Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

05 - Deep FeedForward Neural Networks

François Pitié

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

[4CT6] Deep Learning and its Applications — 2019/2020

FeedForward Neural Network

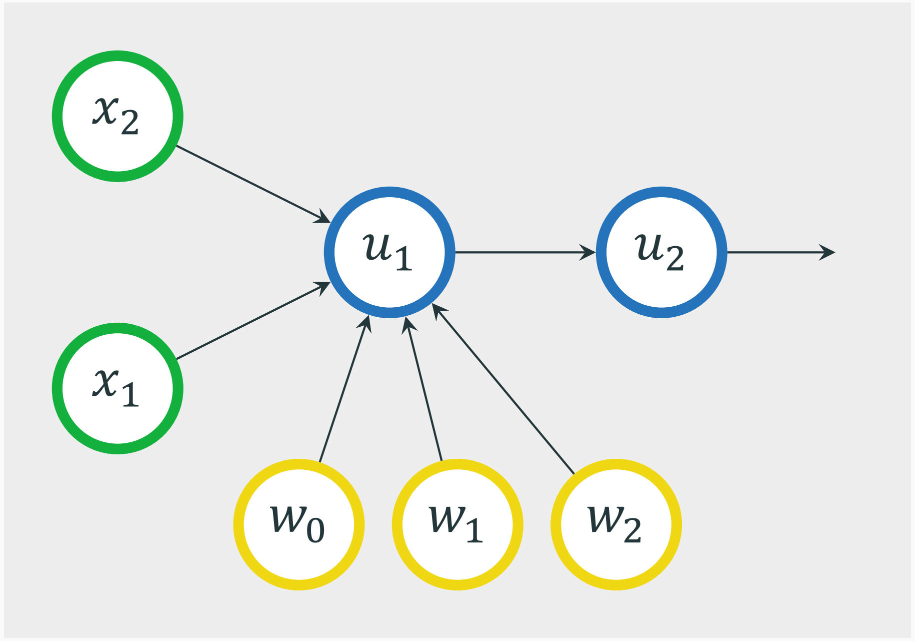
Remember Logistic Regression? The output of the model was

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

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.

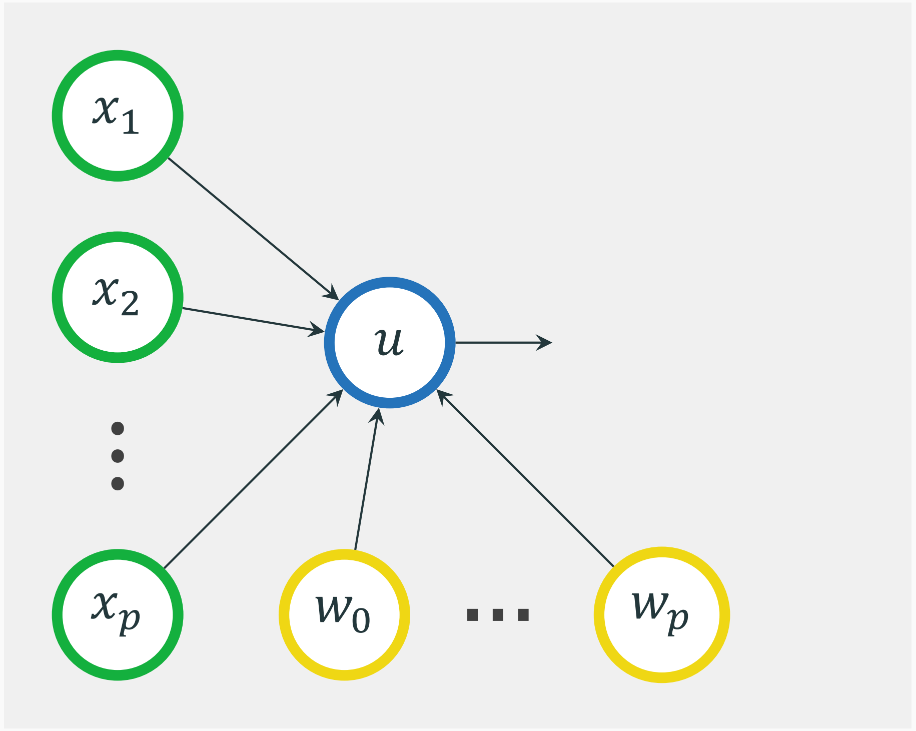
Each node in the graph is called a **unit**.

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 <u>Neural</u> Network?

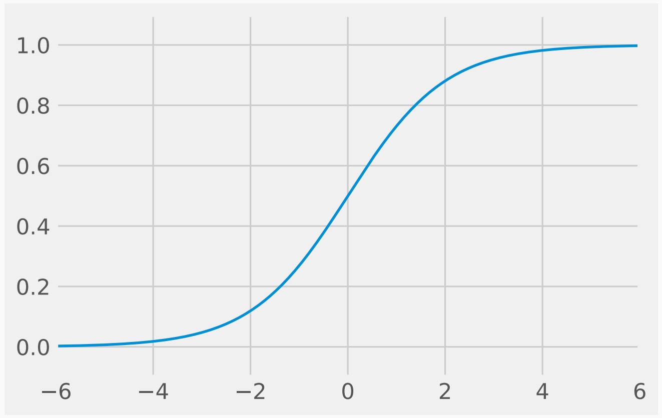
Because it mainly relies on neuron-like units. A **neuron unit** first takes a linear combination of its inputs and then applies a non-linear function f, called **activation function**:



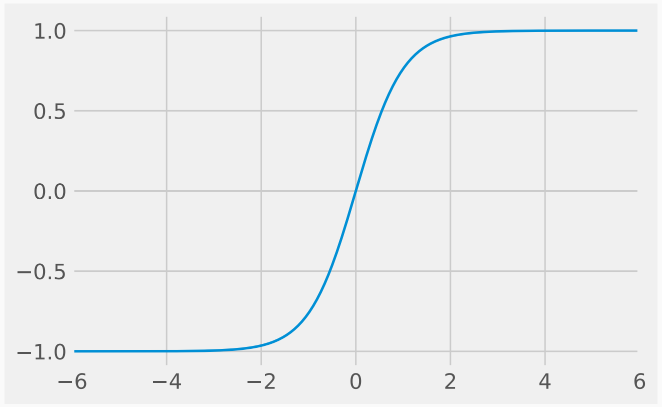
$$u = f(w_0 + \sum_{i=1}^{p} w_i x_i)$$

Activation Functions

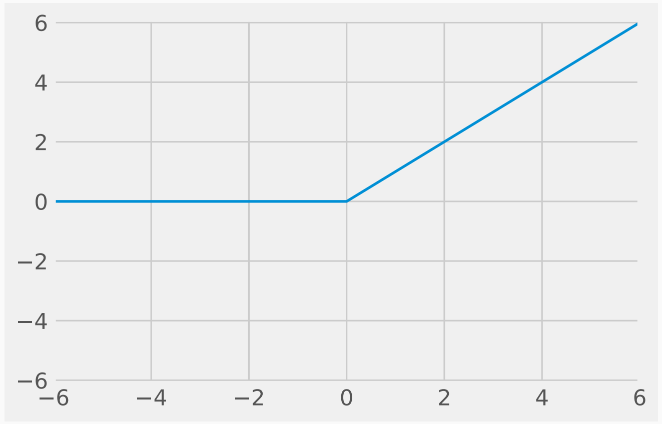
Many activation functions exist. Here are some of the most popular:



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



tanh: $z = \tanh(z)$



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

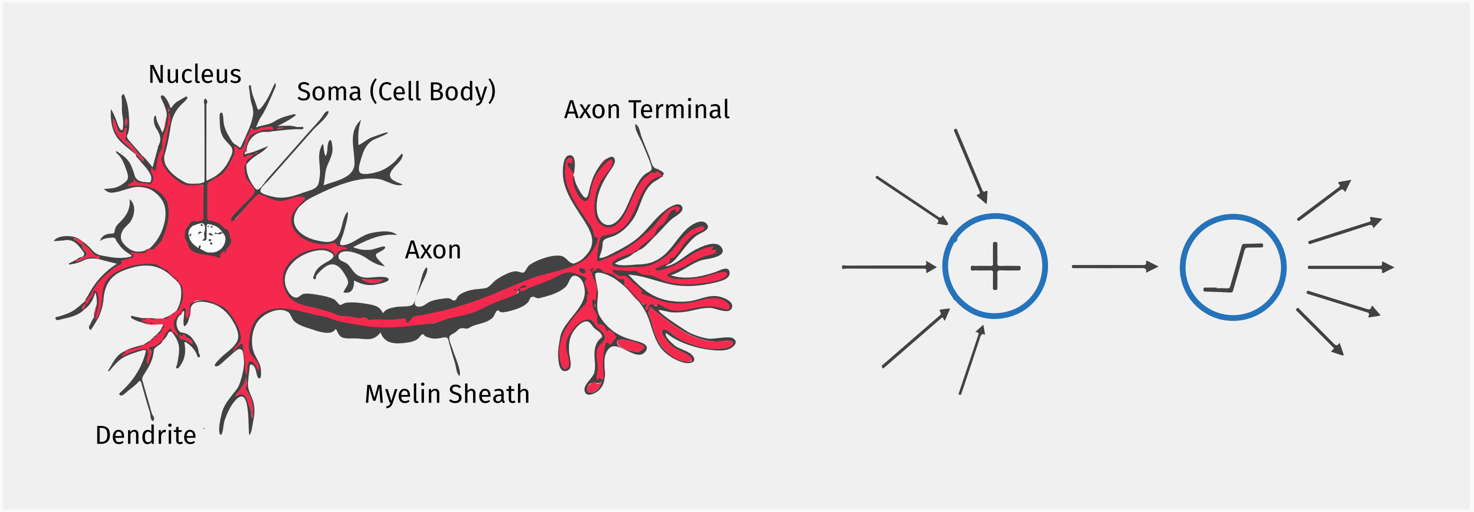
Activation Functions

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 reason to use more exotic functions, we will stick to these known types.

Biological 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.



Neurons

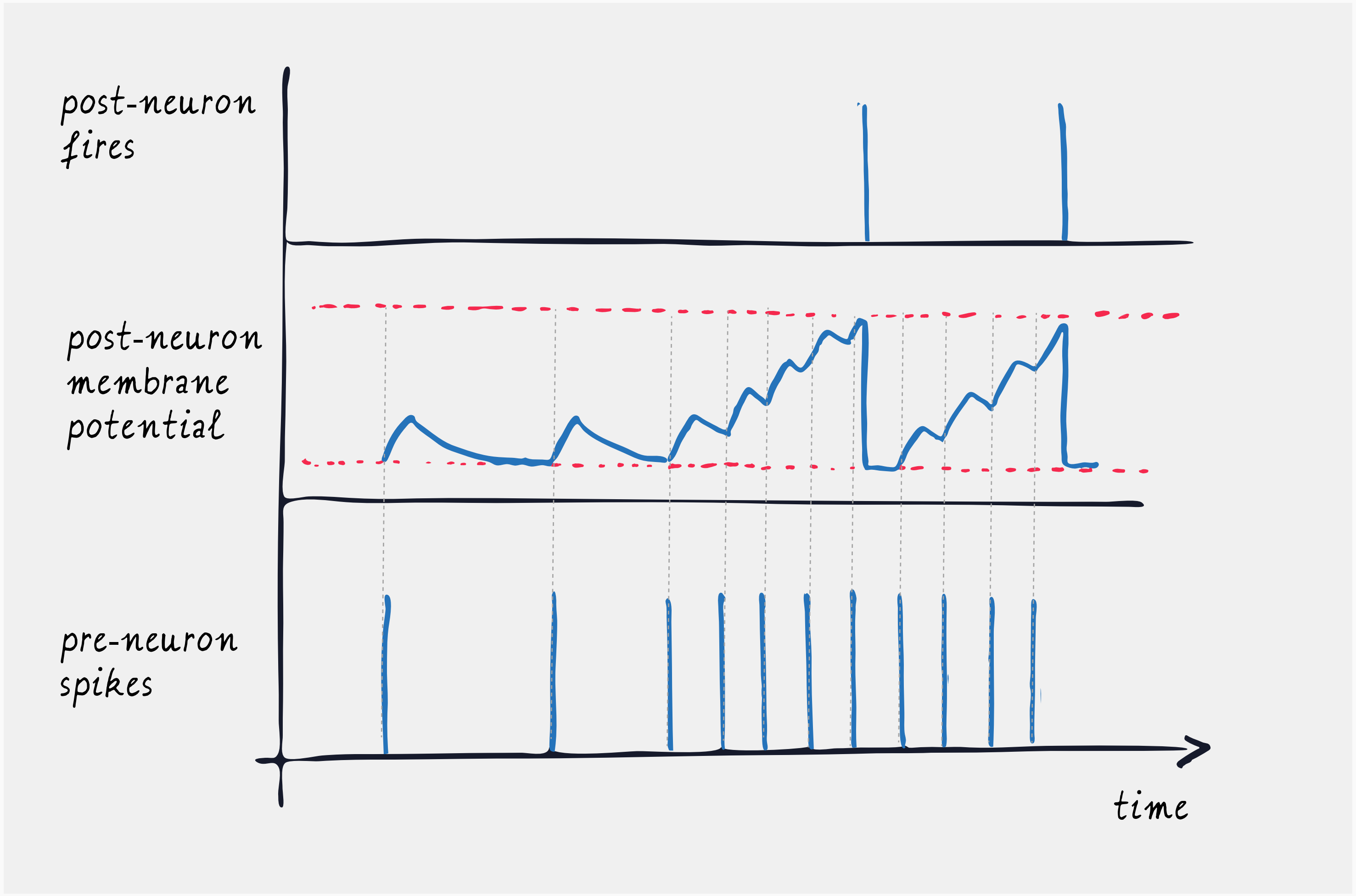
There have been many attempts at mathematically modelling the dynamics of the neuron's electrical input-output voltage. For instance the Integrate-and-Fire model, first proposed in 1907 by Louis Lapicque, relates the input current to the variations of the membrane voltage:

$$I(t) = C_m \frac{dV_m(t)}{dt}$$

The membrane voltage increases with time when input current is provides until it reaches a threshold. At that point a voltage spike occurs and the voltage is reset to a lower potential.

