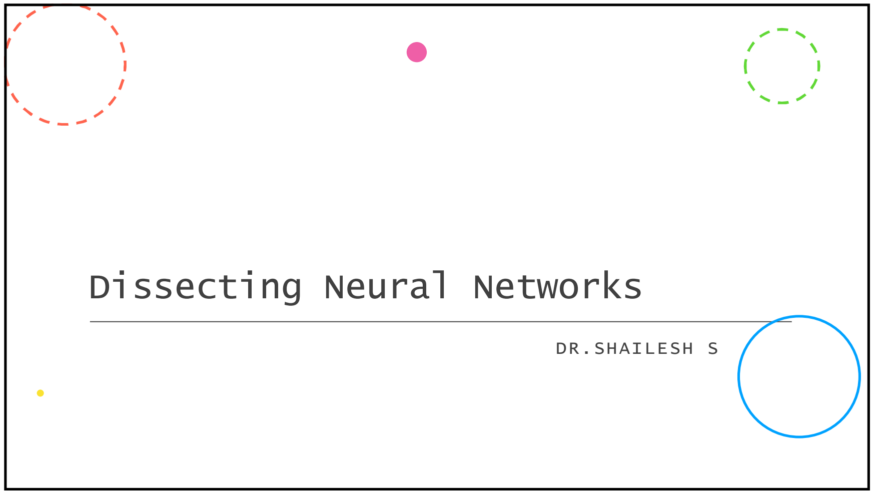
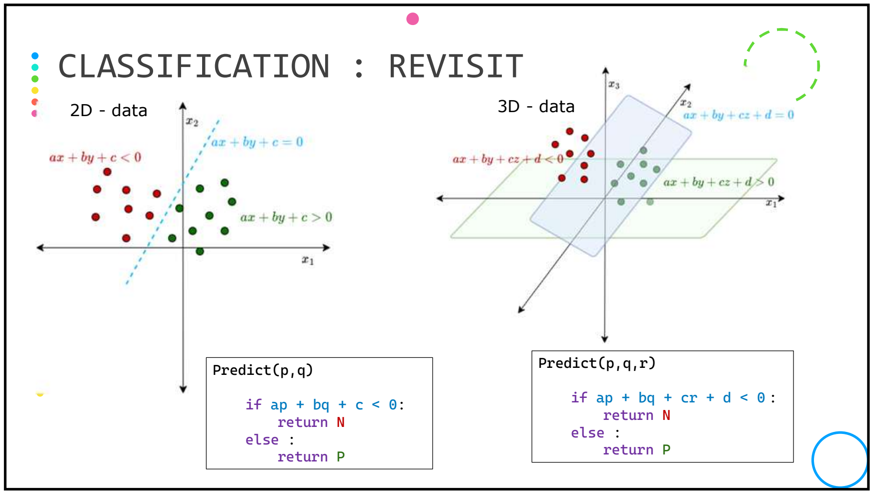
06-07-2025



Dissecting Neural Networks

DR.SHAILESH S



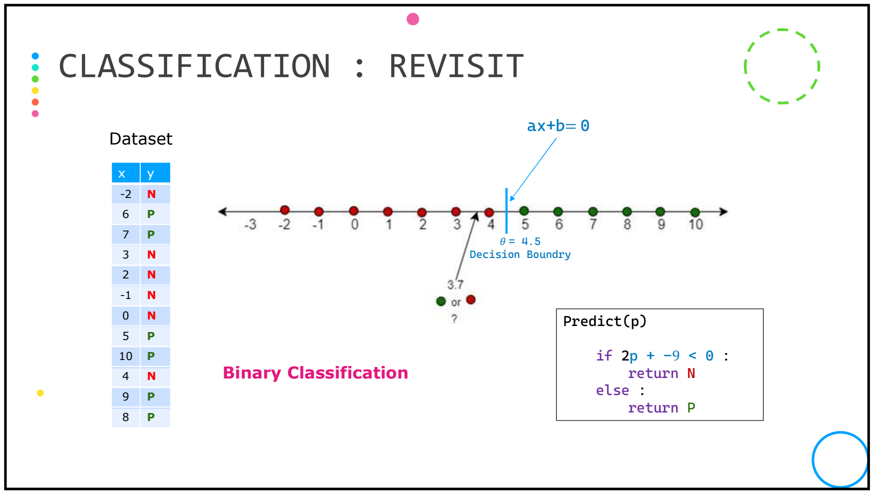
CLASSIFICATION : REVISIT

2D - data

```
Predict(p,q)

    if ap + bq + c < 0:
        return N
    else :
        return P
```

3D - data

```
Predict(p,q,r)

    if ap + bq + cr + d < 0 :
        return N
    else :
        return P
```



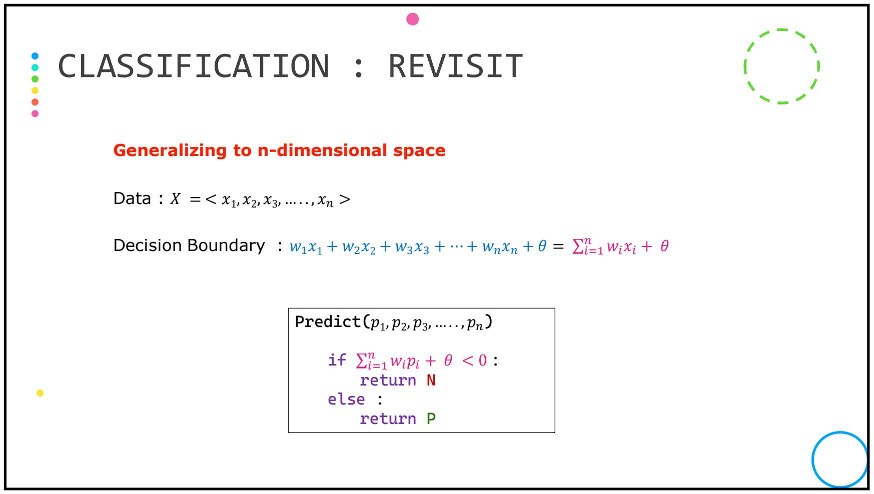
CLASSIFICATION : REVISIT

Dataset

x	y
-2	N
6	P
7	P
3	N
2	N
-1	N
0	N
5	P
10	P
4	N
9	P
8	P

Binary Classification

```
Predict(p)

    if 2p + −9 < 0 :
        return N
    else :
        return P
```



CLASSIFICATION : REVISIT

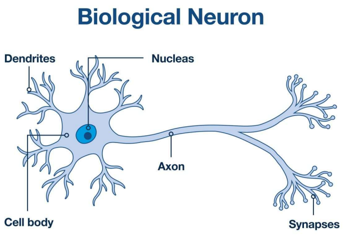
Generalizing to n-dimensional space

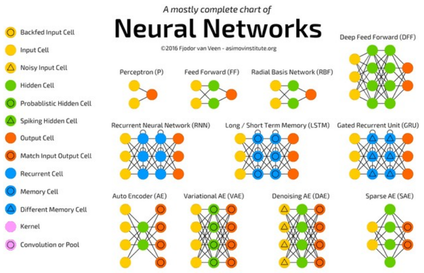
Data : $X = < x_1, x_2, x_3, \ldots\ldots, x_n >$

