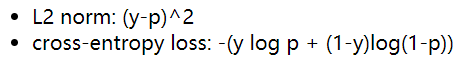
## **Linear Classifiers**

* Sample activation functions. Know the equations for sigmoid and ReLU.
  + Sigmoid: logistic(x)
    - 1/(1+e−x)

The output of logistic function is in the right range to be interpreted as a probability

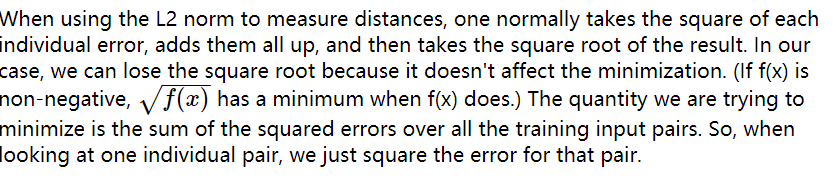
A slow transition across the decision boundary lets us treat values near the boundary as uncertain.

* + ReLU (rectified linear unit)
    - f(x) = x when positive, f(x) = 0 if negative
* Sample loss functions (e.g. 0/1, L1, L2, cross-entropy



Suppose that y is the correct output and p is the output of our activation function

In all cases, we average these losses over all our training pairs.



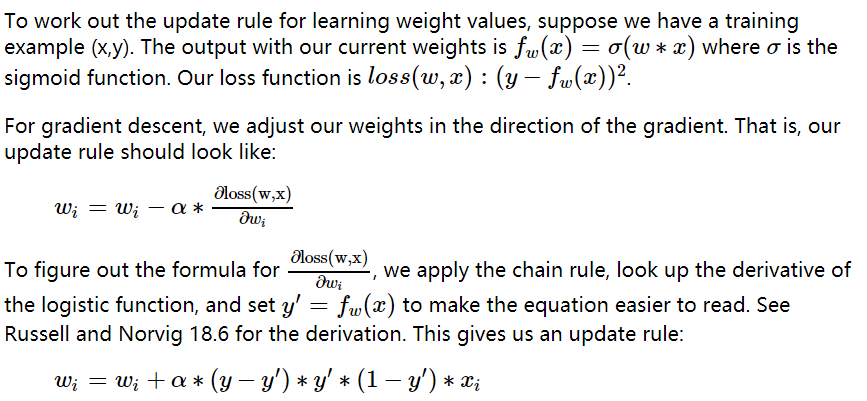
entropy is the number of bits required to (optimally) compress a pool of values.

The high-level idea of cross-entropy is that we have built a model of probabilities (e.g. using some training data) and then use that model to compress a different set of data. Cross-entropy measures how well our model succeeds on this new data. This is one way to measure the difference between the model and the new data.

* What are we minimizing when we adjust the weights? (composition of weighted feature sum, activation function, loss function)

The quantity we are trying to minimize is the sum of the squared errors over all the training input pairs.

* Adjusting weights for a differentiable unit using gradient descent. Know the main equation for updating a weight given the output loss.

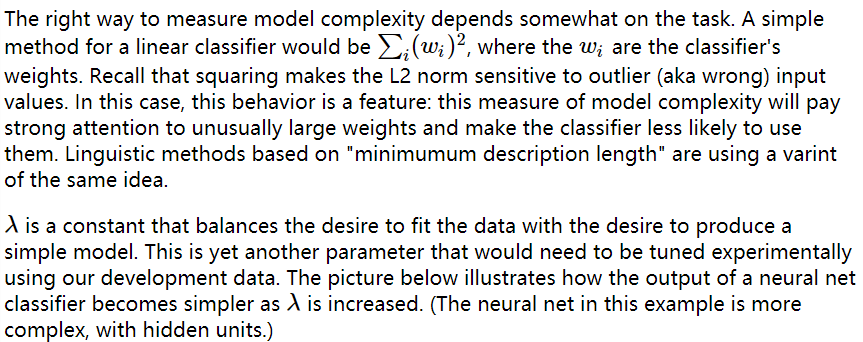


* Regularization.

Any type of model fitting is subject to possible overfitting.

Suppose that our set of training data is T. Then, rather than minimizing loss(model,T), we minimize:





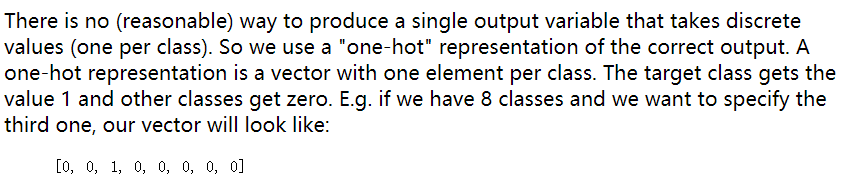
Small lambda means focus on the data: prone to overfitting

Large lambda means not: not fitting

to measure the complexity of the model, pay attention to the wrong input, so make the classifier less likely to use them

* One-hot representation

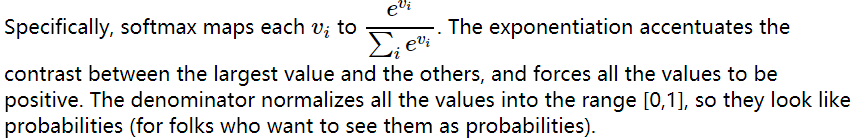
For multi-class perceptron:



* Softmax

For multi-class perceptron:

Softmax is a differentiable function that approximates this discrete behavior. It's best thought of as a version of argmax.



输入为向量，输出为值为0-1之间的向量，和为1。在分类任务中作为概率出现在交叉熵损失函数中。

## **No Calculus details**

You will not need to remember/calculate derivatives for specific functions, or composed sets of functions. I want you to know the high-level picture of what derivative computations you're asking a tool (e.g. pytorch) to do for you.

## **Neural Nets**

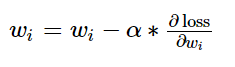
* Design (i.e. connected set of linear classifiers)
  + When we have multiple layers, why should there be a (non-linear) activation function between them?

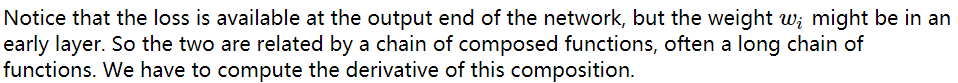
its aim in a neural network is to produce a nonlinear decision boundary via non-linear combinations of the weight and inputs.

* What kinds of functions can a neural net approximate?

To approximate a complex shape with only 2-3 layers, each hidden unit takes on some limited task.

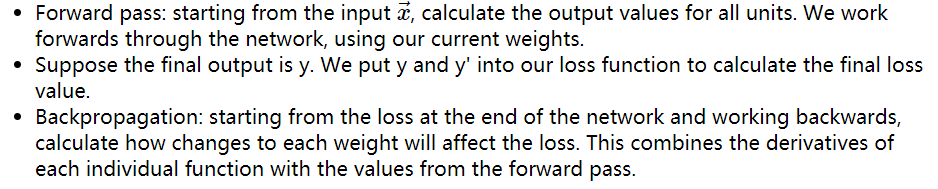
* Training
  + Top-level (aka simple) update equation for a single weight in the network





* + What does backpropagation compute? High-level picture of how it works.

E.g. where do we use the chain rule? Why do we need the forward values?



* + Three challenges:

Symmetry breaking, Overfitting, Vanishing/exploding gradients

* + Symmetry breaking

Perceptron training works fine with all weights initialized to zero. This won't work in a neural net, because each layer typically has many neurons connected in parallel.

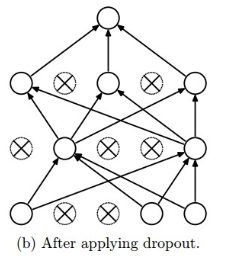
To break symmetry:

1. Set the initial weights to random values.
2. Randomize some aspect of the later training, e.g. ignore some randomly-chosen units on each update.

Dropout

One specific proposal for randomization is dropout: Within the network, each unit pays attention to training data only with probability p. On other training inputs, it stops listening and starts reading its email or something. The units that aren't asleep have to classify that input on their own.

**This can help prevent overfitting/** **symmetry breaking**



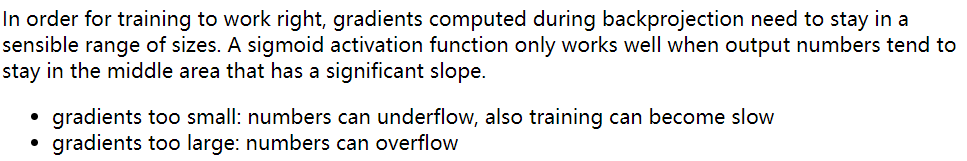
* + Overfittings

**The dropout technique will reduce this problem. Another method is Data augmentation.**

Data augmentation

Data augmentation tackles the fact that training data is always very sparse, but we have additional domain knowledge that can help fill in the gaps. We can make more training examples by perturbing existing ones in ways that shouldn't (ideally) change the network's output. For example, if you have one picture of a cat, make more by translating or rotating the cat.

