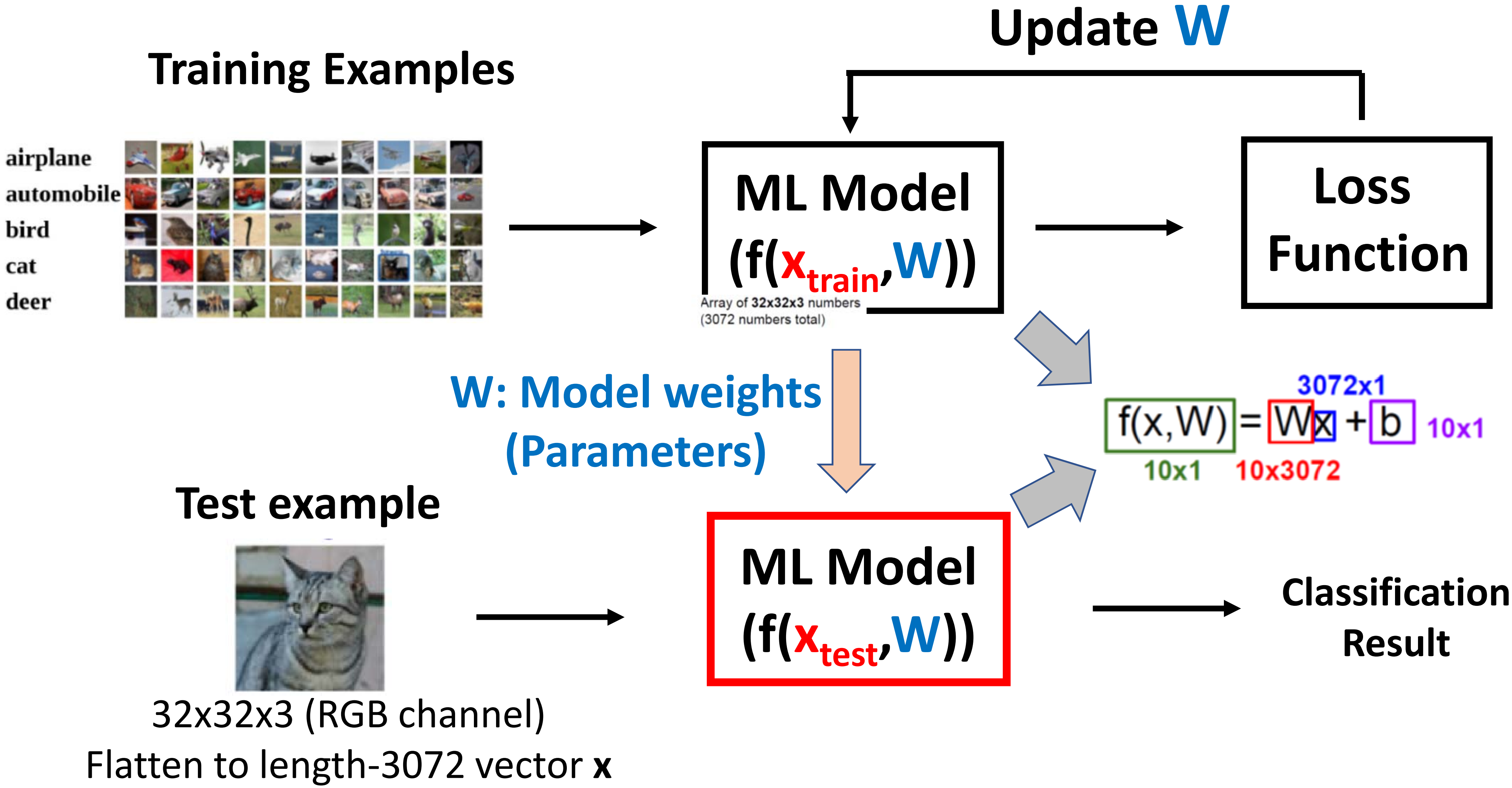
14.332.435/16.332.579
Energy-efficient Machine Learning System

Lecture 5
Neural Network Basics

Bo Yuan

Department of Electrical and Computer Engineering

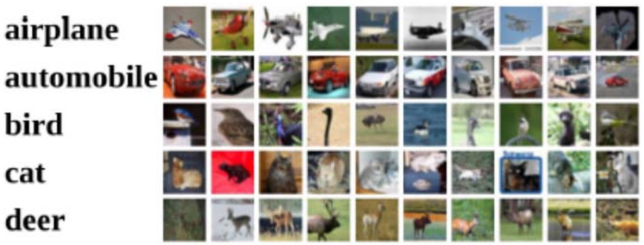
Recall Lecture 2– Linear Classifier



Today's Agenda

Update W

Training Examples



ML Model
(f(x_{train},W))

Loss Function

W: Model weights (Parameters)

$$f(\mathbf{x}, \mathbf{W}) = \left(\mathbf{Act}\left(\mathbf{W_2}\left(\mathbf{Act}(\mathbf{W_1 x} + \mathbf{b_1})\right)\right) + \mathbf{b_2}\right) \ldots$$

Test example

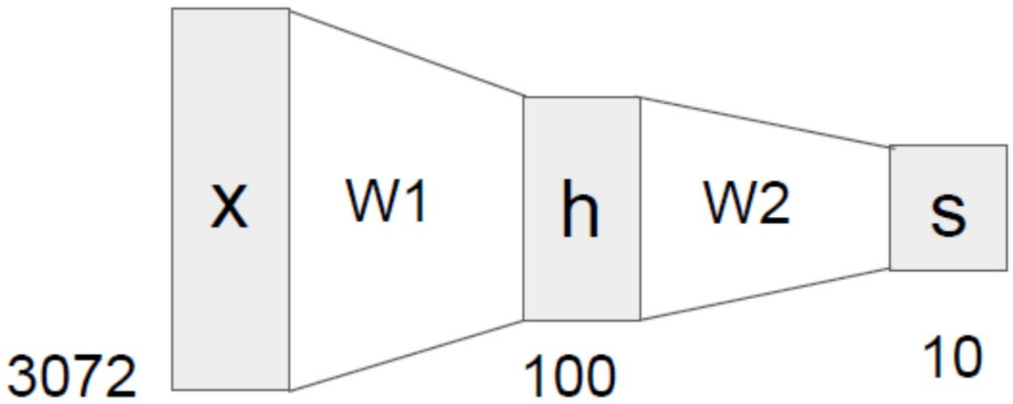
32x32x3 (RGB channel)
Flatten to length-3072 vector **x**

ML Model
(f(x_{test},W))

Classification Result

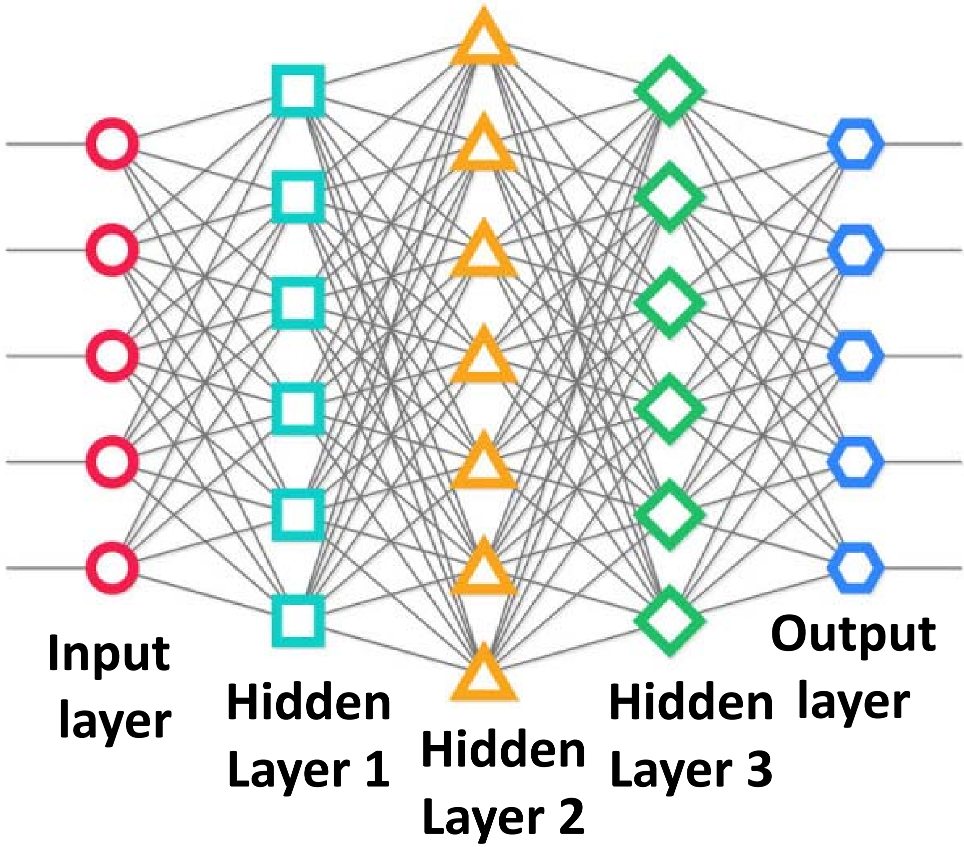
It is Easy to Build a Neural Network

- Multi-layer neural network (NN) can be built on linear classifier
- Score function of linear classifier: $\mathbf{f} = \mathbf{W}\mathbf{x}$
- 2-*layer* NN: $\mathbf{f} = \mathbf{W_2}\,\mathbf{max}(\mathbf{0}, \mathbf{W_1}\mathbf{x})$



