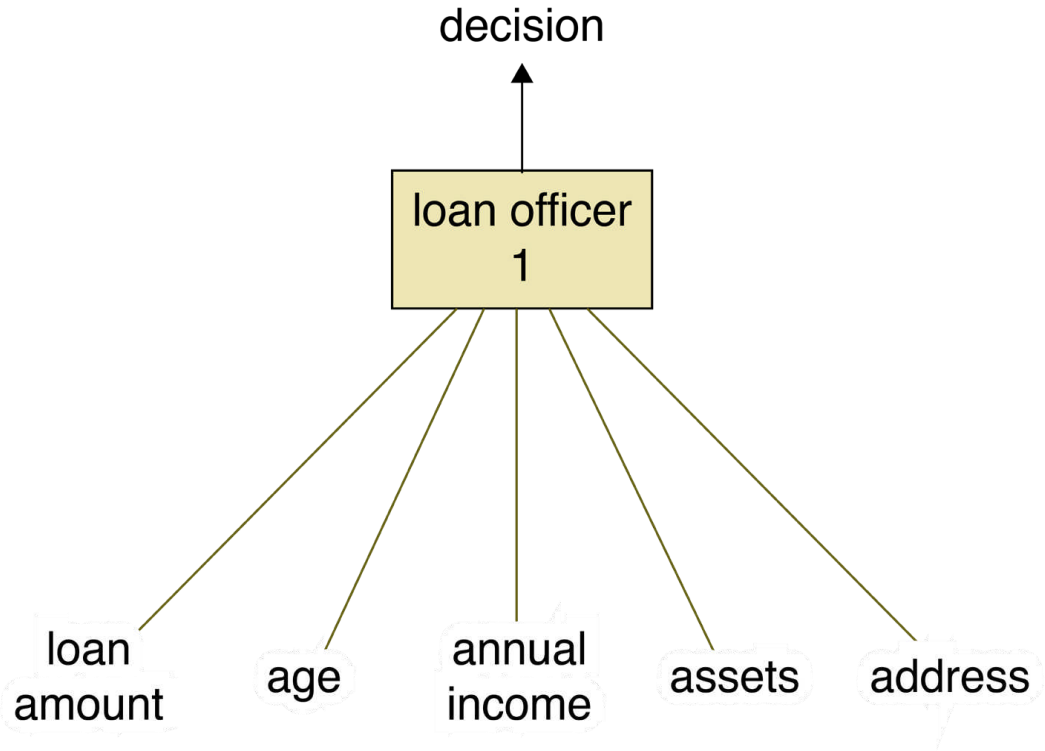
ANN- Neurons

Feedforward Nets, Activation Functions,

I/O Layers

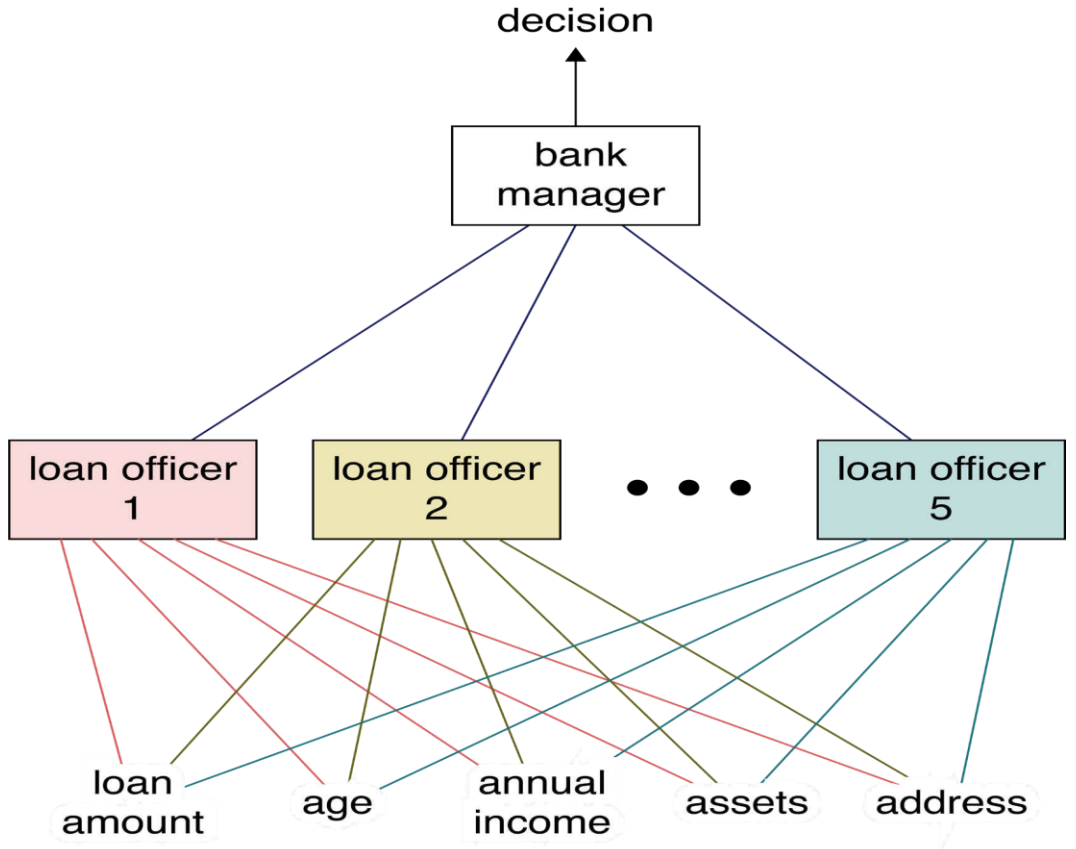
Bank Loan Example –Single layer Neural Network

- Let's consider the process of getting a loan.
- In practice, any loan application is going to be a complex affair.
- Based on the applications for those loans, they came up with rules that would let them predict whether a loan was likely to end up getting paid back or not.
- This is of course just like what a perceptron does. The inputs are weighted and combined to produce a final score, as in Figure.
- Say, loan is rejected



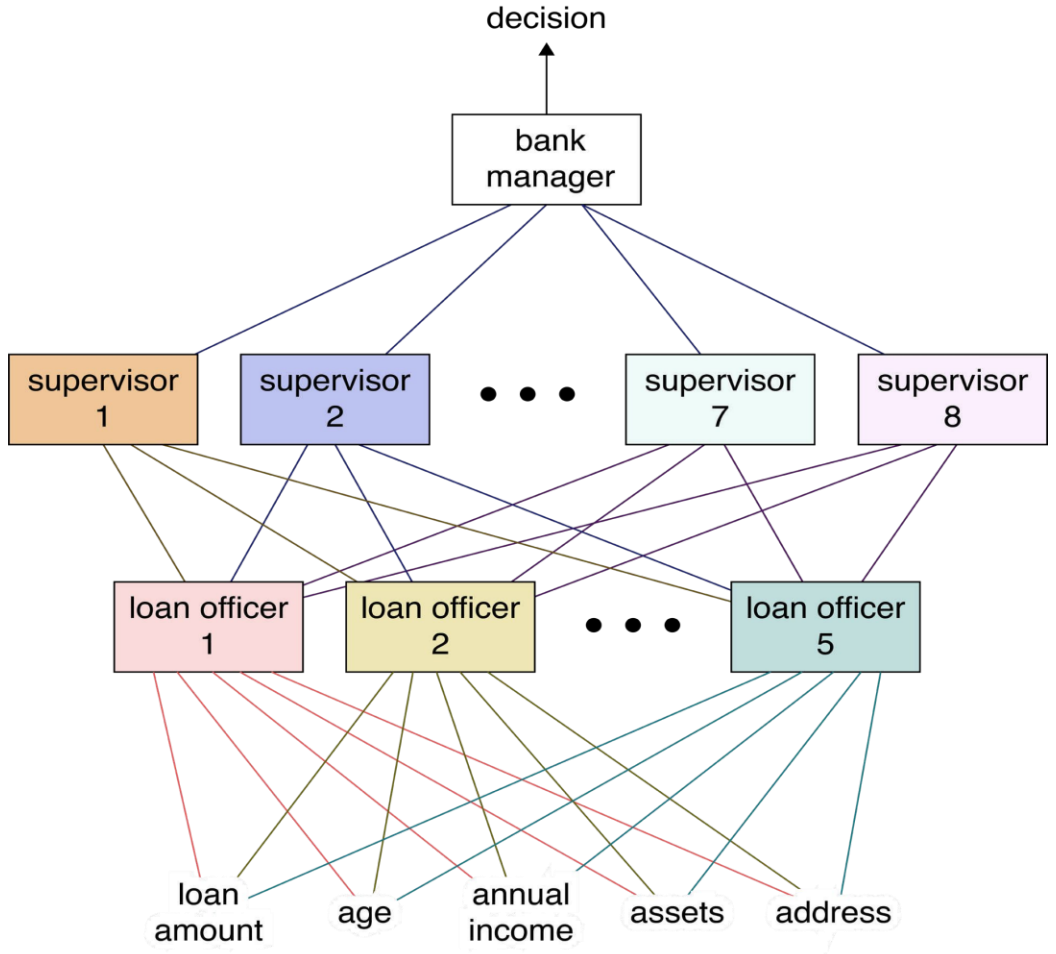
Bank Loan Example – 3 layer Neural Network

- We might try our luck at another bank.
- In this bank, suppose that there are 5 different loan officers, and each one has developed his own idiosyncratic procedure for evaluating the criteria that go into a loan.
- We can draw this as a two-layer neural network.



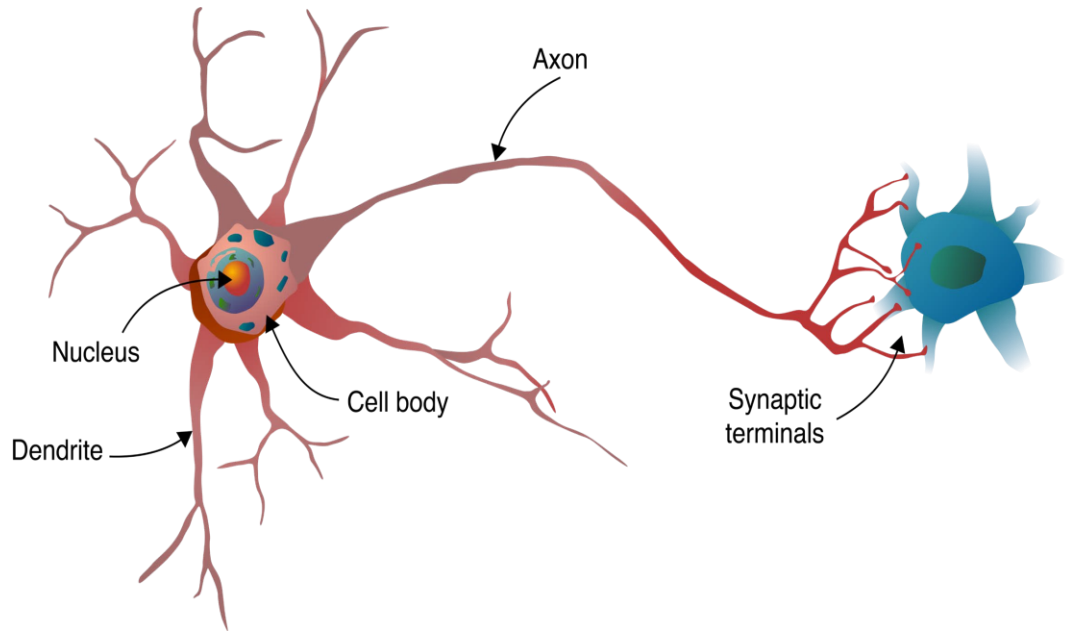
Bank Loan Example – 2 layer Neural Network

- Suppose bank hires a bunch of supervisors to sit between the loan officers and bank manager.
- Suppose we have 5 loan officers, 8 supervisors, and 1 bank manager.
- Then we have a 3-layer neural network to represent this process, as in Figure.
- Supervisors look at the decisions of the loan officers.
- Supervisors combine the results of the loan officers, and then pass their judgements up to the bank manager.
- Bank manager final decision is based on how he chooses to weight the conclusions of the supervisors.



Neurons

- How real biological neurons inspired the artificial neurons we use in machine learning, and how those little bits of processing work alone and in groups.
- Neurons are the nerve cells that make up the brain, used for our cognitive abilities.
- Neurons are information processing machines.
- A sketch of a biological neuron (in red) with a few major structures identified.
- This neuron's outputs are communicated to another neuron in blue), only partially shown.

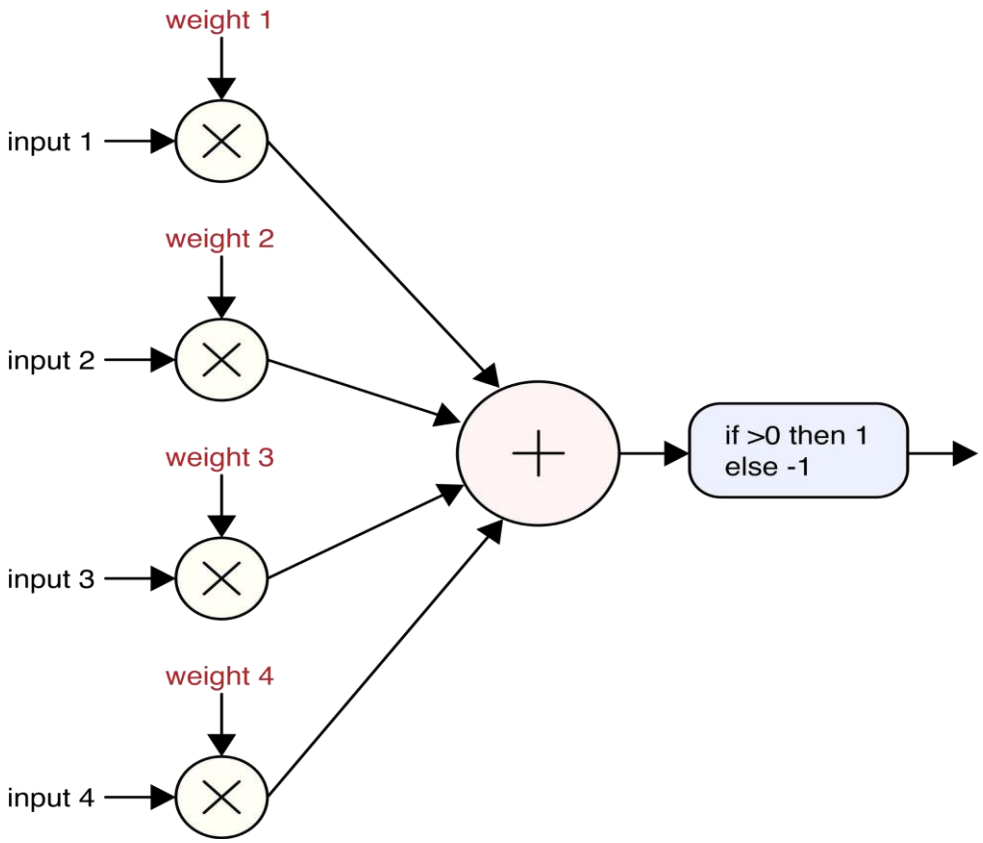


Neurons

- Basis of DL algorithms: Building a network of connected computational elements.
- Fundamental unit of such networks: a small bundle of computation called an artificial neuron, or simply neuron.
- Artificial neuron was inspired by human neurons.
- Important note: Not all machine learning algorithms use neurons
- Many ML techniques use traditional analysis and explicit equations, to find a solution. No neurons are used.
- Neurons are in more general learning systems, such as neural networks that make up deep learning systems.
- In those systems, artificial neurons are indispensable.
- We should know about these artificial neurons so we can use modern deep learning systems.

Perceptron

- The history of artificial neurons began in 1943.
- The "neurons" we use in machine learning are inspired by real neurons.
- In 1957, the perceptron was proposed as a simplified mathematical model of a neuron.
- Figure gives a block diagram of a single perceptron with 4 inputs.
- Perceptron mimics the functional behaviour of neuron.
- A schematic view of a perceptron. Each input is a single number, and it's multiplied by a corresponding real number called its weight.
- The results are all added together, and then tested against a threshold. If results are positive, perceptron outputs +1, else –1.



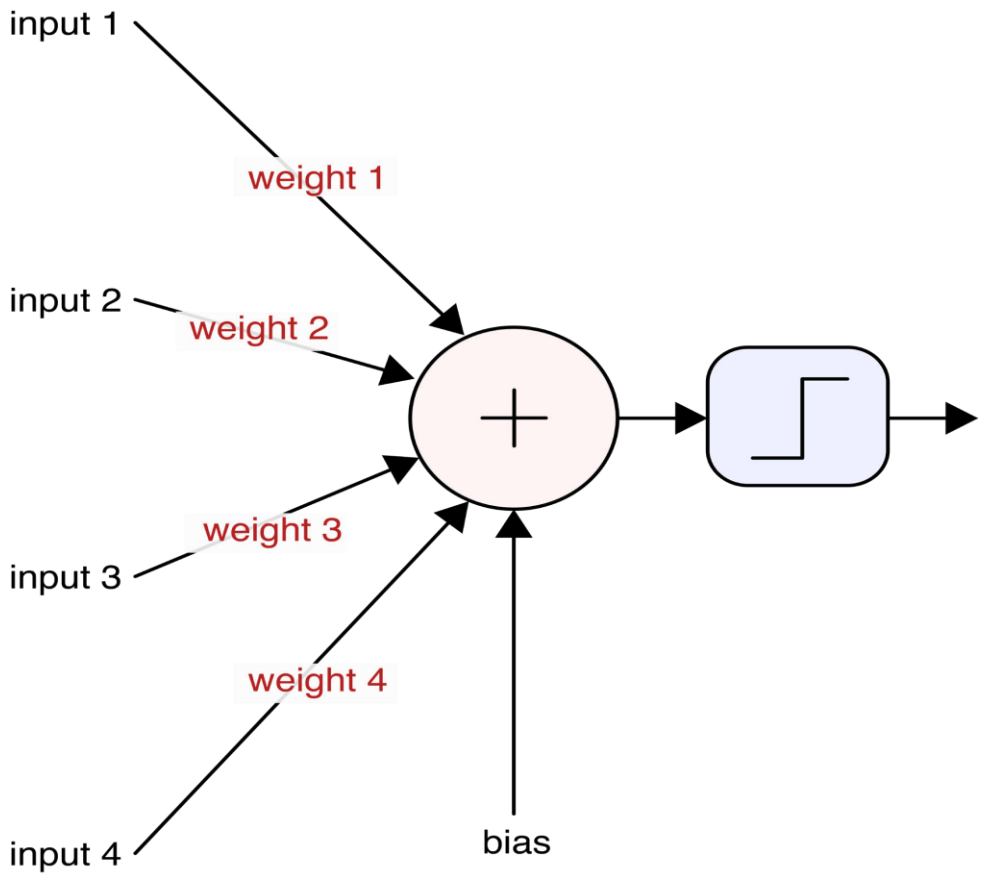
Modern Artificial Neurons

- **Modern Neurons** are only slightly generalized from the original perceptrons.
- Still called "**perceptrons**" or "**neurons**"
- Two changes to original perceptron: one at the input, and one at the output.
  1. Input - Provide each neuron with one more input, called bias. It's a number that is directly added into the sum of all the weighted inputs. Each neuron has its own bias.
  2. Output- Replace threshold with an activation function, i.e., a mathematical function that takes the sum (including the bias) as input and returns a new floating-point value as output.

Modern Artificial Neurons

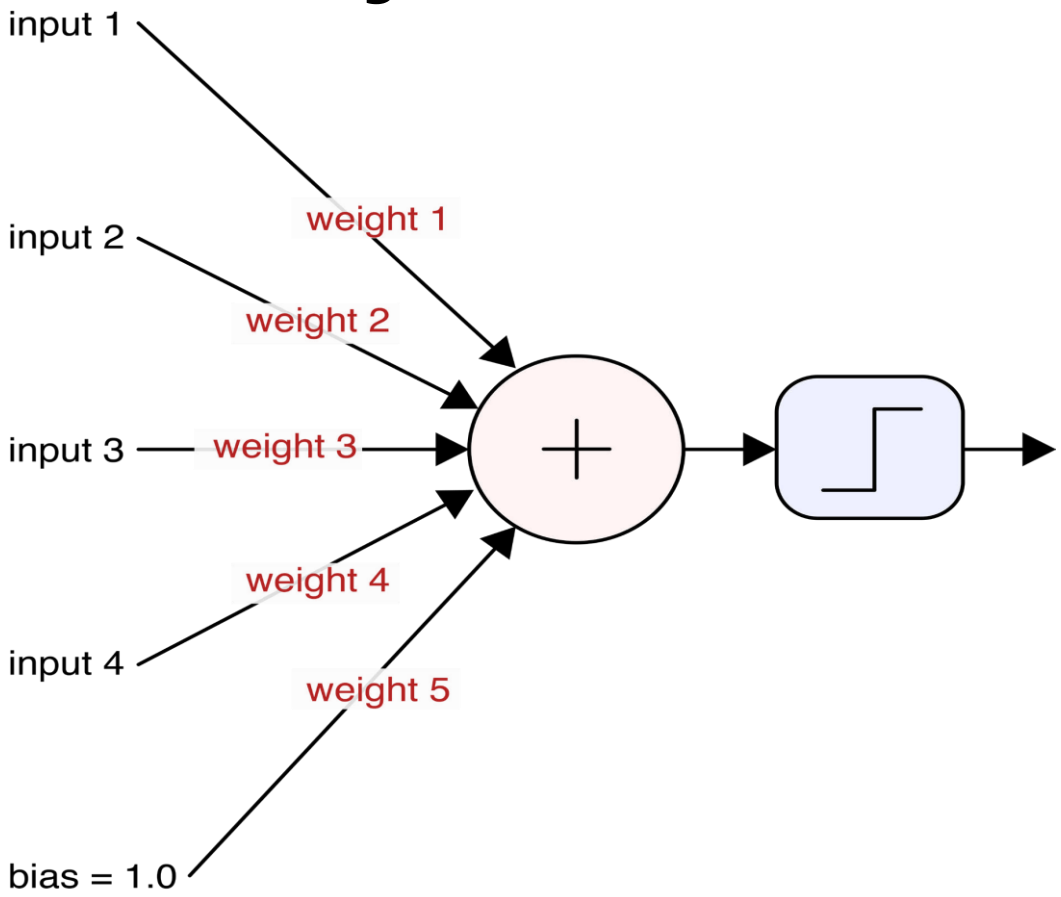
Note:

- In neural network diagrams, weights and the nodes where they multiply are not drawn
- Weights are always there and they always modifies the input . They're just not drawn.
- Output of the activation function might take on any value.
- Variety of activation functions each with its own pros and cons.
- We'll run through them later.
- A neuron is often drawn with the weights on the arrows. This "implicit multiplication" is common in machine learning figures. We've also replaced the threshold function with a step, to remind us that any activation function can follow the sum.



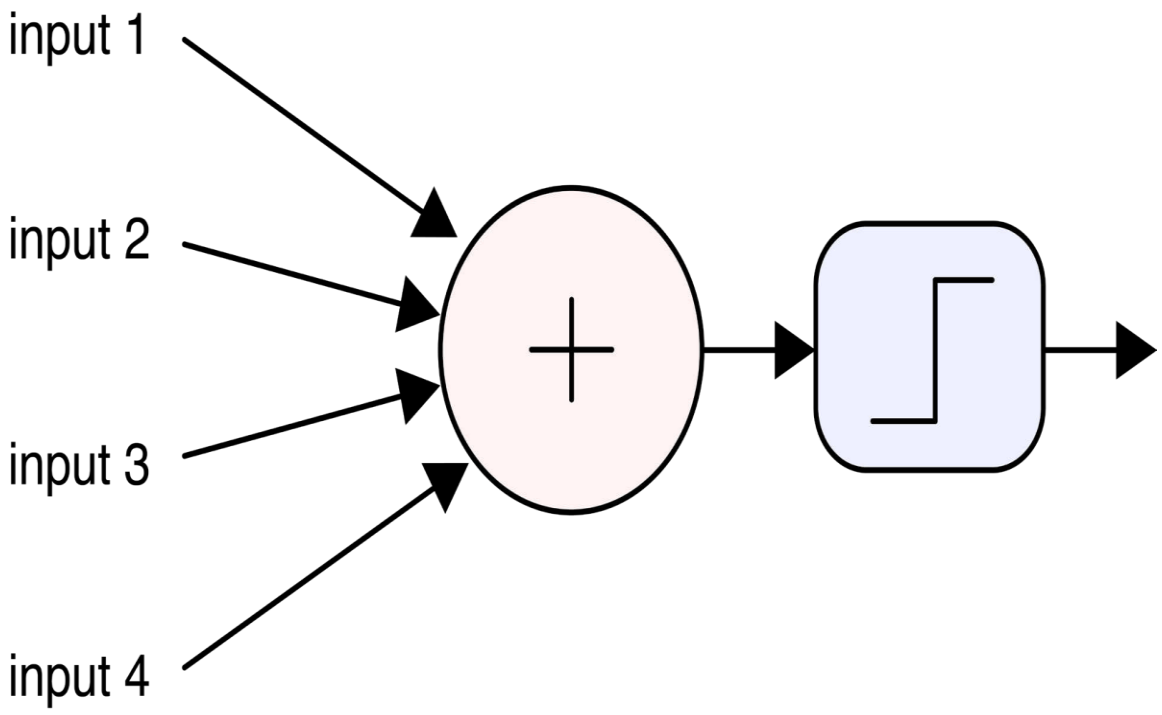
Bias

- The "bias trick" in action. Rather than show the bias term explicitly. We pretend it's another input with its own weight.



