CSC4007 Advanced Machine Learning

Lesson 08: Backpropagation

by Vien Ngo
EEECS / ECIT / DSSC

Some slides adapted from Barry Devereux, Jesus Martinez-del-Rincon, Geoffrey Hinton, Andrew Ng

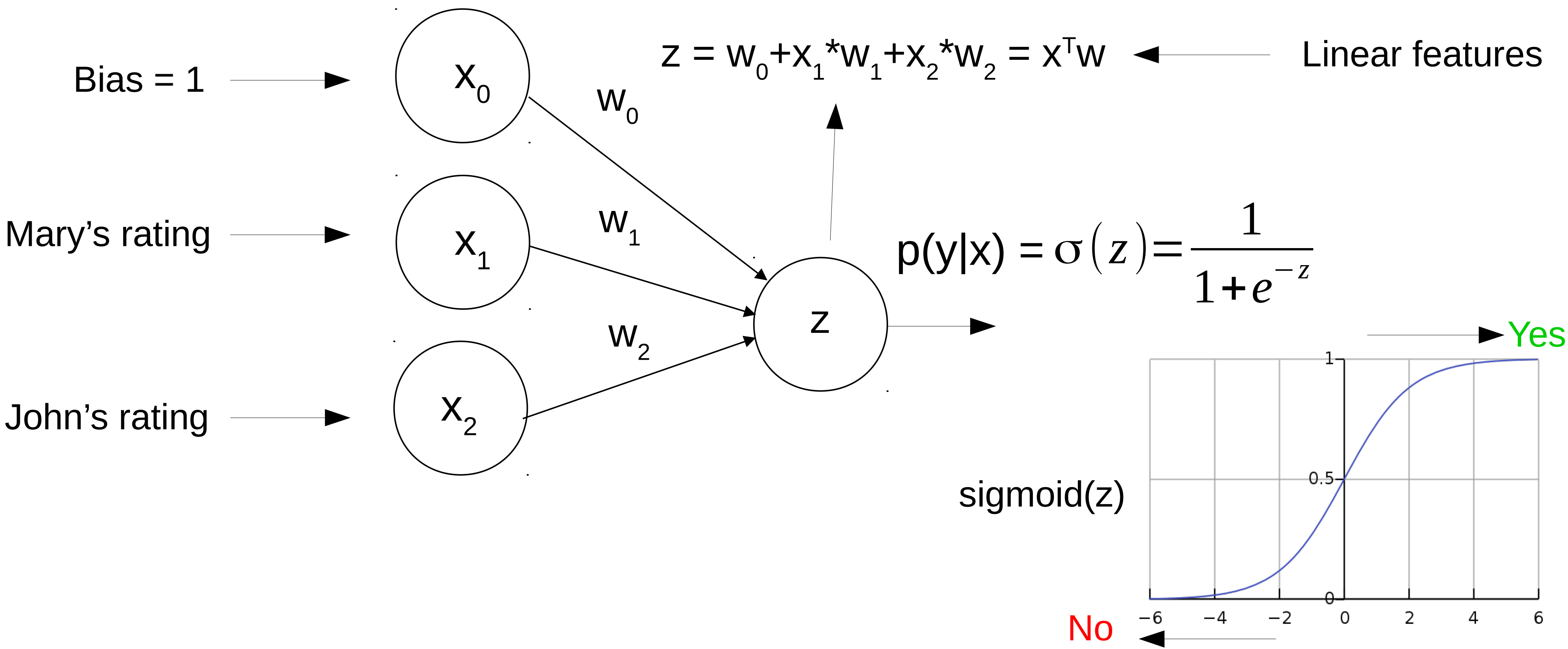
Outline

- Neural network basics and representation
- Perceptron learning, multi-layer perceptron
- **Neural network training: Backpropagation**
- Modern neural network architecture (a.k.a Deep learning):
  - Convolutional neural network (CNN)
  - Recurrent neural network (RNN), long-short term memory network (LSTM)

Neural Networks: Learning

- **NNs for classification**: e.g. logistic regression (for binary classification) = 1 layer NN with sigmoid activations at output

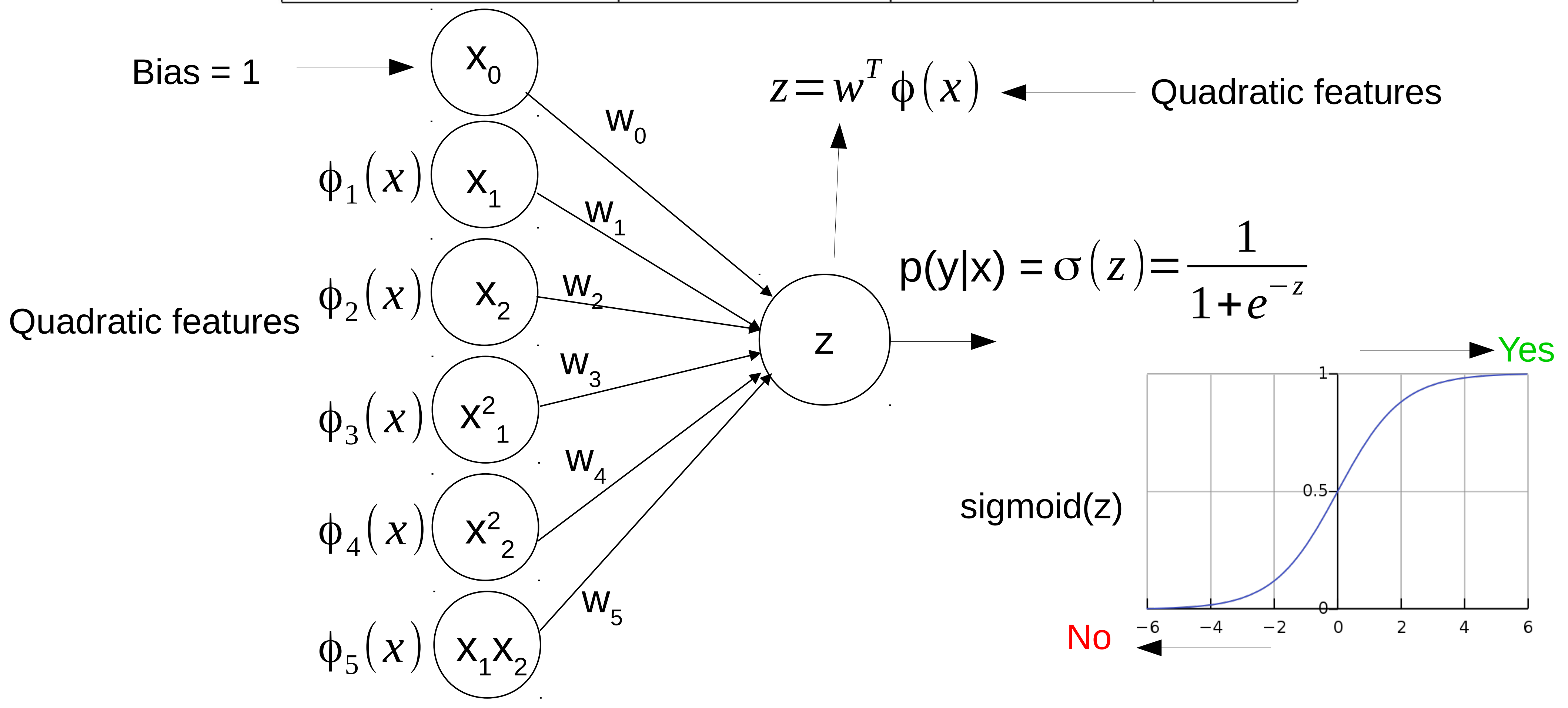
Movie name	Mary's rating	John's rating	I like?
Lord of the Rings II	1	5	No
...	...	...	...
Star Wars I	4.5	4	Yes
Gravity	3	3	?



Neural Networks: Learning

- **NNs for classification**: e.g. logistic regression (for binary classification) = 1 layer NN with sigmoid activations

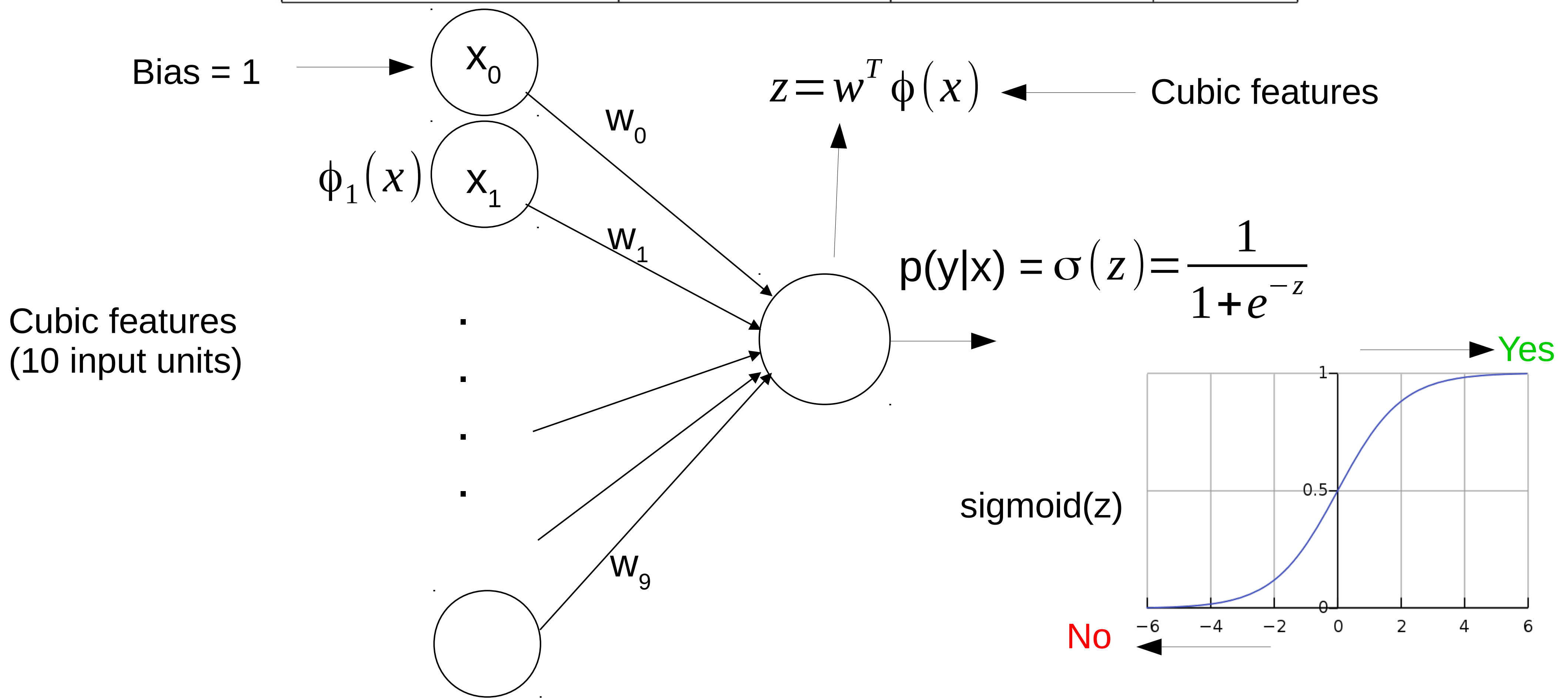
Movie name	Mary's rating	John's rating	I like?
Lord of the Rings II	1	5	No
...	...	...	...
Star Wars I	4.5	4	Yes
Gravity	3	3	?



Neural Networks: Learning

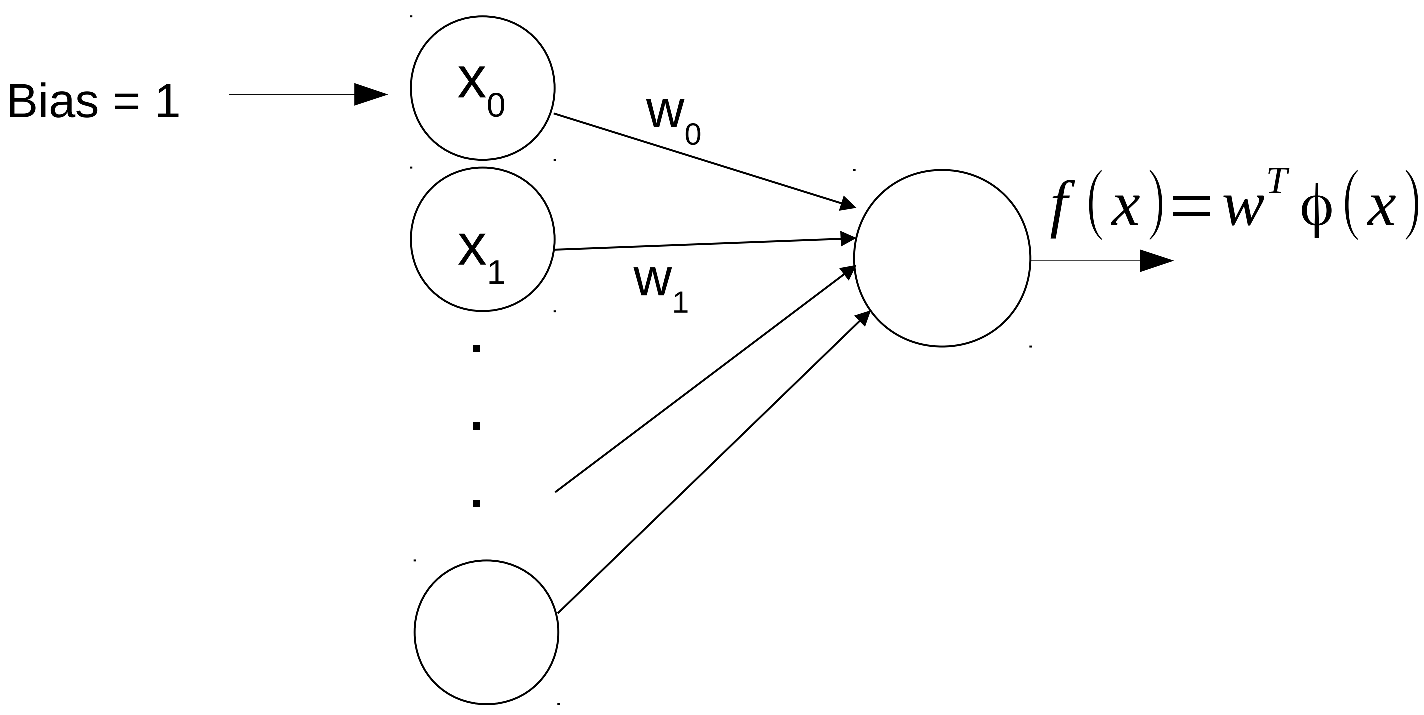
- NNs for classification: e.g. logistic regression (for binary classification) = 1 layer NN with sigmoid activations

Movie name	Mary's rating	John's rating	I like?
Lord of the Rings II	1	5	No
...	...	...	...
Star Wars I	4.5	4	Yes
Gravity	3	3	?



Neural Networks: Learning

- **NNs for regression**: e.g. 1-layer NN without activations to compute continuous-valued prediction.
  - In house price data: if there are d=100 input variables
    - Quadratic features: ~ 5000 features, number of features grows O(d^2)
    - Cubic feature: ~ 170 000 features, number of features grows O(d^3)
    - Not a good way to build classifiers when d is large



Neural Networks: Learning

- In house price data: if there are d=100 input variables
    - Quadratic features: ~ 5000 features, number of features grows $O(d^2)$
    - Cubic feature: ~ 170 000 features, number of features grows $O(d^3)$
    - Not a good way to build classifiers when d is large
- Image data:
    - If we used 50 x 50 pixels --> 2500 pixels, so d = 2500
    - If RGB images then d=7500
- Too big - wayyy too big

Learning with hidden units

- Networks without hidden units (e.g perceptrons) are very limited in the input-output mappings they can model.

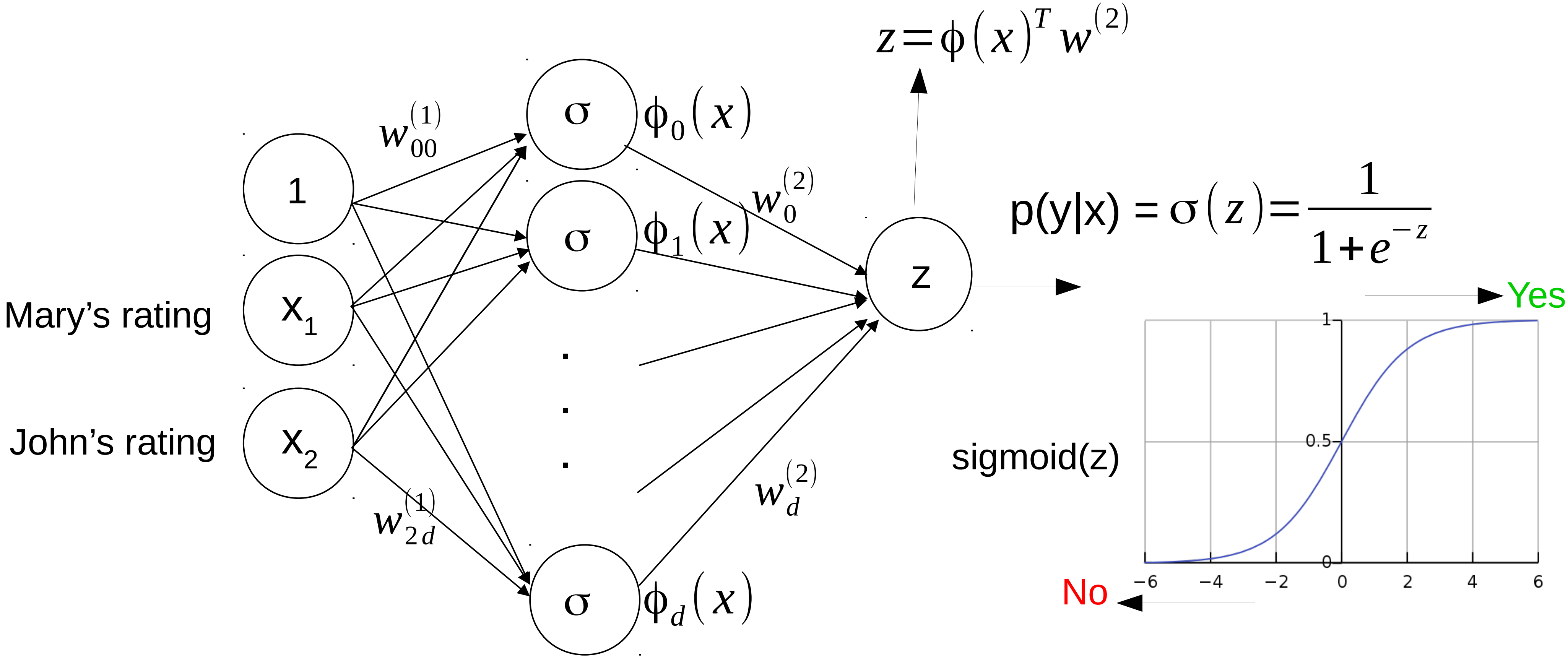
- Adding a layer of hand-coded features (as in a perceptrons for spam email classification using bag of word features) makes them much more powerful but the hard bit is designing the features.
  - We would like to find good features without requiring insights into the task or repeated trial and error where we guess some features and see how well they work.

- **We need to automate the loop of designing features for a particular task and seeing how well they work.**

Neural Networks: Learning

- **NNs for classification**: e.g. with multi-layered NN using sigmoid activations at outputs, certain activations at hidden nodes

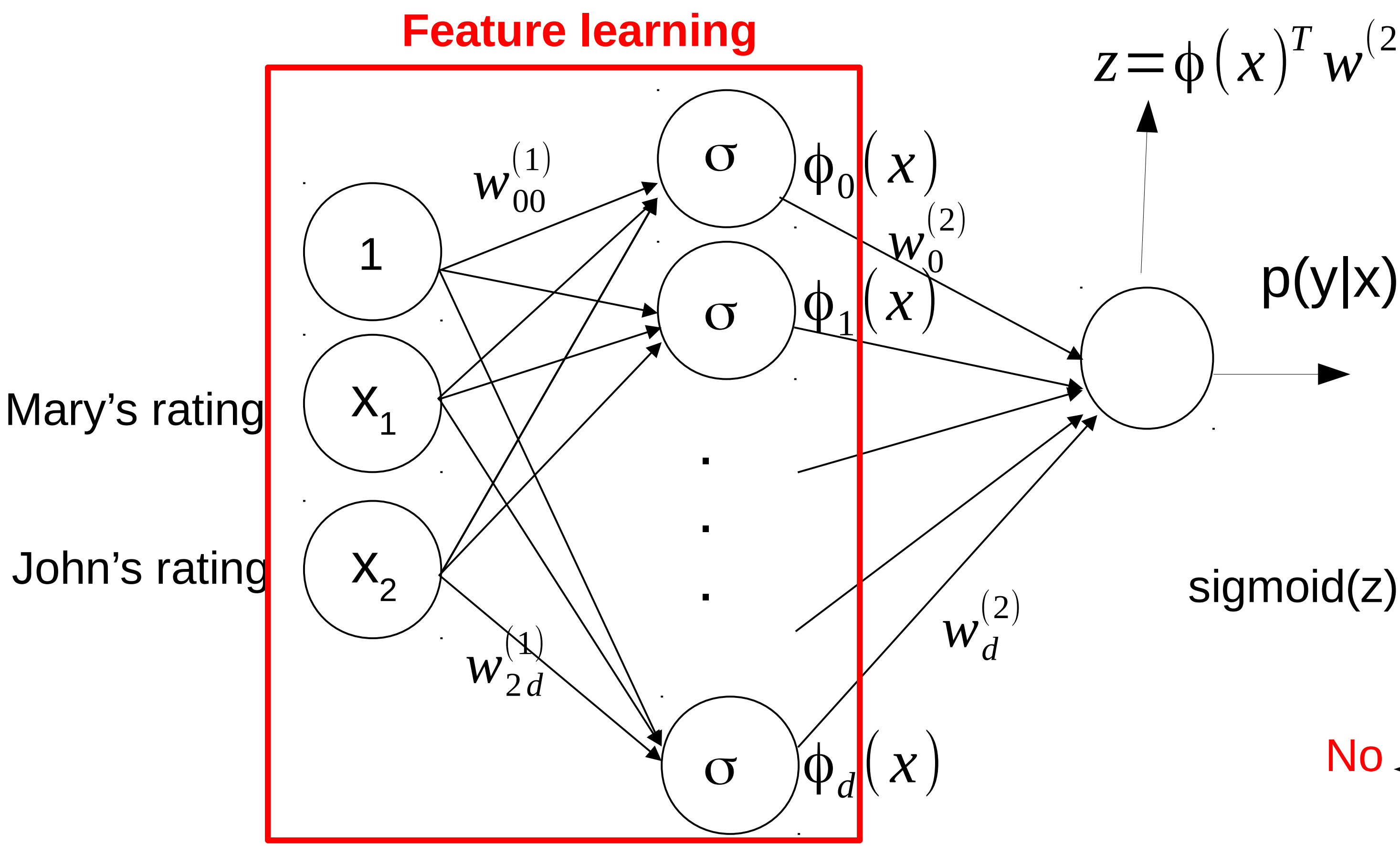
Movie name	Mary's rating	John's rating	I like?
Lord of the Rings II	1	5	No
...	...	...	...
Star Wars I	4.5	4	Yes
Gravity	3	3	?



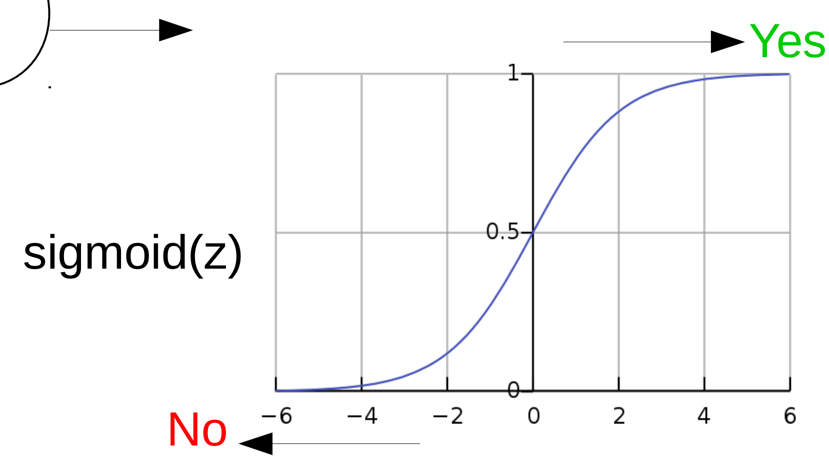
Neural Networks: Learning

- NNs for classification: e.g. with multi-layered NN using sigmoid activations at outputs, other activations at hidden nodes

Movie name	Mary's rating	John's rating	I like?
Lord of the Rings II	1	5	No
...	...	...	...
Star Wars I	4.5	4	Yes
Gravity	3	3	?



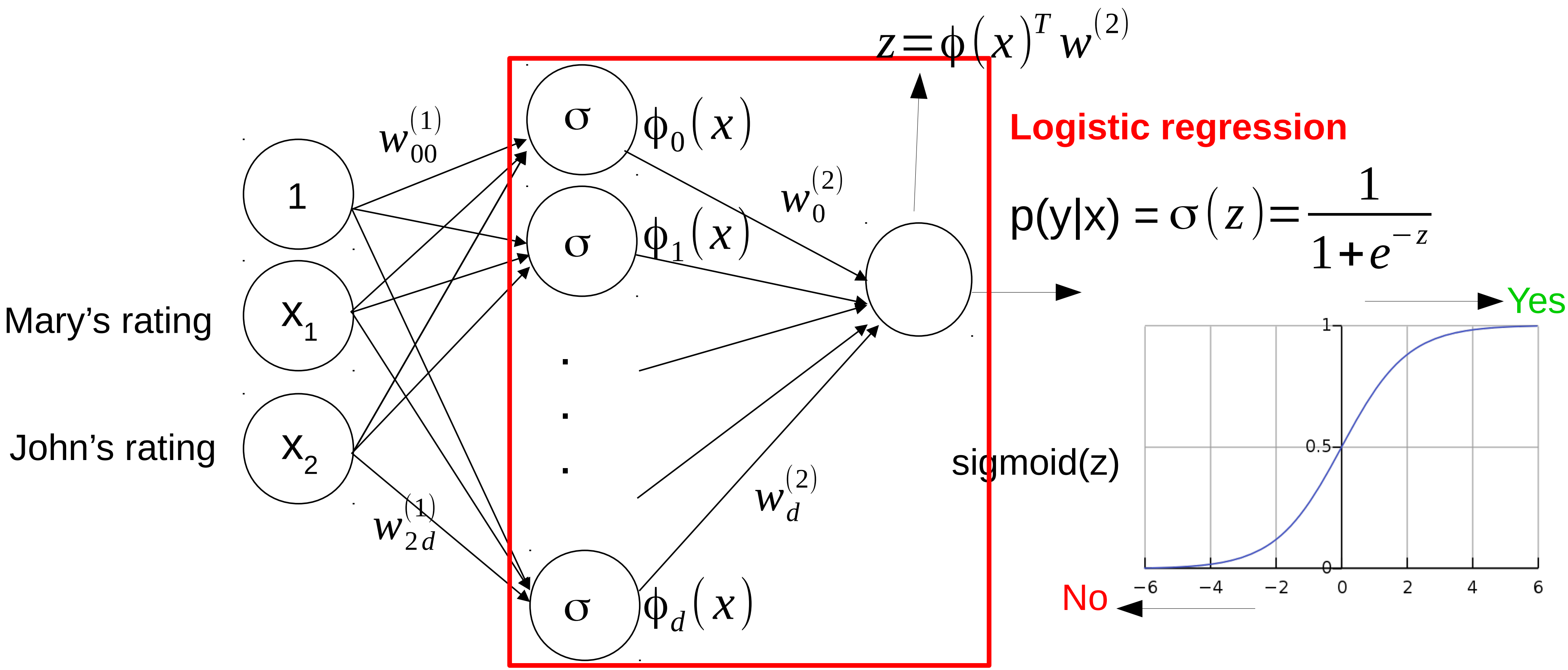
$$p(y|x) = \sigma(z) = \frac{1}{1+e^{-z}}$$



Neural Networks: Learning

- NNs for classification: e.g. with multi-layered NN using sigmoid activations at outputs, other activations at hidden nodes

Movie name	Mary's rating	John's rating	I like?
Lord of the Rings II	1	5	No
...	...	...	...
Star Wars I	4.5	4	Yes
Gravity	3	3	?