* + Vanishing/exploding gradients



The underflow/overflow issues happen because numbers that are somewhat too small/large tend to become smaller/larger.

Several approaches:

1. ReLU is less prone to these problems, but they stop training if inputs force their outputs negative.

So people often use a "leaky ReLU" function which has a very small slope on the negative side, e.g. f(x) = x for positive inputs, f(x) = 0.01x for negative ones.

1. Initialize weights so that different layers end up with the same variance in gradients
2. Gradient clipping: detect excessively high gradients and reduce them.
3. Weight regularization: many of the problematic situations involve excessively large weights. So add a regularization term to the network's loss function that measures the size of the weights (e.g. sum of the squares or magnitudes of the weights).
   * Leaky ReLu

So people often use a "leaky ReLU" function which has a very small slope on the negative side, e.g. f(x) = x for positive inputs, f(x) = 0.01x for negative ones.

* + Weight regularization

Weight regularization: many of the problematic situations involve excessively large weights. So add a regularization term to the network's loss function that measures the size of the weights (e.g. sum of the squares or magnitudes of the weights).

* + Why must we initialize weights to random values rather than zero? (You don't need to give names or details for specific initialization methods.)

To Solve:

1. Symmetry breaking
2. Vanishing/exploding gradients?? A bit different

* Convolutional neural networks
  + What is convolution?

each unit computes a weighted sum of the values in that local region. In signal processing, this is known as "convolution" and the set of weights is known as a "mask."

* + How does a convolutional layer work?

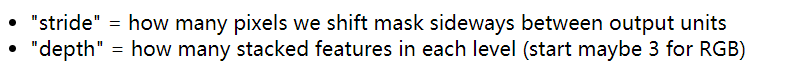
a number of different convolution masks would be useful to apply, each picking out a different type of feature. So, in reality, each network layer has a significant thickness, i.e. a number of different values at each (x,y) location.

* + In what situations would we want a convolutional layer vs. a fully-connected layer?

NN: has only one value at each (x,y) position.

CNN: have depth

* + Depth and stride



* + What is a pooling layer?

reduces the size of the data, by producting an output value only for every kth input value in each dimension.

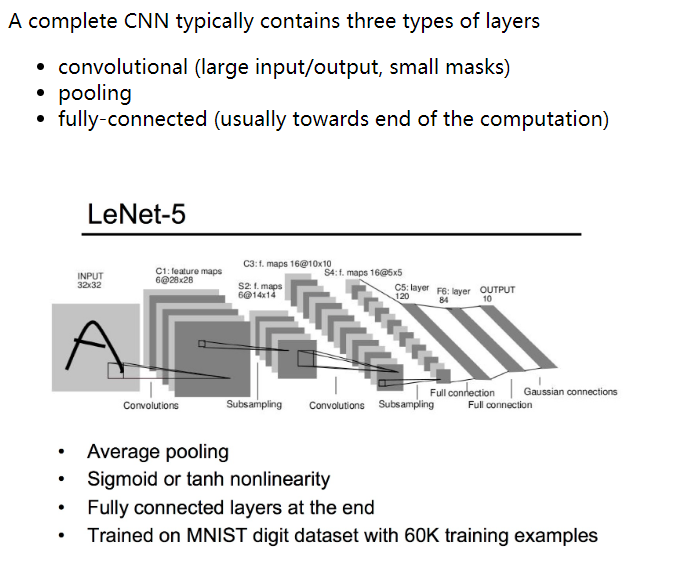
The output values may be either selected input values, or the average over a group of inputs, or the maximum over a group of inputs.

This kind of reduction in size ("downsampling") is especially sensible when data values are changing only slowly across the image. For example, color often changes very slowly except at object boundaries, and the human visual system represents color at a much lower resolution than brightness.

* + Weight/parameter sharing

units in the same layer share a common set of weights and bias. **This cuts down on the number of parameters to train. May worsen performance if different regions in the input images are expected to have different properties,** e.g. the object of interest is always centered.

* + Overall architecture, e.g. what kinds of features are detected in early vs. late layers?



* Generative adversarial neural network

consists of two neural nets that jointly learn a model of input data. The classifier tries to distinguish real training images from similar fake images. The adversary tries to produce convincing fake images.

* Adversarial examples

An adversarial example is an example constructed to produce a completely wrong

answer from a neural net. Typically, small perturbations are added to the values (e.g. color values in a picture) in a way that makes the neural net misbehave while leaving the overall appearance of the example intact.

it is possible to cook up patterns that are fairly close to random noise but push the network's values towards or away from a particular output classification.

* Recurrent neural networks

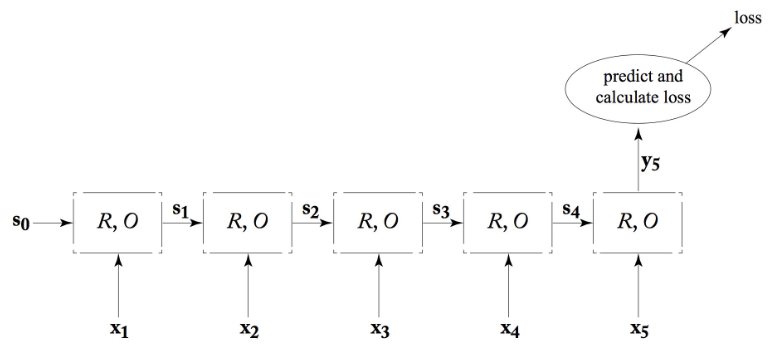
are neural nets that have connections that loop back from a layer to the same layer.

The intent of the feedback loop in the picture is that each unit is connected to all the other units in the layer.

* + High-level view of how they work

When unroll, all copies of the unit share the same parameter values.

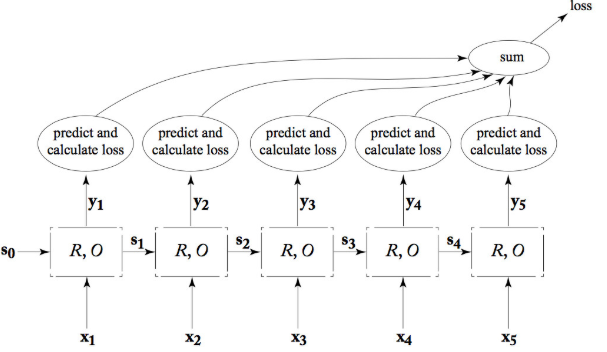
An RNN can be used as a classifier. That is, the system using the RNN only cares about the output from the last timestep, as shown below. In an NLP system, the final output value may actually be complex, e.g. a summary of an entire sentence. Either way, the final output value is fed into a later processing that provides feedback about its performance (the loss value).



* + When would we compute loss from last unit vs. summed over all units?

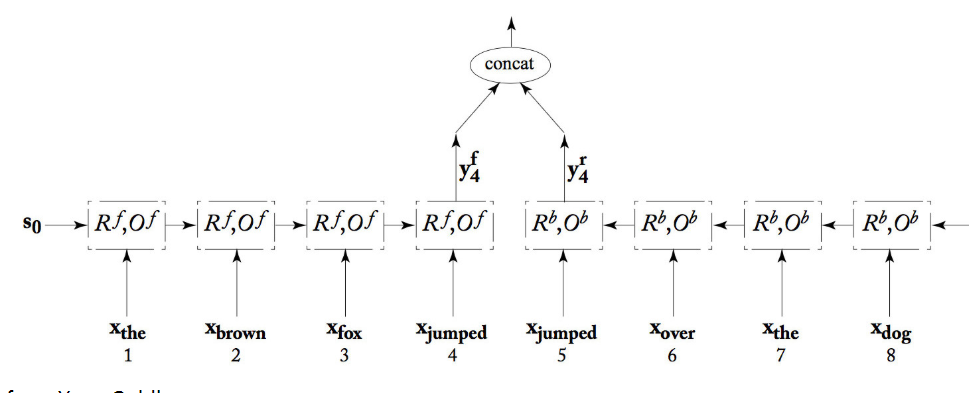
This kind of RNN can be trained in much the same way as a standard ("feedforward") neural net. However, values and error signals propagate **in the time direction.** The forward pass calculates values moving to the right. Backpropagation starts at the final loss node and moves back to the left. This is often called **"backpropagation through time."**

