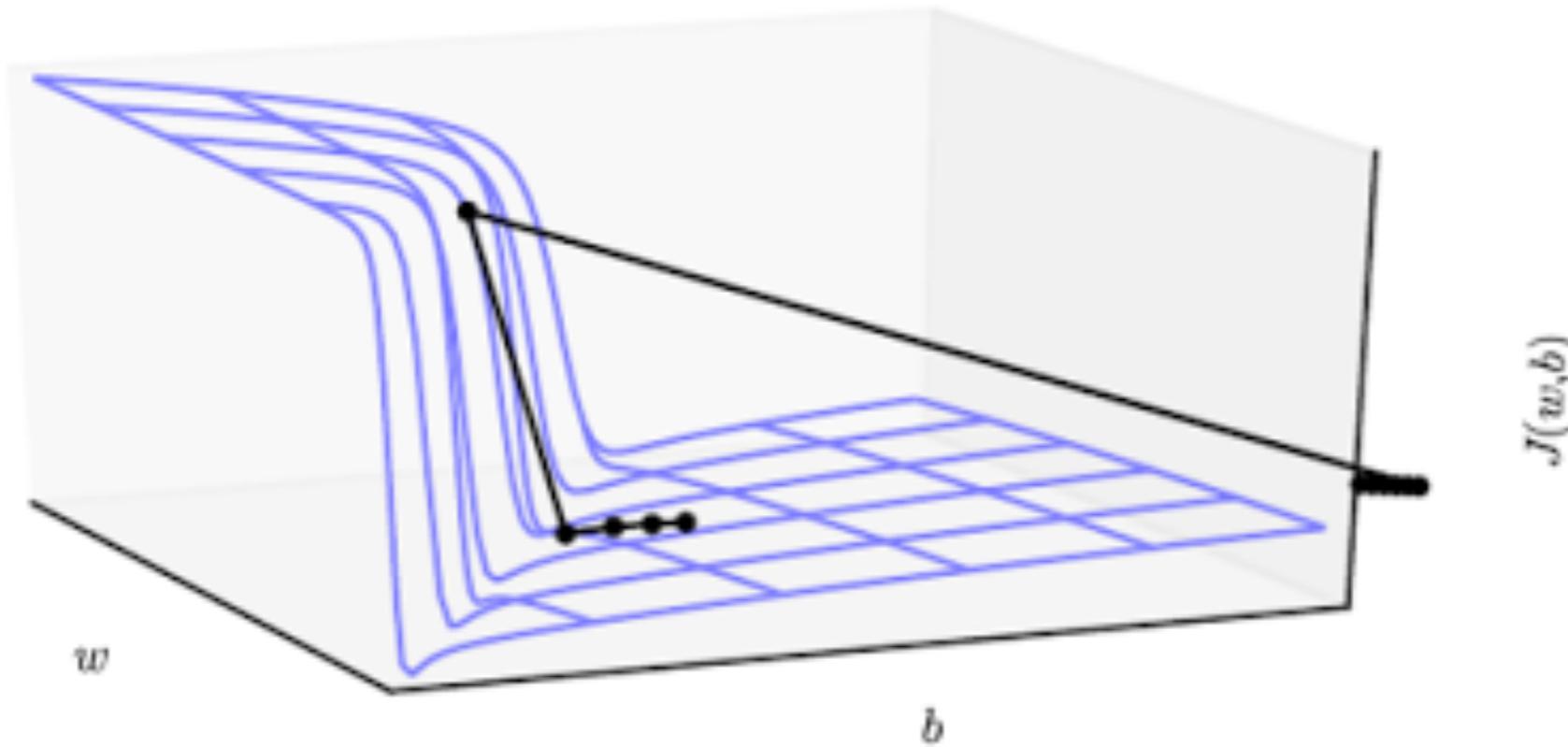


Image from Hinton's Coursera lecture videos.

- For a little intuition, the above image represents the loss of a NN based on the weights and biases (in three dimensions).
- With RNNs in particular, this is a common scenario. A dip in the search space right next to a large wall can lead to the move being made by gradient descent landing on the wall (which results in a huge gradient) or to the top of it with a poor fitness.
- Greedily moving in the direction of steepest descent and moving too far ends up having a very bad result.

Gradient Clipping and Scaling

- Gradient clipping and scaling (used for gradients that are too large) aren't always a necessary solution, sometimes these numerical issues happen because (and you should check for these first):
 - A learning rate that is too high which leads to very large weight updates.
 - Failing to properly normalize data, which can lead to large differences in predicted outputs vs expected outputs.
 - Poor choices of activation functions (identity and ReLU in particular) which allow for unbounded outputs from nodes.

Weight Capping

- Another method (not yet gradient clipping or scaling) which can help is to cap weights:
- After the weight update -- if any weight is < -threshold set the weight = -threshold (t), or if the weight is > threshold (t), set the weight = threshold:

$$w_i = \begin{cases} -s * t, & \text{if } x < -t \\ s * t, & \text{if } x > t \\ w_i, & \text{otherwise} \end{cases}$$

- Sometimes setting it to $0.9 * \text{threshold}$ (or some other scale, s) can improve because it gives the weight the ability to move a bit after being capped.

Gradient Clipping and Scaling

- The first thing you should do is make sure the other potential problems aren't the cause of your exploding gradients and numerical issues. For many networks these may be sufficient.
- However if not you should try gradient clipping or scaling (personally I've found gradient scaling to work better).

Gradient Clipping

- Pascanu et al. found that you should use a different clipping threshold for the output layers as compared to the hidden layers.
- Gradient clipping involves clipping the output and hidden node derivatives in a manner similar to weight clipping.
- Pascanu et al. suggest clipping output derivatives in the range $[-100,100]$, and hidden derivatives in the range $[-10,10]$.

Gradient Scaling

- A smarter method involves using the L2 norm to rescale the gradients.
- If the L2 norm of all the weight updates (gradients/deltas) exceeds a threshold, t , scale all the gradients downwards:

$$g_i = g_i * \frac{t}{L2(g)}, \text{ if } L2(g) > t$$

- Note that this is the L2 norm of the entire gradient (delta) vector g .
- I've found a threshold of 1.0 to work well in practice and this is what Pascanu et al. suggest.

Gradient Boosting

- Hochreiter and Schmidhuber (authors of the seminal LSTM paper) note the opposite problem (and use it as motivation for LSTMs as a potential solution):

"Learning to store information over extended period of time intervals via recurrent backpropagation takes a very long time, mostly due to insufficient, decaying error back flow." [3]

[3] Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." Neural computation 9.8 (1997): 1735-1780.

Gradient Boosting

- For RNNs, vanishing gradients, where they all become 0 or close to it, due to flat valleys/plateaus in the search space are also a significant problem.
- I've found in practice this can significantly speed up training RNNs but haven't seen it in literature yet.
- Perform the same operation, except if the threshold is below a certain value (I use 0.05):

$$g_i = g_i * \frac{t}{L2(g)}, \text{ if } L2(g) < t$$

- Note that in gradient scaling, $L2(g) > t$ which means the weights will all be decreased. For gradient boosting, $L2(g) < t$ so all the weights will be increased.

Dealing with Numeric Issues

- So in summary, if you're seeing numeric issues when training neural networks make sure you check and try the following:
 - Is your learning rate too high?
 - Is your data normalized properly?
 - Are you using the right loss functions?
 - Try clipping/scaling:
 - Weight clipping
 - Gradient clipping
 - Gradient Scaling
- If your RNNs are reaching areas where they stop learning due to vanished gradients, try gradient boosting.

Lecture 7

RNN Memory Cells

DSCI-789: Neural Networks for Data Science

Travis Desell (tjdvse@rit.edu)

Associate Professor

Department of Software Engineering



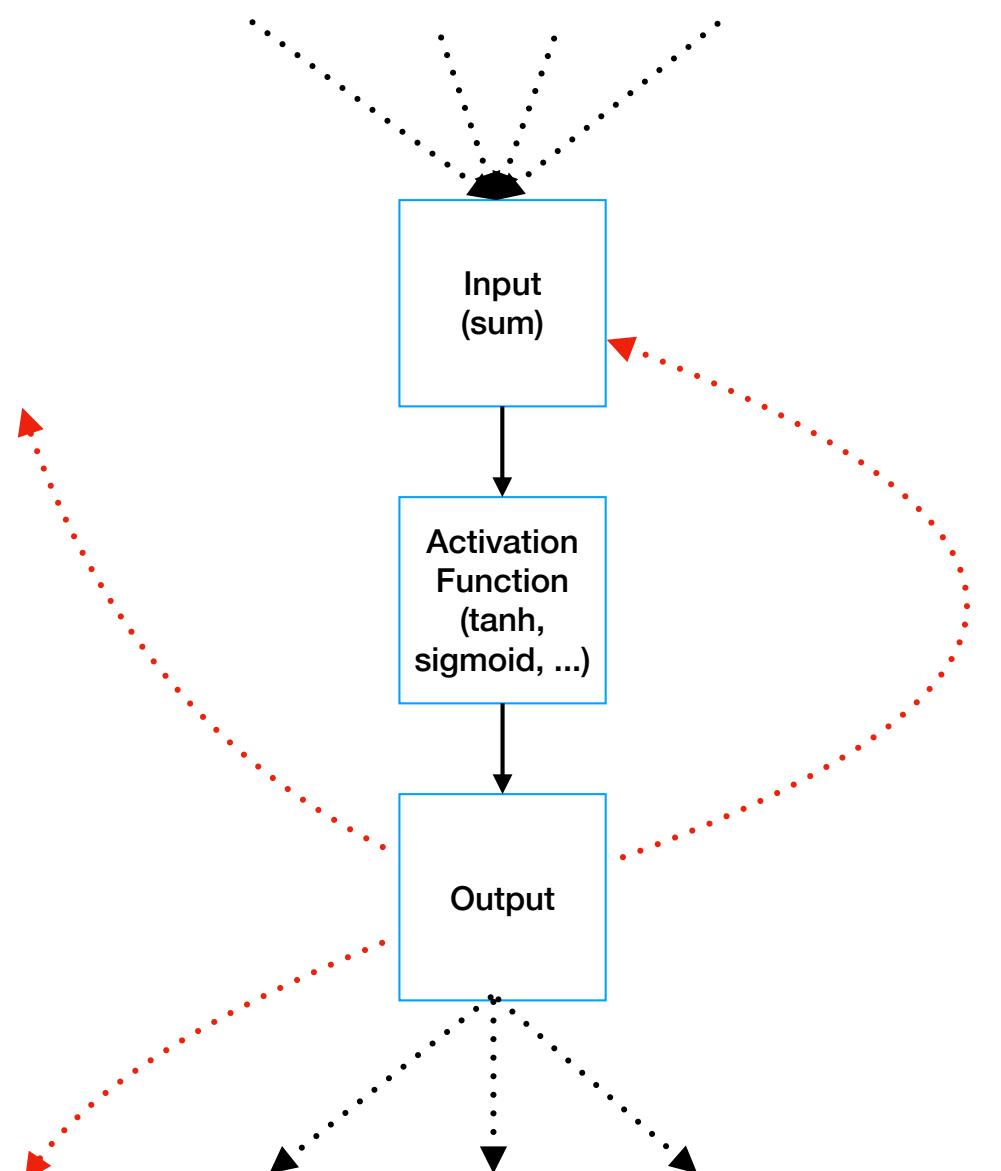
ROCHESTER INSTITUTE OF TECHNOLOGY

Overview

- RNN Memory Cells:
 - LSTM
 - GRU
 - MGU
 - UGRNN
 - Delta-RNN

RNN Memory Cells

Memory Cell Structures - Simple Neuron



- Up until now, all the architectures we've worked with have only used "simple" neurons.
- Simple neurons are the basic neural network building block. Inputs are summed, and activation function is applied and the output is feed forward to other neurons.
- As we've seen, simple (and any other memory cell structure) can also have recurrent edges (in red). These can loop back to the same neuron (e.g., an Elman-like connection) or to any other neuron in the network (shallower, same layer or deeper).

Long Short-Term Memory (LSTM) Cells

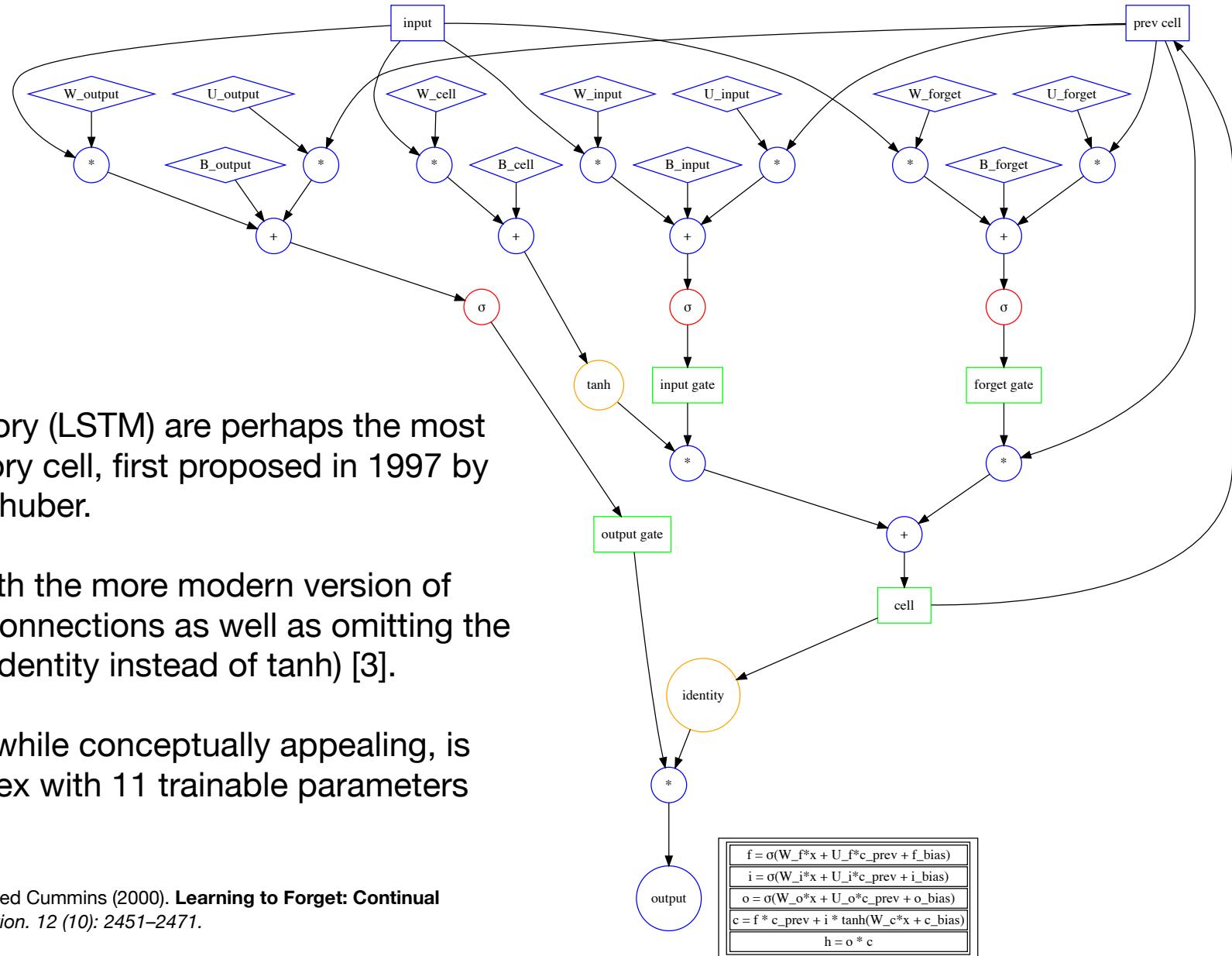
Memory Cell Structures - LSTM

Long Short-Term Memory (LSTM) are perhaps the most well known RNN memory cell, first proposed in 1997 by Hochreiter and Schmidhuber.

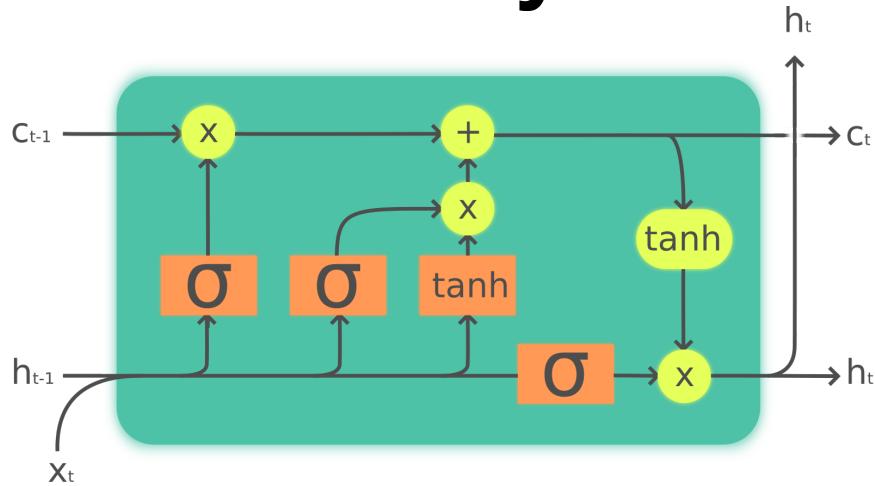
We're going to work with the more modern version of LSTM, with peephole connections as well as omitting the output function (using identity instead of tanh) [3].

This cellular structure, while conceptually appealing, is computationally complex with 11 trainable parameters (blue diamonds).

[1] Felix A. Gers; Jürgen Schmidhuber; Fred Cummins (2000). **Learning to Forget: Continual Prediction with LSTM**. *Neural Computation*. 12 (10): 2451–2471.



Memory Cell Structures - LSTM



Legend:



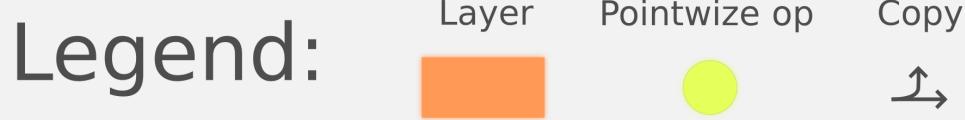
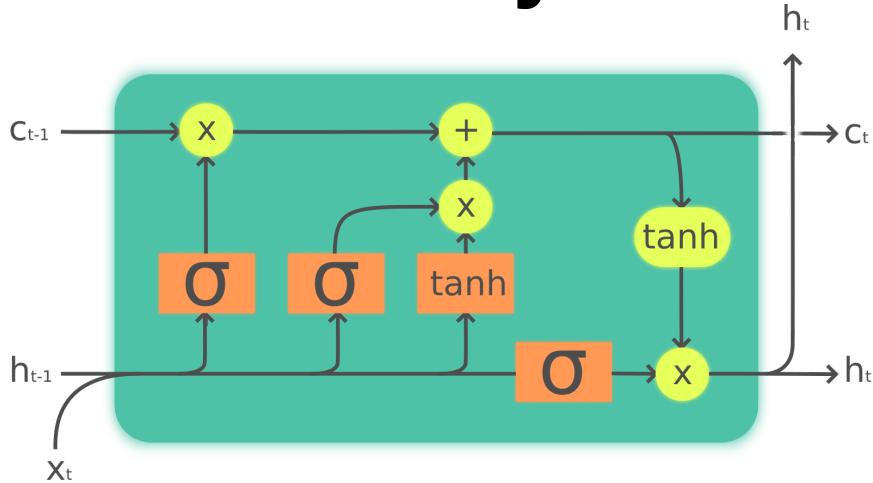
$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$
$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$
$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$
$$\tilde{c}_t = \sigma_h(W_c x_t + U_c h_{t-1} + b_c)$$
$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$$
$$h_t = o_t * \sigma_h(c_t)$$

You'll commonly see diagrams (like above) which unfortunately obscure the details a bit, but we'll get into the lower level details for the above and relate them to how we've been performing backward passes through RNNs.

Top right gives the traditional equation for a the traditional LSTM cell. These are in many times presented as vector operations however we'll stick with individual cells.

Images from: https://en.wikipedia.org/wiki/Long_short-term_memory

Memory Cell Structures - LSTM



The various variables used are:

- f_t : forget gate value at time t
- i_t : input gate value at time t
- o_t : output gate value at time t
- c_t : cell value at time t
- x_t : input value at time t
- h_t : hidden value at time t (i.e., the output of the LSTM cell)
- σ_g : gate activation function (sigmoid)
- σ_c : cell activation function (tanh)
- σ_h : output activation function (identity)

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$

$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$

$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$

$$\tilde{c}_t = \sigma_h(W_c x_t + U_c h_{t-1} + b_c)$$

$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$$

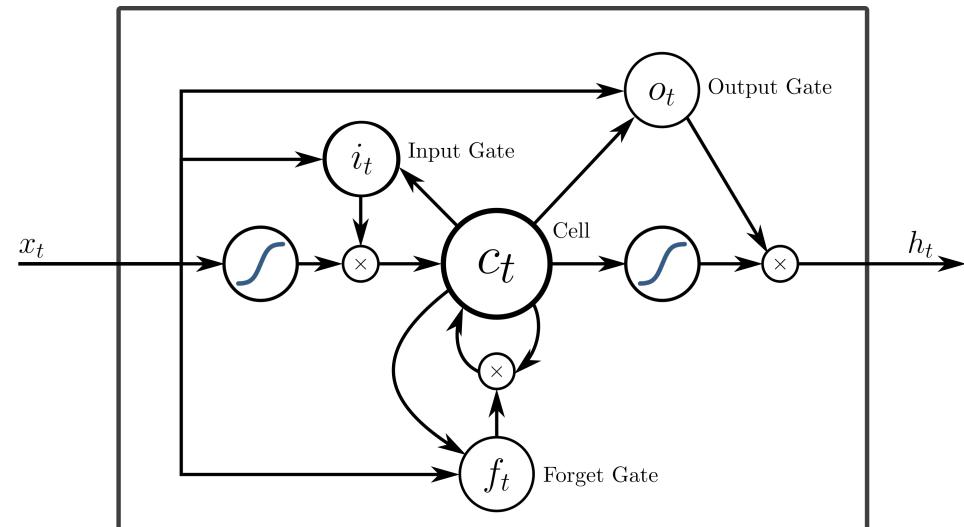
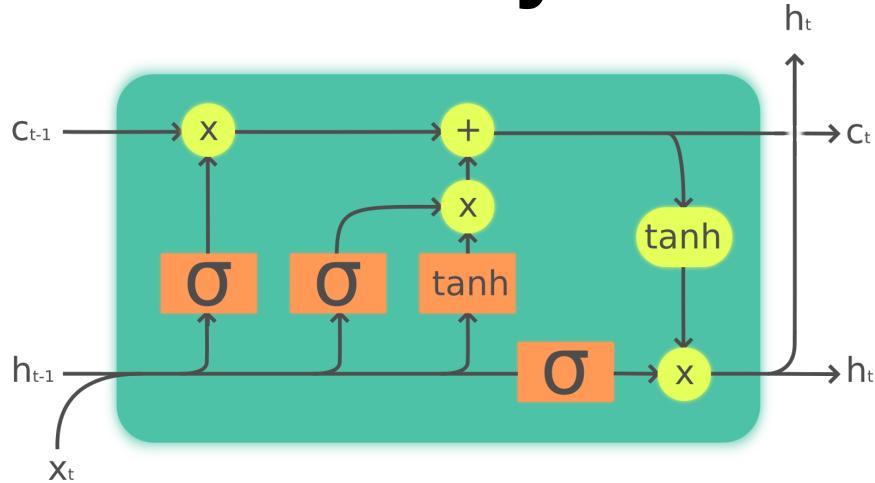
$$h_t = o_t * \sigma_h(c_t)$$

The variables in bold are all trainable parameters. If we were using vector notation there would be one value per cell. Each LSTM cell has 11 trainable parameters.

- **W_f** : forget gate weight(s)
- **W_i** : input gate weight(s)
- **W_o** : output gate weight(s)
- **W_c** : cell weight(s)
- **U_f** : forget gate update weight(s)
- **U_i** : input gate update weight(s)
- **U_o** : output gate update weight(s)
- **b_f** : forget gate bias
- **b_i** : input gate bias
- **b_o** : output gate bias
- **b_c** : cell bias

Images from: https://en.wikipedia.org/wiki/Long_short-term_memory

Memory Cell Structures - LSTM



Legend:

Layer	Pointwise op	Copy

Peephole LSTM [1,2]

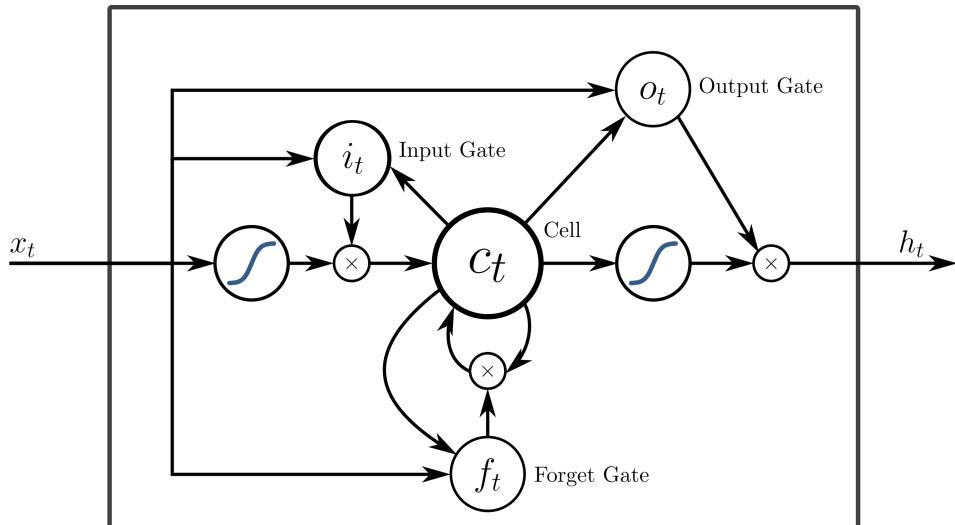
There have been a number of variants on the LSTM over the years, and consensus is that the peephole LSTM [1,2] (top right) generally performs best, so we'll be using that version. It also simplifies the LSTM cell a bit.

[1] Felix A. Gers, Jürgen Schmidhuber (2001). **LSTM Recurrent Networks Learn Simple Context Free and Context Sensitive Languages**. *IEEE Transactions on Neural Networks*. 12 (6): 1333–1340. doi:10.1109/72.963769.

[2] Felix A. Gers, Nicol N. Schraudolph, Jürgen Schmidhuber (2002). **Learning precise timing with LSTM recurrent networks**. *Journal of Machine Learning Research*. 3: 115–143.

Images from: https://en.wikipedia.org/wiki/Long_short-term_memory

Memory Cell Structures - Peephole LSTM



Peephole LSTM [1,2]

$$f_t = \sigma_g(W_f x_t + U_f c_{t-1} + b_f)$$
$$i_t = \sigma_g(W_i x_t + U_i c_{t-1} + b_i)$$
$$o_t = \sigma_g(W_o x_t + U_o c_{t-1} + b_o)$$
$$c_t = f_t * c_{t-1} + i_t * \sigma_c(W_c x_t + b_c)$$
$$h_t = \sigma_h(o_t * c_t)$$

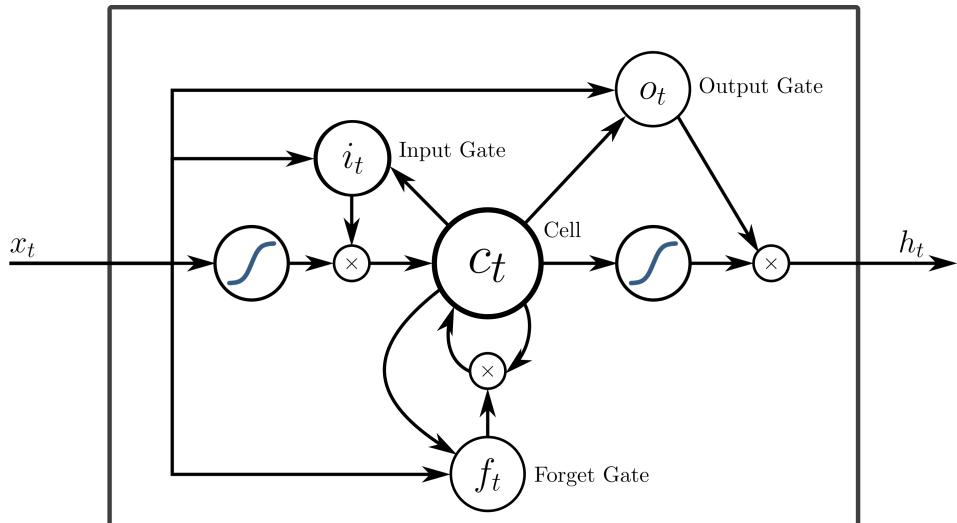
The top right provides the functions for how the output of a peephole LSTM cell is calculated. These are in many times presented as vector operations however we'll stick with individual cells.

