# CS231n DL for CV

Assignment 2

1. Introduction
2. Image Classification ! | **Image Classification with Linear Classifiers**
3. Loss Functions and Optimization ! | **Regularization and Optimization**
4. Neural Networks !
5. CNN
6. Training Neural Networks !! | **CNN Architectures (9)**
7. Training Neural Networks !!
8. Deep Learning Software | **Visualizing and Understanding (12)**
9. CNN Architectures !! | **Object Detection and Image Segmentation (11)**
10. Recurrent Neural Networks
11. Detection and Segmentation | **Attention and Transformers ! (no)**
12. Visualizing and Understanding | **Video Understanding**
13. Generative Models
14. Deep RL | **Self-supervised Learning ! (no)**
15. Efficient Methods and Hardware for DL
16. Adversarial Examples and Adversarial Training

| **Low-Level Vision, 3D Vision, Human-Centered Artificial Intelligence (Healthcare), Fairness in Visual Recognition**

## 1. Introduction

## 2. Image Classification

Nearest Neighbors → K-Nearest Neighbors (hyperparameters: distance metric, K)

Choose hyperparameters using the validation set; only run once on the test set

Linear classification: $f(x, W) = Wx + b$

## 3. Loss Functions and Optimization

Score function: $s = f(x, W) \overset{e.g.}{=} Wx$

Loss function:

Muticlass SVM loss: $L_i = \Sigma_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$

Softmax loss: $L_i = -\log(\frac{e^s k}{\Sigma_j e^{s_j}}) \in (0, \infty)$ where $P(Y = k|X = x_i) = \frac{e^s k}{\Sigma_j e^{s_j}}$

Full loss = data loss + regularization: $L = \frac{1}{N} \Sigma_i L_i + \lambda R(W)$

Regularization: prevent the model from doing too well on training data

Simple examples: L2: $R(W) = \Sigma_k \Sigma_l W_{k,l}^2$ L1: $R(W) = \Sigma_k \Sigma_l |W_{k,l}|$

Elastic net (L1 + L2): $R(W) = \Sigma_k \Sigma_l \beta W_{k,l}^2 + |W_{k,l,}|$

More complex: dropout; batch normalization; stochastic depth, fractional pooling, etc.

Gradient descent

Stochastic gradient descent: approximate sum using a minibatch of examples (e.g. 32/64/128)

```
while True:
  data_batch = sample_training_data(data, 256)
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights -= learning_rate * weights_grad
```

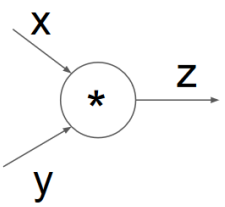
## 4. Neural Networks

Backpropagation

- (Fully-connected) Neural Networks are stacks of linear functions and nonlinear activation functions; they have much more representational power than linear classifiers
- backpropagation = recursive application of the chain rule along a computational graph to compute the gradients of all inputs/parameters/intermediates
- implementations maintain a graph structure, where the nodes implement the forward() / backward() API
- forward: compute result of an operation and save any intermediates needed for gradient computation in memory
- backward: apply the chain rule to compute the gradient of the loss function with respect to the inputs

## Modularized implementation: forward / backward API

Gate / Node / Function object: Actual PyTorch code




(x,y,z are scalars)

```
class Multiply(torch.autograd.Function):
  @staticmethod
  def forward(ctx, x, y):
    ctx.save_for_backward(x, y)
    z = x * y
    return z
  @staticmethod
  def backward(ctx, grad_z):
    x, y = ctx.saved_tensors
    grad_x = y * grad_z   # dz/dx * dL/dz
    grad_y = x * grad_z   # dz/dy * dL/dz
    return grad_x, grad_y
```

Need to cash some values for use in backward

Upstream gradient

Multiply upstream and local gradients

## 5. CNN

ConvNet is a sequence of Convolution Layers, interspersed with activation functions

Convolution layer: convolve the filter with the image — preserve spatial structure

Assume input is $W_1 \times H_1 \times C_1$

Conv layer needs 4 hyperparameters:

- number of filters K
- filter size F
- stride S
- zero padding P

This will produce an output of $W_2 \times H_2 \times K$ where

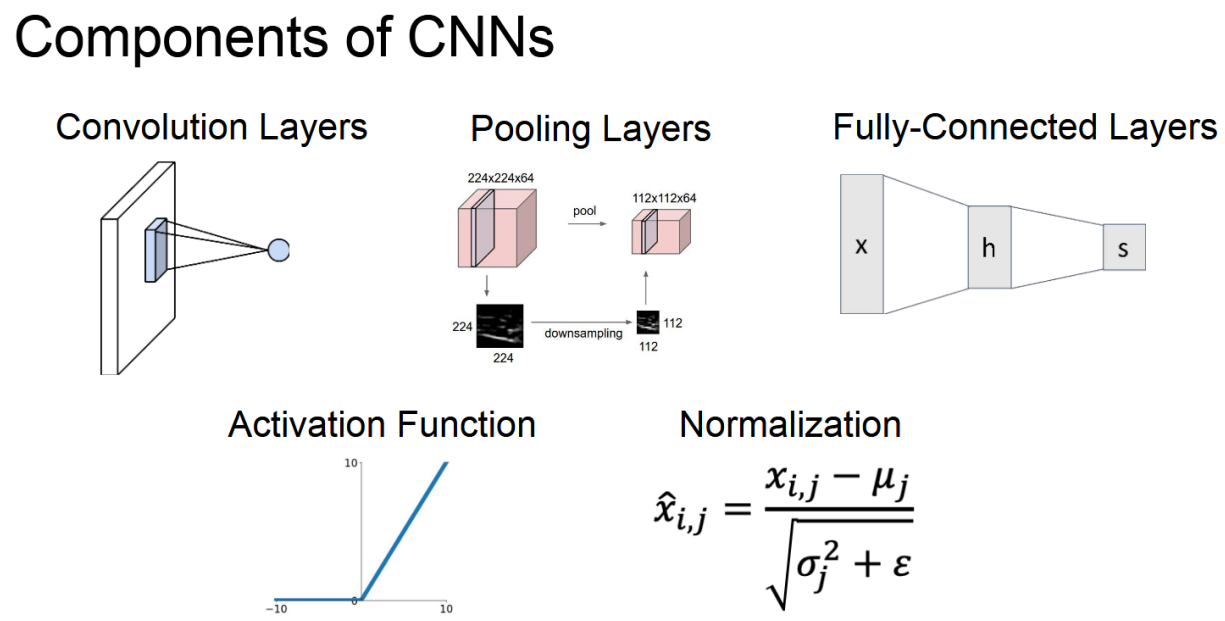
- $W_2 = (W_1 - F + 2P)/S + 1$
- $H_2 = (H_1 - F + 2P)/S + 1$

