618487-2560 : ปัญญาประดิษฐ์สำหรับวิศวกร
Artificial Intelligence for Engineers

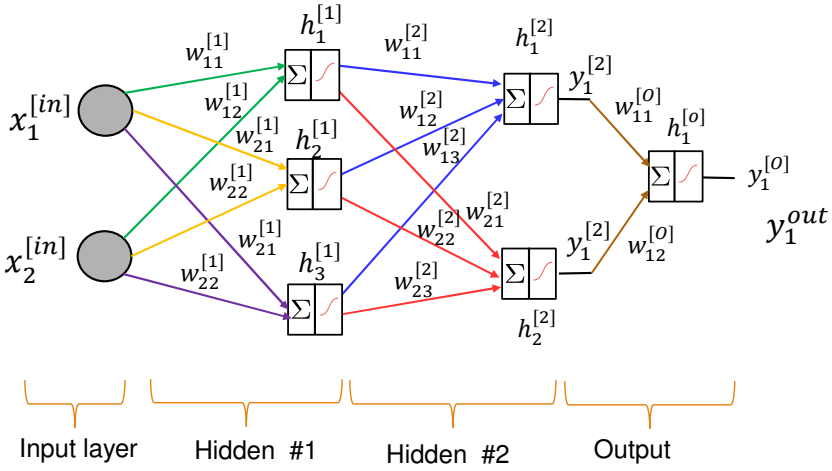
บรรยายครั้งที่ 12

เพอร์เซ็ปตรอนหลายชั้น
(Multilayer Perceptron)

ผศ.ดร. ยุทธนา เจวจินดา
Yutana Jewajinda, PhD.

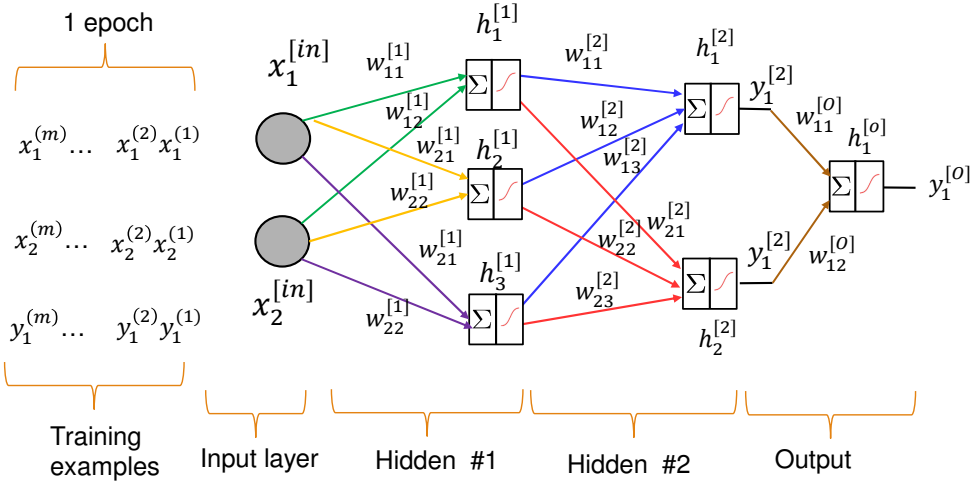
เพอร์เซ็ปตรอนหลายชั้น : นิยามสัญกรณ์

- Input, one or more hidden layers, and Output layer
- How to train them (finding weights)



เพอร์เซ็ปตรอนหลายชั้น : นิยามสัญกรณ์

- Training by *m* examples and learning algorithms
- (m) denoting m[th] number of the examples



เพอร์เซ็ปตรอนหลายชั้น : นิยามสัญกรณ์

- Row and Column vector

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad \boldsymbol{x^T} = [x_1 \quad x_2 \quad x_3]$$

- Dot Product $x \cdot x^T = (x_1x_1 + x_2x_2 + x_3x_3)$
- Weight Matrices of i[th] layer is [i[th]]

$$W^{[i]} = \begin{bmatrix} w_{11}^{[i]} & w_{12}^{[i]} & w_{13}^{[i]} & h_1^{[i]} \\ w_{21}^{[i]} & w_{22}^{[i]} & w_{23}^{[i]} & h_2^{[i]} \end{bmatrix}$$

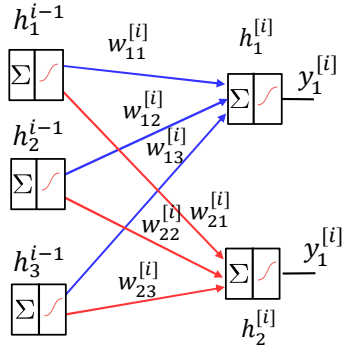
Output (next layer)

Input (previous layer)

