CS60020: Foundations of Algorithm Design and Machine Learning

Sourangshu Bhattacharya

In this Lecture:

- Outline:
  - Stochastic gradient descent (SGD)
  - SGD Convergence
  - Minibatch and Distributed SGD
  - Practical considerations
  - Advancements from SGD.

Much of ML is optimization

Linear Classification

$$\arg\min_{w} \sum_{i=1}^{n} ||w||^2 + C \sum_{i=1}^{n} \xi_i$$
$$\text{s.t. } 1 - y_i x_i^T w \leq \xi_i$$
$$\xi_i \geq 0$$

Maximum Likelihood

$$\arg\max_{\theta} \sum_{i=1}^{n} \log\, p_\theta(x_i)$$

K-Means

$$\arg \min_{\mu_1, \mu_2, \ldots, \mu_k} J(\mu) = \sum_{j=1}^{k} \sum_{i \in C_j} ||x_i - \mu_j||^2$$

Stochastic optimization

- Goal of machine learning :
  - Minimize expected loss

$$\min_h L(h) = \mathbf{E}\left[\text{loss}(h(x), y)\right]$$

given samples $(x_i, y_i)\ i = 1, 2 \ldots m$

- This is Stochastic Optimization
  - Assume loss function is convex

Batch (sub)gradient descent for ML

- Process all examples together in each step

$$w^{(k+1)} \leftarrow w^{(k)} - \eta_t \left(\frac{1}{n} \sum_{i=1}^{n} \frac{\partial L(w, x_i, y_i)}{\partial w} \right)$$

where L is the regularized loss function

- Entire training set examined at each step
- Very slow when *n* is very large

Stochastic (sub)gradient descent

- "Optimize" one example at a time
- Choose examples randomly (or reorder and choose in order)
  - Learning representative of example distribution

```
for $i = 1$ to $n$:
    $w^{(k+1)} \leftarrow w^{(k)} - \eta_t \frac{\partial L(w, x_i, y_i)}{\partial w}$
```

where L is the regularized loss function

Stochastic (sub)gradient descent

```
for $i = 1$ to $n$:
    $w^{(k+1)} \leftarrow w^{(k)} - \eta_t \frac{\partial L(w, x_i, y_i)}{\partial w}$
```

where L is the regularized loss function

- Equivalent to online learning (the weight vector *w* changes with every example)
- Convergence guaranteed for convex functions (to local minimum)

Stochastic gradient descent

- Given dataset $D = \{(x_1, y_1), \dots, (x_m, y_m)\}$
- Loss function: $L(\theta, D) = \frac{1}{N} \sum_{i=1}^{N} l(\theta; x_i, y_i)$
- For linear models: $l(\theta; x_i, y_i) = l(y_i, \theta^T \phi(x_i))$
- Assumption D is drawn IID from some distribution $\mathcal{P}$.
- Problem:

$$\min_{\theta} L(\theta, D)$$

Stochastic gradient descent

- Input: D
- Output: $\bar{\theta}$

Algorithm:

- Initialize θ^0
- For $t = 1, \dots, T$

$$\theta^{t+1} = \theta^t - \eta_t \nabla_\theta l(y_t, \theta^T \phi(x_t))$$

- $\bar{\theta} = \frac{\sum_{t=1}^T \eta_t \theta^t}{\sum_{t=1}^T \eta_t}$.

SGD convergence

- Expected loss: $s(\theta) = E_{\mathcal{P}}[l(y, \theta^T \phi(x)]$
- Optimal Expected loss: $s^* = s(\theta^*) = \min_{\theta} s(\theta)$
- Convergence:

$$E_{\bar{\theta}}[s(\bar{\theta})] - s^* \leq \frac{R^2 + L^2 \sum_{t=1}^{T} \eta_t^2}{2 \sum_{t=1}^{T} \eta_t}$$

- Where: $R = \|\theta^0 - \theta^*\|$
- $L = \max \nabla l(y, \theta^T \phi(x))$

SGD convergence proof

- Define $r_t = \|\theta^t - \theta^*\|$ and $g_t = \nabla_\theta l\left(y_t, \theta^T \phi(x_t)\right)$
- $r_{t+1}^2 = r_t^2 + \eta_t^2 \|g_t\|^2 - 2\eta_t (\theta^t - \theta^*)^T g_t$
- Taking expectation w.r.t $\mathcal{P}, \bar{\theta}$ and using $s^* - s(\theta^t) \geq g_t^T(\theta^* - \theta^t)$, we get:
  $E_{\bar{\theta}}[r_{t+1}^2 - r_t^2] \leq \eta_t^2 L^2 + 2\eta_t(s^* - E_{\bar{\theta}}[s(\theta^t)])$

- Taking sum over $t = 1, \ldots, T$ and using

$$E_{\bar{\theta}}[r_{t+1}^2 - r_0^2] \leq L^2 \sum_{t=0}^{T-1} \eta_t^2 + 2 \sum_{t=0}^{T-1} \eta_t (s^* - E_{\bar{\theta}}[s(\theta^t)])$$

SGD convergence proof

- Using convexity of s:

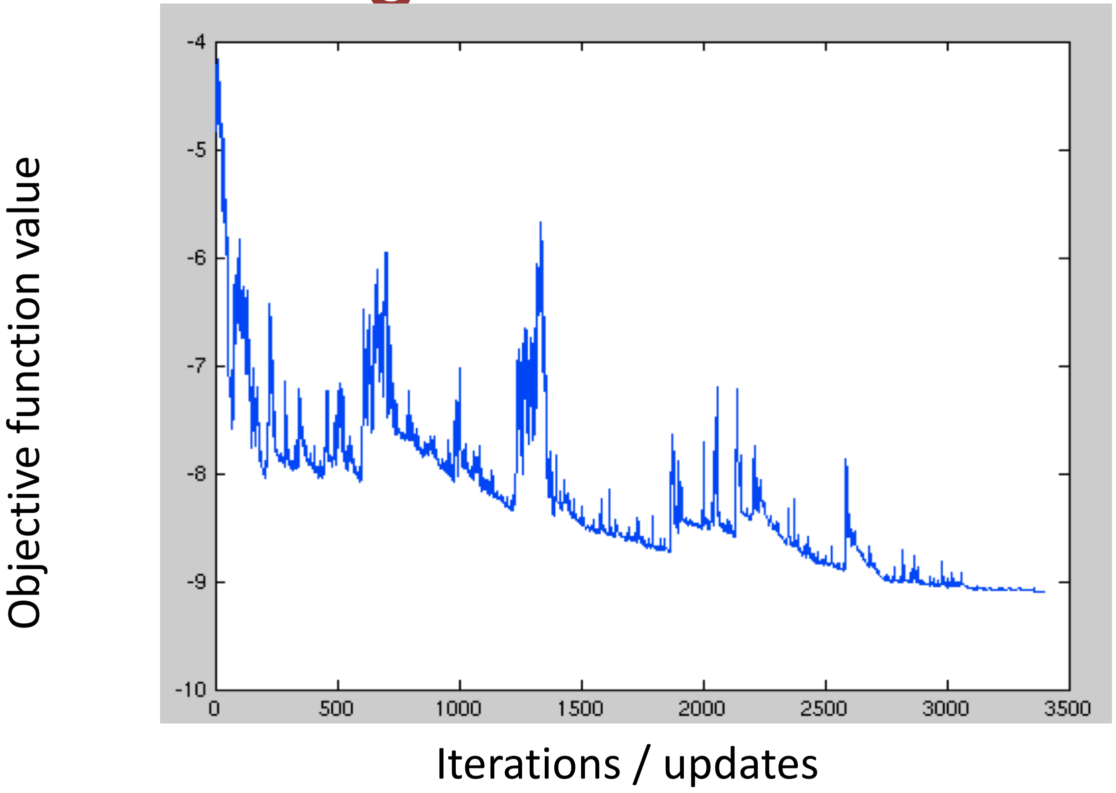
$$\left(\sum_{t=0}^{T-1} \eta_t\right) E_{\bar{\theta}}\,[s(\bar{\theta})] \leq E_{\bar{\theta}}[\sum_{t=0}^{T-1} \eta_t s(\theta^t)]$$

- Substituting in the expression from previous slide:

$$E_{\bar{\theta}}[r_{t+1}^2 - r_0^2] \leq L^2 \sum_{t=0}^{T-1} \eta_t^2 + 2 \sum_{t=0}^{T-1} \eta_t (s^* - E_{\bar{\theta}}[s(\bar{\theta})])$$

- Rearranging the terms proves the result.

SGD convergence



SGD - Issues

- Convergence very sensitive to learning rate (η_t) (oscillations near solution due to probabilistic nature of sampling)
  - Might need to decrease with time to ensure the algorithm converges eventually
- Basically – SGD good for machine learning with large data sets!

