

CSC 578

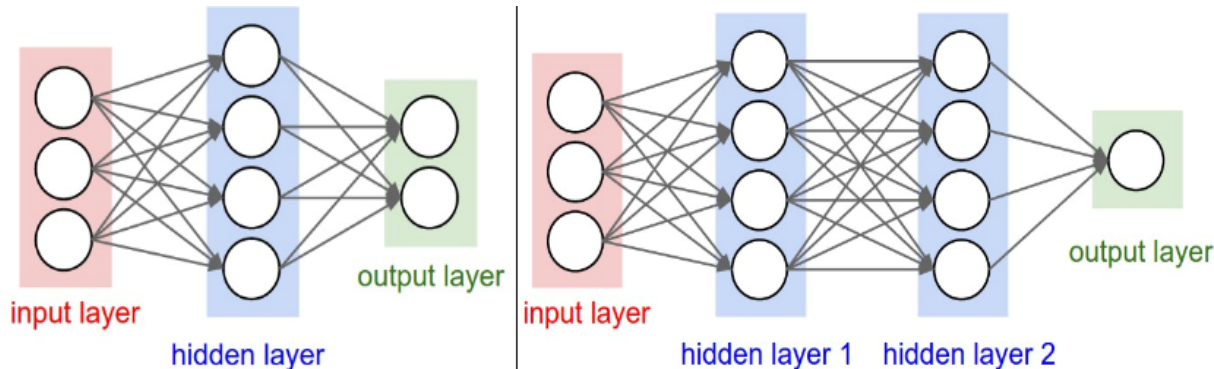
Neural Networks and Deep Learning

2. Backpropagation

(Some figures adapted from [NNDL book](#))

0. Some Terminologies

- “***N-layer*** neural network” – By naming convention, we do NOT include the input layer because it doesn’t have parameters.

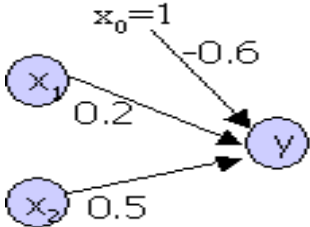
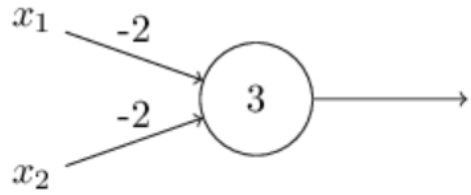


Left: A **2-layer** Neural Network (one hidden layer of 4 neurons (or units) and one output layer with 2 neurons), and three inputs.
Right: A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer. Notice that in both cases there are connections (synapses) between neurons across layers, but not within a layer.

- ***Size*** of the network – usually indicated by the number of nodes in each layer, starting from the input layer. e.g. [3,4,4,1].
- ***Hyper-parameters*** – Parameters in the network/model for which the values can be set by passing in (from outside; e.g. learning rate η), rather than parameters whose values are determined and controlled internally in the algorithm.

