# Building a Multilayer Perceptron

August 22, 2018

## 1 Setting up a neural network

### 1.1 Plan for setting up our first neural network

- A simplified approach views an artificial neural network as a synthetic brain (most researchers don't like the analogy, although we'll use it as heuristic here)
- For a brain to learn there are a couple of steps that need to occur:

    1. Define how the neurons are connected with one another
    2. Expose it some data and some problem (i.e. classification)
    3. Have a mechanism to assess how well it evaluates the data
    4. Have a feedback mechanism to update it's understanding of the data
    5. Repeat steps 3-5 until it has solved the problem

- To build our network we proceed in two sections:

    1. In the first we'll setup the architecture of a neural network so that we can **forward propagate**
    2. In the following section we'll learn how to train a neural network through **back propagation** and gradient descent

- We've already seen a simplified version in the logistic regression unit and will leverage various ideas developed there to build our understanding

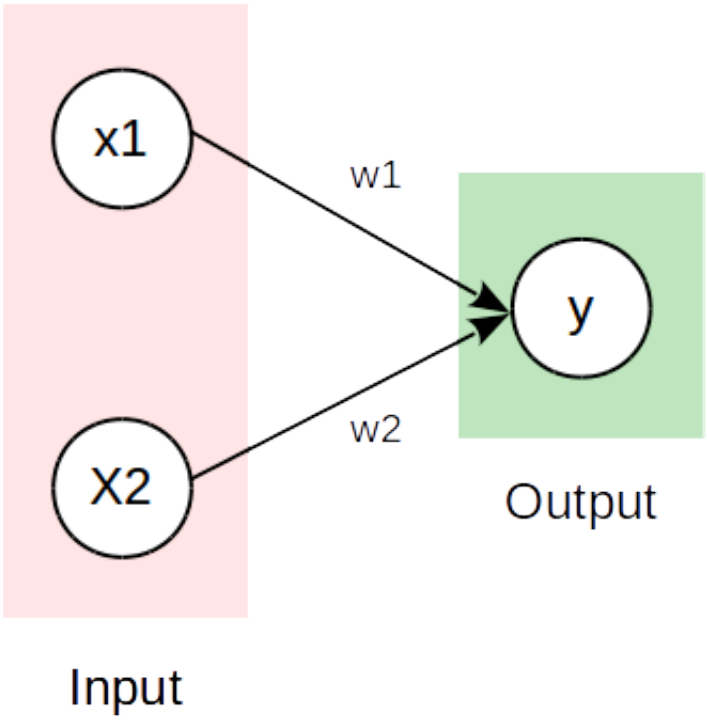
### 1.2 Components of a neural network

We will use graphical models to facilitate our understanding of composing neural networks. In particular we'll show you first how to represent a logistic (sigmoid) function which is a 1-layer neural network where the activation function is logistic unit. Next we will define the architecture of a 2 layer neural network (or equivalently a 1-hidden layer neural network since the hidden layer is never truly observed).

The goal of using graphical models is to simplify networks representations since they can become deep and dense. By using graphical models we plainly explaing relations between different units in the network.

#### 1.2.1 A 1-layer neural network

Let's first visualize our logistic function.




Logistic Function Graph

The graph above provides an expression for how the variables are **related to one another** but doesn't say anything about how the variables are transformed. It is non-specific and need not apply to logistic units. We usually assume that the inputs are additive and that once they get to the second node some type of transformation occurs.

When $y$ is assumed to be a logistic unit the graph can be interpreted as follows:

$$\begin{cases} a = \mathbf{w} \cdot \mathbf{x} + b & \text{Linear Activation Function} \\ y = \sigma(a) = \frac{1}{1+exp(-a)} & \text{Sigmoid transform of } a \end{cases}$$

where: - $a$ is the *activation function*, - $\mathbf{w}$ are the *weights*, - $\mathbf{x}$ are the *features*, - $b$ is the *bias*, - $\sigma(\cdot)$ is the *sigmoid (logistic) transform* and - $y$ is the *target variable* (sometimes denoted $t$ for "target") Clearly we could replace $\sigma(\cdot)$ with some other function and the graph would still hold. Note: If $\sigma(\cdot)$ was replaced by a function $f(\theta)$ of the form:

$$f_Y(y, \theta) = h(y)exp(\eta(\theta) \cdot T(y) - A(\theta))$$

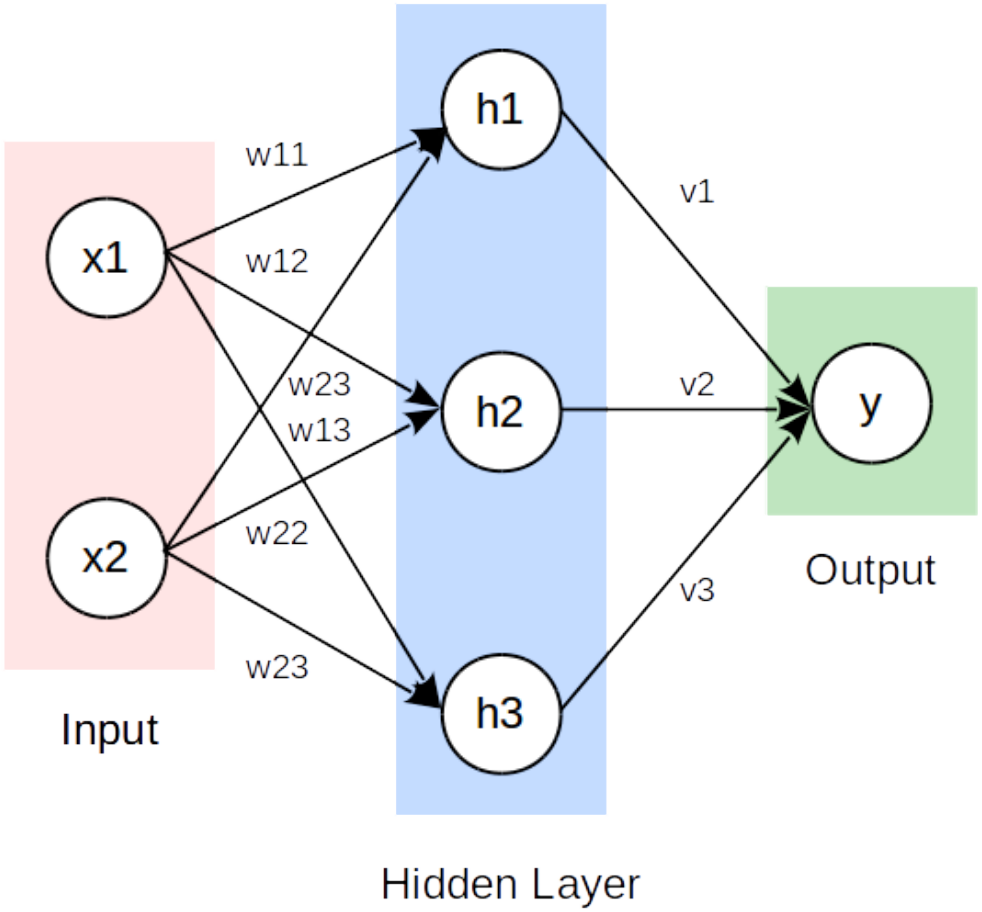
then $y$ would be a random variable from the *linear exponential family* and a subset of *generalized linear models*.

### 1.2.2 2-Layer Neural Network

The graph above can be composed in multiple layers to create hierarchical networks, i.e. networks with more than one layer. Intermediate layers are hidden in the sense that they are never observed. This poses a statistical problem because it means that we will never be able to identify the true parameters of the network (because they can be interchanged among nodes).

*Note:* Identifiability is important in statistics since it ensures with a large enough sample we'll be able to identify the true values of our model. That is we'll be able to know with certainty things like the mean and standard deviation. Since neural networks have *non-identifiable* parameters we'll never be able to learn their true values. In practice this doesn't pose impede the inferential process since we usually seek approximates for the true value. In particular we trade the ability of our models to generalize well for exacctness. This is usually sufficient since finding the optimal parameter set doesn't yield significant gains over local optima.

Now, let's compose our first neural network with 1-hidden layer.




Logistic Function Graph

Assuming we have $n$ training samples with features $\{\mathbf{x}^{(i)}\}_{i=1}^{n} = \{(x_{1i}, x_{2i})\}_{i=1}^{n}$ the network above is composed of the functions as follows:

$$a_i = w_{i1}x_1 + w_{i2}x_2 + b_i^{(1)}; i = 1, 2, 3 \tag{1}$$

$$h_i : a_i \to \frac{1}{1 + exp(-a_i)}; i = 1, 2, 3 \tag{2}$$

$$z = h_1 v_1 + h_2 v_2 + h_3 v_3 + b^{(2)} \tag{3}$$

$$y : z \to \frac{1}{1 + exp(-z)} \tag{4}$$

In the network above we've implictly assumed the hidden nodes $h_i; i = 1, 2, 3$ and and output $y$ are binary outputs with a logistic transform. Other transforms for $h_i$ and $y$ are possible and we'll discuss those in the upcoming sections.

Using $\sigma(a) = \frac{1}{1+exp(-a)}$ to represent a logistic unit and vector notation the notation above simplifies:

$$a_i = \mathbf{w}_i \cdot \mathbf{x} + b_i^{(1)} \qquad h_i = \sigma(a_i) \qquad i = 1, 2, 3 \tag{5}$$

$$z = \mathbf{h} \cdot \mathbf{v} + b^{(2)} \qquad y = \sigma(z) \tag{6}$$

**A note about notation:**

- In the equations above that we're using shorthand for both $h_i$ and $y$ and do not show the variable dependencies.
- In reality they depend on multiple variables. For example in the case of $y$ we have the following functional dependencies:

$$y = y(z(h_1(\mathbf{x}, \mathbf{w}), h_2(\mathbf{x}, \mathbf{w}), h_3(\mathbf{x}, \mathbf{w}), \mathbf{v}))$$

- It is important to understand these relations since we'll need the derivatives of the cost function $J(\mathbf{w}; \mathbf{y}, \mathbf{x})$ to calculate the gradient so that we can train the model. In deeper networks the function composition can become difficult to track.
- Analytic tricks, like recursion, are frequently used to reduce this complexity. We rarely go through the calculations by hand and outsource the symbolic computation to libraries which efficiently generate these graphs and their derivatives.
- In practice understanding the foundations for how these functions are composed and derived is essential to understanding the properties of larger networks. Therefore smaller networks heuristically augment our understandinf of more complicated networks.

