

# Introduction to Deep Learning (IT3320E)

L4, L5 - Training Neural Networks

Hung Son Nguyen

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY  
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Oct. 4-5, 2022

## 1 Universal Approximation Theorem (UAT)

## 2 Training CNN

- Hyperparameters
- Activation functions  $h^k = f(W^k * x + b^k)$
- Loss functions
- Design of algorithms (backpropagation)
- Weight Initialization In Deep Neural Networks
- Regularization to CNN
- Optimizer selection

## General view:

- Gather training data  $(x_1, y_1), \dots, (x_n, y_n)$
- Choose a family of functions, e.g.,  $\mathcal{H}$ , so that there is  $f \in \mathcal{H}$  to ensure  $y_i \approx f(x_i)$  for all  $i = 1, \dots, n$
- setup a loss function  $\ell$
- find  $f \in \mathcal{H}$  to minimize the average loss

$$\min_{f \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^n \ell(y_i, f(x_i))$$

## Neural network view:

- Gather training data  $(x_1, y_1), \dots, (x_n, y_n)$
- Choose a NN with  $k$  neurons, so that there is a group of weights, e.g.,  $(\mathbf{w}_1, \dots, \mathbf{w}_k, b_1, \dots, b_k)$ , to ensure

$$y_i \approx NN_{\mathbf{w}_1, \dots, \mathbf{w}_k, b_1, \dots, b_k}(x_i)$$

for all  $i = 1, \dots, n$

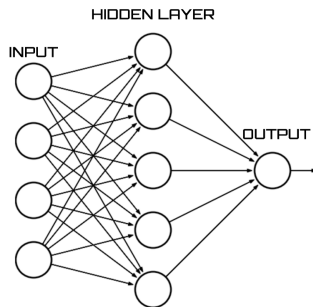
- setup a loss function  $\ell$
- find weights  $\mathbf{w}_1, \dots, \mathbf{w}_k, b_1, \dots, b_k$  to minimize the average loss

$$\min_{\mathbf{w}, \mathbf{b}} \frac{1}{n} \sum_{i=1}^n \ell(y_i, NN_{\mathbf{w}_1, \dots, \mathbf{w}_k, b_1, \dots, b_k}(x_i))$$

## Section 1

# **Universal Approximation Theorem (UAT)**

---



Considers single-hidden layer neural networks of the form

$$\mathbf{x} \mapsto \sum_{k=1}^K v_k \rho(\mathbf{w}_k^T \mathbf{x} - b_k), \quad (1)$$

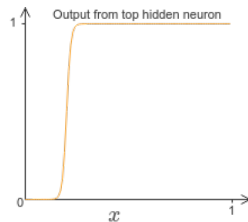
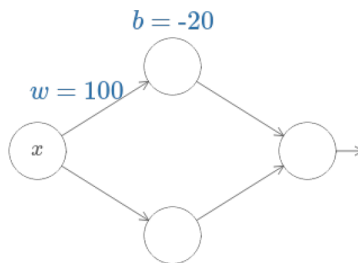
where  $\rho : \mathbb{R} \rightarrow \mathbb{R}$  is a fixed activation function and, for  $k = 1, \dots, K$   $v_k \in \mathbb{R}$  and  $\mathbf{w}_k \in \mathbb{R}^d$  are the weights of the network and  $b_k \in \mathbb{R}$  are the biases of the network.

- The **universal approximation theorem** then says that just about any choice of activation function  $\rho$ , one can **uniformly approximate** arbitrarily well any given continuous function with functions of the form in (1).
- It is important to note that this result does not assume that the width  $K$  is fixed, i.e.,  $K$  can be (and must be) arbitrarily large.
- In other words, there **does not exist** a fixed  $K$  such that every continuous function can be uniformly approximated by functions of the form in (1).

Consider functions  $f : \mathbb{R} \rightarrow \mathbb{R}$  and  $\sigma = \frac{1}{1 + e^{-x}}$

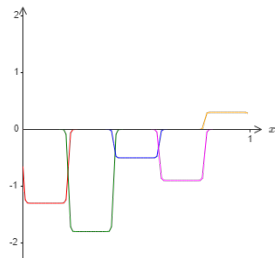
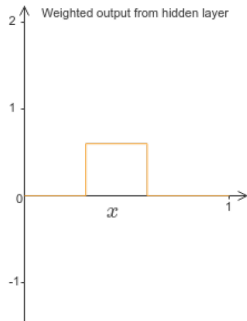
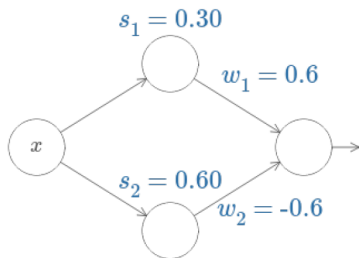
**Step 1:** Building a step function

$$\begin{aligned} y &= \frac{1}{1 + e^{-(wx+b)}} \\ &= \frac{1}{1 + e^{-w(x+b/w)}} \end{aligned}$$



- Larger  $w$ , sharper transition
- Transition around  $-b/w$ , written as  $s$

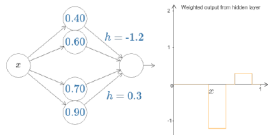
## Step 2: build bump functions



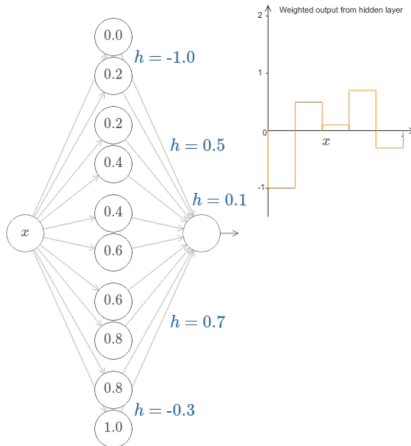
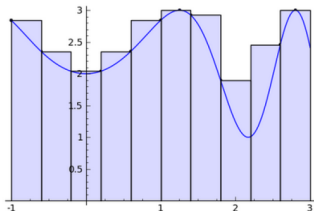
$$0.6 \times \text{step}(0.3) - 0.6 \times \text{step}(0.6)$$

Question: how to control the bump height  $h$ ?

## Step 3: sum up bumps to approximate



ultimate idea ... familiar?



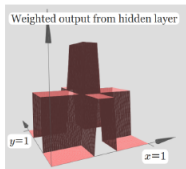
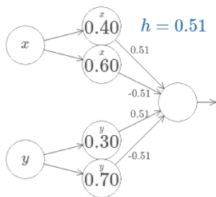
**Message:** all functions  $f : \mathbb{R} \rightarrow \mathbb{R}$  can be “well” approximated using 2-layer NN’s



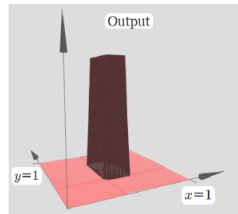
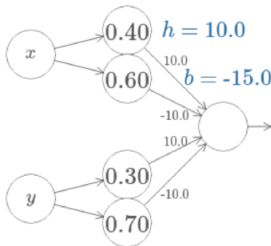
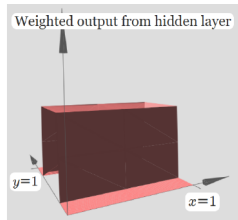
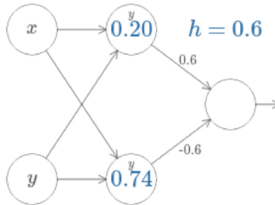
# What about high-dimensional?