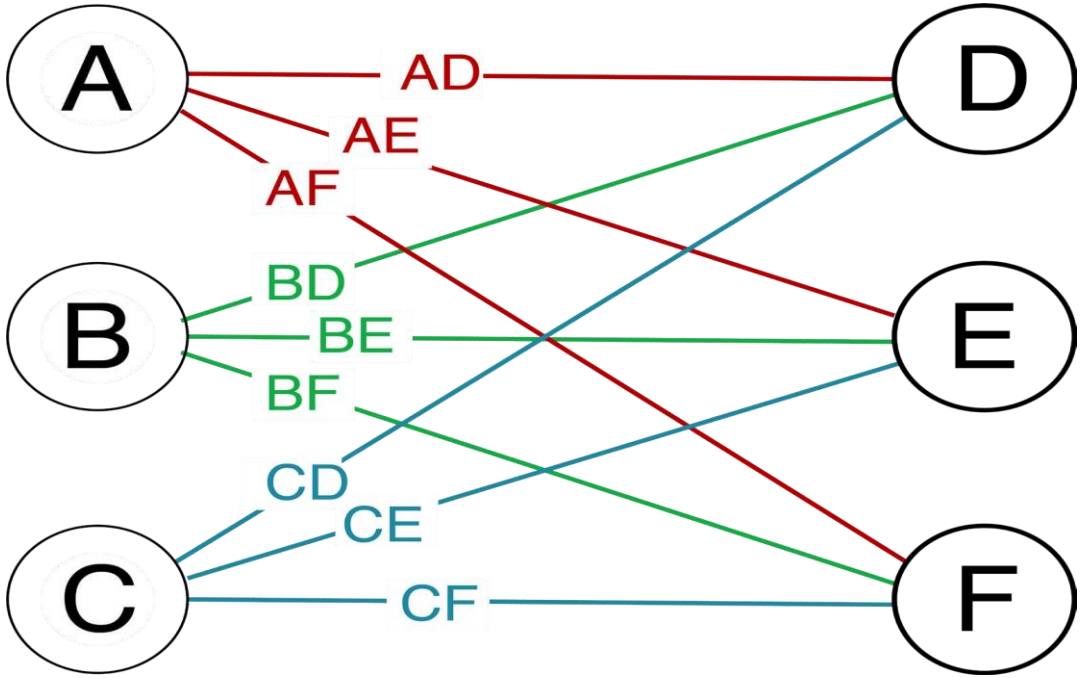
Typical Drawing of Artificial Neuron

- **The bias term and the weights are not shown, but the bias and weights are in there. We are supposed to add them back in mentally.**



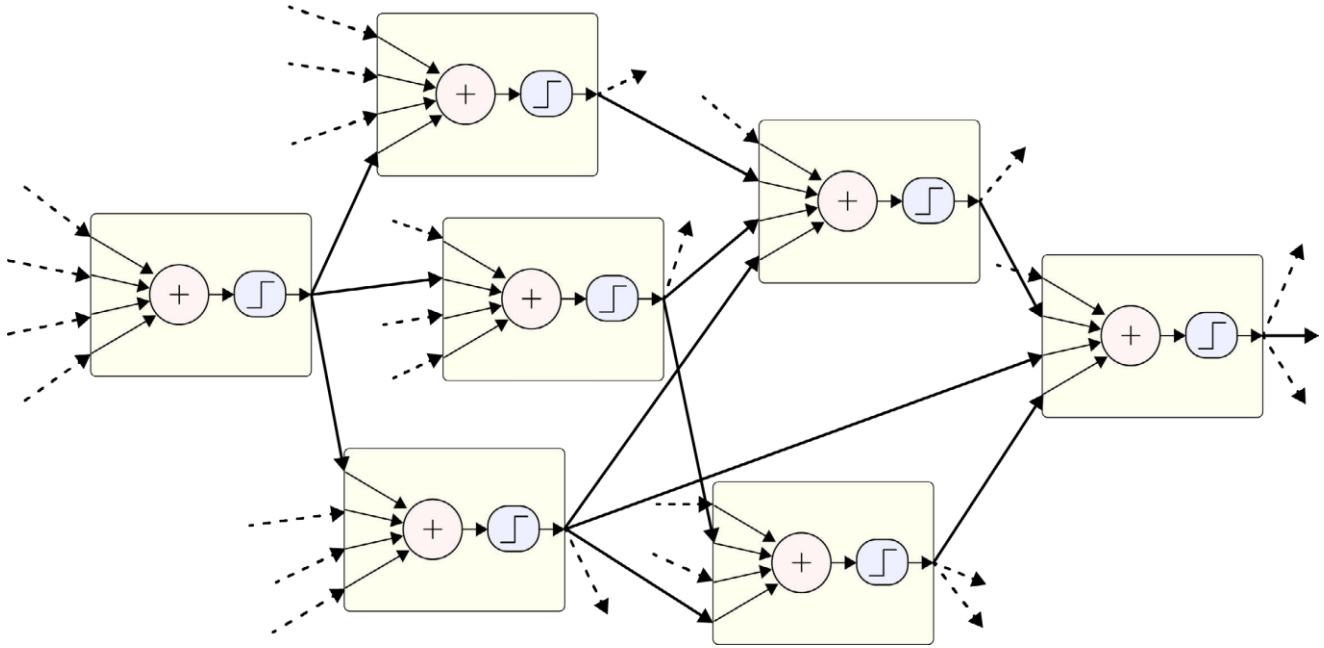
Typical Drawing of Artificial Neuron

- The weights are shown explicitly in this diagram. Following convention, each weight is given a two-letter name formed by combining the names of the neuron that produced the output on that weight's wire, with the name of the neuron that receives the value as input. For example, BF is the weight that multiplies the output of B for use by F



Larger Network of Artificial Neurons

- A piece of a larger network of artificial neurons. Each neuron receives its inputs from other neurons
- The dashed lines show connections coming from outside this little cluster.



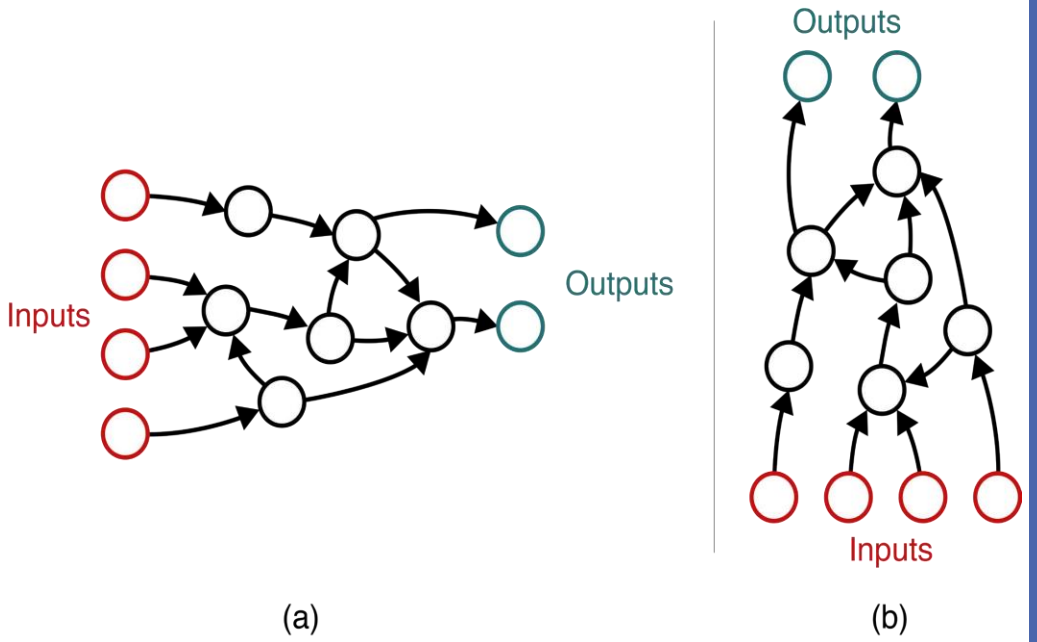
Feedforward Networks

- Deep learning systems are structured as networks of neurons.

- Much of their work is performed as data flows through them in a forward direction, starting at the inputs until it reaches the outputs.

Feedforward Networks

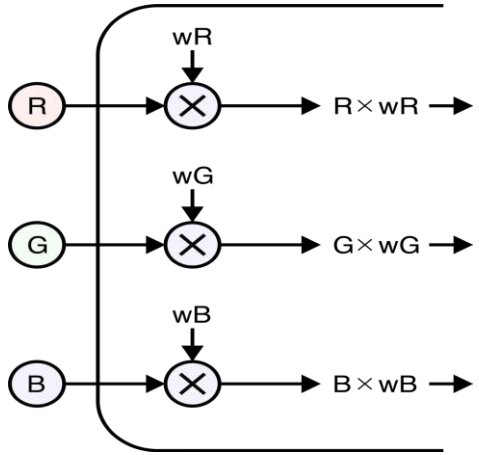
- This kind of graph is like a little factory.
- Raw materials come in one end, and pass through machines that manipulate and combine them, ultimately producing one or more finished products at the end.
- Two neural networks drawn as graphs.
- Data flows from node to node along the edges, following the arrows.
- When the edges are not labeled with an arrow, data usually flows left-to-right or bottom-to-top.
- No data ever returns to a node once it has left.
- In other words, information only flows forwards, and there are no loops.



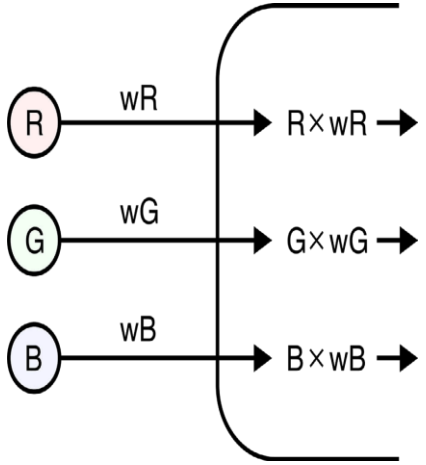
(a) Left-to-right flow. (b) Bottom-to-top flow.

Feedforward Networks

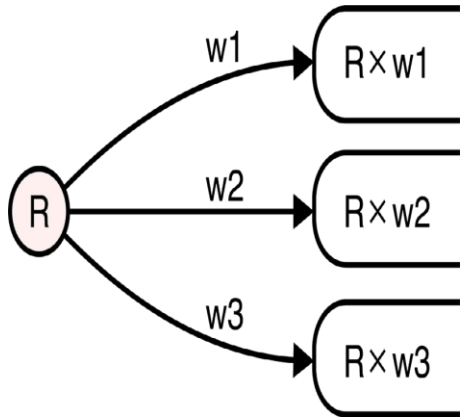
- Top: Three values, named R, G, and B, are heading into a neuron. As each value enters the neuron, it's weighted, or multiplied by a number associated with that input. The resulting scaled value is then used by the rest of the neuron to compute its output value.



- Bottom-Left: The weights are written on the wires carrying the values into the neuron.



- Bottom-Right: A single neuron, here labeled R, sends its output to three different neurons, then R's output value will be weighted by three potentially different values, one on each wire.



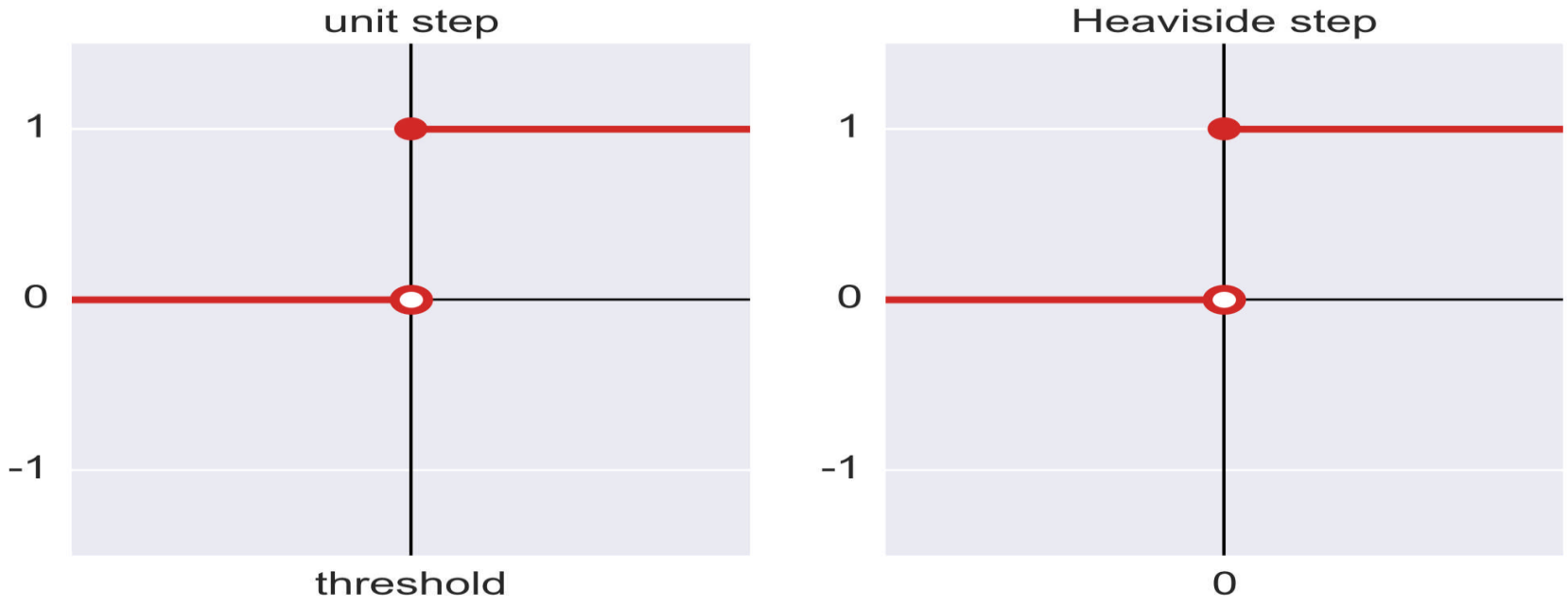
Activation Functions

- The last step in an artificial neuron is to apply an activation function to the value it computes.

- Here we'll see a variety of popular activation functions in use today.

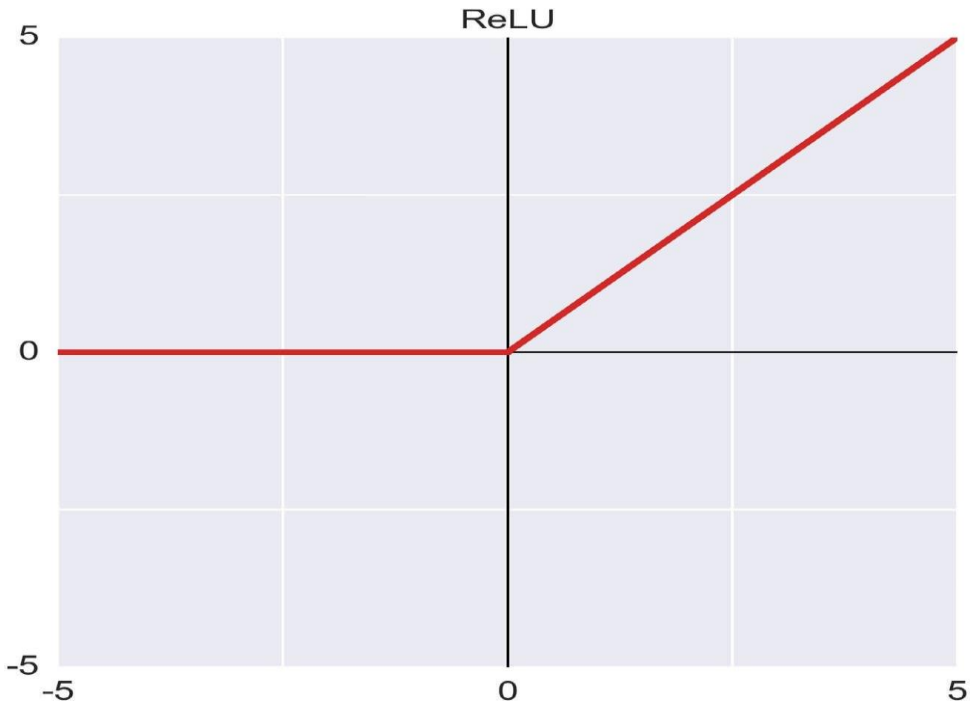
Activation function – Step

- A couple of popular step functions.
- Left: Unit step has a value of 0 to left of the threshold, and 1to the right.
- Right: The Heaviside step is a unit step where the threshold is 0.



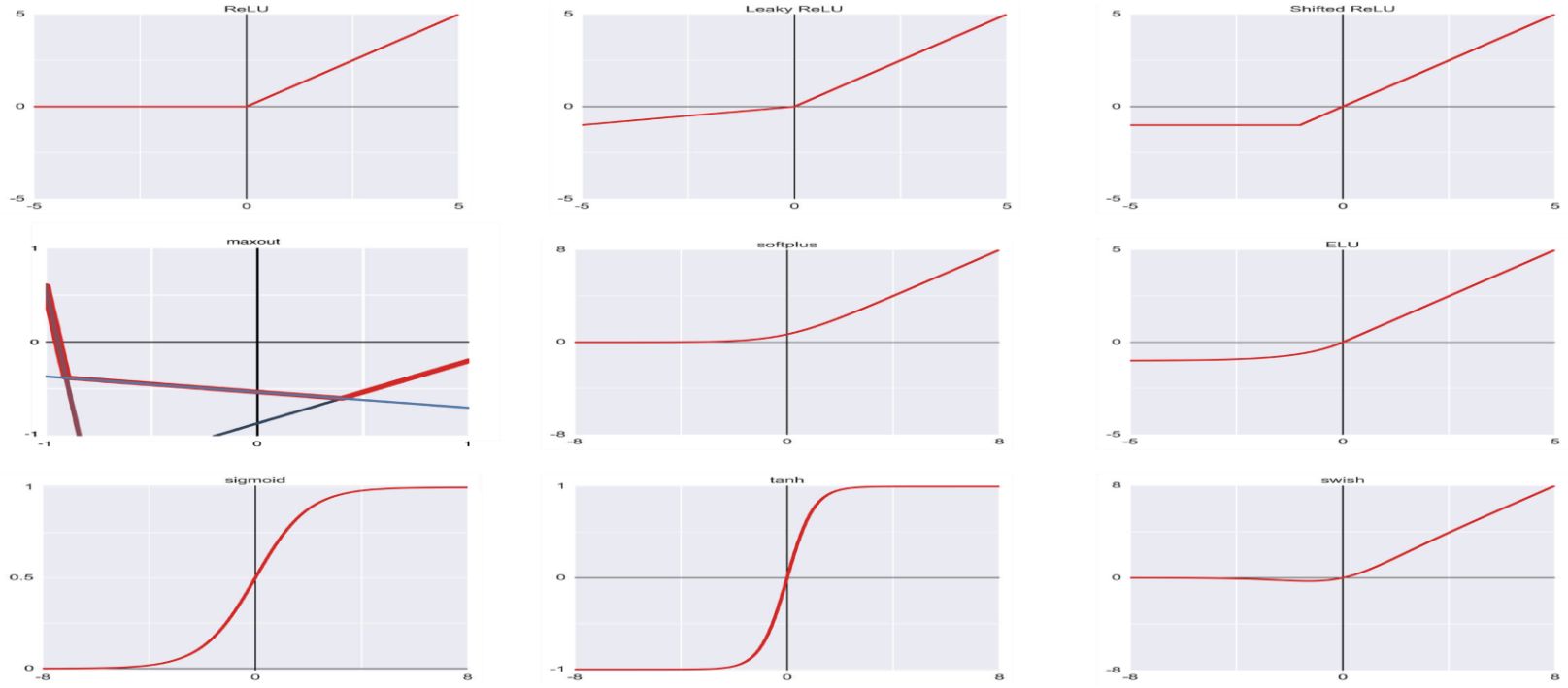
Activation function –ReLU

- ReLU is not a linear function.
- ReLU is popular because it's a simple and fast way to include a non-linearity in our artificial neurons.
- ReLU performs well and is often the first choice of activation function when building a new network.
- There are good mathematical reasons to use ReLU.
- The ReLU, or rectified linear unit.
- Output is o for all negative inputs. Else, output = input



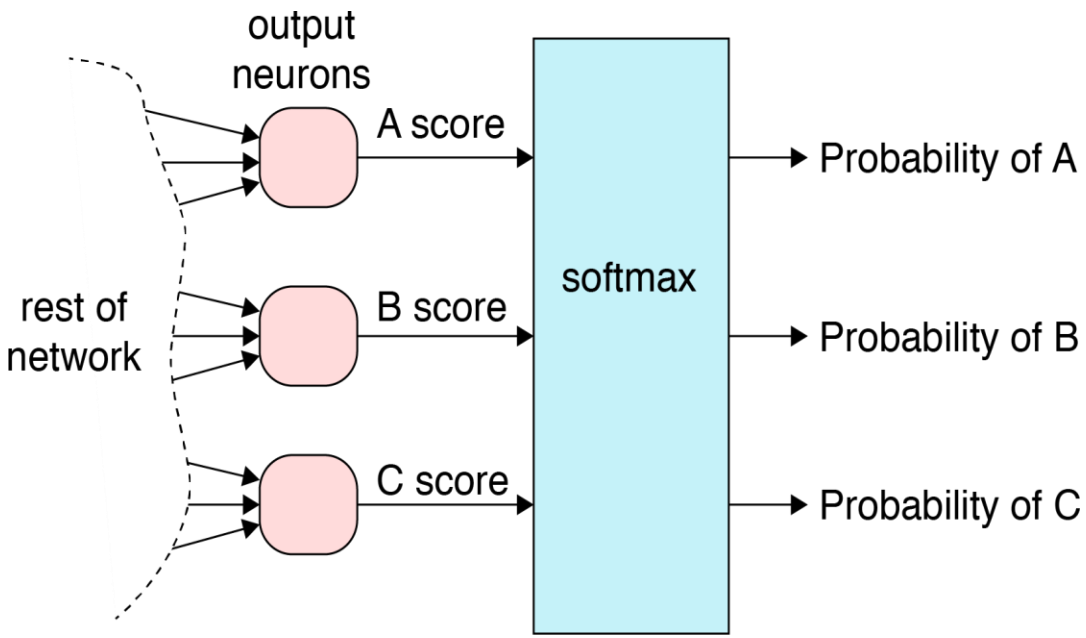
Activation Function Gallery

- A gallery of popular activation functions.
- Top row, left to right: ReLU, leaky ReLU, shifted ReLU.
- Middle row: maxout, softplus, ELU. Bottom row: sigmoid, tanh, swish.



Softmax Function

- There's another kind of operation that we typically apply only at the output neurons of a classifier neural network,
- Used only if there are 2 or more output neurons.
- It's not an activation function.
- Why? Because it applies simultaneously to all the output neurons, not just one.
- The technique is called softmax,
- Often used as a final step for classification networks with multiple outputs.
- Softmax turns numbers that come out of the network into probabilities of classes.
- The softmax function takes all the network's outputs and modifies them simultaneously.
- Result is that scores are turned into probabilities.