**A note about nomenclature** The logistic function is frequently called the sigmoid function since it has a sigmoid (s) shape. Accordingly another name for an artificial neural network with logistic units is called a sigmoid network.

Whereas we used a sigmoid unit as an input we can generalize the idea of linear activation functions to include any functional mapping $f$ that takes in a linear activation. These basic building block units are called *perceptrons*. Accordingly when we build a neural network with many different layers and the transforms are not only sigmoid we call the architecture a **multilayer perceptron**.

### 1.3 Code

In what follows we are going to build 5 functions: 1. Linear acvitation unit 2. Sigmoid unit 3. 2-Layer neural network

Note: in practice we use a stabilizing transform on the sigomid function to avoid under/overflow errors. This becomes particularly important with multclass classification.

*I'm setting up the code so it is obvious (therefore there will be some repitition but hopefully drives home the point)

### 1.3.1 Logistic regression in code

```
In [1]: import matplotlib.pyplot as plt
        import numpy as np

        ############################
        #  Generating fake data    #
        #  Logistic function       #
        ############################
        np.random.seed(123)
        n = 100

        # Setting up activation and weights
        bias = 0.2

        # Randomly intializing 100 x_1,x_2 values
        # in the range (-5,5)
        x = np.random.uniform(low=(-5.0), high=5.0, size=2*n).reshape(n,2)
        x_1 = x[:,0]
        x_2 = x[:,1]

        # Setting up the true weights
        w = np.random.random(2)
        w_1 = w[0]
        w_2 = w[1]

        # Setting up the activation function
        activation = x_1*w_1 + x_2*w_2 + bias
        y = np.round(1 / (1+np.exp(-activation)))
        print('The first 10 values of y are {}'.format(y[0:10]))
```

```
The first 10 values of y are [1. 0. 1. 1. 1. 0. 0. 0. 0. 1.]
```

We are now going to build the general feed forward network for the logistic unit:

```
In [2]: ########################
        #  General functions   #
        ########################
        # Linear activation
        def a(x,w,b):
            a_out = x.dot(w) + b
            return a_out

        # Sigmoid function
        def sigmoid(z):
            s = 1/(1+np.exp(-z))
            return s
```

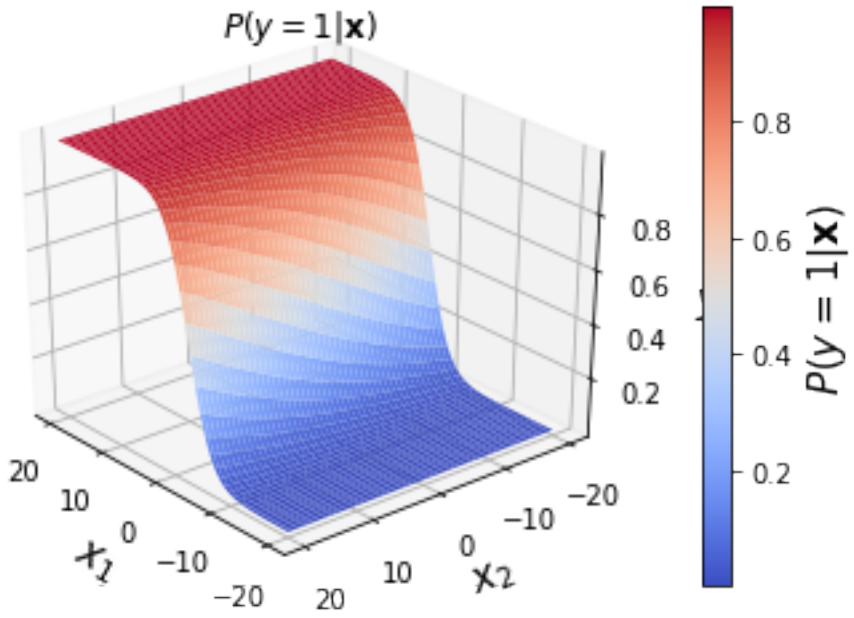
```
        # Logistic unit
        def logistic(x,w,b):
            s = sigmoid(a(x,w,b))
            y = np.round(s)
            return np.array([y,s]).T

        ff_logistic = logistic(x[0:1,:],w,bias)
        print("The first 10 entires are:\n{}".format(ff_logistic[0:10,:]))
```

```
The first 10 entires are:
[[1.         0.75460182]]
```

To gain a better understand let's now plot this in 3d dimensions to understand y

In [3]:
```
        # Plot the logistic output for 2 dimensions

        # Importing librarires
        from mpl_toolkits.mplot3d import Axes3D
        from matplotlib.colors import colorConverter, ListedColormap # some plotting functions
        from mpl_toolkits.mplot3d import Axes3D  # 3D plots
        from matplotlib import cm # Colormaps

        # Define a vector of weights for which we want to plot the output
        nb_of_ys = 200
        coords = np.linspace(-20, 20, num=nb_of_ys) # input
        x1_grid, x2_grid = np.meshgrid(coords, coords) # generate grid
        y_surface = np.zeros((nb_of_ys, nb_of_ys)) # initialize output

        # Fill the output matrix for each combination of input z's
        for i in range(nb_of_ys):
            for j in range(nb_of_ys):
                z = x1_grid[i,j]*w_1 + x2_grid[i,j]*w_2 + bias
                y_surface[i,j] = 1/(1+np.exp(-z))

        # Plot the cost function surfaces for both classes
        fig = plt.figure()

        # Plot the cost function surface for t=1
        ax = fig.gca(projection='3d')
        surf = ax.plot_surface(x1_grid, x2_grid, y_surface, linewidth=0, cmap=cm.coolwarm)
        ax.view_init(elev=30, azim=140)
        cbar = fig.colorbar(surf)
        ax.set_xlabel('$x_1$', fontsize=15)
        ax.set_ylabel('$x_2$', fontsize=15)
        ax.set_zlabel('$y$', fontsize=15)
        ax.set_title ('$P(y=1|\mathbf{x})$')
        cbar.ax.set_ylabel('$P(y=1|\mathbf{x})$', fontsize=15)
```

```
plt.grid()
plt.show()
```




### 1.3.2 Forward propogation and feed forward networks:

The process of inputing the values **x** in the model and having the values propagate forward, in the architecture, to obtain the output $y$ is called *forward propagation*. A similar method called backwards propagation starts from $y$ to get update the weight values. The process is trivial in the case of a logistic units however we'll next review how to do this with 2 layer neural network.

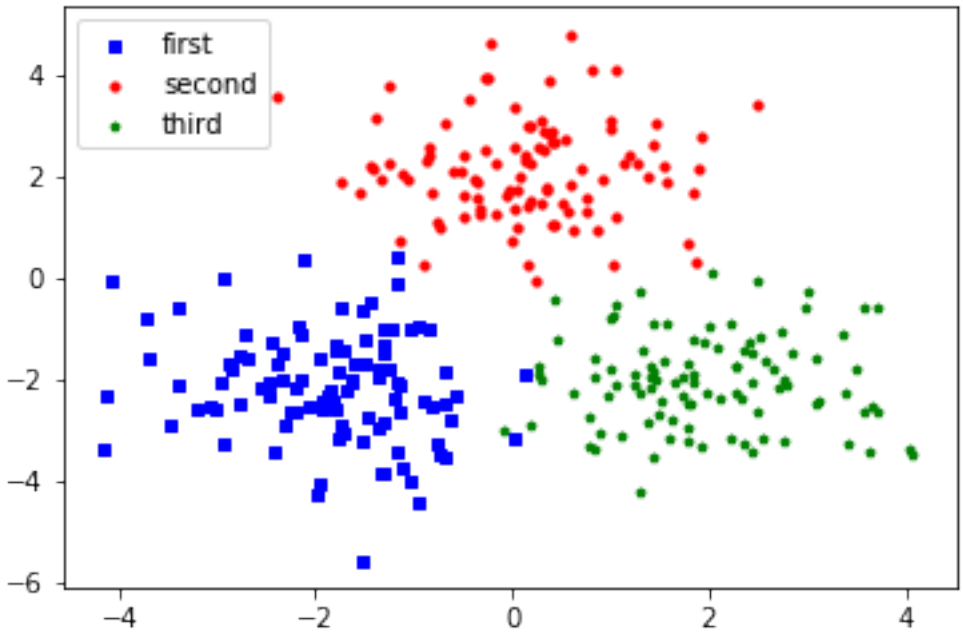
### 1.3.3 Coding a 2-layer neural network

In what follows we first present the code for generating samples from 3 Gaussian distributions centered at the point ((-2,-2),(0,2),(2,-2)). Our goal with our neural network in the training section will be to fit this model. Note that since we dealing with three classes we will only consider one pair and then extend the model to deal with 2 or more different classes. We proceed in this manner so that we can walk through how the feedforward mechanism works in a simplified neural network and then extend the logic to more complicated outputs for multiple classes.

```
In [4]: ############################
        #  Generating fake data    #
        #          for our         #
        # 1 Layer Neural Network   #
        ############################
        n = 100
        n_clusters = 3
        dim = 2
```

```
np.random.seed(1234)
# Setting up 3 clusters with centers at [-2,-2],[0,2],[2,-2]
data = np.random.randn(n * n_clusters * dim).reshape(n * n_clusters, dim)
c1_center, c2_center, c3_center = [[-2,-2],[0,2],[2,-2]]
c1, c2, c3 = data[0:n,:] + c1_center, data[n:(2*n),:] + c2_center, data[(2*n):(3*n),:] +
data = np.vstack((c1,c2,c3))
# Classes of the three clusters
classes = np.array([0]*n + [1]*n + [2]*n)
```

```
fig = plt.figure()
ax1 = fig.add_subplot(111)
ax1.scatter(c1[:,0],c1[:,1], s=10, c='b', marker="s", label='first')
ax1.scatter(c2[:,0],c2[:,1], s=10, c='r', marker="o", label='second')
ax1.scatter(c3[:,0],c3[:,1], s=10, c='g', marker="p", label='third')
plt.legend(loc='upper left');
plt.show()
```




Note that the total number of weights at each layer is equal to:

$$\text{number of weights} = \text{dim inputs} \times \text{number of outputs}$$

In our example we have 2 independent variates connected to 3 hidden units, which implies $2 \times 3 = 6$ weights.

