Carnegie Mellon University

18-819F: Introduction to Quantum Computing 47-779/785: Quantum Integer Programming & Quantum Machine Learning

Deep Learning

Lecture 06

2021.09.21.

Electrical & Computer ENGINEERING

TEPPER

USRA Universities Space Research Association

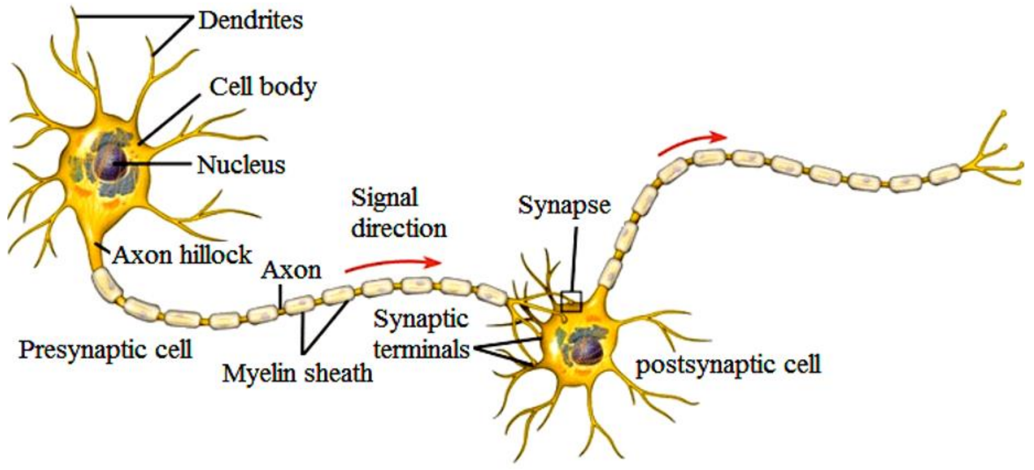
Agenda

- Relationship of artificial neurons to biological ones
  - Activation function as a model for nerve impulse action potential of biological neurons
  - Types of nonlinear activation functions
  - Simple toy model of a deep learning neural network

- Back propagation as a deep neural network training algorithm
  - Nested forward propagation
  - Chain rule of differentiation for connection strength (weight) modification
  - Modification of the connection strengths (weights) by back propagation

Biological Neuron

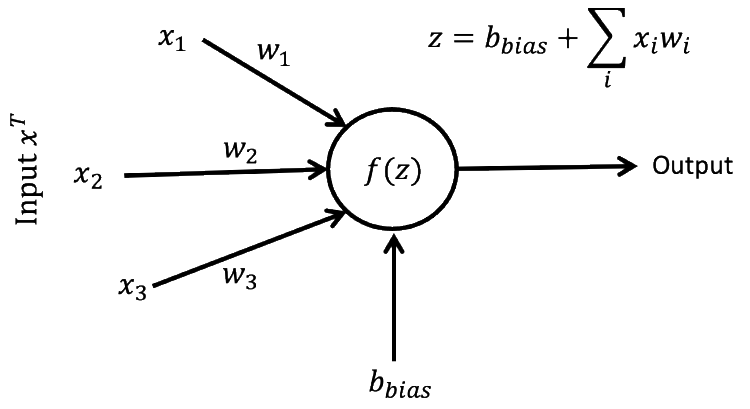
- The building block of the brain is the neuron, whose main components are illustrated in the graphic on the right.
- It is believed that the average human brain has about 86 billion neurons.
- Each neuron is estimated to be connected to about 7×10^3 synapses, connecting it to other neurons.
- The enormous computational abilities of the brain are due to its massive connectivity.



D. A. Torse, R. R. Maggavi, and S. A. Pujari, "Nonlinear blind source separation for EEG signal pre-processing in brain-computer interface system for epilepsy," International Journal of Computer Applications, Vol. **50**(14) 12 - 19 (2012).

Artificial Neuron

- Artificial intelligence systems that emulate the brain are formed from a limited set of interconnections between artificial neurons.
- The simplest model of the artificial neuron is illustrated on the right.
- Inputs, x_i, to the artificial neuron are equivalent to dendrites of the real neuron.
- Synapses are connections from (to) other neurons and are weighted by connection strengths, w_i.
- Cell body is represented by an activation function that operates on the sum of the inputs it received before transmitting to the next neuron through the axon.

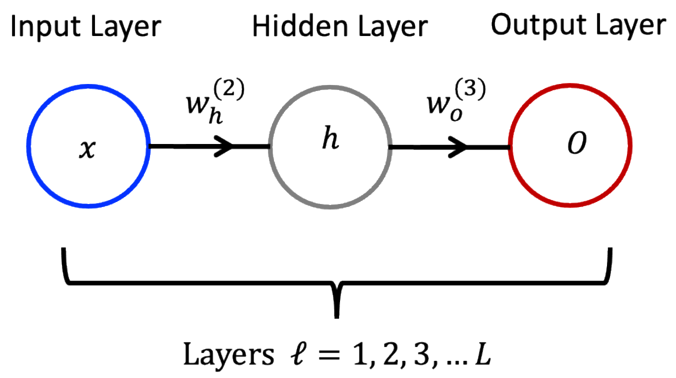


Artificial Neural Network Topologies

- The essence of deep learning is inventing topologies for interconnected artificial neurons.
- If the network or topology has many layers between what is considered the input and the output, then one has a "deep" network.
- Figuring out the optimal strengths or weights of the interconnections is what training and, consequently learning is all about.
- The most common topology of an artificial neural network is one where the signal propagates sequentially forward through the network; there are other types networks.

Simple Neural Network

- The simple network illustrated in the graphic to the right has three main layers:
  - An input layer,
  - A hidden processing layer, and
  - An output layer.
- This simple network structure is prototypical of all feed-forward deep neural networks.
- Input layer holds the data, where each neuron in the input layer represents a unique attribute of the dataset.
- The hidden layer, positioned between the input and the output, does most of the data processing.
- The hidden layer is fully connected – meaning that each neuron in the hidden layer gets input from all previous neurons and transmits its output to every neuron in the next layer.



Neural data processing

- Each neuron in the hidden layer applies an activation function to all the inputs it receives before retransmitting its output to the next set of neurons.

- Activation of an artificial neuron is modeled to mimic the firing of an action potential in real biological neurons.

- The final layer of neuron(s) in a network is the output layer; it receives input from a previous hidden layer and can apply an activation before it outputs the signal.

- In a feed-forward networks, the input data is propagated forward through the network layer-by-layer to the final prediction layer of the network.
- The mathematical effect of a feed-forward network can be represented by

$$Output = Prediction = f\left(f\left(w_h^{(2)}x\right)w_o^{(3)}\right) \quad \text{Eqn. (6.1)}$$

- In the Eqn. (6.1), $f(.)$ is an activation function; there are several activation functions we will discuss.