$w_{11}^{[i]}$

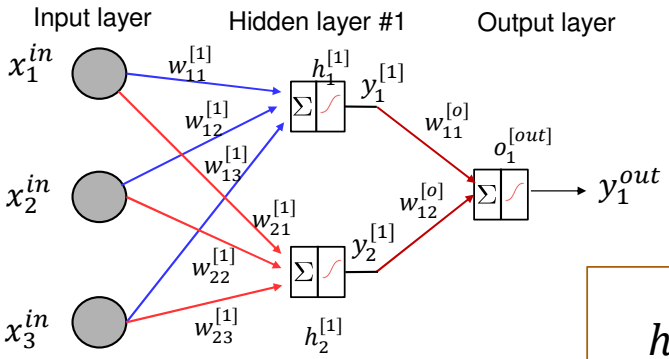
Current Layer neuron #1

Previous Layer neuron #1



เพอร์เซ็ปตรอนหลายชั้น : ชั้นซ่อน 1 ชั้น

- ประกอบด้วย input layer, output layer, and one or more hidden layers



- 3 input
- 1 hidden layer with 2 neurons
- Output layer with 1 neuron

Output $\begin{bmatrix} h_1^{[1]} \\ h_2^{[1]} \end{bmatrix}$ $\begin{bmatrix} w_{11}^{[1]} & w_{12}^{[1]} & w_{13}^{[1]} \\ w_{21}^{[1]} & w_{22}^{[1]} & w_{23}^{[1]} \end{bmatrix}$ Input

$\begin{bmatrix} w_{11}^{[o]} & w_{12}^{[o]} \end{bmatrix}$ o_1^{out}

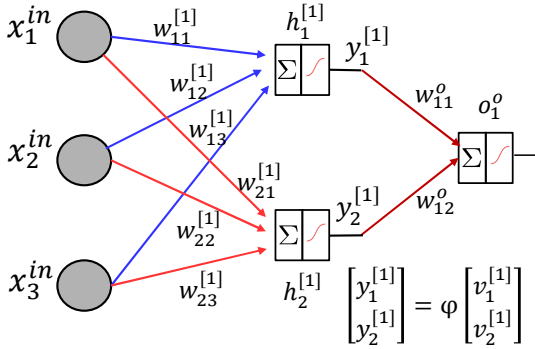
$h_1^{[1]}$ ← Hidden layer #1, Neuron #1

$w_{12}^{[1]}$ ← ith Layer, From neuron #2 of layer 1

Toward neuron #1 of layer i

เพอร์เซ็ปตรอนหลายชั้น : ชั้นซ่อน 1 ชั้น

- การคำนวณ Forward Pass ด้วยค่าอินพุต และน้ำหนัก
- $\varphi(\ldots)$ คือ activation function



$$v_1^{out} = \begin{bmatrix} w_{11}^{o} & w_{12}^{o} \end{bmatrix} \cdot \begin{bmatrix} y_1^{[1]} \\ y_2^{[1]} \end{bmatrix}$$

$$v_1^{out} = w_{11}^{o}\, y_1^{[1]} + w_{12}^{o} y_2^{[1]}$$

$$y_1^{o} = \varphi\left(v_1^{out}\right) \qquad (2)$$

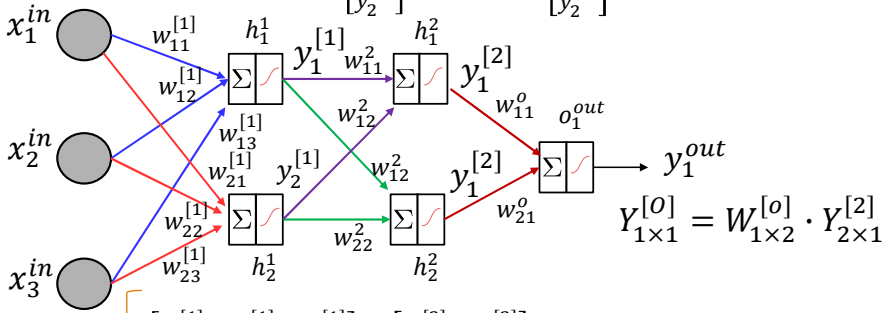
$$\begin{bmatrix} y_1^{[1]} \\ y_2^{[1]} \end{bmatrix} = \varphi \begin{bmatrix} v_1^{[1]} \\ v_2^{[1]} \end{bmatrix} \qquad (1)$$

$$\begin{bmatrix} v_1^{[1]} \\ v_2^{[1]} \end{bmatrix} = \begin{bmatrix} w_{11}^{[1]} & w_{12}^{[1]} & w_{13}^{[1]} \\ w_{21}^{[1]} & w_{22}^{[1]} & w_{23}^{[1]} \end{bmatrix} \bullet \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} \left(x_1 w_{11}^{[1]} + x_2 w_{12}^{[1]} + x_3 w_{13}^{[1]}\right) \\ \left(x_1 w_{21}^{[1]} + x_2 w_{22}^{[1]} + x_3 w_{23}^{[1]}\right) \end{bmatrix}$$

