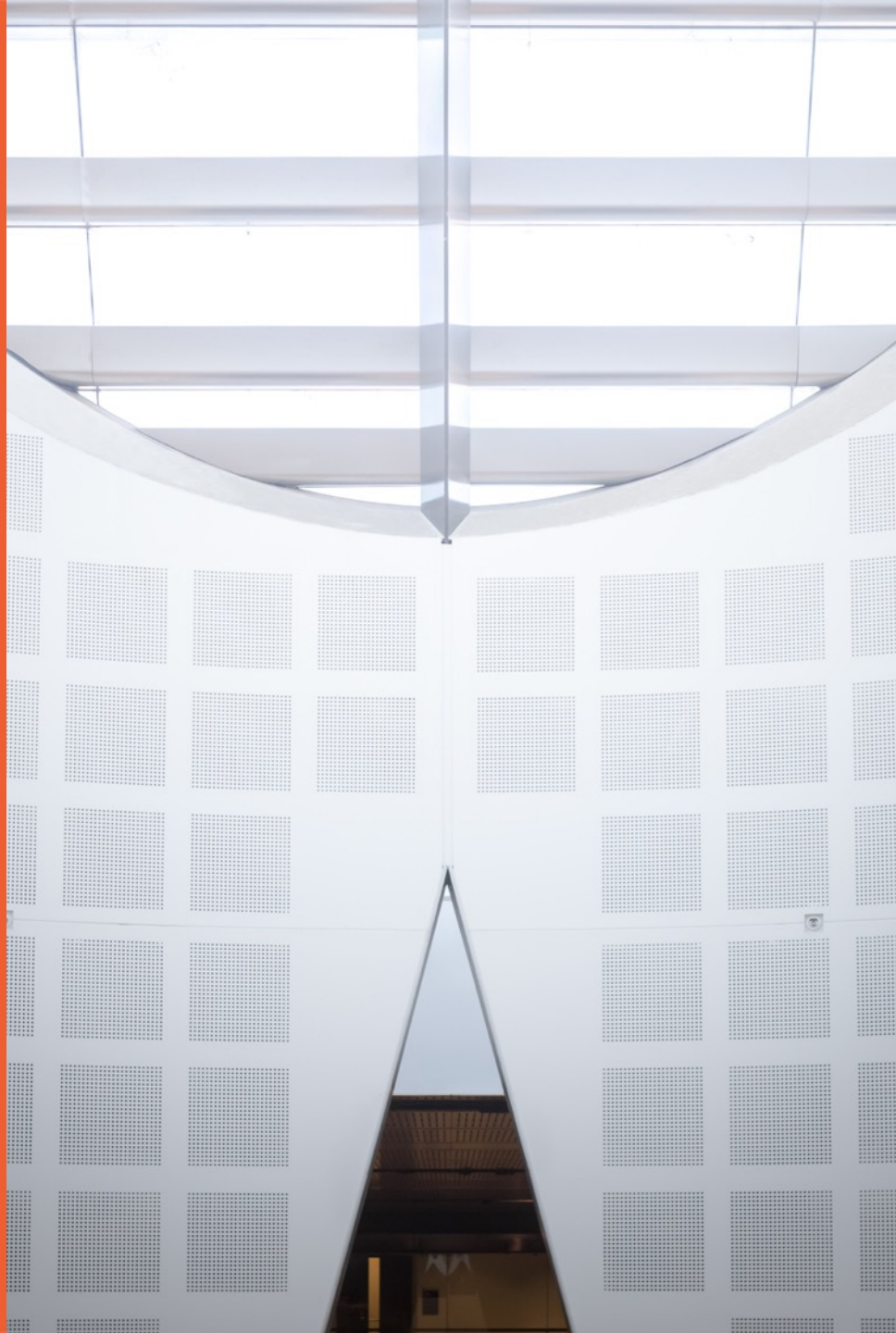
Deep Learning
COMP 5329

Dr Chang Xu
Prof Dacheng Tao
UBTECH Sydney AI Centre

THE UNIVERSITY OF SYDNEY




Daryl
Scott
Bella
Emil
Xinyu
David
Trung
Lily
Prerona
Richard
Handuo
Aditya
Laurence
Sriram
Daniel
Sabarish
Darius
Brian
Nicholas
William
Chenghao
Alister
Wenzhe
Michael
Fangzhou
Elija
Vincent
James
Quang
Rishu
Chengzhi
Sean
Matthew
Shanthini
Mingshen
Kwon
Sylvia
Flora
Luke
Lucas
Chen
Ryan
Rennie
Jialu
Nick
Minjing
Yixiong
Gerhardus
Youyou
Grant
Xia
Samuel
Leslie
Bolivar
Rui
Thomas
Nung
Christine
Qiuqi
Amy
Shailesh
Charles
Kristopher
Nisal
Josh
Phil
Lopez
Tushar
Yanyu
Owen
Scott
Mike
Shumin
Zhan
Yun
Kevin
Tom
Ken
Xinyu
Junpeng
Tianfeng
Ujjwal
Elaine
Joseph
Andrew
Alison
Zhengyang
Sean
Eric
Steven
Wang
Xiong
Chengcheng
Jiahao
Cameron
Jielong
Jun
Sonia
Danqing
Lingzuo
Yuri
Julius
Xin
Lance
Yin
Dongdong
Qiang
Meng
Rui
Francis
Elena
Aoyang
Chester
Yixuan
Zequn
Tim
Sicheng

Learning

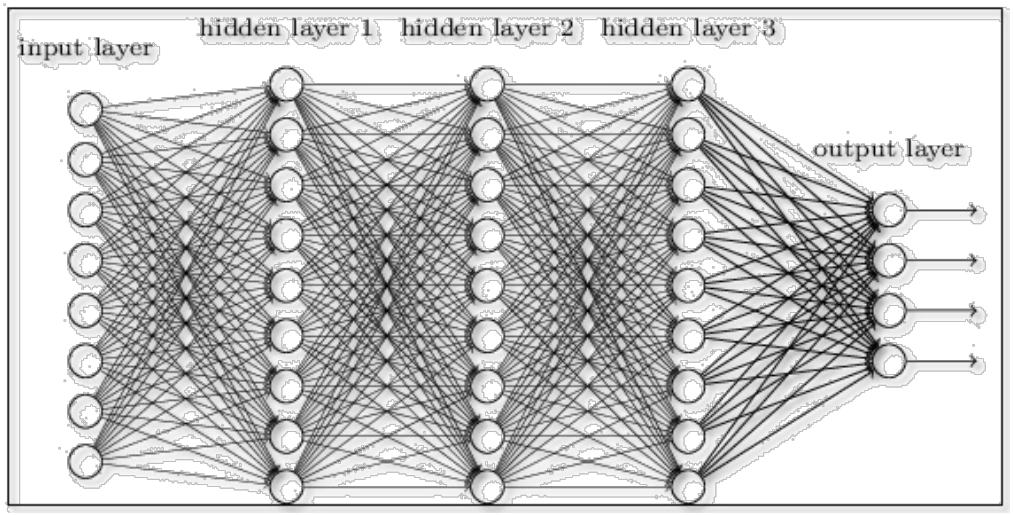
Unit of Study Survey (USS)

What is Deep Learning?

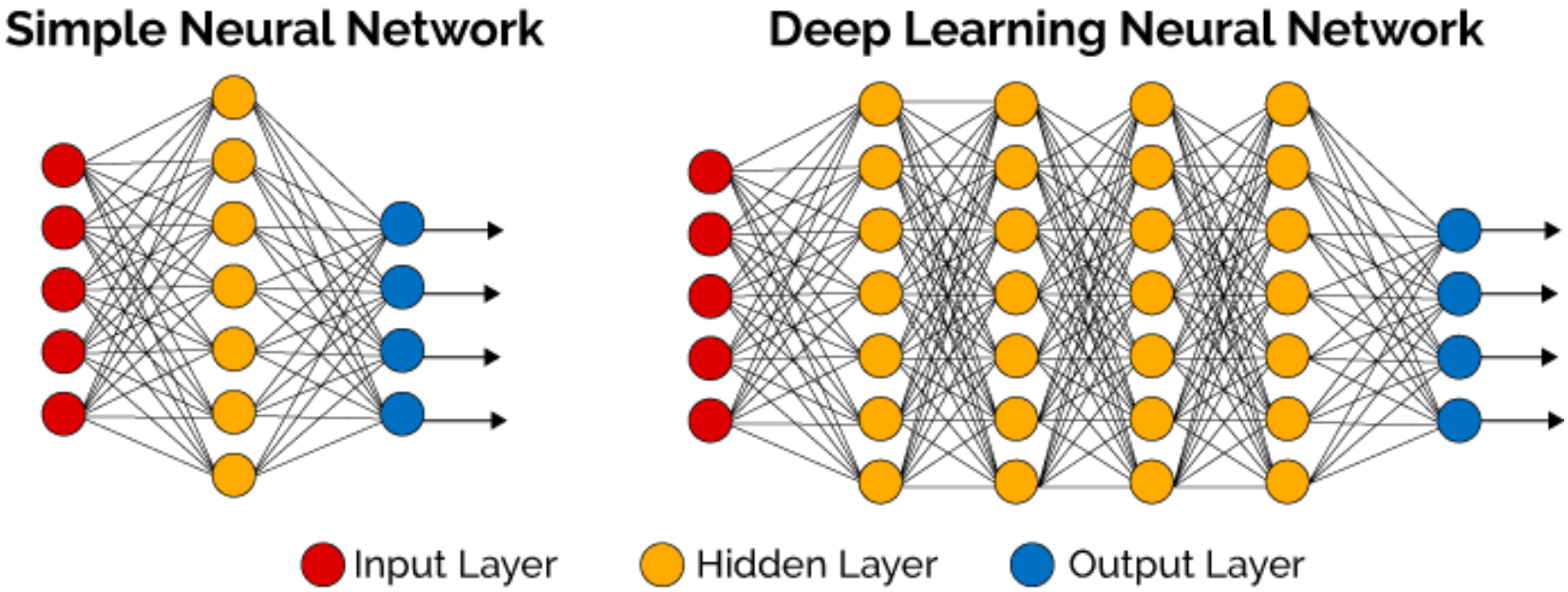
The short answers

1. **'Deep Learning' means** using a **neural network** with **several layers of nodes** between input and output

2. **the series of layers between input & output to do feature identification and processing in a series of stages, just as our brains seem to.**



What is Deep Learning?

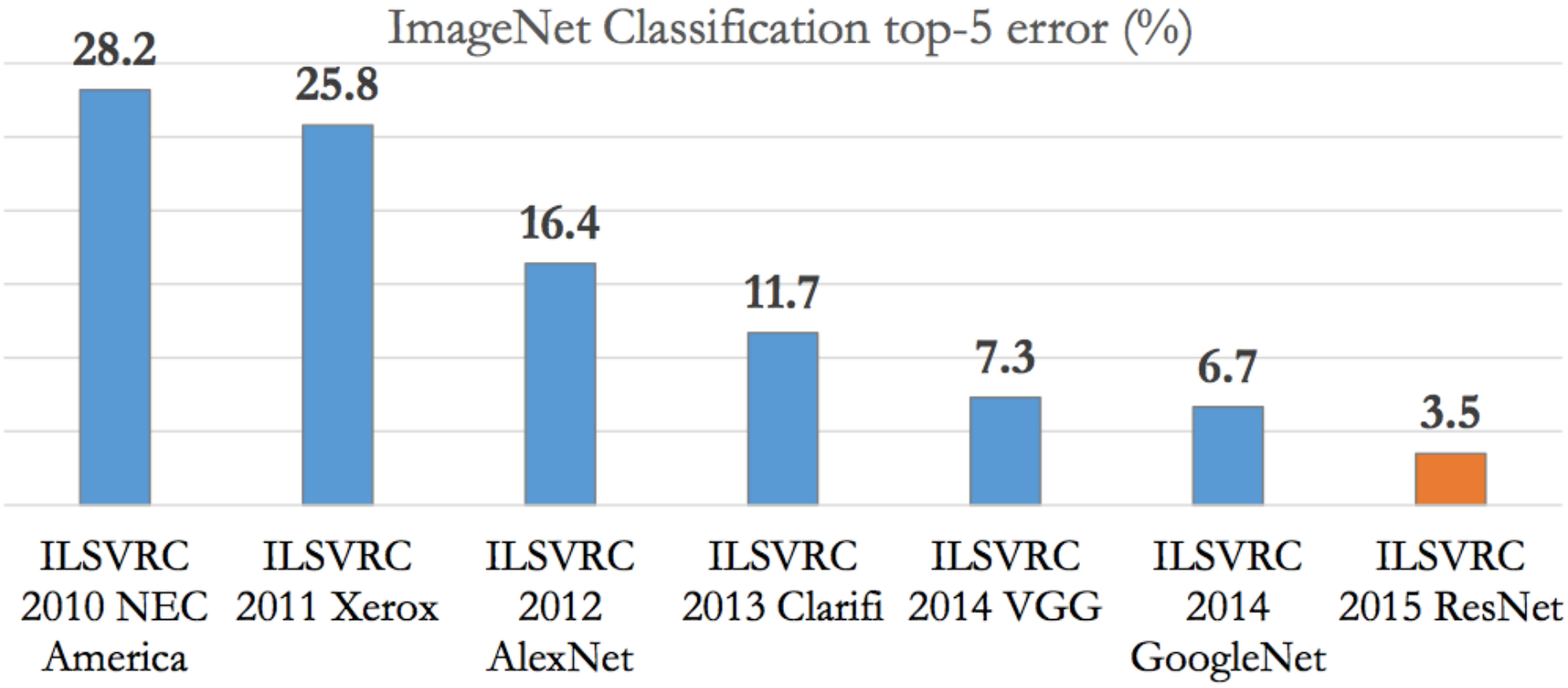


We have always had good algorithms for learning the weights in networks with 1 hidden layer;

But these algorithms are not good at learning the weights for networks with more hidden layers

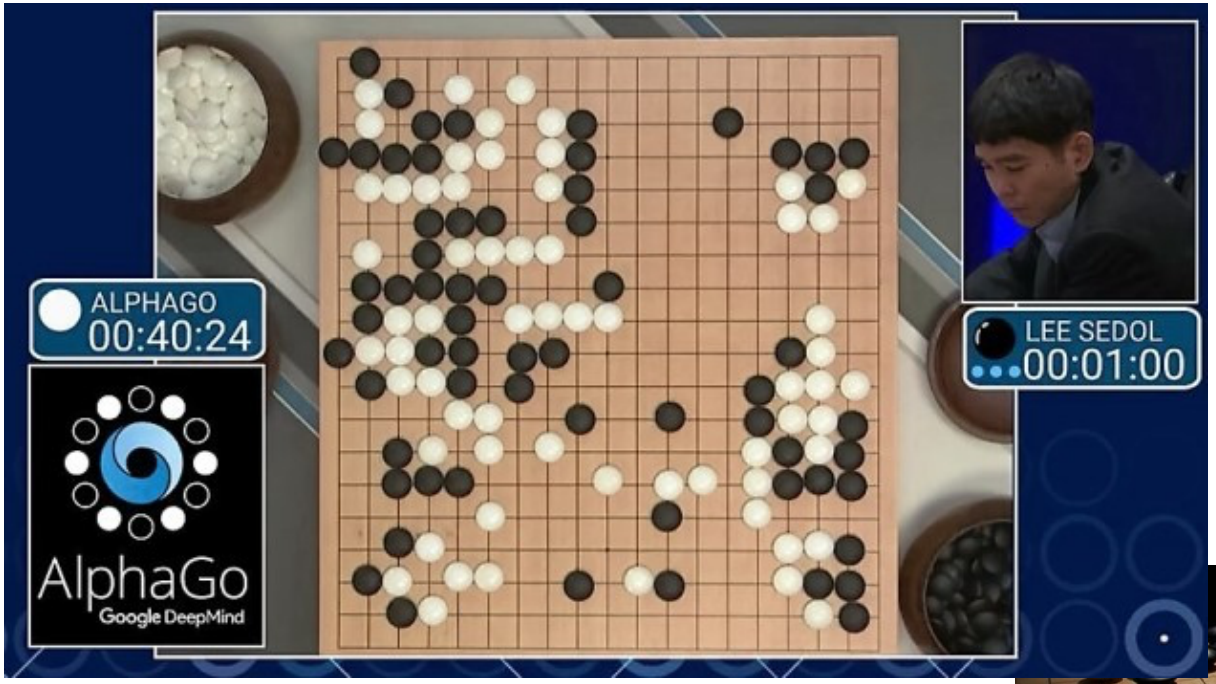
Image credit to becominghuman.ai

Why to Study Deep Learning?



Human: 5.1% Top-5 error rate.

Why to Study Deep Learning?



Google Deep Mind

AlphaGO defeats world champion

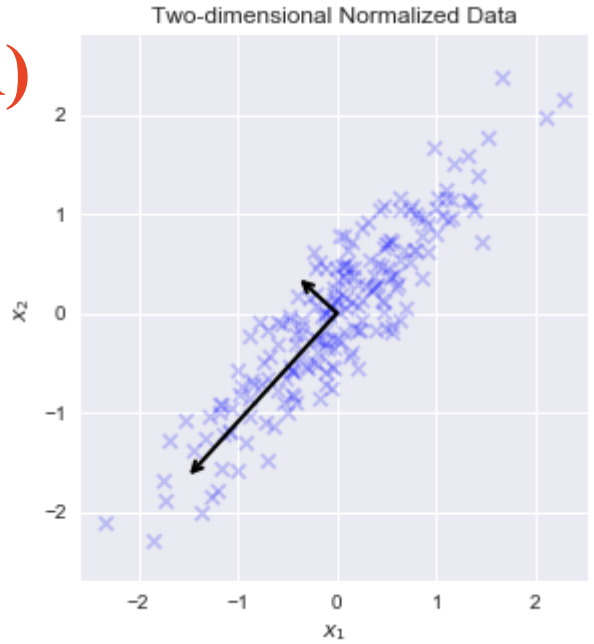
Why to Study Deep Learning?

What to Learn?

1. Introduction to Deep Learning
2. Unsupervised feature learning
3. Multilayer neural network
4. Convolutional neural networks (CNN)
5. Optimization for Training Deep Models
6. Network structures
7. Recurrent Neural Networks and LSTM
8. Deep learning applications
9. Generative Models
10. Deep reinforcement learning

Unsupervised Feature Learning

Principal Component Analysis (PCA)



Objective: Maximize Variance

Let $\boldsymbol{X} = \{\boldsymbol{x}_i \in \boldsymbol{R}^d\}_{1\leq i\leq n}$ be the sample, project $\boldsymbol{X}$ onto a k dimensional subspace ($k<d$) such that the variance of the projected data is maximized.

For example, $k = 1$:

$$\widehat{Var} = \frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{u}_1^\top \boldsymbol{x}_i - \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{u}_1^\top \boldsymbol{x}_i)^2 = \boldsymbol{u}_1^\top \boldsymbol{S}\boldsymbol{u}_1$$

Objective: Minimize Error

$$\min_{\boldsymbol{U}} \frac{1}{n}\sum_{i=1}^{n}\|\boldsymbol{x}_i - \boldsymbol{U}\boldsymbol{U}^\top \boldsymbol{x}_i\|^2 \qquad s.t. \quad \boldsymbol{U}^\top \boldsymbol{U} = \boldsymbol{I}$$

Algorithm: Compute scatter matrix $\boldsymbol{S} = \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{z}_i\boldsymbol{z}_i^\top$ of normalized data, Compute the eigenvectors and eigenvalues by using SVD of $\boldsymbol{S}$. Then transform the data $\boldsymbol{y}_i = \boldsymbol{U}^\top \boldsymbol{x}_i$.

Canonical Correlation Analysis (CCA)

Objective: Maximize correlation of $\boldsymbol{u}^\top X$ and $\boldsymbol{v}^\top Y$, which are projected variables of two random vectors under projection vectors $\boldsymbol{u}$ and $\boldsymbol{v}$.

