Search Algorithm: Problems that involve sequential decision making. Last resort: No efficient Algorithm. Easy to verify solution, but find a solution NP-hard problem Solve complex problem is Real-life, trial-and-error On a computer, search algorithm will explore all paths systematically

Define: A set of states, An initial state, Goal states or a goal test (bool func), A successor(neighbour) func (action take from one state to another) A cost(opt)

Search tree: Construct search tree as explore paths incrementally from the start node. Maintain a frontier of path from the start node.

Frontier contains all the paths available for expansion Expanding: Remove from frontier, generating all neighbor of the last node, addback

```
Algorithm 7 A Generic Search Algorithm
1: procedure SEARCH(Graph, Start node $s$, Goal test $goal(n)$)
2:     frontier := $\{\langle s \rangle\}$
3:     while frontier is not empty do
4:         select and remove path $\langle n_0, \dots, n_k \rangle$ from frontier
5:         if goal($n_k$) then
6:             return $\langle n_0, \dots, n_k \rangle$
7:         for every neighbour $n$ of $n_k$ do
8:             add $\langle n_0, \dots, n_k, n \rangle$ to frontier
9:     return no solution
```

DFS: Frontier as stack (LIFO) b-branching factor(# child). M-depth of tree. d-depth shallowest goal

Space: O(size of frontier) O(bm) linear in m Time: O(# nodes visited) O(b^m).

Completeness: Not guaranteed to find solution. Get stuck in infinite path.(may or not cycle)

No attention to cost.

Space restricted, many solution, perhaps long path. – DFS good

Infinite paths, Solution shallow, multiple path to a node -- DFS bad

BFS: frontier as queue (FIFO) Expands first/oldest node of frontier. Space O(b^d). Time(b^d)

Completeness: YES, but not optimal(is the shallowest)

Space not concern, Want solution with few arcs – BFS good

All solution in deep tree, problem is large and graph is dynamically generate – BFS bad

IDS(iterative-deepening search) : For every depth limit, perform dfs until the limit is reach.

Space O(bd), Time O(b^d) Comp: same as BFS

Heuristic search: estimate how close the state is to goal. Try to find optimal solution. Cost(n): the actual cost of the path ending at n

A search heuristic h(n) is an estimate of the cost of the cheapest path from n to goal. Problem-specific, non-negative, h(n)=0 if goal, easy to compute

LCFS: remove path with lowest cost(n), GBFS: remove the path with lowest h(n), A*: remove the path with lowest cost(n)+h(n)

LCFS: exponential, comp-yes, opt-yes if 1.branching factor finite 2.cost of edges is bounded below a positive const

GBFS: exponential, comp-no(cycle), opt-no(sub-optimal may return) A*: exponential, comp-yes, opt-yes same condition of LCFS

A* optimal h(n) admissible if never over-estimate, theorem of optimality: if h(n) admissible, A* optimal

Constructing admissible heur.. 1. Define a relaxed problem 2. Solve the relaxed problem 3. The cost of optimal solution is admissible heur of original

```
Algorithm 6 Search w/ Multi-Path Pruning
1: procedure SEARCH(Graph, Start node $s$, Goal test $goal(n)$)
2:     frontier := $\{\langle s \rangle\}$;
3:     explored := {};
4:     while frontier is not empty do
5:         select and remove path $\langle n_0, \dots, n_k \rangle$ from frontier;
6:         if $n_k \notin$ explored then
7:             add $n_k$ to explored
8:             if goal($n_k$) then
9:                 return $\langle n_0, \dots, n_k \rangle$;
10:            for every neighbour $n$ of $n_k$ do
11:                add $\langle n_0, \dots, n_k, n \rangle$ to frontier;
12:    return no solution
```

Cycle pruning: following cycle, we should stop expanding path. May cause algorithm not terminal. Add condition before add to front {if n not in n_0 to n_k}

Multiple-path pruning. Already found path to a node, discard other paths.

