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,

https://doi.org/10.1007/978-3-319-94463-0_10

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.

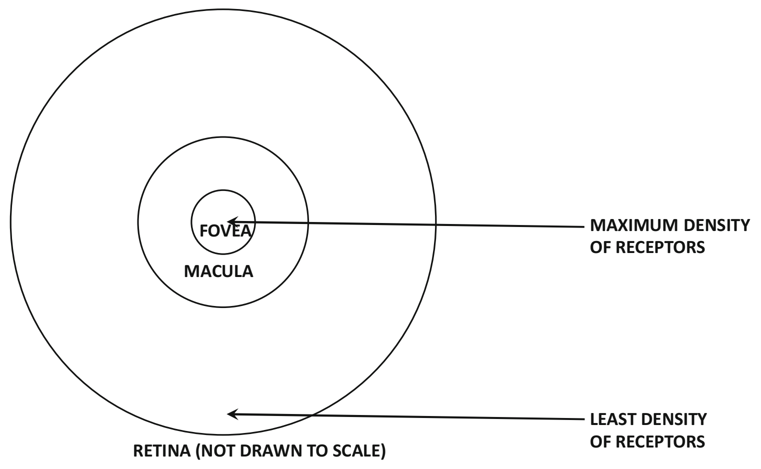

FOVEA
MACULA
MAXIMUM DENSITY OF RECEPTORS
LEAST DENSITY OF RECEPTORS
RETINA (NOT DRAWN TO SCALE)

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

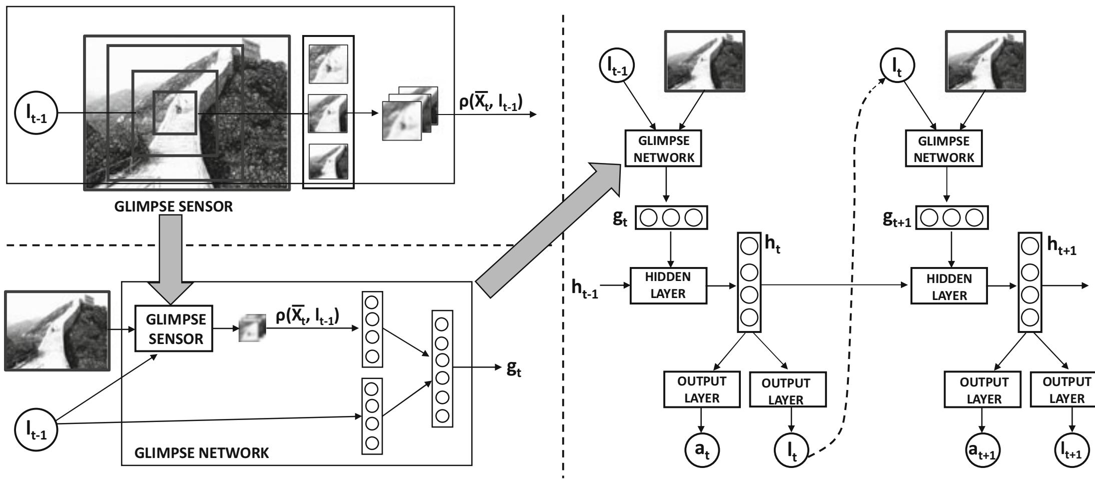


Figure 10.2: The recurrent architecture for leveraging visual attention

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 roll-outs. 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.

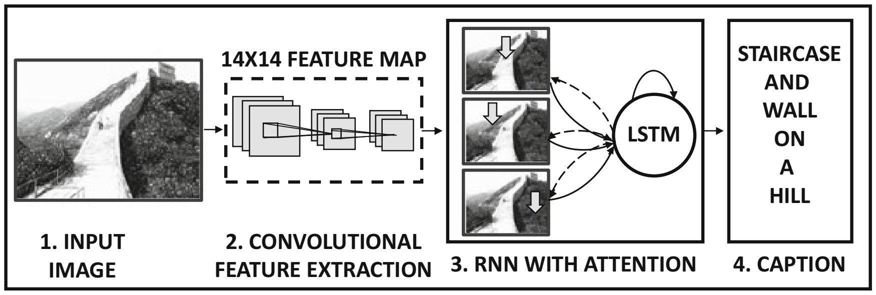


Figure 10.3: Visual attention in image captioning

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:

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

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.

