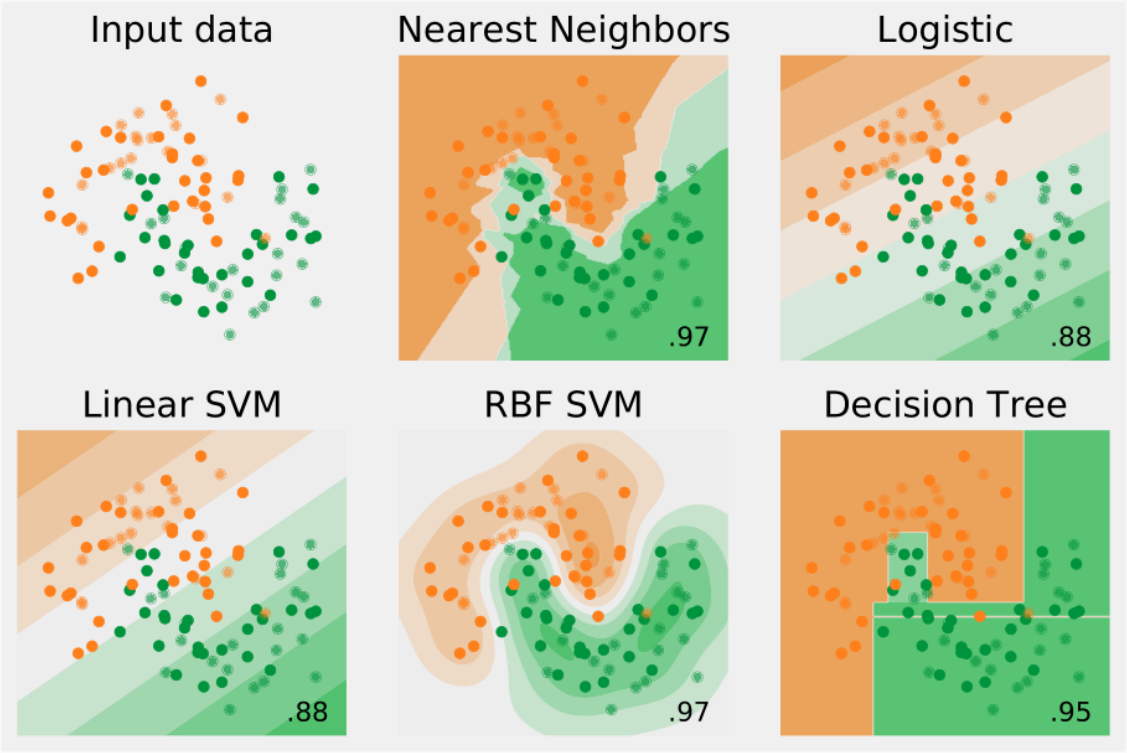
04 - Evaluating Classifier Performance

François Pitié

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

We have seen a number of classifiers (Logistic Regression, SVM, kernel classifiers, Decision Trees, k-NN) but we still haven't talked about their performance.

Recall some of results of classifiers:



How do we measure the performance of a classifier?

How do we compare them?

We need metrics that everybody can agree on.

Metrics for Binary Classifiers

True Positives, False Positives, True Negatives, False Negatives

If you have a binary problem with classes 0 (e.g. negative/false/fail) and 1 (e.g. positive/true/success), you have 4 possible outcomes:

True Positive : you predict $\hat{y} = 1$ and indeed $y = 1$.

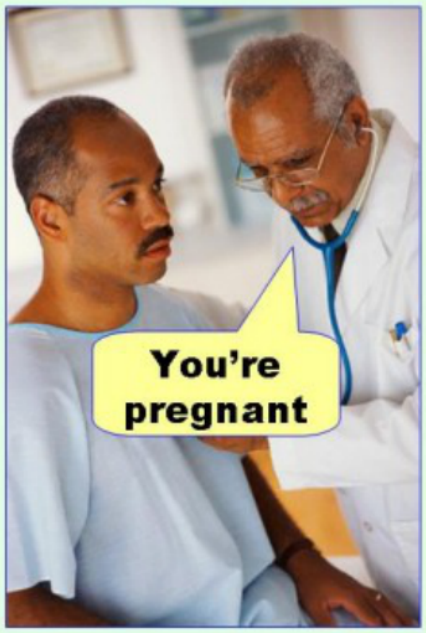
True Negative : you predict $\hat{y} = 0$ and indeed $y = 0$.

False Negative : you predict $\hat{y} = 0$ but in fact $y = 1$.

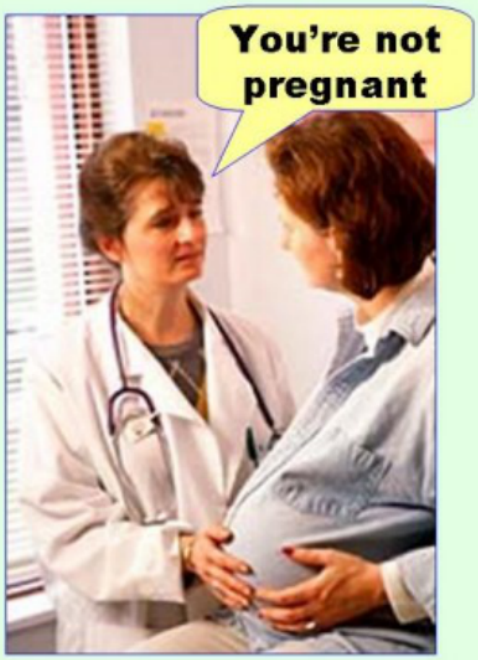
False Positive : you predict $\hat{y} = 1$ but in fact $y = 0$.

In statistics, False Positives are often called type-I errors and False Negatives type-II errors.

Type I error
(false positive)



Type II error
(false negative)



Confusion Matrix

A confusion matrix is a table that reports the number of false positives, false negatives, true positives, and true negatives for each class.

	actual: 0	actual: 1
predicted: 0	#TN	#FN
predicted: 1	#FP	#TP

Incorrect

Correct

For instance, on the first database slide 3

NN:

	actual: 0	actual: 1
predicted: 0	TN=166	FN=21
predicted: 1	FP=25	TP=188

Logistic Regression:

	actual: 0	actual: 1
predicted: 0	TN=152	FN=35
predicted: 1	FP=42	TP=171

Linear SVM:

	actual: 0	actual: 1
predicted: 0	TN=148	FN=39
predicted: 1	FP=41	TP=172

RBF SVM:

	actual: 0	actual: 1
predicted: 0	TN=162	FN=25
predicted: 1	FP=17	TP=196

Decision Tree:

	actual: 0	actual: 1
predicted: 0	TN=170	FN=17
predicted: 1	FP=29	TP=184

From TP, TN, FP, FN, we can derive a number of popular metrics.

Recall (also called sensitivity or true positive rate) is the probability that a positive example is indeed predicted as positive. In other words it is the proportion of positives that are correctly labelled as positives.

$$\text{recall} = \frac{\text{TP}}{P} = \frac{\text{TP}}{\text{TP} + \text{FN}} = p(\hat{y} = 1 | y = 1)$$

Precision is the probability that a positive prediction is indeed positive:

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} = p(y = 1 | \hat{y} = 1)$$

How many of your predictions were right?

False Positive Rate is the proportion of negatives that are incorrectly labelled as positive:

$$\text{FP rate} = \frac{\text{FP}}{N} = \frac{\text{FP}}{\text{FP} + \text{TN}} = p(\hat{y} = 1 | y = 0)$$

Accuracy is the probability that a prediction is correct:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{P + N} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$
$$= p(\hat{y} = 1, y = 1) + p(\hat{y} = 0, y = 0)$$

F1 score is the harmonic mean of precision and recall:

$$F_1 = 2 \cdot \frac{\text{recall} \cdot \text{precision}}{\text{precision} + \text{recall}} = \frac{2\text{TP}}{2\text{TP} + \text{FP} + \text{FN}}$$

Attempt at a single metric
– combining precision & recall

Other derived metrics exist
(see [https://en.wikipedia.org/wiki/Precision_and_recall])

But remember that since there are two types of errors (false positives and false negatives), you will always need at least two metrics to capture the performance of a classifier.

Performing well on a single metric is usually meaningless. A good classifier should perform well on 2 metrics.

Example

Consider a classifier with this confusion matrix:

	actual: 0	actual: 1
predicted: 0	TN=70	FN=5
predicted: 1	FP=15	TP=10

The actual number of positives (class 1) is 15 and the actual number of negatives is 85 (class 0).

The recall is TP/(TP+FN)=10/(5+10)=66.7%

66.7% of actual class 1's will be labelled correctly

The accuracy is (TP+TN)/(TP+FN+TN+FP)=(70+10)/100=80%.

80% of predictions across whole set correctly labelled

If we take a classifier (A) that always returns 1, the confusion matrix for the same problem becomes:

	actual: 0	actual: 1
predicted: 0	TN=0	FN=0
predicted: 1	FP=85	TP=15

$\frac{TP}{P}$

and its recall is 15/(15+0) = 100%!! → accuracy = $\frac{15+0}{15+85}$ = 15%...

If we take instead a classifier (B) that always returns 0, we have:

	actual: 0	actual: 1
predicted: 0	TN=85	FN=15
predicted: 1	FP=0	TP=0

$\frac{TP+TN}{P+N}$

and its accuracy is (85+0)/(100) = 85%!!

↳ recall = $\frac{0}{15}$ = 0%...

But both classifiers (A) and (B) are non informative and really poor choices but you need two metrics to see this:

For (A), the recall is 100% but the accuracy is only 15%.

