CS60010: Deep Learning
Spring 2021

Sudeshna Sarkar

Module 2

Part 4

Multilayer Perceptron

Sudeshna Sarkar

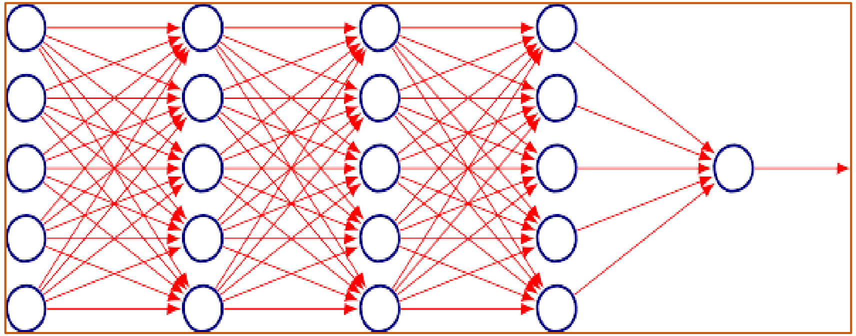
25 Jan 2021

Feedforward Networks and Backpropagation

Introduction

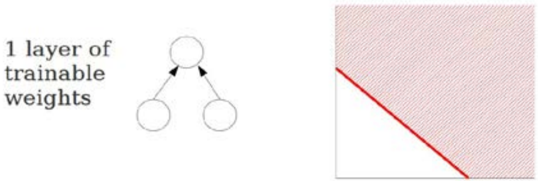
- Goal: Approximate some unknown ideal function $f^*: X \rightarrow Y$
- Ideal classifier: $y = f^*(x)$ for (x, y)
- Feedforward Network: Define parametric mapping $y = f(x; \theta)$
- Learn parameters θ to get a good approximation to f^* from training data
- Function f is a composition of many different functions e.g.

$$f(x) = f^3\left(f^2\left(f^1(x)\right)\right)$$

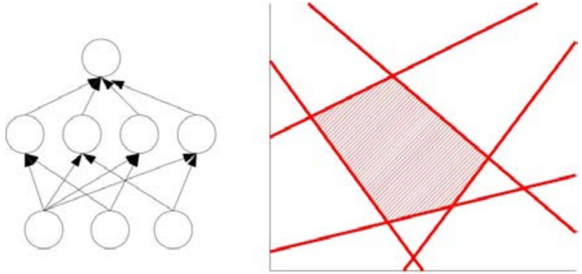


- Training: Optimize θ to drive $f(x; \theta)$ closer to $f^*(x)$
  - Only specifies the output of the *output layers*
  - Output of intermediate layers is not specified by D, hence the nomenclature *hidden layers*
- Neural: Choices of $f^{(i)}$'s and layered organization, loosely inspired by neuroscience

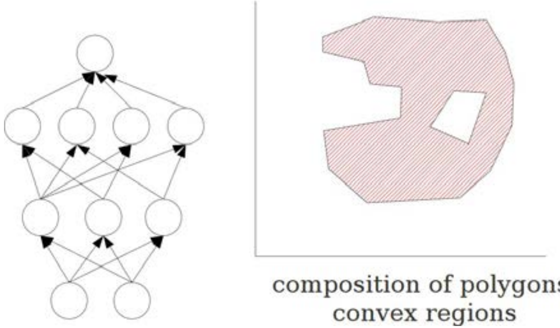
Beyond single layer



separating hyperplane



convex polygon region



composition of polygons:
convex regions

Training a NN

- Train a Neural Network with gradient descent
- But most interesting loss functions are non-convex
- Unlike in convex optimization, no convergence guarantees
- To apply gradient descent: Need to specify cost function, and output representation

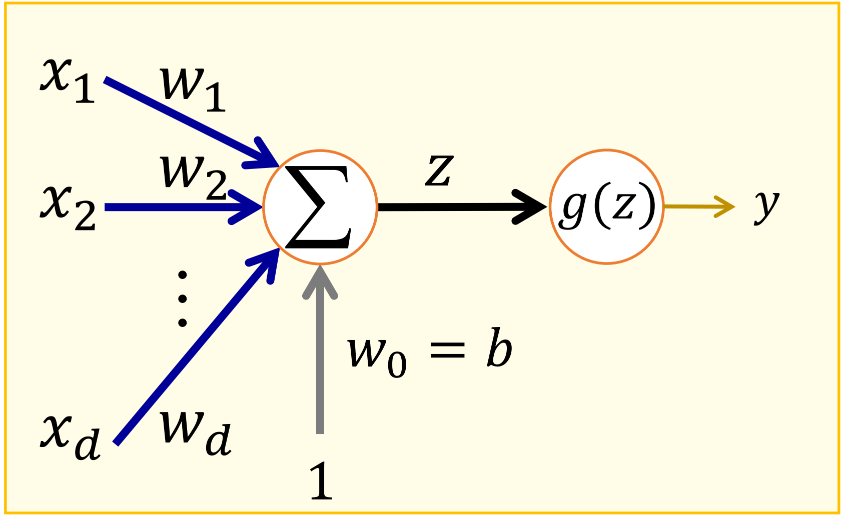
Loss Functions

- Define a distribution p(y|x; θ) and use principle of maximum likelihood.
- We can just use cross entropy between training data and the model's predictions as the cost function:

$$J(\theta) = E_{x,y \sim \hat{p}_{data}} \log p_{model} \cdot (y|x)$$

- Choice of output units is very important for choice of cost function

Artificial Neuron



$\boldsymbol{w} = [w_1\ w_2 \dots w_d]^T$ and $\boldsymbol{x} = [x_1\ x_2 \dots x_d]^T$

$$\boldsymbol{z} = b + \sum_{i=1}^{d} w_i x_i = [\boldsymbol{w}^T b] \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}$$

$$\boldsymbol{y} = g(z)$$

Terminologies

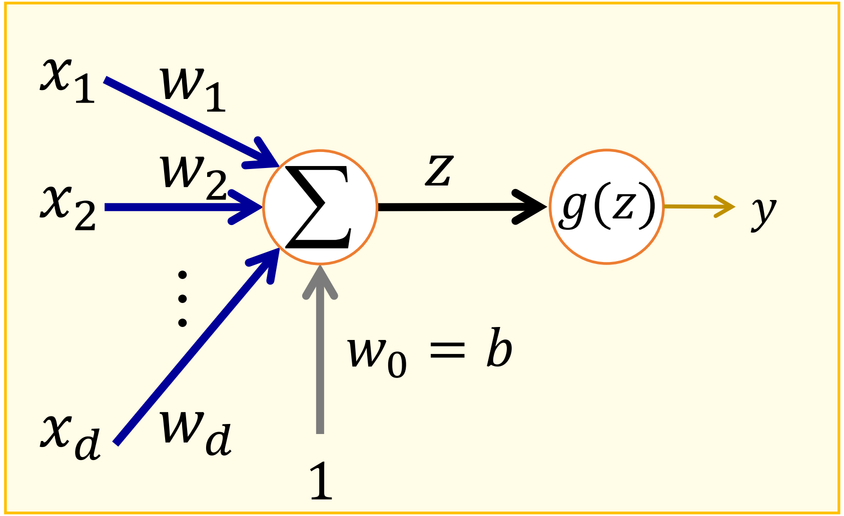
$\boldsymbol{x}$: input, $\boldsymbol{w}$: weights, $\boldsymbol{b}$: bias

z: pre-activation (input activation)

g: activation function

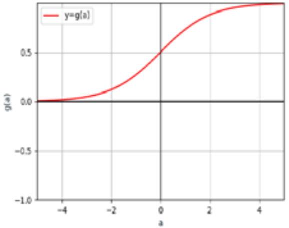
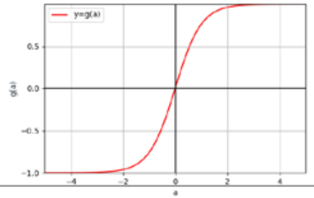
y: activation for output units

Perceptron



$\boldsymbol{x} \in \mathcal{R}^d$ and $y \in \{0, 1\}$ for Binary Classification

Common Activation Functions for Output

Name	Function	Gradient	Graph
Binary step	$sign(z)$	$g'(z) = \begin{cases} 0, & z \neq 0 \\ NA, & z = 0 \end{cases}$	
Sigmoid	$\sigma(z) = \dfrac{1}{1 + \exp(-z)}$	$g'(z) = g(z)(1 - g(z))$	
Tanh	$\tanh z = \dfrac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$	$g'^{(z)} = 1 - g^2(z)$	

Output Units: Linear

$$\hat{y} = w^T a + b$$

Used to produce the mean of a conditional Gaussian distribution:

$$p(\mathbf{y} \mid \mathbf{x}) = N(\mathbf{y}; \hat{\mathbf{y}}, \sigma)$$

Maximizing log-likelihood $\Longrightarrow$ Minimizing squared error

Output Units: Sigmoid

$$\hat{y} = \sigma(w^T a + b)$$

$$J(\theta) = -\log p(y|x)$$
$$= -\log \sigma\big((2y-1)(\boldsymbol{w}^T \boldsymbol{a} + b)\big)$$