Not optimal for LCFS, optimal for A* if consistent heuristic.

To optimal Remove all paths from the front use longer path, change initial segment to short one.

Consistent heuristic for any mn, h(m)-h(n) <= cost(m,n)

Most admissible are consistent.

Constraint Satisfaction Problems (CSP). Each state contains 1. Set X of variables 2. Set D of domains 3. Set C of constraints specifying allowable value combinations. Solution is assignment all variables satisfy all constraint

Notation of Arc: <X,c(X,Y)>: X is the primary variable, Y is the secondary

Arc Consistency: iff for all v in Dx, there is a value w in Dy such that (v,w) satisfies the constraint c(x,y)

```
Algorithm 6 BACKTRACK(assignment, csp)
1: if assignment is complete then return assignment
2: Let var be an unassigned variable
3: for every value in the domain of var do
4:     if adding {var = value} satisfies every constraint then
5:         add {var = value} to assignment
6:         result ← BACKTRACK(assignment, csp)
7:         if result ≠ failure then return result
8:     remove {var = value} from assignment if it was added
9: return failure
```

```
Algorithm 7 The AC-3 Algorithm
1: put every arc in the set $S$.
2: while $S$ is not empty do
3:     select and remove $\langle X, c(X,Y) \rangle$ from $S$
4:     remove every value in $D_X$ that doesn't have a value in $D_Y$
   that satisfies the constraint $c(X,Y)$
5:     if $D_X$ was reduced then
6:         if $D_X$ is empty then return false
7:         for every $Z \neq Y$, add $\langle Z, c'(Z,X) \rangle$ to $S$
   return true
```

guarantee terminate. Time O(cd^3) n variables, c binary constrains, max domain d

Local search: Does not explore the search space systematically. Can find reasonably good states quickly on avg. Not guaranteed to find solution even if exists. Dose not remember a path to current state. Require very little mem. Can Start with complete assignment, then improve. solve pure optimization problem.

State: a complete assignment to all of the variables. Neighbour relation: which states do I explore next. Cost function: how good is each state?

Greedy descent: Start with random state, move to a neighbour with the lowest cost if it' s better than current, stop when no neighbour has lower cost.

Local optimum not global: Random restarts, Random walks.

```
Algorithm 1 Simulated Annealing
1: current ← initial-state
2: T ← a large positive value
3: while $T > 0$ do
4:     next ← a random neighbour of current
5:     $\Delta C$ ← cost(next) - cost(current)
6:     if $\Delta C < 0$ then
7:         current ← next
8:     else
9:         current ← next with probability $p = e^{\frac{-\Delta C}{T}}$
10:    decrease $T$
11: return current
```

Population-Based Algorithm. Beam search: remember k states, choose the k best states out of all of the neightbours. K controls space and parallelism. Space O(k). when k=1 greedy descent. When branching is large.

Stochastic Beam Search: Choose k states probabilistically. Proportional to exp(-cost(A)/T). Main tians diversity in the population of states. Mimic natural selection.

Uncertainty Estimation: Frequentists view: counting the frequencies of events. Bayesian view: start with prior beliefs and update beliefs based on new evidence.

Random var: Has a domain of possible values. Has associated probability distribution, function of domain [0,1]

Inclusion-exclusion principle: P(A^B) = P(A)+P(B)-P(A&B)

Probabilistic model contains a set of random variables. Atomic event assigns a value to every random var in model.

Joint probability distribution assigns a probability to every atomic event.

P(X): prior or unconditional probability. P(X|Y): posterior or conditional probability. Product Rule: P(A|B)=P(A&B)/P(B) Bayes rule:P(X|Y)=(P(Y|X)*P(X))/P(Y)

Universal approach calculating prob: 1. Convert into fraction of two joint prob 2. Calculate joint prob write as summation 3.Calculate every joint prob

(Unconditional) independence iff P(X|Y)=P(X) P(Y|X)=P(Y) P(X&Y)=P(X)P(Y).

