Neural Networks

Collected by Prof. Seungchul Lee
iSystems Design Lab.
http://isystems.unist.ac.kr/
UNIST

Table of Contents

1. Neural Networks

1.1. Recall supervised learning setup

- Input features $x^{(i)} \in \mathbb{R}^n$
- Ouput $y^{(i)}$
- Model parameters $\theta \in \mathbb{R}^k$
- Hypothesis function $h_\theta : \mathbb{R}^n \to y$
- Loss function $\ell : y \times y \to \mathbb{R}_+$

- Machine learning optimization problem

$$\min_\theta \sum_{i=1}^m \ell \left(h_\theta \left(x^{(i)} \right), y^{(i)} \right)$$

  (possibly plus some additional regularization)

- But, many specialized domains required highly engineered special features

1.2. Neural networks

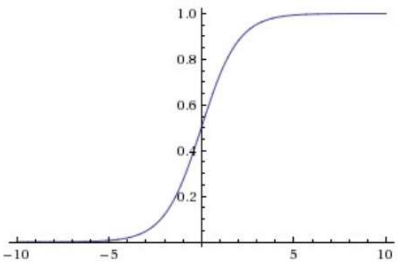
Neural networks are a simply a machine learning algorithm with a more complex hypothesis class, directly incorporating non-linearity (in the parameters)

Example: neural network with one hidden layer

$$h_\theta(x) = \Theta^{(2)} f \left(\Theta^{(1)} x \right)$$

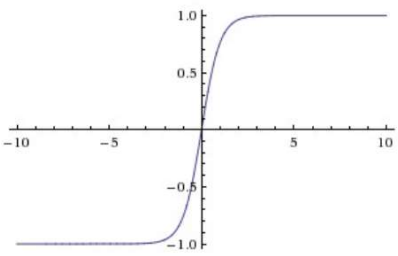
where $\Theta^{(1)} \in \mathbb{R}^{k \times n}$, $\Theta^{(2)} \in \mathbb{R}^{1 \times k}$ and f is some non-linear function applied elementwise to a vector

