Neuron Network

- Timeline

- Artificial neuron
  - McCulloch Pitts Neuron(first one)
    - the inputs (i.e. x n) in here are only 0 or 1 (true or false)
    - Can be used to represent a few Boolean functions

AND function

$y = 1$ if $x_1 + x_2 + x_3 \geq 3$

OR function

$y = 1$ if $x_1 + x_2 + x_3 \geq 1$

    - No learning algorithm to update the neurons
  - The improved artificial neuron (Perceptron)

w_n are the weights,
Θ is the threshold value

- Support real inputs, not just boolean values

$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i \geq \theta$$
$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i < \theta$$

Rewriting the above,

$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i - \theta \geq 0$$
$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i - \theta < 0$$

Single layer perceptron **Multi-layer perceptron**

- could actually learn from past data
  - Why NN can learn
    - by changing weights, the behaviour of neural network can be changed
  - How NN can learn
    - changing weights purposely to reduce the difference between the actual output and the expected output
- Activation / transfer function
  - **hard-limiting function**

$$f(net) = \begin{cases} 0 & \text{if } net < threshold \\ 1 & \text{otherwise} \end{cases}$$

  - **sigmoid function**

$$f(net) = \frac{1}{1+e^{-net}}$$

- A single perceptron will converge only when the problem is linearly separable
- example
  - The activation function employed is a hard-limiting function
    - $I = \Sigma_i x_i w_i$
    - hard-limiting function

$$f(I) = \begin{cases} 0 & \text{if } I < \theta \text{ (threshold)} \\ 1 & \text{otherwise} \end{cases}$$

  - *Perceptron learning algorithm as a formula*

$$\Delta W_i = \alpha\,(T - O)\,X_i = \alpha \times \text{error} \times \text{input-}i$$

$$W_i\,(t+1) = W_i\,(t) + \Delta W_i$$

where $W_i\,(t)$ — the weight at time t (or t-th iteration) ΔW_i — the change made to weight W_i

X_i — the i-th input α — learning step ($0 < \alpha < 1$)

T — target output O — perceptron output

Single perceptron

- $I = \Sigma_i x_i w_i$
- hard-limiting function

$$y = f(I) = \begin{cases} 0 & \text{if } I \le 0 \\ 1 & \text{otherwise} \end{cases}$$

- y = 1 indicate class A
- y = 0 indicate class B
- let initial weights be

  w0 = -0.5, w1 = -1, w2 = 1.5 learning step α = 0.5
- using the 10 given patterns for training (assuming the sequence as appearance)

- for class A: No error for all four patterns

  (-3, 1), (-2, 2), (-1, 2), (-1, 3)
- for class B: No error for five patterns

  (-1,-1), (-2,-2), (-1,-3), (2,-2), (2,1)
- there is an error for the pattern (0, 1) of class B
- learning

  $\Delta W_0 = \alpha\,(T - O)\,X_0 = 0.5 \times (0 - 1) \times 1 = -0.5$

  $\Delta W_1 = \alpha\,(T - O)\,X_1 = 0.5 \times (0 - 1) \times 0 = 0$

  $\Delta W_2 = \alpha\,(T - O)\,X_2 = 0.5 \times (0 - 1) \times 1 = -0.5$
- modified weights

  $W'_0 = -0.5 + (-0.5) = -1$

  $W'_1 = -1$

  $W'_2 = 1.5 + (-0.5) = 1$
- with the new set of weights, all ten patterns are classified correctly (verify by yourself)

Equation for straight line

$x_2 = a \times x_1 + b$

Weighted sum for single perceptron

$w_0 + w_1 \times x_1 + w_2 \times x_2 = 0$

$x_2 = -(w_1/w_2)\,x_1 - (w_0/w_2)$

$-(w_0/w_2)$ — intercept

$-(w_1/w_2)$ — slope

- AI winter
  - The first AI winter(1974)
    - single perceptron cannot do XOR
    - Multiple layers of Perceptron (MLP) can do XOR, but the proposed learning algorithm does not work for that!
    - Solution: BP

        - Backpropagation= a learning algorithm that makes the Neural Network (NN) learn!
    - The second AI winter(By late 80s)
        - already knew that deep neural net was hard to train
        - Deep neural nets trained with backpropagation did not work very well, not as well as nets with fewer layers
        - Support vector machine (came into fashion)
        - Random forests, with lovely mathematical theory behind proved to be effective
        - Why?
            - Labelled datasets were thousands times too small ; deep learning applications generally require a lot of datasets!
            - Computing power was millions times slower back then; computing is now much faster as compared to the past
            - Initialized the weight in stupid ways ; weights are all set to 0 or 1 in the past
            - Used the wrong type of non linearity for activation function
        - Solution: AlexNet
            - Relu

                - The zero value output from ReLU introduces sparsity representation (more zeros in each layer, only the important neurons contribute)
            - Dropout
- Radial Basis Function Networks
    - Architecture

        - Input layer is fully connected to the hidden layer

- The hidden layer is fully connected to the output layer
- Nodes:
  - Each node in the input-layer
  - receives the value of an input variable
- Each hidden node
  - provides a radial basis function of input variables
  - Each node in the output layer
  - corresponds to a non-linear function (mapping) of input variables
- Weights:
  - An adjustable weight vector Wiis on the connections between the input nodes and the i-thinternal node, indicating the centre of the radial basis function (through unsupervised learning)
  - Adjustable weights connecting internal nodes with output nodes (through supervised learning)
- Activation

Hidden layer (kernel nodes)

- *j*-th basis function (kernel) gives the same activation value for all inputs that lie within the same radial distance of its kernel centre c_j
- Kernel nodes often use Gaussian activation basis functions which depend on the distance between the input vector x and the node's centre vector c_i. In this case, the activation value of *i*-th node in hidden layer is calculated by

$$O_i = \exp\left[- \frac{(x - c_i)^T (x - c_i)}{2\sigma_i^2} \right]$$

where σ_i is smoothing parameter (also called normalization factor)

Output unit activation is usually linear combinations of the kernel outputs

$$O_j = \sum_i W_{ji} O_i$$

where W_{ji} is the weight from hidden node *i* to output node *j*.