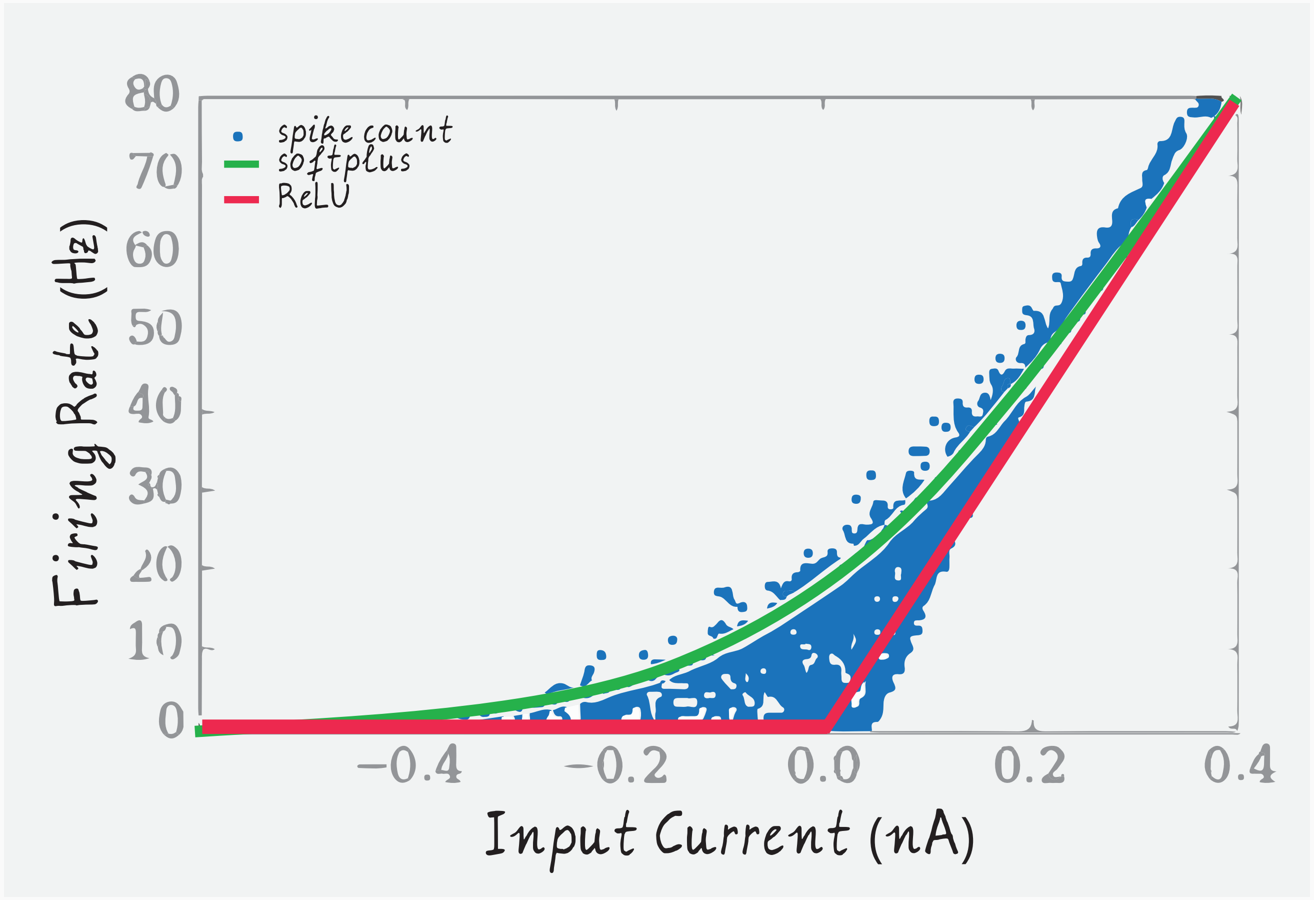
DErived models include the Hodgkin–Huxley model and the Leaky integrate-and-fire model.

Neurons



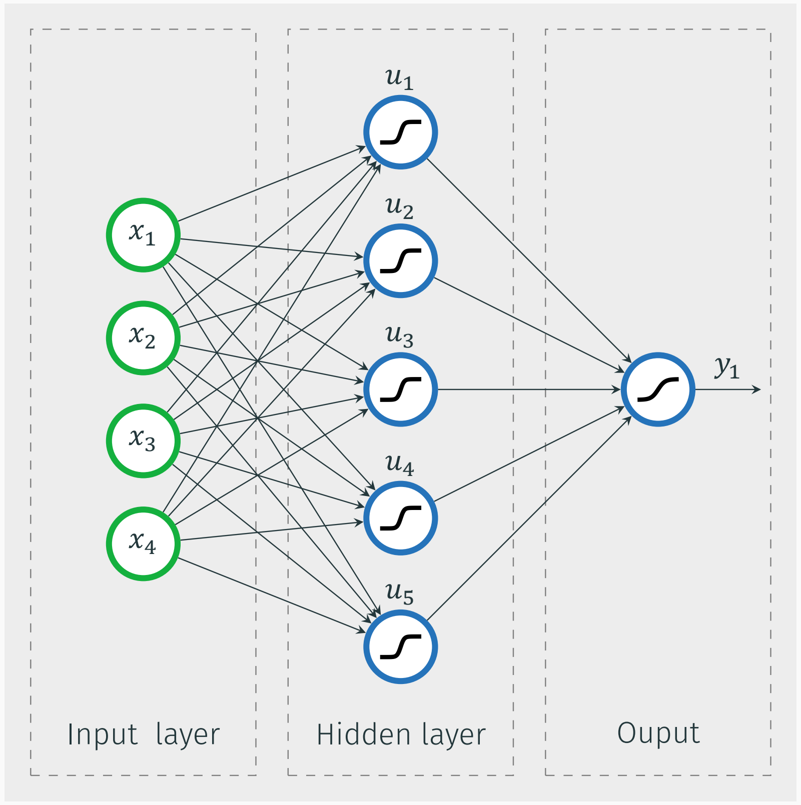
Example showing the dynamics of a Leaky Integrate and Fire neuron model.

Neurons



Example of a neural net simulation of Leaky Integrate and Fire neurons. The frequency of output spikes (firing rate) is increasing with input intensity. The distribution of the output spikes shows that ReLU type activation functions are indeed biologically plausible.

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

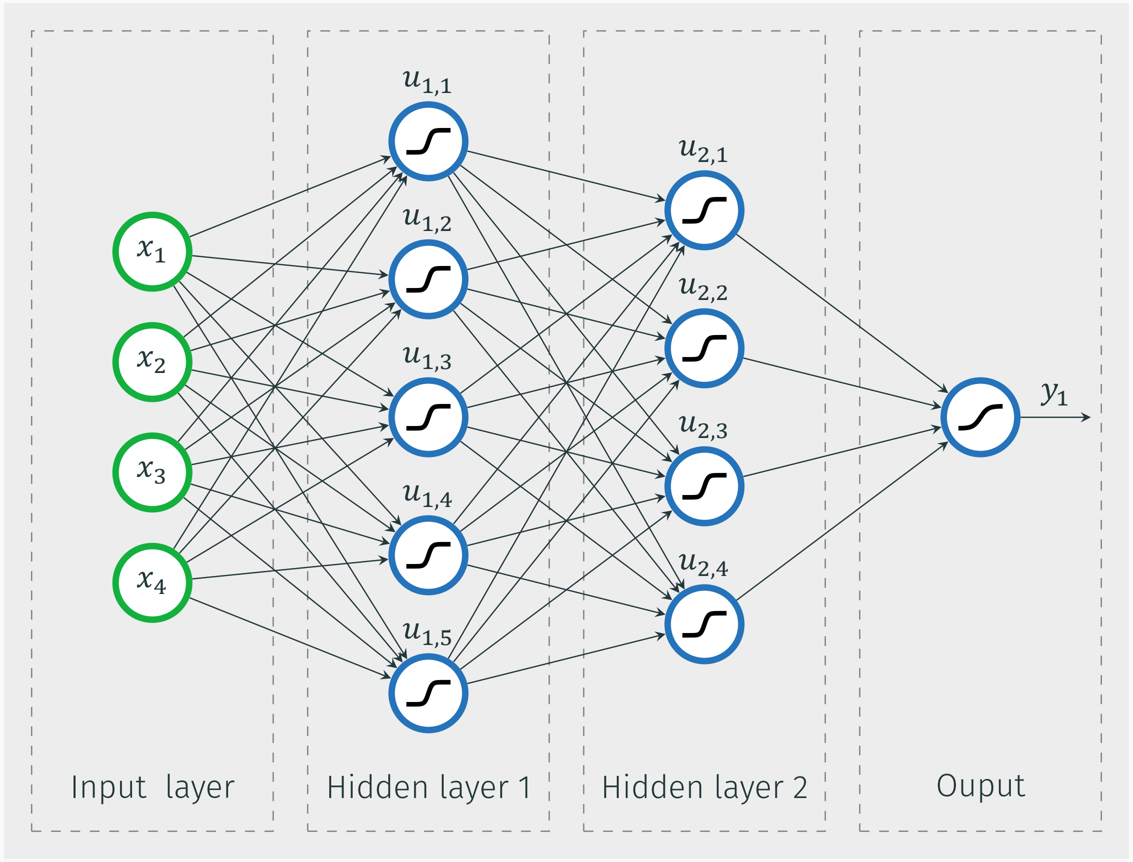


Each blue unit is a neuron with its activation function.

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.

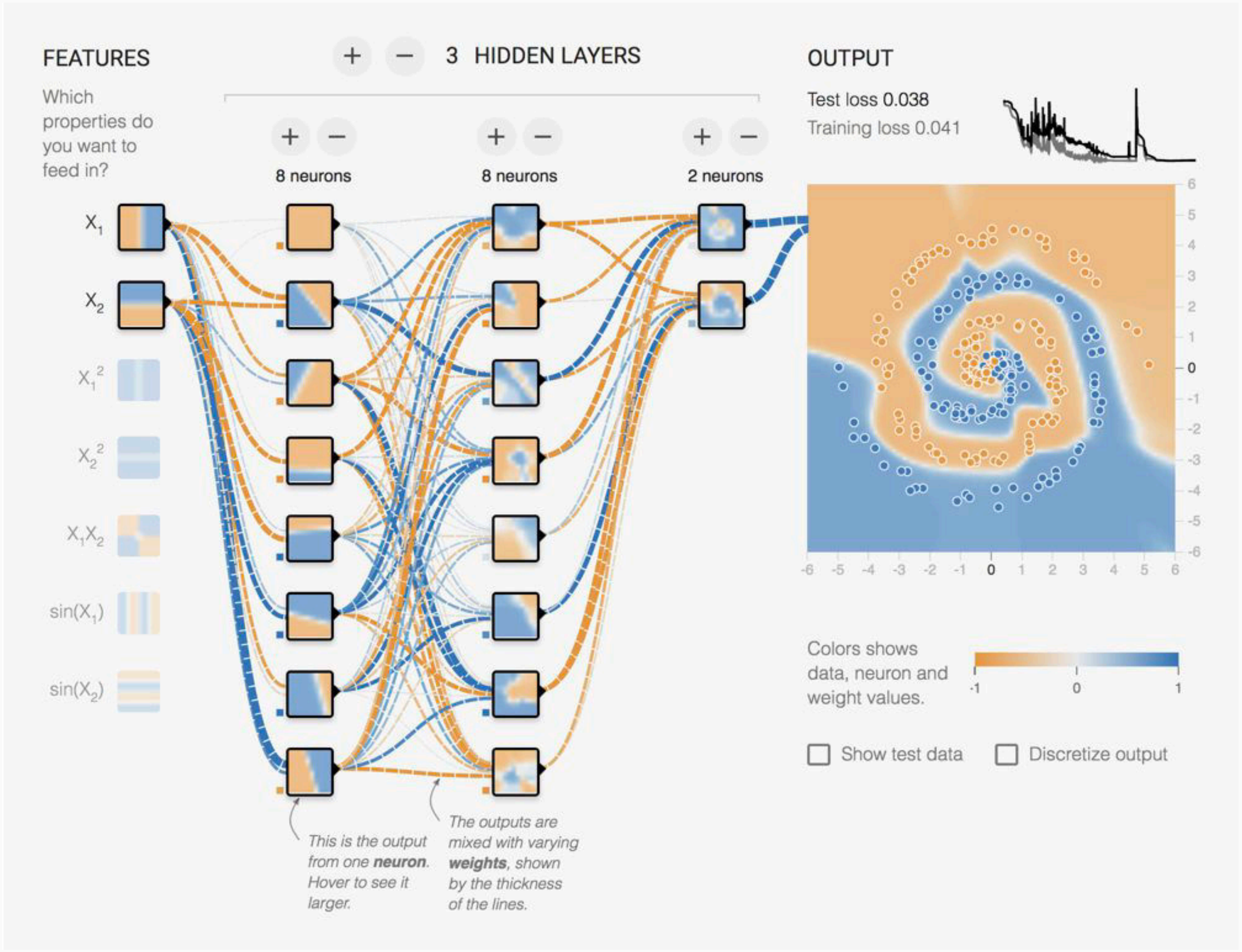
The universal theorem reassures us that neural networks can model pretty much anything.

