AAU
AAiT
SiTE

Course Title: **Deep Learning**

Credit Hour: 3

Instructor: Fantahun B. (PhD) ✉ meetfantaai@gmail.com

Office: NB #

Ch-1B Deep Neural Networks

Mar-2023, AA

Deep Neural Networks

Agenda:

- Gradient Based Learning
- Output Units
- Hidden Units
- Architecture Design
- Backpropagation

Deep Neural Networks

Objectives

After completing this chapter students will be able to:

- Gradient Based Learning
- Identify and differentiate Output Units
- Identify Hidden Units
- Discuss Architecture Design considerations
- Demonstrate how the Backpropagation (backprop) algorithm works.

Deep Neural Networks: Back-Propagation

- In feedforward neural networks which accept an input x and produce an output $\hat{y}$, information flows forward through the network.
- In forward propagation, the inputs x provide the initial information that then propagates up to the hidden units at each layer and finally produces $\hat{y}$. During training, forward propagation can continue onward until it produces a scalar cost $J(\theta)$.
- The back-propagation algorithm (Rumelhart et al., 1986a), often simply called backprop, allows the information from the cost to then flow backwards through the network, in order to compute the gradient.
- Backpropagation, or backward propagation of errors, is an algorithm that is designed to test for errors working back from output nodes to input nodes

Deep Neural Networks: Back-Propagation

- Computing an analytical expression for the gradient is straightforward, but numerically evaluating such an expression can be computationally expensive. The back-propagation algorithm does so using a simple and inexpensive procedure.
- The term back-propagation is often misunderstood as meaning the whole learning algorithm for multi-layer neural networks.
- Actually, back-propagation refers only to the method for computing the gradient, while another algorithm, such as stochastic gradient descent, is used to perform learning using this gradient.
- Furthermore, back-propagation is often misunderstood as being specific to multilayer neural networks, but in principle it can compute derivatives of any function (or report undefined).

Deep Neural Networks: Back-Propagation
1- Computational Graphs

- So far we have discussed neural networks with a relatively informal graph language.
- To describe the back-propagation algorithm more precisely, it is helpful to have a more precise computational graph language.
- Many ways of formalizing computation as graphs are possible.
- Here, we use each node in the graph to indicate a variable. The variable may be a scalar, vector, matrix, tensor, or even a variable of another type.
- We also use an operation, a simple function of one or more variables.

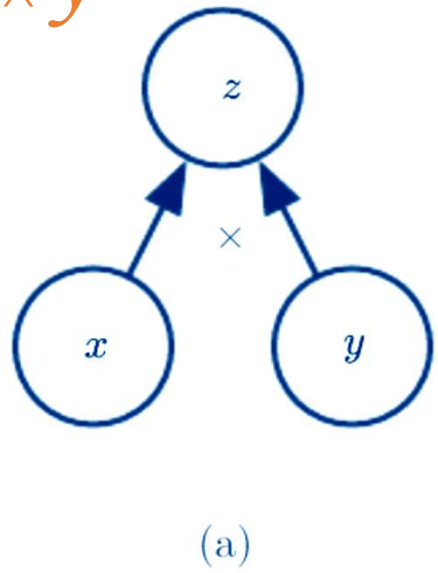
Deep Neural Networks: Back-Propagation
1- Computational Graphs

- Functions more complicated than the operations in this set may be described by composing many operations together.
- If a variable y is computed by applying an operation to a variable x, then we draw a directed edge from x to y.
- We sometimes annotate the output node with the name of the operation applied, and other times omit this label when the operation is clear from context.

Deep Neural Networks: Back-Propagation

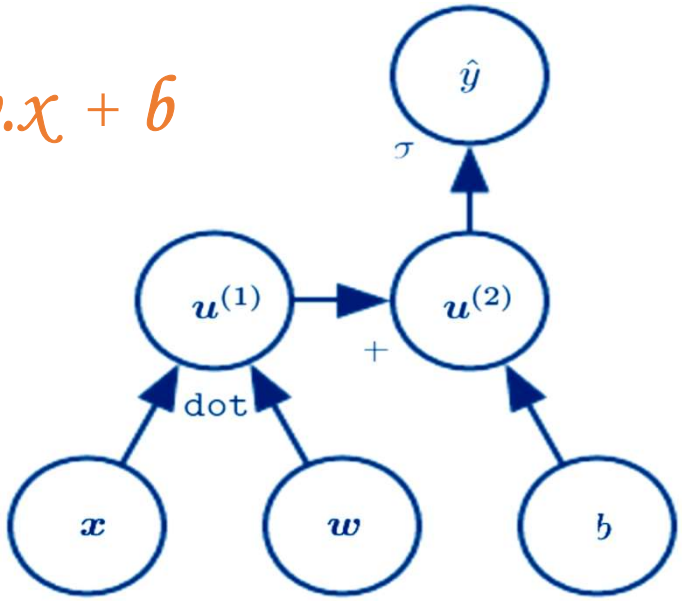
1- Computational Graphs: Examples

$Z = X \times Y$



(a)