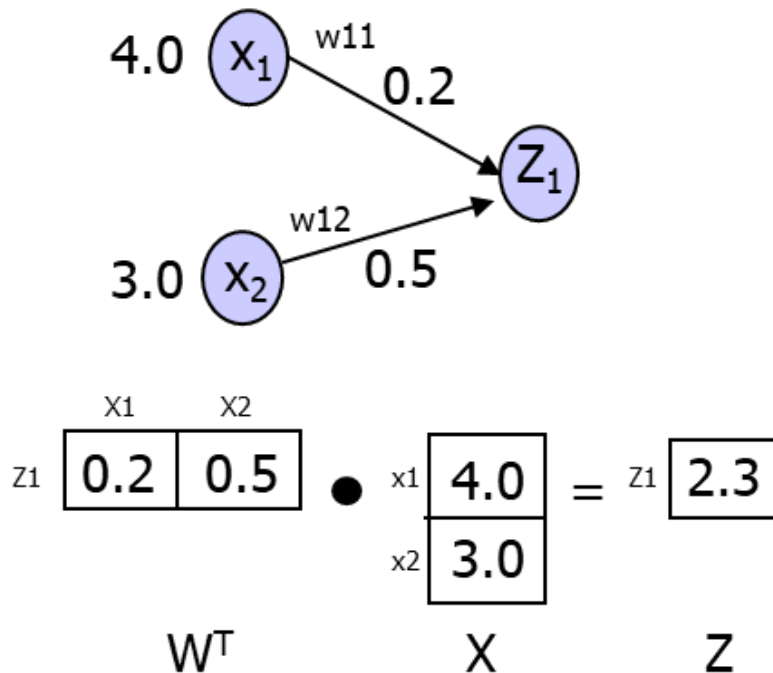
1. Notations in the NNDL book

- Differences in the notations between Mitchell's and NNDL (ch 1)

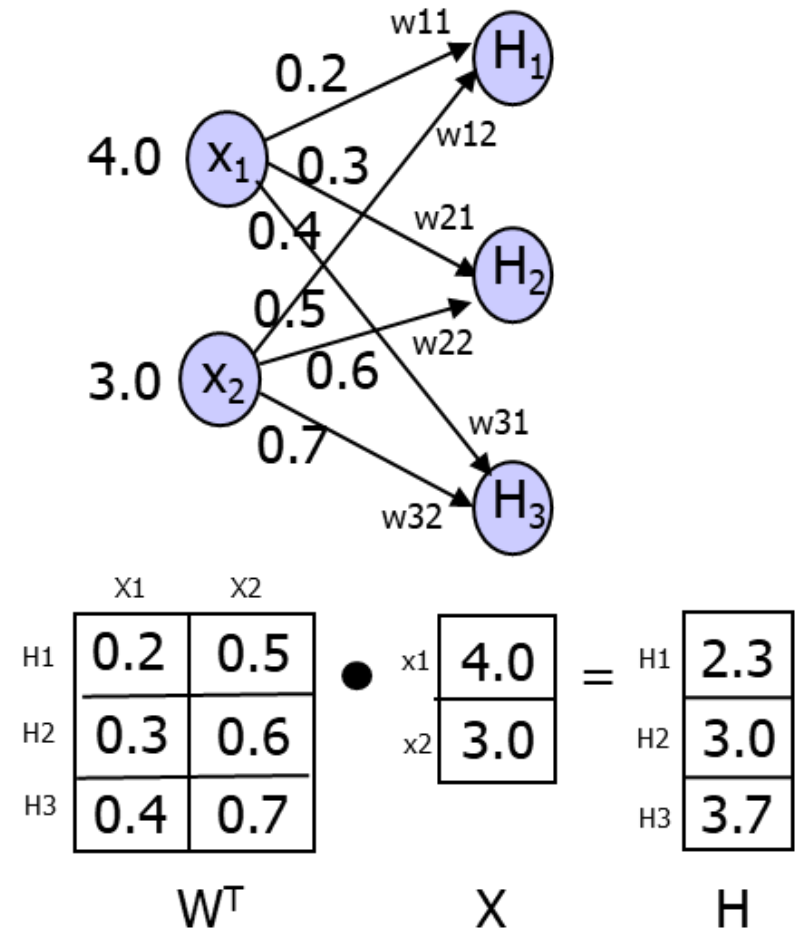
	Mitchell	NNDL
Perceptron Output (in component notation)	$o = \begin{cases} 1 & \text{if } \sum_{i=0}^n w_i x_i > 0 \\ -1 & \text{otherwise} \end{cases}$ 	$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$ 
Vector notation	$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$	$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$ <p>where b is a bias, and $b = -\text{threshold}$</p>
Sigmoid (or logistic) Function σ	$\sigma(\text{net}) = \frac{1}{1 + e^{-\text{net}}}$ <p>where $\text{net} = \sum_{i=0} w_i \cdot x_i$</p>	$\sigma(z) = \frac{1}{1 + e^{-z}}$ <p>where $\mathbf{z} = \mathbf{w} \cdot \mathbf{x} + \mathbf{b}$</p>

