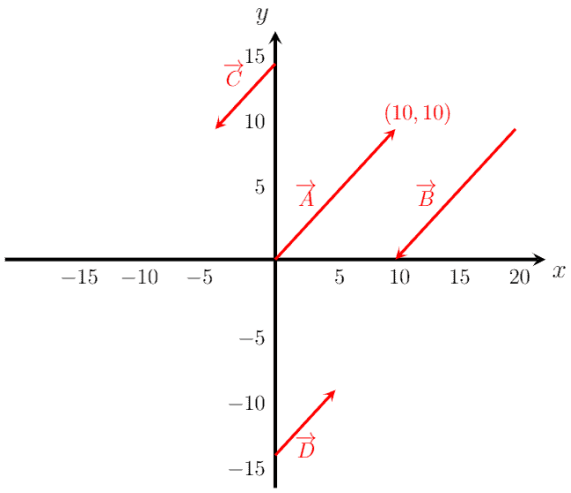
Some derivation of necessary mathematics:

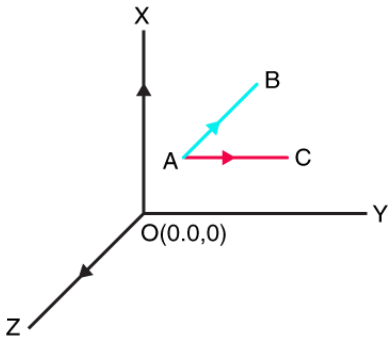
- Vectors
- Differentiation
- Partial differentiation
- Gradient of a Function
- Maxima & Minima

Vectors:

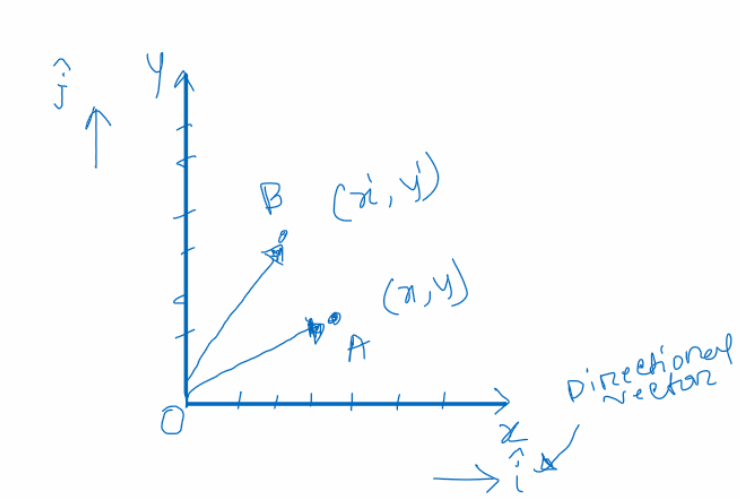
A vector is an object that has both a magnitude and a direction (i.e. 5km/m in north). Geometrically, we can picture a vector as a directed line segment, whose length is the magnitude of the vector and with an arrow indicating the direction. The direction of the vector is from its tail to its head.Two vectors are the same if they have the same magnitude and direction. This means that if we take a vector and translate it to a new position (without rotating it), then the vector we obtain at the end of this process is the same vector we had in the beginning.



Vector in 3D:



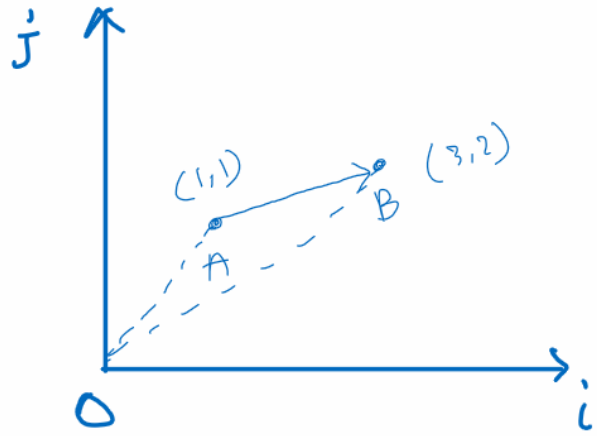
- **Now lets derive some derivation:**



  - $OA = x_{\hat{i}} + y_{\hat{j}}$
  - $OB = x'_{\hat{i}} + y'_{\hat{j}}$

We can also represent vectors like that,

- $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix}$



$\overrightarrow{AB}$ = Length of AB

- $OA = \hat{i} + \hat{j}$
- $OB = 3\hat{i} + 2\hat{j}$

$\overrightarrow{AB} = \overrightarrow{OB} - \overrightarrow{OA}$

$$= (3-1)i + (2-1)j$$

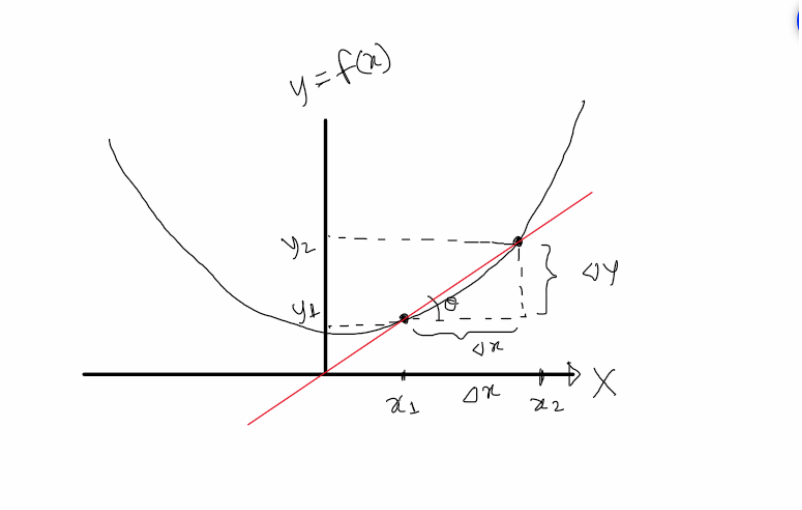
$\therefore \overrightarrow{AB} = 2i + j$

