Lecture 2: Feedforward Neural Networks

Ali Harakeh

University of Waterloo

WAVE Lab
ali.harakeh@uwaterloo.ca

May 9, 2017

Overview

Section 1

Introduction

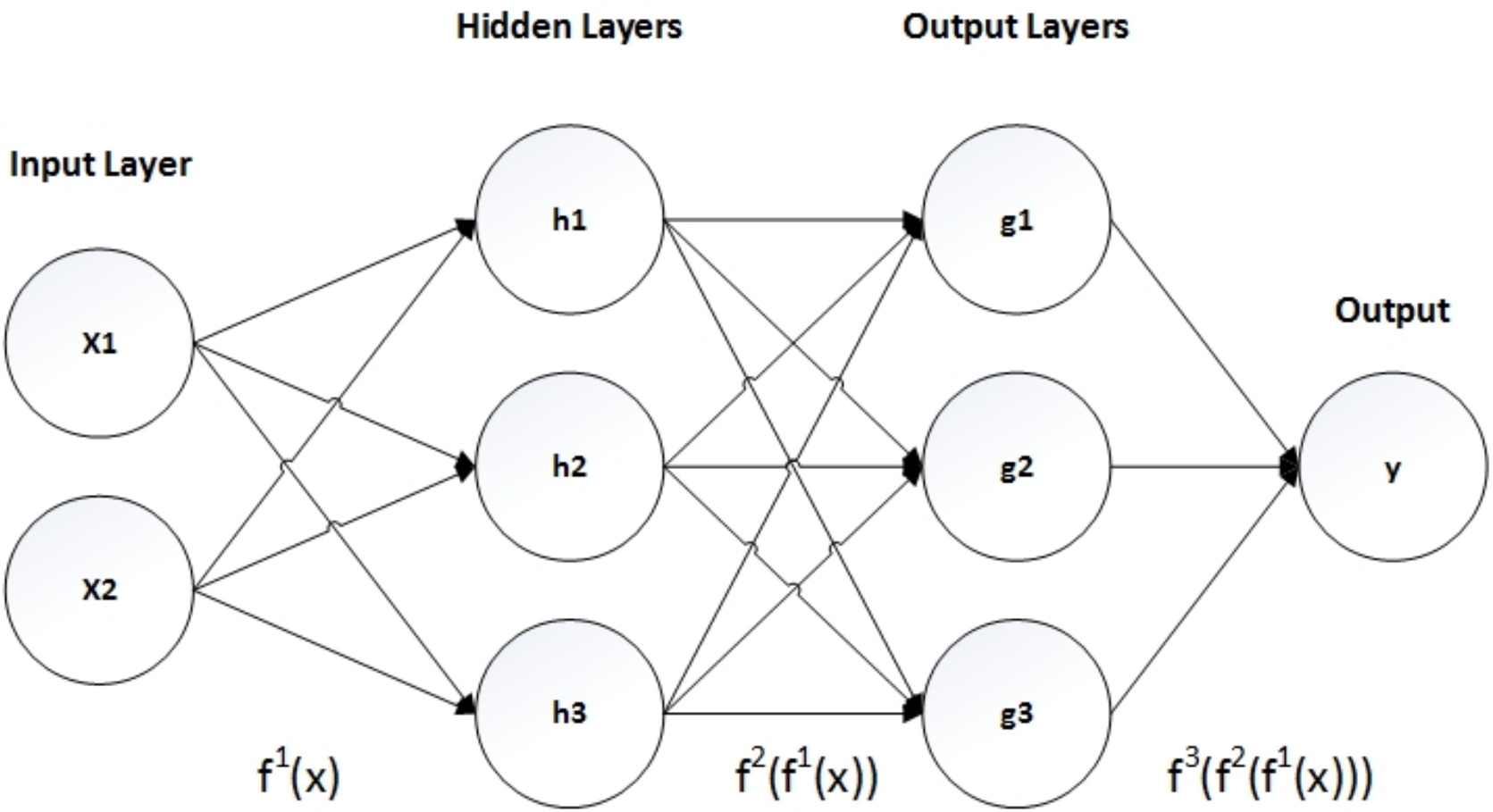
Deep Feedforward Networks

- A **Deep Feedforward Network** defines a mapping $\mathbf{y} = f(\mathbf{x}; \theta)$. During training, the parameters θ are learned so that $\mathbf{y}$ results in the best approximation of the original function $f^*(\mathbf{x})$.
- **Feedforward**: Information flows from the input $\mathbf{x}$ through some intermediate steps, all the way to the output $\mathbf{y}$. There is no **Feedback** connections.

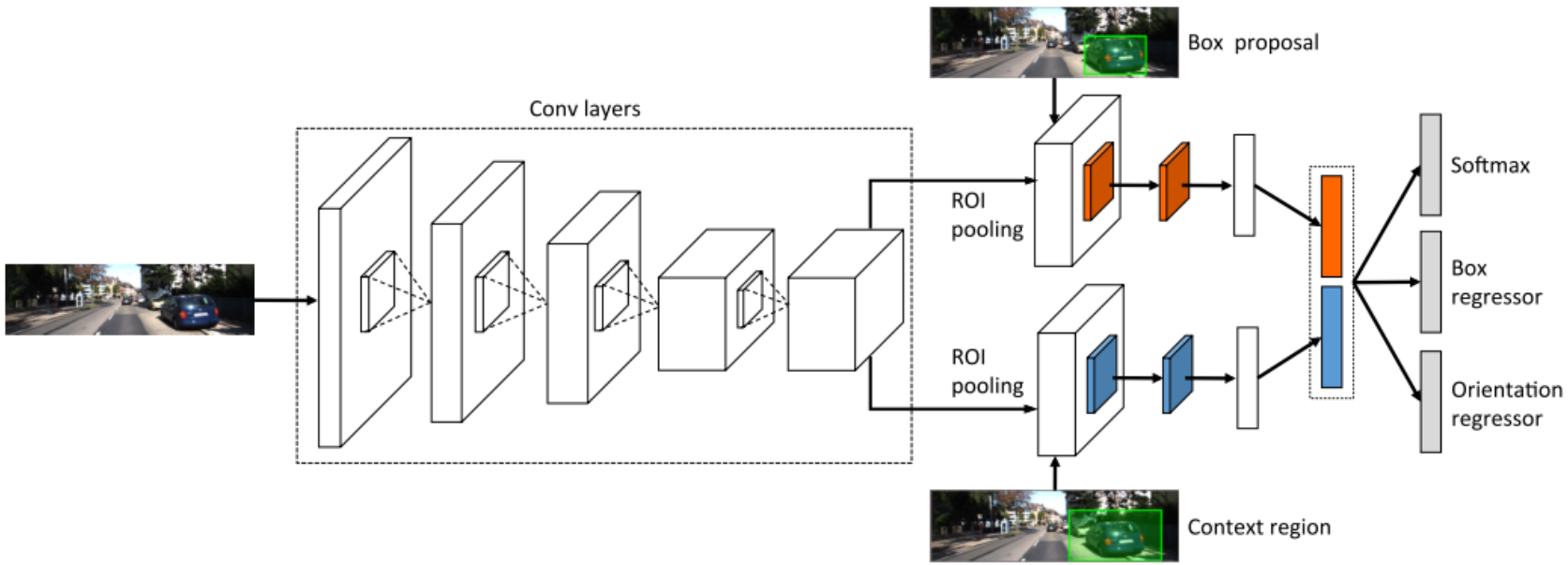
Deep Feedforward Networks

- **Neural Networks**: **Neural** because these models are loosely inspired by neuroscience, **Networks** because these models can be represented as a composition of many functions.
- As an example, a three layer neural network is represented as $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$, where $f^{(1)}$ is called the first layer, $f^{(2)}$ is the second layer, etc ...

