# 05 - Deep FeedForward Neural Networks

François Pitié

Ussher Assistant Professor in Media Signal Processing
Department of Electronic & Electrical Engineering, Trinity College Dublin

# FeedForward Neural Network

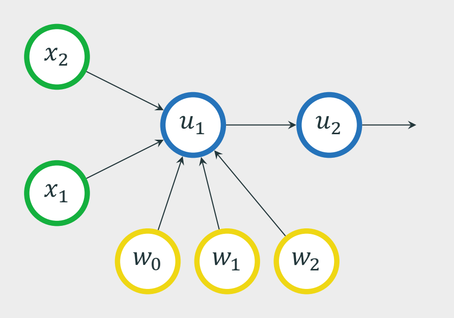
Remember Logistic Regression? The output of the model was

$$h_{\mathbf{w}}(\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}}}$$

This was your first neural network.

Why is it a Network?

For a logistic regression model with two variables, we can represent the model as a network of operations as follows:




$$
\begin{aligned}
u_1 &= w_0 + w_1 x_1 + w_2 x_2 \\
u_2 &= f(u_1) \\
&= \frac{1}{1 + e^{-w_0 - w_1 x_1 - w_2 x_2}}
\end{aligned}
$$

Neural Networks are called networks because they can be associated with a **directed acyclic graph** describing how the functions are composed together.

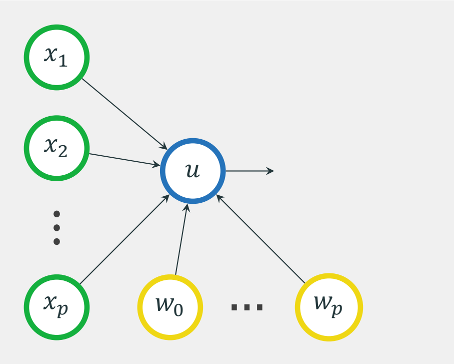
Each node in the graph is called a **unit**.

The starting units (leaves of the graph) correspond either to input values (eg. $x_1$ , $x_2$ ), or model parameters (eg. $w_0$ , $w_1$ , $w_2$ ). All other units (eg. $u_1$ , $u_2$ ) correspond to function outputs.

# Neurons

Why is it a Neural Network?

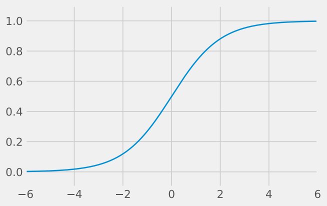
Because it mainly relies on neuron-like units. A **neuron unit** first takes a linear combination of its inputs and then applies a non-linear function $f$, called **activation function**:



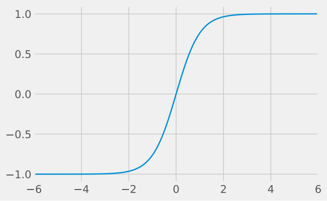

$$u = f(w_0 + \sum_{i=1}^{p} w_i x_i)$$

# Activation Functions

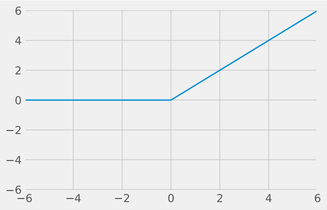
Many activation functions exist. Here are the most popular:




Sigmoid: $z = 1/(1 + \exp(-z))$




tanh: $z = \tanh(z)$




ReLU: $f(z) = \max(0, z)$

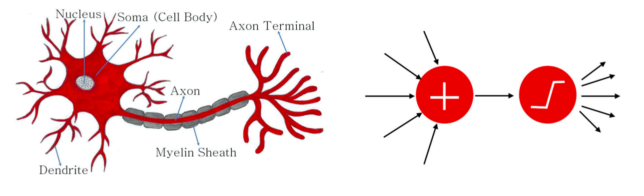
# Activation Functions

Whilst the most frequently used activation functions are **ReLU**, **sigmoid** and **tanh**, many more types of activation functions are possible.

However they tend to perform roughly comparably to these known types. Thus, unless there is a compelling to use more exotic functions, we stick to these known types.

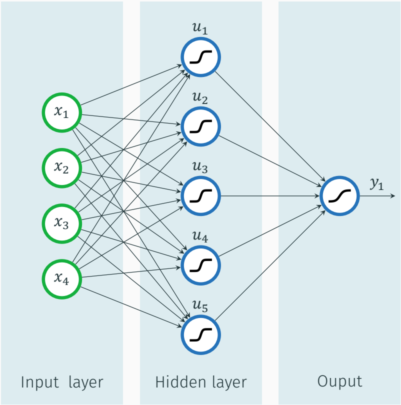
# Neurons

Artificial neurons were originally aiming at modelling biological neurons. We know for instance that the input signals from the dendrites are indeed combined together to go through something resembling a ReLU activation function.




