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
    - L2 norm: (y-p)^2
    - cross-entropy loss: -(y log p + (1-y)log(1-p))

    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.

    When using the L2 norm to measure distances, one normally takes the square of each individual error, adds them all up, and then takes the square root of the result. In our case, we can lose the square root because it doesn't affect the minimization. (If f(x) is non-negative, $\sqrt{f(x)}$ has a minimum when f(x) does.) The quantity we are trying to minimize is the sum of the squared errors over all the training input pairs. So, when looking at one individual pair, we just square the error for that pair.

    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.

  To work out the update rule for learning weight values, suppose we have a training example (x,y). The output with our current weights is $f_w(x) = \sigma(w * x)$ where σ is the sigmoid function. Our loss function is $loss(w, x) : (y - f_w(x))^2$.

  For gradient descent, we adjust our weights in the direction of the gradient. That is, our update rule should look like:

  $$w_i = w_i - \alpha * \frac{\partial \text{loss(w,x)}}{\partial w_i}$$

  To figure out the formula for $\frac{\partial \text{loss(w,x)}}{\partial w_i}$, we apply the chain rule, look up the derivative of the logistic function, and set $y' = f_w(x)$ to make the equation easier to read. See Russell and Norvig 18.6 for the derivation. This gives us an update rule:

  $$w_i = w_i + \alpha * (y - y') * y' * (1 - y') * x_i$$

- 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:

  $$\text{loss(model,T)} + \lambda * \text{complexity(model)}$$

  The right way to measure model complexity depends somewhat on the task. A simple method for a linear classifier would be $\sum_i (w_i)^2$, where the w_i are the classifier's weights. Recall that squaring makes the L2 norm sensitive to outlier (aka wrong) input values. In this case, this behavior is a feature: this measure of model complexity will pay strong attention to unusually large weights and make the classifier less likely to use them. Linguistic methods based on "minimumum description length" are using a varint of the same idea.

  λ is a constant that balances the desire to fit the data with the desire to produce a simple model. This is yet another parameter that would need to be tuned experimentally using our development data. The picture below illustrates how the output of a neural net classifier becomes simpler as λ is increased. (The neural net in this example is more complex, with hidden units.)

  Small lambda means focus on the data: prone to overfitting

  Large lambda means not: not fitting

  $\sum_i (w_i)^2$, 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:

  There is no (reasonable) way to produce a single output variable that takes discrete values (one per class). So we use a "one-hot" representation of the correct output. A one-hot representation is a vector with one element per class. The target class gets the value 1 and other classes get zero. E.g. if we have 8 classes and we want to specify the third one, our vector will look like:

  ```
  [0, 0, 1, 0, 0, 0, 0, 0]
  ```

- 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.

  Specifically, softmax maps each v_i to $\frac{e^{v_i}}{\sum_i e^{v_i}}$. The exponentiation accentuates the contrast between the largest value and the others, and forces all the values to be positive. The denominator normalizes all the values into the range [0,1], so they look like probabilities (for folks who want to see them as probabilities).
  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

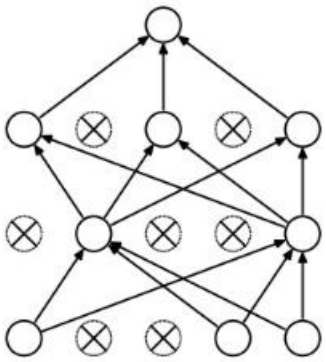
    $$w_i = w_i - \alpha * \frac{\partial \text{loss}}{\partial w_i}$$

    Notice that the loss is available at the output end of the network, but the weight w_i might be in an early layer. So the two are related by a chain of composed functions, often a long chain of functions. We have to compute the derivative of this composition.
  - 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?
    - Forward pass: starting from the input $\vec{x}$, calculate the output values for all units. We work forwards through the network, using our current weights.
    - Suppose the final output is y. We put y and y' into our loss function to calculate the final loss value.
    - Backpropagation: starting from the loss at the end of the network and working backwards, calculate how changes to each weight will affect the loss. This combines the derivatives of each individual function with the values from the forward pass.
  - 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



(b) After applying dropout.

- 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

  In order for training to work right, gradients computed during backprojection need to stay in a sensible range of sizes. A sigmoid activation function only works well when output numbers tend to stay in the middle area that has a significant slope.

  - gradients too small: numbers can underflow, also training can become slow
  - gradients too large: numbers can overflow

  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.
  2) Initialize weights so that different layers end up with the same variance in gradients

3) Gradient clipping: detect excessively high gradients and reduce them.
4) 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
    - "stride" = how many pixels we shift mask sideways between output units
    - "depth" = how many stacked features in each level (start maybe 3 for RGB)
  - 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.

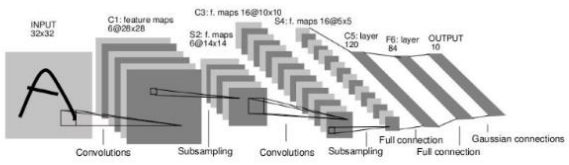
- Weight/parameter sharing
  units in the same layer share a common set of weights and bias. **This cuts down on the number of parameters to train. May worsen performance if different regions in the input images are expected to have different properties,** e.g. the object of interest is always centered.

