Chapter 12

Applications

In this chapter, we describe how to put deep learning models to practical use. We begin by discussing the large scale neural network implementations required for most serious AI applications. Next, we review several specific application areas that deep learning has been used to solve. While one goal of deep learning is to design algorithms that are capable of solving a broad variety of tasks, so far some degree of specialization is needed. For example, vision tasks require processing a large number of input features (pixels) per example. Language tasks require modeling a large number of possible values (words in the vocabulary) per input feature.

12.1 Large Scale Deep Learning

Deep learning is based on the philosophy of connectionism: while an individual biological neuron or an individual feature in a machine learning model is not intelligent, a large population of these neurons or features acting together can exhibit intelligent behavior. It truly is important to emphasize the fact that the number of neurons must be *large*. One of the key factors responsible for the improvement in neural network's accuracy and the improvement of the complexity of tasks they can solve between the 1980s and today is the dramatic increase in the size of the networks we use. As we saw in Chapter 1.2.3, network sizes have grown exponentially for the past three decades, yet artificial neural networks are only as large as the nervous systems of insects.

Because the size of neural networks is of paramount importance, deep learning requires high performance hardware and software infrastructure.

12.1.1 Fast CPU Implementations

Traditionally, neural networks were trained using the CPU of a single machine. Today, this approach is generally considered insufficient. We now mostly use GPU computing or the CPUs of many machines networked together. Before moving to these expensive setups, researchers worked hard to demonstrate that CPUs could not manage the high computational workload required by neural networks.

A description of how to implement efficient numerical CPU code is beyond the scope of this book, but we emphasize here that careful implementation for specific CPU families can yield large improvements. 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. Each new model of CPU has different performance characteristics, so sometimes floating-point implementations can be faster too. 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, including optimizing data structures to avoid cache misses and using vector instructions. Many machine learning researchers neglect these implementation details, but when they restrict the size of the network one can train, they in turn restrict the machine learning capabilities of the network.

12.1.2 GPU Implementations

Most modern neural network implementations are based on graphics processing units. Graphics processing units (GPUs) are specialized hardware components that were originally developed for graphics applications. The consumer market for video gaming systems spurred development of graphics processing hardware. The performance characteristics needed for good video gaming systems turn out to be beneficial for neural networks as well.

Video game rendering requires performing many operations in parallel quickly. Models of characters and environments are specified in terms of lists of 3-D coordinates of vertices. 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 involving 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. The computations also involve processing massive buffers of memory, containing bitmaps describing the texture (color pattern) of each object to be rendered. Together, this results in graphics cards having been designed to have a high degree of parallelism and high memory bandwidth, at the cost of having a lower clock speed and less branching capability relative to traditional CPUs.

Neural networks also benefit from the same performance characteristics. Neural networks usually involve large and numerous buffers of parameters, activation values, and gradient values, each of which must be completely updated during every step of training. These buffers are large enough to fall outside the cache of a traditional desktop computer so the memory bandwidth of the system often becomes the rate limiting factor. GPUs offer a compelling advantage over CPUs due to their high memory bandwidth. Neural network training algorithms typically do not involve much branching or sophisticated control, so they are appropriate for neural network hardware. Since neural networks can be divided into multiple individual "neurons" that can be processed independently from the other neurons in the same layer, neural networks easily benefit from the parallelism of GPU computing.

GPU hardware was originally so specialized that it could only be used for graphics tasks. 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 an early GPU and reported a 3X speedup over their CPU-based baseline. Shortly thereafter, Chellapilla *et al.* (2006) demonstrated that the same technique could be used to accelerate supervised convolutional networks.

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. This platform was rapidly adopted by deep learning researchers soon after it became available (Raina *et al.*, 2009; Ciresan *et al.*, 2010).

Writing efficient code for GP-GPUs remains a difficult task best left to specialists. 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. Different models of GPUs are able to coalesce different kinds of read or write patterns. Typically, memory operations are easier to coalesce if among n threads, thread i accesses byte $i + j$ of memory, and j is a multiple of some power of 2. The exact specifications differ between models of GPU. 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.

Due to the difficulty of writing high performance GPU code, researchers should structure their workflow to avoid needing to write new GPU code in order to test new models or algorithms. Typically, one can do this by building a software library of high performance operations like convolution and matrix multiplication, then specifying models in terms of calls to this library of operations. For example, the machine learning library Pylearn2 (Warde-Farley *et al.*, 2011) specifies all of its machine learning algorithms in terms of calls to the Theano (Bergstra *et al.*, 2010a; Bastien *et al.*, 2012) and cuda-convnet (Krizhevsky, 2010), which provide these high-performance operations. This factored approach can also ease support for multiple kinds of hardware. For example, the same Theano program can run on either CPU or GPU, without needing to change any of the calls to Theano itself. Other libraries like Torch (Collobert *et al.*, 2011b) provide similar features.

12.1.3 Large Scale Distributed Implementations

In many cases, the computational resources available on a single machine are insufficient. We therefore want to distribute the workload of training and inference across many machines.

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.

Data parallelism during training is somewhat harder. We can increase the size of the minibatch used for a single SGD, but usually we get less than linear returns in terms of optimization performance. It would be better to allow multiple machines to compute multiple gradient descent steps in parallel. Unfortunately, the standard defintion of gradient descent is as a completely sequential algorithm: the gradient at step t is a function of the parameters produced by step $t-1$.

This can be solved using *asynchronous stochastic gradient descent* (Bengio *et al.*, 2001a; 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 (Chilimbi *et al.*, 2014; Wu *et al.*, 2015). Academic deep learning researchers typically cannot afford the same scale of distributed learning systems but some research has focused on how to build distributed networks with relatively low-cost hardware available in the university setting (Coates *et al.*, 2013).

12.1.4 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. For applications that do not require personalization, it is possible to train a model once, then deploy it to be used by billions of users. In many cases, the end user is more resource-constrained than the developer. For example, one might train a speech recognition network with a powerful computer cluster, then deploy it on mobile phones.

A key strategy for reducing the cost of inference is *model compression* (Buciluǎ *et al.*, 2006). The basic idea of model compression is to replace the original, expensive model with a smaller model that requires less resources 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 $f(\boldsymbol{x})$, 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 $f(\boldsymbol{x})$, we can generate a training set containing infinitely many examples, simply by applying f to randomly sampled points $\boldsymbol{x}$. We then train the new, smaller, model to match $f(\boldsymbol{x})$ on these points. In order to most efficiently use the capacity of the new, small model, it is best to sample the new $\boldsymbol{x}$ 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).

12.1.5 Dynamic Structure

One strategy for accelerating data processing systems in general is to build systems that have *dynamic structure* in the graph describing the computation needed to process an input. Data processing systems can dynamically determine which subset of many neural networks should be run on a given input. Individual neural networks can also 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, 2013a; Bengio *et al.*, 2013a) , though many kinds of dynamic structure predate this term. 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.

Dynamic structure of computations is a basic computer science principle applied generally throughout the software engineering discipline. The simplest versions of dynamic structure applied to neural networks are based on determining which subset of some group of neural networks (or other machine learning models) should be applied to a particular input.

A venerable strategy for accelerating inference in a classifier is to use a *cascade* of classifiers. The basic idea is that we are trying 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 reject inputs as not containing the object with much less computation. 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 inference in a high capacity model for every example. The system as a whole has somewhat high capacity just from the use of many models, even if all of the individual models have the same capacity. It is also possible to design the cascade so that models that come later have higher capacity. 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. Another version of cascades uses the earlier models to implement a sort of hard attention mechanism: the early members of the cascade localize an object and later members of the cascade performing further processing on it. For example, Google transcribes address numbers from Street View imagery using a two-step cascade that first locates the address number with one machine learning model and then transcribes it with another (Goodfellow *et al.*, 2014d).

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. A simple way to accomplish the union of deep learning and dynamic structure is to train a decision tree in which each node uses a neural network to make the splitting decision (Guo and Gelfand, 1992), though this has typically not been done with the primary goal of accelerating inference computations.

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[1], one per expert, and the final output is obtained by the weighted combination of the experts outputs. In that case, there is no computational saving, but if a single expert is chosen by the gater for each example, we obtain the *hard mixture of experts* (Collobert *et al.*, 2001, 2002), which can considerably speed-up training and inference time.

One major obstacle to using dynamically structured systems is the decreased degree of parallelism that results from the system following different code branches for different inputs. This means that few operations in the network can be described as matrix multiplication or batch convolution on a minibatch of examples. 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

[1] obtained via a softmax non-linearity

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. In some cases, these issues can be mitigated by partitioning the examples into groups that all take the same branch, and processing these groups of examples simultaneously. This can be an acceptable strategy for minimizing the time required to process a fixed amount of examples in an offline setting. In a real-time setting where examples must be processed continuously, partitioning the workload can result in load-balancing issues. For example, if we assign one machine to process the first step in a cascade and another machine to process the last step in a cascade, then the first will tend to be overloaded and the last will tend to be underloaded. Similar issues arise if each machine is assigned to implement different nodes of a neural decision tree.

12.1.6 Specialized Hardware Implementations of Deep Networks

Since the early days of neural networks research, hardware designers have worked on specialized hardware implementations that could speed up training and/or inference of neural network algorithms. See early and more recent reviews of specialized hardware for deep networks (Lindsey and Lindblad, 1994; Beiu *et al.*, 2003; Misra and Saha, 2010).

Different forms of specialized hardware (Graf and Jackel, 1989; Mead and Ismail, 2012; Kim *et al.*, 2009; Pham *et al.*, 2012; Chen *et al.*, 2014a,b) have been considered over the last decades, starting with ASICs (application-specific integrated circuit), either with digital (based on binary representations of numbers), analog (Graf and Jackel, 1989; Mead and Ismail, 2012) (based on physical implementations of continuous values as voltages or currents) or hybrid implementations (combining digital and analog components), and in recent years with the more flexible FPGA (field programmable gated array) implementations (where the particulars of the circuit can be written on the chip after it has been built).

Whereas software implementations on general-purpose processing units (CPUs and GPUs) typically use 32 or 64 bits precision floating point representations of numbers, it has long been known that it was possible to use less precision, at least at inference time (Holt and Baker, 1991; Holi and Hwang, 1993; Presley and Haggard, 1994; Simard and Graf, 1994; Wawrzynek *et al.*, 1996; Savich *et al.*, 2007). This has become a more pressing issue in recent years as deep learning has gained in popularity in industrial products, and as the great impact of faster hardware was demonstrated with GPUs. 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). This is very different from the situation of the 1990's (the previous neural network era) where the hardware implementations of neural networks (which might take two years from inception to availability of a chip) could not keep up with the rapid progress and low prices of general-purpose CPUs. Building specialized hardware is thus a way to push the envelope further, at a time when new hardware designs are being developed for low-power devices such as phones, aiming for general-public applications of deep learning (e.g., with speech, computer vision or natural language).