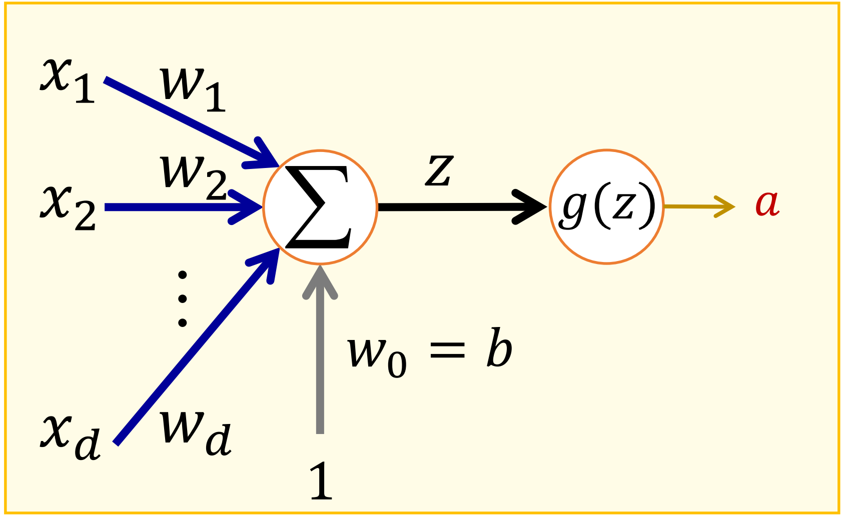
Output Softmax Units

Need to produce a vector $\hat{y}$ with $\hat{y}_i = p(y = i|x)$

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

$$\log \text{softmax}(z)_i = z_i - \log \sum_j \exp(z_j)$$

Artificial Neuron – hidden unit



$\boldsymbol{w} = [w_1\ w_2 \dots w_d]^T$ and $\boldsymbol{x} = [x_1\ x_2 \dots x_d]^T$

$$\boldsymbol{z} = b + \sum_{i=1}^{d} w_i x_i = [\boldsymbol{w}^T b] \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}$$

$$\boldsymbol{a} = g(z)$$

Terminologies

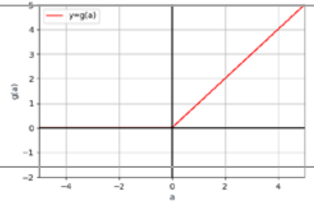
$\boldsymbol{x}$: input, $\boldsymbol{w}$: weights, $\boldsymbol{b}$: bias

z: pre-activation (input activation)

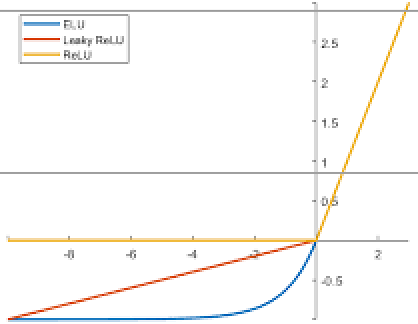
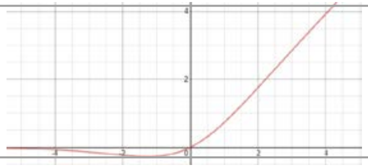
g: activation function

a: activation at hidden units

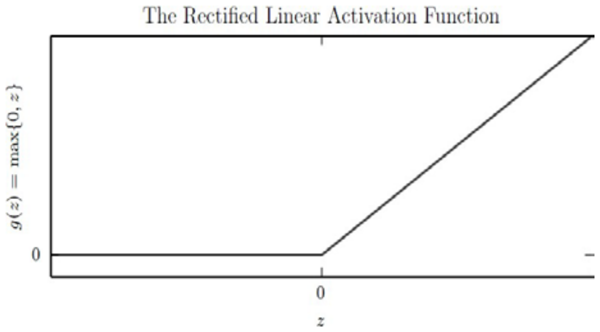
Activation Functions for Hidden Nodes

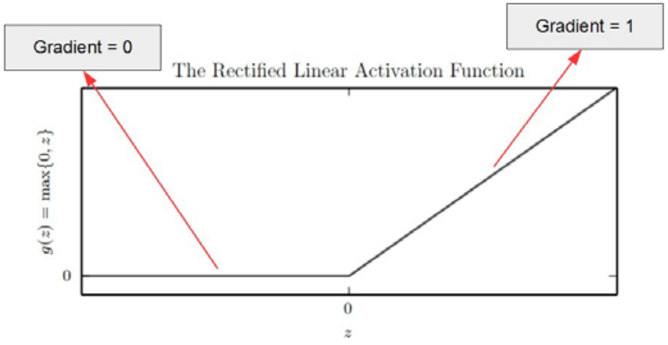
Name	Function	Gradient	Graph
Sigmoid	$\sigma(z) = \frac{1}{1 + \exp(-z)}$	$g'(z) = g(z)(1 - g(z))$	
Tanh	$\tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$	$g'^{(z)} = 1 - g^2(z)$	
ReLU	$g(z) = \max(0, z)$	$g'(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases}$	
softplus	$g(z) = \ln(1 + e^z)$		

More activation functions

Leaky Relu	$g(z) = \begin{cases} \alpha z, & z < 0 \\ z, & z \geq 0 \end{cases}$	$g'(z) = \begin{cases} \alpha, & z < 0 \\ 1, & z \geq 0 \end{cases}$	
ELU	$g(z) = \begin{cases} z, & z > 0 \\ \alpha(e^z - 1), & z \leq 0 \end{cases}$	$g'(z) = \begin{cases} 1, & z > 0 \\ \alpha(e^z), & z \leq 0 \end{cases}$	
swish	$g(z) = z \cdot \sigma(\beta z)$	$g'(z) = \beta g(\beta z) + \sigma(\beta z)(1 - \beta g(\beta z))$	

Rectified Linear Units





- Activation function: $g(z) = \max\{0, z\}$ with $z \in \mathbb{R}$
- Give large and *consistent* gradients when active
- **Good practice:** Initialize **b** to a small positive value (e.g. 0.1) Ensures units are initially active for most inputs and derivatives can pass through

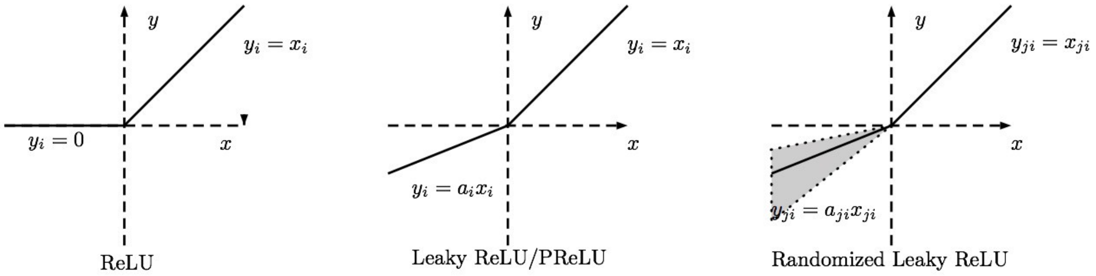
Positives:

- Gives large and *consistent* gradients (does not saturate) when active
- Efficient to optimize, converges much faster than sigmoid or tanh

Negatives:

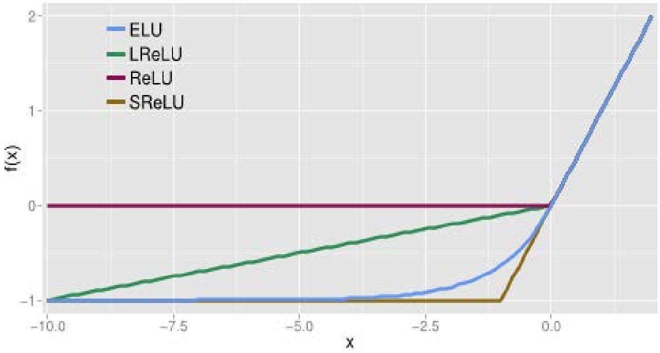
- Non zero centered output
- Units ”die” i.e. when inactive they will never update

Generalized Rectified Linear Units



- Get a non-zero slope when $z_i < 0$
- $g(z, \alpha)_i = \max\{0, z_i\} + \alpha_i \min\{0, z_i\}$
  - Absolute value rectification: $\alpha_i = 1$ gives $g(z) = |z|$
  - Leaky ReLU: Fix α_i to a small value e.g. 0.01
  - Parametric ReLU: Learn α_i
  - Randomized ReLU: Sample α_i from a fixed range during training, fix during testing
  - ....