เพอร์เซ็ปตรอนหลายชั้น : ชั้นซ่อน 2 ชั้น

- 2 hidden layers

$$Y_{2x1}^{[1]} = \begin{bmatrix} y_1^{[1]} \\ y_2^{[1]} \end{bmatrix} \qquad Y_{2x1}^{[2]} = \begin{bmatrix} y_1^{[2]} \\ y_2^{[2]} \end{bmatrix}$$



$$Y_{1\times1}^{[O]} = W_{1\times2}^{[o]} \cdot Y_{2\times1}^{[2]}$$

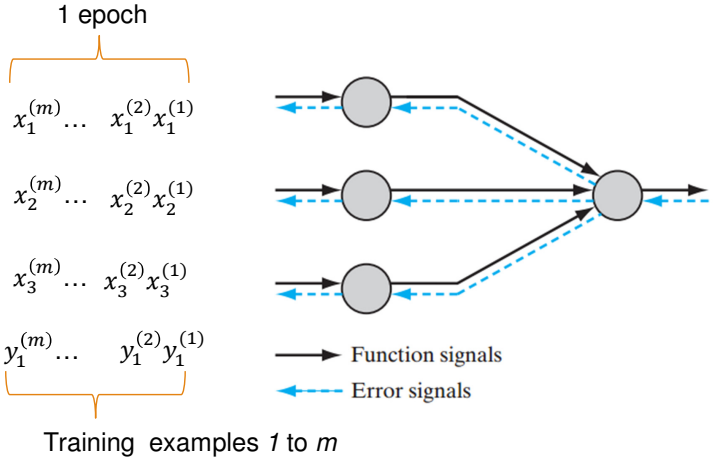
Output $\begin{bmatrix} w_{11}^{[1]} & w_{12}^{[1]} & w_{13}^{[1]} \\ w_{21}^{[1]} & w_{22}^{[1]} & w_{23}^{[1]} \end{bmatrix}$ $\begin{bmatrix} w_{11}^{[2]} & w_{12}^{[2]} \\ w_{21}^{[2]} & w_{22}^{[2]} \end{bmatrix}$ $\begin{bmatrix} w_{11}^{o} & w_{21}^{o} \end{bmatrix}$

x_1^{in} x_2^{in} y_3^{in} y_1^1 y_2^1 y_1^2 y_2^2

input

$$Y_{2\times1}^{[1]} = W_{2\times3}^{[1]} \cdot X_{3x1}^{[in]} \qquad Y_{2\times1}^{[2]} = W_{2\times2}^{[2]} \cdot Y_{2\times1}^{[1]}$$

Back Propagation Algorithm



1. Forward Pass โดยใช้ training set
2. Backward Pass โดยใช้ Back Propagation Algorithm

Batch Learning vs Mini batch

- adjustments to the synaptic weights of the multilayer perceptron are performed after the presentation of all the ***m*** examples in the training set *t* that constitute ***one epoch*** of training
- In other words, the cost function for batch learning is defined by the average error

$$E = \frac{1}{m}\sum_{n=1}^{m} e(n)$$

- Adjustments to the synaptic weights of the multilayer perceptron are made on an epoch-by-epoch basis

Batch Learning vs Mini batch

- adjustments to the synaptic weights of the multilayer perceptron are performed after the presentation of all the ***m*** examples in the training set *t* that constitute ***one epoch*** of training
- In other words, the cost function for batch learning is defined by the average error

$$E = \frac{1}{m}\sum_{n=1}^{m} e(n)$$

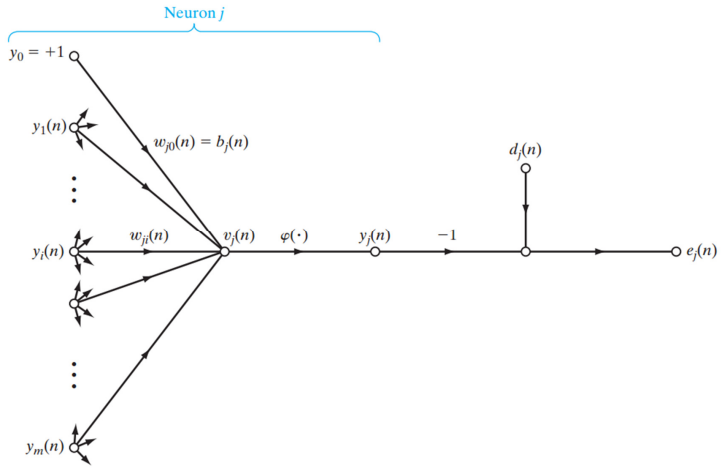
- Adjustments to the synaptic weights of the multilayer perceptron are made on an epoch-by-epoch basis