- Training
  - requires:
    - two parameters (ciand σi) to be found for each kernel node i
    - a full weight set for output nodes
  - two stages:
    - finding centre and smoothing parameter values for the hidden (kernel) nodes (unsupervised learning)
      - Usually Hard C-Means clustering is employed
    - finding weights for the output nodes (supervised learning)
  - Learning for Kernel Centre

$$\mathbf{c}_j = \frac{1}{\mathbf{m}_j} \sum_{\mathbf{x}_i \in \Theta_j} \mathbf{x}_i$$

- If the number of input samples xi is small, set kernel centre ci= xi for each i(a form of self growing network).
- If the number of samples is large, do clustering (e.g. k-means) first and use each cluster prototype as a kernel centre

- Learning for Smoothing Parameter

$$\sigma_j^2 = \frac{1}{m_j} \sum_{\mathbf{x} \in \Theta_j} (\mathbf{x} - \mathbf{c_j})^T (\mathbf{x} - \mathbf{c_j}) \qquad j = 1, 2, \ldots, h$$

  - The smoothing parameters σjare usually found from the average distance between the cluster centres and the training patterns
  - where mj is the number of patterns in cluster Θj

- output layer

Adjust weights by

$$W_{ji}(t+1) = W_{ji}(t) + \Delta W_{ji}$$

where $W_{ji}(t)$ is the weight from hidden node *i* to output node *j* at time *t* (or the *t*-th iteration)

The weight change is computed by

$$\Delta W_{ji} = \eta \delta_j O_i$$

where η is learning rate, O_i is the output at hidden node *i*, δ_j is the error at the output unit *j*:

$$\delta_j = T_j - O_j$$

where T_j is the desired (target) output activation and O_j is the actual output activation at output unit *j*.

Repeat iterations until convergence

- summary
  - There are many variations possible in learning strategies
    - basis functions types
    - selection of centers
      - fixed centersselected randomly
      - self-organized learning (clustering)
      - supervised selection
  - The two-stage training process of RBF permits the use of unlabeledtraining data (unsupervised training methods) while creating kernel nodes and hence, only a relatively small number of labelled data will be needed to find the output layer parameters.
- example

	In1	In2	Out
P1:	0	0	0
P2:	0	1	1
P3:	1	0	1
P4:	1	1	0

For simplicity, the following simple radial basis function is used:

$$O_i = \begin{cases} 1-(x-c_i)^T(x-c_i) & when \quad 1-(x-c_i)^T(x-c_i) \geq 0 \\ 0 & otherwise \end{cases}$$

where c_i is the weight vector of hidden unit Ci (i = 1, 2, 3, 4), x is an input vector (the simple function requires NO smoothing parameter).

The activation of output node is a linear combination of the kernel outputs: Out = $\Sigma_i W_{oi} O_i$

The initial weights of the output layer are:

$W_{o1} = 0.5$, $W_{o2} = 0.5$, $W_{o3} = 0.5$, and $W_{o4} = 0.5$.

The weight adjustment is based on:

$$W_{oi}(t+1) = W_{oi}(t) + \Delta W_{oi}$$

$$\Delta W_{oi} = \eta(T_{out} - A_{out})O_i$$

where $\eta = 0.5$ is the learning rate for output layer, T_{out} is the target output, A_{out} is the actual output of the output node, O_i is the output of kernel node *i*.

Weights for kernel nodes:

W11 = 0, W12 = 0,

W21 = 0, W22 = 1,

W31 = 1, W32 = 0,

W41 = 1, W42 = 1.

Weights for output nodes:

- **Input P1:**
  - **only C1 is activated. Out = 0.5**
  - **Wo1(t+1) = Wo1(t) + ΔWo1 = 0.5 + 0.5 * (0 – 0.5) * 1 = 0.25**
- **Input P2:**
  - **only C2 is activated. Out = 0.5**
  - **Wo2(t+1) = Wo2(t) + ΔWo2 = 0.5 + 0.5 * (1 – 0.5) * 1 = 0.75**

- General Regression Neural Networks (GRNN)
  - Architecture

    - an input-layer
    - three computational layers: pattern, summation, output
  - Nodes & connections

- The input-layer distributes input patterns to the pattern-layer through fully connected links with adjustable weights.
- The pattern-layer is fully connected to the summation-layer units through adjustable weights.
- Each pattern node computes the distance between the input vector and their connection weight vector values.
- The summation-layer has two types of neurons, type A and type B. They perform a linear summation of weighted input from the pattern layer.
- The output layer performs a division operation on the output of the two summation-layer units to produce the estimate of the regression of z on x. One output node needs two corresponding summation nodes (each of type A and B).