Sigmoid



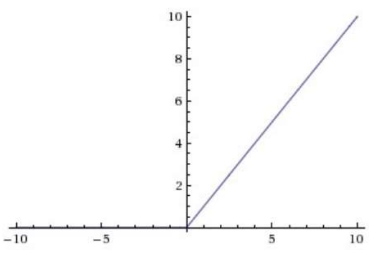
$$g(x) = \frac{1}{1 + e^{-x}}$$

tanh



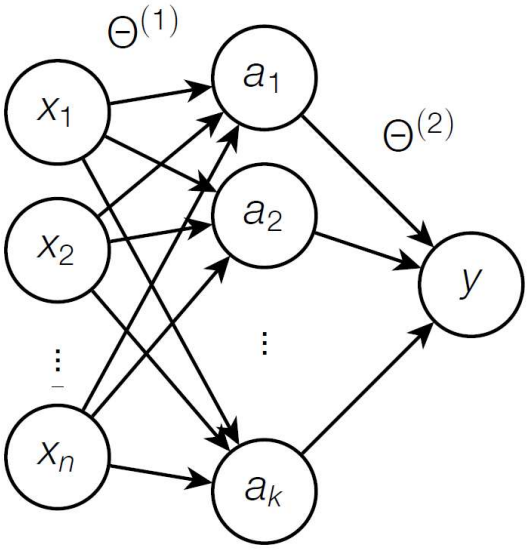
$$g(x) = \tanh(x)$$

Rectified Linear Unit



$$g(x) = \max(0, x)$$

Architectures are often shown graphically

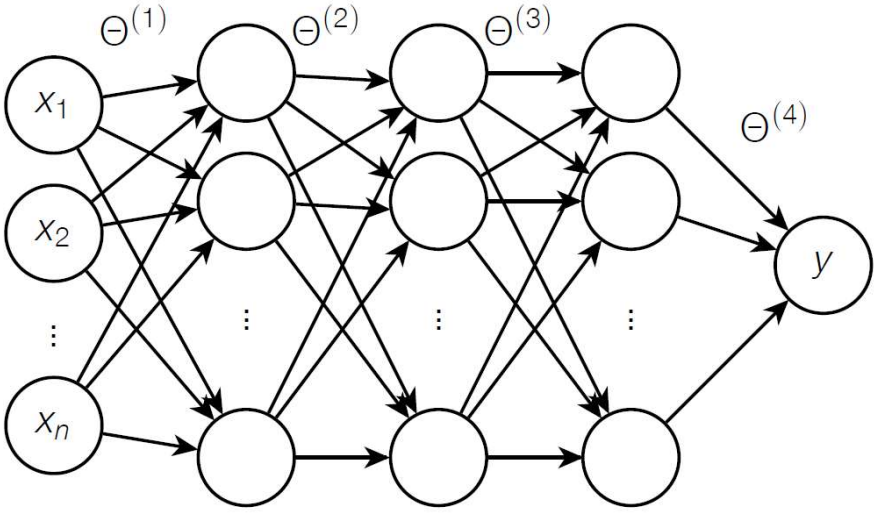


- Middle layer a is referred to as the hidden layer, there is nothing in the data that prescribes what values these should take, left up to the algorithm to decide
- **Viewed another way: neural networks are like classifiers where the features themselves are also learned**

- **Pros**
  - No need to manually engineer good features, just let the neural networks handle this part
- **Cons**
  - Minimizing loss on training data is no longer a convex optimization problem in parameters θ
  - Still need to engineer a good architecture

1.3. Deep learning

"Deep" neural networks typically refer to networks with multiple hidden layers



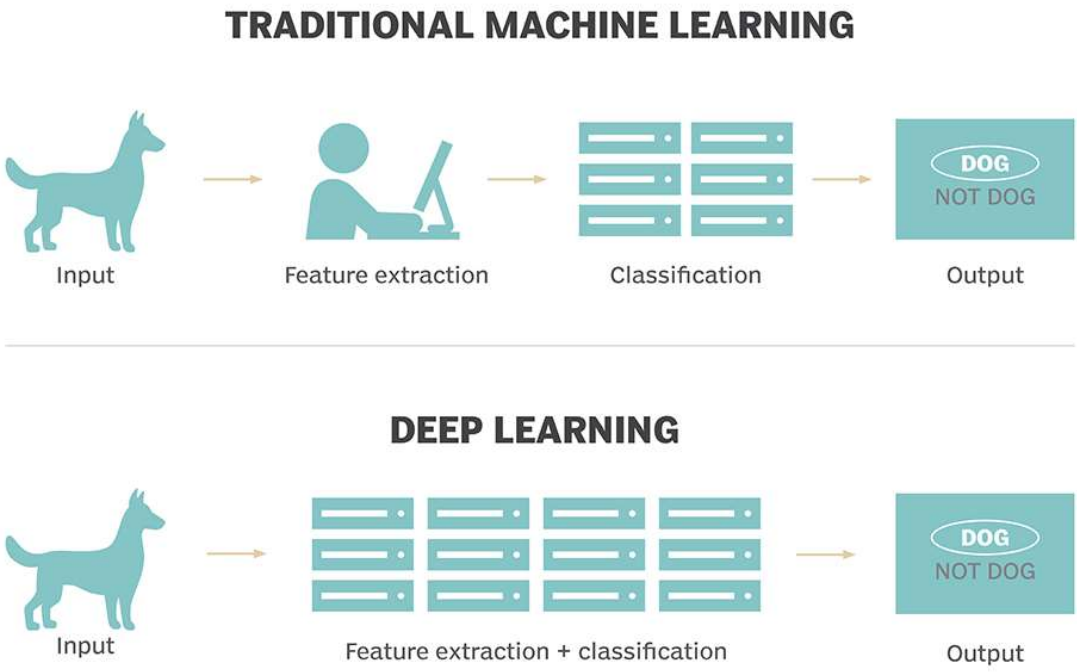
1.4. Machine learning and Neural networks (or Deep learning)