Decision Boundary : $w_1x_1 + w_2x_2 + w_3x_3 + \cdots + w_nx_n + \theta = \sum_{i=1}^{n} w_i x_i + \theta$

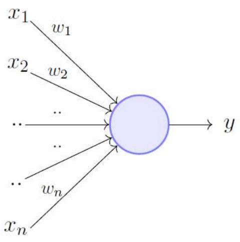
```
Predict($p_1, p_2, p_3, \ldots\ldots, p_n$)

    if $\sum_{i=1}^{n} w_i p_i + \theta < 0$ :
        return N
    else :
        return P
```

NEURAL NETWORKS





PERCEPTRON MODEL



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

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

Rewriting the above,

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

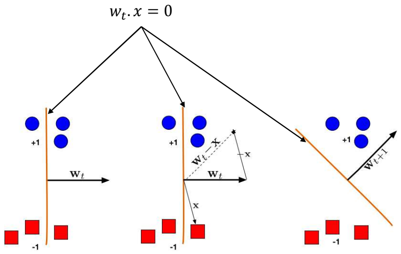
$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i - \theta < 0$$

PERCEPTRON

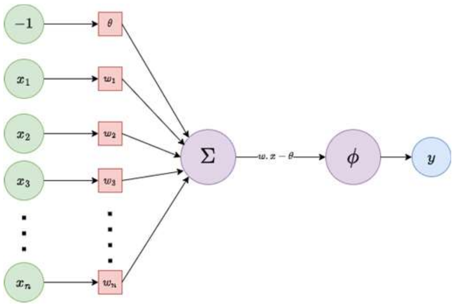
- ➢First neural network learning model in the 1960’s
- ➢Simple and limited (single layer models)
- ➢Basic concepts are similar for multi-layer models so this is a good learning tool
- ➢Still used in many current applications (modems, etc.)

PERCEPTRON ALGORITHM

```
Algorithm: Perceptron Learning Algorithm
$P \leftarrow inputs \quad with \quad label \quad 1;$
$N \leftarrow inputs \quad with \quad label \quad 0;$
Initialize $\mathbf{w}$ randomly;
while $!convergence$ do
    Pick random $\mathbf{x} \in P \cup N$ ;
    if $\mathbf{x} \in P \quad and \quad \mathbf{w.x} < 0$ then
        $\mathbf{w = w + x}$ ;
    end
    if $\mathbf{x} \in N \quad and \quad \mathbf{w.x} \geq 0$ then
        $\mathbf{w = w - x}$ ;
    end
end
//the algorithm converges when all the
 inputs are classified correctly
```



PERCEPTRON MODEL



LEARNING AND GATE

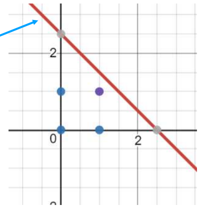
$z = w_1.x_1 + w_2.x_2 - \theta$

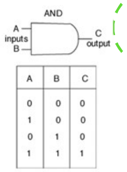
$w_1=1,\ w_2=1,\ \theta= 2.5$

$1\ x_1 + 1\ x_2 - 2.5 = 0$

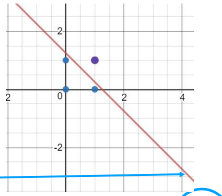
w_1	w_2	-θ	(x_1,x_2)	z
1	1	-2.5	(0,1) -	1x0 + 1x1 - 2.5 = -1.5
1	1	-2.5	(1,1) +	1x1 + 1x1 - 2.5 = -0.5
2	2	-2.5	(0,0) -	2x0 + 2x0 - 2.5 = -2.5
2	2	-2.5	(1,0) -	2x1 + 2x0 - 2.5 = -0.5
2	2			

$2\ x_1 + 2\ x_2 - 2.5 = 0$

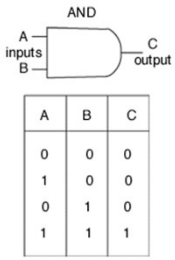




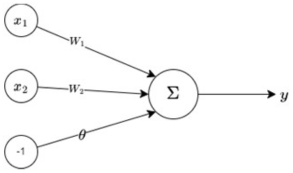
A	B	C
0	0	0
1	0	0
0	1	0
1	1	1



LEARNING AND GATE



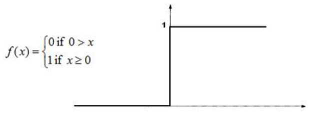
A	B	C
0	0	0
1	0	0
0	1	0
1	1	1



LEARNING AND GATE

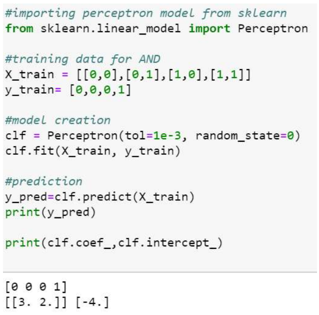
x_1	x_2	$y = 2x_1 + 2x_2 - 2.5$	$f(y)$
0	0	-2.5	0
0	1	-0.5	0
1	0	-0.5	0
1	1	1.5	1

Unit step (threshold)



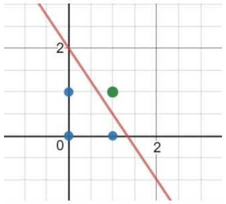
$$f(x) = \begin{cases} 0 \text{ if } 0 > x \\ 1 \text{ if } x \geq 0 \end{cases}$$

IMPLEMENTING AND GATE



```
#importing perceptron model from sklearn
from sklearn.linear_model import Perceptron

#training data for AND
X_train = [[0,0],[0,1],[1,0],[1,1]]
y_train= [0,0,0,1]

#model creation
clf = Perceptron(tol=1e-3, random_state=0)
clf.fit(X_train, y_train)

#prediction
y_pred=clf.predict(X_train)
print(y_pred)

print(clf.coef_,clf.intercept_)
```

```
[0 0 0 1]
[[3. 2.]] [-4.]
```



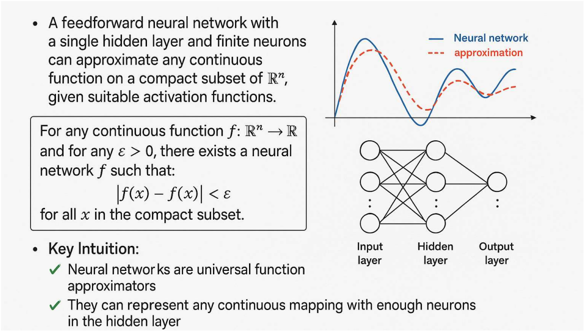
Tools

- Python
- Sklearn

Try It For

OR, NAND, NOR, XOR

UNIVERSAL APPROXIMATION THEOREM



- A feedforward neural network with a single hidden layer and finite neurons can approximate any continuous function on a compact subset of $\mathbb{R}^n$, given suitable activation functions.