For example, $k = 1$:

With the definition of $\widehat{cov}$ and $\widehat{corr}$, model the problem as

$$\max_{\boldsymbol{u}_1, \boldsymbol{v}_1} \frac{\boldsymbol{u}_1^\top \boldsymbol{S}_{xy} \boldsymbol{v}_1}{\sqrt{\boldsymbol{u}_1^\top \boldsymbol{S}_{xx} \boldsymbol{u}_1} \sqrt{\boldsymbol{v}_1^\top \boldsymbol{S}_{yy} \boldsymbol{v}_1}}$$

and equals to: $\max_{\boldsymbol{u}_1, \boldsymbol{v}_1} \boldsymbol{u}_1^\top \boldsymbol{S}_{xy} \boldsymbol{v}_1 \quad s.t. \quad \boldsymbol{u}_1^\top \boldsymbol{S}_{xx} \boldsymbol{u}_1 = 1, \boldsymbol{v}_1^\top \boldsymbol{S}_{yy} \boldsymbol{v}_1 = 1$

Algorithm: Scatter matrix computation, SVD of $\boldsymbol{S}_{xx}^{-1/2} \boldsymbol{S}_{xy} \boldsymbol{S}_{yy}^{-1/2} = \boldsymbol{W}_u \boldsymbol{D} \boldsymbol{W}_v^\top$,

and projection matrix: $\boldsymbol{U} = \boldsymbol{S}_{xx}^{-1/2} \boldsymbol{W}_u[:, 1{:}k]$, and $\boldsymbol{V} = \boldsymbol{S}_{yy}^{-1/2} \boldsymbol{W}_v[:, 1{:}k]$.

Autoencoder (AE)

Objective: Learn discriminative representations (through encoder and decoder)

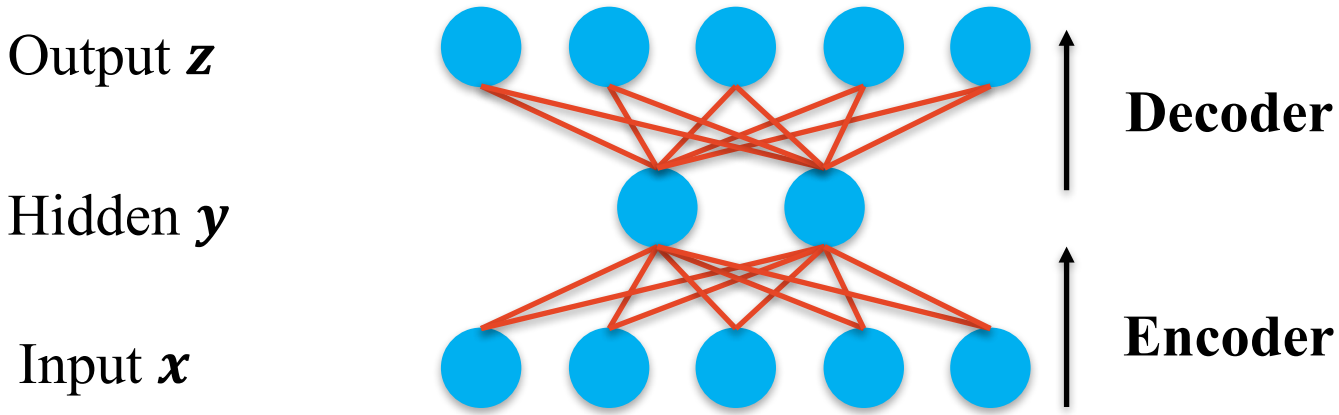
Encoder and decoder function:

$$\boldsymbol{y} = f\left(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right), \boldsymbol{z} = f\left(\boldsymbol{W}^{(2)}\boldsymbol{y} + \boldsymbol{b}^{(2)}\right)$$

Minimize reconstruction error (difference between $\boldsymbol{z}$ and $\boldsymbol{x}$)

- Square loss: $J(\theta) = \|\boldsymbol{x} - \boldsymbol{z}\|^2, \theta = \left\{\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}, \boldsymbol{W}^{(2)}, \boldsymbol{b}^{(2)}\right\}$
- Cross-entropy: $J(\theta) = -\sum_{i=1}^{d} x_i \log(z_i) + (1 - x_i)\log(1 - z_i)$

Algorithm: Backpropagation (BP)

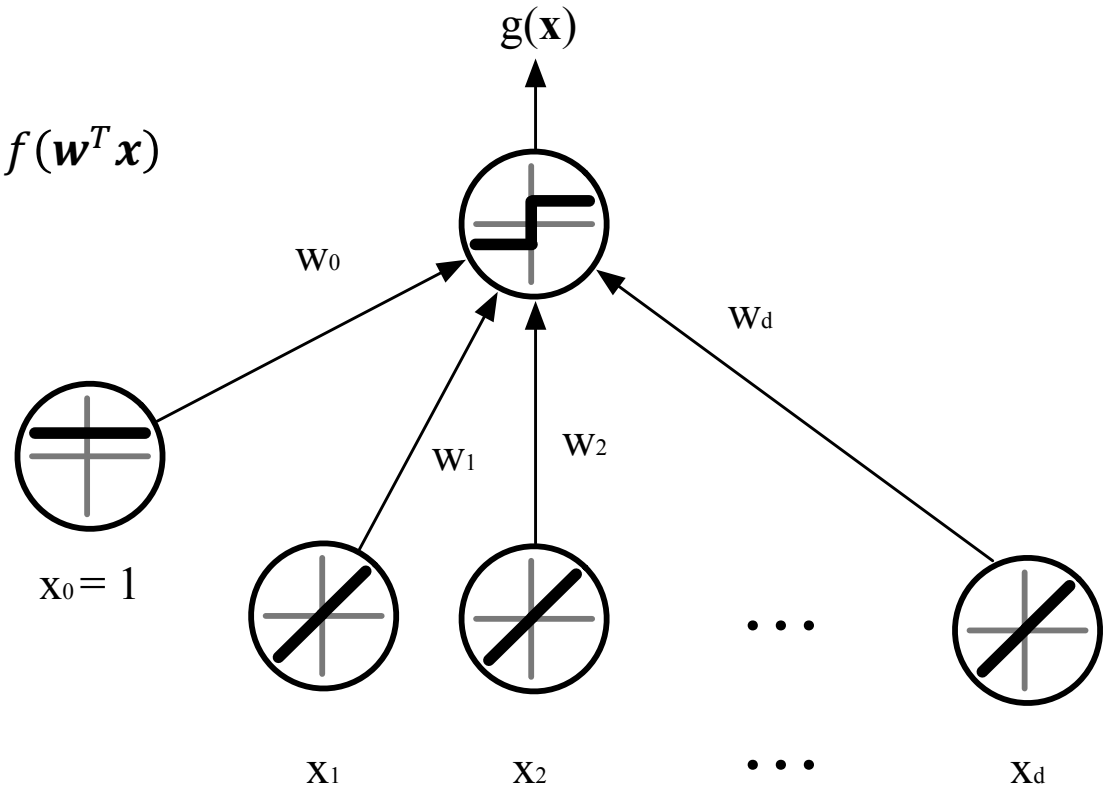


Multilayer Neural Network

Perceptron

$$g(\boldsymbol{x}) = f\left(\sum_{i=1}^{d} x_i w_i + w_0\right) = f(\boldsymbol{w}^T \boldsymbol{x})$$

$$f(s) = \begin{cases} 1, & if\ s \geq 0 \\ -1, & if\ s < 0 \end{cases}$$

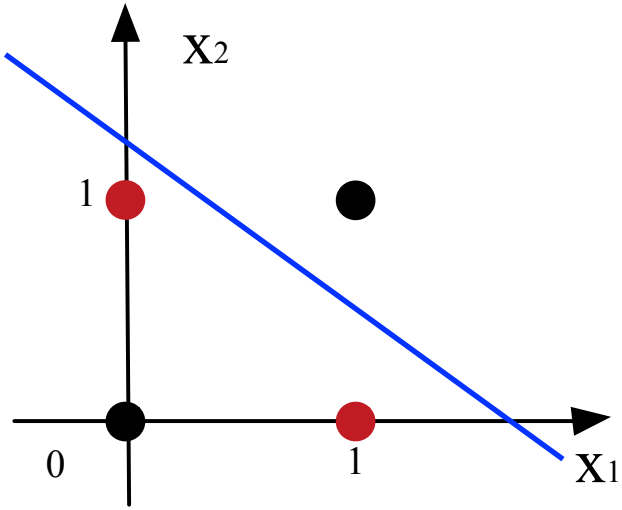


Linear classifier!

Perceptron: Limitations

- Solving XOR (exclusive-or) with Perceptron?

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



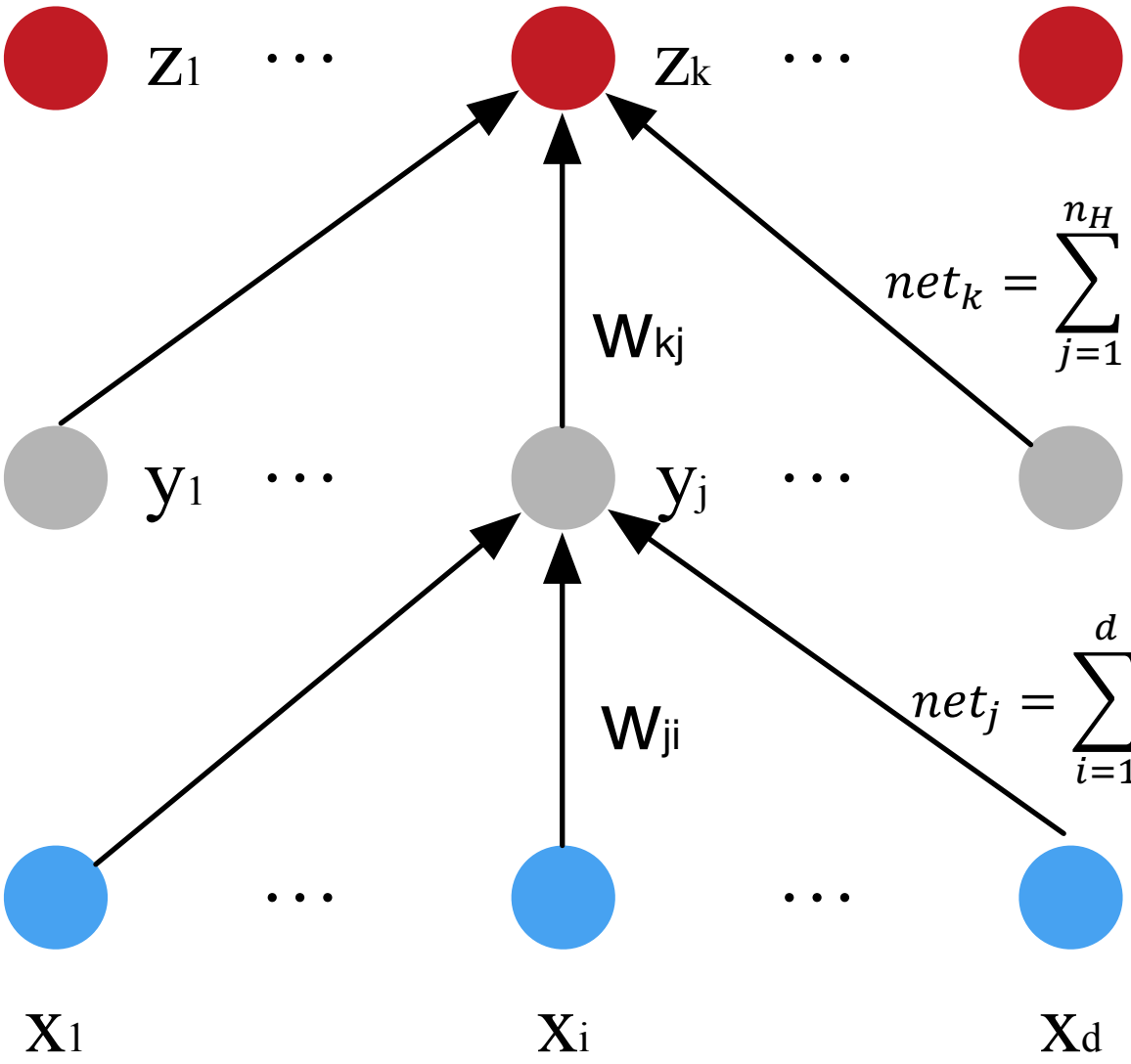
- Linear classifier cannot identify non-linear patterns.

$$w_1 x_1 + w_2 x_2 + b$$

This failure caused the majority of researchers to walk away.

Feedforward

$$z_k = f(net_k)$$
$$= f\left(\sum_{j=1}^{n_H} w_{kj} f(net_j) + w_{k0}\right)$$



$$net_k = \sum_{j=1}^{n_H} y_j w_{kj} + w_{k0} = \sum_{j=0}^{n_H} y_j w_{kj} = \boldsymbol{w}_k^T \boldsymbol{y}$$

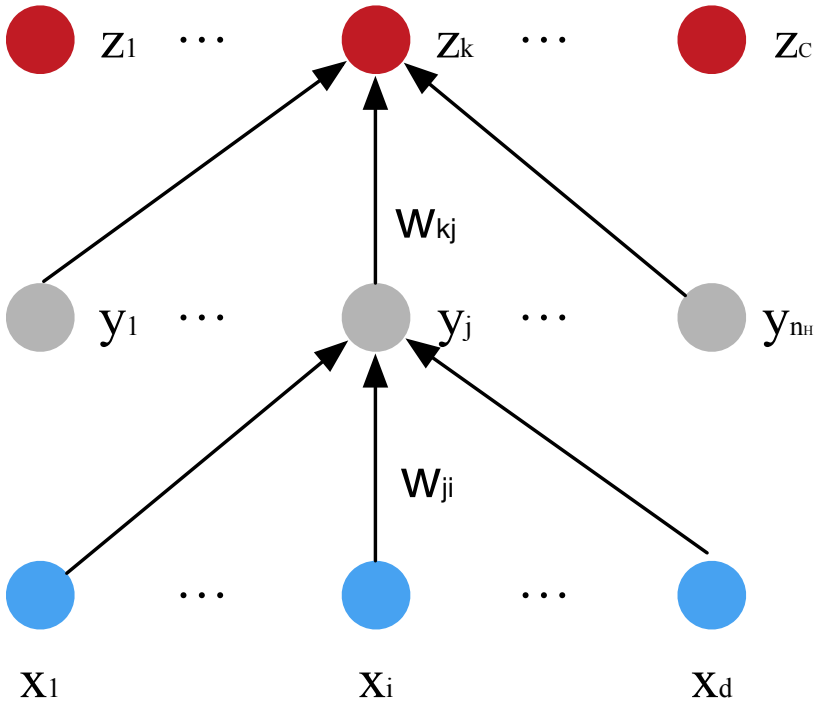
$$net_j = \sum_{i=1}^{d} x_i w_{ji} + w_{j0} = \sum_{i=0}^{d} x_i w_{ji} = \boldsymbol{w}_j^T \boldsymbol{x}$$

Backpropagation summary

$$\delta_k = -\frac{\partial J}{\partial net_k} \qquad \delta_j = -\frac{\partial J}{\partial net_j}$$

- Hidden-to-output weight

$$\Delta w_{kj} = \eta \delta_k \, y_j$$
$$= \eta (t_k - z_k) f'(net_k) y_j$$



- Input-to-hidden

$$\Delta w_{ji} = \eta \delta_j x_i$$
$$= \eta \left[\sum_{k=1}^{C} \delta_k w_{kj} \right] f'(net_j) x_i$$

Universal Approximation Theorem

Let $f(\cdot)$ be a nonconstant, bounded, and monotonically-increasing continuous function. Let I_m denote the m-dimensional unit hypercube $[0,1]^m$. The space of 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 $v_i, b_i \in \mathbb{R}^m$ and real vectors $w_i \in \mathbb{R}^m$, where $i = 1, \cdots, N_i$ such that we may define:

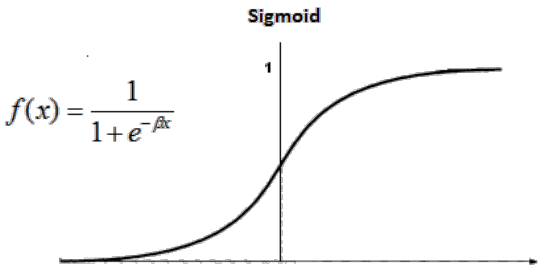
$$\hat{F}(x) = \sum_{i}^{N} v_i f(w_i^T x + b_i)$$

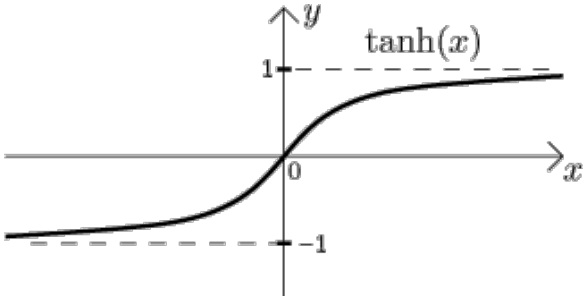
as an approximate realization of the function F where F is independent of f; that is,

$$\left|\hat{F}(x) - F(x)\right| < \varepsilon$$

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

Universality theorem (Hecht-Nielsen 1989): "Neural networks with a single hidden layer can be used to approximate any continuous function to any desired precision."

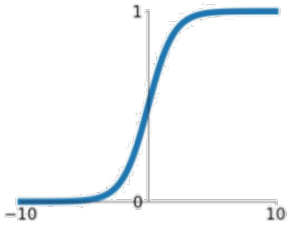




Activation function $f(s)$

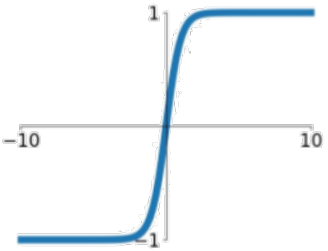
- Popular choice of activation functions (single input)
  - Sigmoid function

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



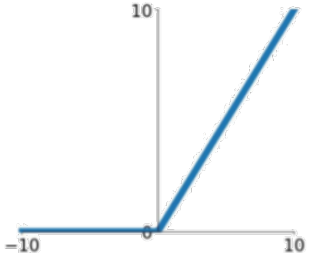
  - Tanh function: shift the center of Sigmoid to the origin

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



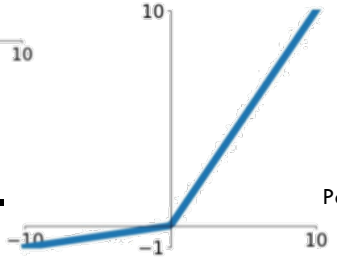
  - Rectified linear unit (ReLU)

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



  - Leaky ReLU

$$f(s) = \begin{cases} s, & \text{if } s \geq 0 \\ \alpha s, & \text{if } s < 0 \end{cases}, \alpha \text{ is a small constant.}$$



Optimization for Training Deep Models

Momentum

The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions.

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

Momentum term γ is usually set to 0.9 or a similar value.

Nesterov accelerated gradient (NAG)

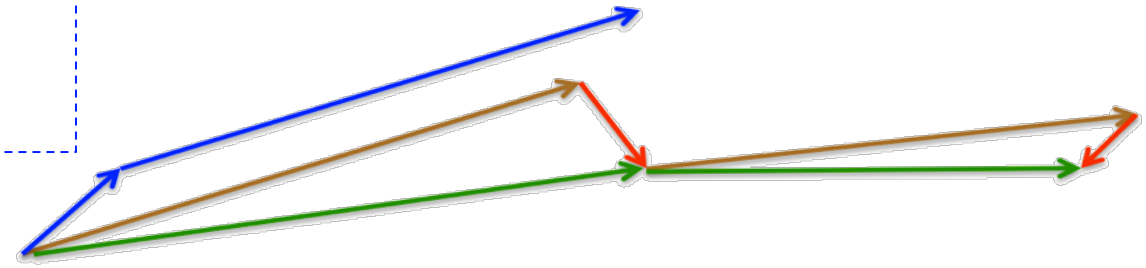
First make a big jump in the direction of the previous accumulated gradient.
Then measure the gradient where you end up.
Finally accumulate the gradient and make a correction.

First make a big jump

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Then measure the gradient where you end up

Finally accumulate the gradient and make a correction



blue vectors = standard momentum
brown vector = jump, red vector = correction,
green vector = accumulated gradient ,

- **Adagrad** $\theta_t = [\theta_{t,1}, \theta_{t,2}, \cdots, \theta_{t,i}, \cdots, \theta_{t,d}]$

- **Original gradient descent:** The same for all parameters

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot \nabla_\theta J(\theta_i)$$

Perform an update for all parameters using the same learning rate.

- **Adagrad:** Adapt the learning rate to the parameters based on the past gradients that have been computed for θ_i.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot \nabla_\theta J(\theta_i)$$

$G_t \in R^{d \times d}$ is a diagonal matrix where each diagonal element is the sum of the squares of the gradients w.r.t. θ_i up to time step t.