(Conditional) independence if P(X|Y^Z)=P(X|Z) P(Y|X^Z)=P(Y|Z). P(Y^X|Z)=P(Y|Z)P(X|Z)

A Bayesian Network: 1. Is a compact version of joint distribution 2. Take advantage of unconditional and conditional independence among vars

Directed acyclic graph(DAG): 1. Each node corresponds to a rand var. 2. X is a parent of Y if there is an arrow from x to y 3. Each node has cond prob distribution

D-separation: E d-separates X and Y iff E blocks every un-directed path between X and Y. If E d-separates X and Y, then X and Y are cond independent given E.

Rule 1: X···A->N->B···Y if N observed, then block path X to Y. Rule 2: X···A<-N->B···Y if N observed, then block path X to Y

Rule 3: X···A->N<-B···Y if N and its descendants are NOT observed, then block path X to Y

Construct Bayesian Network: 1. Order the variables{X1···Xn} 2. For each variable Xi in the ordering, 2.1 Choose the node' s parents: Choose the smallest set of parents from {X1···Xi-1} such that given Parent(Xi), Xi is independent of all nodes in {X1···Xi-1}-Parents(Xi). 2.2 Create a link from each parent of Xi to Xi. 2.3 Write down the conditional probability table.

Variable Elimination Algorithm: Factors f(···), Restrict a variable: Let ···=··· , Sum out a variable, Multiplying factors, Normalize a factor

1. Construct a factor for each conditional probability. 2.Restrict the observed variables to their observed values. 3.Eliminate each hidden variable Xh_j. 3.1 Multiply all the factors that contain Xh_j to get new factor gj. 3.2 Sum out the variable Xh_j from the factor gj. 4.Multiply the remaining factors. 5.Normalize resulting factor.

The Markov assumption: The future is independent of the past given present. (Sensor)Markov assumption: Each state is sufficient to generate its observation

Hidden Markov Model: 1.A Markov Process 2.The state variables are unobservable 3.The evidence variables, which depend on the states, are observable

Filtering: Given observation from 0 to k, what is the probability that I am in particular state at time k. Enumeration O(K*2^K)ops

Forward Recursion: Let f0:k = P(Sk|o0:k) Basecase: f0:0=alphaP(o0|S0)P(S0) Recursion: f0:k=alphaP(ok|Sk)\sum_s{k-1}{P(Sk|sk-1)f0:(k-1)} O(k) ops

$$
\begin{aligned}
& P(S_k|o_{0:k}) \\
&= P(S_k|o_k \wedge o_{0:(k-1)}) && (1)\\
&= \alpha P(o_k|S_k \wedge o_{0:(k-1)})P(S_k|o_{0:(k-1)}) && (2)\\
&= \alpha P(o_k|S_k)P(S_k|o_{0:(k-1)}) && (3)\\
&= \alpha P(o_k|S_k) \sum_{s_{k-1}} P(S_k \wedge s_{k-1}|o_{0:(k-1)}) && (4)\\
&= \alpha P(o_k|S_k) \sum_{s_{k-1}} P(S_k|s_{k-1} \wedge o_{0:(k-1)})P(s_{k-1}|o_{0:(k-1)}) && (5)\\
&= \alpha P(o_k|S_k) \sum_{s_{k-1}} P(S_k|s_{k-1})P(s_{k-1}|o_{0:(k-1)}) && (6)
\end{aligned}
$$