First we'll randomly intialize all the weights:

```
In [5]: ############################
        #      Setting up our      #
```

```
# 2 Layer Neural Network  #
############################

#Setting up dimensions of 2 Layer NN
n_dims = 2 # dimension of input layer
n_hidden_layers = 3

# Settng up the weight parameters for Layer 1
w_11, w_12, w_21, w_22, w_31, w_32 = np.random.random(n_dims * n_hidden_layers)

# Setting up weight parameters for Layer 2
v_1,v_2,v_3 = np.random.random(n_hidden_layers)

# Random intializiation of the biases
# Layer 1
b_11,b_12,b_13 = np.random.random(n_hidden_layers)
b_1 = np.array([b_11,b_12,b_13])
# Layer 2
b_2 = np.random.random(1)

# Restructing for ease of implementation
w_1 = np.array([w_11,w_12])
w_2 = np.array([w_21,w_22])
w_3 = np.array([w_31,w_32])

w = np.array([w_1,w_2,w_3])
v = np.array([v_1,v_2,v_3])
```

Next we build the neural network using the previous logistic function we built. It is used to calculate the output $y$.

```
In [6]: # Defining our neural network
        def ff_nn_2(x, w, v, b_1, b_2, n_dims, n_hidden_layers):
            '''
            A simple 2 layer logistic neural network with binary output.
            '''
            # Setting up our output y
            l,_ = x.shape
            y = np.zeros((l,2))

            for i in range(l):
                x_i = x[i,:]
                # Setting up the hidden units
                h_1 = logistic(x[i,:],w[0,:],b_1[0])
                h_2 = logistic(x[i,:],w[1,:],b_1[1])
                h_3 = logistic(x[i,:],w[2,:],b_1[2])
                h = np.array([h_1,h_2,h_3])[:,1]
                # Calculating the output
```

```
            y[i,:] = logistic(h,v,b_2)

        return np.array(y)

    ff_nn_run_i = ff_nn_2(x, w, v, b_1, b_2, n_dims, n_hidden_layers)
    print('''
    We have the following
    -- Weights For layer 1: --
    W = {weights_layer_1},
    -- Weights For layer 2: --
    V = {weights_layer_2}

    -- First 3 Samples --
    X = {first_3_samples}

    -- Probabilites of y=1|x --
    {first_3_outputs}
    -- Predicted classes --
    {first_3_outputs_pred}
    '''.format(weights_layer_1=w,weights_layer_2=v,
               first_3_samples= data[0:3,:],
               first_3_outputs=ff_nn_run_i[0:3,1],first_3_outputs_pred=ff_nn_run_i[0:3,0]))
```

```
We have the following
-- Weights For layer 1: --
W = [[0.98436901 0.61562226]
 [0.15471633 0.83146947]
 [0.81569707 0.09204188]],
-- Weights For layer 2: --
V = [0.8877899  0.01643906 0.86393011]

-- First 3 Samples --
X = [[-1.52856484 -3.19097569]
 [-0.56729303 -2.3126519 ]
 [-2.72058873 -1.11283706]]

-- Probabilites of y=1|x --
[0.86159527 0.65919976 0.8759911 ]
-- Predicted classes --
[1. 1. 1.]
```

Let's run through how forward propogation works from one sample from the weights above for the first sample.

```
In [7]: # Multiplying weights by input
        print( data[0,:] * w )
```

```
[[-1.50467186 -1.96443566]
 [-0.23649394 -2.65319885]
 [-1.24684586 -0.29370339]]
```

```
In [8]: # Taking the sum and adding in the various to calculate the activation for each hidden l
        a_i =  (data[0,:] * w ).sum(axis=1) + b_1
        print(a_i)
```

```
[-2.57033683 -2.03932623 -0.88767989]
```

```
In [9]: # Passing in the activation to the hidden unit
        h = 1/(1+np.exp(-a_i))
        print(h)
```

```
[0.07107206 0.11513536 0.29158885]
```

```
In [10]: # Taking the linear sum of the activation units
         a_2 = h.dot(v) + b_2
         print(a_2)
```

```
[0.64565741]
```

```
In [11]: # Exponentiating the value to btain the predicted value 0.831
         y = 1/(1+np.exp(-a_2))
         y
```

```
Out[11]: array([0.6560312])
```

While the code above is transparent in terms of how it forward propogates information, it is not efficient and can be vectorized. Below we re-write the code to be more efficient for calculating $y$ in terms of matrices.

```
In [12]: def ff_nn_2_v2(x,w,b_1,b_2):
             a_1 = x.dot(w.T) + b_1
             h = sigmoid(a_1)
             y = logistic(h,v.T,b_2)
             return y
         # Let's check to make sure the two are the same
         # We use the function assert_allclose as a unit test
         np.testing.assert_allclose(ff_nn_2(data, w, v, b_1, b_2, n_dims, n_hidden_layers),
                                    ff_nn_2_v2(data,w,b_1,b_2))
```

Given these two implementations let's do a quick to check for speed differences

```
In [13]: import timeit
         v1 = timeit.Timer(lambda: ff_nn_2(x, w, v, b_1, b_2, n_dims, n_hidden_layers))
         v2 = timeit.Timer(lambda: ff_nn_2_v2(x,w,b_1,b_2))
         v1_time = v1.timeit(number=1000)
         v2_time = v2.timeit(number=1000)
         print('''
         Running each function 1000 we have that
         Version 1: {v1} seconds
         Version 2: {v2} seconds
         Version 2 is x {v_ratio} as fast as version 1
         '''.format(v1=v1_time,v2=v2_time, v_ratio=round(v1_time/v2_time)))
```

```
Running each function 1000 we have that
Version 1: 2.61031383399677 seconds
Version 2: 0.020483225001953542 seconds
Version 2 is x 127 as fast as version 1
```

## 1.4 Hidden unit types and output units

Previously we let the hidden and outputs units be both sigmoid functions. Traditionally there is a class of functions that hidden nodes assume along with problem-specific output nodes.

### 1.4.1 Hidden Units

Here, we introduce the various hidden unit forms and discuss their pros and cons, most of which depend on the way they update the gradient (which we'll discuss more in the next notebook).

- Sigmoid:
- None-linear
- $\sigma(x) = 1/(1 + e^{-x})$
- Maps real-valued number into a range between 0 and 1
- The sigmoid unit becomes saturated for large values of $|x|$, i.e. large negative numbers become 0 and large positive numbers become 1.
- Historically used since it mimics.
- __ In practice:__
  - The sigmoid non-linearity has recently fallen out of favor and it is rarely ever used.
  - Two major drawbacks:

  1. Sigmoids saturate for large values of $|x|$ which and "kill" gradients.
  - When saturated no information is propageted across nodes because the gradient is effectively 0
  - When intializing wieghts caution must be taken not to saturate the neuron
  - If the intial wieghts are too large then the network will not update

  2. Sigmoid outputs are not zero-centered

  - During training this tends to bias the dynamics of graident descent (i.e. neurons always positive or negative)
  - Unstable updates for the gradient
  - Less severe than gradient saturiation

- Hyperbolic Tangent (Tanh).
- Non-linear
- tanh neuron is a scaled sigmoid neuron, in particular the following holds: $tanh(x) = 2\sigma(2x)^{-1}$
- Maps real-valued numbers to the range [-1, 1]
- Its activations can saturate
- Its output is zero-centered
- **In practice**:
  - Tanh non-linearity is always preferred to the sigmoid nonlinearity
- Rectified Linear Unit (ReLU):
- Linear unit
- $f(x) = \max(0, x)$
- Note Krizhevsky et al. (pdf) paper indicates a 6x improvement in convergence with the ReLU unit compared to the tanh unit.
- **In practice:**
  - Popular in the last few years.
  - Pros:
  - Linear so non-saturating
  - (+) Greatly accelerates the convergence of stochastic gradient descent compared to sigmoid/tanh functions.
  - (+) Inexpensive operations to calculate the threshold matrix for gradients when compared to tanh/sigmoid neurons
    - ReLu ReLU can be implemented by thresholding a matrix of activations at zero.
    - Sigmoid/TanH involve expensive operations such as exponentiation
  - Cons:
  - (-) ReLU units can be fragile during training and "die"
  - (-) For large gradients, weight updates may cause ReLU neurons to never activate
  - Subsequent gradient updates will be 0 and the neuron will effectively be dead
  - This can occur in 40% of your network and is known as the 'dying ReLu' problem
  - Seting the learning rate toughtfully is therefore important
- Leaky ReLus
- Piecewise liner
- Variant of ReLu to fix the "dying ReLU" problem
- Includes a parameter $\alpha$ that tunes the slope of the unit for values less than 0
- $f(x) = 1(x < 0)(\alpha x) + 1(x >= 0)(x)$
- **In Practice**:
  - Mixed results of whether their is a benefit to using ReLu units
- Maxout.
- Generlization of ReLy unit
- Introduced by Goodfellow et al.

- $f(x) = max(w_1^T x_1 + b_1, w_2^T x + b_2)$
- ReLU and Leaky ReLU are a subset of the Maxout unit
- **In Practice**:
  - Has benefits of ReLu (linear, no saturation) and none of its drawbacks (dying neurons)
  - Doubles the number of total parameters

Other hidden unit types include: - Radial basis: - $h_i = exp(-\frac{1}{\sigma^2}||W_{:,i} - \mathbf{x}||^2)$ - Softplus: - $g(a) = \log(1 + e^a)$

**Notes:**

- It is rare to mix different neurons in the same network (although nothing saying you can't)
- "What neuron type should I use?" use **ReLU**
- Considerations:
  - Careful with your learning rates
  - Monitor the fraction of "dead" units in a network
- If dead units or setting the learning rate is difficult consider leaky ReLu or maxout units
- **Never use sigmoid**
- If you want to use a sigmoid function use **tanh**
- Results with tanh will still be worse than ReLU/Maxout.