Recent work on low-precision implementations of backprop-based neural nets (Vanhoucke *et al.*, 2011; Courbariaux *et al.*, 2015; 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. Whereas fixed point numbers are restricted to a fixed range (which corresponds to a given exponent in a floating point representation), dynamic fixed point representations share that range among a set of numbers (such as all the weights in one layer). Using fixed point rather than floating point representations and using less bits per number reduces the surface area, power requirements and computing time needed for performing multiplications, and multiplications are the most demanding of the operations needed to use or train a modern deep network with backprop.

12.2 Computer Vision

Computer vision has traditionally been one of the most active research areas for deep learning applications. Many of the most popular standard benchmark tasks for deep learning algorithms are forms of object recognition or optical character recognition.

Computer vision is a very broad field encompassing a wide variety of ways of processing images, and an amazing diversity of applications. 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 research on computer vision has not focused on such exotic applications that expand the realm of what is possible with imagery but rather a small core of AI goals aimed at replicating human abilities. 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. Because generative modeling has been a guiding principle of deep learning research, there is also a large body of work on image synthesis using deep models. While image synthesis *ex nihilo* is usually not considered a computer vision endeavor, models capable of image synthesis are usally useful for image restoration, a computer vision task involving repairing defects in images or removing objects from images.

12.2.1 Preprocessing

Many application areas require sophisticated preprocessing because the original input comes in a form that is difficult for many deep learning architectures to represent. 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. This is the only kind of preprocessing that is strictly necessary. 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 are able to process variable size input if their output varies in size with the input or if they use 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).

Many other kinds of preprocessing are less necessary but help to reduce the size of the model required to obtain good accuracy on the training set, the amount of time required for training, or the size of the training set required to obtain good accuracy on the test set.

Any form of preprocessing that removes some of the complexity from the vision task will accomplish both of these goals. Simpler tasks do not require as large of models to solve, and simpler solutions are more likely to generalize well. Preprocessing of this kind is usually designed to remove some kind of variability in the input data that is easy for a human designer to describe and that the human designer is confident has no relevance to the task. 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.*, 2012a).

Another approach to preprocessing is to artificially introduce more variation into the training set. *Dataset augmentation* gives the model more training data without requiring the collection of as much real data. This kind of preprocessing usually increases the optimal model size and the amount of time required for training.

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. There are many ways of quantifying the contrast of 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. Suupose we have an image represented by a tensor $\mathbf{X} \in \mathbb{R}^{r \times c \times 3}$, with $X_{i,j,0}$ being the red intensity at row i and column j, $X_{i,j,1}$ giving the green intensity and $X_{i,j,2}$ giving the green intensity. Then the contrast of the entire image is given by

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

where $\bar{\mathbf{X}}$ 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 (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. This approach is complicated by the fact that no scaling factor can change the contrast of a zero-contrast image (one whose pixels all have equal intensity). Images with very low but non-zero contrast often have little information content. Dividing by the true stand 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 ϵ. Given an input image $\mathbf{X}$, GCN produces an output image $\mathbf{X}'$, defined such that

$$X'_{i,j,k} = s\frac{X_{i,j,k} - \bar{X}}{\max\left\{\epsilon, \sqrt{\lambda + \frac{1}{3rc}\sum_{i=1}^{r}\sum_{j=1}^{c}\sum_{k=1}^{3}\left(x_{i,j,k} - \bar{X}\right)^2}\right\}}. \tag{12.1}$$

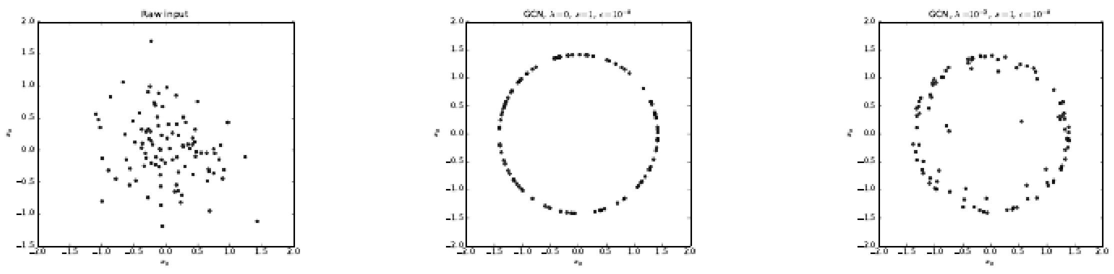

Figure 12.1: GCN maps examples onto a sphere. *Left)* Raw input data may have any norm. *Center)* GCN with $\lambda = 0$ maps all non-zero examples perfectly onto a sphere. *Right)* Regularized GCN, with $\lambda > 0$, draws examples toward the sphere but does not completely discard the variation in their norm.

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 $\lambda = 0$ and avoid division by 0 in extremely rare cases by setting ϵ to an extremely low value like 10^{-8}. This is the approach used by Goodfellow *et al.* (2013a) on the CIFAR-10 dataset. Small images cropped randomly are more likely to have nearly constant intensity, making aggressive regularization more useful. Coates *et al.* (2011) used $\epsilon = 0$ and $\lambda = 10$ on small, randomly selected patches drawn from CIFAR-10.

The scale parameter s can usually be set to 1, as done by Coates *et al.* (2011), or chosen to make each individual pixel have standard deviation across examples close to 1, as done by Goodfellow *et al.* (2013a).

Note that the standard deviation in equation 12.1 is just a rescaling of the L^2 norm of the image. It is preferable to define GCN in terms of standard deviation so that the same s may be used regardless of image size. However, this observation can be useful because it helps to understand GCN as mapping examples to a spherical shell. See Fig. 12.1 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.

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*.

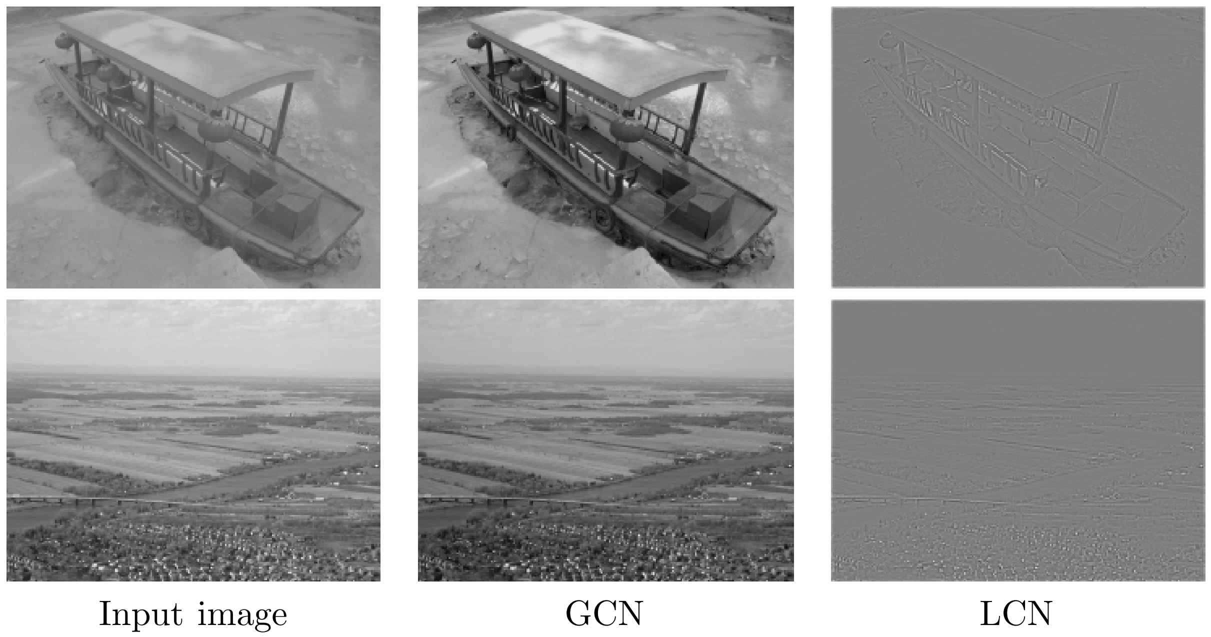

Figure 12.2: A comparison of global and local contrast normalization. Visually, the effects of global contrast normalization are subtle. It places all images on roughly the same scale, which reduces the burden on the learning algorithm to handle multiple scales. Local contrast normalization modifies 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.

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. See Fig. 12.2 for a comparison of global and local contrast normalization.

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).

Local contrast normalization can usually be implemented efficiently by using separable convolution (see Sec. 9.8) to compute feature maps of local means and local standard deviations, then using elementwise subtraction and elementwise division on different feature maps.

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.

Dataset Augmentation

As described in Sec. 7.5, it is easy to improve the generalization of a classifier by increasing the size of the training set by adding extra copies of the training examples that have been modified with transformations that do not change the class. Object recognition is a classification task that is especially amenable to this form of dataset augmentation because the class is invariant to so many transformations and the input can be easily transformed with many geometric operations. As described before, 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.*, 2012a) and non-linear geometric distortions of the input (LeCun *et al.*, 1998b).

12.3 Speech Recognition

The task of speech recognition consists in mapping an acoustic signal corresponding to a spoken natural language utterance into the corresponding sequence of words intended by the speaker. If we denote by $\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \dots \boldsymbol{x}_T)$ the input sequence of acoustic vectors (describing the recorded sounds in discrete time units such as the traditional 20ms frames), and by $\boldsymbol{y} = (y_1, y_2, \dots y_N)$ the target output or linguistic sequence (e.g., whose elements are words or characters from a natural language), the Automatic Speech Recognition (ASR) task can be described as looking for a function $f_{\text{ASR}} \approx f^*_{\text{ASR}}$, where f^*_{ASR} finds the most likely linguistic sequence $\boldsymbol{y}$ given the acoustic sequence $\boldsymbol{X}$:

$$f^*_{\text{ASR}}(\boldsymbol{X}) = \arg\max_{\boldsymbol{y}} P^*(\mathbf{y} \mid \mathbf{X} = \boldsymbol{X}) \tag{12.2}$$

where P^* is the true conditional distribution relating the inputs $\boldsymbol{X}$ to the targets $\boldsymbol{y}$.