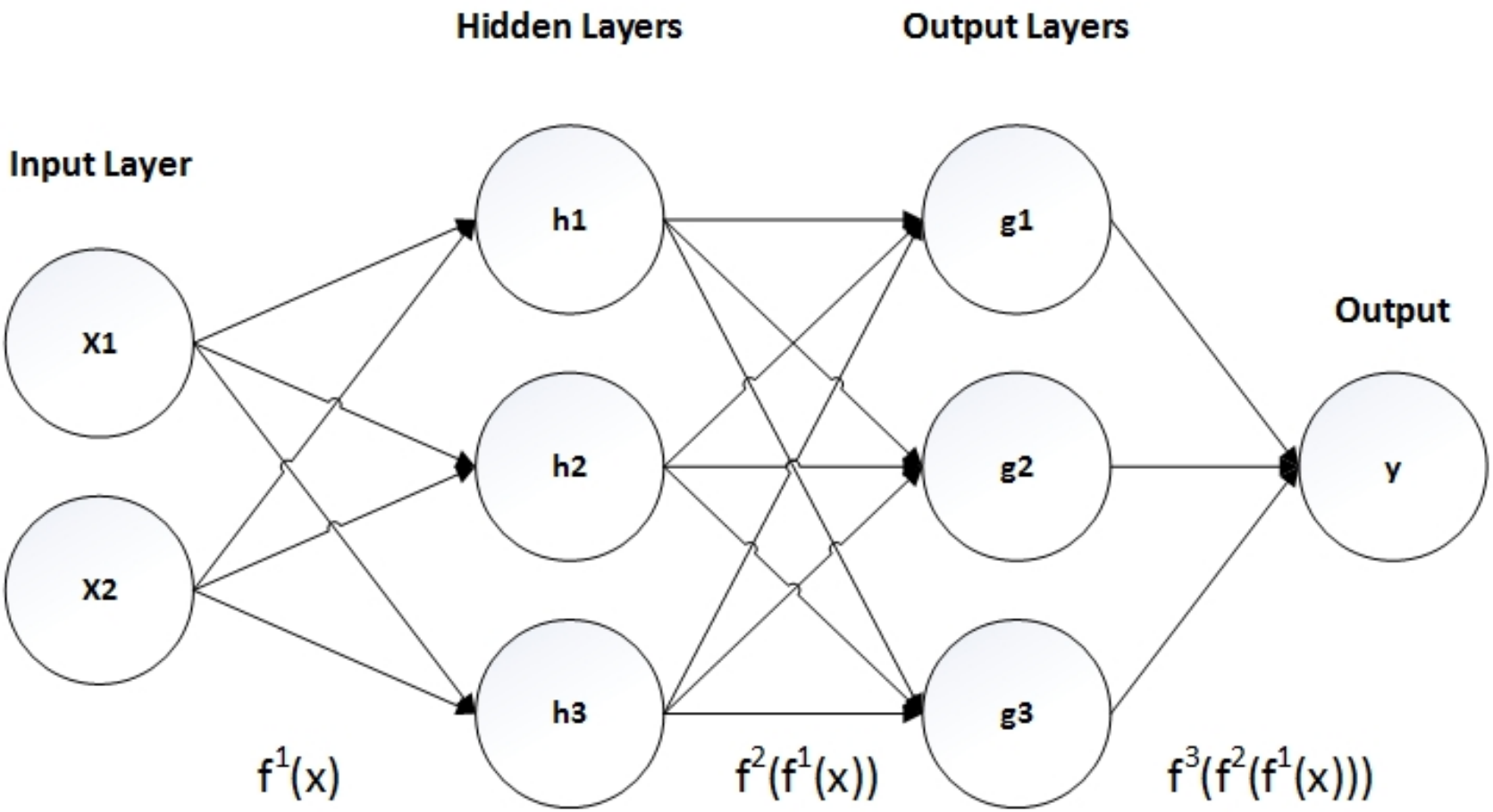
Example:


Hidden Layers
Output Layers
Input Layer
X1
X2
h1
h2
h3
g1
g2
g3
Output
y
$f^1(x)$
$f^2(f^1(x))$
$f^3(f^2(f^1(x)))$

More Realistic Example: Chen et al.(2017)



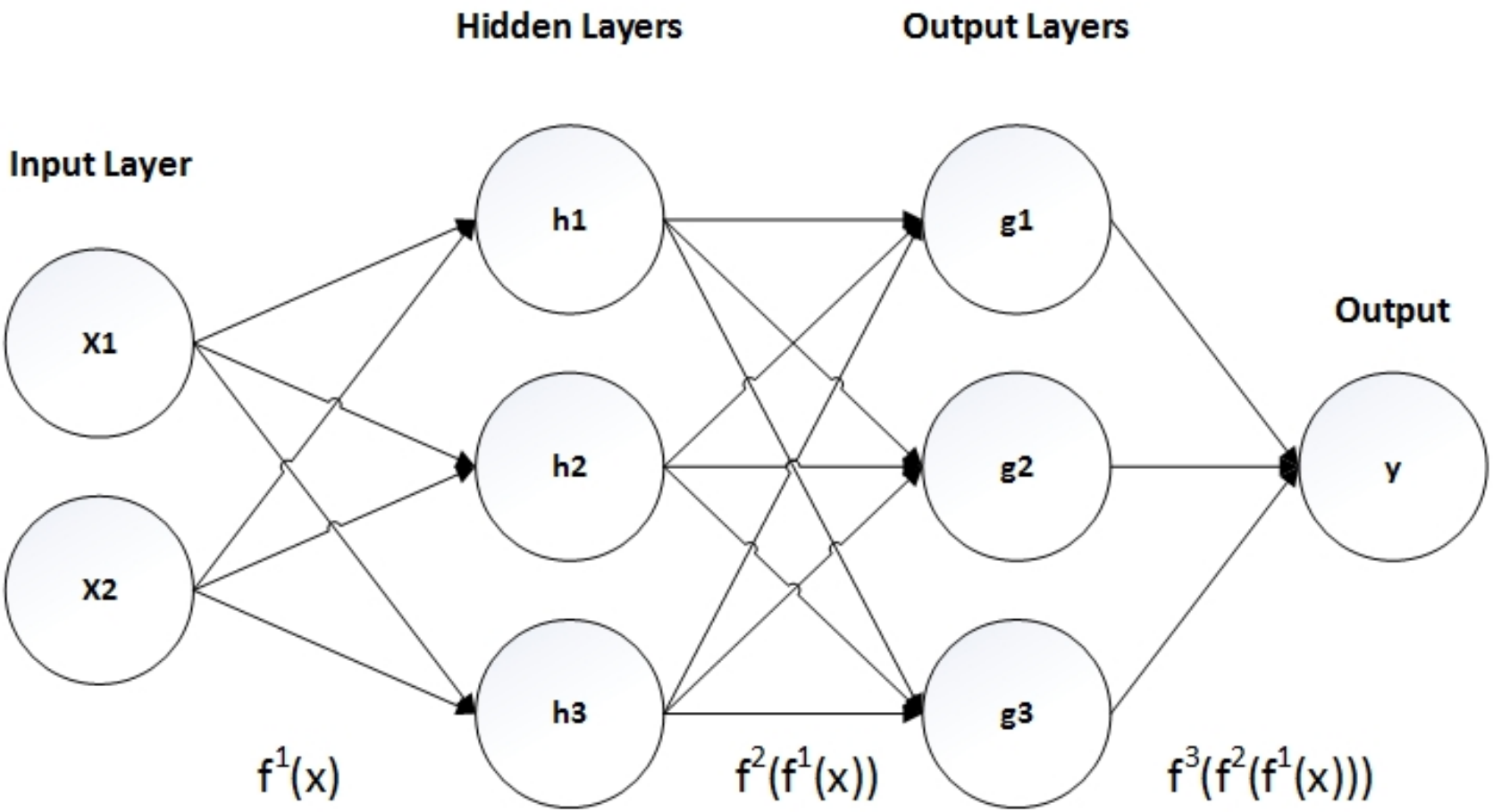
Example:


Hidden Layers
Output Layers
Input Layer
X1
X2
h1
h2
h3
g1
g2
g3
Output
y
$f^1(x)$
$f^2(f^1(x))$
$f^3(f^2(f^1(x)))$

Different Layers of A Neural Network

- **x** is called the **input layer**.
- The final layer $f^{(3)}$ is called the **output layer**.
- The layers in between, $f^{(1)}$ and $f^{(2)}$ are called **hidden layers**.

