ESS2222

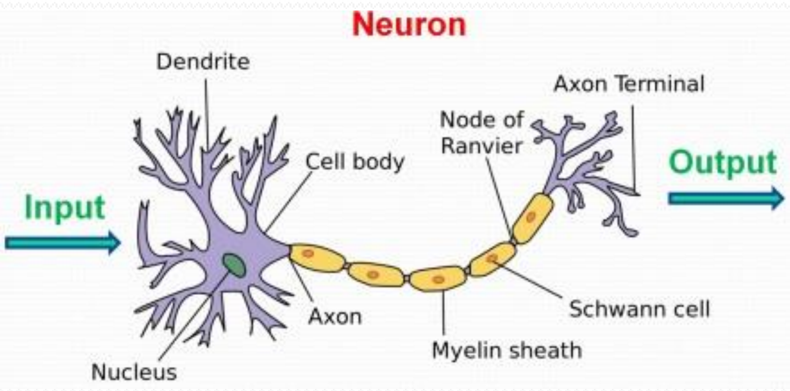
Lecture 6 – Neural Networks

Hosein Shahnas

University of Toronto, Department of Earth Sciences,

Outline

- ❑ **Combining Perceptrons**
- ❑ **Optimization**
- ❑ **Neural Networks**
- ❑ **Applying SGD & Recursion Relation**
- ❑ **Back Propagation Algorithm**

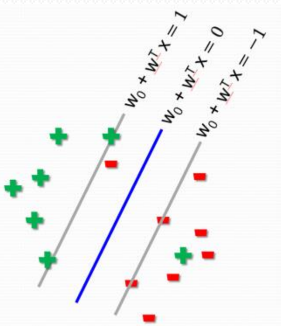


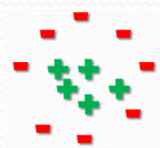
Review of Lecture 5

Soft margin SVM for slightly nonlinear problems
Kernel method for seriously nonlinear problems

Minimize $\frac{1}{2}\|w\|^2 + C\sum_i \xi^i$, $\xi^i \geq 0$

Subject to: $y^i (w_0 + w^T x^i) \geq 1 - \xi^i \quad \forall i$ where $\xi^i \geq 0$





One-vs.-All (OvA)

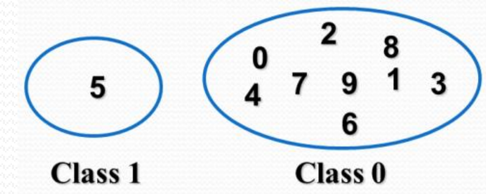
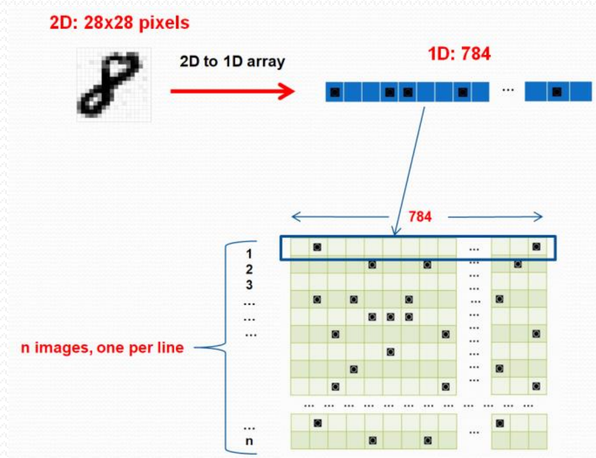


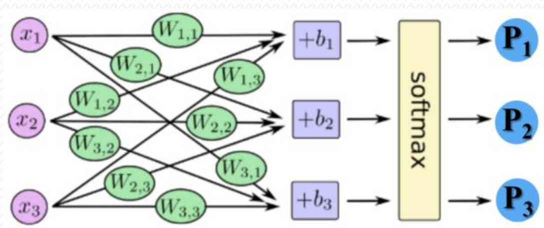
Image Recognition



Softmax

Generalization of the logistic function to multi-class settings

$$\Phi(z_j) = \frac{1}{1+e^{-z_j}} \longrightarrow \text{softmax}(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

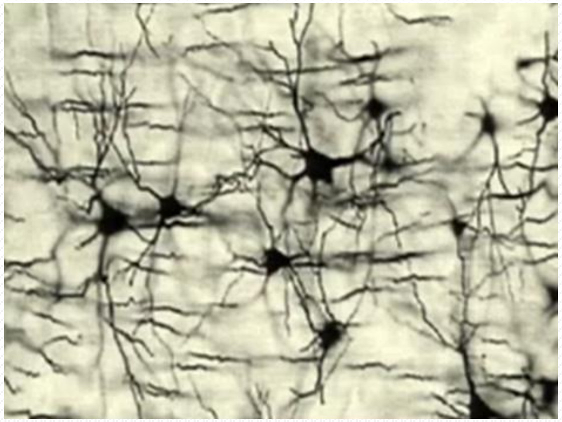


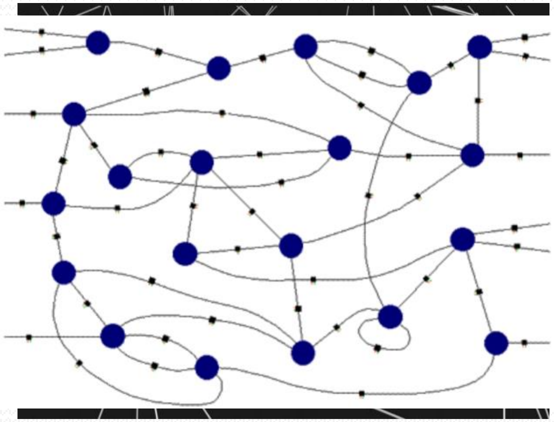
Biological Neural Structure

Biology as inspiration

Engineering

Bilogical function → Biological structure





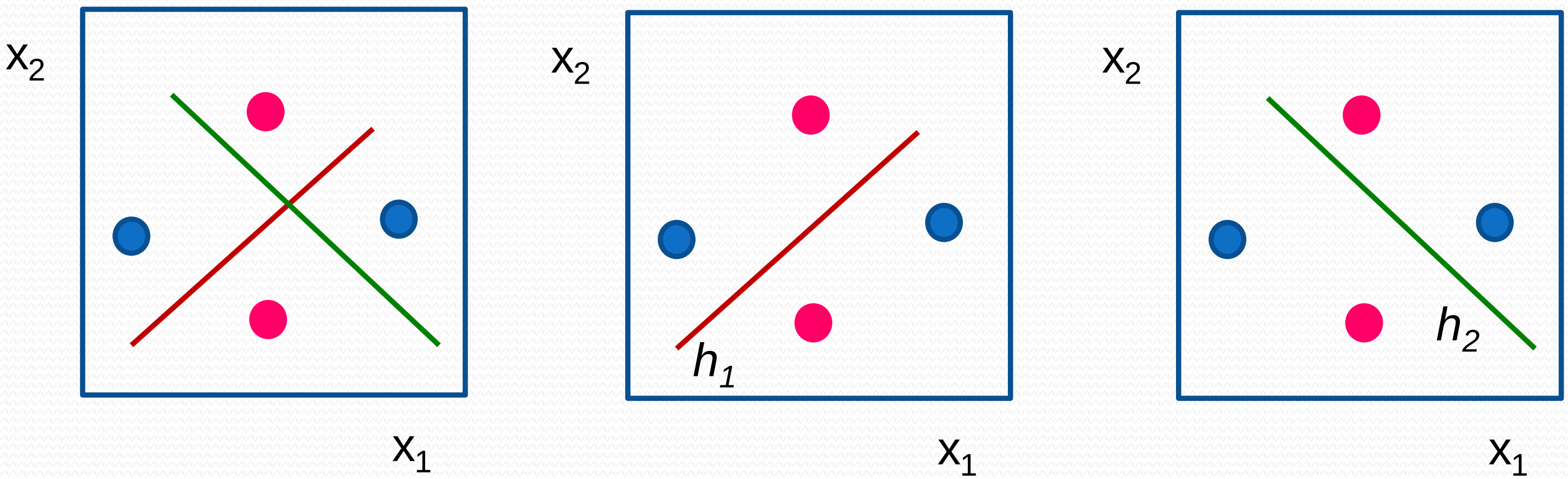
Perceptrons are the building blocks of the neural networks connected by synapses. So we may get the human intelligence by combining these building blocks.