- Activation

From mathematical background, after certain rearrangement (details omitted) to the calculation of the regression of z on x, we can obtain

$$\hat{z} = \frac{\sum_{i=1}^{P} A_i \exp\left(\frac{-D_i^2}{2\sigma^2}\right)}{\sum_{i=1}^{P} B_i \exp\left(\frac{-D_i^2}{2\sigma^2}\right)}$$

where $D_i^2 = (x - w_i)^T(x - w_i)$, P is the number of pattern nodes

Pattern-layer unit *i* computes the distance between its weight vector and the input vector $(x - w_i)$ and transforms this to a scalar activation function value (typically exponential functions - Gaussian).

$$O_i = \exp\left(\frac{-\sum_{j=1}^{n}(x_j - w_{ij})^2}{2\sigma^2}\right)$$

The activation of each summation unit is a linear sum of weighted inputs from the pattern layer

The output layer performs a division operation on the output of the two summation-layer units (type A and B) to produce the regression of z on x

- training
  - In the pattern layer, a new neuron is created for each exemplar pattern (or cluster centre) and the weight values are set equal to the exemplars or cluster centres (centroid values).
  - For the weights in summation-layer, Aiand Biare increased each time a training observation zjfor cluster iis seen (after kobservations):
    - Ai(k) = Ai(k-1) + zj
    - Bi(k) = Bi(k-1) + 1
    - where Ai(0) = 0, Bi(0) = 0
  - σ, determined experimentally, is a smoothing constant that defines the decision surface boundary.
- GRNN Mapping Surface and Smoothing Constant
  - Small values of σgive narrow peaked surfaces that fit well near sample points.
  - Larger values of σresult in flatter, smoother surfaces.
  - Good generalization requires a trade-off between the two extremes.
- Summary
  - Features:
    - Memory-based
    - One-pass learning algorithm

        - A form of self-growing net with one pattern unit created for each new training pattern or cluster centre.
    - Applications:
        - Good for forecasting, control and similar applications where the training data set is generated by an unknown probability distribution.
- Pros & Cons
    - Advantages:
        - The estimate converges to the true regression surface with increasing samples.
        - A GRNN has the ability to work with sparse data and in real-time environments since the regression surface is defined everywhere instantly and the network provides smooth transitions from one observation to another.
        - Training a GRNN is fast and straightforward with only a single pass through the training set needed. The network can begin to perform the regression after a single training sample has been presented.
    - Disadvantages:
        - The net may grow to be very large since one pattern unit is added for each pattern in the training set(unless clustering is used to determine the prototype centres).
        - The computation load for the network is relatively heavy
- Self-Organizing Map (SOM, Kohonen) Network
    - Neural Network for clustering

    - common strategy: winner-takes-all (a competitive learning)
    - Basic architecture of a clustering network:
    - two layers:input layer and output layer
    - Competitive Network Operation

COMPETITIVE "WINNER-TAKES-ALL" NETWROK

- Signals feed forward from input nodes and feed lateral among output neurons, a form of recurrent feedback
- A single-layer network architecture with n inputs and m output units, one output for each class or category
- m prototype weight vectors wi, i= 1, 2, ..., m correspond to the m classes. The network classifies each input pattern x as belonging to class i if
  - |wi-x| ≤|wk-x|k=1, 2, ..., m, k≠i(find the most similar class)

- SOM network
  - Feature mapping converts patterns of arbitrary dimension (the pattern space) into the response of one-or two-dimensional arrays of neurons (the feature space).
  - Consists of a group of geometrically organized neurons in one-or two-dimensions or even higher dimensions.
    - A one-dimensional network is a single layer of units arranged in a row

NEURONS COMPETE AND ONLY ONE "WINS"

    - Input pattern vector fully connected to all neurons
    - Lateral interactions between neurons constrain activations to spatially bounded 'excitation zone'
    - In two-dimensional network, the units are arranged as a lattice array.

- Input vector patterns x are fully distributed to each node in the discrete lattice through adjustable weight vectors wr
- The node with weight vector closest to the input vector becomes the "excitation centre" winner for the lattice

- A form of unsupervised competitive learning where winner shares the gains with neighbours to produce activation zone

- Kohonen Learning Algorithm
  - Kohonen learning rule is one approach to design a self-organizing feature map network

1) Initialize all weights w_r to random numbers following uniform distribution (0,1)
2) Apply an input signal vector x to the network
3) Select the winning output unit as the one with the smallest dissimilarity measure (or largest similarity measure) among all weight vectors w_r and the input vector x

$$\|\mathbf{x}-\mathbf{w}_r\| \le \min_{r'} \|\mathbf{x}-\mathbf{w}_{r'}\|$$

4) Update

the weights of the winner unit:

$$\mathbf{w}_r^{new} = \mathbf{w}_r^{old} + \alpha \cdot h_{rr'} \cdot \left(\mathbf{x} - \mathbf{w}_r^{old}\right)$$

the neighbouring

$$\mathbf{w}_{r'}^{new} = \mathbf{w}_{r'}^{old} + \alpha \cdot h_{rr'} \cdot \left(\mathbf{x} - \mathbf{w}_{r'}^{old}\right)$$

where $h_{rr'}$ is a neighborhood function with maximum value centered at the winning neuron *r* and decreases to zero as the distance between *r* and neighboring units r' increase (i.e. $h_{rr'}$ is defined in terms of the distance between r and r').

5) Repeat steps 2 through 4 until the network weights stabilize
6) Reduce the neighbourhood and learning rate parameters and iterate steps 2 to 5 if necessary.