Differentiation:

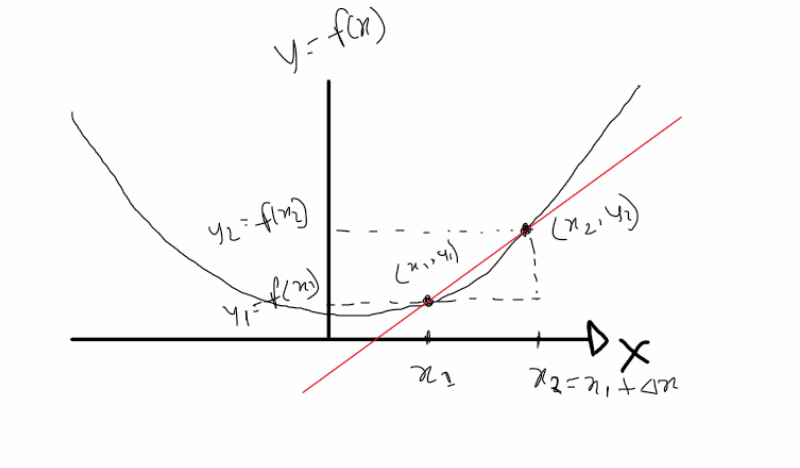
Differentiation generally refers to the rate of change of a function with respect to one of its variables. Here its similar to finding the tangent line slope of a function at some specific point.

- Connecting f,f' and f'' graphically

Lets derive some derivation:



- $slope = \frac{\Delta y}{\Delta x}$
- $slope = tan\theta = \frac{perpendicular}{base}$
- $slope = \frac{y_2 - y_1}{x_2 - x_1}$



$\lim_{x \to 0} \frac{\Delta y}{\Delta x} = \frac{dy}{dx}$

$\lim_{x \to 0} \frac{y_2 - y_1}{\Delta x}$

$\lim_{x \to 0} \frac{f(x_2) - f(x_1)}{\Delta x}$

$\therefore \frac{dy}{dx} = \lim_{x \to 0} \frac{f(x_1 + \Delta x) - f(x_1)}{\Delta x}$

This is the final differentiation equation.If you put any x here you will get the slope at your x point.

Lets see some formulas / rules of Differentiation:

Power Rule:

Here we use the power rule in order to calculate the derivative and it's pretty simple though.

$$if,\ \mathbf{f(x) = x^n}$$

$$then,\ \mathbf{f'(x) = n.x^{n-1}}$$

Examples

The considered function f(x) is equal to x to the fifth.

$$f(x) = x^5$$
$$f'(x) = 5x^{(5-1)}$$
$$f'(x) = 5x^4$$

Product Rule:

If a(x) and b(x) are two differentiable functions, then the product rule is used, where at first time it compute derivative of first function and at second time it compute derivative of second function.

$$\mathbf{f(x) = f(x).g(x)}$$

$$\mathbf{f'(x) = f'(x).g(x) + f(x).g'(x)}$$

Example

$$f(x) = (x^4 + 2).cosx$$
$$\Rightarrow f'(x) = 4x^3.cosx + (x^4 + 2).(-sinx)$$
$$\Rightarrow f'(x) = 4x^3cosx - x^4sinx - 2sinx$$

Partial Differentiation:

Now we will see the Partial Differentiation. Here, the same rules apply while obtaining derivatives with respect to one variable while keeping others constant. This term is used for Multi-Variable Functions.

Example

$$f(x, y) = x^4y$$

Obtaining partial derivative w.r.t x

$$\frac{\partial(x^4y)}{\partial x} = 4x^3y$$

Obtaining partial derivative w.r.t y

$$\frac{\partial(x^4y)}{\partial y} = x^4$$

Gradient of Function:

- Let's say there's a function of two variable x and y $\Rightarrow f(x, y)$
- then $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ is partial derivative w.r.t x and y respectively
- Now Gradient '∇' of f is defined as -

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \end{bmatrix}^T = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

- Its nothing but vector of partial derivatives

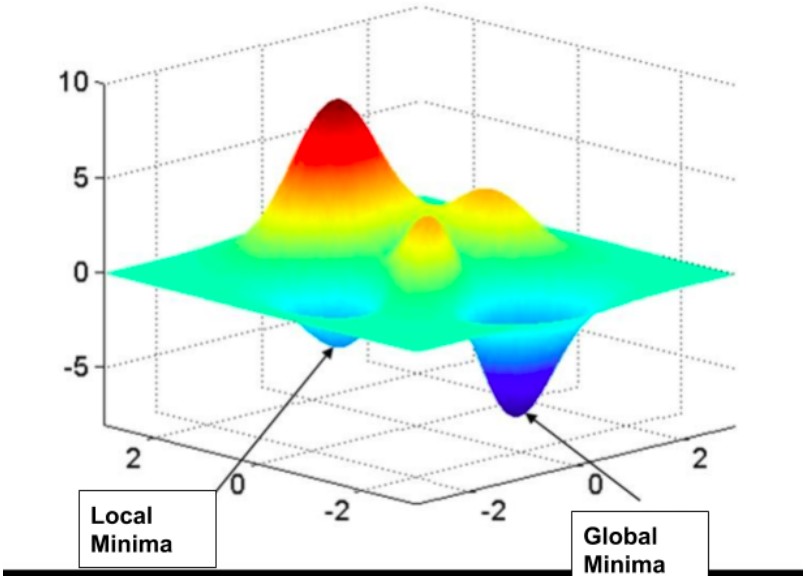
EXAMPLE

$$f(x, y) = 2.x^2 + 4y$$

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} 4x \\ 4 \end{bmatrix}$$

Local and Global Minima:

- For any function their can be many minimum points and among those minimum there can exist an minima point where function has the least value and this point is known as **global minima** and other minimum points are known as **local minima** point.
- In ML/DL finding the global minima of a loss function using optimization techniques like gradient descent plays a very important role.
-