$\hat{Y} = w.x + b$

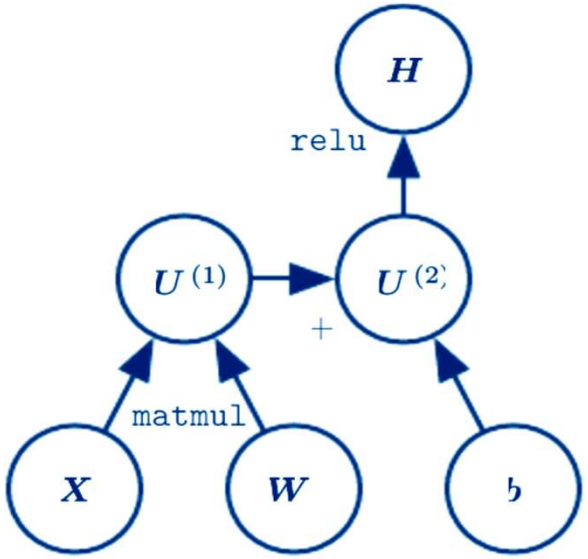


(b)

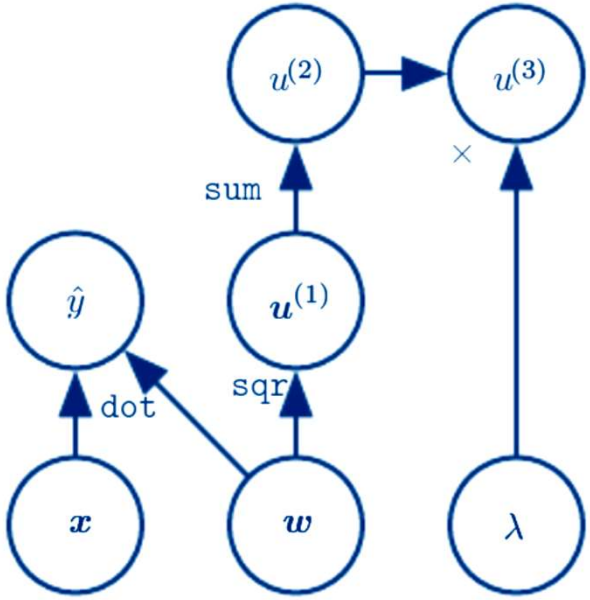
Figure 6.8: Examples of computational graphs. (a) The graph using the x operation to compute z = xy. (b) The graph for the logistic regression prediction $\hat{y} = \sigma(x^T w + b)$. Some of the intermediate expressions do not have names in the algebraic expression but need names in the graph. We simply name the i-th such variable $u^{(i)}$.

Deep Neural Networks: Back-Propagation

1- Computational Graphs: Examples



(c)



(d)

(c) The computational graph for the expression H = max{0, XW + b}, which computes a design matrix of rectified linear unit activations H given a design matrix containing a minibatch of inputs X. (d) A computation graph that applies more than one operation to the weights w of a linear regression model. The weights are used to make both the prediction ŷ and the weight decay penalty $\lambda \sum_i w_i^2$.

.

Deep Neural Networks: Back-Propagation

2- Chain Rule of Calculus

- The chain rule of calculus is used to compute the derivatives of functions formed by composing other functions whose derivatives are known.
- Back-propagation is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient.
- Let x be a real number, and let f and g both be functions mapping from a real number to a real number. Suppose that

  $y = g(x)$ and $z = f(g(x)) = f(y)$.

Then the chain rule states that,

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{6.44}$$

Deep Neural Networks: Back-Propagation

2- Chain Rule of Calculus

- We can generalize this beyond the scalar case. Suppose that $x \in R^m$, $y \in R^n$, g maps from R^m to R^n , and f maps from R^n to R .
- If $y = g(x)$ and $z = f(y)$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}. \tag{6.45}$$

- In vector notation, this may be equivalently written as

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}} z, \tag{6.46}$$

where $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the $n \times m$ Jacobian matrix of g.