### 1.4.2 Output units

The two main types of units that we will deal with in **supervised learning** are 1. Softmax - Used for classification - Generalization of logistic unit - Multiclass classification

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

2. Linear units: - Used for regression - Affine transformations

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

## 1.5 Building more complicated architectures

Given the extensions above we'll now build more sophisticated networks. First let's define the different types of hidden and outputs units

In [14]:
```
########################
#    Hidden Units      #
########################
sigmoid = lambda z: 1.0/(1.0 + np.exp(-z))
tanh = lambda z: np.tanh(z)

# Vectorize applies to each element of an array
ReLu = np.vectorize(lambda z: np.fmax(0,z))
leaky_ReLu = np.vectorize(lambda z,alpha: np.fmax(alpha,1) * z) # assumes alpha < 1
```

```
In [15]: # Check to see if they are doing what they are supposed to
         x_temp = np.round(np.random.rand(2*3).reshape(3,2),3)
         print(x_temp)
         print(sigmoid(x_temp))
         print(tanh(x_temp))
         print(ReLu(x_temp))
         print(leaky_ReLu(x_temp,0.01))
```

```
[[0.545 0.238]
 [0.872 0.177]
 [0.371 0.76 ]]
[[0.63297478 0.55922072]
 [0.70516169 0.54413484]
 [0.59170059 0.68135373]]
[[0.49676344 0.23360578]
 [0.70238885 0.17517446]
 [0.35486609 0.64107696]]
[[0.545 0.238]
 [0.872 0.177]
 [0.371 0.76 ]]
[[0.545 0.238]
 [0.872 0.177]
 [0.371 0.76 ]]
```

```
In [16]: ########################
         #    Output Units      #
         ########################
         softmax = lambda x: np.exp(x)/(np.exp(x).sum(axis=1, keepdims=True))
         linear = lambda : x
```

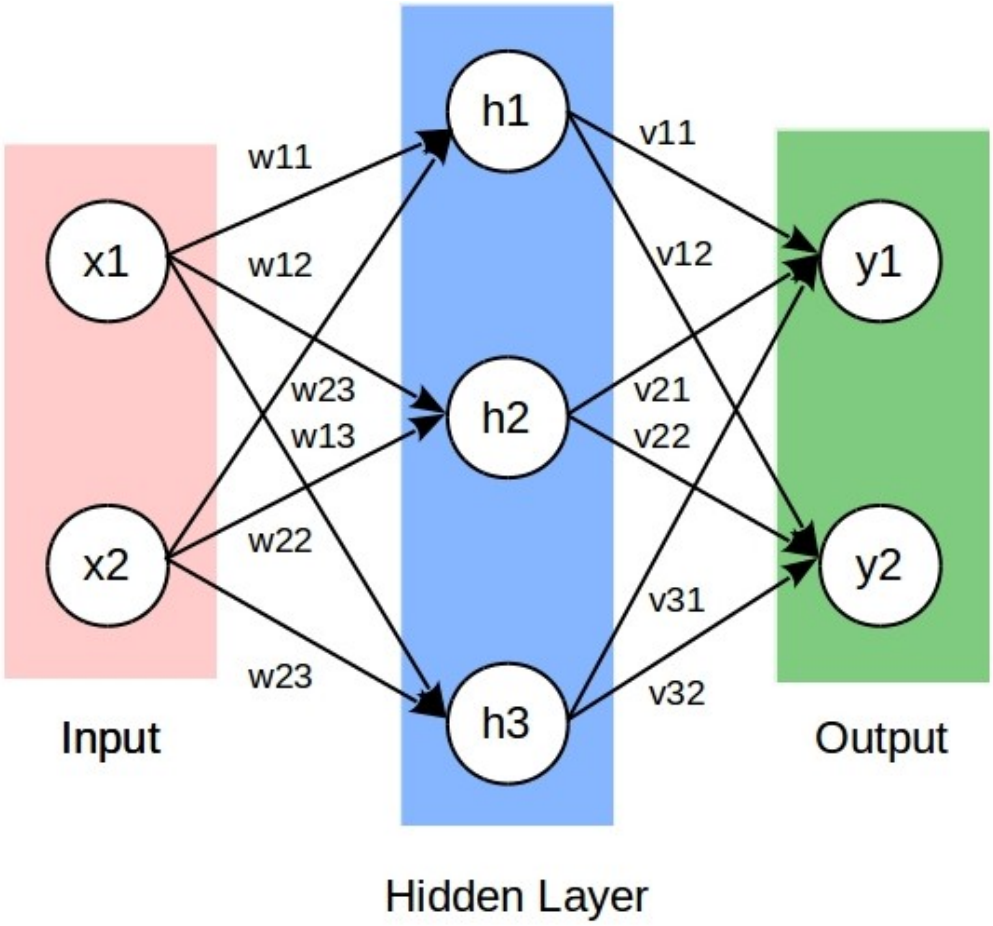
Now we will expand our 2 layer neural network to classify 3 classes rather than 2 using the softmax function.

A couple of things to notice: - The output for each sample will be a $3 \times 1$ probability vector $y$ where $\hat{C} = \arg\max_i(\mathbf{y})$ will be the predicted class. - The number of parameters has increased from (2 x 3) + (3 x 1) + 3 + 1 = 12 to (2 x 3) + (3 x 2) + 2 + 3 = 18 - The number of parameters for a 2 layer neural network is calculated as follows:

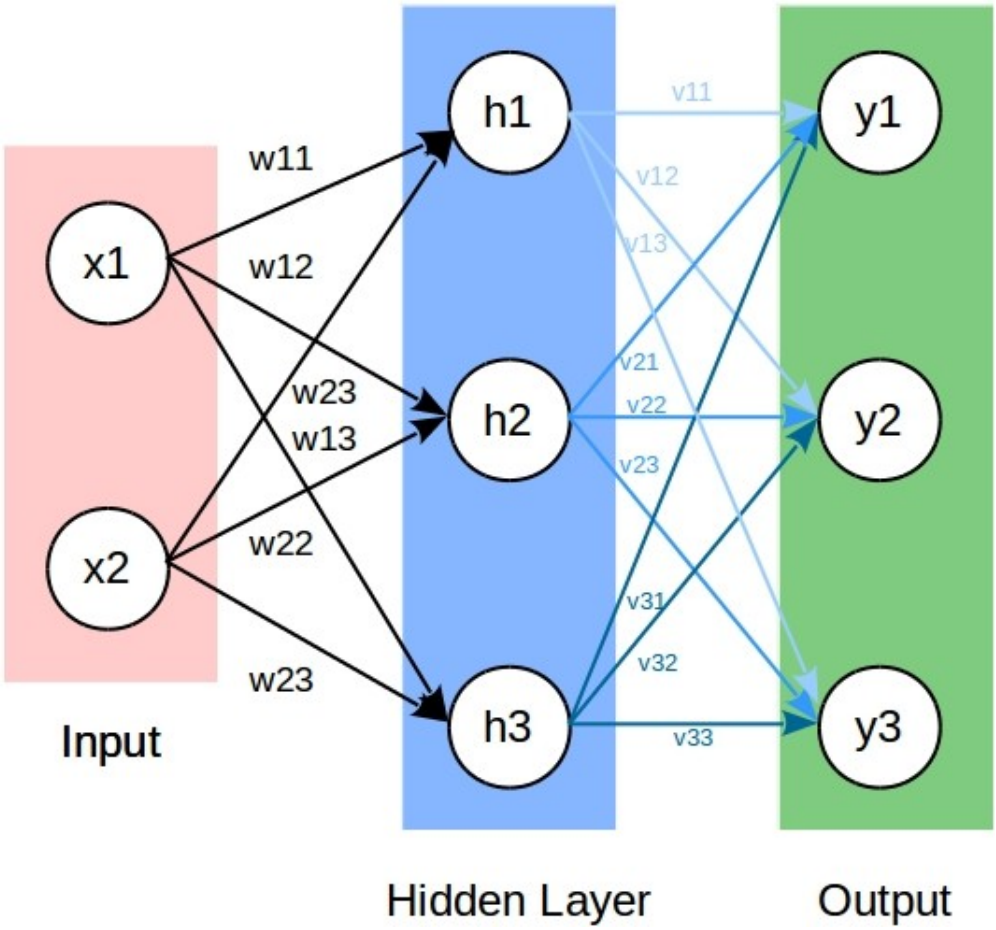
$$\begin{cases} \text{(inputs x \# hiddens layers) + (\#hidden layers x \# outputs) +} \\ \text{\# biases input-> hidden layer) + (\# biases hidden layer-> output layer)} \end{cases}$$

- There are only 2 outputted classes in the diagram rather than 3. - This is because we can obtain the value of the third output by subtracting the probabilities of the first two from 1. - In practice the gain in efficieny is usually not worth subtracting off the values and an additional set parameters. - Being explicit about the numbr of parameters we have (2x3)+(3x3) + 3+3 = 21. - The updated diagram will look like:

```
In [17]: ############################
         #      Setting up our      #
```