Finding Maxima and Minima of a Function:

Calculating Maxima and Minima for a function with a single variable(univariate):

- Find points that satisfies the equation $f'(x) = 0$. These points are known as critical points. Let's say you get two points c1 and c2.
- Find double derivative of $f''(x)$ and find its value at c1 and c2
  - for c1 if -
    - f''(c1) > 0 $\Rightarrow$ its a point of minima and f(c1) is the minimum value
    - f''(c1) < 0 $\Rightarrow$ its a point of maxima and f(c1) is the maximum value

Calculating Maxima and Minima for a function with two variable(multivariate):

- Let $f(x, y)$ is a bivariate function whose local minima or maxima point needs to be calculated.
- Find -
  - $f_x = p = \frac{\partial f}{\partial x}$ and
  - $f_y = q = \frac{\partial f}{\partial y}$.
- Solve $f_x = 0$ and $f_y = 0$ and find stationary or critical points.
- Find -
  - $r = f_{xx} = \frac{\partial f^2}{\partial x^2}$,
  - $s = f_{xy} = \frac{\partial f^2}{\partial xy}$ and
  - $t = f_{yy} = \frac{\partial f^2}{\partial y^2}$
- Lets do the analysis for the critical points that we have obtained. Lets take a critical point (a,b)
  - if $r.t - s^2 > 0$ and
    - if $r > 0 \Rightarrow f(a, b)$ has local minimum at that critical point
    - if $r < 0 \Rightarrow f(a, b)$ has local maximum at that critical point
  - if $r.t - s^2 = 0 \Rightarrow$ test fails.
  - if $r.t - s^2 < 0 \Rightarrow$ its a sadal point at the critical point (i.e. neither max nor minimum)

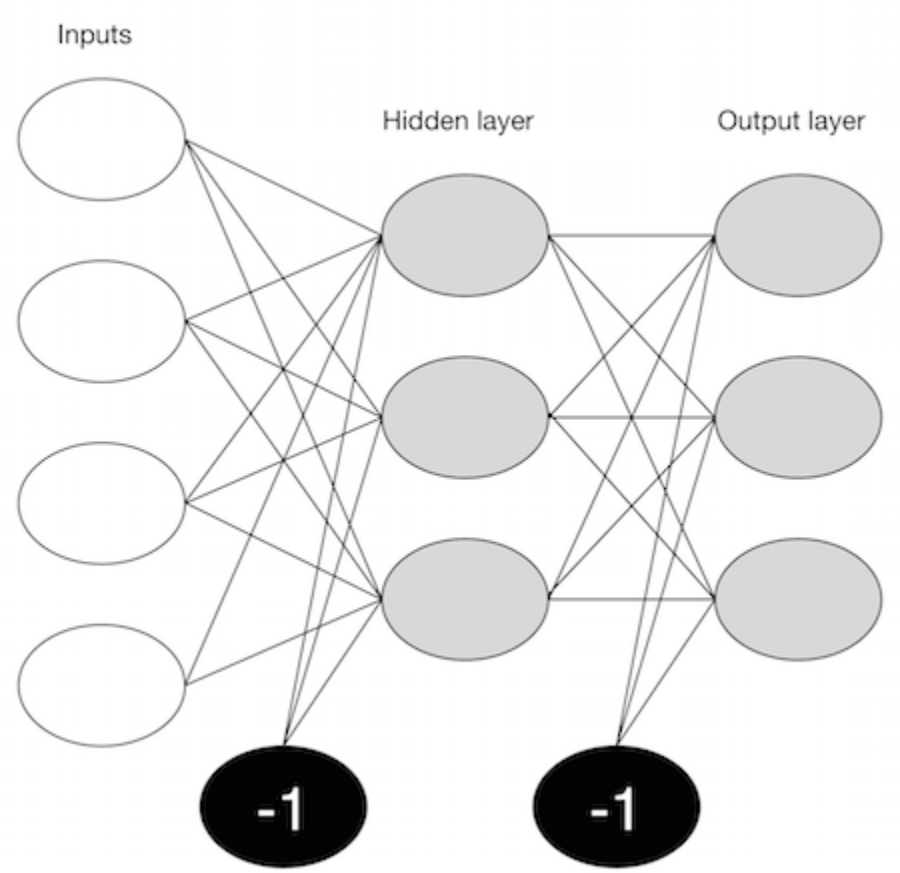
Multi-layer Perceptron

The solution to fitting more complex (*i.e.* non-linear) models with neural networks is to use a more complex network that consists of more than just a single perceptron. The take-home message from the perceptron is that all of the learning happens by adapting the synapse weights until prediction is satisfactory. Hence, a reasonable guess at how to make a perceptron more complex is to simply **add more weights**.

There are two ways to add complexity:

1. Add backward connections, so that output neurons feed back to input nodes, resulting in a **recurrent network**
2. Add neurons between the input nodes and the outputs, creating an additional ("hidden") layer to the network, resulting in a **multi-layer perceptron**

The latter approach is more common in applications of neural networks.



How to train a multilayer network is not intuitive. Propagating the inputs forward over two layers is straightforward, since the outputs from the hidden layer can be used as inputs for the output layer. However, the process for updating the weights based on the prediction error is less clear, since it is difficult to know whether to change the weights on the input layer or on the hidden layer in order to improve the prediction.