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

A complete CNN typically contains three types of layers

- convolutional (large input/output, small masks)
- pooling
- fully-connected (usually towards end of the computation)

LeNet-5

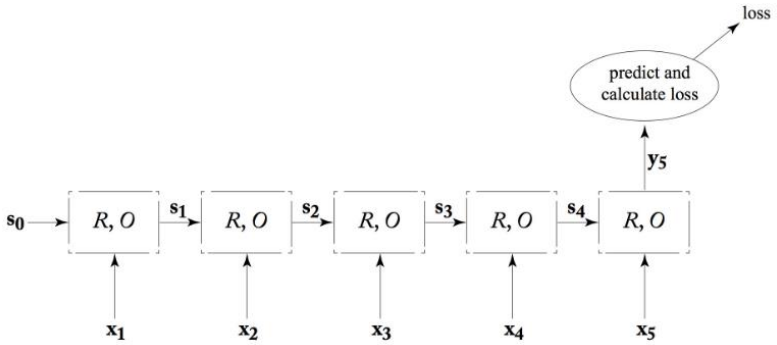


- Average pooling
- Sigmoid or tanh nonlinearity
- Fully connected layers at the end
- Trained on MNIST digit dataset with 60K training examples

- 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.

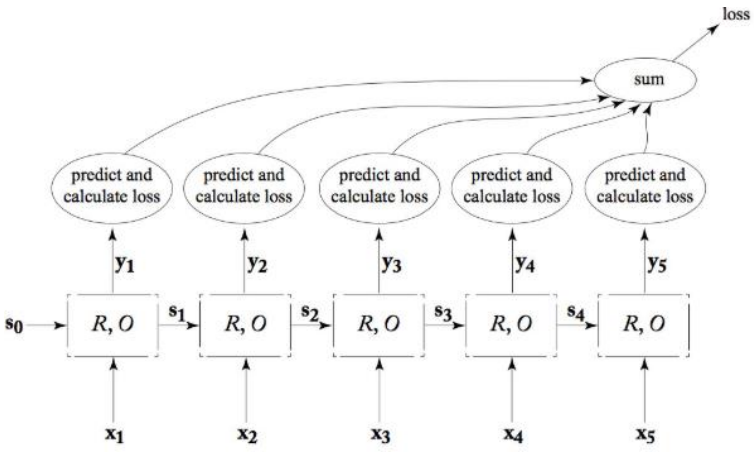
- Recurrent neural networks
  are neural nets that have connections that loop back from a layer to the same layer. The intent of the feedback loop in the picture is that each unit is connected to all the other units in the layer.
  - High-level view of how they work
    When unroll, all copies of the 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).



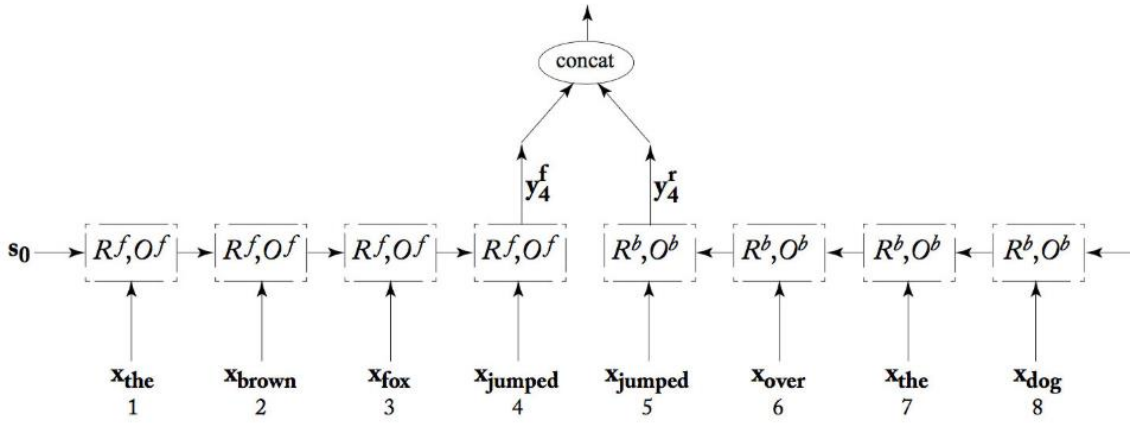
  - When would we compute loss from last unit vs. summed over all units?
    This kind of RNN can be trained in much the same way as a standard ("feedforward") neural net. However, values and error signals propagate **in the time direction.** The forward pass calculates values moving to the right. Backpropagation starts at the final loss node and moves back to the left. This is often called **"backpropagation through time."**

