Large Scale Deep Learning
One neuron is weak on its own, but when combined with many other neurons, it is very powerful
Since size of neural nets is very important, deep learning requires high performance hardware and software infrastructure

CPUs are slower than GPUs or chains of CPUs
Some modifications of CPU, such as tuning them for fixed-point arithmetic, can make them fast

GPUs are good for deep learning
GPUs, created originally for video games, must be able to do many computations in parallel
GPUs deal with fairly simple computations and have high memory, and do not branch out into complex computations as CPUs do
As a result, CPUs have a higher clock speed, but vastly less parallelization

Since the computation of gradients and matrix multiplication at each step often overburdens the memory of a computer system, a GPU's high memory is advantageous
Neural nets do not involve much branching, and neurons can typically be calculated independently from one another, so they may be parallelized

General purpose GPUs could execute arbitrary code and not just rendering subroutines
Nvidia CUDA provided a way to create this arbitrary code, which soon became adapted in DL

GPUs work very differently from CPUs - for example, on a GPU most writable memory locations are not cached, so it may be faster to compute the same thing twice

- GPUs and CPUs use different methodology for high performance

GPU code is inherently multi-threaded
Coalesced reads and writes occurs when several threads can read or write a value they all need at the cost of one memory transaction

Each thread in a group must execute the same instruction simultaneously, making branching difficult
Threads are divided into small groups called warps - each thread in a warp executes the same instruction, so if different threads in the same warp need to execute different code paths, these paths cannot be traversed in parlalel

Researchers should not need to write new GPU code in order to test new models / algorithms
Models should be specified in terms of already existing libraries
It is also possible to make the same parent code work for different hardware

Data parallelism - distributing inference / training examples on different systems
Model parallelism - multiple systems work together on a single datapoint, each processing a different part

Data parallelism is hard; when we inccrease the minibatch size, we get less than linear returns
Multiple machines cannot compute grad. descent steps in parallel since it's a sequential process

Asynchronous gradient descent - several processor cores share the memory representing the parameters; each reads params without a lock, computes the grad, and updates the grad
Learning is faster overall, but processor cores may overwrite each other's progress
Parameters are managed by a parameter server rather than stored in shared memory
Distributed asynchronous grad. descent is the primary method used by large scale companies

It is more important for a model to be low cost when running inference, since the user is likely to have constrained resources
Model compression replaces the original model with a smaller model requiring less memory and runtime to evaluate

Large models learn some trained function f(x) with limited training data, and generate their own data using a generative model or by corrupting training examples
This data is used to train a less constrained and less expensive model
Alternatively,m you can train the smaller model on the same dataset, but train it to copy features of the other model such as the distribution for incorrectly classified classes

One can accelerate data processing systems by building systems which have a dynamic structure in the graph describing the computation needed to process an output
Neural nets can internally determine which subset of features to compute given info from the input
This is called conditional computation (similar to dropout, but not random)
The simplest version of dynamic structure applied to neural nets is determing which subset of networks should be applied to a particular input

The cascade strategy can be used when detecting the presence of a rare object or event
To know for sure the object is present, we must use a high computational cost classifier
However, since the boject is rare, we can use much less computation to reject inputs not containing the input
In this situation, we trian a series of classifiers

- The first set of classifiers ensures we have high recall, by not wrongly rejecting an input
- The final classifier is trained to have high precision

As soon as one element in the cascade denies it during inference, we abandon the example

It is possible to either make the later layers have high capacity, or to have many small capacity models which have high capacity when combined
Another version of cascades uses the earlier model to simplify the task (e.g by cropping the useful portion)

Decision trees are an example of dynamic structure as they determine which subtrees should be evaluated
It is possible to use a neural net to make further splits for a decision tree classifier

One can use a gater to select which of many several expert networks are used to compute the output; alternatively, the gater can generate the softmaxed probability distribution and then assign weights to each of the networks accordingly
If a single expert is chosen by the gater, we obtain a hard mixture of examples

If we want to select different subsets of parameters or units, it is not possible to use a soft swithch since it requires enumerating and computing the outputs for the gater configs
This is the problem of combinatorial gaters, and different solutions have been proposed to deal with it, such as reinforcement learning to learn a form of conditional dropout on certain blocks