Since the 80's and until about 2009-2012, the state-of-the art for speech recognition was held by systems combining hidden Markov models (HMMs) and Gaussian Mixture Models (GMMs). GMMs modeled the association between acoustic features and phonemes, while HMMs modeled the sequential structure of speech (which sequence of Gaussian distributions characterize a phoneme, a word?). HMMs are briefly described in Sec. 19.4.2 (Rabiner, 1989), along with GMMs. Speech recognition was one of the first areas to which neural networks were applied, in the late 80's and early 90's (Bourlard and Wellekens, 1989; Waibel *et al.*, 1989; Robinson and Fallside, 1991; Bengio *et al.*, 1991, 1992b; Konig *et al.*, 1996). Neural net based speech recognition performanced approximately matched the performance of GMMs+HMMs systems at that time. For example, Robinson and Fallside (1991) achieved 26% phoneme error rate on the TIMIT (Garofolo *et al.*, 1993) corpus (with 39 phonemes to discriminate against), which was better or comparable to HMM-based systems. TIMIT has been since then a benchmark for phoneme recognition, playing a role similar to MNIST for object recognition. However, with the complex engineering involved in software systems for speech recognition and effort that had been invested in building these systems on the basis of GMMs+HMMs, the industry did not see a compelling argument for switching to neural networks: as a consequence, both academic and industrial research in using neural nets for speech recognition remained mostly dormant until the late 2000's.

It turned out that with *much larger models*, *deeper models* (more hidden layers), and training with much larget datasets, neural nets could very advantageously replace the GMMs for the task of associating acoustic features to phonemes (or sub-phonemic states). Starting in 2009, unsupervised pretraining was used to build stacks of RBMs taking spectral acoustic representations in a fixed-size input window (around a center frame) and predicting the conditional probabilities of HMM states for that center frame. Early results on the TIMIT dataset suggested that training such deep networks actually helped to significantly improve the HMM recognition rate on TIMIT (Mohamed *et al.*, 2012), bringing down the phoneme error rate from about 26% to 23%. 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 cares not just about recognizing phonemes but about recognizing sequences of words from a large vocabulary. By that time, several of the major speech groups in industry had started exploring deep learning in collaboration with academic researchers. Hinton *et al.* (2012a) describe the breakthroughs achieved by these collaborators, which are now deployed in products such as mobile phones.

As it turned out 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 pre-training 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 (see Figure 2.4 of Deng and Yu (2014)). This created a rapid shift in the speech recognition community towards deep learning, at conferences such as ICASSP. In a matter of two years, most of the industrial products for speech recognition incorporated that innovation and this interest spurred a new wave of explorations for deep learning algorithms and architectures, which is still ongoing.

One of these innovations was the use of convolutional networks (Chapter 9) instead of fully-connected feedforward networks (Sainath *et al.*, 2013). In that case the input spectrogram is seen 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, without the need for the HMM. The first major breakthrough in this direction came from Graves *et al.* (2013) which trained a deep LSTM RNN (see Sec. 10.7.4), using MAP inference over the frame-to-phoneme alignment, as in LeCun *et al.* (1998c) and in the the CTC framework (Graves *et al.*, 2006; Graves, 2012), described in more detail in Sec. 19.4.1. 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%. See Pascanu *et al.* (2014a); Chung *et al.* (2014) for other variants of deep RNNs, applied in other settings.

Following that push, the idea was introduced of using an attention mechanism to let the system learn how to "align" the acoustic-level information with the phonetic-level information Chorowski *et al.* (2014).

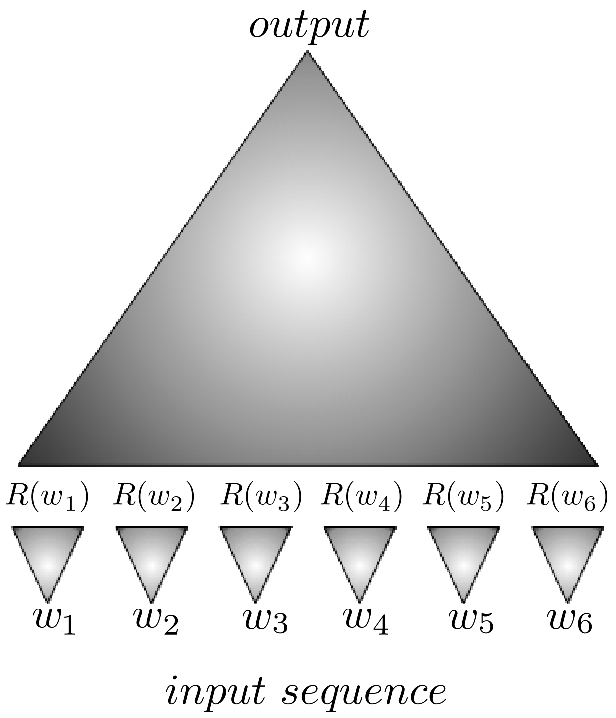
$output$
$R(w_1)$ $R(w_2)$ $R(w_3)$ $R(w_4)$ $R(w_5)$ $R(w_6)$
w_1 w_2 w_3 w_4 w_5 w_6
$input\ sequence$

Figure 12.3: Neural language models and their extensions can always be decomposed into two components: (1) the word embeddings, i.e., a mapping from any word index (a symbol) to a learned vector and (2) other parameters dedicated to the task at hand (such as predicting the next word, or translating one sentence into another), based on those representations. The training objective is defined in terms of the output of the second component. It is the second component that drives the learning of the word embeddings in such a way as to make similar words (according to the task) share attributes or dimensions in their embeddings.

12.4 Natural Language Processing and Neural Language Models

Natural language processing includes applications such as language modeling and machine translation. As with the other applications discussed in this chapter, very generic neural network techniques can be successfully applied to natural language processing. However, to achieve excellent performance and scale well to large applications, some domain-specific strategies become important. Natural language modeling usually forces us to use some of the many techniques that are specialized for processing sequential data. In many case, we choose to regard natural language as a sequence of words, rather than a sequence of individual characters. In this case, because the total number of possible words is so large, we are modeling an extremely high-dimensional and sparse discrete space. Several strategies have been developed to make models of such a space efficient, both in a computational and in a statistical sense.

12.4.1 Historical Perspective

The idea of distributed representations for symbols was introduced by Rumelhart *et al.* (1986a) in one of the first explorations of back-propagation, with symbols corresponding to the identity of family members and the neural network capturing the family relationships between family members, e.g., with examples of the form (Colin, Mother, Victoria). It turned out that the first layer of the neural network learned a representation of each family member, with learned features, e.g. for Colin, representing which family tree Colin was in, what branch of that tree he was in, what generation he was from, etc. One can think of these learned features as a set of attributes and the rest of the neural network computing micro-rules relating these attributes together in order to obtain the desired predictions, e.g., who is the mother of Colin? A similar idea was the basis of the research on neural language model started by Bengio *et al.* (2001b), where this time each symbol represented a word in a natural languge vocabulary, and the task was to predict the next word given a few previous ones. Instead of having a small set of symbols, we have a vocabulary with tens or hundreds of thousands of words (and nowadays it goes up to the million, when considering proper names and misspellings). This raises serious computational challenges, discussed below in Sec. 12.4.4. The basic idea of a neural language models and their extensions, e.g., for machine translation, is illustrated in Figure 12.3 and a specific instance (which was used by Bengio *et al.* (2001b)) is illustrated in Figure 12.3. Figure 12.3 explains the basic of idea of splitting the model into two parts, one for the word embeddings (mapping symbols to vectors) and one for the task to be performed. Sometimes, different maps can be used, e.g., for input words and output words, as in Figure 12.3, or in neural machine translation models (Sec. 12.4.6).

Earlier work had looked at modeling sequences of characters in text using neural networks (Miikkulainen and Dyer, 1991; Schmidhuber, 1996), but it turned out that working with word symbols worked better as a language model and, more importantly, immediately yielded *word embeddings*, i.e., interpretable representations of words, as illustrated in Figures 12.5 and 12.6. Actually, these compelling 2-dimensional visualization arose thanks to the the development of the t-SNE algorithm (van der Maaten and Hinton, 2008a) in 2008, and the first visualization of word embeddings was made by Joseph Turian in 2009: these t-SNE visualizations of word embedding quickly became a standard tool to understand the learned word representations.

The early efforts at training language models typically yielded neural nets that did not beat an n-gram model by themselves, but when adding the probability prediction coming from the neural net and from the n-gram model, one would typically get substantial gains in log-likelihood (Bengio *et al.*, 2003a).

An important development after the demonstration that neural language models could be used to improve upon classical n-gram models in terms of negative log-likelihood (also called perplexity in the language modeling literature) has been the demonstration that these improved language models could yield an improvement in word error rate for state-of-the-art speech recognition systems (Schwenk and Gauvain, 2002a, 2005; Schwenk, 2007). The same technique (replacing the n-gram by a combination of n-gram and neural language model) was then used to improve classical statistical machine translation systems (Schwenk *et al.*, 2006; Schwenk, 2010).

More developments of the original model are described in the sections below.