Machine Learning

- Hand-crafted features
- Depends on expertise

Deep Learning

- Automatic feature extraction
- Depends on network structure



2. Structure of Neural Networks

The neuron

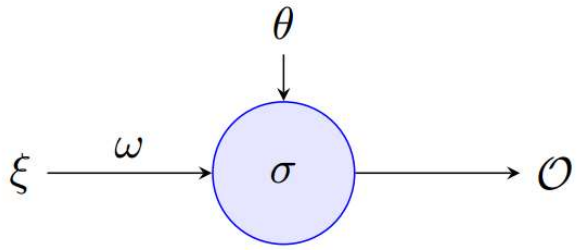
- The sigmoid equation is what is typically used as a transfer function between neurons. It is similar to the step fuction, but is continuous and differentiable.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- One useful property of this transfer function is the simplicity of computing its derivative.

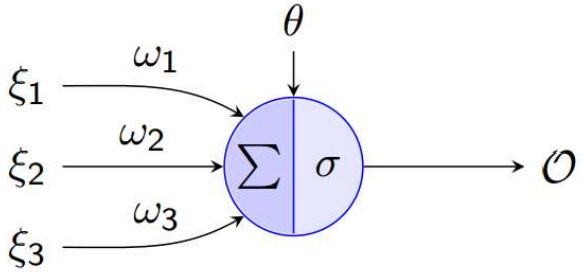
$$\frac{d}{dx}\sigma(x) = \sigma' = \sigma(x)(1 - \sigma(x))$$