Essential Process Steps of a Neural Network

- The simple artificial neural network we introduced performs certain basic tasks that are common to all networks like it; the tasks are enumerated as follows:

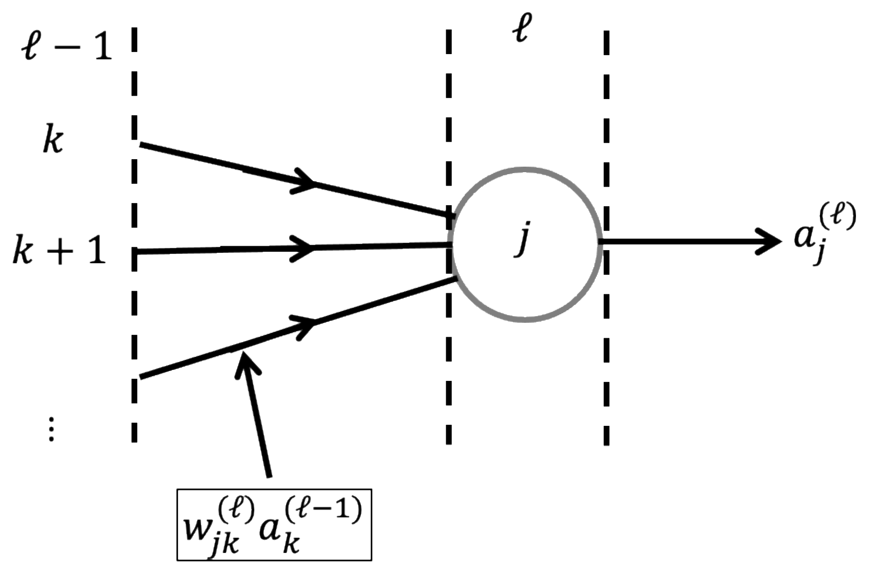
  (1) Calculate weighted input to hidden layer (this involves multiplying the input, x, by the weight, $w_h^{(2)}$);

  (2) Apply an activation function to (or operate on) the input and pass the result to next layer;

  (3) Compute the next layer's input which is now the output of the hidden layer; this computation is accomplished by multiplying the output of the hidden layer by the weight $w_o^{(3)}$.

  (4) Finally, the result is passed to the output layer, which may apply an activation function of its own before outputting the result as the prediction of the simple network.

Nomenclature for a Neuron in a Network

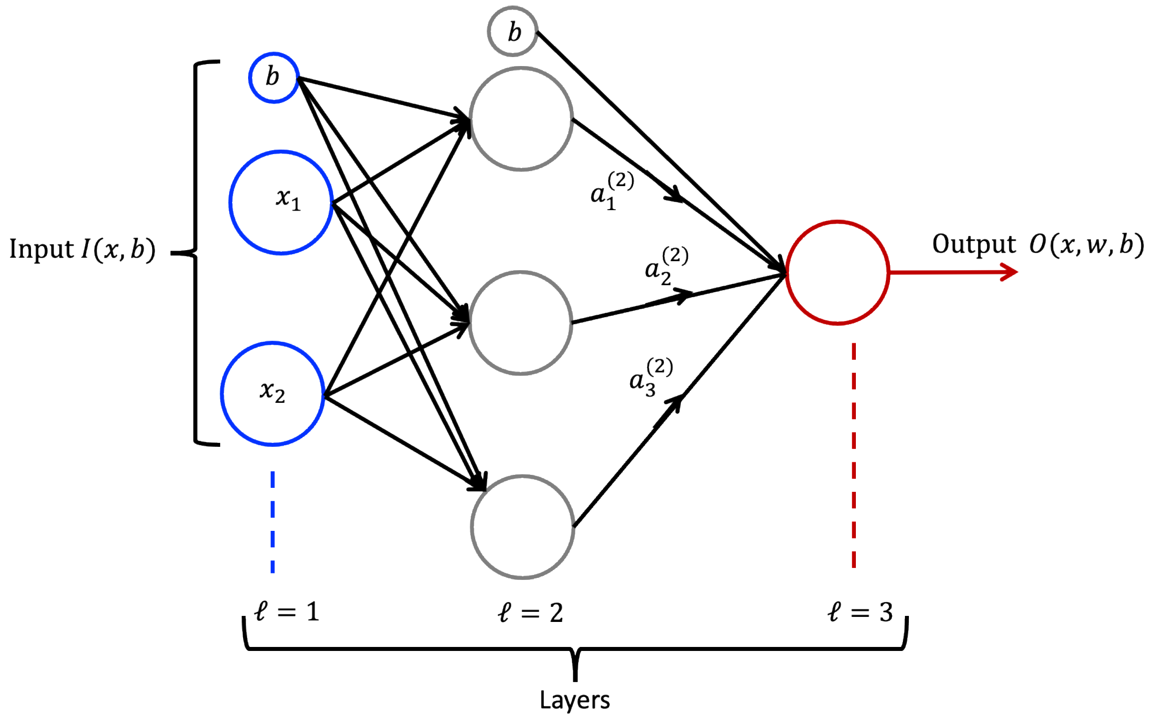


Total input to node j in layer l is:

$$z_j^{(\ell)} = \sum_{k=1}^{K} w_{jk}^{(\ell)} a_k^{(\ell-1)}$$

- Layers in a network are numbered sequentially from the input layer as $\ell = 1$ to the output layer as $\ell = \mathcal{L}$.
- Neurons (nodes) in layer ℓ of a network are indexed as $j = 1, 2, \ldots N$ for N neurons.
- Nodes (or neurons) in layer $\ell - 1$ are indexed as $k = 1, 2, \ldots K$ for K neurons.
- A weight connecting node j in layer l to node k node in layer $\ell - 1$ is denoted as $w_{jk}^{(\ell)}$.
- The activation output of neuron j in layer ℓ is denoted as $a_j^{(\ell)}$.
- Input to a node j in layer ℓ is the activated output from layer $\ell - 1$.

Additional Neurons in a Layer Enhance Computation



- To perform processing tasks on the input so that a network gets results that matter, one generally adds more neurons into each hidden layer of the network;
- The essential features of the power of more neurons in a layer can be appreciated from the network presented in the graphic to the left.
- As more neurons are added, reference to each becomes problematic unless we adhere to a notation for labeling or a strict nomenclature.
- The nomenclature was briefly discussed in general terms in the previous slide; the labeled edges and nodes will be explained in the next slide.