The aim has now diverged from that goal. We are now purely interested in finding models that produce best results in Machine Learning tasks.

Now that we have a neuron, we can combine multiple units to form a **feedforward neural network**.

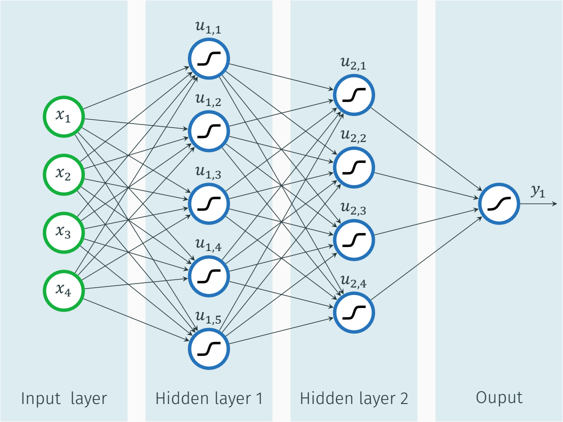



Each blue unit is a neuron with its activation function.

Any node that is not an input or output node is called **hidden** unit. Think of them as intermediate variables.

Most neural networks are organised into layers. Most layer being a function of the layer that preceded it.

If you have 2 or more hidden layers, you have a Deep feedforward neural network:



# Universal approximation theorem

The Universal approximation theorem (Hornik, 1991) says that

*“a single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units”*

The result applies for sigmoid, tanh and many other hidden layer activation functions.

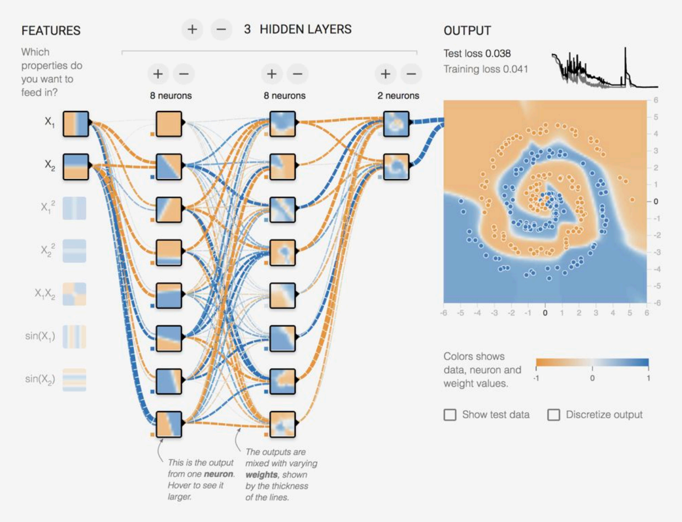
The universal theorem reassures us that neural networks can model pretty much anything.

Although the universal theorem tells us you only need one hidden layer, all recent works show that deeper networks require far fewer parameters and generalise better to the testing set.

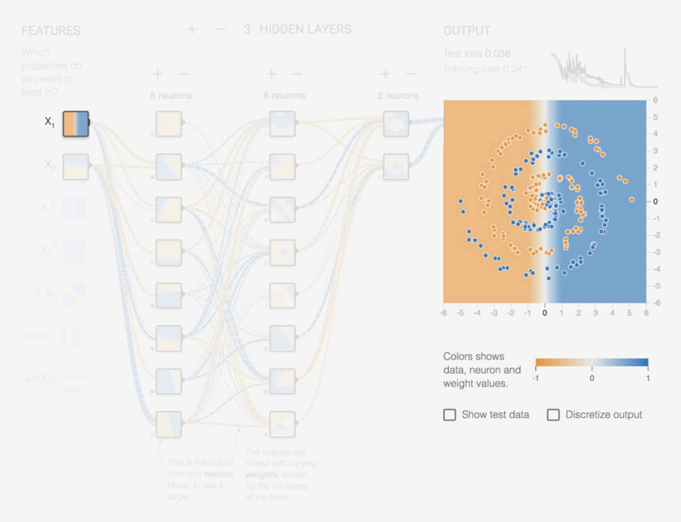
The **architecture** or structure of the network is thus a key design consideration: how many units it should have and how these units should be connected to each other.

This is the main topic of research today. We know that anything can be modelled as a neural net. The challenge is to architect networks that can be efficiently trained and generalise well.