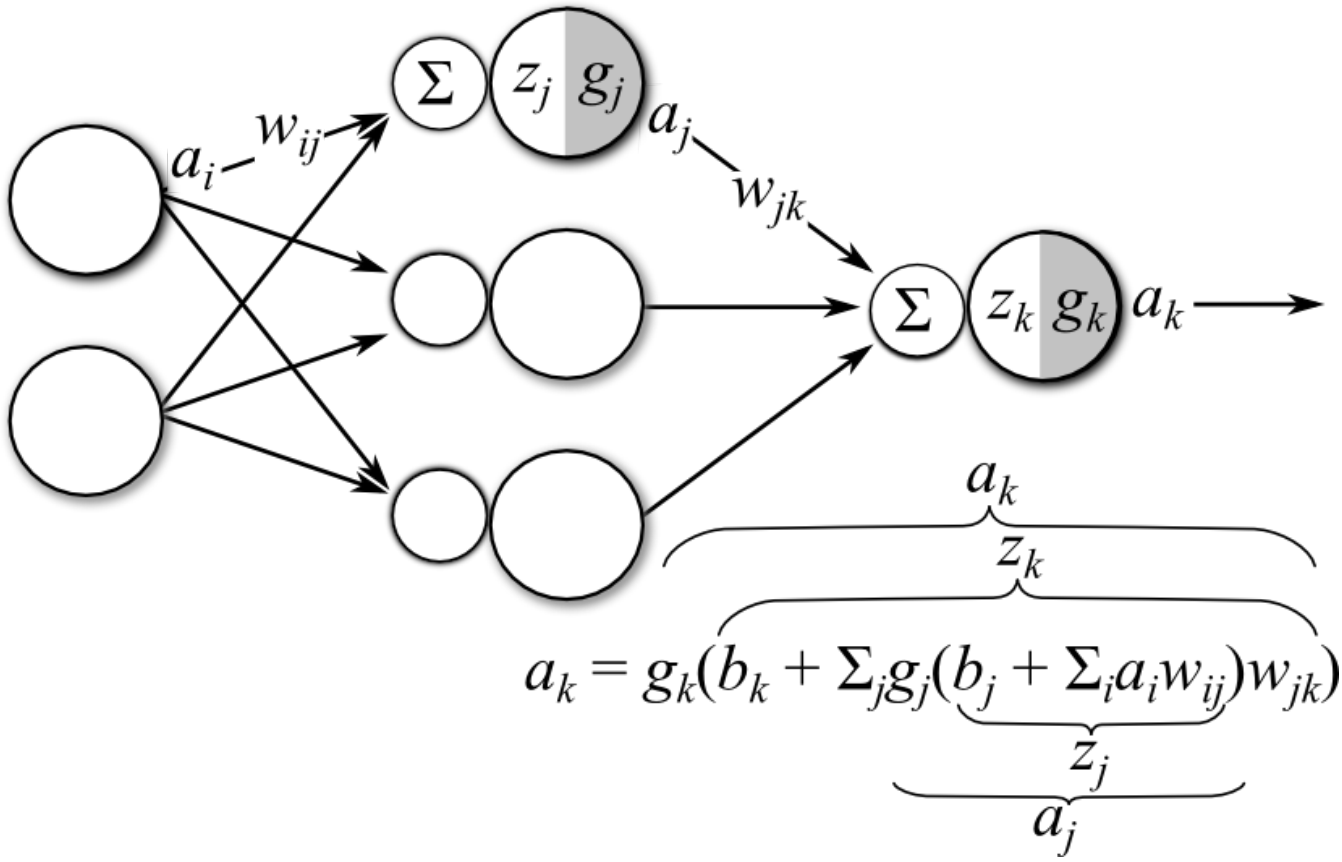
Updating a multi-layer perceptron (MLP) is a matter of:

1. moving forward through the network, calculating outputs given inputs and current weight estimates
2. moving backward updating weights according to the resulting error from forward propagation.

In this sense, it is similar to a single-layer perceptron, except it has to be done twice, once for each layer.

Backpropagation

Backpropagation is a method for efficiently computing the gradient of the cost function of a neural network with respect to its parameters. These partial derivatives can then be used to update the network's parameters using, e.g., gradient descent. This may be the most common method for training neural networks. Deriving backpropagation involves numerous clever applications of the chain rule for functions of vectors.



Review: The chain rule

The chain rule is a way to compute the derivative of a function whose variables are themselves functions of other variables. If C is a scalar-valued function of a scalar z and z is itself a scalar-valued function of another scalar variable w, then the chain rule states that

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial z}\frac{\partial z}{\partial w}$$

For scalar-valued functions of more than one variable, the chain rule essentially becomes additive. In other words, if C is a scalar-valued function of N variables $z_1, \ldots, z_N$, each of which is a function of some variable w, the chain rule states that

$$\frac{\partial C}{\partial w} = \sum_{i=1}^{N} \frac{\partial C}{\partial z_i}\frac{\partial z_i}{\partial w}$$

Notation

In the following derivation, we'll use the following notation:

L - Number of layers in the network.

N^n - Dimensionality of layer $n \in \{0, \dots, L\}$. N^0 is the dimensionality of the input; N^L is the dimensionality of the output.

$W^m \in \mathbb{R}^{N^m \times N^{m-1}}$ - Weight matrix for layer $m \in \{1, \dots, L\}$. W_{ij}^m is the weight between the i^{th} unit in layer m and the j^{th} unit in layer $m-1$.

$b^m \in \mathbb{R}^{N^m}$ - Bias vector for layer m.

σ^m - Nonlinear activation function of the units in layer m, applied element wise.

$z^m \in \mathbb{R}^{N^m}$ - Linear mix of the inputs to layer m, computed by $z^m = W^m a^{m-1} + b^m$.

$a^m \in \mathbb{R}^{N^m}$ - Activation of units in layer m, computed by $a^m = \sigma^m(h^m) = \sigma^m(W^m a^{m-1} + b^m)$. a^L is the output of the network. We define the special case a^0 as the input of the network.

$y \in \mathbb{R}^{N^L}$ - Target output of the network.

C - Cost/error function of the network, which is a function of a^L (the network output) and y (treated as a constant).

