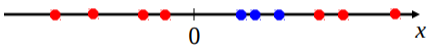
Neural Networks: Motivation

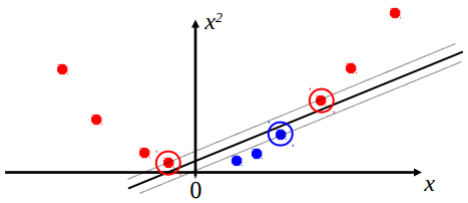
Non-linear Separators

We know how to construct a model that learns a linear boundary for binary classification tasks. This is a powerful technique, and one that works well when the underlying optimal decision boundary is itself linear. However, many practical problems involve the need for decision boundaries that are nonlinear in nature, and our linear perceptron model isn't expressive enough to capture this relationship.

Consider the following set of data:



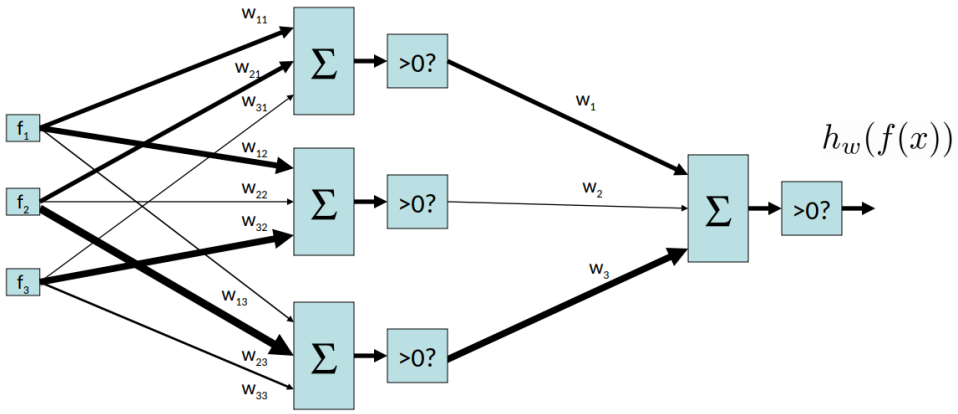
We would like to separate the two colors, and clearly there is no way this can be done in a single dimension (a single dimensional decision boundary would be a point, separating the axis into two regions).
To fix this problem, we can add additional (potentially nonlinear) features to construct a decision boundary from. Consider the same dataset with the addition of x^2 as a feature:



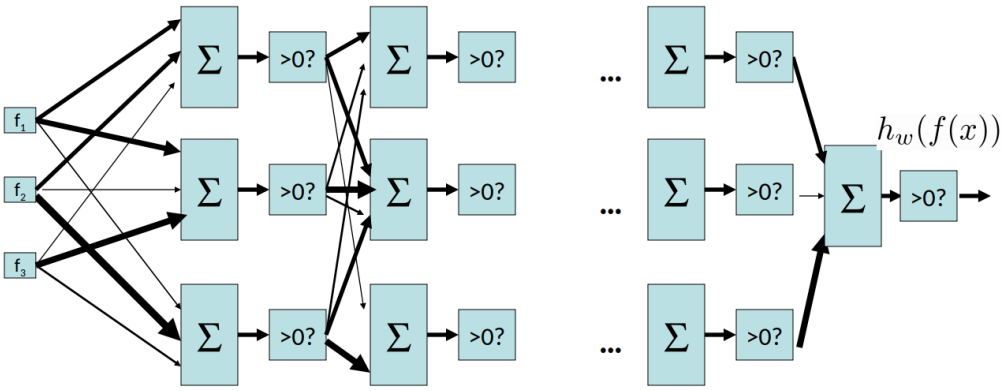
With this additional piece of information, we are now able to construct a linear separator in the two dimensional space containing the points. In this case, we were able to fix the problem by mapping our data to a higher dimensional space by manually adding useful features to datapoints. However, in many high-dimensional problems, such as image classification, manually selecting features that are useful is a tedious problem. This requires domain-specific effort and expertise, and works against the goal of generalization across tasks. A natural desire is to learn these featurization or transformation functions as well, perhaps using a nonlinear function class that is capable of representing a wider variety of functions.