## Similar story:

- Step 1: Build “step” functions
- Step 2: Build “bump” functions
- Step 3: Build “tower” functions
- Step 4: Sum up bumps to approximate

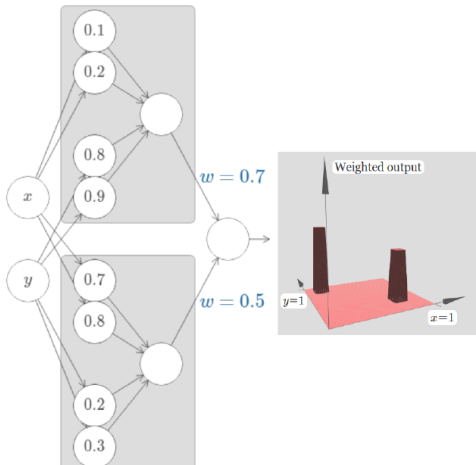


sum up  $x$ ,  $y$  bumps to obtain a  
stair tower

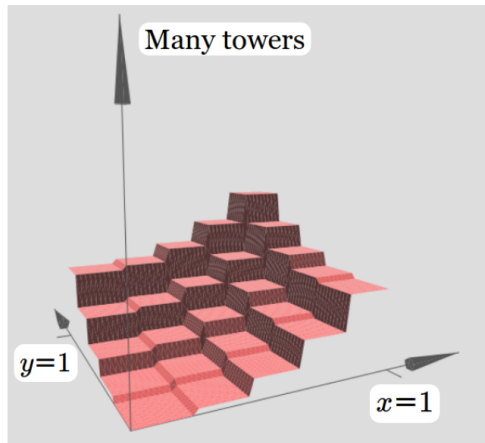


threshold to obtain a sharp tower

## Step 4: sum up towers for approximation



sum up two towers



sum up many towers

**Message:** all functions  $f : \mathbb{R}^2 \rightarrow \mathbb{R}$  can be “well” approximated using **3-layer NN’s**

**Question:** Is it possible to use 2-layer NNs only?

## Theorem (Cybenko UAT, 1989)

Let  $C([0, 1]^n)$  denote the set of all continuous function  $[0, 1]^n \rightarrow \mathbb{R}$ , let  $\sigma : \mathbb{R} \rightarrow \mathbb{R}$  be a nonconstant, bounded, and continuous function (*sigmoidal activation function*). Then given any  $\varepsilon > 0$  and any function  $f \in C([0, 1]^n)$ , there exist an integer  $K$ , real constants  $v_i, b_i \in \mathbb{R}$  and real vectors  $\mathbf{w}_i \in \mathbb{R}^m$  for  $i = 1, \dots, N$ , such that we may define:

$$F(\mathbf{x}) = \sum_{i=1}^K v_i \sigma(\mathbf{w}_i \cdot \mathbf{x} + b_i) = \mathbf{v}^T \cdot \sigma(\mathbf{W}^T \cdot \mathbf{x} + \mathbf{b})$$

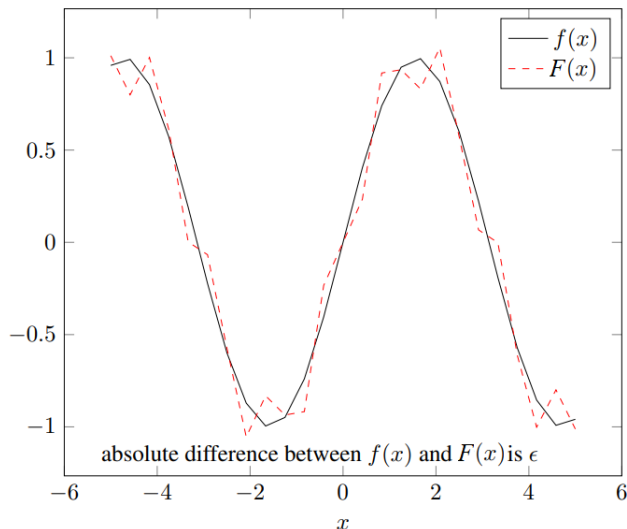
as an approximate realization of the function  $f$ ; that is,

$$\|F(\mathbf{x}) - f(\mathbf{x})\| < \varepsilon$$

for all  $\mathbf{x} \in [0, 1]^n$

Visual “proof” (<http://neuralnetworksanddeeplearning.com/chap4.html>)

Remark: then we say that the class of all the finite sum of the form  $f(x) = \sum_{i=1}^N \alpha_i \sigma(\mathbf{w}_i x + b_i)$  is **dense** in  $C([0, 1]^n)$ .



**Theorem (Leshno et al., 1993)**

Let  $\rho : \mathbb{R} \rightarrow \mathbb{R}$  be any continuous function and let  $\Omega \subset \mathbb{R}^d$  be compact. Then, the set

$$1\text{NN}_\rho := \text{span}\{\mathbf{x} \mapsto \rho(\mathbf{w}^\top \mathbf{x} - b) : \mathbf{w} \in \mathbb{R}^d, b \in \mathbb{R}\}$$

is dense (with respect to the uniform norm) in  $C(\Omega)$ , the space of continuous functions on  $\Omega$ , if and only if  $\rho$  is not an *algebraic polynomial*.

### **Theorem (Komogorov Representation Theorem, 1956)**

*Let  $d \geq 2$ . Then, there exist constants  $\lambda_q > 0$ , for  $q = 1, \dots, d$  and continuous functions  $\phi_p : [0, 1] \rightarrow [0, 1]$ ,  $p = 1, \dots, 2d + 1$ , such that every continuous function  $f : [0, 1]^d \mapsto \mathbb{R}$  admits the representation*

$$f(\mathbf{x}) = f(x_1, \dots, x_d) = \sum_{p=1}^{2d+1} g \left( \sum_{q=1}^d \lambda_q \phi_q(x_q) \right), \quad (2)$$

*for some  $g \in C[0, 1]$  depending on  $f$ .*

### Theorem (Maierov and Pinkus, 1999)

Let  $\Omega \subset \mathbb{R}^d$  be compact. There exists  $K_1$  and  $K_2$  being  $O(d)$  and  $\rho : \mathbb{R} \rightarrow \mathbb{R}$  such that the set of two-hidden layer neural networks with fixed hidden layer widths

$$2\text{NN}_{\rho}^{K_1, K_2} := \left\{ \mathbf{x} \mapsto \sum_{k=1}^{K_2} u_k \rho \left( \sum_{\ell=1}^{K_1} v_{k\ell} \rho(\mathbf{w}_{k\ell}^T \mathbf{x} - b_{k\ell}) - c_k \right) : u_k \in \mathbb{R}, v_{k\ell} \in \mathbb{R}, \mathbf{w}_{k\ell} \in \mathbb{R}^d, b_{k\ell} \in \mathbb{R}, c_k \in \mathbb{R} \right\}.$$