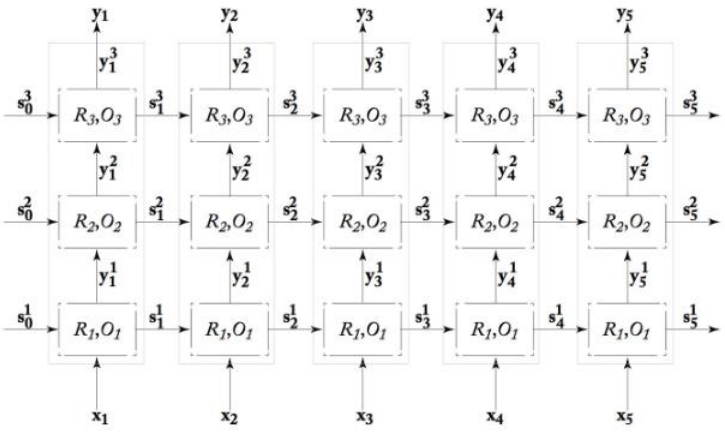
    When applying in NLP for modelling sequential data: mapping words to part-of-speech tags. In this case, we would care about getting the correct output at each timestep (not just the final one). So the connection to our loss function would look like this:



- 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 $s \in S$
  >
  > set of actions $a \in A$
  >
  > 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

     $U(s) = R(s) + \max_{a \in A} * U(move(a,s))$

  2) 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 = $\sum_{s' \in s} P(s'|s,a)U(s')$

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

     $U(s) = R(s) + \max_{a \in A} \sum_{s' \in S} * P(s'|s,a) * U(s')$

  3) 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.

The actual equation should also include a reward delay multiplier γ. 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. So the final "Bellman equation" looks like this:

$$U(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s'|s,a)U(s')$$

- Bellman equation Fixed Policy

$$U(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s,\pi(s))U(s')$$

Our goal is to find the best policy. This is tightly coupled to finding a closed form solution to the Bellman equation, i.e. an actual utility value for each state. Once we know the utilities, the best policy is move towards the neighbors with the highest utility.

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

    - Initialize $U_0(s) = 0$ for all s
    - For i=0 until values converge, update U using the equation

    $$U_{i+1}(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s'|s,a) U_i(s')$$

    Until converged

    From the final converged utility values, we can read off a final policy (bottom right) by essentially moving towards the neighbor with the highest utility. Taking into account the probabilistic nature of our actions, we get the equation

    $$\pi(s) = \operatorname{argmax}_a \sum_{s'} P(s'|s,a) U(s')$$

  - Policy iteration: Policy iteration produces the same solution, but faster.

    - Start with an initial guess for policy π.
    - Alternate two steps:
      - Policy evaluation: use policy $\pi$to estimate utility values U
      - Policy improvement: use utility values U to calculate a new policy π

    ## Utility from estimated policy

    Our Bellman equation (below) finds the value corresponding to the best action we might take in state s.

    $$U(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s,a) U(s')$$

    However, in policy iteration, we already have a draft policy. So we do a similar computation but assuming that we will command action $\pi(s)$.

    $$U(s) = R(s) + \gamma \sum_{s'} P(s'|s,\pi(s)) U(s')$$

    We have one of these equations for each state s. The simplification means that each equation is linear. So we have two options for finding a solution:

    - linear algebra
    - a few iterations of value iteration

    The value estimation approach is usually faster. We don't need an exact (fully converged) solution, because we'll be repeating this calculation each time we refine our policy π.

- Asynchronous dynamic programming

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

  One useful weak to solving Markov Decision Process is "asynchronous dynamic programming." In each iteration, it's not necessary to update all states. We can select only certain states for updating. E.g.

  - states frequently seen in some application (e.g. a game)
  - states for which the Bellman equation has a large error (i.e. compare values for left and right sides of the equation)

- RL

  A reinforcement learner may be

  - Online: interacting directly with the world or
  - Offline: interacting with a simulation of some sort.

  An important hybrid is "experience replay." In this case, we have an online learner. But instead of forgetting its old experiences, it will

  - remember old training sequences,
  - spend some training time replaying these experiences rather than taking new actions in the world

- RL learning loop

  A reinforcement learner repeats the following sequence of steps:

  - take an action
  - observe the outcome (state and reward)
  - update internal representation

  There are two basic choices for the "internal representation":

  - Model-based: explicitly estimate values for $P(s'|s,a)$ and $R(s)$
  - Model-free: estimate "Q" values, which sidestep the need to estimate P and R

- Model-based reinforcement learning

> A model based learner operates much like a naive Bayes learning algorithm, except that it has to start making decisions as it trains. Initially it would use some default method for picking actions (e.g. random). As it moves around taking actions, it tracks counts for what rewards it got in different states and what state transitions resulted from its commanded actions. Periodically it uses these counts to
>
> - Estimate P(S' | s,a) and R(s), and then
> - Use values for P and R to estimate U(s) and $\pi(s)$

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.