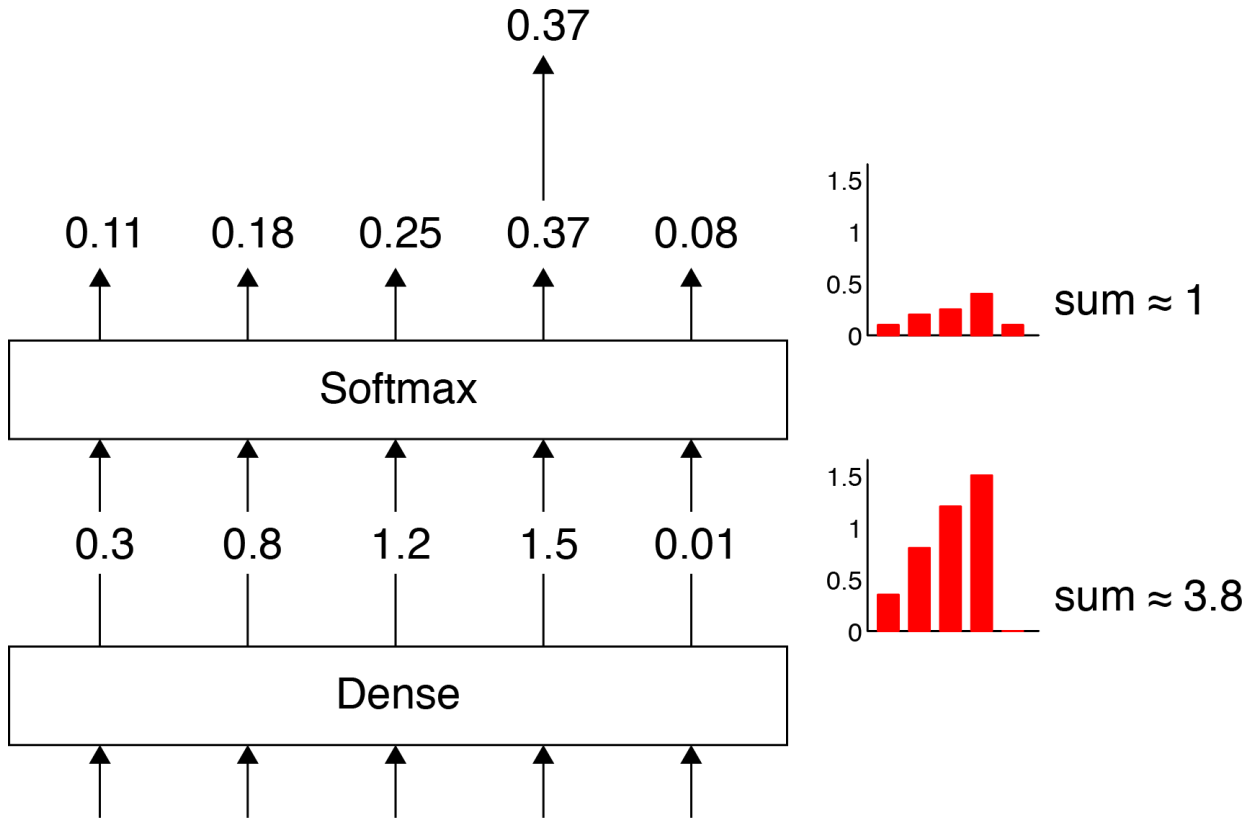


Softmax Function - Example

- Softmax turns our scores into probabilities.
- It's widely used at the end of neural networks that are used for classification.
- Three hypothetical networks, left, middle, and right.
- Bottom row shows outputs of softmax
- Top row: Scores from a classifier, which are inputs to Softmax.
- Bottom row: Results of running the scores in the top row through Softmax.

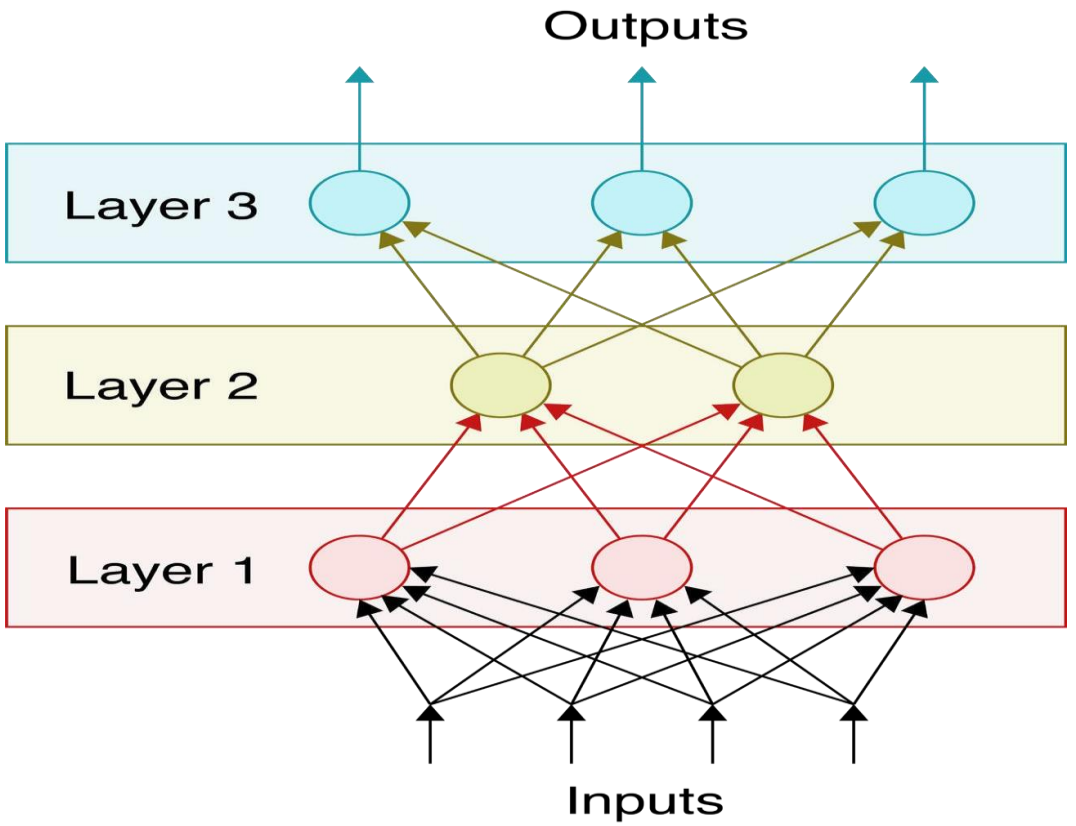
Softmax Function

- Softmax operation changes a list of numbers so that they represent probabilities. Uses a mathematical transformation.
  Result : Every output is from 0 to 1,and sum of all outputs =1.
- In this example, we have five values coming out of a fully-connected, or dense, layer.
- Values are between 0.01 and 1.5, and they sumup to about 3.8.
- After the softmax, the values are from 0 to 1 and add up to 1.



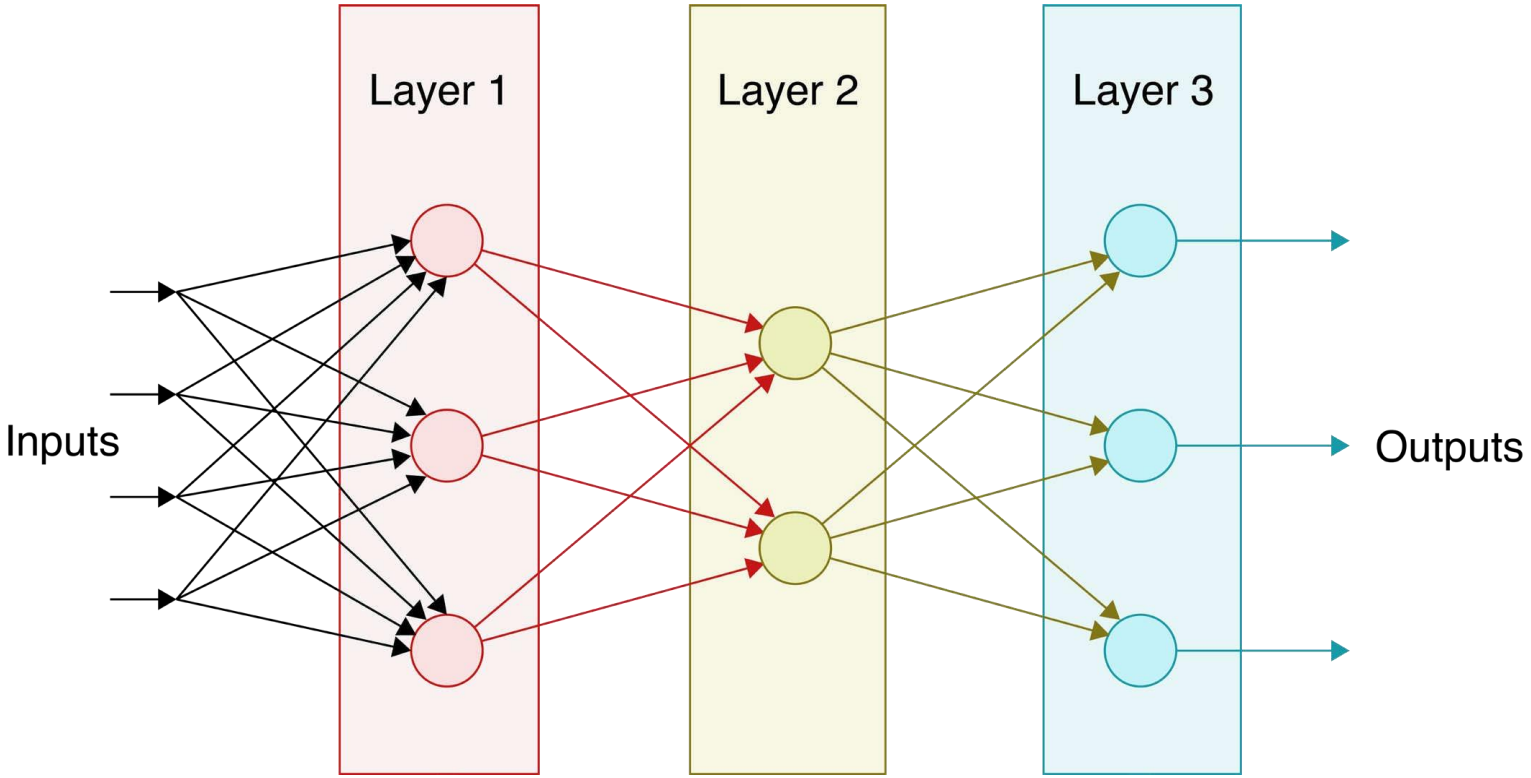
Layers: Input and Output

- A deep learning network, having 4 inputsflowing through 3 layers, creating 3 outputs at the end.
- Network is "fully-connected" - every neuron in each layer receives input from every neuron on the previous layer.
- Output layer: the topmost layer (layer 3 in Figure). It contains the final set of neurons.
- Input layer: It is NOT Layer 1. It is the row of black arrows at the bottom of Figure. The input layer refers to the memory that holds the input values.
- These labels simply refer to the position of the layer in the stack: the input is at the start (the bottom, or left) and the output is at the end (the top, or right).
- Someone looking at this network from above or below, can see only the input layer or the output layer. Most networks have a single input layer and a single output layer.



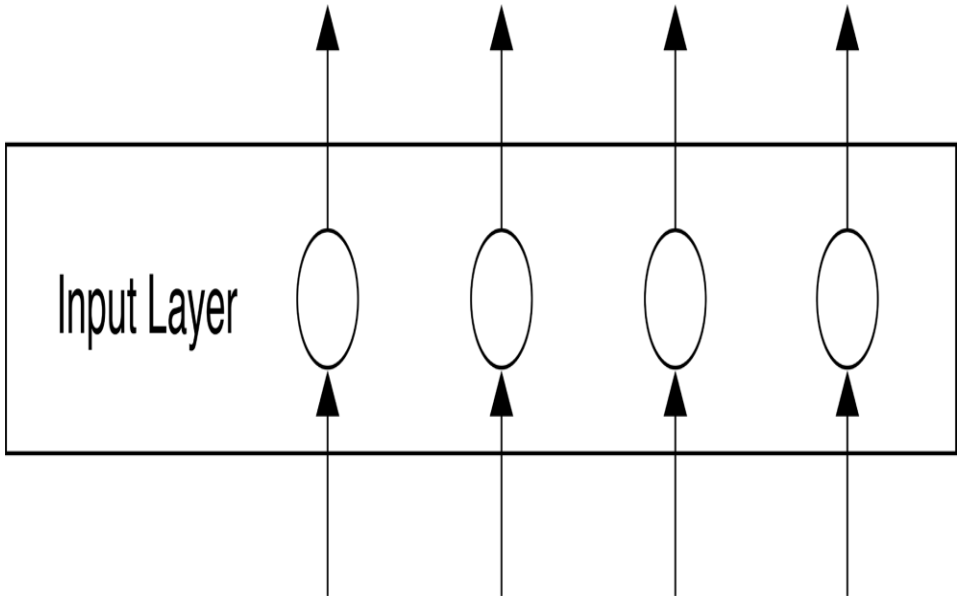
Layers: Input and Output

- The same deep network of Previous Figure, but drawn with data flowing left-to-right.



Input layer – some notes

- The input layer is usually not shown in deep-learning architecture diagrams.
- It is merely some memory that holds the input values.
- Note that these are not neurons, since the input layer does no processing.
- It can be thought of as simply as a collection of chunks of memory, each capable of holding one number of the input.
- The input layer is just a placeholder where we can temporarily store the input data.

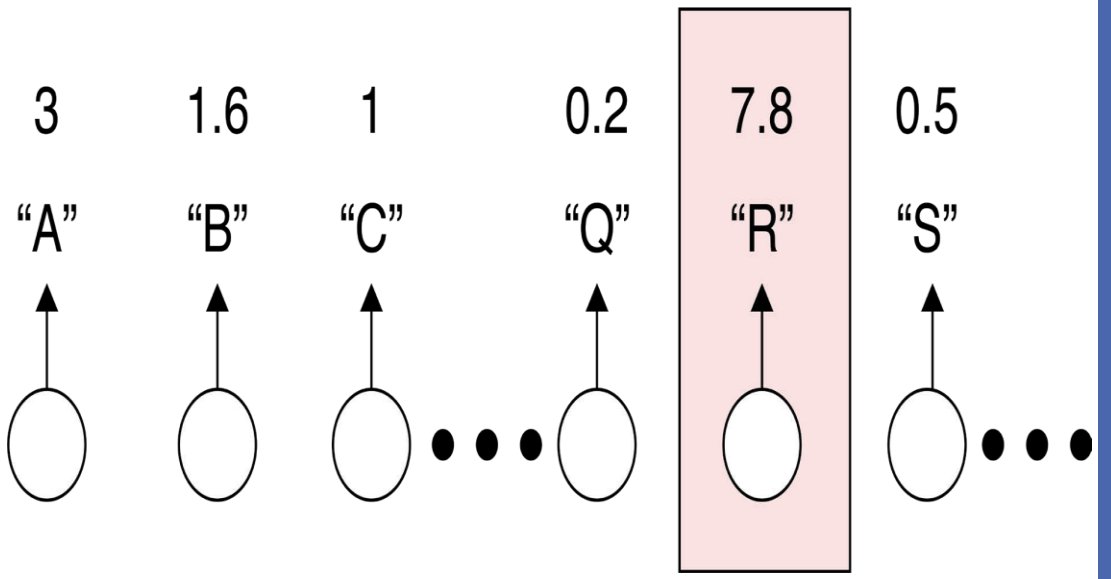


Ouput Layer – Some notes

- Choose the number of neurons in the output layer to match the type of problem we're trying to solve.
- Regression problem:
- A single numerical output, then there will be just one neuron in the output layer, and that neuron's value is our prediction.
- Binary classifier: We have a choice.
- Just one neuron with output values from, say, 0 to 1. Values near 0 mean the input is from one class, while values near 1 mean the input is of the other class.
- Alternatively, we can have two output neurons, one for each category. We'll typically find which neuron has the larger value, and assign the corresponding category to the input.
- Multi-class classifier: Have as many outputs as there are classes.

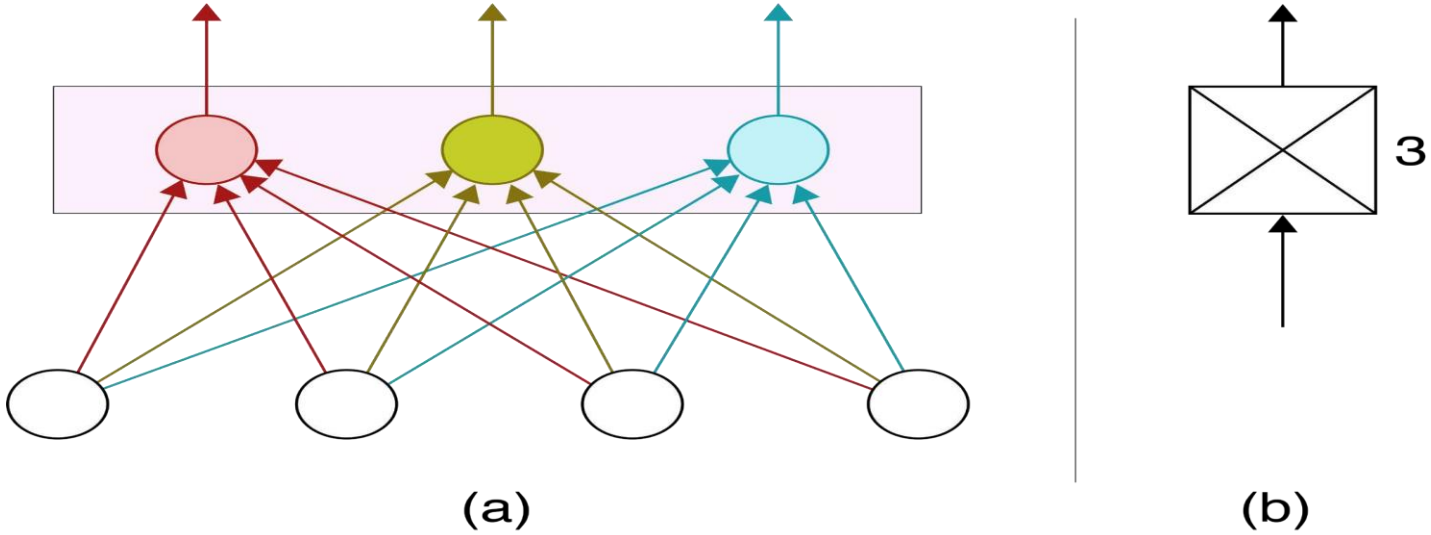
Output Layer Example

- Example: Recognize lower-case letters of the Roman alphabet.
- Have 26 output neurons, one for each letter, providing us with a score for each letter.
- Choose the output with the highest score as the best choice for the category of the input.
- If we want to interpret these outputs as probabilities, we can pass them through a softmax step.
- Figure shows the idea. If we're categorizing individual letters, we might have 26 outputs, each one giving us a score for that letter. Here, the letter "R" has the largest score shown.



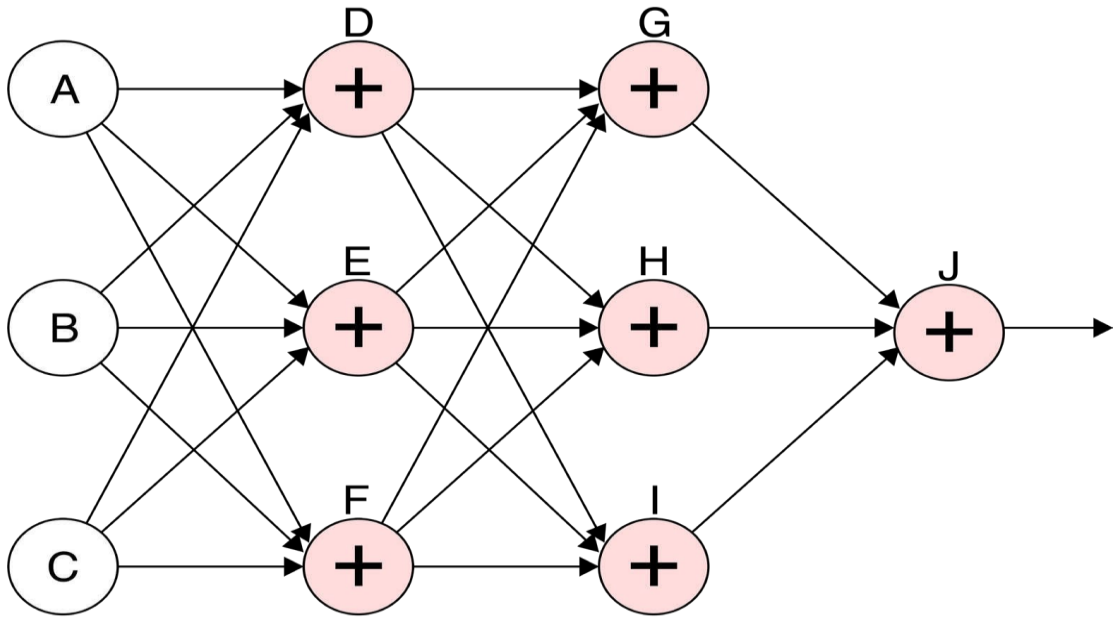
Fully-Connected Layer

- A fully-connected layer, or Multi-Layer Perceptrons (MLPs)
- The colored neurons make up a fully connected layer.
- Each of the neurons in the upper layer receives an input from every neuron in the previous layer.
- Schematic symbol for a fully-connected layer.



Fully Connected Network

- A fully-connected network of 3 inputs, 1 output, and two internal layers of 3 neurons each.
- Each arrow represents a connection and implicitly has an attached weight.

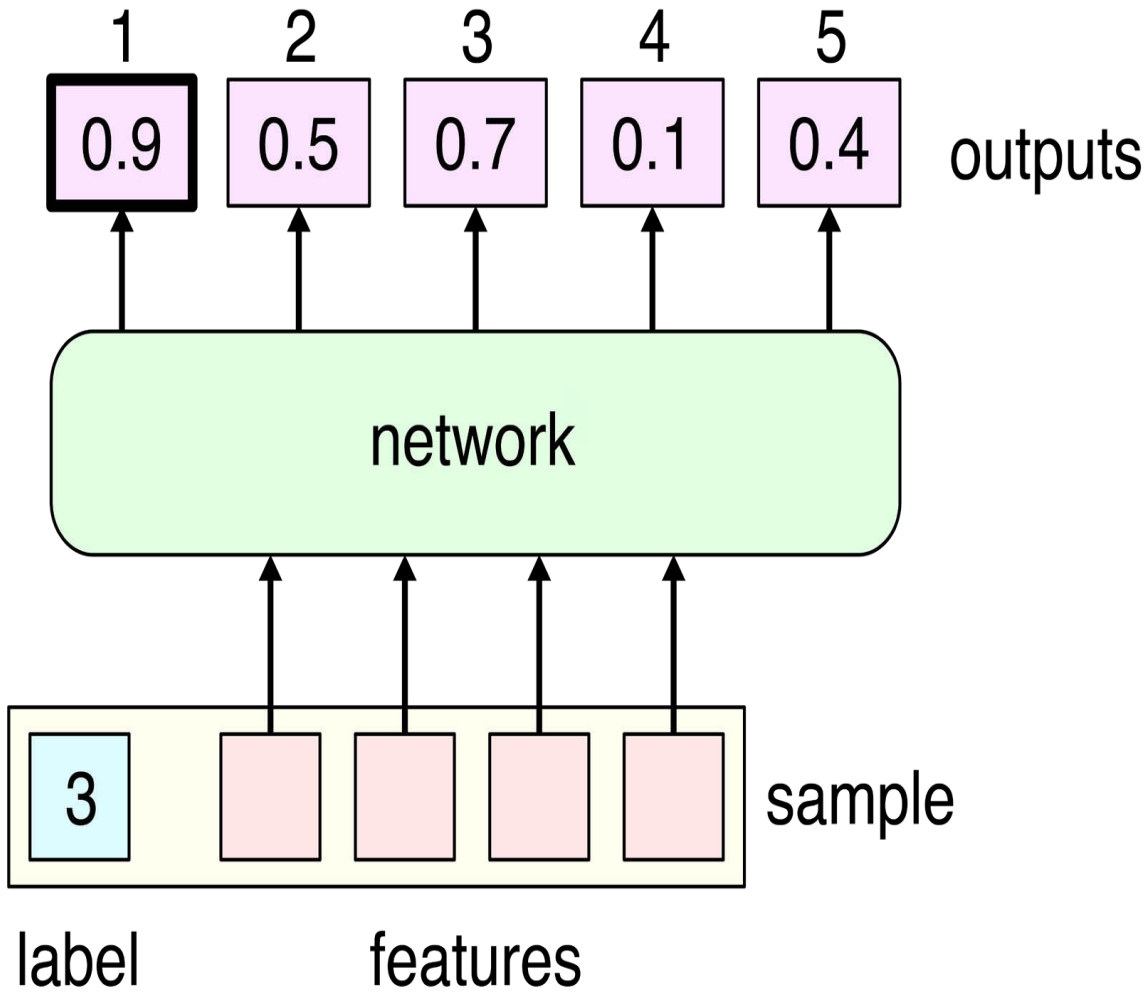


Back Propagation-Introduction

- Backpropagation, or simply backprop, is a low-level algorithm. Without backprop, we wouldn't have today's widespread use of deep learning, because we wouldn't be able to train our models in reasonable amounts of time.
- With backprop, deep learning algorithms are practical and plentiful.
- The backpropagation algorithm is simple, and is implemented efficiently.
- In this module, we'll try to be understand every step of backpropagation.

