**Chapter 12**

**APPLICATIONS**

* 1. **Large Scale Deep Learning**

**1.1.1) Fast CPU Implementations**

* Traditionally, neural networks were trained using the CPU of a single machine.
* We now mostly use GPU computing or the CPUs of many machines networked together.
* For example, in 2011, the best CPUs available could run neural network workloads faster when using fixed-point arithmetic rather than floating-point arithmetic. By creating a carefully tuned fixed-point implementation, Vanhoucke et al. (2011) obtained a 3× speedup over a strong floating-point system.
* The important principle is that careful specialization of numerical computation routines can yield a large payoff. Other strategies, besides choosing whether to use fixed or floating point, include optimizing data structures to avoid cache misses and using vector instructions. Many machine learning researchers neglect these implementation details, but when the performance of an implementation restricts the size of the model, the accuracy of the model suffers.

**1.1.2) GPU Implementations**

* Graphics cards must perform matrix multiplication and division on many vertices in parallel to convert these 3-D coordinates into 2-D on-screen coordinates. The graphics card must then perform many computations at each pixel in parallel to determine the color of each pixel. In both cases, the computations are fairly simple and do not involve much branching compared to the computational workload that a CPU usually encounters. For example, each vertex in the same rigid object will be multiplied by the same matrix; there is no need to evaluate an if statement per-vertex to determine which matrix to multiply by. The computations are also entirely independent of each other, and thus may be parallelized easily.
* Over time, GPU hardware became more flexible, allowing custom subroutines to be used to transform the coordinates of vertices or assign colors to pixels. In principle, there was no requirement that these pixel values actually be based on a rendering task. These GPUs could be used for scientific computing by writing the output of a computation to a buffer of pixel values. Steinkrau *et al.* (2005) implemented a two-layer fully connected neural network on a GPU and reported a 3X speedup over their CPU-based baseline.
* The popularity of graphics cards for neural network training exploded after the advent of *general purpose GPUs*. These GP-GPUs could execute arbitrary code, not just rendering subroutines. NVIDIA’s CUDA programming language provided a way to write this arbitrary code in a C-like language. With their relatively convenient programming model, massive parallelism, and high memory bandwidth, GP-GPUs now offer an ideal platform for neural network programming.
* The techniques required to obtain good performance on GPU are very different from those used on CPU. For example, good CPU-based code is usually designed to read information from the cache as much as possible. On GPU, most writable memory locations are not cached, so it can actually be faster to compute the same value twice, rather than compute it once and read it back from memory.
* GPU code is also inherently multi-threaded and the different threads must be coordinated with each other carefully. For example, memory operations are faster if they can be *coalesced*. Coalesced reads or writes occur when several threads can each read or write a value that they need simultaneously, as part of a single memory transaction.
* Another common consideration for GPUs is making sure that each thread in a group executes the same instruction simultaneously. This means that branching can be difficult on GPU. Threads are divided into small groups called *warps*. Each thread in a warp executes the same instruction during each cycle, so if different threads within the same warp need to execute different code paths, these different code paths must be traversed sequentially rather than in parallel.

**1.1.3) Large Scale Distributed Implementations**

* Distributing inference is simple, because each input example we want to process can be run by a separate machine. This is known as **data parallelism**. It is also possible to get *model parallelism*, where multiple machines work together on a single datapoint, with each machine running a different part of the model. This is feasible for both inference and training.
* It would be better to allow multiple machines to compute multiple gradient descent steps in parallel. Unfortunately, the standard definition of gradient descent is as a completely sequential algorithm: the gradient at step is a function of the parameters t produced by step t− 1.
* This can be solved using **asynchronous stochastic gradient descent**(Bengio *et al.*, 2001; Recht *et al.*, 2011). In this approach, several processor cores share the memory representing the parameters. Each core reads parameters without a lock, then computes a gradient, then increments the parameters without a lock. This reduces the average amount of improvement that each gradient descent step yields, because some of the cores overwrite each other’s progress, but the increased rate of production of steps causes the learning process to be faster overall. Dean *et al.* (2012) pioneered the multi-machine implementation of this lock-free approach to gradient descent, where the parameters are managed by a **parameter server** rather than stored in shared memory. Distributed asynchronous gradient descent remains the primary strategy for training large deep networks and is used by most major deep learning groups in industry

**1.1.4) Model Compression**

* In many commercial applications, it is much more important that the time and memory cost of running inference in a machine learning model be low than that the time and memory cost of training be low. It is possible to train a model once, then deploy it to be used by billions of users.
* A key strategy for reducing the cost of inference is *model compression* (Buciluˇa *et al.*, ). The basic idea of model compression 2006 is to replace the original, expensive model with a smaller model that requires less memory and runtime to store and evaluate.
* Model compression is applicable when the size of the original model is driven primarily by a need to prevent overfitting. In most cases, the model with the lowest generalization error is an ensemble of several independently trained models. Evaluating all n ensemble members is expensive. Sometimes, even a single model generalizes better if it is large (for example, if it is regularized with dropout).
* These large models learn some function f(x), but do so using many more parameters than are necessary for the task. Their size is necessary only due to the limited number of training examples. As soon as we have fit this function f(x), we can generate a training set containing infinitely many examples, simply by applying f to randomly sampled points x. We then train the new, smaller, model to match f (x) on these points. In order to most efficiently use the capacity of the new, small model, it is best to sample the new x points from a distribution resembling the actual test inputs that will be supplied to the model later. This can be done by corrupting training examples or by drawing points from a generative model trained on the original training set.
* Alternatively, one can train the smaller model only on the original training points, but train it to copy other features of the model, such as its posterior distribution over the incorrect classes (Hinton *et al.*, 2014, 2015).

