LING530F: Deep Learning for Natural Language Processing (DL-NLP)

Muhammad Abdul-Mageed

muhammad.mageed@ubc.ca

Natural Language Processing Lab

The University of British Columbia

Table of Contents

1. Artificial Neural Networks: Introduction
   - Biological Inspiration
   - Artificial Neurons
   - A Deep Neural Network

2. Feed Forward Neural Networks
   - Solving XOR
   - Non-Linear Units

Feedforward Neural Networks

Biological Inspiration

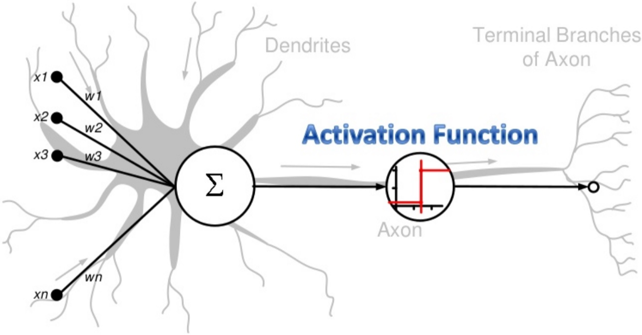

Figure: Information processing in the brain. Weights between neurons model whether they **excite** or **inhibit** one another. Activation can be viewed as **firing rates**. Activation functions and bias model the **thresholded behavior of action potentials**. (Recall: `action potential` is an electric impulse traveling through an axon when a neuron is excited above a threshold). [From Andrew Nelson]

Hubel and Wiesel Cat Experiment

Listen to Neurons

Hubel and Wiesel could listen to the neurons of a cat firing as they moved lines of light in certain directions before the retina of a cat's eye. Listen [here] and [here].

An Artificial Neuron

Figure: One neuron, with an activation function

Neuron With Input

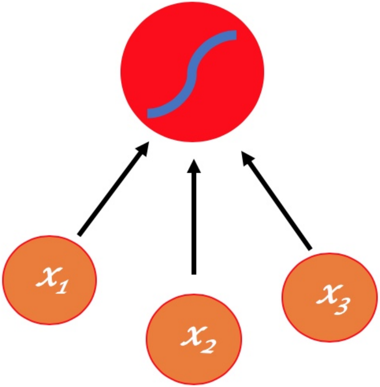

Figure: Input is a vector of **x** with three units, each taking an index j

A Vector For Connection Weights

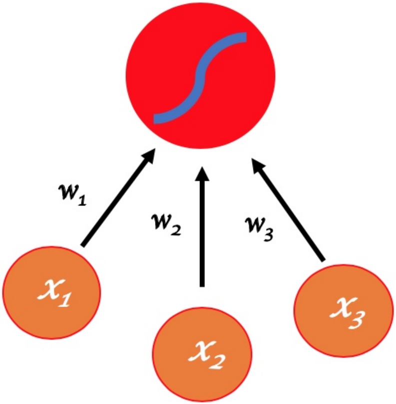

Figure: A vector of free parameters **w** with several items, each taking an index *i*

A Bias Unit

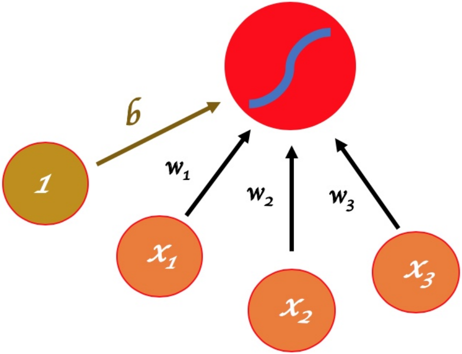

Figure: A bias unit b, equal to 1

A Deep Neural Network

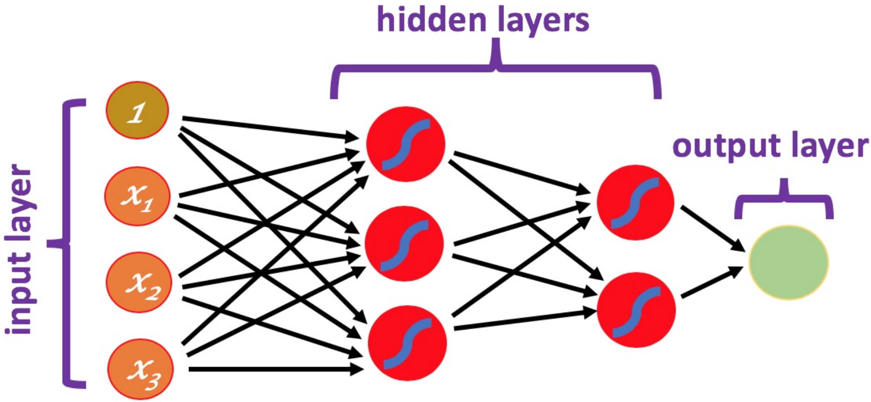

Figure: A **deep neural network**. The network has **2 hidden layers**.

