Chapter 4

Neural Language Models

Neural networks are a very powerful method to model conditional probability distributions with multiple inputs $p(a|b,c,d)$. They are robust to unseen data points — say, an unobserved (a,b,c,d) in the training data. Using traditional statistical estimation methods, we may address such a sparse data problem with back-off and clustering, which require insight into the problem (what part of the conditioning context to drop first?) and arbitrary choices (how many clusters?).

N-gram language models which reduce the probability of a sentence to the product of word probabilities in the context of a few previous words — say, $p(w_i|w_{i-4},w_{i-3},w_{i-2},w_{i-1})$. Such models are a prime example for a conditional probability distribution with a rich conditioning context for which we often lack data points and would like to cluster information. In statistical language models, complex discounting and back-off schemes are used to balance rich evidence from lower order models — say, the bigram model $p(w_i|w_{i-1})$ — with the sparse estimates from high order models. Now, we turn to neural networks for help.

4.1 Feed-Forward Neural Language Models

Figure 4.1 gives a basic sketch of a 5-gram neural network language model. Network nodes representing the context words have connections to a hidden layer, which connects to the output layer for the predicted word.

4.1.1 Representing Words

We are immediately faced with a difficult question: How do we represent words? Nodes in a neural network carry real-numbered values, but words are discrete items out of a very large vocabulary. We cannot simply use token IDs, since the neural network will assume that token 124,321 is very similar to token 124,322 — while in practice these numbers are completely arbitrary. The same arguments applies to the idea of using bit encoding for token IDs. The words $(1,1,1,1,0,0,0,0)^T$ and $(1,1,1,1,0,0,0,1)^T$ have very similar encodings but may have nothing

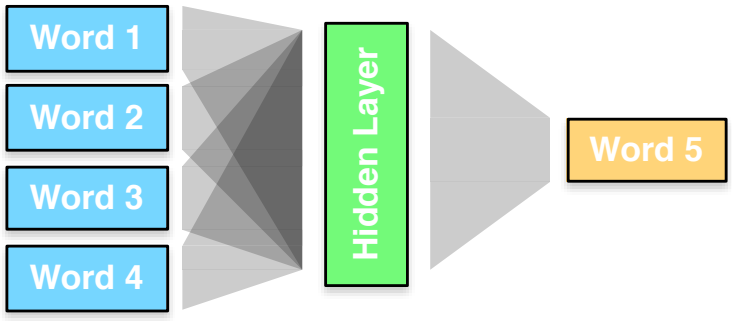

Figure 4.1: Sketch of a neural language model: We predict a word w_i based on its preceding words.

to do with each other. While the idea of using such bit vectors is occasionally explored, it does not appear to have any benefits over what we consider next.

Instead, we will represent each word with a high-dimensional vector, one dimension per word in the vocabulary, and the value 1 for the dimension that matches the word, and 0 for the rest. The type of vectors are called **one hot vector**. For instance:

- *dog* = $(0, 0, 0, 0, 1, 0, 0, 0, 0, ...)^T$
- *cat* = $(0, 0, 0, 0, 0, 0, 0, 1, 0, ...)^T$
- *eat* = $(0, 1, 0, 0, 0, 0, 0, 0, 0, ...)^T$

These are very large vectors, and we will continue to wrestle with the impact of this choice to represent words. One stopgap is to limit the vocabulary to the most frequent, say, 20,000 words, and pool all the other words in an OTHER token. We could also use word classes (either automatic clusters or linguistically motivated classes such as part-of-speech tags) to reduce the dimensionality of the vectors. We will revisit the problem of large vocabularies later.

To pool evidence between words, we introduce another layer between the input layer and the hidden layer. In this layer, each context word is individually projected into a lower dimensional space. We use the same weight matrix for each of the context words, thus generating a continuous space representation for each word, independent of its position in the conditioning context. This representation is commonly referred to as **word embedding**.

Words that occur in similar contexts should have similar word embeddings. For instance, if the training data for a language model frequently contains the n-grams

- *but the cute dog jumped*
- *but the cute cat jumped*
- *child hugged the cat tightly*
- *child hugged the dog tightly*
- *like to watch cat videos*
- *like to watch dog videos*

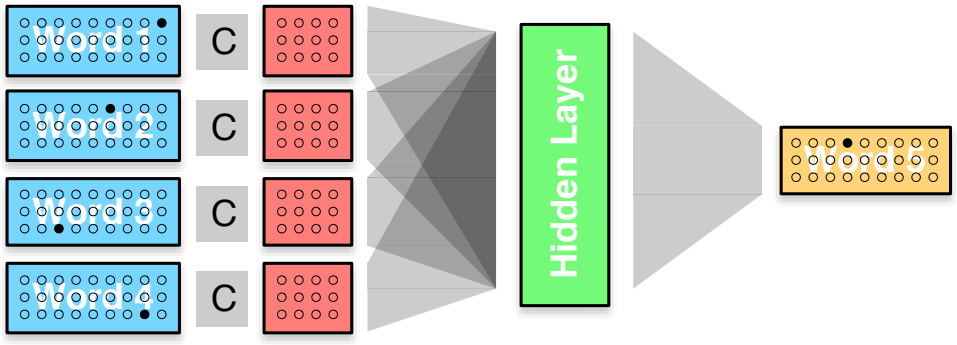

Figure 4.2: Full architecture of a feed-forward neural network language model. Context words $(w_{i-4}, w_{i-3}, w_{i-2}, w_{i-1})$ are represented in a one-hot vector, then projected into continuous space as word embeddings (using the same weight matrix C for all words). The predicted word is computed as a one-hot vector via a hidden layer.