A Detailed Look at Neural Network

- Multiple Layers. Each layer has multiple neurons.
- Input layer: the 1st layer
- Output layer: the last layer
- Hidden layer: all the other layers
- Neurons between adjacent layers are connected (typically)
- Neurons within the same layer are **not** connected

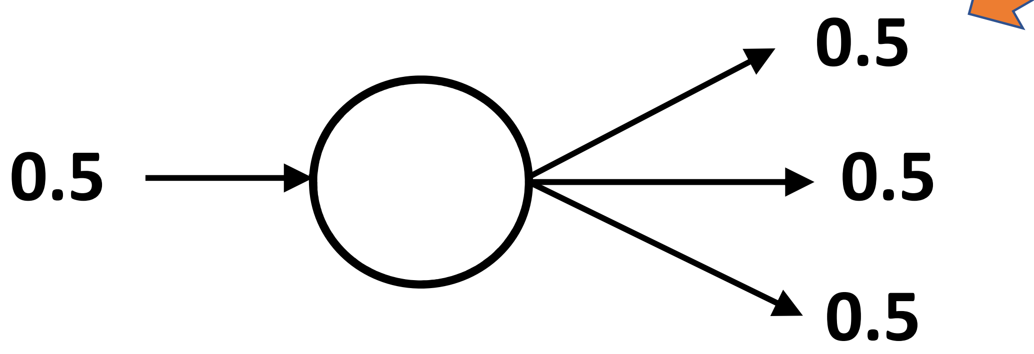


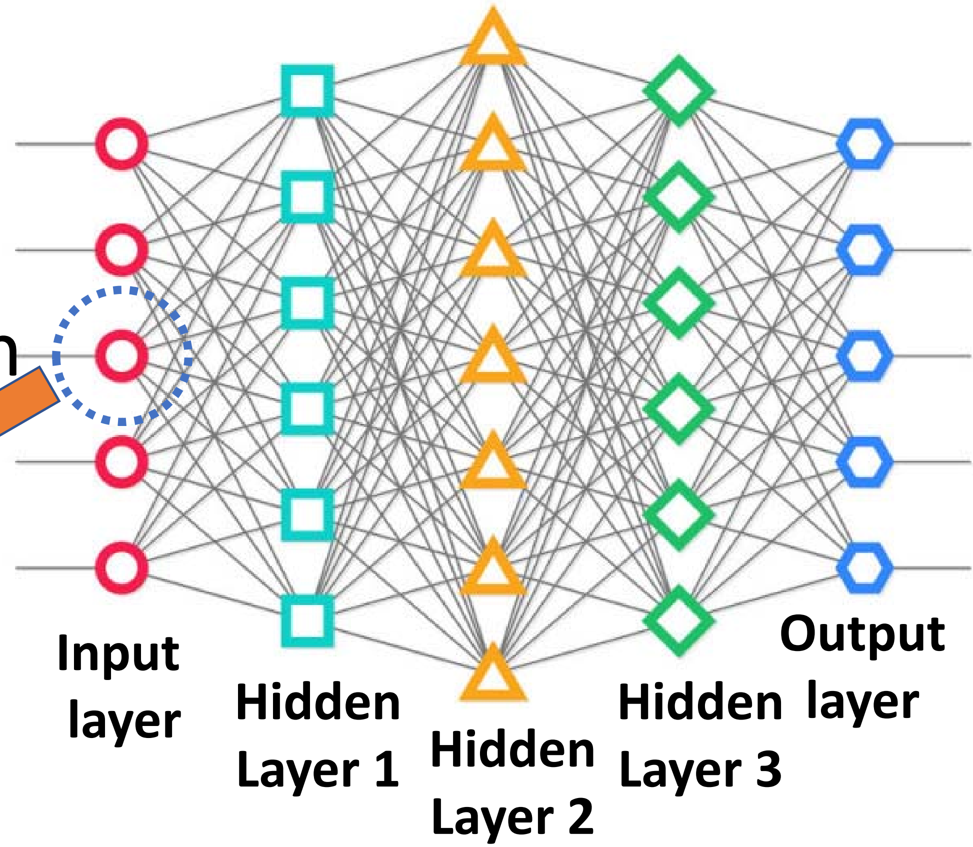
A Detailed Look at Neural Network

- The neurons in input layer is "transparent"

  -- Output is the input

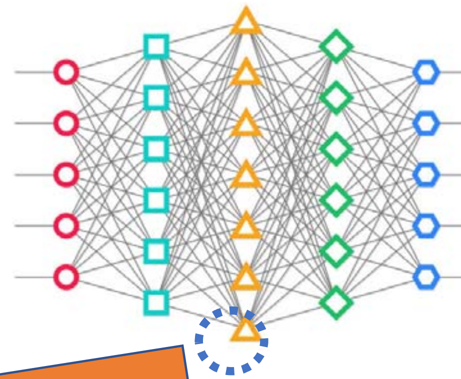
- Use for consistent representation

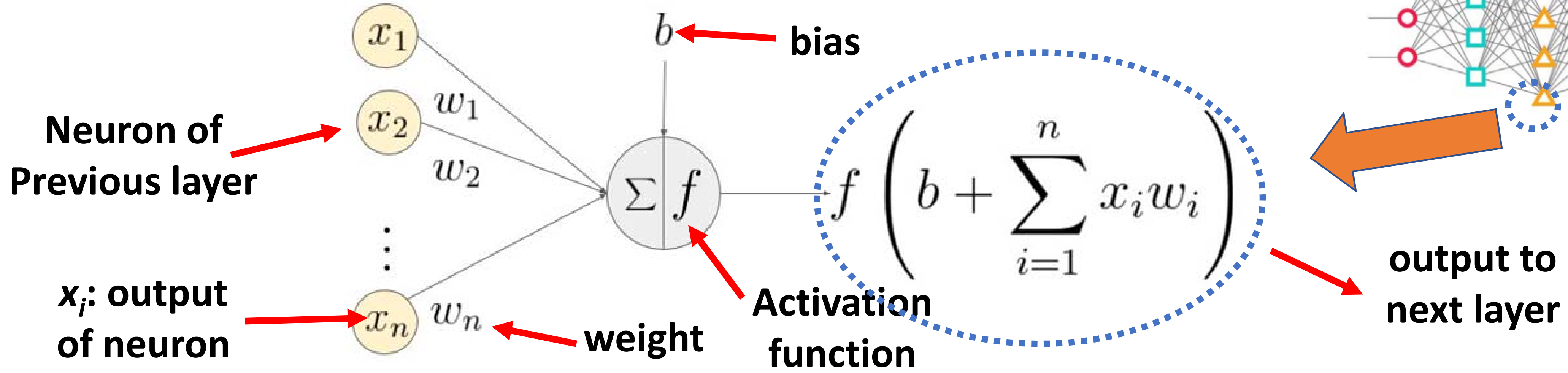




A Detailed Look at Neural Network

- Two parts in the neurons of hidden layer
  -- accumulation of product and activation function
- The connection among neurons has a *weight*
- Weight is the parameter should be **learned**.





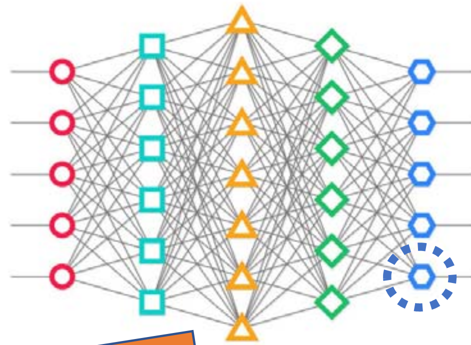
A Detailed Look at Neural Network

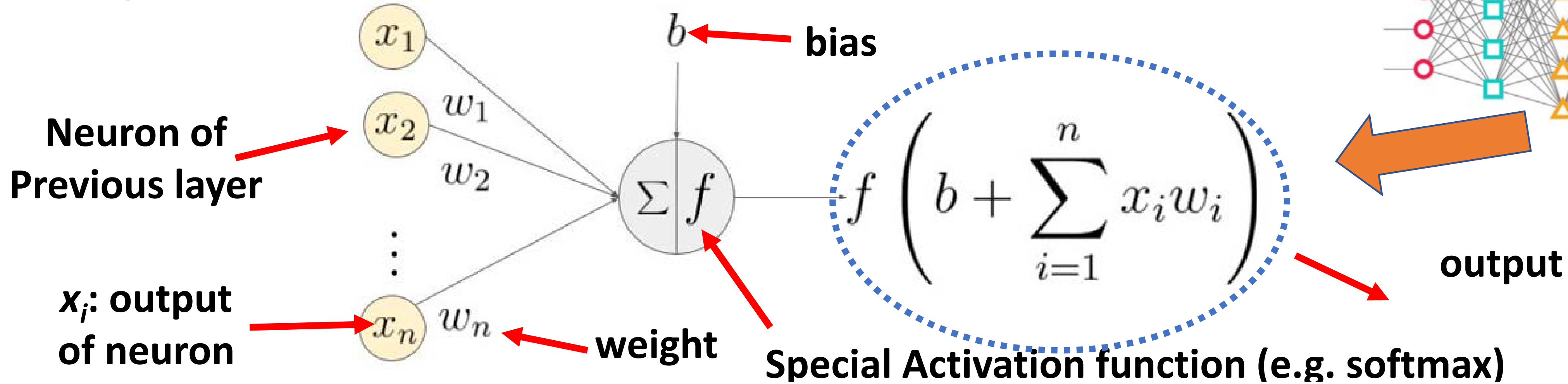
- Neuron of output layer has “special” activation function