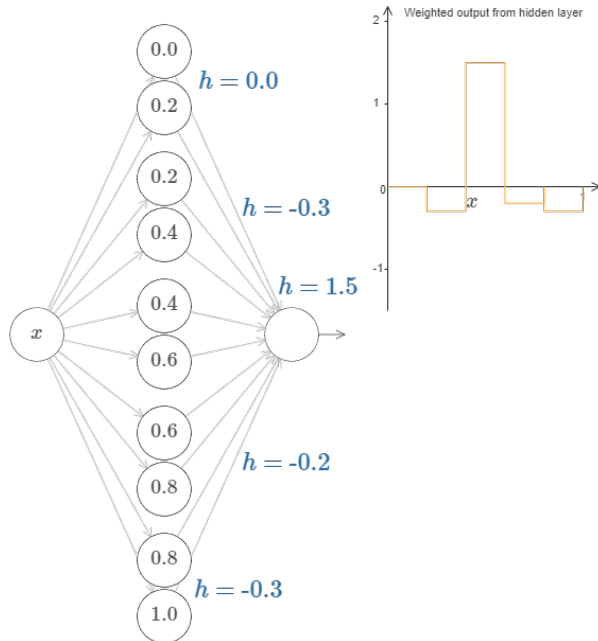
is dense in  $C(\Omega)$ .

In other words, given  $\varepsilon > 0$ , for every  $f \in C(\Omega)$ , there exists a two-hidden layer neural network  $h \in 2\text{NN}_{\rho}^{K_1, K_2}$  such that

$$\|f - h\|_{L^\infty(\Omega)} < \varepsilon$$

In particular,  $K_1 = d$  and  $K_2 = 2d + 1$

- Universality tells us that neural networks can compute any function;
- But, how many terms in the summation (or how many nodes in the network layer) are needed to yield an approximation of given quality?
- empirical evidence suggests that deep networks are the networks best adapted to learn the functions useful in solving many real-world problems.



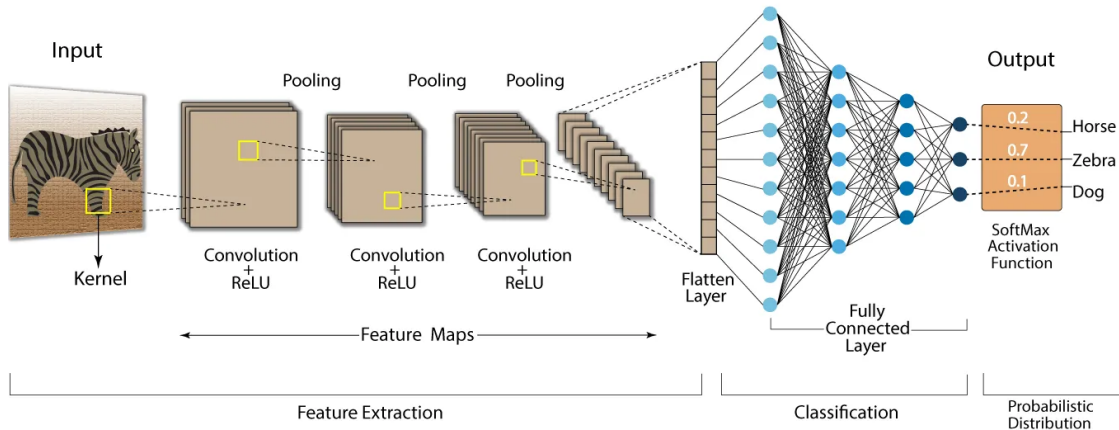


## Section 2

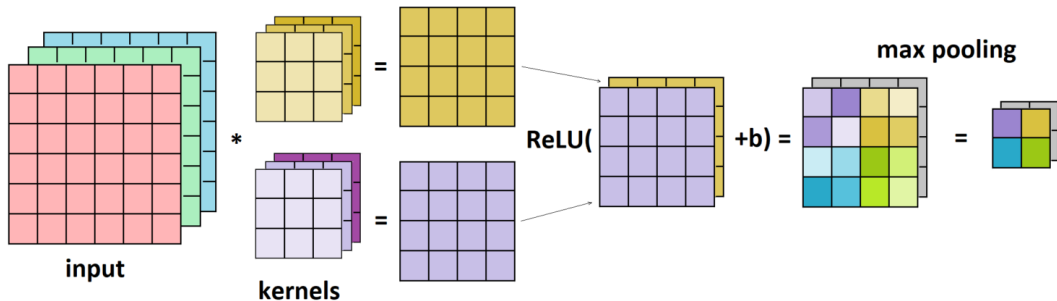
# **Training CNN**

---

## Convolution Neural Network (CNN)



A commonly used type of CNN consists of numerous **convolution layers**, **activation layers (non-linearity)** preceding **sub-sampling (pooling) layers**, while the ending layers are **fully connected (FC)** layers.

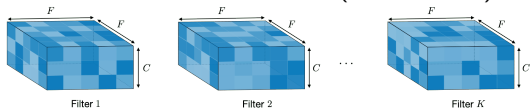


## 1 Universal Approximation Theorem (UAT)

## 2 Training CNN

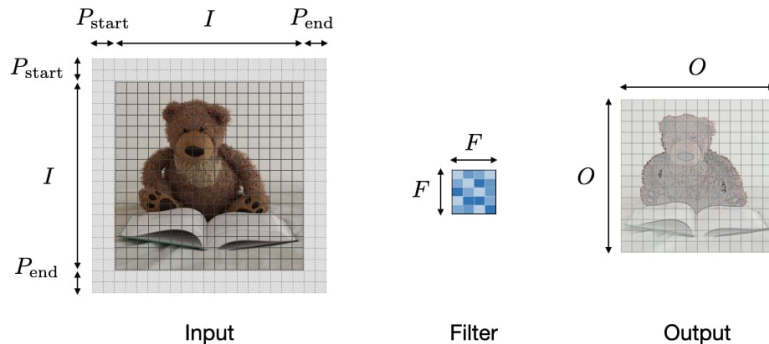
- Hyperparameters
- Activation functions  $h^k = f(W^k * x + b^k)$
- Loss functions
- Design of algorithms (backpropagation)
- Weight Initialization In Deep Neural Networks
- Regularization to CNN
- Optimizer selection

- The input  $x$  of each layer in a CNN model is organized in three dimensions: height, width, and depth, or  $m \times m \times r$ , where the height ( $m$ ) is equal to the width. The depth is also referred to as the channel number.
- For example, in an RGB image, the depth ( $r$ ) is equal to three.
- Several kernels (filters) available in each convolutional layer are denoted by  $K$  and also have three dimensions ( $F \times F \times C$ ), similar to the input image;



## Remarks:

- $F$  must be smaller than  $m$ , while  $C$  is either equal to or smaller than  $r$ .
- the application of  $K$  filters of size  $F \times F$  results in an output feature map of size  $O \times O \times K$ .