Deep Neural Networks: Back-Propagation

2- Chain Rule of Calculus

- Usually we do not apply the back-propagation algorithm merely to vectors, but rather to tensors of arbitrary dimensionality.
- Conceptually, this is exactly the same as back-propagation with vectors. The only difference is how the numbers are arranged in a grid to form a tensor.
- We could imagine flattening each tensor into a vector before we run back-propagation, computing a vector-valued gradient, and then reshaping the gradient back into a tensor.
- In this rearranged view, back-propagation is still just multiplying Jacobians by gradients.

Deep Neural Networks: Back-Propagation

2- Chain Rule of Calculus

- To denote the gradient of a value z with respect to a tensor **X** , we write $\nabla_{\mathbf{X}} z$, just as if **X** were a vector.
- The indices into **X** now have multiple coordinates—for example, a 3-D tensor is indexed by three coordinates. We can abstract this away by using a single variable i to represent the complete tuple of indices.
- For all possible index tuples i, $(\nabla_{\mathbf{X}} z)_i$ gives $\partial z / \partial \mathbf{X}_i$. This is exactly the same as how for all possible integer indices i into a vector, $(\nabla_{\mathbf{x}} z)_i$ gives $\partial z / \partial x_i$.
- Using this notation, we can write the chain rule as it applies to tensors. If **Y** = g(**X**) and z = f(**Y**), then

$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} Y_j) \frac{\partial z}{\partial Y_j}. \tag{6.47}$$

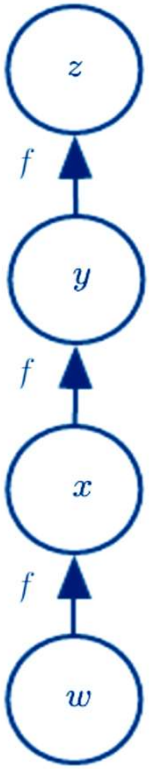
Deep Neural Networks: Back-Propagation

3-Recursively Applying the Chain Rule to Obtain Backprop

- Using the chain rule, it is straightforward to write down an algebraic expression for the gradient of a scalar with respect to any node in the computational graph that produced that scalar. However, actually evaluating that expression in a computer introduces some extra considerations.
- Specifically, many subexpressions may be repeated several times within the overall expression for the gradient.
- Any procedure that computes the gradient will need to choose whether to store these subexpressions or to re-compute them several times.
- An example of how these repeated subexpressions arise is given in Fig. 6.9.

Deep Neural Networks: Back-Propagation

3-Recursively Applying the Chain Rule to Obtain Backprop



- Let w ∈ R be the input to the graph. We use the same function f : R → R as the operation that we apply at every step of a chain: x = f(w), y = f(x), z = f(y). To compute ∂z/∂w , we apply Eq. 6.44 and obtain:

$$\frac{\partial z}{\partial w} \tag{6.50}$$

$$= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \tag{6.51}$$

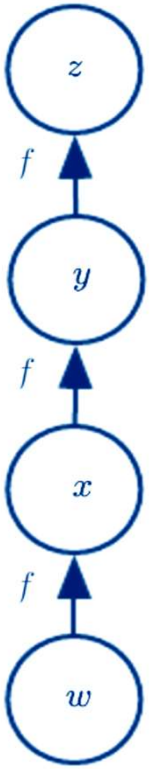
$$= f'(y)f'(x)f'(w) \tag{6.52}$$

$$= f'(f(f(w)))f'(f(w))f'(w) \tag{6.53}$$

Figure 6.9: A computational graph that results in repeated subexpressions when computing the gradient.

Deep Neural Networks: Back-Propagation

3-Recursively Applying the Chain Rule to Obtain Backprop



- Let w ∈ R be the input to the graph. We use the same function f : R → R as the operation that we apply at every step of a chain: x = f(w), y = f(x), z = f(y). To compute ∂z/∂w , we apply Eq. 6.44 and obtain:

$$\frac{\partial z}{\partial w} \tag{6.50}$$

$$= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \tag{6.51}$$

$$= f'(y)f'(x)f'(w) \tag{6.52}$$

$$= f'(f(f(w)))f'(f(w))f'(w) \tag{6.53}$$

Backprop does not re-compute this

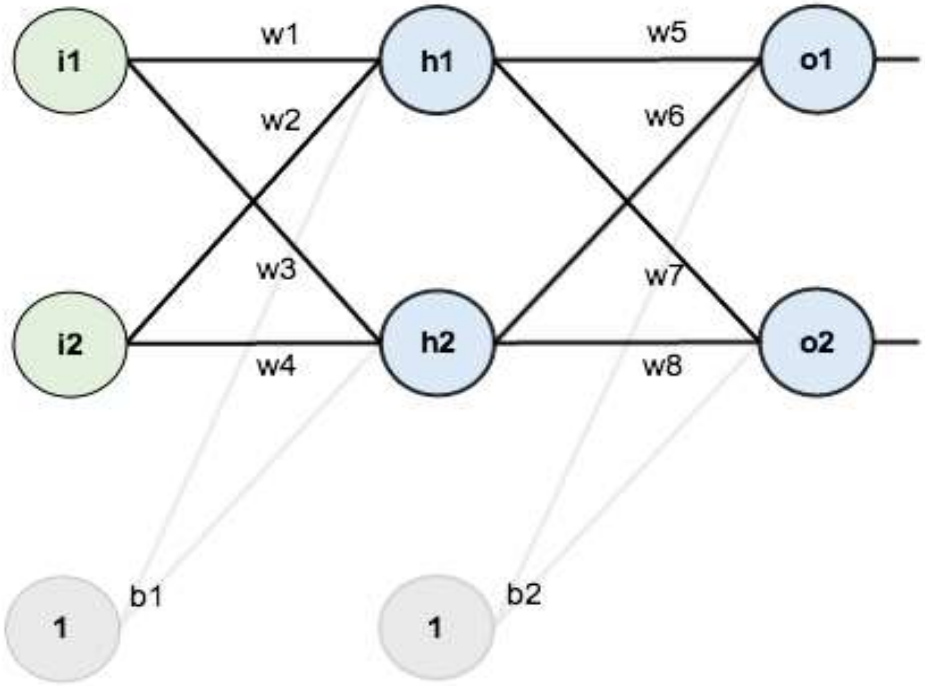
Figure 6.9: A computational graph that results in repeated subexpressions when computing the gradient.

Deep Neural Networks: Back-Propagation
3-Recursively Applying the Chain Rule to Obtain Backprop

- Eq. 6.52 suggests an implementation in which we compute the value of f(w) only once and store it in the variable x. This is the approach taken by the back-propagation algorithm.
- An alternative approach is suggested by Eq. 6.53, where the subexpression f(w) appears more than once. In the alternative approach, f(w) is recomputed each time it is needed.
- When the memory required to store the value of these expressions is low, the back-propagation approach of Eq. 6.52 is clearly preferable because of its reduced runtime.
- However, Eq. 6.53 is also a valid implementation of the chain rule, and is useful when memory is limited.

Deep Neural Networks: Back-Propagation
4-Example implementation: the forward pass

- Consider the following a neural network with two inputs, two hidden neurons, two output neurons, and a bias.