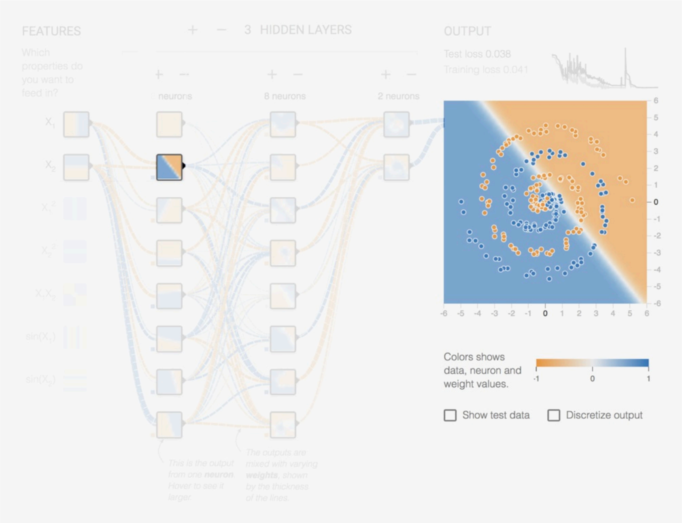
http://playground.tensorflow.org/



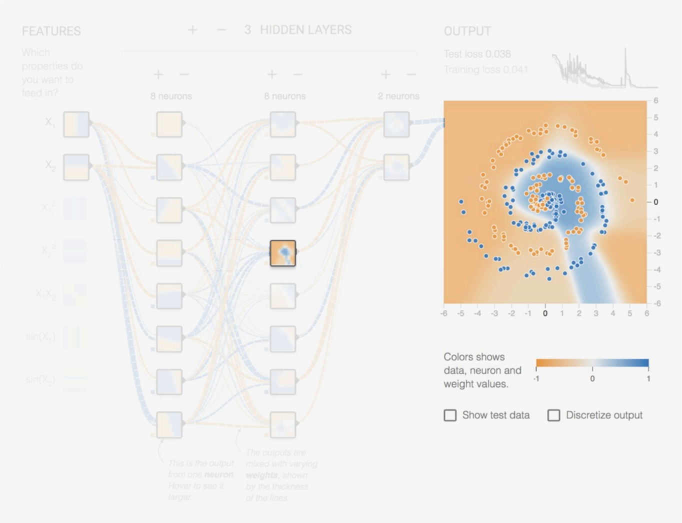

Here is a network with 3 hidden layers of respectively 8, 8 and 2 units.



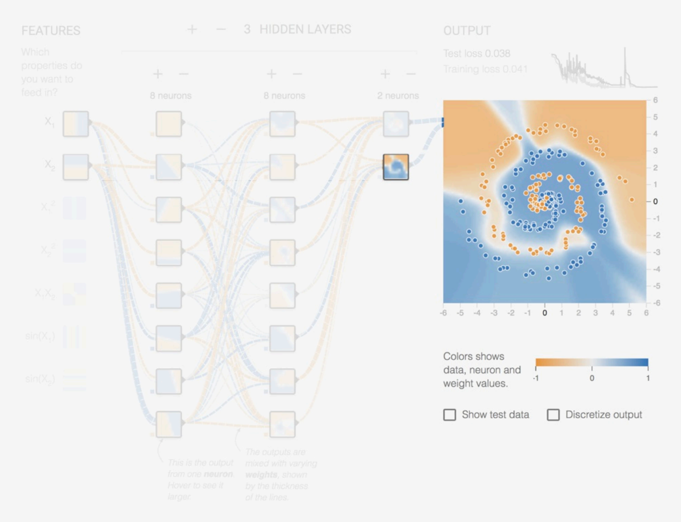

The original features are the x and y coordinates




The units of the first hidden layer produces different rotated versions.




already complex features appear in the second hidden layer




and even more complex features in the third hidden layer

One of the key properties of Neural Nets is their ability to learn arbitrary complex features.

The deeper you go in the network, the more advanced the features are. Thus, even if deeper networks are harder to train, they are more powerful.