Another kind of dynamic structure is a switch where a hidden unit receives inputs from different units based on the context - this is basically an attention mechanism
Contemporary approaches use a weighted average over all possible inputs, but do not receive all the benefits of a fully dynamic structure

Dynamically structured systems decrease the amount of parallelism resulting from the system following different code branches
Few operations can be described as a matrix multiplication over a minibatch of examples, and instead must be done individually (the benefit of a common operation is not there)
We can write specialized sub-routines, but these are difficult to implement

CPUs will be slow due to the lack of cache coherence and GPUs will be slow due to the lack of coalesced memory transactions (whereas this would be very efficient in a parallizable situation)

This may be mitigated by partitioning the examples into groups which can be parallelized
Partitioning the workload can result in load-balancin issues where the first layer is overloaded and the last layer is underloaded

Hardware designers work specifically to make improvements on neural nets
It is possible to use less than the 32-bit precision (which a CPU has) at inference time
Most recent improvements computing speed have come from parallelization, rather than a single CPU or GPU core
Since the rate of progress of CPUs and GPUs have slowed down, hardware improvements are crucial

Recent work showed that 8-16 bits can suffice for training deep networks with backprop
Some forms of dynamic fixed point representations are better than fixed point representation
Traditional fixed point rep. is restricted to a fixed range, while dynamic shares that range only among a layer

Using fixed point instead of floating point and using less bits per number reduces the surface area and hardware requirements

Computer Vision
Computer vision encompasses many things which pertain to how a computer views / senses things in an image

Computer vision requires relatively little preprocessing - the pixel values must be standardized so that the images are on the same scale, for example in the range [0, 1]
Images do not necessarily need to be the same size before being fed to the convnet - some have variable pooling sizes, and others have outputs which automatically scale w/ the inputs

Many forms of preprocessing attempt to remove variability in the data which is not useful; this lets us use a smaller model which is able to generalize better
When training w/ large datasets and models, it is best to let the models learn the variability on their own

Contrast refers to the magnitude of the difference between dark and bright pixels in an image; in deep learning, it refers to the standard deviation of the pixels in an image or region of image
If X is a 3D tensor with the third dimension representing the channel, the contrast of the entire image is given by

$$\sqrt{\frac{1}{3rc}\sum_{i=1}^{r}\sum_{j=1}^{c}\sum_{k=1}^{3}(X_{i,j,k}-\bar{\mathbf{x}})^2}$$

where X is the mean intensity of the entire image

$$\bar{\mathbf{x}}=\frac{1}{3rc}\sum_{i=1}^{r}\sum_{j=1}^{c}\sum_{k=1}^{3}X_{i,j,k}.$$

Global contrast normalization aims to prevent varying contrast by subtracting the mean from each image and rescaling it such that the std across the pixels equals some constant s
This, however, is impossible for images w/ 0 contrast
Images w/ very low nonzero contrast have little information content, and their noise is only amplified by GCN

This encourages the use of lambda which biases the standard deviation; alternatively, the denominator can be constrained to be only ϵ
Given an input X, GCN produces X' such that

$$X'_{i,j,k}=s\frac{X_{i,j,k}-\bar{X}}{\max\left\{\epsilon,\sqrt{\lambda+\frac{1}{3rc}\sum_{i=1}^{r}\sum_{j=1}^{c}\sum_{k=1}^{3}(X_{i,j,k}-\bar{X})^2}\right\}}$$

Datasets w/ cropped objects are likely to contain any images w/ constant intensity, so it is safe to set lambda = 0 and avoid division by zero by letting ϵ = 1e-8
Small images cropped randomly are more likely to have constant intensity making aggressive regularization useful

The scale parameter s can be set to 1, or chosen to make each individual pixel have an std across examples close to 1 (deviation from the mean across space)

The standard deviation in the GCN equation is a rescaling of the L^2 norm of the image
It is preferable to use GCN since standard deviation includes division by the number of pixels, allowing the same s regardless of image size
GCN can be considered mapping examples to a spherical shell

Many models have a hard time discovering that sometimes weights need to be colinear with different biases; they also have a hard time discovering multiple modes along lines
GCN fixes this by simplifying the problem to just a direction, rather than direction and a distance

A different operation, sphering or whitening, rescales the principal components to have equal variance so the normal distribution used by PCA has spherical contours

