Deep Learning

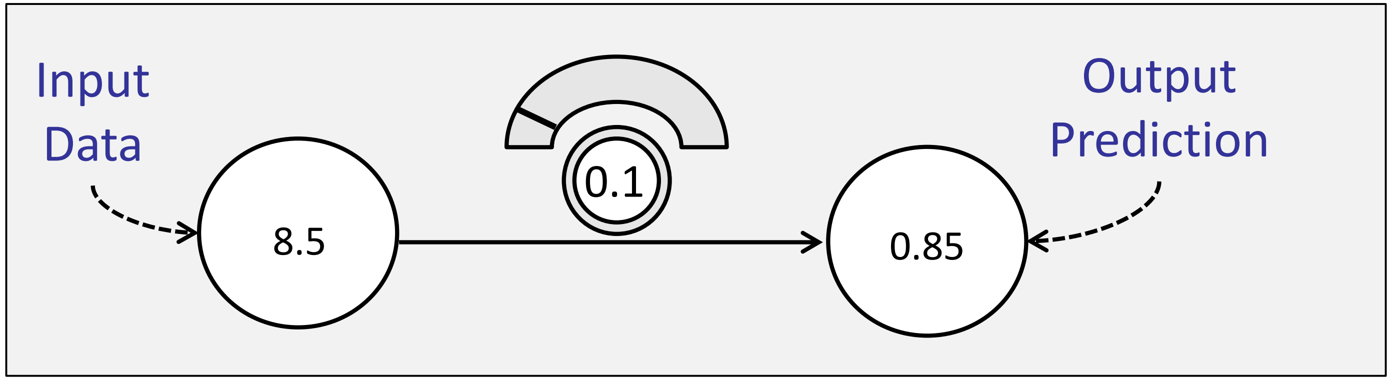
Feed Forward Neural Networks

Tuesday 5th November

Dr. Nicholas Cummins

Lecture and figures based on: Trask, A. *Deep learning*. New York: Manning Publications, 2018.

- **Neural Networks**
  - **Weights represent knowledge**
    - A measure of sensitivity between the input and the prediction
    - It uses *knowledge* captured in the weights to *interpret* the input data to *predict* a certain outcome




```
weight = 0.1
def neural_network(input, weight):
    prediction = input * weight
    return prediction
number_of_offsides = [8.5, 9.5, 10, 9]
input = number_of_offsides[0]
pred = neural_network(input,weight)
print(pred)
```

- **Updating weights to make predictions more accurate**
  - **Compare using a loss function**
    - Evaluate how well the network performed

```
error = ((input * weight)- goal_pred) ** 2
```

  - **Learn weights via Gradient Descent**
    - Adjusting each weight to reduce the error
    - Using the derivate of `weight` and `error` relationship defined through the loss function to adjust the weights

```
weight = weight - (alpha*derivative)
```

Recap

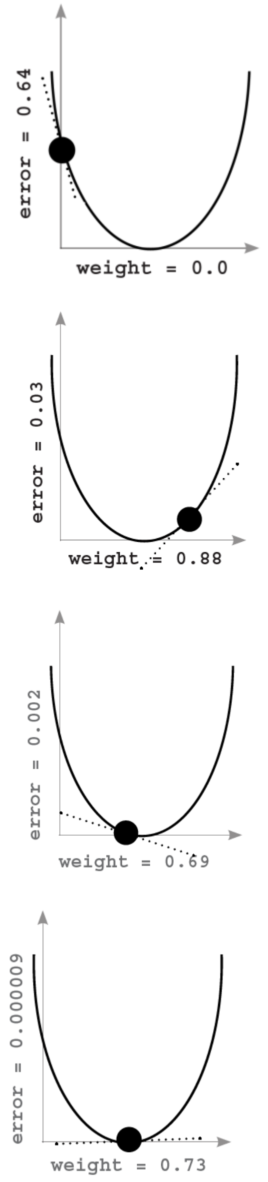
- **Gradient Descent**
  - Learning is adjusting the weight to reduce the `error` to 0
    - The function conforms to the patterns in the data

```
error = (pred - goal_pred) ** 2
```

  - Derivative of the error functions defines the amount that error changes when you change weight

```
derivative = input * (pred - goal_pred)
```

  - How can we use the derivative to minimise the error?
    - Adjust the weights in the opposite direction of the derivative
      1. Calculate the derivative of weight with respect to error
      2. Change weight in the opposite direction of that slope.



Recap

- **Gradient Descent for a single network**

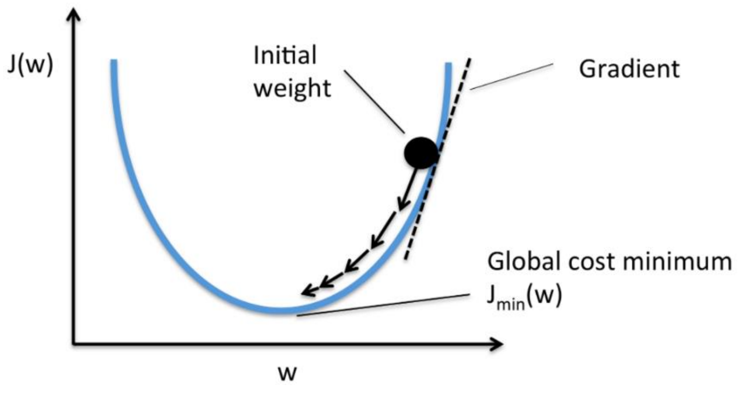
```
pred = input * weight
error = (pred - goal_pred) ** 2
derivative = input * (pred - goal_pred)
weight = weight - (alpha * derivative)
```

  - `error` is a measure of how much the network missed by
    - We define `error` to be always positive
  - `derivative` is the derivate of `weight` and `error`
    - Predicts both direction and amount to adjust the weights
  - `Alpha` scales the weight update
    - Helps minimise divergence effects when the input is large

Today's Lecture

Image source: https://medium.com

- Gradient Descent
  - Networks with multiple inputs
  - Networks with multiple outputs
  - Networks with multiple inputs and outputs
- Correlation
  - Learning Correlation
  - Creating Correlation
- Backpropagation
- Summary



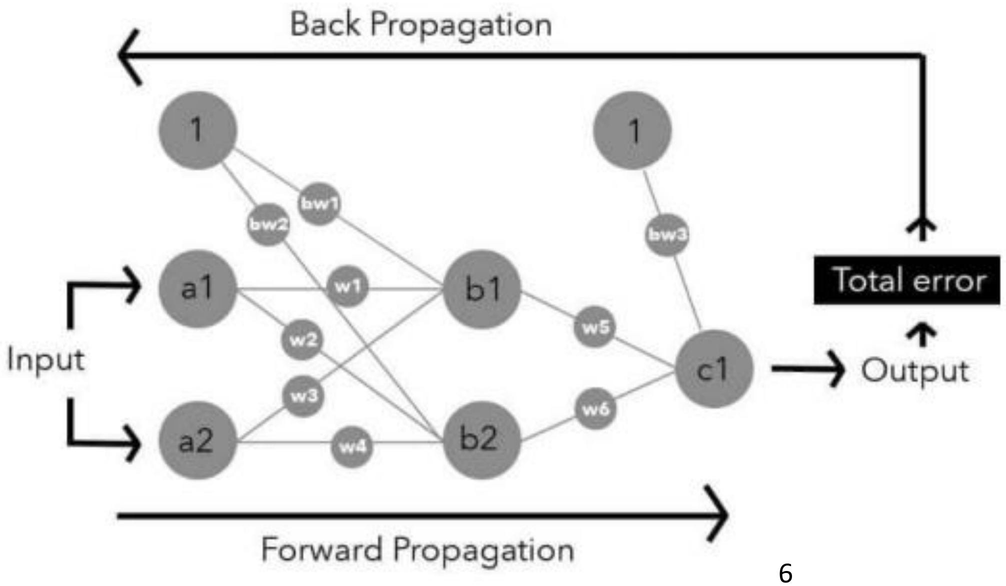
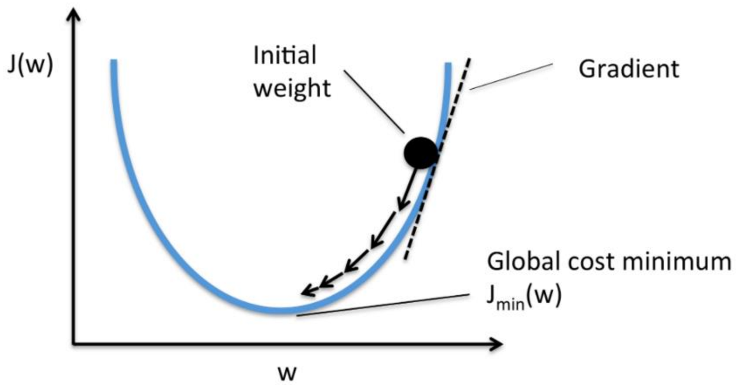
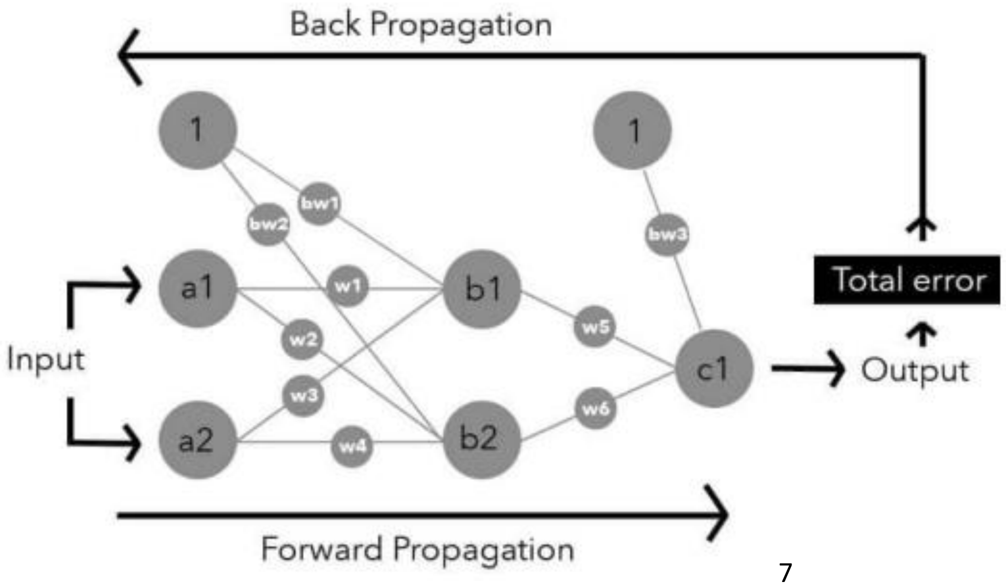


Image source: https://medium.com

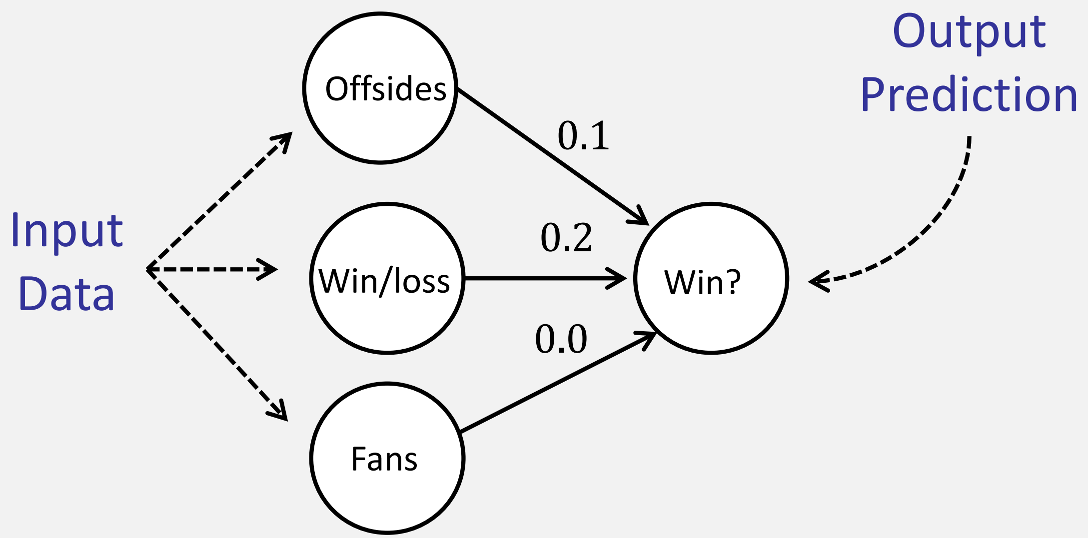
- Gradient Descent
  - Networks with multiple inputs
  - Networks with multiple outputs
  - Networks with multiple inputs and outputs
- Correlation
  - Learning Correlation
  - Creating Correlation
- Backpropagation
- Summary





- **Neural networks can combine intelligence from multiple data points**
  - This allows the network to combine various forms of information to make better-informed decisions

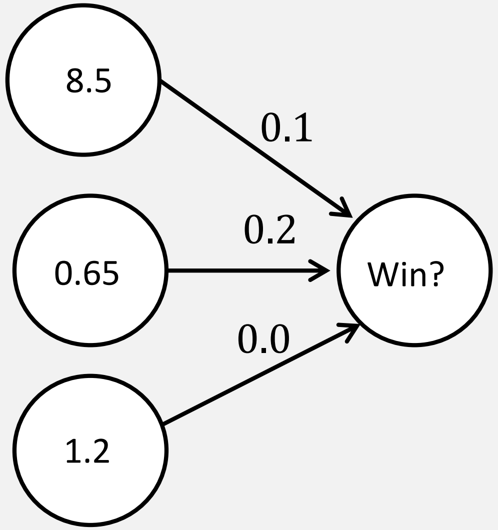
Empty network with multiple inputs




```
weight = [0.1, 02, 0.0]
def neural_network(input, weight):
    pred = w_sum(input * weight)
    return pred
```

- **Multiple each input by its *own* weight**
  - Fundamental weighting mechanism has not changed
  - New property is that there are multiple weights

Inserting a vector

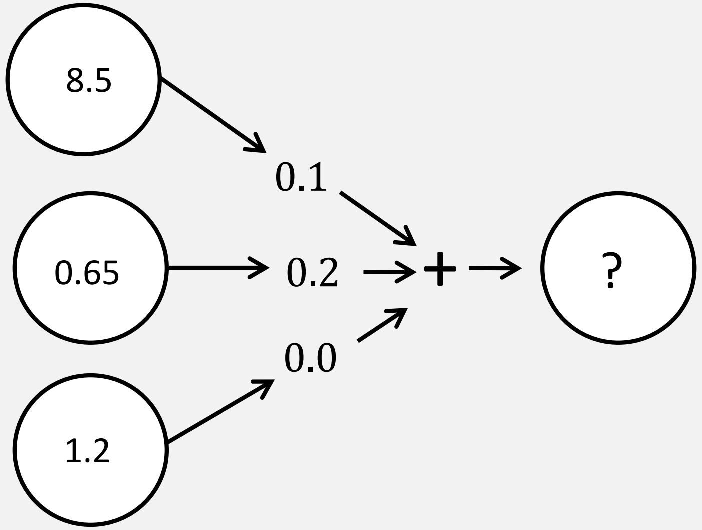



```python
def w_sum(a,b):
    assert(len(a) == len(b))
    output = 0
    for i in range(len(a)):
        output += (a[i] * b[i])
    return output
```

Prediction with multiple inputs

- **Output is a weighted sum**
  - Output gained by summing the respective predictions
    - Weighted sum of the input, achieved via the dot product

Performing weighted sum

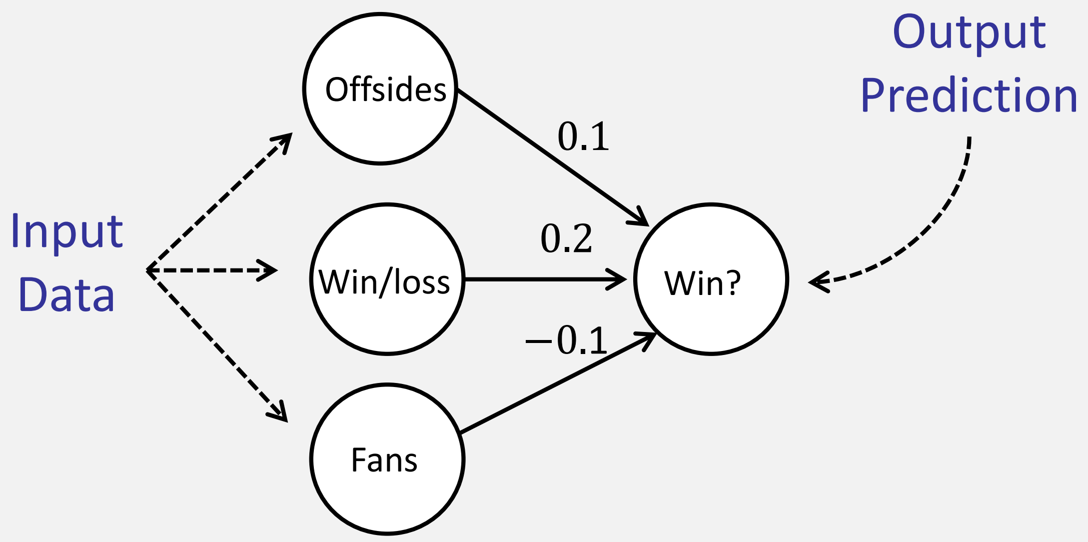


Inputs		Weights		Local Predictions		
(8.5	*	0.1)	=	0.85	=	Offside prediction
(0.65	*	0.2)	=	0.13	=	Win/loss prediction
(1.20	*	0.0)	=	0.00	=	Number of fans prediction

Offside prediction	+	Win/loss prediction	+	Number of fans prediction	=	Final Prediction
0.85	+	0.13	+	0.0	=	0.98

- **Gradient descent also works with multiple inputs**
  - The same techniques for updating single networks can be used to update a network containing multiple weights.

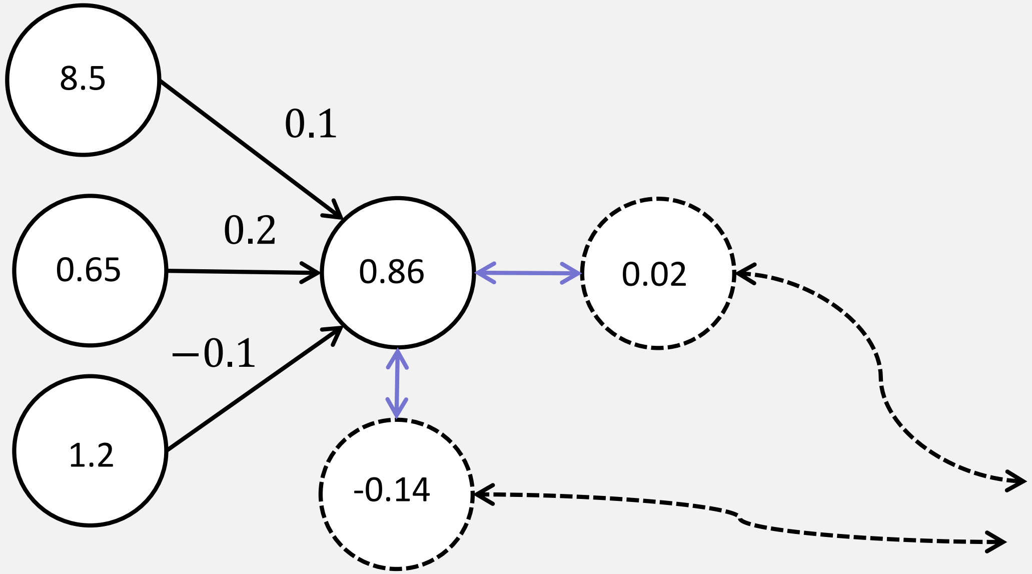
Empty network with multiple inputs




```
def w_sum(a,b):
    assert(len(a) == len(b))
    output = 0
    for i in range(len(a)):
        output += (a[i] * b[i])
    return output

weights = [0.1, 0.2, -0.1]
def neural_network(input, weight):
    pred = w_sum(input * weight)
    return pred
```

- **Basic compare steps have not changed**
  - Make a prediction, and calculate error and delta

Output the prediction, calculate error & delta

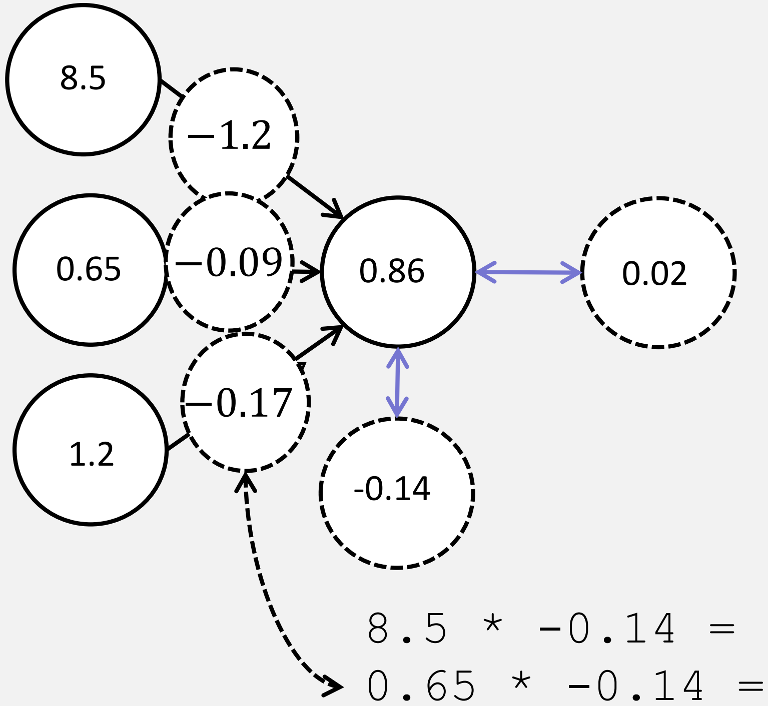



```
offsides = [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]
win_or_lose_binary = [1, 1, 0, 1]

true = win_or_lose_binary[0]

input = [offsides[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)

error = (pred - true) ** 2
delta = pred - true
```

Gradient descent with multiple inputs

- **Basic learn steps have not changed**
  - Calculate a `weight_delta` and update each weight

Output the prediction, calculate error & delta



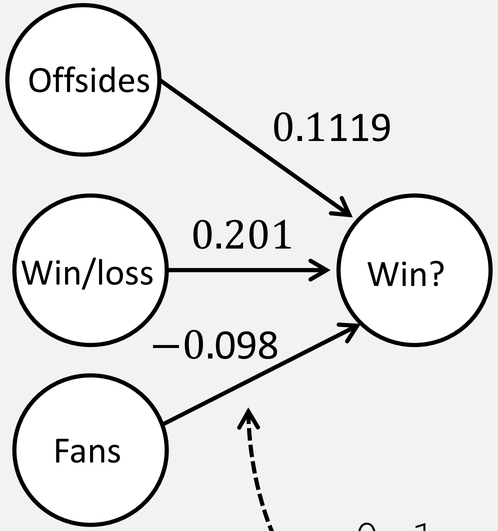

```
def ele_mul(number,vector):
    output = [0,0,0]
    assert(len(output) == len(vector))
    for i in range(len(vector)):
        output[i] = number * vector[i]
    return output
pred = neural_network(input,weight)
error = (pred - true) ** 2
delta = pred - true
weight_deltas = ele_mul(delta,input)
```

```
8.5 * -0.14 = -1.19 = weight_deltas[0]
0.65 * -0.14 = -0.091 = weight_deltas[1]
1.2 * -0.14 = -0.168 = weight_deltas[2]
```

Gradient descent with multiple inputs

- **Basic learn steps have not changed**
  - Last step, update the weights

Update the weights




```
input = [offsides[0],wlrec[0],nfans[0]]
pred = neural_network(input,weight)
error = (pred - true) ** 2
delta = pred - true
weight_deltas = ele_mul(delta,input)

alpha = 0.01
for i in range(len(weights)):
    weights[i] -= alpha * weight_deltas[i]
```

```
0.1 - (-1.19 * 0.01) = 0.1119 = weights[0]
0.2 - (-.091 * 0.01) = 0.2009 = weights[1]
-0.1 - (-.168 * 0.01) = -0.098 = weights[2]
```

