Machine Learning Tutorial
Part 3
[CNN, GAN, RNN, DQN]

Inha University

Prof. Sang-Jo Yoo

sjyoo@inha.ac.kr

Contents (Part 3)

- **Convolutional Neural Network**
  - Background
  - Convolution Layer
  - Padding
  - Pooling Layer
  - CNN Structures
- **Generative Adversarial Network**
  - Generative and Discriminative Models
  - Objective Function of GAN
  - Applications of GAN
- **Recurrent Neural Network**
  - Motivations
  - RNN Model Sequences
  - Vanilla RNN Architecture
  - Long Short Term Memory (LSTM)
- **Reinforcement Learning**
  - Reinforcement Learning Model
  - Q-Learning Algorithm
  - Deep Reinforcement Learning (DQN)
-

Convolutional Neural Network

Inha University

Prof. Sang-Jo Yoo

sjyoo@inha.ac.kr

Background

Convolution Layer

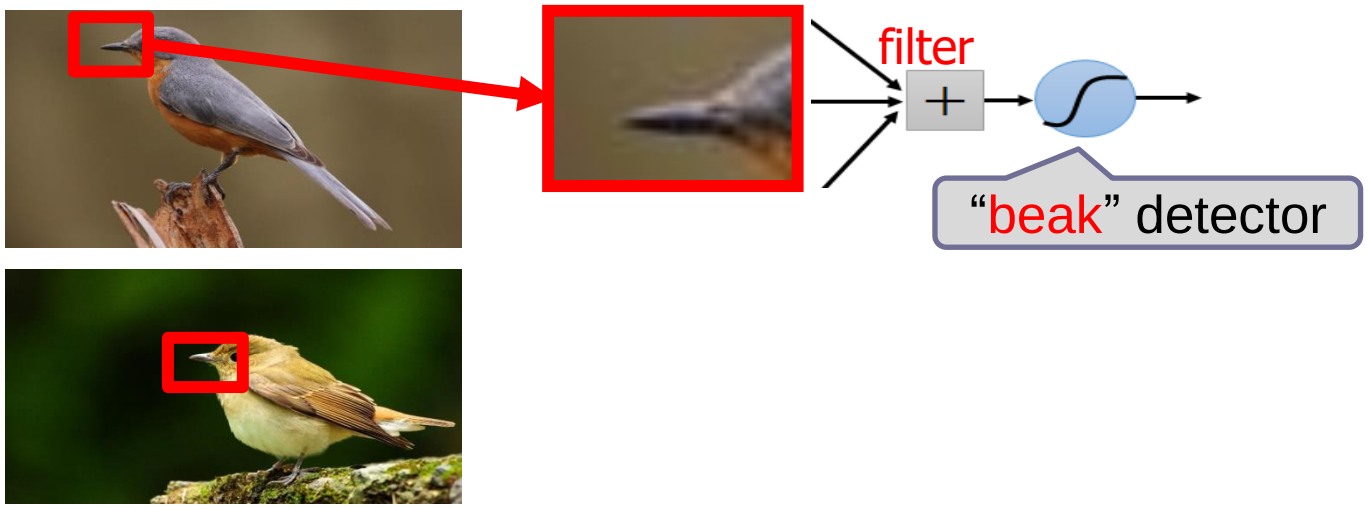
Padding

Pooling Layer

CNN Structures

Background

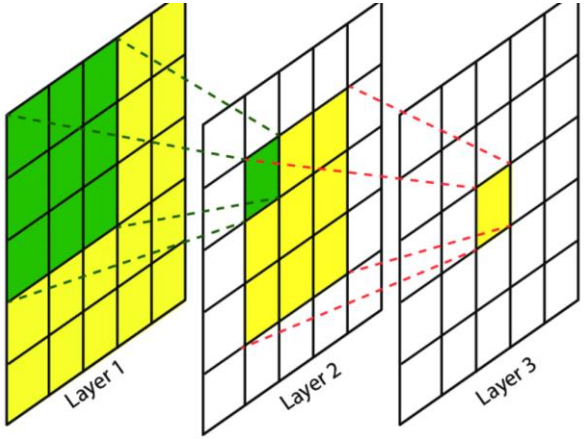
- **Some patterns are much smaller than the whole image.**
  - We can represent a small region with fewer parameters.
- **Same pattern appears in different places: They can be compressed (use the same detector).**
- **What about training a lot of such "small" detectors and each detector must "move around".**



Background

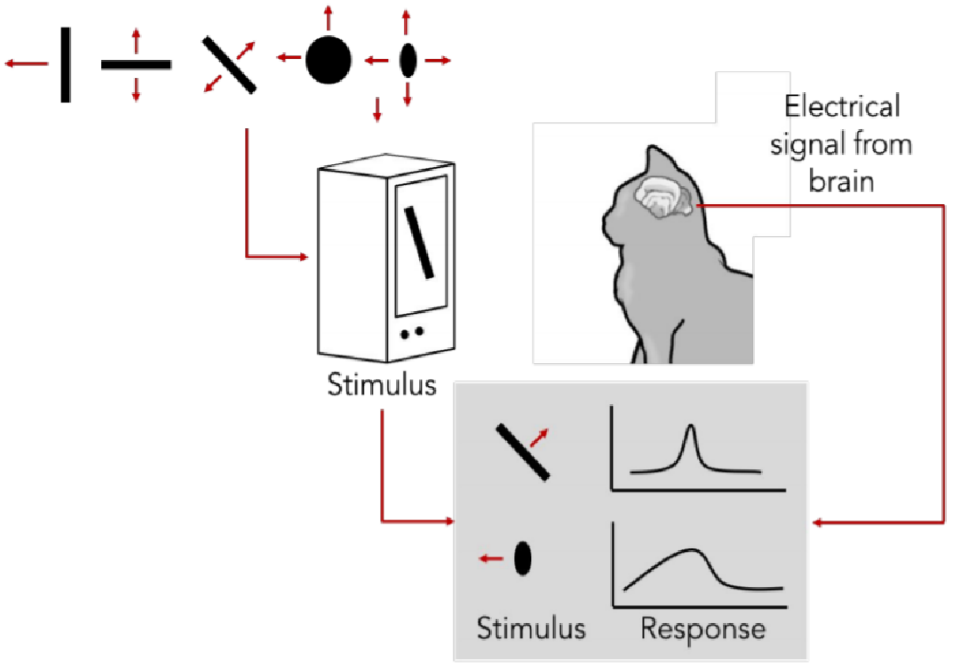
Convolutional Network

- Character recognition – where neighboring pixels will have high correlations and local features (edges, corners, etc.), while distant pixels (features) are un-correlated.
- Natural images have the property of being stationary, meaning that the statistics of one part of the image are the same as any other part.
- While standard NN nodes take input from all nodes in the previous layer, *CNNs enforce that a node receives only a small set of features which are spatially or temporally close to each other called* ***receptive fields***.



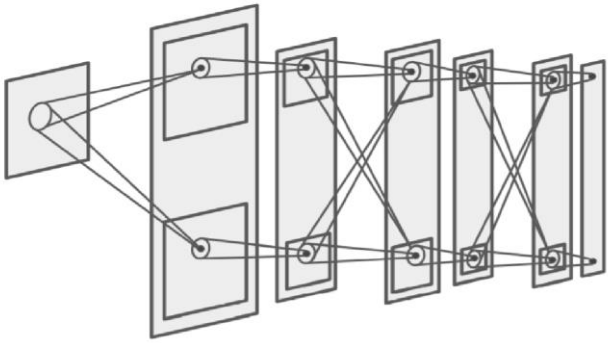
History of CNN

- **Hubel and Wiesel in the 1950s and 1960s**
  - showed that cat and monkey visual cortexes contain neurons that individually respond to small regions of the visual field.
  - Provided the eyes are not moving, the region of visual space within which visual stimuli affect the firing of a single neuron is known as its receptive field. Neighboring cells have similar and overlapping receptive fields.

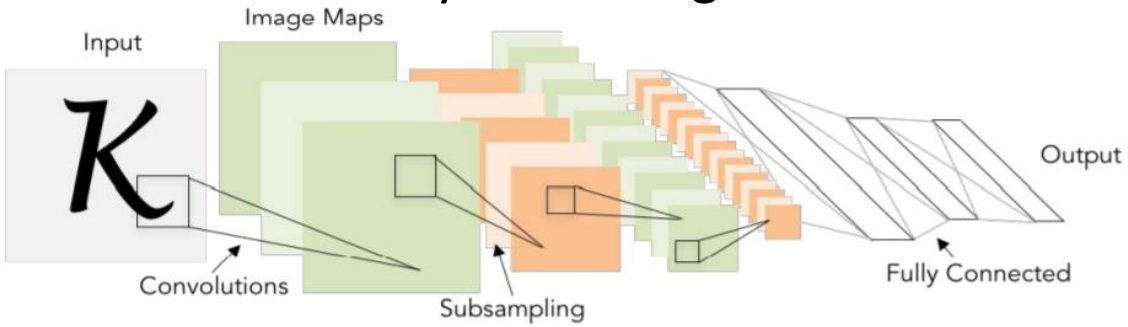


History of CNN

- **Kunihiko Fukushima**
  - The "neocognitron" was introduced by Kunihiko Fukushima in 1980. The neocognitron introduced the two basic types of layers in CNNs: convolutional layers, and down sampling layers.



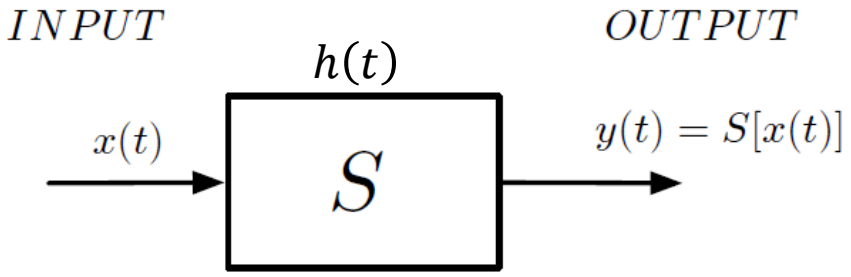
- **Image recognition with CNNs trained by gradient descent**
  - Yann LeCun et al. (1989) used back-propagation to learn the convolution kernel coefficients directly from images of hand-written numbers.



LeNet-5 [LeCun, Bottou, Bengio, Haffner 1998]

Convolution

- **For a system**
  - The behavior of a linear, continuous-time, time-invariant system with input signal $x(t)$ and output signal $y(t)$ is described by the convolution integral.
  - For given input $x(t)$ and output $y(t)$, we can make a system with $h(t)$ that makes $y(t) = x(t) * h(t)$.

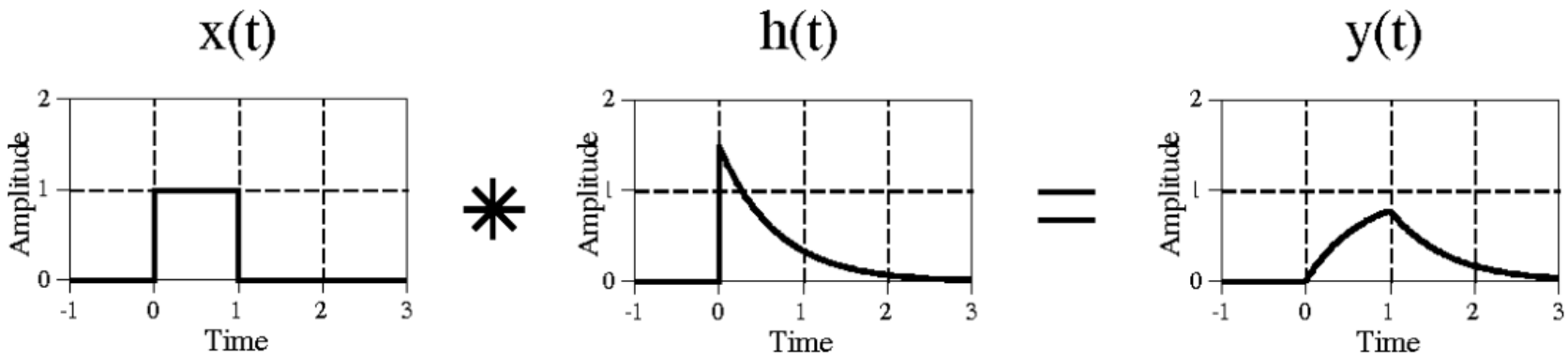


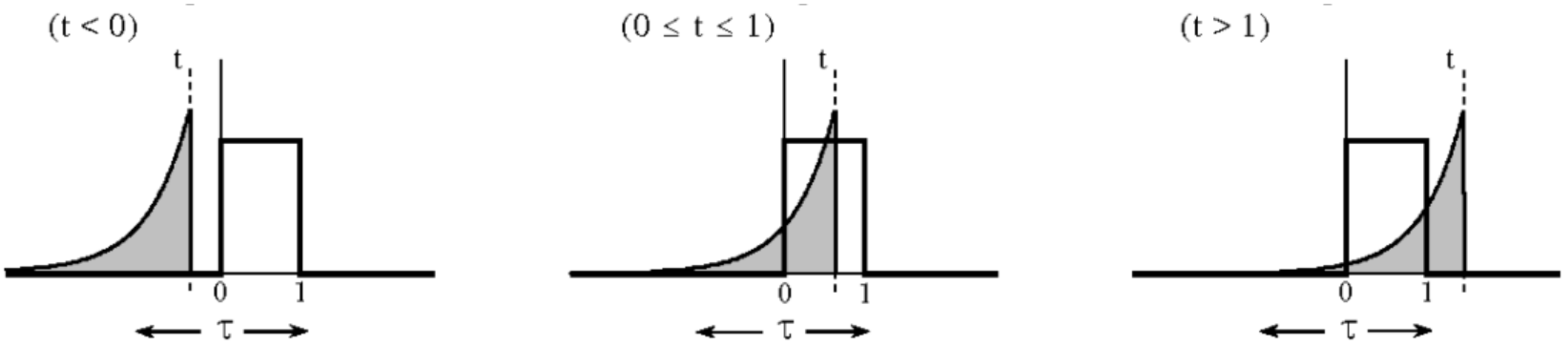
$h(t)$: *impulse response function*

$$y(t) = \int_{-\infty}^{\infty} x(\tau)h(t-\tau)d\tau$$
$$= \int_{-\infty}^{\infty} x(t-\tau)h(\tau)d\tau$$
$$= [x * h](t) = [h * x](t)$$

Convolution

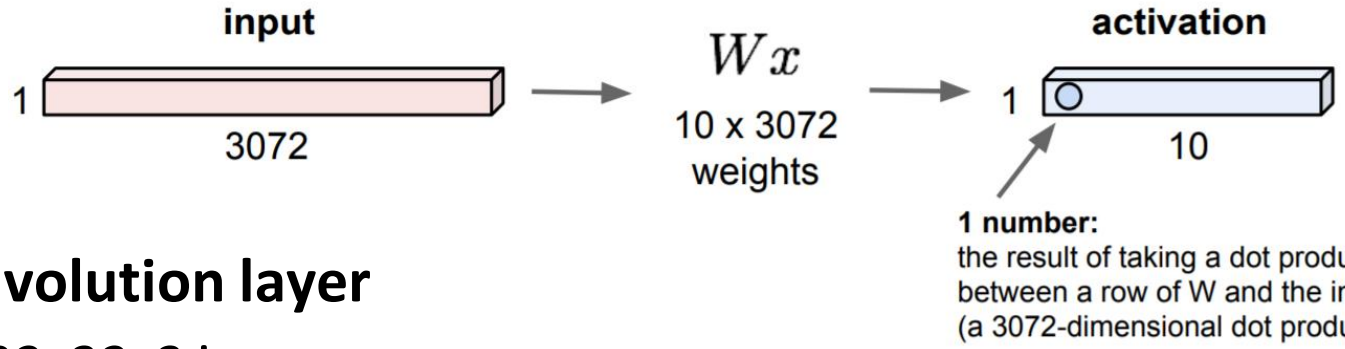
$$y(t) = \int_{-\infty}^{\infty} x(\tau)h(t-\tau)d\tau$$



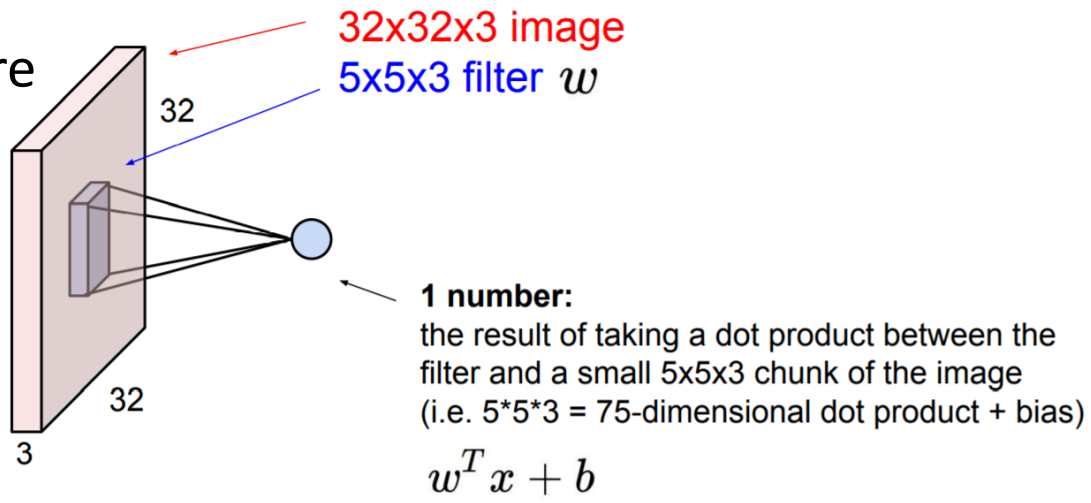


Convolution Layer

- **Fully connected layer**
  - 32x32x3 image -> stretch to 3072 x 1



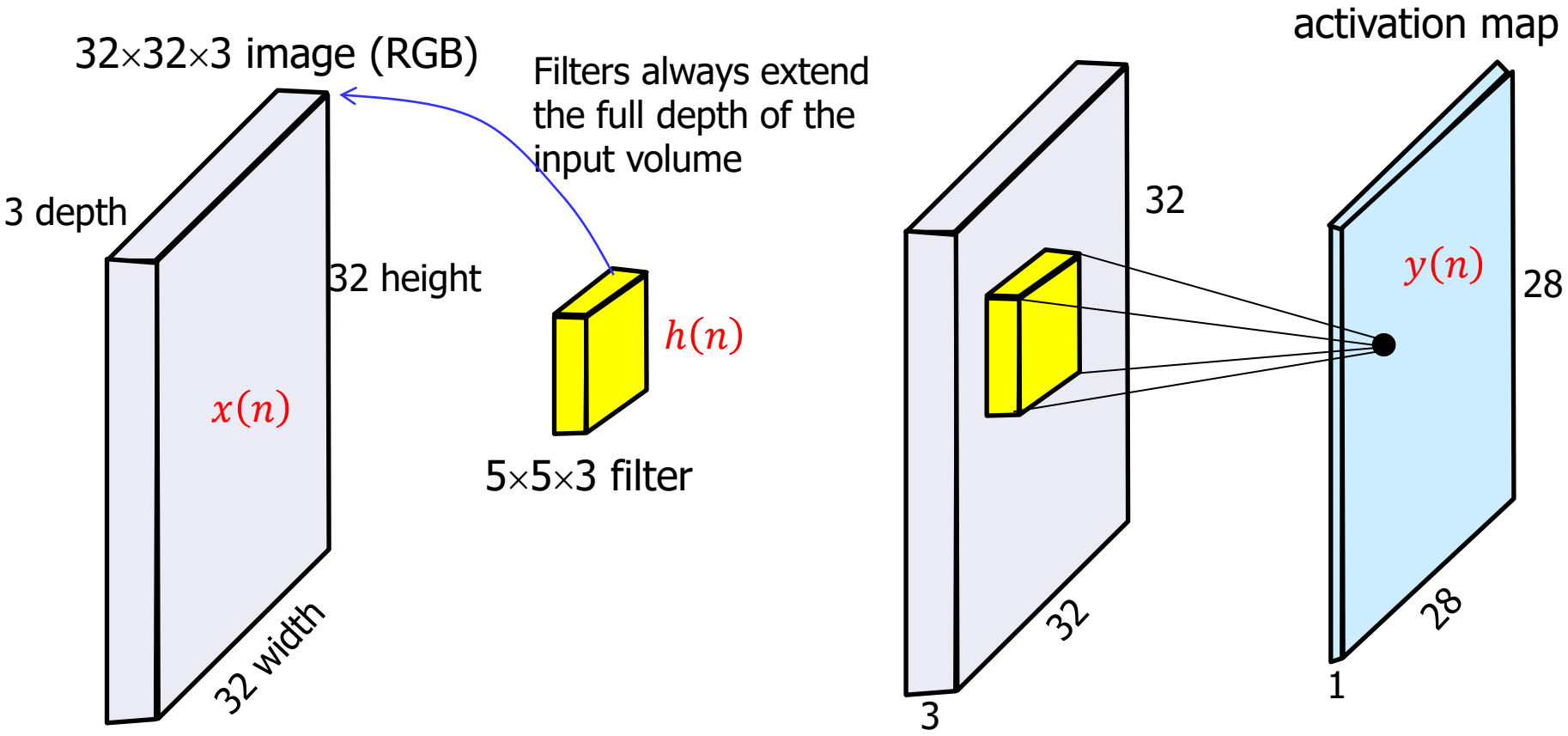
- **Convolution layer**
  - 32x32x3 image ->
    preserve spatial structure



http://cs231n.stanford.edu/slides/2017

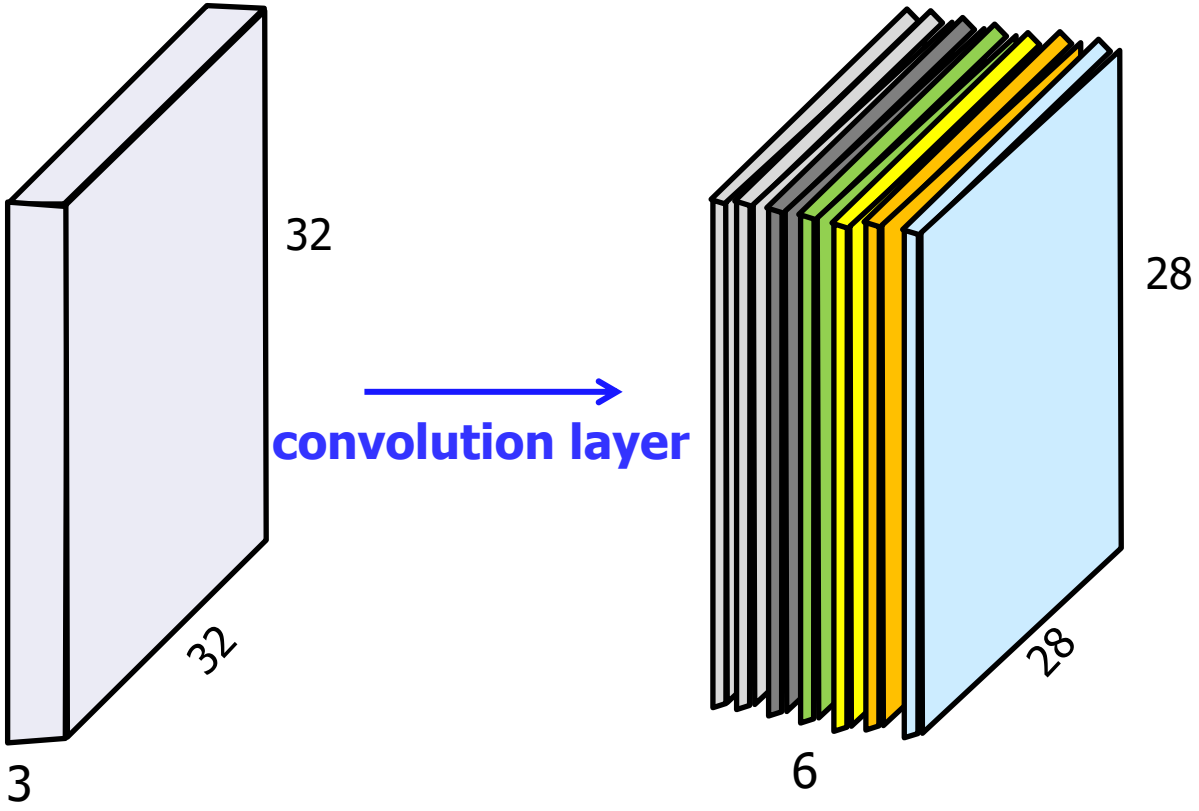
Convolution Layer

- **Convolve the filter with the image: "slide over the image spatially, computing dot products"**



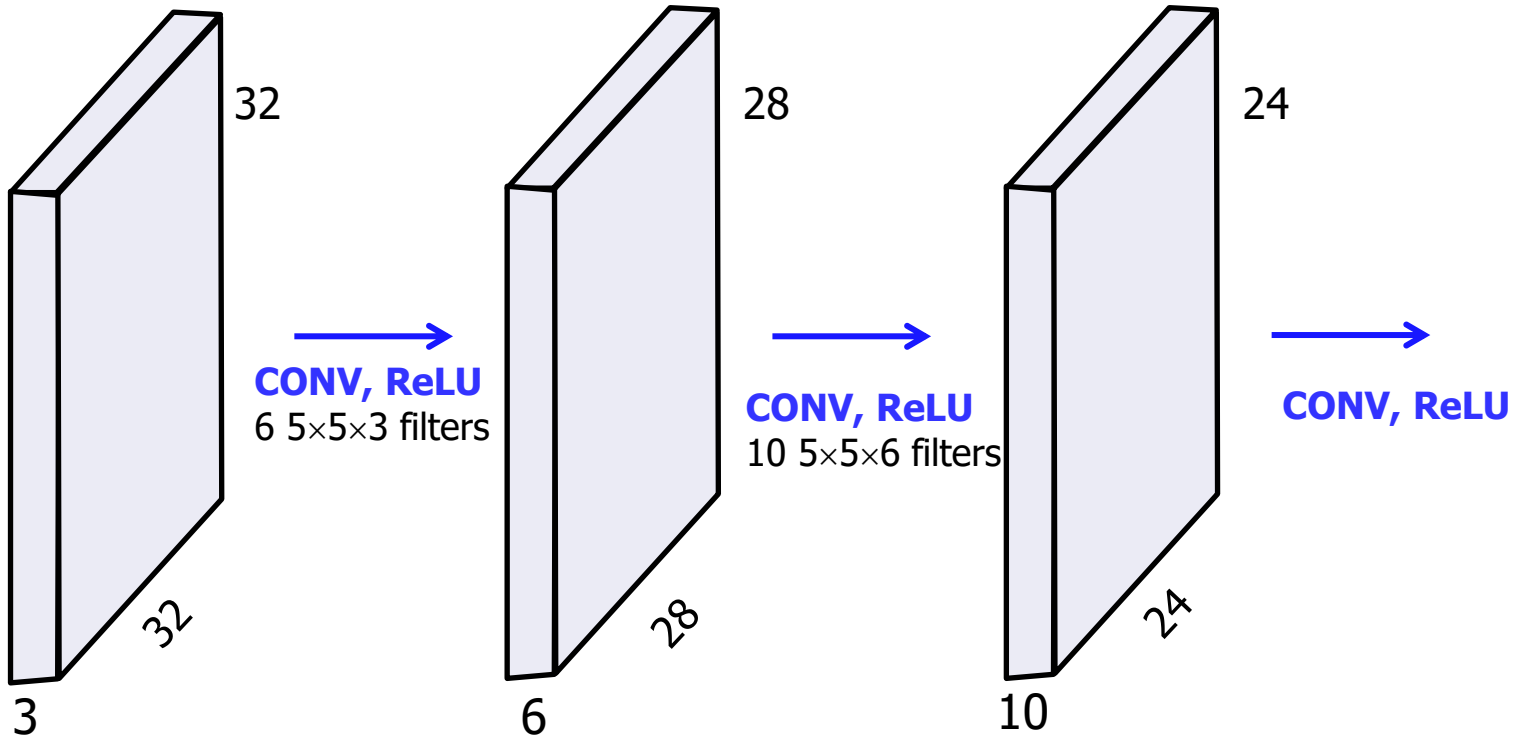
Convolution Layer

- If we have 6 (5×5×3) filters, then we'll get 6 separate activation maps



Convolution Layer

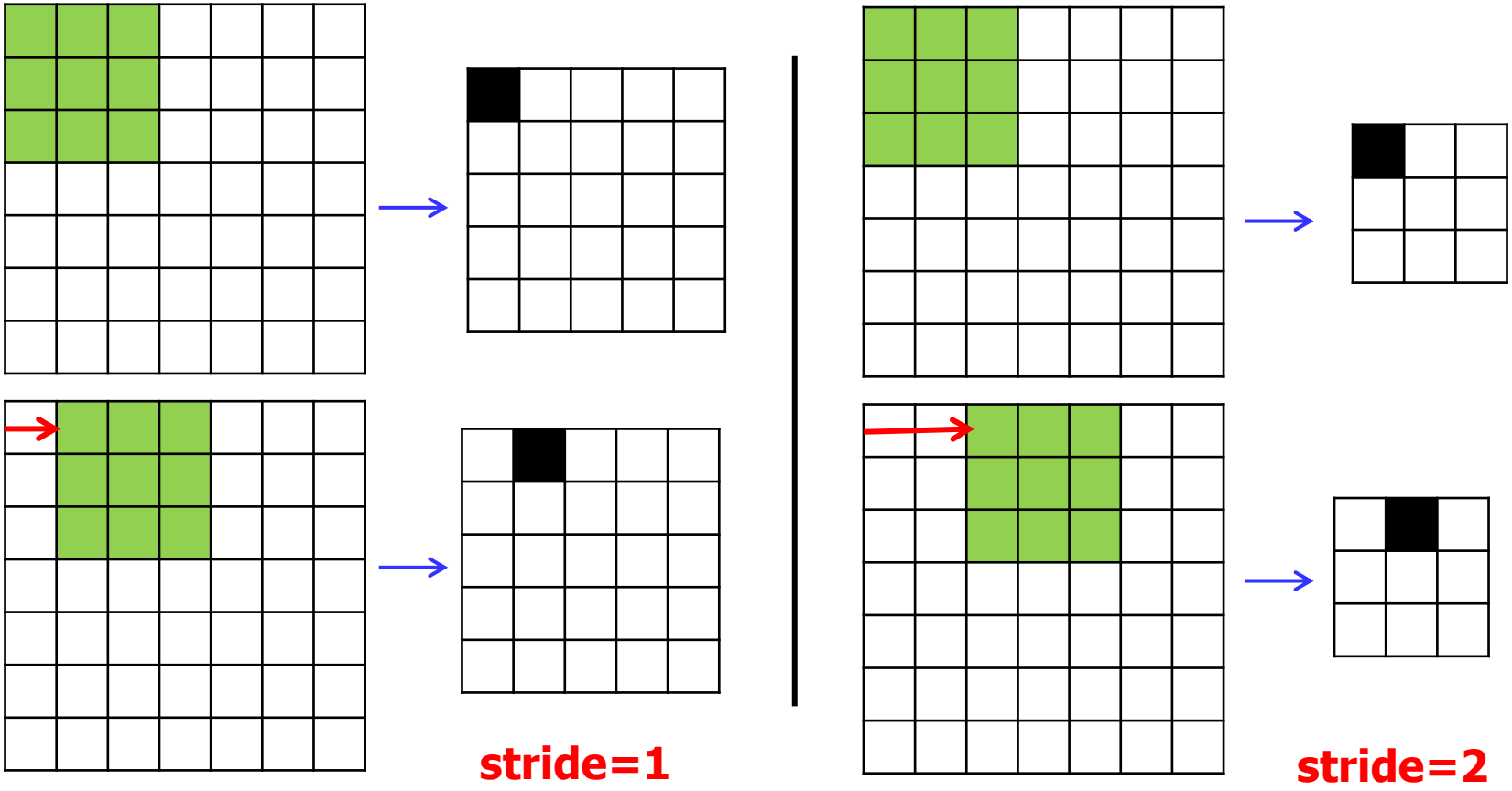
- **Convolutional Neural Network is**
  - a sequence of Convolutional Layers, interspersed with activation functions.