Online Learning

- Adjustments to the synaptic weights of the multilayer perceptron are performed on an example-by-example basis.
- The cost function to be minimized is therefore the total error of all output layer

$$\mathcal{E}(n) = \sum_{j\in C} \mathcal{E}_j(n)$$
$$= \frac{1}{2}\sum_{j\in C} e_j^2(n)$$

Back Propagation Algorithm : Backward pass



Back Propagation Algorithm : Backward pass

the error signal produced at

the output of neuron j is defined by

$$e_j(n) = d_j(n) - y_j(n)$$

instantaneous error of neuron j is defined by

$$\mathscr{E}_j(n) = \frac{1}{2}e_j^2(n)$$

total instantaneous error energy of the whole network as

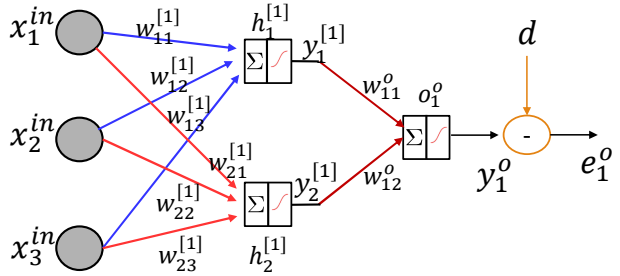
$$\mathscr{E}(n) = \sum_{j \in C} \mathscr{E}_j(n)$$
$$= \frac{1}{2}\sum_{j \in C} e_j^2(n)$$

where the set C includes all the neurons in the output layer

Back Propagation Algorithm : Backward pass

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n)$$

$$y_j(n) = \varphi_j(v_j(n))$$



applies a correction $\Delta w_{ji}(n)$ to the synaptic weight $w_{ji}(n)$

Using chain rule:

$$\frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} = \frac{\partial \mathscr{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

Back Propagation Algorithm : Backward pass

$$\frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} = \frac{\partial \mathscr{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

$$\frac{\partial \mathscr{E}(n)}{\partial e_j(n)} = e_j(n) \quad \frac{\partial e_j(n)}{\partial y_j(n)} = -1 \quad \frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n)) \quad \frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n)$$

Therefore

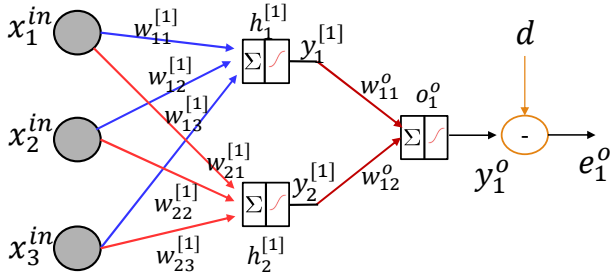
$$\frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n)$$

$$\delta_j(n) = e_j(n)\varphi_j'(v_j(n))$$

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)}$$

Back Propagation Algorithm : Backward pass

Case 1 Neuron *j* is an Output Node



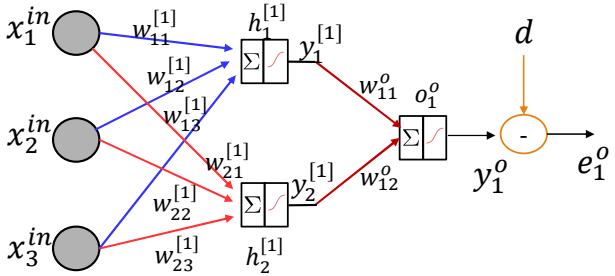
$$e_j(n) = d_j(n) - y_j(n)$$

$$\delta_j(n) = e_j(n)\varphi_j'(v_j(n))$$

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n)$$

Case 2 Neuron j is a Hidden Node

- When neuron j is located in a hidden layer of the network, there is no specified desired response for that neuron The error signal for a hidden neuron would have to be determined by working backwards



$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n), \qquad \text{neuron } j \text{ is hidden}$$
$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n)$$

Logistic/Sigmoid Function

$$\varphi_j(v_j(n)) = \frac{1}{1 + \exp(-a v_j(n))}, \qquad a > 0$$
$$\varphi_j'(v_j(n)) = \frac{a \exp(-a v_j(n))}{[1 + \exp(-a v_j(n))]^2}$$

$$\delta_j(n) = e_j(n)\varphi_j'(v_j(n))$$
$$= a[d_j(n) - o_j(n)]o_j(n)[1 - o_j(n)], \qquad \text{neuron } j \text{ is an output node}$$

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n)$$
$$= a y_j(n)[1 - y_j(n)] \sum_k \delta_k(n) w_{kj}(n), \qquad \text{neuron } j \text{ is hidden}$$