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.

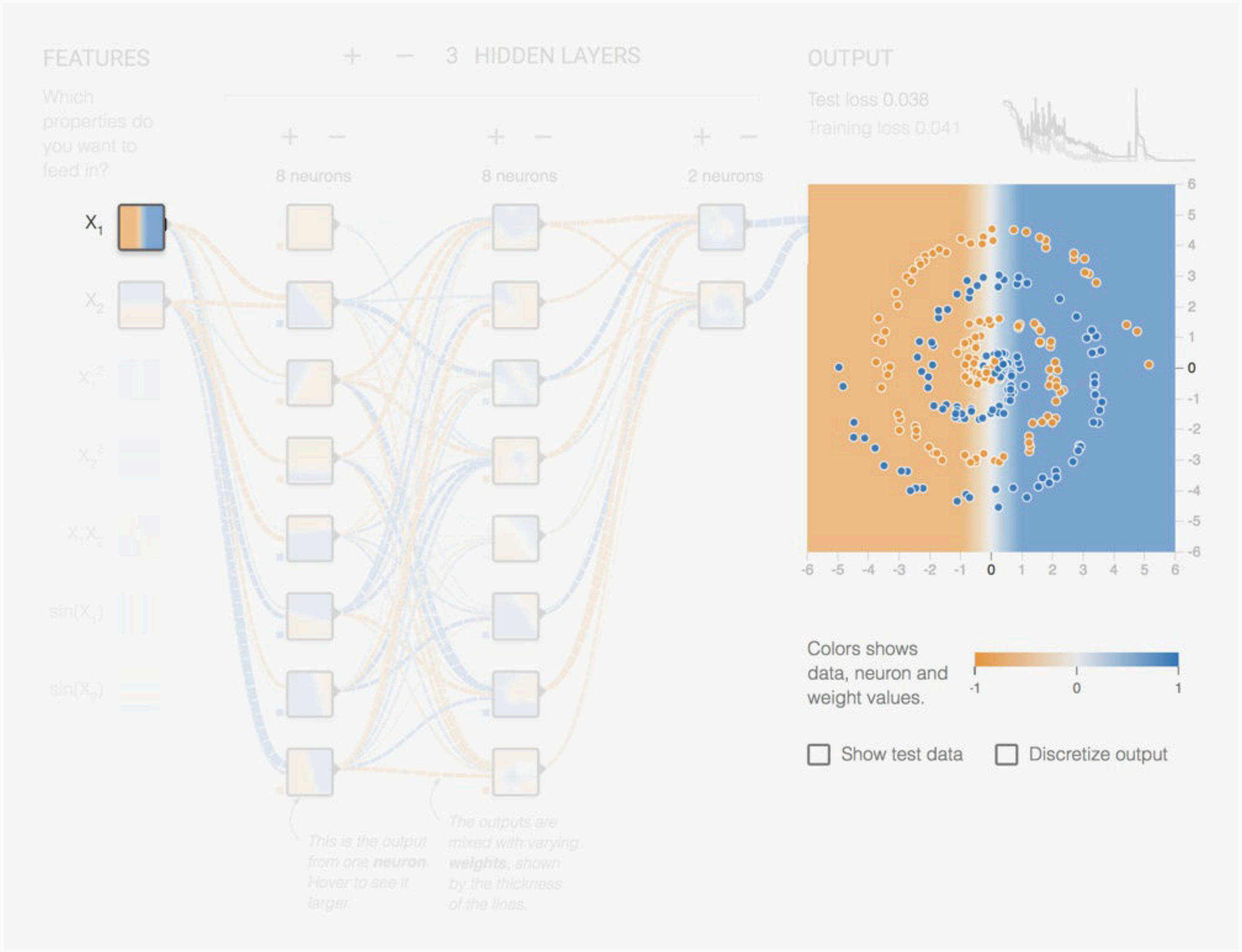
The **architecture** or structure of the network is thus a key design consideration: how many units it should have and how these units should be connected to each other.

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/


FEATURES
Which properties do you want to feed in?
X1
X2
X1^2
X2^2
X1X2
sin(X1)
sin(X2)
+ − 3 HIDDEN LAYERS
8 neurons
8 neurons
2 neurons
OUTPUT
Test loss 0.038
Training loss 0.041
Colors shows data, neuron and weight values.
-1
0
1
Show test data
Discretize output
This is the output from one neuron. Hover to see it larger.
The outputs are mixed with varying weights, shown by the thickness of the lines.

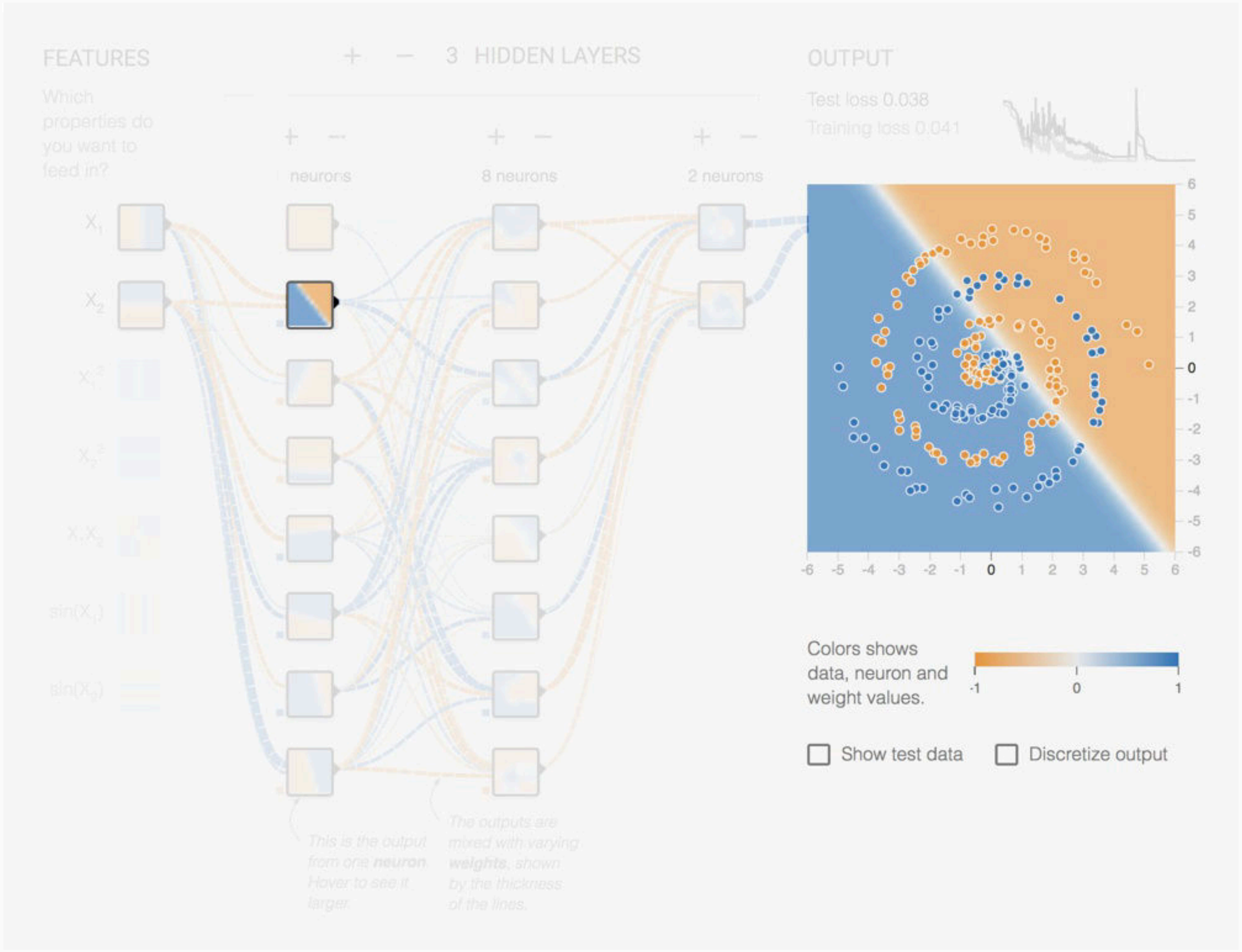
Here is a network with 3 hidden layers of respectively 8, 8 and 2 units.



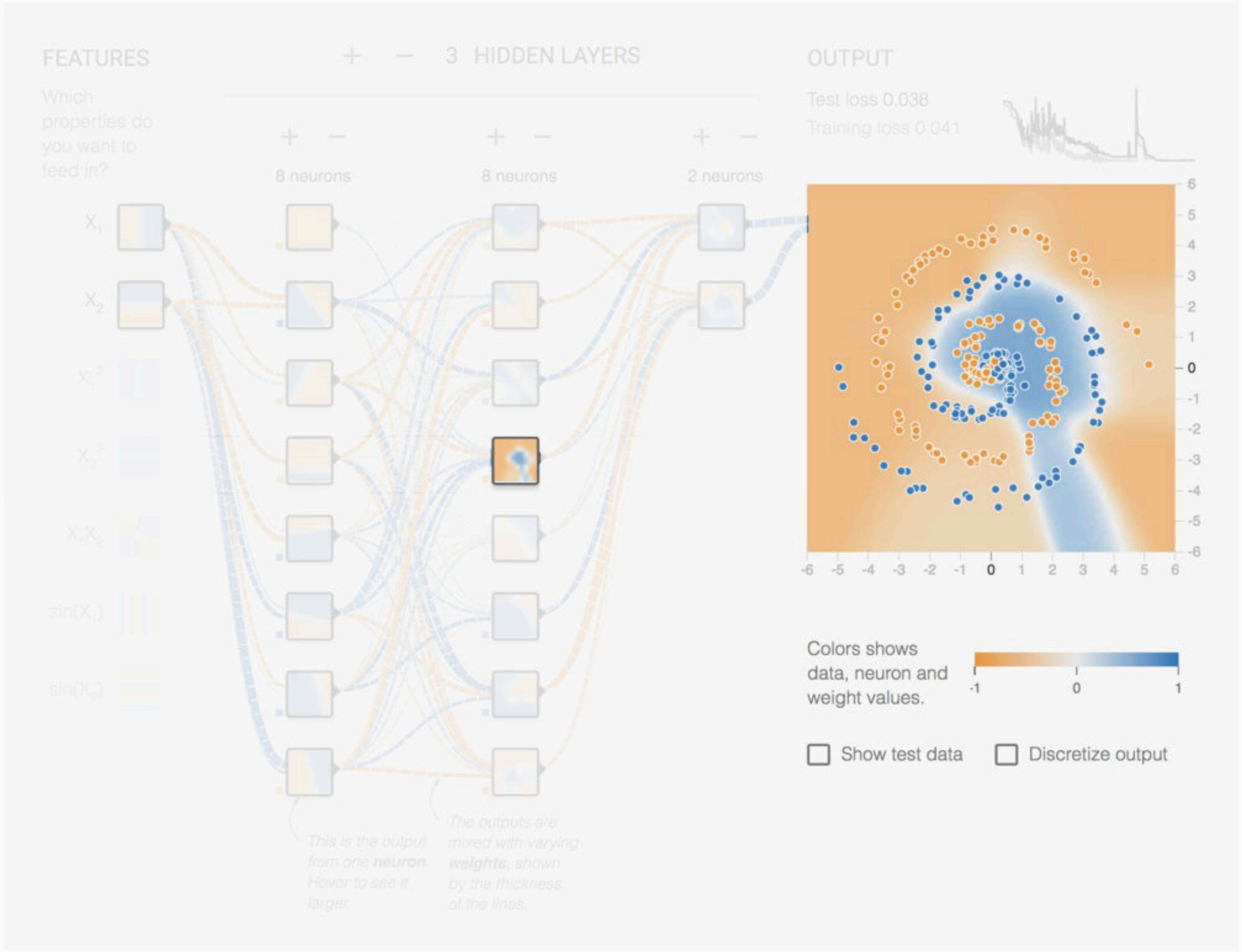
This is the output from one **neuron**. Hover to see it larger.

The outputs are mixed with varying **weights**, shown by the thickness of the lines.

