8.2.5 Pooling

The pooling operation is, however, quite different. The pooling operation works on small grid regions of size $P_q \times P_q$ in each layer, and produces another layer *with the same depth* (unlike filters). For each square region of size $P_q \times P_q$ in each of the d_q activation maps, the *maximum* of these values is returned. This approach is referred to as *max-pooling*. If a stride of 1 is used, then this will produce a new layer of size $(L_q - P_q + 1) \times (B_q - P_q + 1) \times d_q$. However, it is more common to use a stride $S_q > 1$ in pooling. In such cases, the length of the new layer will be $(L_q - P_q)/S_q + 1$ and the breadth will be $(B_q - P_q)/S_q + 1$. Therefore, pooling drastically reduces the spatial dimensions of each activation map.

Unlike with convolution operations, pooling is done at the level of *each* activation map. Whereas a convolution operation simultaneously uses all d_q feature maps in combination with a filter to produce a single feature value, pooling independently operates on each feature map to produce another feature map. Therefore, the operation of pooling does not change the number of feature maps. In other words, the depth of the layer created using pooling is the same as that of the layer on which the pooling operation was performed. Examples of pooling with strides of 1 and 2 are shown in Figure 8.4. Here, we use pooling over 3×3 regions. The typical size P_q of the region over which one performs pooling is 2×2. At a stride of 2, there would be no overlap among the different regions being pooled, and it is quite common to use this type of setting. However, it has sometimes been suggested that it is desirable to have at least some overlap among the spatial units at which the pooling is performed, because it makes the approach less likely to overfit.

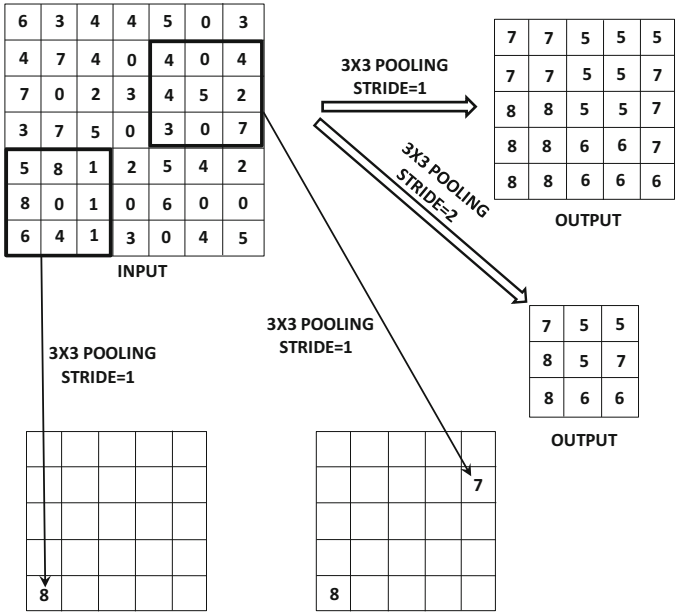


Figure 8.4: An example of a max-pooling of one activation map of size 7×7 with strides of 1 and 2. A stride of 1 creates a 5×5 activation map with heavily repeating elements because of maximization in overlapping regions. A stride of 2 creates a 3×3 activation map with less overlap. Unlike convolution, each activation map is independently processed and therefore the number of output activation maps is exactly equal to the number of input activation maps.

Other types of pooling (like average-pooling) are possible but rarely used. In the earliest convolutional network, referred to as *LeNet-5*, a variant of average pooling was used and was referred[2] to as *subsampling*. In general, max-pooling remains more popular than average pooling. The max-pooling layers are interleaved with the convolutional/ReLU layers, although the former typically occurs much less frequently in deep architectures. This is because pooling drastically reduces the spatial size of the feature map, and only a few pooling operations are required to reduce the spatial map to a small constant size.

It is common to use pooling with 2×2 filters and a stride of 2, when it is desired to reduce the spatial footprint of the activation maps. Pooling results in (some) invariance to translation because shifting the image slightly does not change the activation map significantly. This property is referred to as *translation invariance*. The idea is that similar images often have very different relative locations of the distinctive shapes within them, and translation invariance helps in being able to classify such images in a similar way. For example, one should be able to classify a bird as a bird, irrespective of where it occurs in the image.

Another important purpose of pooling is that it increases the size of the receptive field while reducing the spatial footprint of the layer because of the use of strides larger than 1. Increased sizes of receptive fields are needed to be able to capture larger regions of the image within a complex feature in later layers. Most of the rapid reductions in spatial footprints of the layers (and corresponding increases in receptive fields of the features) are caused by the pooling operations. Convolutions increase the receptive field only gently unless the stride is larger than 1. In recent years, it has been suggested that pooling is not always necessary. One can design a network with only convolutional and ReLU operations, and obtain the expansion of the receptive field by using larger strides within the convolutional operations [184, 466]. Therefore, there is an emerging trend in recent years to get rid of the max-pooling layers altogether. However, this trend has not been fully established and validated, as of the writing of this book. There seem to be at least some arguments in favor of max-pooling. Max-pooling introduces nonlinearity and a greater amount of translation invariance, as compared to strided convolutions. Although nonlinearity can be achieved with the ReLU activation function, the key point is that the effects of max-pooling cannot be exactly replicated by strided convolutions either. At the very least, the two operations are not fully interchangeable.

8.2.6 Fully Connected Layers

Each feature in the final spatial layer is connected to each hidden state in the first fully connected layer. This layer functions in exactly the same way as a traditional feed-forward network. In most cases, one might use more than one fully connected layer to increase the power of the computations towards the end. The connections among these layers are exactly structured like a traditional feed-forward network. Since the fully connected layers are densely connected, the vast majority of parameters lie in the fully connected layers. For example, if each of two fully connected layers has 4096 hidden units, then the connections between them have more than 16 million weights. Similarly, the connections from the last spatial layer to the first fully connected layer will have a large number of parameters. Even though the convolutional layers have a larger number of *activations* (and a larger memory footprint), the fully connected layers often have a larger number of *connections* (and parameter footprint). The reason that activations contribute to the memory footprint

[2]In recent years, subsampling also refers to other operations that reduce the spatial footprint. Therefore, there is some difference between the classical usage of this term and modern usage.

more significantly is that the number of activations are multiplied by mini-batch size while tracking variables in the forward and backward passes of backpropagation. These trade-offs are useful to keep in mind while choosing neural-network design based on specific types of resource constraints (e.g., data versus memory availability). It is noteworthy that the nature of the fully-connected layer can be sensitive to the application at hand. For example, the nature of the fully-connected layer for a classification application would be somewhat different from the case of a segmentation application. The aforementioned discussion is for the most common use-case of a classification application.

The output layer of a convolutional neural network is designed in an application-specific way. In the following, we will consider the representative application of classification. In such a case, the output layer is fully connected to every neuron in the penultimate layer, and has a weight associated with it. One might use the logistic, softmax, or linear activation depending on the nature of the application (e.g., classification or regression).

One alternative to using fully connected layers is to use average pooling across the whole spatial area of the final set of activation maps to create a single value. Therefore, the number of features created in the final spatial layer will be exactly equal to the number of filters. In this scenario, if the final activation maps are of size $7 \times 7 \times 256$, then 256 features will be created. Each feature will be the result of aggregating 49 values. This type of approach greatly reduces the parameter footprint of the fully connected layers, and it has some advantages in terms of generalizability. This approach was used in *GoogLeNet* [485]. In some applications like image segmentation, each pixel is associated with a class label, and one does not use fully connected layers. Fully convolutional networks with 1×1 convolutions are used in order to create an output spatial map.

8.2.7 The Interleaving Between Layers

The convolution, pooling, and ReLU layers are typically interleaved in a neural network in order to increase the expressive power of the network. The ReLU layers often follow the convolutional layers, just as a nonlinear activation function typically follows the linear dot product in traditional neural networks. Therefore, the convolutional and ReLU layers are typically stuck together one after the other. Some pictorial illustrations of neural architectures like *AlexNet* [255] do not explicitly show the ReLU layers because they are assumed to be always stuck to the end of the linear convolutional layers. After two or three sets of convolutional-ReLU combinations, one might have a max-pooling layer. Examples of this basic pattern are as follows:

CRCRP

CRCRCRP

Here, the convolutional layer is denoted by C, the ReLU layer is denoted by R, and the max-pooling layer is denoted by P. This entire pattern (including the max-pooling layer) might be repeated a few times in order to create a deep neural network. For example, if the first pattern above is repeated three times and followed by a fully connected layer (denoted by F), then we have the following neural network:

CRCRPCRCRPCRCRPF

The description above is not complete because one needs to specify the number/size/padding of filters/pooling layers. The pooling layer is the key step that tends to reduce the spatial

footprint of the activation maps because it uses strides that are larger than 1. It is also possible to reduce the spatial footprints with strided convolutions instead of max-pooling. These networks are often quite deep, and it is not uncommon to have convolutional networks with more than 15 layers. Recent architectures also use *skip connections* between layers, which become increasingly important as the depth of the network increases (cf. Section 8.4.5).

LeNet-5

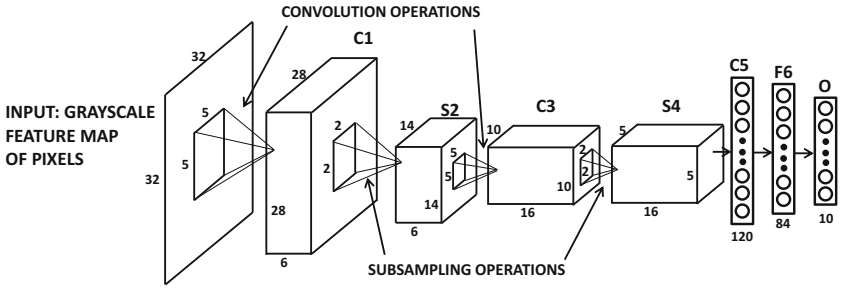
Early networks were quite shallow. An example of one of the earliest neural networks is *LeNet-5* [279]. The input data is in grayscale, and there is only one color channel. The input is assumed to be the ASCII representation of a character. For the purpose of discussion, we will assume that there are ten types of characters (and therefore 10 outputs), although the approach can be used for any number of classes.

The network contained two convolution layers, two pooling layers, and three fully connected layers at the end. However, later layers contain multiple feature maps because of the use of multiple filters in each layer. The architecture of this network is shown in Figure 8.5. The first fully connected layer was also referred to as a convolution layer (labeled as $C5$) in the original work because the ability existed to generalize it to spatial features for larger input maps. However, the specific implementation of *LeNet-5* really used $C5$ as a fully connected layer, because the filter spatial size was the same as the input spatial size. This is why we are counting $C5$ as a fully connected layer in this exposition. It is noteworthy that two versions of *LeNet-5* are shown in Figure 8.5(a) and (b). The upper diagram of Figure 8.5(a) explicitly shows the subsampling layers, which is how the architecture was presented in the original work. However, deeper architectural diagrams like *AlexNet* [255] often do not show the subsampling or max-pooling layers explicitly in order to accommodate the large number of layers. Such a concise architecture for *LeNet-5* is illustrated in Figure 8.5(b). The activation function layers are also not explicitly shown in either figure. In the original work in *LeNet-5*, the sigmoid activation function occurs immediately after the subsampling operations, although this ordering is relatively unusual in recent architectures. In most modern architectures, subsampling is replaced by max-pooling, and the max-pooling layers occur less frequently than the convolution layers. Furthermore, the activations are typically performed immediately after each convolution (rather than after each max-pooling).

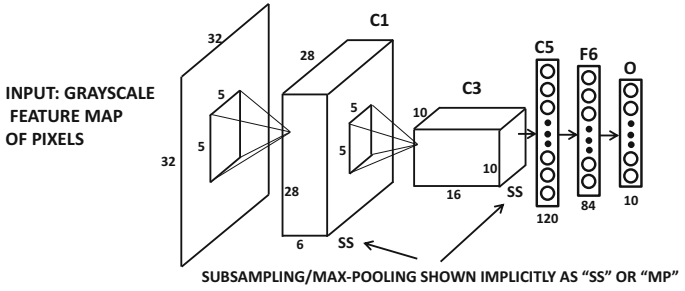
The number of layers in the architecture is often counted in terms of the number of layers with weighted spatial filters and the number of fully connected layers. In other words, subsampling/max-pooling and activation function layers are often not counted separately. The subsampling in *LeNet-5* used 2×2 spatial regions with stride 2. Furthermore, unlike max-pooling, the values were averaged, scaled with a trainable weight and then a bias was added. In modern architectures, the linear scaling and bias addition operations have been dispensed with. The concise architectural representation of Figure 8.5(b) is sometimes confusing to beginners because it is missing details such as the size of the max-pooling/subsampling filters. In fact, there is no unique way of representing these architectural details, and many variations are used by different authors. This chapter will show several such examples in the case studies.

This network is extremely shallow by modern standards; yet the basic principles have not changed since then. The main difference is that the ReLU activation had not appeared at that point, and sigmoid activation was often used in the earlier architectures. Furthermore, the use of average pooling is extremely uncommon today compared to max-pooling. Recent years have seen a move away from both max-pooling and subsampling, with strided convolutions as the preferred choice. *LeNet-5* also used ten radial basis function (RBF)

units in the final layer (cf. Chapter 5), in which the prototype of each unit was compared to its input vector and the squared Euclidean distance between them was output. This is the same as using the negative log-likelihood of the Gaussian distribution represented by that RBF unit. The parameter vectors of the RBF units were chosen by hand, and correspond to a stylized 7×12 bitmap image of the corresponding character class, which were flattened into a $7 \times 12 = 84$-dimensional representation. Note that the size of the penultimate layer is exactly 84 in order to enable the computation of the Euclidean distance between the vector corresponding to that layer and the parameter vector of the RBF unit. The ten outputs in the final layer provide the scores of the classes, and the smallest score among the ten units provides the prediction. This type of use of RBF units is now anachronistic in modern convolutional network design, and one generally tends to work with softmax units with log-likelihood loss on multinomial label outputs. *LeNet-5* was used extensively for character recognition, and was used by many banks to read checks.



(a) Detailed architectural representation



(b) Concise architectural representation

Figure 8.5: LeNet-5: One of the earliest convolutional neural networks.

8.2.8 Local Response Normalization

A trick that is introduced in [255] is that of *local response normalization*, which is always used immediately after the ReLU layer. The use of this trick aids generalization. The basic idea of this normalization approach is inspired from biological principles, and it is intended to create competition among different filters. First, we describe the normalization formula using *all* filters, and then we describe how it is actually computed using only a subset of filters. Consider a situation in which a layer contains N filters, and the activation values of

these N filters at a particular spatial position (x, y) are given by $a_1 \ldots a_N$. Then, each a_i is converted into a normalized value b_i using the following formula:

$$b_i = \frac{a_i}{(k + \alpha \sum_j a_i^2)^\beta} \tag{8.1}$$

The values of the underlying parameters used in [255] are $k = 2$, $\alpha = 10^{-4}$, and $\beta = 0.75$. However, in practice, one does not normalize over all N filters. Rather the filters are ordered arbitrarily up front to define "adjacency" among filters. Then, the normalization is performed over each set of n "adjacent" filters for some parameter n. The value of n used in [255] is 5. Therefore, we have the following formula:

$$b_i = \frac{a_i}{(k + \alpha \sum_{j=i-\lfloor n/2 \rfloor}^{i+\lfloor n/2 \rfloor} a_i^2)^\beta} \tag{8.2}$$

In the above formula, any value of $i - n/2$ that is less than 0 is set to 0, and any value of $i + n/2$ that is greater than N is set to N. The use of this type of normalization is no obsolete, and its discussion has been included here for historical reasons.

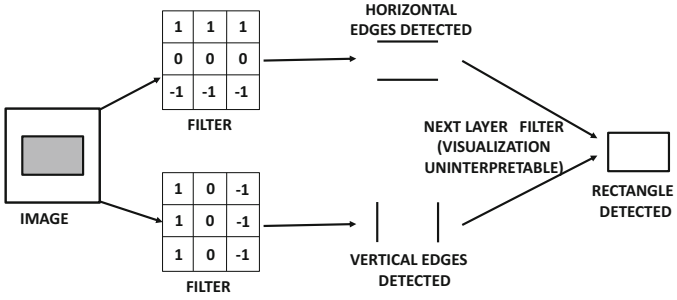


Figure 8.6: Filters detect edges and combine them to create rectangle.

8.2.9 Hierarchical Feature Engineering

It is instructive to examine the activations of the filters created by real-world images in different layers. In Section 8.5, we will discuss a concrete way in which the features extracted in various layers can be visualized. For now, we provide a subjective interpretation. The activations of the filters in the early layers are low-level features like edges, whereas those in later layers put together these low-level features. For example, a mid-level feature might put together edges to create a hexagon, whereas a higher-level feature might put together the mid-level hexagons to create a honeycomb. It is fairly easy to see why a low-level filter might detect edges. Consider a situation in which the color of the image changes along an edge. As a result, the difference between neighboring pixel values will be non-zero only across the edge. This can be achieved by choosing the appropriate weights in the corresponding low-level filter. Note that the filter to detect a horizontal edge will not be the same as that to detect a vertical edge. This brings us back to Hubel and Weisel's experiments in which different neurons in the cat's visual cortex were activated by different edges. Examples of filters detecting horizontal and vertical edges are illustrated in Figure 8.6. The next layer filter works on the hidden features and therefore it is harder to interpret. Nevertheless, the next layer filter is able to detect a rectangle by combining the horizontal and vertical edges.

In a later section, we will show visualizations of how smaller portions of real-world image activate different hidden features, much like the biological model of Hubel and Wiesel in which different shapes seem to activate different neurons. Therefore, the power of convolutional neural networks rests in the ability to put together these primitive shapes into more complex shapes layer by layer. Note that it is impossible for the first convolution layer to learn any feature that is larger than $F_1 \times F_1$ pixels, where the value of F_1 is typically a small number like 3 or 5. However, the next convolution layer will be able to put together many of these patches together to create a feature from an area of the image that is larger. The primitive features learned in earlier layers are put together in a semantically coherent way to learn increasingly complex and interpretable visual features. The choice of learned features is affected by how backpropagation adapts the features to the needs of the loss function at hand. For example, if an application is training to classify images as cars, the approach might learn to put together arcs to create a circle, and then it might put together circles with other shapes to create a car wheel. All this is enabled by the hierarchical features of a deep network.

Recent *ImageNet* competitions have demonstrated that much of the power in image recognition lies in increased depth of the network. Not having enough layers effectively prevents the network from learning the hierarchical regularities in the image that are combined to create its semantically relevant components. Another important observation is that the nature of the features learned will be sensitive to the specific data set at hand. For example, the features learned to recognize trucks will be different from those learned to recognize carrots. However, some data sets (like *ImageNet*) are diverse enough that the features learned by training on these data sets have general-purpose significance across many applications.

8.3 Training a Convolutional Network

The process of training a convolutional neural network uses the backpropagation algorithm. There are primarily three types of layers, corresponding to the convolution, ReLU, and max-pooling layers. We will separately describe the backpropagation algorithm through each of these layers. The ReLU is relatively straightforward to backpropagate through because it is no different than a traditional neural network. For max-pooling with no overlap between pools, one only needs to identify which unit is the maximum value in a pool (with ties broken arbitrarily or divided proportionally). The partial derivative of the loss with respect to the pooled state flows back to the unit with maximum value. All entries other than the maximum entry in the grid will be assigned a value of 0. Note that the backpropagation through a maximization operation is also described in Table 3.1 of Chapter 3. For cases in which the pools are overlapping, let $P_1 \ldots P_r$ be the pools in which the unit h is involved, with corresponding activations $h_1 \ldots h_r$ in the next layer. If h is the maximum value in pool P_i (and therefore $h_i = h$), then the gradient of the loss with respect to h_i flows back to h (with ties broken arbitrarily or divided proportionally). The contributions of the different overlapping pools (from $h_1 \ldots h_r$ in the next layer) are added in order to compute the gradient with respect to the unit h. Therefore, the backpropagation through the maximization and the ReLU operations are not very different from those in traditional neural networks.

8.3.1 Backpropagating Through Convolutions

The backpropagation through convolutions is also not very different from the backpropagation with linear transformations (i.e., matrix multiplications) in a feed-forward network. This point of view will become particularly clear when we present convolutions as a form of matrix multiplication. Just as backpropagation in feed-forward networks from layer $(i+1)$ to layer i is achieved by multiplying the error derivatives with respect to layer $(i+1)$ with the transpose of the forward propagation matrix between layers i and $(i+1)$ (cf. Table 3.1 of Chapter 3), backpropagation in convolutional networks can also be seen as a form of transposed convolution.

First, we describe a simple element-wise approach to backpropagation. Assume that the loss gradients of the cells in layer $(i+1)$ have already been computed. The loss derivative with respect to a cell in layer $(i+1)$ is defined as the partial derivative of the loss function with respect to the hidden variable in that cell. Convolutions multiply the activations in layer i with filter elements to create elements in the next layer. Therefore, a cell in layer $(i+1)$ receives aggregated contributions from a 3-dimensional volume of elements in the previous layer of filter size $F_i \times F_i \times d_i$. At the same time, a cell c in layer i contributes to multiple elements (denoted by set S_c) in layer $(i+1)$, although the number of elements to which it contributes depends on the depth of the next layer and the stride. Identifying this "forward set" is the key to the backpropagation. A key point is that the cell c contributes to each element in S_c in an additive way after multiplying the activation of cell c with a filter element. Therefore, backpropagation simply needs to multiply the loss derivative of each element in S_c with respect to the corresponding filter element and aggregate in the backwards direction at c. For any particular cell c in layer i, the following pseudocode can be used to backpropagate the existing derivatives in layer-$(i+1)$ to cell c in layer-i:

Identify all cells S_c in layer $(i+1)$ to which cell c in layer i contributes;
For each cell $r \in S_c$, let δ_r be its (already backpropagated) loss-derivative with respect to cell r;
For each cell $r \in S_c$, let w_r be weight of filter element used for contributing from cell c to r;
$\delta_c = \sum_{r \in S_c} \delta_r \cdot w_r$;

After the loss gradients have been computed, the values are multiplied with those of the hidden units of the $(i-1)$th layer to obtain the gradients with respect to the weights between the $(i-1)$th and ith layer. In other words, the hidden value at one end point of a weight is multiplied with the loss gradient at the other end in order to obtain the partial derivative with respect to the weight. However, this computation assumes that all weights are distinct, whereas the weights in the filter are shared across the entire spatial extent of the layer. Therefore, one has to be careful to account for shared weights, and sum up the partial derivatives of all copies of a shared weight. In other words, we first pretend that the filter used in each position is distinct in order to compute the partial derivative with respect to each copy of the shared weight, and then add up the partial derivatives of the loss with respect to all copies of a particular weight.

Note that the approach above uses simple linear accumulation of gradients like traditional backpropagation. However, one has to be slightly careful in terms of keeping track of the cells that influence other cells in the next layer. One can implement backpropagation with the help of tensor multiplication operations, which can further be simplified into simple matrix multiplications of derived matrices from these tensors. This point of view will be discussed in the next two sections because it provides many insights on how many aspects of feedforward networks can be generalized to convolutional neural networks.

8.3.2 Backpropagation as Convolution with Inverted/Transposed Filter

In conventional neural networks, a backpropagation operation is performed by multiplying a vector of gradients at layer $(q+1)$ with the transposed weight matrix between the layers q and $(q+1)$ in order to obtain the vector of gradients at layer q (cf. Table 3.1). In convolution neural networks, the backpropagated derivatives are also associated with spatial positions in the layers. Is there an analogous convolution we can apply to the spatial footprint of backpropagated derivatives in a layer to obtain those of the previous layer? It turns out that this is indeed possible.

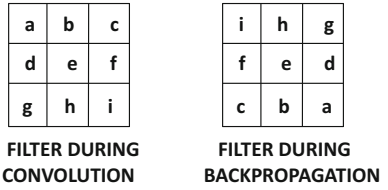


Figure 8.7: The inverse of a kernel for backpropagation

Let us consider the case in which the activations in layer q are convolved with a filter to create those in layer $(q+1)$. For simplicity, consider the case in which depth d_q of the input layer and the depth d_{q+1} of the output layer are both 1; furthermore, we use convolutions with stride 1. In such a case, the convolution filter is inverted both horizontally and vertically for backpropagation. An example of such an inverted filter is illustrated in Figure 8.7. The intuitive reason for this inversion is that the *filter* is "moved around" the spatial area of the input volume to perform the dot product, whereas the backpropagated derivatives are with respect to the *input volume*, whose relative movement with respect to the filter is the opposite of the filter movement during convolutions. Note that the entry in the extreme upper-left of the convolution filter might not even contribute to the extreme upper-left entry in the output volume (because of padding), but it will almost always contribute to the extreme lower-right entry of the output volume. This is consistent with the inversion of the filter. The backpropagated derivative set of the $(q+1)$th layer is convolved with this inverted filter to obtain the backpropagated derivative set of the qth layer. How are the paddings of the forward convolution and backward convolution related? For a stride of 1, the sum of the paddings during forward propagation and backward propagation is $F_q - 1$, where F_q is the side length of the filter for qth layer.

Now consider the case in which the depths d_q and d_{q+1} are no longer 1, but are arbitrary values. In this case, an additional tensor transposition needs to occur. The weight of the (i, j, k)th position of the pth filter in the qth layer is $\mathcal{W} = [w_{ijk}^{(p,q)}]$. Note that i and j refer to spatial positions, whereas k refers to the depth-centric position of the weight. In such a

case, let the 5-dimensional tensor corresponding to the backpropagation filters from layer $q+1$ to layer q be denoted by $\mathcal{U} = [u_{ijk}^{(p,q+1)}]$. Then, the entries of this tensor are as follows:

$$u_{rsp}^{(k,q+1)} = w_{ijk}^{(p,q)} \tag{8.3}$$

Here, we have $r = F_q - i + 1$ and $s = F_q - j + 1$. Note that the index p of the filter identifier and depth k within a filter have been interchanged between $\mathcal{W}$ and $\mathcal{U}$ in Equation 8.3. This is a tensor-centric transposition.

In order to understand the transposition above, consider a situation in which we use 20 filters on the 3-channel RGB volume in order to create an output volume of depth 20. While backpropagating, we will need to take a *gradient volume* of depth 20 and transform to a *gradient volume* of depth 3. Therefore, we need to create 3 filters for backpropagation, each of which is for the red, green, and blue colors. We pull out the 20 spatial slices from the 20 filters that are applied to the red color, invert them using the approach of Figure 8.7, and then create a single 20-depth filter for backpropagating gradients with respect to the red slice. Similar approaches are used for the green and blue slices. The transposition and inversion in Equation 8.3 correspond to these operations.

8.3.3 Convolution/Backpropagation as Matrix Multiplications

It is helpful to view convolution as a matrix multiplication because it helps us define various related notions such as *transposed convolution*, *deconvolution*, and *fractional convolution*. These concepts are helpful not just in understanding backpropagation, but also in developing the machinery necessary for convolutional autoencoders. In traditional feed-forward networks, matrices that are used to transform hidden states in the forward phase are transposed in the backwards phase (cf. Table 3.1) in order to backpropagate partial derivatives across layers. Similarly, the matrices used in encoders are often transposed in the decoders when working with autoencoders in traditional settings. Although the spatial structure of the convolutional neural network does mask the nature of the underlying matrix multiplication, one can "flatten" this spatial structure to perform the multiplication and reshape back to a spatial structure using the known spatial positions of the elements of the flattened matrix. This somewhat indirect approach is helpful in understanding the fact that the convolution operation is similar to the matrix multiplication in feed-forward networks at a very fundamental level. Furthermore, real-world implementations of convolution are often accomplished with matrix multiplication.

For simplicity, let us first consider the case in which the qth layer and the corresponding filter used for convolution both have unit depth. Furthermore, assume that we are using a stride of 1 with zero padding. Therefore, the input dimensions are $L_q \times B_q \times 1$, and the output dimensions are $(L_q - F_q + 1) \times (B_q - F_q + 1) \times 1$. In the common setting in which the spatial dimensions are square (i.e., $L_q = B_q$), one can assume that the spatial dimensions of the input $A_I = L_q \times L_q$ and the spatial dimensions of the output are $A_O = (L_q - F_q + 1) \times (L_q - F_q + 1)$. Here, A_I and A_O are the spatial areas of the input and output matrices, respectively. The input can be representing by flatting the area A_I into an A_I-dimensional column vector in which the rows of the spatial area are concatenated from top to bottom. This vector is denoted by $\overline{f}$. An example of a case in which we use a 2×2 filter on a 3×3 input is shown in Figure 8.8. Therefore, the output is of size 2×2, and we have $A_I = 3 \times 3 = 9$, and $A_O = 2 \times 2 = 4$. The 9-dimensional column vector for the 3×3 input is shown in Figure 8.8. A sparse matrix C is defined in lieu of the filter, which is the

key in representing the convolution as a matrix multiplication. A matrix of size $A_O \times A_I$ is defined in which each row corresponds to the convolution at one of the A_O convolution locations. These rows are associated with the spatial location of the top-left corner of the convolution region in the input matrix from which they were derived. The value of each entry in the row corresponds to one of the A_I positions in the input matrix, but this value is 0, if that input position is not involved in the convolution for that row. Otherwise, the value is set to the corresponding value of the filter, which is used for multiplication. The ordering of the entries in a row is based on the same spatially sensitive ordering of the input matrix locations as was used to flatten the input matrix into an A_I-dimensional vector. Since the filter size is usually much smaller than the input size, most of the entries in the matrix C are 0s, and each entry of the filter occurs in every row of C. Therefore, every entry in the filter is repeated A_O times in C, because it is used for A_O multiplications.

An example of a 4×9 matrix C is shown in Figure 8.8. Subsequent multiplication of C with $\overline{f}$ yields an A_O-dimensional vector. The corresponding 4-dimensional vector is shown in Figure 8.8. Since each of the A_O rows of C is associated with a spatial location, these locations are inherited by $C\overline{f}$. These spatial locations are used to reshape $C\overline{f}$ to a spatial matrix. The reshaping of the 4-dimensional vector to a 2×2 matrix is also shown in Figure 8.8.

This particular exposition uses the simplified case with a depth of 1. In the event that the depth is larger than 1, the same approach is applied for each 2-dimensional slice, and the results are added. In other words, we aggregate $\sum_p C_p \overline{f}_p$ over the various slice indices p and then the results are re-shaped into a 2-dimensional matrix. This approach amounts to a *tensor* multiplication, which is a straightforward generalization of a matrix multiplication. The tensor multiplication approach is how convolution is actually implemented in practice. In general, one will have multiple filters, which correspond to multiple output maps. In such a case, the kth filter will be converted into the sparsified matrix $C_{p,k}$, and the kth feature map of the output volume will be $\sum_p C_{p,k} \overline{f}_p$.

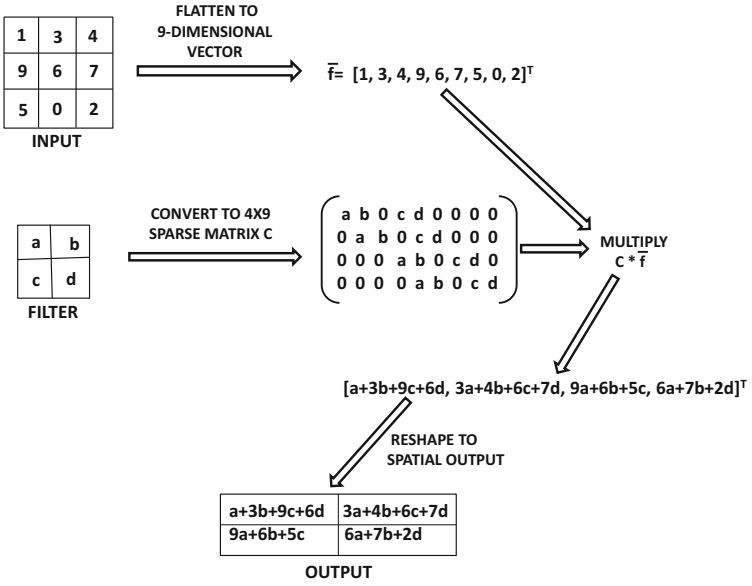


Figure 8.8: Convolution as matrix multiplication

The matrix-centric approach is very useful for performing backpropagation because one can also propagate gradients backwards by using the same approach in the backwards direction, except that the *transposed* matrix C^T is used for multiplication with the flattened vector version of a 2-dimensional slice of the output gradient. Note that the flattening of a gradient with respect to a spatial map can be done in a similar way as the flattened vector $\overline{f}$ is created in the forward phase. Consider the simple case in which both the input and output volumes have a depth of 1. If $\overline{g}$ is the flattened vector gradient of the loss with respect to the output spatial map, then the flattened gradient with respect to the input spatial map is obtained as $C^T\overline{g}$. This approach is consistent with the approach used in feed-forward networks, in which the transpose of the forward matrix is used in backpropagation. The above result is for the simple case when both input and output volumes have depth of 1. What happens in the general case? When the depth of the output volume is $d > 1$, the gradients with respect to the output maps are denoted by $\overline{g}_1 \ldots \overline{g}_d$. The corresponding gradient with respect to the features in the pth spatial slice of the input volume is given by $\sum_{k=1}^{d} C_{p,k}^T \overline{g}_k$. Here, the matrix $C_{p,k}$ is obtained by converting the pth spatial slice of the kth filter into the sparsified matrix as discussed above. This approach is a consequence of Equation 8.3. This type of transposed convolution is also useful for the deconvolution operation in convolution autoencoders, which will be discussed later in this chapter (cf. Section 8.5).

8.3.4 Data Augmentation

A common trick to reduce overfitting in convolutional neural networks is the idea of *data augmentation*. In data augmentation, new training examples are generated by using transformations on the original examples. This idea was briefly discussed in Chapter 4, although it works better in some domains than others. Image processing is one domain to which data augmentation is very well suited. This is because many transformations such as translation, rotation, patch extraction, and reflection do not fundamentally change the properties of the object in an image. However, they do increase the generalization power of the data set when trained with the augmented data set. For example, if a data set is trained with mirror images and reflected versions of all the bananas in it, then the model is able to better recognize bananas in different orientations.

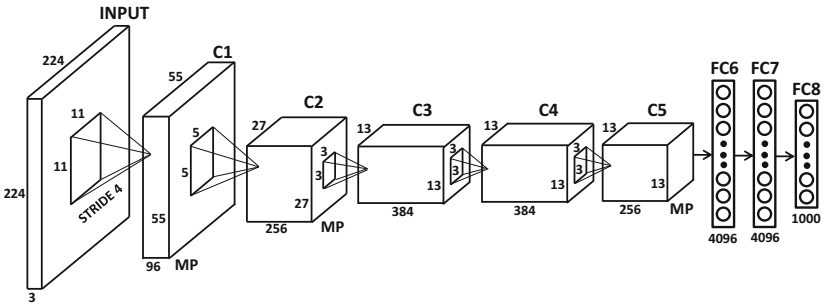
Many of these forms of data augmentation require very little computation, and therefore the augmented images do not need to be explicitly generated up front. Rather, they can be created at training time, when an image is being processed. For example, while processing an image of a banana, it can be reflected into a modified banana at training time. Similarly, the same banana might be represented in somewhat different color intensities in different images, and therefore it might be helpful to create representations of the same image in different color intensities. In many cases, creating the training data set using image patches can be helpful. An important neural network that rekindled interest in deep learning by winning the ILSVRC challenge was *AlexNet*. This network was trained by extracting $224 \times 224 \times 3$ patches from the images, which also defined the input sizes for the networks. The neural networks, which were entered into the ILSVRC contest in subsequent years, used a similar methodology of extracting patches.

Although most data augmentation methods are quite efficient, some forms of transformation that use principal component analysis (PCA) can be more expensive. PCA is used in order to change the color intensity of an image. If the computational costs are high, it becomes important to extract the images up front and store them. The basic idea here is to use the 3×3 covariance matrix of each pixel value and compute the principal components. Then, Gaussian noise is added to each principal component with zero mean and variance of 0.01. This noise is fixed over all the pixels of a particular image. The approach is dependent on the fact that object identity is invariant to color intensity and illumination. It is reported in [255] that data set augmentation reduces error rate by 1%.

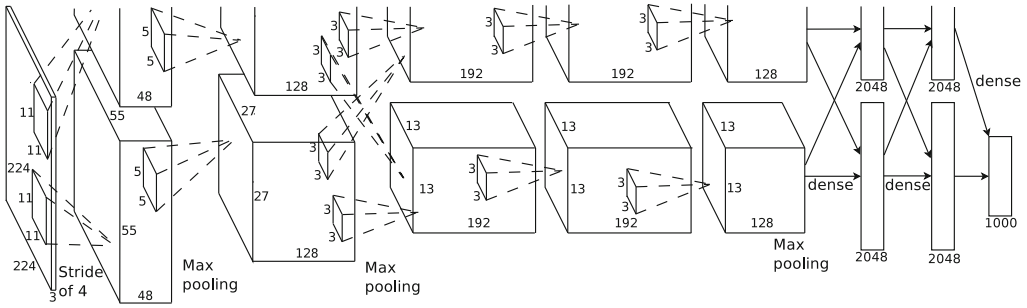
One must be careful not to apply data augmentation blindly without regard to the data set and application at hand. For example, applying rotations and reflections on the MNIST data set [281] of handwritten digits is a bad idea because the digits in the data set are all presented in a similar orientation. Furthermore, the mirror image of an asymmetric digit is not a valid digit, and a rotation of a '6' is a '9.' The key point in deciding what types of data augmentation are reasonable is to account for the natural distribution of images in the full data set, as well as the effect of a specific type of data set augmentation on the class labels.

8.4 Case Studies of Convolutional Architectures

In the following, we provide some case studies of convolutional architectures. These case studies were derived from successful entries to the ILSVRC competition in recent years. These are instructive because they provide an understanding of the important factors in neural network design that can make these networks work well. Even though recent years have seen some changes in architectural design (like ReLU activation), it is striking how similar the modern architectures are to the basic design of *LeNet-5*. The main changes from *LeNet-5* to modern architectures are in terms of the explosion of depth, the use of ReLU activation, and the training efficiency enabled by modern hardware/optimization enhancements. Modern architectures are deeper, and they use a variety of computational, architectural, and hardware tricks to efficiently train these networks with large amounts of data. Hardware advancements should not be underestimated; modern GPU-based platforms are 10,000 times faster than the (similarly priced) systems available at the time *LeNet-5* was proposed. Even on these modern platforms, it often takes a week to train a convolutional neural network that is accurate enough to be competitive at ILSVRC. The hardware, data-centric, and algorithmic enhancements are connected to some extent. It is difficult to try new algorithmic tricks if enough data and computational power is not available to experiment with complex/deeper models in a reasonable amount of time. Therefore, the recent revolution in deep convolutional networks could not have been possible, had it not been for the large amounts of data and increased computational power available today.



(a) Without GPU partitioning



(b) With GPU partitioning (original architecture)

Figure 8.9: The *AlexNet* architecture. The ReLU activations follow each convolution layer, and are not explicitly shown. Note that the max-pooling layers are labeled as MP, and they follow only a subset of the convolution-ReLU combination layers. The architectural diagram in (b) is from [A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. *NIPS Conference*, pp. 1097–1105. 2012.] ©2012 A. Krizhevsky, I. Sutskever, and G. Hinton.

In the following sections, we provide an overview of some of the well-known models that are often used for designing training algorithms for image classification. It is worth mentioning that some of these models are available as pretrained models over *ImageNet*, and the resulting features can be used for applications beyond classification. Such an approach is a form of transfer learning, which is discussed later in this section.

8.4.1 AlexNet

AlexNet was the winner of the 2012 ILSVRC competition. The architecture of *AlexNet* is shown in Figure 8.9(a). It is worth mentioning that there were two parallel pipelines of processing in the original architecture, which are not shown in Figure 8.9(a). These two pipelines are caused by two GPUs working together to build the training model with a faster speed and memory sharing. The network was originally trained on a GTX 580 GPU with 3 GB of memory, and it was impossible to fit the intermediate computations in this amount of space. Therefore, the network was partitioned across two GPUs. The original architecture is shown in Figure 8.9(b), in which the work is partitioned into two GPUs. We also show the architecture without the changes caused by the GPUs, so that it can be more easily compared with other convolutional neural network architectures discussed in this chapter. It is noteworthy that the GPUs are inter-connected in only a subset of the

layers in Figure 8.9(b), which leads to some differences between Figure 8.9(a) and (b) in terms of the actual model constructed. Specifically, the GPU-partitioned architecture has fewer weights because not all layers have interconnections. Dropping some of the interconnections reduces the communication time between the processors and therefore helps in efficiency.

AlexNet starts with $224 \times 224 \times 3$ images and uses 96 filters of size $11 \times 11 \times 3$ in the first layer. A stride of 4 is used. This results in a first layer of size $55 \times 55 \times 96$. After the first layer has been computed, a max-pooling layer is used. This layer is denoted by 'MP' in Figure 8.9(a). Note that the architecture of Figure 8.9(a) is a simplified version of the architecture shown in Figure 8.9(b), which explicitly shows the two parallel pipelines. For example, Figure 8.9(b) shows a depth of the first convolution layer of only 48, because the 96 feature maps are divided among the GPUs for parallelization. On the other hand, Figure 8.9(a) does not assume the use of GPUs, and therefore the width is explicitly shown as 96. The ReLU activation function was applied after each convolutional layer, which was followed by response normalization and max-pooling. Although max-pooling has been annotated in the figure, it has not been assigned a block in the architecture. Furthermore, the ReLU and response normalization layers are not explicitly shown in the figure. These types of concise representations are common in pictorial depictions of neural architectures.

The second convolutional layer uses the response-normalized and pooled output of the first convolutional layer and filters it with 256 filters of size $5 \times 5 \times 96$. No intervening pooling or normalization layers are present in the third, fourth, or fifth convolutional layers. The sizes of the filters of the third, fourth, and fifth convolutional layers are $3 \times 3 \times 256$ (with 384 filters), $3 \times 3 \times 384$ (with 384 filters), and $3 \times 3 \times 384$ (with 256 filters). All max-pooling layers used 3×3 filters at stride 2. Therefore, there was some overlap among the pools. The fully connected layers have 4096 neurons. The final set of 4096 activations can be treated as a 4096-dimensional representation of the image. The final layer of *AlexNet* uses a 1000-way softmax in order to perform the classification. It is noteworthy that the final layer of 4096 activations (labeled by FC7 in Figure 8.9(b)) is often used to create a flat 4096 dimensional representation of an image for applications beyond classification. One can extract these features for any out-of-sample image by simply passing it through the trained neural network. These features often generalize well to other data sets and other tasks. Such features are referred to as FC7 features. In fact, the use of the extracted features from the penultimate layer as FC7 was popularized after *AlexNet*, even though the approach was known much earlier. As a result, such extracted features from the penultimate layer of a convolutional neural network are often referred to as *FC7 features*, irrespective of the number of layers in that network. It is noteworthy that the number of feature maps in middle layers is far larger than the initial depth of the volume in the input layer (which is only 3 corresponding to RGB colors) although their spatial dimensions are smaller. This is because the initial depth only contains the RGB color components, whereas the later layers capture different types of semantic features in the features maps.

Many design choices used in the architecture became standard in later architectures. A specific example is the use of ReLU activation in the architecture (instead of sigmoid or tanh units). The choice of the activation function in most convolutional neural networks today is almost exclusively focused on the ReLU, although this was not the case before *AlexNet*. Some other training tricks were known at the time, but their use in *AlexNet* popularized them. One example was the use of data augmentation, which turned out to be very useful in improving accuracy. *AlexNet* also underlined the importance of using specialized hardware like GPUs for training on such large data sets. Dropout was used with L_2-weight decay in order to improve generalization. The use of Dropout is common in virtually all types of architectures today because it provides an additional booster in most cases. The use of local response normalization was eventually discarded by later architectures.

We also briefly mention the parameter choices used in *AlexNet*. The interested reader can find the full code and parameter files of *AlexNet* at [584]. L_2-regularization was used with a parameter of 5×10^{-4}. *Dropout* was used by sampling units at a probability of 0.5. Momentum-based (mini-batch) stochastic gradient descent was used for training *AlexNet* with parameter value of 0.8. The batch-size was 128. The learning rate was 0.01, although it was eventually reduced a couple of times as the method began to converge. Even with the use of the GPU, the training time of *AlexNet* was of the order of a week.

The final top-5 error rate, which was defined as the fraction of cases in which the correct image was not included in the top-5 images, was about 15.4%. This error rate[3] was in comparison with the previous winners with an error rate of more than 25%. The gap with respect to the second-best performer in the contest was also similar. The use of single convolutional network provided a top-5 error rate of 18.2%, although using an ensemble of seven models provided the winning error-rate of 15.4%. Note that these types of ensemble-based tricks provide a consistent improvement of between 2% and 3% with most architectures. Furthermore, since the executions of most ensemble methods are embarrassingly parallelizable, it is relatively easy to perform them, as long as sufficient hardware resources are available. *AlexNet* is considered a fundamental advancement within the field of computer vision because of the large margin with which it won the ILSVRC contest. This success rekindled interest in deep learning in general, and convolutional neural networks in particular.

8.4.2 ZFNet

A variant of *ZFNet* [556] was the winner of the ILSVRC competition in 2013. Its architecture was heavily based on *AlexNet*, although some changes were made to further improve the accuracy. Most of these changes were associated with differences in hyperparameter choices, and therefore *ZFNet* is not very different from *AlexNet* at a fundamental level. One change from *AlexNet* to *ZFNet* was that the initial filters of size $11 \times 11 \times 3$ were changed to $7 \times 7 \times 3$. Instead of strides of 4, strides of 2 were used. The second layer used 5×5 filters at stride 2 as well. As in *AlexNet*, there are three max-pooling layers, and the same sizes of max-pooling filters were used. However, the first pair of max-pooling layers were performed after the first and second convolutions (rather than the second and third convolutions). As a result, the spatial footprint of the third layer changed to 13×13 rather than 27×27, although all other spatial footprints remained unchanged from *AlexNet*. The sizes of various layers in *AlexNet* and *ZFNet* are listed in Table 8.1.

The third, fourth, and fifth convolutional layers use a larger number of filters in *ZFNet* as compared to *AlexNet*. The number of filters in these layers were changed from $(384, 384, 256)$

[3]The top-5 error rate makes more sense in image data where a single image might contain objects of multiple classes. Throughout this chapter, we use the term "error rate" to refer to the top-5 error rate.

Table 8.1: Comparison of *AlexNet* and *ZFNet*

	AlexNet	*ZFNet*
Volume: Operations:	$224 \times 224 \times 3$ Conv 11×11 (stride 4)	$224 \times 224 \times 3$ Conv 7×7 (stride 2), MP
Volume: Operations:	$55 \times 55 \times 96$ Conv 5×5, MP	$55 \times 55 \times 96$ Conv 5×5 (stride 2), MP
Volume: Operations:	$27 \times 27 \times 256$ Conv 3×3, MP	$13 \times 13 \times 256$ Conv 3×3
Volume: Operations:	$13 \times 13 \times 384$ Conv 3×3	$13 \times 13 \times 512$ Conv 3×3
Volume: Operations:	$13 \times 13 \times 384$ Conv 3×3	$13 \times 13 \times 1024$ Conv 3×3
Volume: Operations:	$13 \times 13 \times 256$ MP, Fully connect	$13 \times 13 \times 512$ MP, Fully connect
FC6: Operations:	4096 Fully connect	4096 Fully connect
FC7: Operations:	4096 Fully connect	4096 Fully connect
FC8: Operations:	1000 Softmax	1000 Softmax

to $(512, 1024, 512)$. As a result, the spatial footprints of *AlexNet* and *ZFNet* are the same in most layers, although the depths are different in the final three convolutional layers with similar spatial footprints. From an overall perspective, *ZFNet* used similar principles to *AlexNet*, and the main gains were obtained by changing the architectural parameters of *AlexNet*. This architecture reduced the top-5 error rate to 14.8% from 15.4%, and further increases in width/depth from the same author(s) reduced the error to 11.1%. Since most of the differences between *AlexNet* and *ZFNet* were those of minor design choices, this emphasizes the fact that small details are important when working with deep learning algorithms. Thus, extensive experimentation with neural architectures are sometimes important in order to obtain the best performance. The architecture of *ZfNet* was made wider and deeper, and the results were submitted to ILSVRC in 2013 under the name *Clarifai*, which was a company[4] founded by the first author of [556]. The difference[5] between *Clarifai* and *ZFNet* was one of width/depth of the network, although exact details of these differences are not available. This entry was the winning entry of the ILSVRC competition in 2013. Refer to [556] for details and a pictorial illustration of the architecture.

8.4.3 VGG

VGG [454] further emphasized the developing trend in terms of increased depth of networks. The tested networks were designed with various configurations with sizes between 11 and 19 layers, although the best-performing versions had 16 or more layers. *VGG* was a top-performing entry on ISLVRC in 2014, but it was not the winner. The winner was *GoogLeNet*, which had a top-5 error rate of 6.7% in comparison with the top-5 error rate of 7.3% for *VGG*. Nevertheless, *VGG* was important because it illustrated several important design principles that eventually became standard in future architectures.

[4] http://www.clarifai.com

[5] Personal communication from Matthew Zeiler.

An important innovation of *VGG* is that it reduced filter sizes but increased depth. It is important to understand that *reduced filter size necessitates increased depth*. This is because a small filter can capture only a small region of the image unless the network is deep. For example, a single feature that is a result of three sequential convolutions of size 3×3 will capture a region in the input of size 7×7. Note that using a single 7×7 filter directly on the input data will also capture the visual properties of a 7×7 input region. In the first case, we are using $3 \times 3 \times 3 = 27$ parameters, whereas we are using $7 \times 7 \times 1 = 49$ parameters in the second case. Therefore, the parameter footprint is smaller in the case when three sequential convolutions are used. However, three successive convolutions can often capture more interesting and complex features than a single convolution, and the resulting activations with a single convolution will look like primitive edge features. Therefore, the network with 7×7 filters will be unable to capture sophisticated shapes in smaller regions.

In general, greater depth forces more nonlinearity and greater regularization. A deeper network will have more nonlinearity because of the presence of more ReLU layers, and more regularization because the increased depth forces a structure on the layers through the use of repeated composition of convolutions. As discussed above, architectures with greater depth and reduced filter size require fewer parameters. This occurs in part because the number of parameters in each layer is given by the square of the filter size, whereas the number of parameters depend linearly on the depth. Therefore, one can drastically reduce the number of parameters by using smaller filter sizes, and instead "spend" these parameters by using increased depth. Increased depth also allows the use of a greater number of nonlinear activations, which increases the discriminative power of the model. Therefore *VGG* always uses filters with spatial footprint 3×3 and pooling of size 2×2. The convolution was done with stride 1, and a padding of 1 was used. The pooling was done at stride 2. Using a 3×3 filter with a padding of 1 maintains the spatial footprint of the output volume, although pooling always compresses the spatial footprint. Therefore, the pooling was done on non-overlapping spatial regions (unlike the previous two architectures), and always reduced the spatial footprint (i.e., both height and width) by a factor of 2. Another interesting design choice of *VGG* was that the number of filters was often increased by a factor of 2 after each max-pooling. The idea was to always increase the depth by a factor of 2 whenever the spatial footprint reduced by a factor of 2. This design choice results in some level of balance in the computational effort across layers, and was inherited by some of the later architectures like *ResNet*.

One issue with using deep configurations was that increased depth led to greater sensitivity with initialization, which is known to cause instability. This problem was solved by using pretraining, in which a shallower architecture was first trained, and then further layers were added. However, the pretraining was not done on a layer-by-layer basis. Rather, an 11-layer subset of the architecture was first trained. These trained layers were used to initialize a subset of the layers in the deeper architecture. *VGG* achieved a top-5 error of only 7.3% in the ISLVRC contest, which was one of the top performers but not the winner. The different configurations of *VGG* are shown in Table 8.2. Among these, the architecture denoted by column D was the winning architecture. Note that the number of filters increase by a factor of 2 after each max-pooling. Therefore, max-pooling causes the spatial height and width to reduce by a factor of 2, but this is compensated by increasing depth by a factor of 2. Performing convolutions with 3×3 filters and padding of 1 does not change the spatial footprint. Therefore, the sizes of each spatial dimension (i.e., height and width) in the regions between different max-pooling layers in column D of Table 8.2 are 224, 112, 56, 28, and 14, respectively. A final max-pooling is performed just before creating the fully connected layer, which reduces the spatial footprint further to 7. Therefore, the first fully

Table 8.2: Configurations used in *VGG*. The term C3D64 refers to the case in which convolutions are performed with 64 filters of spatial size 3×3 (and occasionally 1×1). The depth of the filter matches the corresponding layer. The padding of each filter is chosen in order to maintain the spatial footprint of the layer. All convolutions are followed by ReLU. The max-pool layer is referred to as M, and local response normalization as LRN. The softmax layer is denoted by S, and FC4096 refers to a fully connected layer with 4096 units. Other than the final set of layers, the number of filters always increases after each max-pooling. Therefore, reduced spatial footprint is often accompanied with increased depth.

Name:	**A**	**A-LRN**	**B**	**C**	**D**	**E**
# Layers	11	11	13	16	16	19
	C3D64	C3D64	C3D64	C3D64	C3D64	C3D64
		LRN	C3D64	C3D64	C3D64	C3D64
	M	M	M	M	M	M
	C3D128	C3D128	C3D128	C3D128	C3D128	C3D128
			C3D128	C3D128	C3D128	C3D128
	M	M	M	M	M	M
	C3D256	C3D256	C3D256	C3D256	C3D256	C3D256
	C3D256	C3D256	C3D256	C3D256	C3D256	C3D256
				C1D256	C3D256	C3D256
						C3D256
	M	M	M	M	M	M
	C3D512	C3D512	C3D512	C3D512	C3D512	C3D512
	C3D512	C3D512	C3D512	C3D512	C3D512	C3D512
				C1D512	C3D512	C3D512
						C3D512
	M	M	M	M	M	M
	C3D512	C3D512	C3D512	C3D512	C3D512	C3D512
	C3D512	C3D512	C3D512	C3D512	C3D512	C3D512
				C1D512	C3D512	C3D512
						C3D512
	M	M	M	M	M	M
	FC4096	FC4096	FC4096	FC4096	FC4096	FC4096
	FC4096	FC4096	FC4096	FC4096	FC4096	FC4096
	FC1000	FC1000	FC1000	FC1000	FC1000	FC1000
	S	S	S	S	S	S

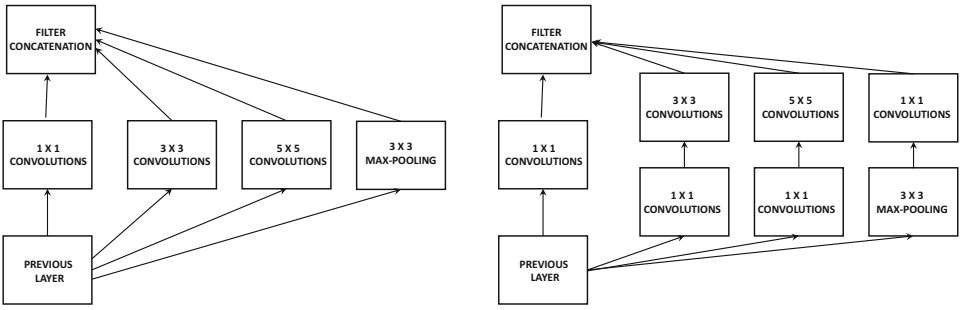
connected layer has dense connections between 4096 neurons and a $7 \times 7 \times 512$ volume. As we will see later, most of the parameters of the neural network are hidden in these connections.

An interesting exercise has been shown in [236] about where most of the parameters and the memory of the activations is located. In particular, the vast majority of the *memory* required for storing the activations and gradients in the forward and backward phases are required by the early part of the convolutional neural network with the largest spatial footprint. This point is significant because the memory required by a mini-batch is scaled by the size of the mini-batch. For example, it is has been shown in [236] that about 93MB are required for each image. Therefore, for a mini-batch size of 128, the total memory requirement would be about 12GB. Although the early layers require the most memory because of their large spatial footprints, they do not have a large parameter footprint because of the sparse connectivity and weight sharing. In fact, most of the parameters are required by the fully connected layers at the end. The connection of the final $7 \times 7 \times 512$ spatial layer (cf. column D in Table 8.2) to the 4096 neurons required $7 \times 7 \times 512 \times 4096 = 102,760,448$ parameters. The total number of parameters in *all* layers was about $138,000,000$. Therefore, *nearly* 75% *of the parameters are in a single layer of connections.* Furthermore, the majority of the remaining parameters are in the final two fully connected layers. In all, dense connectivity accounts for 90% of the parameter footprint in the neural network. This point is significant, as *GoogLeNet* uses some innovations to reduce the parameter footprint in the final layers.

It is notable that some of the architectures allow 1×1 convolutions. Although a 1×1 convolution does not combine the activations of spatially adjacent features, it does combine the feature values of different channels when the depth of a volume is greater than 1. Using a 1×1 convolution is also a way to incorporate additional nonlinearity into the architecture without making fundamental changes at the spatial level. This additional nonlinearity is incorporated via the ReLU activations attached to each layer. Refer to [454] for more details.