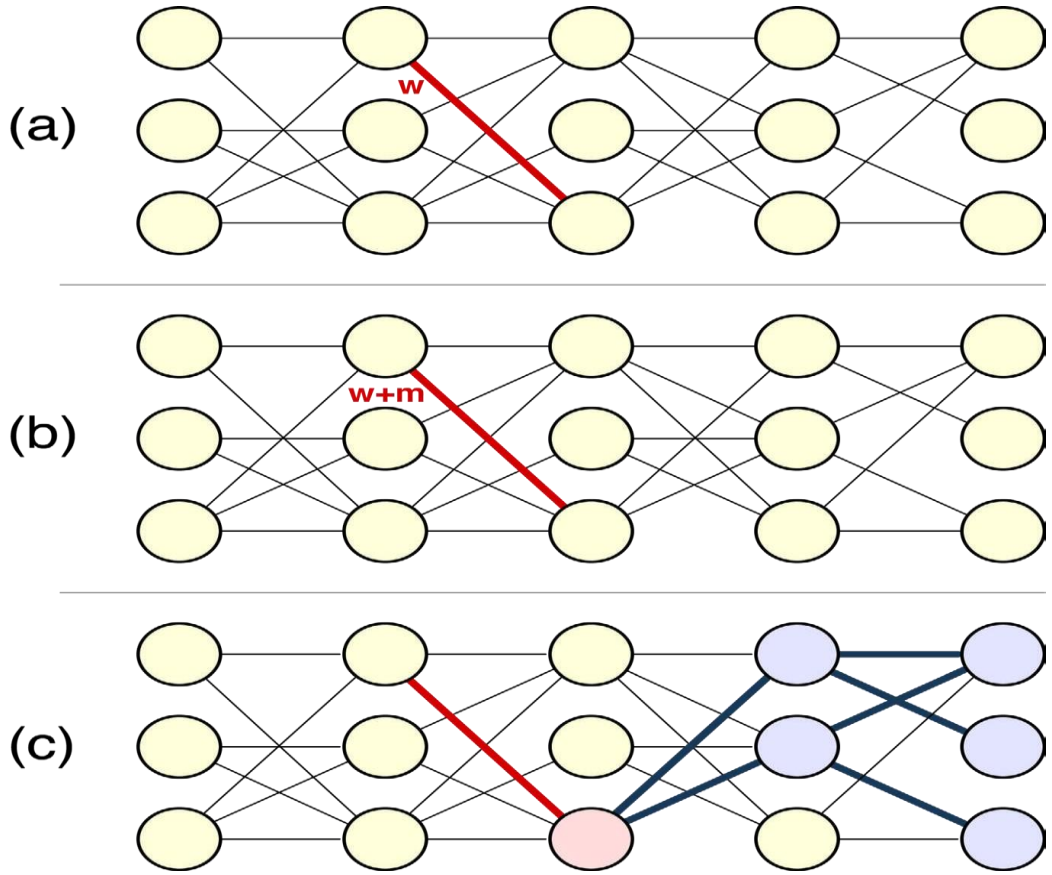
A very slow way to learn– trial and error

- Network designed to classify each input into one of 5 categories.
- Largest output is network's prediction for an input's category.
- Error is difference between correct label and predicted label
- Network wants to minimize the error.
- Start with random weights, giving random outputs, giving error.
- Adjust weights by trial and error, & try to kill the error.
- Very slow !
- Example Classifier



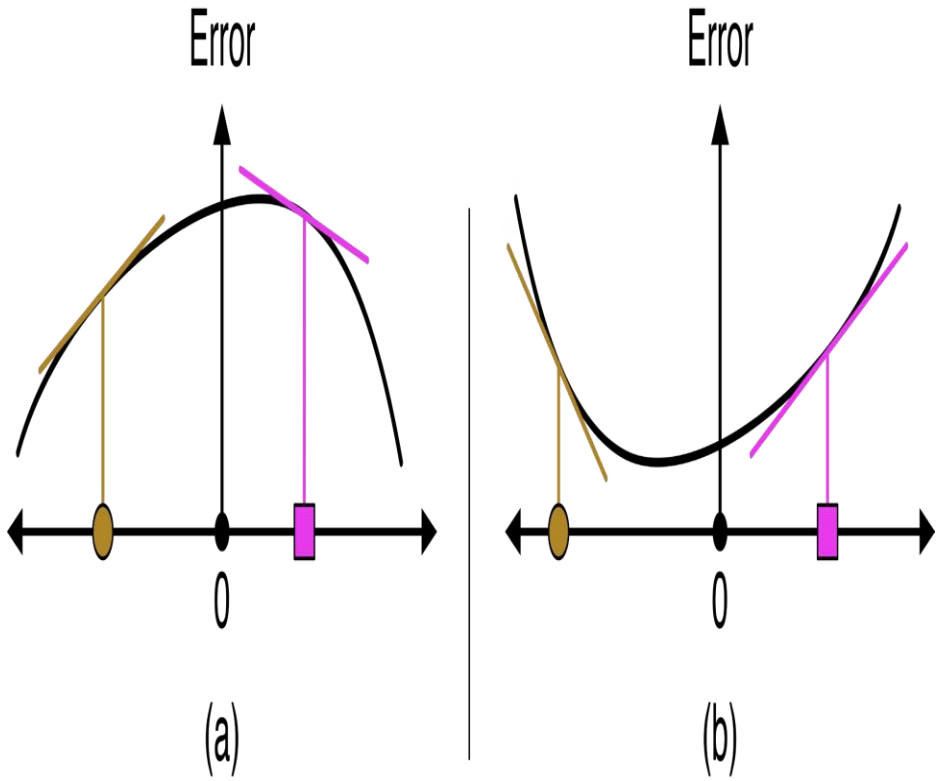
Change in weight leads to change in net's outputs

- Updating a single weight causes a chain reaction that ultimately can change the network's outputs.



Faster way to learn - use gradients

- Gradient tells what happens to error (the black curves) if we move a weight to the right or left.
- Gradient is given by the slope of the curve directly above the point we're interested in.
- If we have the gradient for a weight, we can adjust it exactly as needed to make the error go down.
- If have the gradient for every weight in the whole network, we can update all of the weights simultaneously by adding a small value (positive or negative) to each weight in the direction given by its own individual gradient.
- That would be an immense time- saver.



Plan to reduce network errors

- Goal: Reduce the overall error for a sample, by adjusting network weights.
- Do in two steps:
  - First step: Use backpropagation algorithm to calculate and store a number called the "delta" for every neuron. This number is related to the network's error.
  - Second step: Update weights based on neuron deltas and neuron outputs. This step is called the update step.

What is a “delta”?

- The delta represents gradients.
- Every delta value represents a different gradient.
- For instance:
  - the delta attached to a neuron C describes the gradient of the error with respect to the output of C,
  - the delta for A describes the gradient of the error with respect to the output of A.

The overall plan to train a network

- Run a sample through the network
- Get the prediction
- Compare that prediction to the label to get an error.
- If error is greater than zero, use it to compute the "delta" at every neuron.
- Update each weight , using neuron deltas & neuron outputs
- Then, move on to the next sample, and repeat the process
- Repeat above until the predictions are all perfect or we decide to stop.

The overall plan to train a network

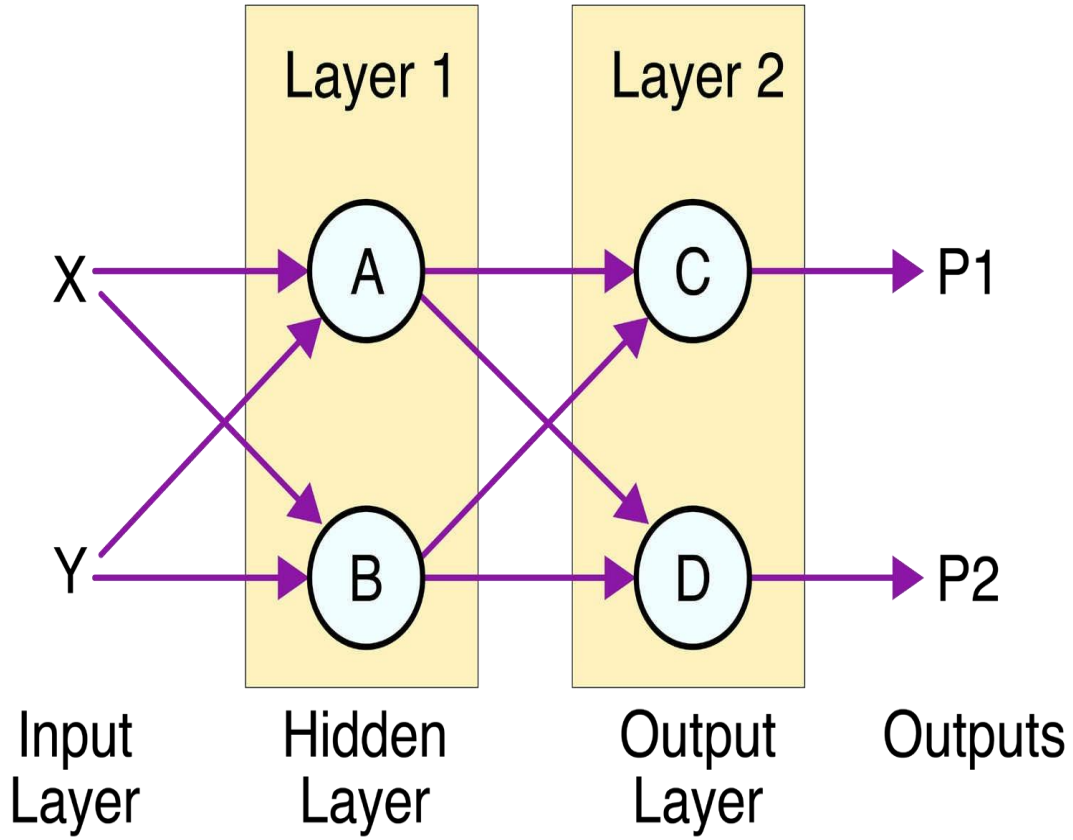
- As we discussed above:
- We'll evaluate one sample, calculate the error, compute the deltas with backprop, and then update the weights using the learning rate.
- Then we'll move on to the next sample.
- Each time we process all the samples in the training set, we say we've completed one epoch of training.
- Repeat above until the predictions are all perfect or we decide to stop.

So, finding deltas is the key to our plan

- Thisplan makes knowing the deltaor gradient an important issue.
- Finding the delta efficiently is the main goal of this module.
- Remember that our goal is to find the deltas for the weights.
- When we know the deltas for all the neurons in a layer, we can update all the weights feeding into that layer.
- Let's see how that's done using a tiny network.

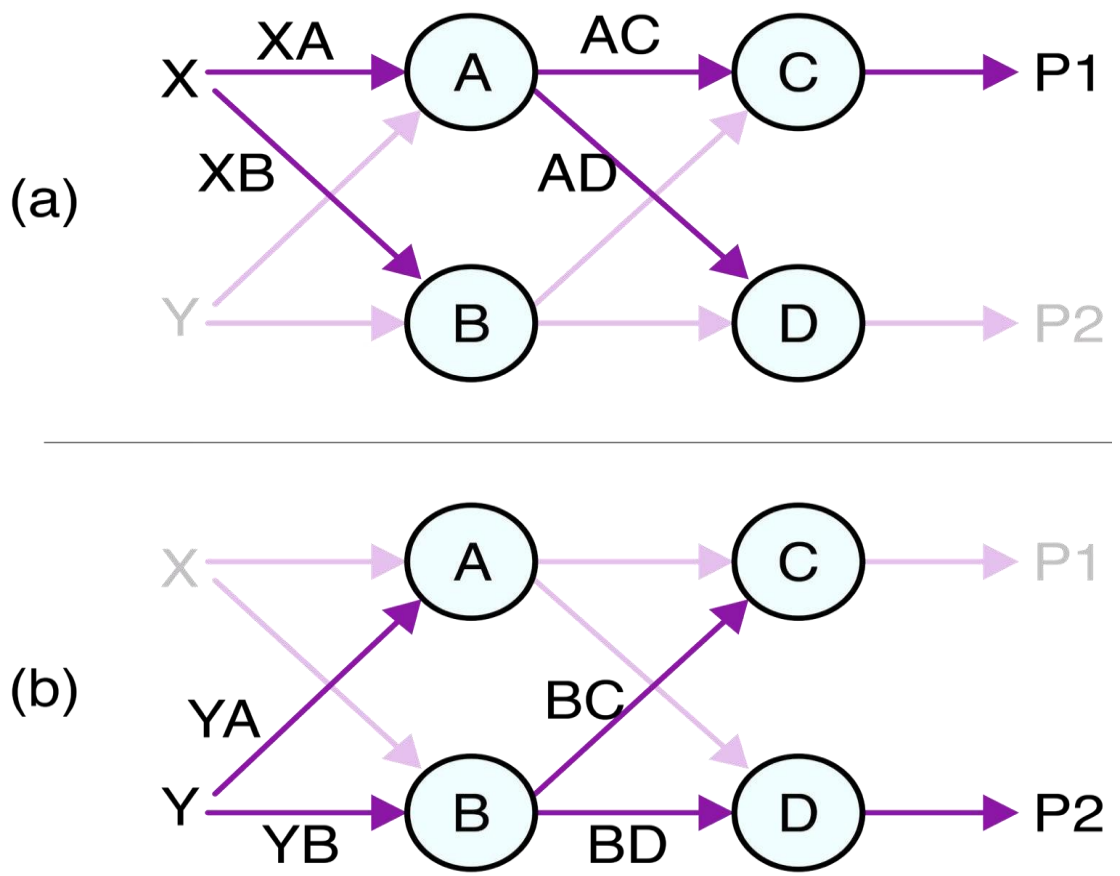
A Tiny DL Network to study Backprop

- Net classifies 2D points into two categories, class 1 and class 2.
- Inputs: X & Y coordinates of each point.
- Outputs: Predictions P1 and P2.
- P1 is the likelihood that sample belongs to class 1, and P2 is likelihood that sample belongs to class 2.
- Whichever is larger is the network's predicted category for this input.



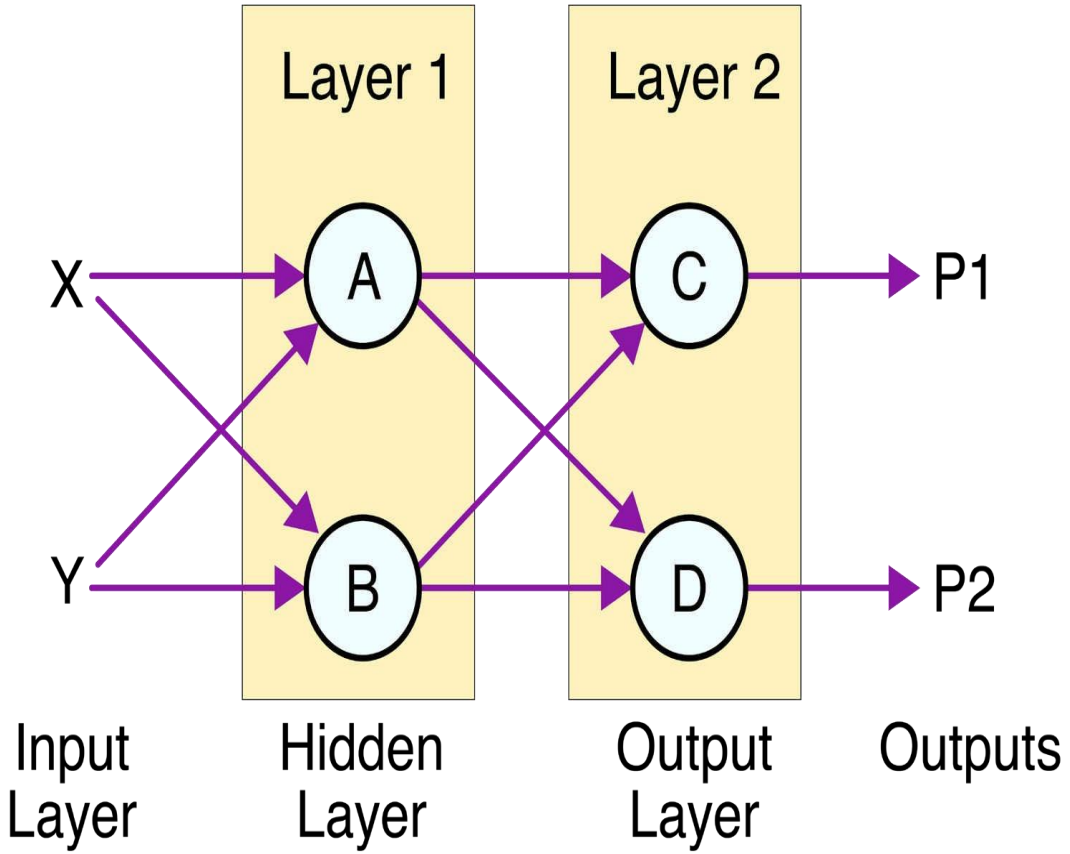
Naming weights in Tiny DL network

- Each weight is named based on the two connecting neurons -with the starting neuron (on the left) first, and then the destination neuron (on the right) second.



Tiny Net is a DL Net

- This is a tiny deep-learning network with two layers.
- The first layer contains neurons A and B.
- The second layer contains neurons C and D.
- The input layer doesn't do any computing, so it's usually not included in the layer count.

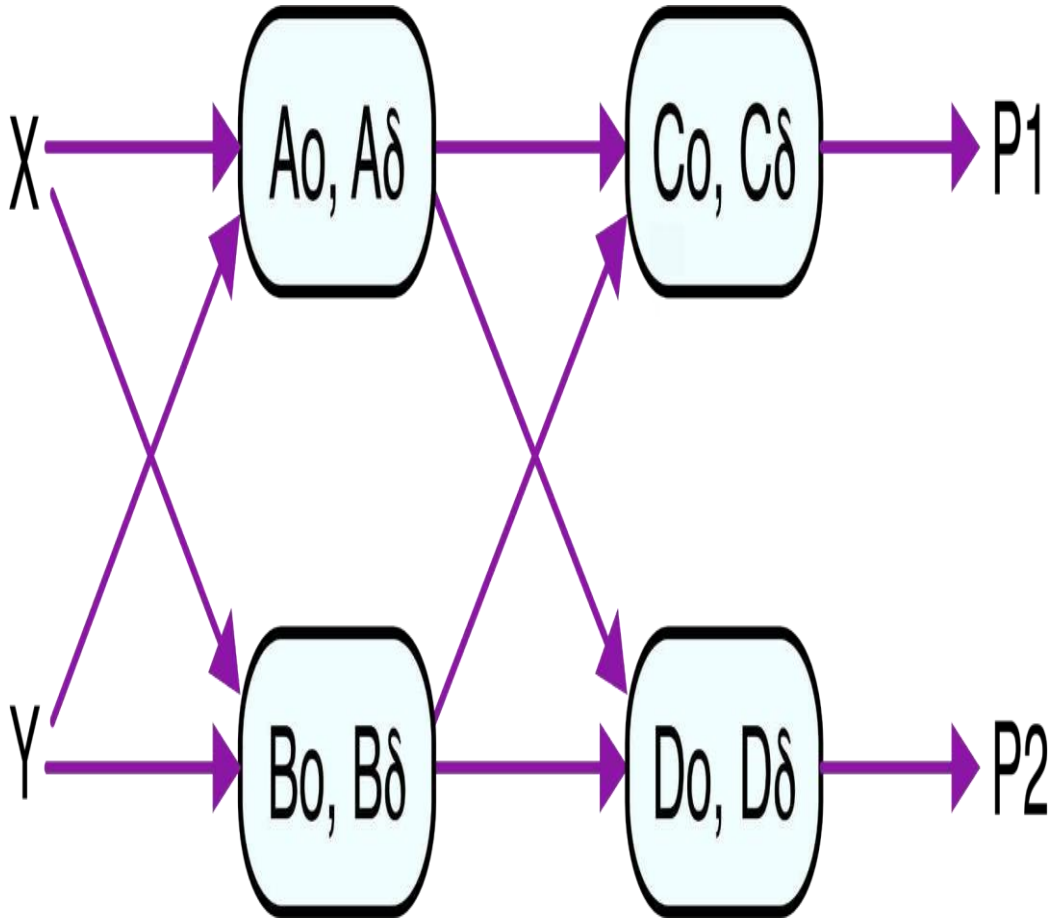


Tiny network with output and delta values for each neuron.

- We want to refer to the output and delta for every neuron.
- Ao and Bo will be the names of the outputs of neurons A and B,
- Aδ and Bδ will be the delta values for those two neurons.

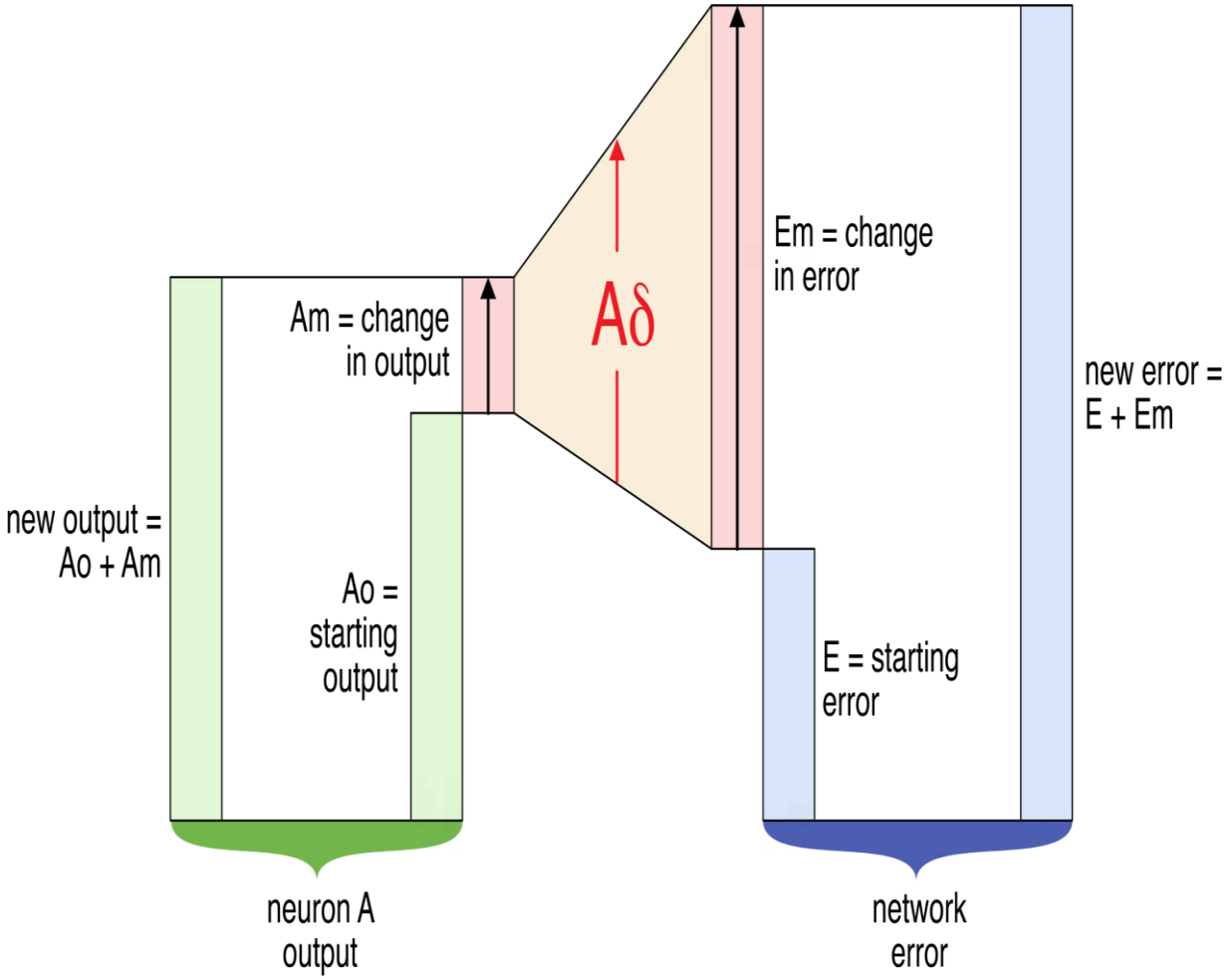
Q: What's Aδ?