Imitating not exact: Imitating biology has a limit. The airplane flies but doesn't flap wings! Our engineering does not depend on the details.

Combining Perceptrons

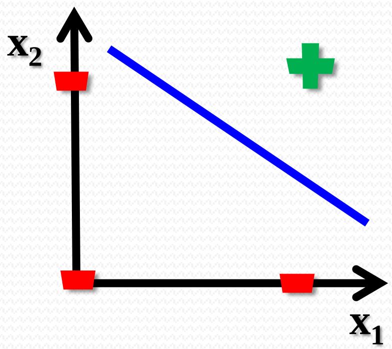
Let's explore what we can do with combinations of perceptrons rather than single ones.

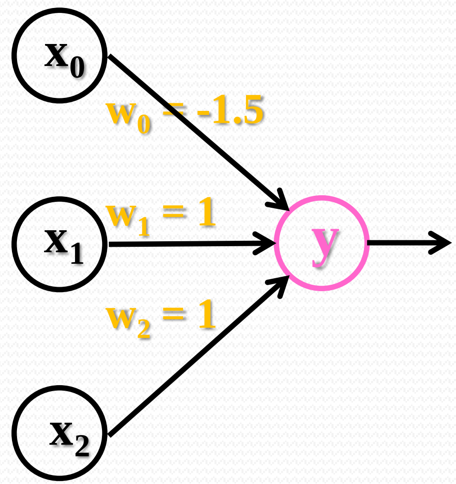
Let's consider the classification problem for which the perceptron algorithm failed.



This problem cannot be classified by a single perceptron. But what about with two perceptrons?

Combining Perceptrons





	AND	
x_1	x_2	Output
0	0	0
0	1	0
1	0	0
1	1	1

$$\sigma \equiv \phi(z) = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases} \quad z = \sum_{i=0}^{n} w_i\, x_i$$

Class 0

Class 1

$$\sigma = x_1\, w_1 + x_2\, w_2 + w_0$$

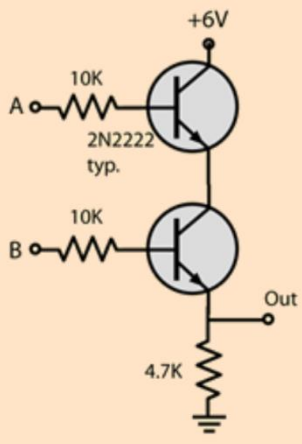
$$\sigma = \phi(1 * 0 + 1 * 0 - 1.5) = 0$$
$$\sigma = \phi(1 * 0 + 1 * 1 - 1.5) = 0$$
$$\sigma = \phi(1 * 1 + 1 * 0 - 1.5) = 0$$
$$\sigma = \phi(1 * 1 + 1 * 1 - 1.5) = 1$$

y

And

$$x_1\, w_1 + x_2\, w_2 + w_0 = 0 \qquad x_2 = \frac{-w_1}{w_2} x_1 - \frac{w_0}{w_2}$$

Logic Gates



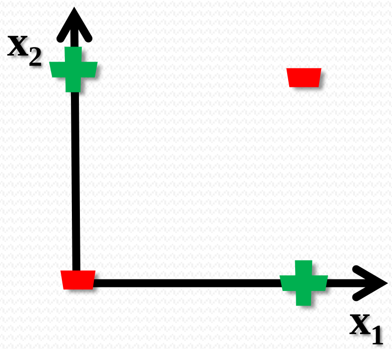
	OR	
x_1	x_2	Output
0	0	0
0	1	1
1	0	1
1	1	1

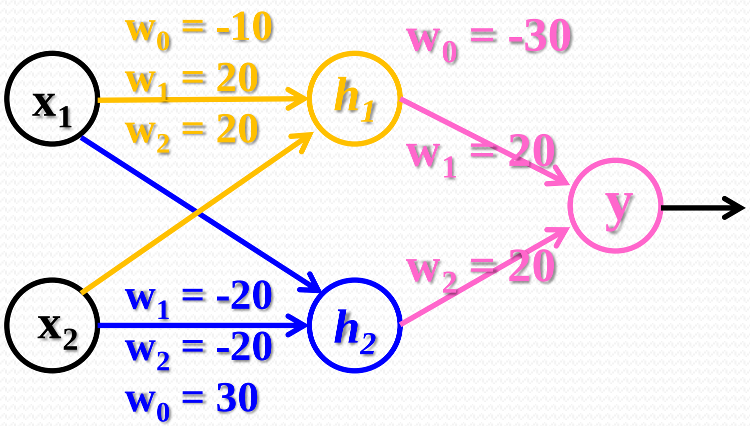
	NAND	
x_1	x_2	Output
0	0	1
0	1	1
1	0	1
1	1	0

	AND	
x_1	x_2	Output
0	0	0
0	1	0
1	0	0
1	1	1

	XOR	
x_1	x_2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Combining Perceptrons





$$\sigma \equiv \phi(z) = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases} \quad z = \sum_i w_i x_i$$

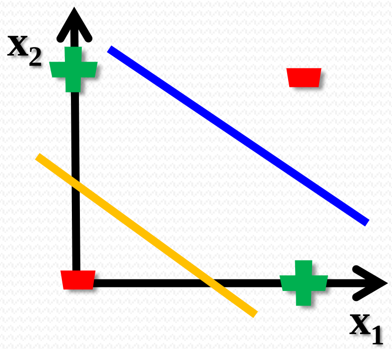
$\sigma = \phi(20 * 0 + 20 * 0 - 10) = 0$
$\sigma = \phi(20 * 1 + 20 * 1 - 10) = 1$
$\sigma = \phi(20 * 0 + 20 * 1 - 10) = 1$
$\sigma = \phi(20 * 1 + 20 * 0 - 10) = 1$