Adadelta – 1 Accumulate over window

- Storing fixed previous squared gradients cannot accumulate to infinity and instead becomes a local estimate using recent gradients.
- Adadelta implements it as an exponentially decaying average of all past squared gradients.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

$$RMS[g]_t = \sqrt{E[g^2]_t + \epsilon}$$

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t$$

Adadelta – 2 Correct units with Hessian approximation

$$H^{-1}g = \Delta\theta$$

$$\Delta\theta = \frac{\frac{\partial J}{\partial \theta}}{\frac{\partial^2 J}{\partial \theta^2}} \Rightarrow H^{-1} \propto \frac{1}{\frac{\partial^2 J}{\partial \theta^2}} \propto \frac{\frac{\partial J}{\partial \theta}}{\frac{\partial^2 J}{\partial \theta^2}} \propto \frac{\Delta\theta}{g}$$

We assume the curvature is locally smooth and approximate $\Delta\theta_t$ by computing the exponentially decaying RMS of $\Delta\theta$.

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t \longrightarrow \Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

RMSprop

- RMSprop and Adadelta have both been developed independently around the same time.
- RMSprop in fact is identical to the first update vector of Adadelta.

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Hinton suggests γ to be set to 0.9, while a good default value for the learning rate η is 0.001.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

Adam

Combine the advantages of **Adgrad** and **RMSprop**.

- Adgrad: adaptive learning rate for different parameters.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} m_t$$

- RMSprop: exponentially decaying average.
  - Store an exponentially decaying average of past squared gradients v_t .

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Keep an exponentially decaying average of past gradients m_t

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

Initialization

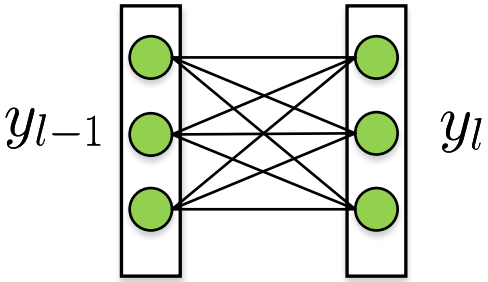
All zero initialization:

Every neuron computes the same output.

They will also compute the same gradients.

They will undergo the exact same parameter updates.

There will be no difference between all the neurons.



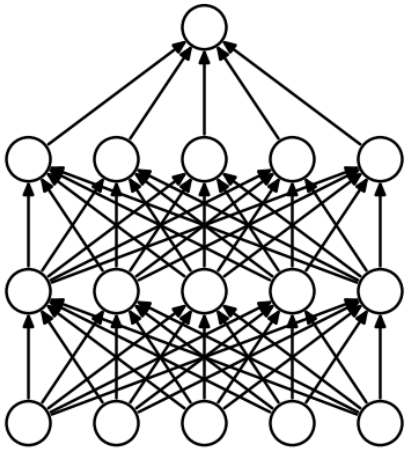
$$y_l = W y_{l-1} + b, \quad \frac{\partial \mathcal{L}}{\partial y_{l-1}} = W^\top \frac{\partial \mathcal{L}}{\partial y_l}$$
$$if \quad W = 0, \quad then \quad \frac{\partial \mathcal{L}}{\partial y_{l-1}} = 0$$
$$then, \quad \frac{\partial \mathcal{L}}{\partial y_m} = 0, \quad m = 0, ...l-1.$$

Initialization (Xavier) -- [Xavier Glorot, Yoshua.Bengio 2010]

The variance of the inputs and outputs are the same for all layers.

But the variance of the gradients might still vanish or explode as we consider deeper networks.

$$s^i = Z^i W^i + b^i$$
$$z^{i+1} = f(s^i)$$
$$\mathrm{Var}[\frac{\partial Cost}{\partial s^i}] = [n\mathrm{Var}[W]]^{d-i}\mathrm{Var}[x]$$
$$\mathrm{Var}[\frac{\partial Cost}{\partial w^i}] = [n\mathrm{Var}[W]]^{d}\mathrm{Var}[x]\mathrm{Var}[\frac{\partial Cost}{\partial s^i}]$$



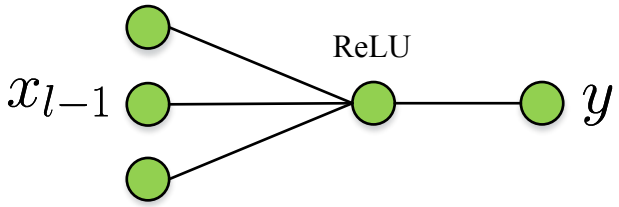
$$W \sim U[-\frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}]$$

U[−a, a] is the uniform distribution in the interval (−a, a).

Initialization（He） -- [Kaiming He, Xiangyu Zhang, et al. 2015]

If we use ReLU as the activation function, n is the number of inputs, then for layer l we have:

$$\mathrm{Var}(y_l) = (n\mathrm{Var}(w))\, E[x_{l-1}^2]$$
$$x_{l-1} = max(0, y_{l-1})$$



If y_{l-1} has zero mean, and has a symmetric distribution around zero:

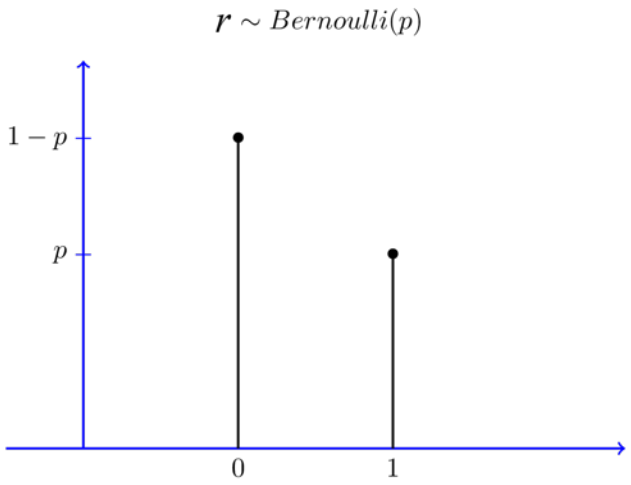
$$E[x_{l-1}^2] = E[(max(0, y_{l-1}))^2] = \frac{1}{2}\mathrm{Var}[y_{l-1}]$$

So we have:

$$\boxed{\mathrm{Var}(w) = \frac{2}{n}}$$

Dropout

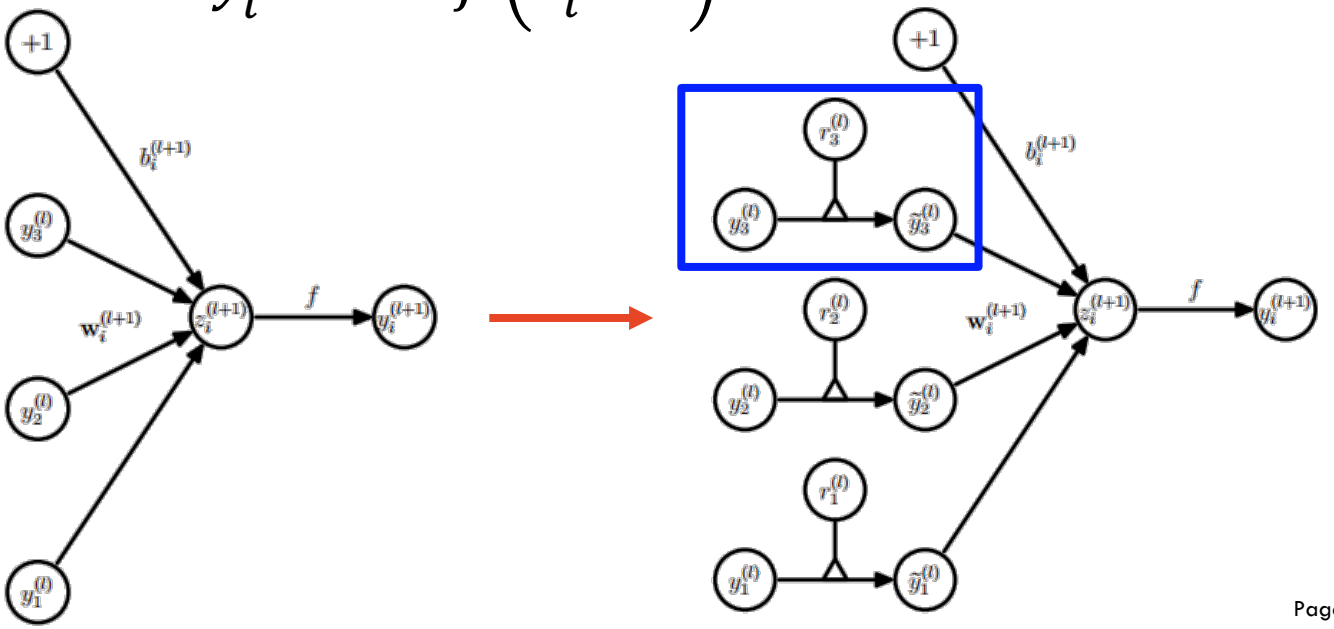
With dropout: $r_i^{(l)} \sim \text{Bernoulli}(p)$



$$\tilde{\boldsymbol{y}}^{(l)} = \boldsymbol{r}^{(l)} * \boldsymbol{y}^{(l)}$$

$$z_i^{(l+1)} = \boldsymbol{w}_i^{(l+1)} \tilde{\boldsymbol{y}}^{(l)} + b_i^{(l+1)}$$

$$y_i^{(l+1)} = f\left(z_i^{(l+1)}\right)$$



Batch Normalization

Input: Network N with trainable parameters Θ;
subset of activations $\{x^{(k)}\}_{k=1}^{K}$
Output: Batch-normalized network for inference, $N_{\text{BN}}^{\text{inf}}$

1: $N_{\text{BN}}^{\text{tr}} \leftarrow N$ // Training BN network
2: **for** $k = 1 \ldots K$ **do**
3: Add transformation $y^{(k)} = \text{BN}_{\gamma^{(k)},\beta^{(k)}}(x^{(k)})$ to $N_{\text{BN}}^{\text{tr}}$ (Alg. 1)
4: Modify each layer in $N_{\text{BN}}^{\text{tr}}$ with input $x^{(k)}$ to take $y^{(k)}$ instead
5: **end for**
6: Train $N_{\text{BN}}^{\text{tr}}$ to optimize the parameters $\Theta \cup \{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^{K}$
7: $N_{\text{BN}}^{\text{inf}} \leftarrow N_{\text{BN}}^{\text{tr}}$ // Inference BN network with frozen // parameters
8: **for** $k = 1 \ldots K$ **do**
9: // For clarity, $x \equiv x^{(k)}, \gamma \equiv \gamma^{(k)}, \mu_{\mathcal{B}} \equiv \mu_{\mathcal{B}}^{(k)}$, etc.
10: Process multiple training mini-batches $\mathcal{B}$, each of size m, and average over them:

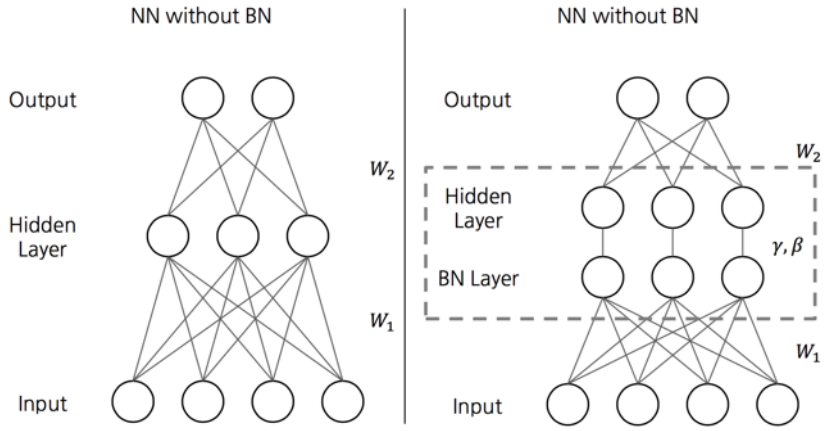
$$\text{E}[x] \leftarrow \text{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$$
$$\text{Var}[x] \leftarrow \frac{m}{m-1}\text{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$$

11: In $N_{\text{BN}}^{\text{inf}}$, replace the transform $y = \text{BN}_{\gamma,\beta}(x)$ with $y = \frac{\gamma}{\sqrt{\text{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\text{E}[x]}{\sqrt{\text{Var}[x]+\epsilon}}\right)$
12: **end for**

Algorithm 2: Training a Batch-Normalized Network

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

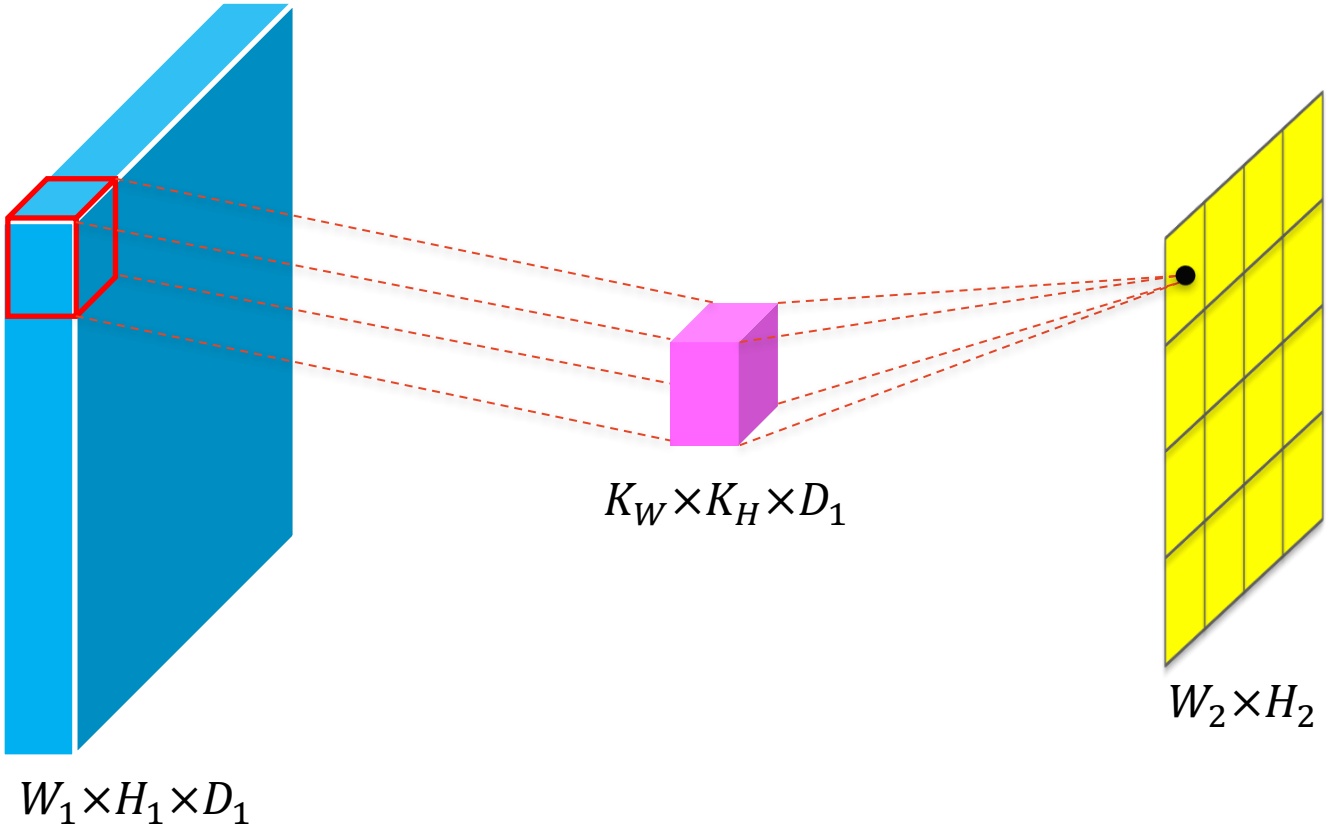


Convolutional Neural Networks

Convolutional Layer

- Suppose stride is (S_W, S_H) and pad is (P_W, P_H)

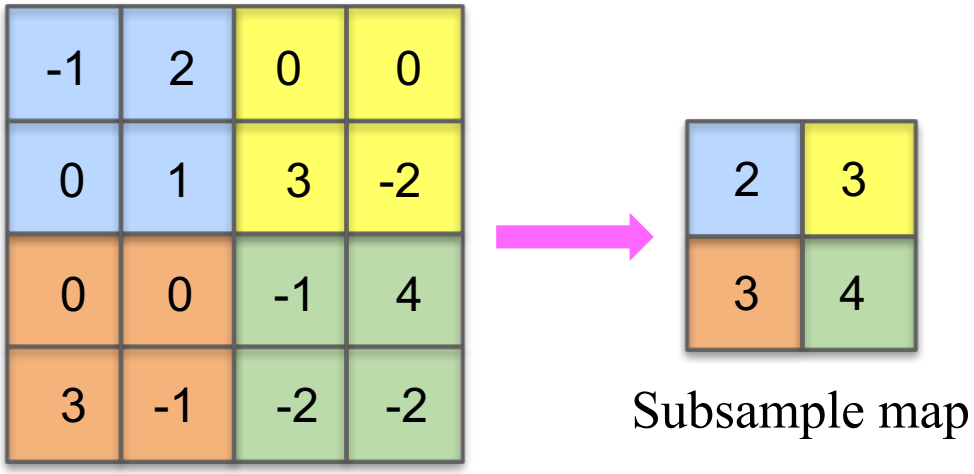
$$W_2 = \frac{W_1 + 2P_W - K_W}{S_W} + 1 \quad \text{and} \quad H_2 = \frac{H_1 + 2P_W - K_H}{S_H} + 1$$



Pooling

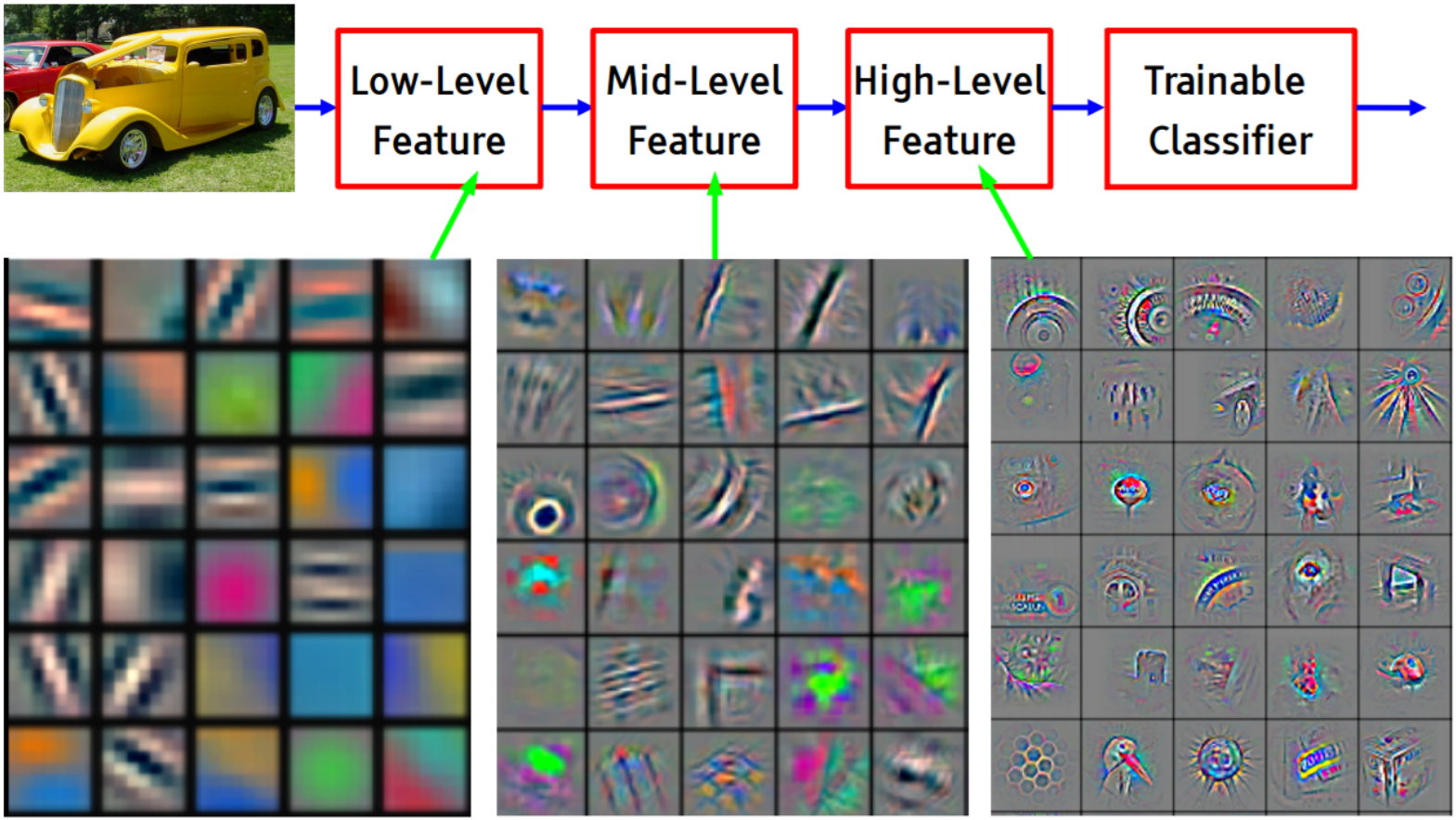
Max pooling

- Filter size: (2,2)
- Stride: (2,2)
- Pooling ops: max(·)