Mini-batch SGD

- Stochastic – 1 example per iteration
- Batch – All the examples!
- Mini-batch SGD:
  - Sample *m* examples at each step and perform SGD on them
- Allows for parallelization, but choice of *m* based on heuristics

Example: Text categorization

- **Example by Leon Bottou:**
  - **Reuters RCV1** document corpus
    - Predict a category of a document
      - One **vs.** the rest classification
  - ***n* = 781,000** training examples (documents)
  - 23,000 test examples
  - ***d* = 50,000** features
    - One feature per word
    - Remove stop-words
    - Remove low frequency words

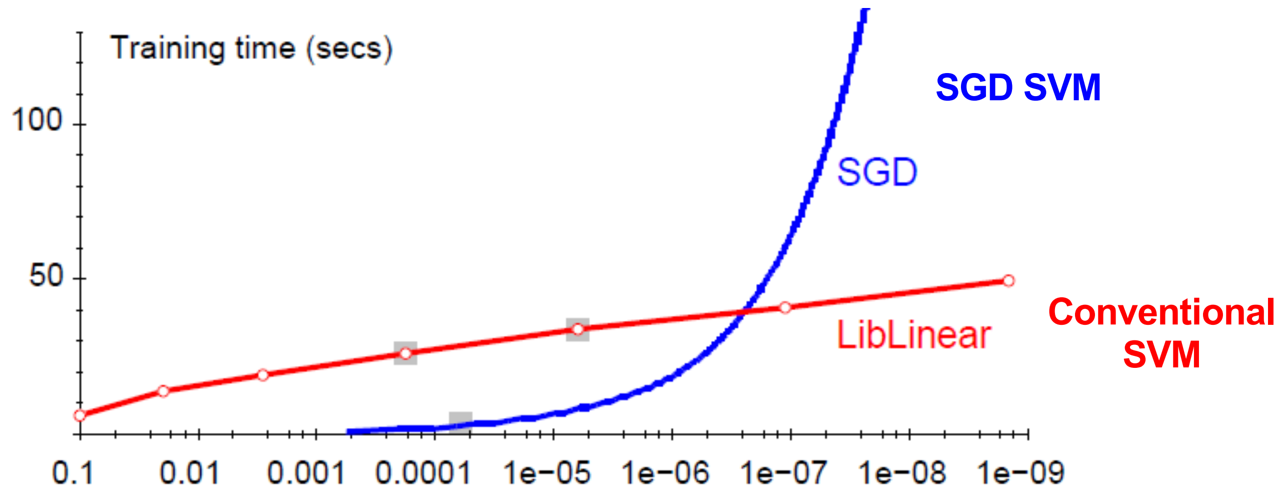
Example: Text categorization

- **Questions:**
  - **(1)** Is **SGD** successful at minimizing ***f(w,b)***?
  - **(2)** How quickly does **SGD** find the min of ***f(w,b)***?
  - **(3)** What is the error on a test set?

	Training time	***Value of f(w,b)***	***Test error***
Standard SVM	23,642 secs	*0.2275*	*6.02%*
"Fast SVM"	66 secs	0.2278	6.03%
SGD SVM	1.4 secs	*0.2275*	*6.02%*

(1) SGD-SVM is successful at minimizing the value of ***f(w,b)***
(2) SGD-SVM is super fast
(3) SGD-SVM test set error is comparable

Optimization “Accuracy”

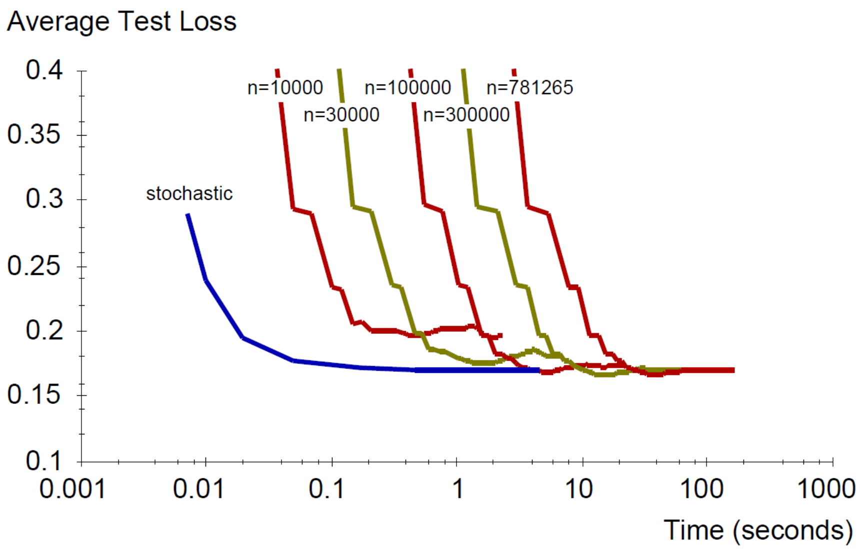


Optimization quality: | *f(w,b)* – *f* (*w*^{opt},*b*^{opt}) |

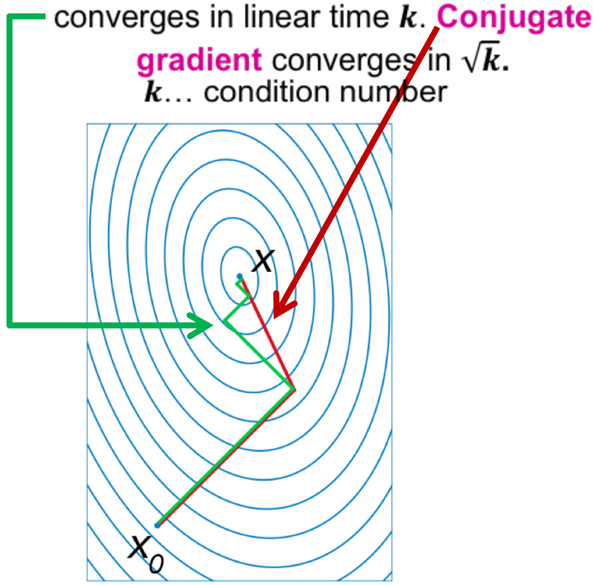
For optimizing *f(w,b) within reasonable* quality
SGD-SVM is super fast

SGD vs. Batch Conjugate Gradient

- **SGD** on full dataset vs. **Conjugate Gradient** on a sample of ***n*** training examples



Theory says: Gradient descent converges in linear time k. **Conjugate gradient** converges in $\sqrt{k}$.
k… condition number



Bottom line: Doing a simple (but fast) SGD update many times is better than doing a complicated (but slow) CG update a few times

Practical Considerations

- **Need to choose learning rate η and t_0**

$$w_{t+1} \leftarrow w_t - \frac{\eta_t}{t+t_0}\left(w_t + C\frac{\partial L(x_i, y_i)}{\partial w}\right)$$

- **Leon suggests:**
  - Choose $\mathbf{t_0}$ so that the expected initial updates are comparable with the expected size of the weights
  - Choose **η**:
    - Select a **small subsample**
    - Try various rates **η** (e.g., 10, 1, 0.1, 0.01, …)
    - Pick the one that most reduces the cost
    - Use **η** for next 100k iterations on the full dataset

Practical Considerations

- **Sparse Linear SVM:**
  - **Feature vector x_i is sparse (contains many zeros)**
    - Do not do: $\boldsymbol{x_i}$ = [*0,0,0,1,0,0,0,0,5,0,0,0,0,0,0,…*]
    - But represent $\mathbf{x_i}$ as a sparse vector $\mathbf{x_i}$**=[*(4,1), (9,5), …*]**
  - **Can we do the SGD update more efficiently?**

$$w \leftarrow w - \eta\left(w + C\frac{\partial L(x_i, y_i)}{\partial w}\right)$$

  - **Approximated in 2 steps:**

$$w \leftarrow w - \eta C\frac{\partial L(x_i, y_i)}{\partial w}$$

cheap: $\boldsymbol{x_i}$ is sparse and so few coordinates ***j*** of ***w*** will be updated

$$w \leftarrow w(1-\eta)$$

expensive: ***w*** is not sparse, all coordinates need to be updated

Practical Considerations

- **Solution 1:** $\boldsymbol{w} = \boldsymbol{s} \cdot \boldsymbol{v}$
  - Represent vector ***w*** as the product of scalar ***s*** and vector ***v***
  - Then the update procedure is:
    - *(1)* $\boldsymbol{v} = \boldsymbol{v} - \boldsymbol{\eta C} \frac{\partial L(x_i, y_i)}{\partial w}$
    - *(2)* $\boldsymbol{s} = \boldsymbol{s}(1 - \boldsymbol{\eta})$
- **Solution 2:**
  - Perform only step **(1)** for each training example
  - Perform step **(2)** with lower frequency and higher $\boldsymbol{\eta}$

Two step update procedure:

(1) $w \leftarrow w - \eta C \frac{\partial L(x_i, y_i)}{\partial w}$

(2) $w \leftarrow w(1 - \eta)$

Practical Considerations

- Stopping criteria:

  How many iterations of SGD?