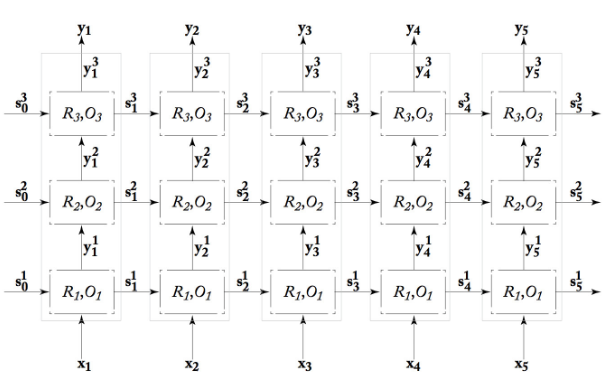
When applying in NLP for modelling sequential data: mapping words to part-of-speech tags. In this case, we would care about getting the correct output at each timestep (not just the final one). So the connection to our loss function would look like this:



* + Bidirectional RNN

We can join two RNN's together into a "bidirectional RNN." One RNN works forwards **from the start of the input**, the second works backwards **from the end of the input.** The output at each position is the concatenation of the two RNN outputs.



deep RNN

Gated RNN:

**In theory, an RNN can remember the entire stream of input values.** However, gradient magnitude tends to decay as you move backwards from the loss signal. **So earlier inputs may not contributed much to the RNN's final answer at the end.**

**To allow RNNs to store information more effectively, researchers use "gated" versions of RNNs.**

Two popular gated RNN models are the "Long Short-Term Memory" (LSTM) and the "Gated Recurrent Unit" (GRU).

## **Markov Decision Processes and Reinforcement Learning**

* Model and terminology for an MDP

MDP:

The basic set-up is an agent (e.g. imagine a robot) moving around a world like the one shown below. Positions add or deduct points from your score. The Agent's goal is to accumulate the most points.

Mathematical Model:

set of states s∈S

set of actions a∈A

reward function R(s)

transition function P(s' | s,a)

The transition function tells us the probability that a commanded action a in a state s will cause a transition to state s'.

Our solution will be a policy π(s) which specifies which action to command when we are in each state.

The background reward for the unmarked states changes the personality of the MDP. **If the background reward is high (lower right below), the agent has no strong incentive to look for the high-reward states. If the background is strongly negative (upper left), the agent will head aggressively for the high-reward states, even at the risk of landing in the -1 location.**

we make the assumption that rewards are better if they occur sooner. The equations in the next section will define a "utility" for each state that takes this into account. The utility of a state is based on its own reward and, also, on rewards that can be obtained from nearby states. **That is, being near a big reward is almost as good as being at the big reward.**

* Quantizing/digitizing continuous state variables
* Bellman equation **Optimal Policy**

1. Here, we start by assuming that an agent will always pick the best Move or action. And we know we can express utility of each state in terms of utilities of adjacent states

U(s) = R(s) + maxa∈A \* U(move(a,s))

1. our action does not control what s’(next state) will be. So to adjust U(S) we compute a sum that is weighted by Probability of s’ (the next state).

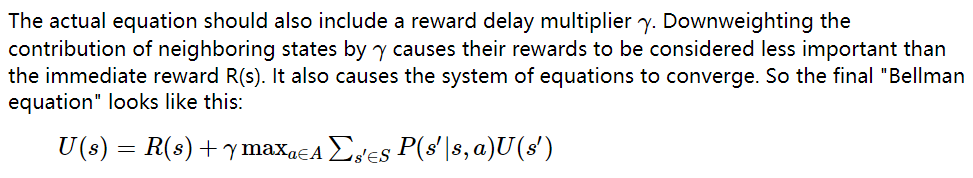
This is done by modeling the next state utility like this

Next\_state\_utility = ∑s′∈SP(s′|s,a)U(s’)

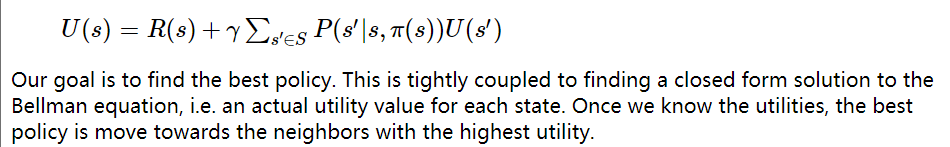
Substitute current expression of utilities for the s’ (adjacent states) with our new expression that is weighted by probabilities of s’

U(s)= R(s) + maxa∈A ∑s′∈S \* P(s′|s,a) \* U(s′)

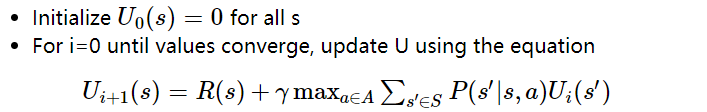
1. Multiply next\_state\_utility with reward delay factor (lambda). Downweighting the contribution of neighboring states by γ causes their rewards to be considered less important than the immediate reward R(s). It also causes the system of equations to converge.



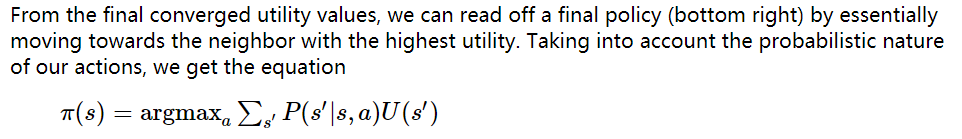
* Bellman equation Fixed Policy



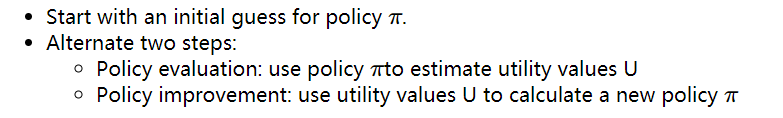
* Methods of solving the Bellman equation
  + Value iteration: repeatedly applies the Bellman equation to update utility values for each state.

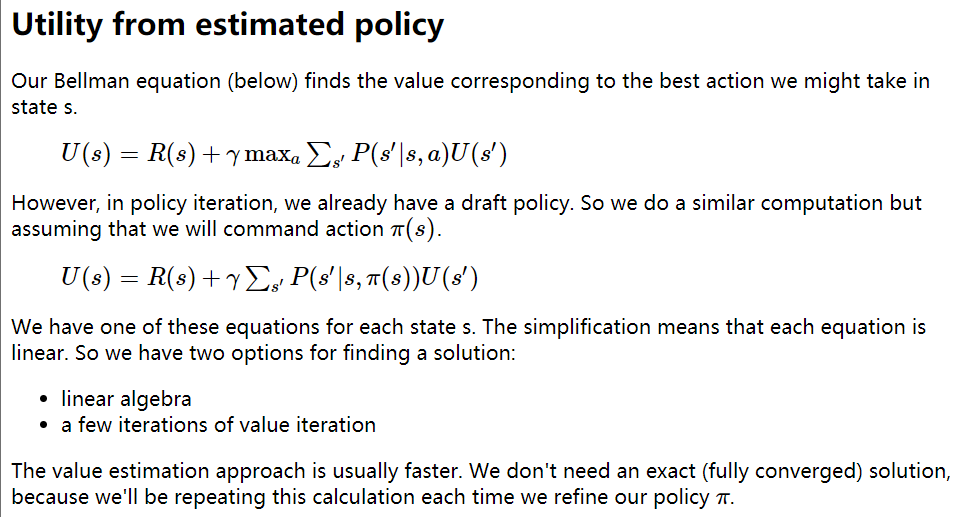


Until converged



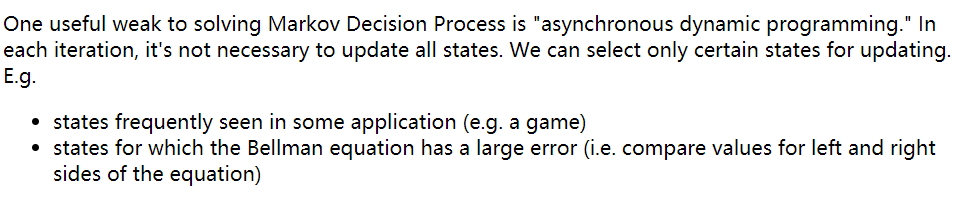
* + Policy iteration: Policy iteration produces the same solution, but faster.



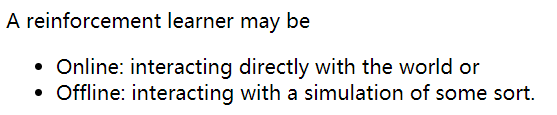


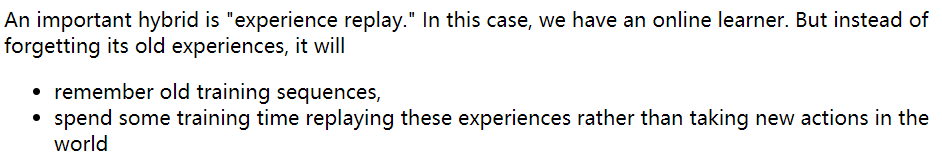
* + Asynchronous dynamic programming

Unnecessary to update all states in each iteration, just update the most common states and the states with large errors.

