An Overview of Different Neural Network Architectures 10

10.1 Energy-Based Models

Energy-based models are a specific class of neural networks. The simplest energy model is the *Hopfield Network* dating back from the 1980s [1]. Hopfield networks are often thought to be very simple, but they are quite different from what we have seen before. The network is made of neurons, and all of these neurons are connected among them with weights w_{ij} connecting neurons n_i and n_j. Each neuron has a threshold associated with it, and we denote it by b_i. All neurons have 1 or -1 in them. If you want to process and image, you can think of -1 as white and 1 as black (no shades of grey here). We denote the inputs we place in neurons by x_i. A simple Hopfield network is shown in Fig. 10.1a.

Once a network is assembled, the training can start. The weights are updated by the following rule, where n denotes an individual training sample:

$$w_{ij} = \sum_{n=1}^{N} x_i^{(n)} x_j^{(n)} \tag{10.1}$$

Then we compute activations for each neuron:

$$y_i = \sum_j w_i j x_j \tag{10.2}$$

There are two possibilities on how to update weights. We can either do it synchronously (all weights at the same time) or asynchronously (one by one, this is the standard way). In Hopfield networks there is no recurrent connections, i.e. $w_{ii} = 0$ for all i, and all connections are symmetric, i.e. $w_{ij} = w_{ji}$. Let us see how the simple Hopfield Network shown in Fig. 10.1b processes the simple 1 by 3 pixel 'images' in Fig. 10.1c, which we represent by vectors $\mathbf{a} = (-1, 1, -1)$, $\mathbf{b} = (1, 1, -1)$ and

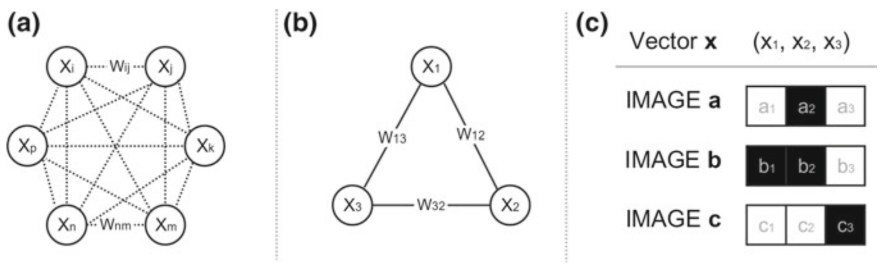

Fig. 10.1 Hopfield networks

$\mathbf{c} = (-1, -1, 1)$. Using the equation above, we calculate the weight updates with the update equation:

$$w_{11} = w_{22} = w_{33} = 0$$

$$w_{12} = a_1 a_2 + b_1 b_2 + c_1 c_2 = -1 \cdot 1 + 1 \cdot 1 + (-1) \cdot (-1) = 1$$

$$w_{13} = -1$$

$$w_{23} = -3$$

Hopfield networks have a global measure of success, similar to the error function of regular neural networks, called the *energy*. Energy is defined for each stage of network training as a single value for the whole network. It is calculated as:

$$ENE = -\sum_{i,j} w_{ij} y_i y_j + \sum_i b_i y_i \tag{10.3}$$

The as learning progresses, ENE either stays the same or diminishes, and this is how Hopfield networks reach local minima. Each local minimum is a memory of some training samples. Remember logical functions and logistic regression? We needed two input neurons and one output neurons for conjunction and disjunction, and an additional hidden one for XOR. We need three neurons in Hopfield networks for conjunction and disjunction and four for XOR.

The next model we briefly present are *Boltzmann machines* first presented in 1985 [2]. At first glance, they are very similar to Hopfield networks, but have input neurons and hidden neurons as well, which are all interconnected with weights. These weights are non-recurrent and symmetrical. A sample Boltzmann machine is displayed in Fig. 10.2a. Hidden units are initialized at random, and they build a hidden representation to mimic the inputs. These form two probability distributions, which can be compared with the Kullback-Leibler divergence $\mathbb{KL}$. The main goal then becomes clear, calculate $\frac{\partial \mathbb{KL}}{\partial w}$, and backpropagate it.