    - $Q(s,a) = R(s) + \gamma \sum_{s'} P(s'|s,a)U(s')$
    - $U(s) = \max_a Q(s,a)$

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

$$U(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s,a)U(s')$$

Remove U(s')

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

Remove P

$$Q(s,a) = R(s) + \gamma \max_{a'} Q(s',a')$$

- TD update algorithm

Q-Value Update equations:

One way to calculate the average of a series of values $V_1 \ldots V_n$ is to start with an initial estimate of zero and average in each incoming value with a an appropriate discount. That is

- V = 0
- loop for t=1 to n
  - $V = V + V_t/n$

Now suppose that we have an ongoing sequence with no obvious endpoint. Or perhaps the sequence is so long that the world might gradually change, so we'd like to slowly adapt to change. Then we can do a similar calculation called a moving average:

- V = 0
- loop for t=1 to forever
  - V = $(1-\alpha)V + \alpha V_t$

If we use this to average our values Q(s,a), we get an update method that looks like:

- Q(s,a) = 0 for all s and all a
- for t = 1 to forever
  - $Q(s,a) = (1-\alpha)Q(s,a) + \alpha[R(s) + \gamma \max_{a'} Q(s',a')]$

Our update equation can also be written as

$$Q(s,a) = Q(s,a) + \alpha[R(s) - Q(s,a) + \gamma \max_{a'} Q(s',a')]$$

TD update Algorithm:

- Q(s,a) = 0 for all s and all a
- for t = 1 to forever
    1. in the current state s, select an action a
    2. observe the reward R(s) and the next state s'
    3. update our Q values $Q(s,a) = Q(s,a) + \alpha[R(s) - Q(s,a) + \gamma \max_{a'} Q(s',a')]$

The obvious choice for an action to command in step 1 is

$$\pi(s) = \text{argmax}_a Q(s,a)$$

But recall from last lecture that the agent also needs to explore. So a better method is

- Sometimes choose $\pi(s) = \text{argmax}_a Q(s,a)$
- Otherwise choose a random or semi-random exploration state.

This creates an inconsistency in our algorithm.

- The maximization $\max_{a'} Q(s',a')$ in step 3 assumes we pick the action that will yield the highest value, but
- Step 1 doesn't always choose the action that currently seems to have the highest value.

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

- pick an initial state s and initial action a
- for t = 1 to forever
    - observe the reward R(s) and the next state s'
    - from next state s', select next action a'
    - update our Q values using $Q(s,a) = Q(s,a) + \alpha[R(s) - Q(s,a) + \gamma Q(s',a')]$
    - (s,a) = (s',a')

- How do TD and SARSA differ?

Solution: In both cases, we're in a state s, we've picked an action a and found the next state s'. TD updates Q(s,a) using the maximum Q(s',a') value over all possible next actions a'. SARSA picks the next action a', then updates Q(s,a) using Q(s',a') for that specific action a'.

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, a **game tree** is **a directed graph whose nodes are positions in a game and whose edges 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 representation.

- What is a "ply"?

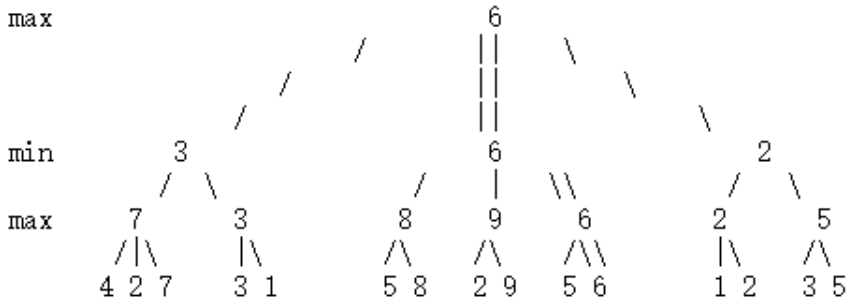
  In two-player sequential games, a **ply** is **one turn taken by one of the players**. The word is used to clarify what is meant when one might otherwise say "turn".

  The word "turn" can be a problem since it means different things in different traditions. For example, in standard chess 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)

  max-value (node)

  - if node is a leaf, return its value
  - else
    - rv = -inf
    - for each action a
      - rv = max(rv, min-value(move(node,a)))
    - return rv

  min-value (node)

  - if node is a leaf, return its value.
  - else
    - rv = +inf
    - for each action a
      - rv = min(rv, max-value(move(node,a)))
    - return rv

  Notice that a game player needs his next move, not the value for the root node of the tree (aka the current game state). So the actual top-level function would