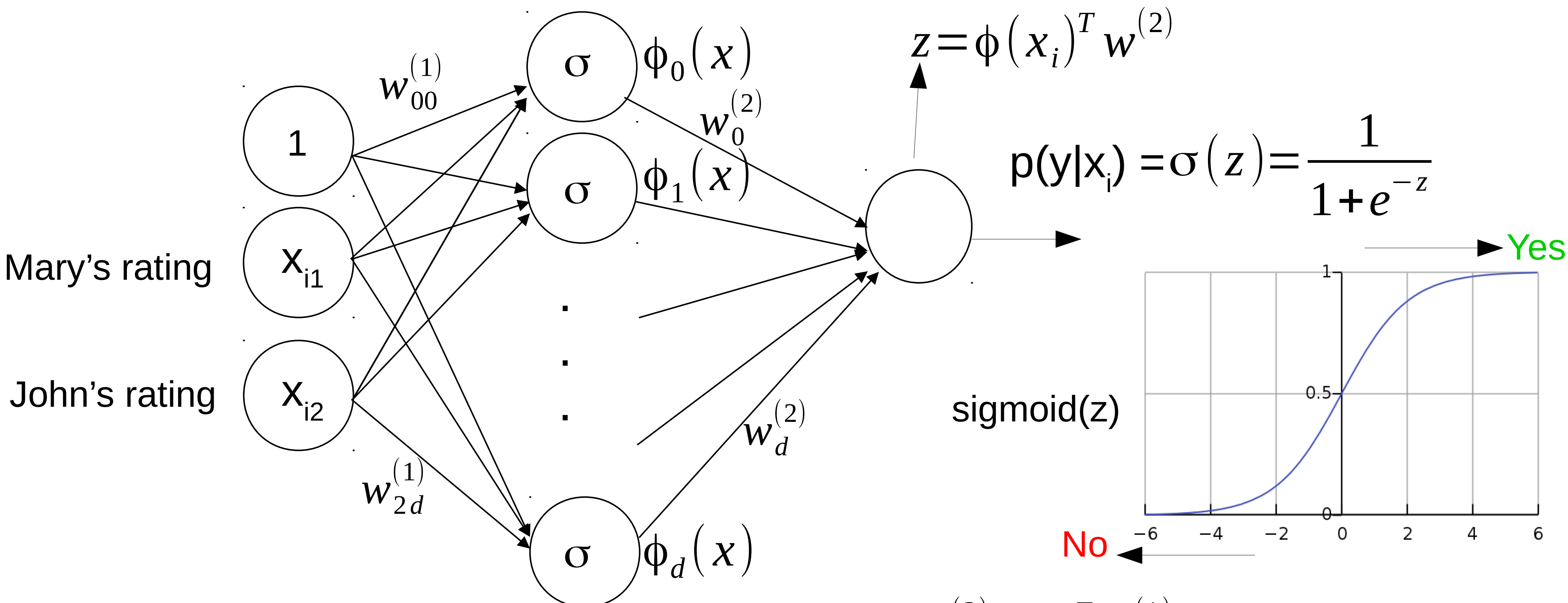


Learning with hidden units

- **We need to automate the loop of designing features for a particular task and seeing how well they work.**
  - Neural Network learning = Automatic feature and task learning

Neural Networks learning: Cost function

- **NNs for classification**: e.g. with multi-layered NN using sigmoid activations at outputs, other activations at hidden nodes



Forward pass: $p(y_i|x_i) = \sigma(w^{(2)} \underbrace{\sigma(x_i^T w^{(1)})}_{\text{Layer 1/hidden layer}})$

– Negative log likelihood

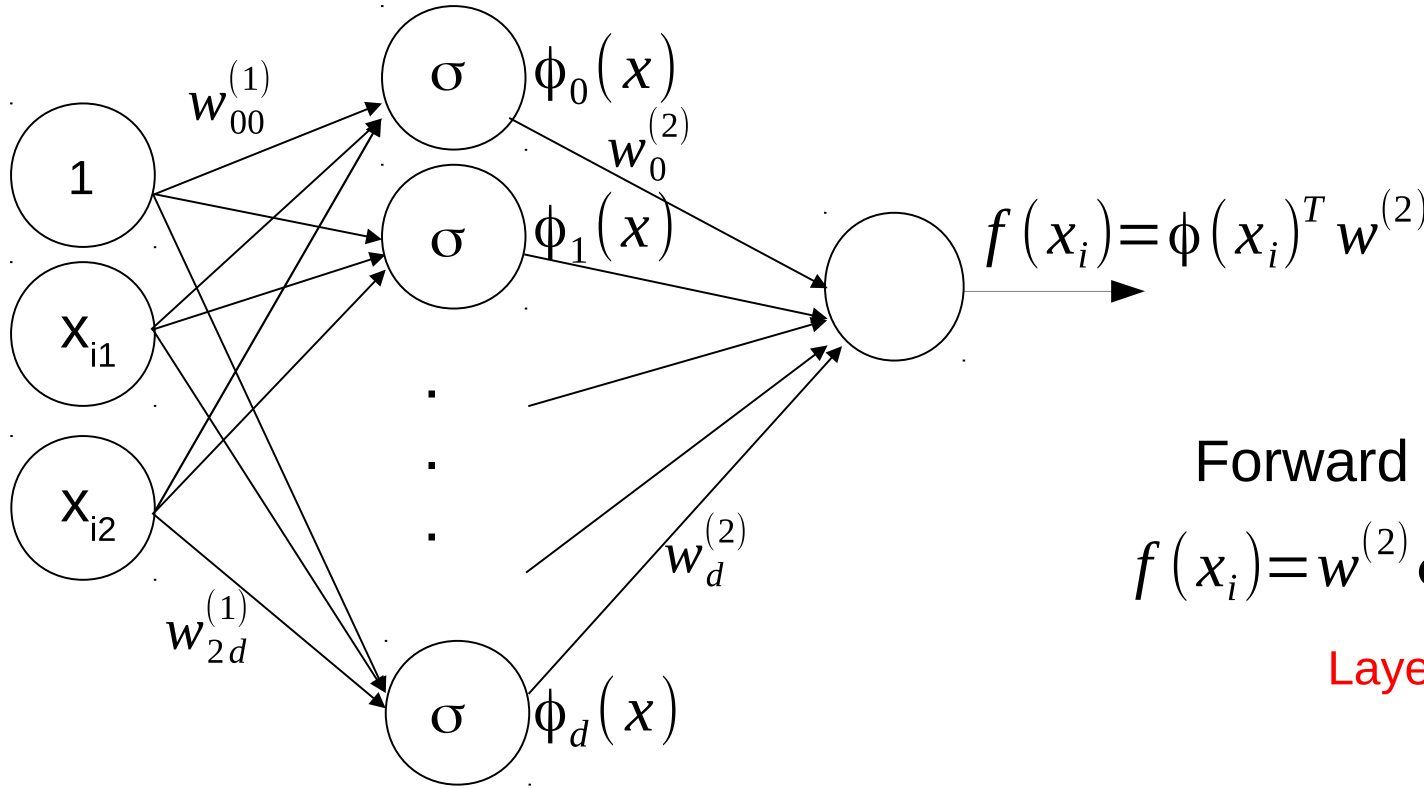
$$E = -\sum_{i=1}^{n} [y_i \log p(y_i|x_i) + (1-y_i) \log(1-p(y_i|x_i))] + \lambda \sum_k \sum_{ij} w_{ij}^{(k)2}$$

Regularization

Where training set is $\{(x_1,y_1), (x_2,y_2), \ldots, (x_n,y_n)\}$

Neural Networks learning: Cost function

- **NNs for regression**: e.g. with multi-layered NN without sigmoid activations at outputs, other activations at hidden nodes



Forward pass:

$$f(x_i) = w^{(2)} \underbrace{\sigma(x_i^T w^{(1)})}_{\text{Layer 1/hidden layer}}$$

- Mean square error (MSE)

$$E = \sum_{i=1}^{n} [y_i - f(x_i)]^2 + \lambda \sum_k \sum_{ij} w_{ij}^{(k)2}$$

Regularization

Where training set is $\{(x_1,y_1), (x_2,y_2), \ldots, (x_n,y_n)\}$

Neural Networks learning: Cost function

- We've already described forward propagation
  - This is the algorithm which takes your neural network and the initial input into that network and pushes the input through the network
  - It leads to the generation of an output hypothesis, which may be a single real number (e.g. binary classification), but can also be a vector (e.g. multi-class classification).

e.g. $p(y_i|x_i) = \sigma(w^{(2)} \sigma(x^T w^{(1)}))$

- The **cost function** is defined on the output from forward propagation, e.g $\bar{y}$=f(x) and the target (expected output y in the training set), for example
  - Negative log likelihood
  - Mean square error, mean absolute error, etc.

- **NN Learning**: Adjusting the weights to minimize the cost function (in order to make sense the inputs and outputs).

Adjusting weights

One simple approach: (Very inefficient)

1. Pick one of the training items at random and present it to the network.

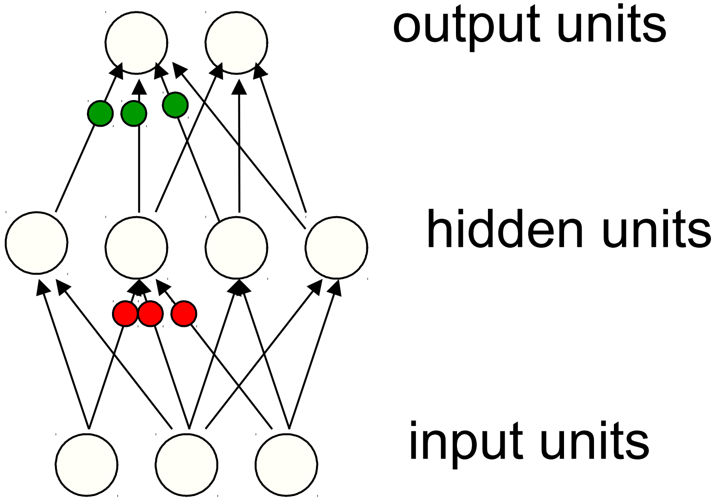
2. Measure the error (difference between observed and expected output)

3. Pick one of the connections at random. Change the value of that connection's weight slightly and get the new output

4. If that change improves the output, then update the weight to the new value. Otherwise, keep the original value

5. Repeat from step 3 a great many times

6. Repeat from step 1 a great many times



Like a random mutation in biology

Survival of the fittest

This is also a form of reinforcement learning

The idea behind backpropagation

- We don’t know what the hidden units ought to do, but we can compute how fast the error changes as we change a hidden activity.
  - Instead of using desired activities to train the hidden units, use error derivatives w.r.t. hidden activities → **gradient descent**
  - Each hidden activity can affect many output units and can therefore have many separate effects on the error. These effects must be combined.
- We can compute error derivatives for all the hidden units efficiently at the same time.
  - Once we have the error derivatives for the hidden activities, its easy to get the error derivatives for the weights going into a hidden unit.→ **updates along the gradient descent direction**

Converting error derivatives into a learning procedure

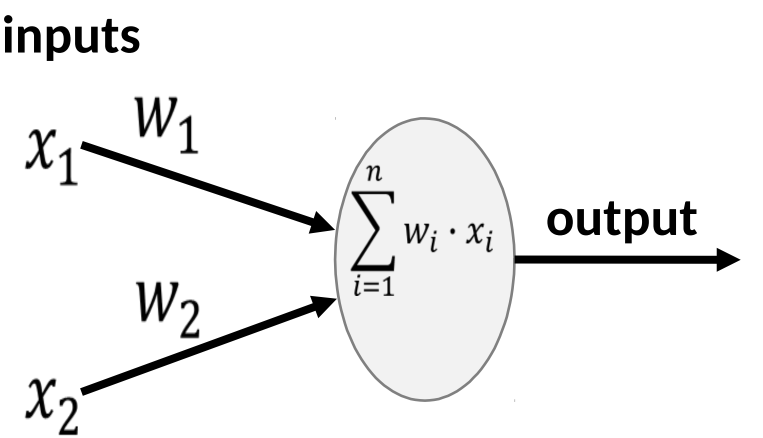
- The backpropagation algorithm is an efficient way of computing the error derivative dE/dw for every weight on a single training case.
- To get a fully specified learning procedure, we still need to make a lot of other decisions about how to use these error derivatives:
  - Optimization issues: How do we use the error derivatives on individual cases to discover a good set of weights?
  - Generalization issues: How do we ensure that the learned weights work well for cases we did not see during training?

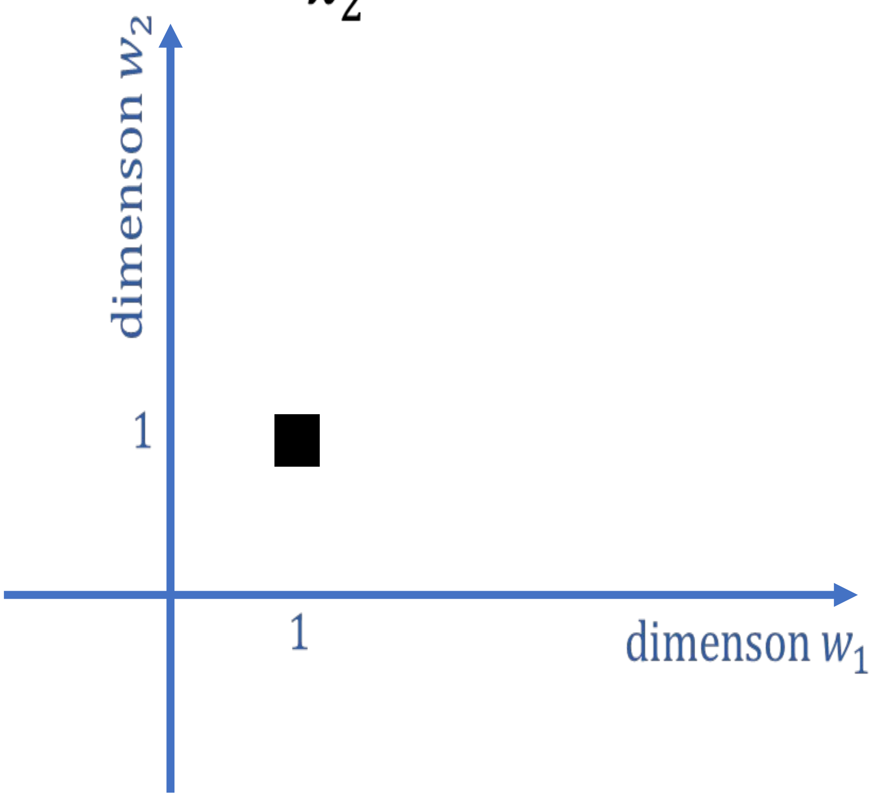
Weight space & gradient descent

Let's take a step back and consider a very simple network with just 2 inputs, and one neuron.

The two weights w_1 and w_2 have particular values, lets say (1,1).

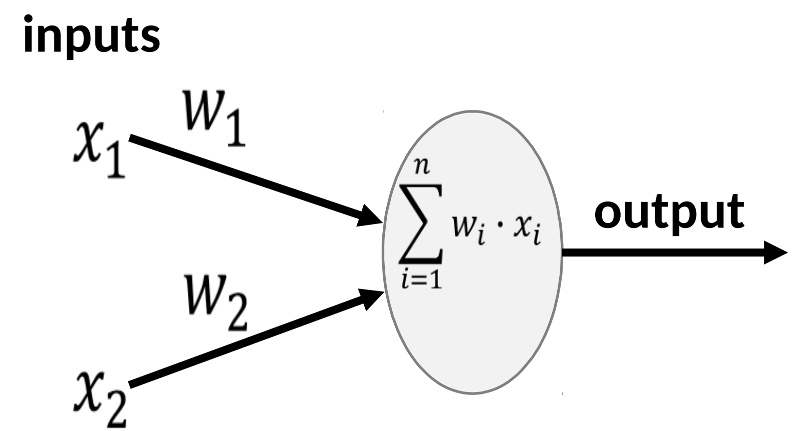
We can visualise the weights as a point in a 2-dimensional plane:





Weight space & gradient descent

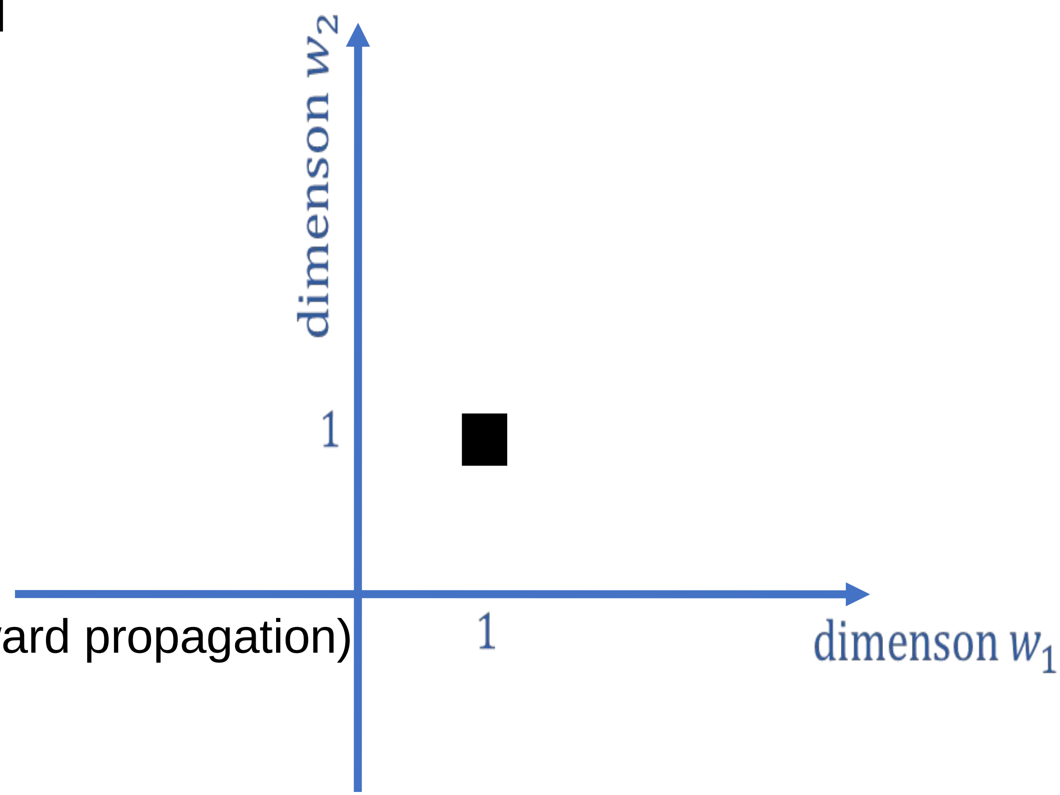
Now, suppose we present a particular item to the network and calculate the error (i.e. the difference between the observed output and the correct output).



Suppose the error is defined as half the square of the difference between the observed/predicted and expected output:

$$E(y, \hat{y}) = \frac{1}{2} \|y - \hat{y}\|^2$$

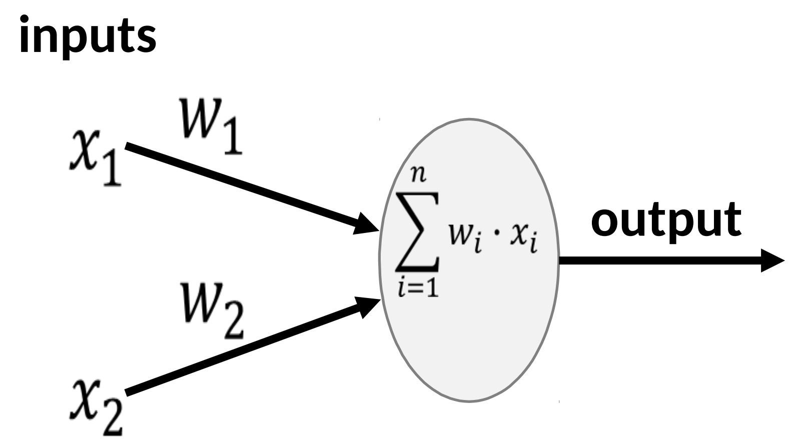
Target/expected predicted (using forward propagation)

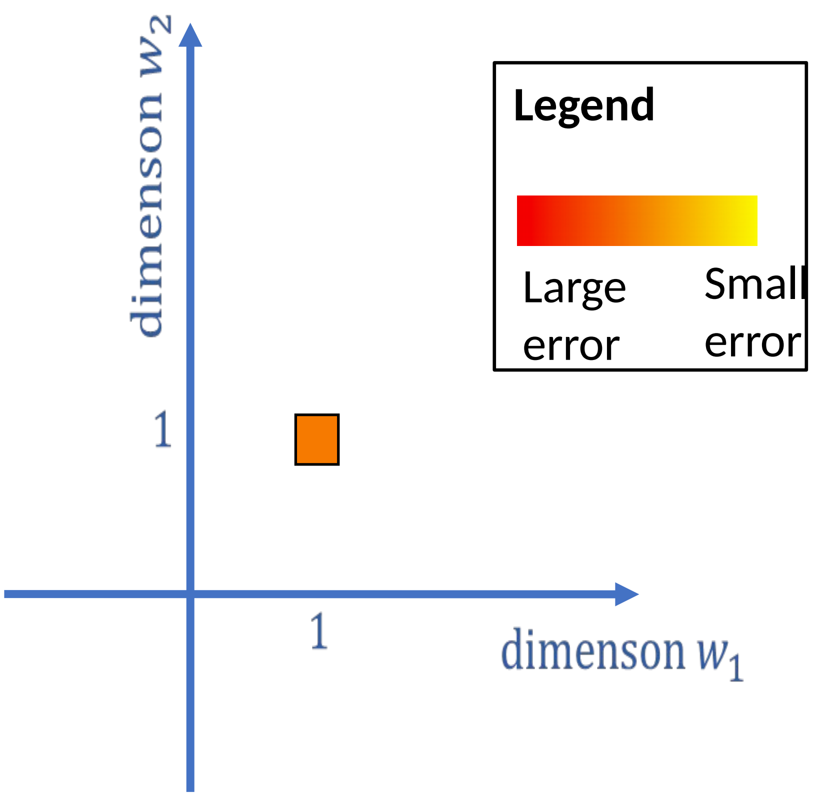


This is a standard choice

Weight space & gradient descent

Let's colour-code the point in weight-space by what the Error is for these weight values





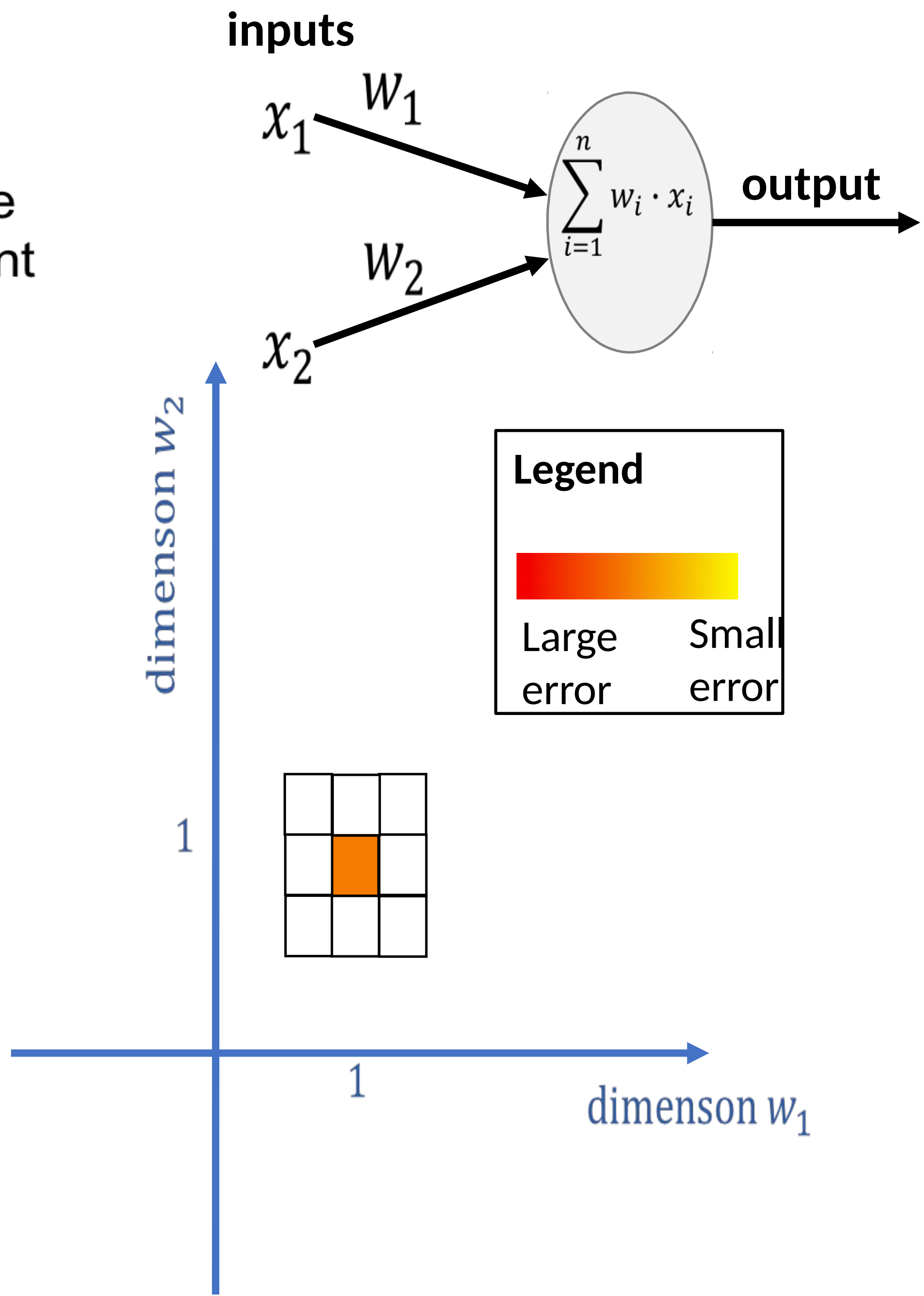
Weight space & gradient descent

Let's next consider points in the feature space that are close to our original point (1,1). E.g.

$$w_1 \in \{0.9, 1.0, 1.1\}$$

$$w_2 \in \{0.9, 1.0, 1.1\}$$

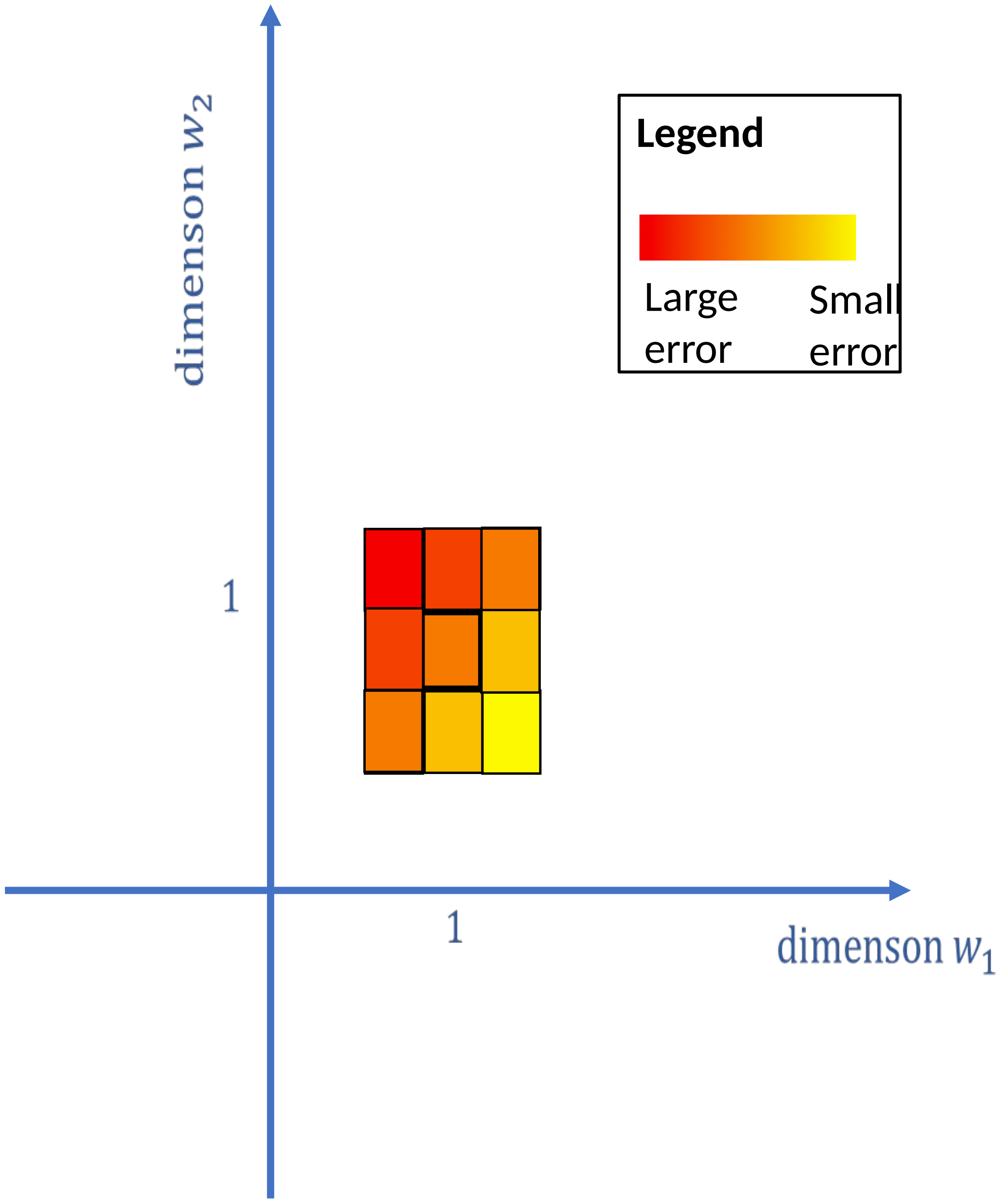
So we are considering 9 weight combinations in total.