Number of parameters: $F^2CK$ and $K$ biases

Pooling layer: makes the representations smaller and more manageable; operates over each activation map independently

Fully Connected Layer (FC layer): contains neurons that connect to the entire input volume, as in ordinary Neural Networks

## 6. Training Neural Networks



Activation functions:

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

tanh

ReLU (Rectified Linear Unit): $f(x) = \max(0, x)$ - does nor saturate, computationally efficient, convergers faster

Leaky ReLU: $f(x) = \max(0.01x, x)$ - compared to ReLU, will not die

ELU (Exponential Linear Unit): $f(x) = \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$

Maxout “Neuron”: $\max(w_1^T x + b_1, w_2^T x + b_2)$ - generalizes ReLU and Leaky ReLU, but doubles the number of parameters

Batch normalization: consider a batch of activations at some layer, to make each dimension zero-mean unit-variance

1. compute mean and variance independently for each dimension
   $\mu_j = \frac{1}{N}\Sigma_1^N x_{i,j}$, $\sigma_j^2 = \frac{1}{N}\Sigma_1^N (x_{i,j} - \mu_j)^2$
2. normalize: $\hat{x_{i,j}} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$
3. scale and shift (to have some flexibility): $\hat{y_{i,j}} = \gamma_j \hat{y_{i,j}} + \beta_j$

Data Preprocessing:

original data → zero-centered data → normalized data

original data → decorrelated data → whitened data (by PCA and whitening of the data)

Weight Initialization:

Initialization too small — activations go to zero, gradients also zero, no learning

Initialization too big — activations saturate (for tanh), gradients zero, no learning

Initialization just right — nice distribution of activations at all layer, learning proceeds nicely

Activation statistics: e.g. W = 0.05 * np.random.randn(Din, Dout)

Xavier Initialization: e.g. W = np.random.randn(Din, Dout) / np.sqrt(Din)

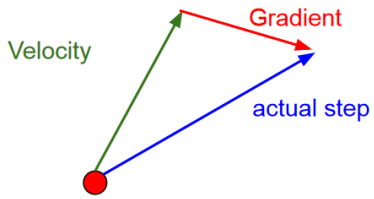
## 7. Training Neural Networks

SGD + Momentum: build up “velocity” as a running mean of gradients; Rho gives “friction”; typically rho=0.9 or 0.99

Nesterov Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$




Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$
$$\tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1}$$
$$= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)$$

“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

AdaGrad & RMSProp (Leaky AdaGrad)

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Adam

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Dropout

Model ensembles

Transfer learning

## 8. Deep Learning Software

CPU vs GPU

CPU: fewer cores, but each core is much faster and much more capable; great at sequential tasks

GPU: more cores, but each core is much slower and "dumber"; great at parallel tasks (e.g. matrix multiplication, 70x faster for deep learning)

Deep learning frameworks

Caffe

TensorFlow

PyTorch

- three levels of abstraction: tensor (imperative ndarray, but runs on GPU), variable (node in a computational graph; stores data and gradient),

module (a neural network layer; may store state or learnable weights)

- tf.keras; torch.nn

## 9. CNN Architectures

AlexNet (2012): first CNN-based winner

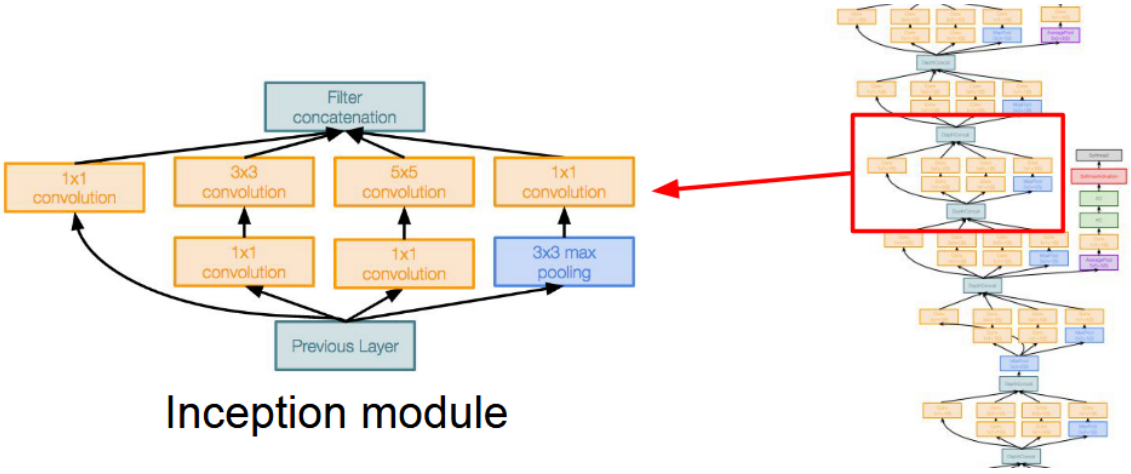
- ARCHITECTURE: CONV1 → MAX POOL1 → NORM1 → CONV2 → MAX POOL2 → NORM2 → CONV3 → CONV4 → CONV5 → MAX POOL3 → FC6 → FC7 → FC8