  - Early stopping with cross validation
    - Create a validation set
    - Monitor cost function on the validation set
    - Stop when loss stops decreasing
  - Early stopping
    - Extract two disjoint subsamples **A** and **B** of training data
    - Train on **A**, stop by validating on **B**
    - Number of epochs is an estimate of ***k***
    - Train for ***k*** epochs on the full dataset

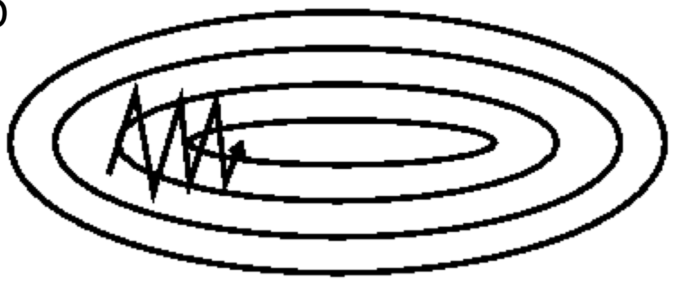
Stochastic gradient descent

- Idea: Perform a parameter update for each training example *x(i)* and label *y(i)*
- Update: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x(i), y(i))$
- Performs redundant computations for large datasets

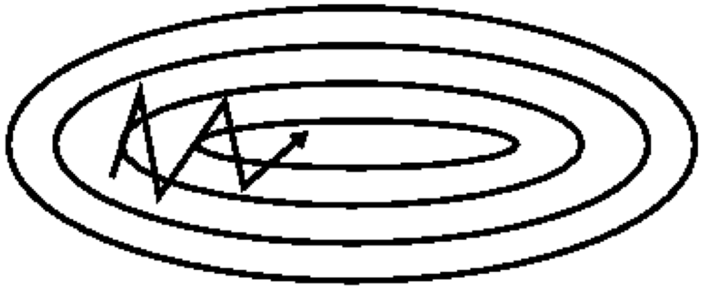
Momentum gradient descent

- Idea: Overcome ravine oscillations by momentum

SGD



SGD with momentum



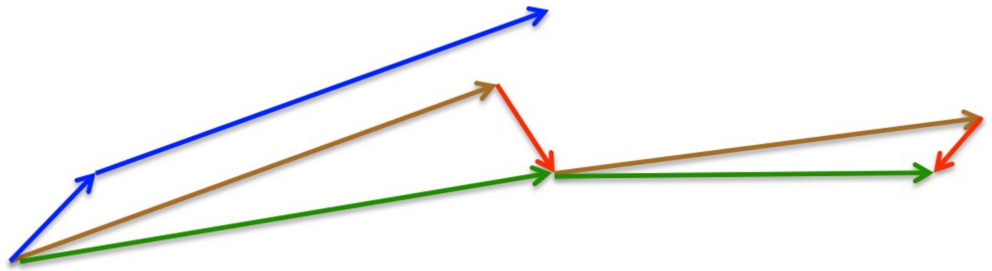
Update:

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

Nesterov accelerated gradient

- Ideas:
  1. Big jump in the direction of the previous accumulated gradient & measure the gradient

  2. Then make a correction.



- Update:

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

  - $\theta = \theta - v_t$

RMSprop

- Idea: Use the second moment of gradient vector to estimate the magnitude of update in a given direction.

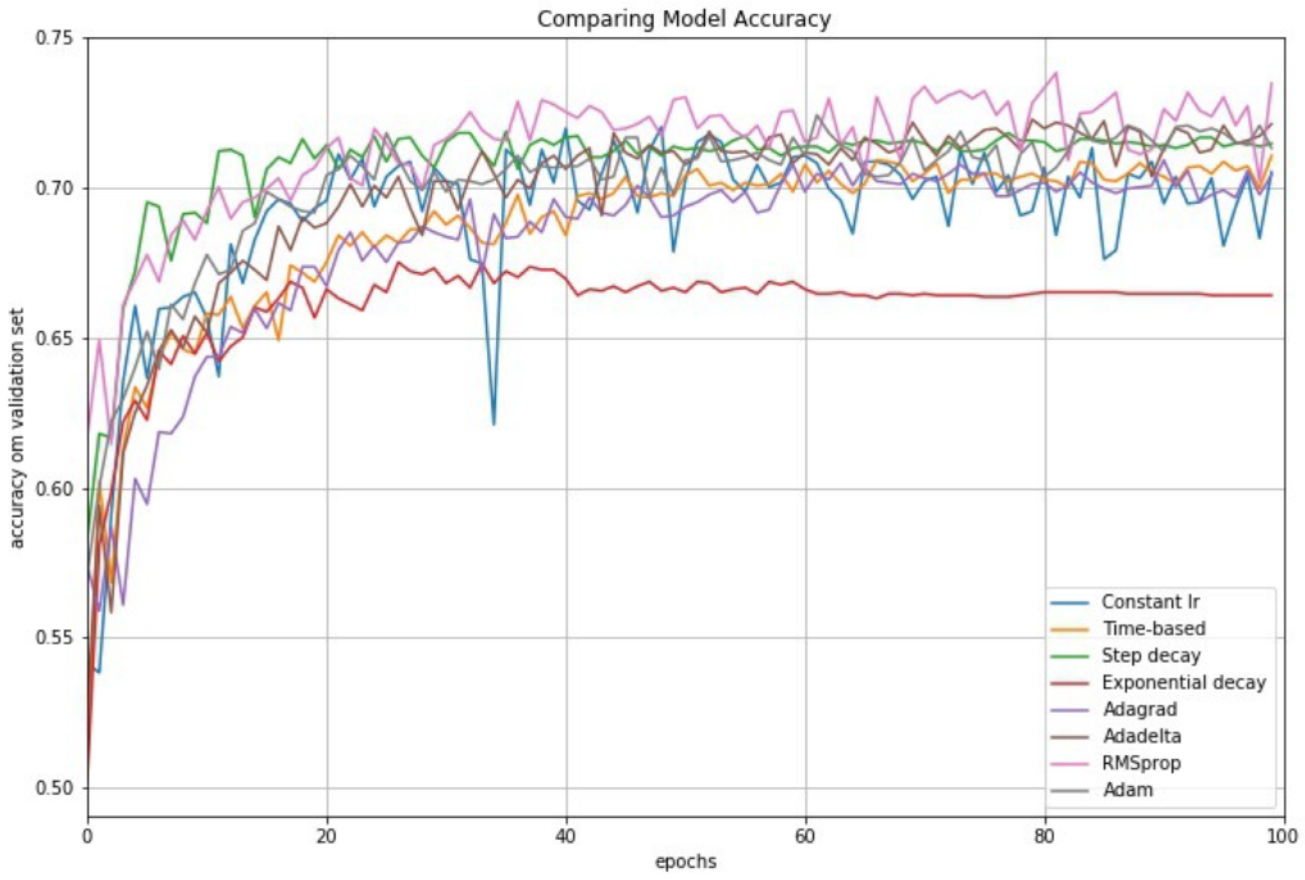
Update:

- $E[g^2]_t = 0.9\ E[g^2]_{t-1} + 0.1\ g_t^2$
- $\Delta\theta_t = -\eta / \sqrt{(E[g^2]_t + \epsilon)} \odot g_t$

ADAM (Adaptive moment)

- Idea: In addition to storing an exponentially decaying average of past squared gradients like RMSprop, Adam also keeps an exponentially decaying average of past gradients.
- Updates:
  - $m_t = \beta_1 m_{t-1} + (1-\beta_1)\, g_t$
  - $v_t = \beta_2 v_{t-1} + (1-\beta_2)\, g_t^2$
  - $\hat{m}_t = m_t / (1 - \beta_1^t)$
  - $\hat{v}_t = v_t / (1 - \beta_2^t)$
  - $\vartheta_{t+1} = \vartheta_t - (\, \eta / (\, \sqrt{\hat{v}_t} + \epsilon\,)\,)\, \hat{m}_t$

Enhancements comparison



References:

- SGD proof by Yuri Nesterov.
- MMDS http://www.mmds.org/
- *Blog of Sebastian Ruder http://ruder.io/optimizing-gradient-descent/*
- *Learning rate comparison https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1*

Motivation

- Supervised Learning needs a lot of labelled data.
- Finding labelled examples is
  - Difficult – Rumour or not ?
  - Expensive – Imagenet cost.

- Humans don’t always need handholding.
  - Trained mind finds it easier to learn related concepts.

- Learning is a continuous process, adapting to changed scenario.

Salvation

- Use unlabeled examples from the same problem / dataset :
  - Semi-supervised learning
  - Active learning
- Use labelled examples from other / related domains / datasets:
  - Transfer learning or Multi-task learning.
- Have access to an environment where one can take some actions and observe rewards :
  - Reinforcement learning.

Reinforcement learning

- Access to environment, gives freedom to "explore" along the boundaries – driving off the road.
- More "difficult" than standard supervised learning,
  - Non-stationarity (Ad-serving on budget)
  - Sequential decision making (planning)
  - Interactive scenarios (chatbot).

Active Learning

Semi-supervised + Active learning

- Early methods for semi-supervised learning:
  - Transductive / Inductive
  - Graph – label propagation vs vector space – SSSVM.
- Active learning:
  - 2 recent approaches – Generalization error based and model weight distinctiveness based.
- A new direction: estimating accuracy from unlabeled data.
- Theory: When does label propagation fail ?

Active Learning

- *The key idea behind active learning is that a machine learning algorithm can achieve greater accuracy with fewer training labels if it is allowed to choose the data from which it learns. An active learner may pose queries, usually in the form of unlabeled data instances to be labeled by an oracle (e.g., a human annotator).*

  [Settles, 2012]

- *"what is the optimal way to choose data points to label such that the highest accuracy can be obtained given a fixed labeling budget."*

  [Sener & Savarese, 2018]