Backpropagation in general

In order to train the network using a gradient descent algorithm, we need to know the gradient of each of the parameters with respect to the cost/error function C; that is, we need to know $\frac{\partial C}{\partial W^m}$ and $\frac{\partial C}{\partial b^m}$. It will be sufficient to derive an expression for these gradients in terms of the following terms, which we can compute based on the neural network's architecture:

- $\frac{\partial C}{\partial a^L}$: The derivative of the cost function with respect to its argument, the output of the network
- $\frac{\partial a^m}{\partial z^m}$: The derivative of the nonlinearity used in layer m with respect to its argument

To compute the gradient of our cost/error function C to W_{ij}^m (a single entry in the weight matrix of the layer m), we can first note that C is a function of a^L, which is itself a function of the linear mix variables z_k^m, which are themselves functions of the weight matrices W^m and biases b^m. With this in mind, we can use the chain rule as follows:

$$\frac{\partial C}{\partial W_{ij}^m} = \sum_{k=1}^{N^m} \frac{\partial C}{\partial z_k^m} \frac{\partial z_k^m}{\partial W_{ij}^m}$$

Note that by definition

$$z_k^m = \sum_{l=1}^{N^m} W_{kl}^m a_l^{m-1} + b_k^m$$

It follows that $\frac{\partial z_k^m}{\partial W_{ij}^m}$ will evaluate to zero when $i \neq k$ because z_k^m does not interact with any elements in W^m except for those in the k^{th} row, and we are only considering the entry W_{ij}^m. When $i = k$, we have

$$\begin{aligned}\frac{\partial z_i^m}{\partial W_{ij}^m} &= \frac{\partial}{\partial W_{ij}^m}\left(\sum_{l=1}^{N^m} W_{il}^m a_l^{m-1} + b_i^m\right) \\ &= a_j^{m-1} \\ \rightarrow \frac{\partial z_k^m}{\partial W_{ij}^m} &= \begin{cases} 0 & k \neq i \\ a_j^{m-1} & k = i \end{cases}\end{aligned}$$

The fact that $\frac{\partial C}{\partial a_k^m}$ is 0 unless $k = i$ causes the summation above to collapse, giving

$$\frac{\partial C}{\partial W_{ij}^m} = \frac{\partial C}{\partial z_i^m} a_j^{m-1}$$

or in vector form

$$\frac{\partial C}{\partial W^m} = \frac{\partial C}{\partial z^m} a^{m-1\top}$$

Similarly for the bias variables b^m, we have

$$\frac{\partial C}{\partial b_i^m} = \sum_{k=1}^{N^m} \frac{\partial C}{\partial z_k^m} \frac{\partial z_k^m}{\partial b_i^m}$$

As above, it follows that $\frac{\partial z_k^m}{\partial b_i^m}$ will evaluate to zero when $i \neq k$ because z_k^m does not interact with any element in b^m except b_k^m. When $i = k$, we have

$$\begin{aligned}\frac{\partial z_i^m}{\partial b_i^m} &= \frac{\partial}{\partial b_i^m}\left(\sum_{l=1}^{N^m} W_{il}^m a_l^{m-1} + b_i^m\right) \\ &= 1 \\ \rightarrow \frac{\partial z_i^m}{\partial b_i^m} &= \begin{cases} 0 & k \neq i \\ 1 & k = i \end{cases}\end{aligned}$$

The summation also collapses to give

$$\frac{\partial C}{\partial b_i^m} = \frac{\partial C}{\partial z_i^m}$$

or in vector form

$$\frac{\partial C}{\partial b^m} = \frac{\partial C}{\partial z^m}$$

Now, we must compute $\frac{\partial C}{\partial z_k^m}$. For the final layer ($m = L$), this term is straightforward to compute using the chain rule:

$$\frac{\partial C}{\partial z_k^L} = \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_k^L}$$

or, in vector form

$$\frac{\partial C}{\partial z^L} = \frac{\partial C}{\partial a^L} \frac{\partial a^L}{\partial z^L}$$

The first term $\frac{\partial C}{\partial a^L}$ is just the derivative of the cost function with respect to its argument, whose form depends on the cost function chosen. Similarly, $\frac{\partial a^m}{\partial z^m}$ (for any layer m includling L) is the derivative of the layer's nonlinearity with respect to its argument and will depend on the choice of nonlinearity. For other layers, we again invoke the chain rule:

$$\begin{aligned}
\frac{\partial C}{\partial z_k^m} &= \frac{\partial C}{\partial a_k^m}\frac{\partial a_k^m}{\partial z_k^m} \\
&= \left(\sum_{l=1}^{N^{m+1}} \frac{\partial C}{\partial z_l^{m+1}}\frac{\partial z_l^{m+1}}{\partial a_k^m}\right)\frac{\partial a_k^m}{\partial z_k^m} \\
&= \left(\sum_{l=1}^{N^{m+1}} \frac{\partial C}{\partial z_l^{m+1}}\frac{\partial}{\partial a_k^m}\left(\sum_{h=1}^{N^m} W_{lh}^{m+1} a_h^m + b_l^{m+1}\right)\right)\frac{\partial a_k^m}{\partial z_k^m} \\
&= \left(\sum_{l=1}^{N^{m+1}} \frac{\partial C}{\partial z_l^{m+1}} W_{lk}^{m+1}\right)\frac{\partial a_k^m}{\partial z_k^m} \\
&= \left(\sum_{l=1}^{N^{m+1}} W_{kl}^{m+1\top}\frac{\partial C}{\partial z_l^{m+1}}\right)\frac{\partial a_k^m}{\partial z_k^m}
\end{aligned}$$

where the last simplification was made because by convention $\frac{\partial C}{\partial z_l^{m+1}}$ is a column vector, allowing us to write the following vector form:

$$\frac{\partial C}{\partial z^m} = \left(W^{m+1\top}\frac{\partial C}{\partial z^{m+1}}\right) \circ \frac{\partial a^m}{\partial z^m}$$

Backpropagation in practice

As discussed above, the exact form of the updates depends on both the chosen cost function and each layer's chosen nonlinearity. The following two table lists the some common choices for non-linearities and the required partial derivative for deriving the gradient for each layer:

Nonlinearity	$a^m = \sigma^m(z^m)$	$\frac{\partial a^m}{\partial z^m}$	**Notes**
Sigmoid	$\frac{1}{1+e^{z^m}}$	$\sigma^m(z^m)(1-\sigma^m(z^m)) = a^m(1-a^m)$	"Squashes" any input to the range $[0, 1]$
Tanh	$\frac{e^{z^m}-e^{-z^m}}{e^{z^m}+e^{-z^m}}$	$1-(\sigma^m(z^m))^2 = 1-(a^m)^2$	Equivalent, up to scaling, to the sigmoid function
ReLU	$\max(0, z^m)$	$0, z^m < 0;\ \ 1, z^m \geq 0$	Commonly used in neural networks with many layers

Similarly, the following table collects some common cost functions and the partial derivative needed to compute the gradient for the final layer:

Cost Function	C	$\frac{\partial C}{\partial a^L}$	**Notes**
Squared Error	$\frac{1}{2}(y-a^L)^\top(y-a^L)$	$y-a^L$	Commonly used when the output is not constrained to a specific range
Cross-Entropy	$(y-1)\log(1-a^L) - y\log(a^L)$	$\frac{a^L-y}{a^L(1-a^L)}$	Commonly used for binary classification tasks; can yield faster convergence

In practice, backpropagation proceeds in the following manner for each training sample:

1. Forward pass: Given the network input a^0, compute a^m recursively by
$$a^1 = \sigma^1(W^1 a^0 + b^1), \ldots, a^L = \sigma^L(W^L a^{L-1} + b^L)$$
2. Backward pass: Compute
$$\frac{\partial C}{\partial z^L} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L}$$
for the final layer based on the tables above, then recursively compute
$$\frac{\partial C}{\partial z^m} = \left(W^{m+1\top}\frac{\partial C}{\partial z^{m+1}}\right) \circ \frac{\partial a^m}{\partial z^m}$$
for all other layers. Plug these values into
$$\frac{\partial C}{\partial W^m} = \frac{\partial C}{\partial z_i^m} a^{m-1\top}$$
and
$$\frac{\partial C}{\partial b^m} = \frac{\partial C}{\partial z^m}$$
to obtain the updates.

Example: Sigmoid network with cross-entropy loss using gradient descent

A common network architecture is one with fully connected layers where each layer's nonlinearity is the sigmoid function $a^m = \frac{1}{1+e^{z^m}}$ and the cost function is the cross-entropy loss $(y-1)\log(1-a^L) - y\log(a^L)$. To compute the updates for gradient descent, we first compute (based on the tables above)

$$\begin{aligned}\frac{\partial C}{\partial z^L} &= \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L} \\ &= \left(\frac{a^L - y}{a^L(1-a^L)}\right) a^L(1-a^L) \\ &= a^L - y\end{aligned}$$

From here, we can compute

$$\begin{aligned}\frac{\partial C}{\partial z^{L-1}} &= \left(W^{L\top}\frac{\partial C}{\partial z^L}\right) \circ \frac{\partial a^{L-1}}{\partial z^{L-1}} \\ &= W^{L\top}(a^L - y) \circ a^{L-1}(1-a^{L-1}) \\ \frac{\partial C}{\partial z^{L-2}} &= \left(W^{L-1\top}\frac{\partial C}{\partial z^{L-1}}\right) \circ \frac{\partial a^{L-2}}{\partial z^{L-2}} \\ &= W^{L-1\top}\left(W^{L\top}(a^L - y) \circ a^{L-1}(1-a^{L-1})\right) \circ a^{L-2}(1-a^{L-2})\end{aligned}$$

and so on, until we have computed $\frac{\partial C}{\partial z^m}$ for $m \in \{1, \ldots, L\}$. This allows us to compute $\frac{\partial C}{\partial W_{ij}^m}$ and $\frac{\partial C}{\partial b_i^m}$, e.g.

$$\begin{aligned}\frac{\partial C}{\partial W^L} &= \frac{\partial C}{\partial z^L} a^{L-1\top} \\ &= (a^L - y)a^{L-1\top} \\ \frac{\partial C}{\partial W^{L-1}} &= \frac{\partial C}{\partial z^{L-1}} a^{L-2\top} \\ &= W^{L\top}(a^L - y) \circ a^{L-1}(1-a^{L-1})a^{L-2\top}\end{aligned}$$

and so on. Standard gradient descent then updates each parameter as follows:

$$W^m = W^m - \lambda\frac{\partial C}{\partial W^m}$$
$$b^m = b^m - \lambda\frac{\partial C}{\partial b^m}$$

Toy Python example

Due to the recursive nature of the backpropagation algorithm, it lends itself well to software implementations. The following code implements a multi-layer perceptron which is trained using backpropagation with user-supplied non-linearities, layer sizes, and cost function.

In [1]:

```
# Ensure python 3 forward compatibility
from __future__ import print_function
import numpy as np

def sigmoid(x):
    return 1/(1 + np.exp(-x))

class SigmoidLayer:
    def __init__(self, n_input, n_output):
        self.W = np.random.randn(n_output, n_input)
        self.b = np.random.randn(n_output, 1)
    def output(self, X):
        if X.ndim == 1:
            X = X.reshape(-1, 1)
        return sigmoid(self.W.dot(X) + self.b)

class SigmoidNetwork:

    def __init__(self, layer_sizes):
        '''
        :parameters:
            - layer_sizes : list of int
                List of layer sizes of length L+1 (including the input dimensionality)
        '''
        self.layers = []
        for n_input, n_output in zip(layer_sizes[:-1], layer_sizes[1:]):
            self.layers.append(SigmoidLayer(n_input, n_output))

    def train(self, X, y, learning_rate=0.2):
        X = np.array(X)
        y = np.array(y)
        if X.ndim == 1:
            X = X.reshape(-1, 1)
        if y.ndim == 1:
            y = y.reshape(1, -1)

        # Forward pass - compute a^n for n in {0, ... L}
        layer_outputs = [X]
        for layer in self.layers:
            layer_outputs.append(layer.output(layer_outputs[-1]))

        # Backward pass - compute \partial C/\partial z^m for m in {L, ..., 1}
        cost_partials = [layer_outputs[-1] - y]
        for layer, layer_output in zip(reversed(self.layers), reversed(layer_outputs[:-1
            cost_partials.append(layer.W.T.dot(cost_partials[-1])*layer_output*(1 - laye
        cost_partials.reverse()

        # Compute weight gradient step
        W_updates = []
        for cost_partial, layer_output in zip(cost_partials[1:], layer_outputs[:-1]):
            W_updates.append(cost_partial.dot(layer_output.T)/X.shape[1])
        # and biases
        b_updates = [cost_partial.mean(axis=1).reshape(-1, 1) for cost_partial in cost_p

        for W_update, b_update, layer in zip(W_updates, b_updates, self.layers):
            layer.W -= W_update*learning_rate
            layer.b -= b_update*learning_rate

    def output(self, X):
```

```
        a = np.array(X)
        if a.ndim == 1:
            a = a.reshape(-1, 1)
        for layer in self.layers:
            a = layer.output(a)
        return a
```