ZFNet (2013): improved hyperparameters over AlexNet

VGGNet (2014): small filters (3x3 conv), deeper networks (16/19 layers)

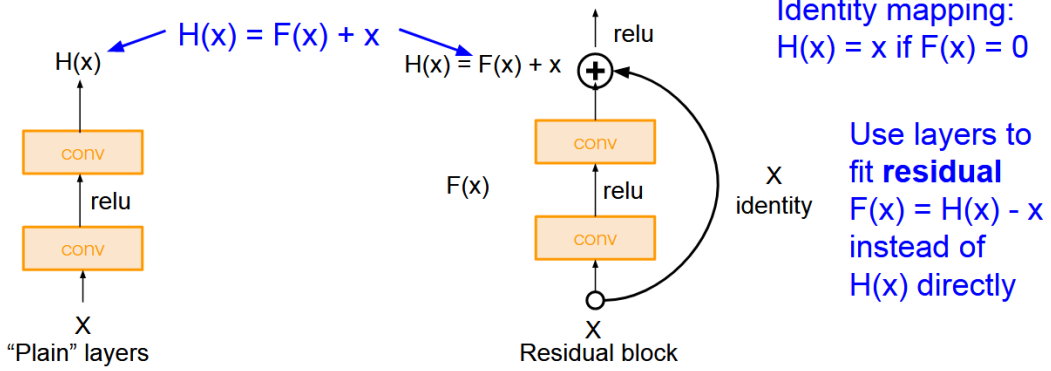
GoogleNet (2014): deeper networks, with computational efficiency

- “inception module”: design a good local network topology (network within a network) andthen stack these modules on top of each other
- add “1*1 conv, 64 filters” bottlenecks to reduces dimension (depth)
- global avg pool instead of FC layers

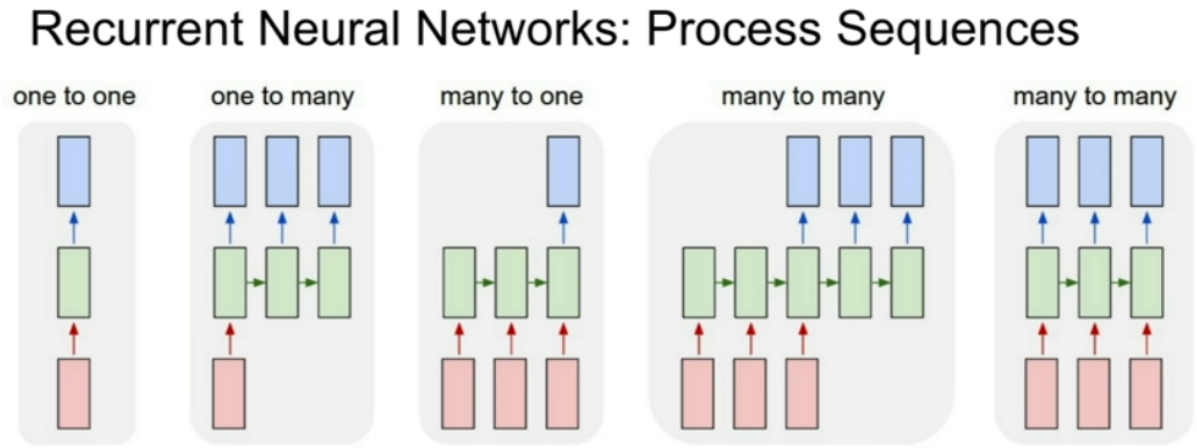



ResNet (2015): very deep networks using residual connections

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping



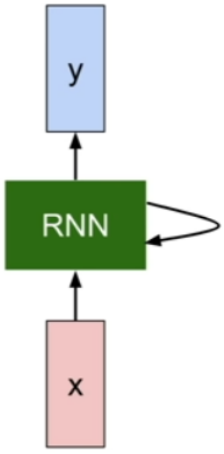

## 10. Recurrent Neural Networks

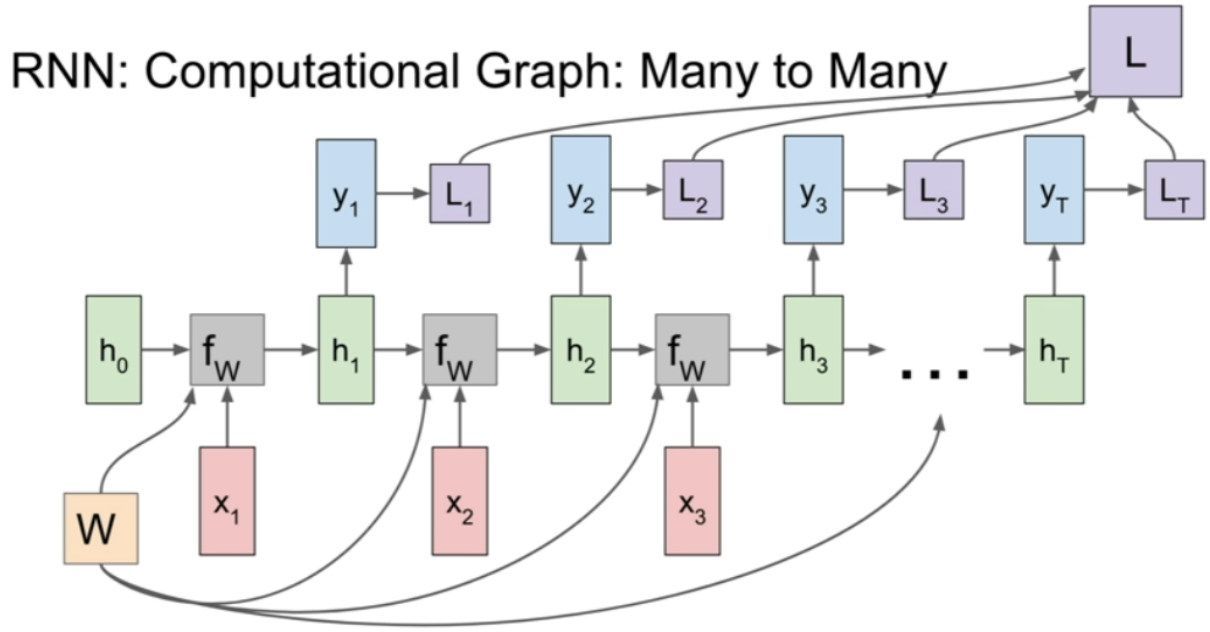



We can process a sequence of vectors **x** by applying a **recurrence formula** at every time step:

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

new state — some function with parameters W — old state — input vector at some time step






e.g. Vanilla RNN

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

$$y_t = W_{hy} h_t$$

Truncated Backpropogation through time: run forward and backward through chunks of the sequence instead of the whole sequence

Vanilla RNN Gradient Flow:

Largest singular value of W > 1: exploding gradients

Largest singular value of W< 1: vanishing gradients → change RNN architecture, LSTM

Long Short Term Memory (LSTM)

f: forget gate; i: input gate; g: gate gate; o: output gate

**Vanilla RNN**

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

**LSTM**

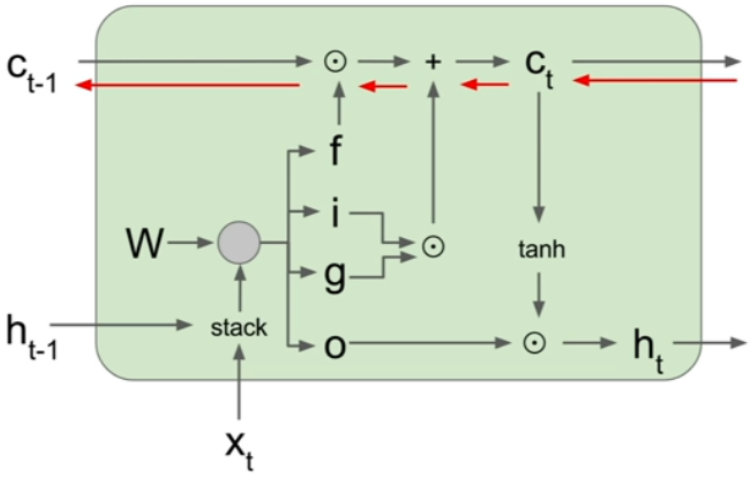
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

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

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

## Long Short Term Memory (LSTM): Gradient Flow

*[Hochreiter et al., 1997]*



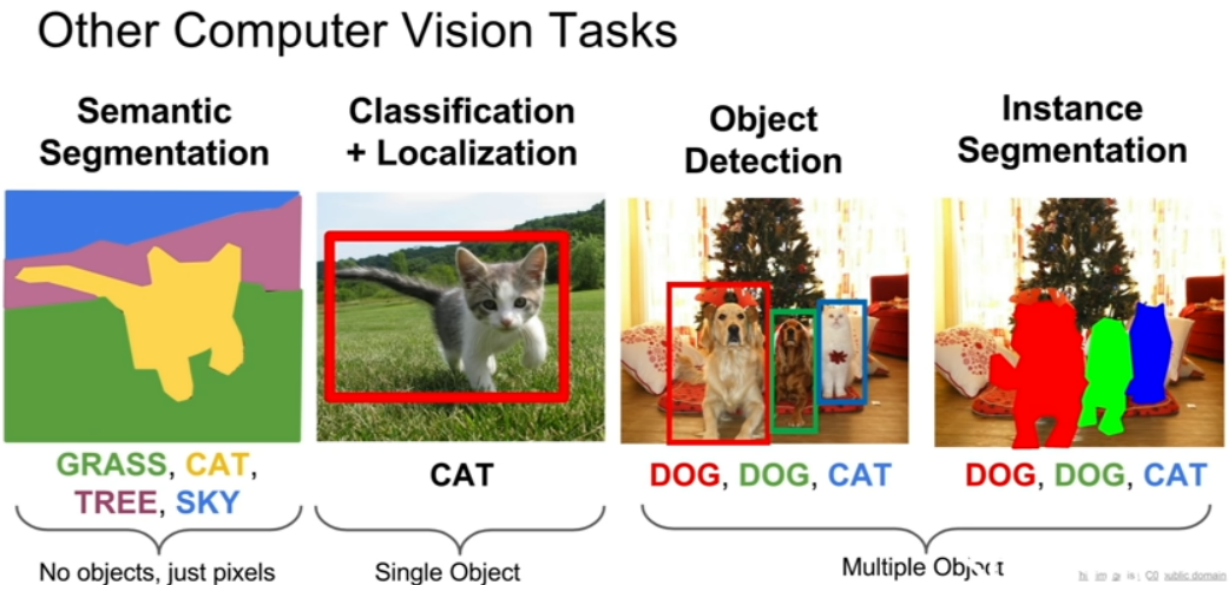

Backpropagation from $c_t$ to $c_{t-1}$ only elementwise multiplication by f, no matrix multiply by W