Weight space & gradient descent

We can compute the Error for these nearby weights, as before.

Given that we want to decrease the error, what would be a good update to the weights?



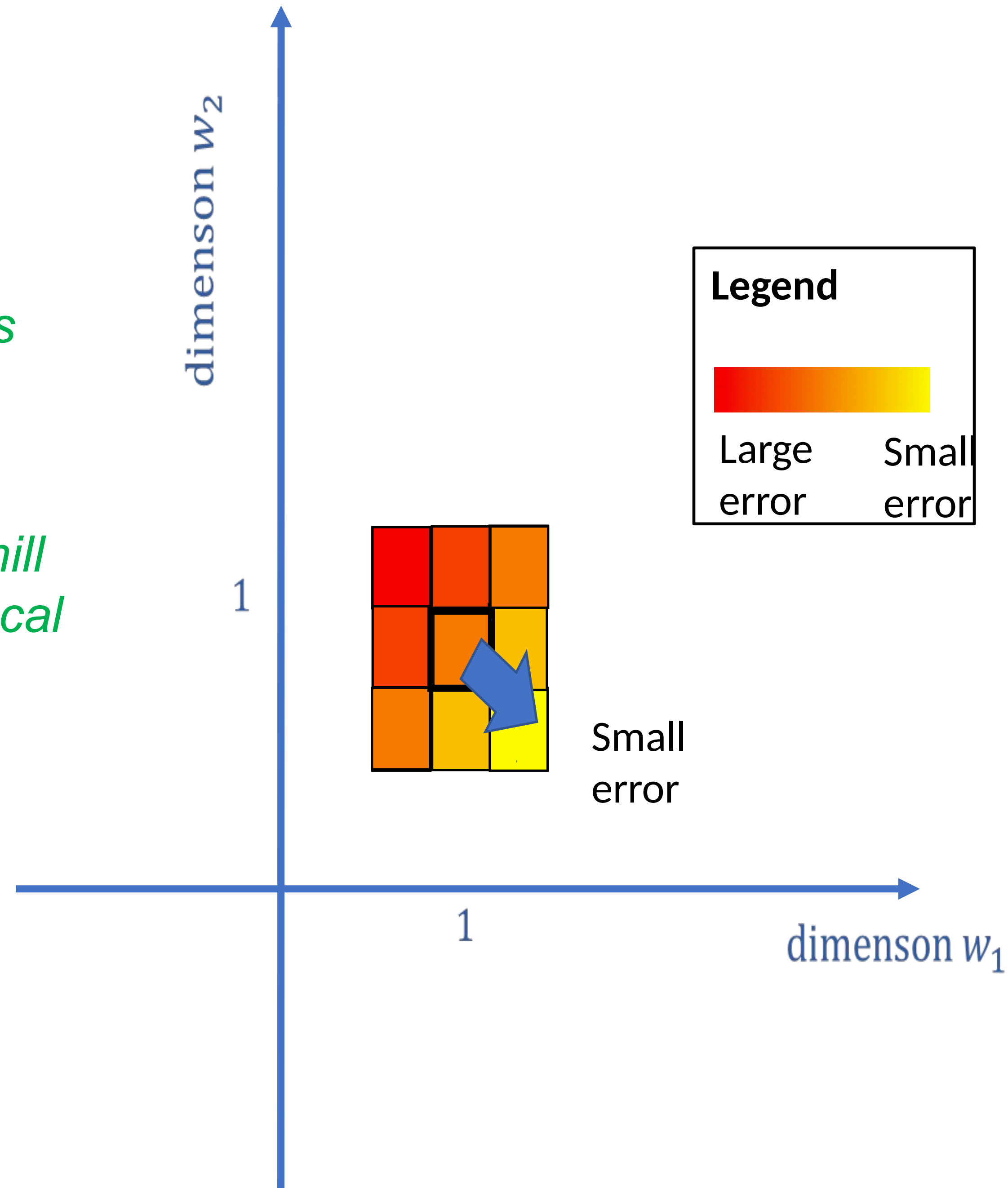
Weight space & gradient descent

We can think of weight-space as a map. Then the Error is like elevation above sea-level.

From a given starting point, we want to get to a place which is as low as possible.

A good strategy is to walk downhill in the steepest direction in the local area.

⇒ "Gradient descent"



Weight space & gradient descent

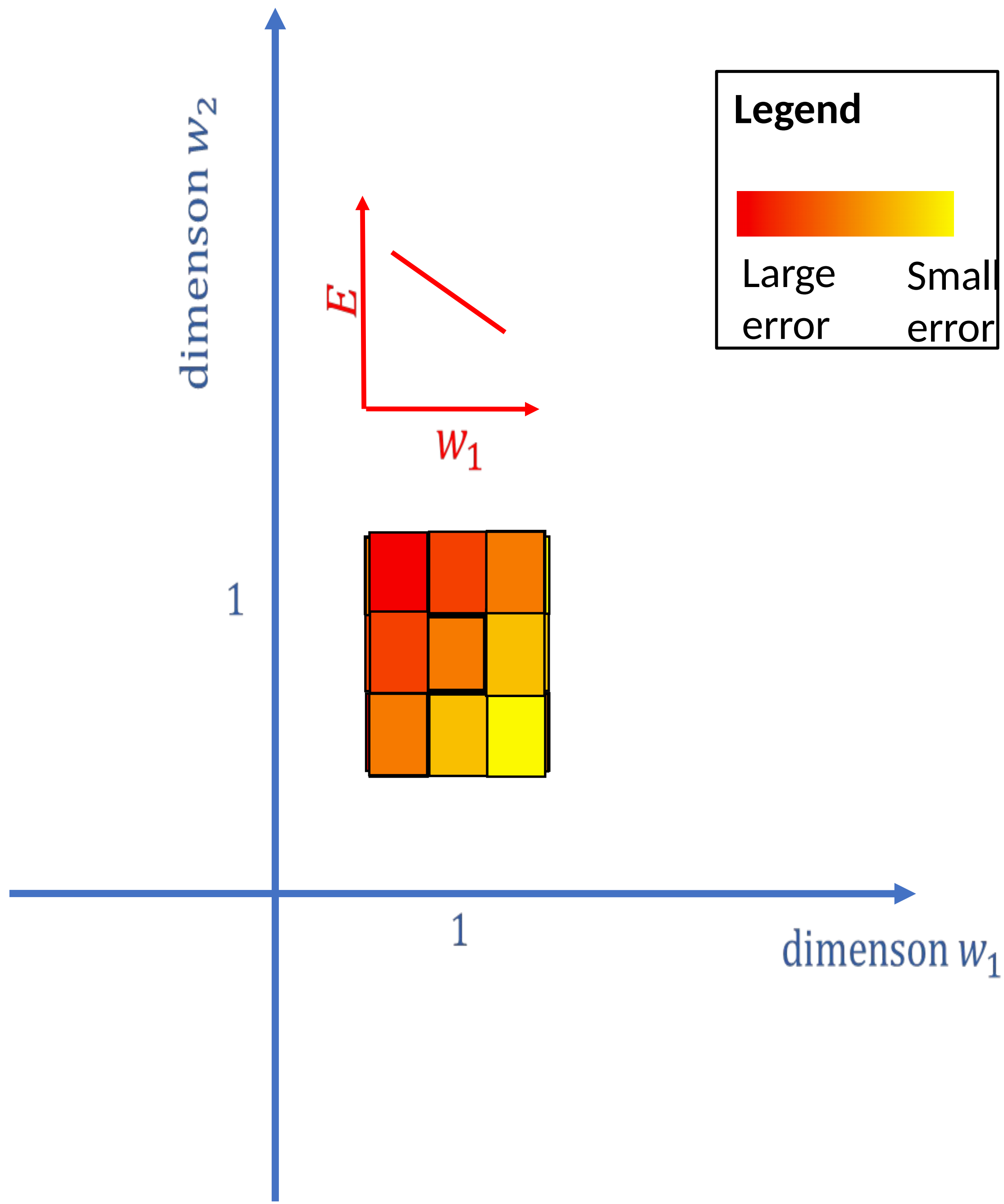
In the local neighbourhood, we need to measure how the Error changes as each weight changes.

i.e. we need to compute the "error gradients"

The change in Error with respect to a small change in w_1 can be written:

$$\frac{\partial E}{\partial w_1}$$

The partial derivative of E with respect to w_1



Weight space & gradient descent

Similarly,

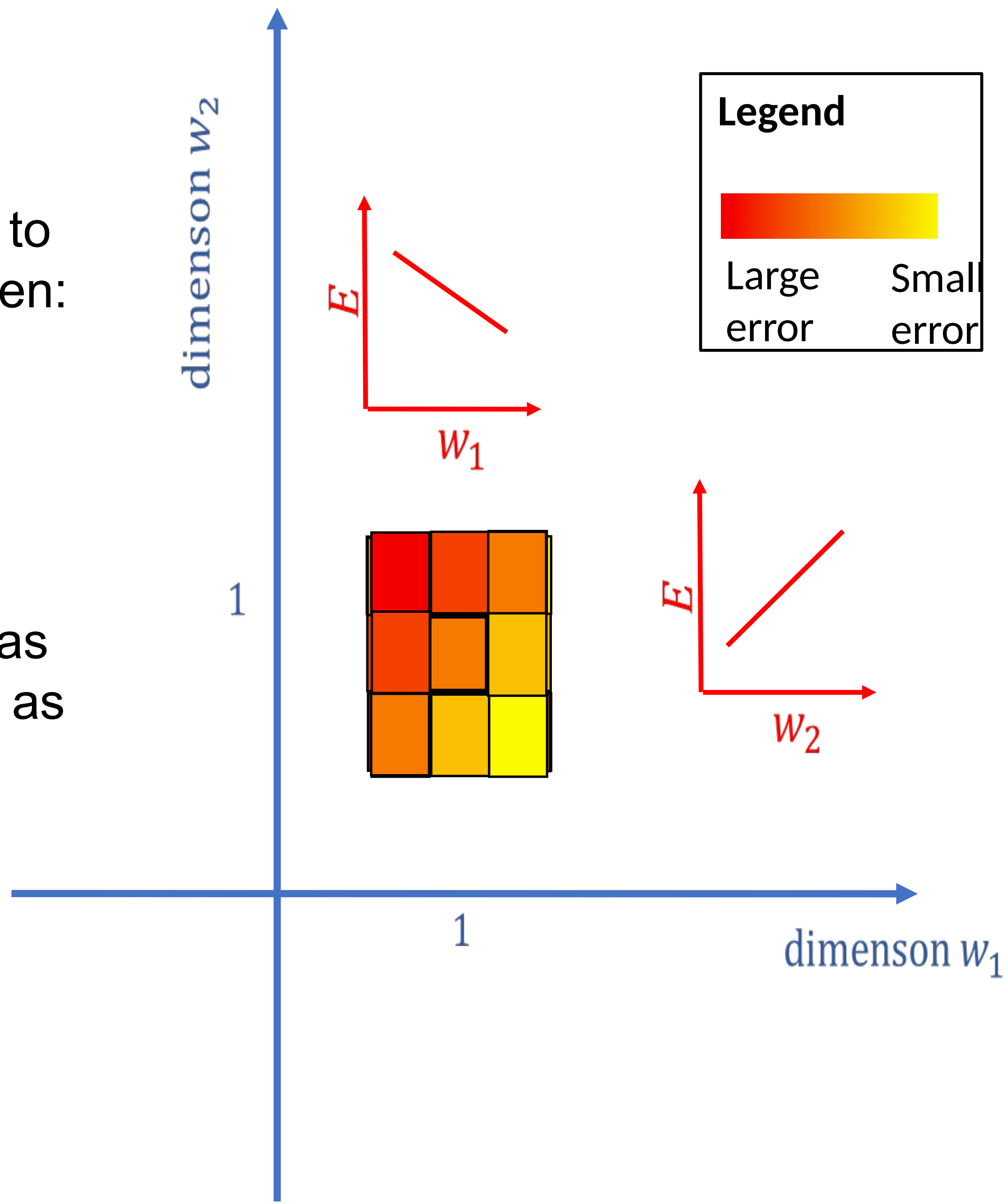
The change in Error with respect to a small change in w_2 can be written:

$$\frac{\partial E}{\partial w_2}$$

In our example, we can see that as w_1 increases, E ***decreases***, and as increases w_2, E ***increases***:

$$\frac{\partial E}{\partial w_1} < 0$$ ***negative slope***

$$\frac{\partial E}{\partial w_2} > 0$$ ***positive slope***



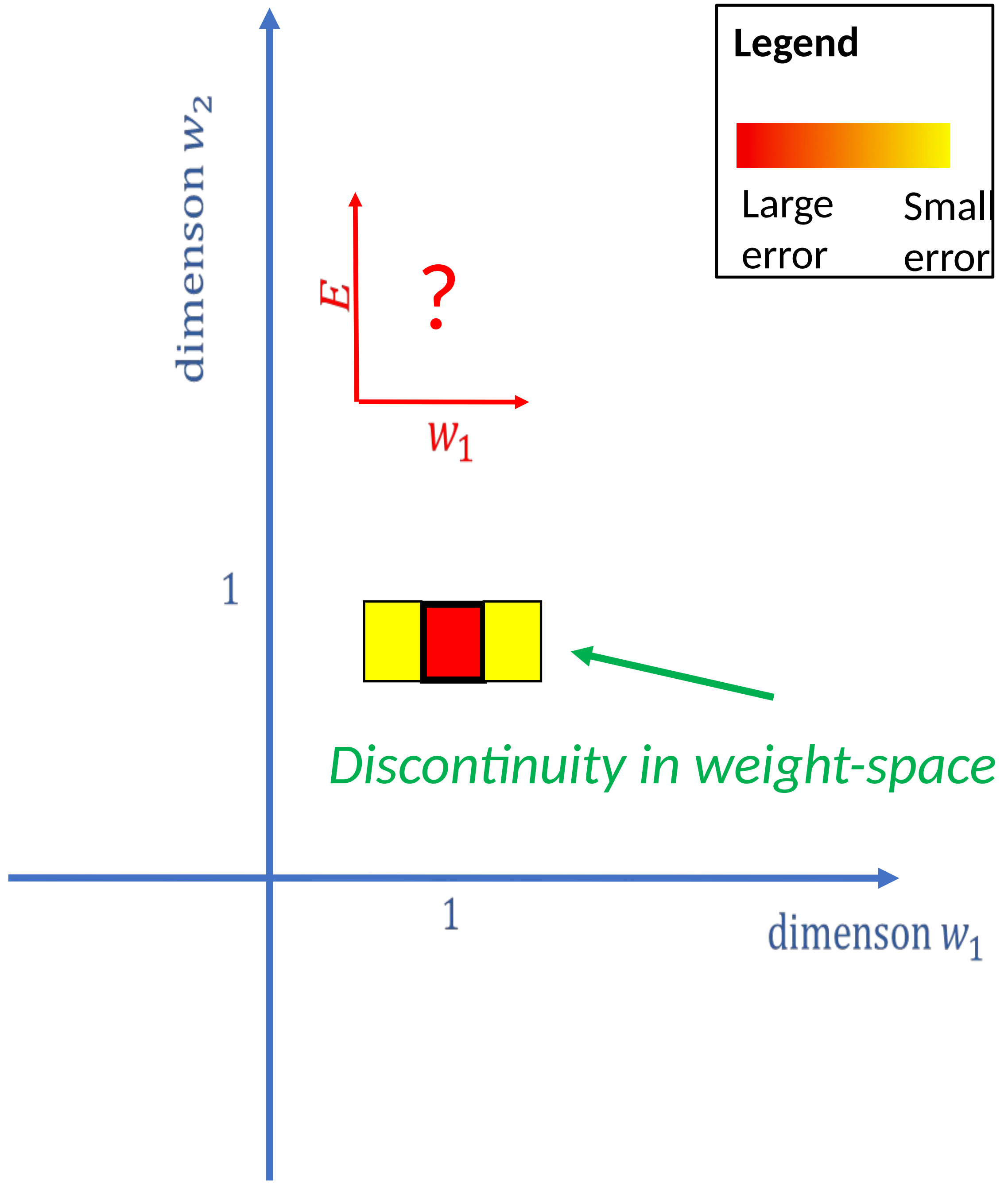
Weight space & gradient descent

By calculating the partial derivatives with respect to the weights, we obtain information about how the weights should be updated.

This depends on the Error surface being smooth (i.e. differentiable with respect to the weights).

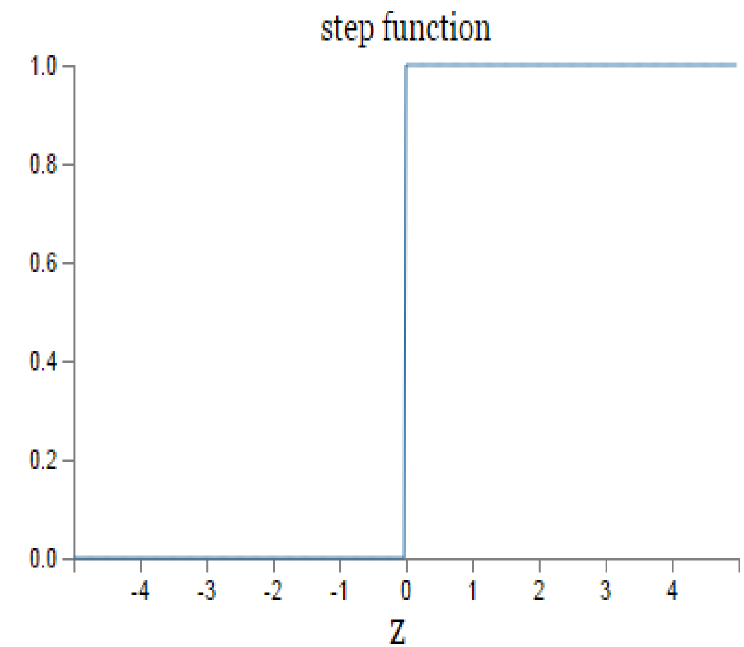
We need rolling hills and valleys in feature space, with no "cliff edges".

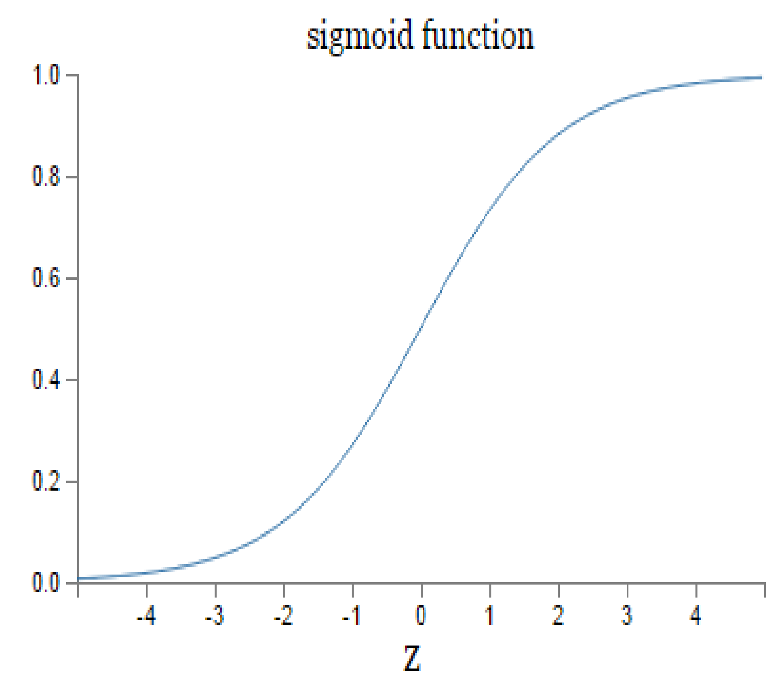
Crucially, this has implications for both how we calculate the Error, and the activation function we use to calculate neuron output.



Activation functions

- Perceptron problem:
  - A small change in the weights or bias of any single perceptron in the network can cause the output of that perceptron to completely flip
- Solution: a ***sigmoid* neuron**
  - Similar to perceptrons
  - but modified so that small changes in their weights and bias cause only a small change in their output.
  - A “smoothed out” perceptron





$$output = \sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b) = \frac{1}{1 + e^{-\boldsymbol{w} \cdot \boldsymbol{x} - b}}$$

which has a nice derivative of

$$\frac{\partial \sigma}{\partial z} = \sigma \cdot (1\text{-}\sigma)$$

where $z = wx + b$

Backpropagation

Error (over all output units j): $E = \frac{1}{2}\left\|t_j - y_j\right\|^2$ *t = target output* *y = observed output*

How error changes as output changes: $\frac{\partial E}{\partial y_j} = -(t_j - y_j)$

Remember calculus?

If $y = x^2$, then $\frac{dy}{dx} = 2x$

If $y = (5 - x)^2$, then $\frac{dy}{dx} = 2(5 - x) * -1$

Including ½ in the error formula gives us a nice derivative.

Backpropagation

- OK, so we can calculate how the error changes as output changes:

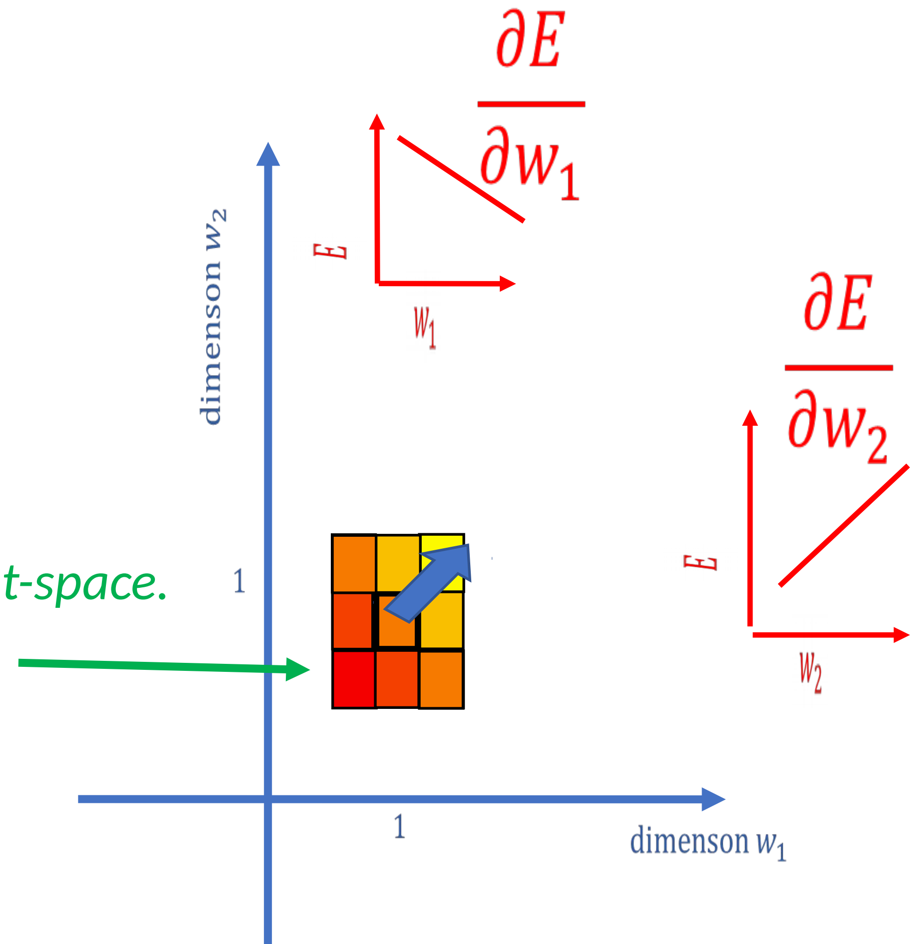
$$\frac{\partial E}{\partial y_j} = -(t_j - y_j)$$

t = target output
y = observed output

- What we really care about though is how the **Error** changes as the **weights** change:

$$\frac{\partial E}{\partial w_i} = ?$$

This is what tells us what direction to move in weight-space.



Backpropagation

Breaking the derivative $\frac{\partial E}{\partial w_i}$ **into its component parts:**

The weights affect the input to the next layer, which affect the output of the next layer, which affect the error value.

More calculus tricks:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

"Chain rule"

z is a variable that depends on y
y is a variable that depends on x
z depends on x indirectly, through the ***intermediate variable*** *y.*

Backpropagation

Breaking the derivative $\frac{\partial E}{\partial w_i}$ **into its component parts:**

The weights affect the input to the next layer, which affect the output of the next layer, which affect the error value.

More calculus tricks:

$$\frac{dz}{dx} = \frac{dz}{\cancel{dy}} \cdot \frac{\cancel{dy}}{dx}$$

"Chain rule"

z is a variable that depends on y
y is a variable that depends on x
z depends on x indirectly, through the ***intermediate variable*** *y.*

Backpropagation

Breaking $\frac{\partial E}{\partial w_i}$ **into its component parts:**

The weights affect the input to the next layer, which affect the output of the next layer, which affect the error value.

The change in the Error depends on the change of the output, which depends on the change in the input, which depends on the change of the weights:

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_i}$$

Input to output j is z_j:

$$z_j = \sum_{i=1}^{H} y_i w_i$$

inputs

output $y_j = \sigma(z_j)$

$$E = -\frac{1}{2}\|t_j - y_j\|^2$$

Backpropagation

Breaking $\frac{\partial E}{\partial w_i}$ into its component parts:

The weights affect the input to the next layer, which affect the output of the next layer, which affect the error value.