8.4.4 GoogLeNet

GoogLeNet proposed a novel concept referred to as an *inception architecture*. An inception architecture is a *network within a network*. The initial part of the architecture is much like a traditional convolutional network, and is referred to as the *stem*. The key part of the network is an intermediate layer, referred to as an *inception module*. An example of an inception module is illustrated in Figure 8.10(a). The basic idea of the inception module is that key information in the images is available at different levels of detail. If we use a large filter, we can capture information in a bigger area containing limited variation; if we use a smaller filter, we can capture detailed information in a smaller area. While one solution would be to pipe together many small filters, this would be wasteful of parameters and depth when it would suffice to use the broader patterns in a larger area. The problem is that we do not know up front which level of detail is appropriate for each region of the image. Why not give the neural network the flexibility to model the image at different levels of granularities? This is achieved with an inception module, which convolves with three different filter sizes in parallel. These filter sizes are 1×1, 3×3, and 5×5. A purely sequential piping of filters of the same size is inefficient when one is faced with objects of different scales in different images. Since all filters on the inception layer are learnable, the neural network can decide which ones will influence the output the most. By choosing filters of different sizes along different paths, different regions are represented at a different level of granularity. *GoogLeNet* is made up of nine inception modules that are arranged sequentially. Therefore, one can choose many alternative paths through the architecture, and the resulting features will represent very different spatial regions. For example, passing through four 3×3 filters



(a) Basic inception module (b) Implementation with 1×1 bottlenecks

Figure 8.10: The inception module of *GoogLeNet*

followed by only 1×1 filters will capture a relatively small spatial area. On the other hand, passing through many 5×5 filters will result in a much larger spatial footprint. In other words, the differences in the scales of the shapes captured in different hidden features will be magnified in later layers. In recent years, batch normalization has been used in conjunction with the inception architecture, which simplifies[6] the network structure from its original form.

One observation is that the inception module results in some computational inefficiency because of the large number of convolutions of different sizes. Therefore, an efficient implementation is shown in Figure 8.10(b), in which 1×1 convolutions are used to first reduce the depth of the feature map. This is because the number of 1×1 convolution filters is a modest factor less than the depth of the input volume. For example, one might first reduce an input depth of 256 to 64 by using 64 different 1×1 filters. These additional 1×1 convolutions are referred to as the *bottleneck operations* of the inception module. Initially reducing the depth of the feature map (with cheap 1×1 convolutions) saves computational efficiency with the larger convolutions because of the reduced depth of the layers after applying the bottleneck convolutions. One can view the 1×1 convolutions as a kind of supervised dimensionality reduction before applying the larger spatial filters. The dimensionality reduction is supervised because the parameters in the bottleneck filters are learned during backpropagation. The bottleneck also helps in reducing the depth after the pooling layer. The trick of bottleneck layers is also used in some other architectures, where it is helpful for improving efficiency and output depth.

The output layer of *GoogLeNet* also illustrates some interesting design principles. It is common to use fully connected layers near the output. However, *GoogLeNet* uses average pooling across the whole spatial area of the final set of activation maps to create a single value. Therefore, the number of features created in the final layer will be exactly equal to the number of filters. An important observation is that the vast majority of parameters are spent in connecting the final convolution layer to the first fully connected layer. This type of detailed connectivity is not required for applications in which only a class label needs to be predicted. Therefore, the average pooling approach is used. However, the average pooled representation completely loses all spatial information, and one must be careful of the types of applications it is used for. An important property of *GoogLeNet* was that it is extremely compact in terms of the number of parameters in comparison with *VGG*, and the number of

[6]The original architecture also contained auxiliary classifiers, which have been ignored in recent years.

parameters in the former is less by an order of magnitude. This is primarily because of the use of average pooling, which eventually became standard in many later architectures. On the other hand, the overall architecture of *GoogLeNet* is computationally more expensive.

The flexibility of *GoogLeNet* is inherent in the 22-layered inception architecture, in which objects of different scales are handled with the appropriate filter sizes. This flexibility of multigranular decomposition, which is enabled by the inception modules, was one of the keys to its performance. In addition, the replacement of the fully connected layer with average pooling greatly reduced the parameter footprint. This architecture was the winner of the ILSVRC contest in 2014, and *VGG* placed a close second. Even though *GoogLeNet* outperformed *VGG*, the latter does have the advantage of simplicity, which is sometimes appreciated by practitioners. Both architectures illustrated important design principles for convolution neural networks. The inception architecture has been the focus of significant research since then [486, 487], and numerous changes have been suggested to improve performance. In later years, a version of this architecture, referred to as *Inception-v4* [487], was combined with some of the ideas in *ResNet* (see next section) to create a 75-layer architecture with only 3.08% error.

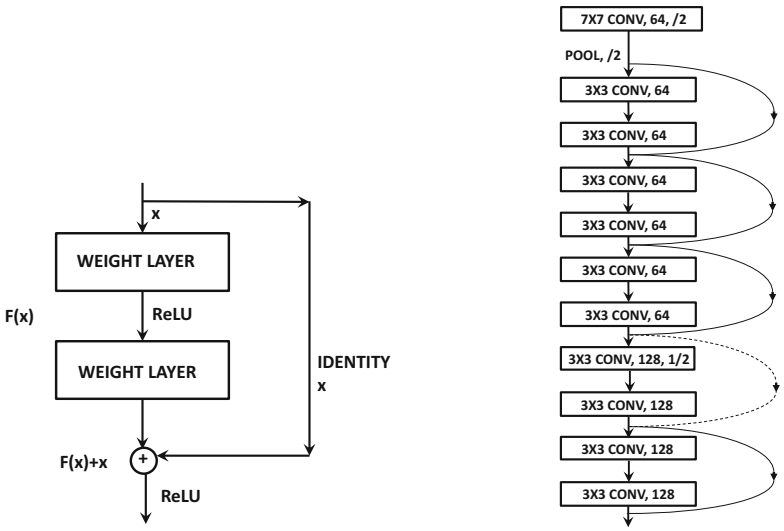
8.4.5 ResNet

ResNet [184] used 152 layers, which was almost an order of magnitude greater than previously used by other architectures. This architecture was the winner of the ILSVRC competition in 2015, and it achieved a top-5 error of 3.6%, which resulted in the first classifier with human-level performance. This accuracy is achieved by an ensemble of *ResNet* networks; even a single model achieves 4.5% accuracy. Training an architecture with 152 layers is generally not possible unless some important innovations are incorporated.

The main issue in training such deep networks is that the gradient flow between layers is impeded by the large number of operations in deep layers that can increase or decrease the size of the gradients. As discussed in Chapter 3, problems such as the vanishing and exploding gradients are caused by increased depth. However, the work in [184] suggests that the main training problem in such deep networks might not necessarily be caused by these problems, especially if batch normalization is used. The main problem is caused by the difficulty in getting the learning process to converge properly in a reasonable amount of time. Such convergence problems are common in networks with complex loss surfaces. Although some deep networks show large gaps between training and test error, the error on both the training ad test data is high in many deep networks. This implies that the optimization process has not made sufficient progress.

Although hierarchical feature engineering is the holy grail of learning with neural networks, its layer-wise implementations force all concepts in the image to require the same level of abstraction. Some concepts can be learned by using shallow networks, whereas others require fine-grained connections. For example, consider a circus elephant standing on a square frame. Some of the intricate features of the elephant might require a large number of layers to engineer, whereas the features of the square frame might require very few layers. Convergence will be unnecessarily slow when one is using a very deep network with a fixed depth across all paths to learn concepts, many of which can also be learned using shallow architectures. Why not let the neural network decide how many layers to use to learn each feature?

ResNet uses *skip connections* between layers in order to enable copying between layers and introduces an *iterative view* of feature engineering (as opposed to a hierarchical view). Long short-term memory networks and gated recurrent units leverage similar principles in sequence data by allowing portions of the states to be copied from one layer to the



(a) Skip-connections in residual module (b) Partial architecture of *ResNet*

Figure 8.11: The residual module and the first few layers of *ResNet*

next with the use of adjustable *gates*. In the case of *ResNet*, the non-existent "gates" are assumed to be always fully open. Most feed-forward networks only contain connections between layers i and $(i+1)$, whereas *ResNet* contains connections between layers i and $(i+r)$ for $r > 1$. Examples of such skip connections, which form the basic unit of *ResNet*, are shown in Figure 8.11(a) with $r = 2$. This skip connection simply copies the input of layer i and adds it to the output of layer $(i+r)$. Such an approach enables effective gradient flow because the backpropagation algorithm now has a super-highway for propagating the gradients backwards using the skip connections. This basic unit is referred to as a *residual module*, and the entire network is created by putting together many of these basic modules. In most layers, an appropriately padded filter[7] is used with a stride of 1, so that the spatial size and depth of the input does not change from layer to layer. In such cases, it is easy to simply add the input of the ith layer to that of $(i+r)$. However, some layers do use strided convolutions to reduce each spatial dimension by a factor of 2. At the same time, depth is increased by a factor of 2 by using a larger number of filters. In such a case, one cannot use the identity function over the skip connection. Therefore, a linear projection matrix might need to be applied over the skip connection in order to adjust the dimensionality. This projection matrix defines a set of 1×1 convolution operations with stride of 2 in order to reduce spatial extent by factor of 2. The parameters of the projection matrix need to be learned during backpropagation.

In the original idea of *ResNet*, one only adds connections between layers i and $(i+r)$. For example, if we use $r = 2$, only skip connections only between successive odd layers are used. Later enhancements like *DenseNet* showed improved performance by adding connections between all pairs of layers. The basic unit of Figure 8.11(a) is repeated in *ResNet*, and therefore one can traverse the skip connections repeatedly in order to propagate input to the output after performing very few forward computations. An example of the first few

[7]Typically, a 3×3 filter is used at a stride/padding of 1. This trend started with the principles in *VGG*, and was adopted by *ResNet*.

layers of the architecture is shown in Figure 8.11(b). This particular snapshot is based on the first few layers of the 34-layer architecture. Most of the skip connections are shown in solid lines in Figure 8.11(b), which corresponds to the use of the identity function with an unchanged filter volume. However, in some layers, a stride of 2 is used, which causes the spatial and depth footprint to change. In these layers, a projection matrix needs to be used, which is denoted by a dashed skip connection. Four different architectures were tested in the original work [184], which contained 34, 50, 101, and 152 layers, respectively. The 152-layer architecture had the best performance, but even the 34-layer architecture performed better than did the best-performing ILSVRC entry from the previous year.

The use of skip connections provides paths of unimpeded gradient flow and therefore has important consequences for the behavior of the backpropagation algorithm. The skip connections take on the function of super-highways in enabling gradient flow, creating a situation where multiple paths of variable lengths exist from the input to the output. In such cases, the shortest paths enable the most learning, and the longer paths can be viewed as residual contributions. This gives the learning algorithm the flexibility of choosing the appropriate level of nonlinearity for a particular input. Inputs that can be classified with a small amount of nonlinearity will skip many connections. Other inputs with a more complex structure might traverse a larger number of connections in order to extract the relevant features. Therefore, the approach is also referred to as residual learning, in which learning along longer paths is a kind of fine tuning of the easier learning along shorter paths. In other words, the approach is well suited to cases in which different aspects of the image have different levels of complexity. The work in [184] shows that the residual responses from deeper layers are often relatively small, which validates the intuition that fixed depth is an impediment to proper learning. In such cases, the convergence is often not a problem, because the shorter paths enable a significant portion of the learning with unimpeded gradient flows. An interesting insight in [505] is that *ResNet* behaves like an ensemble of shallow networks because many alternative paths of shorter length are enabled by this type of architecture. Only a small amount of learning is enabled by the deeper paths, and only when it is absolutely necessary. The work in [505] in fact provides a pictorial depiction of an unraveled architecture of *ResNet* in which the different paths are explicitly shown in a parallel pipeline. This unraveled view provides a clear understanding of why *ResNet* has some similarities with ensemble-centric design principles. A consequence of this point of view is that dropping some of the layers from a trained *ResNet* at prediction time does not degrade accuracy as significantly as other networks like *VGG*.

More insights can be obtained by reading the work on *wide residual networks* [549]. This work suggests that increased depth of the residual network does not always help because most of the extremely deep paths are not used anyway. The skip connections do result in alternative paths and effectively increase the width of the network. The work in [549] suggests that better results can be obtained by limiting the total number of layers to some extent (say, 50 instead of 150), and using an increased number of filters in each layer. Note that a depth of 50 is still quite large from pre-*ResNet* standards, but is low compared to the depth used in recent experiments with residual networks. This approach also helps in parallelizing operations.

Variations of Skip Architectures

Since the architecture of *ResNet* was proposed, several variations were suggested to further improve performance. For example, the independently proposed *highway networks* [161]

Table 8.3: The number of layers in various top-performing ILSVRC contest entries

Name	Year	Number of Layers	Top-5 Error
–	Before 2012	≤ 5	$> 25\%$
AlexNet	2012	8	15.4%
ZfNet/*Clarifai*	2013	8/> 8	14.8% / 11.1%
VGG	2014	19	7.3%
GoogLeNet	2014	22	6.7%
ResNet	2015	152	3.6%

introduced the notion of gated skip connections, and can be considered a more general architecture. In highway networks, gates are used in lieu of the identity mapping, although a closed gate does not pass a lot of information through. In such cases, gating networks do not behave like residual networks. However, residual networks can be considered special cases of gating networks in which the gates are always fully open. Highway networks are closely related to both LSTMs and *ResNets*, although *ResNets* still seem to perform better in the image recognition task because of their focus on enabling gradient flow with multiple paths. The original *ResNet* architecture uses a simple block of layers between skip connections. However, the *ResNext* architecture varies on this principle by using inception modules between skip connections [537].

Instead of using skip connections, one can use convolution transformations between every pair of layers [211]. Therefore, instead of the L transformations in a feed-forward network with L layers, one is using $L(L-1)/2$ transformations. In other words, the concatenation of all the feature maps of the previous $(l-1)$ layers is used by the lth layer. This architecture is referred to as *DenseNet*. Note that the goal of such an architecture is similar to that of skip connections by allowing each layer to learn from whatever level of abstraction is useful.

An interesting variant that seems to work well is the use of *stochastic depth* [210] in which some of the blocks between skip connections are randomly dropped during training time, but the full network is used during testing time. Note that this approach seems similar to *Dropout*, which makes the network thinner rather than shallower by dropping nodes. However, *Dropout* has somewhat different motivations from layer-wise node dropping, because the latter is more focused on improving gradient flow rather than preventing feature co-adaptation.

8.4.6 The Effects of Depth

The significant advancements in performance in recent years in the ILSVRC contest are mostly a result of improved computational power, greater data availability, and changes in architectural design that have enabled the effective training of neural networks with increased depth. These three aspects also support each other, because experimentation with better architectures is only possible with sufficient data and improved computational efficiency. This is also one of the reasons why the fine-tuning and tweaks of relatively old architectures (like recurrent neural networks) with known problems were not performed until recently.

The number of layers and the error rates of various networks are shown in Table 8.3. The rapid increase in accuracy in the short period from 2012 to 2015 is quite remarkable, and is unusual for most machine learning applications that are as well studied as image recognition. Another important observation is that increased depth of the neural network is

closely correlated with improved error rates. Therefore, an important focus of the research in recent years has been to enable algorithmic modifications that support increased depth of the neural architecture. It is noteworthy that convolutional neural networks are among the deepest of all classes of neural networks. Interestingly, traditional feed-forward networks in other domains do not need to be very deep for most applications like classification. Indeed, the coining of the term "deep learning" owes a lot of its origins to the impressive performances of convolutional neural networks and specific improvements observed with increased depth.

8.4.7 Pretrained Models

One of the challenges faced by analysts in the image domain is that *labeled* training data may not even be available for a particular application. Consider the case in which one has a set of images that need to be used for image retrieval. In retrieval applications, labels are not available but it is important for the features to be semantically coherent. In some other cases, one might wish to perform classification on a data set with a particular set of labels, which might be limited in availability and different from large resources like *ImageNet*. These settings cause problems because neural networks require a lot of training data to build from scratch.

However, a key point about image data is that the extracted features from a particular data set are highly reusable across data sources. For example, the way in which a cat is represented will not vary a lot if the same number of pixels and color channels are used in different data sources. In such cases, generic data sources, which are representative of a wide spectrum of images, are useful. For example, the *ImageNet* data set [581] contains more than a million images drawn from 1000 categories encountered in everyday life. The chosen 1000 categories and the large diversity of images in the data set are representative and exhaustive enough that one can use them to extract features of images for general-purpose settings. For example, the features extracted from the *ImageNet* data can be used to represent a completely different image data set by passing it through a pretrained convolutional neural network (like *AlexNet*) and extracting the multidimensional features from the fully connected layers. This new representation can be used for a completely different application like clustering or retrieval. This type of approach is so common, that *one rarely trains convolutional neural networks from scratch*. The extracted features from the penultimate layer are often referred to as FC7 features, which is an inheritance from the name of the layer in *AlexNet*. Of course, an arbitrary convolutional network might not have the same number of layers as *AlexNet*; however, the name FC7 has stuck.

This type of off-the-shelf feature extraction approach [390] can be viewed as a kind of *transfer learning*, because we are using a public resource like *ImageNet* to extract features to solve different problems in settings where enough training data is not available. Such an approach has become standard practice in many image recognition tasks, and many software frameworks like *Caffe* provide ready access to these features [585, 586]. In fact, *Caffe* provides a "zoo" of such pretrained models, which can be downloaded and used [586]. If some additional training data is available, one can use it to fine-tune only the deeper layers (i.e., layers closer to the output layer). The weights of the early layers (closer to the input) are fixed. The reason for training only the deeper layers, while keeping the early layers fixed, is that the earlier layers capture only primitive features like edges, whereas the deeper layers capture more complex features. The primitive features do not change too much with the application at hand, whereas the deeper features might be sensitive to the application at hand. For example, all types of images will require edges of different

orientation to represent them (captured in early layers), but a feature corresponding to the wheel of a truck will be relevant to a data set containing images of trucks. In other words, early layers tend to capture highly generalizable features (across different computer vision data sets), whereas later layers tend to capture data-specific features. A discussion of the transferability of features derived from convolutional neural networks across data sets and tasks is provided in [361].

8.5 Visualization and Unsupervised Learning

An interesting property of convolutional neural networks is that they are highly interpretable in terms of the types of features they can learn. However, it takes some effort to actually interpret these features. The first approach that comes to mind is to simply visualize the 2-dimensional (spatial) components of the filters. Although this type of visualization can provide some interesting visualizations of the primitive edges and lines learned in the first layer of the neural network, it is not very useful for later layers. In the first layer, it is possible to visualize these filters because they operate directly on the input image, and often tend to look like primitive parts of the image (such as edges). However, it is not quite as simple a matter to visualize these filters in later layers because they operate on input volumes that have already been scrambled with convolution operations. In order to obtain any kind of interpretability one must find a way to map the impacts of all operations all the way back to the input layer. Therefore, the goal of visualization is often to identify and highlight the portions of the input image to which a particular hidden feature is responding. For example, the value of one hidden feature might be sensitive to changes in the portion of the image corresponding to the wheel of a truck, and a different hidden feature might be sensitive to its hood. This is naturally achieved by computing the sensitivity (i.e., gradient) of a hidden feature with respect to each pixel of the input image. As we will see, these types of visualizations are closely related to backpropagation, unsupervised learning, and transposed convolutional operations (used for creating the decoder portions of autoencoders). Therefore, this chapter will discuss these closely related topics in an integrated way.

There are two primary settings in which one can encode and decode an image. In the first setting, the compressed feature maps are learned by using any of the supervised models discussed in earlier sections. Once the network has been trained in a supervised way, one can attempt to reconstruct the portions of the image that most activate a given feature. Furthermore, the portions of an image that are most likely to activate a particular hidden feature or a class are identified. As we will see later, this goal can be achieved with various types of backpropagation and optimization formulations. The second setting is purely unsupervised, in which a convolutional network (encoder) is hooked up to a deconvolutional network (decoder). As we will see later, the latter is also a form of transposed convolution, which is similar to backpropagation. However, in this case, the weights of the encoder and decoder are learned jointly to minimize the reconstruction error. The first setting is obviously simpler because the encoder is trained in a supervised way, and one only has to learn the effect of different portions of the input field on various hidden features. In the second setting, the entire training and learning of weights of the network has to be done from scratch.

8.5.1 Visualizing the Features of a Trained Network

Consider a neural network that has already been trained using a large data set like *ImageNet*. The goal is to visualize and understand the impact of the different portions of the input image (i.e., receptive field) on various features in the hidden layers and the output layer (e.g., the 1000 softmax outputs in *AlexNet*). We would like to answer the following questions:

1. Given an activation of a feature anywhere in the neural network for a *particular* input image, visualize the portions of the input to which that feature is responding the most. Note that the feature might be one of the hidden features in the spatially arranged layers, in the fully connected hidden layers (e.g., FC7), or even one of the softmax outputs. In the last of these cases, one obtains some insight of the specific relationship of a particular input image to a class. For example, if an input image is activating the label for "*banana*," we hope to see the parts of the specific input image that look most like a banana.

2. Given a particular feature anywhere in the neural network, find a fantasy image that is likely to activate that feature the most. As in the previous case, the feature might be one of the hidden features or even one of the features from the softmax outputs. For example, one might want to know what type of fantasy image is most likely to classify to a "*banana*" in the trained network at hand.

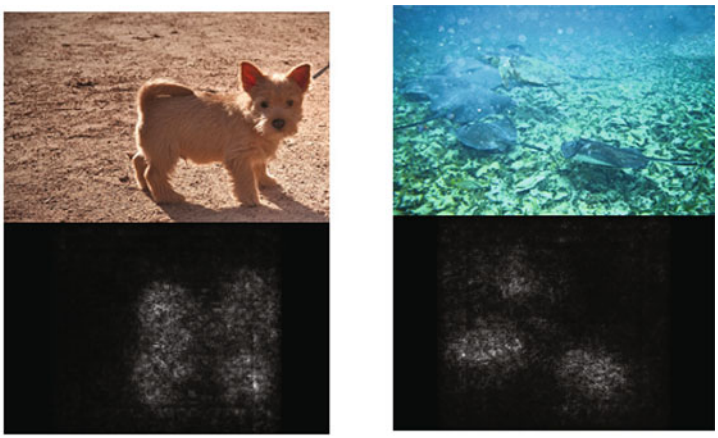
In both these cases, the easiest approach to visualize the impact of specific features is to use gradient-based methods. The second of the above goals is rather hard, and one often does not obtain satisfactory visualizations without careful regularization.

Gradient-Based Visualization of Activated Features

The backpropagation algorithm that is used to train the neural network is also helpful for gradient-based visualization. It is noteworthy that backpropagation-based gradient computation is a form of transposed convolution. In traditional autoencoders, transposed weight matrices (of those used in the encoder layer) are often used in the decoder. Therefore, the connections between backpropagation and feature reconstruction are deep and are applicable across all types of neural networks. The main difference from the traditional backpropagation setting is that our end-goal is to determine the sensitivity of the hidden/output features with respect to *different pixels of the input image* rather than with respect to the weights. However, even traditional backpropagation does compute the sensitivity of the outputs with respect to various layers as an intermediate step, and therefore almost exactly the same approach can be used in both cases.

When the sensitivity of an output o is computed with respect to the input pixels, the visualization of this sensitivity over the corresponding pixels is referred to as a *saliency map* [456]. For example, the output o might be the softmax probability (or unnormalized score before applying softmax) of the class "*banana*." Then, for each pixel x_i in the image, we would like to determine the value of $\frac{\partial o}{\partial x_i}$. This value can be computed by straightforward backpropagation all the way[8] to the input layer. The softmax probability of "*banana*" will be relatively insensitive to small changes in those portions of the image that are irrelevant to the recognition of a banana. Therefore, the values of $\frac{\partial o}{\partial x_i}$ will be close to 0 for such

[8] Under normal circumstances, one only backpropagates to hidden layers as an intermediate step to compute gradients with respect to incoming weights in that hidden layer. Therefore, backpropagation to input layer is never really needed in traditional training. However, backpropagation to the input layer is identical to that with respect to the hidden layers.



(a) (b)

Figure 8.12: Examples of portions of specific images activated by particular class labels. These images appear in the work by Simonyan, Vedaldi, and Zisserman [456]. Reproduced with permission. (©2014 Simonyan, Vedaldi, and Zisserman)

irrelevant regions, whereas the portions of the image that define a banana will have large magnitudes. For example, in the case of *AlexNet*, the entire $224 \times 224 \times 3$ volume defined by $\frac{\partial o}{\partial x_i}$ of *backpropagated gradients* will have portions with large magnitudes corresponding to the banana in the image. To visualize this volume, we first convert it to grayscale by taking the *maximum of the absolute magnitude of the gradient* over the three RGB channels to create a $224 \times 224 \times 1$ map with only non-negative values. The bright portions of this grayscale visualization will tell us which portion of the input image are relevant to the banana. Examples of grayscale visualizations of the portions of the image that excite relevant classes are shown in Figure 8.12. For example, the bright portion of the image in Figure 8.12(a) excites the animal in the image, which also represents its class label. As discussed in Section 2.4 of Chapter 2, this type of approach can also be used for interpretability and feature selection in traditional neural networks (and not just convolutional methods).

This general approach has also been used for visualizing the activations of specific hidden features. Consider the value h of a hidden variable for a particular input image. How is this variable responding to the input image at its current activation level? The idea is that if we slightly increase or decrease the color intensity of some pixels, the value of h will be affected more than if we increase or decrease other pixels. First, the hidden variable h will be affected by a small rectangular portion of the image (i.e., receptive field), which is very small when h is present in early layers but much larger in later layers. For example, the receptive field of h might only be of size 3×3 when it is selected from the first hidden layer in the case of *VGG*. Examples of the image crops corresponding to specific images in which a particular neuron in a hidden layer is highly activated are shown in each row on the right-hand side of Figure 8.13. Note that each row contains a somewhat similar image. This is not a coincidence because that row corresponds to a particular hidden feature, and the variations in that row are caused by the different choices of image. Note that the choices of the image for a row is also not random, because we are selecting the images that most activate that feature. Therefore, all the images will contain the same visual characteristic

that cause this hidden feature to be activated. The grayscale portion of the visualization corresponds to the sensitivity of the feature to the pixel-specific values in the corresponding image crop.

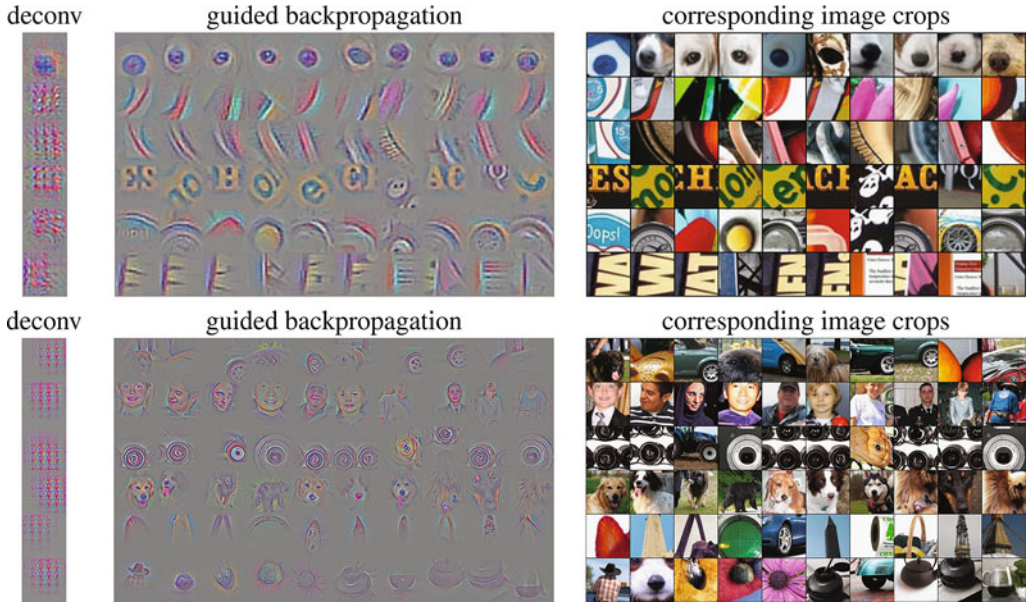


Figure 8.13: Examples of activation visualizations in different layers in Springenberg *et al.*'s work [466]. Reprinted from [466] with permission (©2015 Springenberg, Dosovitskiy, Brox, Riedmiller).

At a high level of activation level of h, some of the pixels in that receptive field will be more sensitive to h than others. By isolating the pixels to which the hidden variable h has the greatest sensitivity and visualizing the corresponding rectangular regions, one can get an idea of what part of the input map most affects a particular hidden feature. Therefore, any particular pixel x_i, we want to compute $\frac{\partial h}{\partial x_i}$, and then visualize those pixels with large values of this gradient. However, instead of backpropagation, the notions of *"deconvnet"* [556] and *guided backpropagation* [466] are sometimes used. The notion of "deconvent" is also used in convolutional autoencoders. The main difference is in terms of how the gradient of the ReLU nonlinearity is propagated backwards. As discussed in Table 3.1 of Chapter 3, the partial derivative of a ReLU unit is copied backwards during backpropagation if the *input* to the ReLU is positive, and is otherwise set to 0. However, in "deconvnet," the partial derivative of a ReLU unit is copied backwards, if this partial derivative is itself larger than 0. This is like using a ReLU on the propagated gradient in the backward pass. In other words, we replace $\overline{g}_i = \overline{g}_{i+1} \odot I(\overline{z}_i > 0)$ in Table 3.1 with $\overline{g}_i = \overline{g}_{i+1} \odot I(\overline{g}_{i+1} > 0)$. Here $\overline{z}_i$ represents the forward activations, and $\overline{g}_i$ represents the backpropagated gradients with respect to the ith layer containing only ReLU units. The function $I(\cdot)$ is an element-wise indicator function, which takes on the value of 1 for each element in the vector argument when the condition is true for that element. In guided backpropagation, we *combine* the conditions used in traditional backpropagation and ReLU by using $\overline{g}_i = \overline{g}_{i+1} \odot I(\overline{z}_i > 0) \odot I(\overline{g}_{i+1} > 0)$. A pictorial illustration of the three variations of backpropagation is shown in Figure 8.14. It is suggested in [466] that guided backpropagation gives better visualizations than "deconvnet," which in turn gives better results than traditional backpropagation.

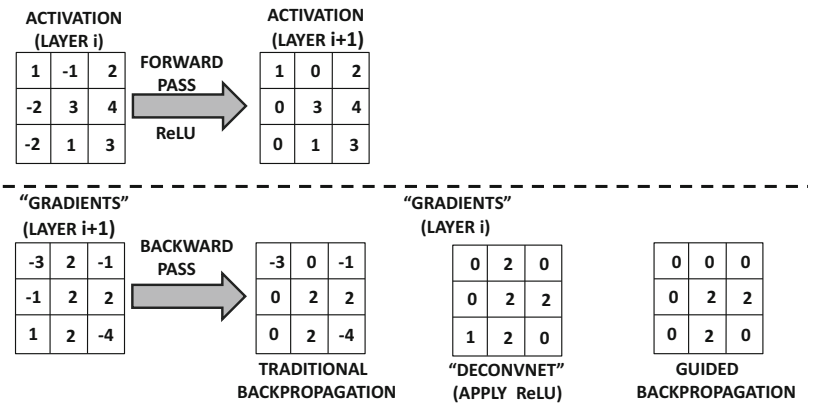


Figure 8.14: The different variations of backpropagation of ReLU for visualization

One way of interpreting the difference between traditional backpropagation and "deconvnet" is by interpreting backwards propagation of gradients as the operations of a decoder with transposed convolutions with respect to the encoder [456]. However, in this decoder, we are again using the ReLU function rather than the gradient-based transformation implied by the ReLU. After all, all forms of decoders use the same activation functions as the encoder. Another feature of the visualization approach in [466] is that it omits the use of pooling layers in the convolutional neural network altogether, and instead relies on strided convolutions. The work in [466] identified several highly activated neurons in specific layers corresponding to specific input images and provided visualizations of the rectangular regions of those images corresponding to the receptive fields of those hidden neurons. We have already discussed earlier that the right-hand side of Figure 8.13 contains the input regions corresponding to specific neurons in hidden layers. The left-hand side of Figure 8.13 also shows the specific characteristics of each image that excite that particular neuron. The visualization on the left-hand side is obtained with guided backpropagation. Note that the upper set of images correspond to the sixth layer, whereas the lower set of images corresponds to the ninth layer of the convolutional network. As a result, the images in the lower set typically corresponds to larger regions of the input image containing more complex shapes.

Another excellent set of visualizations from [556] is shown in Figure 8.15. The main difference is that the work in [556] also uses max-pooling layers, and is based on deconvolutions rather than guided backpropagation. The specific hidden variables chosen are the top-9 largest activations in each feature map. In each case, the relevant square region of the image is shown together with the corresponding visualization. It is evident that the hidden features in early layers correspond to primitive lines, which become increasingly more complex in later layers. This is one of the reasons that convolutional neural networks are viewed as methods that create hierarchical features. The features in early layers tend to be more generic, and they can be used across a wider variety of data sets. The features in later layers tend to be more specific to individual data sets. This is a key property exploited in transfer learning applications, in which pretrained networks are broadly used, and only the later layers are fine-tuned in a manner that's specific to data set and application.

Synthesized Images that Activate a Feature

The above examples tell us the portions of a *particular* image that most affect a particular neuron. A more general question is to ask what kind of image patch would maximally activate a particular neuron. For ease in discussion, we will discuss the case in which the neuron is an output value o of a particular class (i.e., unnormalized output before applying softmax). For example, the value of o might be the unnormalized score for "*banana*." Note that one can also apply a similar approach to intermediate neurons rather than the class score. We would like to learn the input image $\overline{x}$ that maximizes the output o, while applying regularization to $\overline{x}$:

$$\text{Maximize}_{\overline{x}} \; J(\overline{x}) = (o - \lambda ||\overline{x}||^2)$$

Here, λ is the regularization parameter, and is important in order to extract semantically interpretable images. One can use gradient ascent in conjunction with backpropagation in order to learn the input image $\overline{x}$ that maximizes the above objective function. Therefore, we start with a zero image $\overline{x}$ and update $\overline{x}$ using gradient ascent in conjunction with backpropagation with respect to the above objective function. In other words, the following update is used:

$$\overline{x} \Leftarrow \overline{x} + \alpha \nabla_{\overline{x}} J(\overline{x}) \tag{8.4}$$

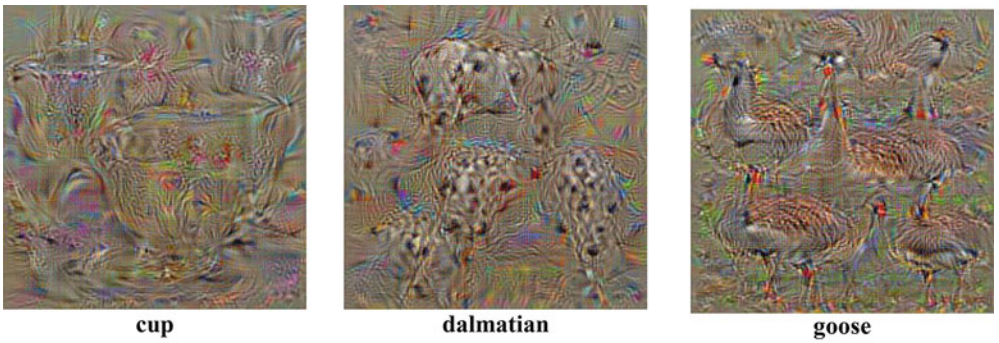
Here, α is the learning rate. The key point is that backpropagation is being leveraged in an unusual way to update the *image pixels* while keeping the (already learned) weights fixed. Examples of synthesized images for three classes are shown in Figure 8.16. Other advanced methods for generating more realistic images on the basis of class labels are discussed in [358].

8.5.2 Convolutional Autoencoders

The use of the autoencoder in traditional neural networks is discussed in Chapters 2 and 4. Recall that the autoencoder reconstructs data points after passing them through a compression phase. In some cases, the data is not compressed although the representations are sparse. The portion before the most compressed layer of the architecture is referred to as the encoder, and the portion after the compressed portion is referred to as the decoder. We repeat the pictorial view of the encoder-decoder architecture for the traditional case in Figure 8.17(a). The convolutional autoencoder has a similar principle, which reconstructs images after passing them through a compression phase. The main difference between a traditional autoencoder and a convolutional autoencoder is that the latter is focused on using spatial relationships between points in order to extract features that have a visual interpretation. The spatial convolution operations in the intermediate layers achieve precisely this goal. An illustration of the convolutional autoencoder is shown in Figure 8.17(b) in comparison with the traditional autoencoder in Figure 8.17(a). Note the 3-dimensional spatial shape of the encoder and decoder in the second case. However, it is possible to conceive of several variations to this basic architecture. For example, the codes in the middle can either be spatial or they can be flattened with the use of fully connected layers, depending on the application at hand. The fully connected layers would be necessary to create a multidimensional code that can be used with arbitrary applications (without worrying about spatial constraints among features). In the following, we will simplify the discussion by assuming that the compressed code in the middle is spatial in nature.

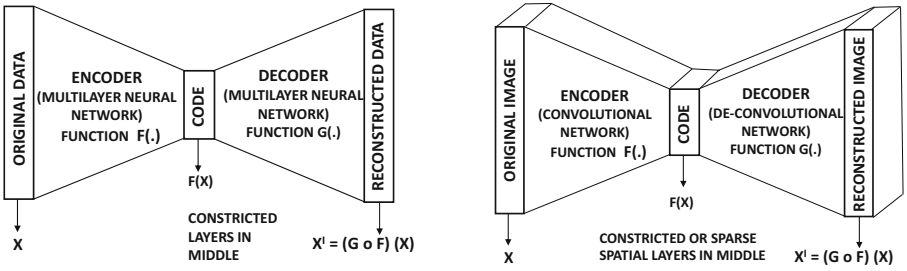
Just as the compression portion of the encoder uses a convolution operation, the decompression operation uses a *deconvolution* operation. Similarly, pooling is matched with an

Figure 8.15: Examples of activation visualizations in different layers based on Zeiler and Fergus's work [556]. Reprinted from [556] with permission. ©Springer International Publishing Switzerland, 2014.



(a) (b) (c)

Figure 8.16: Examples of synthesized images with respect to particular class labels. These examples appear in the work by Simonyan, Vedaldi, and Zisserman [456]. Reproduced with permission (©2014 Simonyan, Vedaldi, and Zisserman)



(a) A traditional autoencoder architecture (b) Convolutional autoencoder

Figure 8.17: A traditional autoencoder and a convolutional autoencoder.

unpooling operation. Deconvolution is also referred to as *transposed convolution*. Interestingly, the transposed convolution operation is the same as that used for backpropagation. The term "deconvolution" is perhaps a bit misleading because every deconvolution is in actuality a convolution with a filter that is derived by transposing and inverting the tensor representing the original convolution filter (cf. Figure 8.7 and Equation 8.3). We can already see that deconvolution uses similar principles to that of backpropagation. The main difference is in terms of how the ReLU function is handled, which makes deconvolution more similar to "deconvnet" or *guided* backpropagation. In fact, the decoder in a convolutional autoencoder performs similar operations to the backpropagation phase of gradient-based visualization. Some architectures do away with the pooling and unpooling operations, and work with only convolution operations (together with activation functions). A notable example is the design of *fully convolutional networks* [449, 466].

The fact that the deconvolution operation is really not much different from a convolution operation is not surprising. Even in traditional feed-forward networks, the decoder part of the network performs the same types of matrix multiplications as the encoder part of the network, except that the transposed weight matrices are used. One can summarize the analogy between traditional autoencoders and convolutional autoencoders in Table 8.4. Note that the relationship between forward backpropagation and backward propagation is

Table 8.4: The relationship between backpropagation and decoders

Linear Operation	Traditional neural networks	Convolutional neural networks
Forward Propagation	Matrix multiplication	Convolution
Backpropagation	Transposed matrix multiplication	Transposed convolution
Decoder layer	Transposed matrix multiplication (Identical to backpropagation)	Transposed convolution (Identical to backpropagation)

similar in traditional and convolutional neural networks in terms of how the corresponding matrix operations are performed. A similar observation is true about the nature of the relationship between encoders and decoders.

There are three operations corresponding to the convolution, max-pooling, and the ReLU nonlinearity. The goal is to perform the inversion of the operations in the decoder layer that have been performed in the encoder layer. There is no easy way to exactly invert some of the operations (such as max-pooling and ReLU). However, excellent image reconstruction can still be achieved with the proper design choices. First, we describe the case of an autoencoder with a single layer with convolution, ReLU, and max-pooling. Then, we discuss how to generalize it to multiple layers.

Although one typically wants to use the inverse of the encoder operations in the decoder, the ReLU is not an invertible function because a value of 0 has many possible inversions. Therefore, a ReLU is replaced by another ReLU in the decoder layer (although other options are possible). Therefore, the architecture of this simple autoencoder is as follows:

$$\underbrace{\text{Convolve} \Rightarrow \text{ReLU} \Rightarrow \text{Max-Pool}}_{\text{Encoder}} \Rightarrow \text{Code} \Rightarrow \underbrace{\text{Unpool} \Rightarrow \text{ReLU} \Rightarrow \text{De-Convolve}}_{\text{Decoder}}$$

Note that the layers are symmetrically arranged in terms of how a matching layer in the decoder undoes the effect of a corresponding layer in the encoder. However, there are many variations to this basic theme. For example, the ReLU might be placed after the deconvolution. Furthermore, in some variations [310], it is recommended to use deeper encoders than the decoders with an asymmetric architecture. However, with a stacked variation of the symmetric architecture above, it is possible to train just the encoder with a classification output layer (and a supervised data set like *ImageNet*) and then use its symmetric decoder (with transposed/inverted filters) to perform "deconvnet" visualization [556]. Although one can always use this approach to initialize the autoencoder, we will discuss enhancements of this concept where the encoder and decoder are jointly trained in an unsupervised way.

We will count each layer like convolution and ReLU as a separate layer here, and therefore we have a total of seven layers including the input. This architecture is simplistic because it uses a single convolution layer in each of the encoders and decoders. In more generalized architectures, these layers are stacked to create more powerful architectures. However, it is helpful to illustrate the relationship of the basic operations like unpooling and deconvolution to their encoding counterparts (like pooling and convolution). Another simplification is that the code is contained in a spatial layer, whereas one could also insert fully connected layers in the middle. Although this example (and Figure 8.17(b)) uses a spatial code, the use of fully connected layers in the middle is more useful for practical applications. On the other hand, the spatial layers in the middle can be used for visualization.

Consider a situation in which the encoder uses d_2 square filters of size $F_1 \times F_1 \times d_1$ in the first layer. Also assume that the first layer is a (spatially) square volume of size $L_1 \times L_1 \times d_1$.

The (i, j, k)th entry of the pth filter in the first layer has weight $w_{ijk}^{(p,1)}$. This notations are consistent with those used in Section 8.2, where the convolution operation is defined. It is common to use the precise level of padding required in the convolution layer, so that the feature maps in the second layer are also of size L_1. This level of padding is $F_1 - 1$, which is referred to as *half-padding*. However, it is also possible to use no padding in the convolution layer, if one uses full padding in the corresponding deconvolution layer. In general, the sum of the paddings between the convolution and its corresponding deconvolution layer must sum to $F_1 - 1$ in order to maintain the spatial size of the layer in a convolution-deconvolution pair.

Here, it is important to understand that although each $W^{(p,1)} = [w_{ijk}^{(p,1)}]$ is a 3-dimensional tensor, one can create a 4-dimensional tensor by including the index p in the tensor. The deconvolution operation uses a transposition of this tensor, which is similar to the approach used in backpropagation (cf. Section 8.3.3). The counter-part deconvolution operation occurs from the sixth to the seventh layer (by counting the ReLU/pooling/unpooling layers in the middle). Therefore, we will define the (deconvolution) tensor $U^{(s,6)} = [u_{ijk}^{(s,6)}]$ in relation to $W^{(p,1)}$. Layer 5 contains d_2 feature maps, which were inherited from the convolution operation in the first layer (and unchanged by pooling/unpooling/ReLU operations). These d_2 feature maps need to be mapped into d_1 layers, where the value of d_1 is 3 for RGB color channels. Therefore, the number of filters in the deconvolution layer is equal to the depth of the filters in the convolution layer and vice vera. One can view this change in shape as a result of the transposition and spatial inversion of the 4-dimensional tensor created by the filters. Furthermore, the entries of the two 4-dimensional tensors are related as follows:

$$u_{ijk}^{(s,6)} = w_{rms}^{(k,1)} \quad \forall s \in \{1 \ldots d_1\}, \forall k \in \{1 \ldots d_2\} \tag{8.5}$$

Here, $r = n - i + 1$ and $m = n - j + 1$, where the spatial footprint in the first layer is $n \times n$. Note the transposition of the indices s and k in the above relationship. This relationship is identical to Equation 8.3. It is not necessary to tie the weights in the encoder and decoder, or even use a symmetric architecture between encoder and decoder [310].

The filters $U^{(s,6)}$ in the sixth layer are used just like any other convolution to reconstruct the RGB color channels of the images from the activations in layer 6. Therefore, a deconvolution operation is really a convolution operation, except that it is done with a transposed and spatially inverted filter. As discussed in Section 8.3.2, this type of deconvolution operation is also used in backpropagation. Both the convolution/deconvolution operations can also be executed with the use of matrix multiplications, as described in that section.

The pooling operations, however, irreversibly lose some information and are therefore impossible to invert exactly. This is because the non-maximal values in the layer are permanently lost by pooling. The max-unpooling operation is implemented with the help of *switches*. When pooling is performed, the precise positions of the maximal values are stored. For example, consider the common situation in which 2×2 pooling is performed at stride 2. In such a case, pooling reduces both spatial dimensions by a factor of 2, and it picks the maximum out of $2 \times 2 = 4$ values in each (non-overlapping) pooled region. The exact coordinate of the (maximal) value is stored, and is referred to as the switch. When unpooling, the dimensions are increased by a factor of 2, and the values at the switch positions are copied from the previous layer. The other values are set to 0. Therefore, after max-unpooling, exactly 75% of the entries in the layer will have uncopied values of 0 in the case of non-overlapping 2×2 pooling.

Like traditional autoencoders, the loss function is defined by the reconstruction error over all $L_1 \times L_1 \times d_1$ pixels. Therefore, if $h_{ijk}^{(1)}$ represents the values of the pixels in the first (input) layer, and $h_{ijk}^{(7)}$ represents the values of the pixels in the seventh (output) layer, the reconstruction loss E is defined as follows:

$$E = \sum_{i=1}^{L_1} \sum_{j=1}^{L_1} \sum_{k=1}^{d_1} (h_{ijk}^{(1)} - h_{ijk}^{(7)})^2 \tag{8.6}$$

Other types of error functions (such as L_1-loss and negative log-likelihood) are also used.

One can use traditional backpropagation with the autoencoder. Backpropagating through deconvolutions or the ReLU is no different than in the case of convolutions. In the case of max-unpooling, the gradient flows only through the switches in an unchanged way. Since the parameters of the encoder and the decoder are tied, one needs to sum up the gradients of the matching parameters in the two layers during gradient descent. Another interesting point is that backpropagating through deconvolutions uses almost identical operations to forward propagation through convolutions. This is because both backpropagation and deconvolution cause successive transpositions of the 4-dimensional tensor used for transformation.

This basic autoencoder can easily be extended to the case where multiple convolutions, poolings, and ReLUs are used. The work in [554] discusses the difficulty with multilayer autoencoders, and proposes several tricks to improve performance. There are several other architectural design choices that are often used to improve performance. One key point is that strided convolutions are often used (in lieu of max-pooling) to reduce the spatial footprint in the encoder, which must be balanced in the decoder with *fractionally* strided convolutions. Consider a situation in which the encoder uses a stride of S with some padding to reduce the size of the spatial footprint. In the decoder, one can increase the size of the spatial footprint by the same factor by using the following trick. While performing the convolution, we stretch the input volume by placing $S-1$ rows of zeros[9] between every pair of rows, and $S-1$ columns of zeros between every pair of columns before applying the filter. As a result, the input volume already stretches by a factor of approximately S in each spatial dimension. Additional padding along the borders can be applied before performing the convolution with the transposed filter. Such an approach has the effect of providing a fractional stride and expanding the output size in the decoder. An alternative approach for stretching the input volume of a convolution is to insert interpolated values (instead of zeros) between the original entries of the input volume. The interpolation is done using a convex combination of the nearest four values, and a decreasing function of the distance to each of these values is used as the proportionality factor of the interpolation [449]. The approach of stretching the inputs is sometimes combined with that of stretching the filters as well by inserting zeros within the filter [449]. Stretching the filter results in an approach, referred to as *dilated convolution*, although its use is not universal for fractionally strided convolutions. A detailed discussion of convolution arithmetic (included fractionally strided convolution) is provided in [109]. Compared to the traditional autoencoder, the convolutional autoencoder is somewhat more tricky to implement, with many different variations for better performance. Refer to the bibliographic nodes.

Unsupervised methods also have applications to improving supervised learning. The most obvious method among them is *pretraining*, which was discussed in Section 4.7 of Chapter 4. In convolutional neural networks, the methodology for pretraining is not very different in principle from what is used in traditional neural networks. Pretraining can also

[9]Example available at http://deeplearning.net/software/theano/tutorial/conv_arithmetic.html.

Figure 8.18: Example of image classification/localization in which the class "*fish*" is identified together with its bounding box. The image is illustrative only.

be performed by deriving the weights from a trained *deep-belief convolutional network* [285]. This is analogous to the approach in traditional neural networks, where stacked Boltzmann machines were among the earliest models used for pretraining.

8.6 Applications of Convolutional Networks

Convolutional neural networks have several applications in object detection, localization, video, and text processing. Many of these applications work on the basic principle of using convolutional neural networks to provide engineered features, on top of which multidimensional applications can be constructed. The success of convolutional neural networks remains unmatched by almost any class of neural networks. In recent years, competitive methods have even been proposed for sequence-to-sequence learning, which has traditionally been the domain of recurrent networks.

8.6.1 Content-Based Image Retrieval

In content-based image retrieval, each image is first engineered into a set of multidimensional features by using a pretrained classifier like *AlexNet*. The pretraining is typically done up front using a large data set like *ImageNet*. A huge number of choices of such pretrained classifiers is available at [586]. The features from the fully connected layers of the classifier can be used to create a multidimensional representation of the images. The multidimensional representations of the images can be used in conjunction with any multidimensional retrieval system to provide results of high quality. The use of neural codes for image retrieval is discussed in [16]. The reason that this approach works is because the features extracted from *AlexNet* have semantic significance to the different types of shapes present in the data. As a result, the quality of the retrieval is generally quite high when working with these features.

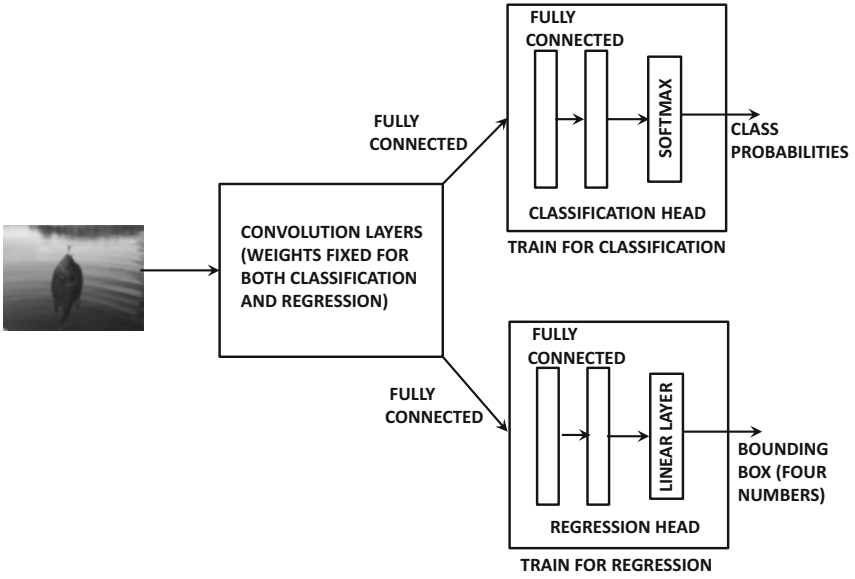


Figure 8.19: The broad framework of classification and localization

8.6.2 Object Localization

In object localization, we have a fixed set of objects in an image, and we would like to identify the rectangular regions in the image in which the object occurs. The basic idea is to take an image with a *fixed* number of objects and encase each of them in a bounding box. In the following, we will consider the simple case in which a single object exists in the image. Image localization is usually integrated with the classification problem, in which we first wish to classify the object in the image and draw a bounding box around it. For simplicity, we consider the case in which there is a single object in the image. We have shown an example of image classification and localization in Figure 8.18, in which the class "*fish*" is identified, and a bounding box is drawn around the portion of the image that delineates that class.

The bounding box of an image can be uniquely identified with four numbers. A common choice is to identify the top-left corner of the bounding box, and the two dimensions of the box. Therefore, one can identify a box with four unique numbers. This is a regression problem with multiple targets. Here, the key is to understand that one can train almost the same model for both classification and regression, which vary only in terms of the final two fully connected layers. This is because the semantic nature of the features extracted from the convolution network are often highly generalizable across a wide variety of tasks. Therefore, one can use the following approach:

1. First, we train a neural network classifier like *AlexNet* or use a pretrained version of this classifier. In the first phase, it suffices to train the classifier only with image-class pairs. One can even use an off-the-shelf pretrained version of the classifier, which was trained on *ImageNet*.

2. The last two fully connected layers and softmax layers are removed. This removed set of layers is referred to as the *classification head*. A new set of two fully connected

Figure 8.20: Example of object detection. Here, four objects are identified together with their bounding boxes. The four objects are "*fish*," "*girl*," "*bucket*," and "*seat*." The image is illustrative only.

layers and a linear regression layer is attached. Only these layers are then trained with training data containing images and their bounding boxes. This new set of layers is referred to as the *regression head*. Note that the weights of the convolution layers are fixed, and are not changed. Both the classification and regression heads are shown in Figure 8.19. Since the classification and regression heads are not connected to one another in any way, these two layers can be trained independently. The convolution layers play the role of creating visual features for both classification and regression.

3. One can optionally fine-tune the convolution layers to be sensitive to both classification and regression (since they were originally trained only for classification). In such a case, both classification and regression heads are attached, and the training data for images, their classes, and bounding boxes are shown to the network. Backpropagation is used to fine-tune all layers. This full architecture is shown in Figure 8.19.

4. The entire network (with both classification and regression heads attached) is then used on the test images. The outputs of the classification head provide the class probabilities, whereas the outputs of the regression head provide the bounding boxes.

One can obtain results of superior quality by using a sliding-window approach. The basic idea in the sliding-window approach is to perform the localization at multiple locations in the image with the use of a sliding window, and then integrate the results of the different runs. An example of this approach is the *Overfeat* method [441]. Refer to the bibliographic notes for pointers to other localization methods.

8.6.3 Object Detection

Object detection is very similar to object localization, except that there is a *variable* number of objects of different classes in the image. In this case, one wishes to identify all the objects in the image together with their classes. We have shown an example of object detection in Figure 8.20, in which there are four objects corresponding to the classes "*fish*," "*girl*," "*bucket*," and "*seat*." The bounding boxes of these classes are also shown in the figure.

Object detection is generally a more difficult problem than that of localization because of the variable number of outputs. In fact, one does not even know a priori how many objects there are in the image. For example, one cannot use the architecture of the previous section, where it is not clear how many classification or regression heads one might attach to the convolutional layers.

The simplest approach to this problem is to use a sliding window approach. In the sliding window approach, one tries all possible bounding boxes in the image, on which the object localization approach is applied to detect a single object. As a result, one might detect different objects in different bounding boxes, or the same object in overlapping bounding boxes. The detections from the different bounding boxes can then be integrated in order to provide the final result. Unfortunately, the approach can be rather expensive. For an image of size $L \times L$, the number of possible bounding boxes is L^4. Note that one would have to perform the classification/regression for each of these L^4 possibilities for each image at test time. This is a problem, because one generally expects the testing times to be modest enough to provide real-time responses.

In order to address this issue *region proposal methods* were advanced. The basic idea of a region proposal method is that it can serve as a general-purpose object detector that merges regions with similar pixels together to create larger regions. Therefore, the region proposal methods are used to first create a set of candidate bounding boxes, and then the object classification/localization method is run in each of them. Note that some candidate regions might not have valid objects, and others might have overlapping objects. These are then used to integrate and identify all the objects in the image. This broader approach has been used in various techniques like *MCG* [172], *EdgeBoxes* [568], and *SelectiveSearch* [501].

8.6.4 Natural Language and Sequence Learning

While the preferred way of machine learning with text sequences is that of recurrent neural networks, the use of convolutional neural networks has become increasingly popular in recent years. At first sight, convolutional neural networks do not seem like a natural fit for text-mining tasks. First, image shapes are interpreted in the same way, irrespective of where they are in the image. This is not quite the case for text, where the position of a word in a sentence seems to matter quite a bit. Second, issues such as position translation and shift cannot be treated in the same way in text data. Neighboring pixels in an image are usually very similar, whereas neighboring words in text are almost never the same. In spite of these differences, the systems based on convolutional networks have shown improved performance in recent years.