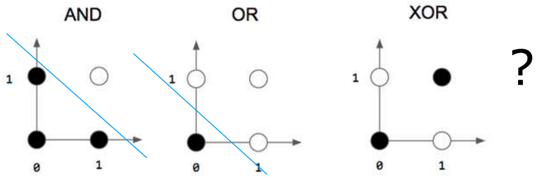
For any continuous function $f\colon \mathbb{R}^n \to \mathbb{R}$ and for any $\varepsilon > 0$, there exists a neural network f such that:

$$\left|f(x) - f(x)\right| < \varepsilon$$

for all x in the compact subset.

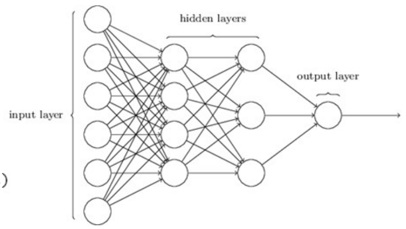
- Key Intuition:
  - ✓ Neural networ ks are universal function approximators
  - ✓ They can represent any continuous mapping with enough neurons in the hidden layer

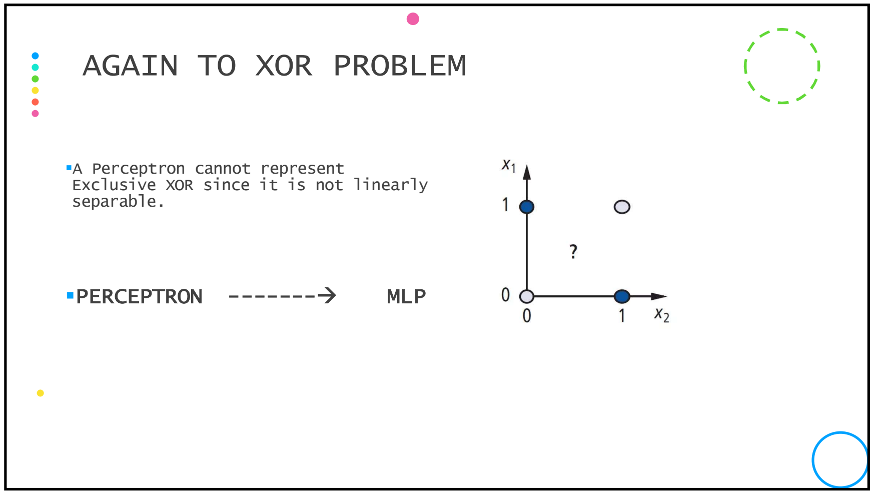
MORE GATES



MULTI LAYER PERCEPTRON(MLP)

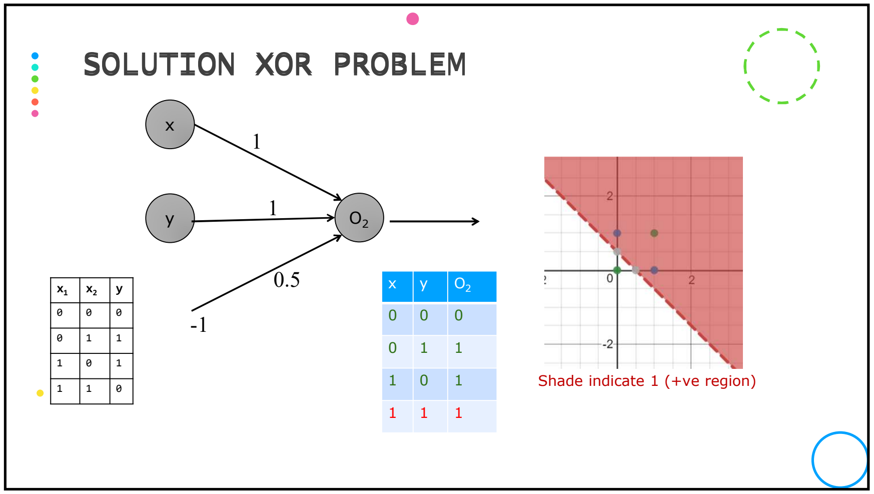
- Feedforward network: The neurons in each layer feed their output forward to the next layer until we get the final output from the neural network.
- There can be any number of hidden layers within a feedforward network.
- The number of neurons can be completely arbitrary.
- MLP used to describe any general feedforward (no recurrent connections) network





AGAIN TO XOR PROBLEM

- A Perceptron cannot represent Exclusive XOR since it is not linearly separable.
- PERCEPTRON --------→ MLP

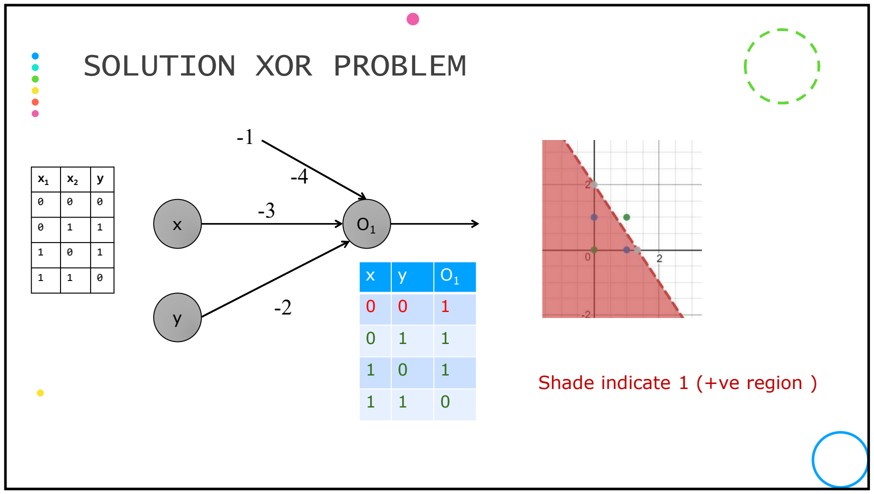


SOLUTION XOR PROBLEM

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

x	y	O_2
0	0	0
0	1	1
1	0	1
1	1	1

Shade indicate 1 (+ve region)