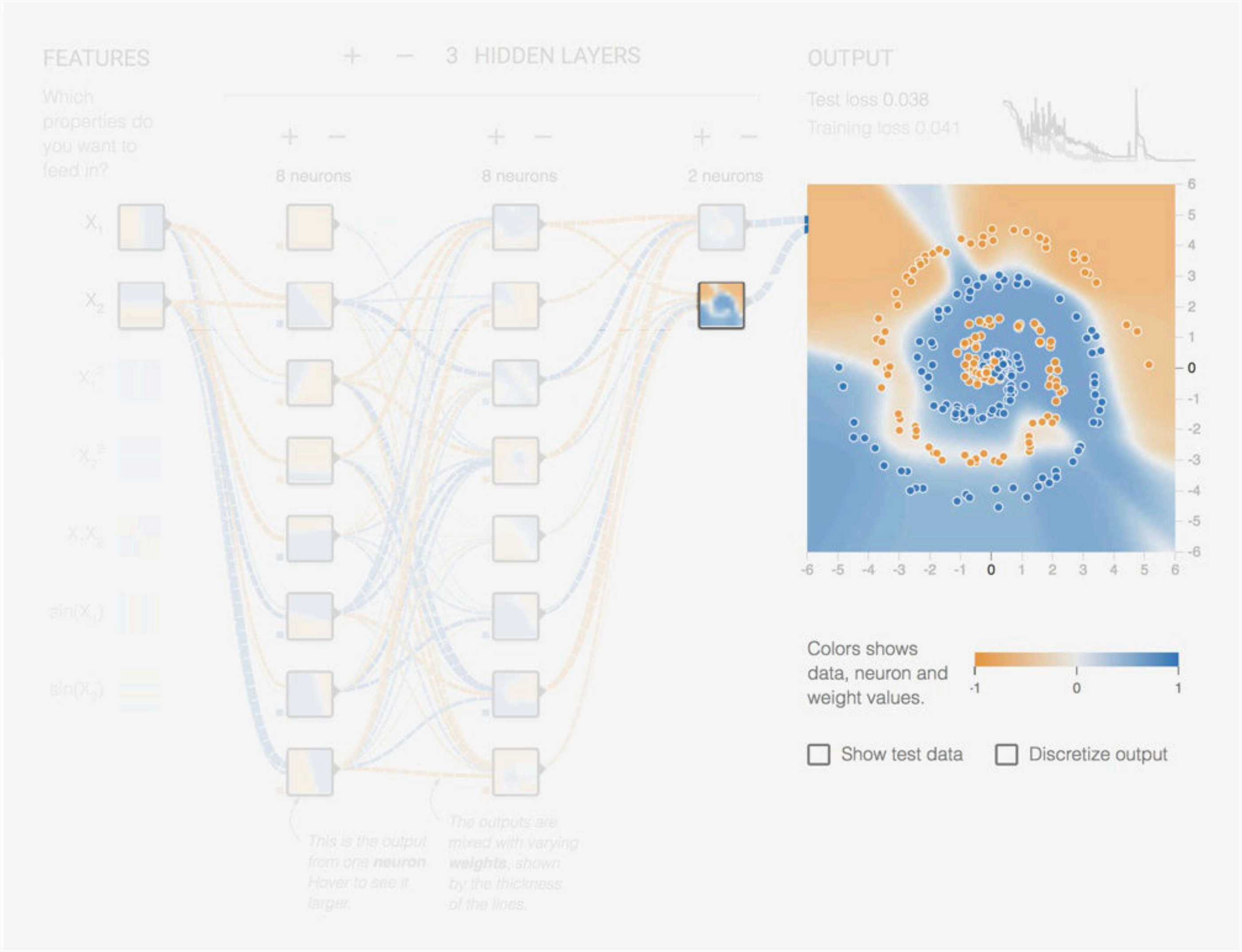
The original features are the x and y coordinates


FEATURES
+ − 3 HIDDEN LAYERS
OUTPUT
Which properties do you want to feed in?
+ −
neurons
8 neurons
2 neurons
Test loss 0.038
Training loss 0.041
X_1
X_2
X_1^2
X_2^2
X_1X_2
$\sin(X_1)$
$\sin(X_2)$
This is the output from one neuron. Hover to see it larger.
The outputs are mixed with varying weights, shown by the thickness of the lines.
Colors shows data, neuron and weight values.
-1
0
1
Show test data
Discretize output

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


FEATURES
Which properties do you want to feed in?
+ − 3 HIDDEN LAYERS
8 neurons
8 neurons
2 neurons
OUTPUT
Test loss 0.038
Training loss 0.041
This is the output from one neuron. Hover to see it larger.
The outputs are mixed with varying weights, shown by the thickness of the lines.
Colors shows data, neuron and weight values.
-1
0
1
Show test data
Discretize output

already complex features appear in the second hidden layer


FEATURES
Which properties do you want to feed in?
X_1
X_2
X_1^2
X_2^2
X_1X_2
$\sin(X_1)$
$\sin(X_2)$
+ − 3 HIDDEN LAYERS
8 neurons
8 neurons
2 neurons
This is the output from one neuron. Hover to see it larger.
The outputs are mixed with varying weights, shown by the thickness of the lines.
OUTPUT
Test loss 0.038
Training loss 0.041
Colors shows data, neuron and weight values.
-1
0
1
Show test data
Discretize output

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.

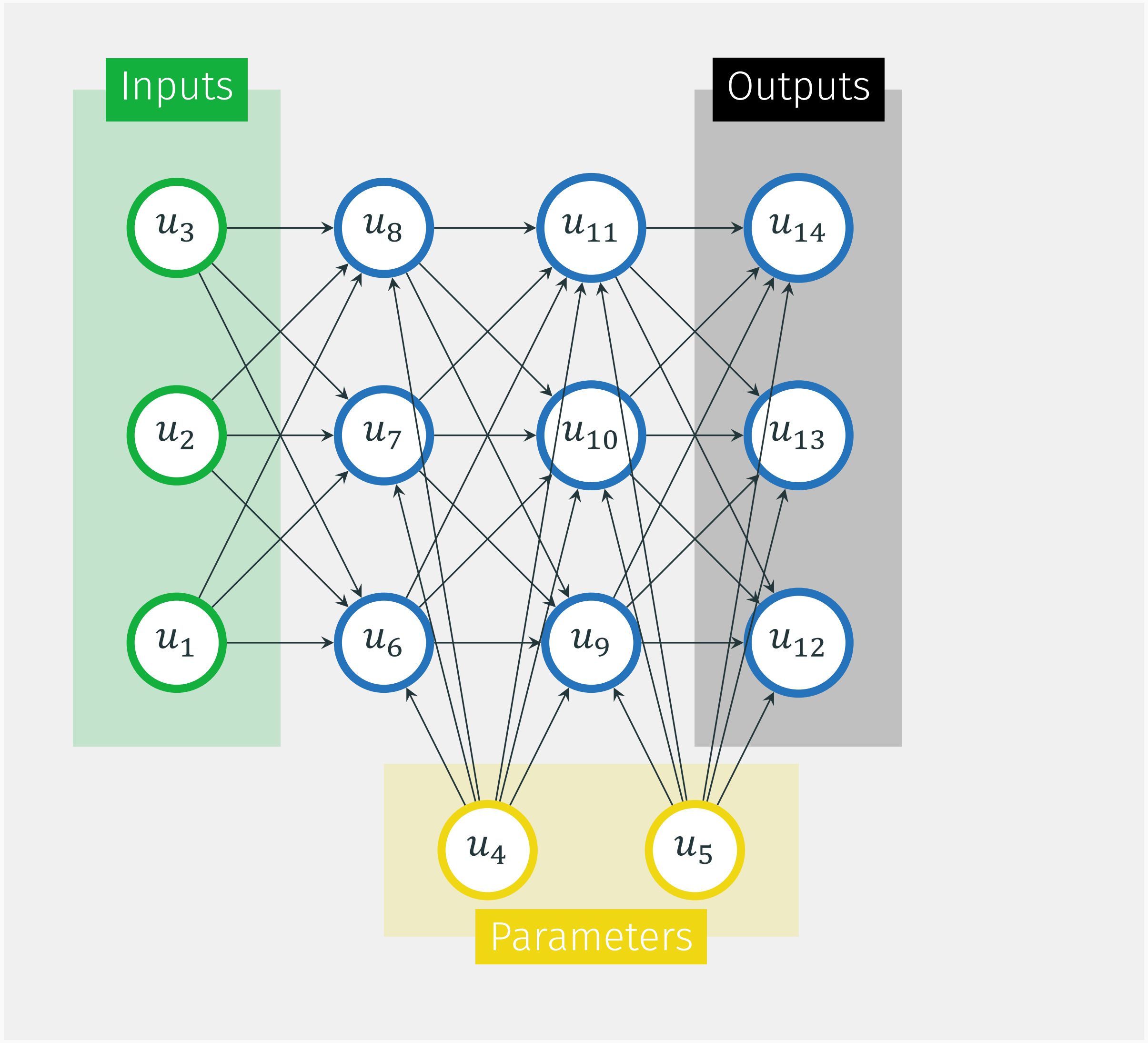
Training

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.



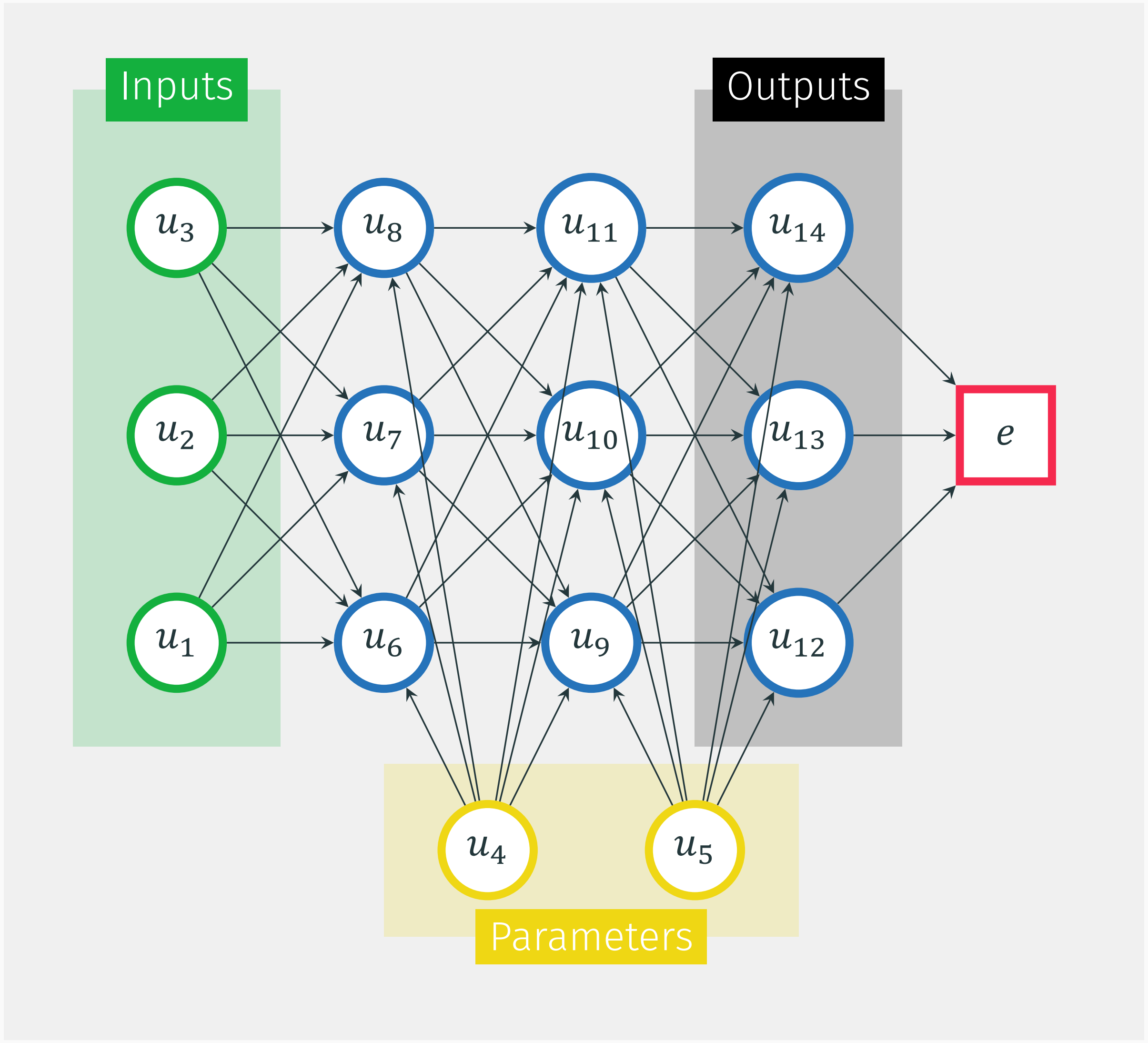
Example of a graph of operations for neural net evaluation.