Gradient descent with multiple inputs

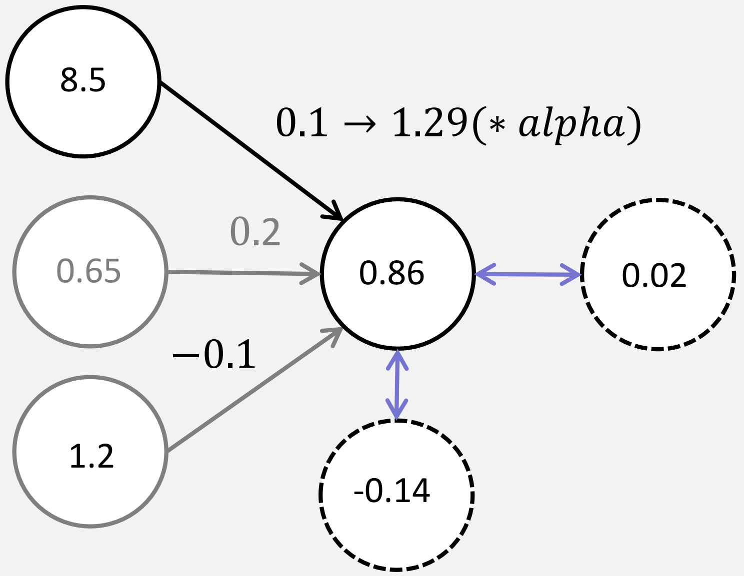
- **Single `delta` to multiple `weight_delta` variables**
  - Reminder of their roles and purposes
  - **`delta`**: A measure of how much higher or lower a node's output value should be
    - Computed by a direct subtraction between the node's predicted value and the node's true value
  - **`weighted_delta`**: A derivative-based estimate of the direction and amount we need to move a `weight` by
    - Computed by multiplying `delta` by a weight's `input`
    - Accounts for effects of scaling, negative reversal, and stopping

Gradient descent with multiple inputs

- **Single `delta` to multiple `weight_delta` variables**
  - Reminder of their roles and purposes
  - Consider a single weight update

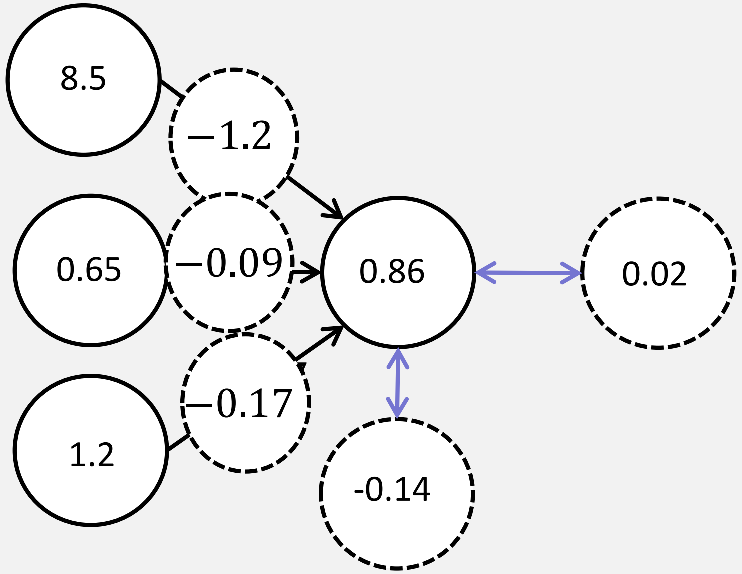
delta: Hey, inputs. Next time, predict a little higher.
Single weight:

- **Stopping** *if* my input was 0, then my weight wouldn't have mattered, and I wouldn't change a thing
- **Negative Reversal:** *if* my input was negative, then I'd want to decrease my weight instead of increase it
- **Scaling:** *my current input* is positive and quite large, so my personal prediction was important I'm going to move my weight up a lot to compensate



Gradient descent with multiple inputs

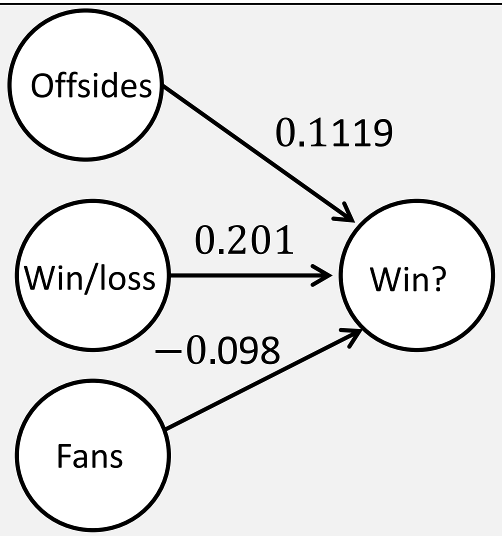
- **Single `delta` to multiple `weight_delta` variables**
  - Each weight has a unique `input` but a common `delta`
  - Each respective `weight_delta` is calculated by multiplying the unique input multiplied by delta




```
8.5 * -0.14 = -1.19 = weight_deltas[0]
0.65 * -0.14 = -0.091 = weight_deltas[1]
1.2 * -0.14 = -0.168 = weight_deltas[2]
```

Gradient descent with multiple inputs

- **Multiple weight updates**
  - Identical procedure to the single-input network,
    - Except performed over multiple weights
  - Once the individual `weight_delta` values have been calculated, multiply by `alpha` and subtract from the `weight`




```
0.1 - (-1.19 * 0.01) = 0.1119 = weights[0]
0.2 - (-.091 * 0.01) = 0.2009 = weights[1]
-0.1 - (-.168 * 0.01) = -0.098 = weights[2]
```

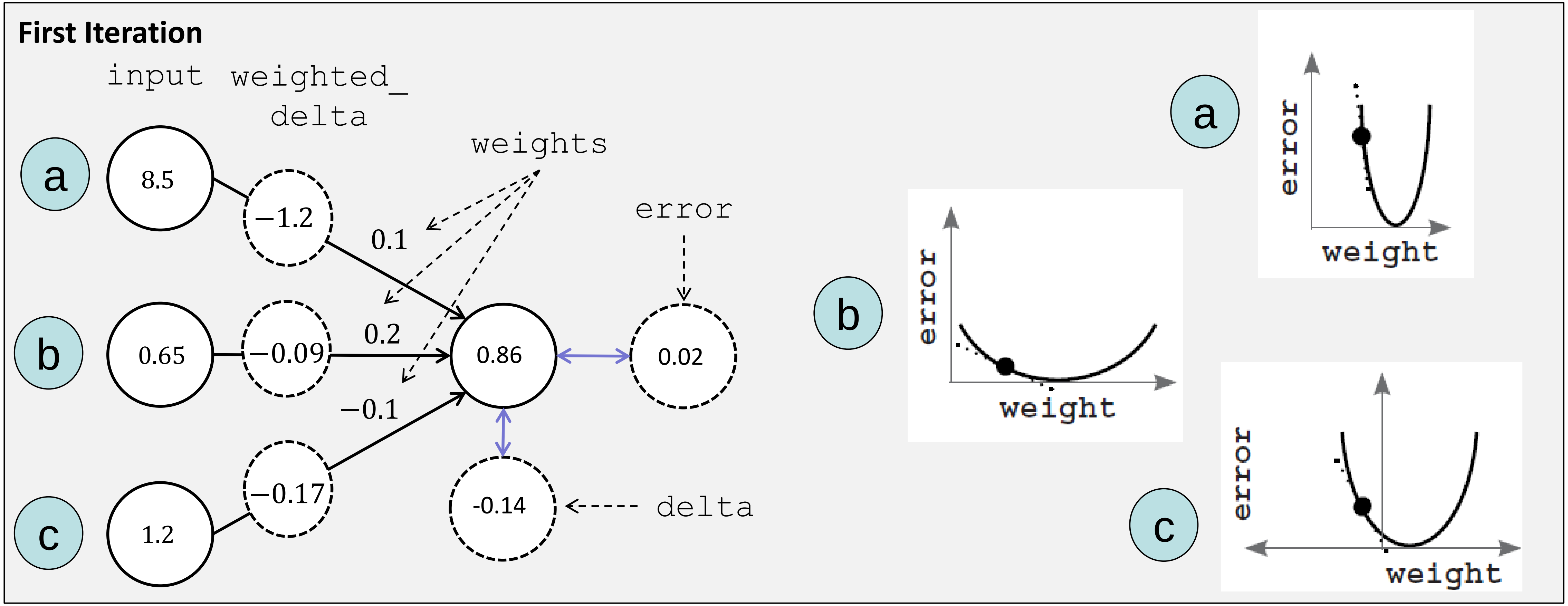
Gradient descent with multiple inputs

- **Multiple iterations of learning**

```
def neural_network(input, weights):
    out = 0
    for i in range(len(input)):
        out += (input[i] * weights[i])
    return out

def ele_mul(scalar, vector):
    out = [0,0,0]
    for i in range(len(out)):
        out[i] = vector[i] * scalar
    return out

offsides = [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]
win_or_lose_binary = [1, 1, 0, 1]
```

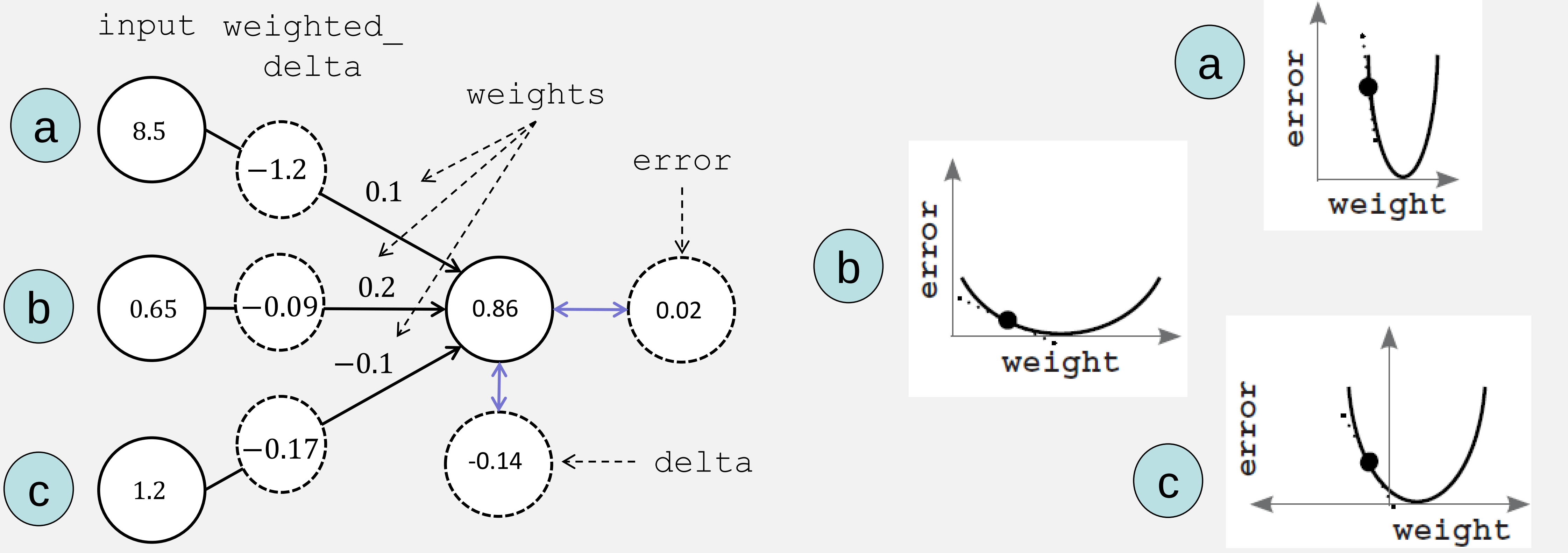
```
true = win_or_lose_binary[0]
alpha = 0.01
weights = [0.1, 0.2, -.1]
input = [offsides[0],wlrec[0],nfans[0]]

for iter in range(3):
    pred = neural_network(input,weights)
    error = (pred - true) ** 2
    delta = pred - true
    weight_deltas=ele_mul(delta,input)
    for i in range(len(weights)):
        weights[i]-=alpha*weight_deltas[i]
```

- **Multiple iterations of learning**

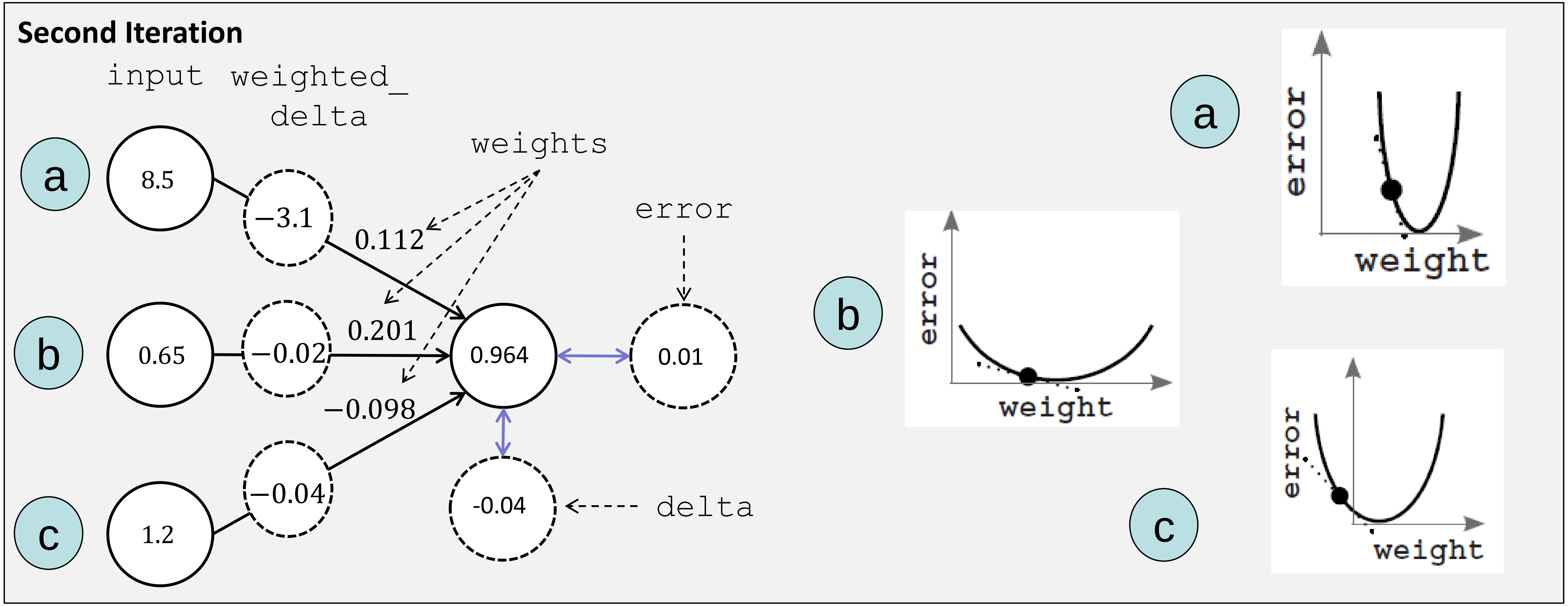


- **Multiple iterations of learning**

First Iteration



- **Multiple iterations of learning**



- **Multiple iterations of learning**

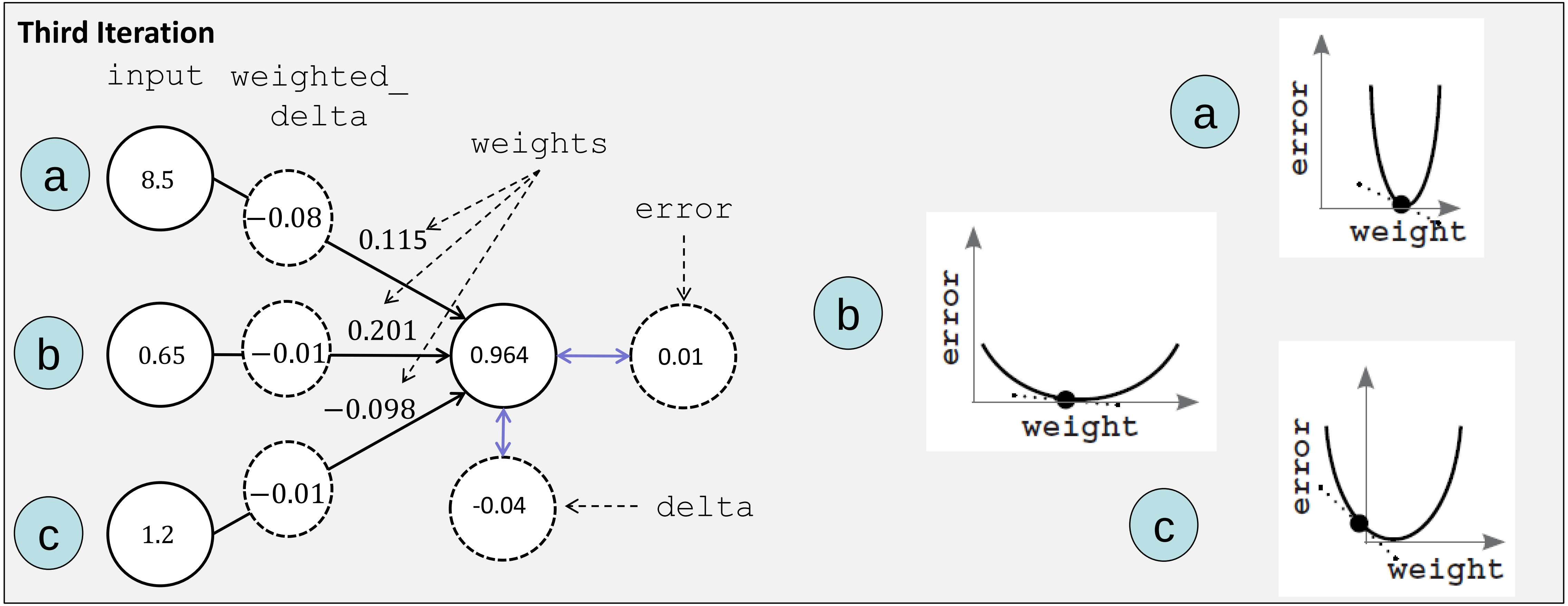
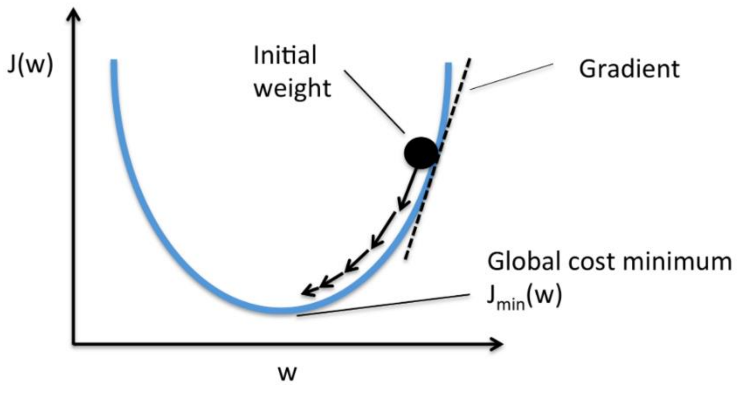
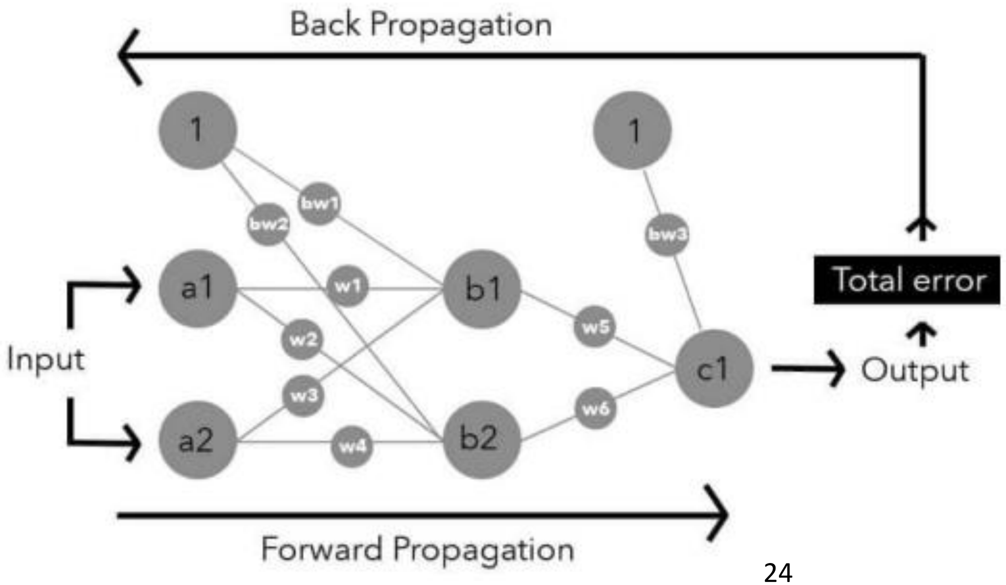


Image source: https://medium.com

- Gradient Descent
  - Networks with multiple inputs
  - Networks with multiple outputs
  - Networks with multiple inputs and outputs
- Correlation
  - Learning Correlation
  - Creating Correlation
- Backpropagation
- Summary

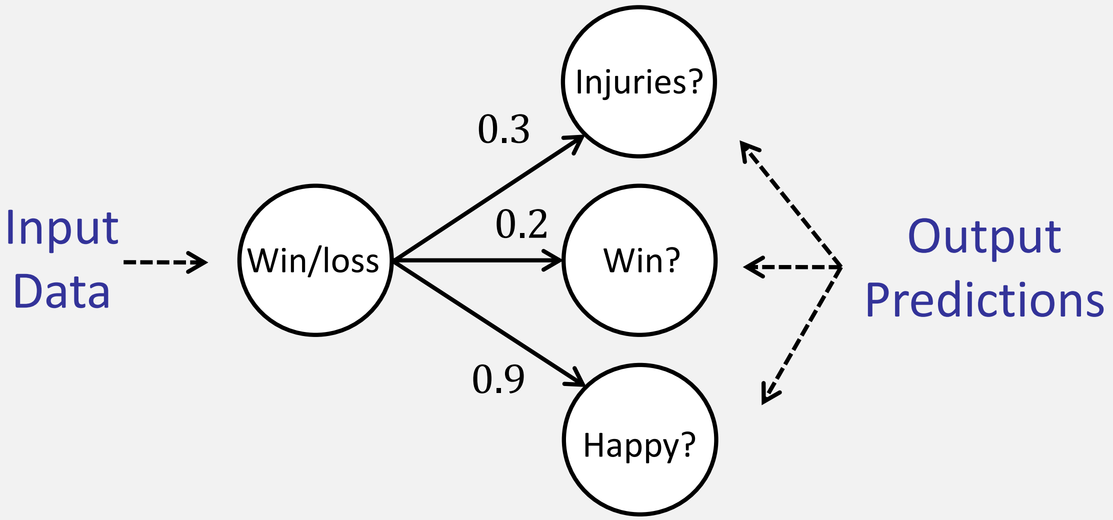




Universität Augsburg University

- **Single input → multiple predictions**
  - Prediction occurs the same as if there were multiple disconnected single-weight neural networks

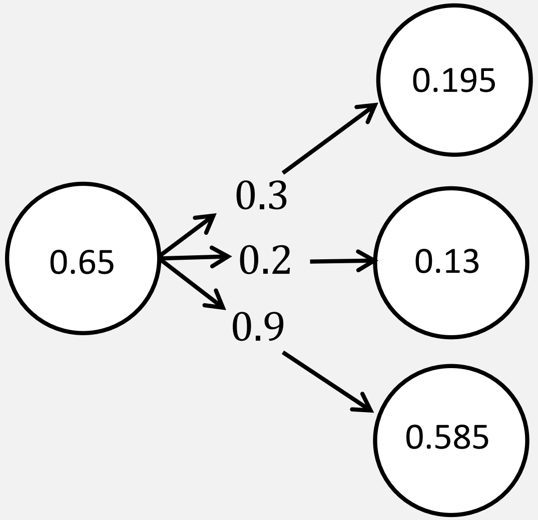
Empty network with multiple outputs




```
weight = [0.3, 0.2, 0.9]
def neural_network(input, weight):
    pred = ele_mul(input * weight)
    return pred
```

- **Independence**
  - Network behaves as three independent components, each receiving the same input data