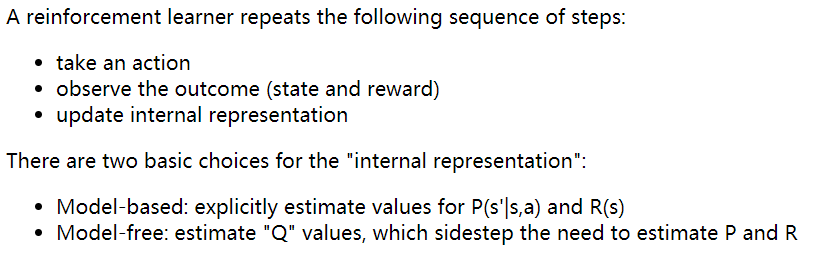


* RL

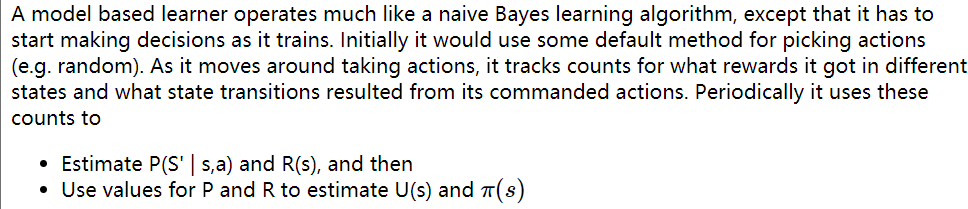




* RL learning loop



* Model-based reinforcement learning



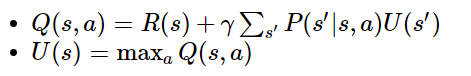
The obvious implementation of a model-based learner tends to be **risk-averse.** We should add exploration to avoid it miss very good possibilities (e.g. large rewards). We can modify our method of selecting actions:

* With probability p, pick π(s).
* With probability 1-p, explore.

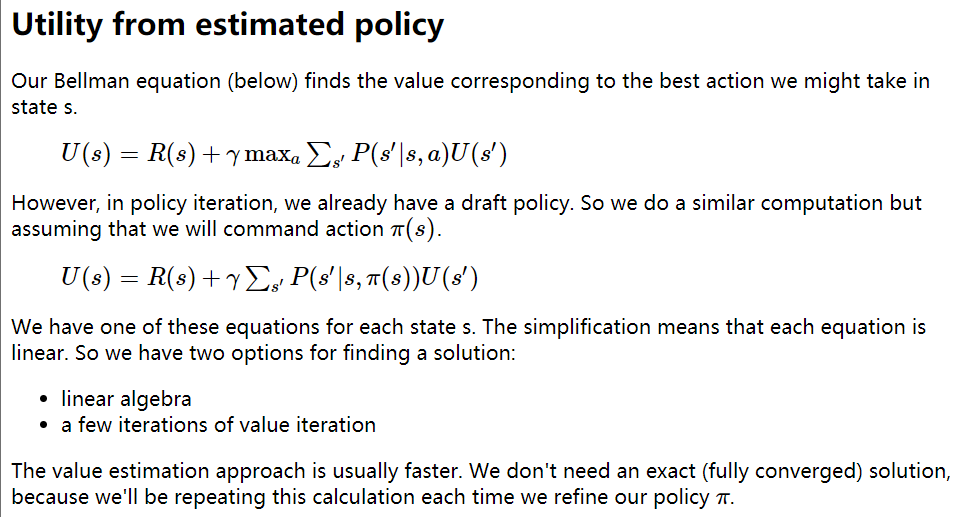
“Explore” could be implemented in various ways,

* Make a uniform random choice among the actions
* Try actions that we haven’t tried “enough” times in the past
* Model-free reinforcement learning
  + Q-learning version of Bellman equation (expressing Q in terms of itself, without reference to the utility or transition probability functions)

Q(s,a) tells us the value of commanding action a when the agent is in state s.



Goal: Express Q by removing dependence on U(s), U(s’) and P(s’|s,a) ,In bellman equation.



Remove U(s’)

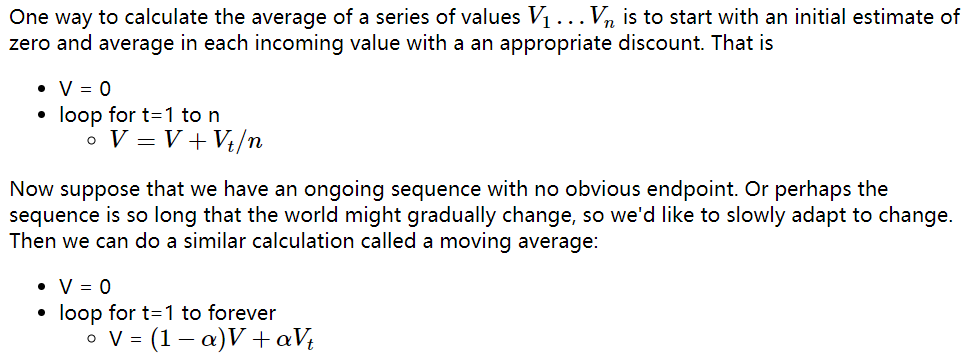


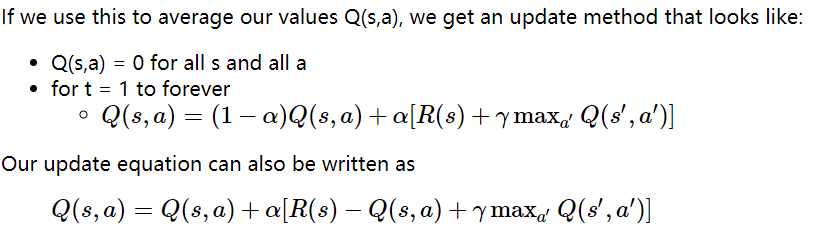
Remove P



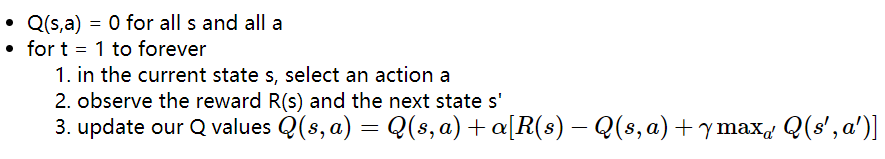
* + TD update algorithm

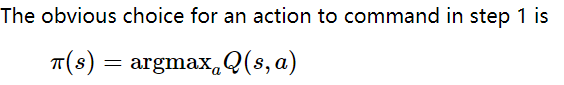
Q-Value Update equations:

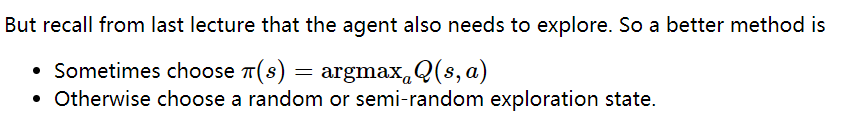


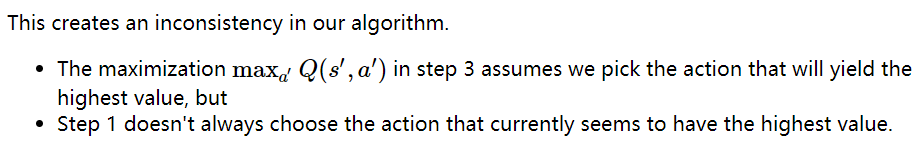


TD update Algorithm:





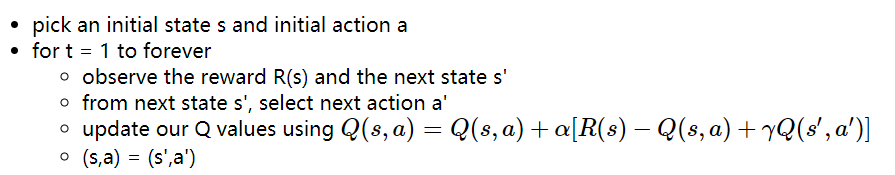




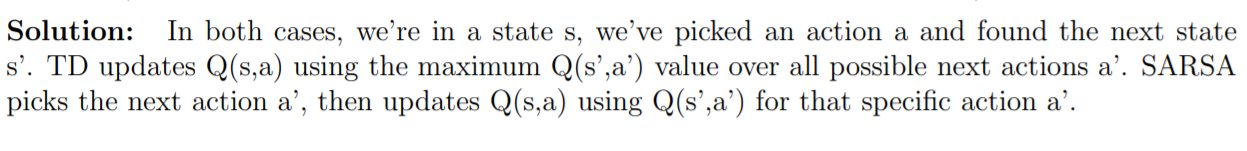
So the update in step 3 is based on a different action from the one actually chosen by the agent.