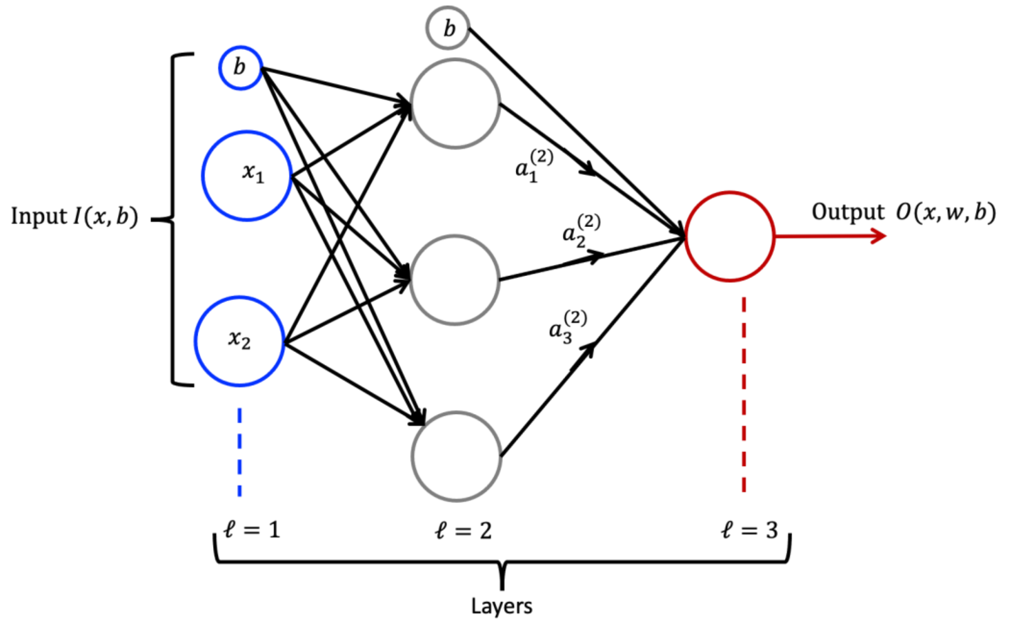
Computation Performed by a Typical Neural Network

- The mathematical computation performed by the network illustrated on the right is:

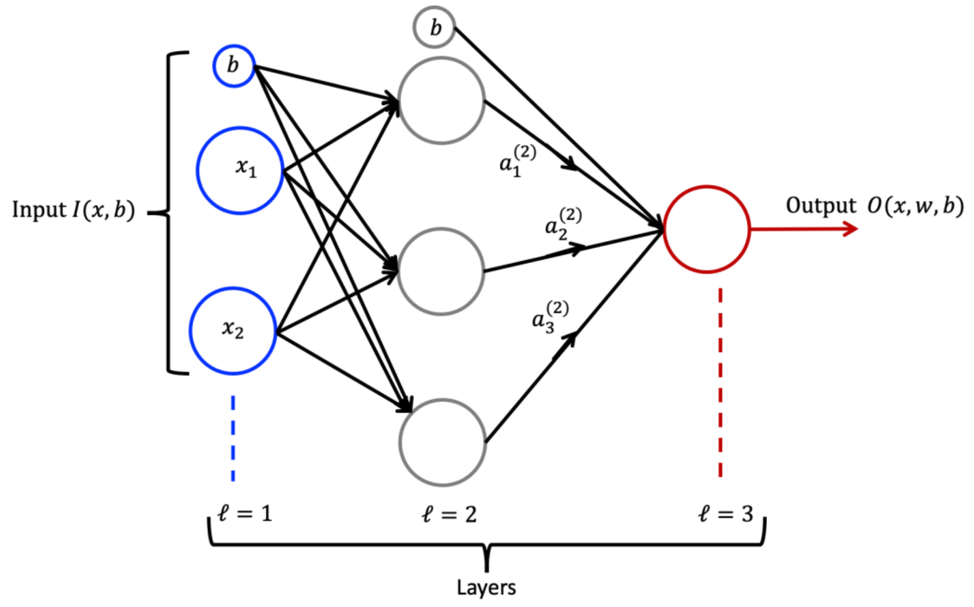
$$
\begin{aligned}
a_1^{(2)} &= f\left(w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + b_1^{(1)}\right)\\
a_2^{(2)} &= f\left(w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + b_2^{(1)}\right)\\
a_3^{(2)} &= f\left(w_{31}^{(1)}x_1 + w_{32}^{(1)}x_2 + b_3^{(1)}\right)\\
O(w,b,x) &= a_1^{(3)} = f\left(w_{11}^{(2)}a_1^{(2)} + w_{12}^{(2)}a_2^{(2)} + w_{13}^{(2)}a_3^{(2)} + b_2^{(2)}\right)
\end{aligned}
$$

- We can also compactly write

$$z_i^{(2)} = \sum_{j=1}^{N} w_{ij}^{(1)}x_j + b_i^{(1)} \text{ and } a_i^{(\ell)} = f\left(z_i^{(\ell)}\right)$$



Streamlining Computational Network Process



- A compact summary of the computational equations of our simple network, which again is shown on the left, are

$$z_i^{(2)} = \sum_{j=1}^{N} w_{ij}^{(1)} x_j + b_i^{(1)} \text{ and } a_i^{(\ell)} = f\left(z_i^{(\ell)}\right);$$

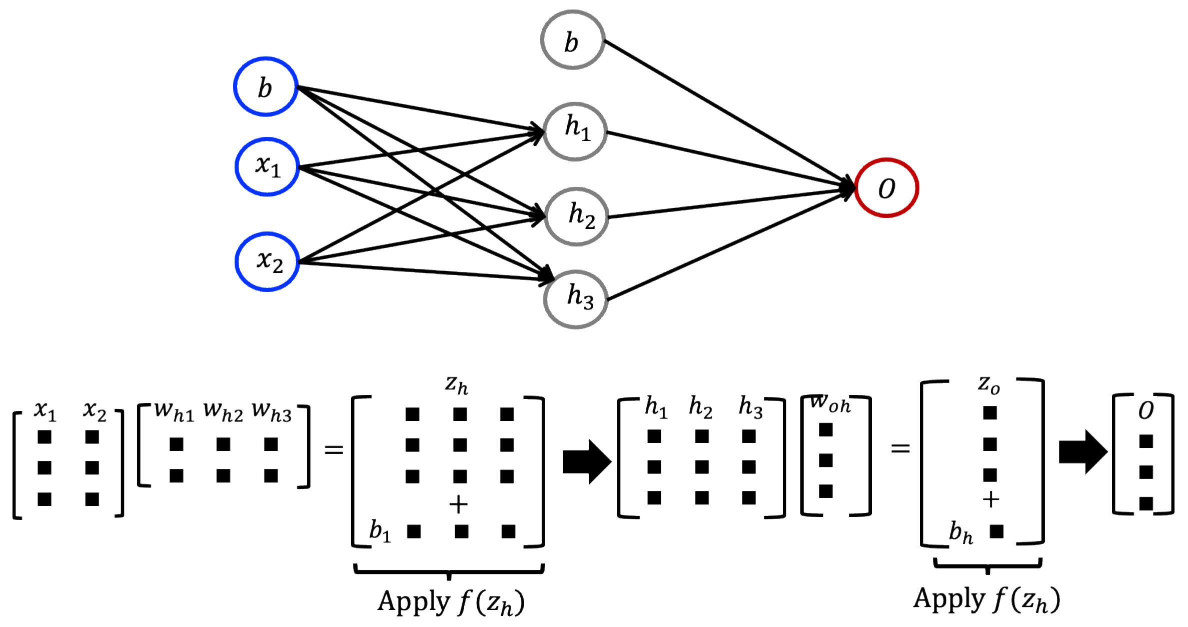
- The activation function can be extended to act on vectors one element at a time $f(z_1, z_2, z_3) = f(z_1), f(z_2), f(z_3)$, we can therefore say

$$\begin{aligned} z^{(2)} &= w^{(1)}x + b^{(1)} \\ a^{(2)} &= f\left(z^{(2)}\right) \\ z^{(3)} &= w^{(2)}a^{(2)} + b^{(2)} \\ O(w, b, x) &= a^{(3)} = f(z^{(3)}) \end{aligned}$$

- Generally, we will have

$$\begin{aligned} z^{(\ell+1)} &= w^{(\ell)}a^{(\ell)} + b^{(\ell)} \\ a^{(\ell+1)} &= f\left(z^{(\ell+1)}\right) \end{aligned}$$

Linear Algebra Simplification of Network Processes



- Another perspective of the prototype network we have been discussing is to consider its linear algebraic representation. This is illustrated below, where the same network is redrawn, and matrices that store input, weights and results of computations are shown below the network.