12.4.2 n-grams

n-grams are sequences of n tokens, where tokens can represent words or other discrete entities depending on the application. n-gram models are estimators of conditional probabilities based on counting relative frequencies of occurrences of n-grams. n-gram models have been the core building block of statistical language modeling for many decades (Jelinek and Mercer, 1980; Katz, 1987; Chen and Goodman, 1999). Like RNNs, they are based on the product rule (or chain rule) decomposition of the joint probability into conditionals, Eq. 10.6. All of these models use estimates of $P(x_t \mid x_1, \ldots, x_{t-1})$ to compute $P(x_1, \ldots, x_T)$. Models based on n-grams have the following additional properties:

1. They estimate these conditional probabilities based only on the last $n-1$ values (to predict the next one)

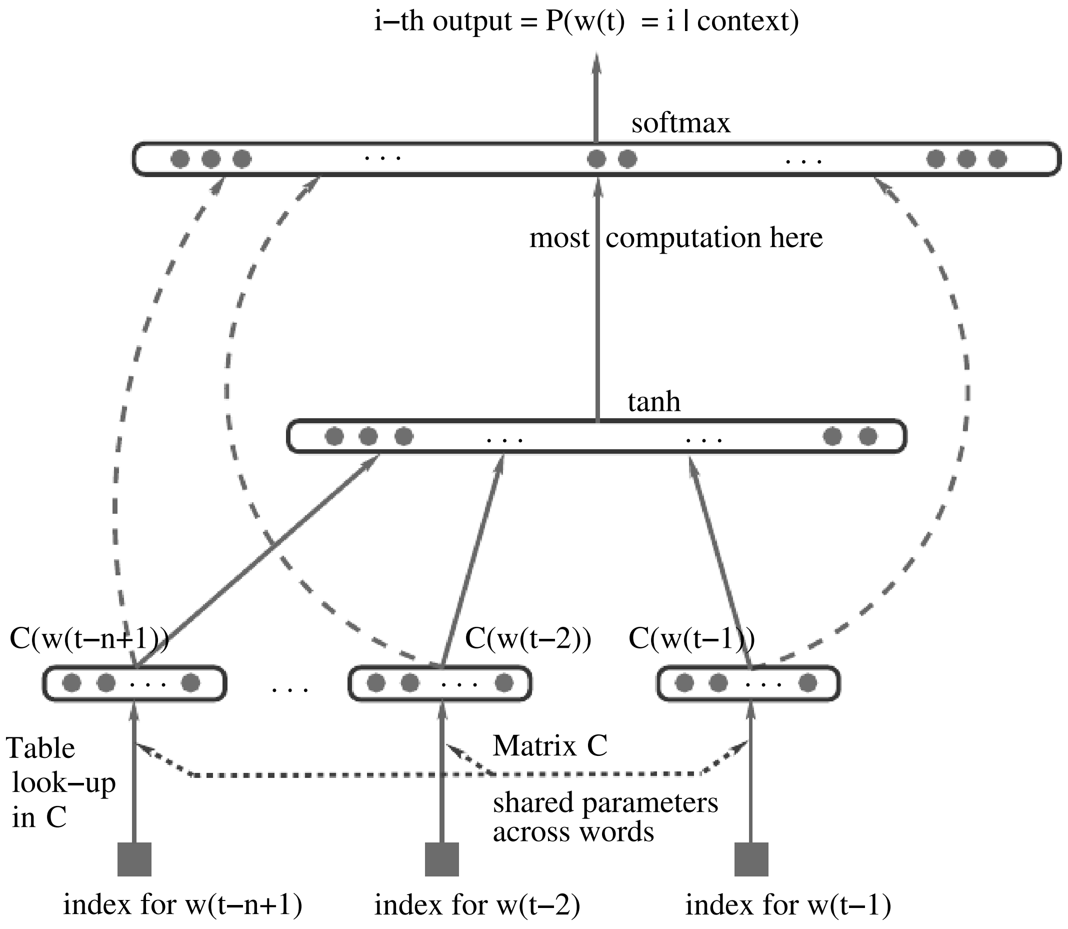

Figure 12.4: This is the original architecture for a neural language model that was developed by Bengio *et al.* (2001b) and is a special case of the general architecture for neural language-related models illustrated in Figure 12.3. Here the "second component" is an ordinary MLP with a single hidden layer and a very large softmax output layer predicting the probability of the next word (given the previous words seen in input).

2. They assume that the data is discrete. Specifically, x_t is a token taken from a finite set $\mathbb{V}$ (for vocabulary).

3. The conditional probability estimates are obtained from frequency counts of all the observed n-grams.

The names *unigram* (for n=1), *bigram* (for n=2), *trigram* (for n=3), and *n-gram* in general, derive from the Latin prefixes for the corresponding numbers and the Greek suffix "-gram" denoting something that is written. Typically the grams in an n-gram model are words or characters.

To train an n-gram model with maximum likelihood, we simply gather counts of n-grams. The probability of an n-gram, $p_n(x_1, \ldots, x_n)$, is given by the number of times we saw that n-gram in the training data, divided by the total number of n-grams in the training set.

Usually we train both an n-gram model and an $n-1$ gram model simultaneously. This makes it easy to compute

$$p(x_t \mid x_{t-n+1}, \ldots, x_t) = p_n(x_{t-n+1}, \ldots, x_t)/p_{n-1}(x_{t-n+1}, x_{t-1}) \tag{12.3}$$

simply by looking up two counts. For this to exactly reproduce inference in p_n, we must omit the final character from each sequence when we train p_{n-1}.

As an example, we demonstrate how to use trigram and bigram counts to estimate 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 $p_3(\texttt{THE DOG RAN})$. Finally, the last word may be predicted using the typical case, of using the conditional distribution $p(\texttt{AWAY} \mid \texttt{DOG RAN})$. Putting this together with Eq.
refeq:ml-ngram, we obtain:

$$p(\texttt{THE DOG RAN AWAY}) = p_3(\texttt{THE DOG RAN})p_3(\texttt{DOG RAN AWAY})/p_2(\texttt{DOG RAN}).$$

A fundamental limitation of maximum likelihood for n-gram models is that p_n is very likely to be zero in many cases, even though the tuple $(x_{t-n+1}, \ldots, x_t)$ may appear in the test set. This can cause two different kinds of catastrophic outcomes. When p_{n-1} is zero, the ratio is undefined, so the model does not even produce a sensible output. When p_{n-1} is non-zero but p_n is zero, the test log-likelihood is $-\infty$. 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. See Chen and Goodman (1999) for a review and empirical comparisons. One basic technique consists in adding non-zero probability mass to all of the possible next symbol values. This method can be justified as Bayesian inference with a uniform or Dirichlet prior over the the count parameters. Another very popular idea consists in backing off, or mixing (as in mixture model), the higher-order n-gram predictor with all the lower-order ones (with smaller n). Back-off methods look-up the lower-order n-grams if the frequency of the context $x_{t-1}, \ldots, x_{t-n+1}$ is too small. More formally, they estimate the distribution over x_t by using contexts $x_{t-n+k}, \ldots, x_{t-1}$, for increasing k, until a sufficiently reliable estimate is found.

n-gram models can be interpreted as non-parametric or as parametric. The vocabulary size $|\mathbb{V}|$ and the n impose a hard upper limit on the number of count parameters that will be stored, so asymptotically, as the size of the training set approaches infinity, the number of parameters remains constant. In this view, n-gram models are parametric, but with very sparse parameters. In realistic use cases, most n-grams are not observed during training, and the size of the model description scales with the size of the training set. In this view, n-gram models are non-parametric. Finally, the learning algorithm encompassing both hyperparameter tuning and parameter learning is trivially non-parametric, in the sense that almost any learning algorithm with configurable capacity is non-parametric, because the hyperparameter selection algorithm can increase n arbitrarily.

Classical n-gram models suffer from the curse of dimensionality (a general problem in machine learning. There are $|\mathbb{V}|^n$ possible n-grams to count and only a limited amount of training data. To overcome this problem, a language model must be able to share knowledge between one word and other semantically similar words.

To overcome this problem, *class-based language models* (Brown *et al.*, 1992; Ney and Kneser, 1993; Niesler *et al.*, 1998) introduce the notion of word categories in order to share statistical strength between words that are semantically related. The idea is to use a clustering algorithm to partition the set of words into clusters or classes, based on their co-occurence frequencies with other words. The model can then use word class IDs rather than individual words 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 clearly gives 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.

12.4.3 How Neural Language Models can Generalize Better

Neural language models are a class of deep model designed to overcome the curse of dimensionality problem. Unlike class-based 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.

The fundamental reason why neural language models can break the barrier encountered with models based on n-grams is that neural language models can share statistical strength between one word (and its context) and other similar words and contexts. This sharing is enabled by learning a distributed representation of each word. For example, if the word `dog` and the word `cat` map to representations that share many attributes (except maybe some indicator of being feline or not, for example), 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. Because there are many such attributes, there are many ways in which generalization can happen, transfering information from each training sentence to an exponentially large number of semantically related sentences. The curse of dimensionality requires the model to generalize to a number of sentences that is exponential in the sentence length. 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 semantic 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 $\sqrt{2}$ from each other. In the embedding space, semantically similar words (or any pair of words sharing some "features" learned by the model) are close to each other. Figure 12.5 demonstrates that semantically related words appear near each other in the embedding space.

Figure 12.6 zooms in on specific areas of such a picture of word embeddings to show more clearly how semantically similar words end up with representations that are close to each other.

12.4.4 High-Dimensional Outputs

A common problem in natural language applications is that it can be very computationally expensive to represent an output distribution over the choice of a word, because the vocabulary size is large. 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 $\mathbb{V}$ with size $|\mathbb{V}|$. The weight matrix describing the linear component of this affine transformation is very large, because its output dimension is $|\mathbb{V}|$. 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 $|\mathbb{V}|$ 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

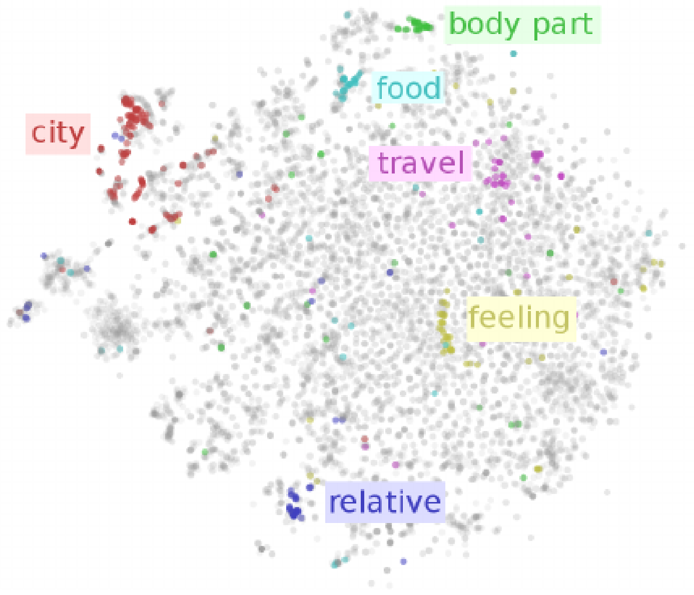

Figure 12.5: Word embeddings obtained from neural language models tend to cluster by semantic categories, as visualized here via t-SNE dimensionality reduction and coloring of words by such categories. Reproduced with permission by Chris Olah from `http://colah.github.io/`, where many more insightful visualizations can be found. 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.