SARSA update algorithm

Adjusts TD update algorithm to align the update with the actual choice of action



* + How do TD and SARSA differ?



The SARSA agent stays further from the hazard, so that the occasional random motion isn't likely to be damaging.

The TD algorithm assumes it will do a better job of following policy, so it sends the agent along the edge of the hazard and it regularly falls in, more exploration.

* Selecting an action
  + Deriving a policy from utility values or from Q values.
  + Incorporating random exploration

To improve performance, we modify our method of selecting actions:

with probability p, pick π(s)

with probability 1-p, explore

You've done an MP using Q-learning with TD update, so you should have a detailed understanding of how it works.

## **Game Search**

* Game tree

In [game theory](https://en.wikipedia.org/wiki/Game_theory), a **game tree** is **a** [**directed graph**](https://en.wikipedia.org/wiki/Directed_graph) **whose** [**nodes**](https://en.wikipedia.org/wiki/Node_(graph_theory)) **are positions in a** [**game**](https://en.wikipedia.org/wiki/Game) **and whose** [**edges**](https://en.wikipedia.org/wiki/Edge_(graph_theory)) **are moves.** The **complete game tree** **for a game is the game tree starting at the initial position and containing all possible moves from each position;** the complete tree is the same tree as that obtained from the [extensive-form game](https://en.wikipedia.org/wiki/Extensive-form_game) representation.

* What is a "ply"?

In two-player [sequential games](https://en.wikipedia.org/wiki/Sequential_game), a **ply** is **one turn taken by one of the players**. The word is used to clarify what is meant when one might otherwise say "turn".

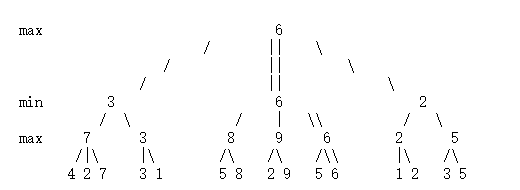
The word "turn" can be a problem since it means different things in different traditions. For example, in standard [chess](https://en.wikipedia.org/wiki/Chess) terminology, one *move* consists of a turn by each player; therefore a ply in chess is a *half-move*.

* Zero-sum games

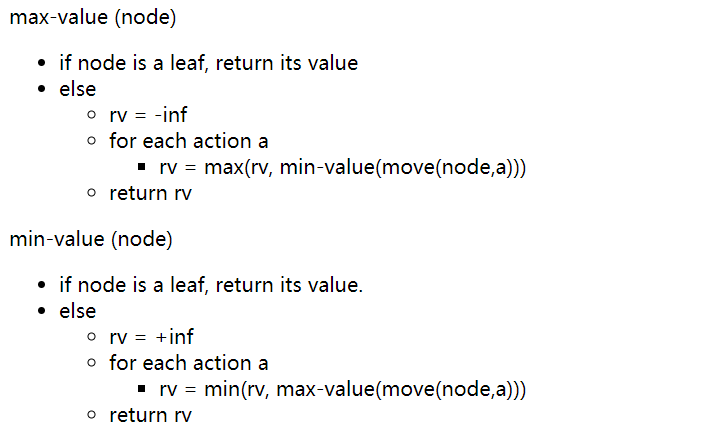
Final rewards for the two (or more) players sum to a **constant**. **(not necessarily zero)** That is, a gain for one player involves equivalent losses for the other player(s)

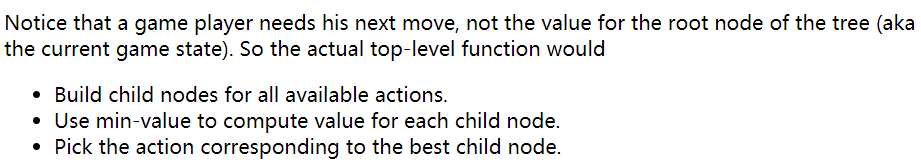
* Basic method
  + Minimax strategy

Minimax search propagates values up the tree, level by level.

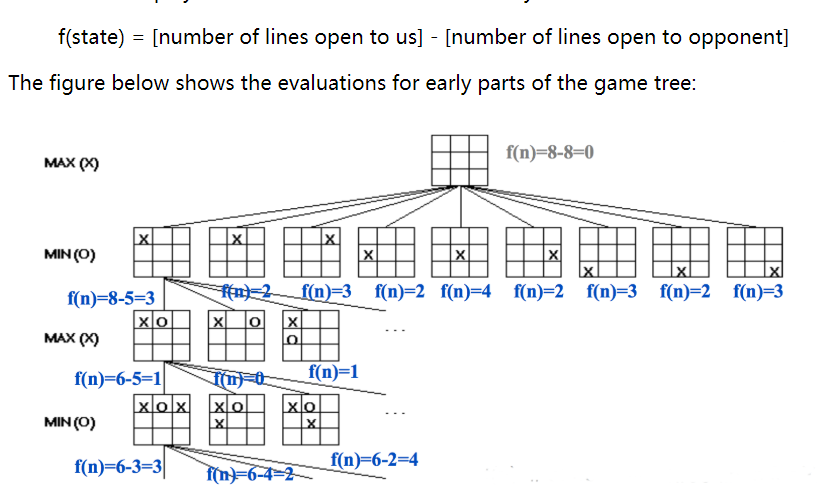


* + Minimax search (using depth-first search)





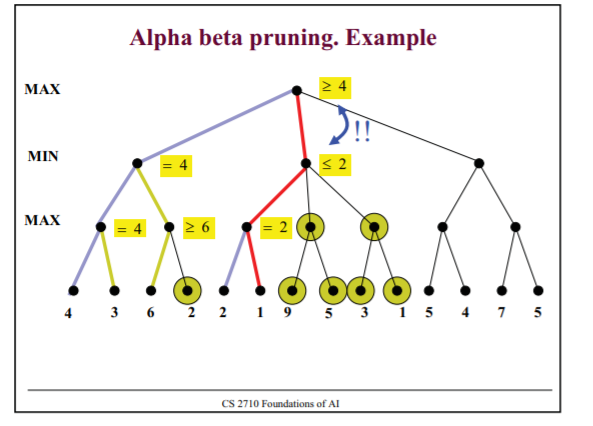
* + Depth cutoff
  + Heuristic state evaluation: number of open 3 lines for players



* Alpha-beta pruning

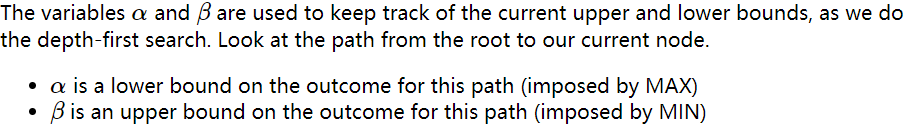
we can often compute the utility of the root without examining all the nodes in the game tree.

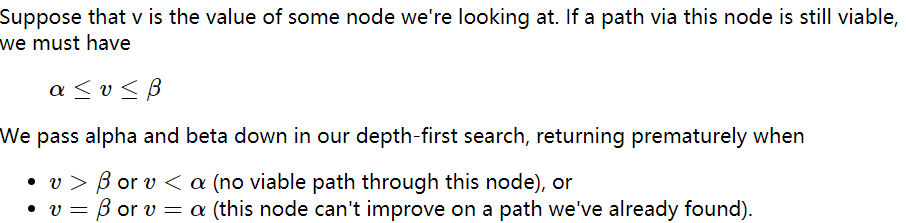
* + How it works



* + How well it works

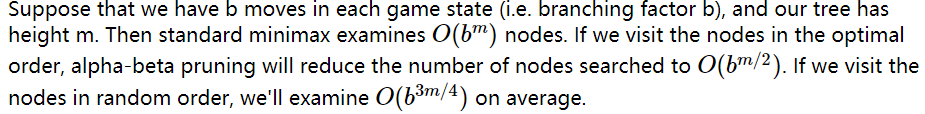
Once we've seen some of the lefthand children of a node, this gives us a preliminary value for that node. For a max node, this is a lower bound on the value at the node. For a min node, this gives us an upper bound. Therefore, as we start to look at additional children, we can abandon the exploration once it becomes clear that the new child can't beat the value that we're currently holding.