Just as an image is represented as a 2-dimensional object with an additional depth dimension defined by the number of color channels, a text sequence is represented as 1-dimensional object with depth defined by its dimensionality of representation. The dimensionality of representation of a text sentence is equal to the lexicon size for the case of one-hot encoding. Therefore, instead of 3-dimensional boxes with a spatial extent and a depth (color channels/feature maps), the filters for text data are 2-dimensional boxes with a window (sequence) length for sliding along the sentence and a depth defined by the lexicon. In later layers of the convolutional network, the depth is defined by the number of feature maps rather than the lexicon size. Furthermore, the number of filters in a given layer defines the number of feature maps in the next layer (as in image data). In image data, one performs convolutions at all 2-dimensional locations, whereas in text data one performs convolutions at all 1-dimensional points in the sentence with the same filter. One challenge

with this approach is that the use of one-hot encoding increases the number of channels, and therefore blows up the number of parameters in the filters in the first layer. The lexicon size of a typical corpus may often be of the order of 10^6. Therefore, various types of pretrained embeddings of words, such as *word2vec* or *GLoVe* [371] are used (cf. Chapter 2) in lieu of the one-hot encodings of the individual words. Such word encodings are semantically rich, and the dimensionality of the representation can be reduced to a few thousand (from a hundred-thousand). This approach can provide an order of magnitude reduction in the number of parameters in the first layer, in addition to providing a semantically rich representation. All other operations (like max-pooling or convolutions) in the case of text data are similar to those of image data.

8.6.5 Video Classification

Videos can be considered generalizations of image data in which a temporal component is inherent to a sequence of images. This type of data can be considered *spatio-temporal data*, which requires us to generalize the 2-dimensional spatial convolutions to 3-dimensional spatio-temporal convolutions. Each frame in a video can be considered an image, and one therefore receives a sequence of images in time. Consider a situation in which each image is of size $224 \times 224 \times 3$, and a total of 10 frames are received. Therefore, the size of the video segment is $224 \times 224 \times 10 \times 3$. Instead of performing spatial convolutions with a 2-dimensional spatial filter (with an additional depth dimension capturing 3 color channels), we perform spatiotemporal convolutions with a 3-dimensional spatiotemporal filter (and a depth dimension capturing the color channels). Here, it is interesting to note that the nature of the filter depends on the data set at hand. A purely sequential data set (e.g., text) requires 1-dimensional convolutions with windows, an image data set requires 2-dimensional convolutions, and a video data set requires 3-dimensional convolutions. We refer to the bibliographic notes for pointers to several papers that use 3-dimensional convolutions for video classification.

An interesting observation is that 3-dimensional convolutions add only a limited amount to what one can achieve by averaging the classifications of individual frames by image classifiers. A part of the problem is that motion adds only a limited amount to the information that is available in the individual frames for classification purposes. Furthermore, sufficiently large video data sets are hard to come by. For example, even a data set containing a million videos is often not sufficient because the amount of data required for 3-dimensional convolutions is much larger than that required for 2-dimensional convolutions. Finally, 3-dimensional convolutional neural networks are good for relatively short segments of video (e.g., half a second), but they might not be so good for longer videos.

For the case of longer videos, it makes sense to combine recurrent neural networks (or LSTMs) with convolutional neural networks. For example, we can use 2-dimensional convolutions over individual frames, but a recurrent network is used to carry over states from one frame to the next. One can also use 3-dimensional convolutional neural networks over short segments of video, and then hook them up with recurrent units. Such an approach helps in identifying actions over longer time horizons. Refer to the bibliographic notes for pointers to methods that combine convolutional and recurrent neural networks.

8.7 Summary

This chapter discusses the use of convolutional neural networks with a primary focus on image processing. These networks are biologically inspired and are among the earliest success stories of the power of neural networks. An important focus of this chapter is the classification problem, although these methods can be used for additional applications such as unsupervised feature learning, object detection, and localization. Convolutional neural networks typically learn hierarchical features in different layers, where the earlier layers learn primitive shapes, whereas the later layers learn more complex shapes. The backpropagation methods for convolutional neural networks are closely related to the problems of deconvolution and visualization. Recently, convolutional neural networks have also been used for text processing, where they have shown competitive performance with recurrent neural networks.

8.8 Bibliographic Notes

The earliest inspiration for convolutional neural networks came from Hubel and Wiesel's experiments with the cat's visual cortex [212]. Based on many of these principles, the notion of the neocognitron was proposed in early work. These ideas were then generalized to the first convolutional network, which was referred to as *LeNet-5* [279]. An early discussion on the best practices and principles of convolutional neural networks may be found in [452]. An excellent overview of convolutional neural networks may be found in [236]. A tutorial on convolution arithmetic is available in [109]. A brief discussion of applications may be found in [283].

The earliest data set that was used popularly for training convolutional neural networks was the MNIST database of handwritten digits [281]. Later, larger datasets like *ImageNet* [581] became more popular. Competitions such as the *ImageNet* challenge (*ILSVRC*) [582] have served as sources of some of the best algorithms over the last five years. Examples of neural networks that have done well at various competitions include *AlexNet* [255], *ZFNet* [556], *VGG* [454], *GoogLeNet* [485], and *ResNet* [184]. The *ResNet* is closely related to highway networks [505], and it provides an iterative view of feature engineering. A useful precursor to *GoogLeNet* was the Network-in-Network (NiN) architecture [297], which illustrated some useful design principles of the inception module (such as the use of bottleneck operations). Several explanations of why *ResNet* works well are provided in [185, 505]. The use of inception modules between skip connections is proposed in [537]. The use of stochastic depth in combination with residual networks is discussed in [210]. Wide residual networks are proposed in [549]. A related architecture, referred to as *FractalNet* [268], uses both short and long paths in the network, but does not use skip connections. Training is done by dropping subpaths in the network, although prediction is done on the full network.

Off-the-shelf feature extraction methods with pretrained models are discussed in [223, 390, 585]. In cases where the nature of the application is very different from *ImageNet* data, it might make sense to extract features only from the lower layers of the pretrained model. This is because lower layers often encode more generic/primitive features like edges and basic shapes, which tend to work across an array of settings. The local-response normalization approach is closely related to the contrast normalization discussed in [221].

The work in [466] proposes that it makes sense to replace the max-pooling layer with a convolutional layer with increased stride. Not using a max-pooling layer is an advantage

in the construction of an autoencoder because one can use a convolutional layer with a fractional stride within the decoder [384]. Fractional strides place zeros within the rows and columns of the input volume, when it is desired to increase the spatial footprint from the convolution operation. The notion of *dilated convolutions* [544] in which zeros are placed within the rows/columns of the filter (instead of input volume) is also sometimes used. The connections between deconvolution networks and gradient-based visualization are discussed in [456, 466]. Simple methods for inverting the features created by a convolutional neural network are discussed in [104]. The work in [308] discuss how to reconstruct an image optimally from a given feature representation. The earliest use the convolutional autoencoder is discussed in [387]. Several variants of the basic autoencoder architecture were proposed in [318, 554, 555]. One can also borrow ideas from restricted Boltzmann machines to perform unsupervised feature learning. One of the earliest such ideas that uses Deep Belief Nets (DBNs) is discussed in [285]. The use of different types of deconvolution, visualization, and reconstruction is discussed in [130, 554, 555, 556]. A very large-scale study for unsupervised feature extraction from images is reported in [270].

There are some ways of learning feature representations in an unsupervised way, which seem to work quite well. The work in [76] clusters on small image patches with a k-means algorithm in order to generate features. The centroids of the clusters can be used to extract features. Another option is use random weights as filters in order to extract features [85, 221, 425]. Some insight on this issue is provided in [425], which shows that a combination of convolution and pooling becomes frequency selective and translation invariant, even with random weights.

A discussion of neural feature engineering for image retrieval is provided in [16]. Numerous methods have been proposed in recent years for image localization. A particularly prominent system in this regard was *Overfeat* [441], which was the winner of the 2013 *ImageNet* competition. This method used a sliding-window approach in order to obtain results of superior quality. Variations of *AlexNet*, *VGG*, and *ResNet* have also done well in the *ImageNet* competition. Some of the earliest methods for object detection were proposed in [87, 117]. The latter is also referred to as the *deformable parts model* [117]. These methods did not use neural networks or deep learning, although some connections have been drawn [163] between deformable parts models and convolutional neural networks. In the deep learning era, numerous methods like *MCG* [172], *EdgeBoxes* [568], and *SelectiveSearch* [501] have been proposed. The main problem with these methods is that they are somewhat slow. Recently, the *Yolo* method, which is a fast object detection method, was proposed in [391]. However, some of the speed gains are at the expense of accuracy. Nevertheless, the overall effectiveness of the method is still quite high. The use of convolutional neural networks for image segmentation is discussed in [180]. Texture synthesis and style transfer methods with convolutional neural networks are proposed in [131, 132, 226]. Tremendous advances have been made in recent years in facial recognition with neural networks. The early work [269, 407] showed how convolutional networks can be used for face recognition. Deep variants are discussed in [367, 474, 475].

Convolutional neural networks for natural language processing are discussed in [78, 79, 102, 227, 240, 517]. These methods often leverage on *word2vec* or GloVe methods to start with a richer set of features [325, 371]. The notion of recurrent and convolutional neural networks has also been combined for text classification [260]. The use of character-level convolutional networks for text classification is discussed in [561]. Methods for image captioning by combining convolutional and recurrent neural networks are discussed in [225, 509]. The use of convolutional neural networks for processing graph-structured data is discussed in [92, 188, 243]. A discussion of the use of convolutional neural networks in time-series and speech is provided [276].

Video data can be considered the spatiotemporal generalization of image data from the perspective of convolutional networks [488]. The use of 3-dimensional convolutional neural networks for large-scale video classification is discussed in [17, 222, 234, 500], and the works in [17, 222] proposed the earliest methods for 3-dimensional convolutional neural networks in video classification. All the neural networks for image classification have natural 3-dimensional counterparts. For example, a generalization of *VGG* to the video domain with 3-dimensional convolutional networks is discussed in [500]. Surprisingly, the results from 3-dimensional convolutional networks are only slightly better than single-frame methods, which perform classifications from individual frames of the video. An important observation is that individual frames already contain a lot of information for classification purposes, and the addition of motion often does not help for classification, unless the motion characteristics are essential for distinguishing classes. Another issue is that the data sets for video classification are often limited in scale compared to what is really required for building large-scale systems. Even though the work in [234] collected a relatively large-scale data set of over a million *YouTube* videos, this scale seems to be insufficient in the context of video processing. After all, video processing requires 3-dimensional convolutions that are far more complex than the 2-dimensional convolutions in image processing. As a result, it is often beneficial to combine hand-crafted features with the convolutional neural network [514]. Another useful feature that has found applicability in recent years is the notion of optical flow [53]. The use of 3-dimensional convolutional neural networks is helpful for classification of videos over shorter time scales. Another common idea for video classification is to combine convolutional neural networks with recurrent neural networks [17, 100, 356, 455]. The work in [17] was the earliest method for combining recurrent and convolutional neural networks. The use of recurrent neural networks is helpful when one has to perform the classification over longer time scales. A recent method [21] combines recurrent and convolutional neural networks in a homogeneous way. The basic idea is to make every neuron in the convolution neural network to be recurrent. One can view this approach to be a direct recurrent extension of convolutional neural networks.

8.8.1 Software Resources and Data Sets

A variety of packages are available for deep learning with convolutional neural networks like *Caffe* [571], *Torch* [572], *Theano* [573], and *TensorFlow* [574]. Extensions of *Caffe* to Python and MATLAB are available. A discussion of feature extraction from *Caffe* may be found in [585]. A "model zoo" of pretrained models from *Caffe* may be found in [586]. *Theano* is Python-based, and it provides high-level packages like *Keras* [575] and *Lasagne* [576] as interfaces. An open-source implementation of convolutional neural networks in MATLAB, referred to as *MatConvNet*, may be found in [503]. The code and parameter files for *AlexNet* are available at [584].

The two most popular data sets for testing convolutional neural networks are *MNIST* and *ImageNet*. Both these data sets are described in detail in Chapter 1. The *MNIST* data set is quite well behaved because its images have been centered and normalized. As a result, the images in *MNIST* can be classified accurately even with conventional machine learning methods, and therefore convolutional neural networks are not necessary. On the other hand, the images in *ImageNet* contain images from different perspectives, and do require convolutional neural networks. Nevertheless, the 1000-category setting of *ImageNet*, together with its large size, makes it a difficult candidate for testing in a computationally efficient way. A more modestly sized data set is *CIFAR-10* [583]. This data set contains only 60,000 instances divided into ten categories, and contains 6,000 color images. Each image

in the data set has size $32 \times 32 \times 3$. It is noteworthy that the *CIFAR-10* data set is a small subset of the *tiny images data set* [642], which originally contains 80 million images. The *CIFAR-10* data set is often used for smaller scale testing, before a more large-scale training is done with *ImageNet*. The *CIFAR-100* data set is just like the *CIFAR-10* data set, except that it has 100 classes, and each class contains 600 instances. The 100 classes are grouped into 10 super-classes.

8.9 Exercises

1. Consider a 1-dimensional time-series with values 2, 1, 3, 4, 7. Perform a convolution with a 1-dimensional filter 1, 0, 1 and zero padding.

2. For a one-dimensional time series of length L and a filter of size F, what is the length of the output? How much padding would you need to keep the output size to a constant value?

3. Consider an activation volume of size $13 \times 13 \times 64$ and a filter of size $3 \times 3 \times 64$. Discuss whether it is possible to perform convolutions with strides 2, 3, 4, and 5. Justify your answer in each case.

4. Work out the sizes of the spatial convolution layers for each of the columns of Table 8.2. In each case, we start with an input image volume of $224 \times 224 \times 3$.

5. Work out the number of parameters in each spatial layer for column D of Table 8.2.

6. Download an implementation of the *AlexNet* architecture from a neural network library of your choice. Train the network on subsets of varying size from the *ImageNet* data, and plot the top-5 error with data size.

7. Compute the convolution of the input volume in the upper-left corner of Figure 8.2 with the horizontal edge detection filter of Figure 8.1(b). Use a stride of 1 without padding.

8. Perform a 4×4 pooling at stride 1 of the input volume in the upper-left corner of Figure 8.4.

9. Discuss the various type of pretraining that one can use in the image captioning application discussed in Section 7.7.1 of Chapter 7.

10. You have a lot of data containing ratings of users for different images. Show how you can combine a convolutional neural network with the collaborative filtering ideas discussed in Chapter 2 to create a hybrid between a collaborative and content-centric recommender system.

Chapter 9

Deep Reinforcement Learning

"The reward of suffering is experience."—Harry S. Truman

9.1 Introduction

Human beings do not learn from a concrete notion of training data. Learning in humans is a continuous experience-driven process in which decisions are made, and the reward/punishment received from the *environment* are used to guide the learning process for future decisions. In other words, learning in intelligent beings is by reward-guided *trial and error*. Furthermore, much of human intelligence and instinct is encoded in genetics, which has evolved over millions of years with another environment-driven process, referred to as *evolution*. Therefore, almost all of biological intelligence, as we know it, originates in one form or other through an interactive process of trial and error with the environment. In his interesting book on artificial intelligence [453], Herbert Simon proposed the *ant hypothesis*:

> "Human beings, viewed as behaving systems, are quite simple. The apparent complexity of our behavior over time is largely a reflection of the complexity of the environment in which we find ourselves."

Human beings are considered simple because they are one-dimensional, selfish, and reward-driven entities (when viewed as a whole), and all of biological intelligence is therefore attributable to this simple fact. Since the goal of artificial intelligence is to simulate biological intelligence, it is therefore natural to draw inspirations from the successes of biological greed in simplifying the design of highly complex learning algorithms.

A reward-driven trial-and-error process, in which a system learns to interact with a complex environment to achieve rewarding outcomes, is referred to in machine learning parlance as *reinforcement learning*. In reinforcement learning, the process of trial and error is driven by the need to maximize the expected rewards over time. Reinforcement learning can be a gateway to the quest for creating truly intelligent *agents* such as game-playing algorithms, self-driving cars, and even intelligent robots that interact with the environment. Simply speaking, it is a gateway to general forms of artificial intelligence. We are not quite there yet. However, we have made huge strides in recent years with exciting results:

1. Deep learners have been trained to play video games by using only the raw pixels of the video console as feedback. A classical example of this setting is the Atari 2600 console, which is a platform supporting multiple games. The input to the deep learner from the Atari platform is the display of pixels from the current state of the game. The reinforcement learning algorithm predicts the actions based on the display and inputs them into the Atari console. Initially, the computer algorithm makes many mistakes, which are reflected in the virtual rewards given by the console. As the learner gains experience from its mistakes, it makes better decisions. This is exactly how humans learn to play video games. The performance of a recent algorithm on the Atari platform has been shown to surpass human-level performance for a large number of games [165, 335, 336, 432]. Video games are excellent test beds for reinforcement learning algorithms, because they can be viewed as highly simplified representations of the choices one has to make in various decision-centric settings. Simply speaking, video games represent toy microcosms of real life.

2. DeepMind has trained a deep learning algorithm *AlphaGo* [445] to play the game of *Go* by using the reward-outcomes in the moves of games drawn from both human and computer self-play. *Go* is a complex game that requires significant human intuition, and the large tree of possibilities (compared to other games like chess) makes it an incredibly difficult candidate for building a game-playing algorithm. *AlphaGo* has not only convincingly defeated all top-ranked *Go* players it has played against [602, 603], but has contributed to innovations in the style of human play by using unconventional strategies in defeating these players. These innovations were a result of the reward-driven experience gained by *AlphaGo* by playing itself over time. Recently, the approach has also been generalized to chess, and it has convincingly defeated one of the top conventional engines [447].

3. In recent years, deep reinforcement learning has been harnessed in self-driving cars by using the feedback from various sensors around the car to make decisions. Although it is more common to use supervised learning (or *imitation learning*) in self-driving cars, the option of using reinforcement learning has always been recognized as a viable possibility [604]. During the course of driving, these cars now consistently make fewer errors than do human beings.

4. The quest for creating self-learning robots is a task in reinforcement learning [286, 296, 432]. For example, robot locomotion turns out to be surprisingly difficult in nimble configurations. Teaching a robot to walk can be couched as a reinforcement learning task, if we do not show a robot what walking looks like. In the reinforcement learning paradigm, we only incentivize the robot to get from point A to point B as efficiently as possible using its available limbs and motors [432]. Through reward-guided trial and error, robots learn to roll, crawl, and eventually walk.

Reinforcement learning is appropriate for tasks *that are simple to evaluate but hard to specify*. For example, it is easy to evaluate a player's performance at the end of a complex game like chess, but it is hard to specify the precise action in every situation. As in biological organisms, reinforcement learning provides a path to the *simplification of learning complex behaviors* by only defining the reward and letting the algorithm learn reward-maximizing behaviors. The complexity of these behaviors is automatically inherited from that of the environment. This is the essence of Herbert Simon's ant hypothesis [453] at the beginning of this chapter. Reinforcement learning systems are inherently *end-to-end systems* in which a complex task is not broken up into smaller components, but viewed through the lens of a simple reward.

The simplest example of a reinforcement learning setting is the *multi-armed bandit problem*, which addresses the problem of a gambler choosing one of many slot machines in order to maximize his payoff. The gambler suspects that the (expected) rewards from the various slot machines are not the same, and therefore it makes sense to play the machine with the largest expected reward. Since the expected payoffs of the slot machines are not known in advance, the gambler has to *explore* different slot machines by playing them and also *exploit* the learned knowledge to maximize the reward. Although exploration of a particular slot machine might gain some additional knowledge about its payoff, it incurs the risk of the (potentially fruitless) cost of playing it. Multi-armed bandit algorithms provide carefully crafted strategies to optimize the trade-off between exploration and exploitation. However, in this simplified setting, each decision of choosing a slot machine is identical to the previous one. This is not quite the case in settings such as video games and self-driving cars with raw sensory inputs (e.g., video game screen or traffic conditions), which define the *state* of the system. Deep learners are excellent at distilling these sensory inputs into *state-sensitive* actions by wrapping their learning process within the exploration/exploitation framework.

Chapter Organization

This chapter is organized as follows. The next section introduces multi-armed bandits, which constitutes one of the simplest stateless settings in reinforcement learning. The notion of states is introduced in Section 9.3. The Q-learning method is introduced in Section 9.4. Policy gradient methods are discussed in Section 9.5. The use of Monte Carlo tree search strategies is discussed in Section 9.6. A number of case studies are discussed in Section 9.7. The safety issues associated with deep reinforcement learning methods are discussed in Section 9.8. A summary is given in Section 9.9.

9.2 Stateless Algorithms: Multi-Armed Bandits

We revisit the problem of a gambler who repeatedly plays slot machines based on previous experience. The gambler suspects that one of the slot machines has a better expected reward than others and attempts to both explore and exploit his experience with the slot machines. Trying the slot machines randomly is wasteful but helps in gaining experience. Trying the slot machines for a very small number of times and then always picking the best machine might lead to solutions that are poor in the long-term. How should one navigate this trade-off between exploration and exploitation? Note that every trial provides the same probabilistically distributed reward as previous trials for a given action, and therefore there is no notion of *state* in such a system. This is a simplified case of traditional reinforcement learning in which the notion of state is important. In a computer video game, moving the

cursor in a particular direction has a reward that heavily depends on the *state* of the video game.

There are a number of strategies that the gambler can use to regulate the trade-off between exploration and exploitation of the search space. In the following, we will briefly describe some of the common strategies used in multi-armed bandit systems. All these methods are instructive because they provide the basic ideas and framework, which are used in generalized settings of reinforcement learning. In fact, some of these stateless algorithms are also used as subroutines in general forms of reinforcement learning. Therefore, it is important to explore this simplified setting.

9.2.1 Naïve Algorithm

In this approach, the gambler plays each machine for a fixed number of trials in the exploration phase. Subsequently, the machine with the highest payoff is used forever in the exploitation phase. Although this approach might seem reasonable at first sight, it has a number of drawbacks. The first problem is that it is hard to determine the number of trials at which one can confidently predict whether a particular slot machine is better than another machine. The process of estimation of payoffs might take a long time, especially in cases where the payoff events are rare compared to non-payoff events. Using many exploratory trials will waste a significant amount of effort on suboptimal strategies. Furthermore, if the wrong strategy is selected in the end, the gambler will use the wrong slot machine forever. Therefore, the approach of fixing a particular strategy forever is unrealistic in real-world problems.

9.2.2 ϵ-Greedy Algorithm

The ϵ-greedy algorithm is designed to use the best strategy as soon as possible, without wasting a significant number of trials. The basic idea is to choose a random slot machine for a fraction ϵ of the trials. These exploratory trials are also chosen at random (with probability ϵ) from all trials, and are therefore fully interleaved with the exploitation trials. In the remaining $(1 - \epsilon)$ fraction of the trials, the slot machine with the best average payoff so far is used. An important advantage of this approach is that one is guaranteed to not be trapped in the wrong strategy forever. Furthermore, since the exploitation stage starts early, one is often likely to use the best strategy a large fraction of the time.

The value of ϵ is an algorithm parameter. For example, in practical settings, one might set $\epsilon = 0.1$, although the best choice of ϵ will vary with the application at hand. It is often difficult to know the best value of ϵ to use in a particular setting. Nevertheless, the value of ϵ needs to be reasonably small in order to gain significant advantages from the exploitation portion of the approach. However, at small values of ϵ it might take a long time to identify the correct slot machine. A common approach is to use *annealing*, in which large values of ϵ are initially used, with the values declining with time.

9.2.3 Upper Bounding Methods

Even though the ϵ-greedy strategy is better than the naïve strategy in dynamic settings, it is still quite inefficient at learning the payoffs of new slot machines. In upper bounding strategies, the gambler does not use the mean payoff of a slot machine. Rather, the gambler takes a more optimistic view of slot machines that have not been tried sufficiently, and therefore uses a slot machine with the best *statistical upper bound* on the payoff. Therefore,

one can consider the upper bound U_i of testing a slot machine i as the sum of expected reward Q_i and one-sided confidence interval length C_i:

$$U_i = Q_i + C_i \tag{9.1}$$

The value of C_i is like a bonus for increased uncertainty about that slot machine in the mind of the gambler. The value C_i is proportional to the standard deviation of the *mean* reward of the tries so far. According to the central limit theorem, this standard deviation is inversely proportional to the square-root of the number of times the slot machine i is tried (under the i.i.d. assumption). One can estimate the mean μ_i and standard deviation σ_i of the ith slot machine and then set C_i to be $K \cdot \sigma_i / \sqrt{n_i}$, where n_i is the number of times the ith slot machine has been tried. Here, K decides the level of confidence interval. Therefore, rarely tested slot machines will tend to have larger upper bounds (because of larger confidence intervals C_i) and will therefore be tried more frequently.

Unlike ϵ-greedy, the trials are no longer divided into two categories of exploration and exploitation; the process of selecting the slot machine with the largest upper bound has the dual effect of encoding both the exploration and exploitation aspects within each trial. One can regulate the trade-off between exploration and exploitation by using a specific level of statistical confidence. The choice of $K = 3$ leads to a 99.99% confidence interval for the upper bound under the Gaussian assumption. In general, increasing K will give large bonuses C_i for uncertainty, thereby causing exploration to comprise a larger proportion of the plays compared to an algorithm with smaller values of K.

9.3 The Basic Framework of Reinforcement Learning

The bandit algorithms of the previous section are stateless. In other words, the decision made at each time stamp has an identical environment, and the actions in the past only affect the knowledge of the agent (not the environment itself). This is not the case in generic reinforcement learning settings like video games or self-driving cars, which have a notion of *state*.

In generic reinforcement learning settings, each action is associated with a reward *in isolation*. While playing a video game, you do not get a reward only because you made a particular move. The reward of a move depends on all the other moves you made in the past, which are incorporated in the *state* of the environment. In a video game or self-driving car, we would need a different way of performing the credit assignment in a particular system state. For example, in a self-driving car, the reward for violently swerving a car in a normal state would be different from that of performing the same action in a state that indicates the danger of a collision. In other words, we need a way to quantify the reward of each action in a way that is specific to a particular system state.

In reinforcement learning, we have an *agent* that interacts with the *environment* with the use of *actions*. For example, the player is the agent in a video game, and moving the joystick in a certain direction in a video game is an action. The environment is the entire set up of the video game itself. These actions change the environment and lead to a new *state*. In a video game, the state represents all the variables describing the current position of the player at a particular point. The environment gives the agent rewards, depending on how well the goals of the learning application are being met. For example, scoring points in a video game is a reward. Note that the rewards may sometimes not be directly associated with a particular action, but with a combination of actions taken some time back. For example, the player might have cleverly positioned a cursor at a particularly convenient

point a few movies back, and actions since then might have had no bearing on the reward. Furthermore, the reward for an action might itself not be deterministic in a particular state (e.g., pulling the lever of a slot machine). *One of the primary goals of reinforcement learning is to identify the inherent values of actions in different states, irrespective of the timing and stochasticity of the reward.*

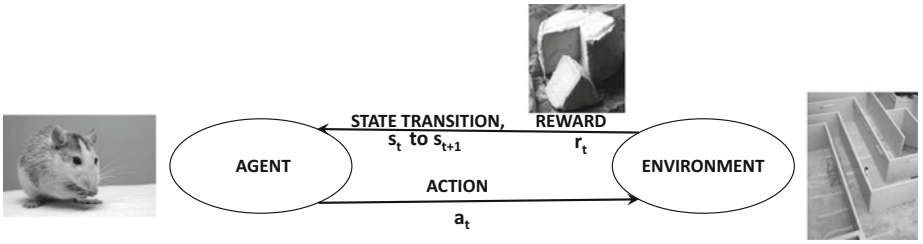


Figure 9.1: The broad framework of reinforcement learning

The learning process helps the agent choose actions based on the inherent values of the actions in different states. This general principle applies to all forms of reinforcement learning in biological organisms, such as a mouse learning a path through a maze to earn a reward. The rewards earned by the mouse depend on an entire sequence of actions, rather than on only the latest action. When a reward is earned, the synaptic weights in the mouse's brain adjust to reflect how sensory inputs should be used to decide future actions in the maze. This is exactly the approach used in deep reinforcement learning, where a neural network is used to predict actions from sensory inputs (e.g., pixels of video game). This relationship between the agent and the environment is shown in Figure 9.1.

The entire set of states and actions and rules for transitioning from one state to another is referred to as a *Markov decision process*. The main property of a Markov decision process is that the state at any particular time stamp encodes all the information needed by the environment to make state transitions and assign rewards based on agent actions. Finite Markov decision processes (e.g., tic-tac-toe) terminate in a finite number of steps, which is referred to as an *episode*. A particular episode of this process is a finite sequence of actions, states, and rewards. An example of length $(n+1)$ is the following:

$$s_0 a_0 r_0 s_1 a_1 r_1 \ldots s_t a_t r_t \ldots s_n a_n r_n$$

Note that s_t is the state *before* performing action a_t, and performing the action a_t causes a reward of r_t and transition to state s_{t+1}. This is the time-stamp convention used throughout this chapter (and several other sources), although the convention in Sutton and Barto's book [483] outputs r_{t+1} in response to action a_t in state s_t (which slightly changes the subscripts in all the results). Infinite Markov decision processes (e.g., continuously working robots) do not have finite length episodes and are referred to as *non-episodic*.

Examples

Although a system state refers to a complete description of the environment, many practical approximations are often made. For example, in an Atari video game, the system state might be defined by a fixed-length window of game snapshots. Some examples are as follows:

1. *Game of tic-tac-toe, chess, or Go:* The state is the position of the board at any point, and the actions correspond to the moves made by the agent. The reward is $+1$, 0, or -1 (depending on win, draw, or loss), *which is received at the end of the game*. Note that rewards are often not received immediately after strategically astute actions.

2. *Robot locomotion:* The state corresponds to the current configuration of robot joints and its position. The actions correspond to the torques applied to robot joints. The reward at each time stamp is a function of whether the robot stays upright and the amount of forward movement from point A to point B.

3. *Self-driving car:* The states correspond to the sensor inputs from the car, and the actions correspond to the steering, acceleration, and braking choices. The reward is a hand-crafted function of car progress and safety.

Some effort usually needs to be invested in defining the state representations and corresponding rewards. However, once these choices have been made, reinforcement learning frameworks are end-to-end systems.

9.3.1 Challenges of Reinforcement Learning

Reinforcement learning is more difficult than traditional forms of supervised learning for the following reasons:

1. When a reward is received (e.g., winning a game of chess), it is not exactly known how much each action has contributed to that reward. This problem lies at the heart of reinforcement learning, and is referred to as the *credit-assignment problem*. Furthermore, rewards may be probabilistic (e.g., pulling the lever of a slot machine), which can only be *estimated* approximately in a data-driven manner.

2. The reinforcement learning system might have a very large number of states (such as the number of possible positions in a board game), and must be able to make sensible decisions in states it has not seen before. This task of model generalization is the primary function of deep learning.

3. A specific choice of action affects the collected data in regard to future actions. As in multi-armed bandits, there is a natural trade-off between exploration and exploitation. If actions are taken only to learn their reward, then it incurs a cost to the player. On the other hand, sticking to known actions might result in suboptimal decisions.

4. Reinforcement learning merges the notion of data collection with learning. Realistic simulations of large physical systems such as robots and self-driving cars are limited by the need to physically perform these tasks and gather responses to actions in the presence of the practical dangers of failures. In many cases, the early portion of learning in a task may have few successes and many failures. *The inability to gather sufficient data in real settings beyond simulated and game-centric environments is arguably the single largest challenge to reinforcement learning.*

In the following sections, we will introduce a simple reinforcement learning algorithm and discuss the role of deep learning methods.

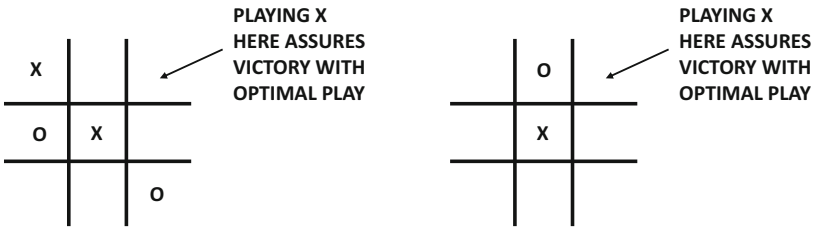
9.3.2 Simple Reinforcement Learning for Tic-Tac-Toe

One can generalize the stateless ϵ-greedy algorithm in the previous section to learn to play the game of tic-tac-toe. In this case, each board position is a state, and the action corresponds to placing 'X' or 'O' at a valid position. The number of valid states of the 3×3 board is bounded above by $3^9 = 19683$, which corresponds to three possibilities ('X', 'O', and blank) for each of 9 positions. Instead of estimating the value of each (stateless) action in multi-armed bandits, we now estimate the value of each state-action *pair* (s, a) based on the historical performance of action a in state s against a fixed opponent. Shorter wins are preferred at discount factor $\gamma < 1$, and therefore the *unnormalized* value of action a in state s is increased with γ^{r-1} in case of wins and $-\gamma^{r-1}$ in case of losses after r moves (including the current move). Draws are credited with 0. The discount also reflects the fact that the significance of an action decays with time in real-world settings. In this case, the table is updated only after all moves are made for a game (although later methods in this chapter allow *online* updates after each move). The normalized values of the actions in the table are obtained by dividing the unnormalized values with the number of times the state-action pair was updated (which is maintained separately). The table starts with small random values, and the action a in state s is chosen greedily to be the action with the highest normalized value with probability $1 - \epsilon$, and is chosen to be a random action otherwise. All moves in a game are credited after the termination of each game. Over time, the values of all state-action pairs will be learned and the resulting moves will also adapt to the play of the fixed opponent. Furthermore, one can even use self-play to generate these tables optimally. When self-play is used, the table is updated from a value in $\{-\gamma^r, 0, \gamma^r\}$ depending on win/draw/loss *from the perspective of the player for whom moves are made.* At inference time, the move with the highest normalized value from the perspective of the player are made.

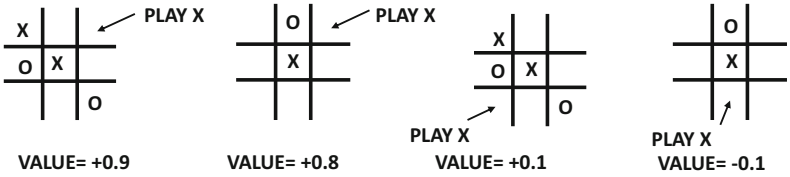
9.3.3 Role of Deep Learning and a Straw-Man Algorithm

The aforementioned algorithm for tic-tac-toe did not use neural networks or deep learning, and this is also the case in many traditional algorithms for reinforcement learning [483]. The overarching goal of the ϵ-greedy algorithm for tic-tac-toe was to learn the inherent *long-term* value of each state-action pair, since the rewards are received long after valuable actions are performed. The goal of the training process is to perform the *value discovery* task of identifying which actions are truly beneficial in the long-term at a particular state. For example, making a clever move in tic-tac-toe might set a trap, which eventually results in assured victory. Examples of two such scenarios are shown in Figure 9.2(a) (although the trap on the right is somewhat less obvious). Therefore, one needs to credit a *strategically* good move favorably in the table of state-action pairs and not just the final winning move. The trial-and-error technique based on the ϵ-greedy method of Section 9.3.2 will indeed assign high values to clever traps. Examples of typical values from such a table are shown in Figure 9.2(b). Note that the less obvious trap of Figure 9.2(a) has a slightly lower value because moves assuring wins after longer periods are discounted by γ, and ϵ-greedy trial-and-error might have a harder time finding the win after setting the trap.

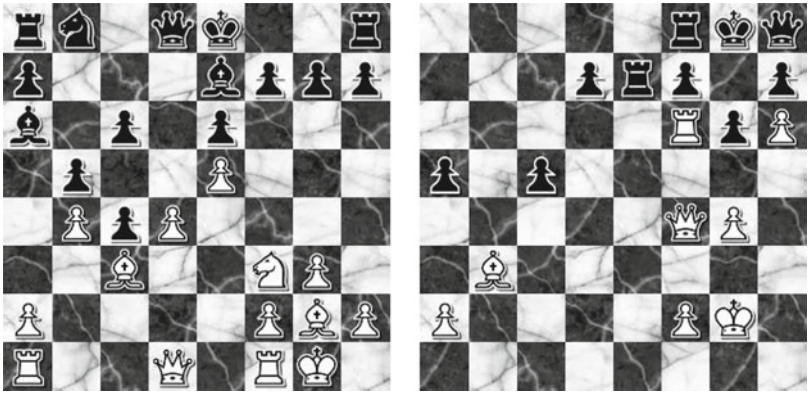
The main problem with this approach is that the number of states in many reinforcement learning settings is too large to tabulate explicitly. For example, the number of possible states in a game of chess is so large that the set of all known positions by humanity is



(a) Two examples from tic-tac-toe assuring victory down the road.



(b) Four entries from the table of state-action values in tic-tac-toe. Trial-and-error learns that moves assuring victory have high value.



(c) Positions from two different games between *Alpha Zero* (white) and *Stockfish* (black) [447]: On the left, white sacrifices a pawn and concedes a passed pawn in order to trap black's light-square bishop behind black's own pawns. This strategy eventually resulted in a victory for white after many more moves than the horizon of a conventional chess-playing program like *Stockfish*. In the second game on the right, white has sacrificed material to incrementally cramp black to a position where all moves worsen the position. Incrementally improving positional advantage is the hallmark of the very best human players rather than chess-playing software like *Stockfish*, whose hand-crafted evaluations sometimes fail to accurately capture subtle differences in positions. The neural network in reinforcement learning, which uses the board state as input, evaluates positions in an integrated way without any prior assumptions. The data generated by trial-and-error provides the only experience for training a very complex evaluation function that is indirectly encoded within the parameters of the neural network. The trained network can therefore *generalize* these learned experiences to new positions. This is similar to how humans learn from previous games to better evaluate board positions.

Figure 9.2: Deep learners are needed for large state spaces like (c).

a minuscule fraction of the valid positions. In fact, the algorithm of Section 9.3.2 is a refined form of *rote learning* in which Monte Carlo simulations are used to refine and remember the long-term values of *seen* states. One learns about the value of a trap in tic-tac-toe only because previous Monte Carlo simulations have experienced victory many times *from that exact board position*. In most challenging settings like chess, one must *generalize* knowledge learned from prior experiences to a state that the learner has not seen before. All forms of learning (including reinforcement learning) are most useful when they are used to generalize known experiences to unknown situations. In such cases, the table-centric forms of reinforcement learning are woefully inadequate. Deep learning models serve the role of *function approximators*. Instead of learning and *tabulating* the values of all moves in all positions (using reward-driven trial and error), one learns the value of each move as a *function* of the input state, based on a *trained model* using the outcomes of prior positions. Without this approach, reinforcement learning cannot be used beyond toy settings like tic-tac-toe.

For example, a straw-man (but not very good) algorithm for chess might use the same ϵ-greedy algorithm of Section 9.3.2, but the values of actions are computed by using the board state as input to a convolutional neural network. The output is the evaluation of the board position. The ϵ-greedy algorithm is simulated to termination with the output values, and the discounted ground-truth value of each move in the simulation is selected from the set $\{\gamma^{r-1}, 0, -\gamma^{r-1}\}$ depending on win/draw/loss and number of moves r to game completion (including the current move). Instead of updating a table of state-action pairs, the parameters of the neural network are updated by treating each move as a training point. The board position is input, and the output of the neural network is compared with the ground-truth value from $\{\gamma^{r-1}, 0, -\gamma^{r-1}\}$ to update the parameters. At inference time, the move with the best output score (with some minimax lookahead) can be used.

Although the aforementioned approach is too naive, a sophisticated system with Monte Carlo tree search, known as *Alpha Zero*, has recently been trained [447] to play chess. Two examples of positions [447] from different games in the match between *Alpha Zero* and a conventional chess program, *Stockfish-8.0*, are provided in Figure 9.2(c). In the chess position on the left, the reinforcement learning system makes a *strategically* astute move of cramping the opponent's bishop at the expense of immediate material loss, which most hand-crafted computer evaluations would not prefer. In the position on the right, *Alpha Zero* has sacrificed two pawns and a piece exchange in order to incrementally constrict black to a point where all its pieces are completely paralyzed. Even though *Alpha Zero* (probably) never encountered these specific positions during training, its deep learner has the ability to extract relevant features and patterns from previous trial-and-error experience in other board positions. In this particular case, the neural network seems to recognize the primacy of spatial patterns representing subtle positional factors over tangible material factors (much like a human's neural network).

In real-life settings, states are often described using sensory inputs. The deep learner uses this input representation of the state to learn the values of specific actions (e.g., making a move in a game) in lieu of the table of state-action pairs. Even when the input representation of the state (e.g., pixels) is quite primitive, neural networks are masters at squeezing out the relevant insights. This is similar to the approach used by humans to process primitive sensory inputs to define the *state* of the world and make decisions about *actions* using our biological neural network. We do not have a table of pre-memorized state-action pairs for every possible real-life situation. The deep-learning paradigm converts the forbiddingly large table of state-action values into a parameterized model mapping states-action pairs to values, which can be trained easily with backpropagation.

9.4 Bootstrapping for Value Function Learning

The simple generalization of the ϵ-greedy algorithm to tic-tac-toe (cf. Section 9.3.2) is a rather naive approach that does not work for *non-episodic settings*. In episodic settings like tic-tac-toe, a fixed-length sequence of at most nine moves can be used to characterize the full and final reward. In non-episodic settings like robots, the Markov decision process may not be finite or might be very long. Creating a sample of the ground-truth reward by Monte Carlo sampling becomes difficult and *online* updating might be desirable. This is achieved with the methodology of *bootstrapping*.

Intuition 9.4.1 (Bootstrapping) *Consider a Markov decision process in which we are predicting values (e.g., long-term rewards) at each time-stamp. We do not need the ground-truth at each time-stamp, as long as we can use a partial simulation of the future to improve the prediction at the current time-stamp. This improved prediction can be used as the ground-truth at the current time stamp for a model without knowledge of the future.*

For example, Samuel's checkers program [421] used the difference in evaluation at the current position and the minimax evaluation obtained by looking several moves ahead with the same function as a "prediction error" in order to update the evaluation function. The idea is that the minimax evaluation from looking ahead is stronger than the one without lookahead and can therefore be used as a "ground truth" to compute the error.

Consider a Markov decision process with the following sequence of states, actions, and rewards:

$$s_0 a_0 r_0 s_1 a_1 r_1 \ldots s_t a_t r_t \ldots$$

For example, in a video game, each state s_t might represent a historical window of pixels [335] with a feature representation $\overline{X}_t$. In order to account for the (possibly) delayed rewards of actions, the cumulative reward R_t at time t is given by the discounted sum of the immediate rewards $r_t, r_{t+1}, r_{t+2}, \ldots r_\infty$ at all future time stamps:

$$R_t = r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + \gamma^3 \cdot r_{t+3} \ldots = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{9.2}$$

The discount factor $\gamma \in (0, 1)$ regulates how myopic we want to be in allocating rewards. The value of γ is less than 1 because future rewards are worth less than immediate rewards. Choosing $\gamma = 0$ will result in myopically setting the full reward R_t to r_t and nothing else. Therefore, it will be impossible to learn a long-term trap in tic-tac-toe. Values of γ that are too close to 1 will result in modeling instability for very long Markov decision processes.

The *Q-function* or *Q-value* for the state-action pair (s_t, a_t) is denoted by $Q(s_t, a_t)$, and is a measure of the *inherent* (i.e., long-term) value of performing the action a_t in state s_t. The Q-function $Q(s_t, a_t)$ represents the best possible reward obtained till the end of the game on performing the action a_t in state s_t. In other words, $Q(s_t, a_t)$ is equal to $\max\{E[R_{t+1}|a_t]\}$. Therefore, if A is the set of all possible actions, then the chosen action at time t is given by the action a_t^* that maximizes $Q(s_t, a_t)$. In other words, we have:

$$a_t^* = \text{argmax}_{a_t \in A} Q(s_t, a_t) \tag{9.3}$$

This predicted action is a good choice for the next move, although it is often combined with an exploratory component (e.g., ϵ-greedy policy) to improve long-term training outcomes.

9.4.1 Deep Learning Models as Function Approximators

For ease in discussion, we will work with the Atari setting [335] in which a fixed window of the last few snapshots of pixels provides the state s_t. Assume that the feature representation of s_t is denoted by $\overline{X}_t$. The neural network uses $\overline{X}_t$ as the input and outputs $Q(s_t, a)$ for each possible legal action a from the universe of actions denoted by the set A.

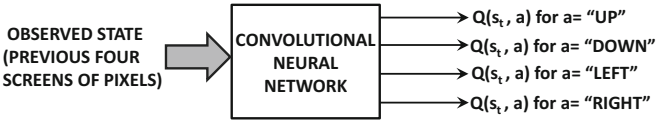


Figure 9.3: The Q-Network for the Atari video game setting

Assume that the neural network is parameterized by the vector of weights $\overline{W}$, and it has $|A|$ outputs containing the Q-values corresponding to the various actions in A. In other words, for each action $a \in A$, the neural network is able to compute the function $F(\overline{X}_t, \overline{W}, a)$, which is defined to be the *learned estimate* of $Q(s_t, a)$:

$$F(\overline{X}_t, \overline{W}, a) = \hat{Q}(s_t, a) \tag{9.4}$$

Note the circumflex on top of the Q-function in order to indicate that it is a predicted value using the learned parameters $\overline{W}$. Learning $\overline{W}$ is the key to using the model for deciding which action to use at a particular time-stamp. For example, consider a video game in which the possible moves are up, down, left, and right. In such a case, the neural network will have four outputs as shown in Figure 9.3. In the specific case of the Atari 2600 games, the input contains $m = 4$ spatial pixel maps in grayscale, representing the window of the last m moves [335, 336]. A convolutional neural network is used to convert pixels into Q-values. This network is referred to as a *Q-network*. We will provide more details of the specifics of the architecture later.

The Q-Learning Algorithm

The weights $\overline{W}$ of the neural network need to be learned via training. Here, we encounter an interesting problem. We can learn the vector of weights only if we have *observed* values of the Q-function. With observed values of the Q-function, we could easily set up a loss in terms of $Q(s_t, a) - \hat{Q}(s_t, a)$ in order to perform the learning after each action. The problem is that the Q-function represents the maximum discounted reward over all *future* combinations of actions, and there is no way of observing it at the current time.

Here, there is an interesting trick for setting up the neural network loss function. According to Intuition 9.4.1, *we do not really need the observed Q-values in order to set up a loss function as long as we know an improved estimate of the Q-values by using partial knowledge from the future.* Then, we can use this improved estimate to create a surrogate "observed" value. This "observed" value is defined by the *Bellman equation* [26], which is a dynamic programming relationship satisfied by the Q-function, and the partial knowledge is the reward observed at the current time-stamp for each action. According to the Bellman equation, we set the "ground-truth" by looking ahead one step and predicting at s_{t+1}:

$$Q(s_t, a_t) = r_t + \gamma \max_a \hat{Q}(s_{t+1}, a) \tag{9.5}$$

The correctness of this relationship follows from the fact that the Q-function is designed to maximize the discounted future payoff. We are essentially looking at all actions one step

ahead in order to create an improved estimate of $Q(s_t, a_t)$. It is important to set $\hat{Q}(s_{t+1}, a)$ to 0 in case the process terminates after performing a_t for episodic sequences. We can write this relationship in terms of our neural network predictions as well:

$$F(\overline{X}_t, \overline{W}, a_t) = r_t + \gamma \text{max}_a F(\overline{X}_{t+1}, \overline{W}, a) \tag{9.6}$$

Note that one must first wait to observe the state $\overline{X}_{t+1}$ and reward r_t by performing the action a_t, before we can compute the "observed" value at time-stamp t on the right-hand side of the above equation. This provides a natural way to express the loss L_t of the neural network at time stamp t by comparing the (surrogate) observed value to the predicted value at time stamp t:

$$L_t = \left\{ \underbrace{[r_t + \gamma \text{max}_a F(\overline{X}_{t+1}, \overline{W}, a)]}_{\text{Treat as constant ground-truth}} - F(\overline{X}_t, \overline{W}, a_t) \right\}^2 \tag{9.7}$$

Therefore, we can now update the vector of weights $\overline{W}$ using backpropagation on this loss function. Here, it is important to note that the target values estimated using the inputs at $(t+1)$ are treated as constant ground-truths by the backpropagation algorithm. Therefore, the derivative of the loss function will treat these estimated values as constants, even though they were obtained from the parameterized neural network with input $\overline{X}_{t+1}$. Not treating $F(\overline{X}_{t+1}, \overline{W}, a)$ as a constant will lead to poor results. This is because we are treating the prediction at $(t+1)$ as an improved estimate of the ground-truth (based on the bootstrapping principle). Therefore, the backpropagation algorithm will compute the following:

$$\overline{W} \Leftarrow \overline{W} + \alpha \left\{ \underbrace{[r_t + \gamma \text{max}_a F(\overline{X}_{t+1}, \overline{W}, a)]}_{\text{Treat as constant ground-truth}} - F(\overline{X}_t, \overline{W}, a_t) \right\} \frac{\partial F(\overline{X}_t, \overline{W}, a_t)}{\partial \overline{W}} \tag{9.8}$$

In matrix-calculus notation, the partial derivative of a function $F()$ with respect to the vector $\overline{W}$ is essentially the gradient $\nabla_{\overline{W}} F$. At the beginning of the process, the Q-values estimated by the neural network are random because the vector of weights $\overline{W}$ is initialized randomly. However, the estimation gradually becomes more accurate with time, as the weights are constantly changed to maximize rewards.

Therefore, at any given time-stamp t at which action a_t and reward r_t has been observed, the following training process is used for updating the weights $\overline{W}$:

1. Perform a forward pass through the network with input $\overline{X}_{t+1}$ to compute $\hat{Q}_{t+1} = \text{max}_a F(\overline{X}_{t+1}, \overline{W}, a)$. The value is 0 in case of termination after performing a_t. *Treating the terminal state specially is important.* According to the Bellman equations, the Q-value at previous time-stamp t should be $r_t + \gamma \hat{Q}_{t+1}$ for observed action a_t at time t. Therefore, instead of using observed values of the target, we have created a *surrogate* for the target value at time t, and we pretend that this surrogate is an observed value given to us.

2. Perform a forward pass through the network with input $\overline{X}_t$ to compute $F(\overline{X}_t, \overline{W}, a_t)$.

3. Set up a loss function in $L_t = (r_t + \gamma Q_{t+1} - F(\overline{X}_t, \overline{W}, a_t))^2$, and backpropagate in the network with input $\overline{X}_t$. Note that this loss is associated with neural network output node corresponding to action a_t, and the loss for all other actions is 0.

4. One can now use backpropagation on this loss function in order to update the weight vector $\overline{W}$. Even though the term $r_t + \gamma Q_{t+1}$ in the loss function is also obtained as a prediction from input $\overline{X}_{t+1}$ to the neural network, it is treated as a (constant) observed value during gradient computation by the backpropagation algorithm.

Both the training and the prediction are performed simultaneously, as the values of actions are used to update the weights and select the next action. It is tempting to select the action with the largest Q-value as the relevant prediction. However, such an approach might perform inadequate exploration of the search space. Therefore, one couples the optimality prediction with a policy such as the ϵ-greedy algorithm in order to select the next action. The action with the largest predicted payoff is selected with probability $(1 - \epsilon)$. Otherwise, a random action is selected. The value of ϵ can be annealed by starting with large values and reducing them over time. Therefore, the *target prediction value* for the neural network is computed using the best possible action in the Bellman equation (which might eventually be different from observed action a_{t+1} based on the ϵ-greedy policy). This is the reason that Q-learning is referred to as an *off-policy algorithm* in which the target prediction values for the neural network update are computed using actions that might be different from the actually observed actions in the future.

There are several modifications to this basic approach in order to make the learning more stable. Many of these are presented in the context of the Atari video game setting [335]. First, presenting the training examples *exactly* in the sequence they occur can lead to local minima because of the strong similarity among training examples. Therefore, a fixed-length history of actions/rewards is used as a pool. One can view this as a history of experiences. Multiple experiences are sampled from this pool to perform mini-batch gradient descent. In general, it is possible to sample the same action multiple times, which leads to greater efficiency in leveraging the learning data. Note that the pool is updated over time as old actions drop out of the pool and newer ones are added. Therefore, the training is still temporal in an approximate sense, but not strictly so. This approach is referred to as *experience replay*, as experiences are replayed multiple times in a somewhat different order than the original actions.

Another modification is that the network used for estimating the target Q-values with Bellman equations (step 1 above) is not the same as the network used for predicting Q-values (step 2 above). The network used for estimating the target Q-values is updated more slowly in order to encourage stability. Finally, one problem with these systems is the sparsity of the rewards, especially at the initial stage of the learning when the moves are random. For such cases, a variety of tricks such as *prioritized experience replay* [428] can be used. The basic idea is to make more efficient use of the training data collected during reinforcement learning by prioritizing actions from which more can be learned.

9.4.2 Example: Neural Network for Atari Setting

For the convolutional neural network [335, 336], the screen sizes were set to 84×84 pixels, which also defined the spatial footprints of the first layer in the convolutional network. The input was in grayscale, and therefore each screen required only a single spatial feature map, although a depth of 4 was required in the input layer to represent the previous four windows of pixels. Three convolutional layers were used with filters of size 8×8, 4×4, and 3×3, respectively. A total of 32 filters were used in the first convolutional layer, and 64 filters were used in each of the other two, with the strides used for convolution being 4, 2,

and 1, respectively. The convolutional layers were followed by two fully connected layers. The number of neurons in the penultimate layer was equal to 512, and that in the final layer was equal to the number of outputs (possible actions). The number of output layers varied between 4 and 18, and was game-specific. The overall architecture of the convolutional network is illustrated in Figure 9.4.

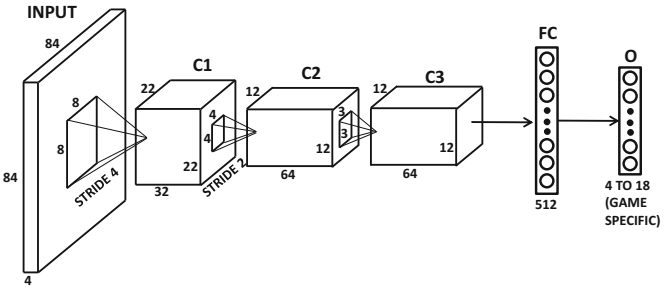


Figure 9.4: The convolutional neural network for the Atari setting

All hidden layers used the ReLU activation, and the output used linear activation in order to predict the real-valued Q-value. No pooling was used, and the strides in the convolution provided spatial compression. The Atari platform supports many games, and the same broader architecture was used across different games in order to showcase its generalizability. There was some variation in performance across different games, although human performance was exceeded in many cases. The algorithm faced the greatest challenges in games in which longer-term strategies were required. Nevertheless, the robust performance of a relatively homogeneous framework across many games was encouraging.

9.4.3 On-Policy Versus Off-Policy Methods: SARSA

The Q-Learning methodology belongs to the class of methods, referred to as *temporal difference learning*. In Q-learning, the actions are chosen according to an ϵ-greedy policy. However, the parameters of the neural network are updated based on the best possible action at each step with the Bellman equation. The best possible action at each step is not quite the same as the ϵ-greedy policy used to perform the simulation. Therefore, Q-learning is an *off-policy reinforcement learning method*. Choosing a different policy for executing actions from those for performing updates does not worsen the ability to find the optimum solutions that are goals of the updates. In fact, since more exploration is performed with a randomized policy, local optima are avoided.

In *on-policy methods*, the actions are consistent with the updates, and therefore the updates can be viewed as policy *evaluation* rather than *optimization*. In order to understand this point, we will describe the updates for the SARSA (State-Action-Reward-State-Action) algorithm, in which the optimal reward in the next step is not used for computing updates. Rather, the next step is updated using the same ϵ-greedy policy to obtain the action a_{t+1} for computing the target values. Then, the loss function for the next step is defined as follows:

$$L_t = \left\{r_t + \gamma F(\overline{X}_{t+1}, \overline{W}, a_{t+1}) - F(\overline{X}_t, \overline{W}, a_t)\right\}^2 \tag{9.9}$$

The function $F(\cdot, \cdot, \cdot)$ is defined in the same way as the previous section. The weight vector is updated based on this loss, and then the action a_{t+1} is executed:

$$\overline{W} \Leftarrow \overline{W} + \alpha \left\{ \underbrace{[r_t + \gamma F(\overline{X}_{t+1}, \overline{W}, a_{t+1})]}_{\text{Treat as constant ground-truth}} - F(\overline{X}_t, \overline{W}, a_t) \right\} \frac{\partial F(\overline{X}_t, \overline{W}, a_t)}{\partial \overline{W}} \tag{9.10}$$

Here, it is instructive to compare this update with those used in Q-learning according to Equation 9.8. In Q-learning, one is using the *best possible* action at each state in order to update the parameters, even though the policy that is actually executed might be ϵ-greedy (which encourages exploration). In SARSA, we are using the action that was actually selected by the ϵ-greedy method in order to perform the update. Therefore, the approach is an *on-policy method*. Off-policy methods like Q-learning are able to decouple exploration from exploitation, whereas on-policy methods are not. Note that if we set the value of ϵ in the ϵ-greedy policy to 0 (i.e., vanilla greedy), then both Q-Learning and SARSA would specialize to the same algorithm. However, such an approach would not work very well because there is no exploration. SARSA is useful when learning cannot be done separately from prediction. Q-learning is useful when the learning can to be done offline, which is followed by exploitation of the learned policy with a vanilla-greedy method at $\epsilon = 0$ (and no need for further model updates). Using ϵ-greedy at inference time would be dangerous in Q-learning, because the policy never pays for its exploratory component and therefore does not learn how to keep exploration safe. For example, a Q-learning based robot will take the shortest path to get from point A to point B even if it is along the edge of the cliff, whereas a SARSA-trained robot will not.