**1.1.5) Dynamic Structure**

* Individual neural networks can exhibit dynamic structure internally by determining which subset of features (hidden units) to compute given information from the input. This form of dynamic structure inside neural networks is sometimes called **conditional computation**(Bengio, 2013; Bengio *et al.*, 2013b).
* Since many components of the architecture may be relevant only for a small amount of possible inputs, the system can run faster by computing these features only when they are needed.
* A venerable strategy for accelerating inference in a classifier is to use a **cascade of classifiers**. The cascade strategy may be applied when the goal is to detect the presence of a rare object (or event). To know for sure that the object is present, we must use a sophisticated classifier with high capacity, that is expensive to run. However, because the object is rare, we can usually use much less computation to reject inputs as not containing the object. In these situations, we can train a sequence of classifiers.
* The first classifiers in the sequence have low capacity, and are trained to have high recall. In other words, they are trained to make sure we do not wrongly reject an input when the object is present. The final classifier is trained to have high precision. At test time, we run inference by running the classifiers in a sequence, abandoning any example as soon as any one element in the cascade rejects it.
* Overall, this allows us to verify the presence of objects with high confidence, using a high capacity model, but does not force us to pay the cost of full inference for every example.
* There are two different ways that the cascade can achieve high capacity.

1. One way is to make the later members of the cascade individually have high capacity. In this case, the system as a whole obviously has high capacity, because some of its individual members do.
2. It is also possible to make a cascade in which every individual model has low capacity but the system as a whole has high capacity due to the combination of many small models.

* Viola and Jones (2001) used a cascade of boosted decision trees to implement a fast and robust face detector suitable for use in handheld digital cameras. Their classifier localizes a face using essentially a sliding window approach in which many windows are examined and rejected if they do not contain faces.
* Decision trees themselves are an example of dynamic structure, because each node in the tree determines which of its subtrees should be evaluated for each input.
* In the same spirit, one can use a neural network, called the **gater**to select which one out of several **expert networks**will be used to compute the output, given the current input.
* The first version of this idea is called the **mixture of experts**(Nowlan, 1990; Jacobs *et al.*, 1991), in which the gater outputs a set of probabilities or weights (obtained via a softmax nonlinearity), one per expert, and the final output is obtained by the weighted combination of the output of the experts.
* Another kind of dynamic structure is a **switch**, where a hidden unit can receive input from different units depending on the context. This dynamic routing approach can be interpreted as an **attention mechanism** (Olshausen *et al.*, 1993).
* So far, the use of a hard switch has not proven effective on large-scale applications. Contemporary approaches instead use a weighted average over many possible inputs, and thus do not achieve all the possible computational benefits of dynamic structure.
* One major obstacle to using dynamically structured systems is the decreased degree of parallelism.
* We can write more specialized sub-routines that convolve each example with different kernels or multiply each row of a design matrix by a different set of columns of weights. Unfortunately, these more specialized subroutines are difficult to implement efficiently.
* CPU implementations will be slow due to the lack of cache coherence and GPU implementations will be slow due to the lack of coalesced memory transactions and the need to serialize warps when members of a warp take different branches.

**1.1.6) Specialized Hardware Implementations of Deep Networks**

* Though software implementations on general-purpose processing units (CPUs and GPUs) typically use 32 or 64 bits of precision to represent floating point numbers, it has long been known that it was possible to use less precision, at least at inference time.
* Another factor that motivates current research on specialized hardware for deep networks is that the rate of progress of a single CPU or GPU core has slowed down, and most recent improvements in computing speed have come from parallelization across cores (either in CPUs or GPUs).
* Recent work on low-precision implementations of backprop-based neural nets (Vanhoucke 2011 Courbariaux *et al.*, 2011; Gupta *et al.*, 2015) suggests that between 8 and 16 bits of precision can suffice for using or training deep neural networks with back-propagation. What is clear is that more precision is required during training than at inference time, and that some forms of dynamic fixed-point representation of numbers can be used to reduce how many bits are required per number.
  1. **Computer Vision**

Applications of computer vision range from reproducing human visual abilities, such as recognizing faces, to creating entirely new categories of visual abilities. As an example of the latter category, one recent computer vision application is to recognize sound waves from the vibrations they induce in objects visible in a video (Davis *et al.*,2014).

Most deep learning for computer vision is used for object recognition or detection of some form, whether this means reporting which object is present in an image, annotating an image with bounding boxes around each object, transcribing a sequence of symbols from an image, or labeling each pixel in an image with the identity of the object it belongs to.