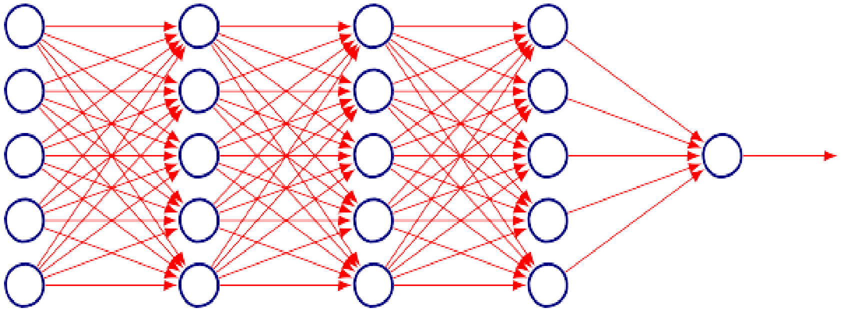
Exponential Linear Units (ELUs)



$$g(z) = \begin{cases} z, & z > 0 \\ \alpha(e^z - 1), & z \leq 0 \end{cases}$$

- All the benefits of ReLU + does not get killed
- Problem: Need to exponentiate

Universality and Depth



- First layer:

$$a^1 = g^1\left({W^1}^T x + b^1\right)$$
$$a^2 = g^2\left({W^2}^T a^1 + b^2\right)$$

- How do we decide depth, width?
- In theory how many layers suffice?

Universality

- Theoretical result [Cybenko, 1989]: 2-layer net with linear output with some squashing non-linearity in hidden units can approximate any continuous function over compact domain to arbitrary accuracy (given enough hidden units!)
- Implication: Regardless of function we are trying to learn, we know a large MLP can represent this function
- But not guaranteed that our training algorithm will be able to learn that function
- Gives no guidance on how large the network will be (exponential size in worst case)

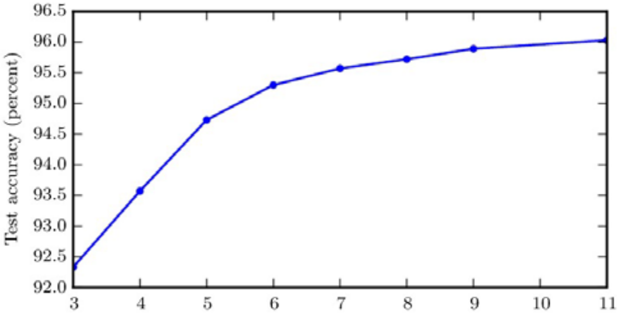
Some results

- (Montufar et al., 2014) Number of linear regions carved out by a deep rectifier network with d inputs, depth l and n units per hidden layer is:

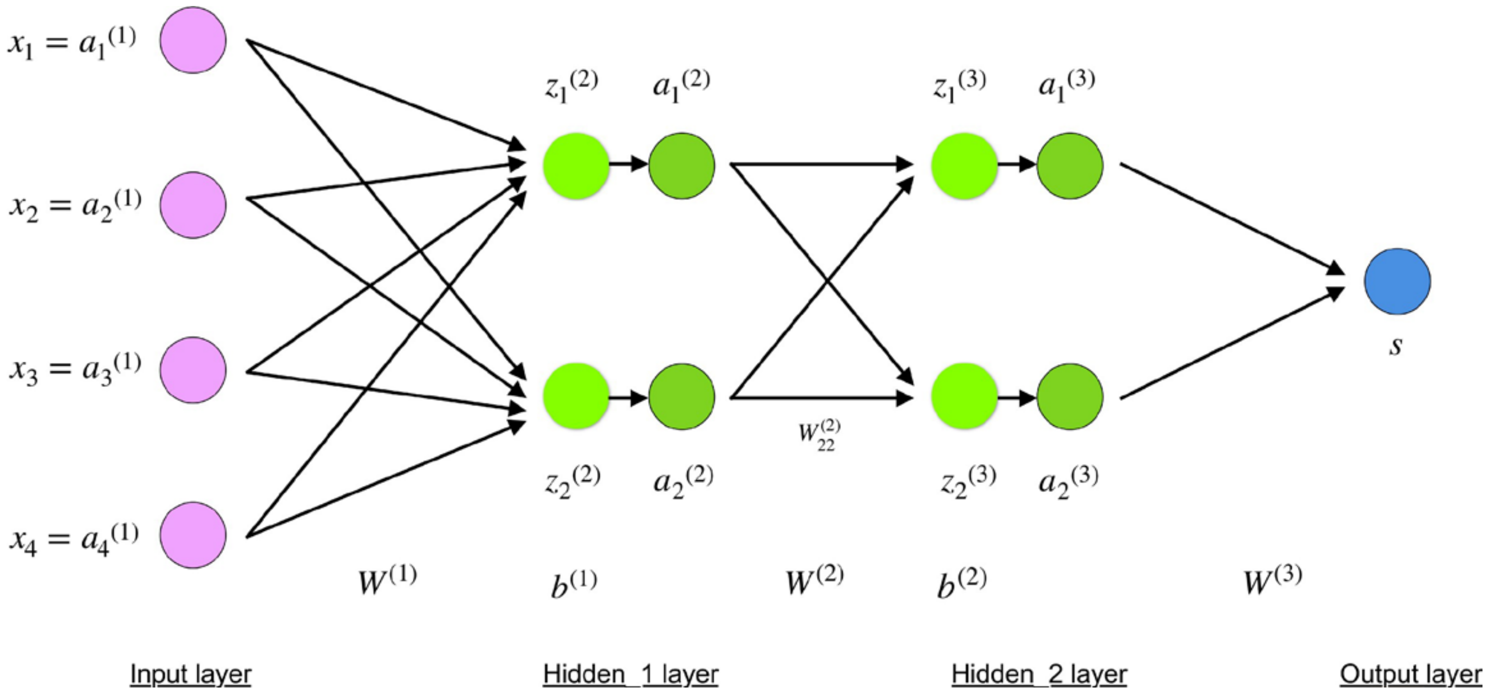
$$O\left(\binom{n}{d}^{d(1-1)} n^d\right)$$

- Exponential in depth!
- They showed functions representable with a deep rectifier network can require an exponential number of hidden units with a shallow network

Advantages of Depth

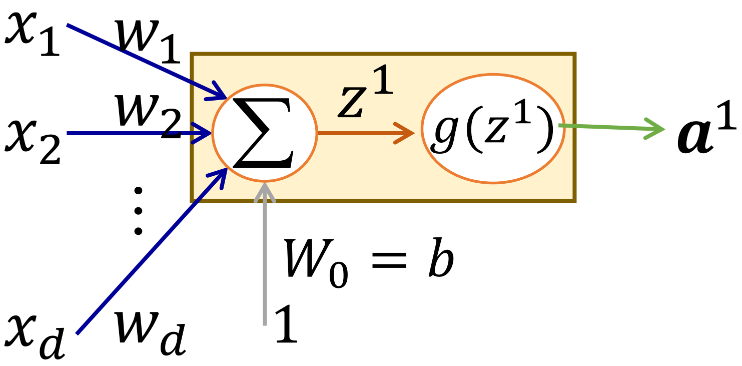


Multilayer Neural Network



https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd

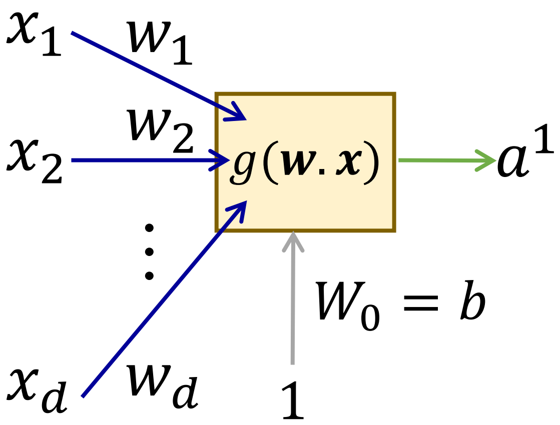
Basic Neural Units



$$z_1^1 = b_1^1 + \sum_{i=1}^{d} w_{1,i}^1 \, x_i = [\mathbf{w}_1^1 b_1^1] \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}$$

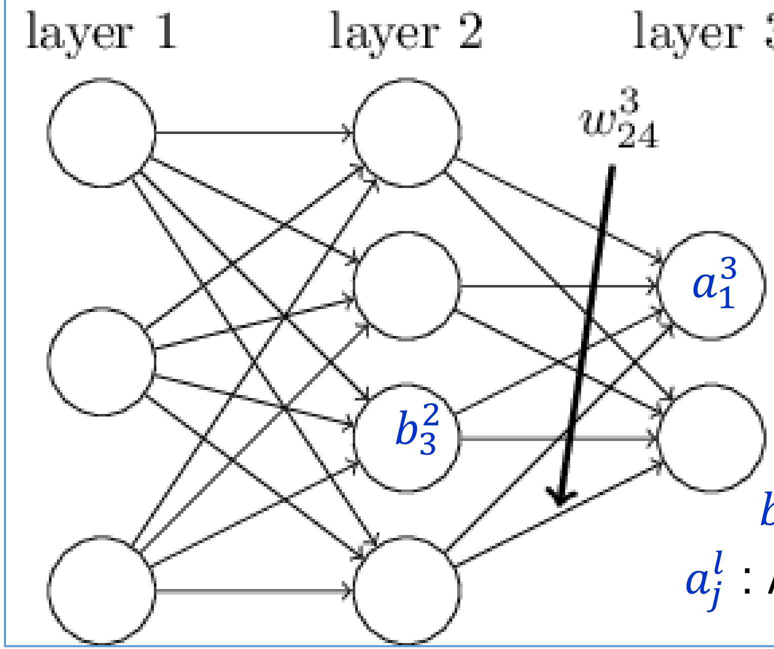
$$[w_{1,1}^1, w_{1,2}^1, \cdots, w_{1,d}^1]$$

$$[x_1 \; x_2 \ldots x_d]^T$$



$$a_1^1 = g(z_1^1)$$

Notations



w^l_{jk} is the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer

b^l_j : Bias for jth neuron in lth layer.