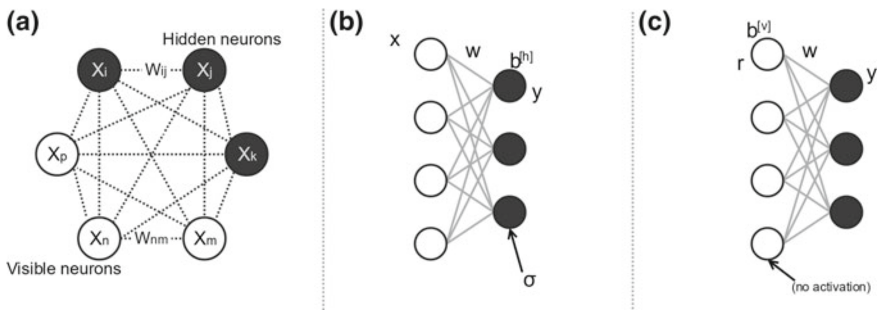

Fig. 10.2 Boltzmann machines and restricted Boltzmann machines

We turn to a subclass of Boltzmann machines, called *restricted Boltzmann machines* (RBM) [3]. Structurally speaking, restricted Boltzmann machines are just Boltzmann machines where there are no connections between neurons of the same layer (hidden to hidden and visible to visible). This seems like a minor point, but this actually makes it possible to use a modification of the backpropagation used in feed-forward networks. The restricted Boltzmann machine therefore has two layers, a visible, and a hidden. The visible layer (this is true for Boltzmann machines in general) is the place where we put in inputs and read out outputs. Denote the inputs with x_i, the biases of the hidden layer with $b_j^{[h]}$. Then, during the forward pass (see Fig. 10.2b), the RBM calculates $\mathbf{y} = \sigma(\mathbf{x}^\top \mathbf{w} + \mathbf{b}^{[h]})$. If we were to stop here, RBMs would be similar to autoencoders, but we have a second phase, the *reconstruction* (see Fig. 10.2c). During the reconstruction, the $\mathbf{y}$ are fed to the hidden layer and then passed to the visible layer. This is done by multiplying them with the same weights, and adding another set of biases, i.e. $\mathbf{r} = \mathbf{y}^\top \mathbf{w} + \mathbf{b}^{[v]}$. The difference between $\mathbf{x}$ and $\mathbf{r}$ is measured with $\mathbb{KL}$ and then this error is used in backpropagation. RBMs are fragile, and every time one gets a nonzero reconstruction, this is a good sign. Boltzmann machines are similar to logical *constraint satisfaction solvers*, but they focus on what Hinton and Sejnowski called 'weak constraints'. Notice that we moved away quite a bit from the energy function, and well back into standard neural network territory.

The final architecture we will briefly discuss is *deep belief networks* (DBN), which are just stacked RBMs. They were introduced in [4] and in [5]. They are conceptually similar to stacked autoencoders, but they can be trained with backpropagation to be generative models, or with *contrastive divergence*. In this setting, they may be even used as classifiers. Contrastive divergence is simply an algorithm that efficiently approximates the gradients of the log-likelihood. A discussion on contrastive divergence is beyond the scope of this book, but we point the interested reader to [6] and [7]. For a discussion about the cognitive aspects of energy-based models, see [8].

10.2 Memory-Based Models

The first memory-based model we will explore are *neural Turing-machines* (NTM) first proposed in [9]. Remember how a Turing-machine works: you have a read-write head and a tape which acts as a memory. The Turing-machine then is given a function in the form of an algorithm and it computes that function (takes in the given inputs and outputs the result). The neural Turing-machine is similar, but the point is to have all components trainable, so that they can do soft computation, and they should also learn how to do it well.

The neural Turing-machine acts similarly to an LSTM. It takes input sequences and outputs sequences. If we want it to output a single result, we just take the last component and discard everything else. The neural Turing-machine is built upon an LSTM, and can be seen as an architecture extending the LSTM similarly how LSTMs builds upon simple recurrent networks.

A neural Turing-machine has several components. The first one is called a *controller*, and a controller is simply an LSTM. Similar to an LSTM, the neural Turing-machine has a temporal component, and all elements are indexed by t, and the state of the machine at time t takes as inputs components calculated at $t - 1$. The controller takes in two inputs: (i) raw inputs at time t, i.e. $\mathbf{x}_t$ and (ii) results of the previous step, $\mathbf{r}_t$. The neural Turing-machine has another major component, the memory, which is just a tensor denoted by M_t (it is usually just a matrix). Memory is not an input to the controller, but it is an input to the step t of the whole neural Turing-machine (the input is M_{t-1}).