For (B), the accuracy is 85% but the recall is 0%.

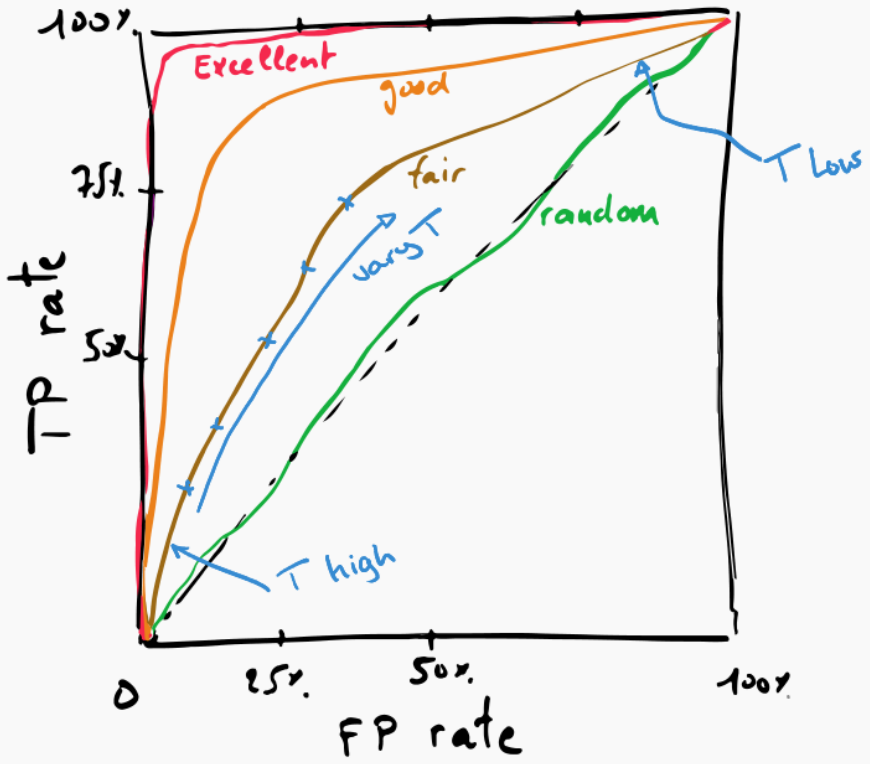
Conclusion: you need two metrics.

ROC

When you start measuring the performance of a classifier, chances are that you can tune a few parameters. For instance, if your classifier is based on a linear classifier model $y = [\mathbf{x}^\mathsf{T}\mathbf{w} > T]$, you can tune the threshold value T. Increasing T means that your classifier will be more conservative about classifying points as $y = 1$.

By varying the parameter T, we can produce a family of classifiers with different performances. We can report the $\mathbf{FP}$ rate = FP/(FP+TN) and $\mathbf{TP}$ rate = TP/(TP+FN) for each of the different values of T on a graph called the receiver operating characteristic curve, or R.O.C. curve.

Here is an example showing the ROC curves for 4 different classifiers.



Can compare curves using area under the curve

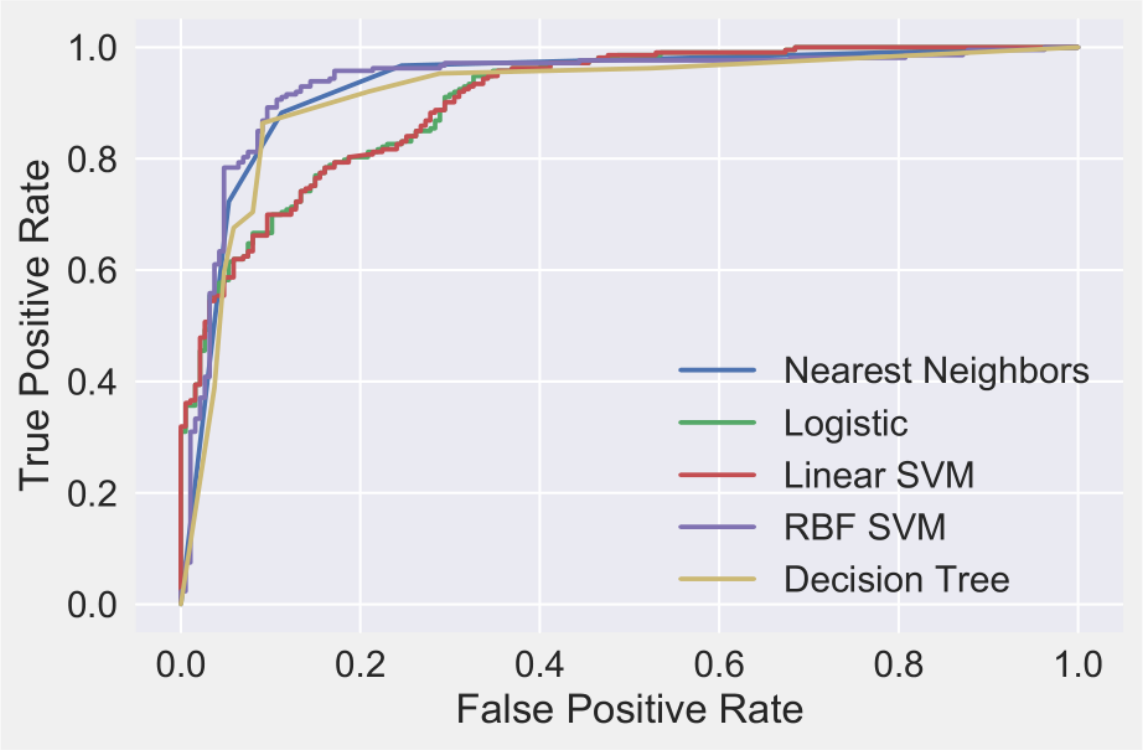
Higher = better

A perfect classifier will have a TP rate or 100% and a FP rate of 0%.

A random classifier will have TP rate equal to the FP rate.

If your ROC curve is below the random classifier diagonal, then you are doing something wrong.

Below are reported a few ROC curves for the problem of slide 3.



Multiclass Classifiers

Binary metrics don't adapt nicely to problems where there are more than 2 classes.

For multiclass problems with n classes, there are $n-1$ possible ways of miss-classifying each class. Thus there are $(n-1) \times n$ types of errors in total.

You can always present your results as a confusion matrix.

For instance:

	actual: 0	actual: 1	actual: 2
predicted: 0	102	10	5
predicted: 1	8	89	12
predicted: 2	7	11	120

No longer TP, FP, TN, FN

You can also think of your multiclass problem as a set of binary problems (does an observation belong to class k or not), and then aggregate the binary metrics in some way.

Next are presented two ways of aggregating metrics for multiclass problems.

In micro averaging, the metric (e.g. precision, recall, F1 score) is computed from the combined true positives, true negatives, false positives, and false negatives of the K classes.

For instance the micro-averaged precision is:

$$\mathrm{microPRE} = \frac{\mathrm{microTP}}{\mathrm{microTP} + \mathrm{microFP}}$$

with $\mathrm{microTP} = \mathrm{TP}_1 + \cdots + \mathrm{TP}_K$, and $\mathrm{microFP} = \mathrm{FP}_1 + \cdots + \mathrm{FP}_K$

In macro-averaging, the performances are averaged over the classes:

$$\mathrm{macroPRE} = \frac{\mathrm{PRE}_1 + \cdots + \mathrm{PRE}_K}{K} \qquad \text{where} \quad \mathrm{PRE}_k = \frac{\mathrm{TP}_k}{\mathrm{TP}_k + \mathrm{FP}_k}$$

Example

```
y_true = [0, 1, 2, 0, 1, 2, 2]
y_pred = [0, 2, 1, 0, 0, 1, 0]
```

we have $\mathrm{TP}_0 = 2$, $\mathrm{TP}_1 = 0$, $\mathrm{TP}_2 = 0$, $\mathrm{FP}_0 = 2$, $\mathrm{FP}_1 = 2$, $\mathrm{FP}_2 = 1$

$$\mathrm{microPRE} = \frac{2+0+0}{(2+0+0)+(1+2+1)} = 0.286$$

$$\mathrm{macroPRE} = \frac{1}{3}\left(\frac{2}{2+2} + \frac{0}{0+2} + \frac{0}{0+1}\right) = 0.167$$

Training/Validation/Testing Sets

Now that we have established metrics, we still need to define the data that will be used for evaluating the metric.

You usually need:

a Training set that you use for learning the algorithm.

a Dev or Validation set, that you use to tune the parameters of the algorithm.

a Test set, that you use to fairly assess the performance of the algorithm. You should not try to optimise for the test set.

Why so many sets? Because you want to avoid over-fitting.

Say your model has many parameters. With enough training, it is likely to overfit your training set. Thus we need a testing set to check the performance on unseen data.

Now, if you tune the parameters of your algorithm to perform better on the testing set, you are likely to overfit it as well.

But we want to use the testing set as a way of accurately estimating the performance on unseen data.

Thus we use a different set, the dev/validation set, for any parameter estimation.

Important: the test and dev sets should contain examples of what you ultimately want to perform well on, rather than whatever data you happen to have for training.

How large do the dev/test sets need to be?

Training sets: as large as you can afford.

Validation/Dev sets with sizes from 1,000 to 10,000 examples are common. With 10,000 examples, you will have a good chance of detecting an improvement of 0.1%.

Test sets should be large enough to give high confidence in the overall performance of your system. One popular heuristic had been to use 30% of your data for your test set. This is makes sense when you have say 100 to 10,000 examples but not anymore as it is common you might have nowadays billion of training examples.