Visualize features

- Give insight into the function of intermediate feature layers and the operation of the classifier



(Zeiler and Fergus, 2014)

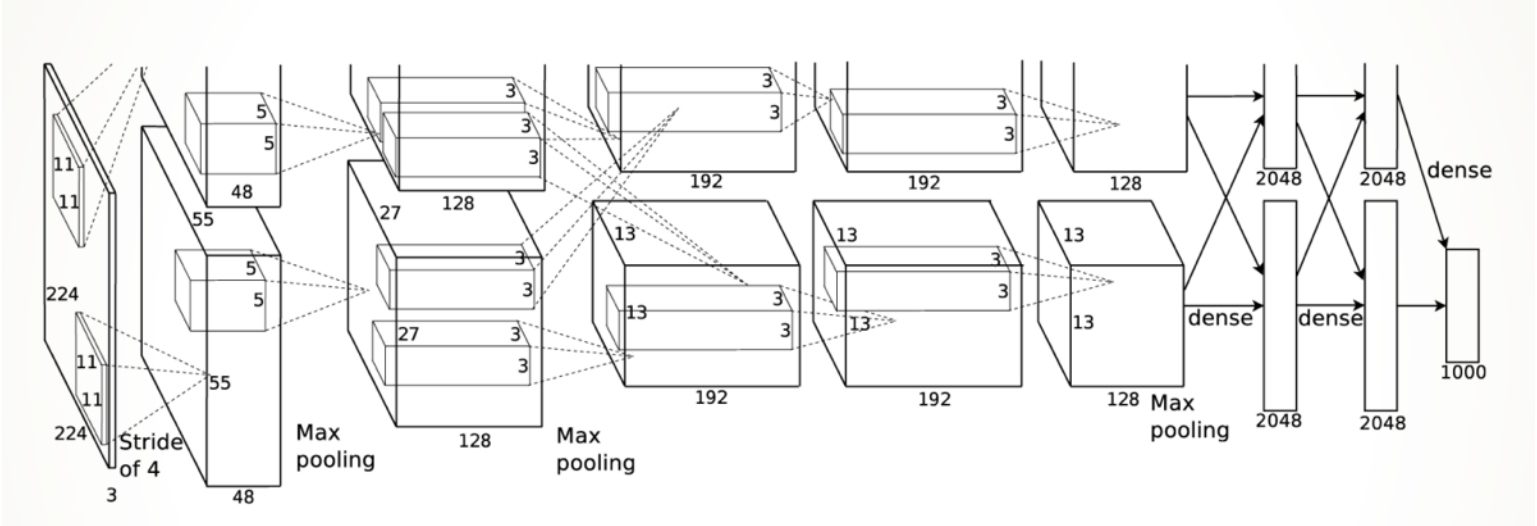
Network Structures

Winning CNN Architectures

Model	AlexNet	ZF Net	GoogLeNet	Resnet
Year	2012	2013	2014	2015
#Layer	8	8	22	152
Top 5 Acc	15.4%	11.2%	6.7%	3.57%
Data augmentation	√	√	√	√
Dropout	√	√		
Batch normalization				√

CNN Architecture: Part I

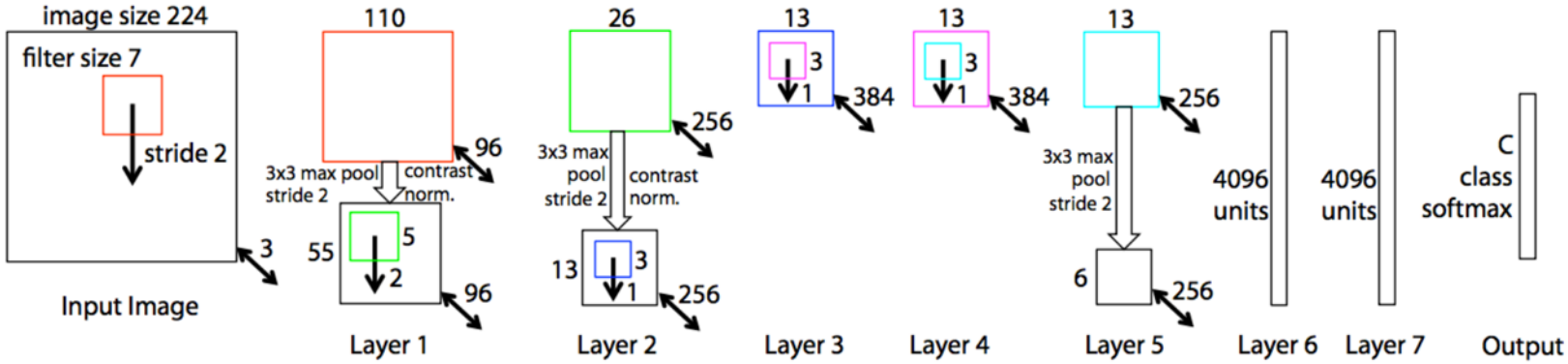
- **AlexNet** [Krizhevsky et al. 2012] -5 conv layers, 3 fully connected layers.



- Activation function: ReLU
- Data Augmentation
- Dropout (drop rate=0.5)
- Local Response Normalization
- Overlapping Pooling

CNN Architecture: Part I

❑ ZFNet [Zeiler and Fergus, 2013]



- An improved version of AlexNet: top-5 error from 16.4% to 11.7%
- First convolutional layer: 11x11 filter, stride=4 -> 7x7 filter, stride=2
- The number of filters increase as we go deeper.

CNN Architecture: Part I

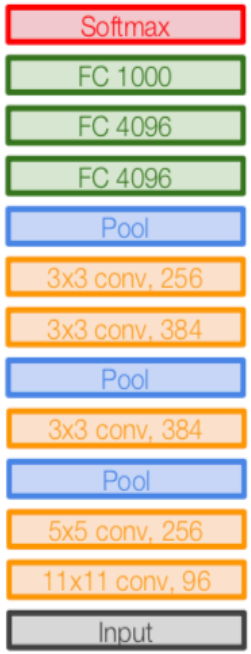
❑ VGGNet [Simonyan and Zisserman, 2014]

Small filters: Why?

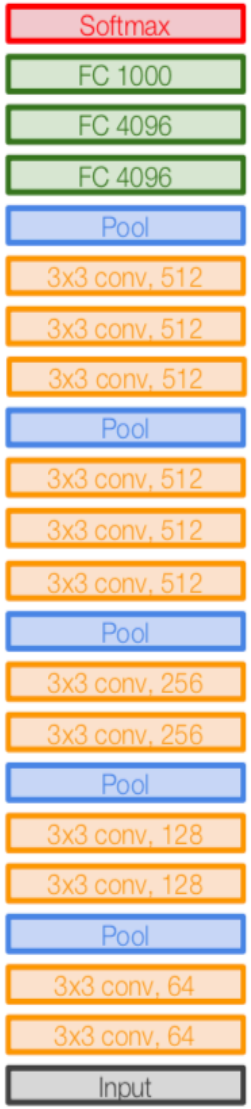
- 3x3 convolutional layers (stride 1, pad 1)
- 2x2 max-pooling, stride 2

Deeper networks:

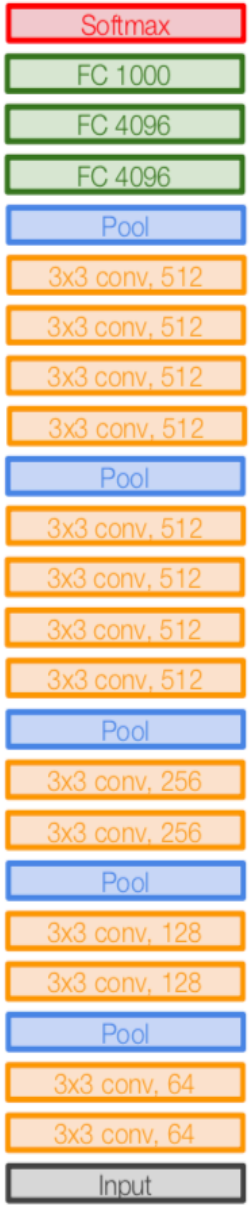
- AlexNet: 8 layers
- VGGNet: 16 or 19 layers



AlexNet



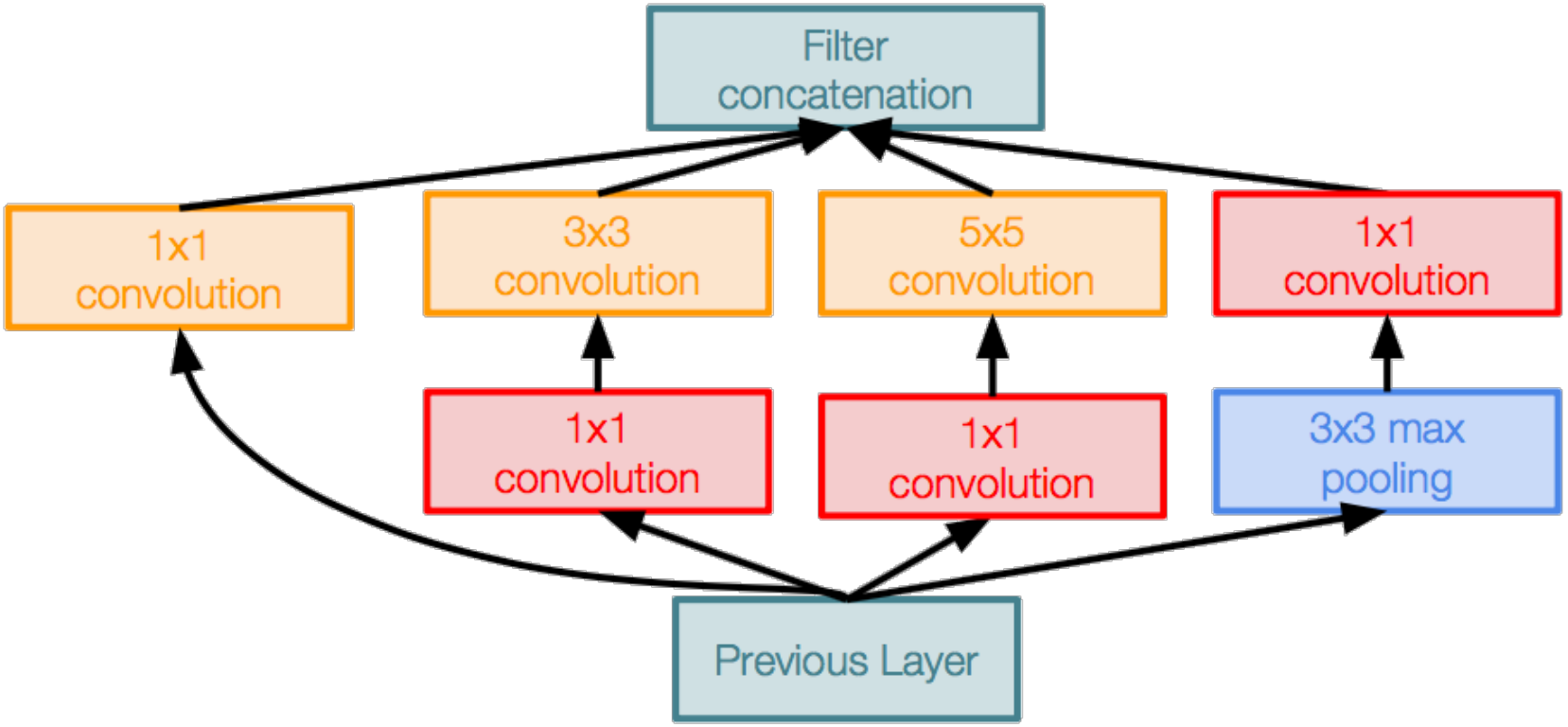
VGG16



VGG19

CNN Architecture: Part I

- **GoogLeNet** [Szegedy et al., 2014]



Inception module

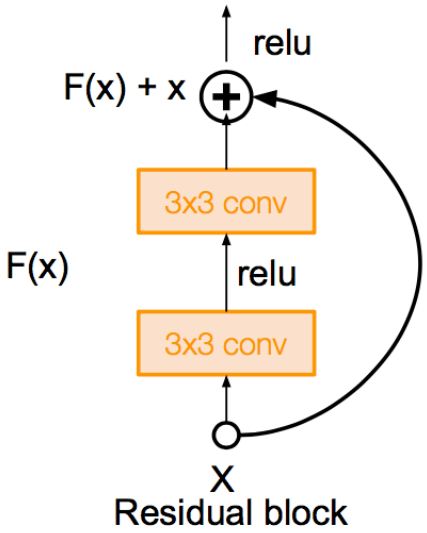
CNN Architecture: Part I

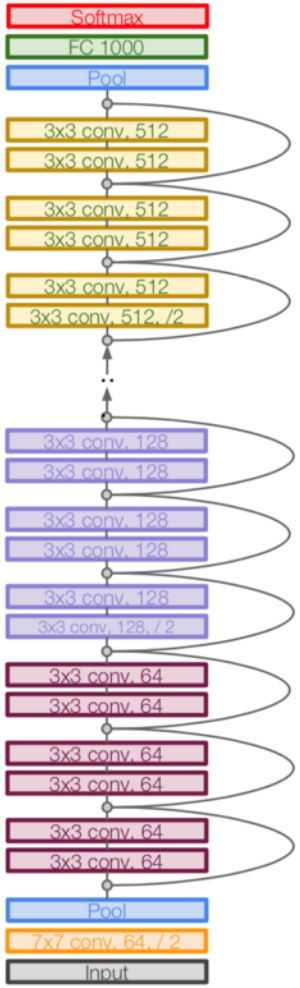
❑ ResNet [He et al., 2015]

Very deep networks using residual connections

Basic design:

- all 3x3 conv (almost)
- spatial size /2 => #filter x2
- just deep
- average pooling (remove FC)
- no dropout



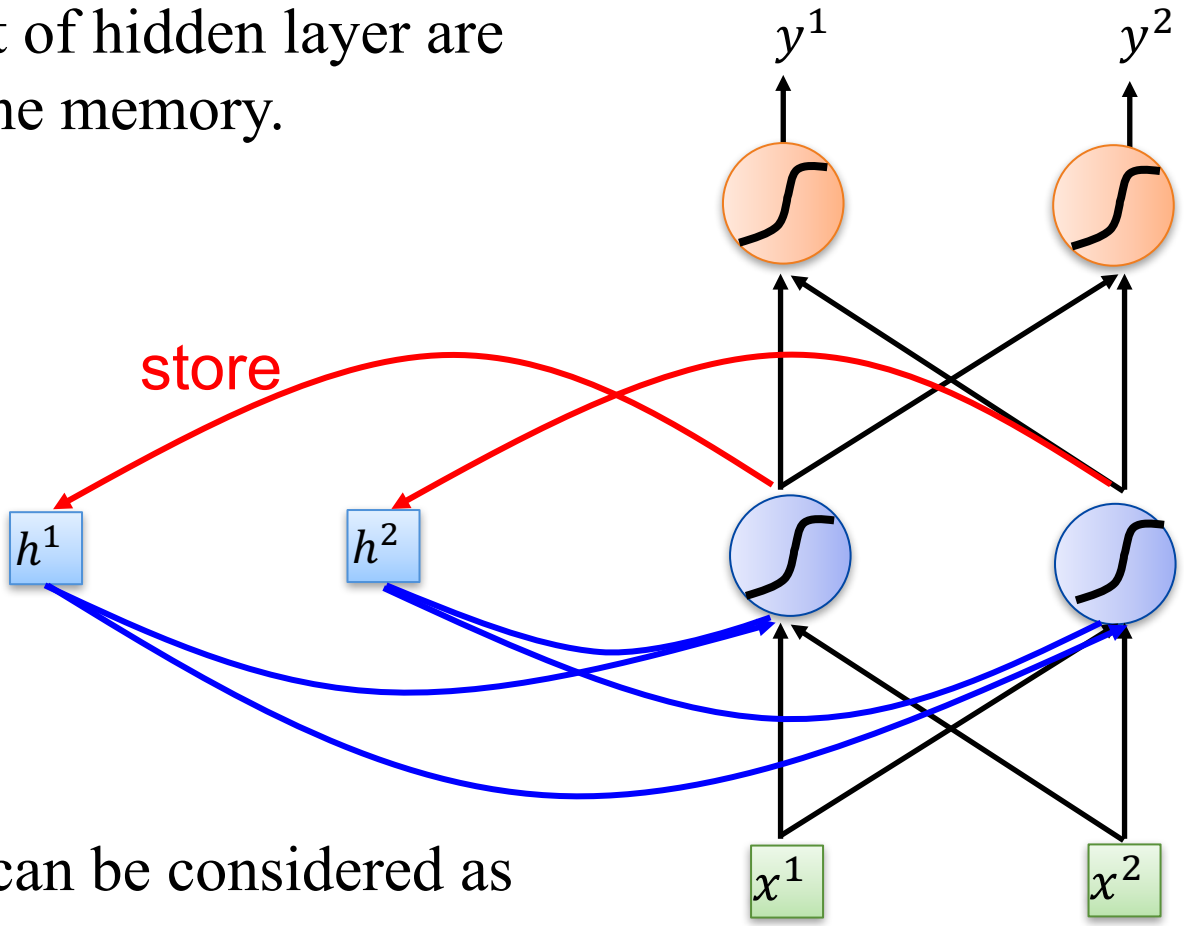


ResNet

Recurrent Neural Network and LSTM

Recurrent Neural Network

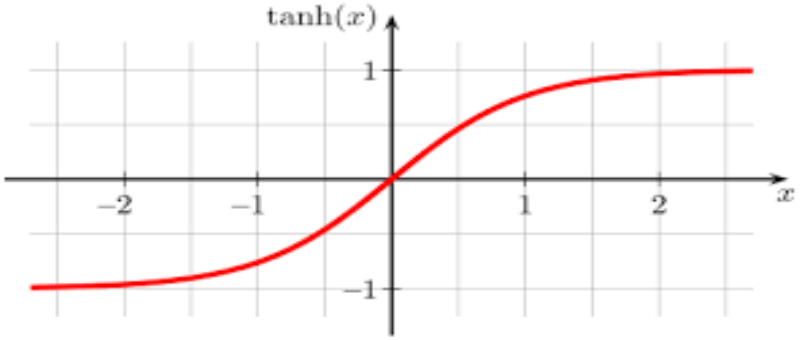
The output of hidden layer are stored in the memory.

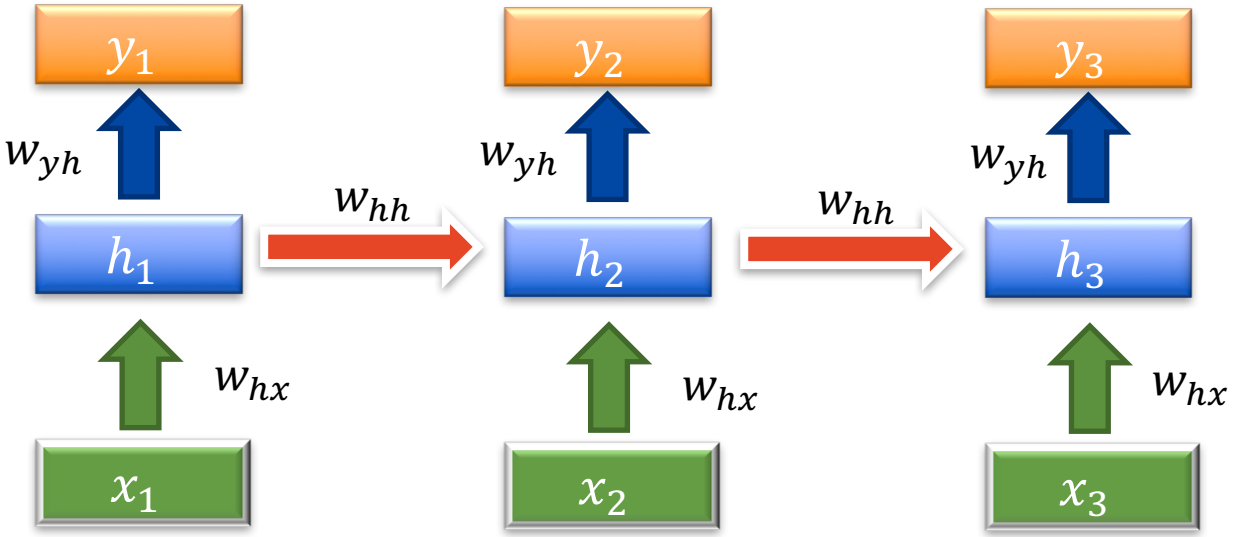


Memory can be considered as another input.