Single input neuron



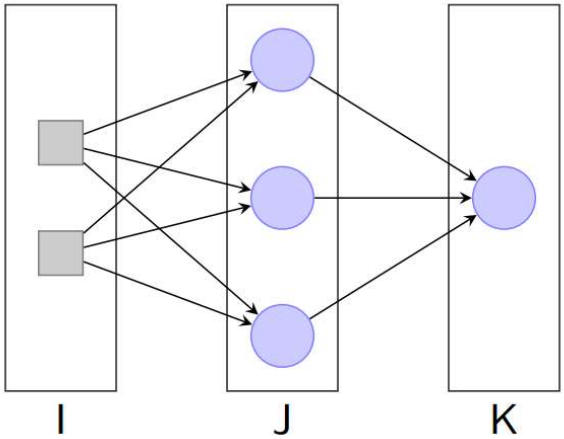
$$O = \sigma(\xi\omega + \theta)$$

Multiple input neuron



$$O = \sigma(\xi_1\omega_1 + \xi_2\omega_2 + \xi_3\omega_3 + \theta)$$

A neural network



3. Learning: Backpropagation Algorithm (Optional)

Notation

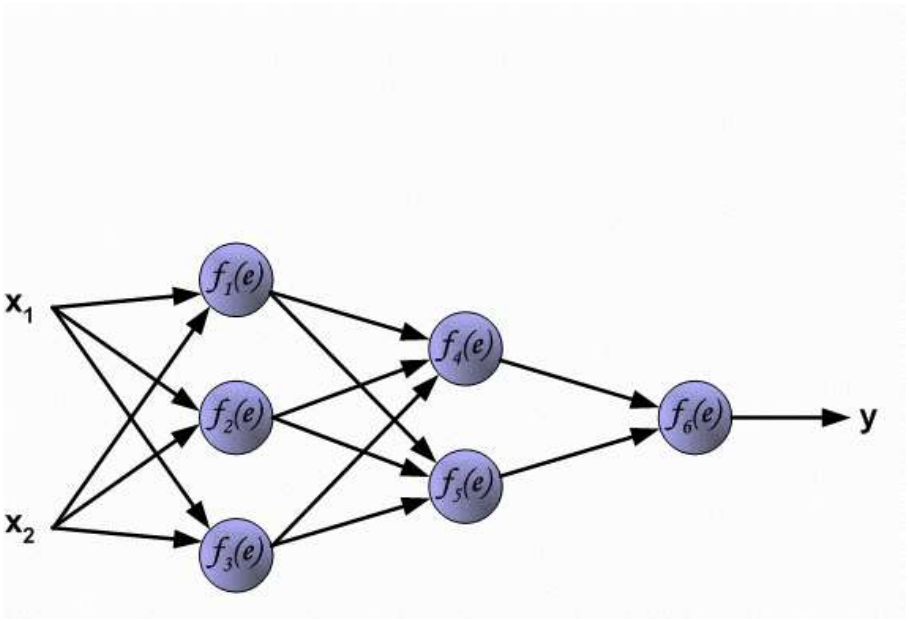
- x_j^ℓ: Input to node j of layer ℓ
- W_{ij}^ℓ: Weight from layer $\ell - 1$ node i to layer ℓ node j
- $\sigma(x) = \frac{1}{1+e^{-x}}$: Sigmoid transfer function
- θ_j^ℓ: Bias of node j of layer ℓ
- O_j^ℓ: Output of node j in layer ℓ
- t_j: Target value of node j of the output layer

The error calculation

Given a set of training data points t_k and output layer output O_k we can write the error as

$$E = \frac{1}{2}\sum_{k \in K}(O_k - t_k)^2$$

- Forward propagation
  - the initial information propagates up to the hidden units at each layer and finally produces output
- Backpropagation
  - allows the information from the cost to flow backwards through the network in order to compute the gradients



We want to calculate $\frac{\partial E}{\partial W_{jk}^\ell}$, the rate of change of the error with respect to the given connective weight, so we can minimize it.

Now we consider two cases: the node is an output node, or it is in a hidden layer

1) Output layer node

$$\begin{aligned}\frac{\partial E}{\partial W_{jk}} &= \frac{\partial}{\partial W_{jk}}\frac{1}{2}(O_k - t_k)^2 = (O_k - t_k)\frac{\partial}{\partial W_{jk}}O_k = (O_k - t_k)\frac{\partial}{\partial W_{jk}}\sigma(x_k) \\ &= (O_k - t_k)\sigma(x_k)(1 - \sigma(x_k))\frac{\partial}{\partial W_{jk}}x_k \\ &= (O_k - t_k)O_k(1 - O_k)O_j\end{aligned}$$

For notation purposes, I will define δ_k to be the expression $(O_k - t_k)O_k(1 - O_k)$, so we can rewrite the equation above as

$$\frac{\partial E}{\partial W_{jk}} = O_j\delta_k$$

2) Hidden layer node

$$\begin{aligned}\frac{\partial E}{\partial W_{ij}} &= \frac{\partial}{\partial W_{ij}}\frac{1}{2}\sum_{k\in K}(O_k - t_k)^2 = \sum_{k\in K}(O_k - t_k)\frac{\partial}{\partial W_{ij}}O_k = \sum_{k\in K}(O_k - t_k)\frac{\partial}{\partial W_{ij}}\sigma(x_k) \\ &= \sum_{k\in K}(O_k - t_k)\sigma(x_k)(1 - \sigma(x_k))\frac{\partial}{\partial W_{ij}}x_k \\ &= \sum_{k\in K}(O_k - t_k)O_k(1 - O_k)\frac{\partial x_k}{\partial O_j}\cdot\frac{\partial O_j}{\partial W_{ij}} = \sum_{k\in K}(O_k - t_k)O_k(1 - O_k)W_{jk}\cdot\frac{\partial O_j}{\partial W_{ij}} \\ &= \frac{\partial O_j}{\partial W_{ij}}\cdot\sum_{k\in K}(O_k - t_k)O_k(1 - O_k)W_{jk} \\ &= O_j(1 - O_j)\frac{\partial x_j}{\partial W_{ij}}\cdot\sum_{k\in K}(O_k - t_k)O_k(1 - O_k)W_{jk} \\ &= O_j(1 - O_j)O_i\cdot\sum_{k\in K}(O_k - t_k)O_k(1 - O_k)W_{jk} \\ &= O_iO_j(1 - O_j)\sum_{k\in K}\delta_kW_{jk}\end{aligned}$$