- Select the optimal set of unlabeled data to annotate within a limited budget

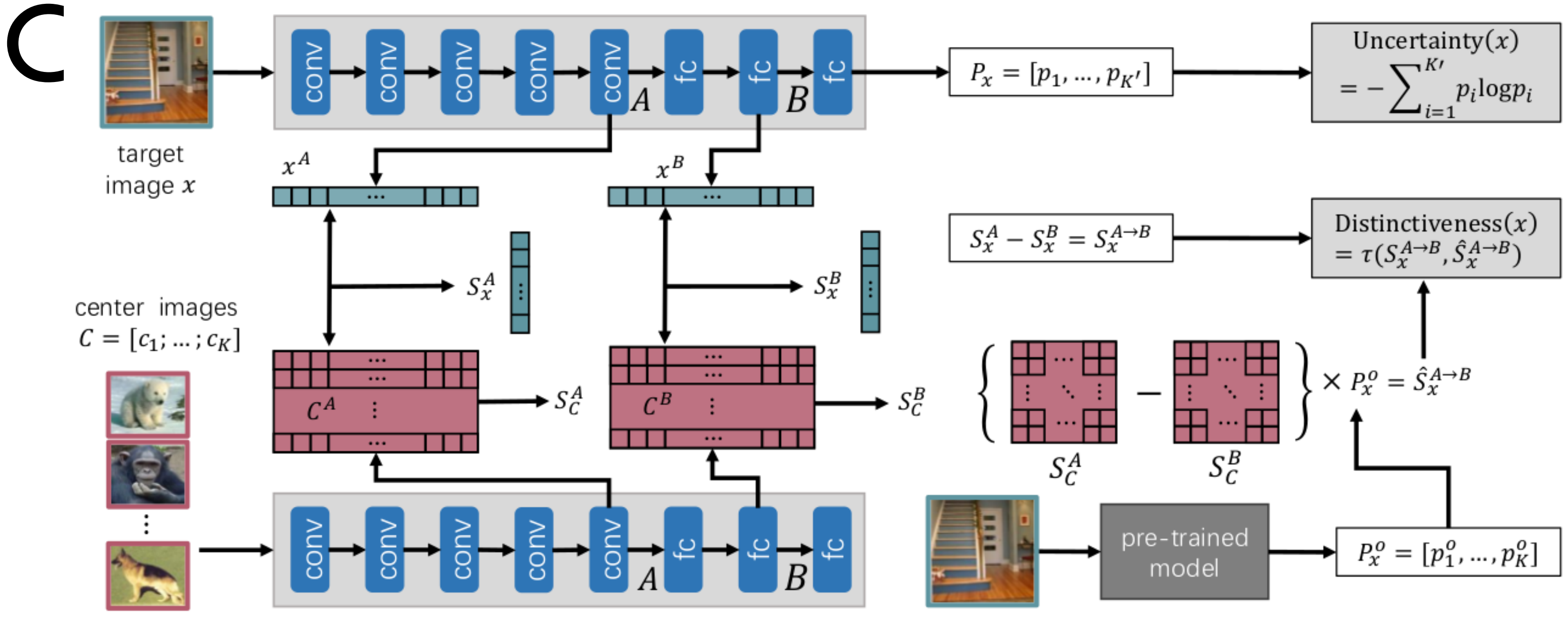
- Perform pre-training using a similar task

2. Cost-effective training of Deep CNNs actively

- Adapting a pretrained model to a new task
- Proposes a general framework of active model adaptation for deep CNNs
- Actively querying data points to label
    - Proposes a novel criterion for selection which best optimizes the feature representation along with the classifier performance
    - Proposes an algorithm that can actively sellect instances to achieve better feature representation

Huang et. al, "Cost-effective Training of Deep CNNs with Active Model Adaptation", KDD 2018

2. Cost-effective training of Deep C



target image x
conv
fc
A
B
x^A
x^B
S_x^A
S_x^B
center images $C = [c_1; \dots; c_K]$
C^A
C^B
S_C^A
S_C^B
$P_x = [p_1, \dots, p_{K'}]$
$\text{Uncertainty}(x) = -\sum_{i=1}^{K'} p_i \log p_i$
$S_x^A - S_x^B = S_x^{A\to B}$
$\text{Distinctiveness}(x) = \tau(S_x^{A\to B}, \hat{S}_x^{A\to B})$
$\times P_x^o = \hat{S}_x^{A\to B}$
pre-trained model
$P_x^o = [p_1^o, \dots, p_K^o]$

Huang et. al, "Cost-effective Training of Deep CNNs with Active Model Adaptation", KDD 2018

Multitask Learning

Multi-task & Transfer learning

- Transfer learning: Already learned model is “adapted” to new task.
- Multi-task learning: The models for multiple tasks are learned simultaneously. Overall performance improves.

Introduction & Motivation

- Machine Learning tasks
  - train a single model or an ensemble of models
  - fine -tune / tweak the models

- By being focussed on one task, we ignore information coming from related tasks, that may be helpful

- By sharing representations between related tasks, we can enable our model to generalize better on our original task

- “MTL improves generalization by leveraging the domain-specific information contained in the training signals of related tasks” [Caruana et.al] [1]

[1] Multitask learning: A knowledge-based source of inductive bias, ICML 1993

Introduction

Formal definition :
Given m learning tasks $\{T_i\}_{1<=i<=m}$ where all the tasks or a subset of them are related, multi-task learning aims to help improve the learning of a model for T_i by using the knowledge contained in all or some of the m tasks

- When <u>different tasks</u> share the <u>same training data</u> samples, MTL reduces to multi-label learning or multi-output regression
- Homogeneous-feature MTL ➔ different tasks lie in the same feature space
- Heterogeneous-feature MTL ➔ different tasks lie in different feature space
- Heterogeneous MTL ➔ different types of supervised tasks

Introduction

- **When to share**

  → make choices between single-task & multi-task models

  → Currently such decision is made by human experts (model selection)

- **What to share →** feature, instance, parameter

- Feature sharing : learn common features among different tasks as a way to share knowledge

  → based on shallow or deep models

  → learned common feature representation can be a <u>subset</u> or a <u>transformation</u> of the original feature representation

Introduction

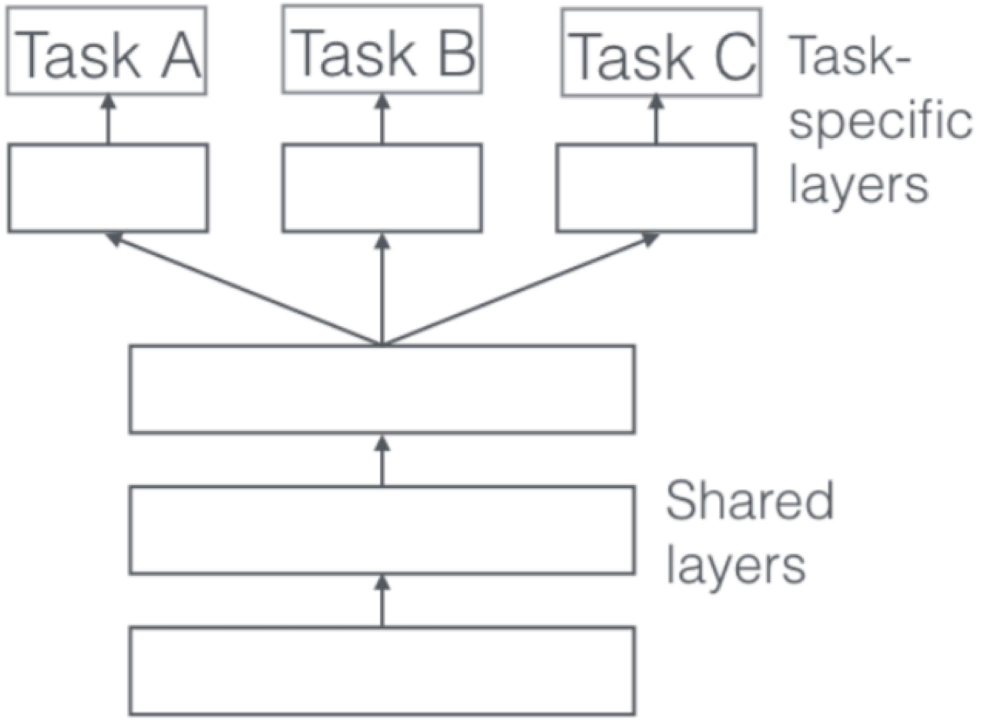
- Instance based : identify useful data instances in a task for other tasks and then shares knowledge via the identified instances

- Parameter-based MTL : uses model parameters in a task to help learn model parameters in other tasks
  → lowrank approach
  → task clustering approach
  → task relation learning approach
  → decomposition approach

Introduction

- Lowrank : interprets the relatedness of multiple tasks as the low rankness of the parameter matrix of these tasks
- Task clustering approach : assumes that all the tasks form a few clusters where tasks in a cluster are related to each other
- Task relation learning : learn quantitative relations between tasks from data automatically
- Decomposition approach : decomposes the model parameters of all the tasks into two or more components, which are penalized by different regularizers

Hard Parameter Sharing

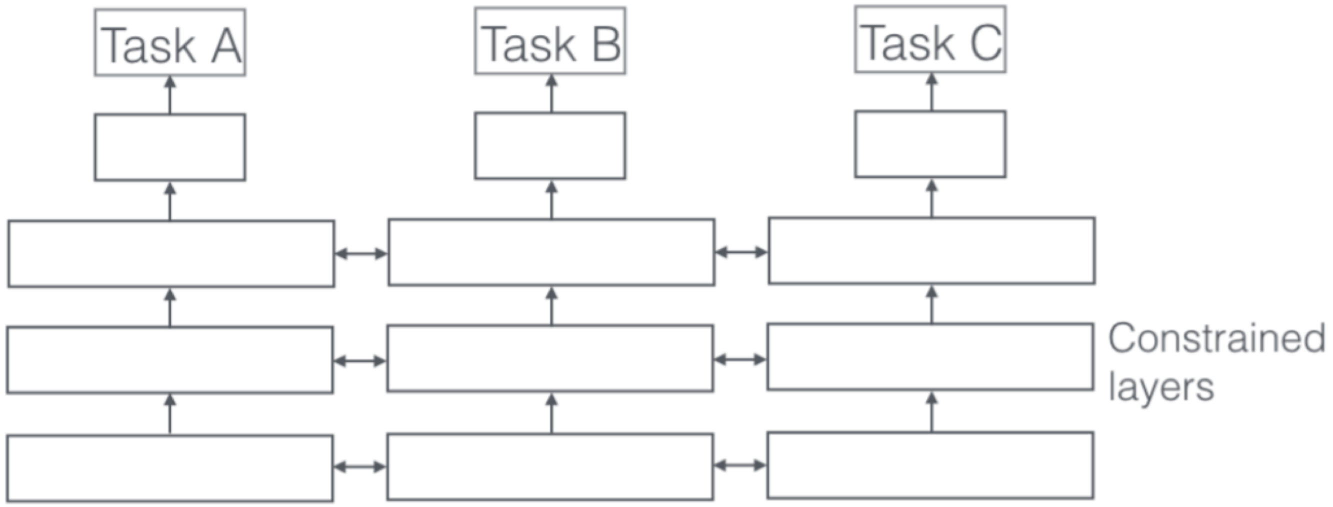


- sharing the hidden layers between all tasks, while keeping several task-specific output layers

- Reduces the risk of overfitting.
  - The more tasks we are learning simultaneously ⇒ find a representation that captures all of the tasks ⇒ less is our chance of overfitting on our original task.