- The output of the hidden layer is acted on by the activation function before it is multiplied by the weights connecting the hidden layer to the output.

- Similarly, the output of the last neuron is acted on by the activation function before it considered as the final network output.

Nerve Impulse Activation in Biological Neuron

- Biological neurons are known to fire off an electrical nerve impulse or action potential when communicating with other neurons.

- A biological neuron gathers all the inputs to its dendrites before firing the nerve impulse; this happens after a threshold of the sum of all inputs is reached.

- One could imagine that for the simple neuron we have been discussing, when the impulse is fired, the output is

$$y = b + \sum_{i=1}^{N} w_i x_i;$$

- However, this is not what happens. The output is a nonlinear function of the total input.

- The nonlinear output means the neurons does perform some processing; to account for this in the artificial neuron, we use an activation function that nonlinearly operates on the sum of the inputs.

Nonlinear Activation Functions

- In the spirit of continuing to emulate biology, several nonlinear functions are routinely used to squash the output of an artificial neuron.
- The type of function chosen depends on the application context and on the efficiency of the computation of the output.
- An activation function applied to the output of a neuron is expected to give a real-valued output of the total input. The out is typically squashed to the interval [0,1].
- In a sense the nonlinear activation function normalizes the outputs of a neuron to 1 so that the the result of the operation of the activation function is between 0 and 1.

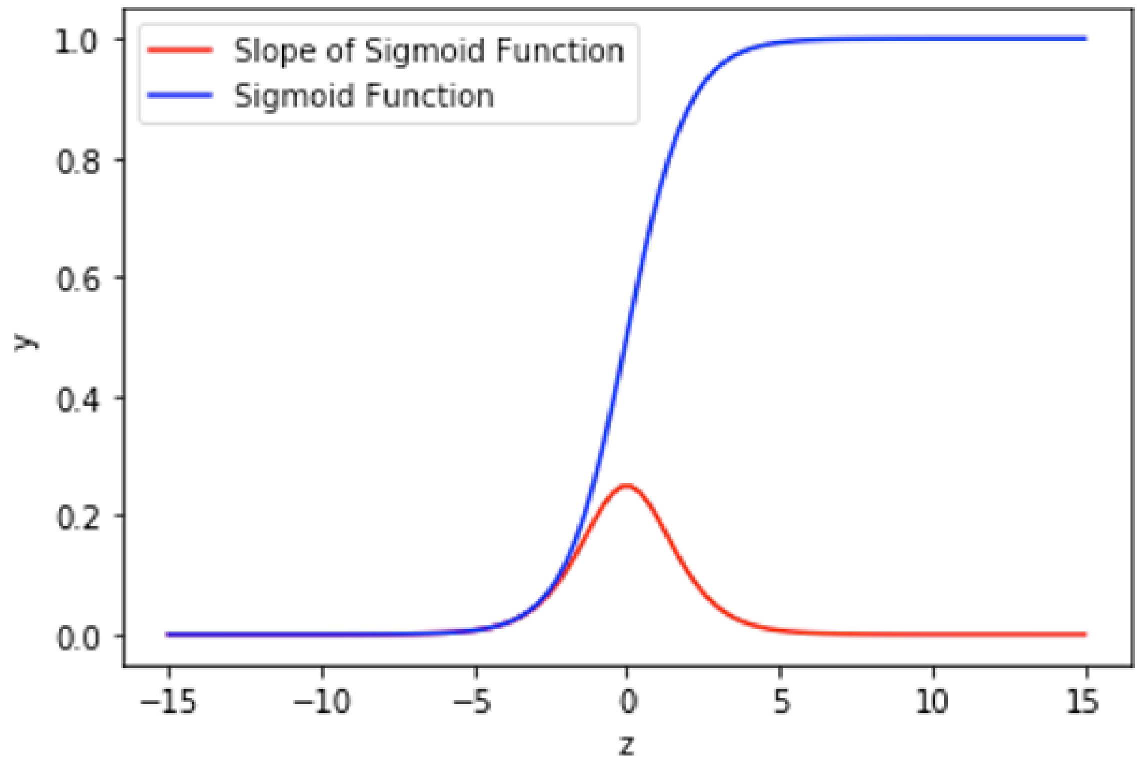
Sigmoid Activation Function

- One of the earliest activation functions is the sigmoid function; this function takes the neuronal output as input; for example,

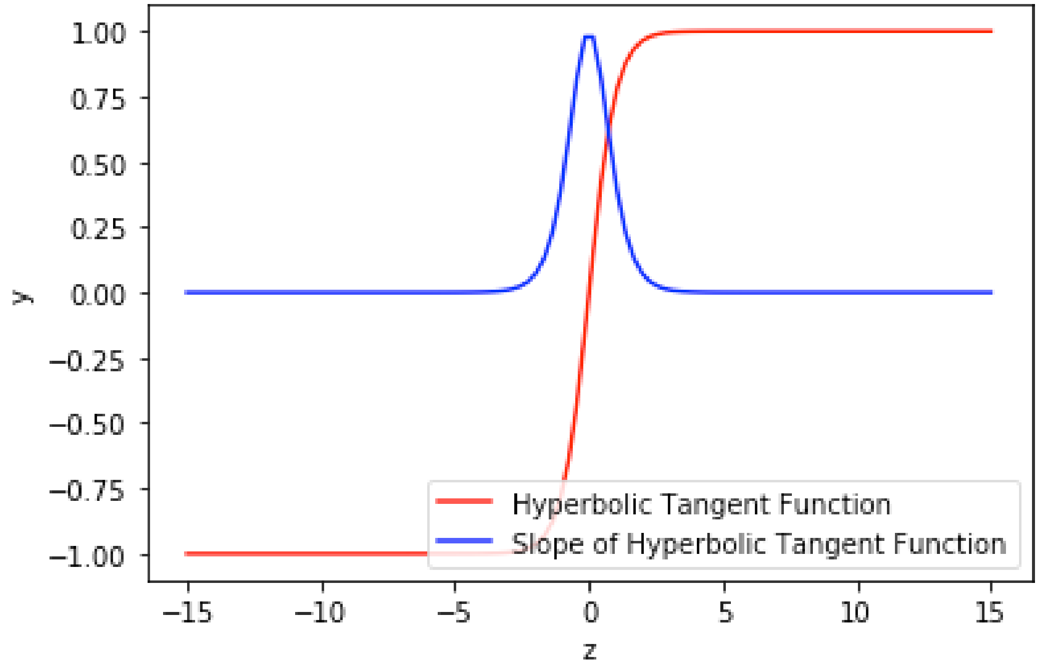
$z = b + \sum_i w_i x_i$ becomes the input to the sigmoid function,

$y(z) = f(z) = \frac{1}{1+e^{-z}}$ Eqn. (6.2).