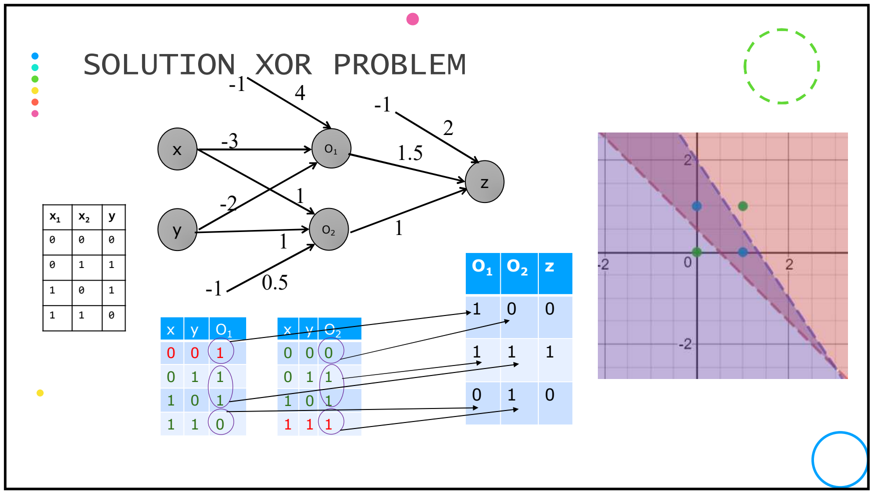


SOLUTION XOR PROBLEM

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

x	y	O_1
0	0	1
0	1	1
1	0	1
1	1	0

Shade indicate 1 (+ve region)



SOLUTION XOR PROBLEM

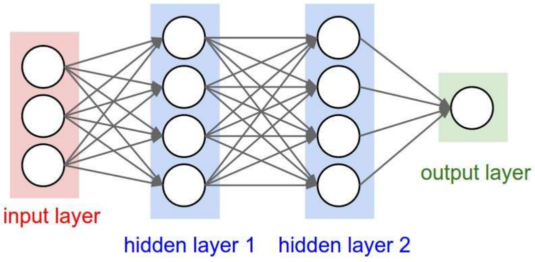
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

x	y	O_1
0	0	1
0	1	1
1	0	1
1	1	0

x	y	O_2
0	0	0
0	1	1
1	0	1
1	1	1

O_1	O_2	z
1	0	0
1	1	1
0	1	0

THREE LAYER NETWORKS



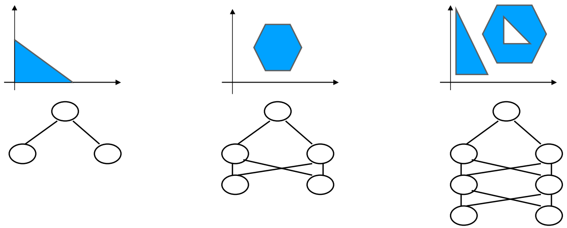
THREE LAYER NETWORKS

- No connections within a layer
- No direct connections between input and output layers
- Fully connected between layers
- Often more than 3 layers
- Number of output units need not equal number of input units
- Number of hidden units per layer can be more or less than input or output units

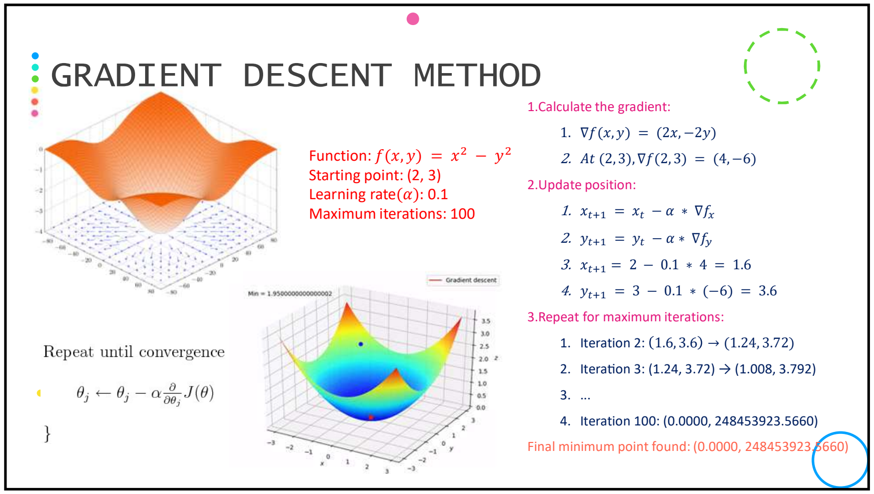
THREE LAYER NETWORKS

- No connections within a layer
- No direct connections between input and output layers
- Fully connected between layers
- Often more than 3 layers
- Number of output units need not equal number of input units
- Number of hidden units per layer can be more or less than input or output units

WHAT DO EACH OF THESE LAYER DO ?



- 1st layer draws linear boundaries
- 2nd layer combines the boundaries
- 3rd layer can generate arbitrarily complex boundaries



GRADIENT DESCENT METHOD

Function: $f(x,y) = x^2 - y^2$
Starting point: (2, 3)
Learning rate(α): 0.1
Maximum iterations: 100

Repeat until convergence

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

}

1.Calculate the gradient:

1. $\nabla f(x,y) = (2x,-2y)$
2. $At\ (2,3), \nabla f(2,3) = (4,-6)$

2.Update position:

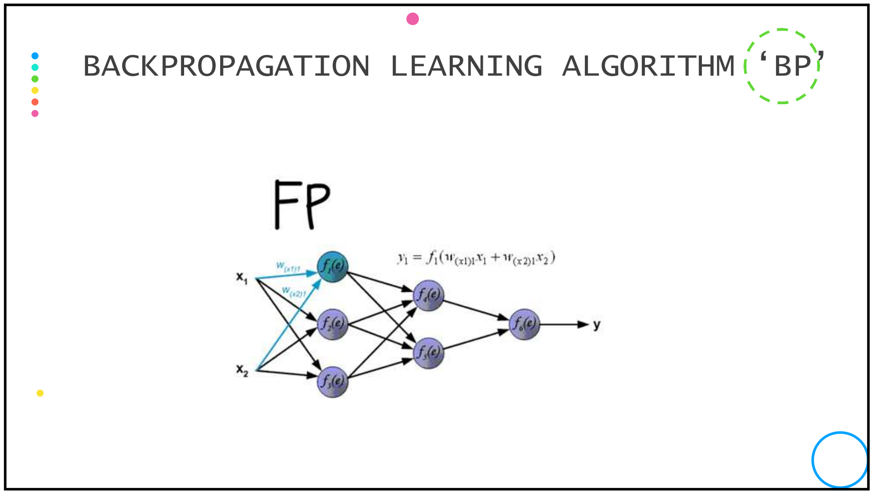
1. $x_{t+1} = x_t - \alpha * \nabla f_x$
2. $y_{t+1} = y_t - \alpha * \nabla f_y$
3. $x_{t+1} = 2 - 0.1 * 4 = 1.6$
4. $y_{t+1} = 3 - 0.1 * (-6) = 3.6$

3.Repeat for maximum iterations:

1. Iteration 2: $(1.6, 3.6) \rightarrow (1.24, 3.72)$
2. Iteration 3: (1.24, 3.72) → (1.008, 3.792)
3. ...
4. Iteration 100: (0.0000, 248453923.5660)

Final minimum point found: (0.0000, 248453923.5660)