The structure of a complete neural Turing-machine is shown in Fig. 10.3, but we have omitted the details.[1] The idea is that the whole neural Turing-machine should be expressed as tensors, and trainable by gradient descent. To enable this, all crisp concepts from regular Turing-machines are fuzzified, so that there is no single memory location which is accessed in separation, but all memory locations are accessed to a certain degree. But in addition to the fuzzy part, the amount of the accessed memory is also trainable, so it changes dynamically.

To reiterate: the neural Turing-machine has an LSTM (controller) which receives the outputs from the previous step, and a fresh vector of inputs, and uses them and a memory matrix to produce outputs and everything is trainable. But how do the components work? Let us now work our way from the memory upward. We will be needing three vectors, all of which the controller will produce: add vector $\mathbf{a_t}$, erase vector $\mathbf{e_t}$, and weighting vector $\mathbf{w_t}$. They are similar but used for different purposes. We will be coming back to them later to explain how they are produced.

Let us see how the memory works. The memory is represented by a matrix (or possibly higher order tensor) M_t. Each row in this matrix is called a *memory location*. If there are n rows in the memory, the controller produces a weighting vector of size n

[1] For a fully detailed view, see the blog entry of one of the creators of the NTM, `https://medium.com/aidangomez/the-neural-turing-machine-79f6e806\penalty-\@Mc0a1`.

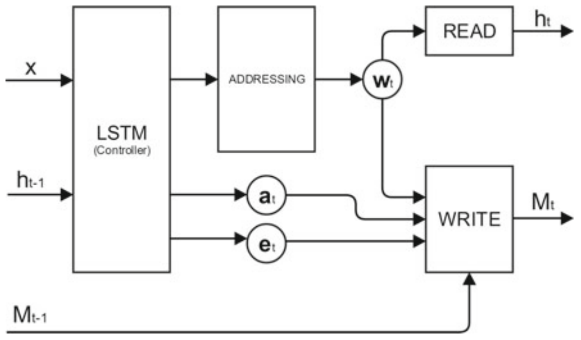

Fig. 10.3 Neural Turing-machines

(components range from 0 to 1) which indicates how much of each of those locations to take in consideration. This can be a crisp access to a one or several locations or a fuzzy access to those locations. Since this vector is trainable, it is almost never crisp. This is the reading operation, defined simply as the Hadamard product (pointwise multiplication) of m by n matrix M_t and B, where B is obtained by transposing the m-dimensional row vector $\mathbf{w_t}$, and then broadcasting its values (just copying this column $n - 1$ times) to match the dimensions of M_t.

The neural Turing-machine will now write. It *always* reads and writes, but sometimes it writes very similar values, so we have the impression that the content is not changed. This is important since it is a common source of confusion thinking that the NTM makes a *decision* whether to (over)write or not. It does not make this decision (it does not have a separate decision mechanism), it always performs the writing, but sometimes the value written is the same as the old value.

The write operation itself is composed by two components: (i) the erase component, and (ii) add component. The *erase* operation resets the components of a memory location to zero only if both the weighting vector $\mathbf{w}_t$ component for that location and the erase vector $\mathbf{e}_t$ component are both 1. In symbols: $\hat{M}_t = M_{t-1} \cdot (\mathbf{I} - \mathbf{w}_t \cdot \mathbf{e}_t)$, where $\mathbf{I}$ is a row vector of 1s, and all products are Hadamard or pointwise, so these multiplications are commutative. To take care of the dimensions, transpose and broadcast as needed. The *add* operation performs exactly the same taking in $\hat{M}_t$ instead of M_{t-1}, but by using the equation: $M_t = \hat{M}_t + \mathbf{w}_t \cdot \mathbf{a}_t)$. Remember, the way these things work is the same, they are all operations on trainable components–there is no intrinsic difference, only operations and trainable differences. We now have to connect the two parts, and this is done by addressing. Addressing is the part which describes how the weighting vectors $\mathbf{w}_t$ are produced. It is a relatively complex procedure involving a number of components, and we refer the reader to the original paper [9] for details. What is important to note is that neural Turing-machines have location-based addressing and content-based addressing.