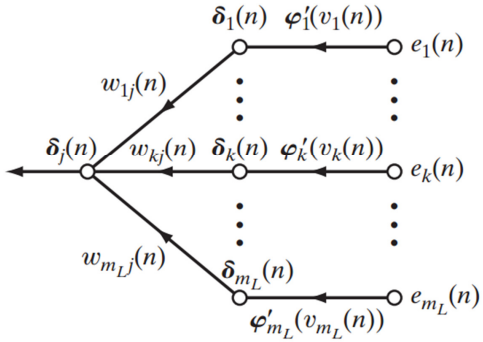
Derivative of Tanh Function

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad , \quad \frac{de^x}{dx} = e^x \quad , \quad \frac{de^{-x}}{dx} = -e^{-x}$$

$$\frac{d\tanh(x)}{dx} = \frac{(e^x + e^{-x})(e^x + e^{-x}) - (e^x - e^{-x})(e^x - e^{-x})^x}{e^x + e^{-x}}$$

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

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



$$\begin{pmatrix} \text{Weight} \\ \text{correction} \\ \Delta w_{ji}(n) \end{pmatrix} = \begin{pmatrix} \text{learning-} \\ \text{rate parameter} \\ \eta \end{pmatrix} \times \begin{pmatrix} \text{local} \\ \text{gradient} \\ \delta_j(n) \end{pmatrix} \times \begin{pmatrix} \text{input signal} \\ \text{of neuron } j, \\ y_i(n) \end{pmatrix}$$

Summary of the Back-Propagation Algorithm

1. Initialization โดยทำการสุ่มค่า Weight และ Bias ที่เข้าแต่ละ Neuron จากการกระจายตัวแบบ uniform โดยให้ mean เป็นศูนย์ และให้ ค่าเบี่ยงแบนมาตรฐาน (Standard deviation) อยู่ในช่วงออก activation function ที่ใช้
2. จัดรูปแบบชุดสอนโดยกำหนดขนาด Batch หรือ Minibatch
3. ทำ Forward Pass 1 epoch ถ้าใช้ Batch หรือ 1 ตัวอย่างถ้าใช้ Online Learning

$$y_j^{(0)}(n) = x_j(n)$$
$$v_j^{(l)}(n) = \sum_i w_{ji}^{(l)}(n) y_i^{(l-1)}(n)$$
$$y_j^{(l)} = \varphi_j(v_j(n))$$

ทำทุกชั้นจนถึงชั้น output และหา error ที่ output เทียบกับค่าที่ถูกต้อง

Summary of the Back-Propagation Algorithm

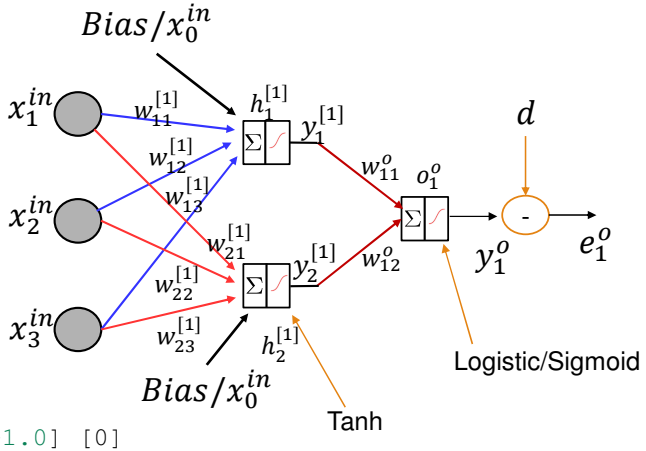
4. *Backward Computation.* Compute the δs (i.e., local gradients) of the network, defined by

$$\delta_j^{(l)}(n) = \begin{cases} e_j^{(L)}(n)\varphi_j'(v_j^{(L)}(n)) & \text{for neuron } j \text{ in output layer } L \\ \varphi_j'(v_j^{(l)}(n)) \sum_k \delta_k^{(l+1)}(n) w_{kj}^{(l+1)}(n) & \text{for neuron } j \text{ in hidden layer } l \end{cases}$$

$$w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) + \alpha[\Delta w_{ji}^{(l)}(n-1)] + \eta\delta_j^{(l)}(n) y_i^{(l-1)}(n)$$

5. *Iteration.* Iterate the forward and backward computations under points 3 and 4 by presenting new epochs of training examples to the network until the chosen stopping criterion is met.