then the language model would benefit from the knowledge that *dog* and *cat* occur in similar contexts and hence are somewhat interchangeable. If we like to predict from a context where *dog* occurs but we have seen this context only with the word *cat*, then we would still like to treat this as positive evidence. Word embeddings enable generalizing between words (clustering) and hence having robust predictions in unseen contexts (back-off).

4.1.2 Neural Network Architecture

See Figure 4.2 for a visualization of the architecture the fully fledged feed forward neural network language model, consisting of the context words as one-hot-vector input layer, the word embedding layer, the hidden layer and predicted output word layer.

The context words are first encoded as one-hot vectors. These are then passed through the embedding matrix C, resulting in a vector of floating point numbers, the word embedding. This embedding vector has typically in the order of 500 or 1000 nodes. Note that we use the same embedding matrix C for all the context words.

Also note that mathematically there is not all that much going on here. Since the input to the multiplication to the matrix C is a one hot vector, most of the input values to the matrix multiplication are zeros. So, practically, we are selecting the one column in the matrix that corresponds to the input word ID. Hence, there is no use for an activation function here. In a way, the embedding matrix a lookup table $C(w_j)$ for word embeddings, indexed by the word ID w_j.

$$C(w_j) = C\ w_j \tag{4.1}$$

Mapping to the hidden layer in the model requires concatenation of all context word embeddings $C(w_j)$ as input to a typical feed-forward layer, say, using tanh as activation function.

$$h = \tanh\Big(b_h + \sum_j H_j C(w_j)\Big) \tag{4.2}$$

The output layer is interpreted as a probability distribution over words. As before, first the linear combination s_i of weights w_{ij} and hidden node values h_j is computed for each node i.

$$s = W\ h \tag{4.3}$$

To ensure that it is indeed a proper probability distribution, we use the **softmax** activation function to ensure that all values add up to one.

$$p_i = \text{softmax}(s_i, \vec{s}) = \frac{e^{s_i}}{\sum_j e^{s_j}} \tag{4.4}$$

What we described here is close to the neural probabilistic language model proposed by Bengio et al. (2003). This model had one more twist, it added direct connections of the context word embeddings to the output word. So, Equation 4.3 is replaced by

$$s = W\ h + \sum_j U\ C(w_j) \tag{4.5}$$

Their paper reports that having such **direct connections** from context words to output words speeds up training, although does not ultimately improve performance. We will encounter the idea of short-cutting hidden layers again a bit later when we discuss deeper models with more hidden layers. They are also called **residual connections**, **skip connections**, or even **highway connections**.

4.1.3 Training

We train the parameters of a neural language model (word embedding matrix, weight matrices, bias vectors) by processing all the n-grams in the training corpus. For each n-gram, we feed the context words into the network and match the network's output against the one-hot vector of the correct word to be predicted. Weights are updated using back-propagation (we will go into details in the next section).

Language models are commonly evaluated by perplexity, which is related to the probability given to proper English text. A language model that likes proper English is a good language model. Hence, the training objective for language models is to increase the likelihood of the training data.

During training, given a context $\mathbf{x} = (w_{n-4}, w_{n-3}, w_{n-2}, w_{n-1})$, we have the correct value for the 1-hot vector $\vec{y}$. For each training example $(\mathbf{x}, \vec{y})$, likelihood is defined as

$$L(\mathbf{x}, \vec{y}; W) = -\sum_k y_k \log p_k \tag{4.6}$$

Note that only one value y_k is 1, the others are 0. So this really comes down to the probability p_k given to the correct word k. Defining likelihood this way allows us to update all weights, also the one that lead to the wrong output words.

4.2 Word Embedding

Before we move on, it is worth reflecting the role of word embeddings in neural machine translation and many other natural language processing tasks. We introduced them here as compact encoding of words in relatively high-dimensional space, say 500 or 1000 floating point numbers. In the field of natural language processing, at the time of this writing, word embeddings have acquired the reputation of almost magical quality.

Consider the role they play in the neural language language that we just described. They represent context words to enable prediction the next word in a sequence.

Recall part of our earlier example:

- *but the cute dog jumped*
- *but the cute cat jumped*

Since *dog* and *cat* occur in similar contexts, their influence on predicting the word *jumped* should be similar. It should be different from words such as *dress* which is unlikely to trigger the completion *jumped*. The idea that words that occur in similar contexts are semantically similar is a powerful idea in lexical semantics.