[1] Felix A. Gers, Jürgen Schmidhuber (2001). **LSTM Recurrent Networks Learn Simple Context Free and Context Sensitive Languages**. *IEEE Transactions on Neural Networks*. 12 (6): 1333–1340. doi:10.1109/72.963769.

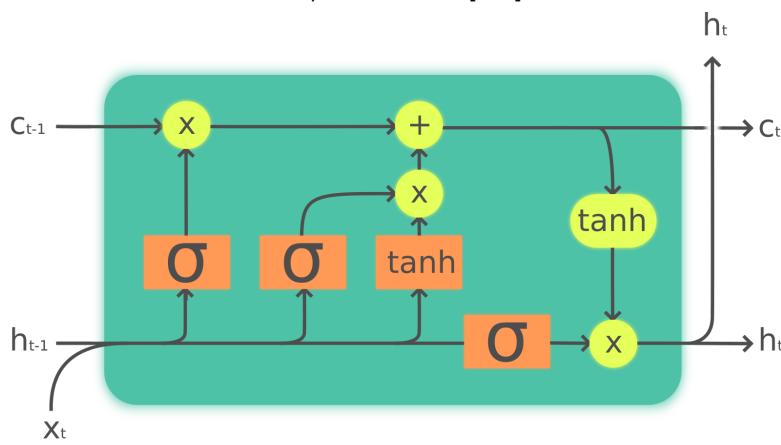
[2] Felix A. Gers, Nicol N. Schraudolph, Jürgen Schmidhuber (2002). **Learning precise timing with LSTM recurrent networks**. *Journal of Machine Learning Research*. 3: 115–143.

Images from: https://en.wikipedia.org/wiki/Long_short-term_memory

Memory Cell Structures - LSTM



Peephole LSTM [1,2]



Legend:



$$f_t = \sigma_g(W_f x_t + U_f c_{t-1} + b_f)$$

$$i_t = \sigma_g(W_i x_t + U_i c_{t-1} + b_i)$$

$$o_t = \sigma_g(W_o x_t + U_o c_{t-1} + b_o)$$

$$c_t = f_t * c_{t-1} + i_t * \sigma_c(W_c x_t + b_c)$$

$$h_t = \sigma_h(o_t * c_t)$$

Note how the peephole LSTM (top) simplifies things. Instead of calculating an intermediate cell state (\tilde{c}_t), and utilizing the previous output of the LSTM, it instead simply uses the previous cell value in calculating the values of the input, forget, and output gates.

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$

$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$

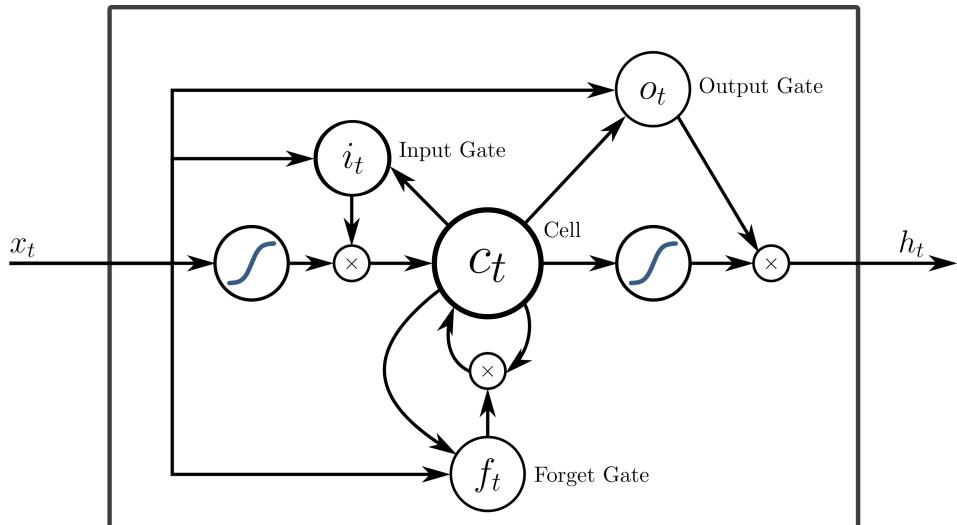
$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$

$$\tilde{c}_t = \sigma_h(W_c x_t + U_c h_{t-1} + b_c)$$

$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$$

$$h_t = o_t * \sigma_h(c_t)$$

Memory Cell Structures - Peephole LSTM



Peephole LSTM [1,2]

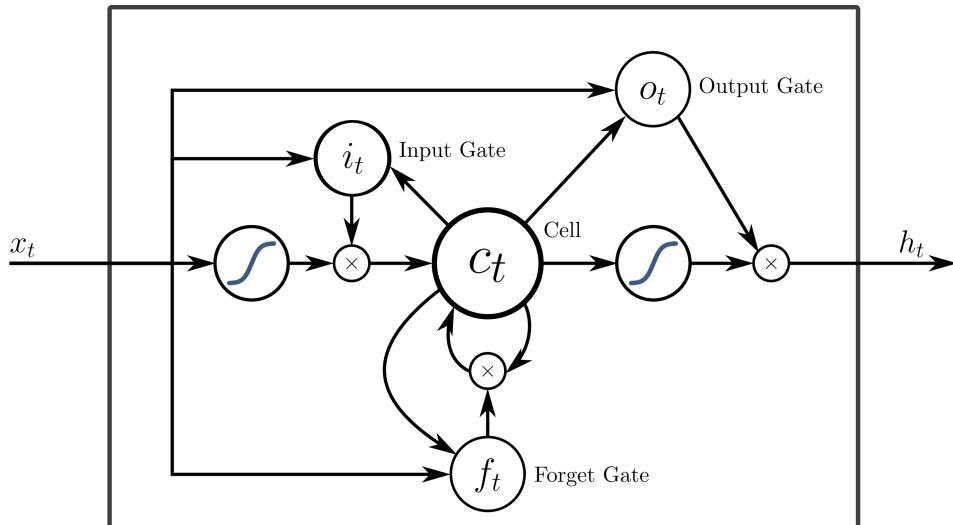
So one more time, the various variables used are:

- f_t : forget gate value at time t
- i_t : input gate value at time t
- o_t : output gate value at time t
- c_t : cell value at time t
- x_t : input value at time t
- h_t : hidden value at time t (i.e., the output of the LSTM cell)
- σ_g : gate activation function (sigmoid)
- σ_c : cell activation function (tanh)
- σ_h : output activation function (identity)

$$f_t = \sigma_g(W_f x_t + U_f c_{t-1} + b_f)$$
$$i_t = \sigma_g(W_i x_t + U_i c_{t-1} + b_i)$$
$$o_t = \sigma_g(W_o x_t + U_o c_{t-1} + b_o)$$
$$c_t = f_t * c_{t-1} + i_t * \sigma_c(W_c x_t + b_c)$$
$$h_t = \sigma_h(o_t * c_t)$$

- **W_f : forget gate weight(s)**
- **W_i : input gate weight(s)**
- **W_o : output gate weight(s)**
- **W_c : cell weight(s)**
- **U_f : forget gate update weight(s)**
- **U_i : input gate update weight(s)**
- **U_o : output gate update weight(s)**
- **b_f : forget gate bias**
- **b_i : input gate bias**
- **b_o : output gate bias**
- **b_c : cell bias**

Memory Cell Structures - Peephole LSTM



Peephole LSTM [1,2]

What is novel about the memory cells over the non-cellular RNNs we've been using is the "gated" connections:

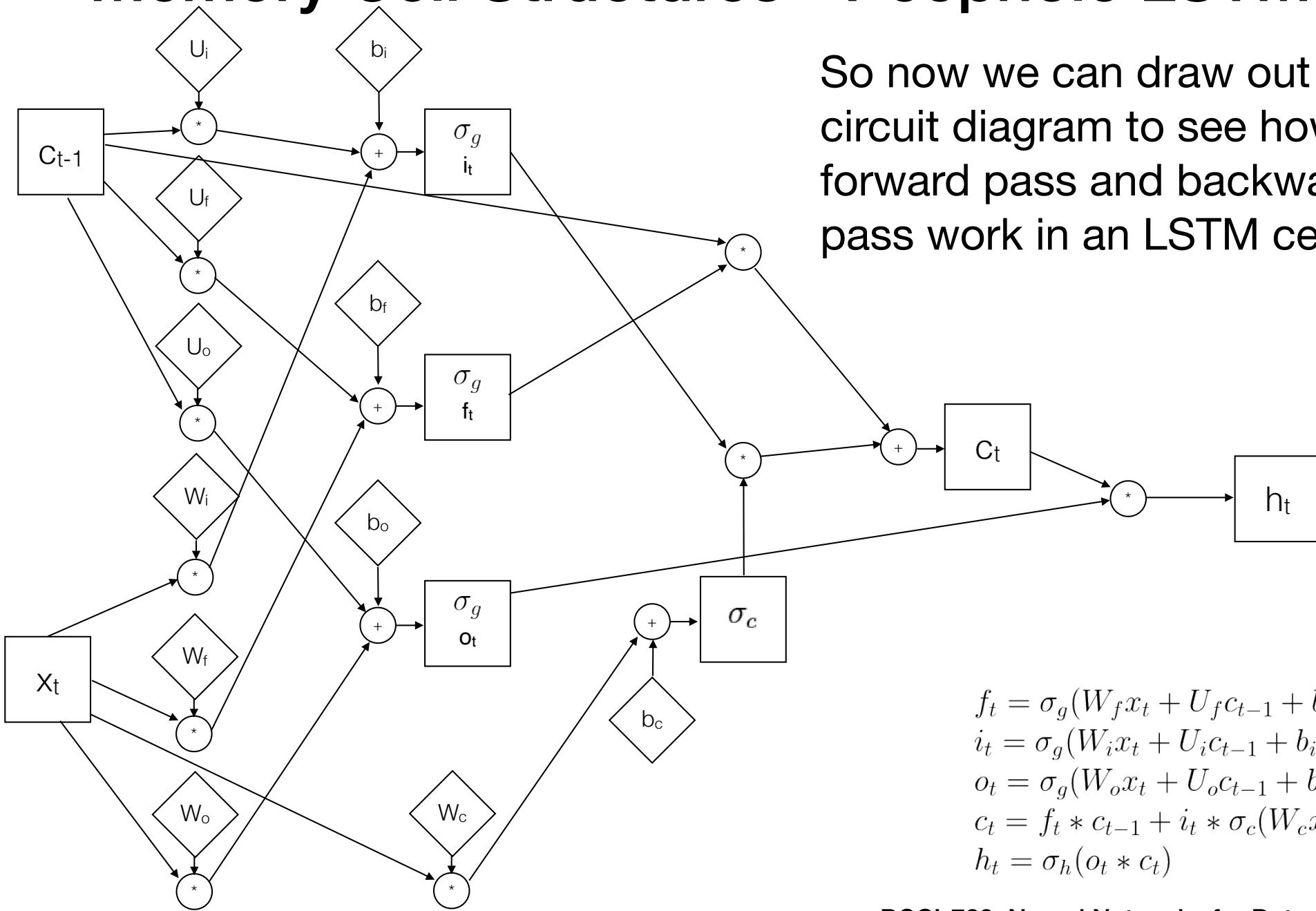
- The value going into the cell is *multiplied* by the output of the input gate, which allows it to control how much information flows into the cell's memory.
- The value flowing out of the cell is *multiplied* by the output of the output gate, again, allowing the output gate to control how much information flows out of the cell's memory.
- Lastly, how much information is passed from one time step to the next is handled by the forget gate, which is multiplied by the previous time step's cell value.

Prior to this, outputs of nodes were only added together at a new node before having the activation function be applied. Using multiplication by the output values allows to control data flow through the cell, not only through the forward pass but also through time.

Also, luckily we already know how to handle a multiply in our backward pass!

$$f_t = \sigma_g(W_f x_t + U_f c_{t-1} + b_f)$$
$$i_t = \sigma_g(W_i x_t + U_i c_{t-1} + b_i)$$
$$o_t = \sigma_g(W_o x_t + U_o c_{t-1} + b_o)$$
$$c_t = f_t * c_{t-1} + i_t * \sigma_c(W_c x_t + b_c)$$
$$h_t = \sigma_h(o_t * c_t)$$

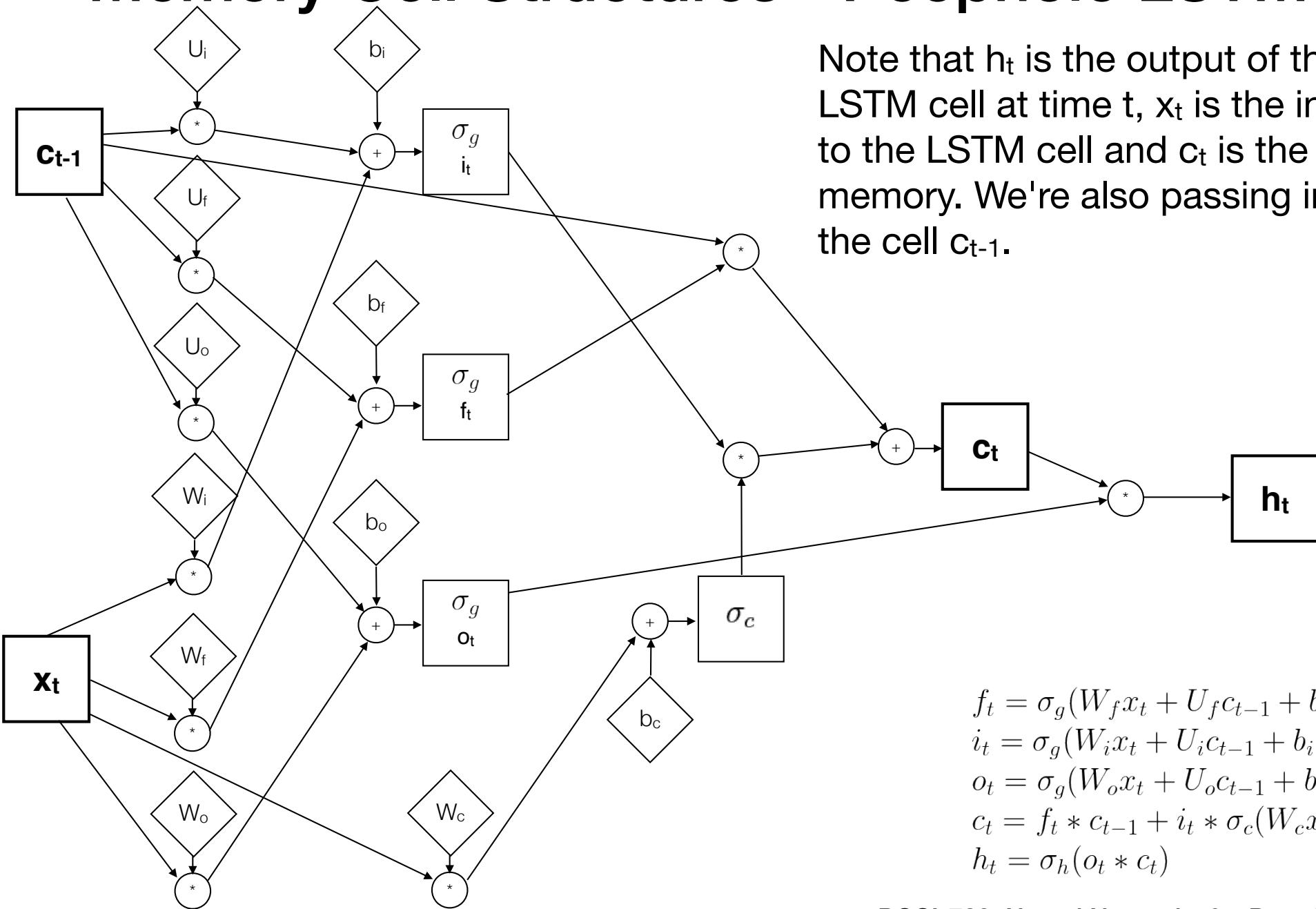
Memory Cell Structures - Peephole LSTM



So now we can draw out our circuit diagram to see how the forward pass and backward pass work in an LSTM cell.

$$\begin{aligned} f_t &= \sigma_g(W_f x_t + U_f c_{t-1} + b_f) \\ i_t &= \sigma_g(W_i x_t + U_i c_{t-1} + b_i) \\ o_t &= \sigma_g(W_o x_t + U_o c_{t-1} + b_o) \\ c_t &= f_t * c_{t-1} + i_t * \sigma_c(W_c x_t + b_c) \\ h_t &= \sigma_h(o_t * c_t) \end{aligned}$$

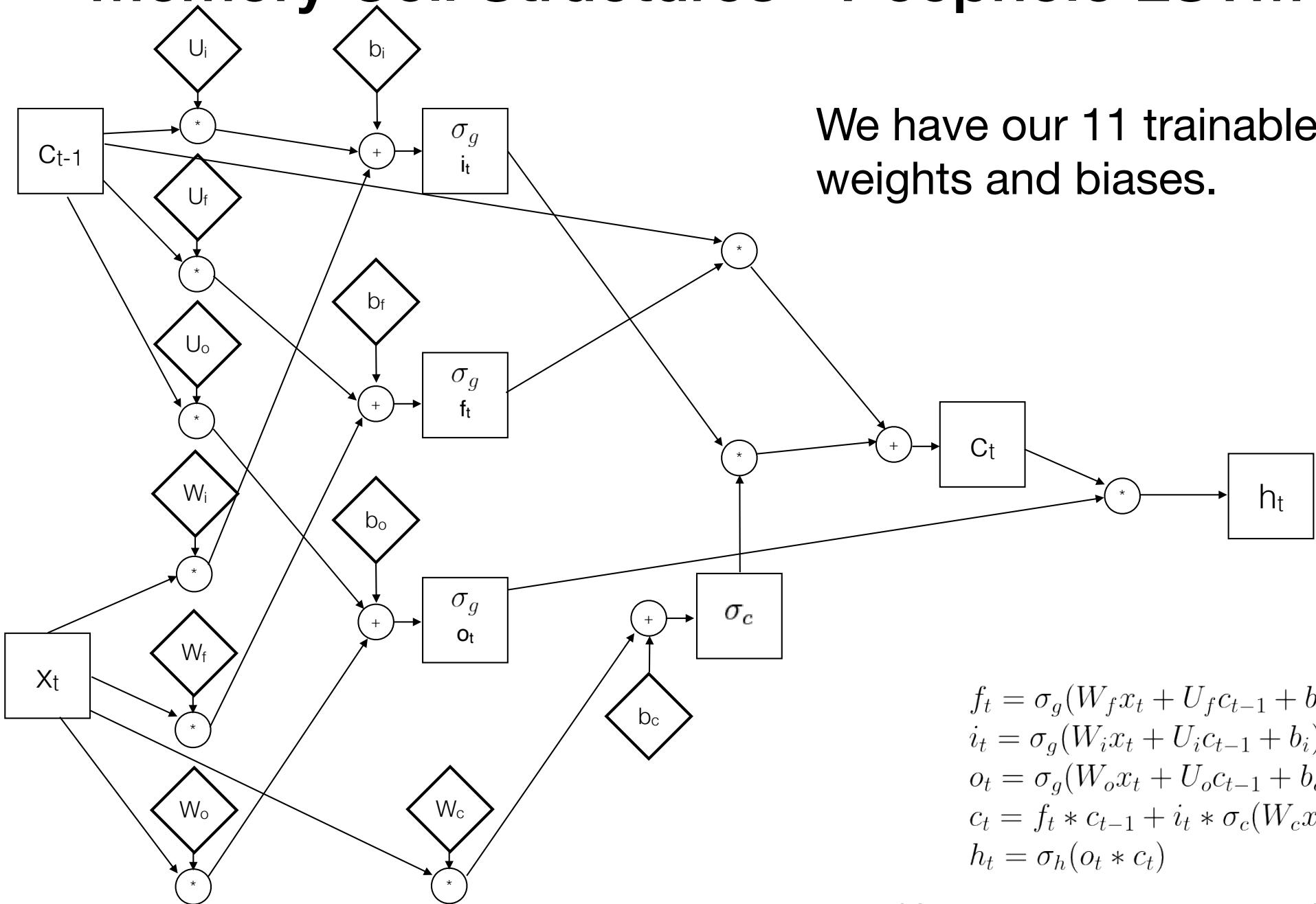
Memory Cell Structures - Peephole LSTM



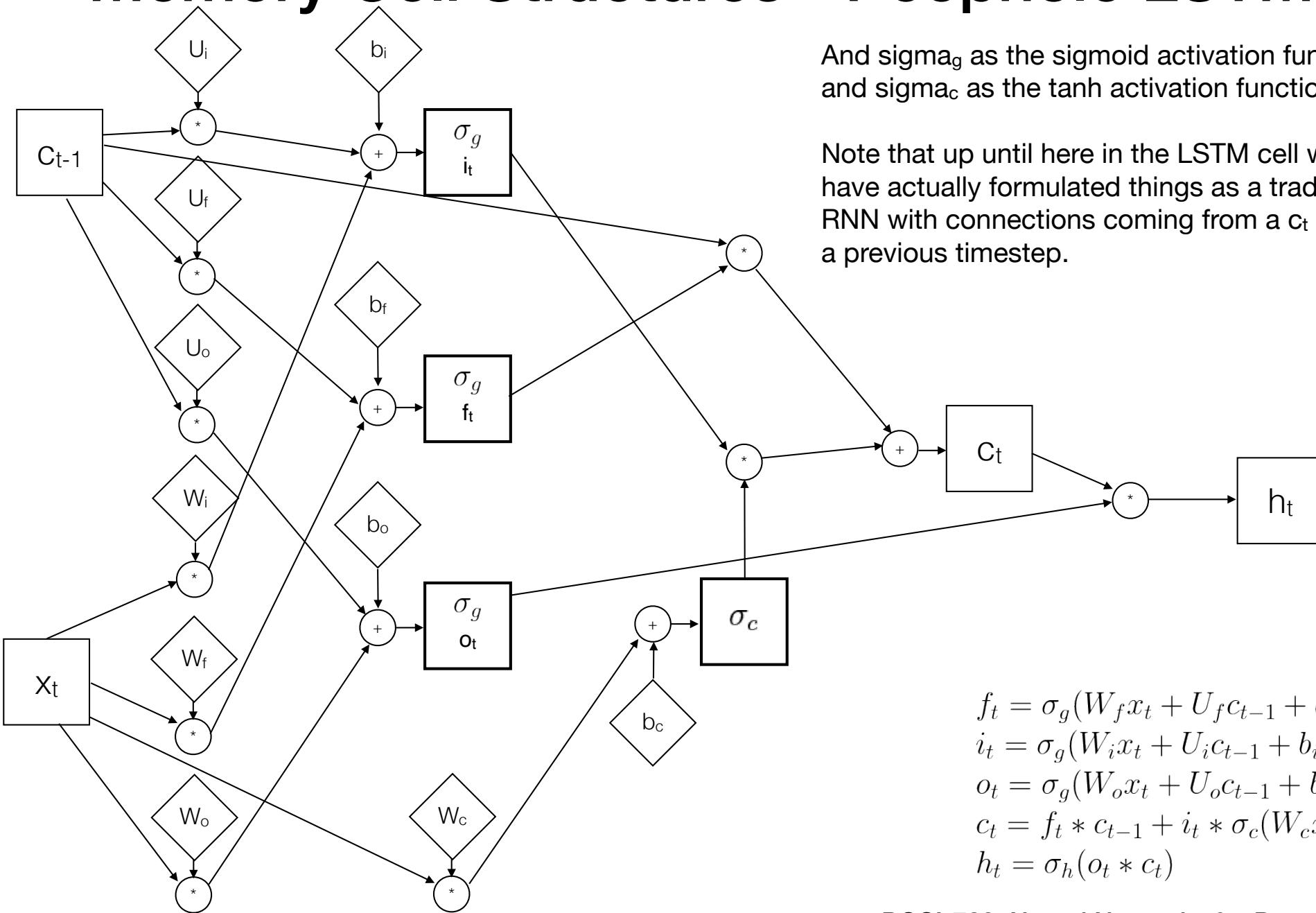
Note that h_t is the output of the LSTM cell at time t , x_t is the input to the LSTM cell and c_t is the cell's memory. We're also passing into the cell c_{t-1} .

$$\begin{aligned}f_t &= \sigma_g(W_f x_t + U_f c_{t-1} + b_f) \\i_t &= \sigma_g(W_i x_t + U_i c_{t-1} + b_i) \\o_t &= \sigma_g(W_o x_t + U_o c_{t-1} + b_o) \\c_t &= f_t * c_{t-1} + i_t * \sigma_c(W_c x_t + b_c) \\h_t &= \sigma_h(o_t * c_t)\end{aligned}$$

Memory Cell Structures - Peephole LSTM



Memory Cell Structures - Peephole LSTM

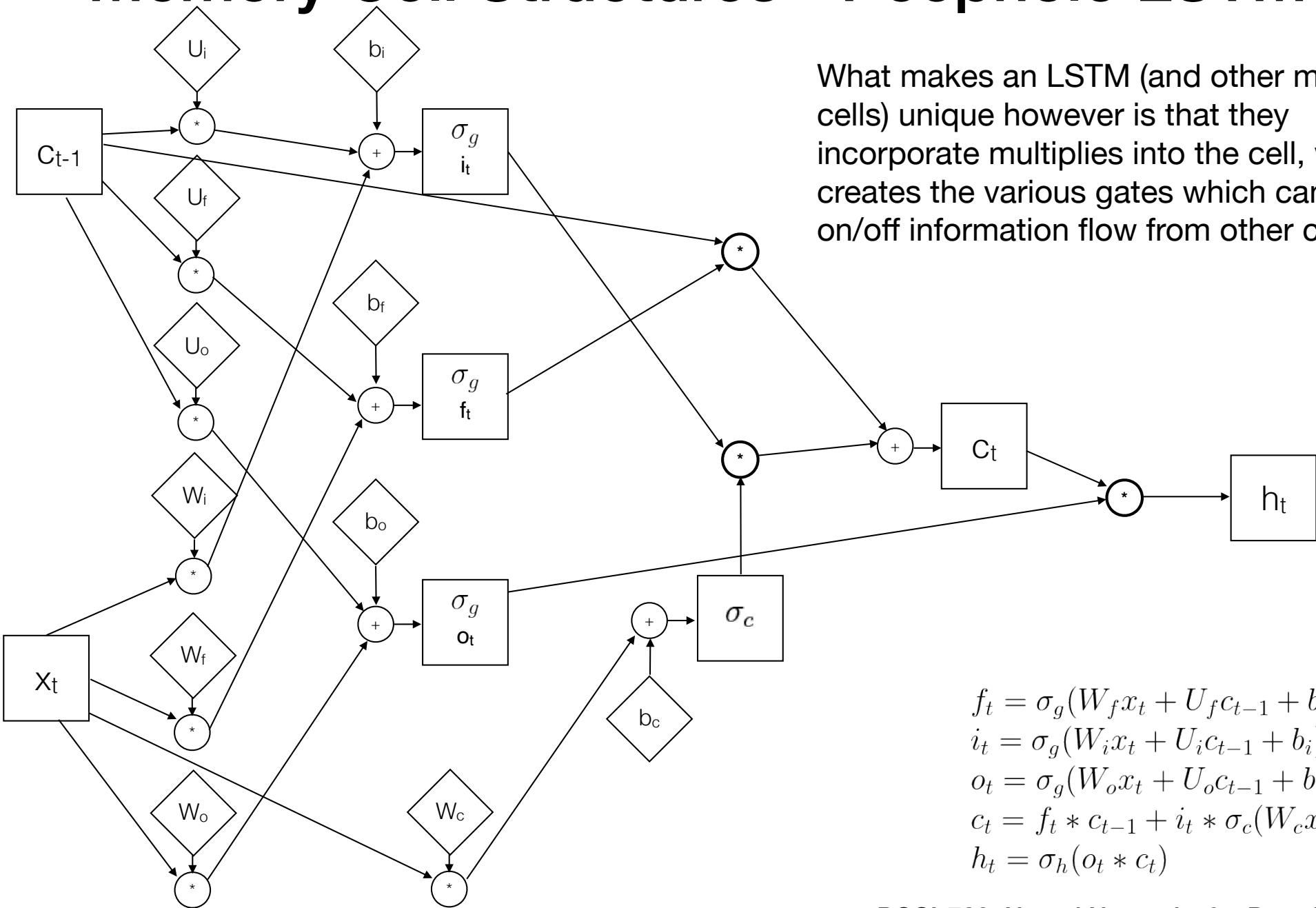


And σ_g as the sigmoid activation functions and σ_c as the tanh activation function.

