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.

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.

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

Regularization: overcome overfitting

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

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

Small lambda means focus on the data: prone to overfitting
Large lambda means not: not fitting

Use $\sum_i (w_i)^2$, 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, one element per class

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。

Neural Nets

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.

Chain rule:

- y = g(x)
- z = f(y) = f(g(x))
- chain rule: $\frac{dz}{dx} = \frac{dz}{dy} * \frac{dy}{dx}$

Forward value

network, we can assemble the value for $\frac{d\,\text{loss}}{dw}$ using

- the derivative of each individual function
- the chain rule: multiply those derivatives
- the forward values: substitute to eliminate all the intermediate variables

- 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.

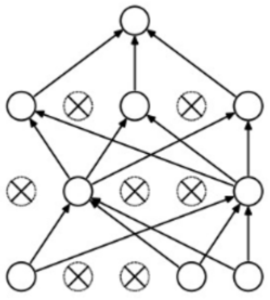
To break symmetry:

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** 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).

How does convolution layer work:

1) 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."
2) each network layer has a significant thickness, i.e. a number of different values at each (x,y) location.

- "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.

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.

A complete CNN typically contains three types of layers

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

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.

[illegible]

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.

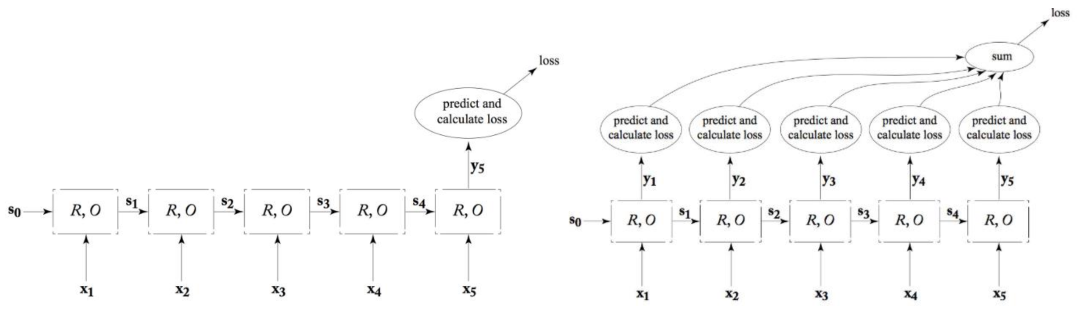
When unroll, all copies of the unit share the same parameter values.

An RNN can be used as a classifier. That is, the system using the RNN only cares about the output from the last timestep

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

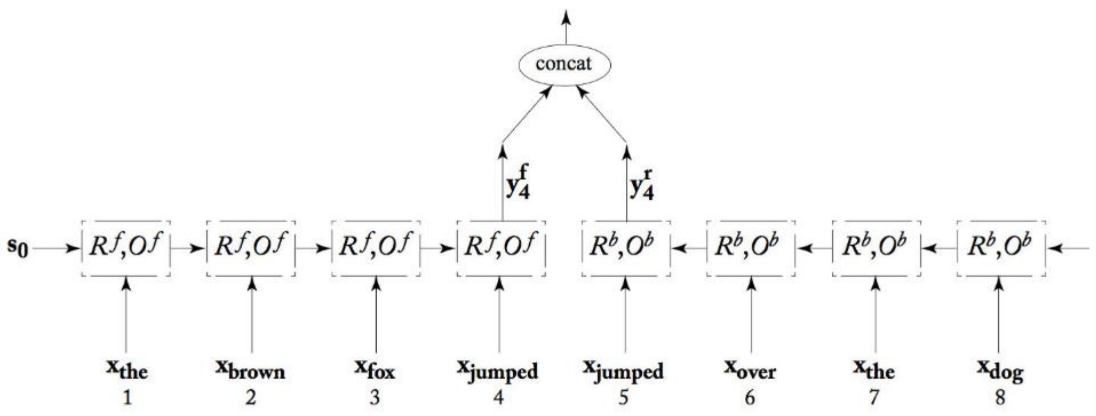
When the RNN only cares about the output from the last timestep: we compute loss from the last unit.

When care about the correctness at each timestep, we sum.



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.



[illegible]

"deep" RNN's, which have more than one processing layer between the input and output streams

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.

Markov Decision Processes and Reinforcement Learning

Mathematical Model:

set of states $s \in S$

set of actions $a \in A$

reward function R(s)

transition function P(s' | s,a)

we make the assumption that rewards are better if they occur sooner.
That is, being near a big reward is almost as good as being at the big reward.

Bellman equation **Optimal Policy**

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

  We don't need a fully converged solution because we'll refine policy in each iteration.

- Asynchronous dynamic programming

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

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

Model based RL:

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 RL (Q-learning)

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

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

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

TD update:

- 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
    - $Q(s,a) = (1-\alpha)Q(s,a) + \alpha[R(s) + \gamma \max_{a'} Q(s',a')]$

Alpha is because there is no obvious endpoint/ world may change

- 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.

Create inconsistency!

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')

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.

TF-IDF: word 2 document

TF: normalized word count

TF = 1+log10(c), c is the count of the word

IDF: inversed document frequency

IDF = log10(1+N/df) N is # of documents, df is # of documents where word occurs

TF-IDF = TF*IDF

PMI: word 2 word

$$I(w,c) = \log_2\left(\frac{P(w,c)}{P(w)P(c)}\right)$$

PPMI:

PPMI = max(0,PMI)

input vectors $v = (v_1, v_2, \ldots, v_n)$ and $w = (w_1, w_2, \ldots, w_n)$

dot product: $v \cdot w = v_1 w_1 + \ldots + v_n w_n$

$\cos(\theta) = \frac{v \cdot w}{|v||w|}$

When building vector space, try to Maximize !!!!dot product!!!!

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

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 $(\mathrm{count}(c))^{\alpha}$. The probabilities used for selecting negative training examples are computed from these smoothed counts.

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)

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

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

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.