a^l_j : Activation of the jth neuron in the lth layer.

$$a^l_j = g\left(\sum_k w^l_{jk}\, a^{l-1}_k + b^l_j\right)$$

Vectorized form: $a^l = g\left(w^l a^{l-1} + b^l\right)$

$$z^l = w^l a^{l-1} + b^l$$

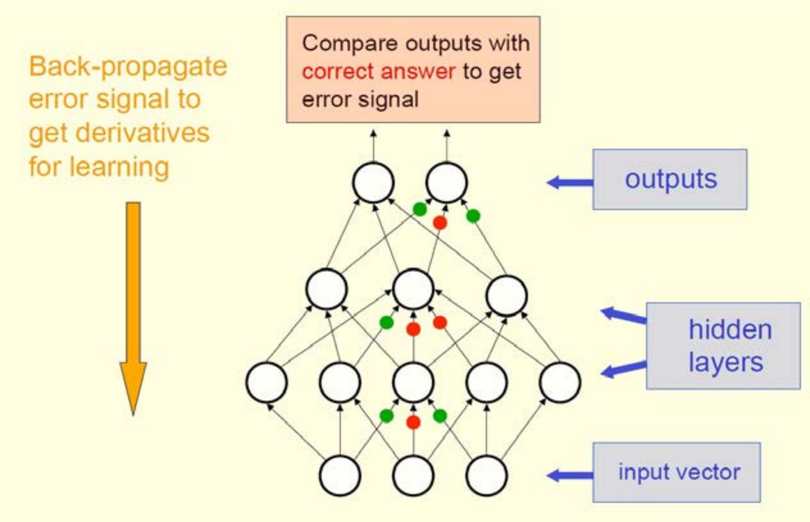
$$a^l = g\left(z^l\right)$$

Backpropagation

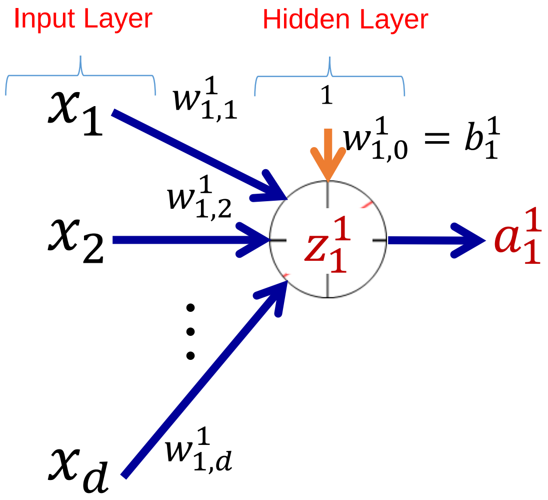
- Feedforward Propagation: Accept input $x^{(i)}$, pass through intermediate stages and obtain output $\hat{y}^{(i)}$
- During Training: Compute scalar cost J(θ)

$$J(\theta) = \sum_i L(NN(x^{(i)}; \theta), y^{(i)})$$

- Backpropagation allows information to flow backwards from cost to compute the gradient



Multilayer Neural Network

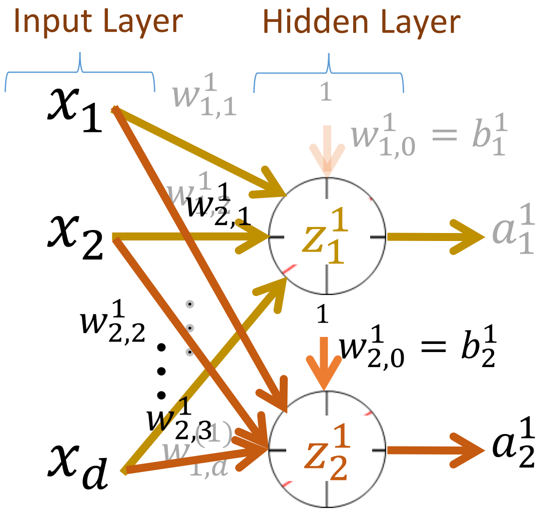


$$z^1_1 = b^1_1 + \sum_{i=1}^{d} w^1_{1,i}\, x_i = [\mathbf{w}^1_1 b^1_1] \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}$$

$$[x_1\ x_2 \ldots x_d]^T$$

$$a^1_1 = g(z^1_1)$$

Multilayer Neural Network



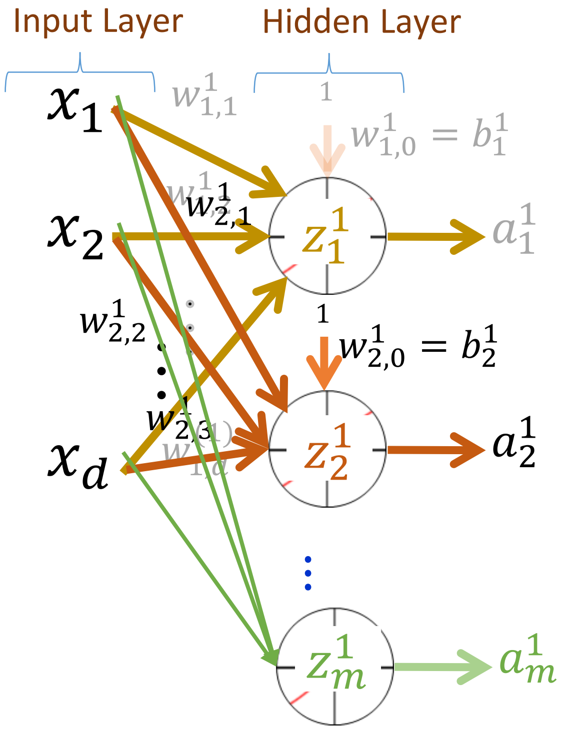
$$z_1^1 = b_1^1 + \sum_{i=1}^{d} w_{1,i}^1 \, x_i = [\mathbf{w}_1^1 b_1^1] \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix} \qquad a_1^1 = g(z_1^1)$$

$$z_2^1 = b_2^1 + \sum_{i=1}^{d} w_{2,i}^1 \, x_i = [\mathbf{w}_2^1 b_2^1] \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix} \qquad a_2^1 = g(z_2^1)$$

.. … …

$$z_m^1 = b_m^1 + \sum_{i=1}^{d} w_{m,i}^1 \, x_i = [\mathbf{w}_m^1 b_m^1] \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix} \; a_m^1 = g(z_m^1)$$

Multilayer Neural Network



$$z_1^1 = [\boldsymbol{w}_1^1 b_1^1]\begin{bmatrix}\boldsymbol{x}\\1\end{bmatrix}$$

$$z_2^1 = [\boldsymbol{w}_2^1 b_2^1]\begin{bmatrix}\boldsymbol{x}\\1\end{bmatrix}$$

$$\vdots$$

$$z_M^1 = [\boldsymbol{w}_m^1 b_2^1]\begin{bmatrix}\boldsymbol{x}\\1\end{bmatrix}$$

$$\begin{bmatrix}z_1^1\\z_2^1\\\vdots\\z_m^1\end{bmatrix} = \begin{bmatrix}\boldsymbol{w}_1^1 & b_1^1\\\boldsymbol{w}_2^1 & b_2^1\\\vdots & \vdots\\\boldsymbol{w}_m^1 & b_m^1\end{bmatrix}\begin{bmatrix}\boldsymbol{x}\\1\end{bmatrix}$$

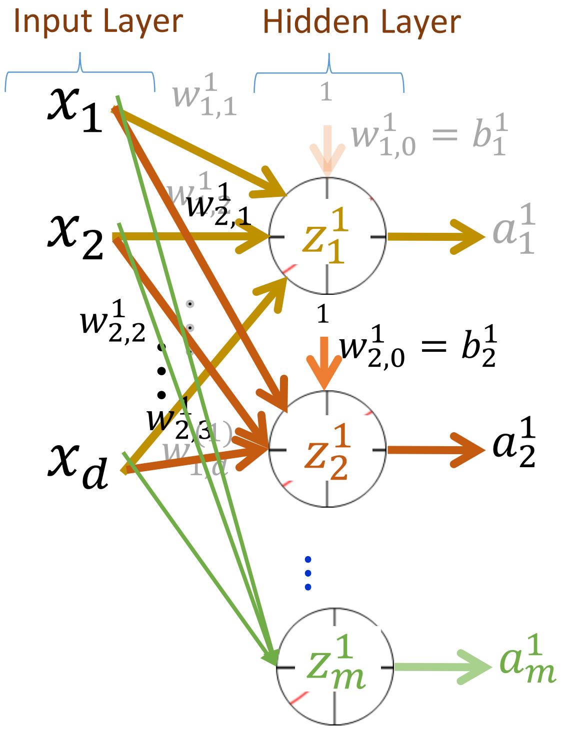
$$\boldsymbol{z}^1 = [\boldsymbol{W}^1\boldsymbol{b}^1]\begin{bmatrix}\boldsymbol{x}\\1\end{bmatrix}$$

$$\begin{bmatrix}a_1^1\\a_2^1\\\vdots\\a_m^1\end{bmatrix} = \begin{bmatrix}g(z_1^1)\\g(z_2^1)\\\vdots\\g(z_m^1)\end{bmatrix}$$

$$\boldsymbol{a}^1 = \boldsymbol{g}\left(\boldsymbol{z}^{(1)}\right)$$

$$a^{(0)} = x$$
$$z^{(1)} = \boldsymbol{w}^{(1)}\boldsymbol{a}^{(0)}$$
$$a^{(1)} = g\left(\boldsymbol{z}^{(1)}\right)$$