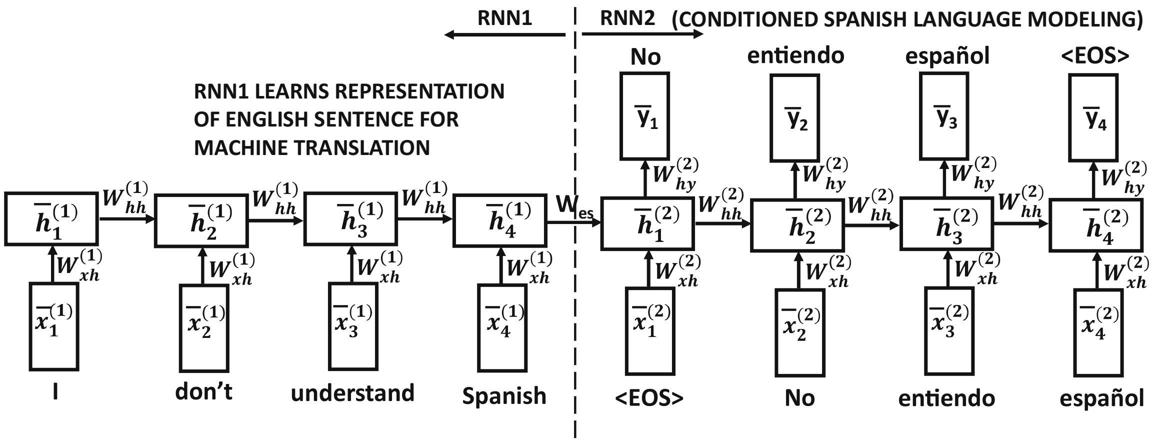
Most of these alternative models for computing the similarity use parameters to regulate the computation, which provides some additional flexibility in relating the source and target positions. The different options for computing the score are as follows:

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

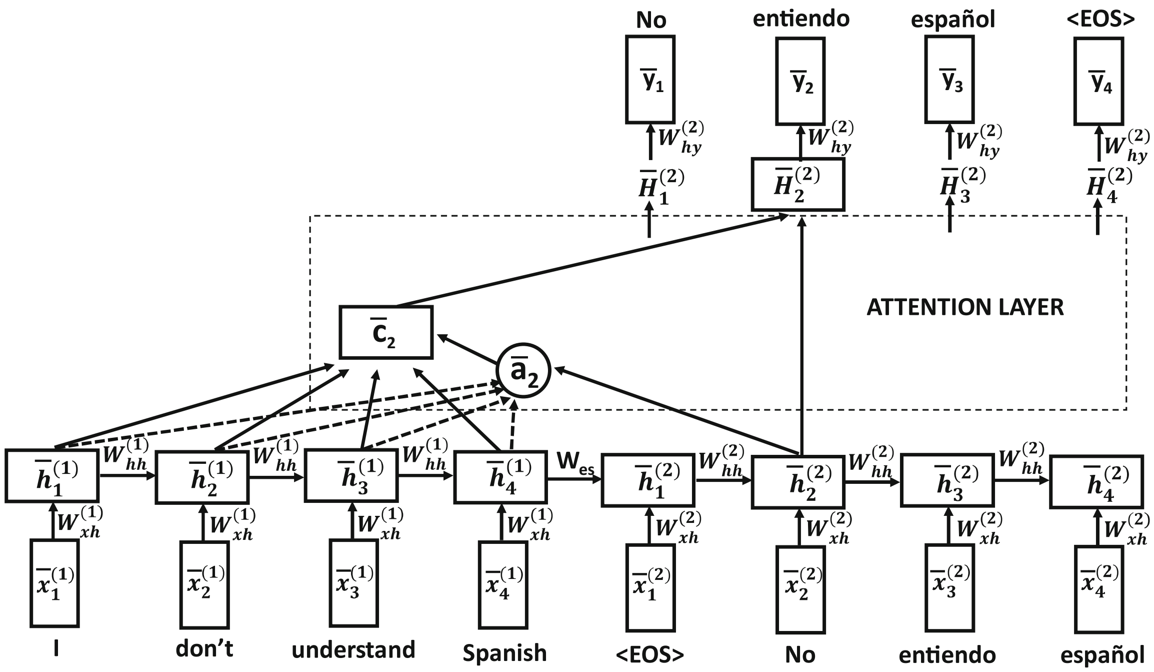
The first of these options is identical to the one discussed in the previous section according to Equation 10.3. The other two models are referred to as *general* and *concat*, respectively, as annotated above. Both these options are parameterized with the use of weight vectors, and the corresponding 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.