At this point in the argument, researchers love to cite John Rupert Firth:

> *You shall know a word by the company it keeps.*

Or, as Ludwig Wittgenstein put it a bit more broadly:

> *The meaning of a word is its use.*

Meaning and semantics are quite difficult concepts with largely unresolved definition. The idea of **distributional lexical semantics** is to define word meaning by their distributional properties, i.e., in which contexts they occur. Words that occur in similar contexts (*dog* and *cat*) should have similar representations. In vector space models, such as the word embeddings that we use here, similarity can be measured by a distance function, e.g., the cosine distance — the angle between the vectors.

If we project the high-dimensional word embeddings down to two dimensions, we can visualize word embeddings as shown in Figure 4.3. In this figure, words that are similar (*drama, theater, festival*) are clustered together.

But why stop there? We would like to have semantic representations so we can carry out semantic inference such as

- *queen = king + (woman – man)*
- *queens = queen + (kings – king)*

Indeed there is some evidence that word embedding allow just that (Mikolov et al., 2013). However, we better stop here and just note that word embeddings are a crucial tool in neural machine translation.

Figure 4.3: Word embeddings projected into 2D. Semantically similar words occur close to each other.

4.3 Efficient Inference and Training

Training a neural language model is computationally expensive. For billion word corpora, even with the use of GPUs, training takes several days with modern compute clusters. Even using a neural language model as a scoring component in statistical machine translation decoding requires a lot of computation. We could restrict its use only to re-ranking n-best lists or lattices, or consider more efficient methods for inference and training.

4.3.1 Caching for Inference

However, with a few considerations, it is actually possible to use this neural language model within the decoder.

- Word embeddings are fixed for the words, so do not actually need to carry out the mapping from one-hot vectors to word embeddings, but just store them beforehand.

- The computation between embeddings and the hidden layer can be also partly carried out offline. Note that each word can occur in one of the 4 slots for conditioning context (assuming a 5-gram language model). For each of the slots, we can pre-compute the matrix multiplication of word embedding vector and the corresponding submatrix of weights. So, at run time, we only have to sum up these pre-computations at the hidden layer and apply the activation function.

- Computing the value for each output node is insanely expensive, since there are as many output nodes as vocabulary items. However, we are interested only in the score for a given word that was produced by the translation model. If we only compute its node value, we have a score that we can use.

The last point requires a longer discussion. If we compute the node value only for the word that we want to score with the language model, we are missing an important step. To obtain a proper probability, we need to normalize it, which requires the computation of the values for all the other nodes.

We could simply ignore this problem and use the scores at face value. More likely words given a context will get higher scores than less likely words, and that is the main objective. But since we place no constraints on the scores, we may work with models where some contexts give high scores to many words, while some contexts do not give preference for any.

It would be great, if the node values in the final layer were already normalized probabilities. There are methods to enforce this during training. Let us first discuss training in detail, and then move to these methods in Section 4.3.

4.3.2 Noise Contrastive Estimation

We discussed earlier the problem that computing probabilities with a neural language model is very expensive due to the need to normalize the output node values y_i using the softmax function. This requires computing values for all output nodes, even if we are only interested in the score for a particular n-gram. To overcome the need for this explicit normalization step, we would like to train a model that already has y_i values that are normalized.

One way is to include the constraint that the normalization factor $Z(x) = \sum_j e^{s_j}$ is close to 1 in the objective function. So, instead of the just the simple likelihood objective, we may include the L2 norm of the log of this factor. Note that if $\log Z(x) \simeq 0$, then $Z(x) \simeq 1$.

$$L(\mathbf{x}, \vec{y}; W) = -\sum_k y_k \log p_k - \alpha \log^2 Z(x) \tag{4.7}$$

Another way to train a self-normalizing model is called noise contrastive estimation. The main idea is to optimize the model so that it can separate correct training examples from artificially created noise examples. This method needs less computation during training, since it does not require the computation of all output node values.

Formally, we are trying to learn the model distribution $p_m(\vec{y}|\mathbf{x}; W)$. Given a noise distribution $p_n(\vec{y}|\mathbf{x})$ — in our case of language modeling a unigram model $p_n(\vec{y})$ is a good choice — we first generate a set of noise examples U_n in addition to the correct training examples U_t. If both sets have the same size $|U_n| = |U_t|$, then the probability that a given example $(\mathbf{x}; \vec{y}) \in U_n \cup U_t$ is predicted to be a correct training example is

$$p(\text{correct}|\mathbf{x}, \vec{y}) = \frac{p_m(\vec{y}|\mathbf{x}; W)}{p_m(\vec{y}|\mathbf{x}; W) + p_n(\vec{y}|\mathbf{x})} \tag{4.8}$$