Multilayer Neural Network



$$z_1^1 = [\boldsymbol{w}_1^1 b_1^1] \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}$$

$$z_2^1 = [\boldsymbol{w}_2^1 b_2^1] \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}$$

$$\vdots$$

$$z_M^1 = [\boldsymbol{w}_m^1 b_2^1] \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} z_1^1 \\ z_2^1 \\ \vdots \\ z_m^1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{w}_1^1 & b_1^1 \\ \boldsymbol{w}_2^1 & b_2^1 \\ \vdots & \vdots \\ \boldsymbol{w}_m^1 & b_m^1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}$$

$$\boldsymbol{z}^1 = [\boldsymbol{W}^1 \boldsymbol{b}^1] \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}$$

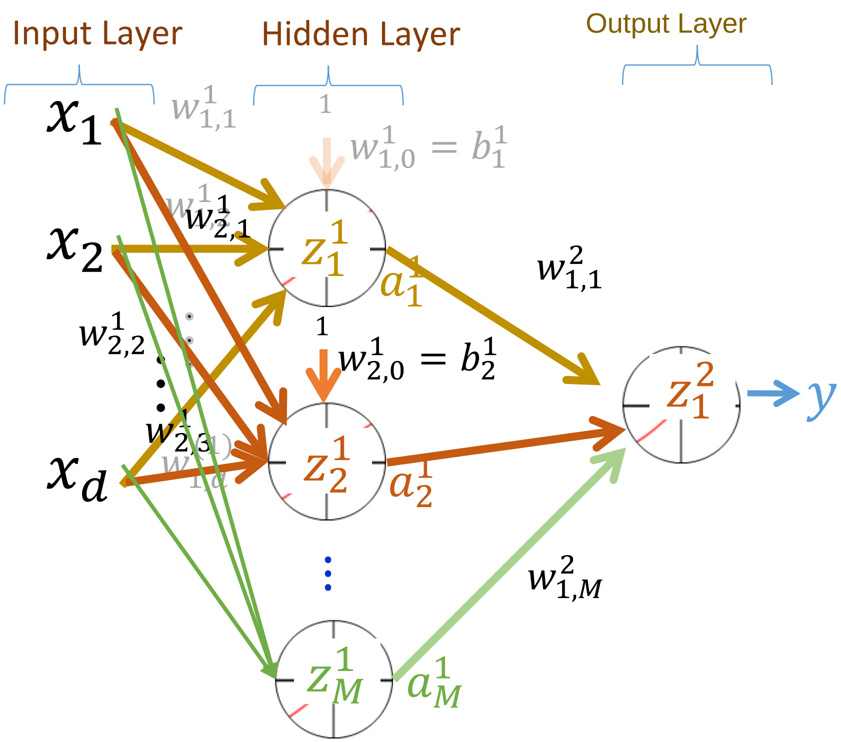
$$\begin{bmatrix} a_1^1 \\ a_2^1 \\ \vdots \\ a_m^1 \end{bmatrix} = \begin{bmatrix} g(z_1^1) \\ g(z_2^1) \\ \vdots \\ g(z_m^1) \end{bmatrix}$$

$$\boldsymbol{a}^1 = \boldsymbol{g}(\boldsymbol{z}^{(1)})$$

$$a^{(0)} = x$$
$$z^{(1)} = \boldsymbol{w}^{(1)} \boldsymbol{a}^{(0)}$$
$$a^{(1)} = g(\boldsymbol{z}^{(1)})$$

W^1 : $m \times n$ matrix
b^1 : $m \times 1$ column vector
X : $d \times 1$ column vector
Z^1 : $m \times 1$ column vector
A^1 : $m \times 1$ column vector

Multilayer Neural Network



Output Layer Pre-activation

$$z^2_1 = [\boldsymbol{w}^2_1 \; b^2_1] \begin{bmatrix} \boldsymbol{a}^1 \\ 1 \end{bmatrix}$$

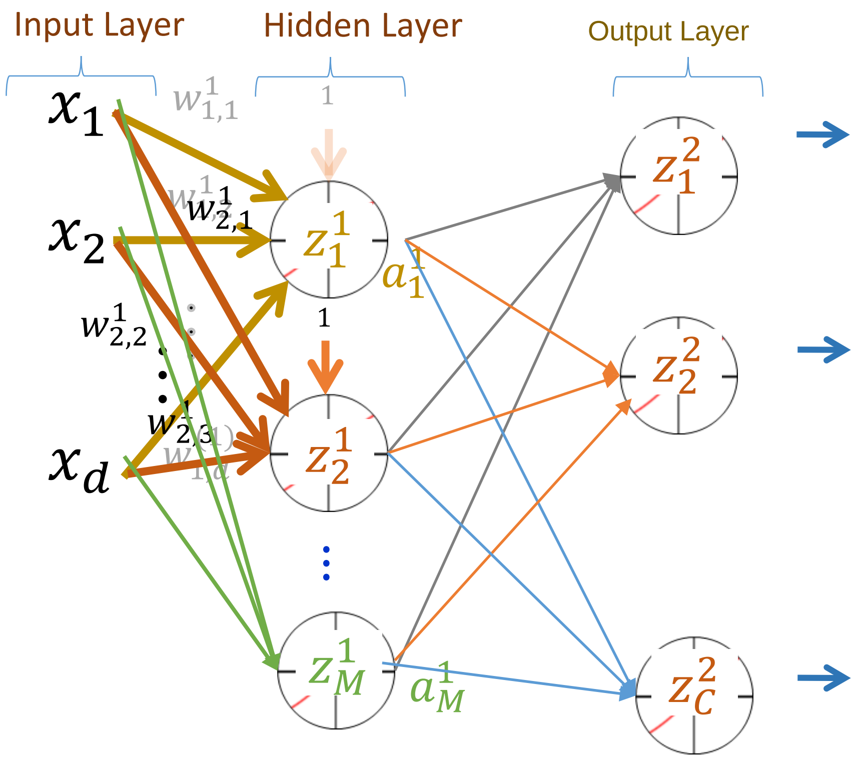
Output Layer Activation

$$y_1 = o\left(\boldsymbol{z}^2_1\right)$$

output

- Sigmoid for 2-class classification
- Softmax for multi-class classification
- Linear for regression

Multilayer Neural Network

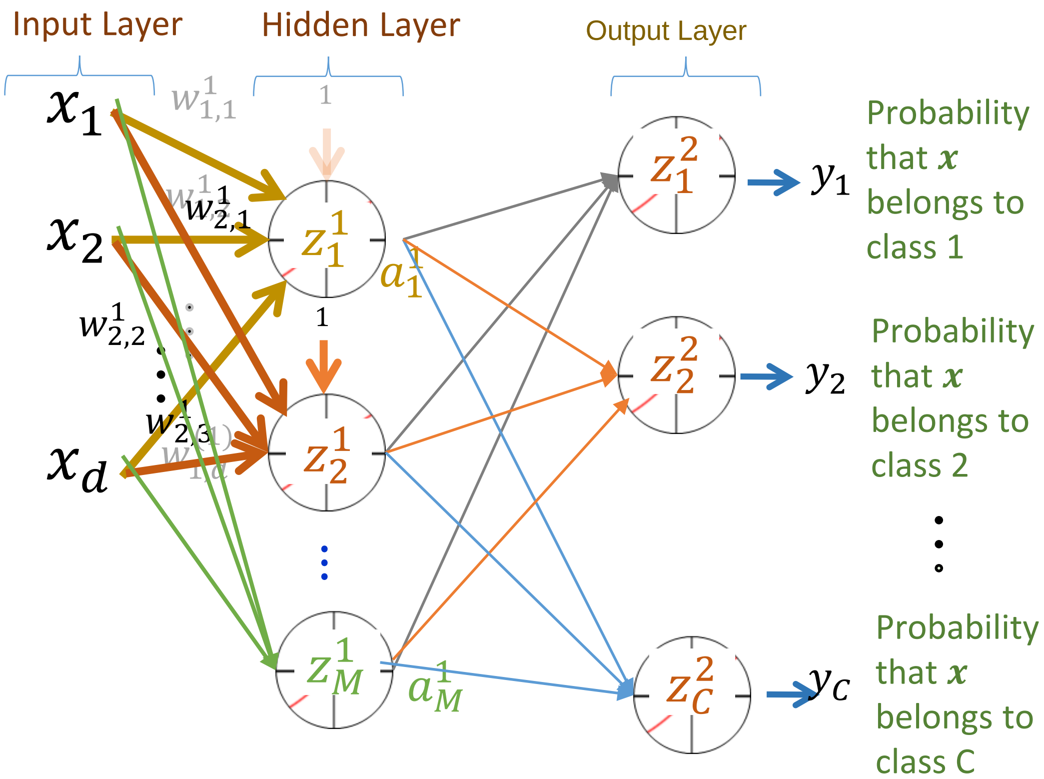


$$y_1 = o_1(z^2_1) = \frac{\exp(z^2_1)}{\sum_c \exp(z^2_c)}$$

$$y_2 = o_2(z^2_2) = \frac{\exp(z^2_2)}{\sum_c \exp(z^2_c)}$$

$$y_C = o_C(z^2_C) = \frac{\exp(z^2_C)}{\sum_c \exp(z^2_c)}$$

Training a Neural Network – Loss Function



Aim to maximize the probability corresponding to the correct class for any example $\boldsymbol{x}$