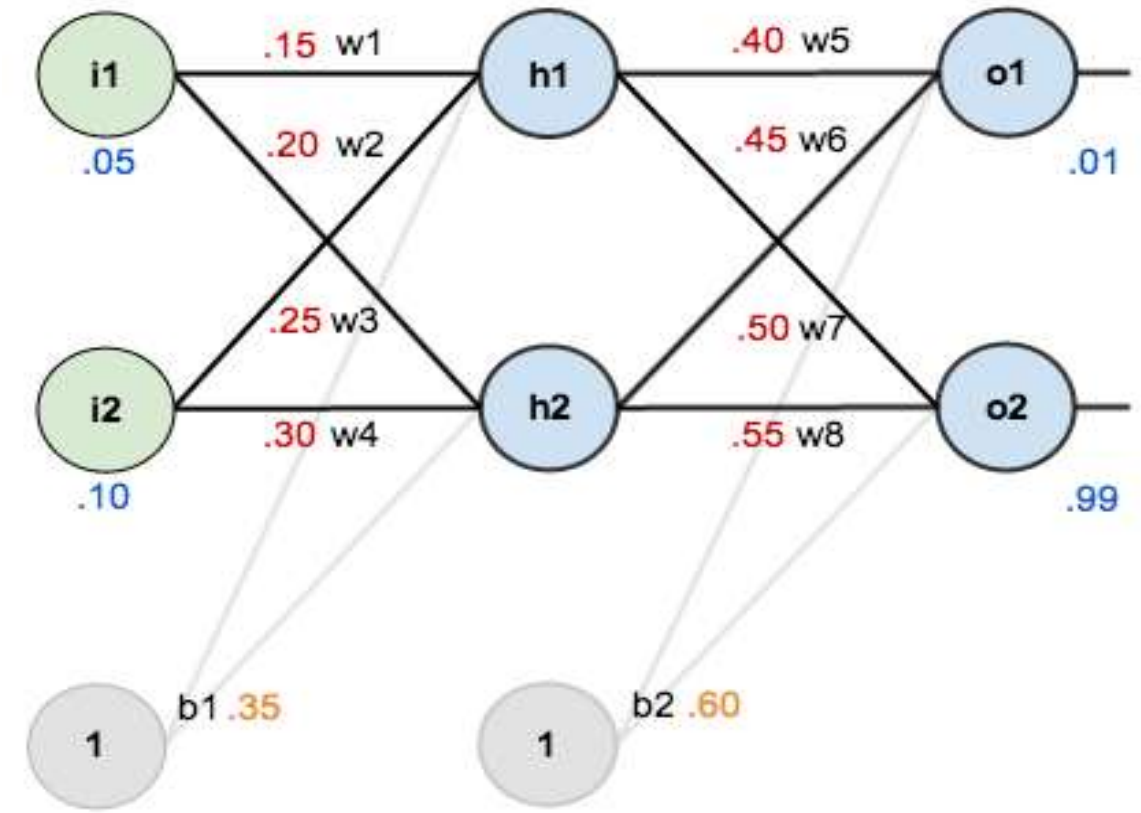


Deep Neural Networks: Back-Propagation
4-Example implementation: the forward pass

- Take these example inputs and plug them in the neural network.

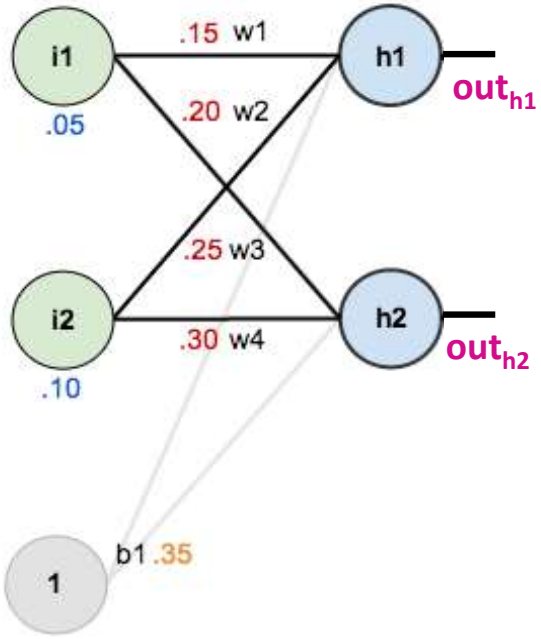
Inputs	Input weights		Hidden weights		Target
	i_1	i_2	h_1	h_2	
0.05	w1=0.15	w3=0.25	w5=0.40	w7=0.50	O_1 = 0.01
0.10	w2=0.20	w4=0.30	w6=0.45	w8=0.55	O_2 = 0.99
Bias = 1	b1=0.35 b2=0.60				

Deep Neural Networks: Back-Propagation
4-Example implementation



Deep Neural Networks: Back-Propagation
4-Example implementation: the forward pass

- Calculate the total net input for h1 and h2 and squash them with the logistic function to get the outputs of h1 and h2:



$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

We then squash it using the logistic function to get the output of h_1:

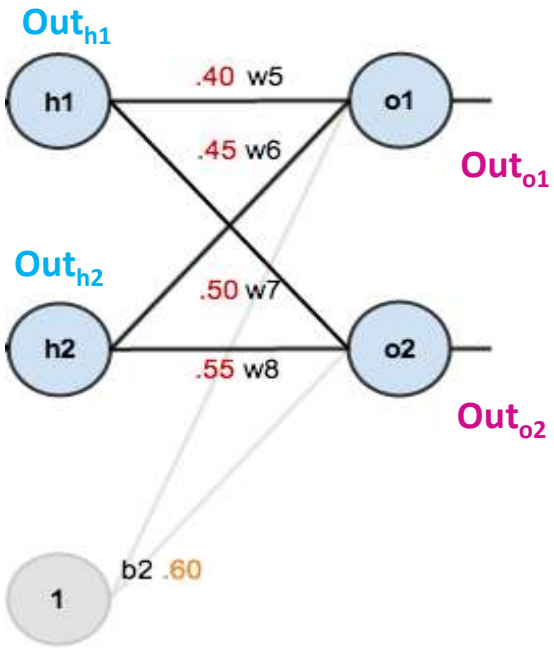
$$out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = \boxed{0.593269992}$$

Carrying out the same process for h_2 we get:

$$out_{h2} = \boxed{0.596884378}$$

Deep Neural Networks: Back-Propagation
4-Example implementation: the forward pass

- Repeat the previous process for the output layer neurons, using outputs from hidden layer neurons as inputs.