  -- E.g. softmax function, identity function

  -- Can also be standard neuron if followed

by other classifier (NN acts as feature extractor)





Another View

- It is also common to take activation as another layer

```python
class AlexNet(nn.Module):

    def __init__(self, num_classes=1000):
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )
```

class **torch.nn.Softmax**(*dim=None*) [source]

Applies the Softmax function to an n-dimensional input Tensor rescaling them so that the elements of the n-dimensional output Tensor lie in the range (0,1) and sum to 1

Softmax is defined as $f_i(x) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$

Example in Tensorflow

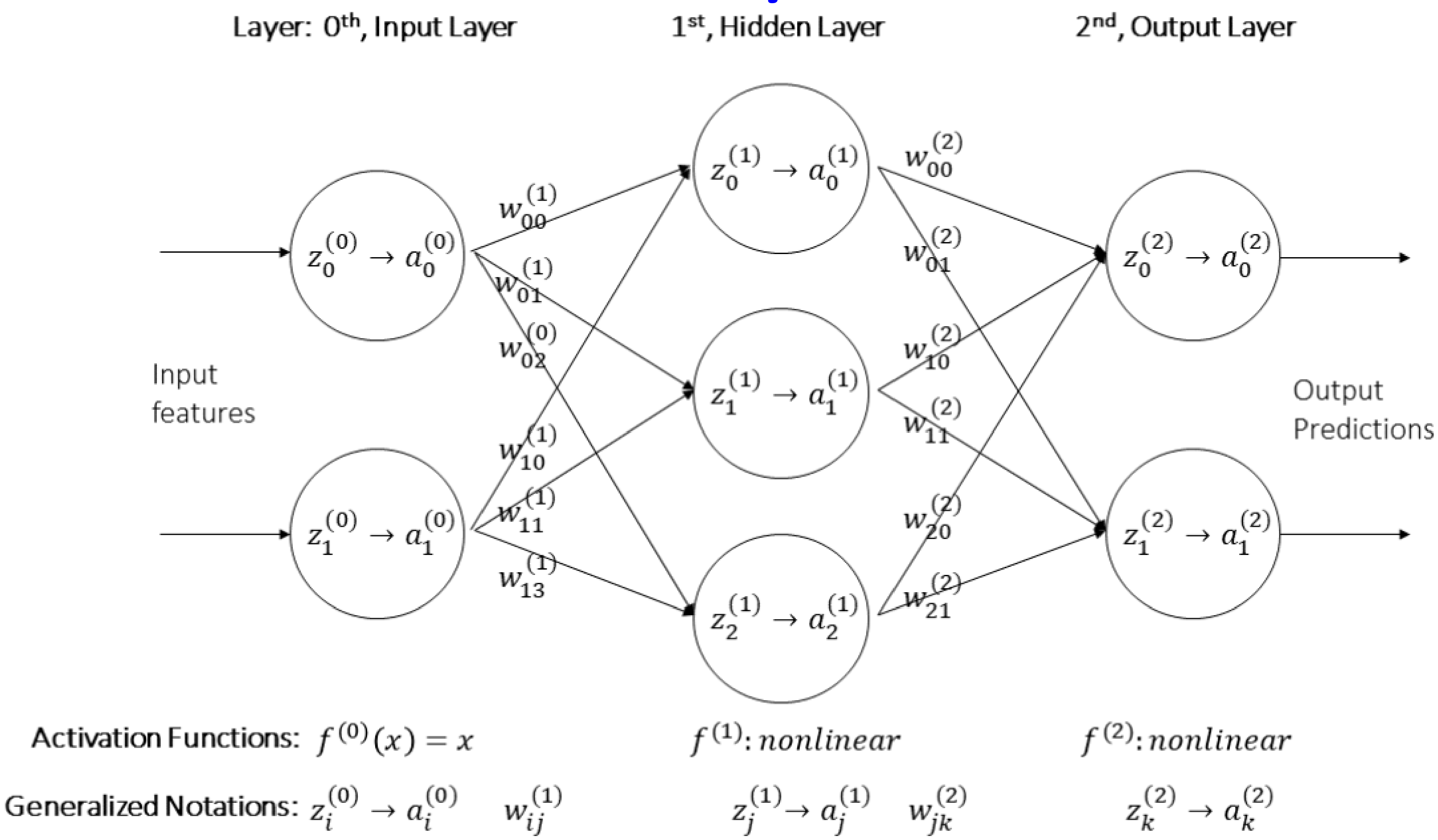
tf.layers.dense

```
tf.layers.dense(
    inputs,
    units,
    activation=None,
    use_bias=True,
    kernel_initializer=None,
    bias_initializer=tf.zeros_initializer(),
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    trainable=True,
    name=None,
    reuse=None
)
```

Arguments:

- `inputs` : Tensor input.
- `units` : Integer or Long, dimensionality of the output space.
- `activation` : Activation function (callable). Set it to None to maintain a linear activation.
- `use_bias` : Boolean, whether the layer uses a bias.

Always Vectorization



$$
\begin{aligned}
a_i^{(0)} &= f^{(0)}(z_i^{(0)}) \\
a_j^{(1)} &= f^{(1)}(z_j^{(1)}) \\
a_k^{(2)} &= f^{(2)}(z_k^{(2)})
\end{aligned}
$$

$$
\begin{aligned}
z_j^{(1)} &= \sum_i w_{(ij)}^{(1)} a_i^{(0)} \\
z_k^{(2)} &= \sum_j w_{(jk)}^{(2)} a_j^{(1)}
\end{aligned}
$$

Always Vectorization

$$z_p^{(l)} = \sum_q w_{(qp)}^{(l)} a_q^{(l-1)}$$

$$\begin{bmatrix} z_0^{(2)} \\ z_1^{(2)} \end{bmatrix} = \begin{bmatrix} w_{00}^{(2)} a_0^{(1)} + w_{10}^{(2)} a_1^{(1)} + w_{20}^{(2)} a_2^{(1)} \\ w_{01}^{(2)} a_0^{(1)} + w_{11}^{(2)} a_1^{(1)} + w_{21}^{(2)} a_2^{(1)} \end{bmatrix}$$

$$\begin{bmatrix} z_0^{(2)} \\ z_1^{(2)} \end{bmatrix} = \begin{bmatrix} w_{00}^{(2)} & w_{01}^{(2)} \\ w_{10}^{(2)} & w_{11}^{(2)} \\ w_{20}^{(2)} & w_{21}^{(2)} \end{bmatrix}^T \begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ a_2^{(1)} \end{bmatrix}$$

$$Z^{(2)} = W^{(2)T} A^{(1)}$$

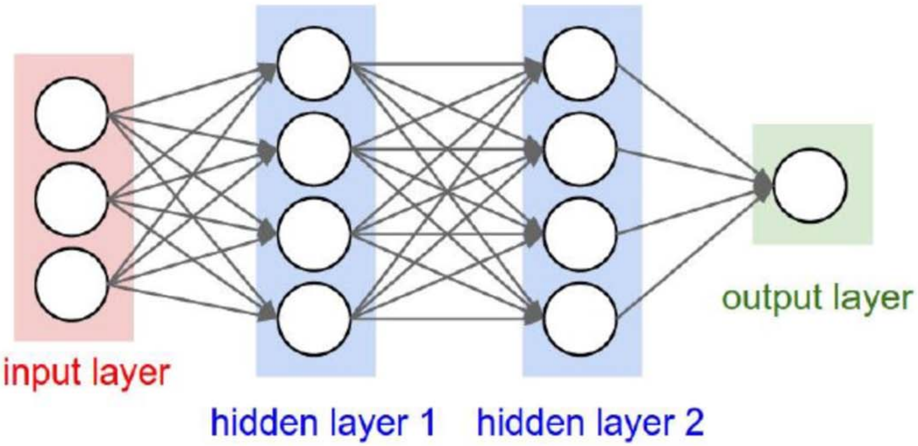
$$\begin{aligned} a_i^{(0)} &= f^{(0)}(z_i^{(0)}) \\ a_j^{(1)} &= f^{(1)}(z_j^{(1)}) \\ a_k^{(2)} &= f^{(2)}(z_k^{(2)}) \end{aligned}$$