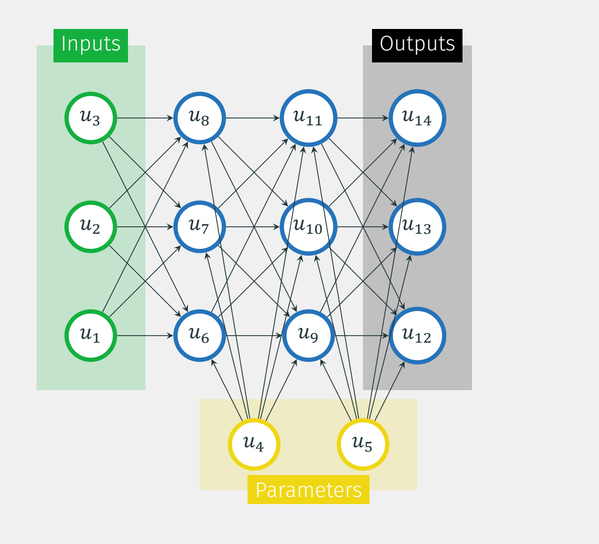
# Training

At its core, a neural net evaluates a function $f$ of the input $\mathbf{x} = (x_1, \cdots, x_p)$ and weights $\mathbf{w} = (w_1, \cdots, w_q)$ and returns output values $\mathbf{y} = (y_1, \cdots, y_r)$:

$$f(x_1, \cdots, x_p, w_1, \cdots, w_q) = (y_1, \cdots, y_r)$$

An example of the graph of operations for **evaluating** the model is presented in the next slide.

To show the universality of the graph representation, all inputs, weights and outputs values have been renamed as $u_i$, where $i$ is the index of the corresponding unit.



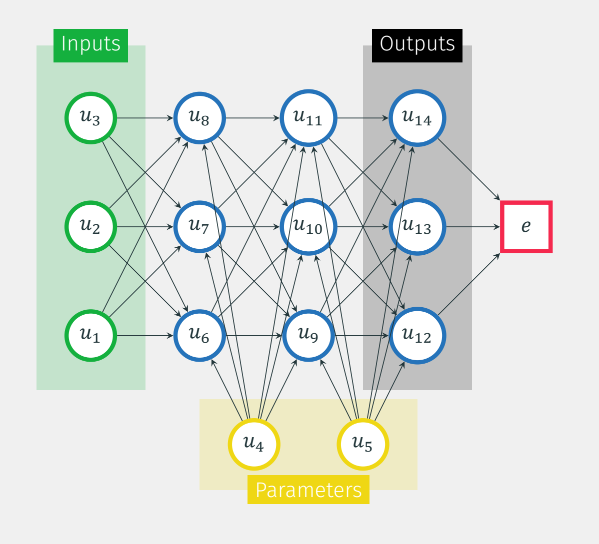

Example of a graph of operations for neural net evaluation.

During **training**, we need to evaluate the output of $f(\mathbf{x}_i, \mathbf{w})$ for a particular observation $\mathbf{x}_i$ and compare it to a observed result $\mathbf{y}_i$. This is done through a loss function $E$.

Typically the loss function aggregates results over all observations:

$$E(\mathbf{w}) = \sum_{i=1}^{n} e(f(\mathbf{x}_i, \mathbf{w}), \mathbf{y}_i)$$

Thus we can build a graph of operations for training. It is the same graph as for evaluation but with all outputs units connected to a loss function unit (see next slide).




Example of a graph for neural net training.

To optimise for the weights $\mathbf{w}$, we resort to a gradient descent approach:

$$\mathbf{w}^{(m+1)} = \mathbf{w}^{(m)} - \eta \frac{\partial e}{\partial \mathbf{w}}(\mathbf{w}^{(m)})$$

where $\eta$ is the learning rate and $e(\mathbf{w}) = \sum_{i=1}^{n} e(f(\mathbf{x}_i, \mathbf{w}), \mathbf{y}_i)$

So we can train any neural net, as long as we know how to compute the gradient $\frac{\partial e}{\partial \mathbf{w}}$.

The Back Propagation algorithm will help us compute this gradient $\frac{\partial e}{\partial \mathbf{w}}$.

# Back-Propagation

Backpropagation (backprop) was pioneered by David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams in 1986.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors." *Cognitive Modeling* 5.3 (1988): 1.

What is the problem with computing the gradient?

Say we want to compute the partial derivative for a particular weight $w_i$. We could naively compute the gradient by numerical differentiation:

$$\frac{\partial e}{\partial w_i} \approx \frac{e(\cdots, w_i + \varepsilon, \cdots) - e(\cdots, w_i, \cdots)}{\varepsilon}$$

with $\varepsilon$ sufficiently small. This is easy to code and quite fast.

Now, modern neural nets can easily have 100M parameters. Computing the gradient this way requires 100M evaluations of the network.

Not a good plan.

Back-Propagation will do it in about 2 evaluations.

Back-Propagation is the very algorithm that made neural nets a viable.

To compute an output $y$ from an input $\mathbf{x}$ in a feedforward net, we process information forward through the graph, evaluate all hidden units $u$ and finally produces $y$. This is called **forward propagation**.

During training, forward propagation continues to produce a scalar error $e(\mathbf{w})$.