RNN1
RNN2 (CONDITIONED SPANISH LANGUAGE MODELING)
RNN1 LEARNS REPRESENTATION OF ENGLISH SENTENCE FOR MACHINE TRANSLATION
No
entiendo
español
<EOS>
I
don't
understand
Spanish
<EOS>
No
entiendo
español

(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.

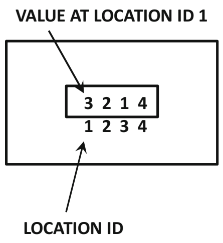
Unlike the image captioning application of the previous section, the machine translation approach is a soft attention model. The hard attention setting seems to be inherently designed for reinforcement learning, whereas the soft attention setting is differentiable, and can be used with backpropagation. 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

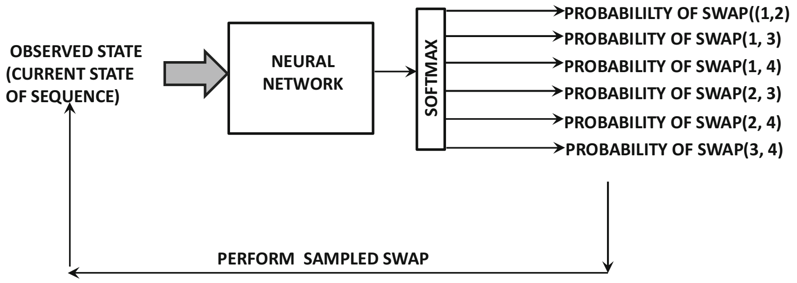
In recent years, several related architectures have been proposed that augment neural networks with *persistent memory* in which the notion of memory is clearly separated from the computations, and one can control the ways in which computations selectively access and modify particular memory locations. 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


OBSERVED STATE (CURRENT STATE OF SEQUENCE)
NEURAL NETWORK
SOFTMAX
PROBABILILTY OF SWAP((1,2)
PROBABILITY OF SWAP(1, 3)
PROBABILITY OF SWAP(1, 4)
PROBABILITY OF SWAP(2, 3)
PROBABILITY OF SWAP(2, 4)
PROBABILITY OF SWAP(3, 4)
PERFORM SAMPLED SWAP

(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 pre-defined 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.

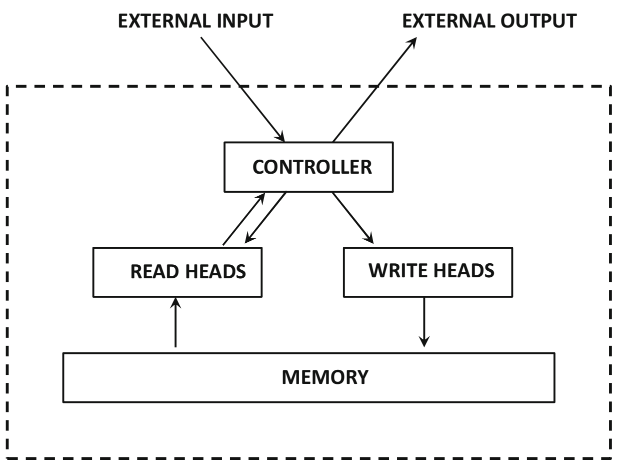


Figure 10.6: The neural Turing machine

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.