Elementwise multiplication

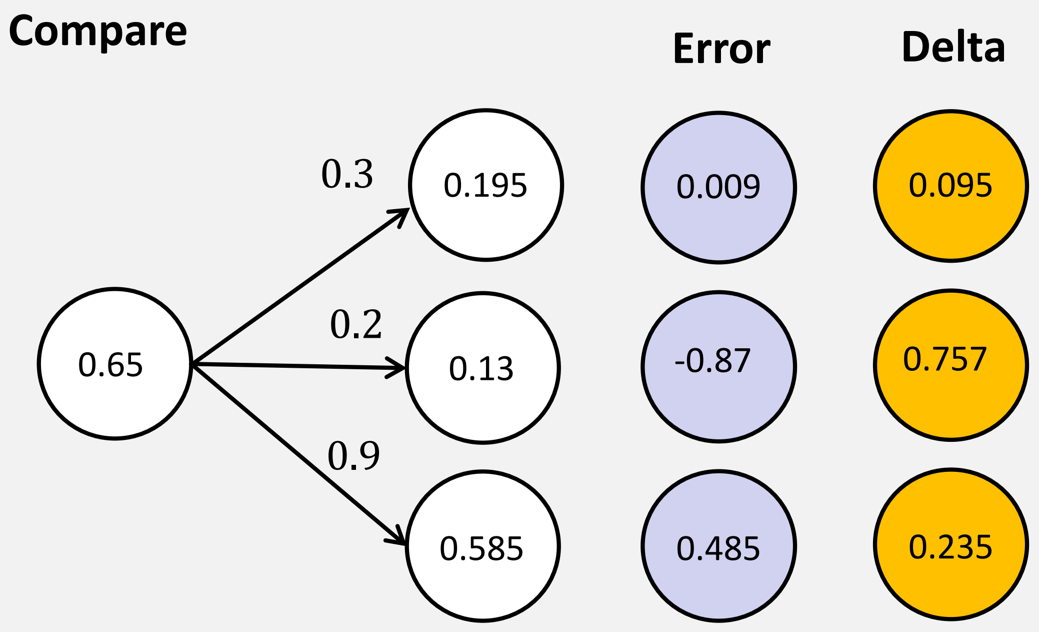



```
def ele_mul(number,vector):
    output = [0,0,0]
    assert(len(output) == len(vector))
    for i in range(len(vector)):
        output[i] = number * vector[i]
    return output
```

Inputs		Weights		Final Predictions		
(0.65	*	0.3)	=	0.195	=	Injury prediction
(0.65	*	0.2)	=	0.13	=	Win prediction
(0.65	*	0.9)	=	0.585	=	Happy prediction

Gradient descent with multiple outputs

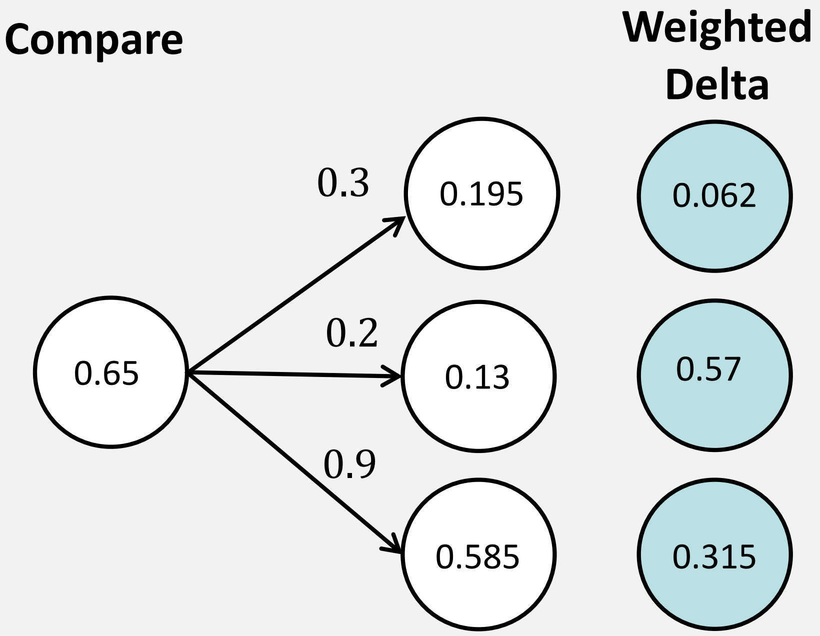
- **Single input → multiple predictions**
  - Making individual predictions and calculate individual `error` and `delta`




```
input = wlrec[0]
true = [hurt[0], win[0], sad[0]]
pred = neural_network(input,weights)

error = [0, 0, 0]
delta = [0, 0, 0]
for i in range(len(true)):
    error[i] = (pred[i] - true[i]) ** 2
    delta[i] = pred[i] - true[i]
```

Gradient descent with multiple outputs

- **Single input → multiple predictions**
  - Calculate each `weighted_delta` and update weights
    - As before the `weight_delta` values are computed by multiplying the input value with the node delta for each weight

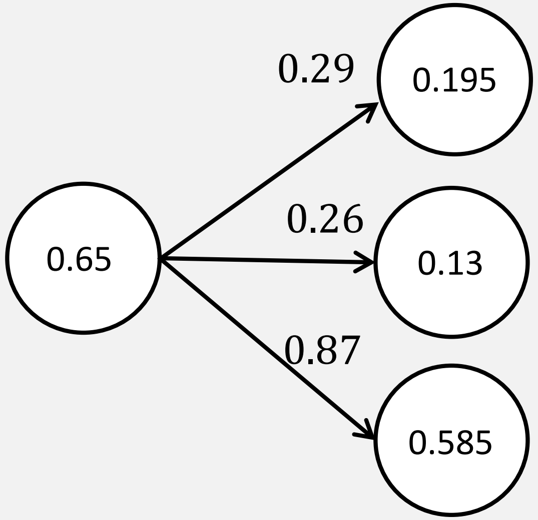



```python
def scalar_ele_mul(number,vector):
    output = [0,0,0]
    assert(len(output) == len(vector))
    for i in range(len(vector)):
        output[i] = number * vector[i]
    return output

weight_deltas = scalar_ele_mul(input,weights)
```

Gradient descent with multiple outputs

- **Single input → multiple predictions**
  - Perform each weight update

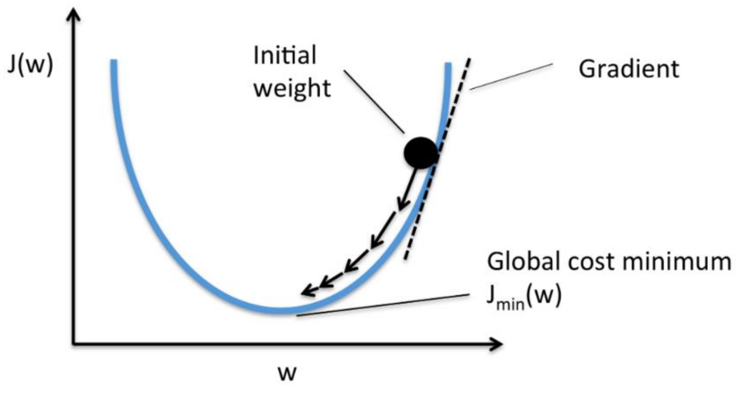
Learn

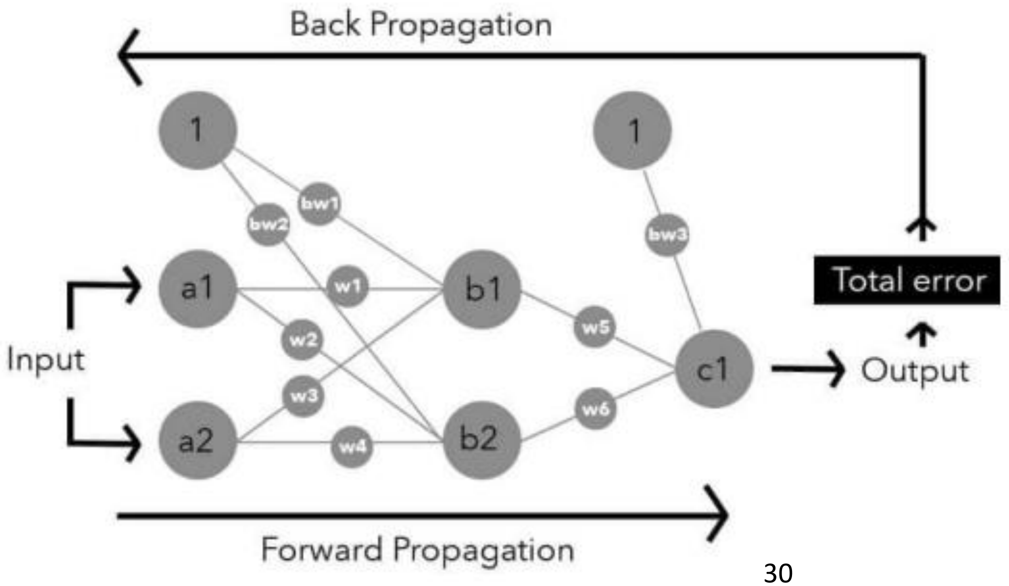



```
alpha = 0.1
for i in range(len(weights)):
    weights[i] -= (weight_deltas[i] * alpha)
```

Image source: https://medium.com

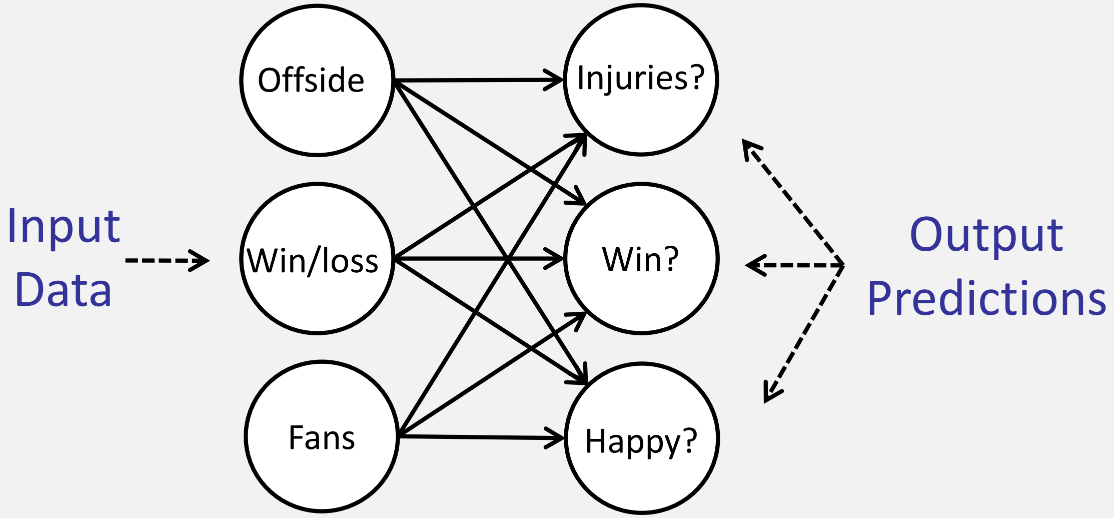
- Gradient Descent
  - Networks with multiple inputs
  - Networks with multiple outputs
  - Networks with multiple inputs and outputs
- Correlation
  - Learning Correlation
  - Creating Correlation
- Backpropagation
- Summary





- **Combining the lot!**
  - Of course, it is straightforward to combine multiple input and multiple output networks to build a network that has both multiple inputs and multiple outputs

Empty network with multiple inputs and outputs

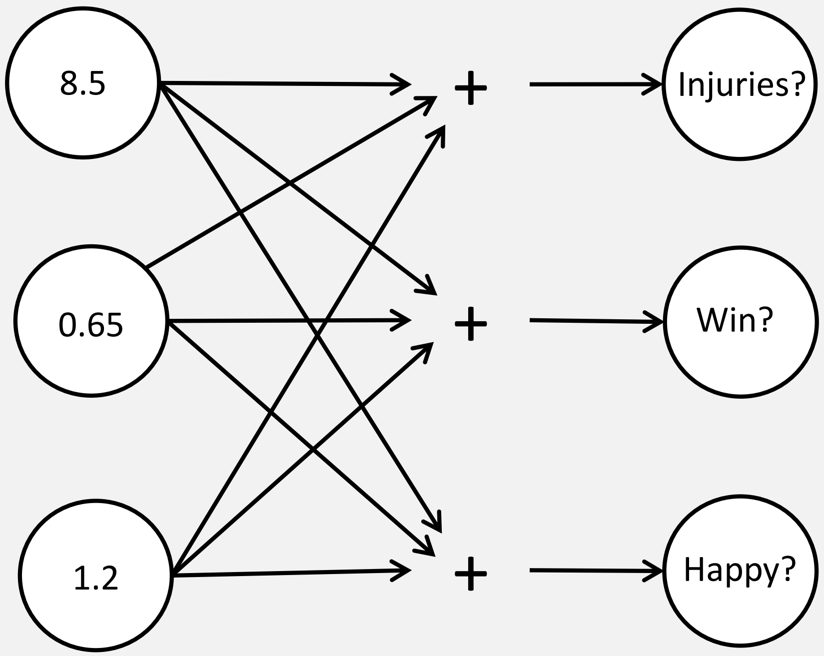



```
              #offside #win #fans
weights = [ [0.1, 0.1, -0.3], #injuries?
            [0.1, 0.2, 0.0], #win?
            [0.0, 1.3, 0.1] ] #happy?

def neural_network(input, weight):
    pred = vec_mat_mul(input * weight)
    return pred
```

Predicting with multiple inputs and outputs

- Each output node takes its own weighted sum of the input and makes a prediction

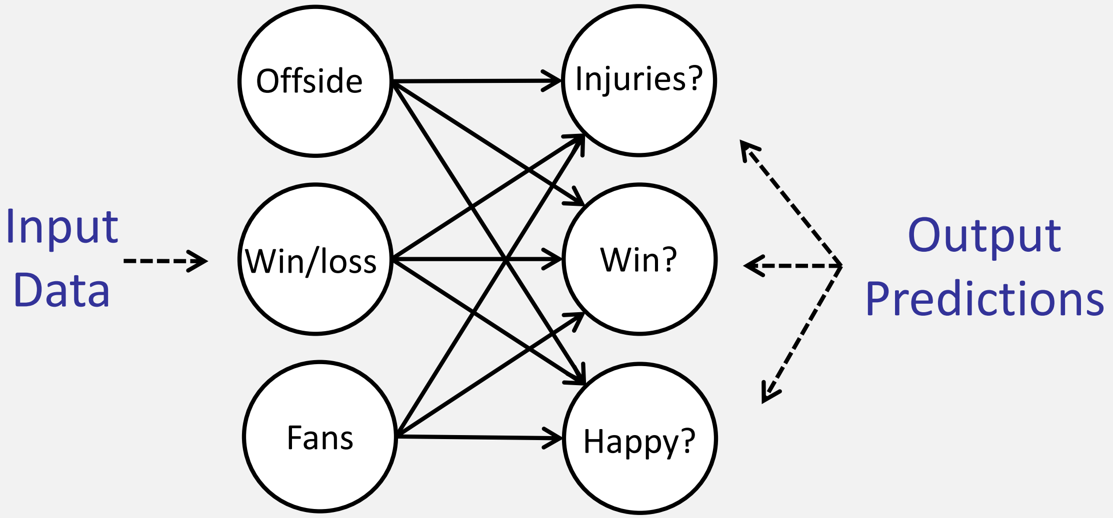
For each output, perform a weighted sum of inputs




```python
def w_sum(a,b):
    assert(len(a) == len(b))
    output = 0
    for i in range(len(a)):
        output += (a[i] * b[i])
    return output

def vect_mat_mul(vect,matrix):
    assert(len(vect) == len(matrix))
    output = [0,0,0]
    for i in range(len(vect)):
        output[i] = w_sum(vect,matrix[i])
    return output
```

- **Combining the lot!**
  - Gradient descent generalizes to arbitrarily large networks

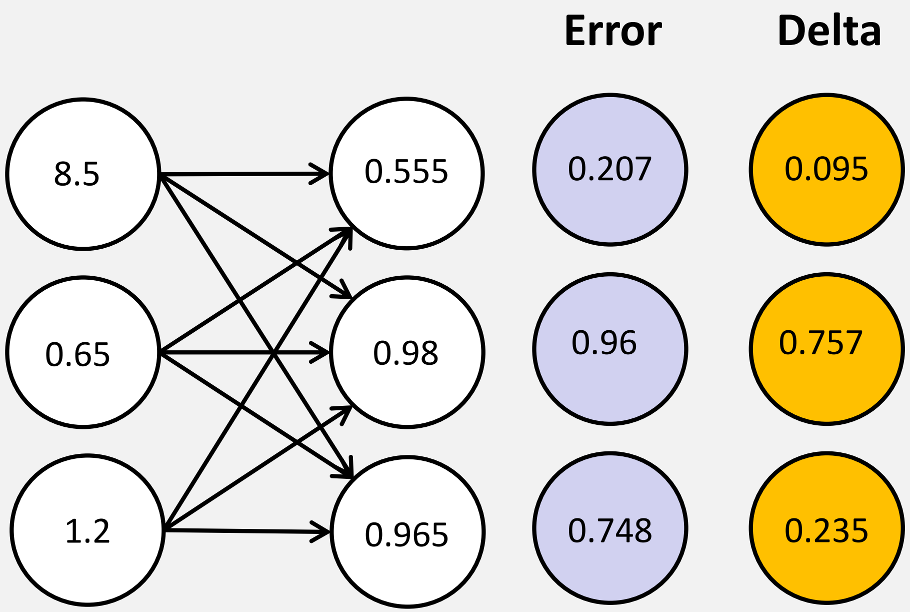
Empty network with multiple inputs and outputs




```
                    #offside #win #fans
weights = [ [0.1, 0.1, -0.3], #injuries?
            [0.1, 0.2, 0.0], #win?
            [0.0, 1.3, 0.1] ] #happy?
def vect_mat_mul(vect,matrix):
    assert(len(vect) == len(matrix))
    output = [0,0,0]
    for i in range(len(vect)):
        output[i] = w_sum(vect,matrix[i])
    return output
def neural_network(input, weights):
    pred = vect_mat_mul(input,weights)
    return pred
```

- **Combining the lot!**

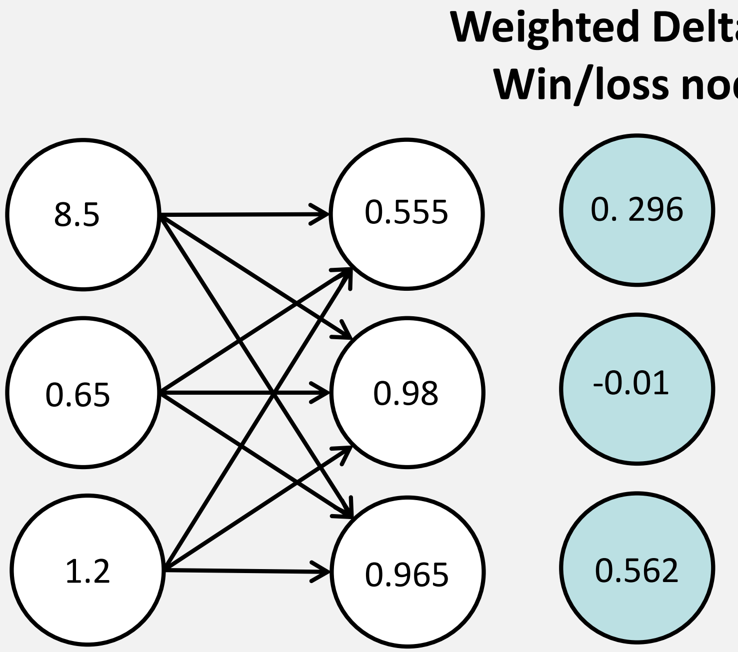
PREDICT: Making a prediction and calculating error and delta




```
Offsides = [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65,0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]
hurt = [0.1, 0.0, 0.0, 0.1]
win = [ 1, 1, 0, 1]
happy = [0.1, 0.0, 0.1, 0.2]
alpha = 0.01

input = [offsides[0],wlrec[0],nfans[0]]
true = [hurt[0], win[0], happy[0]]
pred = neural_network(input,weights)
error = [0, 0, 0]
delta = [0, 0, 0]
for i in range(len(true)):
    error[i] = (pred[i] - true[i]) ** 2
    delta = pred[i] - true[i]
```

- **Combining the lot!**

COMPARE: Calculating each weight_delta and putting it on each weight

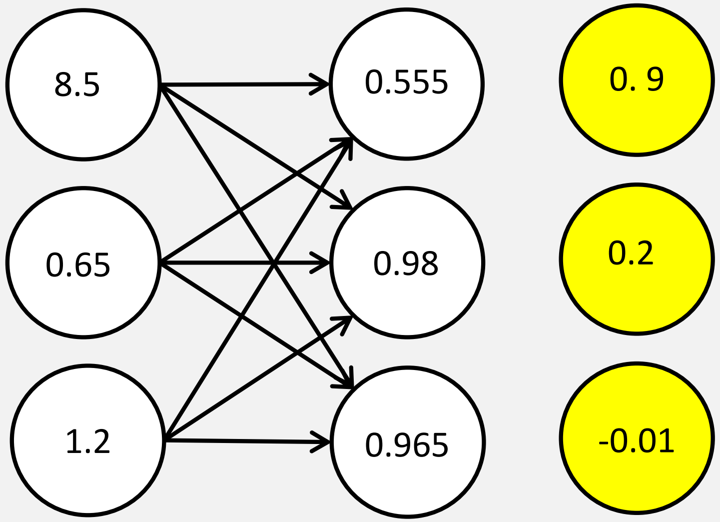



```
def outer_prod(vec_a, vec_b):
    out = zeros_matrix(len(a),len(b))
    for i in range(len(a)):
        for j in range(len(b)):
            out[i][j] = vec_a[i]*vec_b[j]
    return out
weight_deltas = outer_prod(input,delta)
```

- **Combining the lot!**

LEARN: Updating the weights

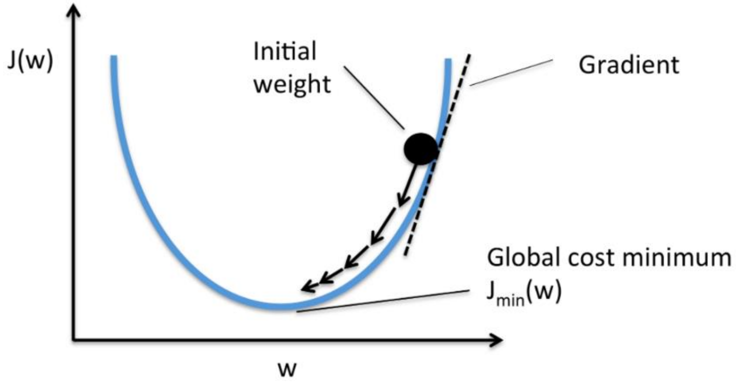
New weights for middle node

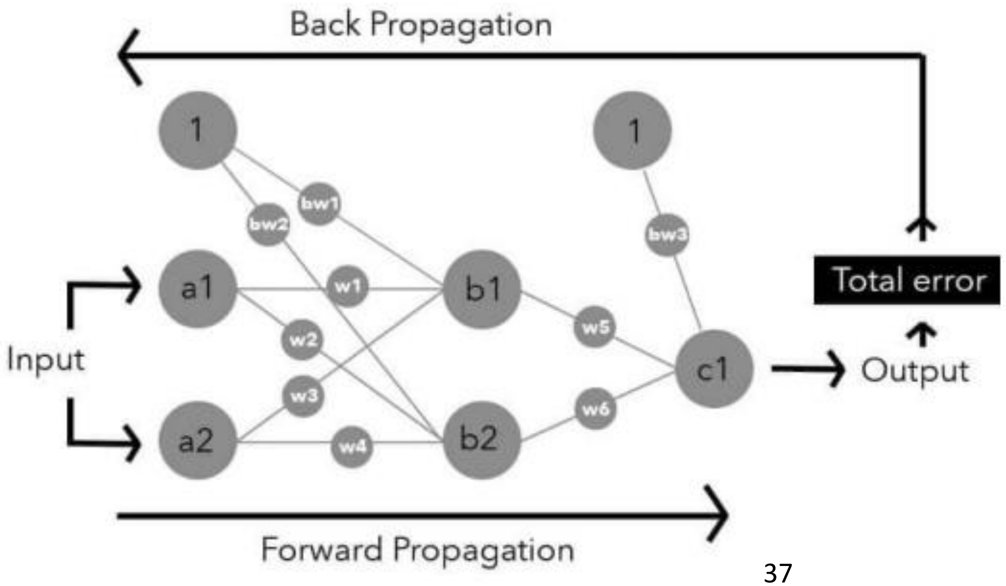



```
for i in range(len(weights)):
    for j in range(len(weights[0])):
        weights[i][j] -= alpha * \ weight_deltas[i][j]
```