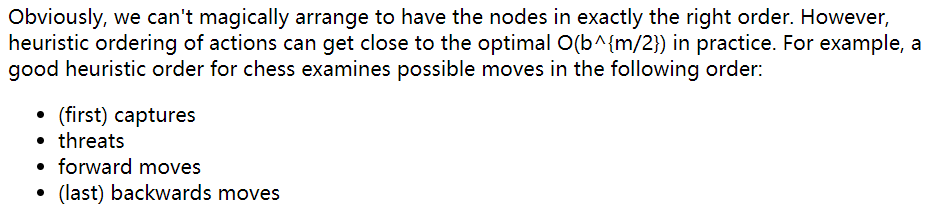




* + Impact of move ordering

Notice that the left-to-right order of nodes matters to the performance of alpha-beta pruning





* + You will not have to write out detailed code for alpha-beta search. Concentrate on understanding what branches it prunes and why.
* Optimizations around depth cutoff
  + Horizon effect

For large games, we have to stop expanding nodes at some cutoff depth. So we won’t see important changes to the game state that happen soon after that number of moves.

we can heuristically choose to stop search somewhat above/below the target depth rather than applying the cutoff rigidly.

* + Quiescence search

extend search further if position is "unstable" (e.g. piece in danger).

* + Singular extension

try a few especially strong moves past the cutoff.

* + Pruning unpromising branches

Evaluate states before we reach cutoff depth. Prune hopeless states without expanding them to cutoff depth

* Other optimizations
  + Memoization (transposition table)

a common way to hit the same state twice is to have two moves (e.g. involving unrelated parts of the board) that can be done in either order.

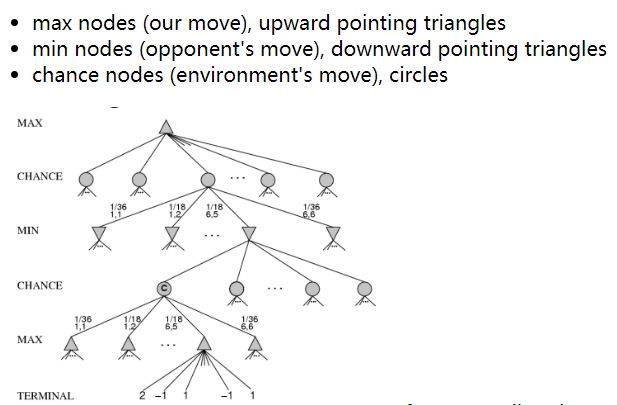
* + Opening moves, endgames

Special data tables for opening moves and endgames.

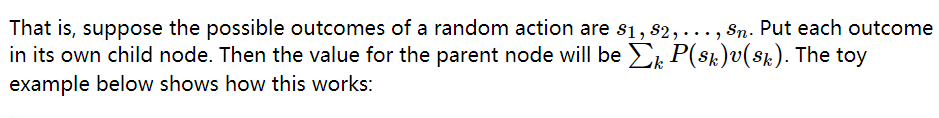
* Stochastic games, games with imperfect information

games with imperfect information

* + How to model in a game tree

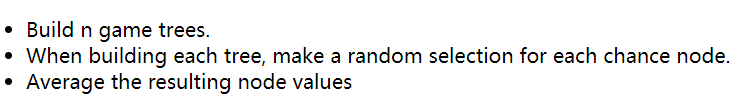


Expectiminimax

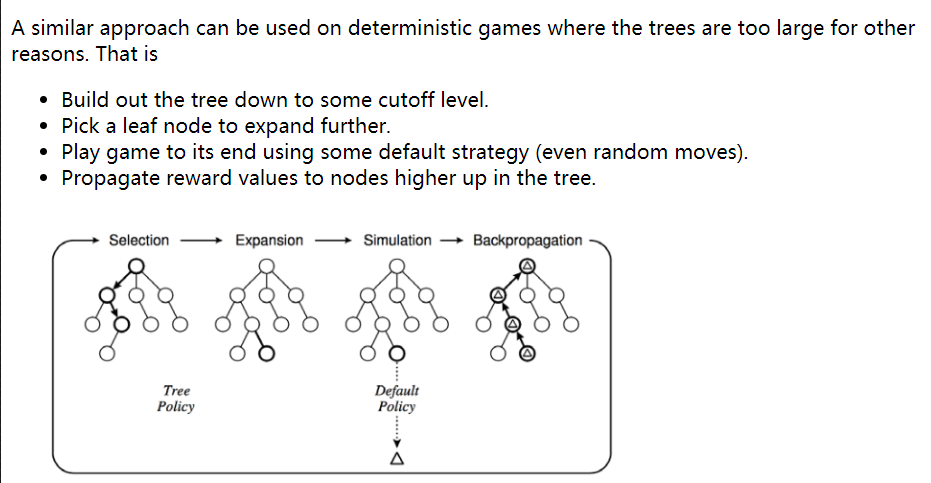


chance nodes add enough level to the tree and, in some cases (e.g. card games), they can have **a high branching factor**. So games involving random section (e.g. poker) can quickly become hard to solve by direct search.

Monte Carlo tree search



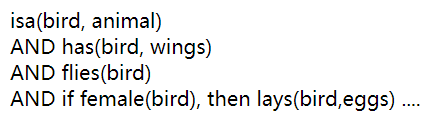
This method limits to the correct node values as n gets large. In practice, we make n as large as resources allow and live with the resulting approximation error.

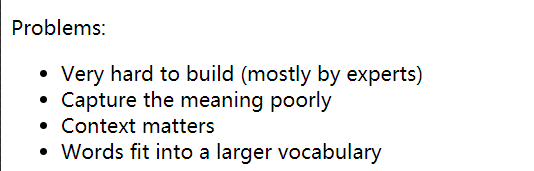


## **Vector Semantics**

* Logic-based vs. context-based representations of meaning

Logical based:



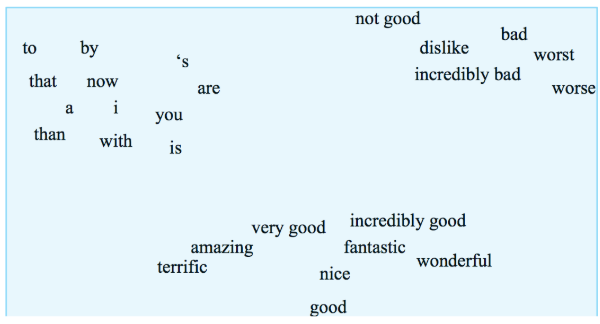


Context Based:

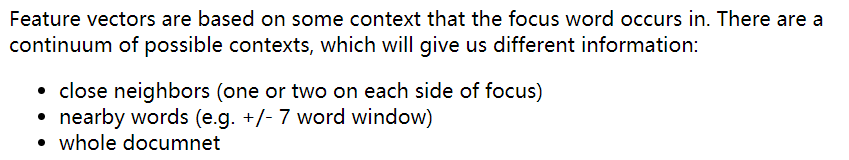
**You shall know a word by the company it keeps**

**represent each word as a vector of numerical feature values.**

These feature vectors are called word embeddings.



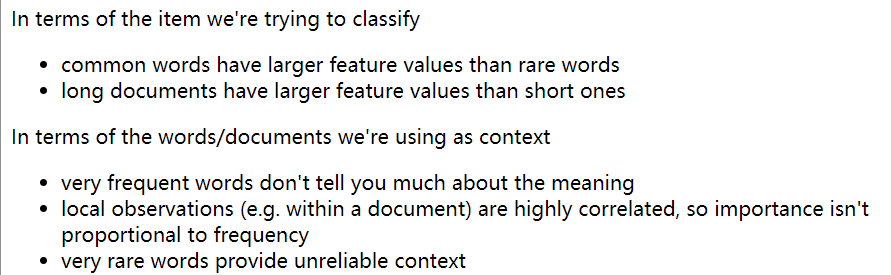
* + principle of contrast: The "Principle of Contrast" **states that differences in form imply differences in meaning.** (Eve Clark 1987, though idea goes back earlier). For example, kids will say things like "that's not an animal, that's a dog." Adults have a better model that words can refer to more or less general categories of objects, but still make mistakes like "that's not a real number, it's an integer." Apparent synonyms seem to inspire a search for some small difference in meaning. For example, how is "graveyard" different from "cemetery"? Perhaps cemeteries are prettier, or not adjacent to a church, or fancier, or ... **cups or bowl**
* how to get vectors:



* Normalization and smoothing
  + What's wrong with the raw vectors?

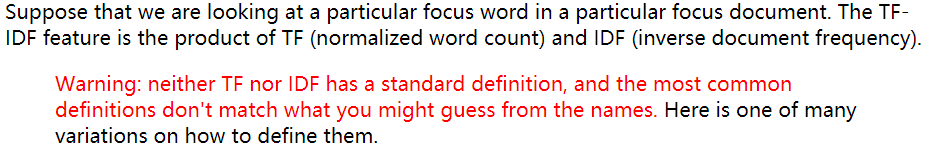
The raw counts require normalization.