The objective of noise contrastive estimation is to maximize $p(\text{correct}|\mathbf{x}, \vec{y})$ for correct training examples $(\mathbf{x}; \vec{y}) \in U_t$ and to minimize it for noise examples $(\mathbf{x}; \vec{y}) \in U_n$. Using log-likelihood, we define the objective function as

$$L = \frac{1}{2|U_t|} \sum_{(\mathbf{x};\vec{y}) \in U_t} \log p(\text{correct}|\mathbf{x}, \vec{y}) \quad + \frac{1}{2|U_n|} \sum_{(\mathbf{x};\vec{y}) \in U_n} \log \left(1 - p(\text{correct}|\mathbf{x}, \vec{y})\right) \tag{4.9}$$

Returning the the original goal of a self-normalizing model, first note that the noise distribution $p_n(\vec{y}|\mathbf{x})$ is normalized. Hence, the model distribution is encouraged to produce comparable values. If $p_m(\vec{y}|\mathbf{x}; W)$ would generally overshoot — i.e., $\sum_{\vec{y}} p_m(\vec{y}|\mathbf{x}; W) > 1$ then it would also give too high values for noise examples. Conversely, generally undershooting would give too low values to correct translation examples.

Training is faster, since we only need to compute the output node value for the given training and noise examples — there is no need to compute the other values, since we do not normalize with the softmax function.

Given the definition of the training objective L, we have now a complete computation graph that we can implement using standard deep learning toolkits, as we have done before. These toolkits compute the gradients $\frac{dL}{dW}$ for all parameters W and use them for parameter updates via gradient descent training (or its variants).

It may not be immediately obvious why optimizing towards classifying correct against noise examples gives rise to a model that also predicts the correct probabilities for n-grams. But this is a variant of methods that are common in statistical machine translation in the tuning phase. MIRA (margin infused relaxation algorithm) and PRO (pairwise ranked optimization) follow the same principle.

4.4 Recurrent Neural Language Models

The feed-forward neural language model that we described above is able to use longer contexts than traditional statistical back-off models, since it has more flexible means to deal with unknown contexts. Namely, the use of word embeddings to make use of similar words, and the robust handling of unseen words in any context position. Hence, it is possible to condition on much larger contexts than traditional statistical models. In fact, large models, say, 20-gram models, have been reported to be used.

Alternatively, instead of using a fixed context word window, **recurrent neural networks** may condition on context sequences of any length. The trick is to re-use the hidden layer when predicting word w_n as additional input to predict word w_{n-1}.

See Figure 4.4 for an illustration. Initially, the model does not look any different from the feed-forward neural language model that we discussed so far. The inputs to the network is the first word of the sentence w_1 and a second set of neurons which at this point indicate the start of the sentence. The word embedding of w_1 and the start-of-sentence neurons first map into a hidden layer h_1, which is then used to predict the output word w_2.

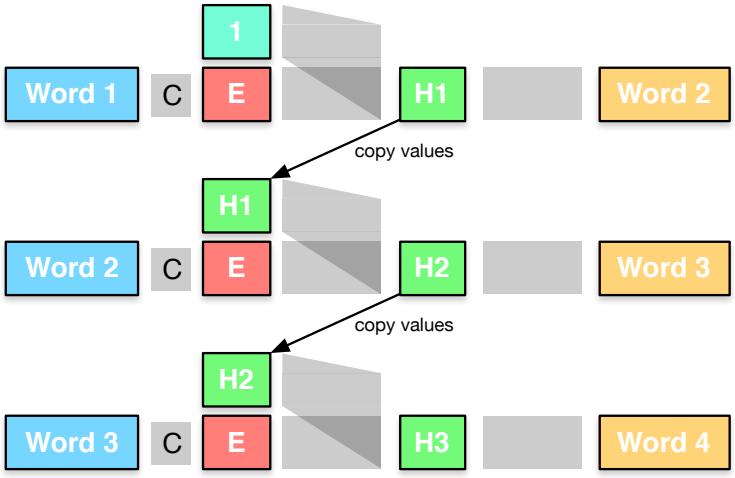

Figure 4.4: Recurrent neural language models: After predicting Word 2 in the context of following Word 1, we re-use this hidden layer (alongside the correct Word 2) to predict Word 3. Again, the hidden layer of this prediction is re-used for the prediction of Word 4.

This model uses the same architecture as before: Words (input and output) are represented with one-hot vectors; word embeddings and the hidden layer use, say, 500 real valued neurons. We use a sigmoid activation function at the hidden layer and the softmax function at the output layer.

Things get interesting when we move to predicting the third word w_3 in the sequence. One input is the directly preceding (and now known) word w_2, as before. However, the neurons in the network that we used to represent start-of-sentence are now filled with values from the hidden layer of the previous prediction of word w_2. In a way, these neurons encode the previous sentence context. They are enriched at each step with information about a new input word and are hence conditioned on the full history of the sentence. So, even the last word of the sentence is conditioned in part on the first word of the sentence. Moreover, the model is simpler: it has less weights than a 3-gram feed-forward neural language model.

How do we train such a model with arbitrarily long contexts?