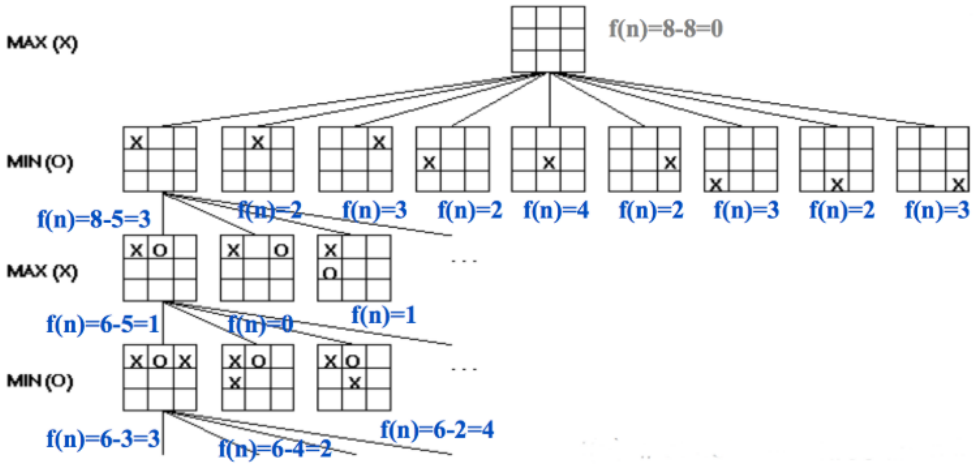
  - Build child nodes for all available actions.
  - Use min-value to compute value for each child node.
  - Pick the action corresponding to the best child node.

- Depth cutoff

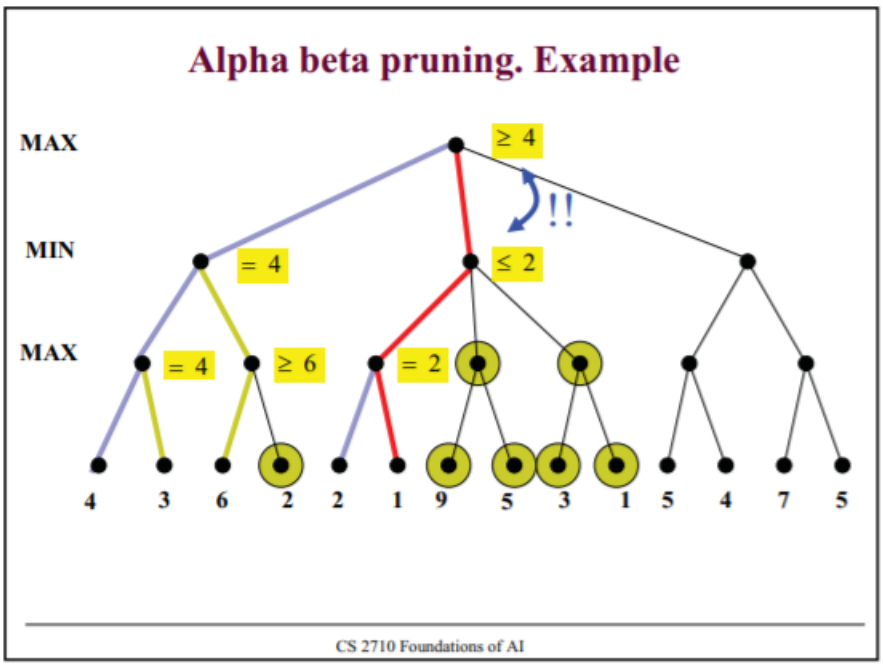
- Heuristic state evaluation: number of open 3 lines for players

  f(state) = [number of lines open to us] - [number of lines open to opponent]

  The figure below shows the evaluations for early parts of the game tree:



- 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.

The variables α and β are used to keep track of the current upper and lower bounds, as we do the depth-first search. Look at the path from the root to our current node.

- α is a lower bound on the outcome for this path (imposed by MAX)
- β is an upper bound on the outcome for this path (imposed by MIN)

Suppose that v is the value of some node we're looking at. If a path via this node is still viable, we must have

$$\alpha \leq v \leq \beta$$

We pass alpha and beta down in our depth-first search, returning prematurely when

- $v > \beta$ or $v < \alpha$ (no viable path through this node), or
- $v = \beta$ or $v = \alpha$ (this node can't improve on a path we've already found).

- Impact of move ordering

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

  Suppose that we have b moves in each game state (i.e. branching factor b), and our tree has height m. Then standard minimax examines $O(b^m)$ nodes. If we visit the nodes in the optimal order, alpha-beta pruning will reduce the number of nodes searched to $O(b^{m/2})$. If we visit the nodes in random order, we'll examine $O(b^{3m/4})$ on average.

  Obviously, we can't magically arrange to have the nodes in exactly the right order. However, heuristic ordering of actions can get close to the optimal O(b^{m/2}) in practice. For example, a good heuristic order for chess examines possible moves in the following order:

  - (first) captures
  - threats
  - forward moves
  - (last) backwards moves

- 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

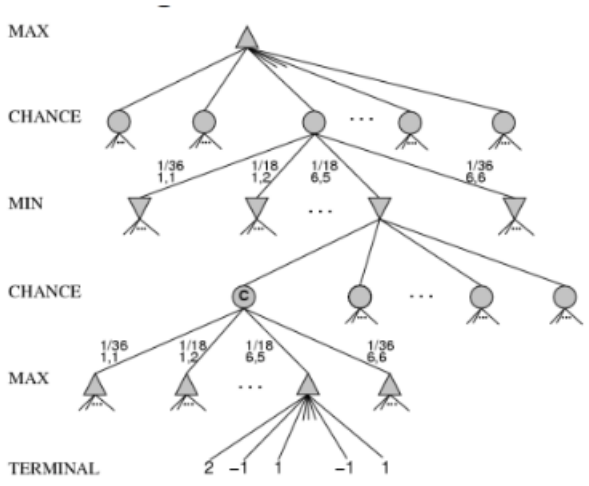
    try a few especially strong moves past the cutoff.
  - Pruning unpromising branches