2 Layer Neural Network




2 Layer Neural Network

```
# 3 Layer Neural Network  #
# With softmax activation #
##########################
np.random.seed(1234)
n_dims= 2
n_hidden_layers = 3
n_outputs = 3
w = np.random.randn(n_dims * n_hidden_layers).reshape(2,3)
v = np.random.randn(n_hidden_layers * n_outputs).reshape(3,3)
b_1 = np.random.randn(n_hidden_layers).reshape(1,3)
b_2 = np.random.randn(n_hidden_layers).reshape(1,3)

def ff_nn_2_v3(x,w,b_1,b_2):
    a_1 = np.dot(x,w) + b_1
    h = sigmoid(a_1)
    a_2 = np.dot(h,v) + b_2
    y = softmax(a_2)
    return y

print('''
The first 5 class probabilities are:
{}
'''.format(ff_nn_2_v3(data,w,b_1,b_2)[0:5,:]))
```

```
The first 5 class probabilities are:
[[0.20216983 0.13695254 0.66087763]
 [0.25471053 0.11371593 0.63157355]
 [0.16181301 0.16726807 0.67091892]
 [0.21431385 0.13150355 0.6541826 ]
 [0.20059063 0.13664066 0.66276871]]
```

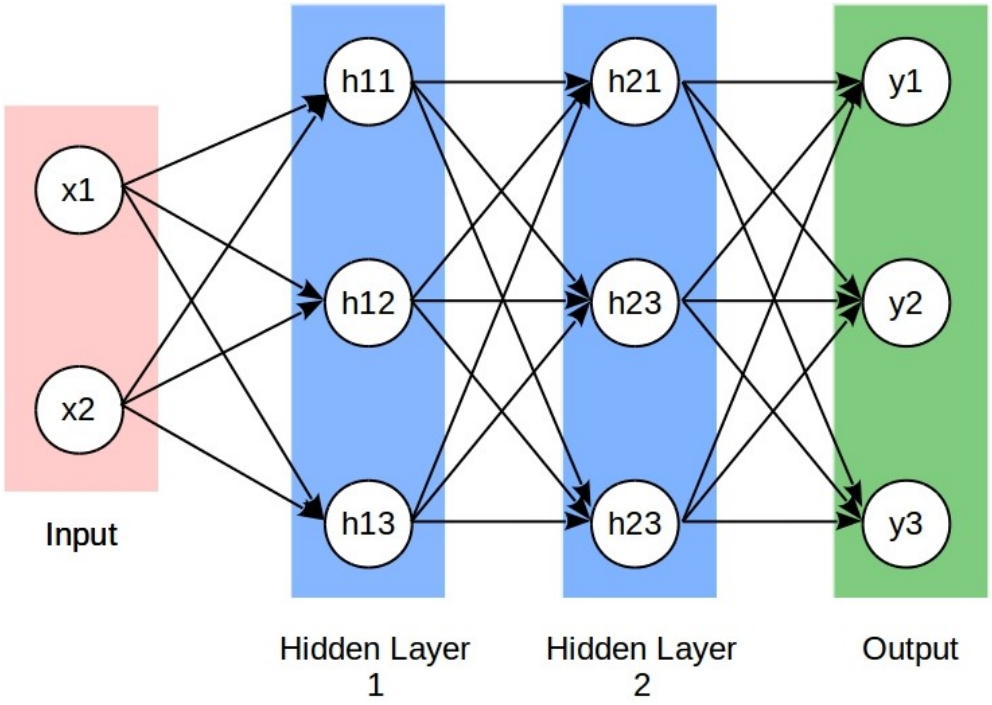
Let's now define a predict function to get the predicted class:

```
In [18]: def predict(y):
             return np.argmax(y,axis=1)
         predict(ff_nn_2_v3(data,w,b_1,b_2)[0:5,:])
```

```
Out[18]: array([2, 2, 2, 2, 2])
```

Next we'll construct a three layer neural network with sigmoid hidden layers and a softmax output layer. In this network there are (2 x 3) + (3 x 3) + (3 x 3) + 3 + 3 + 3 = 33 parameters.

```
In [19]: ############################
         #      Setting up our      #
         # 3 Layer Neural Network  #
         ############################
```




3 Layer neural Network

```
np.random.seed(1234)
np.random.seed(1234)
n_dims= 2
n_h1 = 3
n_h2 = 3
n_outputs = 3

W1 = np.random.randn(n_dims * n_h1).reshape(2,3)
W2 = np.random.randn(n_h1 * n_h2).reshape(3,3)
W3 = np.random.randn(n_h2 * n_outputs).reshape(3,3)
b1 = np.random.randn(n_h1).reshape(1,3)
b2 = np.random.randn(n_h2).reshape(1,3)
b3 = np.random.randn(n_outputs).reshape(1,3)

def ff_nn_3(x,W1,W2,W3,b1,b2,b3):
    a_1 = np.dot(x,W1) + b1
    h1 = sigmoid(a_1)
    a_2 = np.dot(h1,W2) + b2
    h2 = sigmoid(a_2)
    a_3 = np.dot(h2,W3) + b3
    y = softmax(a_3)
    return h1,h2,y

h1,h2,nn3 = ff_nn_3(data,W1,W2,W3,b1,b2,b3)
nn3_pred = predict(nn3)
```

```
In [20]: print('''
         ======================
         -- Sigmoid network --
         ======================
         First 5 class probabilities are:
         {probs}
         --------------------
         First 5 predicted classes:
         {pred}
         --------------------
         First 5 set of weights for h1:
         {w_h1}
         --------------------
         First 5 set of weights for h2:
         {w_h2}
         '''.format(probs = nn3[0:5,:],
                    pred = nn3_pred[0:5],
                    w_h1 = h1[0:5,:],
                    w_h2 = h2[0:5,]))
```

```
======================
-- Sigmoid network --
======================
First 5 class probabilities are:
[[0.14564756 0.118766   0.73558643]
 [0.13115102 0.13289294 0.73595605]
 [0.15343812 0.1130349  0.73352698]
 [0.14177793 0.12236817 0.7358539 ]
 [0.1470072  0.1174373  0.7355555 ]]
--------------------
First 5 predicted classes:
[2 2 2 2 2]
--------------------
First 5 set of weights for h1:
[[0.83134602 0.97468111 0.01280102]
 [0.85492522 0.86678168 0.10075084]
 [0.59471722 0.97268828 0.0146361 ]
 [0.83269313 0.94206466 0.0356623 ]
 [0.84672789 0.99297467 0.00264793]]
--------------------
First 5 set of weights for h2:
[[0.03639988 0.59453961 0.88437948]
 [0.05068968 0.5164686  0.86972511]
 [0.03008168 0.62887252 0.88373331]
 [0.03992494 0.57399953 0.88022924]
 [0.03510774 0.60217442 0.88658792]]
```

We can make the network above more general by including layer specific parameters. For example if we wanted a 3 layer ReLu network then we can setup our function as follows:

```
In [21]: ############################
         #   Setting up General     #
         # 3 Layer Neural Network   #
         ############################

         def ff_nn_3_v2(x,W1,W2,W3,b1,b2,b3, hidden_layer1, hidden_layer2):
             a_1 = np.dot(x,W1) + b1
             h1 = hidden_layer1(a_1)
             a_2 = np.dot(h1,W2) + b2
             h2 = hidden_layer2(a_2)
             a_3 = np.dot(h2,W3) + b3
             y = softmax(a_3)
             return y

         nn3_v2 = ff_nn_3_v2(data,W1,W2,W3,b1,b2,b3, hidden_layer1 = ReLu, hidden_layer2 = ReLu)
         nn3_pred_v2 = predict(nn3_v2)
         print('''
         ==================
         -- ReLu network --
         ==================
         First 5 class probabilities are:
         {probs}
         ---------------------
         First 5 predicted classes:
         {pred}
         '''.format(probs = nn3_v2[0:5,:],
                    pred = nn3_pred_v2[0:5]))
```

```
==================
-- ReLu network --
==================
First 5 class probabilities are:
[[7.13324047e-02 1.48524336e-03 9.27182352e-01]
 [2.23670047e-02 5.09100401e-02 9.26722955e-01]
 [1.96202089e-01 5.32725276e-04 8.03265186e-01]
 [4.48304629e-02 7.88769059e-03 9.47281846e-01]
 [1.23605886e-01 1.30132544e-04 8.76263982e-01]]
---------------------
First 5 predicted classes:
[2 2 2 2 2]
```

### 1.6 The Classification Rate

- We have not leveraged any output information in setting up the our network. That is we have not used our observed value of $y$, the class value.
- We'll go over this in more detail in the next section, however an important question is how well our feedforward network performs
- To assess performance we'll use the *classification rate* metric.
- The classification rate metric assess how many of our predicted values are correct
- For our learning algorithm the empirical cost function is:

$$\text{Classification Rate} = \frac{\sum_{i=1}^{n} \mathbf{I}(\hat{y}_i = y_i)}{n}$$

- Under the prescribed parameter weights the empirical classification rate will asymptotically converge to it's expected value $E(\mathbf{I}(\hat{Y}_i = Y_i))$.
- Given that we have not trained our network the expectation is that the performance for the randomly initialized weights will be poor
- Setting up the classification rate now will allow us to track how our parameters get updated during training in the next section

```
In [22]: ############################
         #         Setting up       #
         #   Classification Rate    #
         #   Performance Metric     #
         # 3 Layer Neural Network   #
         ############################
         def classification_rate(y_true, y_pred):
             class_rate = np.mean(y_true == y_pred)
             print('The classification rate is: {}'.format(class_rate))
             return class_rate
         classification_rate(classes, nn3_pred_v2)
```

```
The classification rate is: 0.34
```

```
Out[22]: 0.34
```

This concludes our section on architectures and building feedforward networks. In the next one we will tackle how to optimize our parameters.

## 2 Training a neural network

In the previous section we went over how to compose the architecture of a neural network. The generated architecture simply specified how nodes interact with one another. When the weights were randomly initialized the neural network performed poorly because no information about the outputs (target) had been incorporated. In this section we focus on training a neural network to find an optimal set of parameters. The process of updating our network based on target output variables is known as **training** (or parameter **inference** in the statistical community.)

## 2.1 Inference in neural networks

Previously we fitted a logistic regression using gradient descent which iteratively minimized the cross-entropy cost function $J$. The algorithm followed 4 steps below: 1. Calculate the gradient of the cost function for weights $\boldsymbol{\omega}$

$$\nabla J(\omega)$$

2. Update the weights for a given learning rate $\alpha$

$$\omega^{(\tau+1)} \leftarrow \omega - \alpha \nabla J(\omega)$$

3. Update the error based on the new parameter inputs
4. Stop updating if the algorithm has converged for some $\Delta$ or a given specified criteria else repeat steps 1-3 We'll use the same approach to train neural networks.