One idea: At the initial stage (predicting the second word from the first), we have the same architecture and hence the same training procedure as for feed-forward neural networks. We assess the error at the output layer and propagate updates back to the input layer. We could process every training example this way — essentially by treating the hidden layer from the previous training example as fixed input the current example. However, this way, we never provide feedback to the representation of prior history in the hidden layer.

The **back-propagation through time** training procedure (see Figure 4.5) unfolds the recurrent neural network over a fixed number of steps, by going back over, say, 5 word predictions. Note that, despite limiting the unfolding to 5 time steps, the network is still able to learn dependencies over longer distances.

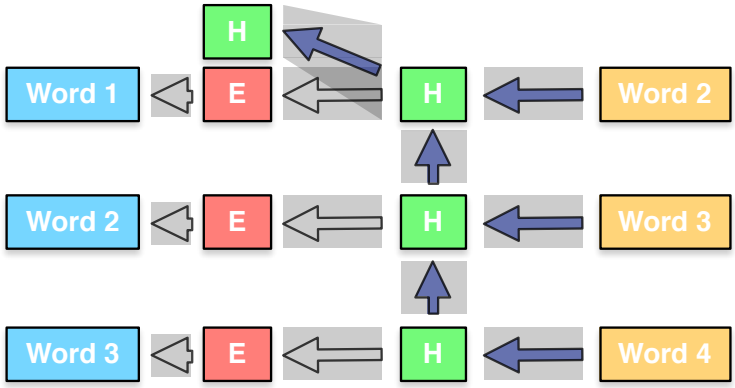

Figure 4.5: Back-propagation through time: By unfolding the recurrent neural network over a fixed number of prediction steps (here: 3), we can derive update formulas based on the training objective of predicting all output words and back-propagation of the error via gradient descent.

Back-propagation through time can be either applied for each training example (here called time step), but this is computationally quite expensive. Each time computations have to be carried out over several steps. Instead, we can compute and apply weight updates in mini-batches (recall Section 2.6.7). First, we process a larger number of training examples (say, 10-20, or the entire sentence), and then update the weights.

Given modern compute power, fully **unfolding the recurrent neural network** has become more common. While recurrent neural networks have in theory arbitrary length, given a specific training example, its size is actually known and fixed, so we can fully construct the computation graph for each given training example, define the error as the sum of word prediction errors, and then carry out back-propagation over the entire sentence. This does require that we can quickly build computation graphs — so-called **dynamic computation graphs** — which is currently supported by some toolkits better than others.

4.5 Long Short-Term Memory Models

Consider the following step during word prediction in a sequential language model:

After much economic progress over the years, the **country** $\rightarrow$ *has*

The directly preceding word *country* will be the most informative for the prediction of the word *has*, all the previous words are much less relevant. In general, the importance of words decays with distance. The hidden state in the recurrent neural network will always be updated with the most recent word, and its memory of older words is likely to diminish over time.

But sometimes, more distant words are much more important, as the following example shows:

The **country** *which has made much economic progress over the years still* $\rightarrow$ *has*

In this example, the inflection of the verb *have* depends on the subject *country* which is separated by a long subordinate clause.

Recurrent neural networks allow modeling of arbitrarily long sequences. Their architecture is very simple. But this simplicity causes a number of problems.

- The hidden layer plays double duty as memory of the network and as continuous space representation used to predict output words.
- While we may sometimes want to pay more attention to the directly previous word, and sometimes pay more attention to the longer context, there is no clear mechanism to control that.
- If we train the model on long sequences, then any update needs to back propagate to the beginning of the sentence. However, propagating through so many steps raises concerns that the impact of recent information at any step drowns out older information.[1]

The rather confusingly named **long short-term memory (LSTM)** neural network architecture addresses these issues. Its design is quite elaborate, although it is not very difficult to use in practice.

A core distinction is that the basic building block of LSTM networks, the so-called **cell**, contains an explicit memory state. The memory state in the cell is motivated by digital memory cells in ordinary computers. Digital memory cells offer operations to read, write, and reset. While a digital memory cell may store just a single bit, a LSTM cell stores a real number.

Furthermore, the read/write/reset operations in a LSTM cell are regulated with a real numbered parameter, which are called **gates** (see Figure 4.6).

- The **input gate** parameter regulates how much new input changes the memory state.
- The **forget gate** parameter regulates how much of the prior memory state is retained (or forgotten).
- The **output gate** parameter regulates how strongly the memory state is passed on to the next layer.

Formally, marking the input, memory, and output values with the time step t, we define the flow of information within a cell as follows.

$$\begin{aligned} \text{memory}^t &= \text{gate}_{\text{input}} \times \text{input}^t + \text{gate}_{\text{forget}} \times \text{memory}^{t-1} \\ \text{output}^t &= \text{gate}_{\text{output}} \times \text{memory}^t \end{aligned} \tag{4.10}$$

[1] Note that there is a corresponding **exploding gradient** problem, where over long distance gradient values become too large. This is typically suppressed by **clipping** gradients, i.e., limiting them to a maximum value set as a hyper parameter.