**1.2.1) Preprocessing**

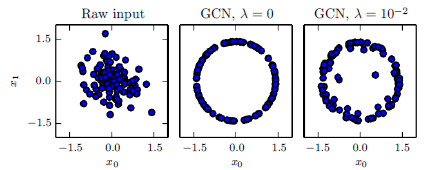
* Computer vision usually requires relatively little of this kind of preprocessing.
* The images should be standardized so that their pixels all lie in the same, reasonable range, like [0,1] or [-1, 1]. Mixing images that lie in [0,1] with images that lie in [0, 255] will usually result in failure.
* Many computer vision architectures require images of a standard size, so images must be cropped or scaled to fit that size. However, even this rescaling is not always strictly necessary.
* Some convolutional models accept variably-sized inputs and dynamically adjust the size of their pooling regions to keep the output size constant (Waibel *et al.*, 1989). Other convolutional models have variable-sized output that automatically scales in size with the input, such as models that denoise or label each pixel in an image (Hadsell *et al.*, 2007).
* Dataset augmentation is an excellent way to reduce the generalization error of most computer vision models. A related idea applicable at test time is to show the model many different versions of the same input (for example, the same image cropped at slightly different locations) and have the different instantiations of the model vote to determine the output. This latter idea can be interpreted as an ensemble approach, and helps to reduce generalization error.
* Other kinds of preprocessing are applied to both the train and the test set with the goal of putting each example into a more canonical form in order to reduce the amount of variation that the model needs to account for. Reducing the amount of variation in the data can both reduce generalization error and reduce the size of the model needed to fit the training set. Simpler tasks may be solved by smaller models, and simpler solutions are more likely to generalize well.
* When training with large datasets and large models, this kind of preprocessing is often unnecessary, and it is best to just let the model learn which kinds of variability it should become invariant to. For
* example, the AlexNet system for classifying ImageNet only has one preprocessing step: subtracting the mean across training examples of each pixel (Krizhevsky *et al.*, 2012).

**1.2.1.1) Contrast Normalization**

* One of the most obvious sources of variation that can be safely removed for many tasks is the amount of contrast in the image. Contrast simply refers to the magnitude of the difference between the bright and the dark pixels in an image. In the context of deep learning, contrast usually refers to the standard deviation of the pixels in an image or region of an image.
* Suppose we have an image represented by a tensor **X**∈Rr×c×3, with Xi,j,1 being the red intensity at row i and column j, Xi,j,2 giving the green intensity and Xi,j,3 giving the blue intensity. Then the contrast of the entire image is given by

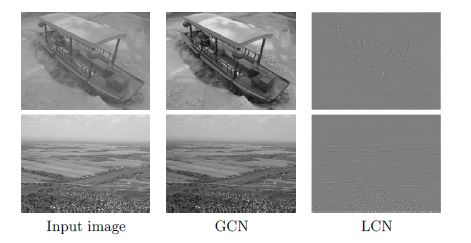
where is the mean intensity of the entire image:

* **Global contrast normalization (GCN)** aims to prevent images from having varying amounts of contrast by subtracting the mean from each image, then rescaling it so that the standard deviation across its pixels is equal to some constant s.
* Datasets consisting of large images cropped to interesting objects are unlikely to contain any images with nearly constant intensity. In these cases, it is safe to practically ignore the small denominator problem by setting λ = 0 and avoid division by 0 in extremely rare cases by setting to an extremely low value like 10−8.
* The scale parameter s can usually be set to 1 or chosen to make each individual pixel have standard deviation across examples close to 1.
* One can understand GCN as mapping examples to a spherical shell. See Fig. below for an illustration. This can be a useful property because neural networks are often better at responding to directions in space rather than exact locations. Responding to multiple distances in the same direction requires hidden units with collinear weight vectors but different biases. Such coordination can be difficult for the learning algorithm to discover. Additionally, many shallow graphical models have problems with representing multiple separated modes along the same line. GCN avoids these problems by reducing each example to a direction rather than a direction and a distance.



**Figure 1:** *GCN maps examples onto a sphere. (Left) Raw input data may have any norm. (Center) GCN with λ = 0 maps all non-zero examples perfectly onto a sphere. Here we use s = 1 and = 10−8 . Because we use GCN based on normalizing the standard deviation rather than the L2 norm, the resulting sphere is not the unit sphere. (Right) Regularized GCN, with λ > 0, draws examples toward the sphere but does not completely discard the variation in their norm. We leave s and the same as before.*

* Counterintuitively, there is a preprocessing operation known as **sphering**and it is not the same operation as GCN. Sphering does not refer to making the data lie on a spherical shell, but rather to rescaling the principal components to have equal variance, so that the multivariate normal distribution used by PCA has spherical contours. Sphering is more commonly known as **whitening**.
* Global contrast normalization will often fail to highlight image features we would like to stand out, such as edges and corners. If we have a scene with a large dark area and a large bright area (such as a city square with half the image in the shadow of a building) then global contrast normalization will ensure there is a large difference between the brightness of the dark area and the brightness of the light area. It will not, however, ensure that edges within the dark region stand out.
* This motivates **local contrast normalization**. Local contrast normalization ensures that the contrast is normalized across each small window, rather than over the image as a whole.
* Various definitions of local contrast normalization are possible. In all cases, one modifies each pixel by subtracting a mean of nearby pixels and dividing by a standard deviation of nearby pixels. In some cases, this is literally the mean and standard deviation of all pixels in a rectangular window centered on the pixel to be modified (Pinto et al., 2008). In other cases, this is a weighted mean and weighted standard deviation using Gaussian weights centered on the pixel to be modified. In the case of color images, some strategies process different color channels separately while others combine information from different channels to normalize each pixel (Sermanet et al., 2012).