BACKPROPAGATION LEARNING ALGORITHM ‘BP’



BACKPROPAGATION LEARNING ALGORITHM ‘BP’

- BP has two phases:

Forward pass phase: computes ‘functional signal’, feed forward propagation of input pattern signals through network

Backward pass phase: computes ‘error signal’, propagates the error backwards through network starting at output units (where the error is the difference between actual and desired
output values)

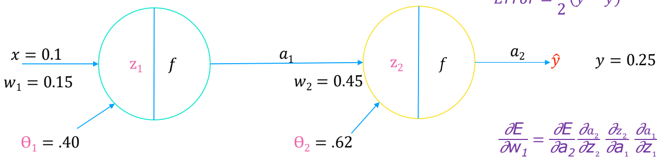
BACKPROPAGATION LEARNING ALGORITHM ‘BP’

- Error gradient along all connection weights were measured by propagating the error from output layer.
- First, a forward pass is performed - output of every neuron in every layer is computed.
- Output error is estimated.
- Then compute how much each neuron in last hidden layer contributed to output error.
- This is repeated backwards until input layer.
- Last step is Gradient Descent on all connection weights using error gradients estimated in previous steps.

BACK PROPOAGATION

$\hat{y} = a_2$

$Error = \frac{1}{2}(y - \hat{y})^2$



$z_{1(w_1, \Theta_1)} = w_1\,x + \Theta_1$

$a_1(z_1) = \sigma(z_1) = \frac{1}{1 + e^{-(z_1)}}$

$z_2(w_2, \Theta_2) = w_2\,x + \Theta_2$

$a_2(z_2) = \sigma(z_2) = \frac{1}{1 + e^{-(z_2)}}$

$w_1 = w_1 - \alpha \frac{\partial E}{\partial w_1}$

$w_2 = w_2 - \alpha \frac{\partial E}{\partial w_2}$

$\Theta_1 = \Theta_1 - \alpha \frac{\partial E}{\partial \Theta_1}$

$\Theta_2 = \Theta_2 - \alpha \frac{\partial E}{\partial \Theta_2}$

$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial a_1}\frac{\partial a_1}{\partial z_1}\frac{\partial z_1}{\partial w_1}$

$\frac{\partial E}{\partial w_2} = \frac{\partial E}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial w_2}$

$\frac{\partial E}{\partial \Theta_1} = \frac{\partial E}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial a_1}\frac{\partial a_1}{\partial z_1}\frac{\partial z_1}{\partial \Theta_1}$

$\frac{\partial E}{\partial \Theta_2} = \frac{\partial E}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial \Theta_2}$

MORE OPTIMIZERS

- **Optimizers**

- Algorithms or methods used to minimize an error function (*Loss function*)or to maximize the efficiency of production.

- Mathematical functions which are dependent on model's learnable parameters i.e Weights & Biases.

- **Gradient Descent**
- **Stochastic Gradient Descent**
- **Stochastic Gradient Descent with Momentum**
- **Mini-Batch Gradient Descent**
- **AdaGrad(Adaptive Gradient Descent)**
- **RMS-Prop (Root Mean Square Propagation)**
- **AdaDelta**
- **Adam(Adaptive Moment Estimation)**

$w_1 = w_1 - \alpha \frac{\partial E}{\partial w_1} = \mathbf{0.15 - 1 * 0.00102 = 0.148}$

$w_2 = w_2 - \alpha \frac{\partial E}{\partial w_2} = \mathbf{0.45 - 1 * 0.0570 = 0.393}$

$\Theta_1 = \Theta_1 - \alpha \frac{\partial E}{\partial \theta_1} = \mathbf{0.40 - 1 * 0.102 = 0.389}$

$\Theta_1 = \Theta_1 - \alpha \frac{\partial E}{\partial \theta_1} = \mathbf{0.62 - 1 * 0.094 = 0.525}$

$z_1(w_1, \theta_1) = w_1\,x + \theta_1 = 0.015 * 0.1 + 0.40 = 0.415$

$a_1(z_1) = \sigma(z_1) = \frac{1}{1 + e^{-(z_1)}} = \frac{1}{1 + e^{-(0.415)}} = 0.6022$

$z_2(w_2, \theta_2) = w_2\,x + \theta_2 = 0.45 * 0.6022 + 0.63 = 0.890$

$a_2(z_2) = \sigma(z_2) = \frac{1}{1 + e^{-(z_2)}} = \frac{1}{1 + e^{-(0.890)}} = 0.7088$

$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial a_1}\frac{\partial a_1}{\partial z_1}\frac{\partial z_1}{\partial w_1}$

$=(a_2 - y)\; a_2\; (1 - a_2)\; w_2\, a_1\; (1 - a_1)\; x$

$=(0.7088-0.25)\; 0.7088\; (1-0.7088)0.1$

$=0.00102$

$\frac{\partial E}{\partial w_2} = \frac{\partial E}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial w_2}$

$=(a_2 - y)\; \sigma(z_2)(1-\sigma(z_2))\; a_1$

$=(a_2 - y)\; a_2\; (1 - a_2)\; a_1$

$=(0.7088-.25)\; 0.7088(1-0.7088)$

$=0.0570$

$\frac{\partial E}{\partial \Theta_1} = \frac{\partial E}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial a_1}\frac{\partial a_1}{\partial z_1}\frac{\partial z_1}{\partial \Theta_1}$

$=(a_2 - y)\; a_2\; (1 - a_2)\; w_2\, a_1\; (1 - a_1)\; 1$

$=(0.7088-0.25)\; 0.7088\; (1-0.7088)1$

$=0.0102$

$\frac{\partial E}{\partial \Theta_2} = \frac{\partial E}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial \Theta_2}$

$=(a_2 - y)\; \sigma(z_2)(1-\sigma(z_2))\; 1$

$=(a_2 - 0.25)\; a_2\; (1 - a_2)$

$=0.0946$

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

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

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

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

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

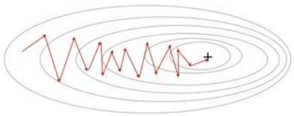
$\sigma'(x) = \sigma(x).\,(1 - \sigma(x))$

STOCHASTIC GRADIENT DESCENT

Stochastic means randomness on which the algorithm is based upon

A variant of gradient descent that involves updating the parameters based on a small, randomly-selected subset of the data rather than the full dataset.