Vector Notation and Multilayer Networks

Single Neuron

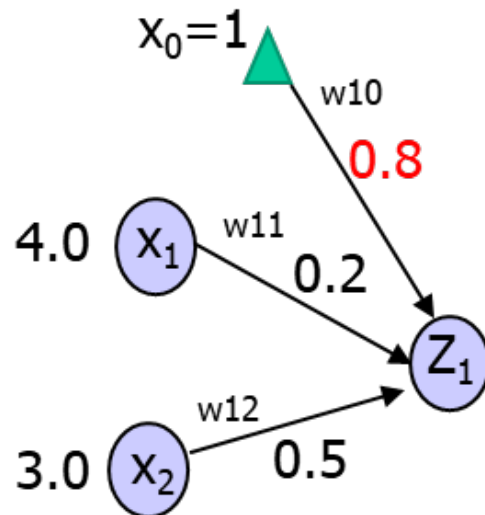


Network of Neurons

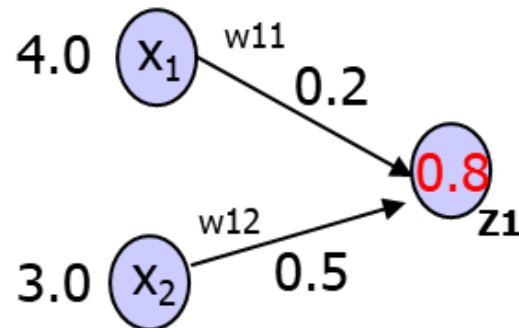


Bias – in a single neuron

Mitchell



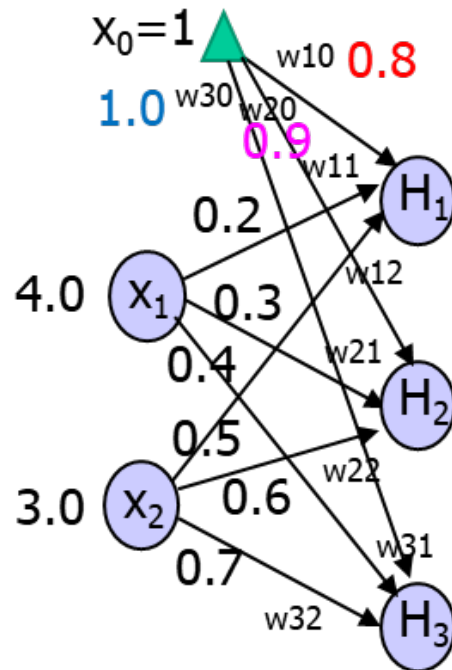
NNDL



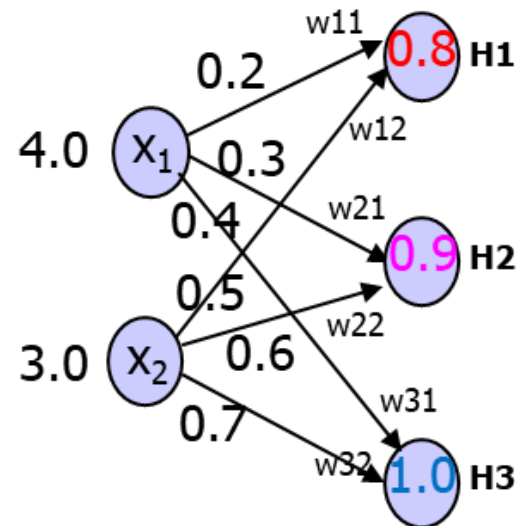
$$\begin{array}{c}
 \begin{array}{cc} x_1 & x_2 \\ \boxed{0.2} & \boxed{0.5} \end{array} \quad \bullet \quad \begin{array}{c} x_1 \\ \boxed{4.0} \\ x_2 \\ \boxed{3.0} \end{array} \quad + \quad \begin{array}{c} b_1 \\ \boxed{0.8} \end{array} = \begin{array}{c} z_1 \\ \boxed{3.1} \end{array} \\
 W^T \qquad \qquad X \qquad \qquad b \qquad \qquad Z
 \end{array}$$