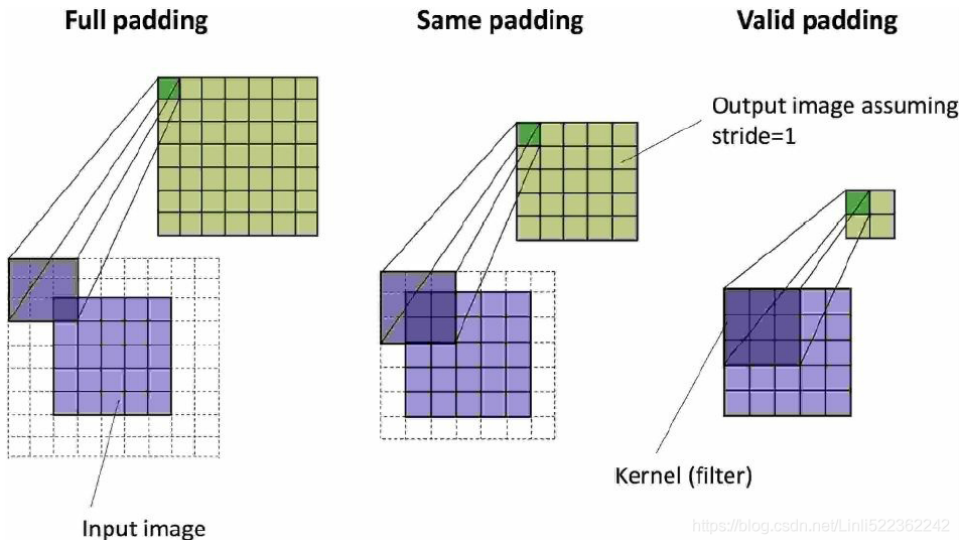
- If  $I$  is the length of the input volume size,  $F$  – the length of the filter,  $P_{start}$ ,  $P_{end}$  – the amounts of zero padding,  $S$  – the stride,
- then the output size  $O$  of the feature map along that dimension is given by:

$$O = \frac{I - F + P_{start} + P_{end}}{S} + 1$$

It is important to know the meaning behind the hyperparameters of the filters.

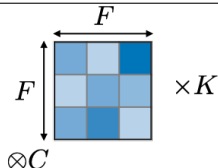
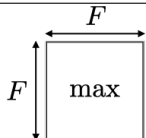
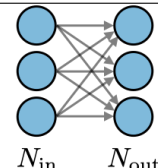
- **Dimensions of a filter:** A filter of size  $F \times F$  applied to an input containing  $C$  channels is a  $F \times F \times C$  tensor that performs convolutions on an input of size  $I \times I \times C$  and produces an output feature map (also called activation map) of size  $O \times O \times 1$ .
- **Stride:** For a convolutional or a pooling operation, the stride  $S$  denotes the number of pixels by which the window moves after each operation.
- **Zero padding:** Zero-padding denotes the process of adding  $P$  zeroes to each side of the boundaries of the input.

This value can either be manually specified or automatically set through one of the three modes detailed below:





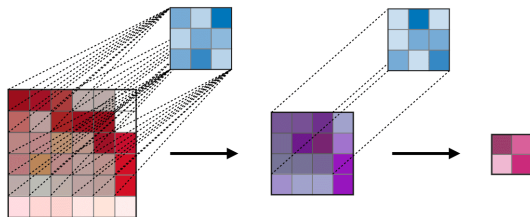
MODE	VALUE	PURPOSE
<b>Full</b>	$P_{\text{start}} = P_{\text{end}} = F - 1$	<ul style="list-style-type: none"> <li>• The <b>maximum</b> padding such that end convolutions are applied on the limits of the input.</li> <li>• Filter 'sees' the input end-to-end</li> </ul>
<b>Same</b>	$P_{\text{start}} = \left\lfloor \frac{S \lceil \frac{I}{S} \rceil - I + F - S}{2} \right\rfloor$ $P_{\text{end}} = \left\lceil \frac{S \lceil \frac{I}{S} \rceil - I + F - S}{2} \right\rceil$	<ul style="list-style-type: none"> <li>• Padding such that feature map size has size <math>\lceil \frac{I}{S} \rceil</math>.</li> <li>• If <math>S = 1</math>, we add padding layers such that the output image has <b>the same dimensions</b> as the input image.</li> <li>• Also called 'half' padding</li> </ul>
<b>Valid</b>	$P = 0$	<ul style="list-style-type: none"> <li>• It implies no padding at all.</li> <li>• The input image is left in its valid/unaltered shape.</li> </ul>

	CONV	POOL	FC
Illustration			
Input size	$I \times I \times C$	$I \times I \times C$	$N_{in}$
Output size	$O \times O \times K$	$O \times O \times C$	$N_{out}$
Nr of param.	$(F \times F \times C + 1) \cdot K$	0	$(N_{in} + 1) \times N_{out}$
Remarks	1 bias param. per filter In most cases, $S < F$ and $K = 2C$ .	Pooling operation done channel-wise. In most cases, $S = F$	Input is flattened 1 bias param. per neuron The number of FC neurons is free of structural constraints

The receptive field at layer  $k$  is the area denoted  $R_k \times R_k$  of the input that each pixel of the  $k$ -th activation map can 'see'. By calling  $F_j$  the filter size of layer  $j$  and  $S_i$  the stride value of layer  $i$  and with the convention  $S_0 = 1$ , the receptive field at layer  $k$  can be computed with the formula:

$$R_k = 1 + \sum_{j=1}^k (F_j - 1) \prod_{i=0}^{j-1} S_i$$

In the example below, we have  $F_1 = F_2 = 3$  and  $S_1 = S_2 = 1$ , which gives  $R_2 = 1 + 2 \cdot 1 + 2 \cdot 1 = 5$



## 1 Universal Approximation Theorem (UAT)

## 2 Training CNN

- Hyperparameters
- Activation functions  $h^k = f(W^k * x + b^k)$
- Loss functions
- Design of algorithms (backpropagation)
- Weight Initialization In Deep Neural Networks
- Regularization to CNN
- Optimizer selection

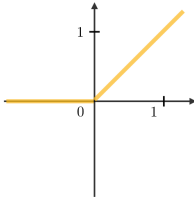
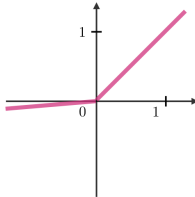
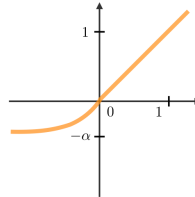
- Non-linear activation layers are employed after **all layers with weights** (so-called learnable layers), such as FC layers and convolutional layers in CNN architecture.
- This means that the mapping of input to output will be **non-linear**; moreover, these layers give the CNN the ability to learn **extra-complicated things**.
- The activation function must also have the ability to **differentiate**, which is an extremely significant feature, as it allows **error back-propagation** to be used to train the network.
- Most commonly used activation functions in CNN and other deep NN:

$$\textbf{Sigmoid: } f(x)_{\text{sigm}} = \frac{1}{1 + e^{-x}} \qquad \frac{\partial f}{\partial x} = f(x) \cdot (1 - f(x))$$

$$\textbf{Tanh: } f(x)_{\text{tanh}} = \frac{e^x - e^{-x}}{e^x + e^{-x}} \qquad \frac{\partial f}{\partial x} = 1 - f^2(x)$$

$$\textbf{ReLU: } f(x)_{\text{ReLU}} = \max(0, x) \qquad \frac{\partial f}{\partial x} = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$$

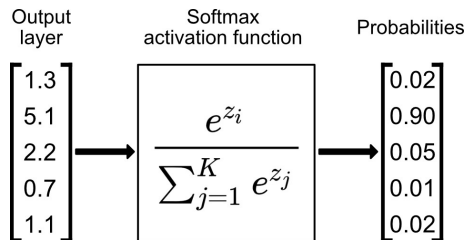
A **dying ReLU** refers to the situations when the neuron always outputs 0 on any input value. In this state, it is difficult to recover because the gradient of 0 is 0.