Multi-layer Perceptron

Let's examine how we can derive a more complex function from our original perceptron architecture. Consider the following setup, a two-layer perceptron, which is a perceptron that takes as input the outputs of another perceptron.



In fact, we can generalize this to an N-layer perceptron:



With this additional structure and weights, we can express a much wider set of functions.

By increasing the complexity of our model, we in turn greatly increase its expressive power. Multi-layer perceptrons give us a generic way to represent a much wider set of functions. In fact, a multi-layer perceptron is a **universal function approximator** and can represent *any* real function, leaving us only with the problem of selecting the best set of weights to parameterize our network. This is formally stated below:

Theorem. (Universal Function Approximators) A two-layer neural network with a sufficient number of neurons can approximate any continuous function to any desired accuracy.

Measuring Accuracy

The accuracy of the binary perceptron after making m predictions can be expressed as:

$$l^{acc}(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m}(\text{sgn}(\boldsymbol{w}\cdot\mathbf{f}(x^{(i)})) == y^{(i)})$$

where $x^{(i)}$ is datapoint i, $\mathbf{w}$ is our weight vector, $\mathbf{f}$ is our function that derives a feature vector from a raw datapoint, and $y^{(i)}$ is the actual class label of $x^{(i)}$. In this context, $\text{sgn}(x)$ represents an **indicator function**, which evaluates to 0 when x is negative, and 1 when x is positive. Taking this notation into account, we can note that our accuracy function above is equivalent to dividing the total number of *correct* predictions by the raw total number of predictions.

Sometimes, we want an output that is more expressive than a binary label. It then becomes useful to produce a probability for each of the N classes we want to classify into, which reflects our a degree of certainty that the datapoint belongs to each of the possible classes. To do so, we transition from storing a single weight vector to storing a weight vector for *each* class j, and estimate probabilities with the **softmax function** $\sigma(x)$. The softmax function defines the probability of classifying $x^{(i)}$ to class j as:

$$\sigma(x^{(i)})_j = \frac{e^{f(x^{(i)})^T w_j}}{\sum_{k=1}^{N} e^{f(x^{(i)})^T w_k}} = P(y^{(i)} = j|x^{(i)})$$

Given a vector that is output by our function f, softmax performs normalization to output a probability distribution. To come up with a general loss function for our models, we can use this probability distribution to generate an expression for the likelihood of a set of weights:

$$\ell(\boldsymbol{w}) = \prod_{i=1}^{m} P(y^{(i)}|x^{(i)};\boldsymbol{w})$$

This expression denotes the likelihood of a particular set of weights explaining the observed labels and datapoints. We would like to find the set of weights that maximizes this quantity. This is identical to finding the maximum of the log-likelihood expression (since log is an increasing function, the maximizer of one will be the maximizer of the other):

$$\ell\ell(\boldsymbol{w}) = \log\prod_{i=1}^{m} P(y^{(i)}|x^{(i)};\boldsymbol{w}) = \sum_{i=1}^{m}\log P(y^{(i)}|x^{(i)};\boldsymbol{w})$$

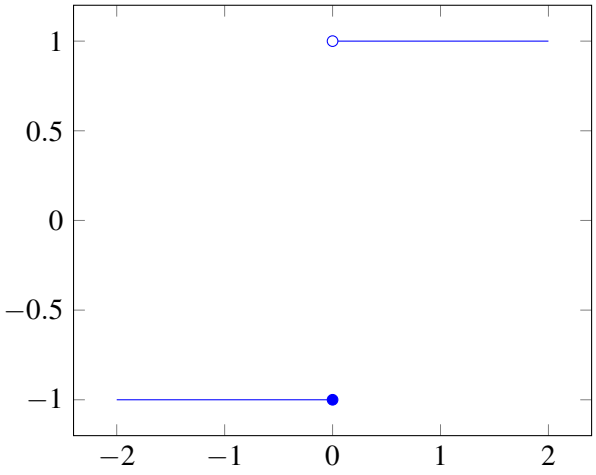
(Depending on the application, the formulation as a sum of log probabilities may be more useful – for example in mini-batched or stochastic gradient descent; see the *Neural Networks: Optimization* section below.) In the case where the log likelihood is differentiable with respect to the weights, we will discuss a simple algorithm to optimize it.