Activation Neighbourhood can be:

- **Square**

- **Hexagon**

- **Smooth functions such as Gaussian or other types may also be used.**

- SOM Mapping Operation

- Mapping the input signal space onto the NN lattice
- Weight vectors of neurons in the neighborhoodof the winning neuron s, are shifted towards the input vector value. Those nearer to shave weights shifted more and those farther away shifted less

- Simulations of SOM

(a) input data uniformly distributed within [0, 1]× [0, 1]
(b) initial weights
(c) weights after 30 iterations
(d) weights after 1000 iterations

(a) input data uniformly distributed within [0, 1]× [0, 1]
(b) initial weights
(c) weights after 30 iterations
(d) weights after 1000 iterations

- Mapping two inputs x1 and x2 on to a SOM array.
- Input patterns are chosen randomly from the unit square
- Line intersections on the maps specify weight vector values for a single neuron
- Lines between nodes on the graph merely connect weight points for neurons that are topologically nearest neighbors

- Pros & Cons
  - Advantages of SOM
    - It is viable alternative to C-Means clustering

- It does so more elegantly, by removing the main drawback of C-Means - specifying the number of clusters upfront
- It provides a method of data compression
- When used for labeled data, this acts as a classifier too.
- Most importantly, it provides the user with much wanted ‘data visualization’feature, which is absent in many of the competing algorithms

- Application
  - Optimization
    - One-dimensional SOM networks have been used to successfully solve optimization problems such as the traveling salesman problem (TSP)
  - Control
    - Robot arm control
      - as a part of a robot control system, SOM network performs “data fusion” to provide the input data to the robot

- CNN: Convolutional neural network
  - Components
    - Back Propagation Algorithm

Total error over all training patterns p and m output units:

$$E_{tot} = \sum_{p=1}^{P} E^p \qquad \text{Total error, all training patterns}$$

$$E^p = \frac{1}{2}\sum_{k=1}^{m}(t_k - z_k)^2 \qquad \text{Error for pattern p}$$

$$z_k = f(I_k) = f(\sum_j w_{kj} y_j) = f(\sum_j w_{kj} f(H_j)) =$$
$$= f(\sum_j w_{kj} f(\sum_i v_{ji} x_i))$$

z_k — computed output

t_k — target output

For output units k = 1, 2, ..., m, adjust the weights to reduce the error for each pattern p (Gradient descent)

$$\Delta w_{kj} = -\eta \frac{\partial E^p}{\partial w_{kj}} = -\frac{\eta}{2}\sum_{k=1}^{m}\frac{\partial (t_k^p - z_k^p)^2}{\partial w_{kj}}$$

Dropping superscripts p

$$\frac{\partial (t_k - z_k)^2}{\partial w_{kj}} = \frac{\partial (t_k - f(I_k))^2}{\partial I_k}\frac{\partial I_k}{\partial w_{kj}} = -2(t_k - z_k) f'(I_k) y_j$$

So

$$\Delta w_{kj} = -\eta \frac{\partial E^p}{\partial w_{kj}} = -\frac{\eta}{2}\sum_{k=1}^{m}\frac{\partial (t_k^p - z_k^p)^2}{\partial w_{kj}}$$
$$= \eta (t_k - z_k) f'(I_k) y_j = \eta \delta_k y_j$$

For hidden units, use the chain rule, get

$$\Delta v_{ji} = \eta f'(H_j) x_i \sum_k \delta_k w_{kj}$$

    - BP is never assured of finding a global minimum.

- Pipeline
  - 1.Initialize the weights to small random numbers
  - 2. Randomly select a training pattern pair (xp, tp) and present the input pattern xpto the network. Compute the corresponding network output pattern zp
  - 3. Compute the error Ep for pattern (xp, tp)
  - 4. Backpropagatethe errors according to the BP weight adjustment formulas (given later)
  - 5. Test the mean square error (MSE) over P training patterns:

$$E = \frac{1}{P}\sum_{p=1}^{P} E^{p}$$

    - If the MSE is below the required threshold, stop. Otherwise, repeat steps 2-5.
  - 6. Test for generalization performance if appropriate
- Summary

Weights initialization

Set all weights to random numbers following Uniform distribution in the range (-1,1)

Calculation of activation

- the activation level of an input unit is determined by the instance presented to the network
- the activation level O_j of a hidden and output unit is determined by

  $O_j = f(\sum W_{ji}O_i - \theta_j)$, where W_{ji} is the weight from an input O_i, θ_j is the node threshold, f is the transfer function.

Weight Updating

1) start at the output nodes and work backward to the hidden layers recursively, adjust weights by
   $W_{ji}^{(t+1)} = W_{ji}^{(t)} + \Delta W_{ji}$
   where $W_{ji}^{(t)}$ is the weight from unit *i* to *j* at time t (or *t*-th iteration), ΔW_{ji} is the weight adjustment

   (Assume a sigmoid function $f(a) = 1/[1 + e^{-a}]$ is used)

- **Weight Updating (cont.)**

  2) The weight change is computed by $\Delta W_{ji} = \eta\delta_j O_i$
     where η is a trial-independent learning rate ($0 < \eta < 1$)

  3) 3a) for the output units: $\delta_j = O_j(1 - O_j)(T_j - O_j)$
     where T_j is the desired (target) output activation and O_j is the actual output activation at output unit *j*
     3b) for the hidden units: $\delta_j = O_j(1 - O_j)\sum_k \delta_k W_{kj}$
     where δ_k is the error term at unit k to which a connection points from hidden unit *j*

- Repeat iterations until convergence in terms of the selected error criterion. An iteration includes presenting an instance, calculating activations, and modifying weights.