The **change in the Error depends on the change of the output**, which depends on the change in the input, which depends on the change of the weights:

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_i}$$

Backpropagation

Breaking $\frac{\partial E}{\partial w_i}$ **into its component parts:**

The weights affect the input to the next layer, which affect the output of the next layer, which affect the error value.

The change in the Error depends on the change of the output, which **depends on the change in the input**, which depends on the change of the weights:

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_i}$$

Backpropagation

Breaking $\frac{\partial E}{\partial w_i}$ **into its component parts:**

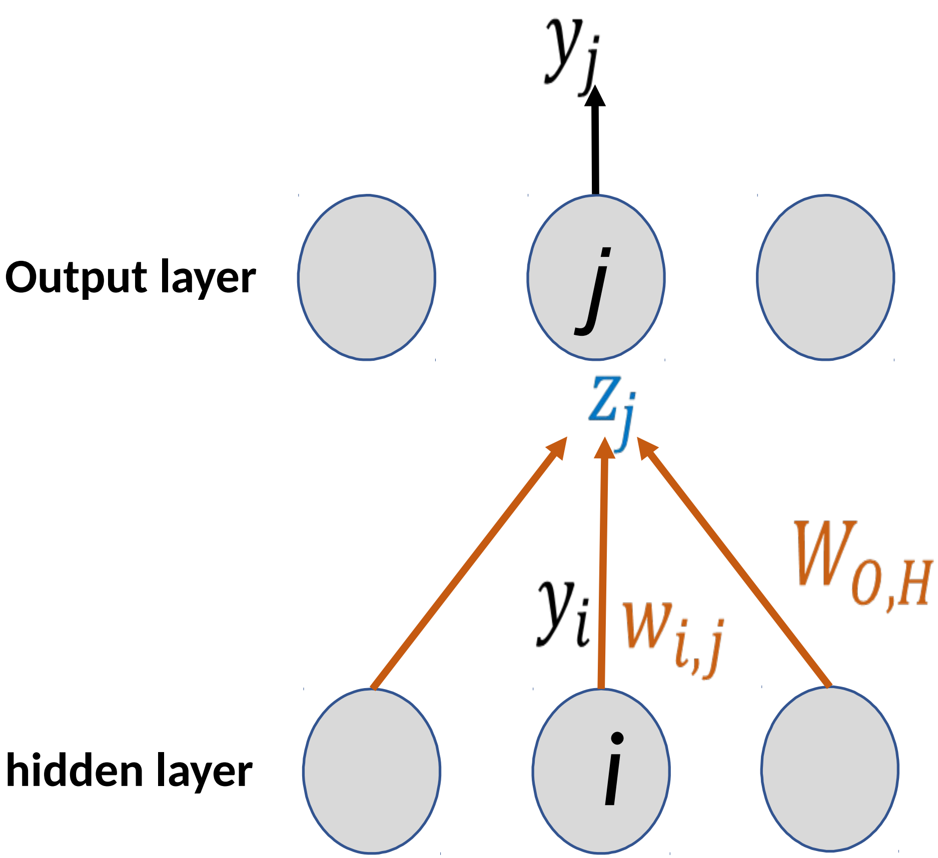
The weights affect the input to the next layer, which affect the output of the next layer, which affect the error value.

The change in the Error depends on the change of the output, which depends on the change in the input, which **depends on the change of the weights**:

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_i}$$

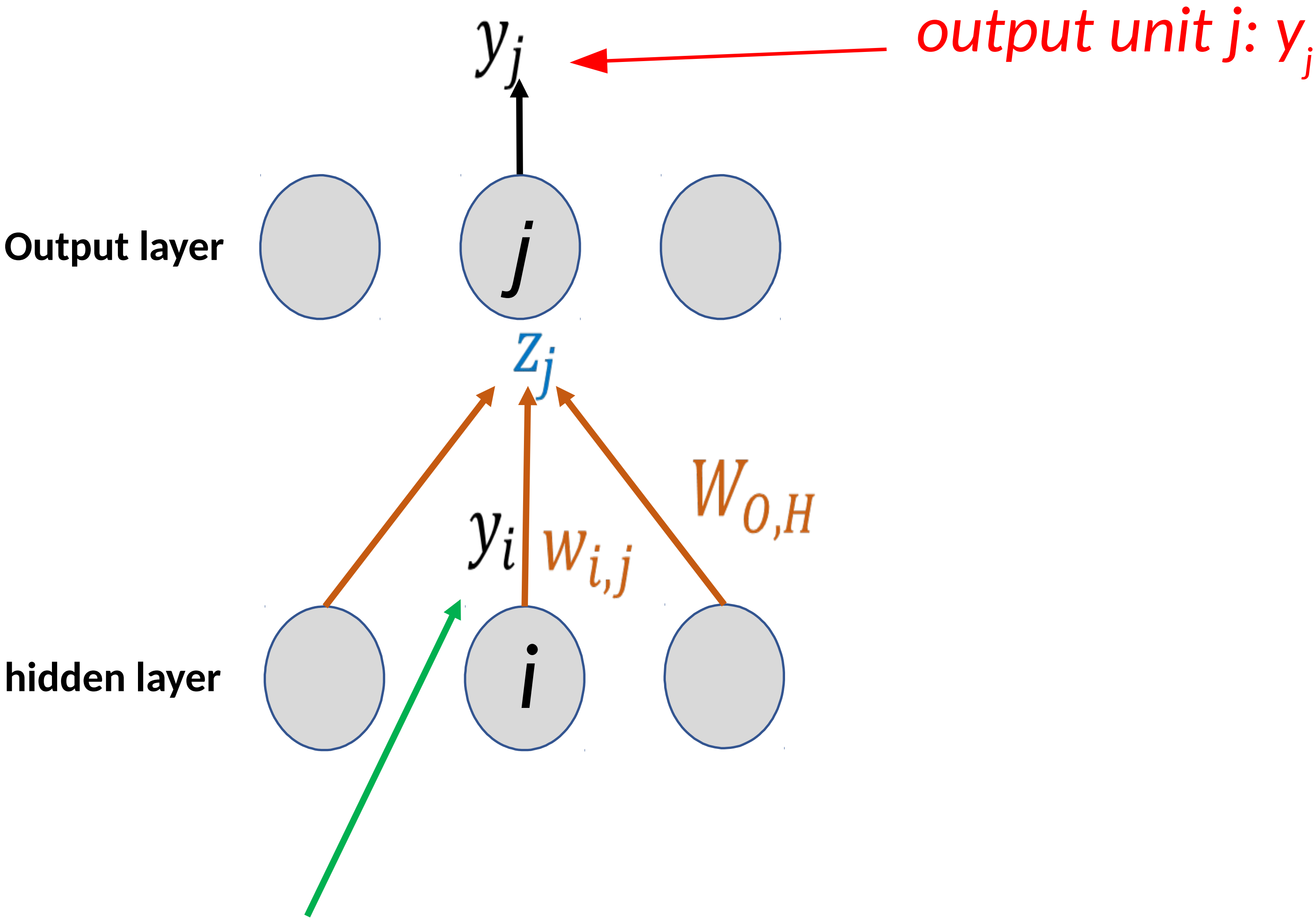
So we need to calculate these three partial derivatives (we've already done)

Backpropagation



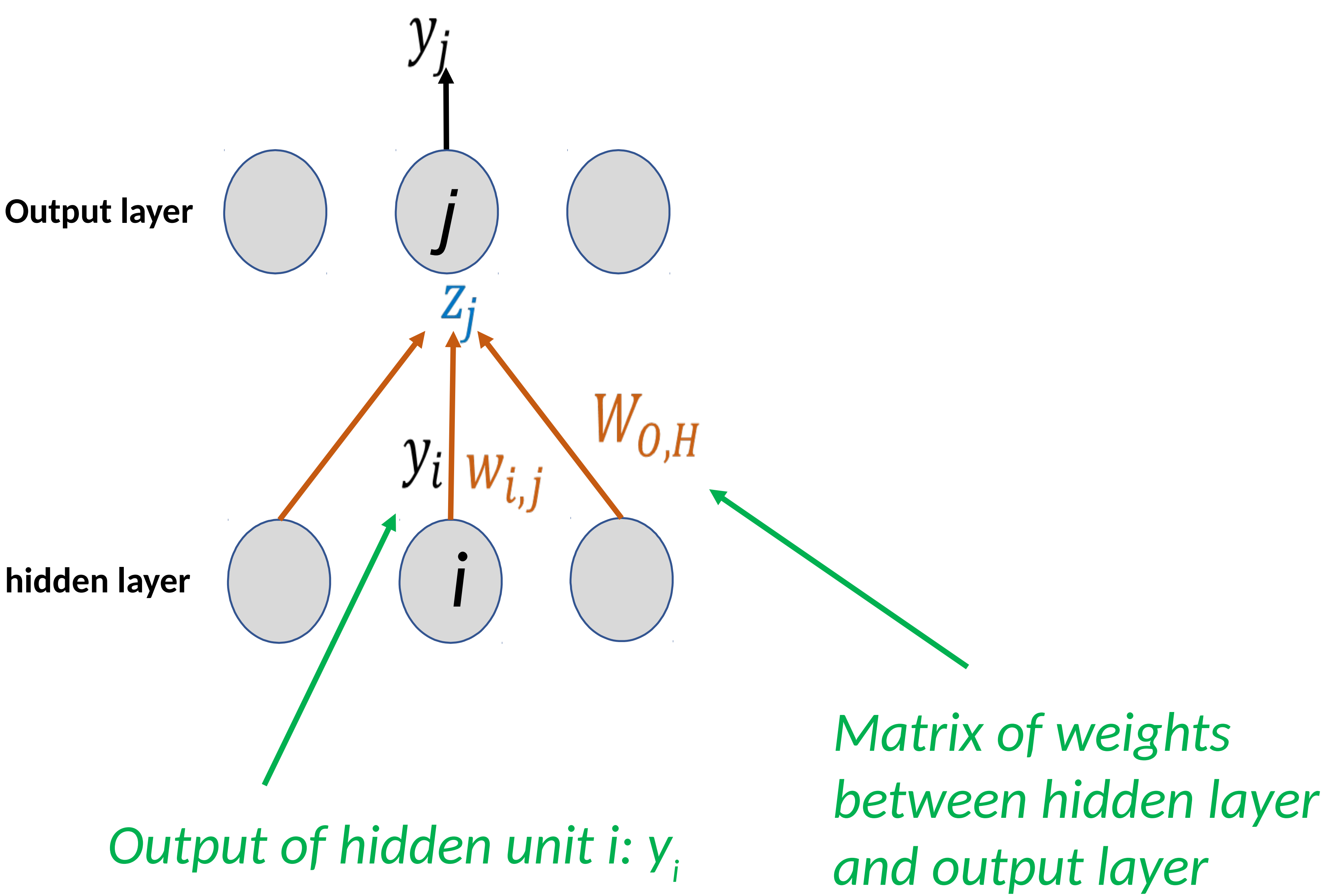
To understand backpropagation, let's consider what happens for particular neurons *j* and *i* in two layers: Output and Hidden

Backpropagation

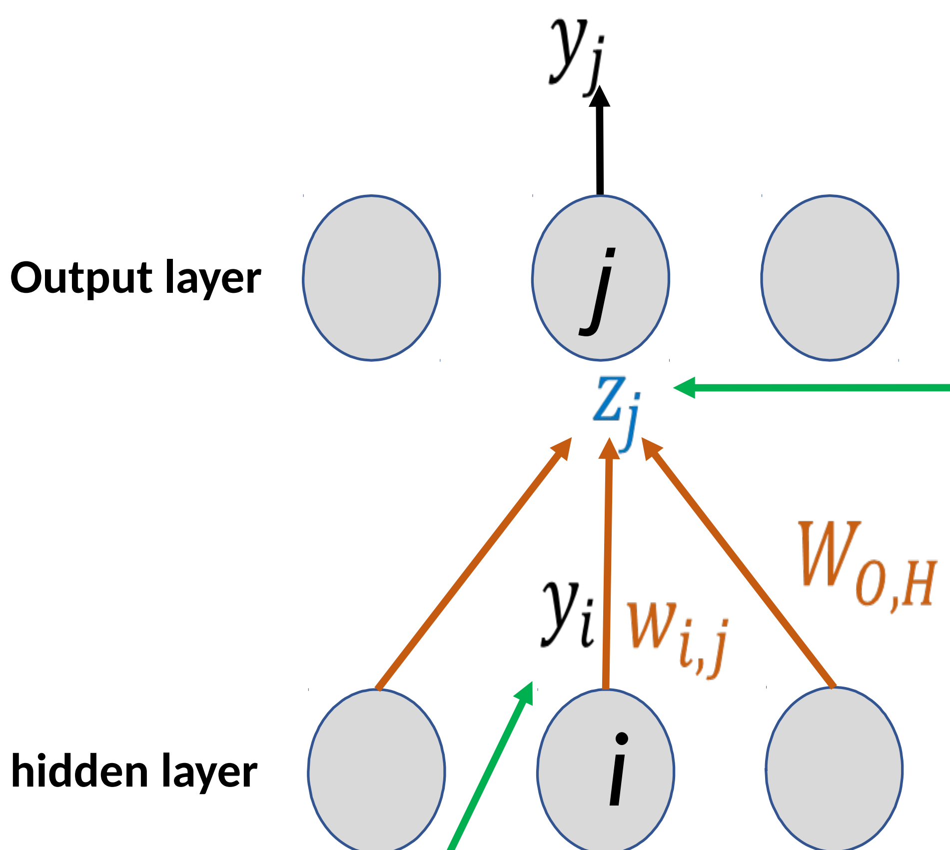


Output of hidden unit i: y_i (now become the inputs to output node j)

Backpropagation



Backpropagation



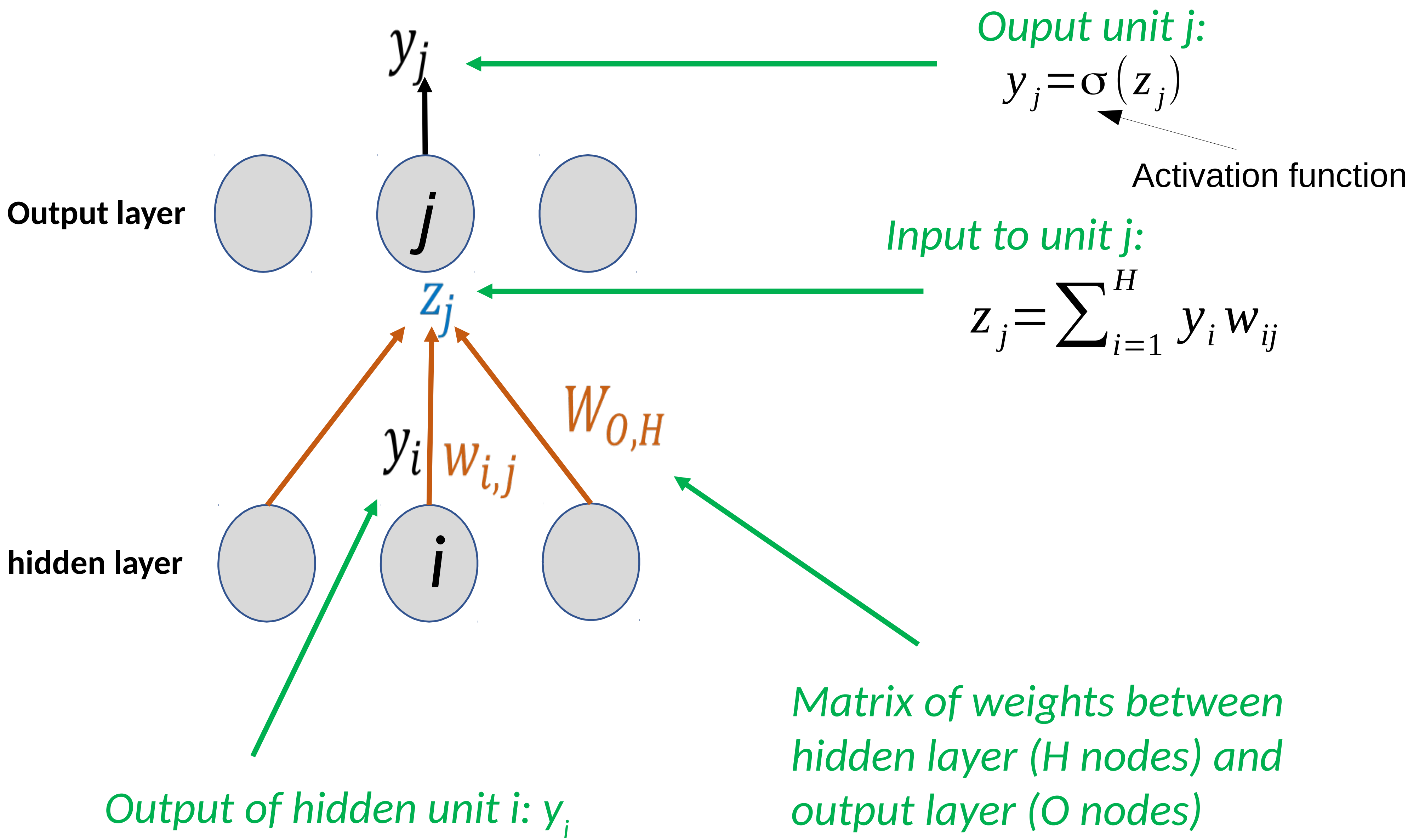
Input to unit j:

$$z_j = \sum_{i=1}^{H} y_i w_{ij}$$

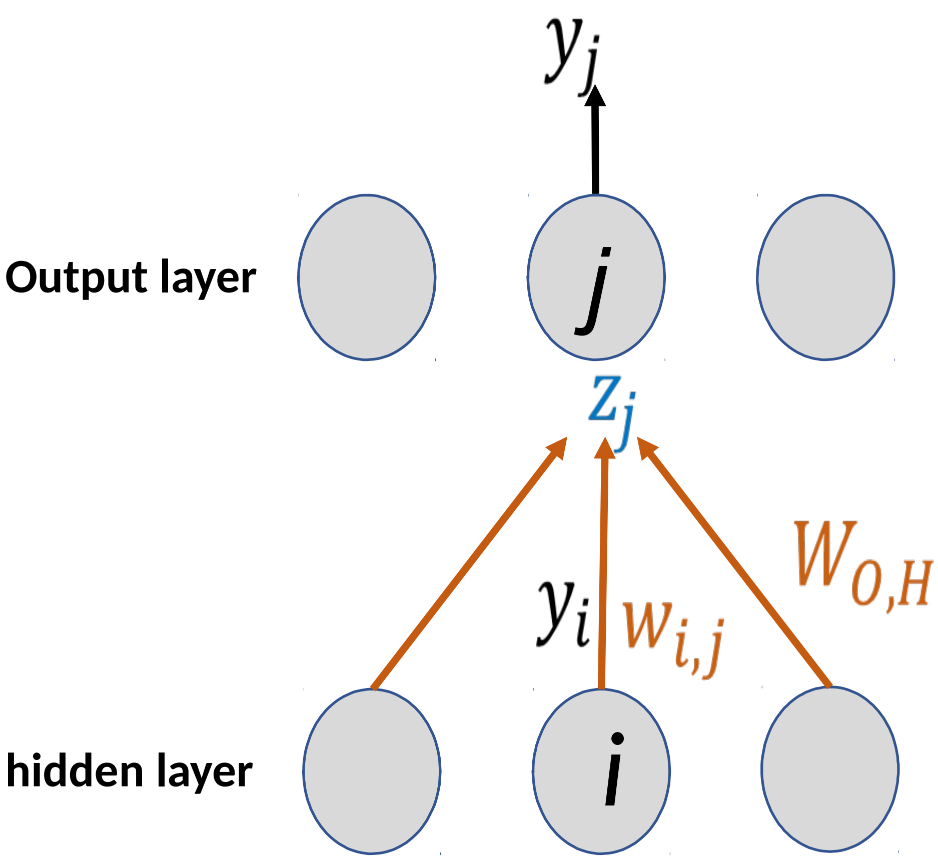
Output of hidden unit i: y_i

Matrix of weights between hidden layer (H nodes) and output layer (O nodes)

Backpropagation



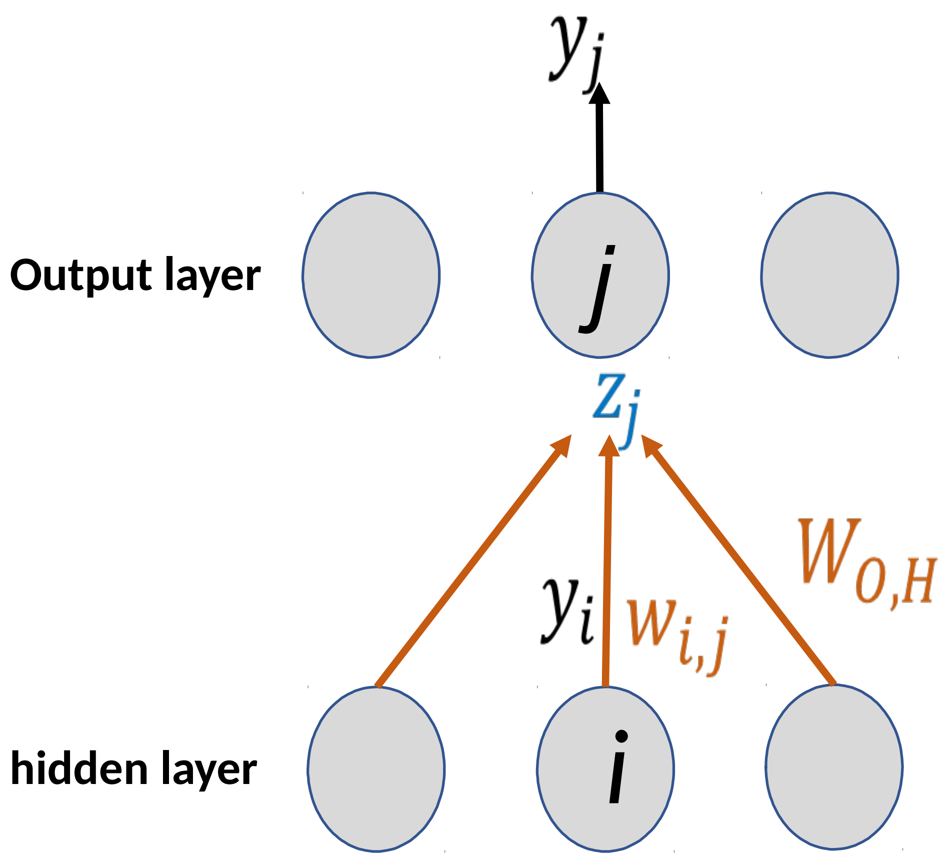
Backpropagation



$$\frac{\partial E}{\partial z_j}$$

How the error changes as the ***input*** *to unit j changes*

Backpropagation

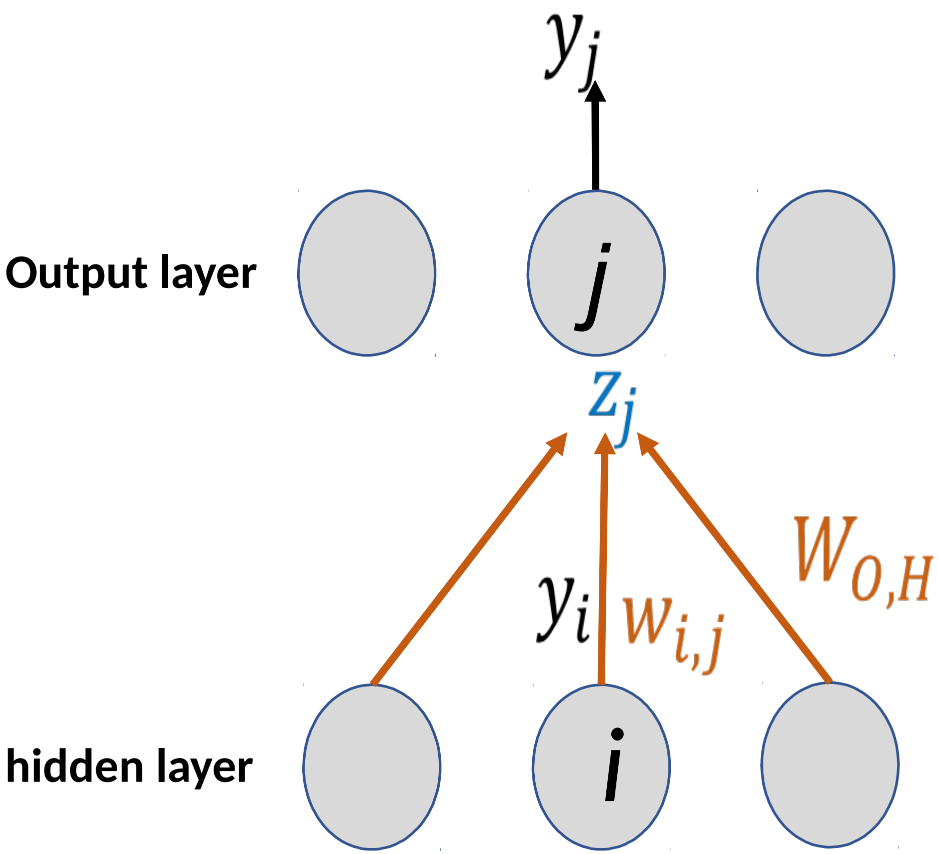


$$\frac{\partial E}{\partial z_j} = \frac{\partial y_j}{\partial z_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j}$$

*How the error changes as the **input** to unit j changes*

(the output of j) is intermediate between z_j and E

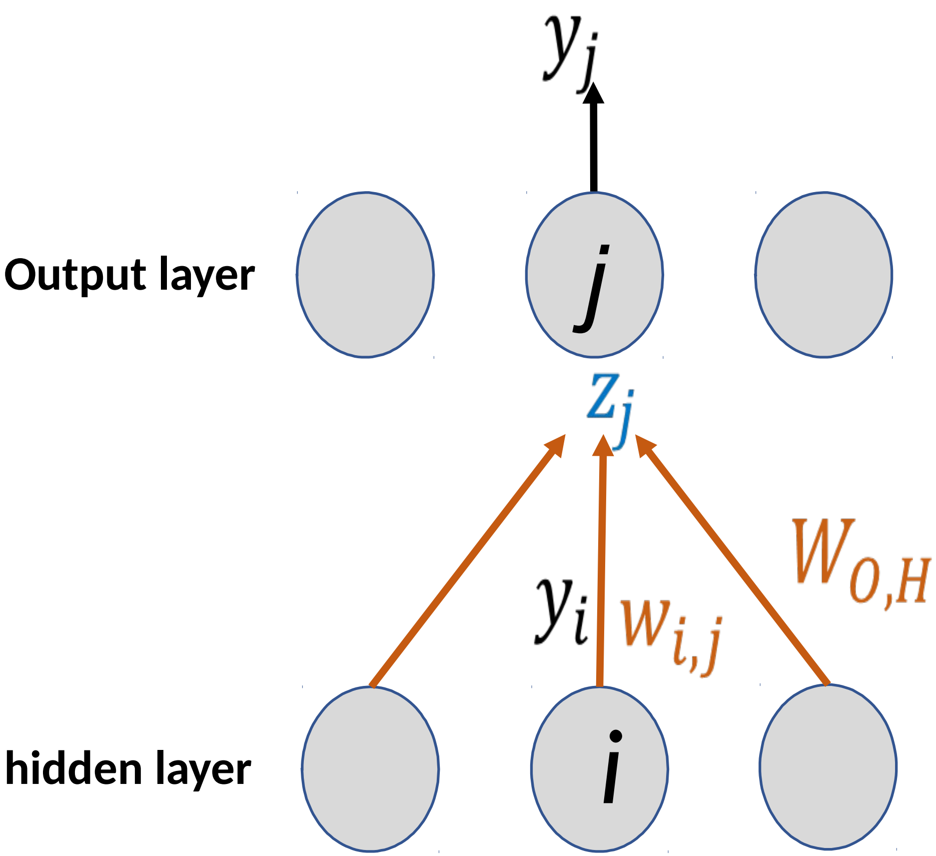
Backpropagation



$$\frac{\partial E}{\partial z_j} = \frac{\cancel{\partial y_j}}{\partial z_j}\frac{\partial E}{\cancel{\partial y_j}} = y_j(1-y_j)\frac{\partial E}{\partial y_j}$$

Chain rule of calculus

Backpropagation

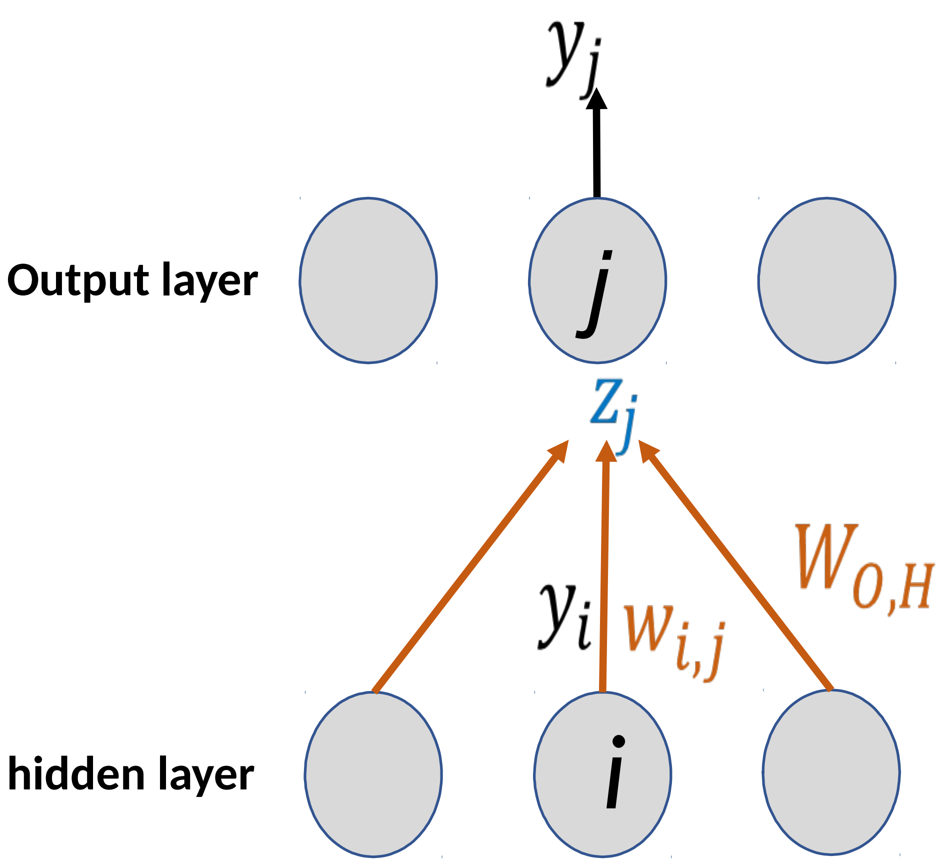


$$\frac{\partial E}{\partial z_j} = \boxed{\frac{\partial y_j}{\partial z_j}} \frac{\partial E}{\partial y_j} = y_j(1 - y_j) \frac{\partial E}{\partial y_j}$$

The derivative of the output function (sigmoid).