Take Away from a Practitioner's point of view

When approaching a new project, your first steps should be to design your Training/Validation/Test sets and decide on the metrics. Only then should you start thinking about which classifier to train.

Remember that the metrics are usually not the be-all and end-all of the evaluation. Each metric can only look at a particular aspect of your problem. You need to monitor multiple metrics.

As the project progresses, it is expected that the datasets will be updated and that new metrics will be introduced.

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

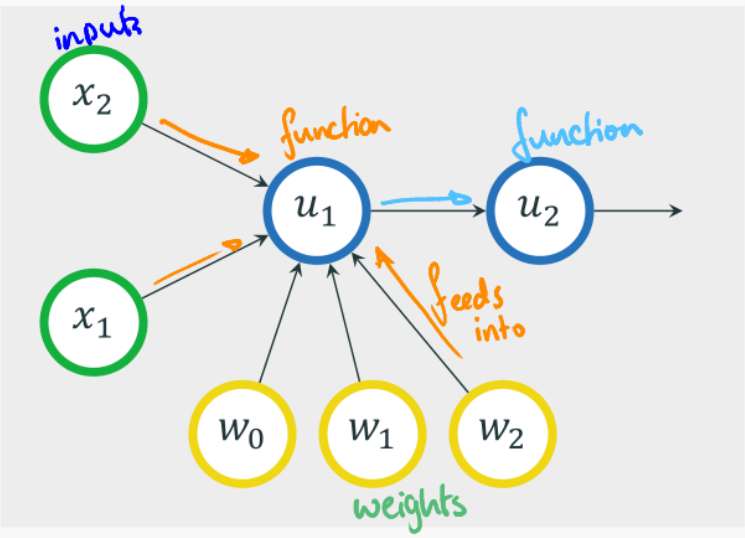
Remember Logistic Regression? The output of the model was

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

This was your first neural network.

Why is it a Network? → "Graph of Operations"

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



$$u_1 = w_0 + w_1 x_1 + w_2 x_2$$

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

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**.

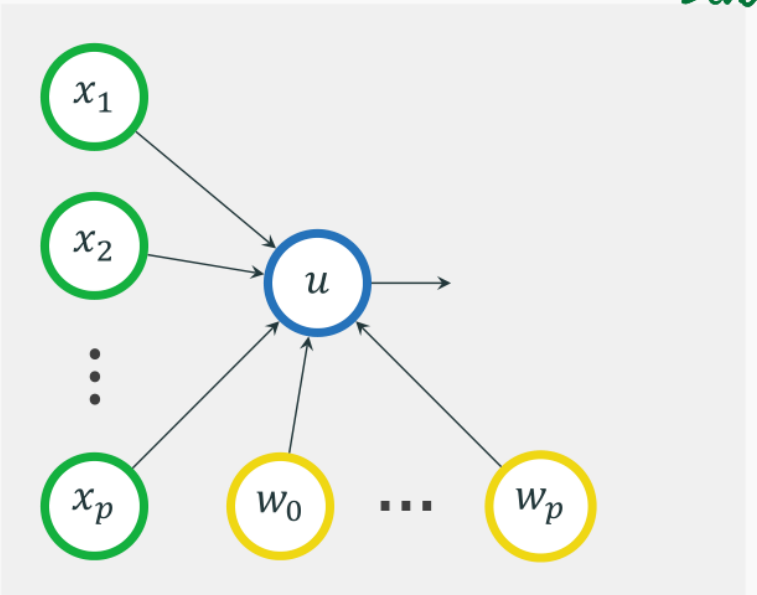
The starting units (leaves of the graph) correspond either to input values (eg. x_1, x_2), or model parameters (eg. w_0, w_1, w_2). All other units (eg. u_1, u_2) correspond to function outputs.

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:

'Dendrites' = inputs from other functions

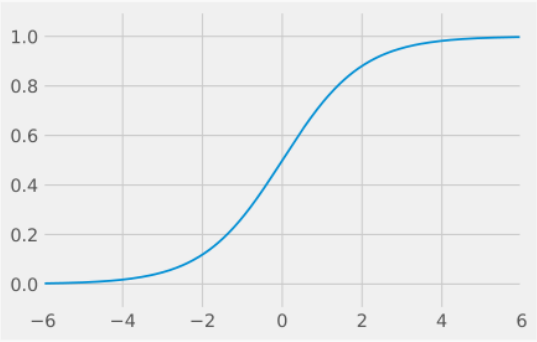


$$u = f(w_0 + \sum_{i=1}^{p} w_i x_i)$$

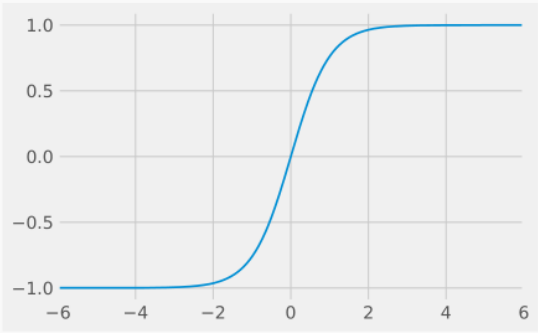
activation function

Activation Functions

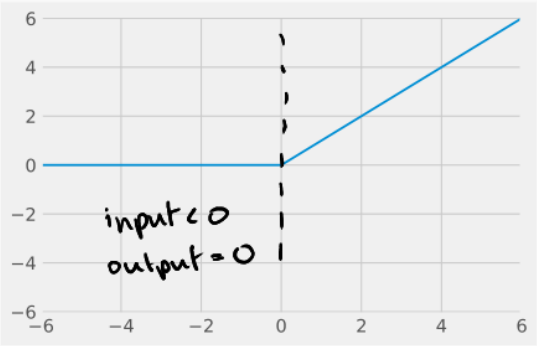
Many activation functions exist. Here are the most popular:



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



tanh: $z = \tanh(z)$



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

Rectified Linear Unit → returns no negative output

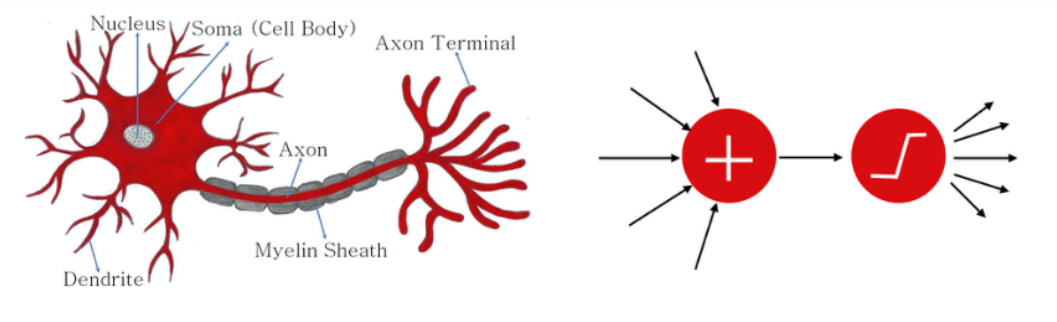
Activation Functions

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.

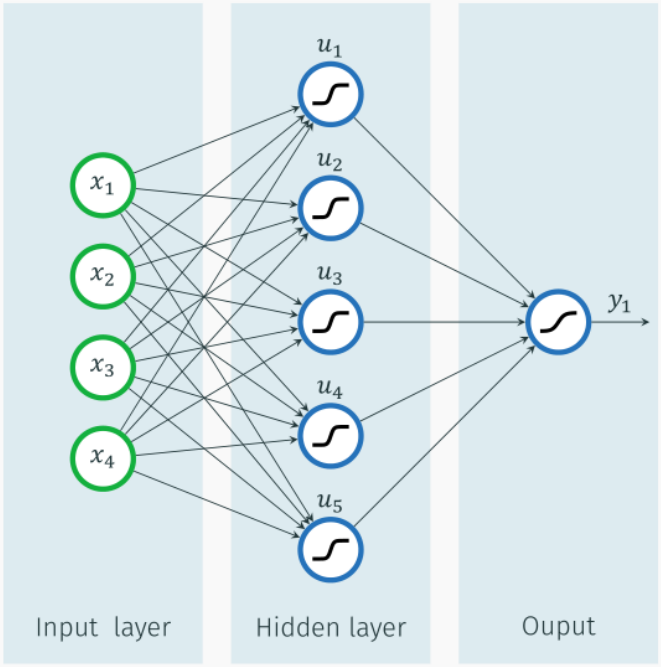
Neurons

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.

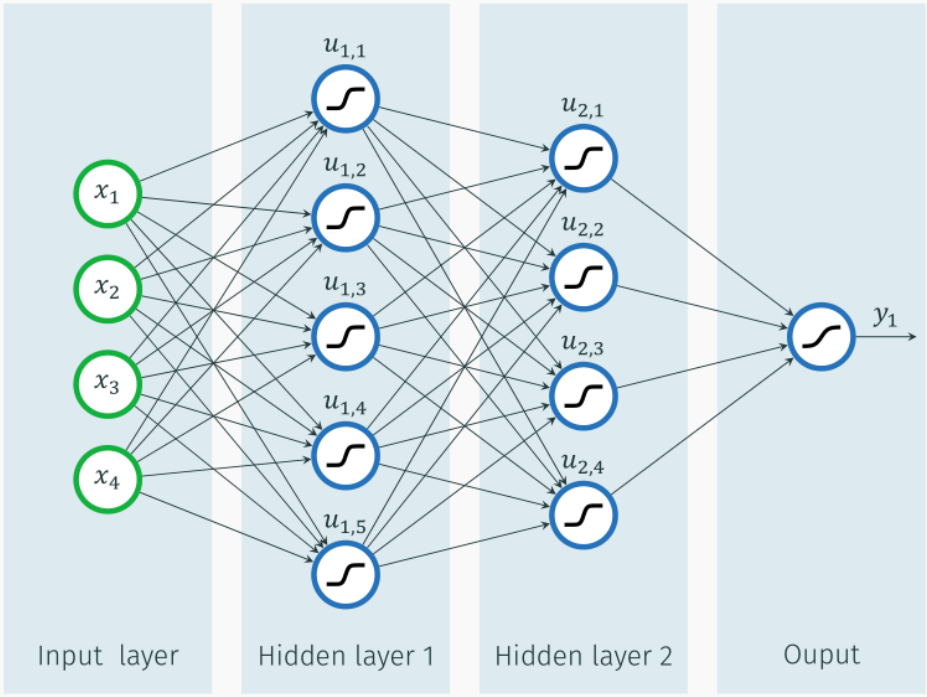


Each blue unit is a neuron with its activation function.