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

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

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

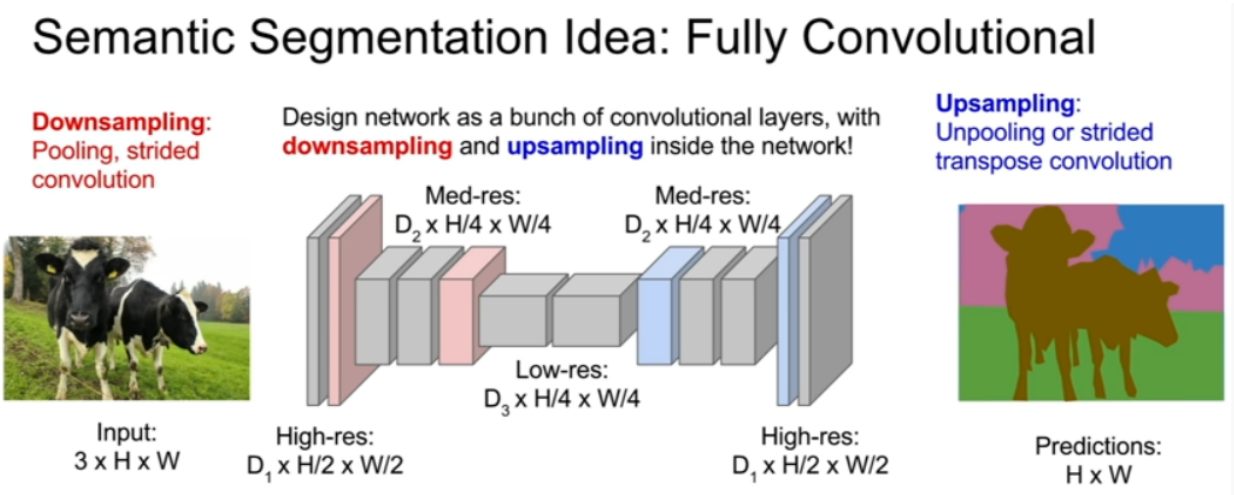
## 11. Detection And Segmentation



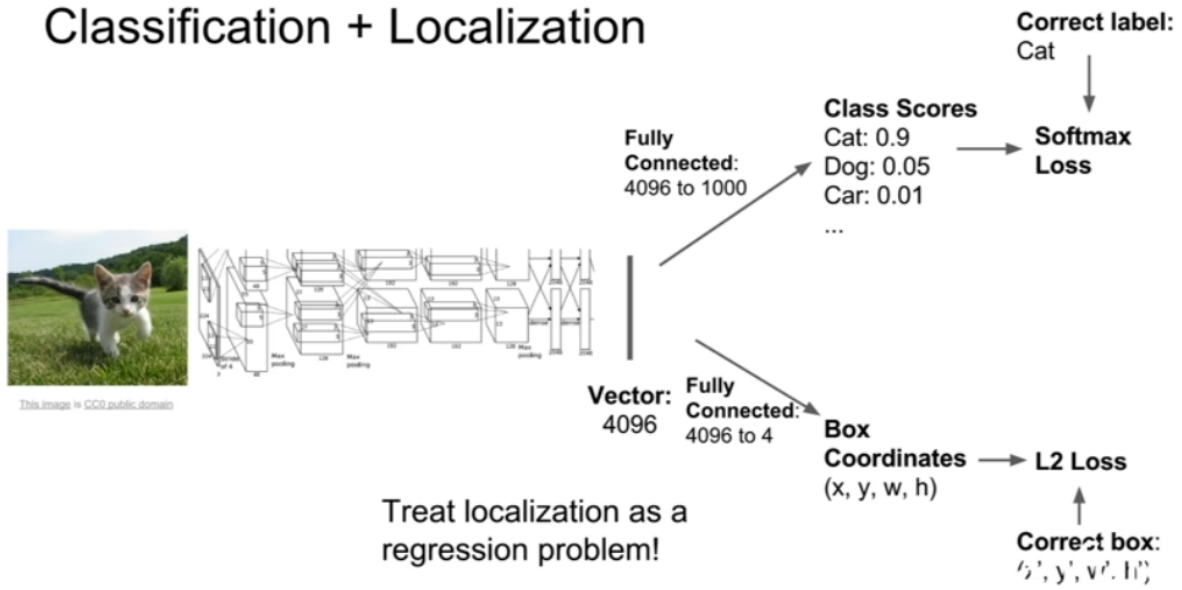

**Semantic Segmentation**

Label each pixel in the image with a category label

Ideas: sliding window (too expensive) → fully convolutional (downsampling & upsampling using average unpooling / max unpooling / transpose convolution)




**Classification + Localization**




**Object Detection**

Object detection as classification: sliding window (too expensive)

Region proposals: R-CNN, ~2k regions of interest → CNN, separated

Fast R-CNN, whole image → convolutional feature map → CNN

Faster R-CNN, insert Region Proposal Network (RPN) - faster region proposal

Detection without Proposals: YOLO (you only look once), SSD (single shot detector)

## 12. Visualizing and Understanding

Last hidden layer: nearest neighbors (vector distance) are likely to be in the same category — it captures some semantic content

→ last layer can be used to dimensionality reduction

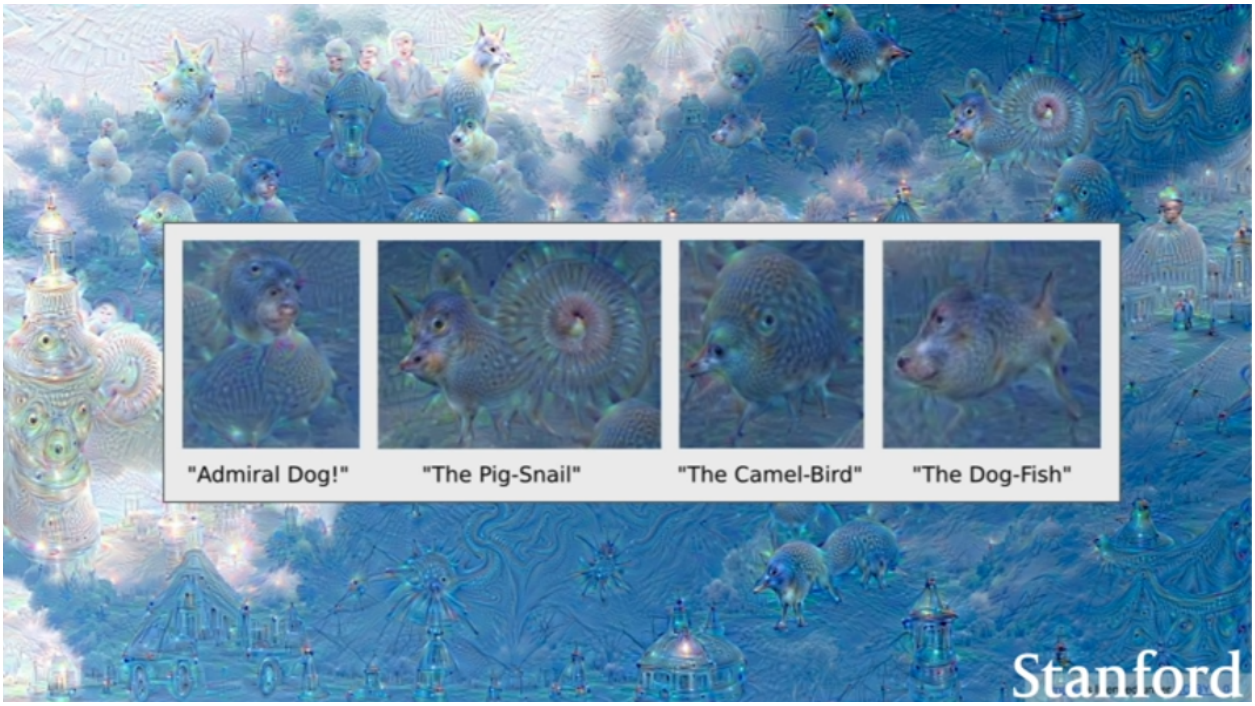
Occlusion experiments: occluding a part of the image every time, to determine which patches of the image contribute maximally to the output of a neural network