Multi-layer Feedforward Neural Networks

We now introduce the idea of an (artificial) neural network. This is much like the multi-layer perceptron, however, we choose a different non-linearity to apply after the individual perceptron nodes. Note that it is these added non-linearities that makes the network as a whole non-linear and more expressive (without them, a multi-layer perceptron would simply be a composition of linear functions and hence also linear). In the case of a multi-layer perceptron, we chose a step function:

$$f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

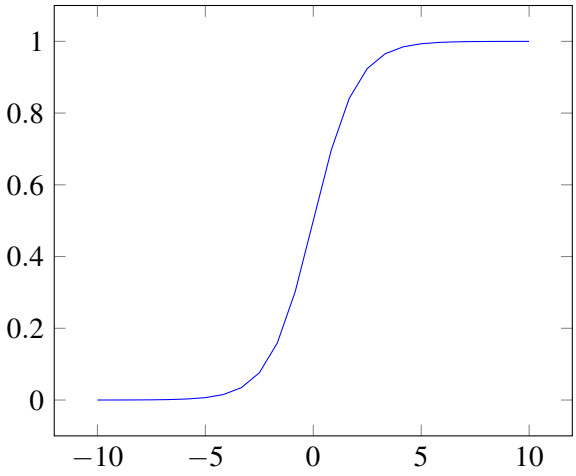
Let's take a look at its graph:



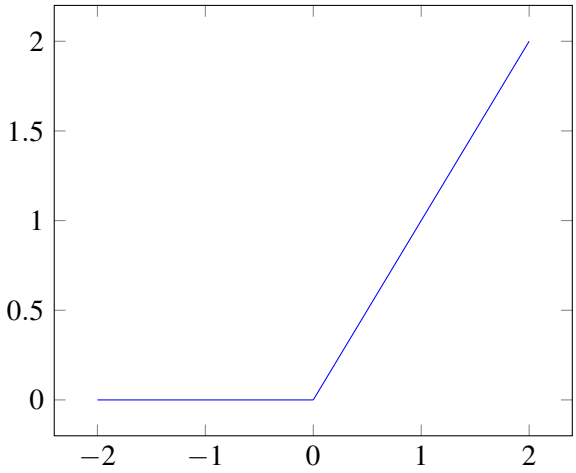
This is difficult to optimize for a number of reasons which will hopefully become clearer when we address gradient descent. Firstly, it is not continuous, and secondly, it has a derivative of zero at all points. Intuitively, this means that we cannot know in which direction to look for a local minima of the function, which makes it difficult to minimize loss in a smooth way.

Instead of using a step function like above, a better solution is to select a continuous function. We have many options for such a function, including the **sigmoid function** (named for the Greek σ or 's' as it looks like an 's') as well as the **rectified linear unit** (ReLU). Let's look at their definitions and graphs below:

Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$



$$\textit{ReLU}: f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$



Calculating the output of a multi-layer perceptron is done as before, with the difference that at the output of each layer we now apply one of our new non-linearities (chosen as part of the architecture for the neural network) instead of the initial indicator function. In practice, the choice of nonlinearity is a design choice that typically requires some experimentation to select a good one for each individual use case.

Loss Functions and Multivariate Optimization

Now we have a sense of how a feed-forward neural network is constructed and makes its predictions, we would like to develop a way to train it, iteratively improving its accuracy, similarly to how we did in the case of the perceptron. In order to do so, we will need to be able to measure their performance. Returning to our log-likelihood function that we wanted to maximize, we can derive an intuitive algorithm to optimize our weights given that our function is differentiable.

Gradient Ascent / Descent

To maximize our log-likelihood function, we differentiate it to obtain a **gradient vector** consisting of its partial derivatives for each parameter:

$$\nabla_w \ell\ell(\mathbf{w}) = \left[\frac{\partial \ell\ell(\mathbf{w})}{\partial \mathbf{w}_1}, \ldots, \frac{\partial \ell\ell(\mathbf{w})}{\partial \mathbf{w}_n} \right]$$

This gradient vector gives the local direction of steepest ascent (or descent if we reverse the vector). **Gradient ascent** is a greedy algorithm that calculates this gradient for the current values of the weight parameters, then updates the parameters along the direction of the gradient, scaled by a **step size**, α. Specifically the algorithm looks as follows:

Initialize weights $\boldsymbol{w}$
For i = 0, 1, 2, ...
  $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \nabla_w \ell\ell(\boldsymbol{w})$

If rather than minimizing we instead wanted to minimize a function f, the update should subtract the scaled gradient ($\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} f(\mathbf{w})$) – this gives the **gradient descent** algorithm.