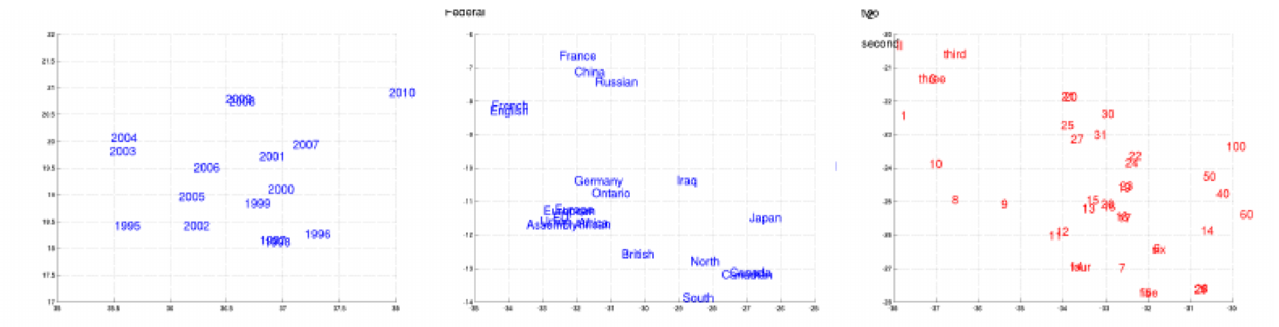

Figure 12.6: Two-dimensional visualizations of word embeddings obtained from a neural machine translation model (Bahdanau *et al.*, 2014), zooming in on specific areas where semantically related words have embedding vectors that are close to each other. Years appear on the left, countries in the middle, and numbers on the right. Compare Fig. 12.5 for a broader perspective. 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.

the correct output. The 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 $\boldsymbol{h}$ is the top hidden layer used to predict the output probabilities $\boldsymbol{p}$. If we parameterize the transformation from $\boldsymbol{h}$ to $\boldsymbol{p}$ with learned weights $\boldsymbol{W}$ and learned biases $\boldsymbol{b}$, then the affine-softmax output layer performs the following computations:

$$a_i = b_i + \sum_j W_{ij} h_j \quad \forall i \in \{1, \ldots, |\mathbb{V}|\}, \tag{12.4}$$

$$p_i = \frac{e^{a_i}}{\sum_{i' \in \{1,\ldots,|\mathbb{V}|\}} e^{a_{i'}}}. \tag{12.5}$$

If $\boldsymbol{h}$ contains n_h elements then the above operation is $O(|\mathbb{V}|n_h)$. With n_h in the thousands and $|\mathbb{V}|$ in the hundreds of thousands, this computation dominates the computation of most neural language models.

Use of a Short List

The first neural language models dealt with this problem (Bengio *et al.*, 2001a, 2003a) by limiting the vocabulary size to 10,000 or 20,000 words. Schwenk and Gauvain (2002b) and Schwenk (2007) built upon this approach by splitting the vocabulary $\mathbb{V}$ into a *shortlist* $\mathbb{L}$ of most frequent words (handled by the neural net) and a tail $\mathbb{T} = \mathbb{V} \backslash \mathbb{L}$ 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 $P(i \in \mathbb{T} \mid C)$. The extra output can then be used to achieve an estimate of the probability distribution over all words in $\mathbb{V}$ as follows:

$$\begin{aligned} P(y = i \mid C) = & 1_{i \in \mathbb{L}} P(y = i \mid C, i \in \mathbb{L})(1 - P(i \in \mathbb{T} \mid C)) \\ & + 1_{i \in \mathbb{T}} P(y = i \mid C, i \in \mathbb{T}) P(i \in \mathbb{T} \mid C) \end{aligned} \tag{12.6}$$

where $P(y = i \mid C, i \in \mathbb{L})$ is provided by the neural language model and $P(y = i \mid C, i \in \mathbb{T})$ is provided by the n-gram model. With slight modification, this approach can also work using an extra output value in the neural language model's softmax layer, rather than a separate sigmoid unit.

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. This disadvantage has stimulated the exploration of alternative methods to deal with high-dimensional outputs, described below.

Hierarchical Softmax

When trying to parametrize and compute a multinoulli probability distribution over a large set (e.g. hundreds of thousands of words) of dimension $|\mathbb{V}|$, a classical approach (Goodman, 2001) is to decompose probabilities hierarchically. Instead of having a number of computations proportional to $|\mathbb{V}|$ (and also proportional to the number of hidden units n_h, in our case), the $|\mathbb{V}|$ factor can be reduced to as low as $\log|\mathbb{V}|$. Bengio (2002) and Morin and Bengio (2005) introduced this factorized approach to the context of neural language models.

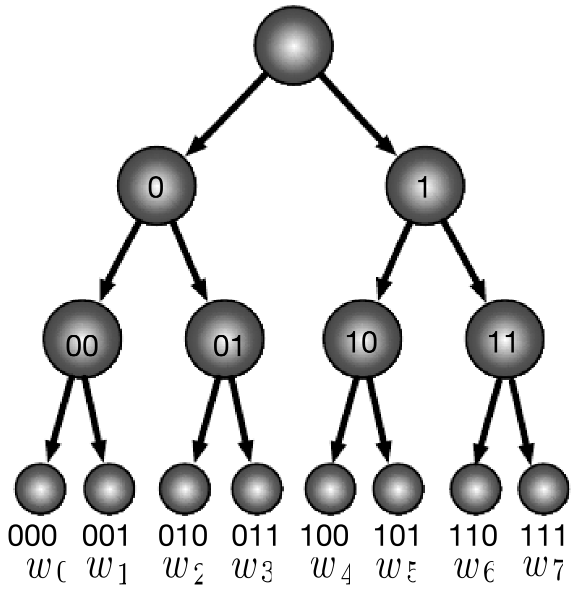

Figure 12.7: Illustration of a hierarchy of word categories, with actual words at the leaves and groups of words at the internal nodes. Any node can be indexed by the sequence of binary decisions (0=left, 1=right) to reach the node from the root. If the tree is sufficiently balanced, the maximum depth (number of binary decisions) is on the order of the logarithm of the number of words $|\mathbb{V}|$: the choice of one out of $|\mathbb{V}|$ words can be obtained by doing $O(\log|\mathbb{V}|)$ operations (one for each of the nodes on the path from the root).

One can think of this hierarchy as building categories of words, then categories of categories of words, then categories of categories of categories of words, etc. Figure 12.7 illustrates a simple example with 8 words $w_0, \ldots, w_7$ organized into a 3-level hierarchy. Super-class 0 contains the classes 00 and 01, which respectively contain the words (w_0, w_1) and (w_2, w_3). Similarly, super-class 1 contains the classes 10 and 11, which respectively contain the words (w_4, w_5) and (w_6, w_7). Hence, computing the probability of a word y can be done by computing three binomial probabilities, associated with the left or right binary decisions associated with the nodes from the root to a node y. Let b_i be the i-th binary decision when traversing the tree towards the value y. Thus, the probability of sampling and output y can be decomposed into a product of conditional probabilities, using the chain rule for conditional probabilities, with each node indexed by the prefix of these bits. For example, node 10 in Figure 12.7 corresponds to the prefix $(b_0(w_4) = 1, b_1(w_4) = 0)$, and the probability of w_4 can be decomposed as follows:

$$\begin{aligned} P(\mathrm{y} = w_4) &= P(\mathrm{b}_0 = 1, \mathrm{b}_1 = 0, \mathrm{b}_2 = 0) \\ &= P(\mathrm{b}_0 = 1) P(\mathrm{b}_1 = 0 \mid \mathrm{b}_0 = 1) P(\mathrm{b}_2 = 0 \mid \mathrm{b}_0 = 1, \mathrm{b}_1 = 0). \end{aligned} \tag{12.7}$$

Each of these conditional probabilities can be computed at one node, starting from the root, and associated with the arc going from a parent node to one of its children nodes.

For neural language models, these probabilities are typically conditioned on some context, so in general we decompose the log-likelihood of the next word y given its context as follows:

$$\log P(y \mid C) = \sum_i \log P(b_i \mid b_1, b_2, \ldots, b_{i-1}, C)$$

where the sum runs over all k bits of y. This can be obtained by computing the sigmoid of k dot products, each with a weight vector indexed by the identifier $n = b_1, b_2, \ldots, b_{i-1}$ associated with some internal node n on the path to y:

$$\begin{aligned} p_n &= \mathrm{sigmoid}(c_n + \boldsymbol{v}_n \cdot \boldsymbol{h}_C) \\ \log P(b_i \mid b_1, b_2, \ldots, b_{i-1}, C) &= b_i \log p_n + (1 - b_i) \log(1 - p_n) \end{aligned} \tag{12.8}$$

which corresponds to the usual Bernoulli cross-entropy for logistic regression and probabilistic binary classification in neural networks (Sec.s 6.3.2 and 6.3.2).

Since the output log-likelihood can be computed efficiently (as low as $\log |\mathbb{V}|$ rather than $|\mathbb{V}|$), so can its gradient with respect to the output parameters (the $\boldsymbol{v}_n$ and c_n above) as well as with respect to the hidden layer activations $\boldsymbol{h}_C$.

Note that in principle we could optimize the tree structure to minimize the expected number of computations, following Shannon's theorem, i.e., by structuring the tree so that the number of bits associated with a word be approximately equal to the logarithm of the frequency of that word. However, in practice, this is typically not worth it because the computation of the output probabilities is only one part of the total computation. For example, if there are several hidden layers of width n_h, then the associated computations grow as $O(n_h^2)$ while the output computations grow as $O(n_h L)$ where L is the average number of bits of output words (weighted by their frequency). Hence, there is not much advantage in making L much less than n_h. Consider that n_h is typically large (e.g., around a thousand or more), and the vocabulary sizes are typically not more than the order of a million. Since $\log_2(10^6)$ is about 20, we could get L on the order of 20 for such a large vocabulary, but in fact it would not make much of a difference to take L on the order of 1000 (the same as n_h), which means that a 2-level tree (which has average depth L on the order of $\sqrt{|\mathbb{V}|}$) is sufficient to reap most of the benefit of a hierarchical softmax, with the typical vocabulary sizes and hidden layer sizes that are currently used. A 2-level tree corresponds to simply defining a set of mutually exclusive words classes.

One question that remains somewhat open is how to best define these word classes, or how to define the word hierarchy in general. Early work used existing hierarchies (Morin and Bengio, 2005) but it can also be learned, ideally jointly with the neural language model, although an exact optimization of the log-likelihood appears intractable because the choice of a word hierarchy is a discrete one, not amenable to gradient-based optimization. However, one could use discrete optimization to approximately optimize the partition of words into word classes.