- A graphic of the sigmoid and its derivative are shown on the right.



Hyperbolic Tangent Activation Function



- Another activation function that was popular in the early days of deep learning was the hyperbolic tangent.

- The hyperbolic tangent limits its output between $[-1, 1]$; this function is mathematically described by

$$y(z) = f(z) = \tanh z = \left(\frac{2}{1+e^{-2z}}\right) - 1;$$

Eqn. (6.3).

- A graph of the hyperbolic function and its derivative are illustrated on the left.

Linear Rectified Unit function

- The most popular activation function is the Linear Rectified Unit (ReLu); for input

$z = b + \sum_i w_i x_i$,

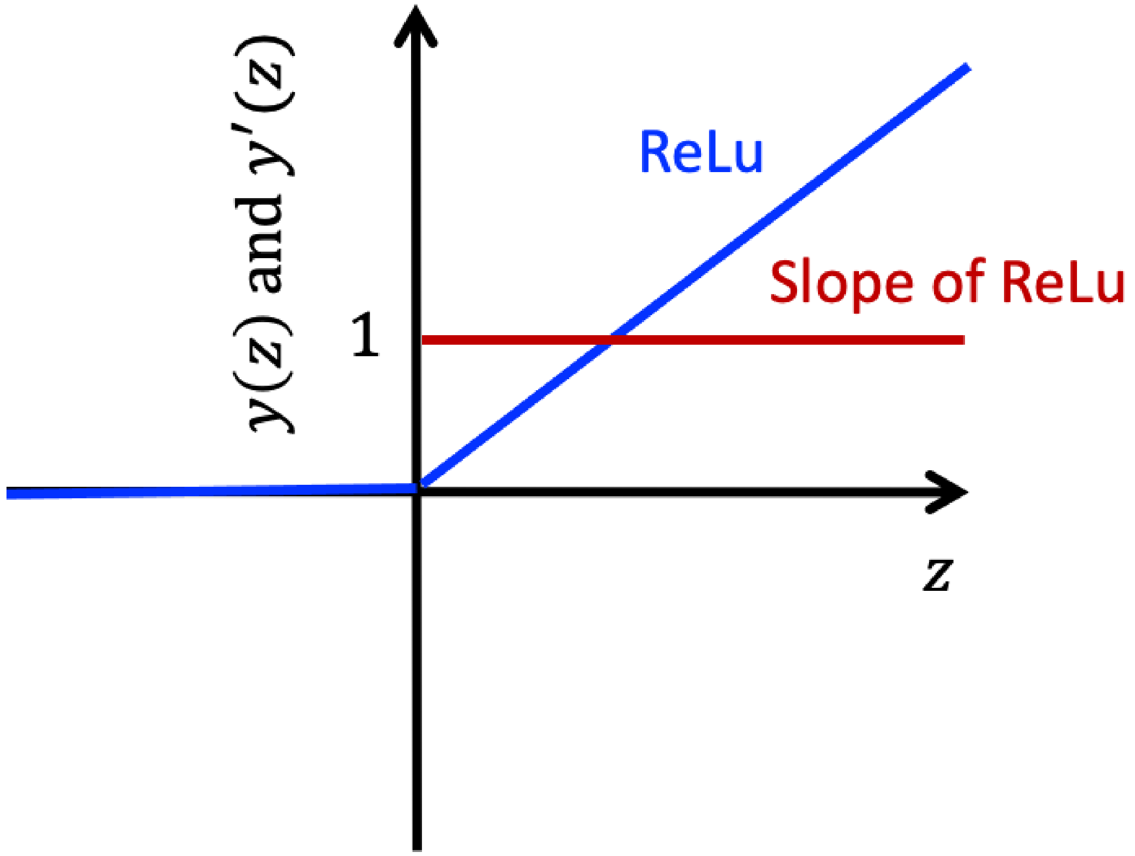
- ReLu is defined as

$$y(z) = f(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases} \quad \text{Eqn. (6.4).}$$

- The derivative of ReLu is the step function, mathematically described by

$$f'(z) = \begin{cases} 1 & z \geq 0 \\ 0 & z < 0 \end{cases} \quad \text{Eqn. (6.5)}$$

- A graphic illustration of ReLu is shown on the right.



Parametric Rectified Linear Unit

- The parametric ReLu function is a variation of the conventional ReLu function except that the it is not zero for $z < 0$; it is instead a linear function with a slope that is adjustable as part of the learning process to arrive at the best "learned function." Again, for our typical neuronal input of

$$z = b + \sum_i w_i x_i,$$

- Parametric ReLu is defined as

$$y(z) = f(z) = \begin{cases} z & \geq 0 \\ \alpha z & z < 0 \end{cases} \qquad \text{Eqn. (6.6).}$$

- The derivative of the parametric ReLu is

$$f'(z) = \begin{cases} 1 & z \geq 0 \\ \alpha & z < 0 \end{cases} \qquad \text{Eqn. (6.7).}$$

Exponential ReLu

- This variant of the ReLu has an exponential component when its input values are negative; as before, for

$$z = b + \sum_i w_i x_i,$$

- The exponential ReLu is defined as

$$y(z) = f(z) = \begin{cases} z & z \geq 0 \\ \alpha(e^z - 1) & z < 0 \end{cases} \quad \text{Eqn. (6.8).}$$

- The derivative of the exponential ReLu is

$$f'(z) = \begin{cases} 1 & z \geq 0 \\ f(z) + \alpha & z < 0 \end{cases} \quad \text{Eqn. (6.9).}$$

Softmax Activation Function

- Softmax is connected to logistic regression. This function takes its inputs and normalizes the values to follow a probability distribution whose total sum must be 1.

- As usual for input given as

$$z = b + \sum_i w_i x_i \quad \text{Eqn. (6.10)}$$

- One writes the softmax function as

$$y(z) = f(z) = P\left(y = j \middle| z^{(i)}\right) = \frac{e^{z^{(i)}}}{\sum_{j=0}^{K} e^{z_k^{(i)}}} \quad \text{Eqn. (6.11).}$$

- If we rewrite (6.10) as $z = w_0 x_0 + w_1 x_1 + \cdots + w_k x_k = \sum_{i=0}^{k} w_i x_i = w^T x$ Eqn. (6.12).
- Note that we have used $w_0 = b$ and $x_0 = 1$ in (6.12) above.

Action of Softmax

- As can seen in Eqn. (6.12), we have used linear algebra to succinctly write the input to softmax. This function is useful in multiclass classification. It returns a probability for a data point for each individual class.

- When applied to a neural network, the output layer would have as many neurons as the number of classes in the target; for two classes, there would be two output neurons, and for three classes, there would be three output neurons.