$$
\begin{aligned}
& P(o_{(k+1):(t-1)}|S_k) \\
&= \sum_{s_{(k+1)}} P(o_{(k+1):(t-1)} \wedge s_{(k+1)}|S_k) && (1)\\
&= \sum_{s_{(k+1)}} P(o_{(k+1):(t-1)}|s_{(k+1)} \wedge S_k) * P(s_{(k+1)}|S_k) && (2)\\
&= \sum_{s_{(k+1)}} P(o_{(k+1):(t-1)}|s_{(k+1)}) * P(s_{(k+1)}|S_k) && (3)\\
&= \sum_{s_{(k+1)}} P(o_{(k+1)} \wedge o_{(k+2):(t-1)}|s_{(k+1)}) * P(s_{(k+1)}|S_k) && (4)\\
&= \sum_{s_{(k+1)}} P(o_{(k+1)}|s_{(k+1)}) * P(o_{(k+2):(t-1)}|s_{(k+1)}) * P(s_{(k+1)}|S_k) && (5)
\end{aligned}
$$

Viterbi Algorithm

Given: π_0 and probabilities P. Return $\hat{s}$ as the output.

- For $k = 1, \cdots, t-1$
  - For $j = 0, \cdots n-1$

$$\pi_k(j) = P(o_k|S_k = j) \max_z [\pi_{k-1}(z)P(S_k = j|S_{k-1} = z)]$$

$$\phi_k(j) = \arg\max_z [\pi_{k-1}(z)P(S_k = j|S_{k-1} = z)]$$

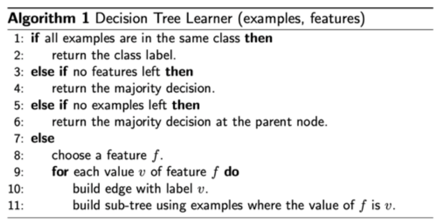
  Find last state $\hat{s}_{t-1} = \arg\max_j \pi_{t-1}(j)$.

- For $k = t-1, \cdots, 1$

$$\hat{s}_{k-1} = \phi_k(\hat{s}_k)$$

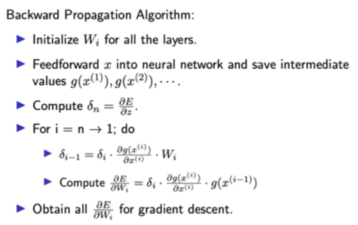
- Return $\hat{s} = \hat{s}_0, \cdots, \hat{s}_{t-1}$.