What are Feedforward Nets?

- Also called **multilayer perceptrons (MLP)**.
- A **mapping function** of input x to a category y s.t. $y = f(x; \theta)$
- **Goal:** To learn the value of the parameters θ.
- It is a **directed acyclic graph**, e.g., $f(x) = f^{(3)}(f^{(2)}(f^{(1)}))$.
- **No feedback connections** in which outputs of the model are fed back into itself, otherwise the network would be a **recurrent neural network**.

Learning Good Representations

- We want to **identify a function** $\phi(x)$ that we use to acquire a new representation of x.
- $\phi(x)$ defines a **hidden layer**.
- In DL, we actually **learn this function** ϕ.
- We parametrize the representation as

$$\phi(x; \theta)$$

  and use the **optimization algorithm** to find the θ that corresponds to a good representation.
- In DL, we use **gradient-based optimization**.

XOR Function

- The **XOR function** ("exclusive or") is an operation on two binary values, x_1 and x_2.
- When *exactly one* of these binary values is equal to 1, the XOR function returns 1. Otherwise, it returns 0.

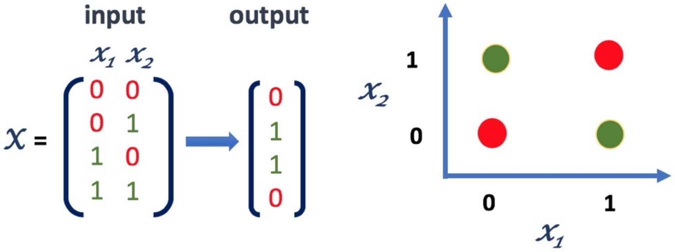

Figure: XOR Function

A Simple Feedforward Neural Network

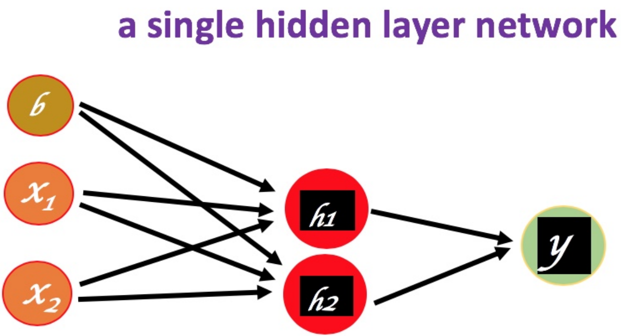

Figure: A simple feedforward network for solving XOR function. Connection weights from input to hidden will be a 2 x 2 matrix $\mathbf{W}$. We will refer to the matrix as $\mathbf{W}^{(1)}$ to denote it is the first layer matrix (input-to-hidden). Weights from hidden to output will be a vector $\mathbf{w}^{(2)}$ of 2 dimensions. We will also add bias.

A Feedforward Neural Network With Weights

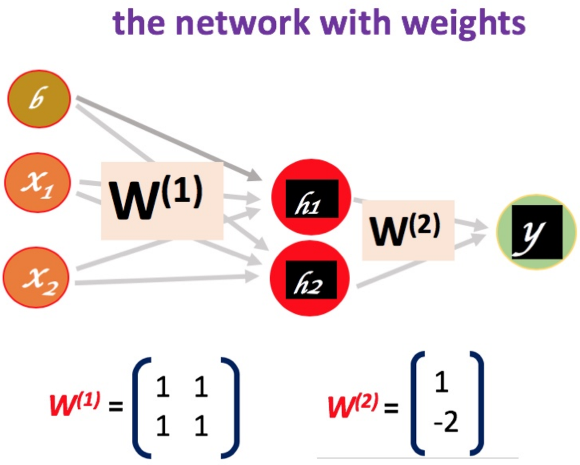

Figure: The network with weights. The hidden layer has one bias unit that is dropped from diagram, for simplicity.

Components of a Feedforward Network (For Solving XOR)

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$c = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$w = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$

$$X = \begin{array}{c} \begin{array}{cc} x_1 & x_2 \end{array} \\ \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \end{array}$$

$$b = \begin{bmatrix} 0 \end{bmatrix}$$

Figure: Ingredients of the Feedforward Network. **(Left) W:** 1st layer weight matrix. **w:** second layer weight vector. **c:** input biases. **(Right) X:** input. **b:** hidden layer bias.