- A: Delta value Aδ relates a change in neuron's output A to a change in the error.
- These are not relative or % changes, but actual amounts
- Similarly, for C and D



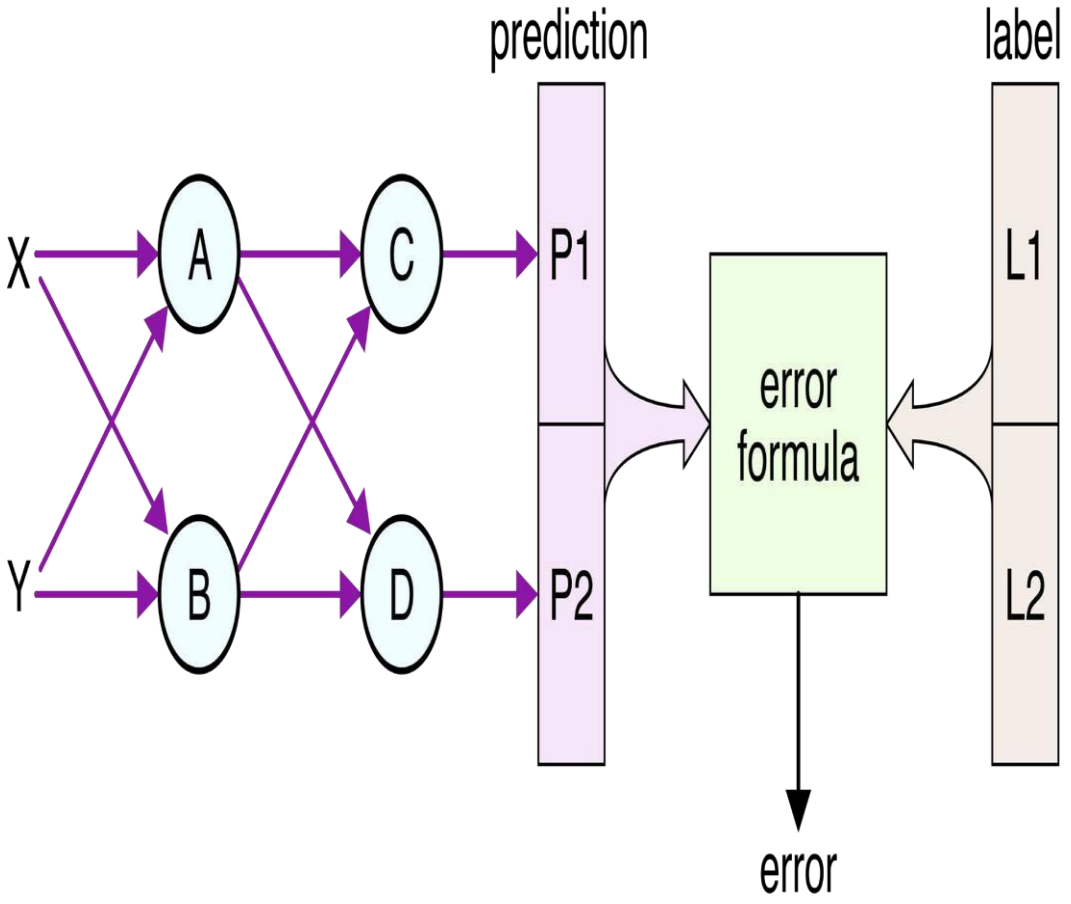
Visualizing how changes in a neuron's output can change the network's error

- Neuron A starts with output Ao
- Change a weight on its inputs in such a way that output goes up by Am.
- This change Am is multiplied by Aδ to give us Em, the change in the error.
- The new error is E+Em.
- In this case, both Am and Aδ are positive, so the change in the error Am×Aδ is also positive, increasing the error.
- Example: if the output of A goes from 3 to 5, that's a change of 2, so the change in the error would be Aδ×2.



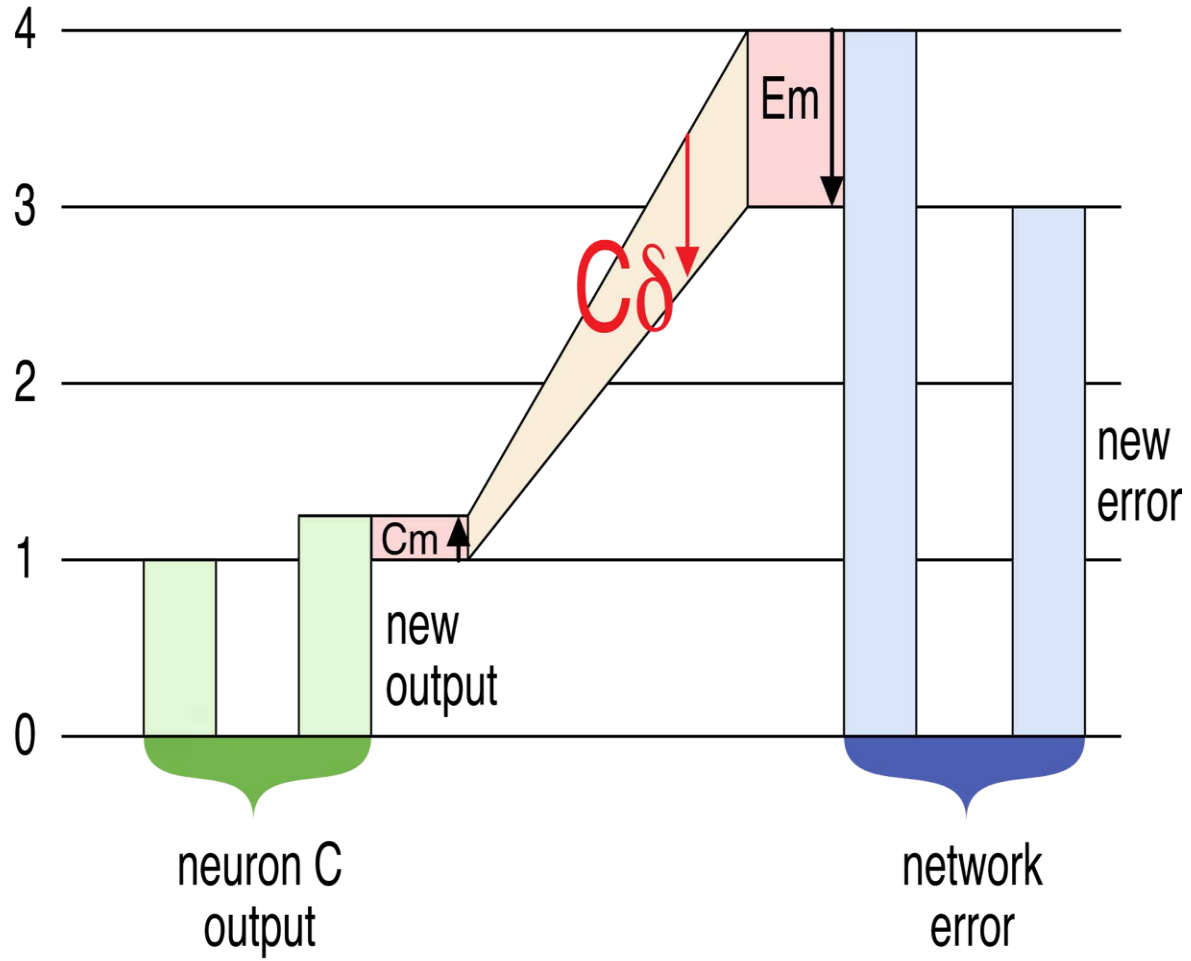
Delta for output Neurons

- To find the error from a specific sample, we start by supplying the sample's features X and Y to the network.
- The outputs are the predictions P1 and P2, telling us the likelihoods that the sample is in class 1 and class 2, respectively.
- We compare those predictions with the one-hot encoded label, and from that come up with a number representing the error.
- If the predictions match the label perfectly, the error is 0.
- Bigger the mismatch, bigger is the error.



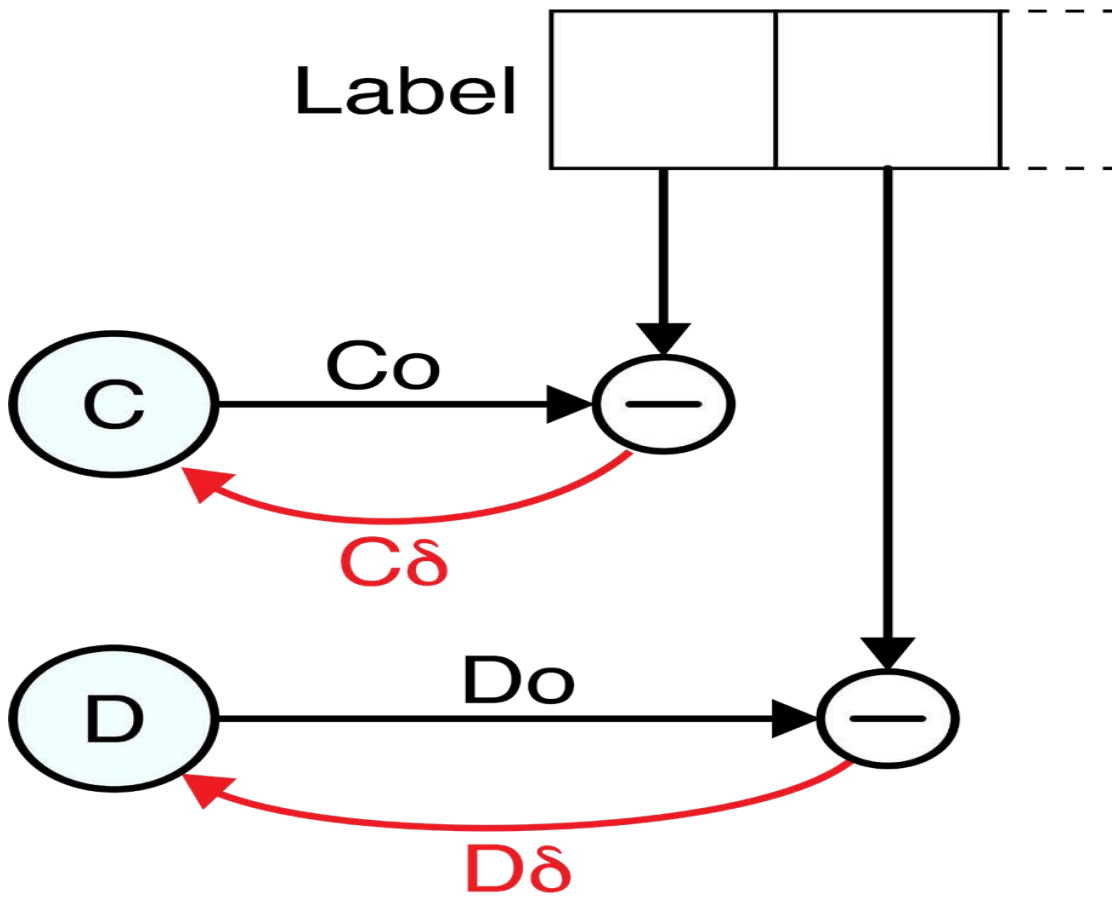
Error diagram illustrating the change in the error from a change in output of neuron C.

- Due to a change in the inputs, the output of C increases by an amount Cm.
- This is amplified by multiplying it with Cδ, giving us the change in the error, Em.
- That is, Em=Cm×Cδ.



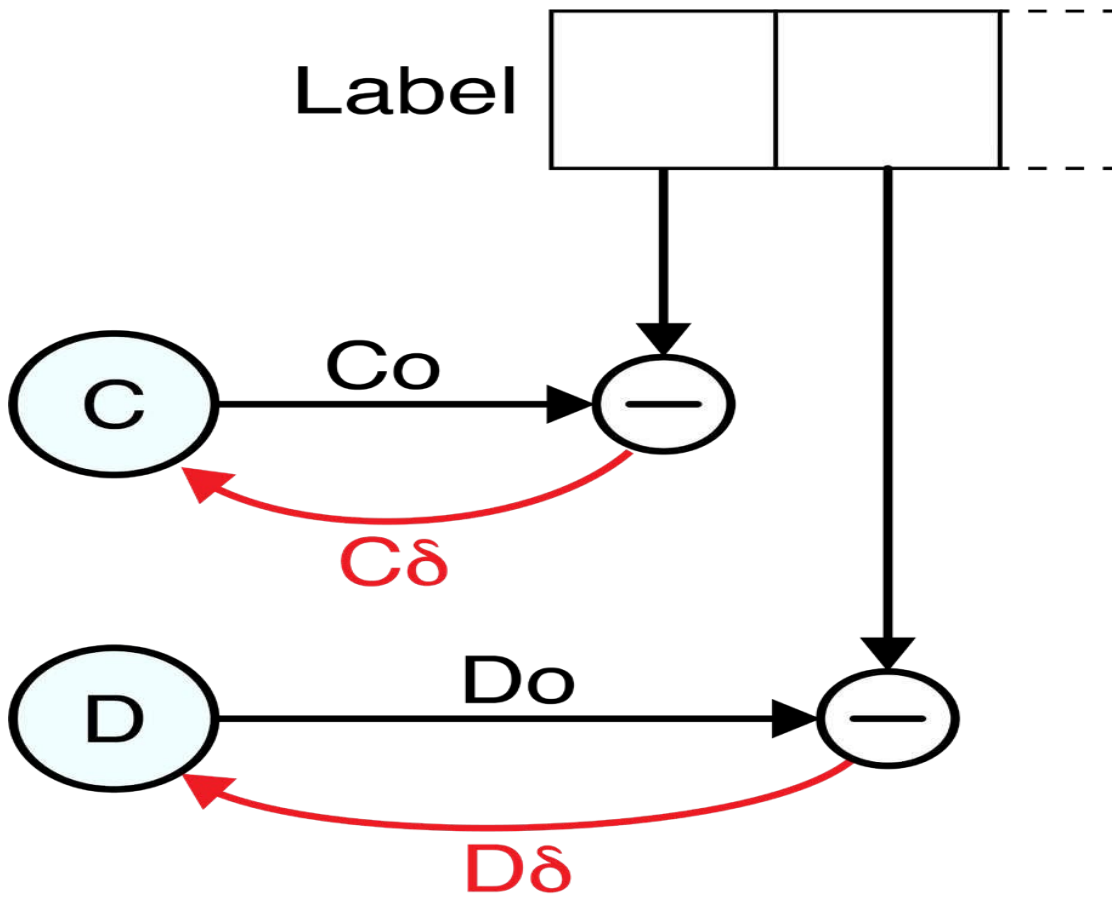
How to calculate "Delta" for Output Neurons

- Lets use the quadratic cost function or mean squared error (or MSE) as error measure.
- For any output neuron:
- Delta = label value – neuron output
- Compute & save that delta value with its neuron (in red).



How to calculate "Delta" for Output Neurons

- Delta = label value – neuron output
- So "delta" is a temporary value that changes with every new label value & every new output of the neuron.
- So, if the neuron's output changes, the delta changes as well.
- Following this observation, any time we update the weights in our network, we'll need to calculate new deltas.

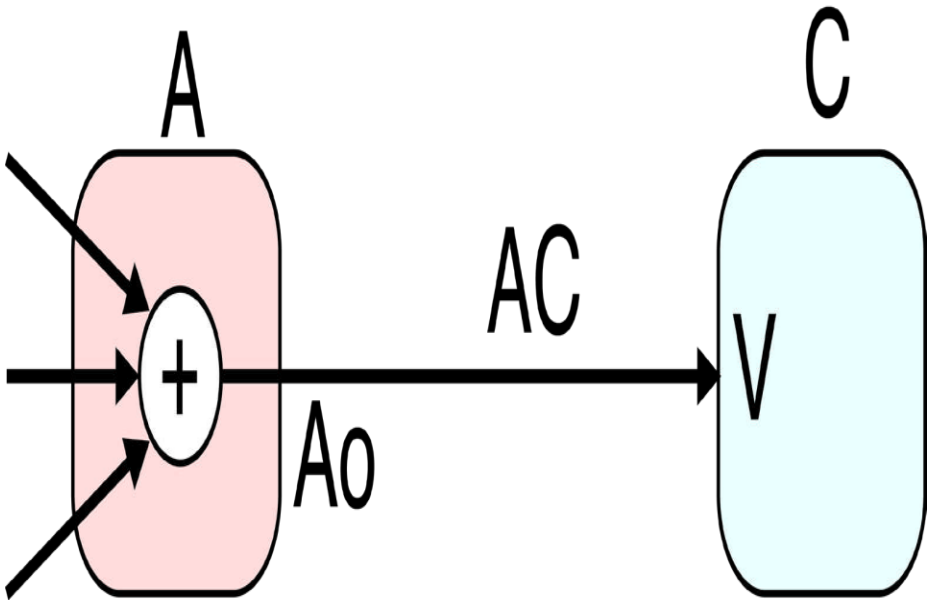


Using Output Deltas to Change Last Weights

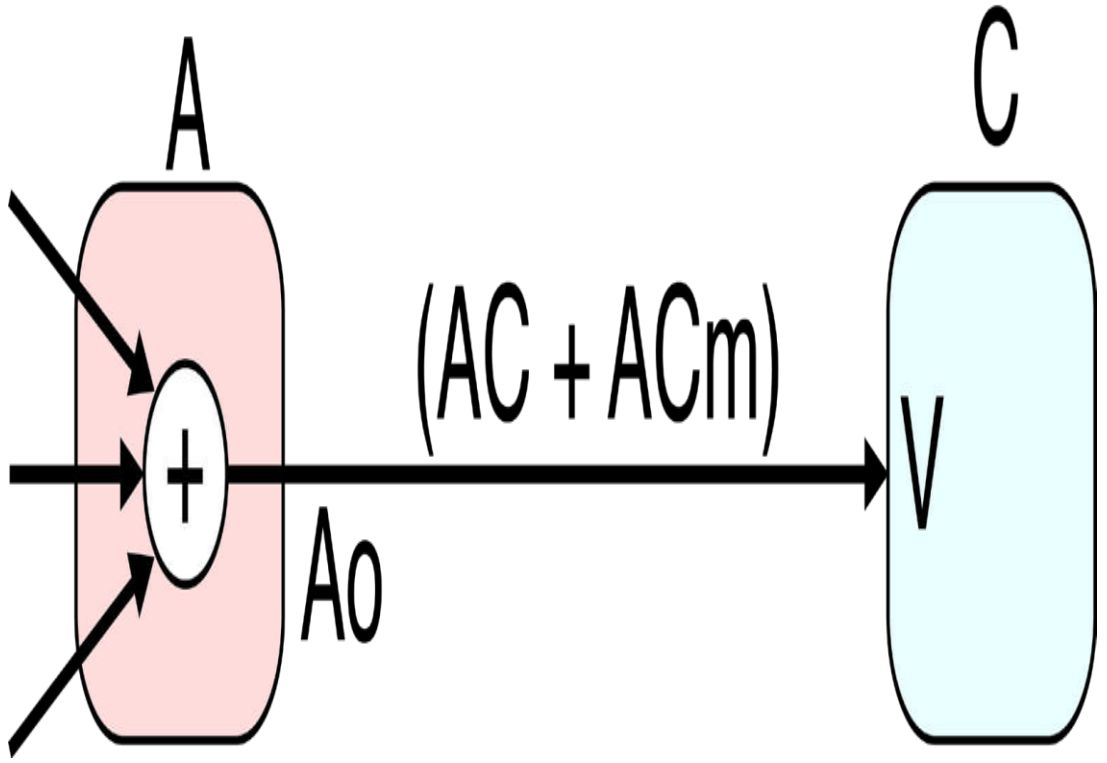
- Remember that our goal is to find the deltas for the weights.
- When we know the deltas for all the neurons in a layer, we can update all the weights feeding into that layer.
- Let's see how that's done.

Computing the value V coming into neuron C

- V = Ao (the output of A) times the weight AC, or V= Ao×AC



- Change weight to AC +ACm, then new V =Ao×(AC+ACm)



Finding out how a change in the last weights affects Network error

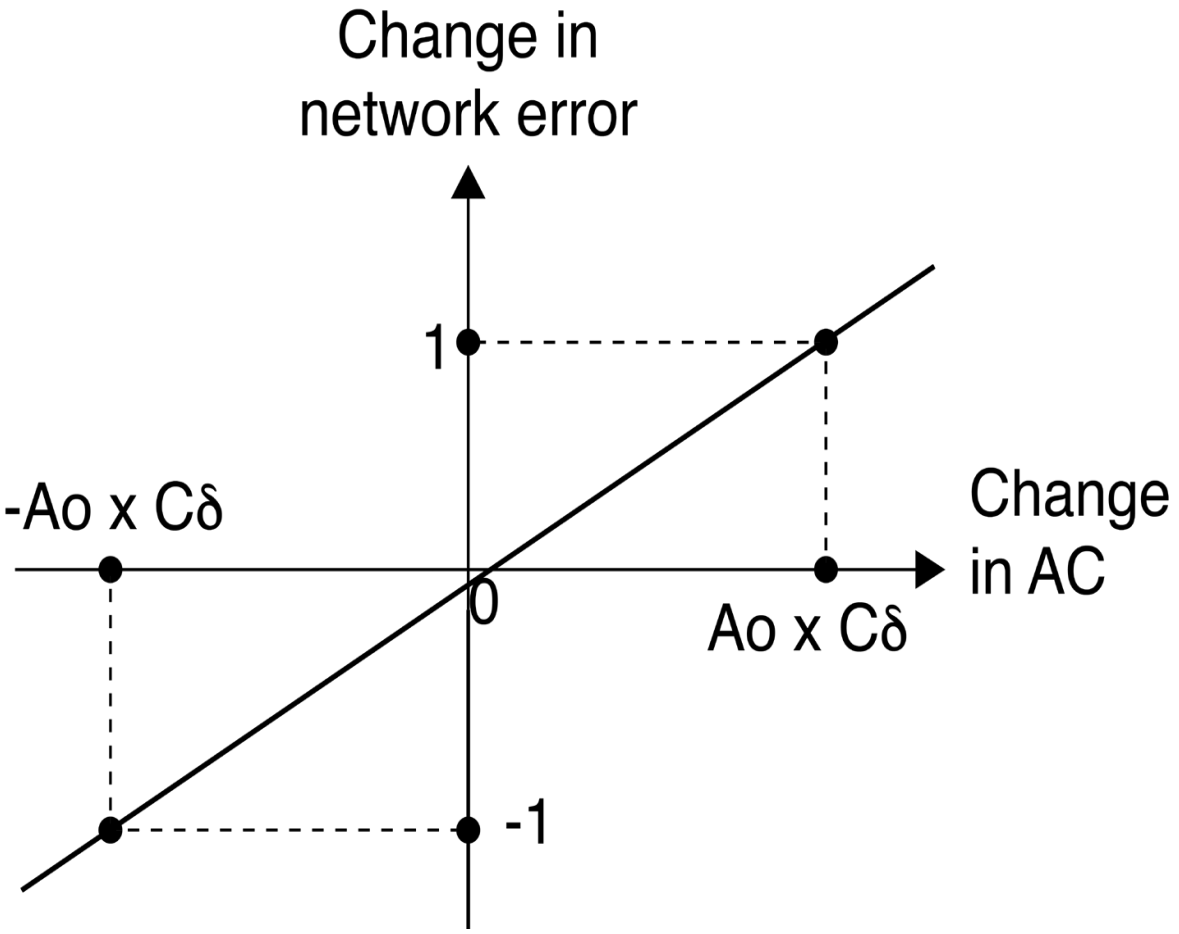
- So, due to modifying the weight, change in V coming into C is Ao×ACm.
- To find change in network error, multiply with Cδ this change in V (going into C).
- This tells us the change in the network's error as a result of modifying the weight.
- We discovered how a change to a weight would affect the network's error !

Finding out how a change in the last weights affects Network error (contd.)

- Let's look at that again.
- If we change the weight AC by adding ACm to it, then the change in the network's error is given by the change in in V coming into C, (Ao×ACm), times the delta for C, or Cδ.
- That is, the change in error = (Ao×ACm)×Cδ

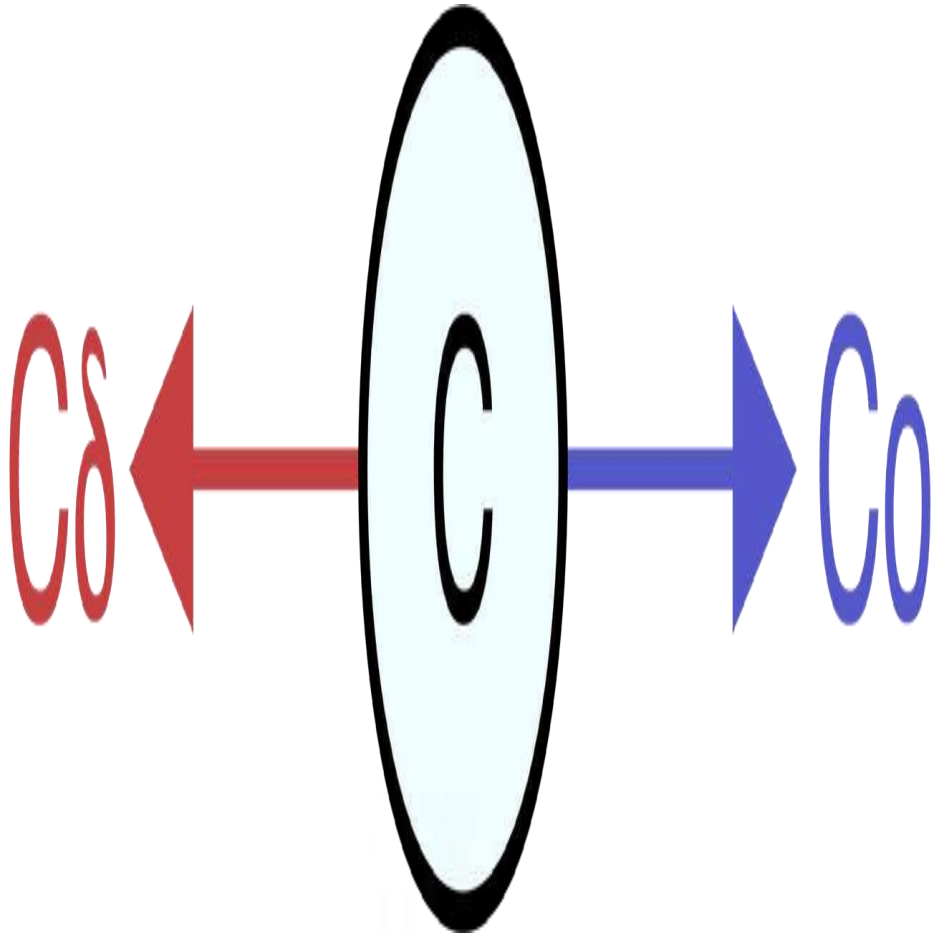
Change in Error Vs. Change in Weight

- To reduce error by unity, decrease AC by Ao×Cδ
- So, new AC = AC - (Ao×Cδ)
- We shall use the above for updating the value of weight AC so as to reduce the error.