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 η 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 ε 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 $k = 1, \dots, n$, $u_k = g_k(x)$. Then the chain-rule tells us that:

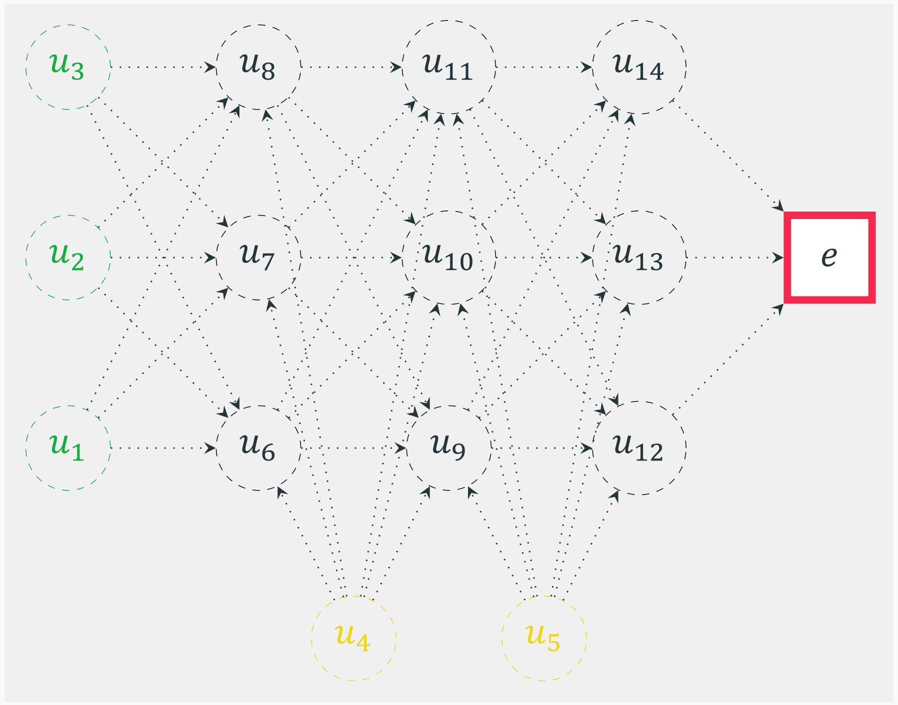
$$\frac{\partial z}{\partial x} = \sum_k \frac{\partial z}{\partial u_k} \frac{\partial u_k}{\partial x} \tag{1}$$

Example

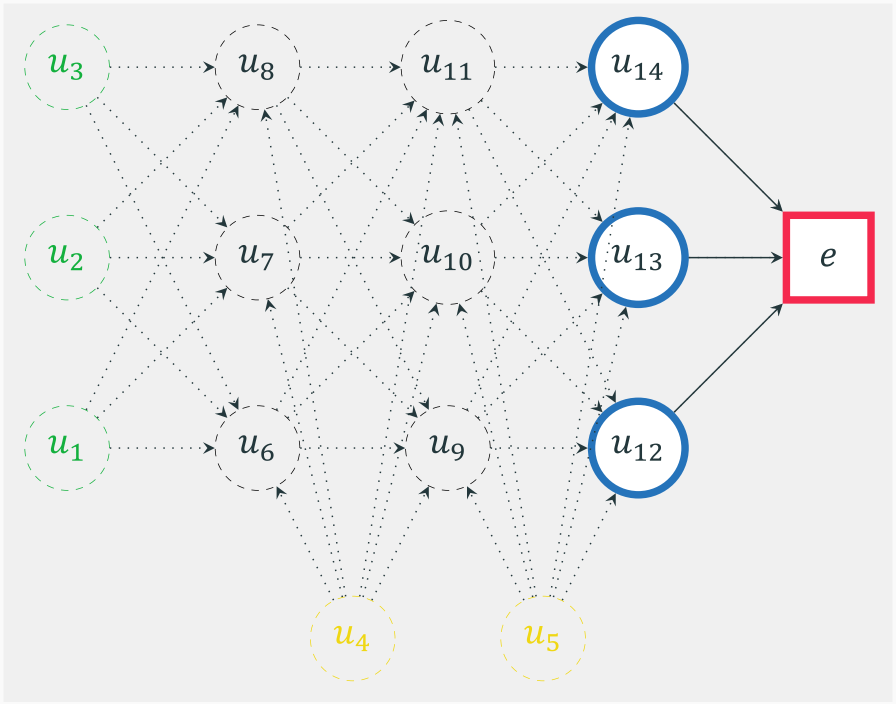
assume that $u(x, y) = x^2 + y^2$, $y(r, t) = r\sin(t)$ and $x(r, t) = r\cos(t)$,

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

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.

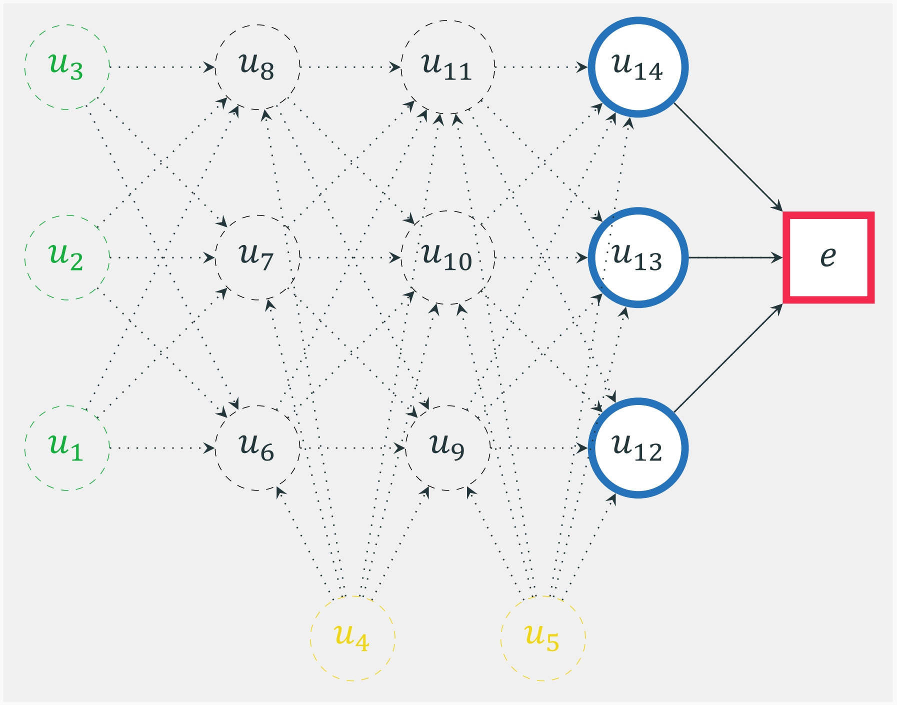


After the forward pass, we have evaluated all units u and finished with the loss e.



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}.

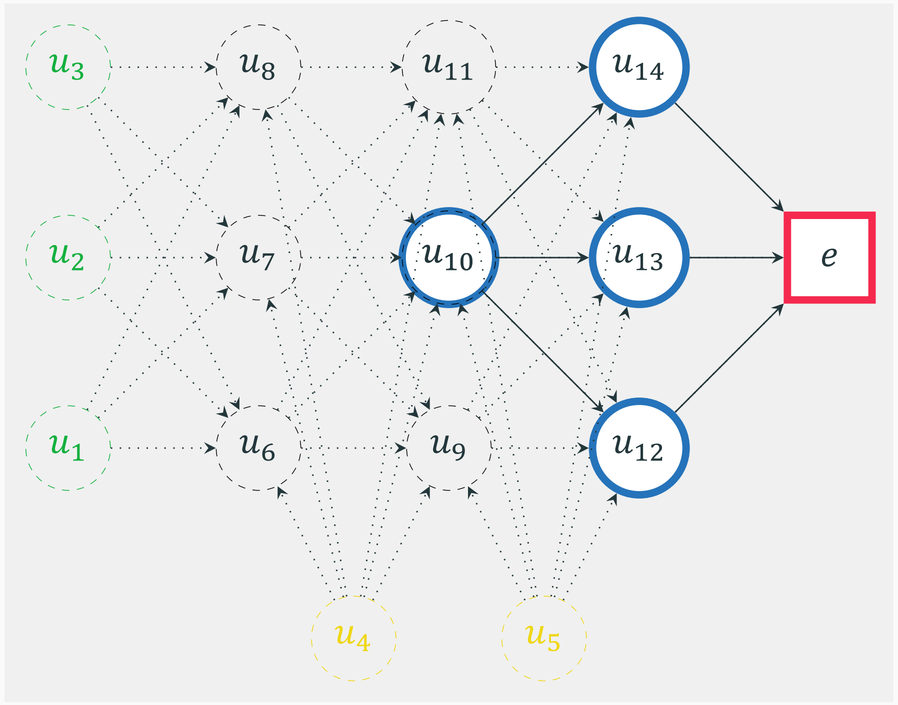


For instance if

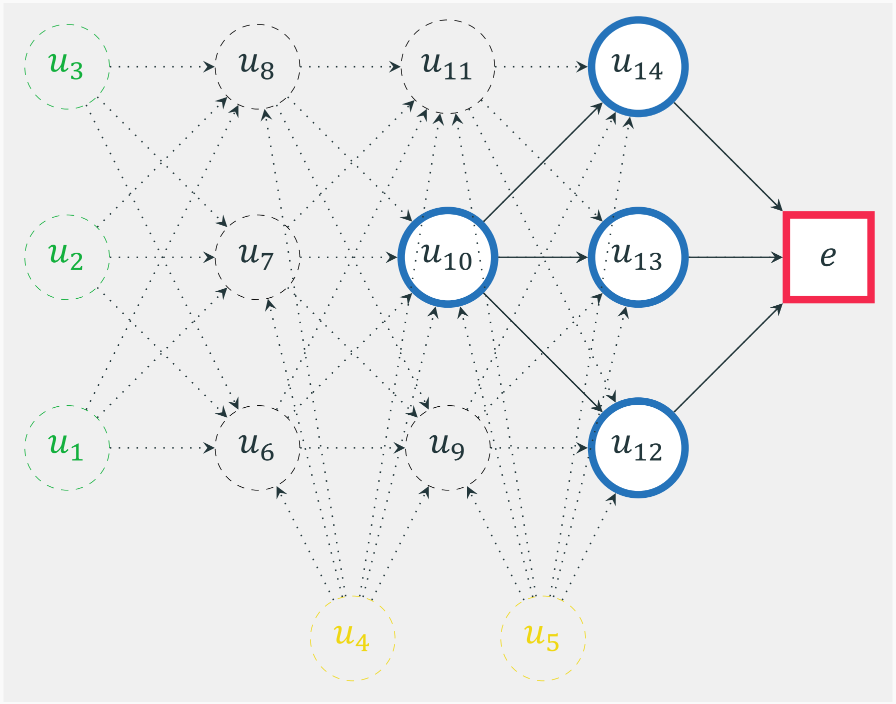
$$e(u_{12}, u_{13}, u_{14}) = (u_{12} - a)^2 + (u_{13} - b)^2 + (u_{14} - c)^2$$

Then

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



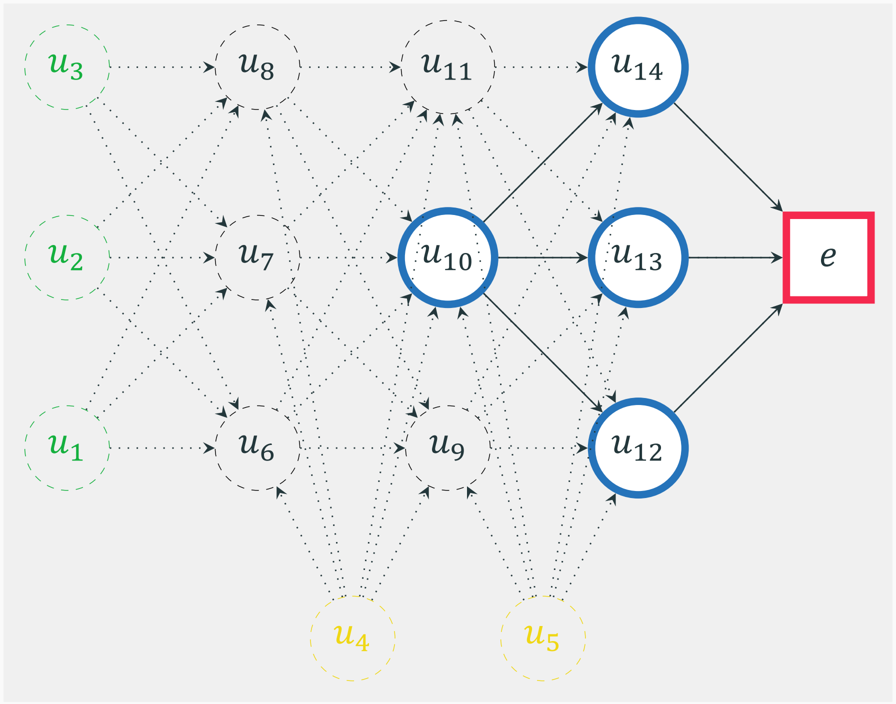
Now that we have computed $\frac{\partial e}{\partial u_{14}}$, $\frac{\partial e}{\partial u_{13}}$ and $\frac{\partial e}{\partial u_{12}}$, how do we compute $\frac{\partial e}{\partial u_{10}}$?



We can use the chain-rule:

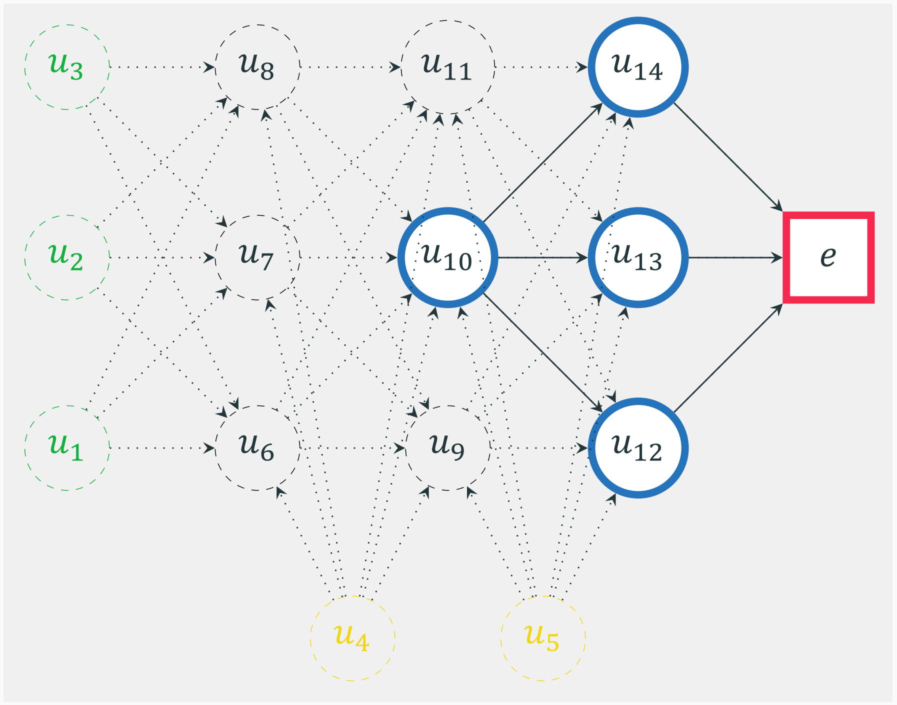
$$\frac{\partial e}{\partial u_i} = \sum_{j \in \text{Outputs}(i)} \frac{\partial u_j}{\partial u_i} \frac{\partial e}{\partial u_j}$$