Interestingly, gradient descent and its' variants remain the predominant form of training neural networks in the community despite having been introduced in the 1970s. Their popularity in part is due to how parameters can be simultaneously updated and in parallel.

Other inferential techniques use gradient descent and add to the process using properties of the underlying function. For example in Nesterov Momentum the previous update is used to update $\alpha$. We'll see some of these methods in practice once we have a good feel for how gradient descent works in the context of training a neural network.

## 2.2 Training a 2-layer network

Here we use our two layer feed forward sigmoid network defined in P5. Recall the hidden units are logistic functions whereas the output node is a softmax transformation (we'll assume two output classes). The network topology is defined as follows:

$$a_j = w_{j1}x_1 + w_{j2}x_2 + w_{j0}; \quad h_j = \frac{1}{1 + exp(-a_j)}; \qquad j = 1, 2, 3 \tag{7}$$

$$z_k = v_{k1}h_1 + v_{k2}h_2 + v_{k3}h_3 + x_{k0}; \quad y_k = \frac{exp(z_k)}{\sum_{k=1}^{2} exp(z_k)}; \quad k = 1, 2 \tag{8}$$

where $y$ is the estimated output value of $T$.

### 2.2.1 Note about notation

The notation surrounding output variables varies dramatically by field and use case. For example in statistics $\hat{y}$ is the estimated value of the target variable $y$. In computer scientist however $y$ is used to represent the output of the neural network and $t$ is it's target or true value. We'll use the two conventions interchangeably and the context will define which is being used.

### 2.2.2 The cost function for a softmax unit

- There are several ways to define the cost function of output units. The way in which we select the cost function should be motivated by the goal of the problem and differentiable properties of the gradient.
- In classification a heuristic approach is to set $J(\boldsymbol{\omega})$ as the classification rate. The draw back however is that it does not consider the implicit stochasticity of the system.

- A more principled approach assumes $T$ follows a random process that depends on the variable inputs $\mathbf{x}$ and parameters $\boldsymbol{\omega}$. In allowing $T$ to be a random variable we leverage the maximum likelihood principle and show that minimizing the log-likelihood of $T$ with respect to $Y$ is similar to minimizing the entropy.
- Accordingly let $T_n \sim Multinomial(n, y_1, ..., y_C); n = 1, ..., N$, where $c$ is the number of classes and $N$ is the number of samples. Then, the likelihood for a realized sample $\mathbf{t}$ is:

$$L = \Pr(\mathbf{T} = \mathbf{t}|\mathbf{y}) = \Pr(T_1 = t_1, ..., T_N = t_N|\mathbf{y}) \tag{9}$$

$$\stackrel{\perp}{=} \prod_{n=1}^{N} \Pr(T_n = T_N|\mathbf{y}) = \prod_{n=1}^{N}\prod_{k=1}^{C} y_{nk}^{t_{nk}} \tag{10}$$

and entropy

$$J(\mathbf{y}) = l = \log(L) = \sum_n \sum_k C t_{nk} \log(y_{nk}). \tag{11}$$

Note that since $\mathbf{y}$ is a function of the weights $\boldsymbol{\omega}$ we re-write $J(\mathbf{y})$ as $J(\boldsymbol{\omega})$ where $\boldsymbol{\omega} = \{\mathbf{w}, \mathbf{v}\}$ here. - Since $J(\boldsymbol{\omega})$ is additive, let $J(\boldsymbol{\omega}) = \sum_n J_n(\boldsymbol{\omega}) = \sum_n J_n$ such that $J_n = \sum_{k=1}^{C} t_k \log(y_k)$.

-* In regression we use least-squared errors for our cost function however this motivated more by statistical properties than the MLE properties

### 2.2.3 Chain-rule

- Calculating the gradient of $J$ requires the chain-rule, for completeness we include it's formulation below along with an example
- Example: Suppose we have the following variables dependencies $x : t, s \to \mathbb{R}$ and that $y : x, t \to \mathbb{R}$ then for $z : y \to \mathbb{R}$ calculate $\frac{\partial z}{\partial t}$

$$\frac{\partial z}{\partial t} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial t} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial t}$$

### 2.2.4 Calculating the Gradient

- For our two layer neural network we need to optimize two sets of weights $\mathbf{w}$ and $\mathbf{v}$.
- Let's first consider the weights $\mathbf{v}$ and then $\mathbf{w}$.
- We will proceed in steps
- Let's first write out the derivative of $J(\mathbf{y})$ with respect to $v_{kj}$ in full then break it into it's constituent parts
- It's going to be a little messy because of the indexes so just bare with me and try and work through the steps:

$$\frac{\partial J_n}{\partial v_{kj}} = \frac{\partial J_n}{\partial y_{k'}}\frac{\partial y_{k'}}{\partial z_k}\frac{\partial z_k}{\partial v_{kj}}; \quad k = 1, ..., C \tag{12}$$

  - $\frac{\partial J_n}{\partial y_{k'}}$, for notation convenience we omit $n$:

$$\frac{\partial J_n}{\partial y_{k'}} = \frac{\partial \sum_{k'=1}^{C} t_{k'} \log(y_{k'})}{\partial y_{k'}} = \frac{t_{k'}}{y_{k'}} \tag{13}$$

– $\frac{\partial y_{k'}}{\partial z_k}$:

$$\frac{\partial y_{k'}}{\partial z_k} = \frac{\partial}{\partial z_k}\left(\frac{exp(z_{k'})}{\sum_{k=1}^{3} exp(z_k)}\right) = \begin{cases} \frac{exp(z_k)(1-exp(z_k))}{\left(\sum_{k=1}^{3} exp(z_k)\right)^2} & k = k' \\ \frac{-exp(z_{k'})exp(z_k))}{\left(\sum_{k=1}^{3} exp(z_k)\right)^2} & k \neq k' \end{cases} \tag{14}$$

$$= y_{k'}(\mathbf{1}(y_k = y_{k'}) - y_k) \tag{15}$$

– $\frac{\partial z_k}{\partial v_{kj}}$:

$$\frac{\partial z_k}{\partial v_{kj}} = \frac{\partial v_{k1}h_1 + v_{k2}h_2 + v_{k3}h_3 + x_{k0}}{\partial v_{kj}} = v_j$$

– Putting it all together we have:

$$\frac{\partial J_n}{\partial v_{kj}} = \frac{\partial J_n}{\partial y_{k'}}\frac{\partial y_{k'}}{\partial z_k}\frac{\partial z_k}{\partial v_{kj}} \tag{16}$$

$$= \frac{t_{k'}}{y_{k'}} y_{k'}(\mathbf{1}(y_k = y_{k'}) - y_k)h_j \tag{17}$$

– However **t** is equal to 1 for only one value and all others are zero. Accordingly the equation above reduces to:

$$\frac{\partial J_n}{\partial v_{kj}} = (t_k - y_k)h_j \tag{18}$$

where $h_0 = 1$.

– In vector notation we have:

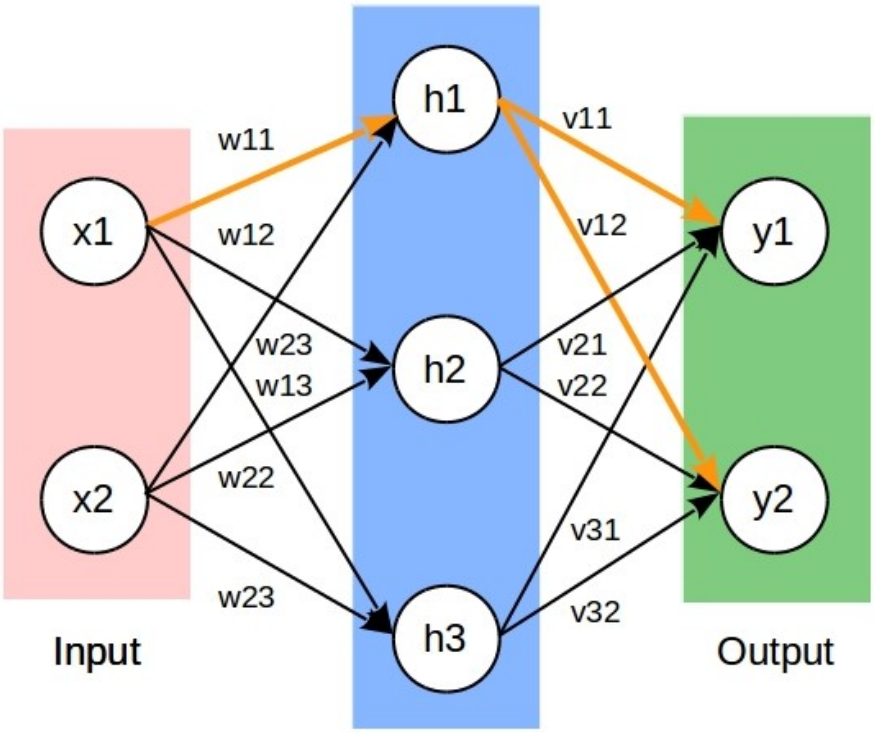
$$\frac{\partial J_n}{\partial v_k} = (t_k - y_k)\mathbf{h} = (t_k - y_k)\begin{cases} h_j & j = 1, 2, 3 \\ 1 & j = 0 \end{cases} \tag{19}$$

- Let's now calculate the derivative of $\frac{\partial J_n}{\partial w_{ji}}$ :

$$\frac{\partial J_n}{\partial w_{ji}} = \sum_{k=1}^{2} \frac{\partial J_n}{\partial y_{k'}}\frac{\partial y_{k'}}{\partial z_k}\frac{\partial z_k}{\partial h_j}\frac{\partial h_j}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}$$

– Notice:

1. We apply the multivariable chain rule when calculating the derivative of $\frac{\partial J_n}{\partial w_{ji}}$ since there is more than one path to go from $J_n$ to $w_{ji}$




- Since we now have more than one path to get from $J_n$ to $w_{ji}$ the errors are said to back propgate

2. We need to calculate that last three derivatives $\frac{\partial z_k}{\partial h_j}\frac{\partial h_j}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}$ in place of the derivative $\frac{\partial z_k}{\partial v_{kj}}$
3. Since $h_j$ and $y_k$ have the same functional form (softmax with 2 classes - i.e. logistic function) it follows that for a logistic transform $\sigma(\cdot)$ that they have the same functional form, i.e.

$$\frac{\partial h_j}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} = h_j(1-h_j)w_{ji} = \sigma(a_j)(1-\sigma(a_j))w_{ji}$$

and

$$\frac{\partial y_{k'}}{\partial z_k}\frac{\partial z_k}{\partial v_{kj}} = y_{k'}\big(\mathbf{1}(z_k = z_{k'} - y_k)h_{kj} = \sigma(z_k)\big(\mathbf{1}_{z_k=z_{k'}} - \sigma(z_k)\big)h_{kj}$$

which implies a recursive property in our neural networks.