Fig 12.1
GCN w/ λ = 0 maps nonzero examples onto a sphere, perfectly
Since GCN was used to normalize the standard deviation, the sphere is not a unit sphere
GCN w/ λ > 0 draws examples to the center of the sphere, but does not completely discard the variation in their norm

GCN encourages light and dark regions to stand apart from one another, but it does not ensure that corners and edges in these regions will stand out
Local contrast normalization ensures that contrast is normalized over each small window, rather than the image as a whole

Various definitions of LCN are possible
All of them involve subtracting the mean and dividing by a standard deviation of nearby pixels; some give Gaussian weights to the nearby pixels based on their distance from the center pixel
Some combine info from different channels while others separate channels

LCN can be achieved by using separable convolutions to compute local standard deviations and mean kernels, and then using element-wise subtraction and division on different feature maps
LCN is differentiable, can be used as a nonlinearity after hidden layers, and can be used to preprocess data

We need to regularize LCN, especially because small windows are likely to contain values that are nearly the same as each other and therefore to have zero standard deviation

Speech Recognition

Speech recognition translates a signal containing spoken language into the words intended by the speaker

Let X = (x^1 … x^t) be a sequence of input vectors of acoustics obtained by splitting the audio every 20 ms

Let y = (y^1 … y^n) represent the resulting words intended by the speaker

Automatic speech recognition task creates a function f^*_{asr} computing the most probable linguistic sequence given the sequence X

$$f^*_{\mathrm{ASR}}(\boldsymbol{X}) = \arg\max_{\boldsymbol{y}} P^*(\mathbf{y} \mid \mathbf{X} = \boldsymbol{X})$$

Solve for y such that P(y) maximized

From the 1980s, GMMs and HMMs were used together; GMMs modeled the association between acoustic features and phoenemes, while HMMs modeled the sequence of phonemes

First, an HMM generates a sequence of phonemes and discrete sub-phonetic states such as beginning, middle, and end

GMM transforms the states, represented by symbols, into audio waveform

For ASRs, HMM-GMM and neural networks had always been about the same in the past, despite the HMM-GMM being more popular

Most neural net related research was based on strengthening the HMM-GMM foundation

In 2009, deep learning based on supervised learning was used for speech recognition

This was based on training unrestricted probabilistic models called restricted Boltzmann machines (RBM) to model the input data

Unsupervised pretraining was used to build feedforward networks whose layers were initialized by an RBM

These networks take spectral acoustic representations in a fixed-size windows around a center frame, and predict the conditional probabilities of HMM states for the center frame

This was followed up by speech recognition which involves not just phenomes but a large vocabulary of words

Deep networks shifted from being based on HMMs and pretraining to dropout and ReLUs

As larger datasets were used and better models were made, pretraining was unnecessary

One of the innovations contributing to this was the use of convnets which replicate weights across frequency and time, where in a 2D input, one dimension corresponds to time and the other to frequency

Previously, time delay networks were used, but they could not replicate over frequency

End-to-end deep learning systems attempt to completely remove the HMM; the first breakthrough was in 2013 with a LSTM RNN

The unfolded graph has 2 kinds of depth: ordinary depth due to a stack of layers, and depth due to unfolding the graph over time

Natural Language Processing
NLP applications are based on language models which define a probability distribution over sequences of words, characters, or bytes in a natural language
Since the number of words is so large, word-based language models must operate on very high dimensional sparse discrete space

Tokens are discrete entities such as characters, words, or bytes
An n-gram is a fixed length sequence containing n tokens

Models based on n-grams define the conditional probability of the n-th token given the preceding n - 1 tokens
Products of conditional distributions define the probability over longer sequences

$$P(x_1, \ldots, x_\tau) = P(x_1, \ldots, x_{n-1}) \prod_{t=n}^{\tau} P(x_t \mid x_{t-n+1}, \ldots, x_{t-1}).$$

This decomposition is based on the chain rule of probability

Max likelihood estimate can be computed simply by counting how many times each possible n-gram appears in the training set

Usually, we train both an n-gram model and an n - 1 gram model at the same time, making it easy to compute the following by looking up stored probabilities:

$$P(x_t \mid x_{t-n+1}, \ldots, x_{t-1}) = \frac{P_n(x_{t-n+1}, \ldots, x_t)}{P_{n-1}(x_{t-n+1}, \ldots, x_{t-1})}$$