The back-propagation algorithm then uses the Chain-Rule to propagate the gradient information from the cost unit back to the weights units.

Recall the chain-rule.

Suppose you have $z = f(y)$ and $y = g(x)$, then

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} = f'(y)g'(x) = f'(g(x))g'(x)$$

In n-dimensions, things are a bit more complicated.

Suppose that $z = f(u_1, \cdots, u_n)$, and that for $k = 1, \ldots, n$, $u_k = g_k(x)$. Then the chain-rule tells us that:

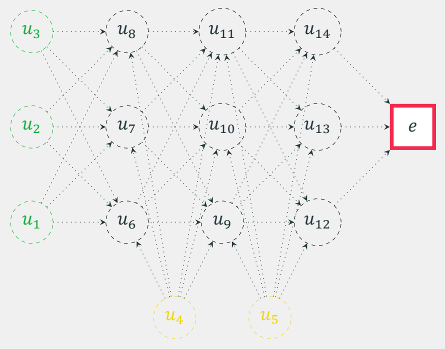
$$\frac{\partial z}{\partial x} = \sum_k \frac{\partial z}{\partial u_k} \frac{\partial u_k}{\partial x} \tag{1}$$

**Example**

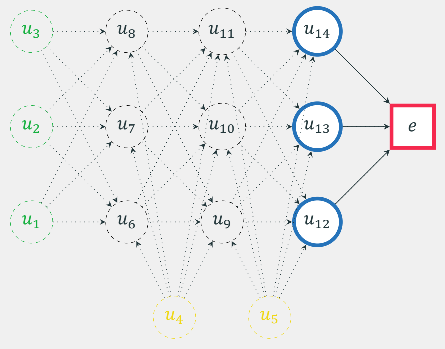
assume that $u(x, y) = x^2 + y^2$, $y(r, t) = r\sin(t)$ and $x(r, t) = r\cos(t)$,

$$\begin{aligned}
\frac{\partial u}{\partial r} &= \frac{\partial u}{\partial x}\frac{\partial x}{\partial r} + \frac{\partial u}{\partial y}\frac{\partial y}{\partial r} \\
&= (2x)(\cos(t)) + (2y)(\sin(t)) \\
&= 2r(\sin^2(t) + \cos^2(t)) \\
&= 2r
\end{aligned}$$

Let’s come back to our neural net example and let’s see how the chain-rule can be used to back-propagate the differentiation.

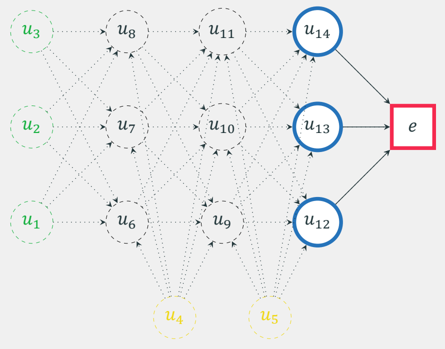



After the forward pass, we have evaluated all units $u$ and finished with the loss $e$.




We can evaluate the partial derivatives $\frac{\partial e}{\partial u_{14}}, \frac{\partial e}{\partial u_{13}}, \frac{\partial e}{\partial u_{12}}$ from the definition of $e$.

Remember that $e$ is simply a function of $u_{12}, u_{13}, u_{14}$.

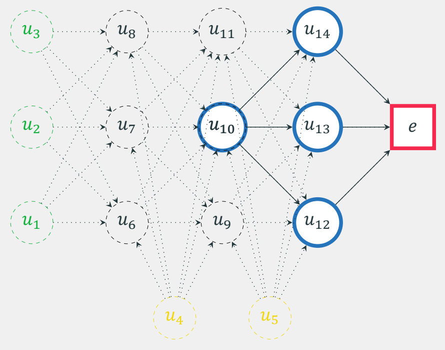



For instance if

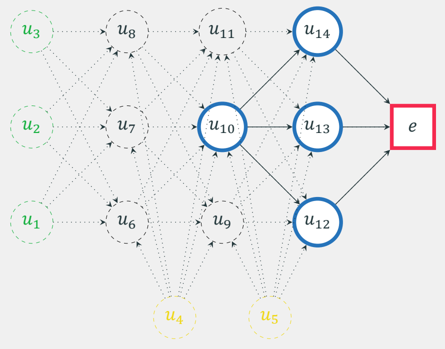
$$e(u_{12}, u_{13}, u_{14}) = (u_{12} - a)^2 + (u_{13} - b)^2 + (u_{14} - c)^2$$