Smoothing: Given observation from day 0 to day t-1, what is probability that I am in a particular state on day k.
Calculating the smoothed probability P(Sk|o0:t-1)=alphaP(Sk|o0:k)P(o(k+1):(t-1)|Sk)=alpha f0:k b(k+1):(t-1)
Calculate f0:k using forward recursion Calculate b(k+1):(t-1) using backward recursion
Basecase bt:(t-1)=1. Recursion: b(k+1):(t-1)=\sum_(sk+1){P(ok+1|sk+1)b(k+2):(t-1)P(sk+1|Sk)}
Viterbi Algorithm: Brutal-Force Decoding: O(n^t) DP: O(t*n^2) length of sequence as t, n-number states
Decision Network: Decision theory, Probability theory, Utility theory. The principle of maximum expectd utility(MEU). Bayesian network+actions+utilities
Circle random var, square Decision nodes, lingxing Utility node
1.set evidence variables for current state 2.for each possible value of decision node a.set decision node to that value b.calculate posterior prob for parent node of the utility node c.calculate expected utility for the action 3.Return action with highest expected utility
Variable elimination algorithm: 1. Remove all variables that are not ancestors of the utility node. 2. Define a factor for every non-decision node. 3. While there are decision nodes remaining 3.1 Eliminate each random variable that is not a parent of a decision node. 3.2. Find the optimal policy for the last decision and eliminate the decision variable. 4. Return the optimal policies. 5. Determine agent expected utility following the optial policy by eliminating all remaining rand vars.
Infinite horizon: the process may go on forever. Indefinite horizon: the agent will eventually stop, but it does not know when it will stop.
Total reward, average reward, discounted reward: sum t0->\inf gamma^tR(St). Policy specifies what agent should do as a function of current state
Q^*(s,a) = R(s) + \sum_s' P(s' |s,a)V^*(s') pi^*(s)=arg max_a Q^*(s,a)
V^pi(x): Value of being in state s following a policy pi V^*(s): value of being in state s following optimla policy Q^pi(s,a): Value of taking action a while in state s and then follow pi pi(a|s): the policy function, converting state into a distribution over actions.
Bellman equation V^*(s)=R(s) + gamma max_a{\sum_s' {P(s' |s,a)V^*(s')}}
Value iteration Let Vi(s) be the values for the ith iteration. 1.start with arbitrary initial values for V0(s). 2.At the ith iteration, compute Vi+1(s) as vi+1(s) = RHS of bellman equation. 3.Terminate when max_s |Vi(s)-Vi+1(s)| is small enough
Policy Iteration: evaluation, improvement V^{pi_i}(s)=R(s)+gamma \sum_s' {P(s' |s,pi_i(s))V^{pi_i}(s')} pi_i+1(s)=arg max_a \sum_s' {P(s' |s,a)V^{pi_i}(s')}
Reinforcement Learning: Passive ADP: 1.Repeat 2 to 5 2.Follow policy pi and generate an experience <s,a,s' ,r> 3.Update reward func:R(s)=r 4.Update the transition probability N(s,a)+=1,N(s,a,s')+=1,P(s' |s,a)=N(s,a,s')/N(s,a) 5.Derive V^pi(s) by using the bellman equation
Active ADP: 1.Initialize R(s),v+(x),N(s,a),N(s,a,s') 2.Repeate 3 to 7 until we have visited each (s,a)at least Ne times and V+(x) converged. 3.Determine the best action a for current state s uing V+(s) a=arg max_a f(sum_s' P(s' |s,a)V+(s'),N(s,a)), f(u,n)=R+ if n<Ne, =u otherwise 4.Take action a and generate an experience<s,a,s' ,r> 5.Update reward function R(s)=r 6.Update the transition probability similar to the previous one 7.Update V+(s) using bellman updates but add f(u,n)
Temporal difference error = (R(s1)+gamma max_a' Q(s2,a'))-Q(s1,a)
Q-learning: Learning V(s) and Q(s,a) are equivalent. Given an experience <s,a,s' ,r>, update Q(s,a)+=alpha(R(s)+gamma max_a' Q(s' ,a')-Q(s,a))
Or Q(s,a)=(1-alpha)Q(s,a)+alpha(R(s)+gamma max_a' Q(s' ,a'))
Alpha controls the size of each update. If alpha decreases as N(s,a) increases, Q values will converge to the optimal values.
Properties of Q-Learning: 1.Learns Q instead of V 2.Model-free 3.Learns an approximation of the optimal Q-val 4.The smaller alpha, the closer it will converge to optimal, but the slower it will converge.
Machine learning: Supervised, unsupervised, reinforcement. Supervised: Classification: target feature are discrete, Regression: target features are continuous.
Bias-Variance: B:If I have infinite data, how well can I fit data with my learned hypothesis. V:How much does the learned hypothesis vary given different train data.
Cross-validataion: Use part of training data as surrogate for test data(validation data) Use validation data to choose the hypothesis.
K-fold Cross Validation: 1.Break training data into K equally sized part 2.Train a learning algorithm on K-1 partitions(train set) 3,Test on the remaining 1 part(validation set) 4.Do this K times, each time testing on a different part 5.Calculate the avg error on K validation sets After that, select on of K trained hypothesis or train a new hypothesis on all of the data, using parameters selected by cross validation.
Empirical Loss function: example S={Xi,Yi}_1^m, hhat=argmin_h:X->Y 1/M \sum_i=1^M l(h(Xi),Yi)
Unsupervised: Clustering – K-mean: Input X in Rm*n,k in N, d(c,x). 1.Initialization: Randomly initialize k centroids. 2.While not converge, do 2.1 Assign each example to the cluster whose centroid is closest. 2.2.Calculate centroid for each cluster c by calculating the average value for each example currently classified as cluster c.
Principal Component Analysis: dimensionality reduction: 1.Mean center data 2.Compute Covariance Matrix Sig 3.Calculate the eigen values and eigen vectors of Sig 3.1.Eigenvector with largest eigen value n1 is the first PC 3.2.Eigenvector with kth largest eigenvalue nk is the kth PC 3.3. nk/Sigknk is the proportion of variance captured by kth PC. Autoencoders: xhat=d(e(x)), E=sum_i{(xi-d(e(xi)))^2}. Linear Autoencoder zhat=Wx, xhat=W^Tzhat
Generative Adversarial Network (GANs):trained with minimax error: E=Ex[log(d(x))]_Ez[log(1-d(g(z)))]. Disc try max E, Gene try min E. g->real image, d->1/2
Decision Trees: Stop in any of following: 1.All the examples belong to the same class. 2.There are no more features to test. 3.There are no more examples
No feature left: Data nosy. Predict the majority class or predict the labels probabilistically.
No example left: A combination is not in training set. Use the majority decision in the examples at the parent node.