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.

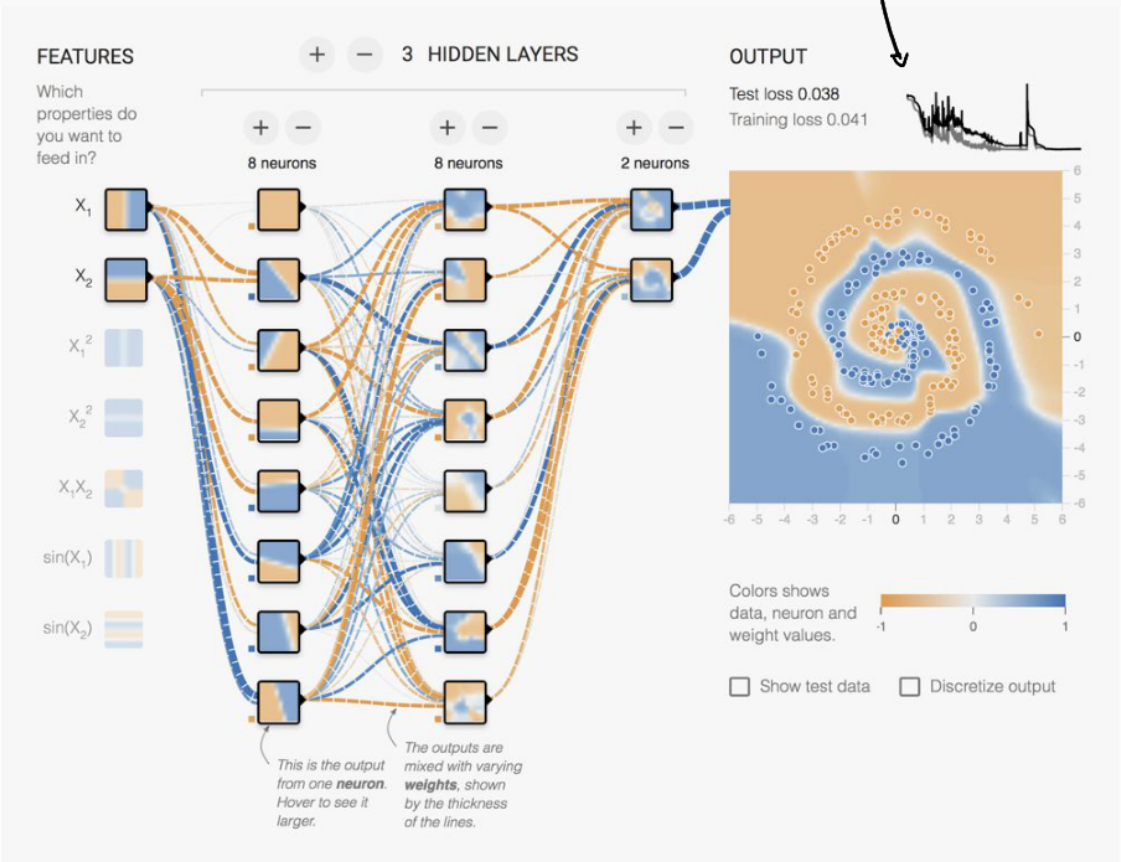
Although the universal theorem tells us you only need one hidden layer, all recent works show that deeper networks require far fewer parameters and generalise better to the testing set.

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.

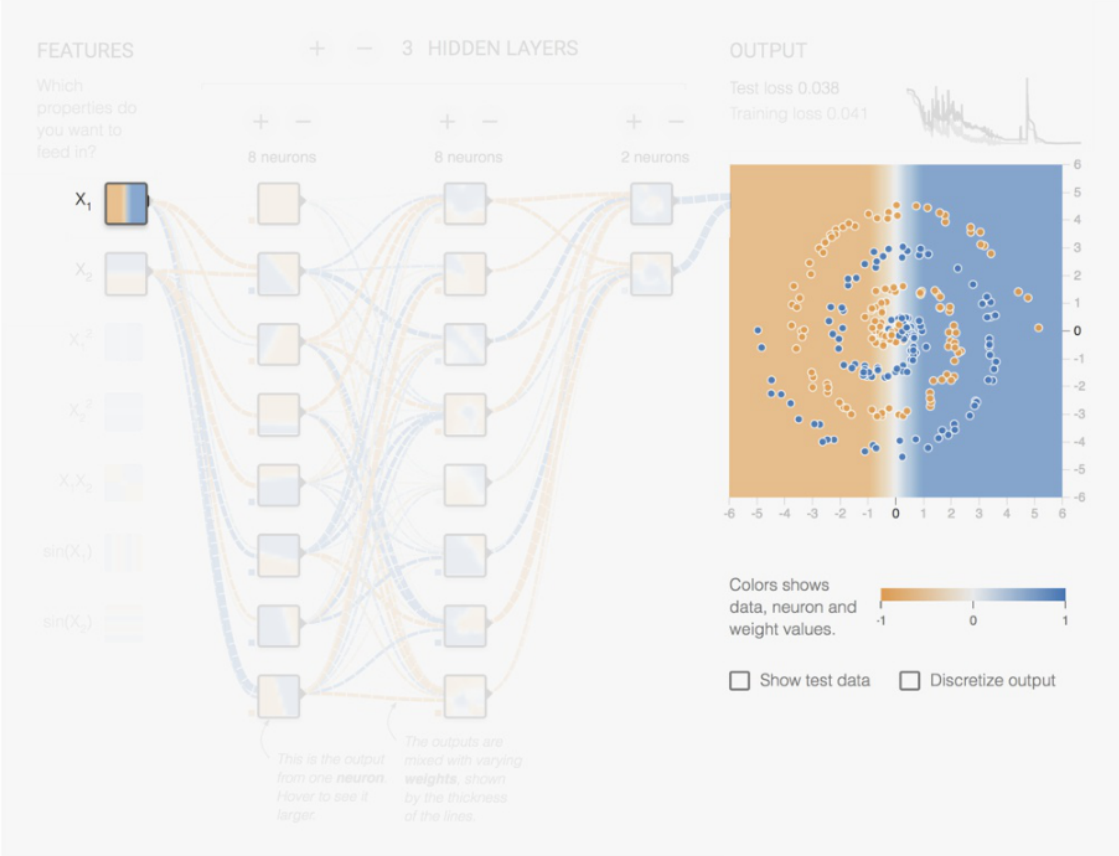
This is the main topic of research today. We know that anything can be modelled as a neural net. The challenge is to architect networks that can be efficiently trained and generalise well.

http://playground.tensorflow.org/

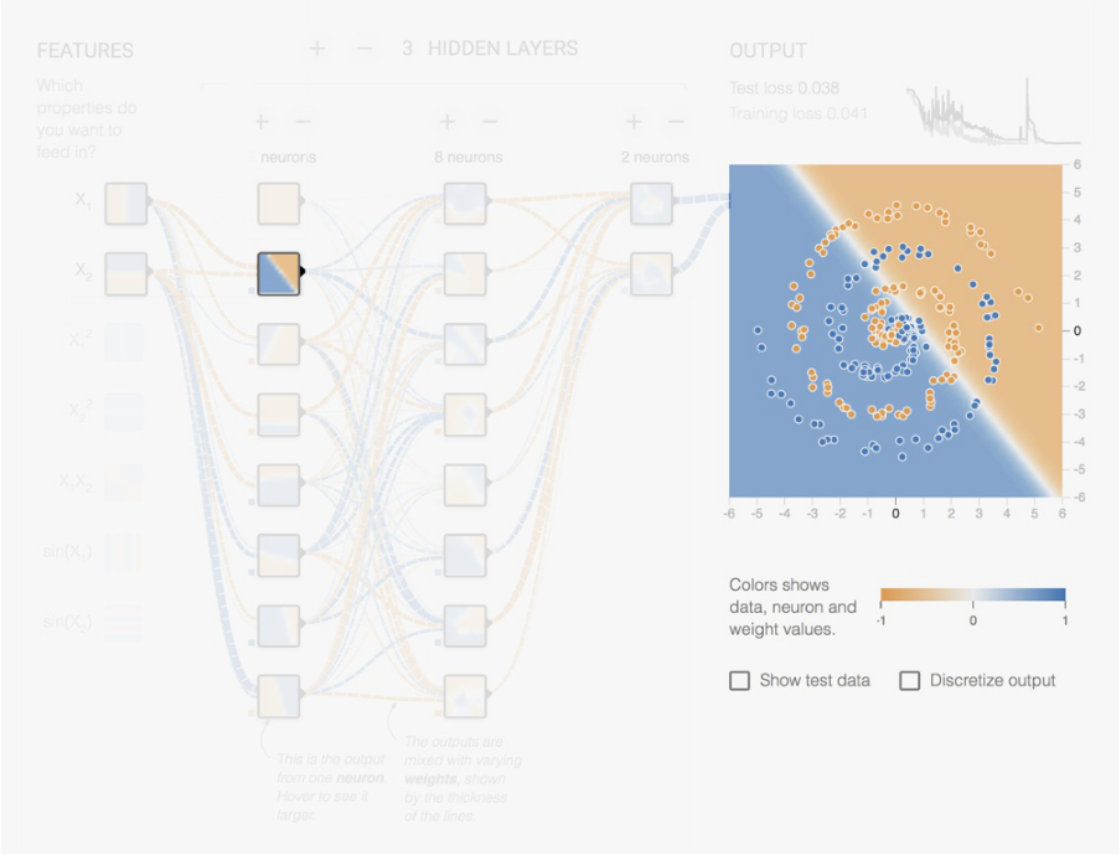
→ Gradient Descent doesn't guarantee convergence on true answer