Then

$$\frac{\partial e}{\partial u_{12}} = 2(u_{12} - a) \quad , \frac{\partial e}{\partial u_{13}} = 2(u_{13} - b) \quad , \frac{\partial e}{\partial u_{14}} = 2(u_{14} - c)$$



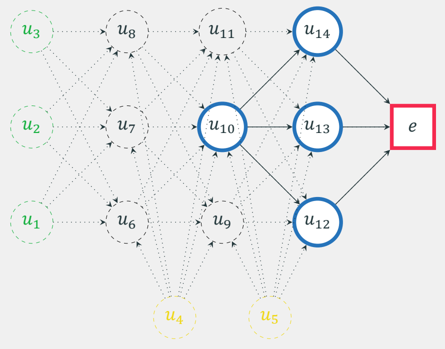

Now that we have computed $\frac{\partial e}{\partial u_{14}}$, $\frac{\partial e}{\partial u_{13}}$ and $\frac{\partial e}{\partial u_{12}}$, how do we compute $\frac{\partial e}{\partial u_{10}}$?




We can use the chain-rule:

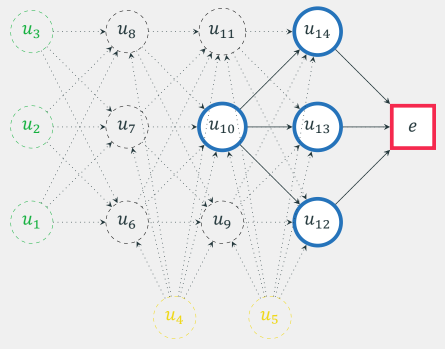
$$\frac{\partial e}{\partial u_i} = \sum_{j \in \text{Outputs}(i)} \frac{\partial u_j}{\partial u_i} \frac{\partial e}{\partial u_j}$$

The Chain Rule links the gradient for $u_i$ to all of the $u_j$ that depend on $u_i$. In our case $u_{14}$, $u_{13}$ and $u_{12}$ depend on $u_{10}$.




So the chain-rule tells us that:

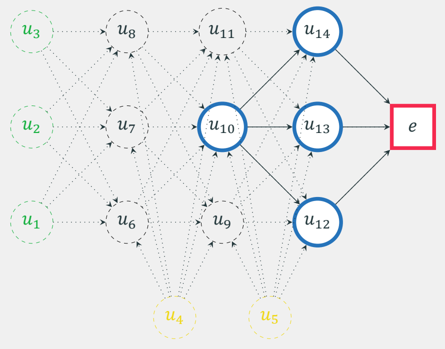
$$\frac{\partial e}{\partial u_{10}} = \frac{\partial u_{14}}{\partial u_{10}} \frac{\partial e}{\partial u_{14}} + \frac{\partial u_{13}}{\partial u_{10}} \frac{\partial e}{\partial u_{13}} + \frac{\partial u_{12}}{\partial u_{10}} \frac{\partial e}{\partial u_{12}}$$




So the chain-rule tells us that:

$$\frac{\partial e}{\partial u_{10}} = \frac{\partial u_{14}}{\partial u_{10}} \frac{\partial e}{\partial u_{14}} + \frac{\partial u_{13}}{\partial u_{10}} \frac{\partial e}{\partial u_{13}} + \frac{\partial u_{12}}{\partial u_{10}} \frac{\partial e}{\partial u_{12}}$$

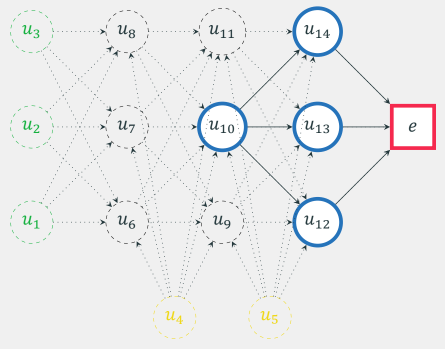
$\frac{\partial e}{\partial u_{12}}$, $\frac{\partial e}{\partial u_{13}}$ and $\frac{\partial e}{\partial u_{14}}$ have already been computed.




So the chain-rule tells us that:

$$\frac{\partial e}{\partial u_{10}} = \frac{\partial u_{14}}{\partial u_{10}} \frac{\partial e}{\partial u_{14}} + \frac{\partial u_{13}}{\partial u_{10}} \frac{\partial e}{\partial u_{13}} + \frac{\partial u_{12}}{\partial u_{10}} \frac{\partial e}{\partial u_{12}}$$

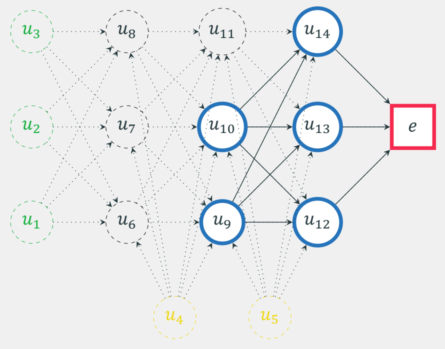
and $\frac{\partial u_{14}}{\partial u_{10}}$, $\frac{\partial u_{13}}{\partial u_{10}}$ and $\frac{\partial u_{12}}{\partial u_{10}}$ can also be derived from the definitions of the functions $u_{12}$, $u_{13}$ and $u_{14}$.