We've already seen on previous slides that the sigmoid function has a nice derivative.

Backpropagation

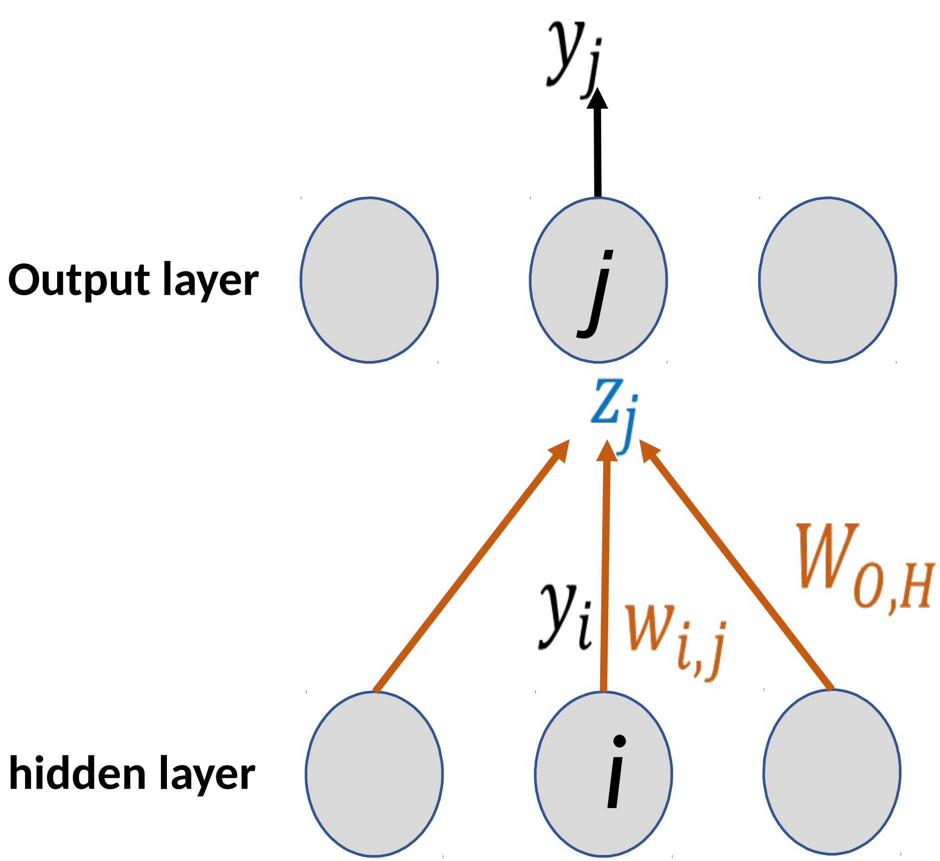


$$\frac{\partial E}{\partial z_j} = \frac{\partial y_j}{\partial z_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j}$$

The derivative of the output function (sigmoid)

Backpropagation

$$\frac{\partial E}{\partial z_j} = \frac{\partial y_j}{\partial z_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j}$$



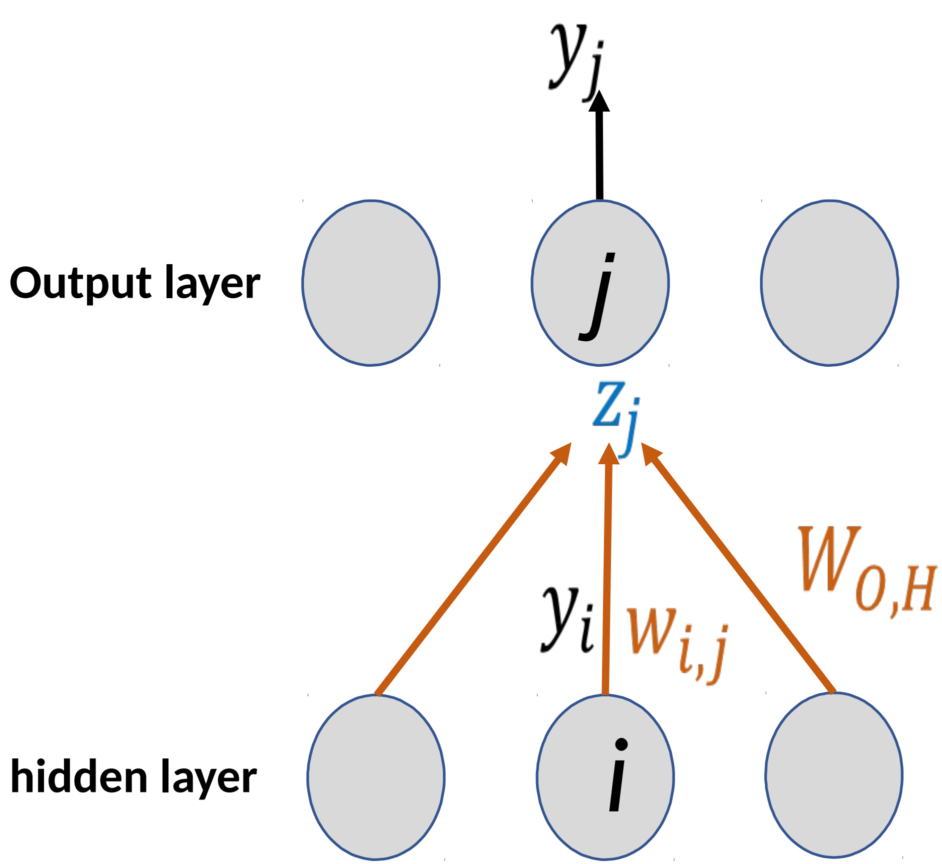
We already calculated this:

If $E = \frac{1}{2}\left\|t_j - y_j\right\|^2$

Then

$$\frac{\partial E}{\partial y_j} = -(t_j - y_j)$$

Backpropagation

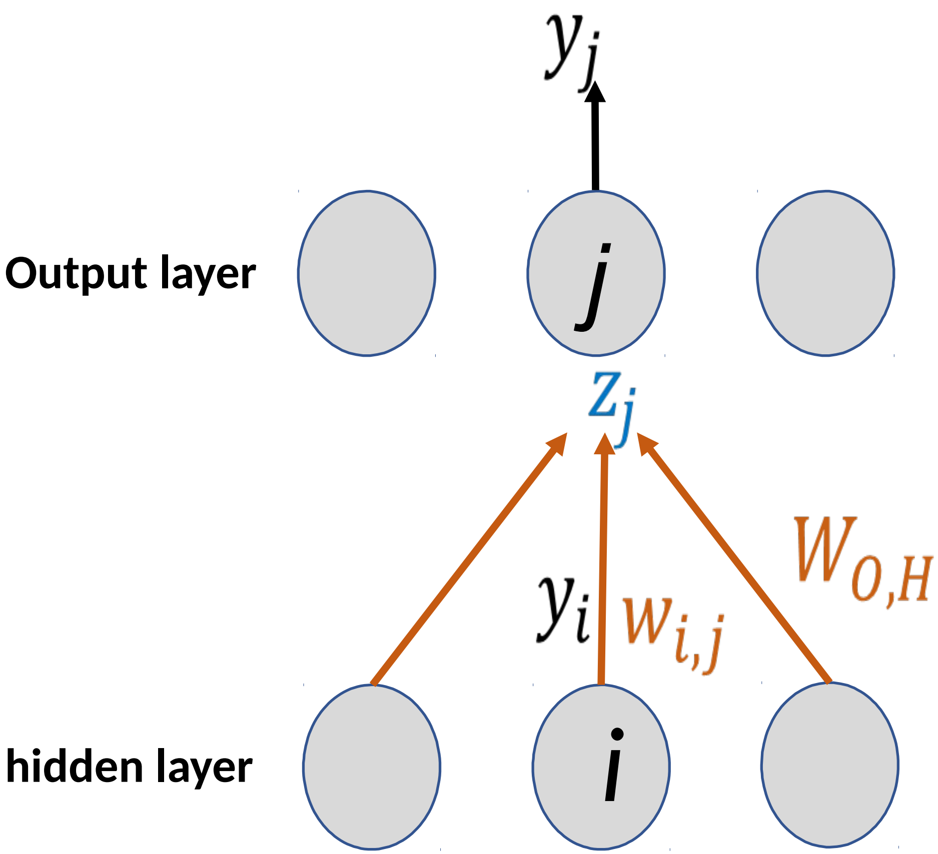


$$\frac{\partial E}{\partial z_j} = \frac{\partial y_j}{\partial z_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j}$$

$$\boxed{\frac{\partial E}{\partial w_{ij}}} = \frac{\partial z_j}{\partial w_{ij}}\frac{\partial E}{\partial z_j} = y_i\frac{\partial E}{\partial z_j}$$

How the error changes as the weight changes
(this is what we ultimately care about)

Backpropagation

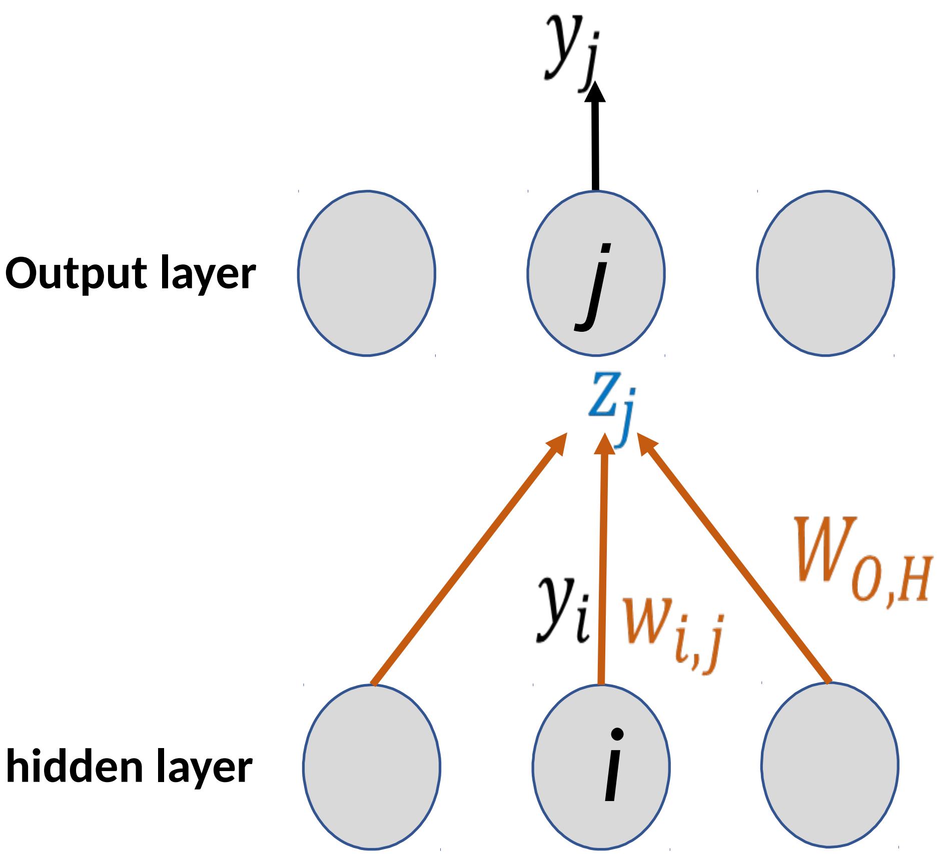


$$\frac{\partial E}{\partial z_j} = \frac{\partial y_j}{\partial z_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}}\frac{\partial E}{\partial z_j} = y_i\frac{\partial E}{\partial z_j}$$

Chain rule

Backpropagation



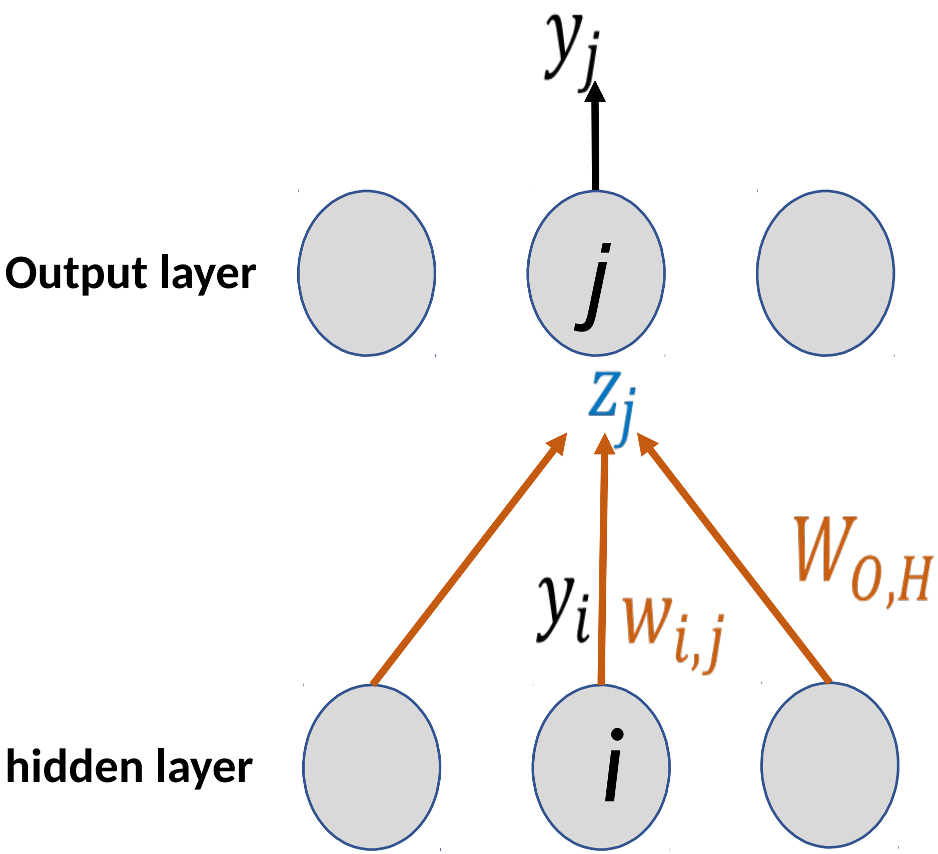
$$\frac{\partial E}{\partial z_j} = \frac{\partial y_j}{\partial z_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}}\frac{\partial E}{\partial z_j} = y_i\frac{\partial E}{\partial z_j}$$

Note: $z_j = \sum_{i \in hidden\,layer} y_i w_{ij}$

This derivative will just be the constant that w_{ij} is multiplied by in z_j, i.e. y_i

Backpropagation



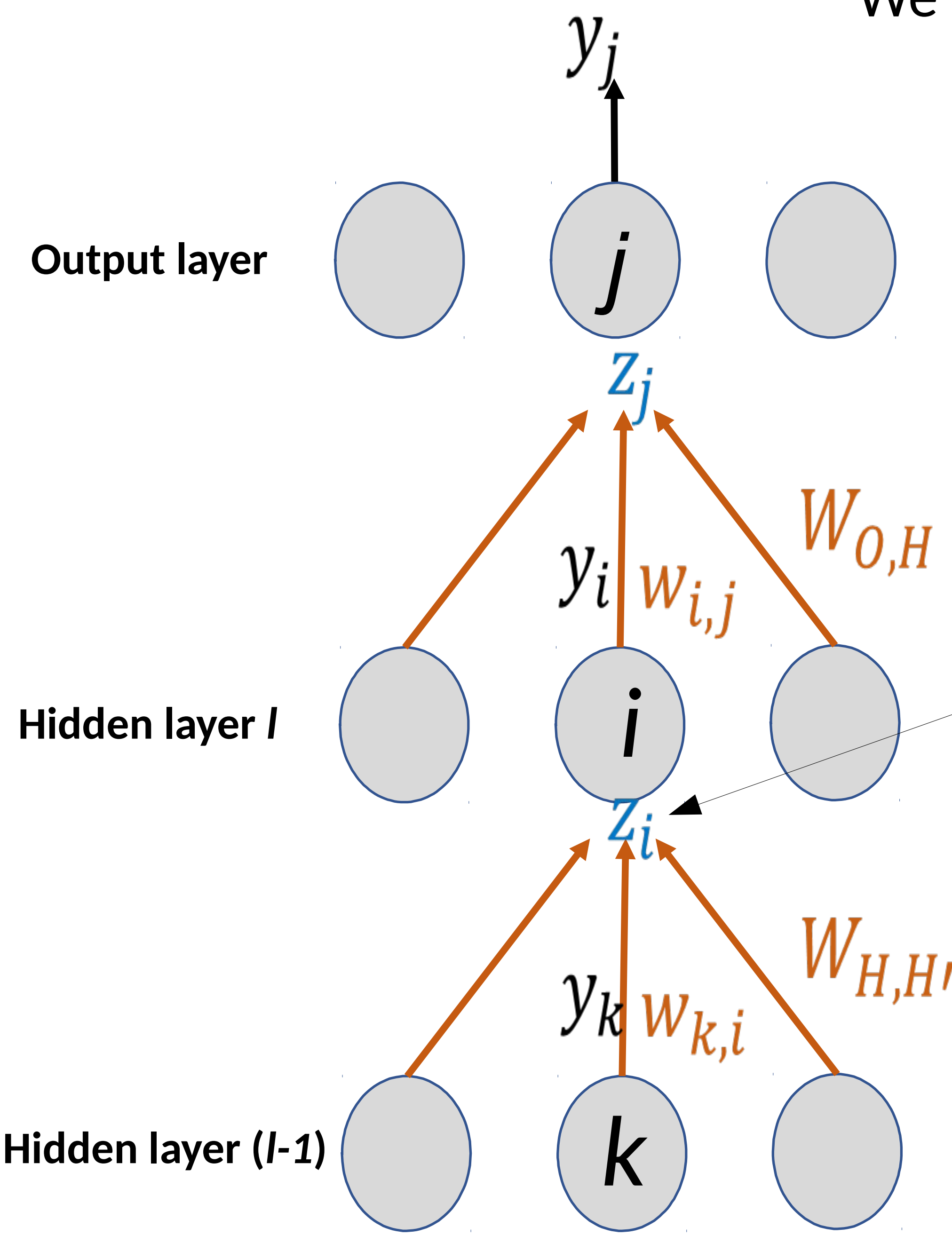
Putting it all together:

$$\frac{\partial E}{\partial w_{ij}} = y_i \frac{\partial E}{\partial z_j} = y_i(y_j(1\text{-}y_j)) \frac{\partial E}{\partial y_j}$$

$$= y_i(y_j(1\text{-}y_j)) \, (-\underbrace{(t_j - y_j)}_{\text{Error between expected target vs. predicted}})$$

We have found a way of expressing the change in the Error with respect to each weight as a function of the target output and the outputs of each neuron in the two layers.

Backpropagation



We can repeat the process for earlier layers.

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial z_i}{\partial w_{ki}} \frac{\partial E}{\partial z_i}$$

Chain rule

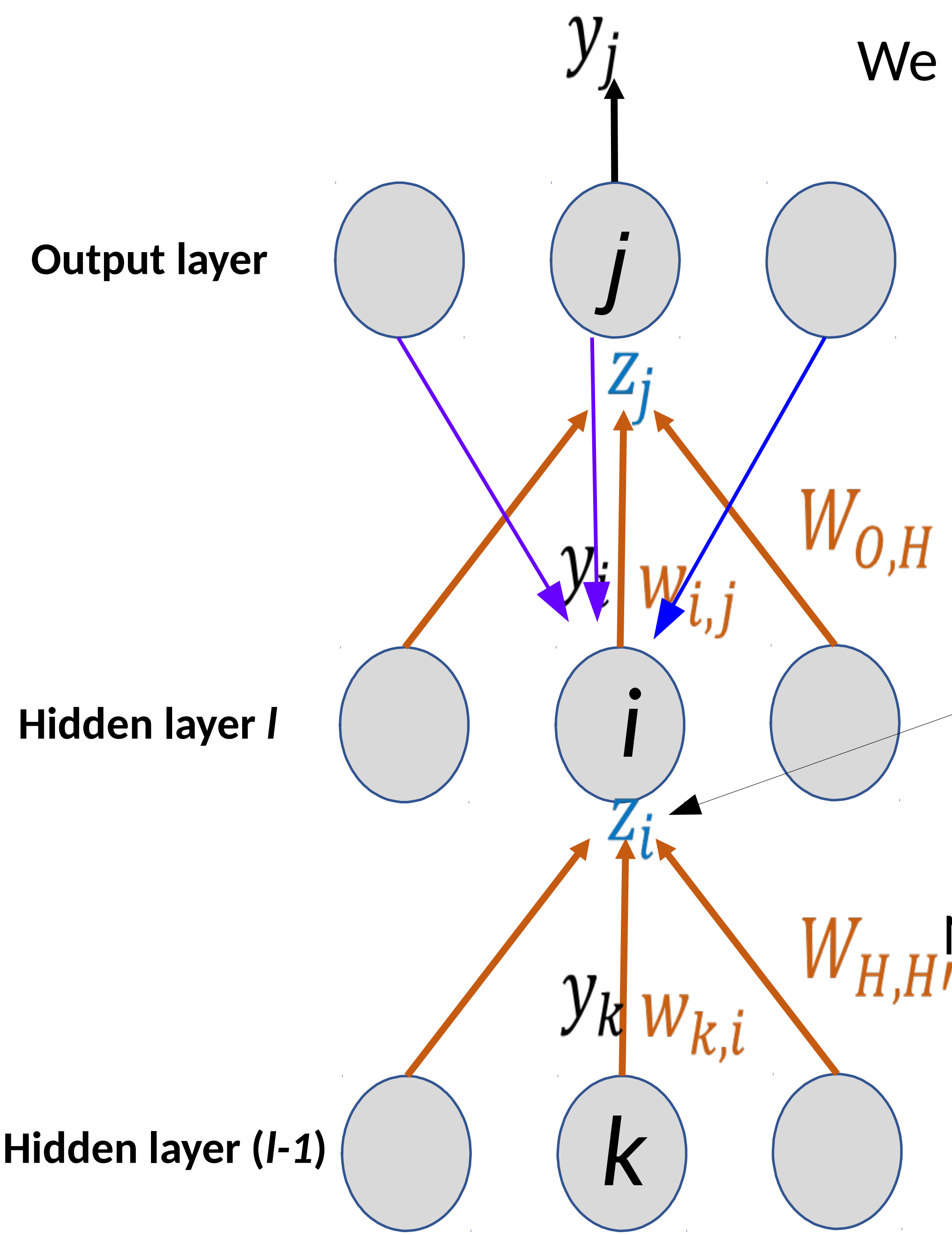
Note: $z_i = \sum_{k \in layer(l-1)} y_k w_{ki}$

$$\frac{\partial E}{\partial w_{ki}} = y_k \frac{\partial E}{\partial z_i}$$

Note that now hidden node *i has no target*

$$\frac{\partial E}{\partial z_i} = ???$$

Backpropagation



We can repeat the process for earlier layers.