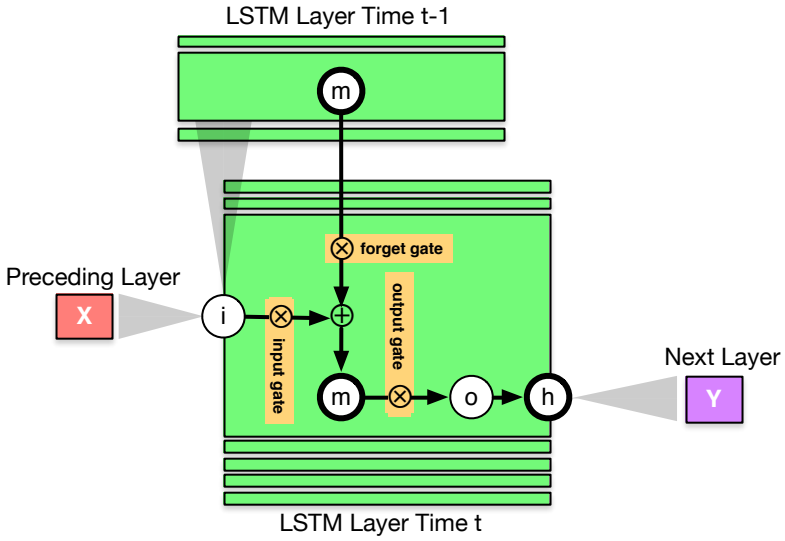

Figure 4.6: A cell in a LSTM neural network. As recurrent neural networks, it receives input from the preceeding layer (x) and the hidden layer values from the previous time step $t-1$. The memory state m is updated from the input state i and the previous time's value of the memory state m^{t-1}. Various gates channel information flow in the cell towards the output value o.

The hidden node value h^t passed on to the next layer is the application of an activation function f to the output value.

$$h^t = f(\text{output}^t) \tag{4.11}$$

An LSTM layer consists of a vector of LSTM cells, just as traditional layers consist of a vector of nodes. The input to LSTM layer is computed in the same way as the input to a recurrent neural network node. Given the node values for the prior layer x^t and the values for the hidden layer from the previous time step h^{t-1}, the input value is the typical combination of matrix multiplication with weights W^x and W^h and an activation function g.

$$\text{input}^t = g\left(W^x x^t + W^h h^{t-1}\right) \tag{4.12}$$

But how are the gate parameters set? They actually play a fairly important role. In particular contexts, we would like to give preference to recent input ($\text{gate}_{\text{input}} \simeq 1$), rather retain past memory ($\text{gate}_{\text{forget}} \simeq 1$), or pay less attention to the cell at the current point in time ($\text{gate}_{\text{output}} \simeq 0$). Hence, this decision has to be informed by a broad view of the context.

How do we compute a value from such a complex conditioning context? Well, we treat it like a node in a neural network. For each gate $a \in (\text{input}, \text{forget}, \text{output})$ we define matrices W^{xa}, W^{ha}, and W^{ma} to compute the gate parameter value by the multiplication of weights and node values in the previous layer x^t, the hidden layer h^{t-1} at the previous time step, and the memory states at the previous time step memory^{t-1}, followed by an activation function h.

$$\text{gate}_a = h\left(W^{xa} x^t + W^{ha} h^{t-1} + W^{ma} \text{memory}^{t-1}\right) \tag{4.13}$$

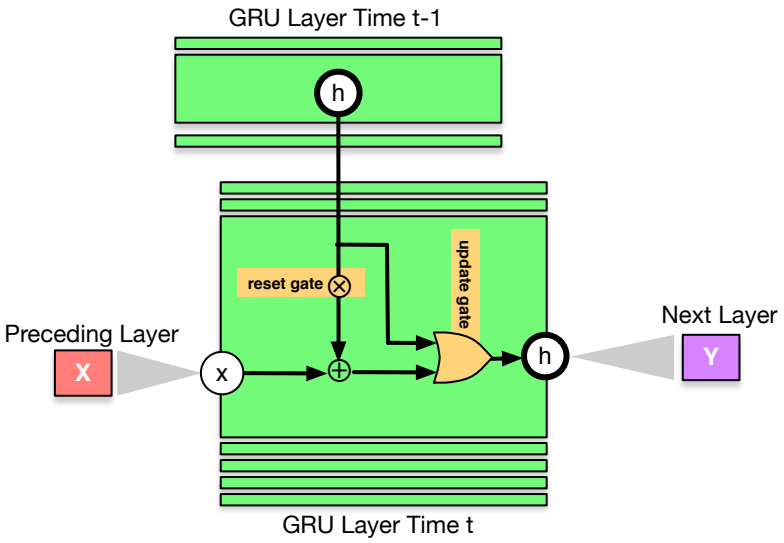

Figure 4.7: Gated Recurrent Unit (GRU): a simplification of long short term memory (LSTM) cells.

LSTM are trained the same way as recurrent neural networks, using back-propagation through time or fully unrolling the network. While the operations within a LSTM cell are more complex than in a recurrent neural network, all the operations are still based on matrix multiplications and differentiable activation functions. Hence, we can compute gradients for the objective function with respect to all parameters of the model and compute update functions.