An important advantage of the hierarchical softmax is that it brings computational benefits both at training time and at test time, if at test time we want to compute the probability of specific words. Of course computing the probability of all $|\mathbb{V}|$ words will remain expensive. Another interesting question is how to pick the most likely word, in a given context, and unfortunately the tree structure does not provide an efficient and exact answer. However, in practice (e.g., for translation or speech recognition), we want to pick the best *sequence* of words, and this typically requires a heuristic search such as the beam search (Sec. 19.5.1).

A disadvantage is that in practice the hierarchical softmax tends to give worse test results than sampling-based methods such as described below, although this may be due to a poor choice of word classes.

Importance Sampling

An idea that is almost as old as the hierarchical softmax, for speeding up training of neural language models, is the use of a sampling technique to approximate the "negative phase" contribution of the gradient, i.e., the "counter-examples" on which the model should give a low score (or high energy), compared to the observed word. See Sec. 18.1 for the decomposition of the log-likelihood into a "positive phase" term (pushing the score of the correct word up, or pushing down its energy, which is easy here because we only have to consider one word) and a "negative phase" term (pushing down the score of all the other words, in proportion to the probability that the model gives them). Using the notation introduced in Eq. 12.4, the gradient can be written as follows:

$$\begin{aligned}\frac{\partial \log P(y \mid C)}{\partial \theta} &= \frac{\partial \log \text{softmax}_y(\boldsymbol{a})}{\partial \theta} \\ &= \frac{\partial}{\partial \theta} \log \frac{e^{a_y}}{\sum_i e^{a_i}} \\ &= \frac{\partial}{\partial \theta}(a_y - \log \sum_i e^{a_i}) \\ &= \frac{\partial a_y}{\partial \theta} - \sum_i P(i \mid C) \frac{\partial a_i}{\partial \theta})\end{aligned} \tag{12.9}$$

where $\boldsymbol{a}$ is the vector of pre-softmax activations (or scores), with one element per word. The first term is the "positive phase" term (pushing a_y up) while the second term is the "negative phase" term (pushing a_i down for all i, with weight $P(i \mid C)$. Since the negative phase term is an expectation, we can estimated by a Monte-Carlo sample. However, that would require sampling from the model itself, i.e., computing $P(i \mid C)$ for all i in the vocabulary, which is precisely what we are trying to avoid.

The solution proposed by Bengio and Sénécal (2003); Bengio and Sénécal (2008) is to sample from another distribution, called the proposal distribution (denoted q), and use appropriate weights to correct for that. This is called *importance sampling* and is introduced in Sec. 14.1.2. But even exact importance sampling is not appropriate because it requires computing weights p_i / q_i, where $p_i = P(i \mid C)$, which can only be computed if all the scores a_i are computed. The solution adopted is called *biased importance sampling*, where the importance weights are normalized to sum to 1, i.e., when negative word n_i is sampled, 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 importance to the N negative samples from q used to form the estimated negative phase contribution to the gradient:

$$\sum_{i=1}^{|\mathbb{V}|} P(i \mid C) \frac{\partial a_i}{\partial \theta}) \approx \frac{1}{N} \sum_{i=1}^{N} w_i \frac{\partial a_{n_i}}{\partial \theta}.$$

A unigram or a bigram distribution works well as the proposal distribution q, because can be easily estimated from the data as well as sampled from very efficiently.

A related application of importance sampling to speed-up training of a larger class of model was introduced by Dauphin *et al.* (2011). These are models where the output is not necessarily a 1-of-n choice (which one can think of as an integer, or as a one-hot vector), but more generally a sparse vector, where only a few of the entries are non-zero. This occurs for example when the output is a *bag-of-words*, i.e., a sparse vector where the non-zeros indicate the presence (0 or 1) or the frequency (a small count) of each word of a document. In the paper, the authors study the case of denoising auto-encoders with a bag-of-words as input. Whereas the sparsity of the input can be easily exploited (by ignoring the zeros in the computation), it is not so clear for the output (reconstruction) units. Because an auto-encoder also predicts its input, the target for the reconstruction is sparse but the prediction (probabilities that any particular word is present) is not. The algorithm ends up minimizing reconstruction error (minus log-likelihood) for the "positive words" (those that are non-zero in the target) and an equal number of "negative words" chosen randomly, but with their gradients reweighted appropriately according to importance sampling.

In all of these cases, the computational complexity of gradient estimation for the output layer is reduced to be proportional to the number of negative samples rather than proportional to the size of the output vector.

Noise-Contrastive Estimation and Ranking Loss

Other approaches based on sampling have been proposed to reduce the computational cost of training neural language models with large vocabularies.

An early one is the ranking loss proposed by Collobert and Weston (2008), in which we view the output of the neural language model for each word as a score and ask that the score of the correct word a_y be ranked high in comparison to the other scores a_i. The ranking loss proposed then is

$$L = \sum_i \max(0, 1 - a_y + a_i). \tag{12.10}$$

Note that the gradient is zero for the i-th term if the score of the observed word, a_y is greater than the score of negative word a_i by a margin of 1. One issue with this criterion is that it does not provide estimated conditional probabilities, which are useful in some applications, e.g., speech recognition, or (conditional) text generation.

A more recently used training objective for neural language model is noise-contrastive estimation, which is introduced in Sec. 18.6. The idea is to turn the training task into a probabilistic classification problem where the learner tries to identify whether a given value is sampled from the data generating distribution (under the probability estimated by the trained model) or from a fixed "noise" model. This approach has been successfully applied to neural language models (Mnih and Teh, 2012; Mnih and Kavukcuoglu, 2013). That probabilistic classifier output (when the output is the observed word y) can be obtained by combining the score of the observed word a_y with a learned parameter that estimates the log of the normalizing constant of the softmax. The training objective also requires that one samples a word from the "noise" distribution, which acts like a proposal distribution for importance sampling.

12.4.5 Combining Neural Language Models with n-grams

A major advantage of n-grams over neural networks is that n-grams achieve high model capacity (by storing the frequencies of very many tuples) while requiring very little computation to process an example (by looking up only a few tuples that match the current context). If we use hash tables or trees to access the counts, the computation used for n-grams is almost independent of capacity. In comparison, doubling a neural network's number of parameters typically also roughly doubles its computation time (exceptions include models that avoid using all parameters on each pass, like embedding layers that index a single embedding, and models that can add parameters while reducing the degree of parameter sharing, like tiled convolutional networks).

One easy way to add capacity is thus to combine both approaches in an ensemble consisting of a neural language model and an n-gram language model (Bengio *et al.*, 2001b, 2003a). As with any ensemble, this technique can also reduce test error, and many ways of combining the ensemble members' predictions are possible (uniform weighting, weights chosen on a validation set, etc.) Later, this ensemble idea was extended Mikolov *et al.* (2011a) the idea of an ensemble approach to include not just two models but a large array of models. It is also possible to pair a neural network with a maximum entropy model and train both jointly (Mikolov *et al.*, 2011b). This approach can be viewed as training a neural network with an extra set of inputs that are connected directly to the output, and not connected to any other part of the model. The extra inputs are indicators for the presence of particular n-grams in the input context, so these variables are very high-dimensional and very sparse. The increase in model capacity is huge—the new portion of the architecture contains up to $|sV|^n$ parameters—but the amount of added computation needed to process an input is minimal because the extra inputs are very sparse.

12.4.6 Neural Machine Translation

The early use of neural networks for machine translation (Schwenk *et al.*, 2006; Schwenk, 2010) took advantage of the good performance of neural language models in order to replace one component of a machine translation system, the statistical language model, which was traditionally done by an n-gram-based model.

These n-gram based model include not just traditional back-off n-gram models (Jelinek and Mercer, 1980; Katz, 1987; Chen and Goodman, 1999) but also so-called maximum entropy language models (Berger *et al.*, 1996), in which an affine / softmax layer predicts the next word given the presence of frequent n-grams in the context (as outlined in the previous section).

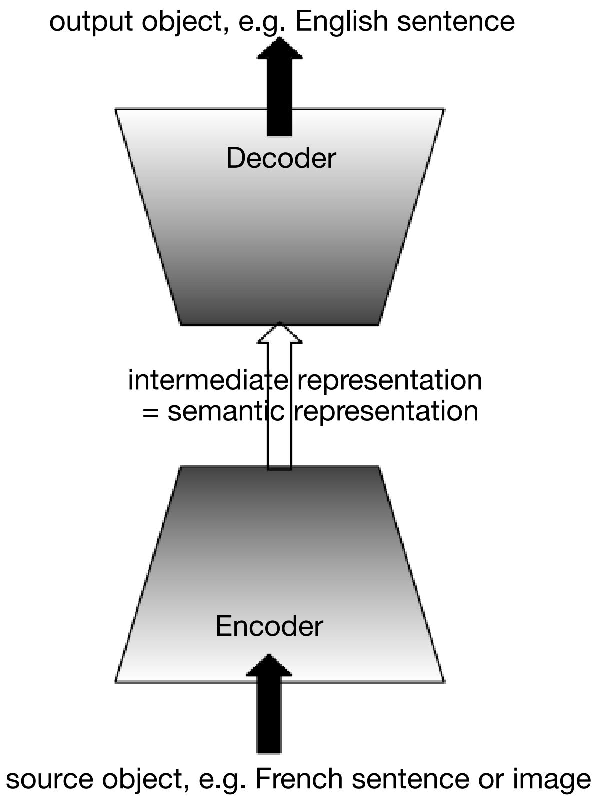
output object, e.g. English sentence
Decoder
intermediate representation
= semantic representation
Encoder
source object, e.g. French sentence or image

Figure 12.8: The encoder-decoder architecture to map back and forth between a surface representation (e.g., sequence of words, image) and a semantic representation. By coupling the encoder for one modality (e.g. French to "meaning") with the decoder for another modality (e.g. "meaning" to English), we can train systems that translate from one modality to another. This idea has been applied successfully not just to machine translation but also to caption generation from images.