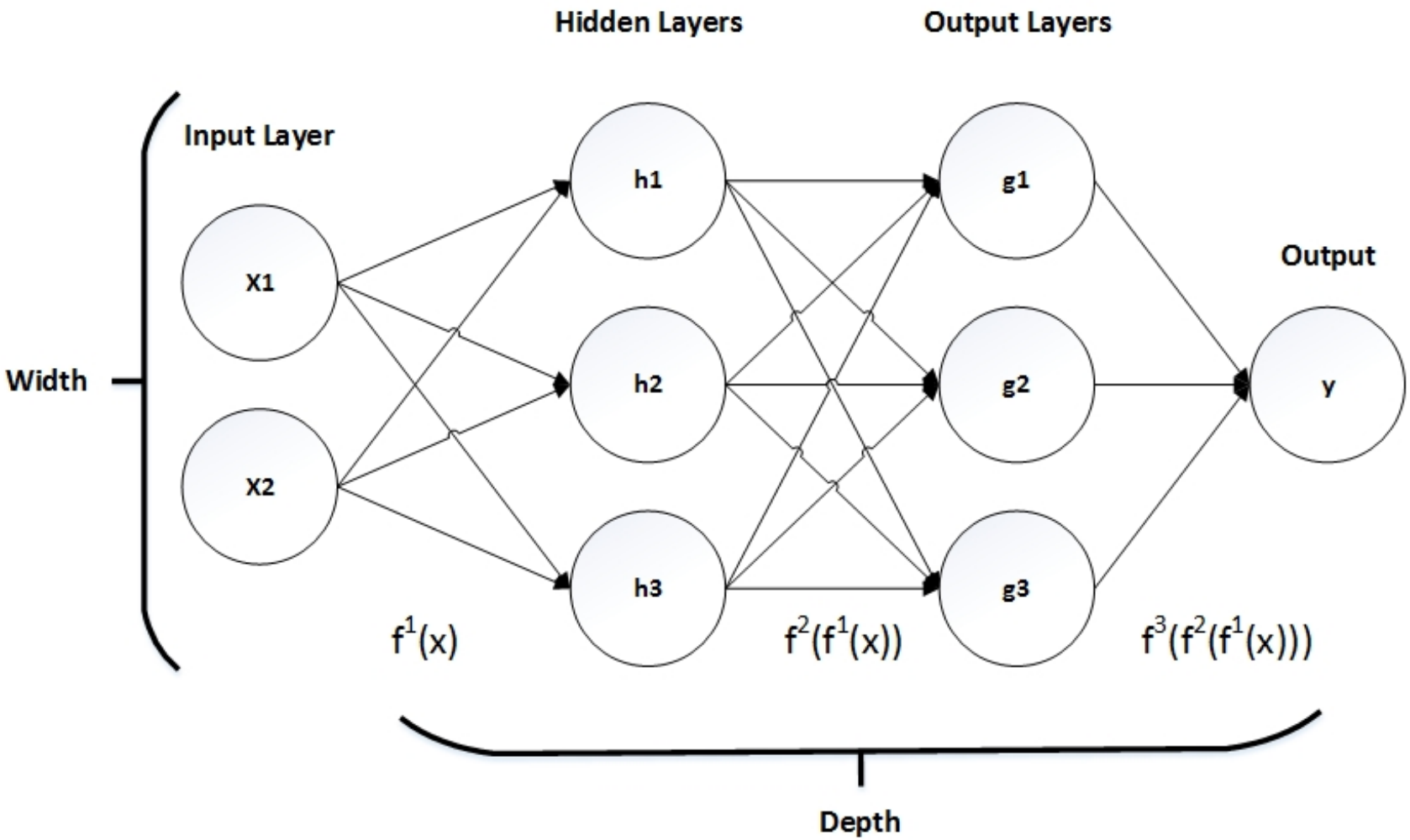
Example:


Hidden Layers
Output Layers
Input Layer
X1
X2
h1
h2
h3
g1
g2
g3
Output
y
$f^1(x)$
$f^2(f^1(x))$
$f^3(f^2(f^1(x)))$

Depth and Width

- The length of the chain of functions in a neural network is called its **depth**.
- The dimensionality of the hidden layers of a neural network is called its **width** .

Example:


Hidden Layers
Output Layers
Input Layer
h1
g1
X1
Output
Width
h2
g2
y
X2
h3
g3
$f^1(x)$
$f^2(f^1(x))$
$f^3(f^2(f^1(x)))$
Depth

The Power of Hidden Layers

- During neural network training, we drive $f(\mathbf{x};\theta)$ to match $f^*(\mathbf{x})$.
- The training data provides us with noisy approximations of $f^*(x)$, each example $\mathbf{x}$ is accompanied by a value or label $\mathbf{y} \approx f^*(\mathbf{x})$.
- Only the output of the neural network is specified for each example. The training data does not specify what the network should do with its hidden layers, the network itself must decide how to modify these layers to best implement an approximation of $f^*(\mathbf{x})$.
- This is referred to as **representation learning**.

Representation Learning

- Consider the linear model: $\mathbf{y} = \mathbf{W}^T\mathbf{x} + \mathbf{b}$.
- What are the advantages of such a model ?
- What are the disadvantages of such a model ?

Representation Learning

- To extend linear models to represent non-linear functions of $\mathbf{x}$, the linear model is usually applied to a transformed input $\phi(\mathbf{x})$.
- How should we choose $\phi(.)$?

Representation Learning

- **1.** Use a very generic $\phi(.)$, such as the features implicitly used by the RBF kernel in kernel machines.
- **2.** Manually engineer $\phi(.)$.
- **3.** Learn $\phi(.)$ from a broad class of functions.

Representation Learning

- The theme of **Representation Learning** extends beyond feedforward networks described in this lecture.
- It is a recurring theme of deep learning that applies to all of the kinds of models described throughout this course.

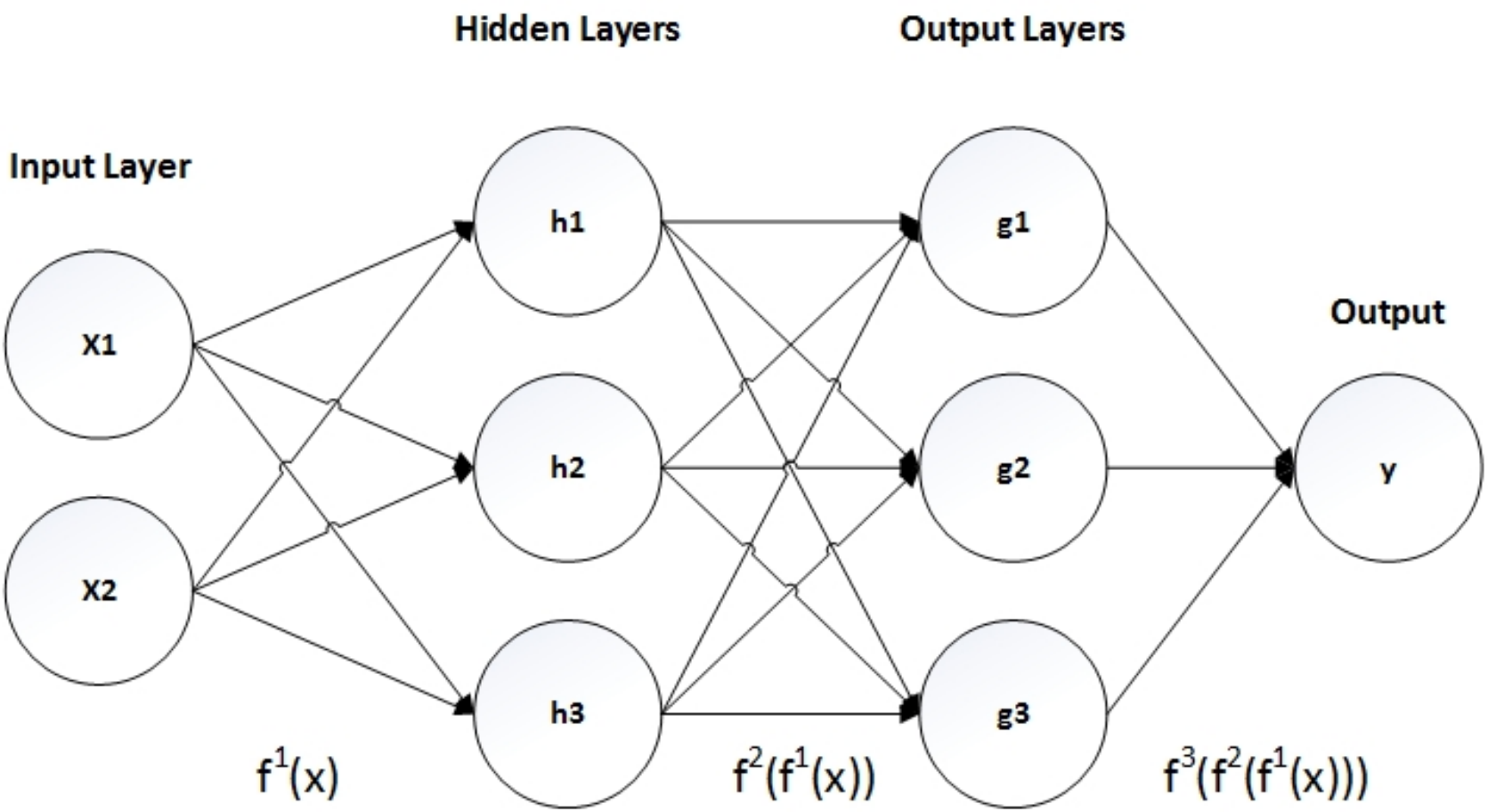
Section 2

The Building Blocks Of Deep Learning

Designing Deep Learning Algorithms

- Not much different than regular machine learning algorithms.
- Specify and optimization procedure, cost function, and model family.
- **Optimization procedures** used for deep learning will have a separate discussion later on throughout the course.
- However, keep in mind the **Gradient** portion of **Gradient** Based Learning !
- For now, we are interested in the design considerations specific to the **cost function** and the **model family**.