ReLU	Leaky ReLU	ELU
$g(z) = \max(0, z)$	$g(z) = \max(\varepsilon z, z)$ with $\varepsilon \ll 1$	$g(z) = \max(\alpha(e^z - 1), z)$ with $\alpha \ll 1$
	$g'(z) = \begin{cases} 1 & z > 0 \\ \varepsilon & z \leq 0 \end{cases}$	$g'(z) = \begin{cases} 1 & z > 0 \\ \alpha \cdot e^z & z \leq 0 \end{cases}$
		
Non-linearity complexities biologically interpretable	Addresses dying ReLU issue for negative values	Differentiable everywhere

The softmax step can be seen as a generalized logistic function that takes as input a vector of scores  $x \in \mathbb{R}^n$  and outputs a vector of output probability  $p \in \mathbb{R}^n$  through a softmax function at the end of the architecture. It is defined as follows:

$$p = \begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix} \quad \text{where} \quad p_i = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

Example



## 1 Universal Approximation Theorem (UAT)

## 2 Training CNN

- Hyperparameters
- Activation functions  $h^k = f(W^k * x + b^k)$
- Loss functions
- Design of algorithms (backpropagation)
- Weight Initialization In Deep Neural Networks
- Regularization to CNN
- Optimizer selection



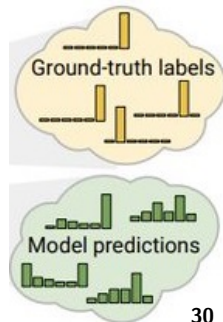
- The final classification is achieved from the output layer, which represents the last layer of the CNN architecture.
- Some loss functions are utilized in the output layer to calculate the predicted error created across the training samples in the CNN model.
- This error reveals the difference between the actual output and the predicted one. Next, it will be optimized through the CNN learning process.
- Two parameters are used by the loss function to calculate the error.

- ❶ The label distribution: actual (designed) output.

$$y = (y_1, \dots, y_n)^T$$

- ❷ The prediction distribution: the CNN estimated output.

$$p = (p_1, \dots, p_n)^T$$



The following concisely explains some of the loss function types:

- **Cross-Entropy or Softmax Loss Function:**

$$H(p, y) = - \sum_{i=1}^n y_i \log(p_i)$$

- **Euclidean Loss Function:**

$$H(p, y) = \frac{1}{2n} \sum_{i=1}^n (p_i - y_i)^2$$

- **Hinge Loss Function:**

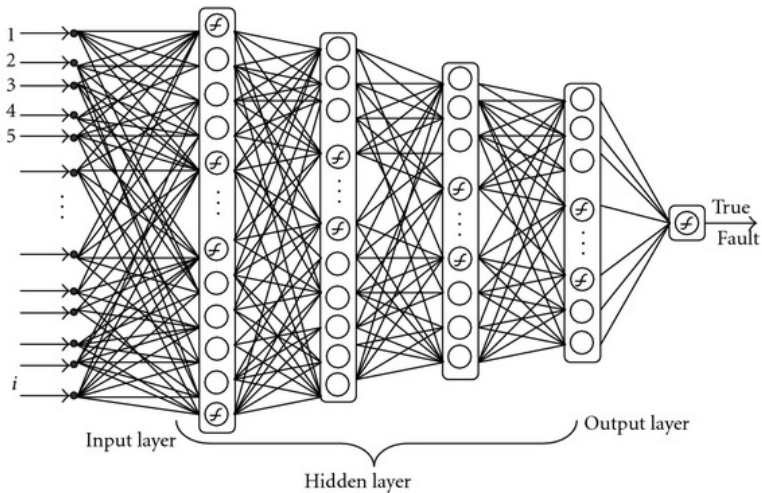
$$H(p, y) = \sum_{i=1}^n \max(0, m - (2y_i - 1)p_i)$$

The margin  $m$  is commonly set to 1

## 1 Universal Approximation Theorem (UAT)

## 2 Training CNN

- Hyperparameters
- Activation functions  $h^k = f(W^k * x + b^k)$
- Loss functions
- Design of algorithms (backpropagation)
- Weight Initialization In Deep Neural Networks
- Regularization to CNN
- Optimizer selection



To convert an MLP from classification to regression, we only need to change the output activation function from logistic to linear.

- The hidden layer non-linearities are smooth functions:

$$h_k^1 = \frac{1}{1 + \exp(-(\sum_d w_{dk}^1 x_d + b_{dk}^1))}$$
$$h_k^i = \frac{1}{1 + \exp(-(\sum_l w_{lk}^i h_l^{(i-1)} + b_{lk}^i))} \text{ for } i = 2, \dots, L$$

To convert an MLP from classification to regression, we only need to change the output activation function from logistic to linear.

- The hidden layer non-linearities are smooth functions:

$$h_k^1 = \frac{1}{1 + \exp(-(\sum_d w_{dk}^1 x_d + b_{dk}^1))}$$
$$h_k^i = \frac{1}{1 + \exp(-(\sum_l w_{lk}^i h_l^{(i-1)} + b_{lk}^i))} \text{ for } i = 2, \dots, L$$

- The output layer activation function is a linear function:

$$\hat{y} = \sum_l w_l^o h_l^L + b^o$$

Let  $\theta$  be the complete collection of parameters defining a neural network model. Our goal is to find the value of  $\theta$  that minimizes the MSE on the training data set

$$\mathcal{D} = \{y_i, \mathbf{x}_i\}_{i=1:N}$$

$$\mathcal{L}_{MSE}(\mathcal{D}|\theta) = \frac{1}{N} \sum_{n=1}^N (y_n - \hat{y}_n)^2$$