However, once we have an architecture for learning neural language model that captures the joint probability $P(\mathrm{w}_1, \mathrm{w}_2, \ldots, \mathrm{w}_T)$ of a sequence of words, we can in principle make the distribution conditional on some generic context C by making some of its parameters a function of C, as explained in Sec. 6.3.2. For example, Devlin *et al.* (2014) beat the state-of-the-art in some statistical machine translation benchmarks by using an MLP to score a phrase $\mathrm{t}_1, \mathrm{t}_2, \ldots, \mathrm{t}_k$ in the target language given a phrase $\mathrm{s}_1, \mathrm{s}_2, \ldots, \mathrm{s}_n$ in the source language, i.e., estimate $P(\mathrm{t}_1, \mathrm{t}_2, \ldots, \mathrm{t}_k \mid \mathrm{s}_1, \mathrm{s}_2, \ldots, \mathrm{s}_n)$ and use it to replace the traditional phrase table (that estimates the same quantity) based on n-grams. To make this translation more flexible, we would like to use a model that can accommodate variable length inputs and variable length outputs. If an RNN is used to capture $P(\mathrm{w}_1, \mathrm{w}_2, \ldots, \mathrm{w}_T)$, we can make the initial state of the RNN or the biases used at each time step for the hidden units be a function of some generic context C. If C is obtained from a source sentence in another language, we can thus train our neural language to translate from one language to another. If we think of C as a semantic summary of the source sentence, it can for example be obtained as the final state of another RNN (the encoder RNN or "reader"), as in Cho *et al.* (2014a); Sutskever *et al.* (2014b); Jean *et al.* (2014), or as the top layer of a convolutional network, as in (Kalchbrenner and Blunsom, 2013). This general idea of an encoder-decoder framework for machine translation is illustrated in Figure 12.8.

This raises the question of representing not just words but sequences of words. The idea of learning a semantic representation of phrases and even sentences so that the representation of the source and target sentences are close to each other and can be mapped from one to the other has been explored (Kalchbrenner and Blunsom, 2013; Cho *et al.*, 2014a; Sutskever *et al.*, 2014b; Jean *et al.*, 2014), first using a combination of convolutions and RNNs (Kalchbrenner and Blunsom, 2013) and then using both RNNs for encoding the source sentence and for generating the output target-language sentence (Cho *et al.*, 2014a; Sutskever *et al.*, 2014b; Jean *et al.*, 2014).

Using an Attention Mechanism and Aligning Pieces of Data

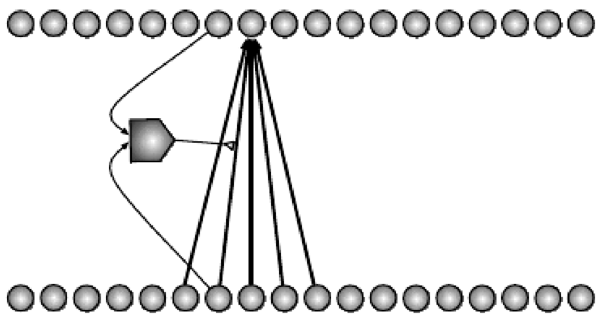

Figure 12.9: Illustration of the attention mechanism used in a neural machine translation system introduced in Bahdanau *et al.* (2014).

Using a fixed-size representation to capture all the semantic details of a very long sentence of say 60 words is very difficult. It can be achieved by training a sufficiently large RNN well enough and for long enough, as demonstrated by Cho *et al.* (2014a); Sutskever *et al.* (2014b). However, this is not how humans translate long sequences of words. What they usually do, after having read the whole sentence or paragraph (to get the context and the jist of what is being expressed), they produce the translated words one at a time, each time focusing on a different part of the input sentence in order to gather the semantic details that are required to produce the next output word. That is exactly the idea that Bahdanau *et al.* (2014) first introduced and that is illustrated in Figure 12.9.

We can think of an attention-based system as having three components:

1. A process that "*reads*" raw data (such as source words in source sentence), and converts them into distributed representations, with one feature vector associated with each word position.

2. A list of feature vectors storing the output of the reader. This can be understood as a "*memory*" containing a sequence of facts, which can be retrieved later, not necessarily in the same order, nor having to visit all of them.

3. A process that "*exploits*" the content of the memory to sequentially perform a task, at each time step having the ability put attention on the content of one memory element (or a few, with a different weight).

The third component generates the translated sentence.

When words in a sentence written in one language are aligned with corresponding words in a translated sentence in another language, it becomes possible to relate the corresponding word embeddings. Earlier work showed that one could learn a kind of translation matrix relating the word embeddings in one language with the word embeddings in another (Kočiský *et al.*, 2014), yielding lower alignmnent error rates than traditional approaches based on the frequency counts in the phrase table. There is even earlier work on learning cross-lingual word vectors (Klementiev *et al.*, 2012). Many extensions to this approach are possible. For example, more efficient cross-lingual alignment (Gouws *et al.*, 2014) allows training on larger datasets.

12.5 Structured Outputs

In principle, if we have good models $P(\mathbf{Y} \mid \boldsymbol{\omega})$ of the joint distribution of random variables $\mathbf{Y} = (\mathbf{y}_1, \dots, \mathbf{y}_k)$, with parameters $\boldsymbol{\omega}$ we can use them to build conditional models $P(\mathbf{Y} \mid \mathbf{X})$ conditioned on some input variables $\mathbf{X}$ by making $\boldsymbol{\omega}$ a parametrized function of the input $\mathbf{X}$. In Chapter 10, in particular with Sec. 10.4, we saw how an RNN can represent a joint distribution over elements of a sequence that can be conditioned. For example, with machine translation (in the previous section, 12.4.6), we condition on another sequence (the source sentence). Conditional joint distribution models are sometimes called "structured output models", to distinguish them from the more usual supervised learning tasks where the outputs represent a single random variable (like a class) or conditionally independent random variables (like different attributes, discussed in Sec. 6.3.2).

In the third part of this book, we will go beyond RNNs as means of capturing the joint distribution between output variables, conditioned on input variables. For example Restricted Boltzmann Machines (RBMs) were made conditional by Taylor *et al.* (2007); Taylor and Hinton (2009) in the context of modeling motion style and by Boulanger-Lewandowski *et al.* (2012) in the context of symbolic sequences describing polyphonic music. In both of these examples, we actually use an RBM as a "output model" for an RNN, i.e., at each time step in the sequence, we want to output a distribution over a group of random variables (articulators for Taylor and Hinton (2009), and musical notes for Boulanger-Lewandowski *et al.* (2012)), given the current state of the RNN. With the RNN-RBM (Boulanger-Lewandowski *et al.*, 2012), a generative model is set up to model a sequence of frames $\boldsymbol{x}_t$, with the following structure. An RNN captures the past context through a state variable $\boldsymbol{h}_t$ that follows a deterministic recurrence

$$\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t).$$

In addition, at each time step, a joint distribution over the elements of the next frame $\boldsymbol{x}_{t+1}$ is formed using an RBM with parameters $\boldsymbol{\omega}$:

$$P(\mathbf{x}_{t+1} = \boldsymbol{x}_{t+1} \mid \boldsymbol{\omega}_t).$$

The RBM parameters $\boldsymbol{\omega}_t = (\boldsymbol{\omega}_{\text{RBM}}, \boldsymbol{\omega}'_t)$ are composed of two subsets of parameters, the parameters $\boldsymbol{\omega}_{\text{RBM}}$ that are ordinary parameters (namely the weights of the RBM and the visible units biases) and the parameters $\boldsymbol{\omega}'_t$ that are conditioned on the RNN state $\boldsymbol{h}_t$ (namely, the hidden units biases):

$$\boldsymbol{\omega}'_t = g(\boldsymbol{h}_t).$$

where both f and g have free parameters that are updated by SGD, along with $\boldsymbol{\omega}_{\text{RBM}}$. Since $\boldsymbol{\omega}_t$ is a function of the past frames, we are modeling the joint distribution of the sequence of frames:

$$\begin{aligned} P(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T) &= \prod_t P(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \ldots, \boldsymbol{x}_1) \\ &= \prod_t P(\boldsymbol{x}_{t+1} \mid \boldsymbol{\omega}_t) = \prod_t P(\mathbf{x}_{t+1} \mid \boldsymbol{h}_t). \qquad (12.11) \end{aligned}$$

Our goal is to construct a vector $\boldsymbol{\delta\omega}$ that approximates the log-likelihood gradient on the RBM parameters. Specifically, we want the condition

$$\boldsymbol{\delta\omega} \approx \nabla_{\boldsymbol{\omega}} \log P(\mathbf{x}_{t+1} = \boldsymbol{x}_{t+1} \mid \boldsymbol{\omega})$$

to hold. The RBM log-likelihood gradient is intractable, but we can achieve the desired approximation by setting $\boldsymbol{\delta\omega}$ to the output of an algorithm called contrastive divergence. The contrastive divergence algorithm estimates the true log-likelihood gradient, but the estimate is only an approximation. In the same way that we have decomposed the RBM parameters ω_t into the two groups $\boldsymbol{\omega}_{\text{RBM}}$ and $\boldsymbol{\omega}'_t$, we can decompose $\boldsymbol{\delta\omega}$ into the corresponding estimated gradients $\boldsymbol{\delta\omega}_{\text{RBM}}$ and $\boldsymbol{\delta\omega}'_t$. The estimated gradient $\boldsymbol{\delta\omega}_{\text{RBM}}$ can be used to update $\boldsymbol{\omega}_{\text{RBM}}$ directly (e.g., by SGD) while $\boldsymbol{\delta\omega}'_t$ can be back-propagated through the RNN (as if it was the true gradient of $\log P(\mathbf{x}_{t+1} = \boldsymbol{x}_{t+1} \mid \boldsymbol{\omega}_t)$ with respect to $\boldsymbol{\omega}'_t$), thereby providing the estimated gradient on the parameters of f and g. The use of the contrastive divergence update to train conditional RBMs was introduced by Taylor *et al.* (2007).

12.6 Other Applications

In this section we cover a few other types of applications of deep learning that are different from the standard object recognition, speech recognition and natural language processing tasks discussed above. The third part of this book will expand that scope even further to include tasks requiring the ability to generate samples or conditional high-dimensional samples (unlike "the next word", in language models).

12.6.1 Dimensionality Reduction and Information Retrieval

Dimensionality reduction was one of the first applications of representation learning and deep learning, and was one of the early motivations for studying auto-encoders. For example, Hinton and Salakhutdinov (2006) trained a stack of RBMs and then used their weights to initialize a deep auto-encoder with gradually smaller hidden layers, culminating in a bottleneck of 30 units. The layer widths were 784, 100, 500, 250 and 30 units per layer in the encoder, and the same in reverse for the decoder. The resulting code yielded less reconstruction error than PCA into 30 dimensions and the learned representation was qualitatively easier to interpret and relate to the underlying categories, with these categories manifesting as well-separated clusters.