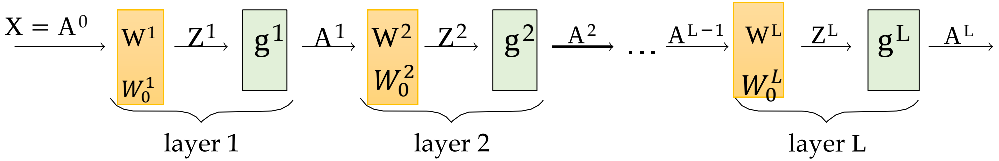
$$\max \boldsymbol{y}_c$$
$$\equiv \max\ (log\ \boldsymbol{y}_c)$$
$$\equiv \min\ (-log\ \boldsymbol{y}_c)$$

Can be equivalently expressed as

$$-\sum_i \prod_{i=c} \log(y_i)$$

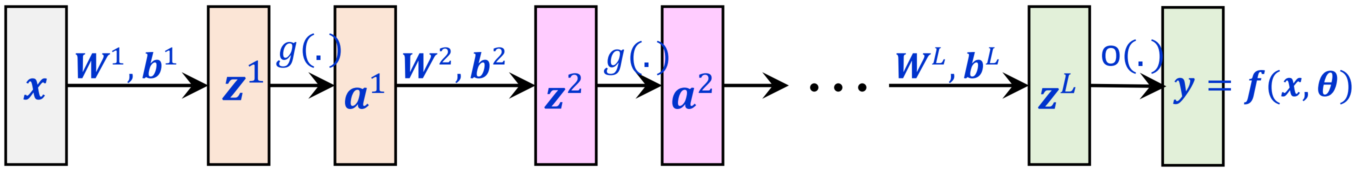
known as <u>cross-entropy loss</u>

Multi layered network



Forward Pass in a Nutshell

$\boldsymbol{\theta}$ is the collection of all learnable parameters i.e., all $\boldsymbol{W}$ and $\boldsymbol{b}$



Hidden layer pre-activation:

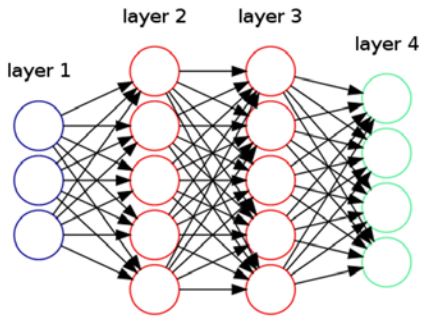
For $l = 1, \dots, L$; $\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}$

Hidden layer activation:

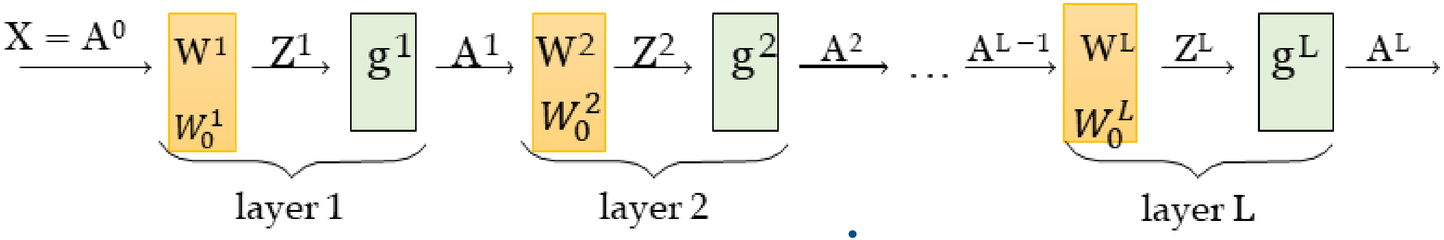
For $l = 1, \dots, L-1$; $\boldsymbol{a}^{(l)} = g(\boldsymbol{z}^{(l)})$

Output layer activation:

For $l = L$; $\boldsymbol{y} = \boldsymbol{a}^{(L)} = o(\boldsymbol{z}^{(L)}) = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\theta})$



Error back-propagation



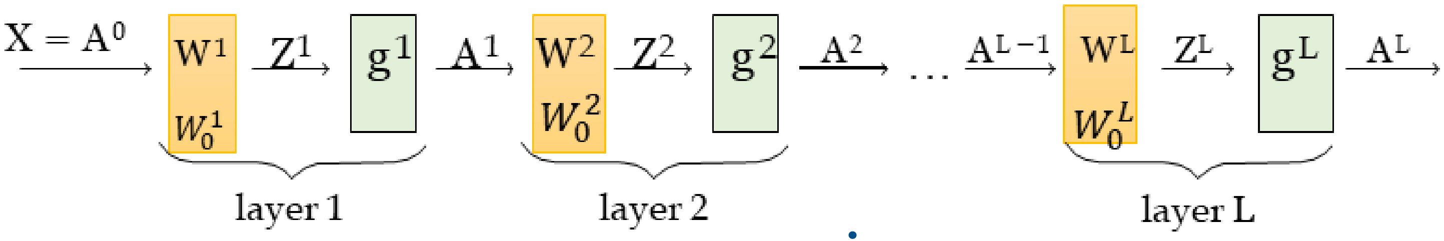
- We will train neural networks using gradient descent methods.
- To do SGD for a training example (x, y), we need to compute

$$\nabla_W Loss(NN(x;W),y)$$

where W represents all weights W^l, W_0^l in all the layers $l = (1, \dots, L)$.

$$\frac{\partial Loss}{\partial W^L} = \underbrace{\frac{\partial Loss}{\partial A^L}}_{\text{Depends on Loss function}} \cdot \underbrace{\frac{\partial A^L}{\partial Z^L}}_{g^{l\prime}} \cdot \underbrace{\frac{\partial Z^L}{\partial W^L}}_{A^{L-1}}$$

Error back-propagation



$$\frac{\partial Loss}{\partial W^L} = \frac{\partial Loss}{\partial A^L} \cdot \frac{\partial A^L}{\partial Z^L} \cdot \frac{\partial Z^L}{\partial W^L}$$

Depends on Loss function; $g^{l\prime}$; A^{L-1}

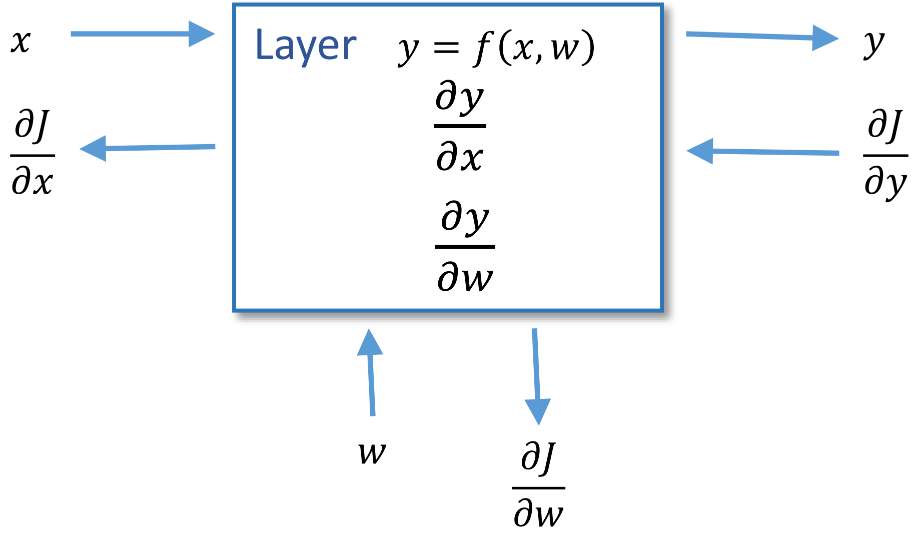
$$\frac{\partial Loss}{\partial W^l} = A^{l-1} \left(\frac{\partial Loss}{\partial Z^l}\right)^T$$