- example

- Initial weights

  W_{12} = -0.02, W_{13} = 0.02, W_{14} = 0.03, W_{1b} = -0.01, W_{23} = 0.01, W_{24} = 0.02, W_{2b} = -0.01

Input	Output	Input	Output
(1, 1)	0	(0, 1)	1
(0, 0)	0	(1, 0)	1

- Calculation of activation (assume b=1.0 and sigmoid activation function used)

  O3 = O4 = 1

  $O2 = 1/[1+e^{-(1\times W23+1\times W24+1\times W2b)}] = 1/[1+e^{-(1\times 0.01+1\times 0.02-1\times 0.01)}] = 0.505$

  $O1 = 1/[1+e^{-(0.505\times W12+1\times W13+1\times W14+1\times W1b)}] = 1/[1+e^{-(0.505\times(-0.02)+1\times 0.02+1\times 0.03-1\times 0.01)}] = 0.508$

- Weight training (assume that the learning rate η = 0.3)

  $\delta_1 = 0.508(1-0.508)(0-0.508) = -0.127$ $\Delta W_{13} = 0.3\times(-0.127)\times 1 = -0.038$

  $\delta_2 = 0.505(1-0.505)[(-0.127)\times(-0.02)] = 0.0006$ $\Delta W_{23} = 0.3\times 0.0006\times 1 = 0.00018$ (omit the rest)

- After many iterations, a set of final weights give the mean squared error of less than 0.01:

  W_{12} = -11.30, W_{13} = 4.98, W_{14} = 4.98, W_{1b} = -2.16, W_{23} = 5.62, W_{24} = 5.62, W_{2b} = -8.83

- Convolution
  - type
    - Conv1D is used for input signals which are similar to the voice

Input				Kernel				Intermediate Output	Output	
1	0	1	0	0	1	0	1	7	7	7
1	1	3	2	0	0	2	0	6	6	13
1	1	0	1	0	1	0	0	8	8	10
2	3	2	1							
0	2	0	1							
1	0	1	0	2	1	0	0	7		
1	1	3	2	0	0	0	1	13		
1	1	0	1	0	3	0	0	10		
2	3	2	1							
0	2	0	1							

    - size

$$M_r = \left\lfloor \frac{W_r + P_r - F_r}{S_r} \right\rfloor + 1$$

$$M_c = N_F$$

- M_r, M_c: Number of rows and columns in the output, respectively
- W_r: Number of rows in the input
- F_r: Filter size (along the rows)
- P_r: Amount of padding (along the rows)
- S_r,: Stride (along the rows)
- N_F,: The number of filters

- For each filter, the number of parameters is

$$W_c \times F_r + 1$$

- Thus, for N_F amount of filters, the number of parameters is

$$N_F \times (W_c \times F_r + 1)$$

- 2D convolution(images)

- size

$$\underbrace{M_r}_{\text{row}} = \left\lfloor \frac{\overbrace{W_r}^{\text{input size}} + \overbrace{P_r}^{\text{padding size}} - \overbrace{F_r}^{\text{filter size}}}{\underbrace{S_r}_{\text{stride length}}} \right\rfloor + 1$$

$$\underbrace{M_c}_{\text{column}} = \left\lfloor \frac{W_c + P_c - F_c}{S_c} \right\rfloor + 1$$

Assume we have an input of size 128 x 128 (row x column) going into a 2D convolution layer

The filter / kernel size for the layer is 7 x 7, and the stride is 2 x 2, no padding is applied, calculate the output size of the 2D convolution

$$M_r = M_c = \left\lfloor \frac{128 + 0 - 7}{2} \right\rfloor + 1$$

$$= \left\lfloor \frac{121}{2} \right\rfloor + 1$$

$$= 61$$

- multi-size -> add

- Conv3D is usually used for videos where you have a frame for each time span
-

- padding

- let ouput size=input size

- Stride -> control output size

- Kernel values are updated by the algorithms based on the training data
- We decide only the size of the Kernels
- Convolutional layers extract features
  - big Kernels extract obvious features,
  - small ones extract more detailed features

- Pooling

- Max Pooling layers help to downscale the feature maps so that only features with significant weights are brought over to the next layer
- size

Output size — input size — filter size

$$M_r = \left\lfloor \frac{W_r - F_r}{S_r} \right\rfloor + 1$$

row — stride length

$$M_c = \left\lfloor \frac{W_c - F_c}{S_c} \right\rfloor + 1$$

column

- Assume we have an input of size 61 x 61 (row x column) going into a 2D pooling layer
- The kernel size for the layer is 4 x 4, and the stride is 2 x 2, calculate the output size of the 2D pooling layer

$$M_r = M_c = \left\lfloor \frac{61-4}{2} \right\rfloor + 1$$

$$= \left\lfloor \frac{57}{2} \right\rfloor + 1$$

$$= \lfloor 28.5 \rfloor + 1 = 29$$

- Activation

Name	Plot	Equation	Derivative (with respect to x)
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a. Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Softsign [7][8]		$f(x) = \frac{x}{1+\lvert x \rvert}$	$f'(x) = \frac{1}{(1+\lvert x \rvert)^2}$

- discrete (e.g., 0 and 1, when the activation function is hard-limiting)
- continuous across a range (e.g., [0, 1] when a sigmoid function is used for activation function)
- Method
  - sigmoid

            - Gradient vanish and poor learning as a consequence
            - hard to update earlier layers, many static or dead neurons
        - Relu(f(x) = max(0, x))
            - Good for update and prevent gradient vanish
    - Regularization L1,L2

To regularize the cost function so that the model parameters will not be tuned to fit the training data too well .