Image source: https://medium.com

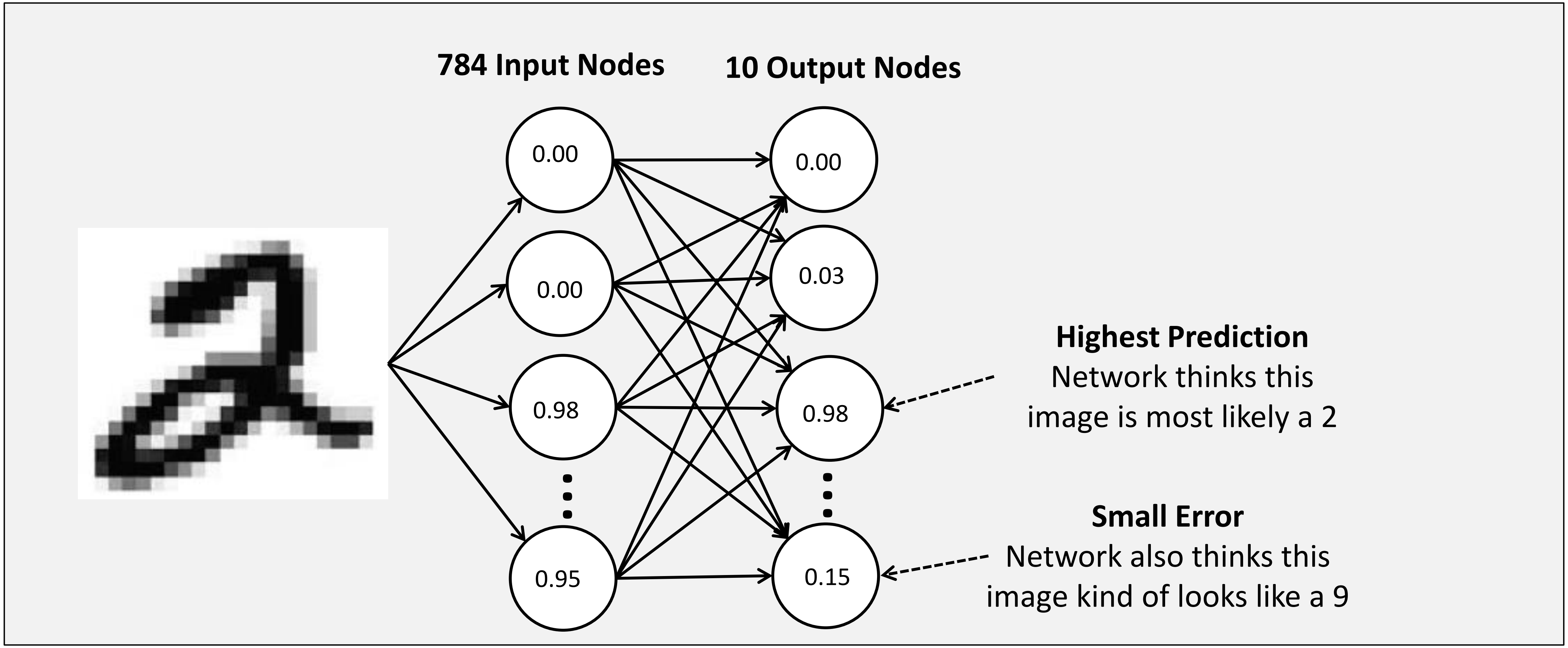
- Gradient Descent
  - Networks with multiple inputs
  - Networks with multiple outputs
  - Networks with multiple inputs and outputs
- Correlation
  - Learning Correlation
  - Creating Correlation
- Backpropagation
- Summary



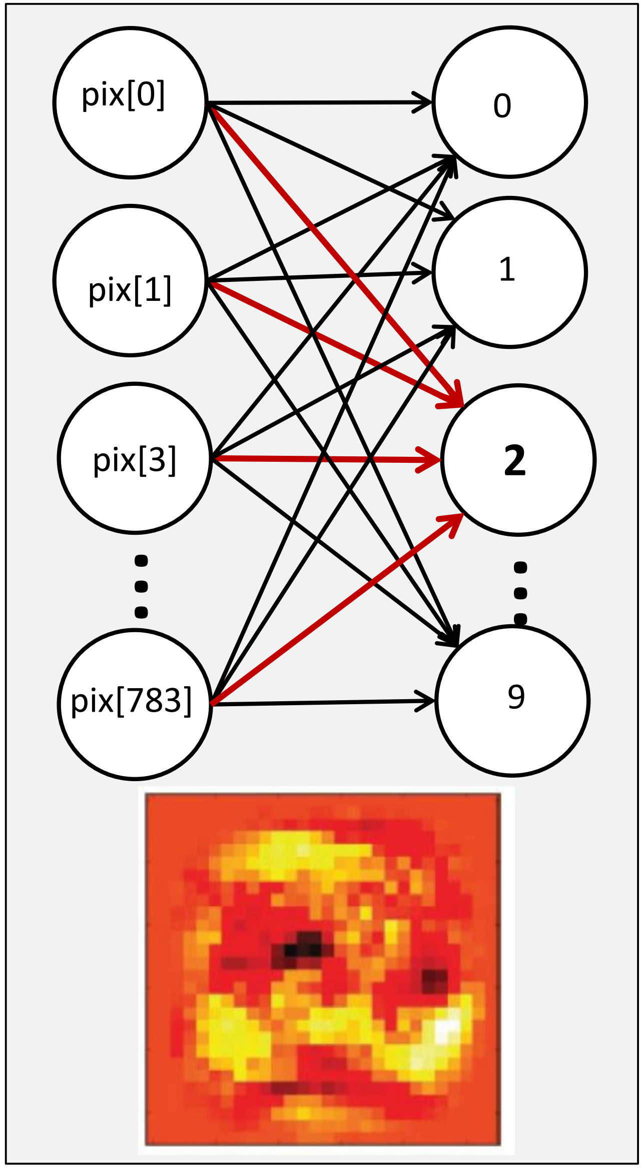


- **Key messages relating to weights so far**
  - **Weights represent knowledge**
    - A network uses this knowledge to interpret the input data
    - Weights are a measure of sensitivity between the input data of the network and its prediction
  - **Each weight tries to reduce the error**
    - We can compute the relationship between the error and any one of the weights so that we know how changing the weight changes the error. You can then use this to reduce the error to 0.
  - **What do they learn in aggregate?**

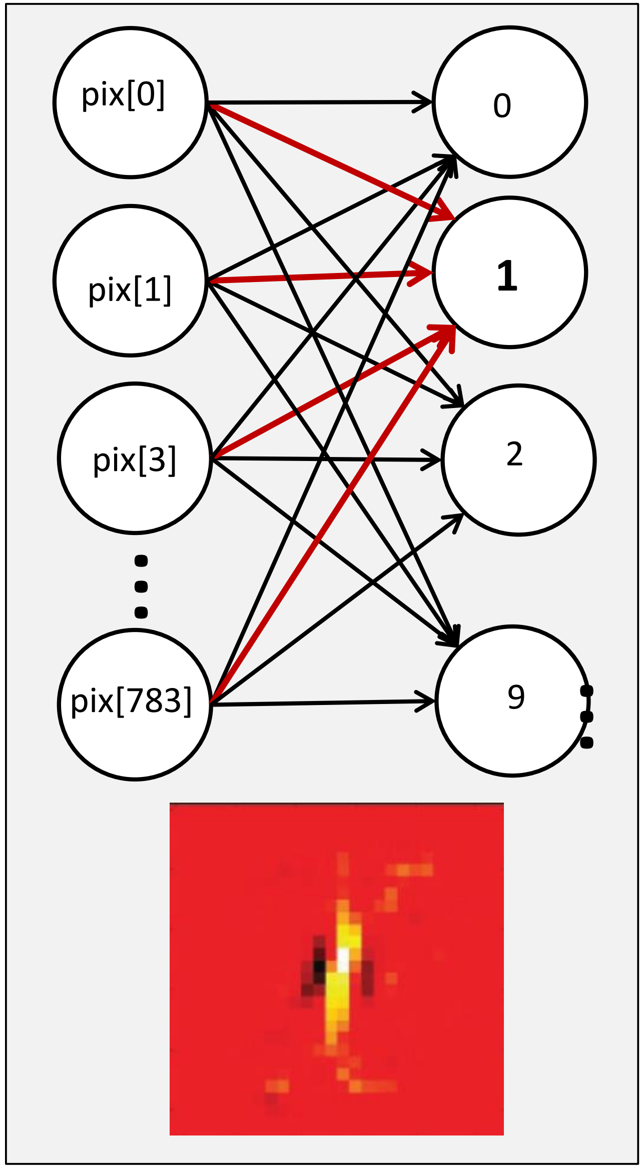
- **What do these weights learn?**

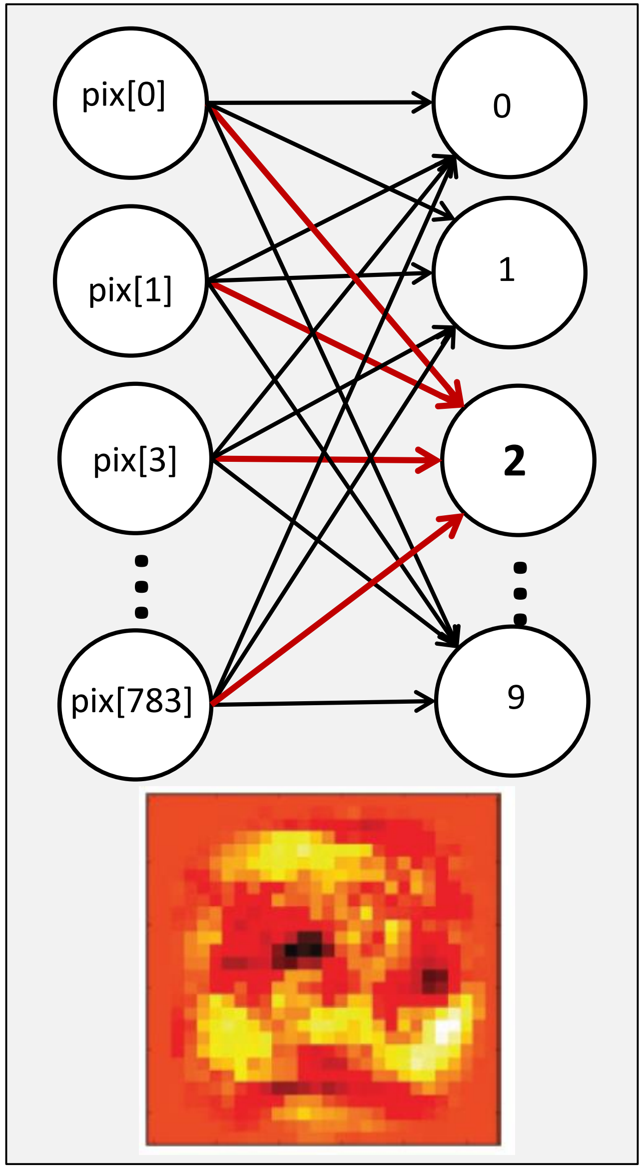


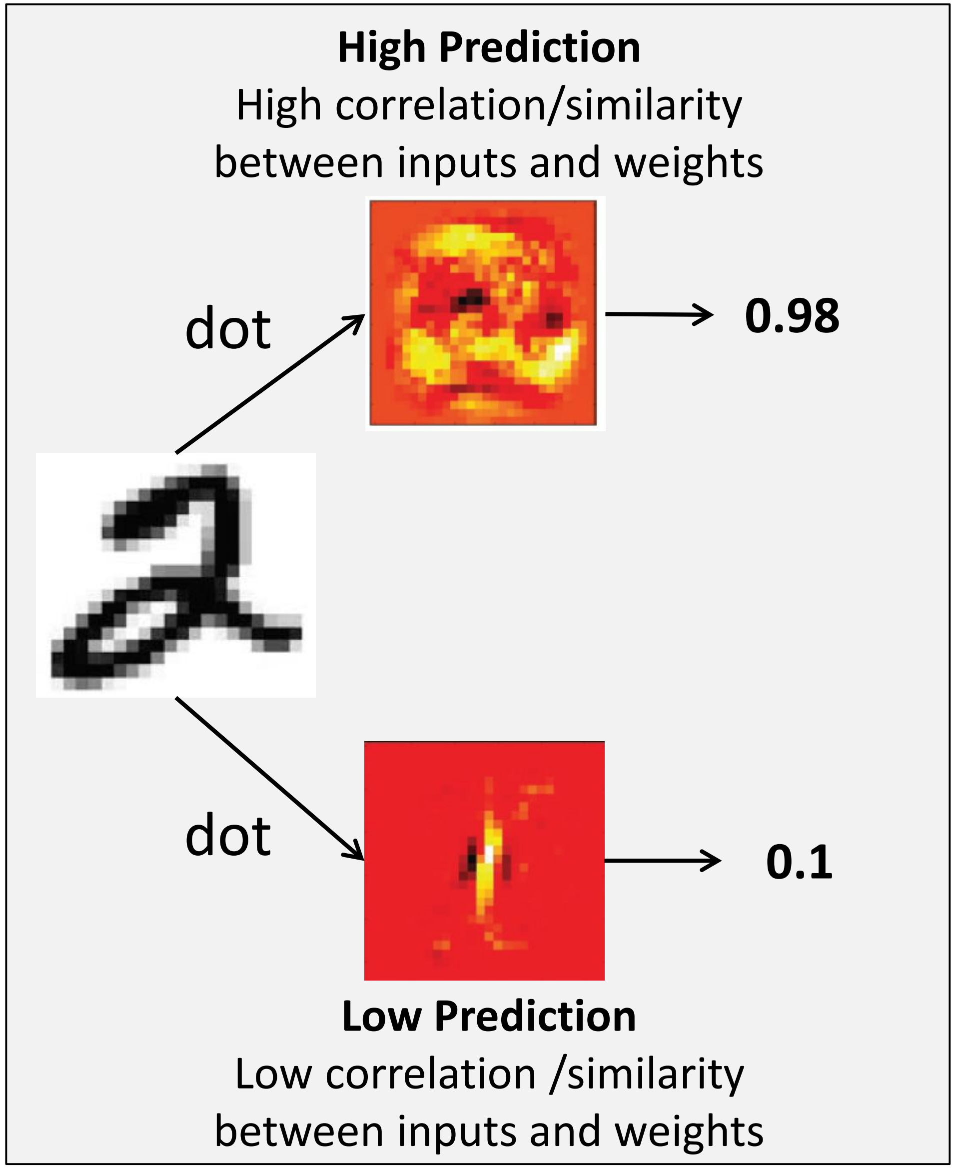
- **What do these weights learn?**
  - **Correlations between input and output**
    - If a weight is high, it means the model believes there's a high degree of correlation between that input and the prediction.
    - If the number is very low (negative), then the network believes there is a very low correlation between that input and the prediction
  - **Why is this?**
    - Weights are found via dot products
    - Dot product encodes similarity



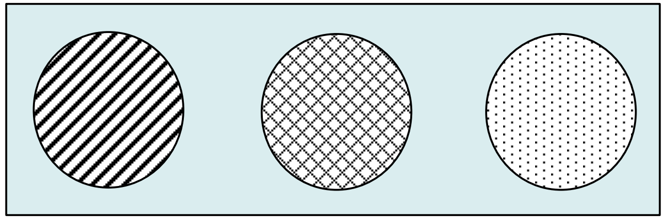
Learning Correlation







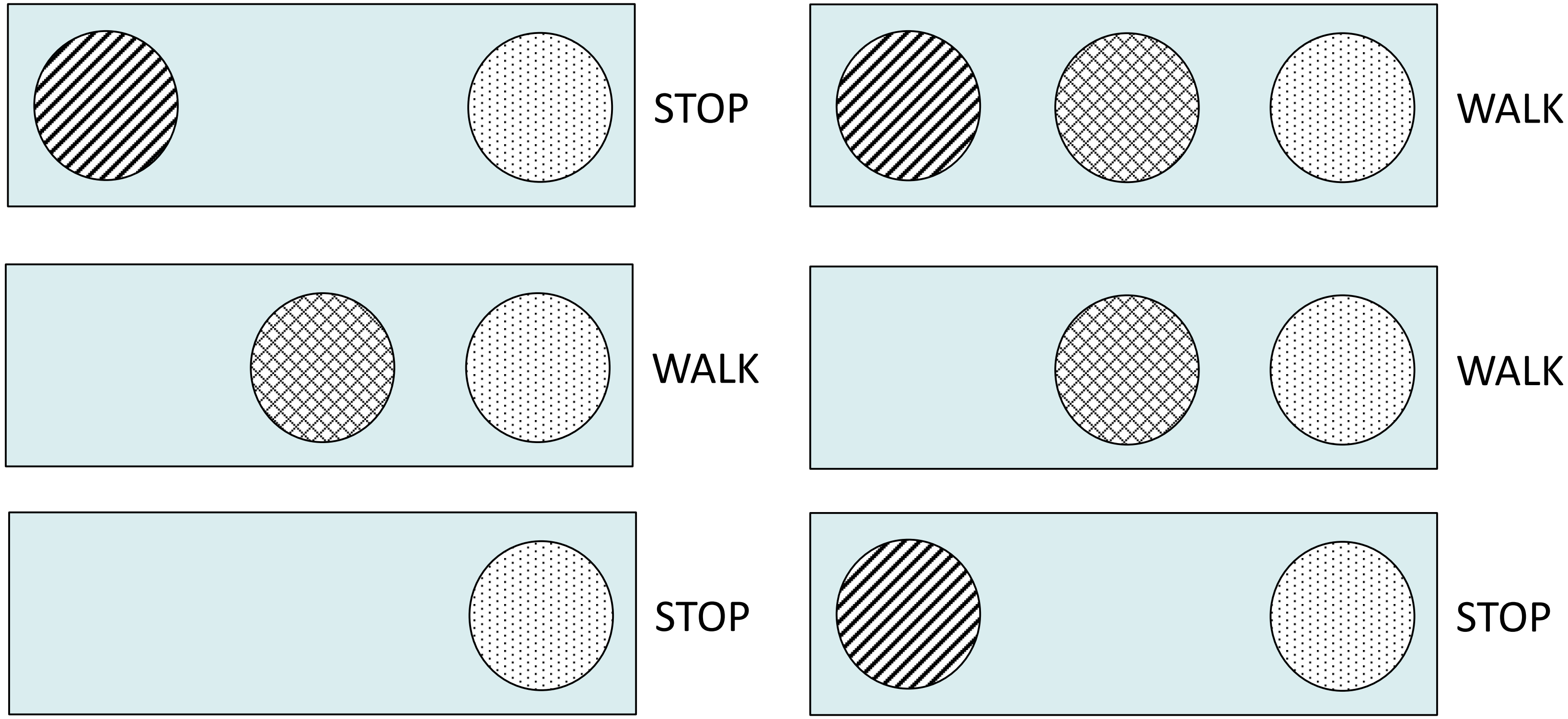
- How does a network learns on entire datasets?
  - Introduce a new toy problem: The streetlight problem
    - Problem: learning to use unfamiliar streetlight in a foreign country



- What we wish to learn: when is safe to cross
- How is this achieved: interpreting the streetlight patterns
  - Observe and record the different streetlight combinations for when people stop or walk

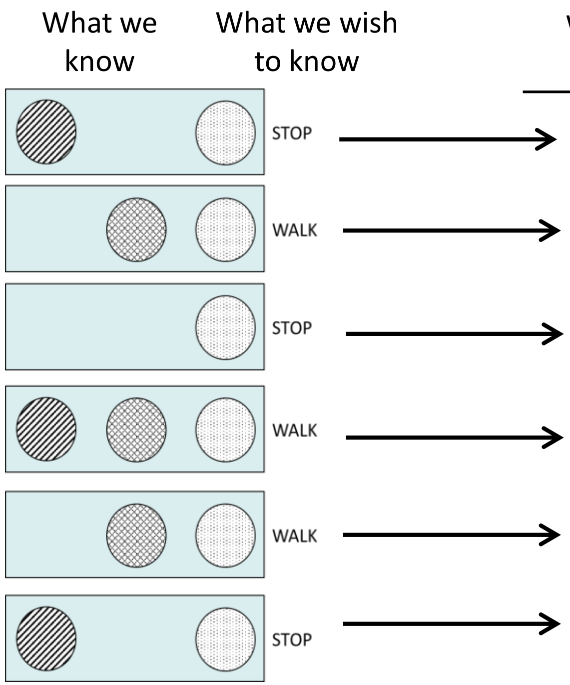
Observing the correlation between light combination and safe crossing

- **Streetlight observations**
  - Observe correlation between the middle light and walking



Learning Correlation

- **How does a network do this?**
  - First step convert observation into datasets
    - What we know, what we wish to know



What we know	What we wish to know
101	0
011	1
001	0
111	1
011	1
101	0

Learning Correlation

```
import numpy as np
weights = np.array([0.5,0.48,-0.7])
alpha = 0.1
streetlights = np.array( [ [ 1, 0, 1 ],
                           [ 0, 1, 1 ],
                           [ 0, 0, 1 ],
                           [ 1, 1, 1 ],
                           [ 0, 1, 1 ],
                           [ 1, 0, 1 ] ] )
walk_vs_stop = np.array( [ 0, 1, 0, 1, 1, 0 ] )

input = streetlights[0]
goal_prediction = walk_vs_stop[0]

for iteration in range(20):
    prediction = input.dot(weights)
    error = (goal_prediction - prediction) ** 2
    delta = prediction - goal_prediction
    weights = weights - (alpha * (input * delta))

print("Error:" + str(error) + " Prediction:" + str(prediction))
```

We are only learning from one sample

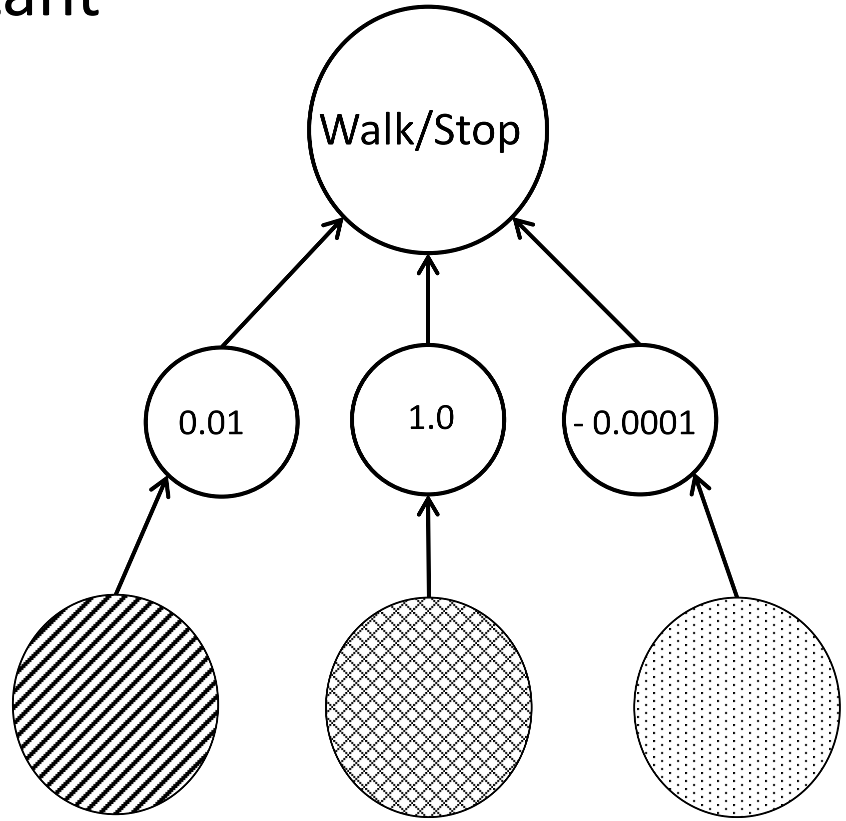
Learning Correlation

```
import numpy as np
weights = np.array([0.5,0.48,-0.7])
alpha = 0.1
streetlights = np.array( [ [ 1, 0, 1 ],
                           [ 0, 1, 1 ],
                           [ 0, 0, 1 ],
                           [ 1, 1, 1 ],
                           [ 0, 1, 1 ],
                           [ 1, 0, 1 ] ] )
walk_vs_stop = np.array( [ 0, 1, 0, 1, 1, 0 ] )

for iteration in range(40):
    error_for_all_lights = 0
    for row_index in range(len(walk_vs_stop)):
        input = streetlights[row_index]
            goal_prediction = walk_vs_stop[row_index]
            prediction = input.dot(weights)
            error = (goal_prediction - prediction) ** 2
            error_for_all_lights += error
            delta = prediction - goal_prediction
            weights = weights - (alpha * (input * delta))
            print("Prediction:" + str(prediction))
        print("Error:" + str(error_for_all_lights) + "\n")
```

```
Prediction:-0.19999999999999996
Prediction:-0.19999999999999996
Prediction:-0.5599999999999999
Prediction:0.6160000000000001
Prediction:0.17279999999999995
Prediction:0.17552
Error:2.6561231104

Prediction:0.14041599999999999
Prediction:0.3066464
Prediction:-0.34513824
Prediction:1.006637344
Prediction:0.4785034751999999
Prediction:0.26700416768
Error:0.9628701776715985
.
.
.
Prediction:-0.002241027381440552 4
Prediction:0.9978745386023716
Prediction:-0.01672126442988494 7
Prediction:1.0151127459893812
Prediction:0.9969492081270097
Prediction:-0.0026256193329783125
Error:0.0005337367732848 8
```