Note that up until here in the LSTM cell we could have actually formulated things as a traditional RNN with connections coming from a c_t node at a previous timestep.

$$\begin{aligned}
 f_t &= \sigma_g(W_f x_t + U_f c_{t-1} + b_f) \\
 i_t &= \sigma_g(W_i x_t + U_i c_{t-1} + b_i) \\
 o_t &= \sigma_g(W_o x_t + U_o c_{t-1} + b_o) \\
 c_t &= f_t * c_{t-1} + i_t * \sigma_c(W_c x_t + b_c) \\
 h_t &= \sigma_h(o_t * c_t)
 \end{aligned}$$

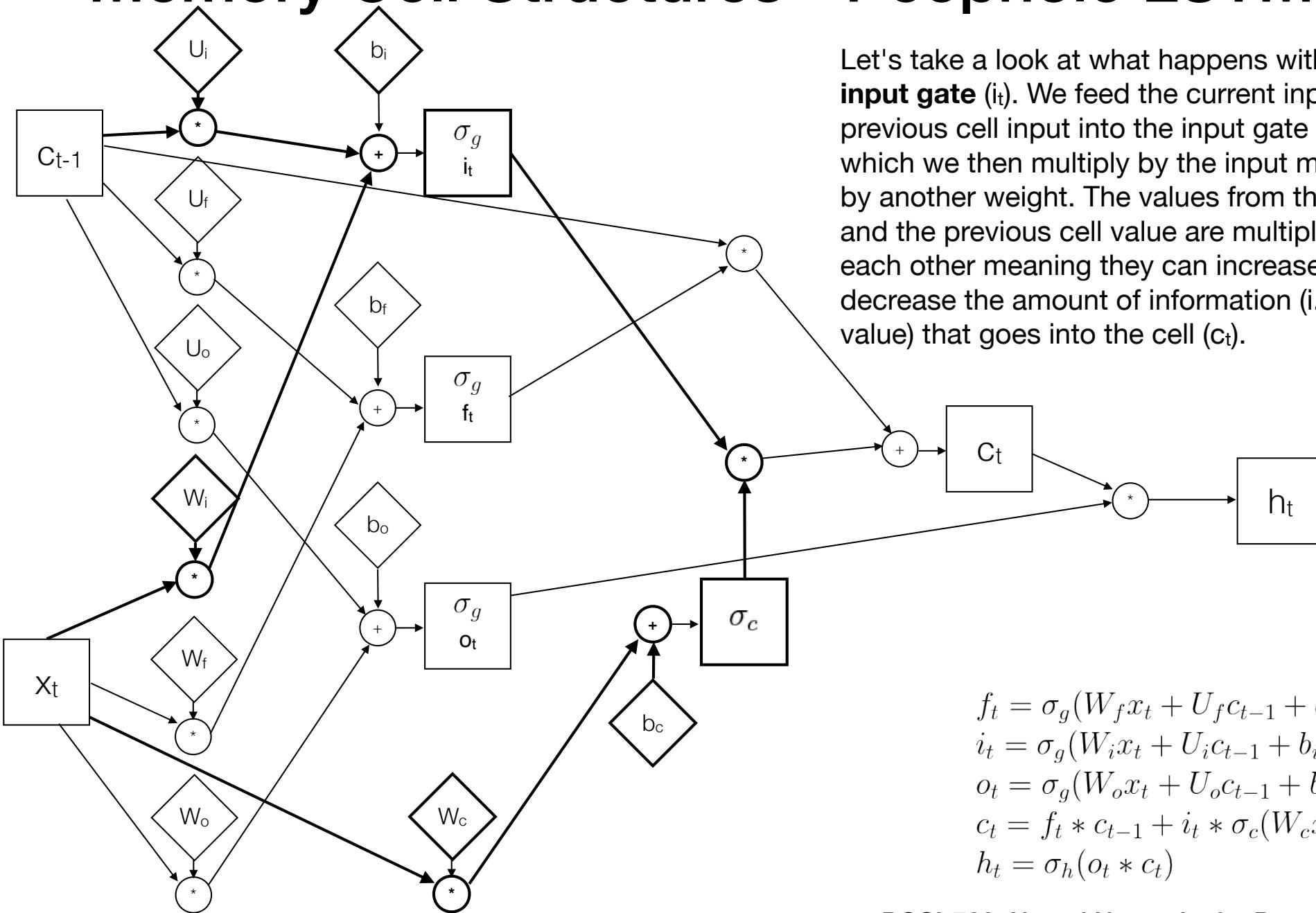
Memory Cell Structures - Peephole LSTM



What makes an LSTM (and other memory cells) unique however is that they incorporate multipliers into the cell, which creates the various gates which can turn on/off information flow from other cells.

$$\begin{aligned}f_t &= \sigma_g(W_f x_t + U_f C_{t-1} + b_f) \\i_t &= \sigma_g(W_i x_t + U_i C_{t-1} + b_i) \\o_t &= \sigma_g(W_o x_t + U_o C_{t-1} + b_o) \\c_t &= f_t * c_{t-1} + i_t * \sigma_c(W_c x_t + b_c) \\h_t &= \sigma_h(o_t * c_t)\end{aligned}$$

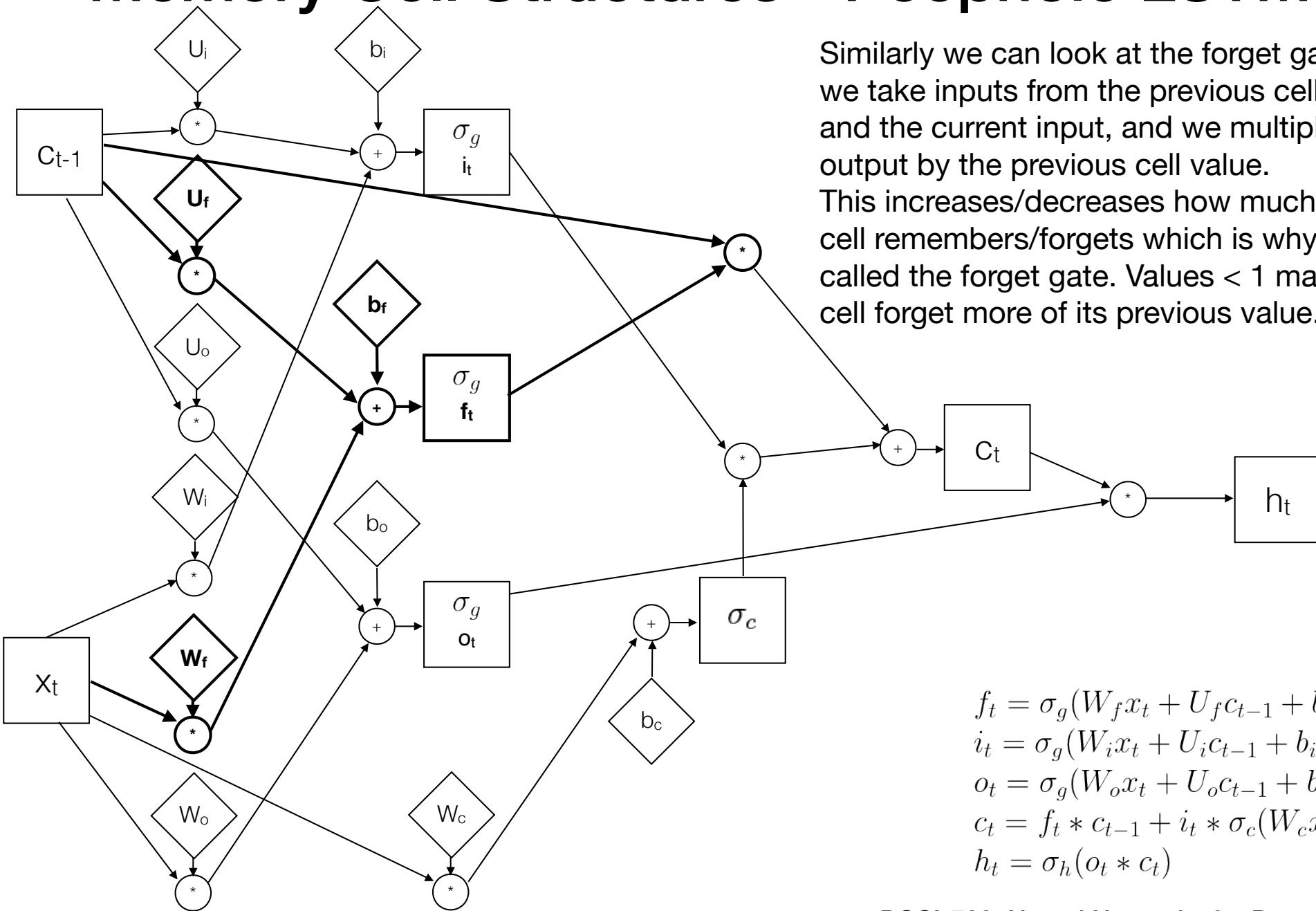
Memory Cell Structures - Peephole LSTM



Let's take a look at what happens with the **input gate** (i_t). We feed the current input, the previous cell input into the input gate (i_t) which we then multiply by the input multiplied by another weight. The values from the input and the previous cell value are multiplied by each other meaning they can increase or decrease the amount of information (i.e., the value) that goes into the cell (c_t).

$$\begin{aligned}
 f_t &= \sigma_g(W_fx_t + U_fc_{t-1} + b_f) \\
 i_t &= \sigma_g(W_ix_t + U_ic_{t-1} + b_i) \\
 o_t &= \sigma_g(W_ox_t + U_oc_{t-1} + b_o) \\
 c_t &= f_t * c_{t-1} + i_t * \sigma_c(W_cx_t + b_c) \\
 h_t &= \sigma_h(o_t * c_t)
 \end{aligned}$$

Memory Cell Structures - Peephole LSTM

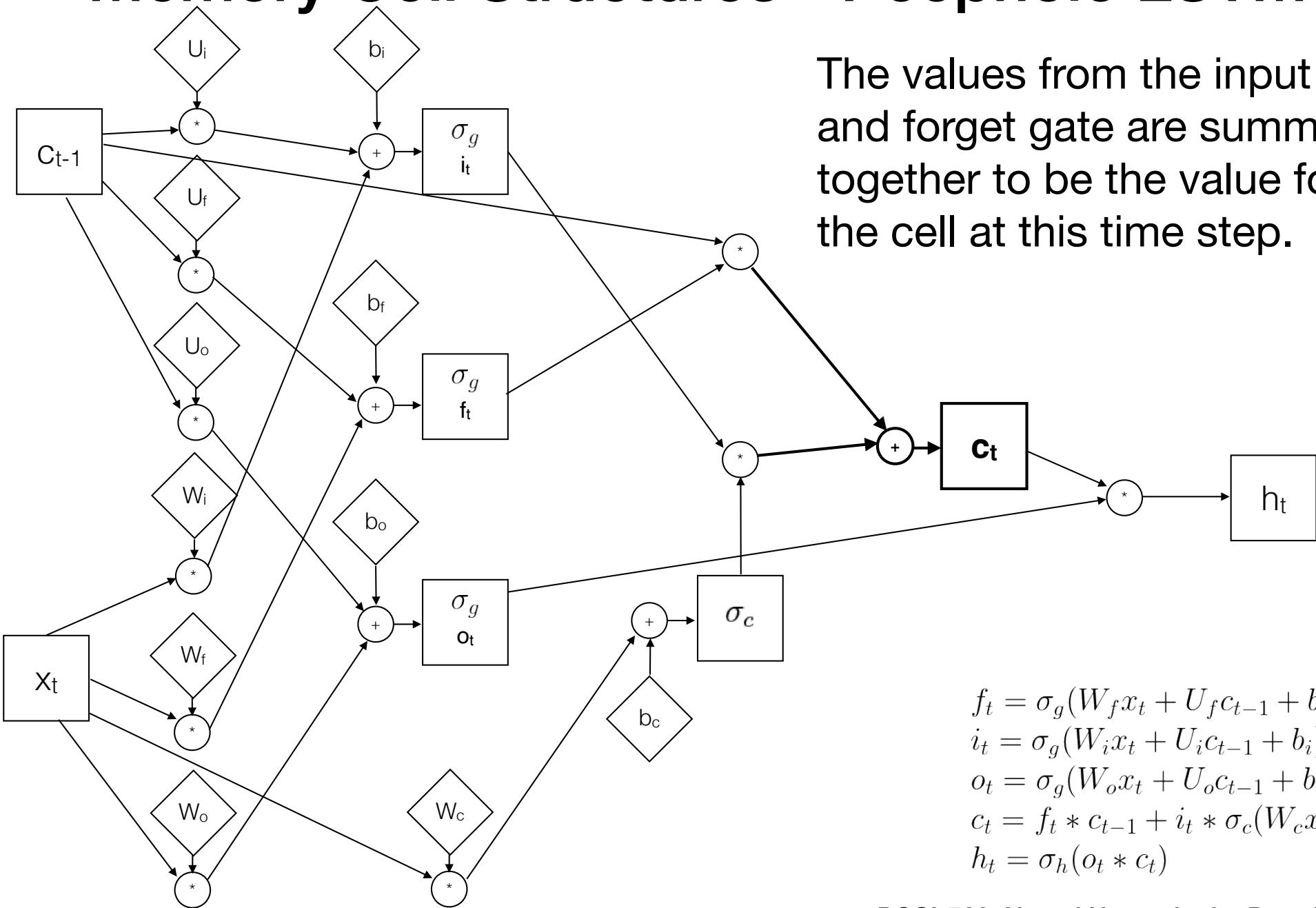


Similarly we can look at the forget gate (f_t), we take inputs from the previous cell value and the current input, and we multiply this output by the previous cell value.

This increases/decreases how much the cell remembers/forgets which is why it's called the forget gate. Values < 1 make the cell forget more of its previous value.

$$\begin{aligned}
 f_t &= \sigma_g(W_f x_t + U_f c_{t-1} + b_f) \\
 i_t &= \sigma_g(W_i x_t + U_i c_{t-1} + b_i) \\
 o_t &= \sigma_g(W_o x_t + U_o c_{t-1} + b_o) \\
 c_t &= f_t * c_{t-1} + i_t * \sigma_c(W_c x_t + b_c) \\
 h_t &= \sigma_h(o_t * c_t)
 \end{aligned}$$

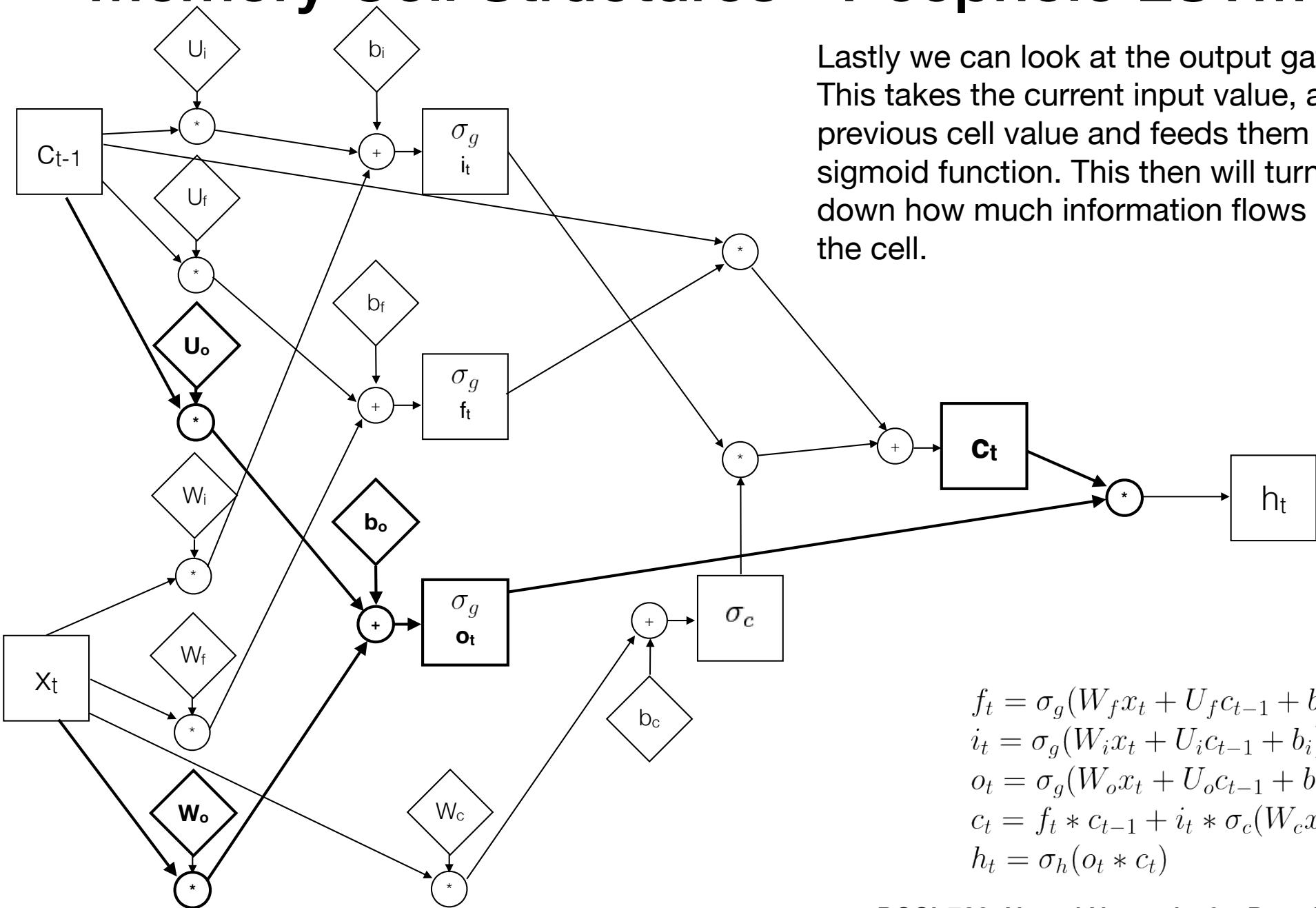
Memory Cell Structures - Peephole LSTM



The values from the input gate and forget gate are summed together to be the value for the cell at this time step.

$$\begin{aligned} f_t &= \sigma_g(W_f x_t + U_f c_{t-1} + b_f) \\ i_t &= \sigma_g(W_i x_t + U_i c_{t-1} + b_i) \\ o_t &= \sigma_g(W_o x_t + U_o c_{t-1} + b_o) \\ c_t &= f_t * c_{t-1} + i_t * \sigma_c(W_c x_t + b_c) \\ h_t &= \sigma_h(o_t * c_t) \end{aligned}$$

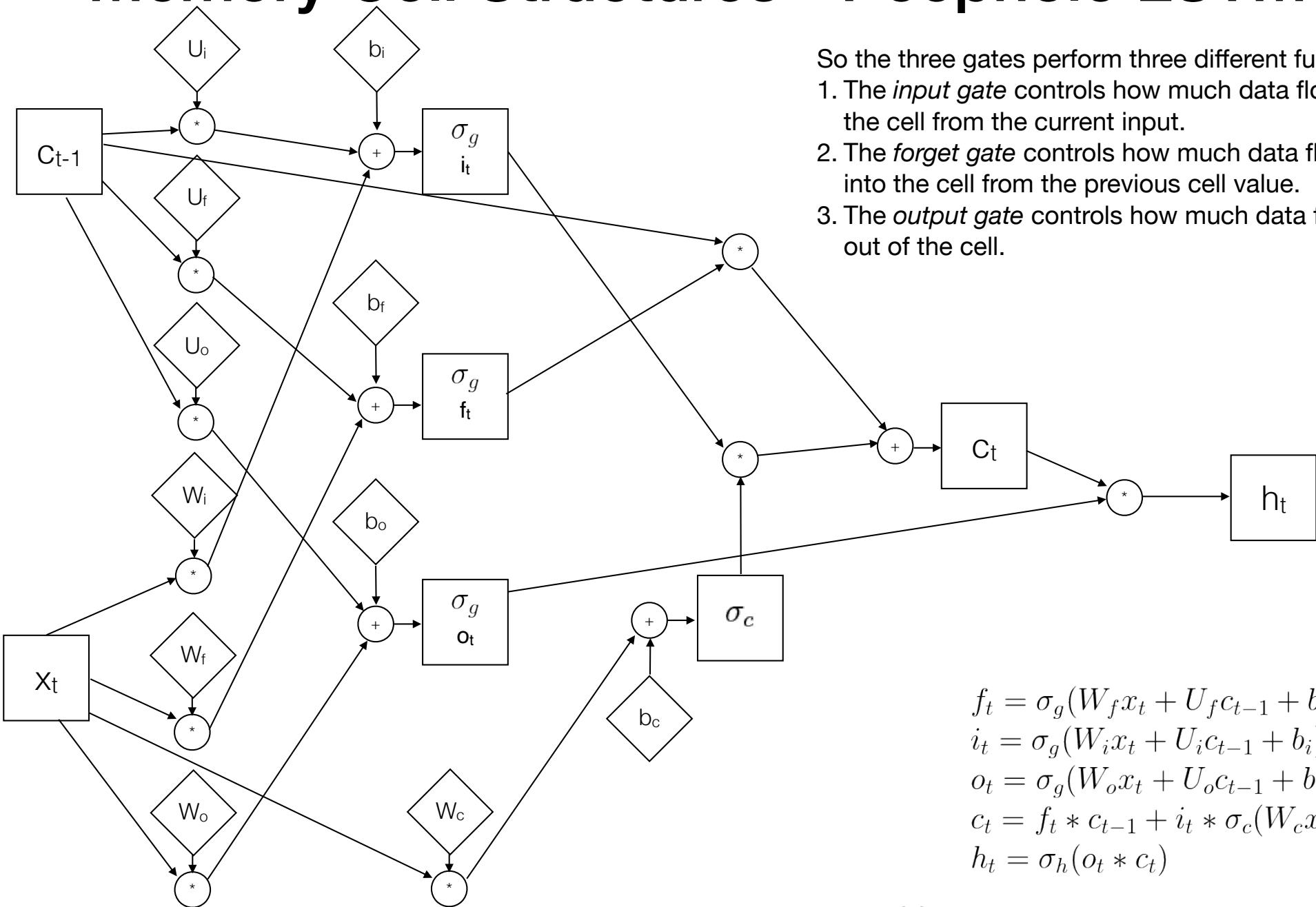
Memory Cell Structures - Peephole LSTM



Lastly we can look at the output gate (o_t). This takes the current input value, and previous cell value and feeds them into a sigmoid function. This then will turn up/ down how much information flows out of the cell.

$$\begin{aligned}
 f_t &= \sigma_g(W_f x_t + U_f c_{t-1} + b_f) \\
 i_t &= \sigma_g(W_i x_t + U_i c_{t-1} + b_i) \\
 o_t &= \sigma_g(W_o x_t + U_o c_{t-1} + b_o) \\
 c_t &= f_t * c_{t-1} + i_t * \sigma_c(W_c x_t + b_c) \\
 h_t &= \sigma_h(o_t * c_t)
 \end{aligned}$$

Memory Cell Structures - Peephole LSTM

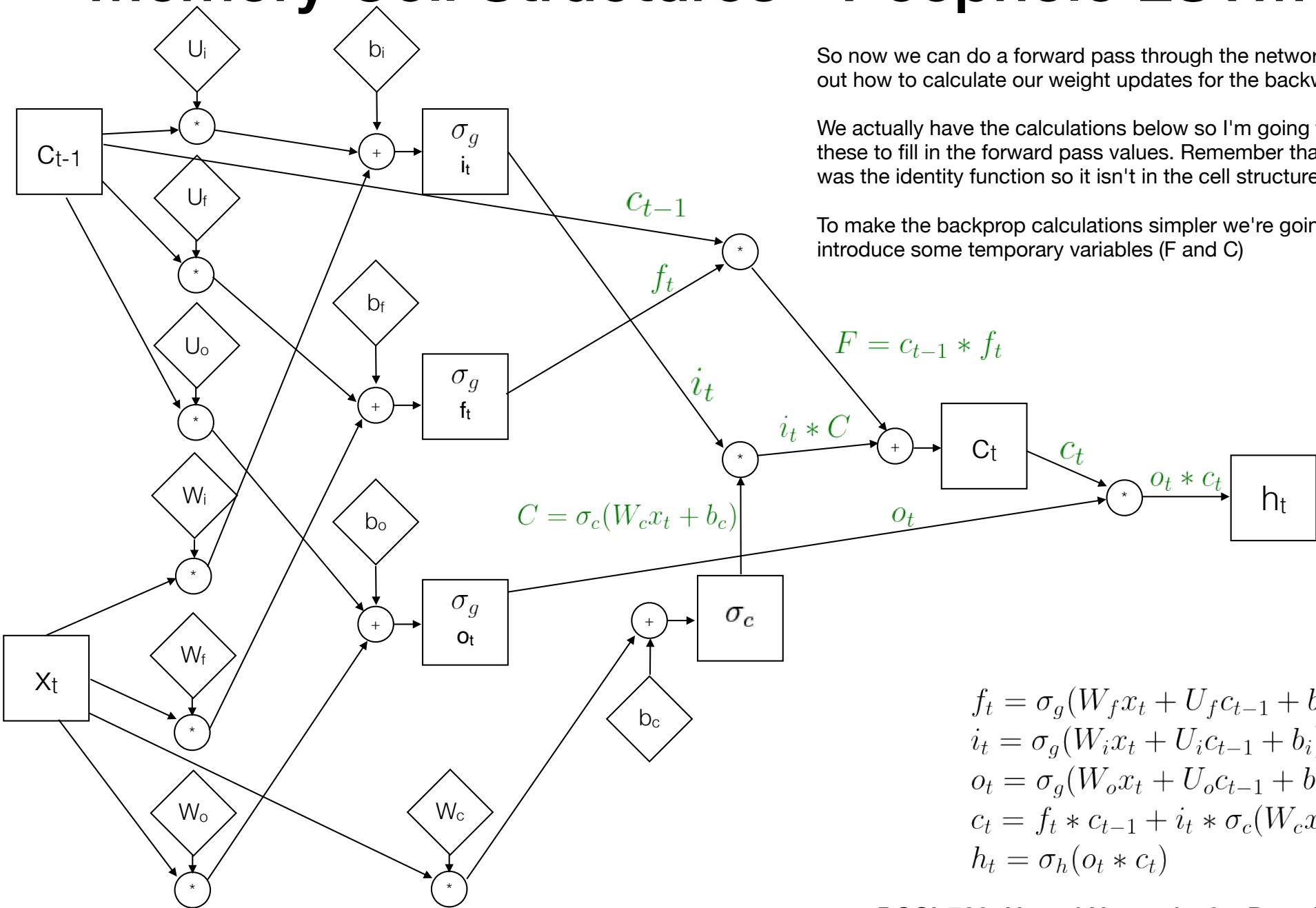


So the three gates perform three different functions:

1. The *input gate* controls how much data flows into the cell from the current input.
2. The *forget gate* controls how much data flows into the cell from the previous cell value.
3. The *output gate* controls how much data flows out of the cell.

$$\begin{aligned}
 f_t &= \sigma_g(W_f x_t + U_f c_{t-1} + b_f) \\
 i_t &= \sigma_g(W_i x_t + U_i c_{t-1} + b_i) \\
 o_t &= \sigma_g(W_o x_t + U_o c_{t-1} + b_o) \\
 c_t &= f_t * c_{t-1} + i_t * \sigma_c(W_c x_t + b_c) \\
 h_t &= \sigma_h(o_t * c_t)
 \end{aligned}$$

Memory Cell Structures - Peephole LSTM

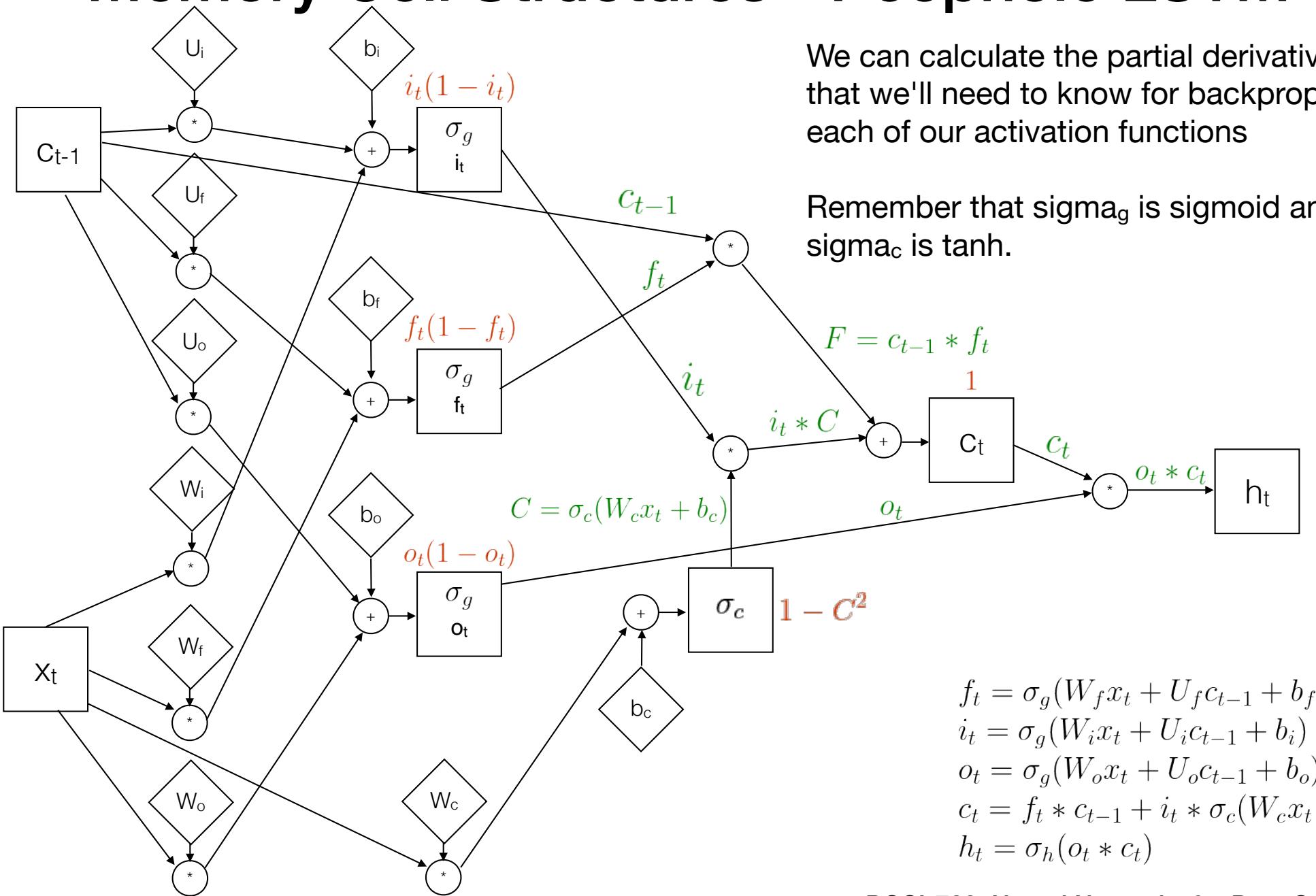


So now we can do a forward pass through the network to figure out how to calculate our weight updates for the backward pass.