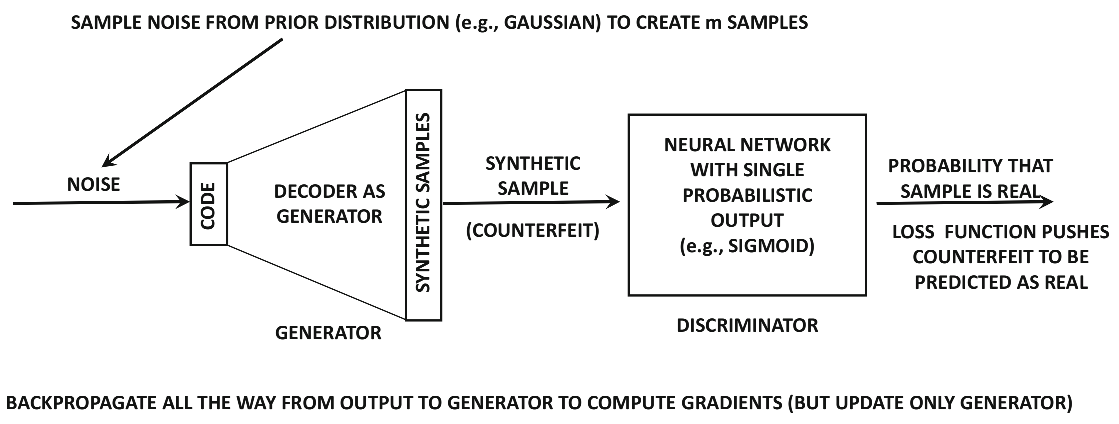

SAMPLE NOISE FROM PRIOR DISTRIBUTION (e.g., GAUSSIAN) TO CREATE m SAMPLES
NOISE
CODE
DECODER AS GENERATOR
SYNTHETIC SAMPLES
GENERATOR
SYNTHETIC SAMPLE
(COUNTERFEIT)
NEURAL NETWORK WITH SINGLE PROBABILISTIC OUTPUT (e.g., SIGMOID)
DISCRIMINATOR
PROBABILITY THAT SAMPLE IS REAL
LOSS FUNCTION PUSHES COUNTERFEIT TO BE PREDICTED AS REAL
BACKPROPAGATE ALL THE WAY FROM OUTPUT TO GENERATOR TO COMPUTE GRADIENTS (BUT UPDATE ONLY GENERATOR)

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

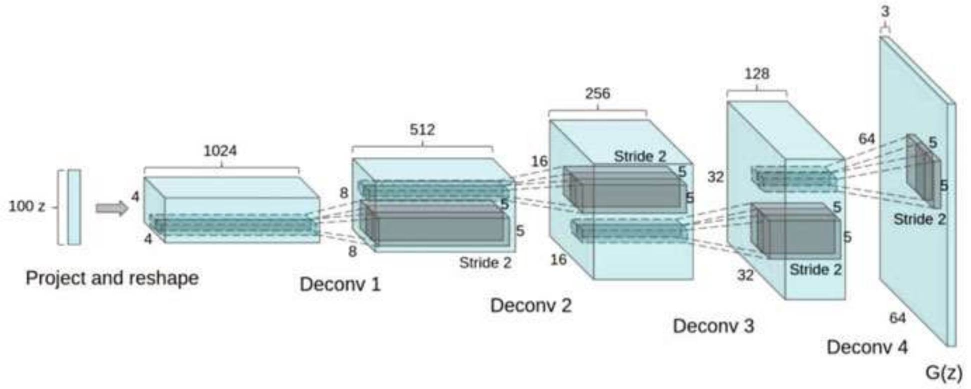
The variational autoencoder and the generative adversarial network were developed independently at around the same time. There are some interesting similarities and differences between these two models. This section will discusses a comparison of these two models.

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).