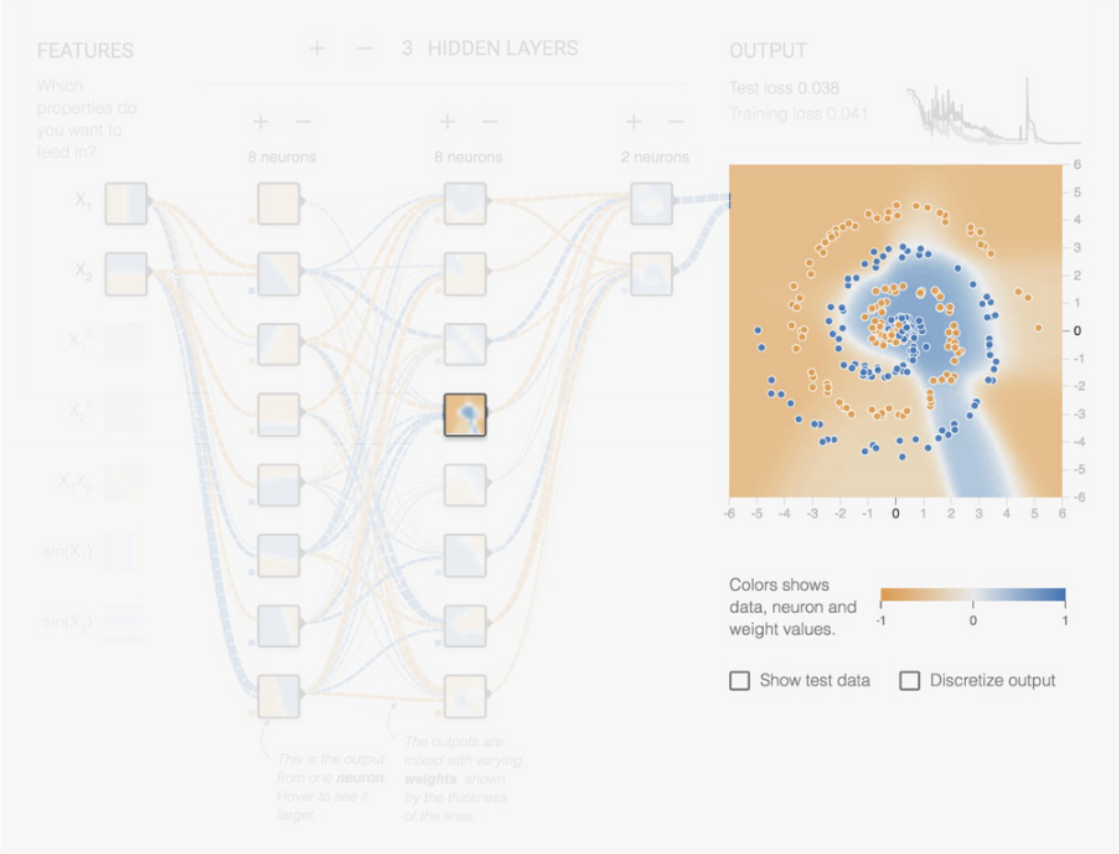
Here is a network with 3 hidden layers of respectively 8, 8 and 2 units.



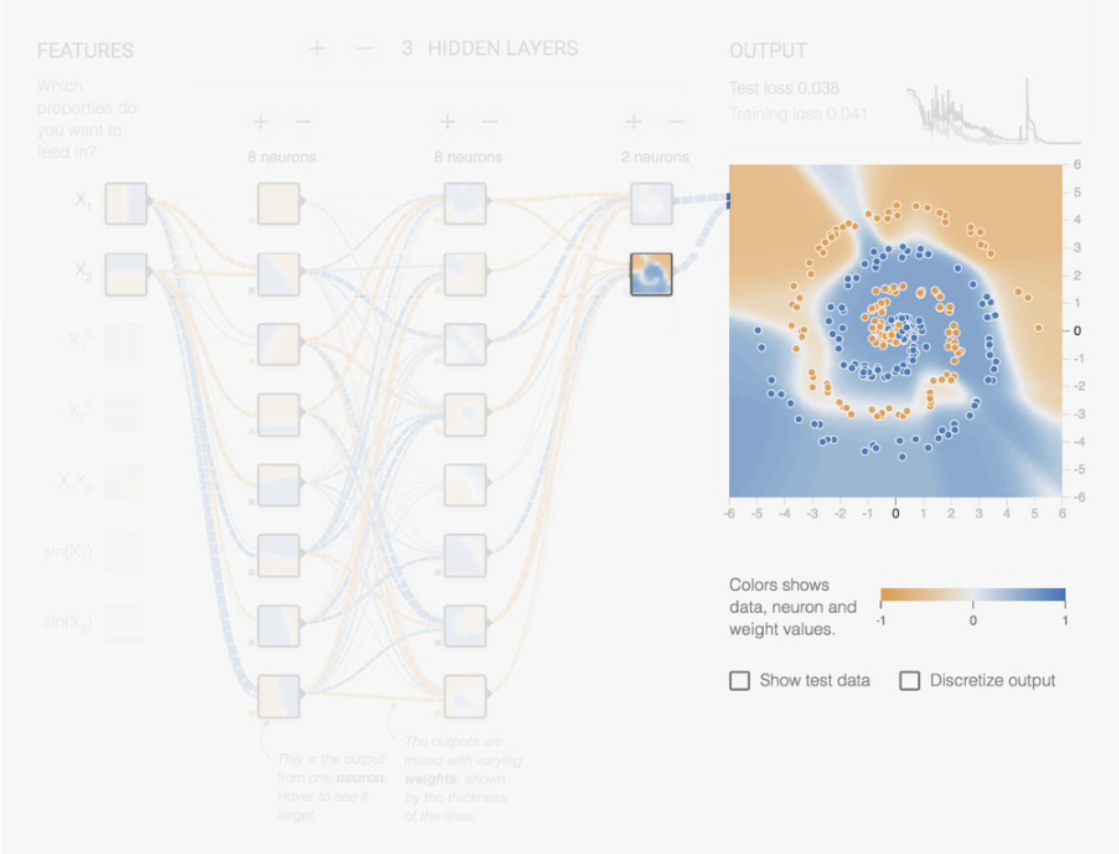
The original features are the x and y coordinates



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.

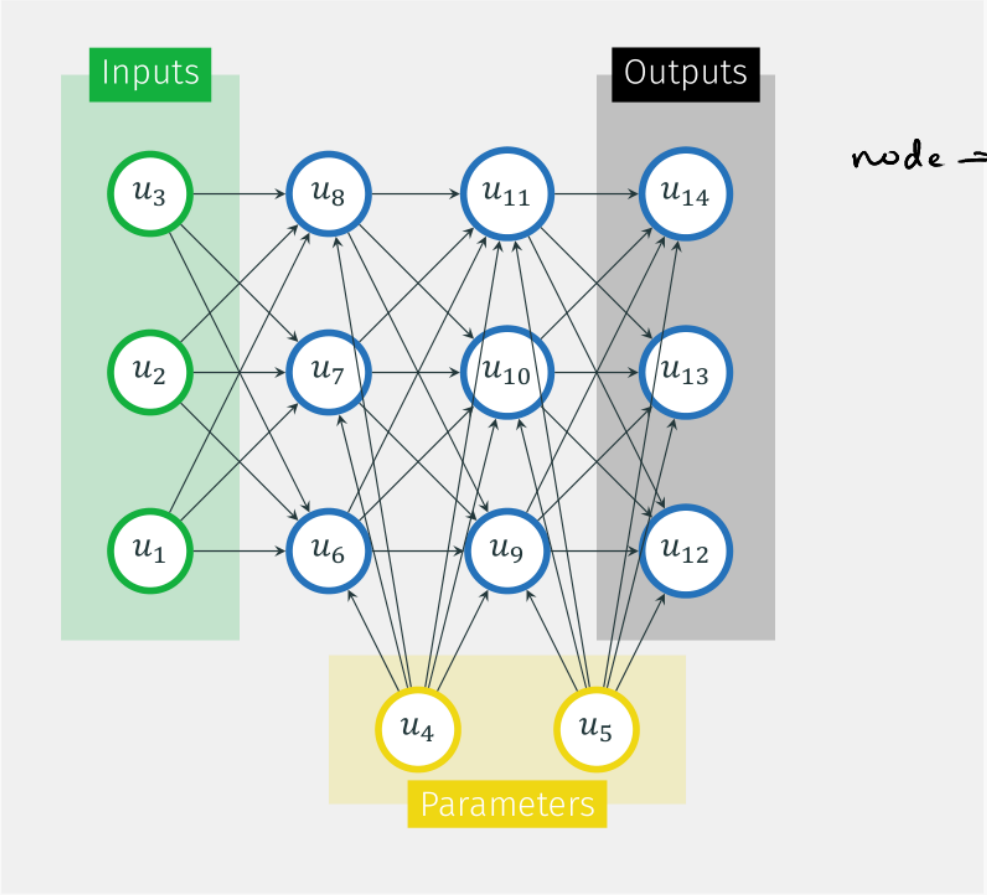
Training

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.