การสร้าง MLP และทำการสอน




```
[1.0, -1.0,-1.0] [0]
[1.0, -1.0, 1.0] [1]
[1.0, 1.0, -1.0] [1]
[1.0, 1.0,  1.0] [0]
```


การสร้าง MLP และทำการสอน

```
import numpy as np

np.random.seed(3) # To make repeatable
LEARNING_RATE = 0.1
index_list = [0, 1, 2, 3] # Used to randomize order

# Define training examples.
x_train = [np.array([1.0, -1.0, -1.0]),
           np.array([1.0, -1.0, 1.0]),
           np.array([1.0, 1.0, -1.0]),
           np.array([1.0, 1.0, 1.0])]
y_train = [0.0, 1.0, 1.0, 0.0] # Output (ground truth)
```

❑ สร้างชุดข้อมูลสอน (Training set)

การสร้าง MLP และทำการสอน

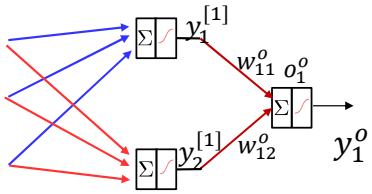
```
def neuron_w(input_count):
    weights = np.zeros(input_count+1)
    for i in range(1, (input_count+1)):
        weights[i] = np.random.uniform(-1.0, 1.0)
    return weights

n_w_o = [neuron_w(2)]
n_w_h1 = [neuron_w(2), neuron_w(2)]
n_yo = [0]
n_yh1 = [0, 0]
n_grad_o = [0]
n_grad_hd1 =  [0, 0]
```

- ฟังก์ชัน neuron_w ทำการสร้างค่าน้ำหนักเป็น Rank 1 array (x,)
- n_w_o เก็บน้ำหนักของชั้นเอาร์พุต สร้างเป็น List of rank 1 array
- n_w_h1 เก็บน้ำหนักของชั้นซ่อน 1 สร้างเป็น List of 2 rank 1 array
- n_yo n_yh1 เอาร์พุตของ neuron ที่ชั้น Output และ Hidden #1
- n_grad_o n_grad_hd1 ค่า Gradient ของชั้น output และ hidden

การสร้าง MLP และทำการสอน

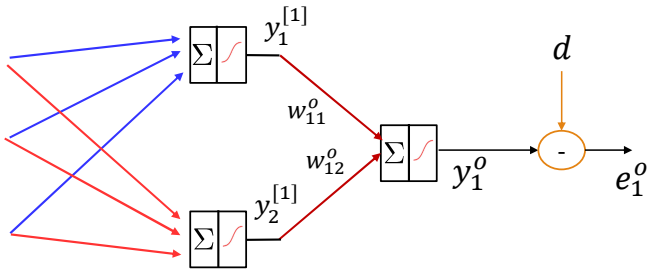
```
def forward_pass(x):
    global n_yo, n_yh1
    n_yh1[0] = np.tanh(np.dot(n_w_h1[0], x)) # hidden 1
    n_yh1[1] = np.tanh(np.dot(n_w_h1[1], x)) # hidden 2
    n2_inputs = np.array([1.0, n_yh1[0], n_yh1[1]]) # 1.0 is bias
    zo = np.dot(n_w_o, n2_inputs)
    n_yo = 1.0 / (1.0 + np.exp(- zo))  #sigmoid
```



- คำนวณ Forward Pass จากตัวอย่างสอน 1 ตัวอย่าง x เป็น rank 1 อะเรย์
- n_yh1[0] เป็นเอาร์พุตของนิวรอนชั้นซ่อน 1 ตัวที่ 1
- n_yh1[1] เป็นเอาร์พุตของนิวรอนชั้นซ่อน 1 ตัวที่ 2
- n_yo เป็นเอาร์พุตของนิวรอนชั้นเอาร์พุต

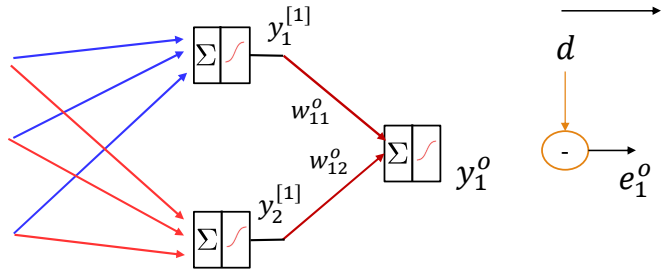
การสร้าง MLP และทำการสอน

```
def backward_pass(y_truth):
    global n_grad_o, n_grad_hd1
    error = -(y_truth - n_yo) # Derivative of loss-func
    derivative_sigmoid  = n_yo * (1.0 - n_yo) # Logistic derivative
    n_grad_o = error * derivative_sigmoid
    derivative_tanh = 1.0 - n_yh1[0]**2 # tanh derivative
    n_grad_hd1[0] = n_w_o[0][1] * n_grad_o * derivative_tanh
    derivative_tanh = 1.0 - n_yh1[1]**2 # tanh derivative
    n_grad_hd1[1] = n_w_o[0][2] * n_grad_o * derivative_tanh
    print('n_grad_o:', n_grad_o)
    print('n_grad_hd1:', n_grad_hd1[0], '   ', n_grad_hd1[1])
```