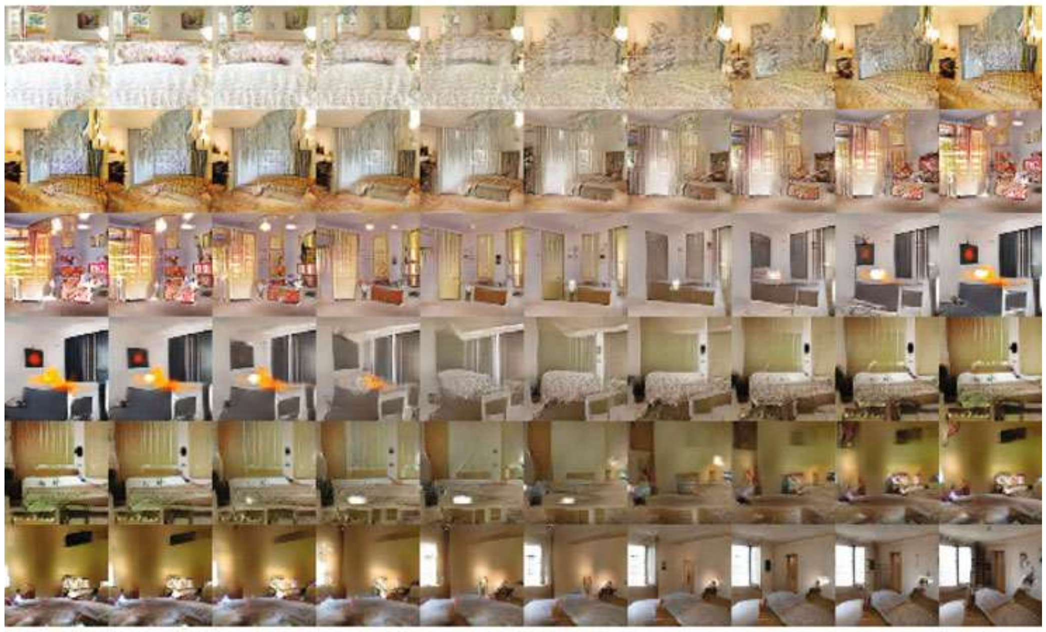
The variational autoencoder is methodologically quite different from the generative adversarial network. The re-parametrization approach used by the variational autoencoder is very useful for training networks with a stochastic nature. Such an approach has the potential to be used in other types of neural 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

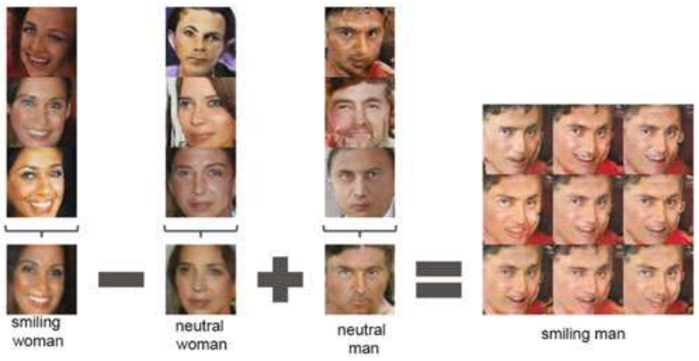
GAN is commonly used is for generating image objects with varying types of context. Indeed, the image setting is, by far, the most common use case of GANs. The generator for the image setting is referred to as a *deconvolutional network*. The most popular way to design a deconvolutional network for the 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