Omit the final character for each sequence when training P_{n-1} to produce inference in P_n

The first words in "The Dog Ran Away" cannot be handled by the default formula using conditional probability since there is no context at the beginning of the sentence
We use the marginal probability (probability irrespective of outcomes of other variables) over the words at the start of the sentence

$$P_3(\texttt{THE DOG RAN}).$$

The last word may be predicted in the typical case using conditional distribution using the last shown formula

$$P(\texttt{THE DOG RAN AWAY}) = P_3(\texttt{THE DOG RAN}) P_3(\texttt{DOG RAN AWAY}) / P_2(\texttt{DOG RAN}).$$

P_n is likely to be 0 in many cases even though $(x_{t-n+1}...x_t)$ may appear in the test set
When P_{n-1} is 0, the ratio is undefined so the model does not even produce a sensible input
When P_n is 0 but P_{n-1} is nonzero, the test log-likelihood is negative infinity

To avoid catastrophic outcomes, n-gram models use some form of smoothing
Smoothing techniques shift probability distributions from observed tuples to unobserved ones which are similar

One technique is adding nonzero probability mass to all possible next token values
Another idea is to have a mixture model containing high-order and low-order n-grams with higher order providing more capacity and lower order being more likely to avoid counts of 0

- If you have a unigram model, then it is more likely a sequence of 1 word appears in the training set

Back-off methods look up lower order n-grams if the frequency of the context of the higher order n-grams given by x_{t-n+1} is too small for the higher order model
Formally, it keeps incrementing k until $x_{t-k+1} \dots x_{t-1}$ has a frequency providing a good estimate

There are V^n possible n-grams and |V| is very large
Most n-grams will not occur in the training set
n-grams are similar to k-nearest neighbors in the sense they are non-parametric local predictors

The problem for a language model is that any two different words have the same distance from one another in one-hot vector space so it is difficult to leverage info from "neighbors", or similar words
Only training examples which repeat the same context are useful for local generalization
To overcome this, a language model must share knowledge between semantically similar words

Class-based language models cluster similar words into the same categories, based on things like how frequently they co-occur - statistical strength is shared between words in the same category
This model uses word class IDs rather than individual word IDs
However, much representational power is lost using this method

Neural language models is a type of language model which overcomes dimensionality issues by using a distributed representation of words
NLMs are able to recognize two words are similar while keeping them distinct

Distributed representations allow the model to treat different words which produce common features similarly
For example, a cat and a dog both are animals, so the prediction for a cat informs / contributes to a prediction for a dog

Each training sentence can transfer information to an exponentially large # of semantically related sentences
Curse of dimensionality requires the model to generalize to a # of sentences which is exponential in the sentence length; the model counters this by relating each training sentence to a exponential # of similar sentences, regardless of length

These representations are called word embeddings
Symbols are high dimensional one-hot vectors in the N dimensional space where N is the vocabulary size, which are each at a distance of sqrt(2) from one another
When they are projected onto a lower dimensional space, similar words are grouped together based on the contexts they appeared, so not all w ords are equidistant

Convolutional embeddings from an image to a fixed vector are similar to NLP in that an original input is modified into a vector which can be processed, which is very different from the original input
Distributed representations are not only good for NLP but for dealing with latent variables in graphical models

Fig 12.3
A 2D visualization of how similar words are grouped together
In multiple dimensions, thesse groupings are more strong since multiple features, each representing different avenues of similarity, can be captured

Outputting words may be difficult if there is a large vocabulary size
The hidden layer generates the outputs which are then softmaxed
The output vocabulary size is |V| meaning the weight matrix is very large and high dimensional (since it produces a high dimensional output)
This brings up memory and computation cost
Matrix multiplication must be applied at train and test time since softmax is normalized across the entire vocabulary
The output layer has a high cost due to the gradient and max likelihood estimation at train time, and generating the probability distribution over all the words at test time

Suppose h produces the outputs y-hat given weights and biases
The equations used are

$$a_i = b_i + \sum_j W_{ij} h_j \quad \forall i \in \{1, \ldots, |\mathbb{V}|\},$$

$$\hat{y}_i = \frac{e^{a_i}}{\sum_{i'=1}^{|\mathbb{V}|} e^{a_{i'}}}.$$

Each column in the row corresponding to the particular word is multiplied by each vector element in h representing the embedding features and added to the bias and then softmax normalized based on the embeddings of all the other words