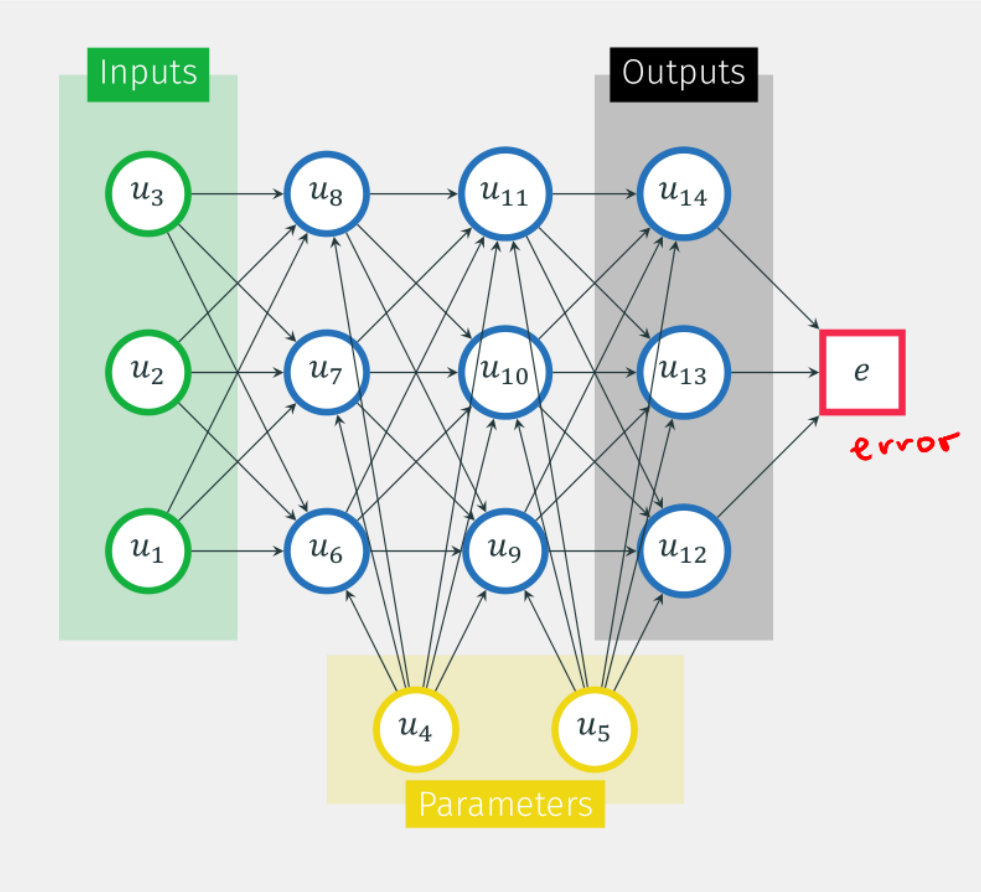
Example of a graph of operations for neural net evaluation.

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 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.

second layer

first layer

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:

output is function of hidden layers

each node/unit is a function of the inputs

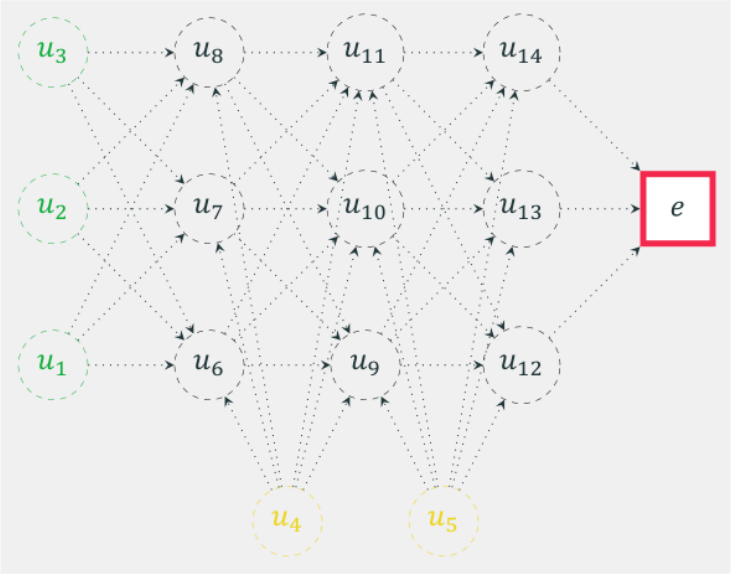
$$\frac{\partial z}{\partial x} = \sum_k \frac{\partial z}{\partial u_k} \frac{\partial u_k}{\partial x} \tag{1}$$

Example

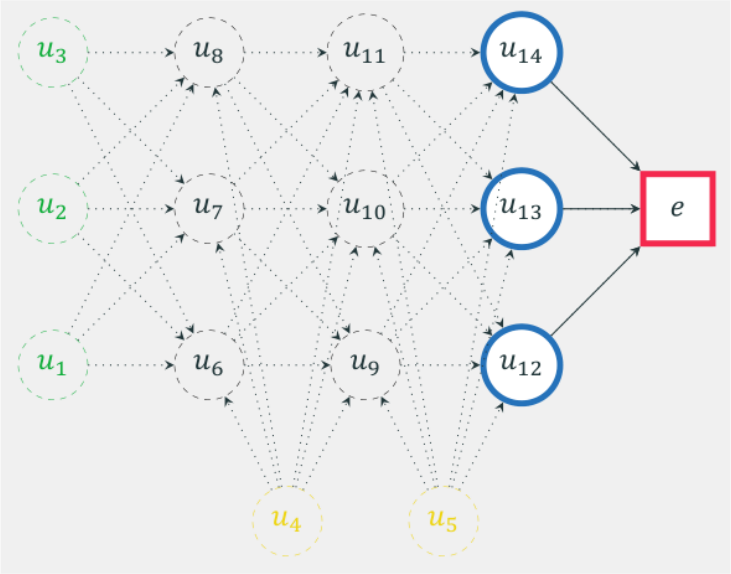
assume that $u(x, y) = x^2 + y^2$, $y(r, t) = r\sin(t)$ and $x(r, t) = r\cos(t)$,

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

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.

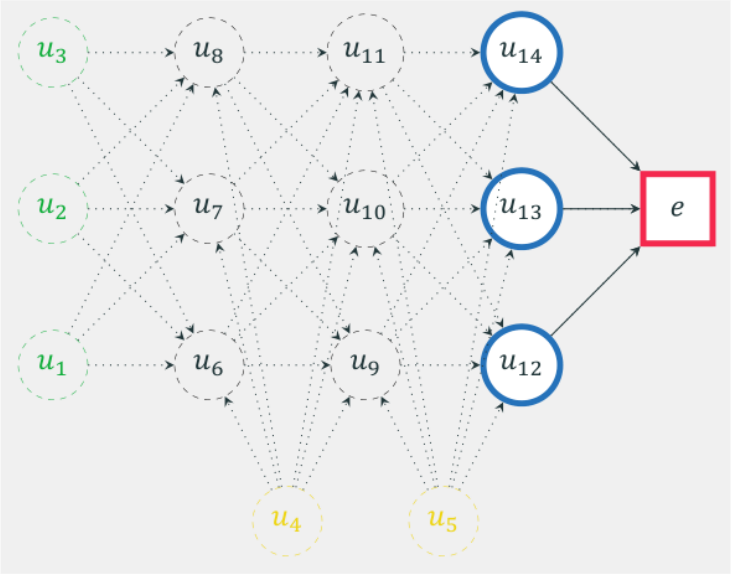


After the forward pass, we have evaluated all units u and finished with the loss e.



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}.