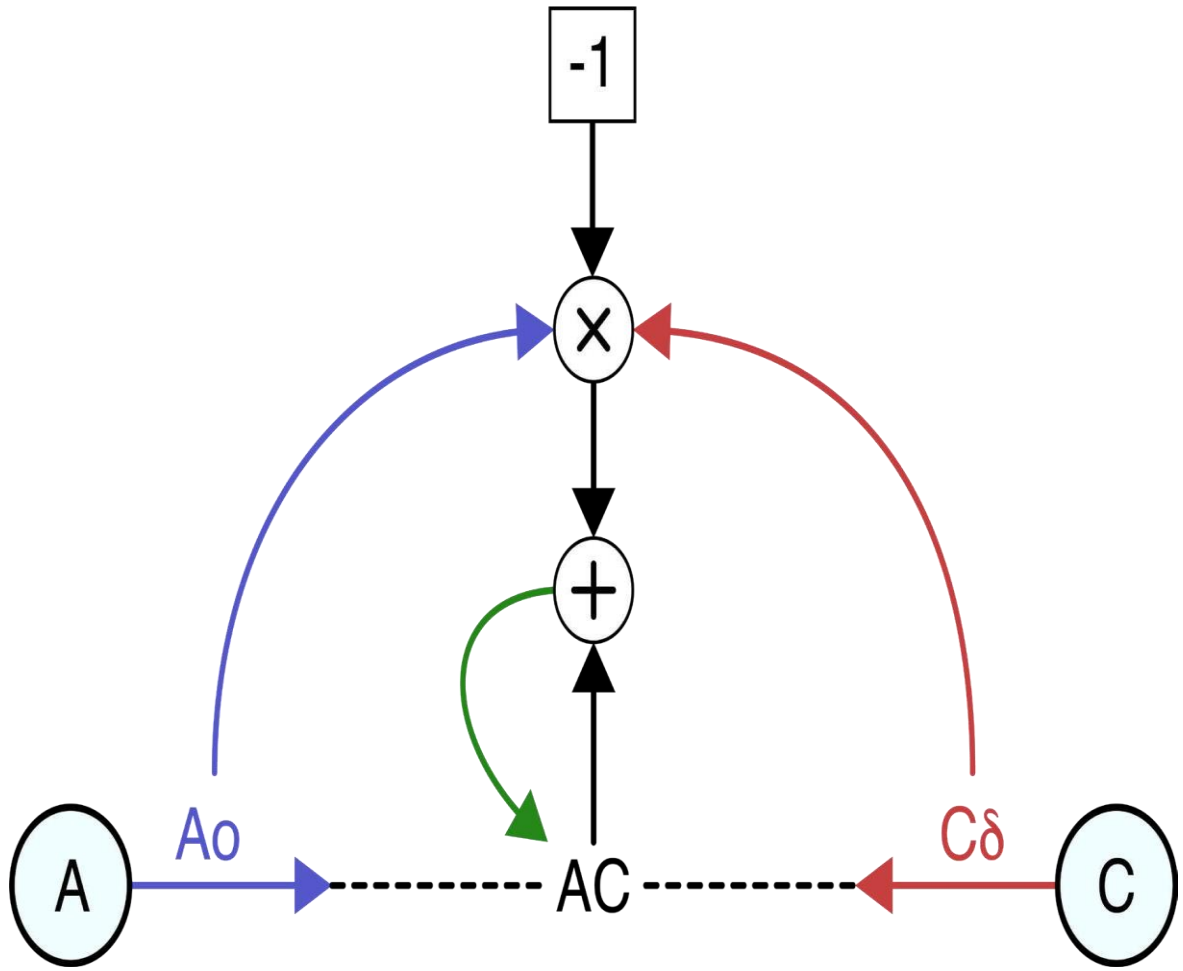
New Convention

- C is a neuron
- Output Co is drawn with an arrow pointing right.
- Delta Cδ is drawn with an arrow pointing left.



How to adjust last weights to reduce network error

- Start with output Ao from neuron A and the delta Cδ from neuron C.
- Multiply them together.
- Subtract this from the current value of AC.
- To show this clearly in the diagram, we instead multiply the product by –1, and then add it to AC.
- The green arrow is the update step, where this result becomes the new value of AC.



How to adjust all other weights?

- We've seen how to adjust the last weights in the network
- How about all the other weights?
- First, we need to figure out the deltas for all the neurons in all the remaining layers.
- Then we can use previous figure to improve all the other weights in the network.

Step 3: Finding Neuron deltas for other layers

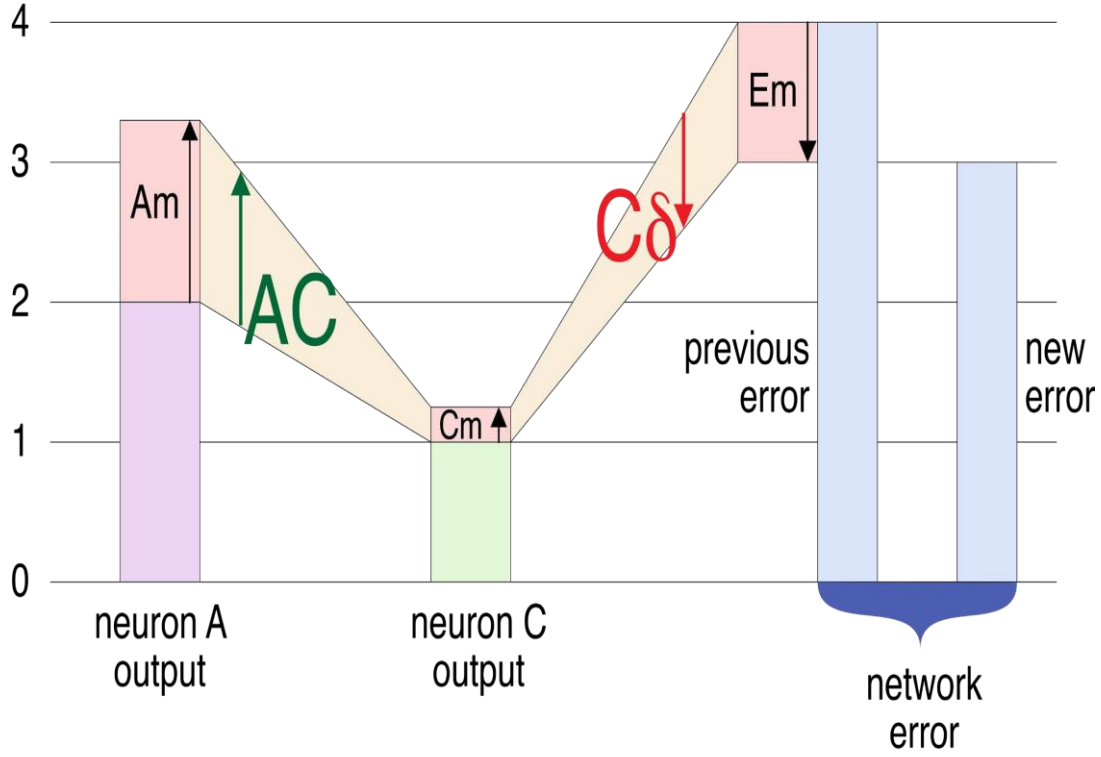
- Remarkable trick of backpropagation: we can use the neuron deltas at one layer to find all the neuron deltas for its preceding layer.
- Knowing the neuron deltas and the neuron outputs tells us how to update all the weights coming into that neuron.
- Let's see how to do that.

Step 3: Finding Neuron deltas for other layers

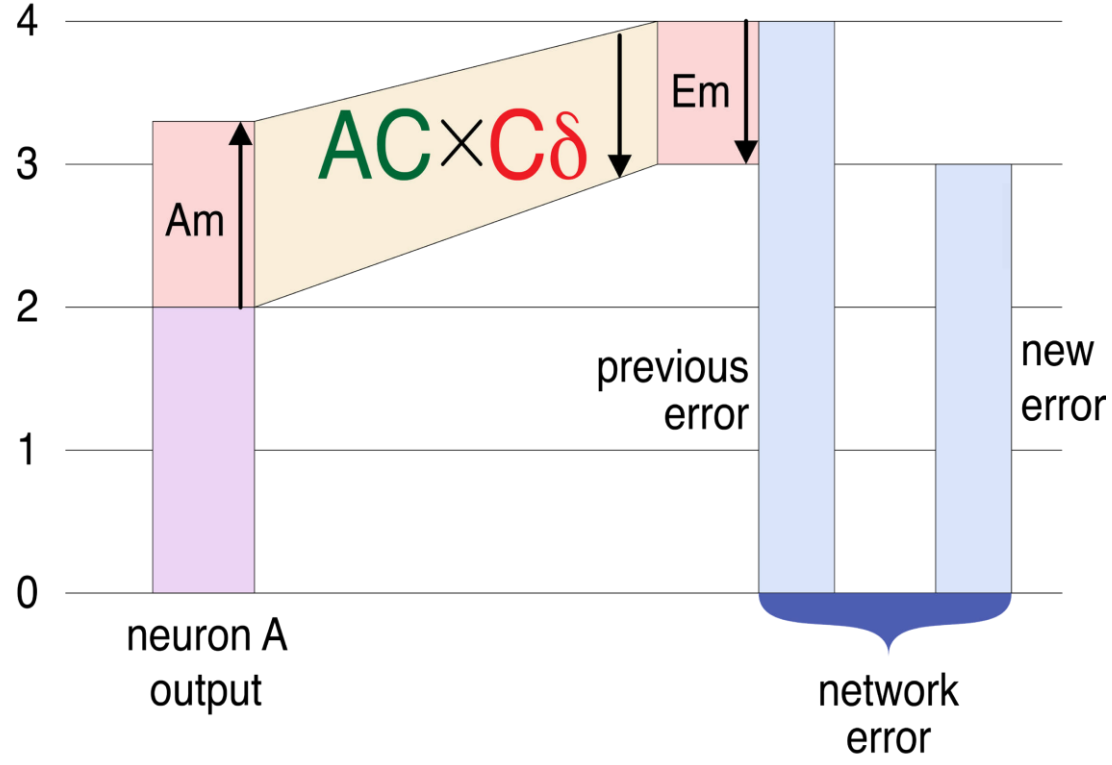
- Now that we have the delta values for the output neurons, we will use them to compute the deltas for neurons on the layer just before the output layer.
- In our tiny DL network, that's just neurons A and B.
- Let's focus for the moment just on neuron A, and its connection to neuron C.

Tracking a change in output of neuron A

- Change Am is multiplied by the weight AC and added to the values accumulated by neuron C. This changes output of C by Cm. This change in C is multiplied by Cδ to find the change in the network error.

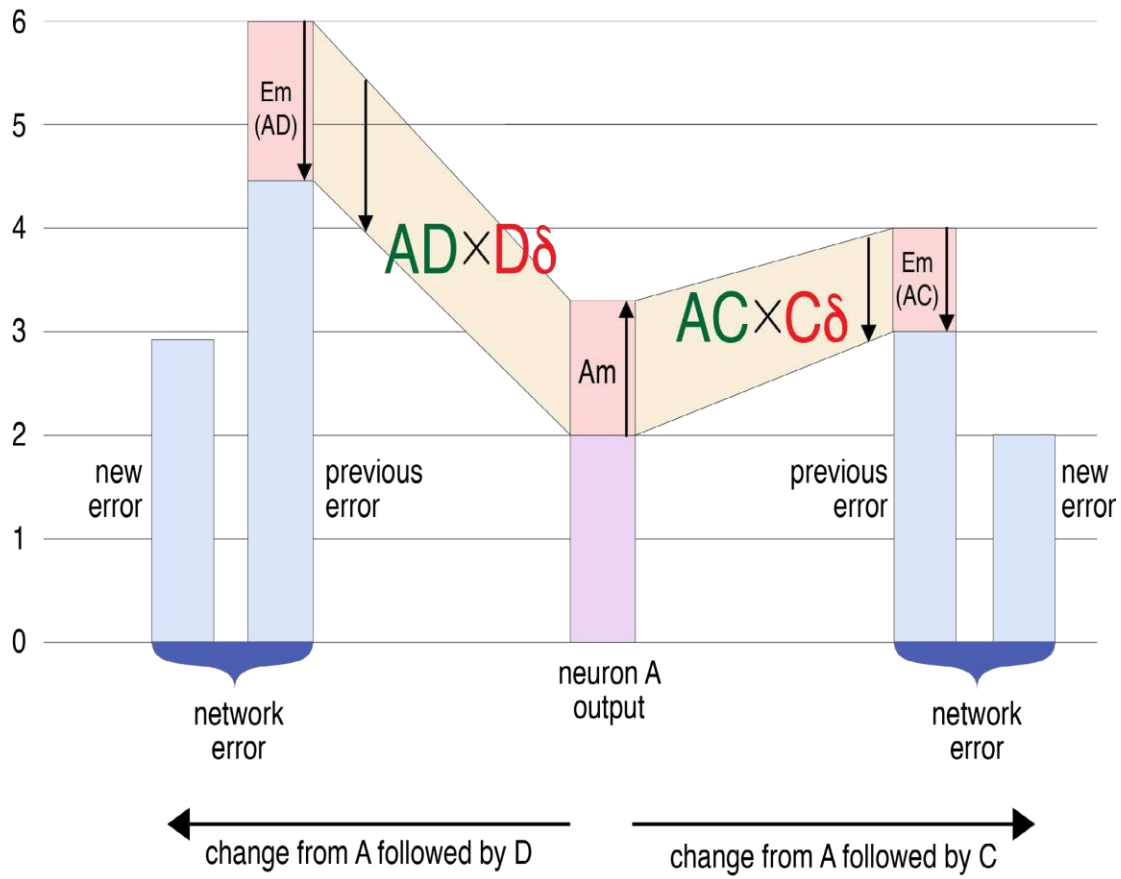


- Mushing together the operations in Left Figure into a more compact diagram below



Tracking a change in output of neuron A

- But now remember the rest of the network: ee that neuron D also uses the output of A.
- If Ao changes due to Am, then the output of D will change as well, and that will also have an effect on the error.
- If Ao changes by Am, the change in the error due to the change in D is given by AC×Dδ.
- Figure shows these two paths (A to C, A to D) at the same time

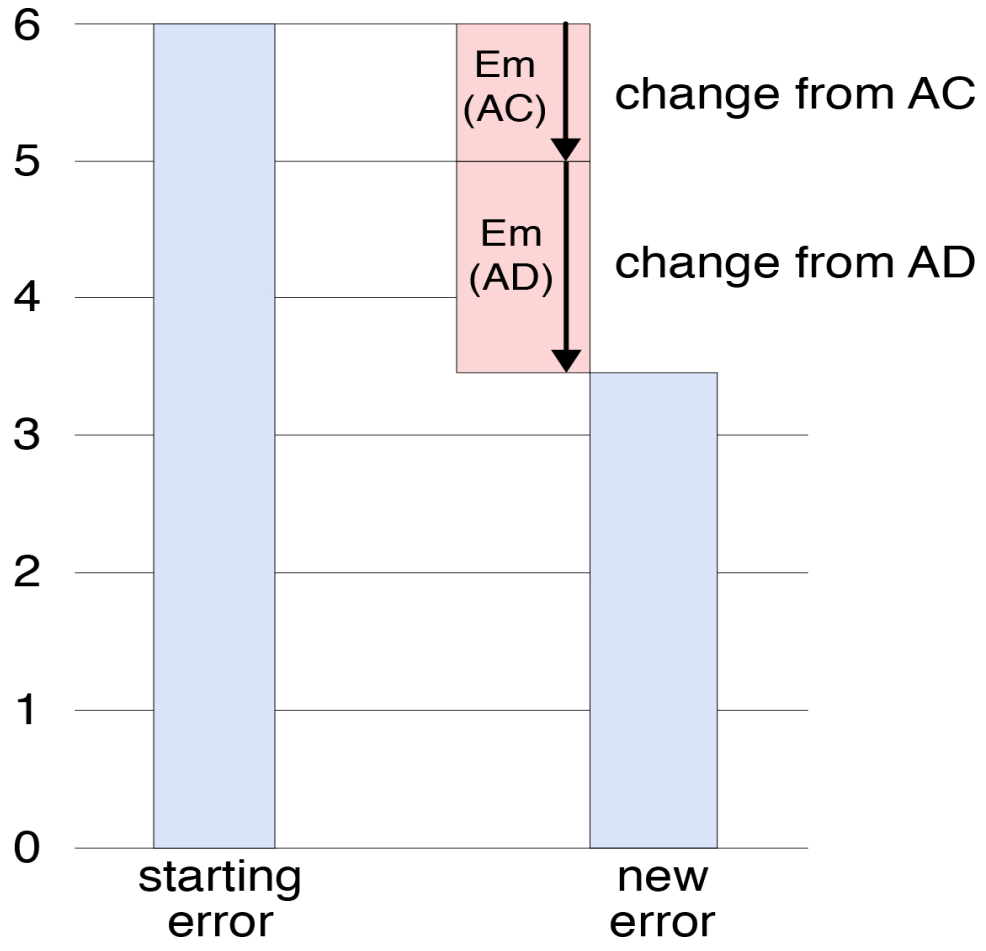


Tracking a change in output of neuron A

- When the output of neuron A is used by both neuron C and neuron D, the resulting changes to the error add together.
- Now that we've handled all the paths from A to the outputs, we can finally write the value for Aδ. Since the errors add together, as in Figure, we can just add up the factors that scale Am.

$A\delta = (AC \times C\delta) + (AD \times D\delta)$

- Now that we've found the value of delta for neuron A, we can repeat the process for neuron B to find its delta.

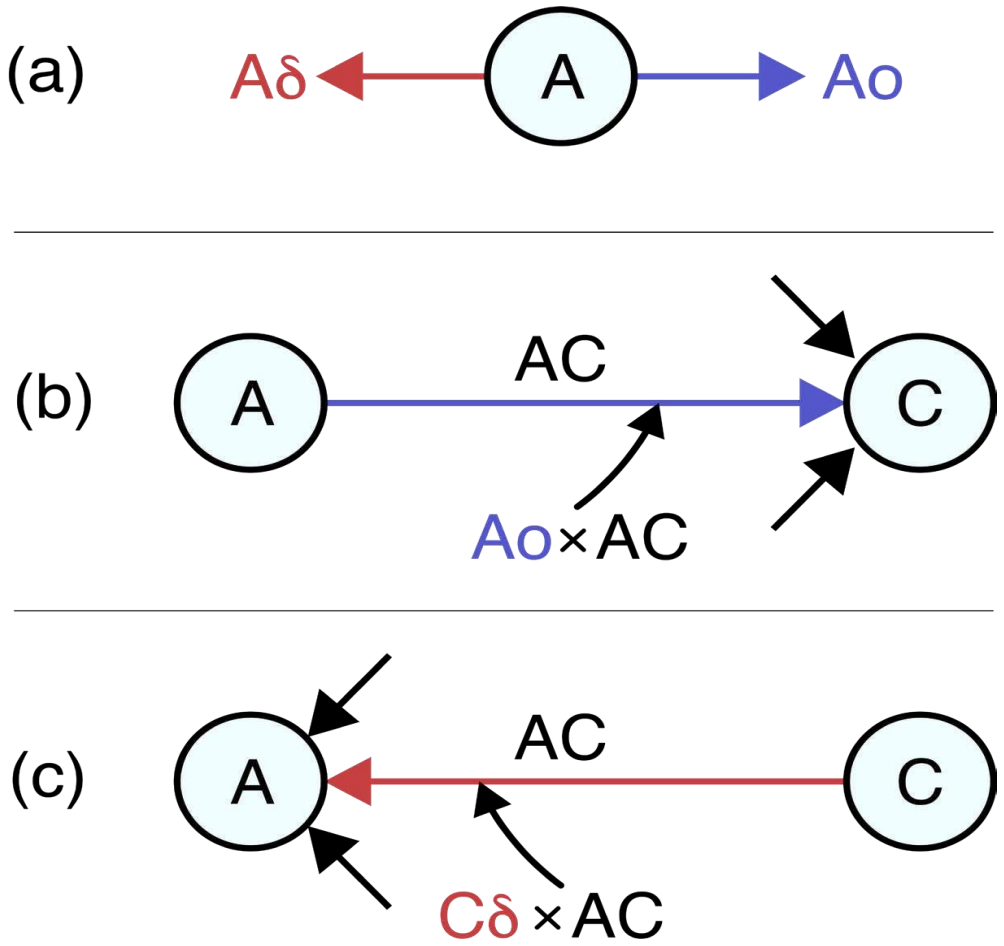


How to find delta for every neuron

- We've actually done something far better than find the delta for just neurons A and B.
- We've found out how to get the value of delta for every neuron in any network, no matter how many layers it has or now many neurons there are!
- That's because everything we've done involves nothing more than a neuron, the deltas of all the neurons in the next layer that use its value as an input, and the weights that join them. With nothing more than that, we can find the effect of a neuron's change on the network's error, even if the output layer is dozens of layers away.

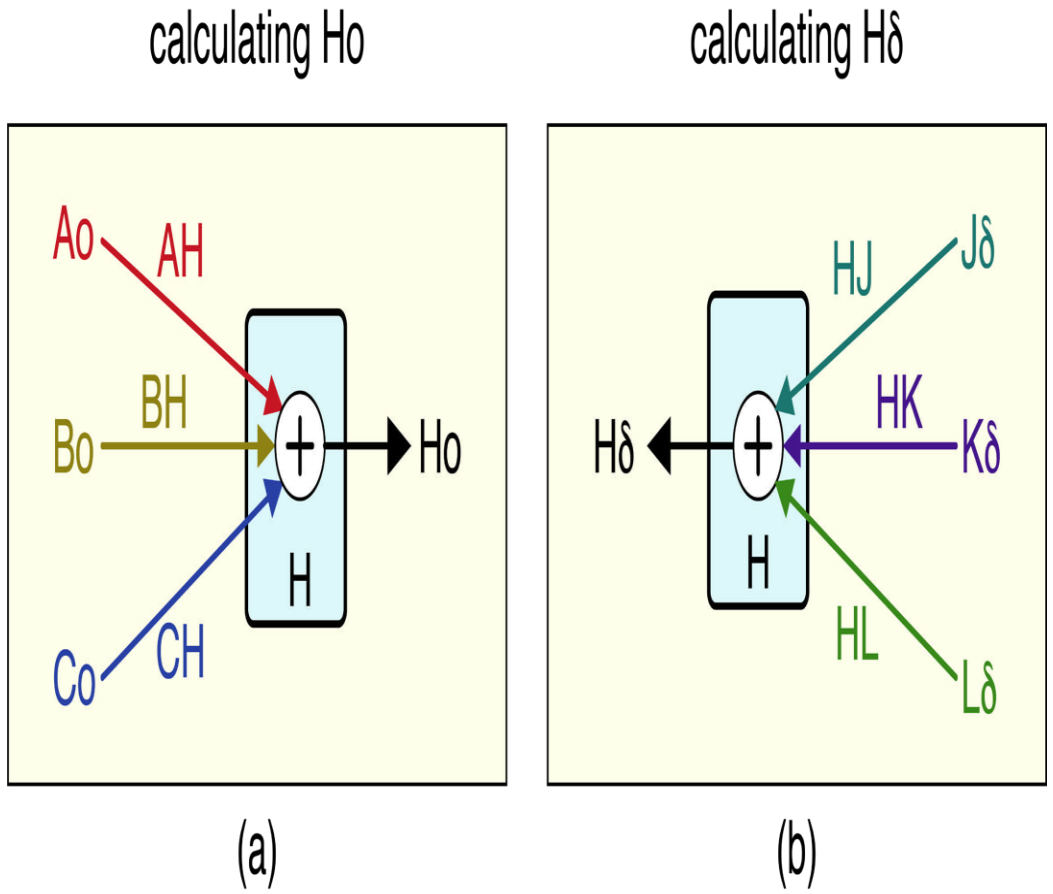
Drawing the values associated with neuron A.

- Draw output Ao as an arrow coming out of the right of the neuron, and the delta Aδ as an arrow coming out of the left.
- The output of A is multiplied by AC on its way to being used by C when we're evaluating a sample.
- The delta of C is multiplied by AC on its to being used by A when we're computing delta values.