We can reduce V into a shortlist L based on the most frequent words which the neural network processes
The tail T = V/L is an n-gram of rarely occurring words

We want to predict whether i given the context C is a member of the tail so we can combine the two predictions
We can add an extra softmax output which computes P(i ϵ T | C)
The extra output can then be used to obtain a probability distribution over all words in V (not just in the tail or shortlist) based on the following formula:

$$P(y = i \mid C) = 1_{i \in \mathbb{L}} P(y = i \mid C, i \in \mathbb{L})(1 - P(i \in \mathbb{T} \mid C)) + 1_{i \in \mathbb{T}} P(y = i \mid C, i \in \mathbb{T}) P(i \in \mathbb{T} \mid C)$$

Notice the use of complements in the formula for predicting P(y = i | C)
P(y = i | C, i ϵ L) is provided by the NLM while P(y = i | C, i ϵ T) is provided by the n-gram

A disadvantage of this technique is that the NLM is restricted to the most frequent words where it is the least useful

Alternatively, by decomposing the probabilities hierachically, we can decrease the # of necessary computations from V to log |V|

This can be thought of as decomposing the words into many categories each with their own sub-categories
The words are the leaves of the tree, and their probability is obtained by multiplying the probabilites at each node
Multiple paths to the same word represent the word having multiple meanings; we must sum over its individual path probabilities
At each node, the probability is computed with a logistic regression model
Each logistic regression model uses the same context C regardless of depth

Since we allow the log likelihood to be computed more easily, the gradients with respect to the weights and the hidden activations can also be computed easily
It is possible to make computation more efficient by reshaping the tree structure by associating the bit numbers with the log of the frequency of the word, but this is practically not worth it

For example, consider you have l fully connected layers each with width n_h
n_b is the weighted average of bits required to identify a word with the weighting given by the word frequency
The # of operations to compute the hidden activations increase by $O(ln_h^2)$ while the # of ops to compute the outputs increase by $O(n_h n_b)$
If $n_b <= ln_h$ which it usually is, then decreasing n_h will give better computational gains than n_b
n_b is usually small enough since it is the log of the # of words, while n_h is on orders of 1000

We can make the depth two, and increase the branching operations to sqrt(V)

The hierarchical structure can be learned or re-used

Ideally, it is jointly learned with the neural language model
However, it is hard to optimize the tree structure since it is discrete which does not work well with gradient based optimization
Discrete optimization techniques can give an approximate solution
The tree structure is not good for selecting the most likely word in a given context, however

We can speed up learning by avoiding explicitly computing the contribution from the gradient of words which do not appear in the next position
Every incorrect word should have a low probability
We can sample a subset of words

$$\begin{aligned}\frac{\partial \log P(y \mid C)}{\partial \theta} &= \frac{\partial \log \text{softmax}_y(\boldsymbol{a})}{\partial \theta} \\ &= \frac{\partial}{\partial \theta} \log \frac{e^{a_y}}{\sum_i e^{a_i}} \\ &= \frac{\partial}{\partial \theta}(a_y - \log \sum_i e^{a_i}) \\ &= \frac{\partial a_y}{\partial \theta} - \sum_i P(y = i \mid C) \frac{\partial a_i}{\partial \theta}\end{aligned}$$

a is before softmax activations
The first term is called the positive phase, and the second term is the negative phase where the gradient of the activation is weighted down by P(y = i | C), the probability of the word appearing
However, we want to avoid computing P(i | C) for all i in the vocabulary, since it is expensive to estimate with our model

Instead of sampling from the model, we can sample from another distribution called the proposal distribution q
We need to modify the weights to correct for the bias induced by sampling from the wrong distribution
This is known as importance sampling
Exact importance sampling is not very efficient since it requires computing weights p_i / q_i where $p_i = P(i \mid C)$ which requires that all a_i scores are computed