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:

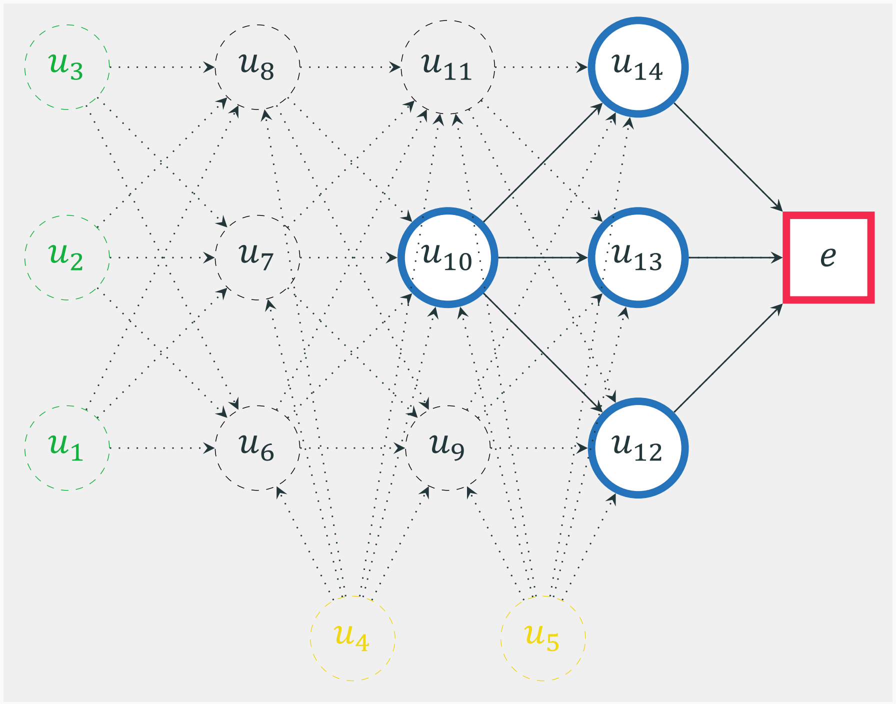
$$\frac{\partial e}{\partial u_{10}} = \frac{\partial u_{14}}{\partial u_{10}} \frac{\partial e}{\partial u_{14}} + \frac{\partial u_{13}}{\partial u_{10}} \frac{\partial e}{\partial u_{13}} + \frac{\partial u_{12}}{\partial u_{10}} \frac{\partial e}{\partial u_{12}}$$



So the chain-rule tells us that:

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

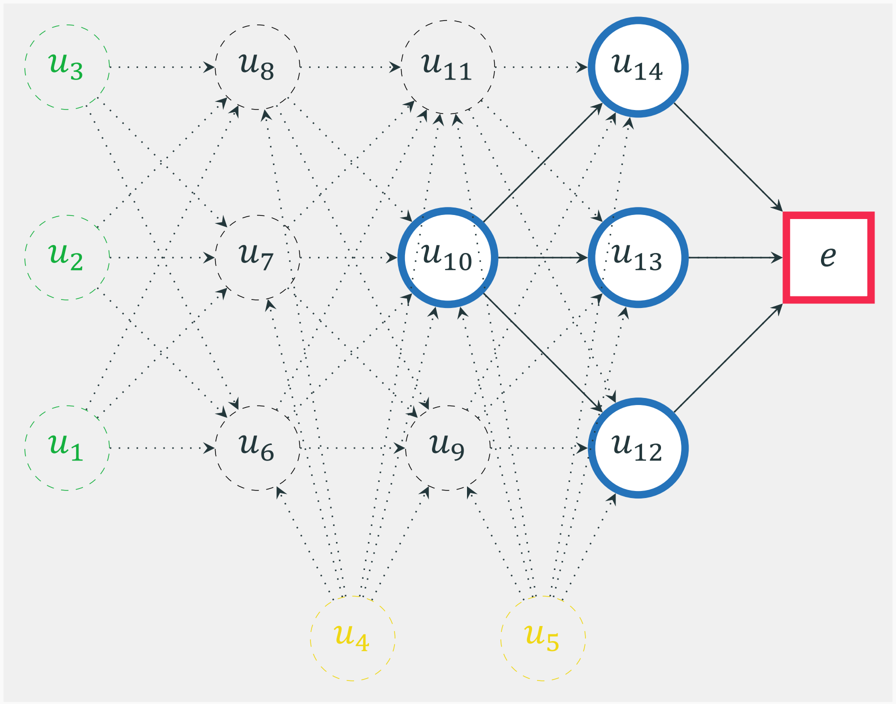
$\frac{\partial e}{\partial u_{12}}$, $\frac{\partial e}{\partial u_{13}}$ and $\frac{\partial e}{\partial u_{14}}$ have already been computed.



So the chain-rule tells us that:

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

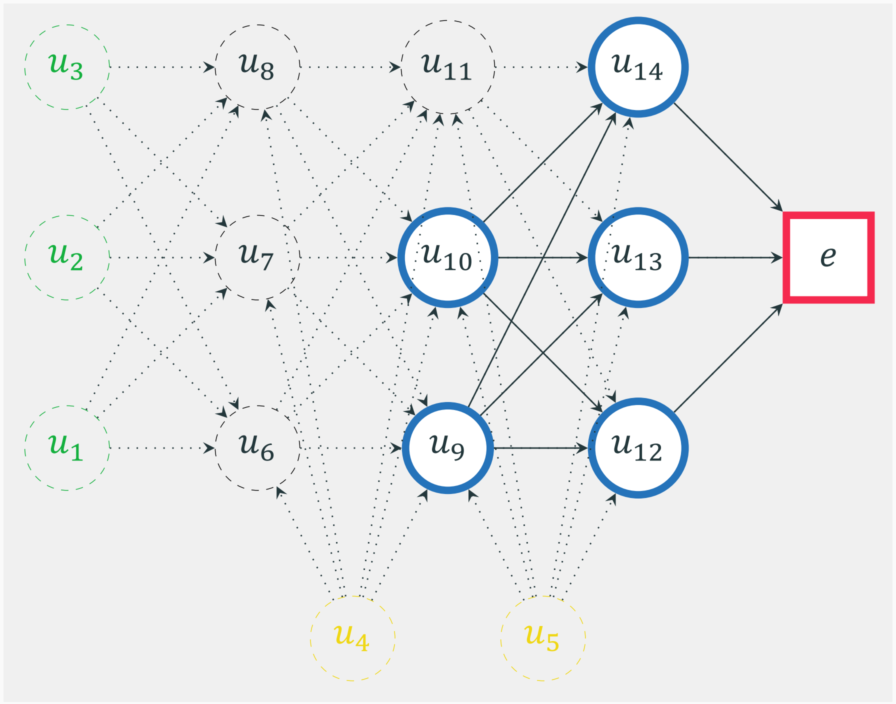
and $\frac{\partial u_{14}}{\partial u_{10}}$, $\frac{\partial u_{13}}{\partial u_{10}}$ and $\frac{\partial u_{12}}{\partial u_{10}}$ can also be derived from the definitions of the functions u_{12}, u_{13} and u_{14}.



So the chain-rule tells us that:

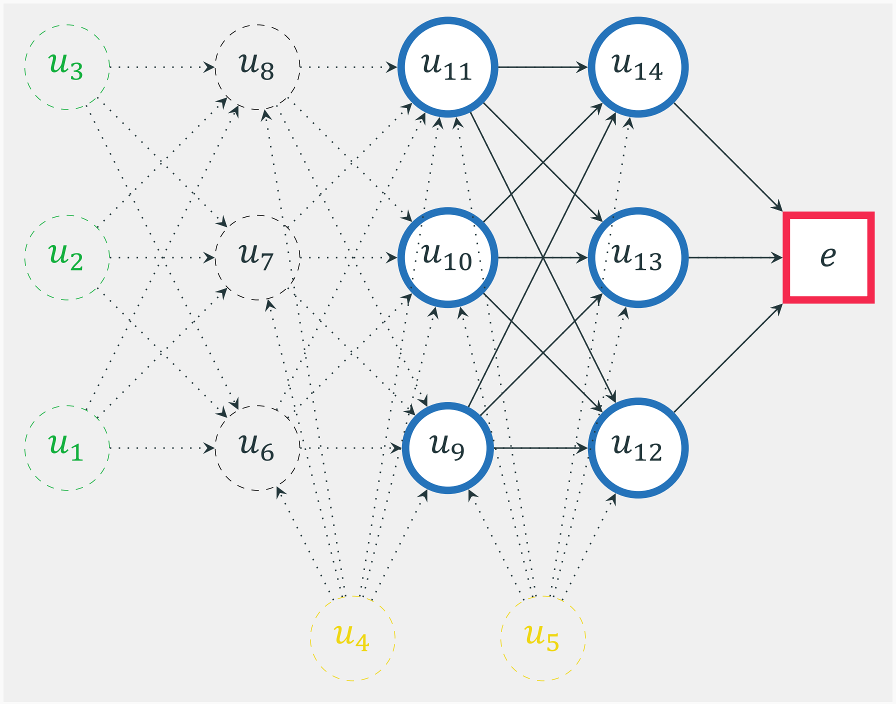
$$\frac{\partial e}{\partial u_{10}} = \frac{\partial u_{14}}{\partial u_{10}} \frac{\partial e}{\partial u_{14}} + \frac{\partial u_{13}}{\partial u_{10}} \frac{\partial e}{\partial u_{13}} + \frac{\partial u_{12}}{\partial u_{10}} \frac{\partial e}{\partial u_{12}}$$

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 $\frac{\partial u_{14}}{\partial u_{10}} = 0.2$



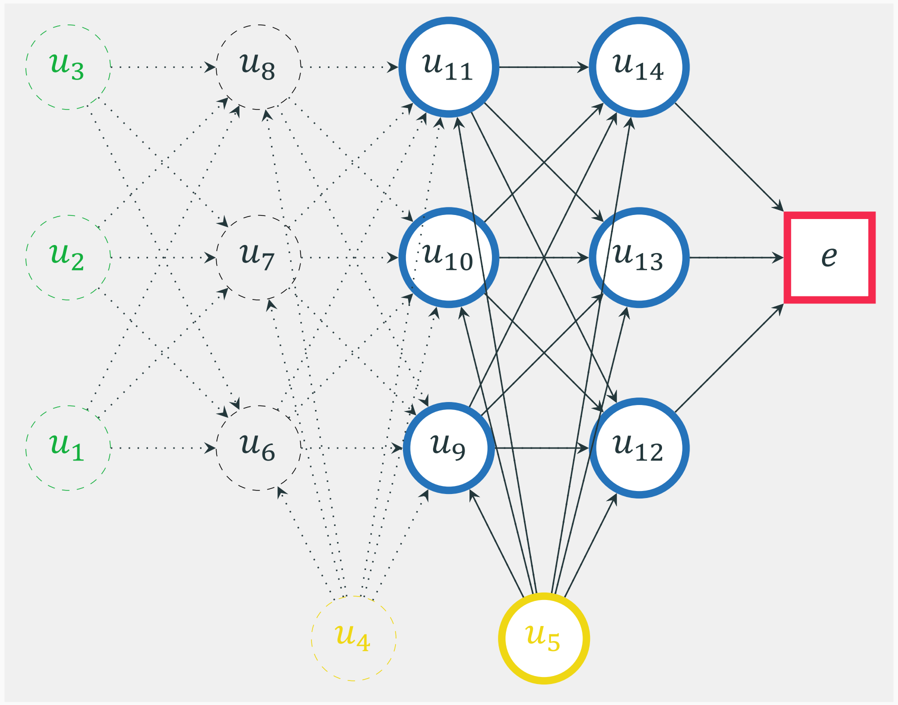
We can now propagate back the computations and derive the gradient for each node at a time.

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



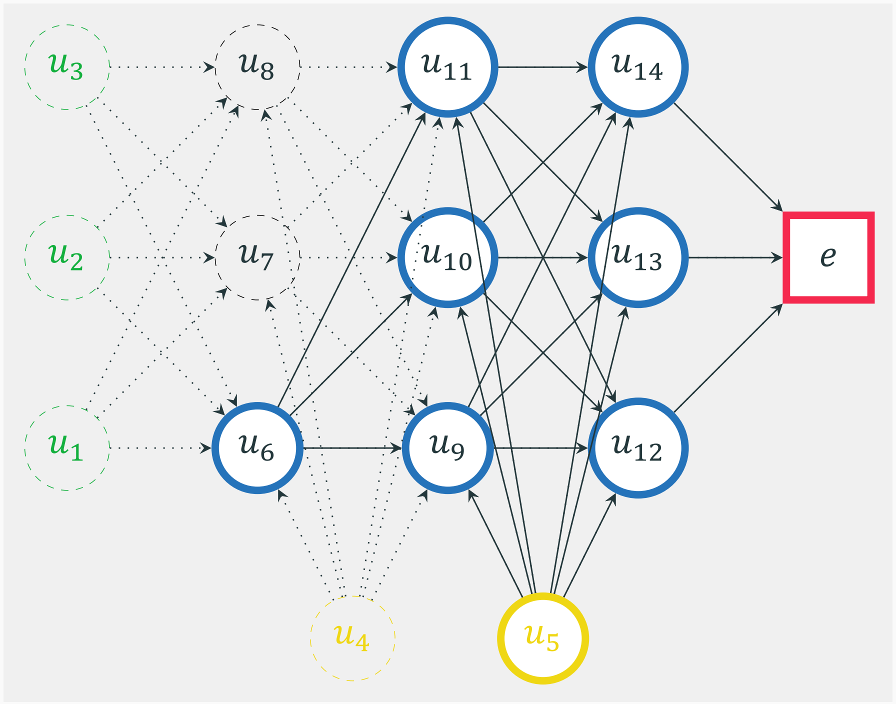
We can now propagate back the computations and derive the gradient for each node at a time.