Convolution Layer

- **A closer look at spatial dimensions**
  - 7×7 input (spatially) , assume 3×3 filter

output size=(N-F)/stride+1



Padding

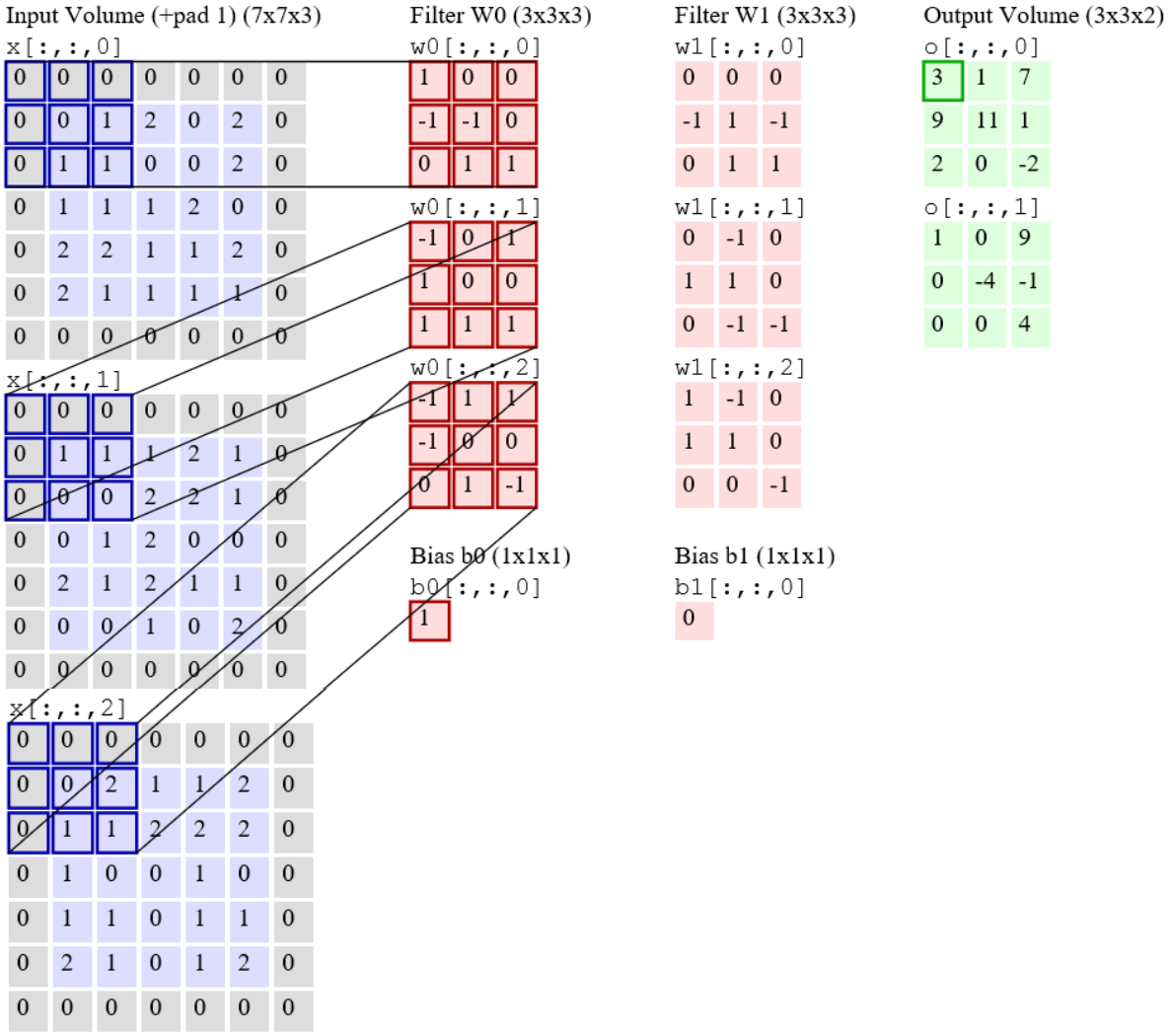
- **In practice**
  - Common to zero pad the border

0	0	0	0	0	0	0	0
0							0
0							0
0							0
0							0
0							0
0							0
0	0	0	0	0	0	0	0

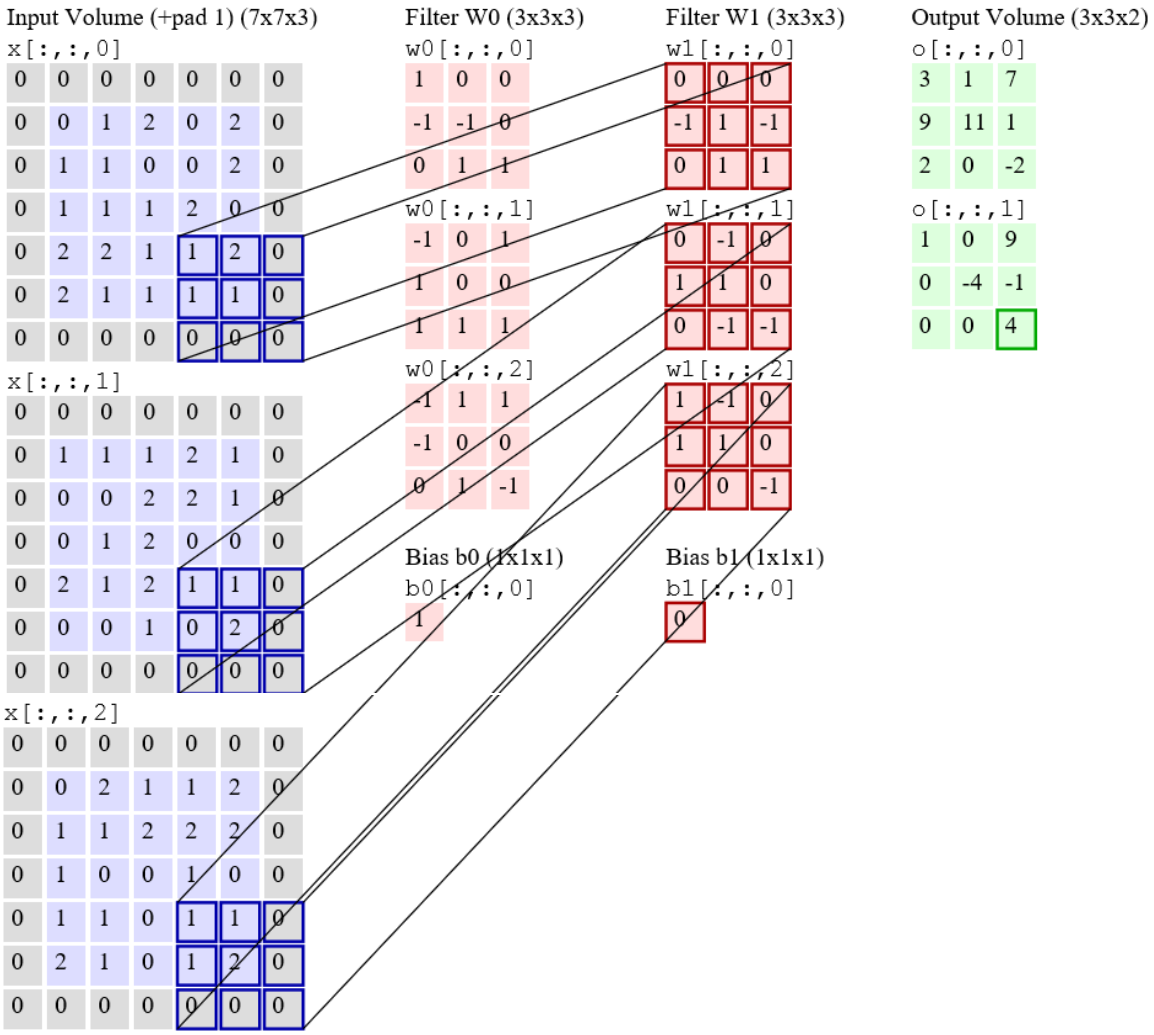
e.g., input 7×7
3×3 filter, applied with **stride 1**
Pad with 1 pixel border →
what is the output?

7×7 output!

Example

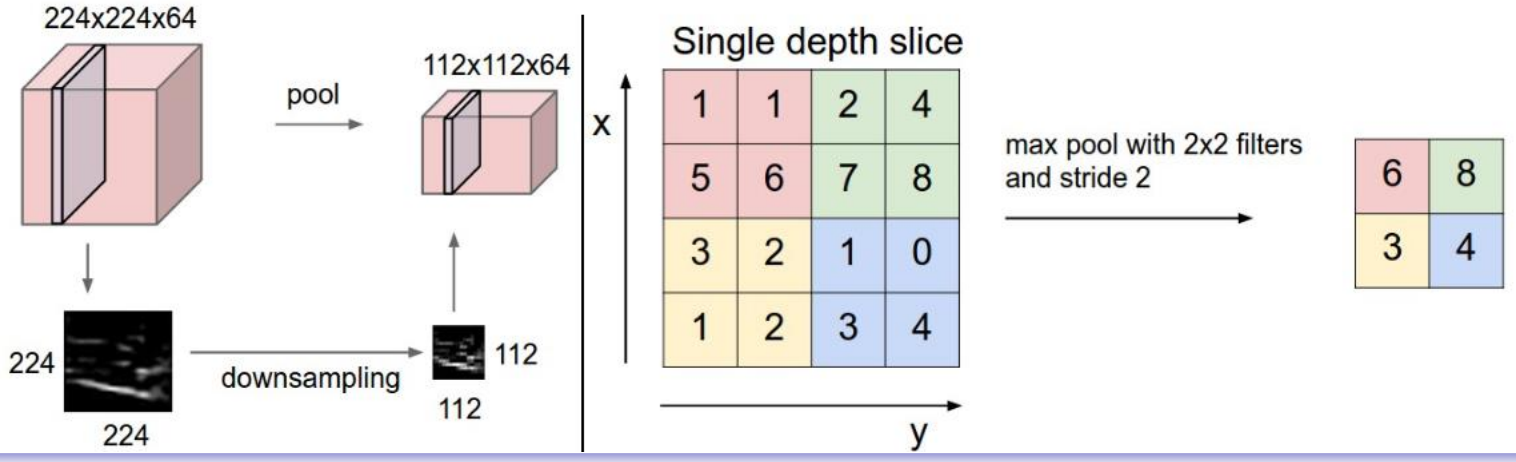

Input Volume (+pad 1) (7x7x3)
x[:,:,0]
0 0 0 0 0 0 0
0 0 1 2 0 2 0
0 1 1 0 0 2 0
0 1 1 1 2 0 0
0 2 2 1 1 2 0
0 2 1 1 1 1 0
0 0 0 0 0 0 0
x[:,:,1]
0 0 0 0 0 0 0
0 1 1 1 2 1 0
0 0 0 2 2 1 0
0 0 1 2 0 0 0
0 2 1 2 1 1 0
0 0 0 1 0 2 0
0 0 0 0 0 0 0
x[:,:,2]
0 0 0 0 0 0 0
0 0 2 1 1 2 0
0 1 1 2 2 2 0
0 1 0 0 1 0 0
0 1 1 0 1 1 0
0 2 1 0 1 2 0
0 0 0 0 0 0 0
Filter W0 (3x3x3)
w0[:,:,0]
1 0 0
-1 -1 0
0 1 1
w0[:,:,1]
-1 0 1
1 0 0
1 1 1
w0[:,:,2]
-1 1 1
-1 0 0
0 1 -1
Bias b0 (1x1x1)
b0[:,:,0]
1
Filter W1 (3x3x3)
w1[:,:,0]
0 0 0
-1 1 -1
0 1 1
w1[:,:,1]
0 -1 0
1 1 0
0 -1 -1
w1[:,:,2]
1 -1 0
1 1 0
0 0 -1
Bias b1 (1x1x1)
b1[:,:,0]
0
Output Volume (3x3x2)
o[:,:,0]
3 1 7
9 11 1
2 0 -2
o[:,:,1]
1 0 9
0 -4 -1
0 0 4

Example

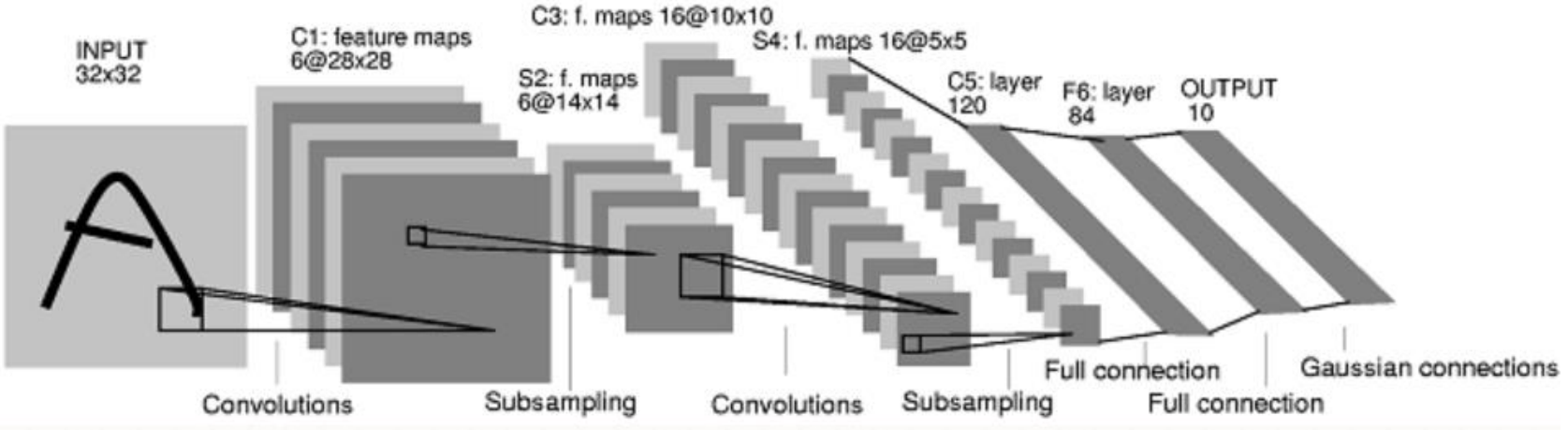

Input Volume (+pad 1) (7x7x3)
x[:,:,0]
0 0 0 0 0 0 0
0 0 1 2 0 2 0
0 1 1 0 0 2 0
0 1 1 1 2 0 0
0 2 2 1 1 2 0
0 2 1 1 1 1 0
0 0 0 0 0 0 0
x[:,:,1]
0 0 0 0 0 0 0
0 1 1 1 2 1 0
0 0 0 2 2 1 0
0 0 1 2 0 0 0
0 2 1 2 1 1 0
0 0 0 1 0 2 0
0 0 0 0 0 0 0
x[:,:,2]
0 0 0 0 0 0 0
0 0 2 1 1 2 0
0 1 1 2 2 2 0
0 1 0 0 1 0 0
0 1 1 0 1 1 0
0 2 1 0 1 2 0
0 0 0 0 0 0 0
Filter W0 (3x3x3)
w0[:,:,0]
1 0 0
-1 -1 0
0 1 1
w0[:,:,1]
-1 0 1
1 0 0
1 1 1
w0[:,:,2]
-1 1 1
-1 0 0
0 1 -1
Bias b0 (1x1x1)
b0[:,:,0]
1
Filter W1 (3x3x3)
w1[:,:,0]
0 0 0
-1 1 -1
0 1 1
w1[:,:,1]
0 -1 0
1 1 0
0 -1 -1
w1[:,:,2]
1 -1 0
1 1 0
0 0 -1
Bias b1 (1x1x1)
b1[:,:,0]
0
Output Volume (3x3x2)
o[:,:,0]
3 1 7
9 11 1
2 0 -2
o[:,:,1]
1 0 9
0 -4 -1
0 0 4

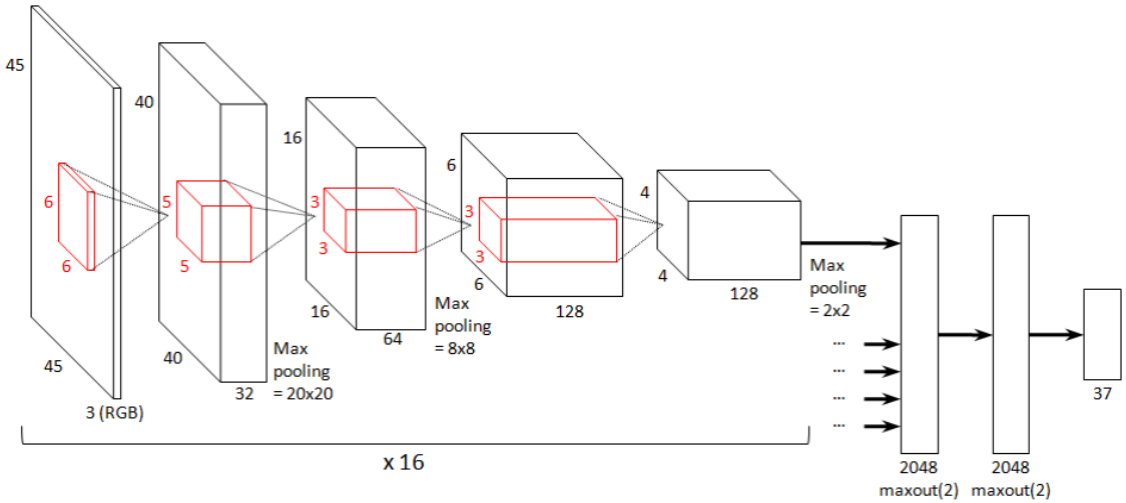
Pooling Layer

- **Polling (Max Pooling)**
  - makes the representations smaller and more manageable.
  - Pooling layer downsamples the volume spatially, independently in each depth slice(activation map) of the input volume.
    - **Left:** In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved.
    - **Right:** The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. Each max is taken over 4 numbers.



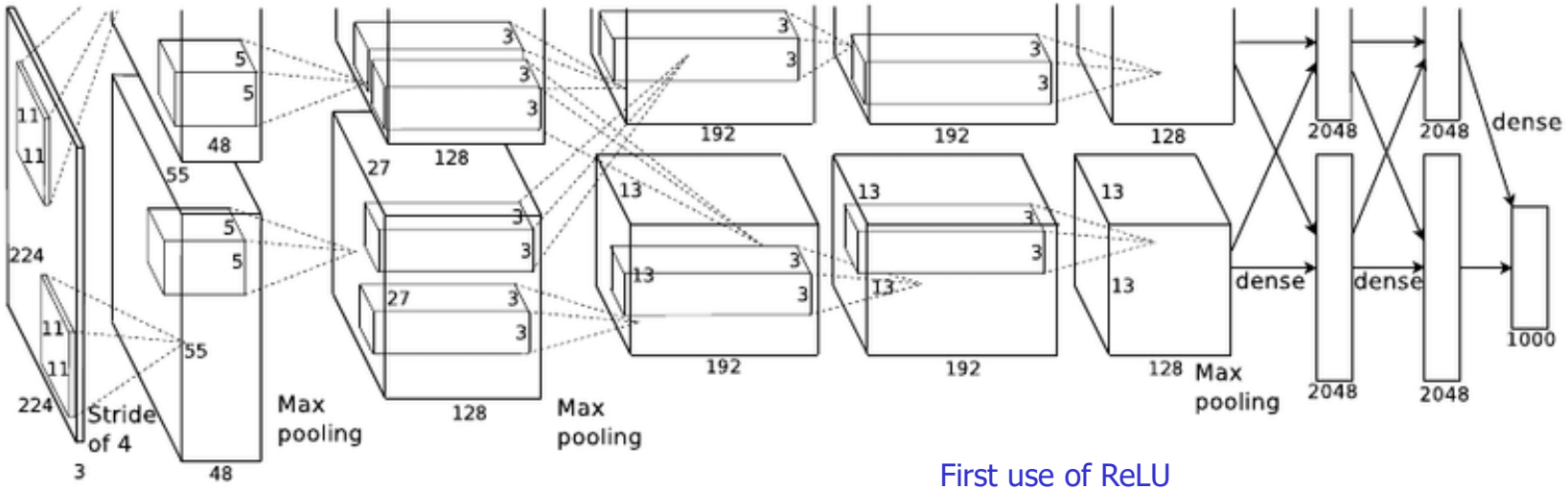
CNN Structure





"AlexNet"

- **ImageNet Classification with Deep Convolutional Neural Networks** *[Krizhevsky, Sutskever, Hinton, 2012]*
  - It is composed of 5 convolutional layers followed by 3 fully connected layers

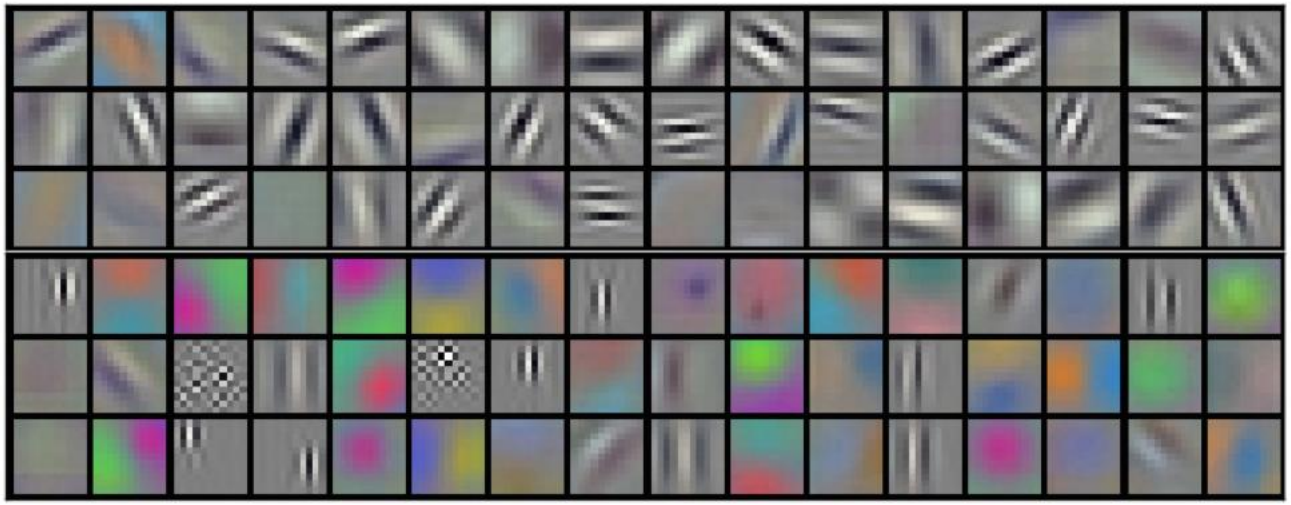


Images: 224x224x3
- F (receptive field size): 11X11
- S (stride) = 4
- Conv layer output: (55x55x48)x2

First use of ReLU
Dropout 0.5 Batch size 128
SGD Momentum 0.9
Learning rate 0.01, L2 Regularization
7 CNN ensemble

“AlexNet”

- **96 convolutional kernels (filters)**
  - 11 x 11 x 3 size kernels.
  - top 48 kernels on GPU 1 : color-agnostic
  - bottom 48 kernels on GPU 2 : color-specific.