$$a_p^{(l)} = f^{(l)}(z_p^{(l)})$$

$$A^{(2)} = f^{(2)}(Z^{(2)})$$

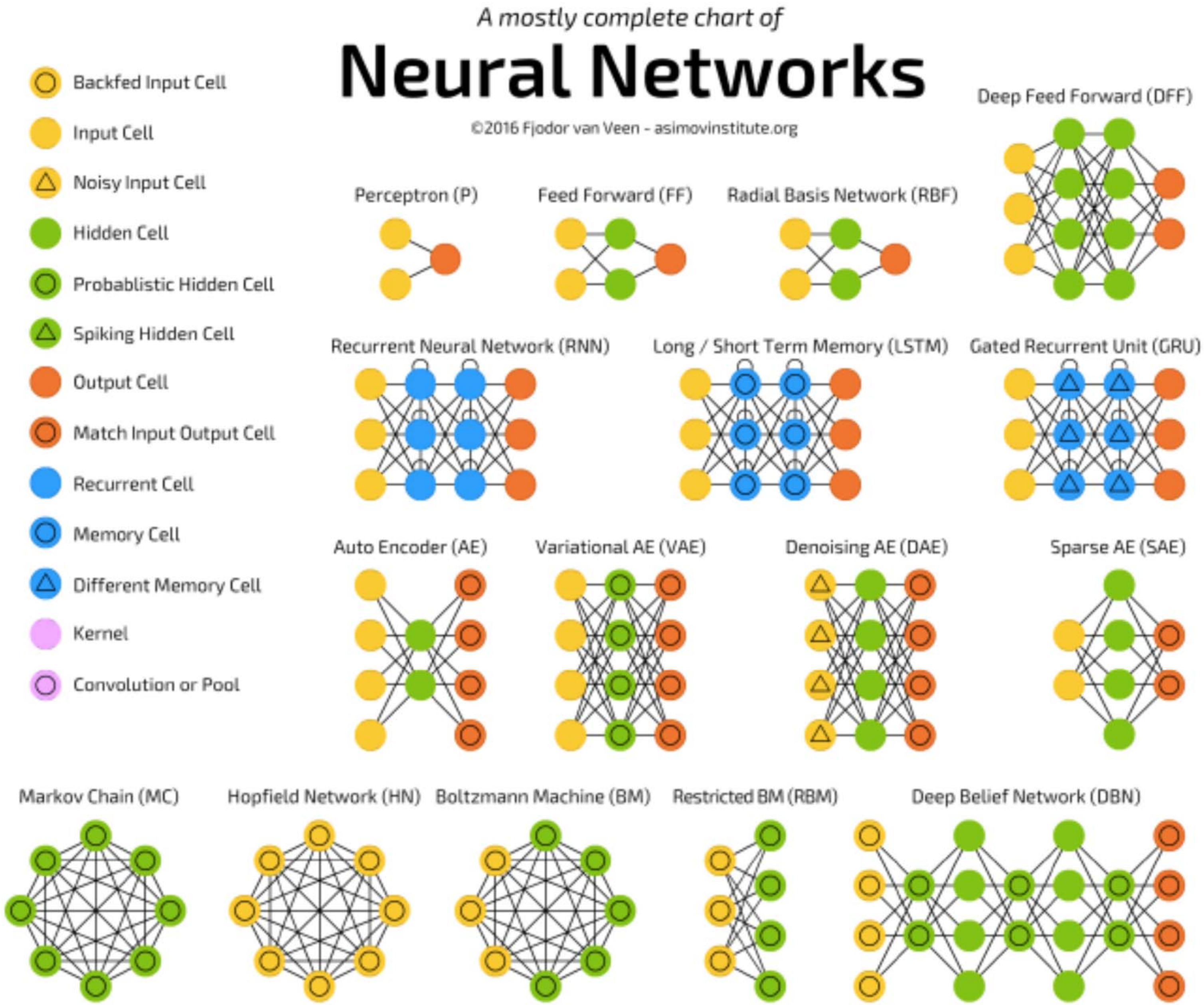
$$\boxed{\begin{aligned} A^{(l)} &= f^{(l)}(Z^{(l)}) \\ Z^{(l)} &= W^{(l)T} A^{(l-1)} \end{aligned}}$$