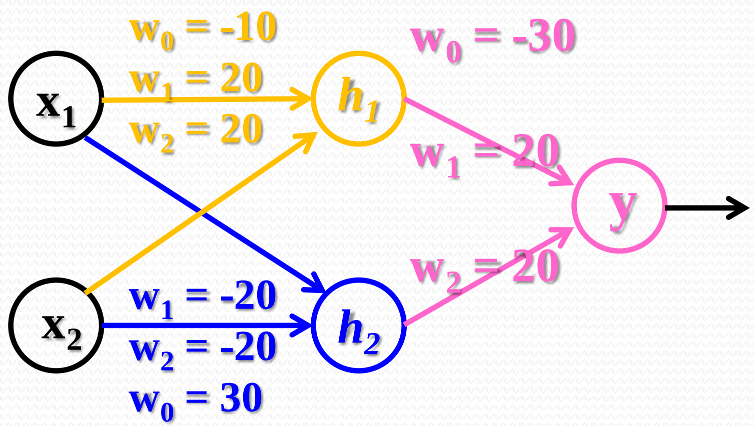
$\sigma = \phi(-20 * 0 - 20 * 0 + 30) = 1$
$\sigma = \phi(-20 * 1 - 20 * 1 + 30) = 0$
$\sigma = \phi(-20 * 0 - 20 * 1 + 30) = 1$
$\sigma = \phi(-20 * 1 - 20 * 0 + 30) = 1$

$\sigma = \phi(20 * 0 + 20 * 1 - 30) = 0$
$\sigma = \phi(20 * 1 + 20 * 0 - 30) = 0$
$\sigma = \phi(20 * 1 + 20 * 1 - 30) = 1$
$\sigma = \phi(20 * 1 + 20 * 1 - 30) = 1$

h_1	h_2	y
Or	Nand	And

Combining Perceptrons





$\sigma \equiv \phi(z) = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases}$ $z = \sum_i w_i x_i$

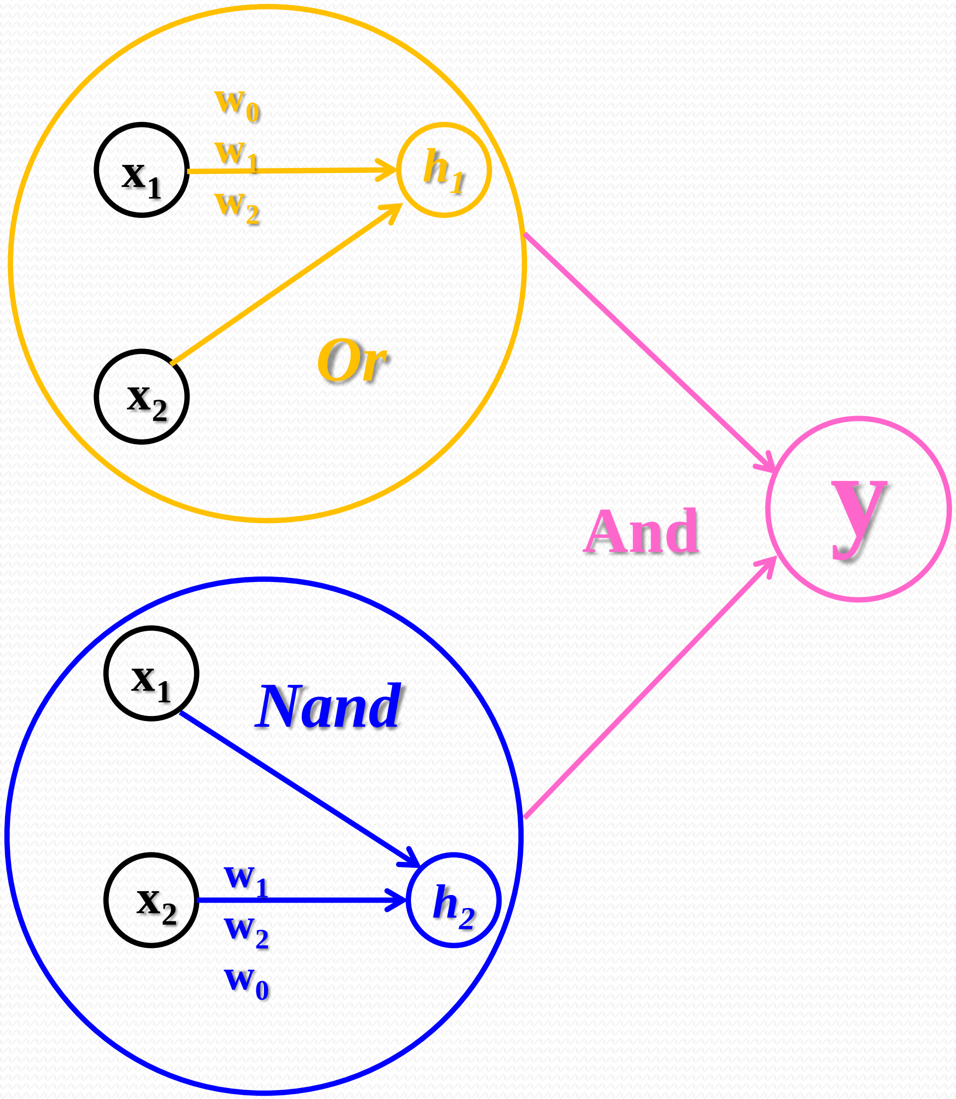
$\sigma = \phi(20 * 0 + 20 * 0 - 10) = 0$
$\sigma = \phi(20 * 1 + 20 * 1 - 10) = 1$
$\sigma = \phi(20 * 0 + 20 * 1 - 10) = 1$
$\sigma = \phi(20 * 1 + 20 * 0 - 10) = 1$

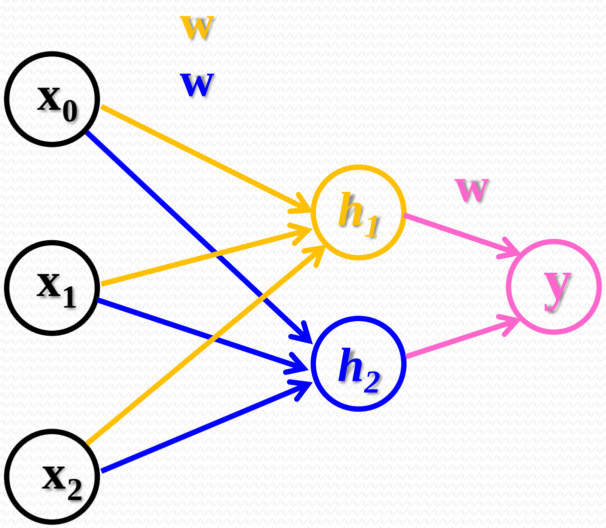
$\sigma = \phi(-20 * 0 - 20 * 0 + 30) = 1$
$\sigma = \phi(-20 * 1 - 20 * 1 + 30) = 0$
$\sigma = \phi(-20 * 0 - 20 * 1 + 30) = 1$
$\sigma = \phi(-20 * 1 - 20 * 0 + 30) = 1$

$\sigma = \phi(20 * 0 + 20 * 1 - 30) = 0$
$\sigma = \phi(20 * 1 + 20 * 0 - 30) = 0$
$\sigma = \phi(20 * 1 + 20 * 1 - 30) = 1$
$\sigma = \phi(20 * 1 + 20 * 1 - 30) = 1$