Learning Correlation

- **What did our light crossing network learn?**
  - It learnt the middle light was most important
  - Note that:
    - The middle weight is near one
    - The left and right weights are near zero
  - The network identified a correlation between the middle node & the output
  - It also identified the randomness between the left and right nodes & the output
    - Reflected by the near zero weights



- **How did the network identify this correlation?**
  - The process of gradient descent can be thought of as applying *upwards* or *downwards* pressure on the weights
    - On average the middle weight received more upward pressure
    - On average the outer weights received more downward pressure

Training Data		Weight Pressure	
1 0 1	0	$-\ 0\ -$	0
0 1 1	1	$0\ +\ +$	1
0 0 1	0	$0\ 0\ -$	0
1 1 1	1	$+\ +\ +$	1
0 1 1	1	$0\ +\ +$	1
1 0 1	0	$-\ 0\ -$	0

- **This pressure comes from the data**
  - Each node is individually trying to correctly predict the outcome
  - Only communication between nodes is the shared error
    - Weight update is multiplying shared error by respective inputs
  - Concept of *Error Attribution*
    - Given the share error, the network must decide weights to update & which to leave

Training Data			Weight Pressure	
1 0 1	0		$-\ 0\ -$	0
0 1 1	1		$0\ +\ +$	1
0 0 1	0	$\longrightarrow$	$0\ 0\ -$	0
1 1 1	1		$+\ +\ +$	1
0 1 1	1		$0\ +\ +$	1
1 0 1	0		$-\ 0\ -$	0

Universität Augsburg University

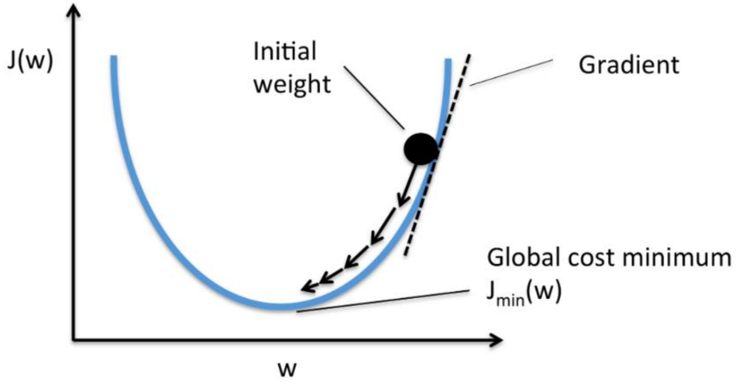
- **This pressure comes from the data**
  - Weight pressure table shows this effect
    - In the first training sample the left and right nodes are uncorrelated with the output so their weights experience downwards pressure
    - In the second training sample the middle and right nodes are correlated with the output and their weights experience upwards pressure

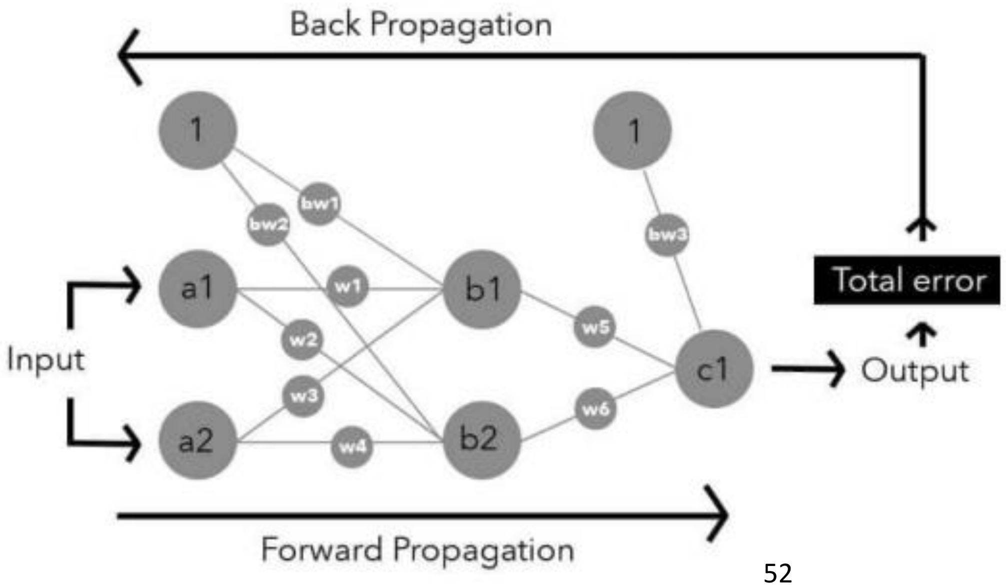
Training Data		→	Weight Pressure	
1 0 1	0		− 0 −	0
0 1 1	1		0 + +	1
0 0 1	0	→	0 0 −	0
1 1 1	1		+ + +	1
0 1 1	1		0 + +	1
1 0 1	0		− 0 −	0

- **How did the network identify this correlation?**
  - Reminder: prediction is a weighted sum of the inputs
  - During training
    - The learning algorithm rewards inputs that correlate with the output with upward pressure on their weight
      - Drives the weights towards 1
    - At the same time, the learning algorithm penalises inputs with discorrelation with downward pressure
      - Drives the weights towards 0
  - Learning rewards correlation with larger weights
  - Learning finds correlation between the two datasets

Image source: https://medium.com

- Gradient Descent
  - Networks with multiple inputs
  - Networks with multiple outputs
  - Networks with multiple inputs and outputs
- **Correlation**
  - Learning Correlation
  - **Creating Correlation**
- Backpropagation
- Summary





- **What happens when there is no correlation?**
  - Considering the following dataset:
    - There is no correlation between *any* input column and the output column.
    - Every weight has an equal amount of upward pressure and downward pressure
  - How can the network learn?

Training Data	
1 0 1	1
0 1 1	1
0 0 1	0
1 1 1	0

→

Weight Pressure	
+ 0 +	1
0 + +	1
0 0 −	0
− − −	0

Creating Correlation

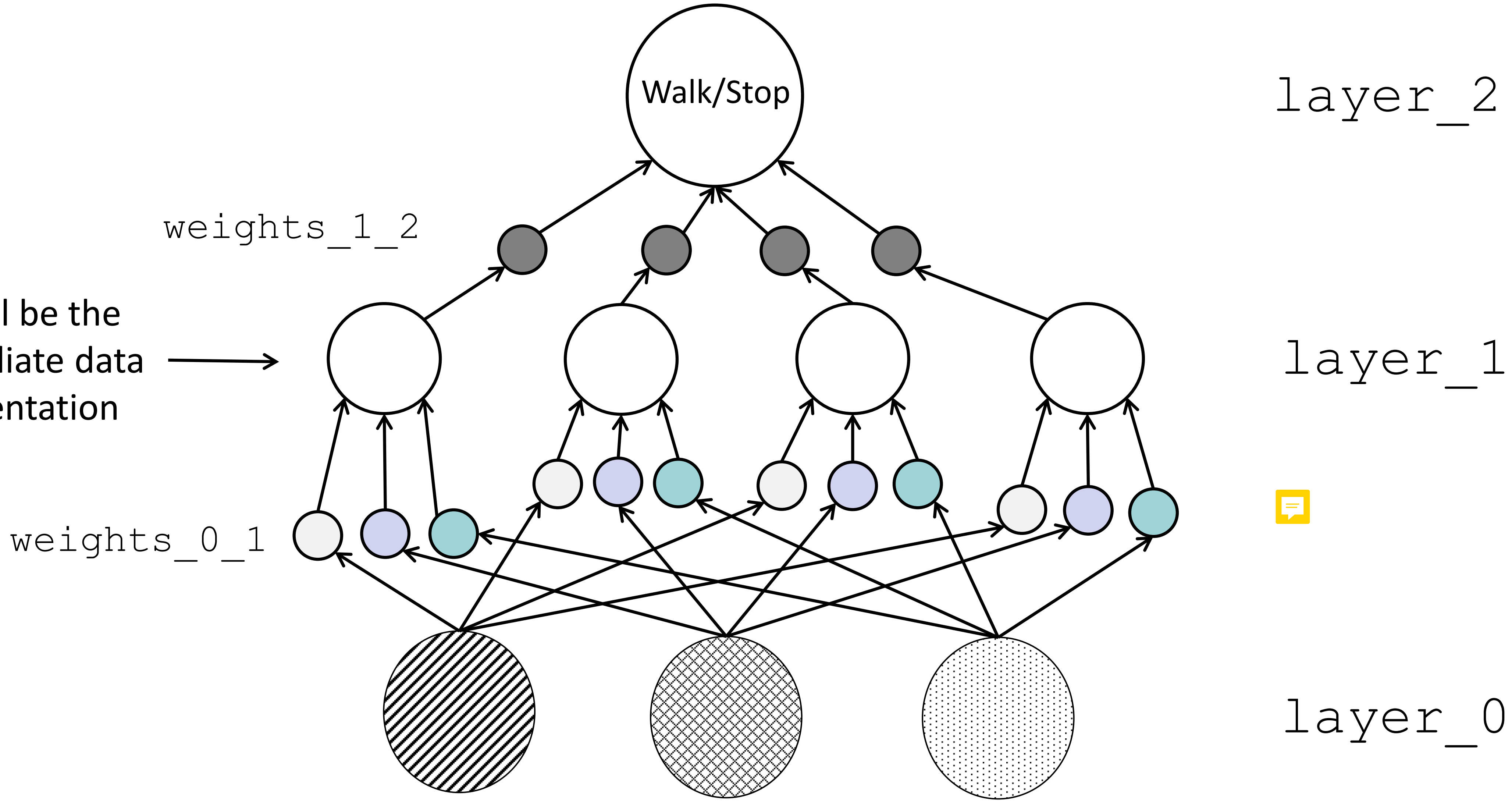
- **Key message**
  - Computing the relationship between the error and weights allows us to change the weights to reduce the error to 0.
  - Adjusting the weights to reduce the error over a series of training examples ultimately searches for correlation between the input and the output layers
  - **If no correlation exists, then the error will never reach 0**
  - This is a major limiting factor, therefore we need to change our approach to cope with this

Creating Correlation

- **What happens when there is no correlation?**
  - Learnt today that neural network is an instrument that searches for correlation between input and output datasets
  - In reality the network searches for correlation between its input and output layers
  - So to learn when there is no correlation, just use more networks (i.e., add another layer)
    - First layer creates a layer with limited correlation with the output
    - Second layer uses this limited correlation to correctly predict the output
  - Hidden layer(s) can be thought of as creating intermediate dataset(s) that has correlation with the output

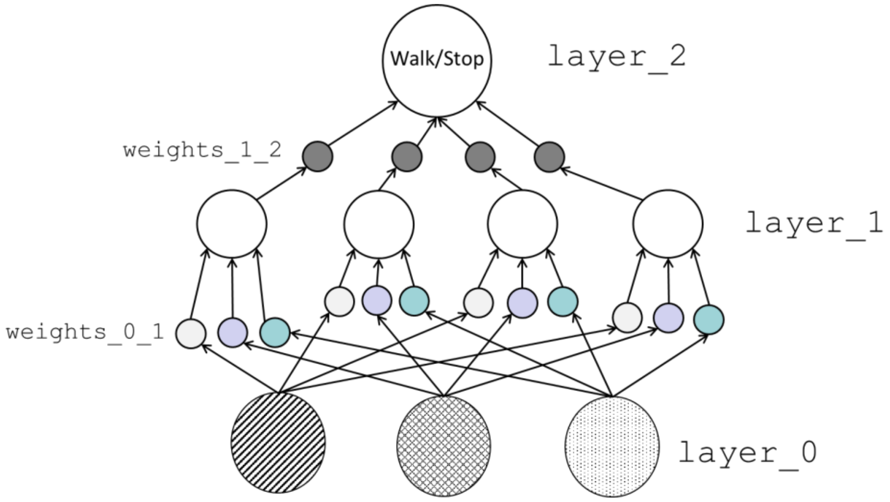
Creating Correlation

This will be the intermediate data representation

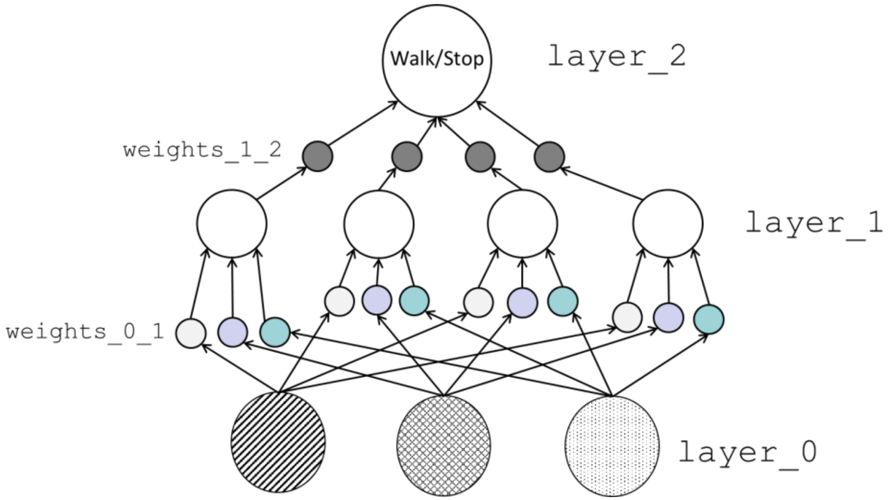


Universität Augsburg University

- **Stacked neural networks**
  - Output of the first network is the input to the next
  - Prediction is identical to what we have already learnt
  - Top half of the network can be trained via gradient descent using methods we have learnt
  - But how to update the weights for the hidden layer?



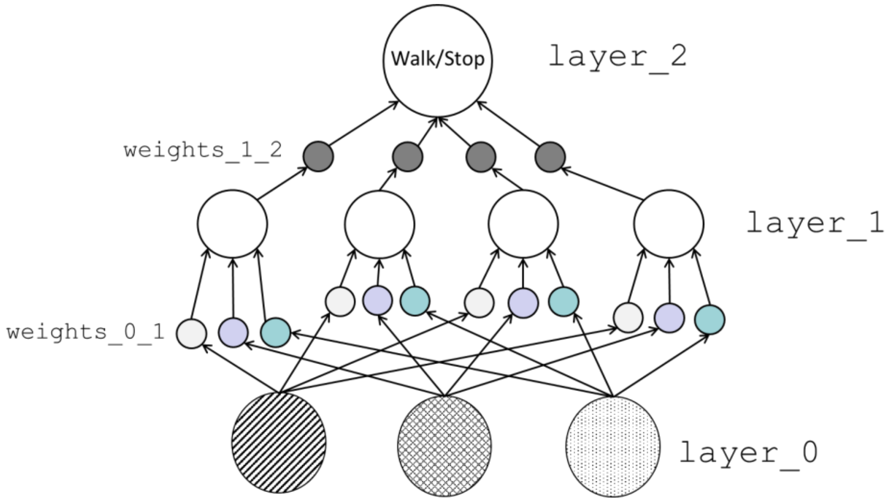
- **Linear vs. nonlinear**
  - Without an addition of non-linear transformations in-between layers, the hidden layers adds nothing new to the network
  - Why?
    - **Any two multiplications can (in general) by accomplished using a single multiplication**



$$1 \times 10 \times 10 = 100$$
$$5 \times 20 = 100$$

$$1 \times 0.25 \times 0.9 = 0.225$$
$$1 \times 0.225 = 0.225$$

- **Linear vs. nonlinear**
  - Stacking linear neural networks does not give and more power
  - It just gives a more computationally expensive version of a single weighted sum
  - Anything that the three-layer linear network can do, the two-layer linear network can also do



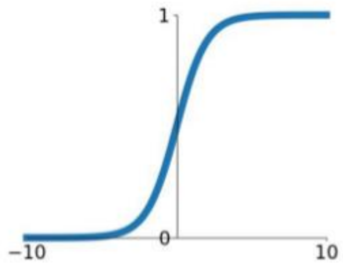
Creating Correlation

- **Linear vs. nonlinear**
  - Moreover, in linear networks the hidden nodes do not have a correlation of their own
  - They are more or less correlated to the various input nodes
  - Ideally we want the middle layer to sometimes correlate with the input and sometimes not correlate
  - Such an effect means the hidden layers can give a correlation of their own
    - This is called conditional correlation
  - This can be done by introducing non-linearities

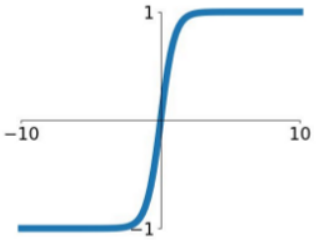
Creating Correlation

Activation Functions

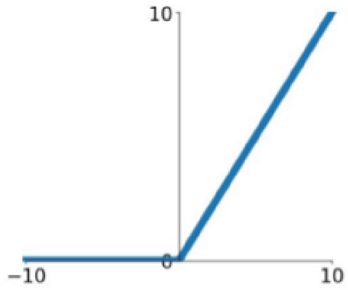
- Uses the weighted input value to determine the level of output activation
  - Introduces nonlinearities into network
- Typical activation functions include
  - Identity $\rightarrow f(x) = x$
  - Logistic $\rightarrow f(x) = 1/(1 + \exp(-x))$
  - Tanh $\rightarrow f(x) = \tanh x$
  - Rectified Linear unit $\rightarrow f(x) = \max(0, x)$
  - Sigmoid $\rightarrow f(x) = 1/(1 + \exp^{x})$



Sigmoid



Tanh



ReLU

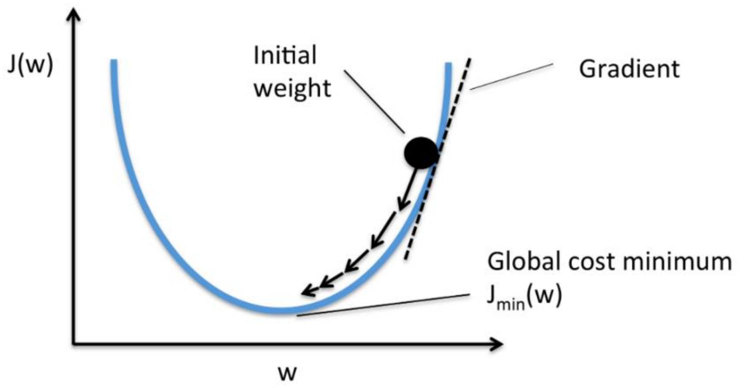
Creating Correlation

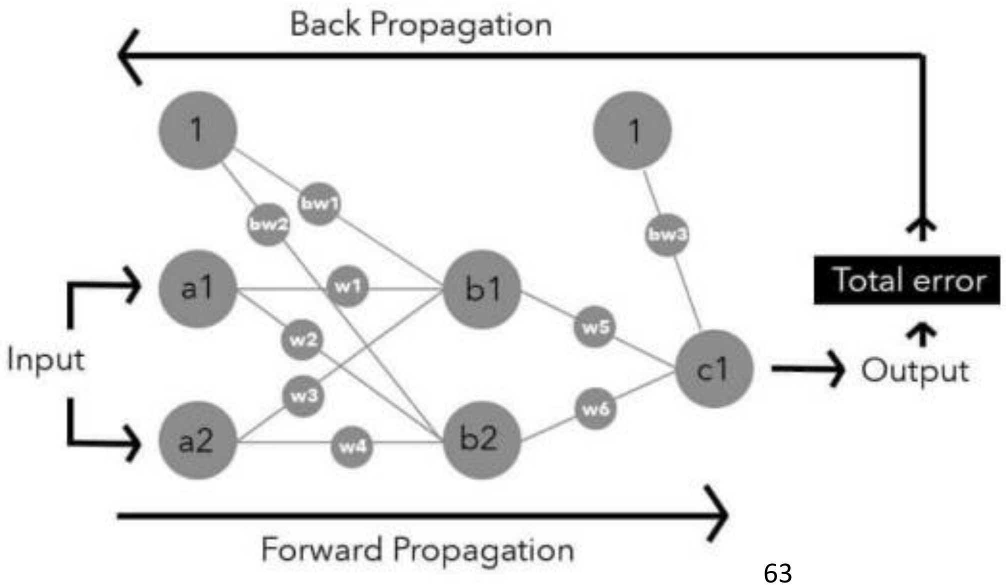
- **What's the point of creating intermediate datasets that have correlation?**
  - Deep learning is all about creating intermediate layers
  - Each node in an intermediate layer represents the presence or absence of a different configuration of inputs
  - No individual input has to correlate directly with the target
  - Middle layer attempts to identify different configurations of the input that may or may not correlate with the output
  - These many different configurations will give the final layer the information (correlation) it needs to perform the prediction

Today's Lecture

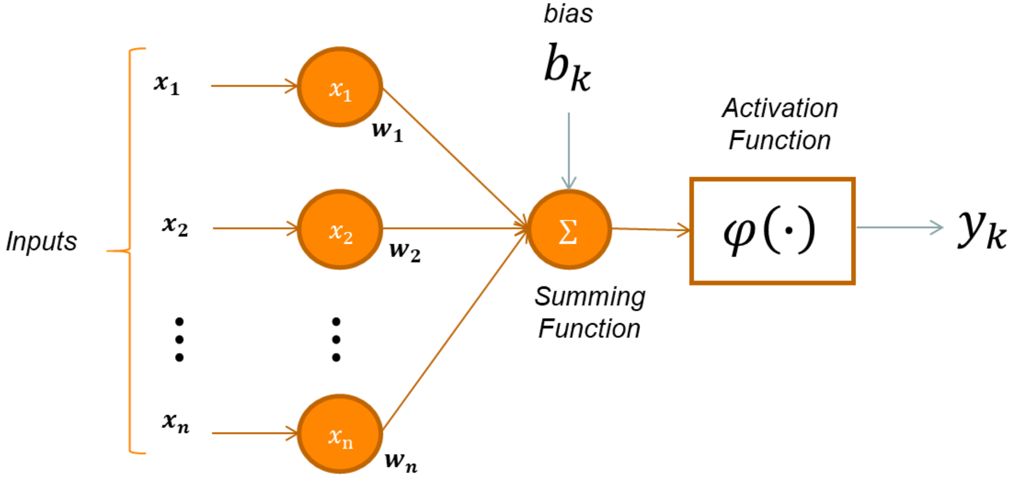
Image source: https://medium.com

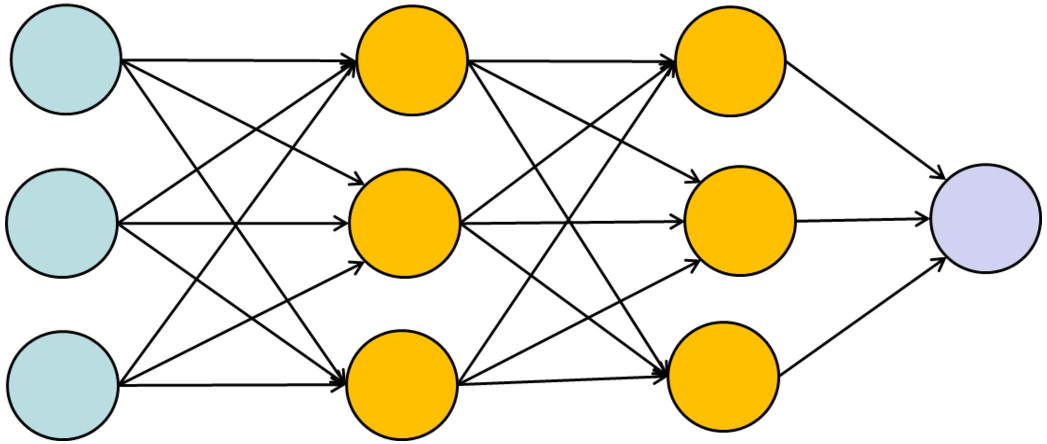
- Gradient Descent
  - Networks with multiple inputs
  - Networks with multiple outputs
  - Networks with multiple inputs and outputs
- Correlation
  - Learning Correlation
  - Creating Correlation
- **Backpropagation**
- Summary