Generative Adversarial Network (GAN)

Inha University

Prof. Sang-Jo Yoo

sjyoo@inha.ac.kr

Generative and Discriminative Models

Generative Adversarial Network

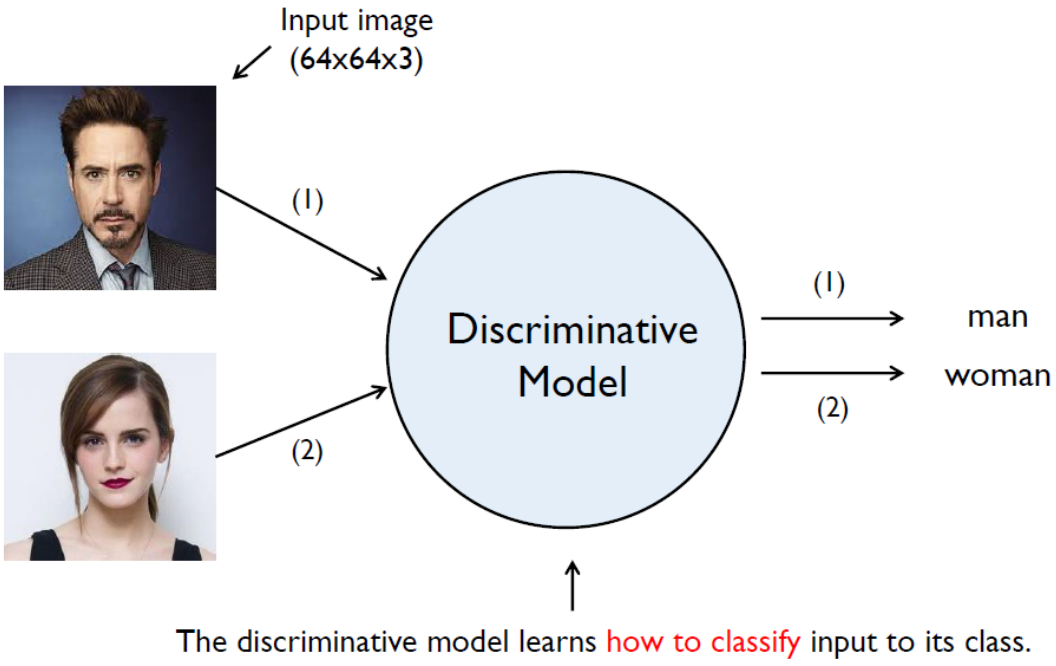
Objective Function of GAN

Applications of GAN

Ref: Yunjey Choi (Korea Univ.)
Namju Kim (GAN Introduction)

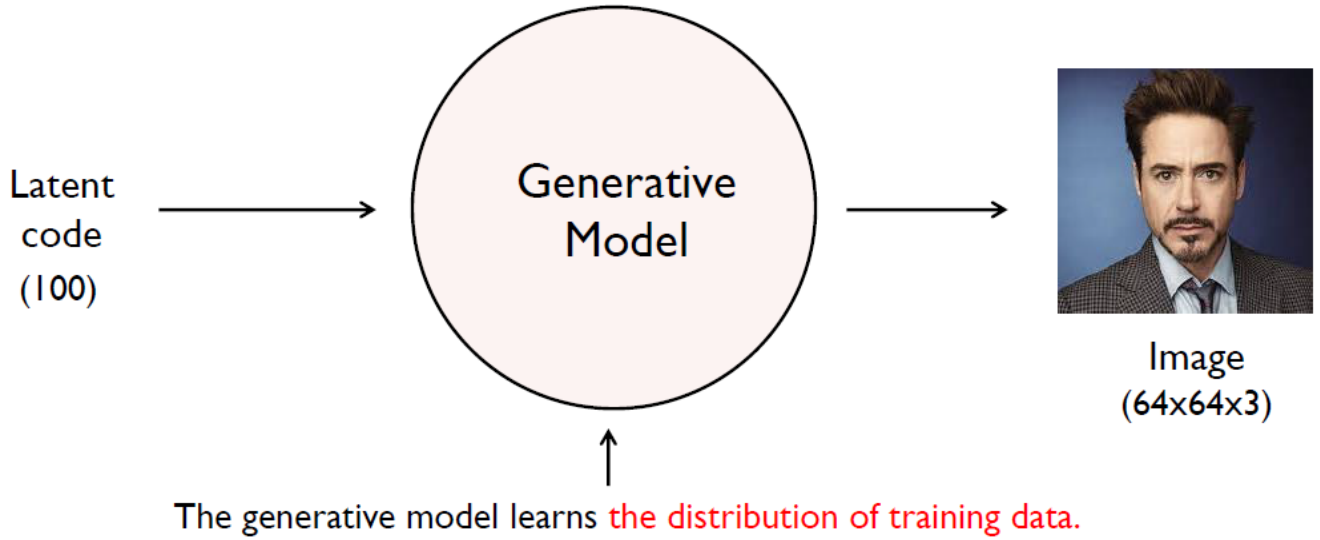
Discriminative Model

- Discriminative model is a model of the conditional probability of the target *Y*, given an observation *x*, symbolically, P (Y | X = x).
  - Training with {data, label}: supervised learning



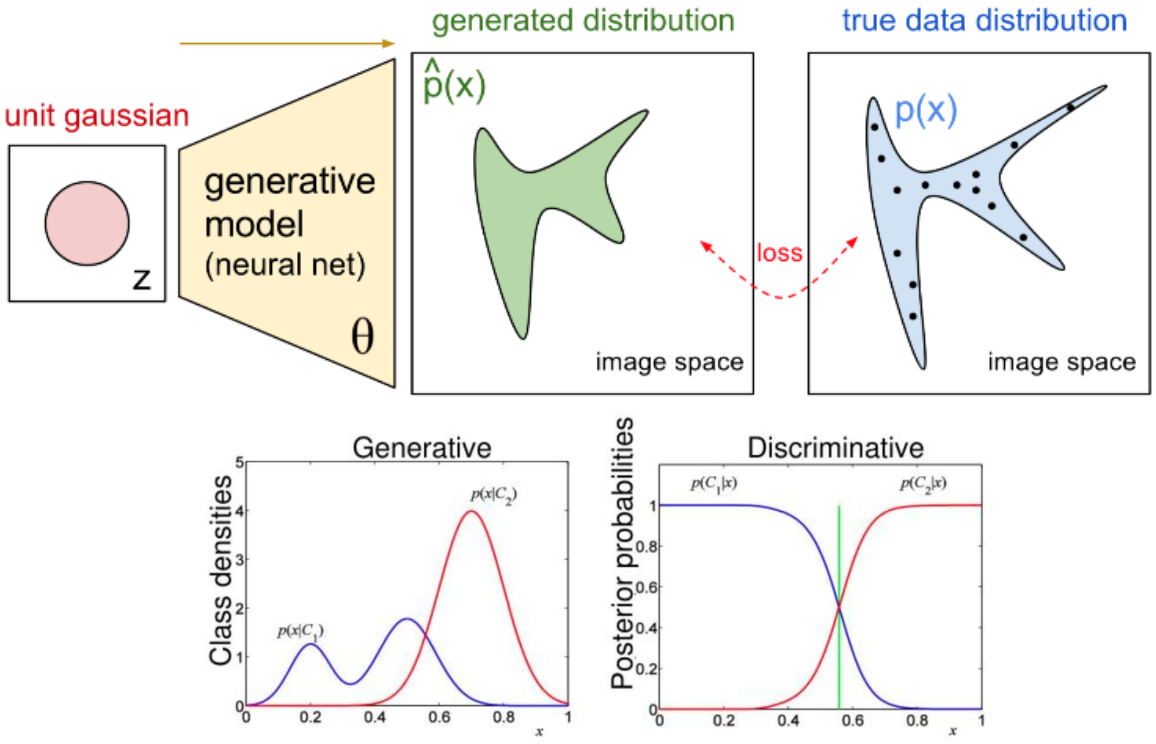
Generative Model

- Generative model is a model of the conditional probability of the observable *X*, given a target *y*, symbolically, P (X | Y = y)
  - A generative model describes how data is generated, in terms of a probabilistic model.
  - Training with data: unsupervised learning



Generative Model

- From the latent code (generally small dimension), generates a fake data that is very similar to the real data in terms of probability distribution functions.
- The generative model can be implemented with any machine learning methods (e.g., DNN or CNN)

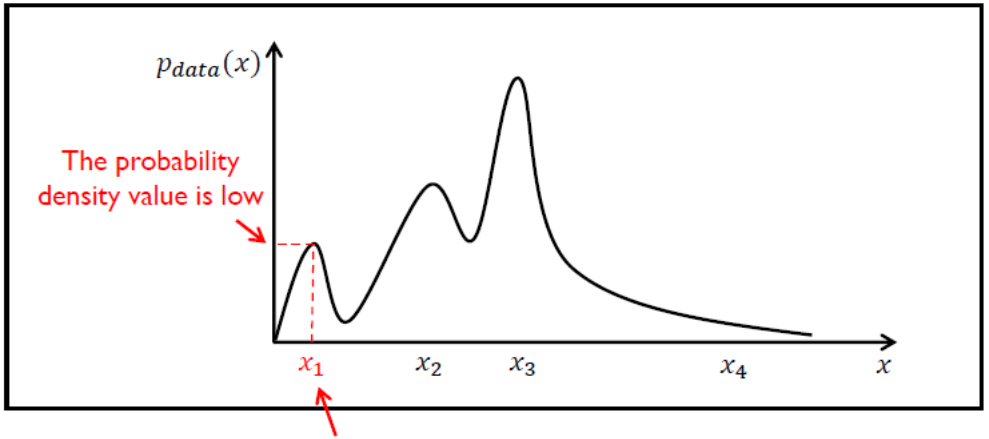


Probability Distribution

Probability density function

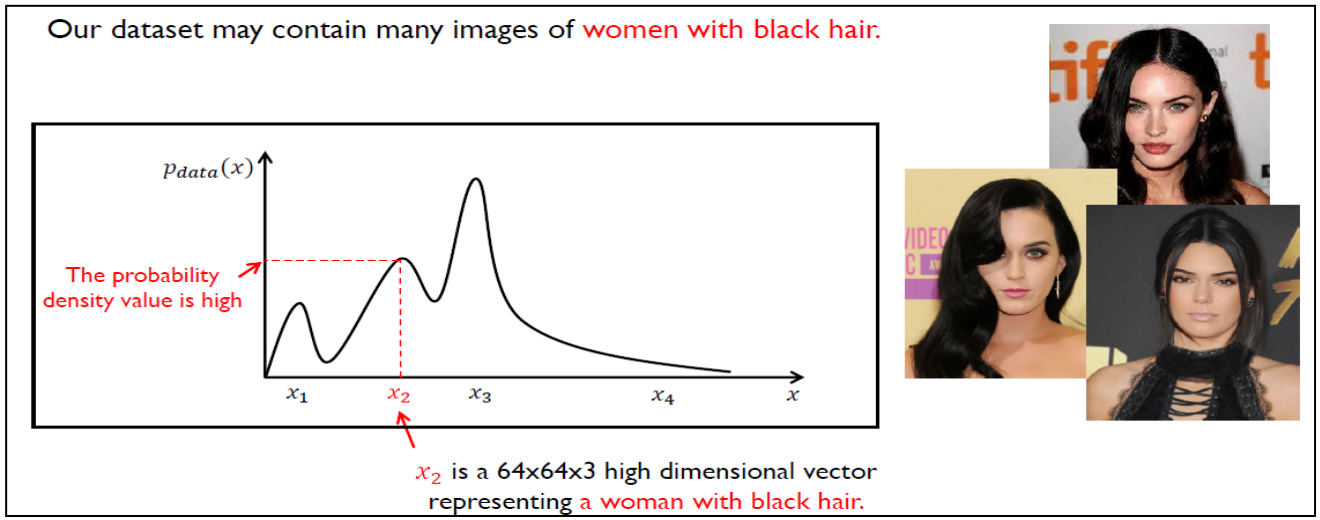
There is a $p_{data}(x)$ that represents the distribution of actual images.

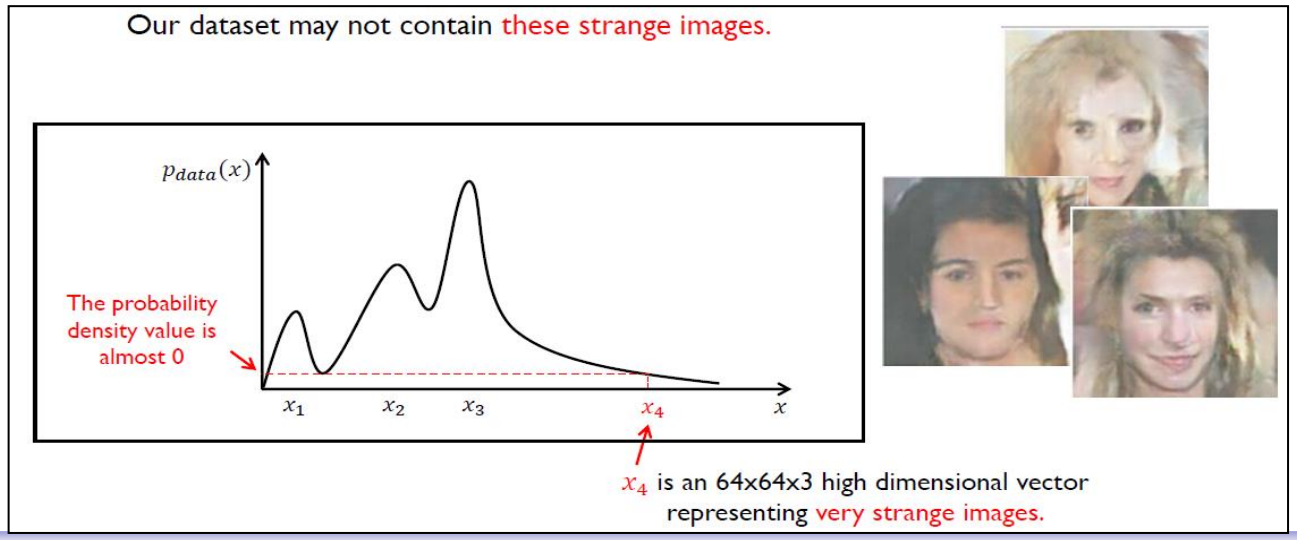
Let's take an example with human face image dataset.
Our dataset may contain few images of men with glasses.



x_1 is a 64x64x3 high dimensional vector representing a man with glasses.

Probability Distribution



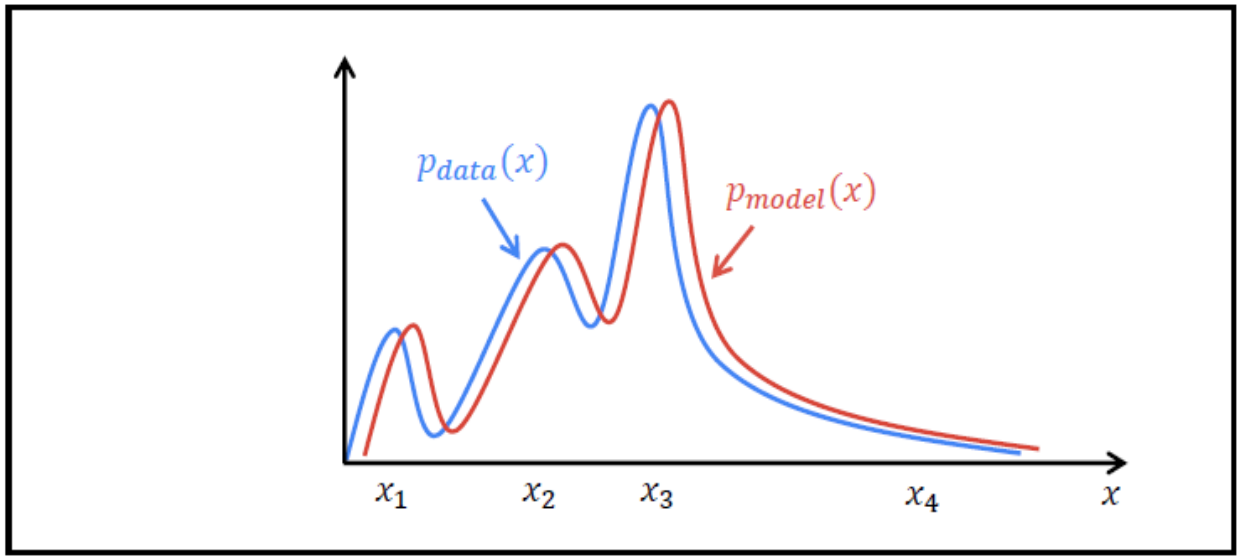


Probability Distribution

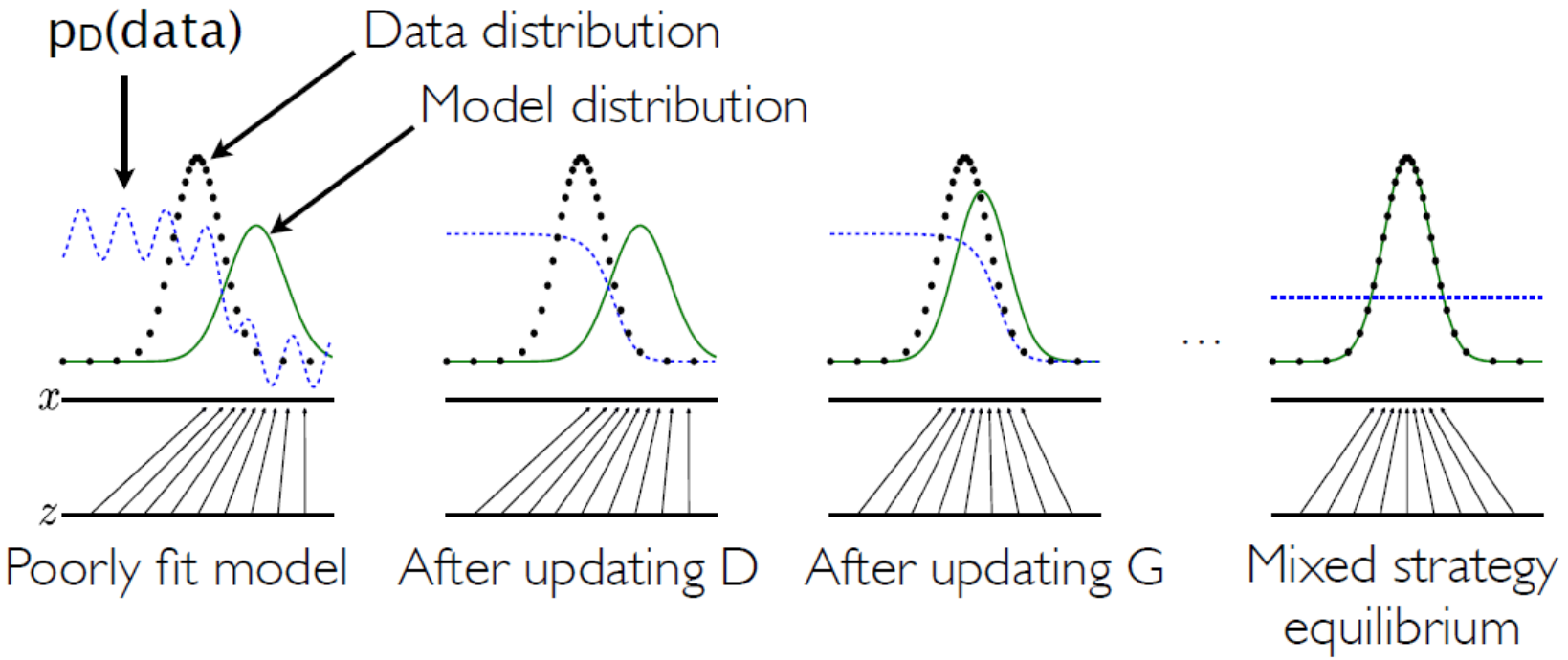
Distribution of images generated by the model

The goal of the generative model is to find a $p_{model}(x)$ that approximates $p_{data}(x)$ well.

Distribution of actual images



GAN Learning Process



Generative Adversarial Network

The discriminator should classify a real image as real.

D

May be high

D(x)

The probability of that x came from the real data (0~1)

Discriminator (Neural Network)

x

Real image

Training with real images

Training with fake images

Generator (Neural Network)

z

G

D

D(G(z))

May be low

Latent Code

The generator should create an image that is indistinguishable from real to deceive the discriminator

G(z)

Fake image

The discriminator should classify a fake image as fake.

Objective Function of GAN

How to optimize 'Discriminator (D)'?

Sample x from real data distribution

Sample latent code z from Gaussian distribution

$$\min_G \max_D V(D,G) = E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$

D should maximize $V(D,G)$

Maximum when $D(x) = 1$

Maximum when $D(G(z)) = 0$

Objective function

X

D

D(x)

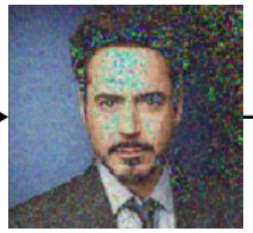
Train D to classify real images as real

Training with real images

Training with fake images

z

G



G(z)

D

D(G(z))

Train D to classify fake images as fake

Objective Function of GAN

How to optimize 'Generator (G)'?

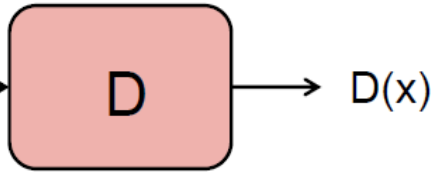
G is independent of this part

$$\min_G \max_D V(D,G) = \cancel{E_{x\sim p_{data}(x)}[\log D(x)]} + E_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$

G should minimize $V(D,G)$

Minimum when $D(G(z)) = 1$

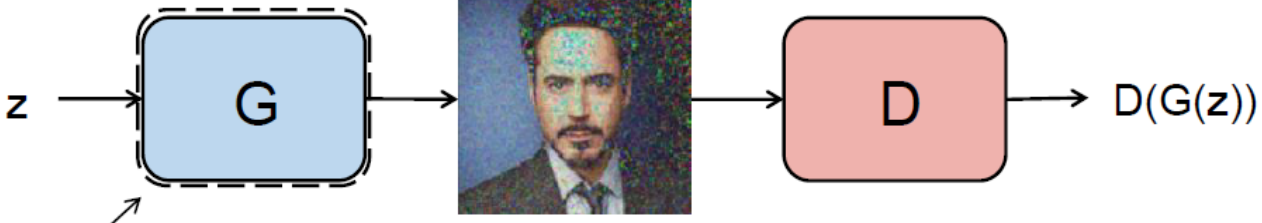
Objective function



x

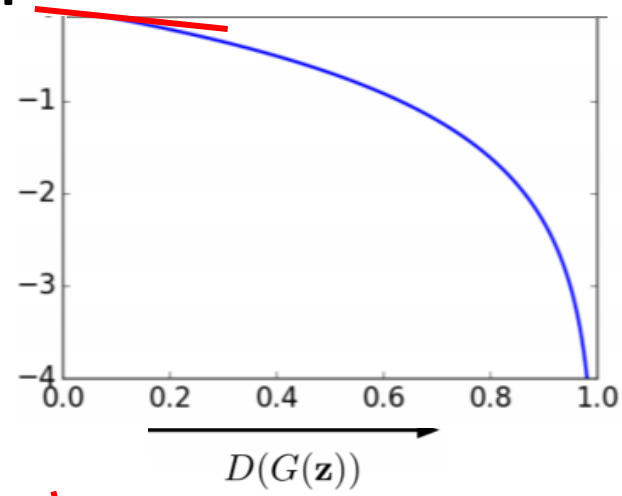
Training with real images

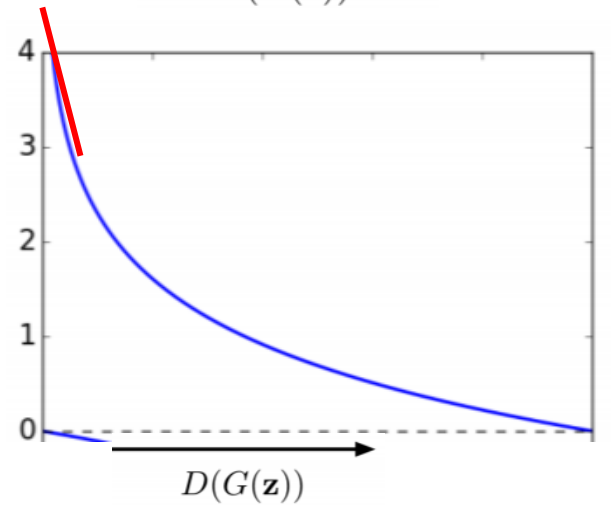
Training with fake images



Objective Function of GAN

- **A better objective function for 'Generator'**
  - **Original:** $\min E_{Z\sim p_Z(z)}[log(1-D(G(z))]$
    - One problem with this is saturation
    - Here, if the generated sample is really bad (at the beginning), the discriminator's prediction is close to 0, and the gradient of the objective function to $D(G(z))$ is very small (≈ 0) so that the learning is very slow (saturated).
  - **Modified:** $\min E_{Z\sim p_Z(z)}[-logD(G(z))]$
    - High gradient at $D\big(G(z)\big)=0$, in which fast learning is required for 'Generator'.





Latent Vector Arithmetic

- Interpolating $z_1 \sim z_2$

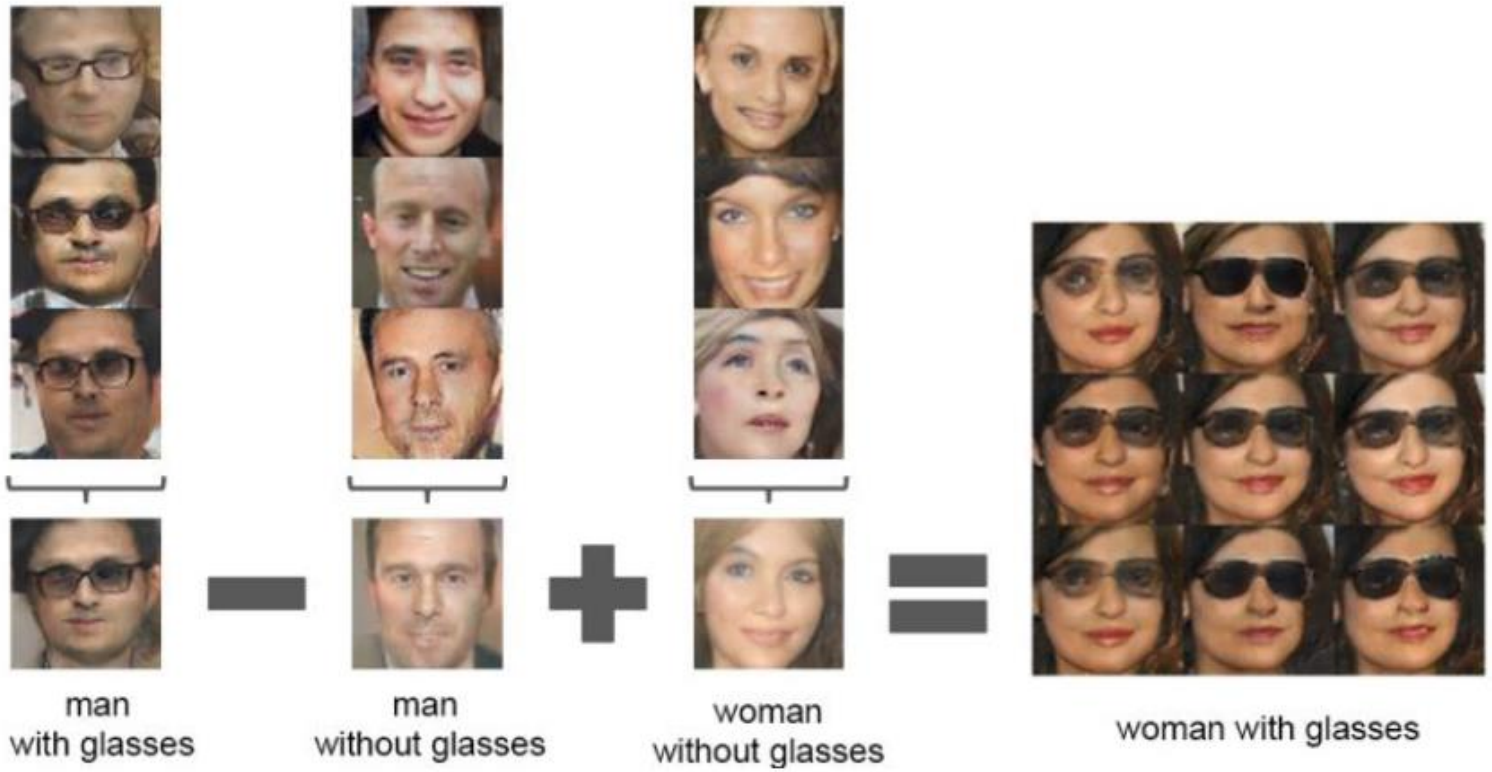
Interpolating between random points in laten space