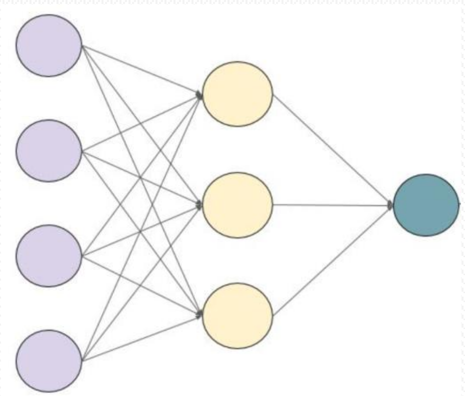
h_1	h_2	y
Or	*Nand*	**And**

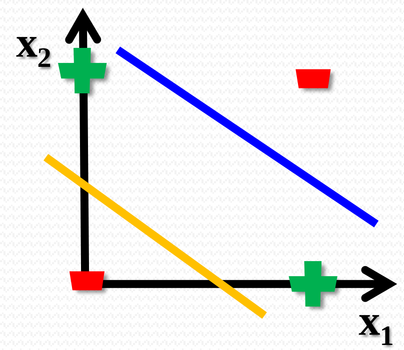
Combining Perceptrons

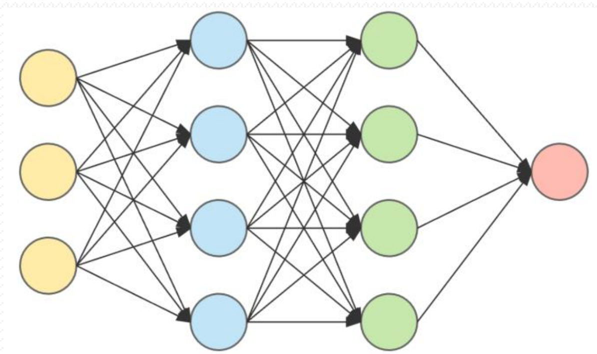


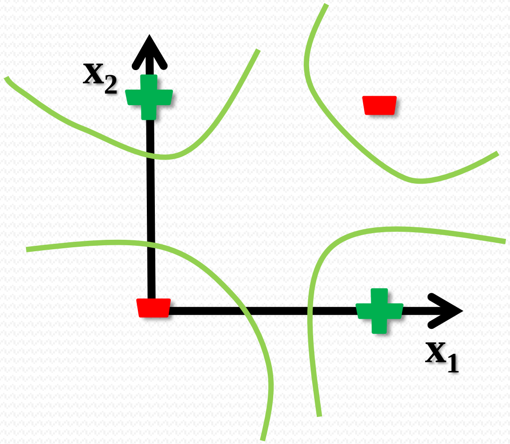


Combining Perceptrons
Multilayer Perceptron

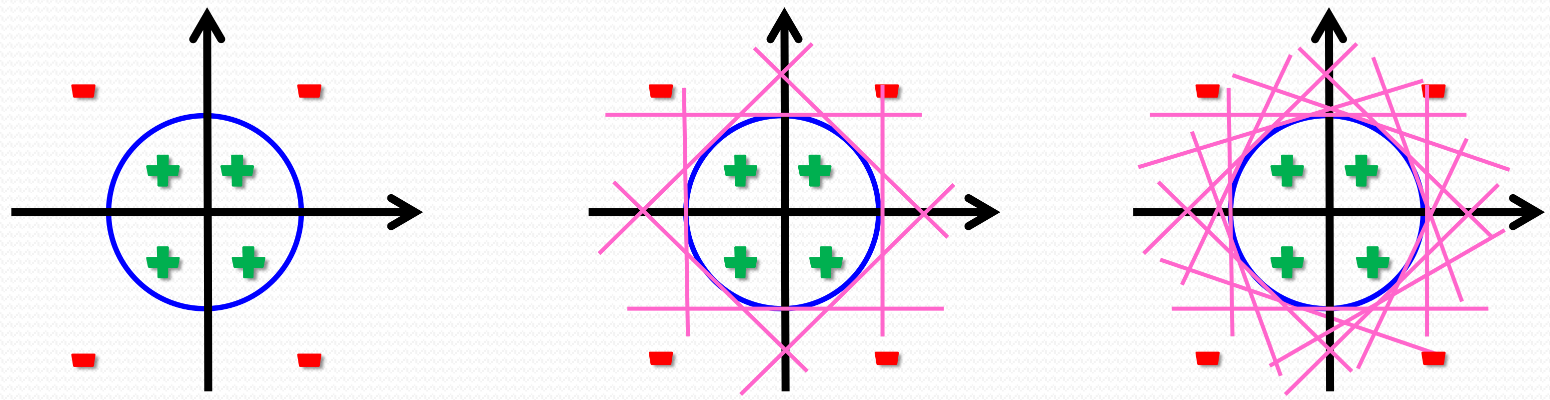








Combining Perceptrons
Multilayer Perceptron



8-Perceptrons 16-Perceptrons

We solved this problem using feature-transformation before.

We can use neural networks.

Optimization

There are many perceptrons (and therefore many parameters), so optimization might be a problem (remember that for single perceptron when the data were nonlinear, we had convergence problem).

Solution:

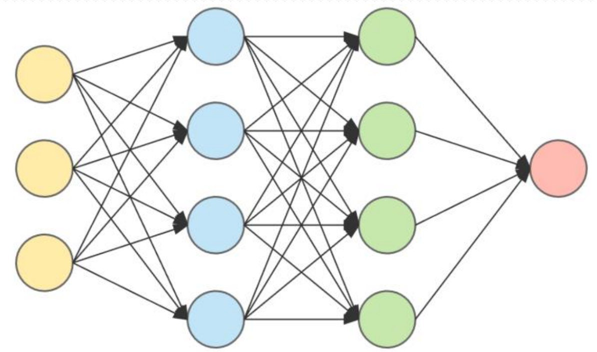
1- We chose a soft threshold (tanh) rather than a hard threshold (step function).

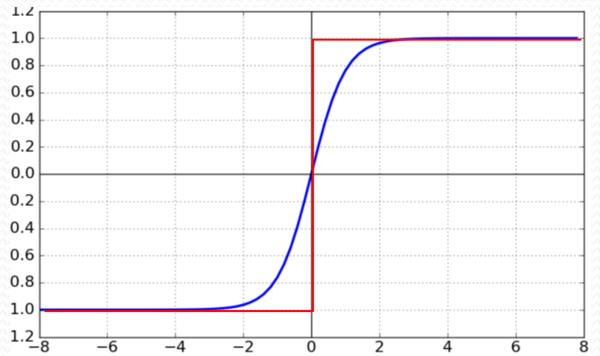
2- We use SGD

3- And an efficient way to find weight factors w.

$$\phi(z) = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$\phi'(z) = 1 - \tanh(z)^2$$





