CS 615 – Deep Learning

Deep Learning Building Blocks

Objectives

- Building a Deep Learning System
- The Input Layer
- Connected Layers
- Activation Layers
- Common Activation Functions

Building a Deep Learning System

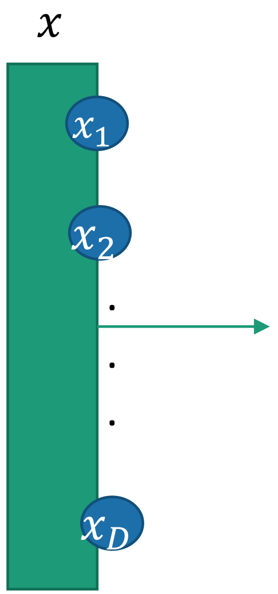
- Creating any deep learning model includes a few key things:

1. The architecture
   - What "modules" are we connecting to one another, and how.
2. The objective function
   - What are we attempting to maximize or minimize?

- Each module in our architecture is referred to as a *layer*.
- The first layer, where our observed data resides, is called the *input layer.*

Input Layer

- The input layer is the raw data, x.
  - **NOTE:** Although the standard linear-algebra convention has a vectors as a *column,* because in ML we typically work with *collection* of observations as a matrix, X, where each *row* is an observation, we'll let an observation x be a *row vector.*
- We visualize this as a set of ***nodes***, one per feature.
- These nodes will be connected to the next layer.
- So that one feature doesn't have more influence than another, just because of its scale, the input layer often *standardizes* the data.
- This means either zero centering the values of each feature or **z-scoring** it (zero centering and making each feature have unit deviation).



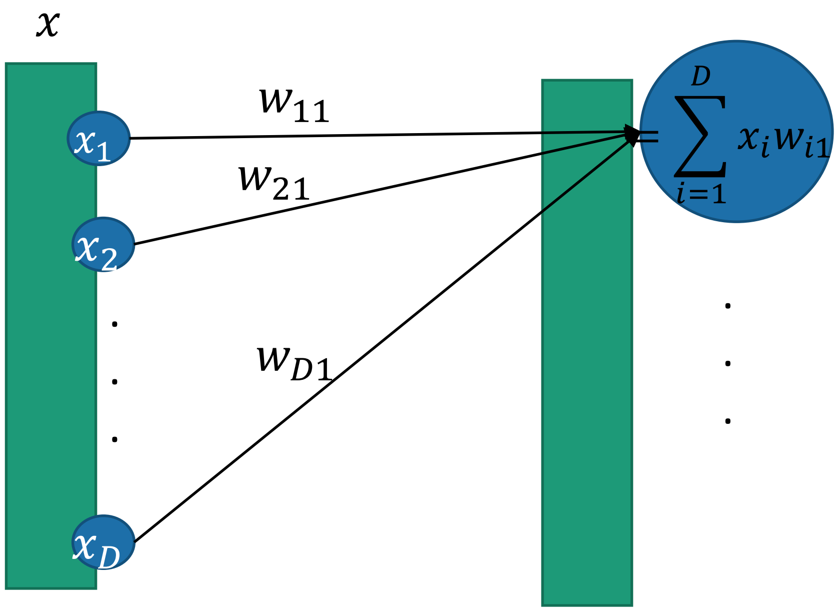
Connected Layers

- Layers that take input from previous layers and use their weighed sum to generate new output values are often referred to as *connected* layers.
- Let w_{ij} be the weight going from node i to node j
- The output at node j is then:

$$h_j = \sum_{i=1}^{D} x_i w_{ij}$$

- If all nodes from one layer connect to all nodes of the next layer, we call these layers *fully connected.*
- Storing all the weights as a matrix, say W, we can compute all the outputs at once as:

$$h = xW$$



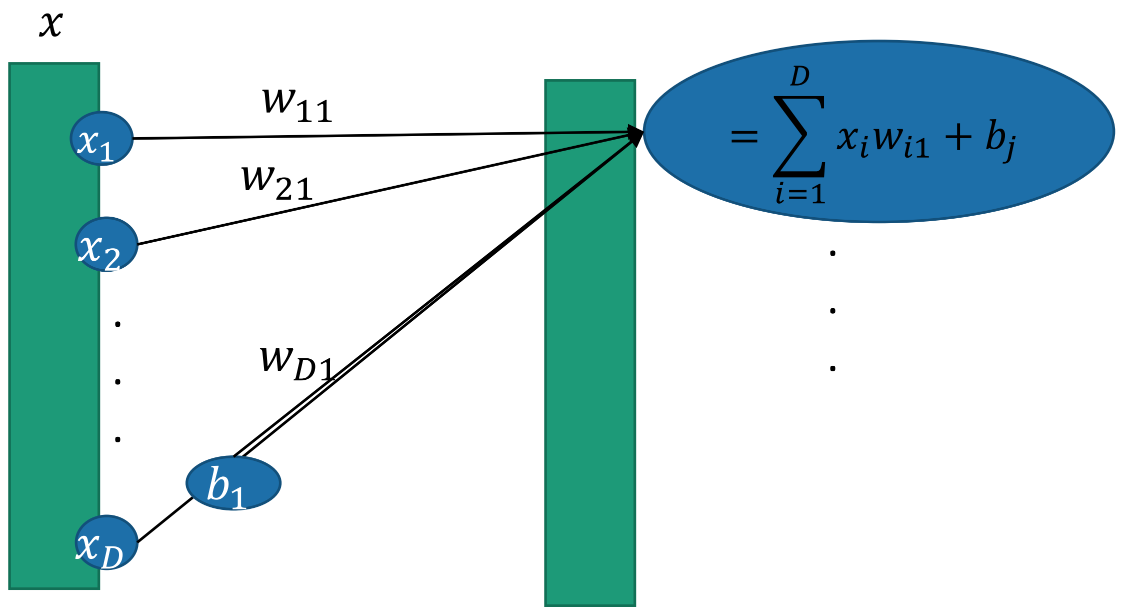
Connected Layers

- Connected layers also tend to have *biases* associated with them.
- These are offset values (almost like a y-intercept in the equation $y = mx + b$)
- Including a bias weight, we get the output at node j is then:

$$h_j = \sum_{i=1}^{D} x_i w_{ij} + b_j$$

- Or in a fully-connected layer:

$$h = xW + b$$



Activation Layers

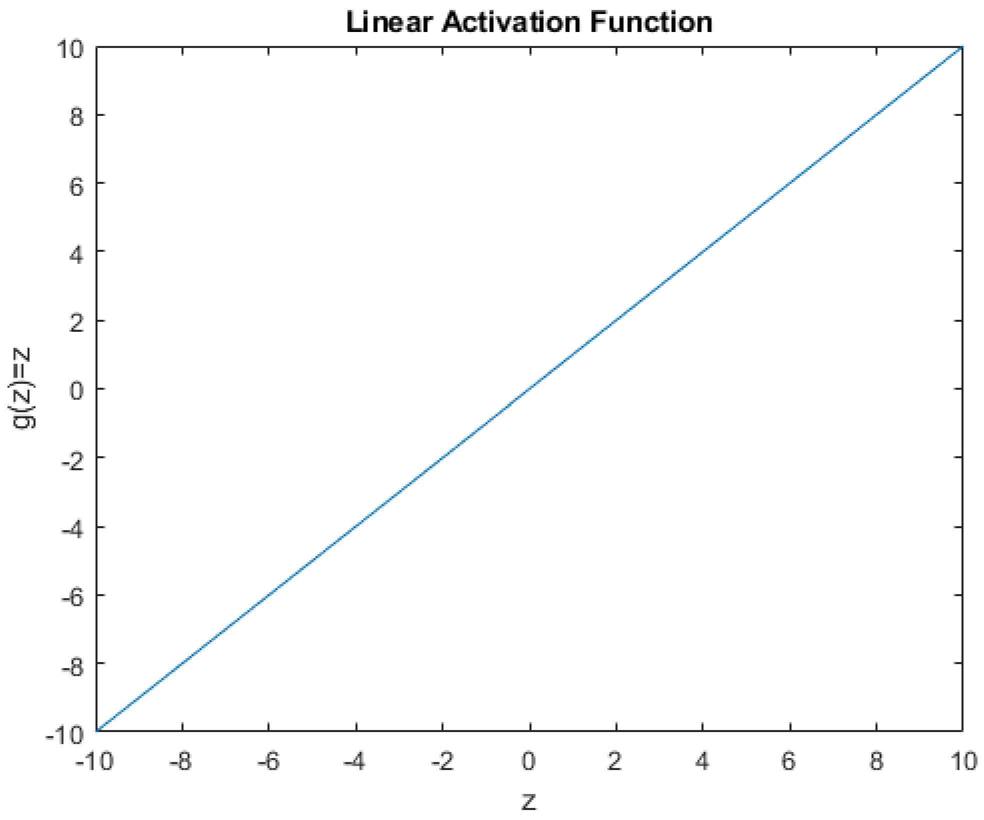
- Layers that apply some element-wise function to a layer to produce some new output are referred to as *activation layers,* and the function being applied are referred to as *activation functions.*
- Some common activation functions include:
  - Linear
  - Rectified Linear Unit (ReLU)
  - Logistic Sigmoid
  - Hyperbolic Tangent (tanh)
  - Softmax
- Let’s look at each of these functions.

Linear Activation Function

- The most straightforward activation function is the linear activation function:

$$g(z) = z$$

- Notes:
  - Differentiable
  - Simple
  - Linear (can't help with complex concepts)

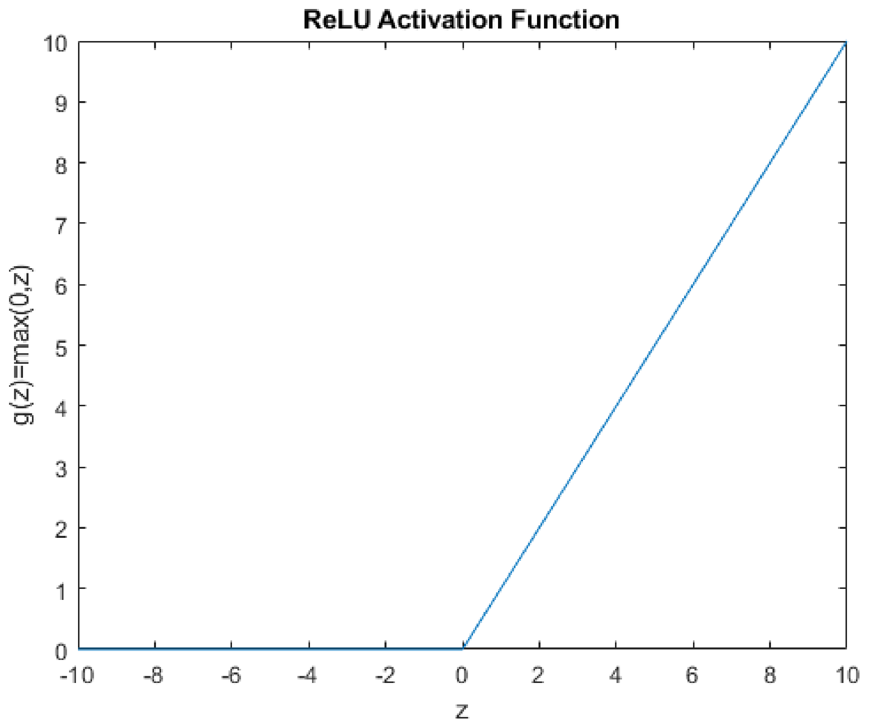


ReLU Activation Function

- Also very straightforward? :

$$g(z) = \max(0, z)$$

- Notes:
  - Piecewise differentiable
  - Simple
  - Avoids negative numbers
  - Non-linear