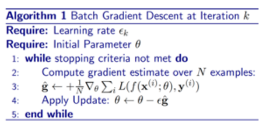
Algorithm 1 Decision Tree Learner (examples, features)

```
1: if all examples are in the same class then
2:     return the class label.
3: else if no features left then
4:     return the majority decision.
5: else if no examples left then
6:     return the majority decision at the parent node.
7: else
8:     choose a feature $f$.
9:     for each value $v$ of feature $f$ do
10:        build edge with label $v$.
11:        build sub-tree using examples where the value of $f$ is $v$.
```



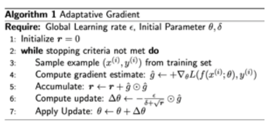
Backward Propagation Algorithm:

- Initialize W_i for all the layers.
- Feedforward x into neural network and save intermediate values $g(x^{(1)}), g(x^{(2)}), \cdots$.
- Compute $\delta_n = \frac{\partial E}{\partial z}$.
- For i = n → 1; do
  - $\delta_{i-1} = \delta_i \cdot \frac{\partial g(x^{(i)})}{\partial x^{(i)}} \cdot W_i$
  - Compute $\frac{\partial E}{\partial W_i} = \delta_i \cdot \frac{\partial g(x^{(i)})}{\partial x^{(i)}} \cdot g(x^{(i-1)})$
- Obtain all $\frac{\partial E}{\partial W_i}$ for gradient descent.



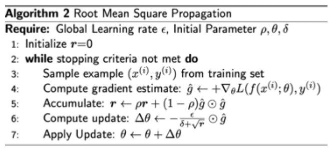
Algorithm 1 Batch Gradient Descent at Iteration k

```
Require: Learning rate $\epsilon_k$
Require: Initial Parameter $\theta$
1: while stopping criteria not met do
2:     Compute gradient estimate over $N$ examples:
3:     $\hat{\mathbf{g}} \leftarrow +\frac{1}{N}\nabla_\theta \sum_i L(f(\mathbf{x}^{(i)};\theta), \mathbf{y}^{(i)})$
4:     Apply Update: $\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$
5: end while
```



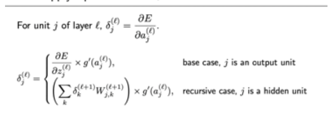
Algorithm 1 Adaptative Gradient

```
Require: Global Learning rate $\epsilon$, Initial Parameter $\theta, \delta$
1: Initialize $r = 0$
2: while stopping criteria not met do
3:     Sample example $(x^{(i)}, y^{(i)})$ from training set
4:     Compute gradient estimate: $\hat{g} \leftarrow +\nabla_\theta L(f(x^{(i)};\theta), y^{(i)})$
5:     Accumulate: $r \leftarrow r + \hat{g} \odot \hat{g}$
6:     Compute update: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta+\sqrt{r}} \odot \hat{g}$
7:     Apply Update: $\theta \leftarrow \theta + \Delta\theta$
```