We need the gradient with respect to each of the parameters. Let's begin with  $w_l^o$ :

$$\frac{\partial \mathcal{L}_{MSE}(\mathcal{D}|\theta)}{\partial w_l^o} = \frac{\partial}{\partial w_l^o} \frac{1}{N} \sum_{n=1}^N (y_n - \hat{y}_n)^2 = 0 \quad (1)$$

$$= \frac{2}{N} \sum_{n=1}^N (y_n - \hat{y}_n) \frac{\partial \hat{y}_n}{\partial w_l^o} \quad (2)$$

$$= \frac{2}{N} \sum_{n=1}^N (y_n - \hat{y}_n) h_l^L \quad (3)$$



It's also useful to define the derivatives wrt the hidden units for a single data case:

$$\varepsilon_k^L = \frac{\partial \mathcal{L}_{MSE}(y, \mathbf{x} | \theta)}{\partial h_k^L} = \frac{\partial}{\partial h_k^L} (y - \hat{y})^2 \quad (4)$$

$$= 2(y - \hat{y}) \frac{\partial \hat{y}}{\partial h_k^L} \quad (5)$$

$$= 2(y - \hat{y}) w_k^o \quad (6)$$

It's also useful to define the derivatives wrt the hidden units for a single data case:

$$\varepsilon_k^L = \frac{\partial \mathcal{L}_{MSE}(y, \mathbf{x}|\theta)}{\partial h_k^L} = \frac{\partial}{\partial h_k^L} (y - \hat{y})^2 \quad (4)$$

$$= 2(y - \hat{y}) \frac{\partial \hat{y}}{\partial h_k^L} \quad (5)$$

$$= 2(y - \hat{y}) w_k^o \quad (6)$$

In general, we can define:  $\varepsilon_k^j = \frac{\partial \mathcal{L}_{MSE}(y, \mathbf{x}|\theta)}{\partial h_k^j}$

Suppose we're trying to compute the derivative with respect to the weight  $w_{kl}^j$  for some layer  $j$  and assume we have  $\varepsilon_l^j$  computed for all hidden units  $l$  in layer  $j$ .

$$\frac{\partial \mathcal{L}_{MSE}(y, \mathbf{x}|\theta)}{\partial w_{kl}^j} = \frac{\partial \mathcal{L}_{MSE}(y, \mathbf{x}|\theta)}{\partial h_l^j} \frac{\partial h_l^j}{\partial w_{kl}^j} \quad (7)$$

$$= \varepsilon_l^j h_l^j (1 - h_l^j) h_k^{j-1} \quad (8)$$

Suppose we're trying to compute the derivative with respect to the weight  $w_{kl}^j$  for some layer  $j$  and assume we have  $\varepsilon_l^j$  computed for all hidden units  $l$  in layer  $j$ .

$$\frac{\partial \mathcal{L}_{MSE}(y, \mathbf{x}|\theta)}{\partial w_{kl}^j} = \frac{\partial \mathcal{L}_{MSE}(y, \mathbf{x}|\theta)}{\partial h_l^j} \frac{\partial h_l^j}{\partial w_{kl}^j} \quad (7)$$

$$= \varepsilon_l^j h_l^j (1 - h_l^j) h_k^{j-1} \quad (8)$$

The total derivative is then given by:

$$\frac{\partial \mathcal{L}_{MSE}(\mathcal{D}|\theta)}{\partial w_{kl}^j} = \frac{1}{N} \sum_{n=1}^N \frac{\partial \mathcal{L}_{MSE}(y_n, \mathbf{x}_n|\theta)}{\partial w_{kl}^j}$$

Suppose we're trying to compute the error with respect to hidden unit  $k$  in layer  $j - 1$  and assume we have  $\varepsilon_l^j$  computed for all hidden units  $l$  in layer  $j$ .

$$\frac{\partial \mathcal{L}_{MSE}(y, \mathbf{x} | \theta)}{\partial h_k^{j-1}} = \sum_l \frac{\partial \mathcal{L}_{MSE}(y, \mathbf{x} | \theta)}{\partial h_l^j} \frac{\partial h_l^j}{\partial h_k^{j-1}} \quad (9)$$

$$= \sum_l \varepsilon_l^j h_l^j (1 - h_l^j) w_{kl}^{j-1} \quad (10)$$

- The Backpropagation algorithm works by making a forward pass through the network for each data case and storing all the hidden unit values.

- The Backpropagation algorithm works by making a forward pass through the network for each data case and storing all the hidden unit values.
- The algorithm then computes the error at the output and makes a backward pass through the network computing the derivatives with respect to the parameters as well as the contribution of each hidden unit to the error. These are the  $\epsilon_k^j$  values.

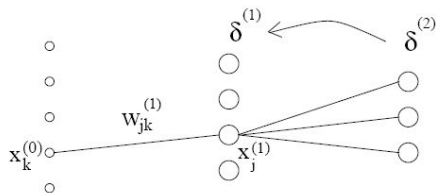
- The Backpropagation algorithm works by making a forward pass through the network for each data case and storing all the hidden unit values.
- The algorithm then computes the error at the output and makes a backward pass through the network computing the derivatives with respect to the parameters as well as the contribution of each hidden unit to the error. These are the  $\varepsilon_k^j$  values.
- The complete computation is just an application of the chain rule with caching of intermediate terms in the neural network graph structure.



Until satisfied, Do

- For each training example, Do

❶ Input the training example to the network and compute the network outputs



- ❷ For each output unit  $k$

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

- ❸ For each hidden unit  $h$

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{h,k} \delta_k$$

- ❹ Update each network weight  $w_{i,j}$

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

where

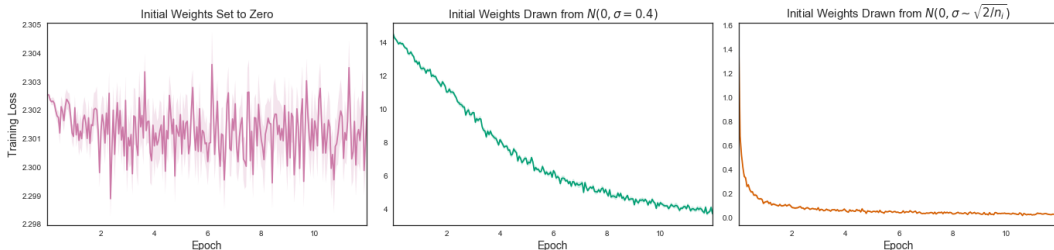
$$\Delta w_{i,j} = \eta \delta_j x_{i,j}$$

## 1 Universal Approximation Theorem (UAT)

## 2 Training CNN

- Hyperparameters
- Activation functions  $h^k = f(W^k * x + b^k)$
- Loss functions
- Design of algorithms (backpropagation)
- Weight Initialization In Deep Neural Networks
- Regularization to CNN
- Optimizer selection

- The optimization algorithm requires a starting point in the space of possible weight values from which to begin the optimization process.
- Weight initialization is a procedure to set the weights of a neural network to small random values that define the starting point for the optimization (learning or training) of the neural network model.
- Each time, a neural network is initialized with a different set of weights, resulting in a different starting point for the optimization process, and potentially resulting in a different final set of weights with different performance characteristics.