A second memory-based model, much simpler and equally powerful is the *memory networks* (MemNN) introduced in [10]. The idea is to extend LSTM to make the long term dependency memory better. Memory networks have several components, and aside from the memory, all of them are neural networks, which makes memory networks even more aligned with the spirit of connectionism than neural Turing-machines, while retaining all the power. The components of the memory network are:

- Memory (M): An array of vectors
- Input feature map (**I**): converts the input into a distributed representation
- Updater (**G**): decides how to update the memory given the distributed representation passed in by **I**
- Output feature map (**O**): receives the input distributed representation and finds supportive vectors from memory, and produces an output vector
- Responder (**R**): Additionally formats the output vectors given by **O**

Their connections are illustrated in Fig. 10.4. All of these components except memory are functions described by neural networks and hence trainable. In a simple version, **I** would be word2vec, **G** would simply store the representation in the next available memory slot, **R** would modify the output by replacing indexes with words and adding some filler words. **O** is the one that does the hard work. It would have to find a number of supporting memories (a single memory scan and update is called a *hop*[2]), and then find a way of 'bundling' them with what **I** has forwarded. This 'bundling' is simple matrix multiplication, of the input and the memory, but with also some additional learned weights. This is how it always should be in connectionists models: just adding, multiplying and weights. And the weights are where the magic happens. A fully trainable complex memory network is presented in [11].

One problem that both neural Turing-machines and memory networks have in common is that they have to use segmented vector-based memory. It would be interesting to see how to make a memory-based model with a continuous memory, perhaps with encoding vectors in floats. But a word of warning, even plain-vanilla memory networks have a lot more trainable parameters than LSTMs, and training could take a lot of time, so one of the major challenges in memory models mentioned in [11] is how to reuse parameters in various components, which would speed up learning. Memory networks memory addressing is only content-based.

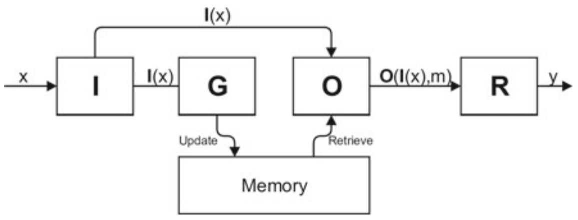

Fig. 10.4 Memory networks

[2]By default, memory networks make one hop, but it has been shown that multiple hops are beneficial, especially in natural language processing.

10.3 The Kernel of General Connectionist Intelligence: The bAbI Dataset

Despite their colourful past, neural networks today are a recognized subfield of AI, and deep learning is making a run for the whole AI. A natural question arises, how can we evaluate neural networks as an AI system, and it seems that the old idea of the Turing test is coming back. Fortunately, there is a dataset of toy tasks called bAbI [12], which was made with the idea of it becoming a kernel for general AI: Any agent hoping to be recognized as general AI should be able to pass all the toy tasks in the bAbI dataset. The bAbI dataset is one of the most important general AI tasks to be confronted with a purely connectionistic approach.

The tasks in the dataset are expressed in natural language, and there are twenty categories of them. The first category addresses single supporting fact, and it has samples that try to capture a simple repetition of what was already stated like the example produced 'Mary went to the bathroom. John moved to the hallway. Mary travelled to the office. Where is Mary?. The next two tasks introduce more supporting facts, i.e. more actions by the same person. The next task focuses on learning and resolving relations, like being given 'the kitchen is north of the bathroom. What is north of the bathroom?. A similar but considerably more complex task is Task 19 (Path finding): 'the kitchen is north of the bathroom. How to get from the kitchen to the bathroom?'. It is the 'flipping' that adds to the complexity. Also, here the task is to produce directions (with multiple steps), where in the relation resolution the network just had to produce the resolvent.

The next task addresses binary answer questions in natural language. Another interesting task is called 'counting', and the information given contains a single agent picking up and dropping stuff. The network has to count how many items he has in his hands at the end of the sequence. The next three tasks are based on negation, conjunction and using three-valued answering ('yes', 'no', 'maybe'). The tasks which address coreference resolution follow. Then come the tasks for time reasoning, positional reasoning and size reasoning (resembling Winograd sentences[3]), and tasks dealing with basic syllogistic deduction and induction. The last task is to resolve the agent's motivation.