L2 regularization $$C = C_0 + \frac{\lambda}{2n}\sum_w w^2$$

L1 regularization $$C = C_0 + \frac{\lambda}{n}\sum_w |w|$$

        - Add a penalty to loss function; the penalty should be proportional to the magnitude of the weight
        - L2

$$w \leftarrow (1 - \underbrace{\alpha\eta}_{\text{weight decay}})w - \eta\frac{\partial J}{\partial w}$$

            - L2 regularization is more often used; it calculates the sum of squared values of the weights
            - L2 approach also called 'weight decay' in the field of neural networks, or 'shrinkage' in statistics
        - L1
            - L1 calculates the sum of the absolute values of weights
            - No clean algebraic solution and can be computational inefficient
            - Encourage weights to be 0.0 if possible. Consequence: sparse weights (weights with more zero values)
- Weight
    - Method
        - It has been shown that setting the initial weights to small (but not too small) random values, say between -1 and +1, is an effective way to avoid shallow troughs and possible entrapment at the start of the training process.
        - Estimate values that are near a minimum solution rather than assigning random values
        - LeCun suggested to draw values from a zero-mean Gaussian distribution with standard deviation 1/sqrt(N), N is the number of connections feeding into the node.
        - Xavier initialization

- Initialize biases to be 0, the weight of each layer is

$$w \sim U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$$

- n is the number of inputs to a neuron in the layer
- U is uniform distribution with a specified interval
- The weight is drawn randomly from the distribution

- designed for Sigmoid and tanh function

- He initializaion

- Initialize biases to be 0, the weight of each layer is

$$w \sim N\left(0, \frac{2}{n}\right)$$

- n is the number of inputs to a neuron in the layer
- N is a normal distribution with a zero mean and a variance of 2/n

- Designed for ReLU
- In actual application, the normal distribution is truncated to avoid unnecessary large values

- disaster of bad initialization
  - Can't set all to 0, nothing will be learned in this case
  - If weights are too small, signal shrinks as it passes through each layer; at later stage, neurons are 'dead', almost no activation coming out from neurons

        - If weights are too large, signal grows massive as it passes through each layer; at later stage, neurons are 'saturated', network becomes unstable
- Dropout

    - Very popular method to regularize neural networks ; effective in preventing overfitting
    - Concept:
        - the zeros will be applied to different neurons in different training epoch
        - only during training phase, not in testing / validation
        - Approximates training a large number of neural networks with different architectures in parallel
        - Every unit of the neural network (except output layer) is given the probability p of being temporarily ignored/muted (i.e. “dropped out”) in calculations
        - Hyper parameter p is called dropout rate and very often its default value is set to 0.5
        - In each iteration, the neurons are randomly selected according to the assigned probability. As a result, each time we work with a smaller neural network
- Loss function
    - MSE

$$\mathbf{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \tilde{y}_i)^2$$

    - cross-entropy cost function

$$C = -\frac{1}{n} \sum_x \sum_j \left[y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L) \right]$$

- Batch normalization

Batch normalization
The solution

- Let v be the output from a neuron; we denote v_i as one of the neuron outputs from an input in a batch

The batch size is m	$B = [v_1, v_2, \cdots, v_i, \cdots, v_m]$
The mini-match mean for v	$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} v_i$
The mini-match variance for v	$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (v_i - \mu_B)^2$
Perform normalization on every item	$\hat{v}_i \leftarrow \frac{v_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$
Scale and shift the output, γ and β are trainable parameters; u_i is the normalized output	$u_i \leftarrow \gamma \hat{v}_i + \beta$

- For each value in the vector, we perform yi =(xi − mean)/standard deviation
- re-scales data so that it has a zero mean and a standard deviation of 1
- Batch normalization is performed channel by channel.All the elements in each channel form the vi and normalized accordingly
- performed before activation

- Learning Rule

- Brief
  - determine how to adapt connection weights in order to optimise the network performance
  - indicate how to calculate the weight adjustment during each training cycle
  - is suspended after training is completed
- Rule of Gradient Descent

Stochastic GD (SGD): Just use one single training example with each iteration and feedback error directly.

Batch GD: Loop over all training examples, accumulate errors and then feedback.

Mini Batch GD: Process **n** training examples at once. This is a good choice for very large data sets.

- momentum

  Adding a momentum term to the weight update rule

  - It has the effect of smoothing the descent down the error curve by adding an averaging of the weight adjustments (exponential smoothing). This can prevent overshooting off the minimum.

    $$\Delta w_{ji}(t+1) = -\eta \frac{\partial E}{\partial w_{ji}(t)} + \alpha \Delta w_{ji}(t)$$

    where α is the momentum coefficient ($0 < \alpha < 1$).
  - Separate momentum terms can be added to each set of the layer weights using different values of α for each layer to achieve even better results.

- adam

  $$m = 0, \quad v = 0 \qquad \text{Initialization}$$

  $$m = \beta_1 \cdot m + (1 - \beta_1) \cdot dw$$

  $$v = \beta_2 \cdot v + (1 - \beta_2) \cdot dw^2$$

  $$\hat{m} = \frac{m}{1 - \beta_1}$$

  $$\hat{v} = \frac{v}{1 - \beta_2}$$

  $$w = w - \alpha \cdot \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$$

- learning rate decay
  - it determines the size of the weight adjustment made at each iteration and hence influences the rate of convergence. It should not be constant throughout the learning process for best results.
  - step decay

    decay learning rate by half every few epochs.