Designing Deep Learning Algorithms



The Cost Function

- The **cost function** is a measure of how well our algorithm performs.
- Training is performed through minimizing the cost function.
- As with traditional machine learning algorithms, most neural networks are trained using maximum likelihood. The **Cost Function** in that case is:

$$J(\theta) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{data}} log(p_{model}(\mathbf{y}|\mathbf{x}))$$

- The above cost is referred to as the **cross entropy** between between the training data and the model distribution.

The Output Layer

- The choice of cost function is tightly coupled with the choice of output unit.
- This is because the form of the cross-entropy function depends on how the output is represented.
- Let $\mathbf{h}(\mathbf{x};\theta)$ be the output of the final **hidden layer**.
- Let $\hat{\mathbf{y}}$ be the output of the **whole model**, or $f(\mathbf{x};\theta)$.

The Output Layer: Linear Units

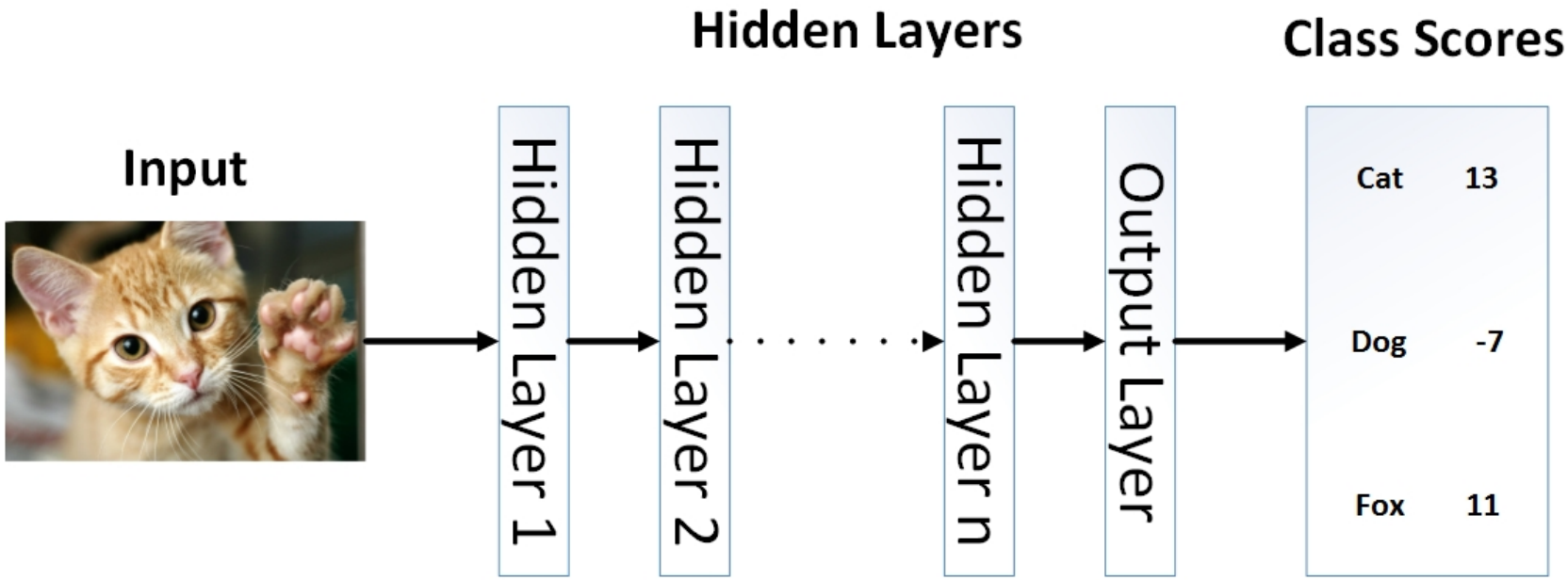
- A simple kind of output unit is one based on an affine transformation with no non-linearity:

$$\hat{\mathbf{y}} = \mathbf{W}^T \mathbf{h} + \mathbf{b}$$

- Often used to produce the mean of a conditional Gaussian distribution, $p(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}; \hat{\mathbf{y}}, \mathbf{I})$
- Using maximum likelihood estimation, the cost function will be the **Mean Square Error**:

$$J(\theta) = \frac{1}{2}\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{data}}||\mathbf{y} - \hat{\mathbf{y}}||_2^2 + const$$

Example: Classification With Linear Output Units, and Multiclass SVM Cost Function



Example: Classification With Linear Output Units, and Multiclass SVM Cost Function

- Define the **Multi-Class SVM Loss** also called the **Hinge Loss** as:

$$J(\theta) = \sum_{i \neq y_{true}} \max(0, f(\mathbf{x}; \theta)_i - f(\mathbf{x}; \theta)_{\mathbf{y}_{true}} + \Delta)$$

- From the previous example, Multi-Class SVM Loss is:

$$J(\theta) = \max(0, -7 - 13 + 1) + \max(0, 11 - 13 + 1) = 0$$

- Other formulations exist, this one follows the Weston and Watkins (1999) version.

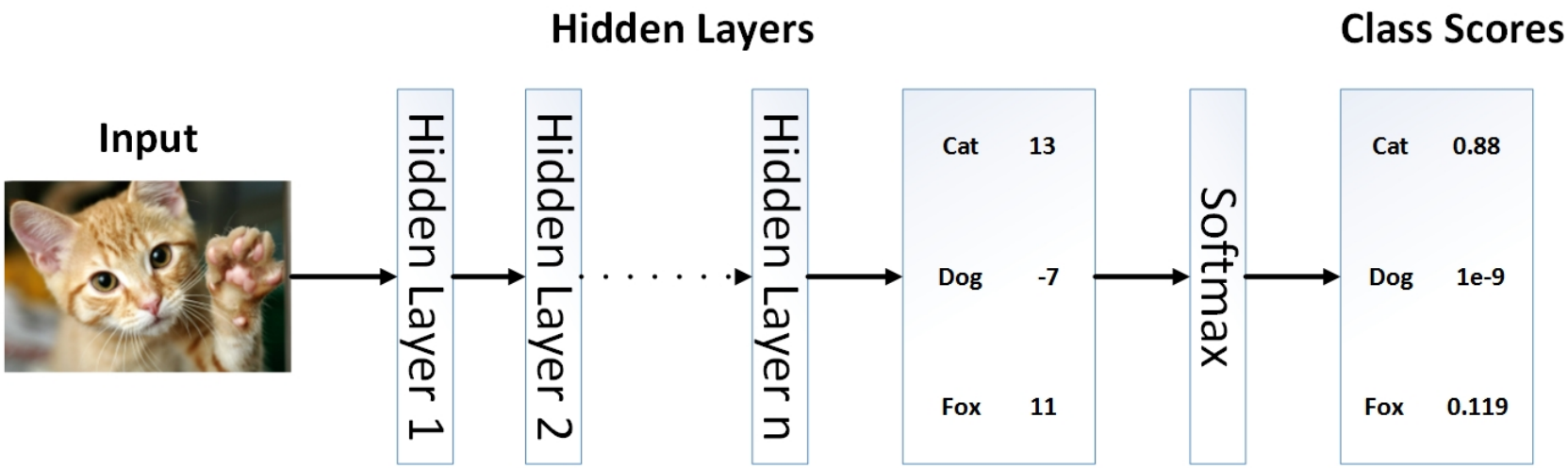
The Output Layer: Softmax Units

- Softmax functions are most often used as the output of a classifier, to represent the probability distribution over K different classes.
- Softmax function on a vector $\mathbf{z}$ is applied element wise as:

$$p(\mathbf{y}_i; \mathbf{z}) = softmax(\mathbf{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}}.$$

- $\mathbf{z}$ is usually the output of a linear unit, $\mathbf{z} = \mathbf{W}^T\mathbf{h} + \mathbf{b}$.