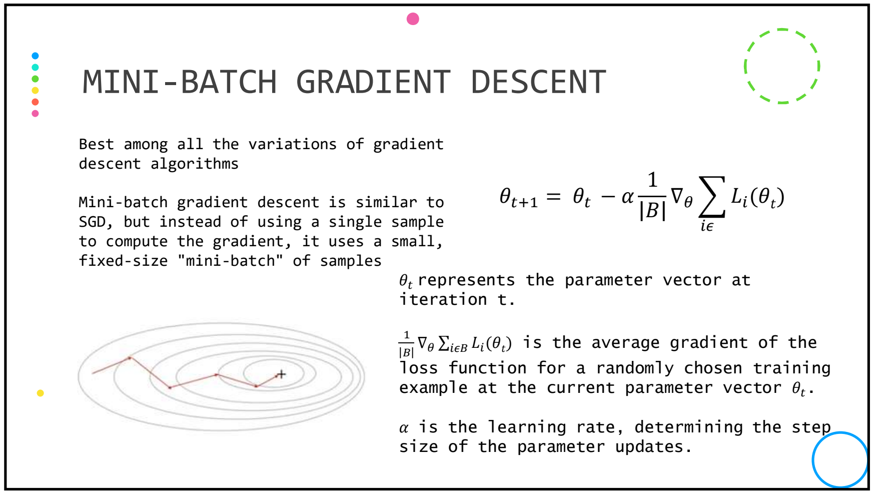
$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta L_i(\theta_t)$$



θ_t represents the parameter vector at iteration t.

$\nabla_\theta L_i(\theta_t)$ is the gradient of the loss function for a randomly chosen training example at the current parameter vector θ_t .

α is the learning rate, determining the step size of the parameter updates.



MINI-BATCH GRADIENT DESCENT

Best among all the variations of gradient descent algorithms

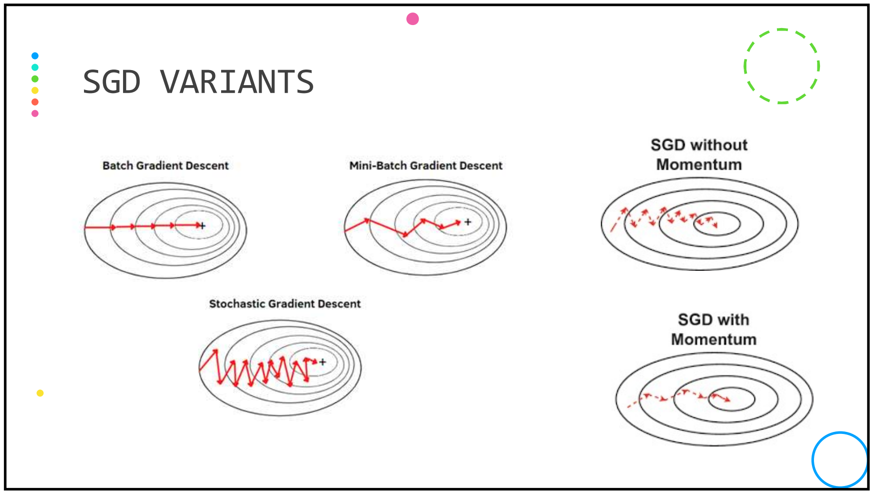
Mini-batch gradient descent is similar to SGD, but instead of using a single sample to compute the gradient, it uses a small, fixed-size "mini-batch" of samples

$$\theta_{t+1} = \theta_t - \alpha \frac{1}{|B|} \nabla_\theta \sum_{i\epsilon} L_i(\theta_t)$$

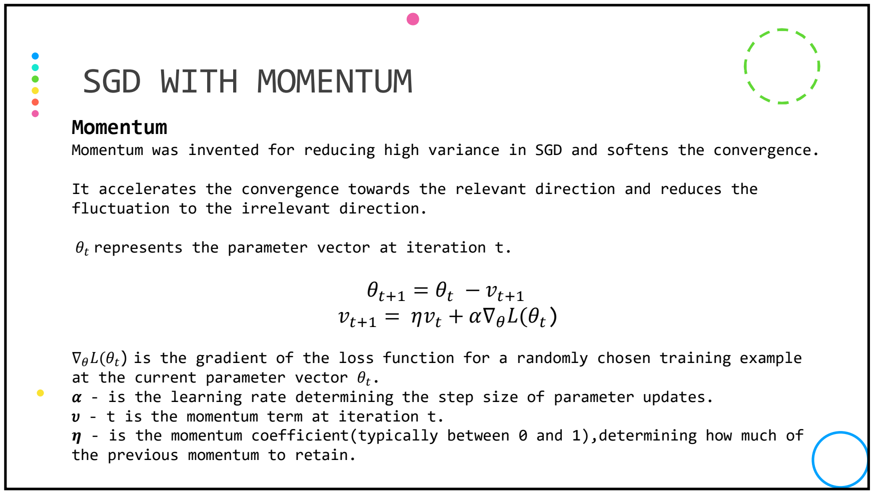
θ_t represents the parameter vector at iteration t.

$\frac{1}{|B|} \nabla_\theta \sum_{i\epsilon B} L_i(\theta_t)$ is the average gradient of the loss function for a randomly chosen training example at the current parameter vector θ_t.

α is the learning rate, determining the step size of the parameter updates.



SGD VARIANTS



SGD WITH MOMENTUM

Momentum

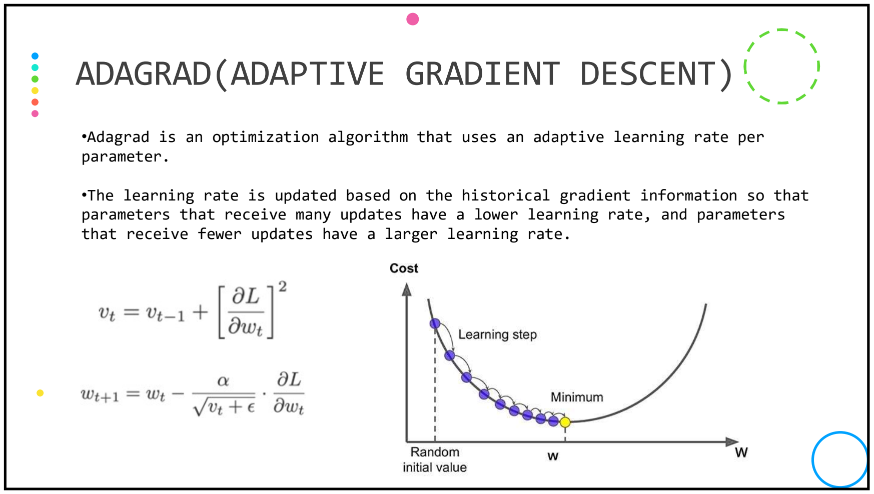
Momentum was invented for reducing high variance in SGD and softens the convergence.

It accelerates the convergence towards the relevant direction and reduces the fluctuation to the irrelevant direction.

θ_t represents the parameter vector at iteration t.

$$\theta_{t+1} = \theta_t - v_{t+1}$$
$$v_{t+1} = \eta v_t + \alpha \nabla_\theta L(\theta_t)$$

$\nabla_\theta L(\theta_t)$ is the gradient of the loss function for a randomly chosen training example at the current parameter vector θ_t.
$\boldsymbol{\alpha}$ - is the learning rate determining the step size of parameter updates.
$\boldsymbol{v}$ - t is the momentum term at iteration t.
$\boldsymbol{\eta}$ - is the momentum coefficient(typically between 0 and 1),determining how much of the previous momentum to retain.