การสร้าง MLP และทำการสอน

```
def adjust_weights(x):
    global n_w_o ,  n_w_h1
    n2_inputs = np.array([1.0, n_yh1[0], n_yh1[1]]) # 1.0 is bias
    n_w_o -= (n2_inputs * LEARNING_RATE * n_grad_o)
    n_w_h1[0] -= (x * LEARNING_RATE * n_grad_hd1[0])
    n_w_h1[1] -= (x * LEARNING_RATE * n_grad_hd1[1])
```



- ปรับค่าน้ำหนักโดยใช้ gradient
- $W_{i+1} = W_i$ - (x * LEARNING_RATE * gradient_of_the neuron)

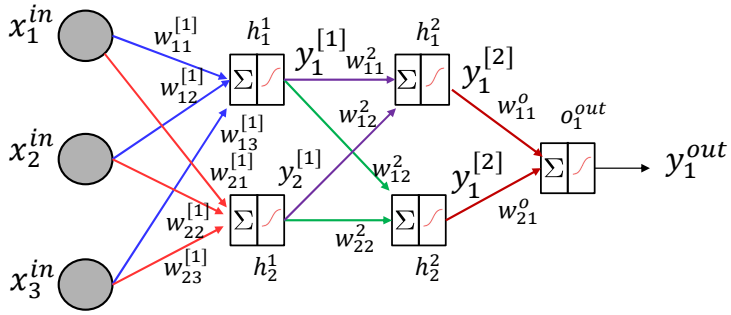
การสร้าง MLP และทำการสอน

```
all_correct = False
MAX_EPOCH = 200
for x in range(MAX_EPOCH):
    print('epoch: ', x)
    all_correct = True
    np.random.shuffle(index_list) # Randomize order
    for i in index_list: # Train on all examples
        forward_pass(x_train[i])
        backward_pass(y_train[i])
        adjust_weights(x_train[i])
        show_learning() # Show updated weights
    for i in range(len(x_train)): # Check if converged
        forward_pass(x_train[i])
```

- สอนด้วยตัวอย่าง และปรับน้ำหนักทุกครั้งที่ สอน 1 ตัวอย่าง
- ขั้นตอนคือ
  1. forward_pass(x_train[i])
  2. backward_pass(y_train[i])
  3. adjust_weights(x_train[i])

กิจกรรมในชั้นเรียนที่ 12-1

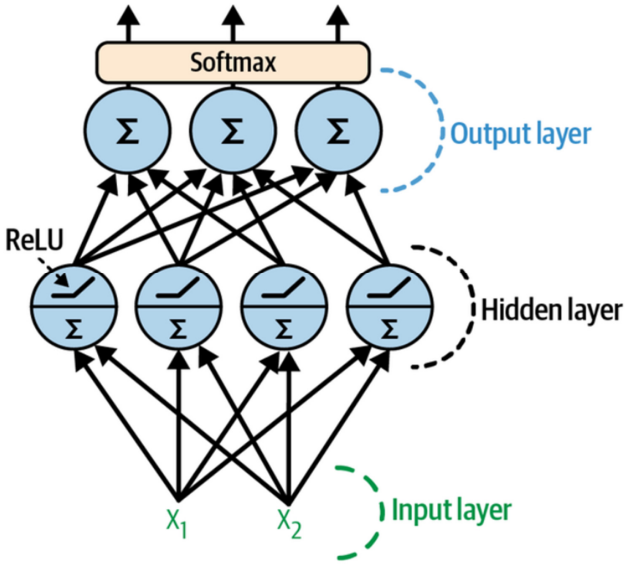
- ทดลอง Run โปรแกรมจากตัวอย่างที่เรียน โดย Download จากเว็บของรายวิชา
  - ทำการปรับจำนวน epoch ที่สอน เป็น 50 และ 300
  - จากนั้นสังเกตการณ์เปลี่ยนแปลงของค่าที่ชั้น output
- แก้ไขโครงสร้างของ MLP และทำการสอน MLP ใหม่โดยโครงสร้างต่อไปนี้



Typical classification MLP architecture

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
# hidden layers	Typically 1 to 5 layers, depending on the task		
# output neurons	1	1 per binary label	1 per class
Output layer activation	Sigmoid	Sigmoid	Softmax
Loss function	X-entropy	X-entropy	X-entropy

Typical classification MLP architecture



Typical classification MLP architecture

Softmax function
(Normalized exponential function)