Learning Without Function Approximators

It is possible to also learn Q-values without using function approximators *in cases where the state-space is very small*. For example, in a toy game like tic-tac-toe, one can learn $Q(s_t, a_t)$ explicitly by using trial-and-error play against a strong opponent. In this case, the Bellman equations (cf. Equation 9.5) are used at each move to update an *array* containing the explicit value of $Q(s_t, a_t)$. Using Equation 9.5 directly is too aggressive. More generally, gentle updates are performed for learning rate $\alpha < 1$:

$$Q(s_t, a_t) \Leftarrow Q(s_t, a_t)(1 - \alpha) + \alpha(r_t + \gamma \text{max}_a Q(s_{t+1}, a)) \tag{9.11}$$

Using $\alpha = 1$ will result in Equation 9.5. Updating the array continually will result in a table containing the correct *strategic* value of each move; see, for example, Figure 9.2(a) for an understanding of the notion of strategic value. Figure 9.2(b) contains examples of four entries from such a table.

One can also use the SARSA algorithm without function approximators by using the action a_{t+1} based on the ϵ-greedy policy. We use a superscript p in $Q^p(\cdot, \cdot)$ to indicate that it is a policy evaluation operator of the policy p (which is ϵ-greedy in this case):

$$Q^p(s_t, a_t) \Leftarrow Q^p(s_t, a_t)(1 - \alpha) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1})) \tag{9.12}$$

This approach is a more sophisticated alternative to the ϵ-greedy method discussed in Section 9.3.2. Note that if action a_t at state s_t leads to termination (for episodic processes), then $Q^p(s_t, a_t)$ is simply set to r_t.

9.4.4 Modeling States Versus State-Action Pairs

A minor variation of the theme in the previous sections is to learn the value of a particular state (rather than state-action pair). One can implement all the methods discussed earlier by maintaining values of states rather than state-action pairs. For example, SARSA can be implemented by evaluating all the values of states resulting from each possible action and selecting a good one based on a pre-defined policy like ϵ-greedy. In fact, the earliest methods for temporal difference learning (or *TD-learning*) maintained values on states rather than state-action pairs. From an efficiency perspective, it is more convenient to output the values of all actions in one shot (rather than repeatedly evaluate each forward state) for value-based decision making. Working with state values rather that state-action pairs becomes useful only when the policy cannot be expressed neatly in terms of state-action pairs. For example, we might evaluate a forward-looking tree of promising moves in chess, and report some averaged value for bootstrapping. In such cases, it is desirable to evaluate states rather than state-action pairs. This section will therefore discuss a variation of temporal difference learning in which states are directly evaluated.

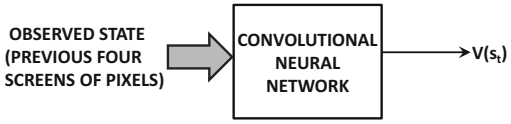


Figure 9.5: Estimating the value of a state with temporal difference learning

Let the value of the state s_t be denoted by $V(s_t)$. Now assume that you have a parameterized neural network that uses the observed attributes $\overline{X}_t$ (e.g., pixels of last four screens in Atari game) of state s_t to estimate $V(s_t)$. An example of this neural network is shown in Figure 9.5. Then, if the function computed by the neural network is $G(\overline{X}_t, \overline{W})$ with parameter vector $\overline{W}$, we have the following:

$$G(\overline{X}_t, \overline{W}) = \hat{V}(s_t) \tag{9.13}$$

Note that the policy being followed to decide the actions might use some arbitrary evaluation of forward-looking states to decide actions. For now, we will assume that we have some reasonable heuristic policy for choosing the actions that uses the forward-looking state values in some way. For example, if we evaluate each forward state resulting from an action and select one of them based on a pre-defined policy (e.g., ϵ-greedy), the approach discussed below is the same as SARSA.

If the action a_t is performed with reward r_t, the resulting state is s_{t+1} with value $V(s_{t+1})$. Therefore, the bootstrapped ground-truth estimate for $V(s_t)$ can be obtained with the help of this lookahead:

$$V(s_t) = r_t + \gamma V(s_{t+1}) \tag{9.14}$$

This estimate can also be stated in terms of the neural network parameters:

$$G(\overline{X}_t, \overline{W}) = r_t + \gamma G(\overline{X}_{t+1}, \overline{W}) \tag{9.15}$$

During the training phase, one needs to shift the weights so as to push $G(\overline{X}_t, \overline{W})$ towards the improved "ground truth" value of $r_t + \gamma G(\overline{X}_{t+1}, \overline{W})$. As in the case of Q-learning, we work with the boot-strapping pretension that the value $r_t + \gamma G(\overline{X}_{t+1}, \overline{W})$ is an observed value given to us. Therefore, we want to minimize the *TD-error* defined by the following:

$$\delta_t = \underbrace{r_t + \gamma G(\overline{X}_{t+1}, \overline{W})}_{\text{"Observed" value}} - G(\overline{X}_t, \overline{W}) \tag{9.16}$$

Therefore, the loss function L_t is defined as follows:

$$L_t = \delta_t^2 = \left\{ \underbrace{r_t + \gamma G(\overline{X}_{t+1}, \overline{W})}_{\text{"Observed" value}} - G(\overline{X}_t, \overline{W}) \right\}^2 \tag{9.17}$$

As in Q-learning, one would first compute the "observed" value of the state at time stamp t using the input $\overline{X}_{t+1}$ into the neural network to compute $r_t + \gamma G(\overline{X}_{t+1}, \overline{W})$. Therefore, one would have to wait till the action a_t has been observed, and therefore the observed features $\overline{X}_{t+1}$ of state s_{t+1} are available. This "observed" value (defined by $r_t + \gamma G(\overline{X}_{t+1}, \overline{W})$) of state s_t is then used as the (constant) target to update the weights of the neural network, when the input $\overline{X}_t$ is used to predict the value of the state s_t. Therefore, one would need to move the weights of the neural network based on the gradient of the following loss function:

$$\begin{aligned}
\overline{W} &\Leftarrow \overline{W} - \alpha \frac{\partial L_t}{\partial \overline{W}} \\
&= \overline{W} + \alpha \left\{ \underbrace{[r_t + \gamma G(\overline{X}_{t+1}, \overline{W})]}_{\text{"Observed" value}} - G(\overline{X}_t, \overline{W}) \right\} \frac{\partial G(\overline{X}_t, \overline{W})}{\partial \overline{W}} \\
&= \overline{W} + \alpha \delta_t (\nabla G(\overline{X}_t, \overline{W}))
\end{aligned}$$

This algorithm is a special case of the $TD(\lambda)$ algorithm with λ set to 0. This special case only updates the neural network by creating a bootstrapped "ground-truth" for the current time-stamp based on the evaluations of the next time-stamp. This type of ground-truth is an inherently myopic *approximation*. For example, in a chess game, the reinforcement learning system might have inadvertently made some mistake many steps ago, and it is suddenly showing high errors in the bootstrapped predictions without having shown up earlier. The errors in the bootstrapped predictions are indicative of the fact that we have received new information about each past state $\overline{X}_k$, which we can use to alter its prediction. One possibility is to bootstrap by looking ahead for multiple steps (see Exercise 7). Another solution is the use of $TD(\lambda)$, which explores the continuum between perfect Monte Carlo ground truth and single-step approximation with smooth decay. The adjustments to older predictions are increasingly discounted at the rate $\lambda < 1$. In such a case, the update can be shown to be the following [482]:

$$\overline{W} \Leftarrow \overline{W} + \alpha \delta_t \sum_{k=0}^{t} \underbrace{(\lambda\gamma)^{t-k} (\nabla G(\overline{X}_k, \overline{W}))}_{\text{Alter prediction of } \overline{X}_k} \tag{9.18}$$

At $\lambda = 1$, the approach can be shown to be equivalent to a method in which Monte-Carlo evaluations (i.e., rolling out an episodic process to the end) are used to compute the ground-truth [482]. This is because we are always using new information about errors to fully correct

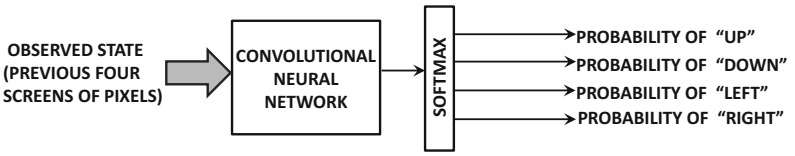


Figure 9.6: The policy network for the Atari video game setting. It is instructive to compare this configuration with the Q-network of Figure 9.3.

our past mistakes without discount at $\lambda = 1$, thereby creating an unbiased estimate. Note that λ is only used for discounting the steps, whereas γ is also used in computing the TD-error δ_t according to Equation 9.16. The parameter λ is *algorithm-specific*, whereas γ is *environment-specific*. Using $\lambda = 1$ or Monte Carlo sampling leads to lower bias and higher variance. For example, consider a chess game in which agents Alice and Bob each make three errors in a single game but Alice wins in the end. This single Monte Carlo roll out will not be able to distinguish the impact of each specific error and will assign the discounted credit for final game outcome to each board position. On the other hand, an n-step temporal difference method (i.e., n-ply board evaluation) might see a temporal difference error for each board position in which the agent made a mistake and was detected by the n-step lookahead. It is only with sufficient data (i.e., more games) that the Monte Carlo method will distinguish between different types of errors. However, choosing very small values of λ will have difficulty in learning openings (i.e., greater bias) because errors with long-term consequences will not be detected. Such problems with openings are well documented [22, 496].

Temporal difference learning was used in Samuel's celebrated checkers program [421], and also motivated the development of TD-Gammon for Backgammon by Tesauro [492]. A neural network was used for state value estimation, and its parameters were updated using temporal-difference bootstrapping over successive moves. The final inference was performed with minimax evaluation of the improved evaluation function over a shallow depth such as 2 or 3. TD-Gammon was able to defeat several expert players. It also exhibited some unusual strategies of game play that were eventually adopted by top-level players.

9.5 Policy Gradient Methods

The value-based methods like Q-learning attempt to predict the value of an action with the neural network and couple it with a generic policy (like ϵ-greedy). On the other hand, policy gradient methods estimate the *probability* of each action at each step with the goal of maximizing the overall reward. Therefore, the policy is itself parameterized, rather than using the value estimation as an intermediate step for choosing actions.

The neural network for estimating the policy is referred to as a *policy network* in which the input is the current state of the system, and the output is a set of probabilities associated with the various actions in the video game (e.g., moving up, down, left, or right). As in the case of the Q-network, the input can be an observed representation of the agent state. For example, in the Atari video game setting, the observed state can be the last four screens of pixels. An example of a policy network is shown in Figure 9.6, which is relevant for the Atari setting. It is instructive to compare this policy network with the Q-network of Figure 9.3. Given an output of probabilities for various actions, we throw a biased die with the faces associated with these probabilities, and select one of these actions. Therefore,

for each action a, observed state representation $\overline{X}_t$, and current parameter $\overline{W}$, the neural network is able to compute the function $P(\overline{X}_t, \overline{W}, a)$, which is the probability that the action a should be performed. One of the actions is sampled, and a reward is observed for that action. If the policy is poor, the action will more likely to be a mistake and the reward will be poor as well. Based on the reward obtained from executing the action, the weight vector $\overline{W}$ is updated for the next iteration. The update of the weight vector is based on the notion of policy gradient with respect to the weight vector $\overline{W}$. One challenge in estimating the policy gradient is that the reward of an action is often not observed immediately, but is tightly integrated into the future sequence of rewards. Often *Monte Carlo policy roll-outs* must be used in which the neural network is used to follow a particular policy to estimate the discounted rewards over a longer horizon.

We want to update the weight vector of the neural network along the gradient of increasing the reward. As in Q-Learning, the expected discounted rewards over a given horizon H are computed as follows:

$$J = E[r_0 + \gamma \cdot r_1 + \gamma^2 \cdot r_2 + \ldots + \gamma^H \cdot r_H] = \sum_{i=0}^{H} E[\gamma^i r_i] \tag{9.19}$$

Therefore, the goal is to update the weight vector as follows:

$$\overline{W} \Leftarrow \overline{W} + \alpha \nabla J \tag{9.20}$$

The main problem in estimating the gradient ∇J is that the neural network only outputs probabilities. The observed rewards are only Monte Carlo samples of these outputs, whereas we want to compute the gradients of *expected* rewards (cf. Equation 9.19). Common policy gradients methods include *finite difference methods*, *likelihood ratio methods*, and *natural policy gradients*. In the following, we will only discuss the first two methods.

9.5.1 Finite Difference Methods

The method of finite differences side-steps the problem of stochasticity with empirical simulations that provide estimates of the gradient. Finite difference methods use weight perturbations in order to estimate gradients of the reward. The idea is to use s different perturbations of the neural network weights, and examine the expected change ΔJ in the reward. Note that this will require us to run the perturbed policy for the horizon of H moves in order to estimate the change in reward. Such a sequence of H moves is referred to as a *roll-out*. For example, in the case of the Atari game, we will need to play it for a trajectory of H moves for each of these s different sets of perturbed weights in order to estimate the changed reward. In games where an opponent of sufficient strength is not available to train against, it is possible to play a game against a version of the opponent based on parameters learned a few iterations back.

In general, the value of H might be large enough that we might reach the end of the game, and therefore the score used will be the one at the end of the game. In some games like *Go*, the score is available only at the end of the game, with a $+1$ for a win and -1 for a loss. In such cases, it becomes more important to choose H large enough so as to play till the end of the game. As a result, we will have s different weight (change) vectors $\Delta\overline{W}_1 \ldots \Delta\overline{W}_s$, together with corresponding changes $\Delta J_1 \ldots \Delta J_s$ in the total reward. Each of these pairs roughly satisfies the following relationship:

$$(\Delta\overline{W}_r)\nabla J^T \approx \Delta J_r \quad \forall r \in \{1 \ldots s\} \tag{9.21}$$

We can create an s-dimensional column vector $\overline{y} = [\Delta J_1 \ldots \Delta J_s]^T$ of the changes in the objective function and an $N \times s$ matrix D by stacking the rows $\Delta \overline{W}_r$ on top of each other, where N is the number of parameters in the neural network. Therefore, we have the following:

$$D[\nabla J]^T \approx \overline{y} \tag{9.22}$$

Then, the policy gradient is obtained by performing a straightforward linear regression of the change in objective functions with respect to the change in weight vectors. By using the formula for linear regression (cf. Section 2.2.2.2 of Chapter 2), we obtain the following:

$$\nabla J^T = (D^T D)^{-1} D^T \overline{y} \tag{9.23}$$

This gradient is used for the update in Equation 9.20. It is required to run the policy for a sequence of H moves for each of the s samples to estimate the gradients. This process can sometimes be slow.

9.5.2 Likelihood Ratio Methods

Likelihood-ratio methods were proposed by Williams [533] in the context of the REINFORCE algorithm. Consider the case in which we are following the policy with probability vector $\overline{p}$ and we want to maximize $E[Q^p(s, a)]$, which is the long-term expected value of state s and each sampled action a from the neural network. Consider the case in which the probability of action a is $p(a)$ (which is output by the neural network). In such a case, we want to find the gradient of $E[Q^p(s, a)]$ with respect to the weight vector $\overline{W}$ of the neural network for stochastic gradient ascent. Finding the gradient of an expectation from sampled events is non-obvious. However, the log-probability trick allows us to convert it into the expectation of a gradient, which is additive over the samples of state-action pairs:

$$\nabla E[Q^p(s, a)] = E[Q^p(s, a) \nabla \log(p(a))] \tag{9.24}$$

We show the proof of the above result in terms of the partial derivative with respect to a single neural network weight w under the assumption that a is a discrete variable:

$$\frac{\partial E[Q^p(s,a)]}{\partial w} = \frac{\partial \left[\sum_a Q^p(s,a) p(a)\right]}{\partial w} = \sum_a Q^p(s,a) \frac{\partial p(a)}{\partial w} = \sum_a Q^p(s,a) \left[\frac{1}{p(a)} \frac{\partial p(a)}{\partial w}\right] p(a)$$
$$= \sum_a Q^p(s,a) \left[\frac{\partial \log(p(a))}{\partial w}\right] p(a) = E\left[Q^p(s,a) \frac{\partial \log(p(a))}{\partial w}\right]$$

The above result can also be shown for the case in which a is a continuous variable (cf. Exercise 1). Continuous actions occur frequently in robotics (e.g., distance to move arm).

It is easy to use this trick for neural network parameter estimation. Each action a sampled by the simulation is associated with the long-term reward $Q^p(s, a)$, which is obtained by Monte Carlo simulation. Based on the relationship above, the gradient of the expected advantage is obtained by multiplying the gradient of the log-probability $\log(p(a))$ of that action (computable from the neural network in Figure 9.6 using backpropagation) with the long-term reward $Q^p(s, a)$ (obtained by Monte Carlo simulation).

Consider a simple game of chess with a win/loss/draw at the end and discount factor γ In this case, the long-term reward of each move is simply obtained as a value from $\{+\gamma^{r-1}, 0, -\gamma^{r-1}\}$, when r moves remain to termination. The value of the reward depends on the final outcome of the game, and number of remaining moves (because of reward

discount). Consider a game containing at most H moves. Since multiple roll-outs are used, we get a whole bunch of training samples for the various input states and corresponding outputs in the neural network. For example, if we ran the simulation for 100 roll-outs, we would get at most $100 \times H$ different samples. Each of these would have a long-term reward drawn from $\{+\gamma^{r-1}, 0, -\gamma^{r-1}\}$. For each of these samples, the reward serves as a weight during a gradient-ascent update of the log-probability of the sampled action.

$$\overline{W} \Leftarrow \overline{W} + Q^p(s, a)\nabla \log(p(a)) \tag{9.25}$$

Here, $p(a)$ is the neural network's output probability of the sampled action. The gradients are computed using backpropagation, and these updates are similar to those in Equation 9.20. This process of sampling and updating is carried through to convergence.

Note that the gradient of the log-probability of the ground-truth class is often used to update softmax classifiers with cross-entropy loss in order to increase the probability of the correct class (which is intuitively similar to the update here). The difference here is that we are weighting the update with the Q-values because we want to push the parameters more aggressively in the direction of highly rewarding actions. One could also use mini-batch gradient ascent over the actions in the sampled roll-outs. Randomly sampling from different roll-outs can be helpful in avoiding the local minima arising from correlations because the successive samples from each roll-out are closely related to one another.

Reducing Variance with Baselines: Although we have used the long-term reward $Q^p(s, a)$ as the quantity to be optimized, it is more common to subtract a baseline value from this quantity in order to obtain its *advantage* (i.e, differential impact of the action over expectation). The baseline is ideally state-specific, but can be a constant as well. In the original work of REINFORCE, a constant baseline was used (which is typically some measure of average long-term reward over all states). Even this type of simple measure can help in speeding up learning because it reduces the probabilities of less-than-average performers and increases the probabilities of more-than-average performers (rather than increasing both at differential rates). A constant choice of baseline does not affect the bias of the procedure, but it reduces the variance. A *state-specific* option for the baseline is the value $V^p(s)$ of the state s immediately *before* sampling action a. Such a choice results in the advantage $(Q^p(s, a) - V^p(s))$ becoming identical to the temporal difference error. This choice makes intuitive sense, because the temporal difference error contains *additional* information about the differential reward of an action beyond what we would know before choosing the action. Discussions on baseline choice may be found in [374, 433].

Consider an example of an Atari game-playing agent, in which a roll-out samples the move UP and output probability of UP was 0.2. Assume that the (constant) baseline is 0.17, and the long-term reward of the action is +1, since the game results in win (and there is no reward discount). Therefore, the score of every action in that roll-out is 0.83 (after subtracting the baseline). Then, the gain associated with all actions (output nodes of the neural network) other than UP at that time-step would be 0, and the gain associated with the output node corresponding to UP would be $0.83 \times \log(0.2)$. One can then backpropagate this gain in order to update the parameters of the neural network.

Adjustment with a state-specific baseline is easy to explain intuitively. Consider the example of a chess game between agents Alice and Bob. If we use a baseline of 0, then each move will only be credited with a reward corresponding to the final result, and the difference between good moves and bad moves will not be evident. In other words, we need to simulate a lot more games to differentiate positions. On the other hand, if we use the value of the state (before performing the action) as the baseline, then the (more refined) temporal difference error is used as the advantage of the action. In such a case, moves

that have greater state-specific impact will be recognized with a higher advantage (within a single game). As a result, fewer games will be required for learning.

9.5.3 Combining Supervised Learning with Policy Gradients

Supervised learning is useful for initializing the weights of the policy network before applying reinforcement learning. For example, in a game of chess, one might have prior examples of expert moves that are already known to be good. In such a case, we simply perform gradient ascent with the same policy network, except that each expert move is assigned the fixed credit of 1 for evaluating the gradient according to Equation 9.24. This problem becomes identical to that of softmax classification, where the goal of the policy network is to predict the same move as the expert. One can sharpen the quality of the training data with some examples of bad moves with a negative credit obtained from computer evaluations. This approach would be considered supervised learning rather than reinforcement learning because we are simply using prior data, and not generating/simulating the data that we learn from (as is common in reinforcement learning). This general idea can be extended to any reinforcement learning setting, where some prior examples of actions and associated rewards are available. Supervised learning is extremely common in these settings for initialization because of the difficultly in obtaining high-quality data in the early stages of the process. Many published works also interleave supervised learning and reinforcement learning in order to achieve greater data efficiency [286].

9.5.4 Actor-Critic Methods

So far, we have discussed methods that are either dominated by *critics* or by *actors* in the following way:

1. The Q-learning and $TD(\lambda)$ methods work with the notion of a value function that is optimized. This value function is a critic, and the policy (e.g., ϵ-greedy) of the actor is directly derived from this critic. Therefore, the actor is subservient to the critic, and such methods are considered *critic-only* methods.

2. The policy-gradient methods do not use a value function at all, and they directly learn the probabilities of the policy actions. The values are often estimated using Monte Carlo sampling. Therefore, these methods are considered *actor-only* methods.

Note that the policy-gradient methods do need to evaluate the advantage of intermediate actions, and this estimation has so far been done with the use of Monte Carlo simulations. The main problem with Monte Carlo simulations is its high complexity and inability to use in an online setting.

However, it turns out that one can learn the advantage of intermediate actions using value function methods. As in the previous section, we use the notation $Q^p(s_t, a)$ to denote the value of action a, when the policy p followed by the policy network is used. Therefore, we would now have two coupled neural networks– a policy network and a Q-network. The policy network learns the probabilities of actions, and the Q-network learns the values $Q^p(s_t, a)$ of various actions in order to provide an estimation of the advantage to the policy network. Therefore, the policy network uses $Q^p(s_t, a)$ (with baseline adjustments) to weight its gradient ascent updates. The Q-network is updated using an on-policy update as in SARSA, where the policy is controlled by the policy network (rather than ϵ-greedy). The Q-network, however, does not directly decide the actions as in Q-learning, because the policy

decisions are outside its control (beyond its role as a critic). Therefore, the policy network is the actor and the value network is the critic. To distinguish between the policy network and the Q-network, we will denote the parameter vector of the policy network by $\overline{\Theta}$, and that of the Q-network by $\overline{W}$.

We assume that the state at time stamp t is denoted by s_t, and the observable features of the state input to the neural network are denoted by $\overline{X}_t$. Therefore, we will use s_t and $\overline{X}_t$ interchangeably below. Consider a situation at the tth time-stamp, where the action a_t has been observed after state s_t with reward r_t. Then, the following sequence of steps is applied for the $(t+1)$th step:

1. Sample the action a_{t+1} using the current state of the parameters in the policy network. Note that the current state is s_{t+1} because the action a_t is already observed.

2. Let $F(\overline{X}_t, \overline{W}, a_t) = \hat{Q}^p(s_t, a_t)$ represent the estimated value of $Q^p(s_t, a_t)$ by the Q-network using the observed representation $\overline{X}_t$ of the states and parameters $\overline{W}$. Estimate $Q^p(s_t, a_t)$ and $Q^p(s_{t+1}, a_{t+1})$ using the Q-network. Compute the TD-error δ_t as follows:

$$\begin{aligned}\delta_t &= r_t + \gamma \hat{Q}^p(s_{t+1}, a_{t+1}) - \hat{Q}^p(s_t, a_t) \\ &= r_t + \gamma F(\overline{X}_{t+1}, \overline{W}, a_{t+1}) - F(\overline{X}_t, \overline{W}, a_t)\end{aligned}$$

3. **[Update policy network parameters]:** Let $P(\overline{X}_t, \overline{\Theta}, a_t)$ be the probability of the action a_t predicted by policy network. Update the parameters of the policy network as follows:

$$\overline{\Theta} \leftarrow \overline{\Theta} + \alpha \hat{Q}^p(s_t, a_t) \nabla_{\Theta} \log(P(\overline{X}_t, \overline{\Theta}, a_t))$$

   Here, α is the learning rate for the policy network and the value of $\hat{Q}^p(s_t, a_t) = F(\overline{X}_t, \overline{W}, a_t)$ is obtained from the Q-network.

4. **[Update Q-Network parameters]:** Update the Q-network parameters as follows:

$$\overline{W} \Leftarrow \overline{W} + \beta \delta_t \nabla_W F(\overline{X}_t, \overline{W}, a_t)$$

   Here, β is the learning rate for the Q-network. A caveat is that the learning rate of the Q-network is generally higher than that of the policy network.

The action a_{t+1} is then executed in order to observe state s_{t+2}, and the value of t is incremented. The next iteration of the approach is executed (by repeating the above steps) at this incremented value of t. The iterations are repeated, so that the approach is executed to convergence. The value of $\hat{Q}^p(s_t, a_t)$ is the same as the value $\hat{V}^p(s_{t+1})$.

If we use $\hat{V}^p(s_t)$ as the baseline, the advantage $\hat{A}^p(s_t, a_t)$ is defined by the following:

$$\hat{A}^p(s_t, a_t) = \hat{Q}^p(s_t, a_t) - \hat{V}^p(s_t)$$

This changes the updates as follows:

$$\overline{\Theta} \leftarrow \overline{\Theta} + \alpha \hat{A}^p(s_t, a_t) \nabla_{\Theta} \log(P(\overline{X}_t, \overline{\Theta}, a_t))$$

Note the replacement of $\hat{Q}(s_t, a_t)$ in the original algorithm description with $\hat{A}(s_t, a_t)$. In order to estimate the value $\hat{V}^p(s_t)$, one possibility is to maintain another set of parameters representing the value network (which is different from the Q-network). The TD-algorithm can be used to update the parameters of the value network. However, it turns out that a

single value-network is enough. This is because we can use $r_t + \gamma \hat{V}^p(s_{t+1})$ in lieu of $\hat{Q}(s_t, a_t)$. This results in an advantage function, which is the same as the TD-error:

$$\hat{A}^p(s_t, a_t) = r_t + \gamma \hat{V}^p(s_{t+1}) - \hat{V}^p(s_t)$$

In other words, we need the single value-network (cf. Figure 9.5), which serves as the critic. The above approach can also be generalized to use the $TD(\lambda)$ algorithm at any value of λ.

9.5.5 Continuous Action Spaces

The methods discussed to this point were all associated with discrete action spaces. For example, in a video game, one might have a discrete set of choices such as whether to move the cursor up, down, left, and right. However, in a robotics application, one might have continuous action spaces, in which we wish to move the robot's arm a certain distance. One possibility is to discretize the action into a set of fine-grained intervals, and use the midpoint of the interval as the representative value. One can then treat the problem as one of discrete choice. However, this is not a particularly satisfying design choice. First, the ordering among the different choices will be lost by treating inherently ordered (numerical) values as categorical values. Second, it blows up the space of possible actions, especially if the action space is multidimensional (e.g., separate dimensions for distances moved by the robot's arm and leg). Such an approach can cause overfitting, and greatly increase the amount of data required for learning.

A commonly used approach is to allow the neural network to output the parameters of a continuous distribution (e.g., mean and standard deviation of Gaussian), and then sample from the parameters of that distribution in order to compute the value of the action in the next step. Therefore, the neural network will output the mean μ and standard deviation σ for the distance moved by the robotic arm, and the actual action a will be sampled from the Gaussian $\mathcal{N}(\mu, \sigma)$ with this parameter:

$$a \sim \mathcal{N}(\mu, \sigma) \tag{9.26}$$

In this case, the action a represents the distance moved by the robot arm. The values of μ and σ can be learned using backpropagation. In some variations, σ is fixed up front as a hyper-parameter, with only the mean μ needing to be learned. The likelihood ratio trick also applies to this case, except that we use the logarithm of the density at a, rather than the discrete probability of the action a.

9.5.6 Advantages and Disadvantages of Policy Gradients

Policy gradient methods represent the most natural choice in applications like robotics that have continuous sequences of states and actions. For cases in which there are multidimensional and continuous action spaces, the number of possible combinations of actions can be very large. Since Q-learning methods require the computation of the maximum Q-value over all such actions, this step can turn out to be computationally intractable. Furthermore, policy gradient methods tend to be stable and have good convergence properties. However, policy gradient methods are susceptible to local minima. While Q-learning methods are less stable in terms of convergence behavior than are policy-gradient methods, and can sometimes oscillate around particular solutions, they have better capacity to reach near global optima.

Policy-gradient methods do possess one additional advantage in that they can learn stochastic policies, leading to better performance in settings where deterministic policies

are known to be suboptimal (such as guessing games) due to being able to be exploited by the opponent. Q-learning provides deterministic policies, and so policy gradients are preferable in these settings because they provide a probability distribution on the possible actions from which the action is sampled.

9.6 Monte Carlo Tree Search

Monte Carlo tree search is a way of improving the strengths of learned policies and values at inference time by combining them with lookahead-based exploration. This improvement also provides a basis for lookahead-based bootstrapping like temporal difference learning. It is also leveraged as a probabilistic alternative to the deterministic minimax trees that are used by conventional game-playing software (although the applicability is not restricted to games). Each node in the tree corresponds to a state, and each branch corresponds to a possible action. The tree grows over time during the search as new states are encountered. The goal of the tree search is to select the best branch to recommend the predicted action of the agent. Each branch is associated with a value based on previous outcomes in tree search from that branch as well as an upper bound "bonus" that reduces with increased exploration. This value is used to set the priority of the branches during exploration. The learned goodness of a branch is adjusted after each exploration, so that branches leading to positive outcomes are favored in later explorations.

In the following, we will describe the Monte Carlo tree search used in *AlphaGo* as a case study for exposition. Assume that the probability $P(s, a)$ of each action (move) a at state (board position) s can be estimated using a policy network. At the same time, for each move we have a quantity $Q(s, a)$, which is the quality of the move a at state s. For example, the value of $Q(s, a)$ increases with increasing number of wins by following action a from state s in simulations. The *AlphaGo* system uses a more sophisticated algorithm that also incorporates some neural evaluations of the board position after a few moves (cf. Section 9.7.1). Then, in each iteration, the "upper bound" $u(s, a)$ of the quality of the move a at state s is given by the following:

$$u(s, a) = Q(s, a) + K \cdot \frac{P(s, a)\sqrt{\sum_b N(s, b)}}{N(s, a) + 1} \tag{9.27}$$

Here, $N(s, a)$ is the number of times that the action a was followed from state s over the course of the Monte Carlo tree search. In other words, the upper bound is obtained by starting with the quality $Q(s, a)$, and adding a "bonus" to it that depends on $P(s, a)$ and the number of times that branch is followed. The idea of scaling $P(s, a)$ by the number of visits is to discourage frequently visited branches and encourage greater exploration. The Monte Carlo approach is based on the strategy of selecting the branch with the largest upper bound, as in multi-armed bandit methods (cf. Section 9.2.3). Here, the second term on the right-hand side of Equation 9.27 plays the role of providing the confidence interval for computing the upper bound. As the branch is played more and more, the exploration "bonus" for that branch is reduced, because the width of its confidence interval drops. The hyperparameter K controls the degree of exploration.

At any given state, the action a with the largest value of $u(s, a)$ is followed. This approach is applied recursively until following the optimal action does not lead to an existing node. This new state s' is now added to the tree as a leaf node with initialized values of each $N(s', a)$ and $Q(s', a)$ set to 0. Note that the simulation up to a leaf node is fully deterministic, and no randomization is involved because $P(s, a)$ and $Q(s, a)$ are deterministically

computable. Monte Carlo simulations are used to estimate the value of the newly added leaf node s'. Specifically, Monte Carlo rollouts from the policy network (e.g., using $P(s, a)$ to sample actions) return either $+1$ or -1, depending on win or loss. In Section 9.7.1, we will discuss some alternatives for leaf-node evaluation that use a value network as well. After evaluating the leaf node, the values of $Q(s'', a'')$ and $N(s'', a'')$ on all edges (s'', a'') on the path from the current state s to the leaf s' are updated. The value of $Q(s'', a'')$ is maintained as the average value of all the evaluations at leaf nodes reached from that branch during the Monte Carlo tree search. After multiple searches have been performed from s, the most visited edge is selected as the relevant one, and is reported as the desired action.

Use in Bootstrapping

Traditionally, Monte Carlo tree search has been used during inference rather than during training. However, since Monte Carlo tree search provides an improved estimate $Q(s, a)$ of the value of a state-action pair (as a result of lookaheads), it can also be used for bootstrapping (Intuition 9.4.1). Monte Carlo tree search provides an excellent alternative to n-step temporal-difference methods. One point about on-policy n-step temporal-difference methods is that they explore a single sequence of n-moves with the ϵ-greedy policy, and therefore tend to be too weak (with increased depth but not width of exploration). One way to strengthen them is to examine all possible n-sequences and use the optimal one with an off-policy technique (i.e., generalizing Bellman's 1-step approach). In fact, this was the approach used in Samuel's checkers program [421], which used the best option in the minimax tree for bootstrapping (and later referred to as *TD-Leaf* [22]). This results in increased complexity of exploring all possible n-sequences. Monte Carlo tree search can provide a robust alternative for bootstrapping, because it can explore multiple branches from a node to generate averaged target values. For example, the lookahead-based ground truth can use the averaged performance over all the explorations starting at a given node.

AlphaGo Zero [447] bootstraps policies rather than state values, which is extremely rare. *AlphaGo Zero* uses the relative visit probabilities of the branches at each node as *posterior* probabilities of the actions at that state. These posterior probabilities are improved over the probabilistic outputs of the policy network by virtue of the fact that the visit decisions use knowledge about the future (i.e., evaluations at deeper nodes of the Monte Carlo tree). The posterior probabilities are therefore bootstrapped as ground-truth values with respect to the policy network probabilities and used to update the weight parameters (cf. Section 9.7.1.1).

9.7 Case Studies

In the following, we present case studies from real domains to showcase different reinforcement learning settings. We will present examples of reinforcement learning in *Go*, robotics, conversational systems, self-driving cars, and neural-network hyperparameter learning.

9.7.1 AlphaGo: Championship Level Play at Go

Go is a two-person board game like chess. The complexity of a two-person board game largely depends on the size of the board and the number of valid moves at each position. The simplest example of a board game is tic-tac-toe with a 3×3 board, and most humans can solve it optimally without the need for a computer. Chess is a significantly more complex game with an 8×8 board, although clever variations of the brute-force approach of *selectively*

exploring the minimax tree of moves up to a certain depth can perform significantly better than the best human today. *Go* occurs at the extreme end of complexity because of its 19×19 board.

Players play with white or black *stones*, which are kept in bowls next to the *Go* board. An example of a *Go* board is shown in Figure 9.7. The game starts with an empty board, and it fills up as players put stones on the board. Black makes the first move and starts with 181 stones in her bowl, whereas white starts with 180 stones. The total number of junctions is equal to the total number of stones in the bowls of the two players. A player places a stone of her color in each move at a particular position (from the bowl), and does not move it once it is placed. A stone of the opponent can be captured by encircling it. The objective of the game is for the player to control a larger part of the board than her opponent by encircling it with her stones.

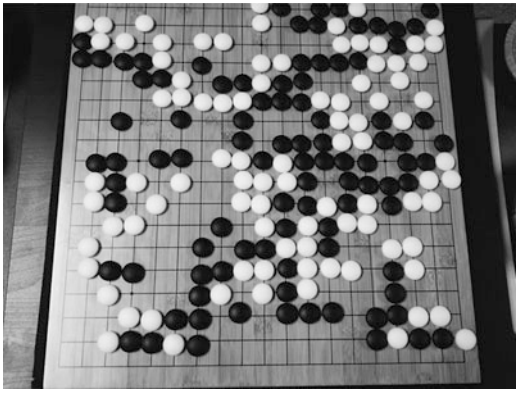


Figure 9.7: Example of a *Go* board with stones.

Whereas one can make about 35 possible moves (i.e., tree branch factor) in a particular position in chess, the average number of possible moves at a particular position in *Go* is 250, which is almost an order of magnitude larger. Furthermore, the average number of sequential moves (i.e., tree depth) of a game of *Go* is about 150, which is around twice as large as chess. All these aspects make *Go* a much harder candidate from the perspective of automated game-playing. The typical strategy of chess-playing software is to construct a minimax tree with all combinations of moves the players can make up to a certain depth, and then evaluate the final board positions with chess-specific heuristics (such as the amount of remaining material and the safety of various pieces). Suboptimal parts of the tree are pruned in a heuristic manner. This approach is simply a improved version of a brute-force strategy in which all possible positions are explored up to a given depth. The number of nodes in the minimax tree of *Go* is larger than the number of atoms in the observable universe, even at modest depths of analysis (20 moves for each player). As a result of the importance of spatial intuition in these settings, humans always perform better than brute force strategies at *Go*. The use of reinforcement learning in *Go* is much closer to what humans attempt to do. We rarely try to explore all possible combinations of moves; rather, we visually learn patterns on the board that are predictive of advantageous positions, and try to make moves in directions that are expected to improve our advantage.

The automated learning of spatial patterns that are predictive of good performance is achieved with a convolutional neural network. The state of the system is encoded in the board position at a particular point, although the board representation in *AlphaGo* includes

some additional features about the status of junctions or the number of moves since a stone was played. Multiple such spatial maps are required in order to provide full knowledge of the state. For example, one feature map would represent the status of each intersection, another would encode the number of turns since a stone was played, and so on. Integer feature maps were encoded into multiple one-hot planes. Altogether, the game board could be represented using 48 binary planes of 19×19 pixels.

AlphaGo uses its win-loss experience with repeated game playing (both using the moves of expert players and with games played against itself) to learn good policies for moves in various positions with a policy network. Furthermore, the evaluation of each position on the *Go* board is achieved with a value network. Subsequently, Monte Carlo tree search is used for final inference. Therefore, *AlphaGo* is a multi-stage model, whose components are discussed in the following sections.

Policy Networks

The policy network takes as its input the aforementioned visual representation of the board, and outputs the probability of action a in state s. This output probability is denoted by $p(s, a)$. Note that the actions in the game of *Go* correspond to the probability of placing a stone at each legal position on the board. Therefore, the output layer uses the softmax activation. Two separate policy networks are trained using different approaches. The two networks were identical in structure, containing convolutional layers with ReLU nonlinearities. Each network contained 13 layers. Most of the convolutional layers convolve with 3×3 filters, except for the first and final convolutions. The first and final filters convolve with 5×5 and 1×1 filters, respectively. The convolutional layers were zero padded to maintain their size, and 192 filters were used. The ReLU nonlinearity was used, and no maxpooling was used in order to maintain the spatial footprint.

The networks were trained in the following two ways:

- *Supervised learning:* Randomly chosen samples from expert players were used as training data. The input was the state of the network, while the output was the action performed by the expert player. The score (advantage) of such a move was always $+1$, because the goal was to train the network to *imitate* expert moves, which is also referred to as *imitation learning*. Therefore, the neural network was backpropagated with the log-likelihood of the probability of the chosen move as its gain. This network is referred to as the SL-policy network. It is noteworthy that these supervised forms of imitation learning are often quite common in reinforcement learning for avoiding cold-start problems. However, subsequent work [446] showed that dispensing with this part of the learning was a better option.

- *Reinforcement learning:* In this case, reinforcement learning was used to train the network. One issue is that *Go* needs two opponents, and therefore the network was played against itself in order to generate the moves. The current network was always played against a randomly chosen network from a few iterations back, so that the reinforcement learning could have a pool of randomized opponents. The game was played until the very end, and then an advantage of $+1$ or -1 was associated with each move depending on win or loss. This data was then used to train the policy network. This network was referred to as the RL-policy network.

Note that these networks were already quite formidable *Go* players compared to state-of-the-art software, and they were combined with Monte Carlo tree search to strengthen them.

Value Networks

This network was also a convolutional neural network, which uses the state of the network as the input and the predicted score in $[-1, +1]$ as output, where $+1$ indicates a perfect probability of 1. The output is the predicted score of the next player, whether it is white or black, and therefore the input also encodes the "color" of the pieces in terms of "player" or "opponent" rather than white or black. The architecture of the value network was very similar to the policy network, except that there were some differences in terms of the input and output. The input contained an additional feature corresponding to whether the next player to play was white or black. The score was computed using a single tanh unit at the end, and therefore the value lies in the range $[-1, +1]$. The early convolutional layers of the value network are the same as those in the policy network, although an additional convolutional layer is added in layer 12. A fully connected layer with 256 units and ReLU activation follows the final convolutional layer. In order to train the network, one possibility is to use positions from a data set [606] of *Go* games. However, the preferred choice was to generate the data set using self-play with the SL-policy and RL-policy networks all the way to the end, so that the final outcomes were generated. The state-outcome pairs were used to train the convolutional neural network. Since the positions in a single game are correlated, using them sequentially in training causes overfitting. It was important to sample positions from different games in order to prevent overfitting caused by closely related training examples. Therefore, each training example was obtained from a distinct game of self-play.

Monte Carlo Tree Search

A simplified variant of Equation 9.27 was used for exploration, which is equivalent to setting $K = 1/\sqrt{\sum_b N(s, b)}$ at each node s. Section 9.6 described a version of the Monte Carlo tree search method in which only the RL-policy network is used for evaluating leaf nodes. In the case of *AlphaGo*, two approaches are combined. First, fast Monte Carlo rollouts were used from the leaf node to create evaluation e_1. While it is possible to use the policy network for rollout, *AlphaGo* trained a simplified softmax classifier with a database of human games and some hand-crafted features for faster speed of rollouts. Second, the value network created a separate evaluation e_2 of the leaf nodes. The final evaluation e is a convex combination of the two evaluations as $e = \beta e_1 + (1 - \beta)e_2$. The value of $\beta = 0.5$ provided the best performance, although using only the value network also provided closely matching performance (and a viable alternative). The most visited branch in Monte Carlo tree search was reported as the predicted move.

9.7.1.1 Alpha Zero: Enhancements to Zero Human Knowledge

A later enhancement of the idea, referred to as *AlphaGo Zero* [446], removed the need for human expert moves (or an SL-network). Instead of separate policy and value networks, a single network outputs both the policy (i.e., action probabilities) $p(s, a)$ and the value $v(s)$ of the position. The cross-entropy loss on the output policy probabilities and the squared loss on the value output were added to create a single loss. Whereas the original version of *AlphaGo* used Monte Carlo tree search only for inference from trained networks, the zero-knowledge versions also use the visit counts in Monte Carlo tree search for training. One can view the visit count of each branch in tree search as a policy *improvement* operator over $p(s, a)$ by virtue of its lookahead-based exploration. This provides a basis for creating bootstrapped ground-truth values (Intuition 9.4.1) for neural network learning. While temporal

difference learning bootstraps state values, this approach bootstraps visit counts for learning policies. The predicted probability of Monte Carlo tree search for action a in board state s is $\pi(s,a) \propto N(s,a)^{1/\tau}$, where τ is a temperature parameter. The value of $N(s,a)$ is computed using a similar Monte Carlo search algorithm as used for *AlphaGo*, where the *prior* probabilities $p(s,a)$ output by the neural network are used for computing Equation 9.27. The value of $Q(s,a)$ in Equation 9.27 is set to the average value output $v(s')$ from the neural network of the newly created leaf nodes s' reached from state s.

AlphaGo Zero updates the neural network by bootstrapping $\pi(s,a)$ as a ground-truth, whereas ground-truth *state values* are generated with Monte Carlo simulations. At each state s, the probabilities $\pi(s,a)$, values $Q(s,a)$ and visit counts $N(s,a)$ are updated by running the Monte Carlo tree search procedure (repeatedly) starting at state s. The neural network from the previous iteration is used for selecting branches according to Equation 9.27 until a state is reached that does not exist in the tree or a terminal state is reached. For each non-existing state, a new leaf is added to the tree with its Q-values and visit values initialized to zero. The Q-values and visit counts of all edges on the path from s to the leaf node are updated based on leaf evaluation by the neural network (or by game rules for terminal states). After multiple searches starting from node s, the *posterior* probability $\pi(s,a)$ is used to sample an action for self-play and reach the next node s'. The entire procedure discussed in this paragraph is repeated at node s' to recursively obtain the next position s''. The game is recursively played to completion and the final value from $\{-1,+1\}$ is returned as the ground-truth value $z(s)$ of uniformly sampled states s on the game path. Note that $z(s)$ is defined from the perspective of the player at state s. The ground-truth values of the probabilities are already available in $\pi(s,a)$ for various values of a. Therefore, one can create a training instance for the neural network containing the input representation of state s, the bootstrapped ground-truth probabilities in $\pi(s,a)$, and the Monte Carlo ground-truth value $z(s)$. This training instance is used to update the neural network parameters. Therefore, if the probability and value outputs for the neural network are $p(s,a)$ and $v(s)$, respectively, the loss for a neural network with weight vector $\overline{W}$ is as follows:

$$L = [v(s) - z(s)]^2 - \sum_a \pi(s,a) \log[p(s,a)] + \lambda ||\overline{W}||^2 \tag{9.28}$$

Here, $\lambda > 0$ is the regularization parameter.

Further advancements were proposed in the form of *Alpha Zero* [447], which could play multiple games such as *Go*, shogi, and chess. *Alpha Zero* has handily defeated the best chess-playing software, *Stockfish*, and has also defeated the best shogi software (*Elmo*). The victory in chess was particularly unexpected by most top players, because it was always assumed that chess required too much domain knowledge for a reinforcement learning system to win over a system with hand-crafted evaluations.

Comments on Performance

AlphaGo has shown extraordinary performance against a variety of computer and human opponents. Against a variety of computer opponents, it won 494 out of 495 games [445]. Even when *AlphaGo* was handicapped by providing four free stones to the opponent, it won 77%, 86%, and 99% of the games played against (the software programs named) *Crazy Stone*, *Zen*, and *Pachi*, respectively. It also defeated notable human opponents, such as the European champion, the World champion, and the top-ranked player.

A more notable aspect of its performance was the way in which it achieved its victories. In several of its games, *AlphaGo* made many unconventional and brilliantly unorthodox moves,

which would sometimes make sense only in hindsight after the victory of the program [607, 608]. There were cases in which the moves made by *AlphaGo* were contrary to conventional wisdom, but eventually revealed innovative insights acquired by *AlphaGo* during self-play. After this match, some top *Go* players reconsidered their approach to the entire game.

The performance of *Alpha Zero* in chess was similar, where it often made material sacrifices in order to incrementally improve its position and constrict its opponent. This type of behavior is a hallmark of human play and is very different from conventional chess software (which is already much better than humans). Unlike hand-crafted evaluations, it seemed to have no pre-conceived notions on the material values of pieces, or on when a king was safe in the center of the board. Furthermore, it discovered most well-known chess openings on its own using self-play, and seemed to have its own opinions on which ones were "better." In other words, it had the ability to discover knowledge on its own. A key difference of reinforcement learning from supervised learning is that *it has the ability to innovate beyond known knowledge through learning by reward-guided trial and error*. This behavior represents some promise in other applications.

9.7.2 Self-Learning Robots

Self-learning robots represent an important frontier in artificial intelligence, in which robots can be trained to perform various tasks such as locomotion, mechanical repairs, or object retrieval by using a reward-driven approach. For example, consider the case in which one has constructed a robot that is *physically* capable of locomotion (in terms of how it is constructed and the movement choices available to it), but it has to learn the precise *choice* of movements in order to keep itself balanced and move from point A to point B. As bipedal humans, we are able to walk and keep our balance naturally without even thinking about it, but this is not a simple matter for a bipedal robot in which an incorrect choice of joint movement could easily cause it to topple over. The problem becomes even more difficult when uncertain terrain and obstacles are placed in the way of a robot.

This type of problem is naturally suited to reinforcement learning, because it is easy to judge whether a robot is walking correctly, but it is hard to specify precise rules about what the robot should do in every possible situation. In the reward-driven approach of reinforcement learning, the robot is given (virtual) rewards every time it makes progress in locomotion from point A to point B. Otherwise, the robot is free to take any actions, and it is not pre-trained with knowledge about the specific choice of actions that would help keep it balanced and walk. In other words, the robot is not seeded with any knowledge of what walking looks like (beyond the fact that it will be rewarded for using its available actions for making progress from point A to point B). This is a classical example of reinforcement learning, because the robot now needs to learn the specific sequence of actions to take in order to earn the goal-driven rewards. Although we use locomotion as a specific example in this case, this general principle applies to any type of learning in robots. For example, a second problem is that of teaching a robot manipulation tasks such as grasping an object or screwing the cap on a bottle. In the following, we will provide a brief discussion of both cases.

9.7.2.1 Deep Learning of Locomotion Skills

In this case, locomotion skills were taught to virtual robots [433], in which the robot was simulated with the *MuJoCo* physics engine [609], which stands for *Multi-Joint Dynamics with Contact*. It is a physics engine aiming to facilitate research and development in robotics,

biomechanics, graphics, and animation, where fast and accurate simulation is needed without having to construct an actual robot. Both a humanoid and a quadruped robot were used. An example of the biped model is shown in Figure 9.8. The advantage of this type of simulation is that it is inexpensive to work with a virtual simulation, and one avoids the natural safety and expense issues that arise with the physical damages in an experimentation framework that is likely to be marred by high levels of mistakes/accidents. On the flip side, a physical model provides more realistic results. In general, a simulation can often be used for smaller scale testing before building a physical model.

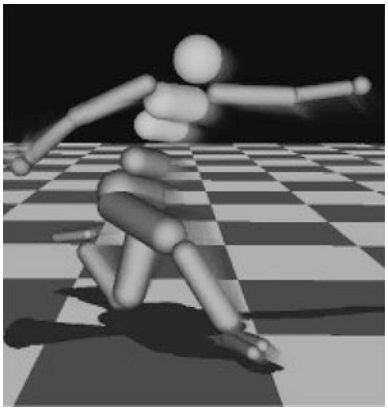


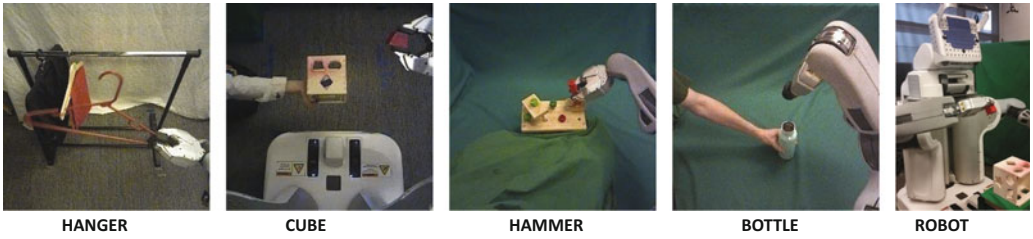
Figure 9.8: Example of the virtual humanoid robot. Original image is available at [609].

The humanoid model has 33 state dimensions and 10 actuated degrees of freedom, while the quadruped model has 29 state dimensions and 8 actuated degrees of freedom. Models were rewarded for forward progress, although episodes were terminated when the center of mass of the robot fell below a certain point. The actions of the robot were controlled by joint torques. A number of features were available to the robot, such as sensors providing the positions of obstacles, the joint positions, angles, and so on. These features were fed into the neural networks. Two neural networks were used; one was used for value estimation, and the other was used for policy estimation. Therefore, a policy gradient method was used in which the value network played the role of estimating the advantage. Such an approach is an instantiation of an actor-critic method.

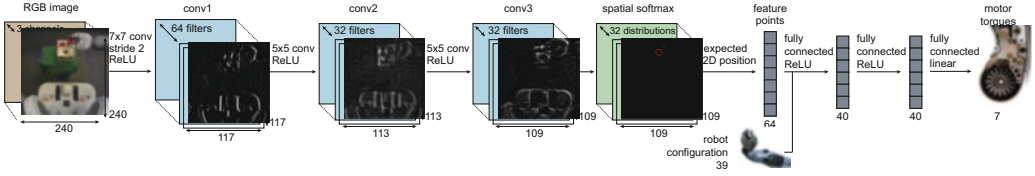
A feed-forward neural network was used with three hidden layers, with 100, 50, and 25 tanh units, respectively. The approach in [433] requires the estimation of both a policy function and a value function, and the same architecture was used in both cases for the hidden layers. However, the value estimator required only one output, whereas the policy estimator required as many outputs as the number of actions. Therefore, the main difference between the two architectures was in terms of how the output layer and the loss function was designed. The generalized advantage estimator (GAE) was used in combination with trust-based policy optimization (TRPO). The bibliographic notes contain pointers to specific details of these methods. On training the neural network for 1000 iterations with reinforcement learning, the robot learned to walk with a visually pleasing gait. A video of the final results of the robot walking is available at [610]. Similar results were also later released by Google DeepMind with more extensive abilities of avoiding obstacles or other challenges [187].

9.7.2.2 Deep Learning of Visuomotor Skills

A second and interesting case of reinforcement learning is provided in [286], in which a robot was trained for several household tasks such as placing a coat hanger on a rack, inserting a block into a shape-sorting cube, fitting the claw of a toy hammer under a nail with various grasps, and screwing a cap onto a bottle. Examples of these tasks are illustrated in Figure 9.9(a) along with an image of the robot. The actions were 7-dimensional joint motor torque commands, and each action required a sequence of commands in order to optimally perform the task. In this case, an actual physical model of a robot was used for training. A camera image was used by the robot in order to locate the objects and manipulate them. This camera image can be considered the robot's eyes, and the convolutional neural network used by the robot works on the same conceptual principle as the visual cortex (based on Hubel and Wiesel's experiments). Even though this setting seems very different from that of the Atari video games at first sight, there are significant similarities in terms of how image frames can help in mapping to policy actions. For example, the Atari setting also works with a convolutional neural network on the raw pixels. However, there were some additional inputs here, corresponding to the robot and object positions. These tasks require a high level of learning in visual perception, coordination, and contact dynamics, all of which need to learned automatically.



(a) Visuomotor tasks learned by robot



(b) Architecture of the convolutional neural network

Figure 9.9: Deep learning of visuomotor skills. These figures appear in [286]. (©2016 Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel)

A natural approach is to use a convolutional neural network for mapping image frames to actions. As in the case of Atari games, spatial features need to be learned in the layers of the convolutional neural network that are suitable for earning the relevant rewards in a task-sensitive manner. The convolutional neural network had 7 layers and 92,000 parameters. The first three layers were convolutional layers, the fourth layer was a spatial softmax, and the fifth layer was a fixed transformation from spatial feature maps to a concise set of two coordinates. The idea was to apply a softmax function to the responses across the spatial feature map. This provides a probability of each position in the feature map. The expected position using this probability distribution provides the 2-dimensional coordinate, which is

referred to as a *feature point*. Note that each spatial feature map in the convolution layer creates a feature point. The feature point can be viewed as a kind of soft argmax over the spatial probability distribution. The fifth layer was quite different from what one normally sees in a convolutional neural network, and was designed to create a precise representation of the visual scene that was suitable for feedback control. The spatial feature points are concatenated with the robot's configuration, which is an additional input occurring only after the convolution layers. This concatenated feature set is fed into two fully connected layers, each with 40 rectified units, followed by linear connections to the torques. Note that only the observations corresponding to the camera were fed to the first layer of the convolutional neural network, and the observations corresponding to the robot state were fed to the first fully connected layer. This is because the convolutional layers cannot make much use of the robot states, and it makes sense to concatenate the state-centric inputs after the visual inputs have been processed by the convolutional layers. The entire network contained about 92,000 parameters, of which 86,000 were in the convolutional layers. The architecture of the convolutional neural network is shown in Figure 9.9(b). The observations consist of the RGB camera image, joint encoder readings, velocities, and end-effector pose.

The full robot states contained between 14 and 32 dimensions, such as the joint angles, end-effector pose, object positions, and their velocities. This provided a practical notion of a state. As in all policy-based methods, the outputs correspond to the various actions (motor torques). One interesting aspect of the approach discussed in [286] is that it transforms the reinforcement learning problem into supervised learning. A *guided policy search* method was used, which is not discussed in this chapter. This approach converts portions of the reinforcement learning problem into supervised learning. Interested readers are referred to [286], where a video of the performance of the robot (trained using this system) may also be found.

9.7.3 Building Conversational Systems: Deep Learning for Chatbots

Chatbots are also referred to as *conversational systems* or *dialog systems*. The ultimate goal of a chatbot is to build an agent that can freely converse with a human about a variety of topics in a natural way. We are very far from achieving this goal. However, significant progress has been made in building chatbots for specific domains and particular applications (e.g., negotiation or shopping assistant). An example of a relatively general-purpose system is Apple's Siri, which is a digital personal assistant. One can view Siri as an open-domain system, because it is possible to have conversations with it about a wide variety of topics. It is reasonably clear to anyone using Siri that the assistant is sometimes either unable to provide satisfactory responses to difficult questions, and in some cases hilarious responses to common questions are hard-coded. This is, of course, natural because the system is relatively general-purpose, and we are nowhere close to building a human-level conversational system. In contrast, closed-domain systems have a specific task in mind, and can therefore be more easily trained in a reliable way.