$m^l \times n^l \quad m^l \times 1 \quad 1 \times n^l$

So, in order to find the gradient of the loss with respect to the weights in the other layers of the network, we just need to be able to find $\frac{\partial Loss}{\partial Z^l}$

Backpropagation

- Compute derivatives per layer, utilizing previous derivatives
- Objective: $\text{Loss}(\boldsymbol{w})$
- Arbitrary layer: $y = f(x, w)$
- Need:
  - $\frac{\partial J}{\partial x} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial x}$
  - $\frac{\partial J}{\partial w} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial w}$



Calculus Chain Rule

- Scalar:
- $y = f(z)$
- $z = g(x)$
- $\frac{dy}{dx} = \frac{dy}{dz}\frac{dz}{dx}$

Multivariate:

$y = f(\mathbf{z})$

$\mathbf{z} = g(x)$

$$\frac{dy}{dx} = \sum_j \frac{\partial y}{\partial z_j}\frac{\partial z_j}{\partial x}$$

Multivariate:

$\mathbf{y} = f(\mathbf{z})$

$\mathbf{z} = g(\mathbf{x})$

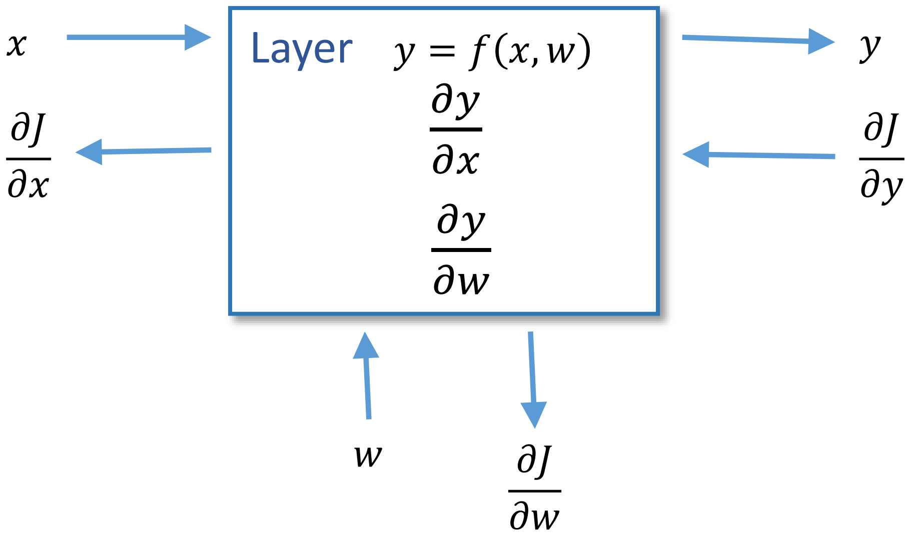
$$\frac{dy_{\mathrm{i}}}{dx_{\mathrm{k}}} = \sum_j \frac{\partial y_i}{\partial z_j}\frac{\partial z_j}{\partial x_{\mathrm{k}}}$$

Backpropagation (layerwise)

- Compute derivatives per layer, utilizing previous derivatives
- Objective: $J(\boldsymbol{w})$
- Arbitrary layer: $y = f(x, w)$
- Init:
  - $\frac{\partial J}{\partial x} = 0$
  - $\frac{\partial J}{\partial w} = 0$
- Compute:

.

  - $\frac{\partial J}{\partial x} \mathrel{+}= \frac{\partial J}{\partial y} \frac{\partial y}{\partial x}$
  - $\frac{\partial J}{\partial w} \mathrel{+}= \frac{\partial J}{\partial y} \frac{\partial y}{\partial w}$



Informal Derivation: Application of Chain Rule

$$\frac{\partial Loss}{\partial Z^1} = \underbrace{\underbrace{\frac{\partial Loss}{\partial A^L} \cdot \frac{\partial A^L}{\partial Z^L} \cdot \frac{\partial Z^L}{\partial A^{L-1}} \cdot \frac{\partial A^{L-1}}{\partial Z^{L-1}} \cdot \cdots \cdot \frac{\partial A^2}{\partial Z^2}}_{\frac{\partial Loss}{\partial Z^2}} \cdot \frac{\partial Z^2}{\partial A^1}}_{\frac{\partial Loss}{\partial A^1}} \cdot \frac{\partial A^1}{\partial Z^1}$$

$\frac{\partial Loss}{\partial A^L}$ is $n^L \times 1$

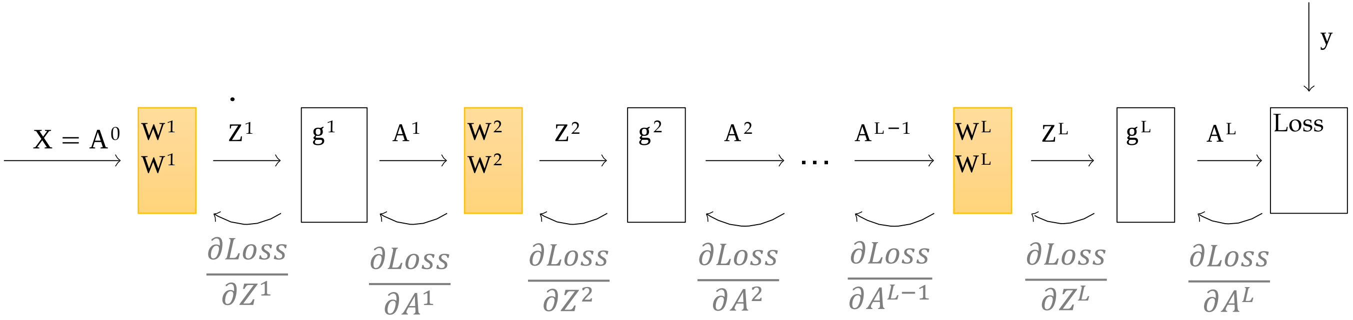
$\frac{\partial Z^l}{\partial A^{l-1}}$ is $m^l \times n^l$ and is just W^l

$\frac{\partial A^l}{\partial Z^l}$ is $n^l \times n^l$. Each element $a_i^l = g^l(z_i^l)$. This means that $\frac{\partial a_i^l}{\partial z_j^l} = 0$ whenever $i \neq j$. So, the off-diagonal elements all 0, and the diagonal elements are $\frac{\partial a_i^l}{\partial z_j^l} = g^{l\prime}(z_j^l)$

Rewrite the equation

$$\frac{\partial Loss}{\partial Z^1} = \frac{\partial Loss}{\partial A^L} \cdot \frac{\partial A^L}{\partial Z^L} \cdot \frac{\partial Z^L}{\partial A^{L-1}} \cdot \frac{\partial A^{L-1}}{\partial Z^{L-1}} \cdot \cdots \cdot \frac{\partial A^2}{\partial Z^2} \cdot \frac{\partial Z^2}{\partial A^1} \cdot \frac{\partial A^1}{\partial Z^1}$$

$$\frac{\partial Loss}{\partial Z^1} = \frac{\partial A^l}{\partial Z^l} \cdot W^{l+1} \cdot \frac{\partial A^{l+1}}{\partial Z^{l+1}} \cdot \cdots \cdot W^{L-1} \cdot \frac{\partial A^{L-1}}{\partial Z^{L-1}} \cdot W^L \cdot \frac{\partial A^L}{\partial Z^L} \cdot \frac{\partial Loss}{\partial A^L}$$



```
SGD-NEURAL-NET(D_n, T, L, (m^1, ..., m^L), (f^1, ..., f^L))
for l = 1 to L
    W^l_ij ~ Gaussian(0, 1/m^l)
    W^l_0j ~ Gaussian(0, 1)
for t = 1 to T
    i = random sample from {1, ..., n}
    A^0 = x^(i)
    // forward pass to compute the output A^L
    for l = 1 to L
        Z^l = W^lT A^(l-1) + W^l_0
        A^l = f^l(Z^l)
    loss = Loss(A^L, y^(i))
    for l = L to 1:
        // error back-propagation
        ∂loss/∂A^l = if l < L then ∂loss/∂Z^(l+1) · ∂Z^(l+1)/∂A^l else ∂loss/∂A^L
        ∂loss/∂Z^l = ∂loss/∂A^l · ∂A^l/∂Z^l
        // compute gradient with respect to weights
        ∂loss/∂W^l = ∂loss/∂Z^l · ∂Z^l/∂W^l
        ∂loss/∂W^l_0 = ∂loss/∂Z^l · ∂Z^l/∂W^l_0
        // stochastic gradient descent update
        W^l = W^l − η(t) · ∂loss/∂W^l
        W^l_0 = W^l_0 − η(t) · ∂loss/∂W^l_0
```

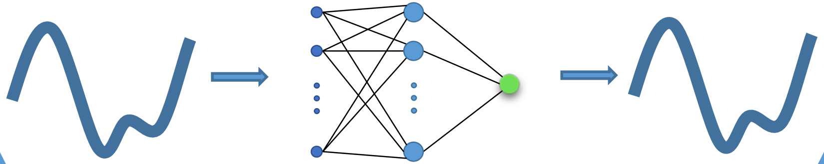
Neural Networks Properties

- Practical considerations
  - Large number of neurons → Danger for overfitting
  - Gradient descent can easily get stuck local optima

- Universal Approximation Theorem:

▪ A two-layer neural network with a sufficient number of neurons can approximate any continuous function to any desired accuracy.