Neural Networks: Optimization

Now that we have a method for computing gradients for all parameters of the network, we can use gradient descent methods to optimize the parameters to get high accuracy on our training data. For example, suppose we have designed some classification network to output probabilities of classes y for data points x, and have m different training datapoints (see the *Measuring Accuracy* section for more on this). Let $\mathbf{w}$ be all the parameters of our network. We want to find values for the parameters $\mathbf{w}$ that maximize the likelihood of the true class probabilities for our data, so we have the following function to run gradient ascent on:

$$\ell\ell(\mathbf{w}) = \log \prod_{i=1}^{m} P(y^{(i)} \mid x^{(i)}; \mathbf{w}) = \sum_{i=1}^{m} \log P(y^{(i)} \mid x^{(i)}; \mathbf{w})$$

where $x^{(1)}, \ldots, x^{(m)}$ are the m datapoints in our training set.

One way to try to minimize the negative of log likelihood is, at each iteration of gradient descent, to use all the data points $x^{(1)}, \ldots, x^{(m)}$ to compute gradients for the parameters $\mathbf{w}$, update the parameters, and repeat until the parameters converge (at which point we've reached a local minimum of the function).

This technique, known as **batch gradient descent**, is rarely done in practice, since datasets are typically large enough that computing gradients for this full likelihood function will be very slow. Instead, we'll typically use **mini-batching**. Mini-batching rotates through randomly sampled batches of k data points at a time, taking one batch for each step of gradient descent and computing gradients of the loss function using only that batch (so that the sum above is over the k datapoints in the batch, rather than all m datapoints in the training set). This allows us to compute each gradient update much more quickly, and often still makes fast progress toward the minimum of the function. The limit where the batch size $k = 1$ is known as **stochastic gradient descent (SGD)**. In SGD, we randomly sample a single example from the training dataset at each step of gradient descent, compute parameter gradients using the network's loss on that single example, update the parameters, and repeat (sampling another example from the training set).

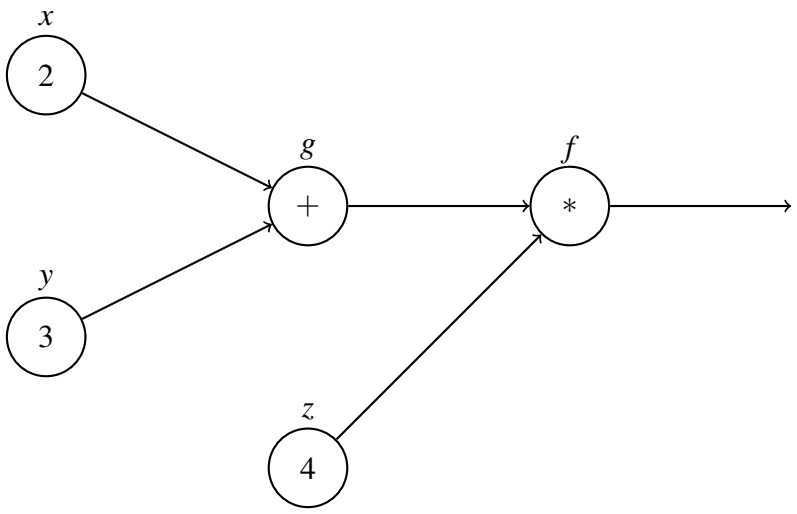
Neural networks are powerful (and universal!) function approximators, but can be difficult to design and train. There's a lot of ongoing research in deep learning focusing on various aspects of neural network design such as :

1. *Network Architectures* - designing a network (choosing activation functions, number of layers, etc.) that's a good fit for a particular problem
2. *Learning Algorithms* - how to find parameters that achieve a low value of the loss function, a difficult problem since gradient descent is a greedy algorithm and neural nets can have many local optima
3. *Generalization and Transfer Learning* - since neural nets have many parameters, it's often easy to overfit training data - how do you guarantee that they also have low loss on testing data you haven't seen before?