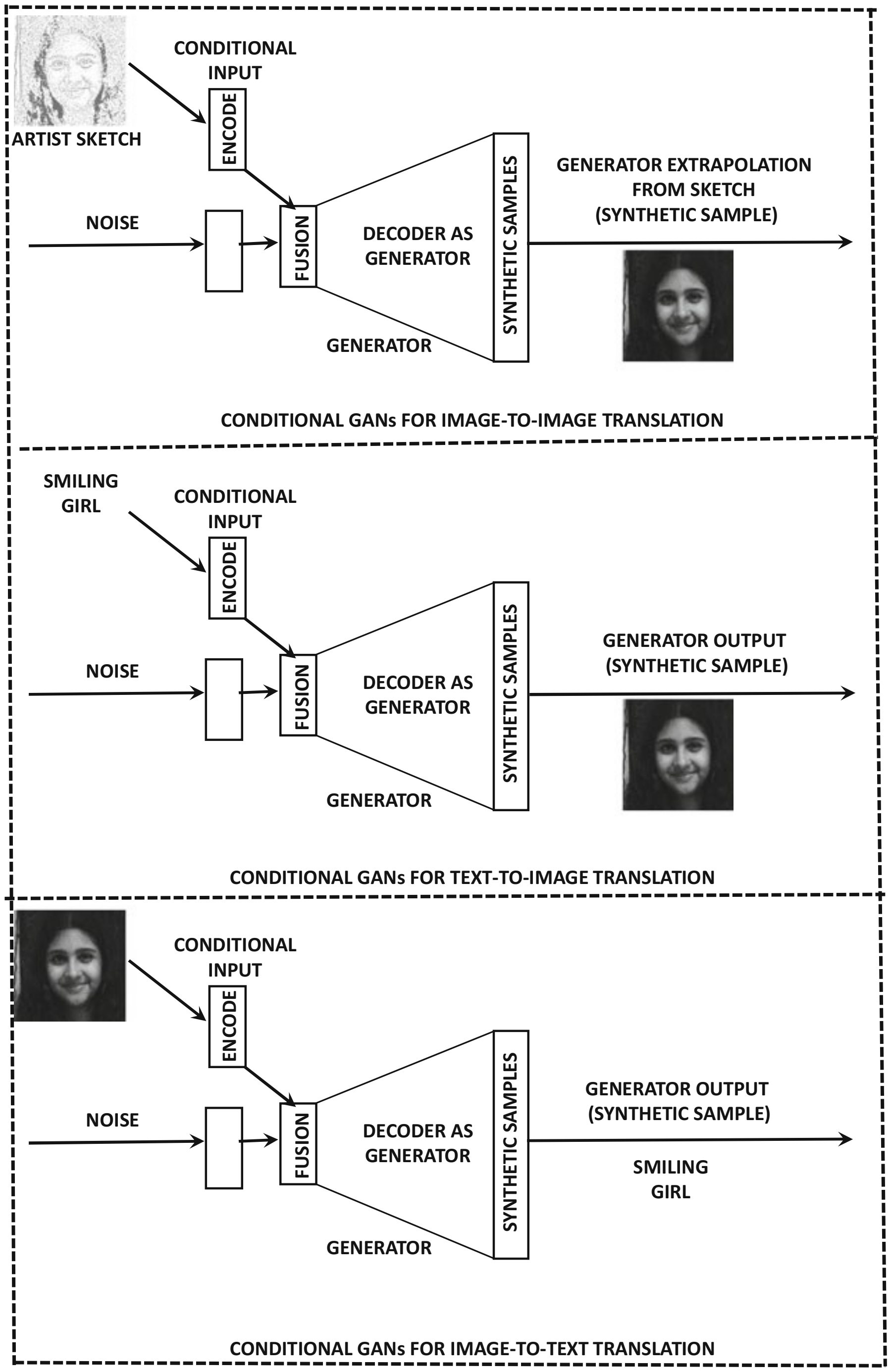


Figure 10.9: Different types of conditional generators for adversarial network. The examples are only illustrative in nature, and they do not reflect actual CGAN output.

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.