$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$$

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = \boxed{0.75136507}$$

And carrying out the same process for o_2 we get:

$$out_{o2} = \boxed{0.772928465}$$

Deep Neural Networks: Back-Propagation
4-Example implementation: the forward pass

- Calculate total error: sum-squared error, you can also use other error functions

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

outputs	predicted	target
1	0.751365070	0.01
2	0.772928465	0.99

$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2 = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$

- Repeating the same process for o_2, we get $E_{o2} = 0.023560026$
- The total error for the neural network is the sum of these errors.

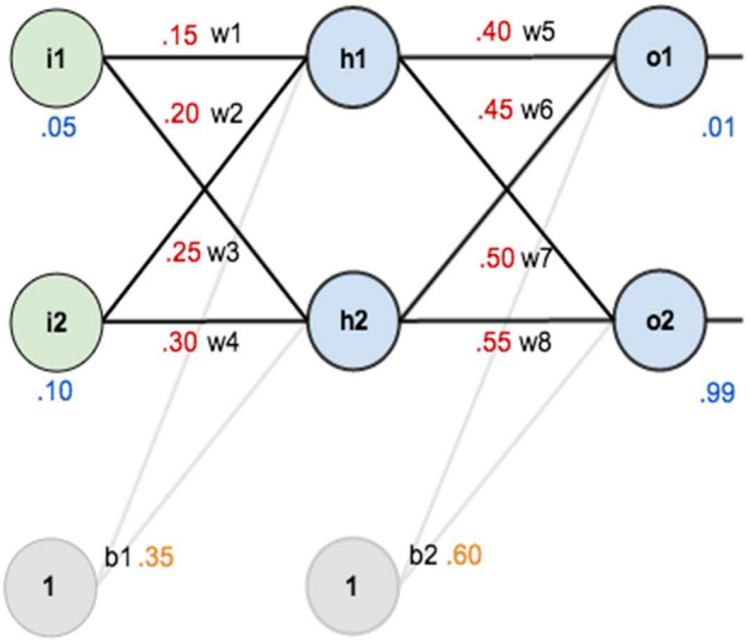
$$E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = \boxed{0.298371109}$$

Deep Neural Networks: Back-Propagation
4-Example implementation: the backward pass
Output Layer

- Consider w_5. We want to know how much a change in w_5 affects the total error, $\frac{\partial E_{total}}{\partial w_5}$ ("the partial derivative of E_{total} with respect to w_5")
- Applying the chain rule:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

- We need to figure out each piece in this equation.



Deep Neural Networks: Back-Propagation
4-Example implementation: the backward pass

Output Layer

$$\frac{\partial E_{total}}{\partial w_5} = \boxed{\frac{\partial E_{total}}{\partial out_{o1}}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

1. How much does the total error change with respect to the out_{o1}?

$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2-1} * -1 + 0$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = \boxed{0.74136507}$$

- Note: *–(target–out) = out–target*

Deep Neural Networks: Back-Propagation
4-Example: the backward pass

Output Layer

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \boxed{\frac{\partial out_{o1}}{\partial net_{o1}}} * \frac{\partial net_{o1}}{\partial w_5}$$

2. How much does the out_{o1} change with respect to its total net input?

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}}$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = \boxed{0.186815602}$$

- Note:

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

$$\frac{\mathrm{d}}{\mathrm{d}x}(f(x)) = \frac{e^x \cdot (1+e^x) - e^x \cdot e^x}{(1+e^x)^2} = \frac{e^x}{(1+e^x)^2} = f(x)\big(1 - f(x)\big)$$

Deep Neural Networks: Back-Propagation
4-Example: the backward pass

Output Layer

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \boxed{\frac{\partial net_{o1}}{\partial w_5}}$$

3. How much does the total net input of o_1 change with respect to w_5?

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1} = \boxed{0.593269992}$$

Deep Neural Networks: Back-Propagation
4-Example implementation: the backward pass

Output Layer

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

4. Putting it all together, we get:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$\frac{\partial E_{total}}{\partial w_5} = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$$

Deep Neural Networks: Back-Propagation
4-Example: the backward pass

Output Layer

5. To decrease the error, we then subtract this value from the current weight (optionally multiplied by some learning rate, eta, we chose it to be 0.5 for now):