So the chain-rule tells us that:

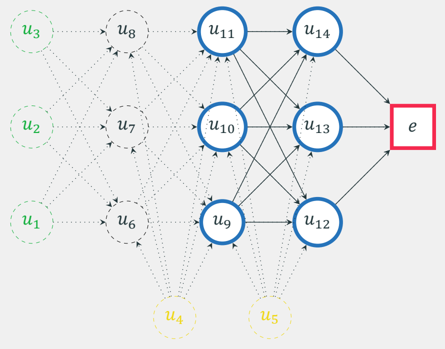
$$\frac{\partial e}{\partial u_{10}} = \frac{\partial u_{14}}{\partial u_{10}} \frac{\partial e}{\partial u_{14}} + \frac{\partial u_{13}}{\partial u_{10}} \frac{\partial e}{\partial u_{13}} + \frac{\partial u_{12}}{\partial u_{10}} \frac{\partial e}{\partial u_{12}}$$

For instance if $u_{14}(u_5, u_{10}, u_{11}, u_9) = u_5 + 0.2u_{10} + 0.7u_{11} + 0.3u_9$, then $\frac{\partial u_{14}}{\partial u_{10}} = 0.2$



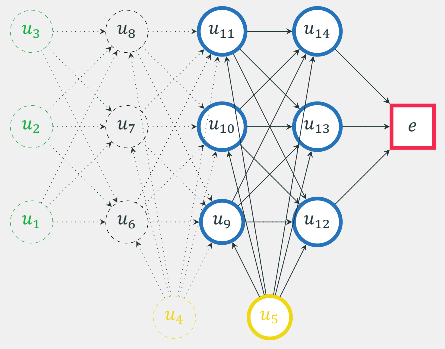

We can now propagate back the computations and derive the gradient for each node at a time.

$$\frac{\partial e}{\partial u_9} = \frac{\partial u_{14}}{\partial u_9}\frac{\partial e}{\partial u_{14}} + \frac{\partial u_{13}}{\partial u_9}\frac{\partial e}{\partial u_{13}} + \frac{\partial u_{12}}{\partial u_9}\frac{\partial e}{\partial u_{12}}$$



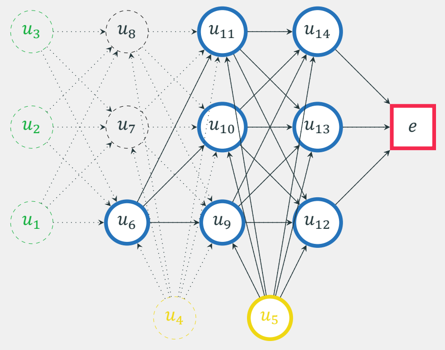

We can now propagate back the computations and derive the gradient for each node at a time.

$$\frac{\partial e}{\partial u_{11}} = \frac{\partial u_{14}}{\partial u_{11}} \frac{\partial e}{\partial u_{14}} + \frac{\partial u_{13}}{\partial u_{11}} \frac{\partial e}{\partial u_{13}} + \frac{\partial u_{12}}{\partial u_{11}} \frac{\partial e}{\partial u_{12}}$$



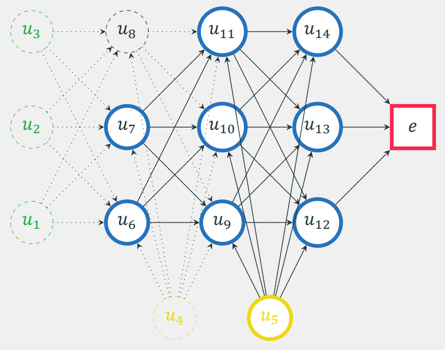

We can now propagate back the computations and derive the gradient for each node at a time.