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial z_i}{\partial w_{ki}} \frac{\partial E}{\partial z_i}$$

Chain rule

Note: $z_i = \sum_{k \in layer(l-1)} y_k w_{ki}$

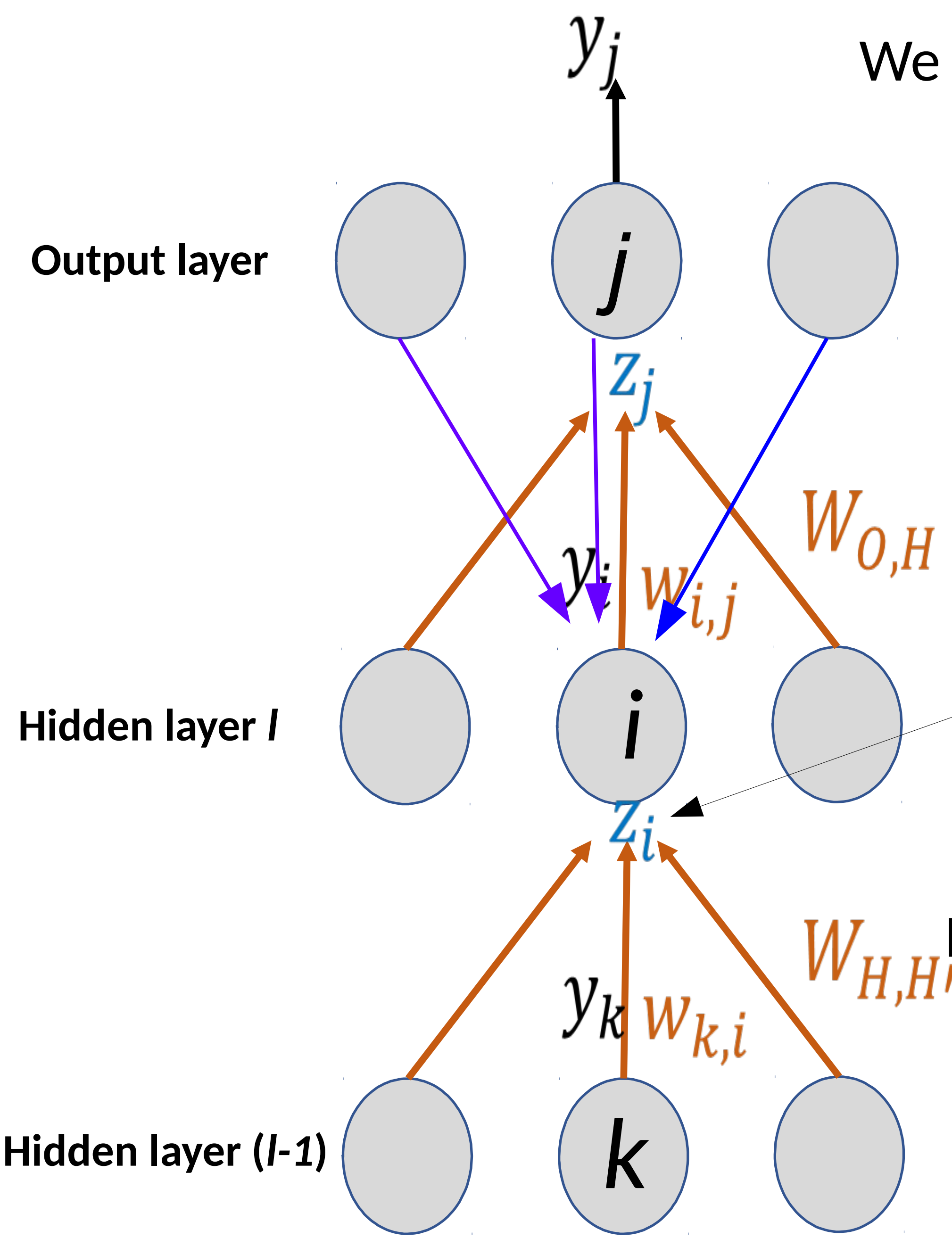
$$\frac{\partial E}{\partial w_{ki}} = y_k \frac{\partial E}{\partial z_i}$$

Note that now hidden node *i has no target*

$$\frac{\partial E}{\partial z_i} = \sum_{j \in O} \frac{\partial y_i}{\partial z_i} \frac{\partial z_j}{\partial y_i} \frac{\partial E}{\partial z_j}$$

Chain rule

Backpropagation



We can repeat the process for earlier layers.

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial z_i}{\partial w_{ki}} \frac{\partial E}{\partial z_i}$$

Chain rule

Note: $z_i = \sum_{k \in layer(l-1)} y_k w_{ki}$

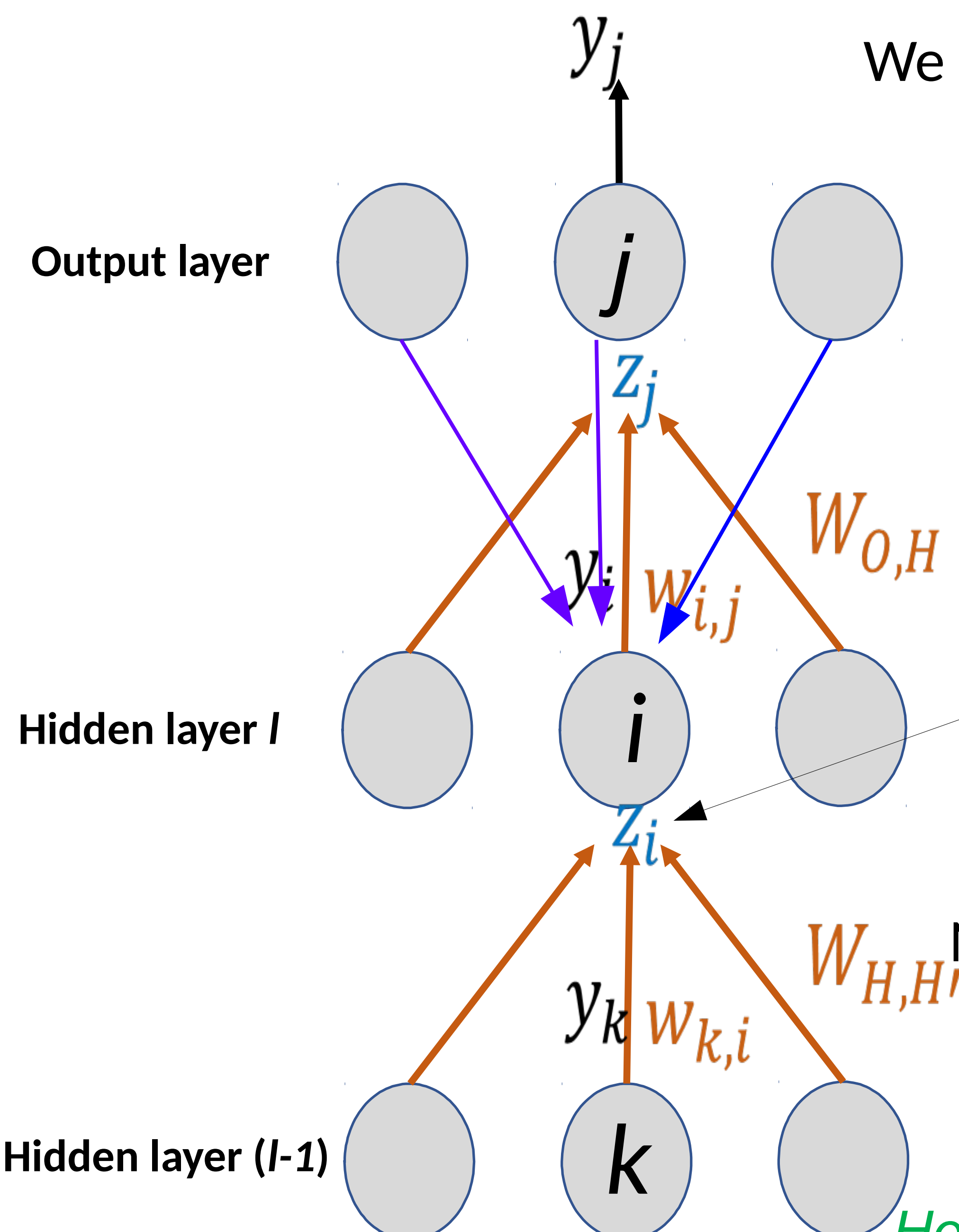
$$\frac{\partial E}{\partial w_{ki}} = y_k \frac{\partial E}{\partial z_i}$$

Note that now *hidden node i has no target*

$$\frac{\partial E}{\partial z_i} = \sum_{j \in O} \frac{\cancel{\partial y_i}}{\partial z_i} \frac{\cancel{\partial z_j}}{\cancel{\partial y_i}} \frac{\partial E}{\cancel{\partial z_j}}$$

Chain rule

Backpropagation



We can repeat the process for earlier layers.

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial z_i}{\partial w_{ki}} \frac{\partial E}{\partial z_i}$$

Chain rule

Note: $z_i = \sum_{k \in layer(l-1)} y_k w_{ki}$

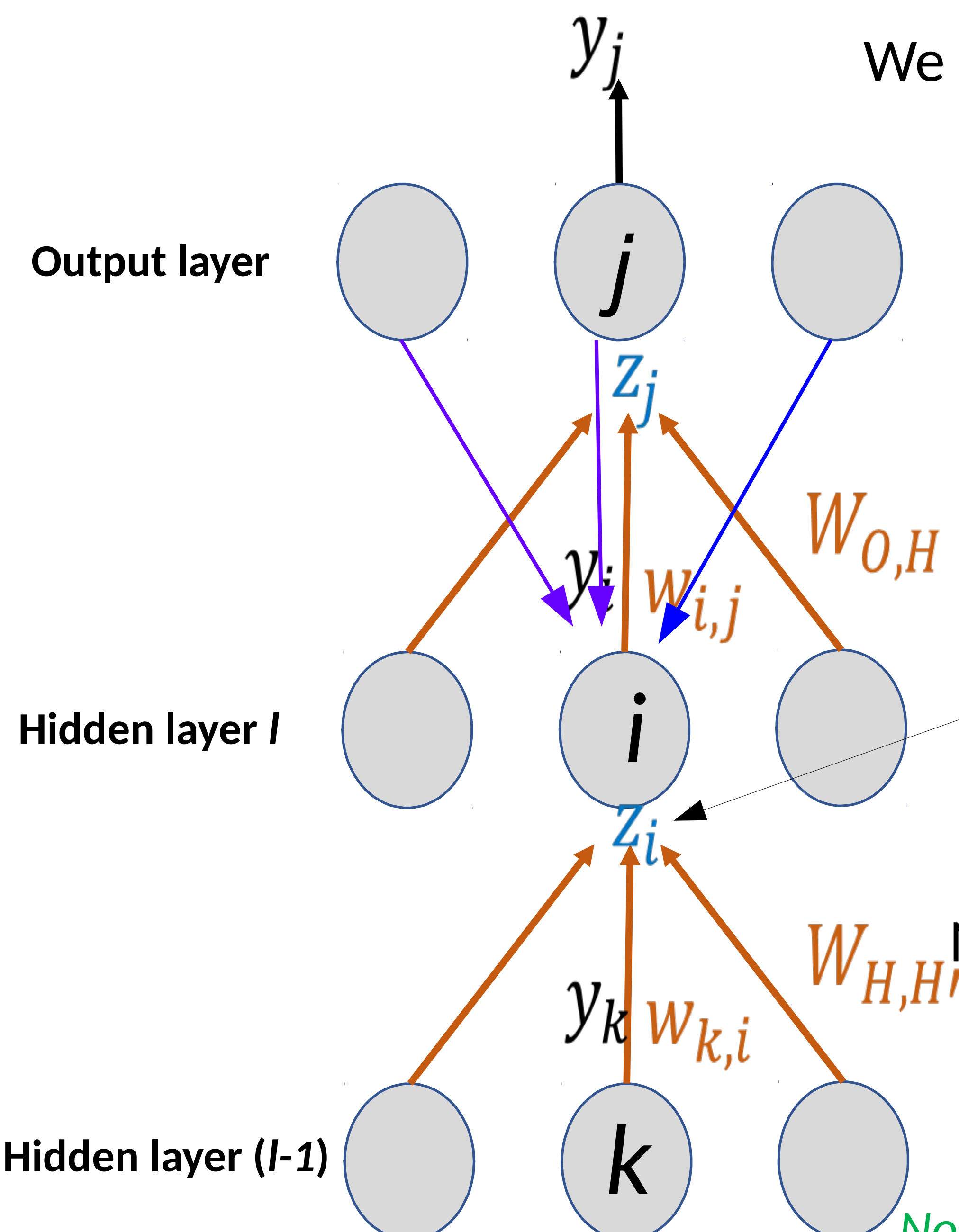
$$\frac{\partial E}{\partial w_{ki}} = y_k \frac{\partial E}{\partial z_i}$$

Note that now hidden node *i* has no target

$$\boxed{\frac{\partial E}{\partial z_i}} = \sum_{j \in O} \frac{\partial y_i}{\partial z_i} \frac{\partial z_j}{\partial y_i} \frac{\partial E}{\partial z_j}$$

*How the error changes as the input of unit **i** changes*

Backpropagation



We can repeat the process for earlier layers.

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial z_i}{\partial w_{ki}} \frac{\partial E}{\partial z_i}$$

Chain rule

Note: $z_i = \sum_{k \in layer(l-1)} y_k w_{ki}$

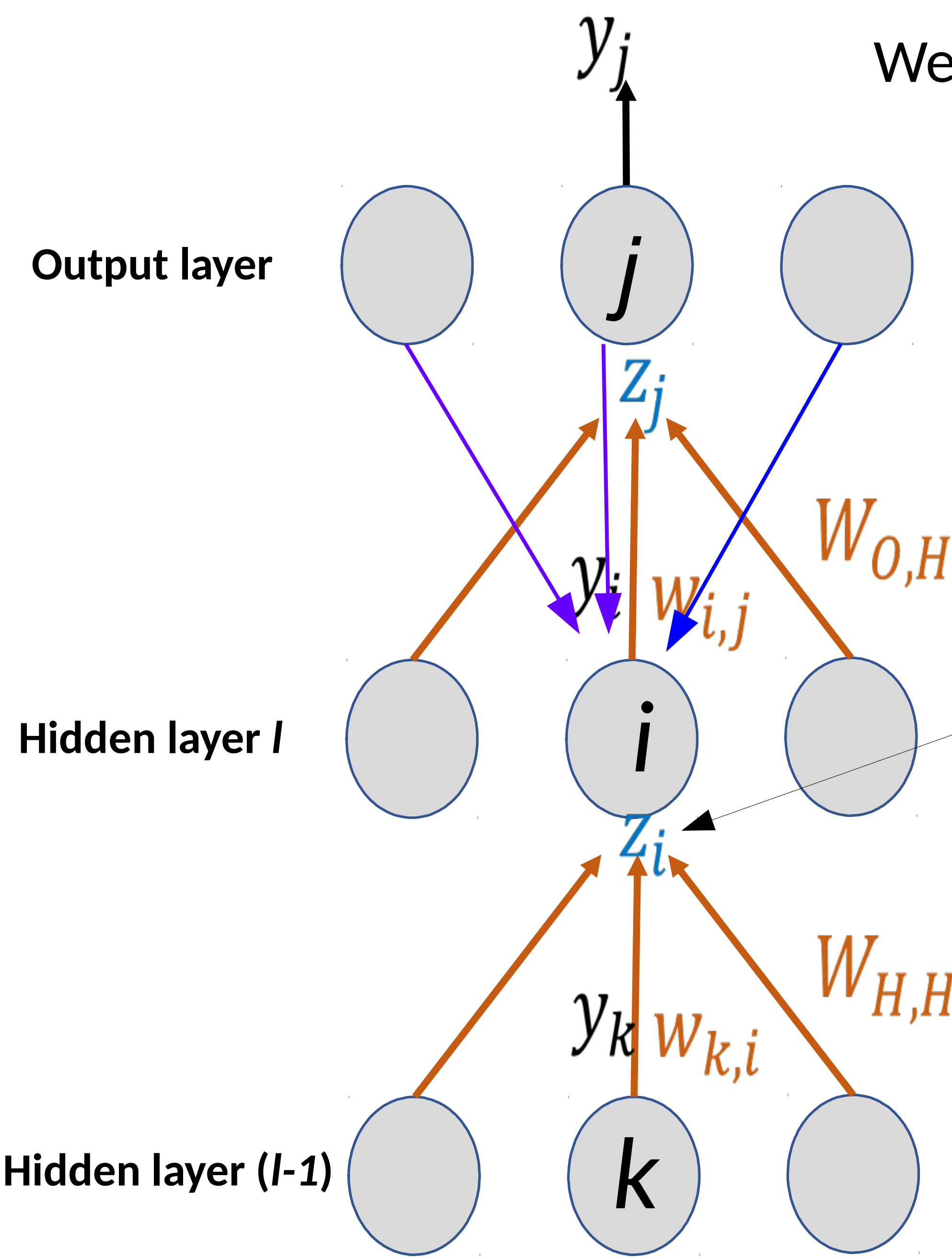
$$\frac{\partial E}{\partial w_{ki}} = y_k \frac{\partial E}{\partial z_i}$$

Note that now hidden node *i has no target*

$$\frac{\partial E}{\partial z_i} = \boxed{\sum_{j \in outputs}} \frac{\partial y_i}{\partial z_i} \frac{\partial z_j}{\partial y_i} \frac{\partial E}{\partial z_j}$$

*Note that how the error changes as z_i changes depends on **all** the units in the subsequent layer.*

Backpropagation



We can repeat the process for earlier layers.

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial z_i}{\partial w_{ki}} \frac{\partial E}{\partial z_i}$$

Chain rule

Note: $z_i = \sum_{k \in layer(l-1)} y_k w_{ki}$

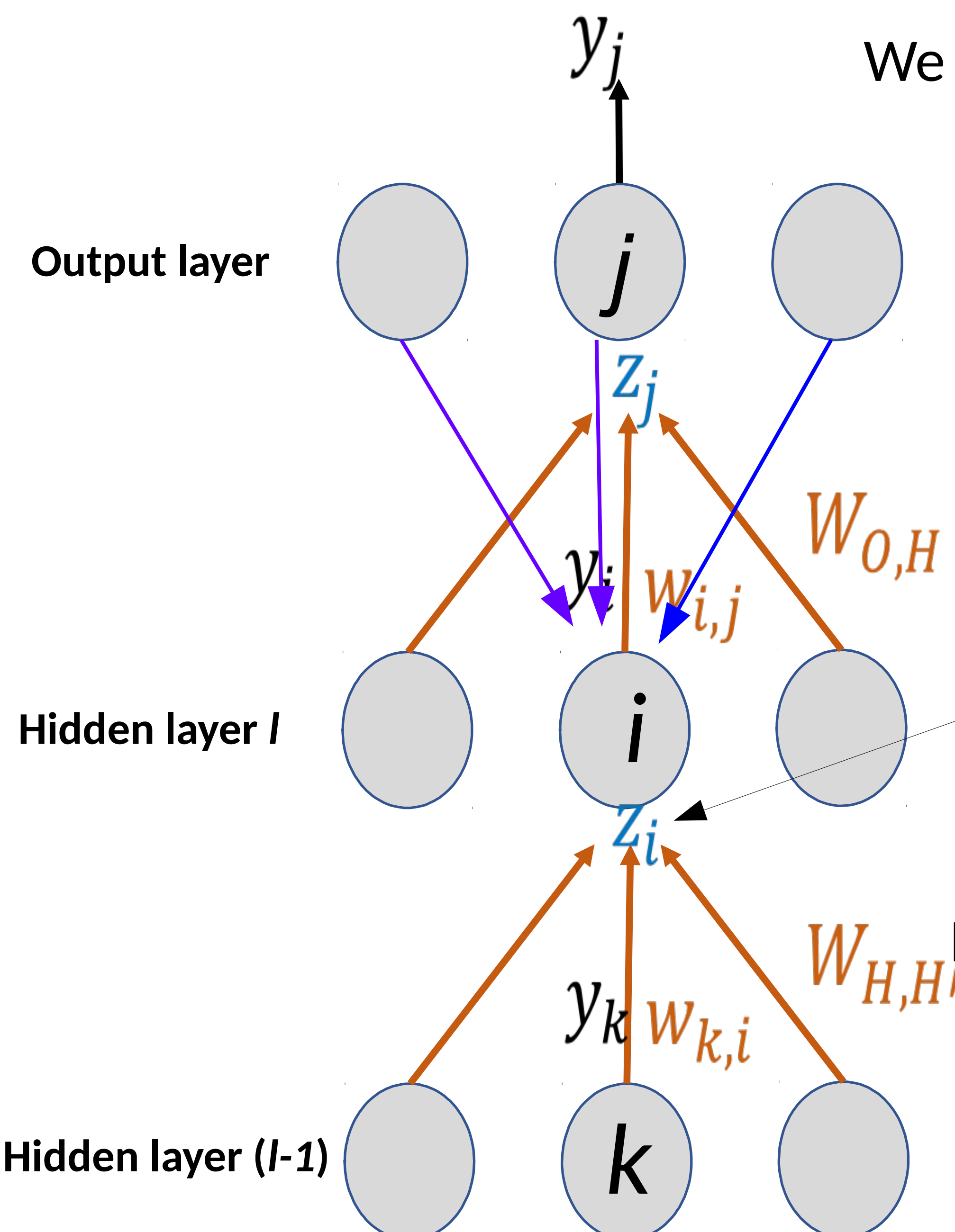
$$\frac{\partial E}{\partial w_{ki}} = y_k \frac{\partial E}{\partial z_i}$$

Note that now hidden node *i has no target*

$$\frac{\partial E}{\partial z_i} = \sum_{j \in outputs} \boxed{\frac{\partial y_i}{\partial z_i}} \frac{\partial z_j}{\partial y_i} \frac{\partial E}{\partial z_j}$$

Note: $y_i = \sigma(z_i) \xrightarrow{\text{If sigmoid}} \frac{\partial y_i}{\partial z_i} = y_i(1 - y_i)$

Backpropagation



We can repeat the process for earlier layers.

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial z_i}{\partial w_{ki}} \frac{\partial E}{\partial z_i}$$

Chain rule

Note: $z_i = \sum_{k \in layer(l-1)} y_k w_{ki}$

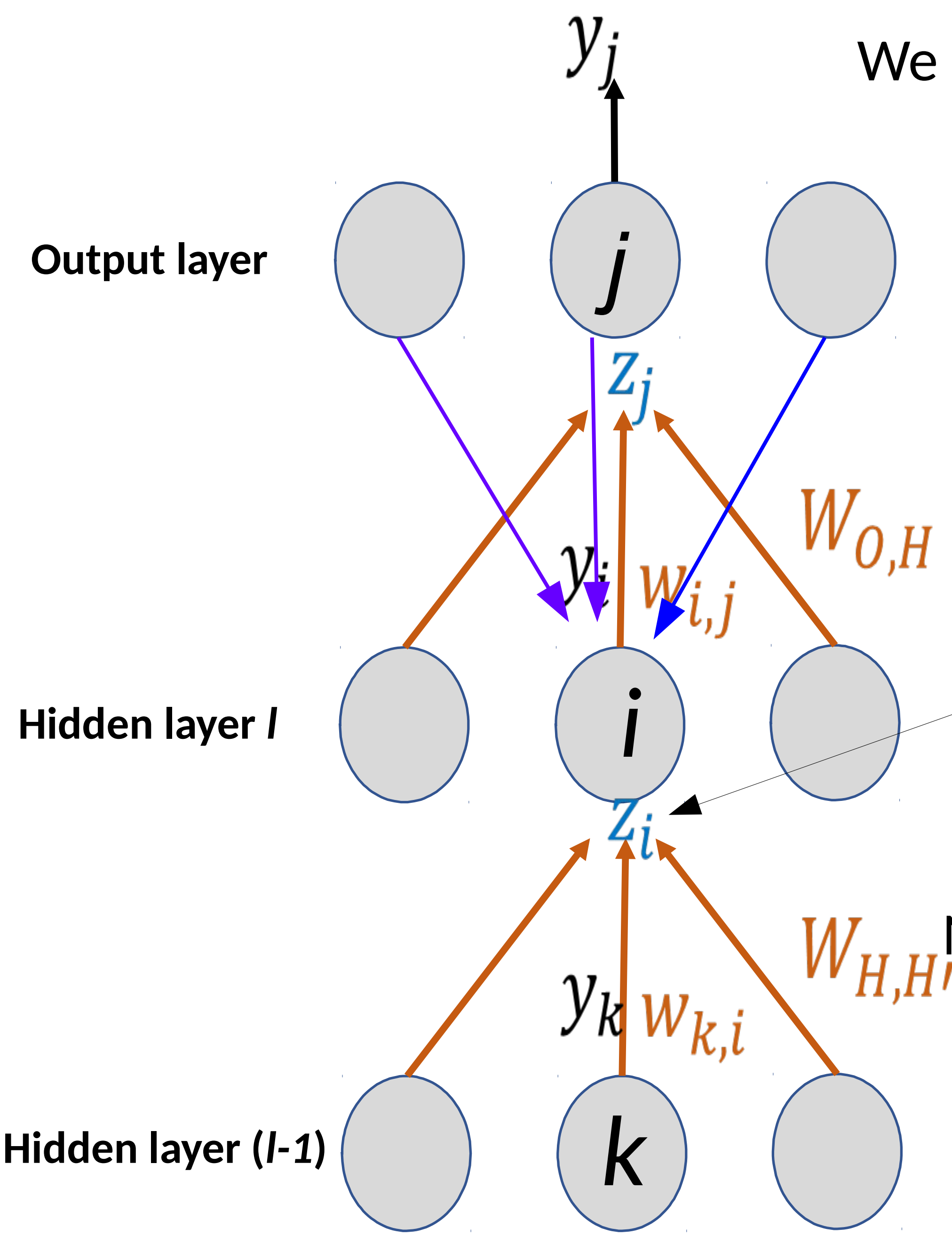
$$\frac{\partial E}{\partial w_{ki}} = y_k \frac{\partial E}{\partial z_i}$$

Note that now hidden node *i has no target*

$$\frac{\partial E}{\partial z_i} = \sum_{j \in outputs} \frac{\partial y_i}{\partial z_i} \boxed{\frac{\partial z_j}{\partial y_i}} \frac{\partial E}{\partial z_j}$$

Note: $z_j = \sum_{i \in layer\, l} y_i w_{ij} \longrightarrow \frac{\partial z_j}{\partial y_i} = w_{ij}$

Backpropagation



We can repeat the process for earlier layers.

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial z_i}{\partial w_{ki}} \frac{\partial E}{\partial z_i}$$

Chain rule

Note: $z_i = \sum_{k \in layer(l-1)} y_k w_{ki}$

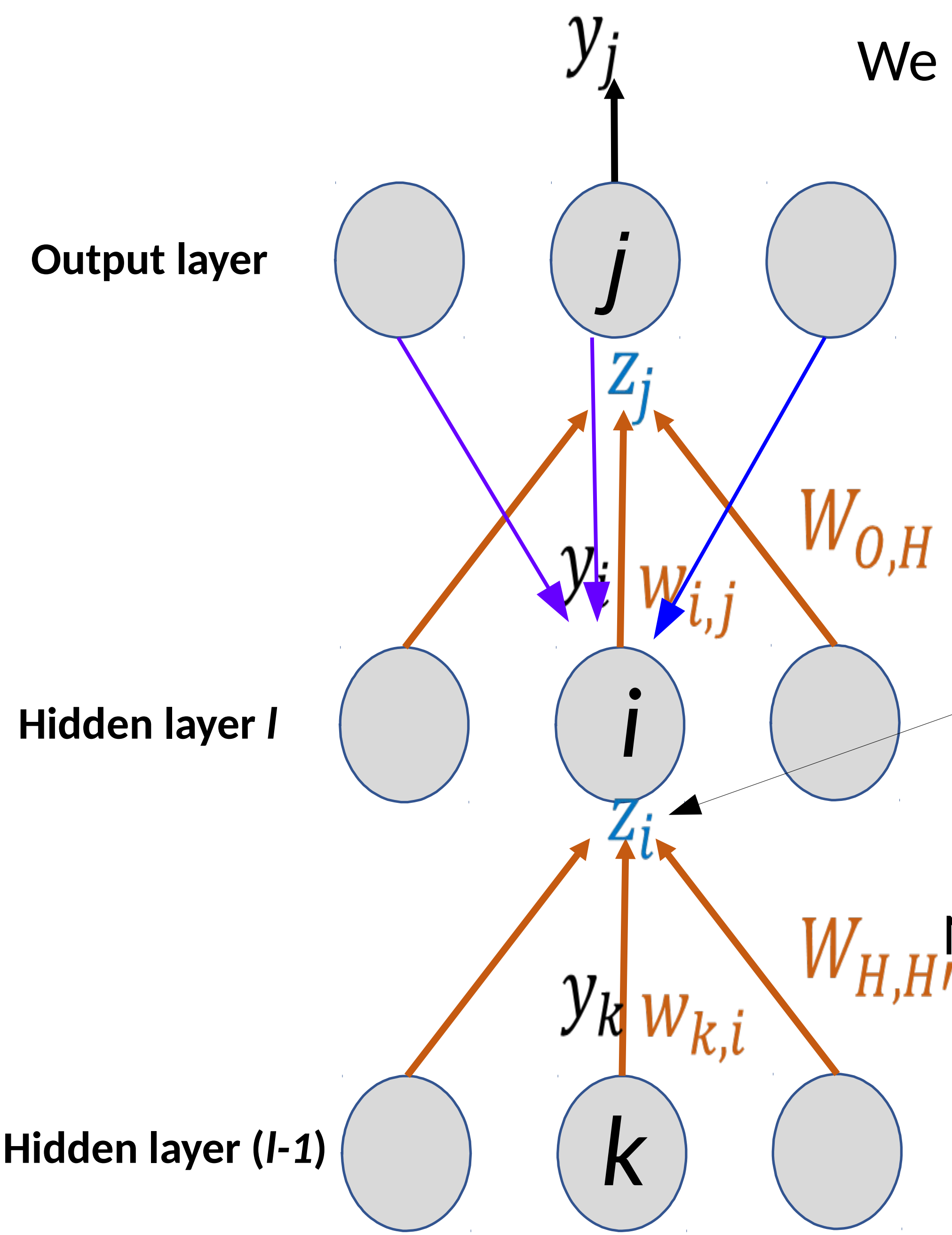
$$\frac{\partial E}{\partial w_{ki}} = y_k \frac{\partial E}{\partial z_i}$$

Note that now hidden node *i has no target*

$$\frac{\partial E}{\partial z_i} = \sum_{j \in outputs} y_i (1 - y_i) w_{ij} \boxed{\frac{\partial E}{\partial z_j}}$$

Previous workout

Backpropagation



We can repeat the process for earlier layers.