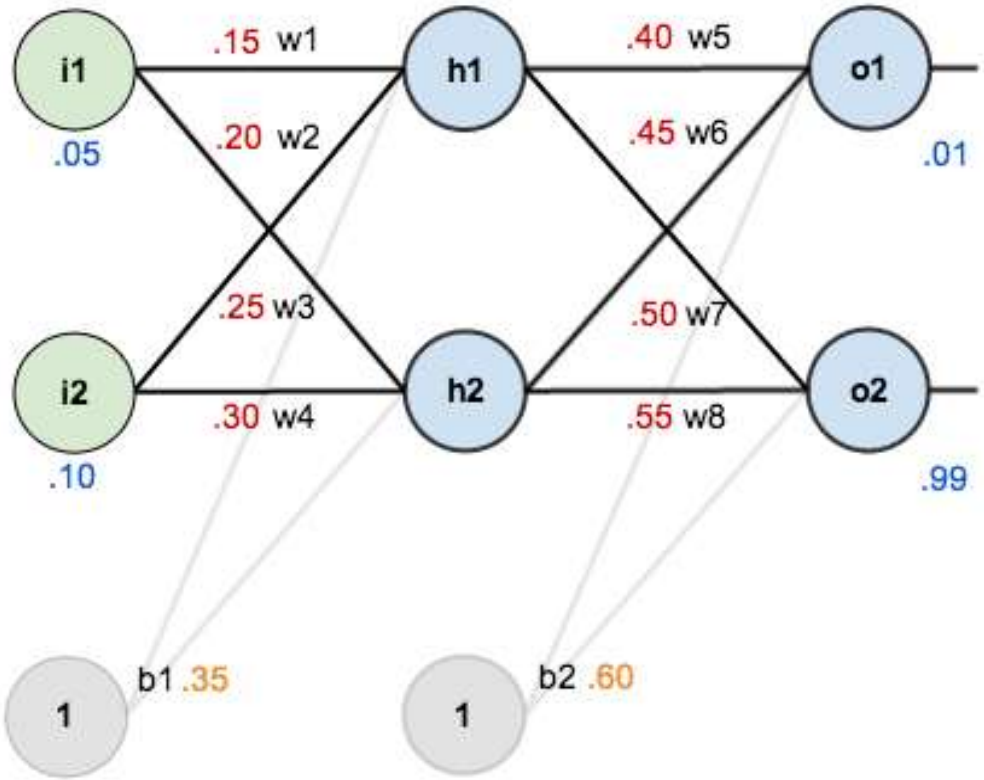
$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

6. We can repeat this process to get the new weights w6, w7, and w8:

$$w_6^+ = 0.408666186$$
$$w_7^+ = 0.511301270$$
$$w_8^+ = 0.561370121$$

Deep Neural Networks: Back-Propagation
4-Example implementation: the backward pass

- We perform the actual updates in the neural network after we have the new weights leading into the hidden layer neurons.
- (ie, we use the original weights, not the updated weights, when we continue the backpropagation algorithm below).
- We continue the backward pass by calculating values for w1, w2, w3, w4?



Deep Neural Networks: Back-Propagation
4-Example: the backward pass

Hidden Layers

- Beginning with w_1, we have

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

- We're going to use a similar process as we did for the output layer, but slightly different to account for the fact that the output of each hidden layer neuron contributes to the output (and therefore error) of multiple output neurons.
- out_{h1} affects both out_{o1} and out_{o2} hence, the $\frac{\partial Etota_l}{\partial w_1}$ needs to consider its effect on both output neurons:

Deep Neural Networks: Back-Propagation

4-Example: the backward pass

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

Hidden Layers

- Hence, $\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$
- Starting with $\frac{\partial E_{o1}}{\partial out_{h1}}$, we have: $\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}}$
- We can calculate $\frac{\partial Eo_1}{\partial outh_1}$ using values we calculated earlier:

$$\frac{\partial E_{o1}}{\partial net_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = 0.74136507 * 0.186815602 = 0.138498562$$

- And $\frac{\partial net_{o1}}{\partial out_{h1}}$ = w5:

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$
$$\frac{\partial net_{o1}}{\partial out_{h1}} = w_5 = 0.40$$

- Together: $\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}} = 0.138498562 * 0.40 = 0.055399425$

Deep Neural Networks: Back-Propagation

4-Example: the backward pass

Hidden Layers

$$\frac{\partial E_{total}}{\partial w_1} = \boxed{\frac{\partial E_{total}}{\partial out_{h1}}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

- Following the same process for $\frac{\partial Eo_2}{\partial outh_1}$, we get:

$$\frac{\partial E_{o2}}{\partial out_{h1}} = -0.019049119$$

- Therefore,

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} = 0.055399425 + -0.019049119 = 0.036350306$$