Calculating output and delta for a neuron H.

- Left: To calculate Ho, we scale the output of each preceding neuron by the weight of its connection and add the results together.
- Right: To calculate Hδ, we scale the delta of each following neuron by the connection's weight andadd the results together.
- This is pleasingly symmetrical.



Some notes on finding deltas with Backprop

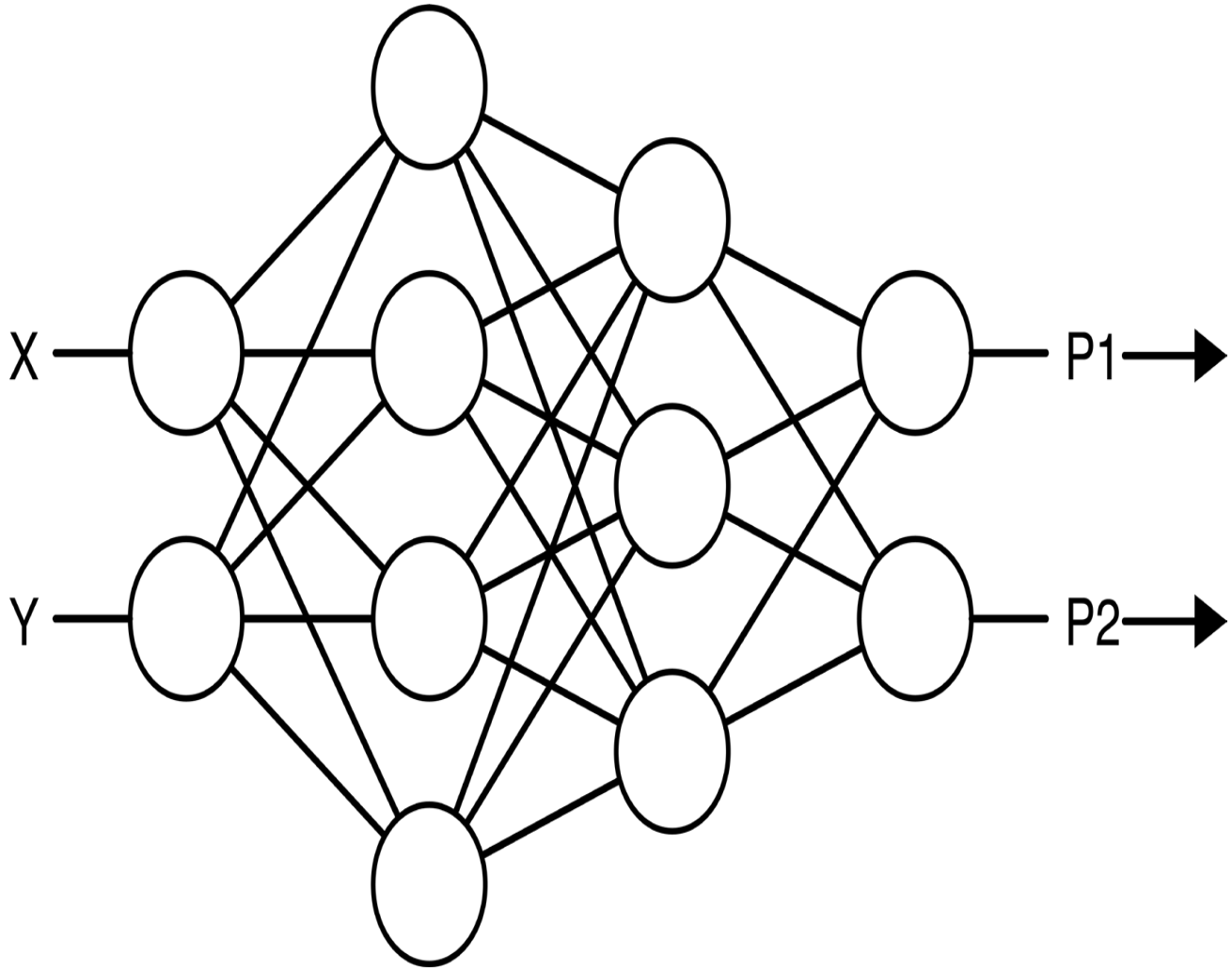
- Calculating delta for a neuron takes the same amount of work (and therefore the same amount of time) as calculating its output.
- So calculating deltas is as efficient as calculating output values.
- Note that Figure doesn't require anything of neuron H except that it has inputs from a preceding layer that travel on connections with weights, and deltas from a following layer that travel on connections with weights.
- So we can apply the left half of Figure, and calculate the output of neuron H as soon as the outputs from the previous layer are available.
- And we can apply the right half of Figure, and calculate the delta of neuron H as soon as the deltas from the following layer are available.
- This also tells us why we had to treat the output layer neurons as a special case: there are no "next layer" deltas to be used.
- This process of finding the delta for every neuron in the network is the backpropagation algorithm.

Some notes on finding deltas with Backprop

- We just saw the backpropagation algorithm, which lets us compute the delta for every neuron that in a network.
- Once all the neuron deltas for any layer (including the output layer) have been found, we can then step back one layer (towards the inputs), and find the deltas all the neurons on that layer, and then step back again, compute all the deltas, step back again, and so on until we reach the input.

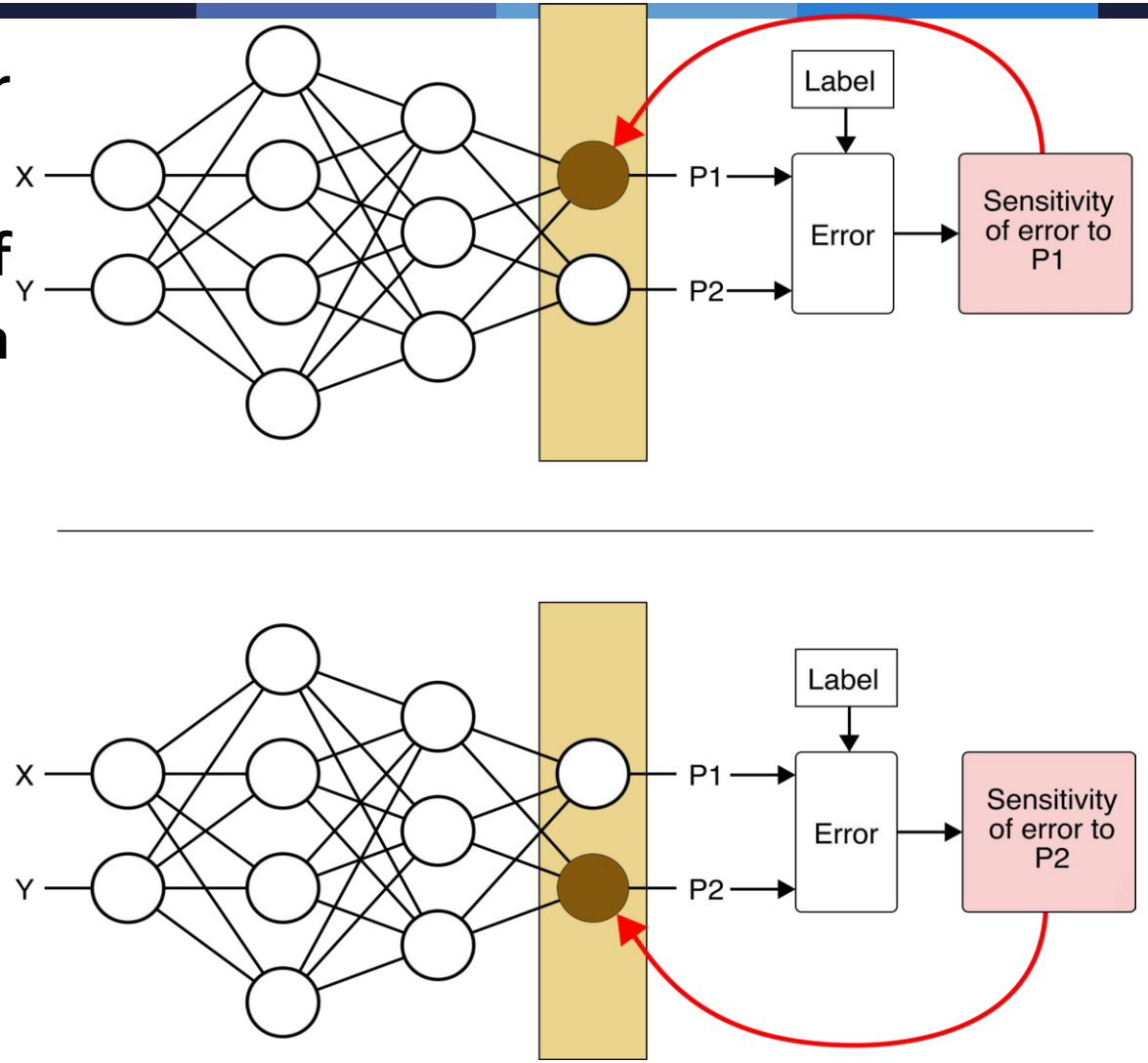
Backprop in Action: Walkthrough example

- A new classifier network with
- 2 inputs,
- outputs,
- hidden layers.



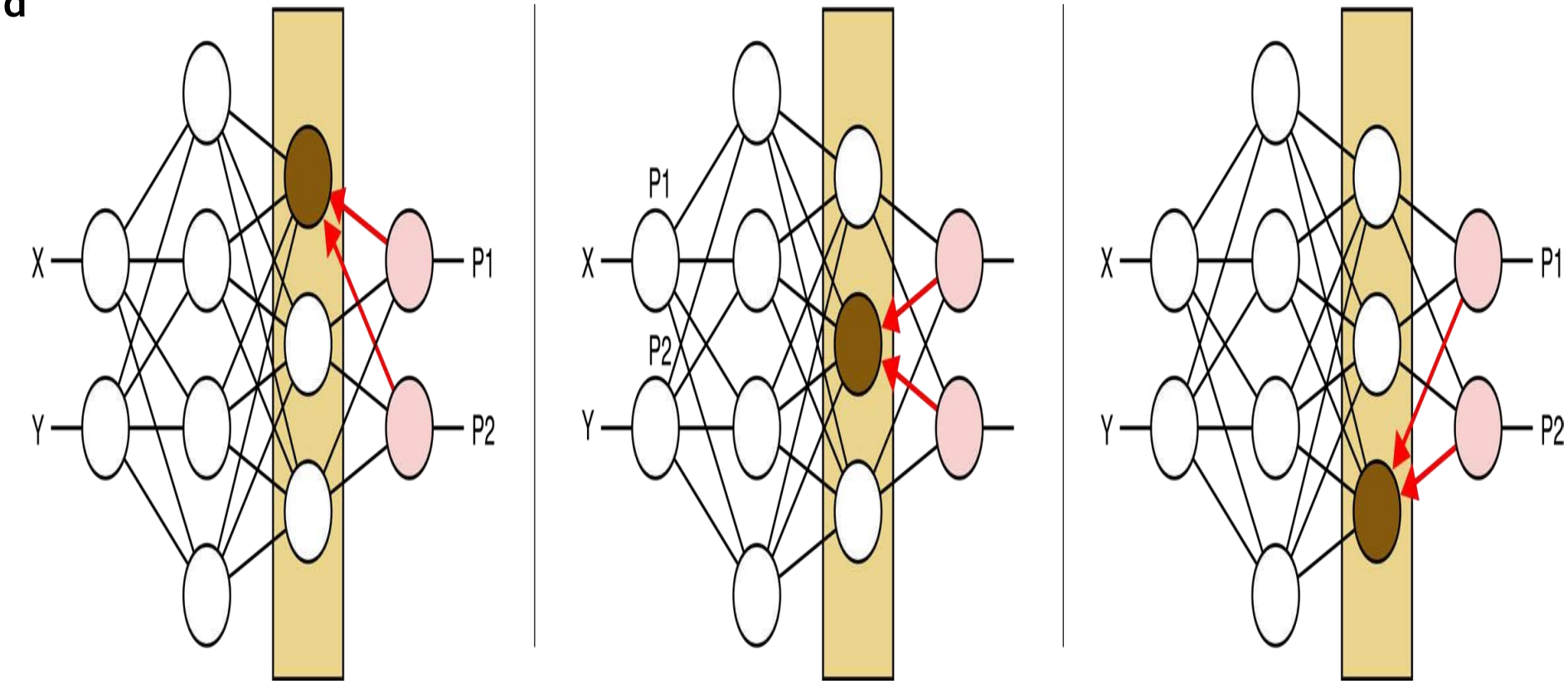
Deltas of Output layer Neurons

- First, find deltas for both output layer neurons.
- If we are using MSE, then sensitivity of error to P1 or P2 is (label value – neuron output)
- So, delta = label value – neuron output



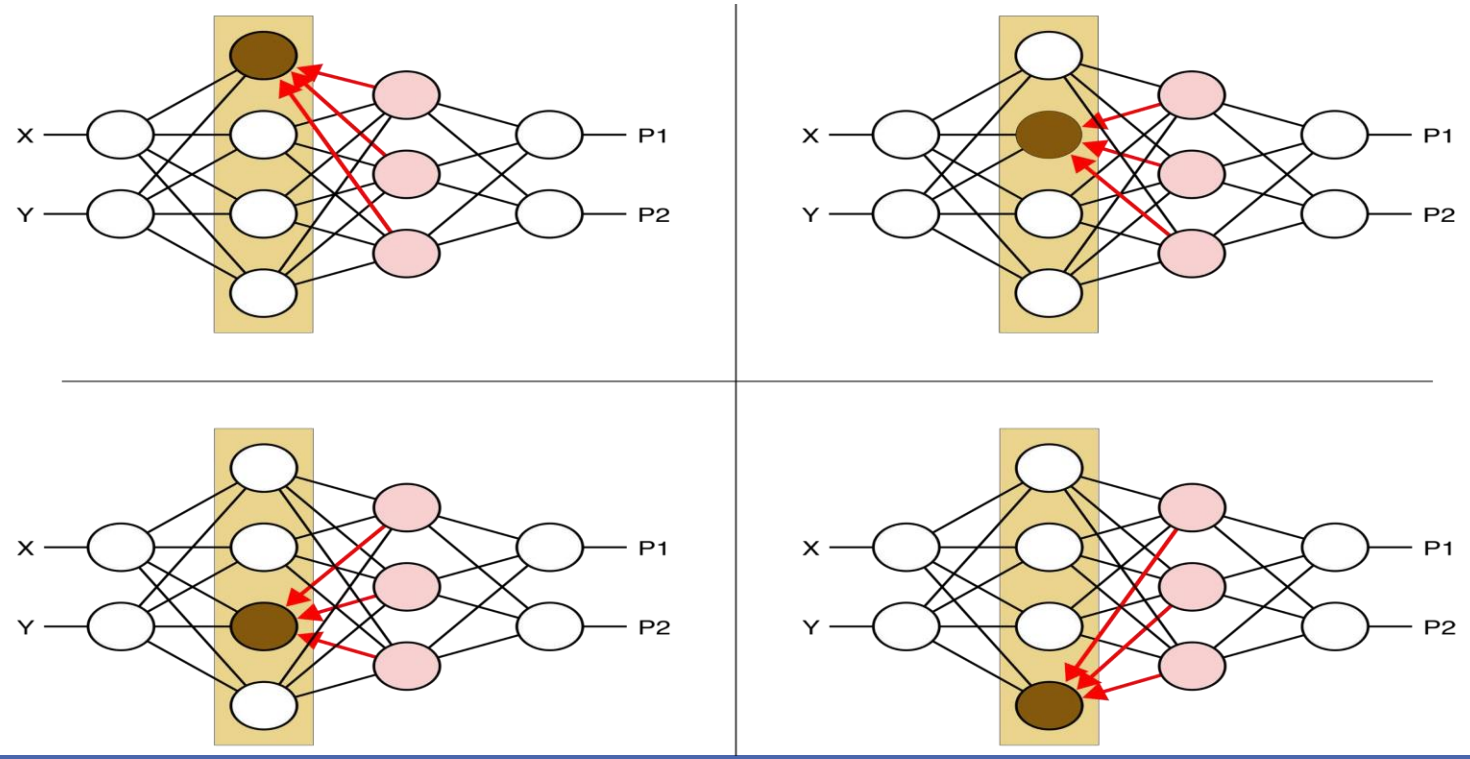
Deltas of Output layer Neurons

- Using backpropagation to find deltas for next-to-output layer (3rd hidden layer) To find the delta for each neuron:
- Find the deltas of the neurons that use its output (outgoing connections), Multiply those deltas by connecting weights, and
- Add results together.



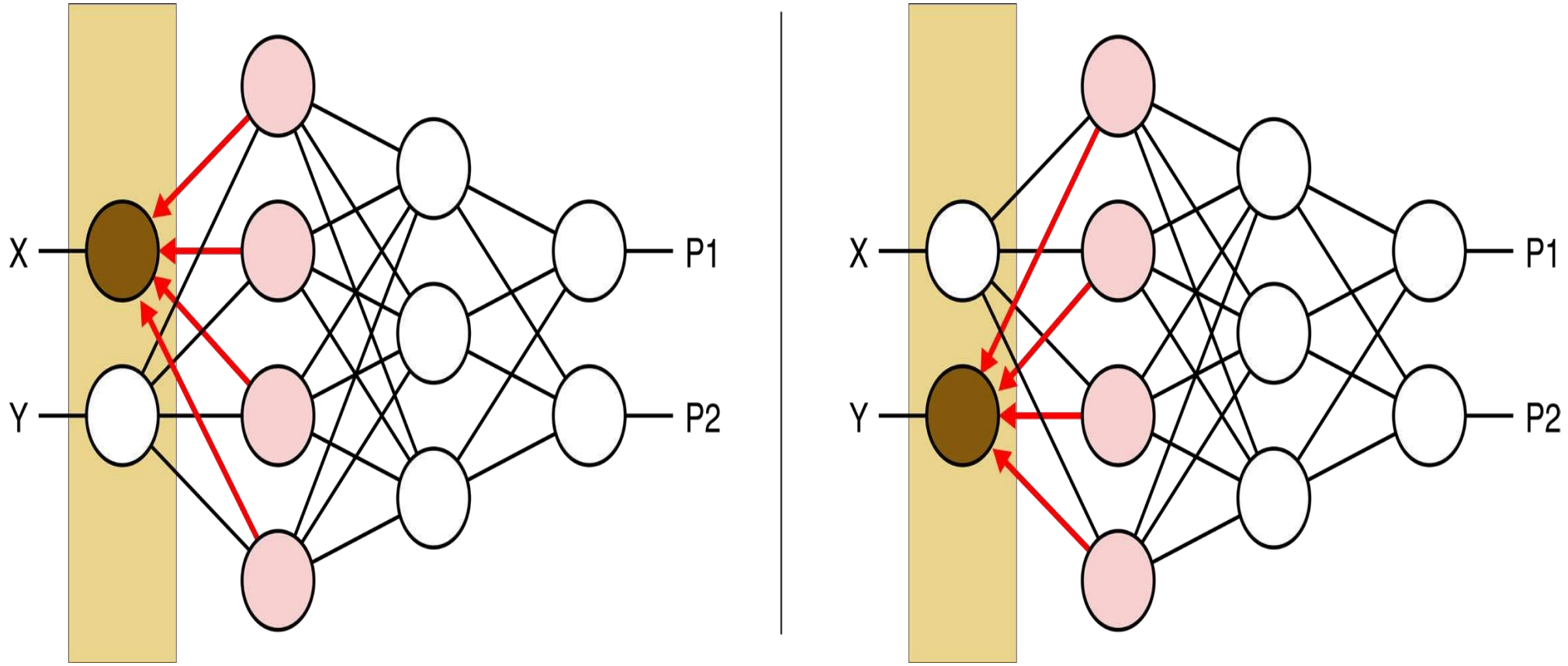
Deltas of Output layer Neurons

- Using backpropagation to find the deltas for (2nd hidden layer). To find delta for each neuron: Find the deltas of the neurons that use its output, Multiply those deltas by connecting weights, & Add results together.
- Exactly the same as for 3rd hidden layer on previous slide



Deltas of Output layer Neurons

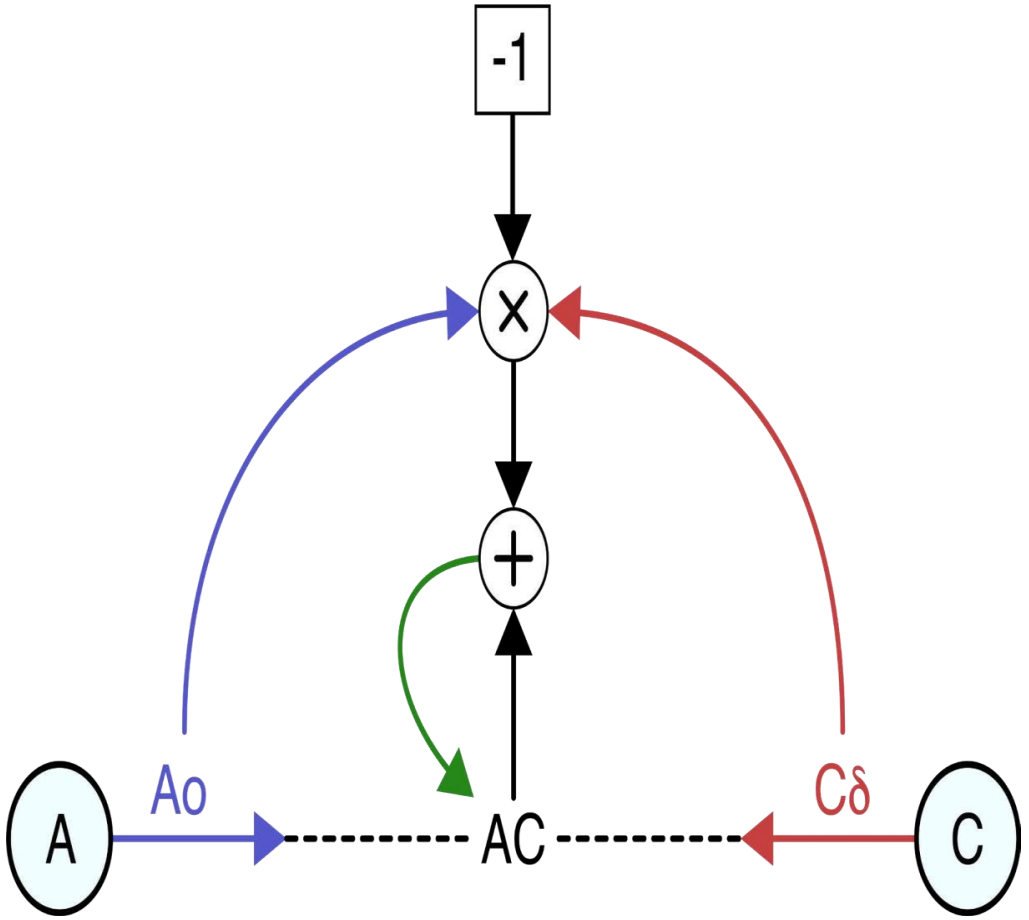
- Using backprop to find the deltas for the first hidden layer.



Update all the weights

- Use the technique in Figure to update every weight to a new and improved value.
- Update weight AC:

$AC = AC - (Ao \times C\delta)$



What is that backwardly propagated?

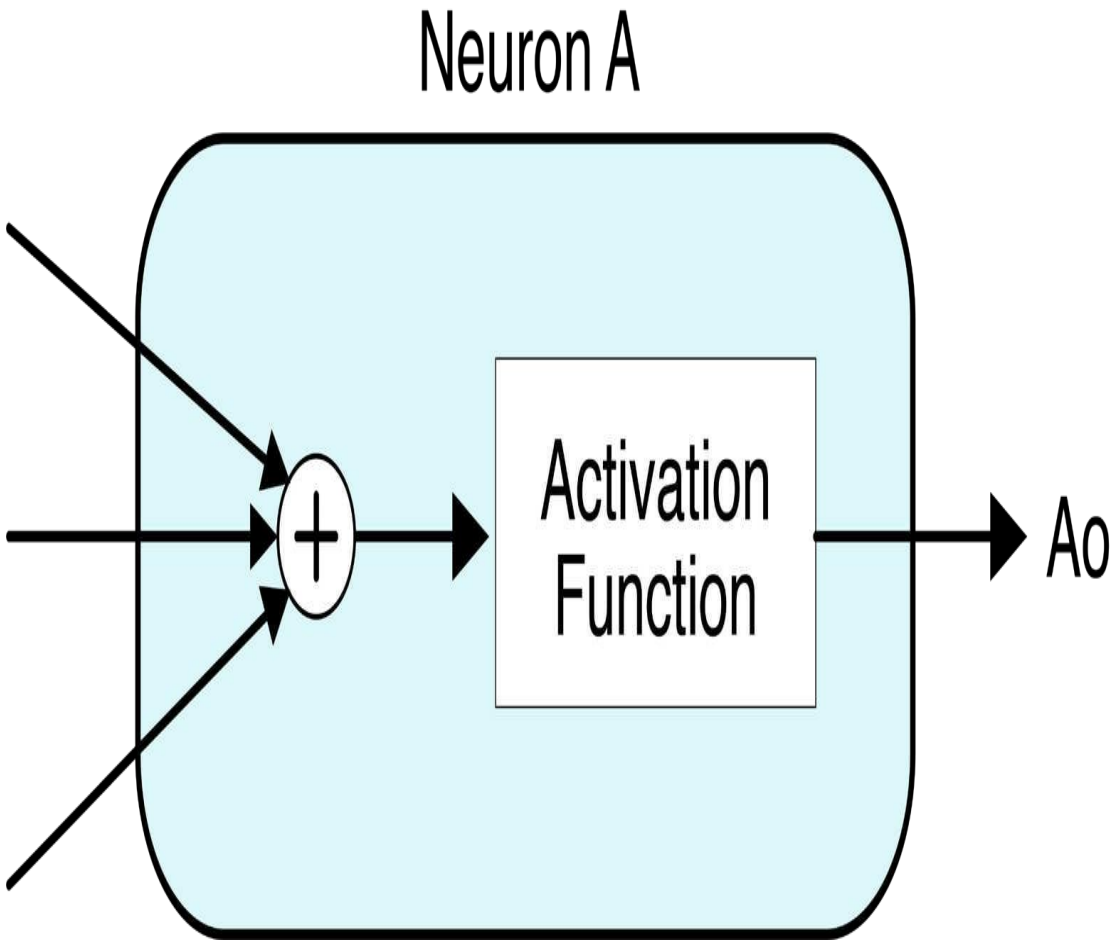
- Previous Figures show why the algorithm is called backwards propagation.
- We're taking the deltas from any layer and propagating, or moving, their information backwards one layer at a time, modifying it as we go.
- Computing each of these delta values is fast. It's just one multiplication per outgoing connection, and then adding those pieces together. That takes almost no time at all.