Visualizing CNN features - (guided) backprop: find the part of an image that a neuron responds to

Visualizing CNN features - gradient ascent: generate a synthetic image that maximally activates a neuron

Fooling images / adversarial examples: (1) Start from an arbitrary image (2) Pick an arbitrary class (3) Modify the image to maximize the class (4) Repeat until network is fooled

DeepDream - amplify existing features: rather than synthesizing an image to maximize a specific neuron, instead amplify the neuron activations at some layer in the network




Neural texture synthesis: reconstruct texture from higher layers recovers larger features from the input texture

Neural style transfer: content + style

## 13. Generative Models

- Unsupervised Learning
  - Generative Models
    - PixelRNN and PixelCNN
    - Variational Autoencoders (VAE)
    - Generative Adversarial Networks (GAN)

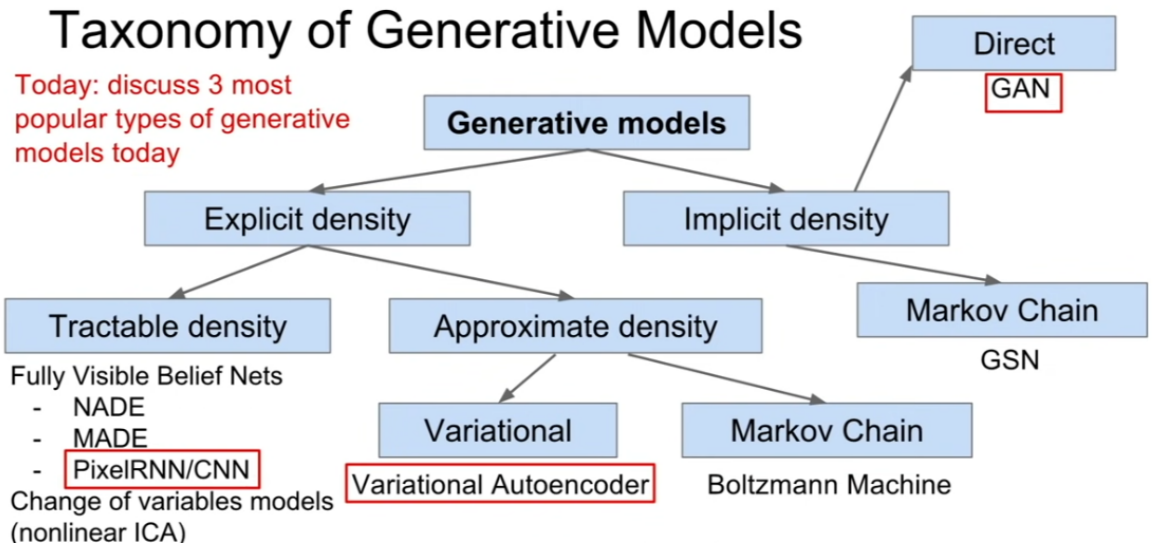
Unsupervised learning:

Data: just data, no label!

Goal: learn some underlying hidden structure of the data (supervised: learn a function x → y)

Examples: clustering, dimensionaity reduction, feture learning, density estimation

Generative models: given training data, generate new samples from same distribution



**PixelRNN and PixelCNN (2016)**

Fully visible belief network (explicit density model)

Use chain rule to decompose likelihood of an image x into product of 1-d distributions:

$p(x) = \Pi_{i=1}^{n} p(x_i | x_1, ..., x_{i-1})$

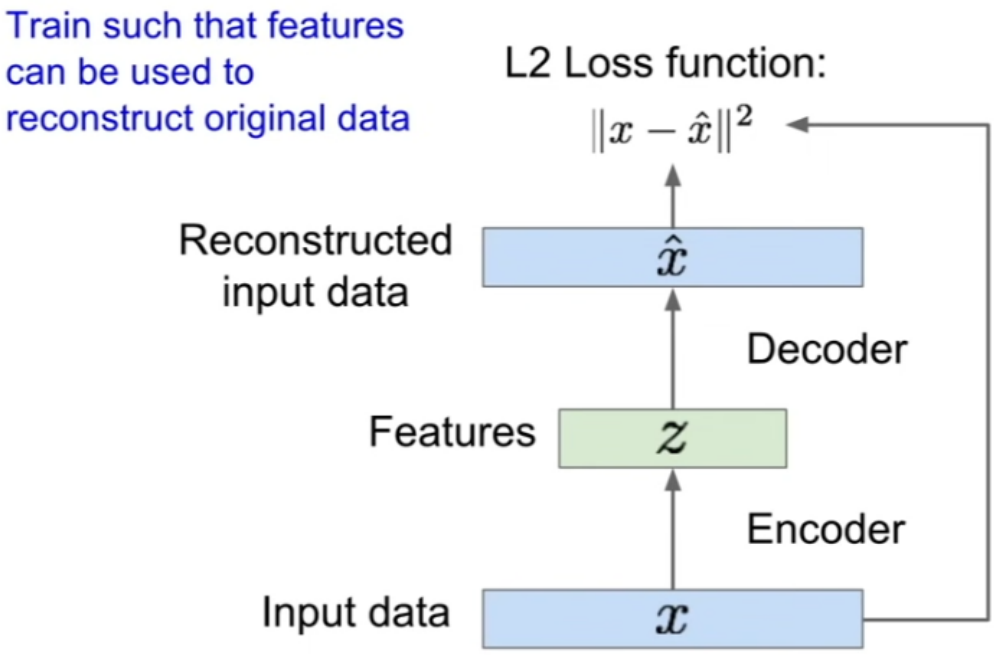
Then maximize likelihood of training images

PixelRNN: generate image pixels starting from corner, dependency on previous pixels modeled using an RNN (LSTM)

PixelCNN: generate image pixels starting from corner, dependency on previous pixels modeled using an CNN over context region, training faster

**Variational Autoencoders (VAE) (2014)**

Autoencoders: unsupervised approach for learning a lower-dimensional feature representation from unlabeled training data

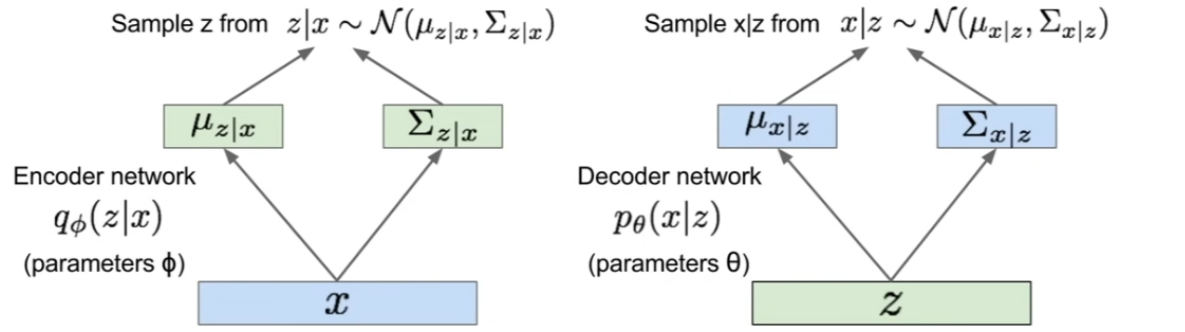



Autoencoders can reconstruct data, and can learn features to initialize a supervised model.

Probabilistic spin on autoencoders - will let us sample from the model to generate data!

# Variational Autoencoders

Since we're modeling probabilistic generation of data, encoder and decoder networks are probabilistic




Encoder and decoder networks also called "recognition"/"inference" and "generation" networks

Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Pro: allows inference of q(z|x), can be useful feature representation for other tasks

Cons: maximize lower bound of likelihood, not as good evaluation as PixelRNN/PixelCNN; samples blurrier and lower quality compared to state-of-the-art (GANs)

**Generative Adversarial Networks (GAN) (2014)**

Generator network: try to fool the discriminator by generating real-looking images

Discriminator networ: try to distinguish between real and fake images

## Training GANs: Two-player game

Ian Goodfellow et al., "Generative Adversarial Nets", NIPS 2014

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[ \mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Aside: Jointly training two networks is challenging, can be unstable. Choosing objectives with better loss landscapes helps training, is an active area of research.

Alternate between:

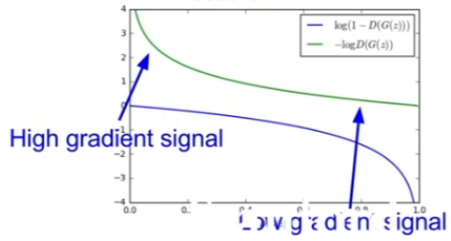
1. **Gradient ascent** on discriminator

$$\max_{\theta_d} \left[ \mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

2. **Instead: Gradient ascent** on generator, different objective

$$\max_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(D_{\theta_d}(G_{\theta_g}(z)))$$

Instead of minimizing likelihood of discriminator being correct, now maximize likelihood of discriminator being wrong.
Same objective of fooling discriminator, but now higher gradient signal for bad samples => works much better! Standard in practice.

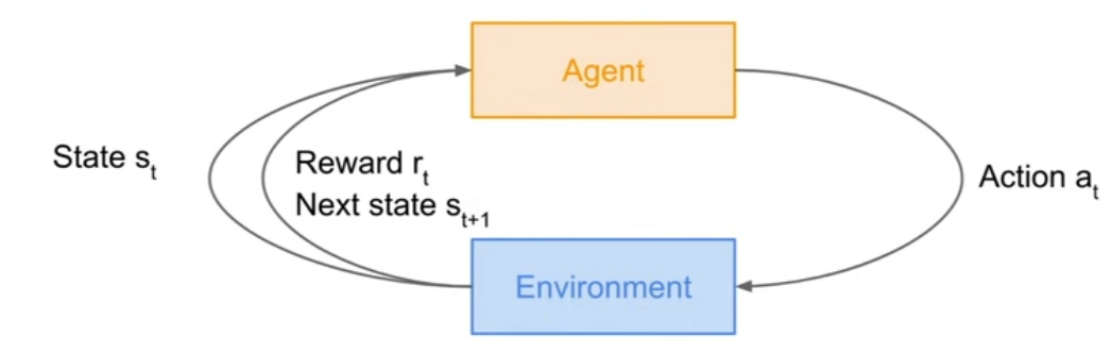



## 14. Deep RL

- What is Reinforcement Learning?
- Markov Decision Processes
- Q-Learning
- Policy Gradients

RL

- Q-Learning
- Policy Gradients




Markov Decision Process

- Mathematical formulation of the RL problem
- Markov property: current state completely characterises the state of the world

Defined by $(S, A, R, P, \gamma)$

$S$: set of possible states

$A$: set of possible actions

$R$: distribution of reward given (state, action) pair

$P$: transition probability

$\gamma$: discount factor

- A policy $\pi$ is a function from S to A that specifies what action to take in each state
- Objective: find optimal policy $\pi^*$ that maximizes $\Sigma_{t>0}\gamma^t r_t$

  Formally: $\pi^* = \arg\max E[\Sigma_{t\geq 0}\gamma^t r_t|\pi]$

How good is a state?