Step function

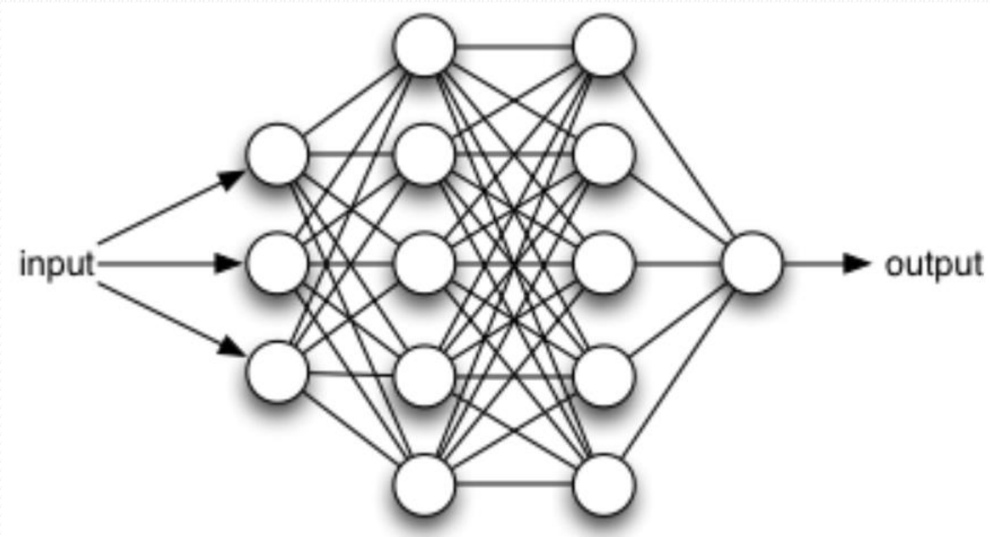
tanh

Input **Hidden layers** **output**

0 **1≤ $l < L$** **L**

Neural Networks

$$w_{ij}{}^{l} \begin{cases} 1 \le l < L & \textit{hidden layers} \\ l = 0 & \textit{input layer} \\ l = L & \textit{output layer} \\ 0 \le i \le d^{(l-1)} & \textit{inputs} \\ 1 \le j \le d^{(l)} & \textit{outputs} \end{cases}$$



$$x_j{}^{l} = \phi\left(z_j{}^{(l)}\right) = \phi\left(\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)}\ x_i{}^{(l-1)}\right)$$

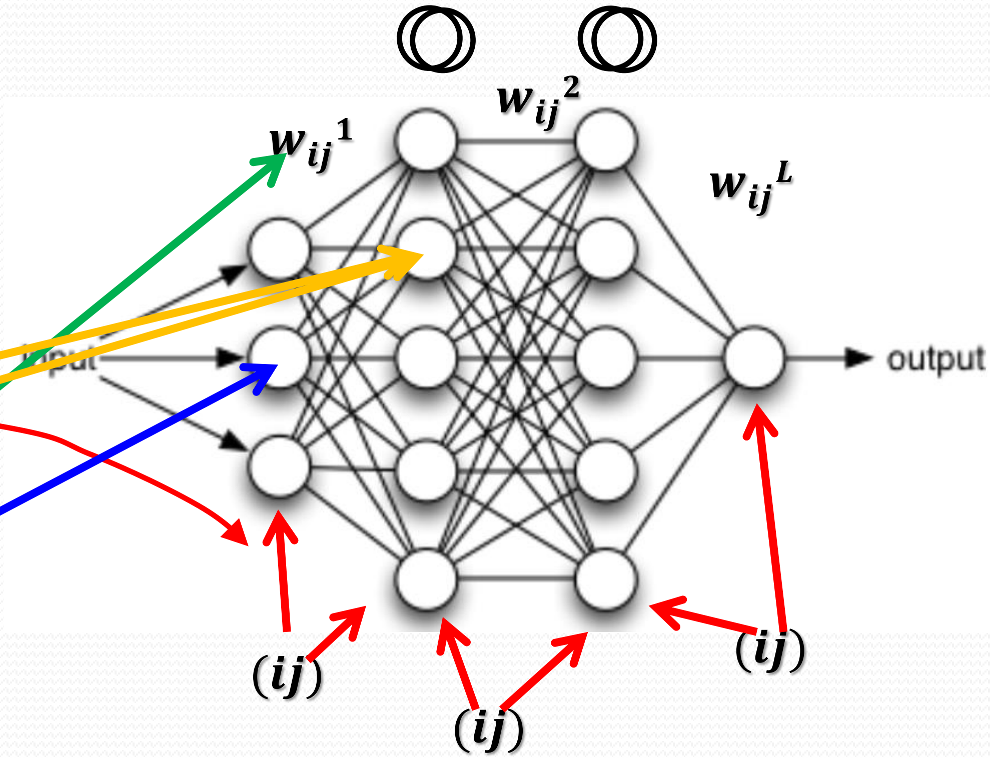
$$\begin{bmatrix} x_1{}^{(l)} \\ x_2{}^{(l)} \\ \\ \\ x_{d^{(l)}} \end{bmatrix}$$

$d^{(0)}$: dimension of the feature space

$$\begin{bmatrix} x_0{}^{(0)} \\ x_1{}^{(0)} \\ x_2{}^{(0)} \end{bmatrix} \begin{bmatrix} x_0{}^{(l)} \\ x_1{}^{(l)} \\ x_2{}^{(l)} \\ x_3{}^{(l)} \\ x_4{}^{(l)} \\ x_5{}^{(1)} \end{bmatrix} \begin{bmatrix} x_0{}^{(2)} \\ x_1{}^{(2)} \\ x_2{}^{(2)} \\ x_3{}^{(2)} \\ x_4{}^{(2)} \\ x_5{}^{(2)} \end{bmatrix} \begin{bmatrix} x_1{}^{(3)} \end{bmatrix}$$

Neural Networks

$$w_{ij}^{l} \begin{cases} 1 \le l < L & \textit{hidden layers} \\ l = 0 & \textit{input layer} \\ l = L & \textit{output layer} \\ 0 \le i \le d^{(l-1)} & \textit{inputs} \\ 1 \le j \le d^{(l)} & \textit{outputs} \end{cases}$$



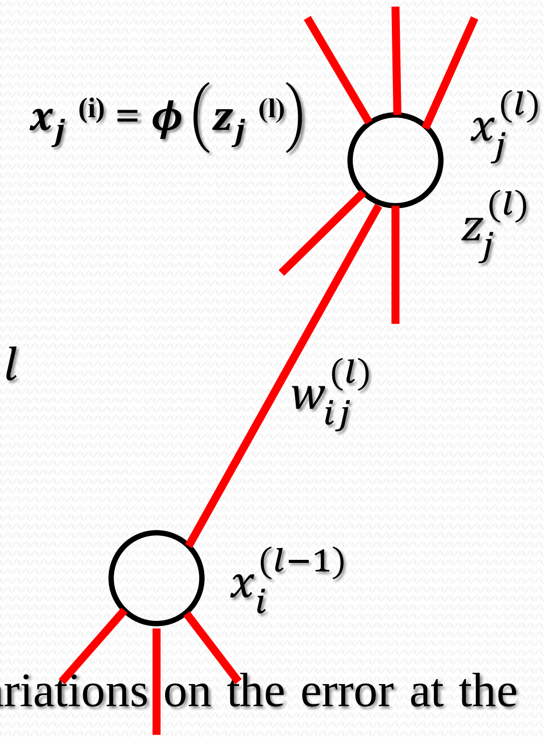
$$x_j^{\,l} = \phi\left(z_j^{(l)}\right) = \phi\left(\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)}\, x_i^{(l-1)}\right)$$

$$\begin{bmatrix} x_1^{(l)} \\ x_2^{(l)} \\ \\ \\ x_{d^{(l)}} \end{bmatrix}$$

$d^{(0)}$: dimension of the feature space

$$\begin{bmatrix} x_0^{(0)} \\ x_1^{(0)} \\ x_2^{(0)} \end{bmatrix} \begin{bmatrix} x_0^{(1)} \\ x_1^{(1)} \\ x_2^{(1)} \\ x_3^{(1)} \\ x_4^{(1)} \\ x_5^{(1)} \end{bmatrix} \begin{bmatrix} x_0^{(2)} \\ x_1^{(2)} \\ x_2^{(2)} \\ x_3^{(2)} \\ x_4^{(2)} \\ x_5^{(2)} \end{bmatrix} \begin{bmatrix} x_1^{(3)} \end{bmatrix}$$

Applying SGD



All the weights $w = \left\{w_{ij}^{(l)}\right\}$ determine $h(x)$.