Bias – in a network of neurons

Mitchell



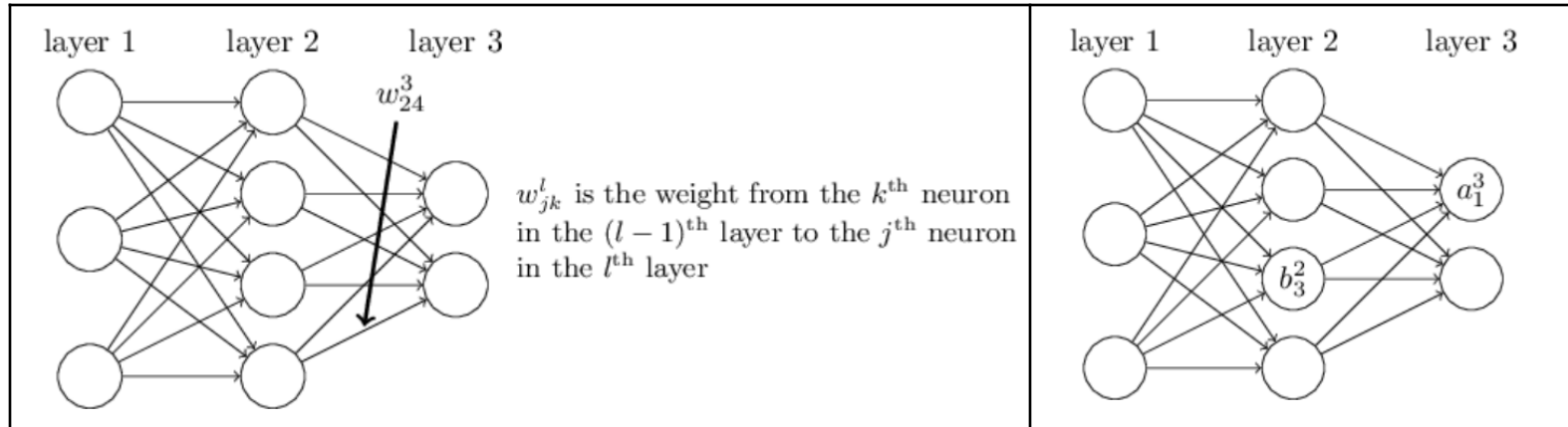
NNDL



$$\begin{array}{c}
 \begin{array}{cc}
 & x_1 & x_2 \\
 \begin{array}{c} H_1 \\ H_2 \\ H_3 \end{array} & \begin{bmatrix} 0.2 & 0.5 \\ 0.3 & 0.6 \\ 0.4 & 0.7 \end{bmatrix} & \bullet & \begin{array}{c} x_1 \\ x_2 \end{array} \begin{bmatrix} 4.0 \\ 3.0 \end{bmatrix} & + & \begin{array}{c} b_1 \\ b_2 \\ b_3 \end{array} \begin{bmatrix} 0.8 \\ 0.9 \\ 1.0 \end{bmatrix} & = & \begin{array}{c} H_1 \\ H_2 \\ H_3 \end{array} \begin{bmatrix} 3.1 \\ 3.9 \\ 4.7 \end{bmatrix} \\
 W^T & & X & & b & & H
 \end{array}
 \end{array}$$

2. BP Algorithm in NNDL book (ch 2)

- Indices and notations



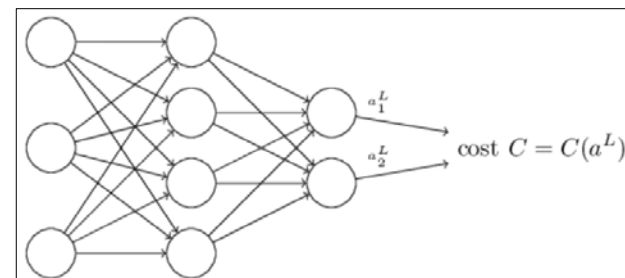
- Activation of a neuron:

- j^{th} neuron in the l^{th} layer: $a_j^l = \sigma(\sum_k w_{jk}^l \cdot a_k^{l-1} + b_j^l)$