Recurrent Neural Network

- Formally
  - x_t is the input
  - h_t is the hidden state
  - y_t is the output
  - $h_0 = \vec{0}$
  - $h_t = \tanh(w_{hh}h_{t-1} + w_{hx}x_t + b_h)$
  - $y_t = w_{yh}h_t$





Recurrent Neural Network

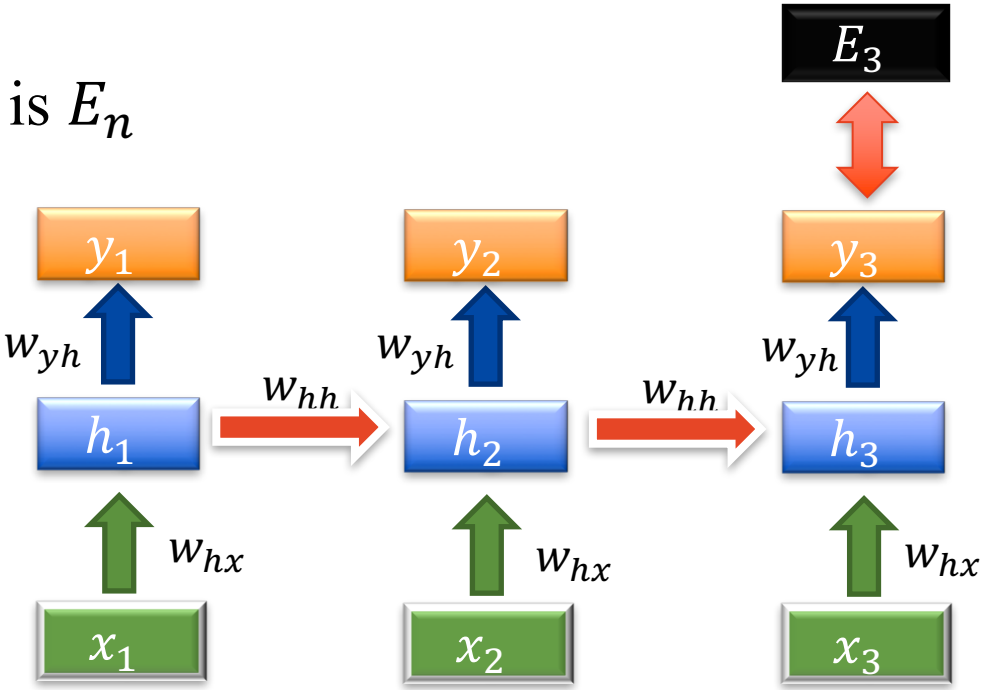
- Vanishing gradient problem
  - $h_t = \tanh(w_{hh}h_{t-1} + w_{hx}x_t + b_h)$
  - $y_t = w_{yh}h_t$
  - For every step, its loss is E_n

$$\frac{\partial E_3}{\partial w_{hh}} = \frac{\partial E_3}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial w_{hh}}$$

$$h_3 = \tanh(w_{hh}h_2 + w_{hx}x_3 + b_h)$$

$$\frac{\partial h_3}{\partial w_{hh}} = (\tanh)'\left(h_2 + w_{hh}\frac{\partial h_2}{\partial w_{hh}}\right)$$

$$\frac{\partial h_2}{\partial w_{hh}} = (\tanh)'\left(h_1 + w_{hh}\frac{\partial h_1}{\partial w_{hh}}\right)$$



$(\tanh)'w_{hh}\ (\tanh)'w_{hh}\ (\tanh)'w_{hh}\ \cdots$

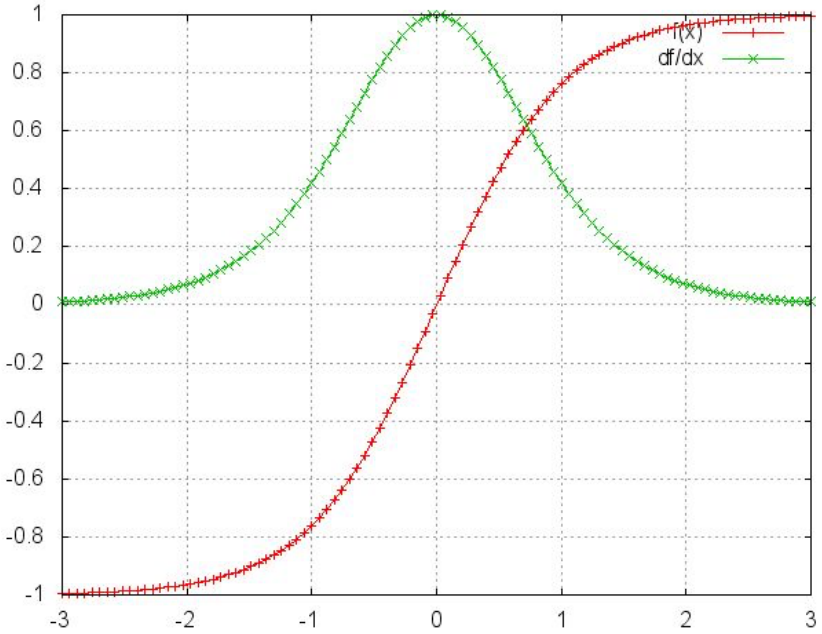
Recurrent Neural Network

- Vanishing gradient problem
  - $h_t = \tanh(w_{hh}h_{t-1} + w_{hx}x_t + b_h)$
  - $y_t = w_{yh}h_t$
  - For every step, its loss is E_n

$$0.9^{1000} \approx 0$$
$$1.01^{1000} \approx 21000$$

$$\frac{\partial E_3}{\partial w_{hh}} = \frac{\partial E_3}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial w_{hh}}$$

$$(\tanh)' w_{hh} (\tanh)' w_{hh} (\tanh)' w_{hh} \cdots$$



$(\tanh)' \leq 1$

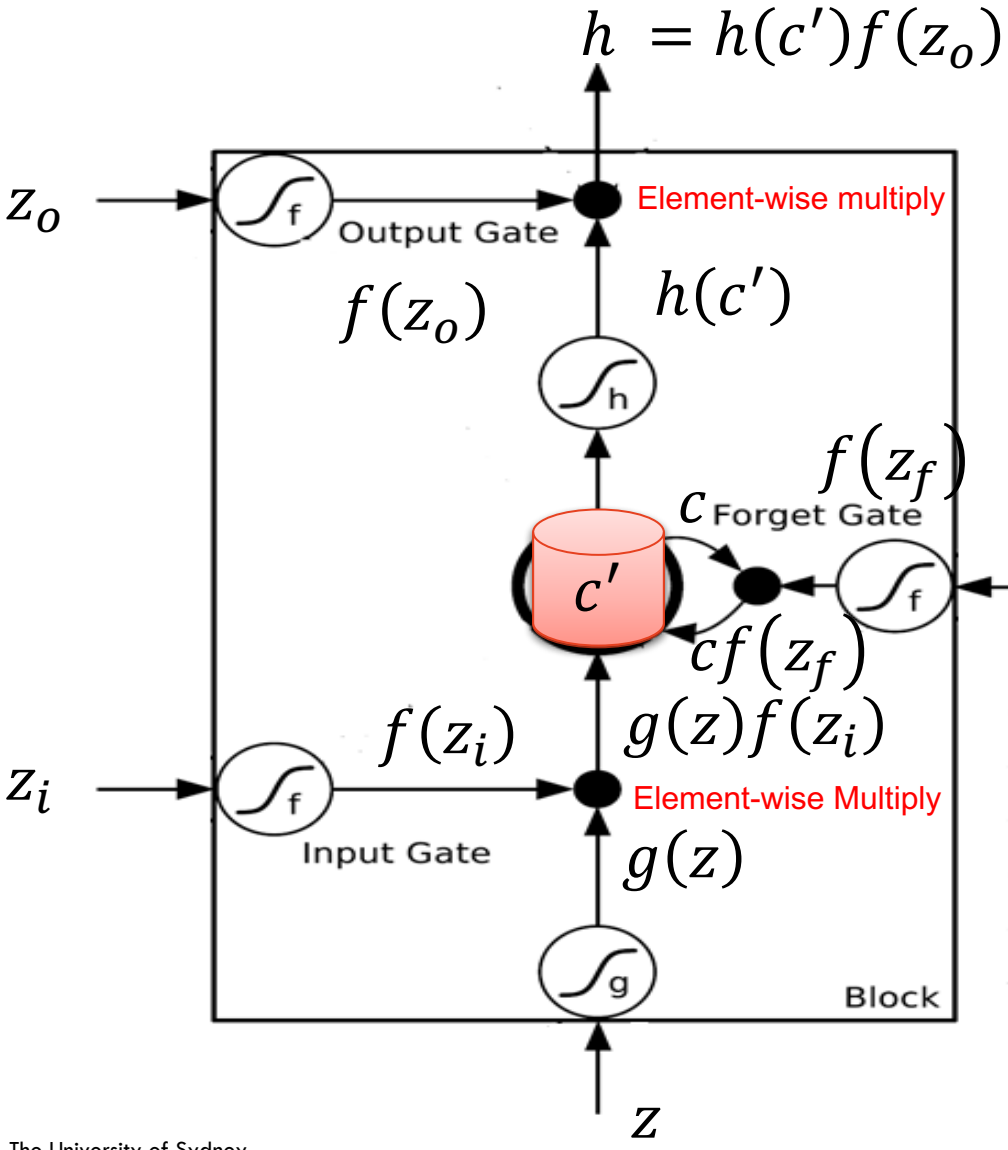
If $0 < w_{hh} < 1$, Vanishing Gradients

$(\tanh)' w_{hh} (\tanh)' w_{hh} (\tanh)' w_{hh} \cdots \to 0$

If w_{hh} is larger, Exploding Gradients

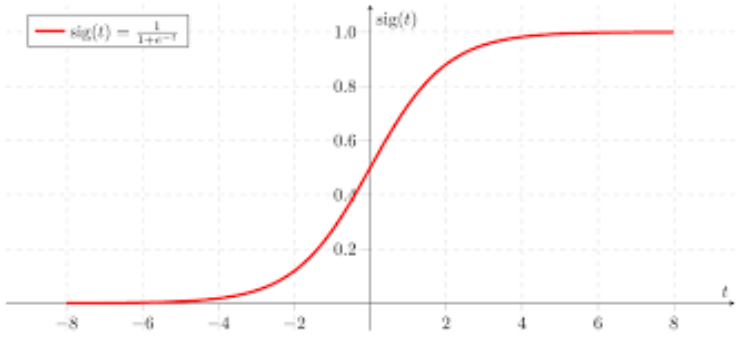
$(\tanh)' w_{hh} (\tanh)' w_{hh} (\tanh)' w_{hh} \cdots \to \infty$

Long Short-Term Memory



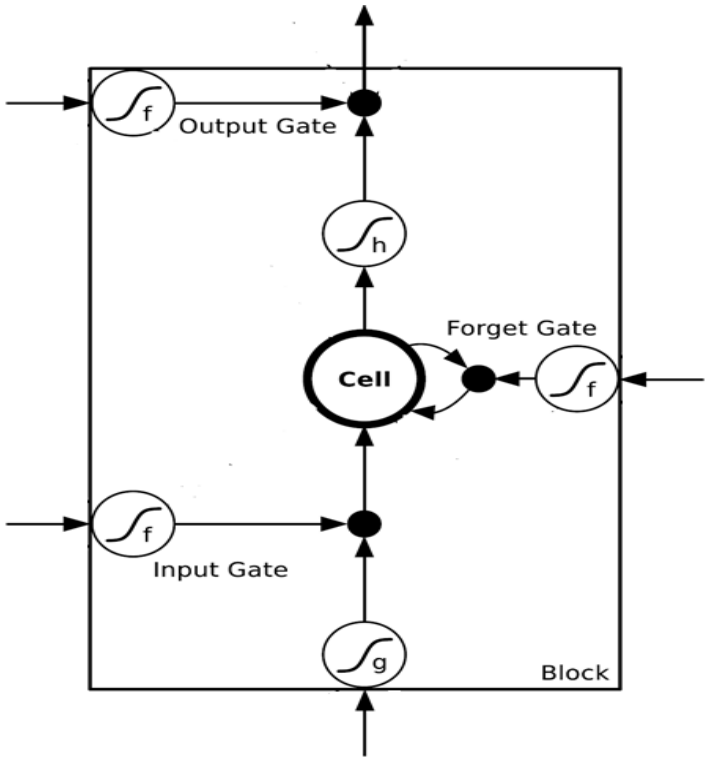
f is usually a sigmoid function
Value From 0 to 1
Mimic open and close gate

$$c' = g(z)f(z_i) + cf(z_f)$$



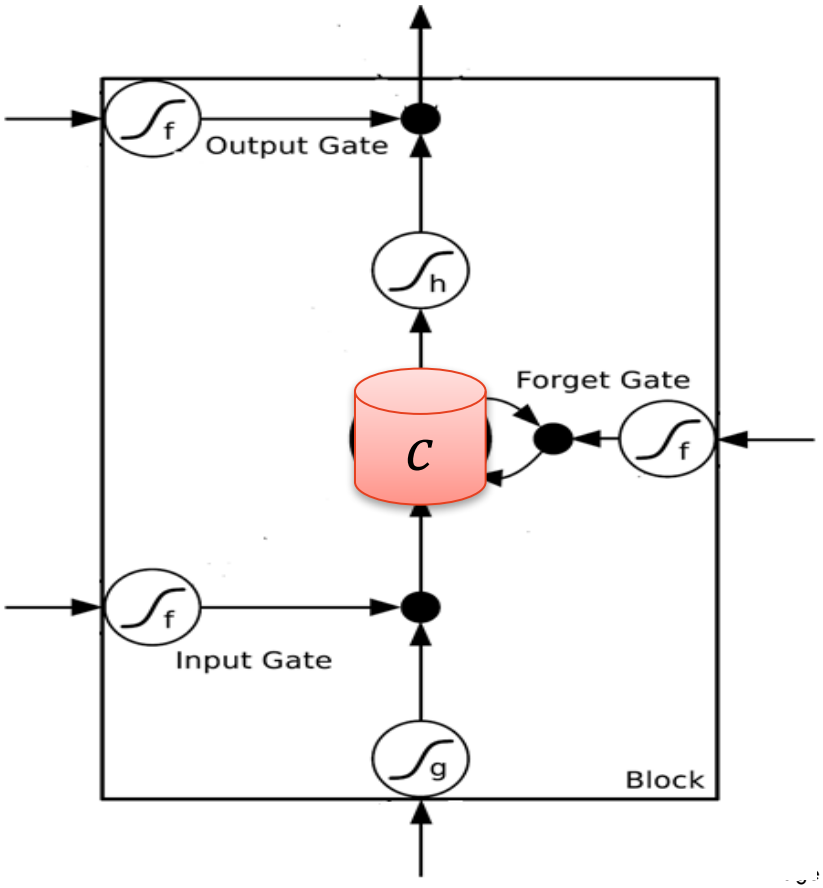
Long Short-Term Memory

- Forget gate
  - $f_t = \sigma(w_{fh}h_{t-1} + w_{fx}x_t + b_f)$
- Input gate
  - $i_t = \sigma(w_{ih}h_{t-1} + w_{ix}x_t + b_i)$
- Output gate
  - $o_t = \sigma(w_{oh}h_{t-1} + w_{ox}x_t + b_o)$



Long Short-Term Memory

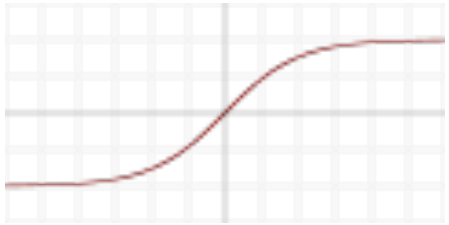
- Candidate cell state
  - $\tilde{c}_t = \tanh(w_{hh}h_{t-1} + w_{hx}x_t + b_h)$
- Update cell state
  - $c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$
- Output state
  - $h_t = o_t * \tanh(c_t)$



Prevent vanishing gradient

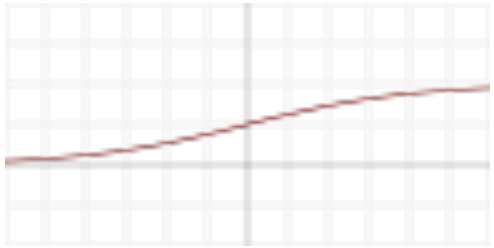
- RNN
  - $h_t = \tanh(w_{hh} h_{t-1} + w_{hx} x_t + b_h)$
- LSTM
  - $c_t = f_t * c_{t-1} + i_t * \widetilde{c}_t$

Back propagation from c_t to c_{t-1} only involves elementwise multiplication with f_t, no matrix multiplication with w_{hh} or tanh.



$f(x) = tanh(x)$

$f'(x) = 1 - f(x) \leq 1$

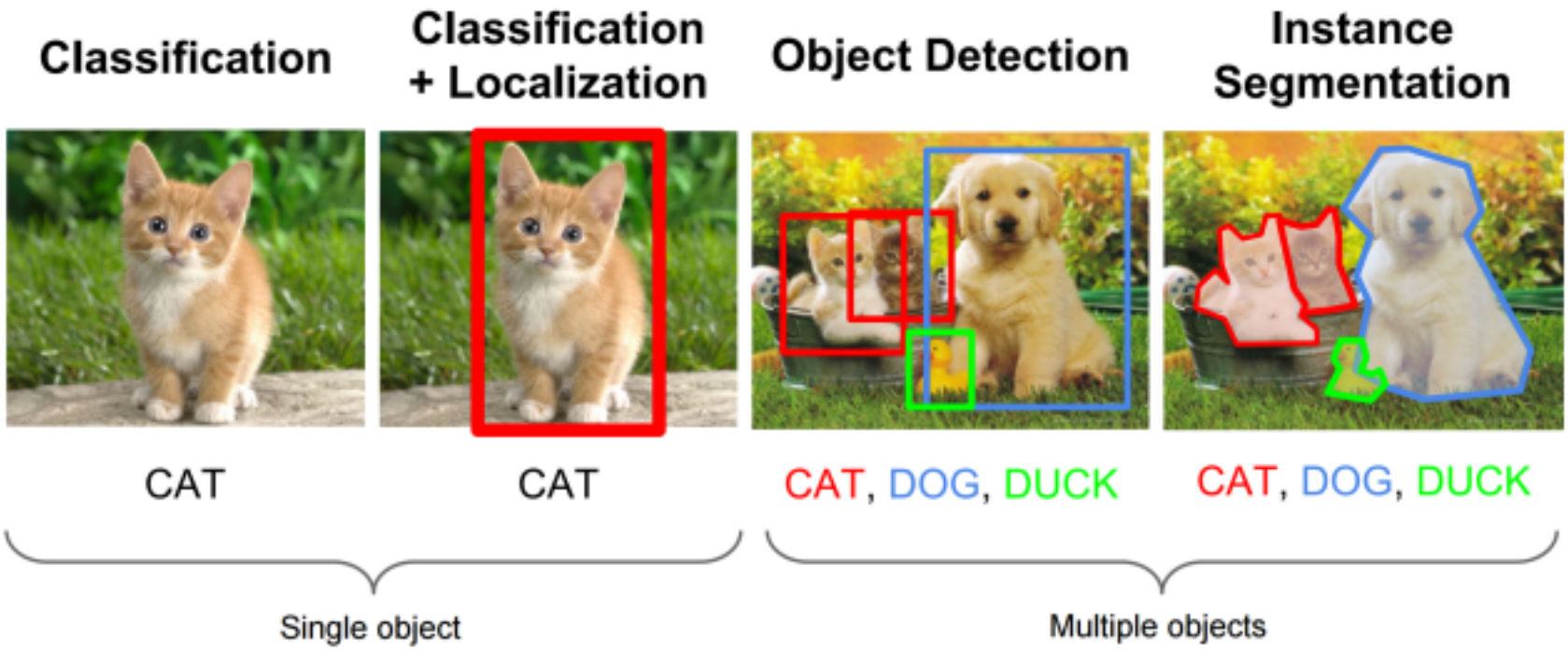


$f(x) = \sigma(x)$

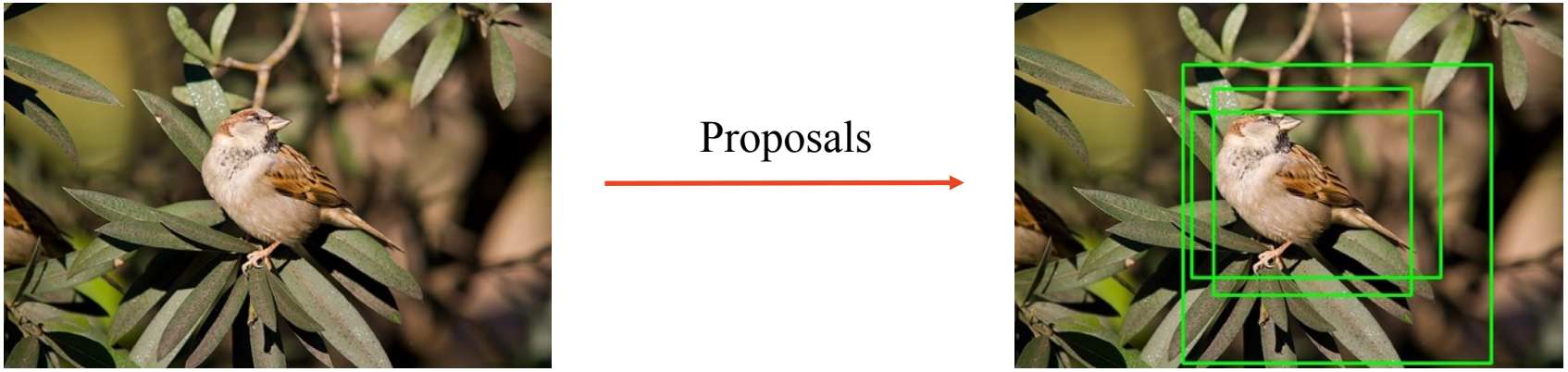
$f'(x) = f(x)\big(1 - f(x)\big).$
≤ 0.25