Pre-Activation in One Unit

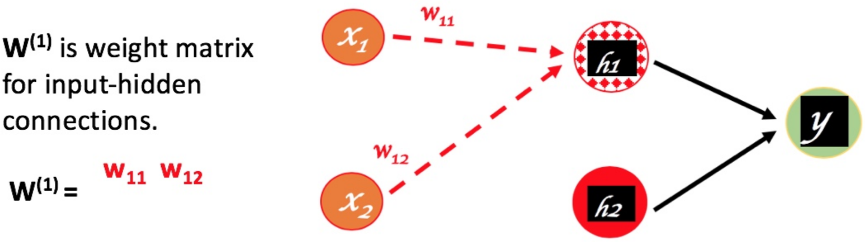

Figure: Connection weights from input to hidden in the 2 x 2 matrix **W** are indexed with i (hidden unit index) and j (input unit index). We refer to the matrix as $\mathbf{W}^{(1)}$ to denote it is the first layer matrix (input-to-hidden). For simplification, bias is dropped.

Pre-Activation in One Unit: More Connection Weights

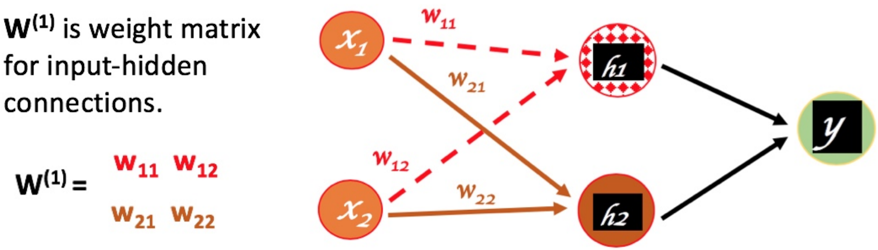

Figure: Connection weights from input to hidden in the 2 x 2 matrix **W** are indexed with i (hidden unit index) and j (input unit index). We refer to the matrix as $\mathbf{W}^{(1)}$ to denote it is the first layer matrix (input-to-hidden).

Calculating a Feedforward Network (Compact)

1: 1-Hidden Layer Feedforward Network

$$\mathbf{h} = f^{(1)}(\mathbf{x}; \mathbf{W}, \mathbf{c})$$

$$\mathbf{y} = f^{(2)}(\mathbf{h}; \mathbf{w}, b)$$

The Whole **feedforward network** then is:

$$\mathbf{f} = (\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b)$$

Computing Hidden Layer

- Let's call $f^{(1)}$: g, and its input: z, so that we have: g(z).
- g will be a nonlinear function, say a Rectified Linear Unit (ReLu), defined as: $g(z) = max(0, z)$.
- The input z: $= \mathbf{W}^{\mathrm{T}}\mathbf{x} + \mathbf{c}$. $\mathbf{W}$: weight matrix.
- Two things happen: (1) The weighted summing, and (2) applying the activation function.

2: Computing Hidden Layer

$$\mathbf{h} = g(\mathbf{W}^{\mathrm{T}}\mathbf{x} + \mathbf{c})$$

Rectified Linear Unit (ReLu)

- For g, we use a **non-linear activation function** like **ReLU**:

3: ReLU Non-Linear Activation Function

$$g(x) = max(0, x)$$

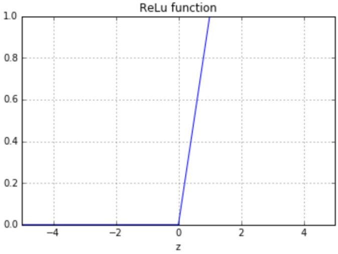

Figure: Rectified Linear Unit (ReLU). We use ReLU for hidden layer activations

- Let's call $f^{(2)}$: o.
- Its input is h, so that we have: o(h).
- o will be a sigmoid (since we do binary activation here).
- For multiclass, we use softmax.

4: Computing Output Layer

Recall:

$$\mathbf{h} = g(\mathbf{W}^{\mathrm{T}}\mathbf{x} + \mathbf{c})$$

Now for output:

$$o = \mathbf{w}^{\mathrm{T}}\mathbf{h} + b.$$

Components of a Feedforward Network (For Solving XOR) I

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \qquad X = \begin{array}{c} \begin{array}{cc} x_1 & x_2 \end{array} \\ \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \end{array}$$

$$c = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$w = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \qquad b = \begin{bmatrix} 0 \end{bmatrix}$$

Figure: Ingredients of the Feedforward Network. **(Left) W:** 1st layer weight matrix. **w:** second layer weight vector. **c:** input biases. **(Right) X:** input. **b:** hidden layer bias.