Radford et al, ICLR 2016

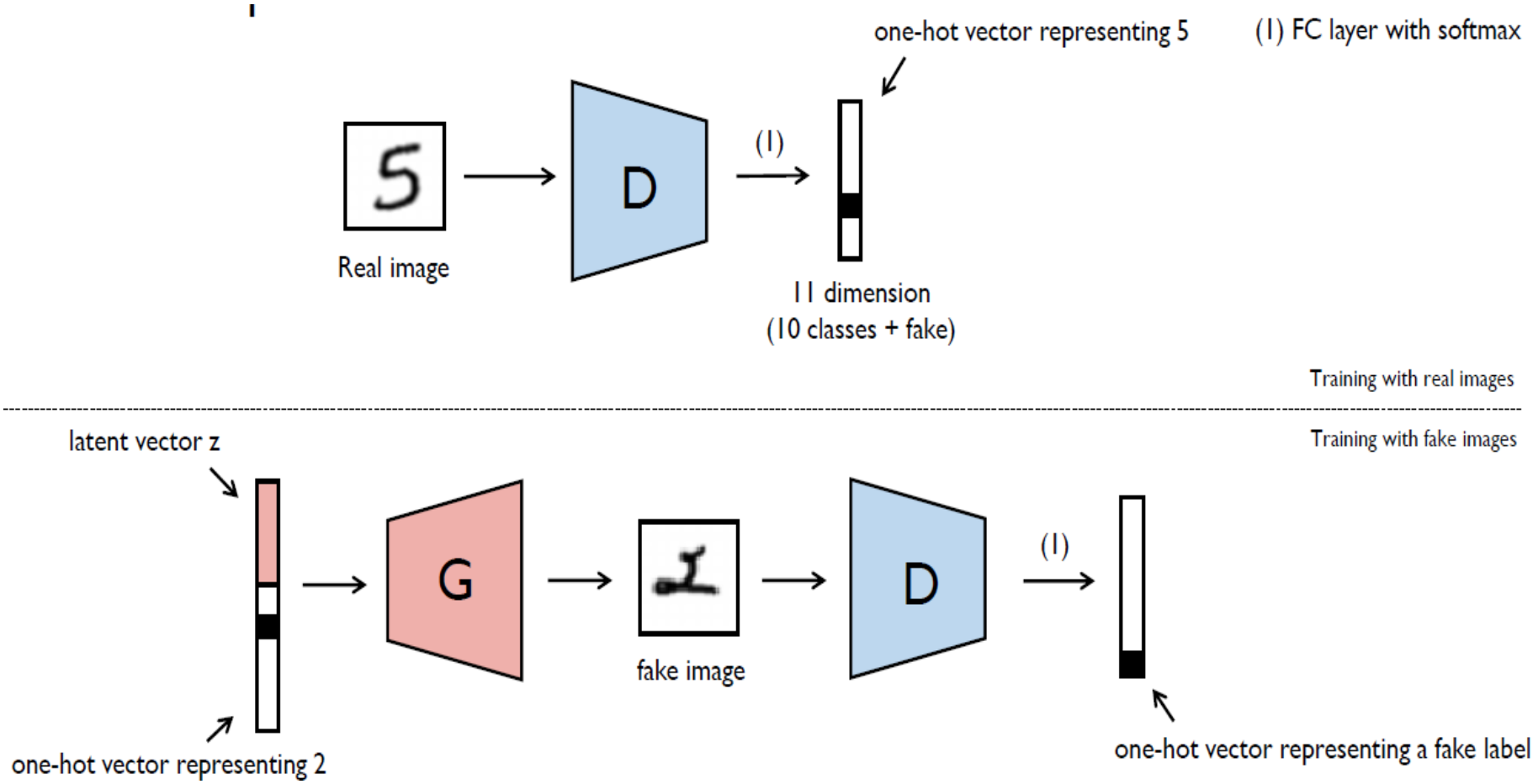
z_1 ⟶ z_2

Latent Vector Arithmetic

- **Latent vector domain operation**
  - $z_1 - z_2 + z_3 = z_4$ will result in fake images $G(z_4)$ (woman with glasses)

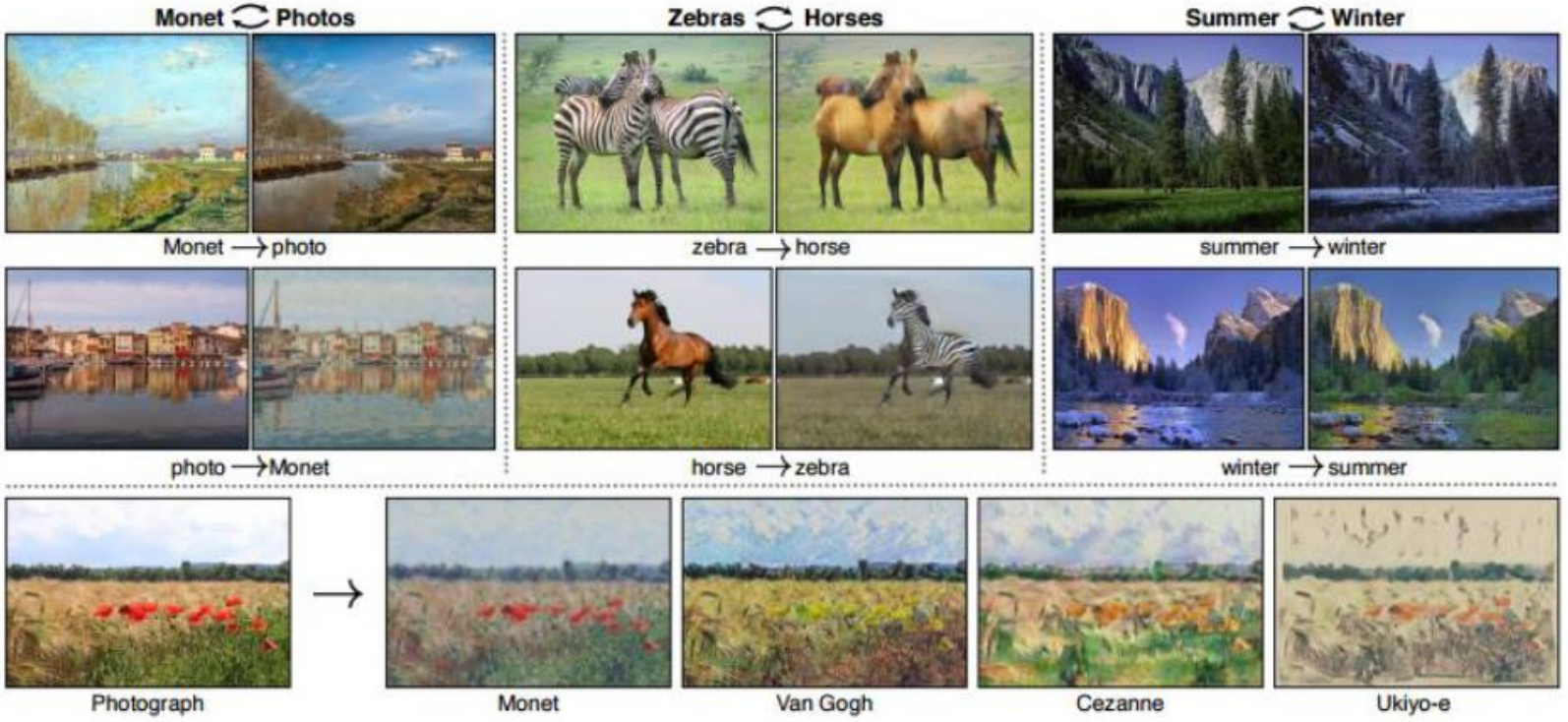


Semi-supervised GAN



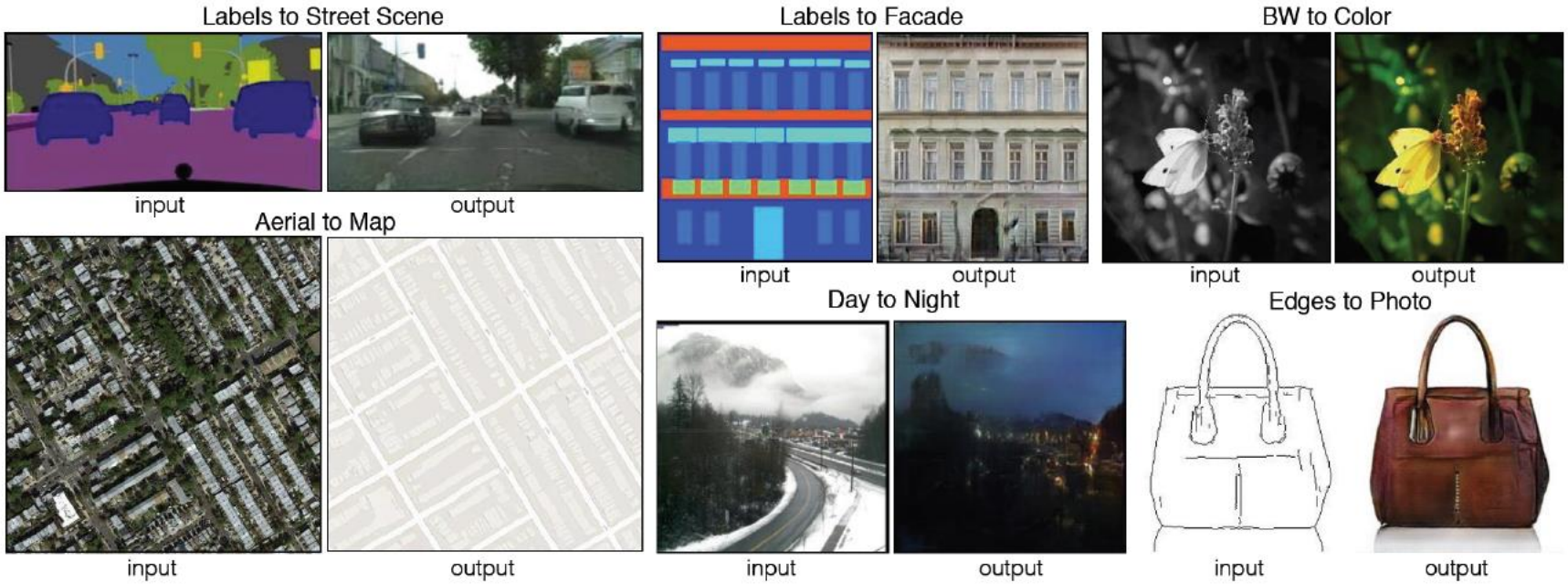
Applications of GAN

- **CycleGAN: Unpaired image-to-image translation**
  - Presents a GAN model that transfer from a source domain A to a target domain B in the absence of paired examples



Applications of GAN

- Image-to-image translation



Applications of GAN

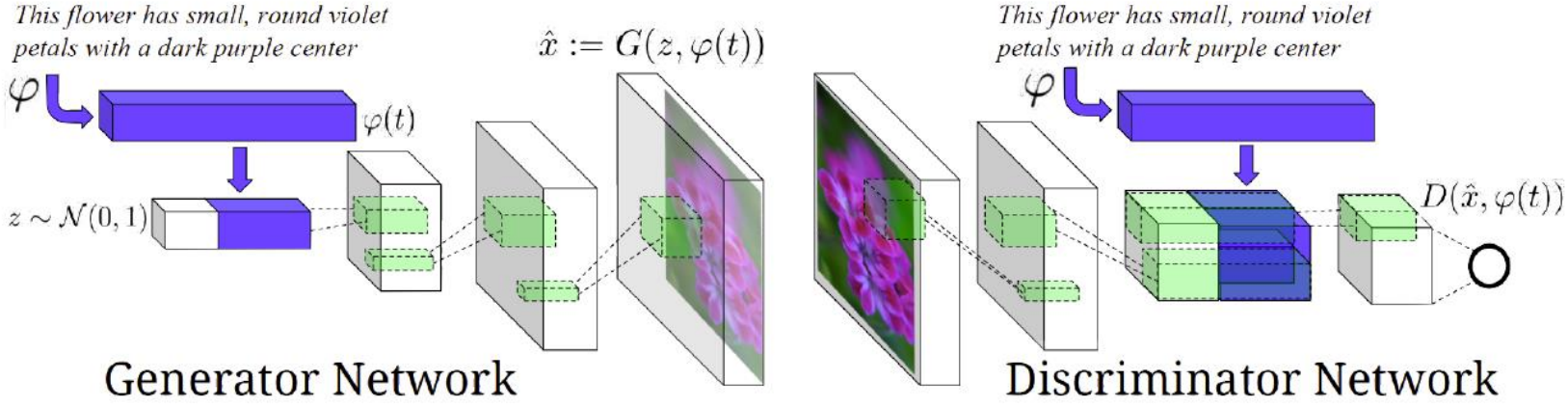
- **Text-to-Image Synthesis**
  - Given a text description, generate images closely associated.

this small bird has a pink breast and crown, and black primaries and secondaries.

this magnificent fellow is almost all black with a red crest, and white cheek patch.

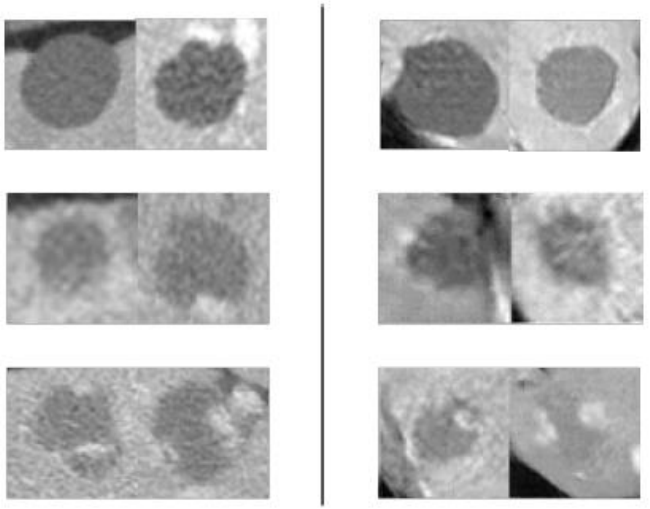
the flower has petals that are bright pinkish purple with white stigma

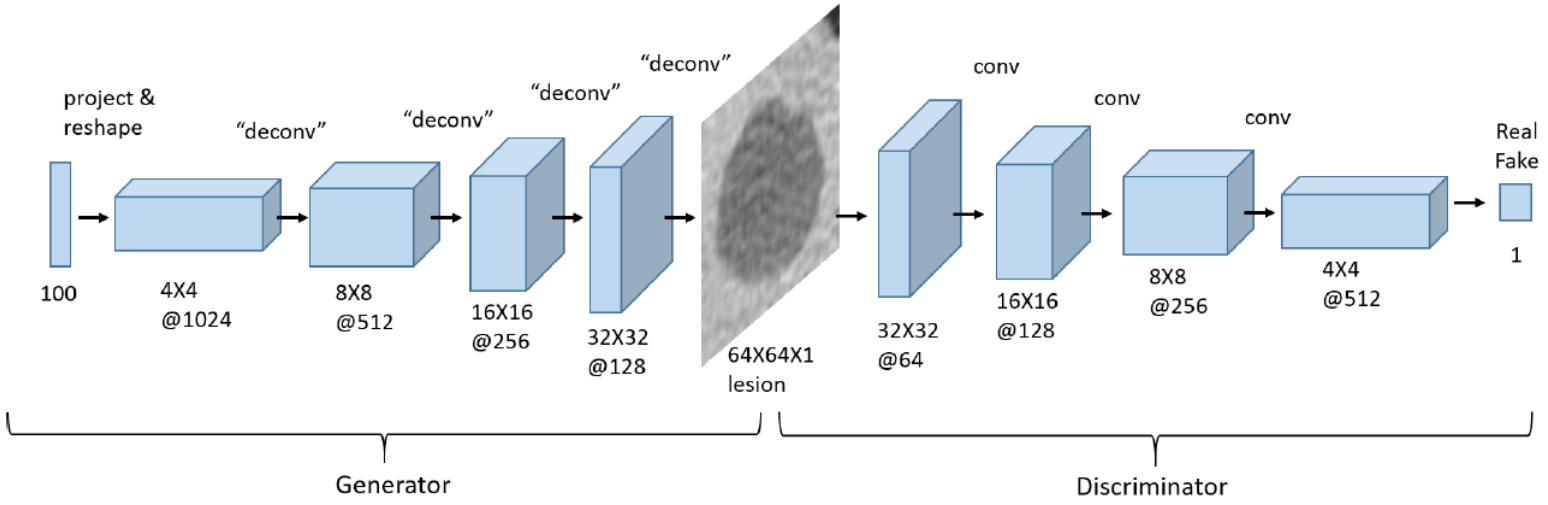
this white and yellow flower have thin white petals and a round yellow stamen



Applications of GAN

- **Data augmentation**
  - Effective training of neural networks requires much data.
  - Generates synthetic data (e.g., medical images) using GAN.





GAN Programming: MNIST

```
# Generator Function Definition

def generator( z) :
        gw1 = tf.get_variable(name = "w1",  shape = [128, 256],
                        initializer= tf.random_normal_initializer(mean=0.0, stddev = 0.01))
        gb1 = tf.get_variable(name = "b1",  shape = [256],
                        initializer = tf.random_normal_initializer(mean=0.0, stddev = 0.01))

        gw2 = tf.get_variable(name = "w2",  shape = [256, 784],
                        initializer= tf.random_normal_initializer(mean=0.0, stddev = 0.01))
        gb2 = tf.get_variable(name = "b2",  shape = [784],
                        initializer = tf.random_normal_initializer(mean=0.0, stddev = 0.01))
       hidden = tf.nn.relu( tf.matmul(z , gw1) + gb1 )
       output = tf.nn.sigmoid( tf.matmul(hidden, gw2) + gb2 )

      return output   # [784] fake image
```

GAN Programming: MNIST

```
# Discriminator Function Definition
def discriminator( x) :
        dw1 = tf.get_variable(name = "w1",  shape = [784, 256],
                        initializer = tf.random_normal_initializer(0.0, 0.01) )
        db1 = tf.get_variable(name = "b1",  shape = [256],
                        initializer = tf.random_normal_initializer(0.0, 0.01) )
        dw2 = tf.get_variable(name = "w2",  shape = [256, 1],
                        initializer = tf.random_normal_initializer(0.0, 0.01) )
        db2 = tf.get_variable(name = "b2",  shape = [1],
                        initializer = tf.random_normal_initializer(0.0, 0.01) )
        hidden = tf.nn.relu( tf.matmul(x , dw1) + db1 )
        output = tf.nn.sigmoid( tf.matmul(hidden, dw2)  + db2 )   #fake=0, real=1
        return output
```

```
# Random Noise 'z' Generation Function
def random_noise(batch_size) :
      return np.random.normal(size=[batch_size , 128])
```

GAN Programming: MNIST

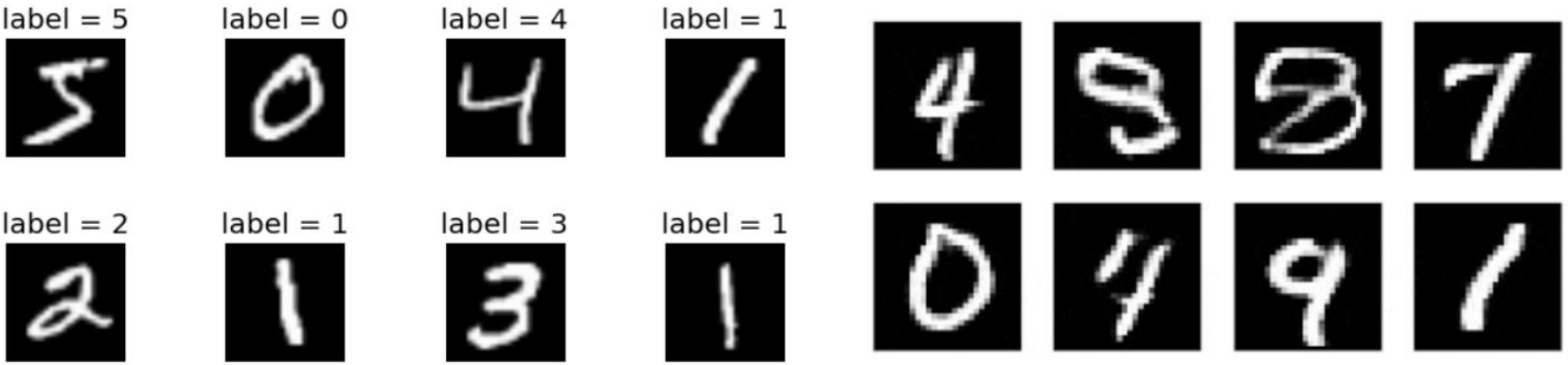
```
import tensorflow as tf
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/")
train_x = mnist.train.images
total_epochs = 100
batch_size = 100
learning_rate = 0.0002

X = tf.placeholder(tf.float32, [None, 784]) # not need Y (unsupervised learning)
Z = tf.placeholder(tf.float32, [None, 128])
fake_x = generator(Z)
result_of_fake = discriminator(fake_x)
result_of_real = discriminator(X)

g_loss = tf.reduce_mean(-tf.log(result_of_fake))  # generator loss function: minimize
d_loss = tf.reduce_mean( tf.log(result_of_real) + tf.log(1 - result_of_fake) )
                                                  # discriminator loss function: maximize
optimizer = tf.train.AdamOptimizer(learning_rate)
g_train = optimizer.minimize(g_loss)
d_train = optimizer.minimize(-d_loss) # with '-' convert to minimize
```

GAN Programming: MNIST

```
with tf.Session as sess :
        sess.run(tf.global_variables_initializer())
        total_batchs = int(train_x.shape[0] / batch_size)
        for epoch in range(total_epochs) :
            for batch in range(total_batchs) :
                batch_x = train_x[batch * batch_size : (batch+1) * batch_size]
                batch_y = train_y[batch * batch_size : (batch+1) * batch_size]
                noise = random_noise(batch_size)
                sess.run(g_train , feed_dict = {Z : noise})
                sess.run(d_train, feed_dict = {X : batch_x , Z : noise})
                gl, dl = sess.run([g_loss, d_loss], feed_dict = {X : batch_x , Z : noise})
```



Recurrent Neural Network

Inha University

Prof. Sang-Jo Yoo

sjyoo@inha.ac.kr

Motivations

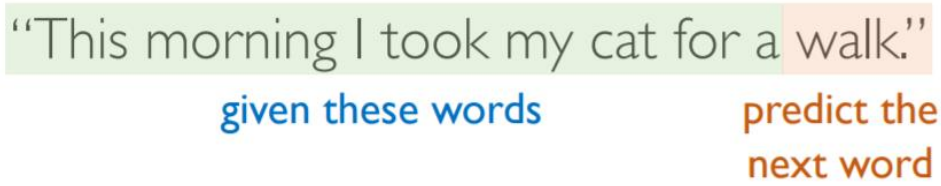
RNN Model Sequences

Vanilla RNN Architecture

Long Short Term Memory (LSTM)

Motivations

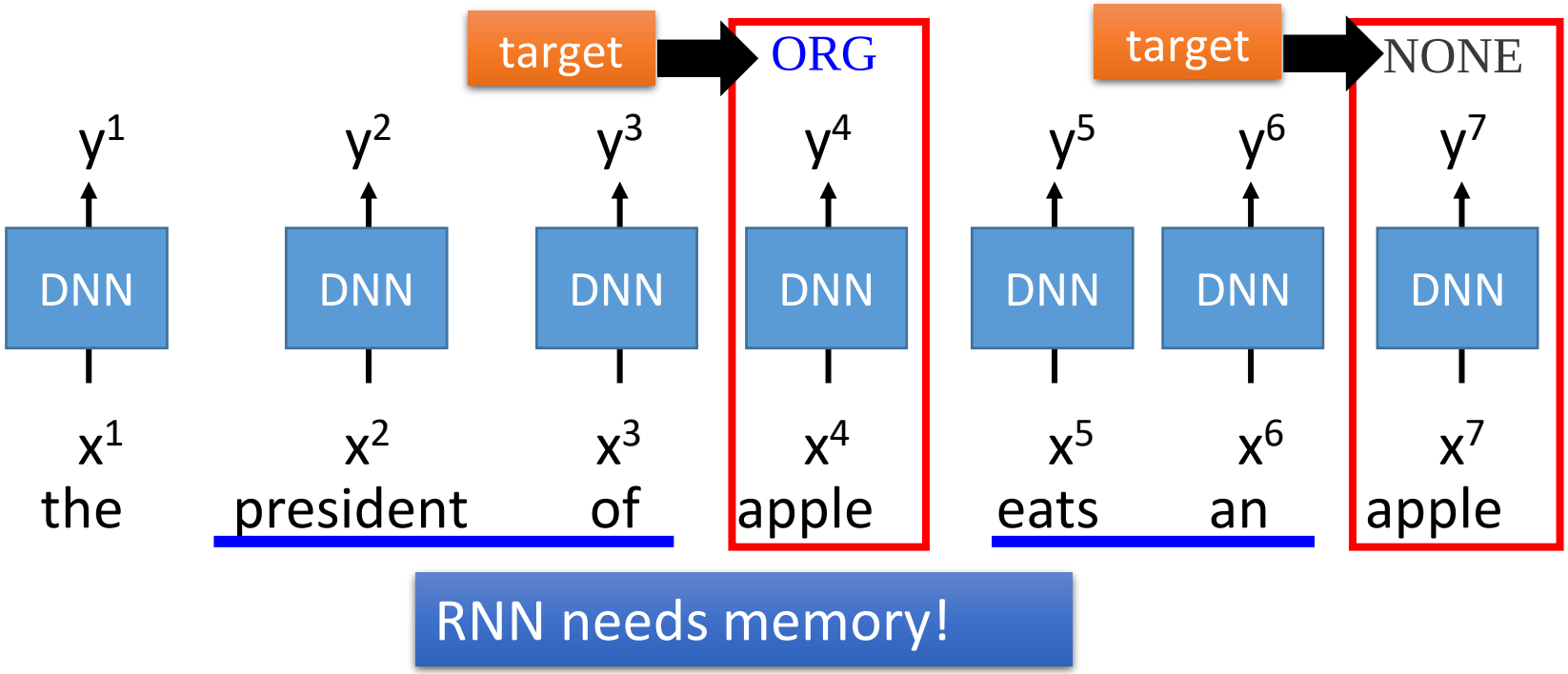
- **Motivations**
  - Not all problems can be converted into one with fixed length inputs and outputs.
  - Problems such as Speech Recognition or Time-series Prediction require a system to store and use context information.



- **Recurrent Neural Network (RNN) takes the previous output or hidden states as inputs.**
  - The composite input at time *t* has some historical information about the happenings at time T < *t*.

Motivations

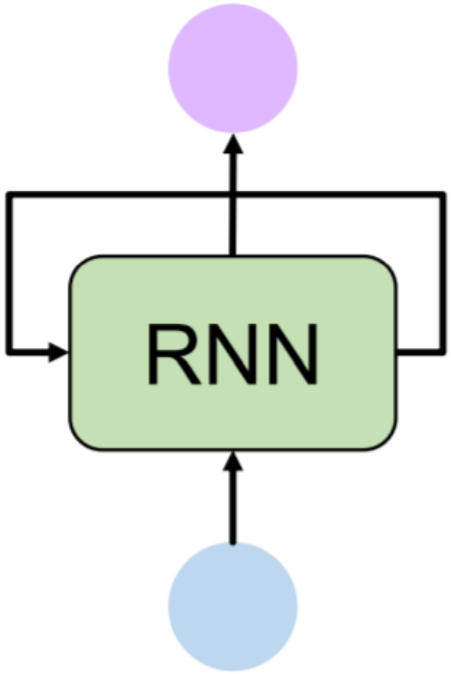
- **Name Entity Recognition**
  - Detecting named entities like name of people, locations, organization, etc. in a sentence.