4.6 Gated Recurrent Units

LSTM cells add a large number of additional parameters. For each gate alone, multiple weight matrices are added. More parameters lead to longer training times and risk overfitting. As a simpler alternative, **gated recurrent units** (GRU) have been proposed and used in neural translation models. At the time of writing, LSTM cells seem to make a comeback in neural machine translation, but both are still commonly used.

See Figure 4.7 for an illustration for GRU cells. There is no separate memory state, just a hidden state that serves both purposes. Also, there are only two gates. These gates are predicted as before from the input and the previous state.

$$\begin{aligned} \text{update}_t &= g(W_{\text{update}}\ \text{input}_t + U_{\text{update}}\ \text{state}_{t-1} + \text{bias}_{\text{update}}) \\ \text{reset}_t &= g(W_{\text{reset}}\ \ \text{input}_t + U_{\text{reset}}\ \ \ \text{state}_{t-1} + \text{bias}_{\text{reset}}) \end{aligned} \tag{4.14}$$

The first gate is used in the combination of the input and previous state. This is combination is identical to traditional recurrent neural network, except that the previous states impact is scaled by the reset gate. Since the gate's value is between 0 and 1, this may give preference to the current input.

$$\text{combination}_t = f(W\ \text{input}_t + U(\text{reset}_t \circ \text{state}_{t-1})) \tag{4.15}$$

Then, the update gate is used for a interpolation of the previous state and the just computed combination. This is done as a weighted sum, where the update gate balances between the two.

$$\begin{aligned}\text{state}_t =&(1 - \text{update}_t) \circ \text{state}_{t-1} + \\ &\text{update}_t \quad \circ \text{combination}_t) + \text{bias}\end{aligned} \tag{4.16}$$

In one extreme case, the update gate is 0, and the previous state is passed through directly. In another extreme case, the update gate is 1, and the new state is mainly determined from the input, with as much impact from the previous state as the reset gate allows.

It may seem a bit redundant to have two operations with a gate each that combine prior state and input. However, these play different roles. The first operation yielding combination_t (Equation 4.15) is a classic recurrent neural network component that allows more complex computations in the combination of input and output. The second operation yielding the new hidden state and the output of the unit (Equation 4.16) allows for bypassing of the input, enabling long-distant memory that simply passes through information and, during back-propagation, passes through the gradient, thus enabling long-distance dependencies.

4.7 Deep Models

The currently fashionable name **deep learning** for the latest wave of neural network research has a real motivation. Large gains have been seen in tasks such as vision and speech recognition due to stacking multiple hidden layers together.

More layers allow for more complex computations, just as having sequences of traditional computation components (Boolean gates) allows for more complex computations such as addition and multiplication of numbers. While this has been generally recognized for a long time, modern hardware finally enabled to train such deep neural networks on real world problems. And we learned from experiments in vision and speech that having a handful, and even dozens of layers does give increasingly better quality.

How does the idea of deep neural networks apply to the sequence prediction tasks common in language? There are several options. Figure 4.8 gives two examples. In shallow neural networks, the input is passed to a single hidden layer, from which the output is predicted. Now, a sequence of hidden layers is used. These hidden layers $h_{t,i}$ may be **deeply stacked**, so that each layer acts like the hidden layer in the shallow recurrent neural network. Its state is conditioned on its value at the previous time step $h_{t-1,i}$ and the value of previous layer in the sequence $h_{t,i-1}$.

$$\begin{aligned} h_{t,1} &= f_1(h_{t-1,1}, x_t) && \text{first layer} \\ h_{t,i} &= f_i(h_{t-1,i}, h_{t,i-1}) && \text{for } i > 1 \\ y_t &= f_{i+1}(h_{t,I}) && \text{prediction from last layer } I \end{aligned} \tag{4.17}$$