ADAGRAD(ADAPTIVE GRADIENT DESCENT)

•Adagrad is an optimization algorithm that uses an adaptive learning rate per parameter.

•The learning rate is updated based on the historical gradient information so that parameters that receive many updates have a lower learning rate, and parameters that receive fewer updates have a larger learning rate.

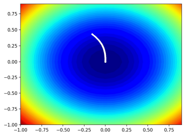
$$v_t = v_{t-1} + \left[\frac{\partial L}{\partial w_t}\right]^2$$

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} \cdot \frac{\partial L}{\partial w_t}$$

ADA DELTA

- AdaDelta is an optimization algorithm similar to RMSProp but does not require a hyperparameter learning rate.
- It uses an exponentially decaying average of the gradients and the squares of the gradients to determine the updated scale.

$$\theta_t = \theta_{t-1} - \eta \cdot (\sqrt{G_t + \epsilon})^{-1/2} \cdot g_t$$



- θ_t : Parameter vector at iteration t
- θ_{t-1} : Parameter vector at iteration $t-1$
- η: Learning rate
- G_t : Diagonal matrix of accumulated squared gradients up to iteration t
- ϵ: Small positive value to prevent division by zero
- g_t : Gradient of the loss function with respect to θ_t Visualization:

ADAM(ADAPTIVE MOMENT ESTIMATION)

- Adam is an optimization algorithm that combines the ideas of SGD with momentum and RMSProp.
- It uses an exponentially decaying average of the gradients and the squares of the gradients to determine the updated scale, similar to RMSProp

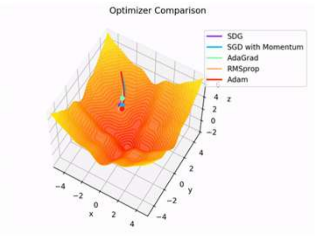
$$\nu_t = \beta_1 * \nu_{t-1} - (1 - \beta_1) * g_t$$

$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2$$

$$\Delta\omega_t = -\eta \frac{\nu_t}{\sqrt{s_t + \epsilon}} * g_t$$

$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

η : *Initial Learning rate*
g_t : *Gradient at time t along* ω^j
ν_t : *Exponential Average of gradients along* ω_j
s_t : *Exponential Average of squares of gradients along* ω_j
β_1, β_2 : *Hyperparameters*



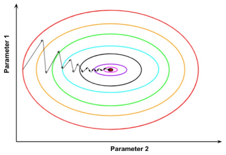
RMSPROP(ROOT MEAN SQUARE PROBABILITY)

- RMSProp is an optimization algorithm similar to Adagrad, but it uses an exponentially decaying average of the squares of the gradients rather than the sum.
- Helps to reduce the monotonic learning rate decay of Adagrad and improve convergence.

$$\mathbf{g}_t = \nabla_\theta L(\theta_t)$$

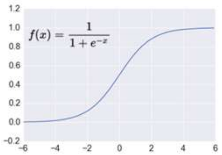
$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + (1 - \gamma)\mathbf{g}_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\mathbf{v}_t} + \epsilon}\mathbf{g}_t$$



$\boldsymbol{\epsilon}$ *is a small positive constant for numerical stability.* $\boldsymbol{\eta}$ *is the learning rate.* $\boldsymbol{\gamma}$ *is the decay rate for the RMSp*

Activation: Sigmoid



$R^n \rightarrow [0,1]$

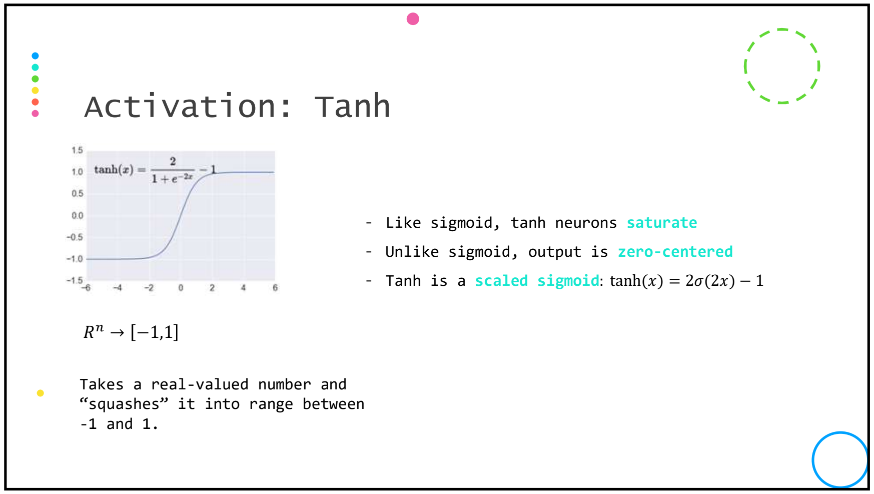
Takes a real-valued number and “squashes” it into range between 0 and 1.

+ Nice interpretation as the firing rate of a neuron

- 0 = not firing at all
- 1 = fully firing

- Sigmoid neurons saturate and kill gradients, thus NN will barely learn

- when the neuron’s activation are 0 or 1 (saturate)
  - 😟gradient at these regions almost zero
  - 😟almost no signal will flow to its weights
  - 😟if initial weights are too large then most neurons would saturate

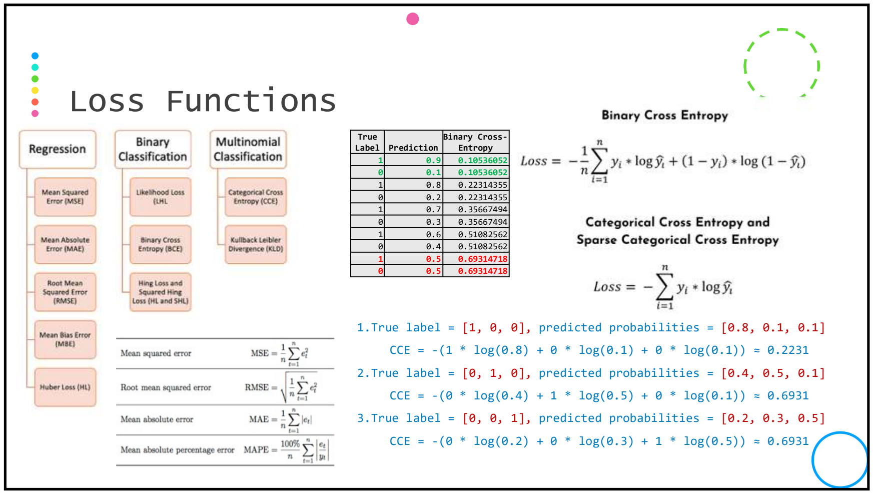


Activation: Tanh

- Like sigmoid, tanh neurons saturate
- Unlike sigmoid, output is zero-centered
- Tanh is a scaled sigmoid: $\tanh(x) = 2\sigma(2x) - 1$

$R^n \to [-1,1]$

Takes a real-valued number and "squashes" it into range between -1 and 1.