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



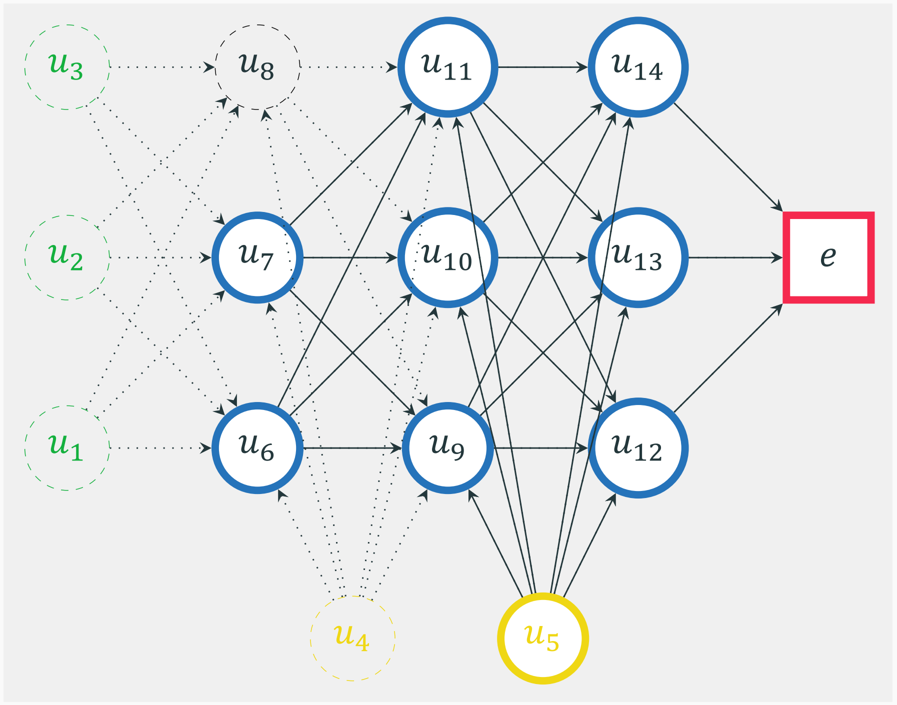
We can now propagate back the computations and derive the gradient for each node at a time.

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



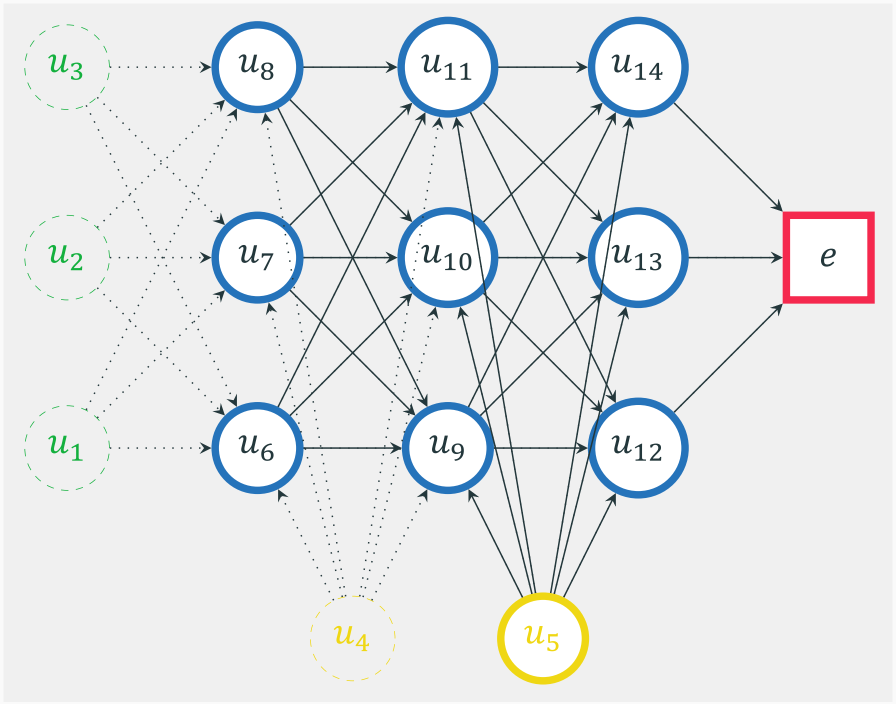
We can now propagate back the computations and derive the gradient for each node at a time.

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



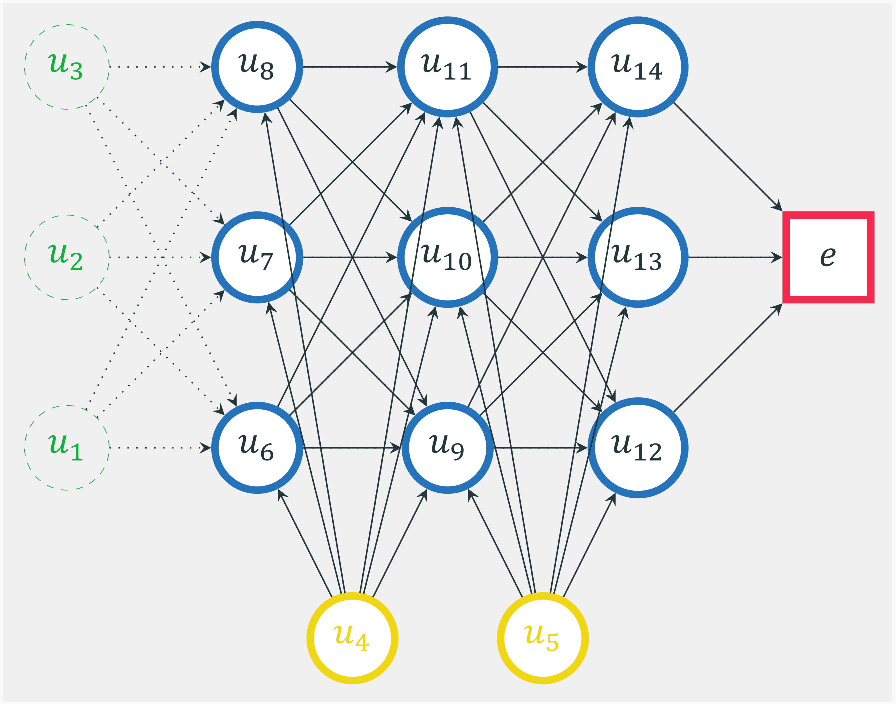
We can now propagate back the computations and derive the gradient for each node at a time.

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



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.

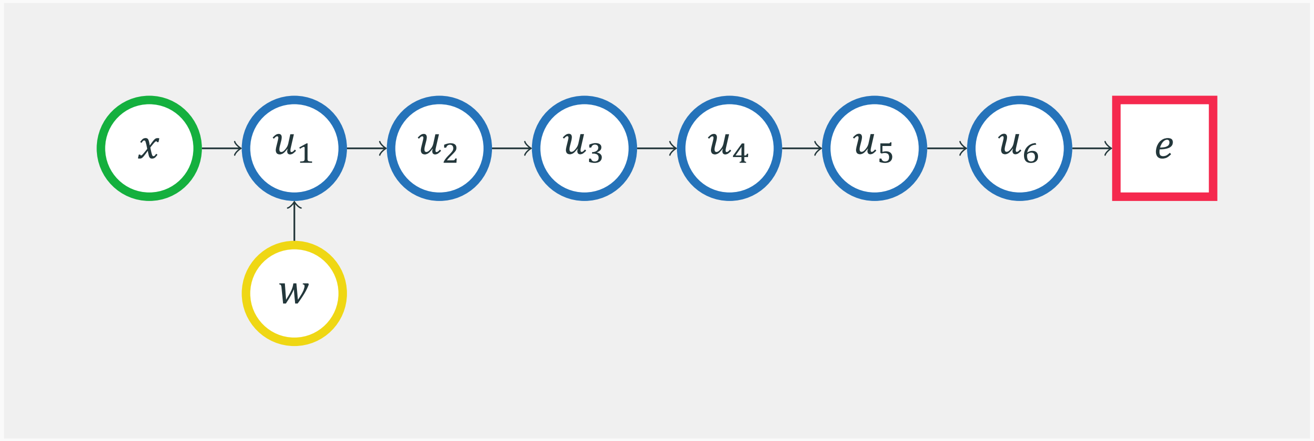
The worst case complexity of backprop is $\mathcal{O}(\mathbf{number\_of\_units}^2)$ but in practice for most network architectures it is $\mathcal{O}(\mathbf{number\_of\_units})$.

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

One major challenge when training deeper networks is the problem of vanishing gradients.

Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen (PDF) (diploma thesis). Technical University Munich, Institute of Computer Science.

Consider training the following deep network with 6 hidden layers:

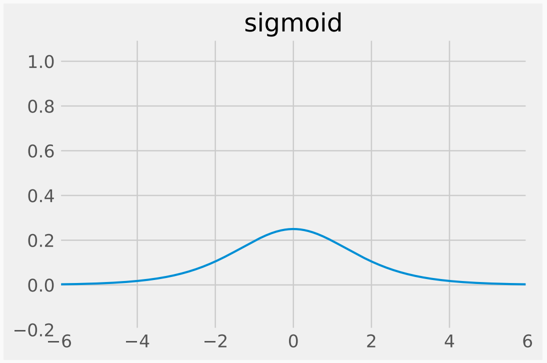


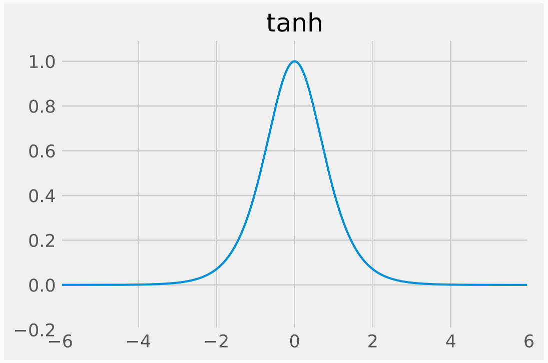
The gradient $\frac{de}{dw}$ is as follows:

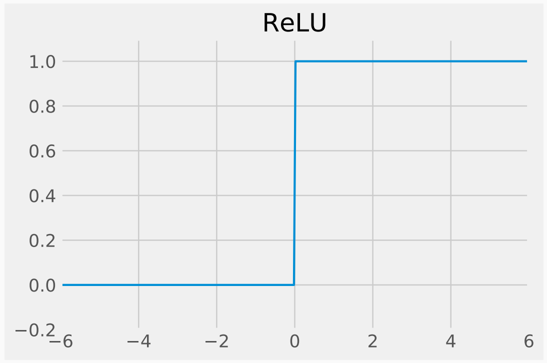
$$\frac{de}{dw} = \frac{de}{du_6}\frac{du_6}{du_5}\frac{du_5}{du_4}\frac{du_4}{du_3}\frac{du_3}{du_2}\frac{du_2}{du_1}\frac{du_1}{dw}$$

It is a product, so if any of $\left|\frac{de}{du_6}\right|, \ldots, \left|\frac{du_1}{dw}\right|$ is near zero, then the resulting gradient will also be near zero.

Below are the derivatives of some popular activation functions:







For most of the input range, the derivatives are zero or near zero. There is thus a great risk for at least one of the units to produce a near zero derivative.

When this happens we have $\frac{de}{dw} \approx 0$ and the gradient descent will get stuck.

The problem of vanishing gradients is a key difficulty when training Deep Neural Networks.

It is also one of the reasons ReLU is sometimes preferred as at least half of the range has a non-null gradient.

Recent modern network architectures try to mitigate this problem (see ResNets and LSTM).

Optimisations for Training Deep Neural Networks

So weights can be trained through a gradient descent algorithm and the gradient at each step can be computed via the Back-Propagation algorithm.

The problem is that gradient descent approaches are not guaranteed to converge to a global minimum.

Tuning the training optimisation will thus be a critical task in the life of a neural networks practitioner. Many trials and errors will be required.

A few optimisation strategies and regularisation techniques are however available to help with the convergence.

Mini-Batch Gradient Descent

Recall the gradient descent algorithm:

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

The loss function is usually constructed as the average error for all observations:

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

We've seen how to compute $\frac{\partial e}{\partial \mathbf{w}}$ for a particular observation $\mathbf{x}_i$ using backpropagation. The combined gradient for E is simply:

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

This requires evaluating the gradient for the entire dataset, which is usually not practical.

A very common approach is instead to compute the gradient over batches of the training data. For instance if the **batch size** is 16, that means that the gradient used in the gradient descend algorithm is only averaged over 16 observations. For instance, starting from observation j:

$$\frac{\partial E}{\partial \mathbf{w}} \approx \frac{1}{16} \sum_{i=j}^{j+15} \frac{\partial e}{\partial \mathbf{w}}(f(\mathbf{x}_i, \mathbf{w}), \mathbf{y}_i)$$

Next evaluation of the gradient starts at $j+16$.

This process is called mini-batch gradient descent.