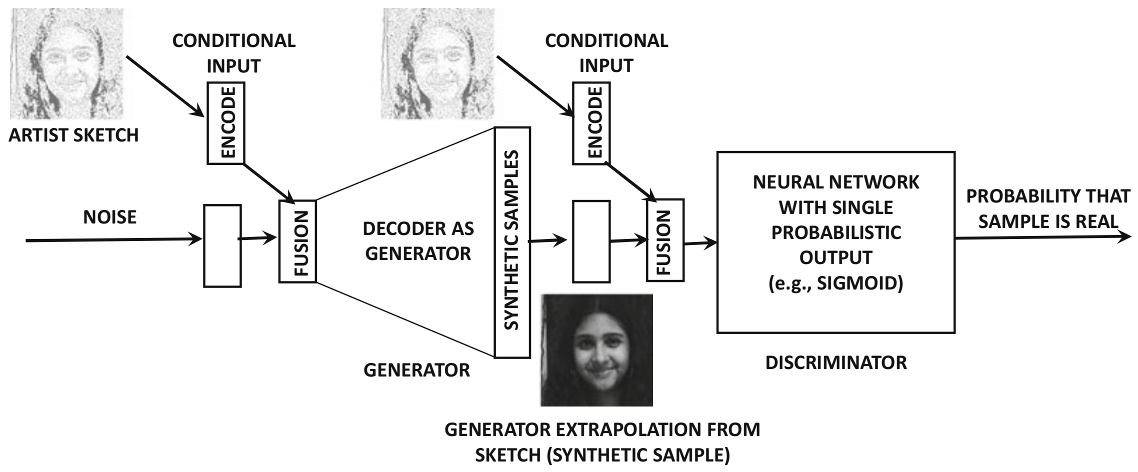

ARTIST SKETCH
CONDITIONAL INPUT
ENCODE
NOISE
FUSION
DECODER AS GENERATOR
GENERATOR
SYNTHETIC SAMPLES
CONDITIONAL INPUT
ENCODE
FUSION
NEURAL NETWORK WITH SINGLE PROBABILISTIC OUTPUT (e.g., SIGMOID)
DISCRIMINATOR
PROBABILITY THAT SAMPLE IS REAL
GENERATOR EXTRAPOLATION FROM SKETCH (SYNTHETIC SAMPLE)

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.

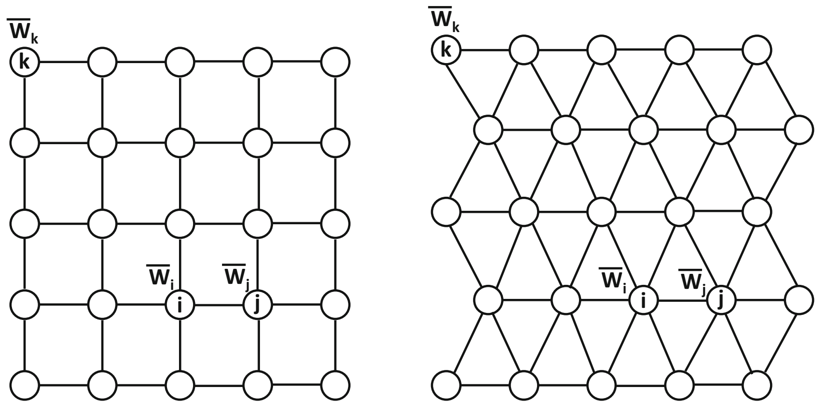
10.5.1 Vector Quantization

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 $\overline{W}_i$:

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

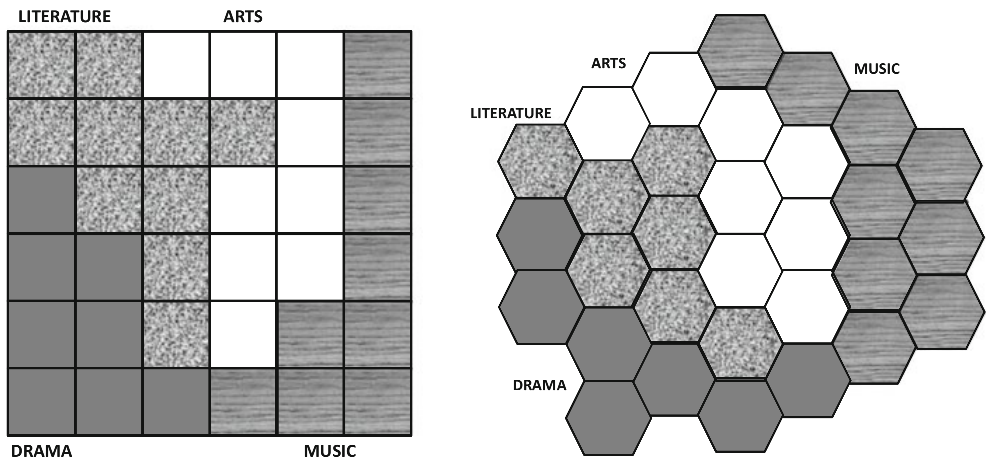
Here, $\alpha > 0$ is the learning rate. It is common to allow the learning rate α to reduce with time. These iterations are continued until convergence is reached. Note that weights that are lattice-adjacent will receive similar updates, and will therefore tend to become more similar over time. *Therefore,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