Value function: $V^\pi(s) = E[\Sigma_{t\geq 0}\gamma^t r_t|s_0 = s, \pi]$

How good ia s state-action pair?

Q-value function: $Q^\pi(s, a) = E[\Sigma_{t\geq 0}\gamma^t r_t|s_0 = s, a_0 = a, \pi]$

$Q^*(s, a) = \max_\pi E[\Sigma_{t\geq 0}\gamma^t r_t|s_0 = s, a_0 = a, \pi]$

$Q^*$ satisfies the Bellman equation: $Q^*(s,a) = E_{s' \sim \epsilon}[r + \gamma \max_{a'} Q^*(s',a')|s,a]$

$\pi^*$ corresponds to taking the best action in any state as specified by $Q^*$.

Value iteration: $Q_{i+1}(s,a) = E[r + \gamma \max_{a'} Q_i(s',a')|s,a]$

$Q_i$ will converge to $Q^*$ as $i \to \infty$.

Problem: must compute $Q(s,a)$ for every state-action pair.

Q-learning: use a function approximator $Q(s,a;\theta) \approx Q^*(s,a)$

If the function approximator is a deep neural network → deep q-learning

Remember: want to find a Q-function that satisfies the Bellman Equation:

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

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right]$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a\right]$

Iteratively try to make the Q-value close to the target value ($y_i$) it should have, if Q-function corresponds to optimal Q* (and optimal policy $\pi^*$)

Backward Pass

Gradient update (with respect to Q-function parameters θ):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot);s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i))\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

Policy Gradients

Q-function can be very complicated!

Define a class of parameterized policier: $\Pi = \{\pi_\theta, \theta \in R^m\}$

For each policy, define its value: $J(\theta) = E[\Sigma_{t \geq 0} \gamma^t r_t | \pi_\theta]$

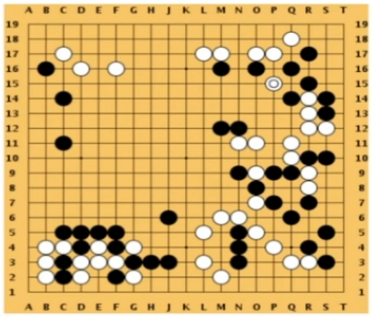
We want to find the optimal policy $\theta^* = \arg\max_\theta J(\theta)$

Actor-Critic Algorithm

We can combine Policy Gradients and Q-learning by training both an actor (the policy) and a critic (the Q-function).

- The actor decides which action to take, and the critic tells the actor how good its action was and how it should adjust.
- Alleviates the task of the critic as it only has to learn the values of state-action pairs generated by the policy

## More policy gradients: AlphaGo



**Overview:**

- Mix of supervised learning and reinforcement learning
- Mix of old methods (Monte Carlo Tree Search) and recent ones (deep RL)

**How to beat the Go world champion:**

- Featurize the board (stone color, move legality, bias, …)
- Initialize policy network with supervised training from professional go games, then continue training using policy gradient (play against itself from random previous iterations, +1 / -1 reward for winning / losing)
- Also learn value network (critic)
- Finally, combine combine policy and value networks in a Monte Carlo Tree Search algorithm to select actions by lookahead search

*[Silver et al., Nature 2016]*

## 15. Efficient Methods and Hardware for DL

Hardware 101: the family

General purpose: CPU, GPU

Specialized HW: FPGA, ASIC

Part 1: Algorithm for Efficient Inference

1. Pruning (iteratively prune & retrain to retain accuracy)
2. Weight sharing
3. Quantization (choose a proper radix point position)
4. Low rank approximation (decompose a convolutional layer to two layers)

5. Binary / ternary net (only use two or three weights to represent the network)
6. Winograd transformation

Part 2: Hardware for Efficient Inference

A common goal: minimize memory access

Part 3: Algorithm for Efficient Training

1. Parallelization (data parallel / model parallel)
2. Mixed precision with FP16 and FP32 (floating point)
3. Model distillation (teacher model → (knowledge) → student model)
4. DSD: Dense-Sparse-Dense training (prune & re-dense)

Part 4: Hardware for Efficient Training

GPU / TPU for training

## 16. Adversarial Examples and Adversarial Training (Ian Goodfellow)

对抗样本（Adversarial examples）是指在数据集中通过故意添加细微的干扰所形成的输入样本
，会导致模型以高置信度给出一个错误的输出。 而人类观察者不会察觉原始样本和对抗样本
之间的差异。

Adversarial examples from overfitting ❌

Adversarial examples from underfitting / excessive linearity

- modern deep nets are very piecewise linear
- nealy linear responses in practice
- small inter-class distances

Fast Gradient Sign Method (FGSM): to add the noise (not random noise) whose direction is the same as the gradient of the cost function with respect to the data

ML has cross-model, cross-dataset generalization → vulnerable to same adverserial examples → transferability attack target model with substitute model

Enhancing transfer with ensembles

Generative modeling is not sufficient to solve the problem

Adversarial training: training on adversarial examples

Conlusion

- Attacking is easy
- Defending is difficult
- Adversarial training provides regularization and semi-supervised learning
- The out-of-domain input problem is a bottleneck for model-based optimization generally