Soft Parameter Sharing

- each task has its own model with its own parameters.
- The distance between the parameters of the model are regularized in order to encourage the parameters to be similar.



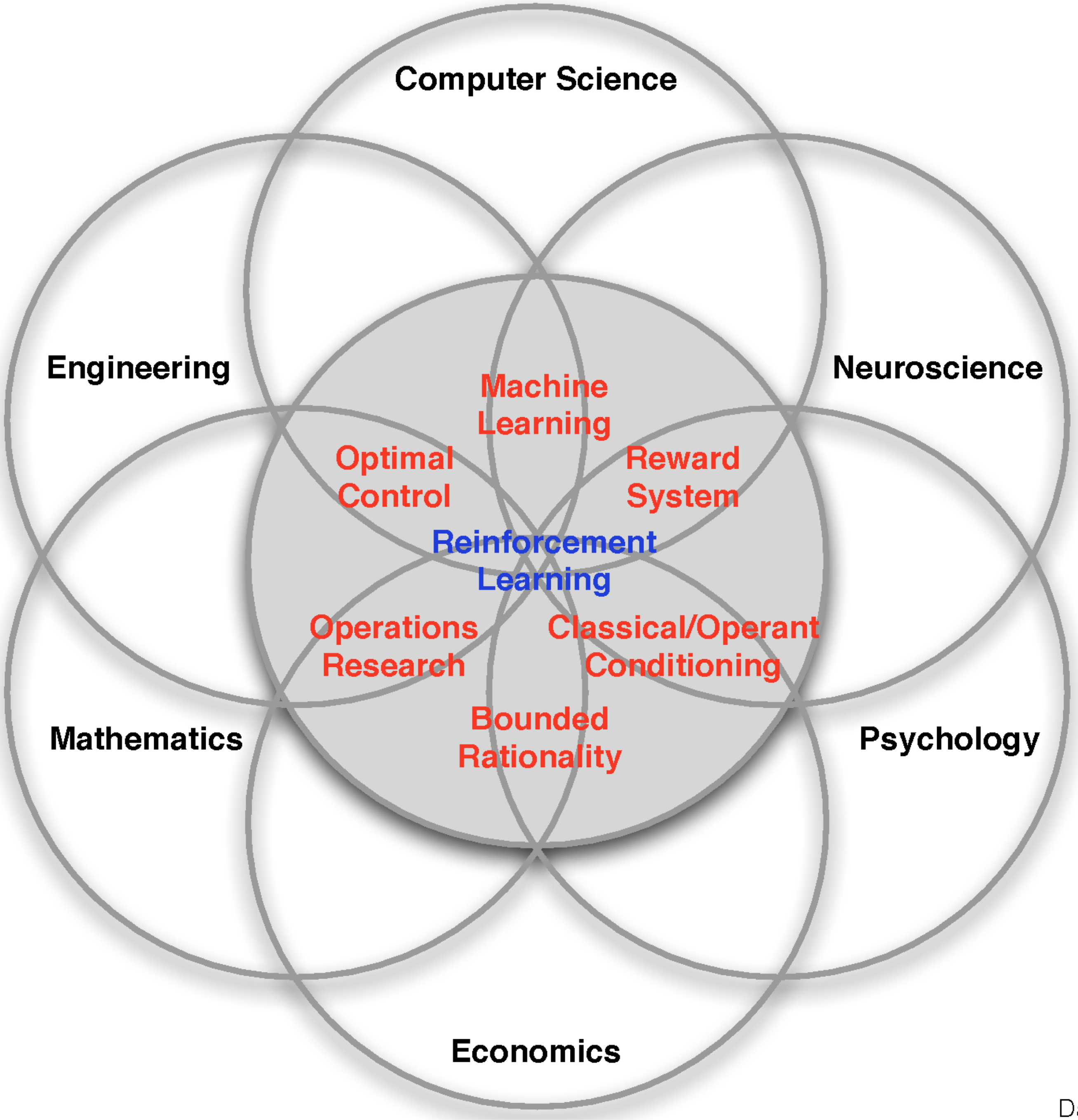
Reinforcement Learning

List of landmark papers

- Deep RL:
  - DQN paper (2015): http://www.nature.com/articles/nature14236
  - A3C paper (2016): https://arxiv.org/abs/1602.01783
  - AlphaGo paper (2016): http://www.nature.com/articles/nature16961
- Imitation Learning:
  - End to end learning for self-driving cars, Bojarski *et al*, 2016

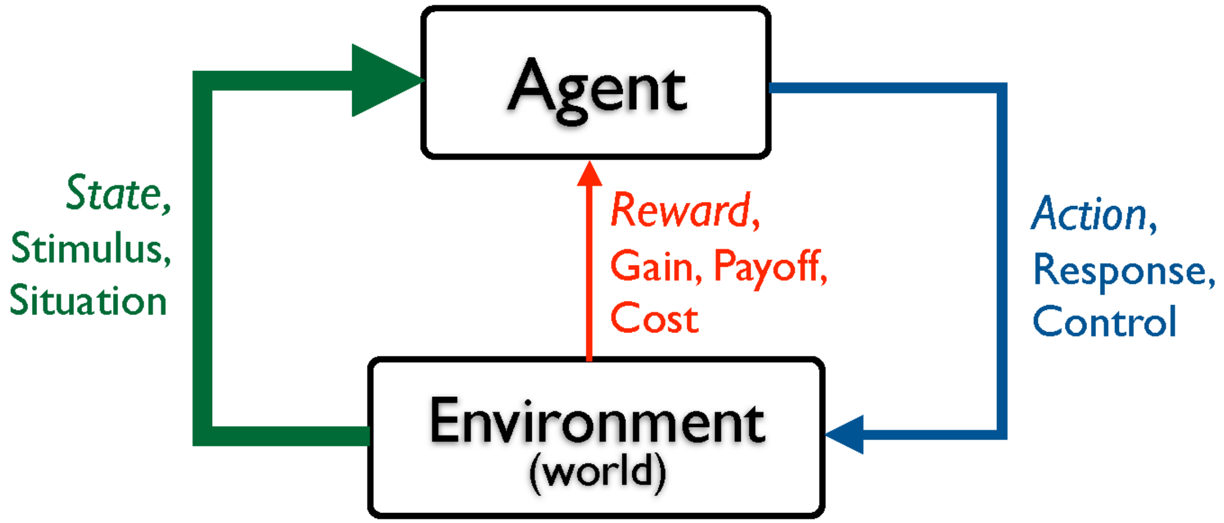
What is Reinforcement Learning?

- Agent-oriented learning—learning by interacting with an environment to achieve a goal
  - more realistic and ambitious than other kinds of machine learning
- Learning by trial and error, with only delayed evaluative feedback (reward)
  - the kind of machine learning most like natural learning
  - learning that can tell for itself when it is right or wrong
- The beginnings of a *science of mind* that is neither natural science nor applications technology



David Silver 2015

The RL Interface



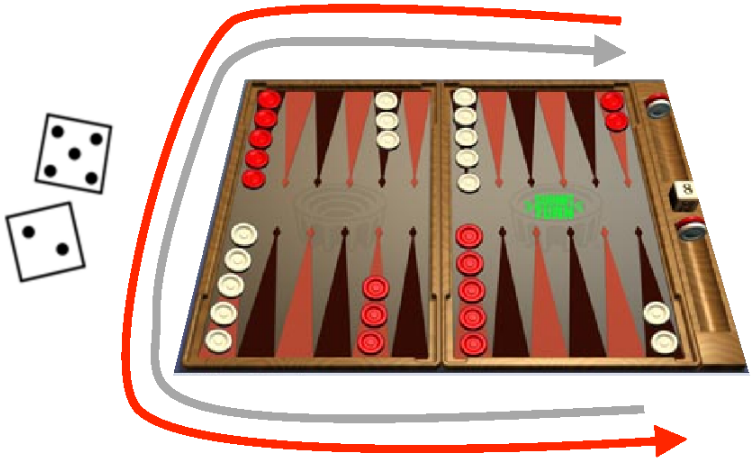
- Environment may be unknown, nonlinear, stochastic and complex
- Agent learns a policy mapping states to actions
    - Seeking to maximize its cumulative reward in the long run