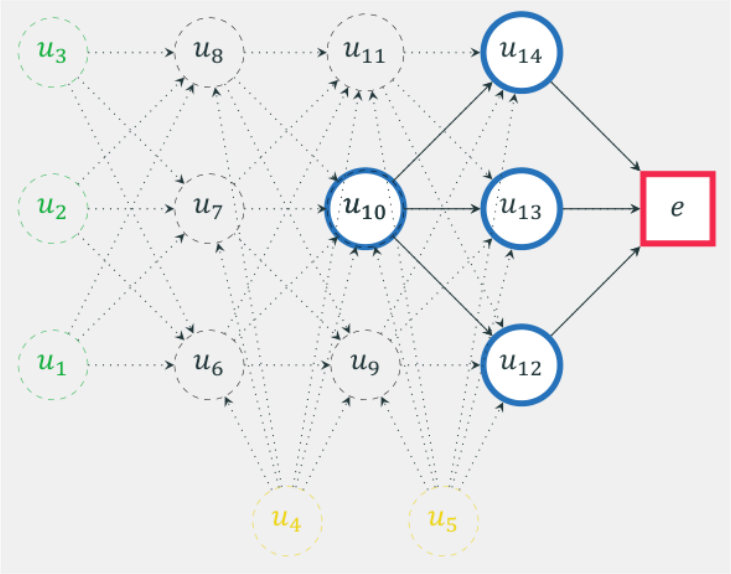


For instance if

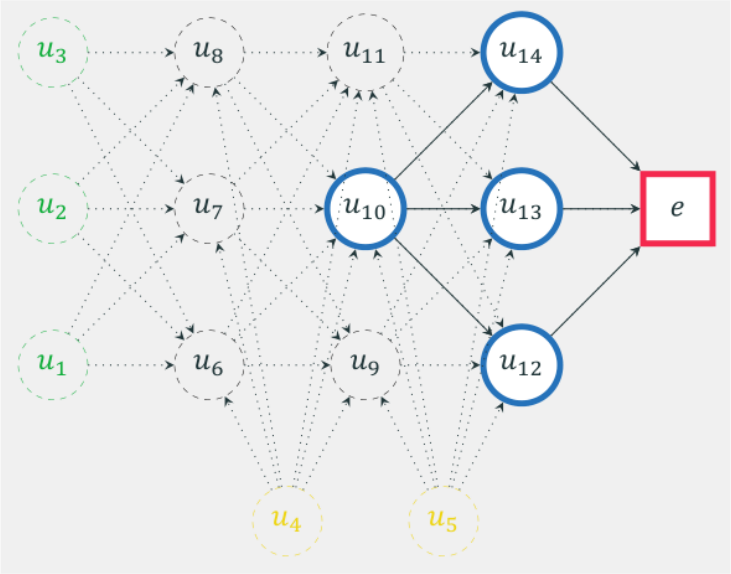
$$e(u_{12}, u_{13}, u_{14}) = (u_{12} - a)^2 + (u_{13} - b)^2 + (u_{14} - c)^2$$

Then

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



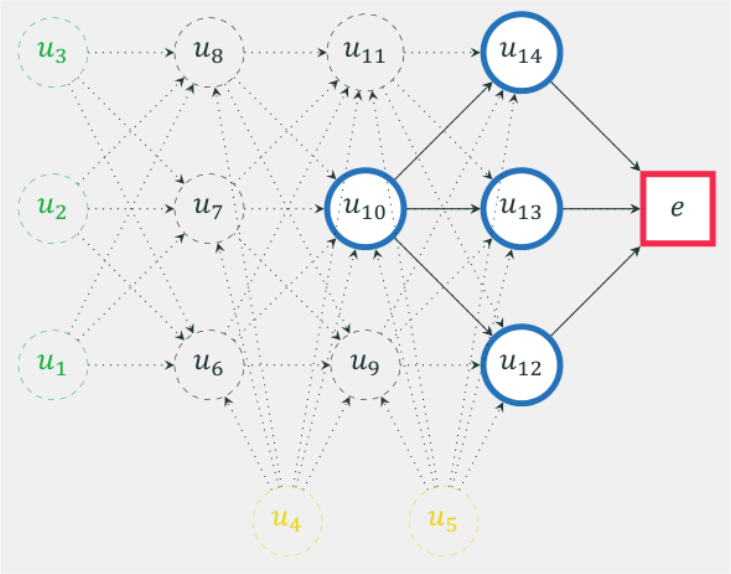
Now that we have computed $\frac{\partial e}{\partial u_{14}}$, $\frac{\partial e}{\partial u_{13}}$ and $\frac{\partial e}{\partial u_{12}}$, how do we compute $\frac{\partial e}{\partial u_{10}}$?



We can use the chain-rule:

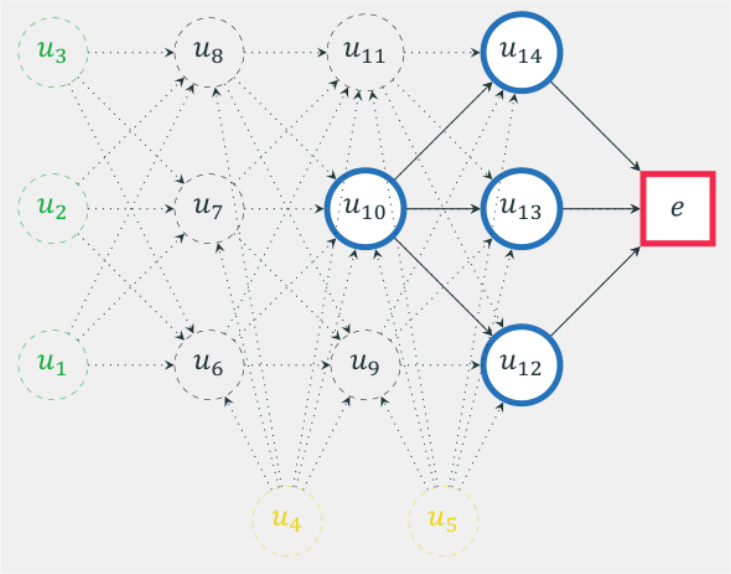
$$\frac{\partial e}{\partial u_i} = \sum_{j \in \text{Outputs}(i)} \frac{\partial u_j}{\partial u_i} \frac{\partial e}{\partial u_j}$$

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:

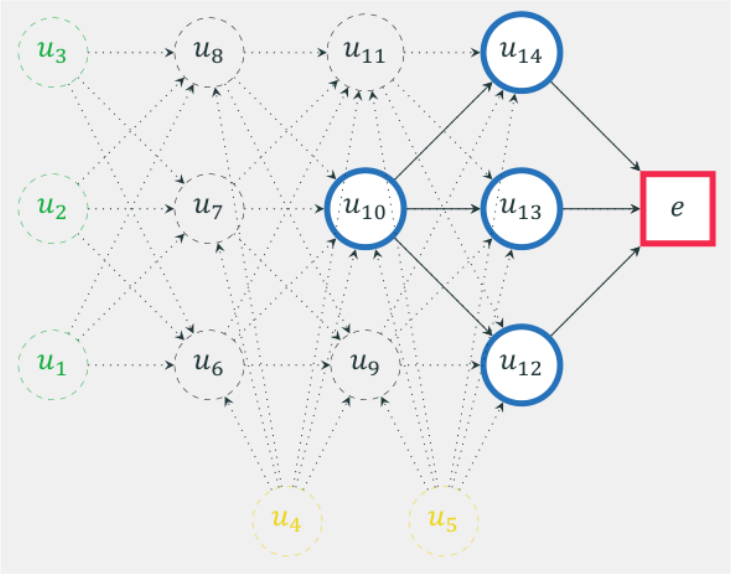
$$\frac{\partial e}{\partial u_{10}} = \frac{\partial u_{14}}{\partial u_{10}}\frac{\partial e}{\partial u_{14}} + \frac{\partial u_{13}}{\partial u_{10}}\frac{\partial e}{\partial u_{13}} + \frac{\partial u_{12}}{\partial u_{10}}\frac{\partial e}{\partial u_{12}}$$



So the chain-rule tells us that:

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

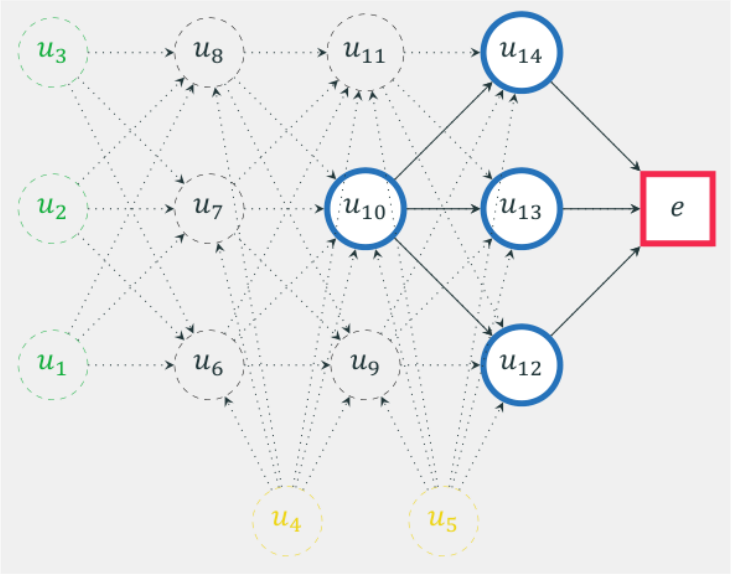
$\frac{\partial e}{\partial u_{12}}$, $\frac{\partial e}{\partial u_{13}}$ and $\frac{\partial e}{\partial u_{14}}$ have already been computed.



So the chain-rule tells us that:

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

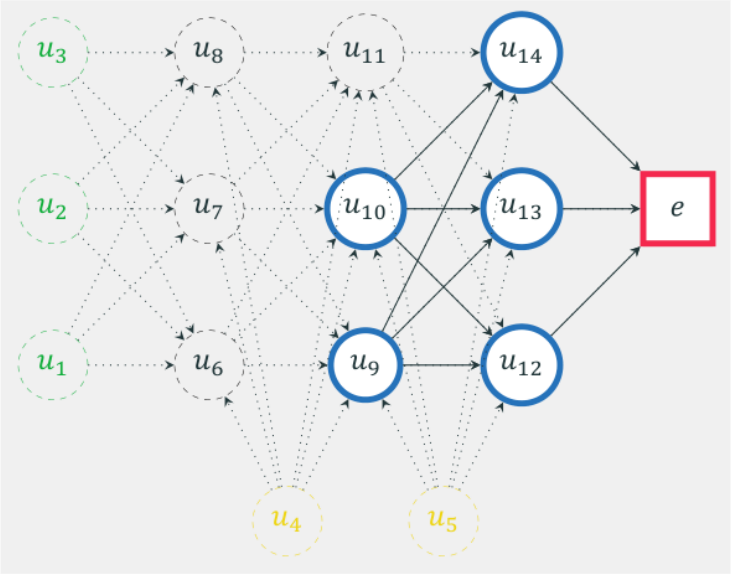
and $\frac{\partial u_{14}}{\partial u_{10}}$, $\frac{\partial u_{13}}{\partial u_{10}}$ and $\frac{\partial u_{12}}{\partial u_{10}}$ can also be derived from the definitions of the functions u_{12}, u_{13} and u_{14}.