**Figure 2:** *A comparison of global and local contrast normalization. Visually, the effects of global contrast normalization are subtle. It places all images on roughly the same scale, which reduces the burden on the learning algorithm to handle multiple scales. Local contrast normalization modifies the image much more, discarding all regions of constant intensity. This allows the model to focus on just the edges. Regions of fine texture, such as the houses in the second row, may lose some detail due to the bandwidth of the normalization kernel being too high.*

* Local contrast normalization is a differentiable operation and can also be used as a nonlinearity applied to the hidden layers of a network, as well as a preprocessing operation applied to the input.
* As with global contrast normalization, we typically need to regularize local contrast normalization to avoid division by zero. In fact, because local contrast normalization typically acts on smaller windows, it is even more important to regularize. Smaller windows are more likely to contain values that are all nearly the same as each other, and thus more likely to have zero standard deviation.

**1.2.1.2) Dataset Augmentation**

* Object recognition is a classification task that is especially amenable to this form of dataset augmentation.
* Classifiers can benefit from random translations, rotations, and in some cases, flips of the input to augment the dataset.
* In specialized computer vision applications, more advanced transformations are commonly used for dataset augmentation. These schemes include **random perturbation of the colors in an image** ( Krizhevsky *et al.*,2012) and **nonlinear geometric distortions** of the input (LeCun *et al.*, 1998b).
* Dividing by the true standard deviation usually accomplishes nothing more than amplifying sensor noise or compression artifacts in such cases. This motivates introducing a small, positive regularization parameter λ to bias the estimate of the standard deviation. Alternately, one can constrain the denominator to be at least. Given an input image **X**, GCN produces an output image **X’**, defined such that
  1. **Speech Recognition**
* Let X = (x(1), x(2) , . . . , x(T)) denote the sequence of acoustic input vectors (traditionally produced by splitting the audio into 20ms frames). Most speech recognition systems preprocess the input using specialized hand-designed features, but some (Jaitly and Hinton, 2011) deep learning systems learn features from raw input. Let y = (y1 , y2 , . . . , yN ) denote the target output sequence (usually a sequence of words or characters). The **automatic speech recognition**(ASR) task consists of creating a function f∗ASR that computes the most probable linguistic sequence y given the acoustic sequence X:

where P ∗ is the true conditional distribution relating the inputs X to the targets y.

* Since the 1980s and until about 2009–2012, state-of-the art speech recognition systems primarily combined hidden Markov models (HMMs) and Gaussian mixture models (GMMs).
* First an HMM generates a sequence of phonemes and discrete sub-phonemic states (such as the beginning, middle, and end of each phoneme), then a GMM transforms each discrete symbol into a brief segment of audio waveform. Although GMM-HMM systems dominated ASR until recently, speech recognition was actually one of the first areas where neural networks were applied.
* Starting in 2009, speech researchers applied a form of deep learning based on unsupervised learning to speech recognition. This approach to deep learning was based on training undirected probabilistic models called restricted Boltzmann machines (RBMs) to model the input data.
* To solve speech recognition tasks, unsupervised pretraining was used to build deep feedforward networks whose layers were each initialized by training an RBM. These networks take spectral acoustic representations in a fixed-size input window (around a center frame) and predict the conditional probabilities of HMM states for that center frame. Training such deep networks helped to significantly improve the recognition rate on TIMIT (Mohamed *et al.*, 2009, 2012a), bringing down the phoneme error rate from about 26% to 20.7%.
* Extensions to the basic phone recognition pipeline included the addition of speaker-adaptive features (Mohamed *et al.*, 2011) that further reduced the error rate. This was quickly followed up by work to expand the architecture from phoneme recognition (which is what TIMIT is focused on) to large-vocabulary speech recognition (Dahl *et al.*, 2012), which involves not just recognizing phonemes but also recognizing sequences of words from a large vocabulary. Deep networks for speech recognition eventually shifted from being based on pretraining and Boltzmann machines to being based on techniques such as rectified linear units and dropout (Zeiler *et al.*, 2013; Dahl *et al.*, 2013).
* Later, as these groups explored larger and larger labeled datasets and incorporated some of the methods for initializing, training, and setting up the architecture of deep nets, they realized that the unsupervised pretraining phase was either unnecessary or did not bring any significant improvement.
* These breakthroughs in recognition performance for word error rate in speech recognition were unprecedented (around 30% improvement) and were following a long period of about ten years during which error rates did not improve much with the traditional GMM-HMM technology, in spite of the continuously growing size of training sets. This created a rapid shift in the speech recognition community towards deep learning.
* One of these innovations was the use of convolutional networks (Sainath *et al.*, 2013) that replicate weights across time and frequency, improving over the earlier time-delay neural networks that replicated weights only across time. The new two-dimensional convolutional models regard the input spectrogram not as one long vector but as an image, with one axis corresponding to time and the other to frequency of spectral components.
* Another important push, still ongoing, has been towards end-to-end deep learning speech recognition systems that completely remove the HMM. The first major breakthrough in this direction came from Graves *et al.* (2013) who trained a deep LSTM RNN (see Sec. 10.10), using MAP inference over the frame-to-phoneme alignment, as in LeCun *et al.* (1998b) and in the CTC framework (Graves *et al.*, 2006; Graves, 2012). A deep RNN (Graves *et al.*, 2013) has state variables from several layers at each time step, giving the unfolded graph two kinds of depth: ordinary depth due to a stack of layers, and depth due to time unfolding. This work brought the phoneme error rate on TIMIT to a record low of 17.7%.
  1. **Natural Language Processing**