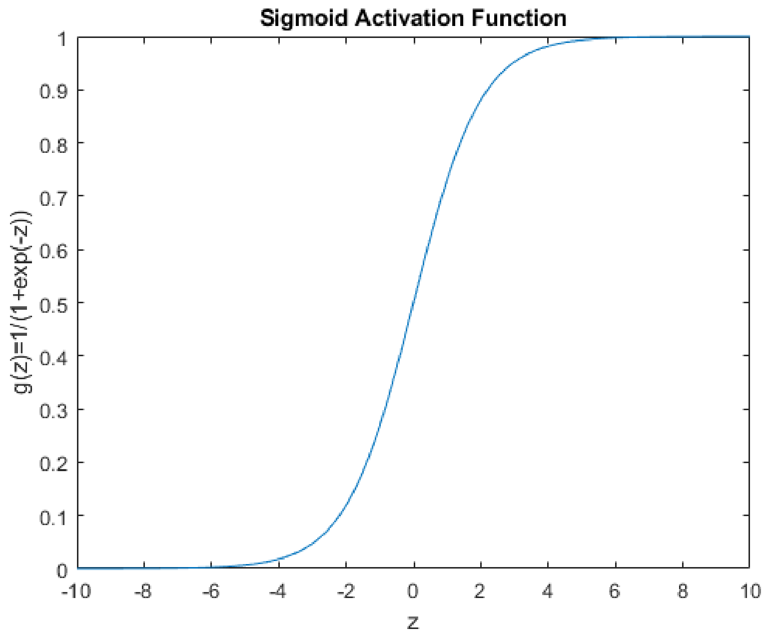


Logistic Sigmoid Activation Function

- The logit (or sigmoid, or logistic) function keeps values in the range of $(0, +1)$

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

- Notes:
    - Differentiable
    - Keeps values in the range of $(0, +1)$
    - Non-linear

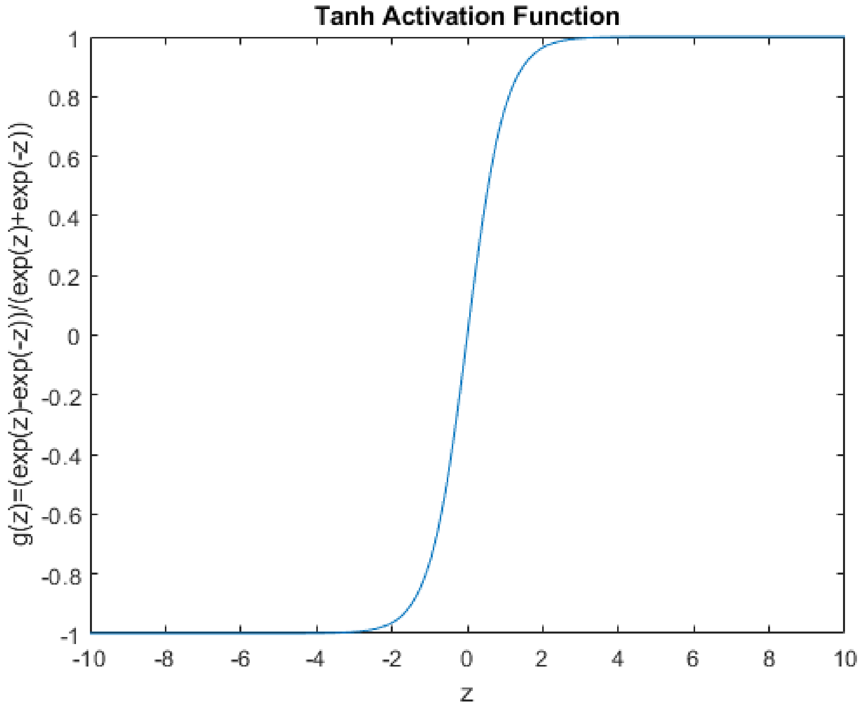


Hyperbolic Tangent Activation Function

- The hyperbolic tangent function is similar to the sigmoid function, but now bounds values from $(-1, +1)$