## Page 301, Deep Learning, 2016.

*... training deep models is a sufficiently difficult task that most algorithms are strongly affected by the choice of initialization. The initial point can determine whether the algorithm converges at all, with some initial points being so unstable that the algorithm encounters numerical difficulties and fails altogether.*

- We cannot initialize all weights to the value 0.0 because **Initializing all the weights with zeros leads the neurons to learn the same features during training.**
- If we forward propagate an input  $(x_1, x_2)$  in a network with 2 hidden units, the output of both hidden units will be  $\text{relu}(\alpha x_1 + \alpha x_2)$ . Thus, both hidden units will have identical influence on the cost, which will lead to identical gradients. Thus, both neurons will evolve symmetrically throughout training, effectively preventing different neurons from learning different things.
- Historically, weight initialization follows simple heuristics, such as:
  - Small random values in the range  $[-0.3, 0.3]$
  - Small random values in the range  $[0, 1]$
  - Small random values in the range  $[-1, 1]$

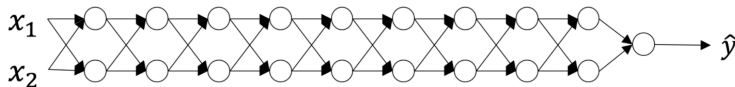
- We almost always initialize all the weights in the model to values drawn randomly from a Gaussian or uniform distribution.
- The choice of Gaussian or uniform distribution does not seem to matter very much, but has not been exhaustively studied.
- The scale of the initial distribution, however, does have a large effect on both the outcome of the optimization procedure and on the ability of the network to generalize.

see:

<https://www.deeplearning.ai/ai-notes/initialization/index.html>

- Despite breaking the symmetry, initializing the weights with values (i) too small or (ii) too large leads respectively to (i) slow learning or (ii) divergence.
- Choosing proper values for initialization is necessary for efficient training.

Consider a 9 layer NN:



Then the output activation is:

$$\hat{y} = a^{[L]} = W^{[L]} W^{[L-1]} W^{[L-2]} \dots W^{[3]} W^{[2]} W^{[1]} x$$

where  $L = 10$  and  $W^{[1]}, W^{[2]}, \dots, W^{[L-1]}$  are all matrices of size  $(2,2)$ . With this in mind, and for illustrative purposes, if we assume  $W^{[1]} = W^{[2]} = \dots = W^{[L-1]} = W$  the output prediction is  $\hat{y} = W^{[L]} W^{L-1} x$

- Consider the case where every weight is initialized slightly larger than the identity matrix.

$$W^{[1]} = W^{[2]} = \dots = W^{[L-1]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$

- This simplifies to  $\hat{y} = W^{[L]} 1.5^{L-1} x$ , and the values of  $a^{[l]}$  increase exponentially with  $l$ .
- When these activations are used in backward propagation, this leads to **the exploding gradient** problem.
- That is, the gradients of the cost with the respect to the parameters are too big.
- This leads the cost to **oscillate** around its minimum value.



- Consider the case where every weight is initialized **slightly smaller** than the identity matrix.

$$W^{[1]} = W^{[2]} = \dots = W^{[L-1]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$$

- This simplifies to  $\hat{y} = W^{[L]} 0.5^{L-1} x$ , and the values of  $a^{[l]}$  decrease exponentially with  $l$ .
- When these activations are used in backward propagation, this leads to the **vanishing gradient problem**.
- That is, the gradients of the cost with the respect to the parameters are too small, leading to convergence of the cost before it has reached the minimum value.

- To prevent the gradients of the network's activations from vanishing or exploding, we will stick to the following rules of thumb:
  - The **mean** of the activations should be **zero**.
  - The **variance** of the activations should stay the same across every layer.
- Nevertheless, more tailored approaches have been developed over the last decade that have become the defacto standard given they may result in a slightly more effective optimization (model training) process.
- These modern weight initialization techniques are divided based on the type of activation function used in the nodes that are being initialized, such as “Sigmoid and Tanh” and “ReLU.”

- The recommended initialization is Xavier initialization (or one of its derived methods), for every layer  $l$ :

$$W^{[l]} \sim \mathcal{N}(\mu = 0, \sigma^2 = \frac{1}{n^{[l-1]}})$$
$$b^{[l]} = 0$$

- In other words, all the weights of layer  $l$  are picked randomly from a normal distribution with mean  $\mu = 0$  and variance  $\sigma^2 = \frac{1}{n^{[l-1]}}$  where  $n^{[l-1]}$  is the number of neuron in layer  $l - 1$ . Biases are initialized with zeros.
- **Normalized Xavier Weight Initialization** In practice, Machine Learning Engineers using Xavier initialization would either initialize the weights as  $\mathcal{N}(0, \frac{1}{n^{[l-1]}})$  or as  $\mathcal{N}(0, \frac{2}{n^{[l-1]} + n^{[l]}})$ . The variance term of the latter distribution is the harmonic mean of  $\frac{1}{n^{[l-1]}}$  and  $\frac{1}{n^{[l]}}$
- Xavier initialization works with **tanh activations**

- In **He Uniform weight initialization**, the weights are assigned from values of a uniform distribution as follows:

$$w_i \sim U \left[ -\sqrt{\frac{6}{n[l-1]}}, \sqrt{\frac{6}{n[l]}} \right]$$

- In **He Normal Initialization**, the weights are assigned from values of a normal distribution as follows:

$$w_i \sim N[0, \sigma]$$

Here,  $\sigma$  is given by:

$$\sigma = \sqrt{\frac{2}{n^l - 1}}$$

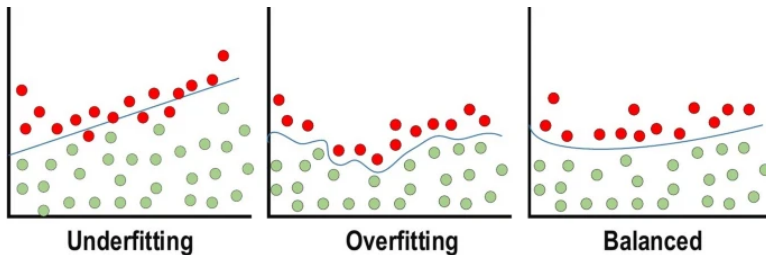
He Normal Initialization is suitable for layers where ReLU activation function is used.

## 1 Universal Approximation Theorem (UAT)

## 2 Training CNN

- Hyperparameters
- Activation functions  $h^k = f(W^k * x + b^k)$
- Loss functions
- Design of algorithms (backpropagation)
- Weight Initialization In Deep Neural Networks
- Regularization to CNN
- Optimizer selection

- For CNN models, over-fitting represents the central issue associated with obtaining well-behaved generalization.
- The model is entitled over-fitted in cases where the model executes especially well on training data and does not succeed on test data (unseen data) which is more explained in the latter section.
- An under-fitted model is the opposite; this case occurs when the model does not learn a sufficient amount from the training data.
- The model is referred to as “just-fitted” if it executes well on both training and testing data.



- ① **Dropout:** This is a widely utilized technique for generalization. During each training epoch, neurons are randomly dropped.
  - the feature selection power is distributed equally across the whole group of neurons, as well as forcing the model to learn different independent features.
  - **training process:** the dropped neuron will not be a part of back-propagation or forward-propagation.
  - **testing process:** the full-scale network is utilized to perform prediction
- ② **Drop-Weights:** This method is highly similar to dropout. In each training epoch, the connections between neurons (weights) are dropped rather than dropping the neurons; this represents the only difference between drop-weights and dropout.
- ③ **Data Augmentation:** utilizes to train the model on a sizeable (artificially expanded) amount of data. This is the easiest way to avoid over-fitting.
- ④ **Batch Normalization:** Subtracting the mean and dividing by the standard deviation will normalize the output at each layer. While it is possible to consider this as a pre-processing task at each layer in the network, it is also possible to differentiate and to integrate it with other networks.