Putting it all together we have that:

$$\frac{\partial J_n}{\partial w_{ji}} = \sum_{k=1}^{2}\frac{\partial J_n}{\partial y_{k'}}\frac{\partial y_{k'}}{\partial z_k}\frac{\partial z_k}{\partial h_j}\frac{\partial h_j}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} \tag{20}$$

$$= \sum_{k=1}^{2}(t_k - y_k)v_{kj}(h_j(1-h_j))x_i \tag{21}$$

We can vectorize our implementation as follows:

$$\frac{\partial J_n}{\partial \mathbf{w}_j} = \sum_{k=1}^{2} \frac{\partial J_n}{\partial y_{k'}} \frac{\partial y_{k'}}{\partial z_k} \frac{\partial z_k}{\partial h_j} \frac{\partial h_j}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} \tag{22}$$

$$= \sum_{k=1}^{2} (t_k - y_k) v_{kj} (h_j(1 - h_j)) \mathbf{x} \tag{23}$$

**Calculating the gradient**

- Now that we have the gradients for our weights let's setup the update rule
- Recall that we were calculating $J_n$ for only one sample, which means the value of the cost function is given by:

$$\nabla J = \left[\frac{\partial J}{\partial \mathbf{v}}, \frac{\partial J}{\partial \mathbf{w}}\right]^T = \sum_{n=1}^{N} \left[\frac{\partial J_n}{\partial \mathbf{v}}, \frac{\partial J_n}{\partial \mathbf{w}}\right]^T$$

- Accordingly, the updates to the weights occur in two steps, one for $\mathbf{v}$ and one for $\mathbf{w}$:

$$\text{Back propogation :} \tag{24}$$

$$\mathbf{v}^{(\tau+1)} = \mathbf{v}^{(\tau+1)} - \sum_{n=1}^{N} \frac{\partial J_n}{\partial \mathbf{v}^{(\tau)}} \tag{25}$$

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau+1)} - \sum_{n=1}^{N} \frac{\partial J_n}{\partial \mathbf{w}^{(\tau)}} \tag{26}$$

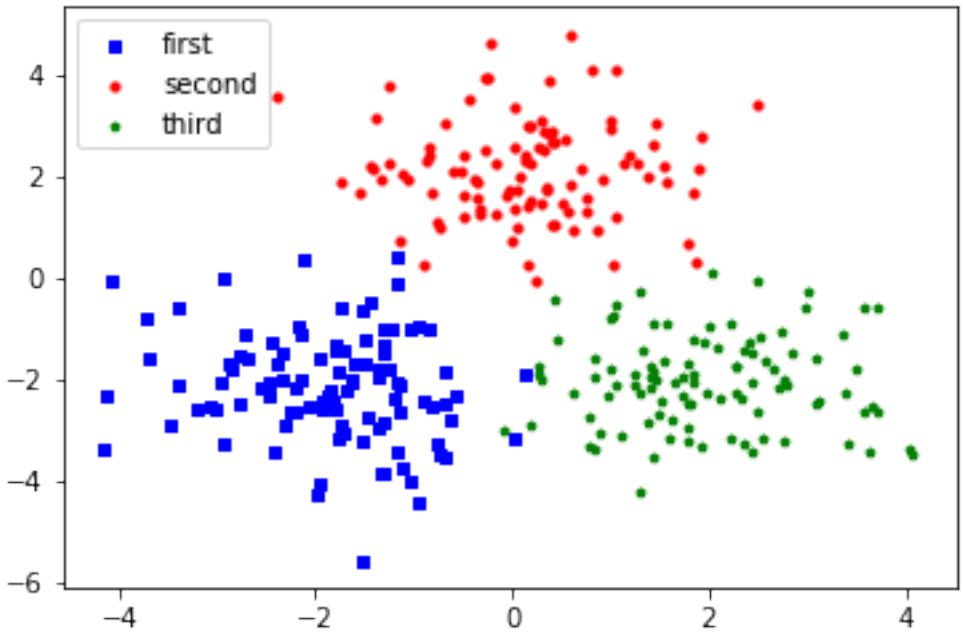
$$\text{Forward proagation :} \tag{27}$$

$$J \leftarrow J(\mathbf{v}^{(\tau+1)}, \mathbf{w}^{(\tau+1)}; \mathbf{x}) \tag{28}$$

Let's now put this together in code, we'll go through several implementations improving speed with each iteration. First let's setup our feed forward network

```
In [23]: ############################
         #  Importing Libraries   #
         ############################
         import numpy as np
         import matplotlib.pyplot as plt

         ############################
         #  Generating fake data   #
         ############################
         N = 100
         n_clusters = 3
         dim = 2
         np.random.seed(1234)

         # Setting up 3 clusters with centers at [-2,-2],[0,2],[2,-2]
         data = np.random.randn(N * n_clusters * dim).reshape(N * n_clusters, dim)
         c1_center, c2_center, c3_center = [[-2,-2],[0,2],[2,-2]]
         c1, c2, c3 = data[0:N,:] + c1_center, data[N:(2*N),:] + c2_center, data[(2*N):(3*N),:]
```

```
data = np.vstack((c1,c2,c3))

# Classes of the three clusters
classes = np.array([0]*N + [1]*N + [2]*N)
fig = plt.figure()
ax1 = fig.add_subplot(111)
ax1.scatter(c1[:,0],c1[:,1], s=10, c='b', marker="s", label='first')
ax1.scatter(c2[:,0],c2[:,1], s=10, c='r', marker="o", label='second')
ax1.scatter(c3[:,0],c3[:,1], s=10, c='g', marker="p", label='third')
plt.legend(loc='upper left');
plt.show()

# One hot encoding for T:
T = np.zeros((N * n_clusters, n_clusters))
for n in range(N * n_clusters):
    T[n, classes[n]] = 1
```




In [24]:
```
############################
#      Setting up our      #
# 2 Layer Neural Network   #
############################
np.random.seed(1)
n_dims= 2
n_h = 3
n_outputs = 3
```

```
# Intilaizing weights
W = np.random.randn(n_dims * n_h).reshape(2,3)
V = np.random.randn(n_h * n_outputs).reshape(3,3)
b1 = np.random.randn(n_h).reshape(1,3)
b2 = np.random.randn(n_outputs).reshape(1,3)

# Activation/ output units
sigmoid = lambda x: 1/(1+np.exp(-x))
softmax = lambda x: np.exp(x)/(np.exp(x).sum(axis=1, keepdims=True))

# Feed forward network
def ff_nn_2(x,W,V,b1,b2):
    a_1 = x.dot(W) + b1
    h = sigmoid(a_1)
    a_2 = h.dot(V) + b2
    exp_a_2 = np.exp(a_2)
    y = exp_a_2/ (exp_a_2.sum(axis=1, keepdims=True))
    return h,y

H, Y = ff_nn_2(data,W,V,b1,b2)
```

In [25]:
```
########################
#  Gradient descent    #
########################

def derivative_v(H, Y, T):
    '''
    Description derivative_v:
    Takes in a vector of hidden units, targeys and predicted values
    and calculates the back propogated updates

    Inputs:
    T: (Target Matrix) - NxK matrix containing N samples and K classes
    Y: (Predicted Matrix) - NxK matrix containing N samples and K predicted classes
    H: (Hidden Layer) - NxJ containing N samples and J hidden units

    Outputs:
    dv
    '''

    # Getting dimensions
    N, K = T.shape
    _,J = H.shape

    # Slow - Recursive implementation
    dv = np.zeros((J, K))
    for n in range(N):
        for j in range(J):
```

```
                    for k in range(K):
                        dv[j,k] += (T[n,k] - Y[n,k])*H[n,j]
            return dv

        print('''
        ------------------
         Derivatives of V:
        ------------------
        {dv}
        '''.format(dv=derivative_v(H,Y,T)))
```

```
------------------
 Derivatives of V:
------------------
[[-5.77182772e+01 -1.46908800e+01  7.24091572e+01]
 [ 8.48206695e+00 -8.55365889e+00  7.15919389e-02]
 [ 5.21896533e+00 -5.63958175e+01  5.11768522e+01]]
```

We can increase the speed by vectorizing the operations:

```
In [26]: def derivative_v_v2(H, T, Y):
             '''
             Description derivative_v:
             Takes in a vector of hidden units, targeys and predicted values
             and calculates the back propogated updated for

             Inputs:
             T: (Target Matrix) - NxK matrix containing N samples and K classes
             Y: (Predicted Matrix) - NxK matrix containing N samples and K predicted classes
             H: (Hidden Layer) - NxJ containing N samples and J hidden units

             Outputs:
             dv
             '''

             dv = H.T.dot(T - Y)
             return dv

         # Unit test to check if they are the same, nothing outputted if true
         assert((derivative_v(H,Y,T)-derivative_v_v2(H,Y,T)).sum()<0.0001)
```