We actually have the calculations below so I'm going to use these to fill in the forward pass values. Remember that σ_h was the identity function so it isn't in the cell structure.

To make the backprop calculations simpler we're going to introduce some temporary variables (F and C)

Memory Cell Structures - Peephole LSTM



We can calculate the partial derivatives that we'll need to know for backprop for each of our activation functions

Remember that σ_g is sigmoid and σ_c is tanh.

$$F = c_{t-1} * f_t$$

$$1$$

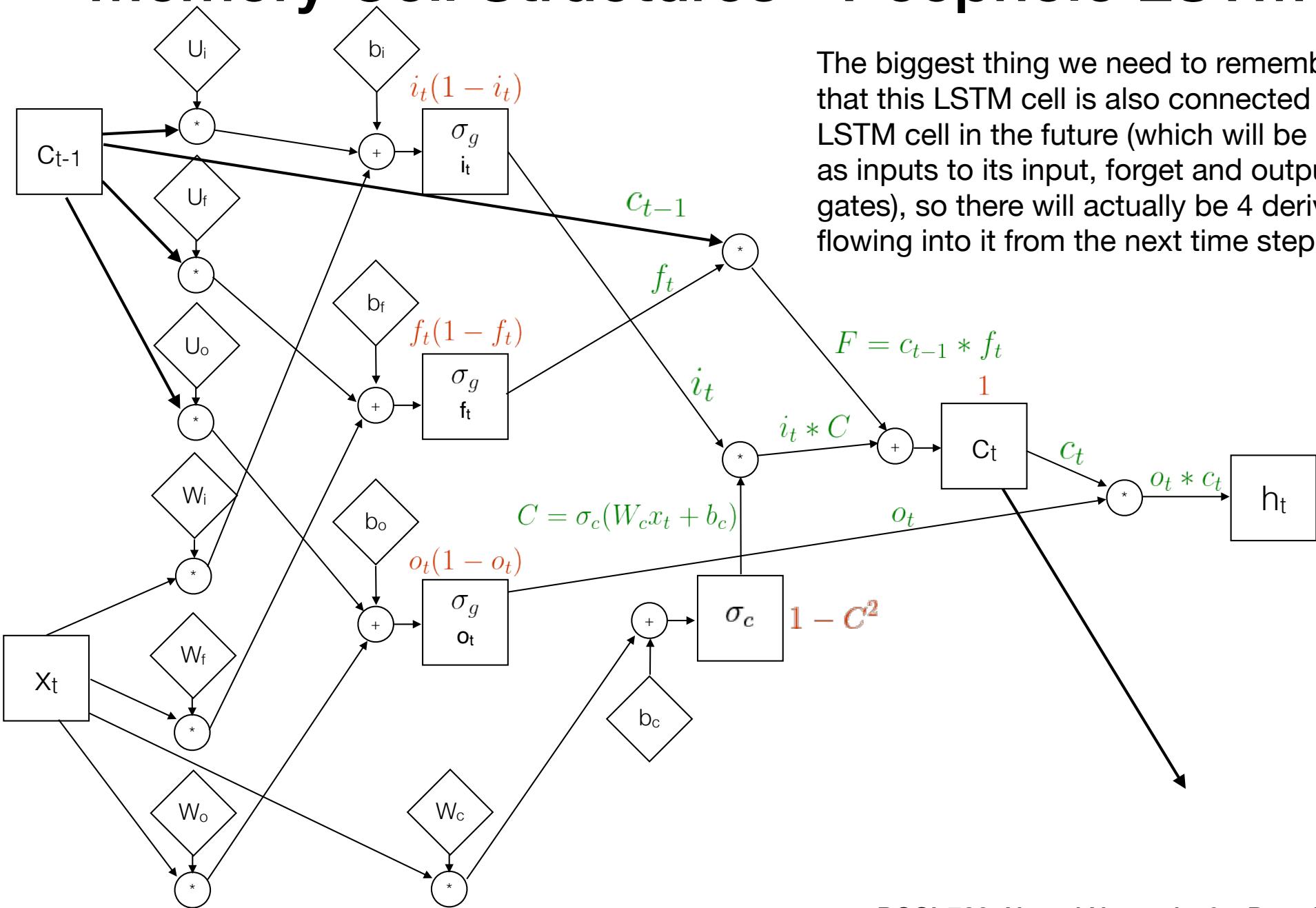
$$C_t$$

$$c_t$$

$$h_t$$

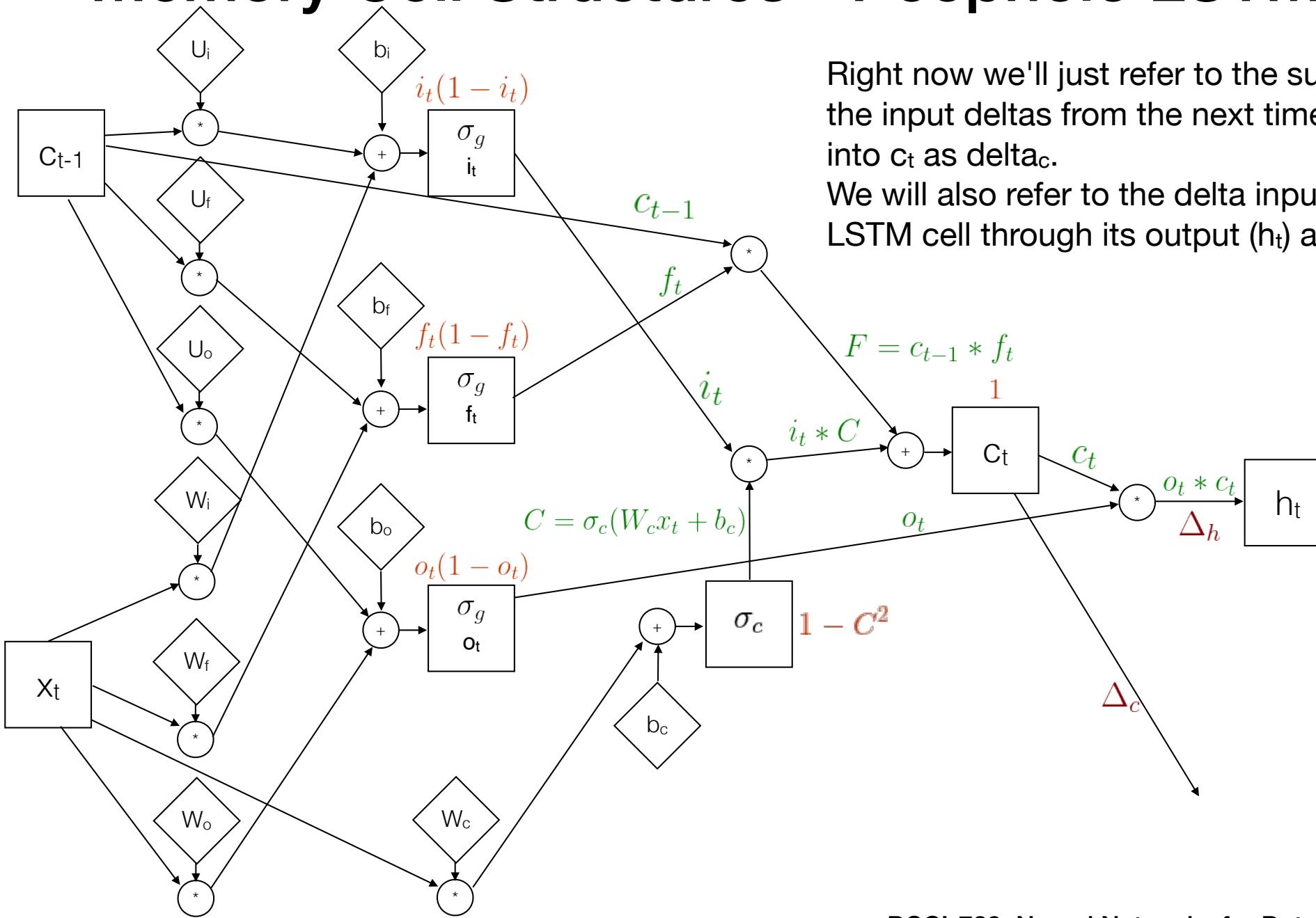
$$\begin{aligned} f_t &= \sigma_g(W_f x_t + U_f c_{t-1} + b_f) \\ i_t &= \sigma_g(W_i x_t + U_i c_{t-1} + b_i) \\ o_t &= \sigma_g(W_o x_t + U_o c_{t-1} + b_o) \\ c_t &= f_t * c_{t-1} + i_t * \sigma_c(W_c x_t + b_c) \\ h_t &= \sigma_h(o_t * c_t) \end{aligned}$$

Memory Cell Structures - Peephole LSTM



The biggest thing we need to remember is that this LSTM cell is also connected to the LSTM cell in the future (which will be using it as inputs to its input, forget and output gates), so there will actually be 4 derivatives flowing into it from the next time step.

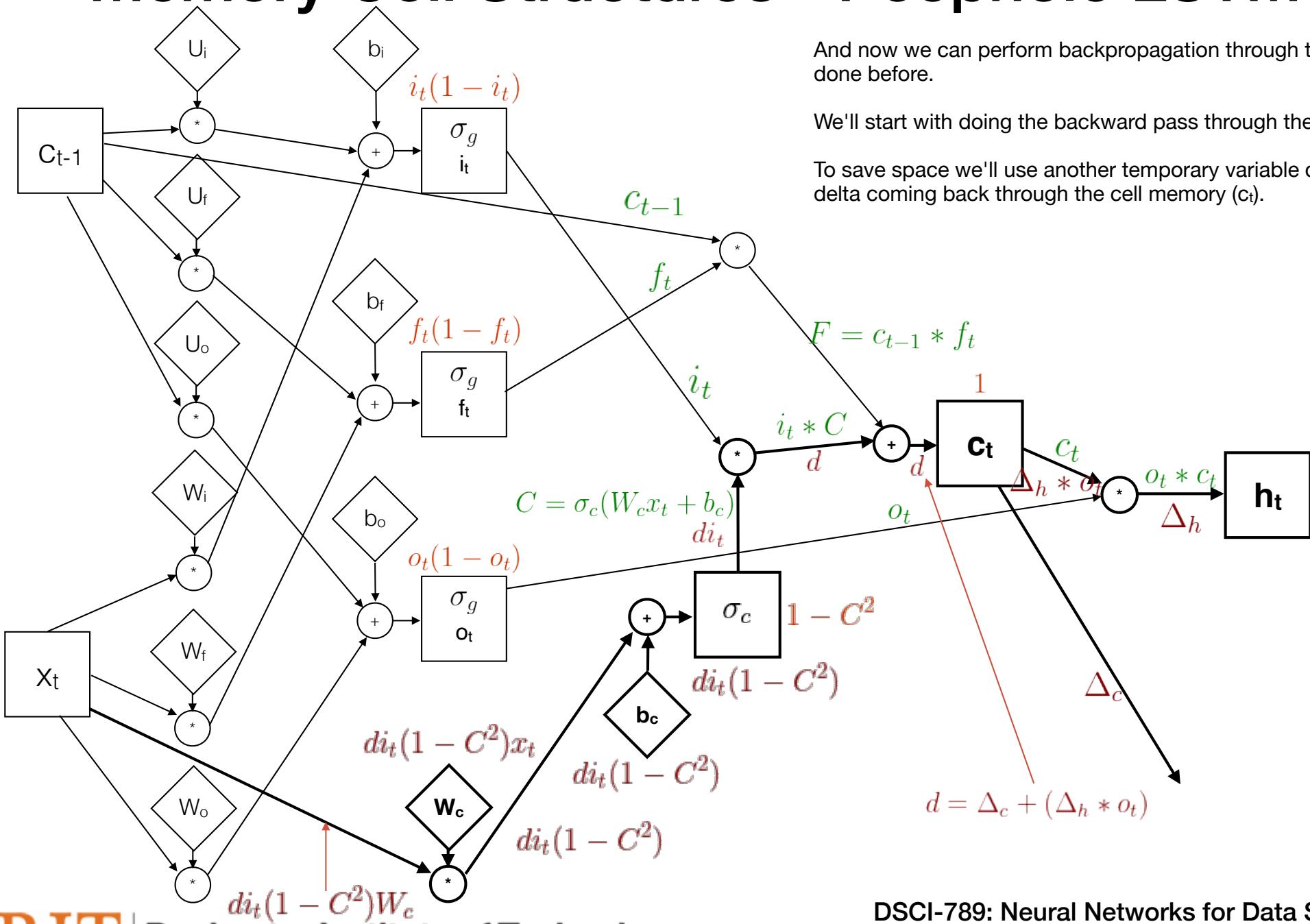
Memory Cell Structures - Peephole LSTM



Right now we'll just refer to the sum of the input deltas from the next time step into c_t as Δ_c .

We will also refer to the delta input to this LSTM cell through its output (h_t) as Δ_h

Memory Cell Structures - Peephole LSTM

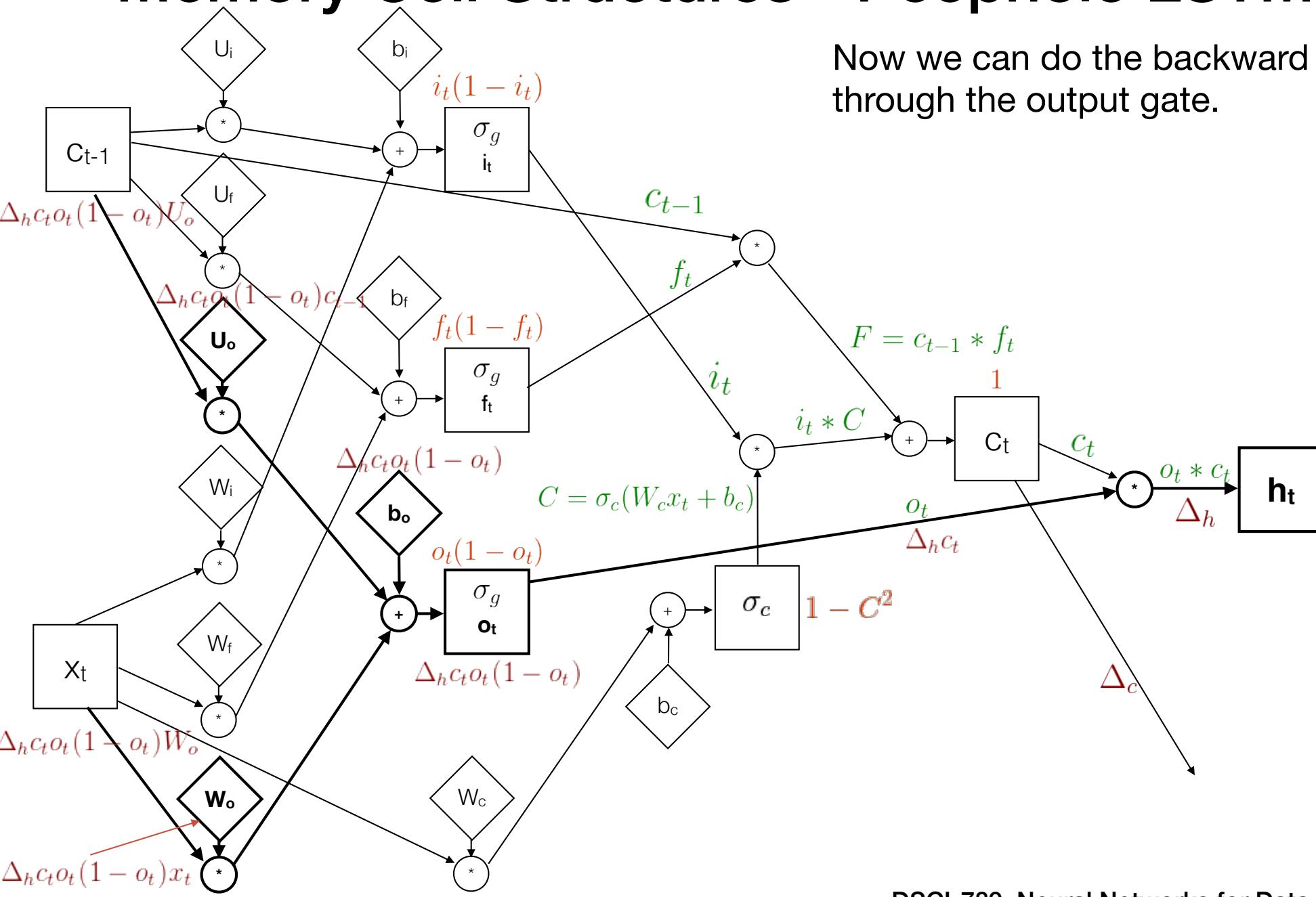


And now we can perform backpropagation through the cell as done before.

We'll start with doing the backward pass through the cell gate.

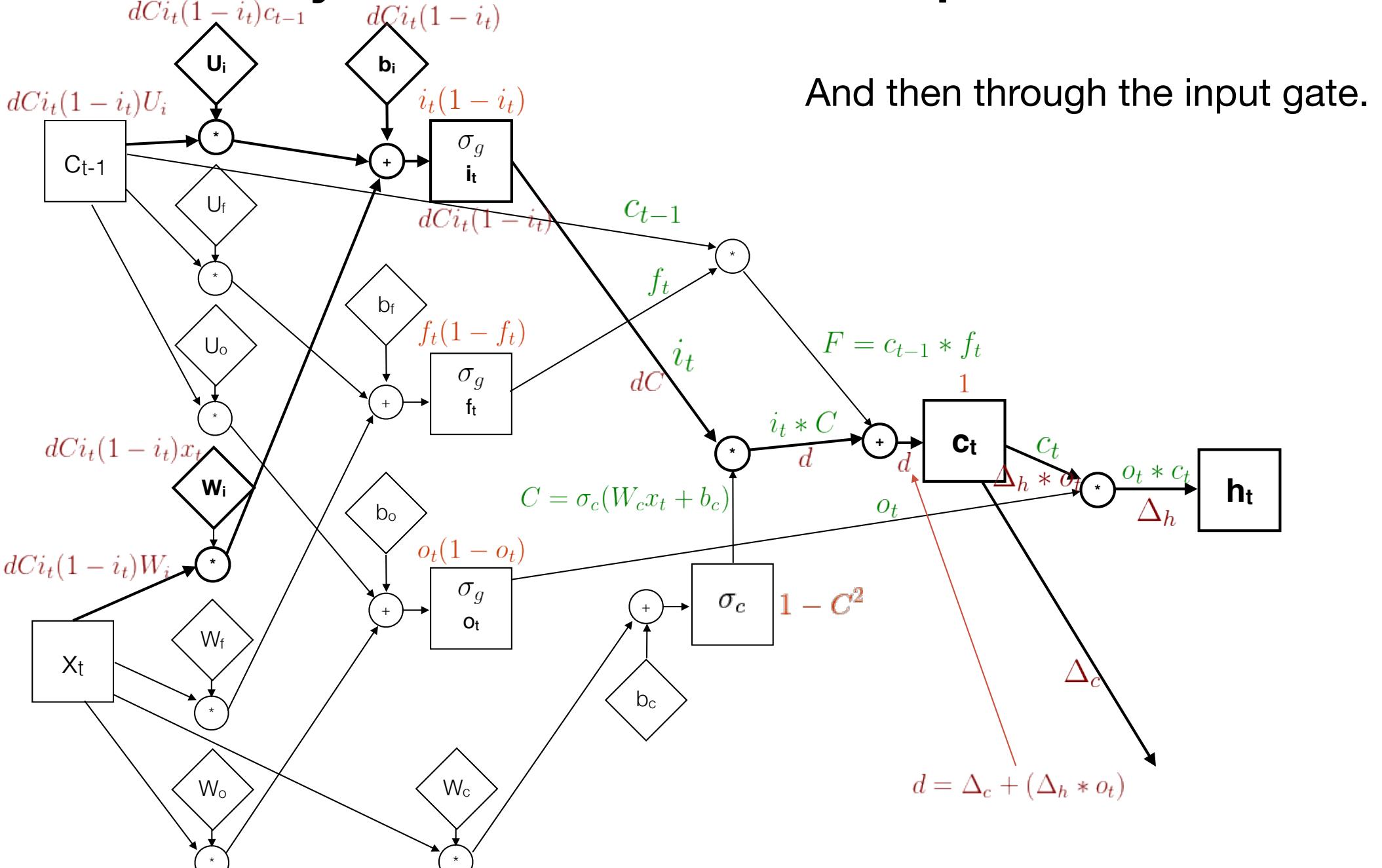
To save space we'll use another temporary variable d for the delta coming back through the cell memory (c_t).

Memory Cell Structures - Peephole LSTM



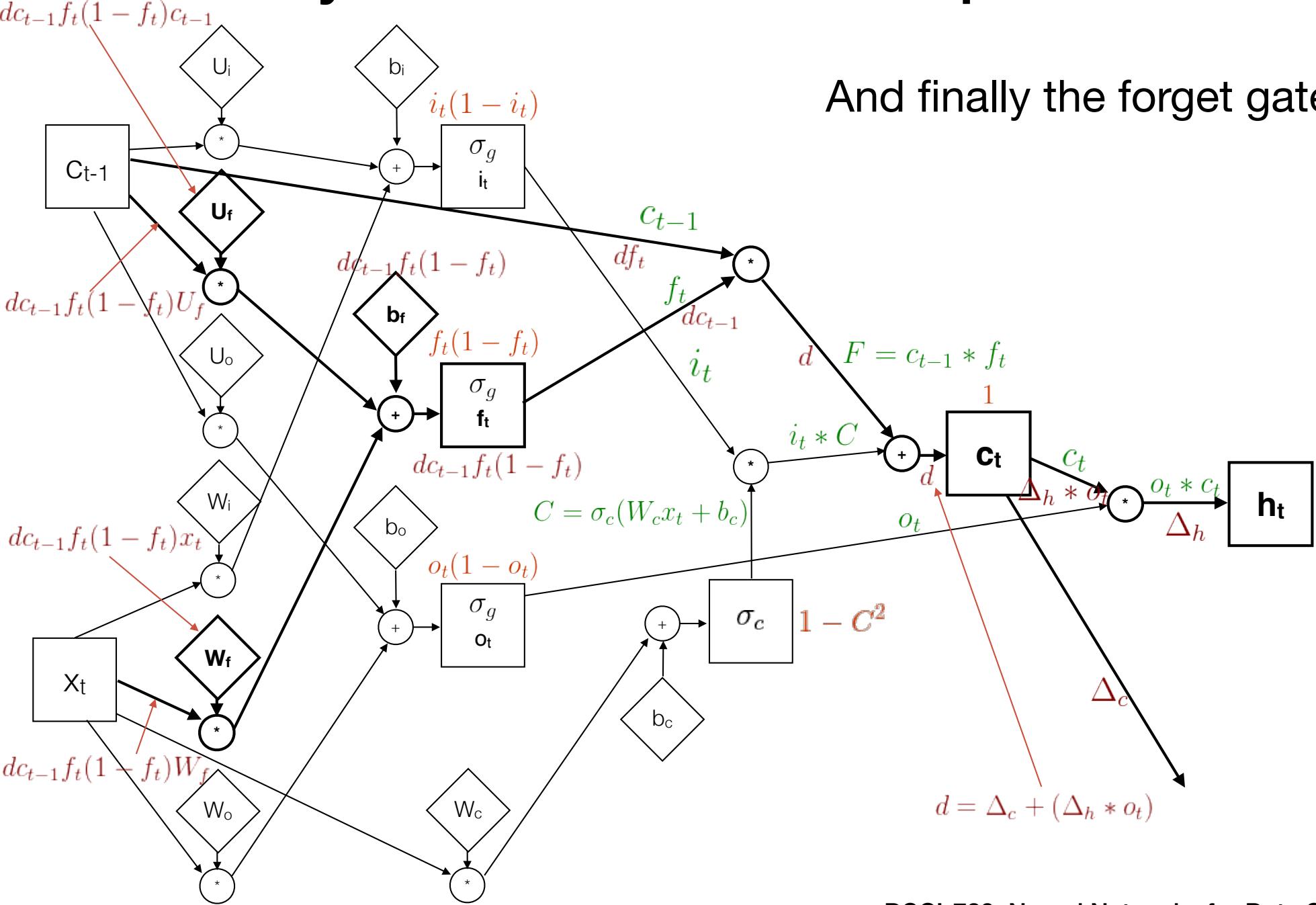
Now we can do the backward pass through the output gate.