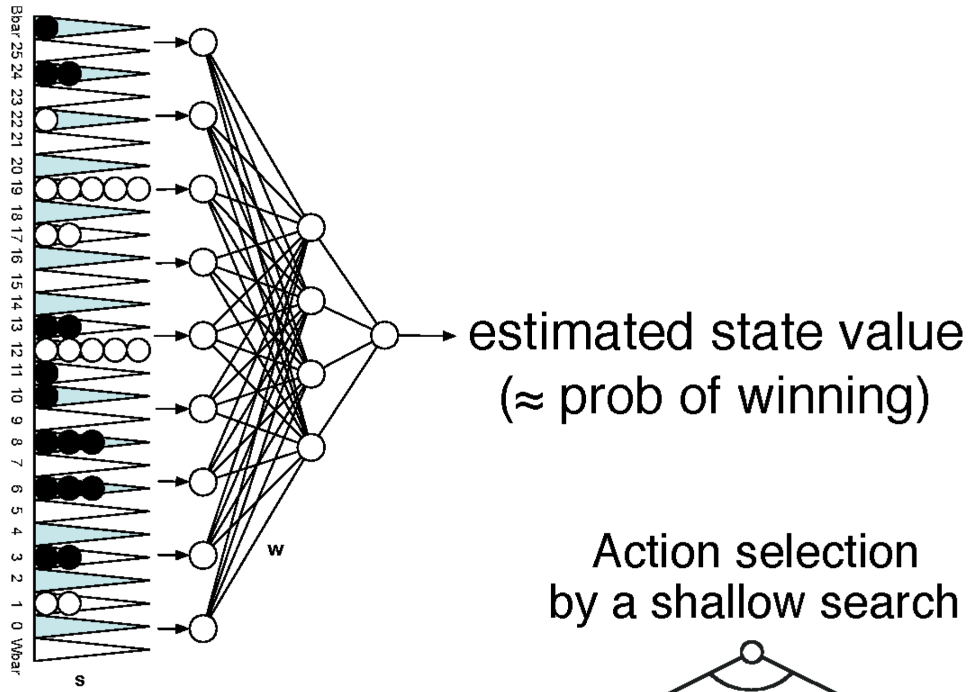
Some RL Successes

- Learned the world's best player of Backgammon (Tesauro 1995) →
- Learned acrobatic helicopter autopilots (Ng, Abbeel, Coates et al 2006+)
- Widely used in the placement and selection of advertisements and pages on the web (e.g., A-B tests)
- Used to make strategic decisions in *Jeopardy!* (IBM's Watson 2011)
- Achieved human-level performance on Atari games from pixel-level visual input, in conjunction with deep learning (Google Deepmind 2015)
- In all these cases, performance was better than could be obtained by any other method, and was obtained without human instruction

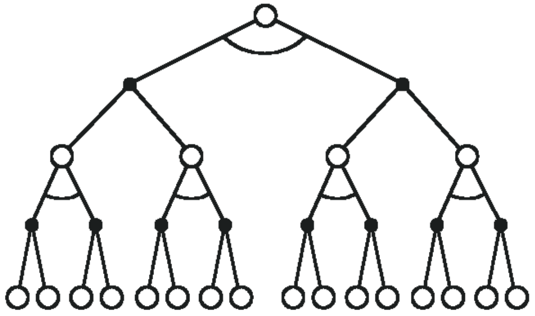
Example: TD-Gammon

Tesauro, 1992-1995




Bbar 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Wbar
s
w
estimated state value
(≈ prob of winning)

Action selection by a shallow search



Start with a random Network

Play millions of games against itself

Learn a value function from this simulated experience

Six weeks later it's the best player of backgammon in the world

Originally used expert handcrafted features, later repeated with raw board positions

Signature challenges of RL

- Evaluative feedback (reward)
- Sequentiality, delayed consequences
- Need for trial and error, to explore as well as exploit
- Non-stationarity
- The fleeting nature of time and online data

Howard, 1960

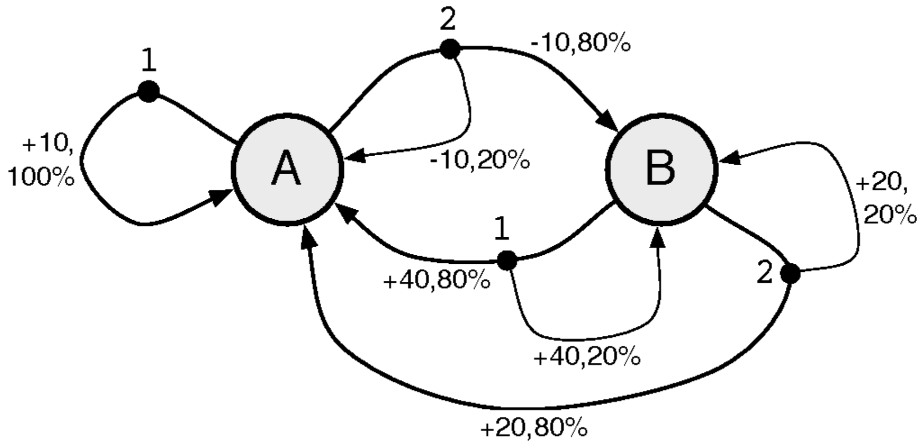
The Environment: A Finite Markov Decision Process (MDP)

- Discrete time $t = 1, 2, 3, \ldots$
- A finite set of states
- A finite set of actions
- A finite set of rewards
- Life is a trajectory:

$$\ldots S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, R_{t+2}, S_{t+2}, \ldots$$

- With arbitrary Markov (stochastic, state-dependent) dynamics:

$$p(r, s'|s, a) = Prob\Big[R_{t+1} = r, S_{t+1} = s' \;\Big|\; S_t = s, A_t = a\Big]$$



Policies

e.g.

State	Action
A $\longrightarrow$	2
B $\longrightarrow$	1

- Deterministic policy

$$a = \pi(s)$$

- An agent following a policy

$$A_t = \pi(S_t)$$

The number of deterministic policies is *exponential* in the *number of states*

- Informally the agent's goal is to choose each action so as to maximize the discounted sum of future rewards,

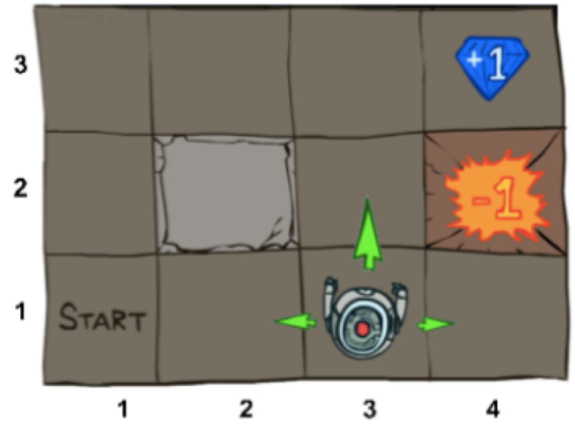
to choose each A_t to maximize $R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$

- We are searching for a policy

Example MDP: Gridworld

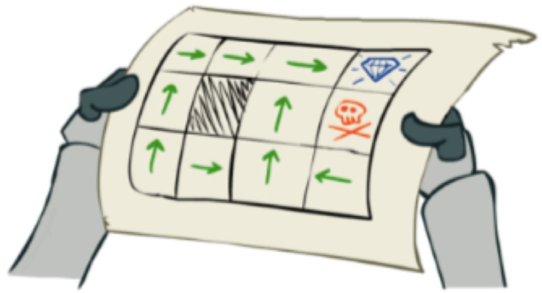
An MDP is defined by:

- Set of states S
- Set of actions A
- Transition function $P(s' \mid s, a)$
- Reward function $R(s, a, s')$
- Start state s_0
- Discount factor γ
- Horizon H



Goal: $$max_\pi \mathrm{E}[\sum_{t=0}^{H} \gamma^t R(S_t, A_t, S_{t+1}) | \pi]$$

π:



Value Function

$$V^*(s) = \max_{\pi} \mathbb{E}\left[\sum_{t=0}^{H} \gamma^t R(s_t, a_t, s_{t+1}) \mid \pi, s_0 = s\right]$$

Value Iteration

```
Algorithm:
    Start with $V_0^*(s) = 0$ for all s.
    For k = 1, ... , H:
        For all states s in S:
            $V_k^*(s) \leftarrow \max_a \sum_{s'} P(s'|s,a)\left(R(s,a,s') + \gamma V_{k-1}^*(s')\right)$
            $\pi_k^*(s) \leftarrow \arg\max_a \sum_{s'} P(s'|s,a)\left(R(s,a,s') + \gamma V_{k-1}^*(s')\right)$

        This is called a value update or Bellman update/back-up
```

Policy Evaluation

- Recall value iteration:

$$V_k^*(s) \leftarrow \max_a \sum_{s'} P(s'|s,a)\left(R(s,a,s') + \gamma V_{k-1}^*(s')\right)$$

- Policy evaluation for a given $\pi(s)$:

$$V_k^\pi(s) \leftarrow \sum_{s'} P(s'|s,\pi(s))(R(s,\pi(s),s') + \gamma V_{k-1}^\pi(s))$$

At convergence:

$$\forall s \quad V^\pi(s) \leftarrow \sum_{s'} P(s'|s,\pi(s))(R(s,\pi(s),s') + \gamma V^\pi(s))$$

Policy Iteration

One iteration of policy iteration:

- Policy evaluation for current policy π_k :
  - Iterate until convergence

$$V_{i+1}^{\pi_k}(s) \leftarrow \sum_{s'} P(s'|s,\pi_k(s))\left[R(s,\pi(s),s') + \gamma V_i^{\pi_k}(s')\right]$$

- Policy improvement: find the best action according to one-step look-ahead

$$\pi_{k+1}(s) \leftarrow \arg\max_a \sum_{s'} P(s'|s,a)\left[R(s,a,s') + \gamma V^{\pi_k}(s')\right]$$