Vectorization in Python



The transpose operation is not always necessary

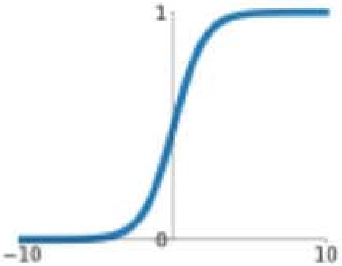
```python
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```


A mostly complete chart of
Neural Networks
©2016 Fjodor van Veen - asimovinstitute.org
Backfed Input Cell
Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Different Memory Cell
Kernel
Convolution or Pool
Perceptron (P)
Feed Forward (FF)
Radial Basis Network (RBF)
Deep Feed Forward (DFF)
Recurrent Neural Network (RNN)
Long / Short Term Memory (LSTM)
Gated Recurrent Unit (GRU)
Auto Encoder (AE)
Variational AE (VAE)
Denoising AE (DAE)
Sparse AE (SAE)
Markov Chain (MC)
Hopfield Network (HN)
Boltzmann Machine (BM)
Restricted BM (RBM)
Deep Belief Network (DBN)

Different Activation Functions

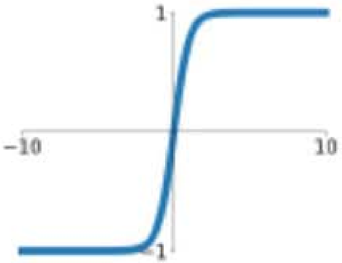
Sigmoid

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



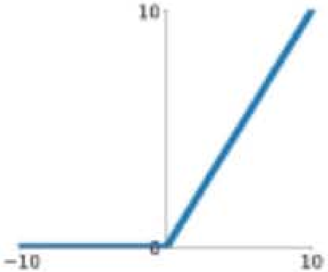
tanh

$\tanh(x)$



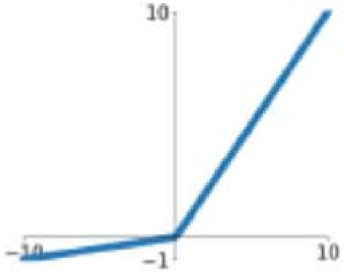
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

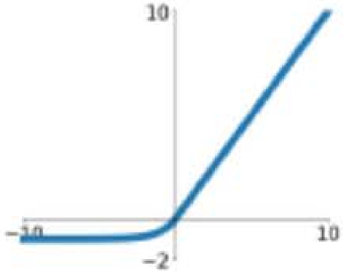


Maxout

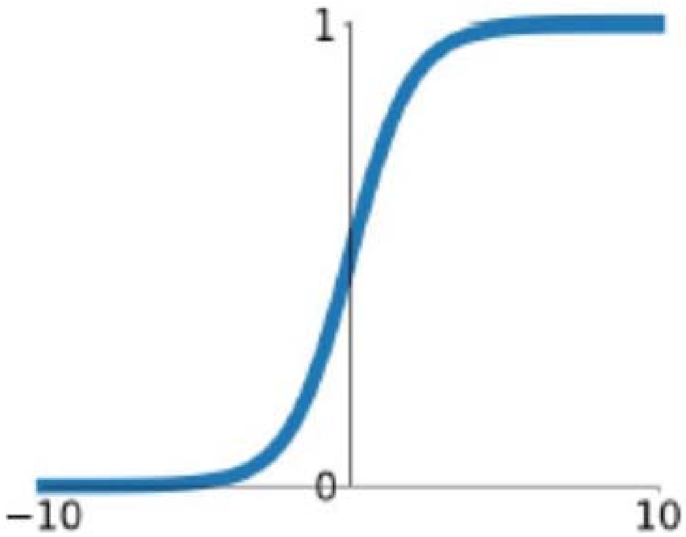
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Sigmoid Activation Function



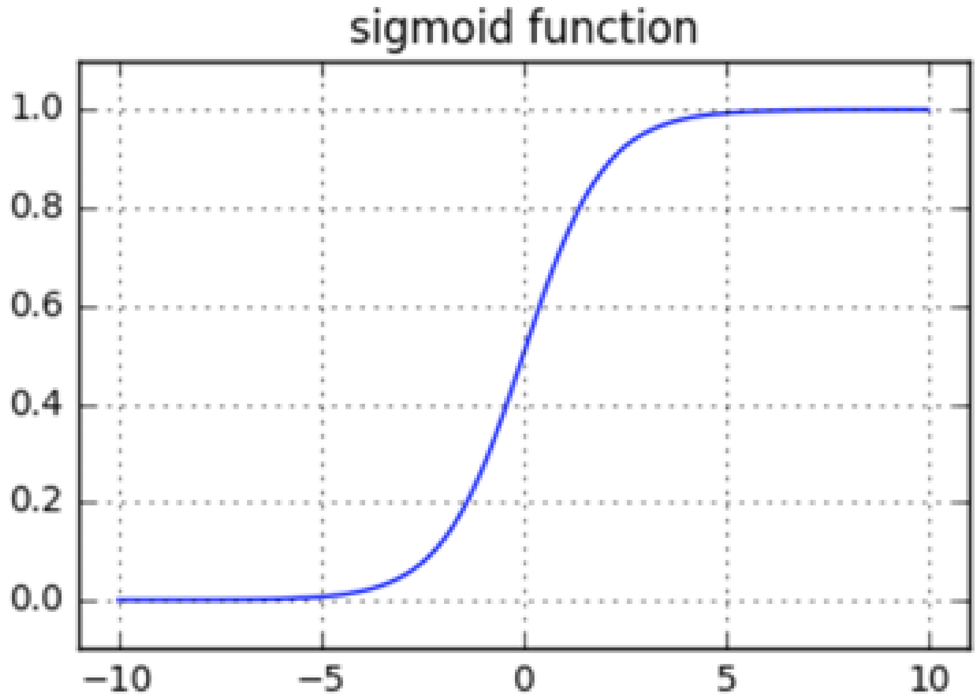
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

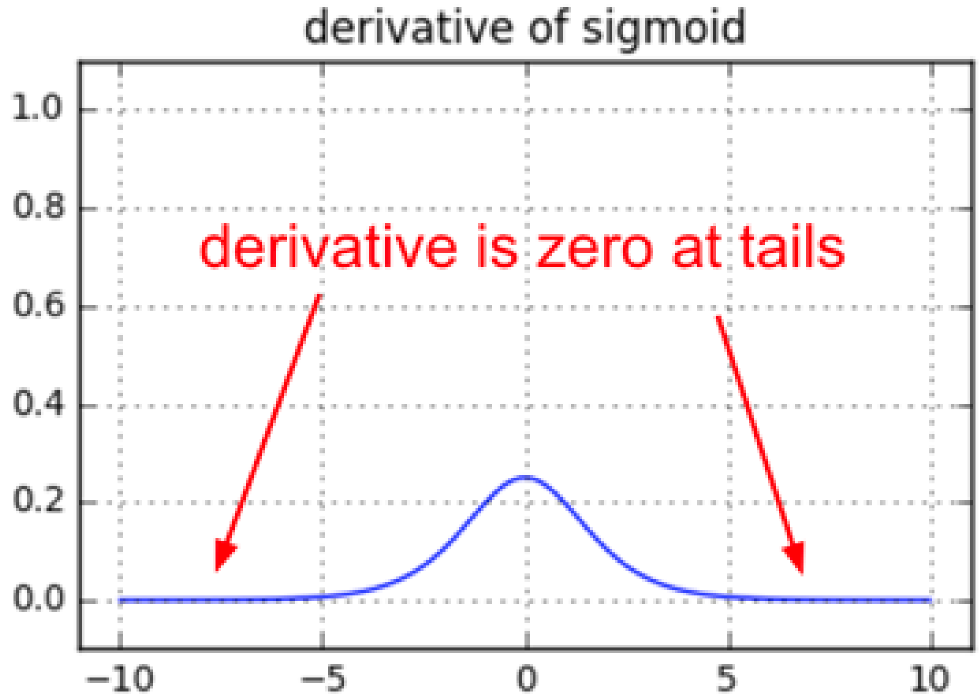
- Squashes numbers to range [0, 1]
- Historically popular
- Still used in RNN

Cons of Sigmoid – Saturation

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

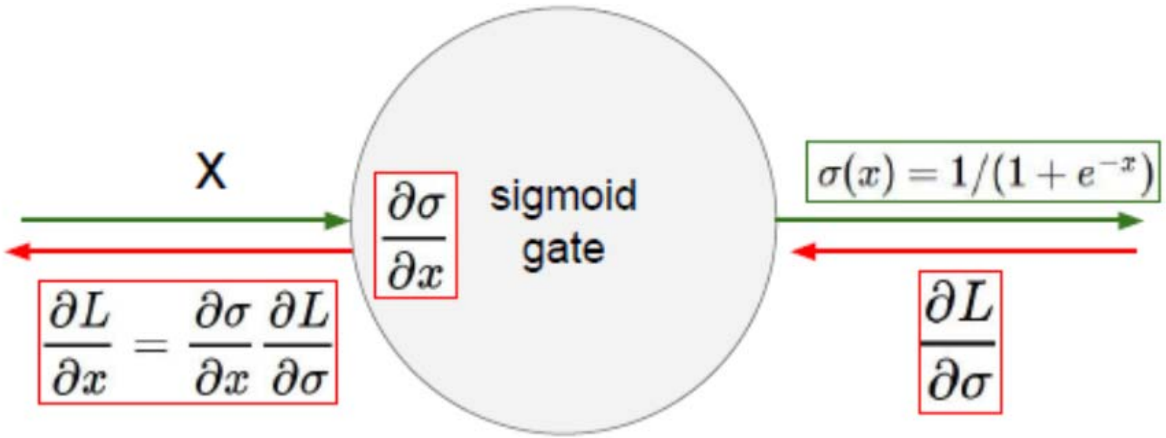
$$\frac{d\sigma(x)}{dx} = (1-\sigma(x))\sigma(x)$$

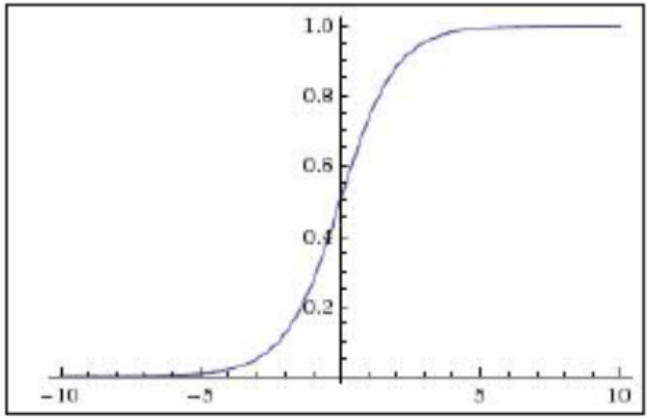




Cons of Sigmoid – Saturation

- Saturated neurons “kill” the gradients





-- What happens when x = -10?

-- What happens when x = 0?

-- What happens when x = 10?

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

Cons of Sigmoid – Non-zero Centered

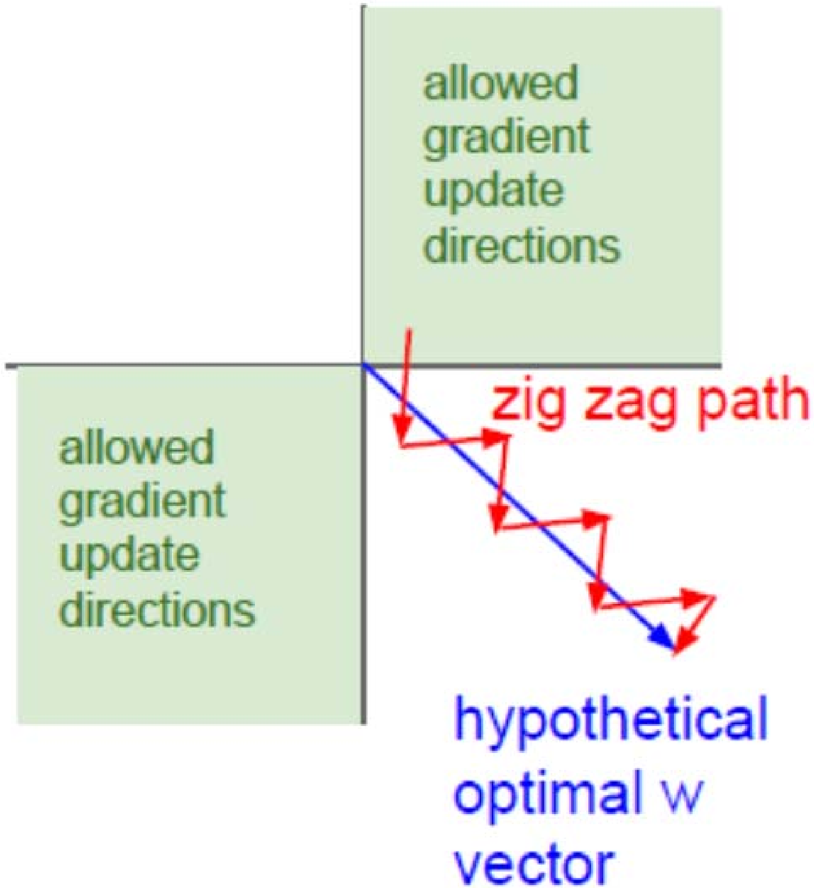
- Consider what happens when the input to a neuron is always positive

$$f\left(\sum_i w_i x_i + b\right)$$

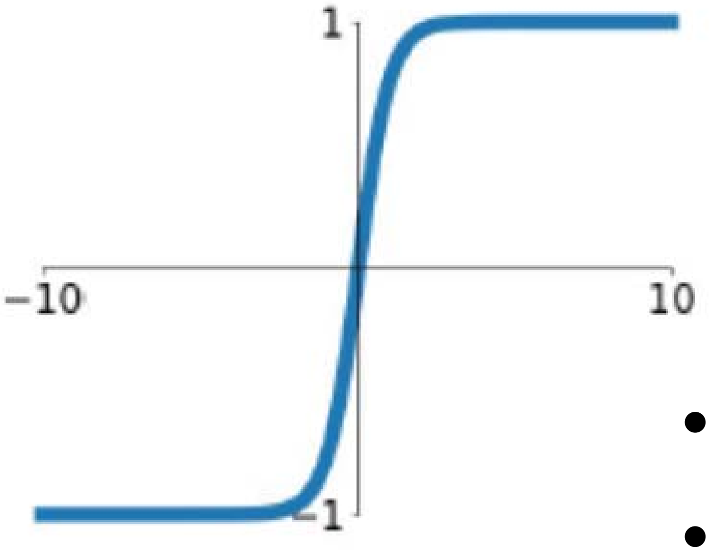
Q: What will be the gradient on **w**?

Cons of Sigmoid – Non-zero Centered

- Slow learning



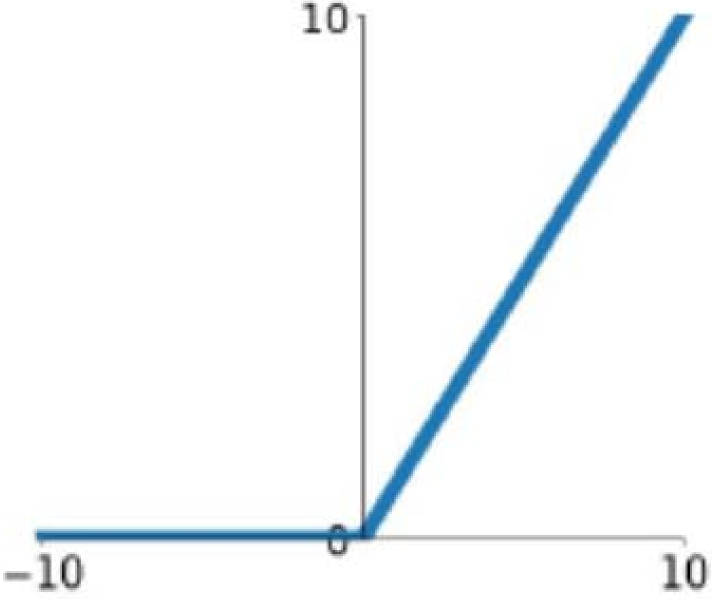
Tanh Activation Function



$$tanh(x) = 2\sigma(2x) - 1$$

- Squashes numbers to range [-1, 1]
- Zero-centered (preferred than sigmoid)
- Still used in RNN