    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
    - max nodes (our move), upward pointing triangles
    - min nodes (opponent's move), downward pointing triangles
    - chance nodes (environment's move), circles



Expectiminimax

That is, suppose the possible outcomes of a random action are $s_1, s_2, \ldots, s_n$. Put each outcome in its own child node. Then the value for the parent node will be $\sum_k P(s_k)v(s_k)$. The toy example below shows how this works:

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.

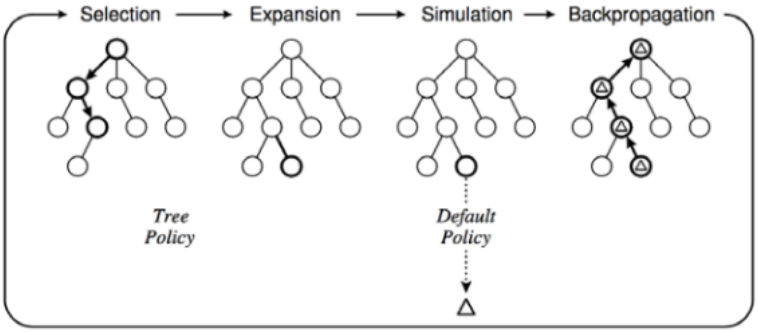
Monte Carlo tree search

- Build n game trees.
- When building each tree, make a random selection for each chance node.
- Average the resulting node values

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.

A similar approach can be used on deterministic games where the trees are too large for other reasons. That is

- Build out the tree down to some cutoff level.
- Pick a leaf node to expand further.
- Play game to its end using some default strategy (even random moves).
- Propagate reward values to nodes higher up in the tree.



Vector Semantics

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

  Logical based:

  isa(bird, animal)
  AND has(bird, wings)
  AND flies(bird)
  AND if female(bird), then lays(bird,eggs) ....

  Problems:

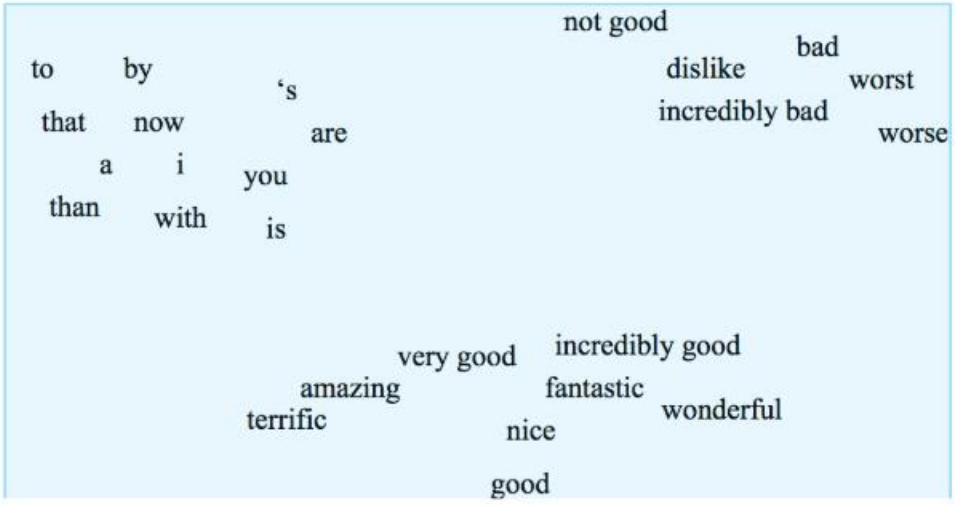
  - Very hard to build (mostly by experts)
  - Capture the meaning poorly
  - Context matters
  - Words fit into a larger vocabulary

  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:

  Feature vectors are based on some context that the focus word occurs in. There are a continuum of possible contexts, which will give us different information:

  - close neighbors (one or two on each side of focus)
  - nearby words (e.g. +/- 7 word window)
  - whole documnet

- 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.

    > In terms of the item we're trying to classify
    >
    > - common words have larger feature values than rare words
    > - long documents have larger feature values than short ones
    >
    > In terms of the words/documents we're using as context
    >
    > - very frequent words don't tell you much about the meaning
    > - local observations (e.g. within a document) are highly correlated, so importance isn't proportional to frequency
    > - very rare words provide unreliable context

- TF-IDF log10

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

  Suppose that we are looking at a particular focus word in a particular focus document. The TF-IDF feature is the product of TF (normalized word count) and IDF (inverse document frequency).