In the following, we will describe a system built by *Facebook* for end-to-end learning of negotiation skills [290]. This is a closed-domain system because it is designed for the particular purpose of negotiation. As a test-bed, the following negotiation task was used. Two agents are shown a collection of items of different types (e.g., two books, one hat, three balls). The agents are instructed to divide these items among themselves by negotiating a split of the items. A key point is that the value of each of the types of items is different for the two agents, but they are not aware of the value of the items for each other. This is often the case in real-life negotiations, where users attempt to reach a mutually satisfactory outcome by negotiating for items of value to them.

The values of the items are always assumed to be non-negative and generated randomly in the test-bed under some constraints. First, the total value of all items for a user is 10. Second, each item has non-zero value to at least one user so that it makes little sense to ignore an item. Last, some items have nonzero values to both users. Because of these constraints, it is impossible for both users to achieve the maximum score of 10, which ensures a competitive negotiation process. After 10 turns, the agents are allowed the option to complete the negotiation with no agreement, which has a value of 0 points for both users. The three item types of books, hats, and balls were used, and a total of between 5 and 7 items existed in the pool. The fact that the values of the items are different for the two users (without knowledge about each other's assigned values) is significant; if both negotiators are capable, they will be able to achieve a total value of larger than 10 for the items between them. Nevertheless, the better negotiator will be able to capture the larger value by optimally negotiating for items with a high value for them.

The reward function for this reinforcement learning setting is the final value of the items attained by the user. One can use supervised learning on previous dialogs in order to maximize the likelihood of utterances. A straightforward use of recurrent networks to maximize the likelihood of utterances resulted in agents that were too eager to compromise. Therefore, the approach combined supervised learning with reinforcement learning. The incorporation of supervised learning within the reinforcement learning helps in ensuring that the models do not diverge from human language. A form of planning for dialogs called *dialog roll-out* was introduced. The approach uses an encoder-decoder recurrent architecture, in which the decoder maximizes the reward function rather than the likelihood of utterances. This encoder-decoder architecture is based on sequence-to-sequence learning, as discussed in Section 7.7.2 of Chapter 7.

To facilitate supervised learning, dialogs were collected from *Amazon Mechanical Turk*. A total of 5808 dialogs were collected in 2236 unique scenarios, where a scenario is defined by assignment of a particular set of values to the items. Of these cases, 252 scenarios corresponding to 526 dialogs were held out. Each scenario results in two training examples, which are derived from the perspective of each agent. A concrete training example could be one in which the items to be divided among the two agents correspond to 3 books, 2 hats, and 1 ball. These are part of the input to each agent. The second input could be the value of each item to the agent, which are (i) Agent A: book:1, hat:3, ball:1, and (ii) Agent B: book:2, hat:1, ball:2. Note that this means that agent A should secretly try to get as many hats as possible in the negotiation, whereas agent B should focus on books and balls. An example of a dialog in the training data is given below [290]:

Agent A: I want the books and the hats, you get the ball.
Agent B: Give me a book too and we have a deal.
Agent A: Ok, deal.
Agent B: ⟨choose⟩

The final output for agent A is 2 books and 2 hats, whereas the final output for agent B is 1 book and 1 ball. Therefore, each agent has her own set of inputs and outputs, and the dialogs for each agent are also viewed from their own perspective in terms of the portions that are reads and the portions that are writes. Therefore, each scenario generates two training examples and the same recurrent network is shared for generating the writes and the final output of each agent. The dialog x is a list of tokens $x_0 \ldots x_T$, containing the turns of each agent interleaved with symbols marking whether the turn was written by an agent or their partner. A special token at the end indicates that one agent has marked that an agreement has been reached.

The supervised learning procedure uses four different gated recurrent units (GRUs). The first gated recurrent unit GRU_g encodes the input goals, the second gated recurrent unit GRU_q generates the terms in the dialog, a forward-output gated recurrent unit $GRU_{\vec{o}}$, and a backward-output gated recurrent unit $GRU_{\overleftarrow{o}}$. The output is essentially produced by a bi-directional GRU. These GRUs are hooked up in end-to-end fashion. In the supervised learning approach, the parameters are trained using the inputs, dialogs, and outputs available from the training data. The loss for the supervised model for a weighted sum of the token-prediction loss of the dialog and the output choice prediction loss of the items.

However, for reinforcement learning, dialog roll-outs are used. Note that the group of GRUs in the supervised model is, in essence, providing probabilistic outputs. Therefore, one can adapt the same model to work for reinforcement learning by simply changing the loss function. In other words, the GRU combination can be considered a type of policy network. One can use this policy network to generate Monte Carlo roll-outs of various dialogs and their final rewards. Each of the sampled actions becomes a part of the training data, and the action is associated with the final reward of the roll-out. In other words, the approach uses *self-play* in which the agent negotiates with itself to learn better strategies. The final reward achieved by a roll-out is used to update the policy network parameters. This reward is computed based on the value of the items negotiated at the end of the dialog. This approach can be viewed as an instance of the REINFORCE algorithm [533]. One issue with self-play is that the agents tend to learn their own language, which deviates from natural human language when both sides use reinforcement learning. Therefore, one of the agents is constrained to be a supervised model.

For the final prediction, one possibility is to directly sample from the probabilities output by the GRU. However, such an approach is often not optimal when working with recurrent networks. Therefore, a two-stage approach is used. First, c candidate utterances are created by using sampling. The expected reward of each candidate utterance is computed and the one with the largest expected value is selected. In order to compute the expected reward, the output was scaled by the probability of the dialog because low-probability dialogs were unlikely to be selected by either agent.

A number of interesting observations were made in [290] about the performance of the approach. First, the supervised learning methods often tended to give up easily, whereas the reinforcement learning methods were more persistent in attempting to obtain a good deal. Second, the reinforcement learning method would often exhibit human-like negotiation tactics. In some cases, it feigned interest in an item that was not really of much value in order to obtain a better deal for another item.

9.7.4 Self-Driving Cars

As in the case of the robot locomotion task, the car is rewarded for progressing from point A to point B without causing accidents or other undesirable road incidents. The car is equipped with various types of video, audio, proximity, and motion sensors in order to record observations. The objective of the reinforcement learning system is for the car to go from point A to point B safely irrespective of road conditions.

Driving is a task for which it is hard to specify the proper rules of action in every situation; on the other hand, it is relatively easy to judge when one is driving correctly. This is precisely the setting that is well suited to reinforcement learning. Although a fully self-driving car would have a vast array of components corresponding to inputs and sensors of various types, we focus on a simplified setting in which a single camera is used [46, 47]. This system is instructive because it shows that even a single front-facing camera is sufficient to accomplish quite a lot when paired with reinforcement learning. Interestingly, this work was inspired by the 1989 work of Pomerleau [381], who built the *Autonomous Land Vehicle in a Neural Network (ALVINN)* system, and the main difference from the work done over 25 years back was one of increased data and computational power. In addition, the work uses some advances in convolutional neural networks for modeling. Therefore, this work showcases the great importance of increased data and computational power in building reinforcement learning systems.

The training data was collected by driving in a wide variety of roads and conditions. The data was collected primarily from central New Jersey, although highway data was also collected from Illinois, Michigan, Pennsylvania, and New York. Although a single front-facing camera in the driver position was used as the primary data source for making decisions, the training phase used two additional cameras at other positions in the front to collect rotated and shifted images. These auxiliary cameras, which were not used for final decision making, were however useful for collecting additional data. The placement of the additional cameras ensured that their images were shifted and rotated, and therefore they could be used to train the network to recognize cases where the car position had been compromised. In short, these cameras were useful for data augmentation. The neural network was trained to minimize the error between the steering command output by the network and the command output by the human driver. Note that this approach tends to make the approach closer to supervised learning rather than reinforcement learning. These types of learning methods are also referred to as *imitation learning* [427]. Imitation learning is often used as a first step to buffer the cold-start inherent in reinforcement learning systems.

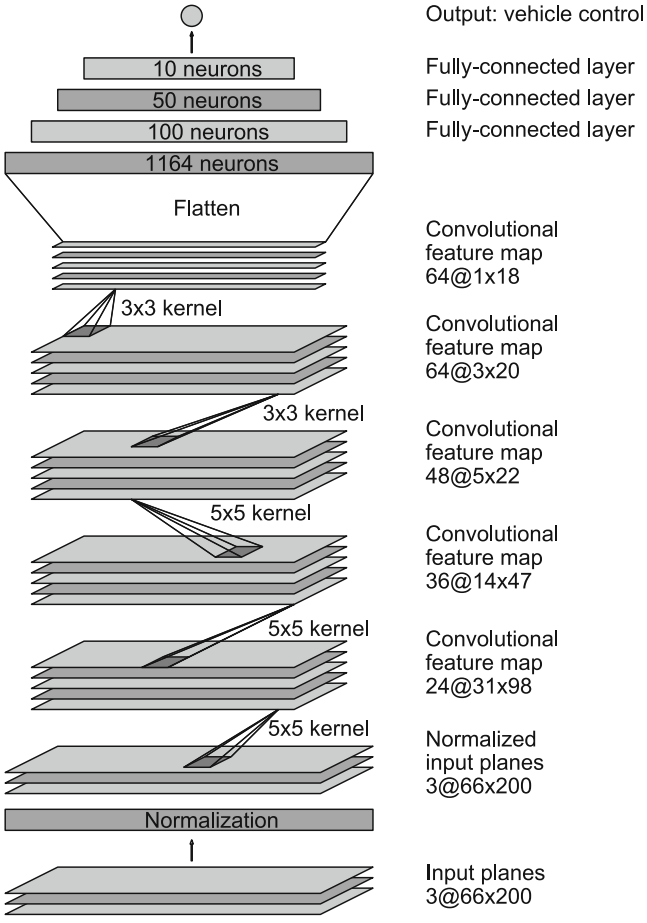


Figure 9.10: The neural network architecture of the control system in the self-driving car discussed in [46] (Courtesy NVIDIA).

Scenarios involving imitation learning are often similar to those involving reinforcement learning. It is relatively easy to use reinforcement setting in this scenario by giving a reward when the car makes progress without human intervention. On the other hand, if the car either does not make progress or requires human intervention, it is penalized. However, this does not seem to be the way in which the self-driving system of [46, 47] is trained. One issue with settings like self-driving cars is that one always has to account for safety issues during training. Although published details on most of the available self-driving cars are limited, it seems that supervised learning has been the method of choice compared to reinforcement learning in this setting. Nevertheless, the differences between using supervised learning and reinforcement learning are not significant in terms of the broader architecture of the neural network that would be useful. A general discussion of reinforcement learning in the context of self-driving cars may be found in [612].

The convolutional neural network architecture is shown in Figure 9.10. The network consists of 9 layers, including a normalization layer, 5 convolutional layers, and 3 fully connected layers. The first convolutional layer used a 5×5 filter with a stride of 2. The next two convolutional layers each used non-strided convolution with a 3×3 filter. These convo-

lutional layers were followed with three fully connected layers. The final output value was a control value, corresponding to the inverse turning radius. The network had 27 million connections and 250,000 parameters. Specific details of how the deep neural network performs the steering are provided in [47].

The resulting car was tested both in simulation and in actual road conditions. A human driver was always present in the road tests to perform interventions when necessary. On this basis, a measure was computed on the percentage of time that human intervention was required. It was found that the vehicle was autonomous 98% of the time. A video demonstration of this type of autonomous driving is available in [611]. Some interesting observations were obtained by visualizing the activation maps of the trained convolutional neural network (based on the methodology discussed in Chapter 8). In particular, it was observed that the features were heavily biased towards learning aspects of the image that were important to driving. In the case of unpaved roads, the feature activation maps were able to detect the outlines of the roads. On the other hand, if the car was located in a forest, the feature activation maps were full of noise. Note that this does not happen in a convolutional neural network that is trained on *ImageNet* because the feature activation maps would typically contain useful characteristics of trees, leaves, and so on. This difference in the two cases is because the convolutional network of the self-driving setting is trained in a goal-driven matter, and it learns to detect features that are relevant to driving. The specific characteristics of the trees in a forest are not relevant to driving.

9.7.5 Inferring Neural Architectures with Reinforcement Learning

An interesting application of reinforcement learning is to learn the neural network architecture for performing a specific task. For discussion purposes, let us consider a setting in which we wish to determine the structure of a convolutional neural architecture for classify a data set like CIFAR-10 [583]. Clearly, the structure of the neural network depends on a number of hyper-parameters, such as the number of filters, filter height, filter width, stride height, and stride width. These parameters depend on one another, and the parameters of later layers depend on those from earlier layers.

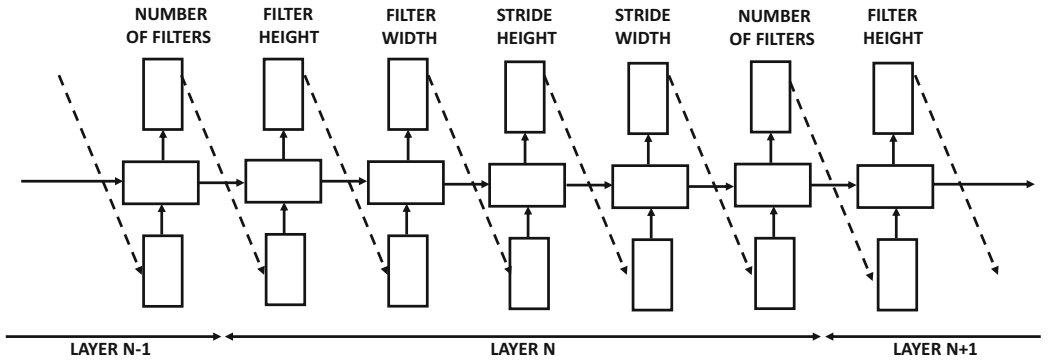


Figure 9.11: The controller network for learning the convolutional architecture of the child network [569]. The controller network is trained with the REINFORCE algorithm.

The reinforcement learning method uses a recurrent network as the *controller* to decide the parameters of the convolutional network, which is also referred to as the *child network* [569]. The overall architecture of the recurrent network is illustrated in Figure 9.11.

The choice of a recurrent network is motivated by the sequential dependence between different architectural parameters. The softmax classifier is used to predict each output as a token rather than a numerical value. This token is then used as an input into the next layer, which is shown by the dashed lines in Figure 9.11. The generation of the parameter as a token results in a discrete action space, which is generally more common in reinforcement learning as compared to a continuous action space.

The performance of the child network on a validation set drawn from CIFAR-10 is used to generate the reward signal. Note that the child network needs to be trained on the CIFAR-10 data set in order to test its accuracy. Therefore, this process requires a full training procedure of the child network, which is quite expensive. This reward signal is used in conjunction with the REINFORCE algorithm in order to train the parameters of the controller network. Therefore, the controller network is really the policy network in this case, which generates a sequence of inter-dependent parameters.

A key point is about the number of layers of the child network (which also decides the number of layers in the recurrent network). This value is not held constant but it follows a certain schedule as training progresses. In the early iterations, the number of layers is fewer, and therefore the learned architecture of the convolutional network is shallow. As training progresses, the number of layers slowly increases over time. The policy gradient method is not very different from what is discussed earlier in this chapter, except that a recurrent network is trained with the reward signal rather than a feed-forward network. Various types of optimizations are also discussed in [569], such as efficient implementations with parallelism and the learning of advanced architectural designs like skip connections.

9.8 Practical Challenges Associated with Safety

Simplifying the design of highly complex learning algorithms with reinforcement learning can sometimes have unexpected effects. By virtue of the fact that reinforcement learning systems have larger levels of freedom than other learning systems, it naturally leads to some safety related concerns. While biological greed is a powerful factor in human intelligence, it is also a source of many undesirable aspects of human behavior. The simplicity that is the greatest strength of reward-driven learning is also its greatest pitfall in biological systems. Simulating such systems therefore results in similar pitfalls from the perspective of artificial intelligence. For example, poorly designed rewards can lead to unforeseen consequences, because of the exploratory way in which the system learns its actions. Reinforcement learning systems can frequently learn unknown "cheats" and "hacks" in imperfectly designed video games, which tells us a cautionary tale of what might happen in a less-than-perfect real world. Robots learn that simply pretending to screw caps on bottles can earn faster rewards, as long as the human or automated evaluator is fooled by the action. In other words, the design of the reward function is sometimes not a simple matter.

Furthermore, a system might try to earn virtual rewards in an "unethical" way. For example, a cleaning robot might try to earn rewards by first creating messes and then cleaning them [10]. One can imagine even darker scenarios for robot nurses. Interestingly, these types of behaviors are sometimes also exhibited by humans. These undesirable similarities are a direct result of simplifying the learning process in machines by leveraging the simple greed-centric principles with which biological organisms learn. Striving for simplicity results in ceding more control to the machine, which can have unexpected effects. In some cases, there are ethical dilemmas in even designing the reward function. For example, if it becomes inevitable that an accident is going to occur, should a self-driving car save its

driver or two pedestrians? Most humans would save themselves in this setting as a matter of reflexive biological instinct; however, it is an entirely different matter to incentivize a learning system to do so. At the same time, it would be hard to convince a human operator to trust a vehicle where her safety is not the first priority for the learning system. Reinforcement learning systems are also susceptible to the ways in which their human operators interact with them and manipulate the effects of their underlying reward function; there have been occasions where a chatbot was taught to make offensive or racist remarks.

Learning systems have a harder time in generalizing their experiences to new situations. This problem is referred to as *distributional shift*. For example, a self-driving car trained in one country might perform poorly in another. Similarly, the exploratory actions in reinforcement learning can sometimes be dangerous. Imagine a robot trying to solder wires in an electronic device, where the wires are surrounded with fragile electronic components. Trying exploratory actions in this setting is fraught with perils. These issues tell us that we cannot build AI systems with no regard to safety. Indeed, some organizations like *OpenAI* [613] have taken the lead in these matters of ensuring safety. Some of these issues are also discussed in [10] with broader frameworks of possible solutions. In many cases, it seems that the human would have to be involved in the loop to some extent in order to ensure safety [424].

9.9 Summary

This chapter studies the problem of reinforcement learning in which agents interact with the environment in a reward-driven manner in order to learn the optimal actions. There are several classes of reinforcement learning methods, of which the Q-learning methods and the policy-driven methods are the most common. Policy-driven methods have become increasingly popular in recent years. Many of these methods are end-to-end systems that integrate deep neural networks to take in sensory inputs and learn policies that optimize rewards. Reinforcement learning algorithms are used in many settings like playing video or other types of games, robotics, and self-driving cars. The ability of these algorithms to learn via experimentation often leads to innovative solutions that are not possible with other forms of learning. Reinforcement learning algorithms also pose unique challenges associated with safety because of the oversimplification of the learning process with reward functions.

9.10 Bibliographic Notes

An excellent overview on reinforcement learning may be found in the book by Sutton and Barto [483]. A number of surveys on reinforcement learning are available at [293]. David Silver's lectures on reinforcement learning are freely available on *YouTube* [619]. The method of temporal differences was proposed by Samuel in the context of a checkers program [421] and formalized by Sutton [482]. Q-learning was proposed by Watkins in [519], and a convergence proof is provided in [520]. The SARSA algorithm was introduced in [412]. Early methods for using neural networks in reinforcement learning were proposed in [296, 349, 492, 493, 494]. The work in [492] developed TD-Gammon, which was a backgammon playing program.

A system that used a convolutional neural network to create a deep Q-learning algorithm with raw pixels was pioneered in [335, 336]. It has been suggested in [335] that the approach presented in the paper can be improved with other well-known ideas such as prioritized sweeping [343]. Asynchronous methods that use multiple agents in order to perform the learning are discussed in [337]. The use of multiple asynchronous threads avoids the problem

of correlation within a thread, which improves convergence to higher-quality solutions. This type of asynchronous approach is often used in lieu of the experience replay technique. Furthermore, an n-step technique, which uses a lookahead of n steps (instead of 1 step) to predict the Q-values, was proposed in the same work.

One drawback of Q-learning is that it is known to overestimate the values of actions under certain circumstances. An improvement over Q-learning, referred to as *double Q-learning*, was proposed in [174]. In the original form of Q-learning, the same values are used to both select and evaluate an action. In the case of double Q-learning, these values are decoupled, and therefore one is now learning two separate values for selection and evaluation. This change tends to make the approach less sensitive to the overestimation problem. The use of prioritized experience replay to improve the performance of reinforcement learning algorithms under sparse data is discussed in [428]. Such an approach significantly improves the performance of the system on Atari games.

In recent years, policy gradients have become more popular than Q-learning methods. An interesting and simplified description of this approach for the Atari game of *Pong* is provided in [605]. Early methods for using finite difference methods for policy gradients are discussed in [142, 355]. Likelihood methods for policy gradients were pioneered by the REINFORCE algorithm [533]. A number of analytical results on this class of algorithms are provided in [484]. Policy gradients have been used in for learning in the game of *Go* [445], although the overall approach combines a number of different elements. Natural policy gradients were proposed in [230]. One such method [432] has been shown to perform well at learning locomotion in robots. The use of *generalized advantage estimation (GAE)* with continuous rewards is discussed in [433]. The approach in [432, 433] uses natural policy gradients for optimization, and the approach is referred to as *trust region policy optimization* (TRPO). The basic idea is that bad steps in learning are penalized more severely in reinforcement learning (than supervised learning) because the quality of the collected data worsens. Therefore, the TRPO method prefers second-order methods with conjugate gradients (see Chapter 3), in which the updates tend to stay within good regions of trust. Surveys are also available on specific types of reinforcement learning methods like actor-critic methods [162].

Monte Carlo tree search was proposed in [246]. Subsequently, it was used in the game of *Go* [135, 346, 445, 446]. A survey on these methods may be found in [52]. Later versions of *AlphaGo* dispensed with the supervised portions of learning, adapted to chess and shogi, and performed better with zero initial knowledge [446, 447]. The *AlphaGo* approach combines several ideas, including the use of policy networks, Monte Carlo tree search, and convolutional neural networks. The use of convolutional neural networks for playing the game of *Go* has been explored in [73, 307, 481]. Many of these methods use supervised learning in order to mimic human experts at *Go*. Some TD-learning methods for chess, such as *NeuroChess* [496], *KnightCap* [22], and *Giraffe* [259] have been explored, but were not as successful as conventional engines. The pairing of convolutional neural networks and reinforcement learning for spatial games seems to be a new (and successful) recipe that distinguishes *Alpha Zero* from these methods. Several methods for training self-learning robots are presented in [286, 432, 433]. An overview of deep reinforcement learning methods for dialog generation is provided in [291]. Conversation models that use only supervised learning with recurrent networks are discussed in [440, 508]. The negotiation chatbot discussed in this chapter is described in [290]. The description of self-driving cars is based on [46, 47]. An MIT course on self-driving cars is available at [612]. Reinforcement learning has also been used to generate structured queries from natural language [563], or for learning neural architectures in various tasks [19, 569].

Reinforcement learning can also improve deep learning models. This is achieved with the notion of *attention* [338, 540], in which reinforcement learning is used to focus on selective parts of the data. The idea is that large parts of the data are often irrelevant for learning, and learning how to focus on selective portions of the data can significantly improve results. The selection of relevant portions of the data is achieved with reinforcement learning. Attention mechanisms are discussed in Section 10.2 of Chapter 10. In this sense, reinforcement learning is one of the topics in machine learning that is more tightly integrated with deep learning than seems at first sight.

9.10.1 Software Resources and Testbeds

Although significant progress has been made in designing reinforcement learning algorithms in recent years, commercial software using these methods is still relatively limited. Nevertheless, numerous software testbeds are available that can be used in order to test various algorithms. Perhaps the best source for high-quality reinforcement learning baselines is available from *OpenAI* [623]. *TensorFlow* [624] and *Keras* [625] implementations of reinforcement learning algorithms are also available.

Most frameworks for testing and development of reinforcement learning algorithms are specialized to specific types of reinforcement learning scenarios. Some frameworks are lightweight, and can be used for quick testing. For example, the ELF framework [498], created by *Facebook*, is designed for real-time strategy games, and is an open-source and light-weight reinforcement learning framework. The *OpenAI Gym* [620] provides environments for development of reinforcement learning algorithms for Atari games and simulated robots. The *OpenAI Universe* [621] can be used to turn reinforcement learning programs into Gym environments. For example, self-driving car simulations have been added to this environment. An Arcade learning environment for developing agents in the context of Atari games is described in [25]. The *MuJoCo* simulator [609], which stands for Multi-Joint dynamics with Contact, is a physics engine, and is designed for robotics simulations. An application with the use of *MuJoCo* is described in this chapter. *ParlAI* [622] is an open-source framework for dialog research by *Facebook*, and is implemented in Python. Baidu has created an open-source platform of its self-driving car project, referred to as *Apollo* [626].

9.11 Exercises

1. The chapter gives a proof of the likelihood ratio trick (cf. Equation 9.24) for the case in which the action a is discrete. Generalize this result to continuous-valued actions.

2. Throughout this chapter, a neural network, referred to as the policy network, has been used in order to implement the policy gradient. Discuss the importance of the choice of network architecture in different settings.

3. You have two slot machines, each of which has an array of 100 lights. The probability distribution of the reward from playing each machine is an unknown (and possibly machine-specific) function of the pattern of lights that are currently lit up. Playing a slot machine changes its light pattern in some well-defined but unknown way. Discuss why this problem is more difficult than the multi-armed bandit problem. Design a deep learning solution to optimally choose machines in each trial that will maximize the average reward per trial at steady-state.

4. Consider the well-known game of rock-paper-scissors. Human players often try to use the history of previous moves to guess the next move. Would you use a Q-learning or a policy-based method to learn to play this game? Why? Now consider a situation in which a human player samples one of the three moves with a probability that is an unknown function of the history of 10 previous moves of each side. Propose a deep learning method that is designed to play with such an opponent. Would a well-designed deep learning method have an advantage over this human player? What policy should a human player use to ensure probabilistic parity with a deep learning opponent?

5. Consider the game of tic-tac-toe in which a reward drawn from $\{-1, 0, +1\}$ is given at the end of the game. Suppose you learn the values of all states (assuming optimal play from both sides). Discuss why states in non-terminal positions will have non-zero value. What does this tell you about credit-assignment of intermediate moves to the reward value received at the end?

6. Write a Q-learning implementation that learns the value of each state-action pair for a game of tic-tac-toe by repeatedly playing against human opponents. No function approximators are used and therefore the entire table of state-action pairs is learned using Equation 9.5. Assume that you can initialize each Q-value to 0 in the table.

7. The two-step TD-error is defined as follows:

$$\delta_t^{(2)} = r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) - V(s_t)$$

(a) Propose a TD-learning algorithm for the 2-step case.

(b) Propose an on-policy n-step learning algorithm like SARSA. Show that the update is truncated variant of Equation 9.16 after setting $\lambda = 1$. What happens for the case when $n = \infty$?

(c) Propose an off-policy n-step learning algorithm like Q-learning and discuss its advantages/disadvantages with respect to (b).

Chapter 10

Advanced Topics in Deep Learning

> "Instead of trying to produce a program to simulate the adult mind, why not rather try to produce one which simulates the child's? If this were then subjected to an appropriate course of education one would obtain the adult brain."—Alan Turing in *Computing Machinery and Intelligence*

10.1 Introduction

This book will cover several advanced topics in deep learning, which either do not naturally fit within the focus of the previous chapters, or because their level of complexity requires separate treatment. The topics discussed in this chapter include the following:

1. *Attention models:* Humans do not actively use all the information available to them from the environment at any given time. Rather, they focus on specific portions of the data that are relevant to the task at hand. This biological notion is referred to as *attention*. A similar principle can also be applied to artificial intelligence applications. Models with attention use reinforcement learning (or other methods) to focus on smaller portions of the data that are relevant to the task at hand. Such methods have recently been leveraged for improved performance.

2. *Models with selective access to internal memory:* These models are closely related to attention models, although the difference is that the attention is focused primarily on specific parts of the stored data. A helpful analogy is to think of how memory is accessed by humans to perform specific tasks. Humans have a huge repository of data within the memory cells of their brains. However, at any given point, only a small part of it is accessed, which is relevant to the task at hand. Similarly, modern computers have significant amounts of memory, but computer programs are designed to access it in a selective and controlled way with the use of variables, which are indirect *addressing mechanisms*. All neural networks have memory in the form of hidden states. However,

it is so tightly integrated with the computations that it is hard to separate data access from computations. By controlling reads and writes to the internal memory of the neural network more selectively and explicitly introducing the notion of addressing mechanisms, the resulting network performs computations that reflect the human style of programming more closely. Often such networks have better generalization power than more traditional neural networks when performing predictions on out-of-sample data. One can also view selective memory access as applying a form of attention *internally* to the memory of a neural network. The resulting architecture is referred to as a *memory network* or *neural Turing machine*.

3. *Generative adversarial networks:* Generative adversarial networks are designed to create generative models of data from samples. These networks can create realistic looking samples from data by using two adversarial networks. One network generates synthetic samples (generator), and the other (which is a discriminator) classifies a mixture of original instances and generated samples as either real or synthetic. An adversarial game results in an improved generator over time, until the discriminator is no longer able to distinguish between real and fake samples. Furthermore, by conditioning on a specific type of context (e.g., image caption), it is also possible to guide the creation of specific types of desired samples.

Attention mechanisms often have to make hard decisions about specific parts of the data to attend to. One can view this choice in a similar way to the choices faced by a reinforcement learning algorithm. Some of the methods used for building attention-based models are heavily based on reinforcement learning, although others are not. Therefore, it is strongly recommended to study the materials in Chapter 9 before reading this chapter.

Neural Turing machines are related to a closely related class of architectures referred to as memory networks. Recently, they have shown promise in building question-answering systems, although the results are still quite primitive. The construction of a neural Turing machine can be considered a gateway to many capabilities in artificial intelligence that have not yet been fully realized. As is common in the historical experience with neural networks, more data and computational power will play the prominent role in bringing these promises to reality.

Most of this book discusses different types of feed-forward networks, which are based on the notion of changing weights based on errors. A completely different way of learning is that of *competitive learning*, in which the neurons compete for the right to respond to a subset of the input data. The weights are modified based on the winner of this competition. This approach is a variant of Hebbian learning discussed in Chapter 6, and is useful for unsupervised learning applications like clustering, dimensionality reduction and compression. This paradigm will also be discussed in this chapter.

Chapter Organization

This chapter is organized as follows. The next section discusses attention mechanisms in deep learning. Some of these methods are closely related to deep learning models. The augmentation of neural networks with external memory is discussed in Section 10.3. Generative adversarial networks are discussed in Section 10.4. Competitive learning methods are discussed in Section 10.5. The limitations of neural networks are presented in Section 10.6. A summary is presented in Section 10.7.

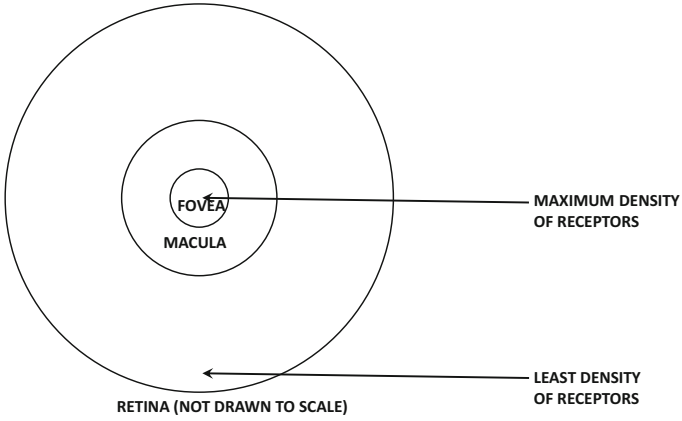


Figure 10.1: Resolutions in different regions of the eye. Most of what we focus on is captured by the macula.

10.2 Attention Mechanisms

Human beings rarely use all the available sensory inputs in order to accomplish specific tasks. Consider the problem of finding an address defined by a specific house number on a street. Therefore, an important component of the task is to identify the number written either on the door or the mailbox of a house. In this process, the retina often has an image of a broader scene, although one rarely focuses on the full image. The retina has a small portion, referred to as the *macula* with a central *fovea*, which has an extremely high resolution compared to the remainder of the eye. This region has a high concentration of color-sensitive cones, whereas most of the non-central portions of the eye have relatively low resolution with a predominance of color-insensitive rods. The different regions of the eye are shown in Figure 10.1. When reading a street number, the fovea *fixates* on the number, and its image falls on a portion of the retina that corresponds to the macula (and especially the fovea). Although one is aware of the other objects outside this central field of vision, it is virtually impossible to use images in the peripheral region to perform detail-oriented tasks. For example, it is very difficult to read letters projected on peripheral portions of the retina. The foveal region is a tiny fraction of the full retina, and it has a diameter of only 1.5 mm. The eye effectively transmits a high-resolution version of less than 0.5% of the surface area of the image that falls on the full retina. This approach is biologically advantageous, because only a carefully selected part of the image is transmitted in high resolution, and it reduces the internal processing required *for the specific task at hand*. Although the structure of the eye makes it particularly easy to understand the notion of selective attention towards visual inputs, this selectivity is not restricted only to visual aspects. Most of the other senses of the human, such as hearing or smells, are often highly focussed depending on the situation at hand. Correspondingly, we will first discuss the notion of attention in the context of computer vision, and then discuss other domains like text.

An interesting application of attention comes from the images captured by *Google Streetview*, which is a system created by Google to enable Web-based retrieval of images of various streets in many countries. This kind of retrieval requires a way to connect houses with their street numbers. Although one might record the street number during image capture, this information needs to be distilled from the image. Given a large image of the frontal

part of a house, is there a way of systematically identifying the numbers corresponding to the street address? The key here is to be able to systematically focus on small parts of the image to find what one is looking for. The main challenge here is that there is no way of identifying the relevant portion of the image with the information available up front. Therefore, an iterative approach is required in searching specific parts of the image with the use of knowledge gained from previous iterations. Here, it is useful to draw inspirations from how biological organisms work. Biological organisms draw quick visual cues from whatever they are focusing on in order to identify *where to next look* to get what they want. For example, if we first focus on the door knob by chance, then we know from experience (i.e., our trained neurons tell us) to look to its upper left or right to find the street number. This type of iterative process sounds a lot like the reinforcement learning methods discussed in the previous chapter, where one iteratively obtains cues from previous steps in order to learn what to do to earn *rewards* (i.e., accomplish a task like finding the street number). As we will see later, many applications of attention are paired with reinforcement learning.

The notion of attention is also well suited to natural language processing in which the information that we are looking for is hidden in a long segment of text. This problem arises frequently in applications like machine translation and question-answering systems where the entire sentence needs to be coded up as a fixed length vector by the recurrent neural network (cf. Section 7.7.2 of Chapter 7). As a result, the recurrent neural network is often unable to focus on the appropriate portions of the source sentence for translation to the target sentence. In such cases, it is advantageous to align the target sentence with appropriate portions of the source sentence during translation. In such cases, attention mechanisms are useful in isolating the relevant parts of the source sentence while creating a specific part of the target sentence. It is noteworthy that attention mechanisms need not always be couched in the framework of reinforcement learning. Indeed, most of the attention mechanisms in natural language models do not use reinforcement learning, but they use attention to weight specific parts of the input in a soft way.

10.2.1 Recurrent Models of Visual Attention

The work on recurrent models of visual attention [338] uses reinforcement learning to focus on important parts of an image. The idea is to use a (relatively simple) neural network in which only the resolution of specific portions of the image centered at a particular location is high. This location can change with time, as one learns more about the relevant portions of the image to explore over the course of time. Selecting a particular location in a given time-stamp is referred to as a *glimpse*. A recurrent neural network is used as the controller to identify the precise location in each time-stamp; this choice is based on the feedback from the glimpse in the previous time-stamp. The work in [338] shows that using a simple neural network (called a "glimpse network") to process the image together with the reinforcement-based training can outperform a convolutional neural network for classification.

We consider a dynamic setting in which the image may be partially observable, and the portions that are observable might vary with time-stamp t. Therefore, this setting is quite general, although we can obviously use it for more specialized settings in which the image $\overline{X}_t$ is fixed in time. The overall architecture can be described in a modular way by treating specific parts of the neural network as black-boxes. These modular portions are described below:

1. *Glimpse sensor:* Given an image with representation $\overline{X}_t$, a *glimpse sensor* creates a retina-like representation of the image. The glimpse sensor is conceptually assumed

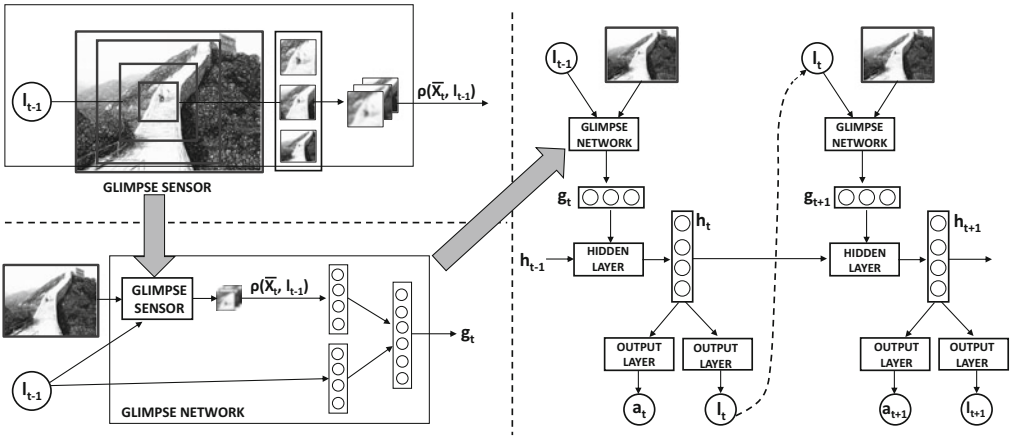


Figure 10.2: The recurrent architecture for leveraging visual attention

to not have full access to the image (because of bandwidth constraints), and is able to access only a small portion of the image in high-resolution, which is centered at l_{t-1}. This is similar to how the eye accesses an image in real life. The resolution of a particular location in the image reduces with distance from the location l_{t-1}. The reduced representation of the image is denoted by $\rho(\overline{X}_t, l_{t-1})$. The glimpse sensor, which is shown in the upper left corner of Figure 10.2, is a part of a larger glimpse network. This network is discussed below.

2. *Glimpse network:* The glimpse network contains the glimpse sensor and encodes both the glimpse location l_{t-1} and the glimpse representation $\rho(\overline{X}_t, l_{t-1})$ into hidden spaces using linear layers. Subsequently, the two are combined into a single hidden representation using another linear layer. The resulting output g_t is the input into the tth time-stamp of the hidden layer in the recurrent neural network. The glimpse network is shown in the lower-right corner of Figure 10.2.

3. *Recurrent neural network:* The recurrent neural network is the main network that is creating the action-driven outputs in each time-stamp (for earning rewards). The recurrent neural network includes the glimpse network, and therefore it includes the glimpse sensor as well (since the glimpse sensor is a part of the glimpse network). This output action of the network at time-stamp t is denoted by a_t, and rewards are associated with the action. In the simplest case, the reward might be the class label of the object or a numerical digit in the *Google Streetview* example. It also outputs a location l_t in the image for the next time-stamp, on which the glimpse network should focus. The output $\pi(a_t)$ is implemented as a probability of action a_t. This probability is implemented with the softmax function, as is common in policy networks (cf. Figure 9.6 of Chapter 9). The training of the recurrent network is done using the objective function of the REINFORCE framework to maximize the expected reward over time. The gain for each action is obtained by multiplying $\log(\pi(a_t))$ with the advantage of that action (cf. Section 9.5.2 of Chapter 9). Therefore, the overall approach is a reinforcement learning method in which the attention locations and actionable outputs are learned simultaneously. It is noteworthy that the history of actions of this recurrent network is encoded within the hidden states h_t. The overall

architecture of the neural network is illustrated on the right-hand side of Figure 10.2. Note that the glimpse network is included as a part of this overall architecture, because the recurrent network utilizes a glimpse of the image (or current state of scene) in order to perform the computations in each time-stamp.

Note that the use of a recurrent neural network architecture is useful but not necessary in these contexts.

Reinforcement Learning

This approach is couched within the framework of reinforcement learning, which allows it to be used for any type of visual reinforcement learning task (e.g., robot selecting actions to achieve a particular goal) instead of image recognition or classification. Nevertheless, supervised learning is a simple special case of this approach.

The actions a_t correspond to choosing the choosing the class label with the use of a softmax prediction. The reward r_t in the tth time-stamp might be 1 if the classification is correct after t time-stamps of that roll out, and 0, otherwise. The overall reward R_t at the tth time-stamp is given by the sum of all discounted rewards over future time stamps. However, this action can vary with the application at hand. For example, in an image captioning application, the action might correspond to choosing the next word of the caption.

The training of this setting proceeds in a similar manner to the approach discussed in Section 9.5.2 of Chapter 9. The gradient of the expected reward at time-stamp t is given by the following:

$$\nabla E[R_t] = R_t \nabla \log(\pi(a_t)) \tag{10.1}$$

Backpropagation is performed in the neural network using this gradient and policy rollouts. In practice, one will have multiple rollouts, each of which contains multiple actions. Therefore, one will have to add the gradients with respect to all these actions (or a mini-batch of these actions) in order to obtain the final direction of ascent. As is common in policy gradient methods, a baseline is subtracted from the rewards to reduce variance. Since a class label is output at each time-stamp, the accuracy will improve as more glimpses are used. The approach performs quite well using between six and eight glimpses on various types of data.

10.2.1.1 Application to Image Captioning

In this section, we will discuss the application of the visual attention approach (discussed in the previous section) to the problem of image captioning. The problem of image captioning is discussed in Section 7.7.1 of Chapter 7. In this approach, a single feature representation $\overline{v}$ of the entire image is input to the *first time-stamp* of a recurrent neural network. When a feature representation of the entire image is input, it is only provided as input at the first time-stamp when the caption begins to be generated. However, when attention is used, we want to focus on the portion of image that corresponds to the word being generated. Therefore, it makes sense to provide different attention-centric inputs at different time-stamps. For example, consider an image with the following caption:

"*Bird flying during sunset.*"

The attention should be on the location in the image corresponding to the wings of the bird while generating the word "*flying*," and the attention should be on the setting sun, while generating the word "*sunset*." In such a case, each time-stamp of the recurrent neural

network should receive a representation of the image in which the attention is on a specific location. Furthermore, as discussed in the previous section, the values of these locations are also generated by the recurrent network in the previous time-stamp.

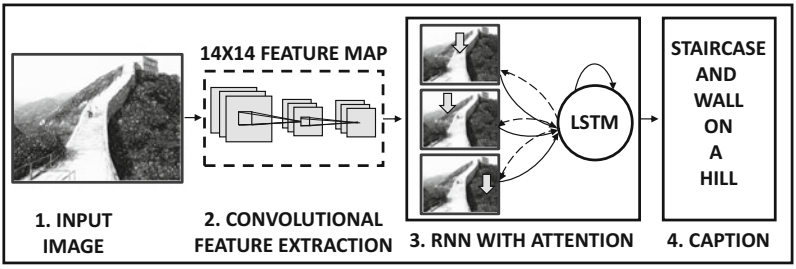


Figure 10.3: Visual attention in image captioning

Note that this approach can already be implemented with the architecture shown in Figure 10.2 by predicting one word of the caption in each time-stamp (as the action) together with a location l_t in the image, which will be the focus of attention in the next time-stamp. The work in [540] is an adaptation of this idea, but it uses several modifications to handle the higher complexity of the problem. First, the glimpse network does use a more sophisticated convolutional architecture to create a 14×14 feature map. This architecture is illustrated in Figure 10.3. Instead of using a glimpse sensor to produce the modified version of the image in each time-stamp, the work in [540] starts with L different preprocessed variants on the image. These preprocessed variants are centered at different locations in the image, and therefore the attention mechanism is restricted to selecting from one of these locations. Then, instead of producing a location l_t in the $(t-1)$th time-stamp, it produces a probability vector $\overline{\alpha}_t$ of length L indicating the relevance of each of the L locations for which representations were preprocessed in the convolutional neural network. In hard attention models, one of the L locations is sampled by using the probability vector $\overline{\alpha}_t$, and the preprocessed representation of that location is provided as input into the hidden state h_t of the recurrent network at the next time-stamp. In other words, the glimpse network in the classification application is replaced with this sampling mechanism. In soft attention models, the representation models of all L locations are averaged by using the probability vector $\overline{\alpha}_t$ as weighting. This averaged representation is provided as input to the hidden state at time-stamp t. For soft attention models, straightforward backpropagation is used for training, whereas for hard attention models, the REINFORCE algorithm (cf. Section 9.5.2 of Chapter 9) is used. The reader is referred to [540] for details, where both these types of methods are discussed.

10.2.2 Attention Mechanisms for Machine Translation

As discussed in Section 7.7.2 of Chapter 7, recurrent neural networks (and specifically their long short-term memory (LSTM) implementations) are used frequently for machine translation. In the following, we use generalized notations corresponding to any type of recurrent neural network, although the LSTM is almost always the method of choice in these settings. For simplicity, we use a single-layer network in our exposition (as well as all the illustrative figures of the neural architectures). In practice, multiple layers are used, and it is relatively easy to generalize the simplified discussion to the multi-layer case. There are several ways in which attention can be incorporated in neural machine translation. Here,

we focus on a method proposed in Luong *et al.* [302], which is an improvement over the original mechanism proposed in Bahdanau *et al.* [18].

We start with the architecture discussed in Section 7.7.2 of Chapter 7. For ease in discussion, we replicate the neural architecture of that section in Figure 10.4(a). Note that there are two recurrent neural networks, of which one is tasked with the encoding of the source sentence into a fixed length representation, and the other is tasked with decoding this representation into a target sentence. This is, therefore, a straightforward case of sequence-to-sequence learning, which is used for neural machine translation. The hidden states of the source and target networks are denoted by $h_t^{(1)}$ and $h_t^{(2)}$, respectively, where $h_t^{(1)}$ corresponds to the hidden state of the tth word in the source sentence, and $h_t^{(2)}$ corresponds to the hidden state of the tth word in the target sentence. These notations are borrowed from Section 7.7.2 of Chapter 7.

In attention-based methods, the hidden states $h_t^{(2)}$ are transformed to enhanced states $H_t^{(2)}$ with some additional processing from an *attention layer*. The goal of the attention layer is to incorporate context from the source hidden states into the target hidden states to create a new and enhanced set of target hidden states.

In order to perform attention-based processing, the goal is to find a source representation that is close to the current target hidden state $h_t^{(2)}$ being processed. This is achieved by using the similarity-weighted average of the source vectors to create a context vector $\overline{c}_t$:

$$\overline{c}_t = \frac{\sum_{j=1}^{T_s} \exp(\overline{h}_j^{(1)} \cdot \overline{h}_t^{(2)}) \overline{h}_j^{(1)}}{\sum_{j=1}^{T_s} \exp(\overline{h}_j^{(1)} \cdot \overline{h}_t^{(2)})} \tag{10.2}$$

Here, T_s is the length of the source sentence. This particular way of creating the context vector is the most simplified one among all the different versions discussed in [18, 302]; however, there are several other alternatives, some of which are parameterized. One way of viewing this weighting is with the notion of an *attention variable* $a(t, s)$, which indicates the importance of source word s to target word t:

$$a(t, s) = \frac{\exp(\overline{h}_s^{(1)} \cdot \overline{h}_t^{(2)})}{\sum_{j=1}^{T_s} \exp(\overline{h}_j^{(1)} \cdot \overline{h}_t^{(2)})} \tag{10.3}$$

We refer to the vector $[a(t, 1), a(t, 2), \ldots a(t, T_s)]$ as the attention vector $\overline{a}_t$, and it is specific to the target word t. This vector can be viewed as a set of probabilistic weights summing to 1, and its length depends on the source sentence length T_s. It is not difficult to see that Equation 10.2 is created as an attention-weighted sum of the source hidden vectors, where the attention weight of target word t towards source word s is $a(t, s)$. In other words, Equation 10.2 can be rewritten as follows:

$$\overline{c}_t = \sum_{j=1}^{T_s} a(t, j) \overline{h}_j^{(1)} \tag{10.4}$$

In essence, this approach identifies a contextual representation of the source hidden states, which is most relevant to the current target hidden state being considered. Relevance is defined by using the dot product similarity between source and target hidden states, and is captured in the attention vector. Therefore, we create a new target hidden state $H_t^{(2)}$ that combines the information in the context and the original target hidden state as follows:

$$\overline{H}_t^{(2)} = \tanh\left(W_c \left[\begin{array}{c} \overline{c}_t \\ \overline{h}_t^2 \end{array} \right]\right) \tag{10.5}$$

Once this new hidden representation $\overline{H}_t^{(2)}$ is created, it is used in lieu of the original hidden representation $\overline{h}_t^{(2)}$ for the final prediction. The overall architecture of the attention-sensitive system is given in Figure 10.4(b). Note the enhancements from Figure 10.4(a) with the addition of an attention mechanism. This model is referred to as the *global attention model* in [302]. This model is a *soft* attention model, because one is weighting all the source words with a probabilistic weight, and hard judgements are not made about which word is the most relevant one to a target word. The original work in [302] discusses another *local* model, which makes hard judgements about the relevance of target words. The reader is referred to [302] for details of this model.

Refinements

Several refinements can improve the basic attention model. First, the attention vector $\overline{a}_t$ is computed by exponentiating the raw dot products between $\overline{h}_t^{(1)}$ and $\overline{h}_s^{(2)}$, as shown in Equation 10.3. These dot products are also referred to as *scores*. In reality, there is no reason that similar positions in the source and target sentences should have similar hidden states. In fact, the source and target recurrent networks do not even need to use hidden representations of the same dimensionality (even though this is often done in practice). Nevertheless, it was shown in [302] that dot-product based similarity scoring tends to do very well in global attention models, and was the best option compared to parameterized alternatives. It is possible that the good performance of this simple approach might be a result of its regularizing effect on the model. The parameterized alternatives for computing the similarity performed better in local models (i.e., hard attention), which are not discussed in detail here.

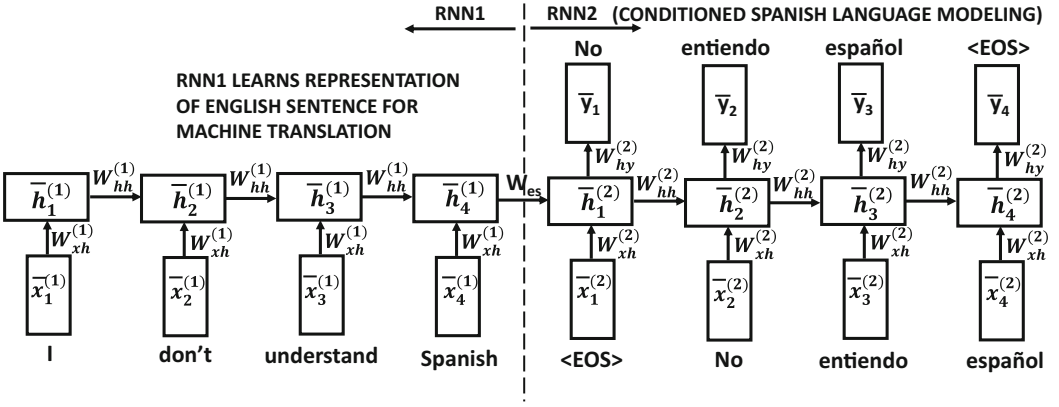
Most of these alternative models for computing the similarity use parameters to regulate the computation, which provides some additional flexibility in relating the source and target positions. The different options for computing the score are as follows:

$$\text{Score}(t,s) = \begin{cases} \overline{h}_s^{(1)} \cdot \overline{h}_t^{(2)} & \text{Dot product} \\ (\overline{h}_t^{(2)})^T W_a \overline{h}_s^{(1)} & \text{General: Parameter matrix } W_a \\ \overline{v}_a^T \tanh\left(W_a \left[\begin{array}{c} \overline{h}_s^{(1)} \\ \overline{h}_t^2 \end{array}\right]\right) & \text{Concat: Parameter matrix } W_a \text{ and vector } \overline{v}_a \end{cases} \tag{10.6}$$

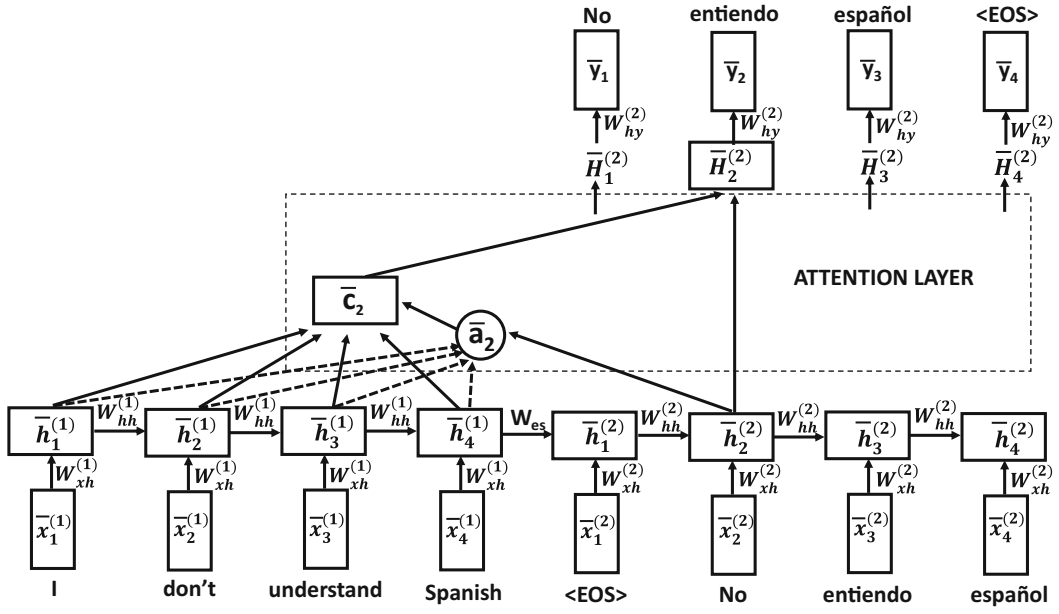
The first of these options is identical to the one discussed in the previous section according to Equation 10.3. The other two models are referred to as *general* and *concat*, respectively, as annotated above. Both these options are parameterized with the use of weight vectors, and the corresponding parameters are also annotated above. After the similarity scores have been computed, the attention values can be computed in an analogous way to the case of the dot-product similarity:

$$a(t,s) = \frac{\exp(\text{Score}(t,s))}{\sum_{j=1}^{T_s} \exp(\text{Score}(t,j))} \tag{10.7}$$

These attention values are used in the same way as in the case of dot product similarity. The parameter matrices W_a and $\overline{v}_a$ need to be learned during training. The *concat* model was proposed in earlier work [18], whereas the *general* model seemed to do well in the case of hard attention.



(a) Machine translation without attention



(b) Machine translation with attention

Figure 10.4: The neural architecture in (a) is the same as the one illustrated in Figure 7.10 of Chapter 7. An extra attention layer has been added to (b).

There are several differences of this model [302] from an earlier model presented in Bahdanau *et al.* [18]. We have chosen this model because it is simpler and it illustrates the basic concepts in a straightforward way. Furthermore, it also seems to provide better performance according to the experimental results presented in [302]. There are also some differences in the choice of neural network architecture. The work in Luong *et al.* used a

uni-directional recurrent neural network, whereas that in Bahdanau *et al.* emphasizes the use of a bidirectional recurrent neural network.

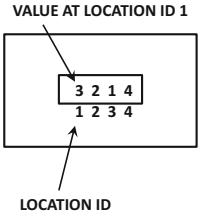
Unlike the image captioning application of the previous section, the machine translation approach is a soft attention model. The hard attention setting seems to be inherently designed for reinforcement learning, whereas the soft attention setting is differentiable, and can be used with backpropagation. The work in [302] also proposes a local attention mechanism, which focuses on a small window of context. Such an approach shares some similarities with a hard mechanism for attention (like focusing on a small region of an image as discussed in the previous section). However, it is not completely a hard approach either because one focuses on a smaller portion of the sentence using the importance weighting generated by the attention mechanism. Such an approach is able to implement the local mechanism without incurring the training challenges of reinforcement learning.

10.3 Neural Networks with External Memory

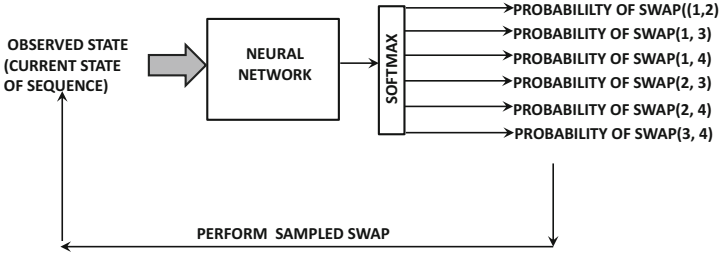
In recent years, several related architectures have been proposed that augment neural networks with *persistent memory* in which the notion of memory is clearly separated from the computations, and one can control the ways in which computations selectively access and modify particular memory locations. The LSTM can be considered to have persistent memory, although it does not clearly separate the memory from the computations. This is because the computations in a neural network are tightly integrated with the values in the hidden states, which serve the role of storing the intermediate results of the computations.