- Repeat until policy converges
- At convergence: optimal policy; and converges faster than value iteration under some conditions

```
1. Initialization
   $v(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
       $\Delta \leftarrow 0$
       For each $s \in \mathcal{S}$:
           $temp \leftarrow v(s)$
           $v(s) \leftarrow \sum_{s'} p(s'|s,\pi(s))\Big[r(s,\pi(s),s') + \gamma v(s')\Big]$
           $\Delta \leftarrow \max(\Delta, |temp - v(s)|)$
   until $\Delta < \theta$  (a small positive number)

3. Policy Improvement
   $policy\text{-}stable \leftarrow true$
   For each $s \in \mathcal{S}$:
       $temp \leftarrow \pi(s)$
       $\pi(s) \leftarrow \arg\max_a \sum_{s'} p(s'|s,a)\Big[r(s,a,s') + \gamma v(s')\Big]$
       If $temp \neq \pi(s)$, then $policy\text{-}stable \leftarrow false$
   If $policy\text{-}stable$, then stop and return $v$ and $\pi$; else go to 2
```

Figure 4.3: Policy iteration (using iterative policy evaluation) for v_*. This algorithm has a subtle bug, in that it may never terminate if the policy continually switches between two or more policies that are equally good. The bug can be fixed by adding additional flags, but it makes the pseudocode so ugly that it is not worth it. :-)

finding optimal value function

```
Initialize array $v$ arbitrarily (e.g., $v(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $temp \leftarrow v(s)$
        $v(s) \leftarrow \max_a \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma v(s')]$
        $\Delta \leftarrow \max(\Delta, |temp - v(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi$, such that
    $\pi(s) = \arg\max_a \sum_{s'} p(s'|s,a)\Big[r(s,a,s') + \gamma v(s')\Big]$
```

Figure 4.5: Value iteration.

one policy update (extract policy from the optimal value function)

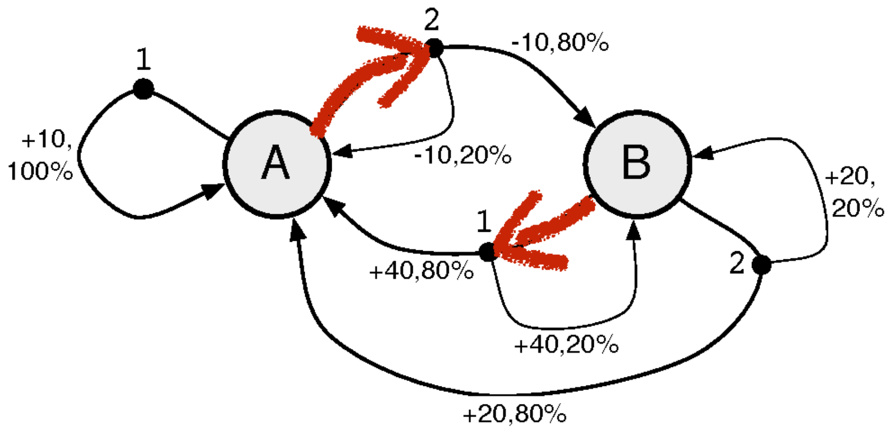
Action-value functions

- An action-value function says how good it is to be in a state, take an action, and thereafter follow a policy:

$$q_\pi(s,a) = \mathbb{E}\Big[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \;\Big|\; S_t = s, A_t = a, A_{t+1:\infty} \sim \pi\Big]$$

Action-value function
for the optimal policy and γ=0.9

State	Action	Value
A	1	130.39
A	2	133.77
B	1	166.23
B	2	146.23



Q-Values

$Q^*(s, a)$ = expected utility starting in s, taking action a, and (thereafter) acting optimally

Bellman Equation:

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)(R(s,a,s') + \gamma \max_{a'} Q^*(s',a'))$$

Q-Value Iteration:

$$Q^*_{k+1}(s,a) \leftarrow \sum_{s'} P(s'|s,a)(R(s,a,s') + \gamma \max_{a'} Q^*_k(s',a'))$$

Optimal policies

- A policy π_* is optimal if it maximizes the action-value function:

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

- Thus all optimal policies share the same optimal value function
- Given the optimal value function, it is easy to act optimally:

$$\pi_*(s) = \arg\max_a q_*(s,a)$$

"greedification"

- We say that the optimal policy is greedy with respect to the optimal value function
- There is always at least one deterministic optimal policy

Q-learning, the simplest RL algorithm

1. Initialize an array $Q(s,a)$ arbitrarily
2. Choose actions in any way, perhaps based on Q, such that all actions are taken in all states (infinitely often in the limit)
3. On each time step, change one element of the array:

$$\Delta Q(S_t, A_t) = \alpha\left(\underbrace{R_{t+1} + \gamma \max_a Q(S_{t+1}, a)}_{\text{target}} - Q(S_t, A_t)\right)$$

4. If desired, reduce the step-size parameter α over time

- Theorem: For appropriate choice of 4, Q converges to q_*, and its greedy policy to an optimal policy π_* (Watkins & Dayan 1992)
- This is kind of amazing — learning long-term optimal behavior without any model of the environment, for arbitrary MDPs!

Policy improvement theorem

- Given the value function for *any policy* π:

$$q_\pi(s,a) \qquad \text{for all } s,a$$

- It can always be greedified to obtain a *better policy*:

$$\pi'(s) = \arg\max_a q_\pi(s,a)$$

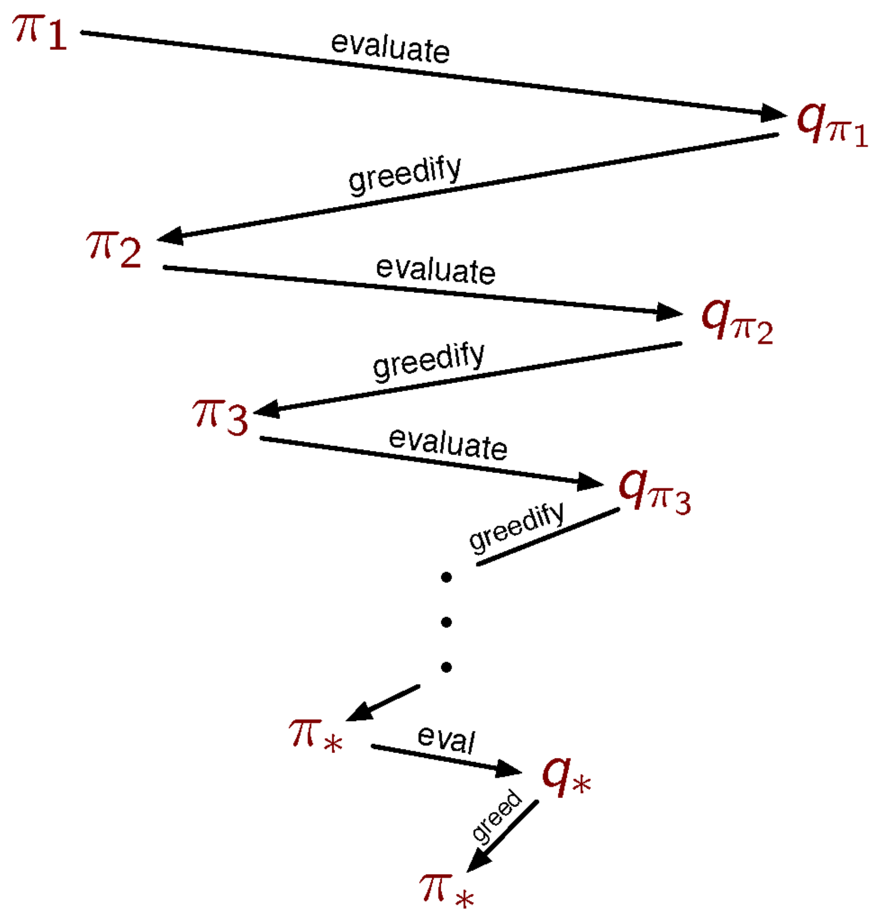
(π' is not unique)

- where better means:

$$q_{\pi'}(s,a) \geq q_\pi(s,a) \qquad \text{for all } s,a$$

- with equality only if <u>both policies are optimal</u>

The dance of policy and value (Policy Iteration)



Any policy evaluates to a unique value function (soon we will see how to learn it)

which can be greedified to produce a better policy

That in turn evaluates to a value function

which can in turn be greedified…

Each policy is *strictly better* than the previous, until *eventually both are optimal*

There are *no local optima*

The dance converges in a finite number of steps, usually very few

The dance is very robust

- to initial conditions
- to delayed and asynchronous updating, as in parallel and distributed implementations
- to incomplete evaluation and greedification
  - updating only some states but not others
  - updating only part of the way
- to randomization and noise
- in particular, it works if only a single state is updated at a time by a random amount that is only correct in expectation

The Explore/Exploit dilemma

- You can't do the action that you think is best all the time
  - because you will miss out big—forever—if you are wrong
  - to find the real best action, you must explore them all…an infinite number of times!
- You also can't explore all the time
  - because then you would never get any advantage of your learning
- Thus you must both explore and exploit, but neither to excess. What is the right balance?

How did Q-learning escape the dilemma?