- As an example, consider a network for classifying some things into three classes, meaning the are three output neurons; if the output neurons yield the output vector $[1.2, 0.8, 0.4]$, then when one applies softmax to this output vector, the result is:

$$P(y=1|1.2) = \frac{e^{1.2}}{e^{1.2}+e^{0.8}+e^{0.4}} = 0.47;\;\; P(y=2|0.8) = \frac{e^{0.8}}{e^{1.2}+e^{0.8}+e^{0.4}} = 0.32;\;\; P(y=3|0.4) = \frac{e^{0.4}}{e^{1.2}+e^{0.8}+e^{0.4}} = 0.21;$$

- Notice that the output has been mapped to the vector $[0.47, 0.32, 021]$, whose components add to 1.

Swish Activation Function

- As usual, we provide the input to the activation functions as

  $z = b + \sum_i w_i x_i \equiv \sum_{i=0}^{k} w_i x_i$ as long as we define $w_0 = b$ and $x_0 = 1$.
- Then the Swish function would be defined as

  $$f(z) = \frac{z}{1+e^{-z}} \quad \text{Eqn. (6.13);}$$
- This function is smooth and has outputs from $z = -\infty$ to $z = +\infty$.

- It is computationally as efficient as the ReLu but has better performance in very deep neural network models.

- The Swish function is a Google property (they invented and popularized its use).

Back Propagation

- Backpropagation is used to adjust the weights in a neural network in proportion to how much each weight contributes to the overall error (cost function).

- The process iteratively reduces each weight's error until eventually the weights yield more accurate predictions.

- While forward propagation is a long series of nested equations, back propagation can be thought of as application of the chain rule of differentiation to the nested equations.

- For example, for the nested function

$$f(x) = a\left(b\left(c(x)\right)\right) \text{ Eqn. (6.14).}$$

- The derivative is $f'(x) = \frac{\partial f}{\partial a} . \frac{\partial a}{\partial b} . \frac{\partial b}{\partial c} . \frac{\partial c}{\partial x}$.

Cost Function Derivatives

- To find the derivative of a cost function with respect to any weight in the network, one needs for a single neuron (i) the weighted input $z = w_i x_i$, (ii) the derivatives $\partial z/\partial x_i = w_i$ and $\partial z/\partial w_i = x_i$, and (iii) the activation function;

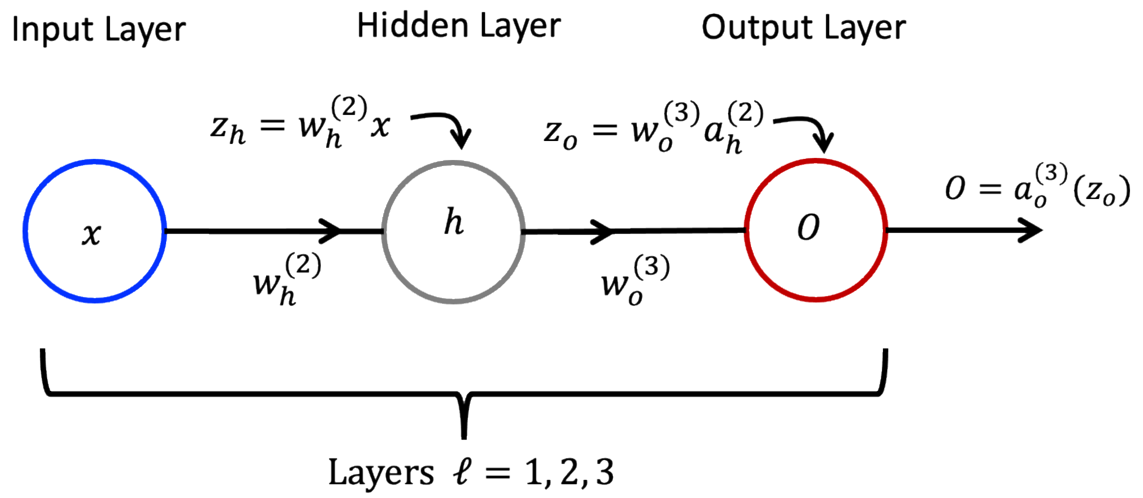
- This example we will use ReLu, R, which we defined earlier as
$$R = \max(0, z)$$
$$R' = \begin{cases} 1 & z \geq 0 \\ 0 & z < 0 \end{cases}$$

- Let the cost function be: $C = \frac{1}{2}(\tilde{y} - y)^2$ Eqn. (6.15), where $\tilde{y}$ is the activated output and y the target. We also get that $\partial C/\partial \tilde{y} = (\tilde{y} - y)$.

- Cost is a nested function given as $C\left(R\left(z(w_i x_i)\right)\right)$.

- By the chain rule, $C'(w_i) = \frac{\partial C}{\partial R}.\frac{\partial R}{\partial z}.\frac{\partial z}{\partial w_i} = (\tilde{y} - y).R'(z).x_i$

Back Propagation in a Simple Network



- The essential idea behind backpropagation can be understood from the simple network we have been using as a stand-in for multi-layer deep learning networks. Relevant details are now included in the simple network in the graphic on the left.

- Note that the outputs after the second- and third-layer neurons are the activated inputs from the previous layer. This means

$h = a_h^{(2)}(z_h)$ and $O = a_o^{(3)}(z_o)$ and of course $z_h(x) = w_h^{(2)} x$ and $z_o\left(a_h^{(2)}\right) = w_o^{(3)} a_h^{(2)}$.

Cost Function Dependence on Connection Weights

- For the simple network to learn, we must modify the connection weights to minimize the cost function after each forward propagation pass through the network until the cost is at its minimum. With α as the learning rate, weights in layer ℓ are modified iteratively according to

$$w^{(\ell)} = w^{(\ell)} - \alpha\left(\frac{\partial C}{\partial w^{(\ell)}}\right) \text{ Eqn. (6.16).}$$

- For our simple network, the cost function can be written as

$$C = \frac{1}{2}\left(a_o^{(3)} - y\right)^2 \text{ Eqn. (6.17).}$$

- We have introduced a factor of $1/2$ for the convenience of canceling out the 2 from differentiation of of the cost function.

- The derivatives of the cost function with the respect to $w_0^{(3)}$ and $w_h^{(2)}$ are

$$\frac{\partial C}{\partial w_o^{(3)}} = \frac{\partial C}{\partial a_o^{(3)}} . \frac{\partial a_o^{(3)}}{\partial z_o} . \frac{\partial z_o}{\partial w_o^{(3)}} = \left(a_o^{(3)} - y\right) . \frac{\partial a_o^{(3)}}{\partial z_o} . a_h^{(2)} \quad \text{Eqn. (6.18).}$$

$$\frac{\partial C}{\partial w_h^{(2)}} = \frac{\partial C}{\partial a_o^{(3)}} . \frac{\partial a_o^{(3)}}{\partial z_o} . \frac{\partial z_o}{\partial a_h^{(2)}} . \frac{\partial a_h^{(2)}}{\partial z_h} . \frac{\partial z_h}{\partial w_h^2} = \left(a_o^{(3)} - y\right) . \frac{\partial a_o^{(3)}}{\partial z_o} . w_o^{(3)} . \frac{\partial a_h^{(2)}}{\partial z_h} . x \quad \text{Eqn. (6.19).}$$

Memoization

- In our simple network, the derivatives of the cost function with respect to $w_o^{(3)}$ and $w_h^{(2)}$ in Eqn. (6.18) and (6.19) contain common shared terms (high-lighted in blue). If we had a network with more layers than the three layers, we would find this pattern repeated for other derivatives with respect to the weights. This offers an opportunity to NOT keep recalculating the common shared terms each weight deeper into the network (from the output).