  - 1/t decay

    $$\eta = \eta_0 / (1 + kt)$$

  - exponential decay

$$\eta = \eta_0 * e^{-kt}$$

- Overfit
  - problem

    Generalization is the ability of a network to correctly classify a pattern it has not seen (not been trained on). NNs generalize when they recall full patterns from partial or noisy input patterns, when they recognize patterns not previously trained on or when they predict new outcomes from past behaviors.

    Networks can be *overtrained*. It means that they memorize the training set and are unable to generalize well.

  - solution

    Overtraining can usually be avoided by:

    - choosing a large training set when possible
    - selecting the patterns randomly during training
    - stopping the training process before excessive training occurs (monitoring the training process and stopping after the error drops to a fairly low level)
    - introducing noise directly into the training patterns
    - eliminate unnecessary hidden-layer nodes & weights
    - Adding regularization terms such as L2 regularization
    - using a combination of the above
    - Using a global optimization based training algorithm, e.g. genetic algorithm (GA)

    Generalization is influenced by the size of training set, the architecture of the network, and the physical complexity of the problem at hand. In practice, we need the size of the training set, *N*, satisfy the condition

$$N = O\left(\frac{W}{\varepsilon}\right)$$

    where *W* is the total number of free parameters (i.e., weights and biases) in the network, ε denotes the fraction of error rate permitted on the test set, *O* denotes the order of quantity enclosed within. Eg., with an error of 10%, *N* should be about 10 times the number of free parameters in the network.

    - 1. Choose the right activation function
    - 2. Use the suitable initialization
    - 3. Regularize the weights
    - 4. Add dropout between layers
    - 5. Batch normalization
- parameter number

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 16, 16, 32)        896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 8, 8, 32)          0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 8, 8, 64)          18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 4, 4, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 1024)              0
_________________________________________________________________
dense (Dense)                (None, 256)               262400
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 1285
=================================================================
Total params: 283,077
Trainable params: 283,077
Non-trainable params: 0
```

- Basic Network

- the input layer —input units (nodes)
  - encode the instance presented to the network for processing
  - example: attribute values of an object
- the hidden layer —hidden units (nodes)

  The most important parameter, the number of hidden nodes, is empirically chosen as a number around or close to

  $$\sqrt{nin \times non}$$

  where *nin* and *non* are respectively the number of input and output nodes (Timothy Masters)

  - are not directly observable
  - provide non-linearity for the network
  - an NN may have more than one hidden layers
- the output layer —output units (nodes)
  - encode possible concepts (or values) to be assigned to the instance under consideration
  - example: a class of objects
- convolutional layer

  - Performs 3 separate 2D convolutions (with padding) to generate 3 intermediate outputs
- Activation

- Add bias to each convolution output, and apply activation function to get the final output for the convolutional layer

- Pooling

- Apply 2 x 2 max pooling (stride 2) on the outputs from the first convolutional layer

- Dropout

- Depth
  - The importance of depth
    - large number of parameters do not necessary make a net powerful. Second, removing feature extractor layers hurt a net more
    - without the classifier the model still work fines
    - But when number of layers reduced by
    - half, the performance suffered a lot. The

        - power of deep net comes from its
        - depth
    - the curse of depth
        - errors explode or pretty small
        - the earlier layers(feature extraction) learn very slowly
        - since the training error from deeper net is higher, it is safe to say that, the added layers do not perform identity mapping -> hard to train
    - Solution

$$\mathbf{y} = H(\mathbf{x}) = F(\mathbf{x}) + \mathbf{x}$$

- When $F(\mathbf{x})$ approaches 0, then we have the identity mapping
- It is easier to get non-linear layers pushed to values of 0 rather than values of 1 (remember vanishing gradient?)

        - residual mapping
            - y = H(x)= F (x) + x
            - H(x) = F (x) + x
            - F (x) = H(x) − x
        - Resnet

ResNet: H/W改变时，1x1 conv, s=2, 丢失信息
ResNet-B: 调整PathA, s=2作用于3x3 conv, 不丢失信息
ResNet-C: 在ResNet-B的基础上,将第一个7x7的conv换做3个3x3的卷积核
ResNet-D: 由ResNet-B启发，在ResNet-C的基础上将每一个stage里第一个shortcut里的conv前接一个avg-pool，同时将该conv的stride由2变为1

- Batch size
    - Large batch size: (much) faster training, but accuracy may suffer
    - Small batch size: training gets noisy, offering regularizing effect and lower generalization
- Shared layers
    - Shared layers are often used to encode inputs from similar spaces (say, two different pieces of text that feature similar vocabulary). They enable sharing of information across these different inputs, and they make it possible to train such a model on less data. If a given word is seen in one of the inputs, that will benefit the processing of all inputs that pass through the shared layer.
- General Design Guidelines

- Carefully select the proper network architecture for the given task (number of hidden nodes) and a good choice of activation functions. Experimentation is usually helpful. More number of hidden nodes than what is required causes “over-fitting”– making its choice a gamble
- Choose a large, reliable training set and divide it into appropriate training, testing and validation sets for use in the corresponding operations
- Pre-process and analyse the training data for maximum utility and effectiveness (such as data smoothing, data normalization, etc.)
- Select good initial training parameter values (ηand α) and decide on changes to be made (dynamically) during training (experiment with the use of different values). Consider using modified BP training methods (e.g. Adam, AdaGrad, RMSProp, etc.) when the network is very large or the problem complex
- Once acceptable performance has been realized by the network, optimize it by pruning nodes and weights

- RNN: Recurrent neural network
  - Architecture

  - chop the series into segments, each segment with a fixed amount of time steps, called length
  - example

For RNN, the we feed the input segment by segment or row by row

$$\begin{array}{ccccc} 0, & 1, & 1, & 3, & 2 \\ 3, & 2, & 2, & 4, & 5 \\ 4, & 5, & 4, & 7, & 8 \end{array} \begin{array}{c} \longrightarrow \mathbf{x}_1 \\ \longrightarrow \mathbf{x}_2 \\ \longrightarrow \mathbf{x}_3 \end{array}$$

- Assume the $\mathbf{W}$ is a m x n matrix, $\mathbf{x}$ is n x 1 matrix, the matrix-vector dot product is

$$\mathbf{W} \cdot \mathbf{x} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} w_{11}x_1 + w_{12}x_2 + \cdots + w_{1n}x_n \\ w_{21}x_1 + w_{22}x_2 + \cdots + w_{2n}x_n \\ \vdots \\ w_{m1}x_1 + w_{m2}x_2 + \cdots + w_{mn}x_n \end{bmatrix}$$

- The m determines the size of the product output, i.e. the output feature size
  - Assume

$$\mathbf{W} = \begin{bmatrix} 1 & 0 & 0 & 1 & -1 \\ 2 & 0 & 0 & -1 & 1 \end{bmatrix}$$

  - and we know

$$\mathbf{x}_1 = [0 \quad 1 \quad 1 \quad 3 \quad 2]$$

$$\mathbf{W} \cdot \mathbf{x}_1 = \begin{bmatrix} 1 & 0 & 0 & 1 & -1 \\ 2 & 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 \times 0 + 0 \times 1 + 0 \times 1 + 1 \times 3 + (-1) \times 2 \\ 2 \times 0 + 0 \times 1 + 0 \times 1 + (-1) \times 3 + 1 \times 2 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

- The same U and W for x1, x2,x3
- U and W are changed only during
- updating phase of training
- U is a square matrix

- parameter number

Layer (type)	Output Shape	Param #	
input_1 (InputLayer)	(None, 16, 64)	0	
simple_rnn (SimpleRNN)	(None, 32)	3104	→ 32 x 64 + 32 x 32 + 32 = 2048 + 1024 + 32 = 3104
dense (Dense)	(None, 1)	33	→ 32 x 1 + 1 = 33