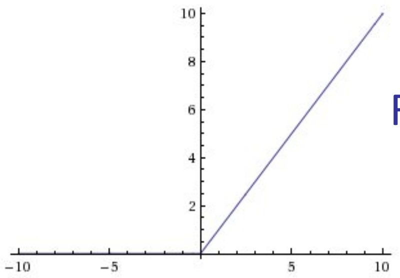


- **Neural Networks**
  - Use layered combinations of perceptrons make complex predictions



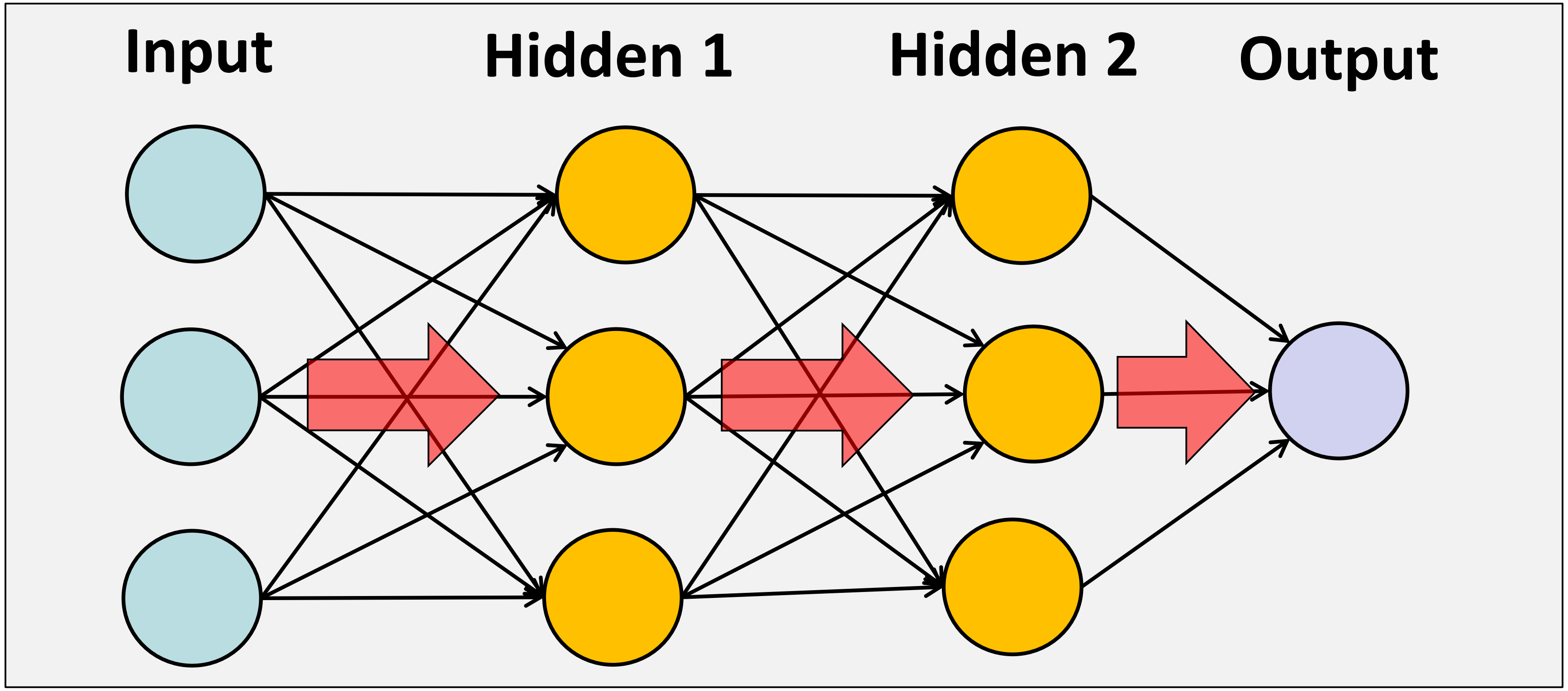


$$\varphi\left(\begin{pmatrix} w_1 & w_2 & \dots & b \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ \dots \\ 1 \end{pmatrix}\right) = \varphi(w_1x_1 + w_2x_2 + \dots + b\,) = y_k$$

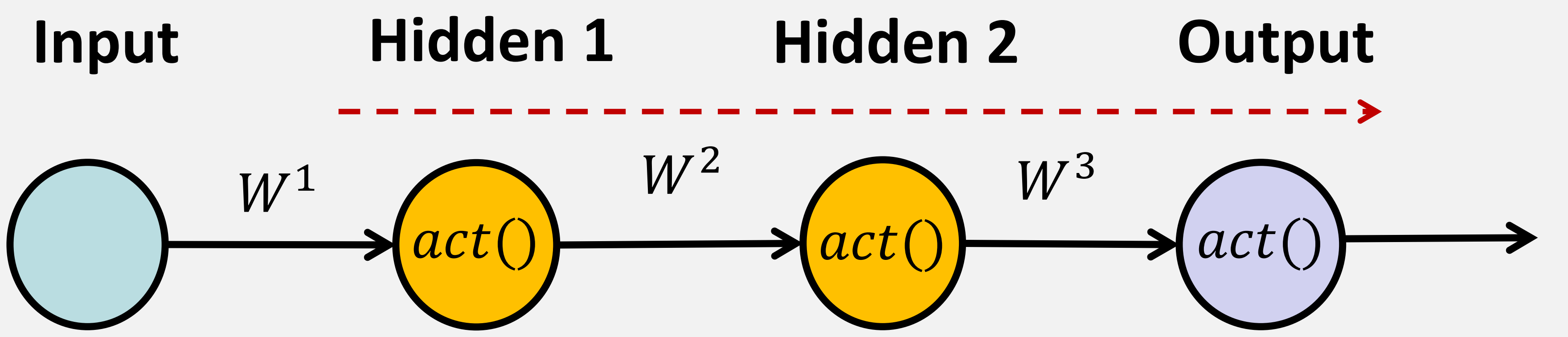


Rectified Linear unit
$\varphi(x) = \max(0, x)$

- **Forward Propagation**
  - Information flows from input to output to make a predication

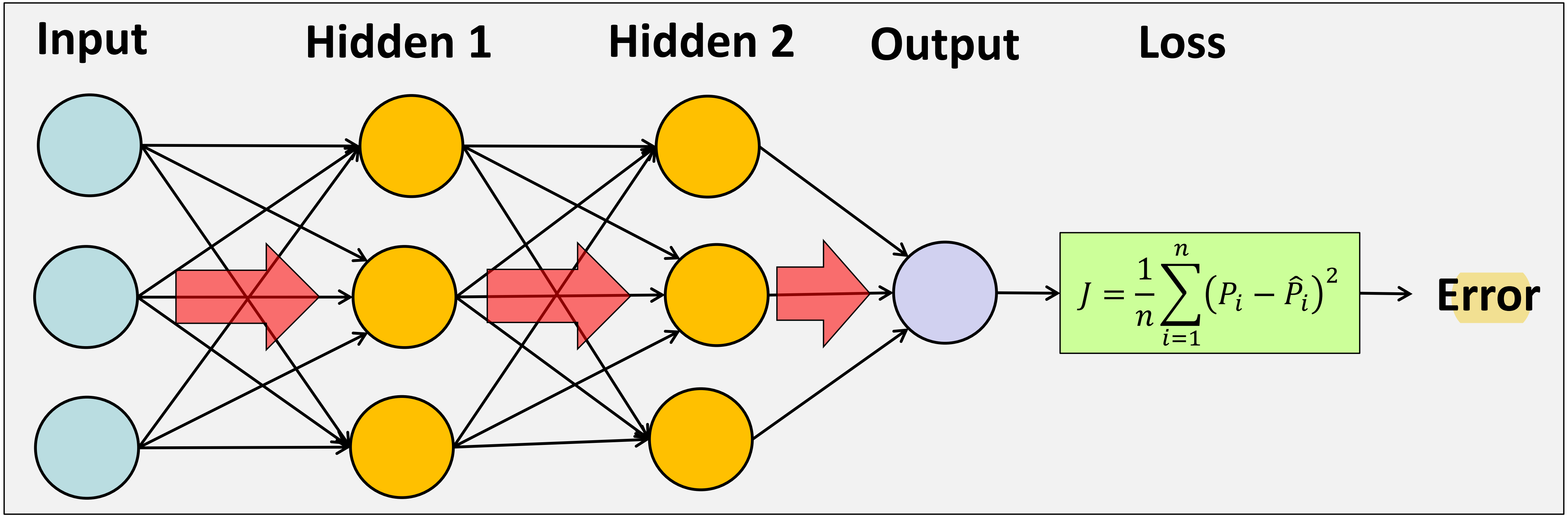


- **Forward Propagation**
  - Each neuron is a function of the previous one connected to it
    - Output is a composite function of the weights, inputs, and activations
      - Change any one of these and ultimately the output well change



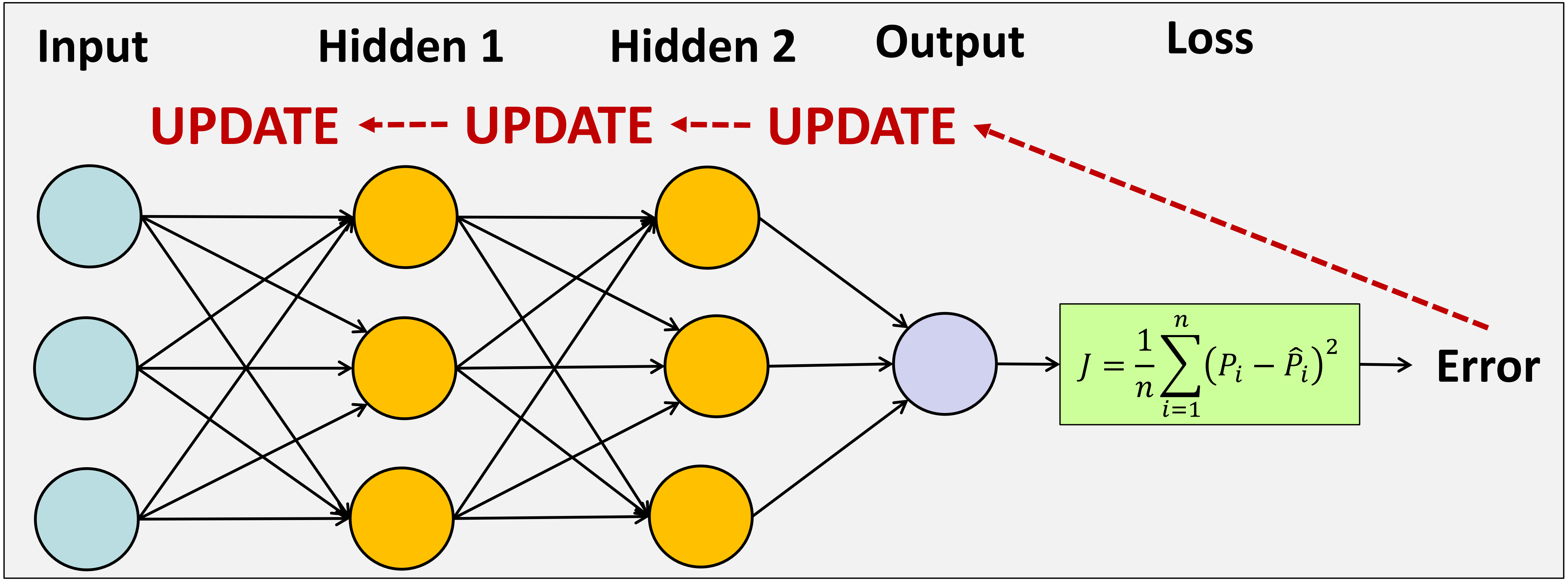
$$output = act\left(W^3 \times act\left(W^2 \times act(W^1 \times input)\right)\right)$$

- **Calculate Error/Loss**
  - Calculate difference between predication and target

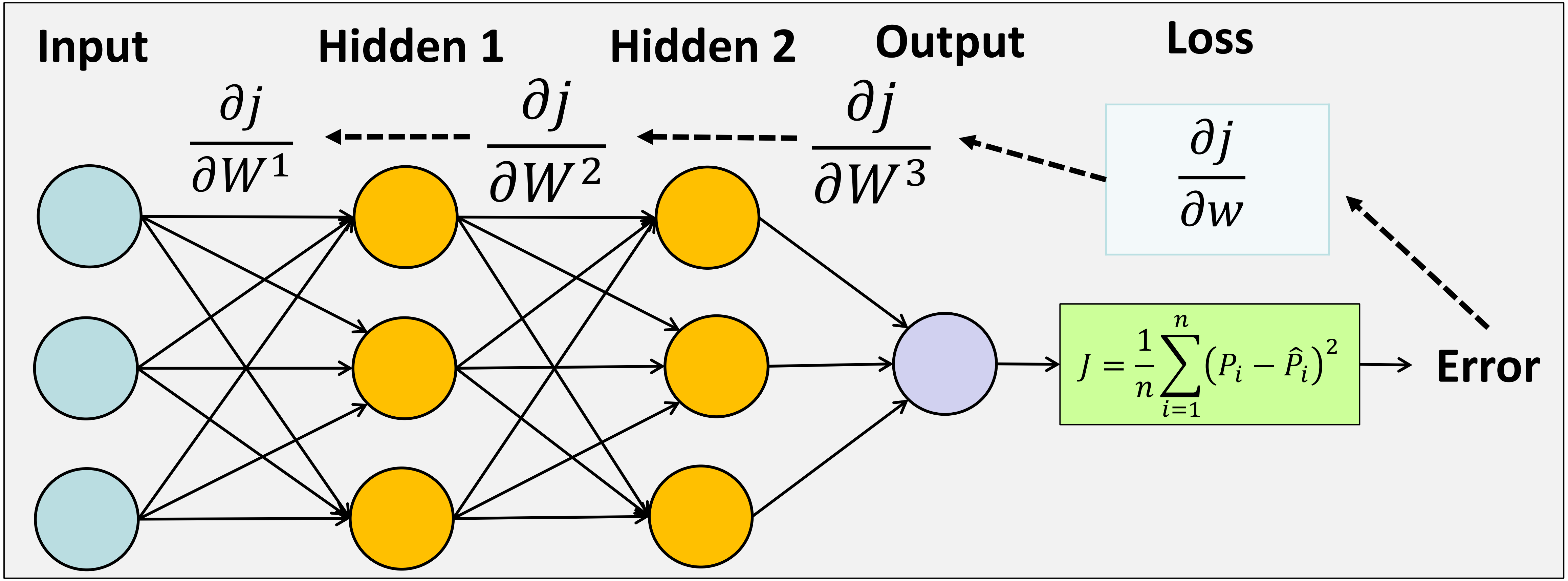


- **Propagating error back through the network**
  - A neural network learns via a error function
    - This function defines the relationship between the error and the weights of the network
    - Update weights via derivative of this function → $W \mathrel{+}= W + \alpha \frac{\partial j}{\partial w}$
  - Backpropagation is a method for computing the derivatives with respect to every weight in a network
    - With this derivative we perform our gradient descent update
  - It is known as backpropagation as we use it to travers the output error back through the network

- **Perform Gradient descent**
  - Output value effected by weights at all layers

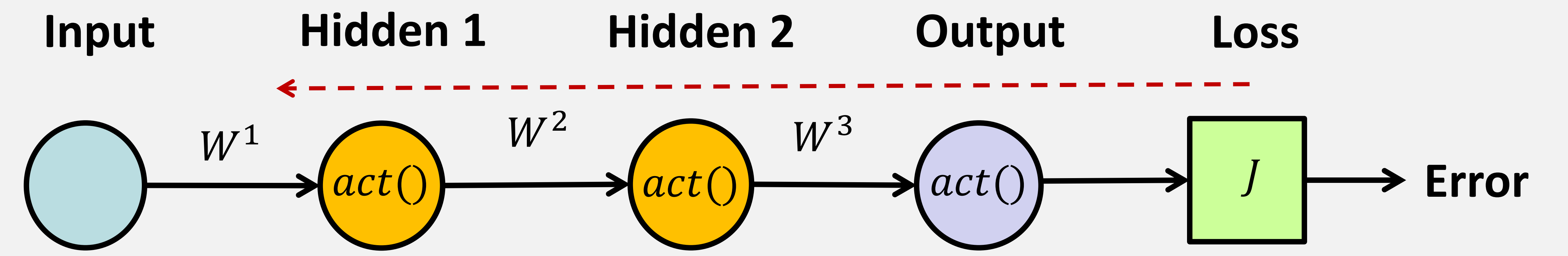


- **Backpropagation**
  - Tool to calculate the gradient of the loss function



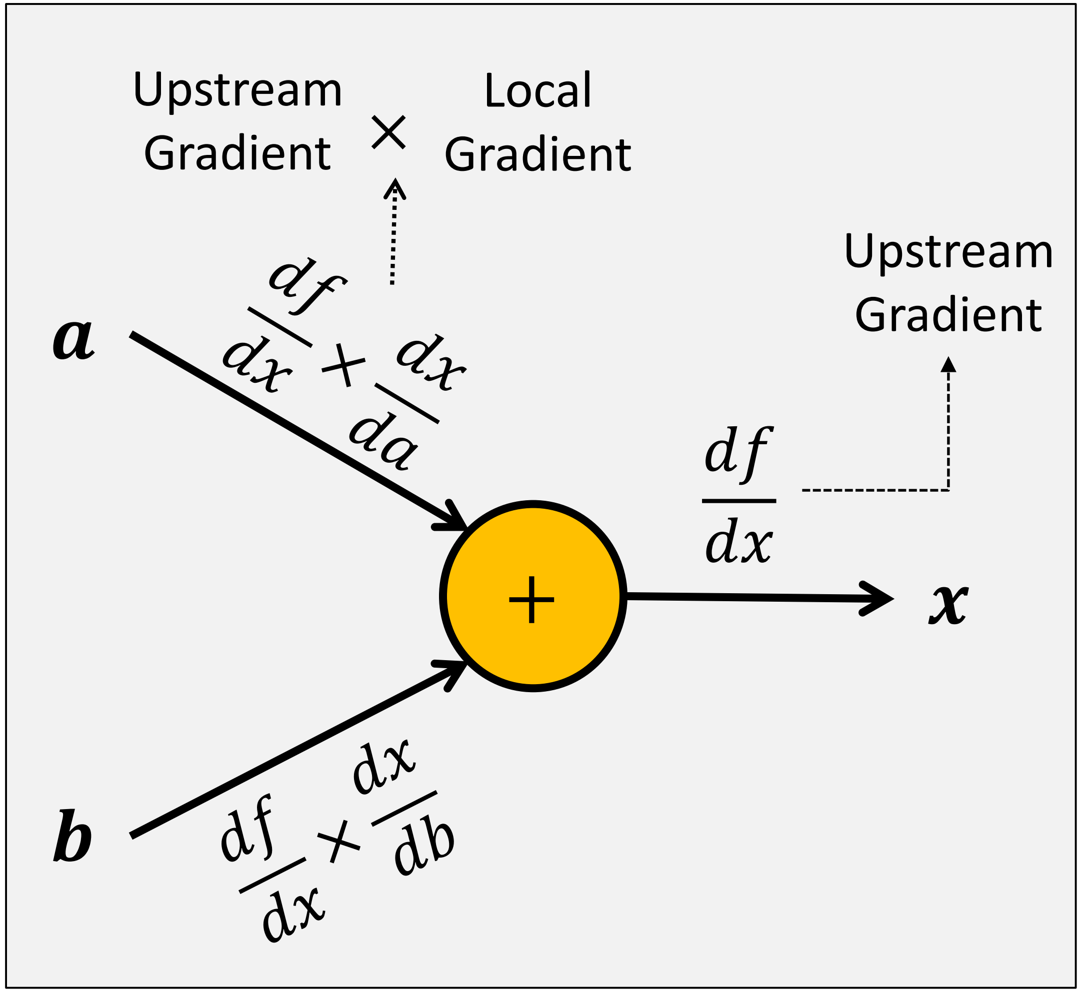
Backpropagation

- **Calculating gradient for arbitrary weight**
  - Iteratively apply the chain rule
  - Note: Error is now a function of the output and hence a function of the input, weights, and activation functions



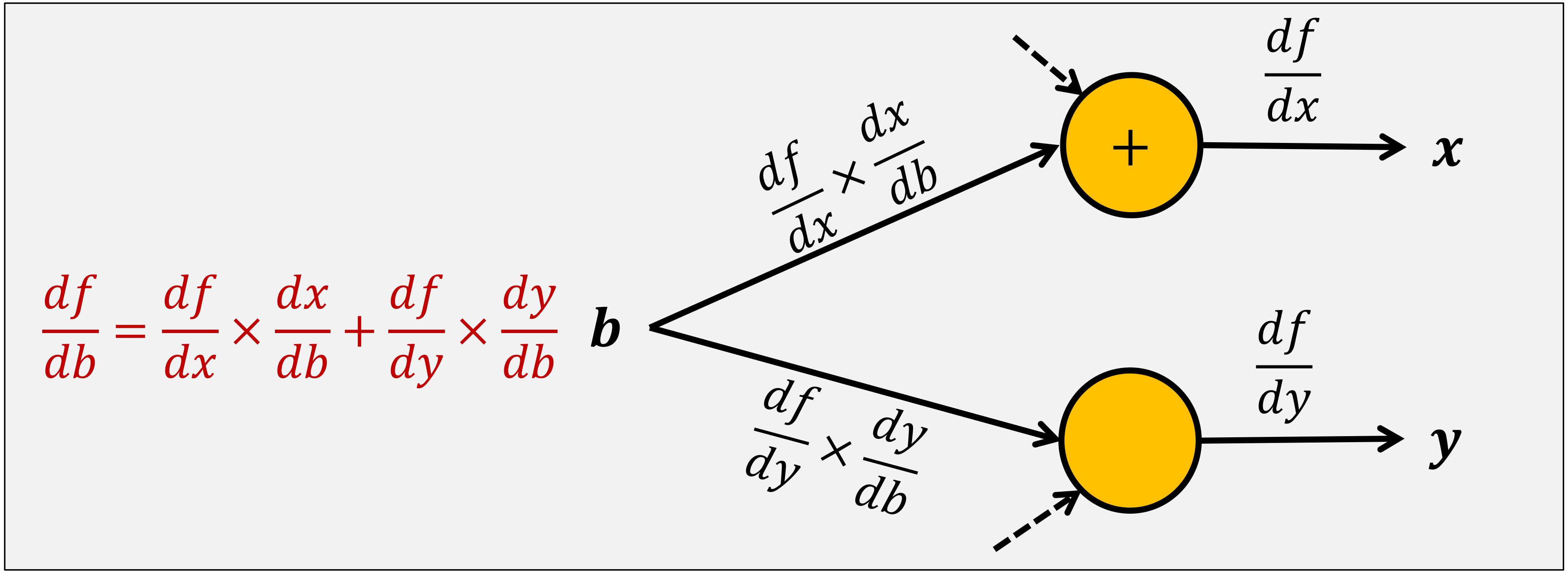
$$\frac{\partial error}{\partial W^1} = \frac{\partial error}{\partial output} \times \frac{\partial output}{\partial hidden2} \times \frac{\partial hidden2}{\partial hidden1} \times \frac{\partial hidden1}{\partial W^1}$$

- **Calculating gradient via backpropagation**
  - Local gradients can be calculated before starting the backpropagation process
  - Iteratively apply the chain rule
    - The derivative of the output of the network with respect to a local variable is found by multiplying the local gradient with the upstream gradient