Similar to before we will now define all terms besides O_i to be $\delta_j = O_j(1 - O_j)\sum_{k\in K}\delta_kW_{jk}$, so we have

$$\frac{\partial E}{\partial W_{ij}} = O_i\delta_j$$

How weights affect errors

- For an output layer node $k \in K$

$$\frac{\partial E}{\partial W_{jk}} = O_j\delta_k$$

  where

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

- For a hidden layer node $j \in J$

$$\frac{\partial E}{\partial W_{ij}} = O_i\delta_j$$

  where

$$\delta_j = O_j(1 - O_j)\sum_{k\in K}\delta_kW_{jk}$$

What about the bias?

If we incorporate the bias term θ into the equation you will find that

$$\frac{\partial O}{\partial \theta} = 1$$

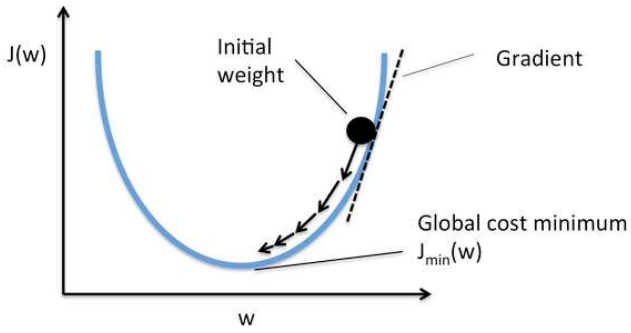
This is why we view the bias term as output from a node which is always one. This holds for any layer ℓ, a substitution into the previous equations gives us that

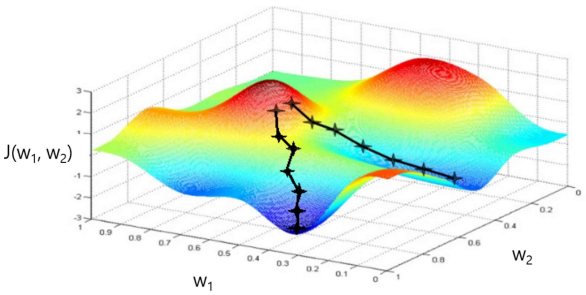
$$\frac{\partial E}{\partial \theta} = \delta_\ell$$

(Stochastic) Gradient Descent

- Negative gradients points directly downhill of cost function
- We can decrease cost by moving in the direction of the negative gradient (η is a learning rate)

$$W := W - \eta \nabla_W \left(h_W \left(x^{(i)} \right), y^{(i)} \right)$$





The backpropagation algorithm using gradient descent

1. Run the network forward with your input data to get the netwrok output
2. For each output node compute

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

3. For eatch hidden node calculate

$$\delta_j = O_j (1 - O_j) \sum_{k \in K} \delta_k W_{jk}$$

4. Update the weights and biases as follows
   Given

$$\Delta W = -\eta \delta_\ell O_{\ell - 1}$$
$$\Delta \theta = -\eta \delta_\ell$$

   apply

$$W \leftarrow W + \Delta W$$
$$\theta \leftarrow \theta + \Delta \theta$$

4. Implementation in Python and numpy (Optional)

For the other neural network guides we will mostly rely on the excellent TensorFlow (https://www.tensorflow.org/)'s optimizations and speed. However, to demonstrate the basics of neural networks, we'll use `numpy` so we can see exactly what's happening every step of the way.