The Output Layer: Softmax Units



- Why is it called soft**max** ?

The Output Layer: Softmax Units

- The cost function is then derived using maximum likelihood:

$$J(\theta) = \log p(\mathbf{y}_i; \mathbf{z}) = \log softmax(\mathbf{z})_i = \mathbf{z}_i - \log \sum_j e^{\mathbf{z}_j}$$

- The first term $\mathbf{z}_i$ always has a direct contribution to the cost function, thus learning can proceed even if the second term becomes very small.
- When maximizing the log-likelihood, the first term encourages $\mathbf{z}_i$ to be pushed up, while the second term encourages all of $\mathbf{z}$ to be pushed down.

The Output Layer: Other Output Types

- The **linear** and **softmax** output units described above are the most common output layers.
- **Sigmoid** units are sometimes used for binary classification.
- **Gaussian mixture models** are employed in mixture density networks. Their usage is primarily multimodal regression. However, it has been reported that gradient-based optimization of conditional Gaussian mixtures on the output of neural networks can be unreliable due to numerical instability.

The Hidden Units:

- So far, we have focused on design choices for neural networks that are common to most parametric models.
- **Hidden units** are what makes deep learning unique.
- Note that these slides are in no way a comprehensive list of hidden units.
- The design of hidden units is an extremely active area of research and does not yet have many definitive guiding theoretical principles.

The Hidden Units:

- So far, we have focused on design choices for neural networks that are common to most parametric models.
- **Hidden units** are what makes deep learning unique.
- Note that these slides are in no way a comprehensive list of hidden units.
- The design of hidden units is an extremely active area of research and does not yet have many definitive guiding theoretical principles.

The Hidden Units:

- Unless otherwise stated, all hidden units discussed in these slides start off with a linear transformation of the input:

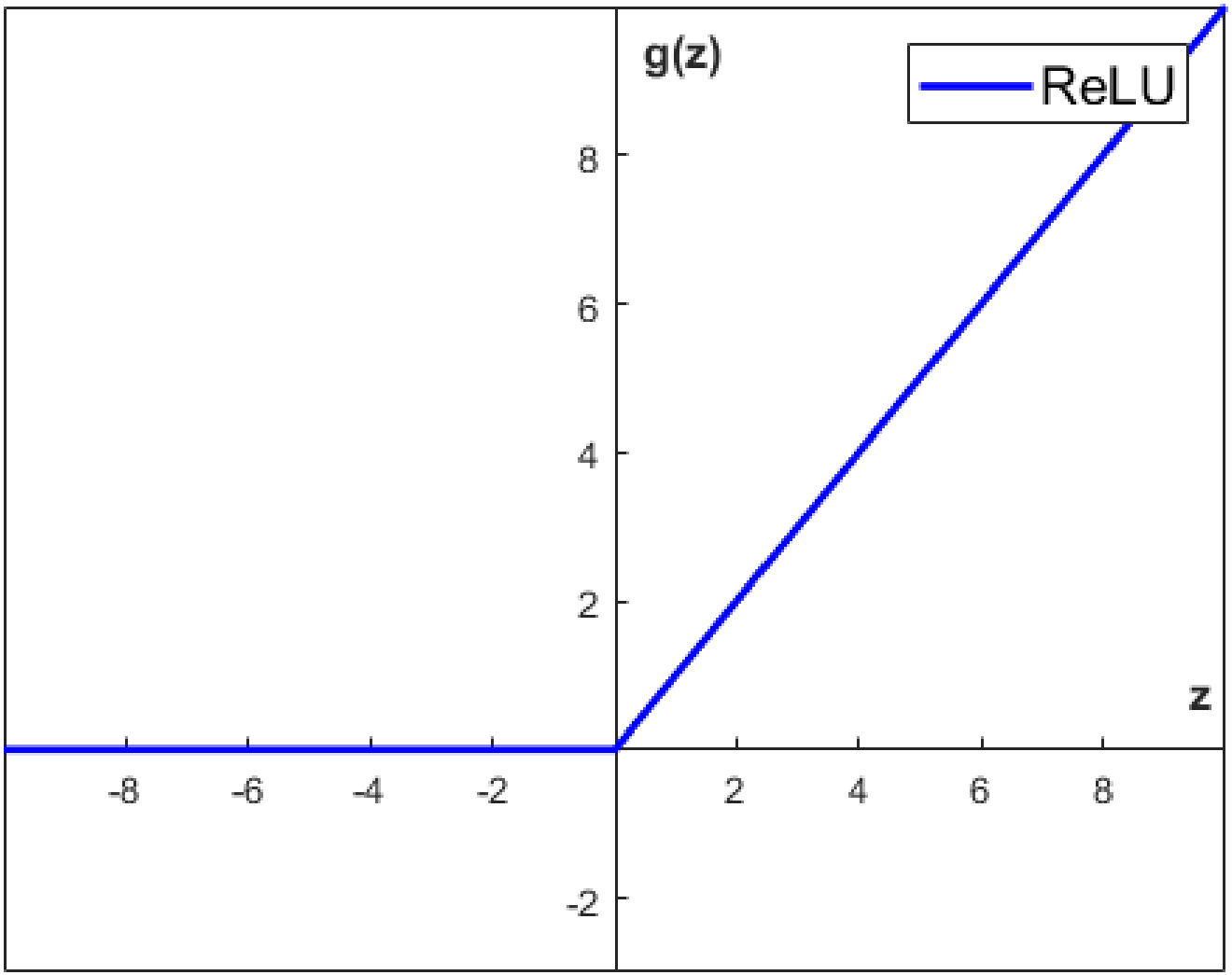
$$\mathbf{z} = \mathbf{W}^T \mathbf{h} + \mathbf{b}$$

- The linear transformation is followed by an **element wise, nonlinear** function $g(\mathbf{z})$. This function is usually called the **activation** function.

The Rectified Linear Units: ReLU

- The ReLU hidden unit are the default choice of activation function for Feedforward Neural Networks.
- The ReLU hidden unit use the function $\max(0, \mathbf{z})$ as its activation function.
- ReLUs are easy to optimize because they are very similar to linear units.
- Derivative through ReLU remain large whenever the unit is **active**.
- The derivative is also consistent, and the gradient direction is far more useful for learning than it would be with activation functions that introduce second order effects.

The Rectified Linear Unit: ReLU



The Rectified Linear Unit: ReLU

- One drawback of ReLUs is that they cannot learn via gradient based methods on examples for which their activation function is zero.
- However, this drawback is not as sever as what we refer to as the **"dead"** ReLU phenomenon.

The Rectified Linear Units: ReLU

- A large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again.
- If this happens, then the gradient flowing through the unit will forever be zero from that point on. That is, the ReLU units can irreversibly die during training since they can get knocked off the data manifold.