In [2]:
```
nn = SigmoidNetwork([2, 2, 1])
X = np.array([[0, 1, 0, 1],
              [0, 0, 1, 1]])
y = np.array([0, 1, 1, 0])
for n in range(int(1e3)):
    nn.train(X, y, learning_rate=1.)
print("Input\tOutput\tQuantized")
for i in [[0, 0], [1, 0], [0, 1], [1, 1]]:
    print("{}\t{:.4f}\t{}".format(i, nn.output(i)[0, 0], 1*(nn.output(i)[0] > .5)))
```

```
Input   Output  Quantized
[0, 0]  0.0148  [0]
[1, 0]  0.9825  [1]
[0, 1]  0.9825  [1]
[1, 1]  0.0280  [0]
```

In [4]:
```
import ipywidgets as widgets
from ipywidgets import *
import matplotlib.pyplot as plt
logistic = lambda h, beta: 1./(1 + np.exp(-beta * h))

@interact(beta=(-1, 25))
def logistic_plot(beta=5):
    hvals = np.linspace(-2, 2)
    plt.plot(hvals, logistic(hvals, beta))
```

```
interactive(children=(IntSlider(value=5, description='beta', max=25, min=-1), Output
()), _dom_classes=('widget…
```

This has the advantage of having a simple derivative:

$$\frac{dg}{dh} = \beta g(h)(1 - g(h))$$

Alternatively, the hyperbolic tangent function is also sigmoid:

$$g(h) = \tanh(h) = \frac{\exp(h) - \exp(-h)}{\exp(h) + \exp(-h)}$$

In [5]:
```
hyperbolic_tangent = lambda h: (np.exp(h) - np.exp(-h)) / (np.exp(h) + np.exp(-h))

@interact(theta=(-1, 25))
def tanh_plot(theta=5):
    hvals = np.linspace(-2, 2)
    h = hvals*theta
    plt.plot(hvals, hyperbolic_tangent(h))
```

```
interactive(children=(IntSlider(value=5, description='theta', max=25, min=-1), Output
()), _dom_classes=('widge…
```

Gradient Descent

The simplest algorithm for iterative minimization of differentiable functions is known as just **gradient descent**. Recall that the gradient of a function is defined as the vector of partial derivatives:

$$\nabla f(x) = [\partial f x_1, \partial f x_2, \ldots, \partial f x_n]$$

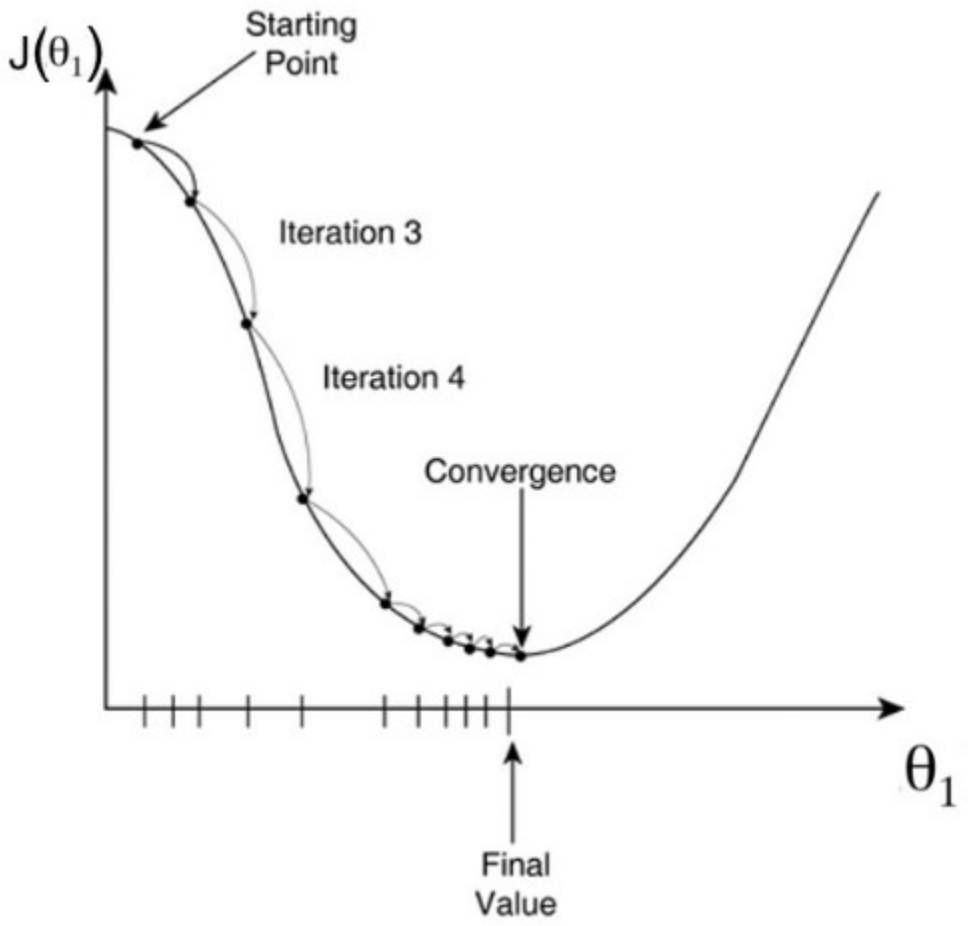
and that the gradient of a function always points towards the direction of maximal increase at that point.