Neural Turing machines are neural networks with *external memory*. The base neural network can read or write to the external memory and therefore plays the role of a controller in guiding the computation. With the exception of LSTMs, most neural networks do not have the concept of persistent memory over long time scales. In fact, the notions of computation and memory are not clearly separated in traditional neural networks (including LSTMs). The ability to manipulate persistent memory, when combined with a clear separation of memory from computations, leads to a *programmable computer* that can simulate algorithms from examples of the input and output. This principle has led to a number of related architectures such as *neural Turing machines* [158], *differentiable neural computers* [159], and *memory networks* [528].

Why is it useful to learn from examples of the input and output? Almost all general-purpose AI is based on the assumption of being able to simulate biological behaviors in which we only have examples of the input (e.g., sensory inputs) and outputs (e.g., actions), without a crisp definition of the algorithm/function that was actually computed by that set of behaviors. In order to understand the difficulty in learning from example, we will first begin with an example of a sorting application. Although the definitions and algorithms for sorting are both well-known and crisply defined, we explore a fantasy setting in which we do not have access to these definitions and algorithms. In other words, the algorithm starts with a setting in which it has no idea of what sorting looks like. It only has examples of inputs and their sorted outputs.



(a) Output screen



(b) Policy network

Figure 10.5: The output screen and policy network for learning the fantasy game of sorting

10.3.1 A Fantasy Video Game: Sorting by Example

Although it is a simple matter to sort a set of numbers using any known sorting algorithm (e.g., quicksort), the problem becomes more difficult if we are not told that the function of the algorithm is to sort the numbers. Rather, we are only given *examples* of pairs of scrambled inputs and sorted outputs, and we have to automatically learn *sequences of actions* for any given input, so that the output reflects what we have learned from the examples. The goal is, therefore, to learn to sort by example using a specific set of predefined actions. This is a generalized view of machine learning where our inputs and outputs can be in almost any format (e.g., pixels, sounds), and goal is to learn to transform from input to output by a sequence of *actions*. These actions are the elementary steps that we are allowed to perform in our algorithm. We can already see that this action-driven approach is closely related to the reinforcement learning methodologies discussed in Chapter 9.

For simplicity, consider the case in which we want to sort only sequences of four numbers, and therefore we have four positions on our "video game screen" containing the current status of the original sequence of numbers. The screen of the fantasy video game is shown in Figure 10.5(a). There are 6 possible actions that the video game player can perform, and each action is of the form SWAP(i, j), which swaps the content of locations i and j. Since there are four possible values of each of i and j, the total number of possible actions is given by $\binom{4}{2} = 6$. The objective of the video game is to sort the numbers by using as few swaps as possible. We want to construct an algorithm that plays this video game by choosing swaps judiciously. Furthermore, the machine learning algorithm is not seeded with the knowledge that the outputs are supposed to be sorted, and it only has examples of inputs and outputs in order to build a model that (ideally) learns a policy to convert inputs into their sorted versions. Further, the video game player is not shown the input-output pairs but only incentivised with rewards when they make "good swaps" that progress towards a proper sort.

This setting is almost identical to the *Atari* video game setting discussed in Chapter 9. For example, we can use a policy network in which the current sequence of four numbers as the input to the neural network and the output is a probability of each of the 6 possible actions. This architecture is shown in Figure 10.5(b). It is instructive to compare this architecture with the policy network in Figure 9.6 of Chapter 9. The advantage for each action can be modeled in a variety of heuristic ways. For example, a naive approach would be to roll out the policy for T swapping moves and set the reward to $+1$, if we are able to obtain the correct output by then, and to -1, otherwise. Using smaller values of T would tend to favor speed over accuracy. One can also define more refined reward functions in which the reward for a sequence of moves is defined by how much closer one gets to the known output.

Consider a situation in which the probability of action $a = \text{SWAP}(i, j)$ is $\pi(a)$ (as output by the softmax function of the neural network) and the advantage is $F(a)$. Then, in policy gradient methods, we set up an objective function J_a, which is the expected advantage of action a. As discussed in Section 9.5 of Chapter 9, the gradient of this advantage with respect to the parameters of the policy network is given by the following:

$$\nabla J_a = F(a) \cdot \nabla \log(\pi(a)) \tag{10.8}$$

This gradient is added up over a minibatch of actions from the various rollouts, and used to update the weights of the neural network. Here, it is interesting to note that reinforcement learning helps us in implementing a policy for an algorithm that learns from examples.

10.3.1.1 Implementing Swaps with Memory Operations

The above video game can also be implemented by a neural network in which the allowed operations are memory read/writes and we want to sort the sequence in as few memory read/writes as possible. For example, a candidate solution to this problem would be one in which the state of the sequence is maintained in an external memory with additional space to store temporary variables for swaps. As discussed below, swaps can be implemented easily with memory read/writes. A recurrent neural network can be used to copy the states from one time-stamp to the next. The operation $\text{SWAP}(i, j)$ can be implemented by first *reading* locations i and j from memory and storing them in temporary registers. The register for i can then be written to the location of j in memory, and that for j can be written to location for i. Therefore, a sequence of memory read-writes can be used to implement swaps. In other words, we could also implement a policy for sorting by training a "controller" recurrent network that decides which locations of memory to read from and write to. However, if we create a generalized architecture with memory-based operations, the controller might learn a more efficient policy than simply implementing swaps. Here, it is important to understand that it is useful to have some form of persistent memory that stores the current state of the sorted sequence. The states of a neural network, including a (vanilla) recurrent neural network, are simply too transient to store this type of information.

Greater memory availability increases the power and sophistication of the architecture. With smaller memory availability, the policy network might learn only a simple $O(n^2)$ algorithm using swaps. On the other hand, with larger memory availability, the policy network would be able to use memory reads and writes to synthesize a wider range of operations, and it might be able to learn a much faster sorting algorithm. After all, a reward function that credits a policy for getting the correct sort in T moves would tend to favor polices with fewer moves.

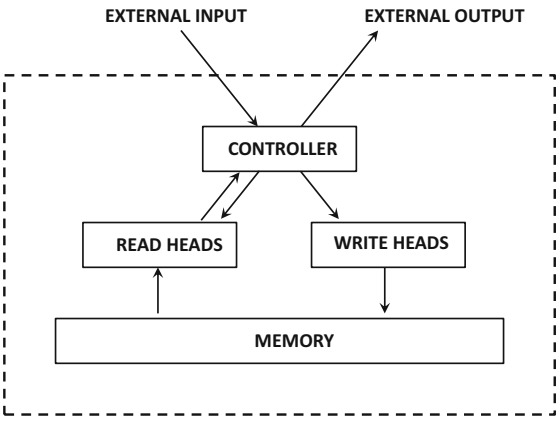


Figure 10.6: The neural Turing machine

10.3.2 Neural Turing Machines

A long-recognized weakness of neural networks is that they are unable to clearly separate the internal variables (i.e., hidden states) from the computations occurring inside the network, which causes the states to become transient (unlike biological or computer memory). A neural network in which we have an external memory and the ability to read and write to various locations in a controlled manner is very powerful, and provides a path to simulating general classes of algorithms that can be implemented on modern computers. Such an architecture is referred to as a neural Turing machine or a *differentiable neural computer*. It is referred to as a *differentiable* neural computer, because it learns to simulate algorithms (which make discrete sequences of steps) with the use of continuous optimization. Continuous optimization has the advantage of being *differentiable*, and therefore one can use backpropagation to learn optimized algorithmic steps on the input.

It is noteworthy that traditional neural networks also have memory in terms of their hidden states, and in the specific case of an LSTM, some of these hidden states are designed to be persistent. However, neural Turing machines clearly distinguish between the external memory and the hidden states within the neural network. The hidden states within the neural network can be viewed in a similar way to CPU registers that are used for transitory computation, whereas the external memory is used for persistent computation. The external memory provides the neural Turing machine to perform computations in a more similar way to how human programmers manipulate data on modern computers. This property often gives neural Turing machines much better generalizability in the learning process, as compared to somewhat similar models like LSTMs. This approach also provides a path to defining persistent data structures that are well separated from neural computations. The inability to clearly separate the program variables from computational operations has long been recognized as one of the key weaknesses of traditional neural networks.

The broad architecture of a neural Turing machine is shown in Figure 10.6. At the heart of the neural Turing machine is a controller, which is implemented using some form of a recurrent neural network (although other alternatives are possible). The recurrent architecture is useful in order to carry over the state from one time-step to the next, as the neural Turing machine implements any particular algorithm or policy. For example, in our sorting game, the current state of the sequence of numbers is carried over from one step to the next. In each time-step, it receives inputs from the environment, and writes outputs to

the environment. Furthermore, it has an external memory to which it can read and write with the use of reading and writing *heads*. The memory is structured as an $N \times m$ matrix in which there are N memory cells, each of which is of length m. At the tth time-stamp, the m-dimensional vector in the ith row of the memory is denoted by $\overline{M}_t(i)$.

The heads output a special *weight* $w_t(i) \in (0, 1)$ associated with each location i at time-stamp t that controls the degree to which it reads and writes to each output location. In other words, if the read head outputs a weight of 0.1, then it interprets anything read from the ith memory location after scaling it with 0.1 and adds up the weighted reads over different values of i. The weight of the write head is also defined in an analogous way for writing, and more details are given later. Note that the weight uses the time-stamp t as a subscript; therefore a separate set of weights is emitted at each time-stamp t. In our earlier example of swaps, this weight is like the softmax probability of a swap in the sorting video game, so that a discrete action is converted to a soft and differentiable value. However, one difference is that the neural Turing machine is not defined stochastically like the policy network of the previous section. In other words, we do not use the weight $w_t(i)$ to sample a memory cell stochastically; rather, it defines how much we read from or erase the contents of that cell. It is sometimes helpful to view each update as the expected amount by which a stochastic policy would have read or updated it. In the following, we provide a more formal description.

If the weights $w_t(i)$ have been defined, then the m-dimensional vector at location i can be read as a weighted combination of the vectors in different memory locations:

$$r_t = \sum_{i=1}^{N} w_t(i) \overline{M}_t(i) \tag{10.9}$$

The weights $w_t(i)$ are defined in such a way that they sum to 1 over all N memory vectors (like probabilities):

$$\sum_{i=1}^{N} w_t(i) = 1 \tag{10.10}$$

The writing is based on the principle of making changes by first erasing a portion of the memory and then adding to it. Therefore, in the ith time-stamp, the write head emits a weighting vector $w_t(i)$ together with length-m erase- and add-vectors $\overline{e}_t$ and $\overline{a}_t$, respectively. Then, the update to a cell is given by a combination of an erase and an addition. First the erase operation is performed:

$$\overline{M}'_t(i) \Leftarrow \underbrace{\overline{M}_{t-1}(i) \odot (1 - w_t(i)\overline{e}_t(i))}_{\text{Partial Erase}} \tag{10.11}$$

Here, the $\odot$ symbol indicates elementwise multiplication across the m dimensions of the ith row of the memory matrix. Each element in the erase vector $\overline{e}_t$ is drawn from $(0, 1)$. The m-dimensional erase vector gives fine-grained control to the choice of the elements from the m-dimensional row that can be erased. It is also possible to have multiple write heads, and the order in which multiplication is performed using the different heads does not matter because multiplication is both commutative and associative. Subsequently, additions can be performed:

$$\overline{M}_t(i) = \underbrace{\overline{M}'_t(i) + w_t(i)\overline{a}_t}_{\text{Partial Add}} \tag{10.12}$$

If multiple write heads are present, then the order of the addition operations does not matter. However, all erases must be done before all additions to ensure a consistent result irrespective of the order of additions.

Note that the changes to the cell are extremely gentle by virtue of the fact that the weights sum to 1. One can view the above update as having an intuitively similar effect as stochastically picking one of the N rows of the memory (with probability $w_t(i)$) and then sampling individual elements (with probabilities $\overline{e}_t$) to change them. However, such updates are not differentiable (unless one chooses to parameterize them using policy-gradient tricks from reinforcement learning). Here, we settle for a soft update, where all cells are changed slightly, so that the differentiability of the updates is retained. Furthermore, if there are multiple write heads, it will lead to more aggressive updates. One can also view these weights in an analogous way to how information is selectively exchanged between the hidden states and the memory states in an LSTM with the use of sigmoid functions to regulate the amount read or written into each long-term memory location (cf. Chapter 7).

Weightings as Addressing Mechanisms

The weightings can be viewed in a similar way to how addressing mechanisms work. For example, one might have chosen to sample the ith row of the memory matrix with probability $w_t(i)$ to read or write it, which is a *hard* mechanism. The soft addressing mechanism of the neural Turing machine is somewhat different in that we are reading from and writing to all cells, but changing them by tiny amounts. So far, we have not discussed *how* this addressing mechanism of setting $w_t(i)$ works. The addressing can be done either by content or by location.

In the case of addressing by content, a vector $\overline{v}_t$ of length-m, which is the *key vector*, is used to weight locations based on their dot-product similarity to $\overline{v}_t$. An exponential mechanism is used for regulating the importance of the dot-product similarity in the weighting:

$$w_t^c(i) = \frac{\exp(\text{cosine}(\overline{v}_t, \overline{M}_t(i)))}{\sum_{j=1}^{N} \exp(\text{cosine}(\overline{v}_t \cdot \overline{M}_t(j)))} \tag{10.13}$$

Note that we have added a superscript ro $w_t^c(i)$ to indicate that it is a purely content-centric weighting mechanism. Further flexibility is obtained by using a temperature parameter within the exponents to adjust the level of sharpness of the addressing. For example, if we use the temperature parameter β_t, the weights can be computed as follows:

$$w_t^c(i) = \frac{\exp(\beta_t \text{cosine}(\overline{v}_t, \overline{M}_t(i)))}{\sum_{j=1}^{N} \exp(\beta_t \text{cosine}(\overline{v}_t \cdot \overline{M}_t(j)))} \tag{10.14}$$

Increasing β_t makes the approach more like hard addressing, while reducing β_t is like soft addressing. If one wants to use only content-based addressing, then one can use $w_t(i) = w_t^c(i)$ for the addressing. Note that pure content-based addressing is almost like random access. For example, if the content of a memory location $\overline{M}_t(i)$ includes its location, then a key-based retrieval is like soft random access of memory.

A second method of addressing is by using sequential addressing with respect to the location in the previous time-stamp. This approach is referred to as location-based addressing. In location-based addressing, the value of the content weight $w_t^c(i)$ in the current iteration, and the final weights $w_{t-1}(i)$ in the previous iteration are used as starting points. First, *interpolation* mixes a partial level of random access into the location accessed in the previous iteration (via the content weight), and then a *shifting* operation adds an element of

sequential access. Finally, the softness of the addressing is *sharpened* with a temperature-like parameter. The entire process of location-based addressing uses the following steps:

$$\text{Content Weights}(\overline{v}_t, \beta_t) \Rightarrow \text{Interpolation}(g_t) \Rightarrow \text{Shift}(\overline{s}_t) \Rightarrow \text{Sharpen}(\gamma_t)$$

Each of these operations uses some outputs from the controller as input parameters, which are shown above with the corresponding operation. Since the creation of the content weights $w_t^c(i)$ has already been discussed, we explain the other three steps:

1. *Interpolation:* In this case, the vector from the previous iteration is combined with the content weights $w_t^c(i)$ created in the current iteration using a single interpolation weight $g_t \in (0, 1)$ that are output by the controller. Therefore, we have:

$$w_t^g(i) = g_t \cdot w_t^c(i) + (1 - g_t) \cdot w_{t-1}(i) \tag{10.15}$$

   Note that if g_t is 0, then the content is not used at all.

2. *Shift:* In this case, a rotational shift is performed in which a normalized vector over integer shifts is used. For example, consider a situation where $s_t[-1] = 0.2$, $s_t[0] = 0.5$ and $s_t[1] = 0.3$. This means that the weights should shift by -1 with gating weight 0.2, and by 1 with gating weight 0.3. Therefore, we define the shifted vector $w_t^s(i)$ as follows:

$$w_t^s(i) = \sum_{i=1}^{N} w_t^g(i) \cdot s_t[i - j] \tag{10.16}$$

   Here, the index of $s_t[i - j]$ is applied in combination with the modulus function to adjust it back to between -1 and $+1$ (or other integer range in which $s_t[i - j]$ is defined).

3. *Sharpening:* The process of sharpening simply takes the current set of weights, and makes them more biased towards 0 or 1, values without changing their ordering. A parameter $\gamma_t \geq 1$ is used for the sharpening, where larger values of γ_t create shaper values:

$$w_t(i) = \frac{[w_t^s(i)]^{\gamma_t}}{\sum_{j=1}^{N} [w_t^s(j)]^{\gamma_t}} \tag{10.17}$$

   The parameter γ_t plays a similar role as the temperature parameter β_t in the case of content-based weight sharpening. This type of sharpening is important because the shifting mechanism introduces a certain level of blurriness to the weights.

The purpose of these steps is as follows. First, one can use a purely content-based mechanism by using a gating weight g_t of 1. One can view a content-based mechanism as a kind of random access to memory with the key vector. Using the weight vector $w_{t-1}(i)$ in the previous iteration within the interpolation has the purpose of enabling sequential access from the reference point of the previous step. The shift vector defines how much we are willing to move from the reference point provided by the interpolation vector. Finally, sharpening helps us control the level of softness of addressing.

Architecture of Controller

An important design choice is that of the choice of the neural architecture in the controller. A natural choice is to use a recurrent neural network in which there is already a notion of temporal states. Furthermore, using an LSTM provides additional internal memory to the

external memory in the neural Turing machine. The states within the neural network are like CPU registers that are used for internal computation, but they are not persistent (unlike the external memory). It is noteworthy that once we have a concept of external memory, it is not absolutely essential to use a recurrent network. This is because the memory can capture the notion of states; reading and writing from the same set of locations over successive time-stamps achieves temporal statefulness, as in a recurrent neural network. Therefore, it is also possible to use a feed-forward neural network for the controller, which offers better transparency compared to the hidden states in the controller. The main constraint in the feed-forward architecture is that the number of read and write heads constrain the number of operations in each time-stamp.

Comparisons with Recurrent Neural Networks and LSTMs

All recurrent neural networks are known to be *Turing complete* [444], which means that they can be used to simulate any algorithm. Therefore, neural Turing machines do not *theoretically* add to the inherent capabilities of any recurrent neural network (including an LSTM). However, despite the Turing completeness of recurrent networks, there are severe limitations to their practical performance as well as generalization power on data sets containing longer sequences. For example, if we train a recurrent network on sequences of a certain size, and then apply on test data with a different size distribution, the performance will be poor.

The controlled form of the external memory access in a neural Turing machine provides it with practical advantages over a recurrent neural network in which the values in the transient hidden states are tightly integrated with computations. Although an LSTM is augmented with its own internal memory, which is somewhat resistant to updates, the processes of computation and memory access are still not clearly separated (like a modern computer). In fact, the amount of computation (i.e., number of activations) and the amount of memory (i.e., number of hidden units) are also tightly integrated in all recurrent neural networks. Clean separation between memory and computations allows control on the memory operations in a more interpretable way, which is at least somewhat similar to how a human programmer accesses and writes to internal data structures. For example, in a question-answering system, we want to be able to able to read a passage and then answer questions about it; this requires much better control in terms of being able to read the story into memory in some form.

An experimental comparison in [158] showed that the neural Turing machine works better with much longer sequences of inputs as compared to the LSTM. One of these experiments provided both the LSTM and the neural Turing machine with pairs of input/output sequences that were identical. The goal was to copy the input to the output. In this case, the neural Turing machine generally performed better as compared to the LSTM, especially when the inputs were long. Unlike the un-interpretable LSTM, the operations in the memory network were far more interpretable, and the copying algorithm implicitly learned by the neural Turing machine performed steps that were similar to how a human programmer would perform the task. As a result, the copying algorithm could generalize even to longer sequences than were seen during training time in the case of the neural Turing machine (but not so much in the case of the LSTM). In a sense, the intuitive way in which a neural Turing machine handles memory updates from one time-stamp to the next provides it a helpful regularization. For example, if the copying algorithm of the neural Turing machine mimics a human coder's style of implementing a copying algorithm, it will do a better job with longer sequences at test time.

In addition, the neural Turing machine was experimentally shown to be good at the task of *associative recall*, in which the input is a sequence of items together with a randomly chosen item from this sequence. The output is the next item in the sequence. The neural Turing machine was again able to learn this task better than an LSTM. In addition, a sorting application was also implemented in [158]. Although most of these applications are relatively simple, this work is notable for its *potential* in using more carefully tuned architectures to perform complex tasks. One such enhancement was the differentiable neural computer [159], which has been used for complex tasks of reasoning in graphs and natural languages. Such tasks are difficult to accomplish with a traditional recurrent network.

10.3.3 Differentiable Neural Computer: A Brief Overview

The differentiable neural computer is an enhancement over the neural Turing machines with the use of additional structures to manage memory allocation and keeping track of temporal sequences of writes. These enhancements address two main weaknesses of neural Turing machines:

1. Even though the neural Turing machine is able to perform both content- and location-based addressing, there is no way of avoiding the fact that it writes on overlapping blocks when it uses shift-based mechanisms to address contiguous blocks of locations. In modern computers, this issue is resolved by proper memory allocation during running time. The differentiable neural computer incorporates memory allocation mechanisms within the architecture.

2. The neural Turing machine does not keep track of the order in which memory locations are written. Keeping track of the order in which memory locations are written is useful in many cases such as keeping track of a sequence of instructions.

In the following, we will discuss only a brief overview of how these two additional mechanisms are implemented. For more detailed discussions of these mechanisms, we refer the reader to [159].

The memory allocation mechanism in a differentiable neural computer is based on the concepts that (i) locations that have just been written but not read yet are probably useful, and that (ii) the reading of a location reduces its usefulness. The memory allocation mechanism keeps track of a quantity referred to as the *usage* of a location. The usage of a location is automatically increased after each write, and it is potentially decreased after a read. Before writing to memory, the controller emits a set of free gates from each read head that determine whether the most recently read locations should be freed. These are then used to update the usage vector from the previous time-stamp. The work in [159] discusses a number of algorithms for how these usage values are used to identify locations for writing.

The second issue addressed by the differentiable neural computer is in terms of how it keeps track of the sequential ordering of the memory locations at which the writes are performed. Here, it is important to understand that the writes to the memory locations are soft, and therefore one cannot define a strict ordering. Rather, a soft ordering exists between all pairs of locations. Therefore, an $N \times N$ temporal link matrix with entries $L_t[i, j]$ is maintained. The value of $L_t[i, j]$ always lie in the range $(0, 1)$ and it indicates the degree to which row i of the $N \times m$ memory matrix was written to just after row j. In order to update the temporal link matrix, a precedence weighting is defined over the locations in the memory rows. Specifically, $p_t(i)$ defines the degree to which location i was the last one

written to at the tth time-stamp. This precedence relation is used to update the temporal link matrix in each time-stamp. Although the temporal link matrix potentially requires $O(N^2)$ space, it is very sparse and can therefore be stored in $O(N \cdot \log(N))$ space. The reader is referred to [159] for additional details of the maintenance of the temporal link matrix.

It is noteworthy that many of the ideas of neural Turing machines, memory networks, and attention mechanisms are closely related. The first two ideas were independently proposed at about the same time. The initial papers on these topics tested them on different tasks. For example, the neural Turing machine was tested on simple tasks like copying or sorting, whereas the memory network was tested on tasks like question-answering. However, this difference was also blurred at a later stage, when the differentiable neural computer was tested on the question-answering tasks. Broadly speaking, these applications are still in their infancy, and a lot needs to be done to bring them to a level where they can be commercially used.

10.4 Generative Adversarial Networks (GANs)

Before introducing generative adversarial networks, we will first discuss the notions of the *generative* and *discriminative* models, because they are both used for creating such networks. These two types of learning models are as follows:

1. *Discriminative models:* Discriminative models directly estimate the conditional probability $P(y|\overline{X})$ of the label y, given the feature values in $\overline{X}$. An example of a discriminative model is logistic regression.

2. *Generative models:* Generative models estimate the joint probability $P(\overline{X}, y)$, which is a generative probability of a data instance. Note that the joint probability can be used to estimate the conditional probability of y given $\overline{X}$ by using the Bayes rule as follows:
$$P(y|\overline{X}) = \frac{P(\overline{X}, y)}{P(\overline{X})} = \frac{P(\overline{X}, y)}{\sum_z P(\overline{X}, z)} \tag{10.18}$$
An example of a generative model is the naïve Bayes classifier.

Discriminative models can only be used in supervised settings, whereas generative models are used in both supervised and unsupervised settings. For example, in a multiclass setting, one can create a generative model of only one of the classes by defining an appropriate prior distribution on that class and then sampling from the prior distribution to generate examples of the class. Similarly, one can generate each point in the entire data set from a particular distribution by using a probabilistic model with a particular prior. Such an approach is used in the variational autoencoder (cf. Section 4.10.4 of Chapter 4) in order to sample points from a Gaussian distribution (as a prior) and then use these samples as input to the decoder in order to generate samples like the data.

Generative adversarial networks work with two neural network models simultaneously. The first is a generative model that produces synthetic examples of objects that are similar to a real repository of examples. Furthermore, the goal is to create synthetic objects that are so realistic that it is impossible for a trained observer to distinguish whether a particular object belongs to the original data set, or whether it was generated synthetically. For example, if we have a repository of car images, the generative network will use the generative model to create synthetic examples of car images. As a result, we will now end up with both

real and fake examples of car images. The second network is a discriminative network that has been trained on a data set which is labeled with the fact of whether the images are synthetic or fake. The discriminative model takes in inputs of either real examples from the base data or synthetic objects created by the generator network, and tries to discern as to whether the objects are real or fake. In a sense, one can view the generative network as a "counterfeiter" trying to produce fake notes, and the discriminative network as the "police" who is trying to catch the counterfeiter producing fake notes. Therefore, the two networks are adversaries, and training makes both adversaries better, until an equilibrium is reached between them. As we will see later, this adversarial approach to training boils down to formulating a minimax problem.

When the discriminative network is correctly able to flag a synthetic object as fake, the fact is used by the generative network to modify its weights, so that the discriminative network will have a harder time classifying samples generated from it. After modifying the weights of the generator network, new samples are generated from it, and the process is repeated. Over time, the generative network gets better and better at producing counterfeits. Eventually, it becomes impossible for the discriminator to distinguish between real and synthetically generated objects. In fact, it can be formally shown that the *Nash equilibrium* of this minimax game is a (generator) parameter setting in which the distribution of points created by the generator is the same as that of the data samples. For the approach to work well, it is important for the discriminator to be a high-capacity model, and also have access to a lot of data.

The generated objects are often useful for creating large amounts of synthetic data for machine learning algorithms, and may play a useful role in data augmentation. Furthermore, by adding context, it is possible to use this approach for generating objects with different properties. For example, the input might be a text caption, such as "*spotted cat with collar*," and the output will be a fantasy image matching the description [331, 392]. The generated objects are sometimes also used for artistic endeavors. Recently, these methods have also found application in image-to-image translation. In image-to-image translation, the missing characteristics of an image are completed in a realistic way. Before discussing the applications, we will first discuss the details of training a generative adversarial network.

10.4.1 Training a Generative Adversarial Network

The training process of a generative adversarial network proceeds by alternately updating the parameters of the generator and the discriminator. Both the generator and discriminator are neural networks. The discriminator is a neural network with d-dimensional inputs and a single output in $(0, 1)$, which indicates the probability whether or not the d-dimensional input example is real. A value of 1 indicates that the example is real, and a value of 0 indicates that the example is synthetic. Let the output of the discriminator for input $\overline{X}$ be denoted by $D(\overline{X})$.

The generator takes as input noise samples from a p-dimensional probability distribution, and uses it to generate d-dimensional examples of the data. One can view the generator in an analogous way to the decoder portion of a variational autoencoder (cf. Section 4.10.4 of Chapter 4), in which the input distribution is a p-dimensional point drawn from a Gaussian distribution (which is the *prior* distribution), and the output of the decoder is a d-dimensional data point with a similar distribution as the real examples. The training process here is, however, very different from that in a variational autoencoder. Instead of using the reconstruction error for training, the discriminator error is used to train the generator to create other samples like the input data distribution.

The goal for the discriminator is to correctly classify the real examples to a label of 1, and the synthetically generated examples to a label of 0. On the other hand, the goal for the generator is generate examples so that they fool the discriminator (i.e., encourage the discriminator to label such examples as 1). Let R_m be m randomly sampled examples from the real data set, and S_m be m synthetic samples that are generated by using the generator. Note that the synthetic samples are generated by first creating a set N_m of p-dimensional noise samples $\{\overline{Z}_m \ldots \overline{Z}_m\}$, and then applying the generator to these noise samples as the input to create the data samples $S_m = \{G(\overline{Z}_1) \ldots G(\overline{Z}_m)\}$. Therefore, the *maximization* objective function J_D for the discriminator is as follows:

$$\text{Maximize}_D \, J_D = \underbrace{\sum_{\overline{X} \in R_m} \log\left[D(\overline{X})\right]}_{m \text{ samples of real examples}} + \underbrace{\sum_{\overline{X} \in S_m} \log\left[1 - D(\overline{X})\right]}_{m \text{ samples of synthetic examples}}$$

It is easy to verify that this objective function will be maximized when real examples are correctly classified to 1 and synthetic examples are correctly classified to 0.

Next we define the objective function of the generator, whose goal is to fool the discriminator. For the generator, we do not care about the real examples, because the generator only cares about the sample it generates. The generator creates m synthetic samples, S_m, and the goal is to ensure that the discriminator recognizes these examples as genuine ones. Therefore, the generator objective function, J_G, *minimizes* the likelihood that these samples are flagged as synthetic, which results in the following optimization problem:

$$\begin{aligned}\text{Minimize}_G \, J_G &= \underbrace{\sum_{\overline{X} \in S_m} \log\left[1 - D(\overline{X})\right]}_{m \text{ samples of synthetic examples}} \\ &= \sum_{\overline{Z} \in N_m} \log\left[1 - D(G(\overline{Z}))\right]\end{aligned}$$

This objective function is minimized when the synthetic examples are incorrectly classified to 1. By minimizing the objective function, we are effectively trying to learn parameters of the generator that fool the discriminator into incorrectly classifying the synthetic examples to be true samples from the data set. An alternative objective function for the generator is to maximize $\log\left[D(\overline{X})\right]$ for each $\overline{X} \in S_m$ instead of minimizing $\log\left[1 - D(\overline{X})\right]$, and this alternative objective function sometimes works better during the early iterations of optimization.

The overall optimization problem is therefore formulated as a minimax game over J_D. Note that maximizing J_G over different choices of the parameters in the generator G is the same as maximizing J_D because $J_D - J_G$ does not include any of the parameters of the generator G. Therefore, one can write the overall optimization problem (over both generator and discriminator) as follows:

$$\text{Minimize}_G \, \text{Maximize}_D \, J_D \tag{10.19}$$

The result of such an optimization is a *saddle point* of the optimization problem. Examples of what saddle points look like with respect to the topology of the loss function are shown[1] in Figure 3.17 of Chapter 3.

[1]The examples in Chapter 3 are given in a different context. Nevertheless, if we pretend that the loss function in Figure 3.17(b) represents J_D, then the annotated saddle point in the figure is visually instructive.

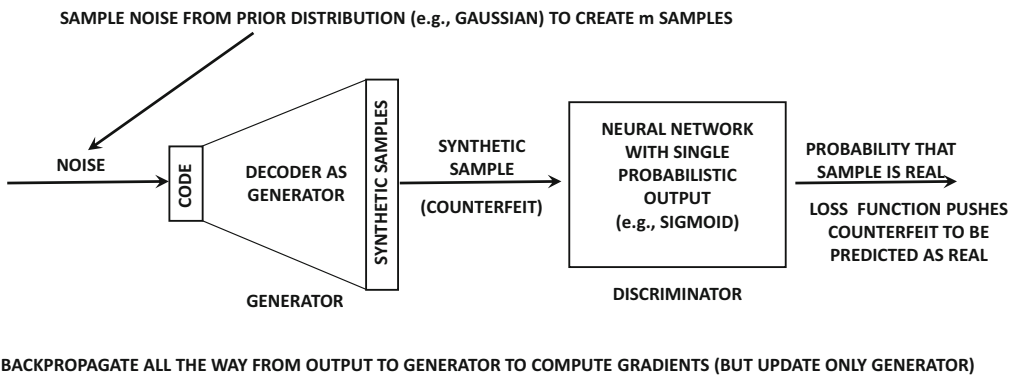


Figure 10.7: Hooked up configuration of generator and discriminator for performing gradient-descent updates on generator

Stochastic gradient ascent is used for learning the parameters of the discriminator and stochastic gradient descent is used for learning the parameters of the generator. The gradient update steps are alternated between the generator and the discriminator. In practice, however, k steps of the discriminator are used for each step of the generator. Therefore, one can describe the gradient update steps as follows:

1. **(Repeat k times):** A mini-batch of size $2 \cdot m$ is constructed with an equal number of real and synthetic examples. The synthetic examples are created by inputting noise samples to the generator from the prior distribution, whereas the real samples are selected from the base data set. Stochastic gradient ascent is performed on the parameters of the discriminator so as the maximize the likelihood that the discriminator correctly classifies both the real and synthetic examples. For each update step, this is achieved by performing backpropagation on the discriminator network with respect to the mini-batch of $2 \cdot m$ real/synthetic examples.

2. **(Perform once):** Hook up the discriminator at the end of the generator as shown in Figure 10.7. Provide the generator with m noise inputs so as to create m synthetic examples (which is the current mini-batch). Perform stochastic gradient descent on the parameters of the generator so as to minimize the likelihood that the discriminator correctly classifies the synthetic examples. The minimization of $\log\left[1 - D(\overline{X})\right]$ in the loss function explicitly encourages these counterfeits to be predicted as real.

   Even though the discriminator is hooked up to the generator, the gradient updates (during backpropagation) are performed with respect to the parameters of only the generator network. Backpropagation will automatically compute the gradients with respect to both the generator and discriminator networks for this hooked up configuration, but only the parameters of the generator network are updated.

The value of k is typically small (less than 5), although it is also possible to use $k = 1$. This iterative process is repeated to convergence until Nash equilibrium is reached. At this point, the discriminator will be unable to distinguish between the real and synthetic examples.

There are a few factors that one needs to be careful of during the training. First, if the generator is trained too much without updating the discriminator, it can lead to a situation in which the generator repeatedly produces very similar samples. In other words, there will be very little diversity between the samples produced by the generator. This is the

reason that the training of the generator and discriminator are done simultaneously with interleaving.

Second, the generator will produce poor samples in early iterations and therefore $D(\overline{X})$ will be close to 0. As a result, the loss function will be close to 0, and its gradient will be quite modest. This type is saturation causes slow training of the generator parameters. In such cases, it makes sense to maximize $\log\left[D(\overline{X})\right]$ instead of minimizing $\log\left[1 - D(\overline{X})\right]$ during the early stages of training of the generator parameters. Although this approach is heuristically motivated, and one can no longer write a minimax formulation like Equation 10.19, it tends to work well in practice (especially in the early stages of the training when the discriminator rejects all samples).

10.4.2 Comparison with Variational Autoencoder

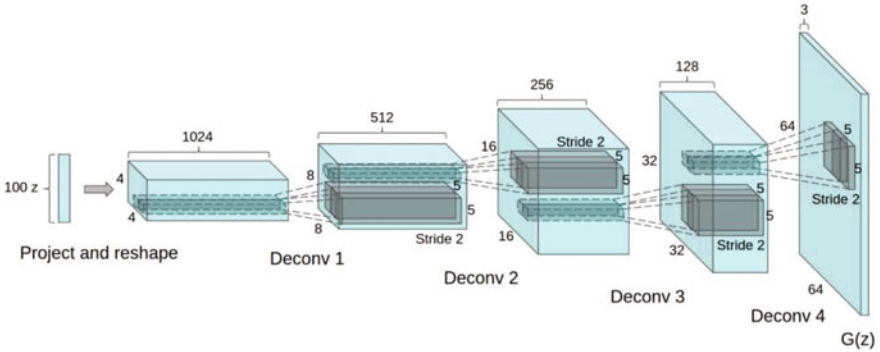
The variational autoencoder and the generative adversarial network were developed independently at around the same time. There are some interesting similarities and differences between these two models. This section will discusses a comparison of these two models.

Unlike a variational autoencoder, only a decoder (i.e., generator) is learned, and an encoder is not learned in the training process of the generative adversarial network. Therefore, a generative adversarial network is not designed to reconstruct specific input samples like a variational autoencoder. However, both models can generate images like the base data, because the hidden space has a known structure (typically Gaussian) from which points can be sampled. In general, the generative adversarial network produces samples of better quality (e.g., less blurry images) than a variational autoencoder. This is because the adversarial approach is specifically designed to produce realistic images, whereas the regularization of the variational autoencoder actually hurts the quality of the generated objects. Furthermore, when reconstruction error is used to create an output for a specific image in the variational autoencoder, it forces the model to average over all plausible outputs. Averaging over plausible outputs, which are often slightly shifted from one another, is a direct cause of blurriness. On the other hand, a method that is specifically designed to produce objects of a quality that fool the discriminator will create a single object in which the different portions are in harmony with one another (and therefore more realistic).

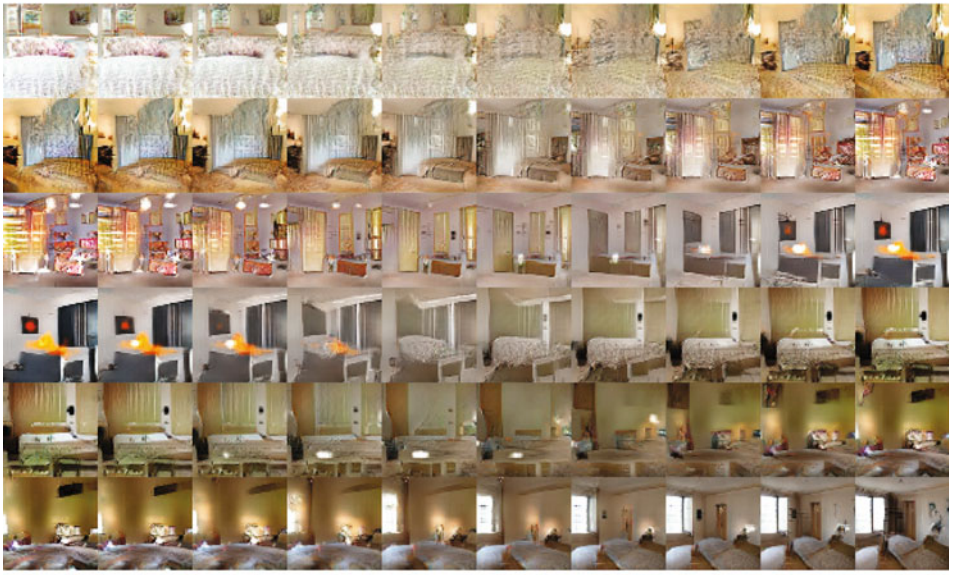
The variational autoencoder is methodologically quite different from the generative adversarial network. The re-parametrization approach used by the variational autoencoder is very useful for training networks with a stochastic nature. Such an approach has the potential to be used in other types of neural network settings with a generative hidden layer. In recent years, some of the ideas in the variational autoencoder have been combined with the ideas in generative adversarial networks.

10.4.3 Using GANs for Generating Image Data

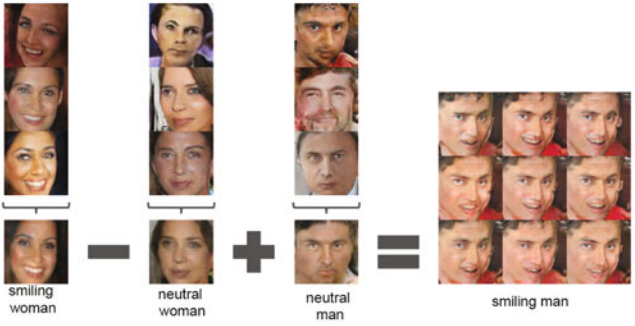
GAN is commonly used is for generating image objects with varying types of context. Indeed, the image setting is, by far, the most common use case of GANs. The generator for the image setting is referred to as a *deconvolutional network*. The most popular way to design a deconvolutional network for the GAN is discussed in [384]. Therefore, the corresponding GAN is also referred to as a DCGAN. It is noteworthy that the term "deconvolution" has generally been replaced by transposed convolution in recent years, because the former term is somewhat misleading.



(a) Convolution architecture of DCGAN



(b) Smooth image transitions caused by changing input noise are shown in each row



(c) Arithmetic operations on input noise have semantic significance

Figure 10.8: The convolutional architecture of DCGAN and generated images. These figures appeared in [A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015]. ©2015 Alec Radford. Used with permission.

The work in [384] starts with 100-dimensional Gaussian noise, which is the starting point of the decoder. This 100-dimensional Gaussian noise is reshaped into 1024 feature maps of size 4×4. This is achieved with a fully connected matrix multiplication with the 100-dimensional input, and the result is reshaped into a tensor. Subsequently, the depth of each layer reduces by a factor of 2, while increasing the lengths and widths by a factor of 2. For example, the second layer contains 512 feature maps, whereas the third layer contains 256 feature maps.

However, increasing length and width with convolution seems odd, because a convolution with even a stride of 1 tends to reduce spatial map size (unless one uses additional zero padding). So how can one use convolutions to increase lengths and widths by a factor of 2? This is achieved by using *fractionally strided convolutions* or *transposed convolutions* at a fractional value of 0.5. These types of transposed convolutions are described at the end of Section 8.5.2 of Chapter 8. The case of fractional strides is not very different from unit strides, and it can be conceptually viewed as a convolution performed after stretching the input volume spatially by either inserting zeros between rows/columns or by inserted interpolated values. Since the input volume is already stretched by a particular factor, applying convolution with stride 1 on this input is equivalent to using fractional strides on the original input. An alternative to the approach of fractionally strided convolutions is to use pooling and unpooling in order to manipulate the spatial footprints. When fractionally strided convolutions are used, no pooling or unpooling needs to be used. An overview of the architecture of the generator in DCGAN is given in Figure 10.8. A detailed discussion of the convolution arithmetic required for fractionally strided convolutions is available in [109].

The generated images are sensitive to the noise samples. Figure 10.8(b) shows examples of the images are generated using the different noise samples. An interesting example is shown in the sixth row in which a room without a window is gradually transformed into one with a large window [384]. Such smooth transitions are also observed in the case of the variational autoencoder. The noise samples are also amenable to vector arithmetic, which is semantically interpretable. For example, one would subtract a noise sample of a neutral woman from that of a smiling woman and add the noise sample of a smiling man. This noise sample is input to the generator in order to obtain an image sample of a smiling man. This example [384] is shown in Figure 10.8(c).

The discriminator also uses a convolutional neural network architecture, except that the leaky ReLU was used instead of the ReLU. The final convolutional layer of the discriminator is flattened and fed into a single sigmoid output. Fully connected layers were not used in either the generator or the discriminator. As is common in convolutional neural networks, the ReLU activation is used. Batch normalization was used in order to reduce any problems with the vanishing and exploding gradient problems [214].

10.4.4 Conditional Generative Adversarial Networks

In conditional adversarial generative networks (CGANs), both the generator and the discriminator are conditioned on an additional input object, which might be a label, a caption, or even another object of the same type. In this case, the input typically correspond to *associated pairs of target objects and contexts*. The contexts are typically related to the target objects in some domain-specific way, which is learned by the model. For example, a context such as "*smiling girl*" might provide an image of a smiling girl. Here, it is important to note that there are many possible choices of images that the CGAN can create for smiling girls, and the specific choice depends on the value of the noise input. Therefore, the CGAN can create a universe of target objects, based on its creativity and imagination. In general, if

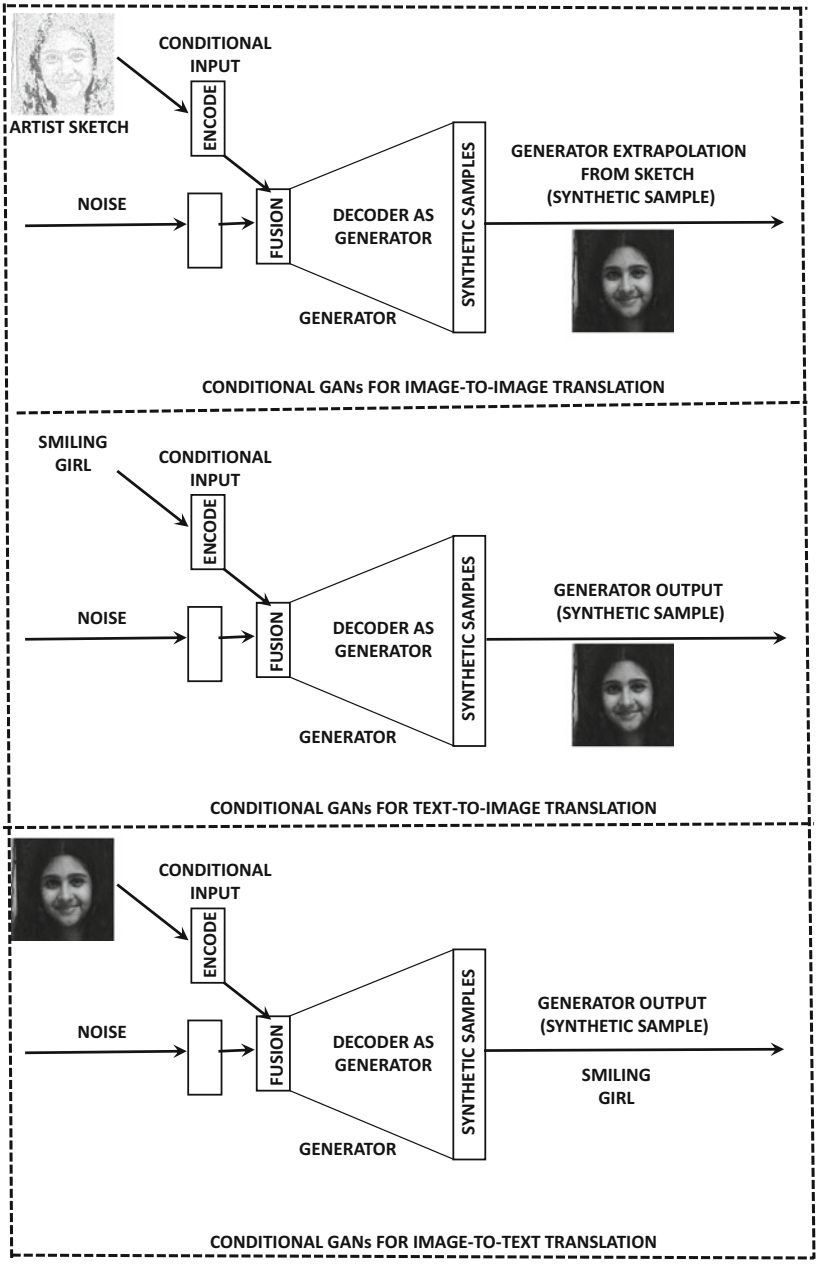


Figure 10.9: Different types of conditional generators for adversarial network. The examples are only illustrative in nature, and they do not reflect actual CGAN output.

the context is more complex than the target output, this universe of target objects tends to shrink, and it is even possible for the generator to output fixed objects irrespective of the noise input to the generator. Therefore, it is more common for the contextual inputs to be simpler than the objects being modeled. For example, it is more common for the context to be a caption and the object to be an image, rather than the converse. Nevertheless, both situations are technically possible.

Examples of different types of conditioning in conditional GANs are shown in Figure 10.9. The context provides the additional input needed for the conditioning. In general, the context may be of any object data type, and the generated output may be of any other data type. The more interesting cases of CGAN use are those in which the context contains much less complexity (e.g., a caption) as compared to the generated output (e.g., image). In such cases, CGANs show a certain level of creativity in filling in missing details. These details can change depending on the noise input to the generator. Some examples of the object-context pairs may be as follows:

1. Each object may be associated with a label. The label provides the conditioning for generating images. For example, in the MNIST data set (cf. Chapter 1), the conditioning might be a label value from 0 to 9, and the generator is expected to create an image of that digit, when provided that conditioning. Similarly, for an image data set, the conditioning might be a label like "*carrot*" and the output would be an image of a carrot. The experiments in the original work on conditional adversarial nets [331] generated a 784-dimensional representation of a digit based on a label from 0 to 9. The base examples of the digits were obtained from the MNIST data set (cf. Section 1.8.1 of Chapter 1).

2. The target object and its context might be of the same type, although the context might be missing the rich level of detail in the target object. For example, the context might be a human artist's sketch of a purse, and the target object might be an actual photograph of the same purse with all details filled in. Another example could be an artist's sketch of a criminal suspect (which is the context), and the target object (output of generator) could be an extrapolation of the actual photograph of the person. The goal is to use a given sketch to generate various realistic samples with details filled in. Such an example is illustrated in the top part of Figure 10.9. When the contextual objects have complex representations such as images or text sentences, they may need to be converted to a multidimensional representation with an encoder, so that they can be fused with multidimensional Gaussian noise. This encoder might be a convolutional network in the case of image context or a recurrent neural network or *word2vec* model in the case of text context.

3. Each object might be associated with a textual description (e.g., image with caption), and the latter provides the context. The caption provides the conditioning for the object. The idea is that by providing a context like "*blue bird with sharp claws*," the generator should provide a fantasy image that reflects this description. An example of an illustrative image generated using the context "*smiling girl*" is illustrated in Figure 10.9. Note that it is also possible to use an image context, and generate a caption for it using a GAN, as shown in the bottom of the figure. However, it is more common to generate complex objects (e.g., images) from simpler contexts (e.g., captions) rather than the reverse. This is because a variety of more accurate supervised learning methods are available when one is trying to generate simple objects (e.g., labels or captions) from complex objects (e.g., images).

4. The base object might be a photograph or video in black and white (e.g., classic movie), and the output object might be the color version of the object. In essence, the GAN learns from examples of such pairs what is the most realistic way of coloring a black-and-white scene. For example, it will use the colors of trees in the training data to give corresponding colors in the generated object without changing its basic outline.

In all these cases, it is evident that GANs are very good at *filling in missing information*. The unconditional GAN is a special case of this setting in which all forms of context are missing, and therefore the GAN is forced to create an image without any information. The conditional case is potentially more interesting from an application-centric point of view because one often has setting where a small amount of partial information is available, and one must extrapolate in a realistic way. When the amount of available context is very small, missing data analysis methods will not work because they require significantly more context to provide reconstructions. On other hand, GANs do not promise faithful reconstructions (like autoencoders or matrix factorization methods), but they provide realistic extrapolations in which missing details are filled into the object in a realistic and harmonious way. As a result, the GAN uses this freedom to generate samples of high quality, rather than a blurred estimation of the average reconstruction. Although a given generation may not perfectly reflect a given context, one can always generate multiple samples in order to explore different types of extrapolations of the same context. For example, given the sketch of a criminal suspect, one might generate different photographs with varying details that are not present in the sketch. In this sense, generative adversarial networks exhibit a certain level of artistry/creativity that is not present in conventional data reconstruction methods. This type of creativity is essential when one is working with only a small amount of context to begin with, and therefore the model needs to be have sufficient freedom to fill in missing details in a reasonable way.

It is noteworthy that a wide variety of machine learning problems (including classification) can be viewed as missing data imputation problems. Technically, the CGAN can be used for these problems as well. However, the CGAN is more useful for specific types of missing data, where the missing portion is too large to be faithfully reconstructed by the model. Although one can even use a CGAN for classification or image captioning, this is obviously not the best use[2] of the generator model, which is tailored towards generative creativity. When the conditioning object is more complex as compared to the output object, it is possible to get into situations where the CGAN generates a fixed output irrespective of input noise.

In the case of the generator, the inputs correspond to a point generated from the noise distribution and the conditional object, which are combined to create a single hidden code. This input is fed into the generator (decoder), which creates a conditioned sample for the data. For the discriminator, the input is a sample from the base data and its context. The base object and its conditional input are first fused into a hidden representation, and the discriminator then provides a classification of whether the same is real or generated. The overall architecture for the training of the generator portion is shown in Figure 10.10. It is instructive to compare this architecture with that of the unconditional GAN in Figure 10.7. The main difference is that an additional conditional input is provided in the second case. The loss function and the overall arrangement of the hidden layers is very similar in both

[2]It turns out that by modifying the *discriminator* to output classes (including the *fake* class), one can obtain state-of-the-art semi-supervised classification with very few labels [420]. However, using the *generator* to output the labels is not a good choice.

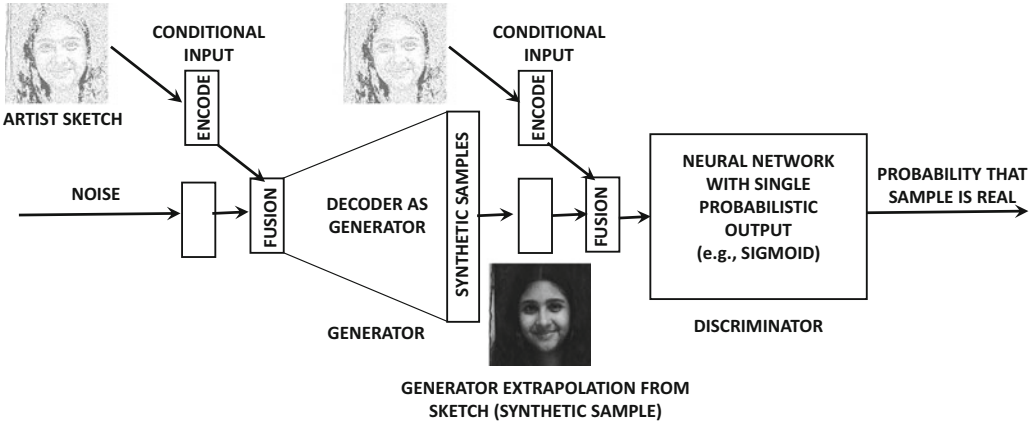


Figure 10.10: Conditional generative adversarial network for hooked up discriminator: It is instructive to compare this architecture with that of the unconditional generative adversarial network in Figure 10.7.

cases. Therefore, the change from an unconditional GAN to a conditional GAN requires only minor changes to the overall architecture. The backpropagation approach remains largely unaffected, except that there are some additional weights in the portion of the neural network associated with the conditioning inputs that one might need to update.

An important point about using GANs with various data types is that they might require some modifications in order to perform the encoding and decoding in a data-sensitive way. While we have given several examples from the image and text domain in our discussion above, most of the description of the algorithm is focussed on vanilla multidimensional data (rather than image or text data). Even when the label is used as the context, it needs to be encoded into a multidimensional representation (e.g., one-hot encoding). Therefore, both Figures 10.9 and 10.10 contain a specifically denoted component for encoding the context. In the earliest work on conditional GANs [331], the pre-trained *AlexNet* convolution network [255] is used as the encoder for image context (without the final label prediction layer). *AlexNet* was pre-trained on the *ImageNet* database. The work in [331] even uses a multimodal setting in which an image is input together with some text annotations. The output is another set of text tags further describing the image. For text annotations, a pre-trained *word2vec* (skip-gram) model is used as the encoder. It is noteworthy that it is even possible to fine-tune the weights of these pre-trained encoder networks while updating the weights of the generator (by backpropagating beyond the generator into the encoder). This is particularly useful if the nature of the data set for object generation in the GAN is very different from the data sets on which the encoders were pretrained. However, the original work in [331] fixed these encoders to their pre-trained configurations, and was still able to generate reasonably high-quality results.

Although the *word2vec* model is used in the specific example above for encoding text, several other options can be used. One option is to use a recurrent neural network, when the input is a full sentence rather than a word. For words, a character-level recurrent network can also be used. In all cases, it is possible to start with an appropriately pre-trained encoder, and then fine-tune it during CGAN training.

10.5 Competitive Learning

Most of the learning methods discussed in this book are based on updating the weights in the neural network in order to correct for errors. Competitive learning is a fundamentally different paradigm in which the goal is not to map inputs to outputs in order to correct errors. Rather, the neurons compete for the right to respond to a subset of similar input data and push their weights closer to one or more input data points. Therefore, the learning process is also very different from the backpropagation algorithm used in neural networks.

The broad idea in training is as follows. The activation of an output neuron increases with greater similarity between the weight vector of the neuron and the input. It is assumed that the weight vector of the neuron has the same dimensionality as the input. A common approach is to use the Euclidian distance between the input and the weight vector in order to compute the activation. Smaller distances lead to larger activations. The output unit that has the highest activation to a given input is declared the winner and moved closer to the input.

In the winner-take-all strategy, only the winning neuron (i.e., neurons with largest activation) is updated and the remaining neurons remain unchanged. Other variants of the competitive learning paradigm allow other neurons to participate in the update based on pre-defined neighborhood relationships. Furthermore, some mechanisms are also available that allow neurons to inhibit one another. These mechanisms are forms of regularization that can be used to learn representations with a specific type of pre-defined structure, which is useful in applications like 2-dimensional visualization. First, we discuss a simple version of the competitive learning algorithm in which the winner-take-all approach is used.

Let $\overline{X}$ be an input vector in d dimensions, and $\overline{W}_i$ be the weight vector associated with the ith neuron in the same number of dimensions. Assume that a total of m neurons is used, where m is typically much less than the size of the data set n. The following steps are used by repeatedly sampling $\overline{X}$ from the input data and making the following computations:

1. The Euclidean distance $||\overline{W}_i - \overline{X}||$ is computed for each i. If the pth neuron has the smallest value of the Euclidean distance, then it is declared as the winner. Note that the value of $||\overline{W}_i - \overline{X}||$ is treated as the activation value of the ith neuron.