Biases importance sampling ensures that the weights sum to 1
For every negative word n_i the associated gradient is weighted by:

$$w_i = \frac{p_{n_i}/q_{n_i}}{\sum_{j=1}^{N} p_{n_j}/q_{n_j}}.$$

These weights are used to give the appropriate contribution to the gradient of the m negative samples from q

$$\sum_{i=1}^{|\mathbb{V}|} P(i \mid C) \frac{\partial a_i}{\partial \theta} \approx \frac{1}{m} \sum_{i=1}^{m} w_i \frac{\partial a_{n_i}}{\partial \theta}.$$ This uses alternatiev distribution with a slightly higher error for computational efficiency

Since it is easy to estimate the parameters θ for a unigram or bigram, they work well as a proposal distribution q
Importance sampling is good for models which have sparse outputs - for example, a bag of words with v_i which indicates whether or not word exists
Loss functions may not work well with sparse outputs
Additionally, the model might initially output vectors which are not truly sparse (mostly nonzero) and when they are compared to 0s, this becomes computationally inefficient

The computational complexity for the gradient is reduced to be proportional to the number of negative samples (only sampled) rather than proportional to the size of the output vector

Noise-contrastive estimation tries to rank the correct word's score a_y higher than the other words' scores a_i
The ranking loss is given by:

$$L = \sum_i \max(0, 1 - a_y + a_i).$$

If $a_y > a_i$ by a margin of 1, then the gradient is 0 (indicating the weight are good enough)
However, this does not make use of context which may be useful in many cases

An n-gram stores tuples of frequencies which allow it to easily retrieve / process an example
Neural networks on the other hand, require increased computation time for more parameters
There are some exceptions:

- An embedding layer only use a single embedding each pass, so it is unaffected by higher vocabulary sizes
- Tiled convolutions can reduce parameter sharing while increasing # of kernels to keep the computation the same

An NLM and an n-gram can be combined in an ensemble and perform well since they are likely to make independent mistakes
Neural networks can be combined with max entropy models which can be thought of as having an extra input which is directly connected to the output; this extra input highlights particular n-grams and is sparse
The # of params increases to $|sV|^n$ but the extra inputs are very sparse so this isn't too bad

Machine translation can be used to translate from one language to another
One component proposes translations, while the other chooses the best grammatical one
Recently, n-gram language models are being upgraded to NLMs

The n-gram language models used max entropy language models where the softmax layer predicted the next words given frequent n-grams in the context

It is worth using context in machine translation; conditional MLPs have been beating n-grams lately

The MLP based approach requires the inputs to be preprocessed to be fixed size
To accomodate variable length inputs, we can use an RNN
The inputs are usually processed by a model to generate the context C which is then given to an RNN to output the translated task

The models must have a way to represent source sentences
Earlier models could only represent words or individual phrases
It might be beneficial to give similar representations to sentences with similar meanings; this was first explored using a combination of convolutions and RNNs
Now, we have an RNN to propose translations and another to score proposals

Using a fixed size representation might be restrictive to the model's ability to learn
An RNN will probably learn if it has the capacity and enough data
It is sometimes better, however, to read the whole sentence to get the context / gist of it, and then to generate the translated words one at a time, focusing on individual parts of the input sentence to get the details needed to produce the next output

Fig 12.6
An attention mechanism in this case is a weighted average
The feature vectors h^t receive weights $\boldsymbol{\alpha}^t$ between 0 and 1, which are generated by the model
$\boldsymbol{\alpha}^t$ is intended to concentrate around one h^t
This provides an easily differentiable approximation

An attention based system first reads raw data such as words in a sentence and converts them into distributed representations with each feature vector corresponding to a word position
These feature vectors can be thought of as a "memory"
Another component of the system exploits this memory (potentially re-arranging it) to generate the output, and giving certain parts more weight

If two words in different languages are aligned, then having similar word embeddings is effective

Other Applications
Regression in recommender systems is used to predict the expected gain of providing a recommendation to a user
Classification is used to predict the probability of the user doing something given that recommendation

The earliest recommender systems used collaborative filtering which relied on user's having similar tastes - if two people are similar based on previous items bought, and user 1 likes item D, then item D is also recommended to user 2

Parametric representations rely on learning a distributed representation such as an embedding for each user and item

The prediction is obtained by a dot product between the user embedding and item embedding
Let R be the prediction matrix, A be the user embedding , B be the item embedding, b and c are the user and item bias, respectively
The prediction is given by:

$$\hat{R}_{u,i} = b_u + c_i + \sum_j A_{u,j} B_{j,i}.$$

Ideally, we would like to minimize R - R-hat

In order to obtain the embeddings, we can use SVD to decompose R into UDV, with A corresponding to UD and B corresponding to V
The issue with this is that SVD treats missing entries arbitrarily as if they corresponded to a target value of 0, but we do not want to pay any cost for missing entries