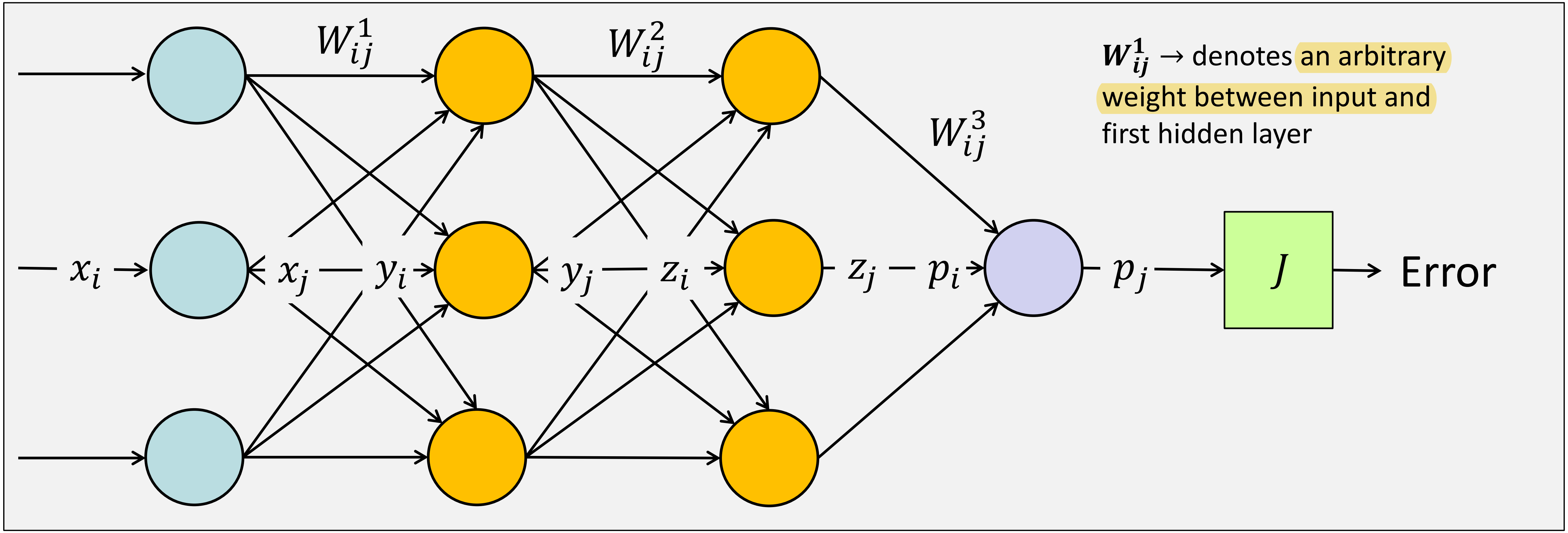


Universität Augsburg University

- **Calculating gradient via backpropagation**
  - Multivariate chain rule: Gradients add at branches



- **Example: Calculating** $\frac{\partial error}{\partial W_{ij}^{1}} = \frac{\partial j}{\partial W_{ij}^{1}}$



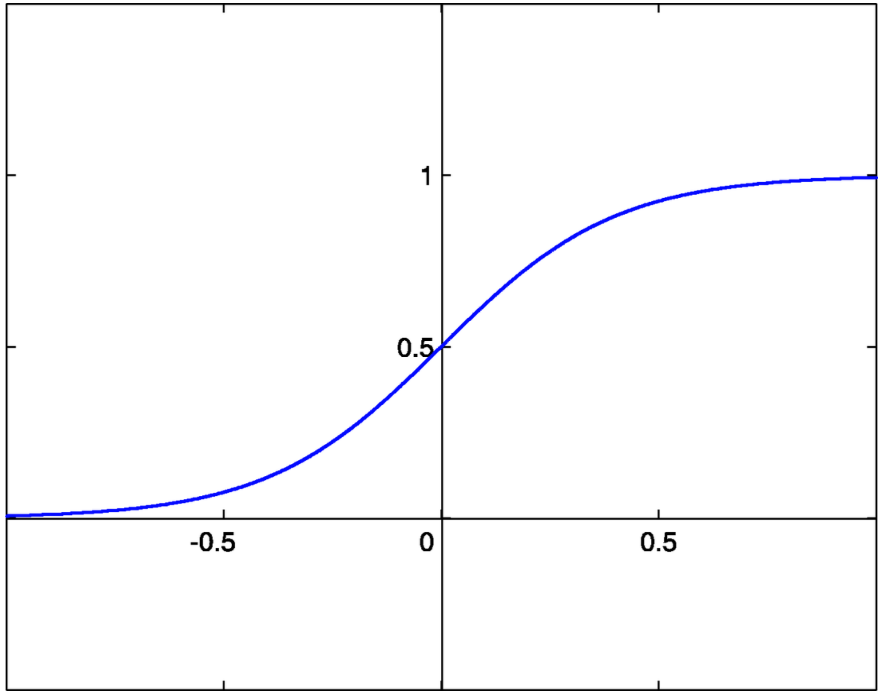
$W_{ij}^1 \rightarrow$ denotes an arbitrary weight between input and first hidden layer

- **Example: Calculating** $\frac{\partial j}{\partial W_{ij}^{1}}$
- **Sigmoid activation function**

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

$\sigma(x)$

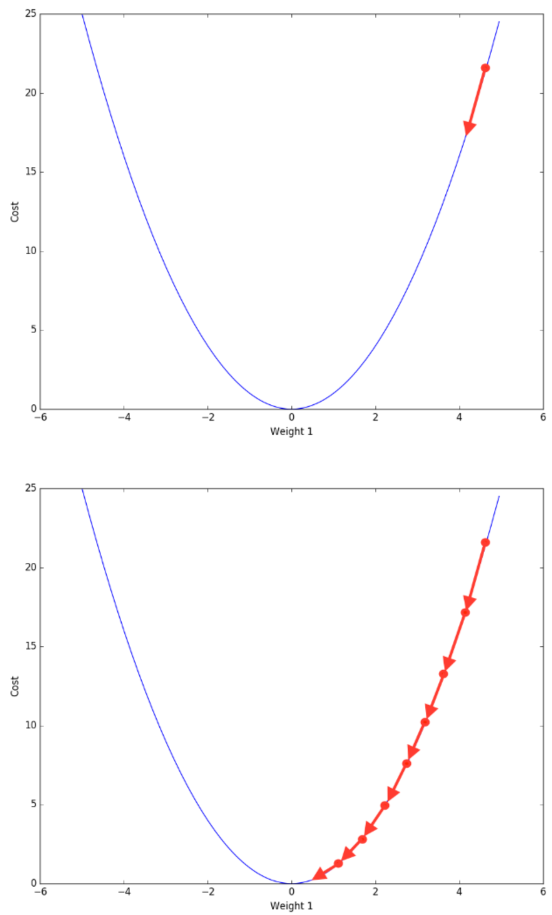
$$\frac{d\sigma(x)}{d(x)} = \sigma(x)(1 - \sigma(x))$$



- **Example: Calculating** $\frac{\partial j}{\partial W_{ij}^{1}}$
- **Error function** $\boldsymbol{J(W)}$
  - Output estimations denoted as vector $\boldsymbol{p}$
  - Actual (target) outputs as vector **a**

$$J = \frac{1}{2}(\boldsymbol{p} - \boldsymbol{a})^2$$

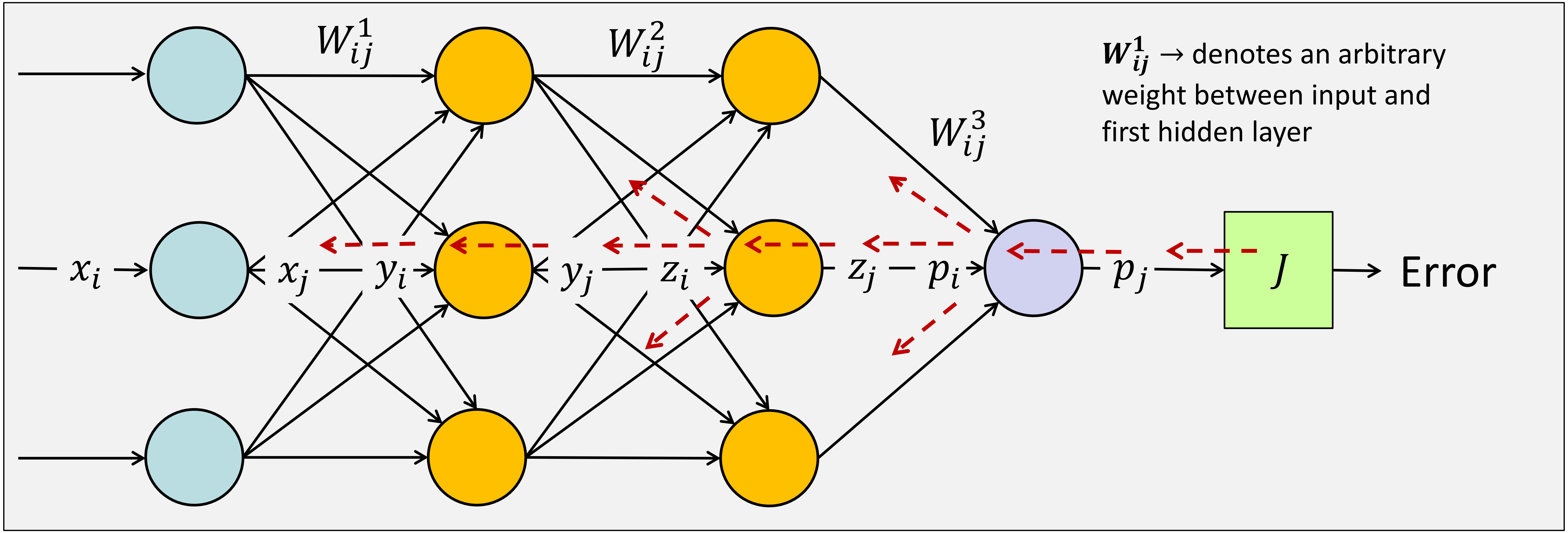
$$\frac{dj}{dp} = (\boldsymbol{p} - \boldsymbol{a})$$



- **Example: Calculating** $\frac{\partial j}{\partial W_{ij}^{1}}$
- **Notation (Generalised)**
  - x_i, x_j - any arbitrary input/output pair for the input layer
    - Note as this is the input: $x_i = x_j$
  - y_i, y_j - any arbitrary input/output pair for hidden layer 1
    - Note: $y_j = \sigma(y_i) = \sigma(\sum_j x_j)$
  - z_i, z_j - any arbitrary input/output pair for hidden layer 2
  - p_i, p_j - input/output pair for output layer
    - Note: sigmoid activation used on output

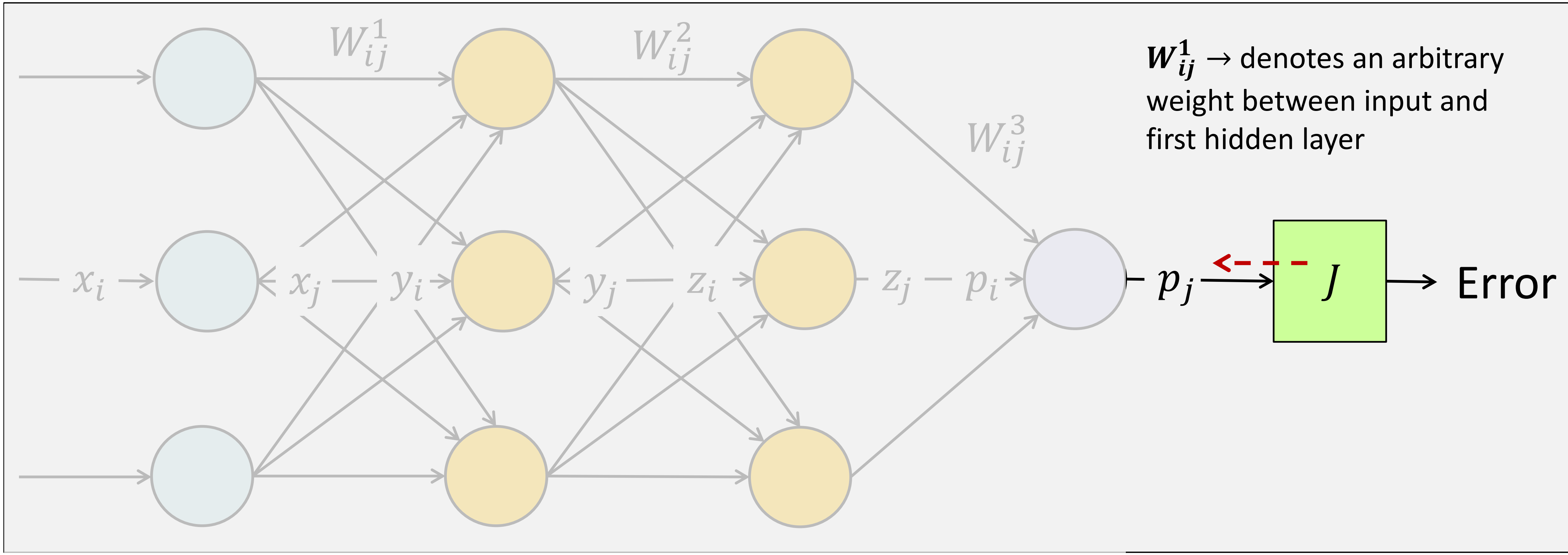
- **Example: Calculating $\frac{\partial j}{\partial W_{ij}^{1}}$**
- **Notation (Generalised)**
  - The weights are organised into three separate variables: $\boldsymbol{W}^1, \boldsymbol{W}^2\ \boldsymbol{W}^3$
    - Each $\boldsymbol{W}$ is a matrix denote all weights at the respective layer
  - W_{ij}^{L} is any single arbitrary weight at a given layer
    - $\boldsymbol{W}_i^L$ denotes all the weights that connect arbitrary neuron i at the preceding layer
    - W_{ij}^{L} is the weight that connects arbitrary neuron i at the preceding layer to an arbitrary neuron j at the next layer.

- **Example: Calculating** $\frac{\partial j}{\partial W_{ij}^{1}}$



$\boldsymbol{W_{ij}^{1}}$ → denotes an arbitrary weight between input and first hidden layer

- **Calculating $\frac{\partial J}{\partial W_{ij}^1}$ → starting at p_j**



$\boldsymbol{W_{ij}^1}$ → denotes an arbitrary weight between input and first hidden layer

Backpropagation

- Starting at $\boldsymbol{p}_j$

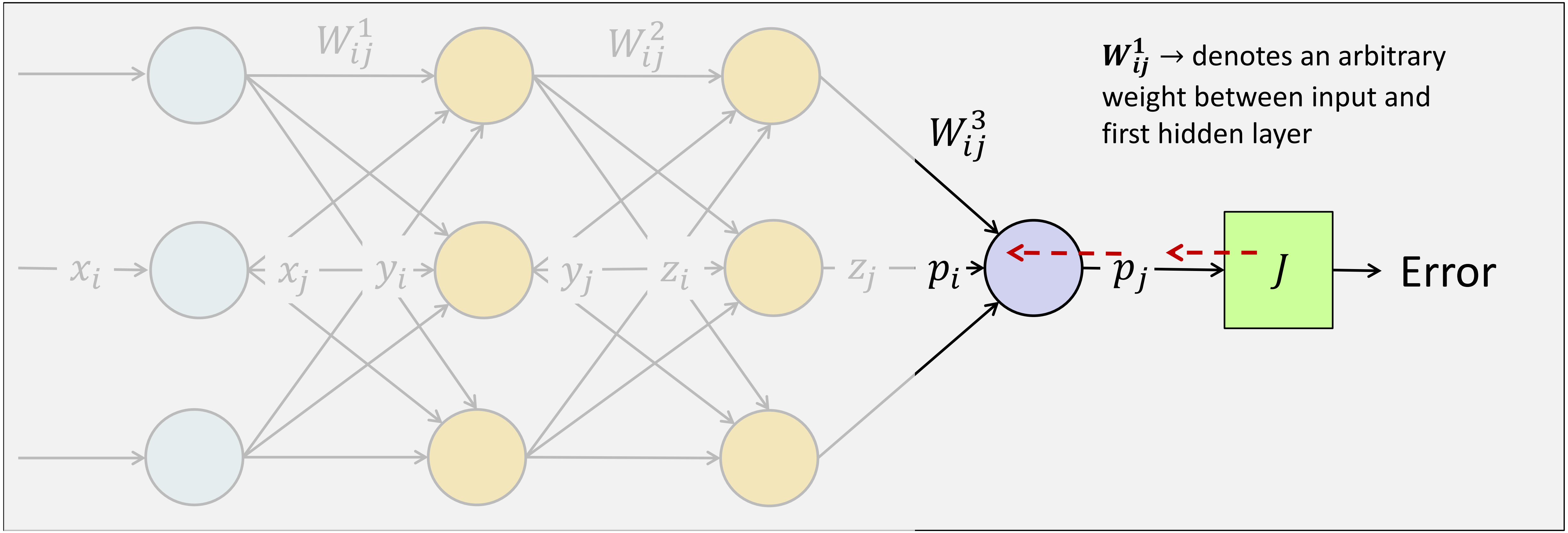
$$\frac{\partial \boldsymbol{J}}{\partial \boldsymbol{W}_{ij}^{1}} = \frac{\partial \boldsymbol{J}}{\partial \boldsymbol{p}_j} \frac{\partial \boldsymbol{p}_j}{\partial \boldsymbol{W}_{ij}^{1}}$$

  - Noting $\frac{dj}{dp} = (\boldsymbol{p} - \boldsymbol{a})$

$$\frac{\partial \boldsymbol{J}}{\partial \boldsymbol{W}_{ij}^{1}} = (\boldsymbol{p}_j - \boldsymbol{a}) \frac{\partial \boldsymbol{p}_j}{\partial \boldsymbol{W}_{ij}^{1}}$$

Backpropagation

- **Calculating $\frac{\partial J}{\partial W_{ij}^1}$ → propagating p_j to p_i**



$\boldsymbol{W_{ij}^1}$ → denotes an arbitrary weight between input and first hidden layer

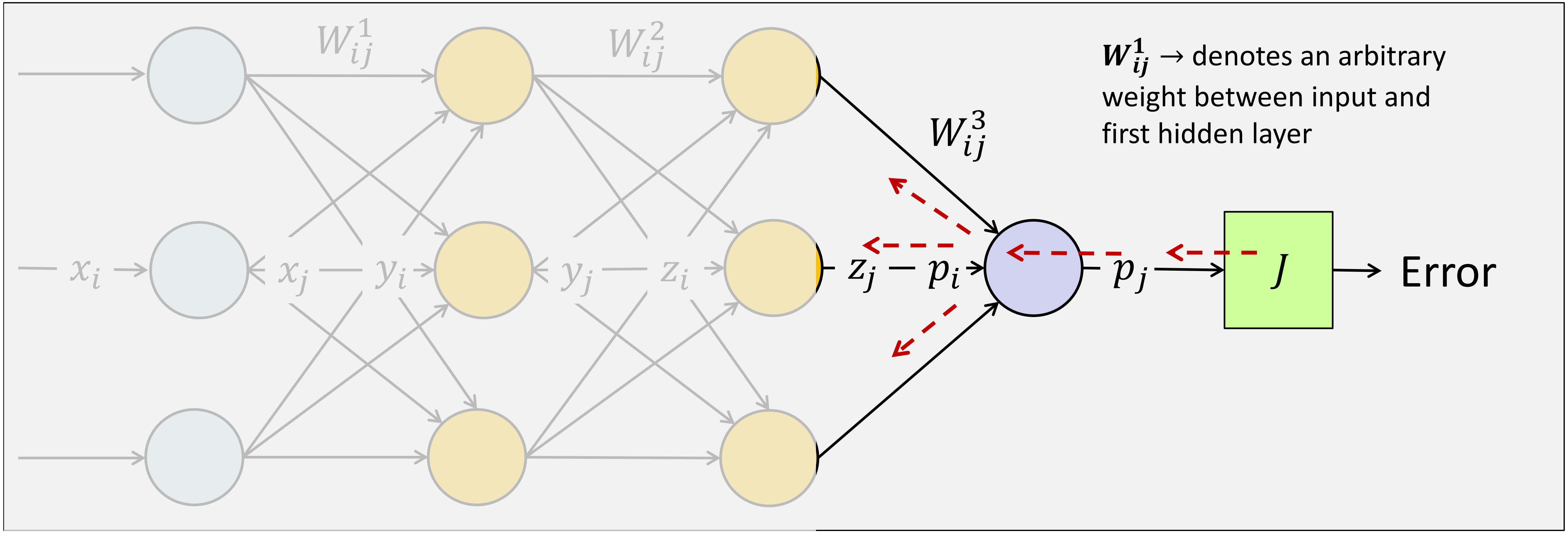
- **Propagating one layer back until to $\boldsymbol{p_i}$**

$$\frac{\partial \boldsymbol{J}}{\partial \boldsymbol{W}_{ij}^{1}} = (\boldsymbol{p}_j - \boldsymbol{a})\frac{\partial \boldsymbol{p}_j}{\partial \boldsymbol{W}_{ij}^{1}} = (\boldsymbol{p}_j - \boldsymbol{a})\frac{\partial \boldsymbol{p}_j}{\partial \boldsymbol{p_i}}\frac{\partial \boldsymbol{p_i}}{\partial \boldsymbol{W}_{ij}^{1}}$$

- Noting $\frac{\partial \boldsymbol{p}_j}{\partial \boldsymbol{p_i}} = \boldsymbol{p}_j(1 - \boldsymbol{p}_j)$ ← Derivative of sigmoid with simplified notation

$$\frac{\partial \boldsymbol{J}}{\partial \boldsymbol{W}_{ij}^{1}} = (\boldsymbol{p}_j - \boldsymbol{a})\boldsymbol{p}_j(1 - \boldsymbol{p}_j)\frac{\partial \boldsymbol{p_i}}{\partial \boldsymbol{W}_{ij}^{1}}$$

- **Calculating $\frac{\partial J}{\partial W^1_{ij}} \rightarrow$ propagating p_i to z_j**



- **Propagating from p_i to z_j**
  - Need to take into account action of the different weights

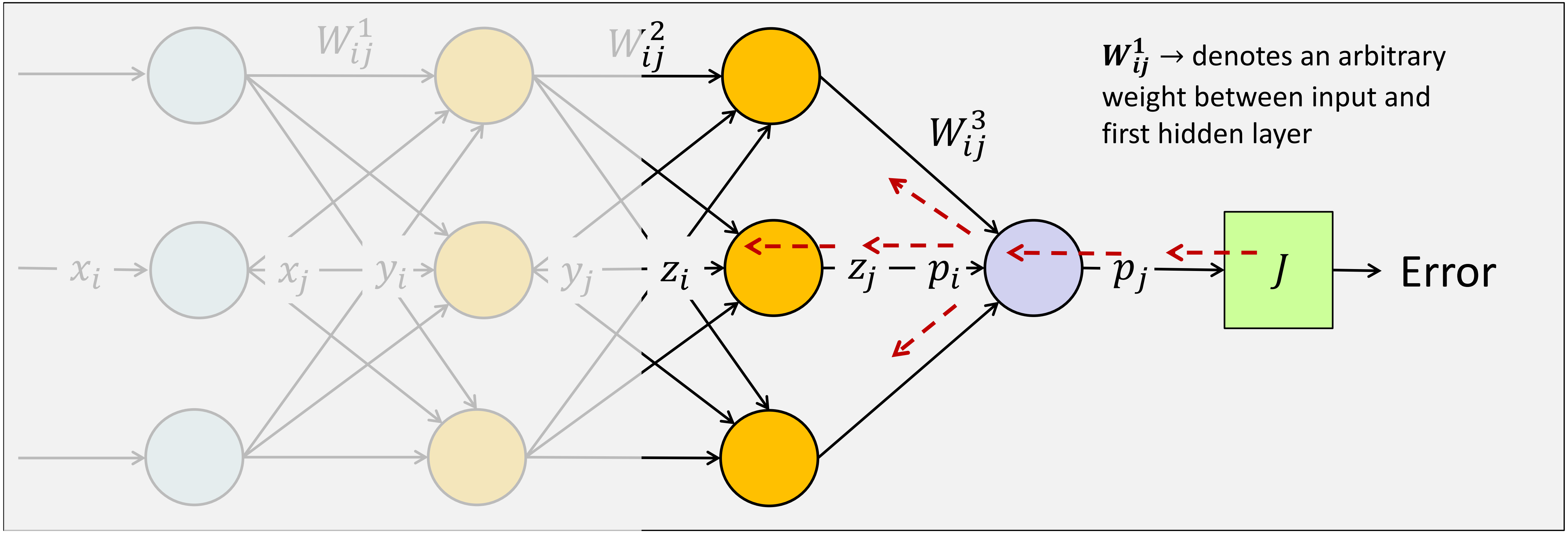
$$\frac{\partial \boldsymbol{p_i}}{\partial \boldsymbol{W}_{ij}^1} = \sum_j \frac{\partial \boldsymbol{p_i}}{\partial \boldsymbol{z}_j} \frac{\partial \boldsymbol{z}_j}{\partial \boldsymbol{W}_{ij}^1}$$