- Vector notation: $a^l = \sigma(w^l \cdot a^{l-1} + b^l)$

- Cost function (quadratic):

- $C = \frac{1}{2} \sum_j (y_j - a_j^L)^2 = \frac{1}{2} \|y - a^L\|^2$



	Mitchell	NNDL
Objective Function (to minimize)	<p>Error (<i>Sum of Squared Error</i>)</p> $E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$ <p><u>Note</u>: No other error/cost function is used in the book.</p>	<p>Cost function (<i>Quadratic cost; MSE</i>)</p> $C(w, b) \equiv \frac{1}{2n} \sum_x \ y(x) - a\ ^2$ <p>where n is the total number of training examples. So the cost is the mean error.</p>
Gradient of Error/Cost function	$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_n} \right]$	$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \dots, \frac{\partial C}{\partial v_m} \right)^T$
Weight change	<p>Weight vector:</p> $\Delta \vec{w} = -\eta \nabla E[\vec{w}]$ <p>Individual weight:</p> $\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$	<p>Vector notation:</p> $\Delta v = -\eta \nabla C$ <p>where $v = v_1, v_2, \dots$</p>
Weight update rule	$w_i \leftarrow w_i + \Delta w_i$	$v \rightarrow v' = v - \eta \nabla C = v + \Delta v$

	Mitchell	NNDL
Weight Update: batch vs. stochastic	<p>Batch: $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$</p> <p>Stochastic/ Online: $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$</p> <p>where $E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$</p> <p>$E_d[\vec{w}] \equiv \frac{1}{2} (t_d - o_d)^2$</p>	<p>Batch: $w_k \rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$</p> <p>$b_l \rightarrow b'_l = b_l - \eta \frac{\partial C}{\partial b_l}$</p> <p>Mini-batch/ stochastic: $\nabla C \approx \frac{1}{m} \sum_{j=1}^m \nabla C_{X_j}$</p> <p>$w_k \rightarrow w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$</p> <p>$b_l \rightarrow b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l},$</p>

Note: the larger the mini-batch, the smaller the learning rate will need to be in order to train as accurately.

Algorithm Descriptions

Mitchell	NNDL (ch 2)
<p>Initialize all weights to small random numbers. Until satisfied, Do</p> <ul style="list-style-type: none"> For each training example, Do <ol style="list-style-type: none"> Input the training example to the network and compute the network outputs For each output unit k $\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$ For each hidden unit h $\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{h,k} \delta_k$ Update each network weight $w_{i,j}$ $w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$ <p>where</p> $\Delta w_{i,j} = \eta \delta_j x_{i,j}$ 	<ol style="list-style-type: none"> Input x: Set the corresponding activation a^1 for the input layer. Feedforward: For each $l = 2, 3, \dots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$. Output error δ^L: Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$. Backpropagate the error: For each $l = L - 1, L - 2, \dots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$. Output: The gradient of the cost function is given by $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$ and $\frac{\partial C}{\partial b_j^l} = \delta_j^l$.

- The Hadamard product, $s \odot t$

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 * 3 \\ 2 * 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}$$

- The **four fundamental equations**:

Given $z^l = w^l \cdot a^{l-1} + b^l$ (or $z_j^l = \sum_k w_{jk}^l \cdot a_k^{j-1} + b_j^k$),

Error:

1. **An equation for the error in the output layer, δ^L** : The components of δ^L are given by

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L). \quad (\text{BP1})$$

2. **An equation for the error δ^l in terms of the error in the next layer, δ^{l+1}** : In particular

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \quad (\text{BP2})$$

Rate of change of the cost:

3. **An equation for the rate of change of the cost with respect to any bias in the network:** In particular:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \quad (\text{BP3})$$

4. **An equation for the rate of change of the cost with respect to any weight in the network:** In particular:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l. \quad (\text{BP4})$$

4. Implementation

- NNDL book code uses Numpy heavily. In particular,
 - Data is stored in **multidimensional Numpy arrays** – called *tensors*.
 - **Vector operations** as well as scalar operations are used.