Loss Functions

Regression
Mean Squared Error (MSE)
Mean Absolute Error (MAE)
Root Mean Squared Error (RMSE)
Mean Bias Error (MBE)
Huber Loss (HL)
Binary Classification
Likelihood Loss (LHL
Binary Cross Entropy (BCE)
Hing Loss and Squared Hing Loss (HL and SHL)
Multinomial Classification
Categorical Cross Entropy (CCE)
Kullback Leibler Divergence (KLD)

True Label	Prediction	Binary Cross-Entropy
1	0.9	0.10536052
0	0.1	0.10536052
1	0.8	0.22314355
0	0.2	0.22314355
1	0.7	0.35667494
0	0.3	0.35667494
1	0.6	0.51082562
0	0.4	0.51082562
1	0.5	0.69314718
0	0.5	0.69314718

Mean squared error	$\text{MSE} = \frac{1}{n}\sum_{t=1}^{n} e_t^2$
Root mean squared error	$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{t=1}^{n} e_t^2}$
Mean absolute error	$\text{MAE} = \frac{1}{n}\sum_{t=1}^{n} \lvert e_t \rvert$
Mean absolute percentage error	$\text{MAPE} = \frac{100\%}{n}\sum_{t=1}^{n} \left\lvert \frac{e_t}{y_t} \right\rvert$

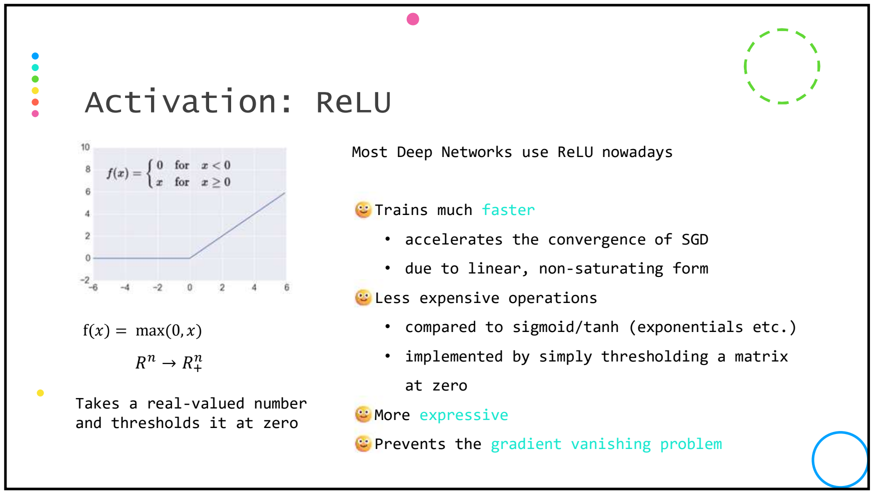
Binary Cross Entropy

$$Loss = -\frac{1}{n}\sum_{i=1}^{n} y_i * \log \hat{y}_i + (1-y_i) * \log(1-\hat{y}_i)$$

Categorical Cross Entropy and Sparse Categorical Cross Entropy

$$Loss = -\sum_{i=1}^{n} y_i * \log \hat{y}_i$$

1. True label = [1, 0, 0], predicted probabilities = [0.8, 0.1, 0.1]
   CCE = -(1 * log(0.8) + 0 * log(0.1) + 0 * log(0.1)) ≈ 0.2231
2. True label = [0, 1, 0], predicted probabilities = [0.4, 0.5, 0.1]
   CCE = -(0 * log(0.4) + 1 * log(0.5) + 0 * log(0.1)) ≈ 0.6931
3. True label = [0, 0, 1], predicted probabilities = [0.2, 0.3, 0.5]
   CCE = -(0 * log(0.2) + 0 * log(0.3) + 1 * log(0.5)) ≈ 0.6931



Activation: ReLU

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

$R^n \to R_+^n$

Takes a real-valued number and thresholds it at zero

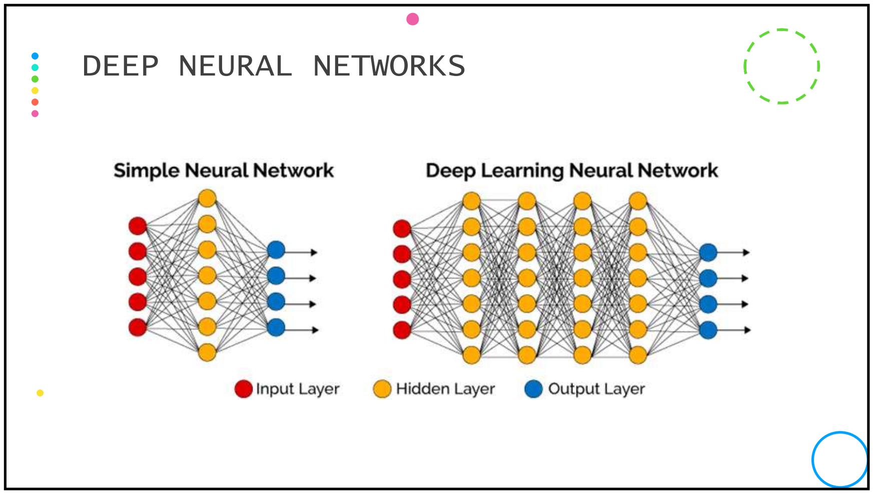
Most Deep Networks use ReLU nowadays

😊Trains much faster
- accelerates the convergence of SGD
- due to linear, non-saturating form

😊Less expensive operations
- compared to sigmoid/tanh (exponentials etc.)
- implemented by simply thresholding a matrix at zero

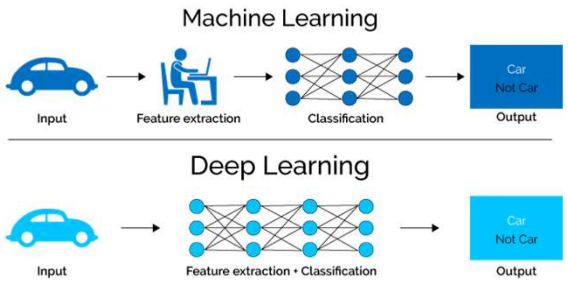
😊More expressive

😊Prevents the gradient vanishing problem



DEEP NEURAL NETWORKS

MACHINE VS DEEP LEARNING



👤 Dr. Shailesh Sivan

☎ +91 8907230664

✉ shaileshsivan@cusat.ac.in

https://shaileshsivan.info

TAKE AWAYS

- Mathematical Foundations for Optimization
- Neural Networks
- Perceptron
- Multilayer Perceptron
- Deep Neural Networks
- Gradient Descent
- Back Propagation
- Optimizers
- Activation Functions
- Loss Functions