Memory Cell Structures - Peephole LSTM

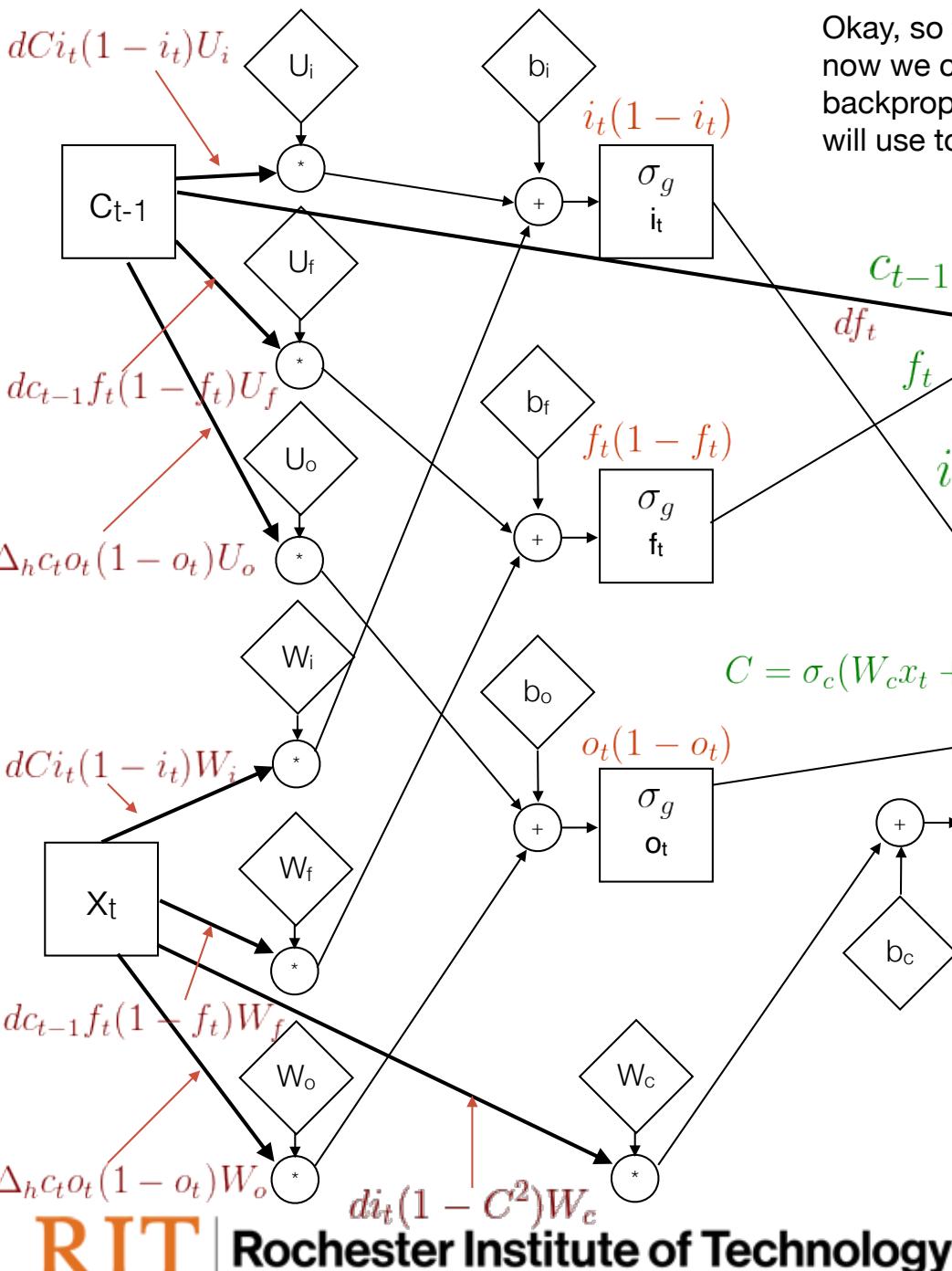


Memory Cell Structures - Peephole LSTM

And finally the forget gate.

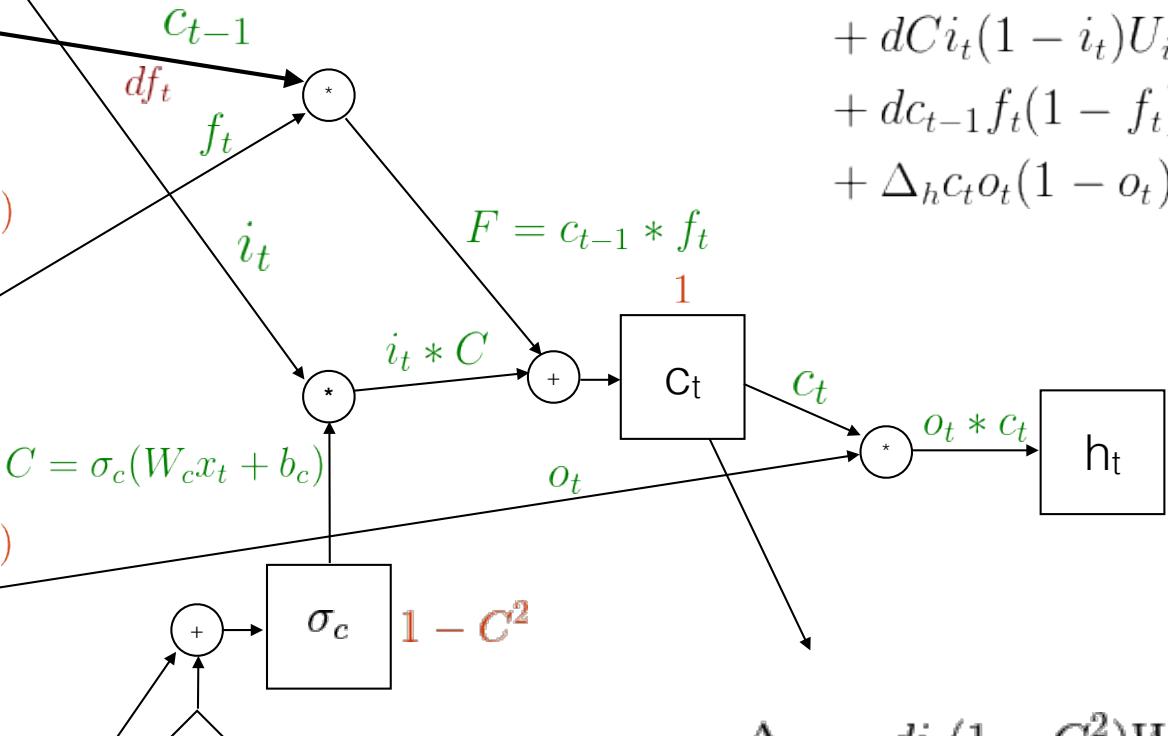


Memory Cell Structures - Peephole LSTM



Okay, so that was a lot, but we calculated the deltas for each weight, and now we can sum up all the deltas which will be passed to c_{t-1} for when we do backprop on the previous time step, as well as all the deltas for x_t which we will use to backprop over all incoming weights to the LSTM at this time step.

$$\begin{aligned}\Delta_{c_{t-1}} = & df_t \\ & + dCi_t(1 - i_t)U_i \\ & + dc_{t-1}f_t(1 - f_t)U_f \\ & + \Delta_h c_t o_t(1 - o_t)U_o\end{aligned}$$



$$\begin{aligned}\Delta_{x_t} = & di_t(1 - C^2)W_c \\ & + dCi_t(1 - i_t)W_i \\ & + dc_{t-1}f_t(1 - f_t)W_f \\ & + \Delta_h c_t o_t(1 - o_t)W_o\end{aligned}$$

Memory Cell Structures - Peephole LSTM

Forward pass equations:

$$f_t = \sigma_g(W_f x_t + U_f c_{t-1} + b_f)$$

$$i_t = \sigma_g(W_i x_t + U_i c_{t-1} + b_i)$$

$$o_t = \sigma_g(W_o x_t + U_o c_{t-1} + b_o)$$

$$c_t = f_t * c_{t-1} + i_t * \sigma_c(W_c x_t + b_c)$$

$$h_t = \sigma_h(o_t * c_t)$$

Backward pass equations and deltas:

Bais deltas:

$$d = \Delta_c + (\Delta_h * o_t)$$

$$C = \sigma_c(W_c x_t + b_c)$$

$$\Delta_{b_i} = dCi_t(1 - i_t)$$

$$\Delta_{b_f} = dc_{t-1}f_t(1 - f_t)$$

$$\Delta_{b_o} = \Delta_h c_t o_t (1 - o_t)$$

$$\Delta_{b_e} = di_t(1 - C^2)$$

Weight deltas:

$$\Delta_{U_i} = dCi_t(1 - i_t)c_{t-1}$$

$$\Delta_{U_f} = dc_{t-1}f_t(1 - f_t)c_{t-1}$$

$$\Delta_{U_o} = \Delta_h c_t o_t (1 - o_t)c_{t-1}$$

$$\Delta_{W_i} = dCi_t(1 - i_t)x_t$$

$$\Delta_{W_f} = dc_{t-1}f_t(1 - f_t)x_t$$

$$\Delta_{W_o} = \Delta_h c_t o_t (1 - o_t)x_t$$

$$\Delta_{W_e} = di_t(1 - C^2)x_t$$

So to sum up, we have the forward pass equations, and then all the deltas for the biases, weights and those that get passed to the previous time step cell, as well as back through the input.

Input and Cell deltas:

$$\begin{aligned}\Delta_{c_{t-1}} = & df_t \\ & + dCi_t(1 - i_t)U_i \\ & + dc_{t-1}f_t(1 - f_t)U_f \\ & + \Delta_h c_t o_t (1 - o_t)U_o \\ \Delta_{x_t} = & di_t(1 - C^2)W_c \\ & + dCi_t(1 - i_t)W_i \\ & + dc_{t-1}f_t(1 - f_t)W_f \\ & + \Delta_h c_t o_t (1 - o_t)W_o\end{aligned}$$

Memory Cell Structures - Peephole LSTM

Forward pass equations:

$$f_t = \sigma_g(W_f x_t + U_f c_{t-1} + b_f)$$

$$i_t = \sigma_g(W_i x_t + U_i c_{t-1} + b_i)$$

$$o_t = \sigma_g(W_o x_t + U_o c_{t-1} + b_o)$$

$$c_t = f_t * c_{t-1} + i_t * \sigma_c(W_c x_t + b_c)$$

$$h_t = \sigma_h(o_t * c_t)$$

Note that there is a lot of duplication in the equations for the deltas, so when calculating them in the LSTM cell make use of temporary variables to save on computation (don't duplicate multiplications, etc if you don't have to).

Backward pass equations and deltas:

Bais deltas:

$$d = \Delta_c + (\Delta_h * o_t)$$

$$C = \sigma_c(W_c x_t + b_c)$$

$$\Delta_{b_i} = dC i_t (1 - i_t)$$

$$\Delta_{b_f} = d c_{t-1} f_t (1 - f_t)$$

$$\Delta_{b_o} = \Delta_h c_t o_t (1 - o_t)$$

$$\Delta_{b_e} = d i_t (1 - C^2)$$

Weight deltas:

$$\Delta_{U_i} = d C i_t (1 - i_t) c_{t-1}$$

$$\Delta_{U_f} = d c_{t-1} f_t (1 - f_t) c_{t-1}$$

$$\Delta_{U_o} = \Delta_h c_t o_t (1 - o_t) c_{t-1}$$

$$\Delta_{W_i} = d C i_t (1 - i_t) x_t$$

$$\Delta_{W_f} = d c_{t-1} f_t (1 - f_t) x_t$$

$$\Delta_{W_o} = \Delta_h c_t o_t (1 - o_t) x_t$$

$$\Delta_{W_e} = d i_t (1 - C^2) x_t$$

$$\begin{aligned}\Delta_{c_{t-1}} = & df_t \\& + d C i_t (1 - i_t) U_i \\& + d c_{t-1} f_t (1 - f_t) U_f \\& + \Delta_h c_t o_t (1 - o_t) U_o \\\\Delta_{x_t} = & d i_t (1 - C^2) W_c \\& + d C i_t (1 - i_t) W_i \\& + d c_{t-1} f_t (1 - f_t) W_f \\& + \Delta_h c_t o_t (1 - o_t) W_o\end{aligned}$$

Memory Cell Structures - Peephole LSTM

Forward pass equations:

$$f_t = \sigma_g(W_f x_t + U_f c_{t-1} + b_f)$$

$$i_t = \sigma_g(W_i x_t + U_i c_{t-1} + b_i)$$

$$o_t = \sigma_g(W_o x_t + U_o c_{t-1} + b_o)$$

$$c_t = f_t * c_{t-1} + i_t * \sigma_c(W_c x_t + b_c)$$

$$h_t = \sigma_h(o_t * c_t)$$

Be sure to remember that the value $\Delta_{c_{t-1}}$ is passed in as Δ_c when doing backprop for the previous time step. For the last time step, Δ_c would be 0.

Backward pass equations and deltas:

Bais deltas:

$$d = \Delta_c + (\Delta_h * o_t)$$

$$C = \sigma_c(W_c x_t + b_c)$$

$$\Delta_{b_i} = dC i_t (1 - i_t)$$

$$\Delta_{b_f} = dC_{t-1} f_t (1 - f_t)$$

$$\Delta_{b_o} = \Delta_h c_t o_t (1 - o_t)$$

$$\Delta_{b_e} = d i_t (1 - C^2)$$

Weight deltas:

$$\Delta_{U_i} = dC i_t (1 - i_t) c_{t-1}$$

$$\Delta_{U_f} = dC_{t-1} f_t (1 - f_t) c_{t-1}$$

$$\Delta_{U_o} = \Delta_h c_t o_t (1 - o_t) c_{t-1}$$

$$\Delta_{W_i} = dC i_t (1 - i_t) x_t$$

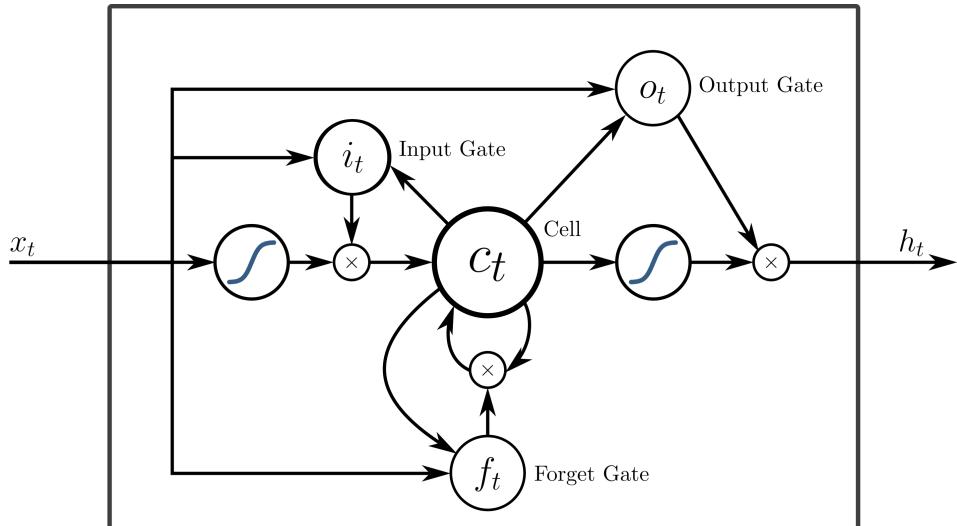
$$\Delta_{W_f} = dC_{t-1} f_t (1 - f_t) x_t$$

$$\Delta_{W_o} = \Delta_h c_t o_t (1 - o_t) x_t$$

$$\Delta_{W_e} = d i_t (1 - C^2) x_t$$

$$\begin{aligned}\Delta_{c_{t-1}} = & df_t \\& + dC i_t (1 - i_t) U_i \\& + dC_{t-1} f_t (1 - f_t) U_f \\& + \Delta_h c_t o_t (1 - o_t) U_o \\\\Delta_{x_t} = & d i_t (1 - C^2) W_c \\& + dC i_t (1 - i_t) W_i \\& + dC_{t-1} f_t (1 - f_t) W_f \\& + \Delta_h c_t o_t (1 - o_t) W_o\end{aligned}$$

LSTM Initialization



Peephole LSTM [1,2]

As other memory cell structures (which we'll discuss in a bit) gained in popularity, Jozefowicz et al. did a study to understand why GRUs were outperforming LSTMs consistently [1]. They found that adding 1 to the forget gate bias when initializing weights significantly improved its performance. So when initializing LSTMs you can use the same RNN weight initialization methods as before, but add 1 to the forget gate bias (b_f) for improved results.

Conceptually this should make sense as well. The weight initialization methods typically set weights to a small value near 0. If the output of the forget gate is very small it essentially erases any memory from previous time steps which makes it harder for the LSTM cell to learn. Adding 1 to the bias pushes the output of the forget gate to near 1, which means initializing it to a point where it's preserving the information from previous time steps and using that as a starting point instead.

[1] Jozefowicz, Rafal, Wojciech Zaremba, and Ilya Sutskever. *An empirical exploration of recurrent network architectures*. International conference on machine learning. 2015. <http://proceedings.mlr.press/v37/jozefowicz15.pdf>

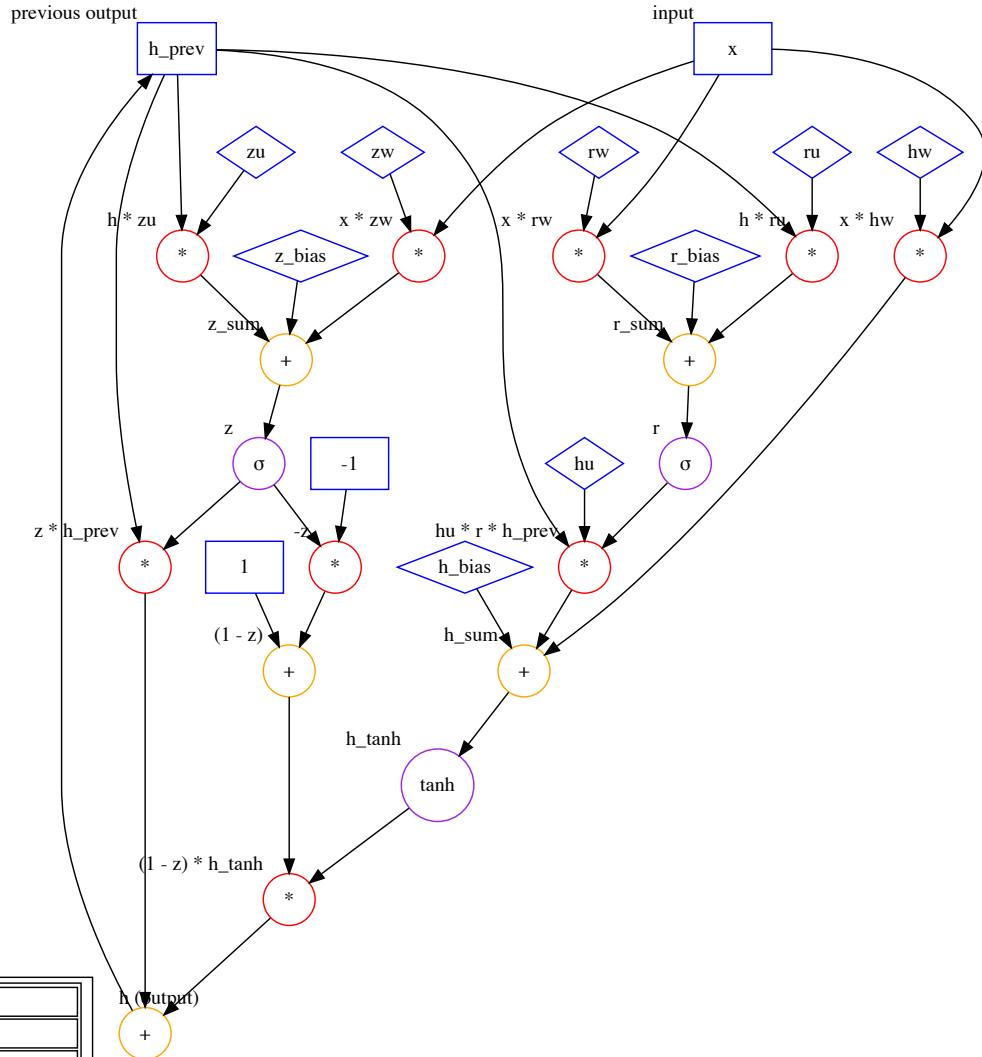
Other Memory Cell Structures

Memory Cell Structures - GRU

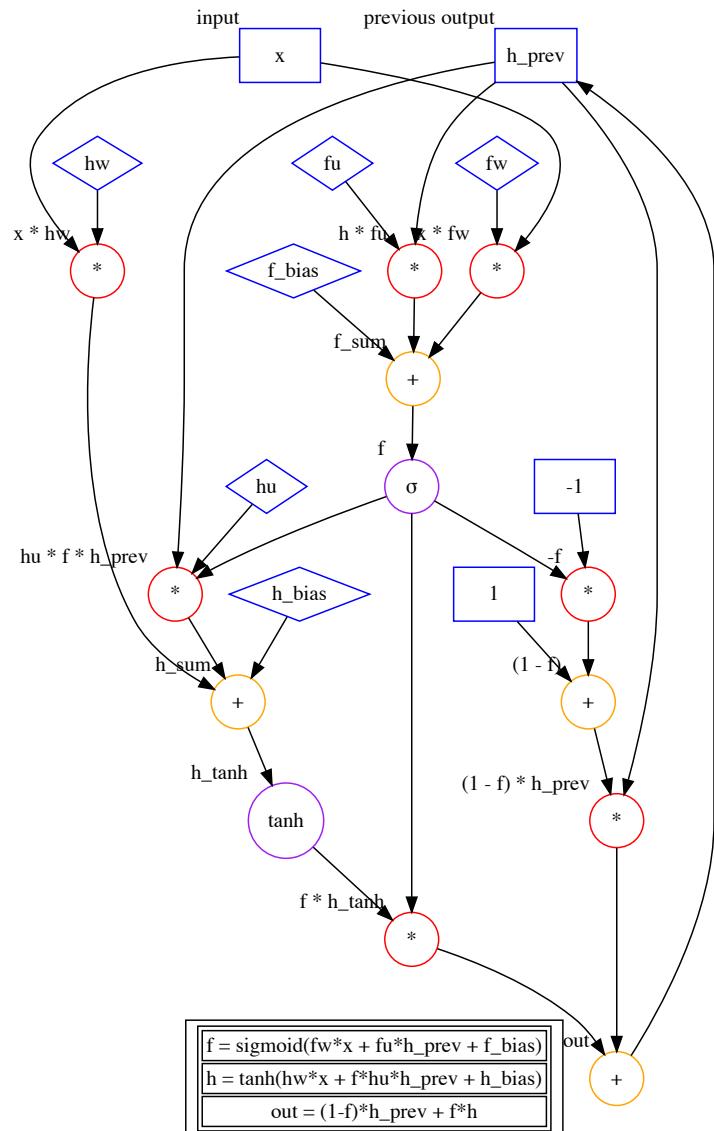
Gated recurrent units (GRUs) were first introduced in 2014 by Kyunghyun Cho et al. [2], which are similar to LSTM cells except without an output gate. As such it requires fewer trainable parameters than an LSTM (9).

[2] Cho, Kyunghyun; van Merriënboer, Bart; Gulcehre, Caglar; Bahdanau, Dzmitry; Bougares, Fethi; Schwenk, Holger; Bengio, Yoshua (2014). **Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.** *arXiv:1406.1078*

$$\begin{aligned} z &= \sigma(zw^*x + zu^*h_{\text{prev}} + z_bias) \\ r &= \sigma(rw^*x + ru^*h_{\text{prev}} + r_bias) \\ h &= z^*h_{\text{prev}} + (1-z)^*\tanh(hw^*x + hu^*r^*h_{\text{prev}} + h_bias) \end{aligned}$$



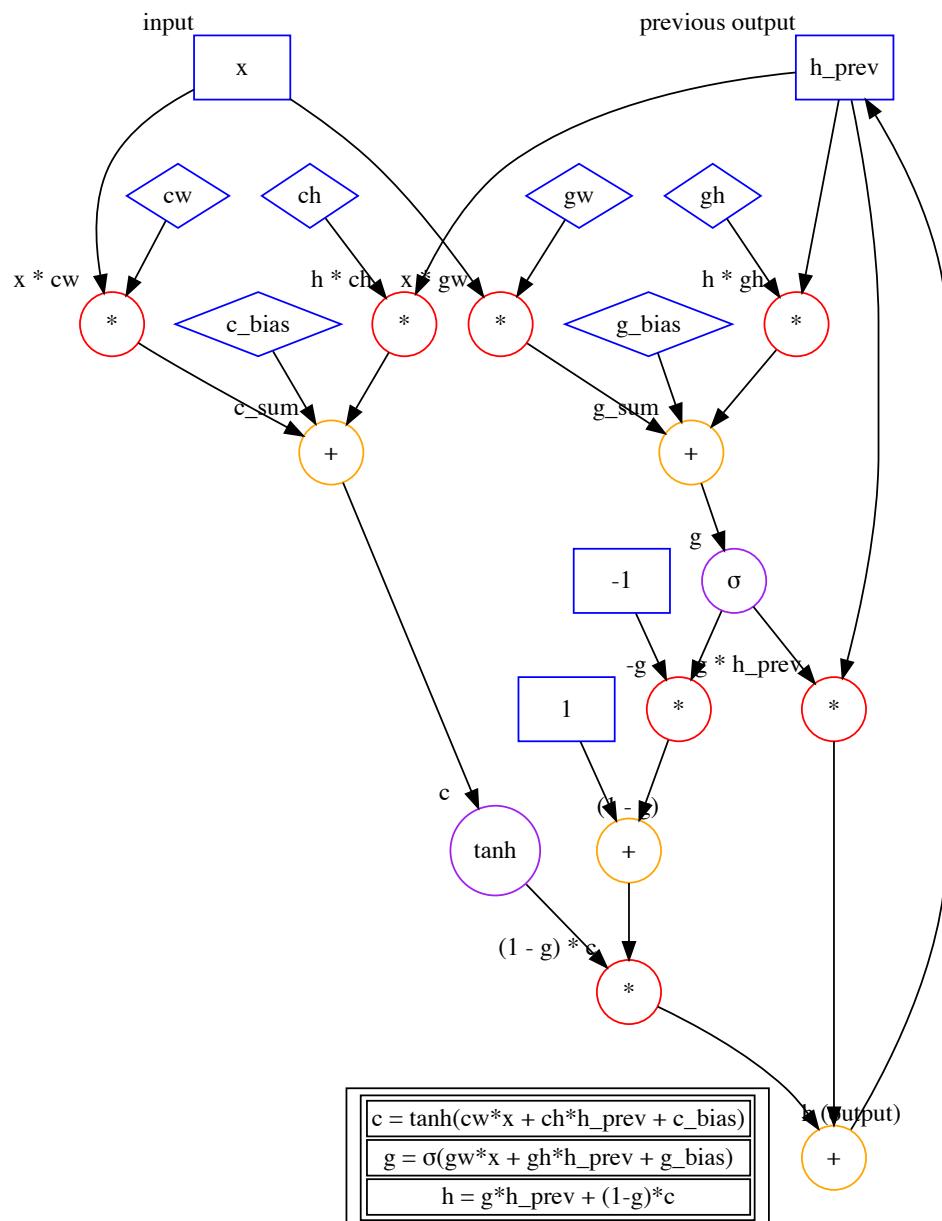
Memory Cell Structures - MGU



Minimal gated units (MGUs) were proposed by Zhou et al. in 2016 [3]. It is another example of an effective recurrent cell with a low number of trainable parameters (6).