2.2.1 Scalars (0D tensors)

A tensor that contains only one number is called a *scalar* (or scalar tensor, or 0-dimensional tensor, or 0D tensor). In Numpy, a float32 or float64 number is a scalar tensor (or scalar array). You can display the number of axes of a Numpy tensor via the `ndim` attribute; a scalar tensor has 0 axes (`ndim == 0`). The number of axes of a tensor is also called its *rank*. Here's a Numpy scalar:

```
>>> import numpy as np
>>> x = np.array(12)
>>> x
array(12)
>>> x.ndim
0
```

2.2.2 Vectors (1D tensors)

An array of numbers is called a *vector*, or 1D tensor. A 1D tensor is said to have exactly one axis. Following is a Numpy vector:

```
>>> x = np.array([12, 3, 6, 14])
>>> x
array([12, 3, 6, 14])
>>> x.ndim
1
```

2.2.3 Matrices (2D tensors)

An array of vectors is a *matrix*, or 2D tensor. A matrix has two axes (often referred to *rows* and *columns*). You can visually interpret a matrix as a rectangular grid of numbers. This is a Numpy matrix:

```
>>> x = np.array([[5, 78, 2, 34, 0],
                  [6, 79, 3, 35, 1],
                  [7, 80, 4, 36, 2]])

>>> x.ndim
2
```

The entries from the first axis are called the *rows*, and the entries from the second axis are called the *columns*. In the previous example, `[5, 78, 2, 34, 0]` is the first row of `x` and `[5, 6, 7]` is the first column.

2.2.4 3D tensors and higher-dimensional tensors

If you pack such matrices in a new array, you obtain a 3D tensor, which you can visually interpret as a cube of numbers. Following is a Numpy 3D tensor:

```
>>> x = np.array([[[5, 78, 2, 34, 0],
                  [6, 79, 3, 35, 1],
                  [7, 80, 4, 36, 2]],
                 [[5, 78, 2, 34, 0],
                  [6, 79, 3, 35, 1],
                  [7, 80, 4, 36, 2]],
                 [[5, 78, 2, 34, 0],
                  [6, 79, 3, 35, 1],
                  [7, 80, 4, 36, 2]]])

>>> x.ndim
3
```

Numpy vector operation examples

```
import numpy as np

# an numpy 1d array
ar = np.array([5, 8, 4, 1, 2])
print("(0) ar=%s\n" % (ar))

# (1) scalar operation - max(), argmax()
max = np.max(ar)
minindex = np.argmax(ar)
print("(1) max=%d, argmax=%d\n" % (max, minindex))

# (2) vector operation - add 2 to all elements
ar2 = ar + 2
print("(2) ar2=%s\n" % (ar2))

# (3) array slicing
print("(3-1) ar[1:] =%s" % (ar[1:]))
print("(3-2) ar[:-1] =%s" % (ar[:-1]))
print("(3-3) ar[1:-1]=%s\n" % (ar[1:-1]))
```

(0) ar=[5 8 4 1 2]

(1) max=8, argmax=1

(2) ar2=[7 10 6 3 4]

(3-1) ar[1:] =[8 4 1 2]

(3-2) ar[:-1] =[5 8 4 1]

(3-3) ar[1:-1]=[8 4 1]


```

# (4) array shape
print ("(4-0) ar.shape={a.shape}\n".format(a=ar)) # row vector

art = ar.reshape(-1,1)
print ("(4-1) art.shape={a.shape} after reshape(-1,1)".format(a=art)) # column vector
print ('{}\n'.format(art))

art[1]=0
print ("(4-2) art.shape={a.shape} after art[1]=0".format(a=art))
print ('{}\n'.format(art))

art2 = art * 2
print ("(4-3) art2.shape={a.shape}".format(a=art2))
print ('{}\n'.format(art2))

combined = np.append(art, art2, axis=1)
print ("(4-4) combined.shape={a.shape}".format(a=combined)) # 2D array
print ('{}\n'.format(combined))

c2=combined[:,1].reshape(-1,1)
print ("(4-5) combined[:,1].shape={a.shape} after reshape(-1,1)".format(a=c2))
print ('{}\n'.format(c2))