RELU Activation Function



RELU: Rectified Linear Unit

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

- Squashes numbers to range [0, ∞]
- Does not saturate (half region)
- Most popular in CNN and FCN

Pros of RELU

- Not always saturation
  -- Half region
- Low-cost computation
  -- Max function is simple
- Fast convergence speed
  -- Very important

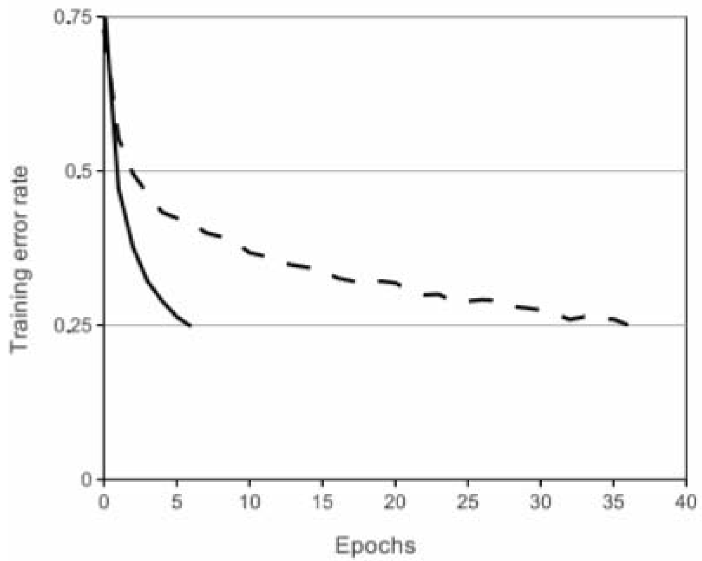
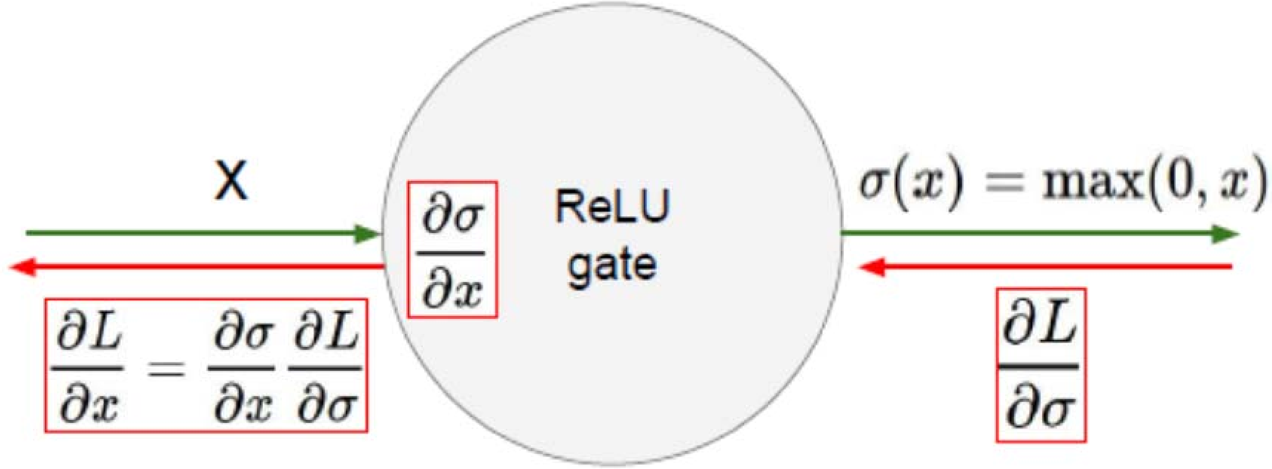


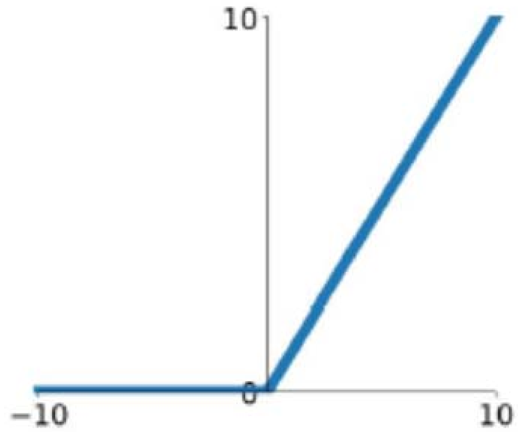
Figure 1: A four-layer convolutional neural network with ReLUs **(solid line)** reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons **(dashed line)**. The learning rates for each network were chosen independently to make training as fast as possible. No regularization of any kind was employed. The magnitude of the effect demonstrated here varies with network architecture, but networks with ReLUs consistently learn several times faster than equivalents with saturating neurons.

Krizhevsky et al., NIPS 2012

Con of RELU – Saturation

- When x<0, saturated neurons “kill” the gradients





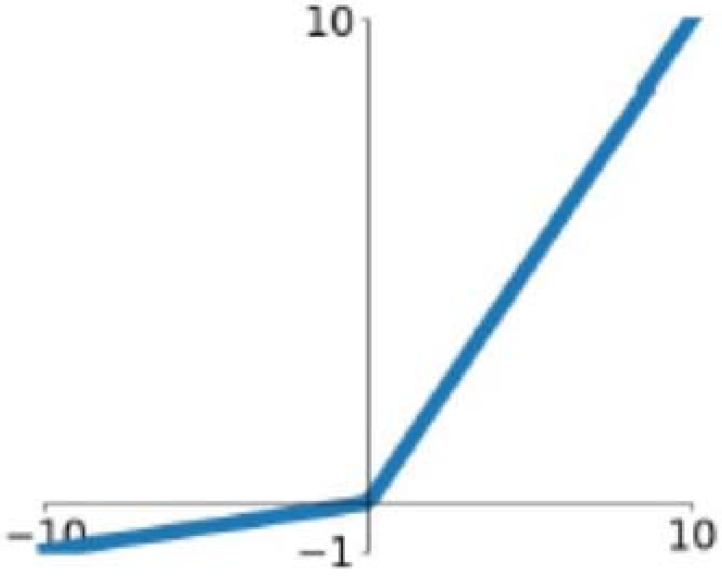
-- What happens when x = -10?

-- What happens when x = 0?

-- What happens when x = 10?

$$\frac{dRelu(x)}{dx} = ?$$

Leaky RELU Activation Function



$$LRelu(x) = \max(\alpha x, x)$$

α is small

- Squashes numbers to range [-∞, ∞]
- Does not saturate (full region)
- α is small, e.g. 0.01, can be learned

Tips for Choosing Activation Functions

- Typically choose ReLU
  -- Careful with learning rates
- Try leaky ReLU/ELU if ReLU does not work well
- Sometimes try tanh, but not expect too much
- No sigmoid for CNN, you can use in RNN

Importunate of Activation Function

- Activation function provides *non-linearity*

- Think about a simple one-hidden layer neural network

$$y = f(\mathbf{W_2} f(\mathbf{W_1 x}))$$

Q: What happens if there is no activation function?

Representation Power

- Neural Network is a universal function approximator

  -- E.g. classify images can be viewed as a function you cannot write explicitly

In the mathematical theory of artificial neural networks, the **universal approximation theorem** states[1] that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of $\mathbf{R}^n$, under mild assumptions on the activation function. The theorem thus states that simple neural networks can *represent* a wide variety of interesting functions when given appropriate parameters; however, it does not touch upon the algorithmic learnability of those parameters.