Neural Networks: Backpropagation

To efficiently calculate the gradients for each parameter in a neural network, we will use an algorithm known as **backpropagation**. Backpropagation represents the neural network as a dependency graph of operators and operands, called a **computational graph**, such as the one shown below:

The graph structure allows us to efficiently compute both the network's error (loss) on input data, as well as the gradients of each parameter with respect to the loss. These gradients can be used in gradient descent to adjust the network's parameters and minimize the loss on the training data.



The Chain Rule

The chain rule is the fundamental rule from calculus which both motivates the usage of computation graphs and allows for a computationally feasible backpropagation algorithm. Mathematically, it states that for a variable z which is a function of n variables $x_1, \ldots, x_n$ and each x_i is a function of m variables $t_1, \ldots, t_m$, then we can compute the derivative of z with respect to any t_i as follows:

$$\frac{\partial f}{\partial t_i} = \frac{\partial f}{x_1} \cdot \frac{\partial x_1}{\partial t_i} + \frac{\partial f}{x_2} \cdot \frac{\partial x_2}{\partial t_i} + \ldots + \frac{\partial f}{x_n} \cdot \frac{\partial x_n}{\partial t_i}$$

In the context of computation graphs, this means that to compute the gradient of a given node t_i with respect to the output z, we take a sum of children(t_i) terms.

The Backpropagation Algorithm

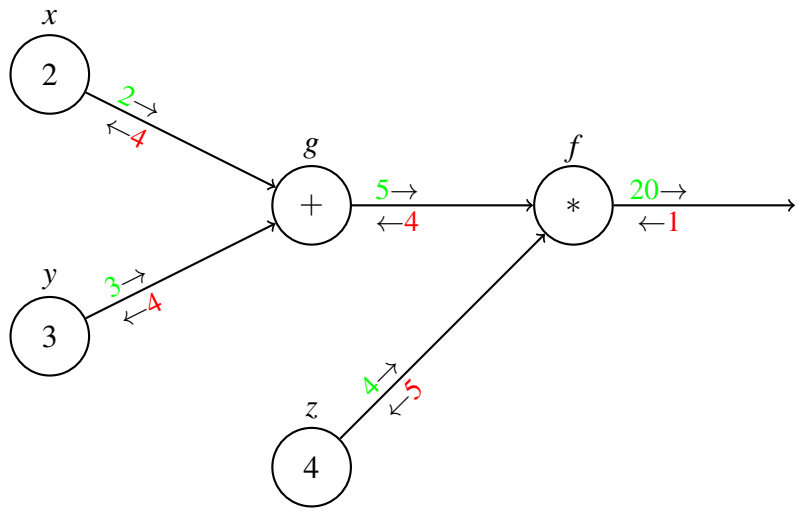


Figure 1: A computation graph for computing $(x+y)*z$ with the values $x=2$, $y=3$, $z=4$.

Figure 1 shows an example computation graph for computing $(x+y)*z$ with the values $x=2$, $y=3$, $z=4$. We will write $g=x+y$ and $f=g*z$. Values in green are the outputs of each node, which we compute in the **forward pass**, where we apply each node's operation to its input values coming from its parent nodes.

Values in red after each node give gradients of the function computed by the graph, which are computed in the **backward pass**: the value after each node is the partial derivative of the last node f value with respect to the variable at that node. For example, the red value 4 after g is $\frac{\partial f}{\partial g}$, and the red value 4 after x is $\frac{\partial f}{\partial x}$. In our simple example, f is just a multiplication node which outputs the product of its two input operands, but in a real neural network the final node will usually compute the loss value that we are trying to minimize.

The backward pass computes gradients by starting at the final node (which has a gradient of 1 since $\frac{\partial f}{\partial f}=1$) and passing and updating gradients backward through the graph. Intuitively, each node's gradient measures how much a change in that node's value contributes to a change in the final node's value. This will be the product of how much the node contributes to a change in its child node, with how much the child node contributes to a change in the final node. Each node receives and combines gradients from its children, updates this combined gradient based on the node's inputs and the node's operation, and then passes the updated gradient backward to its parents. Computation graphs are a great way to visualize repeated application of the chain rule from calculus, as this process is required for backpropagation in neural networks.