Error on sample (x_n, y_n): $e(w) = e(h(x_n), y_n)$

To implement SGD, we need to calculate: $\nabla e(w) = \frac{\partial e(w)}{\partial w_{ij}^{(l)}} \quad \forall\, i, j, l$

Computing $\frac{\partial e(w)}{\partial w_{ij}^{(l)}}$:

$\frac{\partial e(w)}{\partial w_{ij}^{(l)}}$ can be calculated by perturbing $w_{ij}^{(l)}$ and observing the variations on the error at the output and get numerical estimates for partial derivatives. The problem with this approach is that we have to do this for all $w_{ij}^{(l)}$.

But we can obtain a recursion relation and then get all coefficients using this formula.

Recursion Relation

$$\frac{\partial e(w)}{\partial w_{ij}^{(l)}} = \frac{\partial e(w)}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}}$$ (chain rule)

Let $\delta_j^{(l)} = \frac{\partial e(w)}{\partial z_j^{(l)}}$ $\longrightarrow$ $\frac{\partial e(w)}{\partial w_{ij}^{(l)}} = \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} \delta_j^{(l)}$

But since $z_j^{(l)} = \sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}$ $\longrightarrow$ $\frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} = x_i^{(l-1)}$

Then: $\frac{\partial e(w)}{\partial w_{ij}^{(l)}} = \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} \delta_j^{(l)} = x_i^{(l-1)} \delta_j^{(l)}$

The only thing we need is $\delta_j^{(l)} = \frac{\partial e(w)}{\partial z_j^{(l)}}$

If we can find a recursion relation for $\delta_j^{(l)}$, then we can compute all of them by knowing one of them.

We compute $\delta_j^{(l)}$ for the final layer, because if we know δ later we can obtain δ earlier (back propagation).

δ for Final Layer

For l = L, j = 1, $\delta_j^{(l)} = \frac{\partial e(w)}{\partial z_j^{(l)}} \rightarrow \delta_1^{(L)} = \frac{\partial e(w)}{\partial z_1^{(L)}}$

We have $e(w) = e(h(x_n), y_n)$, but for the final layer: $h(x_n) = \phi(z_1^{(L)}) = x_1^{(L)}$

Then: $e(w) = e(x_1^{(L)}, y_n)$

If $e(w) = (h(x_n) - y_n)^2$ then $e(w) = (x_1^{(L)} - y_n)^2$

For tanh-activation function: $\phi(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$, $\phi'(z) = 1 - \tanh(z)^2$

Back Propogation of δ

Now we want to calculate: $\delta_i^{(l-1)} = \frac{\partial e(w)}{\partial z_i^{(l-1)}}$

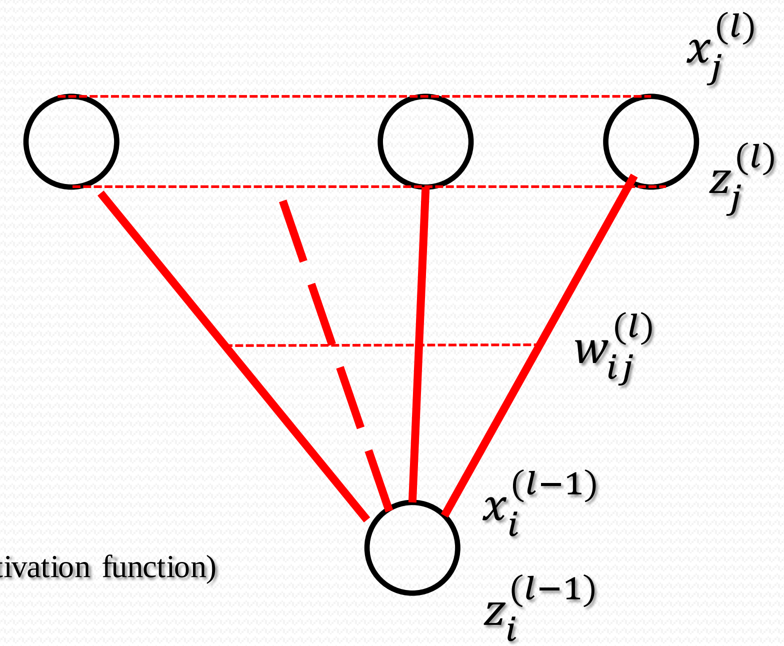
$$\delta_j^{(l-1)} = \sum_j^{d(l)} \frac{\partial e(w)}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial x_i^{(l-1)}} \frac{\partial x_i^{(l-1)}}{\partial z_i^{(l-1)}} \quad \text{(chain rule)}$$

$$\delta_j^{(l-1)} = \sum_j^{d(l)} = \delta_i^{(l)} \quad w_{ij}^{(l)} \quad \phi'(z_i^{(l-1)})$$

$$\delta_j^{(l-1)} = (1 - (x_i^{(l-1)})^2) \quad \sum_j^{d(l)} \delta_i^{(l)} \quad w_{ij}^{(l)} \quad \text{(For tanh-activation function)}$$

$$\frac{\partial e(w)}{\partial \boldsymbol{w}_{\boldsymbol{ij}}^{(l)}} = x_i^{(l-1)} \boldsymbol{\delta_j}^{(\mathbf{l})}$$

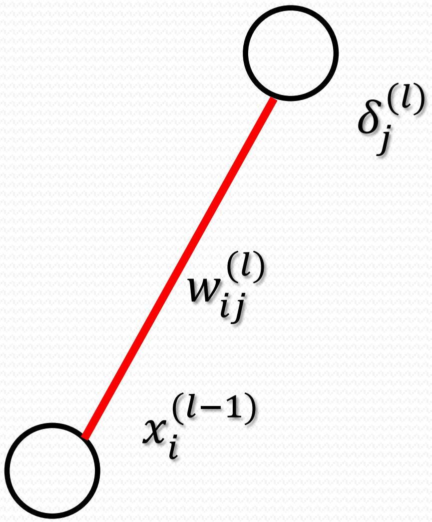
$$\Delta w^{(l)}{}_{ij} = -\eta \frac{\partial e(w)}{\partial \boldsymbol{w}_{\boldsymbol{ij}}^{(l)}}$$