Let $\varphi : \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded, and continuous function. Let I_m denote the m-dimensional unit hypercube $[0,1]^m$. The space of real-valued continuous functions on I_m is denoted by $C(I_m)$. Then, given any $\varepsilon > 0$ and any function $f \in C(I_m)$, there exist an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$ for $i = 1, \ldots, N$, such that we may define:

$$F(x) = \sum_{i=1}^{N} v_i \varphi \left(w_i^T x + b_i \right)$$

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

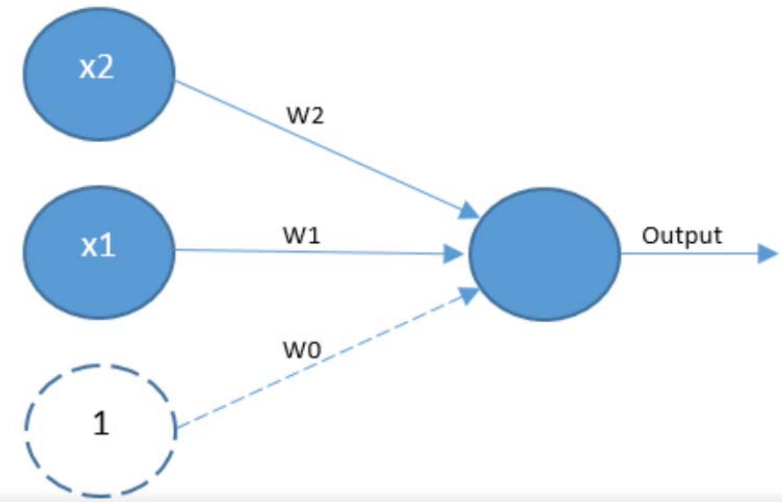
$$\boxed{|F(x) - f(x)| < \varepsilon}$$

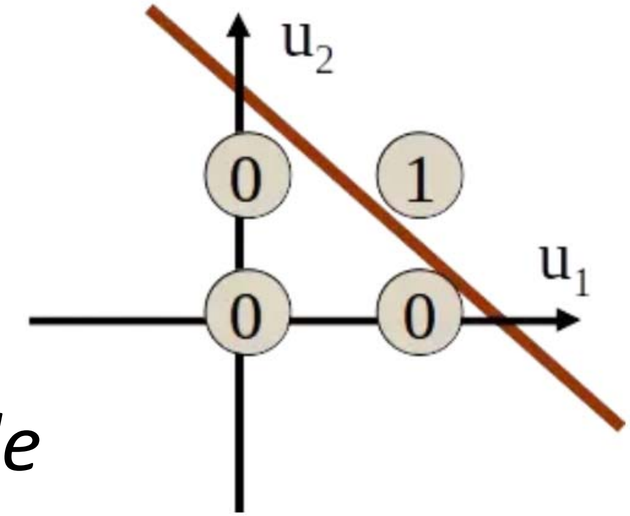
for all $x \in I_m$. In other words, functions of the form $F(x)$ are dense in $C(I_m)$.

Why Need Hidden Layer

- XOR problem

Input 1	Input 2	Output
0	0	0
0	1	1
1	1	0
1	0	1





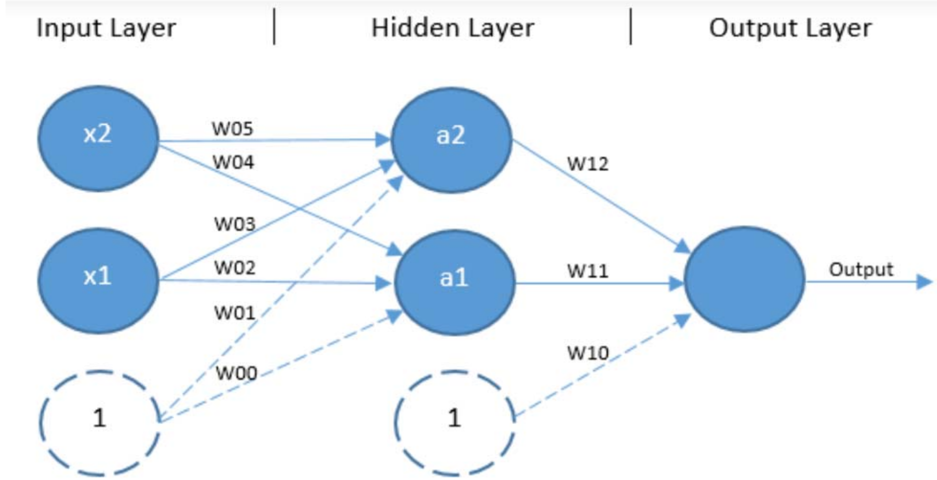
XOR is not linear separable

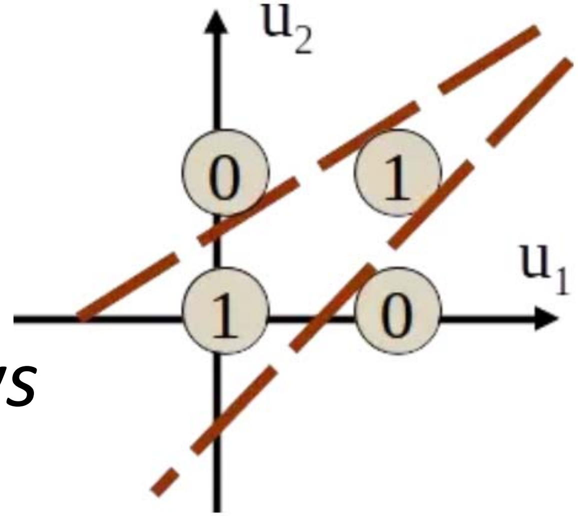
Jayesh Bapu Ahire@Medium

Why Need Hidden Layer

- XOR problem

Input 1	Input 2	Output
0	0	0
0	1	1
1	1	0
1	0	1





Multi-layer perceptions draws multiple lines

Jayesh Bapu Ahire@Medium

Why Use Many Layers

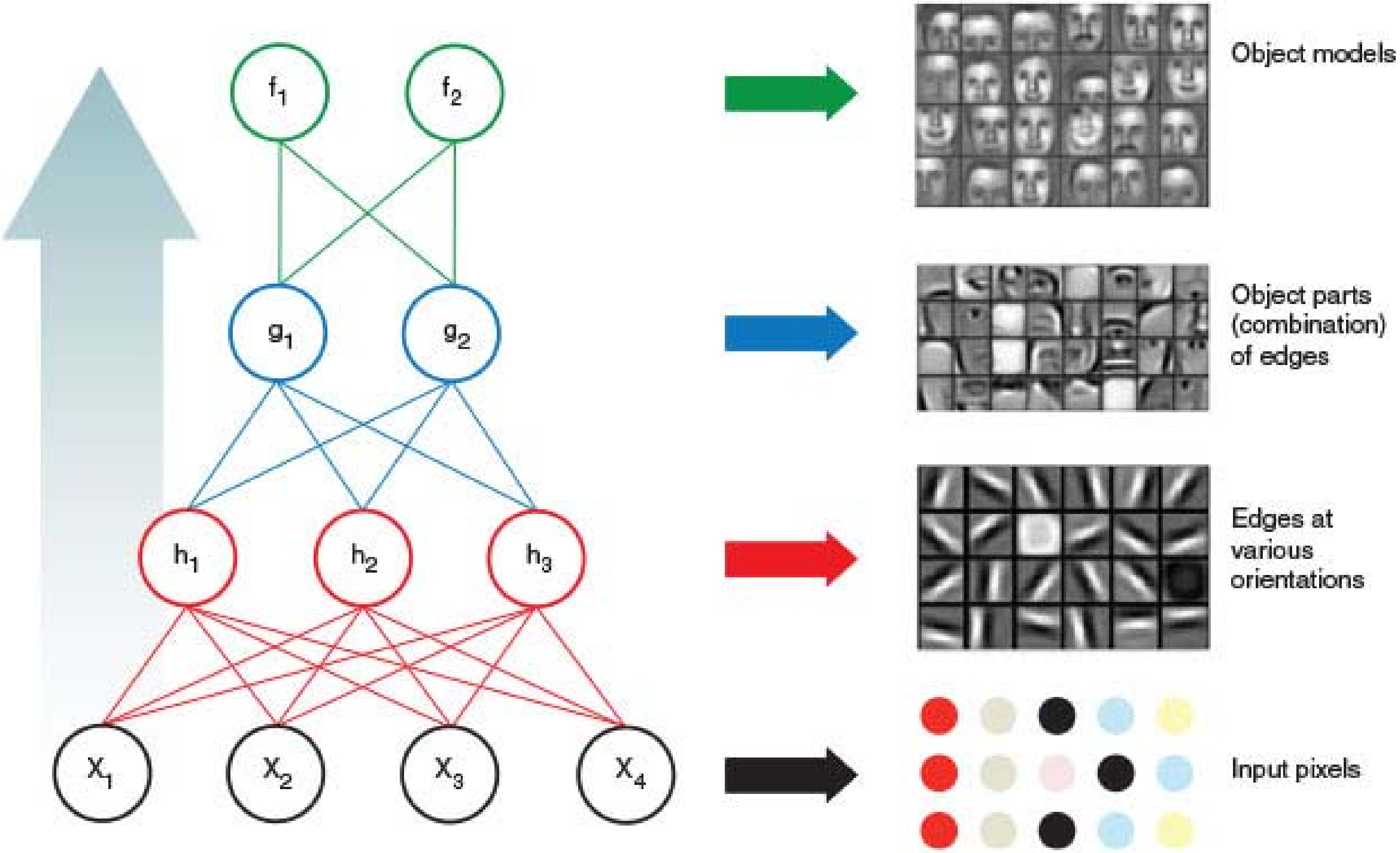
Deeper vs Wider?

- You can increase either depth or width to gain better performance
- Deeper network needs less neurons to achieve the same approximation error

First, we consider univariate functions on a bounded interval and require a neural network to achieve an approximation error of ε uniformly over the interval. We show that shallow networks (i.e., networks whose depth does not depend on ε) require $\Omega(\text{poly}(1/\varepsilon))$ neurons while deep networks (i.e., networks whose depth grows with $1/\varepsilon$) require $\mathcal{O}(\text{polylog}(1/\varepsilon))$ neurons. We then extend these results

Shiyu Liang etal. ICLR 2017

Why Use Many Layers



Deeper architecture facilitates hierarchy learning

Going deeper with convolutions

Christian Szegedy
Google Inc.

Wei Liu
University of North Carolina, Chapel Hill

Yangqing Jia
Google Inc.

Pierre Sermanet
Google Inc.

Scott Reed
University of Michigan

Dragomir Anguelov
Google Inc.

Dumitru Erhan
Google Inc.

Vincent Vanhoucke
Google Inc.

Andrew Rabinovich
Google Inc.

Abstract