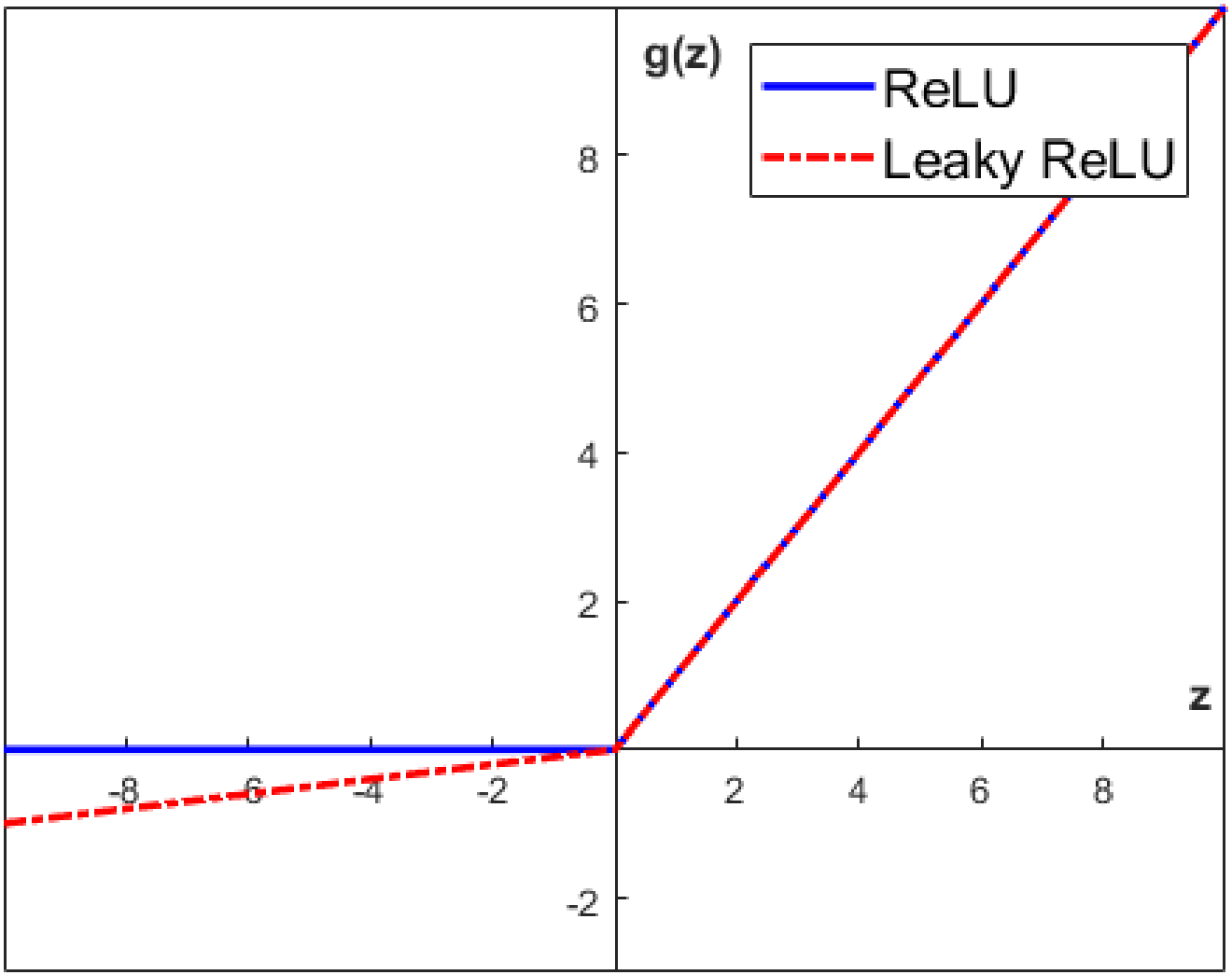
The Rectified Linear Unit: Leaky ReLU

- The **Leaky ReLu** tries to remedy the "dead" ReLU problem by allowing learning to proceed even with $\mathbf{z} \leq 0$.
- This is done through extending the activation function to be:

$$g(\mathbf{z}) = \max(0, \mathbf{z}) + \alpha \min(0, \mathbf{z})$$

- α is usually set to 0.1.
- Parametric ReLu uses the same concept as the Leaky Relu, but treats α as a learnable parameter.

The Rectified Linear Units: Leaky ReLU



The Rectified Linear Units: Maxout

- The Maxout units computes the the function:

$$g(\mathbf{h}) = \max_k (\mathbf{W}_k^T \mathbf{h} + \mathbf{b}_k)$$

- **Maxout** units can be thought of as a generalization to the ReLU units.
- Maxout units can learn a piecewise linear, convex function with up to k pieces. This can be thought of as **learning the activation function itself** !
- With large enough k, Maxout units can learn to approximate any convex function up to an arbitrary fidelity.

The Rectified Linear Units: Maxout (Goodfellow et al. 2014)

- The Maxout units computes the the function:

$$g(\mathbf{h}) = \max_k (\mathbf{W}_k^T \mathbf{h} + \mathbf{b}_k)$$

- **Maxout** units can be thought of as a generalization to the ReLU units.
- Maxout units can learn a piecewise linear, convex function with up to k pieces. This can be thought of as **learning the activation function itself** !
- With large enough k, Maxout units can learn to approximate any convex function up to an arbitrary fidelity.

The Rectified Linear Units: Maxout

- The Maxout units enjoy the benefetis of ReLUs with out having any of their drawbacks.
- Because each unit is driven by multiple filters, Maxout units have some redundancy that helps them to resist a phenomenon called **catastrophic forgetting**, in which neural networks forget how to perform tasks that they were trained on in the past.
- However, each Maxout unit is now parametrized by k weight vectors instead of just one,so Maxout units typically need more regularization than ReLUs.

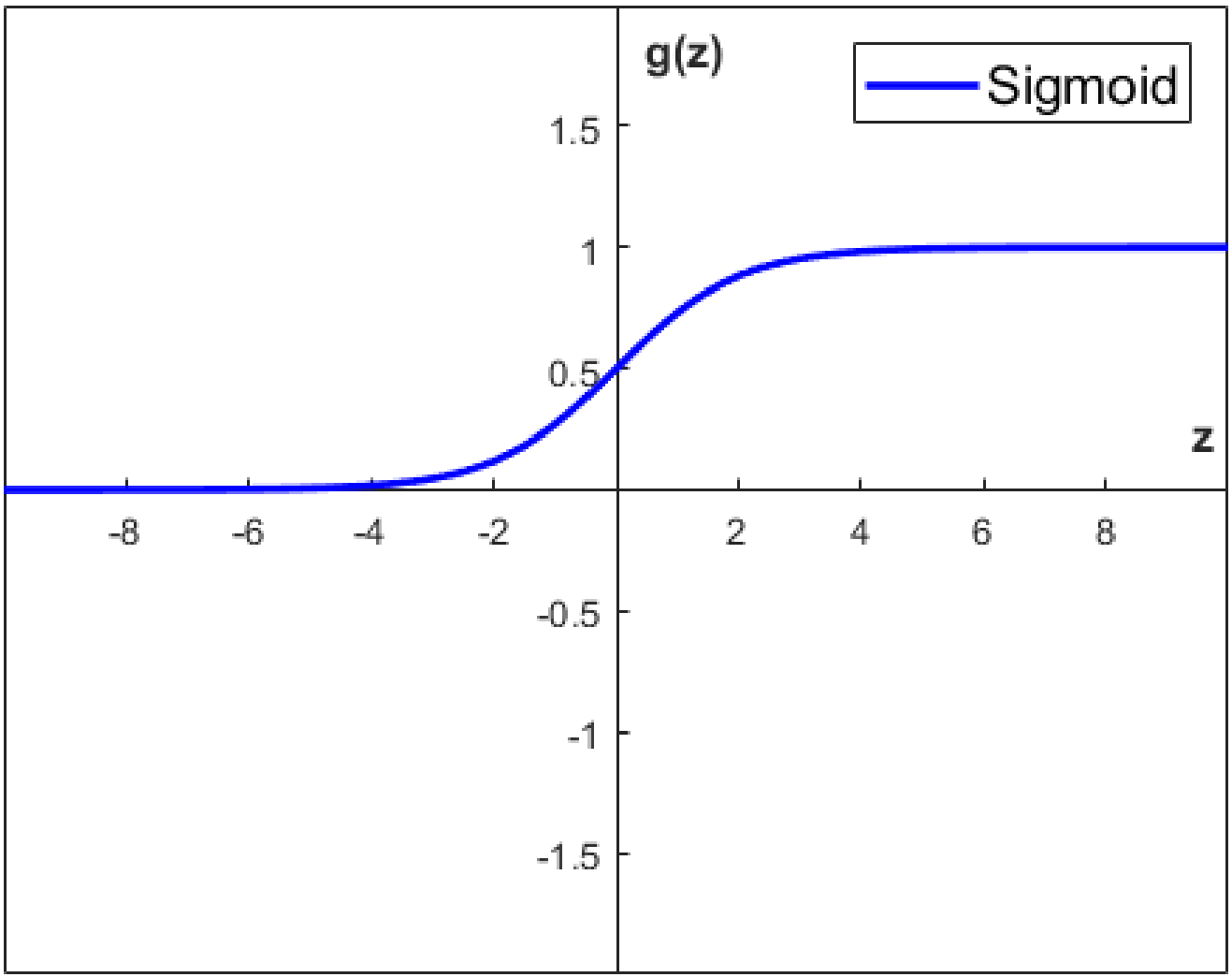
Sigmoidal Units

- Prior to the introduction of ReLUs, most neural networks used the sigmoid activation function:

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

- The sigmoid activation function "squashes" its input to a value between 0 and 1.
- The sigmoid function has seen frequent use historically since it has a nice interpretation as the firing rate of a neuron: from not firing at all (0) to fully-saturated firing at an assumed maximum frequency (1).