Our goal during backpropagation is to determine the gradient of output with respect to each of the inputs. As you can see in Figure 1, in this case we want to compute the gradients $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$, and $\frac{\partial f}{\partial z}$:

1. Since f is our final node, it has gradient $\frac{\partial f}{\partial f}=1$. Then we compute the gradients for its children, g and z. We have $\frac{\partial f}{\partial g}=\frac{\partial}{\partial g}(g\cdot z)=z=4$, and $\frac{\partial f}{\partial z}=\frac{\partial}{\partial z}(g\cdot z)=g=5$.

2. Now we can move on upstream to compute the gradients of x and y. For these, we'll use the chain rule and reuse the gradient we just computed for g, $\frac{\partial f}{\partial g}$.

3. For x, we have $\frac{\partial f}{\partial x}=\frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$ by the chain rule – the product of the gradient coming from g with the partial derivative for x at this node. We have $\frac{\partial g}{\partial x}=\frac{\partial}{\partial x}(x+y)=\frac{\partial}{\partial x}x+\frac{\partial}{\partial x}y=1+0$, so $\frac{\partial f}{\partial x}=4\cdot 1=4$. Intuitively, the amount that a change in x contributes to a change in f is the product of the amount that a change in g contributes to a change in f, with the amount that a change in x contributes to one in g.

4. The process for computing the gradient of the output with respect to y is almost identical. For y we have $\frac{\partial f}{\partial y}=\frac{\partial f}{\partial g}\frac{\partial g}{\partial y}$ by the chain rule – the product of the gradient coming from g with the partial derivative for y at this node. We have $\frac{\partial g}{\partial y}=\frac{\partial}{\partial y}(x+y)=\frac{\partial}{\partial y}x+\frac{\partial}{\partial y}y=0+1$, so $\frac{\partial f}{\partial y}=4\cdot 1=4$.

Since the backward pass step for a node in general depends on the node's inputs (which are computed in the forward pass), and gradients computed "downstream" of the current node by the node's children (computed earlier in the backward pass), we cache all of these values in the graph for efficiency. Taken together, the forward and backward pass over the graph make up the backpropagation algorithm.

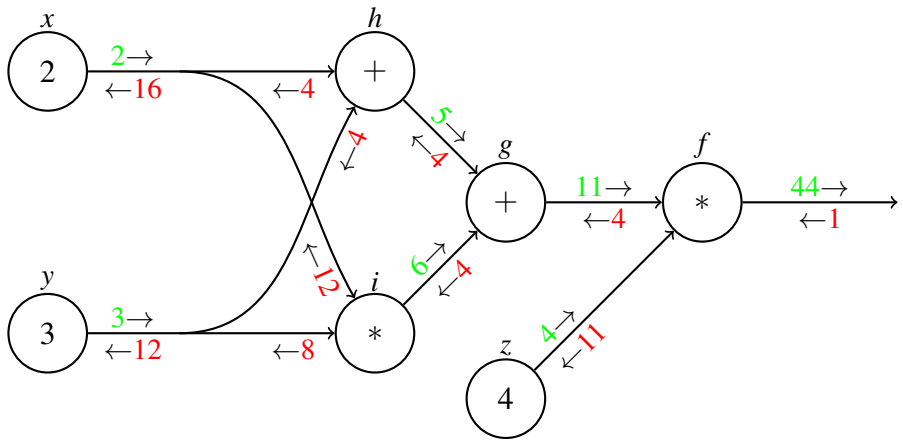


Figure 2: A computation graph for computing $((x+y)+(x\cdot y))\cdot z$, with $x=2$, $y=3$, $z=4$.

For an example of applying the chain rule for a node with multiple children, consider the graph in Figure 2, representing $((x+y)+(x\cdot y))\cdot z$, with $x=2$, $y=3$, $z=4$. x and y are each used in 2 operations, and so each has two children. By the chain rule, their gradient values are the sum of the gradients computed for them by their children (i.e. gradient values add at path junctions). For example, to compute the gradient for x, we have

$$\frac{\partial f}{\partial x}=\frac{\partial f}{\partial h}\frac{\partial h}{\partial x}+\frac{\partial f}{\partial i}\frac{\partial i}{\partial x}=4\cdot 1+4\cdot 3=4+12=16$$