Calculating The Network II

We can now walk through the way that the model processes a batch of inputs. Let $\boldsymbol{X}$ be the design matrix containing all four points in the binary input space, with one example per row:

$$\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}. \tag{6.7}$$

The first step in the neural network is to multiply the input matrix by the first layer's weight matrix:

$$\boldsymbol{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}. \tag{6.8}$$

Next, we add the bias vector $\boldsymbol{c}$, to obtain

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \tag{6.9}$$

Figure: Calculating the Feedforward I. (Goodfellow et al., 2016, p. 176)

Calculating The Network III

In this space, all of the examples lie along a line with slope 1. As we move along this line, the output needs to begin at 0, then rise to 1, then drop back down to 0. A linear model cannot implement such a function. To finish computing the value of $\boldsymbol{h}$ for each example, we apply the rectified linear transformation:

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \tag{6.10}$$

This transformation has changed the relationship between the examples. They no longer lie on a single line. As shown in figure 6.1, they now lie in a space where a linear model can solve the problem.

We finish by multiplying by the weight vector $\boldsymbol{w}$:

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \tag{6.11}$$

Figure: Calculating the Feedforward II. (Goodfellow et al., 2016, p. 176)

Sigmoid Units for Bernoulli Output Distributions

- Used for tasks requiring predicting a **binary value** for y.
- Example: **positive** vs. **negative** sentiment.

5: Sigmoid Function

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

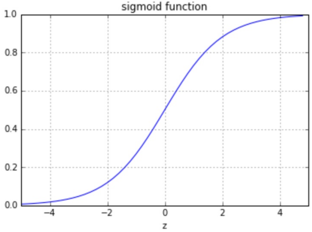

Figure: A plot of sigmoid.

On Sigmoid Units

6: Properties of Sigmoid

As activation becomes very small (with $e^{very\_large\_number}$), it gos to zero:

$$a \rightarrow -\infty : \sigma(a) \rightarrow 0$$

As activation becomes very large (with $e^{-very\_large\_number}$), it gos to 1:

$$a \rightarrow \infty : \sigma(a) \rightarrow 1$$

If a $= 0$, the function outputs $\frac{1}{2}$
The function is differentiable, with a nice form:

$$\Delta\sigma(a) = \sigma(a)(1 - \sigma(a))$$

Sigmoid Code Example I

```
import numpy as np
def sigmoid(x, derivative=False):
    return x*(1-x) if derivative else 1/(1+np.exp(-x))

"""
Note 1: We threshold value of activation a at -1 from below
(which will be what we will get if we used an activation
function like tanh [ouputs values between -1 and 1] from previous layer. If output
activation came from a ReLu, it will be bounded by 0/zero from below)
"""
a=0.01
print(sigmoid(a))
```

```
0.502499979167
```

```
a=1000
print(sigmoid(a))
```

```
1.0
```

```
a=-0.5
print(sigmoid(a))
```

```
0.377540668798
```

Sigmoid Code Example II

```
import numpy as np
def sigmoid(x, derivative=False):
    return x*(1-x) if derivative else 1/(1+np.exp(-x))

"""
Note 2: The function can take an array, although as an output unit it is
used over a single unit for binary classification
"""
a=np.array([0.01, 0.2, 0.5, 0.8, 1.0, 2.5, 5.0, 100.0, 5000.0])
print(sigmoid(a))
```

```
[ 0.50249998  0.549834    0.62245933  0.68997448  0.73105858  0.92414182
  0.99330715  1.          1.        ]
```

Softmax Units for Categorical Output Distributions

- A **Categorical distribution** (also called generalized Bernoulli or Multinoulli) is a **probability distribution over a discrete variable with n possible outcomes**.
- Example: **{joy, sadness, anger, surprise}** for emotion is an example.
- We can use the **softmax** function for this.
- **Softmax is often used as the output of a classifier**, to represent a probability distribution of n different classes.
- To calculate a softmax, we produce a vector $\hat{\mathbf{y}}$, with $\hat{y}_i = P(y = i|x)$:

Softmax Output

- **each element of $\hat{y}_i$ is between** 0 **and** 1.
- The **entire vector $\hat{\mathbf{y}}$ sums to** 1 so that it represents a valid probability distribution.

Softmax

7: Softmax

First, a linear layer predicts unnormalized log probabilities:

$$z = W^\top h + b$$

where

$$z_i = \log \hat{P}(y = i|x)$$

Then the softmax can **exponentiate and normalize z** to obtain $\hat{\mathbf{y}}$:

$$softmax(z)_i = \frac{exp(z_i)}{\sum_c exp(z_c)}.$$

Where the **sum is over all the units/classes** (= same number of classes we are predicting).
We then predict the class with the **highest predicted probability**.