2. The pth neuron is updated using the following rule:

$$\overline{W}_p \Leftarrow \overline{W}_p + \alpha(\overline{X} - \overline{W}_p) \tag{10.20}$$

   Here, $\alpha > 0$ is the learning rate. Typically, the value of α is much less than 1. In some cases, the learning rate α reduces with progression of the algorithm.

The basic idea in competitive learning is to view the weight vectors as prototypes (like the centroids in k-means clustering), and then move the (winning) prototype a small distance towards the training instance. The value of α regulates the fraction of the distance between the point and the weight vector, by which the movement of $\overline{W}_p$ occurs. Note that k-means clustering also achieves similar goals, albeit in a different way. After all, when a point is assigned to the winning centroid, it moves that centroid by a small distance towards the training instance at the end of the iteration. Competitive learning allows some natural variations of this framework, which can be used for unsupervised applications like clustering and dimensionality reduction.

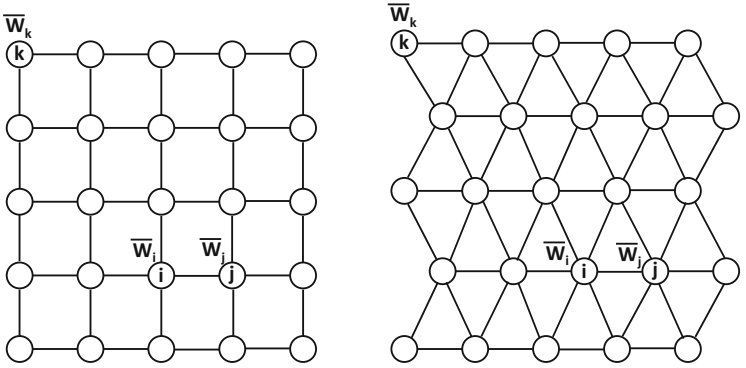
10.5.1 Vector Quantization

Vector quantization is the simplest application of competitive learning. Some changes are made to the basic competitive learning paradigm with the notion of *sensitivity*. Each node has a sensitivity $s_i \geq 0$ associated with it. The sensitivity value helps in balancing the points among different clusters. The basic steps of vector quantization are similar to those in the competitive learning algorithm except for differences caused by how s_i is updated and used in the computations. The value of s_i is initialized to 0 for each point. In each iteration, the value of s_i is increased by $\gamma > 0$ for non-winners and set to 0 for the winner. Furthermore, to choose the winner, the smallest value of $||\overline{W}_i - \overline{X}|| - s_i$ is used. Such an approach tends to make the clusters more balanced, even if the different regions have widely varying density. This approach ensures that points in dense regions are typically very close to one of the weight vectors and the points in sparse regions are approximated very poorly. Such a property is common in applications like dimensionality reduction and compression. The value of γ regulates the effect of sensitivity. Setting γ to 0 reverts to pure competitive learning as discussed above.

The most common application of vector quantization is compression. In compression, each point is represented by its closest weight vector $\overline{W}_i$, where i ranges from 1 to m. Note that the value of m is much less than the number of points n in the data set. The first step is to construct a code book containing the vectors $\overline{W}_1 \ldots \overline{W}_m$, which requires a space of $m \cdot d$ for a data set of dimensionality d. Each point is stored as an index value from 1 through m, depending on its closest weight vector. However, only $\log_2(m)$ bits are required in order to store each data point. Therefore, the overall space requirement is $m \cdot d + \log_2(m)$, which is typically much less than the original space required $n \cdot d$ of the data set. For example, a data set containing 10 billion points in 100 dimensions requires space in the order of 4 Terabytes, if 4 bytes are required for each dimension. On the other hand, by quantizing with $m = 10^6$, the space required for the code-book is less than half a Gigabyte, and 20 bits are required for each point. Therefore, the space required for the points (without the code-book) is less than 3 Gigabytes. Therefore, the overall space requirement is less than 3.5 Gigabytes including the code-book. Note that this type of compression is lossy, and the error of the approximation of the point $\overline{X}$ is $||\overline{X} - \overline{W}_i||$. Points in dense regions are approximated very well, whereas outliers in sparse regions are approximated poorly.

10.5.2 Kohonen Self-Organizing Map

The Kohonen self-organizing map is a variation on the competitive learning paradigm in which a 1-dimensional string-like or 2-dimensional lattice-like structure is imposed on the neurons. For greater generality in discussion, we will consider the case in which a 2-dimensional lattice-like structure is imposed on the neurons. As we will see, this type of lattice structure enables the mapping of all points to 2-dimensional space for visualization. An example of a 2-dimensional lattice structure of 25 neurons arranged in a 5×5 rectangular grid is shown in Figure 10.11(a). A hexagonal lattice containing the same number of neurons is shown in Figure 10.11(b). The shape of the lattice affects the shape of the 2-dimensional regions in which the clusters will be mapped. The case of 1-dimensional string-like structure is similar. The idea of using the lattice structure is that the values of $\overline{W}_i$ in adjacent lattice neurons tend to be similar. Here, it is important to define separate notations to distinguish between the distance $||\overline{W}_i - \overline{W}_j||$ and the distance on the lattice. The distance between adjacent pairs of neurons on the lattice is exactly one unit. For example, the distance between the neurons i and j based on the lattice structure in Figure 10.11(a) is 1 unit, and the



(a) Rectangular (b) Hexagonal

Figure 10.11: An example of a 5×5 lattice structure for the self-organizing map. Since neurons i and j are close in the lattice, the learning process will bias the values of $\overline{W}_i$ and $\overline{W}_j$ to be more similar. The rectangular lattice will lead to rectangular clustered regions in the resulting 2-dimensional representation, whereas the hexagonal lattice will lead to hexagonal clustered regions in the resulting 2-dimensional representation.

distance between neurons i and k is $\sqrt{2^2 + 3^2} = \sqrt{13}$. The vector-distance in the original input space (e.g., $||\overline{X} - \overline{W}_i||$ or $||\overline{W}_i - \overline{W}_j||$) is denoted by a notation like $Dist(\overline{W}_i, \overline{W}_j)$. On the other hand, the distance between neurons i and j along the lattice structure is denoted by $LDist(i, j)$. Note that the value of $LDist(i, j)$ is dependent only on the indices (i, j), and is independent of the values of the vectors $\overline{W}_i$ and $\overline{W}_j$.

The learning process in the self-organizing map is regulated in such a way that the closeness of neurons i and j (based on lattice distance) will also bias their weight vectors to be more similar. In other words, *the lattice structure of the self-organizing maps acts as a regularizer in the learning process.* As we will see later, imposing this type of 2-dimensional structure on the learned weights is helpful for visualizing the original data points with a 2-dimensional embedding.

The overall self-organizing map training algorithm proceeds in a similar way to competitive learning by sampling $\overline{X}$ from the training data, and finding the winner neuron based on the Euclidean distance. The weights in the winner neuron are updated in a manner similar to the vanilla competitive learning algorithm. However, the main difference is that a damped version of this update is also applied to the lattice-neighbors of the winner neuron. In fact, in soft variations of this method, one can apply this update to all neurons, and the level of damping depends on the lattice distance of that neuron to the winning neuron. The damping function, which always lies in $[0, 1]$, is typically defined by a Gaussian kernel:

$$Damp(i, j) = \exp\left(-\frac{LDist(i, j)^2}{2\sigma^2}\right) \tag{10.21}$$

Here, σ is the bandwidth of the Gaussian kernel. Using extremely small values of σ reverts to pure winner-take-all learning, whereas using larger values of σ leads to greater regularization in which lattice-adjacent units have more similar weights. For small values of σ, the damping function will be 1 only for the winner neuron, and it will be 0 for all other neurons. Therefore, the value of σ is one of the parameters available to the user for tuning. Note that many other kernel functions are possible for controlling the regularization and damping. For example,

instead of the smooth Gaussian damping function, one can use a thresholded step kernel, which takes on a value of 1 when $LDist(i, j) < \sigma$, and 0, otherwise.

The training algorithm repeatedly samples $\overline{X}$ from the training data, and computes the distances of $\overline{X}$ to each weight $\overline{W}_i$. The index p of the winning neuron is computed. Rather than applying the update only to $\overline{W}_p$ (as in winner-take-all), the following update is applied to each $\overline{W}_i$:

$$\overline{W}_i \Leftarrow \overline{W}_i + \alpha \cdot Damp(i, p) \cdot (\overline{X} - \overline{W}_i) \quad \forall i \tag{10.22}$$

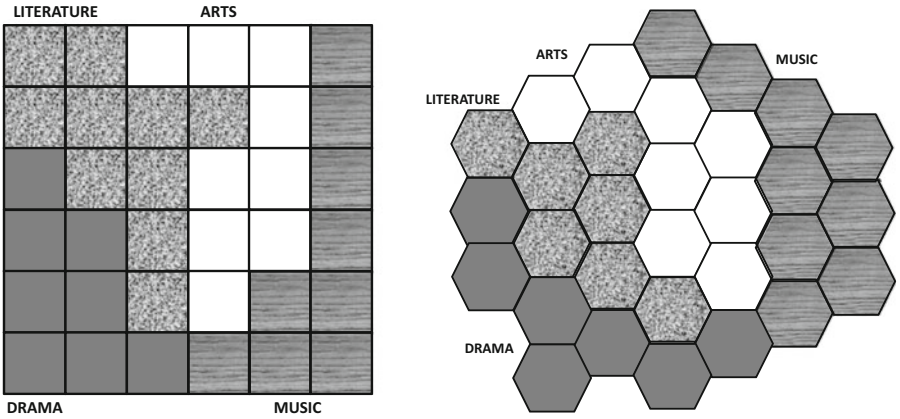
Here, $\alpha > 0$ is the learning rate. It is common to allow the learning rate α to reduce with time. These iterations are continued until convergence is reached. Note that weights that are lattice-adjacent will receive similar updates, and will therefore tend to become more similar over time. *Therefore, the training process forces lattice-adjacent clusters to have similar points, which is useful for visualization.*

Using the Learned Map for 2D Embeddings

The self-organizing map can be used in order to induce a 2-dimensional embedding of the points. For a $k \times k$ grid, all 2-dimensional lattice coordinates will be located in a square in the positive quadrant with vertices $(0, 0)$, $(0, k-1)$, $(k-1, 0)$, and $(k-1, k-1)$. Note that each grid point in the lattice is a vertex with integer coordinates. The simplest 2-dimensional embedding is simply by representing each point $\overline{X}$ with its closest grid point (i.e., winner neuron). However, such an approach will lead to overlapping representations of points. Furthermore, a 2-dimensional representation of the data can be constructed and each coordinate is one of $k \times k$ values from $\{0 \ldots k-1\} \times \{0 \ldots k-1\}$. This is the reason that the self-organizing map is also referred to as a *discretized* dimensionality reduction method. It is possible to use various heuristics to disambiguate these overlapping points. When applied to high-dimensional document data, a visual inspection often shows documents of a particular topic being mapped to a particular local regions. Furthermore, documents of related topics (e.g., politics and elections) tend to get mapped to adjacent regions. Illustrative examples of how a self-organizing map arranges documents of four topics with rectangular and hexagonal lattices are shown in Figure 10.12(a) and (b), respectively. The regions are colored differently, depending on the majority topic of the documents belonging to the corresponding region.

Self-organizing maps have a strong neurobiological basis in terms of their relationship with how the mammalian brain is structured. In the mammalian brain, various types of sensory inputs (e.g., touch) are mapped onto a number of folded planes of cells, which are referred to as *sheets* [129]. When parts of the body that are close together receive an input (e.g., tactile input), then groups of cells that are physically close together in the brain will also fire together. Therefore, proximity in (sensory) inputs is mapped to proximity in neurons, as in the case of the self-organizing map. As with the neurobiological inspiration of convolutional neural networks, such insights are always used for some form of regularization.

Although Kohonen networks are used less often in the modern era of deep learning, they have significant potential in the unsupervised setting. Furthermore, the basic idea of competition can even be incorporated in multi-layer feed-forward networks. Many competitive principles are often combined with more traditional feed-forward networks. For example, the r-sparse and winner-take-all autoencoders (cf. Section 2.5.5.1 of Chapter 2) are both based on competitive principles. Similarly, the notion of local response normalization (cf. Section 8.2.8 of Chapter 8) is based on competition between neurons. Even the notions of attention discussed in this chapter use competitive principles in terms of focusing on a subset of the activations. Therefore, even though the self-organizing map has become



(a) Rectangular lattice (b) Hexagonal lattice

Figure 10.12: Examples of 2-dimensional visualization of documents belonging to four topics

less popular in recent years, the basic principles of competition can also be integrated with traditional feed-forward networks.

10.6 Limitations of Neural Networks

Deep learning has made significant progress in recent years, and has even outperformed humans on many tasks like image classification. Similarly, the success of reinforcement learning to show super-human performance in some games that require sequential planning has been quite extraordinary. Therefore, it is tempting to posit that artificial intelligence might eventually come close to or even exceed the abilities of humans in a more generic way. However, there are several fundamental technical hurdles that need to be crossed before we can build machines that learn and think like people [261]. In particular, neural networks require large amounts of training data to provide high-quality results, which is significantly inferior to human abilities. Furthermore, the amount of energy required by neural networks for various tasks far exceeds that consumed by the human for similar tasks. These observations put fundamental constraints on the abilities of neural networks to exceed certain parameters of human performance. In the following, we discuss these issues along with some recent research directions.

10.6.1 An Aspirational Goal: One-Shot Learning

Although deep learning has received increasing attention in recent years because of its success on large-scale learning tasks (compared to the mediocre performance in early years on smaller data sets), this also exposes an important weakness in current deep learning technology. For tasks like image classification, where deep learning has exceeded human performance, it has done so in a *sample-inefficient fashion*. For example, the *ImageNet* database contains more than a million images, and a neural network will often require thousands of samples of a class in order to properly classify it. Humans do not require tens of thousands of images of a truck, to learn that it is a truck. If a child is shown a truck once, she will often be able to recognize another truck even when it is of a somewhat different model, shape, and color. This suggests that humans have much better ability to generalize

to new settings as compared to artificial neural networks. The general principle of being able to learn from just one or very few examples is referred to as *one-shot learning*.

The ability of humans to generalize with fewer examples is not surprising because the connectivity of the neurons in the human brain is relatively sparse and has been carefully designed by nature. This architecture has evolved over millions of years, and has been passed down from generation to generation. In an indirect sense, the human neural connection structure already encodes a kind of "knowledge" gained from the "evolution experience" over millions of years. Furthermore, humans also gain knowledge over their lifetime over a variety of tasks, which helps them learn specific tasks faster. Subsequently, learning to do *specific* tasks (like recognizing a truck) is simply a fine-tuning of the encoding a person is both born with and which one gains over the course of a lifetime. In other words, humans are masters of transfer learning both within and across generations.

Developing generalized forms of transfer learning, so that the training time spent on particular tasks is not thrown away but is reused is a key area of future research. To a limited extent, the benefits of transfer learning have already been demonstrated in deep learning. As discussed in Chapter 8, convolutional neural networks like *AlexNet* [255] are often pre-trained on large image repositories like *ImageNet*. Subsequently, when the neural network needs to be applied to a new data set, the weights can be fine-tuned with the new data set. Often far fewer number of examples are required for this fine-tuning, because most of the basic features learned in earlier layers do not change with the data set at hand. In many cases, the learned features can also be generalized across tasks by removing the later layers or the network and adding additional task-specific layers. This general principle is also used in text mining. For example, many text feature learning models like *word2vec* are reused across many text mining tasks, even when they were pre-trained on different corpora. In general, the knowledge transfer can be in terms of the extracted features, the model parameters, or other contextual information.

There is another form of transfer learning, which is based on the notion of learning *across tasks*. The basic idea is to always reuse the training work that has already been done either fully or partially in one task in order to improve its ability to learn another task. This principle is referred to as *learning-to-learn*. Thrun and Platt defined [497] learning-to-learn as follows. Given a family of tasks, a training experience for each task, and a family of performance measures (one for each task), an algorithm is said to *learn-to-learn* if its performance at each task improves both with experience *and* the number of tasks. Central to the difficulty of learning-to-learn is the fact that the tasks are all somewhat different and it is therefore challenging to perform experience transfer across tasks. Therefore, the rapid learning occurs within a task, whereas the learning is guided by knowledge gained more gradually across tasks, which captures the way in which task structure varies across target domains [416]. In other words, there is a two-tiered organization of how tasks are learned. This notion is also referred to as *meta-learning*, although this term is overloaded and is used in several other concepts in machine learning. The ability of learning-to-learn is a uniquely biological quality, where living organisms tend to show improved performance even at weakly related tasks, as they gain experience over other tasks. At a weak level, even the pre-training of networks is an example of learning-to-learn, because we can use the weights of the network trained on a particular data set and task to another setting, so that learning in the new setting occurs rapidly. For example, in a convolutional neural network, the features in many of the early layers are primitive shapes (e.g., edges), and they retain their usability irrespective of the kind of task and data set that they are applied on. On the other hand, the final layer might be highly task specific. However, training a single layer requires much less data than the entire network.

Early work on one-shot learning [116] used Bayesian frameworks in order to transfer the learned knowledge from one category to the next. Some successes have been shown at meta-learning with the use of structured architectures that leverage the notions of attention, recursion, and memory. In particular, good results have been shown on the task of learning across categories with neural Turing machines [416]. The ability of memory-augmented networks to learn from limited data has been known for a long time. For example, even networks with internal memory like the LSTM have been shown to exhibit impressive performance for learning never-before seen quadratic functions with a small number of examples. The neural Turning machine is an even better architecture in this respect, and the work in [416] shows how it can be leveraged for meta-learning. Neural Turing machines have also been used to build matching networks for one-shot learning [507]. Even though these works do represent advances in the abilities to perform one-shot learning, the capabilities of these methods are still quite rudimentary compared to humans. Therefore, this topic remains an open area for future research.

10.6.2 An Aspirational Goal: Energy-Efficient Learning

Closely related to the notion of sample efficiency is that of *energy efficiency*. Deep learning systems that work on high-performance hardware are energy inefficient, and require large amounts of power to function. For example, if one uses multiple GPU units in parallel in order to accomplish a compute-intensive task, one might easily use more than a kilowatt of power. On the other hand, a human brain barely requires twenty watts to function, which is much less than the power required by a light bulb. Another point is that the human brain often does not perform detailed computations exactly, but simply makes estimates. In many learning settings, this is sufficient and can sometimes even add to generalization power. This suggests that energy-efficiency may sometimes be found in architectures that emphasize generalization over accuracy.

Several algorithms have recently been developed that trade-off accuracy in computations for improved power-efficiency of computations. Some of these methods also show improved generalization because of the noise effects of the low-precision computations. The work in [83] proposes methods for using binary weights in order to perform efficient computations. An analysis of the effect of using different representational codes on energy efficiency is provided in [289]. Certain types of neural networks, which contain *spiking neurons*, are known to be more energy-efficient [60]. The notion of spiking neurons is directly based on the biological model of the mammalian brain. The basic idea is that the neurons do not fire at each propagation cycle, but they fire only when the *membrane potential* reaches a specific value. The membrane potential is an intrinsic quality of a neuron associated with its electrical charge.

Energy efficiency is often achieved when the size of the neural network is small, and redundant connections are pruned. Removing redundant connections also helps in regularization. The work in [169] proposes to learn weights and connections in neural networks simultaneously by pruning redundant connections. In particular, weights that are close to zero can be removed. As discussed in Chapter 4, training a network to give near-zero weights can be achieved with L_1-regularization. However, the work in [169] shows that L_2-regularization gives higher accuracy. Therefore, the work in [169] uses L_2-regularization and prunes the weights that are below a particular threshold. The pruning is done in an iterative fashion, where the weights are retrained after pruning them, and then the low-weight edges are pruned again. In each iteration, the trained weights from the previous phase are used for the next phase. As a result, the dense network can be sparsified into a network

with far fewer connections. Furthermore, the dead neurons that have zero input connections and output connections are pruned. Further enhancements were reported in [168], where the approach was combined with Huffman coding and quantization for compression. The goal of quantization is to reduce the number of bits representing each connection. This approach reduced the storage required by *AlexNet* [255] by a factor of 35, from about 240MB to 6.9MB with no loss of accuracy. As a result, it becomes possible to fit the model into on-chip SRAM cache rather than off-chip DRAM memory. This has advantages from the perspective of speed, energy efficiency, as well as the ability to perform mobile computation in embedded devices. In particular, a hardware accelerator has been used in [168] in order to achieve these goals, and this acceleration is enabled by the ability to fit the model on the SRAM cache.

Another direction is to develop hardware that is tailored directly to neural networks. It is noteworthy that there is no distinction between software and hardware in humans; while this distinction is helpful from the perspective of computer maintenance, it is also a source of inefficiency that is not shared by the human brain. Simply speaking, the hardware and software are tightly integrated in the brain-inspired model of computing. In recent years, progress has been made in the area of *neuromorphic computing* [114]. This notion is based on a new chip architecture containing spiking neurons, low-precision synapses, and a scalable communication network. Readers are referred to [114] for the description of a convolutional neural network architecture (based on neuromorphic computing) that provides state-of-the-art image-recognition performance.

10.7 Summary

In this chapter, several advanced topics in deep learning have been discussed. The chapter starts with a discussion of attention mechanisms. These mechanisms have been used for both image and text data. In all cases, the incorporation of attention has improved the generalization power of the underlying neural network. Attention mechanisms can also be used to augment computers with external memory. A memory-augmented network has similar theoretical properties as a recurrent neural network in terms of being Turing complete. However, it tends to perform computations in a more interpretable way, and therefore generalizes well to test data sets that are somewhat different from the training data. For example, one can accurately work with longer sequences than the training data set contains in order to perform classification. The simplest example of a memory-augmented network is a neural Turing machine, which has subsequently been generalized to the notion of a differentiable neural computer.

Generative adversarial networks are recent techniques that use an adversarial interaction process between a generative network and a discriminative network in order to generate synthetic samples that are similar to a database of real samples. Such networks can be used as generative models that create input samples for testing machine learning algorithms. In addition, by imposing a conditional on the generative process, it is possible to create samples with different types of contexts. These ideas have been used in various types of applications such as text-to-image and image-to-image translation.

Numerous advanced topics have also been explored in recent years such as one-shot learning and energy-efficient learning. These represent areas in which neural network technology greatly lags the abilities of humans. Although significant advances have been made in recent years, there is significant scope of future research in these areas.

10.8 Bibliographic Notes

Early techniques for using attention in neural network training were proposed in [59, 266]. The recurrent models of visual attention discussed in this chapter are based on the work in [338]. The recognition of multiple objects in an image with visual attention is discussed in [15]. The two most well known models are neural machine translation with attention are discussed in [18, 302]. The ideas of attention have also been extended to image captioning. For example, the work in [540] presents methods for image captioning based on both soft and hard attention models. The use of attention models for text summarization is discussed in [413]. The notion of attention is also useful for focusing on specific parts of the image to enable visual question-answering [395, 539, 542]. A useful mechanism for attention is the use of *spatial transformer networks*, which can selectively crop out or focus on portions of an image. The use of attention models for visual question answering is discussed in [299].

Neural Turing machines [158] and memory networks [473, 528] were proposed around the same time. Subsequently, the neural Turing machine was generalized to a differential neural computer with the use of better memory allocation mechanisms and those for tracking the sequence of writes. The neural Turing machine and differentiable neural computer have been applied to various tasks such as copying, associative recall, sorting, graph querying and language querying. On the other hand, the primary focus of memory networks [473, 528] has been on language understanding and question answering. However, the two architectures are quite similar. The main difference is that the model in [473] focusses on content-based addressing mechanisms rather than location-based mechanisms; doing so reduces the need for sharpening. A more focussed study on the problem of question-answering is provided in [257]. The work in [393] proposes the notion of a neural program interpreter, which is a recurrent and compositional neural network that learns to represent and execute programs. An interesting version of the Turing machine has also been designed with the use of reinforcement learning [550, 551], and it can be used for learning wider classes of complex tasks. The work in [551] shows how simple algorithms can be learned from examples. The parallelization of these methods with GPUs is discussed in [229].

Generative adversarial networks (GANs) have been proposed in [149], and an excellent tutorial on the topic may be found in [145]. An early method proposed a similar architecture for generating chairs with convolutional networks [103]. Improved training algorithms are discussed in [420]. The main challenges in training adversarial networks have to do with *instability* and *saturation*. A theoretical understanding of some of these issues, together with some principled methods for addressing them are discussed in [11, 12]. Energy-based GANs are proposed in [562], it is claimed that they have better stability. Adversarial ideas have also been generalized to autoencoder architectures [311]. Generative adversarial networks are used frequently in the image domain to generate realistic images with various properties [95, 384]. In these cases, a deconvolution network is used in the generator, and therefore the resulting GAN is referred to as a DCGAN. The idea of conditional generative networks and their use in generating objects with context is discussed in [331, 392]. The approach has also been used recently for image to image translation [215, 370, 518]. Although generative adversarial networks are often used in the image domain, they have also been extended recently to sequences [546]. The use of CGANs for predicting the next frame in a video is discussed in [319].

The earliest works on competitive learning may be found in [410, 411]. Gersho and Gray [136] provide an excellent overview of vector quantization methods. Vector quantization methods are alternatives to sparse coding techniques [75]. Kohonen's self-organizing feature map was introduced in [248], and more detailed discussions from the same author

may be found in [249, 250]. Many variants of this basic architecture, such as *neural gas*, are used for incremental learning [126, 317].

A discussion of learning-to-learn methods may be found in [497]. The earliest methods in this area used Bayesian models [116]. Later methods focused on various types of neural Turing machines [416, 507]. Zero-shot learning methods are proposed in [364, 403, 462]. Evolutionary methods can also be used to perform long-term learning [543]. Numerous methods have also been proposed to make deep learning more energy-efficient, such as the use of binary weights [83, 389], specially designed chips [114], and compression mechanisms [213, 168, 169]. Specialized methods have also been developed [68] for convolutional neural networks.

10.8.1 Software Resources

The recurrent model for visual attention is available at [627]. The MATLAB code for the attention mechanism for neural machine translation discussed in this chapter (from the original authors) may be found in [628]. Implementations of the Neural Turing Machine in *TensorFlow* may be found in [629, 630]. The two implementations are related because the approach in [630] adopts some of the portions of [629]. An LSTM controller is used in the original implementation. Implementations in *Keras*, *Lasagne*, and *Torch* may be found in [631, 632, 633]. Several implementations from *Facebook* on memory networks are available at [634]. An implementation of memory networks in *TensorFlow* nay be found in [635]. An implementation of dynamic memory networks in *Theano* and *Lasagne* is available at [636].

An implementation of DCGAN in *TensorFlow* may be found in [637]. In fact, several variants of the GAN (and other topics discussed in this chapter) are available from this contributor [638]. A *Keras* implementation of the GAN may be found in [639]. Implementations of various types of GANs, including the Wasserstein GAN and the variational autoencoder may be found in [640]. These implementations are executed in *PyTorch* and *TensorFlow*. An implementation of the text-to-image GAN in *TensorFlow* is provided in [641], and this implementation is built on top of the aforementioned DCGAN implementation [637].

10.9 Exercises

1. What are the main differences in the approaches used for training hard-attention and soft-attention models?

2. Show how you can use attention models to improve the token-wise classification application of Chapter 7.

3. Discuss how the k-means algorithm is related to competitive learning.

4. Implement a Kohonen self-organizing map with (i) a rectangular lattice, and (ii) a hexagonal lattice.

5. Consider a two-player game like GANs with objective function $f(x, y)$, and we want to compute $\min_x \max_y f(x, y)$. Discuss the relationship between $\min_x \max_y f(x, y)$ and $\max_y \min_x f(x, y)$. When are they equal?

6. Consider the function $f(x, y) = \sin(x + y)$, where we are trying to minimize $f(x, y)$ with respect to x and maximize with respect to y. Implement the alternating process of gradient descent and ascent discussed in the book for GANs to optimize this function. Do you always get the same solution over different starting points?

Bibliography

[1] D. Ackley, G. Hinton, and T. Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive Science*, 9(1), pp. 147–169, 1985.

[2] C. Aggarwal. Data classification: Algorithms and applications, *CRC Press*, 2014.

[3] C. Aggarwal. Data mining: The textbook. *Springer*, 2015.

[4] C. Aggarwal. Recommender systems: The textbook. *Springer*, 2016.

[5] C. Aggarwal. Outlier analysis. *Springer*, 2017.

[6] C. Aggarwal. Machine learning for text. *Springer*, 2018.

[7] R. Ahuja, T. Magnanti, and J. Orlin. Network flows: Theory, algorithms, and applications. *Prentice Hall*, 1993.

[8] E. Aljalbout, V. Golkov, Y. Siddiqui, and D. Cremers. Clustering with deep learning: Taxonomy and new methods. *arXiv:1801.07648*, 2018.
https://arxiv.org/abs/1801.07648

[9] R. Al-Rfou, B. Perozzi, and S. Skiena. Polyglot: Distributed word representations for multilingual nlp. *arXiv:1307.1662*, 2013.
https://arxiv.org/abs/1307.1662

[10] D. Amodei *at al.* Concrete problems in AI safety. *arXiv:1606.06565*, 2016.
https://arxiv.org/abs/1606.06565

[11] M. Arjovsky and L. Bottou. Towards principled methods for training generative adversarial networks. *arXiv:1701.04862*, 2017.
https://arxiv.org/abs/1701.04862

[12] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein gan. *arXiv:1701.07875*, 2017.
https://arxiv.org/abs/1701.07875

[13] J. Ba and R. Caruana. Do deep nets really need to be deep? *NIPS Conference*, pp. 2654–2662, 2014.

[14] J. Ba, J. Kiros, and G. Hinton. Layer normalization. *arXiv:1607.06450*, 2016.
https://arxiv.org/abs/1607.06450

[15] J. Ba, V. Mnih, and K. Kavukcuoglu. Multiple object recognition with visual attention. *arXiv: 1412.7755*, 2014.
https://arxiv.org/abs/1412.7755

[16] A. Babenko, A. Slesarev, A. Chigorin, and V. Lempitsky. Neural codes for image retrieval. *arXiv:1404.1777*, 2014.
https://arxiv.org/abs/1404.1777

[17] M. Baccouche, F. Mamalet, C. Wolf, C. Garcia, and A. Baskurt. Sequential deep learning for human action recognition. *International Workshop on Human Behavior Understanding*, pp. 29–39, 2011.

[18] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *ICLR*, 2015. Also *arXiv:1409.0473*, 2014.
https://arxiv.org/abs/1409.0473

[19] B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. *arXiv:1611.02167*, 2016.
https://arxiv.org/abs/1611.02167

[20] P. Baldi, S. Brunak, P. Frasconi, G. Soda, and G. Pollastri. Exploiting the past and the future in protein secondary structure prediction. *Bioinformatics*, 15(11), pp. 937–946, 1999.

[21] N. Ballas, L. Yao, C. Pal, and A. Courville. Delving deeper into convolutional networks for learning video representations. *arXiv:1511.06432*, 2015.
https://arxiv.org/abs/1511.06432

[22] J. Baxter, A. Tridgell, and L. Weaver. Knightcap: a chess program that learns by combining td (lambda) with game-tree search. *arXiv cs/9901002*, 1999.

[23] M. Bazaraa, H. Sherali, and C. Shetty. Nonlinear programming: theory and algorithms. *John Wiley and Sons*, 2013.

[24] S. Becker, and Y. LeCun. Improving the convergence of back-propagation learning with second order methods. *Proceedings of the 1988 connectionist models summer school*, pp. 29–37, 1988.

[25] M. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47, pp. 253–279, 2013.

[26] R. E. Bellman. Dynamic Programming. *Princeton University Press*, 1957.

[27] Y. Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1), pp. 1–127, 2009.

[28] Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE TPAMI*, 35(8), pp. 1798–1828, 2013.

[29] Y. Bengio and O. Delalleau. Justifying and generalizing contrastive divergence. *Neural Computation*, 21(6), pp. 1601–1621, 2009.

[30] Y. Bengio and O. Delalleau. On the expressive power of deep architectures. *Algorithmic Learning Theory*, pp. 18–36, 2011.

[31] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. *NIPS Conference*, 19, 153, 2007.

[32] Y. Bengio, N. Le Roux, P. Vincent, O. Delalleau, and P. Marcotte. Convex neural networks. *NIPS Conference*, pp. 123–130, 2005.

[33] Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum learning. *ICML Conference*, 2009.

[34] Y. Bengio, L. Yao, G. Alain, and P. Vincent. Generalized denoising auto-encoders as generative models. *NIPS Conference*, pp. 899–907, 2013.

[35] J. Bergstra *et al.* Theano: A CPU and GPU math compiler in Python. *Python in Science Conference*, 2010.

[36] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kegl. Algorithms for hyper-parameter optimization. *NIPS Conference*, pp. 2546–2554, 2011.

[37] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13, pp. 281–305, 2012.

[38] J. Bergstra, D. Yamins, and D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. *ICML Confererence*, pp. 115–123, 2013.

[39] D. Bertsekas. Nonlinear programming *Athena Scientific*, 1999.

[40] C. M. Bishop. Pattern recognition and machine learning. *Springer*, 2007.

[41] C. M. Bishop. Neural networks for pattern recognition. *Oxford University Press*, 1995.

[42] C. M. Bishop. Bayesian Techniques. Chapter 10 in "Neural Networks for Pattern Recognition," pp. 385–439, 1995.

[43] C. M Bishop. Improving the generalization properties of radial basis function neural networks. *Neural Computation*, 3(4), pp. 579–588, 1991.

[44] C. M. Bishop. Training with noise is equivalent to Tikhonov regularization. *Neural computation*, 7(1),pp. 108–116, 1995.

[45] C. M. Bishop, M. Svensen, and C. K. Williams. GTM: A principled alternative to the self-organizing map. *NIPS Conference*, pp. 354–360, 1997.

[46] M. Bojarski *et al.* End to end learning for self-driving cars. *arXiv:1604.07316*, 2016.
https://arxiv.org/abs/1604.07316

[47] M. Bojarski *et al.* Explaining How a Deep Neural Network Trained with End-to-End Learning Steers a Car. *arXiv:1704.07911*, 2017.
https://arxiv.org/abs/1704.07911

[48] H. Bourlard and Y. Kamp. Auto-association by multilayer perceptrons and singular value decomposition. *Biological Cybernetics*, 59(4), pp. 291–294, 1988.

[49] L. Breiman. Random forests. *Journal Machine Learning archive*, 45(1), pp. 5–32, 2001.

[50] L. Breiman. Bagging predictors. *Machine Learning*, 24(2), pp. 123–140, 1996.

[51] D. Broomhead and D. Lowe. Multivariable functional interpolation and adaptive networks. *Complex Systems*, 2, pp. 321–355, 1988.

[52] C. Browne *et al.* A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1), pp. 1–43, 2012.

[53] T. Brox and J. Malik. Large displacement optical flow: descriptor matching in variational motion estimation. *IEEE TPAMI*, 33(3), pp. 500–513, 2011.

[54] A. Bryson. A gradient method for optimizing multi-stage allocation processes. *Harvard University Symposium on Digital Computers and their Applications*, 1961.

[55] C. Bucilu, R. Caruana, and A. Niculescu-Mizil. Model compression. *ACM KDD Conference*, pp. 535–541, 2006.

[56] P. Bühlmann and B. Yu. Analyzing bagging. *Annals of Statistics*, pp. 927–961, 2002.

[57] M. Buhmann. Radial Basis Functions: Theory and implementations. *Cambridge University Press*, 2003.

[58] Y. Burda, R. Grosse, and R. Salakhutdinov. Importance weighted autoencoders. *arXiv:1509.00519*, 2015.
https://arxiv.org/abs/1509.00519

[59] N. Butko and J. Movellan. I-POMDP: An infomax model of eye movement. *IEEE International Conference on Development and Learning*, pp. 139–144, 2008.

[60] Y. Cao, Y. Chen, and D. Khosla. Spiking deep convolutional neural networks for energy-efficient object recognition. *International Journal of Computer Vision*, 113(1), 54–66, 2015.

[61] M. Carreira-Perpinan and G. Hinton. On Contrastive Divergence Learning. *AISTATS*, 10, pp. 33–40, 2005.

[62] S. Chang, W. Han, J. Tang, G. Qi, C. Aggarwal, and T. Huang. Heterogeneous network embedding via deep architectures. *ACM KDD Conference*, pp. 119–128, 2015.

[63] N. Chawla, K. Bowyer, L. Hall, and W. Kegelmeyer. SMOTE: synthetic minority oversampling technique. *Journal of Artificial Intelligence Research*, 16, pp. 321–357, 2002.

[64] J. Chen, S. Sathe, C. Aggarwal, and D. Turaga. Outlier detection with autoencoder ensembles. *SIAM Conference on Data Mining*, 2017.

[65] S. Chen, C. Cowan, and P. Grant. Orthogonal least-squares learning algorithm for radial basis function networks. *IEEE Transactions on Neural Networks*, 2(2), pp. 302–309, 1991.

[66] W. Chen, J. Wilson, S. Tyree, K. Weinberger, and Y. Chen. Compressing neural networks with the hashing trick. *ICML Confererence*, pp. 2285–2294, 2015.

[67] Y. Chen and M. Zaki. KATE: K-Competitive Autoencoder for Text. *ACM KDD Conference*, 2017.

[68] Y. Chen, T. Krishna, J. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1), pp. 127–138, 2017.

[69] K. Cho, B. Merrienboer, C. Gulcehre, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *EMNLP*, 2014.
https://arxiv.org/pdf/1406.1078.pdf

[70] J. Chorowski, D. Bahdanau, D. Serdyuk, K. Cho, and Y. Bengio. Attention-based models for speech recognition. *NIPS Conference*, pp. 577–585, 2015.

[71] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv:1412.3555*, 2014.
https://arxiv.org/abs/1412.3555

[72] D. Ciresan, U. Meier, L. Gambardella, and J. Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural Computation*, 22(12), pp. 3207–3220, 2010.

[73] C. Clark and A. Storkey. Training deep convolutional neural networks to play go. *ICML Confererence*, pp. 1766–1774, 2015.

[74] A. Coates, B. Huval, T. Wang, D. Wu, A. Ng, and B. Catanzaro. Deep learning with COTS HPC systems. *ICML Confererence*, pp. 1337–1345, 2013.

[75] A. Coates and A. Ng. The importance of encoding versus training with sparse coding and vector quantization. *ICML Confererence*, pp. 921–928, 2011.

[76] A. Coates and A. Ng. Learning feature representations with k-means. *Neural networks: Tricks of the Trade*, Springer, pp. 561–580, 2012.

[77] A. Coates, A. Ng, and H. Lee. An analysis of single-layer networks in unsupervised feature learning. *AAAI Conference*, pp. 215–223, 2011.

[78] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12, pp. 2493–2537, 2011.

[79] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. *ICML Conference*, pp. 160–167, 2008.

[80] J. Connor, R. Martin, and L. Atlas. Recurrent neural networks and robust time series prediction. *IEEE Transactions on Neural Networks*, 5(2), pp. 240–254, 1994.

[81] T. Cooijmans, N. Ballas, C. Laurent, C. Gulcehre, and A. Courville. Recurrent batch normalization. *arXiv:1603.09025*, 2016.
https://arxiv.org/abs/1603.09025

[82] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3), pp. 273–297, 1995.

[83] M. Courbariaux, Y. Bengio, and J.-P. David. BinaryConnect: Training deep neural networks with binary weights during propagations. *arXiv:1511.00363*, 2015.
https://arxiv.org/pdf/1511.00363.pdf

[84] T. Cover. Geometrical and statistical properties of systems of linear inequalities with applications to pattern recognition. *IEEE Transactions on Electronic Computers*, pp. 326–334, 1965.

[85] D. Cox and N. Pinto. Beyond simple features: A large-scale feature search approach to unconstrained face recognition. *IEEE International Conference on Automatic Face and Gesture Recognition and Workshops*, pp. 8–15, 2011.

[86] G. Dahl, R. Adams, and H. Larochelle. Training restricted Boltzmann machines on word observations. *arXiv:1202.5695*, 2012.
https://arxiv.org/abs/1202.5695

[87] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. *Computer Vision and Pattern Recognition*, pp. 886–893, 2005.

[88] Y. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *NIPS Conference*, pp. 2933–2941, 2014.

[89] N. de Freitas. Machine Learning, University of Oxford (Course Video), 2013. https://www.youtube.com/watch?v=w2OtwL5T1ow&list=PLE6Wd9FREdyJ5lbFl8Uu–GjecvVw66F6

[90] N. de Freitas. Deep Learning, University of Oxford (Course Video), 2015. https://www.youtube.com/watch?v=PlhFWT7vAEw&list=PLjK8ddCbDMphIMSXn-1IjyYpHU3DaUYw

[91] J. Dean *et al.* Large scale distributed deep networks. *NIPS Conference*, 2012.

[92] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *NIPS Conference*, pp. 3844–3852, 2016.

[93] O. Delalleau and Y. Bengio. Shallow vs. deep sum-product networks. *NIPS Conference*, pp. 666–674, 2011.

[94] M. Denil, B. Shakibi, L. Dinh, M. A. Ranzato, and N. de Freitas. Predicting parameters in deep learning. *NIPS Conference*, pp. 2148–2156, 2013.

[95] E. Denton, S. Chintala, and R. Fergus. Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks. *NIPS Conference*, pp. 1466–1494, 2015.

[96] G. Desjardins, K. Simonyan, and R. Pascanu. Natural neural networks. *NIPS Congference*, pp. 2071–2079, 2015.

[97] F. Despagne and D. Massart. Neural networks in multivariate calibration. *Analyst*, 123(11), pp. 157R–178R, 1998.

[98] T. Dettmers. 8-bit approximations for parallelism in deep learning. *arXiv:1511.04561*, 2015. https://arxiv.org/abs/1511.04561

[99] C. Ding, T. Li, and W. Peng. On the equivalence between non-negative matrix factorization and probabilistic latent semantic indexing. *Computational Statistics and Data Analysis*, 52(8), pp. 3913–3927, 2008.

[100] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell. Long-term recurrent convolutional networks for visual recognition and description. *IEEE conference on computer vision and pattern recognition*, pp. 2625–2634, 2015.

[101] G. Dorffner. Neural networks for time series processing. *Neural Network World*, 1996.

[102] C. Dos Santos and M. Gatti. Deep Convolutional Neural Networks for Sentiment Analysis of Short Texts. *COLING*, pp. 69–78, 2014.

[103] A. Dosovitskiy, J. Tobias Springenberg, and T. Brox. Learning to generate chairs with convolutional neural networks. *CVPR Conference*, pp. 1538–1546, 2015.

[104] A. Dosovitskiy and T. Brox. Inverting visual representations with convolutional networks. *CVPR Conference*, pp. 4829–4837, 2016.

[105] K. Doya. Bifurcations of recurrent neural networks in gradient descent learning. *IEEE Transactions on Neural Networks*, 1, pp. 75–80, 1993.

[106] C. Doersch. Tutorial on variational autoencoders. *arXiv:1606.05908*, 2016. https://arxiv.org/abs/1606.05908

[107] H. Drucker and Y. LeCun. Improving generalization performance using double backpropagation. *IEEE Transactions on Neural Networks*, 3(6), pp. 991–997, 1992.

[108] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12, pp. 2121–2159, 2011.

[109] V. Dumoulin and F. Visin. A guide to convolution arithmetic for deep learning. *arXiv:1603.07285*, 2016.
https://arxiv.org/abs/1603.07285

[110] A. Elkahky, Y. Song, and X. He. A multi-view deep learning approach for cross domain user modeling in recommendation systems. *WWW Conference*, pp. 278–288, 2015.

[111] J. Elman. Finding structure in time. *Cognitive Science*, 14(2), pp. 179–211, 1990.

[112] J. Elman. Learning and development in neural networks: The importance of starting small. *Cognition*, 48, pp. 781–799, 1993.

[113] D. Erhan, Y. Bengio, A. Courville, P. Manzagol, P. Vincent, and S. Bengio. Why does unsupervised pre-training help deep learning?. *Journal of Machine Learning Research*, 11, pp. 625–660, 2010.

[114] S. Essar *et al.* Convolutional neural networks for fast, energy-efficient neuromorphic computing. *Proceedings of the National Academy of Science of the United States of America*, 113(41), pp. 11441–11446, 2016.

[115] A. Fader, L. Zettlemoyer, and O. Etzioni. Paraphrase-Driven Learning for Open Question Answering. *ACL*, pp. 1608–1618, 2013.

[116] L. Fei-Fei, R. Fergus, and P. Perona. One-shot learning of object categories. *IEEE TPAMI*, 28(4), pp. 594–611, 2006.

[117] P. Felzenszwalb, R. Girshick, D. McAllester, and D. Ramanan. Object detection with discriminatively trained part-based models. *IEEE TPAMI*, 32(9), pp. 1627–1645, 2010.

[118] A. Fader, L. Zettlemoyer, and O. Etzioni. Open question answering over curated and extracted knowledge bases. *ACM KDD Conference*, 2014.

[119] A. Fischer and C. Igel. An introduction to restricted Boltzmann machines. *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, pp. 14–36, 2012.

[120] R. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7: pp. 179–188, 1936.

[121] P. Frasconi, M. Gori, and A. Sperduti. A general framework for adaptive processing of data structures. *IEEE Transactions on Neural Networks*, 9(5), pp. 768–786, 1998.

[122] Y. Freund and R. Schapire. A decision-theoretic generalization of online learning and application to boosting. *Computational Learning Theory*, pp. 23–37, 1995.

[123] Y. Freund and R. Schapire. Large margin classification using the perceptron algorithm. *Machine Learning*, 37(3), pp. 277–296, 1999.

[124] Y. Freund and D. Haussler. Unsupervised learning of distributions on binary vectors using two layer networks. *Technical report*, Santa Cruz, CA, USA, 1994

[125] B. Fritzke. Fast learning with incremental RBF networks. *Neural Processing Letters*, 1(1), pp. 2–5, 1994.

[126] B. Fritzke. A growing neural gas network learns topologies. *NIPS Conference*, pp. 625–632, 1995.

[127] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4), pp. 193–202, 1980.

[128] S. Gallant. Perceptron-based learning algorithms. *IEEE Transactions on Neural Networks*, 1(2), pp. 179–191, 1990.

[129] S. Gallant. Neural network learning and expert systems. *MIT Press*, 1993.

[130] H. Gao, H. Yuan, Z. Wang, and S. Ji. Pixel Deconvolutional Networks. *arXiv:1705.06820*, 2017.
https://arxiv.org/abs/1705.06820

[131] L. Gatys, A. S. Ecker, and M. Bethge. Texture synthesis using convolutional neural networks. *NIPS Conference*, pp. 262–270, 2015.

[132] L. Gatys, A. Ecker, and M. Bethge. Image style transfer using convolutional neural networks. *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2414–2423, 2015.

[133] H. Gavin. The Levenberg-Marquardt method for nonlinear least squares curve-fitting problems, 2011.
http://people.duke.edu/~hpgavin/ce281/lm.pdf

[134] P. Gehler, A. Holub, and M. Welling. The Rate Adapting Poisson (RAP) model for information retrieval and object recognition. *ICML Confererence*, 2006.

[135] S. Gelly *et al.* The grand challenge of computer Go: Monte Carlo tree search and extensions. *Communcations of the ACM*, 55, pp. 106–113, 2012.

[136] A. Gersho and R. M. Gray. Vector quantization and signal compression. *Springer Science and Business Media*, 2012.

[137] A. Ghodsi. STAT 946: Topics in Probability and Statistics: Deep Learning, *University of Waterloo*, Fall 2015.
https://www.youtube.com/watch?v=fyAZszlPphs&list=PLehuLRPyt1Hyi78UOkMP–WCGRxGcA9NVOE

[138] W. Gilks, S. Richardson, and D. Spiegelhalter. Markov chain Monte Carlo in practice.*CRC Press*, 1995.

[139] F. Girosi and T. Poggio. Networks and the best approximation property. *Biological Cybernetics*, 63(3), pp. 169–176, 1990.

[140] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *AISTATS*, pp. 249–256, 2010.

[141] X. Glorot, A. Bordes, and Y. Bengio. Deep Sparse Rectifier Neural Networks. *AISTATS*, 15(106), 2011.

[142] P. Glynn. Likelihood ratio gradient estimation: an overview, *Proceedings of the 1987 Winter Simulation Conference*, pp. 366–375, 1987.

[143] Y. Goldberg. A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research (JAIR)*, 57, pp. 345–420, 2016.

[144] C. Goller and A. Küchler. Learning task-dependent distributed representations by backpropagation through structure. *Neural Networks*, 1, pp. 347–352, 1996.

[145] I. Goodfellow. NIPS 2016 tutorial: Generative adversarial networks. *arXiv:1701.00160*, 2016. https://arxiv.org/abs/1701.00160

[146] I. Goodfellow, O. Vinyals, and A. Saxe. Qualitatively characterizing neural network optimization problems. *arXiv:1412.6544*, 2014. [Also appears in *International Conference in Learning Representations*, 2015]
https://arxiv.org/abs/1412.6544

[147] I. Goodfellow, Y. Bengio, and A. Courville. Deep learning. *MIT Press*, 2016.

[148] I. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio. Maxout networks. *arXiv:1302.4389*, 2013.

[149] I. Goodfellow *et al.* Generative adversarial nets. *NIPS Conference*, 2014.

[150] A. Graves, A. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. *Acoustics, Speech and Signal Processing (ICASSP)*, pp. 6645–6649, 2013.

[151] A. Graves. Generating sequences with recurrent neural networks. *arXiv:1308.0850*, 2013. https://arxiv.org/abs/1308.0850

[152] A. Graves. Supervised sequence labelling with recurrent neural networks *Springer*, 2012. http://rd.springer.com/book/10.1007%2F978-3-642-24797-2

[153] A. Graves, S. Fernandez, F. Gomez, and J. Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. *ICML Conferer-ence*, pp. 369–376, 2006.

[154] A. Graves, M. Liwicki, S. Fernandez, R. Bertolami, H. Bunke, and J. Schmidhuber. A novel connectionist system for unconstrained handwriting recognition. *IEEE TPAMI*, 31(5), pp. 855–868, 2009.

[155] A. Graves and J. Schmidhuber. Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures. *Neural Networks*, 18(5–6), pp. 602–610, 2005.

[156] A. Graves and J. Schmidhuber. Offline handwriting recognition with multidimensional recurrent neural networks. *NIPS Conference*, pp. 545–552, 2009.

[157] A. Graves and N. Jaitly. Towards End-To-End Speech Recognition with Recurrent Neural Networks. *ICML Conference*, pp. 1764–1772, 2014.

[158] A. Graves, G. Wayne, and I. Danihelka. Neural turing machines. *arXiv:1410.5401*, 2014. https://arxiv.org/abs/1410.5401

[159] A. Graves *et al.* Hybrid computing using a neural network with dynamic external memory. *Nature*, 538.7626, pp. 471–476, 2016.

[160] K. Greff, R. K. Srivastava, J. Koutnik, B. Steunebrink, and J. Schmidhuber. LSTM: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 2016. http://ieeexplore.ieee.org/abstract/document/7508408/

[161] K. Greff, R. K. Srivastava, and J. Schmidhuber. Highway and residual networks learn unrolled iterative estimation. *arXiv:1612.07771*, 2016.
https://arxiv.org/abs/1612.07771

[162] I. Grondman, L. Busoniu, G. A. Lopes, and R. Babuska. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *IEEE Transactions on Systems, Man, and Cybernetics*, 42(6), pp. 1291–1307, 2012.

[163] R. Girshick, F. Iandola, T. Darrell, and J. Malik. Deformable part models are convolutional neural networks. *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 437–446, 2015.

[164] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. *ACM KDD Conference*, pp. 855–864, 2016.

[165] X. Guo, S. Singh, H. Lee, R. Lewis, and X. Wang. Deep learning for real-time Atari game play using offline Monte-Carlo tree search planning. *Advances in NIPS Conference*, pp. 3338–3346, 2014.

[166] M. Gutmann and A. Hyvarinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. *AISTATS*, 1(2), pp. 6, 2010.

[167] R. Hahnloser and H. S. Seung. Permitted and forbidden sets in symmetric threshold-linear networks. *NIPS Conference*, pp. 217–223, 2001.

[168] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, and W. Dally. EIE: Efficient Inference Engine for Compressed Neural Network. *ACM SIGARCH Computer Architecture News*, 44(3), pp. 243–254, 2016.

[169] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural networks. *NIPS Conference*, pp. 1135–1143, 2015.

[170] L. K. Hansen and P. Salamon. Neural network ensembles. *IEEE TPAMI*, 12(10), pp. 993–1001, 1990.

[171] M. Hardt, B. Recht, and Y. Singer. Train faster, generalize better: Stability of stochastic gradient descent. *ICML Confererence*, pp. 1225–1234, 2006.

[172] B. Hariharan, P. Arbelaez, R. Girshick, and J. Malik. Simultaneous detection and segmentation. *arXiv:1407.1808*, 2014.
https://arxiv.org/abs/1407.1808

[173] E. Hartman, J. Keeler, and J. Kowalski. Layered neural networks with Gaussian hidden units as universal approximations. *Neural Computation*, 2(2), pp. 210–215, 1990.

[174] H. van Hasselt, A. Guez, and D. Silver. Deep Reinforcement Learning with Double Q-Learning. *AAAI Conference*, 2016.

[175] B. Hassibi and D. Stork. Second order derivatives for network pruning: Optimal brain surgeon. *NIPS Conference*, 1993.

[176] D. Hassabis, D. Kumaran, C. Summerfield, and M. Botvinick. Neuroscience-inspired artificial intelligence. *Neuron*, 95(2), pp. 245–258, 2017.

[177] T. Hastie, R. Tibshirani, and J. Friedman. The elements of statistical learning. *Springer*, 2009.

[178] T. Hastie and R. Tibshirani. Generalized additive models. *CRC Press*, 1990.

[179] T. Hastie, R. Tibshirani, and M. Wainwright. Statistical learning with sparsity: the lasso and generalizations. *CRC Press*, 2015.

[180] M. Havaei *et al.* Brain tumor segmentation with deep neural networks. *Medical Image Analysis*, 35, pp. 18–31, 2017.

[181] S. Hawkins, H. He, G. Williams, and R. Baxter. Outlier detection using replicator neural networks. *International Conference on Data Warehousing and Knowledge Discovery*, pp. 170–180, 2002.

[182] S. Haykin. Neural networks and learning machines. *Pearson*, 2008.

[183] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *IEEE International Conference on Computer Vision*, pp. 1026–1034, 2015.

[184] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

[185] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. *European Conference on Computer Vision*, pp. 630–645, 2016.

[186] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T. S. Chua. Neural collaborative filtering. *WWW Conference*, pp. 173–182, 2017.

[187] N. Heess *et al.* Emergence of Locomotion Behaviours in Rich Environments. *arXiv:1707.02286*, 2017.
https://arxiv.org/abs/1707.02286
Video 1 at: https://www.youtube.com/watch?v=hx_bgoTF7bs
Video 2 at: https://www.youtube.com/watch?v=gn4nRCC9TwQ&feature=youtu.be

[188] M. Henaff, J. Bruna, and Y. LeCun. Deep convolutional networks on graph-structured data. *arXiv:1506.05163*, 2015.
https://arxiv.org/abs/1506.05163

[189] M. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6), 1952.

[190] G. Hinton. Connectionist learning procedures. *Artificial Intelligence*, 40(1–3), pp. 185–234, 1989.

[191] G. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8), pp. 1771–1800, 2002.

[192] G. Hinton. To recognize shapes, first learn to generate images. *Progress in Brain Research*, 165, pp. 535–547, 2007.

[193] G. Hinton. A practical guide to training restricted Boltzmann machines. *Momentum*, 9(1), 926, 2010.

[194] G. Hinton. Neural networks for machine learning, *Coursera Video*, 2012.

[195] G. Hinton, P. Dayan, B. Frey, and R. Neal. The wake–sleep algorithm for unsupervised neural networks. *Science*, 268(5214), pp. 1158–1162, 1995.

[196] G. Hinton, S. Osindero, and Y. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7), pp. 1527–1554, 2006.

[197] G. Hinton and T. Sejnowski. Learning and relearning in Boltzmann machines. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, MIT Press, 1986.

[198] G. Hinton and R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313, (5766), pp. 504–507, 2006.

[199] G. Hinton and R. Salakhutdinov. Replicated softmax: an undirected topic model. *NIPS Conference*, pp. 1607–1614, 2009.

[200] G. Hinton and R. Salakhutdinov. A better way to pretrain deep Boltzmann machines. *NIPS Conference*, pp. 2447–2455, 2012.

[201] G. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv:1207.0580*, 2012. https://arxiv.org/abs/1207.0580

[202] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *NIPS Workshop*, 2014.

[203] R. Hochberg. Matrix Multiplication with CUDA: A basic introduction to the CUDA programming model. *Unpublished manuscript*, 2012. http://www.shodor.org/media/content/petascale/materials/UPModules/matrixMultiplication/moduleDocument.pdf

[204] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8), pp. 1735–1785, 1997.

[205] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, *A Field Guide to Dynamical Recurrent Neural Networks*, IEEE Press, 2001.

[206] T. Hofmann. Probabilistic latent semantic indexing. *ACM SIGIR Conference*, pp. 50–57, 1999.

[207] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *National Academy of Sciences of the USA*, 79(8), pp. 2554–2558, 1982.

[208] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5), pp. 359–366, 1989.