[3] Gou-Bing Zhou, Jianxin Wu, Chen-Lin Zhang and Zhi-Hua Zhou. **Minimal gated unit for recurrent neural networks.** *International Journal of Automation and Computing* 13.3 (2016): 226-234.
APA

Memory Cell Structures - UGRNN

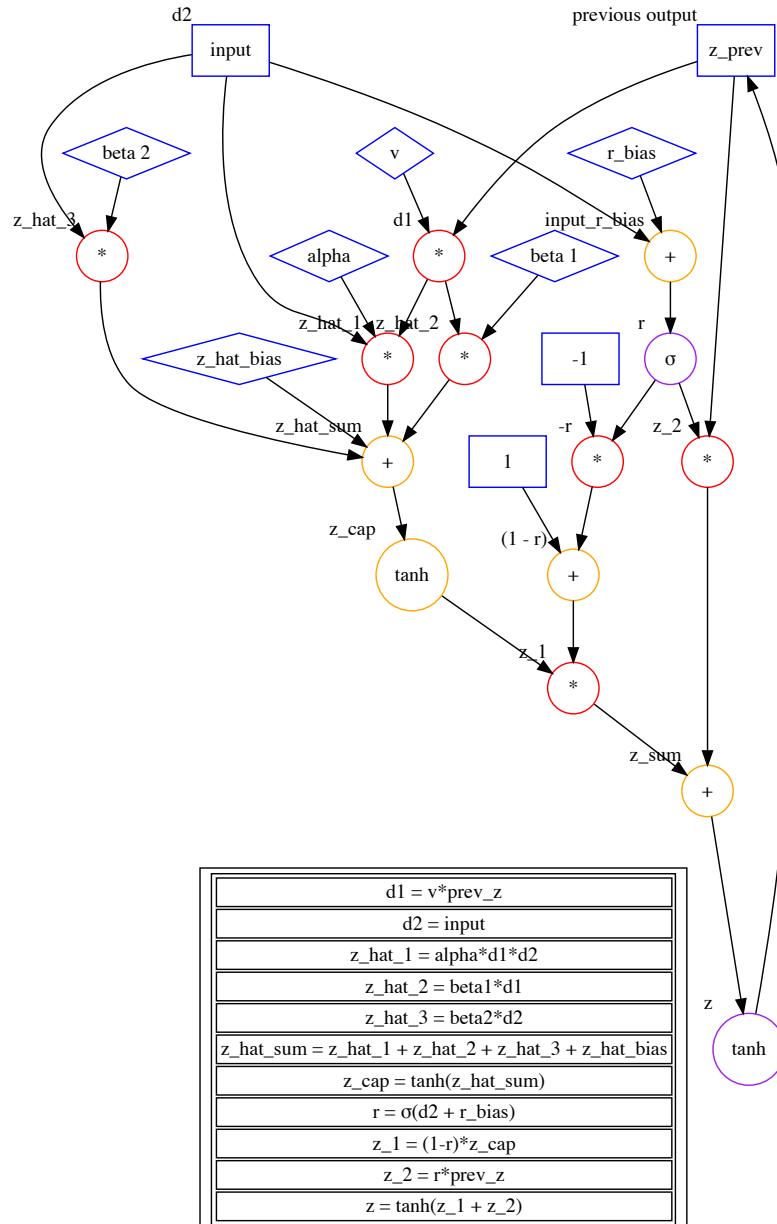


Update gate recurrent neural networks (UGRNNs) were introduced in 2016 by Collins et al. [4]

UGRNNs are another simple model with only 6 trainable parameters.

[4] Collins, Jasmine, Jascha Sohl-Dickstein, and David Sussillo. **Capacity and trainability in recurrent neural networks**. *arXiv preprint arXiv:1611.09913* (2016).

Memory Cell Structures - Delta-RNN



Delta-RNN cells were first developed by Ororbia et al. in 2017 [5], and have shown to have comparable performance to other memory cells with fewer trainable parameters (6).

[5] Ororbia II, Alexander G., Tomas Mikolov, and David Reitter. **Learning simpler language models with the differential state framework.** *Neural computation* 29.12 (2017): 3327-3352.

Lecture 8

Convolutional Neural Networks and the Forward Pass

DSCI-789: Neural Networks for Data Science

Travis Desell (tjdvse@rit.edu)
Associate Professor
Department of Software Engineering



ROCHESTER INSTITUTE OF TECHNOLOGY

Overview

- Images as Neural Network Inputs
 - MNIST
 - CIFAR-10
- Convolutional Neural Network Components:
 - Nodes - Feature Maps
 - Edges - Filters
 - Pooling
- CNN Architectures:
 - Simple Test Architectures
 - LeNet-5
 - AlexNet
 - VGGNet
 - ResNet

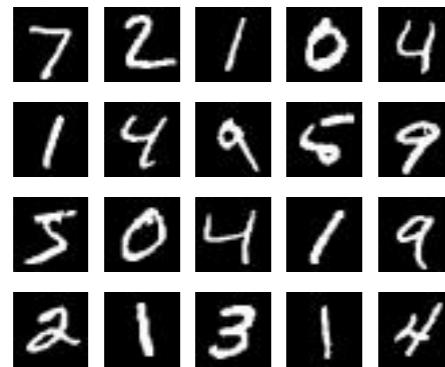
Images as Neural Network Inputs

Images as Neural Network Inputs

- Up until now, the largest input we've had to a neural network was the mushroom data set with 126 inputs.
- Using images as inputs can quickly blow this out of the water.
- Images typically have 1, 3 or 4 channels. Black and white images have 1 channel, RGB (red green blue) images have 3, and sometimes there is an additional 4th alpha channel (for transparency).

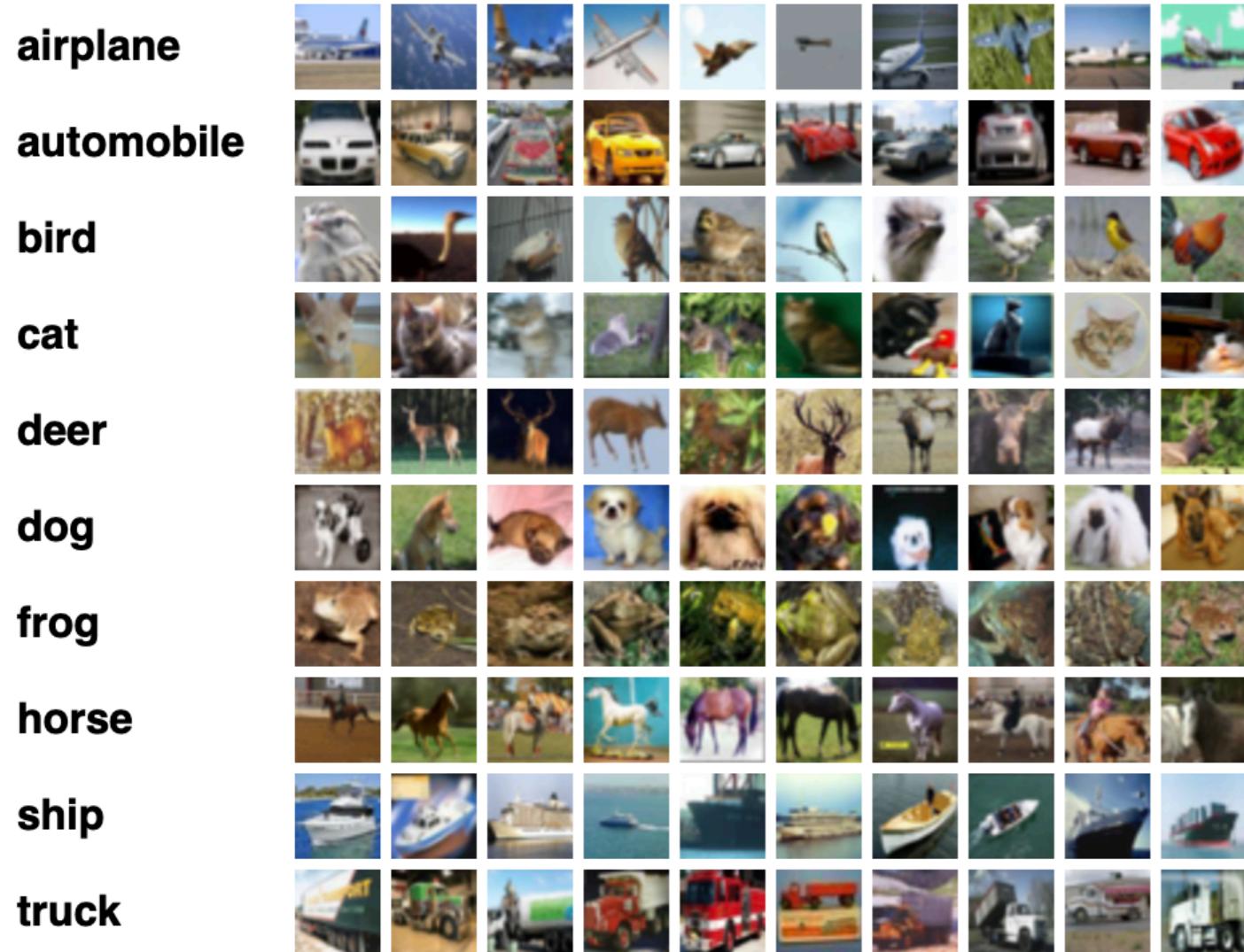
MNIST

- The MNIST data set is one of the most studied data sets for convolutional neural networks.
- It contains 60k training images and 10k testing images.
- Each image is black and white, with 28x28 pixels (see left for examples).
- They represent different handwritten digits, 0-9.
- With 28x28 pixels as an input, this puts our neural network inputs at 784, an order of magnitude larger than the mushroom data set.



<http://yann.lecun.com/exdb/mnist/>

CIFAR-10



- CIFAR-10 is another well studied data set that is even more challenging than MNIST.
- It contains 50k training images and 10k testing images.
- Each image is color (RGB), with 32x32 pixels (see left for examples), from 10 different classes.
- With 32x32x3 pixels as an input, this puts our neural network inputs at 3072, an order of magnitude larger than MNIST.
- There is also CIFAR-100 which takes the same images but divides the 10 classes into 10 subclasses.

<https://www.cs.toronto.edu/~kriz/cifar.html>

Images as Neural Network Inputs

- So even with the simplest 1 layer fully connected neural network for MNIST, with an equal number of hidden nodes to inputs, we would have a neural network with 614,656 weights from the input to hidden nodes, 784 bias weights, and 7,840 weights from the hidden nodes to the 10 output nodes: 623,280 weights in total. If we wanted to have a second layer, this would add an additional $614,656 + 784$ weights!
- For CIFAR-10 and a similar network we would have 9,437,184 weights from the input to hidden nodes, 3,072 bias weights, and 30,720 weights from the hidden nodes to the 10 outputs nodes: 9,740,976 weights in total. If we wanted to have a second layer, this would add an additional $3,437,184 + 3,072$ weights!
- Not only do these networks have a massive amount of weights, they do not train very well or lead to accurate results. Convolutional Neural Networks (CNNs) provide a much more effective way to structure our neural networks assuming images as input.

Convolutional Neural Network Components

CNN Components

- CNNs generally consist of three different components:
 - Nodes in a CNN are typically referred to as *feature maps*. Each feature map is either 2 or 3 dimensional, with a width, height and number of channels.
 - Edges in a CNN are typically referred to as *filters*. Each filter is also 2 or 3 dimensional. Each edge acts as a convolutional filter from one node to another, reducing the input node size by the filter size.
 - CNNs also have pooling layers, which act to reduce feature map sizes without requiring any trainable parameters (weights).

CNN Components

- CNNs are almost always trained with minibatch gradient descent to better utilize GPU resources and for other techniques such as batch normalization, so each node will have 3 dimensions (batch size x width x height) or 4 dimensions (batch size x channels x width x height), making them more complex than standard nodes or recurrent nodes.

CNN Nodes - Feature Maps

CNN Nodes - Feature Maps (2D)

255	0	0	0	255
0	255	0	255	0
0	0	255	0	0
0	255	0	255	0
255	0	0	0	255

- For single channel images (like MNIST) each feature map (or node in the network) will have a width and height.
- For the input nodes, each will have the same width and height as the input images.
- Left shows a sample 5x5 image of an X. Pixel values are usually stored as bytes, each with a value between 0 and 255 (0 as black, 255 as white).

CNN Nodes - Feature Maps (2D)



- Each node will take a batch size of inputs, so there will be multiple inputs being fed into each node, one for each image in the batch.

CNN Nodes - Feature Maps (3D)

255	0	0	0	255
0	255	0	255	0
0	0	255	0	0
0	255	0	255	0
255	0	0	0	255

red

255	0	0	0	255
0	255	0	255	0
0	0	255	0	0
0	255	0	255	0
255	0	0	0	255

green

255	0	0	0	255
0	255	0	255	0
0	0	255	0	0
0	255	0	255	0
255	0	0	0	255

blue

- For color input images (like CIFAR-10) each input images has 3 channels, so there will be a red channel input, green channel input, and blue channel input.
- It is possible to scale this up even farther (e.g., if the image has an alpha channel), or if it is multispectral imagery from more advanced sensors.

CNN Nodes - Feature Maps (3D)

batch size	255	0	0	0	255
red channel	255	0	0	0	255
	255	0	0	0	255
	0	255	0	255	0
	0	0	255	0	0
	0	255	0	255	0
	255	0	0	0	255

- Similarly, batches of images are fed in, so each channel will have multiple images. So in this case a CNN node will contain 4 dimensional data.

batch size	255	0	0	0	255
green channel	255	0	0	0	255
	255	0	0	0	255
	0	255	0	255	0
	0	0	255	0	0
	0	255	0	255	0
	255	0	0	0	255

batch size	255	0	0	0	255
blue channel	255	0	0	0	255
	255	0	0	0	255
	0	255	0	255	0
	0	0	255	0	0
	0	255	0	255	0
	255	0	0	0	255

Feature Maps - Bias

- There are two different methods for handling bias in feature maps:
 - *tied bias*: where the entire feature map has one bias value which is added to each cell.
 - *untied bias*: where each cell has its own bias value.
- Untied bias generally results in CNNs with greater learning potential but comes with more complexity, e.g., a CIFAR-10 node with tied bias would only have 1 additional weight for the bias, while the untied version would have 3,072 additional weights for the biases.
- It can be good to investigate if your CNN actually needs untied biases or not, because if it does not you can make a more efficient CNN.

Feature Maps - Activation Functions

- Common activation functions for CNNs are ReLU, ReLU-5, ReLU-6, Leaky ReLU and linear. Occasionally you will see sigmoid or tanh but these aren't commonly used anymore (or used only in the final layers which are one dimensional and fully connected).
- To forward propagate through a CNN node, simply apply the activation function to each cell after adding the bias to it.

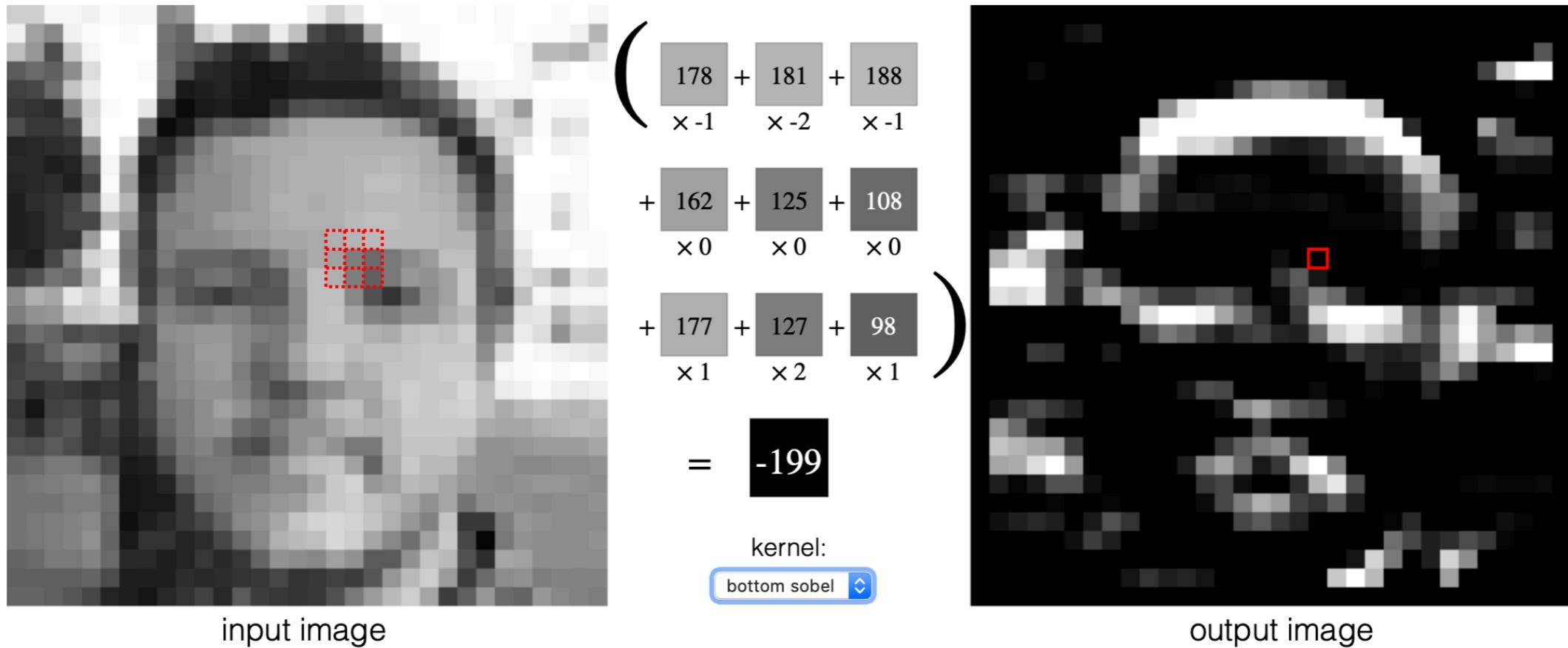
CNN Nodes - Input Node Padding

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	255	0	0	0	255	0	0
0	0	0	255	0	255	0	0	0
0	0	0	0	255	0	0	0	0
0	0	0	255	0	255	0	0	0
0	0	255	0	0	0	255	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

- For input nodes, many CNN architectures add padding to get the inputs to be a particular size, and also to reduce biasing pixels farther from the edges (we'll see why that would happen when we look at filters).
- A common approach is 0-padding, where 0 values are added (the left shows 0-padding of 2).
- However some works have shown that using Gaussian noise (randomly generated numbers with a normal distribution) instead of 0-padding can provide better predictions.

CNN Edges - Filters

CNN Edges - Filters

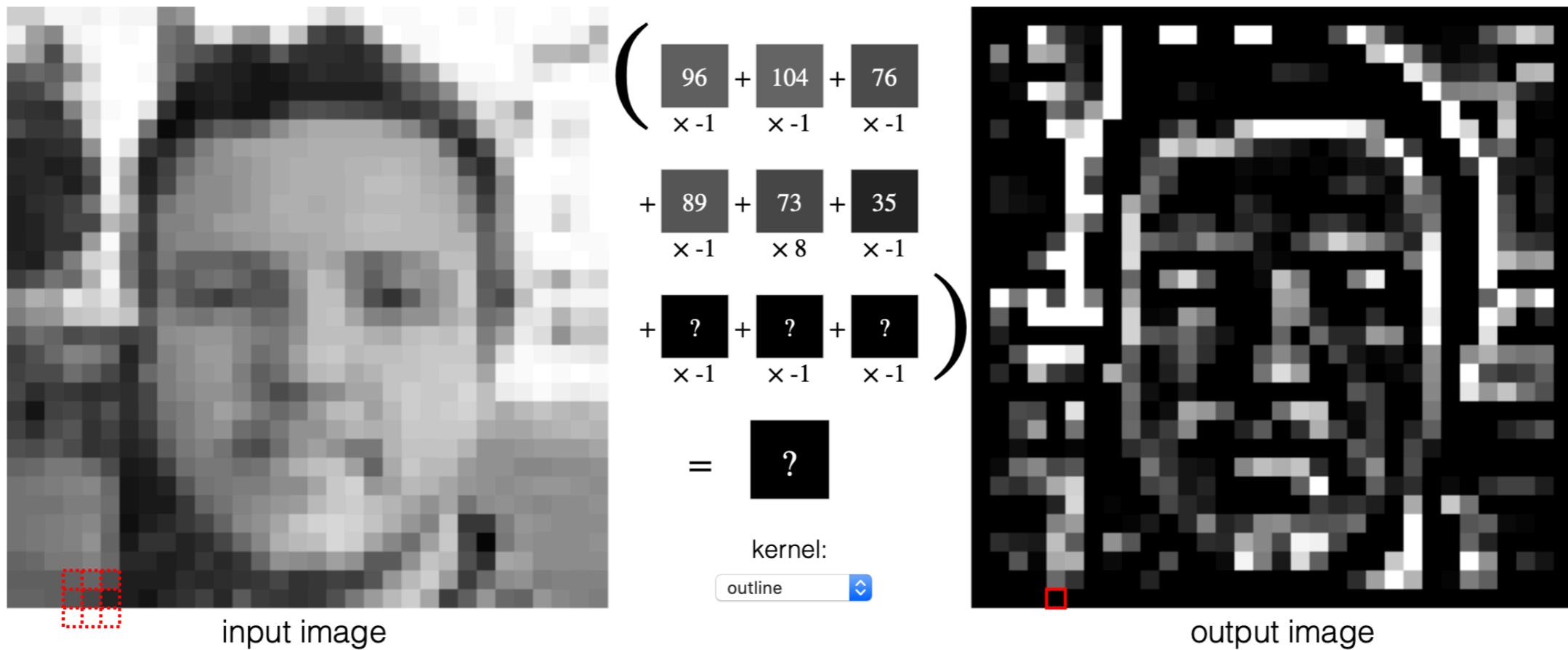


-1	-2	-1
0	0	0
1	2	1

- *Filters* (also called *convolutions* or *kernels*) provide a way to extract *features* from images (which is why CNN nodes are called feature maps).
- The above provides an example of applying a "bottom sobel" filter (right) to the above image using a convolution.

<https://setosa.io/ev/image-kernels/>

CNN Edges - Filters



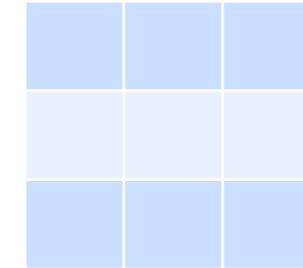
- Using a different filter (outline, to the left) different features are extracted from the image.
- The idea behind training CNNs is to learn the filter values which produce the features which can best perform the prediction task.

<https://setosa.io/ev/image-kernels/>

CNN Edges - Filters (2D)

255	0	0	0	255
0	255	0	255	0
0	0	255	0	0
0	255	0	255	0
255	0	0	0	255

-1	-1	-1
-1	2	-1
-1	-1	-1

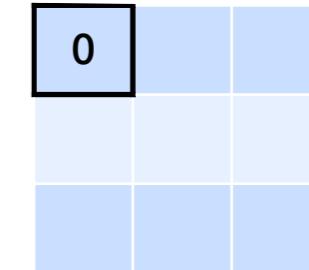


- Filters provide an efficient way to generate feature maps without having to fully connect cells between CNN nodes.
- Given inputX, inputY, filterX and filterY:
 - $\text{outputX} = \text{inputX} - \text{filterX} + 1$
 - $\text{outputY} = \text{inputY} - \text{filterY} + 1$

CNN Edges - Filters (2D)

255	0	0	0	255
0	255	0	255	0
0	0	255	0	0
0	255	0	255	0
255	0	0	0	255

-1	-1	-1
-1	2	-1
-1	-1	-1



- Given the filter size, for every filterX x filterY in the input, each cell is multiplied by the corresponding filter cell and these are all summed together into the corresponding output cell.
- $$\text{output}[0][0] = (255 * -1) + (0 * -1) + (0 * -1) + (0 * -1) + (255 * 2) + (0 * -1) + (0 * -1) + (0 * -1) + (255 * -1) = 0$$

CNN Edges - Filters (2D)

255	0	0	0	255
0	255	0	255	0
0	0	255	0	0
0	255	0	255	0
255	0	0	0	255

-1	-1	-1
-1	2	-1
-1	-1	-1

0	-765	

- This is *strided* over the inputs.
- $\text{output}[0][1] = (0 * -1) + (0 * -1) + (0 * -1) + (255 * -1) + (0 * 2) + (255 * -1) + (0 * -1) + (255 * -1) + (0 * -1) = -765$

CNN Edges - Filters (2D)

255	0	0	0	255
0	255	0	255	0
0	0	255	0	0
0	255	0	255	0
255	0	0	0	255

-1	-1	-1
-1	2	-1
-1	-1	-1

0	-765	0

- This is *strided* over the inputs.
- $\text{output}[0][2] = (0 * -1) + (0 * -1) + (255 * -1) + (0 * -1) + (255 * 2) + (0 * -1) + (255 * -1) + (0 * -1) + (0 * -1) = 0$

CNN Edges - Filters (2D)

255	0	0	0	255
0	255	0	255	0
0	0	255	0	0
0	255	0	255	0
255	0	0	0	255

-1	-1	-1
-1	2	-1
-1	-1	-1

0	-765	0
-765		

- This is *strided* over the inputs.
- $\text{output}[1][0] = (0 * -1) + (255 * -1) + (0 * -1) + (0 * -1) + (0 * 2) + (255 * -1) + (0 * -1) + (255 * -1) + (0 * -1) = -765$

CNN Edges - Filters (2D)

255	0	0	0	255
0	255	0	255	0
0	0	255	0	0
0	255	0	255	0
255	0	0	0	255

-1	-1	-1
-1	2	-1
-1	-1	-1

0	-765	0
-765	-510	

- This is *strided* over the inputs.
- $\text{output}[1][1] = (255 * -1) + (0 * -1) + (255 * -1) + (0 * -1) + (255 * 2) + (0 * -1) + (255 * -1) + (0 * -1) + (255 * -1) = -510$

CNN Edges - Filters (2D)

255	0	0	0	255
0	255	0	255	0
0	0	255	0	0
0	255	0	255	0
255	0	0	0	255

-1	-1	-1
-1	2	-1
-1	-1	-1

0	-765	0
-765	-510	-765

- This is *strided* over the inputs.
- $\text{output}[1][2] = (0 * -1) + (255 * -1) + (0 * -1) + (255 * -1) + (0 * 2) + (0 * -1) + (0 * -1) + (255 * -1) + (0 * -1) = -765$

CNN Edges - Filters (2D)

255	0	0	0	255
0	255	0	255	0
0	0	255	0	0
0	255	0	255	0
255	0	0	0	255

-1	-1	-1
-1	2	-1
-1	-1	-1

0	-765	0
-765	-510	-765
0		

- This is *strided* over the inputs.
- $\text{output}[2][0] = (0 * -1) + (0 * -1) + (255 * -1) + (0 * -1) + (255 * 2) + (0 * -1) + (255 * -1) + (0 * -1) + (0 * -1) = 0$

CNN Edges - Filters (2D)

255	0	0	0	255
0	255	0	255	0
0	0	255	0	0
0	255	0	255	0
255	0	0	0	255

-1	-1	-1
-1	2	-1
-1	-1	-1

0	-765	0
-765	-510	-765
0	-765	

- This is *strided* over the inputs.
- $\text{output}[2][1] = (0 * -1) + (255 * -1) + (0 * -1) + (255 * -1) + (0 * 2) + (255 * -1) + (0 * -1) + (0 * -1) + (0 * -1) = 0$

CNN Edges - Filters (2D)

255	0	0	0	255
0	255	0	255	0
0	0	255	0	0
0	255	0	255	0
255	0	0	0	255

-1	-1	-1
-1	2	-1
-1	-1	-1

0	-765	0
-765	-510	-765
0	-765	0

- This is *strided* over the inputs.
- $\text{output}[2][2] = (255 * -1) + (0 * -1) + (0 * -1) + (0 * -1) + (255 * 2) + (0 * -1) + (0 * -1) + (0 * -1) + (255 * -1) = 0$

CNN Edges - Filters (2D)

255	0	0	0	255
0	255	0	255	0
0	0	255	0	0
0	255	0	255	0
255	0	0	0	255

-1	-1	-1
-1	2	-1
-1	-1	-1

0	-765	0
-765	-510	-765
0	-765	0

- What is nice about this is that going from a 5x5 input to a 3x3 input only required 3x3 (9) weights -- we can reuse them as we stride the filter over the image.
- If this had been fully connected, it would have required 5x5 (25) x 3x3 (9) = 225 weights.