  > Warning: neither TF nor IDF has a standard definition, and the most common definitions don't match what you might guess from the names. Here is one of many variations on how to define them.

  > To compute TF, we smooth the raw count c for the word. $\log c$ is a better measure of the importance of the word. Many studies of humans suggest that our perceptions are well modelled by a log scale. Also, the log transformation reduces the impact of correlations (repeated words) within a document. However, $\log c$ maps 1 to zero and exaggerates the importance of very rare words. So it's more typical to use
  >
  > $$\text{TF} = 1 + \log_{10}(c)$$

  The document frequency (DF) of the word, is df/N, where N is the total number of documents and df is the number our word appears in. When DF is small, our word provides a lot of information about the topic. When DF is large, our word is used in a lot of different contexts and so provides little information about the topic.

  The normalizing factor IDF is also typically put through a log transformation, for the same reason that we did this to TF:

  $$\text{IDF} = \log_{10}(N/df)$$

  To avoid exaggerating the importance of very small values of N/df, it's typically better to use this:

  $$\text{IDF} = \log_{10}(1 + N/df)$$

  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.

  > Suppose that w occurs with probability P(w) and c with probability P(c). If the two were appearing independently in the text, the probability of seeing them together would be P(w)P(c). Our actual observed probability is P(w,c). So we can use the following fraction to gauge how far away they are from independent:
  >
  > $$\frac{P(w,c)}{P(w)P(c)}$$

> Putting this on a log scale gives us the pointwise mutual information (PMI)
>
> $$I(w,c) = \log_2(\frac{P(w,c)}{P(w)P(c)})$$

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. This observation leads some researchers to use the positive PMI (PPMI):

PPMI = max(0,PMI)

- 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

    input vectors $v = (v_1, v_2, \ldots, v_n)$ and $w = (w_1, w_2, \ldots, w_n)$
    dot product: $v \cdot w = v_1 w_1 + \ldots + v_n w_n$
    $\cos(\theta) = \frac{v \cdot w}{|v||w|}$

- 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)

Solution: Any matrix can be decomposed into three matrices, with the middle one being diagonal. The first matrix maps our original coordinates into an orthogonal set of coordinates, the values in the middle matrix indicate how important each of these coordinates is, and the third matrix maps us back to the original coordinate system.

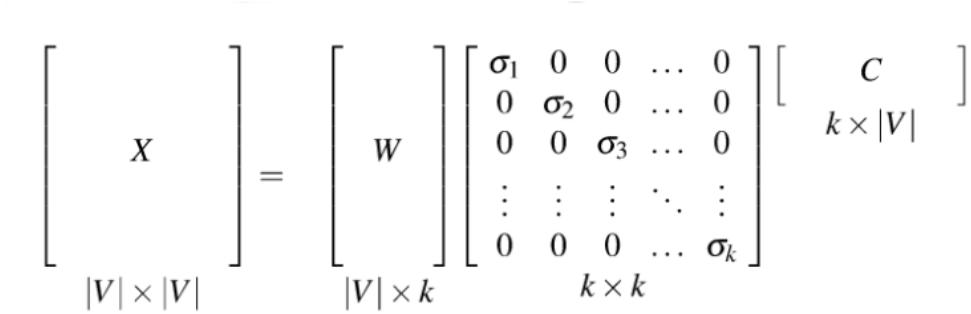
The SVD is used to reduce the dimensionality of data vectors. We use the first and second matrices to transform them into the new coordinate system, but keep only the k most important of the new coordinates.

Theorem: any rectangular matrix can be decomposed into three matrices:

- W: maps original coordinates into new ones
- S: diagonal matrix of "singular values"
- C: maps new coordinates back to original ones

$$\underset{|V|\times|V|}{\begin{bmatrix} \\ X \\ \\ \end{bmatrix}} = \underset{|V|\times|V|}{\begin{bmatrix} \\ W \\ \\ \end{bmatrix}} \underset{|V|\times|V|}{\begin{bmatrix} \sigma_1 & 0 & 0 & \dots & 0 \\ 0 & \sigma_2 & 0 & \dots & 0 \\ 0 & 0 & \sigma_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \sigma_V \end{bmatrix}} \underset{|V|\times|V|}{\begin{bmatrix} \\ C \\ \\ \end{bmatrix}}$$

Recall that our goal was to create shorter vectors. The most important information lies in the top-left part of the S matrix, where the S values are high. So let's consider only the top k dimensions from our new coordinate system: The matrix W tells us how to map our input sparse feature vectors into these k-dimensional dense feature vectors.



X = WSC

- Word2vec
  - Main outline of algorithm

    In broad outline word2vec has three steps

    - Gather pairs (w,c) where w is our focus word and c is a nearby context word,
    - Embed all words into k-dimensional space (e.g. randomly).
    - Iteratively adjust the embedding so that pairs (w,c) end up with similar coordinates.

    A pair (w,c) is considered "similar" if the dot product $w \cdot c$ is large.