Universal Approximation

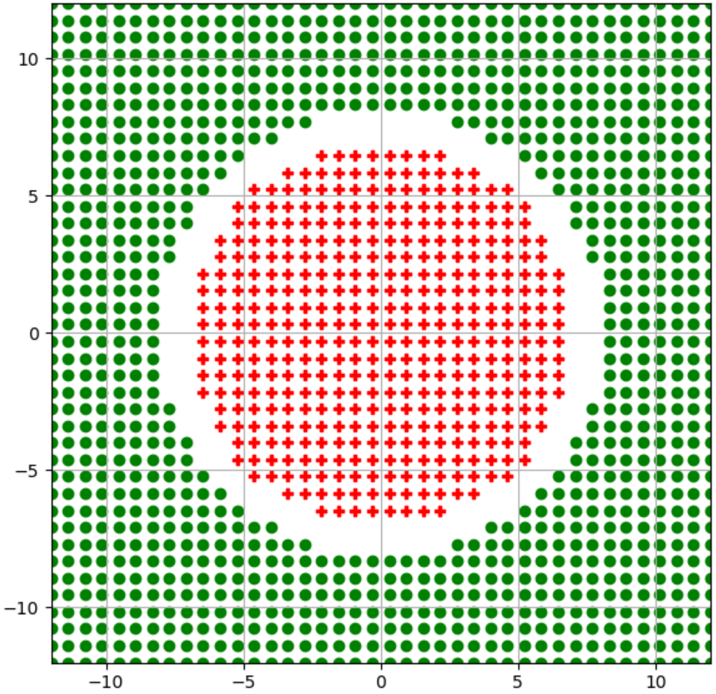


Theorem

A visual proof that neural nets can compute any function

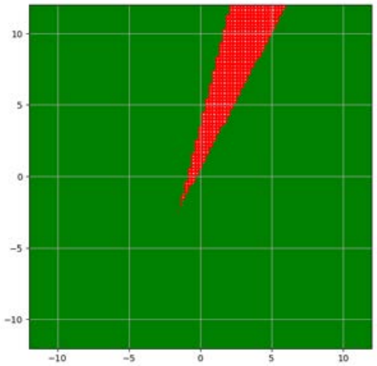
- http://neuralnetworksanddeeplearning.com/chap4.html

Training Multilayer Neural Network for non-linearly Separable Data

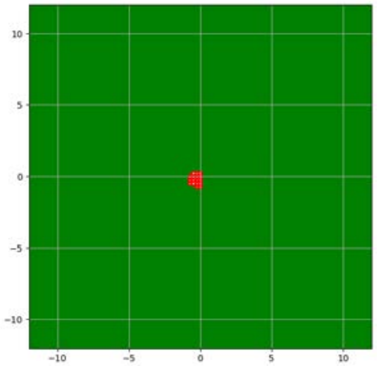


Training Data

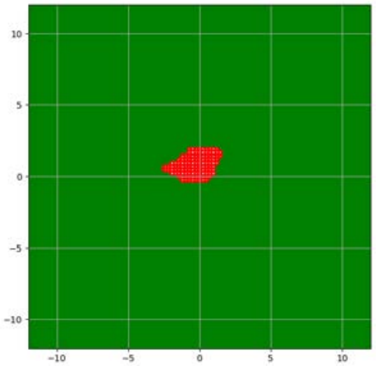
Learned Decision Boundary with Single Hidden Layer



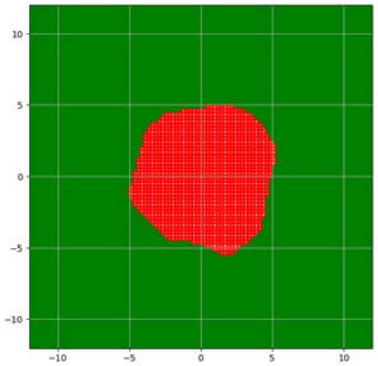
# of hidden neurons = 2



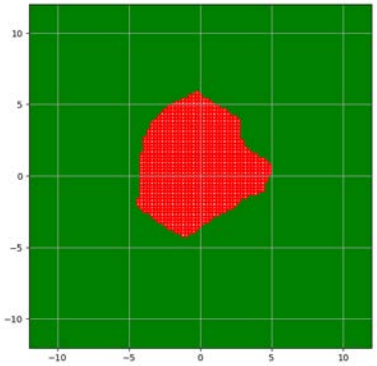
# of hidden neurons = 4



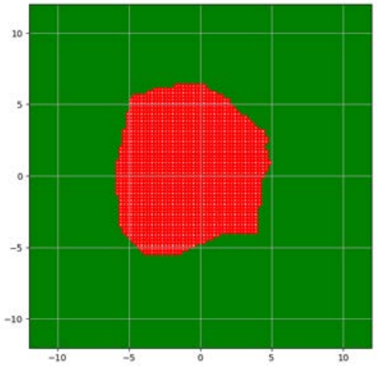
# of hidden neurons = 8



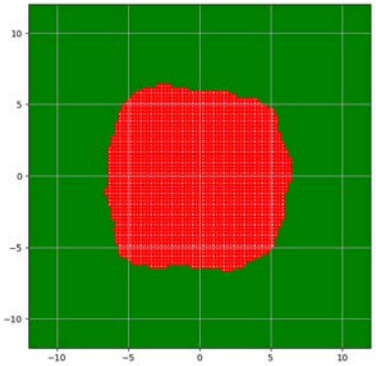
# of hidden neurons = 16



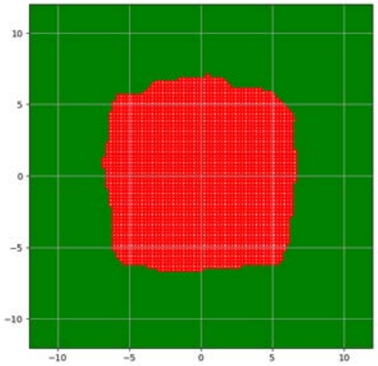
# of hidden neurons = 32



# of hidden neurons = 64

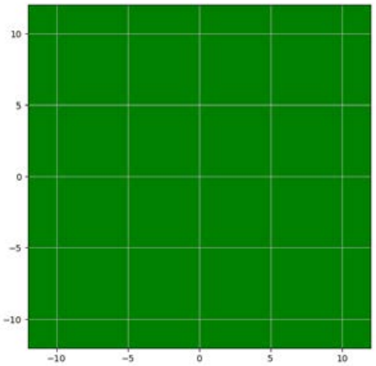


# of hidden neurons = 128

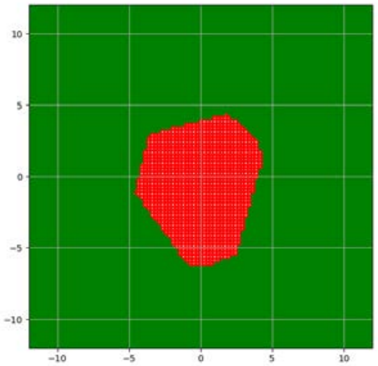


# of hidden neurons = 256

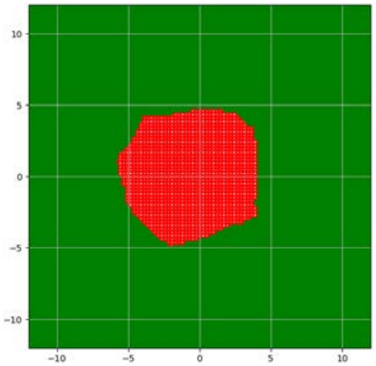
Learned Decision Boundary with Two Hidden Layers



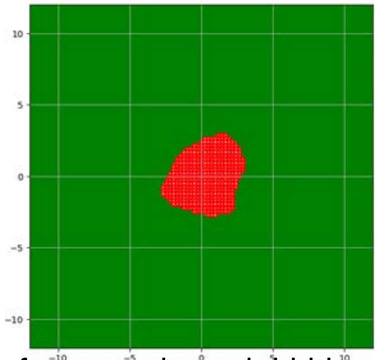
# of neurons in each hidden layer = 2



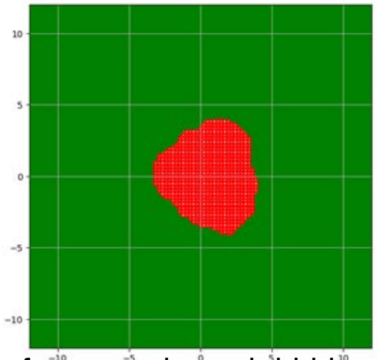
# of neurons in each hidden layer = 4



# of neurons in each hidden layer = 8



# of neurons in each hidden layer = 16



# of neurons in each hidden layer = 32