Sequence Modeling: design criteria

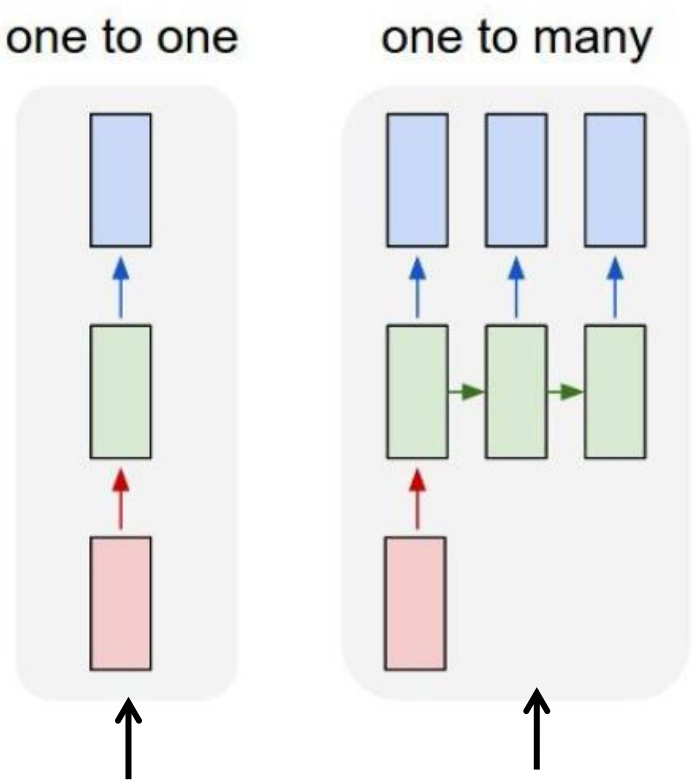
- **To model sequences, we need to:**

1. Handle variable-length sequences
2. Track long-term dependencies
3. Maintain information about order
4. Share parameters across the sequence



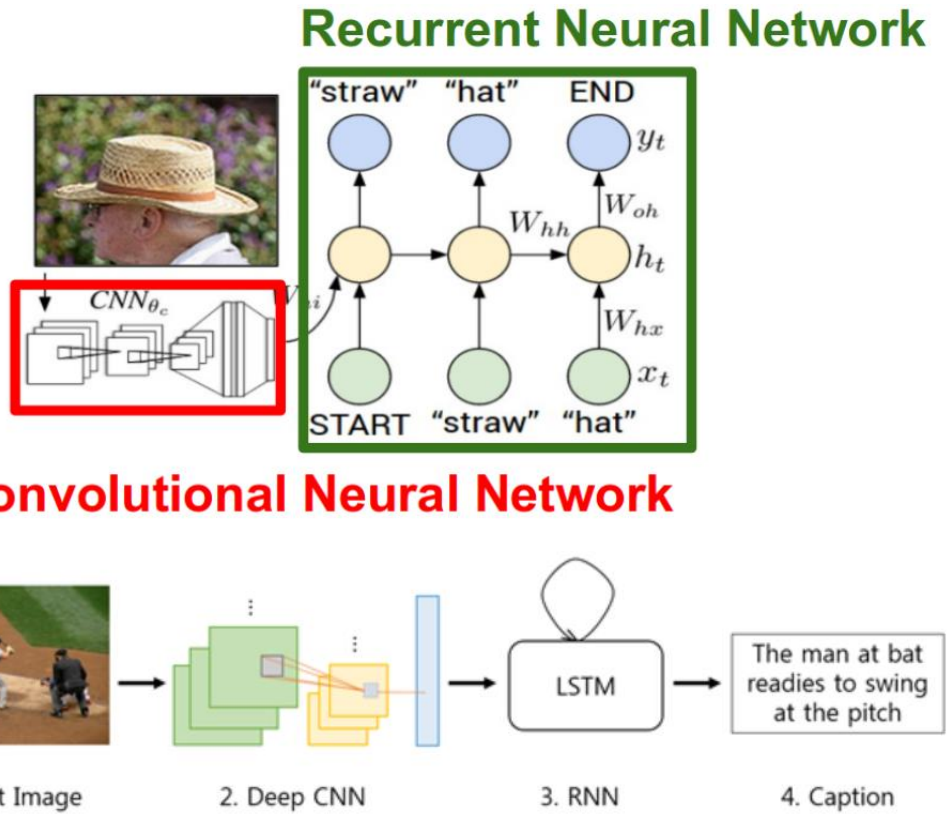
- **Today: Recurrent Neural Networks (RNNs) as an approach to sequence modeling problems**

RNN Model Sequences



Vanilla
Neural Network

e.g. Image Captioning
Image → sequence of words



RNN Model Sequences

Image Captioning: Example Results

A cat sitting on a suitcase on the floor

A cat is sitting on a tree branch

A dog is running in the grass with a frisbee

A white teddy bear sitting in the grass

Two people walking on the beach with surfboards

A tennis player in action on the court

Two giraffes standing in a grassy field

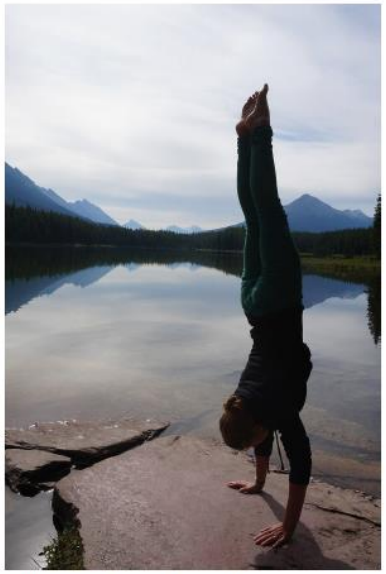
A man riding a dirt bike on a dirt track

RNN Model Sequences

Image Captioning: Failure Cases

A woman is holding a cat in her hand

A person holding a computer mouse on a desk

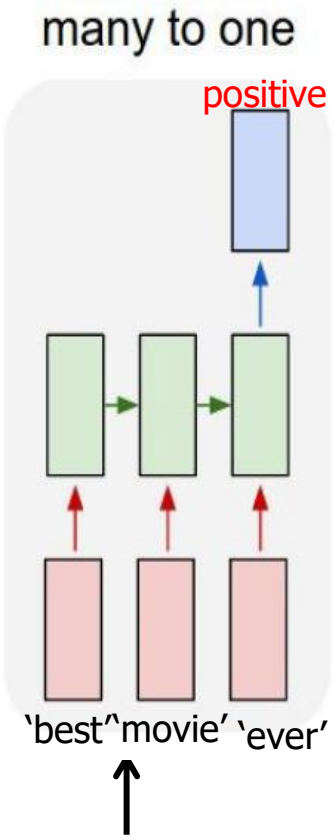


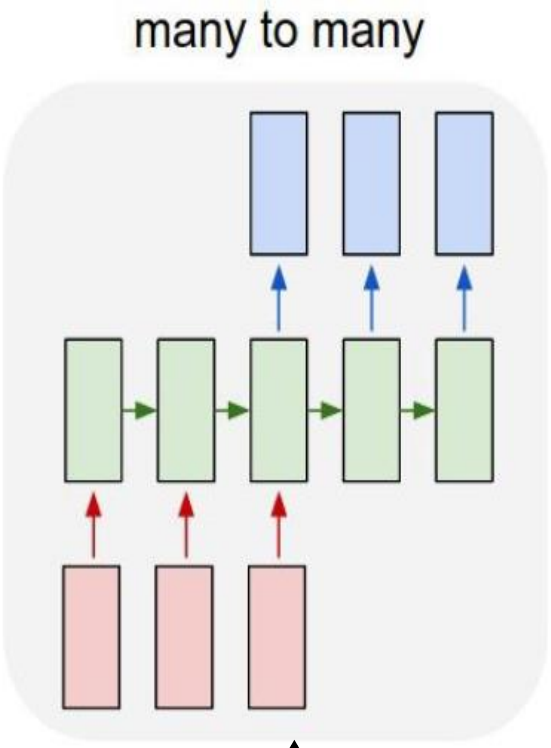
A woman standing on a beach holding a surfboard

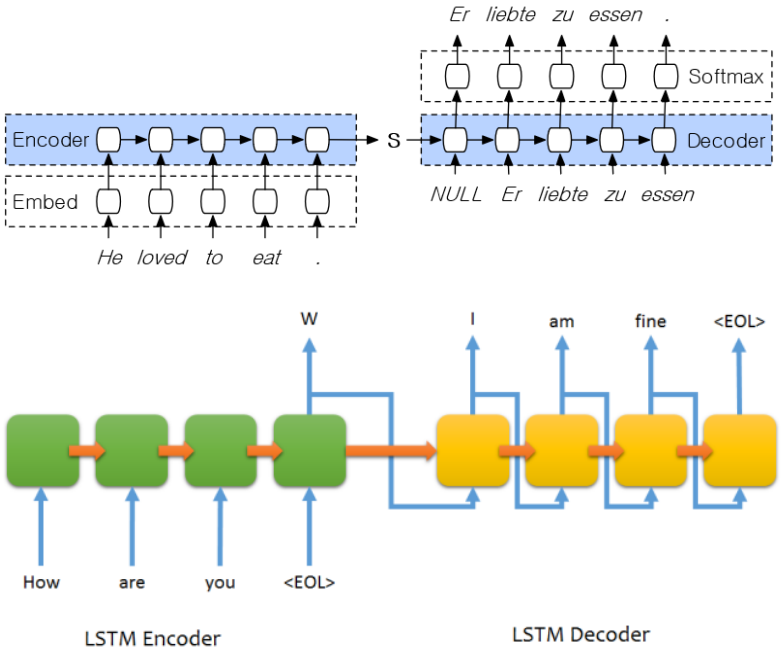
A bird is perched on a tree branch

A man in a baseball uniform throwing a ball

RNN Model Sequences



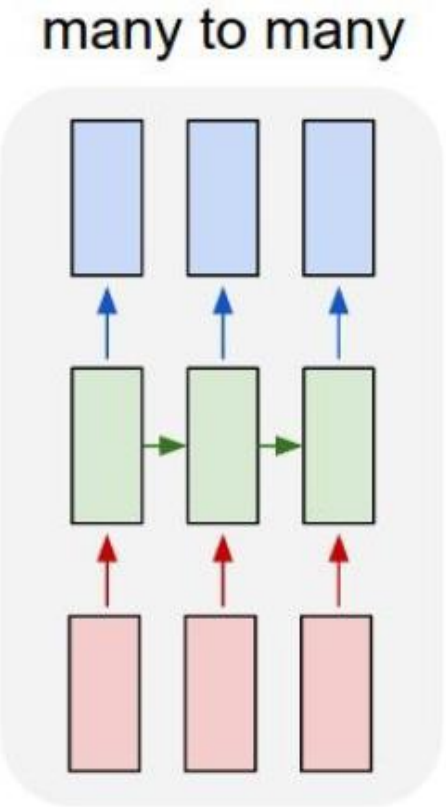


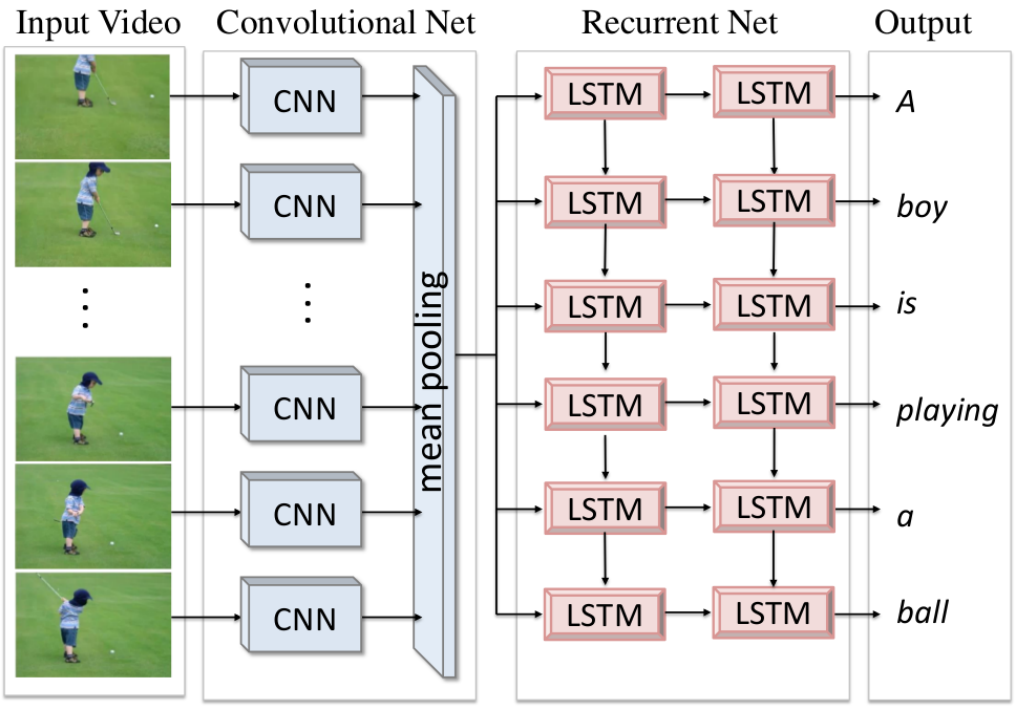


e.g. Sentiment Classification
Sequence of words → sentiment

e.g. Machine Translation
Sequence of words → Sequence of words

RNN Model Sequences





e.g. Video Classification on Frame Level

Recurrent Neural Network

- We can process a sequence of vectors **x** by applying a **recurrence formula** at every time step:
  - Notice: the same function and the same set of parameters are used at every time step.

$$h_t = f_W(h_{t-1}, x_t)$$

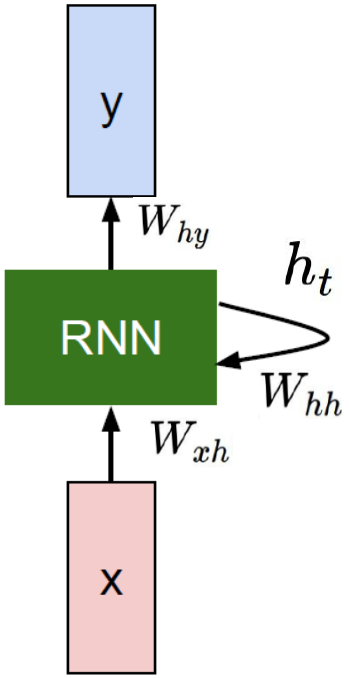
new state — old state — input vector at some time step

some function with parameters W

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

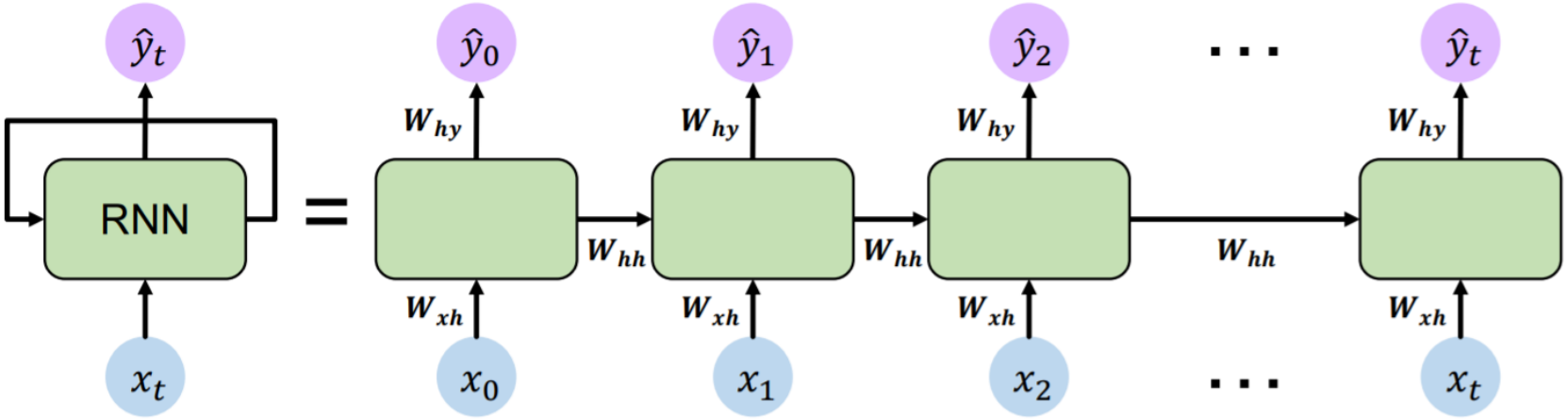
$$y_t = W_{hy}h_t$$

(Vanilla) RNN



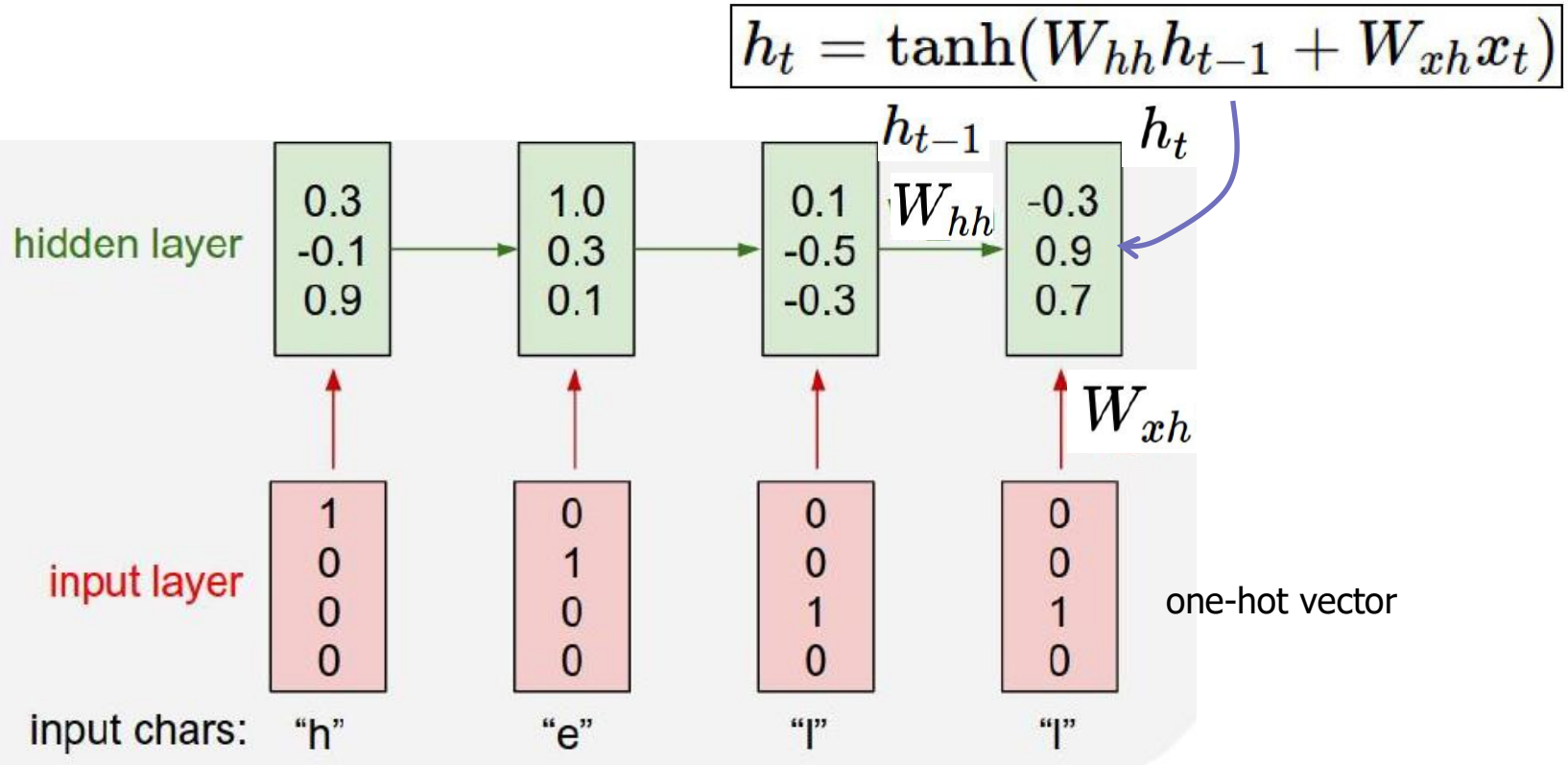
Recurrent Neural Network

- **RNN computational graph**
  - Re-use the same weight matrices at every time step



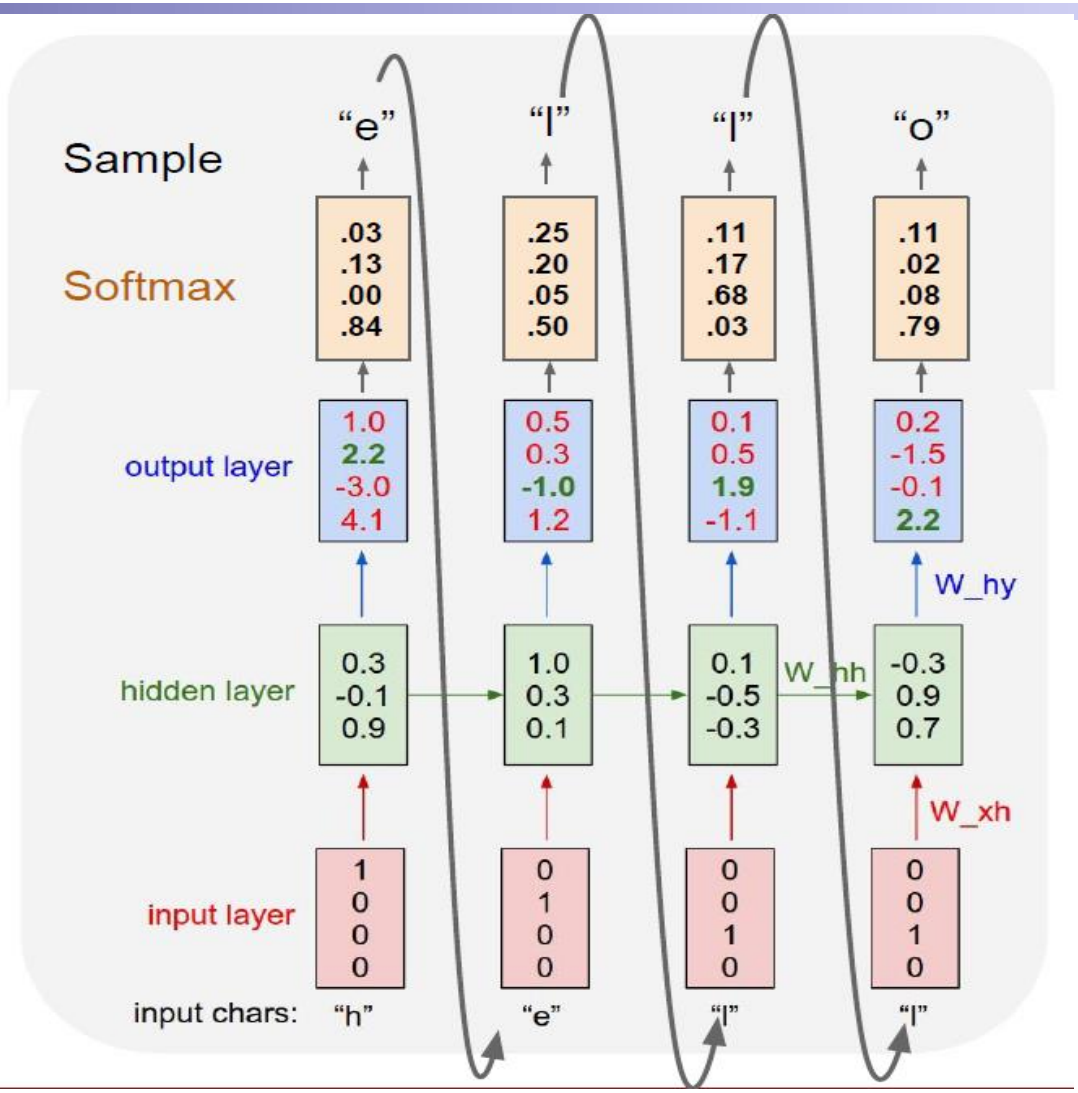
Recurrent Neural Network

- **Example: Character-level language model**
  - Vocabulary: [h,e,l,o]
  - Example training sequence: “hello”



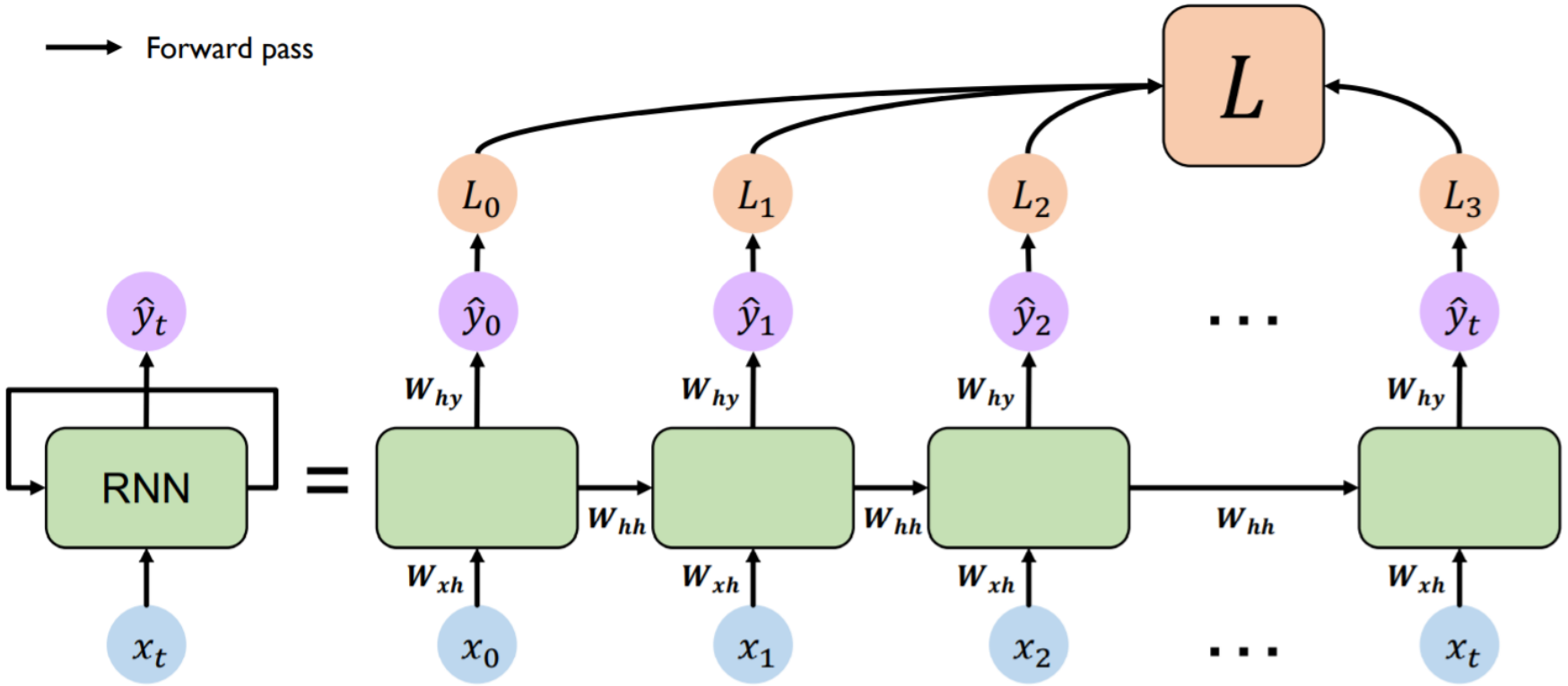
Recurrent Neural Network

- At test-time, sample characters one at a time, feed back to model.
- Training an RNN: backpropagation through time



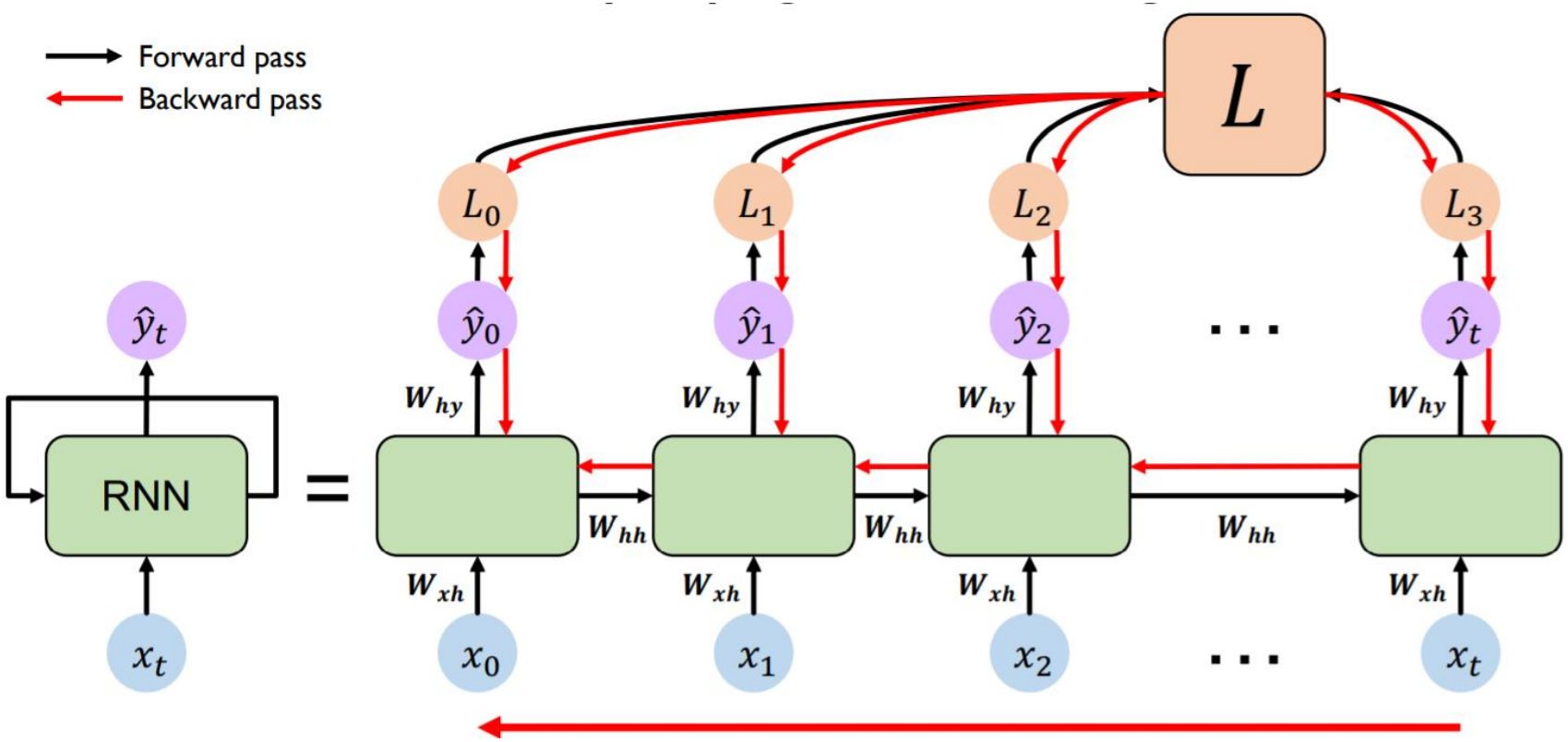
RNN Loss Function

- **Loss function: forward pass**



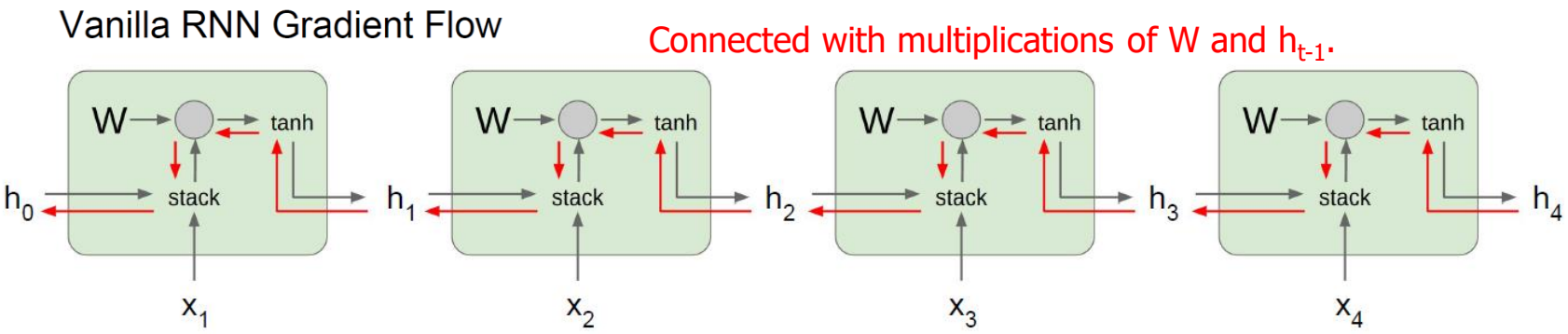
RNN Loss Function

- **Backpropagation for parameter updating**



Gradient Vanishing Problem

- **During the backpropagation, we have the gradient vanishing problem.**
  - The vanishing (and exploding) gradient problem is caused by the repeated use of the recurrent weight matrix in RNN.
  - Computing gradient of h_0 involves many factors of W and repeated tanh.
    - If either of these factors is smaller than 1, then the gradients may vanish in time; if larger than 1, then exploding might happen. For example, the *tanh* derivative is <1 for all inputs except 0; *sigmoid* is even worse and is always ≤0.25 .