Let's now move onto setting up the derivative for $W$:

```
In [27]: def derivative_w(data, T, Y, H, V):
             '''
             Description derivative_v:
```

```
            Takes in a vector of hidden units, targeys and predicted values
            and calculates the back propogated updated for

            Inputs:
            data: (Sample) - Nxn_dims - sample matrix
            H: (Hidden Layer) - NxJ containing N samples and J hidden units
            V: (Hidden Weight Matrix) - n_hidden_layers x n_outputs matrix
            T: (Target Matrix) - NxK matrix containing N samples and K classes
            Y: (Predicted Matrix) - NxK matrix containing N samples and K predicted classes

            Outputs:
            dw
            '''
            # Vectorized version
            dw_2 = (T - Y).dot(V.T) * H * (1 - H)
            dw_2 = data.T.dot(dw_2)

            return dw_2

        print('''
        ------------------
         Derivatives of W:
        ------------------
        {dw}
        '''.format(dw = derivative_w(data, T, Y, H, V)))
```

```
------------------
 Derivatives of W:
------------------
[[-31.39030402  18.07760568  21.62027152]
 [-37.09958634  41.53504906 -12.72079506]]
```

We've set up the derivatives for the weights however we need to consider the biases as well:

```
In [28]: # ---- Derivatives of the bias terms ----
         def derivative_b2(T, Y):
             return (T - Y).sum(axis=0)


         def derivative_b1(T, Y, H, V):
             return ((T - Y).dot(V.T) * H * (1 - H)).sum(axis=0)
```

Finally setting up the cost function we have:

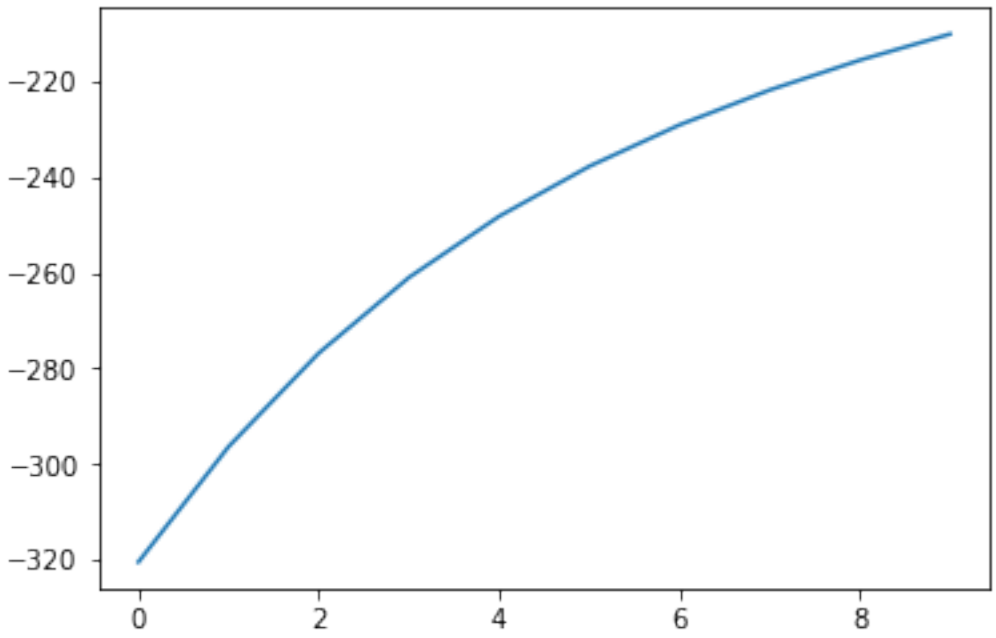
```
In [29]: def cost(T, Y):
             tot = T * np.log(Y)
             return tot.sum()
```

Putting it all together:

```
In [30]: np.random.seed(1)
         n_dims= 2
         n_h = 3
         n_outputs = 3

         # Intilaizing weights
         W = np.random.randn(n_dims , n_h)
         V = np.random.randn(n_h , n_outputs)
         b1 = np.random.randn(n_h)
         b2 = np.random.randn(n_outputs)

         # Some helper functions
         predict = lambda y: np.argmax(y,axis=1)
         classification_rate = lambda y_true, y_pred: np.mean(np.array(y_true) == np.array(y_pre
         # Setting up constants
         learning_rate = 10e-7
         costs = []

         # Running neural network
         for epoch in range(10000):
             hidden, output = ff_nn_2(data, W, V, b1,b2)
             if epoch % 1000 == 0:
                 P = predict(output)
                 c = cost(T, output)
                 r = classification_rate(classes, P)
                 print("cost:", c, "classification_rate:", r)
                 costs.append(c)

             # Gradient descent
             V  += learning_rate * derivative_v_v2(H,T, output)
             b2 += learning_rate * derivative_b2(T, output)
             W  += learning_rate *  derivative_w(data, T, output, H, V)
             b1 += learning_rate * derivative_b1(T, output, H, V)


         plt.plot(costs)
         plt.show()
```

```
cost: -320.5080195131097 classification_rate: 0.42
cost: -296.39723655779954 classification_rate: 0.4266666666666667
cost: -276.8417363259807 classification_rate: 0.43333333333333335
cost: -261.0504429295131 classification_rate: 0.44333333333333336
cost: -248.26395147589318 classification_rate: 0.4266666666666667
cost: -237.81632648428135 classification_rate: 0.39
cost: -229.1615446256186 classification_rate: 0.36666666666666664
cost: -221.87270677101793 classification_rate: 0.4533333333333333
```

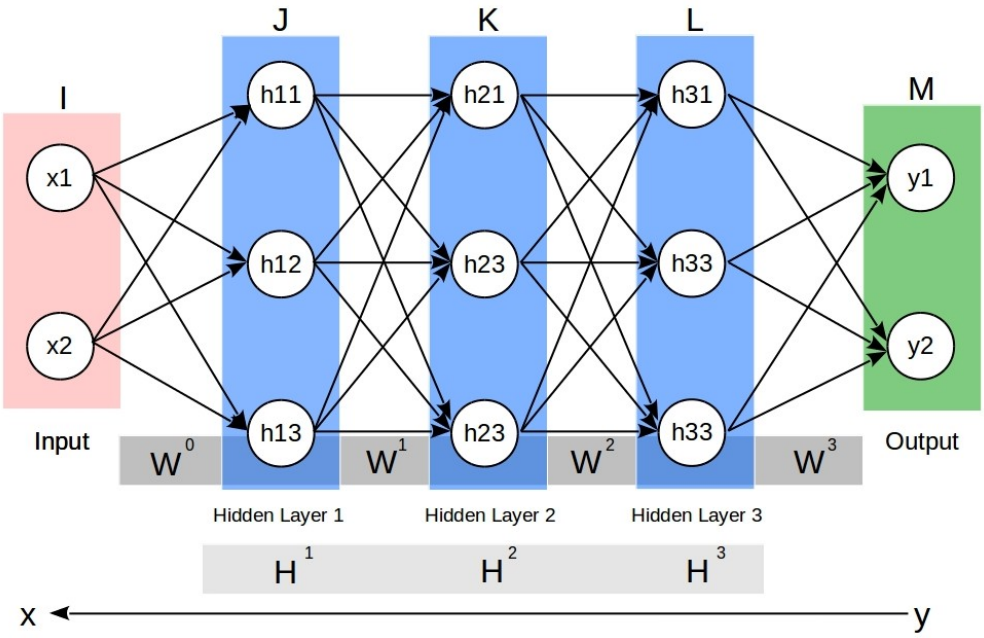
```
cost: -215.6264693679105 classification_rate: 0.54
cost: -210.182584922862 classification_rate: 0.5766666666666667
```




### 2.3 Recursiveness and derivatives

- In the previous section we saw how to construct a neural network
- When calculating the gradient we saw some of the recursive properties of the gradient when using sigmoid inputs and a softmax output
- Since each layer in the network has the same functional form we can generalize the recursive relations to deeper networks
- To generalize the recursive relations let's setup some notation for a 4-layer neural network

Using the diagram above we can define the recursive relations for parameter updates as fol-




Recursiveness of neural networks

lows: - Hidden units are logistic functions:

$$\frac{\partial J_n}{\partial w^3_{ml}} = \underbrace{(t_m - y_m)h^3_l}_{softmax} \tag{29}$$

$$\frac{\partial J_n}{\partial w^2_{lk}} = \sum_M (t_m - y_m) w^3_{lm} \underbrace{h^3_l(1-h^3_l)h^2_k}_{logistic} \tag{30}$$

$$\frac{\partial J_n}{\partial w^1_{kj}} = \sum_M \sum_L (t_m - y_m) w^3_{lm} h^3_l (1-h^3_l) w^2_{lj} \underbrace{h^2_k(1-h^2_k)h^1_j}_{logistic} \tag{31}$$

$$\frac{\partial J_n}{\partial w^0_{ji}} = \sum_M \sum_L \sum_K (t_m - y_m) w^3_{lm} h^3_l (1-h^3_l) w^2_{lj} h^2_k (1-h^2_k) w^1_{kj} \underbrace{h^1_j(1-h^1_j)x_{ji}}_{logistic} \tag{32}$$

- Hidden units are ReLu:

$$\frac{\partial J_n}{\partial w^3_{ml}} = \underbrace{(t_m - y_m)h^3_l}_{softmax} \tag{33}$$

$$\frac{\partial J_n}{\partial w^2_{lk}} = \sum_M (t_m - y_m) w^3_{lm} \underbrace{\mathbf{1}_{(a_l>0)}h^2_k}_{ReLu} \tag{34}$$

$$\frac{\partial J_n}{\partial w^1_{kj}} = \sum_M \sum_L (t_m - y_m) w^3_{lm} w^2_{lj} \mathbf{1}_{(a_l>0)} \underbrace{\mathbf{1}_{(a_k>0)}h^1_j}_{ReLu} \tag{35}$$

$$\frac{\partial J_n}{\partial w^0_{ji}} = \sum_M \sum_L \sum_K (t_m - y_m) w^3_{lm} w^2_{lj} w^1_{kj} \mathbf{1}_{(a_l>0)} \mathbf{1}_{(a_k>0)} \underbrace{\mathbf{1}_{(a_j>0)}x_{ji}}_{ReLu} \tag{36}$$