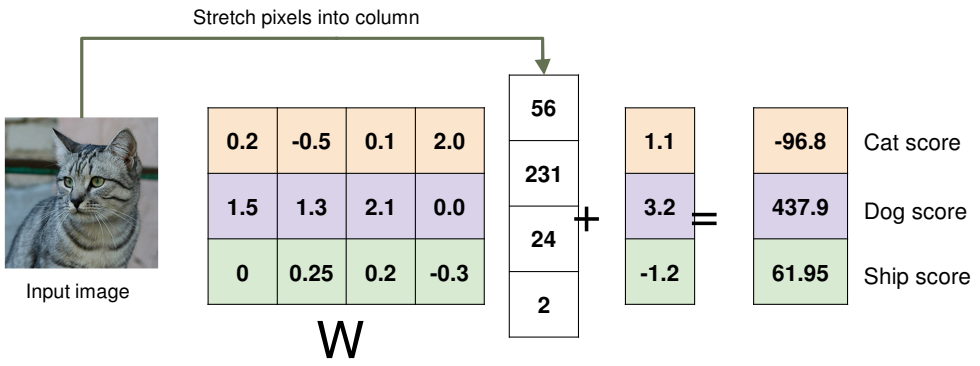
$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

- If we take an input of [1,2,3,4,1,2,3], the softmax of that is [0.024, 0.064, 0.175, 0.475, 0.024, 0.064, 0.175].
- The softmax function highlights the largest values and suppress other values.
- softmax is differentiable.

การใช้งาน Tensorflow/Keras สร้าง MLP

การใช้งาน Tensorflow/Keras สร้าง MLP

Example with an image with 4 pixels, and 3 classes (cat/dog/ship)



การใช้งาน Tensorflow/Keras สร้าง MLP

```
import tensorflow as tf

fashion_mnist = tf.keras.datasets.fashion_mnist.load_data()
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist
X_train, y_train = X_train_full[:-5000], y_train_full[:-5000]
X_valid, y_valid = X_train_full[-5000:], y_train_full[-5000:]
```

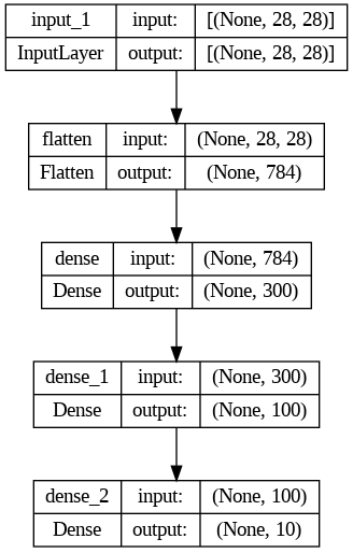
- Fashion MNIST ชุดสอนมี 60,000 ภาพ
- ชุดทดสอบมี test set 10,000 ภาพ
- แบ่งเป็นชุดสอน 55000 ภาพ และ ชุด validation 5000 ภาพ
- แต่ภาพมีขนาด 28 x 28 Pixels
- มี 10 ประเภท class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat", "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]

การใช้งาน Tensorflow/Keras สร้าง MLP

```
tf.random.set_seed(42)
model = tf.keras.Sequential()
model.add(tf.keras.layers.InputLayer(input_shape=[28, 28]))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(300, activation="relu"))
model.add(tf.keras.layers.Dense(100, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

```
Model: "sequential"
_________________________________________________________________
___ Layer (type) Output Shape Param #
=================================================================
=== flatten (Flatten) (None, 784) 0
dense (Dense) (None, 300) 235500
dense_1 (Dense) (None, 100) 30100
dense_2 (Dense) (None, 10) 1010
=================================================================
=== Total params: 266,610 Trainable params: 266,610 Non-
trainable params: 0
```


การใช้งาน Tensorflow/Keras สร้าง MLP

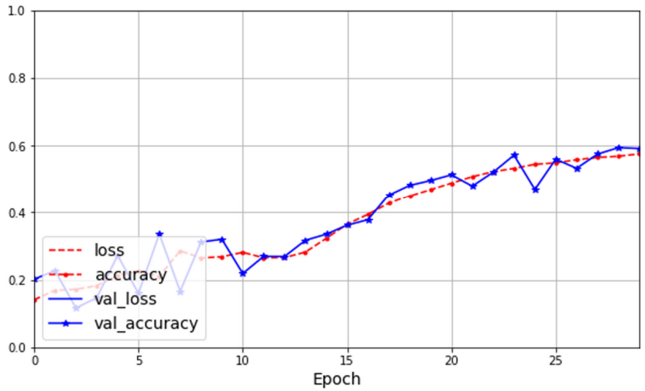


การใช้งาน Tensorflow/Keras สร้าง MLP

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

```
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid))
```

```
model.evaluate(X_test, y_test)
```



Validation accuracy
Acc

กิจกรรมในชั้นเรียนที่ 12-2

- ❑ ใช้ Project โค๊ดจาก web ของรายวิชา บรรยายครั้ง 10 ในรูปไฟล์ Jupyer notebook
  1. keras_mlp.ipnb
- ❑ ทำลอง Run ตัวอย่างที่เรียน โดยกด Shift + Enter ในแต่โค๊ดบล็อก
- ❑ ทดลองแก้ไขโครงสร้างของ MLP โดยการเพิ่มลดชั้น หรือ ลดจำนวนนิวรอนในชั้น