$$\frac{\partial e}{\partial u_5} = \frac{\partial u_{14}}{\partial u_5}\frac{\partial e}{\partial u_{14}} + \frac{\partial u_{13}}{\partial u_5}\frac{\partial e}{\partial u_{13}} + \frac{\partial u_{12}}{\partial u_5}\frac{\partial e}{\partial u_{12}} + \frac{\partial u_{11}}{\partial u_5}\frac{\partial e}{\partial u_{11}} + \frac{\partial u_{10}}{\partial u_5}\frac{\partial e}{\partial u_{10}} + \frac{\partial u_9}{\partial u_5}\frac{\partial e}{\partial u_9}$$



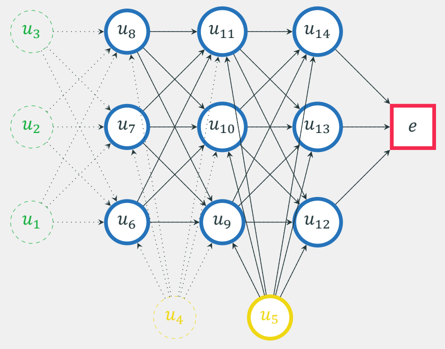

We can now propagate back the computations and derive the gradient for each node at a time.

$$\frac{\partial e}{\partial u_6} = \frac{\partial u_{14}}{\partial u_6}\frac{\partial e}{\partial u_{14}} + \frac{\partial u_{13}}{\partial u_6}\frac{\partial e}{\partial u_{13}} + \frac{\partial u_{12}}{\partial u_6}\frac{\partial e}{\partial u_{12}}$$



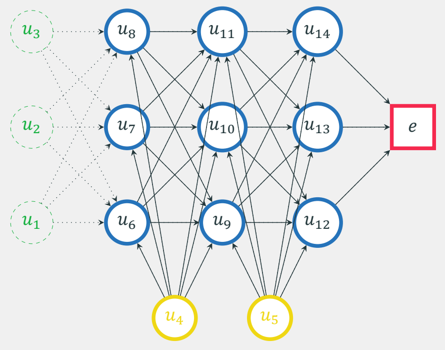

We can now propagate back the computations and derive the gradient for each node at a time.

$$\frac{\partial e}{\partial u_7} = \frac{\partial u_{14}}{\partial u_7}\frac{\partial e}{\partial u_{14}} + \frac{\partial u_{13}}{\partial u_7}\frac{\partial e}{\partial u_{13}} + \frac{\partial u_{12}}{\partial u_7}\frac{\partial e}{\partial u_{12}}$$




We can now propagate back the computations and derive the gradient for each node at a time.

$$\frac{\partial e}{\partial u_8} = \frac{\partial u_{14}}{\partial u_8}\frac{\partial e}{\partial u_{14}} + \frac{\partial u_{13}}{\partial u_8}\frac{\partial e}{\partial u_{13}} + \frac{\partial u_{12}}{\partial u_8}\frac{\partial e}{\partial u_{12}}$$




We can now propagate back the computations and derive the gradient for each node at a time.

$$\frac{\partial e}{\partial u_4} = \frac{\partial u_8}{\partial u_4}\frac{\partial e}{\partial u_8} + \frac{\partial u_7}{\partial u_4}\frac{\partial e}{\partial u_7} + \frac{\partial u_6}{\partial u_4}\frac{\partial e}{\partial u_6} + \frac{\partial u_{11}}{\partial u_4}\frac{\partial e}{\partial u_{11}} + \frac{\partial u_{10}}{\partial u_4}\frac{\partial e}{\partial u_{10}} + \frac{\partial u_9}{\partial u_4}\frac{\partial e}{\partial u_9}$$

So Back Propagation proceeds by induction.

Assume that we know how to compute $\frac{\partial E}{\partial u_j}$ for a subset of units $\mathcal{K}$ of the network. Pick a node $i$ outside of $\mathcal{K}$ but with all of its outputs in $\mathcal{K}$.

We can compute $\frac{\partial e}{\partial u_i}$ using the chain-rule:

$$\frac{\partial e}{\partial u_i} = \sum_{j \in \mathbf{Outputs}(i)} \frac{\partial e}{\partial u_j} \frac{\partial u_j}{\partial u_i}$$

We have already computed $\frac{\partial e}{\partial u_j}$ for $j \in \mathcal{K}$ and we can compute directly $\frac{\partial u_j}{\partial u_i}$ by differentiating the function $u_j$ with respect to its input $u_i$.

We can stop the back propagation once $\frac{\partial e}{\partial u_i}$ has been computed for all the parameter units in the graph.

Back-Propagation is the fastest method we have to compute the gradient in a graph.

The worst case complexity of backprop is $\mathcal{O}(\text{number\_of\_units}^2)$ but in practice for most network architectures it is $\mathcal{O}(\text{number\_of\_units})$.

Back-Propagation is what makes training deep neural nets possible.