Sigmoidal Units



Sigmoidal Units

- Th sigmoidal units have been abandoned in Feedforward Neural Networks for the following reasons:
  - Sigmoidal units saturate and kill gradients. When the value of **z** is at the 0 tail of the sigmoid function, the local gradient is very small. This will result in the sigmoid "Killing" the gradient and almost no signal will flow through the neuron to its weights and recursively to its data. If the value of **z** is on the 1 tail (when the initial weights are very large for example), then the gradient is also almost zero and the network will barely learn.
  - The outputs of Sigmoidal units are not zero-centered. This is less sever than gradient killing, but also affects the gradient decent dynamics.

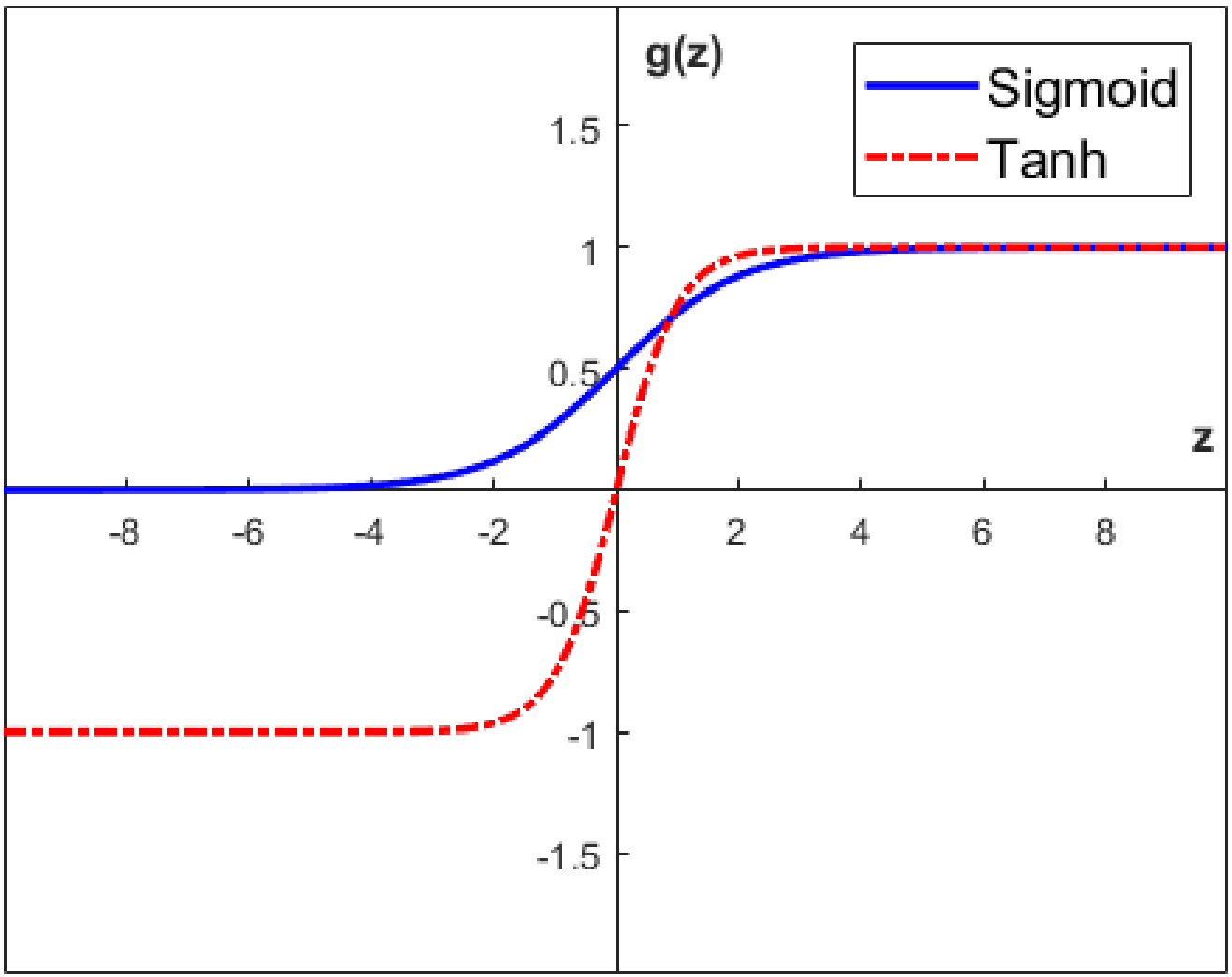
Tanh Units

- Tanh is one other non-linearity that was used prior to ReLUs:

$$g(\mathbf{z}) = tanh(\mathbf{z}) = 2\sigma(2\mathbf{z}) - 1$$

- The Tanh unit attempts to fix the zero-centering problem of the sigmoidal unit. However, the tanh function also has the sigmoid function's same gradient killing characteristics.
- Tanh units is almost always favoured over the sigmoidal unit.

The Rectified Linear Units: Leaky ReLU



Sigmoidal and Tanh Units

- Recurrent networks, many probabilistic models, and some autoencoders have additional requirements that rule out the use of piecewise linear activation functions and make sigmoidal and tanh units more appealing despite the drawbacks of saturation.