We propose a deep convolutional neural network architecture codenamed Inception, which was responsible for setting the new state of the art for classification and detection in the ImageNet Large-Scale Visual Recognition Challenge 2014 (ILSVRC14). The main hallmark of this architecture is the improved utilization of the computing resources inside the network. This was achieved by a carefully crafted design that allows for increasing the depth and width of the network while keeping the computational budget constant. To optimize quality, the architectural decisions were based on the Hebbian principle and the intuition of multi-scale processing. One particular incarnation used in our submission for ILSVRC14 is called GoogLeNet, a 22 layers deep network, the quality of which is assessed in the context of classification and detection.

Simply Using Deep is not Good

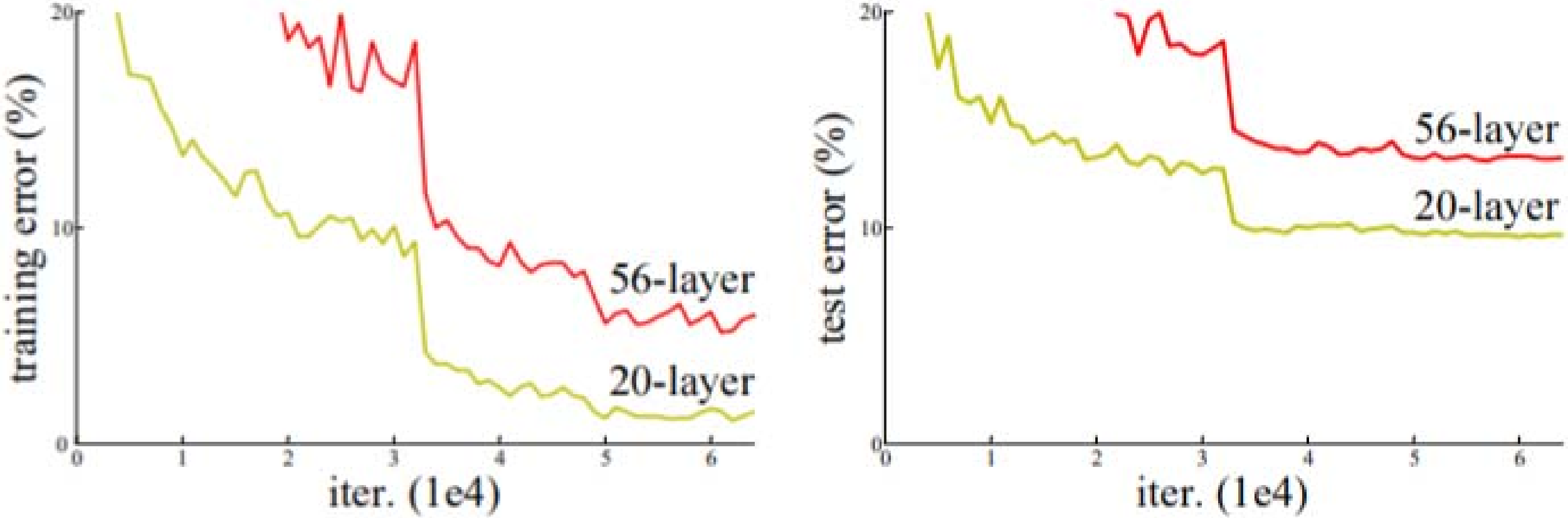
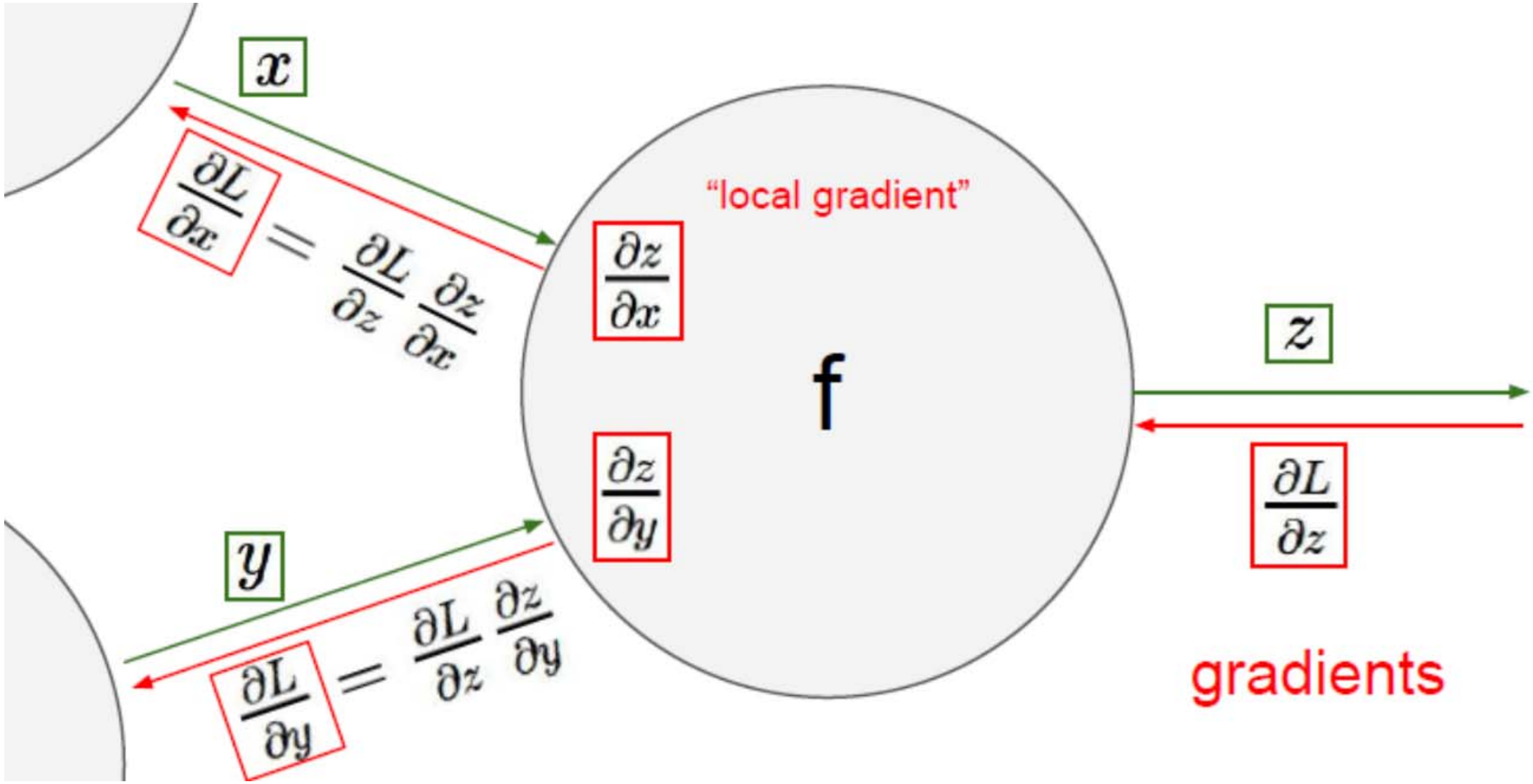


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

Kaiming He etal. CVPR 2015

Recall Vanish Gradient



Solution for Vanish Gradient

- Better initialization
- Faster hardware
- Using residual block

when the vanishing gradient problem was recognized. Schmidhuber notes that this "is basically what is winning many of the image recognition competitions now", but that it "does not really overcome the problem in a fundamental way"[11] since the original models tackling the vanishing gradient problem by

Note that ResNets are an ensemble of relatively shallow Nets and do not resolve the vanishing gradient problem by preserving gradient flow throughout the entire depth of the network – rather, they avoid the problem simply by constructing ensembles of many short networks together. (Ensemble by Construction[16])

Acknowledgement

Many materials of the slides of this course are adopted and re-produced from several deep learning courses and tutorials.

- Prof. Fei-fei Li, Stanford, CS231n: Convolutional Neural Networks for Visual Recognition (online available)
- Prof. Andrew Ng, Stanford, CS230: Deep learning (online available)
- Prof. Yanzhi Wang, Northeastern, EECE7390: Advance in deep learning
- Prof. Jianting Zhang, CUNY, CSc G0815 High-Performance Machine Learning: Systems and Applications
- Prof. Vivienne Sze, MIT, "Tutorial on Hardware Architectures for Deep Neural Networks"
- Pytorch official tutorial https://pytorch.org/tutorials/