Q-learning, the simplest RL algorithm

1. Initialize an array $Q(s,a)$ arbitrarily

2. Choose actions in any way, perhaps based on Q, such that all actions are taken in all states (infinitely often in the limit)

3. On each time step, change one element of the array:

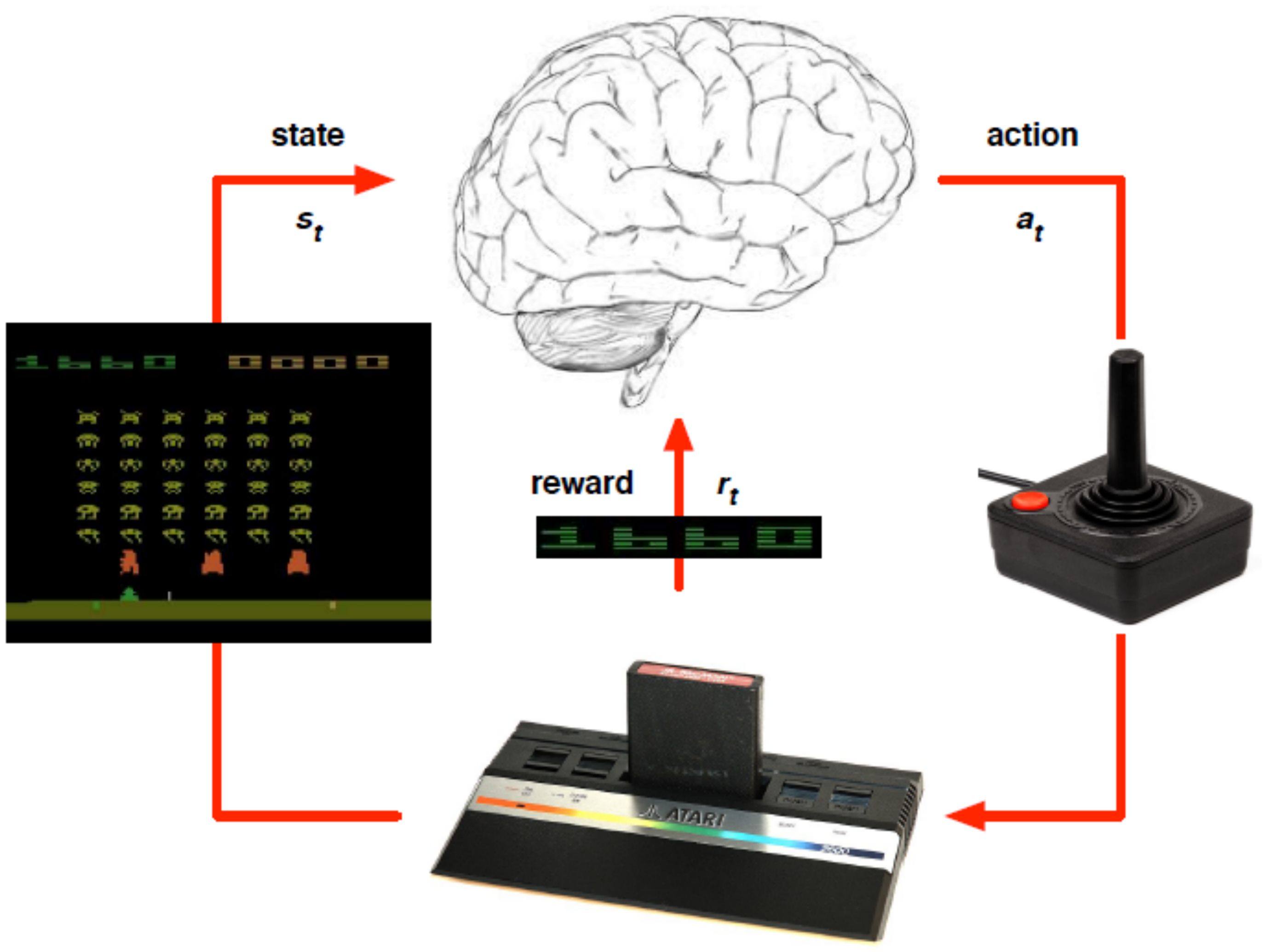
$$\Delta Q(S_t, A_t) = \alpha\Big(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big)$$

4. If desired, reduce the step-size parameter α over time

Deep RL

- Use deep neural networks to represent
  - Value function
  - Policy
  - Model
- Optimize loss function by stochastic gradient descent

Deep RL in Atari

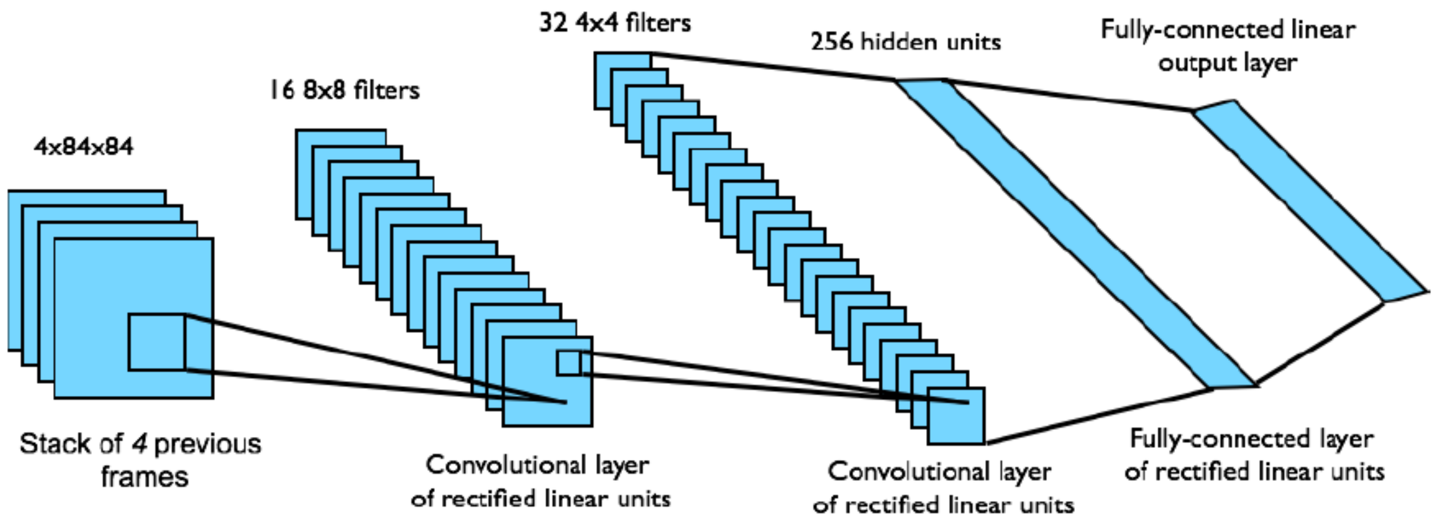


Value-based Deep RL

- An example is Deep Q-Networks (DQN)
  - Q-Learning with experience replay
  - To remove correlations, build dataset from agent’s own experience
  - Sample experiences from dataset and apply update
  - To deal with non-stationarity, target parameters are held fixed
- DQN paper: http://www.nature.com/articles/nature14236

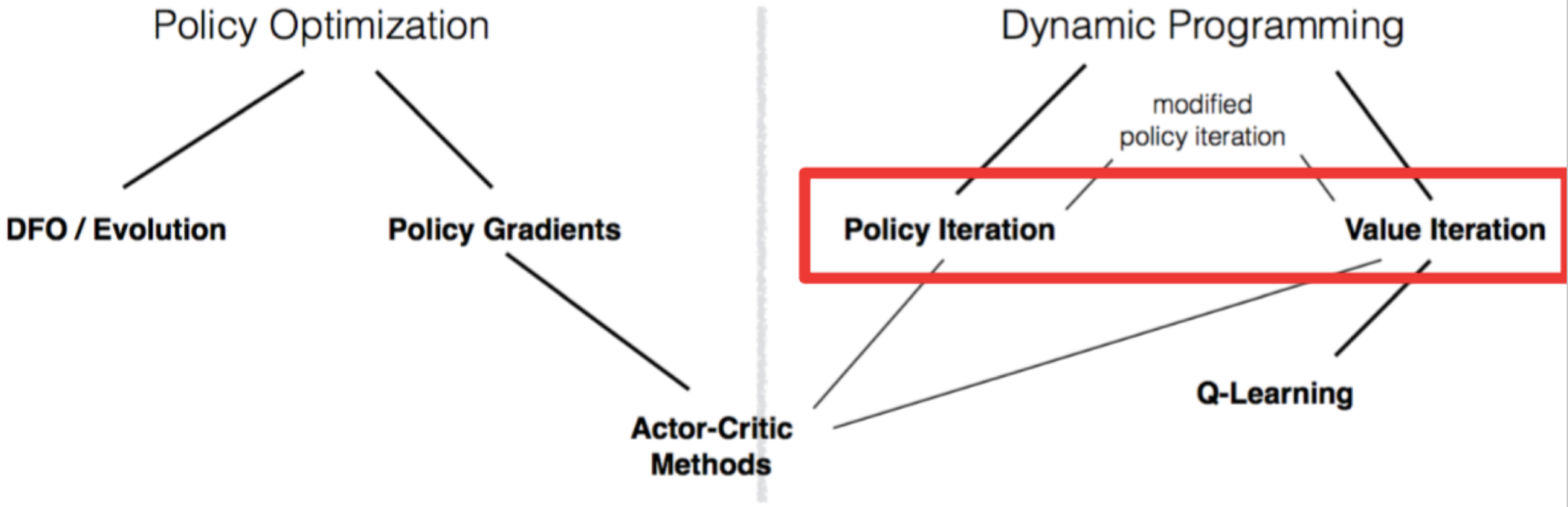
DQN in Atari

- End-to-end learning of values Q(s; a) from pixels s
- Input state s is stack of raw pixels from last 4 frames
- Output is Q(s; a) for 18 joystick/button positions
- Reward is change in score for that step


4x84x84
16 8x8 filters
32 4x4 filters
256 hidden units
Fully-connected linear output layer
Stack of 4 previous frames
Convolutional layer of rectified linear units
Convolutional layer of rectified linear units
Fully-connected layer of rectified linear units

Network architecture and hyperparameters fixed across all games

RL Algorithms Landscape



- Thanks