Algorithm 2 Root Mean Square Propagation

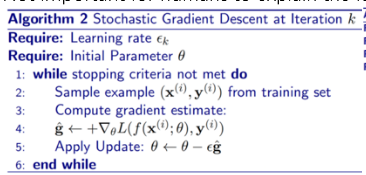
```
Require: Global Learning rate $\epsilon$, Initial Parameter $\rho, \theta, \delta$
1: Initialize $r$=0
2: while stopping criteria not met do
3:     Sample example $(x^{(i)}, y^{(i)})$ from training set
4:     Compute gradient estimate: $\hat{g} \leftarrow +\nabla_\theta L(f(x^{(i)};\theta), y^{(i)})$
5:     Accumulate: $r \leftarrow \rho r + (1-\rho)\hat{g} \odot \hat{g}$
6:     Compute update: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta+\sqrt{r}} \odot \hat{g}$
7:     Apply Update: $\theta \leftarrow \theta + \Delta\theta$
```



For unit j of layer ℓ, $\delta_j^{(\ell)} = \frac{\partial E}{\partial a_j^{(\ell)}}$.

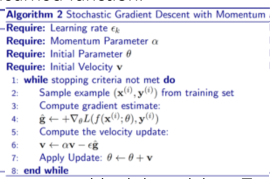
$$\delta_j^{(\ell)} = \begin{cases} \frac{\partial E}{\partial z_j^{(\ell)}} \times g'(a_j^{(\ell)}), & \text{base case, } j \text{ is an output unit} \\ \left(\sum_k \delta_k^{(\ell+1)} W_{j,k}^{(\ell+1)}\right) \times g'(a_j^{(\ell)}), & \text{recursive case, } j \text{ is a hidden unit} \end{cases}$$

Entropy I(P(c1),⋯,P(ck))=-\sum_i=1^k{P(ci)log_2(P(ci))} ID3: Calculate entropy of every attribute of S, Partition S into subset using minimized entropy, Make decision tree node contain that attribute, Recurse on subsets using remaining attributes.
Neural Netwroks: activation function: should nonlinear, should mimic the behaviour of real neurons, should be differentiable almost everywhere
Sigmoid function g(x) = 1/1+(e^-kx) For very large or very small x, g(x) -> 1 or 0. Vanishing gradient problem: when x is large or small, small change, not learn
Rectified linear unit(ReLU) g(x)=max(0,x_. The dying ReLU problem: when inputs approach 0 or -,gradient ->0, not learn. Leak ReLU g(x)=max(0,x)+k*min(0,x)
Feedforward network: no loops, only in one direction. Recurrent network: feed its output back into input, short memory. h_t=fW(ht-1,xt), yt=fY(ht)
Gradient Descent: Initialize weights randomly. Change each weight in proportion to the negative of the partial derivative of the error wrt weight W:=W-eta dE/dW. Eta is the learning rate. Terminate after some # of steps, when error is small or when the changes get small.
Incremental gradient descent: update weights after each example. Stochastic gradient descent :same as incremental version but each example is chosen randomly.
Batched gradient descent: update weights after a batch of examples. Forward pass: compute the error E given the inputs and the weights. Backward pass: compute the gradients dE/dW^2j,k and dE/dW^1i,j. When to use NN: high dimensional or real-valued inputs, noisy(sensor) data. Form of target function is unkown(no model). Not important for humans to explain the learned function.