BN can be employed to reduce the “internal covariance shift” of the activation layers:

- **Internal covariance shift:** the variation in the activation distribution in each layer
- This shift becomes very high due to the continuous weight updating through training, which may occur if the samples of the training data are gathered from numerous dissimilar sources (for example, day and night images).
- Thus, the model will consume extra time for convergence, and in turn, the time required for training will also increase.
- To resolve this issue, a **BN layer** is applied in the CNN architecture.

The advantages of utilizing batch normalization are as follows:

- It prevents the problem of vanishing gradient from arising.
- It can effectively control the poor weight initialization.
- It significantly reduces the time required for network convergence
- It struggles to decrease training dependency across hyper-parameters.
- Chances of overfitting are reduced, since it has a minor influence on regularization.

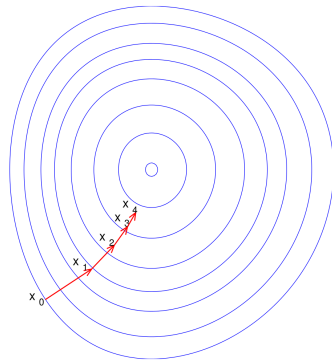
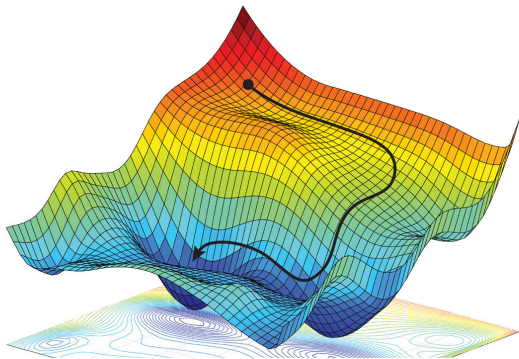


## 1 Universal Approximation Theorem (UAT)

## 2 Training CNN

- Hyperparameters
- Activation functions  $h^k = f(W^k * x + b^k)$
- Loss functions
- Design of algorithms (backpropagation)
- Weight Initialization In Deep Neural Networks
- Regularization to CNN
- Optimizer selection

- Two major issues are included in the learning process:
  - the first issue is the learning algorithm selection (optimizer),
  - the second issue is the use of many enhancements (such as AdaDelta, Adagrad, and momentum) along with the learning algorithm to enhance the output.
- **Loss functions**, which are determined on numerous learnable parameters (e.g. biases, weights, etc.) or minimizing the error (variation between actual and predicted output), are the core purpose of all supervised learning algorithms.
- The techniques of gradient-based learning for a CNN network appear as the usual selection.
- The network parameters should always update though all training epochs, while the network should also look for the locally optimized answer in all training epochs in order to minimize the error.



We start with a guess  $\mathbf{x}_0$  for a local minimum of  $F$ , and considers the sequence  $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \dots$  such that

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla F(\mathbf{x}_n), \quad n \geq 0.$$

- To minimize the training error, this algorithm repetitively updates the network parameters through every training epoch.
  - it needs to compute the objective function gradient (slope) by applying a first-order derivative with respect to the network parameters.
  - Next, the parameter is updated in the reverse direction of the gradient to reduce the error.
  - The parameter updating process is performed though network back-propagation, in which the gradient at every neuron is back-propagated to all neurons in the preceding layer.

$$w_{ij}^t = w_{ij}^{t-1} - \Delta w_{ij}^t, \quad \Delta w_{ij}^t = \eta * \frac{\partial E}{\partial w_{ij}}$$

- The learning rate  $\eta$  is defined as the step size of the parameter updating. The training epoch represents a complete repetition of the parameter update that involves the complete training dataset at one time. Note that it needs to select the learning rate wisely so that it does not influence the learning process imperfectly, although it is a hyper-parameter.

Different alternatives of the gradient-based learning algorithm are available and commonly employed; these include the following:

- **Batch Gradient Descent (BGD):** The standard formula involves calculations over full training set  $X$ , at each gradient descent step.
- **Stochastic Gradient Descent (SGD):** picks a random instance in the training set at every step and computes the gradients based only on that instance.
  - *it is much faster than Batch version*
  - *random nature  $\implies$  it is much less regular than Batch Gradient Descent*
  - *good advantage: when the cost function is irregular, the randomness can help jump out of local minima.*
- **Mini-Batch Gradient Descent:** *at each step, we compute the gradients on small random sets of instances called mini-batches. It will end up walking a bit closer to minimum compared to SGD, but it may be harder for him to escape local minima. The advantage of this method comes from combining the advantages of both BGD and SGD techniques. Thus, it has a steady convergence, more computational efficiency and extra memory effectiveness.*

The following describes several enhancement techniques in gradient-based learning algorithms (usually in SGD), which further powerfully enhance the CNN training process.

- **Momentum:** For neural networks, this technique is employed in the objective function. It enhances both the accuracy and the training speed by summing the computed gradient at the preceding training step, which is weighted via a factor  $\lambda$  (known as the **momentum factor**).
- However, it therefore simply becomes stuck in a local minimum rather than a global minimum. This represents the main disadvantage of gradient-based learning algorithms. Issues of this kind frequently occur if the issue has no convex surface (or solution space).
- Together with the learning algorithm, momentum is used to solve this issue, which can be expressed mathematically as

$$\Delta w_{ij}^t = \left( \eta * \frac{\partial E}{\partial w_{ij}} \right) + (\lambda * \Delta w_{ij}^{t-1})$$

- The **momentum factor** value is maintained within the range 0 to 1; in turn, the step size of the weight updating increases in the direction of the bare minimum to minimize the error.
- As the value of the momentum factor becomes very low, the model loses its ability to avoid the local bare minimum.
- By contrast, as the momentum factor value becomes high, the model develops the ability to converge much more rapidly.
- If a high value of momentum factor is used together with **learning rate**, then the model could miss the global bare minimum by crossing over it.
- However, when the gradient varies its direction continually throughout the training process, then the suitable value of the momentum factor (which is a hyper-parameter) causes a smoothening of the weight updating variations.

- It is another optimization technique or learning algorithm that is widely used.
- Adam represents the latest trends in deep learning optimization.
- This is represented by the Hessian matrix, which employs a second-order derivative.
- Adam is a learning strategy that has been designed specifically for training deep neural networks.
- More memory efficient and less computational power are two advantages of Adam.
- The mechanism of Adam is to calculate **adaptive LR for each parameter** in the model.
- It integrates the pros of both Momentum and RMSprop. It utilizes the squared gradients to scale the learning rate as RMSprop and it is similar to the momentum by using the moving average of the gradient.

$$w_{ij}^t = w_{ij}^{t-1} - \frac{\eta}{\sqrt{\widehat{E[\delta^2]}^t + \epsilon}} * \widehat{E[\delta]}^t$$



The most active solutions that may improve the performance of CNN are:

- Expand the dataset with data augmentation or use transfer learning (explained in latter sections).
- Increase the training time.
- Increase the depth (or width) of the model.
- Add regularization.
- Increase hyperparameters tuning.

The End  
Thank You!