More naturally occurring languages are often ambiguous and defy formal description.

To achieve excellent performance and to scale well to large applications, some domain-specific strategies become important.

In many cases, we choose to regard natural language as a sequence of words, rather than a sequence of individual characters or bytes. Because the total number of possible words is so large, word-based language models must operate on an extremely high-dimensional and sparse discrete space.

**1.4.1) n-grams**

* A languagemodeldefines a probability distribution over sequences of tokens in a natural language. Depending on how the model is designed, a token may be a word, a character, or even a byte.
* An n-gram is a sequence of n tokens
* Models based on n-grams define the conditional probability of the n-th token given the preceding n − 1 tokens. The model uses products of these conditional distributions to define the probability distribution over longer sequences:
* Training n-gram models is straightforward because the maximum likelihood estimate can be computed simply by counting how many times each possible n gram occurs in the training set.
* For small values of n , models have particular names: **unigram**for n=1, **bigram**for n=2, and **trigram**for n=3.
* Usually we train both an n-gram model and an n−1 gram model simultaneously. This makes it easy to compute

simply by looking up two stored probabilities

* As an example, we demonstrate how a trigram model computes the probability of the sentence “THE DOG RAN AWAY.” The first words of the sentence cannot be handled by the default formula based on conditional probability because there is no context at the beginning of the sentence. Instead, we must use the marginal probability over words at the start of the sentence. We thus evaluate P3 (THE DOG RAN). Finally, the last word may be predicted using the typical case, of using the conditional distribution P (AWAY | DOG RAN). Putting this together with above equation, we obtain:

P(THE DOG RAN AWAY) = P3(THE DOG RAN)P3(DOG RAN AWAY)/P2(DOG RAN)

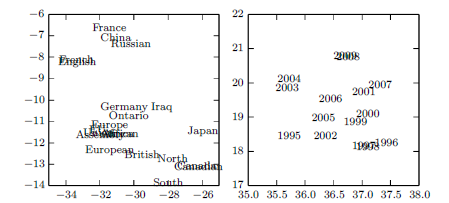
* A fundamental limitation of maximum likelihood for n-gram models is that Pn as estimated from training set counts is very likely to be zero in many cases, even though the tuple

(xt-n+1,...,xt ) may appear in the test set. This can cause two different kinds of catastrophic outcomes. When Pn−1 is zero, the ratio is undefined, so the model does not even produce a sensible output. When Pn−1 is non-zero but Pn is zero, the test log-likelihood is −∞. To avoid such catastrophic outcomes, most n-gram models employ some form of **smoothing**. Smoothing techniques shift probability mass from the observed tuples to unobserved ones that are similar.

* One basic technique consists of adding non-zero probability mass to all of the possible next symbol values.
* Another very popular idea is to form a mixture model containing higher-order and lower-order n-gram models, with the higher-order models providing more capacity and the lower-order models being more likely to avoid counts of zero. *Back-off methods* look-up the lower-order n-grams if the frequency of the context xt−1, . . . , xt−n+1 is too small to use the higher-order model.
* Classical n -gram models are particularly vulnerable to the curse of dimensionality. There are |V|n possible n-grams and |V| is often very large. Even with a massive training set and modest n, most n-grams will not occur in the training set. One way to view a classical n-gram model is that it is performing nearest-neighbor lookup.
* The problem for a language model is even more severe than usual, because any two different words have the same distance from each other in one-hot vector space. It is thus difficult to leverage much information from any “neighbors”—only training examples that repeat literally the same context are useful for local generalization. To overcome these problems, a language model must be able to share knowledge between one word and other semantically similar words.
* To improve the statistical efficiency of n -gram models, class-based language models (Brown et al., 1992; Ney and Kneser, 1993; Niesler et al., 1998) introduce the notion of word categories and then share statistical strength between words that are in the same category. The idea is to use a clustering algorithm to partition the set of words into clusters or classes, based on their co-occurrence frequencies with other words. The model can then use word class IDs rather than individual word IDs to represent the context on the right side of the conditioning bar. Composite models combining word-based and class-based models via mixing or back-off are also possible. Although word classes provide a way to generalize between sequences in which some word is replaced by another of the same class, much information is lost in this representation.