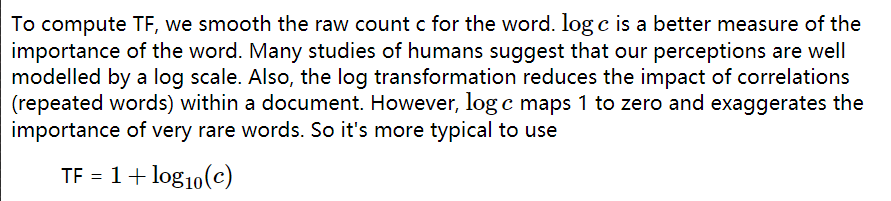
Vectors are very long and too sparse, so we need to map them into a lower-dimensional space.

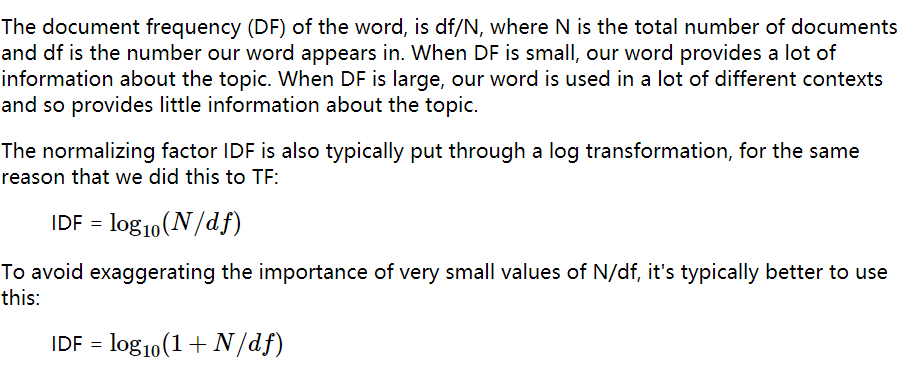


* + TF-IDF log10

TF-IDF normalization maps word counts into a better measure of their importance for classification.



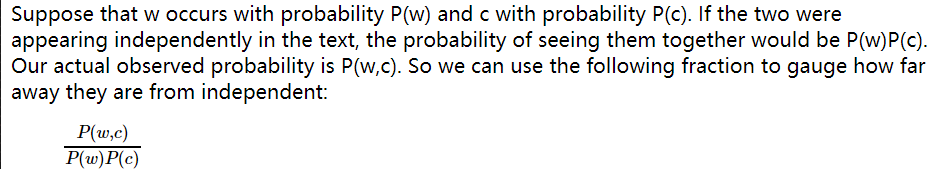


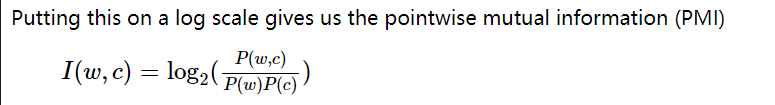


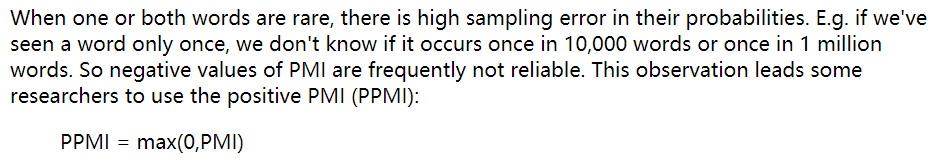
TF\*IDF(difference between “1+”)

* + PMI and PPMI log2 (Positive Pointwise Mutual Information)

modelling a word's meaning on the basis of words seen near it in running text.







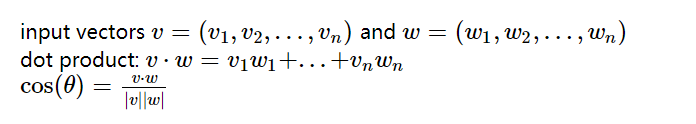
* + When would a PMI value be negative? When would negative values be reliable vs just noise?

When **one or both words are rare,** there is high sampling error in their probabilities. E.g. if we've seen a word only once, we don't know if it occurs once in 10,000 words or once in 1 million words. So negative values of PMI are frequently not reliable.

**if both words are quite common, they would be reliable.**

For example, "the of" is infrequent because it violates English grammar.

* Formal representations for words
  + one-hot representations for words: the key issue is that one-hot representations of words don't scale well. E.g. if we have a 100,000 word vocabulary, we use vectors of length 100,000. We should be able to give each word a distinct representation using a lot fewer parameters.
  + word embeddings/feature vectors: We're hoping to map ("embed") each word into a position in a vector space with a modest number of dimensions (e.g. 100) such that similar words are near one another. Ideally so it looks clean and reasonable as in the picture below. Our measures of similarity will be based on what words occur near one another in a text data corpus, because that's the type of data that's available in quantity.
  + cosine/dot product similarity



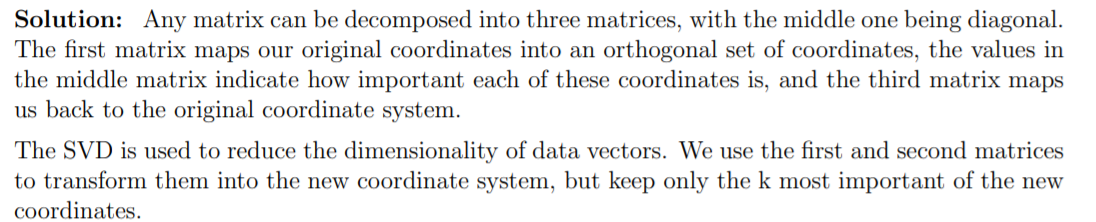
* Building feature vectors
  + relating words to documents

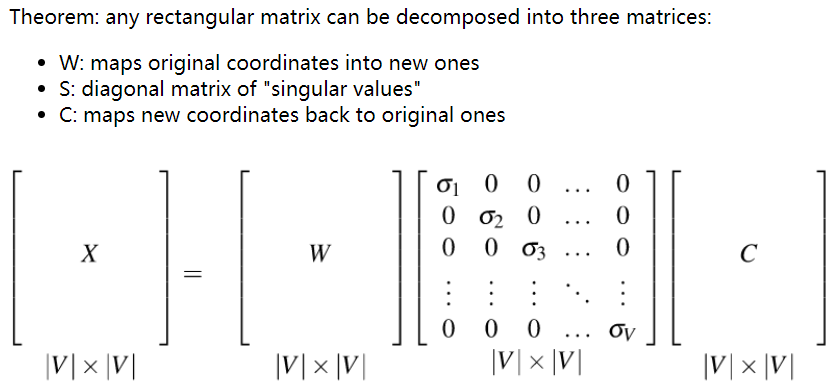
For example, we might count how often each word occurs in each of a group of documents. These counts give each word a vector of numbers, one per document.

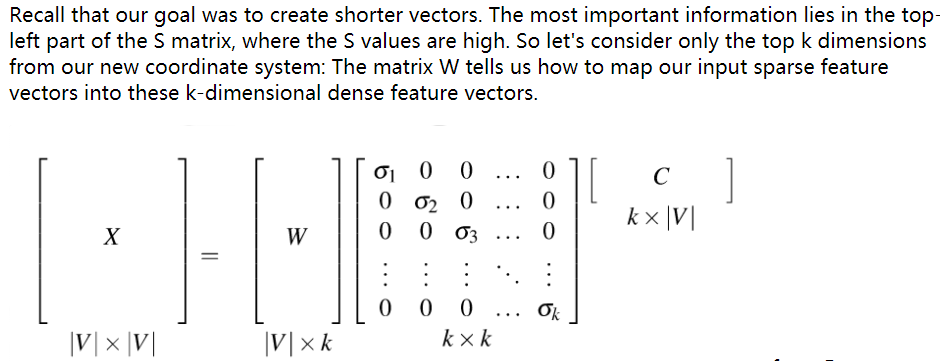
* + relating words to words

we can use nearby words as context. Our 2D data table will relate each focus word to each context word.