The authors of the dataset tested a number of methods against the data, but the results for plain (non-tweaked) memory networks[10] are the most interesting, since they represent what a pure connectionist approach can achieve. We reproduce the list of accuracies for plain memory networks [12], and refer the reader to the original paper for other results.

[3]Winograd sentences are sentences of a particular form, whare the computer should resolve the coreference of a pronoun. They were proposed as an alternative to the Turing test, since the turing test has some deep flaws (deceptive behaviour is encouraged), and it is hard to quantify its results and evaluate it on a large scale. Winograd sentences are sentances of the form 'I tried to put the book in the drwer but it was too [big/small]', and they are named after Terry Winograd who first considered them in the 1970s [13].

1. Single supporting fact: 100%
2. Two supporting facts: 100%
3. Three supporting facts: 20%
4. Two argument relations: 71%
5. Three argument relations: 83%
6. Yes-no questions: 47%
7. Counting: 68%
8. Lists: 77%
9. Simple negation: 65%
10. Indefinite knowledge: 59%
11. Basic coreference: 100%
12. Conjunction: 100%
13. Compound coreference: 100%
14. Time reasoning: 99%
15. Basic deduction: 74%
16. Basic induction: 27%
17. Positional reasoning: 54%
18. Size reasoning: 57%
19. Path Finding: 0%
20. Agent's motivations: 100%

These results point at a couple of things. First, it is amazing how well memory networks address coreference resolution. It is also remarkable how well the memory network performs on pure deduction. But the most interesting part is how the problems arise from inference-heavy tasks where deduction has to be applied to obtain the result (as opposed to basic deduction, where the emphasis is on form). The most representative of these tasks are path finding and size reasoning. We find it interesting since memory networks have a memory component, but not a component for reasoning, and it would seem that memory is more helpful in form-based reasoning such as deduction. It is also interesting that the *tweaked* memory network jumped to 100% on induction but dropped to 73% on deduction. The question on how to get a neural network to reason seems to be of paramount importance in getting past these benchmarks made by memory networks.

References

1. J.J. Hopfield, Neural networks and physical systems with emergent collective computational abilities. Proc. Nat. Acad. Sci. U.S.A **79**(8), 2554–2558 (1982)
2. D.H. Ackley, G.E. Hinton, T. Sejnowski, A learning algorithm for boltzmann machines. Cogn. Sci. **9**(1), 147–169 (1985)
3. P. Smolensky, Information processing in dynamical systems: foundations of harmony theory, in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, ed. by D.E. Rumelhart, J.L. McClelland, the PDP Research Group, (MIT Press, Cambridge)

4. G.E. Hinton, S. Osindero, Y.-W. Teh, A fast learning algorithm for deep belief nets. Neural Comput. **18**(7), 1527–1554 (2006)
5. Y. Bengio, P. Lamblin, D. Popovici, H. Larochelle, Greedy layer-wise training of deep networks, in *Proceedings of the 19th International Conference on Neural Information Processing Systems* (MIT Press, Cambridge, 2006), pp. 153–160
6. Y. Bengio, Learning deep architectures for AI. Found. Trends Mach. Learn. **2**(1), 1–127 (2009)
7. I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning* (MIT Press, Cambridge, 2016)
8. W. Bechtel, A. Abrahamsen, *Connectionism and the Mind: Parallel Processing, Dynamics and Evolution in Networks* (Blackwell, Oxford, 2002)
9. A. Graves, G. Wayne, I. Danihelka, Neural turing machines (2014), arXiv:1410.5401
10. J. Weston, S. Chopra, A. Bordes, Memory networks, in *ICLR* (2015), arXiv:1410.3916
11. S. Sukhbaatar, A. Szlam, J. Weston, End-to-end memory networks (2015), arXiv:1503.08895
12. J. Weston, A. Bordes, S. Chopra, A.M. Rush, B. van Merriënboer, A. Joulin, T. Mikolov, Towards ai-complete question answering: A set of prerequisite toy tasks, in *ICLR* (2016), arXiv:1502.05698
13. T. Winograd, *Understanding Natural Language* (Academic Press, New York, 1972)