**1.4.2) Neural Language Models**

* Neural language modelsor NLMs are a class of language model designed to overcome the curse of dimensionality problem
* Unlike classbased n -gram models, neural language models are able to recognize that two words are similar without losing the ability to encode each word as distinct from the other. Neural language models share statistical strength between one word (and its context) and other similar words and contexts.
* For example, if the word dog and the word cat map to representations that share many attributes, then sentences that contain the word cat can inform the predictions that will be made by the model for sentences that contain the word dog, and vice-versa.
* There are many ways in which generalization can happen, transferring information from each training sentence to an exponentially large number of semantically related sentences. The model counters this curse by relating each training sentence to an exponential number of similar sentences.
* We sometimes call these word representations **word embeddings**. In this interpretation, we view the raw symbols as points in a space of dimension equal to the vocabulary size. The word representations embed those points in a feature space of lower dimension. In the original space, every word is represented by a one-hot vector, so every pair of words is at Euclidean distance from each other. In the embedding space, words that frequently appear in similar contexts (or any pair of words sharing some “features” learned by the model) are close to each other. This often results in words with similar meanings being neighbors.
* Neural networks in other domains also define embeddings. For example, a hidden layer of a convolutional network provides an “image embedding”.



**Figure 3:** *Two-dimensional visualizations of word embeddings obtained from a neural machine translation model (Bahdanau et al., 2015), zooming in on specific areas where semantically related words have embedding vectors that are close to each other. Countries appear on the left and numbers on the right. Keep in mind that these embeddings are 2-D for the purpose of visualization. In real applications, embeddings typically have higher dimensionality and can simultaneously capture many kinds of similarity between words.*

**1.4.3) High-Dimensional Outputs**

* In many natural language applications, we often want our models to produce words (rather than characters) as the fundamental unit of the output. For large vocabularies, it can be very computationally expensive.
* The naive approach to representing such a distribution is to apply an affine transformation from a hidden representation to the output space, then apply the softmax function. Suppose we have a vocabulary **V** with size | **V** |. The weight matrix describing the linear component of this affine transformation is very large, because its output dimension is | **V** |. This imposes a high memory cost to represent the matrix, and a high computational cost to multiply by it. Because the softmax is normalized across all | **V** | outputs, it is necessary to perform the full matrix multiplication at training time as well as test time—we cannot calculate only the dot product with the weight vector for the correct output. The high computational costs of the output layer thus arise both at training time (to compute the likelihood and its gradient) and at test time (to compute probabilities for all or selected words).
* Suppose that h is the top hidden layer used to predict the output probabilities . If we parametrize the transformation from h to with learned weights W and learned biases b, then the affine-softmax output layer performs the following computations:

If h contains nh elements then the above operation is O(|V| nh). With nh in the thousands and |V| in the hundreds of thousands, this operation dominates the computation of most neural language models.

**1.4.3.1) Use of a Short List**

* The first neural language models (Bengio *et al.*, 2001, 2003) dealt with the high cost of using a softmax over a large number of output words by limiting the vocabulary size to 10,000 or 20,000 words. Schwenk and Gauvain (2002) and Schwenk (2007) built upon this approach by splitting the vocabulary V into a *shortlist* L of most frequent words (handled by the neural net) and a tail T = V \L of more rare words (handled by an n-gram model). To be able to combine the two predictions, the neural net also has to predict the probability that a word appearing after context C belongs to the tail list. This may be achieved by adding an extra sigmoid output unit to provide an estimate of P (I ∈ T |C ). The extra output can then be used to achieve an estimate of the probability distribution over all words in V as follows:

+ 1i∈T P(y = i | C, i ∈ T) P( i ∈ T| C )

where P(y = i | C, i ∈ L) is provided by the neural language model and P (y = i | C, i ∈ T) is provided by the n-gram model.

* An obvious disadvantage of the short list approach is that the potential generalization advantage of the neural language models is limited to the most frequent words, where, arguably, it is the least useful.

**1.4.3.2) Hierarchical Softmax**

* 1. **Other Applications**

**1.5.1) Recommender Systems**

* Two major types of applications can be distinguished:

1. online advertising
2. item recommendations

* Both rely on predicting the association between a user and an item, either to predict the probability of some action (the user buying the product, or some proxy for this action) or the expected gain (which may depend on the value of the product) if an ad is shown or a recommendation is made regarding that product to that user.
* This often ends up being either a regression problem (predicting some conditional expected value) or a probabilistic classification problem (predicting the conditional probability of some discrete event).
* The early work on recommender systems relied on minimal information as inputs for these predictions: the user ID and the item ID.
* Suppose that user 1 and user 2 both like items A, B and C. From this, we may infer that user 1 and user 2 have similar tastes. If user 1 likes item D, then this should be a strong cue that user 2 will also like D. Algorithms based on this principle come under the name of collaborative filtering. Both non-parametric approaches (such as nearest-neighbor methods based on the estimated similarity between patterns of preferences) and parametric methods are possible.
* Parametric methods often rely on learning a distributed representation (also called an embedding) for each user and for each item. Bilinear prediction of the target variable (such as a rating) is a simple parametric method that is highly successful and often found as a component of state-of-the-art systems. The prediction is obtained by the dot product between the user embedding and the item embedding (possibly corrected by constants that depend only on either the user ID or the item ID).
* Let be the matrix containing our predictions, **A** a matrix with user embeddings in its rows and **B** a matrix with item embeddings in its columns. Let **b** and **c** be vectors that contain respectively a kind of bias for each user (representing how grumpy or positive that user is in general) and for each item (representing its general popularity). The bilinear prediction is thus obtained as follows:
* Typically, one wants to minimize the squared error between predicted ratings and actual ratings **R**u,i.
* One way to obtain these embeddings is by performing a singular value decomposition of the matrix **R** of actual targets (such as ratings). This corresponds to factorizing **R = UDV’** (or a normalized variant) into the product of two factors, the lower rank matrices **A = UD** and **B = V’**. One problem with the SVD is that it treats the missing entries in an arbitrary way, as if they corresponded to a target value of 0. Instead we would like to avoid paying any cost for the predictions made on missing entries. Fortunately, the sum of squared errors on the observed ratings can also be easily minimized by gradient-based optimization.
* The SVD and the bilinear prediction of above equation both performed very well in the competition for the Netflix prize (Bennett and Lanning, 2007), aiming at predicting ratings for films, based only on previous ratings by a large set of anonymous users.
* Beyond these bilinear models with distributed representations, one of the first uses of neural networks for collaborative filtering is based on the RBM undirected probabilistic model (Salakhutdinov et al., 2007).
* RBMs were an important element of the ensemble of methods that won the Netflix competition (Töscher et al., 2009; Koren, 2009). More advanced variants on the idea of factorizing the ratings matrix have also been explored in the neural networks community (Salakhutdinov and Mnih, 2008).
* However, there is a basic limitation of collaborative filtering systems: when a new item or a new user is introduced, its lack of rating history means that there is no way to evaluate its similarity with other items or users (respectively), or the degree of association between, say, that new user and existing items. This is called the problem of **cold-start recommendations**.
* A general way of solving the cold-start recommendation problem is to introduce extra information about the individual users and items. For example, this extra information could be user profile information or features of each item. Systems that use such information are called **content-based recommender systems**. The mapping from a rich set of user features or item features to an embedding can be learned through a deep learning architecture (Huang *et al.*, 2013; Elkahky *et al.*, 2015).

**1.5.1.1) Exploration Versus Exploitation**

* When making recommendations to users, an issue arises that goes beyond ordinary supervised learning and into the realm of reinforcement learning. Many recommendation problems are most accurately described theoretically as **contextual bandits** (Langford and Zhang, 2008; Lu *et al.*, 2010).
* The issue is that when we use the recommendation system to collect data, we get a biased and incomplete view of the preferences of users: we only see the responses of users to the items they were recommended and not to the other items.
* More importantly, we get no information about what outcome would have resulted from recommending any of the other items. This would be like training a classifier by picking one class for each training example x (typically the class with the highest probability according to the model) and then only getting as feedback whether this was the correct class or not. Clearly, each example conveys less information than in the supervised case where the true label y is directly accessible, so more examples are necessary. Worse, if we are not careful, we could end up with a system that continues picking the wrong decisions even as more and more data is collected, because the correct decision initially had a very low probability: until the learner picks that correct decision, it does not learn about the correct decision. This is similar to the situation in reinforcement learning where only the reward for the selected action is observed.
* The bandits scenario is a special case of reinforcement learning, in which the learner takes only a single action and receives a single reward. The bandit problem is easier in the sense that the learner knows which reward is associated with which action. In the general reinforcement learning scenario, a high reward or a low reward might have been caused by a recent action or by an action in the distant past. The term contextual bandits refers to the case where the action is taken in the context of some input variable that can inform the decision. For example, we at least know the user identity, and we want to pick an item. The mapping from context to action is also called a policy. The feedback loop between the learner and the data distribution (which now depends on the actions of the learner) is a central research issue in the reinforcement learning and bandits literature.
* Reinforcement learning requires choosing a tradeoff between exploration and exploitation. Exploitation refers to taking actions that come from the current, best version of the learned policy—actions that we know will achieve a high reward. Exploration refers to taking actions specifically in order to obtain more training data. If we know that given context x, action a gives us a reward of 1, we do not know whether that is the best possible reward. We may want to exploit our current policy and continue taking action a in order to be relatively sure of obtaining a reward of 1. However, we may also want to explore by trying action a’. We do not know what will happen if we try action a’. We hope to get a reward of 2, but we run the risk of getting a reward of 0. Either way, we at least gain some knowledge.
* Exploration can be implemented in many ways, ranging from occasionally taking random actions intended to cover the entire space of possible actions, to model-based approaches that compute a choice of action based on its expected reward and the model’s amount of uncertainty about that reward.
* Many factors determine the extent to which we prefer exploration or exploitation. One of the most prominent factors is the time scale we are interested in. If the agent has only a short amount of time to accrue reward, then we prefer more exploitation. If the agent has a long time to accrue reward, then we begin with more exploration so that future actions can be planned more effectively with more knowledge. As time progresses and our learned policy improves, we move toward more exploitation.
* Supervised learning has no tradeoff between exploration and exploitation because the supervision signal always specifies which output is correct for each input. There is no need to try out different outputs, we always know that the label is the best output.
* Another difficulty arising in the context of reinforcement learning, besides the exploration-exploitation trade-off, is the difficulty of evaluating and comparing different policies.
* Reinforcement learning involves interaction between the learner and the environment. This feedback loop means that it is not straightforward to evaluate the learner’s performance using a fixed set of test set input values.