Deep Learning Applications

Computer Vision Tasks



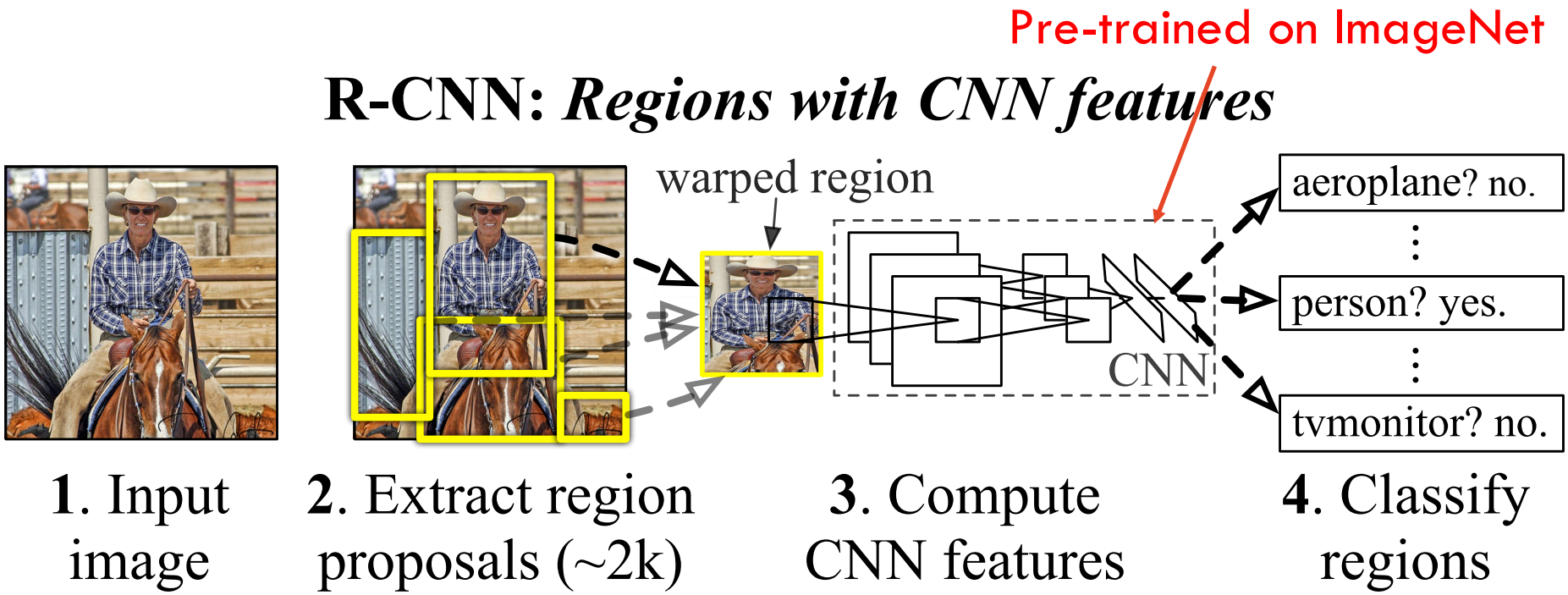
Typical architecture



1.Region proposal: Given an input image find all possible places where objects can be located. The output of this stage should be a list of bounding boxes of likely positions of objects. These are often called region proposals or regions of interest.

2.Final classification: for every region proposal from the previous stage, decide whether it belongs to one of the target classes or to the background. Here we could use a deep convolutional network.

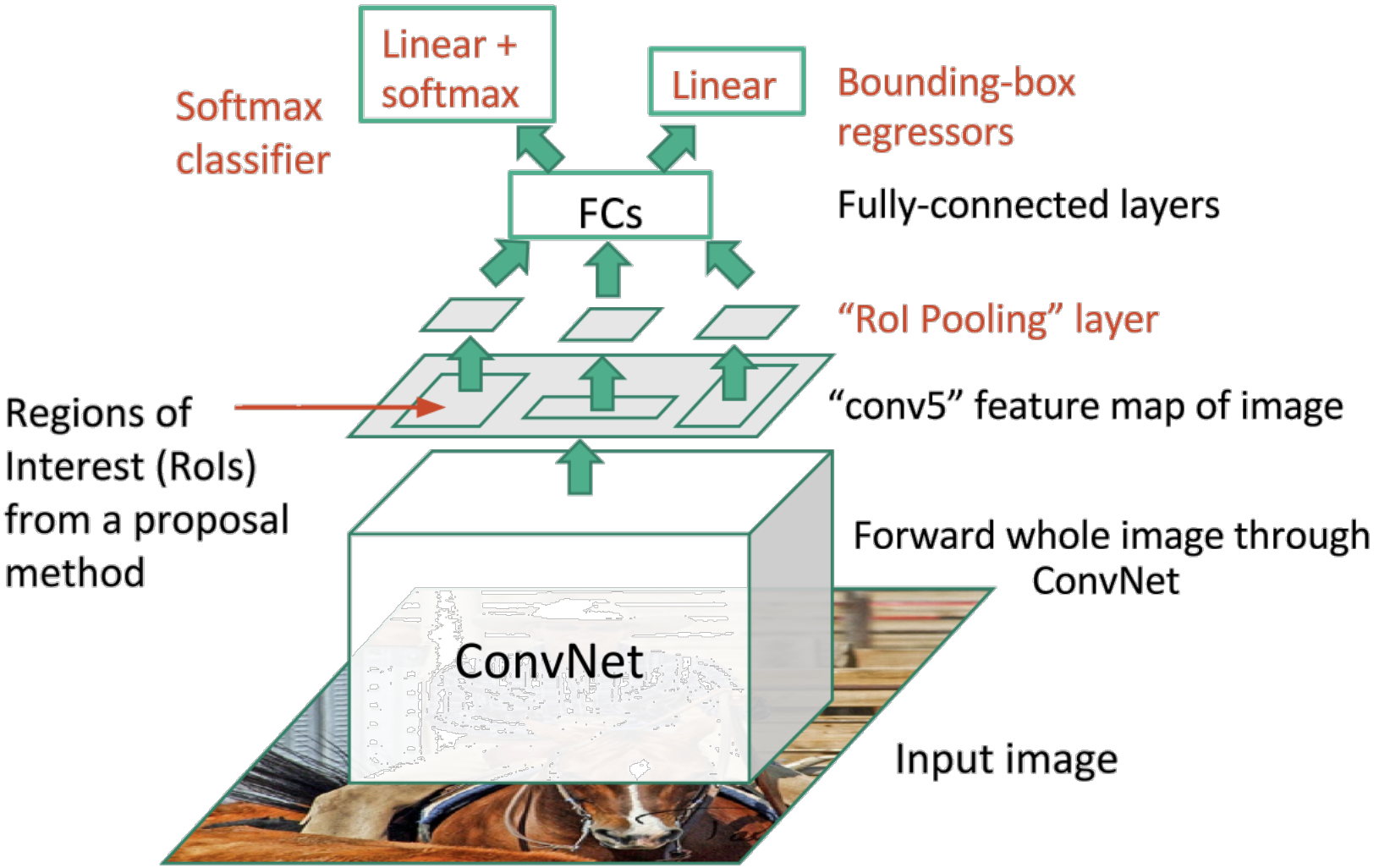
Region CNN (R-CNN)



Object Detection to Image Classification

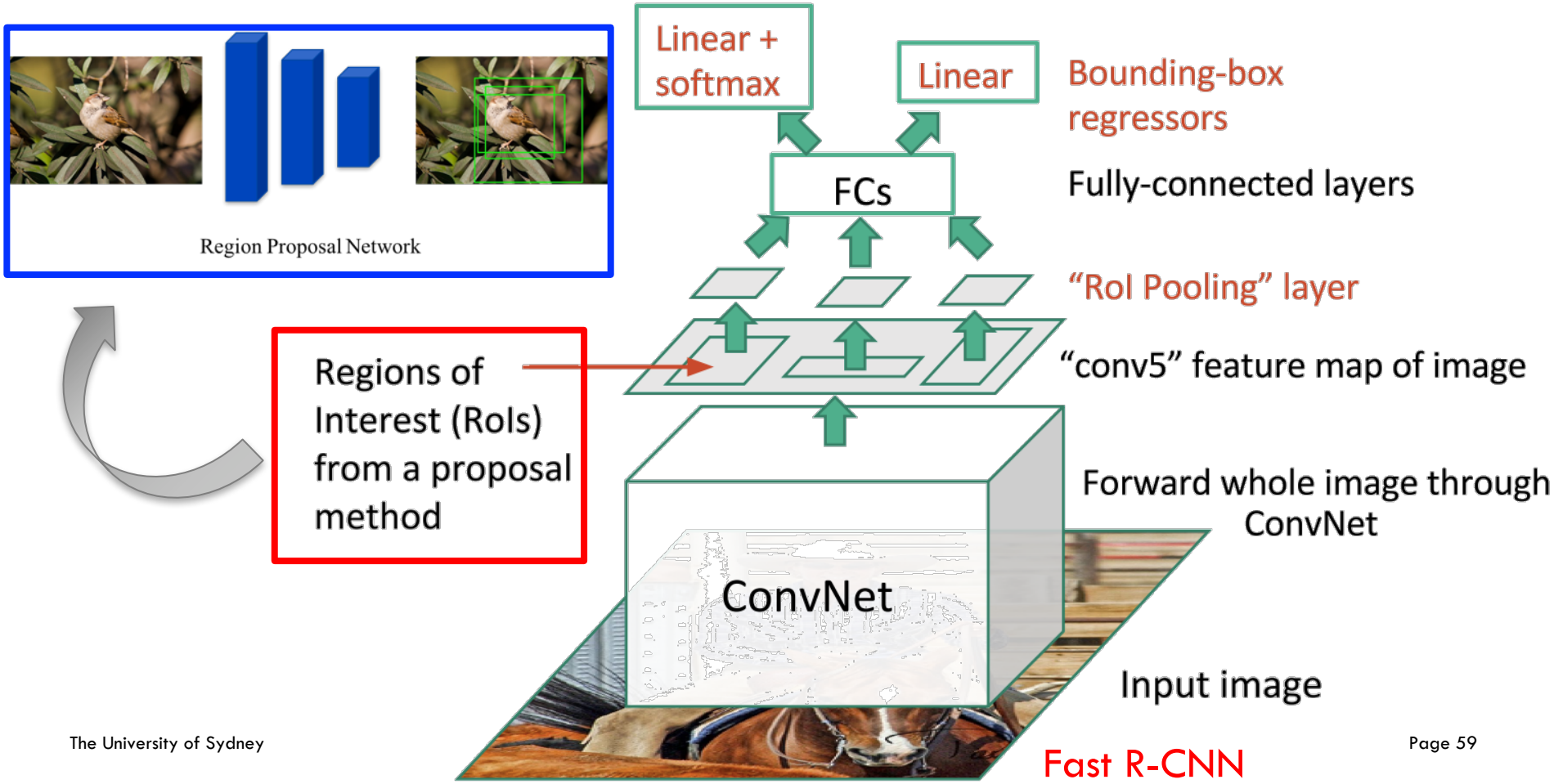
[Girshick14] R. Girshick, J. Donahue, S. Guadarrama, T. Darrell, J. Malik: **Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation**, CVPR 2014

Fast R-CNN


Linear + softmax
Linear
Softmax classifier
Bounding-box regressors
FCs
Fully-connected layers
"RoI Pooling" layer
Regions of Interest (RoIs) from a proposal method
"conv5" feature map of image
Forward whole image through ConvNet
ConvNet
Input image

Faster R-CNN

Replace the slow selective search algorithm with a fast neural net - region proposal network (RPN).



Region Proposal Network

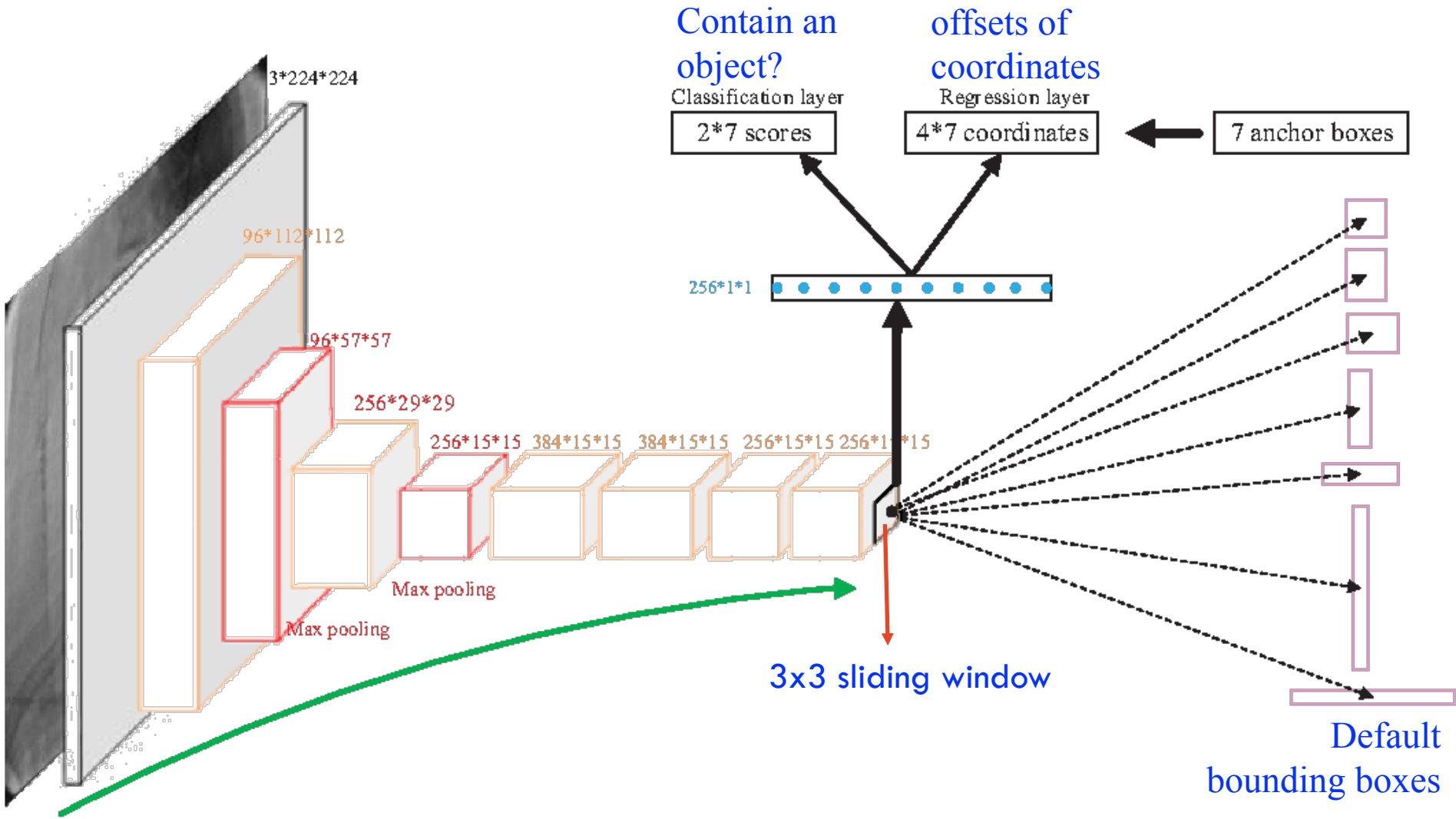


Image Credit to [Li Wang et al. 2017]

RoIAlign

Realigning RoI Pooling to be More Accurate.