CNN Edges - Filters (2D) with Padding

0	0	0	0	0
0	5	2	1	0
0	1	3	2	0
0	7	1	9	0
0	0	0	0	0

-1	-1	-1
-1	2	-1
-1	-1	2

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

- Now we can work through an example with padding.
- The input feature map has a padding of 1, and the output feature map has a padding of 2.
- Both are 0 padded.
- The non-padding cells are outlined in dark grey.
- Our filter is still 3x3.
- Note with an input padding of 1 and a 3x3 filter, the output feature map is the same size as the input (not counting padding) -- using a padding and filter size resulting in the same output feature map size is common in modern CNN architectures (and we'll see more of that later).

CNN Edges - Filters (2D) with Padding

0	0	0	0	0
0	5	2	1	0
0	1	3	2	0
0	7	1	9	0
0	0	0	0	0

-1	-1	-1
-1	2	-1
-1	-1	2

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	13	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

- We stride over the input node the same as before, except this time we need to make sure that the output of the filter is offset by the padding.

CNN Edges - Filters (2D) with Padding

0	0	0	0	0
0	5	2	1	0
0	1	3	2	0
0	7	1	9	0
0	0	0	0	0

-1	-1	-1
-1	2	-1
-1	-1	2

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	13	-2	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

- We stride over the input node the same as before, except this time we need to make sure that the output of the filter is offset by the padding.

CNN Edges - Filters (2D) with Padding

0	0	0	0	0	0
0	5	2	1	0	0
0	1	3	2	0	0
0	7	1	9	0	0
0	0	0	0	0	0

-1	-1	-1
-1	2	-1
-1	-1	2

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	13	-2	-5	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

- We stride over the input node the same as before, except this time we need to make sure that the output of the filter is offset by the padding.

CNN Edges - Filters (2D) with Padding

0	0	0	0	0
0	5	2	1	0
0	1	3	2	0
0	7	1	9	0
0	0	0	0	0

-1	-1	-1
-1	2	-1
-1	-1	2

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	13	-2	-5	0	0
0	0	-13	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

- We stride over the input node the same as before, except this time we need to make sure that the output of the filter is offset by the padding.

CNN Edges - Filters (2D) with Padding

0	0	0	0	0
0	5	2	1	0
0	1	3	2	0
0	7	1	9	0
0	0	0	0	0

-1	-1	-1
-1	2	-1
-1	-1	2

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	13	-2	-5	0	0
0	0	-13	5	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

- We stride over the input node the same as before, except this time we need to make sure that the output of the filter is offset by the padding.

CNN Edges - Filters (2D) with Padding

0	0	0	0	0	0
0	5	2	1	0	0
0	1	3	2	0	0
0	7	1	9	0	0
0	0	0	0	0	0

-1	-1	-1
-1	2	-1
-1	-1	2

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	13	-2	-5	0	0	0
0	0	-13	5	-12	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

- We stride over the input node the same as before, except this time we need to make sure that the output of the filter is offset by the padding.

CNN Edges - Filters (2D) with Padding

0	0	0	0	0
0	5	2	1	0
0	1	3	2	0
0	7	1	9	0
0	0	0	0	0

-1	-1	-1
-1	2	-1
-1	-1	2

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	13	-2	-5	0	0
0	0	-13	5	-12	0	0
0	0	9	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

- We stride over the input node the same as before, except this time we need to make sure that the output of the filter is offset by the padding.

CNN Edges - Filters (2D) with Padding

0	0	0	0	0
0	5	2	1	0
0	1	3	2	0
0	7	1	9	0
0	0	0	0	0

-1	-1	-1
-1	2	-1
-1	-1	2

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	13	-2	-5	0	0
0	0	-13	5	-12	0	0
0	0	9	-20	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

- We stride over the input node the same as before, except this time we need to make sure that the output of the filter is offset by the padding.

CNN Edges - Filters (2D) with Padding

0	0	0	0	0
0	5	2	1	0
0	1	3	2	0
0	7	1	9	0
0	0	0	0	0

-1	-1	-1
-1	2	-1
-1	-1	2

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	13	-2	-5	0	0
0	0	-13	5	-12	0	0
0	0	9	-20	12	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

- We stride over the input node the same as before, except this time we need to make sure that the output of the filter is offset by the padding.

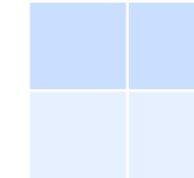
CNN Edges - Filters (3D)

1	4	3	2
6	-2	-3	5
7	8	-1	0
4	3	2	1

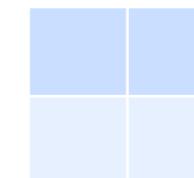
3	2	3	1
4	5	4	2
-2	3	-3	1
-1	-1	2	4

-5	-3	2	3
-4	-1	-2	-3
3	6	3	2
2	5	4	2

-1	-1	-1
-1	2	-1
-1	-1	-1



-2	1	-2
1	2	1
-2	1	-2



- This can also work in 3 (or more) dimensions, striding over each.
- Here's an example of a 4x4x3 input, a 3x3x2 filter, which results in a 2x2x2 output.

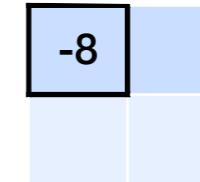
CNN Edges - Filters (3D)

1	4	3	2
6	-2	-3	5
7	8	-1	0
4	3	2	1

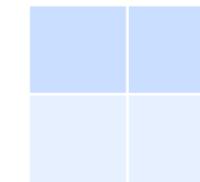
3	2	3	1
4	5	4	2
-2	3	-3	1
-1	-1	2	4

-5	-3	2	3
-4	-1	-2	-3
3	6	3	2
2	5	4	2

-1	-1	-1
-1	2	-1
-1	-1	-1



-2	1	-2
1	2	1
-2	1	-2



- Start with the first two channels (RG) and sum up the products of the filters over the inputs.
- output[0][0][0] =

$$(1 * -1) + (4 * -1) + (3 * -1) +$$

$$(6 * -1) + (-2 * 2) + (-3 * -1) +$$

$$(7 * -1) + (8 * -1) + (-1 * -1) +$$

$$(3 * -2) + (2 * 1) + (3 * -2) +$$

$$(4 * 1) + (5 * 2) + (4 * 1) +$$

$$(-2 * -2) + (3 * 1) + (-3 * -2)$$

CNN Edges - Filters (3D)

1	4	3	2
6	-2	-3	5
7	8	-1	0
4	3	2	1

3	2	3	1
4	5	4	2
-2	3	-3	1
-1	-1	2	4

-5	-3	2	3
-4	-1	-2	-3
3	6	3	2
2	5	4	2

-1	-1	-1
-1	2	-1
-1	-1	-1

-8	-24

-2	1	-2
1	2	1
-2	1	-2

- Start with the first two channels (RG) and sum up the products of the filters over the inputs.
- output[0][0][1] =

$$(4 * -1) + (3 * -1) + (2 * -1) +$$

$$(-2 * -1) + (-3 * 2) + (5 * -1) +$$

$$(8 * -1) + (-1 * -1) + (0 * -1) +$$

$$(2 * -2) + (3 * 1) + (1 * -2) +$$

$$(5 * 1) + (4 * 2) + (2 * 1) +$$

$$(3 * -2) + (-3 * 1) + (1 * -2)$$

CNN Edges - Filters (3D)

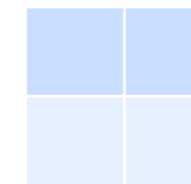
1	4	3	2
6	-2	-3	5
7	8	-1	0
4	3	2	1

3	2	3	1
4	5	4	2
-2	3	-3	1
-1	-1	2	4

-5	-3	2	3
-4	-1	-2	-3
3	6	3	2
2	5	4	2

-1	-1	-1
-1	2	-1
-1	-1	-1

-8	-24
-13	



-2	1	-2
1	2	1
-2	1	-2

- Start with the first two channels (RG) and sum up the products of the filters over the inputs.
- output[0][1][0] =

$$(6 * -1) + (-2 * -1) + (-3 * -1) +$$

$$(7 * -1) + (8 * 2) + (-1 * -1) +$$

$$(4 * -1) + (3 * -1) + (2 * -1) +$$

$$(4 * -2) + (5 * 1) + (4 * -2) +$$

$$(-2 * 1) + (3 * 2) + (-3 * 1) +$$

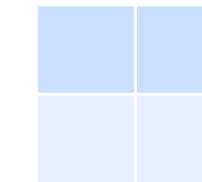
$$(-1 * -2) + (-1 * 1) + (2 * -2)$$

CNN Edges - Filters (3D)

1	4	3	2
6	-2	-3	5
7	8	-1	0
4	3	2	1

-1	-1	-1
-1	2	-1
-1	-1	-1

-8	-24
-13	-32



3	2	3	1
4	5	4	2
-2	3	-3	1
-1	-1	2	4

-2	1	-2
1	2	1
-2	1	-2

-5	-3	2	3
-4	-1	-2	-3
3	6	3	2
2	5	4	2

- Start with the first two channels (RG) and sum up the products of the filters over the inputs.
- output[0][1][1] =

$$(-2 * -1) + (-3 * -1) + (5 * -1) +$$

$$(8 * -1) + (-1 * 2) + (0 * -1) +$$

$$(3 * -1) + (2 * -1) + (1 * -1) +$$

$$(5 * -2) + (4 * 1) + (2 * -2) +$$

$$(3 * 1) + (-3 * 2) + (1 * 1) +$$

$$(-1 * -2) + (2 * 1) + (4 * -2)$$

CNN Edges - Filters (3D)

1	4	3	2
6	-2	-3	5
7	8	-1	0
4	3	2	1

3	2	3	1
4	5	4	2
-2	3	-3	1
-1	-1	2	4

-5	-3	2	3
-4	-1	-2	-3
3	6	3	2
2	5	4	2

-1	-1	-1
-1	2	-1
-1	-1	-1

-8	-24
-13	-32

-2	1	-2
1	2	1
-2	1	-2

-15

- And then stride on the z dimension (channels).
- $\text{output}[1][0][0] =$

$$(3 * -1) + (2 * -1) + (3 * -1) +$$

$$(4 * -1) + (5 * 2) + (4 * -1) +$$

$$(-2 * -1) + (3 * -1) + (-3 * -1) +$$

$$(-5 * -2) + (-3 * 1) + (2 * -2) +$$

$$(-4 * 1) + (-1 * 2) + (-2 * 1) +$$

$$(3 * -2) + (6 * 1) + (3 * -2)$$

CNN Edges - Filters (3D)

1	4	3	2
6	-2	-3	5
7	8	-1	0
4	3	2	1

3	2	3	1
4	5	4	2
-2	3	-3	1
-1	-1	2	4

-5	-3	2	3
-4	-1	-2	-3
3	6	3	2
2	5	4	2

-1	-1	-1
-1	2	-1
-1	-1	-1

-2	1	-2
1	2	1
-2	1	-2

-8	-24
-13	-32
-15	-25

- And then stride on the z dimension (channels).
- $\text{output}[1][0][1] =$

$$(2 * -1) + (3 * -1) + (1 * -1) +$$

$$(5 * -1) + (4 * 2) + (2 * -1) +$$

$$(3 * -1) + (-3 * -1) + (1 * -1) +$$

$$(-3 * -2) + (2 * 1) + (3 * -2) +$$

$$(-1 * 1) + (-2 * 2) + (-3 * 1) +$$

$$(6 * -2) + (3 * 1) + (2 * -2)$$

CNN Edges - Filters (3D)

1	4	3	2
6	-2	-3	5
7	8	-1	0
4	3	2	1

3	2	3	1
4	5	4	2
-2	3	-3	1
-1	-1	2	4

-5	-3	2	3
-4	-1	-2	-3
3	6	3	2
2	5	4	2

-1	-1	-1
-1	2	-1
-1	-1	-1

-2	1	-2
1	2	1
-2	1	-2

-8	-24
-13	-32

-15	-25
20	

- And then stride on the z dimension (channels).

- $\text{output}[1][1][0] =$

$$(4 * -1) + (5 * -1) + (4 * -1) + \\ (-2 * -1) + (3 * 2) + (-3 * -1) + \\ (-1 * -1) + (-1 * -1) + (2 * -1) +$$

$$(-4 * -2) + (-1 * 1) + (-2 * -2) + \\ (3 * 1) + (6 * 2) + (3 * 1) + \\ (2 * -2) + (5 * 1) + (4 * -2)$$

CNN Edges - Filters (3D)

1	4	3	2
6	-2	-3	5
7	8	-1	0
4	3	2	1

3	2	3	1
4	5	4	2
-2	3	-3	1
-1	-1	2	4

-5	-3	2	3
-4	-1	-2	-3
3	6	3	2
2	5	4	2

-1	-1	-1
-1	2	-1
-1	-1	-1

-2	1	-2
1	2	1
-2	1	-2

-8	-24
-13	-32
-15	-25
20	-16

- And then stride on the z dimension (channels).
- $\text{output}[1][1][1] =$

$$(5 * -1) + (4 * -1) + (2 * -1) +$$

$$(3 * -1) + (-3 * 2) + (1 * -1) +$$

$$(-1 * -1) + (2 * -1) + (4 * -1) +$$

$$(-1 * -2) + (-2 * 1) + (-3 * -2) +$$

$$(6 * 1) + (3 * 2) + (2 * 1) +$$

$$(5 * -2) + (4 * 1) + (2 * -2)$$

CNN Edges - Filters

- Typically the first set of convolutions (given a 3 channel input) will be x by y by 3 to make the rest of the feature maps in the network 2 dimensional, but this isn't always the case.
- A big part of CNN design is determining what size feature maps and filters to have such that they can connect together properly.

Pooling

Pooling

- The last major component of CNNs are pooling layers.
- Pooling layers are a way to downsample feature maps without requiring any trainable parameters (weights).
- They're effective ways of extracting important information from the feature maps and reducing the size of the CNN.
- The most common pooling layers are *max pooling* and *average pooling*.

Pooling

-5	-3	2	3	5	3
-4	-1	-2	-3	4	1
3	6	3	2	-2	-1
2	5	4	2	6	-4
1	3	4	-1	2	-3
2	7	5	-2	-9	3

- Pooling operations are applied to feature maps, resulting in a smaller output feature map (so they are similar to a filter). However they are typically applied to all nodes in a layer, which is why they are referred to as pooling layers.
- Pooling operations have both a *pool size* and a *stride*.
- Lets look at a max pooling operation on the above feature map with a pool size of 2 and a stride of 2 (the most common pooling operation).

Max Pooling

-5	-3	2	3	5	3
-4	-1	-2	-3	4	1
3	6	3	2	-2	-1
2	5	4	2	6	-4
1	3	4	-1	2	-3
2	7	5	-2	-9	3

max pool:
size 2x2, stride 2

The diagram illustrates a max pooling operation. A 2x2 input window is highlighted with a black border, containing the values -5, -3, 2, and 3. An arrow points from this window to a 1x1 output feature map. The output map contains a single value, -1, which is also enclosed in a black border.

-1

- Max pooling selects the largest value within the pool, and selects it for the output feature map. After that it strides over to the next pool.
- Since we're using a pool size of 2, we select the first 2x2 block of cells for the first pooling operation.
- With a stride of 2, we know we'll have $\text{inputX}/\text{stride}$ by $\text{inputY}/\text{stride}$ outputs (reducing the number of cells by a factor of 4!)

Max Pooling

-5	-3	2	3	5	3
-4	-1	-2	-3	4	1
3	6	3	2	-2	-1
2	5	4	2	6	-4
1	3	4	-1	2	-3
2	7	5	-2	-9	3

max pool:
size 2x2, stride 2

A diagram illustrating max pooling. On the left is a 6x6 input matrix with values ranging from -9 to 7. A 2x2 kernel is shown in the top-left corner, highlighted with a black border. An arrow points from this kernel to the right, leading to a 3x3 output matrix. The output matrix has only two non-zero values: a '3' in the top-left position and a '-1' in the second row, first column. All other positions are zero.

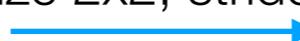
-1	3	

- Now we stride over by 2, and select the next max value in the pool.

Max Pooling

-5	-3	2	3	5	3
-4	-1	-2	-3	4	1
3	6	3	2	-2	-1
2	5	4	2	6	-4
1	3	4	-1	2	-3
2	7	5	-2	-9	3

max pool:
size 2x2, stride 2



-1	3	5

- And so on.

Max Pooling

-5	-3	2	3	5	3
-4	-1	-2	-3	4	1
3	6	3	2	-2	-1
2	5	4	2	6	-4
1	3	4	-1	2	-3
2	7	5	-2	-9	3

max pool:
size 2x2, stride 2

-1	3	5
6		

- Then we stride down by 2 to the next set of rows.

Max Pooling

-5	-3	2	3	5	3
-4	-1	-2	-3	4	1
3	6	3	2	-2	-1
2	5	4	2	6	-4
1	3	4	-1	2	-3
2	7	5	-2	-9	3

max pool:
size 2x2, stride 2

-1	3	5
6	4	

- And continue.

Max Pooling

-5	-3	2	3	5	3
-4	-1	-2	-3	4	1
3	6	3	2	-2	-1
2	5	4	2	6	-4
1	3	4	-1	2	-3
2	7	5	-2	-9	3

max pool:
size 2x2, stride 2



-1	3	5
6	4	6

- And continue.

Max Pooling

-5	-3	2	3	5	3
-4	-1	-2	-3	4	1
3	6	3	2	-2	-1
2	5	4	2	6	-4
1	3	4	-1	2	-3
2	7	5	-2	-9	3

max pool:
size 2x2, stride 2

-1	3	5
6	4	6
7		

- And continue.

Max Pooling

-5	-3	2	3	5	3
-4	-1	-2	-3	4	1
3	6	3	2	-2	-1
2	5	4	2	6	-4
1	3	4	-1	2	-3
2	7	5	-2	-9	3

max pool:
size 2x2, stride 2

-1	3	5
6	4	6
7	5	

- And continue.

Max Pooling

-5	-3	2	3	5	3
-4	-1	-2	-3	4	1
3	6	3	2	-2	-1
2	5	4	2	6	-4
1	3	4	-1	2	-3
2	7	5	-2	-9	3

max pool:
size 2x2, stride 2

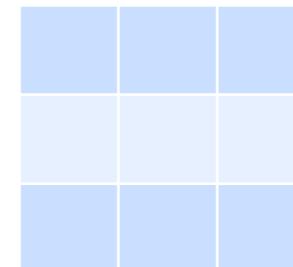
-1	3	5
6	4	6
7	5	3

- And now we have performed a max pooling operation with a size of 2x2 and a stride of 2, which resulted in a 4x reduction in cells, without adding any weights, which extracted the strongest positive signals from the input feature map.
- Note it is also possible to do absolute max pooling (which will take the largest positive or negative value).

Average Pooling

-5	-3	2	3	5	3	3	2
-4	-1	-2	-3	4	1	3	
3	6	3	2	-2	-1	4	
2	5	4	2	6	-4	5	
1	3	4	-1	2	-3	6	
2	7	5	-2	-9	3	2	
5	-2	3	8	-2	-1	-1	

avg pool:
size 3x3, stride 2



- Average pooling is similar, and does just what the name suggests. It takes the average of all the pooled values and passes that to the output feature map.
- Lets do this with a different size pool (3x3) but keeping the same stride of 2.

Average Pooling

-5	-3	2	3	5	3	2	
-4	-1	-2	-3	4	1	3	
3	6	3	2	-2	-1	4	
2	5	4	2	6	-4	5	
1	3	4	-1	2	-3	6	
2	7	5	-2	-9	3	2	
5	-2	3	8	-2	-1	-1	

avg pool:
size 3x3, stride 2

The diagram illustrates the average pooling process. A 3x3 input grid is shown on the left, with a 3x3 kernel highlighted in blue. An arrow points from the input grid to a 1x1 output grid on the right. The output grid contains a single value, -0.11, which is the result of averaging the values within the highlighted 3x3 kernel.

-0.11		

- Now we start with a 3x3 block of cells and find the average.

Average Pooling

-5	-3	2	3	5	3	2
-4	-1	-2	-3	4	1	3
3	6	3	2	-2	-1	4
2	5	4	2	6	-4	5
1	3	4	-1	2	-3	6
2	7	5	-2	-9	3	2
5	-2	3	8	-2	-1	-1

avg pool:
size 3x3, stride 2

-0.11	1.33

- And stride over 2 for the next average.

Average Pooling

-5	-3	2	3	5	3	2
-4	-1	-2	-3	4	1	3
3	6	3	2	-2	-1	4
2	5	4	2	6	-4	5
1	3	4	-1	2	-3	6
2	7	5	-2	-9	3	2
5	-2	3	8	-2	-1	-1

avg pool:
size 3x3, stride 2

-0.11	1.33	2.11

- And so on.

Average Pooling

-5	-3	2	3	5	3	2	
-4	-1	-2	-3	4	1	3	
3	6	3	2	-2	-1	4	
2	5	4	2	6	-4	5	
1	3	4	-1	2	-3	6	
2	7	5	-2	-9	3	2	
5	-2	3	8	-2	-1	-1	

avg pool:
size 3x3, stride 2

-0.11	1.33	2.11
3.44		

- Now we stride down 2 and continue.

Average Pooling

-5	-3	2	3	5	3	2
-4	-1	-2	-3	4	1	3
3	6	3	2	-2	-1	4
2	5	4	2	6	-4	5
1	3	4	-1	2	-3	6
2	7	5	-2	-9	3	2
5	-2	3	8	-2	-1	-1

avg pool:
size 3x3, stride 2

-0.11	1.33	2.11
3.44	2.22	

- And continue.

Average Pooling

-5	-3	2	3	5	3	2	
-4	-1	-2	-3	4	1	3	
3	6	3	2	-2	-1	4	
2	5	4	2	6	-4	5	
1	3	4	-1	2	-3	6	
2	7	5	-2	-9	3	2	
5	-2	3	8	-2	-1	-1	

avg pool:
size 3x3, stride 2

-0.11	1.33	2.11
3.44	2.22	1.44

- And continue.

Average Pooling

-5	-3	2	3	5	3	2	
-4	-1	-2	-3	4	1	3	
3	6	3	2	-2	-1	4	
2	5	4	2	6	-4	5	
1	3	4	-1	2	-3	6	
2	7	5	-2	-9	3	2	
5	-2	3	8	-2	-1	-1	

avg pool:
size 3x3, stride 2

-0.11	1.33	2.11
3.44	2.22	1.44
3.11		

- And stride down by another 2 for the last set of rows.

Average Pooling

-5	-3	2	3	5	3	2
-4	-1	-2	-3	4	1	3
3	6	3	2	-2	-1	4
2	5	4	2	6	-4	5
1	3	4	-1	2	-3	6
2	7	5	-2	-9	3	2
5	-2	3	8	-2	-1	-1

avg pool:
size 3x3, stride 2

-0.11	1.33	2.11
3.44	2.22	1.44
3.11	0.88	

- And continue.

Average Pooling

-5	-3	2	3	5	3	2	
-4	-1	-2	-3	4	1	3	
3	6	3	2	-2	-1	4	
2	5	4	2	6	-4	5	
1	3	4	-1	2	-3	6	
2	7	5	-2	-9	3	2	
5	-2	3	8	-2	-1	-1	

avg pool:
size 3x3, stride 2

-0.11	1.33	2.11
3.44	2.22	1.44
3.11	0.88	-0.33

- And now we've completed an average pooling operation with a size of 3x3 and a stride of 2.

Pooling

- The most common pooling sizes/strides are a pool size of 2x2 and a stride of 2, or a pool size of 3x3 and a stride of 2.
- But there are a number of other possibilities, especially when input images are larger. It is also possible to do pooling in 3 dimensions, however most CNNs keep the pooling operations within a channel. Your pools also do not need to be square!

Fractional Max Pooling

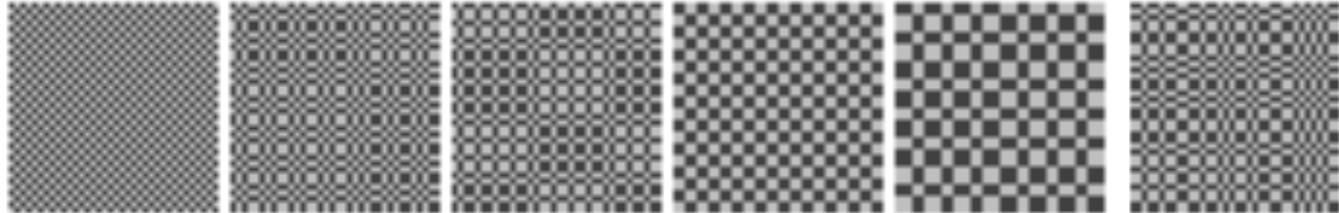


Figure 1: Left to right: A 36×36 square grid; disjoint pseudorandom FMP regions with $\alpha \in \{\sqrt[3]{2}, \sqrt{2}, 2, \sqrt{5}\}$; and disjoint random FMP regions for $\alpha = \sqrt{2}$. For $\alpha \in (1, 2)$ the rectangles have sides of length 1 or 2. For $\alpha \in (2, 3)$ the rectangles have sides of length 2 or 3. [Please zoom in if the images appear blurred.]

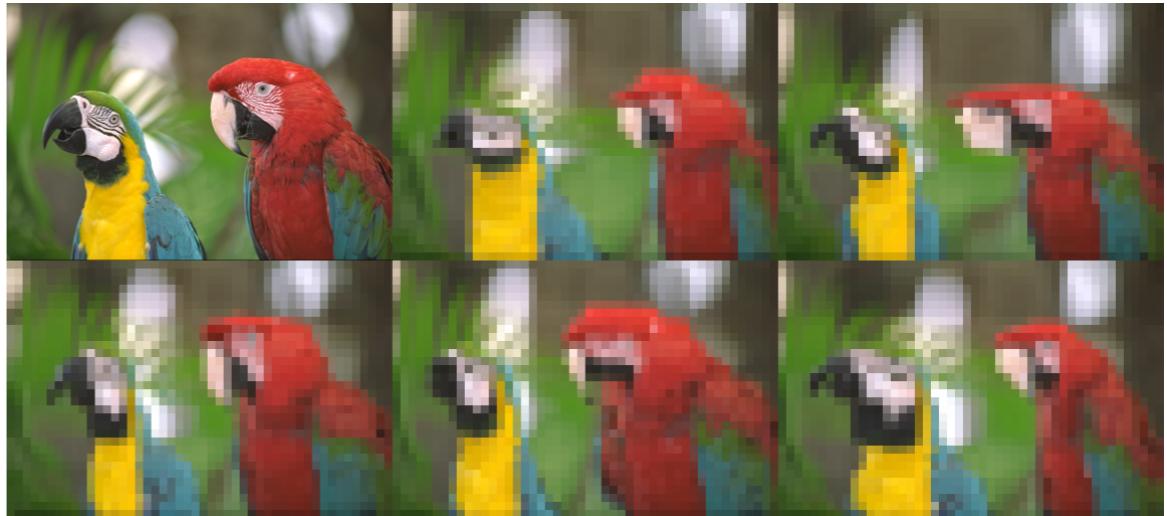


Figure 2: Top left, ‘Kodak True Color’ parrots at a resolution of 384×256 . The other five images are one-eighth of the resolution as a result of 6 layers of average pooling using disjoint random FMP $\sqrt{2}$ -pooling regions.

- There are even fancier versions of pooling, such as fractional max pooling [1], which uses different randomly selected pool sizes and strides for each forward pass which has shown strong results and can improve generalization (i.e., predictions on unseen images).