- Now that we have $\frac{\partial Etota_l}{\partial outh_1}$, we need to figure out $\frac{\partial out_{h1}}{\partial net_{h1}}$ and then for each weight:

Deep Neural Networks: Back-Propagation

4-Example: the backward pass

Hidden Layers

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \boxed{\frac{\partial out_{h1}}{\partial net_{h1}}} * \frac{\partial net_{h1}}{\partial w_1}$$

- Now that we have $\frac{\partial Etota_l}{\partial outh_1}$, we need to figure out $\frac{\partial out_{h1}}{\partial net_{h1}}$ and then for each weight:

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}}$$

$$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.59326999(1 - 0.59326999) = \boxed{0.241300709}$$

Deep Neural Networks: Back-Propagation

4-Example: the backward pass

Hidden Layers

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

- Now, we calculate the partial derivative of the total net input to h_1 with respect to w_1 the same as we did for the output neuron:

$$net_{h1} = w_1 * i_1 + w_3 * i_2 + b_1 * 1$$

$$\frac{\partial net_{h1}}{\partial w_1} = i_1 = 0.05$$

- Putting it all together, we have:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$$

Deep Neural Networks: Back-Propagation
4-Example: the backward pass

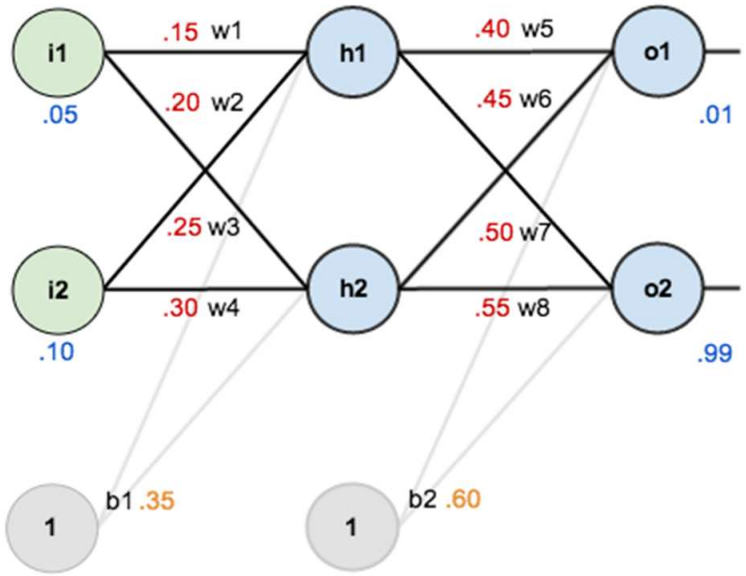
Hidden Layers

- We can now update w_1:

$$w_1^+ = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 * 0.000438568 = 0.149780716$$

- Repeating this for w2, w3, and w4

$$w_2^+ = 0.19956143$$
$$w_3^+ = 0.24975114$$
$$w_4^+ = 0.29950229$$



Deep Neural Networks: Back-Propagation
4-Example: the backward pass

- Finally, we've updated all of our weights.
- As you remember, when we fed forward the 0.05 and 0.1 inputs originally, the error on the network was 0.298371109.
- After this first round of backpropagation, the total error is now down to 0.291027924.
- The change may seem insignificant, but after repeating this process 10,000 times, for example, the error falls to 0.0000351085.

Note: You can go through it by yourself and check out for errors, if any.

Deep Neural Networks: Back-Propagation

backprop	Inputs	predicted	target	Error
Original weights	0.05	0.751365070	0.01	0.298371109
	0.1	0.772928465	0.99	
After 2nd backprop	0.05			0.291027924
	0.1			
		.		
After about 10^4 backprops	0.05	0.015912196	0.01	**0.0000351085**
	0.1	0.984065734	0.99	

Deep Neural Networks

Exercise:

- Activation Functions:
  - Implement DNNs with sigmoid, tanh and ReLU activation functions. Show that ReLU fixes the vanishing gradient problem experience by the previous two. Provide precise and too the point explanation.
  - Compare the performance difference in using the various ReLU families.
- Depth of ANNs:
  - Appreciate the role of depth by trying different depth implementations of artificial neural networks. Use plots to support your argument. Test accuracies or loss could be plotted against depth.
- The backpropagation algorithm:
  - Experiment on the backpropagation algorithm. Intermediate results of weight updates and loss could be printed/visualized to get good understanding of what is happening.