- Noting $\frac{\partial \boldsymbol{p_i}}{\partial \boldsymbol{z}_j} = \boldsymbol{W}_{ij}^3$, note $\frac{\partial \boldsymbol{p_i}}{\partial \boldsymbol{z}_j}$ is derivate wrt any arbitrary $\boldsymbol{z}_j$

$$\frac{\partial \boldsymbol{p_i}}{\partial \boldsymbol{W}_{ij}^1} = \boldsymbol{W}_{ij}^3 \frac{\partial \boldsymbol{z}_j}{\partial \boldsymbol{W}_{ij}^1}$$

Ignoring summation for the moment to make notation easier

- **Calculating $\frac{\partial J}{\partial W_{ij}^{1}}$ → propagating from z_j to z_i**



W_{ij}^1 → denotes an arbitrary weight between input and first hidden layer

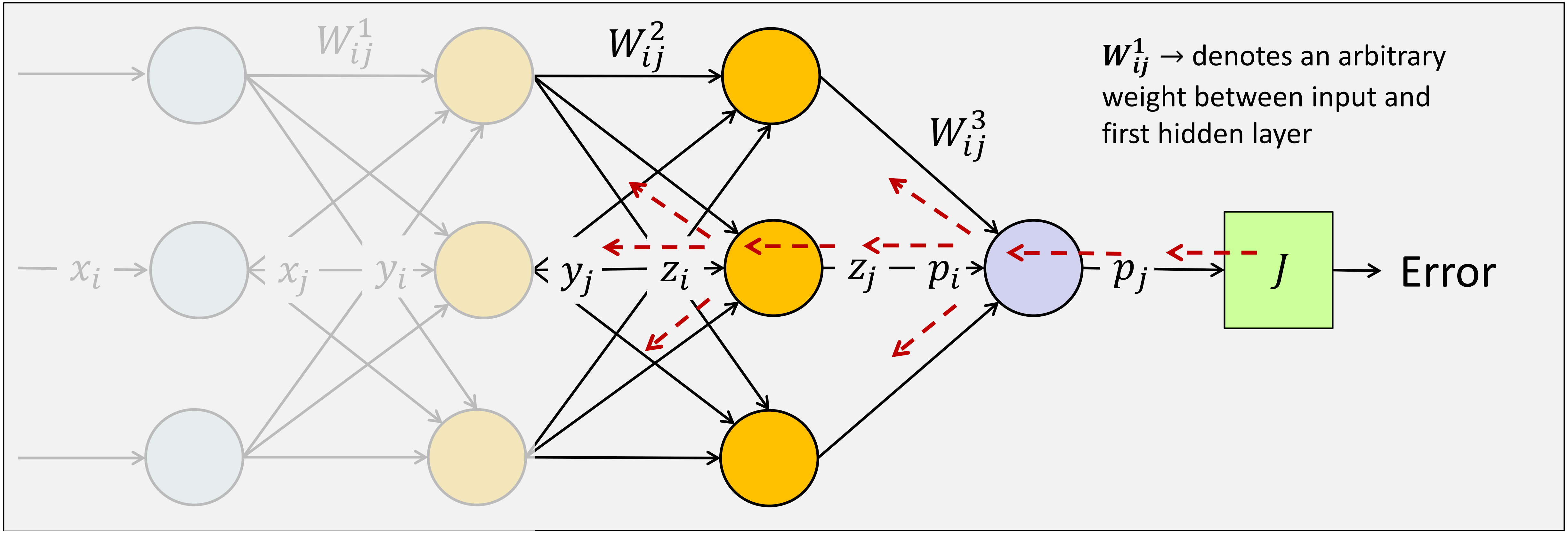
- **Propagating from z_j to z_i**

$$\frac{\partial \boldsymbol{z}_j}{\partial \boldsymbol{W}_{ij}^1} = \frac{\partial \boldsymbol{z}_j}{\partial \boldsymbol{z}_i}\frac{\partial \boldsymbol{z}_i}{\partial \boldsymbol{W}_{ij}^1}$$

  - Noting $\frac{\partial \boldsymbol{z}_j}{\partial \boldsymbol{z}_i} = \boldsymbol{z}_j(1 - \boldsymbol{z}_j)$ ← Derivative of sigmoid with simplified notation

$$\frac{\partial \boldsymbol{z}_j}{\partial \boldsymbol{W}_{ij}^1} = \boldsymbol{z}_j(1 - \boldsymbol{z}_j)\frac{\partial \boldsymbol{z}_i}{\partial \boldsymbol{W}_{ij}^1}$$

- **Calculating $\frac{\partial J}{\partial W_{ij}^{1}}$ → propagating from z_i to y_j**



W_{ij}^1 → denotes an arbitrary weight between input and first hidden layer

- **Propagating from z_i to y_j**
  - Again, need to take into account action of the different weights

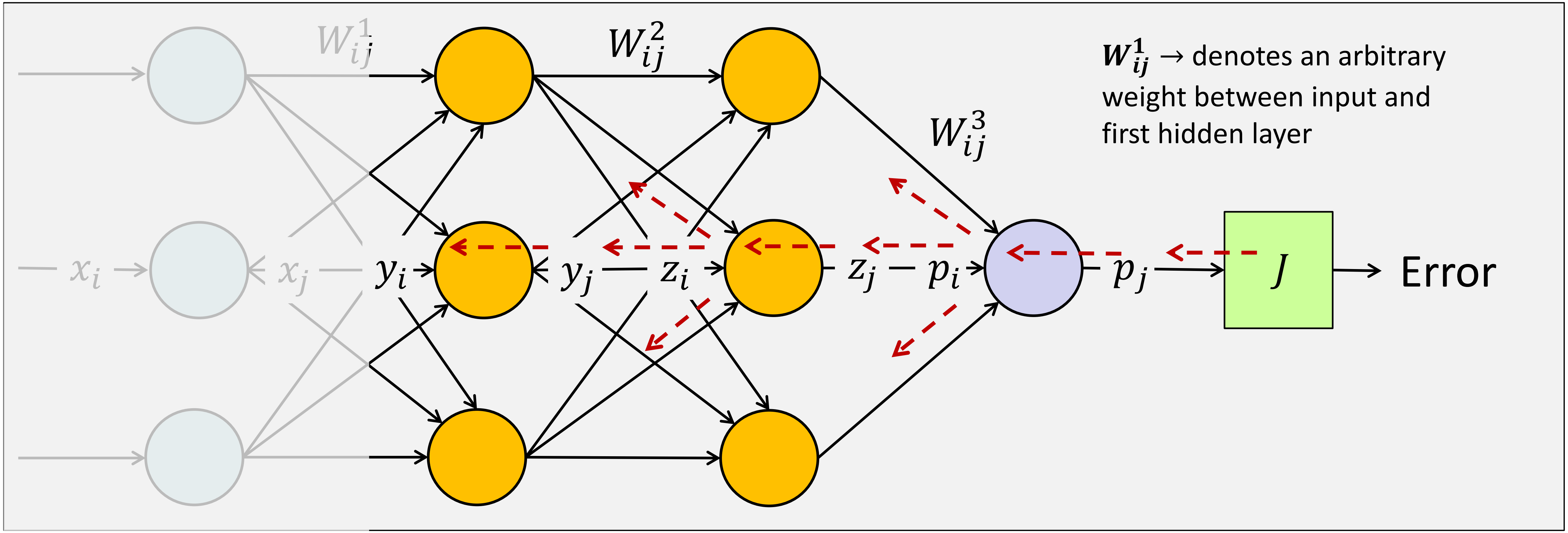
$$\frac{\partial \boldsymbol{z}_i}{\partial \boldsymbol{W}^1_{ij}} = \sum_j \frac{\partial \boldsymbol{z}_i}{\partial \boldsymbol{y}_j} \frac{\partial \boldsymbol{y}_j}{\partial \boldsymbol{W}^1_{ij}}$$

- Noting $\frac{\partial \boldsymbol{z}_i}{\partial \boldsymbol{y}_j} = \boldsymbol{W}^2_{ij}$, note $\frac{\partial \boldsymbol{z}_i}{\partial \boldsymbol{y}_j}$ is derivate wrt any arbitrary $\boldsymbol{y}_j$

$$\frac{\partial \boldsymbol{z}_i}{\partial \boldsymbol{W}^1_{ij}} = \boldsymbol{W}^2_{ij} \frac{\partial \boldsymbol{y}_j}{\partial \boldsymbol{W}^1_{ij}}$$

Ignoring summation for the moment to make notation easier

- **Calculating $\frac{\partial J}{\partial W_{ij}^1}$ → propagating from y_j to y_i**



$\boldsymbol{W_{ij}^1}$ → denotes an arbitrary weight between input and first hidden layer

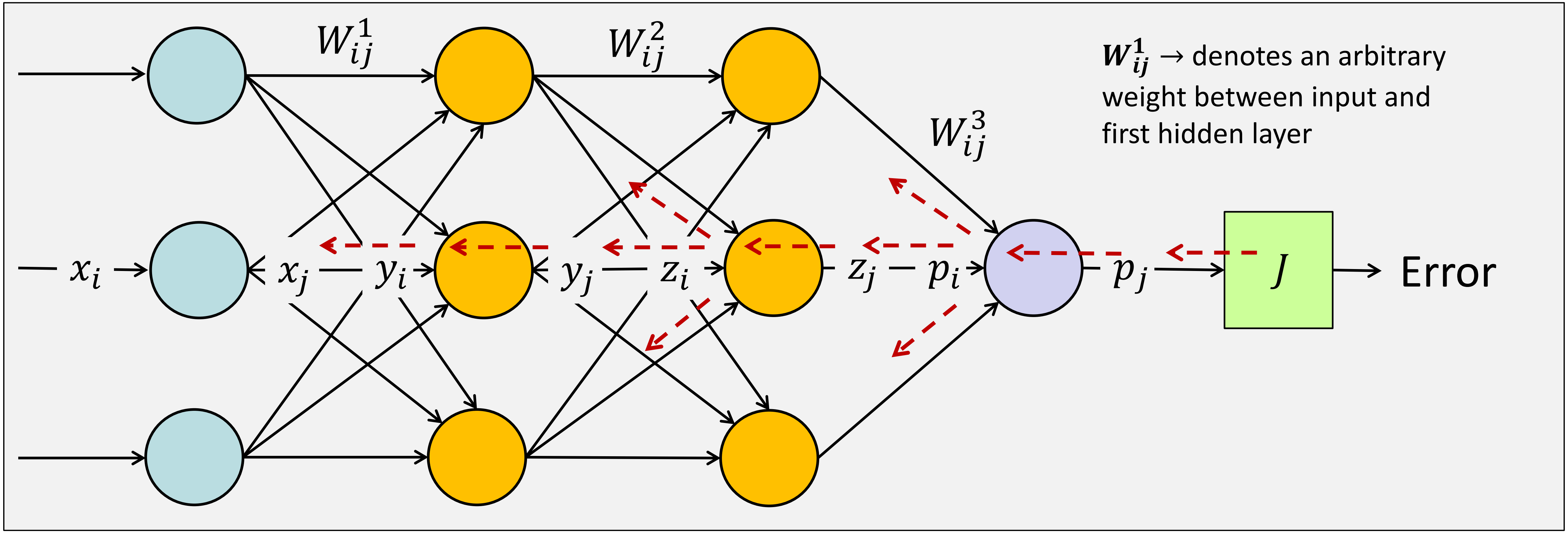
- **Propagating from $\boldsymbol{y}_j$ to $\boldsymbol{y}_i$**

$$\frac{\partial \boldsymbol{y}_j}{\partial \boldsymbol{W}_{ij}^1} = \frac{\partial \boldsymbol{y}_j}{\partial \boldsymbol{y}_i} \frac{\partial \boldsymbol{y}_i}{\partial \boldsymbol{W}_{ij}^1}$$

– Noting $\frac{\partial \boldsymbol{y}_j}{\partial \boldsymbol{y}_i} = \boldsymbol{y}_j(1 - \boldsymbol{y}_j)$ ← Derivative of sigmoid with simplified notation

$$\frac{\partial \boldsymbol{y}_j}{\partial \boldsymbol{W}_{ij}^1} = \boldsymbol{y}_j(1 - \boldsymbol{y}_j) \frac{\partial \boldsymbol{y}_i}{\partial \boldsymbol{W}_{ij}^1}$$

- **Calculating $\frac{\partial J}{\partial W^1_{ij}}$ → propagating from y_i to x_j**



W^1_{ij} → denotes an arbitrary weight between input and first hidden layer

- **Propagating from y_i to x_j**
  - Have reached $\boldsymbol{W}_{ij}^{1}$ $\rightarrow$ what we are backpropagating to

$$\frac{\partial \boldsymbol{y}_j}{\partial \boldsymbol{W}_{ij}^{1}} = \sum_j \boldsymbol{x}_j$$

$\sum_j \boldsymbol{x}_j$ is a numerical value so cannot be derived further

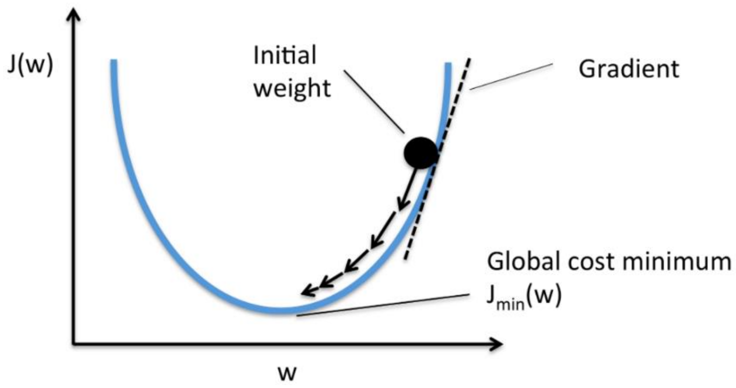
  - So putting it all together:

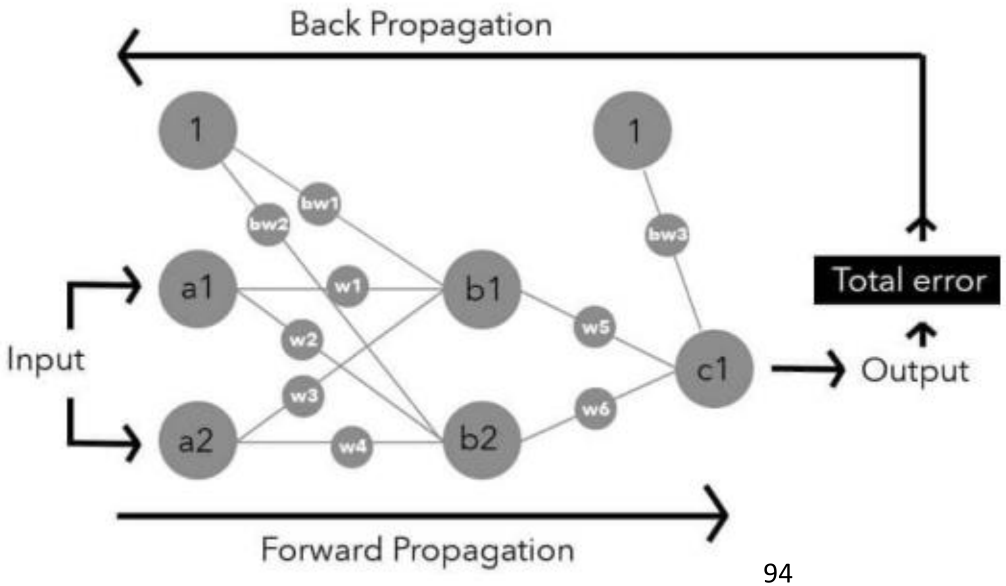
$$\frac{\partial \boldsymbol{J}}{\partial \boldsymbol{W}_{ij}^{1}} = (\boldsymbol{p}_j - \boldsymbol{a})\boldsymbol{p}_j(1 - \boldsymbol{p}_j) \sum_j \left[\boldsymbol{W}_{ij}^{3} \boldsymbol{z}_j (1 - \boldsymbol{z}_j) \sum_j \left[\boldsymbol{W}_{ij}^{2} \boldsymbol{y}_j (1 - \boldsymbol{y}_j) \sum_j \boldsymbol{x}_j \right] \right]$$

Today's Lecture

Image source: https://medium.com

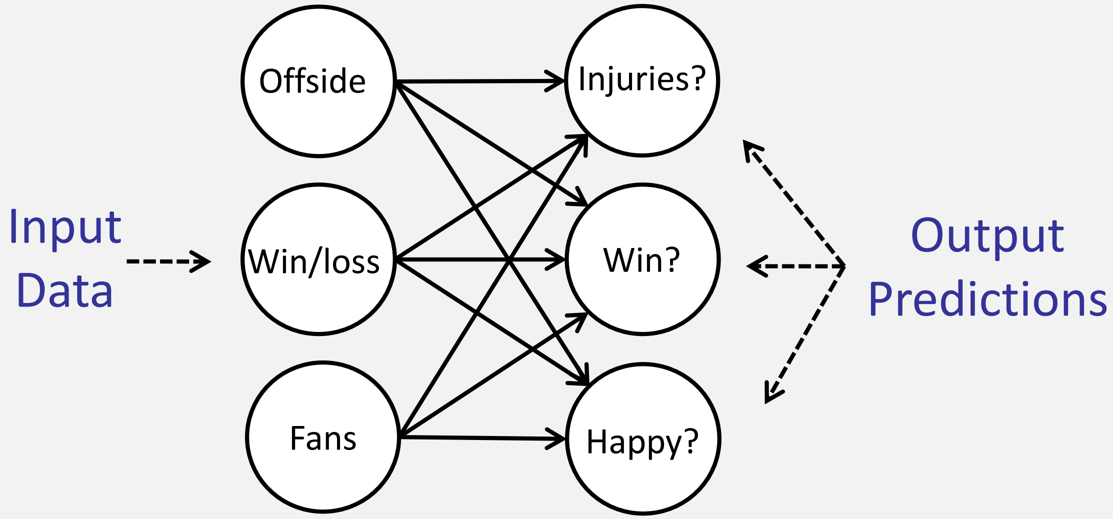
- Gradient Descent
  - Networks with multiple inputs
  - Networks with multiple outputs
  - Networks with multiple inputs and outputs
- Correlation
  - Learning Correlation
  - Creating Correlation
- Backpropagation
- Summary





- **Gradient descent generalies to arbitrarily large networks**

Empty network with multiple inputs and outputs



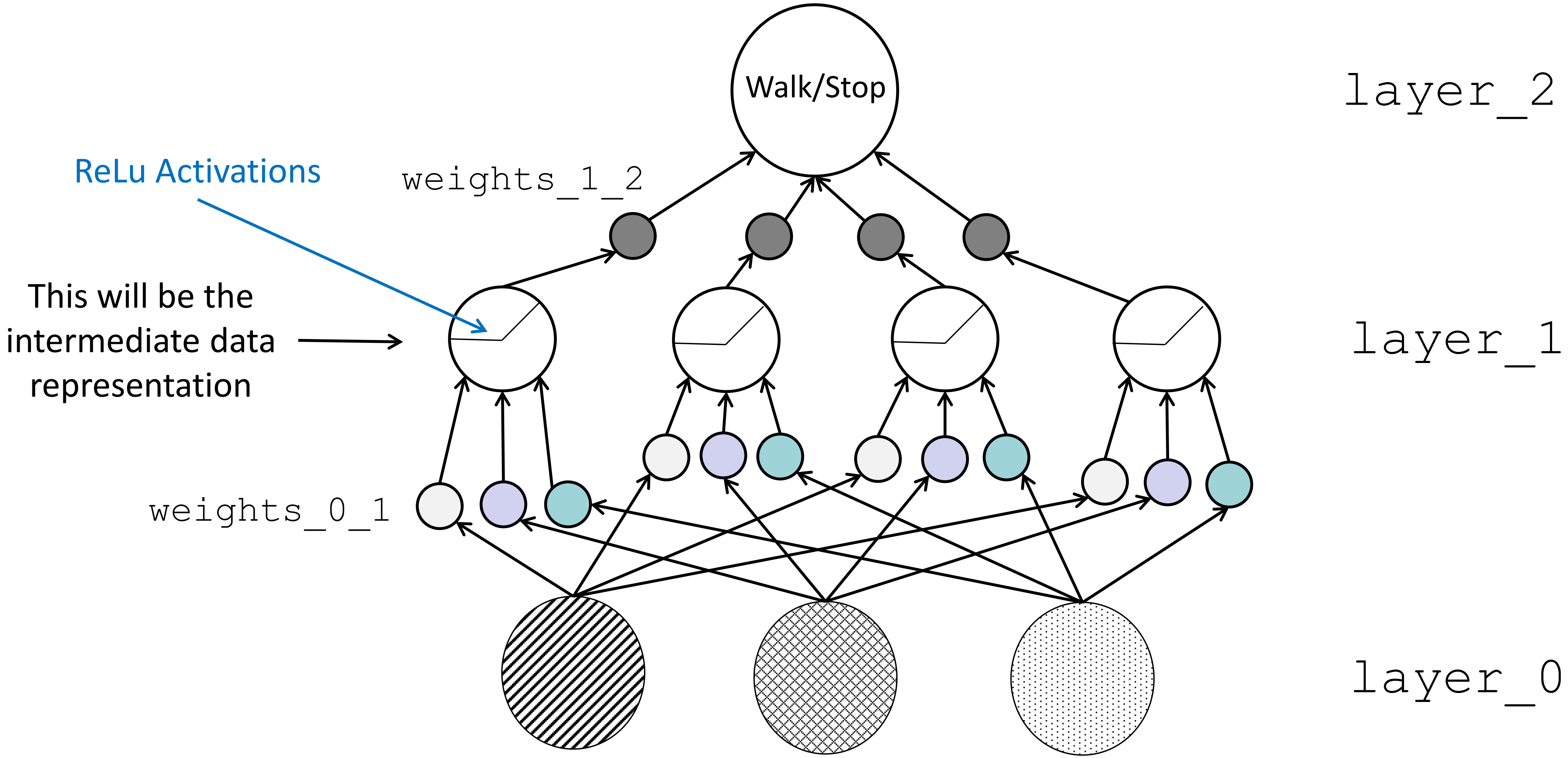

```
pred = neural_network(input,weights)
error = [0, 0, 0]
delta = [0, 0, 0]
for i in range(len(true)):
    error[i] = (pred[i] - true[i]) ** 2
    delta = pred[i] - true[i]

weight_deltas = outer_prod(input,delta)

for i in range(len(weights)):
    for j in range(len(weights[0])):
        weights[i][j] -= alpha * \ weight_deltas[i][j]
```

Summary

- **Weights learn correlations between input and output**
  - Prediction is a weighted sum of the inputs – the dot product
  - During training
    - The learning algorithm rewards inputs that correlate with the output with upward pressure on their weight
      - Drives the weights towards 1
    - At the same time, the learning algorithm penalises inputs with discorrelation with downward pressure
      - Drives the weights towards 0
  - Learning rewards correlation with larger weights
  - Learning finds correlation between the two datasets

- **Creating Correlation**
  - To learn when there is no correlation, just use more layers
    - First layer creates a layer with limited correlation with the output
    - Second layer uses this limited correlation to correctly predict the output
  - Hidden layer(s) can be thought of as creating intermediate dataset(s) that has correlation with the output
  - Stacking linear neural networks does not give and more power
    - A more computationally expensive version of a single weighted sum
    - Use non-linear activation functions to induce conditional correlation between layer

Summary



Walk/Stop
layer_2
ReLu Activations
weights_1_2
This will be the intermediate data representation
layer_1
weights_0_1
layer_0

- **Backpropagation**
  - Update weights to minimise loss function
    - This is achieved by taking the gradient of loss function with respect to the weights
  - Not a trivial process as neural networks are a series of layers
    - The output is a composite function of the weights, inputs, and activation functions
  - Backpropagation is a tool to calculate the gradient of the loss function with respect to any single weight in a network
    - Allows us to calculate the gradient at all weights
    - This is achieved through iterative applications of the chain rule

Coming Up

- **Next week**
  - Loss functions
    - The difference between what was predicted and what it should have been predicted
  - Activation functions
    - A function applied to neurons in a layer during prediction
  - Optimisers
    - Different algorithms for updating neural network weights
  - Recurrent Neural Networks
    - Model *sequences* of data