In the edge case where the batch size is 1, the gradient is recomputed for each observation:

$$\frac{\partial E}{\partial \mathbf{w}} \approx \frac{\partial e}{\partial \mathbf{w}}(f(\mathbf{x}_i, \mathbf{w}), \mathbf{y}_i)$$

and the process is called Stochastic Gradient Descent (SGD)

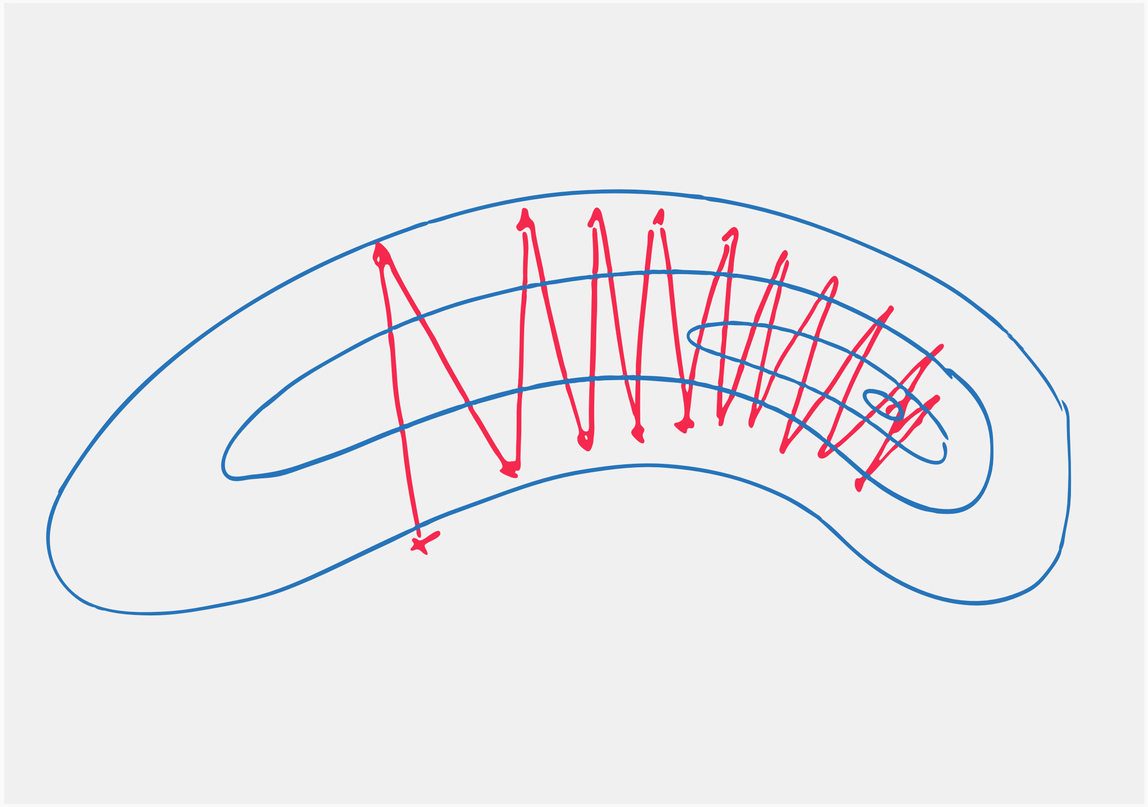
The smaller the batch size, the faster a step of the gradient descent can be done. Small batch sizes also mean that the computed gradient is a bit “noisier”, and likely to change from batch to batch. This randomness can help escape from local minimums but can also lead to poor convergence.

An epoch is a measure of the number of times all of the training vectors are used once to update the weights.

After one epoch, the gradient descent will have done $n/\text{BatchSize}$ steps.

Note that after each epoch the samples are usually shuffled so as to avoid cyclical repetitions.

Gradient Descent Algorithms



Sometimes the vanilla gradient descent is not the most efficient way. In this example the loss function forms a deep and long valley. In this scenario the gradient descent is not very efficient at reaching the minimum.

In particular, we can observe that the steepest descent of the gradient can take many changes of direction.

One technique to reduce that problem is to cool down —or **decay** — the learning rate over time.

Another approach is to reuse previous gradients to influence the current update. For instance, the **momentum update** technique tries to average out the gradient direction over time as follows:

$$\begin{aligned}\mathbf{v}^{(m+1)} &= \mu\mathbf{v}^{(m)} - \eta\frac{\partial E}{\partial \mathbf{w}}(\mathbf{w}^{(m)}) \\ \mathbf{w}^{(m+1)} &= \mathbf{w}^{(m)} + \mathbf{v}^{(m+1)}\end{aligned}$$

with $\mu \approx 0.9$

Other more sophisticated techniques exist. To name of few:

Nesterov Momentum, Nesterov's Accelerated Momentum (NAG), Adagrad, RMSprop, Adam and Nadam.

Adam and **Nadam** are known to usually perform best but this may change depending on your problem. So it's okay to try a few out.

http://cs231n.github.io/neural-networks-3/

Constraints and Regularisers

Regularisers

L2 regularisation is the most common form of regularisation. It is the Tikhonov regularisation of Least Squares. To add a L2 regularisation on a particular weight w_i, we simply add a penalty to the loss function:

$$E'(\mathbf{w}) = E(\mathbf{w}) + \lambda w_i^2$$

L1 regularisation is another common form of regularisation:

$$E'(\mathbf{w}) = E(\mathbf{w}) + \lambda |w_i|$$

L1 regularisation can have the desirable effect of settings weights w_i to zero and thus simplifying the network.

`https://keras.io/regularizers/`

Constraints

Max norm constraint. We can also enforce an absolute upper bound on the magnitude of the weights by clamping w_i. For instance:

$$w_i \to \min(w_i, c)$$

This bounds the updates and prevents the network from totally diverging in training.

`https://keras.io/constraints/`

Dropout & Layer Noise

We know that one way of fighting overfitting is to use more data.

One cheap way to obtain more data is to take the original observations and add a bit of random noise to the features. This can be done synthetically by adding noise to the original dataset (eg. adding Gaussian noise to the original features).

One step further is to add noise, during the training phase, to the hidden units themselves.

`https://keras.io/layers/noise/`

Dropout & Layer Noise

Dropout is a simple and recently introduced regularisation technique. In training, units are randomly switched off (ie. set the their output to zero).

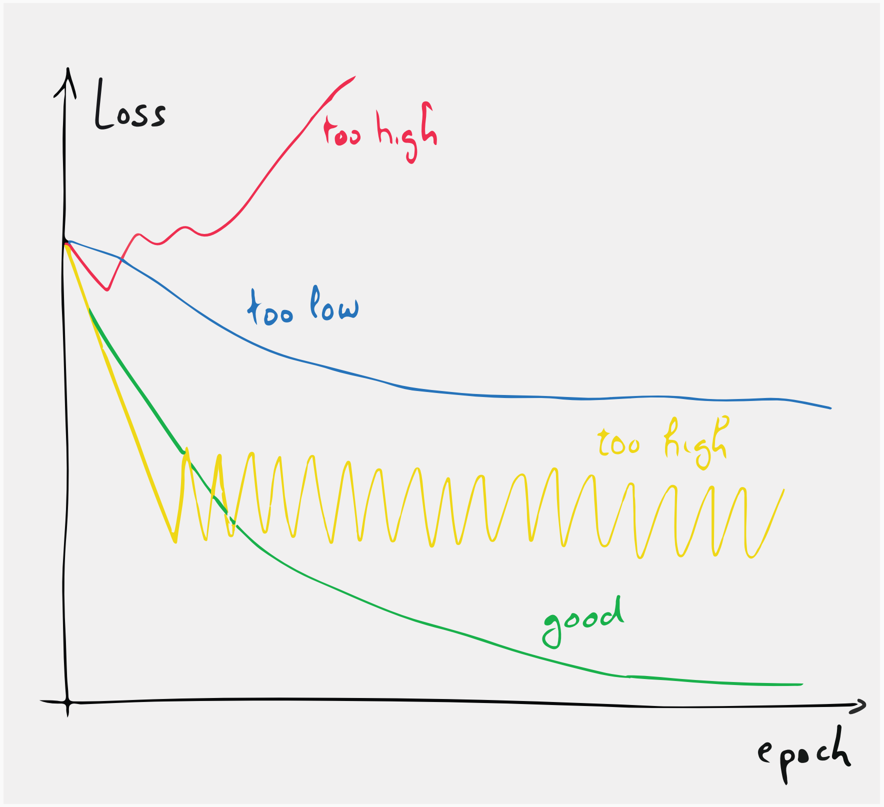
This technique can be also seen as adding a (multiplicative) noise to the layers.

`https://keras.io/layers/core/#dropout`

Monitoring

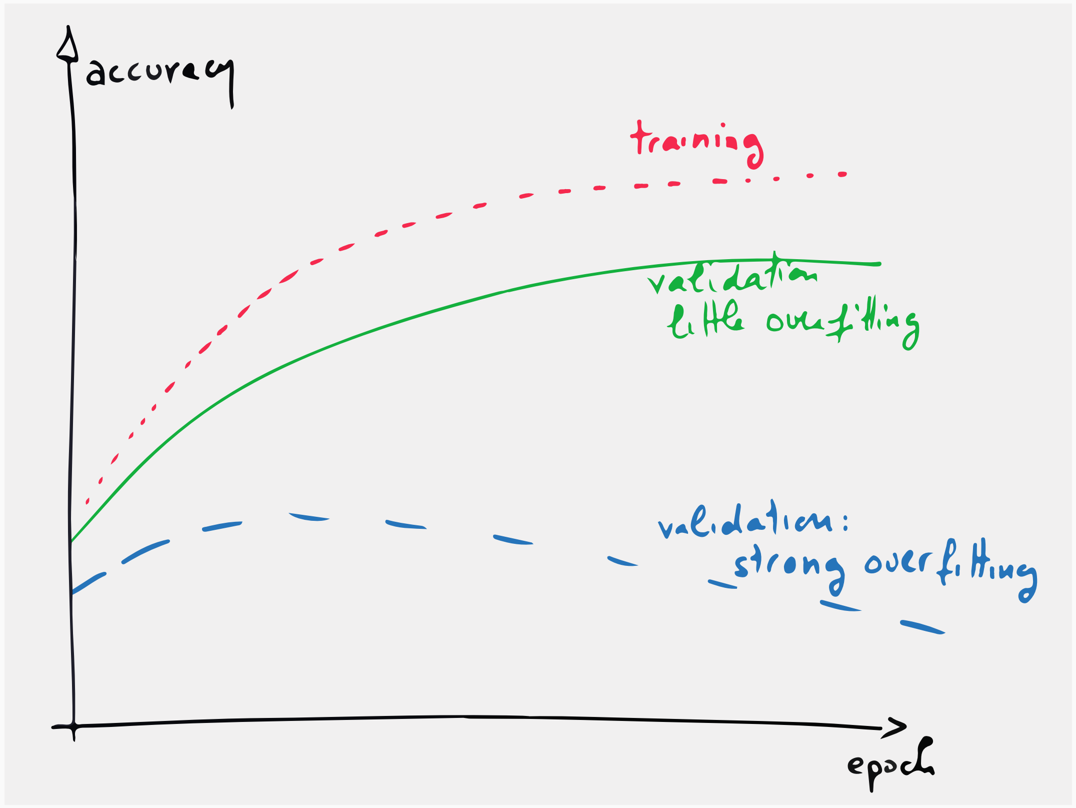
Training can take a long time. From an hour to a few days. You need to carefully monitor the loss function during training. The more information you track, the easier it is to diagnostic your training.

Monitoring the learning rate



Similarly to what we saw with logistic regression, the trend of the loss function during training can tell us whether the learning rate is suitable.

Monitoring for overfitting



Also, remember from Least Squares that good performance in training and poor performance in testing/validation is a sign of overfitting.

Take Away

You can see Deep Neural Networks as a framework to model any function as a network of basic units (the neurons).

The universal approximation theorem guarantees us that any function can indeed be approximated this way.

How to best architect the networks is main research question today. Deeper networks are known to be more efficient but harder to train because of the problem of vanishing gradients.

Training the weights of the network can be done through a gradient descent algorithm. The gradient at each step can be computed via the Back-Propagation algorithm.

Gradient descent approaches are however not guaranteed to converge to a global minimum. We must thus carefully monitor the optimisation. A few optimisation strategies and regularisation techniques are available to help with the convergence. Training networks requires a lot of trial and error.

Useful Resources

[1] Deep Learning (MIT press) from Ian Goodfellow et al. - chapters 6, 7 & 8 https://www.deeplearningbook.org

[2] Brandon Rohrer YT channel https://youtu.be/ILsA4nyG7I0

[3] 3Blue1Brown YT video https://youtu.be/aircAruvnKk

[4] 3Blue1Brown YT video https://youtu.be/IHZwWFHWa-w

[5] Stanford CS class CS231n http://cs231n.github.io

[6] Tensorflow playground https://playground.tensorflow.org

[7] Michael Nielsen's webpage http://neuralnetworksanddeeplearning.com/

Week 8 Quiz

We have seen so far a number of new terms:

`unit, hidden unit, hidden layer, epoch, minibatch, weight decay, regularisation, activation function, logit, feedforward network, backpropagation, ReLU, softmax, sigmoid, maximum likelihood, cross-entropy, loss function, error function, cost function, training set, testing set, validation set, ROC curve, F1 score, accuracy, recall, precision, confusion matrix, overfitting, underfitting, etc.`

You must be familiar with all the terms highlighted in blue or in **bold** in the handouts for the week 8 quiz. Quiz will cover all material from Lecture 0 to Lecture 6 (convolutional neural nets).

OK, so that was a lot of information. Now, it’s not all bad. The reward is that in about 20 lines of code you can create a network, train it and achieve stellar performance.

Here is an example on how to achieve 98.40% accuracy on MNIST:

https://github.com/fchollet/keras/blob/master/examples/mnist_mlp.py