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial z_i}{\partial w_{ki}} \frac{\partial E}{\partial z_i}$$

Chain rule

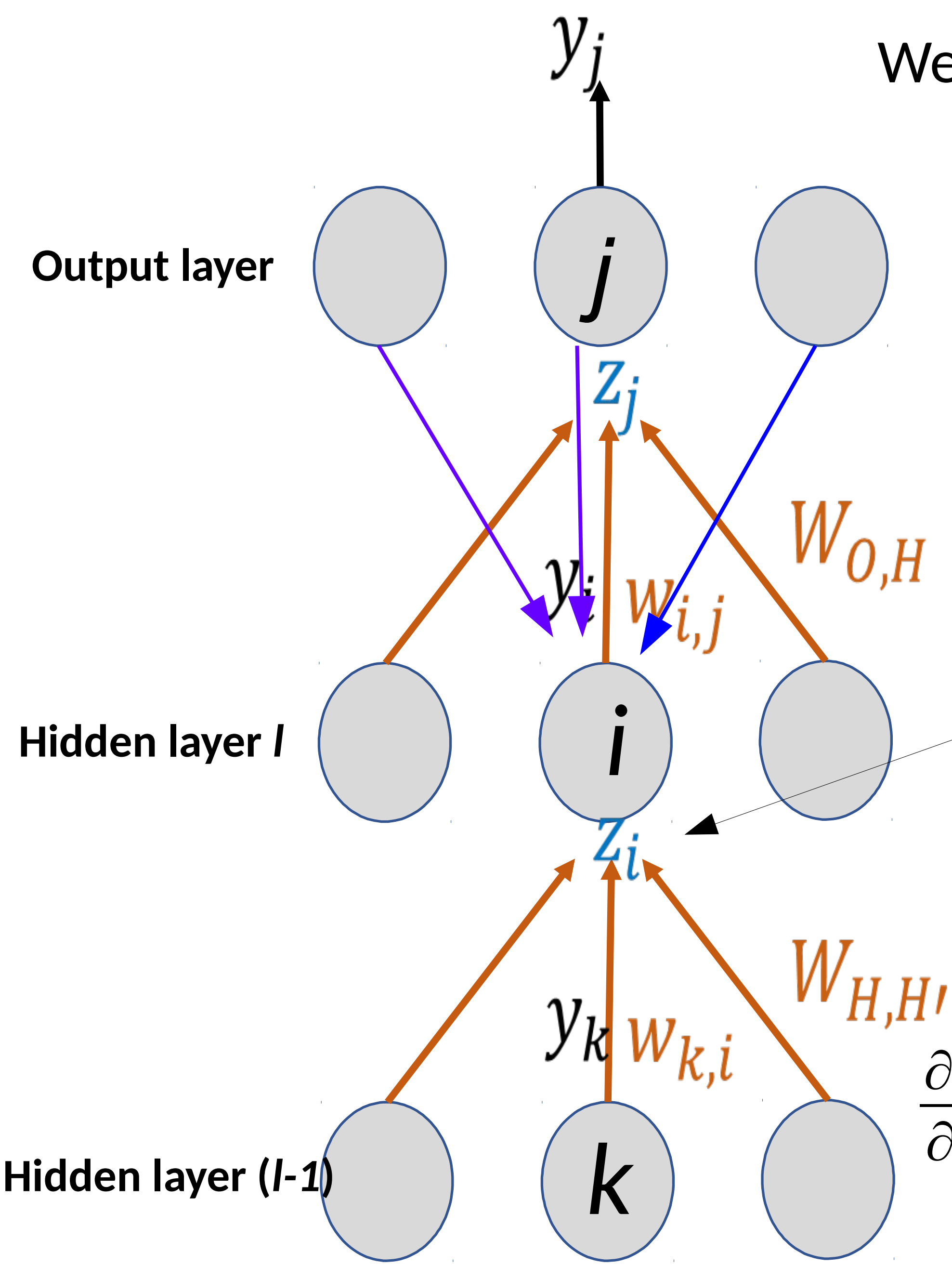
Note: $z_i = \sum_{k \in layer(l-1)} y_k w_{ki}$

$$\frac{\partial E}{\partial w_{ki}} = y_k \frac{\partial E}{\partial z_i}$$

Note that now hidden node *i* has no target

$$\frac{\partial E}{\partial z_i} = y_i(1 - y_i) \underbrace{\sum_{j \in outputs} w_{ij} \frac{\partial E}{\partial z_j}}_{\text{Weighted errors from later layers}}$$

Backpropagation



We can repeat the process for earlier layers.

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial z_i}{\partial w_{ki}} \frac{\partial E}{\partial z_i}$$

Chain rule

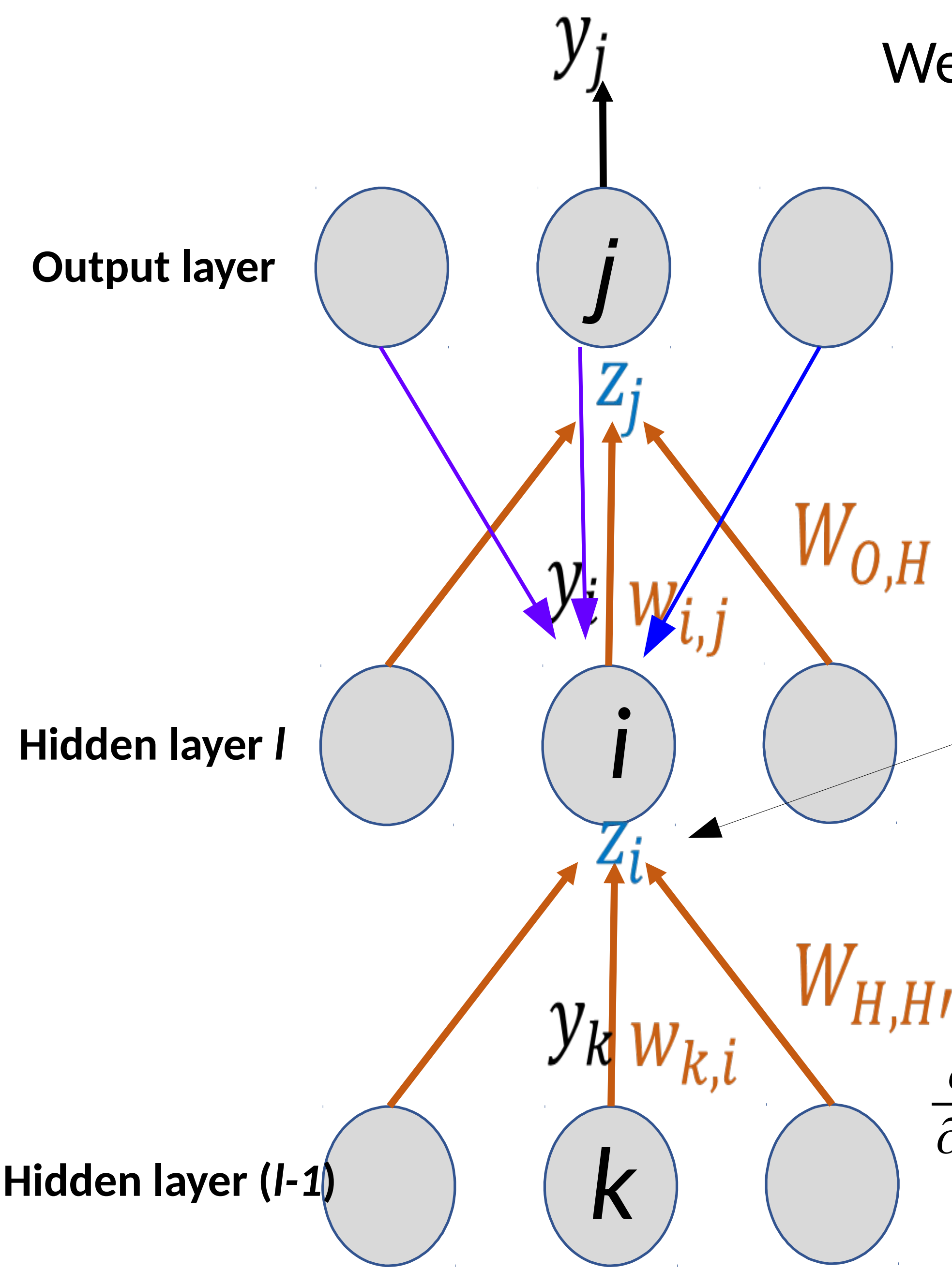
Note: $z_i = \sum_{k \in layer(l-1)} y_k w_{ki}$

$$\frac{\partial E}{\partial w_{ki}} = y_k \frac{\partial E}{\partial z_i}$$

Note that now hidden node *i has no target*

$$\frac{\partial E}{\partial z_i} = y_i(1-y_i) \underbrace{\sum_{j \in outputs} w_{ij} y_j (1-y_j)(-(t_j - y_j))}_{\text{Weighted errors are back-propagated}}$$

Backpropagation



We can repeat the process for earlier layers.

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial z_i}{\partial w_{ki}} \frac{\partial E}{\partial z_i}$$

Chain rule

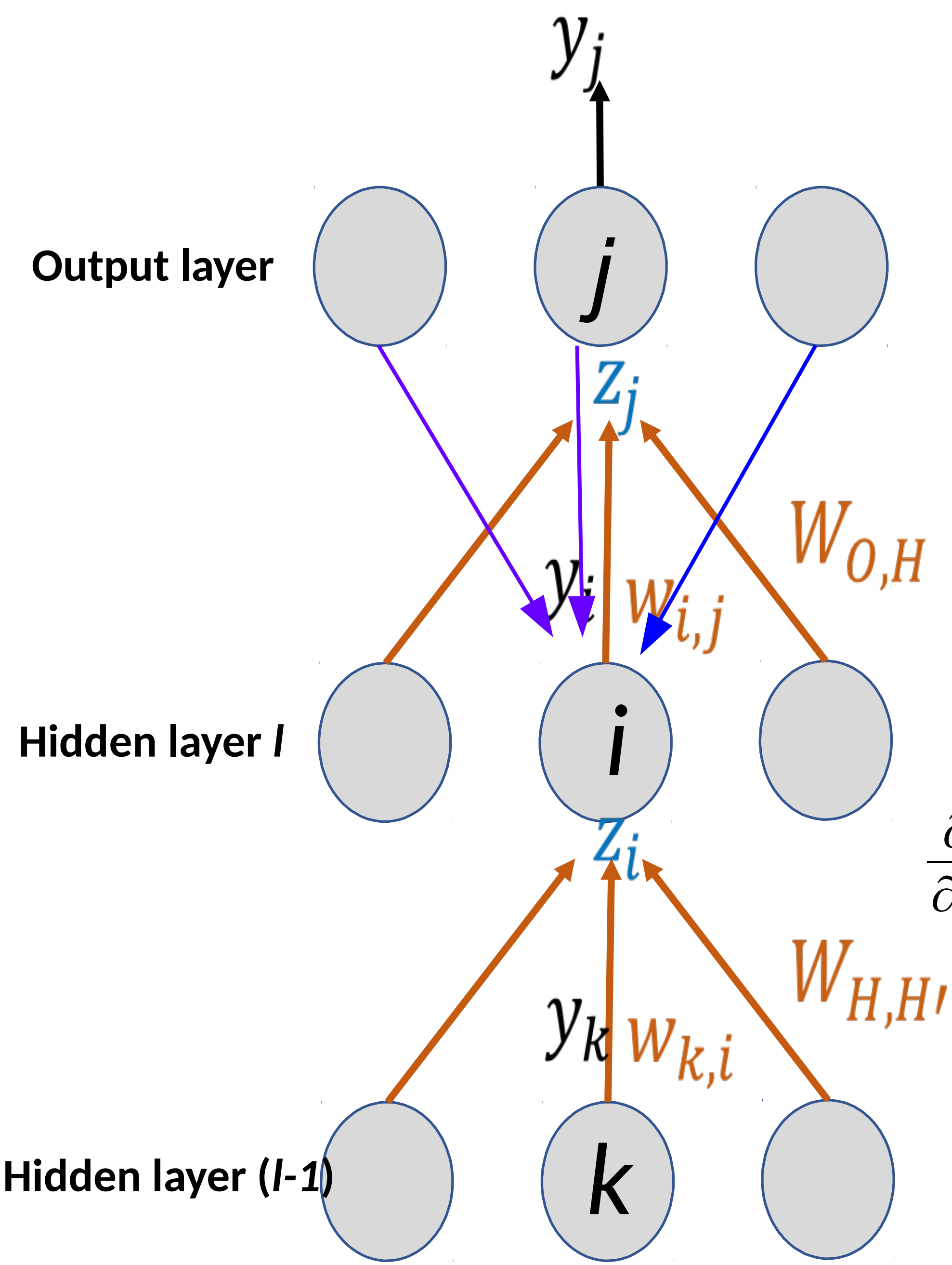
Note: $z_i = \sum_{k \in layer(l-1)} y_k w_{ki}$

$$\frac{\partial E}{\partial w_{ki}} = y_k \frac{\partial E}{\partial z_i}$$

Note that now hidden node *i has no target*

$$\frac{\partial E}{\partial w_{ki}} = y_k y_i (1 - y_i) \underbrace{\sum_{j \in outputs} w_{ij} y_j (1 - y_j)(-(t_j - y_j))}_{\text{Weighted errors are back-propagated}}$$

Backpropagation: Summary



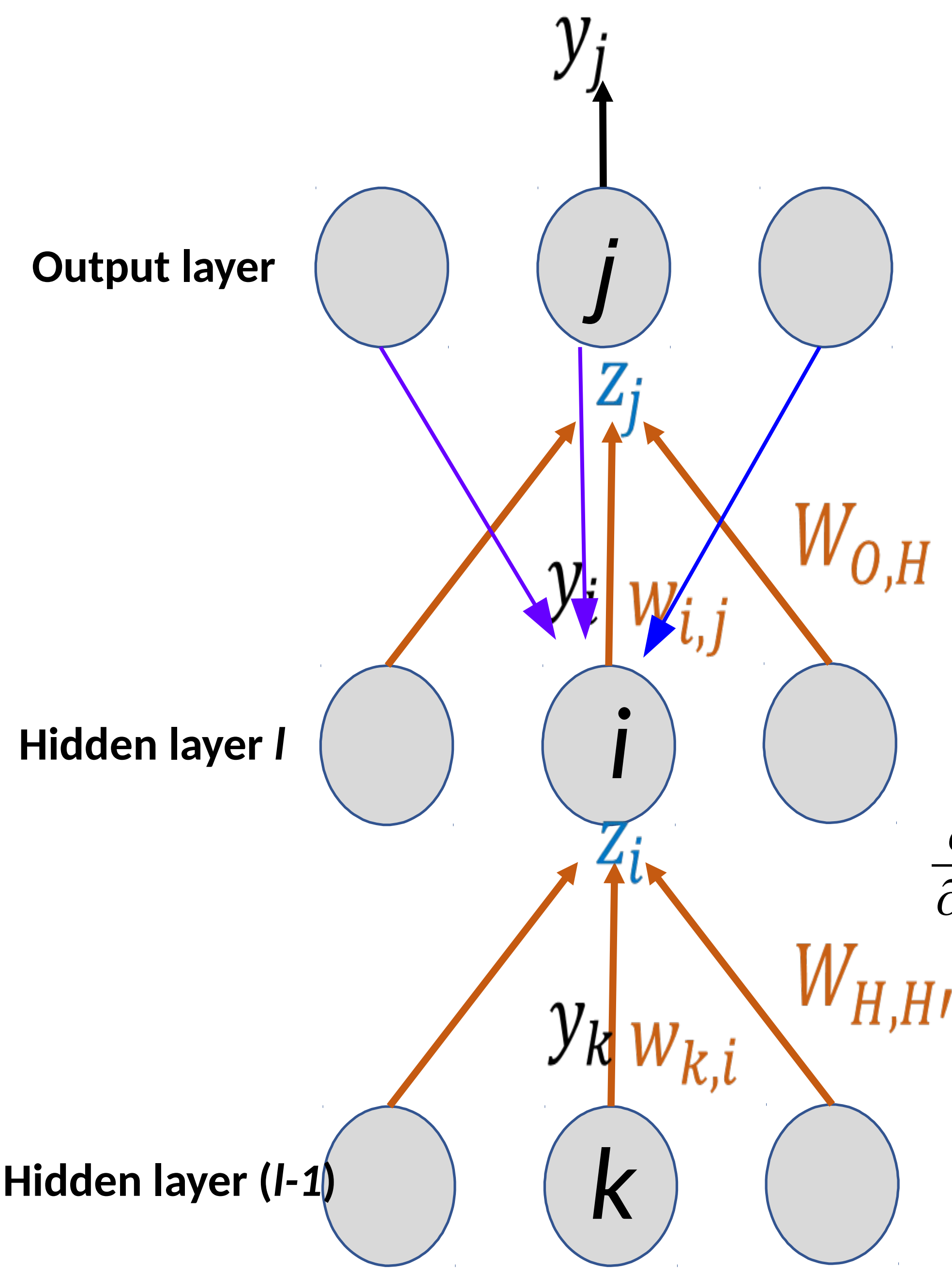
Weights of output nodes

$$\frac{\partial E}{\partial w_{ij}} = y_i \, y_j \left(1 - y_j\right) \underbrace{\left(-\left(t_j - y_j\right)\right)}_{\text{Error of this output node}}$$

Weights of hidden nodes

$$\frac{\partial E}{\partial w_{ki}} = y_k \, y_i \left(1 - y_i\right) \underbrace{\sum_{j \in outputs} w_{ij} \, y_j \left(1 - y_j\right)\left(-\left(t_j - y_j\right)\right)}_{\text{Weighted errors are back-propagated}}$$

Backpropagation: Summary



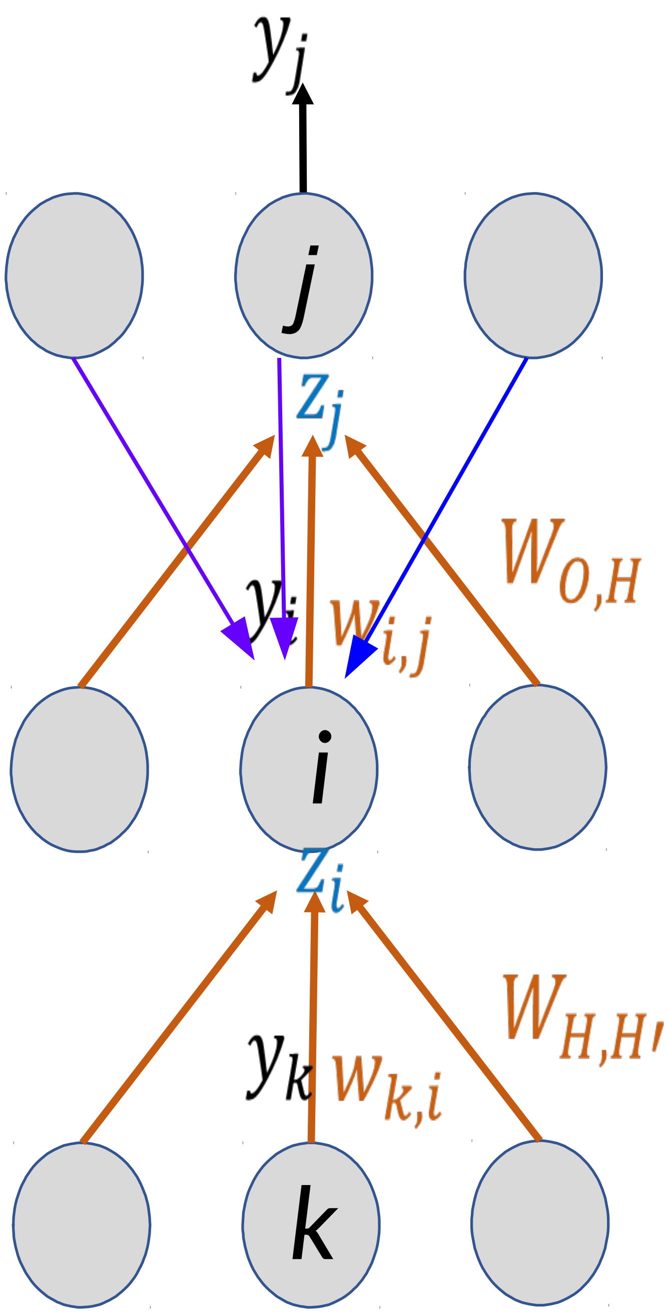
Weights of output nodes

$$\frac{\partial E}{\partial w_{ij}} = y_i \underbrace{y_j(1-y_j)(-(t_j-y_j))}_{\text{This error part will be back-propagated}}$$

Weights of hidden nodes

$$\frac{\partial E}{\partial w_{ki}} = y_k\, y_i(1-y_i) \sum_{j \in outputs} w_{ij} \underbrace{y_j(1-y_j)(-(t_j-y_j))}_{\text{These errors were back-propagated (computed previously)}}$$

Backpropagation: Summary



Weights of output nodes

$$\frac{\partial E}{\partial w_{ij}} = out_i \, error_j$$

Weights of hidden nodes

$$\frac{\partial E}{\partial w_{ki}} = out_k \, error_i = out_k \, gradientOfactivation_i \sum_j w_{ij} \, error_j$$

Updating the weights

$$w_{ij}^{(new)} = w_{ij}^{(old)} - \eta \frac{\partial E}{\partial w_{ij}} \quad \text{And} \quad w_{ki}^{(new)} = w_{ki}^{(old)} - \eta \frac{\partial E}{\partial w_{ki}}$$

We've worked this term out mathematically on the previous slides, so we can just plug the value in here.

- *Note that bigger derivatives (i.e. steeper slopes) means bigger changes to the weights.*

- *Typically, the weight update is calculated over many training items (a "batch")*

Highly recommended:
https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/
http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/

Updating the weights

$$w_{ij}^{(new)} = w_{ij}^{(old)} - \eta \frac{\partial E}{\partial w_{ij}}$$

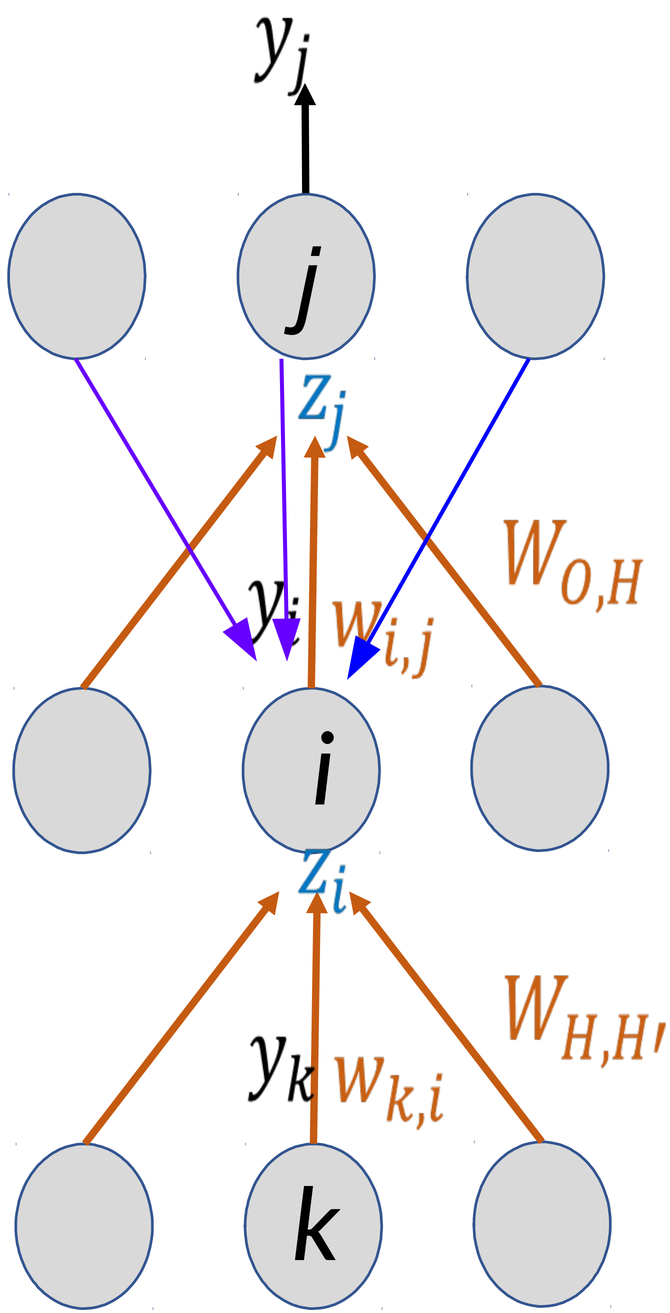
The learning rate (a hyperparameter). The bigger the learning rate, the greater the change to the weights.

Updating the weights

$$w_{ij}^{(new)} = w_{ij}^{(old)} - \eta \frac{\partial E}{\partial w_{ij}}$$

*Minus sign: we move the weights in the direction **opposite** the gradient (i.e. the direction that reduces rather than increases the error)*

Backpropagation: Algorithm



1. First apply the inputs to the network and work out the output: e.g. y_k, y_i, y_j.

- Initial output could be anything, as initial weights are random.