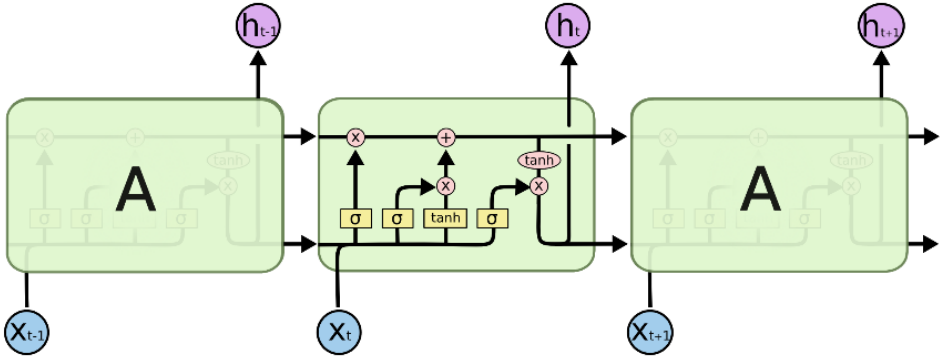


Long Short Term Memory (LSTM)

Long Short Term Memory

- Developed by Sepp Hochreiter and Jürgen Schmidhuber in 1997.





- LSTMs were developed to deal with the exploding and vanishing gradient problems that can be encountered when training traditional RNNs.
- is composed of a **cell** (the memory part of the LSTM unit) and three "regulators", usually called **gates**, of the flow of information inside the LSTM unit: an **input gate**, an **output gate** and a **forget gate**

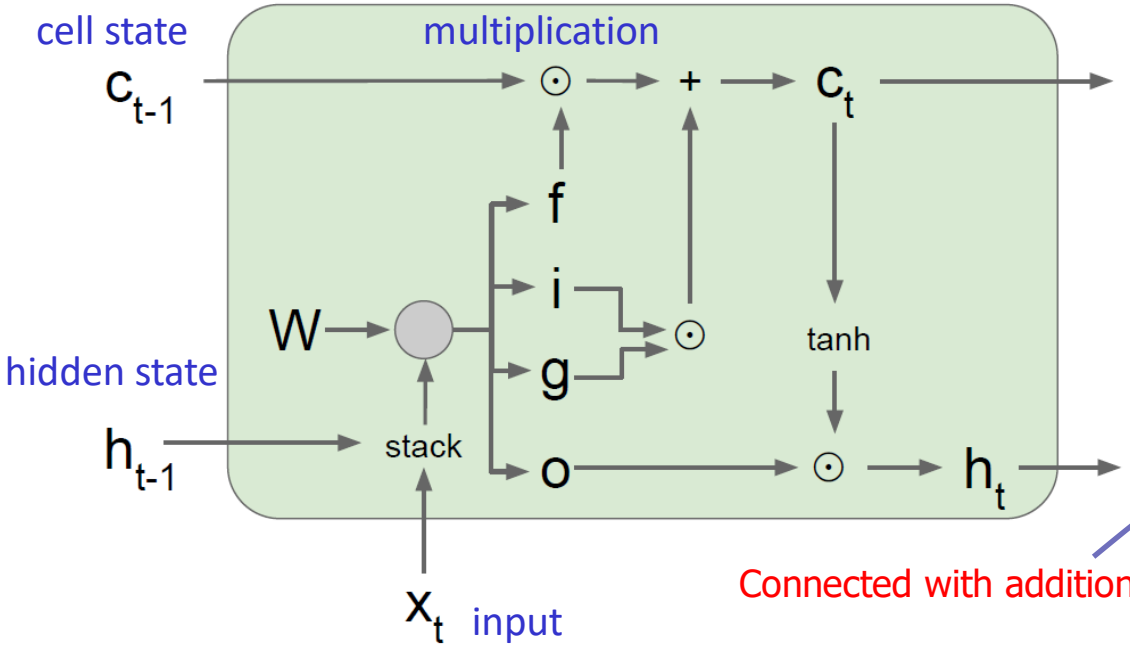
Long Short Term Memory (LSTM)

$$h_t = \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

Vanilla RNN

Connected with multiplications of W and h_{t-1}.

f: Forget gate, Whether to erase cell
i: Input gate, whether to write to cell
g: Gate gate (?), How much to write to cell
o: Output gate, How much to reveal cell



LSTM

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

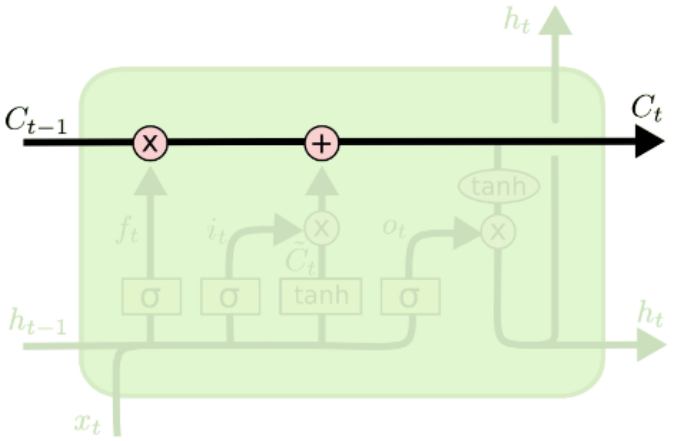
$$h_t = o \odot \tanh(c_t)$$

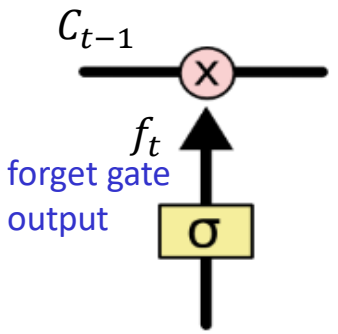
Connected with additions

Long Short Term Memory (LSTM)

Step-by-Step LSTM Walk Through

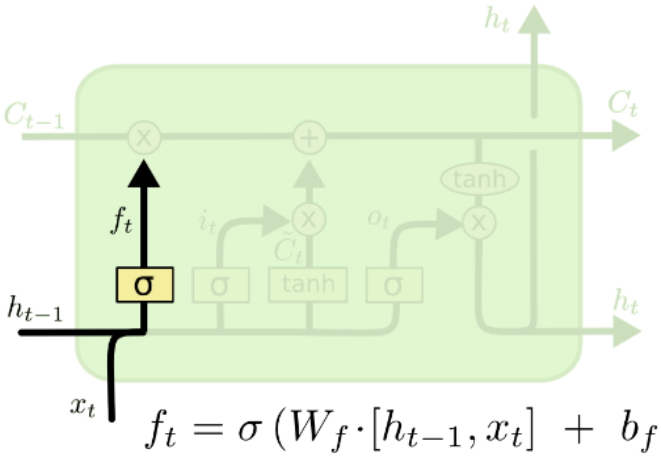
- The key to LSTMs is the **cell state**, the horizontal line running through the top of the diagram.
  - is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged. (Simple addition can avoid gradient vanishing problem)
  - Cell state has the ability to remove or add information to the cell state, carefully regulated by structures called gates.
  - The sigmoid layer (forget gate) outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means "let nothing through," while a value of one means "let everything through!".





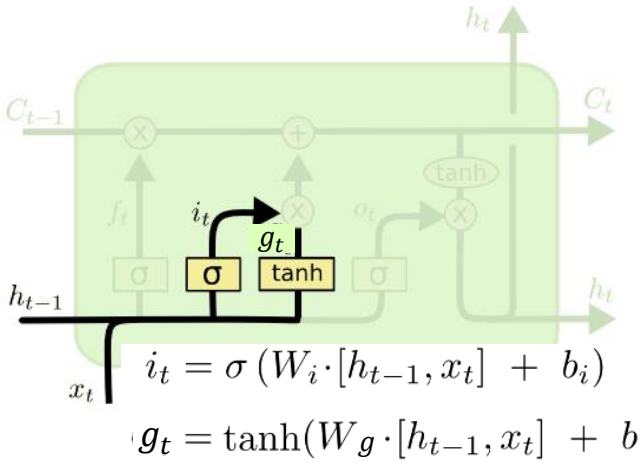
Long Short Term Memory (LSTM)

- The first step in our LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the "**forget gate layer**."
  - Example of a language model trying to predict the next word based on all the previous ones, when we see a new subject, we want to forget the gender of the old subject (f_t will be 0).



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

- The next step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the "**input gate layer**" decides which values of g_t will be updated how much.
  - Next, a *tanh* layer (**gate gate**?) creates a vector of new candidate values, g_t that could be added to the state. In the next step, we'll combine these two to create an update to the state.

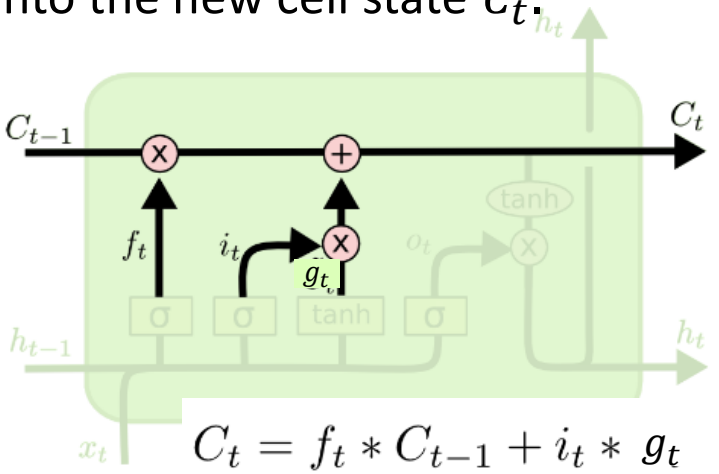


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$g_t = \tanh\left(W_g \cdot [h_{t-1}, x_t] \; + \; b_g\right)$$

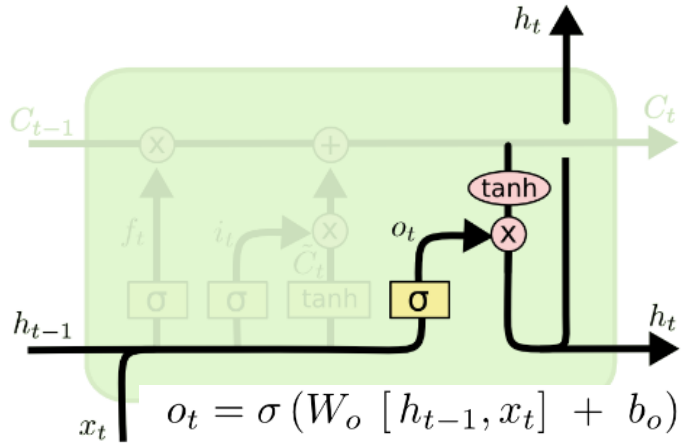
Long Short Term Memory (LSTM)

- It's now time to update the old cell state, C_{t-1}, into the new cell state C_t.
  - We multiply the old state by f_t, forgetting the things. Then we add $i_t \times g_t$, this is the new candidate values, scaled by how much we decided to update each value of g_t.



$$C_t = f_t * C_{t-1} + i_t * g_t$$

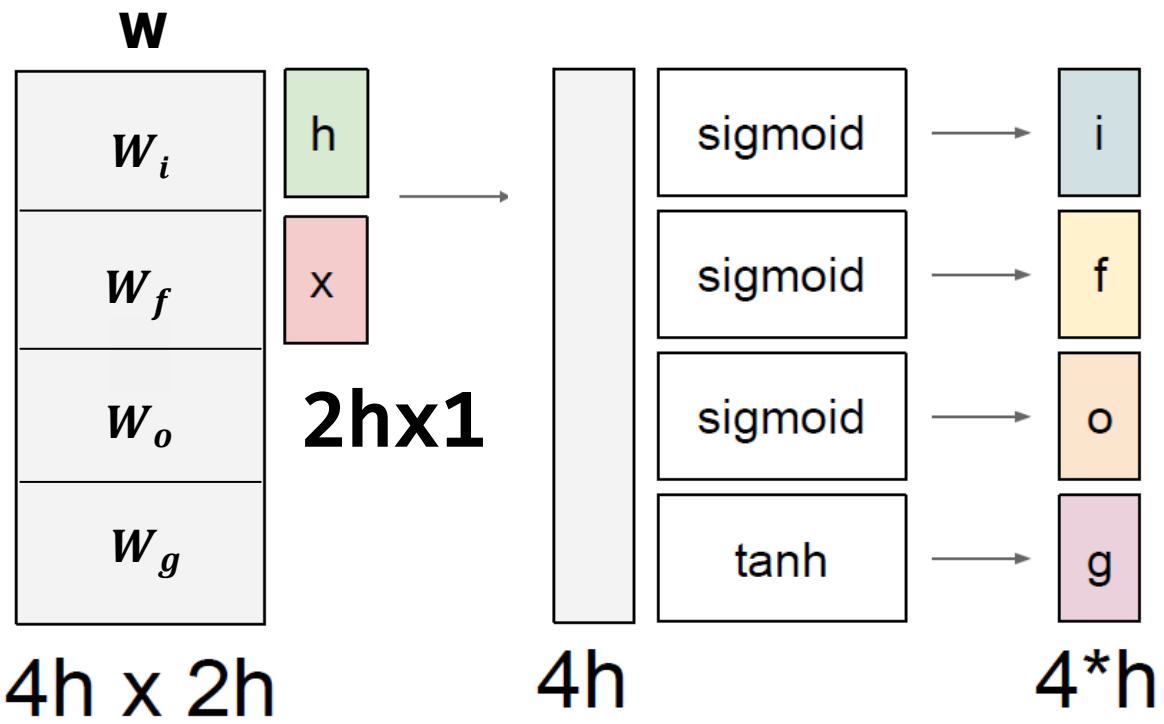
- Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version (by output gate o_t).
- First, we run a sigmoid layer ("**output gate layer o_t**") which decides what parts of the cell state we're going to output.
- Then, we put the cell state through *tanh* (to push the values to be between −1 and 1) and multiply it by the output of the output gate.



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

Long Short Term Memory (LSTM)



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

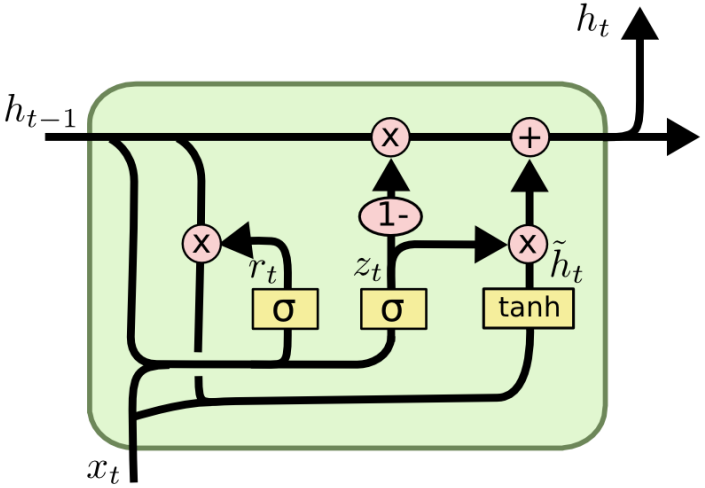
$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Gated Recurrent Unit (GRU)

Gated Recurrent Unit (GRU)

- Introduced in 2014 by Kyunghyun Cho et al
- Similar with LSTM but with only two gates and less parameters.
- The "update gate z_t" determines how much of previous memory to be kept.
- The "reset gate r_t" determines how to combine the new input with the previous memory.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

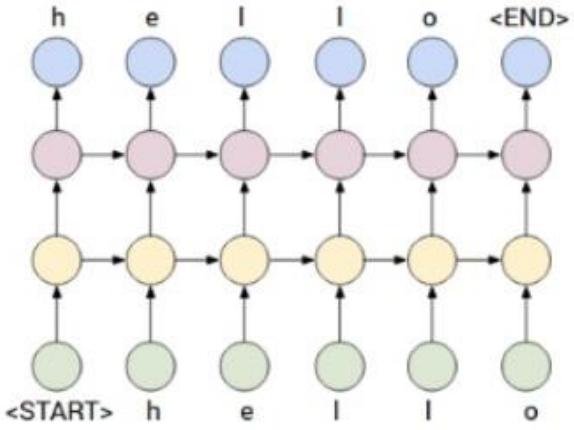
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

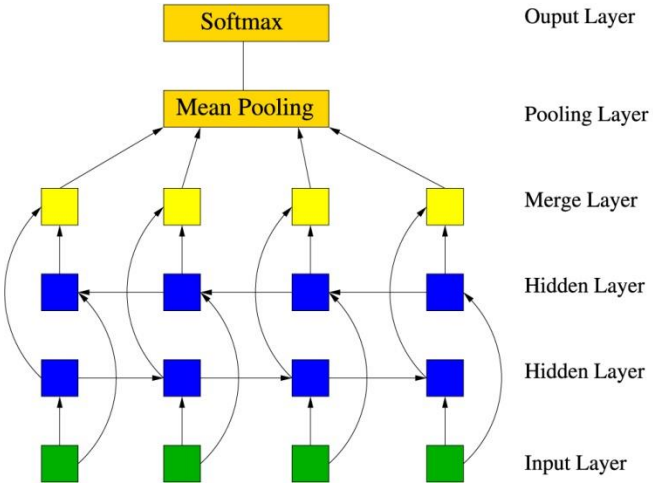
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

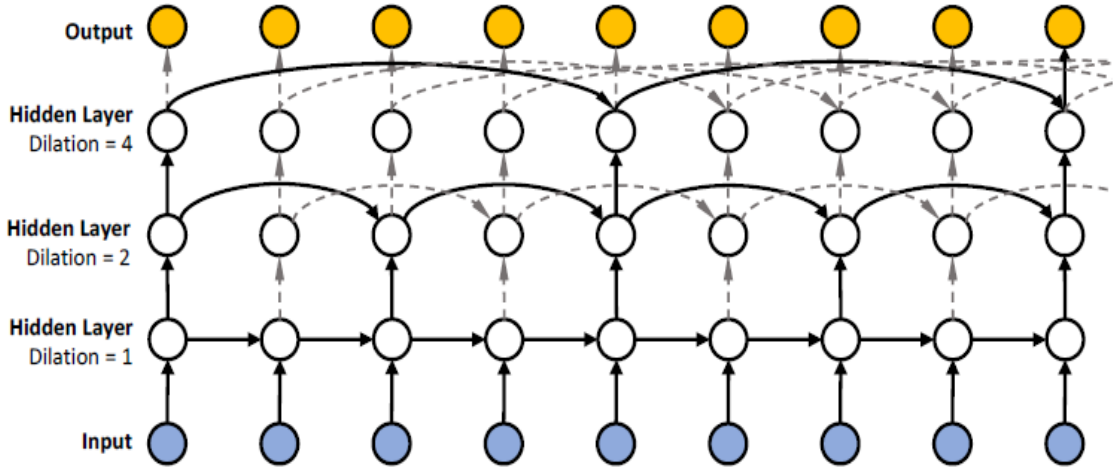
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

RNN Architectures

- **Possible RNN architectures**







Reinforcement Learning

Inha University

Prof. Sang-Jo Yoo

sjyoo@inha.ac.kr

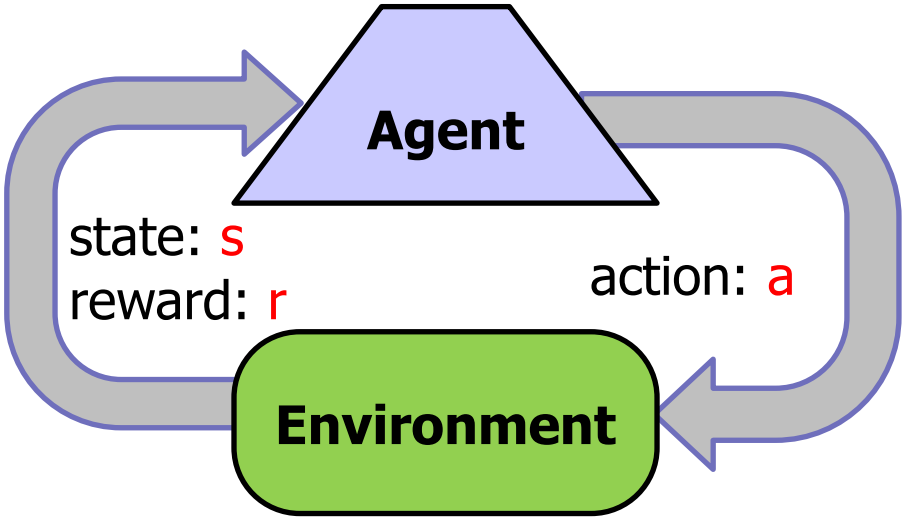
Reinforcement Learning Model

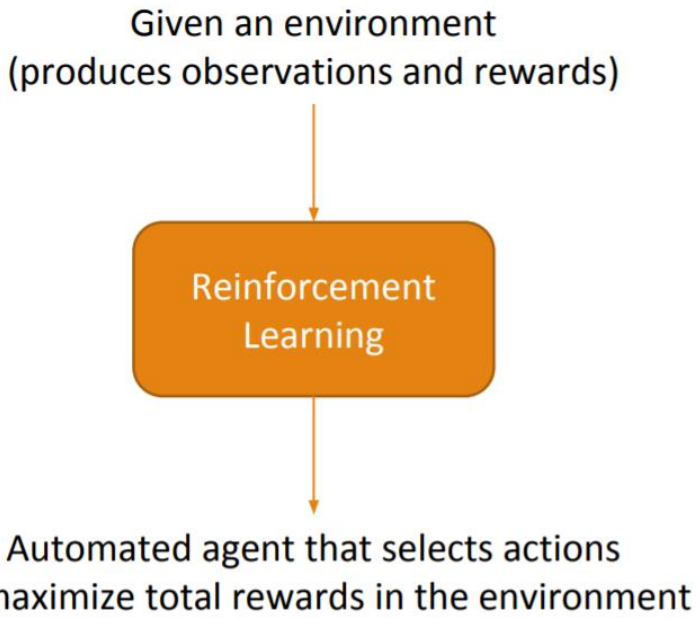
Q-Learning Algorithm

Deep Reinforcement Learning (DQN)

Ref: https://www.ics.uci.edu/~xhx/courses/CS273P/

Reinforcement Learning Model





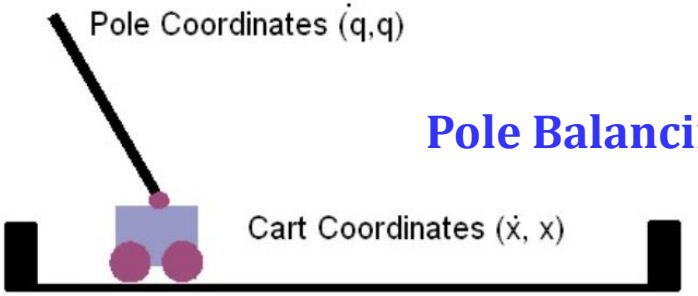
- **Basic idea:**
  - Receiving feedback in the form of rewards.
  - Agent's utility is defined by the reward function.
  - Must (learn to) act so as to maximize expected rewards.
    - May be better to sacrifice short term reward for long term benefit .
  - All learning is based on observed samples of outcomes.

Reinforcement Learning Model

- **More general than supervised learning**
  - Reinforcement learning differs from standard supervised learning in that correct input/output pairs are never presented, nor sub-optimal actions explicitly corrected.
  - Learn from interaction with environment to achieve a goal.
    - It receives some evaluation of its action (reinforcement), but is not told of which action is the correct one to achieve its goal.
- **The basic reinforcement learning model consists of:**
  1. A set of environment states ***S***
  2. A set of actions ***A***
  3. Rules of transitioning between states
  4. Rules that determine the scalar immediate reward of a transition
  5. Rules that describe what the agent observes.

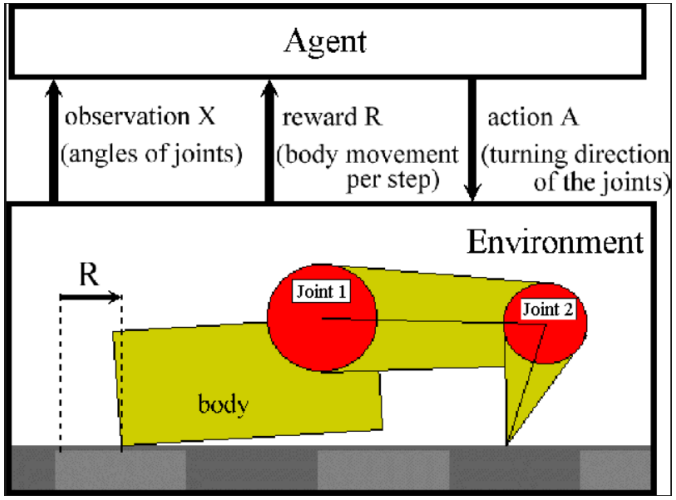
Reinforcement Learning Examples

Pole Balancing

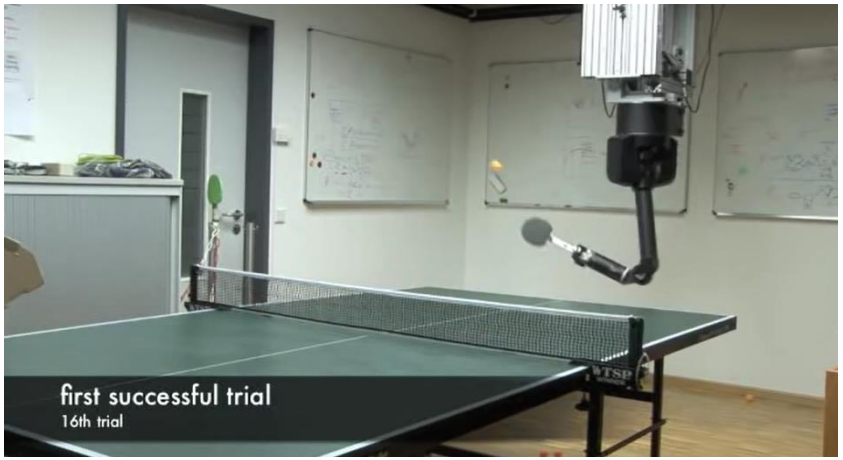


Playing Game

Move To The Front



Robot Table Tennis



Markov Property

Markov property

- "The future is independent of the past, given the present."