Equivalently, it points *away* from the direction of maximum decrease - thus, if we start at any point, and keep moving in the direction of the negative gradient, we will eventually reach a local minimum.

This simple insight leads to the Gradient Descent algorithm. Outlined algorithmically, it looks like this:

1. Pick a point x_0 as your initial guess.
2. Compute the gradient at your current guess: $v_i = \nabla f(x_i)$
3. Move by α (your step size) in the direction of that gradient: $x_{i+1} = x_i + \alpha v_i$
4. Repeat steps 1-3 until your function is close enough to zero (until $f(x_i) < \varepsilon$ for some small tolerance ε)

Note that the step size, α, is simply a parameter of the algorithm and has to be fixed in advance.



Notice that the hyperbolic tangent function asymptotes at -1 and 1, rather than 0 and 1, which is sometimes beneficial, and its derivative is simple:

$$\frac{d\tanh(x)}{dx} = 1 - \tanh^2(x)$$

Performing gradient descent will allow us to change the weights in the direction that optimally reduces the error. The next trick will be to employ the **chain rule** to decompose how the error changes as a function of the input weights into the change in error as a function of changes in the inputs to the weights, multiplied by the changes in input values as a function of changes in the weights.

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial h}\frac{\partial h}{\partial w}$$

This will allow us to write a function describing the activations of the output weights as a function of the activations of the hidden layer nodes and the output weights, which will allow us to propagate error backwards through the network.

The second term in the chain rule simplifies to:

$$\begin{aligned}\frac{\partial h_k}{\partial w_{jk}} &= \frac{\partial \sum_l w_{lk}a_l}{\partial w_{jk}} \\ &= \sum_l \frac{\partial w_{lk}a_l}{\partial w_{jk}} \\ &= a_j\end{aligned}$$

where a_j is the activation of the jth hidden layer neuron.

For the first term in the chain rule above, we decompose it as well:

$$\frac{\partial E}{\partial h_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial h_k} = \frac{\partial E}{\partial g(h_k)}\frac{\partial g(h_k)}{\partial h_k}$$

The second term of this chain rule is just the derivative of the activation function, which we have chosen to have a convenient form, while the first term simplifies to:

$$\frac{\partial E}{\partial g(h_k)} = \frac{\partial}{\partial g(h_k)}\left[\frac{1}{2}\sum_k (t_k - y_k)^2\right] = t_k - y_k$$

Combining these, and assuming (for illustration) a logistic activation function, we have the gradient:

$$\frac{\partial E}{\partial w} = (t_k - y_k)y_k(1 - y_k)a_j$$

Which ends up getting plugged into the weight update formula that we saw in the single-layer perceptron:

$$w_{jk} \leftarrow w_{jk} - \eta(t_k - y_k)y_k(1 - y_k)a_j$$

Note that here we are *subtracting* the second term, rather than adding, since we are doing gradient descent.

We can now outline the MLP learning algorithm:

1. Initialize all w_{jk} to small random values
2. For each input vector, conduct forward propagation:
   - compute activation of each neuron j in hidden layer (here, sigmoid):

$$h_j = \sum_i x_i v_{ij}$$

$$a_j = g(h_j) = \frac{1}{1 + \exp(-\beta h_j)}$$

- when the output layer is reached, calculate outputs similarly:

$$h_k = \sum_k a_j w_{jk}$$

$$y_k = g(h_k) = \frac{1}{1 + \exp(-\beta h_k)}$$

3. Calculate loss for resulting predictions:
   - compute error at output:

$$\delta_k = (t_k - y_k) y_k (1 - y_k)$$

4. Conduct backpropagation to get partial derivatives of cost with respect to weights, and use these to update weights:
   - compute error of the hidden layers:

$$\delta_{hj} = \left[\sum_k w_{jk} \delta_k \right] a_j (1 - a_j)$$

   - update output layer weights:

$$w_{jk} \leftarrow w_{jk} - \eta \delta_k a_j$$

   - update hidden layer weights:

$$v_{ij} \leftarrow v_{ij} - \eta \delta_{hj} x_i$$

Return to (2) and iterate until learning completes. Best practice is to shuffle input vectors to avoid training in the same order.

Its important to be aware that because gradient descent is a hill-climbing (or descending) algorithm, it is liable to be caught in local minima with respect to starting values. Therefore, it is worthwhile training several networks using a range of starting values for the weights, so that you have a better chance of discovering a globally-competitive solution.

One useful performance enhancement for the MLP learning algorithm is the addition of **momentum** to the weight updates. This is just a coefficient on the previous weight update that increases the correlation between the current weight and the weight after the next update. This is particularly useful for complex models, where falling into local minima is an issue; adding momentum will give some weight to the previous direction, making the resulting weights essentially a weighted average of the two directions. Adding momentum, along with a smaller learning rate, usually results in a more stable algorithm with quicker convergence. When we use momentum, we lose this guarantee, but this is generally seen as a small price to pay for the improvement momentum usually gives.

A weight update with momentum looks like this:

$$w_{jk} \leftarrow w_{jk} - \eta \delta_k a_j + \alpha \Delta w_{jk}^{t-1}$$

where α is the momentum (regularization) parameter and Δw_{jk}^{t-1} the update from the previous iteration.

The multi-layer pereptron is implemented below in the `MLP` class. The implementation uses the scikit-learn interface, so it is uses in the same way as other supervised learning algorithms in that package.