Lower-dimensional representations can improve performance on many tasks, such as classification. Models of smaller spaces consume less computational resources. Many forms of imensionality reduction operation place semantically related examples near each other, as observed by Salakhutdinov and Hinton (2007) and Torralba *et al.* (2008). The hints provided by the mapping to the lower-dimensional space aid generalization.

One task that benefits even more than usual from dimensionality reduction is *information retrieval*, the task of finding entries in a database that resemble a query entry. This task derives the usual benefits from dimensionality reduction that other tasks do, but also derives the additional benefit that search can become extremely efficient in certain kinds of low dimensional spaces. Specifically, if we train the dimensionality reduction algorithm to produce a code that is low-dimensional and *binary*, then we can store all database entries in a hash table mapping binary code vectors to entries, and perform information retrieval by returning all database entries that have the same binary code as the query. We can also search over slightly less similar entries very efficiently, just by flipping individual bits from the encoding of the query. This approach to information retrieval via dimensionality reduction and binarization is called *semantic hashing* (Salakhutdinov and Hinton, 2007), and has been applied to both textual input (Salakhutdinov and Hinton, 2007), and images (Torralba *et al.*, 2008; Weiss *et al.*, 2008).

To produce binary codes for semantic hashing, one typically uses an encoding function with sigmoids on the final layer. The sigmoid units must be trained to be saturated to nearly 0 or nearly 1 for all input values. One trick that can accomplish this is simply to inject additive noise just before the sigmoid non-linearity during training. The magnitude of the noise should increase over time. To fight that noise and preserve as much information as possible, the network must increase the magnitude of the inputs to the sigmoid function, until saturation occurs.

The idea of learning a hashing function has been further explored in several directions, including the idea of training the representations so as to optimize a loss more directly linked to the task of finding nearby examples in the hash table (Norouzi and Fleet, 2011).

There are other applications of deep learning in information retrieval. In particular, using neural language models and trained word embeddings can make natural language queries more robust because two semantically similar queries would tend to be associated to nearby representations, if these are based on their word embeddings. For example Sordoni *et al.* (2015) suggest a better query based on the recent queries and results previously obtained, using a novel hierarchical recurrent neural network architecture in which each a low-level recurrent net processes the sequence of words in one query and outputs a representation that is the input for a query-level recurrent network, with one “time step” per query in the sequence of queries.

12.6.2 Recommender Systems

One of the major families of applications of machine learning in the information technology sector is the ability to make recommendations of items to potential users or customers. Two major types of applications can be distinguished: online advertising and item recommendations (often these recommendations are still for the purpose of selling a product). 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. The internet is currently financed in great part by various forms of online advertising, and there are major parts of the economy that rely on buying online. Companies such as Amazon and eBay are known to use machine learning and specifically deep learning[2] for their product recommendations. Sometimes, the items are not products that are actually for sale. Examples include the recommendation of posts on social networks, recommending movies to watch, recommending jokes, recommending advice from experts, matching players for video games, or matching people in dating services.

Often, this association problem is handled like a supervised learning problem: given some information about the item and about the user, predict the proxy of interest (user clicks on ad, user enters a rating, user clicks on a "like" button, user buys product, user spends some amount of money on the product, user spends time visiting a page for the product, etc). 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 the minimal information available as inputs for these predictions: the user ID and the item ID. In this context, the only way to generalize is to rely on the similarity between the patterns of values of the target variable for different users or for different items. If two users both like A, B and C, and 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, the latter often relying on learning a distributed representation (also called embedding) for each user and for each item. The simplest one of these, and a highly successful method that one often finds as a component of state-of-the-art systems, corresponds to a bilinear prediction of the target variable (such as a rating). 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 u or the item ID u). Let $\hat{\boldsymbol{R}}$ be the matrix containing our predictions, $\boldsymbol{A}$ a matrix with user embeddings

[2]`http://tamebay.com/2015/08/ebay-reveals-how-its-using-deep-learning-in-search.html`, `http://www.wired.com/2015/04/now-anyone-can-tap-ai-behind-amazons-recommendations/`

in its rows and $\boldsymbol{B}$ a matrix with item embeddings in its columns. Let $\boldsymbol{b}$ and $\boldsymbol{c}$ 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:

$$\hat{R}_{u,i} = b_u + c_i + \sum_j A_{u,j} B_{j,i}. \tag{12.12}$$

Typically one wants to minimize the squared error between predicted ratings $\hat{R}_{u,i}$ and actual ratings $\boldsymbol{R}_{u,i}$. User embeddings and item embeddings can then be conveniently visualized when they are first reduced to a low dimension (two or three), or they can be used to compare users or items against each other, just like word embeddings. One way to obtain these embeddings is by performing a singular value decomposition of the matrix $\boldsymbol{R}$ of actual targets (such as ratings). This corresponds to factorizing $\boldsymbol{R} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}'$ (or a normalized variant) into the product of two factors, the lower rank matrices $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{D}$ and $\boldsymbol{B} = \boldsymbol{V}'$. 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. The good news is that the sum of squared errors on the observed ratings can also be easily minimized by gradient-based optimization. It turned out that the SVD or bilinear prediction of Eq. 12.12 performed very well in the competition for the Netflix prize[3], aiming at predicting ratings for films, based only on previous ratings by a large set of anonymous users. Many machine learning experts participated in this competition, which took place between 2006 and 2009. It raised the level of research in recommender systems using advanced machine learning and yielded improvements in recommender systems. Even though it did not win by itself, the simple bilinear prediction or SVD was a component of the ensemble models presented by most of the competitors, including the winners (Töscher *et al.*, 2009; Koren, 2009).

Beyond these bilinear models based on distributed representations, one of the first uses of neural networks for collaborative filtering is based on the RBM (Salakhutdinov *et al.*, 2007). RBMs turned out to be 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 considered, there is no way to evaluate its similarity with other items or users (respectively), or the degree of association between, say,

[3] `http://www.netflixprize.com/`

that new user and existing items. This is called the problem of cold-start recommendations, and a general way of solving it is to introduce extra information about the individual users and items, such as user profile information or item features, and systems that use such information are called *content-based recommender systems*. The mapping from a rich set of user features or item features to their embedding can be learned through a deep learning architecture (Huang *et al.*, 2013; Elkahky *et al.*, 2015)

Specialized deep learning architectures such as convolutional networks have also been applied to learn to extract features from rich content such as from musical audio tracks, for music recommendation (van den Oörd *et al.*, 2013). In that work, the convolutional net takes acoustic features as input and computes an embedding for the associated song. The dot product between this song embedding and the embedding for a user is then used to predict whether a user will listen to a particular song.

Exploration Versus Exploitation

When making recommendations to users, an issue arises that goes beyond ordinary supervised learning, and into the realm of reinforcement learning, into what is called *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. In addition, in some cases we may not get any information on users for whom no recommendation has been made (for example, with ad auctions, it may be that the price proposed for an ad was below a minimum price threshold, or does not win the auction, so the ad is not shown at all). 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 input case (typically the class with the highest probability) and then only getting as feedback whether this was the correct class or not. Clearly, each example conveys less information 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 doesn't learn about the correct decision. This is similar to the situation in reinforcement learning where only the reward for the selected actions are observed. The difference between the bandits situation and the more general reinforcement learning situation is that with bandits there is only action and one reward per learning episode, whereas in general there might be a sequence of actions and rewards, with no clear assignment of credit or blame to the different actions. 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 $\boldsymbol{x}$, 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 to determine if one is better than the model's current output—we always know that the label is the best output.

Besides the exploration-exploitation trade-off, the feedback loop between learning and the environment via actions and observed examples also makes it less trivial to evaluate and compare different policies on data that was generated using another policy, but solutions exist (Dudik *et al.*, 2011).

12.6.3 Knowledge Representation, Reasoning and Question Answering

Deep learning approaches have been very successful in language modeling, machine translation and natural language processing thanks to the concept of word vectors or word embeddings (Bengio *et al.*, 2001b), derived from the earlier idea of distributed representations for symbols (Rumelhart *et al.*, 1986a). This is still a kind of shallow representation that captures semantic knowledge at the level of individual words and concepts. How about representing phrases and relations between words? How about representing facts and knowledge, so as to be able to answer questions? Machine learning is already used for this purpose in search engines but much more remains to be done to explore how to answer these questions.

Knowledge, Relations and Question Answering

One such interesting question is how distributed representations can be trained to capture the *relations* between two entities. These relations allow use to formalize facts about objects and how objects interact with each other.

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 while those who are not in the set do not. 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 $\mathbb{S} = \{(1, 2), (1, 3), (2, 3)\}$. Once this relation is defined, we can use it like a verb. Because $(1, 2) \in \mathbb{S}$, we say that 1 is less than 2. Because $(2, 1) \notin \mathbb{S}$, we can not say that 2 is less than 1. Of course, the entities that are related to one another need not be numbers. We could define a relation `is_a_type_of` containing tuples like (`dog`, `mammal`).

In the context of AI, we think of a relation as a sentence in a syntactically simple and highly structured language. 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

$$(\text{subject}, \text{verb}, \text{object}$$

with values

$$(\text{entity}_i, \text{relation}_j, \text{entity}_k).$$

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

$$(\text{entity}\ _i, \text{attribute}_j\ ).$$

For example, we could define the `has_fur` attribute, and apply it to entities like `dog`.

Many applications require representing relations and reasoning about them. How should we best do this within the context of neural networks?

Machine learning models of course require training data. 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*, which stores this same kind of information, albeit not formatted as three token sentences. 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`[4], etc. to more specialized knowledge bases, like `GeneOntology`[5]. 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); 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 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*: predicting

[4]Respectively available from these web sites: `freebase.com`, `cyc.com/opencyc`, `wordnet.princeton.edu`, `wikiba.se`

[5]`geneontology.org`

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 labour, 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); Garcia-Duran *et al.* (2015) for examples of such an application. In general, only positive examples of facts are known, so the metrics used (and also the objective function) are those used in information retrieval, based on ranking and precision. For example, precision @10% counts how many time the "correct" fact appears among the 10% highest scoring ones, when we consider all the corrupted variants of the fact (e.g., by replacing one of entities by any one in the set of entities). 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 it can 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, as described in Sec. 10.7.5. Memory networks were first proposed to solve a toy question answering task (Weston *et al.*, 2014). 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.