* Singular value decomposition (Principal Components Analysis)

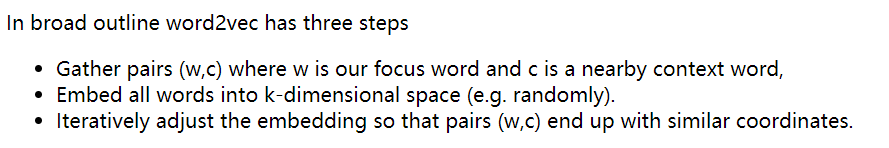


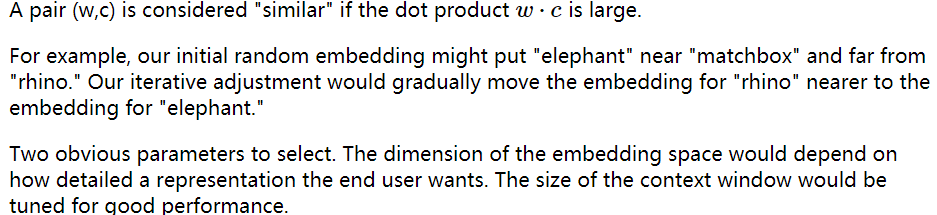




X = WSC

* Word2vec
  + Main outline of algorithm

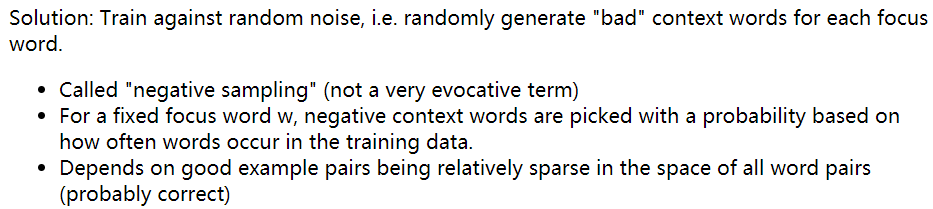




* + Negative sampling

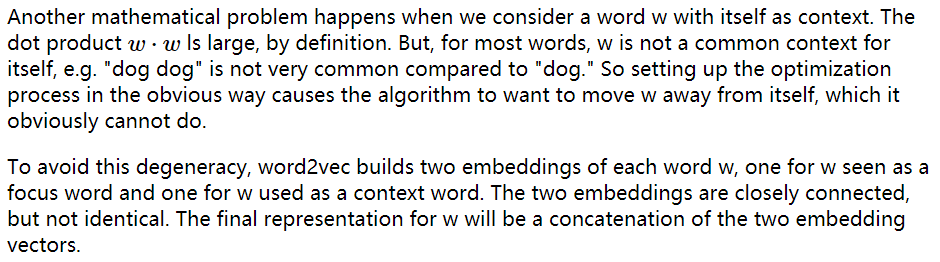
our data doesn't provide negative examples.

a great solution for the above optimization problem is to map all words to the same embedding vector. That defeats the purpose of having word embeddings.

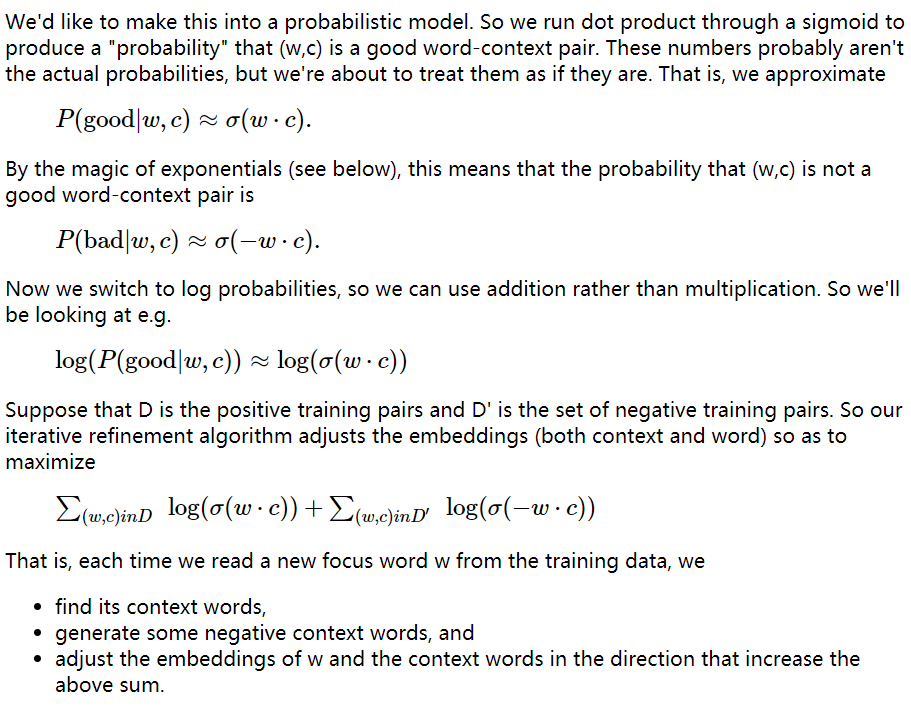


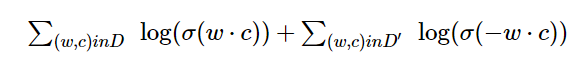
The negative training data will be corrupted by containing some good examples, but this corruption should be **a small percentage of the negative training data.**

* + Two embedding

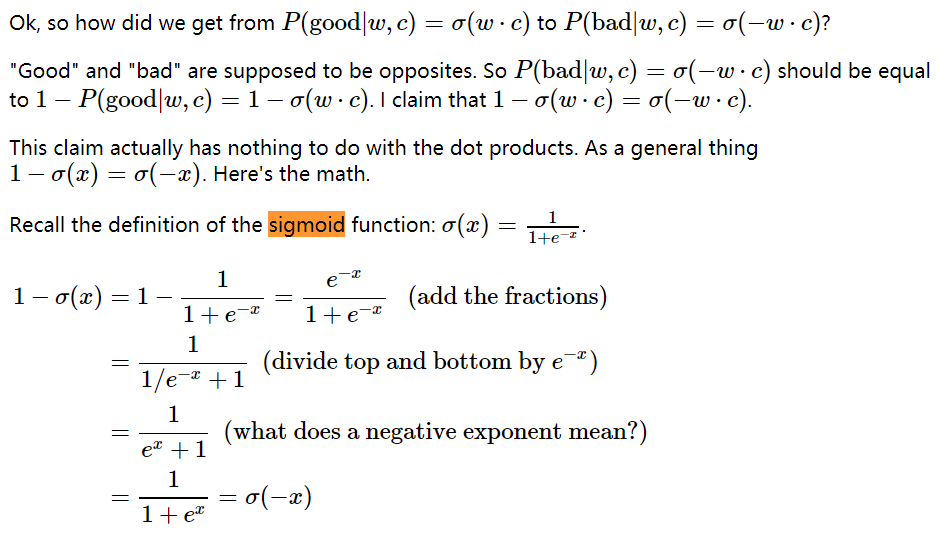


* + How to create the vector space



Maximize 

* + Sigmoid function (definition, relating probabilities of good and bad)



* + Why are words and contexts embedded separately?

To process when w == c, avoid degeneracy like “dog dog” pair

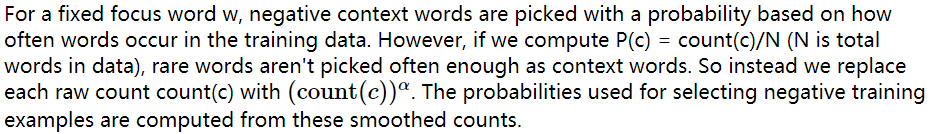
Two embedding

* Build training examples:

In the basic algorithm, we consider the input (focus) words one by one. For each focus word, we extract all words within +/- k positions as positive context words.

We also randomly generate a set of negative context words. This produces a set of positive pairs (w,c) and a set of negative pairs (w,c') that are used to update the embeddings of w, c, and c'.

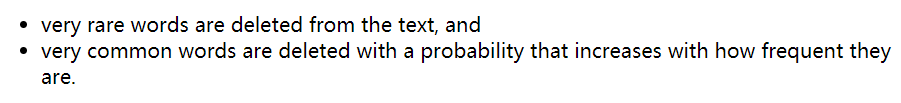
* Word2vec details
  + **Uses more negative examples than positive ones by a factor of 2 up to 20**
  + Postive training examples are weighted by 1/m, where m is the distance between the focus and context word. I.e. so adjacent context words are more important than words with a bit of separation.
  + Raising context counts to a power



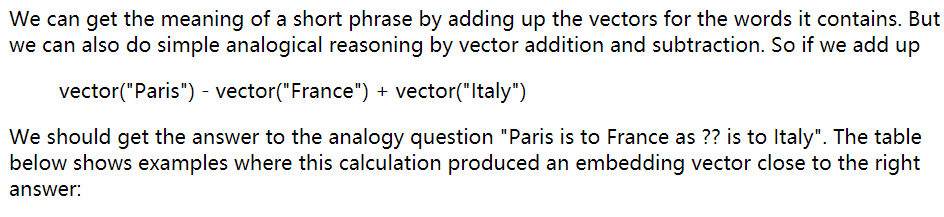
α is usually set to 0.75.

need normalize after this

* + Deleting rare words, subsampling frequent ones



* How does word2vec model analogies and compositional semantics?



* How do we evaluate word embedding methods?