- The savings in time and effort for not repeating the calculations goes by the term memoization; this is a valid computer science term which means do not recompute the same result over and over.

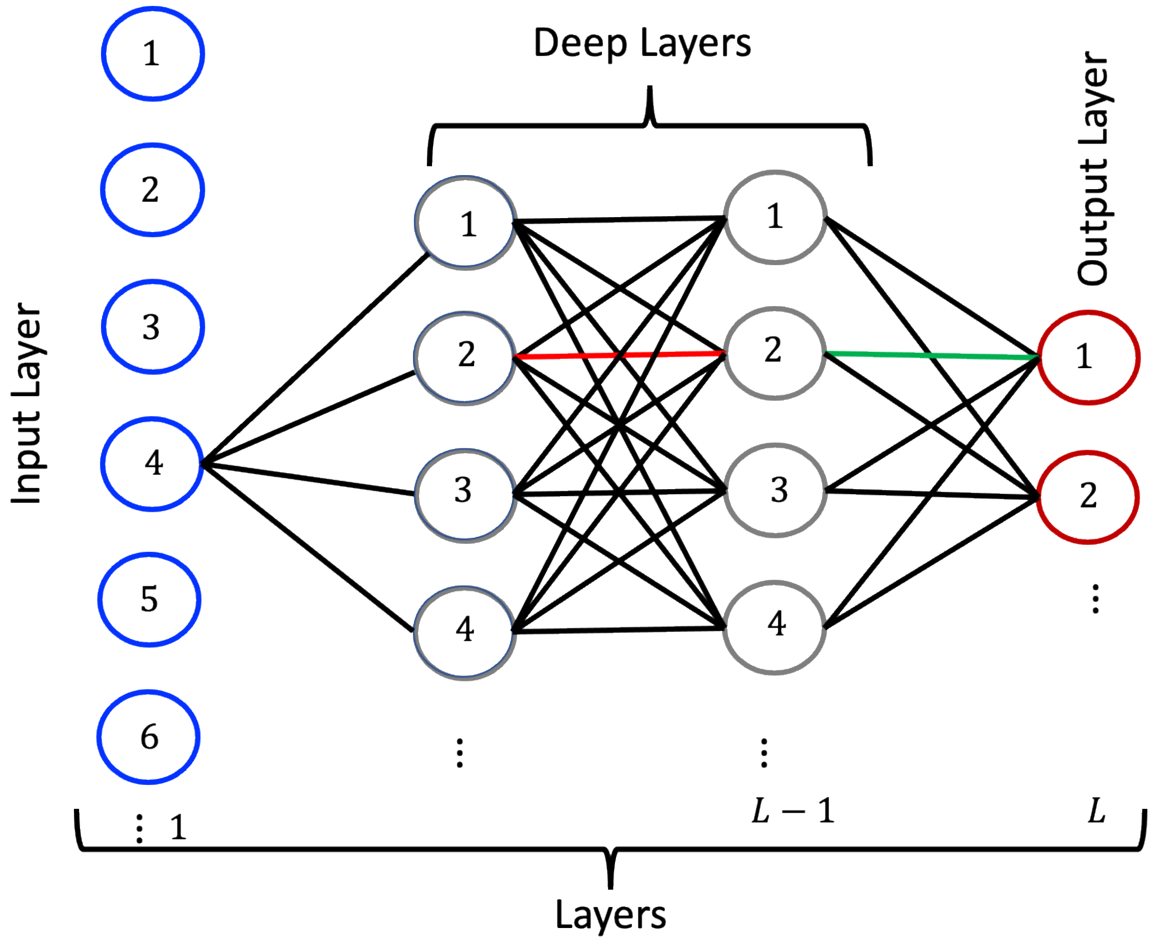
Back Propagation in a General Network

- The general network depicted in the graphic on the right is representative of typical practical networks that can be used for some machine learning task.

- The cost function is the sum of all the losses summed over all the output nodes j; thus,

$$C = \frac{1}{2}\sum_{j=1}^{n}\left(a_j^{(L)} - y_i\right)^2 \quad \text{Eqn. (6.20).}$$

- Suppose we want to adjust the strength (weight) of connection between node 2 in layer $L-1$ and node 1 in layer L (green connection); our first task is to compute the derivative of the cost function with respect to the weight of interest which in this case is $\partial C/\partial w_{12}^{L}$.



Nested Derivative Calculations

- We know that the cost function in (6.20) is a function of $a_1^{(L)}$, which in turn is a function of $z_1^{(L)}$, which itself is a function of $w_{12}^{(L)}$. As we have seen before, this means the cost function can be written as

$$C = C\left(a_1^{(L)}\left(z_1^{(L)}\left(w_{12}^{(l)}\right)\right)\right) \text{ Eqn. (6.22).}$$

- By the chain rule of differentiation, we then have

$$\frac{\partial C}{\partial w_{12}^{(L)}} = \left(\frac{\partial C}{\partial a_1^{(L)}}\right)\left(\frac{\partial a_1^{(L)}}{\partial z_1^{(L)}}\right)\left(\frac{\partial z_1^{(L)}}{\partial w_{12}^{(L)}}\right) \text{ Eqn. (6.23).}$$

- Each term in (6.23) can be calculated separately, thus

$$\frac{\partial C}{\partial a_1^{(L)}} = \frac{\partial}{\partial a_1^{(L)}}\left[\frac{1}{2}\sum_{j=1}^{n}\left(a_j^{(L)} - y_i\right)^2\right] = \frac{1}{2}\frac{\partial}{\partial a_1^{(L)}}\left[\left(a_1^{(L)} - y_1\right)^2 + \left(a_2^{(L)} - y_2\right)^2 + \cdots\right] = \left(a_1^{(L)} - y_1\right) \text{ Eqn. (6.24).}$$

Nested Derivative Calculations …

- The second term of (6. 23), for each node j in layer L we have

$$a_j^{(L)} = \xi^L(z_j^{(L)})$$

- For the node of interest $j = 1$ in layer L, we have $a_1^{(L)} = \xi^{(L)}\ (z_1^{(L)})$ so that

$$\frac{\partial a_1^{(L)}}{\partial z_1^{(L)}} = \frac{\partial}{\partial z_1^{(L)}}\left[\xi^{(L)}(z_1^{(L)})\right] = \frac{\partial \xi^{(L)}(z_1^{(L)})}{\partial z_1^{(L)}} \quad \text{Eqn. (6.25).}$$