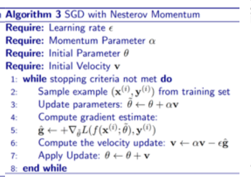
Algorithm 2 Stochastic Gradient Descent at Iteration k

```
Require: Learning rate $\epsilon_k$
Require: Initial Parameter $\theta$
1: while stopping criteria not met do
2:     Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set
3:     Compute gradient estimate:
4:     $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)};\theta), \mathbf{y}^{(i)})$
5:     Apply Update: $\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$
6: end while
```



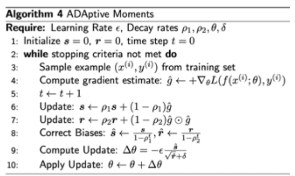
Algorithm 2 Stochastic Gradient Descent with Momentum

```
Require: Learning rate $\epsilon_k$
Require: Momentum Parameter $\alpha$
Require: Initial Parameter $\theta$
Require: Initial Velocity $\mathbf{v}$
1: while stopping criteria not met do
2:     Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set
3:     Compute gradient estimate:
4:     $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)};\theta), \mathbf{y}^{(i)})$
5:     Compute the velocity update:
6:     $\mathbf{v} \leftarrow \alpha\mathbf{v} - \epsilon\hat{\mathbf{g}}$
7:     Apply Update: $\theta \leftarrow \theta + \mathbf{v}$
8: end while
```



Algorithm 3 SGD with Nesterov Momentum

```
Require: Learning rate $\epsilon$
Require: Momentum Parameter $\alpha$
Require: Initial Parameter $\theta$
Require: Initial Velocity $\mathbf{v}$
1: while stopping criteria not met do
2:     Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set
3:     Update parameters: $\tilde{\theta} \leftarrow \theta + \alpha\mathbf{v}$
4:     Compute gradient estimate:
5:     $\hat{\mathbf{g}} \leftarrow +\nabla_{\tilde{\theta}} L(f(\mathbf{x}^{(i)};\tilde{\theta}), \mathbf{y}^{(i)})$
6:     Compute the velocity update: $\mathbf{v} \leftarrow \alpha\mathbf{v} - \epsilon\hat{\mathbf{g}}$
7:     Apply Update: $\theta \leftarrow \theta + \mathbf{v}$
8: end while
```



Algorithm 4 ADAptive Moments

```
Require: Learning Rate $\epsilon$, Decay rates $\rho_1, \rho_2, \theta, \delta$
1: Initialize $s = 0$, $r = 0$, time step $t = 0$
2: while stopping criteria not met do
3:     Sample example $(x^{(i)}, y^{(i)})$ from training set
4:     Compute gradient estimate: $\hat{g} \leftarrow +\nabla_\theta L(f(x^{(i)};\theta), y^{(i)})$
5:     $t \leftarrow t + 1$
6:     Update: $s \leftarrow \rho_1 s + (1-\rho_1)\hat{g}$
7:     Update: $r \leftarrow \rho_2 r + (1-\rho_2)\hat{g} \odot \hat{g}$
8:     Correct Biases: $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$, $\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$
9:     Compute Update: $\Delta\theta = -\epsilon\frac{\hat{s}}{\sqrt{\hat{r}}+\delta}$
10:    Apply Update: $\theta \leftarrow \theta + \Delta\theta$
```

BGD: pos: Gradient Estimates are stable
Neg: Need to compute the gradients over the entire training for one update.
SGD: learning rate decayed linearly epk=(1-alpha)ep0 + alpha eptau with alpha = k/tau. Tau is usually set to the number of iterations needed for a large # of passes through the data. Eptau should roughly be set to a small number. Gradient estimate can be very noisy. Use large mini-batches. Adv: computation time per update doesnot depend on # of training example N. Allows converge on extreme large dataset.
Nesterov Momentum: first take a step in the direction of the accumulated gradient, then calculate the gradient and make a correction.
AdaGrad maintains a per-parameter learning rate that improves the performance on problems with sparse gradients, RMSProp also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight. Momentum Method can maintain a velocity term to keep track of the history gradients.