Fortunately, minimizing the sum of squares on just the observed ratings can be done through gradient based optimization

Other collaborative filtering approaches such as an RBM and more complicated factoring methods have been exlpored
The problem of cold start recommendations: when a user who hasn't rated anything before joins, it is hard to tell what to recommend to them
A solution for this is content-based recommendations, where other features such as their age given on their user profile is used to generate the embeddings

Contextual bandits refers to the fact that we only take one action in order to obtain a certain recommendation - we do not know the result of obtaining another recommendation, so our result is biased
Furthermore, the ad may not get shown at all due to failing to meet a minimum threshold, so we do not know cases which users might be more interested in
This requires us to have more examples, since in a typical supervised problem, the model knows what the true label y is

The system may keep on picking the wrong class because the model gives the correct class a very low probability
Only one positive reward may be observed, and it causes a feedback loop

The mapping from context to action is called a policy

The main problem, which is what reinforcement learning attempts to solve, is exploration vs exploitation
We want to make sure we are getting the full picture by seeing what all recommendations do, but uat the same time, we want to make use of what we know to be already good
If we take some action a which gives a reward of 1, we are not sure whether it is the best
We may wis to try an action a' to be sure => we might get 2, but we risk getting 0 as well
There are multiple approaches to exploration such as using a model's predicted expected value and uncertainty to choose an action, or to choose an action randomly

If we have a smaller time scale, then we want to explore quickly and focus more on exploitation
If we have a longer time scale, then we have more time to explore and use that exploration to make more informed decision
The more info we have, the more we can exploit the model
In supervised learning, there is no tradeoff between exploration vs exploitation because we know which label is the best choice

It is hard to decide which policy to use since it is hard to evaluate the performance on the test set when the reward is being used as part of the feedback loop to determine which inputs will be seen

A research frontier is to develop embeddings for phrases and for relationships between words and facts

Distributed representations can be used to capture facts about relationships between objects
A binary relation is a set of ordered pairs of objects; pairs in the set have the relation, while pairs not in the set do not have the relation
For example, if S contains all pairs where the first object is less than the second object, the ordered pair (2, 1) is not in the set S, and does not have the relation

In the context of AI, we can think of a relation in sentences as playing the role of the verb
The sentence is structured as (subject, verb, object) corresponding to entity_i relation_j entity_k
Alternatively, this can be just represented as entity_i attribute_j where the attribute is only one argument, such as has_fur (notice the object is part of the attribute)

We can infer relations between entities from training data consisting of unstructured natural language
There are structured databases which give explicit connections - e.g relational database
A knowledge base is a database intended to convey commonsense knowledge of everyday life
Representations for entities and relations can be learned by considering each triplet (any two entities and their relations) in a knowledge base as a training example and maximizing a training objective which captures their joint distribution

A common approach is to extend neural language models to maximize such an objective
Neural language models give distributed representations for each word and capture how likely a word is to occur after another word
In fact, the two tasks are so closely related, their databases (natural language and knowledge bases) are combined and used for the same task

Early work on learning relations between entities treated entities differently from relations; a relation was a matrix mapping between two different entities, acting as an operator
Now, it is possible to make a relation like an entity which requires more flexible machinery

Link prediction attempts to predict new facts based on old facts
Many true relations are absent from knowledge since facts are hard coded when creating a database
Evaluating performance on this is difficult because we only have a dataset of positive examples
Therefore, the metrics area bsed on the model predicting more likely correct facts

A good way to generate negative data is to take a true fact and create corrupted versions of the facts by replacing one of the entities with another
The precision at 10% metric counts how many times the model predicts the top 10% of corrupted facts as correct

Word sense disambiguation attempts to choose the correct which version of the word is the correct one given context

Knowledge of relations with a reasoning process and natural language allows us to build a general question answering system
This requires some form of memory however
An explicit memory mechanism, as was used with RNNs can be used to retrieve the specific declarative facts needed
One way is to extend the GRU; read the input into the memory and produce the answer given the contents of the memory