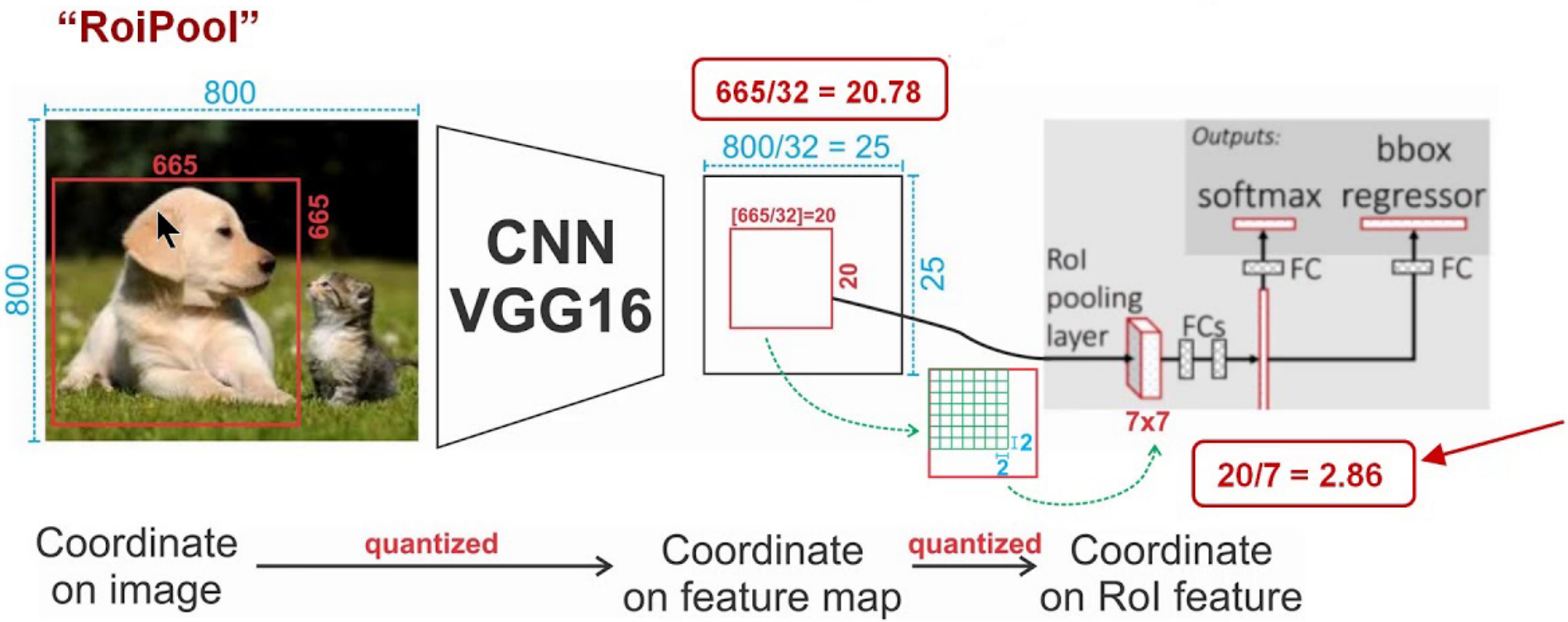


Image credit to Ardian Umam

Mask R-CNN

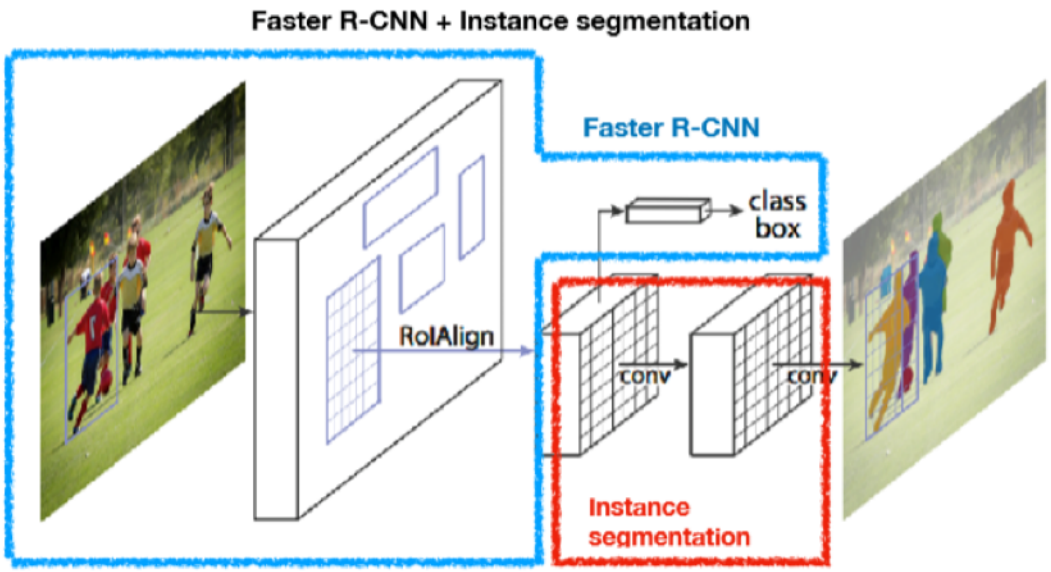
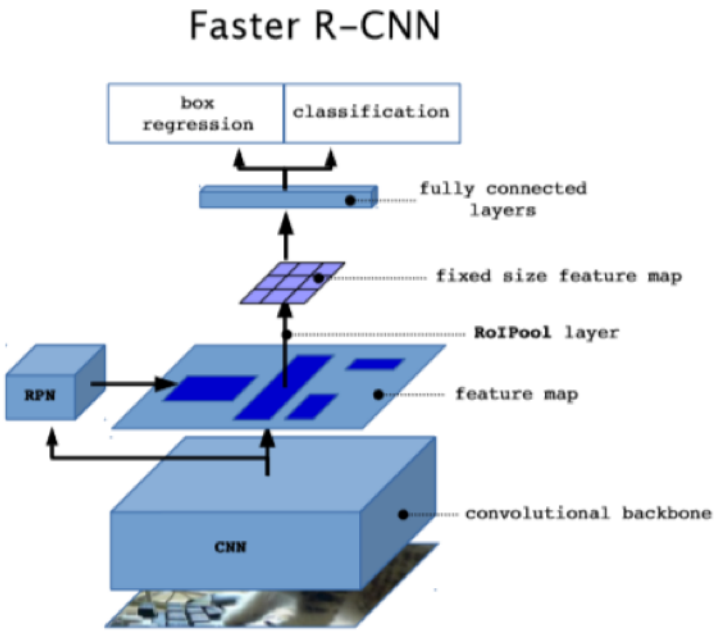


Figure 1. The **Mask R-CNN** framework for instance segmentation.



Mask R-CNN
= Instance Segmentation+ Faster R-CNN.

Credit To: https://www.slideshare.net/windmdk/mask-rcnn

Deep Generative Models

Generative Modeling

- Density Estimation

- Sample Generation

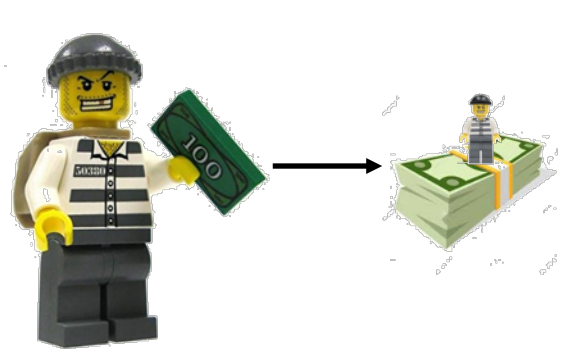
Training Sample

Generated Sample

Image credit to [Ian Goodfellow, NIPS Tutorial on Generative Model 2016]

Generative Adversarial Networks

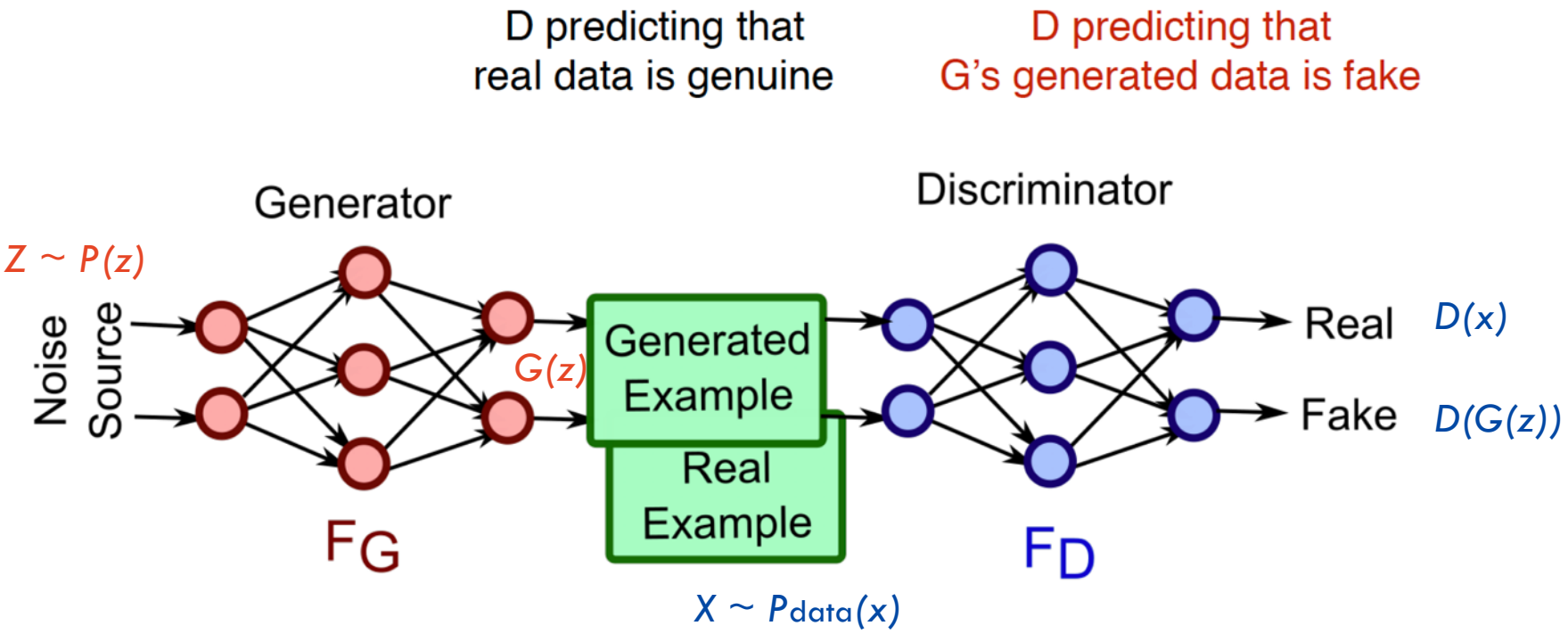
- A counterfeiter-police game between two components: a generator **G** and a discriminator **D**
- **G**: counterfeiter, trying to fool police with fake currency
- **D**: policy, trying to detect the counterfeit currency
- Competition drives both to improve, until counterfeits are *indistinguishable* from genuine currency



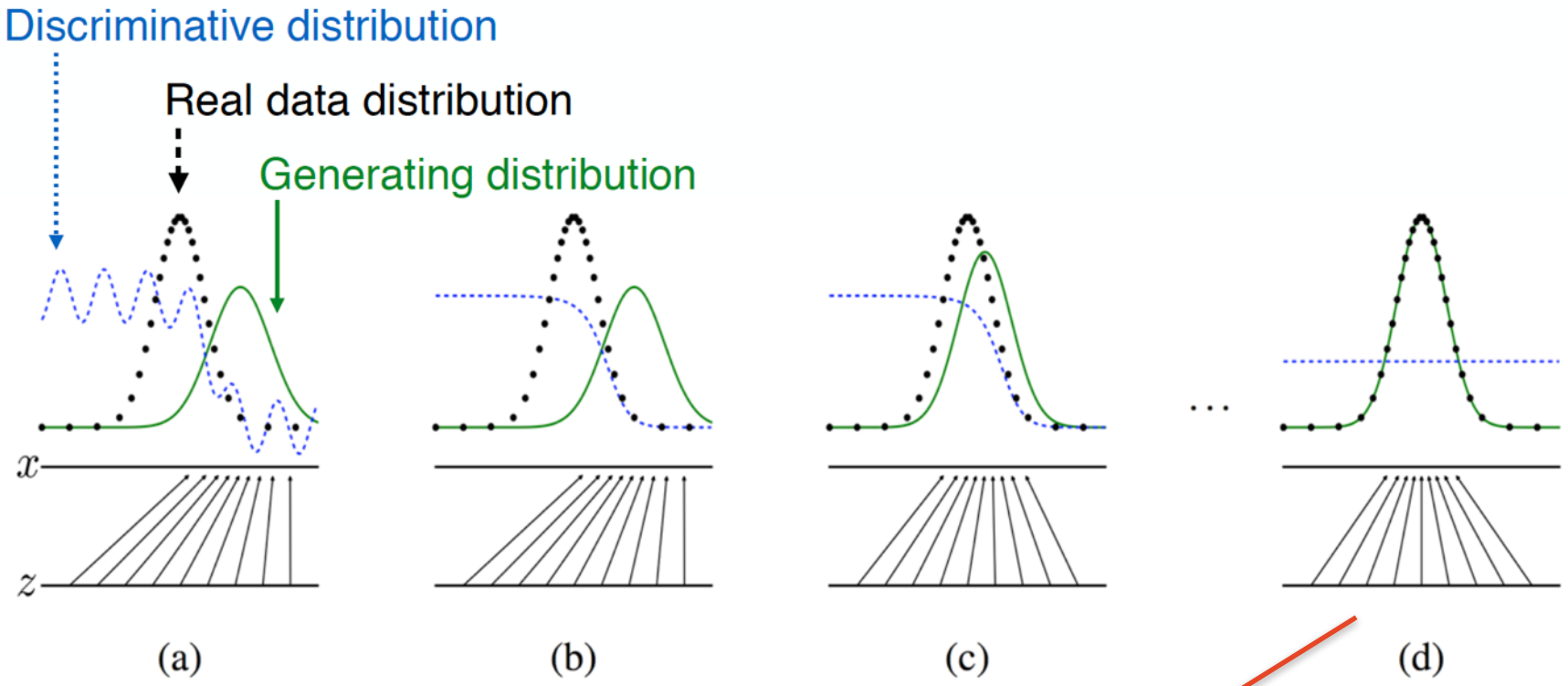
Generative Adversarial Networks

- A two-player game between two components: a generator **G** and a discriminator **D**

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$



A simple example



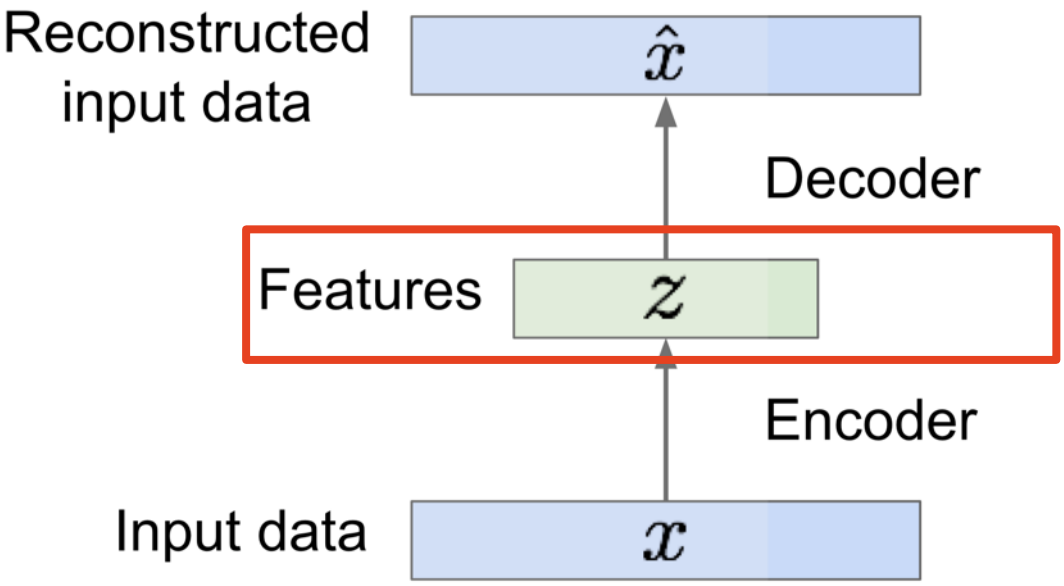
After several steps of training, if G and D have enough capacity, they will reach a point at which both cannot improve because $p_g = p_{data}$. The discriminator is unable to differentiate between the two distributions, i.e. **D(x) = 1/2** .

Image credit to [Ian Goodfellow et al., Generative Adversarial Nets, NIPS 2016]

Recap: Autoencoder

Autoencoders can reconstruct data, and can learn features to initialize a supervised model. Features capture factors of variation in training data.

Can we generate new images from an autoencoder?

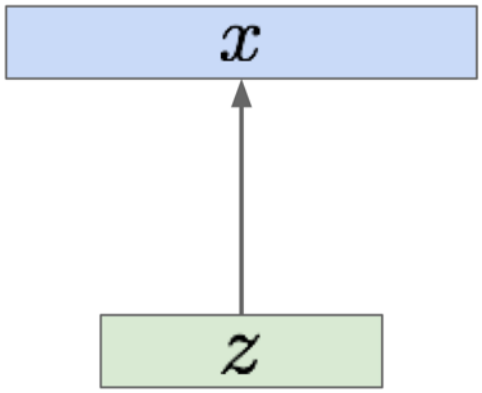


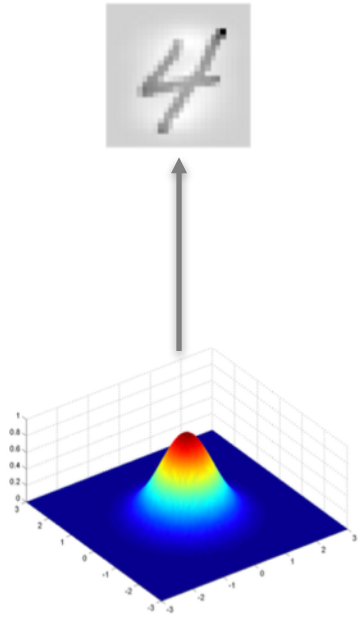
Variational Autoencoders

- Probabilistic spin on autoencoders - will let us sample from the model to generate data!
- Assume training data $\{x^{(i)}\}_{i=1}^{N}$ is generated from underlying unobserved (latent) representation **z**

Sample from true conditional $p_{\vartheta^*}(x \mid z^{(i)})$

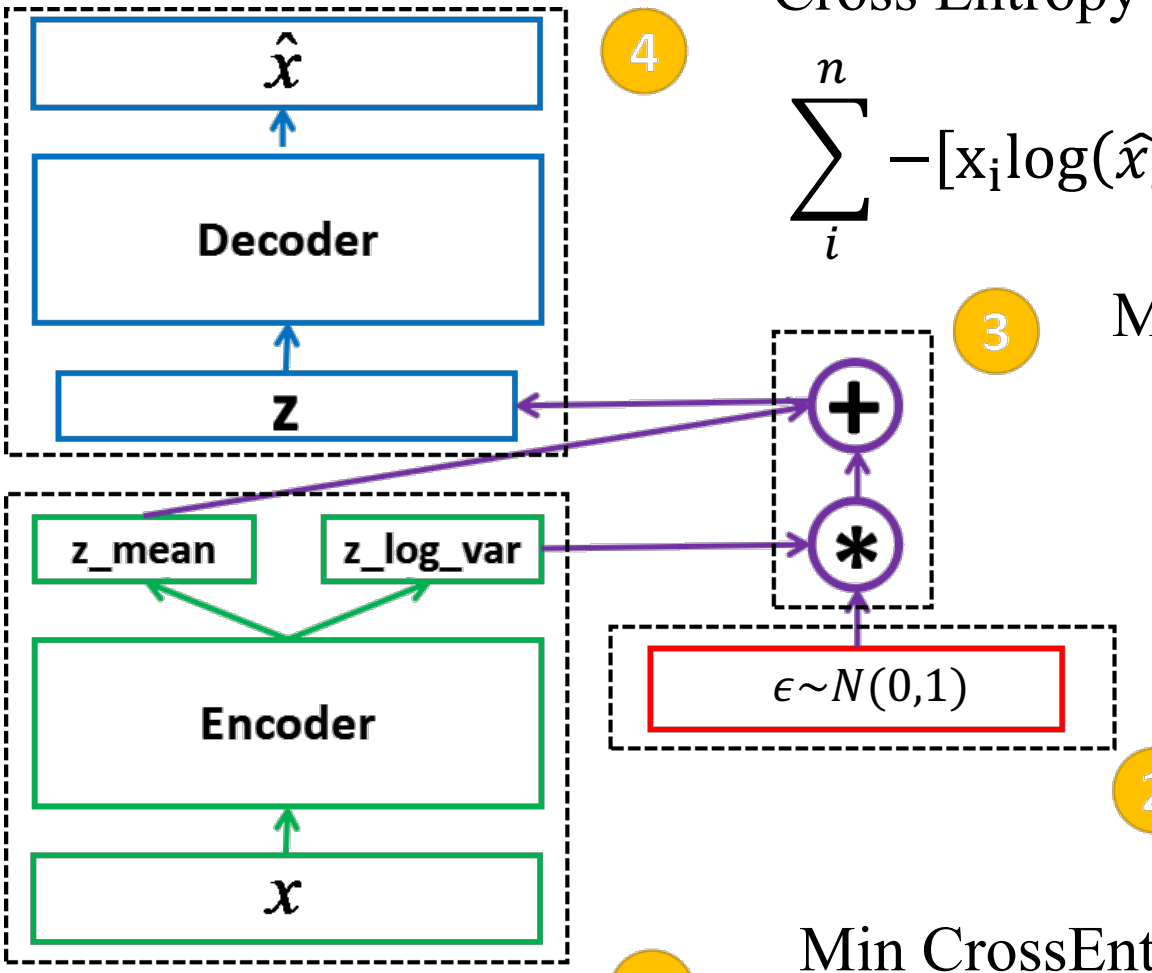
Sample from true prior $p_{\vartheta^*}(z)$





Choose prior p(z) to be simple, e.g. Gaussian. Reasonable for latent attributes, e.g. pose.