[209] Y. Hu, Y. Koren, and C. Volinsky. Collaborative filtering for implicit feedback datasets. *IEEE International Conference on Data Mining*, pp. 263–272, 2008.

[210] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Weinberger. Deep networks with stochastic depth. *European Conference on Computer Vision*, pp. 646–661, 2016.

[211] G. Huang, Z. Liu, K. Weinberger, and L. van der Maaten. Densely connected convolutional networks. *arXiv:1608.06993*, 2016. https://arxiv.org/abs/1608.06993

[212] D. Hubel and T. Wiesel. Receptive fields of single neurones in the cat's striate cortex. *The Journal of Physiology*, 124(3), pp. 574–591, 1959.

[213] F. Iandola, S. Han, M. Moskewicz, K. Ashraf, W. Dally, and K. Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv:1602.07360*, 2016. https://arxiv.org/abs/1602.07360

[214] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv:1502.03167*, 2015.

[215] P. Isola, J. Zhu, T. Zhou, and A. Efros. Image-to-image translation with conditional adversarial networks. *arXiv:1611.07004*, 2016.
https://arxiv.org/abs/1611.07004

[216] M. Iyyer, J. Boyd-Graber, L. Claudino, R. Socher, and H. Daume III. A Neural Network for Factoid Question Answering over Paragraphs. *EMNLP*, 2014.

[217] R. Jacobs. Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1(4), pp. 295–307, 1988.

[218] M. Jaderberg, K. Simonyan, and A. Zisserman. Spatial transformer networks. *NIPS Conference*, pp. 2017–2025, 2015.

[219] H. Jaeger. The "echo state" approach to analysing and training recurrent neural networks – with an erratum note. *German National Research Center for Information Technology GMD Technical Report*, 148(34), 13, 2001.

[220] H. Jaeger and H. Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304, pp. 78–80, 2004.

[221] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? *International Conference on Computer Vision (ICCV)*, 2009.

[222] S. Ji, W. Xu, M. Yang, and K. Yu. 3D convolutional neural networks for human action recognition. *IEEE TPAMI*, 35(1), pp. 221–231, 2013.

[223] Y. Jia *et al.* Caffe: Convolutional architecture for fast feature embedding. *ACM International Conference on Multimedia*, 2014.

[224] C. Johnson. Logistic matrix factorization for implicit feedback data. *NIPS Conference*, 2014.

[225] J. Johnson, A. Karpathy, and L. Fei-Fei. Densecap: Fully convolutional localization networks for dense captioning. *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4565–4574, 2015.

[226] J. Johnson, A. Alahi, and L. Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. *European Conference on Computer Vision*, pp. 694–711, 2015.

[227] R. Johnson and T. Zhang. Effective use of word order for text categorization with convolutional neural networks. *arXiv:1412.1058*, 2014.
https://arxiv.org/abs/1412.1058

[228] R. Jozefowicz, W. Zaremba, and I. Sutskever. An empirical exploration of recurrent network architectures. *ICML Confererence*, pp. 2342–2350, 2015.

[229] L. Kaiser and I. Sutskever. Neural GPUs learn algorithms. *arXiv:1511.08228*, 2015.
https://arxiv.org/abs/1511.08228

[230] S. Kakade. A natural policy gradient. *NIPS Conference*, pp. 1057–1063, 2002.

[231] N. Kalchbrenner and P. Blunsom. Recurrent continuous translation models. *EMNLP*, 3, 39, pp. 413, 2013.

[232] H. Kandel, J. Schwartz, T. Jessell, S. Siegelbaum, and A. Hudspeth. Principles of neural science. *McGraw Hill*, 2012.

[233] A. Karpathy, J. Johnson, and L. Fei-Fei. Visualizing and understanding recurrent networks. *arXiv:1506.02078*, 2015.
https://arxiv.org/abs/1506.02078

[234] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 725–1732, 2014.

[235] A. Karpathy. The unreasonable effectiveness of recurrent neural networks, *Blog post*, 2015.
http://karpathy.github.io/2015/05/21/rnn-effectiveness/

[236] A. Karpathy, J. Johnson, and L. Fei-Fei. Stanford University Class CS321n: Convolutional neural networks for visual recognition, 2016.
http://cs231n.github.io/

[237] H. J. Kelley. Gradient theory of optimal flight paths. *Ars Journal*, 30(10), pp. 947–954, 1960.

[238] F. Khan, B. Mutlu, and X. Zhu. How do humans teach: On curriculum learning and teaching dimension. *NIPS Conference*, pp. 1449–1457, 2011.

[239] T. Kietzmann, P. McClure, and N. Kriegeskorte. Deep Neural Networks In Computational Neuroscience. *bioRxiv, 133504*, 2017.
https://www.biorxiv.org/content/early/2017/05/04/133504

[240] Y. Kim. Convolutional neural networks for sentence classification. *arXiv:1408.5882*, 2014.

[241] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980*, 2014.
https://arxiv.org/abs/1412.6980

[242] D. Kingma and M. Welling. Auto-encoding variational bayes. *arXiv:1312.6114*, 2013.
https://arxiv.org/abs/1312.6114

[243] T. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv:1609.02907*, 2016.
https://arxiv.org/pdf/1609.02907.pdf

[244] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220, pp. 671–680, 1983.

[245] J. Kivinen and M. Warmuth. The perceptron algorithm vs. winnow: linear vs. logarithmic mistake bounds when few input variables are relevant. *Computational Learning Theory*, pp. 289–296, 1995.

[246] L. Kocsis and C. Szepesvari. Bandit based monte-carlo planning. *ECML Conference*, pp. 282–293, 2006.

[247] R. Kohavi and D. Wolpert. Bias plus variance decomposition for zero-one loss functions. *ICML Conference*, 1996.

[248] T. Kohonen. The self-organizing map. Neurocomputing, 21(1), pp. 1–6, 1998.

[249] T. Kohonen. Self-organization and associative memory. *Springer*, 2012.

[250] T. Kohonen. Self-organizing maps, *Springer*, 2001.

[251] D. Koller and N. Friedman. Probabilistic graphical models: principles and techniques. *MIT Press*, 2009.

[252] E. Kong and T. Dietterich. Error-correcting output coding corrects bias and variance. *ICML Conference*, pp. 313–321, 1995.

[253] Y. Koren. Factor in the neighbors: Scalable and accurate collaborative filtering. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 4(1), 1, 2010.

[254] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv:1404.5997*, 2014.
https://arxiv.org/abs/1404.5997

[255] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. *NIPS Conference*, pp. 1097–1105. 2012.

[256] M. Kubat. Decision trees can initialize radial-basis function networks. *IEEE Transactions on Neural Networks*, 9(5), pp. 813–821, 1998.

[257] A. Kumar *et al.* Ask me anything: Dynamic memory networks for natural language processing. *ICML Confererence*, 2016.

[258] Y. Koren. Collaborative filtering with temporal dynamics. *ACM KDD Conference*, pp. 447–455, 2009.

[259] M. Lai. Giraffe: Using deep reinforcement learning to play chess. *arXiv:1509.01549*, 2015.

[260] S. Lai, L. Xu, K. Liu, and J. Zhao. Recurrent Convolutional Neural Networks for Text Classification. *AAAI*, pp. 2267–2273, 2015.

[261] B. Lake, T. Ullman, J. Tenenbaum, and S. Gershman. Building machines that learn and think like people. *Behavioral and Brain Sciences*, pp. 1–101, 2016.

[262] H. Larochelle. Neural Networks (Course). Universite de Sherbrooke, 2013.
https://www.youtube.com/watch?v=SGZ6BttHMPw&list=PL6Xpj9I5qXYEcOhn7–TqghAJ6NAPrNmUBH

[263] H. Larochelle and Y. Bengio. Classification using discriminative restricted Boltzmann machines. *ICML Conference*, pp. 536–543, 2008.

[264] H. Larochelle, M. Mandel, R. Pascanu, and Y. Bengio. Learning algorithms for the classification restricted Boltzmann machine. *Journal of Machine Learning Research*, 13, pp. 643–669, 2012.

[265] H. Larochelle and I. Murray. The neural autoregressive distribution estimator. *International Conference on Artificial Intelligence and Statistics*, pp. 29–37, 2011.

[266] H. Larochelle and G. E. Hinton. Learning to combine foveal glimpses with a third-order Boltzmann machine. *NIPS Conference*, 2010.

[267] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. *ICML Confererence*, pp. 473–480, 2007.

[268] G. Larsson, M. Maire, and G. Shakhnarovich. Fractalnet: Ultra-deep neural networks without residuals. *arXiv:1605.07648*, 2016.
https://arxiv.org/abs/1605.07648

[269] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back. Face recognition: A convolutional neural-network approach. *IEEE Transactions on Neural Networks*, 8(1), pp. 98–113, 1997.

[270] Q. Le *et al.* Building high-level features using large scale unsupervised learning. *ICASSP*, 2013.

[271] Q. Le, N. Jaitly, and G. Hinton. A simple way to initialize recurrent networks of rectified linear units. *arXiv:1504.00941*, 2015.
https://arxiv.org/abs/1504.00941

[272] Q. Le and T. Mikolov. Distributed representations of sentences and documents. *ICML Conference*, pp. 1188–196, 2014.

[273] Q. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Ng, On optimization methods for deep learning. *ICML Conference*, pp. 265–272, 2011.

[274] Q. Le, W. Zou, S. Yeung, and A. Ng. Learning hierarchical spatio-temporal features for action recognition with independent subspace analysis. *CVPR Conference*, 2011.

[275] Y. LeCun. Modeles connexionnistes de l'apprentissage. *Doctoral Dissertation*, Universite Paris, 1987.

[276] Y. LeCun and Y. Bengio. Convolutional networks for images, speech, and time series. *The Handbook of Brain Theory and Neural Networks*, 3361(10), 1995.

[277] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553), pp. 436–444, 2015.

[278] Y. LeCun, L. Bottou, G. Orr, and K. Muller. Efficient backprop. in G. Orr and K. Muller (eds.) *Neural Networks: Tricks of the Trade*, Springer, 1998.

[279] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), pp. 2278–2324, 1998.

[280] Y. LeCun, S. Chopra, R. M. Hadsell, M. A. Ranzato, and F.-J. Huang. A tutorial on energy-based learning. *Predicting Structured Data*, MIT Press, pp. 191–246,, 2006.

[281] Y. LeCun, C. Cortes, and C. Burges. The MNIST database of handwritten digits, 1998.
http://yann.lecun.com/exdb/mnist/

[282] Y. LeCun, J. Denker, and S. Solla. Optimal brain damage. *NIPS Conference*, pp. 598–605, 1990.

[283] Y. LeCun, K. Kavukcuoglu, and C. Farabet. Convolutional networks and applications in vision. *IEEE International Symposium on Circuits and Systems*, pp. 253–256, 2010.

[284] H. Lee, C. Ekanadham, and A. Ng. Sparse deep belief net model for visual area V2. *NIPS Conference*, 2008.

[285] H. Lee, R. Grosse, B. Ranganath, and A. Y. Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. *ICML Conference*, pp. 609–616, 2009.

[286] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39), pp. 1–40, 2016.
Video at: https://sites.google.com/site/visuomotorpolicy/

[287] O. Levy and Y. Goldberg. Neural word embedding as implicit matrix factorization. *NIPS Conference*, pp. 2177–2185, 2014.

[288] O. Levy, Y. Goldberg, and I. Dagan. Improving distributional similarity with lessons learned from word embeddings. *Transactions of the Association for Computational Linguistics*, 3, pp. 211–225, 2015.

[289] W. Levy and R. Baxter. Energy efficient neural codes. *Neural Computation*, 8(3), pp. 531–543, 1996.

[290] M. Lewis, D. Yarats, Y. Dauphin, D. Parikh, and D. Batra. Deal or No Deal? End-to-End Learning for Negotiation Dialogues. *arXiv:1706.05125*, 2017.
https://arxiv.org/abs/1706.05125

[291] J. Li, W. Monroe, A. Ritter, M. Galley,, J. Gao, and D. Jurafsky. Deep reinforcement learning for dialogue generation. *arXiv:1606.01541*, 2016.
https://arxiv.org/abs/1606.01541

[292] L. Li, W. Chu, J. Langford, and R. Schapire. A contextual-bandit approach to personalized news article recommendation. *WWW Conference*, pp. 661–670, 2010.

[293] Y. Li. Deep reinforcement learning: An overview. *arXiv:1701.07274*, 2017.
https://arxiv.org/abs/1701.07274

[294] Q. Liao, K. Kawaguchi, and T. Poggio. Streaming normalization: Towards simpler and more biologically-plausible normalizations for online and recurrent learning. *arXiv:1610.06160*, 2016.
https://arxiv.org/abs/1610.06160

[295] D. Liben-Nowell, and J. Kleinberg. The link-prediction problem for social networks. *Journal of the American Society for Information Science and Technology*, 58(7), pp. 1019–1031, 2007.

[296] L.-J. Lin. Reinforcement learning for robots using neural networks. *Technical Report*, DTIC Document, 1993.

[297] M. Lin, Q. Chen, and S. Yan. Network in network. *arXiv:1312.4400*, 2013.
https://arxiv.org/abs/1312.4400

[298] Z. Lipton, J. Berkowitz, and C. Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv:1506.00019*, 2015.
https://arxiv.org/abs/1506.00019

[299] J. Lu, J. Yang, D. Batra, and D. Parikh. Hierarchical question-image co-attention for visual question answering. *NIPS Conference*, pp. 289–297, 2016.

[300] D. Luenberger and Y. Ye. Linear and nonlinear programming, *Addison-Wesley*, 1984.

[301] M. Lukosevicius and H. Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3), pp. 127–149, 2009.

[302] M. Luong, H. Pham, and C. Manning. Effective approaches to attention-based neural machine translation. *arXiv:1508.04025*, 2015.
https://arxiv.org/abs/1508.04025

[303] J. Ma, R. P. Sheridan, A. Liaw, G. E. Dahl, and V. Svetnik. Deep neural nets as a method for quantitative structure-activity relationships. *Journal of Chemical Information and Modeling*, 55(2), pp. 263–274, 2015.

[304] W. Maass, T. Natschlager, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11), pp. 2351–2560, 2002.

[305] L. Maaten and G. E. Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9, pp. 2579–2605, 2008.

[306] D. J. MacKay. A practical Bayesian framework for backpropagation networks. *Neural Computation*, 4(3), pp. 448–472, 1992.

[307] C. Maddison, A. Huang, I. Sutskever, and D. Silver. Move evaluation in Go using deep convolutional neural networks. *International Conference on Learning Representations*, 2015.

[308] A. Mahendran and A. Vedaldi. Understanding deep image representations by inverting them. *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5188–5196, 2015.

[309] A. Makhzani and B. Frey. K-sparse autoencoders. *arXiv:1312.5663*, 2013.
https://arxiv.org/abs/1312.5663

[310] A. Makhzani and B. Frey. Winner-take-all autoencoders. *NIPS Conference*, pp. 2791–2799, 2015.

[311] A. Makhzani, J. Shlens, N. Jaitly, I. Goodfellow, and B. Frey. Adversarial autoencoders. *arXiv:1511.05644*, 2015.
https://arxiv.org/abs/1511.05644

[312] C. Manning and R. Socher. CS224N: Natural language processing with deep learning. *Stanford University School of Engineering*, 2017.
https://www.youtube.com/watch?v=OQQ-W_63UgQ

[313] J. Martens. Deep learning via Hessian-free optimization. *ICML Conference*, pp. 735–742, 2010.

[314] J. Martens and I. Sutskever. Learning recurrent neural networks with hessian-free optimization. *ICML Conference*, pp. 1033–1040, 2011.

[315] J. Martens, I. Sutskever, and K. Swersky. Estimating the hessian by back-propagating curvature. *arXiv:1206.6464*, 2016.
https://arxiv.org/abs/1206.6464

[316] J. Martens and R. Grosse. Optimizing Neural Networks with Kronecker-factored Approximate Curvature. *ICML Conference*, 2015.

[317] T. Martinetz, S. Berkovich, and K. Schulten. 'Neural-gas' network for vector quantization and its application to time-series prediction. *IEEE Transactions on Neural Network*, 4(4), pp. 558–569, 1993.

[318] J. Masci, U. Meier, D. Ciresan, and J. Schmidhuber. Stacked convolutional auto-encoders for hierarchical feature extraction. *Artificial Neural Networks and Machine Learning*, pp. 52–59, 2011.

[319] M. Mathieu, C. Couprie, and Y. LeCun. Deep multi-scale video prediction beyond mean square error. *arXiv:1511.054*, 2015.
https://arxiv.org/abs/1511.05440

[320] P. McCullagh and J. Nelder. Generalized linear models *CRC Press*, 1989.

[321] W. S. McCulloch and W. H. Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4), pp. 115–133, 1943.

[322] G. McLachlan. Discriminant analysis and statistical pattern recognition *John Wiley & Sons*, 2004.

[323] C. Micchelli. Interpolation of scattered data: distance matrices and conditionally positive definite functions. *Constructive Approximations*, 2, pp. 11–22, 1986.

[324] T. Mikolov. Statistical language models based on neural networks. *Ph.D. thesis, Brno University of Technology*, 2012.

[325] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv:1301.3781*, 2013.
https://arxiv.org/abs/1301.3781

[326] T. Mikolov, A. Joulin, S. Chopra, M. Mathieu, and M. Ranzato. Learning longer memory in recurrent neural networks. *arXiv:1412.7753*, 2014.
https://arxiv.org/abs/1412.7753

[327] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. *NIPS Conference*, pp. 3111–3119, 2013.

[328] T. Mikolov, M. Karafiat, L. Burget, J. Cernocky, and S. Khudanpur. Recurrent neural network based language model. *Interspeech*, Vol 2, 2010.

[329] G. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K. J. Miller. Introduction to WordNet: An on-line lexical database. *International Journal of Lexicography*, 3(4), pp. 235–312, 1990.
https://wordnet.princeton.edu/

[330] M. Minsky and S. Papert. Perceptrons. An Introduction to Computational Geometry, *MIT Press*, 1969.

[331] M. Mirza and S. Osindero. Conditional generative adversarial nets. *arXiv:1411.1784*, 2014.
https://arxiv.org/abs/1411.1784

[332] A. Mnih and G. Hinton. A scalable hierarchical distributed language model. *NIPS Conference*, pp. 1081–1088, 2009.

[333] A. Mnih and K. Kavukcuoglu. Learning word embeddings efficiently with noise-contrastive estimation. *NIPS Conference*, pp. 2265–2273, 2013.

[334] A. Mnih and Y. Teh. A fast and simple algorithm for training neural probabilistic language models. *arXiv:1206.6426*, 2012.
https://arxiv.org/abs/1206.6426

[335] V. Mnih *et al.* Human-level control through deep reinforcement learning. *Nature*, 518 (7540), pp. 529–533, 2015.

[336] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv:1312.5602.*, 2013.
https://arxiv.org/abs/1312.5602

[337] V. Mnih *et al.* Asynchronous methods for deep reinforcement learning. *ICML Confererence*, pp. 1928–1937, 2016.

[338] V. Mnih, N. Heess, and A. Graves. Recurrent models of visual attention. *NIPS Conference*, pp. 2204–2212, 2014.

[339] H. Mobahi and J. Fisher. A theoretical analysis of optimization by Gaussian continuation. *AAAI Conference*, 2015.

[340] G. Montufar. Universal approximation depth and errors of narrow belief networks with discrete units. *Neural Computation*, 26(7), pp. 1386–1407, 2014.

[341] G. Montufar and N. Ay. Refinements of universal approximation results for deep belief networks and restricted Boltzmann machines. *Neural Computation*, 23(5), pp. 1306–1319, 2011.

[342] J. Moody and C. Darken. Fast learning in networks of locally-tuned processing units. *Neural Computation*, 1(2), pp. 281–294, 1989.

[343] A. Moore and C. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13(1), pp. 103–130, 1993.

[344] F. Morin and Y. Bengio. Hierarchical Probabilistic Neural Network Language Model. *AISTATS*, pp. 246–252, 2005.

[345] R. Miotto, F. Wang, S. Wang, X. Jiang, and J. T. Dudley. Deep learning for healthcare: review, opportunities and challenges. *Briefings in Bioinformatics*, pp. 1–11, 2017.

[346] M. Müller, M. Enzenberger, B. Arneson, and R. Segal. Fuego - an open-source framework for board games and Go engine based on Monte-Carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2, pp. 259–270, 2010.

[347] M. Musavi, W. Ahmed, K. Chan, K. Faris, and D. Hummels. On the training of radial basis function classifiers. *Neural Networks*, 5(4), pp. 595–603, 1992.

[348] V. Nair and G. Hinton. Rectified linear units improve restricted Boltzmann machines. *ICML Conference*, pp. 807–814, 2010.

[349] K. S. Narendra and K. Parthasarathy. Identification and control of dynamical systems using neural networks. *IEEE Transactions on Neural Networks*, 1(1), pp. 4–27, 1990.

[350] R. M. Neal. Connectionist learning of belief networks. *Artificial intelligence*, 1992.

[351] R. M. Neal. Probabilistic inference using Markov chain Monte Carlo methods. *Technical Report CRG-TR-93-1*, 1993.

[352] R. M. Neal. Annealed importance sampling. *Statistics and Computing*, 11(2), pp. 125–139, 2001.

[353] Y. Nesterov. A method of solving a convex programming problem with convergence rate $O(1/k^2)$. *Soviet Mathematics Doklady*, 27, pp. 372–376, 1983.

[354] A. Ng. Sparse autoencoder. *CS294A Lecture notes*, 2011.
https://nlp.stanford.edu/~socherr/sparseAutoencoder_2011new.pdf
https://web.stanford.edu/class/cs294a/sparseAutoencoder_2011new.pdf

[355] A. Ng and M. Jordan. PEGASUS: A policy search method for large MDPs and POMDPs. *Uncertainity in Artificial Intelligence*, pp. 406–415, 2000.

[356] J. Y.-H. Ng, M. Hausknecht, S. Vijayanarasimhan, O. Vinyals, R. Monga, and G. Toderici. Beyond short snippets: Deep networks for video classification. *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4694–4702, 2015.

[357] J. Ngiam, A. Khosla, M. Kim, J. Nam, H. Lee, and A. Ng. Multimodal deep learning. *ICML Conference*, pp. 689–696, 2011.

[358] A. Nguyen, A. Dosovitskiy, J. Yosinski, T., Brox, and J. Clune. Synthesizing the preferred inputs for neurons in neural networks via deep generator networks. *NIPS Conference*, pp. 3387–3395, 2016.

[359] J. Nocedal and S. Wright. Numerical optimization. *Springer*, 2006.

[360] S. Nowlan and G. Hinton. Simplifying neural networks by soft weight-sharing. *Neural Computation*, 4(4), pp. 473–493, 1992.

[361] M. Oquab, L. Bottou, I. Laptev, and J. Sivic. Learning and transferring mid-level image representations using convolutional neural networks. *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1717–1724, 2014.

[362] G. Orr and K.-R. Müller (editors). Neural Networks: Tricks of the Trade, *Springer*, 1998.

[363] M. J. L. Orr. Introduction to radial basis function networks, *University of Edinburgh Technical Report, Centre of Cognitive Science*, 1996.
ftp://ftp.cogsci.ed.ac.uk/pub/mjo/intro.ps.Z

[364] M. Palatucci, D. Pomerleau, G. Hinton, and T. Mitchell. Zero-shot learning with semantic output codes. *NIPS Conference*, pp. 1410–1418, 2009.

[365] J. Park and I. Sandberg. Universal approximation using radial-basis-function networks. *Neural Computation*, 3(1), pp. 246–257, 1991.

[366] J. Park and I. Sandberg. Approximation and radial-basis-function networks. *Neural Computation*, 5(2), pp. 305–316, 1993.

[367] O. Parkhi, A. Vedaldi, and A. Zisserman. Deep Face Recognition. *BMVC*, 1(3), pp. 6, 2015.

[368] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. *ICML Conference*, 28, pp. 1310–1318, 2013.

[369] R. Pascanu, T. Mikolov, and Y. Bengio. Understanding the exploding gradient problem. *CoRR, abs/1211.5063*, 2012.

[370] D. Pathak, P. Krahenbuhl, J. Donahue, T. Darrell, and A. A. Efros. Context encoders: Feature learning by inpainting. *CVPR Conference*, 2016.

[371] J. Pennington, R. Socher, and C. Manning. Glove: Global Vectors for Word Representation. *EMNLP*, pp. 1532–1543, 2014.

[372] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. *ACM KDD Conference*, pp. 701–710.

[373] C. Peterson and J. Anderson. A mean field theory learning algorithm for neural networks. *Complex Systems*, 1(5), pp. 995–1019, 1987.

[374] J. Peters and S. Schaal. Reinforcement learning of motor skills with policy gradients. *Neural Networks*, 21(4), pp. 682–697, 2008.

[375] F. Pineda. Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, 59(19), 2229, 1987.

[376] E. Polak. Computational methods in optimization: a unified approach. *Academic Press*, 1971.

[377] L. Polanyi and A. Zaenen. Contextual valence shifters. *Computing Attitude and Affect in Text: Theory and Applications*, pp. 1–10, Springer, 2006.

[378] G. Pollastri, D. Przybylski, B. Rost, and P. Baldi. Improving the prediction of protein secondary structure in three and eight classes using recurrent neural networks and profiles. *Proteins: Structure, Function, and Bioinformatics*, 47(2), pp. 228–235, 2002.

[379] J. Pollack. Recursive distributed representations. *Artificial Intelligence*, 46(1), pp. 77–105, 1990.

[380] B. Polyak and A. Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4), pp. 838–855, 1992.

[381] D. Pomerleau. ALVINN, an autonomous land vehicle in a neural network. *Technical Report*, Carnegie Mellon University, 1989.

[382] B. Poole, J. Sohl-Dickstein, and S. Ganguli. Analyzing noise in autoencoders and deep networks. *arXiv:1406.1831*, 2014.
https://arxiv.org/abs/1406.1831

[383] H. Poon and P. Domingos. Sum-product networks: A new deep architecture. *Computer Vision Workshops (ICCV Workshops)*, pp. 689–690, 2011.

[384] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv:1511.06434*, 2015.
https://arxiv.org/abs/1511.06434

[385] A. Rahimi and B. Recht. Random features for large-scale kernel machines. *NIPS Conference*, pp. 1177–1184, 2008.

[386] M.' A. Ranzato, Y-L. Boureau, and Y. LeCun. Sparse feature learning for deep belief networks. *NIPS Conference*, pp. 1185–1192, 2008.

[387] M.' A. Ranzato, F. J. Huang, Y-L. Boureau, and Y. LeCun. Unsupervised learning of invariant feature hierarchies with applications to object recognition. *Computer Vision and Pattern Recognition*, pp. 1–8, 2007.

[388] A. Rasmus, M. Berglund, M. Honkala, H. Valpola, and T. Raiko. Semi-supervised learning with ladder networks. *NIPS Conference*, pp. 3546–3554, 2015.

[389] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *European Conference on Computer Vision*, pp. 525–542, 2016.

[390] A. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson. CNN features off-the-shelf: an astounding baseline for recognition. *IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 806–813, 2014.

[391] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 779–788, 2016.

[392] S. Reed, Z. Akata, X. Yan, L. Logeswaran, B. Schiele, and H. Lee. Generative adversarial text to image synthesis. *ICML Conference*, pp. 1060–1069, 2016.

[393] S. Reed and N. de Freitas. Neural programmer-interpreters. *arXiv:1511.06279*, 2015.

[394] R. Rehurek and P. Sojka. Software framework for topic modelling with large corpora. *LREC 2010 Workshop on New Challenges for NLP Frameworks*, pp. 45–50, 2010.
https://radimrehurek.com/gensim/index.html

[395] M. Ren, R. Kiros, and R. Zemel. Exploring models and data for image question answering. *NIPS Conference*, pp. 2953–2961, 2015.

[396] S. Rendle. Factorization machines. *IEEE ICDM Conference*, pp. 995–100, 2010.

[397] S. Rifai, P. Vincent, X. Muller, X. Glorot, and Y. Bengio. Contractive auto-encoders: Explicit invariance during feature extraction. *ICML Conference*, pp. 833–840, 2011.

[398] S. Rifai, Y. Dauphin, P. Vincent, Y. Bengio, and X. Muller. The manifold tangent classifier. *NIPS Conference*, pp. 2294–2302, 2011.

[399] D. Rezende, S. Mohamed, and D. Wierstra. Stochastic backpropagation and approximate inference in deep generative models. *arXiv:1401.4082*, 2014. https://arxiv.org/abs/1401.4082

[400] R. Rifkin. Everything old is new again: a fresh look at historical approaches in machine learning. *Ph.D. Thesis*, Massachusetts Institute of Technology, 2002.

[401] R. Rifkin and A. Klautau. In defense of one-vs-all classification. *Journal of Machine Learning Research*, 5, pp. 101–141, 2004.

[402] V. Romanuke. Parallel Computing Center (Khmelnitskiy, Ukraine) represents an ensemble of 5 convolutional neural networks which performs on MNIST at 0.21 percent error rate. Retrieved 24 November 2016.

[403] B. Romera-Paredes and P. Torr. An embarrassingly simple approach to zero-shot learning. *ICML Confererence*, pp. 2152–2161, 2015.

[404] X. Rong. word2vec parameter learning explained. *arXiv:1411.2738*, 2014. https://arxiv.org/abs/1411.2738

[405] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6), 386, 1958.

[406] D. Ruck, S. Rogers, and M. Kabrisky. Feature selection using a multilayer perceptron. *Journal of Neural Network Computing*, 2(2), pp. 40–88, 1990.

[407] H. A. Rowley, S. Baluja, and T. Kanade. Neural network-based face detection. *IEEE TPAMI*, 20(1), pp. 23–38, 1998.

[408] D. Rumelhart, G. Hinton, and R. Williams. Learning representations by back-propagating errors. *Nature*, 323 (6088), pp. 533–536, 1986.

[409] D. Rumelhart, G. Hinton, and R. Williams. Learning internal representations by back-propagating errors. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, pp. 318–362, 1986.

[410] D. Rumelhart, D. Zipser, and J. McClelland. Parallel Distributed Processing, *MIT Press*, pp. 151–193, 1986.

[411] D. Rumelhart and D. Zipser. Feature discovery by competitive learning. *Cognitive science*, 9(1), pp. 75–112, 1985.

[412] G. Rummery and M. Niranjan. Online Q-learning using connectionist systems (Vol. 37). *University of Cambridge, Department of Engineering*, 1994.

[413] A. M. Rush, S. Chopra, and J. Weston. A Neural Attention Model for Abstractive Sentence Summarization. *arXiv:1509.00685*, 2015. https://arxiv.org/abs/1509.00685

[414] R. Salakhutdinov, A. Mnih, and G. Hinton. Restricted Boltzmann machines for collaborative filtering. *ICML Confererence*, pp. 791–798, 2007.

[415] R. Salakhutdinov and G. Hinton. Semantic Hashing. *SIGIR workshop on Information Retrieval and applications of Graphical Models*, 2007.

[416] A. Santoro, S. Bartunov, M. Botvinick, D. Wierstra, and T. Lillicrap. One shot learning with memory-augmented neural networks. *arXiv: 1605:06065*, 2016. https://www.arxiv.org/pdf/1605.06065.pdf

[417] R. Salakhutdinov and G. Hinton. Deep Boltzmann machines. *Artificial Intelligence and Statistics*, pp. 448–455, 2009.

[418] R. Salakhutdinov and H. Larochelle. Efficient Learning of Deep Boltzmann Machines. *AISTATs*, pp. 693–700, 2010.

[419] T. Salimans and D. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *NIPS Conference*, pp. 901–909, 2016.

[420] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen. Improved techniques for training gans. *NIPS Conference*, pp. 2234–2242, 2016.

[421] A. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3, pp. 210–229, 1959.

[422] T Sanger. Neural network learning control of robot manipulators using gradually increasing task difficulty. *IEEE Transactions on Robotics and Automation*, 10(3), 1994.

[423] H. Sarimveis, A. Alexandridis, and G. Bafas. A fast training algorithm for RBF networks based on subtractive clustering. *Neurocomputing*, 51, pp. 501–505, 2003.

[424] W. Saunders, G. Sastry, A. Stuhlmueller, and O. Evans. Trial without Error: Towards Safe Reinforcement Learning via Human Intervention. *arXiv:1707.05173*, 2017.
https://arxiv.org/abs/1707.05173

[425] A. Saxe, P. Koh, Z. Chen, M. Bhand, B. Suresh, and A. Ng. On random weights and unsupervised feature learning. *ICML Confererence*, pp. 1089–1096, 2011.

[426] A. Saxe, J. McClelland, and S. Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv:1312.6120*, 2013.

[427] S. Schaal. Is imitation learning the route to humanoid robots? *Trends in Cognitive Sciences*, 3(6), pp. 233–242, 1999.

[428] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv:1511.05952*, 2015.
https://arxiv.org/abs/1511.05952

[429] T. Schaul, S. Zhang, and Y. LeCun. No more pesky learning rates. *ICML Confererence*, pp. 343–351, 2013.

[430] B. Schölkopf, K. Sung, C. Burges, F. Girosi, P. Niyogi, T. Poggio, and V. Vapnik. Comparing support vector machines with Gaussian kernels to radial basis function classifiers. *IEEE Transactions on Signal Processing*, 45(11), pp. 2758–2765, 1997.

[431] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61, pp. 85–117, 2015.

[432] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. *ICML Conference*, 2015.

[433] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *ICLR Conference*, 2016.

[434] M. Schuster and K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11), pp. 2673–2681, 1997.

[435] H. Schwenk and Y. Bengio. Boosting neural networks. *Neural Computation*, 12(8), pp. 1869–1887, 2000.

[436] S. Sedhain, A. K. Menon, S. Sanner, and L. Xie. Autorec: Autoencoders meet collaborative filtering. *WWW Conference*, pp. 111–112, 2015.

[437] T. J. Sejnowski. Higher-order Boltzmann machines. *AIP Conference Proceedings*, 15(1), pp. 298–403, 1986.

[438] G. Seni and J. Elder. Ensemble methods in data mining: Improving accuracy through combining predictions. *Morgan and Claypool*, 2010.

[439] I. Serban, A. Sordoni, R. Lowe, L. Charlin, J. Pineau, A. Courville, and Y. Bengio. A hierarchical latent variable encoder-decoder model for generating dialogues. *AAAI*, pp. 3295–3301, 2017.

[440] I. Serban, A. Sordoni, Y. Bengio, A. Courville, and J. Pineau. Building end-to-end dialogue systems using generative hierarchical neural network models. *AAAI Conference*, pp. 3776–3784, 2016.

[441] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv:1312.6229*, 2013. https://arxiv.org/abs/1312.6229

[442] A. Shashua. On the equivalence between the support vector machine for classification and sparsified Fisher's linear discriminant. *Neural Processing Letters*, 9(2), pp. 129–139, 1999.

[443] J. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. *Technical Report, CMU-CS-94-125*, Carnegie-Mellon University, 1994.

[444] H. Siegelmann and E. Sontag. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1), pp. 132–150, 1995.

[445] D. Silver *et al.* Mastering the game of Go with deep neural networks and tree search. *Nature*, 529.7587, pp. 484–489, 2016.

[446] D. Silver *et al.* Mastering the game of go without human knowledge. *Nature*, 550.7676, pp. 354–359, 2017.

[447] D. Silver *et al.* Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv*, 2017.
https://arxiv.org/abs/1712.01815

[448] S. Shalev-Shwartz, Y. Singer, N. Srebro, and A. Cotter. Pegasos: Primal estimated subgradient solver for SVM. *Mathematical Programming*, 127(1), pp. 3–30, 2011.

[449] E. Shelhamer, J., Long, and T. Darrell. Fully convolutional networks for semantic segmentation. *IEEE TPAMI*, 39(4), pp. 640–651, 2017.

[450] J. Sietsma and R. Dow. Creating artificial neural networks that generalize. *Neural Networks*, 4(1), pp. 67–79, 1991.

[451] B. W. Silverman. Density Estimation for Statistics and Data Analysis. *Chapman and Hall*, 1986.

[452] P. Simard, D. Steinkraus, and J. C. Platt. Best practices for convolutional neural networks applied to visual document analysis. *ICDAR*, pp. 958–962, 2003.

[453] H. Simon. The Sciences of the Artificial. *MIT Press*, 1996.

[454] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556*, 2014.
https://arxiv.org/abs/1409.1556

[455] K. Simonyan and A. Zisserman. Two-stream convolutional networks for action recognition in videos. *NIPS Conference*, pp. 568–584, 2014.

[456] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv:1312.6034*, 2013.

[457] P. Smolensky. Information processing in dynamical systems: Foundations of harmony theory. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Volume 1: Foundations. pp. 194–281, 1986.

[458] J. Snoek, H. Larochelle, and R. Adams. Practical bayesian optimization of machine learning algorithms. *NIPS Conference*, pp. 2951–2959, 2013.

[459] R. Socher, C. Lin, C. Manning, and A. Ng. Parsing natural scenes and natural language with recursive neural networks. *ICML Confererence*, pp. 129–136, 2011.

[460] R. Socher, J. Pennington, E. Huang, A. Ng, and C. Manning. Semi-supervised recursive autoencoders for predicting sentiment distributions. *Empirical Methods in Natural Language Processing (EMNLP)*, pp. 151–161, 2011.

[461] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. Manning, A. Ng, and C. Potts. Recursive deep models for semantic compositionality over a sentiment treebank. *Empirical Methods in Natural Language Processing (EMNLP)*, p. 1642, 2013.

[462] Socher, Richard, Milind Ganjoo, Christopher D. Manning, and Andrew Ng. Zero-shot learning through cross-modal transfer. *NIPS Conference*, pp. 935–943, 2013.

[463] K. Sohn, H. Lee, and X. Yan. Learning structured output representation using deep conditional generative models. *NIPS Conference*, 2015.

[464] R. Solomonoff. A system for incremental learning based on algorithmic probability. *Sixth Israeli Conference on Artificial Intelligence, Computer Vision and Pattern Recognition*, pp. 515–527, 1994.

[465] Y. Song, A. Elkahky, and X. He. Multi-rate deep learning for temporal recommendation. *ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 909–912, 2016.

[466] J. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for simplicity: The all convolutional net. *arXiv:1412.6806*, 2014.
https://arxiv.org/abs/1412.6806

[467] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1), pp. 1929–1958, 2014.

[468] N. Srivastava and R. Salakhutdinov. Multimodal learning with deep Boltzmann machines. *NIPS Conference*, pp. 2222–2230, 2012.

[469] N. Srivastava, R. Salakhutdinov, and G. Hinton. Modeling documents with deep Boltzmann machines. *Uncertainty in Artificial Intelligence*, 2013.

[470] R. K. Srivastava, K. Greff, and J. Schmidhuber. Highway networks. *arXiv:1505.00387*, 2015.
https://arxiv.org/abs/1505.00387

[471] A. Storkey. Increasing the capacity of a Hopfield network without sacrificing functionality. *Artificial Neural Networks*, pp. 451–456, 1997.

[472] F. Strub and J. Mary. Collaborative filtering with stacked denoising autoencoders and sparse inputs. *NIPS Workshop on Machine Learning for eCommerce*, 2015.

[473] S. Sukhbaatar, J. Weston, and R. Fergus. End-to-end memory networks. *NIPS Conference*, pp. 2440–2448, 2015.

[474] Y. Sun, D. Liang, X. Wang, and X. Tang. Deepid3: Face recognition with very deep neural networks. *arXiv:1502.00873*, 2013.
https://arxiv.org/abs/1502.00873

[475] Y. Sun, X. Wang, and X. Tang. Deep learning face representation from predicting 10,000 classes. *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1891–1898, 2014.

[476] M. Sundermeyer, R. Schluter, and H. Ney. LSTM neural networks for language modeling. *Interspeech*, 2010.

[477] M. Sundermeyer, T. Alkhouli, J. Wuebker, and H. Ney. Translation modeling with bidirectional recurrent neural networks. *EMNLP*, pp. 14–25, 2014.

[478] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. *ICML Confererence*, pp. 1139–1147, 2013.

[479] I. Sutskever and T. Tieleman. On the convergence properties of contrastive divergence. *International Conference on Artificial Intelligence and Statistics*, pp. 789–795, 2010.

[480] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. *NIPS Conference*, pp. 3104–3112, 2014.

[481] I. Sutskever and V. Nair. Mimicking Go experts with convolutional neural networks. *International Conference on Artificial Neural Networks*, pp. 101–110, 2008.

[482] R. Sutton. Learning to Predict by the Method of Temporal Differences, *Machine Learning*, 3, pp. 9–44, 1988.

[483] R. Sutton and A. Barto. Reinforcement Learning: An Introduction. *MIT Press*, 1998.

[484] R. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. *NIPS Conference*, pp. 1057–1063, 2000.

[485] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–9, 2015.

[486] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2818–2826, 2016.

[487] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. *AAAI Conference*, pp. 4278–4284, 2017.

[488] G. Taylor, R. Fergus, Y. LeCun, and C. Bregler. Convolutional learning of spatio-temporal features. *European Conference on Computer Vision*, pp. 140–153, 2010.

[489] G. Taylor, G. Hinton, and S. Roweis. Modeling human motion using binary latent variables. *NIPS Conference*, 2006.

[490] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. *ACM KDD Conference*, pp. 847–855, 2013.

[491] T. Tieleman. Training restricted Boltzmann machines using approximations to the likelihood gradient. *ICML Conference*, pp. 1064–1071, 2008.

[492] G. Tesauro. Practical issues in temporal difference learning. *Advances in NIPS Conference*, pp. 259–266, 1992.

[493] G. Tesauro. Td-gammon: A self-teaching backgammon program. *Applications of Neural Networks*, Springer, pp. 267–285, 1992.

[494] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3), pp. 58–68, 1995.

[495] Y. Teh and G. Hinton. Rate-coded restricted Boltzmann machines for face recognition. *NIPS Conference*, 2001.

[496] S. Thrun. Learning to play the game of chess *NIPS Conference*, pp. 1069–1076, 1995.

[497] S. Thrun and L. Platt. Learning to learn. *Springer*, 2012.

[498] Y. Tian, Q. Gong, W. Shang, Y. Wu, and L. Zitnick. ELF: An extensive, lightweight and flexible research platform for real-time strategy games. *arXiv:1707.01067*, 2017. https://arxiv.org/abs/1707.01067

[499] A. Tikhonov and V. Arsenin. Solution of ill-posed problems. *Winston and Sons*, 1977.

[500] D. Tran *et al.* Learning spatiotemporal features with 3d convolutional networks. *IEEE International Conference on Computer Vision*, 2015.

[501] R. Uijlings, A. van de Sande, T. Gevers, and M. Smeulders. Selective search for object recognition. *International Journal of Computer Vision*, 104(2), 2013.

[502] H. Valpola. From neural PCA to deep unsupervised learning. *Advances in Independent Component Analysis and Learning Machines*, pp. 143–171, Elsevier, 2015.

[503] A. Vedaldi and K. Lenc. Matconvnet: Convolutional neural networks for matlab. *ACM International Conference on Multimedia*, pp. 689–692, 2005. http://www.vlfeat.org/matconvnet/

[504] V. Veeriah, N. Zhuang, and G. Qi. Differential recurrent neural networks for action recognition. *IEEE International Conference on Computer Vision*, pp. 4041–4049, 2015.

[505] A. Veit, M. Wilber, and S. Belongie. Residual networks behave like ensembles of relatively shallow networks. *NIPS Conference*, pp. 550–558, 2016.

[506] P. Vincent, H. Larochelle, Y. Bengio, and P. Manzagol. Extracting and composing robust features with denoising autoencoders. ICML Confererence, pp. 1096–1103, 2008.

[507] O. Vinyals, C. Blundell, T. Lillicrap, and D. Wierstra. Matching networks for one-shot learning. *NIPS Conference*, pp. 3530–3638, 2016.

[508] O. Vinyals and Q. Le. A Neural Conversational Model. *arXiv:1506.05869*, 2015. https://arxiv.org/abs/1506.05869

[509] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. *CVPR Conference*, pp. 3156–3164, 2015.

[510] J. Walker, C. Doersch, A. Gupta, and M. Hebert. An uncertain future: Forecasting from static images using variational autoencoders. *European Conference on Computer Vision*, pp. 835–851, 2016.

[511] L. Wan, M. Zeiler, S. Zhang, Y. LeCun, and R. Fergus. Regularization of neural networks using dropconnect. *ICML Conference*, pp. 1058–1066, 2013.

[512] D. Wang, P. Cui, and W. Zhu. Structural deep network embedding. *ACM KDD Conference*, pp. 1225–1234, 2016.

[513] H. Wang, N. Wang, and D. Yeung. Collaborative deep learning for recommender systems. *ACM KDD Conference*, pp. 1235–1244, 2015.

[514] L. Wang, Y. Qiao, and X. Tang. Action recognition with trajectory-pooled deep-convolutional descriptors. *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4305–4314, 2015.

[515] S. Wang, C. Aggarwal, and H. Liu. Using a random forest to inspire a neural network and improving on it. *SIAM Conference on Data Mining*, 2017.

[516] S. Wang, C. Aggarwal, and H. Liu. Randomized feature engineering as a fast and accurate alternative to kernel methods. *ACM KDD Conference*, 2017.

[517] T. Wang, D. Wu, A. Coates, and A. Ng. End-to-end text recognition with convolutional neural networks. *International Conference on Pattern Recognition*, pp. 3304–3308, 2012.

[518] X. Wang and A. Gupta. Generative image modeling using style and structure adversarial networks. *ECCV*, 2016.

[519] C. J. H. Watkins. Learning from delayed rewards. *PhD Thesis*, King's College, Cambridge, 1989.

[520] C. J. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3–4), pp. 279–292, 1992.

[521] K. Weinberger, B. Packer, and L. Saul. Nonlinear Dimensionality Reduction by Semidefinite Programming and Kernel Matrix Factorization. *AISTATS*, 2005.

[522] M. Welling, M. Rosen-Zvi, and G. Hinton. Exponential family harmoniums with an application to information retrieval. *NIPS Conference*, pp. 1481–1488, 2005.

[523] A. Wendemuth. Learning the unlearnable. *Journal of Physics A: Math. Gen.*, 28, pp. 5423–5436, 1995.

[524] P. Werbos. Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. *PhD thesis, Harvard University*, 1974.

[525] P. Werbos. The roots of backpropagation: from ordered derivatives to neural networks and political forecasting (Vol. 1). *John Wiley and Sons*, 1994.

[526] P. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10), pp. 1550–1560, 1990.

[527] J. Weston, A. Bordes, S. Chopra, A. Rush, B. van Merrienboer, A. Joulin, and T. Mikolov. Towards ai-complete question answering: A set of pre-requisite toy tasks. *arXiv:1502.05698*, 2015.
https://arxiv.org/abs/1502.05698

[528] J. Weston, S. Chopra, and A. Bordes. Memory networks. *ICLR*, 2015.

[529] J. Weston and C. Watkins. Multi-class support vector machines. *Technical Report CSD-TR-98-04*, Department of Computer Science, Royal Holloway, University of London, May, 1998.

[530] D. Wettschereck and T. Dietterich. Improving the performance of radial basis function networks by learning center locations. *NIPS Conference*, pp. 1133–1140, 1992.

[531] B. Widrow and M. Hoff. Adaptive switching circuits. *IRE WESCON Convention Record*, 4(1), pp. 96–104, 1960.

[532] S. Wieseler and H. Ney. A convergence analysis of log-linear training. *NIPS Conference*, pp. 657–665, 2011.

[533] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3–4), pp. 229–256, 1992.

[534] C. Wu, A. Ahmed, A. Beutel, A. Smola, and H. Jing. Recurrent recommender networks. *ACM International Conference on Web Search and Data Mining*, pp. 495–503, 2017.

[535] Y. Wu, C. DuBois, A. Zheng, and M. Ester. Collaborative denoising auto-encoders for top-n recommender systems. *Web Search and Data Mining*, pp. 153–162, 2016.

[536] Z. Wu. Global continuation for distance geometry problems. *SIAM Journal of Optimization*, 7, pp. 814–836, 1997.

[537] S. Xie, R. Girshick, P. Dollar, Z. Tu, and K. He. Aggregated residual transformations for deep neural networks. *arXiv:1611.05431*, 2016.
https://arxiv.org/abs/1611.05431

[538] E. Xing, R. Yan, and A. Hauptmann. Mining associated text and images with dual-wing harmoniums. *Uncertainty in Artificial Intelligence*, 2005.

[539] C. Xiong, S. Merity, and R. Socher. Dynamic memory networks for visual and textual question answering. *ICML Confererence*, pp. 2397–2406, 2016.

[540] K. Xu *et al.* Show, attend, and tell: Neural image caption generation with visual attention. *ICML Confererence*, 2015.

[541] O. Yadan, K. Adams, Y. Taigman, and M. Ranzato. Multi-gpu training of convnets. *arXiv:1312.5853*, 2013.
https://arxiv.org/abs/1312.5853

[542] Z. Yang, X. He, J. Gao, L. Deng, and A. Smola. Stacked attention networks for image question answering. *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 21–29, 2016.

[543] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9), pp. 1423–1447, 1999.

[544] F. Yu and V. Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv:1511.07122*, 2015.
https://arxiv.org/abs/1511.07122

[545] H. Yu and B. Wilamowski. Levenberg–Marquardt training. *Industrial Electronics Handbook*, 5(12), 1, 2011.

[546] L. Yu, W. Zhang, J. Wang, and Y. Yu. SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient. *AAAI Conference*, pp. 2852–2858, 2017.

[547] W. Yu, W. Cheng, C. Aggarwal, K. Zhang, H. Chen, and Wei Wang. NetWalk: A flexible deep embedding approach for anomaly Detection in dynamic networks, *ACM KDD Conference*, 2018.

[548] W. Yu, C. Zheng, W. Cheng, C. Aggarwal, D. Song, B. Zong, H. Chen, and W. Wang. Learning deep network representations with adversarially regularized autoencoders. *ACM KDD Conference*, 2018.

[549] S. Zagoruyko and N. Komodakis. Wide residual networks. *arXiv:1605.07146*, 2016. https://arxiv.org/abs/1605.07146

[550] W. Zaremba and I. Sutskever. Reinforcement learning neural turing machines. *arXiv:1505.00521*, 2015.

[551] W. Zaremba, T. Mikolov, A. Joulin, and R. Fergus. Learning simple algorithms from examples. *ICML Confererence*, pp. 421–429, 2016.

[552] W. Zaremba, I. Sutskever, and O. Vinyals. Recurrent neural network regularization. *arXiv:1409.2329*, 2014.

[553] M. Zeiler. ADADELTA: an adaptive learning rate method. *arXiv:1212.5701*, 2012. https://arxiv.org/abs/1212.5701

[554] M. Zeiler, D. Krishnan, G. Taylor, and R. Fergus. Deconvolutional networks. *Computer Vision and Pattern Recognition (CVPR)*, pp. 2528–2535, 2010.

[555] M. Zeiler, G. Taylor, and R. Fergus. Adaptive deconvolutional networks for mid and high level feature learning. *IEEE International Conference on Computer Vision (ICCV)—*, pp. 2018–2025, 2011.

[556] M. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. *European Conference on Computer Vision*, Springer, pp. 818–833, 2013.

[557] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning requires rethinking generalization. *arXiv:1611.03530*. https://arxiv.org/abs/1611.03530

[558] D. Zhang, Z.-H. Zhou, and S. Chen. Non-negative matrix factorization on kernels. *Trends in Artificial Intelligence*, pp. 404–412, 2006.

[559] L. Zhang, C. Aggarwal, and G.-J. Qi. Stock Price Prediction via Discovering Multi-Frequency Trading Patterns. *ACM KDD Conference*, 2017.

[560] S. Zhang, L. Yao, and A. Sun. Deep learning based recommender system: A survey and new perspectives. *arXiv:1707.07435*, 2017. https://arxiv.org/abs/1707.07435

[561] X. Zhang, J. Zhao, and Y. LeCun. Character-level convolutional networks for text classification. *NIPS Conference*, pp. 649–657, 2015.

[562] J. Zhao, M. Mathieu, and Y. LeCun. Energy-based generative adversarial network. *arXiv:1609.03126*, 2016. https://arxiv.org/abs/1609.03126

[563] V. Zhong, C. Xiong, and R. Socher. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *arXiv:1709.00103*, 2017. https://arxiv.org/abs/1709.00103

[564] C. Zhou and R. Paffenroth. Anomaly detection with robust deep autoencoders. *ACM KDD Conference*, pp. 665–674, 2017.

[565] M. Zhou, Z. Ding, J. Tang, and D. Yin. Micro Behaviors: A new perspective in e-commerce recommender systems. *WSDM Conference*, 2018.

[566] Z.-H. Zhou. Ensemble methods: Foundations and algorithms. *CRC Press*, 2012.

[567] Z.-H. Zhou, J. Wu, and W. Tang. Ensembling neural networks: many could be better than all. *Artificial Intelligence*, 137(1–2), pp. 239–263, 2002.

[568] C. Zitnick and P. Dollar. Edge Boxes: Locating object proposals from edges. *ECCV*, pp. 391–405, 2014.

[569] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv:1611.01578*, 2016.
https://arxiv.org/abs/1611.01578

[570] https://deeplearning4j.org/

[571] http://caffe.berkeleyvision.org/

[572] http://torch.ch/

[573] http://deeplearning.net/software/theano/

[574] https://www.tensorflow.org/

[575] https://keras.io/

[576] https://lasagne.readthedocs.io/en/latest/

[577] http://www.netflixprize.com/community/topic_1537.html

[578] http://deeplearning.net/tutorial/lstm.html

[579] https://arxiv.org/abs/1609.08144

[580] https://github.com/karpathy/char-rnn

[581] http://www.image-net.org/

[582] http://www.image-net.org/challenges/LSVRC/

[583] https://www.cs.toronto.edu/~kriz/cifar.html

[584] http://code.google.com/p/cuda-convnet/

[585] http://caffe.berkeleyvision.org/gathered/examples/feature_extraction.html

[586] https://github.com/caffe2/caffe2/wiki/Model-Zoo

[587] http://scikit-learn.org/

[588] http://clic.cimec.unitn.it/composes/toolkit/

[589] https://github.com/stanfordnlp/GloVe

[590] https://deeplearning4j.org/

[591] https://code.google.com/archive/p/word2vec/

[592] https://www.tensorflow.org/tutorials/word2vec/

[593] https://github.com/aditya-grover/node2vec

[594] https://www.wikipedia.org/

[595] https://github.com/caglar/autoencoders

[596] https://github.com/y0ast

[597] https://github.com/fastforwardlabs/vae-tf/tree/master

[598] https://science.education.nih.gov/supplements/webversions/BrainAddiction/guide/lesson2-1.html

[599] https://www.ibm.com/us-en/marketplace/deep-learning-platform

[600] https://www.coursera.org/learn/neural-networks

[601] https://archive.ics.uci.edu/ml/datasets.html

[602] http://www.bbc.com/news/technology-35785875

[603] https://deepmind.com/blog/exploring-mysteries-alphago/

[604] http://selfdrivingcars.mit.edu/

[605] http://karpathy.github.io/2016/05/31/rl/

[606] https://github.com/hughperkins/kgsgo-dataset-preprocessor

[607] https://www.wired.com/2016/03/two-moves-alphago-lee-sedol-redefined-future/

[608] https://qz.com/639952/googles-ai-won-the-game-go-by-defying-millennia-of-basic-human-instinct/

[609] http://www.mujoco.org/

[610] https://sites.google.com/site/gaepapersupp/home

[611] https://drive.google.com/file/d/0B9raQzOpizn1TkRIa241ZnBEcjQ/view

[612] https://www.youtube.com/watch?v=1L0TKZQcUtA&list=PLrAXtmErZgOeiKm4sgNOkn–GvNjby9efdf

[613] https://openai.com/

[614] http://jaberg.github.io/hyperopt/

[615] http://www.cs.ubc.ca/labs/beta/Projects/SMAC/

[616] https://github.com/JasperSnoek/spearmint

[617] https://deeplearning4j.org/lstm

[618] http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[619] https://www.youtube.com/watch?v=2pWv7GOvuf0

[620] https://gym.openai.com

[621] https://universe.openai.com

[622] https://github.com/facebookresearch/ParlAI

[623] https://github.com/openai/baselines

[624] https://github.com/carpedm20/deep-rl-tensorflow

[625] https://github.com/matthiasplappert/keras-rl

[626] http://apollo.auto/

[627] https://github.com/Element-Research/rnn/blob/master/examples/

[628] https://github.com/lmthang/nmt.matlab

[629] https://github.com/carpedm20/NTM-tensorflow

[630] https://github.com/camigord/Neural-Turing-Machine

[631] https://github.com/SigmaQuan/NTM-Keras

[632] https://github.com/snipsco/ntm-lasagne

[633] https://github.com/kaishengtai/torch-ntm

[634] https://github.com/facebook/MemNN

[635] https://github.com/carpedm20/MemN2N-tensorflow

[636] https://github.com/YerevaNN/Dynamic-memory-networks-in-Theano

[637] https://github.com/carpedm20/DCGAN-tensorflow

[638] https://github.com/carpedm20

[639] https://github.com/jacobgil/keras-dcgan

[640] https://github.com/wiseodd/generative-models

[641] https://github.com/paarthneekhara/text-to-image

[642] http://horatio.cs.nyu.edu/mit/tiny/data/

[643] https://developer.nvidia.com/cudnn

[644] http://www.nvidia.com/object/machine-learning.html

[645] https://developer.nvidia.com/deep-learning-frameworks

Index