4.1. A basic neural network with numpy

First we'll import numpy:

In [1]:

```
import numpy as np
```

With machine learning we are trying to find a hidden function that describes data that we have. Here we are going to cheat a little and define the function ourselves and then use that to generate data. Then we'll try to "reverse engineer" our data and see if we can recover our original function.

In [2]:

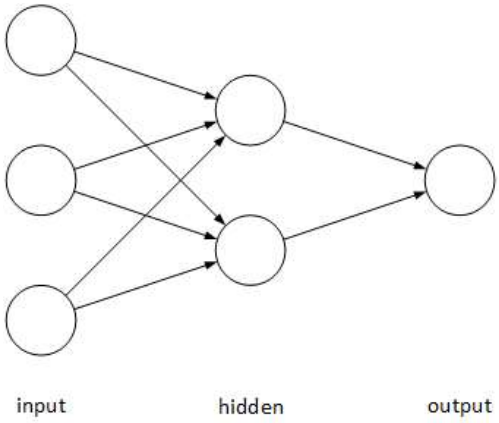
```
def unknown_function(X):
    coeff = np.array([[2., -1., 5.]])
    return np.dot(X, coeff.T)

X = np.array([
    [4.,9.,1.],
    [2.,5.,6.],
    [1.,8.,3.]
])

t = unknown_function(X) # target
print(t)
```

```
[[  4.]
 [ 29.]
 [  9.]]
```

Now we are going to set up our simple neural network. It will have just one hidden layer with two units (which we will refer to as unit 1 and unit 2).



First we have to define the weights (i.e. parameters) of our network.

- We have three inputs each going into two units, then one bias value for each unit, so we have eight parameters for the hidden layer.
- Then we have the output of those two hidden layer units going to the output layer, which has only one unit - this gives us two more parameters, plus one bias value.
- So in total, we have eleven parameters.

Let's set them to arbitrary values for now (random initialization).

In [3]:

```
# initial hidden layer weights
hidden_layer_weights = np.array([
    [0.5, 0.5, 0.5],    # unit 1
    [0.1, 0.1, 0.1]     # unit 2
])
hidden_layer_biases = np.array([1. ,1.])

# initial output layer weights
output_weights = np.array([[1., 1.]])
output_biases = np.array([1.])
```

We'll use $\tanh$ activations for our hidden units, so let's define that real quick:

In [4]:

```
def activation(X):
    return np.tanh(X)
```

$\tanh$ activations are quite common, but you may also encounter sigmoid activations and, more recently, ReLU activations (which output 0 when $x \leq 0$ and output x otherwise). These activation functions have different benefits:

- ReLUs in particular are robust against training difficulties that come when dealing with deeper networks.

To make things clearer later on, we'll also define the linear function that combines a unit's input with its weights:

In [5]:

```
def linear(input, weights, biases):
    return np.dot(input, weights.T) + biases
```

Now we can do a forward pass with our inputs X to see what the predicted outputs are.

4.2. Forward pass

First, we'll pass the input through the hidden layer:

In [6]:

```
hidden_linout = linear(X, hidden_layer_weights, hidden_layer_biases)
hidden_output = activation(hidden_linout)

print('hidden output')
print(hidden_output)
```

```
hidden output
[[ 0.99999977  0.98367486]
 [ 0.99999939  0.9800964 ]
 [ 0.99999834  0.97574313]]
```

(We're keeping the neuron unit's intermediary value, `hidden_linout` for use in backpropagation.)

Then we'll take the hidden layer's output and pass it through the output layer to get our predicted outputs:

In [7]:

```
output_linout = linear(hidden_output, output_weights, output_biases)
output_output = output_linout # no activation function on output layer

predicted = output_output
print('predicted')
print(predicted)
```

```
predicted
[[ 2.98367463]
 [ 2.98009578]
 [ 2.97574147]]
```

Now let's compute the mean squared error of our predictions:

In [8]:

```
mse = np.mean((t - predicted)**2)/2
print('mean squared error')
print(mse)
```

```
mean squared error
119.060003918
```

Now we can take this error and backpropagate it through the network. This will tell us how to update our weights.