```
Total params: 3,137
Trainable params: 3,137
Non-trainable params: 0
```

  - p = (lout × lin) + (lout × lout) + lout

- Problem
  - the more we perform recurrent operation, mathematically, that is equivalent to adding more layers to the net

- vanishing/explode gradient comes in
- the improvement -> LSTM

- multiplication
  - The length of v and x must be equal

$$\mathbf{v} \odot \mathbf{x} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \odot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} v_1 x_1 \\ v_2 x_2 \\ \vdots \\ v_n x_n \end{bmatrix}$$

- carry

$$\mathfrak{w}_i = \sigma\left(\mathbf{U}_{\mathfrak{w}} \cdot \mathbf{h}_{i-1} + \mathbf{W}_{\mathfrak{w}} \cdot \mathbf{x}_i + \mathbf{b}_{\mathfrak{w}}\right)$$

$$\mathfrak{f}_i = \sigma\left(\mathbf{U}_{\mathfrak{f}} \cdot \mathbf{h}_{i-1} + \mathbf{W}_{\mathfrak{f}} \cdot \mathbf{x}_i + \mathbf{b}_{\mathfrak{f}}\right)$$

$$\mathfrak{h}_i = \sigma\left(\mathbf{U}_{\mathfrak{h}} \cdot \mathbf{h}_{i-1} + \mathbf{W}_{\mathfrak{h}} \cdot \mathbf{x}_i + \mathbf{b}_{\mathfrak{h}}\right)$$

- σ is the sigmoid function, and let's denote ⊙ as element-wise multiplication, and we have

$$\mathbf{c}_i = \mathfrak{w}_i \odot \mathfrak{h}_{i-1} + \mathbf{c}_{i-1} \odot \mathfrak{f}_i$$

- number of parameters

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 16, 64)            0
_________________________________________________________________
lstm (LSTM)                  (None, 32)                12416
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 33
=================================================================
Total params: 12,449
Trainable params: 12,449
Non-trainable params: 0
_________________________________________________________________
```

```
(32x32 + 32x64 + 32) x 4
= 3104 x 4
= 12416
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         (None, 3, 5)              0
_________________________________________________________________
lstm_1 (LSTM)                (None, 3, 7)              364
_________________________________________________________________
lstm_2 (LSTM)                (None, 9)                 612
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 10
=================================================================
Total params: 986
Trainable params: 986
Non-trainable params: 0
_________________________________________________________________
```

The output shape is `(None, 3, 7)` because of returned sequences, else it will simply be `(None, 7)`

The number of rows of the output of a lstm that returns sequences, is always equal to **the number of rows** of the input to the unit

  - $p = (lout \times lout + lout \times lin + lout) \times 4$
- Stacking
  - It is possible to stack recurrent layers and make the net deeper

- for LSTM to Dense, only the final output sequence fed into Dense layer

- CNN VS RNN

	CNN	RNN / LSTM
Usage	Suitable for spatial data, e.g. image, video	Suitable for temporal data (sequential data), e.g. text, speech
Capability	Considered more powerful than RNN	Less powerful and slower in calculation
Input	Take fixed size inputs and generate fixed size outputs	Can handle arbitrary input/output lengths
Nature	Use local connectivity pattern (through 2D convolution)	Use time series information

-