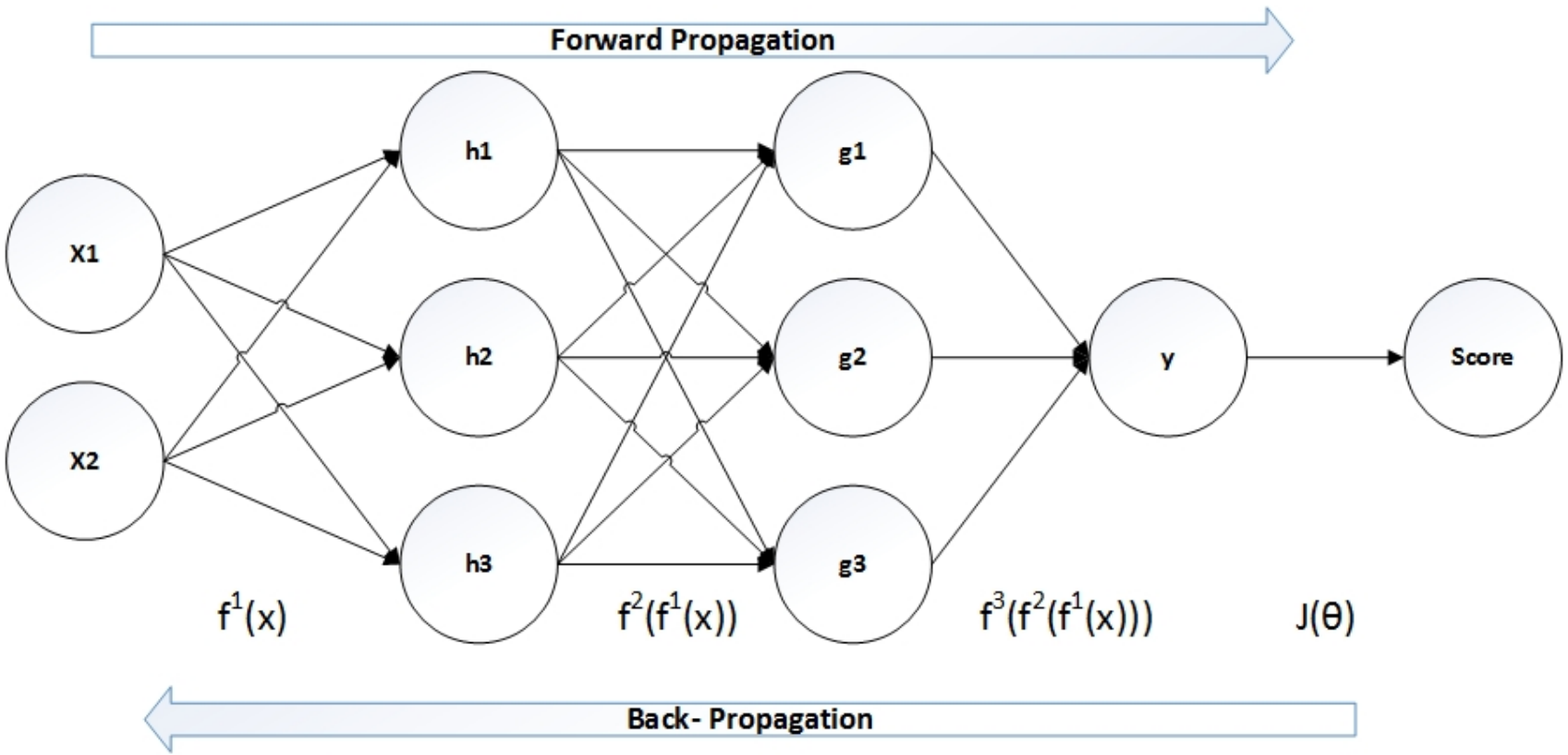
Section 3

Computing The Derivative: Back-Propagation

Forward Propagation And Back Propagation

- When we use the a feedforward neural network to accept an input $\mathbf{x}$ and produce an output $\hat{\mathbf{y}}$, information flows from the input to the cost function $J(\theta)$.
- During training, we need to update the parameters according to the cost function. Back Propagation or **Backprop** for short, allows us to propagate information from the cost function through the parameters.

Forward Propagation And Back Propagation



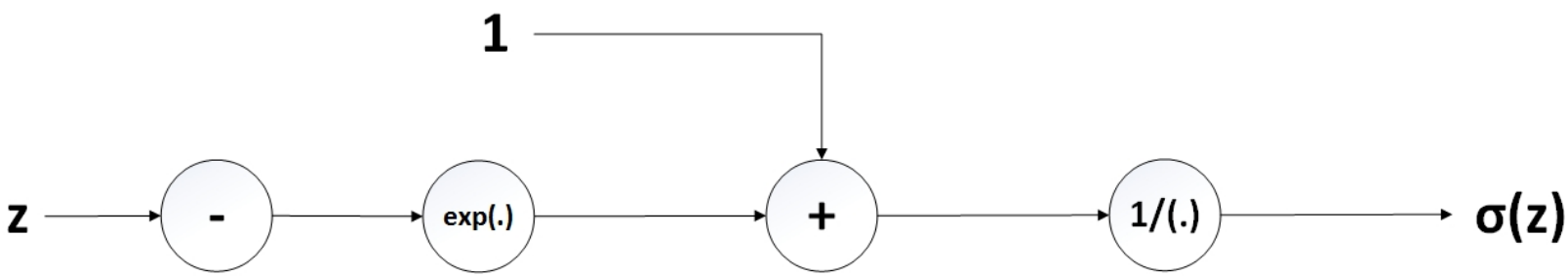
Backprop

- Backprop is **not** the whole learning algorithm, it is merely a method to compute the derivatives. It can be used to compute the derivative of any function, and is not limited to deep neural networks training.
- Backprop relies on applying the **chain rule** recursively to obtain $\nabla J(\theta)$
- Modifying the parameters is done by SGD or other optimization algorithms.

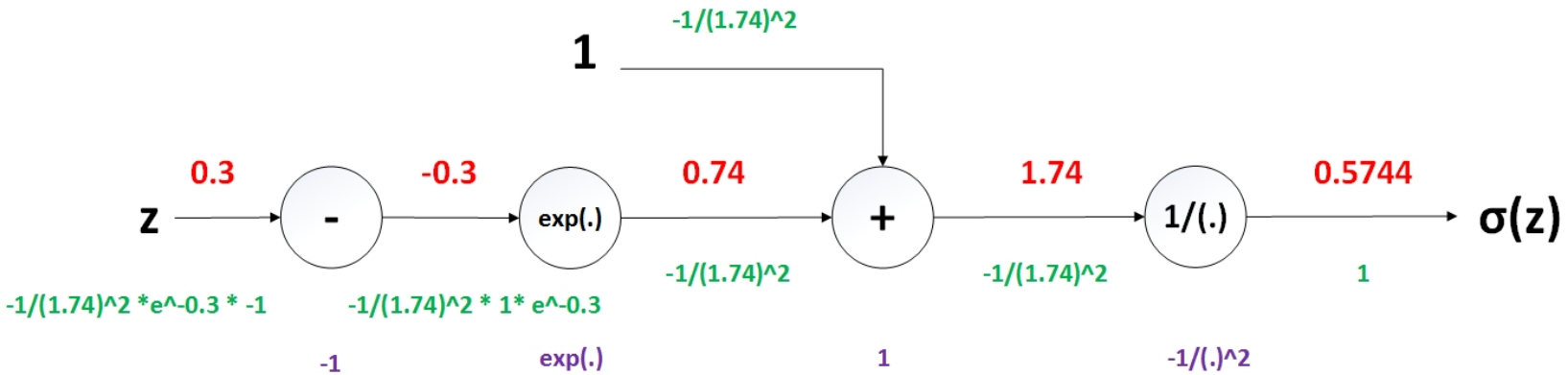
Computational Graph

- Backprop can be easily understood when applied on computational graphs.
- In these slides, nodes of the graph indicate operations.

Example: Sigmoid Function



Example: Sigmoid Function



- Red indicates the forward pass.
- Green indicates the backward pass.

Example: Sigmoid Function

- If we can derive a complex function's gradient, we can assume it is a single operation in our computational graph.
- The derivative of the sigmoid function can be expressed as:

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

Example: Multivariate Function

- The following function is completely irrelevant to deep learning, it will only be used in an example of Backprop.
- Let us formulate the backward pass of the function:

$$f(x, y) = \frac{x + \sigma(y)}{\sigma(x) + (x + y)^2}$$

Section 4

Universal Approximation Properties Of Feedforward Neural Networks

Universal Approximation Properties Of Feedforward Neural Networks

- Feedforward networks can be seen as efficient nonlinear function approximators based on using gradient descent to minimize the error in a function approximation. .
- At first glance, we might presume that learning a nonlinear function requires designing a specialized model family for the kind of nonlinearity we want to learn.
- Fortunately for us, feedforward networks with hidden layers provide a universal approximation framework.

Universal Approximation Properties Of Feedforward Neural Networks

- The **Universal Approximation Theorem** (Hornik *et* al. 1989, Cybenko, 1989) states that a feedforward neural network with a **linear output layer** and at least **one hidden layer** with a **squashing activation function** (sigmoid or tanh for example) can approximate any **Borel measurable** function from one finite dimensional space to the other, with any desired **non-zero amount of error**, provided that the network has **enough hidden units** (width).

Universal Approximation Properties Of Feedforward Neural Networks

- The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well (Hornik *et* al. 1990).
- **Borel measurable**: For the scope of this course, any continuous function on a closed and bounded set in $\mathbb{R}^n$ is borel measurable, and thus can be approximated by a feedforward neural network.
- An extension of this theorem has been provided for the case of ReLU activation functions (Leshno *et* al., 1993).

Universal Approximation Properties Of Feedforward Neural Networks

- Even though the theoretical implications of this theorem are "nice", a feedforward network may fail to *learn* a function even though it can *represent* it.
- Why ?

Universal Approximation Properties Of Feedforward Neural Networks

- Optimization procedures used to train neural networks provide no guarantees on finding the correct parameters corresponding to a desired function. (No global convergence guarantees)
- **The No Free Lunch** theorem shows that there is no universally superior machine learning algorithm. Feedforward networks provide a universal system for representing functions, in the sense that, given a function, there exists a feedforward network that approximates the function. There is no universal procedure for examining a training set of specific examples and choosing a function that will generalize to points not in the training set.

Universal Approximation Properties Of Feedforward Neural Networks

- The universal approximation theorem says that there exists a network large enough to achieve any degree of accuracy we desire, but the theorem does not say how large this network will be. The single hidden layer might be infeasably large. Furthermore, it may fail to generalize well due to overfitting.
- In many circumstances, using deeper models can reduce the number of units in each hidden layer that are required to represent the desired function and can reduce the amount of generalization error.

Conclusion

- In general, the standard choice to increase the capacity of feedforward networks is to go deeper, rather than wider.
- Choosing a deep model encodes a very general belief that the function we want to learn should involve composition of several simpler functions.
- The above prior helps us overcome the **curse of dimensionality** !

Conclusion

Next Lecture, Regularization.