Back Propogation Algorithm

For tanh-activation function

1 - Initialize $w_{ij}^{(l)}$ at random
2 - For t = 0, 1, 2 ………
3 - pick n ∈ $\{1, 2, \ldots\ldots N\}$ (random pickup, i.e. SGD)
4 - Forward compute all $x_j^{(l)}$
5 - Backward compute all $\delta_j^{(l)}$
6 - Update weights $w_{ij}^{(l)} = w_{ij}^{(l)} - x_i^{(l-1)} \delta_j^{(l)}$
7 - Iterate until the stopping criterion is achieved.
8 - Return the final weights $w_{ij}^{(l)}$



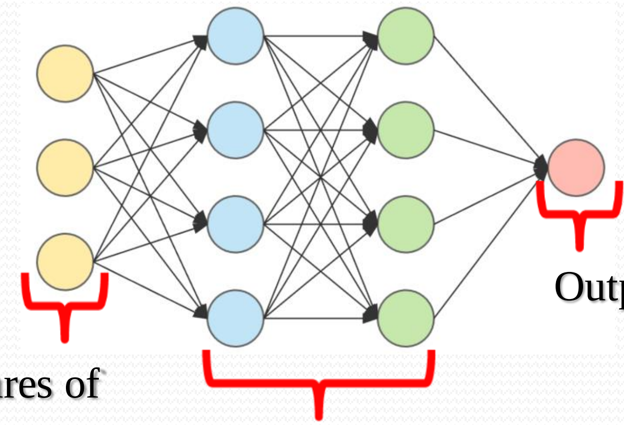
Be careful: Initialize $w_{ij}^{(l)}$ at random and not to zero

If we do so, either $x_j^{(l)}$ or $\delta_j^{(l)}$ will become zero and therefore not useful.

Remark

Neural networks can be thought as Learned Nonlinear Transform. Note that the nonlinear transformation of features (e.g., polynomial, RBF, etc.) are not learned transformation.

Since in the hidden layers the features are higher order features (leaned features), then we can implement a better learning. Indeed the network looks for weight factors for a proper transform the factors that fits data.



Dropout

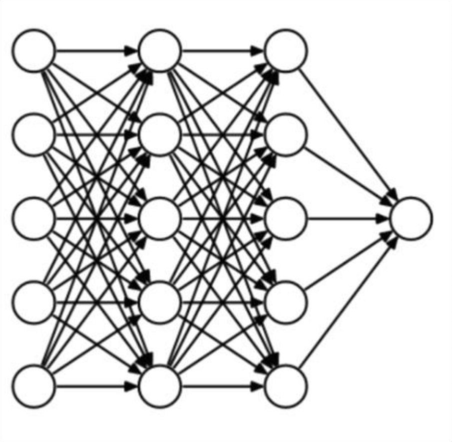
Dropout is a regularization technique for neural network models (Srivastava, et al. , 2014). This is a simple way to prevent neural networks from overfitting.

Some key points:

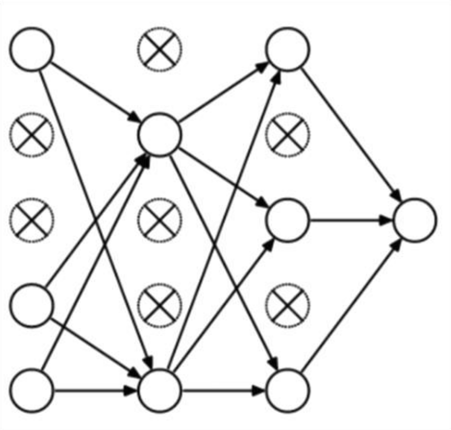
1) Use 20%-50% dropout

2) Dropout with larger network in general provides better performance, giving the model more of an opportunity to learn independent representations.

3) Dropout can be used on visible (input) as well as hidden layers



Standard neural network

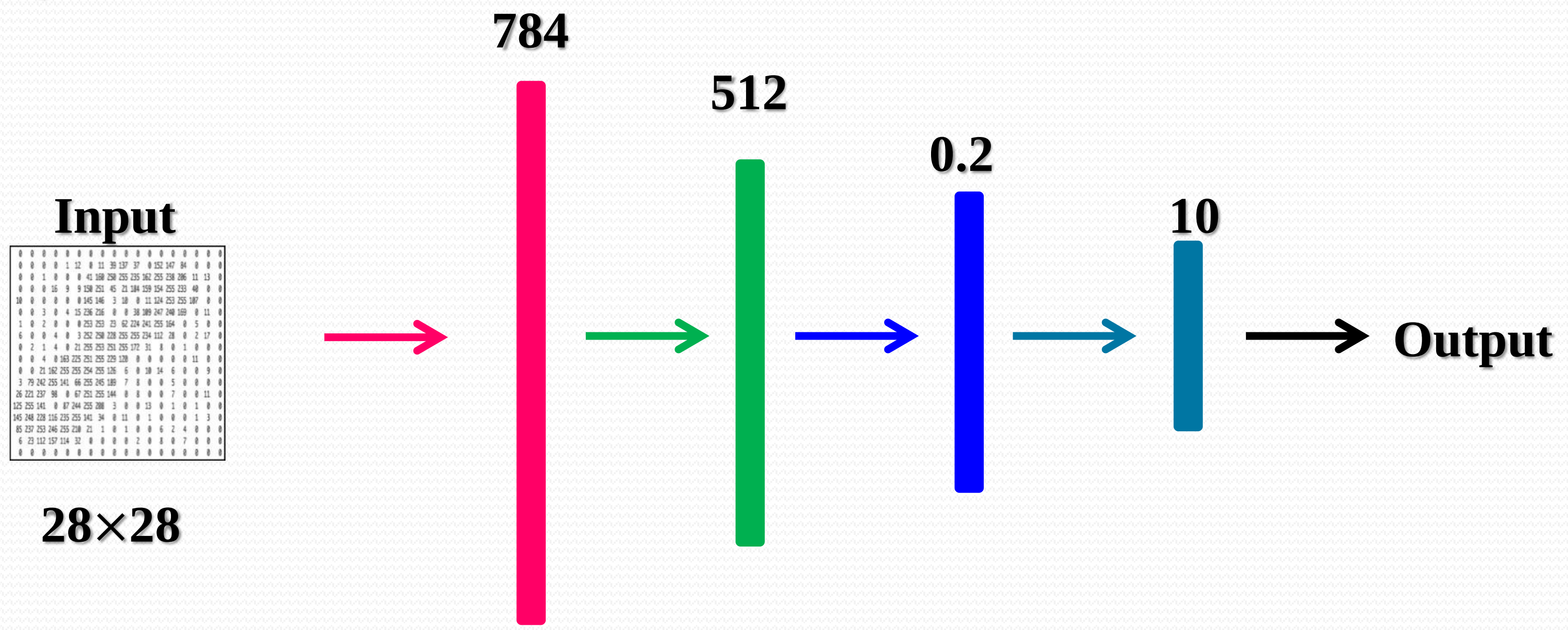


Neural network with dropout

A Simple Network

Example 1: MNIST Database - Handwritten digits

A simple sequential deep learning model for handwritten digits recognition using Keras and TensorFlow,