$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

- Notes:
  - Differentiable
  - Keeps values in the range of $(-1, +1)$
  - Allows for negative numbers
  - Non-linear
  - Symmetric



Softmax Activation Function

- Occasionally we want the output of a layer to be a valid probability distribution (values are the in the range of $[0,1]$) and sum to one.
- One such activation function that does this is the *softmax* activation function.
- Given a **vector** z the softmax is defined as:

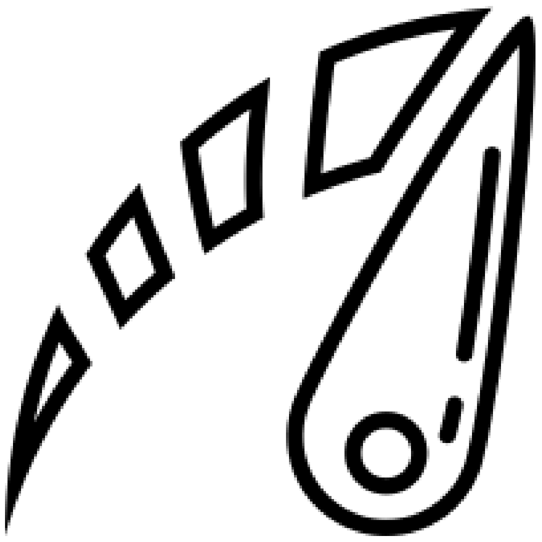
$$g(z) = \frac{e^z}{\sum_i e^{z_i}}$$

- Notes:
  - Keeps values in the range of $[0,1]$
  - Non-linear
  - Results in valid probability distribution.
- To avoid over/underflow it is common to compute this as:

$$g(z) = \frac{e^{z-\max(z)}}{\sum_i e^{z_i-\max(z)}}$$

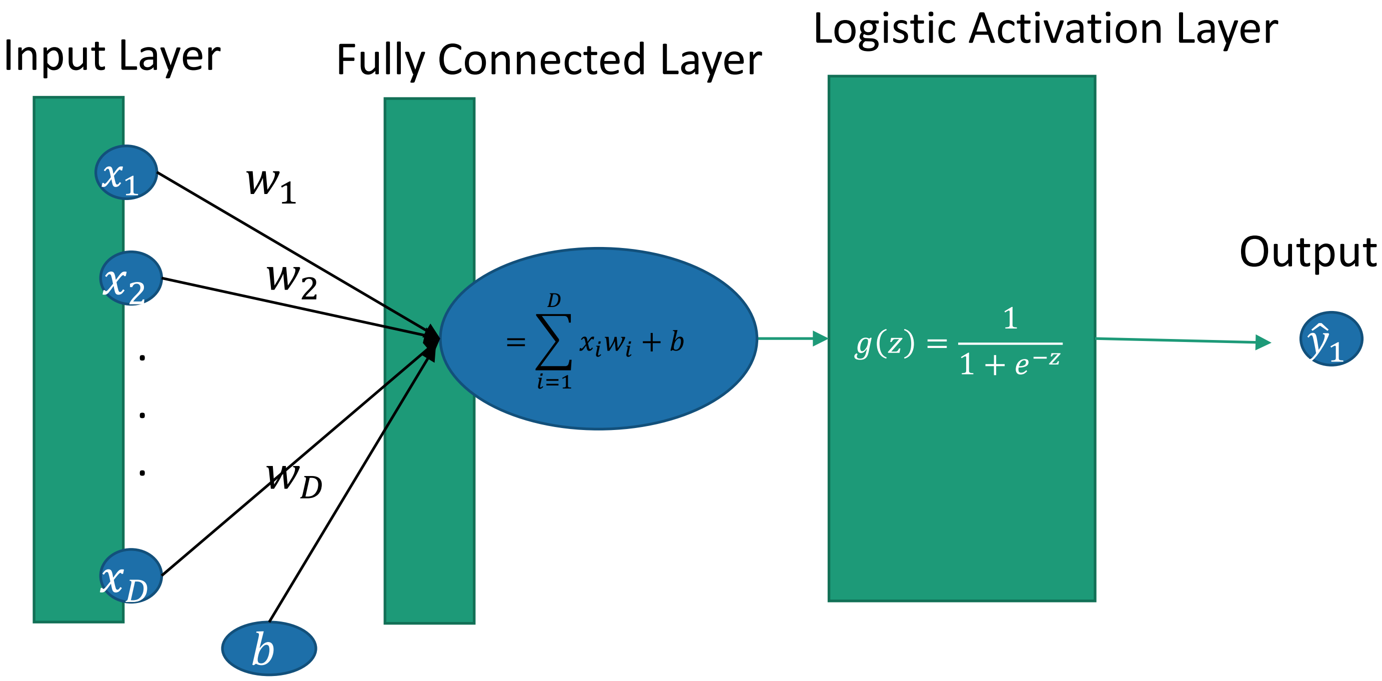
Speed Up

- Note that many of the activation functions directly map an input to an output.
  - That is, each output is dependent on one and only one input.
  - The softmax is an exception.
- In these cases, our multiplication (and division) is typically *element-wise* (as opposed to standard matrix multiplication).
- This is often referred to as the *Hadamard Product* and is denoted as $\circ$ and referred to as the *Hadamard Product*.



Example Architecture

- Now we can define a deep learning architecture!



Modules

- To enforce the idea of modularity, and to reflect how things are done in most DL APIs, we're going to create a *class* for each of our modules.
- Each class will contain information relevant to that type of module.
- These include some subset of
  - A matrix of weights
  - A vector of biases
  - Vectors for the mean and std of the data
- In addition, it will have at least two methods:
  - `forward`
  - `backward`
  - `gradient`

Modules

- The `forward` method takes in data, processes it (according to the module's job) and returns the processed data.
- The `gradient` method computes the change in the outputs as a function of the inputs.
- The `backward` method takes in gradient information, if appropriate, updates weights, and returns the gradient information passed backwards through this module.
- We will talk a lot about the `backward` process (and gradient computation) in the coming weeks.
- But we should already be able to implement the `forward` method for all the modules discussed.

Modules

- To reduce code redundancy and enforce class requirements, we'll start off with an abstract base class that our modules will inherit from.
- This class will include:
  - A constructor with attributes to store the previous incoming and outgoing data.
    - And associated getter and setter methods.
  - A general backward method (coming soon).
  - An abstract forward method.
  - An abstract gradient method.

```
##########BASE CLASS###########
class Layer(ABC):
  def __init__(self):
    self.__prevIn = []
    self.__prevOut = []

  def setPrevIn(self,dataIn):
    self.__prevIn = dataIn

  def setPrevOut(self, out):
    self.__prevOut = out

  def getPrevIn(self):
    return self.__prevIn;

  def getPrevOut(self):
    return self.__prevOut

  def backward(self, gradIn):
    #TODO
    pass

  @abstractmethod
  def forward(self,dataIn):
    pass

  @abstractmethod
  def gradient(self):
    pass
```

Modules

- Here's an example of a SigmoidLayer class implemented in Python.

```
class SigmoidLayer(Layer):
  def __init__(self):
    super().__init__()

  def forward(self,dataIn):
    self.setPrevIn(dataIn)
    Y = 1/(1+np.exp(-dataIn))
    self.setPrevOut(Y)
    return Y

  def gradient(self):
    #TODO
    pass
```