    For example, our initial random embedding might put "elephant" near "matchbox" and far from "rhino." Our iterative adjustment would gradually move the embedding for "rhino" nearer to the embedding for "elephant."

    Two obvious parameters to select. The dimension of the embedding space would depend on how detailed a representation the end user wants. The size of the context window would be tuned for good performance.
  - 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.

    Solution: Train against random noise, i.e. randomly generate "bad" context words for each focus word.

    - Called "negative sampling" (not a very evocative term)
    - For a fixed focus word w, negative context words are picked with a probability based on how often words occur in the training data.
    - Depends on good example pairs being relatively sparse in the space of all word pairs (probably correct)

    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

  Another mathematical problem happens when we consider a word w with itself as context. The dot product $w \cdot w$ Is large, by definition. But, for most words, w is not a common context for itself, e.g. "dog dog" is not very common compared to "dog." So setting up the optimization process in the obvious way causes the algorithm to want to move w away from itself, which it obviously cannot do.

  To avoid this degeneracy, word2vec builds two embeddings of each word w, one for w seen as a focus word and one for w used as a context word. The two embeddings are closely connected, but not identical. The final representation for w will be a concatenation of the two embedding vectors.

- How to create the vector space

  We'd like to make this into a probabilistic model. So we run dot product through a sigmoid to produce a "probability" that (w,c) is a good word-context pair. These numbers probably aren't the actual probabilities, but we're about to treat them as if they are. That is, we approximate

  $$P(\text{good}|w,c) \approx \sigma(w \cdot c).$$

  By the magic of exponentials (see below), this means that the probability that (w,c) is not a good word-context pair is

  $$P(\text{bad}|w,c) \approx \sigma(-w \cdot c).$$

  Now we switch to log probabilities, so we can use addition rather than multiplication. So we'll be looking at e.g.

  $$\log(P(\text{good}|w,c)) \approx \log(\sigma(w \cdot c))$$

  Suppose that D is the positive training pairs and D' is the set of negative training pairs. So our iterative refinement algorithm adjusts the embeddings (both context and word) so as to maximize

  $$\sum_{(w,c)inD} \log(\sigma(w \cdot c)) + \sum_{(w,c)inD'} \log(\sigma(-w \cdot c))$$

  That is, each time we read a new focus word w from the training data, we

  - find its context words,
  - generate some negative context words, and
  - adjust the embeddings of w and the context words in the direction that increase the above sum.

Maximize

$$\sum_{(w,c)inD} \log(\sigma(w \cdot c)) + \sum_{(w,c)inD'} \log(\sigma(-w \cdot c))$$

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

  Ok, so how did we get from $P(\text{good}|w,c) = \sigma(w \cdot c)$ to $P(\text{bad}|w,c) = \sigma(-w \cdot c)$?

  "Good" and "bad" are supposed to be opposites. So $P(\text{bad}|w,c) = \sigma(-w \cdot c)$ should be equal to $1 - P(\text{good}|w,c) = 1 - \sigma(w \cdot c)$. I claim that $1 - \sigma(w \cdot c) = \sigma(-w \cdot c)$.

  This claim actually has nothing to do with the dot products. As a general thing $1 - \sigma(x) = \sigma(-x)$. Here's the math.

  Recall the definition of the sigmoid function: $\sigma(x) = \frac{1}{1+e^{-x}}$.

$$
\begin{aligned}
1 - \sigma(x) &= 1 - \frac{1}{1+e^{-x}} = \frac{e^{-x}}{1+e^{-x}} \quad \text{(add the fractions)} \\
&= \frac{1}{1/e^{-x}+1} \quad \text{(divide top and bottom by } e^{-x}\text{)} \\
&= \frac{1}{e^{x}+1} \quad \text{(what does a negative exponent mean?)} \\
&= \frac{1}{1+e^{x}} = \sigma(-x)
\end{aligned}
$$

- 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

For a fixed focus word w, negative context words are picked with a probability based on how often words occur in the training data. However, if we compute P(c) = count(c)/N (N is total words in data), rare words aren't picked often enough as context words. So instead we replace each raw count count(c) with $(\text{count}(c))^{\alpha}$. The probabilities used for selecting negative training examples are computed from these smoothed counts.

α is usually set to 0.75.

need normalize after this

- Deleting rare words, subsampling frequent ones
  - very rare words are deleted from the text, and
  - very common words are deleted with a probability that increases with how frequent they are.

- How does word2vec model analogies and compositional semantics?

  We can get the meaning of a short phrase by adding up the vectors for the words it contains. But we can also do simple analogical reasoning by vector addition and subtraction. So if we add up

  vector("Paris") - vector("France") + vector("Italy")

  We should get the answer to the analogy question "Paris is to France as ?? is to Italy". The table below shows examples where this calculation produced an embedding vector close to the right answer:

- How do we evaluate word embedding methods?

  Word embedding models are evaluated in two ways:

  - Analogy and similar tasks (e.g. Paris is to France as ? is to Italy)
  - Using the embedding as the first stage for solving a larger task (e.g. question answering)