> **Definition**: A state S_t is Markov if and only if
> $$P[S_{t+1}|S_t] = P[S_{t+1}|S_1, S_2, \cdots, S_t]$$

- Once the state is known, the history may be thrown away.

State transition matrix

- For a Markov state s and successor state s', the state transition probability is defined by

$$P_{ss'} = P[S_{t+1} = s'|S_t = s]$$

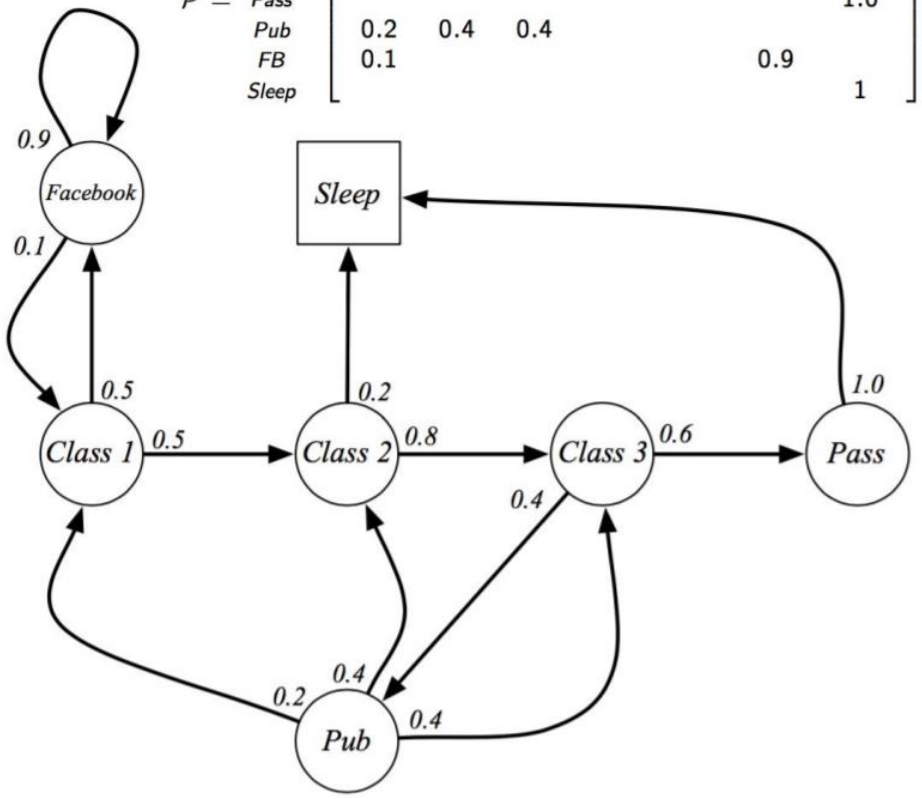
$$\mathcal{P} = \text{from} \begin{bmatrix} & & \text{to} \\ \mathcal{P}_{11} & \cdots & \mathcal{P}_{1n} \\ \vdots & & \\ \mathcal{P}_{n1} & \cdots & \mathcal{P}_{nn} \end{bmatrix}$$

Markov Process

- **A Markov process is a memoryless random process, i.e. a sequence of random states $S_1, S_2, \cdots$ with Markov property.**
  - A random process is a time-varying function that assigns the outcome of a random experiment to each time instant.

Definition: : A Markov Process is a tuple $\langle S, P \rangle$
- S is a set of states,
- P is a state transition probability matrix

$$\mathcal{P} = \begin{array}{c} \\ C1 \\ C2 \\ C3 \\ Pass \\ Pub \\ FB \\ Sleep \end{array} \begin{array}{c} \begin{array}{ccccccc} C1 & C2 & C3 & Pass & Pub & FB & Sleep \end{array} \\ \begin{bmatrix} & 0.5 & & & & 0.5 & \\ & & 0.8 & & & & 0.2 \\ & & & 0.6 & 0.4 & & \\ & & & & & & 1.0 \\ 0.2 & 0.4 & 0.4 & & & & \\ 0.1 & & & & & 0.9 & \\ & & & & & & 1 \end{bmatrix} \end{array}$$



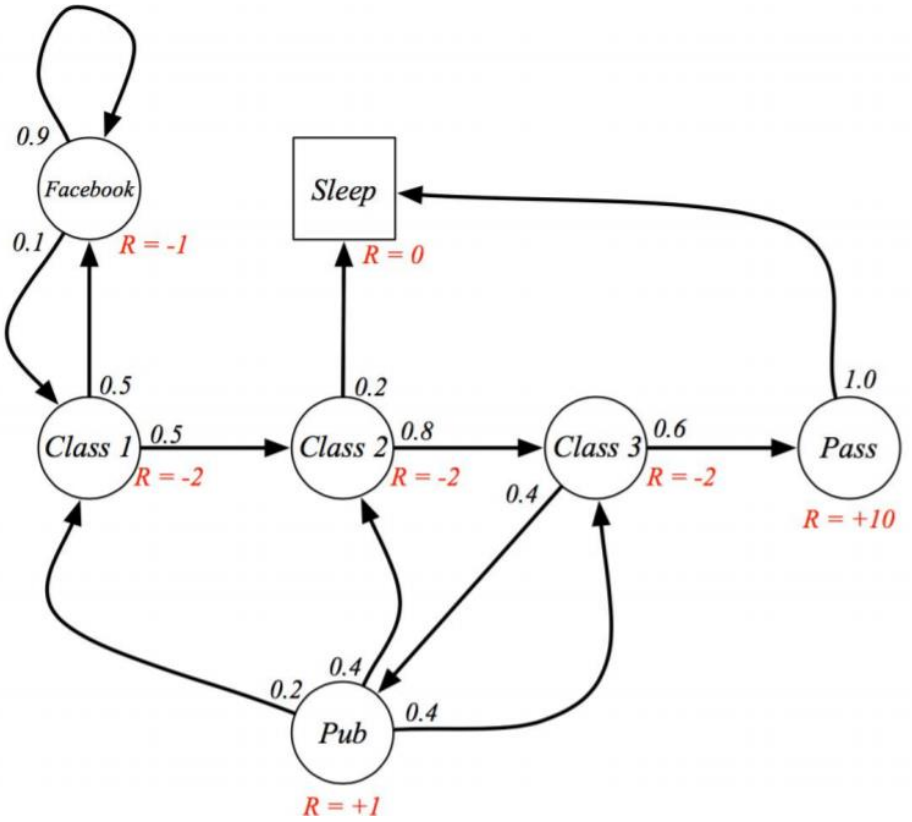
- **Example: Student Markov Process**

Markov Reward Process (MRP)

- **Markov reward process (MRP)**
  - Is a Markov process with reward value.

Definition: Markov Reward Process is a tuple $\langle S, P, R, \gamma \rangle$

- S is a set of states
- P is a state transition probability matrix
  $P_{ss'} = P[S_{t+1} = s' | S_t = s]$
- R is a reward function
  $R_s = E[R_{t+1} | S_t = s]$
  (delayed reward at state s)
- γ is a discount factor,
  $\gamma \in [0,1]$



Markov Reward Process (MRP)

Return

- The return $\boldsymbol{G_t}$ is the total discounted reward from time step t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

  - The discount $\gamma \in [0,1]$ is the present value of future rewards.

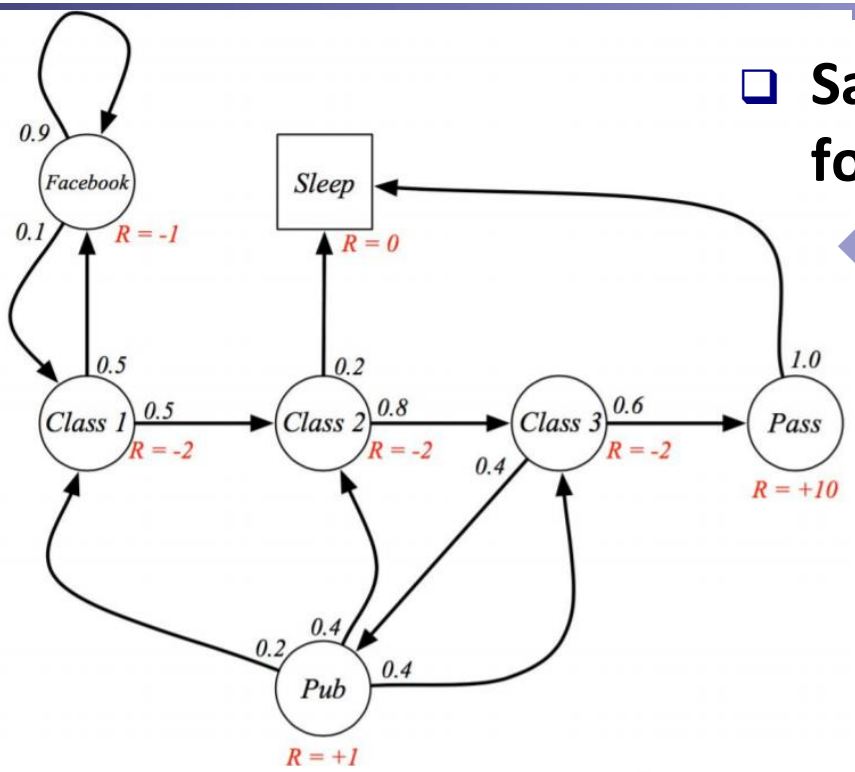
Value function

- The value function $\boldsymbol{v(s)}$ gives the long-term value of state s.
- The state value function $\boldsymbol{v(s)}$ of an MRP is the expected return starting from state s.

$$v(s) = E[G_t | S_t = s]$$

  - Depending on the path (state transitions) from state s, different return G_t is obtained. $\rightarrow$ expectation of G_t

Markov Reward Process (MRP)



- **Sample returns for sample episodes for 'Student MRP'**
  - starting from $S_1 = C1$ with $\gamma = 1/2$

$$G_1 = R_2 + \gamma R_3 + \ldots + \gamma^{T-2} R_T$$

C1 C2 C3 Pass Sleep	$v_1 = -2 - 2 * \frac{1}{2} - 2 * \frac{1}{4} + 10 * \frac{1}{8}$	=	−2.25
C1 FB FB C1 C2 Sleep	$v_1 = -2 - 1 * \frac{1}{2} - 1 * \frac{1}{4} - 2 * \frac{1}{8} - 2 * \frac{1}{16}$	=	−3.125
C1 C2 C3 Pub C2 C3 Pass Sleep	$v_1 = -2 - 2 * \frac{1}{2} - 2 * \frac{1}{4} + 1 * \frac{1}{8} - 2 * \frac{1}{16} \ldots$	=	−3.41
C1 FB FB C1 C2 C3 Pub C1 …	$v_1 = -2 - 1 * \frac{1}{2} - 1 * \frac{1}{4} - 2 * \frac{1}{8} - 2 * \frac{1}{16} \ldots$	=	−3.20
FB FB FB C1 C2 C3 Pub C2 Sleep			

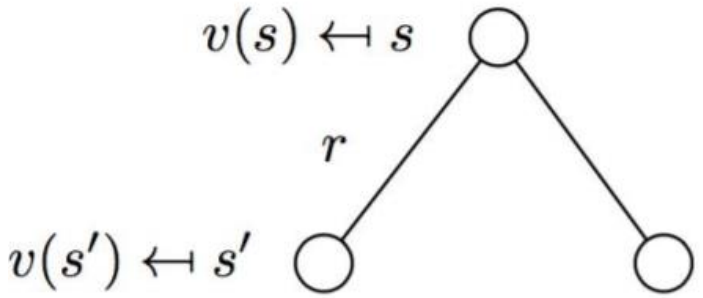
Bellman Equations for MRP

- **The value function $v(s)$ at state s can be decomposed into two parts:**
  - Immediate reward: R_{t+1}
  - Discounted value of successor state: $\gamma v(S_{t+1})$

$$
\begin{aligned}
v(s) &= E[G_t | S_t = s] \\
&= E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots | S_t = s] \\
&= E[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \cdots) | S_t = s] \\
&= E[R_{t+1} + \gamma G_{t+1} | S_t = s] \\
&= E[R_{t+1} + \gamma v(S_{t+1}) | S_t = s]
\end{aligned}
$$

Bellman Equation for MRP

$$v(s) = E[R_{t+1} + \gamma v(S_{t+1}) | S_t = s]$$
$$= R_s + \gamma \sum_{s' \in S} P_{ss'} v(s')$$



- **How we can get $v(s)$ at each state s?**
  - The Bellman equation can be expressed concisely using matrices, $\boldsymbol{v = R + \gamma P v}$

$$\begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix} = \begin{bmatrix} \mathcal{R}_1 \\ \vdots \\ \mathcal{R}_n \end{bmatrix} + \gamma \begin{bmatrix} \mathcal{P}_{11} & \dots & \mathcal{P}_{1n} \\ \vdots & & \\ \mathcal{P}_{11} & \dots & \mathcal{P}_{nn} \end{bmatrix} \begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix}$$

  - The Bellman equation is a linear equation.
  - It can be solved directly.

$$\boldsymbol{v = R + \gamma P v}$$
$$\boldsymbol{(I - \gamma P) v = R}$$
$$\boldsymbol{v = (I - \gamma P)^{-1} R}$$

Bellman Equations for MRP

- **Solving the Bellman equation**

$$\boldsymbol{v} = (\boldsymbol{I} - \boldsymbol{\gamma P})^{-1}\boldsymbol{R}$$

  - Computational complexity is $O(n^3)$ for n states.
  - Direct solution only possible for small MRPs.
  - There are many iterative methods for large MRPs.
    - Dynamic programming
    - Monte-Carlo evaluation
    - Temporal-Difference learning

$$v(s) = R_s + \gamma \sum_{s' \in S} P_{ss'} v(s')$$

$$v(s) \leftarrow v(s) + \alpha[R_s + \gamma \sum_{s' \in S} P_{ss'} v(s') - v(s)]$$

Markov Decision Process

- **A Markov Decision Process (MDP) is a Markov reward process with decision.**

Definition: A Markov Decision Process is a tuple $\langle S, A, P, R, \gamma \rangle$

- S is a finite set of states
- A is a finite set of actions
- P is a state transition probability matrix

  $$P_{ss'}^{a} = P\,[S_{t+1} = s' | S_t = s, A_t = a]$$

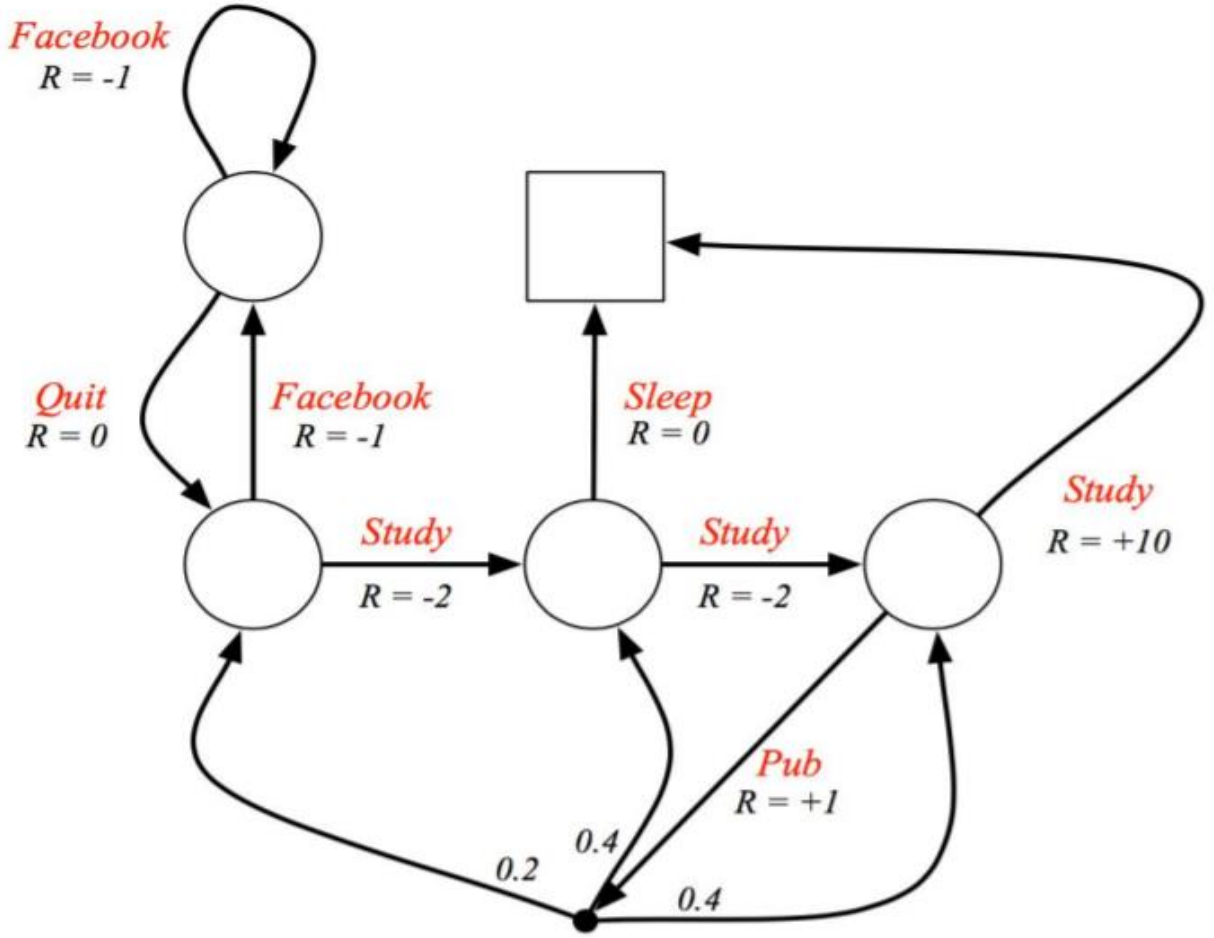
  ✓ For an action, the next state is determined with probability $P_{ss'}^{a}$ (if $P_{ss'}^{a} = 1$, it is deterministic)
- R is a reward function, $R_s^a = E[R_{t+1} | S_t = s, A_t = a]$
- γ *is a discount factor,* $\gamma \in [0,1]$

Definition: A policy is a distribution over actions given states,

$$\pi(a|s) = P\,[A_t = a | S_t = s]$$

Markov Decision Process

- **The Student MDP**



Markov Decision Process

- **Given an MDP $M = \langle S, A, P, R, \gamma \rangle$ and policy π**

$$P_{ss'}^{\pi} = \sum_{a \in A} \pi(a|s) P_{ss'}^{a} \qquad R_s^{\pi} = \sum_{a \in A} \pi(a|s) R_s^{a}$$

Definition: The state-value function $v_\pi(s)$ of an MDP is the expected return starting from state s, and then following policy π

$$v_\pi(s) = E_\pi[G_t | S_t = s]$$

Definition: The action-value function $q_\pi(s, a)$ of an MDP is the expected return starting from state s, taking action a, and then following policy π

$$q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a]$$

MDP: Bellman Expected Equation

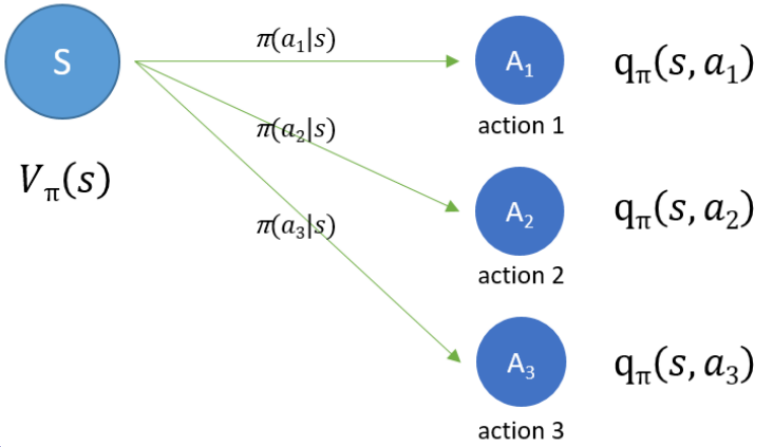
- The state-value function can be again be decomposed into immediate reward plus discounted value of successor state,

$$v_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s]$$

- The action-value function can similarly be decomposed

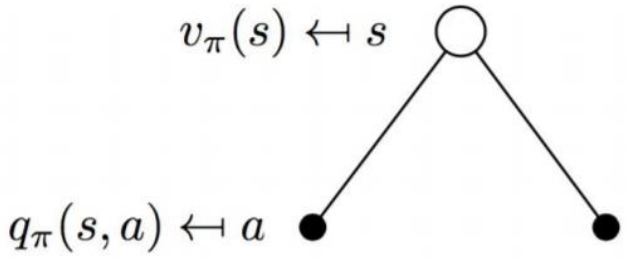
$$q_\pi(s,a) = E_\pi[G_t|S_t = s, A_t = a]$$
$$= E_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a]$$

- Relationship between $v_\pi(s)$ and $q_\pi(s,a)$

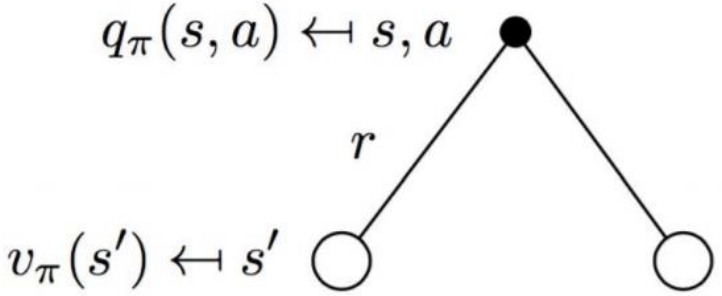


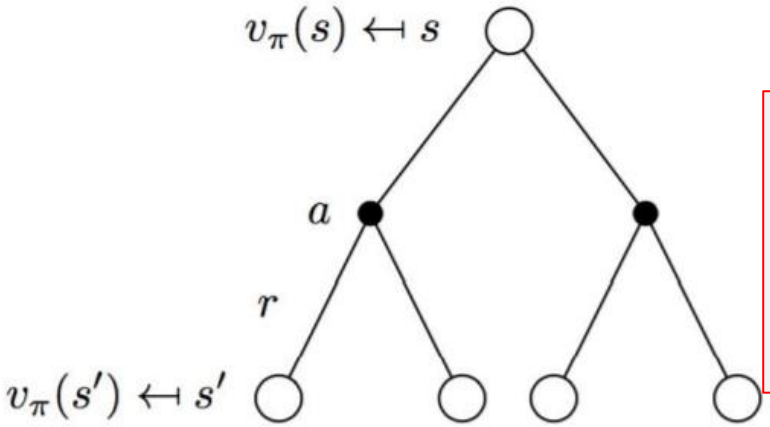
MDP: Bellman Expected Equation

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) q_\pi(s,a)$$



$$q_\pi(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s')$$

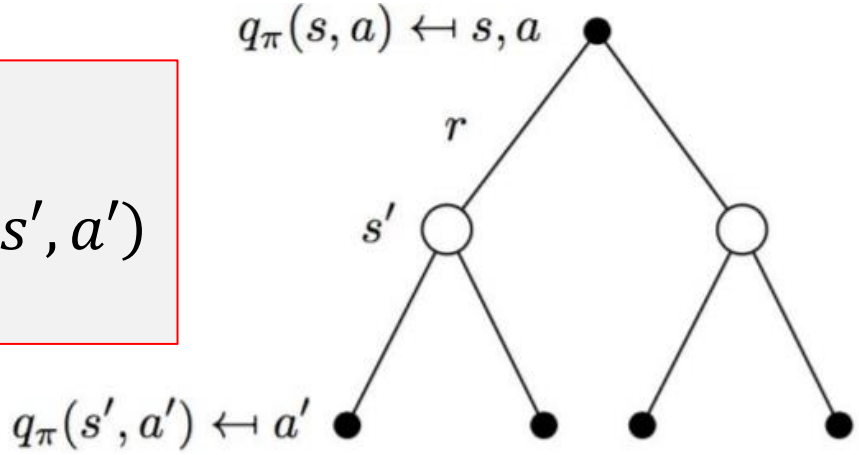




$$v_\pi(s) = \sum_{a \in A} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s') \right)$$

MDP: Bellman Expected Equation

$$q_\pi(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a'|s') q_\pi(s',a')$$



Definition: The optimal state-value function $v_*(s)$ is the maximum value function over all policies

$$v_*(s) = \max_\pi v_\pi(s)$$

Definition: The optimal action-value function $q_*(s,a)$ is the maximum action-value function over all policies

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

MDP: Optimal Policy

- **Theorem:** For any Markov Decision Process,
  - There exist an optimal policy π_* that is better than or equal to all other policies π
  - All optimal policies achieve the optimal value function,

$$v_{\pi_*}(s) = v_*(s)$$

  - All optimal policies achieve the optimal action-value function,

$$q_{\pi_*}(s,a) = q_*(s,a)$$

- Optimal policy:

$$\pi_*(a|s) = \begin{cases} 1 & if\ a = \underset{\forall a \in A}{\operatorname{argmax}}\, q_*(s,a) \\ 0 & otherwise \end{cases}$$

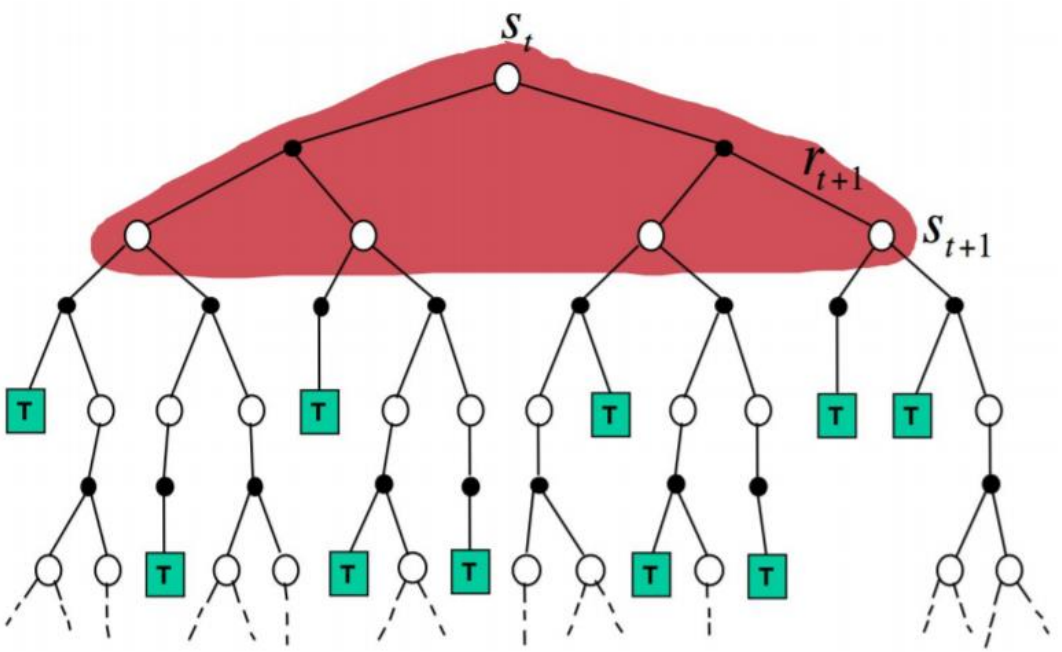
- Optimal state-value function:

$$v_*(s) = \max_a q_*(s,a)$$

MDP: Solving Bellman Equations

- **There are many iterative methods for large MRPs.**
  - Dynamic programming

$$v_\pi(s) \leftarrow \sum_{a \in A} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s') \right)$$