So the chain-rule tells us that:

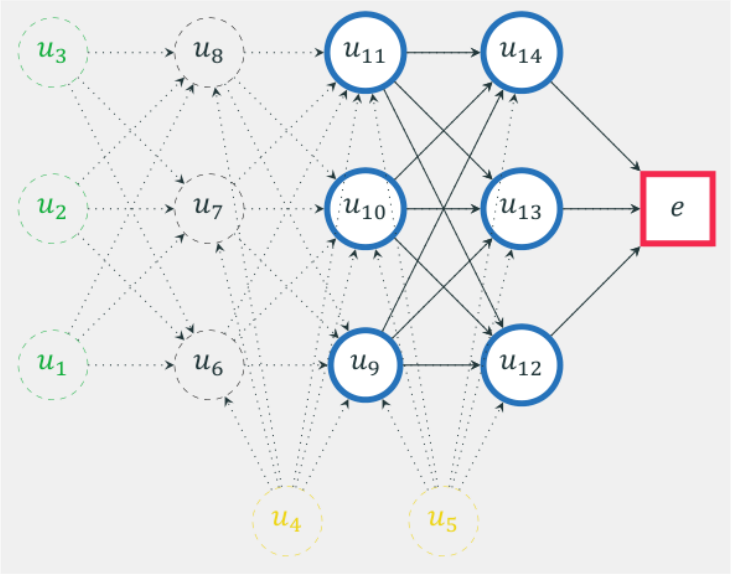
$$\frac{\partial e}{\partial u_{10}} = \frac{\partial u_{14}}{\partial u_{10}}\frac{\partial e}{\partial u_{14}} + \frac{\partial u_{13}}{\partial u_{10}}\frac{\partial e}{\partial u_{13}} + \frac{\partial u_{12}}{\partial u_{10}}\frac{\partial e}{\partial u_{12}}$$

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



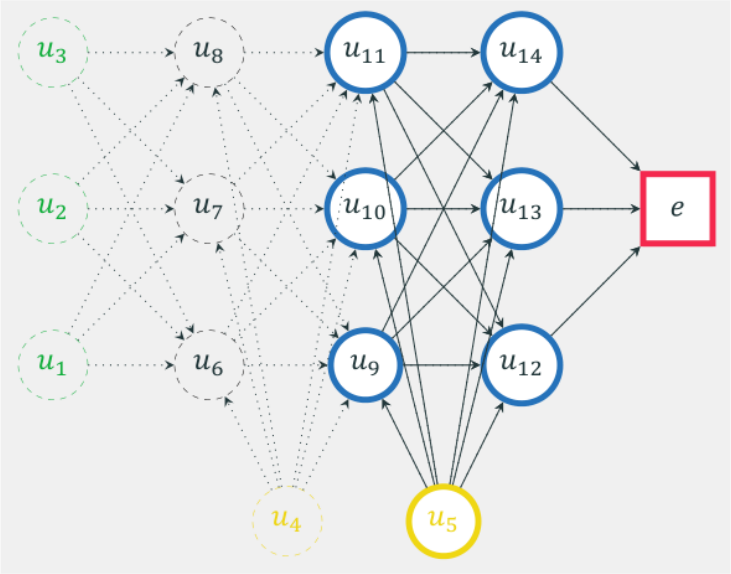
We can now propagate back the computations and derive the gradient for each node at a time.

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



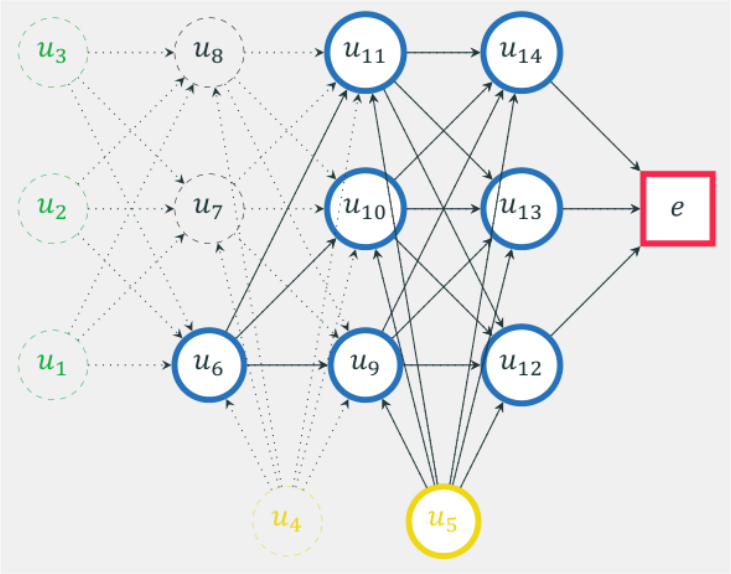
We can now propagate back the computations and derive the gradient for each node at a time. We only visit each node once!

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



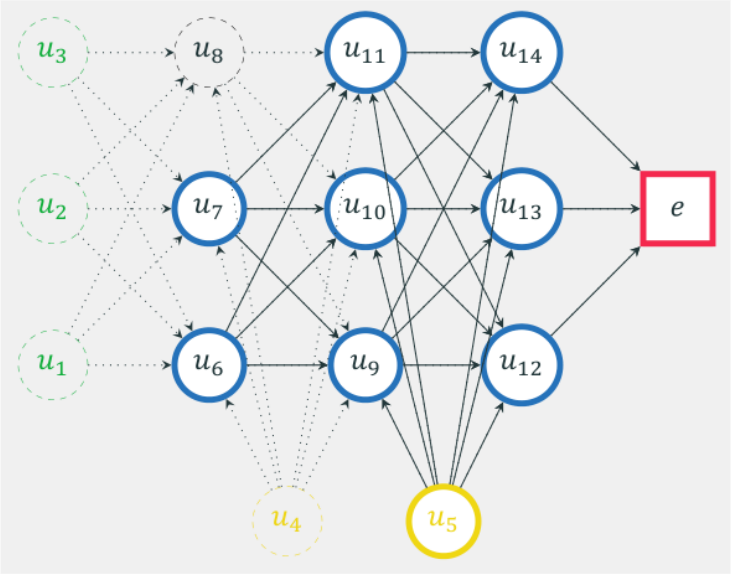
We can now propagate back the computations and derive the gradient for each node at a time.

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



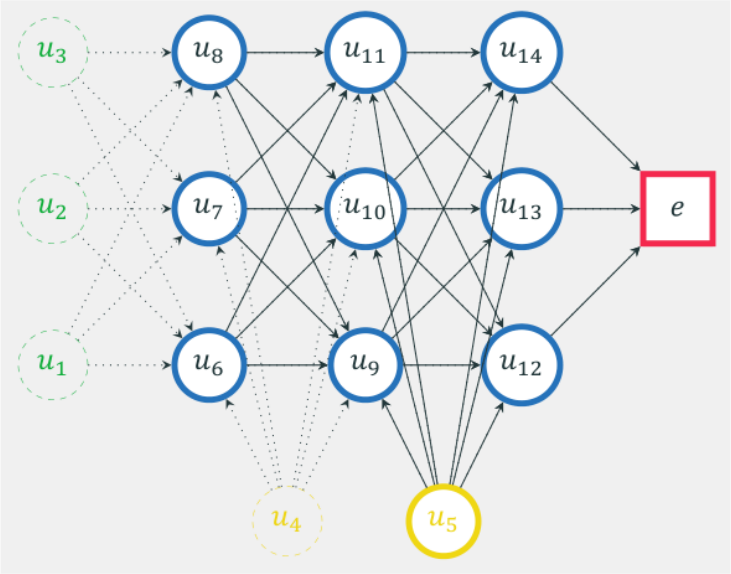
We can now propagate back the computations and derive the gradient for each node at a time.

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



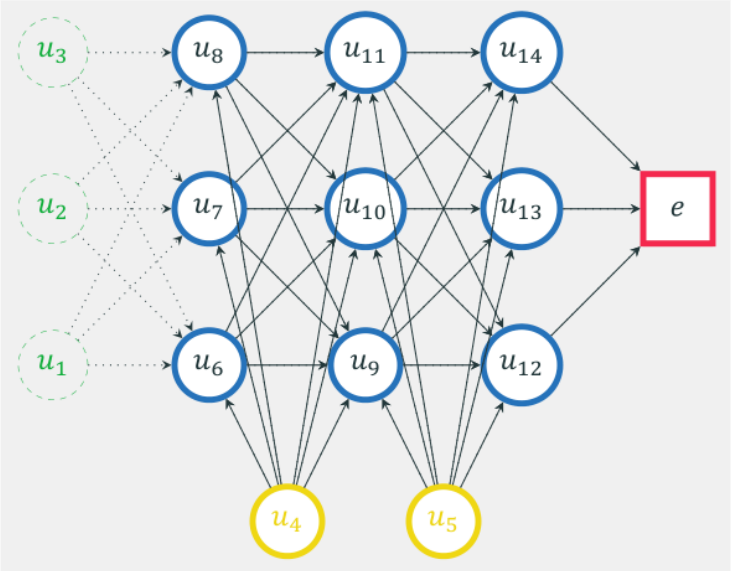
We can now propagate back the computations and derive the gradient for each node at a time.

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



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.