**1.5.2) Knowledge Representation, Reasoning and Question Answering**

* In mathematics, a **binary relation**is a set of ordered pairs of objects. Pairs that are in the set are said to have the relation.
* For example, we can define the relation “is less than” on the set of entities {1, 2, 3} by defining the set of ordered pairs S = {(1 ,2), (1, 3), (2, 3)}.
* We could define a relation “is\_a\_type\_of” containing tuples like (dog, mammal).
* The relation plays the role of a verb, while two arguments to the relation play the role of its subject and object. These sentences take the form of a triplet of tokens

(subject, verb, object)

with values

(entityi, relationj, entityk).

* We can also define an **attribute**, a concept analogous to a relation, but taking only one argument:

(entityi, attributej).

For example, we could define the “has\_fur” attribute, and apply it to entities like dog.

* We can infer relations between entities from training datasets consisting of unstructured natural language. There are also structured databases that identify relations explicitly. A common structure for these databases is the **relational database**.
* When a database is intended to convey commonsense knowledge about everyday life or expert knowledge about an application area to an artificial intelligence system, we call the database a knowledge base. Knowledge bases range from general ones like Freebase, OpenCyc, WordNet, or Wikibase, 1 etc. to more specialized knowledge bases, like GeneOntology.2 Representations for entities and relations can be learned by considering each triplet in a knowledge base as a training example and maximizing a training objective that captures their joint distribution (Bordes et al., 2013a).
* In addition to training data, we also need to define a model family to train. A common approach is to extend neural language models to model entities and relations. Neural language models learn a vector that provides a distributed representation of each word. They also learn about interactions between words, such as which word is likely to come after a sequence of words, by learning functions of these vectors. We can extend this approach to entities and relations by learning an embedding vector for each relation. In fact, the parallel between modeling language and modeling knowledge encoded as relations is so close that researchers have trained representations of such entities by using *both* knowledge bases *and* natural language sentences (Bordes *et al.*, 2011, 2012; Wang *et al.*, 2014a) or combining data from multiple relational databases (Bordes *et al.*, 2013b). Many possibilities exist for the particular parametrization associated with such a model.
* Early work on learning about relations between entities (Paccanaro and Hinton, 2000) posited highly constrained parametric forms (“linear relational embeddings”), often using a different form of representation for the relation than for the entities.
* For example, Paccanaro and Hinton (2000) and Bordes *et al.* (2011) used vectors for entities and matrices for relations, with the idea that a relation acts like an operator on entities. Alternatively, relations can be considered as any other entity (Bordes *et al.*, 2012), allowing us to make statements about relations, but more flexibility is put in the machinery that combines them in order to model their joint distribution.
* A practical short-term application of such models is **link prediction** i.e. predicting missing arcs in the knowledge graph. This is a form of generalization to new facts, based on old facts.
* Most of the knowledge bases that currently exist have been constructed through manual labor, which tends to leave many and probably the majority of true relations absent from the knowledge base. See Wang *et al.* (2014b), Lin *et al.* (2015) and Garcia-Duran *et al.* (2015) for examples of such an application.
* Evaluating the performance of a model on a link prediction task is difficult because we have only a dataset of positive examples (facts that are known to be true). If the model proposes a fact that is not in the dataset, we are unsure whether the model has made a mistake or discovered a new, previously unknown fact.
* A common way to construct interesting examples that are probably negative (facts that are probably false) is to begin with a true fact and create corrupted versions of that fact, for example by replacing one entity in the relation with a different entity selected at random. The popular precision at 10% metric counts how many times the model ranks a “correct” fact among the top 10% of all corrupted versions of that fact.
* Another application of knowledge bases and distributed representations for them is word-sense disambiguation (Navigli and Velardi, 2005; Bordes et al., 2012), which is the task of deciding which of the senses of a word is the appropriate one, in some context.
* Eventually, knowledge of relations combined with a reasoning process and understanding of natural language could allow us to build a general question answering system. A general question answering system must be able to process input information and remember important facts, organized in a way that enables it to retrieve and reason about them later. This remains a difficult open problem which can only be solved in restricted “toy” environments. Currently, the best approach to remembering and retrieving specific declarative facts is to use an explicit memory mechanism.
* Kumar et al. (2015) have proposed an extension that uses GRU recurrent nets to read the input into the memory and to produce the answer given the contents of the memory.