- For the third term in (6.23), we notice that for each node j in layer L, we have

$$z_j^{(l)} = \sum_{k=1}^{n} w_{jk}^{(L)} a_k^{(L-1)}.$$

- We are interested in the case for $j = 1$, thus

$$\frac{\partial z_1^{(L)}}{\partial w_{12}^{(L)}} = \frac{\partial}{\partial w_{12}^{(L)}}\left[\sum_{k=1}^{n} w_{12}^{(L)} a_k^{(L)}\right] = \frac{\partial}{\partial w_{12}^{(L)}}\left[w_{11}^{(L)} a_1^{(L-1)} + w_{12}^{(L)} a_2^{(L-1)} + w_{13}^{(L)} a_3^{(L-1)} \ldots\right] = a_2^{(L-1)} \quad \text{Eqn. (6.26).}$$

Nested Derivative Calculations …

- Finally, we collect all the partial derivatives from (6.24) to (6.26) to write

$$\frac{\partial C}{\partial w_{12}^{(L)}} = \left(a_1^{(L)} - y_1\right)\left(\frac{\partial \xi^{(L)}(z_1^{(L)})}{\partial z_1^{(L)}}\right)\left(a_2^{(L-1)}\right) \quad \text{Eqn. (6.27).}$$

- For all training samples, N, we are ultimately interested in the average derivative, which is

$$\frac{\partial C}{\partial w_{12}^{(L)}} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial C_i}{\partial w_{12}^{(L)}} \quad \text{Eqn. (6.28).}$$

- Now that we have the derivative of the cost function with respect to the the weight $w_{12}^{(L)}$, if we wanted to adjust the value of this weight, we must follow the rule from (6.16) which requires us to make the following adjustment

$$w_{12}^{(L)} = w_{12}^{(L)} - \alpha\frac{\partial C}{\partial w_{12}^{(L)}} \quad \text{Eqn. (6.29).}$$

Effect of Weights Deeper in the Network

- The effect of the interconnection weight $w_{22}^{(L-1)}$ (high-lighted in red in our general network graphic on slide 29), which is a bit deeper in the network from the output side of the network can be calculated by first determining how the cost function derivative changes with this weight, thus

$$\frac{\partial C}{w_{22}^{(L-1)}} = \left(\frac{\partial C}{\partial a_2^{(L-1)}}\right)\left(\frac{\partial a_2^{(L-1)}}{\partial z_2^{(L-1)}}\right)\left(\frac{\partial z_2^{(L-1)}}{\partial w_{22}^{(L-1)}}\right) \quad \text{Eqn. (6.30).}$$

- This looks like what we have already done, but it is not! This is because for a node j in layer L, the loss depends on $a_j^{(L)}$, which depends on $z_j^{(L)}$. But $z_j^{(L)}$ depends on all weights connected to node j from the $L-1$ layer as well as the all the activation outputs of layer $L-1$. We see then that $z_j^{(L)}$ depends on $a_2^{(l-1)}$. Thus

$$\frac{\partial C}{\partial a_2^{(L-1)}} = \sum_{j=1}^{N}\left[\left(\frac{\partial C}{\partial a_j^{(L)}}\right)\left(\frac{\partial a_2^{(L)}}{\partial z_j^{(L)}}\right)\left(\frac{\partial z_j^{(L)}}{\partial a_2^{(L-1)}}\right)\right] \quad \text{Eqn. (6.31)}$$

Effect of Weights Deeper in the Network

- To compute (6.31), we need to find

$\frac{\partial z_j^{(L)}}{\partial a_2^{(L-1)}}$, but for each node j in layer L we have

$z_j^{(L)} = \sum_{k=1}^{N} w_{jk}^{(L)} a_k^{(L-1)}$ Eqn. (6.32), hence

$$\frac{\partial z_j^{(L)}}{\partial a_2^{(L-1)}} = \frac{\partial}{\partial a_2^{(L-1)}}\left[\sum_{k=1}^{N} w_{jk}^{(L)} a_k^{(L-1)}\right] = \frac{\partial}{\partial a_2^{(L-1)}}\left[w_{j1}^{(L)} a_1^{(L-1)} + w_{j2}^{(L)} a_2^{(L-1)} + \cdots\right] = w_{j2}^{(L)} \text{ Eqn. (6.33).}$$

- Now we can write

$$\frac{\partial C}{\partial a_2^{(L-1)}} = \sum_{j=1}^{N}\left[\left(\frac{\partial C}{\partial a_j^{(L)}}\right)\left(\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}}\right)\left(\frac{\partial z_j^{(L)}}{\partial a_2^{(L-1)}}\right)\right] = \sum_{j=1}^{N}\left[\left(a_j^{(L)} - y_j\right)\left(\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}}\right) w_{j2}^{(L)}\right] \text{ Eqn. (6.34).}$$

Effect of Weights Deeper in the Network…

- The average derivative for all training samples can now be written as

$$\frac{\partial C}{\partial a_2^{(l-1)}} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial C_i}{\partial a_2^{(L-1)}} \quad \text{Eqn. (6.35).}$$

- All the terms in Eqn. (6.31) have now been determined and one can finally proceed with the computation of cost derivative with the weight $w_{22}^{(L-1)}$. Thus

$$\frac{\partial C}{\partial w_{22}^{(L-1)}} = \left(\frac{\partial C}{\partial a_2^{(L-1)}}\right)\left(\frac{\partial a_2^{(L-1)}}{\partial z_2^{(L-1)}}\right)\left(\frac{\partial z_2^{(L-1)}}{\partial w_{22}^{(L-1)}}\right) \quad \text{Eqn. (6.36).}$$

- Although we have only shown the computation of derivatives for two weights, the principle is clear; one begins at the output and progresses deeper into the network until the first weight that connects the inputs to the first hidden layer is reached.
- It should also now be obvious why training a deep neural network is time-consuming.

Regularization and overfitting

- To prevent deep learning networks from overfitting the training dataset, it is common to constrain the range of weights the network can learn. This is done by writing the cost function (in our case, the $L2$) as:

$$C(w,b) = \left[\frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}\left(a_i^{(L)}(x_i) - y_i\right)^2\right] + \frac{\lambda}{2}\left(w_{jk}^{(\ell)}\right)^2$$

- The last term in the equation above penalizes the cost function, with λ called the weight decay parameter, which controls the relative importance of the two terms in the cost function expression above.

- As reference for this topic and most of deep learning, please see (free online book):

“Deep Learning,” I. Goodfellow, Y. Bengio, and A. Courville (MIT Press 2016).
https://www.deeplearningbook.org/

On Cost (Objective) Functions

- All our examples and discussions so far have focused on using the ”means squared error” or the MSE expression as the cost function. This function is also called the $L2$ cost function.

- You should be aware there are other cost functions; a few are listed below:
  1. Cross-entropy (Bernoulli negative log likelihood)
  2. Hellinger distance
  3. Kullback-Liebler Divergence
  4. Itakura-Saito Distance
- The choice on which one is best depends on the particular application at hand. The L2 cost function just happens to be the simples and most convenient.

Summary

- Introduced the artificial neurons and its relationship to the biological one
  - Discussed construction of multi-layered artificial neural networks
  - Nonlinear activation functions

- Back propagation as a training model for artificial neural networks
  - Chain rule of differentiation and nested functions
  - Connection strength (weight) modification during network training