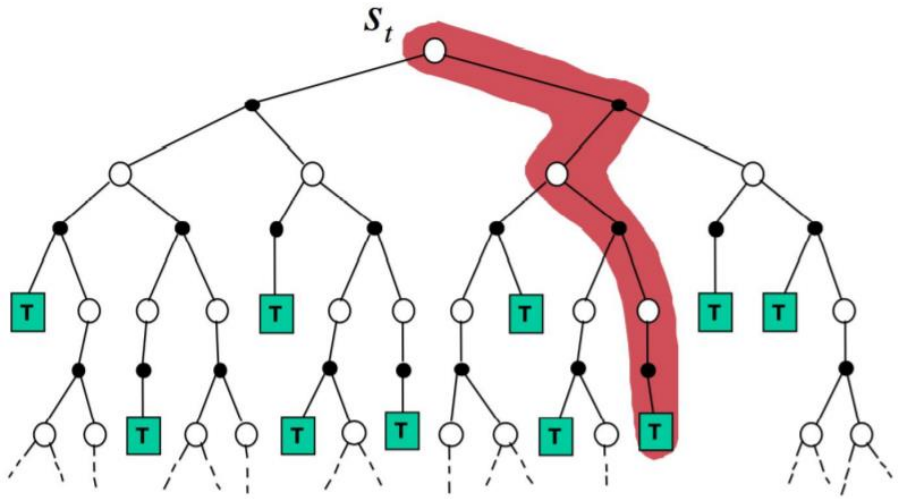


MDP: Solving Bellman Equations

- Monte-Carlo evaluation

$$v_\pi(s) = E_\pi[G_t | S_t = s]$$

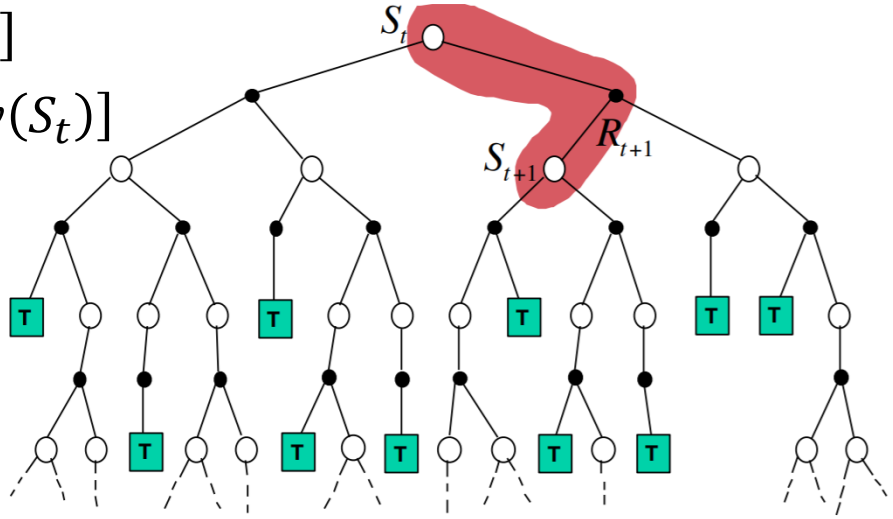
$$v(S_t) \leftarrow v(S_t) + \alpha[G_t - v(S_t)]$$



- Temporal-Difference learning

$$v_\pi(s) = E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s]$$

$$v(S_t) \leftarrow v(S_t) + \alpha[(R_{t+1} + \gamma v(S_{t+1})) - v(S_t)]$$



Reinforcement Learning Models

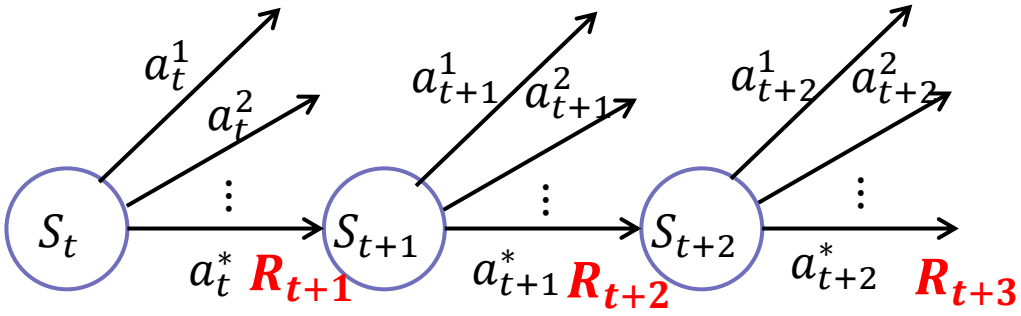
- **Model-based/Model-free, On-policy/Off-policy**

Model	Model-based	Agent knows everything about the MDP model including P_{ss}^{a},	Dynamic Programming
	Model-free	Agent observes everything as it goes	Monte-Carlo Temporal-Difference Q-learning
Policy	On-Policy	Target policy = Behavior policy	Monte-Carlo Temporal-Difference (SARSA)
	Off-Policy	Target policy ≠ Behavior policy	Q-learning

Q-Learning

Architecture

- Each state has actions to move to the next states.
- After action, a reward is given.



a_t^i: action i at time t, S_t: state at time t
a_t^*: action at t that the agent selected
R_{t+1}: reward from a_t^*

Q-Learning

- **When an agent takes action a_t^* in state S_t at time t, the predicted rewards is defined as**

$$\boldsymbol{Q(S_t, a_t^*) = E\{R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots\}}$$

$$\gamma = discount\ factor\ for\ the\ future\ reward$$

$$Q(S_t, a_t^*) = E\left\{\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right\} = E\left\{R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2}\right\}$$
$$= E\{R_{t+1} + \gamma Q(S_{t+1}, a_{t+1}^*)\}$$

  - As a result, the Q value at time t is easily calculated by R_{t+1} and Q value of the next step.

Q-Learning

$$Q(S_t, a_t^*) = E\{R_{t+1} + \gamma Q(S_{t+1}, a_{t+1}^*)\}$$

if we assume every time we select the best action, then the iterative update form is

$$Q(S_t, a_t^*) \leftarrow Q(S_t, a_t^*) + \alpha\{R_{t+1} + \gamma Q(S_{t+1}, a_{t+1}^*) - Q(S_t, a_t^*)\}$$

error term: estimated reward value – current value

$$Q(S_t, a_t^*) \leftarrow (1-\alpha) Q(S_t, a_t^*) + \alpha\{R_{t+1} + \gamma Q(S_{t+1}, a_{t+1}^*)\}$$

- **Iterative Update**

$$Q(S_t, a_t) \leftarrow (1-\alpha) Q(S_t, a_t) + \alpha\{R_{t+1} + \gamma max_{a_{t+1}} Q(S_{t+1}, a_{t+1})\}$$

α: *learning rate*

current value

learning rate

estimated future value

Exploration vs Exploitation

- Exploration vs Exploitation
  - Online decision making involves a fundamental choice:
    - ***Exploitation***: make the best decision given current information
    - ***Exploration***: gather more information
  - The best long–term strategy may involve short–term sacrifices.
  - Gather enough information to make the best overall decisions.

- **Exploration methods**
  - ϵ-greedy: one of the most used exploration strategies. It uses $0 \leq \epsilon \leq 1$ as parameter of exploration.
    - The agent chooses the action with the highest Q-value in the current state with probability $1 - \epsilon$, and a random action otherwise.

Exploration vs Exploitation

- Decaying E-greedy

$$\epsilon = \frac{\epsilon_{init}}{(i+1)} \qquad i = iteration\ number$$

- Boltzmann (or softmax) exploration:
  - One drawback of ϵ -greedy exploration is that the exploration action is selected uniform randomly from the set of possible actions.
  - Using softmax exploration with intermediate values for τ, the agent still most likely selects the best action, but other actions are ranked instead of randomly chosen.

$$p_{a_k} = \frac{e^{Q(s,a_k)/\tau}}{\sum_{i=1}^{m} e^{Q(s,a_i)/\tau}}$$

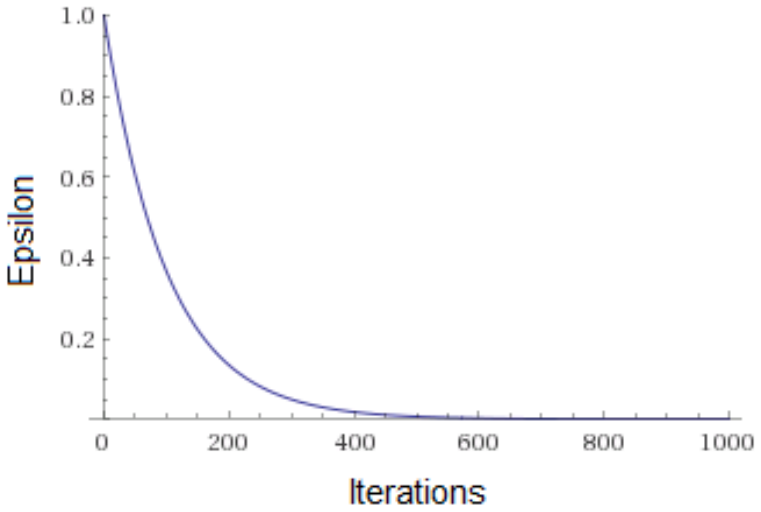
  - When$\tau = 0$ the agent does not explore at all (greedy), and when $\tau = 1$ the agent selects softmax random actions.

Exploration vs Exploitation

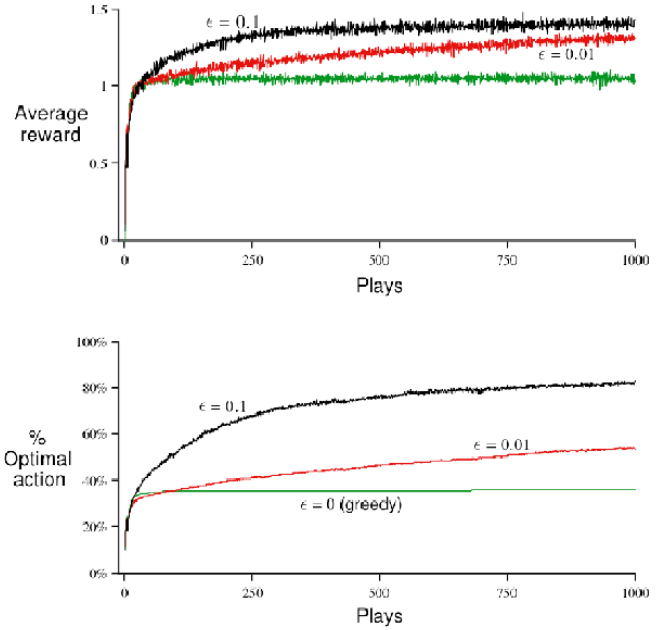
- Add random noise
  - selected action a :

$$a = argmax_{a'}\{Q(s,a') + random\_noise\} \text{ or}$$

$$a = argmax_{a'}\left\{Q(s,a') + \left(\frac{random\_noise}{i+1}\right)\right\}$$



Decaying E-greedy (ϵ_{init}=0.99)



Q-Learning Algorithm

0 $Q(s,a) \leftarrow initialization\ (e, g.all\ zeros)\ ,\ \forall s,a$
$s \leftarrow initial\ state\ //random$

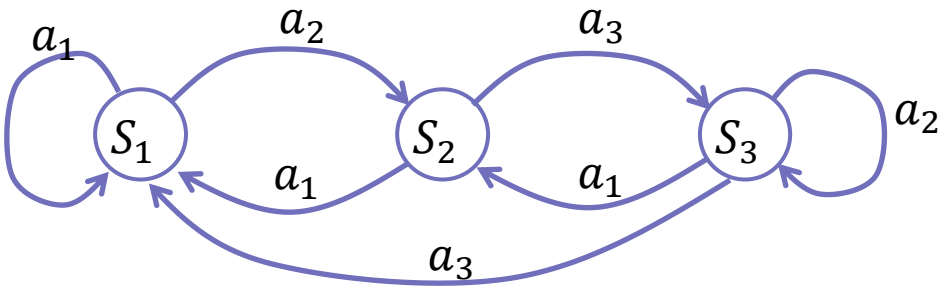
1 $if\ (random\ number \leq \epsilon)$
$a = random\ choice$
$else$
$a = argmax_a Q(s,a)$

2 $apply\ action\ a$
s_{new}= $observed\ new\ state$
$R = reward\ from\ the\ environment$

3 $Q(s,a) = (1-\alpha)Q(s,a) + \alpha\{R + \gamma max_{a'} Q(s_{new}, a')\}$

4 $s \leftarrow s_{new}$
$Go\ to\ 1$

Q Matrix Update Example



$\alpha = 0.5, \gamma = 0.5$

- We don't know the state transition diagram and transition probability unlike MDP (Markov Decision Process).
- We only know possible action set for each state.

$$Q = \begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \end{array} \begin{array}{c} \begin{array}{ccc} a_1 & a_2 & a_3 \end{array} \\ \begin{bmatrix} 0 & 1 & - \\ 1 & - & 2 \\ 0 & 1 & 2 \end{bmatrix} \end{array}$$

$$Q = \begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \end{array} \begin{array}{c} \begin{array}{ccc} a_1 & a_2 & a_3 \end{array} \\ \begin{bmatrix} 0 & 2.5 & - \\ 1 & - & 2 \\ 0 & 1 & 2 \end{bmatrix} \end{array}$$

$.initial\ state = s_1$

$.select\ action\ argmax_a Q(s_1, a);$

$argmax_a\{Q(s_1, a_1), Q(s_1, a_2)\} = argmax_a\{0,1\} = a_2$

$.perform\ action\ a_2$

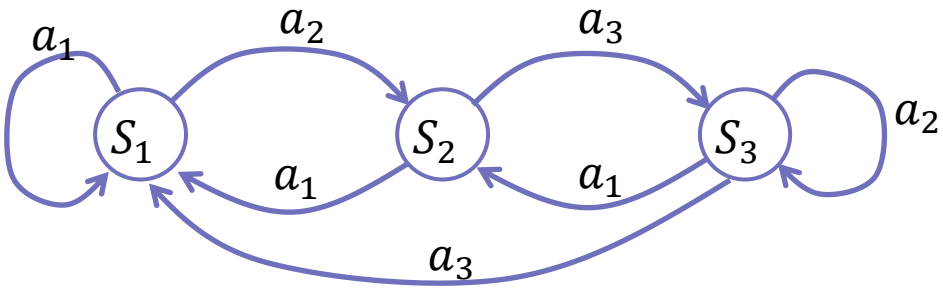
$.observed\ state = s_2,$

$measured\ (computed)\ reward = +3$

$.update\ Q(s_1, a_2) = (1-\alpha)Q(s,a) + \alpha\{r + \gamma max_{a'} Q(s_{new}, a')\}$

$$Q(s_1, a_2) = 0.5Q(s_1, a_2) + 0.5\{3 + 0.5max_{a'} Q(s_2, a')\}$$
$$= 0.5 \times 1 + 0.5\{3 + 0.5\max(Q(s_2, a_1), Q(s_2, a_3))\}$$
$$= 0.5 \times 1 + 0.5\{3 + 0.5\max(1,2)\} = 2.5$$

Q Matrix Update Example



$\alpha = 0.5, \gamma = 0.5$

$$Q = \begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \end{array} \begin{array}{c} \begin{array}{ccc} a_1 & a_2 & a_3 \end{array} \\ \begin{bmatrix} 0 & 2.5 & - \\ 1 & - & 2 \\ 0 & 1 & 2 \end{bmatrix} \end{array}$$

$$Q = \begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \end{array} \begin{array}{c} \begin{array}{ccc} a_1 & a_2 & a_3 \end{array} \\ \begin{bmatrix} 0 & 2.5 & - \\ 1 & - & 0 \\ 0 & 1 & 2 \end{bmatrix} \end{array}$$

. $current\ state = s_2$

. $select\ action\ argmax_a Q(s_2, a)$;

$argmax_a\{Q(s_2, a_1), Q(s_2, a_3)\} = argmax_a\{1,2\} = a_3$

. $perform\ action\ a_3$

. $observed\ state = s_3$,

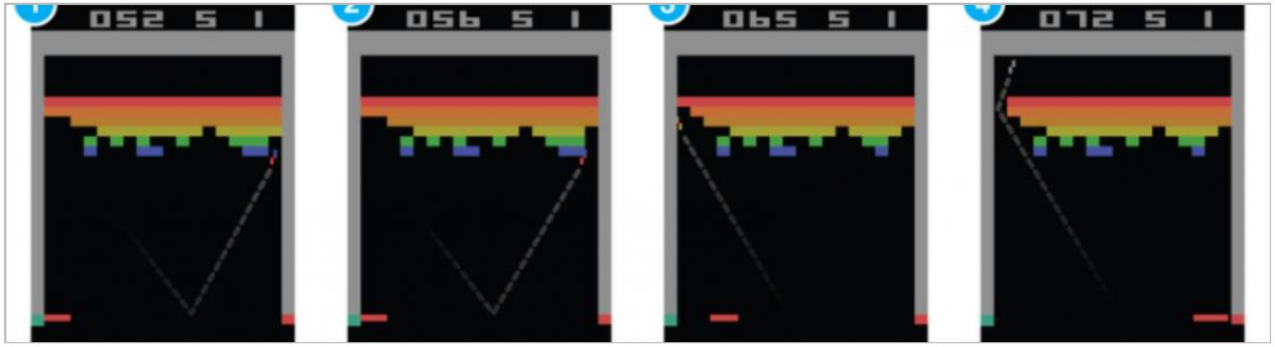
$measured\ (or\ defined)\ reward = -3$

. $update\ Q(s_2, a_3) = (1-\alpha)Q(s, a) + \alpha\{r + \gamma max_{a'} Q(s_{new}, a')\}$

$$Q(s_2, a_3) = 0.5Q(s_2, a_3) + 0.5\{-3 + 0.5max_{a'} Q(s_3, a')\}$$
$$= 0.5 \times 2 + 0.5\{-3 + 0.5\max[Q(s_3, a_1), Q(s_3, a_2), Q(s_3, a_3)]\}$$
$$= 1 + 0.5\{-3 + 0.5\max(0,1,2)\} = 0$$

Deep Reinforcement Learning

- **Problems of Reinforcement Learning (Q-Learning)**
  - For large scale or infinite dimensional state space, it is very hard to learn all {state, action} Q table values.
  - If the current situation is a state in the following arcade game, how many states exist?



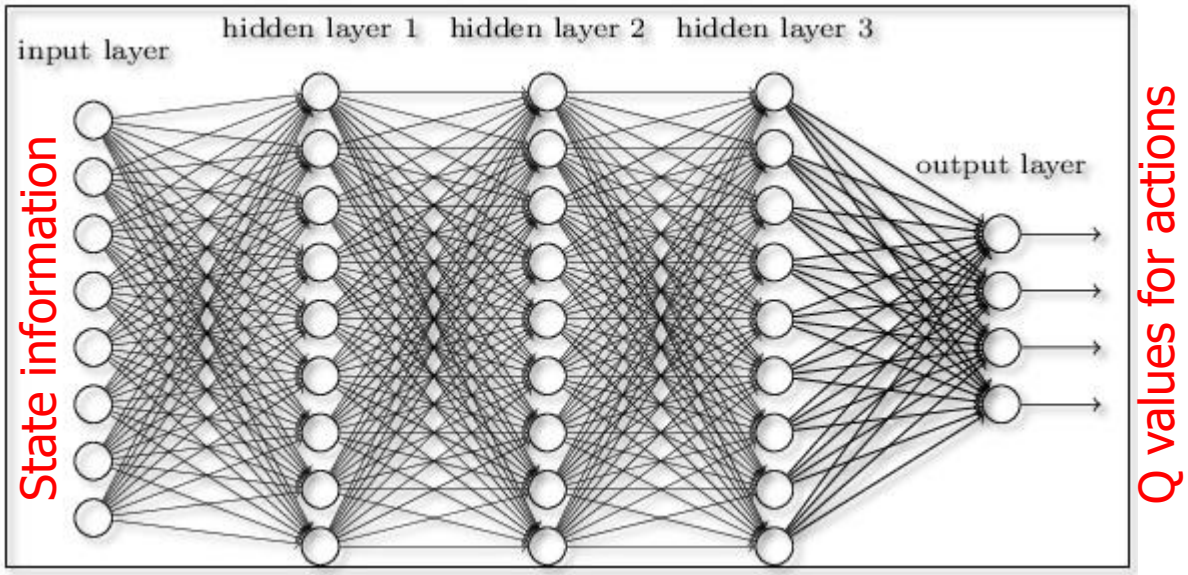
- **Combine supervised deep learning with reinforcement learning.**
  - We don't have any labels for the states (inputs of DNN), but loss function can be defined.

Deep Reinforcement Learning

$Q =$

State \ Action	0	1	2	3	4	5
0	−1	−1	−1	−1	0	−1
1	−1	−1	−1	0	−1	100
2	−1	−1	−1	0	−1	−1
3	−1	0	0	−1	0	−1
4	0	−1	−1	0	−1	100
5	−1	0	−1	−1	0	100

<Q table>



- **Possible solution:**
  - How about using a function approximator for Q-table?
  - Instead of finding out all Q-table entities, approximating them using DNN.
    - For the given input (state information), determine the Q values of the possible actions of the sate.

Deep Reinforcement Learning

- **Q-value approximator**

$$\hat{Q}(s,a;\theta) \approx Q^*(s,a) = E\{R + \gamma max_{a'} Q^*(s',a')\}, \quad \theta\text{=weights}$$

- **Choose NN θ to minimize**

$$\min_{\theta} \sum_{t=0}^{T} \left[\left\{R_{t+1} + \gamma max_{a'} \hat{Q}(s_{t+1}, a' \,|\theta)\right\} - \hat{Q}(s_t, a_t|\theta)\right]^2$$

Loss Function

  - classical Q-learning

$$Q(S_t, a_t) \leftarrow Q(S_t, a_t) + \alpha\{R_{t+1} + \gamma max_{a_{t+1}} Q(S_{t+1}, a_{t+1}) - Q(S_t, a_t)\}$$

target prediction

  - DNN learning: minimize a Loss Function
    - Agent performs the best action for the current state based on the Q table (output of the DNN) → get the reward and new state → evaluate the loss function (batch) → update the weights of DNN

Algorithm

Algorithm 1 Deep Q-learning

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 $\phi_{t+1} = \phi(s_{t+1})$

    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 equation for updating θ
  end for
end for

input image

preprocessed image:
resize, filtered, gray colored

Q-Network

- **Problem**: Reinforcement learning is known to be unstable or even to diverge when use a nonlinear function approximator such as a neural network.
  - Correlation between samples
  - Small updates to Q value may significantly change the policy.
- **Deep Q Network (DQN)**

Human-level control through deep reinforcement learning

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg & Demis Hassabis

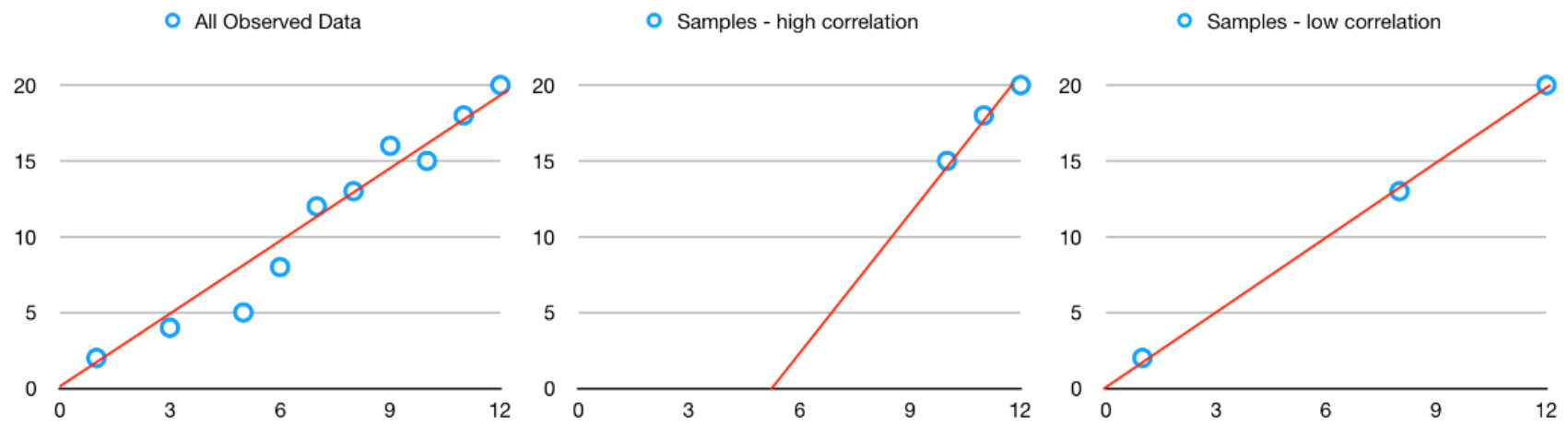
Nature

Google DeepMind

- Learn to master 49 different Atari games directly from game screens
- Beat the best performing learner from the same domain in 43 games

Deep Q Network (DQN)

- **Solutions in DQN:**
  - Go deep
  - Capture and replay: reduce the correlations between samples
  - Separate networks: non-stationary targets → stationary targets
- **Strong correlation of consecutive input samples (sequence)**
  - results in non-optimal prediction



Deep Q Network (DQN)

- **Solutions in DQN:**
  - Go deep
  - Capture and replay: reduce the correlations between samples
  - Separate networks: non-stationary targets → stationary targets
- **Strong correlation of consecutive input samples (sequence)**
  - results in non-optimal prediction
  - suppose NN can be modeled as $\hat{Q}(s,a|\theta) = x(s)^T W_a$
  - loss function $L(W_a) = \frac{1}{2}\left[Q^*(s,a) - \hat{Q}(s,a|\theta)\right]^2$

$$= \frac{1}{2}\left[Q^*(s,a) - x(s)^T W_a\right]^2$$

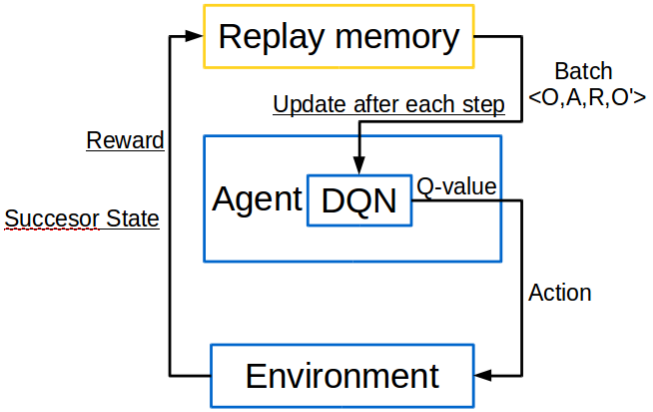
  - stochastic gradient decent update

$$\nabla_{W_a} L(W_a) = (Q^*(s,a) - x(s)^T W_a)x(s)$$

    - If consecutive $x(s)$ are similar, then the update goes to a biased direction.

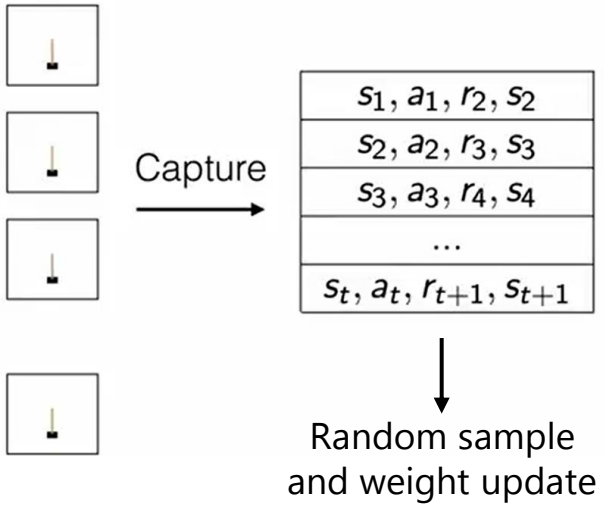
Deep Q Network (DQN)

- To remove correlations, build data-set from agent's own experience
  - Sample experiences from data-set and apply update.



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 $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for



Deep Q Network (DQN)

- **Target of loss function**

$$Loss = \sum_{j=1}^{mb} \left[\underbrace{\left\{ r_j + \gamma max_{a'_j} \hat{Q}(s'_j, a'_j|\theta) \right\}}_{\text{target}} - \underbrace{\hat{Q}(s_j, a_j|\theta)}_{\text{estimate}} \right]^2$$

target
(considered as a fixed value)

estimate

  - Take action a_j at state s_j, get reward r_j move to the new state s'_j
- Target is not fixed value. Every batch time it varies as $\hat{Q}(s_j, a_j|\theta)$ does.
- To help improve stability, fix the target network weights used in the target calculation for multiple updates
- Use a different set of weights to compute target than is being updated.
  - DQN uses experience replay and fixed Q-targets using the previous batch sample results.

Deep Q Network (DQN)

- **Target network separation**
  - To deal with non-stationarity, target parameters θ^- are held fixed.

$$Loss = \sum_{j=1}^{mb} \left[\left\{ r_j + \gamma max_{a'_j} \hat{Q}(s'_j, a'_j | \theta^-) \right\} - \hat{Q}(s_j, a_j | \theta) \right]^2$$

    - for θ^-, use the previous step θ

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$, add it to $\mathcal{B}$
2. sample mini-batch $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j\}$ from $\mathcal{B}$ uniformly
3. compute $y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}'_j)$ using *target* network $Q_{\phi'}$
4. $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\mathbf{s}_j, \mathbf{a}_j)(Q_\phi(\mathbf{s}_j, \mathbf{a}_j) - y_j)$ $(\phi := \theta)$
5. update ϕ'

Nature 2015 (DQN Algorithm)

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode $= 1, M$ **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For $t = 1$,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    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 D

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

DQN Architecture