Variational Autoencoders



Cross Entropy

$$\sum_{i}^{n} -[x_i \log(\hat{x}_i) + (1 - x_i)\log(1 - \hat{x}_i)]$$

MSE $$\sum_{i}^{n} (x_i - \hat{x}_i)^2$$

KL Divergence

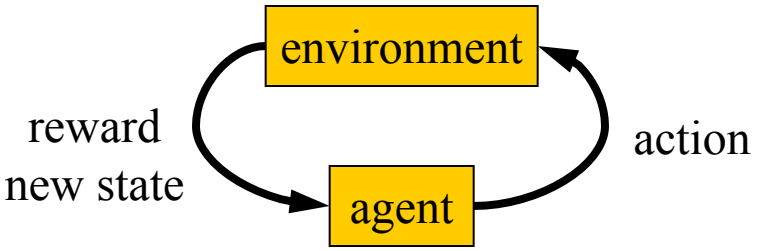
$$-0.5(1 + \log \sigma^2 - \mu^2 - \sigma^2)$$

Min CrossEntropy + KLDivergence

Deep Reinforcement Learning

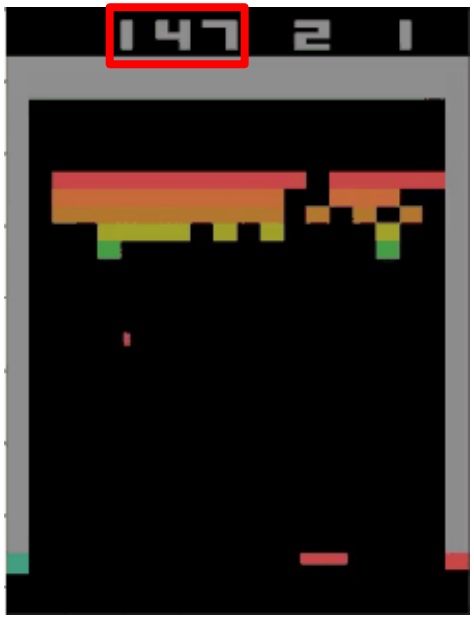
Markov Decision Process (MDP)

- set of states S, set of actions A, initial state S_0
- transition model P(s,a,s')
  - P($frame_{(t)}$, right, $frame_{(t')}$) = 0.8
- reward function r(s)
  - r($frame_{(t)}$) = +1
- goal: maximize cumulative reward in the long run



- policy: mapping from S to A
  - a=π(s) or π(s, a) (deterministic vs. stochastic)

- reinforcement learning
  - transitions and rewards usually not available
  - how to change the policy based on experience
  - how to explore the environment



Computing return from rewards

- episodic (vs. continuing) tasks
  - "game over" after N steps
  - optimal policy depends on N; harder to analyze

- additive rewards
  - $V(s_0, s_1, \ldots) = r(s_0) + r(s_1) + r(s_2) + \ldots$
  - infinite value for continuing tasks

- discounted rewards
  - $V(s_0, s_1, \ldots) = r(s_0) + \gamma * r(s_1) + \gamma^2 * r(s_2) + \ldots$
  - value bounded if rewards bounded

The goal of RL is to find the policy which maximizes the expected return.

Value Function

- A value function is the prediction of the future return
- Two definitions exist for the value function
  - State value function
  - "How much reward will I get from state s?"
  - expected return when starting in s and following π

$$V^{\pi}(s) = \mathbb{E}\left\{ \sum_{k=0}^{\infty} \gamma^k r_k \middle| s_0 = s, \pi \right\}$$

$$V^{\pi}(s) = \mathbb{E}\left\{ Q^{\pi}(s,a) \middle| a \sim \pi(s,\cdot) \right\}$$

  - State-action value function
  - "How much reward will I get from action a in state s?"
  - expected return when starting in s, performing a, and following π

$$Q^{\pi}(s,a) = \mathbb{E}\left\{ \sum_{k=0}^{\infty} \gamma^k r_k \middle| s_0 = s, a_0 = a, \pi \right\}$$

Bellman Equation and Optimality

- Value functions decompose into Bellman equations, i.e. the value functions can be decomposed into

  immediate reward plus *discounted value of successor state*

$$V^{\pi}(s) = \mathbb{E}\left\{ \sum_{k=0}^{\infty} \gamma^k r_k \,\middle|\, s_0 = s, \pi \right\} \longrightarrow V^{\pi}(s) = \mathbb{E}\left\{ R(s,a,s') + \gamma V^{\pi}(s') \right\}$$

$$Q^{\pi}(s,a) = \mathbb{E}\left\{ \sum_{k=0}^{\infty} \gamma^k r_k \,\middle|\, s_0 = s, a_0 = a, \pi \right\}$$

$$Q^{\pi}(s,a) = \mathbb{E}\left\{ R(s,a,s') + \gamma Q^{\pi}(s',a') \right\}$$

- An optimal value function is the maximum achievable value.

$$V^*(s) = \max_{\pi} V^{\pi}(s) \quad Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a)$$

Bellman Optimality Equation

- Optimality for value functions is governed by the Bellman optimality equations.

- Two equations:

$$V^*(s) = \max_a \mathbb{E}\left\{R(s,a,s') + \gamma V^*(s')\right\}$$

$$Q^*(s,a) = \mathbb{E}\left\{R(s,a,s') + \max_{a'} \gamma Q^*(s',a')\right\}$$

$$\pi^*(s) = \arg\max_a Q^*(s,a)$$

Q-Learning

$$Q^*(s,a) = \mathbb{E}\left\{ R(s,a,s') + \max_{a'} \gamma Q^*(s',a') \right\}$$

Initialize $Q(s,a)$ arbitrarily.

Start with s.

Before taking action a, we calculate the current expected return as

$$Q(s,a)$$

After taking action a, and observing r and s', we calculate the target expected return as

$$R(s,a,s') + \max_{a'} \gamma Q(s',a')$$

$$\Delta Q(s,a) = R(s,a,s') + \max_{a'} \gamma Q(s',a') - Q(s,a)$$

$$Q(s,a) \leftarrow Q(s,a) + \alpha \Delta Q(s,a)$$

Q-Learning

$$Q^*(s,a) = \mathbb{E}\left\{ R(s,a,s') + \max_{a'} \gamma Q^*(s',a') \right\}$$

Initialize Q(s, a) arbitrarily.

Repeat (for each episode)

Initialize *s*

Repeat (for each step of the episode)

Choose *a* from *s* using a policy

Take action *a*, observe *r, s'*

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

$$s \leftarrow s'$$

Exploration and Expliotation

Initialize Q(s, a) arbitrarily.

Repeat (for each episode)

Initialize s

Repeat (for each step of the episode)

Choose a from s using a policy

Take action a, observe r, s'

Update Q and s….

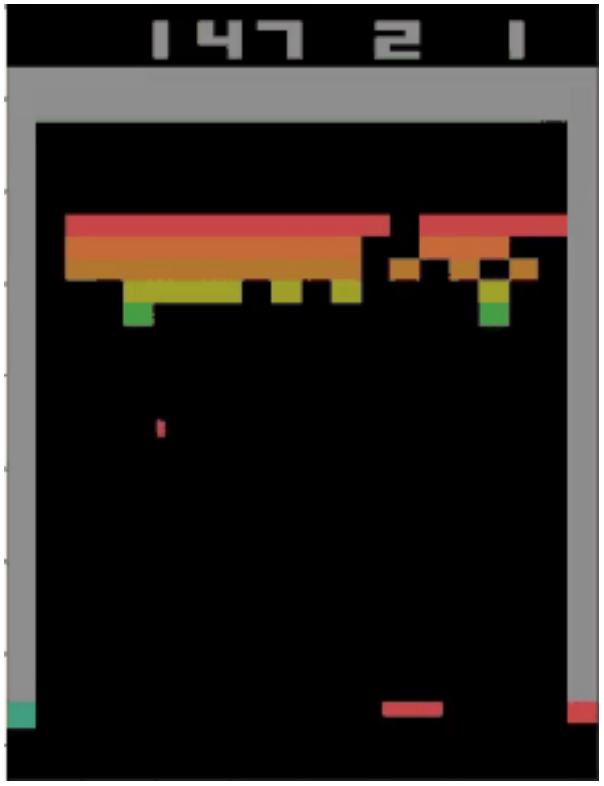
Random policy; Exploration

Greedy policy; Exploitation

$$\pi(s) = \arg\max_a Q(s, a)$$

ϵ-greedy policy: With probability ϵ select a random action

Deep Q-Learning



states

actions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Noop	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Fire	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Right	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Left	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

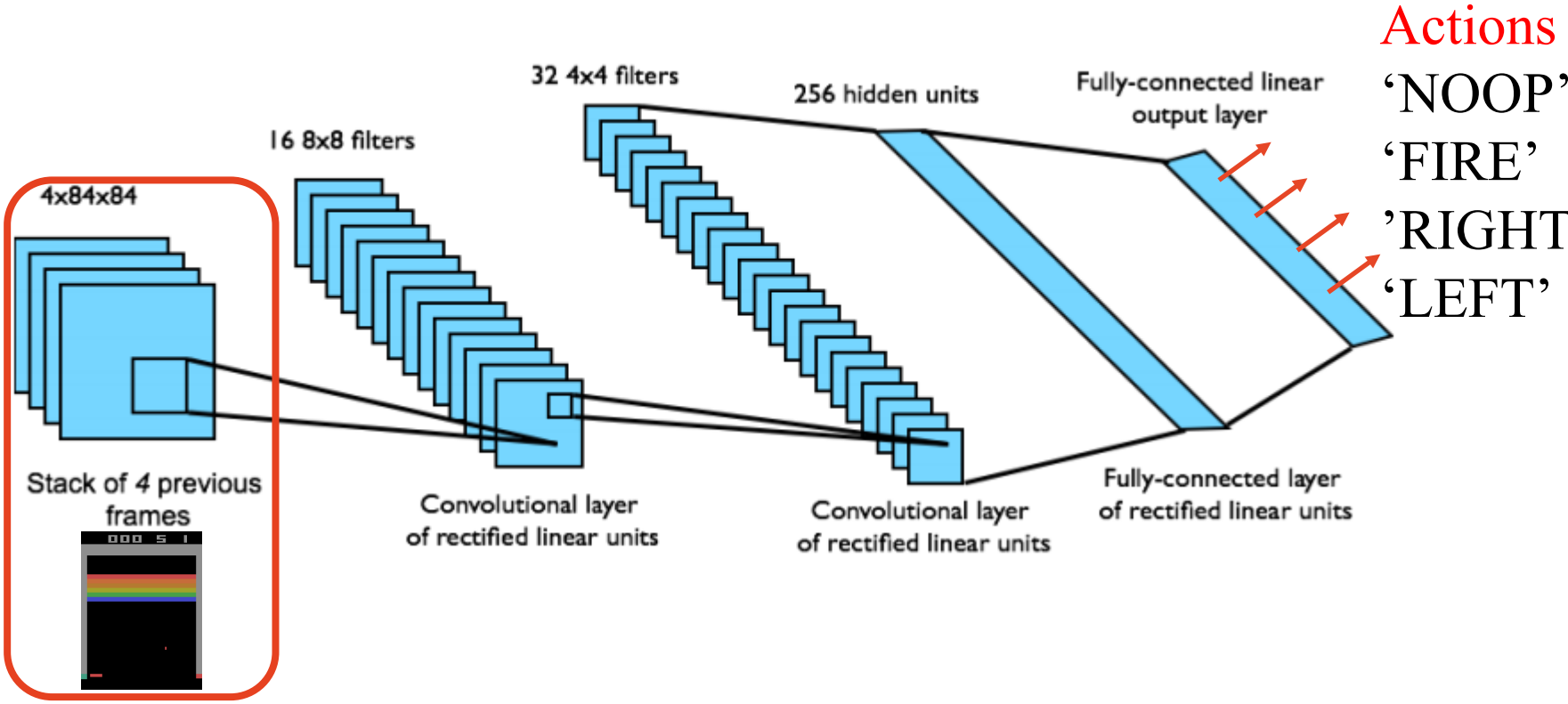
$$\#\text{states} = 256^{210\times160}$$

Value Function Approximation

$$Q(s,a) = f(s,a,w)$$

Deep Q-Learning

- End-to-end learning of state-action values from raw pixels
- Input state is stack of raw pixels from last 4 frames
- Output are state-action values from all possible actions



From the *Tutorial: Deep Reinforcement Learning* by David Silver, Google DeepMind

Deep Q-Networks (DQN)

- Optimal Q-values obey Bellman equation

$$Q^*(s,a) = \mathbb{E}_{s'}\left\{ r + \gamma \max_{a'} Q(s',a') | s,a \right\}$$

- Treat right-hand size as a target and minimize MSE loss by SGD

$$I = \left(r + \gamma \max_{a'} Q_w(s',a') - Q_w(s,a) \right)^2$$

Target

- Divergence issues using neural networks due to
  - Correlations between samples
  - Non-stationary targets

From the *Tutorial: Deep Reinforcement Learning* by David Silver, Google DeepMind

Algorithm

Algorithm 1 Deep Q-learning with Experience Replay

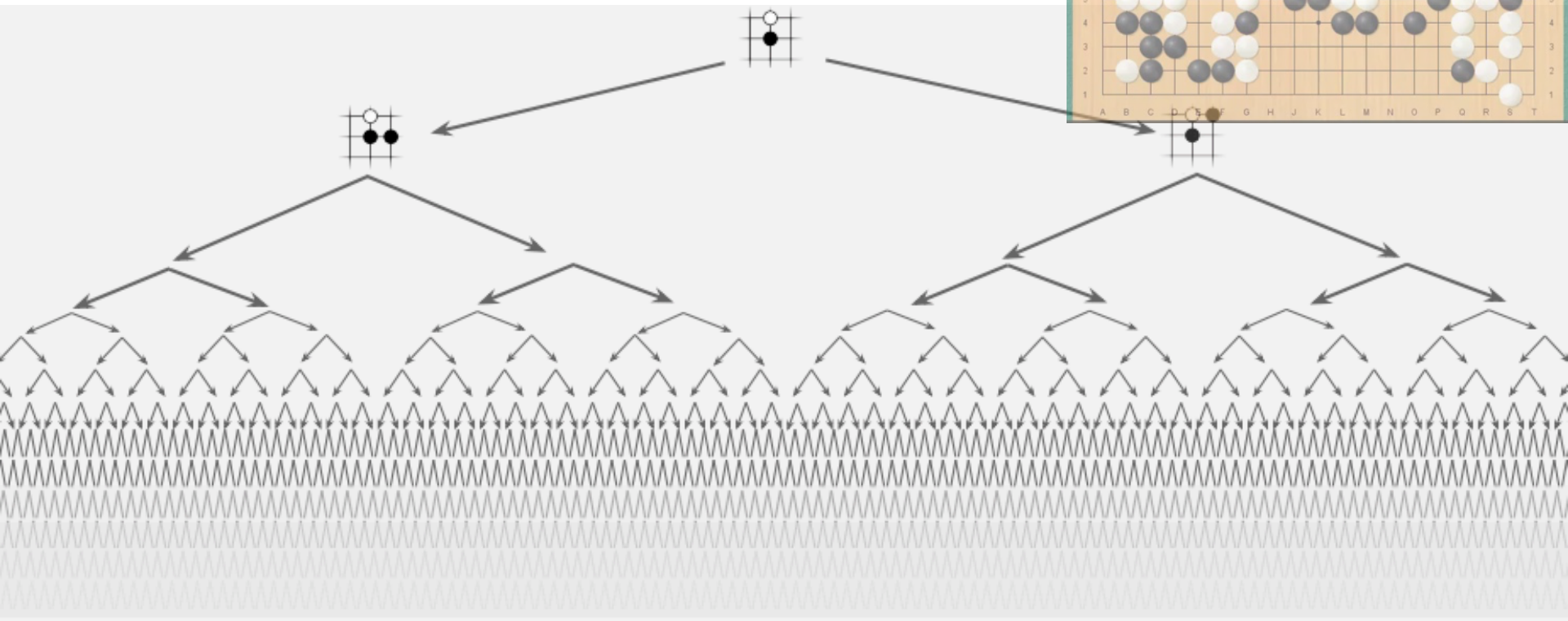
Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to
  end for
end for

ϵ-greedy

Deep RL in Alpha Go

- To win a game, exhaustive search?

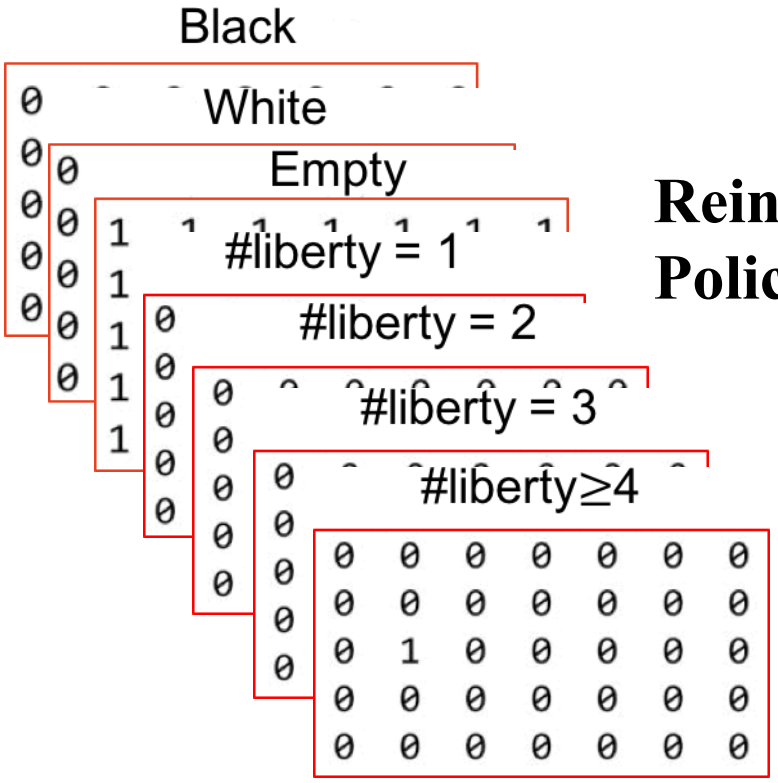
Monte Carlo search tree



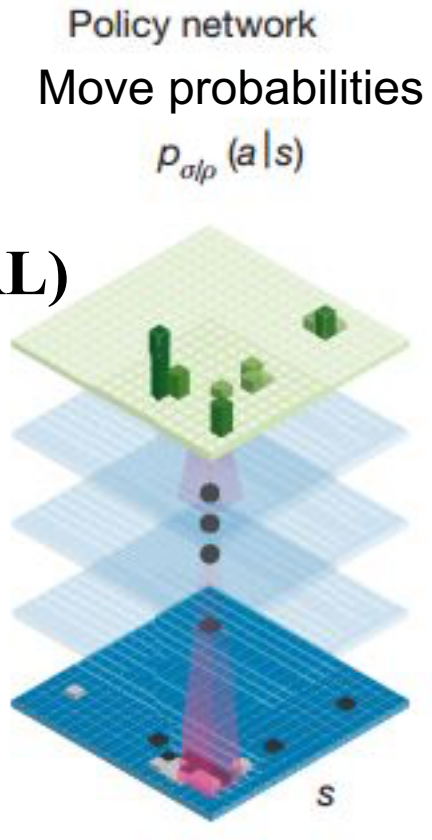
From https://icml.cc/2016/tutorials/AlphaGo-tutorial-slides.pdf

Policy Network

Supervised Learning (SL)
Policy Network



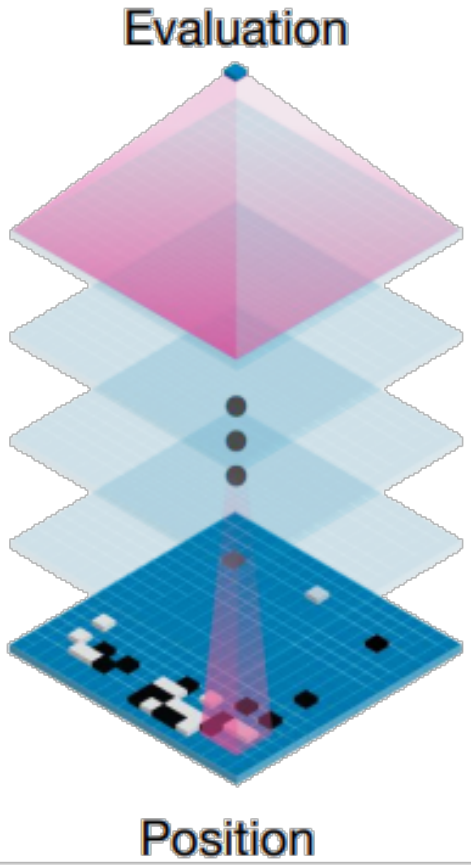
Reinforcement Learning (RL)
Policy Network

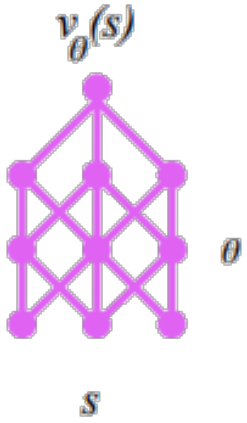


Convolutional Neural Network

Value Network

- To estimating probability that current move leads to win





From https://icml.cc/2016/tutorials/AlphaGo-tutorial-slides.pdf

Date	Time	Writing Time (minutes)	Venue	Building	Room
Friday 22/06/2018	1:50pm	120	Teachers College 323A	A22	323A
Friday 22/06/2018	1:50pm	120	VSCC Seminar Room 115	B22	115
Friday 22/06/2018	1:50pm	120	Quadrangle Building Latin 2 S225	A14	S225
Friday 22/06/2018	1:50pm	120	Quadrangle Building Room S241	A14	S241
Friday 22/06/2018	1:50pm	120	Woolley N208	A20	N208

EXAM WRITING TIME: 2 hours

- 8 choice questions
- 2 calculation questions
- 4 essay questions

- Calculator - non-programmable
- One A4 sheet of handwritten and/or typed notes double sided

Thanks!