```

(4-0) ar.shape=(5,)

(4-1) art.shape=(5, 1) after reshape(-1,1)

```

[[5]
 [0]
 [4]
 [1]
 [2]]

```

(4-2) art.shape=(5, 1) after art[1]=0

```
[[5]
 [0]
 [4]
 [1]
 [2]]
```

(4-3) art2.shape=(5, 1)

```
[[10]
 [ 0]
 [ 8]
 [ 2]
 [ 4]]
```

(4-4) combined.shape=(5, 2)

```
[[ 5 10]
 [ 0  0]
 [ 4  8]
 [ 1  2]
 [ 2  4]]
```

(4-5) combined[:,1].shape=(5, 1) after reshape(-1,1)

```
[[10]
 [ 0]
 [ 8]
 [ 2]
 [ 4]]
```

NNDL BP Code

```
class Network(object):

    def __init__(self, sizes):
        """The list ``sizes`` contains the number of neurons in the
        respective layers of the network.  For example, if the list
        was [2, 3, 1] then it would be a three-layer network, with the
        first layer containing 2 neurons, the second layer 3 neurons,
        and the third layer 1 neuron.  The biases and weights for the
        network are initialized randomly, using a Gaussian
        distribution with mean 0, and variance 1.  Note that the first
        layer is assumed to be an input layer, and by convention we
        won't set any biases for those neurons, since biases are only
        ever used in computing the outputs from later layers."""
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]

    def feedforward(self, a):
        """Return the output of the network if ``a`` is input."""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a
```

```

def SGD(self, training_data, epochs, mini_batch_size, eta,
        test_data=None):
    """Train the neural network using mini-batch stochastic
    gradient descent. The ``training_data`` is a list of tuples
    ``(x, y)`` representing the training inputs and the desired
    outputs. The other non-optional parameters are
    self-explanatory. If ``test_data`` is provided then the
    network will be evaluated against the test data after each
    epoch, and partial progress printed out. This is useful for
    tracking progress, but slows things down substantially."""
    if test_data: n_test = len(test_data)
    n = len(training_data)
    for j in xrange(epochs):
        random.shuffle(training_data)
        mini_batches = [
            training_data[k:k+mini_batch_size]
            for k in xrange(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)
        if test_data:
            print "Epoch {0}: {1} / {2}".format(
                j, self.evaluate(test_data), n_test)
        else:
            print "Epoch {0} complete".format(j)

```

```

def update_mini_batch(self, mini_batch, eta):
    """Update the network's weights and biases by applying
    gradient descent using backpropagation to a single mini batch.
    The ``mini_batch`` is a list of tuples ``(x, y)``, and ``eta``
    is the learning rate."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    self.weights = [w-(eta/len(mini_batch))*nw
                     for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb
                   for b, nb in zip(self.biases, nabla_b)]

```

```

>>> import network
>>> net = network.Network([784, 30, 10])
>>> net.SGD(training_data, 30, 10, 3.0,
            test_data=test_data)

```

```

class Network(object):
...
    def backprop(self, x, y):
        """Return a tuple "(nabla_b, nabla_w)" representing the
        gradient for the cost function C_x. "nabla_b" and
        "nabla_w" are layer-by-layer lists of numpy arrays, similar
        to "self.biases" and "self.weights"."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        # Note that the variable l in the loop below is used a little

```

```

# Note that the variable l in the loop below is used a little
# differently to the notation in Chapter 2 of the book. Here,
# l = 1 means the last layer of neurons, l = 2 is the
# second-last layer, and so on. It's a renumbering of the
# scheme in the book, used here to take advantage of the fact
# that Python can use negative indices in lists.
for l in xrange(2, self.num_layers):
    z = zs[-l]
    sp = sigmoid_prime(z)
    delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
    nabla_b[-l] = delta
    nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
return (nabla_b, nabla_w)

...

def cost_derivative(self, output_activations, y):
    """Return the vector of partial derivatives \partial C_x /
    \partial a for the output activations."""
    return (output_activations-y)

def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))

```