[1] <https://arxiv.org/pdf/1412.6071.pdf>

Convolutional Neural Network Architectures

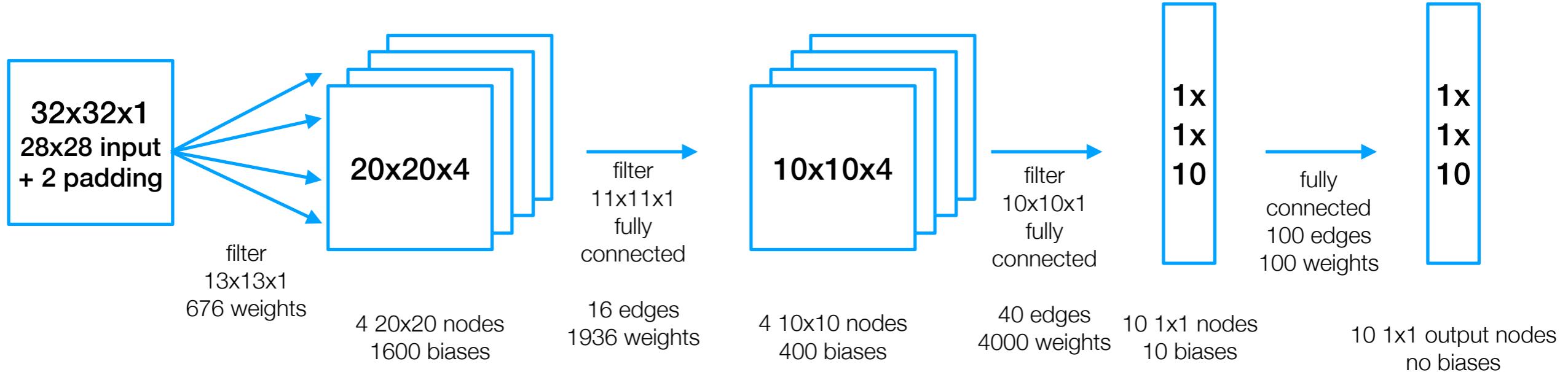
Designing a CNN

- The trick behind designing a CNN is to select a combination of an input node padding, and then filters, feature maps and pooling operations to eventually get down to $1 \times 1 \times 1$ (i.e., single value) nodes.
- These single value node layers are known as dense layers and typically there are a couple of them fully connected to the final output nodes.
- Most CNNs follow this general theme of design.

Simple Test CNNs

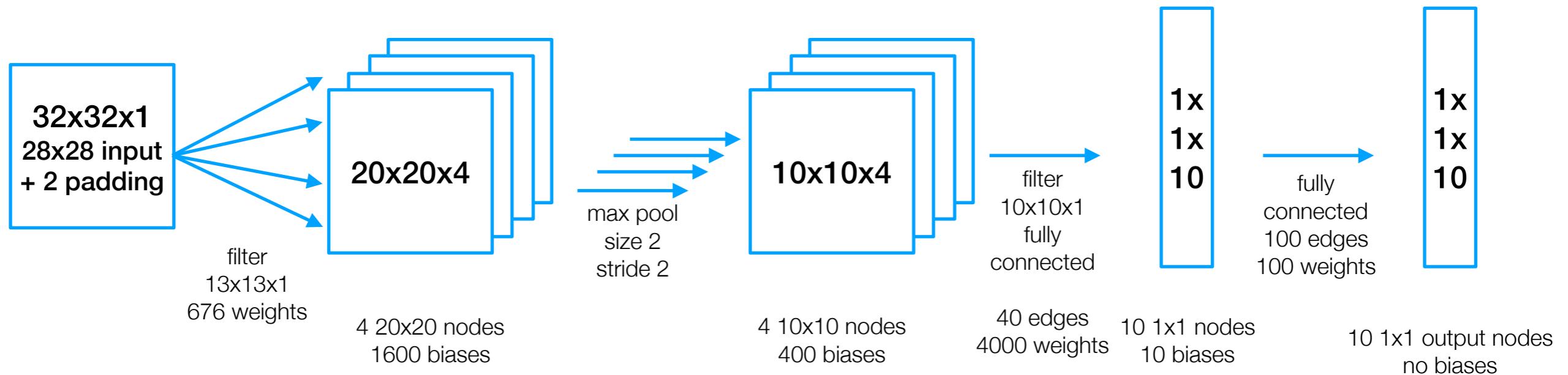
- For your programming assignments I made two different CNNs (one with pooling, the other without) so you can quickly test if your backpropagation is working correctly.
- These small networks were designed to keep the number of weights low so that calculating the numeric gradient didn't take forever so the tests could be completed faster. They won't actually work well.
- The following slides describe these "small" architectures.

Simple Test Architectures (Small, No Pooling - MNIST)



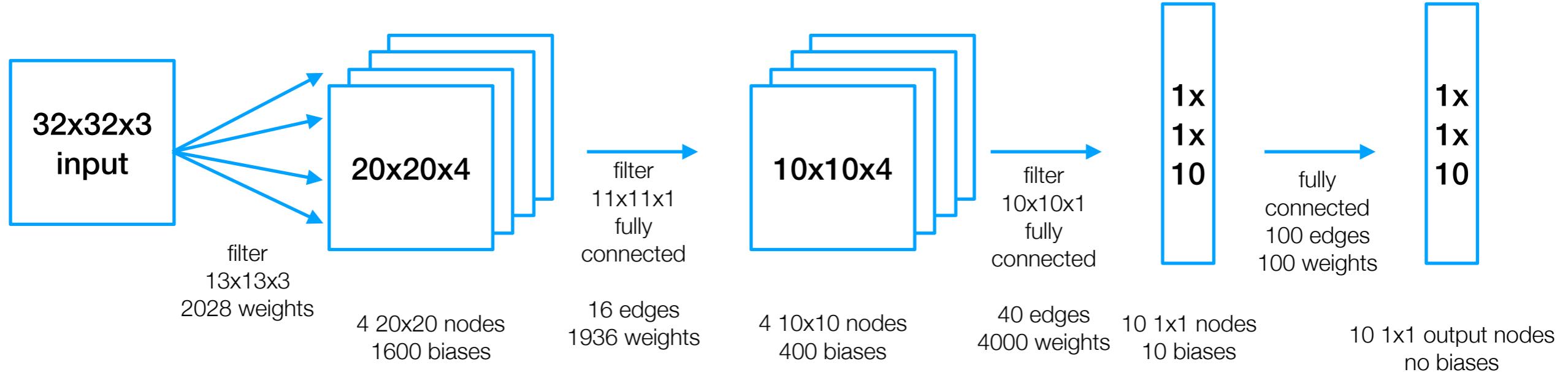
- It's common to see diagrams somewhat similar to these describing the architecture of a CNN.
- For this architecture, the MNIST data ($28 \times 28 \times 1$) is padded by 2 to make it ($32 \times 32 \times 1$). Then 4 13×13 filters are used as edges to 4 20×20 feature maps.
- The feature maps are fully connected to the next set of 4 10×10 feature maps with 16 11×11 filters.
- Then the 4 10×10 feature maps are fully connected to 10 1×1 simple neurons (i.e., the first dense layer). This makes 40 filters (4 inputs fully connected to 10 outputs).
- Then the first dense layer of 10 neurons is fully connected to 10 output nodes.
- In total this network has: 8722 weights

Simple Test Architectures (Small - MNIST)



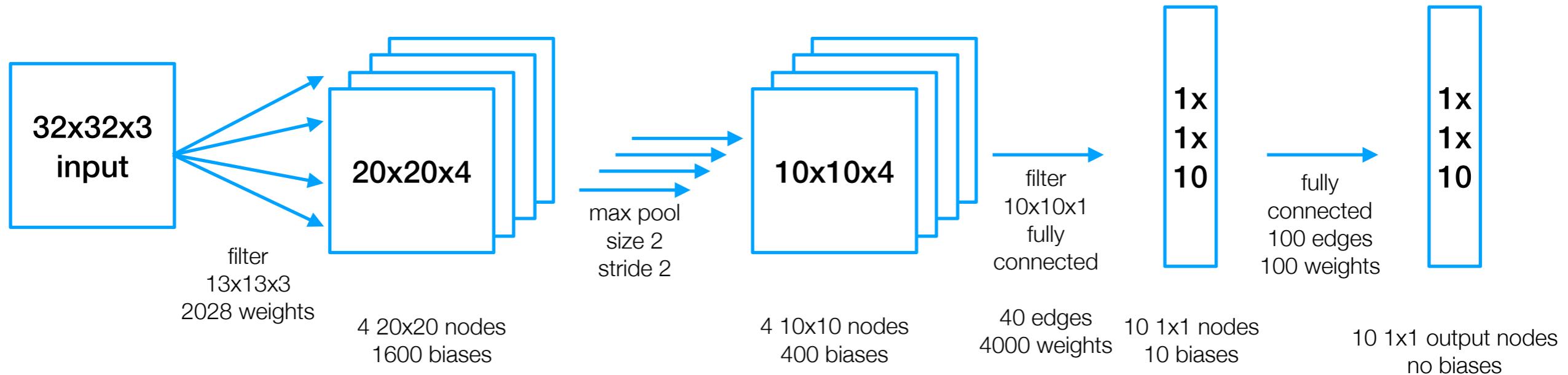
- This architecture is almost identical, except the filters between the second and third layer are replaced with a max pooling operation.
- For this architecture, the MNIST data (28x28x1) is padded by 2 to make it (32x32x1). Then 4 13x13 filters are used as edges to 4 20x20 feature maps.
- Each one of the 4 20x20 feature maps has a max pooling operation applied to it (pool size 2, stride 2) resulting in the 4 10x10 feature maps in the next layer.
- Then the 4 10x10 feature maps are fully connected to 10 1x1 simple neurons (i.e., the first dense layer). This makes 40 filters (4 inputs fully connected to 10 outputs).
- Then the first dense layer of 10 neurons is fully connected to 10 output nodes.
- total: 6786 weights

Simple Test Architectures (Small, No Pooling - CIFAR)



- Similar to the MNIST version this uses just a three dimensional filter to get to the same first layer, instead of a 13x13x1 filter it uses a 13x13x3 filter to get to 20x20x1 feature maps in the second layer.
- For this architecture, the CIFAR data (32x32x3) is not padded so it stays the same. Then 4 13x13x3 filters are used as edges to 4 20x20 feature maps.
- Each one of the 4 20x20 feature maps has 4 11x11 filter operations applied to it, fully connecting it to the next 4 10x10 feature maps in the next layer.
- Then the 4 10x10 feature maps are fully connected to 10 1x1 simple neurons (i.e., the first dense layer). This makes 40 filters (4 inputs fully connected to 10 outputs).
- Then the first dense layer of 10 neurons is fully connected to 10 output nodes.
- total: 10074 weights

Simple Test Architectures (Small - CIFAR)



- Similar to the MNIST version this uses just a three dimensional filter to get to the same first layer, instead of a $13 \times 13 \times 1$ filter it uses a $13 \times 13 \times 3$ filter to get to $20 \times 20 \times 1$ feature maps in the second layer.
- For this architecture, the CIFAR data ($32 \times 32 \times 3$) is not padded to it stays the same. Then 4 $13 \times 13 \times 3$ filters are used as edges to 4 20×20 feature maps.
- Each one of the 4 20×20 feature maps has a max pooling operation applied to it (pool size 2, stride 2) resulting in the 4 10×10 feature maps in the next layer.
- Then the 4 10×10 feature maps are fully connected to 10 1×1 simple neurons (i.e., the first dense layer). This makes 40 filters (4 inputs fully connected to 10 outputs).
- Then the first dense layer of 10 neurons is fully connected to 10 output nodes.

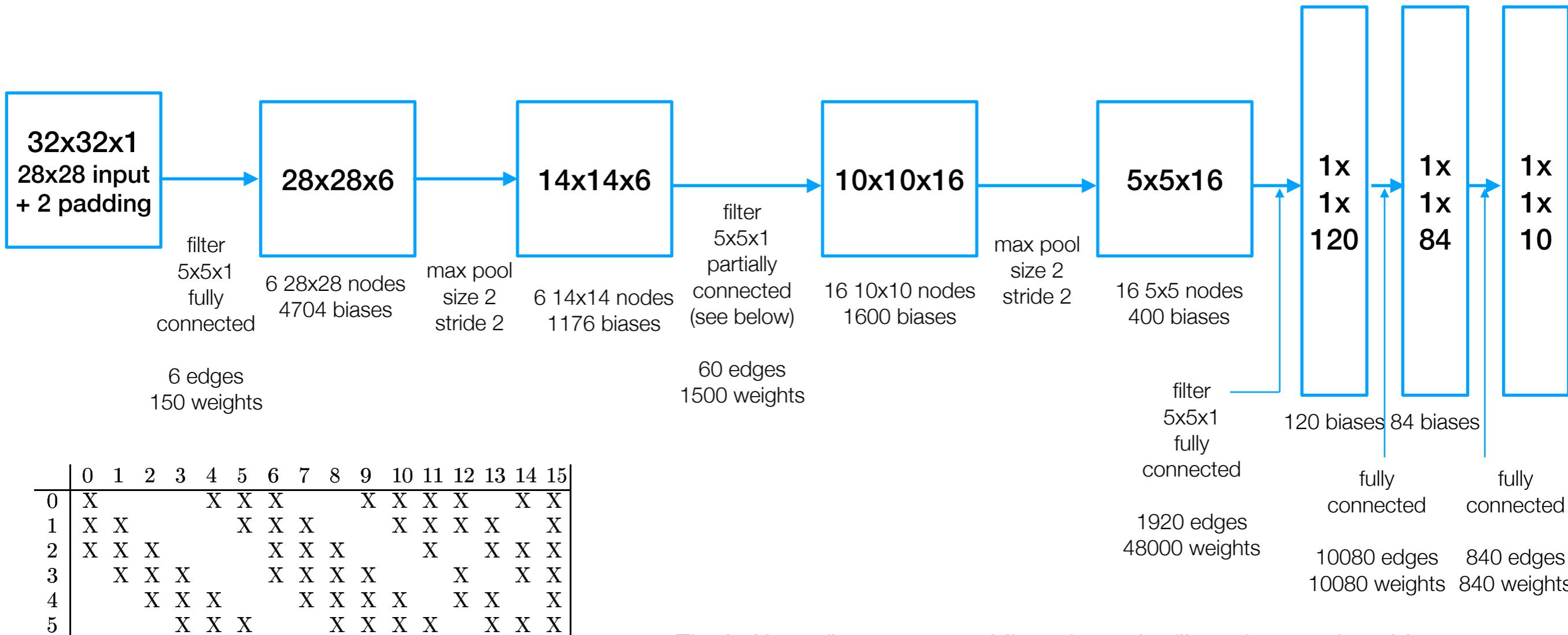
total: 8138 weights

LeNet-5

- Now we can get to our first "real" CNN architecture. The LeNet-5 was one of the first successful CNN architectures for MNIST [1] (with over 25k citations!).

[1] LeCun, Yann, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition." *Proceedings of the IEEE* 86, no. 11 (1998): 2278-2324.

LeNet-5



Connecting the $14 \times 14 \times 6$ layer to the $10 \times 10 \times 16$ layer is the most complex part of the LeNet-5. The y-axis (0-5) is the input feature map, and the x-axis (0-15) is the output feature map. If there is an X then there is a 5×5 filter between the two.

- The LeNet-5 first uses a padding of 2 and a filter of 5 to reduce bias towards non-edge pixels in the image (note how this brings the 2nd layer back to a 28×28 feature map, the same size as the inputs).
- Then it uses max pooling and 5×5 filters to bring things down to the first dense layer of 120 nodes.
- This is fully connected to a further dense layer of 84 nodes is then fully connected to the output layer of 10 nodes.
- In total this network has: 68654 weights

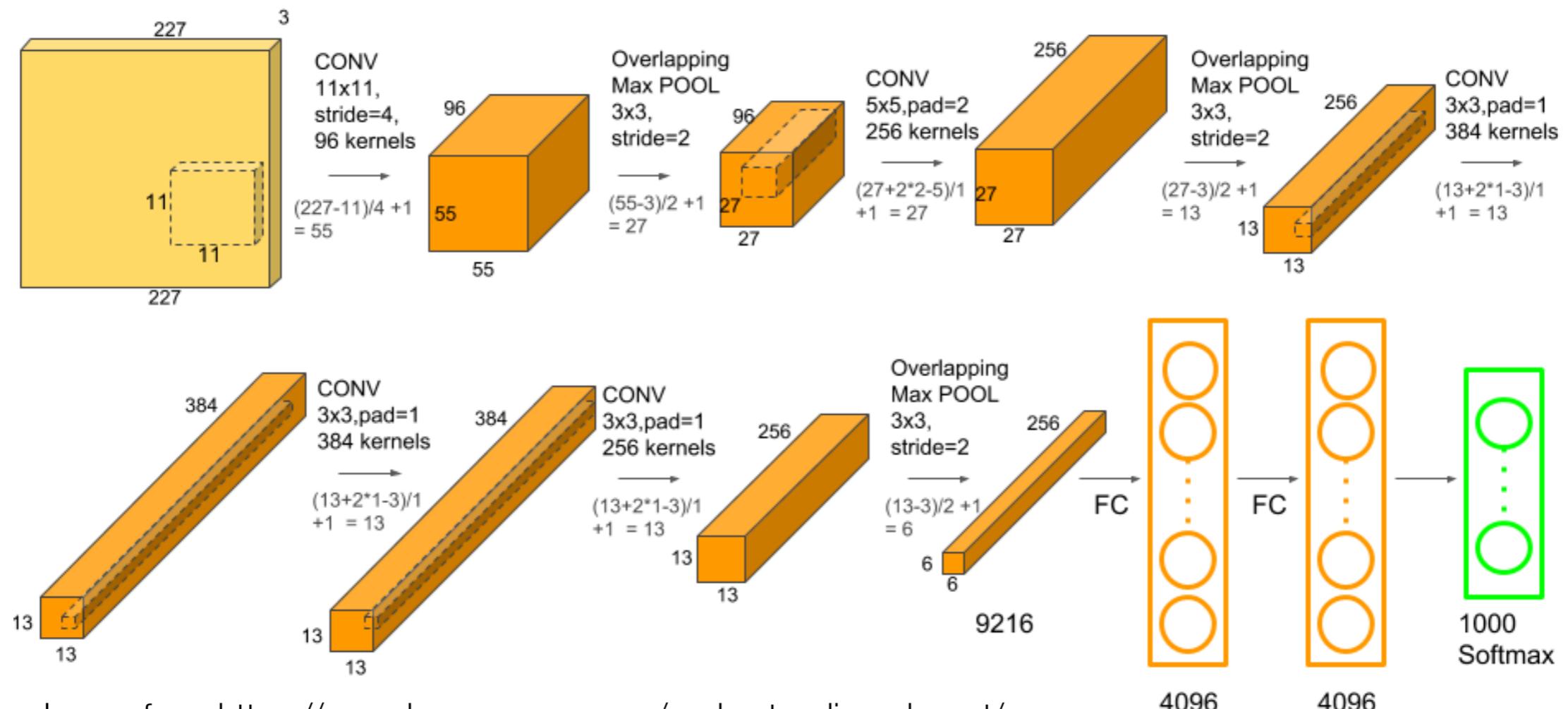
AlexNet

- In 2010, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) was launched. Much more challenging than CIFAR-10 or CIFAR-100 it has 1000 target classes, and 1 million hand annotated images. The full ImageNet dataset has 20,000 classes and over 14 million images [3].
- Alex Krizhevsky (from Hinton's lab) came up with AlexNet which achieve a top-5 error rate (how often the target wasn't in the top 5 outputs) of 15.3% of this new challenging dataset, where the 2nd best result in the competition was at 26.2% (very far behind).
- This (huge) CNN was a game changer and changed the deep learning landscape.

[2] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." In Advances in neural information processing systems, pp. 1097-1105. 2012. <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>

[3] <https://machinelearningmastery.com/introduction-to-the-imagenet-large-scale-visual-recognition-challenge-ilsvrc/>

AlexNet



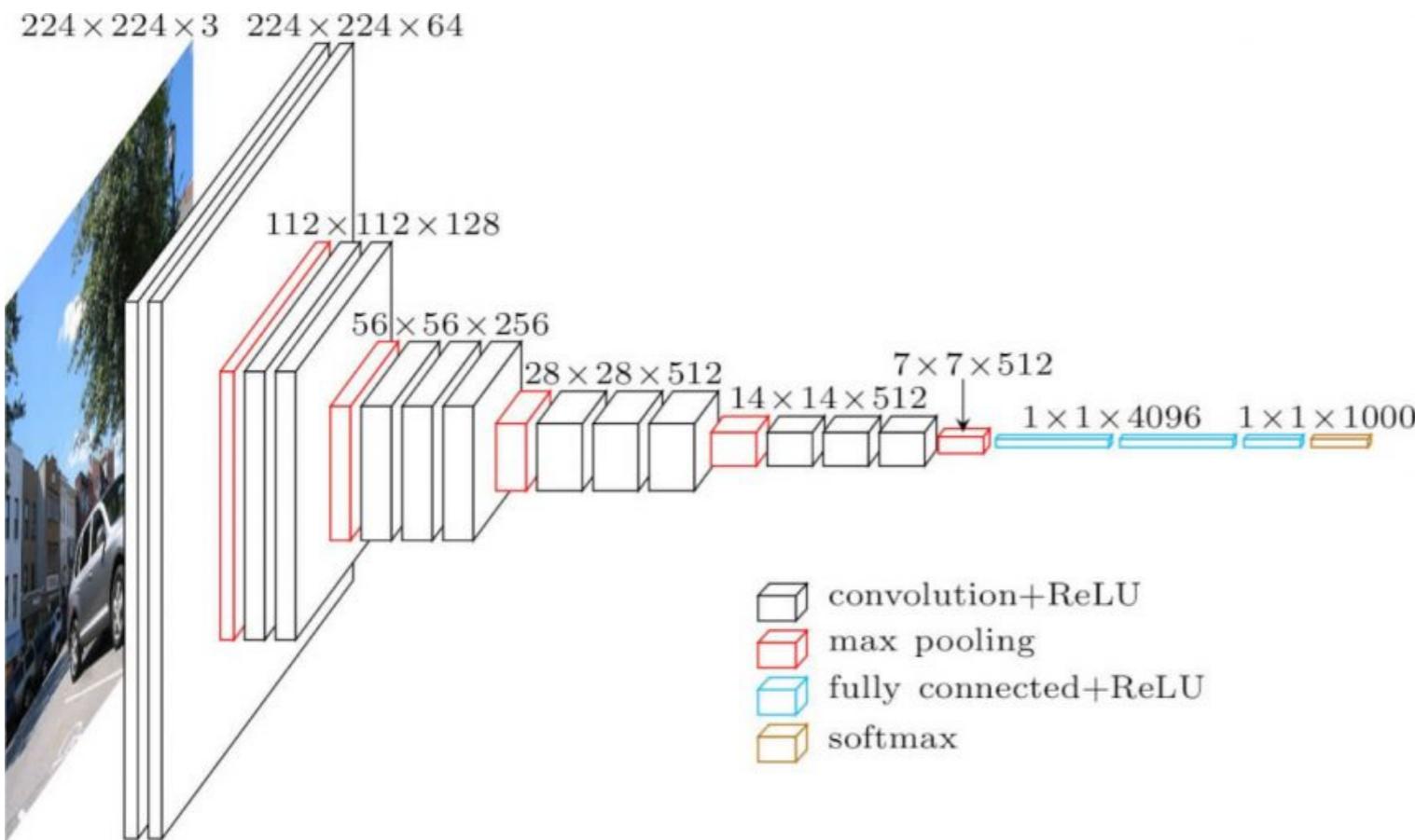
- The images from ImageNet are varying sizes. The first step of AlexNet is to resize the image down to a 256x256x3 image.
- They used data augmentation (more on that later), by using mirrored images and random subsamples. After the resize, inputs to the CNN were randomly selected 227x227x3 regions within the resized image (effectively "upsampling" by providing variations on the input image).
- After that it follows the above layers (including 3x3 pooling with stride 2 layers, which were novel at the time). It also used ReLU activation functions, a similarly novel concept at the time (and showed that using ReLU over tanh achieved a 25% error rate 6 times faster on CIFAR-10). It also used Dropout (more on that later) to prevent overfitting.
- At the time this took 5-6 days to train using two GTX 580 3GB GPUs.
- In total this network has: 650,000 neurons and 60 million weights! This was a *huge* network at the time. Just compare to LeNet-5.

VGG (Visual Geometry Group) Net

- In 2014, VGGNet [4] was the runner up to that years ILSVRC, achieving a top-5 error rate of 8.8 for VGG16 and 9.0 for VGG19. It notably used weights from smaller VGG versions to as pre-trained values to initialize the larger VGGNets to speed their convergence.
- VGGNet has a number of variants (VGG16 and VGG19 are popular), and is a conceptually appealing architecture which can be adapted to different input sizes.
- VGGNets are huge: VGG16 has 138 million weights and VGG19 has 144 million weights.

[4] Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556. <https://arxiv.org/pdf/1409.1556.pdf>

VGG16

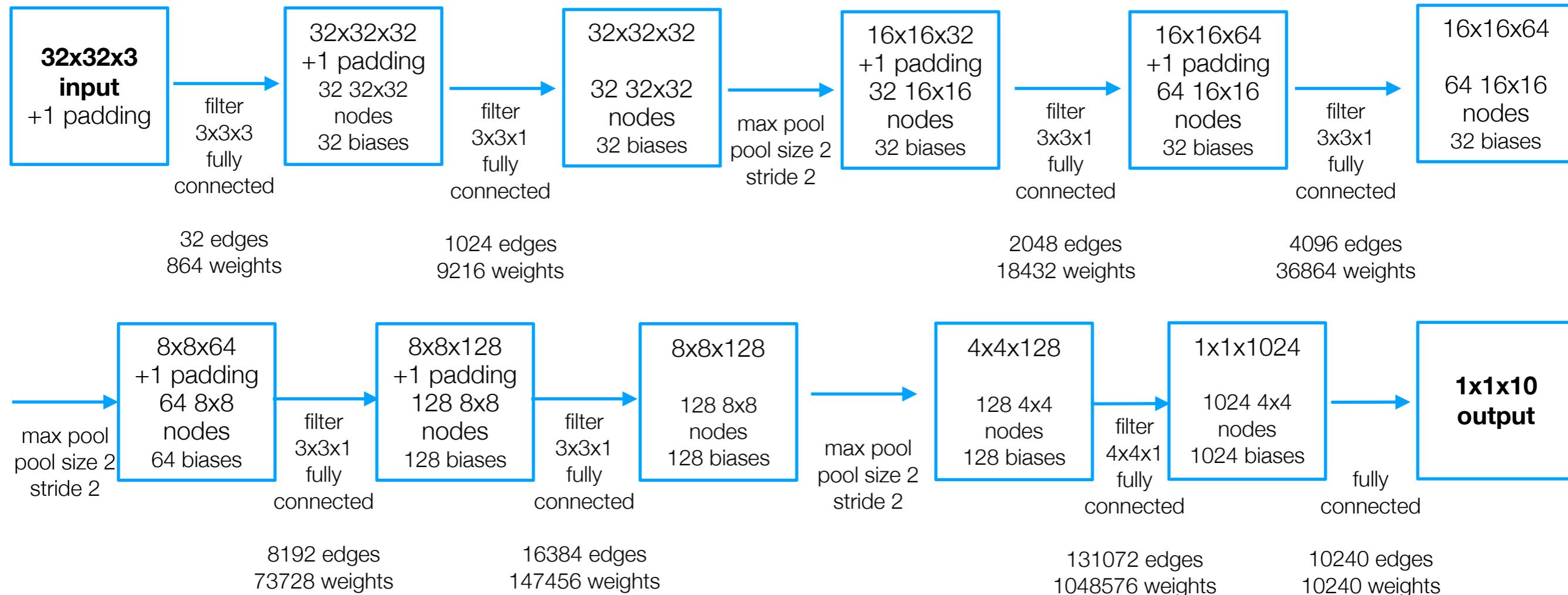


Images from <https://neurohive.io/en/popular-networks/vgg16/>

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
FC-4096	FC-4096	FC-4096	FC-4096	FC-4096	FC-4096
FC-1000					
soft-max					

- Top left is a visualization of VGG16, and the top right shows various configurations for VGGNets (VGG16 is column D, VGG19 is column E).
- It pads input layers by 1 and fully connects between layers with a 3x3 filter (which results in the next layer having the same dimension). This is how it does progressive layers of filters.
- It does 3 layers of 3x3 filters (padding the input by 1) followed by 2x2 max pooling with a stride of two, after max pooling the number of output feature maps doubles. It does this progressively until it has 512 7x7 feature maps, which are then fully connected to a 4096 node dense layer.
- This fully connects to another 4096 dense layer, which fully connects to the 1000 node softmax output layer.
- Note: VGG usually uses tied biases.

Small VGGNet for CIFAR-10



- Lets build a small VGG for CIFAR-10 with tied baises.
- We start with a much smaller input, and we can make the number of layers and feature maps a bit less (32, 64 and 128 vs. 64, 128, 256 and 512).
- The same concept applies though, 3x3 filters
- Total weights: 2,385,280