2. Calculate the error for neurons j (at output layer):

$$error_j = y_j(1-y_j)(-(t_j-y_j))$$

3. Calculate the error for hidden neurons I

$$error_i = y_i(1-y_i)\sum_{j\in outputs} w_{ij}\, error_j$$

4. Update the weights:

$$w_{ij}^{(new)} = w_{ij}^{(old)} - \eta\, y_i\, error_j$$

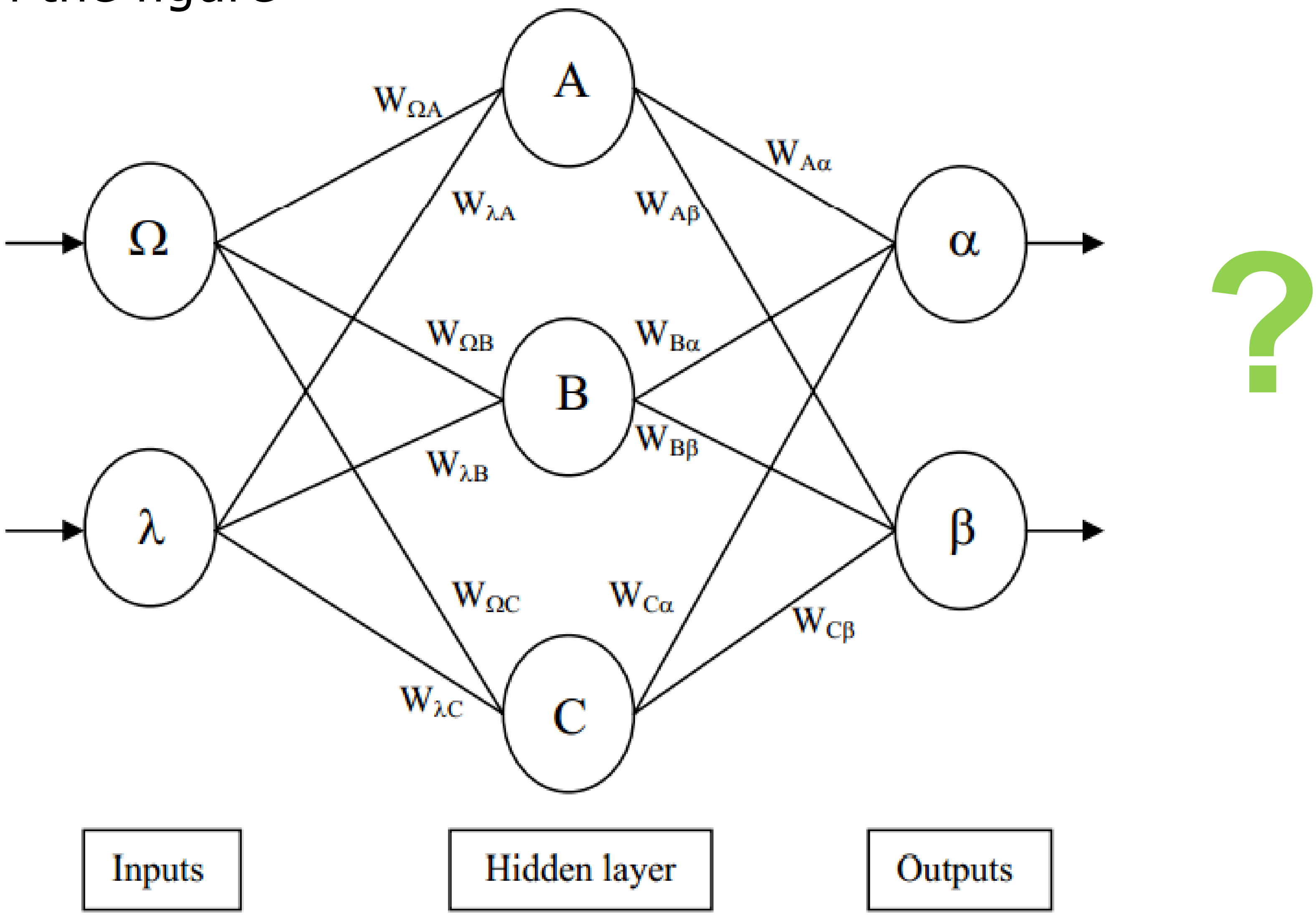
$$w_{ki}^{(new)} = w_{ki}^{(old)} - \eta\, y_k\, error_i$$

5. Repeat until convergence!

Assume: all activations are sigmoid

Backpropagation: Example 1

- Obtain all the new weights expression for a full sized network with 2 inputs, 3 hidden layer neurons and 2 output neurons as shown in the figure



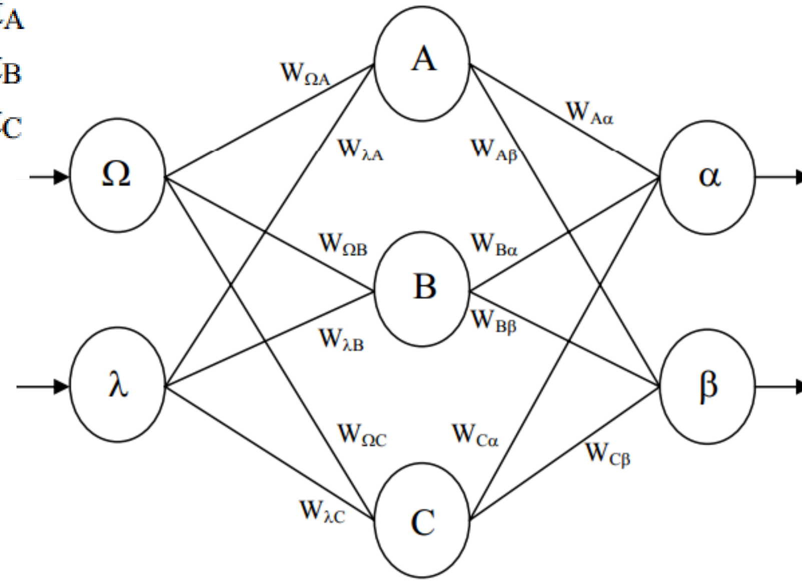
Exercise 1: Solution

1. Calculate errors of output neurons

$$\delta_\alpha = out_\alpha\,(1 - out_\alpha)\,(Target_\alpha - out_\alpha)$$
$$\delta_\beta = out_\beta\,(1 - out_\beta)\,(Target_\beta - out_\beta)$$

2. Change output layer weights

$$W^+_{A\alpha} = W_{A\alpha} - \eta\delta_\alpha\, out_A \qquad W^+_{A\beta} = W_{A\beta} - \eta\delta_\beta\, out_A$$
$$W^+_{B\alpha} = W_{B\alpha} - \eta\delta_\alpha\, out_B \qquad W^+_{B\beta} = W_{B\beta} - \eta\delta_\beta\, out_B$$
$$W^+_{C\alpha} = W_{C\alpha} - \eta\delta_\alpha\, out_C \qquad W^+_{C\beta} = W_{C\beta} - \eta\delta_\beta\, out_C$$



3. Calculate (back-propagate) hidden layer errors

$$\delta_A = out_A\,(1 - out_A)\,(\delta_\alpha W_{A\alpha} + \delta_\beta W_{A\beta})$$
$$\delta_B = out_B\,(1 - out_B)\,(\delta_\alpha W_{B\alpha} + \delta_\beta W_{B\beta})$$
$$\delta_C = out_C\,(1 - out_C)\,(\delta_\alpha W_{C\alpha} + \delta_\beta W_{C\beta})$$

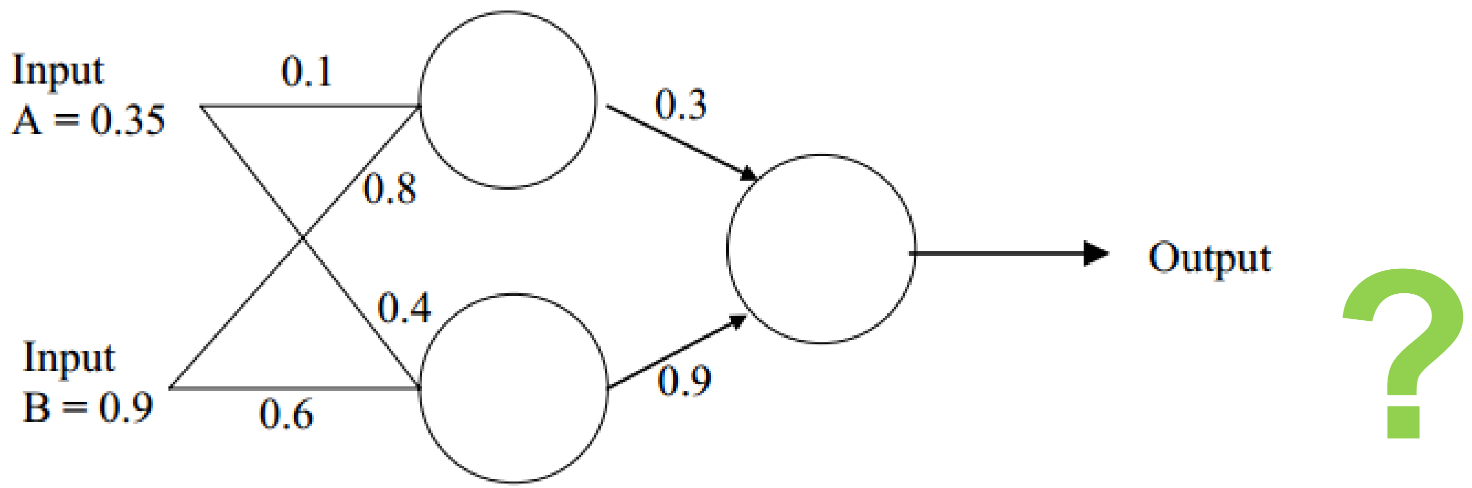
4. Change hidden layer weights

$$W^+_{\lambda A} = W_{\lambda A} - \eta\delta_A\, in_\lambda \qquad W^+_{\Omega A} = W^+_{\Omega A} - \eta\delta_A\, in_\Omega$$
$$W^+_{\lambda B} = W_{\lambda B} - \eta\delta_B\, in_\lambda \qquad W^+_{\Omega B} = W^+_{\Omega B} - \eta\delta_B\, in_\Omega$$
$$W^+_{\lambda C} = W_{\lambda C} - \eta\delta_C\, in_\lambda \qquad W^+_{\Omega C} = W^+_{\Omega C} - \eta\delta_C\, in_\Omega$$

- W+ represents the new, recalculated, weight, whereas W (without the superscript) represents the old weight.
- The constant η is the learning rate

Backpropagation: Example 2

- Consider the simple network below:



Assume that the neurons have a Sigmoid activation function (bias=0) and

- ✓ Perform a forward pass on the network.
- ✓ Perform a backpropagation pass (training) once (desired output= 0.5, learning rate = 1).
- ✓ Perform a further forward pass and comment on the result.

Example 2: Solution

i. Perform a forward pass on the network using equations:

Bias: $c = 0$

$$input = \boldsymbol{w} \cdot \boldsymbol{x} + c$$
$$output = \sigma(\boldsymbol{w} \cdot \boldsymbol{x} + c)$$

$\text{Input}_{\text{Top}}$ = 0.35*0.1+0.9*0.8 = 0.76

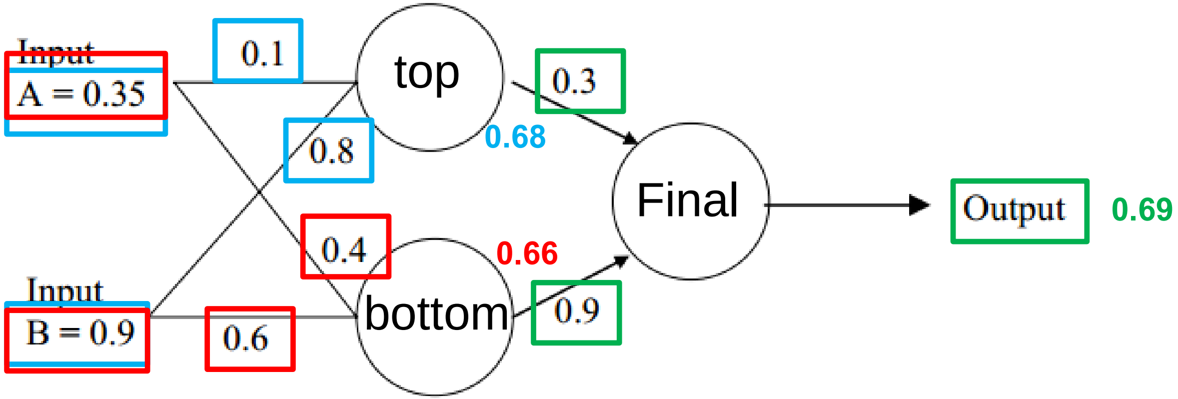
$\text{Output}_{\text{Top}} = \sigma(0.76) = \frac{1}{1+e^{-0.76}} = 0.68$

$\text{Input}_{\text{Bottom}}$ = 0.35*0.4+0.9*0.6 = 0.68

$\text{Output}_{\text{Bottom}} = \sigma(0.68) = \frac{1}{1+e^{-0.68}} = 0.66$

$\text{Input}_{\text{Final}}$ = 0.3*0.68+0.9*0.66 = 0.80

$\text{Output}_{\text{Final}} = \sigma(0.80) = \frac{1}{1+e^{-0.80}} = 0.69$



Example 2: Solution

ii. Perform a backpropagation pass (training) once (desired output= 0.5, learning rate = 1).

a. Output error

$\text{ERROR}_F = -\text{OUTPUT}_F * (1-\text{OUTPUT}_F) * (\text{DESIRED\_OUTPUT} - \text{OUTPUT}_F) = -0.69 * (1-0.69) * (0.5 - 0.69) = 0.041$

b. New weights for output layer

$\text{NEW\_WEIGHT}_{TF} = \text{WEIGHT}_{TF} - \text{ERROR}_F * \text{OUTPUT}_T * \text{LEARNING\_RATE} = 0.3 - 0.041 * 0.68 * 1 = 0.27$

$\text{NEW\_WEIGHT}_{BF} = \text{WEIGHT}_{BF} - \text{ERROR}_F * \text{OUTPUT}_B * \text{LEARNING\_RATE} = 0.9 - 0.041 * 0.66 * 1 = 0.87$

TF: top-final

BF: bottom-final

c. Errors for hidden layers:

$\text{ERROR}_T = \text{OUTPUT}_T * (1-\text{OUTPUT}_T) * (\text{WEIGHT}_{TF} * \text{ERROR}_F) = 0.68 * (1-0.68) * (0.27 * (0.041)) = 0.0024$

$\text{ERROR}_B = \text{OUTPUT}_B * (1-\text{OUTPUT}_B) * (\text{WEIGHT}_{BF} * \text{ERROR}_F) = 0.66 * (1-0.66) * (0.87 * (0.041)) = 0.008$

d. New hidden layer weights:

$\text{NEW\_WEIGHT}_{AT} = \text{WEIGHT}_{AT} - \text{ERROR}_T * \text{INPUT}_A * \text{LEARNING\_RATE} = 0.1 - 0.0024 * 0.35 * 1 = 0.099$

$\text{NEW\_WEIGHT}_{BT} = \text{WEIGHT}_{BT} - \text{ERROR}_T * \text{INPUT}_B * \text{LEARNING\_RATE} = 0.8 - 0.0024 * 0.9 * 1 = 0.798$

$\text{NEW\_WEIGHT}_{AB} = \text{WEIGHT}_{AB} - \text{ERROR}_B * \text{INPUT}_A * \text{LEARNING\_RATE} = 0.4 - 0.008 * 0.35 * 1 = 0.397$

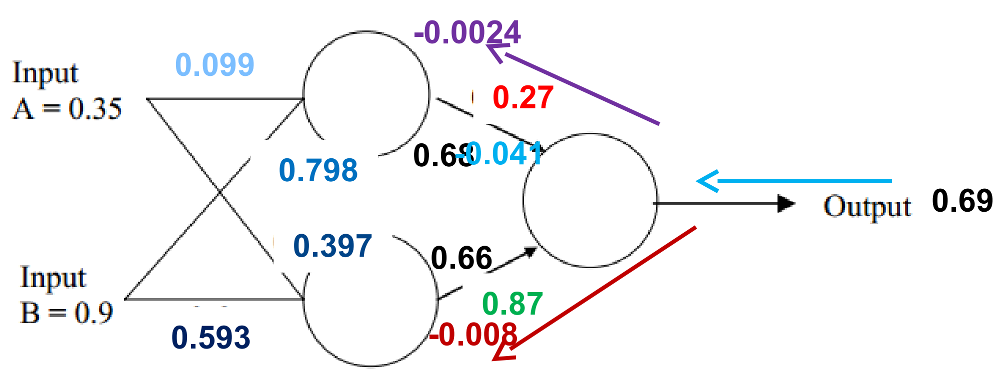
$\text{NEW\_WEIGHT}_{BB} = \text{WEIGHT}_{BB} - \text{ERROR}_B * \text{INPUT}_B * \text{LEARNING\_RATE} = 0.6 - 0.008 * 0.9 * 1 = 0.593$

AT: A-top

AB: A-bottom

BT: B-top

BB: B-bottom



Example 2: Solution

iii. Perform a further forward pass and comment on the result.

$Input_{Top}$ = 0.35*0.099+0.9*0.798 = 0.75

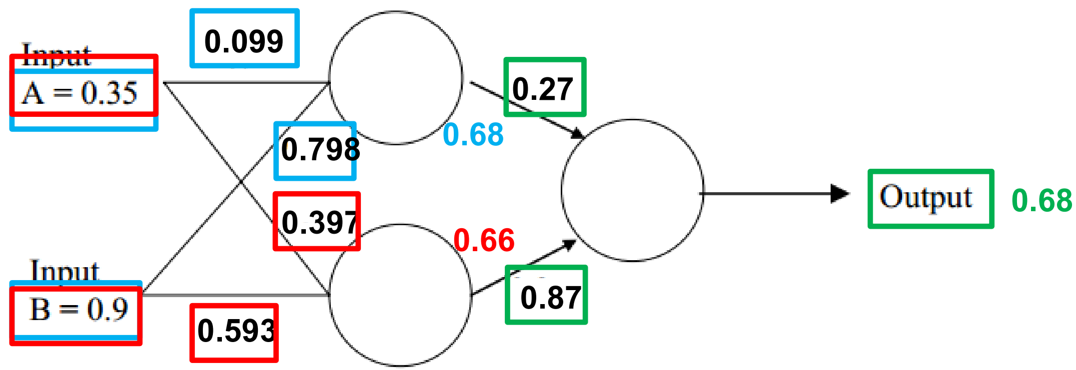
$Output_{Top} = \sigma(0.75) = \frac{1}{1+e^{-0.75}} = 0.68$

$Input_{Bottom}$ = 0.35*0.397+0.9*0.593 = 0.67

$Output_{Bottom} = \sigma(0.67) = \frac{1}{1+e^{-0.67}} = 0.66$

$Input_{Final}$ = 0.27*0.68+0.87*0.66 = 0.76

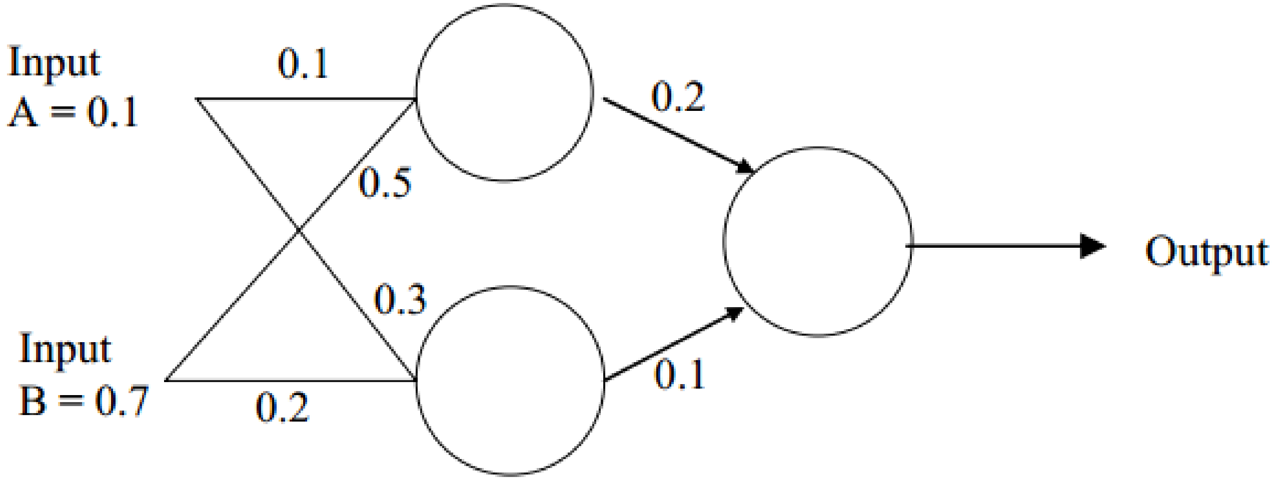
$Output_{Final} = \sigma(0.76) = \frac{1}{1+e^{-0.76}} = 0.68$



Guess output has reduced from 0.69 to 0.68, that is closer to the desired output (0.5), so the learning is working

Example 3

- Consider the simple network below:



- Assume that the neurons have a Sigmoid activation function and
  i. Perform a forward pass on the network.
  ii. Perform a reverse pass (training) once (desired output= 1, learning rate = 2).
  iii. Perform a further forward pass and comment on the result.

Example 3: Solution

i. Perform a forward pass on the network using equations:

$$input = \boldsymbol{w} \cdot \boldsymbol{x} + c$$
$$output = \sigma(\boldsymbol{w} \cdot \boldsymbol{x} + c)$$

$Input_{Top}$ = 0.1*0.1+0.7*0.5 = 0.36

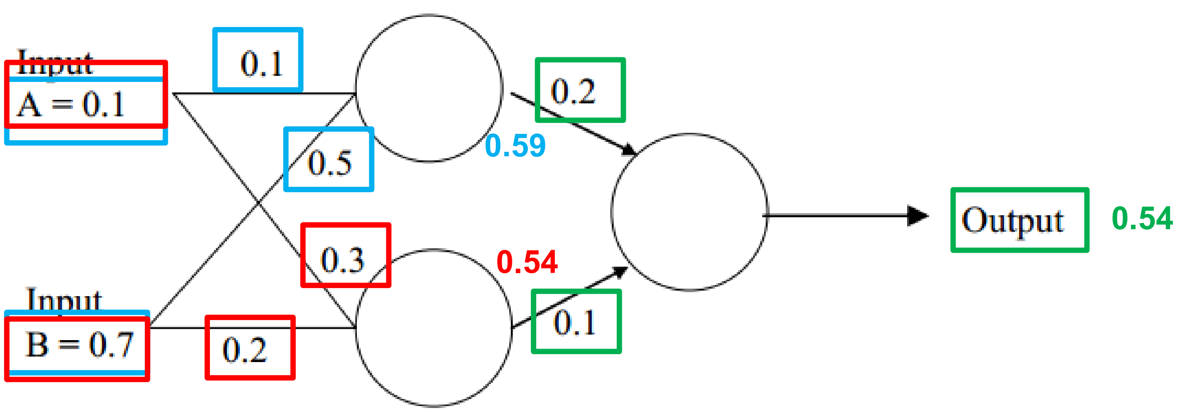
$Output_{Top} = \sigma(0.36) = \frac{1}{1+e^{-0.36}} = 0.59$

$Input_{Bottom}$ = 0.1*0.3+0.7*0.2 = 0.17

$Output_{Bottom} = \sigma(0.17) = \frac{1}{1+e^{-0.17}} = 0.54$

$Input_{Final}$ = 0.2*0.59+0.1*0.54 = 0.17

$Output_{Final} = \sigma(0.17) = \frac{1}{1+e^{-0.17}} = 0.54$



Example 3: Solution

ii. Perform a reverse pass (training) once (desired output= 1, learning rate = 2).

a. Output error

$ERROR_F$ = - $OUTPUT_F$ * (1- $OUTPUT_F$) * (DESIRED_OUTPUT - $OUTPUT_F$) = -0.54* (1- 0.54) * (1 – 0.54)= -0.11

b. New weights for output layer

NEW_WEIGHT_{TF} = $WEIGHT_{TF}$ - $ERROR_F$ * $OUTPUT_T$* LEARNING_RATE = 0.2 - (-0.11) * 0.59* 2 = 0.33

NEW_WEIGHT_{BF} = $WEIGHT_{BF}$ - $ERROR_F$ * $OUTPUT_B$* LEARNING_RATE = 0.1 - (-0.11) * 0.54* 2 = 0.22

c. Errors for hidden layers:

$ERROR_T$ = $OUTPUT_T$ * (1- $OUTPUT_T$) * ($WEIGHT_{TF}$ *$ERROR_F$) = 0.59 * (1- 0.59) * (0.33 *(-0.11)) =- 0.009

$ERROR_B$ = $OUTPUT_B$ * (1- $OUTPUT_B$) * ($WEIGHT_{BF}$ *$ERROR_F$) = 0.54 * (1- 0.54) * (0.22 *(-0.11)) = -0.006

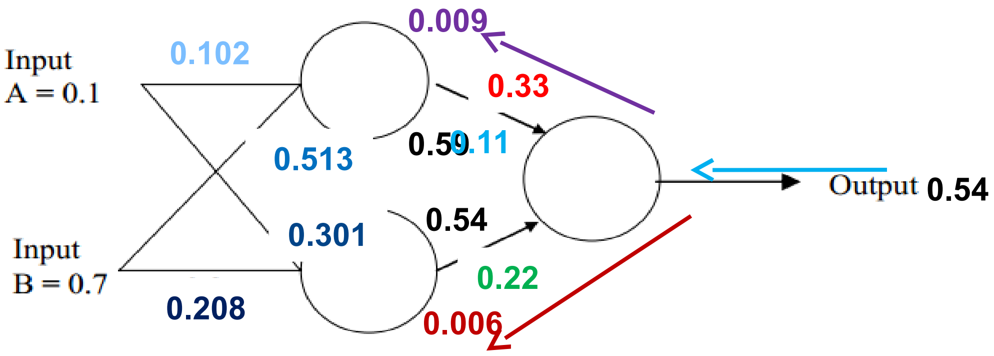
d. New hidden layer weights:

NEW_WEIGHT_{AT} = $WEIGHT_{AT}$ - $ERROR_T$ * $INPUT_A$* LEARNING_RATE = 0.1 - (-0.009) * 0.1 * 2 = 0.102

NEW_WEIGHT_{BT} = $WEIGHT_{BT}$ - $ERROR_T$ * $INPUT_B$* LEARNING_RATE = 0.5 - (-0.009) * 0.7 * 2 = 0.513

NEW_WEIGHT_{AB} = $WEIGHT_{AB}$ - $ERROR_B$ * $INPUT_A$* LEARNING_RATE = 0.3 - (-0.006) * 0.1 * 2 = 0.301

NEW_WEIGHT_{BB} = $WEIGHT_{BB}$ - $ERROR_B$ * $INPUT_B$* LEARNING_RATE = 0.2 - (-0.006) * 0.7 * 2 = 0.208



Example 3: Solution

iii. Perform a further forward pass and comment on the result.

$Input_{Top}$ = 0.1*0.102+0.7*0.513 = 0.37

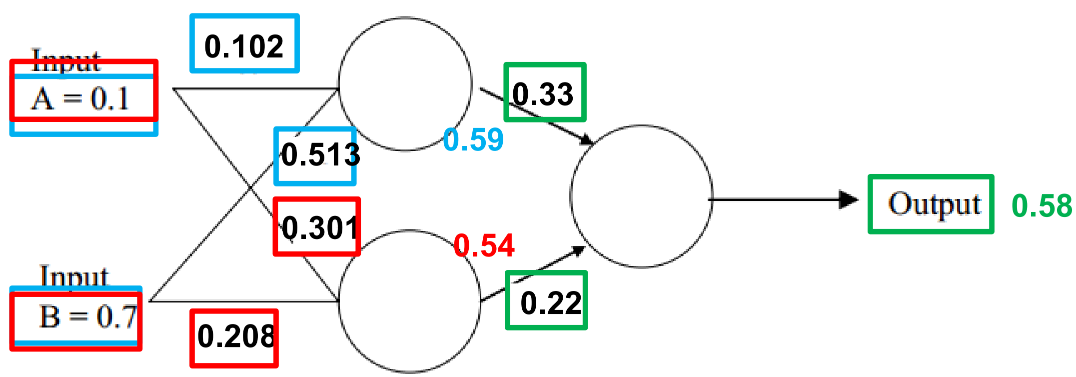
$Output_{Top} = \sigma(0.37) = \frac{1}{1+e^{-0.37}}$=0.59

$Input_{Bottom}$ = 0.1*0.301+0.7*0.208 = 0.17

$Output_{Bottom} = \sigma(0.17) = \frac{1}{1+e^{-0.17}}$=0.54

$Input_{Final}$ = 0.33*0.59+0.22*0.54 = 0.31

$Output_{Final} = \sigma(0.31) = \frac{1}{1+e^{-0.31}}$=0.58



Guess output has increased from 0.54 to 0.58, that is closer to the desired output (1), so the learning is working

Optimization issues in using the weight derivatives

- How often to update the weights
  - Online: after each training instance.
  - Full batch: after a full sweep through the training data (computing averaged gradients over all training instances).
  - Mini-batch: after a small sample of training cases (e.g. learning for big-data problems).
- How much to update
  - Use a fixed learning rate?
  - Adapt the global learning rate?
  - Adapt the learning rate on each connection separately?
  - Don’t use steepest descent?

Backpropagation: Summary

- An algorithm is an efficient way of computing the error derivative for every weight on a single training case.
- Backpropagation is commonly used by the gradient descent optimization algorithm to adjust the weights of the network.
- It works very well when training a big network on big data if combined with other techniques.