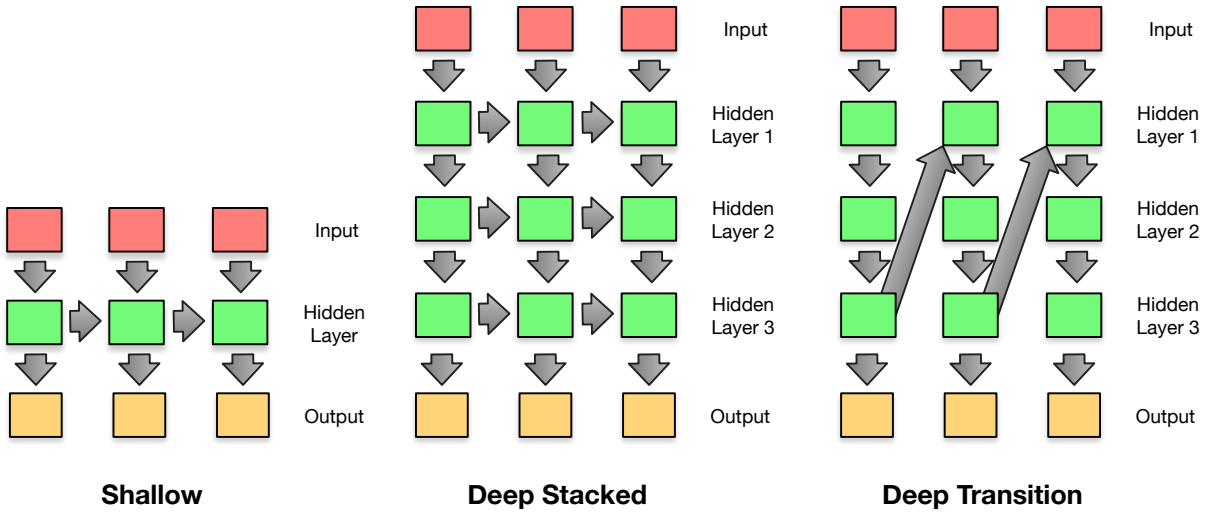

Figure 4.8: Deep recurrent neural networks. The input is passed through a few hidden layers before an output prediction is made. In deep stacked models, the hidden layers are also connected horizontally, i.e., a layer's values at time step t depends on its value at time step $t-1$ as well as the previous layer at time step t. In deep transitional models, the layers at any time step t are sequentially connected and first hidden layer is also informed by the last layer at time step $t-1$.

Or, the hidden layers may be directly connected in **deep transitional** networks, where the first hidden layer $h_{t,1}$ is informed by the last hidden layer at the previous time step $h_{t-1,I}$, but all other hidden layers are not connected to values from previous time steps.

$$\begin{aligned} h_{t,1} &= f_1(h_{t-1,I}, x_t) && \text{first layer} \\ h_{t,i} &= f_i(h_{t,i-1}) && \text{for } i > 1 \\ y_t &= f_{i+1}(h_{t,I}) && \text{prediction from last layer } I \end{aligned} \tag{4.18}$$

In all these equations, the function f_i may be a feedforward layer (matrix multiplication plus activation function), an LSTM cell or a GRU cell.

Experiments with using neural language models in traditional statistical machine translation have shown benefits with 3–4 hidden layers (Luong et al., 2015a).

While modern hardware allows training of deep models, they do stretch computational resources to their practical limit. Not only are there more computations in the neural network, convergence of training is typically slower. Adding skip connections (linking the input directly to the output or the final hidden layer) sometimes speeds up training, but we still talking about a several times longer training times than shallow networks.

Further Readings The first vanguard of neural network research tackled language models. A prominent reference for neural language model is Bengio et al. (2003), who implement an n-gram language model as a feed-forward neural network with the history words as input and the predicted word as output. Schwenk et al. (2006) introduce such language models to machine translation (also called

“continuous space language models"), and use them in re-ranking, similar to the earlier work in speech recognition. Schwenk (2007) propose a number of speed-ups. They made their implementation available as a open source toolkit (Schwenk, 2010), which also supports training on a graphical processing unit (GPU) (Schwenk et al., 2012).

By first clustering words into classes and encoding words as pair of class and word-in-class bits, Baltescu et al. (2014) reduce the computational complexity sufficiently to allow integration of the neural network language model into the decoder. Another way to reduce computational complexity to enable decoder integration is the use of noise contrastive estimation by Vaswani et al. (2013), which roughly self-normalizes the output scores of the model during training, hence removing the need to compute the values for all possible output words. Baltescu and Blunsom (2015) compare the two techniques - class-based word encoding with normalized scores vs. noise-contrastive estimation without normalized scores - and show that the letter gives better performance with much higher speed.

As another way to allow straightforward decoder integration, Wang et al. (2013) convert a continuous space language model for a short list of 8192 words into a traditional n-gram language model in ARPA (SRILM) format. Wang et al. (2014) present a method to merge (or “grow") a continuous space language model with a traditional n-gram language model, to take advantage of both better estimate for the words in the short list and the full coverage from the traditional model.

Finch et al. (2012) use a recurrent neural network language model to rescore n-best lists for a transliteration system. Sundermeyer et al. (2013) compare feed-forward with long short-term neural network language models, a variant of recurrent neural network language models, showing better performance for the latter in a speech recognition re-ranking task. Mikolov (2012) reports significant improvements with reranking n-best lists of machine translation systems with a recurrent neural network language model.

Neural language model are not deep learning models in the sense that they use a lot of hidden layers. Luong et al. (2015a) show that having 3-4 hidden layers improves over having just the typical 1 layer.

Language Models in Neural Machine Translation: Traditional statistical machine translation models have a straightforward mechanism to integrate additional knowledge sources, such as a large out of domain language model. It is harder for end-to-end neural machine translation. Gülçehre et al. (2015) add a language model trained on additional monolingual data to this model, in form of a recurrently neural network that runs in parallel. They compare the use of the language model in re-ranking (or, re-scoring) against deeper integration where a gated unit regulates the relative contribution of the language model and the translation model when predicting a word.