Parallely compute for entire layer (GPU)?

- Backprop becomes highly efficient when we use parallel hardware like a GPU.
- We can use parallely multiply all the deltas and weights for an entire layer at once, as
  - Neurons on a given layer of a feed-forward network don't interact,
  - Weights and deltas that get multiplied are already computed,
  - Computing an entire layer's worth of deltas in parallel saves us a lot of time.
- Example: If each layer had 100 neurons, and we have 4 layers as in walkthrough example, computing all 400 deltas would take only the same time required to find 4 deltas! (if we have enough GPU cores)

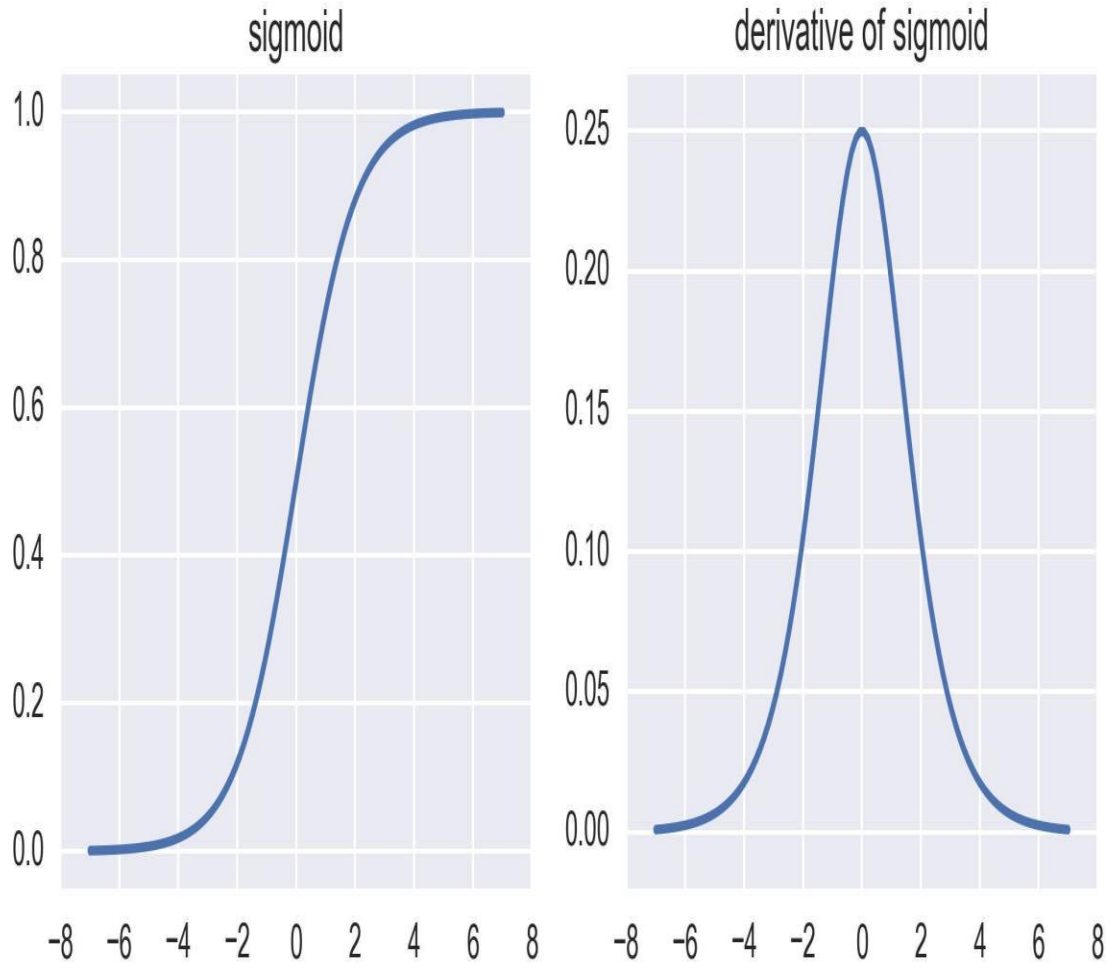
Now lets account for Activation function in backprop

- When we calculate the output of a neuron, the sum of the weighted inputs goes through an activation function before it leaves the neuron, as shown in Figure



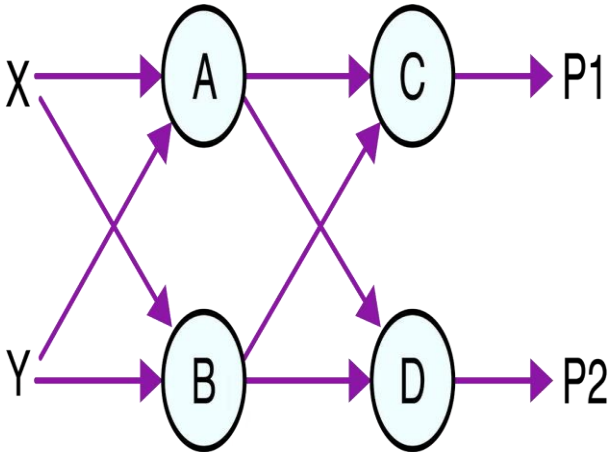
Choose Sigmoidal Activation function

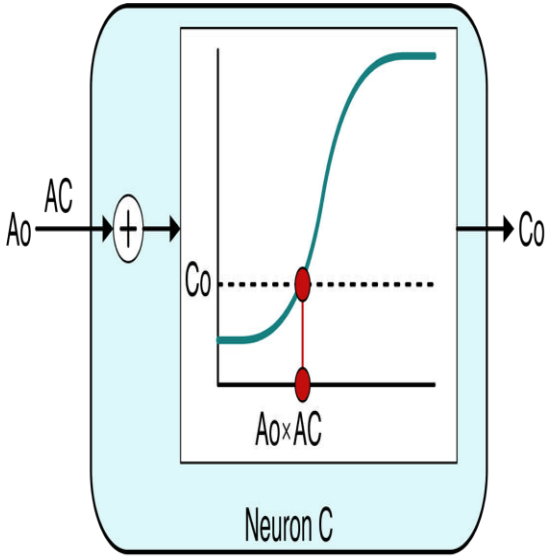
- Let's pick the sigmoid activation function.
- We won't use any particular qualities of the sigmoid, so our discussion will be applicable to any activation function.

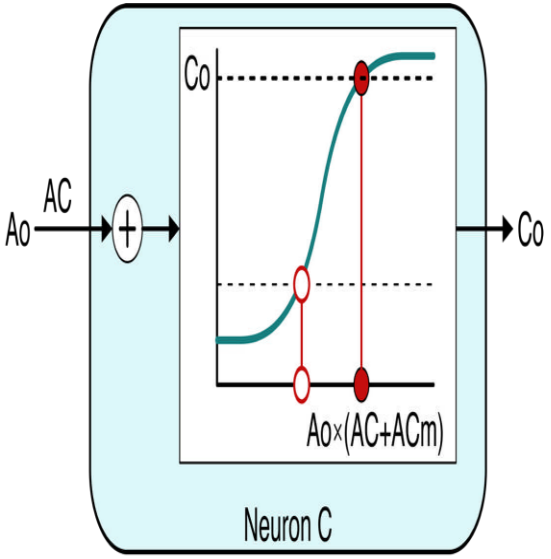


Choose Sigmoidal Activation function

- Consider neuron C of tiny network example (we saw earlier) Consider only input from A to C and ignore input from B to C.
- The input to the activation function = output of A times weight AC. We can find the value of the activation function at that point, which gives us the output Co.
- Left: Output of A is Ao and weight is AC, so value into activation fn is Ao×AC.
- Right: Output of A is Ao and weight is (AC+Acm), so value into activation fn is Ao×(AC+ACm)

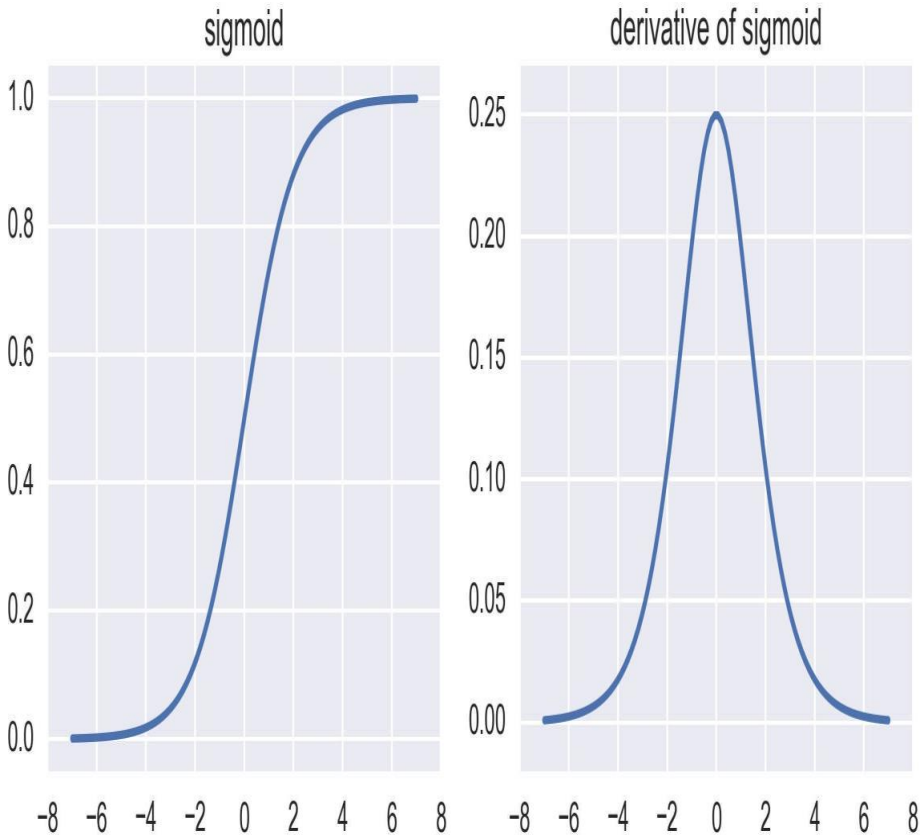






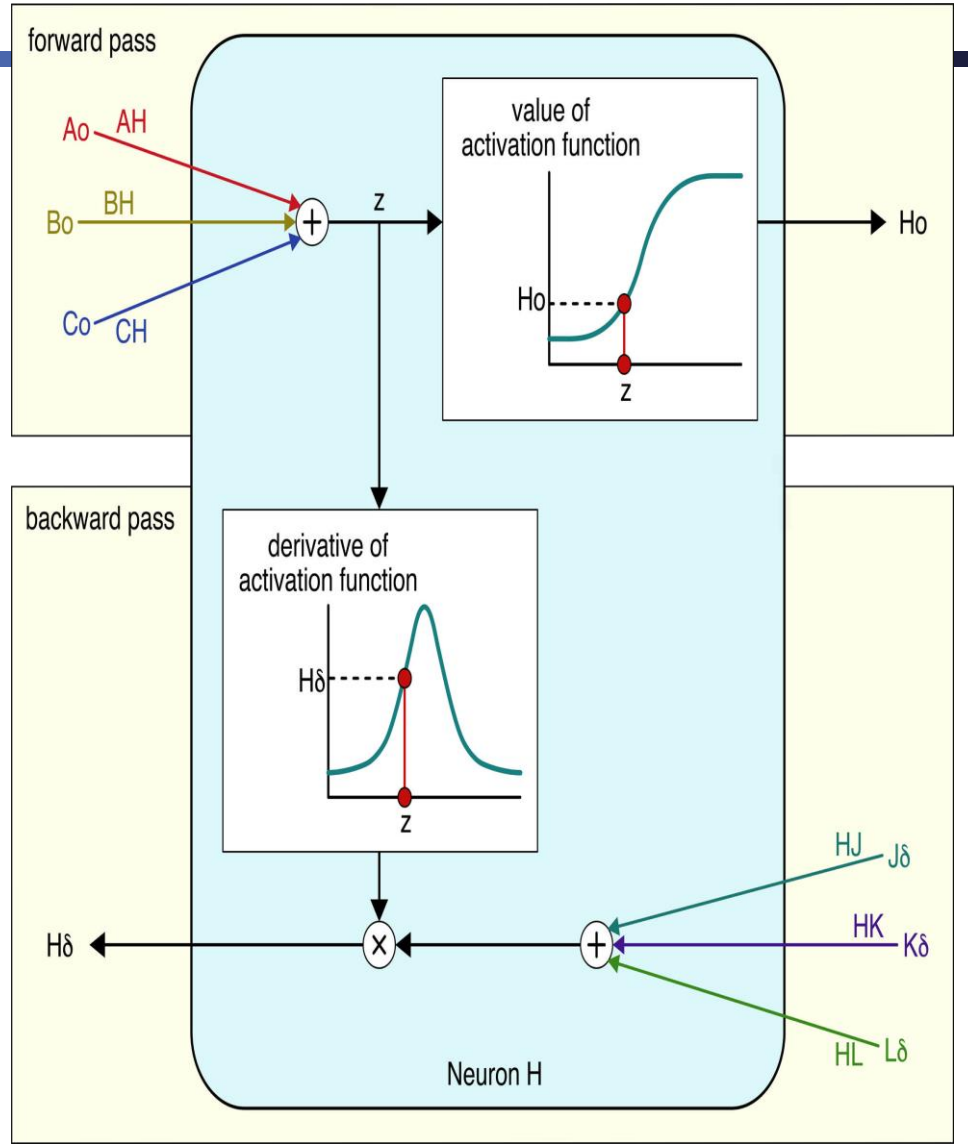
How to account for activation function

- For each neuron:
- Multiply delta by derivative of the activation function, evaluated at the same point used during the forward pass.
- The new delta accounts for the activation function's effect on network error.
- Perform this step immediately after computing the delta.



Network evaluation and backpropagation of deltas in a nutshell

- Top: In the forward pass, the weighted inputs are added together, giving us a value we call z. The value of the activation function at z is our output Ho.
- Bottom: In the backward pass, the weighted deltas are added together, and we use the z from before to look up the derivative of the activation function.
- We multiply the sum of the weighted deltas by this derivative value, giving us Hδ.



Compact and Local calculations !

- Notice how compact and local everything is.

- Forward pass: depends only on the output values of the previous layer, and the weights that connect to them.

- Backward pass: depends only on the deltas from the following layer, the weights that connect to them, and the activation function.

Same goes for other activation functions!

- Everything we did with the sigmoid can be applied to the ReLU, without change.
- Same goes for any other activation function.
- That brings us to the end of basic backpropagation

Learning rate

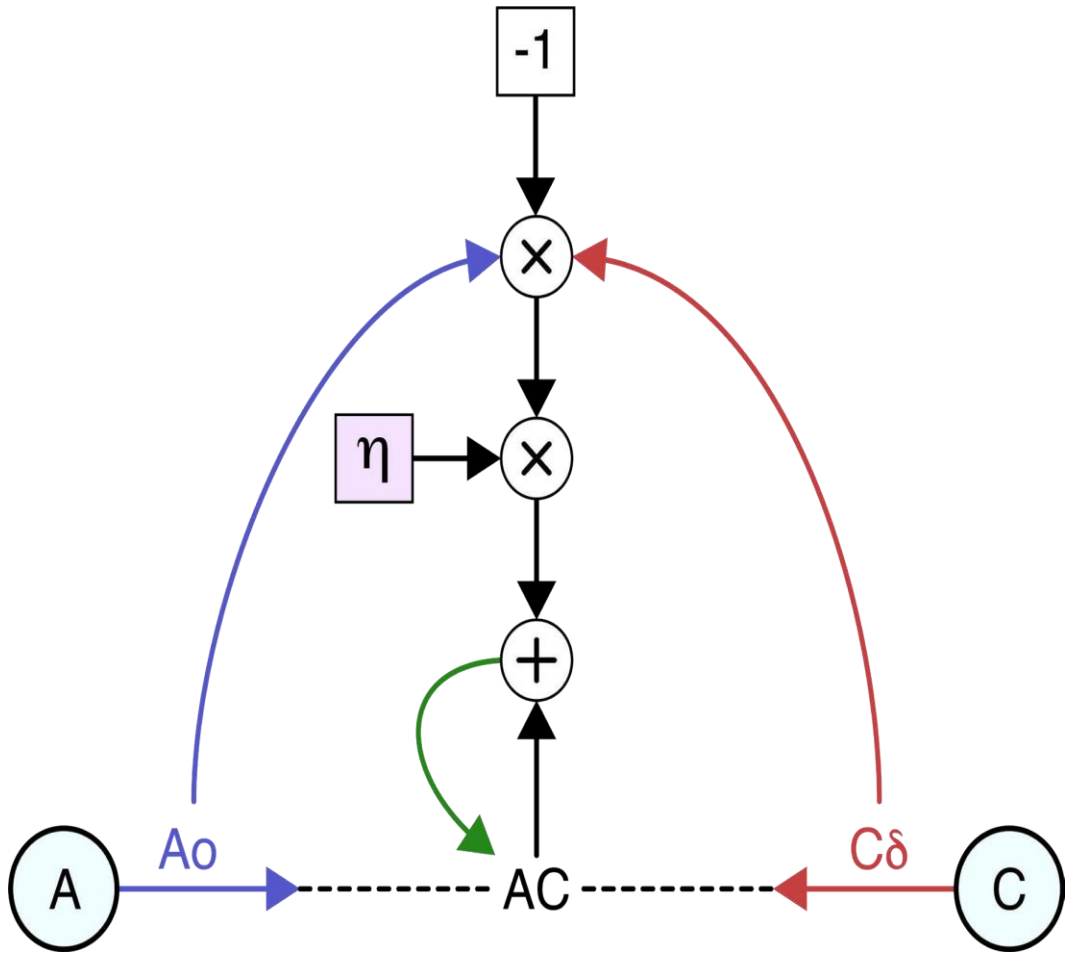
- Changing a weight by a lot in a single step is often a recipe for trouble.
- If we change a weight too much, net overreacts and overshoots, increasing error.
- If we change a weight too little, net does only tiniest bit of learning, slowing everything down.
- In practice, we control the amount of change to the weights during every update with a hyperparameter called the learning rate, η (eta).
- η is a number between 0 and 1, and it tells weights how much of their newly- computed value to use when they update.

Setting learning rate value

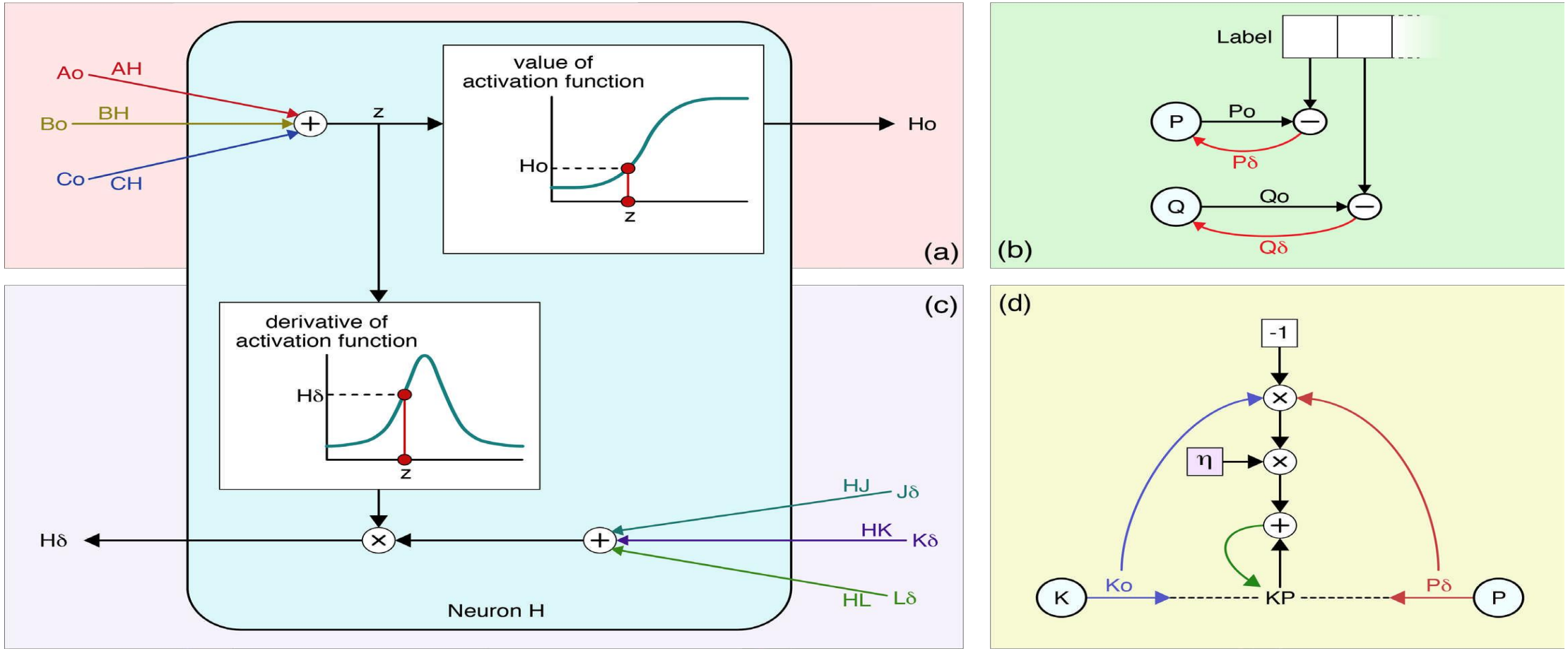
- If we set the learning rate to 0, the weights don't change at all. Our system will never change and never learn.
- If we set the learning rate to 1, the system will apply big changes to the weights, and might overshoot the mark.
  - If this happens a lot, the network can spend its time constantly overshooting and then compensating, with the weights bouncing around and never settling into their best values.
- So, we usually set the learning rate somewhere between these extremes.
- Next figure shows how the learning rate is applied.
  - We just scale the value of $-(Ao \times C\delta)$ by η before adding it back in to AC.

Learning rate controls the weight update

- Here we see an extra step that multiplies the value $-(Ao \times C\delta)$ by the learning rate η before adding it to AC.
- When η is a small positive number (say 0.01), then each change will be small, which often helps the network learn.



Backprop algorithm, along with weight update, in a nutshell.



(a) forward step, (b) finding the deltas for output neurons, (c) propagates the deltas backwards, and (d) updates the weights

Backprop and Gradient Descent

- When we change the weights, we're changing them in order to follow the gradient of the error.
- This is an example of gradient descent
- Backpropagation is an algorithm that efficiently updates our weights using gradient descent, since the deltas it computes describe that gradient.
- So a nice way to summarize backprop is to say that it moves the gradient of the error backwards, modifying it to account for each neuron's contribution.

Gradient Descent-Introduction

- Optimization is a big part of machine learning.
- Gradient descent is a simple optimization procedure that you can use with many machine learning algorithms.
- Batch gradient descent (BGD) refers to calculating the derivative from all training data before calculating an update.
- Stochastic gradient descent (SGD) refers to calculating the derivative from each training data instance and calculating the update immediately.

Convergence to global or local minimum

- When the learning rates decrease with an appropriate rate, and subject to relatively mild assumptions, stochastic gradient descent converges almost surely to a global minimum when the objective function is convex or pseudo-convex, and otherwise converges almost surely to a local minimum.