4.3. Backpropagation

Since backpropagation is essentially a chain of derivatives (that is used for gradient descent), we'll need the derivative of our activation function, so let's define that first:

In [9]:

```
def activation_deriv(X):
    return 1 - np.tanh(X)**2
```

Then we want to set a learning rate - this is a value from 0 to 1 which affects how large we tweak our parameters by for each training iteration.

- You don't want to set this to be too large or else training will never converge (your parameters might get really big and you'll start seeing a lot of `nan` values).
- You don't want to set this to be too small either, otherwise training will be very slow. There are more sophisticated forms of gradient descent that deal with this, but those are beyond the scope of this guide.

In [10]:

```
learning_rate = 0.001
```

First we'll propagate the error through the output layer (I won't go through the derivation of each step but they are straightforward to work out if you know a bit about derivatives):

In [11]:

```
# derivative of mean squared error
error = predicted - t

# delta for the output layer (no activation on output layer)
delta_output = error

# output layer updates
output_weights_update = delta_output.T.dot(hidden_output)
output_biases_update = delta_output.sum(axis=0)
```

Then through the hidden layer:

In [12]:

```
# push back the delta to the hidden layer
delta_hidden = delta_output*output_weights*activation_deriv(hidden_linout)

# hidden layer updates
hidden_weights_update = delta_hidden.T.dot(X)
hidden_biases_update = delta_hidden.sum(axis=0)
```

Then we can apply the updates:

In [13]:

```
output_weights -= output_weights_update*learning_rate
output_biases -= output_biases_update*learning_rate

hidden_layer_weights -= hidden_weights_update*learning_rate
hidden_layer_biases -= hidden_biases_update*learning_rate
```

That's one training iteration! In reality, you would do this many, many times - feedforward, backpropagate, update weights, then rinse and repeat. That's the basics of a neural network - at least, the "vanilla" kind. There are other more sophisticated kinds (recurrent and convolutional neural networks are two of the most common) that are covered in other guides.

In [14]:

```
print(predicted)
```

```
[[ 2.98367463]
 [ 2.98009578]
 [ 2.97574147]]
```

4.4. Neural Networks in python

In [15]:

```
# hidden layer weights
hidden_layer_weights = np.array([
    [0.5, 0.5, 0.5],    # unit 1
    [0.1, 0.1, 0.1]     # unit 2
])
hidden_layer_biases = np.array([1. ,1.])

# output layer weights
output_weights = np.array([[1., 1.]])
output_biases = np.array([1.])

for i in range(1000):

    hidden_linout = linear(X, hidden_layer_weights, hidden_layer_biases)
    hidden_output = activation(hidden_linout)

    output_linout = linear(hidden_output, output_weights, output_biases)
    output_output = output_linout # no activation function on output layer

    predicted = output_output

    # derivative of mean squared error
    error = predicted - t

    # delta for the output layer (no activation on output layer)
    delta_output = error

    # output layer updates
    output_weights_update = delta_output.T.dot(hidden_output)
    output_biases_update = delta_output.sum(axis = 0)

    # push back the delta to the hidden layer
    delta_hidden = delta_output*output_weights*activation_deriv(hidden_linout)

    # hidden layer updates
    hidden_weights_update = delta_hidden.T.dot(X)
    hidden_biases_update = delta_hidden.sum(axis = 0)

    output_weights -= output_weights_update*learning_rate
    output_biases -= output_biases_update*learning_rate

    hidden_layer_weights -= hidden_weights_update*learning_rate
    hidden_layer_biases -= hidden_biases_update*learning_rate

print(t)
print(predicted)
```

```
[[  4.]
 [ 29.]
 [  9.]]
[[  5.53866817]
 [ 26.56379198]
 [  7.78180535]]
```