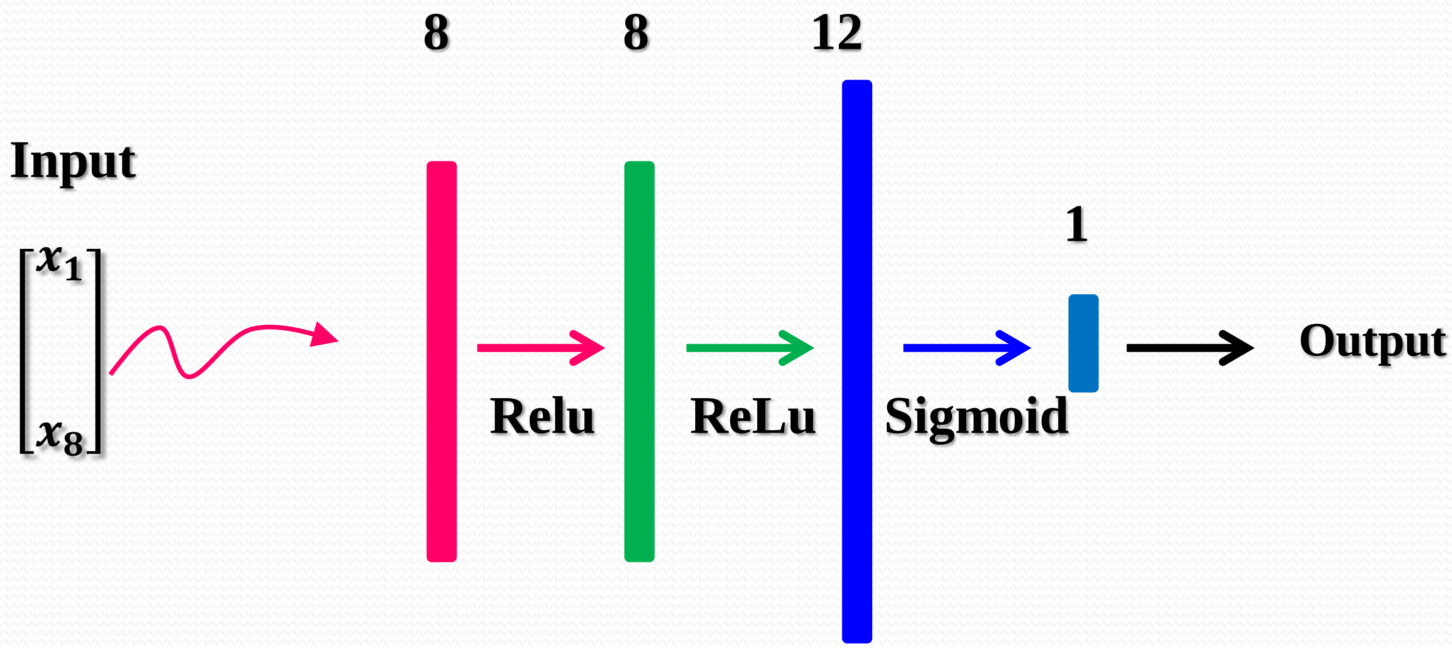

```
tf.keras.layers.Flatten(input_shape=(28, 28))
tf.keras.layers.Dense(512, activation=tf.nn.relu)
tf.keras.layers.Dropout(0.2)
tf.keras.layers.Dense(10, activation=tf.nn.softmax)
```

A Simple Network
MNIST handwritten digits

```
#============================================================ import tensorflow and keras libraries
import tensorflow as tf
mnist = tf.keras.datasets.mnist
import sys
#============================================================ import tensorflow and keras libraries

#============================================================ import MNIST data
# MNIST (Modified National Institute of Standards and Technology database) a large database of handwritten digits
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0     # scale
print('x_train.shape= ', x_train.shape)
print('x_test.shape = ', x_test.shape)
#sys.exit()
#============================================================ import MNIST data

#============================================================ define a sequential model (deep learning) with keras
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),        # flatten data 28x28 --> 784
  tf.keras.layers.Dense(512, activation=tf.nn.relu),    # dense layer =512
  tf.keras.layers.Dropout(0.2),                         # dropout 512*0.2
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)   # output layer = 10
])
#============================================================ define a sequential model (deep learning) with keras

#============================================================ compile the model
'''
Cross entropy loss function: measures the dissimilarity between the distribution of observed class labels
and the predicted probabilities of class lables.
Categorical refers to the possibility of having more than two classes.
Sparse refers to using a single integer from zero to the number of classes minus one (e.g. { 0; 1; or 2 }
for a three-class problem), instead of a dense one-hot encoding of the class label (e.g. { 1,0,0; 0,1,0; or 0,0,1 }
for a class label for the same three-class problem).
'''

model.compile(optimizer='adam',                                 # adam optimizer
              loss='sparse_categorical_crossentropy',           # define the loss function
              metrics=['acc'])                            # define the metric ('accuracy', 'mse', 'msle', 'mae)
#============================================================ compile the model

#============================================================ fit the model
model.fit(x_train, y_train, epochs=10)
#============================================================ fit the model

#============================================================ evaluate the model
print('====================================================test')
model.evaluate(x_test, y_test)
print('====================================================test')
#============================================================ evaluate the model
sys.exit()
```

Example 2: Pima Indians onset of diabetes dataset

A simple sequential deep learning model for predicting handwritten digits using Keras,




```
model.add(Dense(8, input_dim=8, activation='relu'))
model.add(Dense(12, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

A Simple Network
Onset of Diabetes Dataset

```python
#https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/
# Create your first MLP in Keras
#=========================================================
from keras import optimizers
from keras.models import Sequential
from keras.layers import Dense
import keras
import numpy
import sys
#=========================================================
# fix random seed for reproducibility
#numpy.random.seed(7)
#========================================================= import data
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
#========================================================= import data

#========================================================= split data inti test and sample data
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
print('X.shape = ', X.shape)
print('Y.shape = ', Y.shape)
#========================================================= split data inti test and sample data

#========================================================= define a sequential model (deep learning) with keras
# create model
model = Sequential()
model.add(Dense(8, input_dim=8, activation='relu'))
model.add(Dense(12, activation='relu'))
#model.add(keras.layers.Dropout(0.1))
model.add(Dense(1, activation='sigmoid'))
#========================================================= define a sequential model (deep learning) with keras

```

A Simple Network
Onset of Diabetes Dataset

```
#========================================================== compile the model
# Compile model                                                         # set loss function, optimizer, metrics
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

'''
#model.compile(loss='binary_crossentropy', optimizer='sgd', metrics=['accuracy'])
sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])

Adagrad = optimizers.Adagrad(lr=0.01, epsilon=None, decay=0.0)
model.compile(loss='binary_crossentropy', optimizer=Adagrad, metrics=['accuracy'])
'''
#========================================================== compile the model

#========================================================== fit the model
# Fit the model
model.fit(X, Y, epochs=1000, batch_size=10)                    # set batch size and epoch-number

#model.fit(X, Y, epochs=500)
#========================================================== fit the model

#========================================================== evaluate the model
# evaluate the model
scores = model.evaluate(X, Y)
print('======================================= test')
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
print('======================================= test')
#========================================================== evaluate the model
sys.exit()
```